The Essense of Computer System Engineering

Prabhas Chongstitvatana

Chulalongkorn University

To my father I dedicate this book.

Preface

The computer system engineering requires a unified view of hardware and software, from the level of data path to application software. Understanding the relationship throughout all levels is an essential knowledge to deal with complex computer systems today.

Most students in computer science or computer engineering department must face a rapid change in technologies. It is no wonder they usually feel bewildered from menageries of subjects that are driven by the most up-to-date technologies. Teachers also face the same dilemma, if we teach only the fundamentals students will not be able to practice. The tools change too fast. There are too many subjects to teach and there is not enough time to learn. Throughout my years of teaching, there is a swing from giving too much practical projects to not giving any practical projects. Parnas, a famous professor in computer science who invented the term "module", once lamented that we were giving out too many useless projects to our students. I am also a victim of these "fashions". I am afraid that my students will not know the latest software tools. I am also afraid that my students do not know enough fundamentals. The fact is both the number of tools and fundamentals always increase. Both our students and us, teachers, are facing the impossible task.

Computer science is a young science. The first electronic computer was invented in the 1950 era. Computer engineering is not yet a mature engineering discipline. The microelectronics era has just begun in 1965. I do believe that the present state of the art in computing, the technologies that are in used nowadays, have just scratched the surface of possibilities. There are so many wonderful inventions lying in the future. Let us be patient.

What has been taught in computer science and computer engineering today is adequate. Many new subjects find their way into the curriculum. Most old fundamentals have been through revision and condensation. However, helping students to navigate through these subjects is not always a success as it should be. The most striking feeling in my teaching career is that students in the senior year, who have been through the whole curriculum, are unable to "put all the pieces together" to form a coherent view of the field they studied. There are too many subjects that they become unrelated, inconsistent, or sometimes irrelevant.

It is my attempt in this book to put a whole picture to students, to ask them to *build* a whole system by themselves. The essences of computer system engineering are the ability to understand components and compose them into a system. To tackle a project of this size is not possible without critical assessment of the system we intend to build. My selections are: language, compiler, code generator, processor and operating system. All these components are implemented under one language which itself is also a subject to be studied. Each component is carefully designed so that it is as simple as possible and yet is able to demonstrate the principle of its operation.

Building a whole computer system always gives a good insight to understand the current technologies. The ability to ask "what if" questions throughout all levels of a computer system is a great benefit. Students can discover many principles by themselves. They can also gain a large factor of confidence in handling a complex system. They will understand how components interact, how to make tradeoff between various constraints and how to choose among alternatives.

The book is divided into nine chapters. They present a sequence of system building, from the language to processor, and then the operating system is implemented to complete the whole system. The outline of each chapter is as follows.

Chapter 1 An overview of computer system engineering − the chapter introduces a computer system structure, the relationships between hardware and software, the nature of computation, the performance issue and a brief history of computer.

Chapter 2 High level language Nut − the chapter lays down the first tool for building our computer system, the language, Nut; the design principle is discussed; the syntax and semantic of the language are given; the internal forms and the instruction set are studied.

Chapter 3 Nut compiler − the compiler for Nut language is studied; the parser, the symbol table, the run-time support and finally the evaluator.

Chapter 4 Code generation − the N-code to machine code translation is studied; the target machine code, S-code, is elaborated; a three-address instruction set is also illustrated.

Chapter 5 Microprogramming − in order to study a processor, the control unit is implemented in microprogram; the chapter discusses the basic and variations of microprogramming and illustrates a systematic design of microprogram.

Chapter 6 Processor Sx − the main processor, its data path and its instruction set are studied; the execution cycle is explained; the microprogram is developed; the performance on benchmark programs is measured.

Chapter 7 Performance enhancement − to address the issue of performance of a processor, this chapter studied stack frame caching; the data path is modified; the effect of the mechanism is analysed.

Chapter 8 Operating System Nos − the Nut operating system is studied in this chapter; its service includes interprocess communication, message passing and timer; the supervisor that provides an interface to the processor simulator is discussed.

Chapter 9 Optimisation − the last chapter takes a look at the performance improvement at every level of the system; macro expansion, introduction of new primitives, improve code generation, and new instructions.

The prerequisite is basic knowledge of digital design, computer architecture, and some knowledge in computer language and compiler. No extensive knowledge of operating systems is required. Some programming skill is required as students will run and modify various given simulators.

The book is suitable for senior undergraduate students and the first year graduate students. I do not cover the basic of all topics discussed in this book. I assume that they can be learnt from other textbooks or students already acquired them before tackling the project in this book. However, this book also serves many practitioners who want to reinforce or refresh some of their skills. I have used various chapters of this book to teach graduate students for a number of years. Graduate students usually have diverse background; teachers must supplement the necessary materials. I also offer this course for senior undergraduate students who have a uniform background. The material must be condensed to fit the available learning load.

We will be using one high level language to describe and to model an executable system in all levels of details throughout. This will force us to be consistent and to be complete in the sense that every concept can be implemented and measured in a system. We have three kinds of languages used in this text. The first one is a normal English text. It will be printed in Times-Roman font. The second one is the pseudo code, used to describe the algorithm or the specification. It is printed in *Italics*. The third language is the implementation language, it is the executable code, and is printed in Arial font.

For example, a quicksort algorithm can be described as follows.

An English prose

> A quicksort algorithm does the sorting in place using the partition function to divided elements into two sub-arrays according to the pivot. Then it is applied recursively to sort both sub-arrays.
>
> The input is an array `ax[p..r]`. `q` is the pivot, it divides `ax` into two sub-arrays `ax[p..q]` and `ax[q+1..r]`.

An algorithm

```
quicksort(ax,p,r)
  if(p < r)
    q = partition(ax, p, r)
    quicksort(ax, p, q)
    quicksort(ax, q+1, r)
```

A concrete program

```
(def quicksort (ax p r) (q)
   (if (< p r)
      (do
      (set q (partition ax p r))
      (quicksort ax p q)
      (quicksort ax (+ q 1) r))))
```

All programs discussed in this text will be presented as many individual functions that will be composed together, similar to constructing a shape from *building blocks*.

This is not a read-only text. It is a *construction kit* for a whole computer system. Students will be given a number of pieces of software (a set of functions) that can be put together to perform a task. Some piece will be left out so that students have to write it by themselves. However, with the given pieces, the system can be built to demonstrate its working condition and will process the input and will give the correct output. Students are asked to extend this working prototype, to implement some piece in a different way. Students can measure all the effect

from their changes. They can decide after analysing the execution profile, whether any change is good or whether the change is cost effective.

Because of the highly vertical integration of this system, the effect of any level can be observed, from the top level of applications down to the lowest level of data path and machine cycle. This can bring insight and appreciation of seeing a computer system *as a whole,* completely transparent.

The appendices provide all the code referred to in the text. However, the initial tools such as an executable Nut-compiler and the most up-to-date version of program (including updates and bug-fixes) are available from the website supporting this course. They should be used to build the system.

http://www.cp.eng.chula.ac.th/faculty/pjw/ecse/

Acknowledgements

First and foremost I thank my parents who gave me spirit, knowledge and compassion. I thank all my teachers; Susak Thongthammachart is my mentor, Yuen Pooworawan is my true advisor, Boonklee Plunksiri taught me the wonderful world of switching and finite state automata, Paisan Saguanmoo gave me the first glimpse of computer architecture, Robin Popplestone is my research inspiration, Tim Smithers carefully shaped my idea, Chris Malcolm carried me through my doctoral program, and all teachers who taught me. My existence in the past fifteen years owed so much to my students as much as to my colleagues. Somchai Prasitjutrakul is my best friend from the first question he asked me "why do you want to be a teacher?". I think now I can answer that question with confidence. I thank all my colleagues in my department and in other universities. I thank all my students, without them, there would not be this book. They are my source of motivation. Several times indeed, they suffered from my effort in teaching! However, looking back, I think all my students taught me so well how to be a good teacher. I thank the department of computer engineering, Chulalongkorn university that provides me with the best environment to teach and to be taught. It is also the place where I spent most of my time wondering about the meaning of education. Lastly, I thank my family, my wife who did all the figures in this book, and my two daughters who suffered negligence during my time focusing on this book. They gave me the energy!

Prabhas Chongstitvatana

Contents

Chapter 4 Code Generation 85

Chapter 5 Microprogramming 115

Chapter 8 Nut Operating System 183

Chapter 9 Optimisation 213

Chapter 1

Computer System Engineering

This chapter covers basic knowledge of the subject. An overview and the perspective of computer system engineering are given. The components and the organisation of computer systems in many levels of abstractions are discussed. The relationship between architecture and computer languages is important and several issues have been addressed. One important aspect of modern computer systems, the performance issue, is discussed. Finally, a brief history of computer is narrated. Computer history itself is a very fascinating subject.

1.1 Introduction

A computer system consists of many parts. A part can be divided into subparts and forms a hierarchy. Computer system engineering concerns how to compose these parts to provide a system that has desired functions under various constraints. A computer system has a central processing unit (CPU), memory, input/output, interconnections. A CPU consists of an arithmetic logic unit (ALU), data path, and a control unit. The memory subsystem consists of hierarchical structure: cache memory (high speed memory), main memory, virtual memory. The input/output system consists of various peripherals such as a visual display unit, a keyboard, input devices, an interface to the network, various kinds of secondary storage, bulk memory, a hard disk etc. The interconnections link every parts together, the internal bus, the external bus, I/O channels, ports.

There are many possibilities of choosing and integrating various *components* of a system to satisfy a set of constraints stated in a requirement. A computer system engineer must make decision how to select and integrate various components such as processors, memory, input/output into a computer system. A computer system is driven by the advancement of technology. Various parts of a computer system can be either hardware or software. Hardware and software are interchangeable depending on technology.

Computation

A computer system performs computation. What is computation? Computation can be defined as *symbols transformation*. It is a process that transforms input symbols to output symbols. *Symbol* is an abstraction. A symbol can represent something in the real world, or it can represent some mathematical object. The real world is connected to a computation by sensors and actuators. A sensor transforms real world events, such as temperature, into symbols that are fed into computation. An actuator transforms symbols from a computation to affect the real world. An actuator such as a motor has effect in the real world. It may change the state of the world. The relationship between computation and the real world can be shown as the figure 1.1.

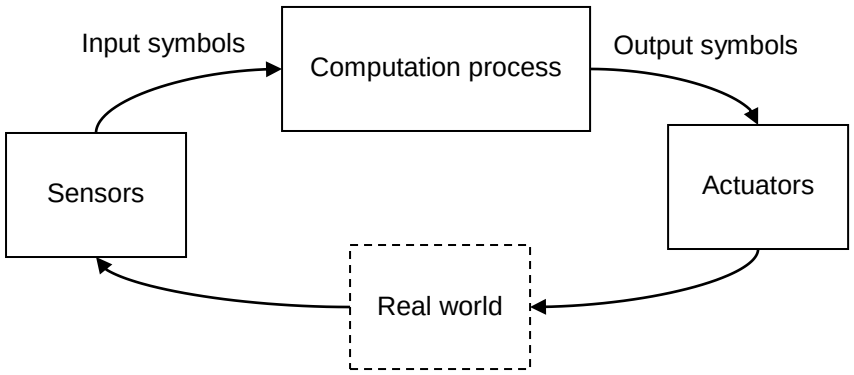


Figure 1.1 Relationship between computation and real world

Software is a specification of a computation. From this point of view, a software does not describe sensors, actuators nor the events in the real world. Hence, it is necessarily incomplete, i.e. it cannot describe the computation plus the real world connected to that computation completely.

Hardware and Software

The most important property of computer systems is it *programmability*. This property differentiates a computer from all other artifacts. Software is the result of this property. Software as a specification of computation enables a computer

to be multi-function, and even adaptive. An application software runs on a computer system. At the bottom level of a computer there are electronic circuits which are called *hardware*. The interface between a program and a hardware is the instruction set. An instruction set defines an abstraction of hardware. This abstraction allows a programmer to *program* a hardware to perform multiple functions.

Components of a computer

There are many possibilities in realizing a programmable system. The most influencial concept is the *stored program* concept invented by John Von Neumann. In this model, a computer is composed of two parts: processor and memory. Memory stores both data and program. Furthurmore, memory can be accessed directly at any location. This is called *random access model*. Other possible realization of a programmable system includes data flow architecture, systolic architecture etc. We will restrict our study to the stored program concept.

A processor is connected to memory through two ports: address and data. The *access* to memory by a processor is done by sending an address to a memory device then a value can be read or write through the data port. The size of value (measured in the number of bits) that can be accessed is the width of the data. This size defines the bit-size of a processor, such as 8-bit, 16-bit, 32-bit, 64-bit processor.

A processor contains an arithmetic-logic unit (ALU), registers, a program counter (PC), an instruction register (IR) and a countrol unit. An ALU performs arithmetic and logic functions: add, substract, multiply, divide, and, or, not and others. Registers are the fast memory used by a processor to store the intermediate results. A program counter keeps track where the current instruction is. It is changed by instructions that alter the flow of control of a program (if-then-else, loop, and function call in a high level language). An instruction register stores the current instruction fetched from memory. Its content (the instruction) signals the control unit to initiate the execution of that instruction. The control unit sends control signals to all parts in the processor to co-ordinate their activities. The control unit is a large finite state machine. It is the most complex part of a processor.

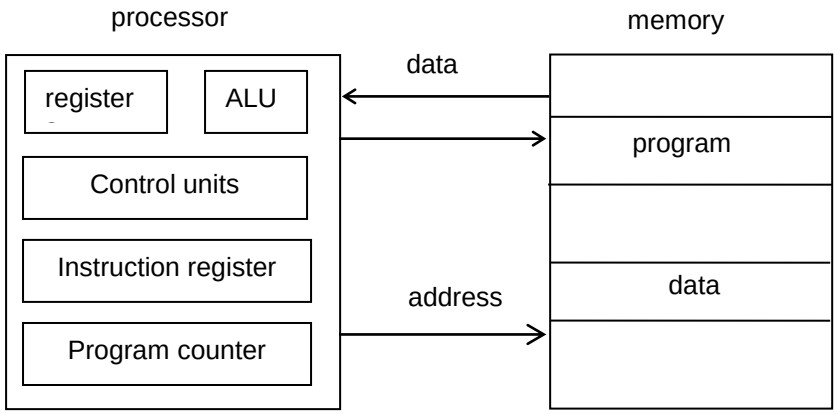


Figure 1.2 Components of a computer

1.2 Computer system structure

A computer system can be seen as many level of descriptions, from the applications to the lowest level of electronic circuits. A computer system consists of many parts of which can be regarded as *layers*. These layers are described at different *level of abstraction*. There are many ways to define the level of abstractions. For example, a computer system at the bottom level consists of the actual hardware devices: a central processing unit, a memory, input/output devices and interconnections. These hardware devices can be described at the level of: functional units, finite state machines, logic gates down to the electronic circuits. On top of hardware of the system, an operating system gives services to application programs. The interface between programs and hardware is the instruction set description. A computer system can also be viewed as having two aspects: physical and logical. The *physical* system is composed of the actual physical components. The *logical* system describes the design and the organization.

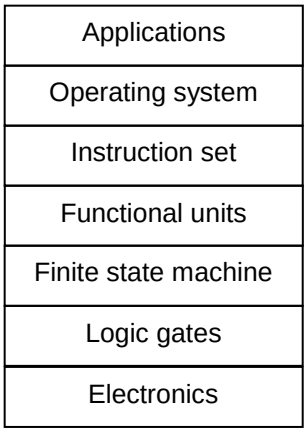


Figure 1.3 The level of description of computer systems

Application level is what a user typically sees a computer system, running his/her application programs. An application is usually written in a computer language which used many system functions provided by the operating system. An *operating system* is abstraction layers that separate a user program from the underlying system-dependent hardware and peripherals.

The level of traditional computer architecture begins at the *instruction set*. An instruction set is what a programmer at the lowest level sees of a processor (programming in an assembly language). In the past, instruction set design is at the very heart of computer design. The concept of the family of computer was promoted by IBM around 1970. They proposed the concept of one instruction set with different level of performance for many models. This concept is possible because of the research effort of IBM in using "microprogram" as the method to implement a control unit. However as the present day processor designs converge, their instruction sets become more similar than different. The effort of the designer had turned to other important issues in computer design.

Finite state machine description is a mathematical description of the *behaviour* of a system. It is becoming an important tool for verification of the correct behaviour of the hardware during designing of a processor. As a processor becomes more and more complex, a mathematical tool is required in order to guarantee the correct working behaviour since an exhaustive testing is impossible and partial testing is expensive (but still indispensable). Presently it is estimated

that more than half of the cost in developing a processor is spent on verifying that the design works according to its specification.

The lower level of *logic gates* and *electronics* describe the logical and actual circuits of a computer system and belongs to the realm of an electrical engineer.

This level of abstraction enables separate layers to be designed and implemented independently. It also provides a high degree of tolerance to changes. A change in one layer has limited effect on other layers. This degree of *decoupling* is important as a computer system is highly changeable and technology-dependent. The changes are very frequent; a new microelectronic fabrication process leads to a higher speed device, a new version of operating system provides more functionality, new applications are created. Without separation into layers all these changes will interact in a complex and uncontrollable way. The level of abstraction is a key concept in designing and implementing a complex system.

1.3 Computer hardware

The basic elements are logic gates. A complete set of gates composed of: AND, OR, NOT gates. (This is not the only basis, there are several others). NAND gate (NOR gate) is complete because it can performed the same function as AND, OR, NOT gates. Logic gates are used to build larger *functional units* which are the building blocks of a computer. There are two types of logic gates, one with memory and one without.

A *combinational logic* has no memory; its output is the function of its input only. To create memory, the output is fed back to the input. The resulting circuit is called a sequential logic.

A *sequential logic* is the logic gate with memory. The basic element is called flip-flop. There are many types of flip-flop such as RS, JK, T and D-type flip-flop. A sequential logic has “states”. The output depends on both inputs and states. There are two types, synchronous and asynchronous. A *synchronous logic* has a common clock. It is a rule of thumb for design engineers to choose a synchronous logic because it is much simpler to design and to debug. One draw back of synchronous design is that the maximum speed of the clock is determined by the slowest part of the circuit. Therefore it is a worst-case design. An *asynchronous logic* has no central clock, hence it can be much faster than synchronous design when the clock rate is very high and clock skew becomes a problem. The output of one stage is used to drive the next stage. It is difficult to

arrange the timing for the circuit to operate properly as the delay of each element affects the timing of the whole circuit. There are large variation of delay when fabricating each logic element. This fact often makes asynchronous design impractical or very expensive.

An example of asynchronous design illustrates the point above. The super computer ILLIAC from the university of Illinois at Urbana-Champaign has asynchronous design to achieve high clock rate [BEL71]. Each connecting wire has to be trimmed manually to properly adjusted the delay time of each module. In the era of VLSI, most design is synchronous because it is much easier to get the design to work properly. Presently due to the advancement of asynchronous design methodology and the promise of very high speed result (and low power consumption) the asynchronous design is coming back. It is an active area of research. There are many standard textbooks on digital logic design which students can explore the subject in much more details such as the one by Katz [KAT93].

In order for a computer to execute a program, many functional units are necessary. Functional units are the building blocks of processors. These building blocks plus the control unit constitute the basic structure of a processor. Basic units to perform arithmetic functions are: adder, multiplier, shifter etc. A functional unit may be built on smaller units, for example, in an adder, a Half adder is built out of basic gates and two Half adders combined into a Full adder. The length of operand affects the speed of adder circuits. The delay comes from the need to propagate the carry bits. *Carry-look-ahead* logic, invented by Charles Babbage [LEE95] who was considered the father of modern computer, is used to speed up the propagation of the carry bits.

Instruction execution cycle

Instructions reside in memory. This is why this architecture is called *stored program*. Instructions can be accessed from a processor similar to any piece of data in memory. A sequence of instructions is a program. A processor starts running a program by reading instructions from memory and executing them one instruction at a time.

The cycle starts by a processor sending the address of the current instruction to memory via the address bus. The current instruction is read from the memory via the data bus and is stored in the instruction register (IR). IR causes the control unit to co-ordinate activities in the processor to execute that instruction. The

processor then starts to read the next instruction (the program counter is increment to point to the next instruction) and executing it and so on.

The result of executing of an instruction can effect many parts: registers, data in the memory, or the program counter. When an instruction changes the program counter, it causes the program to change the flow, either the program entering the loop or selecting the next statement depending on the conditional statement.

Hardware level

A processor consists of a data path and a control unit. A data path contains all the necessary computing elements to carry out a computation task. The control unit sends control signals to harmonise the data flow in the data path so that the desired computation occurs. To give an analogy of a processor to an orchestra, the data path is the musician, the control unit is the conductor.

Components in a data path consist of: logic, register, multiplexer and bus.

- *Logic* is a combinational function, $out = f(in_1 \ldots in_n)$ where f is a Boolean function {not, and, or}. For example

$$out = \bar{x}_1 + x_2 \cdot x_3 + x_1 \cdot \bar{x}_2$$

  Where $\bar{}$ denotes not, + denotes or, · denotes and functions. A Boolean expression can be represented as a truth table. An enumeration of all cases of input values to output values. Logic minimisation is a process to realise a desired function with minimum number of logic elements (such as gates). Logic minimisation is an NP-hard problem.

- *Registers* are storage elements, out(t+1) = in(t), with the control signal "load" (the change can be either on the positive or the negative edge of the clock depends on the model). The width of a register defines the number of bits that can be stored.

- *Muliplexors* have *n* inputs (of width *m*) and select one input to be the output, called *n:1* multiplexer. The control signal to determine the output is called the select signal.

- *Bus* consists of wires and buffers. Wires carry data (signal). Buffer controls the traffic of data from any element to a bus. A bus can be shared to reduce

the number of wire within a circuit. A bus can broadcast data to many receivers limited by the *fan-out* electrical characteristic of the bus, the ability to drive other circuits.

With these four elements: logic, register, multiplexor and bus, a processor can be built.

Data path

The simplest data path consists of a *loop* from registers to functional units (logic) and back.

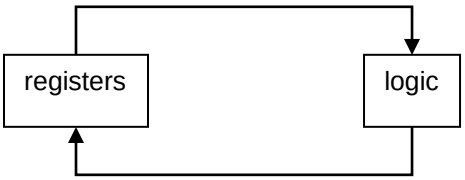


Figure 1.4 A simple data path

For example, suppose there are two registers, named A and B, and an adder. This data path can perform

$$A = A + B$$

with the following control,

1. read two registers into two inputs of the adder.
2. the adder outputs the result of adding its two inputs.
3. the result is written back to a register.

There can be multiple function units working in parallel. The result is more work done in one cycle round the loop. There are complexities involving in doing many tasks concurrently such as competing for the same resource.

1.4 How a processor performs computation

Suppose we want to calculate value of a polynomial

$$f(x) = a\,x + b\,x^2$$

The functional units required to do this computation are multipliers and adders. The desired computation can be performed by directly connecting an appropriate number of functional units together (Fig 1.5).

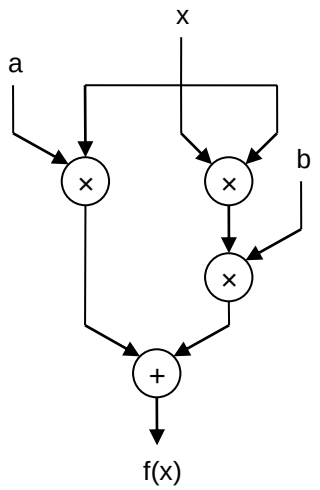


Figure 1.5 A computation graph to evaluate a polynomial

The solution of this computation problem becomes a graph whose nodes are functional units and arcs are connections of data through these units. The computation is performed by the flow of data. In this model every units can be active concurrently. "Programming" in this model becomes specifying the computation graph.

Another way to compute $f(x)$ is by sequentialise the operations (Fig 1.6). The required functional units are memory and a general processing unit. A memory stored all the necessary values: input x, constant a, b, temporary places to keep intermediate values $t1$, $t2$, and the final result $f(x)$. The memory can be read and written to. Two values can be read from memory at once and the data is fed to a general processing unit, so called Arithmetic Logic Unit (ALU).

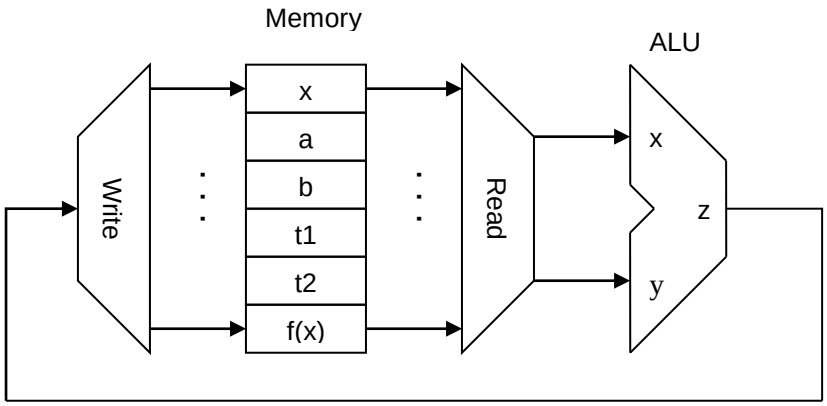


Figure 1.6 A sequential model of computation

The processing unit can perform multiplication and addition. It has internal storage to store two input values and one output value. In general, ALU can do a number of computations. Assume its inputs are *x, y*, output *z*, ALU performs $z = f(x, y)$ where $f = \{ add, sub, mul, increment, ... \}$. The output of the processing unit (z) is connected to the write port of the memory. Now the desired computation can be performed by executing these steps:

```
read(x,a)
alu(mul)
write(t1)
read(x,x)
alu(mul)
write(t2)
read(t2,b)
alu(mul)
write(t2)
read(t1,t2)
alu(add)
write(result)
```

Sequential approach to computation enables functional units to be reused as the computation is performed step-by-step. The intermediate values can be saved in the memory and they can be used in the later steps. The general processing unit can perform a number of different functions such as add, subtract, so that only one unit is sufficient for most kinds of computation. The trade-off is the speed as

the computation becomes sequential there is no opportunity for concurrent operations as in the graph model. Sequential machines are highly flexible, use less resource to implement a computation but are slower than the graph machines. However both graph model and sequential model are similar in the sense that the computation is carried out by directing the flow of data through functional units.

The step-by-step instructions of computation in sequential machines become "program". Burks, Goldstein and Von Neumann [BUR46] are the first to propose that programs can reside in the same memory as data. This gives rise to a class of architecture called *Stored program computer* (Fig 1.7).

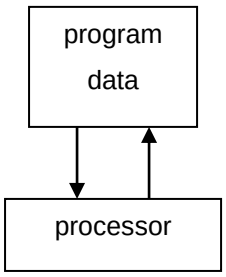


Figure 1.7 Von Neumann (or Princeton) architecture

This is the most popular organisation even today. Storing programs and data in the same memory enables a processor to be able to manipulate programs easily. The main disadvantage is the limit of memory bandwidth, which affects the speed of running an application. As the need for more complex applications which required large amount of computation increases, having only one connection between a processor and a memory becomes bottleneck. This phenomenon is called *Von Neumann bottleneck*.

Other organisation is possible such as storing programs and data in separate memories (Fig. 1.8).

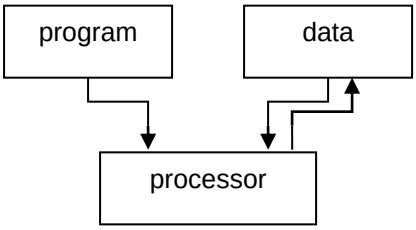


Figure 1.8 Harvard architecture

This organisation is called *Harvard architecture* and is extensively used in the high-speed processor for the purpose for signal processing. This class of processors is called Digital Signal Processor (DSP). DSP has many applications. It is used in modems, in sound synthesizer, in graphic generators etc.

1.5 Computer languages and architecture

Programming techniques influence the design of computers since the early days of assembly language programming. Most computers today are implemented as sequential machines. They are suitable to be programmed in a class of high level programming language, procedural languages. The examples of procedural languages are C, Pascal, C++, Java. In these languages, the computation is viewed as step-by-step manipulation of values of variables stored in memory.

There are other paradigms of programming. Backus, the father of FORTRAN, gave a lecture in the occasion of his reception of Turing award, titled "Can computers be liberated from Von Neumann bottleneck?" [BAC78]. This lecture advocated a different programming paradigm called Functional Programming. In functional paradigm, programming is viewed as the activity of composing functions. The computation of a function has an important property of *referencial transparency*. This means the result of computing a function depends only on its arguments and does not change by where the function resides. This property is contrasted to procedural programming which computes by *side effect*, i.e. manipulation of variables depends on states. Functional programming helps to promote the correctness of programs. As this paradigm of programming views computation as composing functions, it maps nicely to the graph model of computation. Many proposals being put forward to build machines those are suitable for this class of programming languages, for example a graph reduction machine [KOO90].

Different programming paradigms lead to different architectures. Logic programming paradigm (Prolog programming language and others) requires architecture capable of inferring facts and rules and ability to backtrack efficiently, for example [WAR83]. A LISP machine has special instructions to manipulate the type-tag bits [STE88]. Japanese proposed and built various types of these machines in the period of their research on Fifth generation computer. Presently, object-orientated programming paradigm is becoming the dominate paradigm. The object-oriented programming languages (Java, C++, Smalltalk etc.) will benefit from machines whose architecture are suitable to implement them.

1.6 Performance

This section discusses the performance issue. How performance of a computer system is defined and measured. There are many standard references used to interpret performance figures. Performance can be used in a relative sense, it is the measurement of one system compares to another system.

The first commercial electronic computer appeared around 1950. In the first 25 years the performance improvement came mostly from technology and better computer architecture. Later, the improvement mostly came from the advent of microelectronics. The speed of components increased 18-35% per year. Technology progresses from vacuum tubes to transistors to integrated circuits. The birth of microprocessor around 1970 [FAG96] has great impact on performance of computers. The growth of performance has been highest for microprocessors. Since 1980 the performance double every two years. For example, around 1980 the first IBM PC appeared. Its CPU was an Intel 8088, a 16-bit CPU with 8 MHz clock. It had 16K bytes of memory, one floppy disk and no hard disk. The later model offered 5M bytes hard disk (so called IBM XT). Today, a PC is equipped with 32/64-bit CPU with 3 GHz clock, 1G bytes of memory and 100 G bytes disk. Its performance is around 10,000 times of the first PC.

Performance is measured by running *mixed jobs*. Therefore it is not an absolute figure. It depends on the kind of jobs that are used to measure the performance. One phenomenon that occurs in the computer technology is that the performance of a processor has been double every 18 months. This observation is proposed by Moore [SCH97], who is the pioneer (among a number of other engineers) of integrated circuit fabrication. He was with Fairchild, one of the earliest IC

manufacturer. That observation is known as *Moore's law*. The main reason that makes this law possible is the rapid advance of the IC manufacturing technique, the shrinking of the physical dimension of the electronic circuits. For the last 30 years semiconductor technology has been roughly quadrupling every three years. This gives an exponential base of about 1.59 instead of the base 2 proposed in Moore's original paper. A more accurate formula for Moore's law is:

$$N_{\text{device on chip}} = 1.59^{(\text{year} - 1959)}$$

We define performance as:

Performance = how fast a processor complete its job.

Performance is measured by its execution time of a suite of programs called *benchmark programs*. The execution time depends on three factors.

execution time = number of instruction used × cycle per instruction × cycle time

These factors depend on various designs:

- *number of instruction* depends on instruction set design
- *cycle per instruction* depends on micro architecture
- *cycle time* depends on technology

The performance can also be measured by response time and throughput. The response time is the time between the starting of a user job and the time when the computer replies. Under multiple jobs, a better measurement is the throughput. Throughput measures how many jobs can be completed in a unit time. The response time is called *latency* of a system. The throughput is also called the *bandwidth* of a system.

Performance = how fast a computer can run
performance = response time (latency)
performance = throughput (bandwidth)

Relative performance

To compare the performance of two machines, it is natural to state "X is n% faster than Y". The ratio of the execution time is used to state how much one

machine is faster than the other machine. The performance is the inverse of the execution time. The following relationships can be derived.

X is n% faster than Y

$$\text{execution time Y / execution time X} = 1 + n/100$$
$$\text{performance} = 1/\text{ execution time} \quad (\text{or } 1/t)$$
$$\text{execution time Y / execution time X} = \text{performance X / performance Y}$$
$$n = (\text{performance X} - \text{performance Y}) / \text{performance Y}$$

Amdalh's law

The performance improvement can be measured in term of "speedup". With the advent of speed enhancement design such as pipeline and parallelism, Amdalh's law [AMD67] states how much performance improvement can be achieved for a given task using the enhancement. The speedup is defined as follows.

$$\text{speedup} = P_e / P$$
$$\text{speedup} = T / T_e$$

Where P_e is performance with enhancement use, P is performance without enhancement use, T_e is execution time with enhancement use, T is execution time without enhancement use.

If enhancement is used only partially, the speedup will be severely limited. Let f be the fraction that enhancement is used.

$$\text{execution time new} = \text{execution time old} \left((1 - f) + f / \text{speedup} \right)$$

$$\text{speedup overall} = 1 / ((1 - f) + f / \text{speedup})$$

Therefore the limit depends on how much the enhancement has been used. In achieving speedup by parallelization, Amdalh's law predicts that speedup will be limited by the sequential part of the program. Let see some numerical example.

Example: A computer has an enhancement with 10 times speedup. That enhancement is used only 50% of the time. What is the overall speedup?

$$\text{speedup overall} = 1/ ((1 - 0.5) + 0.5/10) = 1.82$$

Please note that Amdalh's law applies only with the problem of fixed size. When the problem size much larger than the machine, Amdalh's law does not applied. This is why the massively parallel machine is still possible.

1.7 Brief history of computer

The history of computer is full of interesting episodes. We will to start off with asking the question "Who made the first computer?" To find out the answer we need to clarify some definition. What kind of machine is considered to be a computer?

In mechanical era, the computing machine is really a mechanical calculator. In 1890, Charles Babbage designed and attempted to build Analytical Engine, which contained many ideas that are used in modern computers such as Arithmetic Logic Unit. However, it was never finished as the British government finally stopped funding for the construction of Babbage's Analytical Engine.

The MARK 1 (also known as the IBM automatic sequence controlled calculator) developed in 1944 at Harvard University by Howard Aiken with the assistance of Grace Hopper. It was used, by the US Navy, for gunnery and ballistic calculations, and kept in operation until 1959. The computer was controlled by pre-punched paper tape and could carry out addition, subtraction, multiplication, division and reference to previous results. Numbers were stored and counted mechanically using 3000 decimal storage wheels. It was electro-mechanical computer and was slow requiring 3-5 seconds for a multiplication operation. This machine is a *configurable calculator*, in an essence it is an implementation of Babbage's machine with newer technology.

When does a machine become a computer? We will define a modern computer as *a general purpose programmable machine*. The "programmability" is considered an essential characteristic of a computer. Alan Turing was the genius who proved that the general purpose computer was possible and simple in 1937 in his seminal paper "On computable numbers" [TUR37]. To have this programmability a computer must have the *stored program*.

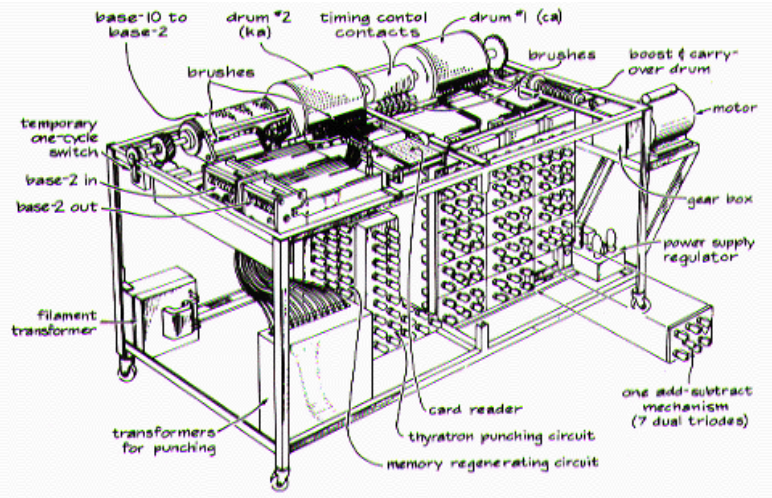


Figure 1.9 The ABC diagram [IOW99]

The ABC (Atanasoff Berry Computer) was built in 1937-1942 at Iowa State University by John V. Atanasoff and Clifford Berry [BUR88] [MOL88]. It introduced the ideas of binary arithmetic, regenerative memory, and logic circuits. This machine was essentially a powerful configurable calculator. Mauchly spent many days with Atanasoff in 1940 studying this machine. This was the first computer to use electronic valves (tubes) to perform arithmetic. Atanasoff stopped developing this with the advent of war, and never returned to it. This machine doesn't have the "stored program" ability.

In 1943 Flowers in Bletchley Park built the first Colossus machine, a programmable computer specially designed to crack the German Enigma military cypher machines. It is not a *general purpose* and has no *stored program*. In 1944 Zuse in Germany started work on a truly general purpose programmable computer of modern type, known as the Z4. The end of the war interrupted development. Zuse's earlier machines (Z1-Z3) were elegant and sophisticated in design, for example using the much more economical binary representation of numbers, but were basically modernised Babbage machines.

A group of scientists and engineers at the University of Pennsylvania, Moore School of Electrical Engineering built ENIAC (Electronic Numerical Integrator and Computer) in 1946 [BUR81]. It was programmed by a plug board, which wired up the different calculation units in the right configuration, to evaluate a particular polynomial. Eckert and Mauchly, the designers, at this time patented a

digital computing device, and are often claimed to be the inventors of the first computer. It was later proven in a 1973 US court battle between Honeywell and Sperry Rand that while spending five days at Atanastoff's lab, Mauchly observed the ABC and read its 35-page manual. Later it was proven that Mauchly had used this information in constructing the ENIAC. Therefore, John Vincent Atanasoff is now (by some US historians) heralded as the inventor of the first electronic computer.

In 1945 John Von Neumann published the EDVAC report, a review of the design of the ENIAC, and a proposal for the design of EDVAC. This is widely regarded as the origin of the idea of the modern computer, containing the crucial idea of the *stored program*. A processor fetches instructions from memory. It also reads and writes data to and from memory. This is called *Von Neumann architecture* where data and instruction co-resides in a memory. This idea came from the proposal of an electronic computer by US Army Ordnance in 1946. Surprisingly, Von Neumann himself is not the first author of that proposal [BUR46]. However, Von Neumann name is honored because of his contribution to the development of this type of computer which has now becomes ubiquitous. The implementation of this design was completed in 1952.

In 1946 The National Physical Laboratory appointed Turing, who had been developing ideas of implementing his Turing Machine concept of general purpose computation in electronic form, to a rival British project intended to outclass EDVAC, known as the ACE. ACE design was at the time the most advanced and most detailed computer design in existence. Its construction was completed in 1950 and named the Pilot ACE.

On 21st June 1948 the first stored program ran on the Small-Scale Experimental Machine (SSEM), nicknamed "Baby", the precursor of the Manchester Mk 1 [LAV80]. So Manchester machine was the first to work.

Figure 1.10 SSEM Baby from Manchester University archive [MAN]

The first program was written by Tom Kilburn. It was a program to find the highest proper factor of any number *a*. This was done by trying every integer *b* from *a* − *1* downward until one was found that divided exactly into *a*. The necessary divisions were done not by long division but by repeated subtraction of *b* (because the "Baby" only had a hardware subtractor).

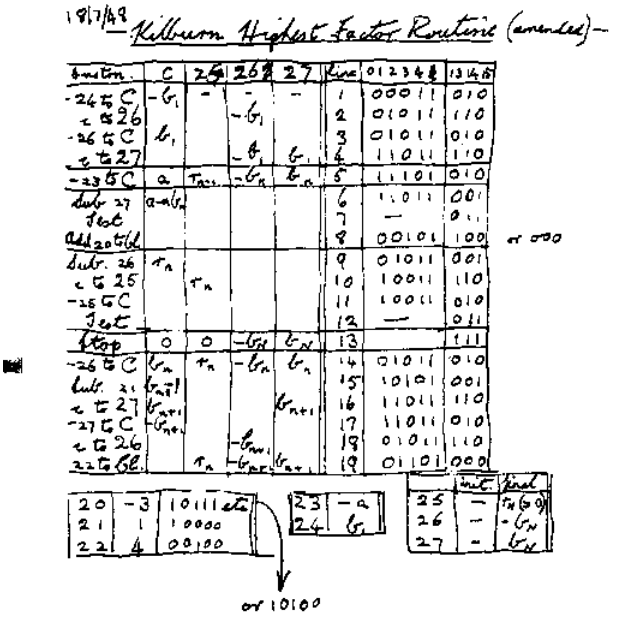


Figure 1.11 The first program [MAN98]

Trying the program on 2^{18}; here around 130,000 numbers were tested, which took about 2.1 million instructions and involved 3.5 million store accesses. The correct answer was obtained in a 52-minute run.

By April 1949 the Manchester Mark 1 had been finished and was generally available for scientific computation in the University. With the integration of a high speed magnetic drum by the autumn; this was the first machine with a fast electronic and magnetic two-level store (i.e. the capability for virtual memory).

In 1951 the UNIVAC 1 commercial computer was produced in US, based on the EDVAC design, and made by Eckert and Mauchly, who by this time had sold their UNIVAC Company to Remington Rand. It employed decimal arithmetic.

We will stop our trip to the history of computer here. To find out more, there is a wonderful journal devoted to all aspects of history of computing, "Annals of the History of Computing", IEEE Computer Society.

1.8 Summary

We have outlined the whole spectrum of a computer system. A computer system can be understood as layers of abstraction. Each layer has well defined characteristic. A computer system engineering requires understanding each component and how many components interact in a computer system. The most important character of a computer system is its programmability. We have focused on computation and its relation to the hardware level, the data path. Computation can be realised as parallel or sequential in a data path. There is interchangeability between hardware and software. This fact gives rise to many choices in the design of a computer system.

References

[AMD67] Amdahl, G., "Validity of the single processor approach to achieving large scale computing capabilities", AFIPS Conf. Proc., April 1967, pp. 483-485.

[BAC78] Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, August 1978, 20(8):613-641.

[BEL71] Bell, C., and Newell, A. Computer structure: Readings and examples. McGraw-Hill, 1971.

[BUR46] Burks, A. W., Goldstein, H. H. and von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", US Army Ordnance Department Report 1946.

[BUR81] Burks, A., and Burks, A., The ENIAC: First General Purpose Electronic Computer, The University of Michigan Press, Ann Arbor, Michigan, 1981.

[BUR88] Burks, A., and Burks, A., The First Electronic Computer: The Atanasoff Story, the University of Michigan Press, Ann Arbor, Michigan, 1988.

[FAG96] Faggin, F., Hoff, M., Mazor, S., and Shima, M., "The history of 4004", IEEE Micro, December, 1996, pp.10-20.

[GOL47] Goldstein, H., von Neumann, J., and Burks, A., "Report on the mathematical and logical aspects of an electronic computing instrument", Institute of advanced study, 1947.

[IOW99] Iowa State University, Department of computer science, http:// www.cs.iastate.edu/jva/jva-archive.shtml

[KAT93] Katz, R., Contemporary Logic Design, Addison-Wesley, 1993.

[KOO90] Koopman, P., An Architecture for Combinator Graph Reduction, Academic Press, 1990.

[LAV80] Lavington, S., Early British Computers, Manchester University Press, 1980.

[LEE95] Lee, J., Computer Pioneers, IEEE CS Press, Los Alamitos, California, 1995.

[MAN98] Manchester university, computer science department, MARK1, http:// www.computer50.org/mark1/firstprog.html

[MAN] The university of Manchester celebrates the birth of the modern computer, http://www.computer50.org/mark1/

[MOL88] Mollenhoff, C., Atanasoff: Forgotten Father of the Computer, ISU Press, 1988.

[SCH97] Schaller, R., "Moore's Law: Past, Present and Future", IEEE Spectrum, June, 1997.

[STE88] Steenkiste, P., Hennessy, J., "Lisp on a reduced-instruction-set computer: characterization and optimization", Computer, vol.21, no. 7, July 1988, pp.34-45.

[STN80] Stern, N., "Who invented the first electronic digital computer?", Annals of the History of Computing, 2:4 (October), 375-376.

[TUR37] Turing, A., "On Computable Numbers, with an application to the Entscheidungsproblem", Proc. Lond. Math. Soc. (2) 42 pp 230-265 (1936-7); correction ibid. 43, pp 544-546 (1937).

[WAR83] Warren, D., "An abstract Prolog instruction set, Technical report 309, SRI, 1983.

Exercises

1.1 Write two realisations of the computation of summation of *1..n*, one is data flow paradigm, the other one is in a conventional data path.

1.2 A conventional way of thinking about program is that a program processes input to output. A new way of thinking about program is that an event occurs then program responds to it. One can form "if..then" rules into a program to reflect this new thinking. Write a program to sum *1..n* using the "if..then" rules.

1.3 Suppose we do not have programs. How can we built a circuit to solve Tower of Hanoi problem? (It requires a recursive program to solve it).

1.4 The performance factors: the number of instruction executed and the cycle per instruction are interrelated. Can it be possible that we design a computer system to succeed in reducing both factors at the same time? Please give examples from existing computer systems.

1.5 The question "who built the first electronic computer?" was a topic of debate in the last decade. There was the case of Maunchly and Eckert versus Atanasoff. In the end Atanasoff is credited. Look up the detail of the case in the internet. Describe what happened.

24

Chapter 2

High Level Language Nut

Programming is still an art. It requires skill which is acquired through a lot of practice. Its foundation lays in mathematics. The study of programs as an object in itself is interesting and useful. By such study we can understand more thoroughly the relationship between a program and the result we want it to accomplish. It is my intention in this chapter to introduce the study of programs. It will give some insight into programming and gives an appreciation of programs as beautiful man-made objects. A particular high level language called Nut is defined. Nut is the language used to describe all aspects of the system studied in this text. Its syntax and semantic including the internal form will be studied in this chapter.

2.1 Motivation

I will describe a language, Nut language. Nut is inspired by a language defined by S. Kamin in the chapter 1 of his textbook [KAM90]. The beauty of this language stems from its smallness and its elegance. There are 11 words which are already defined (called reserved words). Only one form of syntax rule is required, using only two characters as syntactic features (the left and right parenthesis). The grammar for this language can be written down in just a few lines. Despite of its look of a toy-language, the beauty of its completeness can be illustrated by showing that the whole executable system including a parser and an evaluator can be completely written in itself.

2.2 Language Nut

Nut employs the same syntax. It is actually originated in LISP [MCA65]. Nut has a few simple data types such as array and string. The aim of Nut language is for teaching. It has been used in several computer architecture classes to teach

how high level programming languages and machine codes are related. The whole language translation process is simple enough that students can modify it to generate code for their studies.

Nut has a very simple syntax. It is a prefix language and has only one form, (op arg*). It is designed to be minimal to make it easy to understand. The internal code is a non-linear code (called N-code). N-code is the data structure representing a program in Nut language. It has a simple static memory model for efficiency, and it also has dynamic allocation for flexibility.

The basic element in Nut is an expression. An expression returns a value, except for an assignment which does not return any value. A variable is evaluated to its value. Nut has a very small set of operators as it is intended to be used as a teaching tool. It has a small set of reserved words:

def, let, enum, if, while, do, set, setv, vec, new, sys.

The operators are: + - = < >

Variables

Nut has three types of variable: global, local, array. A global variable must be declared outside a function definition before it is used, for example (let v) declares a global variable v. A local variable's scope is in its defined function. An array variable has its space allocated by calling (new n) where n is the size of the array and assigns the return value to the array variable. An array is dynamically created. Its space is allocated from the heap.

Simple illustrative examples of Nut programs

The easiest way to introduce a new language is to illustrate many examples of the use of elements of language. The Nut language is printed in Arial font.

1. A simple expression

   b + c + d => (+ b (+ c d))

2. An assignment

`a = b - d` => (set a (- b d))

3. A while loop

```
while i < 11
  i = i + 1
```

(while (< i 11) (set i (+ i 1)))

4. An if expression

```
if a > 2 then b = 3 else b = 4
```

(if (> a 2) (set b 3) (set b 4))

5. A sequence of expression

```
s = 0
a = a + 2
b = 3
```

(do (set s 0) (set a (+ a 2)) (set b 3))

Access data structure

6. Declaring and allocating an array. A global variable must be declared before its use.

`ax[20]` =>

(let ax)
(set ax (new 20))

7. Getting a value of an element of an array

`ax[i]` => (vec ax i)

8. Setting a value of an element of an array

`ax[k] = 4` => (setv ax k 4)

9. Defining a function

```
sq(x) is x * x
```

```
(def sq (x) () (* x x))
```

A function with local variables, swap interchanges ax[a] and ax[b] using a local variable t.

```
(def swap (ax a b) (t)
  (do
  (set t (vec ax a))
  (setv ax a (vec ax b))
  (setv ax b t)))
```

10. To help readability, the enum is used to create symbolic names.

```
(enum 10 xAdd xSub xLit)
```

xAdd is 10, xSub 11, xLit 12.

Some elegant examples: define some primitives using only: if , =, <.

```
(def and (x y)() (if x y 0))
(def or (x y)() (if x 1 y))
(def not (x)() (if x 0 1))
(def eq (x y)() (= x y))
(def neq (x y)() (not (= x y )))
(def lt (x y)() (< x y))
(def le (x y)() (or (< x y ) (= x y )))
(def gt (x y)() (not ( le x y )))
(def ge (x y)() (not (< x y )))
```

2.3 Nut syntax

Every sentence in Nut is expression. An expression has the form

(op e)

Where e denotes an expression, op can be any reserved word or a user-defined word. The control-op has the following syntax.

(set name e)
(if e1 e2 e3)
(while e1 e2)
(do e1 e2 ... en)

The name of a variable and a user-defined word can be any string of characters except the reserved words. The syntax for defining a user function (not built-in) is

(def name (formals) (locals) e)

where formals are the list of formal parameters, locals is the list of local variables. e is the body of the function.

The grammar for Nut is as follows. (* denotes zero or more repetition, terminal symbols are in bold)

```
toplevel ->   e | define-op
e ->          name | control-op | value-op | data-op
control-op -> ( if e1 e2 e3 ) |
              ( while e1 e2 ) |
              ( do e1 e2 ... en )
value-op ->   ( op args )
data-op ->    ( set name e ) |
              ( vec name idx ) |
              ( setv name idx v )
define-op ->
              ( def name ( formals ) ( locals ) e ) |
              ( let name ) |
              ( enum number name ... )
op ->          + | - | = | < | > | name
args ->       name* | number*
formals ->    name*
locals ->     name*
number ->     integer
```

A *name* is the identifier name. There are three types of names: global, local and enumerate. A *global* variable must be declared (using "let") before its use. A *local* variable is declared inside a scope of the function definition. The *enumerate* is used as a symbolic name referring to some constant value. The *define-op* is the defining operator. There are three define-ops: def, let, enum. The *value-op* is the value producing operator. The operators are + − = < >. The *control-op* is the flow control operator: if, while, do. The *data-op* is the data access operator: set, setv, vec.

2.4 Nut semantic

To understand the meaning of a program, the meaning of each of its element must be understood. The arithmetic operators (*value-op*) have their usual meaning on the domain of integer. They evaluate all their arguments which must return integers then apply the operator to these arguments and return the value of integer. The *value-op* including a function call evaluates all of its arguments before applying the operator. This is called *call-by-value* semantic. The other possible meaning is the *call-by-reference*, it is not used in this language. The *control-op* treats its arguments in a different way.

```
(set name e)
```

"set" assigns a value of an expression e to the variable "name". A variable can be local or global. A variable is local when its name is listed in the formal parameters of the current function otherwise it is global. The returned value is the value of e.

```
(if e1 e2 e3)
```

"if" evaluates e1 and if its value is non-zero (true) it evaluates e2 otherwise evaluates e3. The returned value is the value of the last expression it evaluates.

```
(while e1 e2)
```

"while" is an iterative operator. It evaluates e1, if its value is non-zero it evaluates e2. This process is repeated until e1 returns zero. The returned value is the value of e2 before the loop terminate.

```
(do e1 e2 ... en)
```

"do" is a sequencing operator. It evaluates e1 e2 ... en sequentially and returns the value of en.

```
(def name (formals) (locals) e )
```

The *define-op* is used to define a user-defined function. Recursion is quite natural in Nut.

```
fib n is
  if n < 3 then
     return 1
 else
    return fib(n-1) + fib(n-2)
```

```
(def fib (n) ()
   (if (< n 3)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Example: a program to solve tower of Hanoi problem

```
(let num)                ; a global array

; define function "mov" with 3 arguments: n, from, t
; and one local variable: other

(def mov (n from t) (other)
   (if (= n 1)
      (do
      (setv num from (vec num (- from 1)))
      (setv num t (+ (vec num t) 1))
      ; else
      (do
      (set other (- 6 (- from t)))
      (mov (- n 1) from other)
      (mov 1 from t)
      (mov (- n 1) other t))))
```

```
(def main () (disk)
   (do
   (set num (new 4))
   (set disk 6)
   (setv num 0 0)
   (setv num 1 disk)
   (setv num 2 0)
   (setv num 3 0)
   (mov disk  1 3)))
```

System calls

To enable input/output and other system functions, Nut uses a primitive "sys". Sys has a variable number of arguments; the first one is a constant, the number that identifies the system function. Sys is used to implement library functions such as print, printchar, etc. Its implementation is dependent on the platform. For a PC, sys is implemented with an implementation language (our implementation used C). The following is the list of available system functions:

(sys 1 a)	print integer
(sys 2 c)	print character
(sys 3)	get character

Many system calls are introduced in the later chapters to facilitate low-level system dependent functions.

2.5 Data structures

How all the data structure can be implemented in a system which provides only scalar values in integer domain? For example, how to implement a pointer (so that we can have linked-list and others)? In order to provide an aggregate of data, a general mechanism to provide an indirect access to memory is an array. An indirection can be regarded as accessing an array using the "base" address and the "index". An index is not an "address", it is an ordinary integer. If we know the "base" of the data, then the reference to the data is just an offset (the index) from the base. To access an array we need 3 operators: new, setv, vec in this syntax:

```
(new size)
(vec name index)
(setv name index value)
```

"new" allocates memory of "size", where "size" is an expression, for example (* 4 10) or 40. What really is "name"? It is a variable, just like the variable that is defined and set its value by "set", for example,

```
(set name 1)
```

and the name stores the reference to that memory.

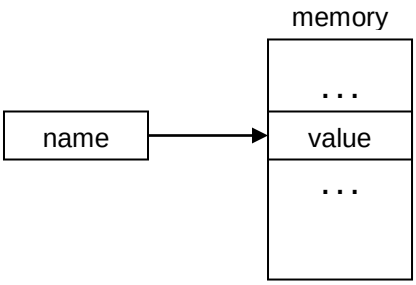


Figure 2.1 A name stored a reference to a memory

The memory in our system is a heap, a large block of memory with the base address at "heap". To address anywhere in the heap we use "ref" which is an index into this array of integer.

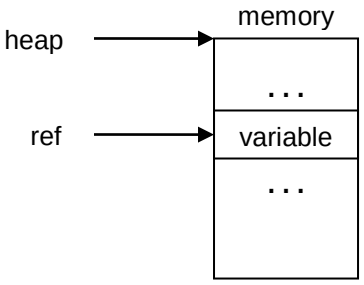


Figure 2.2 A heap in a memory

"vec" evaluates its argument ("name"), gets its value, which is the "ref" to the data segment and using this reference (plus index) to get the value. This indirection is called *dereferencing*. "setv" similarly performs storing a value into a name indirectly.

With "vec" and "setv" you can define access functions to your user-defined data structure. A calculation on pointer to a variable becomes an ordinary arithmetic on integer because the reference is just an integer.

The following is a program to copy one array to another (in the example copy y to x)

```
(enum 10 N)
(let a1)
(let a2)

(def array-copy (x y n) ()
   (if (= 0 n) 0
       (do
       (setv y n (vec x n))
       (array-copy x y (- n 1)))))

(def main () ()
   (do
   (set a1 (new N))
   (set a2 (new N))
   (array-copy a1 a2 N)))
```

2.6 String

An array is used to store a string in Nut. A constant string is useful in a source program, for example to present an error message. It is converted into a constant array at compile time. Strings in Nut are implemented with a word-aligned addressing in mind. A string is an array of integer. The string is terminated by an integer 0. See the following program for string manipulation, a string copy.

```
; copy s1 = s2

(def strcpy (s1 s2) (i)
   (do
   (set i 0)
   (while (neq (vec s2 i) 0)
      (do
      (setv s1 i (vec s2 i)
      (set i (+ i 1))))
   (setv s1 i 0)))

(def main () (s1)
   (do
   (set s1 (new 20))
   (strcpy s1 "test string")))
```

The compiler translated a constant string in the program text into a constant pointed to data segment storing the string.

2.7 Readability

How easy it is to read a program? This is very much dependent on prior experience. It is a matter of syntax or *form* of the language. Three major types of syntax (based on the concept of operator) are: prefix, infix, and postfix. Most of us grow up to be familiar with infix syntax; (a * 2) + b. For us, this is easier to read than prefix syntax; (* a (+ 2 b)), or postfix syntax; a 2 b + *. The meaning of three forms is the same. However, the difficulty of parsing them is different. The infix syntax requires specifying precedence of operators for the correct association and needs parentheses in places where that precedence must be overridden. A grammar can be written to deal with the precedence. On the other hand, parsing of a prefix and a postfix expression is trivial. Parsing the infix and prefix expression naturally results in a structure of tree while the postfix expression can be transformed into a linear structure easily. However, although a prefix language is trivial to parse, it tends to need a lot of parentheses especially on the far right-hand of the expression which is hard to get it right without the help from an editor that can match parentheses automatically.

The *model* of language also affects its form. The current language distinguishes between *statement* and *expression*. An expression has well-defined mathematical meaning, evaluating an expression returns a value. A language can have

expression as the only basic unit. This will make it more compact. Consider the following example:

(if x y 0) is the same as if(x) then return y; else return 0;

We are more familiar with the right-hand side than the left-hand side (LISP). However you can notice that the left-hand side is much more compact than the right-hand side. The meaning is "evaluate x, if true then evaluate y else evaluate 0". The value returned is the value of the last evaluated expression. There is no need to explicitly "return". The examples of real languages with different syntax are; prefix language, LISP [MCA65], postfix language, FORTH [MOO70] and Postscript.

Let us consider an example of adding one to a variable.

infix syntax	a = a + 1
prefix syntax	(= a (+ a 1))
postfix syntax	&a a 1 + =

For the infix and prefix syntax, the operator "=" (assign) treats its first argument "a" as special, it is an address. For postfix syntax this must be done explicitly using another operator "&". You can not write it the other way. The postfix expression must be understood using the *model* of stack. The central concept is the evaluation stack. Evaluating a variable pushes its value into the stack. An operator takes its argument from the stack and pushes its result back. *Form* also affects the way an operator works. This is an infix language (C):

a[1] = a[2] + 1;

It actually means *(&a + 1) = *(&a + 2) + 1;

The "=" here does not have the same meaning as in a = a + 1 because it takes the left-hand argument as an expression which must be evaluated to give a value as address where as the "=" in a = a + 1 takes a simple value directly. The parser must know this difference.

2.8 Iteration versus Recursion

Programs can be written in iterative or recursive style. The following examples contrast two styles.

```
(def findName2 (name i) (found)
   (do
   (set found 0)
   (while (and (<= i numNames) (not found))
      (if (streq (def-name-at i) name)
         (set found 1)
         (set i (+ 1 i))))
   (if found i 0)))

(def findName3 (name i)
   (if (> i numNames) 0
   (if (streq (def-name-at i) name) i
   (findName3 name (+ 1 i)))))
```

"findName2" performs a linear search for a name in the symbol table (def-name). "findName2" is iterative and uses "found" to break the while loop. "i" is set to "i + 1" for the next iteration. "findName3" is recursive, "i + 1" is passed as a parameter to the next recursion. Please note the absence of "set" in the recursive version.

The next example is the function "atoi" which converts a string such as "-1234" into its value -1234.

```
(def atoi4 (s1 i) (m)
   (do
   (set m 0)
   (if (= 45 (vec s1 0)) (set i 1) 0)
   (while (!= 0 (vec s1 i))
      (do
      (set m (+ (* 10 m) (- (vec s1 i) 48)))
      (set i (+ 1 i))))
   (if (= 45 (vec s1 0))
      (- 0 m)
      m)))

(def atoi (s1) ()
   (if (= 45 (vec s1 0))
      (- 0 (atoi2 (+ 1 s1) 0))
      (atoi2 s1 0)))
```

```
(def atoi2 (s1 m) ()
   (if (= 0 (vec s1 0))
      m
      (atoi2 (+ 1 s1) (+ (* 10 m) (- (vec s1 0) 48)))))
```

"atoi4" uses iteration with "i" as an index of character and "m" as a local variable storing the converted value. "atoi" and "atoi2" are the recursive version. "atoi" handles the negative sign and calls "atoi2" to convert the string. You can see the simplicity of the structure in the recursive version and the lack of "set".

You may think that recursion consumes more memory and runs slower than iteration. Let us expose more details of this argument. First, the memory concern, most procedural languages use stack to store all local variables and actual parameters. Recursive call will consumes this stack where as iteration does not. However, for the case that the recursive call is the last function executed in a user-defined function, so called *tail-recursion*, this stack growth can be eliminated.

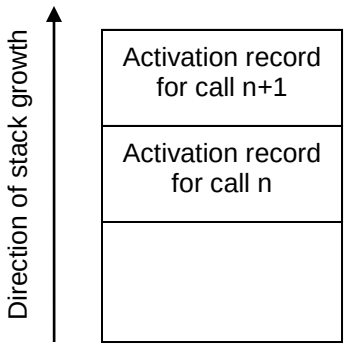


Figure 2.3 A nested call (including recursion) causes growing of activation records

We can eliminate the activation record of the next call (n+1) by realising that the call is the last function executed hence all local variables and parameters of the current activation need not to be saved (as they are not used anymore). The actual parameters of the next recursive call can substitute the current activation

record in-place. A parser or a compiler can recognise tail-recursion and performs this optimisation.

Second, the speed concern, the speed of recursive call can be slow due to the overhead of a function call. A function call requires calculating a number of pointers to adjust the stack. Where as for the iteration the loop can be achieved by "jumping" which is a cheaper operation than a call. It depends on the implementation how much this difference will be.

2.9 Internal forms

When an expression (in a source language) is processed, it is transformed into an internal form before it is evaluated (the internal form is also used to generate executable codes). This internal form has the structure in the form of a tree (inverted, the root is at the top). This internal form is distinct from the surface language. One surface language may have different internal forms and different surface languages may have the same internal form. You can think of an internal form as a machine language and a surface language as a high level language. However, an internal form is not a machine language. It is not directly executable by any processor (except you want to design a special processor for it). There is a program that takes an internal form and runs it. This program is called in many names: an interpreter, a virtual machine or an evaluator.

Suppose we have a function power(x, y) which raises x to the power y.

```
(def power (x y)
   (if (= 0 y) 1
   (if (= 1 y) x
   (* x (power x (- y 1))))))
```

The expression defining the body of power can be drawn as Fig. 2.4. A general purpose linked structure is used to represent this tree structure, called *list*. List composed from two kinds of nodes: dot-pair and atom. A dot-pair stores two components; first component is a pointer to an element of the list and second component is a link to other dot-pair. An atom stores information (or element of list). See the following example: (atom is shown in CAPITAL letter and list is `(.)`. / is NULL pointer signifying the end of a list.)

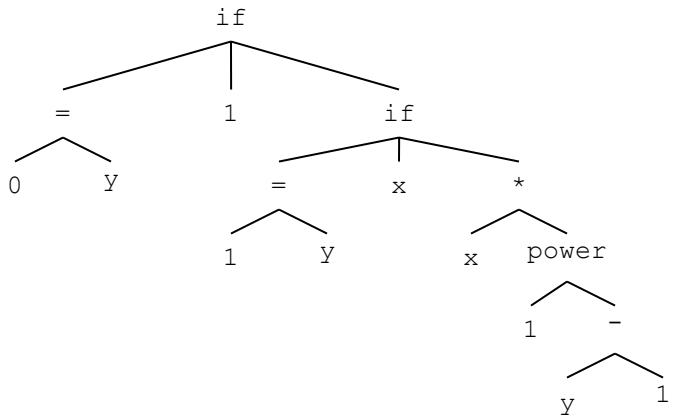


Figure 2.4 The tree representing an expression

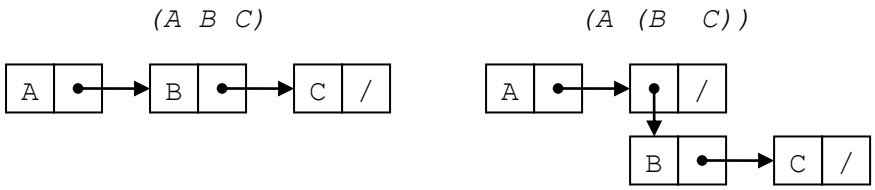


Figure 2.5 Lists can be represented by linked dot-pairs and atoms

The internal form composed of the linked-list nodes with two fields: head and tail. Now we will draw the previous program (power) in this concrete form. The type of node is denoted by `V` (value), `L` (local), `G` (global) and `A` (application).

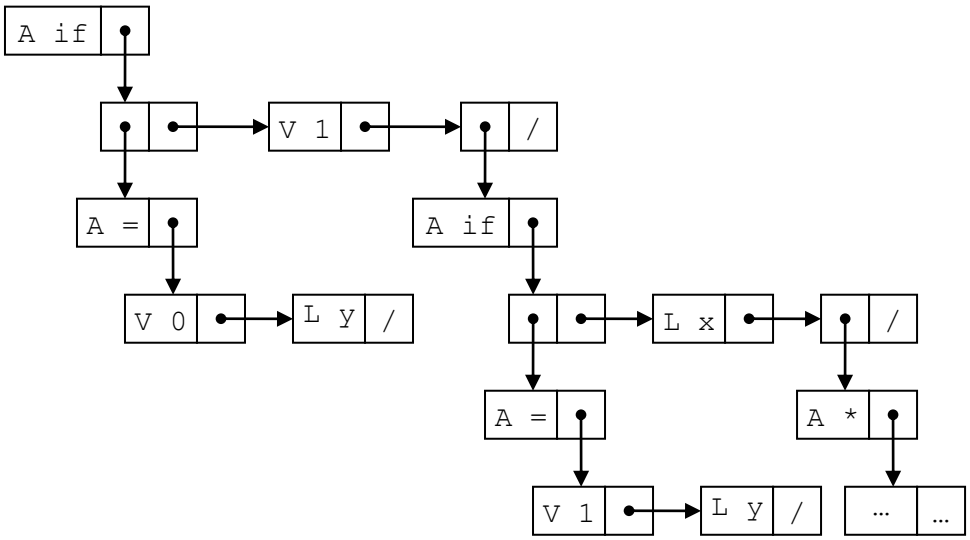
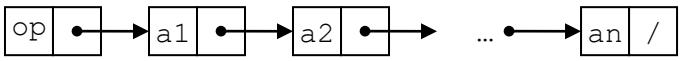


Figure 2.6 The internal form showing the function (power x y)

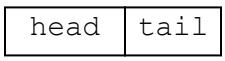
How this internal form is implemented depends on the choice of data structure. In the next section we will discuss this implementation issue in more details.

2.10 N-code

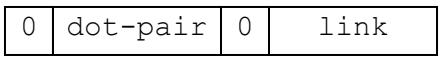
N-code is the internal form of Nut language. The structure of program is a list, composed of dot-pairs. An instruction has the form (op a_1... a_n) represented by

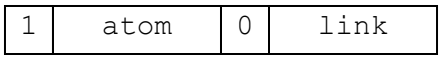


"op" is an atom. Arguments can be either atom or list. A dot-pair composed of a pair of head and tail cells:

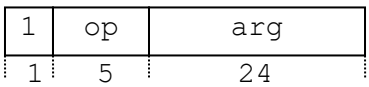


The head stores an atom or a pointer to other cell. If it is an atom, the first bit is "1", otherwise it is a dot-pair (a pointer to other cell), and the first bit is "0". The tail stores a pointer to other cell, called "`link`". The basic data structure is a pair of consecutive cells which each cell is large enough to store an atom or a link. There are two kinds of pairs: dot-pair/link and atom/link.





An atom encodes an instruction and one argument.



For a 32-bit system, a cell is 32-bit. A pointer to cell is 31-bit (as one bit is used to encode atom/dot-pair). The "`op`" is 7-bit, the "`arg`" is 24-bit.

2.11 N-code instruction set

N-code instruction set is a definition for the internal representation of Nut language. The instruction follows from the Nut language pluses some extra instructions to implement precise operational semantic of Nut language. The instruction set is divided into four groups: control, value, arithmetic and system. Each instruction has the form of an atom with 7-bit opcode and 24-bit argument.

Control	if while do call fun
Value	get put ld st ldx stx ldy sty lit str
Arithmetic	+ − = < >
System	new sys

Encoding

Table 2.1 N-code and its encoding

1 if	2 while	3 do	5 new	6 add
7 sub	10 eq	11 lt	12 gt	13 call
14 get	15 put	16 lit	17 ldx	18 stx
19 fun	20 sys	25 ld	26 st	27 ldy
28 sty	32 str			

Totally there are 22 instructions in N-code instruction set. Only value-instructions have arguments, denoted by "op.arg". "fun" has special arguments (to be explained later). "call" has a pointer to its body of a function (the N-code) as its argument.

To understand its operational semantic, we need to know its run-time environment. The run-time environment consists of an evaluation stack and the data segment which provides the place to hold all global, strings and array data.

The evaluation (execution) of a program employs a stack data structure. This evaluation stack has two purposes, one is to store a dynamic local context, called *activation record*, and the second purpose is to be a temporary stack to store the intermediate results. All local variables are accessed through the activation record. When a function is evaluated, it has its local environment (local variables and stack area). The activation record is maintained through two global pointers: FP (frame pointer), and SP (stack pointer). FP points to the activation record. SP points to the temporary stack area. SP is on top of FP.

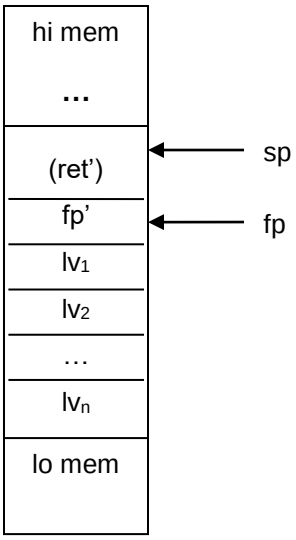


Figure 2.8 An activation record

An activation record has the following structure. At FP, the previous FP (FP') is stored so that the old context can be restored after the current context is complete at the end of a function call. The return address is stored next on the top of FP. This return address is used to restore the instruction pointer (or so called program counter) to enable a program to return to its caller. Storing a return address in the context is necessary in a real processor which implements a data path based on traditional architecture. It is not necessary if we implement an evaluator of N-code as software because the evaluator can be implemented as recursive calls to evaluate each instruction and follows the "`link`" field without using any instruction pointer (the next chapter discusses this Nut-evaluator)[1].

To access local variables, the argument of value-instruction is an index relative to the frame pointer. For example, to get a value of a local variable 3, the instruction "get.3" accesses Mem[FP-3] where Mem[.] is the N-machine memory. Usually this part of memory is called *stack segment*.

[1] Another possibility is to implement a special processor to execute N-code directly using the recursive evaluation style (with recursive microprogramming). This alternative also does not need to store a return address in the context.

The program written in N-code is presented as a list of N-code. It looks similar to the source language Nut but the node values are the operational codes not the tokens of the high level language.

Here is a simple example. A program in Nut to compute a Fibonacci value is shown below.

```
(def fib (n) ()
   (if (< n 3)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This program when translated into N-code will look like this.

```
(fun.1
   (if (lt get.1 lit.3)
      lit.1
      (add (call.fib (sub get.1 lit.1)) (call.fib (sub get.1 lit.2)))))
```

The object code represented in the code segment is shown below. The format of object code is as follows. Each line of object code represent one pair of cells written as a tuple of {*address tag op arg link*} where *address* denotes the address of this cell in the code segment, *tag* denotes the first bit – 0 for dot-pair, 1 for atom, *op* denotes the operation code, *arg* denotes its argument, *link* denotes the address of the next cell. The code is generated using the preorder traversal of the source expression; hence the code is generated with the left-most, depth-first order. The last line of the object is the entry point of the expression.

```
2 1 16 3 0
4 1 14 1 2
6 1 11 0 4
8 1 16 1 0
10 1 14 1 8
12 1 7 0 10
14 0 0 12 0
16 1 13 44 14
18 1 16 2 0
20 1 14 1 18
22 1 7 0 20
```

```
24 0 0 22 0
26 1 13 44 24
28 0 0 26 0
30 0 0 16 28
32 1 6 0 30
34 0 0 32 0
36 1 16 1 34
38 0 0 6 36
40 1 1 0 38
42 0 0 40 0
44 1 19 257 42
```

The code above can be read as follows.

```
44 1 19 257 42
```

It is an atom "fun", the next link pointed to Mem[42].

```
42 0 0 40 0
```

This is a dot-pair with the Mem[40] as the first element and the only element of this list as the next link contains 0 signified the end of list. This 0 is usually called a NIL atom.

```
40 1 1 0 38
```

This is an atom "if", the next link pointed to Mem[38].

```
38 0 0 6 36
```

This is a dot-pair, the first element is Mem[6], the next link pointed to Mem[36].

```
6 1 11 0 4
```

This is an atom "lt", with its argument at Mem[4].

```
4 1 14 1 2
```

This is an atom "get.1" and the next argument is Mem[2].

```
2 1 16 3 0
```

This is an atom "get.3" and the list of argument ends here. These three lines can be written out as:

```
(lt get.1 get.3)
```

Other lines of object code can be read similarly. We will not pursue reading the object code of the argument of "if" at Mem[36] any further.

2.12 Meaning of instructions

Now we discuss the meaning of each instruction with respect to its run-time environment. Mem[.] denotes the memory. SS[.] denotes part of the memory that is designated for the stack segment. The code and data are stored in Mem[.] and are called the code segment and the data segment respectively.

The notion of meaning is best explained as the effect of each instruction on its environment. This style of describing the meaning to a program is called *operational semantic* (other ways to describe semantic are axiomatic, denotational and functional). This can be presented as a function "eval()" which takes a valid expression of N-code and produces its result. This function eval() is the evaluator of N-code. It can be implemented both in software (as a virtual machine) or hardware (a special processor that executes N-code directly).

Control-instruction

```
(if e1 e2 e3)
```

"if" does a conditional execution. If eval(e1) is true then eval(e2) else eval(e3)

```
(while e1 e2)
```

"while" performs a repeat loop. While eval(e1) is true eval(e2) repeatedly, it returns the last eval(e2)

```
(do e1 ... en)
```

"do" is a sequencing operator. eval(e1) then eval(e2) ... eval(en) return eval(en)

(call.x e1 e2..en)

The above expression calls a function with the argument list (e1..en). The element of this list is evaluated one-by-one, eval(e1)... eval(en), the results are pushes to the evaluation stack and then goto eval the body of function at x.

(fun.a.v e)

"fun" is an operational code at the beginning of a function definition. It creates a new activation record. The arguments of the function call are passed from the evaluation stack to this environment, and the body of function is evaluated. Once the evaluation of the body is finished, the activation record is deleted. Two parameters are required to handle creation and deletion of the activation record: arity and the size of frame. The encoding is "fun.a.v" where a is arity, v is the size of frame. The size, v, is used in the deletion of activation record, k is v-arity+1, used in the creation of activation record

The action of "fun.a.v" is:
SS[.] denotes stack segment

```
k = v-a+1              offset from SP
SS[sp+k] = fp          new frame
fp = sp+k
sp = fp
v = eval(e)            eval body
sp = fp-v-1            delete frame
fp = SS[fp]            restore old FP
```

Value-instruction

The argument is the index to a local variable. It is relative to the frame pointer.

get.a return SS[FP-a].

(put.a e) SS[FP-a] = eval(e), return eval(e).

(ld.a) load, a is global, return Mem[a].

(st.a e) store, a is global, Mem[a] = eval(e), return eval(e).

(ldx.a e) load with index, a is local, return Mem[SS[FP-a] + eval(e)].

(stx.a e1 e2) store with index, a is local, Mem[SS[FP-a] + eval(e1)] = eval(e2), return eval(e2).

(ldy.a e) load with index, a is global, return Mem[Mem[a] + eval(e)] .

(sty.a e1 e2) store with index, a is global, Mem[Mem[a] + eval(e1)] = eval(e2), return eval(e2).

lit.a return a.

str.a a string constant, a is a pointer to a string, return a.

Arithmetic

(bop e1 e2) bop are + − = < >. The operators have their usual meaning, return eval(e1) bop eval(e2).

System

System instructions perform the task of input/output and other services related to operating system. On a real processor, the system instructions are implemented differently due to their dependency on a target machine. However, we define these instructions for their use in the simulation. The result of input/output can be simulated on the simulator.

(new e) return pointer to a newly allocated chunk of memory of size eval(e).

(sys.a e) system call sys.a
a = 1 print integer eval(e)
a = 2 print character eval(e)
return eval(e)

Example of programs written in N-code

An expression

```
(= a (+ b 1))
```

```
(put.a (+ get.b lit.1))
```

Function definitions

```
(def double (x) () (+ x x))
```

```
(fun.1.1 (add get.1 get.1))
```

```
(def sum (a b s) ()
  (if (> a b)
      s
      (sum (+ a 1) b (+ s a))))
```

```
(fun.3.3
   (if (gt get.a get.b)
      get.s
      (call.sum (add get.a lit.1) get.b (add get.s get.a))))
```

A quicksort program

```
(def partition (a p r) (x i j flag)
   (do
   (set x (vec a p))
   (set i (- p 1))
   (set j (+ r 1))
   (set flag 1)
   (while flag
      (do
      (set j (- j 1))
      (while (> (vec a j) x)
         (set j (- j 1)))
      (set i (+ i 1))
      (while (< (vec a i) x)
         (set i (+ i 1)))
      (if (< i j) (swap a i j) (set flag 0))))
   j ))
```

```
                    (fun.3.7
                       (do
                       (put.4 (ldx.1 get.2))
                       (put.5 (- get.2 lit.1))
                       (put.6 (+ get.3 lit.1))
                       (put.7 lit.1 )
                       (while get.7
                          (do
                          (put.6 (- get.6 lit.1))
                          (while (> (ldx.1 get.6) get.4)
                             (put.6 (- get.6 lit.1)))
                          (put.5 (+ get.5 lit.1))
                          (while (< (ldx.1 get.5) get.4)
                             (put.5 (+ get.5 lit.1)))
                          (if (< get.5 get.6)
                             (call.swap get.1 get.5 get.6 )(put.7 lit.0)))
                       get.6))

(def quicksort (a p r) (q)
   (if (< p r)
      (do
         (set q (partition a p r))
         (quicksort a p q)
         (quicksort a (+ q 1) r))
      0))

                    (fun.3.4
                       (if (< get.2 get.3)
                          (do
                             (put.4 (call.90 get.1 get.2 get.3))
                             (call.135 get.1 get.2 get.4)
                             (call.135 get.1 (+ get.4 lit.1 )get.3))
                          lit.0))
```

2.13 Summary

We have introduced a high level language that will be used to describe all levels of the computer system in this text, Nut. The language itself is intended to be a minimal language in a sense that it is a very small language usable for our purpose and yet Nut is complete. It can be used to write its own compiler and evaluator which will be the topic of the next chapter. The smallness of Nut allows us to investigate its semantic in full details. The intermediate representation of Nut language is N-code. N-code is a *concrete* presentation of Nut language. The operational semantic of N-code can be defined over its run-time environment. Given this semantic, a code generator for a target processor can be implemented or a special processor can be designed to directly execute N-code.

2.14 Further reading

Language design is a topic of broad spectrum. Most languages in the past have been constrained by the machines that existed in their period. Overwhelming concern was the issue of machine efficiency. A large survey of computer language is described in [HOR83]. However, as the computing machines become faster and are available abundantly, the emphasis is shifted to the topic of compatibility and standardisation. The history of programming languages is interesting, see [WEX78] [BER96]. It takes time for a language to be widely used and for programmers who are skillful with a language to become available for the industry. Presently, Java [JOY00] dominates the programming in IT industries. It popularises the concept of the *intermediate* language (as JVM the virtual machine [LIN97] is available on almost any platform). The computer language is still evolving. A new language, especially the dynamic language such as the special purpose scripting language, comes into being every year. A special purpose language has its advantage that it can be designed to facilitate the specific programming task, such as Game design, or real-time control task. Special language features can be embedded as primitives in the language such that they are very easy to use. This can reduce the error from programmers. For example, a concurrent language for control tasks can have the message-passing primitives including the semaphore as primitive data types [CHO98]. The future language will take human behaviour into account more than just the machine that run it.

References

[BER96] Bergin, T., Gibson, R., Gibson, R. Jr., History of programming languages, vol.2, ACM Press, Addison Wesley, 1996.

[CHO98] Chongstitvatana, P. A multi-tasking environment for real-time control. Final report, Faculty of Engineering, Chulalongkorn university, research project number 132-MRD-2537, 1998. Also available at http://www.cp.eng.chula.ac.th/faculty/pjw/r1/

[HOR83] Horowitz, E., Programming languages: a grand tour, Computer Science Press, 1983.

[JOY00] Joy, B., (Ed), Steele, G., Gosling, J., Bracha, G. Java(TM) Language Specification (2nd Ed), Addison Wesley, 2000.

[KAM90] Kamin, S. Programming Languages: An interpreter-based approach, Addison-Wesley, 1990.

[LIN97] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Addison Wesley, 1997.

[MCA65] McCarthy, J. et al. LISP 1.5 Programmer's Manual, MIT press, 1965.

[MOO70] Moore, C., and Leach, G. FORTH: A language for interactive computing, 1970.

[WEX78] Wexelblat, R., History of programming languages, ACM Press, 1978.

Exercises

2.1 Write a program in Nut to do reversing elements in an array. Try to compile and run it to see the result. Observe the object code. How is the object code (in N-code) corresponded to the source code?

2.2 Familiarise yourself with writing a program with the recursive style. Try the following.

a) Do the question 1 using recursion.
b) Search for an element in a linked list using recursion.

c) Write sum *1..n* using recursion. (Hint: use an accumulating parameter)

2.3 Write a Nut program to read the object code and print it out as a readable N-code in the form of expression in parenthesis.

2.4 The object code (N-code) includes many pointers to other cells. Suggest a way to save memory by reducing these pointers.

2.5 Due to the way N-code is represented, a maximum literal representable in Nut language is 24-bit (becuase a code in n-code is a 32-bit cell, an opcode is 7-bit, an argument is 24-bit). How to represent a larger literal?

Chapter 3

Nut Compiler

In this chapter we are going to describe Nut compiler which is written in Nut. A compiler can be written using the target language (of the compiler). Writing a compiler with its own target language demonstrates two points:

1. The language is not trivial. At least it can be used to write some complex program such as a compiler. This shows a kind of *completeness* of the language.
2. The understanding of meaning the language is complete enough to use it to write such a non-trivial program.

And our favourite third point:

3. It is beautiful, in a sense that the language is self-describing.

Writing a compiler with the target language is not new (for example you can write a C compiler in C and compile your C compiler into the executable code using any existing C compiler)[1]. It has been practiced especially in the early days of computer software development. Some conceptual difficulty must be overcome concerning the confusion of the "compiler" and the "compiled" program (since we use the compiler to compile itself!). The run-time facilities are always posing difficulty as the memory is shared between the compiler and the compiled program. However, these points are not a priority in our study.

A compiler translates a source code to a target code. In our study, the Nut compiler translates a Nut program to an N-code object. A code generator translates an object code to a machine specific code. Our code generator translates an N-code object to machine codes.

---

[1] The question arises as how the first compiler was written? This question is explored in [CHO05].

```
compiler
  input source program (in nut)
  output n-code

code generator
  input n-code
  output machine code of a specific processor
```

To develop "Nut-in-Nut" compiler, we will use a special version of Nut compiler (written in C), "nutc". This version of Nut-compiler has many additional operators that support compilation. To run the compiler, a version of the *N-code virtual machine* (or the interpreter for N-code), called "nvm" is used. Nvm contains many supporting functions to facilitate compilation (which are cumbersome to write in Nut or which we are not interested in discussing). Remember that our goal is to develop the Nut compiler itself including its own virtual machine. Nutc and nvm are the tools to bootstrap these programs. Once our version of compiler and virtual machine are working, the initial tools will become unnecessary.

Next, we will explain the output of the compiler. It is important to understand the N-code; it is what the compiler must produce.

3.1 N-code

An example, the source program to be compiled:

```
(def add1 x ()
  (+ x 1))

(def main () () (sys 1 (add1 2))
```

Where (sys 1 x) is a system call (analogous to OS call for I/O). It will print an integer x to the screen.

The N-code of the above program in human-readable form is:

```
add1
(fun.1.1 (add get.1 lit.1 ))
```

```
main
(fun.0.0 (sys.1 (call.17 lit.2 )))
```

and in an *absolute* object form (as in the output file a.obj):

```
22 22
2 1 16 1 0
4 1 14 1 2
6 1 6 0 4
8 0 0 6 0
10 1 19 257 8
12 1 16 2 0
14 1 13 10 12
16 0 0 14 0
18 1 20 1 16
20 0 0 18 0
22 1 19 0 20
0

17 add1 3 10 1 1
19 main 3 22 0 0
```

There are three parts in the object file:

1 the code
2 the data (the line contains a zero)
3 the symbol table

The code is a contiguous block of memory. The code "main" started at 22. The format of the object code is:

```
{address tag op arg next}
```

tag 0 is dot-pair
tag 1 is atom
op arg is the operator
next is the address of the next cell

Now, we will read the object code as follows.

```
22 1 19 0 20
```

It is an atom "fun.0" next is 20

```
20 0 0 18 0
```

It is a list (dot-pair), the head pointed to 18 and the next is NIL.

```
18 1 20 1 16
```

It is an atom "sys.1", next is 16 (the argument of sys.1)

```
16 0 0 14 0
```

It is a list, the head is 14, next is NIL.

```
14 1 13 10 12
```

It is an atom "call.10", next is 12 (the argument of the function)

```
12 1 16 2 0
```

It is an atom "lit.2", next is NIL.

At 10, is the "fun.x" (the function "add1") etc.

3.2 Compiler

The whole compiler is about 500 lines. We will describe the compiler using a pseudo code and sometimes in Nut language to illustrate some concrete implementation. Full listing of the compiler in Nut is available in the appendix B. We will refer to the source using the notation [name lineno].

The compiler has four main functions.

```
main [nut 432]
  readinfile
  parse
  resolve
  outobj
```

"readinfile" reads the source program from a standard input stream (*stdin* in Unix). The compiler reads the whole input at once and keeps it in a big array of characters (maximum input size is 50 Kbytes). "parse" is the main parser that scans the input stream and generates N-code. "resolve" performs renaming and binding of the actual code to generate the executable code. "outobj" prints out the object file to a standard output (*stdout*).

We will concentrate on "parse". To understand "resolve" you need to know the run-time system which is the topic of the next section.

Nut has a trivial syntax by design hence the parser for Nut is very simple. Our parser is a *recursive descent* parser. It calls parsing routines recursively with one look ahead symbol and never backtrack (this is called LL(1) parser [AHO86]). This is a simple and straightforward kind of parser. You can read more about this kind of parser in any standard textbook in compiler.

Before we look into the parser, we need to be able to scan the input which is the stream of characters and forms "token". This is called *lexical analyser*. The tokens are separated with special characters called *separator*. There are only three separators in Nut: space, "(" and ")". "tokenise" is one of the system call that is implemented in the "nvm", (sys 3). It parses the token and returns a string of characters (string of Nut language). Here is the sample use of "tokenise":

```
; token is a global variable pointed to the string
(let tok)

(def tokenise () ()    [nut 169]
   (set tok (sys 3))

; tokenise input stream until end of file
(def testtok () ()
   (do
   (tokenise)
   (while (!= (vec tok 0) EOF)
      (do
      (prstr tok)
      (space)
      (tokenise)))))
```

Where "prstr" is a function to print a Nut string. The end of file is signified by the first character of the returned string as EOF (127). Run testtok with the example stream and here is the output:

```
( def add1 x ( ) ( + x 1 ) ) ( def main ( ) ( )
( sys 1 (
add1 2 ) )
```

Now we take a look at the parser.

```
parse   [nut 309]
  tokenise
  while not EOF
    expect "("
    tokenise
    if token == "def"
      parseDef
    if token == "let"
      parseLet
    if token == "enum"
      parseEnum
    tokenise
```

The parser gets a token and calls "parseDef" or "parseLet" or "parseEnum" according to the token then loops until it reaches the end of file.

```
parseDef   [nut 269]
  tokenise              get fun name
  parseNL               get formal arg
  parseNL               get local
  tokenise
  e = parseExp          get body
  tokenise              skip ")"
  update symtab
  out (fun.k e)
```

"parseDef" parses the "header" of the function definition then the main part is "parseExp" to parse the body of the function definition. The declaration part of a function definition composed of:

(def add1 x () ...)

"add1" is the function name. "x" is the list of formal parameters. "( )" is the list of local variables. The list of formal and locals is parsed by "parseNL" (parse name list) which will store the formal and local names in the symbol table and gives them the references as a running number *1..n* in order of their appearance.

For sake of clarity, let's assume that the name list will not be an atom.

```
parseNL  [nut 185]
  tokenise
  while token != ")"
    installLocal token
    tokenise
```

"parseNL" merely gets a name and stores it in the symbol table using "installLocal" until exhausting the list (found the token ")").

Before getting into "parseExp", let's study the symbol table.

Symbol table

The symbol table is a one-dimension array of the entry. Each entry has five fields: *name, type, value, arity, lv*. "*name*" is a pointer to a string, the symbol. "*type*" is the type of symbol. The type value is shown below. "*value*" stores the value of the symbol, which is a reference for function, local/global variable, the opcode of an operator, the syscall number of "sys.x" or the value of an enum symbol. "*arity*" and "*lv*" are for the function symbol, storing its arity and total number of local variables.

Table 3.1 Type of symbols

2	VAR is a local variable
3	FUN is a function
4	OP is an operator
5	OPX is an op that has one special argument
6	SYS is "sys.x"
7	UD is undefined
8	GVAR is a global variable
10	ENUM is an enum symbol

The function "install nm" does a scan for "nm" (value of a pointer to a string, the string of symbol) in the symbol table. Searching a symbol table is efficiently implemented using a hash table. In our implementation, for simplicity, a sequential search is used. If "nm" is already present, "install" returns the index to that entry. If it is a new symbol, it is inserted into the symbol table and the function returns its index. Initially, all keywords are primed into the symbol table. They are treated the same as any other symbol.

The main part of "install" is shown here. The "getName", "setName", "setType" are the access functions for the fields in the symbol table. The variable "numNames" is the number of symbols in the table. "esize" is a constant value of 5, the size of an entry in the table. The "str=" is a string comparison function. The "newName nm" returns a copy of the string "nm".

```
; search symtab for nm
; if found, return its index, if not found, insert it

(def install nm (i flag end)   [nut 77]
   (do
   (set i 0)
   (set flag 1)
   (set end (* esize numNames))
   (while (and flag (< i end))   ; sequential search
      (if (str= (getName i) nm)
         (set flag 0)
         ; else
         (set i (+ i esize))))
   (if flag                      ; not found
      (do
      (if (> i MAXNAMES)
         (error "symtab overflow"))
      (setName i (newName nm))
      (setType i tyUD)
      (set numNames (+ numNames 1))))
   i))
```

We conclude the discussion of the symbol table here and go back to discuss the parser. The next function is "parseExp".

```
; An expression is a list, a number, a string or a name
parseExp   [nut 255]
  if token == "("
    tokenise
    nm = parseName
    e = parseEL
    out (nm e)
  if isNumber token
    n = atoi token
    out lit.n
  if isString token
    e = makestring token+1
    out str.e
  parseName                it is OP, OPX, VAR, FUN
```

The function "parseExp" parses an expression in Nut language. An expression can be: a name (such as a variable), a number, a constant string, a list (function application). For a list, "parseExp" uses two auxiliary functions to parse: parseName, parseEL (expression list).

```
parseName   [nut 214]
  type = symbol.type
  v = symbol.val
  switch type
    OP: out v
    VAR: out get.v
    GVAR: out ld.v
    FUN: out call.idx
    OPX:
      tokenise             get var name
      ty2 = symbol.type
      v2 = symbol.val
      if ty2 == VAR
        switch v
          SET: out put.v2
          SETV: out stx.v2
          VEC:  out ldx.v2
      else if ty2 == GVAR
        switch v
          SET: out st.v2
```

```
        SETV: out sty.v2
        VEC:  out ldy.v2
    SYS:
      tokenise
      k = atoi token
      out sys.k
    ENUM:
      out lit.v
```

"parseName" takes one token and depends on its type, it outputs an appropriate N-code. The table below shows the type and the N-code associated with it.

Table 4.2 Type and the associated N-code

OP	an operator, out *opcode*
VAR	a local var, out *get.ref*
GVAR	a global var, out *ld.ref*
FUN	a fun, out *call.ref*
SYS	a system call, gets sys num, out *sys.k*
ENUM	a enum symbol, out *lit.ref*
OPX	a special op

The operator of type OPX takes the next token as an "unevaluated name", i.e. a reference of that name, not its value. Three operators are OPX: "set", "setv" and "vec". Two possibilities for the next token, either it is a local variable or a global variable.

Local out put.ref, stx.ref, ldx.ref
Global out st.ref, sty.ref, ldy.ref

The last piece of the parser, "parseEL" recursively parses the rest of the list.

```
parseEL  [nut 243]
  tokenise
  if token == ")" return NIL
  e = parseExp
  e2 = parseEL
  out (e e2)
```

Now that the major part of compiler is completed, we turn our attention to the housekeeping task. The remaining parts are the "resolve" and the low level "tokenise". We will discuss only their pseudo code.

```
Resolve   [nut 368]
  for all func in symtab
    reName call and local var

reName   [nut 356]
  if op == get, put, ldx, stx
    rename local var          (lv-arg+1) 1..n to n..1
  if op == call
    update reference
```

The rename function changes the number of local variables by reversing their order. The reason ties to the way an activation record is created. This run-time behaviour is discussed in the next section. Once the compiler reaches the end of input source, all references to functions should be known. Initially the "call" to a function has the argument as the index to that function in the symbol table. "resolve" also instantiates the actual reference to all "call" instructions. Now the last bit, the tokeniser.

```
tokenise
  skip blank
  get a char
  if isSpecial  return
  if isQuote                  it is a string
    get to other quote
  else
    get to delimiter          LP RP blank
```

"tokenise" leaves a token string in token[.]. The tokeniser is implemented in the nvm and is available to be used as a system call, (sys 3). See the earlier discussion of the function "tokenise".

3.3 How to compile and run Nut-compiler

The first Nut-compiler is written in C, "nutc". It is used to compile the Nut-compiler describing in this chapter. Nutc outputs the object code to a file, named

"a.obj". This object is executable by a Nut virtual machine "nvm". Here is a sample session of compiling the Nut-compiler in the file "nut.txt":

```
c:>nutc < nut.txt

!=
(fun.2.2 (if (eq get.1 get.2 )lit.0 lit.1 ))
and
(fun.2.2 (if get.1 get.2 lit.0 ))
...
main
(fun.0.0 (do
(call.79 )(call.124 )(call.157 )(call.29 )(st.3
(sys.9 ))(call.145 )(call.155 )(call.156 )(call.2
3 )))
```

A lot of human-readable N-code is displayed on the screen. It can be visually checked whether it is correct (no error message). The object code can be executed under the nvm simulator. Nvm loads "a.obj" by default (to save *stdin* for Nut-compiler to use to read its source) and then starts the execution. The result is the execution of Nut-in-Nut compiler, now is in an executable form in "a.obj". The compiler reads the source from *stdin*. Suppose we compile the simple example shows at the beginning of this chapter. Suppose it is in the file "t2.txt".

```
c:>nvm < t2.txt

add1
(fun.1.1 (add get.1 lit.1 ))
main
(fun.0.0 (sys.1 (call.75 lit.2 )))

9392 9392
9372 1 16 1 0
9374 1 14 1 9372
9376 1 6 0 9374
9378 0 0 9376 0
9380 1 19 257 9378
9382 1 16 2 0
9384 1 13 9380 9382
9386 0 0 9384 0
9388 1 20 1 9386
```

```
9390 0 0 9388 0
9392 1 19 0 9390
0
3
add1 3 9380 1 1
x 2 1 0 0
main 3 9392 0 0
```

The object code is also outputted to *stdout* [2]. The whole output can be redirect to a file, then select only the code segment to be the object file. Rename the object file to "a.obj". This object file can be executed under nvm.

```
c:>nvm < t2.txt > t2.obj
```

Edit "t2.obj" to eliminate surplus listing at the beginning then run it.

```
c:>ren a.obj nut.obj
c:>copy t2.obj a.obj
```

Rename the previous "a.obj" which is the N-code of Nut-compiler and save it in other name, "nut.obj". Prepare "t2.obj" to be executed under "nvm" by renaming it to "a.obj". Now, run it.

```
c:>nvm
3
c:>
```

This concludes the compilation part. The compiler we discussed so far has no error recovery capability. The error recovery is very important in a practical compiler. It helps programmers to find errors in the program being developed. However, including error recovery will make the compiler itself much more complex. Hence it has been omitted in this presentation.

---

[2] The reason why the object of the sample program started at some what far address is because there is the nut-compiler itself (in N-code) already resided in the memory. The compiler (in N-code) takes around 3600 words; the associated data including the symbol table occupies another 5700 words.)

3.4 Run-time system and the evaluator

The evaluator (function eval) is the machine that executes the internal forms (N-code). The listing of the evaluator in Nut (N-code evaluator) is in the appendix D. The global data is allocated from the data segment when the variable is defined. The local data is dynamic and is allocated from the stack segment. The local data is created when passing the actual parameters to a function and is destroyed when exit from the function. Because the function call has the behaviour of a last-in-first-out queue (LIFO) as the earliest call will exit the last, a stack structure is suitable for allocating the local data for function calls.

Using a stack gains a huge benefit of an automatic reclamation of the memory when the local data is no longer in used. (You may think this is obvious but this is the beauty of it. Think about other alternative way of storing local data such as linked-list. The local data once ceased to exist will have to be reclaimed by some method).

The global and local data can be handled in the same way except that the global data is in the data segment and the local data is in the stack segment.

The evaluation (execution) of a program employs a stack data structure. All variables are accessed through the structure called *activation record*. When a function is evaluated, it has its local environment (local variables and stack area). The activation record is maintained through two global pointers: FP (frame pointer), and SP (stack pointer).

The argument of value-instruction is the index relative to the frame pointer. For example, to get a value of a local variable 3, the instruction "get.3", the access is SS[FP-3] where SS[.] is the memory. Usually this part of memory is called *stack segment*.

Most instructions take their arguments from the evaluation stack. The result (if any) is pushed back to the stack. In this sense, N-code is said to be stack-based instructions. The evaluation stack is local to the current activation record (from FP upward, pointed to by SP).

The instruction "fun.a.v" creates a new activation record; passing arguments from the evaluation stack to this environment, (eval e) where e is the body of function, and deletes the activation record. Two parameters are required to handle creation and deletion of an activation record: *arity* and the *size* of frame.

The encoding is "fun.a.v" where *a* is the arity, *v* is the size of frame. They are used in the deletion of activation record. The value *k* is *v– arity+1*, it is used in the creation of activation record. The operational semantic of "fun.a.v" is discussed in detailed in Chapter 2.

3.5 Run-time supports

To actually run N-code, the simulator provides run-time supports. The memory model is an important factor. The actual memory is provided through the implementation language (C in our case). In general, three parts of memory exist:

- code segment − storing N-code
- data segment − storing static/dynamic data
- stack segment − the run-time stack storing activation record and evaluation stack

The pictorial view of the memory is given below. The memory is M[.] with size MEMEND. The code segment and data segment occupied the memory to the maximum limit MEMMAX. The rest is the stack segment. This is how the nvm (the base simulator written in C) arranges its memory.

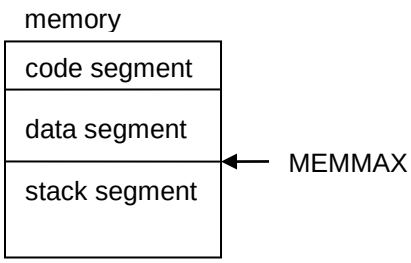


Figure 4.1 The memory layout of the base simulator

How the evaluator arranges its memory?

The loader loads the object into the memory. The N-code starts at the address 2. The data starts at the address 0 and must be relocated to be next to the code

segment. In relocating the data, the instructions that involve global variables: ld, st, ldy, sty, str, must have their arguments offset (this part is done in the function "resolve").

The base simulator (nvm) loads the object of the evaluator (a.obj) first, and then starts executing it. This causes the evaluator to read the object code from *stdin* and evaluating it. The evaluator must relocate its N-code to begin behind a.obj and also its data behind its N-code. To relocate the code, the argument to a call instruction must be offset. To relocate the data, the argument to the instruction accessing globals must be offset. The N-code, data, stack of the evaluator is actually resided in the data segment of the base simulator (nvm). The picture of the memory becomes:

Once the object code is loaded by the evaluator, it starts its execution by allocating its stack segment and sets SP, FP appropriately.

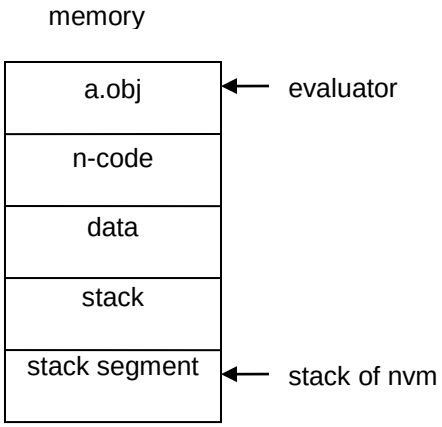


Figure 4.2 The memory layout after loading the evaluator

All registers: FP, SP, have been declared as global variables. The stack segment is allocated. The evaluator initialises them before use.

```
(let tok DP CS M)       ; token, data pointer, code segment, memory
(let SS SP FP)          ; stack, stack pointer, frame pointer

(def init () ()  [eval 23]
   (do
   (set M 0)                    ; base ads, absolute
   (set SS (new STKMAX))        ; allocate stack
   (set FP SS)
   (set SP SS)))
```

3.6 Evaluator

The evaluator is implemented as a separate program. It takes N-code produced from the Nut compiler and runs it. The simulator started by reading the whole input stream (N-code object) into a buffer. Then processes it to instantiate the code segment properly and initialises the simulator variables then begins the evaluation.

```
main   [eval 214]
   readinfile
   loadobj
   initialise
   eval start
```

The main part is the "eval". We will concentrate on this function. The function "loadobj" performs the housekeeping for the relocating the code and data segment to the appropriate address in the memory. It will be discussed later.

The example below is a fragment of program consists of three functions: prints, add1 and main. (sys 1) prints an integer. (sys 2) prints a character.

```
(let tv)
(def prints s ()    ; print string
   (if (vec s 0)
      (do
      (sys 2 (vec s 0))
      (prints (+ s 1)))))

(def add1 x () (+ x 1))
```

```
(def main () ()
   (do
   (set tv 5)
   (prints "string")
   (sys 1 (add1 11))))
```

This is its N-code in a readable form. A constant string is kept in the data segment at the location 2, it is presented in the object code as "str.2".

```
prints
(fun.1.1 (if (ldx.1 lit.0 )(do (sys.2 (ldx.1 lit.0 ))(call.15 (add get.1
lit.1 ))))
add1
(fun.1.1 (add get.1 lit.1 ))
main
(fun.0.0 (do (st.1 lit.5 )(call.15 str.2)(sys.1 (call.17 lit.11 ))))
```

The object code of the above fragment is shown below. It is the input stream of the evaluator.

```
2 1 16 0 0
4 1 17 1 2
6 1 16 0 0
8 1 17 1 6
10 0 0 8 0
12 1 20 2 10
...
66 0 0 64 0
68 0 0 56 66
70 0 0 52 68
72 1 3 0 70
74 0 0 72 0
76 1 19 0 74
```

The "eval" function takes a pointer to an expression and evaluates it. "eval" traverses the expression list, when it finds an atom, it evaluates that atom immediately. When if finds a list, the list is in the form (op arg*), it decomposes the list into its operator (op) and the argument list (e1). The action of evaluation is taken according to the operators. The main part of "eval" is this multi-way branch to do each operation.

```
eval e   [eval 160]
  if e is nil return nil
  get the operator and its arg-list, e1
  decode op arg
  switch op
    ADD
      v = (eval arg1 e1) + (eval arg2 e1)
    IF
      if (eval arg1 e1) != 0
        v = eval arg2 e1
      else
        v = eval arg3 e1
    CALL
      eval all arg and push them to eval stack
      v = eval the function
    LIT
      v = arg
    GET
      v = SS[fp-arg]
    ...
  else
    error "unknown op"
return v
```

To limit the scope of discussion, we will discuss in details only the subset of the instruction which is suitable to run the following example only (about 11 instructions). These instructions are {fun, if, ldx, lit, do, sys, call, add, get, st}. The value-op are {add, get, lit, str, ldx}. The control-op are {do, if, fun, call}. The other instructions are {sys, st}. Their actions are very like the description of their operational semantic discussed in Chapter 2.

We start with the straightforward value-op.

```
(if (= op xADD)   [eval 181]
   (set v (+ (eval (arg1 e1)) (eval (arg2 e1))))
```

arg1, arg2, arg3 are the access functions to get the first, second and third argument from the argument list. The variable v is the value returned by the function "eval". The operator "add" evaluates two arguments and adds them.

The operator "get" gets a value of a local variable from the activation record.

```
(if (= op xGET)  [eval 194]
   (set v (vec M (- FP arg)))
```

"lit" and "str" have the same effect. Their difference in the operation code is used to distinguish two operators. The loader must relocate everything stored in the data segment. "str" has its argument as a pointer to a constant string stored in the data segment, therefore its argument must be identified and relocate at the load time.

```
(if (= op xLIT)  [eval 190]
   (set v arg)

(if (= op xSTR)  [eval 192]
   (set v arg)
```

"ldx" takes the base address from the argument list, and takes the index from its argument. The effective address is calculated as base + M[FP-arg]. The value is taken from the data segment. The data segment is just a location in the memory.

```
(if (= op xLDX)  [eval 202]
   (do
   (set idx (eval (arg1 e1)))
   (set v (vec M (+ (vec M (- FP arg)) idx))))
```

The control-op alters the sequence of evaluation.

```
(if (= op xDO)  [eval 176]
   (while e1
      (do
      (set v (eval (head e1)))
      (set e1 (tail e1))))
```

Where (head e) is the first argument of e, (tail e) is the rest of e without (head e). "do" evaluates all the elements in the list. "if" evaluates the condition and selects one of the alternate actions.

```
(if (= op xIF)  [eval 171]
     (if (eval (arg1 e1))
       (set v (eval (arg2 e1)))
       ; else
       (set v (eval (arg3 e1))))
```

"call" evaluates all of its arguments and their value in the stack before evaluating the function.

```
(if (= op xCALL)  [eval 183]
     (do
     (while e1                    ; eval all arg
       (do                        ; and push it to stack
       (push (eval (head e1)))
       (set e1 (tail e1))))
     (set v (eval arg)))          ; eval function
```

And here is how to push a value to the evaluation stack. The evaluation stack is an array of memory pointed to by SP.

```
; push a value to the evaluation stack
(def push e ()  [eval 124]
   (do
   (set SP (+ SP 1))
   (if (> SP (+ SS STKMAX))
      (error "stack overflow"))
   (setv M SP e)))
```

When evaluating a function, the "fun" performs a complicate task of creating a new activation record, setting a new SP, evaluating the body of function, then restore old activation record and SP. The passing of parameters is through the evaluation stack where the new activation record just happens to overlap these variables. There is no copying of passing parameters; they are arranged such that the overlap occurred properly. The numbering of the variables must be ordered in the reversed order of their appearance. This is done during the final compilation phase.

```
(if (= op xFUN)  [eval 145]
   (do
   (set v (& arg 255))          ; decode a, v
   (set a (>> arg 8))
   (set k (+ (- v a) 1))
   (setv M (+ SP k) FP)         ; save old FP
   (set FP (+ SP k))            ; new frame
   (set SP FP)
   (set v (eval (arg1 e)))      ; eval body
   (set SP (- (- FP v) 1))      ; delete frame
   (set FP (vec M FP)))         ; restore FP
```

"st" gets the value of local variable and stores it to the memory. The address is the first argument of argument list.

```
(if (= op xST)  [eval 198]
   (do
   (set v (eval (arg1 e1)))
   (setv M arg v))
```

The last instruction is special. It performs the input/output which are dependent on the underlying physical system. It looks strange that xSYS calls to (sys 1) and (sys 2), but this is absolutely correct because these two functions implement the correct actions; to print integer and character to a display.

```
(if (= op xSYS)   [eval 208]
   (do
   (set v NIL)
   (set a (eval (arg1 e1)))
   (if (= arg 1) (sys 1 a)
   (if (= arg 2) (sys 2 a)
   ; else
   (error "undef sys"))))
```

The rest of the instructions are evaluated similarly. This will conclude our presentation of the main eval.

Now we turn our attention to the housekeeping task, the relocation of the code segment. To relocate the code, the argument to call instruction must be offset. To relocate the data, the argument to the instruction accessing globals must be offset. Assuming the object codes have been read and the following variables have been instantiated: ads, type, op, arg, next. The following code fragment creates a new node for this code and does the relocation.

```
(set CS (sys 9))  [eval 80]              ; find start of code segment
(set DP (+ (+ CS end) 2))                ;  start of data segment
(if (= type 1)
   (set a (reName op arg))
   ; else dot-pair
   (set a (shift (+ (<< op 24) arg) CS)))
(set a2 (new 2))                         ; create a new node
(sethead a2 a)
(settail a2 (shift next CS))             ; reloc the next
```

where CS is the start of our code segment (not the N-code of eval itself!), DS is the start of data segment, "sethead" and "settail" update the head and next cells, "shift" performs the offset calculation. The renaming of an atom is done in "reName". "mkAtom" creates an atom from op and arg.

```
; relocate arg of an op
(def reName (op arg) ()   [eval 62]
   (do
   (if (= op xCALL)
      (set arg (shift arg CS))
   (if (or (= op xLD) (= op xST))
      (set arg (shift arg DP))
   (if (or (= op xLDY) (= op STY))
      (set arg (shift arg DP))
   (if (= op xSTR)
      (set arg (shift arg DP))))))
   (mkATOM op arg)))

; offset a by disp, code segment started at 2
(def shift (a disp) ()  [eval 56]
   (if (> a 0)
      (- (+ a disp) 2)
      0))
```

Please bear in mind that the evaluator runs on top of the base simulator "nvm". The N-code of the evaluator itself is loaded as "a.obj" by the "nvm" then the evaluator starts by reading the *stdin* stream which must be the object code to be evaluated.

Understanding "eval" let us confirm the meaning of each N-code instruction. In fact, the "eval" itself can be regarded as the specification of the operational semantic of N-code.

3.7 Lab session

Use the Nut compiler to compile a program, "quick.txt" (a quicksort program). First, compile the Nut-compiler into "a.obj". Then run the compiler with nvm to compile "quick.txt" (the source program). The output is put into a file and edits it to be "quick.obj" (an N-code object). Run "quick.obj" using nvm to see the result. First, compile the compiler.

```
C:\test>nutc < nut.txt
```

Then, use the compiler to compile "quick.txt".

```
C:\test>nvm < quick.txt > q.obj
```

Edit "q.obj" and put it to the file "quick.obj". Copy "quick.obj" to "a.obj". It will be the input of nvm. The file "quick.obj" looks like this.

```
516 516
2 1 14 1 0
4 1 20 1 2
6 0 0 4 0
8 1 19 257 6
10 1 14 1 0
12 1 20 2 10
...
510 0 0 474 508
512 1 3 0 510
514 0 0 512 0
516 1 19 1 514
```

```
0

27
print 3 12952 1 1
...
swap 3 13072 3 4
partition 3 13238 3 7
flag 2 7 0 0
quicksort 3 13298 3 4
q 2 4 0 0
inita 3 13352 2 3
s 2 2 0 0
show 3 13410 2 3
main 3 13460 0 1
```

Run the N-code object of the quicksort program using nvm.

```
C:\test>copy quick.obj a.obj
C:\test>nvm
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

C:\test>
```

3.8 Further reading

The intermediate code is a widely known technique used almost in all compilers. It separates the task of compilation into two major phases; the first phase is to parse the source language into this intermediate code and the second phase is to generate the target language (usually machine codes of the target machine) from this intermediate code. This helps to simplify the compiler and also separating the target dependent part from the compiler. This separation is useful when there are many target machines. The first phase can remain the same, only the code generator needs to be done for each new machine.

The evaluator or the virtual machine is one of the most important ideas in computer science. The *emulation* of other machine is a powerful concept. It has been the major cause of the success of computer industry both in terms of producing different hardware that can use the same executable software and in terms of software that can be run on different platform virtually unchanged. The hardware example is the IBM S360 family [PAD81] that can emulate many early computers of the same company to such a degree that the customers bought the new machines to run their existing software unchanged. The software example is the JVM [VEN98] which is the virtual machine that is available on almost any machines. The use of intermediate code for the purpose of porting a compiler to a new machine is popularised by P-code [BUR78] [WIR91]. Early Pascal language was compiled into P-code. This made Pascal compiler to be rapidly available throughout the microcomputer communities because the task of creating an executable Pascal compiler was reduced to porting the P-code evaluator. The latest software emulation can be seen from the Apple computer company where their latest computers use a different processor than any of their previous product but the company can made a large number of their existing software available under this new processor in a short time using the emulation.

The technique of writing an evaluator in its own language is called *meta interpreter* [STR88] or more specifically *meta circular interpreter*. It was in practice in early days of computing, the example can be drawn from LISP. The interpreter of LISP was usually written in LISP [MCA65]. The meta interpreter is also useful to reason about the meaning of program and its correctness.

References

[AHO86] Aho, A., Sethi, R., and Ullman, J., Compilers: Principles, techniques, and tools. Addison-Wesley, 1986.

[BER78] Berry, R., "Experience with the Pascal-P compiler", Software – Practice and Experience, 8:617-627, 1978.

[CHO05] Chongstitvatana, P., "Self-Generating Systems: How a $10{,}000{,}000_2$-line Compiler Assembles Itself," Proc. of National Computer Science and Engineering Conference, Bangkok, 2005.

[PAD81] Padegs, A., "System/360 and beyond", IBM Journal of research and development, September 1981.

[STR88] Sterling, L., "Constructing Meta-interpreters for Logic Programs", in Advanced School on Foundation of Logic Programming, Italy, 1988 .

[VEN98] Venners, B. Inside the Java Virtual Machine, McGraw Hill, 1998.

[WIR81] Wirth, N. "Pascal-S: a subset and its implementation", in Pascal – The language and its implementation, Barron, D. (ed.), pp. 199-260, Wiley, 1981.

Excercises

3.1 The Nut-compiler has not been finished. There are no "let", "enum" and string. Extend Nut-compiler (nut.txt) to include them. You can have a look at "nut3-compiler.txt" for a guide, or you can look at the C source in "nut31/compile" directory, nut.c.

3.2 Complete the Nut-in-Nut compiler. Use Nut completion kit. Write additional functions so that Nut compiler is able to handle the full Nut language.

3.3 Extend the Nut-in-Nut compiler to include "let" and "enum".

3.4 Nut-compiler does not have the operator: mul, div. Add it to the compiler and simulator (simulator is optional) (Hint: at compiler, you should look at the following functions:

```
#define xMUL 8  in nut.h
add reserved word to keynames[]  in nut.c
prAtom()  in data.c
```

3.5 Strings in Nut can be more efficient by packing 4 characters into one word. Do it.

3.6 The symbol table uses a sequential search [nut 82]. It is not efficient. Implement a more efficient method for searching the symbol table. (Hint: a hash table is a standard way to handle a symbol table. It has a constant running time for searching.)

3.7 How many symbols are there in the symbol table when we compile the Nut-compiler?

3.8 The N-code object can be made *relocatable*, i.e. not dependent on the absolute location in the memory. This can be achieved by *linearising* the N-code tree. The simplest form is the *prefix* form. See the following example:

Source (+ 2 3) becomes readable N-code: (+ lit.2 lit.3) which is stored in the memory (say starts at 6).

```
2 1 16 3 0
4 1 16 2 2
6 1 6 0 4
```

This object is not relocatable, it embeds the absolute location in the "*next*" link. A list can be represented by prefixing it with its length, no "*next*" link is necessary.

(+ lit.2 lit.3)

becomes

3 + lit.2 lit.3

another example:

(+ lit.2 (+ lit.3 lit.4))

becomes

3 + lit.2 3 + lit.3 lit.4

This representation can be converted into an N-code tree (at any location).

Write a program to output N-code object in linear form to a file and read it back into the memory properly at a different location. You can use "prList" to print the readable N-code out to check it.

3.9 Study eval-in-nut (eval.txt) and try to add some missing operators to it.

3.10 In the main loop of Nut-evaluator [eval 160], the evaluator spent most of its time is checking the opcode and performs the operation accordingly. It uses the form of (if (= op xxx) …). This is a sequential test. Suggest a way to improve the efficiency of the main evaluator loop.

Chapter 4

Code Generation

To actually run a program on a real machine, the intermediate code must be translated into machine codes of that machine. To generate machine codes, the instruction set of the target machine must be studied. We will study processors in details in Chapter 5. There will be two illustrative processors. The first one has instructions of the type zero-address, so called a stack-based instruction set. The second one will be a more conventional three-address instruction set. It is easier to translate N-code to a stack-based instruction set. Therefore we will study the code generator for this instruction set. However, the code generation for three-address instruction set will also be discussed. We shall begin with the discussion of the target instruction set.

4.1 S-code

The instruction set for our stack-based processor is called S-code. The processor itself is named Sx processor. S-code is designed for simplicity; the emphasis is on a small number of instructions. It is also quite fast to be interpreted by a software virtual machine. From S-code, it is easy to generate machine dependent code for a specific purpose, such as, small code size (byte-code, nibble-code) [KOT03], high performance (extended code) [CHO97], or to fit a particular hardware. In our system, S-code is the machine code of Sx processor which has been designed to execute S-code directly in hardware.

S-code has a fixed-length 32-bit instruction format. It is not compact but it is reasonably fast when interpreting. This format simplifies the code address calculation and allows code and data segment to be the same size (integer) as opposed to other format such as the byte-coded instruction format (as in JVM [LIN97]). There are two types of instructions: zero-argument and one-argument. The zero-argument instructions are mostly related to the arithmetic and logic operations. The one-argument instructions are the access operations to variables

and the control-flow operations. The description of the instruction set is as follows.

Notation

n is a 24-bit constant (2-complement)
x is a 32-bit value
v is a variable reference, for a global variable, it is an index to code segment, for a local variable, it is an offset to a current activation record in stack segment.
f is a reference to *CS*.
DS[] is the data segment, *SS[]* is the stack segment.
pc is a program counter, pointed to the current instruction.
stack notation: (before -- after)

Zero argument instructions

add, sub, mul, div, mod	are integer arithmetic, take two operands from the stack and push the result back. (a b -- a op b)
shl, shr	take two operands: number, no-of-bit and shift the number and push the result back. shr is an arithmetic shift, preserved sign.
band, bor, bxor, eq, lt, le, ge, gt	are logical, take two operands from the stack and push (T/1, F/0) back. (a b -- 0/1)
bnot	is bit inverse, takes one operand and push the result back. (a -- ~a)
ldx	takes an address ads, an index idx, and returns DS[ads+idx]. (ads idx -- DS[ads+idx])
stx	takes an address ads, an index idx, a value x, and store x to DS[ads+idx]. (ads idx x --)
case	takes a value (key), compares it to the range of label, goto the matched label, or goto else/exit if the key is out of range. (key --)
array	allocate x words in Data segment, return ref v to the allocated data. (x -- v)

One argument instructions

lit n	push n (-- n)
inc v	increment local variable, SS[fp+v]++
dec v	decrement local variable, SS[fp+v]--
ld v	push DS[v]. (-- DS[v])
st v	takes a value x and store to DS[v] = x. (x --)
get v	get local variable v. (-- SS[fp+v])
put v	store a value x to local variable v. (x --)
call f	new activation record, goto f in CS
ret n	returns from a function call, n is the size of activation record. remove the current activation record. Return a value if function returns a value.
fun n	function header, n is the number of local variables
jmp n	jump to pc+n in CS
jt n	jump pc+n if top of stack = 1, pop
jf n	jump pc+n if top of stack = 0, pop
sys n	call a system function n, for interfacing to external functions, the arguments are in the stack, the number of arguments can vary.

4.2 S-code format

Each instruction is 32-bit. The right-most 8-bit is the operational code. The left-most 24-bit is an optional argument. For a virtual machine, this format allows simple opcode extraction by bitwise-and with a mask without shifting, but it needs 8-bit right-shift to extract an argument. Because zero-argument instructions are used more frequent, this format is fast for decoding an instruction. However, a decoder in hardware can tap any bit freely, therefore any format will be equally fast to decode.

Encoding

```
0 --      1  add    2  sub     3  mul     4  div
5  band   6  bor    7  bxor    8  not     9  eq
10 ne     11 lt     12 le      13 ge      14 gt
15 shl    16 shr    17 mod     18 ldx     19 stx
20 ret    21 --     22 array   23 <end>   24 get
25 put    26 ld     27 st      28 jmp     29 jt
30 jf     31 lit    32 call    33 --      34 inc
35 dec    36 sys    37 case    38 fun
```

The "*end*" is a pseudo instruction. It does not existed in a real processor. It is used to stop the processor simulation. S-code supports high-level function call directly similar to N-code. The run-time data structure must be understood. An activation record stored a computation state. It is resided in the stack segment. The computation state consists of: PC (return address), FP, all local variables. SP needs not be stored as it will be recovered properly when return. The necessary information, the size of the activation record, is stored as the argument of "*ret*" instruction. The following diagram shows the layout of an activation record in the stack segment (notice that it is exactly the same as the activation record of N-code).

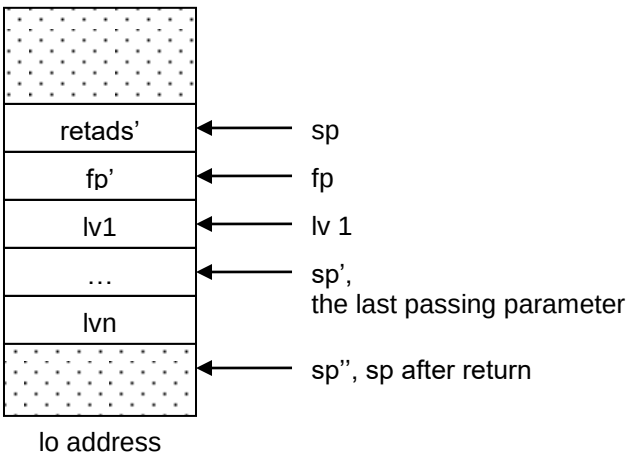


Figure 4.1 The activation record to support S-code

A function call creates a new activation record. The new FP is *SP + k*. The value k is the argument of "`fun k`", *k = n - arity + 1*. The new activation record overlaps the evaluation stack such that the passing parameters become the local variables of the new activation record. A local variable is indexed by an offset from the current FP. The numbering of the local variables causes the first passing parameter to be the *n-th* local variable and so on. This fact is handled by the code generator. A function call does the following:

1 decode *k* at function header
2 create new activation record, save old FP
3 set new SP
4 save return address
5 goto body of function

When returning, "`ret m`", *m* is size of the activation record + 1. When restoring SP (not considering the return value yet):

```
sp'' = fp - m
```

A return does the following:

1 restore PC
2 if there is a return value
3 restore SP and FP
4 push the return value
5 else
6 restore SP and FP

"`case`" is a multiway branch instruction. It requires a jump-table. The layout of code in "`case`" is as follows:

```
case
lit low
lit hi
jmp else
jump table
...
code of each case
```

A *case* does:

```
1  extract range of label: low, high
2  if key < low or key > high
3    PC = PC + 3    goto else-case
4  else
5    PC = PC+key-low+4  goto matched label
```

In this implementation, the jump-table is filled with the labels in the range (from low to high), hence, finding the matched label is simply an index calculation, a constant time operation. This enables the *case* instruction to be fast but it consumes the memory in the code segment as large as the range of label. This is wasteful if the label is not dense. For the case of sparse label, a binary search can be used. The jump-table is the sorted label of the pair (*label, goto code*). This is not implemented as it is not suitable to be converted into a machine specific instruction (maps to a real processor). Because Nut language does not yet support multiway branch, the "*case*" instruction is not implemented by Sx processor.

Input of the code generator is an N-code object. Output is the S-code object. Let's study some examples of programs in S-code. Let *a*, *b*, *c* be locals; *d*, *e* be globals; *L*, *M* be labels.

```
a = a + 1

  get a, lit 1, add, put a

a = b[i]

  get b, get i, ldx, put a

d[i] = b

  ld d, get i, get b, stx

e = add2(a,b)

  get a, get b, call add2, st e

if (a == 1) then b = 2 else b = 3
```

```
   get a, lit 1, eq, jf L,
   lit 2, put b, jmp M,
L: lit 3, put b,
M:
```

Let give one example what the code generator do. The source program in Nut,

```
(def add1 x () (+ x 1))

(def main () ()
  (sys 1 (add1 22)))
```

is compiled into N-code object,

```
add1
(fun.1.1 (+ get.1 lit.1 ))
main
(fun.0.0 (sys.1 (call.80 lit.22 )))

22 22
2 1 16 1 0
4 1 14 1 2
6 1 6 0 4
8 0 0 6 0
10 1 19 257 8
12 1 16 22 0
14 1 13 10 12
16 0 0 14 0
18 1 20 1 16
20 0 0 18 0
22 1 19 0 20
0
```

The S-code generator takes this N-code object and outputs S-code object. The format of S-code object will be discussed later.

```
5678920
1 12
2080 23 294 280 287 1 532 294
5663 800 292 276
1000 999
```

It means the following:

```
 1 Call 8
 2 End
 3 Fun 1
 4 Get 1
 5 Lit 1
 6 Add
 7 Ret 2
 8 Fun 1
 9 Lit 22
10 Call 3
11 Sys 1
12 Ret 1
```

The N-code and S-code are quite similar as they are both stack-based instruction sets. The mapping between N-code and S-code is simple (see Table 4.1). Only the control-op must be transformed to jump. To distinguish between two instruction sets the N-code is prefixed with "*x*" and S-code with "*ic*".

4.3 How the code generator works?

In the last chapter, the evaluator, "eval", evaluates the N-code and returns the result. The evaluator performs its task by traversing the N-code tree and applies the operators to their arguments. The code generator follows the same pattern. It uses a variant of "eval". In other words, the generator reads the input N-code object and traverses the N-code. Instead of executing it by applying the operators to their arguments, the generator outputs the corresponding S-code. The mapping between N-code and S-code is simple. Most of the code is one-to-one mapping. However, the *addresses* of N-code and S-code are different. This is handled using the *associative* list of N-code address to S-code address. The only instruction that need to relocate its argument is "*call*" using "insertLab" and "assoc".

Table 4.1 Mapping between N-code and S-code

n-code	***s-code***
xLIT.a	*icLit.a*
xGET.a	*icGet.a*
xPUT.a	*icPut.a*
xLD.a	*icLd.a*
(xADD e1 e2	*e1 e2 icAdd*
(xST.a e)	*e icSt.a*
(xLDX.a e)	*e icGet.a icLdx*
(xSTX.a e v)	*e v icGet.a icStx*
(xLDY.a e)	*e icLd.a icLdx*
(xSTY.a e v)	*e v icLd.a icStx*
(xFUN.a.v e)	*icFun.k e icRet.m* *where k = v-a+1, g = v+1*
(xCALL.a e...)	*e ... icCall.a*
(xIF e1 e2 e3)	*e1* *icJf F* *e2* *icJmp E* *F: e3* *E:*
(xWHILE e1 e2)	*L: e1* *icJf E* *e2* *icJmp L* *E:* or better *icJmp I* *L: e2* *I: e1* *icJt L*
(xDO e1 e2 ...)	*e1 e2*

Let look at the "eval" for code generator. "out" outputs an S-code. The whole S-code is stored in an array, XS[.]. XP is the current S-code address. The listing of the code generator is presented in the appendix E.

```
eval e [gen 224]                ; S-code generator

 ...                             ; e1 is the argument list
 switch op

 ADD
   eval head e1
   eval arg2 e1
   out icAdd
 LIT
   out icLit arg
 GET
   out icGet arg
 FUN
   insertLab ads XP              ; ads is N-code address
                                 ; XP is S-code address
   lv = arg & 255
   arity = arg >> 8              ; decode a.v
   out icFun (lv-arity+1)
   eval head e1
   out icRet (lv+1)
 CALL
   while e1 not empty            ; generate all arguments
      eval head e1
      e1 = tail e1
   out icCall (assoc arg)        ; map address to S-code
 ...
 else
   error "unknown op"
```

For the control-op, the iteration is achieved by the jump instructions. The first one, "*do*", just generates the S-code one-by-one corresponding to the elements in the argument list of N-code (*e1*).

```
DO  [gen 239]
  while e1 not empty
    eval head e1
    e1 = tail e1
```

The "*if*" generates the testing for the conditional and the alternatives. The first jump, "*icJf*", jumps over the true-alternative (to label *F*). The second jump is the jump at the end to exit (label *E*).

The pattern for code generation is:

```
(xIF e1 e2 e3)

  e1
  icJf F
  e2
  icJmp E
F: e3
E:
```

This is how the generator works. The variable *ads* is used to mark the place where the offset of the jump will be updated. All jumps in S-code are relative. Their displacements are calculated relative to the current address (XP).

```
IF   [gen 186]                ; e1 = (cond true false)
  eval head e1                 ; gen cond
  out icJf 0                   ; <1>
  ads = XP - 1                 ; mark S-code ads
  eval arg2 e1                 ; gen true
  if (arg3 e1) = NIL
    patch ads (XP-ads)         ; patch jf at <1>
  else
    out icJmp 0                ; <2>
    patch ads (XP-ads)
    ads = XP - 1               ; mark S-code ads
    eval arg3 e1               ; gen false
    patch ads (XP-ads)         ; patch jmp at <2>
```

There are two ways to generate code for the while expression. The first one is straightforward.

```
(xWHILE e1 e2)

L: e1
   icJf E
   e2
   icJmp L
E:
```

The code is generated in order of the appearance of the arguments, *e1* then *e2*. However, each time around the loop there will be two jumps. To improve the quality a bit, we can turn around the order and use the conditional to perform the loop back.

```
   icJmp I
L: e2
I: e1
   icJt L
```

The first jump jumps into the conditional. Only the first time around the loop that requires two jumps; the subsequent iteration requires only one jump.

```
WHILE                    ; e1 = (cond body)
  out icJmp 0
  ads = XP - 1           ; mark the loop back address
  eval arg2 e1           ; gen body
  patch ads (XP-ads)     ; jump into cond
  eval head e            ; gen cond
  out icJt (XP-ads+1)    ; loop back
```

Here are the actual nut code to generate S-code for the "*if*" and "*while*" control-op.

```
; e = (cond true false)
(def genif e (ads e3)  [gen 186]
   (do
   (eval (head e))                ; gen cond
   (outa icJf 0)
   (set ads (- XP 1))
```

```
    (eval (arg2 e))                 ; gen if-true
    (set e3 (arg3 e))
    (if (= e3 NIL)
        (patch ads (- XP ads))
        (do                         ; else
        (outa icJmp 0)
        (patch ads (- XP ads))
        (set ads (- XP 1))
        (eval e3)                   ; gen else
    (patch ads (- XP ads))))))

(def genwhile e ads  [gen 204]
    (do
    (outa icJmp 0)
    (set ads (- XP 1))
    (eval (arg2 e))                 ; gen body
    (patch ads (- XP ads))
    (eval (head e))                 ; gen cond
    (outa icJt (- (+ ads 1) XP))))

; change arg, preserve op
(def patch (ads v) ()   [gen 167]
    (setv XS ads (+ (<< v 8) (& (vec XS ads) 255))))
```

The associative list has two operations: insert-label and get the associated address of the label. atab is the array storing the tuple {*label*, *address*} where *label* is the N-code address, *address* is the S-code address. numLab is the number of tuples stored in the associative table.

```
; n1 is the label, n2 is the address
(def insertLab (n1 n2) (i)  [gen 135]
    (do
    (set i (+ (* numLab esize) 2))      ; start at 2
    (setv atab i n1)
    (setv atab (+ i 1) n2)
    (set numLab (+ numLab 1))
    (if (> numLab MAXLAB)
        (error “label table full”))))
```

```
; search assoc for n1
; if found, return adddress, else 0
(def assoc n1 (i flag end)   [gen 121]
   (do
   (set i 2)                          ; start at 2
   (set flag 1)
   (set end (+ (* esize numLab) 2))
   (while (and flag (< i end))
      (if (= (vec atab i) n1)         ; sequential search
         (set flag 0)
         ; else
         (set i (+ i esize))))
   (if flag
      0                               ; not found
      (vec atab (+ i 1)))))           ; found, return n2
```

The output S-code must be of the correct form so that the processor simulator can read it properly. Here is the format of the S-code object file.

```
magic
start end     (end inclusive)
code*         (code segment)
start end
data*         (data segment)
```

Where *magic* = 5678920, it is used to distinguish the object code between N-code and S-code. *start*, *end* are the addresses denoting the starting and ending addresses of the block of data that follow. Take a look at the previous example of the S-code object.

```
5678920
1 12
2080 23 294 280 287 1 532 294
5663 800 292 276
1000 999
```

5678920 denotes that this is the S-code object. 1 12 are the starting and ending addresses of the code block. The length of the code is 12. 2080..276 are the codes. 1000 999 denote the starting and ending addresses of the data block. There is no data block in this example (the ending address is smaller than the starting address).

4.4 Three-address code generation

S-code is very similar to N-code, they are both stack-based. It is easy and very straightforward to translate N-code to S-code. However, there is no modern processor that has stack-based instruction set. We now turn our attention to another more conventional instruction set, a three-address instruction set. The processor that has this instruction set, S2 is a register-based processor. As the subsequent components of our system will be based on stack-based instructions, we will only discuss a general scheme of code generation for three-address instruction set. To begin, we discuss the overview of the processor and the instruction set.

S2 is a simple 32-bit processor for educational purpose. It exists in a simulator, although some implementation at Hardware Description Language for S2 exists. S2 is developed from S1 [CHO01], a simple 16-bit processor used for teaching several classes in the past 10 years. S2 has an adequate instruction set to demonstrate the high level language and the assembly language relationships. Comparing to a real processor (such as Intel Pentium [INT01]), S2 lacks OS supporting functions, I/O and interrupts, and performance enhancing features (such as MMX [PEL97]).

S2 description

S2 has 32 registers, *r0...r31*, *r0* is special and always has a zero value. S2 has 32-bit address space, it can access 4G words of memory. Addressing is in word (32-bit) unit. S2 has no byte-access instruction. All instructions are 32-bit long (fixed length, one size). S2 has flags that indicate result of previous operations. Flags are: Z zero, S sign, C carry, O overflow/underflow.

S2 addressing mode

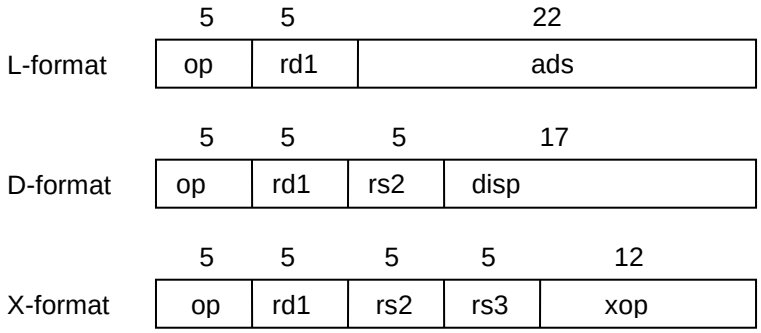
S2 has four addressing modes: absolute, displacement, index, and immediate. The *absolute* mode has 22-bit range (0..4M). The *displacement* mode uses one register and a 17-bit value (0..128K). The *index* mode employs two registers.

Lastly, the *immediate* mode uses the literal value in the instruction. Depend on what instruction the literal is 22-bit (load/store) or 17-bit (arithmetic). For example, to load a value from memory into a register, all four addressing modes are as follows:

Absolute	`ld r1,ads`	`R[r1] = M[ads]`
Immediate	`ld r1,#n`	`R[r1] = n`
Displacement	`ld r1,d(r2)`	`R[r1] = M[d + R[r2]]`
Index	`ld r1,(r2+r3)`	`R[r1] = M[R[r2] + R[r3]]`

The opcode format and assembly language format for S2 follow the tradition `dest = source1 op source2` from PDP [BEL76], VAX [LEV89] and IBM S360 [AMD64].

S2 instruction format



(rd dest, rs source, ads and disp are sign extended)

Figure 4.2 S2 instruction format

Opcode encoding

The S2 instruction set, its encoding and its format is shown in Table 4.2 and Table 4.3.

Table 4.2 S2 opcode encoding and format. (1) jump condition uses r1 as condition, the coding in r1 field: 0 always, 1 eq, 2 neq, 3 lt, 4 le, 5 ge, 6 gt (2) extended instruction. The code 14..30 are undefined.

Opcode	Op	Mode	Format
0	ld	absolute	L
1	ld	displacement	D
2	ld	immediate	L
3	st	absolute	D
4	st	displacement	L
5	jmp (1)	absolute	L
6	jal	absolute	L
7	add	immediate	D
8	sub	immediate	D
9	mul	immediate	D
10	div	Immediate	D
11	and	Immediate	D
12	or	immediate	D
13	xor	immediate	D
31	xop (2)		

Meaning

The meaning of each instruction is as follows. We use the following notation to describe the instruction; "*op dest src1 src2*". *R0* always returns the value 0.

```
ld r1,ads          R[r1] = M[ads]
ld r1,#n           R[r1] = n
ld r1,d(r2)        R[r1] = M[ d + R[r2] ]
ld r1,(r2+r3)      R[r1] = M[ R[r2] + R[r3] ]
st ads,r1          M[ads] = R[r1]
st d(r2),r1        M[ d + R[r2] ] = R[r1]
st (r2+r3),r1      M[ R[r2] + R[r3] ] = R[r1]
jmp cond,ads       if cond true PC = ads
jal r1,ads         R[r1] = PC; PC=ads; jump and link
jr  r1             PC = R[r1]; return from subroutine
```

Table 4.3 S2 instruction encoding for xop. (1) use r1. (2) use r1 as the number of trap function. The code 13..4095 are undefined.

Xop	Op	Mode	Format
0	add	register	X
1	sub	register	X
2	mul	register	X
3	div	register	X
4	and	register	X
5	or	register	X
6	xor	register	X
7	shl	register	X
8	shr	register	X
9	ld	index	X
10	st	register	X
11	jr (1)	special	X
12	trap (2)	special	X

The arithmetic operations are two-complement integer arithmetic.

```
add r1,r2,r3      R[r1] = R[r2] + R[r3]
add r1,r2,#n      R[r1] = R[r2] + sign extended n
```

The instruction add, sub affect Z, C − C indicates carry (*add*) or borrow (*sub*). The instruction mul, div affect Z, O − O indicates overflow (*mul*) or underflow (*div*) and divide by zero.

The logical operations are bitwise operations. They affect Z, S bits.

```
and r1,r2,r3      R[r1] = R[r2] bitand R[r3]
and r1,r2,#n      R[r1] = R[r2] bitand sign
                  extended n
or xor  . . .
shl r1,r2         R[r1] = R[r2]  shift left one bit
shr r1,r2         R[r1] = R[r2]  shift right one bit
```

As *r0* always is zero, many instructions can be synthesis using *r0*.

```
or r1,r2,r0          move r1 <- r2
or r1,r0,r0          clear r1
sub r0,r1,r2         compare r1 r2  affects flags
```

To complement a register, *xor* with *0xFFFFFFFF* (-1) can be used.

```
xor r1,r2,#-1        r1 = complement r2
```

How an expression be transformed into sequence of instructions

An instruction in machine language composed of an operator and operands. The number of operands varies from zero (stack instruction), one, two, and three. Our hypothetical S2 processor is a 3-operand machine. Each instruction has the form "*op r1 r2 r3*" and all operands are registers. Having three operands means each operation can take two inputs from two operands and stores the result in the third operand. It is suitable for binary operations such as; *add*, *sub* etc. To translate an expression into S2 instruction, each result of a binary operation needs to store in a temporary register.

```
a * b + c - d

t1 = a * b
t2 = t1 + c
t3 = t2 - d
```

The input variables (*a*, *b*, *c*, *d*) and the temporary variables will be assigned registers. Let *r1 = a*, *r2 = b*, *r3 = c*, *r4 = d*, *r5 = t1*, *r6 = t2*, *r7 = t3*. The above expression can be written in S2 instructions as follows:

```
mul r5 r1 r2
add r6 r5 r3
sub r7 r6 r4
```

In fact, at most two temporary variables are needed for any arbitrary long sequence of binary operations as the temporary value can be accumulated using

just one register and another register is used to hold one operand of the binary operator.

Let use only *t1*,

```
t1 = a * b
t1 = t1 + c
t1 = t1 - d
```

For any expression that has parentheses to control the order of evaluation, the expression can be transformed into postfix ordering:

```
(a * b) + (c * d)
```

This expression is transformed into:

```
t1 = a * b
t2 = c * d
t1 = t1 + t2

((a * b) + (c * d)) / f)

t1 = a * b
t2 = c * d
t1 = t1 + t2
t1 = t1 / f
```

Registers can be regarded as local variables. To access global variables, "*ld*" "*st*" is used with their associated addressing mode to transfer values to and from global memory to registers. Then, the arithmetic-logic operation can be performed on registers.

Access simple scalar

Let *A*, *B*, *C*, *D* be global variables, the expression

```
A = B + C - D
```

can be translated into the following sequence.

Let *r1 = A*, *r2 = B*, *r3 = C*, *r4 = D*

```
ld r2 B
ld r3 C
ld r4 D
add r2 r2 r3
sub r1 r2 r4
st A r1
```

Access an array

Let *ax[]* be an array, the expression

```
i = 2
b = ax[i]
ax[i+2] = c
```

can be translated into the following sequence.

Let *r1 = i*, *r2 = b*, *r3* be the base address of *ax*, *r4 = c*, *r5 = temp*.

```
ld r1 #2
ld r2 (r3+r1)
add r5 r1 #2
st (r3+r5) r4
```

The displacement-addressing mode is used to access data structure, where the offset to the field is known at compile-time. For example, the "head" function accesses the first cell and "tail" function accesses the second cell. These function definitions are found in Nut-compiler.

```
(def head e () (vec e 0))
(def tail e () (vec e 1))
```

They are translated into the following sequence. Let *r1* be the input expression, *r2* be the return value.

```
head:  ld r2 0(r1)
tail:  ld r2 1(r1)
```

Using jump for conditional branching

```
      jmp cond ads
```

cond = eq, neq, lt, le, gt, ge, always

There are four flags in the processor: Sign, Zero, Carry, Overflow (S,Z,C,O). Each flag is one bit. They are like global variables. Flags are set by ALU instructions such as, *add, sub, mul, div, and, xor* etc. The *ld/st* instructions do not change flags. The condition is decided by flags. Flags are set by the previous ALU instruction. To compare two variables, subtract instruction is used and flags S, Z will be affected. Let two variables be in *r1* and *r2*, the instruction "*sub r0 r1 r2*" will compare these variables and sets the Sign and Zero flags without altering any register (because *r0* is always zero). For example *eq* is Z = 1; *lt* is S = 1; *le* is S = 1 or Z = 1. Subsequently, the jump instruction can test the flags that affect the control flow.

Using jump to do if-then-else

(if (> a b) e1 e2)

Let *r1 = a*, *r2 = b*,

```
            ld r1 a
            ld r2 b
            sub r0 r1 r2  ; compare a b
            jmp gt L1     ; if a > b then
            <code of e2>
            jmp always exit
      L1:
            <code of e1>
      exit:
```

Generate code for a simple while loop

The following expression can be translated to S2 code as follows.

```
(do
(set s 0)
(set i 1)
(while (<= i 10)
   (do
   (set s (+ s i))
   (set i (+ i 1))))
```

Let *r1 = s*, *r2 = i*

```
        ld r1 #0       ; s = 0
        ld r2 #1
loop:
        sub r0 r2 #10
        jmp gt exit    ; while i <= 10
        add r1 r1 r2   ; s = s + i
        add r2 r2 #1   ; i = i + 1
        jmp always loop
exit:
```

Function call

The part of program that is reused is made into a subroutine. When a main program calls a subroutine, the body of that subroutine is executed and then the flow goes back to the caller at the location after the line that call that subroutine. This transfer of flow requires saving of the program counter (PC) which at the time of call pointed to the next instruction. The return from a subroutine call requires restoring PC. There are two instructions for implementing a subroutine call: *jump and link*, and *jump register*.

```
jal rx ads
```

"*jump and link*" saves PC to *rx* and jump to *ads*.

```
jr rx
```

"*jump register*" restores PC from *rx*.

The register "*rx*" is called a link register. It stores the return address. It is complicate when the call is recursive because the link register must then be saved/restored properly. Here is a simple call. For simplicity, the parameters can be passed through registers.

```
(def sq (x) () (* x x))

(def main () (a b)
   (set a 2)
   (set b (sq a)))
```

The above program can be translated into S2 code as follows.

Let *r1 = a*, *r2 = b*, *r3 = x*, *r4 = link*, *r5* be return value.

```
main:
      ld r1 #2
      add r3 r1 r0      ; binding a, x
      jal r4 sq         ; call sq
      add r2 r5 r0      ; b = sq(a)
      <end>
sq:
      mul r5 r3 r3
      jr r4             ; return
```

Please note that we use "*add rx ry r0*" to do *rx = ry* (moving a value between two registers). *r4* is used as a link register to store the return address. The passing of a parameter is done by assigning *x = a*, ("*add r3 r1 r0*"). The return value is stored in *r5*.

To pass parameters from "caller" to "callee", we generate the code to transfer variables using the evaluation stack. Any registers that will be used by a subroutine must be saved upon entry into that subroutine and must be restored upon exiting it. This is called *callee-save*. The subroutine takes responsible in saving and restoring link register and all registers local to it in order not to interfere with values of the caller. An alternative is to use *caller-save* where the caller must save/restore its own registers. Let *sp* be a register that is the stack pointer.

To push a register "*x*",

```
add sp sp #1
st 0(sp) x
```

To pop a value to a register "*x*",

```
ld x 0(sp)
sub sp sp #1
```

When multiple values are pushed into a stack, the compiler can use displacement to give an offset to the stack pointer. The stack pointer can be adjusted at the end of the sequence. For example, to push three registers.

```
st 1(sp) first
st 2(sp) second
st 3(sp) third
add sp sp #3
```

And similarly for popping multiple values.

```
ld third 0(sp)
ld second -1(sp)
ld first -2(sp)
sub sp sp #3
```

Please note that the offset of *sp* started from 1 when push and 0 when pop due to asymmetric of two operations in terms of the initial position of *sp*, and the order of operands are reversed.

Now let us do the previous example of the function call again with the code for manipulating the activation record fully expanded.

```
(def sq (x) () (* x x))

(def main () (a b)
   (set a 2)
   (set b (sq a)))
```

This program can be translated into S2 code as follows.

Let *r1 = a*, *r2 = b*, *r30 = link*, *r31* be the return value.

```
main:
      ld r1 #2
      add sp sp #1       ; pass a on eval stack
      st 0(sp) r1
      jal r30 sq
      add r2 r31 r0      ; b = sq(a)
      <end>

sq:                      ; let r1 = x
      <save reg>
      <pass param>
      mul r31 r1 r1
      <ret>
      jr r30
```

Where *<save reg>* is the code to save the registers used in this subroutine. First, the link register is pushed, then other registers.

```
  st 1(sp) r30
  st 2(sp) r1
  add sp sp #2
```

<pass param> is the code to pass parameters to the local registers. The passed parameters are on the evaluation stack before *<save reg>*. The size of *<save reg>* is used as an offset to access the passed parameters.

```
  ld r1 -2(sp)
```

At the end of subroutine *<ret>*, the saved registers are restored and the passed parameters are popped from the evaluation stack.

```
ld r1 0(sp)
ld r30 -1(sp)
sub sp sp #3
```

The subroutine "*sq*" is shown in full below.

```
sq:                     ; let r1 = x
      st 1(sp) r30      ; <save reg>
      st 2(sp) r1
      add sp sp #2
      ld r1 -2(sp)      ; <pass param>
      mul r31 r1 r1
      ld r1 0(sp)       ; <ret>
      ld r30 -1(sp)
      sub sp sp #3
      jr r30
```

4.5 Lab session

Compile some Nut programs to get the object files then generate S-code from these object files.

Compile the Nut-compiler:

```
c:>nutc < nut.txt
```

And use nut-compiler to compile the example, let it be "*t3.txt*", the output goes to "*t3.obj*":

```
c:>nvm < t3.txt > t3.obj
```

Edit *t3.obj* to get rid of the listing at the beginning. Now compile the S-code generator, "*gen.txt*":

```
c:>nutc < gen.txt
```

Use it to generate the final S-code object:

```
c:>nvm < t3.obj > t3s.obj
```

We can use the S-code virtual machine, *svm.exe* to run it and to generate a readable S-code. To generate a readable S-code from an S-code object:

```
c:>svm -l < t3s.obj
```

Run it.

```
c:>svm < t3s.obj
```

4.6 Summary

The code generation from N-code to S-code is straightforward. Both instruction sets are similar. They are based on stack, zero-address instructions. We have described the plan how to map from one code to another. The format of the target object code has been studied. The mechanism to generate the object code is elaborated. The general framework to generate the object code is similar to executing the N-code using the evaluator of the last chapter, "eval". The code generator traverses the N-code and outputs the associated S-code. The control-flow instruction of N-code is realised using the jump instruction of S-code. Therefore the tree-structure of N-code has been transformed to a linear sequence of S-code instructions.

To illustrate the method of generating object code for a conventional processor, a register-based processor, S2, is demonstrated. One important aspect of generating code for a register-based instruction set is that of register allocation. The result of an operation must be placed explicitly into a register, unlike stack-based instruction where the result is placed on the evaluation stack. We have not touched the subject of code optimisation where the output code can be improved in terms of speed of execution or the size of the code. This is not the main concern for our study. Many textbooks on compiler are the excellent source [AHO86] [LOU97]. However, in terms of performance of a system as a whole, we will study it in Chapter 9.

References

[AHO86] Aho, A., Sethi, R., Ullman, J., Compiler: Principles, Techniques, and Tools, Addison Wesley, 1986.

[AMD64] Amdahl, G., Blaauw, G., and Brooks, F., "Architecture of the IBM System/360", IBM Journal of Research and Development, April 1964.

[BEL76] Bell, C., and Strecker, W., "Computer structures: What we have learned from the PDP-11", Proc. of 3rd annual symposium on computer architecture, (1976): 1-14.

[CHO97] Chongstitvatana, P., "Post processing optimization of byte-code instructions by extension of its virtual machine", Conf. of Electrical Engineering, Bangkok, 1997.

[CHO01] Chongstitvatana, P. "Computer Architecture: A synthesis approach", 2001.

[INT01] Intel Corp. Intel Pentium 4 processor optimization reference manual, Document 248966-04. Aurora, CO, 2001.

[KOT03] Kotrajaras, V., and Chongstitvatana, P., "Nibbling Java Byte Code for Resource-Critical Devices", Proc. of National Computer Science and Engineering Conference, Thailand, 2003.

[LEV89] Levy, H. and Eckhouse, R., Computer programming and architecture: the VAX, 2nd ed., Digital press, 1989.

[LIN97] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Addison Wesley, 1997.

[LOU97] Louden, K., Compiler Construction: Principles and Practice, PWS Pub., 1997.

[PEL97] Peleg, A., Wilkie, S, and Weiser, U., "Intel MMX for Multimedia PCs", Communications of the ACM, January 1997.

Exercises

4.1 Modify the code generator to generate the two-jump while. Measure the number of instruction used while running a program. Compare it with the one-jump while.

4.2 Implement the code generator that use the instruction `inc` and `dec` by recognising the following N-code:

```
(put.a (ADD get.a lit.1))
(put.a (SUB get.a lit.1))
```

4.3 The associative table is a linear array. The searching is sequential. Reimplement the associative table to be more efficient. (Hint: use other data structure, or use hash table).

4.4 Write a code generator for S2 instruction set using the scheme outlined in this chapter.

4.5 There are both advantage and disadvantage of using *callee-save* versus *caller-save*. Some compiler does both depending on the context (the C compiler for VAX under the operating system VMS). Modify the code generator above to do *caller-save* where the caller must save/restore its own registers.

4.6 Suggest some way to implement a simple code optimisation to improve the speed of execution. (Hint: replace a long sequence of code with a shorter one).

Chapter 5

Microprogramming

A processor is composed of a data path and a control unit. A data path of a processor consists of execution units, such as an ALU, a shifter, registers, and their interconnects. A control unit is considered to be the most complex part of a processor. Its function is to control various units in the data path. The control unit realises the behaviour of a processor as specified by its micro-operations. The performance of a control unit is crucial as it determines the clock cycle of the processor.

A control unit can be implemented in either hardwired or microprogram. A hardwired control unit is a large FSM (finite state machine) sending control signals to a data path. A microprogrammed control unit [FLY71] is a complex programmable unit that outputs control signals to a data path according to its *microprogram.* A microprogrammed control unit can be regarded as a simple computer. In this view, a processor has another simple processor inside it which is its control unit. Controlling a data path is represented by its microprogram. We will discuss microprogramming concept in details in this chapter. It will be used in the next chapter to design the main processor used in our system.

5.1 Hardwired control unit

In the past, a hardwired control unit was very difficult to design hence its engineering cost was very high. Presently, the emphasis of computer design is the performance therefore hardwired is the choice. Also the CAD tools for logic design have improved to the point that a complex design can be mostly automated. Therefore almost all processors of today use hardwired control units.

Starting with a behavioural description of the control unit, a state diagram of micro-operations is constructed. Most states are simply driven by clock and only transition to the next state. Some states branches to different states depend on conditions such as testing conditional codes or decoding the opcode.

From the state diagram, a hardware realization can be constructed almost automatically by some CAD tools. Explanation of logic design for sequential circuits and logic minimization can be consulted from many basic textbooks on the subject such as Katz [KAT96]. The control circuit is implemented using Programmable Logic Array (PLA). In general, any sequential circuit (which can implement any state machine) can be constructed from a combinational circuit with feedback. The feedback signals represent the states. If the feedback path uses no clock, the circuit is called *asynchronous*. If the feedback path uses a latch with clock, the circuit is called *synchronous*. Synchronous circuits are used almost exclusively for sequential circuits today as they are easier to design and can be implemented reliably. Most of the CAD tools handle synchronous circuits. Asynchronous circuit has been used for the reason of performance as in many early computer designs, for example, ILLIAC and many computers in the class called supercomputer. But it is difficult to implement reliably and it is still much more difficult to do systematic design of a complex machine using asynchronous circuits. The combinational part of the control circuit can be regarded as a memory where its content is the map of the inputs to the outputs (states are considered to be a part of the outputs). This view of combination circuit as a memory is called Random Access Memory model (RAM) of computation machines.

5.2 Microprogrammed control unit

Maurice Wilkes invented microprogram in 1953 [WIL53] [WIL85]. He realised an idea that made a control unit easier to design and is more flexible. His idea is that a control unit can be implemented as a memory which contains patterns of the control bits and part of the flow control for sequencing those patterns. A microprogram control unit is actually like a miniature computer which can be programmed to sequence the patterns of control bits. Its “program” is called microprogram to distinguish it from an ordinary computer program. Using microprogram, a control unit can be implemented for a complex instruction set which is impossible to do by hardwired.

How microprogram work

The simplest way to understand a microprogrammed control unit is to regard it as a ROM which implements a finite state machine (FSM). A general sequential circuit consists of a combinational circuit which some outputs are fed back to inputs. If the feedback part is synchronised with a clock using a register to latch the signal, then the circuit is said to be *synchronous*. That is, the output changes at the edge of the clock, between the clocks, the output does not change. If the output is fed back directly, it is said to be *asynchronous*, the output changes dependent on the delay of the combinational part and it can change ("oscillate") a number of times before the signal is settled (or never settled). This is the most general way to visualise a sequential circuit. Any combinational circuit can be directly implemented as a truth table, although it is not efficient in terms of size. A read-only-memory (ROM) stored the truth table. In a sense, this ROM can be regarded as a "program".

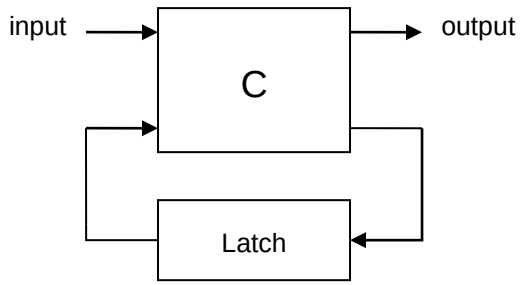


Figure 5.1 A general sequential circuit

A control unit outputs the control signals to control various parts of the data path such as selecting a multiplexer, latching a register, or control the operation of an ALU. The first stage to see how a ROM can be used as a control unit is that the ROM can output a fixed sequence of control signals simply by cycling the address of the ROM. The content of this ROM is a microprogram, but it is comparable to a *straight line* program, i.e. the one without any transfer of control. This is how the inventor of microprogram, Maurice Wilkes, discovered it. We will call each entry in this ROM a *microword*.

A microprogram counter is used to cycling the sequence of control. If a part of ROM, says a number of the right most bits, is used as the next address, then these bits can be loaded into the microprogram counter to alter the sequence of control. We now have a microprogram with "goto". Some control bits can be used to test the condition, such as the zero flag. The result of testing can be used to decide whether to load the microprogram counter or to increment it. This is equivalent to the statement "if-then". Changing the behaviour of the control unit can be done simply by changing this microprogram.

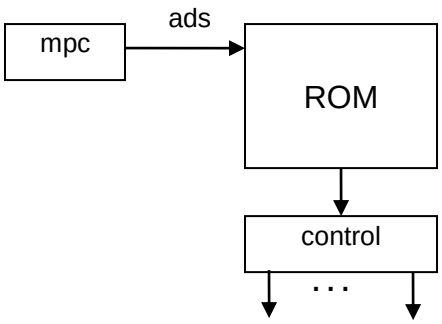


Figure 5.2 A fixed sequence control unit using ROM

Conditionals are the bits that are used to determine the flow of microprogram; loop, branching, next instruction etc. Its input comes from the data path (usually from the conditional code register). The next address determines the next microword to be executed.

A microprogram is executed as follows.

1. A microword at the location specified by the microprogram counter is read out; control bits are latched at an output buffer which is connected to the data path.
2. If the conditional field is specified and the test for conditional is true, the next address of microprogram will come from the next address field otherwise the microprogram counter will be incremented (execute the next microword).

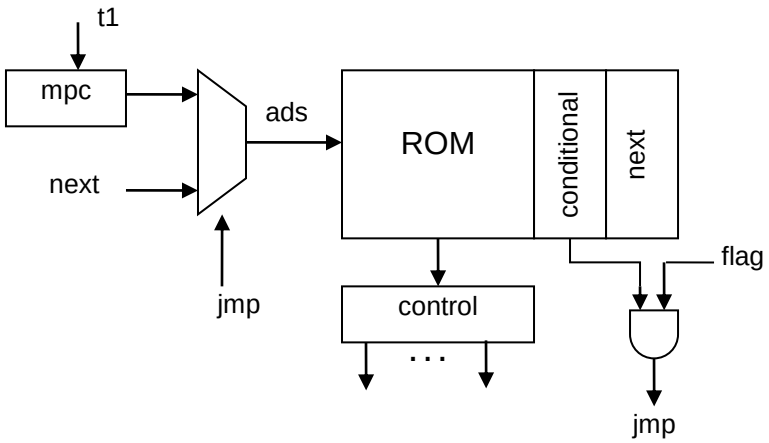


Figure 5.3 A fully function microprogrammed control unit

What that has been described is called *horizontal microprogram*. The microword can have other formats. There are several possibilities (Fig. 5.4):

1. Single format, one address as just described above.
2. Single format, two addresses, contain two next addresses field, one for result of test true, and the other for result of test false.
3. Multiple formats, such as, one format for the control bits without the next address field and another format for *jump on condition* with the address field. The advantage is that the microword can be shorter than the single format. The disadvantage is that to "jump" will take one extra cycle.

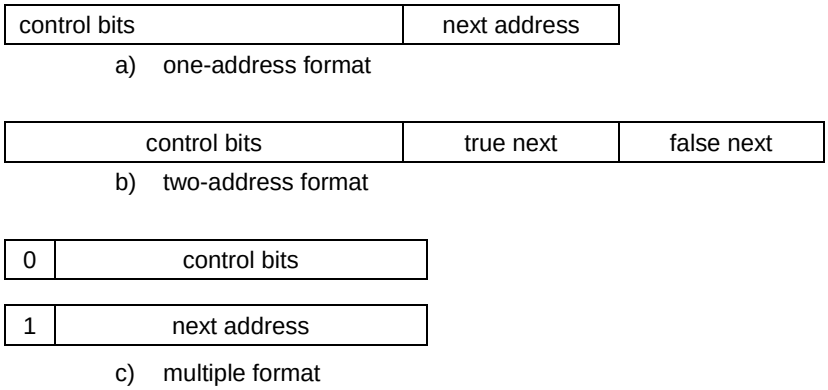


Figure 5.4 Several formats of microword

Horizontal microprogram allows each control bit to be independent from other therefore enables maximum simultaneous events and also offers great flexibility. It is also waste a lot bit. For each field of a microword, there may be a group of bits that are not activated at the same time therefore they can be *encoded* to use a fewer bit. A decoder is required to *decode* these bits and to connect them to the data path. This approach is called *vertical microprogram*. There are many possibilities to compact the micro memory to be as small as possible, sometime trading off speed for space, for example, a *two-level microprogram*. The first level is *vertical* i.e. maximally encoded; the microword of the level one is pointed to the *horizontal word* of the second level. This is rather like the first level is composed entirely from *subroutine call* and the second level is the subroutine.

The control unit just described has the output of control unit directly mapped to control signals. It is possible to have more than one output format that map the output of the control unit to the actual control signals, for example using the first bit to choose the format, 1 to mean the control bit, 0 to mean the test and jump to other microprogram address. This is called *two-format microprogram*. It is the effort to reduce the size of microprogram because control and test can be mutually exclusive. Other variation is *nano-program* where the group of microprogram word is regarded as a reusable subroutine then the "program" becomes the code to call these subroutines. You can read about the historical record of microprogram era of architecture in [SIE82].

5.3 Realisation of microprogrammed systems

This section discusses the equivalence of hardware and software in realising a sequential system. This concept will be illustrated by a simple example of designing a 4-bit comparator in both hardwired and microprogrammed systems (this example is due to [MAN92]).

An assembly of logic elements, whether combinational (AND, OR, NOT, NAND gates, demultiplexors, multiplexor etc) or sequential (flip-flops, registers etc.) is called a *hardwired logic*. By incorporating memories and the content of memory in the test or assignment elements, the system is called a *microprogrammed logic system*; the content is the "microprogram". A microprogrammed system can be used to realise a synchronous sequential system, that is, it can be used to implement a control unit.

Example A 4-bit comparator input : A0, A1, B0, B1, and Z is { EQ, LT, GT } can be described with the logic expression of Z as follows.

Z = (A1' B1' A0 B0' + A1 B1' + A1 B1 A0 B0') . GT + (A1' B1' A0' B0' + A1 'B1' A0 B0 + A1 B1 A0' B0' + A1 B1 A0 B0) . EQ + (A1' B1' A0' B0' + A1' B1 + A1 B1 A0' B0) . LT

Where A' is NOT A

The expression can be tabulated in the table below.

number	A1	B1	A0	B0	Z
0	0	0	0	0	EQ
1	0	0	0	1	LT
2	0	0	1	0	GT
3	0	0	1	1	EQ
4..7	0	1	X	X	LT
8..11	1	0	X	X	GT
12	1	1	0	0	EQ
13	1	1	0	1	LT
14	1	1	1	0	GT
15	1	1	1	1	EQ

This expression can be represented as a diagram of *test* and *assignment* primitives that is traversed sequentially by using synchronous sequential system which each clock reads an element of the diagram and executes the primitive.

Fig. 5.5 shows the diagram of the comparator,

Z = compare(A,B)

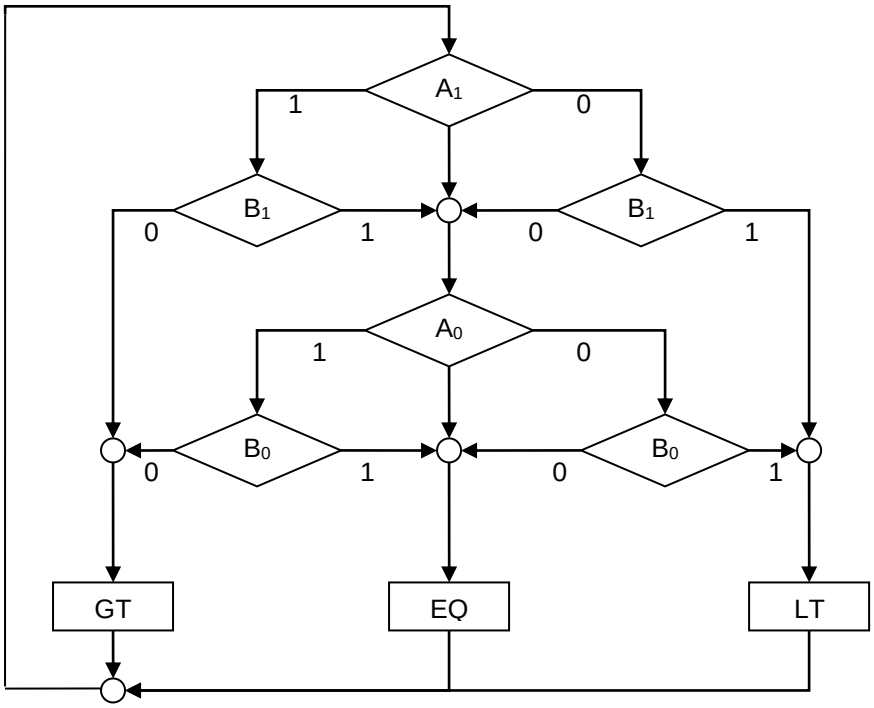


Figure 5.5 The diagram of compare(A,B)

Each primitive (*test*, *assignment*) can be described as follows.

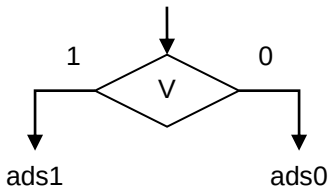


Figure 5.6 The test element

test

if V is true then goto ads1 else goto ads0

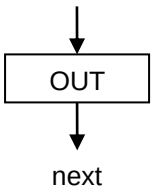


Figure 5.7 The assignment element

assignment

output OUT and goto next

The above diagram can be translated into a microprogram as follows.

```
0     if A1 then goto 1 else goto 2
1     if B1 then goto 3 else goto 6
2     if B1 then goto 8 else goto 3
4     if A0 then goto 4 else goto 5
5     if B0 then goto 7 else goto 6
6     R = GT goto 0
7     R = EQ goto 0
8     R = LT goto 0
```

Next, the microprogram is encoded to map the primitives to a concrete representation. The 4 cases of test inputs {A1 B1 A0 B0} are encoded into 2 bits. The output { EQ LT GT} is encoded into 3 bits using unary code.

input	i1	i0
A1	0	0
B1	0	1
A0	1	0
B0	1	1

output	z2	z1	z0
GT	1	0	0
EQ	0	1	0
LT	0	0	1

The microword has two types: *test*, *assignment*. The address field has 4 bits to cover the whole microprogram address (0 . . 8)

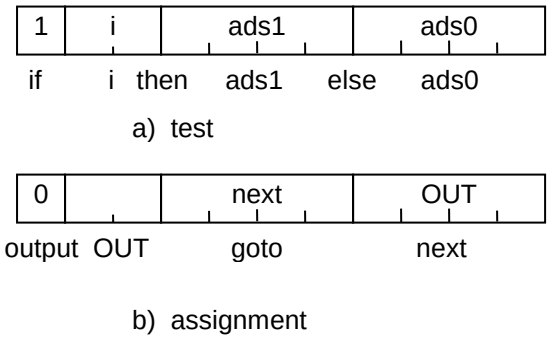


Figure 5.8 The microword format for compare

The microprogram then can be written as follows.

ads	**T**	**i1 i0**	**ads1, next**	**ads0, z**
0000	1	0 0	0001	0010
0001	1	0 1	0011	0110
0010	1	0 1	1000	0011
0011	1	1 0	0100	0101
0100	1	1 1	0111	0110
0101	1	1 1	1000	0111
0110	0	- -	0000	-100
0111	0	- -	0000	-010
1000	0	- -	0000	-001

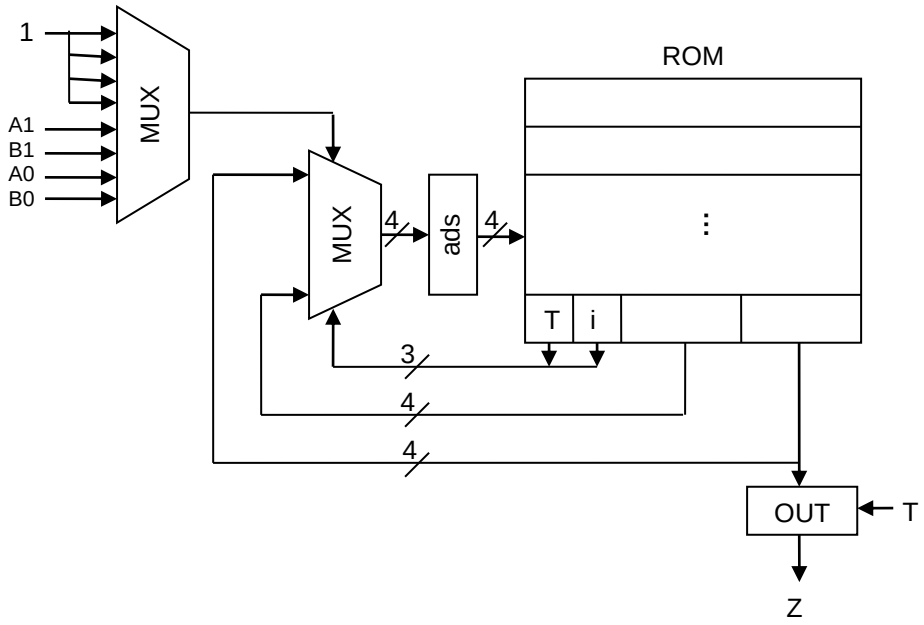


Figure 5.9 The microprogrammed unit to realise the function compare

The microprogrammed unit to realise the function compare is shown in Fig. 5.9. How many cycles it takes to evaluate compare(A, B)? Observing the diagram (Fig. 5.5), on the longest path, there are 5 steps to traverse the diagram hence it takes 5 cycles to evaluate this function using the microprogrammed unit above.

5.4 Equivalence of hardware and software

The definition of microprogramming is due to Wilkes, who in 1953 suggested a method for designing the control unit of a processor, based on the use of sequence of microwords – a microprogram – held in a read only memory (ROM). In this context, microprogramming is generally understood as the technique of producing interpreters for high-level language.

At that time random access memory (RAM) that was available was much slower than the processor, leads to CISC (Complex Instruction Set Computer) to achieve high speed the microprogram of CISC are organised horizontally; the need to control a complex processing unit requires each microword to consist of a large number of bits, often over 100.

Firmware, specification of a microprogram, is not an interpretation algorithm but a logic system. The concept of vertically organised microprogram follows that each microword is of fewer bits than in horizontally organised microprogram. The resulting simplicity enables a true optimization of the software to be achieved. Firmware is the transformation and equivalence between hardware (logic systems) and software (microprogram). This hardware-software equivalence is a particular case of the equivalence between space and time.

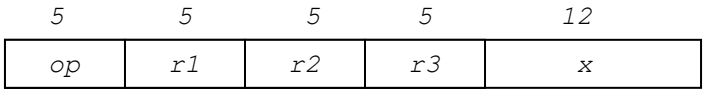
5.5 Microprogram for a simple data path

Next we discuss an example of writing a microprogram to control a simple data path. This data path is not a fully functioned processor. However, most of the essential components are present. The example of the control will be the action of fetching an instruction from the memory and executing it.

Data path specification

Data path has a 32-bit ALU. The data width is also 32-bit. It has 32 registers. The register bank has one write port, two read ports. ALU has one function, *add*. The program counter (PC) is 32-bit. It has an increment operation to increment PC. The memory interface unit has two registers: MAR (memory address register), MBR (memory buffer register). The instruction register (IR) stores the most recent instruction.

The memory contains a code segment storing a program (in machine code). Each instruction is a fixed length 32-bit wide and has the following format, *op* 5-bit, *r1* 5-bit, *r2* 5-bit, *r3* 5-bit, don't care 12-bit.



The field "*op*" specifies operation of the instruction. It has only one operation "*add*" with the code *00001*.

```
ADD  00001
```

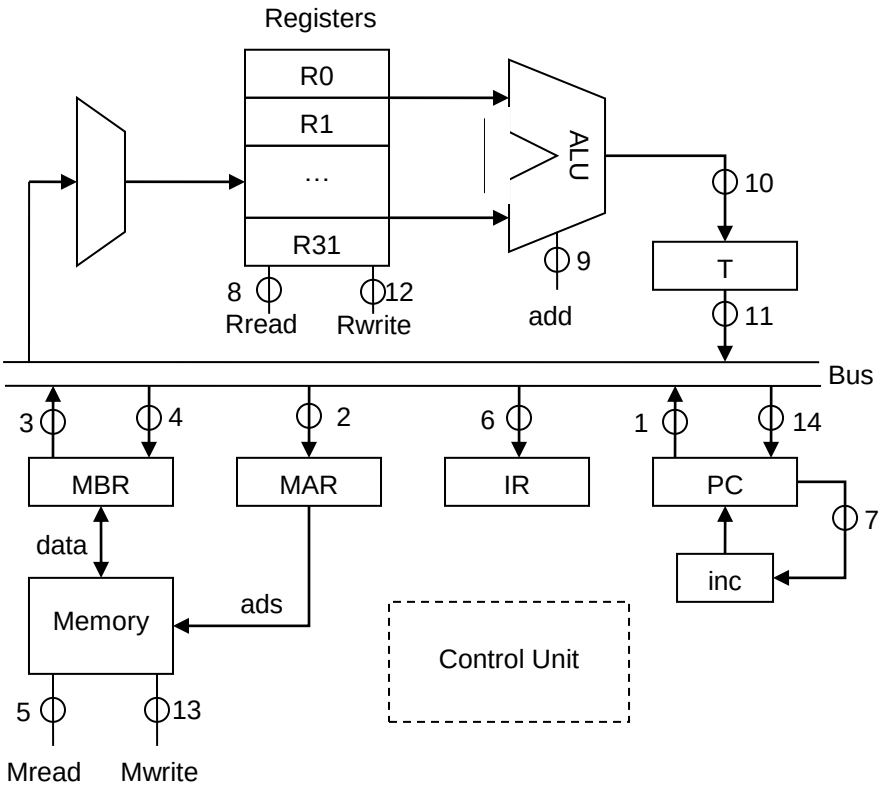


Figure 5.10 A simple data path

r1, r2, r3 specifies a register number, 32 registers requires 5 bits

Example

```
add r1 r2 r3
```

means *R[r2]+R[r3] -> R[r1]*

```
00001 00001 00010 00011 x...x
```

The behaviour of the data path can be described in a register transfer level (RTL). The register transfer level specifies how data is flowed between components in the data path. Let's call this description, *microsteps*.

notation

```
<label>
source->destination
```

microsteps

```
<fetch>
  PC -> MAR
  Mread -> MBR
  MBR -> IR
  PC + 1
  decode (goto add)

<add>
  R[r2] + R[r3] -> T
  T -> R[r1]
  goto fetch
```

Each step takes one cycle. Some step can be overlapped if they are independent, for example, PC + 1 can be activated concurrently of any step in *<fetch>*.

Now we want to explore in more details how to realise this behaviour (the microstep). Each connection has an ON/OFF gate (valve) associate with it. The data flow can occur when the gate is ON. To transfer data from a source through bus to a destination, two gates are opened, one is the gate from source to bus, another one is the gate from bus to destination, for example, to do

```
PC -> MAR
```

Two gates are: *PC>BUS* and *BUS>MAR*. The first gate transfers data from *PC* to *BUS*. The second gate transfers data from *BUS* to *MAR*. Here is the list of all gates in the example.

notation

```
gate number, gate name

1 PC>BUS
2 BUS>MAR
3 MBR>BUS
4 BUS>MBR
5 Mread
6 BUS>IR
7 PC+1
8 Rread
9 ALU:add
10 ALU>T
11 T>BUS
12 Rwrite
13 Mwrite
14 BUS>PC
```

Each microstep can be written as the state of these gates. We use the convention that when a gate's name is written it is activated (or ON) otherwise it is idles (or OFF).

```
<fetch>
  PC>BUS, BUS>MAR
  Mread
  MBR>BUS, BUS>IR
  PC+1

<add>
  Rread, ALU:add, ALU>T
  T>BUS, Rwrite
```

These events can be written using gate numbers as follows.

```
<fetch>
  1,2
  5
  3,6
  7
```

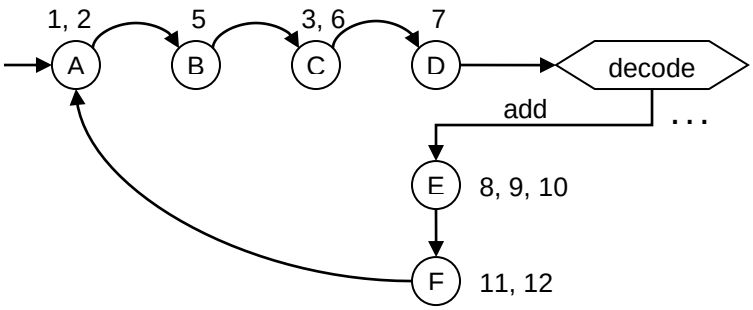
```
<add>
  8,9,10
  11,12
```

A finite state machine is used to realise the control unit. A FSM for this control unit is shown below. The "*decode*" is a combinational circuit to do a multiway branch to an appropriate execution state according to the opcode-field on an instruction in IR (in this case only one instruction "*add*").

notation: *state {activation; next state}*

```
A {1,2; B}
B {5; C}
C {3,6; D}
D {7; decode}

E {8,9,10; F}
F {11,12; A}
```



This FSM can be implemented using various technologies. The design and implementation of a FSM is greatly simplified by the use of CAD tools.

5.6 How complicate is a control unit?

The bound of complexity of control is

$$States \times Control\ inputs \times Control\ outputs.$$

A control unit is implemented as a sequential synchronous machine. It can be described as

$$O = f(I)$$

O output is a function of *I* input, *f* is a Boolean function which is purely combinational. To have state feedback, a part of output is stored in a memory to be fed back to input. This memory is synchronised with a clock so that changes at its output (the state) happen at the edge of clock. Between edges of a clock, its output does not change.

$$Oc = f(Ic,Is)$$
$$Is(t+1) = Os(t)$$

Where *Oc* is the output, *Os* is the state output, *Ic* is the input, *Is* is the state input, *t* is time.

The size of a combination circuit with *I* input and *O* output is $O\,2^{I}$. This is the size required to store the output as a function of inputs.

The control unit in the example has 15 outputs, 5 inputs from opcode-field of IR and 6 states (fetch+add). *Os* must be 3 bits to contain 6 states.

Oc is 15
Ic is 5
Is is 3

So its size is $(15+3)2^{(5+3)}$, approximately 5000 bits (4608). The table that contains *f* and the state feedback can be viewed as a kind of program.

$$O = f(I)$$

```
address   O = f(I)   state
I       00110101.... 001
...     10001010.... 010
...     ...
```

The height of this table is 2^I, 256 in our example. The width of each row in this table is 15+3 bits. The content of this table is regarded as a program. We can write it down as the event of activation of gates

```
fetch {1,2; B}
B {5; C}
C {3,6; D}
D {7; decode}
add {8,9,10; F}
F {11,12; fetch}
```

This is microprogram, voila!

5.7 Advantage and disadvantage of microprogram

Advantage

Making change to a hardwired control unit implies global change, that is, the circuit will be almost totally changed. Hence, it is costly and time consuming although the present CAD tools do reduce most of the burden in this area. In contrary, for a microprogrammed control unit, making change to it is just changing the microprogram, the bit pattern in the micromemory. There are tools to generate these bits from a human-readable microprogram, hence making change to a microprogram is similar to edit-compile a program. The circuit for control unit does not change. This enables adding new instructions, modifies addressing mode, or updating the version of control behaviour easy to do.

Disadvantage

Microprogram relies on a fast micromemory. It requires a high speed memory. In fact, the architect of an early microprogrammed machine, IBM S360 family, depended on this crucial technology, which was still in the development at that time. The breakthrough in memory technology came, and S360 became the most

successful family of computers. A hardwired control unit is much faster. Microprogramming is inherently very low level, making it hard to be absolutely correct. Microprogramming is by nature concurrent, many events occur at the same time, so it is difficult to develop and debug (for a good reading that told a story related to this process, read Tracy Kidder's “The soul of a new machine” [KID00]).

5.8 Summary

Microprogram describes the step-by-step execution of the control unit. For our study, this has advantage of being able to change the behaviour of the control unit without changing the circuits. The operational meaning of each instruction can be described using microprogram. By considering steps of control as a program, it creates a new level of abstraction which simplifies the implementation of a complex instruction set. Using this abstraction, building a control unit is similar to writing a program describing the behaviour of an instruction set. We studied various forms of microprogram and their realisation. A systematic method to realise a general microprogrammed system is illustrated. An example of writing a microprogram for a simple data path is demonstrated.

The technology of microprogram has a big impact on building and designing computers during the era of 1970-1980. Many computer manufacturers adopted this technology. The modern computer aided design tools and the development of modern computer architecture have replaced it. However, microprogramming is an excellent tool to understand computer system behaviour. We will use it to study the processor in the next chapter.

5.9 Further Reading

Maurice Wilkes is the inventor of microprogram. His idea was too far ahead of its time as it required a high speed memory which was not possible at that time. Microprogram approach for control unit has several advantages:

1. One computer model can be microprogrammed to *emulate* other model.
2. One instruction set can be used throughout different models of hardware.
3. One hardware can realised many instruction sets. Therefore it is possible to choose the set that is most suitable for an application.

Microprogram becomes obsolete mainly because the present design emphasises the performance and microprogram is slower than hardwired. The change in instruction set design toward a minimum number of cycle per instruction simplifies the instruction set to the point that microprogram is not really required. Also the design of hardwired control unit can be mostly automated as opposed to microprogram which must be written and debug. Hence, for the current instruction set architecture, a hardwired control unit offers a lower engineering cost.

As the history tells us, the design of microprocessors followed the same trend as the earlier computer design. Because of the resource limit (the number of transistor in a chip), hardwired control was implemented and the instruction set architecture was toward a simple design. Ease of change popularised microprogramming. Microprogram made it possible to achieve more complex instruction set. With a much larger micro memory a machine as elaborate as the VAX is possible [LEV89]. In 1984, DEC wanted to offer a cheaper machine with the same instruction set as VAX. They reduced the instructions interpreted by microcode by trapping some instructions and performing them in software. They discovered that 20% of VAX instructions occupied 60% of the microcode, and yet they are used (executed) only 0.2% of the time. Their simpler subset of VAX, called MicroVAX-1, implemented 80% of VAX instruction in microcode, other 20% is trapped to software, has the size of micromemory reduced from 480K (VAX) to 64K, and perform 90% of the performance of VAX-11/780. This is also evidence toward a new thinking in instruction set design.

References

[FLY71] Flynn, M., and Rosin, R., “Microprogramming: An introduction and a viewpoint”, IEEE Trans. on Computers, July 1971.

[KAT93] Katz, R., Contemporary Logic Design, Addison-Wesley, 1993.

[KID00] Kidder, T., The soul of a new machine, Back bay books, 2000.

[LEV89] Levy, H. and Eckhouse, R., Computer programming and architecture: The VAX, 2nd ed., Digital press, 1989.

[MAN92] Mange, D., Microprogrammed systems: an introduction to firmware theory, Chapman & Hall, 1992.

[WIL53] Wilkes, M., and Stringer, J., "Microprogramming and the design of the control circuits in an electronic digital computer", Proc. of the Cambridge philosophical society, April 1953. Reprinted in [SIE82].

[WIL85] Wilkes, M., Memoirs of a computer pioneer, MIT Press, 1985.

[SIE82] Siewiorek, D., Bell, C., and Newell, A. Computer structures: Principles and examples. McGraw-Hill, 1982.

Exercises

5.1 Using the simple data path in this chapter, write a microprogram to perform the "swap two registers" operation.

5.2 This is one of my favourite exercises. Suppose we want to write a program to perform a search in a list. We will build a machine specifically to do this task. Assuming that the data structure has been loaded into the memory by some mechanism. Using the simple data path in this chapter, write a microprogram to perform this task WITHOUT the "program" in the code segment. That is, the entire program is in the control unit, or it is in the form of microprogram only.

5.3 To speed up the execution of microprogram, some designer used a *delay* branch, that is, the goto field of microprogram will be delayed by one cycle during the test condition is activated. Design and demonstrate this mechanism. How it will affect the microprogram?

5.4 There are many variations of microprogram: two-level microprogram, nano-program. They are a *compressed* form of describing step-of-control. Please suggest a design of microprogram system that is aimed to be minimal in terms of the size of the microprogram.

5.5 Modern processors have pipeline. Pipeline allows concurrent execution of functional units in the data path. How the step of execution of such data path is described using microprogram?

5.6 Modern processors have many functional units, that is, they are superscalar machines. The instruction stream is reordered such that these functional units can be used as much as possible at the same time. This

is a kind of *packing* many instructions into one long instruction for concurrent execution. Please describe a microprogrammed version of such machine. Is it simpler or more complex?

Chapter 6

Sx Processor

In this chapter, we discuss the main processor of our system, the Sx processor. It is a stack-based processor. Its instruction set is S-code (see Chapter 4). The data path width is 32 bits. The control unit uses 2-phase clock [BUR04a]. For the purpose of teaching cycle-accurate execution, it uses a microprogrammed control unit.

6.1 Data path

Sx has seven special purpose registers (no visible user registers): *TS*, *FP*, *SP*, *NX*, *FF*, *IR* and *PC*. *TS* caches the top of stack value.

TS top of stack
FP frame pointer
SP stack pointer
NX temp register
FF temp register
IR instruction register
PC program counter

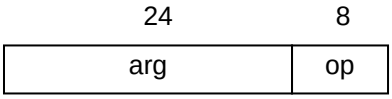
The program counter, *PC*, can be updated independent of other registers. This allows fetching an instruction in one cycle. The data path consists of one ALU connected to the register bank. The output of ALU, *tbus*, goes back to the register bank. The memory is interfaced to the processor through the bus interface unit (BIU). The BIU interfaces the data input, *din*, and the data output, *dout*, to the memory data bus. *din* is selected from *TS* or *FP*. The input of the register bank, *bus*, is multiplexed from *tbus*, *dbus* and *PC*. The address bus, *abus*, is multiplexed from *PC* and *tbus*. The *PC* can be updated with *PC*+1 or

PC+*arg* or *tbus*. The ALU has two ports: *p1*, *p2* and can perform many functions. There are two flags: Zero, and Sign.

Table 6.1 The function of ALU, the inputs are a, b. a is at the port p1. t is the output.

Add: t = b + a	Sub: t = b - a	Mul: t = b * a	Div: t = b / a
Band: t = b & a	Bor: t = b | a	Bxor: t = b ^ a	Not: t = ! a
Shl: t = b << a	Shr: t = b >> a	Eq: t = b == a	Ne: t = b != a
Lt: t = b < a	Le: t = b <= a	Gt: t = b > a	Ge: t = b >= a
Inc: t = a + 1	Dec: t = a - 1	SUB2: t = a - b	P1: t = a
P2: t = b	Z: t = a == 0		

The instruction register, *IR*, has the operation code at the right-most 8-bit and the argument at the left-most 24-bit. The argument field is signed extended to 32 bits. When the instruction requires no argument, the argument field is zero.



Using 2-phase clock enables read-modify-write of registers in one cycle. Reading from registers and memory will be on the positive edge and writing to registers will be on the negative edge. The basic cycles in the control unit are:

- read-modify-write registers
- register transfer
- memory read
- memory write

Memory access

Before going into details of each control cycle, one important consideration is how the memory is accessed (read/write) in each control cycle. A memory access is assumed to take a full cycle. The memory access time is assumed to be half of the processor cycle.

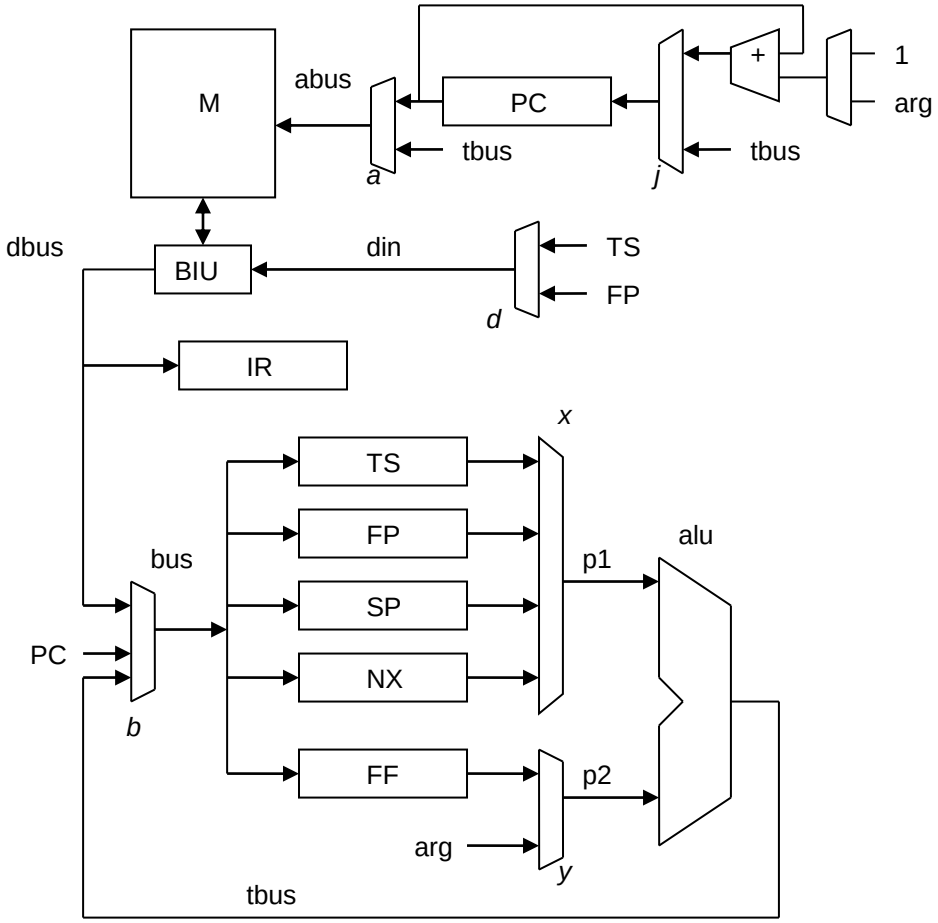


Figure 6.1 The Sx data path

A memory access is initiated by setting the address through *abus*, for a read, a memory read signal is asserted (*mR*). The data from the memory is ready at the middle of the cycle. The data from *dbus* is latched to a register in the middle of the cycle, at the negative edge of the clock.

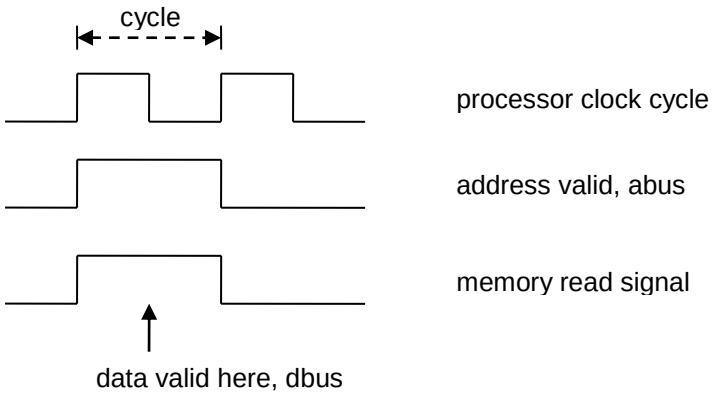


Figure 6.2 A memory read cycle

A memory write cycle is similar. The address and data are asserted at the beginning of the cycle. The memory write signal is asserted (*mW*). The data will be written in the memory in the middle of the cycle, at the negative edge of the clock.

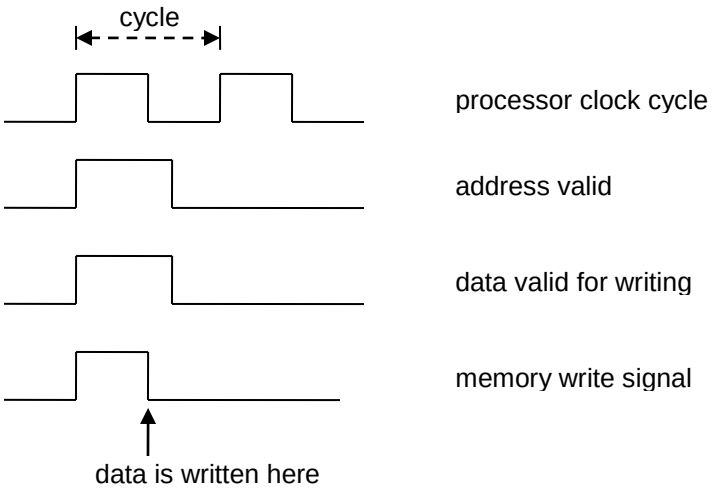


Figure 6.3 A memory write cycle

Register access

The basic read-modify-write starts at the positive edge of the clock. The data are read from the registers into the ALU ports through the multiplexor *x* and *y*. The ALU outputs the result to *tbus*. At the negative edge, *tbus* is fed back to the input of registers, *bus*, through the multiplexor *b* and is latched into the designated register.

read-modify-write a register

```
pos R -> alu -> tbus
neg tbus -> R
```

register transfer

```
pos R1 -> tbus
neg tbus -> R2
```

6.2 Execution cycle

The processor begins its execution cycle with fetching an instruction from the memory which is complete in the first half of the cycle. The instruction in stored in the instruction register (IR). It is decoded through a read-only-memory, called micro-ROM, that stored the address of the microprogram control. The control step then transfers to the appropriate microprogram step. At the end of microprogram step of the instruction, the control is transferred back to fetch the next instruction. A register transfer language (RTL) is used to describe these steps of execution. RTL notation mainly describes the transfer between two registers, *dest = source*. In our notation, RTL does not specify the actual concurrent operation beyond what that can be written as *dest = source*. We will fully specify the concurrent operations in the control unit using the microprogram notation.

Execution cycle in RTL

The registers in the data path are *IR*, *TS*, *FP*, *SP*, *NX*, *FF*, and *PC*. In some operation that there are a number of arguments, the picture of the data in the evaluation stack will be shown in this notation, {.. top of stack}. Each operation is labeled as <*op*>. *M[.]* is the memory.

A shorthand notation is used to describe two often used stack operations: *push* and *pop*.

```
[push x]
sp = sp + 1
M[sp] = x

[pop x]
x = M[sp]
sp = sp - 1
```

The instruction fetch cycle is,

```
ir = M[pc].
```

An operation on the ALU is specified by the operation code field. The opcode bits determine the ALU function. The binary operations are: *add*, *sub*, *mul*, *div*, *band*, *bor*, *bxor*, *shl*, *shr*, *eq*, *ne*, *lt*, *le*, *gt*, *ge*, *inc*, *dec*. In a binary operation, the second argument is in the top of stack; the first argument is in the evaluation stack pointed to by *SP*. Please note the order of argument. The second argument is popped to *FF*, and then two arguments are fed to the ALU. The result is stored back to *TS*.

```
<bop>
pop ff
ts = ts op ff
```

The unary operation affects only the *TS*.

```
<uop>
ts = op ts
```

The access operations to local variables are "*get*" and "*put*". "*get*" must pushes *TS* first to make room for the new data that will be taken from the activation record, *M[FP-arg]*. "*put*" stores *TS* to the activation record then it pops the evaluation stack to *TS* (caching the top of stack).

```
<get>
push ts
ts = M[fp-arg]

<put>
M[fp-arg] = ts
pop ts
```

"*ld*" and "*st*" are similar to get and put but access to the memory instead of the activation record.

```
<ld>
push ts
ts = M[arg]

<st>    {data}
M[arg] = ts
pop ts
```

The "*ldx*" and "*stx*" are a bit more complicate as they have a number of arguments. "*ldx*" takes the *base* from the stack, using *FF* to store it. "*stx*" takes two arguments from the stack, the first one is *idx*, and the second one is *base*. The effective address is calculated using the ALU.

```
<ldx>              {base idx}
pop ff             base
ts = M[ff+ts]

<stx>              {base idx data}
pop nx             idx
pop ff             base
M[ff+nx] = ts
pop ts
```

The literal instruction is simply pushing the argument to *TS*.

```
<lit>
push ts
ts = arg
```

The control transfer operations are: unconditional jump, conditional jump, call and return. "*jmp*" is straightforward. "*jt*" and "*jf*" inspect the zero flag, which reflected the value of *TS*, and transfer the control step accordingly. The evaluation stack is popped to get rid of the old *TS*.

```
<jmp>
pc = pc + arg

<jt>
if ts != 0
  pc = pc + arg
else
  pc = pc + 1
pop ts

<jf>
if ts == 0
  pc = pc + arg
else
  pc = pc + 1
pop ts
```

The "*call*" is perhaps the most complex instruction in this instruction set. It creates a new activation record and transfers the control step to the called function. The new activation record is created on top of the current evaluation stack, overlapping the evaluation stack by the amount of the arity of the called function to pass the parameters. Hence, the new *FP* is offset from the current *SP*. This offset is computed by the code generator and it becomes the argument of the function header, the "*fun*" instruction. "*call*" fetches the function header to get the offset, then uses the offset to set up a new *FP* location and saves the current *FP* there. The *FP* and *SP* are updated to the new location. Next, it pushes the return address and finally jumps to the function body.

```
<call ads>
push ts                 flush eval stack
ts = pc + 1             save ret ads to ts
nx = arg                save call ads to nx
ir = M[arg]             fetch at ads
M[fp+arg] = fp          save old fp
fp = sp = fp + arg      new fp, sp
push ts                 save ret ads
pc = nx + 1             jump to body
```

The "*ret*" instruction sets *PC* to the return address, restores the old *SP*, and restores to the previous activation record. As it is different between returning and not returning a value, it is necessary to decide whether there is a return value or not. The condition *SP = FP* indicates that the *net effect* of the evaluation stack is that the stack is back to its initial state, there is no value to return. The argument of "*ret*" is the offset to set SP back.

```
<ret>
pc = M[fp+1]            restore ret ads
if sp == fp             no return value
  sp = fp - arg         restore sp
  pop ts                cache top of stack
  fp = M[fp]            restore fp
else                    return a value
  sp = fp - arg
  fp = M[fp]
```

If the *net effect* cannot be assumed (because some anomaly in the stack manipulation), then an alternative is to do flow analysis at the compile time to decide whether a function returns a value or not. The "*ret*" instruction must be spilt into two instructions, one without a return value and one with it. Let it be "*ret*" and "*retv*", then the following steps are their execution cycles.

```
<ret>
pc = M[fp+1]
sp = fp - arg
fp = M[fp]
```

```
<retv>
pc = M[fp+1]
sp = fp - arg
pop ts
fp = M[fp]
```

Microprogram

Next, we describe the actual microprogram level. The whole microprogram on Sx processor is presented in the appendix H. The difference between RTL and microprogram is that microprogram specifies the concurrent operations on the data path, including the signals asserted on the multiplexor and ALU. The microprogram level exposed more details that are necessary to realise on actual circuits. A control signal in the microprogram can be regarded as an event that occurs in the data path, such events are latching a data to a register, selecting a multiplexor, memory read, memory write, etc. The notation used in writing microprogram is as follows.

```
src->dest
```

denotes the event that transfer data from a source to a destination where source and destination can be a wire or a register. A wire represents a connection or the input/output of a component.

```
alu(p1 op p2)->dest
```

denotes the ALU performing the "*op*" on its two input ports, *p1* and *p2*, and its output is connected to *dest*, where *dest* can be a wire or a register.

```
mR(ads)->dest
src->mW(ads)
```

mR denotes memory read with the address from the source *ads*, the data is transferred to dest. *mW* denotes memory write with the address sets to the source and the address is *ads*. *src* and *dest* can be a wire or a register. The concurrent events are specified in the microprogram by writing them on the same line. Each event is separated from other event by ",". The order of events is unimportant because they occur in the same clock cycle. However, some event occurs on the positive edge of the clock, some event occurs on the negative edge

of the clock. Reading from registers and memory will be on positive edge and writing to registers will be on negative edge.

```
src->dest, mR(ads)->dest, ...
```

The "jump" of the microprogram is achieved by loading the "next microaddress" bit to the microprogram counter. It can be unconditional or conditional. The *next address* is written as `<label>`. There are three "jump" events in Sx data path.

`ifT`	jump if ts is not zero
`ifF`	jump if ts is zero
`decode`	multiway branch according to opcode

`PC` has special events.

`pc+1`	is increment `PC` by 1
`pc+arg`	is increment `PC` by `arg`

We have a shorthand notation for `SP`.

`sp-1`	is `alu(sp-1)->sp`
`sp+1`	is `alu(sp+1)->sp`

The microprogram for Sx is followed from its RTL description. We begin with the instruction fetch.

```
<fetch>   [micro 47]
mR(pc)->ir, decode
```

Where `decode` is a control signal to look up the microprogram address according to the opcode field on the instruction register, `IR`.

Next is the binary operation.

```
<bop>   [micro 49]
mR(sp)->ff
sp-1
alu(ts op ff)->ts, pc+1, <fetch>
```

Please note that the *PC* is incremented at the end of the instruction cycle and then the microprogram is jumped back to the instruction fetch at the beginning.

```
<uop>   [micro 53]
alu(ts op ?)->ts, pc+1, <fetch>

<get>   [micro 55]
sp+1
ts->mW(sp)
alu(fp-arg)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

"*get*" pushes ts and loads *M[FP-arg]* using ALU to do the effective address calculation. The address is presented and the memory read signal is asserted. The data is ready and is latched to *TS*.

```
<put>   [micro 59]
alu(fp-arg)->tbus, ts->mW(tbus)
mR(sp)->ts
sp-1, pc+1, <fetch>
```

"*put*" writes *TS* to *M[fp-arg]* then pops to *TS*. The *SP*-1 and *PC*+1 can be concurrent because *SP*-1 uses the ALU while *PC*+1 does not use ALU. *PC* has its own adder.

```
<ld>   [micro 64]
sp+1
ts->mW(sp)
mR(arg)->ts, pc+1, <fetch>

<st>   [micro 68]
ts->mW(arg)
mR(sp)->ts
sp-1, pc+1, <fetch>
```

"*ld*" and "*st*" are similar to "*get*" and "*put*" but "*ld*" and "*st*" access the memory using direct address from the argument of the instruction.

```
<ldx> [micro 70]                {ads idx}
mR(sp)->ff
sp-1
alu(ff+ts)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

"*ldx*" gets the base address to *FF*. The *index* is at *TS*. The effective address is calculated using ALU and the value is fetched from the memory.

```
<stx>  [micro 74]               {ads idx val}
mR(sp)->nx                      pop idx to nx
sp-1                            pop ads to ff
mR(sp)->ff
alu(nx+ff)->tbus, ts->mW(tbus)
sp-1
mR(sp)->ts                      cache ts
sp-1, pc+1, <fetch>
```

"*stx*" has three arguments. It gets the *index* to *NX*, and the base address to *FF*. The effective address is calculated using ALU. The value in *TS* is stored to that address. Finally, the top of stack is cached to *TS*.

```
<lit>  [micro 80]
sp+1
ts->mW(sp)
arg->ts, pc+1, <fetch>

<jmp> [micro 84]
pc+arg, <fetch>

<jt>  [micro 86]
alu(ts=0), ifT <j3>             if true, don't jump
<j2>                            jump
pc+arg, mR(sp)->ts
sp-1, <fetch>

<jf>  [micro 92]
alu(ts=0), ifT <j2>             if true, jump
<j3>                            don't jump
pc+1, mR(sp)->ts
sp-1, <fetch>
```

The "*jt*" and "*jf*" use the event "*ifT*" to do conditional branching. The branching is the "goto" style of programming which is quite natural in a microprogram. It saves the microprogram space.

```
<call>   [micro 98]
sp+1
ts->mW(sp), pc+1                    flush stack
pc->ts
arg->tbus->nx, mR(tbus)->ir         fetch fun, nx=ads
alu(sp+arg)->tbus, fp->mW(tbus)     save old fp
alu(sp+arg)->fp->sp                 new fp, sp
alu(nx+1)->pc, <fetch>
```

The event "*arg->tbus->nx*" uses ALU to pass *arg* through. This event saves the address of the called function to *NX*. To get the offset, the "*fun*" instruction is fetched to *IR* which then its argument is used. There are two concurrent register writes in the event "*alu(sp+arg)->fp->sp*". The address of the body of the function, *NX*+1, is updated to *PC*.

```
<ret>   [micro 106]
sp->ff
alu(fp=ff), ifF <r2>
ts->pc                              do ret
alu(fp-arg)->sp
mR(sp)->ts
sp-1
mR(fp)->fp, <fetch>                 restore fp
<r2>                                do retv
alu(fp+1)->tbus, mR(tbus)->ff       ret ads
ff->pc
alu(fp-arg)->sp
mR(fp)->fp, <fetch>
```

The "*ret*" tests the condition *FP = SP* to decide whether there is a value to return or not. To do the test, *SP* is moved to *FF* to use ALU operation. When doing "*ret*", the return address is in *TS* but when doing "*retv*", the return address is in *M[FP+1]*. *FF* is used to pass the value through *PC*.

```
<sys>   [micro 119]
<array>
<end>
trap, pc+1, <fetch>
```

The instructions "*sys*", "*array*" and "*end*" have no implementation on the real processor. They are used in the simulator. The event "*trap*" is used by the simulator to handle these instructions.

After the microprogram is completely written, the number of cycle taken by each instruction is known. They are shown in the table below.

Table 6.2 The number of cycle for each instruction

bop 4	uop 3	get 4	put 4
ld 4	st 4	ldx 4	stx 8
lit 4	jmp 2	jt 4	jf 4
call 8	ret 8	retv 7	

6.3 Performance

A number of benchmark programs are compiled and then run on the Sx processor simulator. Table 6.3 reports the number of instructions, the number of cycles and the cycle-per-instruction number for each program.

"bubble" is a bubble sort program sorting an array of 20 integers, initially the value in the array is in descending order and sort to ascending order. "hanoi" is a program to solve Hanoi problem with 6 disks. "matmul" is a matrix multiplication program; the input is two matrices of the size 4 x 4. "perm" is a program to do all permutation of {0,1,2,3}. "queen" is a program to find all configurations of 8-queen problem. "quick" is a quicksort program with a similar input to "bubble". "sieve" is a program to find prime numbers less than 1000 using "Sieve of Eratosthenes" algorithm. "aes" is a program to do AES (Advanced Encryption Standard) block cipher (128, 128) bit key. The average cycle-per-instruction number of Sx processor is 4.3. This is quite good comparing to the stack-based processor of an earlier design [BUR04c], a 16-bit processor runs the same "aes" in 284108 cycles.

Table 6.3 The performance of Sx processor

program	noi	cycle	cpi
bubble	10068	44214	4.39
hanoi	2312	10092	4.37
matmul	3043	12880	4.23
perm	4868	20932	4.30
queen	618665	2576210	4.16
quick	3172	13539	4.27
sieve	28026	124338	4.44
aes	30579	131560	4.29

6.4 Sx processor simulator

The design of Sx processor with the detailed design of the data path and its control unit using microprogram is complete enough to be realised on real silicon using either FPGA (Field Programmable Gate Array) technology or ASIC (Application Specific Integrated Circuit). However, it is much easier to study it using a simulator. The Sx processor simulator performs cycle-accurate simulation of Sx processor executing programs. The simulator executes step-by-step microprogram of Sx. It is used to validate the microprogram and to collect the performance statistics.

Data path

The data path consists of registers, multiplexors, combinational circuits such as ALU and wires. The registers and wires are simulated as variables of type integer capable of holding 32-bit values. The multiplexors are simulated as if..then statements to update the output wires. The simulated ALU performs the expected operations on its input ports and updates the flags. The combinational circuits can be simulated by statements to update the output wires.

Control unit

A straightforward way to simulate the microprogram control unit is to regard the microprogram as a ROM, a two-dimensional array of bits. Each address is called a microprogram word. One microprogram word is executed in one cycle. Each word contains event-control bits where each bit represents an event in the data path. The event that is active is 1, otherwise it is 0. Each event has its symbolic name. The simulation is run as event-driven. The main simulation loop looks at each microprogram word and scans the event bits to find the active one then performs the action for that event. This include the control transfer of microprogram address which updates the microprogram counter. The simulation loop continues until the "*end*" instruction throws a trap with the event "*trap*".

```
62
  0 00000000000010000000000001000000100010000001
  1 00100001000100000000000100000010100000000010
  2 00100010000000000001000000001000000000000011
  3 10001010000001000000001000100001000000000000
  4 10000010000001000000001000100001000000000000
. . .
 55 00010010000001000000000100100001000000000000
 56 10000010000000000000000100000100000000111001
 57 01000101000100000100000000100000100000111010
 58 10000010000000000001000000100000000000111011
 59 01000100001100000100000000000000010000111100
 60 00010010000001000000000100100001000000000000
 61 00000000000001000000000000000000100001000000
```

Figure 6.4 The microprogram ROM

The events are defined as follows.

multiplexor *x* selects {*ts, fs, sp, nx*}
multiplexor *y* selects {*ff, arg*}
multiplexor *b* selects {*tbus, dbus, pc*}
multiplexor *d* selects {*fp, ts*}
multiplexor *a* selects {*pc, tbus*}
multiplexor *j* selects {*pc+1, pc+arg, tbus*}
alu events are {*add, sub, inc, dec, z, eq, op, p1, p2*}

load registers events are {*ir, ts, fp, sp, nx, ff, pc*}
memory events are {*mR, mW*}
next micro-address events are {*ifT, ifF, decode, trap*}

We use the naming convention as follows. The multiplexor has its name as a prefix followed by its choice, for example, mux *x* selects ts is written as *x.ts*. The ALU is similar, ALU performs *inc* is written as *alu.inc*. The load register is written with a prefix "l" followed by the name of the register, *lpc* is load *PC*.

The registers are *IR*, *PC*, *TS*, *FP*, *SP*, *NX*, *FF*. *Z* is the zero flag. The wires are *p1*, *p2*, *tbus*, *abus*, *dbus*, *bus*, *pcin*. The functions *IRarg()*, *IRop()* decode the *op* and *arg* field of *IR*. *alu()* performs ALU operations. *udecode()* returns the microaddress corresponded to the current opcode. *m2* is the next microaddress, specified as the next address field in the microprogram word.

Let *mx[mpc][bit]* be the microprogram ROM. The main simulation loop is.

```
while (running)
  m2 = next micro address field
  for i = 0 to microwidth-1
     s = scan for active event in mx[mpc][i]
     do s
  mpc = m2
```

For each event in a microprogram word. Let *s* be the event that is active.

```
switch(s){   [sx 120]
    case x.ts:     p1 = TS
    case x.fp:     p1 = FP
    case x.sp:     p1 = SP
    case x.nx:     p1 = NX
    case y.ff:      p2 = FF
    case y.arg:     p2 = IRarg()
    case alu.add:  tbus = alu(icAdd,p1,p2)
    case alu.sub:  tbus = alu(FSUB,p1,p2)
    case alu.inc:  tbus = alu(icInc,p1,p2)
    case alu.dec:  tbus = alu(icDec,p1,p2)
    case alu.z:    tbus = alu(FZ,p1,p2)
```

```
        case alu.eq:    tbus = alu(icEq,p1,p2)
        case alu.p1:    tbus = alu(FP1,p1,p2)
        case alu.p2:    tbus = alu(FP2,p1,p2)
        case alu.op:    tbus = alu(IRop(),p1,p2)
        case a.pc:      abus = PC
        case a.tbus:    abus = tbus
        case d.ts:      dbus = TS
        case d.fp:      dbus = FP
        case mR:        dbus = M[abus]
        case mW:        M[abus] = dbus
        case b.tbus:    bus = tbus
        case b.dbus:    bus = dbus
        case b.pc:      bus = PC
        case j.pc1:     pcin = PC + 1
        case j.pcarg:   pcin = PC + IRarg()
        case j.tbus:    pcin = tbus
        case lpc:       PC = pcin
        case lir:       IR = dbus
        case lts:       TS = bus
        case lfp:       FP = bus
        case lsp:       SP = bus
        case lnx:       NX = bus
        case lff:       FF = bus
        case ifT:       m2 = (Z == 0) ? m2 : mpc+1
        case ifF:       m2 = (Z == 1) ? m2 : mpc+1
        case decode:    m2 = udecode()
        case trap:      trap(IRop(),IRarg())
    }
```

The simulator is sequential, that is, it simulates each event one by one. Therefore the order of scanning the event list (the bits in a microprogram word) is important to get the correct result. All the positive-edge events must be updated before the negative-edge events. Within the same group the input side is updated to the output side. For example the read-modify-write loop of a register, the read side must be performed, then goes through the modify operation (with some function) from input to output, finally the write is performed to that register. With these rules the order of events are:

```
mux x, mux y, alu,
mux a, mux d, mR, mW,
mux b, mux j,
```

```
load registers,
ifT, ifF, decode, trap.
```

6.5 Lab session

A tool is provided to write a microprogram. The "mgen" tool takes the input file as a microprogram specification and outputs the microprogram ROM as shown in Fig 6.4. The microprogram must be written in the following form. The specification composed of two sections, the first section is the signal definition and the second section is the microprogram. The signal definition lists all the events, the order of the event is important as the simple implementation of the simulator will simulate each event according to this order (this can be relaxed in the alternative implementation, see the exercise). Here is the example of the signal definition, the section starts with "`.s`". The line started with `..` is the comment line.

```
.. sx microprogram v 1.0   [micro 1]
..
.s
x.ts
x.fp
x.sp
...
alu.add
alu.sub
...
.. load registers
lir
lts
lfp
lsp
lnx
lff
lpc
mR
mW
.. next micro ads
ifT
ifF
```

```
decode
trap
```

After the signal definition the next section is the microprogram section. Each line consists of,

```
[:label]  event* [/label] ;
```

A line starts with a label, follows by events and the next micro-address label, and ends with ";". The starting label and the micro-address label are optional. The microprogram section starts with "*.m*" and ends with "*.e*". Here is an example.

```
.m          [micro 45]
:fetch
  a.pc mR lir decode ;
:bop
  x.sp alu.p1 a.tbus mR b.dbus lff ;
  x.sp alu.dec b.tbus lsp ;
  x.ts y.ff alu.op b.tbus lts j.pc1 lpc /fetch ;
...
:jmp
  j.pcarg lpc /fetch ;
:jt
  x.ts alu.z ifT /j3 ;
:j2
  j.pcarg lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
  x.sp alu.dec b.tbus lsp /fetch ;
:jf
  x.ts alu.z ifT /j2 ;
:j3
  j.pc1 lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
  x.sp alu.dec b.tbus lsp /fetch ;
:end
  trap j.pc1 lpc /fetch ;
.e
```

The "*fetch*" reads as, mux *a* selects *PC* (to be the address of the memory operation), memory read, load *IR*, jump to the corresponding micro-address. The "*bop*" reads as *SP* goes through ALU to *tbus*, mux *a* selects *tbus* (to be the address of the memory operation), memory read, mux *b* selects *dbus* (to be the input of registers), load register *FF*. Then, *SP* goes through ALU to do -1 and

back to bus to write to *SP*. Then, mux *x* selects *TS*, mux *y* selects *FF*, ALU performs a function according to the opcode, mux *b* selects *tbus* (to be the input of registers), load register *TS*, at the same time, *PC* is updated +1, then jump to "*fetch*", the instruction fetch.

How to microprogram Sx

To write microprogram for Sx, the microprogram specification is in the file "mspec.txt". "mgen" transforms the specification to a ROM file. Store it in the name "mpgm.txt". The source for "mgen" can be found in sx0.zip package.

```
c:> mgen < mspec.txt > mpgm.txt
```

Here is what "mpgm.txt" looked like.

```
62
  0 00000000000010000000000001000000100010000001
  1 00100001000100000000000100000010100000000010
  2 00100010000000000001000000001000000000000011
  3 10001010000001000000001000100001000000000000
  4 10000010000001000000001000100001000000000000
 . . .
 55 00010010000001000000000100100001000000000000
 56 10000010000000000000000100000100000000111001
 57 01000101000100000100000000100000100000111010
 58 10000010000000000001000000100000000000111011
 59 01000100001100000100000000000000010000111100
 60 00010010000001000000000100100001000000000000
 61 00000000000001000000000000000000100000100000
```

A few right most bits are the next microprogram address. "mgen" also generates binding of symbolic names to numeric values which are used in the simulator, "mspec.h". This is it:

```
#define s_x_ts 0
#define s_x_fp 1
#define s_x_sp 2
#define s_x_nx 3
#define s_y_ff 4
```

```
. . .
#define a_fetch 0
#define a_bop 1
#define a_uop 4
#define a_get 5
#define a_put 8
#define a_popts 9
#define a_ld 11
. . .
#define a_end 61
#define MCWIDTH 38
#define MAWIDTH 6
#define MLEN 62
```

The event names prefixed "s_" are the signal events, prefixed "a_" are the address of the label of microprogram. The MCWIDTH is the number of the control bits. The MAWIDTH is the number of bit of the microprogram address field. The MLEN is the number of microprogram word. "mgen" also generates a listing file "mlist.txt". It is used for debugging.

The next step is to convert this micro-ROM into an event-list. "sxgen" combines "mgen" and conversion to event-list. (If you are using sx1, sx2 simulator, sxgen is inside the simulator, you don't have to do it explicitly). "sxgen" reads the files "mpgm.txt" and "mspec.h" then generates "sxbit.h" which must be compiled with the simulator.

```
c:> sxgen < mspec.txt
```

The "sxbit.h" contains the binding, the op-decoder-rom (udop[]), the pointer to event-list (mw[]), the event-list itself (mx[]) and finally the next-address-rom (nxt[]).

```
#define s_x_ts 0
#define s_x_fp 1
. . .
#define a_sys 51
#define a_array 51
#define a_end 51
#define MCWIDTH 38
#define MAWIDTH 6
#define MLEN 52
```

```
int udop[] = {
0, 1, 1, 1, 1, 1, 1, 1, 4, 1,
1, 1, 1, 1, 1, 1, 1, 0, 15, 18,
40, 0, 51, 51, 5, 8, 11, 14, 26, 27,
30, 23, 33, 0, 0, 0, 51, 0, 0, 0, 0 };

int mw[] = {
0, 5, 12, 17, 25, 32, 37, 43, 53, 60,
67, 74, 79, 85, 94, 100, 107, 112, 122, 129,
134, 141, 148, 153, 158, 164, 171, 174, 178, 187,
192, 196, 205, 210, 215, 223, 226, 234, 241, 248,
253, 258, 263, 268, 274, 281, 286, 293, 300, 305,
311, 318, 0 };

int mx[] = {
12, 32, 25, 36, -1,
2, 23, 11, 32, 7, 30, -1,
2, 19, 6, 28, -1,
0, 4, 22, 6, 13, 31, 26, -1,
. . .
1, 5, 17, 6, 28, -1,
1, 23, 11, 32, 7, 27, -1,
13, 31, 37, -1,
0 };

int nxt[] = {
1, 2, 3, 0, 0, 6, 7, 0, 9, 10,
0, 12, 13, 0, 9, 16, 17, 0, 19, 20,
21, 22, 9, 24, 25, 0, 0, 31, 29, 0,
28, 32, 0, 34, 35, 36, 37, 38, 39, 0,
41, 47, 43, 44, 45, 46, 0, 48, 49, 50,
0, 0, 0 };
```

You must recompile the simulator to include your new signal definitions, or new instruction labels. The processor simulator takes a proper object file as input and run it. If it is `sx0`, the processor is run in a batch mode. For `sx1` and `sx2` they run in interactive mode. You can ask for help by typing “h”. The following session is `sx0` running quicksort.

```
D:\sx\test>sx qs.obj
load program, last address 203
DP 1000
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4820 instructions, 20231 clocks, CPI 4.20

D:\sx\test>
```

6.6 Summary

In this chapter we have learnt the detailed design of a stack-based processor that can execute S-code instructions directly. This processor is the basic component of our hardware system. The data path is quite simple. The processor is aimed for clarity. It is simple enough to be studied and to be modified without due complexity by students. At the same time, it retained the flavour of reality that the design is complete enough to be realised as a real processor. The control unit has been detailed down to the cycle-by-cycle execution of an instruction. Microprogram represents the specification of the execution behaviour. Writing a microprogram starts with a RTL description which concerns only the registers transfer. Then, the specification is refined to a microprogram level which also concerns concurrency of operations. With microprogram fully specified, the number of cycle taken by each instruction can be calculated. The algorithm to simulate the processor has been presented. The processor simulator is event-driven. The microprogram is regarded as events occur in the data path.

In the laboratory session, we learn how to write a concrete microprogram. Tools are given to convert this concrete specification into a data structure suitable for simulation. The processor simulator itself has been described in details enough that students can modify it to include other instructions and/or additional features. We shall see in the next chapter how to improve the performance of this Sx processor.

6.7 Further reading

Stack-based processors have been a popular architecture in the past due to its simplicity and its compatibility with *structured programming* paradigm around 1970-1980. Burroughs has developed many commercial machines based on this

type of architecture (Burroughs B5500) [BUR68]. For more recent discussion of stack-based processor see Koopman [KOO89] which discussed the strength of this architecture including a comprehensive survey of many stack-based processors. The weakness of stack architecture lies in its performance. As RISC type of processors [PAT82] [PAT85] [HEN84] [STA88] becomes popular during 1980-1990, it dominates all the market with its high performance. Nowadays, all processors are register-based. We shall discuss the use of registers in the next chapter.

References

[BUR68] Burroughs B5500 Electronic Information Processing System: Operation manual. Burroughs Corp. Detroit, 1968.

[BUR04a] Burutarchanai, A., and Chongstitvatana, P., "Design of a two-phased clocked control unit for performance enhancement of a stack processor", National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.

[BUR04b] Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., "A fast instruction fetch unit for an embedded stack processor", Proc. of Int. Conf. on Information and Communication Technologies, Thailand, 18-19 November 2004.

[BUR04c] Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON, Thailand, 21-24 November 2004.

[HEN84] Hennessy, J., "VLSI Processor Architecture", IEEE Trans. on Computers, December 1984.

[KOO89] Koopman, J., Stack Computers: the new wave, Ellis Horwood, 1989.

[PAT82] Patterson, D., and Sequin, C., "A VLSI RISC", Computer, Septermber, 1982.

[PAT85] Patterson, D., "Reduced Instruction Set Computers", Communications of the ACM, January, 1985.

[STA88] Stallings, W., "Reduced instruction set computer architecture", Proc. of the IEEE, vol. 76, no. 1, January 1988, pp. 38-55.

Exercises

6.1 Compile and run Sx-kit

6.2 Write three benchmark programs in nut: hanoi, matmul, (bubble, quick already existed in Sx-kit) run it under sx-simulator to test the correctness.

6.3 Modify the simulator to count the frequency of each instruction. Run the above benchmarks to find out the dynamic instruction count.

6.4 Write additional microprogram for the instruction "*inc*" and "*dec*".

6.5 The combination of test-and-jump occurs frequently in the program. Write new instructions, such as, *jump-if-less-than*, *jump-if-equal* etc. Modify the code generator to output the new instructions. Measure their effectiveness.

6.6 Suggest some way to improve the speed of Sx processor.

6.7 A simple implementation of the processor simulator is to scan the microprogram ROM and execute the active event directly. The next micro-address is also converted into an integer. This makes the processor simulator 5 to 10 times slower than the implementation illustrated in the chapter. Do it and compare the speed.

Chapter 7

Performance Enhancement

In this chapter, we will concentrate on performance of the processor. Performance improvement starts with an analysis of the execution profile to understand where the data path spends most of its time. The effort is then directed to redesign the data path, usually by increasing the concurrent operations in the data path. There are interactions amongst choices of design. Choosing one will affect another. The gain in terms of performance must be weighted against the increased complexity in terms of the circuit size or the resource. For the purpose of our study we will not change the instruction set. The performance improvement will come from the change of *micro-architecture* only.

7.1 Profile analysis

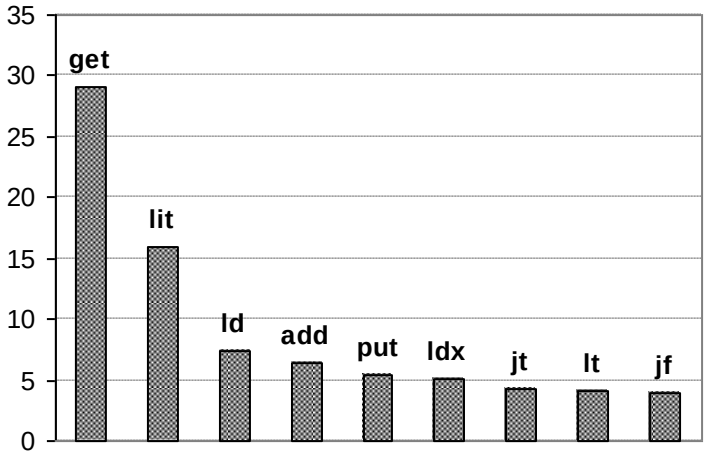


Figure 7.1 The most frequently used instructions

The profile is collected from running the benchmark programs in Chapter 6 (Table 6.3). In an analysis of the profile of execution of instructions, most frequently used instructions are:

```
get, lit, ld, add, put, ldx, jt, lt, jf.
```

These instructions altogether are more than 80% of all instructions executed in the suite of benchmark programs[1]. To improve the performance, the effort should be spent on improving these instructions.

Let the shorthand notation of *push/pop* be

```
push x is
  sp+1->sp
  x->mW(sp)

pop x is
  mR(sp)->x
  sp-1->sp
```

The microprograms for the instructions: *get*, *lit*, *ld*, *add*, *put*, *ldx*, *jt*, *lt*, *jf* are:

```
<get>
  push ts
  alu(fp-arg)->tbus, mR(tbus)->ts

<lit>
  push ts
  arg->ts

<ld>
  push ts
  mR(arg)->ts

<bop>
  pop ff
  alu(ts op ff)->ts
```

[1] Notable is the "inc v" instruction which is not generated from this compiler (gen.txt). It is used often but has not been included in this experiment.

```
<put>
  alu(fp-arg)->tbus, ts->mW(tbus)
  pop ts

<ldx>                          ; {ads idx}
  pop ff
  alu(ts+ff)->tbus, mR(tbus)->ts

<jt>
  alu(ts=0), ifT <j3>     ; if true, don't jump
<j2>
  pc+arg
  pop ts

<jf>
  alu(ts=0), ifT <j2>     ; if true, jump
<j3>
  pc+1
  pop ts
```

We can observe that all instructions perform *push* and *pop*. This is because two reasons. The first reason is that it is the nature of the stack-based instruction set to access data from the evaluation stack. The second reason is that the top of stack is cached in *TS*, therefore there is a lot of traffic between *TS* and the stack segment. In Sx processor, *push* and *pop* do one memory access and use ALU to do increment/decrement *SP*.

7.2 Key ideas

There are two key ideas:

1. The operations *push/pop* can be done in one cycle if *SP* can be incremented/decremented independent of ALU and they can achieve pre-increment and post-decrement at the proper negative-edge of the clock.
2. To improve "*get*", the most frequently used instruction, the local variable must be stored in a register instead of memory as *push/pop* also accesses memory. If it is done properly "*get*" will take just one cycle. Let *v[.]* denotes the *caching* register bank. It is connected to *TS* in the data path (see Fig. 7.3).

```
<get>
  $1 push ts, $2 v[arg]->ts
```

Where $1 is positive-edge and $2 is negative-edge, *v[.]* is the cache register. The old value *TS* is pushed into memory at $1, before the new value from *v[arg]* is written to *TS* at $2.

Push/pop

To push a register to memory in one clock, the "*sp*+1" must appear at the address bus from the beginning of $1, *TS* is presented to data bus at the same time, at the beginning of $2 memory write signal is ended (it is assumed that the value is written into memory here), the value of "*sp*+1" is also written to SP at this time.

```
push ts is
  sp+1->sp
  ts->mW(sp)

$1 sp+1->abus, ts->dbus, $2 mW(abus), sp+1->sp
```

Popping a register can be done in one cycle. The value "*sp*" is presented to the address bus at $1. The memory is read. At $2, the data is latched to a register, at the same time, "*sp*-1" is written to *SP* (post-decrement).

```
pop x is
  mR(sp)->x
  sp-1->sp

$1 sp->abus, mR(abus)->dbus, $2 dbus->x, sp-1->sp
```

With this new *push/pop*, other instructions will also improve. "*lit*" takes only one cycle for execution.

```
<lit>
  $1 push ts, $2 arg->ts
```

"*ld*" cannot be done in one cycle as it reads the memory twice, the first one for push *TS*, the second one for the value. Therefore "*ld*" takes 2 cycles for execution.

```
<ld>
  push ts
  mR(arg)->ts
```

All the binary operations now take 2 cycles.

```
<bop>
  pop ff
  alu(ts op ff)->ts
```

"*put*" can be done in one cycle. *TS* is read at $1 and transfer to *v[arg]*. A value in the evaluation stack is popped into *TS* at $2.

```
<put>
  ts->v[arg], pop ts
```

Similarly to *bop*, "*ldx*" takes 2 cycles. "*jt*" and "*jf*" take 3 cycles.

Implementing the SP unit

The SP unit performs pre-increment at $1, post-decrement at $2, and loads a value from bus at $2. There is a feed forward path from the adder "*sp*+1" to achieve the pre-increment. All multiplexors are asserted at $1, latching the register *SP* is at $2.

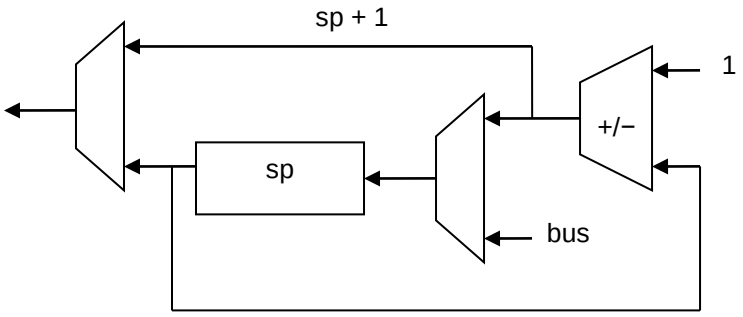


Figure 7.2 The SP unit

Stack frame

A number of registers are used to cache a part of stack frame. This is called the *stack frame caching* [CHO06]. The stack frame remains unchanged from the original design. The local variables, $lv_1..lv_n$, are cached into *v[1]..v[n]* the cache registers. When the context is changed by *call/ret*, these registers are affected. Before a new activation record is created the *old* cached registers must be written back to the current activation record. And vice versa, upon returned from a call, after the activation record is deleted and the old one restored, the cache registers must be refreshed (re-cached) from the activation record. This behaviour is the same as saving/restoring registers upon call/ret on a register-based processor. However, in Sx, these saving/restoring are performed at the microprogram level instead of at the instruction level.

```
call
* save v to the current stack frame
  push ts (flush stack)
  create a new frame
  save fp' and return address
* cache v from the new frame
  update sp

ret
  restore return address, sp
  restore the old frame
* cache v of this current frame (restore old v)
  if it is "ret" pop ts
```

The lines with * are the additional work that must be done to do stack frame caching. The microprograms for *call/ret* for saving/caching *v[.]* are as follows.

```
<save v>
  alu(fp-n)->fp
  vn->mW(fp), alu(fp+1)->fp
  ...
  v1->mW(fp), alu(fp+1)->fp
```

```
<cache v>
  alu(fp-n)->fp
  mR(fp)->vn, alu(fp+1)->fp
  ...
  mR(fp)->v1, alu(fp+1)->fp
```

If the size of caching register is *n* then the extra cycle in `call/ret` instruction is O(3(*n*+1)).

7.3 New data path

The enhanced Sx, or Sx2, has many additional functional units, notably the SP unit and the `v[.]`, cache registers. The number of `v[.]` is 4. However, the major change is in the control unit. There are many more control signals to control the additional functional units and there are more steps of control.

The events are defined as follows.

multiplexor `x` selects {`ts, fs, nx`}
multiplexor `y` selects {`ff, u, arg`}
multiplexor `b` selects {`tbus, dbus, sp`}
multiplexor `d` selects {`fp, ts, v, u`}
multiplexor `a` selects {`pc, tbus, fp, spu`}
multiplexor `j` selects {`pc+1, pc+arg, tbus`}
multiplexor `si` selects {`sp+1, sp-1, sp+arg, tbus`}
multiplexor `so` selects {`spx, sp`}
multiplexor `z` selects {`dbus, ts`}
multiplexor `w` selects {`v1, v2, v3, v4, varg`}
multiplexor `t` selects {`vout, pc, bus`}
multiplexor `u` selects {`dbus, iru`}

alu events are {*add, sub, inc, dec, z, eq, op, p1, p2, add2*}
load registers events are {*ir, ts, fp, sp, nx, ff, pc, v1, v2, v3, v4, varg, u*}
memory events are {*mR, mW*}
next micro-address events are {*ifT, ifF, decode, ifu0, ifp0, ifargm, skipu, trap*}

The new events on the next micro-address {*ifu0, ifp0, ifargm, skipu*} and the register U require further explanation. They are necessary for the control of saving/caching the stack frame. The simple analysis of the previous section has the worst case additional running time for using stack frame caching in O(3(*n*+1)) cycles. However, it is not the case that a function will use all *v* registers. Let *maxv* be the number of *v* registers, *fs* be the size of activation record. If the size of activation record is less than *maxv* then only *v[1]..v[fs]* must be saved/cached. Let *u* be *max(fs, maxv)*; it is stored in the register U. The U register is used to skip a number of microprogram words to achieve this effect. The control signal is "*skipu*". "*skipu*" sets the next microprogram address to *mpc+(maxv-u)*. This offset is already stored in the next microprogram address field. The microprogram below shows the part to save *v* registers at the function call.

```
<save v>
   alu(fp-u)->fp, skipu
   v[4]->mW(fp), fp+1->fp
   v[3]->mW(fp), fp+1->fp
   v[2]->mW(fp), fp+1->fp
   v[1]->mW(fp), fp+1->fp, <fetch>
```

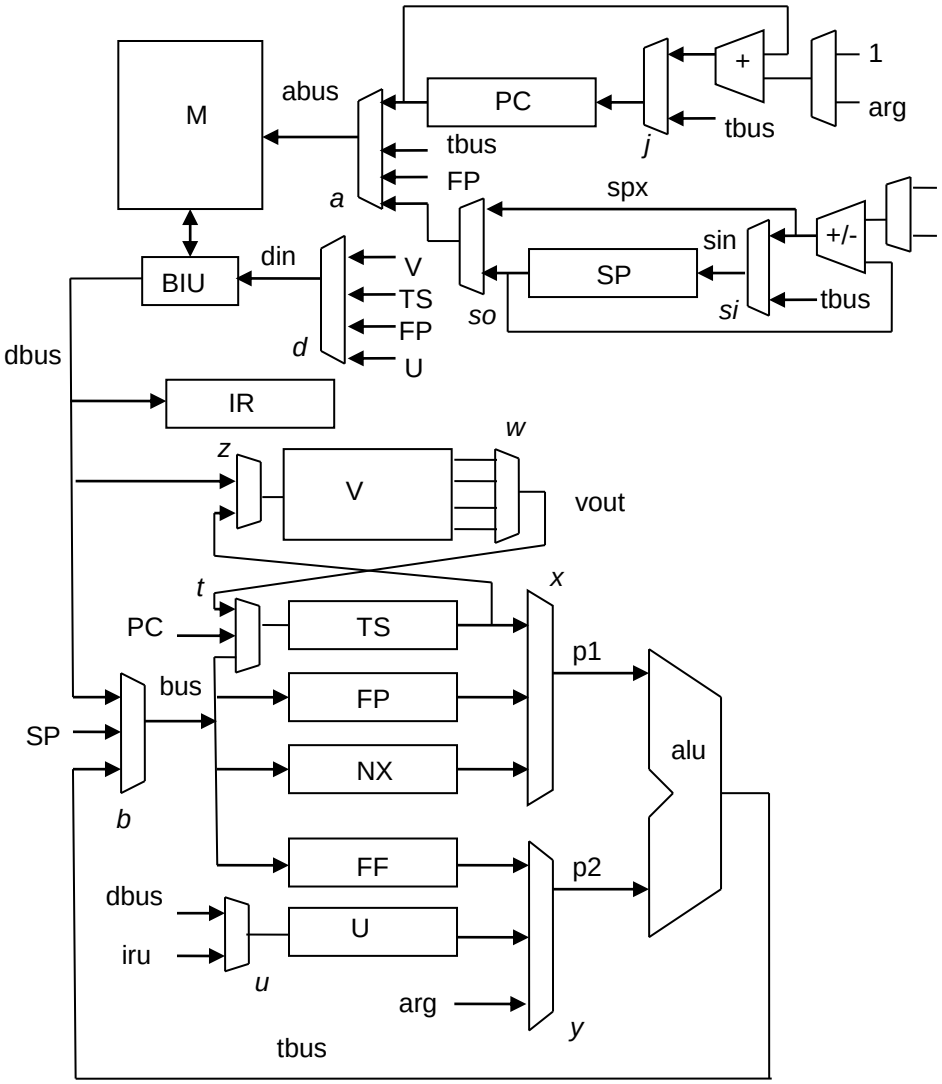
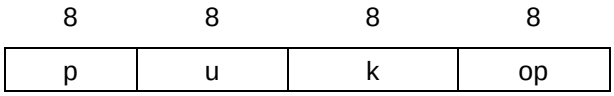


Figure 7.3 The Sx2 data path

Caching *v* registers can be achieved similarly. In fact, when calling a function, not even U registers need to be cached, only the passing parameters (*p*) need to be cached from the evaluation stack (it is a save when $p < u$). However, it becomes too complex to do in a simple microprogram such as this due to the ordering the variables. Therefore, a tradeoff has been made not to exploit this fact. One special case has been implemented, when $p = 0$ to bypass the passing parameter caching (using the event "*ifp0*"). These two parameters, *p* and *u*, are encoded in the argument of "*fun*" instruction with the following format.

fun.p.u.k



Where *k* is the frame size, *p* is the arity, *u* is *max(fs, maxv)*. This is done by the code generator or at the loader of the processor simulator. The U register is valid throughout the current context; it is used when "*call*" and "*ret*".

7.4 Microprogram of Sx2

Here is the microprogram of the Sx2 processor in whole with the explanation.

```
<fetch>   [micro 205]
  mR(pc)->ir, decode
```

The effect of concurrency of SP unit with other operations can be observed in almost every instruction.

```
<bop>   [micro 207]
  mR(sp)->ff, sp+1->sp
  alu(ts op ff)->ts, pc+1, <fetch>

<uop>   [micro 210]
  alu(ts op ?)->ts, pc+1, <fetch>
```

When *arg* > *maxv*, the "*get*" accesses normal memory. Even in this case the step of execution is shortening due to the SP unit. When *arg* <= *maxv*, the access

in on *v* registers and the execution takes only one cycle. The "*decode*" event performs a check on the argument of "*get*" and branches to the proper "*get x*" microprogram address where *x* is *1..maxv*. The pre-increment using "*sp*+1" feed-forward path can be seen.

```
<get>  [micro 212]
  ts->mW(sp+1), sp+1->sp ; push ts
  alu(fp-arg)->tbus, mR(tbus)->ts, pc+1, <fetch>

<get1>
  ts->mW(sp+1), v[1]->ts, sp+1->sp, pc+1, <fetch>

<get2>
  ts->mW(sp+1), v[2]->ts, sp+1->sp, pc+1, <fetch>

<get3>
  ts->mW(sp+1), v[3]->ts, sp+1->sp, pc+1, <fetch>

<get4>
  ts->mW(sp+1), v[4]->ts, sp+1->sp, pc+1, <fetch>
```

"*put*" is similarly decoded. The post-decrement of SP unit allows the instruction to be executed in one cycle.

```
<put>  [micro 223]
  alu(fp-arg)->tbus, ts->mW(tbus)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put1>
  ts->v[1], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put2>
  ts->v[2], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put3>
  ts->v[3], mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<put4>
  ts->v[4], mR(sp)->ts, sp-1->sp, pc+1, <fetch>
```

```
<ld>  [micro 235]
  ts->mW(sp+1), sp+1->sp
  mR(arg)->ts, pc+1, <fetch>

<st>  [micro 238]
  ts->mW(arg)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<ldx>  [micro 240]                    ; {ads idx}
  mR(sp)->ff, sp-1->sp                ; pop ads
  alu(ff+ts)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

"*stx*" benefits from the SP unit the most as it pops the stack many times. In the original Sx, "*stx*" takes 7 cycles, now it takes 4 cycles.

```
<stx>  [micro 243]                    ; {ads idx val}
  mR(sp)->nx, sp-1->sp                ; pop idx
  mR(sp)->ff, sp-1->sp                ; pop ads
  alu(nx+ff)->tbus, ts->mW(tbus)
  mR(sp)->ts, sp-1->sp, pc+1, <fetch>

<lit>  [micro 247]
  ts->mW(sp+1), sp+1->sp, arg->ts, pc+1, <fetch>

<jmp>  [micro 249]
  pc+arg, <fetch>

<jt>  [micro 251]
  alu(ts=0), ifT j3                   ; if true, don't jump
<j2>
  pc+arg, mR(sp)->ts, sp-1->sp, <fetch>

<jf>  [micro 256]
  alu(ts=0), ifT j2                       ; if true, jump
<j3>
  pc+1, mR(sp)->ts, sp-1->sp, <fetch>
```

Sx2 breaks *call/fun* into two instructions to reduce the maximum length of any single instruction. The "*call*" instruction saves the return address to *TS* and saves *v* registers. The "*fun*" creates the new activation record and caches the passing parameters from the evaluation stack to *v* registers.

```
<call>  [micro 261]                          ; store  ret ads on ts
  ts->mW(sp+1), sp+1->sp, pc+1               ; flush ts
  pc->ts, arg->pc, if u=0 <fetch>            ; save ret ads
<save v>
  alu(fp-u)->fp, skipu
  v[4]->mW(fp), fp+1->fp
  v[3]->mW(fp), fp+1->fp
  v[2]->mW(fp), fp+1->fp
  v[1]->mW(fp), fp+1->fp, <fetch>

<fun>  [micro 270]                           ; fun.p.u.k
  fp->mW(sp+k), sp+k->sp                     ; save old fp, new sp
  sp->fp                                     ; new fp
  u->mW(sp+1), iru->u, sp+1->sp              ; push u
  pc+1, if p=0 <fetch>
<cache v>
  alu(fp-u)->fp, skipu
  mR(fp)->v[4], fp+1->fp
  mR(fp)->v[3], fp+1->fp
  mR(fp)->v[2], fp+1->fp
  mR(fp)->v[1], fp+1->fp, <fetch>

<ret>  [micro 281]
  sp-1->ff
  alu(fp=ff), ifF <r2>                       ; test for retv
  ts->pc                                     ;  ret ads on TS
  mR(sp)->u                                  ; pop u
  alu(fp-arg)->sp
  mR(sp)->ts, sp-1->sp, if u=0 <r3>          ; if u=0 skip cachev
  mR(fp)->fp, <cachev>
<r2>
  alu(fp+2)->tbus, mR(tbus)->ff              ; ret ads on frame
  ff->pc
  alu(fp+1)->tbus, mR(tbus)->u               ; pop u
  alu(fp-arg)->sp, if u=0 <r3>               ; skip cachev
  mR(fp)->fp, <cachev>
<r3>
  mR(fp)->fp, <fetch>                        ; restore fp
```

In writing the microprogram for the instructions "*inc*" and "*dec*", a different style is used. Instead of decoding to "*inc1*" . . "*inc4*", a test is made to check the range of the argument. If *arg* > *maxv* then it is a normal operation, else the access is on *v* registers. The event "*ifargm*" does the test. The *TS* is saved to *NX* as the operation uses *TS*. When the operation is completed, *TS* is restored from *NX*.

```
<inc>   [micro 300]
  ts->nx, v[arg]->ts, ifargm <inc2>  ; save ts to nx
  alu(ts+1)->ts                      ; op on v reg
  ts->v[arg], nx->ts, pc+1, <fetch>
<inc2>
  alu(fp-arg)->tbus, mR(tbus)->ts    ; a normal op
  alu(ts+1)->ts
  alu(fp-arg)->tbus, ts->mW(tbus)
  nx->ts, pc+1, <fetch>

<dec>   [micro 310]
  ts->nx, v[arg]->ts  ifargm <dec2>
  alu(ts-1)->ts
  ts->v[arg], nx->ts, pc+1, <fetch>
<dec2>
  alu(fp-arg)->tbus, mR(tbus)->ts
  alu(ts-1)->ts
  alu(fp-arg)->tbus, ts->mW(tbus)
  nx->ts, pc+1, <fetch>

<sys>   [micro 320]
<array>
<end>
  trap, pc+1, <fetch>
```

7.5 Performance

Table 7.1 shows the number of cycle used by each instruction. The number in parentheses is the number of cycle of the original Sx for comparison. Please observe that almost all instructions are faster. The "*call/fun*", "*ret*" are slow in the worst case, for example, call+fun is 16 cycles (Sx is only 8 cycles).

"*inc*" and "*dec*" in a normal case are the same as Sx (due to the test for the range of argument) but they are twice as fast if the argument is in the cache register.

Table 7.1 The number of cycle used by each instruction of Sx2. (n) shows the number of Sx.

bop 3 (4)	uop 2 (3)	get 3 (4)	get1..4 2 (4)
put 3 (4)	put1..4 2 (4)	ld 3 (4)	st 3 (4)
ldx 3 (4)	stx 5 (8)	lit 2 (4)	jmp 2 (2)
jt 3 (4)	jf 3 (4)	call max 7 (8)	fun max 9 (0)
ret max 12 (8)	retv max 12 (7)	retv max 12 (7)	inc1..4 3 (6)
dec 6 (6)	dec1..4 3 (6)		

A number of benchmark programs are compiled and then run on the Sx2 processor simulator. The table below reports the number of instruction, the number of cycle and the cycle-per-instruction number for each program.

Table 7.2 The performance of Sx2 processor

	Sx			**Sx2**		
program	noi	cycle	cpi	noi	cycle	cpi
bubble	10068	44214	4.39	10262	32090	3.13
hanoi	2312	10092	4.37	2377	7544	3.17
matmul	3043	12880	4.23	3097	9348	3.02
perm	4868	20932	4.30	4935	14663	2.97
queen	618665	2576210	4.16	620724	1717782	2.77
quick	3172	13539	4.27	3224	9551	2.96
sieve	28026	124338	4.44	28029	75204	2.68
aes	30579	131560	4.29	30724	90498	2.95

The average CPI of Sx2 is 2.9. From the table, comparing the number of clock between the original Sx and Sx2, the average ratio is 0.70. That is, Sx2 is 30% faster than the original Sx.

Other interesting observation is the size of microprogram. Sx2 is obviously more complex. The size of its microprogram is larger. We calculate the size of microprogram as the number of bit in the ROM. Here is the comparison.

Sx width 38 length 62 38x62 = 2356 bits
Sx2 width 71 length 74 71x74 = 5254 bits

Therefore, the complexity in the control unit of Sx2 is double of Sx.

7.6 Summary

To improve the performance of Sx processor, we employ the technique of stack frame caching. The stack frame caching relies on fast registers to cache a part of the stack frame so that the access to these variables takes only one cycle. The separation of SP from the ALU path to have its own increment/decrement, the SP unit, helps to shorten the cycle of the push/pop values from the evaluation stack. There are many approaches to enhance the performance of a processor. In general, the memory sub-system has the major impact on performance. However, in our presentation, the speed of memory, its access time, is assumed to be one cycle, therefore it does not affect our design. This is not a realistic assumption for a general purpose processor but in the context of implementing the design on FPGA with its internal memory block, this is correct.

7.7 Further reading

The conventional approaches to performance enhancement are to use pipeline and multiple functional units. These techniques have been used successfully in every commercial processor available today. Most computer architecture textbook described these methods. The most widely used text written by the computer architects who invent the concept of reduced instruction set computer (RISC), is the text by Hennessy and Patterson [HEN03]. The pipeline technique is perhaps the earliest technique for performance enhancement. It has been used for many complex functional units such as floating-point calculation [KOG81]. Multiple functional units were the landmark of *super computer* in its era. In fact, the first one to employ multiple function units successfully is CDC6600, the most exciting computer architecture of its day [THO70].

References

[CHO06] Chongstitvatana, P., "Stack frame caching", Proc. of National Computer Science and Engineering Conference, Thailand, 2006. (being written)

[HEN03] Hennessy, J., and Patterson, D., Computer Architecture: a quantitative approach, 3rd ed. Morgan Kaufmann, 2003.

[KOG81] Kogge, P., The architecture of pipelined computers, McGraw-Hill, 1981.

[THO70] Thornton, J., Design of a computer: the Control Data 6600, Scott, Foresman and Company, 1970.

Exercises

7.1 Run Sx2, try to write a microprogram for some new instruction and test it.

7.2 Compare the performance with Sx1.

7.3 Discuss the finding; suggest some way to improve the performance by adding some new instruction (counting the total cycle used to complete a task).

7.4 Improve the microstep of some instruction. You don't have to simulate the execution. You can calculate the number clock from the profile.

7.5 If the number of cache registers is changed, for example, 8, what is the impact on the performance?

7.6 The memory latency is one of the most important factors in determining the performance of a processor. Suppose the latency of memory is increased to 2 cycles for read and write to memory. What is its impact on performance? Assume the cache register latency is one cycle.

Chapter 8

Nut Operating System

In this chapter we develop Nut operating system (Nos). This operating system runs on the Sx processor. Nos is a preemptive multitask operating system. This operating system has many services: interprocess communication, shared resources, process synchronisation, and a real time clock. As we have the processor simulator, the details of the implementation of interrupt and task switching can be studied down to the level of machine cycle. A supervisor program (Nos supervisor) is created to interface between Nos and the Sx processor simulator. Nos is designed to be very limited. It does not have virtual memory, the file system nor the network. Although it is very limited, it does offer an insight into the essential part of operating systems.

8.1 Operating system concepts

A process is a basic unit of abstraction to build concurrent execution of multiple programs. A process is a program in execution. A program is a *static* part whereas a process is a dynamic part. One program can be executed by many processes. A process consists of: code segment, data segment, stack segment. They can be shared or separated. When they are shared, only a single address space is needed hence the implementation is simple.

To achieve concurrency using one processor, each process will be allocated a slice of time for its execution. All processes will be scheduled to be executed by time multiplexing. For example, two processes A and B, to run concurrently they will be executed like this:

```
A B A B A B A B
```

A simple programming abstraction to achieve this is *co-routine* where A calls B then B calls A but not starting A at the beginning. A resumes the execution at the point where A has previously stopped.

Several processes can be active at the same time. The concurrency is achieved via multi-threading (light weight process). A heavy weight process is a process with a separate address space. It needs a mechanism to do the address mapping between virtual address and physical address. A light weight process has single address space. A thread is a trace of execution. Concurrency with a single thread process is achieved by co-operative process (via co-routine). A multi-thread process has several traces at the same time. This can be accomplished by pre-emptive scheduler with time-slicing. A light weight process is much cheaper to create than a heavy weight process.

To manage multiple programs, a scheduler keeps a list of all processes. When starting an execution of a time slice, a process will be selected to be run according to some policy. A process is run until it is:

1. time-out using up its allotted time
2. blocked the process requests a resource and has to wait for it.
3. terminated the process runs to its completion.

We can model the behaviour of the process as shown in Table 8.1.

Table 8.1 The state of a process

State	Event	Next state
READY	Task switch	RUNNING
RUNNING	Time-out	READY
RUNNING	Terminate	DEAD
RUNNING	Block	WAIT
WAIT	Wakeup	READY

To execute a process on a processor, the computation state (C-state), i.e. all variables pertain to that process must be save/restore properly upon a task switching. These are PC, FP, SP, and TS including separate stack segment for each process. These values are stored in a data structure called the *process descriptor*. The task-switch is defined as follows.

```
task-switch
   if only one process do nothing
   save current state, set it to READY
   select next process
   make it runnable, set it to RUNNING
```

To understand how a process requests a resource we first study how a resource is shared.

In sharing a resource, it is necessary to ensure *mutual exclusion*. That is, during a period of one process using that resource, other process that also want that resource must wait. Dijkstra invented "semaphore" [DIJ65] to achieve this "mutex" behaviour. A semaphore is a variable (can be binary, 0/1 or an integer) associated with a resource. It indicates availability of the resource to the process that requested it. Two operations are defined on a semaphore: wait, signal (originally Dijkstra called it P, V). Associated with each semaphore is a waiting list of the process that waits on this semaphore. They are defined as follows:

```
Wait sem
   if value of sem <= 0
      block this process, set it to WAIT
      put this process to waiting list of
      this sem
  else
      sem - 1

Signal sem
   if there is process waiting for this sem
      move that process out of waiting list
      wakeup that process
   else
      sem + 1
```

These two operations must be atomic, that is, they must run to completion without interruption (can not task switch in between).

For resource sharing, we let only one process to acquire it, all other processes that request that resource will be waiting in the waiting list. The code where this mutual exclusion is required is said to be *critical section*. A semaphore is used to protect this section. Start with `sem` = 1.

```
....
; critical section
wait sem
... code to access shared resource
signal sem
```

The first process that acquires this section will "close the door". After it finishes with this section, it "open the door" to let other process in. A semaphore is the basis that other mechanism can be built such as process synchronisation or interprocess communication.

Two processes need to be *synchronised* at some point in the program. Two semaphores are used to achieve it. Let two processes be A and B. Assume A arrives to the synchronise point before B. Then A must wait for B and vice versa if B arrives before A.

```
    A                    B

signal sem1          wait sem1
wait sem2            signal sem2
```

The sequence can be rearranged and the behaviour is still correct:

```
   A                    B

signal sem1          signal sem2
wait sem2            wait sem1
```

The interprocess communication is achieved using synchronous message-passing. It combines communication and synchronisation in a single high-level primitive. Other alternative models of message-based process synchronisation are: asynchronous (no wait) and remote invocation.

There are relationships between asynchronous, synchronous and remote invocation semantics. Two asynchronous events can constitute a synchronous relationship if an acknowledgement message is always sent (and waited for). Two synchronous communications can be used to construct a remote invocation. It could be argued that the asynchronous model gives the greatest flexibility but there are a number of drawbacks:

1. Potentially infinite buffers are needed to store messages that have not been read yet.
2. In asynchronous model, more communication are needed, hence programs are more complex.

Also, a synchronous model can emulate an asynchronous communication simply by using a buffer process.

A system is said to be *hard real-time* if it has deadlines that cannot be missed for if they are, the system fails [LEI80]. A system is *soft real-time* if the application tolerant of missed deadlines. A system is *interactive* if it does not have specified deadlines but strives for adequate response times.

Two types of process are present in the real-time domain: periodic and aperiodic. Periodic processes sample data or execute a control loop and have explicit deadlines that must be met. Aperiodic processes (or sporadic) arise from external asynchronous events. These processes have specified response time associated with them. The process must be analysed to give its worst-case execution time, also may obtained average execution time [BUR01].

To schedule real-time tasks, "schedulability" is an important concept. Given a collection of processes and all associated deadlines, determine if this set of processes is schedulable. This means that it is possible for all deadlines to be met indefinitely into the future. In general, necessary and sufficient conditions for schedulability are not known. However, there are many different algorithms presented in the literature which test for schedulability under certain preconditions and restrictions [SHA88] [LEH89].

8.2 Nut operating system

Nut operating system (Nos) is written in Nut. It runs as a user program. Nos models concurrency using process. The share-resource accesses are controlled with semaphores. The processes in Nos communicate with each other through message passing. The crucial real-time functions are supported. Therefore Nos can support real-time tasks. Nos supports the following functions:

- create a process
- terminate a process
- manage the process queue
- task switching
- wait/signal a semaphore
- send/receive a message
- get a real-time clock
- set a timer

8.3 Process

A process is an independent computation which can run concurrently with other process. A process is declared as a normal defined function. Initial values can be passed as parameters at the starting time of a process. A process will end its execution by self-termination when it executes the last instruction at the end of program. This is different from the execution of a function which ends its execution by returning to the caller. A program that calls a process will start that process execution, that program then will continue to work without waiting. A process never returns to its caller.

Each process has its own stack segment. In Nos, there is a single address space, the stack segment of all processes are in the same address space. The advantage is that there is no translation between virtual address and physical address therefore it is fast and simple. The disadvantage is that there is no protection between processes. Each process has its process descriptor (PD) to store the necessary information. A process descriptor in Nos consists of 12 fields.

1. link previous
2. link next
3. process id
4. process status
5. PC
6. SP
7. FP
8. TS
9. in-box
10. await-box
11. message
12. timer

The links previous and next are used to form the list of processes, for example the process queue. The process identifier is a unique number used to label a process. The process status holds the state of process (to be explained later). The fields {PC, SP, FP, TS} hold the computation state of the process. The mail-box (in-box and await-box) is used to communicate between processes. The in-box is the list of processes that sent messages to this process. The await-box is the list of processes that are waiting for messages from this process. The timer holds the timer value of this process.

8.4 Scheduler

When a process is created it is ready to start the execution. Its PD will be linked to the *ready list* (the process queue) which is a doubly linked circular list used by the scheduler. A scheduler has the duty of selecting a process to run from the ready list. The scheduling policy of Nos is a Round-Robin policy where an equal time-slice is allocated for every process and the process is scheduled on first-come first-serve basis. A scheduler will enable a process in the ready list to run until its time-slice is over and then switches to the next process in the list. If a process enters a *wait* state, it is said to be blocked. A process is usually blocked because it performs some operation that requires waiting for another process, such as waiting for the receiver to retrieve a message. When a process is blocked, its PD will be removed from the ready list. The process in *wait* state can be awaken by other process. Its PD will be inserted into the end of the ready list. To perform the switch from one process to another process (task switching), the current state of computation {PC, SP, FP, TS} of the active process is saved in its PD and the state of computation of the process to be run is restored. The first process to be active is the process to run the main program. A process is run until it is time-out, or it is blocked or it is terminated. "switchp" is the task switcher function in Nos. Here is how "switchp" work.

```
switchp
  if status is time-out or blocked
    runnable next task
  else status is terminated
    delete this task
    runnable next task
```

Where status is the state of the process, runnable marks the process as running. Here is the actual code in Nut.

```
(def switchp () ()  [nos 127]
   (do
   (di)
   (if (or (= status TIMEOUT) (= status LOCKED))
      (do
      (setValue activep READY)
      (set activep (getNext activep))   ; switch next
      (runnable activep))
      ; else                            ; status STOPPED
      (do
      (setValue activep DEAD)
      (set activep (deleteDL activep))
      (if (!= activep 0)
         (runnable activep))))))
```

Before we go into more details of the operating system functions, we must understand the basic of running a task first.

8.5 Nos supervisor (Noss)

Nos is executed under Nos supervisor (Noss). Nos supervisor runs on top of the processor simulator. Noss is a privileged program. A privileged program is a program that is "out-of-bound" of user programs. A privileged program provides mechanism for executing a user program, for example, the Sx processor simulator is a privileged program that *executes* S-code. In our implementation, privileged programs are written in C. User programs are written in Nut. The relationship between Nos supervisor and the processor simulator can be understood by regarding Noss as issuing an *interrupt* to the processor.

The processor continuously executes a program (running a process) until an interrupt occurs then the supervisor takes action. The interrupt can occur only at the end of executing an instruction. There are three interrupt events: time-out, stop, block.

time-out − the current process has used up its own time-slice.
stop − the current process runs to completion.
block − the current process is blocked (to be resumed later).

The supervisor (Noss) takes action in response to the interrupt events as follows.

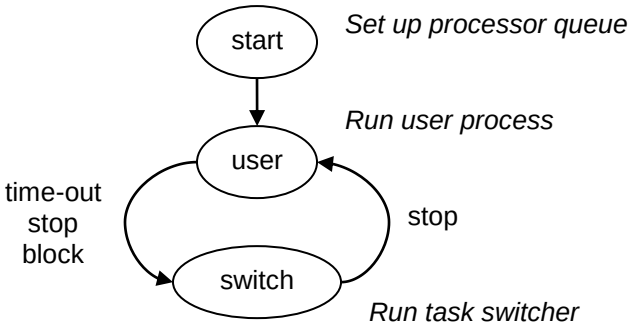


Figure 8.1 Noss state diagram

At start, the main program creates processes and the process queue. Noss schedules only two kinds of processes: user process and switchp. "switchp" is created and has its own PD but it is privileged and never enters the process queue. Once the process queue is ready, at the state user, a user process is run. The user process is run until it is time-out/stopped/blocked. Then Noss calls the "switchp", at the state switch. "switchp" runs to completion. It must not be interrupted as it is manipulating the process queue. Then the state is going back to run a user process.

At the end of task-switch, Noss always runs the next task. This is accomplished by restoring the computation state (C-state) of that process. This means the PC, SP, FP, TS of that process are restored to the processor simulator and then the processor simulator continues to execute until the interrupt occurs.

The process descriptor contains C-state. Saving and restoring C-state are the act of transferring C-state between the processor simulator and the process descriptor. Noss does the restoring of C-state. This restoring will affect the flow of execution, as the instruction pointer is changed; it does a jump in the program. As Noss is responsible to run the task-switcher, it must save C-state. Saving C-state can not be done in user-space as the precise state has been changed when trying to run the "saving state" function. So, save-C-state is done in the main

loop of Noss (in C). This gives the save-C-state a special privilege (so called *kernel* in OS vocabulary). The following pseudo code described Noss.

```
Noss  [noss 43]
  if there is no process in the queue
     stop the whole simulation
  else
     update status to nos
     if state = switch
        save current user process
        restore "switchp"
        state = user
     else state = user
        restore active process
        state = switch
```

8.6 Simulation of interrupt

The processor simulator always runs a program in a tight loop. The processor fetches an instruction and executes it. To simulate an interrupt, the processor calls Noss from time to time (this is called *yield*). The interrupt interval is controlled by counting the cycle used since the last call to Noss.

The processor returns the control back to Noss after three conditions:

1. Its time-slice has been used up. This is called "time-out".
2. The process has been blocked by executing some operation. This is called "blocked".
3. The task is completed. The program reached "end". This is called "stopped".

When the processor hands the control back to Noss, Noss calls task-switch. The task-switch code is in the user-space. The task-switch is the function "switchp". "switchp" requires the knowledge of the status of the completion of the previous task: time-out, blocked, or ended. This information is provided by Noss via the variable "status", as Noss controls the processor simulator it knows how the task has returned the control.

Noss is minimal in the sense that it does not do a lot of things by itself. The only thing it does is to call "switchp". Noss monitors the state of computation of a process through two global variables: status and activep.

8.7 Processor simulator

In the processor simulator, the main simulation loop is "eval". It executes a fixed number of cycles. This is the main fetch-execute cycle of the processor (in fact most processor simulator are like this):

```
eval
  count = 0
  loop
    if count > limit break
    fetch an instruction
    execute the instruction
    count = count + 1
```

To implement interrupts, a flag (intflag) is used to disable the break. This flag can be turned on/off by the system calls.

```
eval2
  count = 0
  loop
    if intflag == 1
      if count > limit break
    fetch an instruction
    execute the instruction
    count = count + 1
```

The system calls that support Nos are:

20 disable interrupt
21 enable interrupt
22 block a process

The following code describes the main loop of the processor simulator.

```
Eval   [noss 71]
  set PC
  while runflag = 1
    run one instruction
    yield

yield
  if intflag = 1
    if no of cycle used > TIMEOUT
       noss(time_out)
```

Where runflag controls the termination of Noss itself, intflag is the interrupt flag used to disable/enable interrupt.

8.8 How a process is created

A function "run" is used to create a process and put it to the process queue. Although "run" looks like a normal function, it can not be compiled into a normal function call. The argument to "run" is a function which will be turned into a process so it should not be evaluated. A call to "run" is compiled into the code that passes an address of the function. The argument to "call.run" in N-code is just a user-function call with its arguments as usual. However, this argument will not be evaluated. Instead, the address of the code of this call will be generated as an argument of "run". This code will be activated by the scheduler as a process. See the following example:

```
(def add (a b) () (+ a b))
(def run (f) () 0)
(def main () ()
   (do
   (run (add 4 5))))

add
(fun.2.2 (+ get.1 get.2 ))
run
(fun.1.1 (lit.0 ))
main
(fun.0.0 (do (call.17 (call.14 lit.4 lit.5 ))))
```

The generate S-code is as follows. See line 12-18.

```
 1 Call main
 2 End
 3 Fun add
 4 Get 2
 5 Get 1
 6 Add
 7 Ret 3
 8 Fun run
 9 Lit 0
10 Ret 2
11 Fun main
12 Lit 15
13 Call run
14 Jmp 19
15 Lit 4
16 Lit 5
17 Call add
18 End
19 Ret 1
```

The line "(run (add 4 5))" becomes

```
12 Lit 15       address of code (add 4 5)
13 Call run
14 Jmp 19       do not execute now
15 Lit 4        the code (call.add lit.4 lit.5)
16 Lit 5
17 Call add
18 End
19 . . .
```

8.9 How to generate code for run

The S-object must contain the symbol table with the correct references. The S-object is generated by "gen.txt". However, "gen.txt" just passes the symbol table through. The symbol table is read from N-object. The N-object is generated by "nut.txt", the compiler. The current version dumps everything in the symbol table.

The symbols that must be exported are of type FUN and GVAR only. The following tasks must be done.

1. Change “nut.txt” to output only the necessary symbols.
2. Change “gen.txt” to output the S-code reference. However the number of symbol does not change.

at nut.txt

The dumpsym is responsible to output the symbol table. It also relocates the references to functions such that the code segment starts at 2. It is necessary as Nut-compiler is used under “nvm” where both the compiler and the user program to be compiled occupy the same code segment. Therefore the user program in the code segment will not start at 2. We would like the object to be relocatable; therefore the user code should start at 2.

When starting the compiler, (sys 9) is used to find out where to user code segment is. The global variable “Start” stores this location, and it is used to relocate the reference to all function call when output the object. The following code is added in “nut.txt” at dumpsym, to output the correct reference. The data segment is not relocated as it is already started at 0.

```
(if (= ty tyFUN)
   (set n (shift (getVal i) Start))
   ; else
   (set n (getVal i)))
   (print n) (space)
```

at gen.txt

Here is the added code to outsym, to output the symbol table.

```
(set ty (atoi tok))
(tokenise)                ; ref, reloc
(if (= ty tyFUN)
   (do
   (set ref (shift (atoi tok) CS))
   (print (assoc ref)))
   ; else
   (prstr tok))
  (space)
```

When reading the symbol table from N-object, the generator recognises the type "fun" and outputs the S-address corresponding to the N-address.

To generate the code for the expression (run (fn ...)), the code to call "run" is generated and the address pointed to (fn ..) is generated as its argument.

```
    lit x
    call run
    jmp y
x:  ...
    call fn
    end
y:  ...
```

The address of *x* is at the next 3 words. *jmp y* skips the code (fn...). The call to (fn ..) is deferred and "*run*" will use *x* as the starting address of the process which calls (fn...). When the process returned, it will be terminated by "*end*". This is in the function gencall.

```
; convert arg to index to symtab
; e is arglist
(def gencall (arg e) (idx a)
   (do
   (set idx (searchRef arg))
   (if (= idx Runidx)              ; is "run"
      (do
      (outa icLit (+ XP 3))        ; point to code of process
      (outa icCall idx)            ; call run
      (set a XP)
      (outa icJmp 0)
      (eval (head e))
      (outs icEnd)
      (patch a (- XP a)))          ; jump over
      ; else
      (do                          ; normal call
      (while e
         (do
         (eval (head e))
         (set e (tail e))))
     (outa icCall idx)))))
```

Example session

The following example shows how a process is created and run. A user program is written as (count) and integrated with Nos in "main":

```
; ---- application --------

(def count (n) (i)
   (do
   (set i 0)
   (while (< i n)
      (do
      (set i (+ i 1))
      (print i) (space)))))

(def main () (p)
   (do
   (sys 5)
   (set activep 0)
   (set sseg 1000)
   (set p (run (count 500)))
   (bootnos)))
```

The line (run (count 500)) creates a process to run (count 500). (bootnos) starts the process running.

To run NOSS, first compile user functions with NOS in nos.txt:

```
e:\test>nut < nos.txt
print
(fun.1.1 (sys.1 get.1 ))
printc
(fun.1.1 (sys.2 get.1 ))
nl
(fun.0.0 (sys.2 lit.10 ))
space
(fun.0.0 (sys.2 lit.32 ))
not
(fun.1.1 (if get.1 lit.0 lit.1 ))
. . .
```

Then run NOSS with the executable, let it be “a.obj”.

```
e:\test>noss a.obj
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49
*
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
66 67 68 69 70 71 72 73 74 75 76
. . .
*
487 488 489 490 491 492 493 494 495 496 497 498
499 500
*
9345 clocks

e:\test>
```

The “*” indicates the task-switching (every 1000 cycles).

8.10 Interprocess communication

Nos provides two ways to communicate (passing some values) between processes:

1 by share variables
2 by message passing

Share variables

A semaphore is used to provide mutual exclusion of access to share variables. The share variable will be accessed by only one process at a time. (Remember that processes can be concurrent therefore at any time there can be more than one process trying to access the same variable). A semaphore is implemented as a special global variable with two fields: value, wait-list. The access to a semaphore is done via two functions: signal, wait. They are atomic operations. The operation runs to completion without interrupt. This is achieved by *disable interrupt* at the beginning of the function and *enables interrupt* before return.

```
;  semaphore field: sval(value) slist(wait-list)

(def signal (s) (p)  [nos 166]
   (do
   (di)
   (set p (getslist s))
   (if (!= p 0)
      (do
      (setslist s (deleteDL p))
      (wakeup p))
      ; else
      (setsval s (+ (getsval s) 1)))
   (ei)))

(def wait (s) (v p)  [nos 178]
   (do
   (di)
   (set v (getsval s))
   (if (<= v 0)
      (do                              ; block activep to WAIT
      (set p activep)
      (set activep (deleteDL activep))
      (setValue p WAIT)                ; to wait-list
      (setslist s (appendDL (getslist s) p))
      (blockp))                        ; block
      ; else
      (setsval s (- v 1)))
   (ei)))
```

Where "blockp" blocks the current process (and calls the supervisor), "wakeup" puts the process p in the process queue, ready to be scheduled to run.

```
(def initsem (v) (s1)  [nos154]
   (do
   (set s1 (new 2))
   (setsval s1 v)
   (setslist s1 0)         ;; wait-list nil
   s1))
```

```
(def wakeup (p) ()  [nos 161]
   (do
   (setValue p READY)
   (set activep (appendDL activep p))))
```

A "monitor" can be constructed to provide an abstract data type to protect shared variables. The access is done via the parameter "cmd". The monitor uses the associated semaphore to perform *mutual exclusion* access. Only one process can be inside the monitor at one time.

```
(def monitor (cmd) ()
   (do
   (wait sem1)
   (if (= cmd 1)
     ... access shared variables
     ; else
     ... access shared variables
   (signal sem1)))
```

8.11 Message passing

The message passing in Nos is implemented as a blocking protocol where the sender and receiver wait until the exchange is completed before continuing. This is done using two mail-boxes: in-box and await-box. Here is the pseudo code for the "send" and "receive" operations.

```
send p mess
  if there is a process p wait for it
    put mess to p's buffer
    wakeup p
  else
    block itself
    append itself to p's in-box

receive p
  if there is a process p mail in in-box
    take the message from p's buffer
    wakeup p
  else
    block itself
    append itself to p's await-box
```

The Nut program implementing "send" and "receive" is as follows.

```
; p is pointer to process
(def send (p mess) (m box)  [nos 221]
   (do
   (di)
   (set box (getAwait activep))
   (set m (findmail p box))
   (if (= m 0)
      (do
      (set m activep)            ; self
      (setMsg m mess)
      (set activep (deleteDL activep))
      (setMbox p (appendDL (getMbox p) m))
      (setValue m SEND)
      (blockp))
      ; else
      (do                        ; p is waiting
      (setMsg p mess)
      (set m (deleteDL p))
      (if (= box p)
         (setAwait activep m))
   (wakeup p)))
   (ei)))

(def receive (p) (m box)  [nos 243]
   (do
   (di)
   (set box (getMbox activep))
   (set m (findmail p box))
   (if (= m 0)
      (do                        ; put to await p
      (set m activep)            ; self
      (set activep (deleteDL activep))
      (setAwait p (appendDL (getAwait p) m))
      (setValue m RECEIVE)
      (blockp)
      (getMsg m))                ; retrieve from self
      ; else
      (do                        ; already in mbox
```

```
    (set m (deleteDL p))
    (if (= box p)
       (setMbox activep m))
    (getMsg p)                    ; retrieve mbox
    (wakeup p)))
 (ei)))
```

There are two buffers, one in the sender and other in the receiver. The process descriptor is attached to the in-box/await-box so that waking up a process associated with the mail is simple. "findmail" searches for a message from a process p in the mail-box. "blockp" blocks the current process (and calls the supervisor). The state "SEND/RECEIVE" indicate that the process is blocked by the send/receive operation. "wakeup" puts the process p in the process queue, ready to be scheduled to run.

Example of use of send/receive message

We write two functions, one is the producer that sends the message *2..n,* the other is the consumer. The consumer receives the message until the end of message is reached (-1).

```
(def produce (n) (i)  [nos 270]
   (do
   (set i 2)
   (while (< i n)
      (do
      (send p2 i)
      (set i (+ i 1))))
   (send p2 (- 0 1))))

; receive 2..n from p1 ended with -1
(def consume () (m flag)  [nos 281]
   (do
   (set flag 1)
   (while flag
      (do
      (set m (receive p1))
      (if (< m 0)
         (set flag 0))))
   (nl)))
```

Create and run producer and consumer.

```
(def main () ()  [nos 292]
   (do
   (di)                          ; disable interrupt
   (set activep 0)               ; init task-list
   (set sseg 1000)               ; init stack segment
   (set pid 1)                   ; init process id
   (set psw (run (switchp)))
   (set activep 0)
   (set p1 (run (produce 1000)))
   (set p2 (run (consume)))
   (bootnos)))
```

Suppose a producer streams the messages (integers) *2..n* to a consumer. The producer's output is marked "`!n`" and the receiver's output is marked " `"n`". The task-switched is marked "*". The trace is:

```
!2  * "2  * !3 !4  * "3 "4  * !5 !6  * "5 "6  *
!7 !8  * "7 "8  * !9  *  "9 . . .
```

Let two processes be s, r. This behaviour can be explained by inspecting the trace of execution of two processes.

notation

sM send in-box
sA send await
rM receive in-box
rA receive await
sB sender block
rB receiver block

The trace is:

1 producer: sM sB *
2 consumer: rM rA rB *
3 producer: sA sM sB *
4 consumer: rM rA rB *
. . .

The first line says that the sender just sent a message to the receiver's in-box then itself is blocked. The second line is quite interesting. It says the receiver retrieves the message from the sender's buffer and then continues to execute its program which does "receive p". This call forces the receiver to send itself to the sender's await-box, and then itself is blocked. This mean "r" is waiting for a message from "s". Once "s" wakeups "r", the process "r" will have a message in its buffer. Line 3, 4 can be similarly explained.

This benchmark has been compiled with all optimisation turned on (macro, primitives, extended instructions). Sending and receiving 1000 messages take total 171037 instructions. Therefore the number of instruction for passing (send and receive) one message is 171037/1000 = 171 instructions/message.

8.12 Timer

To facilitate a real-time system, some operating system functions needed to be supported. In our system, the real-time clock is the clock of running the processor. The function

(gettime)

returns the real-time clock. The function

(timer t)

sets a timer to be time-out at t cycles in the future, not earlier than (gettime)+t. A timer is used to schedule a task according to some real-time deadline.

How a timer is implemented?

A timer stores its time value as a field in PD. A timer list keeps track of the processes that have been scheduled to time-out in the future by "timer". The time value in PD is an absolute time. When a timer is set to t, the time value in PD is set to (gettime)+t. The process that executes "timer" is blocked. It is removed from the process queue and it is added to the timer list. The timer list is sorted according to the time values from earliest time to the latest. This list will be processed by a timer process which is scheduled by the supervisor, Noss.

Timer process

The time value in the list is compared to the master time (the global variable clock in the processor simulator). If it is less than the master time the owner process of this timer is awaken. As the timer list is sorted in ascending order of time value, only the first one is consulted if it is time-out then the next one is consulted and so on.

Granularity of timer

How precise the timer is depends on how often the timer process is scheduled to run. The overhead depends on this rate. It is reasonable to have the granularity at most the same as "quanta", the time interval of the interrupt. Then, the timer process can be scheduled to run after the task switcher.

What Noss needs to do?

Noss needs to run "timer" after "switchp". The time-out timer process will be queued either at the front of the process queue or the back depends on the scheduling policy. To simulate the real-time, if the timer list is not empty and the process queue is empty, then the first process in the timer list should be scheduled to be run. The master time should be updated to advance to the time value of that process. This is similar to an ordinary event-driven simulation based on time.

8.13 Lab session

We run two processes sharing two variables through a monitor. The monitor has two functions: 1) increment the value, and 2) getting the value. The first process increments the value and waits for the second process to get the value. This procedure is repeated until 20 times. To synchronise both processes so that incrementing/getting value will be "in sync", another variable, "empty" is used to signal whether the value has been used. The monitor protects these global variables. The program for the experiment is shown below.

```
(let ff empty)          ; shared variables
(let sem1)              ; semaphore

(def mon1 (cmd) ()
   (do
   (wait sem1)
   (if (= cmd 1)
      (if (= empty 0)
         (do
         (set ff (+ ff 1))
         (set empty 1)))
      ; else
      (if (= empty 1)
         (set empty 0)))
      ff
   (signal sem1)))

(def inc () (i n)
   (do
   (set i 0)
   (while (< i 20)
      (do
      (set n (mon1 1))
      (set i (+ i 1))))))

(def pff () (n)
   (do
   (set n (mon1 2))
   (print n)
   (while (< n 20)
      (do
      (set n (mon1 2))
      (print n)))))

(def main () (p1 p2)
   (do
   (di)
   (set activep 0)
   (set sseg 4000)
   (set pid 1)
   (set psw (run (switchp)))
```

```
(set activep 0)
(set ff 0)
(set empty 0)
(set sem1 (initsem 1))
(set p1 (run (inc)))
(set p2 (run (pff)))
(bootnos)))
```

Append the above program to Nos, name it "nos1.txt". Compile it and generate an executable code, let is be "ns1.obj". Then run it under Noss.

```
c:\prabhas\test> nvm nut.obj < nos1.txt > nos1.obj
c:\prabhas\test> nvm gen2.obj < nos1.obj > ns1.obj
c:\prabhas\test>noss ns1.obj
load program, last address 503
DP 1008
 *  *  *  * 1 *  * 2 *  * 3 *  * 4 *  * 5 *  * 6 *  * 7 *  *
8 *  * 9 *  * 10 * * 11 *  * 12 *  * 13 *  * 14 *  * 15 *  *
16 *  * 17 *  * 18 *  * 19 *  * 20

6592 inst. 33289 cycles (system 12009 user 21280) switchp 42
```

The "*" shows the task switch to run user processes. The output shows that two processes synchronised properly. There are 42 task switching, 40 comes from switching between two processes, each 20 times.[1] The system cycle reports the number of cycle used in the Nos itself. The user cycle reports the number of cycle used to actually running the user process, (inc) and (pff). You can observe that the system consumes about half of the cycles.

8.14 Summary

In this chapter we have developed an operating system, Nos. The operating system is preemptive. It supports multi-thread. Two simple interprocess-communication methods have been implemented: semaphore and message passing. Some facilities for real-time processes are outlined.

[1] What does two other task switching come from? This question is left to be investigated by the interested reader

It is not a surprise about the fact that Nut language can be extended minimally to write the whole operating system. The design of the extension is critical. Using the model of interrupt is a good framework to implement a simulator to run the operating system. The supervisor program, Nos supervisor (Noss), mediates between Nos and the processor simulator, Sx. The processor runs its user program continuously until an interrupt event occurs, then it hands back the control to the supervisor. The supervisor, Noss, performs the task of saving/restoring the computation state of the process to/from the process descriptor. Noss does a minimal job of intervention. Majority of the task switching and other operating system service functions are done in the user-space by Nos.

Semaphore, monitor and messaging are progressive development toward a higher abstraction which is easier to use. The behaviour of these services can be observed. The implementation is short and simple enough to be experimented with. The processor simulator gives us the detail at the level of cycle-by-cycle execution such that the effect of the system as a whole can be studied.

8.15 Further reading

Operating systems have been developed over the past 50 years. The major breakthroughs in operating system technology from the 1950s to 1990s have been collected in the book by Hansen [HAN01]. The earliest time-sharing systems were the Compatible Time-Sharing System (CTSS) developed at MIT [COR62] and The Multiplexed Information and Computing Services (MULTICS) [COR65]. Many textbooks cover operating systems, including Stallings [STA00], Tanenbaum [TAN01], and Silberschartz et al [SIL03].

References

[BUR01] Burns, A. and Wellings, A., Real-time systems and programming languages, 3rd ed. Addison-Wesley, 2001.

[COR62] Corbato, F., Merwin-Daggett, M., and Daley, R., “An experimental time-sharing system”, Proc. of the AFIPS Fall Joint Conference, pp.335-344, 1962.

[COR65] Corbato, F., and Vyssotosky, V., "Introduction and overview of the MULTICS system", Proc. of the AFIPS Fall Joint Computer Conference, pp.185-196, 1965.

[DIJ65] Dijkstra, E., "Solution of a problem in concurrent programming control", Communication of the ACM, 8(9):569, 1965.

[HAN01] Hansen, P. (ed.), Classic Operating Systems, Springer-Verlag, 2001.

[HOA74] Hoare, C.A.R., "Monitors : an operating system structuring concept", Comm. ACM, 17(10):549-557, 1974.

[LEH89] Lehoczky, J.P., Sha, L. and Ding, Y., "The rate monotonic scheduling algorithm – Exact characterization and average case behavior", Proc. IEEE Real-time Systems Symp., pp. 166-171, 1989.

[LEI80] Leinbaugh, D.W., "Guaranteed response time in a hard real-time environment", IEEE trans. on software engineering, January 1980.

[SHA88] Sha, L., An overview of real-time scheduling algorithms, Software Engineering Institute, Carnegie Mellon University, 1988.

[SIL03] Silberschatz, A., Galvin, P., Gagne, G., Operating System Concepts, 6th ed. John Wiley, 2003.

[STA00] Stallings, W., Operating Systems, 4th ed. Prentice Hall, 2000.

[TAN01] Tanenbaum, A., Modern Operating Systems, Prentice Hall, 2001.

Excercises

8.1 Vary the quanta and observe the behaviour of Nos running some applications.

8.2 Write timer and its associated function, gettime.

8.3 Implement producer/consumer processes with a buffer of size *n*.

8.4 How to create and destroy a process dynamically?

8.5 How to improve the performance of Nos?

8.6 Nos has no input, to allow concurrency, the input can be simulated through the console application. A console accepts input and feed it as a stream to the receiving process. Write a console application.

8.7 Discuss the co-operative process. How can it be implemented? Co-operative process can be implemented at a lower cost than the pre-emptive OS. Write co-operative process in Nut and discuss its cost.

8.8 Nos has a single address space. To protect resources used by a process, a virtual memory is necessary. Discuss how to implement a virtual memory under our framework.

8.9 To implement interrupt properly, we rely on Noss as a privilege process. Noss is written in C and works in cooperation with the Sx processor simulator. Is it possible to write Noss in Nut as a user program?

Chapter 9

Optimisation

In this chapter, we will study many methods to improve the performance of the system that we have built in the previous chapters. The system consists of some simple applications running on a concurrent operating system written in a high level language. The platform (the hardware system, a virtual one) is a stack-based processor. It is microprogrammable, so its instruction set can be extended. The high level language itself can also be changed. We have written the compiler for the language and the code generator to generate code for the target machine. These programs can be modified.

Studying a computer system as a whole can reveal the relationship between components. Their interactions can be complex and interesting. The optimisation aims to improve mainly the performance, to complete a task with fewer numbers of cycles. As we built all components by ourselves, we can change the system at every level. We can experiment with any component and observe the change. We can instrument our system to collect statistics easily.

There is no separate *lab session* in this chapter as the work is spread out in all sections. Each optimisation method will be tried and data collected. The analysis follows each experiment.

9.1 Framework

There are many levels in which to aim an optimisation for. The highest level is an algorithmic level. We will not explore this topic; instead we refer to many excellent textbooks in the field [COR01] [PRA01] [KLI05]. What we will explore is at a more concrete level: the language level, the code generation level and the microprogramming level.

At the language level, the macro expansion will be studied. The extension of the language itself to include new operators will be tried. At the code generation level, many techniques will be investigated: supporting new instructions such as increment, decrement (which are implemented in the previous chapter), improving some simple sequence of code, and eliminating some code. At the instruction level, a few new instructions will be designed and implemented. The microarchitecture level has been demonstrated in the chapter 7, showing an improving data path of the Sx processor.

What are we going to measure and how?

To observe any improvement we need to set up a controlled environment. Several methods will be applied to improve a system that performs the same task. The effects of these different methods can then be observed and compared. The benchmark programs are a set of programs (or tasks) representing the kind of workload that we expect in our work. We elected two programs as our workload representatives: the Nut compiler and the Nut operating system.

The first benchmark is the compiler benchmark. The original Nut compiler, "nut.txt", is used to compile itself. This represents a substantial work that is moderately complex and contains many well-known problems in computer science. Nut compiler is also a non-trivial program that will exercise a large repertoire of instructions.

The second benchmark is the message passing benchmark. An application program performs producer/consumer type of behaviour. It sends and receives 100 simple messages. This benchmark tests the operating system, the task switcher and represents the fundamental operation in the operating system, the interprocess communication. The program is "nos2.txt". It contains Nos and the messaging services: send, receive. There are 200 task switches.

The data collection consists of the profile of running the benchmark programs. Two statistics are collected:

1. Frequency of each instruction used
2. Frequency of each line of program used

The statistic 1 let us know what instruction to improve. The statistic 2 let us know where the programs spend its time.

Tools

The set of tools that will be used are:

1 The original Nut compiler, the source is "nut.txt"; the N-object code is "nut.obj".
2 The code generator, the source is "gen.txt"; the N-object is "gen.obj".
3 The evaluator of N-object, the Nut virtual machine, nvm. It is an executable code running on a real computer.
4 The Sx processor, its simulator is used to execute the S-object which is considered to be the "grounded" level for measuring the number of cycle used to run benchmark programs.

Baseline

We establish the *baseline* data to be compared with the result of the methods suggested in this chapter.

This is the profile of the compiler benchmark. Let "nuts.obj" be the S-code of Nut-compiler, "nut.txt". We run the following task to collect the statistic.

```
c:>sx nuts.obj < nut.txt
```

The base Nut-compiler compiles the original Nut-compiler. "`sx`" will output a profile file, "prof.txt" showing the frequency of each instruction used and the frequency of each line of program used. Table 9.1 shows the profile of the number of instruction used the functions. Only the functions that consume more than 100 (x1000) instructions are shown. The total number of instruction executed in this benchmark is 8585 (x1000).

Table 9.1 The number of instruction (x1000) used in functions in the compiler benchmark (anything less than 50 is not shown)

!=	738
and	963
str=	4414
getName	416
install	1456
	7987

Observation

In terms of functions that consume most of cycle, the "str=" is the first one, followed by "install", "!=", "and", "getName". They are summed up to 93% of total instruction executed. The "str=", string comparison function, alone consumes 50% of cycle. This function is used almost entirely on the task related to the symbol table. This fact suggests that we should concentrate on supporting this function in machine instructions. We should also consider changing the access methods of the symbol table.

The next step is to collect the data of the message passing benchmark. The profile of running nos2 is shown in Table 9.2.

They are accounted for 82% of total number of instructions executed. The total number of instruction executed in this benchmark is 43580.

The functions in the table are mostly the functions accessing data structure in the form (vec a n), (setv a n), where a is a local variable, n is a constant.

Table 9.2 The number of instruction used in functions (anything less than 1000 is not shown) in the message passing benchmark.

!=	1993
or	1295
ei/di	1194
getNext	1992
setNext	2020
setPrev	2020
appendDL	3413
deleteDL	2700
setValue	3005
switchp	3804
findmail	3465
send	3267
receive	2772
produce	1581
consume	1395
	35916

9.2 Macro expansion

To reduce the overhead of a function call, a function can be defined as a "macro". A macro definition is just like a function definition; the difference is that the body of a macro definition is substituted into the call. Hence, the size of a program with a macro is larger. The advantage is that it will be executed faster. The syntax of a macro definition is similar to a function definition, only the keyword is "defm" instead of "def". For example,

```
(defm print (x) () (sys 1 x))
```

Whenever the macro appears in the program, the macro body is substituted. In the following definition, "print" will be substituted.

```
(def report (a) ()
   (print a)
   ...
```

The expression will become

```
(def report (a) ()
   (sys 1 a)
   ...
```

Macro is suitable for defining the access function such as the following functions (from symbol table access functions in "nut.txt")

```
(def getName idx () (vec symtab idx))
(def getType idx () (vec symtab (+ idx 1)))
(def setName (idx nm) () (setv symtab idx nm))
(def setType (idx ty) () (setv symtab (+ idx 1) ty))
```

These functions will be executed much faster because there is no overhead associated with "call" and "return" such as create/destroy the stack frame. Our macro definition cannot have local variables because the local variables in the body of the macro must be appended to the list of local variables of the caller, that is, extending the environment of the caller. We opt to demonstrate only a simple macro substitution without changing the caller environment.

The Nut-language is extended to have macros. The new keyword "defm" is recognised by the extended compiler, "nut4.txt". The macro definition will be parsed as a normal function definition, only that its type will be "macro" (instead of "func"). The expression that called the macro can be recognised by inspecting its type. For the macro call, the body of macro will be expanded with the proper binding of actual parameters to the formal parameters defined in the macro definition.

The main function of the macro expansion is the function "subst". The function "subst" takes apart the body of macro definition one by one element, and maps that item to the corresponding element in the actual parameter list using "mapATOM".

```
; do macro expansion
(def domacro (nm e1) (arg body)
   (do
   (set arg (de_arg nm))
   (set body (pick (getVal arg) 2))
   (subst body e1)))

; e1 is the body of macro def, e2 is the actual arg list
(def subst (e1 e2) (e)
   (if (= e1 NIL) NIL
      ; else
      (do
      (set e (head e1))
      (if (isATOM e)
         (cons (mapATOM e e2) (subst (tail e1) e2))
         ; else
         (cons (subst e e2) (subst (tail e1) e2))))))
```

Where (pick e n) gets the n-th element of the list e. Most of the work is done in "mapATOM", where the term "get.a" or "put.a" in the body of macro definition is substituted with the corresponding actual parameters from the argument list. The following rules are the rules for substitution:

```
mapATOM a e2
   e3 = (pick e2 n)
   if a = get.n  out e3
   if a = put.n
     if e3 = get.n out put.n   local
     if e3 = ld.n  out st.n    global
   if a = ldx.n
     if e3 = get.n out ldx.n   local
     if e3 = ld.n  out ldy.n   global
   if a = stx.n
     if e3 = get.n out stx.n   local
     if e3 = ld.n  out sty.n   global
   otherwise
     out a                     do not substitute
```

Where a is the atom in the macro body, e2 is the argument list of the caller. The function (pick e n) select n-th element of e.

The function "mapATOM" and its related functions are shown below.

```
; return n-th element of e
(def pick (e n) ()
   (if (= e NIL) NIL
   (if (< n 1) NIL
   (if (= n 1) (head e)
   (pick (tail e) (- n 1))))))

; return op1/op2 depends on e3 is get/ld
(def map2 (e3 op1 op2) (e n)
   (do
   (set n (de_arg e3))
   (if (isOp e3 xGET)(set e (mkATOM op1 n))
   (if (isOp e3 xLD)(set e (mkATOM op2 n))
   (error "no ld/get in caller macro expansion")))
   e))

; map atom a with n-arg in e2, e2 is the actual arg list
(def mapATOM (a e2) (e e3 c n)
   (do
   (set c (de_op a))
   (set n (de_arg a))
   (set e3 (pick e2 n))                ; n-arg of caller
   (if (= c xGET) (set e e3)
   (if (= c xPUT) (set e (map2 e3 xPUT xST))
   (if (= c xLDX) (set e (map2 e3 xLDX xLDY))
   (if (= c xSTX) (set e (map2 e3 xSTX xSTY))
   (set e a)))))                       ; no substitution
   e))
```

Example

```
(defm inc2 (a b) ()
   (a = a + b))
```

The expression (inc2 x y) will be expanded as follows. The macro body is:

(put.a (add get.a get.b))

The atoms in the macro body are: 1) put.a 2) add 3) get.a 4) get.b. Let the arguments be two cases: locals, globals, then (inc2 x y) is compiled to:

1. x, y are locals, the call expression is (call.inc2 get.x get.y)

   The substitution for each atom is as follows.
   atom put.a map to get.x, out put.x
   atom add don't map, out add
   atom get.a map to get.x, out get.x
   atom get.b map to get.y, out get.y

   The output is (put.x (add get.x get.y))

2. x, y are globals, the call expression is (call.inc2 ld.x ld.y)

   The substitution for each atom is as follows.
   atom put.a map to ld.x, out st.x
   atom add don't map, out add
   atom get.a map to ld.x, out ld.x
   atom get.b map to ld.y, out ld.y

   The output is (st.x (add ld.x ld.y))

Similarly for the atom "ldx" and "stx". The atom in the actual parameters can not be global as it is a call by value.

How to do macro

The Nut-compiler ("nut.txt") is modified to expand macro ("nut4.txt"). The compiler itself (the target) has changed almost all the access functions to be the macros ("nutm.txt"). The compiler, nut4.txt, is used to compile the macro-version of compiler ("nutm.txt"). Then the macro-version compiler is used to compile the original, "nut.txt" for benchmarking. This is the step of the work.

1 Produce the compiler that can expand macro.

```
c:>nvm nut.obj < nut4.txt > nut4.obj
```

2 Use this compiler to compile the macro-version of compiler

```
c:>nvm nut4.obj < nutm.txt > nutm.obj
```

3 Generate S-code for “nutm.obj”.

```
c:>nvm gen.obj < nutm.obj > nutms.obj
```

The final executable code is the compiler with most access functions inlined, “nutms.obj”. We use this compiler to compile the original Nut-compiler to collect profiling statistics.

4 Run the compiler benchmark

```
c:>sx nutms.obj < nut.txt
```

The second benchmark is similar.

1 First, compile the macro-version of Nos, “nos2m.txt”.

```
c:>nvm nut4.obj < nos2m.txt > nos2m.obj
```

2 Produce the executable code.

```
c:>nvm gen.obj < nos2m.obj > ns2.obj
```

3 Run Nos (with most access functions inlined) under Noss.

```
c:>noss ns2.obj
```

The result from macro expansion (inline) to eliminate calls is shown in Table 9.4. In the compiler benchmark, using macro is 30% faster (in terms of cycle), and 46% faster in the message passing benchmark.

Table 9.3 The frequency of each instruction used

	compiler instr. X1000					message passing instr.				
	base	macro	prim	codegen	extend	base	macro	prim	codegen	extend
Add	244	244	244	121	121	107	107	107	9	9
Sub	4	4	4	4	4	1	1	1	1	1
Mul	1	1	1	1	1	0	0	0	0	0
Div	0	0	0	0	0	0	0	0	0	0
Band	13	13	227	227	227	0	0	0	0	0
Bor	0	0	2	2	2	0	0	200	200	200
Bxor	0	0	0	0	0	0	0	0	0	0
Not	0	0	0	0	0	0	0	0	0	0
Eq	379	275	256	228	220	1496	1397	1197	798	400
Ne	0	0	123	120	0	0	0	299	0	0
Lt	112	111	112	112	112	198	198	198	198	198
Le	0	0	0	0	0	0	0	0	0	0
Ge	0	0	0	0	0	0	0	0	0	0
Gt	1	1	1	1	1	0	0	0	0	0
Shl	2	2	2	2	2	0	0	0	0	0
Shr	4	4	4	4	4	0	0	0	0	0
Mod	0	0	0	0	0	0	0	0	0	0
Ldx	364	364	364	364	107	1094	1094	1094	1094	0
Stx	6	6	6	6	1	1727	1727	1727	1727	0
Ret	629	132	289	289	289	5896	1105	5397	5397	5397
Array	1	1	1	1	1	3	3	3	3	3
End	0	0	0	0	0	203	203	203	203	203
Get	2774	1949	2201	2078	863	12613	6608	11714	11616	8795
Put	824	824	824	701	137	1694	1694	1694	1596	1596
Ld	121	121	121	121	121	1812	1713	1812	1812	1812
St	3	3	3	3	3	414	414	414	414	414
Jmp	355	355	229	220	115	1098	1098	800	401	401
Jt	339	339	339	352	231	496	496	496	496	895
Jf	620	620	281	268	124	1597	1597	1098	1098	301
Lit	1126	1021	897	742	398	5947	5848	5549	4753	1932
Call	629	132	289	289	289	5896	1105	5397	5397	5397
Inc	0	0	0	123	3	0	0	0	98	98
Dec	0	0	0	0	0	0	0	0	0	0
Sys	23	23	23	23	23	1288	1288	1288	1288	1288
Jne	0	0	0	0	7	0	0	0	0	398
Ldxv	0	0	0	0	230	0	0	0	0	1094
Stxv	0	0	0	0	5	0	0	0	0	1727
Seqi	0	0	0	0	120	0	0	0	0	0
total	**8585**	**6556**	**6855**	**6412**	**3777**	**43580**	**27696**	**40688**	**38599**	**32559**

Table 9.4 The profile of the benchmarks, comparing the baseline and the macro expansion. (1) is macro/base

	compiler		message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
macro	6556	26422	27696	120644 (22724:97920)	114
(1) %	76.4	69.6	63.6	54.6	

Observing the frequency of instruction used in Table 9.3, the "call" (plus "ret") is reduced by 80% in macro version and 30% of "get" is reduced (in getting the parameters). Similar reduction is observed in Nos benchmark, 80% reduction in "call", and 48% reduction in "get". So macro expansion is highly effective.

9.3 Introduce new primitives into the language

Nut is designed to be minimal. Many basic and frequently used functions are written as user-defined functions, such as !=, >=, <=, and, or, not. As the processor Sx already has machine instructions to support these functions, they should be considered as built-in operators of the language. The code generator can be modified to convert the call to these functions into generating the associated instructions of the Sx processor. This is not the same as macro expansion because the way Sx instruction behave is not exactly the same as the same operation written in Nut-language, for example the "and" function, in Nut.

```
(def and (a b) (if a b 0))
```

The meaning of this "and" is "if *a* is true then the result depends on *b*, else the result is false". Its semantic is the "short-cut and" where the argument is evaluated enough to know the result (not always evaluate all arguments as in *eager evaluation* semantic). However, the machine instruction "Band" will evaluate all arguments. So there is a difference. The macro expansion will

preserve the semantic of the function. The machine primitive will be faster but not always. In some case where evaluating arguments is costly, "short cut" semantic may be faster because it may evaluate less number of arguments.

Here is how the code generator is modified. The functions !=, >=, <=, and, or, not are still written as user-defined functions in the source program, their use will be compiled into function calls. The code generation for "call" checks the index to these functions and generates Sx machine instructions instead of a normal call.

```
; e is arglist
(def gencall (arg e) (idx a)
   (do
   (set idx (searchRef arg))
   ...
   (if (= idx yNe) (genop icNe 0 e)
   (if (= idx yLe) (genop icLe 0 e)
   (if (= idx yGe) (genop icGe 0 e)
   (if (= idx yAnd) (genop icBand 0 e)
   (if (= idx yOr) (genop icBor 0 e)
   (if (= idx yNot) (genop icNot 0 e)
   ; else
   (genop icCall idx e))))))))))        ; normal call

; eval arg-list (e), out code op.arg
(def genop (op arg e) ()
   (do
   (while e
      (do
      (eval (head e))
      (set e (tail e))))
   (outa op arg)))
```

Where yNe, yLe, yGe, yAnd, yOr, yNot are the indexes to the user-defined functions !=, <=, >=, and, or, not. These indexes are found in the symbol table which is read by the code generator. The instructions icNe, icLe, icGe, icBand, icBor, icNot are the native Sx instructions for not-equal, less-than-or-equal, greater-than-or-equal, bitwise-and, bitwise-or, logical-not operations.

For example the following code fragment from “nut.txt” (the last line of function “str=”)

```
. . .
(and (= c1 0) (= c2 0))))
```

is normally compiled to call to the defined function “and”.

```
133 Get 1
134 Lit 0
135 Eq
136 Call and
```

With this code generator it is compiled into:

```
133 Get 1
134 Lit 0
135 Eq
136 Band
```

Let this code generator be “gen5.txt”. The following steps produce the profile statistic.

1 Compile the code generator.

```
c:>nvm nut.obj < gen5.txt > gen5.obj
```

2 Generate the compiler.

```
c:>nvm gen5.obj < nut.obj > nutp.obj
```

3 Run “nutp.obj” to compile “nut.txt”.

```
c:>sx nutp.obj < nut.txt
```

Similarly for the operating system, the step of work is as follows.

1 Generate the operating system using the modified code generator, “gen5.obj”.

```
c:>nvm gen5.obj < nos2.obj > nos2p.obj
```

Table 9.5 The profile of the benchmarks, comparing the baseline and the primitive. (1) is primitive/base

	compiler		message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
primitive	6855	28987	40688	206450 (44468:161982)	222
(1) %	79.8	76.2	93.3	93.4	

2 Run "nos2p.obj" to collect statistics.

```
c:>noss nos2p.obj
```

The result of running benchmarks using the code generator "gen5.txt" is shown in Table 9.5.

In terms of speedup (cycle), using only primitives is not as effective as macro expansion as it is only 24% faster (macro is 30%) but they are comparable. However, in the message passing benchmark, the primitives are not used much, the speedup is only 7% and 8% in task switching (for macro, 46% and 50%). So using primitives is not effective in the operating system as much as using macro expansion.

Inspecting the compiler task profile revealed that the reduction in the number of "call" is 56%, and "get" is 21%, not as much as in macro expansion (89% and 48% consecutively). In the message passing benchmark, the native instructions "Bor" and "Ne" generated by the modified code generator, are executed only 499 times, merely 1.2% of the total number of instruction executed.

9.4 Improving the quality of code from the code generator

The next step, the quality of code from the code generator can be improved. This method can be applied without changing the instruction set or the language. Some sequence of operations can be replaced by a shorter sequence of operations without affecting the result, hence making them faster.

We elected to do the following code optimisation, which are not difficult to implement.

1 Introduce inc/dec local variables as there are native instructions supported in Sx processor (as done in the exercise 6.4 of Chapter 6).
2 Improve jmp to jmp, jmp to ret, to "short cut" them.
3 Change (!= p 0) to p.
4 Change the conditional (= a 0) in "if" expression, there are two possibilities.
  4.1 (if (= a 0) x y) is replaced by (if a y x)
  4.2 (if (= a 0) x) is replaced by (if a skip x)

The expression with inc/dec can be detected from the N-code.

```
(set a (+ a 1))
```

It is normally compiled into,

```
get.a lit.1 add put.a
```

It can be replaced by generating the native code "Inc.a", "Dec.a".

The last two rules come from the observation that in the "if" expression, the conditional becomes:

```
get.a lit.0 eq jf
```

The sequence "lit.0 eq if" can be replaced by "jt". So the conditional becomes,

```
get.a jt
```

The rule 4.1 and 4.2 used this fact.

The code generator fragment for doing "inc/dec" is as follows. "genput" is the main function. It is called when the operator "put.a" is encountered at the beginning of the expression (in N-code) (put.a (add get.a lit.1)). "isIncDec" checks whether the expression is in the increment/decrement expression. If it is then generate the native instruction, otherwise generate the normal unary-op code (in "genuop").

```
(def genuop (op arg e) ()
   (do
   (eval e)
   (outa op arg)))

(def isIncDec (op arg e) ()
   (do
   (if (isATOM e) 0
   (if (!= (head e) (mkATOM op 0)) 0
   (if (!= (arg2 e) (mkATOM xGET arg)) 0
   (if (!= (arg3 e) (mkATOM xLIT 1)) 0
   1))))))

(def genput (op arg e) ()
   (if (isIncDec xADD arg e)
      (outa icInc arg)
   (if (isIncDec xSUB arg e)
      (outa icDec arg)
   ; else
   (genuop op arg e))))
```

The code fragment for transforming (!= p 0) to p is as follows (at the function "gencall"), similar to the generating the primitives. Where yNe is the index to the user-defined function (def != ...). If the call is "!=" and the second argument is "lit.0" then generate the native code, otherwise generate the normal code (as function call).

```
(if (= idx yNe)
   (if (isOpArg (arg2 e) xLIT 0)     ; (!= p 0) -> p
      (eval (head e))
      ; else
      (genop icNe 0 e))
```

The code fragment for "short-cut" jump is straight forward and we will not elaborate any further. The code fragment for performing reduction in "if" expression is as follows. Where "genif2" is the normal "if" generation, "iseq0" checks the expression of the form (= a 0).

```
(def genif e (e1 e2 e3 ads)
   (if (iseq0 (head e))              ; (= a 0)
      (do
      (set e1 (arg2 (head e)))       ; e1 is a
      (set e2 (arg2 e))
      (set e3 (arg3 e))
      (if e3                         ; (if (= a 0) x y) -> (if a y x)
         (genif2 (cons e1 (cons e3 (cons e2 NIL))))
         ; else
         (do                         ; (if (= a 0) x) -> if a skip x
         (eval e1)                   ; a
         (outa icJt 0)
         (set ads (- XP 1))
         (eval e2)                   ; x
         (patch ads (- XP ads)))))
   ; else
   (genif2 e)))                      ; normal if
```

The code generator, "gen5.txt" is modified with these rules. The result is the code generator which generates primitives: !=, <=, >=, and, or, not and with the above improvement, inc, dec, short-cut jump, reduce != and improve conditional (if (= a 0)..). Let this code generator be "gen6.txt".

The step to run benchmarks of this new code generator is:

1 Compile the code generator.

```
c:>nvm nut.obj < gen6.txt > gen6.obj
```

Table 9.6 The profile of the benchmarks. Comparing the primitive and the codegen. (1) is codegen/primitive

	Compiler		Message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
primitive	6855	28987	40688	206450 (44468:161982)	222
codegen	6412	27483	38599	199088 (44036:155052)	220
(1) %	93.5	94.8	94.8	96.4	

2 Use the new code generator to generate the executable object of the compiler.

```
c:>nvm gen6.obj < nut.obj > nutg.obj
```

3 Use the new compiler to compile the benchmark.

```
c:>sx nutg.obj < nut.txt
```

Similarly for the operating system benchmark, do the following steps.

1 Generate the executable Nos.

```
c:>nvm gen6.obj < nos2.obj > nos2g.obj
```

2 Run the new Nos2.

```
c:>noss nos2g.obj
```

The result is shown in Table 9.6.

The "gen6.txt" code generation is built on top of the "gen5.txt" (primitive) code generation so the performance improvement is considered relative to "gen5.txt" (a kind of further improvement of "gen5.txt"). The improvement in code generation further reduces the cycle by 5% in compiler benchmark and 4% in message passing benchmark. It reduces the task switching cycle by only 1%. Inspecting the profile of the compiler benchmark revealed that the "inc" is used 123 times, the instructions in the sequence that it replaced, consists of "get.a lit.1 add put.a" are reduced by the same amount. The profile of message passing benchmark is similar, but the "short-cut jump" which is not much in effect in the

compiler benchmark, is quite effective here, the number of "jmp" is reduced by 50% (from 800 to 401).

9.5 Instruction set level

The next level is the level of the instruction set. Although the instruction set is considered as "given" in any real computer system, to understand why an instruction set is designed that way, one should try to experiment with the effect of instruction set design [CHO03]. Observing the profile of the baseline benchmarks, two facts emerge.

1 The compiler benchmark is dominated by the execution of the function "str=".
2 Both benchmarks, the instructions to access data structure, mostly "ldx" and "stx" are used often, 370/8585 = 4.3% in compiler benchmark, 2821/43580 = 6.5% in message passing benchmark.

By introducing new instructions, these functions can be much faster. Implementing new instructions in the processor can be accomplished by writing new microprograms for those instructions in the Sx processor. Tools are available to create and execute new instructions in the Sx processor.

Let us start with the fact 1, the string comparison function. Here is the "str=" function from the Nut-compiler, "nut.txt".

```
; test string equal
(def str= (s1 s2) (flag i c1 c2)  [lib 28]
   (do
   (set flag 1)
   (set i 0)
   (while flag
      (do
      (set c1 (vec s1 i))
      (set c2 (vec s2 i))
      (if (!= c1 c2) (set flag 0)
      (if (= c1 0)   (set flag 0)
      (if (= c2 0)   (set flag 0))))
      (set i (+ i 1))))
   (and (= c1 0) (= c2 0))))
```

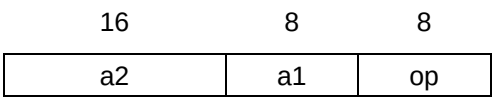
We should design the new instruction for the *inner* loop of this function. The inner loop does fetching two characters and compares them, at the same time it must checks the termination of strings. We do not try to do iteration within an instruction because it causes a long multiple cycles which is not desirable, especially at the microarchitecture level. It can cause unpredictable delay. But we try to do as much as possible in an instruction, so we will include the increment of index (set i (+ i 1)) in the instruction. Let the new instruction for string comparison be "string equal and increment" (seqi). Here is its pseudo code. Let *p1* and *p2* be two pointers to strings, *s1* and *s2* consecutively.

```
seqi p1 p2
  c1 = *p1              fetch a character
  if c1 == 0 ret 0      s1 terminate
  c2 = *p2              fetch a character
  if c2 == 0 ret 0      s2 terminate
  if c1 != c2 ret 0     if not equal ret false
  p1++                  increment both pointers
  p2++
  ret 1                 ret true
```

"seqi" fetches and compares two characters from two pointers to strings. It returns true if they are equal, otherwise it returns false, including when either pointers pointed to a terminal character. If it returns true, it also increment both pointers. This is very nice abstraction of the inner loop of the "str=" function. It does not do iteration and it does almost everything else (as much as possible). With the primitive "seqi" as a built-in operator, the "str=" function can be written as follows.

```
(def str= (s1 s2) ()
  (do
  (while (seqi s1 s2) (nop))              ; loop until false
  (and (= (vec s1 0) 0) (= (vec s2 0) 0))))
```

The instruction "seqi" has two arguments. This will introduce a new instruction format to the S-code instruction set. The two-address format is as follows.



Therefore some new signals in the data path must be added to decode this instruction format. Let they be *y.a1* and *y.a2* to feed the argument *a1* and *a2* to the y-mux in the data path.

The next step, we must write the microprogram for the "seqi" instruction. Here is its microprogram.

```
<seqi>                                ; seqi.a1.a2
  sp+1, pc+1
  ts->mW(sp)                          ; push ts
  alu(fp-a1)->tbus, mR(tbus)->nx      ; read p1
  mR(nx)->ts->ff                      ; read *p1,save
  alu(ts=0) ifT fetch                 ; ret 0
  alu(fp-a2)->tbus, mR(tbus)->nx      ; read p2
  mR(nx)->ts                          ; read *p2
  alu(ts=0) ifT fetch                 ; ret 0
  alu(ts=ff)->ts ifF fetch            ; ret 0
  alu(nx+1)->ts                       ; p2++
  alu(fp-a2)->tbus, ts->mW(tbus)      ; update p2
  alu(fp-a1)->tbus, mR(tbus)->nx      ; read p1
  alu(nx+1)->ts->ff                   ; p1++
  alu(fp-a1)->tbus, ts->mW(tbus)      ; update p1
  alu(nx!=ff)->ts fetch               ; 1->ts
```

The last line of microprogram is a good trick. To create a "*true*" value, we use the fact that "*nx*" is not equal to "*ff*" (we do not have any "*true*" value as a constant in the data path). This fact follows from the assignment in the line "*alu(nx+1)->ts->ff*", therefore the "*nx*" and "*ff*" will be definitely not the same. To do the function "not-equal" in the arithmetic logic unit, a signal "*alu.ne*" is added to the signal list in the microprogram specification, "mspec.txt". The timing of the "seqi" instruction is as follows. If it returns false, it takes one of the following cycle 6 or 9 or 10 cycles; if it returns true, it takes 16 cycles.

Now, let us turn our attention to the fact 2, accessing data structure. By inspecting the listing of the code (S-code), the data structure access is mostly in this form: (vec a index), (setv a index value), where a is the base address, index is mostly a constant. This observation suggests immediately the following instructions (also in two-argument format):

```
ldxv.a.x        TS = M[ M[fp-a] + x]
stxv.a.x        M[ M[fp-a] + x ] = TS
```

This is a kind of index addressing mode where the index is a constant. They are useful to access a structured data type such as a record. This kind of data is used very often in the benchmark programs.

Here are their microprograms.

```
<ldxv>   [micro 329]
  sp+1
  ts->mW(sp)              ; push ts
  alu(fp-a1)->tbus, mR(tbus)->ts
  alu(ts+a2)->tbus, mR(tbus)->ts fetch

<stxv>   [micro 335]
  alu(fp-a1)->tbus, mR(tbus)->nx
  alu(nx+a2)->tbus, ts->mW(tbus)
  mR(sp)->ts           ; pop ts
  sp-1, pc+1, fetch
```

Now the sequence of instruction for (vec a 0) and (setv a 3 44) which are normally compiled into:

```
get.a lit.0 ldx
```

and

```
get.a lit.3 lit.4 stx
```

will become a shorter sequence.

```
ldx.a.0
```

and

```
lit.44 stxv.a.3
```

"ldxv" takes slightly more cycle than "ldx" ("ldxv" 5, "ldx" 4 cycles). Surprisingly "stxv" takes less cycle than "stx" ("stxv" 5, "stx" 7 cycles). This is

because the "stxv" has its two arguments ready in the instruction; it does not have to pop the evaluation stack as much as "stx".

Finally, the sequence "eq, jf" is founded often; it will be replaced by a new instruction "jne". In fact, many of this "eq, jf" have already been optimised away when transforming (if (= a 0) x y) to eliminate the (= a 0) but the remaining is still significant.

```
<jne>   [micro 339]
  mR(sp)->ff
  sp-1
  alu(ts=ff), ifT j3
<jump>                      ; jump
  pc+arg, mR(sp)->ts        ; pop ts
  sp-1, fetch
<j3>                        ; don't jump
  pc+1, mR(sp)->ts          ; pop ts
  sp-1, fetch
```

How to generate microprogram

Now we have four new instructions: seqi, ldxv, stxv, jne (the inc, dec instructions have already been implemented in the previous chapter). These four instructions must be installed into the processor simulator. Assume all the additional signal definition and the appropriated microprogram are added into the input file, the microprogram specification, "mspect.txt".

It is two steps to generate a microprogram. First, generate a microprogram bit-image. A microprogram bit-image is the raw data to be loaded into the microprogram control ROM. Second, to efficiently run the processor simulator, this bit-image is converted to an event list. It is 10 times faster when performing simulation using an event list.

1 Generate the bit-image file, mpgm.txt

```
c:>mgen > mpgm.txt
```

mgen implicitly takes the input file "mspec.txt". mgen outputs another file "mspec.h" as a header file of the signal definition.

2 Then generate the header file to be include in compiling the processor generator.

```
c:>sxgen
```

The `sxgen` reads input from two files "mpgm.txt" and "mspec.h" then it outputs one file "sxbit.h" which includes all signal definitions and the event list data structure to be used by the processor simulator.

The processor simulator, `sx`, has a small modification to include the new events and the order of execution of the events; two new functions to decode the IR fields (see the Sx processor simulator listing in the appendix G).

3 Recompile the Sx processor simulator.

```
c:>make sx
```

Similarly for the Noss simulator, include this "sxbit.h" and recompile.

```
c:>make noss
```

Code generation for new instructions

A new code generator is written on top of the previous code generator, "gen6.txt". It becomes "gen7.txt". This code generator accepts the primitive "seqi" and also generates code using "ldxv", "stxv" and "jne" as applicable. Let the new "str=" be included into a modified Nut-compiler (we do not touch anything else in "nut.txt"), named "nut5.txt". The steps of work to perform statistic collection for this final benchmark are.

1 Compile this new code generator, "gen7.txt".

```
c:>nvm nut.obj < gen7.txt > gen7.obj
```

2 Compile "nut5.txt" to N-object.

```
c:>nvm nut.obj < nut5.txt > nut5.obj
```

3 Use "gen7.obj" code generator to generate code from this modified compiler.

```
c:>nvm gen7.obj < nut5.obj > nut5x.obj
```

4 Run this new compiler (that includes the extended instructions) with the updated simulator to compile the compiler benchmark. The "x" suffix in the final object code signified the extended instruction.

```
c:>sx nut5x.obj < nut.txt
```

The message passing benchmark is similar.

1 Use "gen7.obj" to generate code.

```
c:>nvm gen7.obj < nos2.obj > nos2x.obj
```

2 Run it under the updated noss.

```
c:>noss nos2x.obj
```

The result is shown in Table 9.7.

Table 9.7 The profile of the benchmarks. Comparing the baseline, the codegen, and the extend. (1) is extend/codegen (2) is extend/base

	Compiler		Message passing		
	instr. x1000	cycle x1000	instr	cycle (system:user)	cycle/ switch
base	8585	38033	43580	220915 (49274:171641)	246
codegen	6412	27483	38599	199088 (44036:155052)	220
extend	3777	17922	32559	171637 (37791:133486)	189
(1) %	58.9	65.2	84.4	86.2	
(2) %	44.0	47.1	74.7	77.7	

In terms of cycle, the new instruction set (labeled "extend" in the table) further reduce (relative to codegen) for the compiler benchmark, 35%, for the message passing benchmark, 14%. This shows that the "seqi" is very effective as it is designed to speedup the "str=" in the compiler. As a whole, comparing the

number of cycle from this new “extend” optimisation with the baseline, the result shows that for the compiler benchmark, the speedup is more than twice the baseline, 53%, for the message passing, the speedup is 23%. It is worth noticing that for the message passing benchmark, the macro expansion is more effective, the speedup is 46%. The fastest task switching takes only 114 cycles.

The table below shows the summary of the speedup figure of each optimisation method. The chart from this table is shown in Fig. 9.1.

Table 9.8. The summary of speedup of each optimisation method in terms of cycle

	compiler	message passing
base	0.0	0.0
macro	30.5	45.4
primitive	23.8	6.5
codegen	27.7 (5.2)	9.0 (3.6)
extend	52.9 (34.8)	22.3 (13.8)

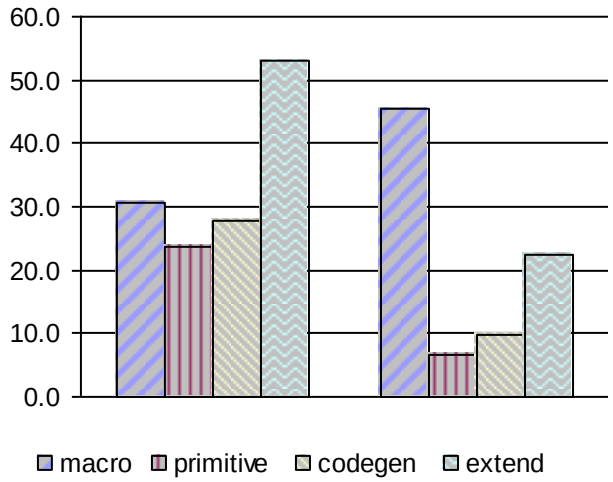


Figure 9.1 Chart of speedup (%) of each optimisation method in terms of cycle

The speedup is calculated from (1-(A/B)) x 100, where A is a method, B is the baseline. Speedup means how much faster the system A compared to the system B. If A is 20% faster than B, A completes the task in 20% less cycle than B. The figure in parentheses show the incremental speedup, for example, in the compiler benchmark, the method "codegen" is 5.2% relatively faster than the method "primitive", although it is 27.7% faster than the baseline. This shows the speedup factor from the incremental change of the method (the "codegen" method implemented many techniques *in addition to* the method "primitive").

9.6 Microarchitecture level

The chapter 7 has developed a processor that retains the same instruction set, S-code, but has 30% faster data path. The improvement at this level comes from the *architectural* design and the advancement in fabrication technology. These two factors are interrelated. The Sx2 processor is faster due to the *stack frame caching*, the use of fast registers to provide access to local variables instead of accessing them from the memory. The *parallelism* or the concurrent use of units in data path also provides performance enhancement. The use of separate unit for updating the stack pointer in Sx2 is an example of this case. Many other standard methods such as pipelining, using multiple functional units, *instruction level parallelism* are extensively discussed in many computer architecture textbooks [HEN03] [PAT98] [STO93].

9.7 Summary

In this chapter we have shown a wide range of performance improvement methods. The techniques are applicable at every level, from the top level (at the application software), down to the hardware level (at the level of data path and microprogram). At the highest abstraction level, it is a well-known fact that, the change at an algorithmic level will have the greatest impact on performance. We observe data that support this hypothesis. In the compiler benchmark profile, it is clear that the access to symbol table is the performance bottleneck. The symbol table requires a lot of string comparison. If however, the data structure is changed to a hash table, then the number of string comparison will be reduced dramatically. Our symbol table executes most accessed functions in O(n) where n is the number of the entry because it used sequential search. This will change to a constant access time, O(1), with a hash table (see Exercise 9.1).

The macro expansion is studied and implemented. It has a high impact on performance without having to change the underlying programs much (such as the code generation and the instruction set). Another change at the language level is to introduce more operators. Our experiment shows that this is also effective although not as much as macro expansion. However it is not effective in the message passing benchmark because these new operators are not used very often. This shows that it is not easy to gain performance in general by this method. It is sensitive to the type of applications.

The next level is the code generation level. The quality of code can be improved. Most methods at this level are classified as *code optimisation* in the literature of compiler. We choose to perform a number of significant code improvements. The result shows that the performance gain is not large but it is very logical method. The rule for replacing some sequence with a shorter sequence to improve performance is numerous (in fact, it is combinatorial even!). One can almost invent a new rule by inspection the output code. However, how often that sequence will be used is not always easy to predict.

The instruction set level is interesting. Inventing new instructions is not practical in a real computer system where hardware is *given*. However, our study shows that a huge performance gain is possible. The "string equal and increment" instruction is an extreme example. It is more than twice as fast as the code without this instruction in the compiler benchmark. However, it has a limit application to string comparison. This is why the multimedia extension of an instruction set is so important for modern processors.

9.8 Further reading

In general, a programming language should not be designed with paramount of efficiency in mind. The high level language should reflect the efficiency of human programmers, at the cognitive level. It should reduce the burden of human programmers. During early development of computer systems many components were being developed: the hardware, the compiler, the operating system, the user interface etc. The designers and the developers faced the difficulty of not having the machine fast enough to run the intended applications. This must be the nature of computer technology that human can imagine new kind of application beyond what a computer system can support at the time. So, in all history of development of computer systems, efficiency is always a

paramount factor. The C language [KER78] reflects this fact. Pascal [WIR71] also reflects this concern. Computer language design is still an open, endless quest for a tool of thought to invent the next generation machines. The history of computer development can be read in IEEE Annals of the history of computing.

For code optimisation, many compiler related textbooks are excellent source of information [AHO86] [FRA95] [LOU97]. Although it seems that code optimisation is used for code generation, it application can be wider range. For example, it is also applicable at the *meta* level programming, to build a virtual machine (or environment) [CHO98], to prototype a new computer system.

References

[AHO86] Aho, A., Sethi, R. and Ullman, J. Compilers: Principles, techniques, and tools, Addison-Wesley, 1986.

[CHO98] Chongstitvatana, P. A multi-tasking environment for real-time control. Final report, Faculty of Engineering, Chulalongkorn university, research project number 132-MRD-2537, 1998. Also available on-line at http: //www.cp.eng.chula.ac.th/faculty/pjw/r1/

[CHO03] Chongstitvatana, P., "The Art of Instruction Set Design", Electrical Engineering Conference, Thailand, 2003.

[COR01] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., Introduction to algorithms, 2nd ed., MIT Press, 2001.

[FRA95] Fraser, C. and Hanson, D. A retargetable C compiler: design and implementation, Benjamin/Cummings Pub., 1995.

[HEN03] Hennessy, J., and Patterson, D., Computer Architecture: a quantitative approach, 3rd ed. Morgan Kaufmann, 2003.

[KER78] Kernighan, B., and Ritchie, D., The C programming language, Prentice-Hall, 1978.

[KLI05] Klienberg, J., and Tardos, E., Algorithm Design, Addison Wesley, 2005.

[LOU97] Louden, K. Compiler construction: Principles and practice, Inter. Thompson Pub., 1997.

[PAT98] Patterson, D., and Hennessy, J., Computer Organization and Design: the hardware/software interface, 2nd ed. Morgan Kaufmann, 1998.

[PRA01] Prasitjutrakul, S., Analysis and design of algorithms, NECTEC, 2001. (in Thai).

[SEB04] Sebesta, R. Concepts of programming languages, 6th ed. Pearson/Addison-Wesley, 2004.

[STO93] Stone, H., High performance Computer architecture, McGraw-Hill, 1993.

[WIR71] Wirth, N., "The programming language Pascal", Acta Informatica, 1(1):35-63, 1971.

Exercises

9.1 Do some simple optimization. At the level of algorithm, change the symbol table access to use hash table. Run the compiler benchmark with this new compiler. Collect the profile and discuss the result.

9.2 Do some language extension. In the Nut-compiler and the code generator, there are frequent uses of multiway branch. To do multiway branch, nested-if is used. This is very flexible but it is inefficient to do sequential testing on the conditions. Also, the syntax for deep nested-if is quite cumbersome, especially the closing parenthesis. The number of right parenthesis is equal to the depth of nested-if.

```
(if (= op xADD) (doadd)
(if (= op xSUB) (dosub)
...
; else
(error "unknown op") ...))
```

To help improving the syntax in general case, a new control-flow operator is needed. The expression (cond....) in LISP is a good example. In Nut, we want to do:

```
(switch
  (cond1) (action1)
  (cond2) (action2) ...
  true (default action))
```

This syntax will reduce the number of the closing right parenthesis to one.

To improve the efficiency, if the condition is testing the equality of a variable with many constants, then we can do similar to "switch, case" in C or Pascal. The implementation can use a jump-table indexed by the value of that variable.

```
(case op
  (label xADD (doadd))
  (label xSUB (dosub))
  ...
  (else (default action)))
```

The N-code for "case" will contain a jump-table of the association list of (label jump-to) which is implemented as an array of cells. To conform to the existing N-code only the head can be an atom, therefore the head is a label instruction (a new instruction), the tail is a pointer to the action body.

In N-code, assume op is local, len is length of the jump-table, and the last entry is an instruction to terminate the table. The "case" construct is compiled to:

```
(case.op
  ; jump table
  (label.xADD L1)
  (label.xSUB L2)
  ...
  (else.0 Ln))
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

If the label is sorted according to the key (constants) then the jump table can be a binary search table. The length of jump-table must be known.

```
(case2.op lit.len
  ; jump table, label is sorted by its arg
  [xADD L1
   xSUB L2
  ...
   NIL Ln])
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

Another way which is faster is to use label to index into the jump table directly. The size of jump table is equal to the range of index. The label instruction is not necessary. The range of index must be known.

```
(case3.op lit.lo lit.hi
  ; jump table, label is sorted by its arg
  [L1 L2 ... Ln])
  ; actions
  <L1> (action 1)
  <L2> (action 2)
  ...
  <Ln> (default action)
```

For the second kind and third kind of "case", the jump-table itself does not conform the format of N-code. To implement it, it is not really a list; it is an array (which is a data). To have an array in the code segment, an additional representation must be designed.

Implement a multiway branch. Run the benchmark and measure its effectiveness.

9.3 Write a code generator for S2, the register-based processor. Compile all the benchmarks to target this instruction set. Run and collect the performance statistic. Compare it with Sx processor. Discuss the result.

9.4 One can always argue that some inefficiency comes from the quality of a compiler. To prove or disprove this belief, we can write a segment of program in machine language then compare the execution time with the

one produced by a compiler. Write the string comparison function in S-code by hand. Try to optimise it by speed. Compare the running time with the one produced by Nut-compiler.

9.5 Invent new instructions by combining two instructions into a new one. Produce a log file of the frequency of pair of instructions and select five most frequently used pairs. Write their microprogram. Integrate these new instructions into the Sx processor simulator. Modify the code generator to output these new instructions. Run the benchmarks and discuss the result. How much do you expect the improvement will be?

9.6 Performance measurement is sensitive to benchmark programs. Write a new set of benchmark programs, for example, image processing. Measure the performance of our system on this new benchmark. What is the difference and why?

Appendices

Appendix A

Common Functions

```
 1 ; Library of common functions
 2
 3 (def != (a b) () (if (= a b) 0 1))
 4 (def >= (a b) () (if (< a b) 0 1))
 5 (def and (a b) ()(if a b 0))
 6 (def or (a b) () (if a 1 b))
 7 (def not a () (if a 0 1))
 8 (def nop () () 0)
 9 (def print a () (sys 1 a))
10 (def printc c () (sys 2 c))
11 (def space () () (sys 2 32))
12 (def nl () () (sys 2 10))
13 (def exit () () (sys 13))
14
15 ; --------- string ----------
16
17 ; input output of string functions are pointers
18 ; to dereference it, use (vec s 0)
19
20 ; print string
21 (def prstr s ()
22   (while (vec s 0)
23     (do
24     (printc (vec s 0))
25     (set s (+ s 1)))))
26
27 ; test string equal
28 (def str= (s1 s2) (flag i c1 c2)
29   (do
30   (set flag 1)
31   (set i 0)
32   (while flag
33     (do
34     (set c1 (vec s1 i))
35     (set c2 (vec s2 i))
36     (if (!= c1 c2) (set flag 0)
37     (if (= c1 0)   (set flag 0)
38     (if (= c2 0)   (set flag 0))))
39     (set i (+ i 1))))
40   (and (= c1 0) (= c2 0))))
```

```
41
42 ; find string length
43 (def strlen s ()
44   (if (vec s 0)
45     (+ 1 (strlen (+ s 1)))
46     0))
47
48 ; copy string
49 (def strcpy (s1 s2) a
50   (do
51   (set a (vec s2 0))
52   (while a
53     (do
54     (setv s1 0 a)
55     (set s1 (+ s1 1))
56     (set s2 (+ s2 1))
57     (set a (vec s2 0))))
58   (setv s1 0 0)))
59
60 ; check is string a number, no sign
61 (def isNumber s a
62   (do
63   (set a (vec s 0))
64   (while (and (> a 47) (< a 58))   ; isdigit a
65     (do
66     (set s (+ s 1))
67     (set a (vec s 0))))
68   (= a 0)))                        ; true if reach end
69
70 (def isString s () (= (vec s 0) 34)) ; start with quote
71
72 ; convert string to number, no sign
73 (def atoi s (a v)
74   (do
75   (set v 0)
76   (set a (vec s 0))
77   (while a
78     (do
79     (set v (- (+ (* v 10) a) 48))
80     (set s (+ s 1))
81     (set a (vec s 0))))
82   v))
83
84 (def error s ()
85   (do
86   (prstr s)
87   (sys 12)            ; line no
88   (sys 13)))          ; exit
89
90 ; ---------- data ---------
91
```

```
 92 (def head e () (vec e 0))
 93 (def tail e () (vec e 1))
 94 (def sethead (e v) () (setv e 0 v))
 95 (def settail (e v) () (setv e 1 v))
 96
 97 (def cons (e list) h
 98   (do
 99   (set h (new 2))
100   (setv h 0 e)        ; set head
101   (setv h 1 list)     ; set tail
102   h))
103
104 (def arg1 e () (head e))
105 (def arg2 e () (head (tail e)))
106 (def arg3 e () (head (tail (tail e))))
107
108 ; End
```

Appendix B

Nut Compiler

```
 1 ; nut compiler  in nut
 2 ;   nut completion kit
 3
 4 ; include lib.txt
 5
 6 ; -------- header ----------
 7
 8 (enum 127 EOF)
 9 (enum 0 NIL)
10 (enum 3000 MAXNAMES)
11 (enum 3000 MAXSTR)
12 (enum 3000 MEMMAX)
13 (enum 1
14   xIF xWHILE xDO xUD1 xNEW xADD xSUB xMUL xDIV
15   xEQ xLT xGT xCALL xGET xPUT xLIT xLDX xSTX xFUN
16   xSYS xSET xSETV xVEC xUD2 xLD xST xLDY xSTY xUD3
17   xUD4 xUD5 xSTR xBAND xSHR xSHL)
18 (enum 2
19   tyVAR tyFUN tyOP tyOPX tySYS tyUD tyGVAR tyXX tyENUM)
20
21 ; tok   -- token string
22 ; LP RP -- string parenthesis
23 ; mem   -- data segment
24 ; DP    -- pointer to mem[]
25
26 (let tok LP RP DP mem)
27
28 (def init () ()
29   (do
30   (set mem (new MEMMAX))     ; global var area
31   (set DP 0)
32   (set LP "(")
33   (set RP ")")))
34
35 ; ------- symbol table --------
36
37 ; symbol table is an array
38 ; each item has five elements: name, type, val, arity, lv
```

```
39 ; name pointed to a string which is allocated separately
40
41 (enum 5 esize)              ; size of each element
42
43 ; symtab -- symbol table
44 ; symstr -- symbol string table
45 ; symp   -- pointer to symstr[]
46 ; numNames -- number of symbol in symtab
47 ; numLocal -- number of local var
48
49 (let symtab symstr symp numNames numLocal)
50
51 ; access functions
52
53 (def getName idx () (vec symtab idx))
54 (def getType idx () (vec symtab (+ idx 1)))
55 (def getVal idx () (vec symtab (+ idx 2)))
56 (def getArity idx () (vec symtab (+ idx 3)))
57 (def getLv idx () (vec symtab (+ idx 4)))
58 (def setName (idx nm) () (setv symtab idx nm))
59 (def setType (idx ty) () (setv symtab (+ idx 1) ty))
60 (def setVal (idx v) () (setv symtab (+ idx 2) v))
61 (def setArity (idx v) () (setv symtab (+ idx 3) v))
62 (def setLv (idx v) () (setv symtab (+ idx 4) v))
63
64 ; allocate new string from symstr[]
65 (def newName nm (k v)
66   (do
67   (set k (+ (strlen nm) 1))
68   (set v (+ symstr symp))
69   (set symp (+ symp k))
70   (if (> symp MAXSTR)
71     (error "symbol string full"))
72   (strcpy v nm)
73   v))
74
75 ; search symtab for nm,
76 ; if found, return its index, else, insert it
77 (def install nm (i flag end)
78   (do
79   (set i 0)
80   (set flag 1)
81   (set end (* esize numNames))
82   (while (and flag (< i end))
83     (if (str= (getName i) nm)      ; sequential search
84       (set flag 0)
85       ; else
86       (set i (+ i esize))))
87   (if flag                         ; not found
88     (do
89     (if (> i MAXNAMES)
```

```
 90       (error "symtab overflow"))
 91     (setName i (newName nm))
 92     (setType i tyUD)
 93     (set numNames (+ numNames 1))))
 94   i))
 95
 96 ; install nm as local variable
 97 (def installLocal nm idx
 98   (do
 99   (set numLocal (+ numLocal 1))
100   (set idx (install nm))
101   (setType idx tyVAR)
102   (setVal idx numLocal)))
103
104 (def dumpsym () (i end)
105   (do
106   (print (- numNames 18)) (nl)
107   (set i 90)                        ; 18 keywords
108   (set end (* esize numNames))
109   (while (< i end)
110     (do
111     (prstr (getName i)) (space)
112     (print (getType i)) (space)
113     (print (getVal i)) (space)
114     (print (getArity i)) (space)
115     (print (getLv i)) (nl)
116     (set i (+ i esize))))))
117
118 (def insertsym (nm ty op) (idx)
119   (do
120   (set idx (install nm))
121   (setType idx ty)
122   (setVal idx op)))
123
124 ; initially keywords are inserted into symtab
125 ; its value is its opcode
126 (def initsym () ()
127   (do
128   (set symtab (new MAXNAMES))
129   (set symstr (new MAXSTR))        ; symbol string
130   (set symp 0)                     ; symstr pointer
131   (set numNames 0)
132   (set numLocal 0)
133   (insertsym "if" tyOP xIF)
134   (insertsym "while" tyOP xWHILE)
135   (insertsym "set" tyOPX xSET)
136   (insertsym "setv" tyOPX xSETV)
137   (insertsym "do" tyOP xDO)
138   (insertsym "new" tyOP xNEW)
139   (insertsym "+" tyOP xADD)
140   (insertsym "-" tyOP xSUB)
```

```
141   (insertsym "*" tyOP xMUL)
142   (insertsym "/" tyOP xDIV)
143   (insertsym "=" tyOP xEQ)
144   (insertsym "<" tyOP xLT)
145   (insertsym ">" tyOP xGT)
146   (insertsym "vec" tyOPX xVEC)
147   (insertsym "sys" tySYS xSYS)
148   (insertsym "&" tyOP xBAND)
149   (insertsym ">>" tyOP xSHR)
150   (insertsym "<<" tyOP xSHL)))
151
152 ; ---------- data ---------
153
154 ; allocate n int from mem[]
155 (def newdata n (a)
156   (do
157   (if (>= DP MEMMAX)
158     (error "out of memory"))
159   (set a DP)
160   (set DP (+ DP n))
161   a))
162
163 (def isATOM e () (< e 0))    ; MSB bit 1
164 (def mkATOM (op arg) ()
165   (+ (<< (+ (& op 127) 128) 24) (& arg 16777215)))
166
167 ; --------- parser ----------
168
169 (def tokenise () ()
170   (do
171   (set tok (sys 3))
172 ;  (printc 34) (pstrs tok) (space)
173   (nop)))
174
175 (def expect s ()
176   (if (not (str= tok s))
177     (do
178     (prstr "expect ")
179     (error s))))
180
181 (def prList e () (sys 10 e))
182
183 ; parse name list, in fun header
184 ; NL can be singleton or list
185 (def parseNL () ()
186   (do
187   (tokenise)
188   (if (str= tok LP)
189     (do
190     (tokenise)
191     (while (not (str= tok RP))
```

```
192       (do
193       (installLocal tok)
194       (tokenise))))
195     ; else
196     (installLocal tok))))
197
198 ; parse local var
199 (def doVar (op arg) (v)
200   (do
201   (if (= op xSET) (set v (mkATOM xPUT arg))
202   (if (= op xSETV) (set v (mkATOM xSTX arg))
203   (if (= op xVEC) (set v (mkATOM xLDX arg))
204   (error "unknown op")))))
205   v))
206
207 ; parse global var
208 (def doGvar (op arg) () 0)
209
210 ; parse each enum sym
211 (def doEnum n () 0)
212
213 ; parse a name
214 (def parseName () (idx n n2 ty v)
215   (do
216   (set idx (install tok))
217   (set n (getVal idx))
218   (set ty (getType idx))
219   (if (= ty tyOP) (set v (mkATOM n 0))
220   (if (= ty tyVAR) (set v (mkATOM xGET n))
221   (if (= ty tyGVAR) (set v (mkATOM xLD n))
222   (if (= ty tyFUN) (set v (mkATOM xCALL idx))
223   (if (= ty tyOPX)
224     (do
225     (tokenise)        ; get var name
226     (set idx (install tok))
227     (set n2 (getVal idx))
228     (if (= (getType idx) tyVAR)
229       (set v (doVar n n2))
230       ; else it is Gvar
231       (set v (doGvar n n2))))
232   (if (= ty tySYS)
233     (do
234     (tokenise)        ; get sys num
235     (set v (mkATOM xSYS (atoi tok))))
236   (if (= ty tyENUM)
237     (set v (doEnum n)))))))))
238   v))
239
240 (def parseExp () () 0)             ; forward declaration
241
242 ; parse expression list
```

```
243 (def parseEL () v
244   (do
245   (tokenise)
246   (if (str= tok RP)
247     (set v NIL)
248     ; else
249     (set v (cons (parseExp) (parseEL))))
250   v))
251
252 (def doString s () 0)
253
254 ; parse expression
255 (def parseExp () v
256   (do
257   (if (str= tok LP)                ; it is a list
258     (do
259     (tokenise)
260     (set v (cons (parseName) (parseEL))))
261   (if (isNumber tok)               ; it is a number
262     (set v (mkATOM xLIT (atoi tok)))
263   (if (isString tok)
264     (set v (doString (+ tok 1)))
265   (set v (parseName)))))
266   v))
267
268 ; parse function definition
269 (def parseDef () (idx arity e k)
270   (do
271   (tokenise)
272   (set idx (install tok))
273   (setType idx tyFUN)
274   (set numLocal 0)
275   (parseNL)
276   (set arity numLocal)
277   (parseNL)
278   (tokenise)
279   (set e (parseExp))
280   (tokenise)                       ; skip RP
281   (if (isATOM e)
282     (set e (cons e NIL)))          ; body must be list
283   (setArity idx arity)
284   (setLv idx numLocal)
285   (set k (+ (* arity 256) (& numLocal 255)))
286   (setVal idx (cons (mkATOM xFUN k) (cons e NIL)))
287   idx))
288
289 (def prName idx () (prstr (getName idx)))
290
291 ; parse "let" expression
292 (def parseLet () (idx x)
293   (do
```

```
294   (tokenise)
295   (while (not (str= tok RP))
296     (do
297     (set idx (install tok))
298     (if (!= (getType idx) tyUD)
299       (error "redefine global var"))
300     (setType idx tyGVAR)
301     (setVal idx (newdata 1))
302     (prstr tok) (nl)
303     (tokenise)))))
304
305 ; parse "enum" expression
306 (def parseEnum () () 0)
307
308 ; the main parser
309 (def parse () (idx)
310   (do
311   (tokenise)
312   (while (!= (vec tok 0) EOF)
313     (do
314     (expect LP)
315     (tokenise)
316     (if (str= tok "def")
317       (do
318       (set idx (parseDef))
319       (prName idx) (nl)
320       (prList (getVal idx)) (nl))
321     (if (str= tok "let")
322       (parseLet)
323     (if (str= tok "enum")
324       (parseEnum)
325     ; else
326     (error "unknown keyword"))))
327     (tokenise)))))
328
329 ; ------- rename ---------
330
331 ; resolve scans symtab to find fun def and rename it
332 ; rename local var from 1..n to n..1
333 ; and instantiates call.idx to the actual reference
334
335 (let LV) ; number of local var in a fun def
336
337 (def de_op x () (& (>> x 24) 127))
338 (def de_arg x () (& x 16777215))   ; x & 0x0ffffff
339
340 ; get/put/ldx/stx rename 1..n to n..1
341 ; call instantiate ref
342 (def reATOM e (op arg)
343   (do
344   (set op (de_op (head e)))
```

```
345   (set arg (de_arg (head e)))
346   (if (= op xFUN)
347     (set LV (& arg 255))
348   (if (or (= op xGET) (= op xPUT))
349     (sethead e (mkATOM op (+ (- LV arg) 1)))
350   (if (or (= op xLDX) (= op xSTX))
351     (sethead e (mkATOM op (+ (- LV arg) 1)))
352   (if (= op xCALL)
353     (sethead e (mkATOM op (getVal arg)))))))))
354
355 ; traverse n-code with one lookahead
356 (def reName e ()
357   (if (!= e NIL)
358     (if (isATOM (head e))
359       (do
360       (reATOM e)
361       (reName (tail e)))
362       ; else
363       (do
364       (reName (head e))
365       (reName (tail e))))))
366
367 ; scan symtab for fundef and reName its body
368 (def resolve () (i end e)
369   (do
370   (set i 90)
371   (set end (* esize numNames))
372   (while (< i end)                  ; scan symtab
373     (do
374     (if (= (getType i) tyFUN)
375       (do
376       (set e (getVal i))            ; body
377       (reName e)))
378     (set i (+ i esize))))))
379
380 ; relocate arg in atom call.arg by disp
381 (def reloc (a disp) (v op arg)
382   (do
383   (set op (de_op a))
384   (set arg (de_arg a))
385   (if (= op xCALL)
386     (set v (mkATOM op (+ (- arg disp) 2)))
387     ; else
388     (set v a))
389   v))
390
391 ; shift a dot-pair by disp, except NIL
392 (def shift (a disp) ()
393   (if a
394     (+ (- a disp) 2)
395     0))
```

```
396
397 ; relocate start to 2
398 (def outobj2 (start end) (i a b ty)
399   (do
400   (set a (getVal (install "main")))
401   (print (shift a start)) (space)
402   (print (- (shift end start) 2)) (nl)
403   (set i start)
404   (while (< i end)           ; code segment
405     (do
406     (set a (head i))
407     (set b (tail i))
408     (if (isATOM a)
409       (do
410       (set ty 1)
411       (set a (reloc a start)))
412       ; else dot-pair
413       (do
414       (set ty 0)
415       (set a (shift a start))))
416     (print (shift i start)) (space)
417     (print ty) (space)
418     (print (de_op a)) (space)
419     (print (de_arg a)) (space)
420     (print (shift b start)) (nl)
421     (set i (+ i 2))))
422   (print DP) (nl)            ; data segment
423   (set i 0)
424   (while (< i DP)
425     (do
426     (print (vec mem i)) (space)
427     (set i (+ i 1))
428     (if (= (& i 7) 0) (nl))))
429   (nl)
430   (dumpsym)))
431
432 (def main () (start end)
433   (do
434   (init)
435   (initsym)
436   (set start (sys 9))
437   (sys 11)            ; readinfile
438   (parse)
439   (resolve)
440   (set end (sys 9))
441   (outobj2 start end)
442   (nop)))
443
444 ; End
```

Appendix C

Nut Completion Solution

```
 1 ; nut-completion solution
 2
 3 ; allocate string space from mem[]
 4 ; copy s to there
 5 (def mkSTR s (s2)
 6   (do
 7   (set s2 (newdata (+ (strlen s) 1)))
 8   (strcpy (+ mem s2) s)
 9   s2))
10
11 (def doGvar (op arg) (v)
12   (do
13   (if (= op xSET) (set v (mkATOM xST arg))
14   (if (= op xSETV) (set v (mkATOM xSTY arg))
15   (if (= op xVEC) (set v (mkATOM xLDY arg))
16   (error "unknown op"))))
17   v))
18
19 (def doEnum n () (mkATOM xLIT n))
20
21 (def doString s () (mkATOM xSTR (mkSTR s)))
22
23 (def parseEnum () (idx k)
24   (do
25   (tokenise)
26   (if (not (isNumber tok))
27     (error "expect number"))
28   (set k (atoi tok))
29   (tokenise)
30   (while (not (str= tok RP))
31     (do
32     (set idx (install tok))
33     (if (!= (getType idx) tyUD)
34       (error "redefine enum name"))
35     (setType idx tyENUM)
36     (setVal idx k)
37     (set k (+ k 1))
38     (tokenise)))))
39
40 ; End
```

Appendix D

N-code Evaluator

```
 1 ; eval.txt   n-code evaluator
 2 ;
 3 ;   eval in nut
 4 ;   partially implement only 12 operators
 5 ;   to run the test program "t2.txt"
 6
 7 ; include lib.txt
 8
 9 ; -------- header ----------
10
11 (enum 0 NIL)
12 (enum 3000 STKMAX)    ; run-time stack
13
14 (enum 1
15   xIF xWHILE xDO xUD1 xNEW xADD xSUB xMUL xDIV
16   xEQ xLT xGT xCALL xGET xPUT xLIT xLDX xSTX xFUN
17   xSYS xSET xSETV xVEC xUD2 xLD xST xLDY xSTY xUD3
18   xUD4 xUD5 xSTR xBAND xSHR xSHL)
19
20 (let tok DP CS M)     ; data pointer, code segment
21 (let SS SP FP)        ; stack, stk pointer, frame pointer
22
23 (def init () ()
24   (do
25   (set M 0)                        ; base ads, absolute
26   (set SS (new STKMAX))
27   (set FP SS)
28   (set SP SS)))
29
30 ; ----- system --------
31
32 (def tokenise () ()
33   (set tok (sys 3)))
34
35 (def prList e () (sys 10 e))
36
37 (def de_op x () (& (>> x 24) 127))
38 (def de_arg x () (& x 16777215))   ; x & 0x0ffffff
39
40 (def isATOM e () (< e 0))          ; MSB bit 1
41 (def mkATOM (op arg) ()
```

```
42   (+ (<< (+ (& op 127) 128) 24) (& arg 16777215)))
43
44 ; ------- load oject -----------
45
46 (let Start)                  ; ads of "main"
47
48 ; read a token from stdin
49 ; and convert it to a number
50 (def read () ()
51   (do
52   (tokenise)
53   (atoi tok)))
54
55 ; offset a by disp, code segment started at 2
56 (def shift (a disp) ()
57   (if a
58     (- (+ a disp) 2)
59     0))
60
61 ; relocate arg of an op
62 (def reName (op arg) ()
63   (do
64   (if (= op xCALL)
65     (set arg (shift arg CS))
66   (if (or (= op xLD) (= op xST))
67     (set arg (shift arg DP))
68   (if (or (= op xLDY) (= op STY))
69     (set arg (shift arg DP))
70   (if (= op xSTR)
71     (set arg (shift arg DP))))))
72   (mkATOM op arg)))
73
74 ; load object, code segment, data segment
75 (def loadobj () (flag end a a2 ads ty op arg next)
76   (do
77   (set CS (sys 9))           ; start of code segment
78   (set Start (read))         ; ads of "main"
79   (set end (read))
80   (set DP (+ (+ CS end) 2)) ;  start of data segment
81   (set flag 1)
82   (while flag
83     (do
84     (set ads (read))
85     (set ty (read))
86     (set op (read))
87     (set arg (read))
88     (set next (read))
89     (if ty
90       (set a (reName op arg))
91       ; else dot-pair
92       (set a (shift (+ (<< op 24) arg) CS)))
```

```
 93     (set a2 (new 2))
 94     (sethead a2 a)
 95     (settail a2 (shift next CS))
 96     (if (= ads end) (set flag 0))))
 97
 98     ; load data segment
 99     (set a 0)
100     (set end (read))
101     (set a2 (new end))       ; alloc the whole block
102     (while (< a end)
103       (do
104       (setv a2 a (read))
105       (set a (+ a 1))))))
106
107 (def listall () (i a op end)
108   (do
109   (set i CS)
110   (set end (sys 9))
111   (while (< i end)
112     (do
113     (set a (head i))
114     (set op (de_op a))
115     (if (= op xFUN)
116       (do
117       (print i) (space)
118       (prList i) (nl)))
119     (set i (+ i 2))))))
120
121 ; --------- eval ----------
122
123 ; push a value to run-time stack
124 (def push e ()
125   (do
126   (set SP (+ SP 1))
127   (if (> SP (+ SS STKMAX))
128     (error "stack overflow"))
129   (setv M SP e)))
130
131 (def eval e () 0)            ; forward declaration
132
133 ; system call, implement print, printc
134 (def syscall (arg e) (v a1)
135   (do
136   (set v NIL)
137   (set a1 (eval (arg1 e)))
138   (if (= arg 1) (sys 1 a1)
139   (if (= arg 2) (sys 2 a1)
140   ; else
141   (error "undef sys")))
142   v))
143
```

```
144 ; fun.a.v no recode
145 (def funcall (arg e) (k v a)
146   (do
147   (set v (& arg 255))       ; decode a, v
148   (set a (>> arg 8))
149   (set k (+ (- v a) 1))
150   (setv M (+ SP k) FP)      ; save old FP
151   (set FP (+ SP k))         ; new frame
152   (set SP FP)
153   (set v (eval (arg1 e)))   ; eval body
154   (set SP (- (- FP v) 1))   ; delete frame
155   (set FP (vec M FP))       ; restore FP
156   v))
157
158 ; the main interpreter for n-code
159 ; partially implement only 12 operators
160 (def eval e (e1 op arg v idx)
161   (if (= e NIL)
162     NIL
163     ; else
164     (do
165     (if (not (isATOM e))    ; if it is a list
166       (do                   ; set e1 to arglist
167       (set e1 (tail e))     ; and e to operator
168       (set e (head e))))
169     (set op (de_op e))
170     (set arg (de_arg e))    ; decode operator
171     (if (= op xIF)
172       (if (eval (arg1 e1))
173         (set v (eval (arg2 e1)))
174         ; else
175         (set v (eval (arg3 e1))))
176     (if (= op xDO)
177       (while e1
178         (do
179         (set v (eval (head e1)))
180         (set e1 (tail e1))))
181     (if (= op xADD)
182       (set v (+ (eval (arg1 e1)) (eval (arg2 e1))))
183     (if (= op xCALL)
184       (do
185       (while e1             ; eval all arg
186         (do                 ; and push it to stack
187         (push (eval (head e1)))
188         (set e1 (tail e1))))
189       (set v (eval arg)))   ; eval body of fun
190     (if (= op xLIT)
191       (set v arg)
192     (if (= op xSTR)
193       (set v arg)
194     (if (= op xGET)
```

```
195       (set v (vec M (- FP arg)))
196     (if (= op xLD)
197       (set v (vec M arg))
198     (if (= op xST)
199       (do
200       (set v (eval (arg1 e1)))
201       (setv M arg v))
202     (if (= op xLDX)
203       (do
204       (set idx (eval (arg1 e1)))
205       (set v (vec M (+ (vec M (- FP arg)) idx))))
206     (if (= op xFUN)
207       (set v (funcall arg e1))
208     (if (= op xSYS)
209       (set v (syscall arg e1))
210     ; else
211     (error "unknown op")))))))))))))
212     v)))
213
214 (def main () ()
215   (do
216   (sys 11)            ; readinfile
217   (loadobj)
218 ;  (listall)
219   (init)
220   (eval (shift Start CS))))
221
222 ; End
223
224 ;  t2.txt
225 ;  input test file for eval.txt
226 ;(enum 10 xAA xBB)
227 ;(let gv tv)
228 ;
229 ;(def prints s ()
230 ;  (if (vec s 0)
231 ;    (do
232 ;    (sys 2 (vec s 0))
233 ;    (prints (+ s 1)))))
234 ;
235 ;(def add1 x () (+ x 1))
236 ;
237 ;(def main () (a)
238 ;  (do
239 ;  (set tv 5)
240 ;  (setv a 1 22)
241 ;  (vec a 2)
242 ;  (setv gv 2 33)
243 ;  (vec gv 3)
244 ;  (prints "string")
245 ;  (sys 1 (add1 xBB))))
```

Appendix E

Code Generator

```
 1 ; gen.txt   n-code to s-code generator
 2 ;   gen completion kit
 3 ;   edit evalx, genwhile
 4
 5 ; include lib.txt
 6
 7 ; -------- header ----------
 8
 9 (enum 127 EOF)
10 (enum 0 NIL)
11 (enum 1000 MAXSYS)          ; start of DS for s-code
12 (enum 3000 MEMMAX)
13
14 ; n-code
15 (enum 1
16   xIF xWHILE xDO xUD1 xNEW xADD xSUB xMUL xDIV
17   xEQ xLT xGT xCALL xGET xPUT xLIT xLDX xSTX xFUN
18   xSYS xSET xSETV xVEC xUD2 xLD xST xLDY xSTY xUD3
19   xUD4 xUD5 xSTR xBAND xSHR xSHL)
20
21 ; s-code used with som-v2
22 (enum 1
23   icAdd icSub icMul icDiv icBand icBor icBxor icNot icEq
24   icNe icLt icLe icGe icGt icShl icShr icMod icLdx icStx
25   icRet icRetv icArray icEnd icGet icPut icLd icSt icJmp icJt
26   icJf icLit icCall icUd1 icInc icDec icSys icUd2 icFun)
27
28 (enum 5678920 magic)        ; header of s-code v2 object
29
30 (let tok DP Dend CS)
31 (let XS XP)                 ; s-code area, pointer
32 (let atab numLab)           ; assoc table
33
34 (def init () ()
35   (do
36   (set XS (new MEMMAX))     ; s-code
37   (set XP 3)                ; s-code pointer
38   (set atab (new 1000))     ; assoc table, max 490 labels
39   (set numLab 0)))
```

```
40
41 ; ----- system --------
42
43 (def tokenise () ()
44   (set tok (sys 3)))
45
46 (def prList e () (sys 10 e))
47
48 (def de_op x () (& (>> x 24) 127))
49 (def de_arg x () (& x 16777215))   ; x & 0x0ffffff
50
51 (def isATOM e () (< e 0))    ; MSB bit 1
52 (def mkATOM (op arg) ()
53   (+ (<< (+ (& op 127) 128) 24) (& arg 16777215)))
54
55 ; ------- load object -----------
56
57 (let Start)
58
59 (def read () ()
60   (do
61   (tokenise)
62   (atoi tok)))
63
64 (def shift (a disp) ()
65   (if a
66     (- (+ a disp) 2)
67     0))
68
69 (def reName (op arg) (v v2)
70   (do
71   (set v2 (+ arg MAXSYS))    ; for DS
72   (if (= op xCALL) (set v (shift arg CS))
73   (if (= op xLD) (set v v2)
74   (if (= op xST) (set v v2)
75   (if (= op xLDY) (set v v2)
76   (if (= op xSTY) (set v v2)
77   (if (= op xSTR) (set v v2)
78   ; else
79   (set v arg)))))))
80   (mkATOM op v)))
81
82 (def loadobj () (flag end a a2 ads ty op arg next)
83   (do
84   (set CS (sys 9))           ; start of code segment
85   (set Start (read))
86   (set end (read))
87   (set DP (+ CS end))
88   (set flag 1)
89   (while flag
90     (do
```

```
 91     (set ads (read))
 92     (set ty (read))
 93     (set op (read))
 94     (set arg (read))
 95     (set next (read))
 96     (if ty
 97       (set a (reName op arg))
 98       ; else dot-pair
 99       (set a (shift (+ (<< op 24) arg) CS)))
100     (set a2 (new 2))
101     (sethead a2 a)
102     (settail a2 (shift next CS))
103     (if (= ads end) (set flag 0))))
104
105     ; load data segment
106     (set a 0)
107     (set Dend (read))
108     (while (< a Dend)
109       (do
110       (set a2 (new 1))
111       (setv a2 0 (read))
112       (set a (+ a 1))))))
113
114 ; ---- assoc table for label -----
115
116 (enum 2 esize)
117 (enum 490 MAXLAB)           ; max no of label
118
119 ; search assoc for n1
120 ; if found, return n2, else 0
121 (def assoc n1 (i flag end)
122   (do
123   (set i 2)                 ; start at 2
124   (set flag 1)
125   (set end (+ (* esize numLab) 2))
126   (while (and flag (< i end))
127     (if (= (vec atab i) n1) ; sequential search
128       (set flag 0)
129       ; else
130       (set i (+ i esize))))
131   (if flag
132     0                       ; not found
133     (vec atab (+ i 1)))))   ; found, return n2
134
135 (def insertLab (n1 n2) (i)
136   (do
137   (set i (+ (* numLab esize) 2)) ; start at 2
138   (setv atab i n1)
139   (setv atab (+ i 1) n2)
140   (set numLab (+ numLab 1))
141   (if (> numLab MAXLAB)
```

```
142     (error "label table full"))))
143
144 (def dumpassoc () (i end)
145   (do
146   (set i 2)
147   (set end (+ (* numLab esize) 2))
148   (while (< i end)
149     (do
150     (print (vec atab i)) (space)
151     (print (vec atab (+ i 1))) (nl)
152     (set i (+ i esize))))))
153
154 ; -------- icode -------------
155
156 (def outa (op arg) ()
157   (do
158   (setv XS XP (+ (<< arg 8) op))
159   (set XP (+ XP 1))))
160
161 (def outs op ()
162   (do
163   (setv XS XP op)
164   (set XP (+ XP 1))))
165
166 ; change arg, preserve op
167 (def patch (ads v) ()
168   (setv XS ads (+ (<< v 8) (& (vec XS ads) 255))))
169
170 ; --------- eval ----------
171
172 (def eval e () 0)           ; forward declaration
173
174 (def genbop (op e1 e2) ()
175   (do
176   (eval e1)
177   (eval e2)
178   (outs op)))
179
180 (def genuop (op arg e) ()
181   (do
182   (eval e)
183   (outa op arg)))
184
185 ; e = (cond true false)
186 (def genif e (ads e3)
187   (do
188   (eval (head e))           ; gen cond
189   (outa icJf 0)
190   (set ads (- XP 1))
191   (eval (arg2 e))           ; gen if-true
192   (set e3 (arg3 e))
```

```
193   (if (= e3 NIL)
194     (patch ads (- XP ads))
195     ; else
196     (do                       ; else
197     (outa icJmp 0)
198     (patch ads (- XP ads))
199     (set ads (- XP 1))
200     (eval e3)                 ; gen else
201     (patch ads (- XP ads))))))
202
203 ; stub definition
204 (def genwhile (e) (ads) 0)
205
206 ; stub definition, edit at 0
207 (def evalx (op arg e) ()     ; e is arg-list
208   (if (= op xNEW) 0
209   (if (= op xSUB) 0
210   (if (= op xMUL) 0
211   (if (= op xDIV) 0
212   (if (= op xBAND) 0
213   (if (= op xSHR) 0
214   (if (= op xSHL) 0
215   (if (= op xEQ) 0
216   (if (= op xLT) 0
217   (if (= op xGT) 0
218   (if (= op xSTX) 0
219   (if (= op xLDY) 0
220   (if (= op xSTY) 0
221   ; else
222   (error "unknown op")))))))))))))))
223
224 (def eval e (ads e1 op arg lv arity)
225   (if (= e NIL)
226     NIL
227     ; else
228     (do
229     (set ads e)
230     (if (not (isATOM e))
231       (do
232       (set e1 (tail e))
233       (set e (head e))))
234     (set op (de_op e))
235     (set arg (de_arg e))
236 ; (prList e)
237     (if (= op xIF) (genif e1)
238     (if (= op xWHILE) (genwhile e1)
239     (if (= op xDO)
240       (while e1
241         (do
242         (eval (head e1))
243         (set e1 (tail e1))))
```

```
244     (if (= op xADD)
245       (genbop icAdd (head e1) (arg2 e1))
246     (if (= op xCALL)
247       (do
248       (while e1
249         (do
250         (eval (head e1))
251         (set e1 (tail e1))))
252       (outa icCall (assoc arg)))
253     (if (= op xLIT) (outa icLit arg)
254     (if (= op xSTR) (outa icLit arg)
255     (if (= op xGET) (outa icGet arg)
256     (if (= op xPUT) (genuop icPut arg (head e1))
257     (if (= op xLD) (outa icLd arg)
258     (if (= op xST) (genuop icSt arg (head e1))
259     (if (= op xLDX)
260       (do
261       (outa icGet arg)
262       (eval (head e1))
263       (outs icLdx))
264     (if (= op xFUN)
265        (do
266        (insertLab ads XP)
267        (set lv (& arg 255))
268        (set arity (>> arg 8))
269        (outa icFun (+ (- lv arity) 1))
270        (eval (head e1))
271        (outa icRet (+ lv 1)))
272     (if (= op xSYS) (genuop icSys arg (head e1))
273     ; else
274     (evalx op arg e1)))))))))))))))
275     )))
276
277 (def outsobj () (i end)
278   (do
279   (print magic) (nl)
280   (print 1) (space)
281   (print (- XP 1)) (nl)
282   (set i 1)
283   (while (< i XP)
284     (do
285     (print (vec XS i)) (space)
286     (if (= (& i 7) 0) (nl))
287     (set i (+ i 1))))
288   (nl)
289   (print MAXSYS) (space)           ; data segment
290   (print (+ MAXSYS (- Dend 1))) (nl)
291   (set i DP)
292   (set end (+ DP Dend))
293   (while (< i end)
294     (do
```

```
295     (print (vec i 0)) (space)
296     (if (= (& i 7) 0) (nl))
297     (set i (+ i 1))))
298   (nl)))
299
300 (def genall () (i op end)
301   (do
302   (set i CS)
303   (set end DP)
304   (while (< i end)
305     (do
306     (set op (de_op (vec i 0)))
307     (if (= op xFUN) (eval i))
308     (set i (+ i 2))))
309   (set i XP)
310   (set XP 1)
311   (outa icCall (assoc (shift Start CS)))
312   (outs icEnd)
313   (set XP i)))
314
315 (def main () ()
316   (do
317   (sys 11)            ; readinfile
318   (loadobj)
319   (init)
320   (genall)
321   (outsobj)))
322
323 ; End
```

Appendix F

Code Generator Completion Solution

```
; solution for codegen completion task

(def genwhile e (ads)
  (do
  (outa icJmp 0)
  (set ads (- XP 1))
  (eval (arg2 e))                  ; gen body
  (patch ads (- XP ads))
  (eval (head e))                  ; gen cond
  (outa icJt (- (+ ads 1) XP))))

(def evalx (op arg e1) ()
    (if (= op xNEW)
      (do
      (eval (head e1))
      (outs icArray))
    (if (= op xSUB) (genbop icSub (head e1) (arg2 e1))
    (if (= op xMUL) (genbop icMul (head e1) (arg2 e1))
    (if (= op xDIV) (genbop icDiv (head e1) (arg2 e1))
    (if (= op xBAND) (genbop icBand (head e1) (arg2 e1))
    (if (= op xSHR) (genbop icShr (head e1) (arg2 e1))
    (if (= op xSHL) (genbop icShl (head e1) (arg2 e1))
    (if (= op xEQ) (genbop icEq (head e1) (arg2 e1))
    (if (= op xLT) (genbop icLt (head e1) (arg2 e1))
    (if (= op xGT) (genbop icGt (head e1) (arg2 e1))
    (if (= op xSTX)                ; base idx val
      (do
      (outa icGet arg)
      (eval (head e1))
      (eval (arg2 e1))
      (outs icStx))
    (if (= op xLDY)
      (do
      (outa icLd arg)
      (eval (head e1))
      (outs icLdx))
    (if (= op xSTY)                ; base idx val
      (do
      (outa icLd arg)
      (eval (head e1))
```

```
41       (eval (arg2 e1))
42       (outs icStx))
43     (error "unknown op")))))))))))))))
44
45 ; End
```

Appendix G

Sx Simulator

.

```
 1 /* Sx cpu micro architecture simulator
 2
 3    P. Chongstitvatana
 4    Department of Computer Engineering
 5    Chulalongkorn University
 6 */
 7
 8 #include "sx.h"
 9
10 void run(void){
11     while(runflag) {
12         run3();
13         ninst++;
14     }
15 }
16
17 void initchip(void){
18     clock = 0;
19     ninst = 0;
20     runflag = 1;
21     TS = 0;
22     NX = 0;
23     FF = 0;
24     FP = SSBASE;
25     SP = SSBASE;
26     PC = 1;
27     Z = 0;
28     S = 0;
29     mpc = 0;
30 }
31
32 int main(int argc, char *argv[]){
33   if( argc < 2 ) {
34     printf("usage : sx objfile\n");
35     exit(0);
36   }
37   loadobj(argv[1]);
38   initchip();
39   run();
```

```
40   return 0;
41 }
42
43 // ----------------------------------------
44
45 // Sx microprogram simulation
46
47 #include "sx.h"
48 #include "sxbit.h"          // signal and rom definition
49
50 #define  PRIVATE            static
51
52 int  TS, FP, SP, NX, FF, IR, PC, Z, S, M[MAXMEM];
53 int  mpc;                   // micro PC
54 int  dbus,tbus,p1,p2,abus;   // internal bus, alu ports
55 int  bus,pcin;              // wires
56 int  clock, ninst, runflag;
57
58 // int udop[], mw[], mx[], nxt[]  are defined in sxbit.h
59
60 // IR bits
61 int IRop(void)  { return( IR & 255 ); } // bit 7..0
62 int IRarg(void)  { return( IR >> 8 ); } // signx bit 31..8
63
64 /* alu operation input : a, b
65  affect condition code Z,S
66  order of operand  ts = ff op ts
67  ff is the first operand !
68  */
69 int alu(int op, int a, int b){
70   int t;
71   switch(op) {
72   case icAdd: t = b + a; break;
73   case icSub: t = b - a; break;  // ordering!
74   case icMul: t = b * a; break;
75   case icDiv: t = b / a; break;
76   case icBand: t = b & a; break;
77   case icBor: t = b | a; break;
78   case icBxor: t = b ^ a; break;
79   case icNot: t = ! a; break;
80   case icShl: t = b << a; break;
81   case icShr: t = b >> a; break;
82   case icEq: t = b == a; break;
83   case icNe: t = b != a; break;
84   case icLt: t = b < a; break;
85   case icLe: t = b <= a; break;
86   case icGt: t = b > a; break;
87   case icGe: t = b >= a; break;
88   case icInc: t = a + 1; break;
89   case icDec: t = a - 1; break;
90   case FSUB: t = a - b; break;
```

```
 91   case FP1: t = a; break;
 92   case FP2: t = b; break;
 93   case FZ: t = a == 0; break;
 94   }
 95   if( t == 0 ) Z = 1; else Z = 0;
 96   if( t < 0 )  S = 1; else S = 0;
 97   return t;
 98 }
 99
100 // decode instruction to microprogram address
101 PRIVATE int udecode(void){
102   int a;
103   a = udop[IRop()];
104   if( a == 0 )
105     error("undefined opcode");
106   return a;
107 }
108
109 // run3 is the new event-list simulator
110 // run event of each field
111 // execute one instruction (a number of microstep)
112 void run3(void){
113   int m2, k, s;
114   while(1) {
115     clock++;
116     m2 = nxt[mpc];
117     k = mw[mpc];             // begin of each event-list
118     s = mx[k];
119     while( s >= 0 ){         //  for all events in one word
120         switch(s){           //  for each event
121         case s_x_ts:  p1 = TS; break;
122         case s_x_fp:  p1 = FP; break;
123         case s_x_sp:  p1 = SP; break;
124         case s_x_nx:  p1 = NX; break;
125         case s_y_ff:  p2 = FF; break;
126         case s_y_arg: p2 = IRarg(); break;
127
128         case s_alu_add: tbus = alu(icAdd,p1,p2); break;
129         case s_alu_sub: tbus = alu(FSUB,p1,p2); break;
130         case s_alu_inc: tbus = alu(icInc,p1,p2); break;
131         case s_alu_dec: tbus = alu(icDec,p1,p2); break;
132         case s_alu_z:   tbus = alu(FZ,p1,p2); break;
133         case s_alu_eq:  tbus = alu(icEq,p1,p2); break;
134         case s_alu_p1:  tbus = alu(FP1,p1,p2); break;
135         case s_alu_p2:  tbus = alu(FP2,p1,p2); break;
136         case s_alu_op:  tbus = alu(IRop(),p1,p2); break;
137
138         case s_a_pc:    abus = PC; break;
139         case s_a_tbus:  abus = tbus; break;
140         case s_d_ts:    dbus = TS; break;
141         case s_d_fp:    dbus = FP; break;
```

```
142         case s_mR:      dbus = M[abus]; break;
143         case s_mW:      M[abus] = dbus; break;
144
145         case s_b_tbus:  bus = tbus; break;
146         case s_b_dbus:  bus = dbus; break;
147         case s_b_pc:    bus = PC; break;
148         case s_j_pc1:   pcin = PC + 1; break;
149         case s_j_pcarg: pcin = PC + IRarg(); break;
150         case s_j_tbus:  pcin = tbus; break;
151
152         case s_lpc:     PC = pcin; break;
153         case s_lir:     IR = dbus; break;
154         case s_lts:     TS = bus; break;
155         case s_lfp:     FP = bus; break;
156         case s_lsp:     SP = bus; break;
157         case s_lnx:     NX = bus; break;
158         case s_lff:     FF = bus; break;
159
160         case s_ifT:     m2 = (Z == 0) ? m2 : mpc+1; break;
161         case s_ifF:     m2 = (Z == 1) ? m2 : mpc+1; break;
162         case s_decode:  m2 = udecode(); break;
163         case s_trap:    trap(IRop(),IRarg()); break;
164         }
165         k++;
166         s = mx[k];
167     }
168     mpc = m2;         // next mpc
169     if(mpc == 0) break;
170   }
171 }
172 /*
173 // order of events on data path, use in "sxgen"
174 PRIVATE int sig[] = {
175     s_x_ts, s_x_fp, s_x_sp, s_x_nx, s_y_ff, s_y_arg,
176     s_alu_add, s_alu_sub, s_alu_inc, s_alu_dec, s_alu_z,
177     s_alu_eq, s_alu_p1, s_alu_p2, s_alu_op,
178     s_a_pc, s_a_tbus, s_d_fp, s_d_ts,
179     s_mR, s_mW, s_b_dbus, s_b_tbus, s_b_pc,
180     s_j_pc1, s_j_pcarg, s_j_tbus,
181     s_lpc, s_lir, s_lts, s_lfp, s_lsp, s_lnx, s_lff,
182     s_ifT, s_ifF, s_decode, s_trap, -1
183 };
184 */
185 // End
```

Appendix H

Microprogram

```
 1 .. sx microprogram
 2 ..
 3 .s
 4 .. -------- signal definitions --------
 5 x.ts
 6 x.fp
 7 x.sp
 8 x.nx
 9 y.ff
10 y.arg
11 b.tbus
12 b.dbus
13 b.pc
14 d.fp
15 d.ts
16 a.tbus
17 a.pc
18 j.pc1
19 j.pcarg
20 j.tbus
21 alu.add
22 alu.sub
23 alu.inc
24 alu.dec
25 alu.z
26 alu.eq
27 alu.op
28 alu.p1
29 alu.p2
30 .. load registers
31 lir
32 lts
33 lfp
34 lsp
35 lnx
36 lff
37 lpc
38 mR
39 mW
```

```
40 .. next micro ads
41 ifT
42 ifF
43 decode
44 trap
45 .m
46 ..  -------- micro program --------
47 :fetch
48   a.pc mR lir decode ;
49 :bop
50   x.sp alu.p1 a.tbus mR b.dbus lff ;
51   x.sp alu.dec b.tbus lsp ;
52   x.ts y.ff alu.op b.tbus lts j.pc1 lpc /fetch ;
53 :uop
54   x.ts alu.op b.tbus lts j.pc1 lpc /fetch ;
55 :get
56   x.sp alu.inc b.tbus lsp ;
57   d.ts x.sp alu.p1 a.tbus mW ;
58   x.fp y.arg alu.sub a.tbus mR b.dbus lts j.pc1 lpc /fetch ;
59 :put
60   x.fp y.arg alu.sub a.tbus d.ts mW ;
61 :popts
62   x.sp alu.p1 a.tbus mR b.dbus lts ;
63   x.sp alu.dec b.tbus lsp j.pc1 lpc /fetch ;
64 :ld
65   x.sp alu.inc b.tbus lsp ;
66   d.ts x.sp alu.p1 a.tbus mW ;
67   y.arg alu.p2 a.tbus mR b.dbus lts j.pc1 lpc /fetch ;
68 :st
69   d.ts y.arg alu.p2 a.tbus mW /popts ;
70 :ldx
71   x.sp alu.p1 a.tbus mR b.dbus lff ;
72   x.sp alu.dec b.tbus lsp ;
73   x.ts y.ff alu.add a.tbus mR b.dbus lts j.pc1 lpc /fetch ;
74 :stx
75   x.sp alu.p1 a.tbus mR b.dbus lnx ;
76   x.sp alu.dec b.tbus lsp ;
77   x.sp alu.p1 a.tbus mR b.dbus lff ;
78   x.nx y.ff alu.add a.tbus d.ts mW ;
79   x.sp alu.dec b.tbus lsp /popts ;
80 :lit
81   x.sp alu.inc b.tbus lsp ;
82   d.ts x.sp alu.p1 a.tbus mW ;
83   y.arg alu.p2 b.tbus lts j.pc1 lpc /fetch ;
84 :jmp
85   j.pcarg lpc /fetch ;
86 :jt
87   x.ts alu.z ifT /j3 ;
88 :j2
89 ..  <jump>
90   j.pcarg lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
```

```
 91   x.sp alu.dec b.tbus lsp /fetch ;
 92 :jf
 93   x.ts alu.z ifT /j2 ;
 94 :j3
 95 ..  <don't jump>
 96   j.pc1 lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
 97   x.sp alu.dec b.tbus lsp /fetch ;
 98 :call
 99   x.sp alu.inc b.tbus lsp ;
100   x.sp alu.p1 a.tbus d.ts mW j.pc1 lpc ;
101   b.pc lts ;
102   y.arg alu.p2 b.tbus lnx a.tbus mR lir ;
103   x.sp y.arg alu.add a.tbus d.fp mW ;
104   x.sp y.arg alu.add b.tbus lfp lsp ;
105   x.nx alu.inc j.tbus lpc /fetch ;
106 :ret
107   x.sp alu.p1 b.tbus lff ;
108   x.fp y.ff alu.eq ifF /r2 ;
109   x.ts alu.p1 j.tbus lpc ;
110   x.fp y.arg alu.sub b.tbus lsp ;
111   x.sp alu.p1 a.tbus mR b.dbus lts ;
112   x.sp alu.dec b.tbus lsp ;
113   x.fp alu.p1 a.tbus mR b.dbus lfp /fetch ;
114 :r2
115   x.fp alu.inc a.tbus mR b.dbus lff ;
116   y.ff alu.p2 j.tbus lpc ;
117   x.fp y.arg alu.sub b.tbus lsp ;
118   x.fp alu.p1 a.tbus mR b.dbus lfp /fetch ;
119 :sys
120 :array
121 :end
122   trap j.pc1 lpc /fetch ;
123 ..
124 .e
125 .. ==================================
126 .. sx2 microprogram
127 ..
128 .s
129 .. -------- signal definition --------
130 x.ts
131 x.fp
132 x.nx
133 y.ff
134 y.arg
135 y.u
136 b.tbus
137 b.dbus
138 b.sp
139 d.fp
140 d.ts
141 d.v
```

```
142 d.u
143 a.tbus
144 a.pc
145 a.sp
146 a.fp
147 j.pc1
148 j.pcarg
149 j.tbus
150 so.sp
151 so.spx
152 si.inc
153 si.dec
154 si.k
155 si.tbus
156 w.v1
157 w.v2
158 w.v3
159 w.v4
160 w.varg
161 z.dbus
162 z.ts
163 t.v
164 t.pc
165 t.bus
166 u.iru
167 u.dbus
168 alu.add
169 alu.sub
170 alu.inc
171 alu.dec
172 alu.add2
173 alu.z
174 alu.eq
175 alu.op
176 alu.p1
177 alu.p2
178 .. load registers
179 lir
180 lts
181 lfp
182 lsp
183 lnx
184 lff
185 lpc
186 lv1
187 lv2
188 lv3
189 lv4
190 lvarg
191 lu
192 mR
```

```
193 mW
194 .. next micro ads
195 ifT
196 ifF
197 ifu0
198 ifp0
199 ifargm
200 skipu
201 decode
202 trap
203 .m
204 .. -------- micro program --------
205 :fetch
206   a.pc mR lir decode ;
207 :bop
208   so.sp si.dec lsp a.sp mR b.dbus lff ;
209   x.ts y.ff alu.op b.tbus t.bus lts j.pc1 lpc /fetch ;
210 :uop
211   x.ts alu.op b.tbus t.bus lts j.pc1 lpc /fetch ;
212 :get
213   so.spx si.inc lsp a.sp d.ts mW ;
214   x.fp y.arg alu.sub a.tbus mR b.dbus t.bus lts j.pc1 lpc /fetch ;
215 :get1
216 so.spx si.inc lsp a.sp d.ts mW w.v1 t.v lts j.pc1 lpc /fetch;
217 :get2
218 so.spx si.inc lsp a.sp d.ts mW w.v2 t.v lts j.pc1 lpc /fetch;
219 :get3
220 so.spx si.inc lsp a.sp d.ts mW w.v3 t.v lts j.pc1 lpc /fetch;
221 :get4
222 so.spx si.inc lsp a.sp d.ts mW w.v4 t.v lts j.pc1 lpc /fetch;
223 :put
224   x.fp y.arg alu.sub a.tbus d.ts mW ;
225 :popts
226  so.sp si.dec lsp a.sp mR b.dbus t.bus lts j.pc1 lpc /fetch ;
227 :put1
228 z.ts lv1 so.sp si.dec lsp a.sp mR b.dbus t.bus lts j.pc1 lpc /fetch ;
229 :put2
230 z.ts lv2 so.sp si.dec lsp a.sp mR b.dbus t.bus lts j.pc1 lpc /fetch ;
231 :put3
232 z.ts lv3 so.sp si.dec lsp a.sp mR b.dbus t.bus lts j.pc1 lpc /fetch ;
233 :put4
234 z.ts lv4 so.sp si.dec lsp a.sp mR b.dbus t.bus lts j.pc1 lpc /fetch ;
235 :ld
236   so.spx si.inc lsp a.sp d.ts mW ;
237   y.arg alu.p2 a.tbus mR b.dbus t.bus lts j.pc1 lpc /fetch ;
238 :st
239   d.ts y.arg alu.p2 a.tbus mW /popts ;
240 :ldx
241   so.sp si.dec lsp a.sp mR b.dbus lff ;
242   x.ts y.ff alu.add a.tbus mR b.dbus t.bus lts j.pc1 lpc /fetch ;
243 :stx
```

```
 244   so.sp si.dec lsp a.sp mR b.dbus lnx ;
 245   so.sp si.dec lsp a.sp mR b.dbus lff ;
 246   x.nx y.ff alu.add a.tbus d.ts mW /popts ;
 247 :lit
 248  so.spx si.inc lsp a.sp d.ts mW y.arg alu.p2 b.tbus t.bus lts
j.pc1 lpc /fetch ;
 249 :jmp
 250   j.pcarg lpc /fetch ;
 251 :jt
 252   x.ts alu.z ifT /j3 ;
 253 :j2
 254 ..  <jump>
 255 j.pcarg lpc so.sp si.dec lsp a.sp mR b.dbus t.bus lts /fetch;
 256 :jf
 257   x.ts alu.z ifT /j2 ;
 258 :j3
 259 ..  <don't jump>
 260  j.pc1 lpc so.sp si.dec lsp a.sp mR b.dbus t.bus lts /fetch ;
 261 :call
 262   so.spx si.inc lsp a.sp d.ts mW j.pc1 lpc ;
 263   t.pc lts y.arg alu.p2 j.tbus lpc ifu0 /fetch ;
 264 .. <save v>
 265   x.fp y.u alu.sub b.tbus lfp skipu ;
 266   w.v4 a.fp d.v mW x.fp alu.inc b.tbus lfp ;
 267   w.v3 a.fp d.v mW x.fp alu.inc b.tbus lfp ;
 268   w.v2 a.fp d.v mW x.fp alu.inc b.tbus lfp ;
 269   w.v1 a.fp d.v mW x.fp alu.inc b.tbus lfp /fetch ;
 270 :fun
 271   si.k lsp so.spx a.sp d.fp mW ;
 272   b.sp lfp ;
 273   si.inc so.spx lsp a.sp d.u mW u.iru lu ;
 274   j.pc1 lpc ifp0 /fetch ;
 275 :cachev
 276   x.fp y.u alu.sub b.tbus lfp skipu ;
 277   a.fp mR z.dbus lv4 x.fp alu.inc b.tbus lfp ;
 278   a.fp mR z.dbus lv3 x.fp alu.inc b.tbus lfp ;
 279   a.fp mR z.dbus lv2 x.fp alu.inc b.tbus lfp ;
 280   a.fp mR z.dbus lv1 x.fp alu.inc b.tbus lfp /fetch ;
 281 :ret
 282   si.dec so.spx b.sp lff ;
 283   x.fp y.ff alu.eq ifF /r2 ;
 284 .. <doret>
 285   x.ts alu.p1 j.tbus lpc ;
 286   so.sp a.sp mR u.dbus lu ;
 287   x.fp y.arg alu.sub si.tbus lsp ;
 288   so.sp si.dec lsp a.sp mR b.dbus t.bus lts ifu0 /r3 ;
 289   a.fp mR b.dbus lfp /cachev ;
 290 :r2
 291 .. <doretv>
 292   x.fp alu.add2 a.tbus mR b.dbus lff ;
 293   y.ff alu.p2 j.tbus lpc ;
```

```
294   x.fp alu.inc a.tbus mR u.dbus lu ;
295   x.fp y.arg alu.sub si.tbus lsp ifu0 /r3 ;
296   a.fp mR b.dbus lfp /cachev ;
297 :r3
298 ..  <ret end>
299   a.fp mR b.dbus lfp /fetch ;
300 :inc
301   x.ts alu.p1 b.tbus lnx w.varg t.v lts ifargm /inc2 ;
302   x.ts alu.inc b.tbus t.bus lts ;
303   z.ts lvarg x.nx alu.p1 b.tbus t.bus lts j.pc1 lpc /fetch ;
304 :inc2
305 ..  <update SS>
306   x.fp y.arg alu.sub a.tbus mR b.dbus t.bus lts ;
307   x.ts alu.inc b.tbus t.bus lts ;
308   x.fp y.arg alu.sub a.tbus d.ts mW ;
309   x.nx alu.p1 b.tbus t.bus lts j.pc1 lpc /fetch ;
310 :dec
311   x.ts alu.p1 b.tbus lnx w.varg t.v lts ifargm /dec2 ;
312   x.ts alu.dec b.tbus t.bus lts ;
313   z.ts lvarg x.nx alu.p1 b.tbus t.bus lts j.pc1 lpc /fetch ;
314 :dec2
315 ..  <update SS>
316   x.fp y.arg alu.sub a.tbus mR b.dbus t.bus lts ;
317   x.ts alu.dec b.tbus t.bus lts ;
318   x.fp y.arg alu.sub a.tbus d.ts mW ;
319   x.nx alu.p1 b.tbus t.bus lts j.pc1 lpc /fetch ;
320 :sys
321 :array
322 :end
323   trap j.pc1 lpc /fetch ;
324 ..
325 .e
326
327 .. -------- extended instructions --------
328
329 :ldxv
330   x.sp alu.inc b.tbus lsp ;
331   d.ts x.sp alu.p1 a.tbus mW ;
332   x.fp y.a1 alu.sub a.tbus mR b.dbus lts ;
333   x.ts y.a2 alu.add a.tbus mR b.dbus lts j.pc1 lpc /fetch ;
334
335 :stxv
336   x.fp y.a1 alu.sub a.tbus mR b.dbus lnx ;
337   x.nx y.a2 alu.add a.tbus d.ts mW /popts ;
338
339 :jne
340   x.sp alu.p1 a.tbus mR b.dbus lff ;
341   x.sp alu.dec b.tbus lsp ;
342   x.ts y.ff alu.eq ifT /j3 ;
343   j.pcarg lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
344   x.sp alu.dec b.tbus lsp /fetch ;
```

```
345
346 :seqi
347   x.sp alu.inc b.tbus lsp j.pc1 lpc ;
348   d.ts x.sp alu.p1 a.tbus mW ;
349   x.fp y.a1 alu.sub a.tbus mR b.dbus lnx ;
350   x.nx alu.p1 a.tbus mR b.dbus lts lff ;
351   x.ts alu.z ifT /fetch ;
352   x.fp y.a2 alu.sub a.tbus mR b.dbus lnx ;
353   x.nx alu.p1 a.tbus mR b.dbus lts ;
354   x.ts alu.z ifT /fetch ;
355   x.ts y.ff alu.eq b.tbus lts ifF /fetch ;
356   x.nx alu.inc b.tbus lts ;
357   x.fp y.a2 alu.sub a.tbus d.ts mW ;
358   x.fp y.a1 alu.sub a.tbus mR b.dbus lnx ;
359   x.nx alu.inc b.tbus lts lff ;
360   x.fp y.a1 alu.sub a.tbus d.ts mW ;
361   x.nx y.ff alu.ne b.tbus lts /fetch ;
362
363 .. End
```

Appendix I

NOS Supervisor

```
 1 /* noss.c     nos supervisor
 2
 3  */
 4
 5 #include "sx.h"
 6
 7 #define PSW  1              // noss state
 8 #define USER 2
 9
10 extern int clock;           // sxm.c
11
12 int a_activep, a_status, a_psw; // binding to user space
13 int intflag;                 // interrup flag
14 int dt;                     // interrupt timer
15 int noss_state;
16 int userp;                  // pdes of current user process
17 int sclock = 0;             // count time spent in switchp
18 int n_switch = 0;
19
20 // save c-state to process descriptor
21 PRIVATE void saveCstate(int a){
22     M[a + 4] = FP;
23     M[a + 5] = SP;
24     M[a + 6] = PC;
25     M[a + 7] = TS;
26 }
27
28 // restore c-state from a process desc.
29 PRIVATE void restoreCstate(int a){
30     FP = M[a + 4];
31     SP = M[a + 5];
32     PC = M[a + 6];    // cause a jump
33     TS = M[a + 7];
34 }
35
36 PRIVATE void stat(void){
37     sclock += (clock-dt);
38     printf(" * ");
39     n_switch++;
```

```
40 }
41
42 // event = time_out, stopped, blocked
43 void noss(int event){
44     if(M[a_activep] == 0){          // no process
45         runflag = 0;                // stop simulation
46         return;
47     }
48     M[a_status] = event;            // update to nos
49     switch(noss_state){
50     case PSW:                       // run task switcher
51         saveCstate(userp);
52         restoreCstate(M[a_psw]);
53         noss_state = USER;
54         break;
55     case USER:                      // run user process
56         userp = M[a_activep];
57         restoreCstate(userp);
58         intflag = 1;                // enable interrupt
59         noss_state = PSW;
60         stat();                     // collect statistics
61     }
62     dt = clock;                     // reset timer
63 }
64
65 void yield(void){
66     if(intflag && (clock-dt) > TIMEOUT)
67         noss(time_out);
68 }
69
70 // processor execution
71 void eval(int ref){
72     PC = ref;
73     while(runflag ) {
74         run3();                     // execute one instruction
75         yield();
76     }
77 }
78
79 void trap(int op, int arg){         // special functions
80     switch(op) {
81     case icEnd : noss(stopped);    break;
82     case icArray: TS = xalloc(TS); break;
83     case icSys:
84         switch(arg){
85         case 1:                     // print ts
86             printf("%d",TS);
87             popts();
88             break;
89         case 2 :                    // printc ts
90             printf("%c",TS);
```

```
 91             popts();
 92             break;
 93         case 13: runflag = 0; break;
 94         case 20: intflag = 1; break;
 95         case 21: intflag = 0; break;
 96         case 22: noss(blocked); break;
 97         default:
 98             error("unknown syscall");
 99         }
100         break;
101     default:
102         error("unknown trap");
103     }
104 }
105
106 PRIVATE void boot(void){
107     a_activep = searchsym("activep");
108     a_status = searchsym("status");
109     a_psw = searchsym("psw");
110     dt = 0;
111     intflag = 0;
112     noss_state = USER;
113     eval(1);                        // boot nos
114 }
115
116 int main(int argc, char *argv[]){
117   if( argc < 2 ) {
118     printf("usage : noss objfile\n");
119     exit(0);
120   }
121   loadobj(argv[1]);
122   initchip();
123   boot();
124   return 0;
125 }
126 // End
```

Appendix J

Nut Operating System (NOS)

```
 1 ;; nos   nut operating system
 2 ;;
 3
 4 (def print (x) () (sys 1 x))
 5 (def printc (c) () (sys 2 c))
 6 (def nl () () (sys 2 10))
 7 (def space () () (sys 2 32))
 8 (def not (b) () (if b 0 1))
 9 (def != (a b) () (if (= a b) 0 1))
10 (def <= (a b) () (if (> a b) 0 1))
11 (def >= (a b) () (if (< a b) 0 1))
12 (def or (a b) () (if a 1 b))
13
14 ;; ----------------------
15 ;; global var
16
17 ;; activep  the active process
18 ;; status   10 time-out, 11 stopped, 12 blocked
19 ;; pid      number of process created
20
21 (let activep status pid psw)
22 (let sseg                  ;; free stack segment
23
24 (enum 10 TIMEOUT STOPPED BLOCKED)
25
26 ;; ----------------------
27
28 ;; process descriptor
29 ;;   field:
30 ;;   0 next, 1 prev,                double link
31 ;;   2 id,   3 value,
32 ;;   4 fp,   5 sp,   6 ip,  7 ts,    context
33 ;;   8 inbox, 9 awaitbox, 10 msg     mail box
34 ;;
35 ;;   process state (value) :
36 ;;    1 READY, 2 RUNNING, 3 WAIT, 4 DEAD, 5 SEND, 6 RECEIVE
37
38 (enum 1 READY RUNNING WAIT DEAD SEND RECEIVE)
39
```

```
40 (def ei () () (sys 20))            ;; enable int
41 (def di () () (sys 21))            ;; disable int
42 (def blockp () () (sys 22))        ;; block current process
43
44 ;; doubly linked list
45 (def getNext (a) () (vec a 0))
46 (def getPrev (a) () (vec a 1))
47 (def setNext (a v) () (setv a 0 v))
48 (def setPrev (a v) () (setv a 1 v))
49
50 ;; append a2 to the end of a1
51 (def appendDL (a1 a2) (b)
52   (if (= a1 0)
53     (do
54     (setNext a2 a2)  ;; only one item
55     (setPrev a2 a2)
56     a2)
57     ;; else
58     (do
59     (set b (getPrev a1))
60     (setNext a2 a1)
61     (setPrev a1 a2)
62     (setNext b a2)
63     (setPrev a2 b)
64     a1)))
65
66 (def deleteDL (b) (a c)
67   (if (= b (getNext b))
68     0    ;; delete singleton
69     ;; else
70     (do
71     (set a (getPrev b))
72     (set c (getNext b))
73     (setNext a c)
74     (setPrev c a)
75     c)))
76
77 ;; process descriptor access functions
78 (def getId (p) () (vec p 2))
79 (def getValue (p) () (vec p 3))
80 (def setId (p id) () (setv p 2 id))
81 (def setValue (p v) () (setv p 3 v))
82
83 (def newp () (p)
84   (do
85   (set p (new 11))          ;; new pdes
86   (setNext p 0)
87   (setPrev p 0)
88   (setValue p READY)
89   (setv p 4 sseg)           ;; set fp'
90   (setv p 5 (+ sseg 1))     ;; set sp'
```

```
 91   (setv p 6 0)              ;; set ip'
 92   (set sseg (+ sseg 1000))
 93   (setv p 7 0)
 94   (setv p 8 0)
 95   p))
 96
 97 ;; -------- process management -------
 98
 99 ;; show process list a
100 (def showp (a) (p)
101   (do
102   (set p a)
103   (while (!= p 0)
104     (do
105     (print (vec p 2)) (space)
106     (print (vec p 4)) (space)
107     (print (vec p 7)) (space)
108     (print (vec p 8)) (space)
109     (set p (getNext p))
110     (if (= p a) (set p 0))))
111   (nl)))
112
113 ;; return p
114 (def run (ads) (p)
115   (do
116   (set p (newp))                  ;; new pdes
117   (setId p pid)
118   (set pid (+ pid 1))
119   (setv p 6 ads)                  ;; set ip' to call.fun
120   (set activep (appendDL activep p))
121   p))
122
123 (def runnable (p) () (setValue p RUNNING))
124
125 ;; nos sim is responsible to save C-state
126 ;; before running switchp
127 (def switchp () ()
128   (do
129   (di)
130   (if (or (= status TIMEOUT) (= status BLOCKED))
131     (do
132     (setValue activep READY)
133     (set activep (getNext activep)) ;; switch next
134     (runnable activep))
135     ; else                         ;; status STOPPED
136     (do
137     (setValue activep DEAD)
138     (set activep (deleteDL activep))
139     (if (!= activep 0)
140       (runnable activep))))))
141
```

```
142 (def bootnos () ()
143   (runnable activep))
144
145 ;; ---- semaphore ------------
146 ;; field: sval(value) slist(wait-list)
147 ;; semaphore access functions
148
149 (def getsval (s) () (vec s 0))
150 (def getslist (s) () (vec s 1))
151 (def setsval (s v) () (setv s 0 v))
152 (def setslist (s v) () (setv s 1 v))
153
154 (def initsem (v) (s1)
155   (do
156   (set s1 (new 2))
157   (setsval s1 v)
158   (setslist s1 0)     ;; wait-list nil
159   s1))
160
161 (def wakeup (p) ()
162   (do
163   (setValue p READY)
164   (set activep (appendDL activep p))))
165
166 (def signal (s) (p)
167   (do
168   (di)
169   (set p (getslist s))
170   (if (!= p 0)
171     (do
172     (setslist s (deleteDL p))
173     (wakeup p))
174     ;; else
175     (setsval s (+ (getsval s) 1)))
176   (ei)))
177
178 (def wait (s) (v p)
179   (do
180   (di)
181   (set v (getsval s))
182   (if (<= v 0)
183     (do                     ;; block activep to WAIT
184     (set p activep)
185     (set activep (deleteDL activep))
186     (setValue p WAIT)       ;; to wait-list
187     (setslist s (appendDL (getslist s) p))
188     (blockp))               ;; block
189     ;; else
190     (setsval s (- v 1)))
191   (ei)))
192
```

```
193 ;; --------- mailbox ----------
194
195 (def getMbox (p) () (vec p 8))
196 (def getAwait (p) () (vec p 9))
197 (def getMsg (p) () (vec p 10))
198 (def setMbox (p m) () (setv p 8 m))
199 (def setAwait (p m) () (setv p 9 m))
200 (def setMsg (p m) () (setv p 10 m))
201
202 ;; search mail p in the box
203 ;; return mail if found else 0
204 (def findmail (p box) (x y)
205   (do
206   (set y 0)                  ;; ret value
207   (set x box)
208   (while (!= x 0)
209     (if (= x p)
210       (do
211       (set y x)
212       (set x 0))             ;; exit
213       ;; else
214       (do
215       (set x (getNext x))
216       (if (= x box)
217         (set x 0)))))        ;; not found
218   y))
219
220 ;; p is pointer to process
221 (def send (p mess) (m box)
222   (do
223   (di)
224   (set box (getAwait activep))
225   (set m (findmail p box))
226   (if (= m 0)
227     (do
228     (set m activep)   ;; self
229     (setMsg m mess)
230     (set activep (deleteDL activep))
231     (setMbox p (appendDL (getMbox p) m))
232     (setValue m SEND)
233     (blockp))
234     ;; else
235     (do                      ;; p is waiting
236     (setMsg p mess)
237     (set m (deleteDL p))
238     (if (= box p)
239       (setAwait activep m))
240     (wakeup p)))
241   (ei)))
242
243 (def receive (p) (m box)
```

```
244   (do
245   (di)
246   (set box (getMbox activep))
247   (set m (findmail p box))
248   (if (= m 0)
249     (do                     ;; put to await p
250     (set m activep)         ;; self
251     (set activep (deleteDL activep))
252     (setAwait p (appendDL (getAwait p) m))
253     (setValue m RECEIVE)
254     (blockp)
255     (getMsg m))             ;; retrieve from self
256     ;; else
257     (do                     ;; already in mbox
258     (set m (deleteDL p))
259     (if (= box p)
260       (setMbox activep m))
261     (getMsg p)              ;; retrieve mbox
262     (wakeup p)))
263   (ei)))
264
265 ;; ---- application --------
266
267 (let p1 p2)
268
269 ;; send 2..n to p2 ended with -1
270 (def produce (n) (i)
271   (do
272   (set i 2)
273   (while (< i n)
274     (do
275     (printc 33) (print i) (space)
276     (send p2 i)
277     (set i (+ i 1))))
278   (send p2 (- 0 1))))
279
280 ;; receive 2..n from p1 ended with -1
281 (def consume () (m flag)
282   (do
283   (set flag 1)
284   (while flag
285     (do
286     (set m (receive p1))
287     (printc 34) (print m) (space)
288     (if (< m 0)
289       (set flag 0))))
290   (nl)))
291
292 (def main () ()
293   (do
294   (di)
```

```
295   (set activep 0)
296   (set sseg 4000)
297   (set pid 1)
298   (set psw (run (switchp)))
299   (set activep 0)
300   (set p1 (run (produce 100)))
301   (set p2 (run (consume)))
302   (bootnos)))
303
304 ; End
```

References

[AMD64] Amdahl, G., Blaauw, G., and Brooks, F., “Architecture of the IBM System/360”, IBM Journal of Research and Development, April 1964.

[AMD67] Amdahl, G., “Validity of the single processor approach to achieving large scale computing capabilities”, AFIPS Conf. Proc., April 1967, pp. 483-485.

[AHO86] Aho, A., Sethi, R., and Ullman, J., Compilers: Principles, techniques, and tools. Addison-Wesley, 1986.

[BAC78] Backus, J. “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs”, Communications of the ACM, August 1978, 20(8):613-641.

[BEL71] Bell, C., and Newell, A. Computer structure: Readings and examples. McGraw-Hill, 1971.

[BER78] Berry, R., “Experience with the Pascal-P compiler”, Software – Practice and Experience, 8:617-627, 1978.

[BER96] Bergin, T., Gibson, R., Gibson, R. Jr., History of programming languages, vol.2, ACM Press, Addison Wesley, 1996.

[BEL76] Bell, C., and Strecker, W., “Computer structures: What we have learned from the PDP-11”, Proc. of 3rd annual symposium on computer architecture, (1976): 1-14.

[BUR46] Burks, A. W., Goldstein, H. H. and von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument”, US Army Ordnance Department Report 1946.

[BUR88] Burks, A., and Burks, A., The First Electronic Computer: The Atanasoff Story, the University of Michigan Press, Ann Arbor, Michigan, 1988.

[BUR81] Burks, A., and Burks, A., The ENIAC: First General Purpose Electronic Computer, The University of Michigan Press, Ann Arbor, Michigan, 1981.

[BUR68] Burroughs B5500 Electronic Information Processing System: Operation manual. Burroughs Corp. Detroit, 1968.

[BUR01] Burns, A. and Wellings, A., Real-time systems and programming languages, 3rd ed. Addison-Wesley, 2001.

[BUR04a] Burutarchanai, A., and Chongstitvatana, P., "Design of a two-phased clocked control unit for performance enhancement of a stack processor", National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.

[BUR04b] Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., "A fast instruction fetch unit for an embedded stack processor", Proc. of Int. Conf. on Information and Communication Technologies, Thailand, 18-19 November 2004.

[BUR04c] Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON, Thailand, 21-24 November 2004.

[CHO97] Chongstitvatana, P. "Post processing optimization of byte-code instructions by extension of its virtual machine", Conf. of Electrical Engineering, Bangkok, 1997.

[CHO98] Chongstitvatana, P. A multi-tasking environment for real-time control. Final report, Faculty of Engineering, Chulalongkorn university, research project number 132-MRD-2537, 1998. Also available on-line at http: //www.cp.eng.chula.ac.th/faculty/pjw/r1/

[CHO01] Chongstitvatana, P. "Computer Architecture: A synthesis approach," 2001.

[CHO03] Chongstitvatana, P., "The Art of Instruction Set Design", Electrical Engineering Conference, Thailand, 2003.

[CHO05] Chongstitvatana, P., "Self-Generating Systems: How a $10{,}000{,}000_2$-line Compiler Assembles Itself," Proc. of National Computer Science and Engineering Conference, Bangkok, 2005.

[CHO06] Chongstitvatana, P., "Stack frame caching", Proc. of National Computer Science and Engineering Conference, Thailand, 2006. (being written)

[COR62] Corbato, F., Merwin-Daggett, M., and Daley, R., "An experimental time-sharing system", Proc. of the AFIPS Fall Joint Conference, pp.335-344, 1962.

[COR65] Corbato, F., and Vyssotosky, V., "Introduction and overview of the MULTICS system", Proc. of the AFIPS Fall Joint Computer Conference, pp.185-196, 1965.

[COR01] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., Introduction to algorithms, 2nd ed., MIT Press, 2001.

[DIJ65] Dijkstra, E., "Solution of a problem in concurrent programming control", Communication of the ACM, 8(9):569, 1965.

[FAG96] Faggin, F., Hoff, M., Mazor, S., and Shima, M., "The history of 4004", IEEE Micro, December, 1996, pp.10-20.

[FLY71] Flynn, M., and Rosin, R., "Microprogramming: An introduction and a viewpoint", IEEE Trans. on Computers, July 1971.

[FRA95] Fraser, C. and Hanson, D. A retargetable C compiler: design and implementation, Benjamin/Cummings Pub., 1995.

[GOL47] Goldstein, H., von Neumann, J., and Burks, A., "Report on the mathematical and logical aspects of an electronic computing instrument", Institute of advanced study, 1947.

[HAN01] Hansen, P. (ed.), Classic Operating Systems, Springer-Verlag, 2001.

[HEN84] Hennessy, J., "VLSI Processor Architecture", IEEE Trans. on Computers, December 1984.

[HEN03] Hennessy, J., and Patterson, D., Computer Architecture: a quantitative approach, 3rd ed. Morgan Kaufmann, 2003.

[HOA74] Hoare, C.A.R., Monitors : an operating system structuring concept, Comm. ACM, 17(10):549-557, 1974.

[HOR83] Horowitz, E., Programming languages: a grand tour, Computer Science Press, 1983.

[INT01] Intel Corp. Intel Pentium 4 processor optimization reference manual, Document 248966-04. Aurora, CO, 2001.

[IOW99] Iowa State University, Department of computer science, http:// www.cs.iastate.edu/ jva/ jva-archive.shtml

[JOY00] Joy, B., (Ed), Steele, G., Gosling, J., Bracha, G. Java(TM) Language Specification (2nd Ed), Addison Wesley, 2000.

[KAM90] Kamin, S. Programming Languages: An interpreter-based approach, Addison-Wesley, 1990.

[KAT93] Katz, R., Contemporary Logic Design, Addison-Wesley Pub Co., 1993.

[KER78] Kernighan, B., and Ritchie, D., The C programming language, Prentice-Hall, 1978.

[KID00] Kidder, T., The soul of a new machine, Back bay books, 2000.

[KLI05] Klienberg, J., and Tardos, E., Algorithm Design, Addison Wesley, 2005.

[KOG81] Kogge, P., The architecture of pipelined computers, McGraw-Hill, 1981.

[KOT03] Kotrajaras, V., and Chongstitvatana, P. "Nibbling Java Byte Code for Resource-Critical Devices, Proc. of National Computer Science and Engineering Conference," 2003.

[KOO89] Koopman, J., Stack Computers: the new wave, Ellis Horwood, 1989.

[KOO90] Koopman, P., An Architecture for Combinator Graph Reduction, Academic Press, 1990.

[LAV80] Lavington, S., Early British Computers, Manchester University Press, 1980.

[LEE95] Lee, J., Computer Pioneers, IEEE CS Press, Los Alamitos, California, 1995.

[LEH89] Lehoczky, J.P., Sha, L. and Ding, Y., "The rate monotonic scheduling algorithm – Exact characterization and average case behavior", Proc. IEEE Real-time Systems Symp., pp. 166-171, 1989.

[LEI80] Leinbaugh, D.W., "Guaranteed response time in a hard real-time environment", . IEEE trans. on software engineering, January 1980.

[LEV89] Levy, H. and Eckhouse, R., Computer programming and architecture: the VAX, 2nd ed., Digital press, 1989.

[LIN97] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Addison Wesley, 1997.

[LOU97] Louden, K., Compiler Construction: Principles and Practice, PWS Pub., 1997.

[MAN92] Mange, D., Microprogrammed systems: an introduction to firmware theory, Chapman & Hall, 1992.

[MAN98] Manchester university, computer science department, MARK1, http:// www.computer50.org/ mark1/ firstprog.html

[MAN] The university of Manchester celebrates the birth of the modern computer, http://www.computer50.org/mark1/

[MCA65] McCarthy, J. et al. LISP 1.5 Programmer's Manual, MIT press, 1965.

[MOO70] Moore, C., and Leach, G. FORTH : A language for interactive computing, 1970.

[MOL88] Mollenhoff, C., Atanasoff: Forgotten Father of the Computer, ISU Press, 1988.

[PAD81] Padegs, A., "System/360 and beyond", IBM Journal of research and development, September 1981.

[PAT82] Patterson, D., and Sequin, C., "A VLSI RISC", Computer, Septermber, 1982.

[PAT85] Patterson, D., "Reduced Instruction Set Computers", Communications of the ACM, January, 1985.

[PAT98] Patterson, D., and Hennessy, J., Computer Organization and Design: the hardware/software interface, 2nd ed. Morgan Kaufmann, 1998.

[PEL97] Peleg, A., Wilkie, S, and Weiser, U. "Intel MMX for Multimedia PCs," Communications of the ACM, January 1997.

[PRA01] Prasitjutrakul, S., Analysis and design of algorithms, NECTEC, 2001. (in Thai).

[SCH97] Schaller, R., "Moore's Law: Past, Present and Future." IEEE Spectrum, June, 1997.

[SEB04] Sebesta, R. Concepts of programming languages, 6th ed. Pearson/Addison-Wesley, 2004.

[SHA88] Sha, L., An overview of real-time scheduling algorithms, Software Engineering Institute, Carnegie Mellon University, 1988.

[SIE82] Siewiorek, D., Bell, C., and Newell, A. Computer structures: Principles and examples. McGraw-Hill, 1982.

[SIL03] Silberschatz, A., Galvin, P., Gagne, G., Operating System Concepts, 6th ed. John Wiley, 2003.

[STA88] Stallings, W., "Reduced instruction set computer architecture", Proc. of the IEEE, vol. 76, no. 1, January 1988, pp. 38-55.

[STA00] Stallings, W., Operating Systems, 4th ed. Prentice Hall, 2000.

[STE88] Steenkiste, P., Hennessy, J., "Lisp on a reduced-instruction-set computer: characterization and optimization", Computer, vol.21, no. 7, July 1988, pp.34-45.

[STN80] Stern, N. “Who invented the first electronic digital computer?” Annals of the History of Computing, 2:4 (October), 375-376.

[STO93] Stone, H., High performance Computer architecture, McGraw-Hill, 1993.

[STR88] Sterling, L., “Constructing Meta-interpreters for Logic Programs”, in Advanced School on Foundation of Logic Programming, Italy, 1988 .

[TAN01] Tanenbaum, A., Modern Operating Systems, Prentice Hall, 2001.

[THO70] Thornton, J., Design of a computer: the Control Data 6600, Scott, Foresman and Company, 1970.

[TUR37] Turing, A., “On Computable Numbers, with an application to the Entscheidungsproblem”, Proc. Lond. Math. Soc. (2) 42 pp 230-265 (1936-7); correction ibid. 43, pp 544-546 (1937).

[VEN98] Venners, B. Inside the Java Virtual Machine, McGraw Hill, 1998.

[WAR83] Warren, D., “An abstract Prolog instruction set, Technical report 309, SRI, 1983.

[WEX78] Wexelblat, R., History of programming languages, ACM Press, 1978.

[WIL53] Wilkes, M., and Stringer, J., “Microprogramming and the design of the control circuits in an electronic digital computer”, Proc. of the Cambridge philosophical society, April 1953. Reprinted in [SIE82].

[WIL85] Wilkes, M., Memoirs of a computer pioneer, MIT Press, 1985.

[WIR71] Wirth, N., “The programming language Pascal”, Acta Informatica, 1(1):35-63, 1971.

[WIR81] Wirth, N. “Pascal-S: a subset and its implementation”, in Pascal – The language and its implementation, Barron, D. (ed.), pp. 199-260, Wiley, 1981.

Publications related to this project

[BUR04a] Burutarchanai, A., and Chongstitvatana, P., "Design of a two-phased clocked control unit for performance enhancement of a stack processor", National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.

[BUR04b] Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., "A fast instruction fetch unit for an embedded stack processor", Proc. of Int. Conf. on Information and Communication Technologies (ICT 2004), 18-19 November, 2004. Thailand.

[BUR04c] Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.

[CHO98] Chongstitvatana, P., "A multitasking environment for real-time control", The Engineering Research Fund, Faculty of Engineering, Chulalongkorn University, research project number 132-MRD-2537, 1998. http://www.cp.eng.chula.ac.th/faculty/pjw/r1/

[CHO97] Chongstitvatana, P., "Post processing optimization of byte-code instructions by extension of its virtual machine", Conf. of Electrical Engineering, Bangkok, 1997.

[CHO02] Chongstitvatana, P., and Kotrajaras, V., "Instruction compression by nibble coding: war on the old front", IEEE Thailand section: Silver Jubilee Symposium, 15 Nov 2002.

[CHO03] Chongstitvatana, P., "The Art of Instruction Set Design", Electrical Engineering Conference, Thailand, 2003.

[CHO05a] Chongstitvatana, P., "A compact code 16-bit processor for embedded applications", Joint conf. of computer science and software engineering, Nov 2005, Thailand.

[CHO05b] Chongstitvatana, P., "Self-Generating Systems: How a $10{,}000{,}000_2$-line Compiler Assembles Itself," Proc. of National Computer Science and Engineering Conference, Bangkok, 2005.

[CHO06] Chongstitvatana, P., "Stack frame caching", Proc. of National Computer Science and Engineering Conference, Thailand, 2006.

[KOT03] Kotrajaras, V., and Chongstitvatana, P., “Nibbling Java Byte Code for Resource-Critical Devices, Proc. of National Computer Science and Engineering Conference,” 2003.

[NAN04] Nanthanavoot P. and Chongstitvatana, P., “Code-Size Reduction for Embedded Systems using Bytecode Translation Unit”, Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.

[NAN05] Nanthanavoot, P., Burutarchanai, A., and Chongstitvatana, P., “Instruction packing for a 32-bit resource efficient processor,” National Science and Technology Development Agency (NSTDA) Annual Conference, Thailand, 27-30 March 2005 (in Thai).

[PIR03] Piromsopa, P., Bavonparadon, P., and Chongstitvatana, P., “Hardware multiplexing: towards a resource efficient reconfigurable processor”, 3rd Inter. Symposium on Communications and Information Technologies, Thailand, 2003.

[WON99] Wongsiriprasert, C., and Chongstitvatana, P., “Performance comparison between two virtual machine interpreters: stack-based vs. register-based”, Proc. of 3rd Annual National Symposium on Computational Science and Engineering, Bangkok, 1999, pp. 401-406.

[SRI06] Sattayawiboon, C., Sripornprasert, J., Tansutthiwess, S., Tonteerawong, P., and Chongstitvatana, P., “A stack processor with integrated display circuit for a low cost CD-ROM reading device”, ECTI International Conference, May 10-13, Thailand, 2006.

[TCH98] Taechashong, P. and Chongstitvatana, P., “A VLSI design of a load/store unit for a RISC microprocessor”, Proc. of The Second Annual National Symposium on Computational Science and Engineering, Bangkok, March 25-27, 1998, pp. 244-248.