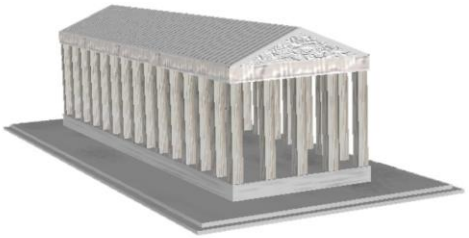
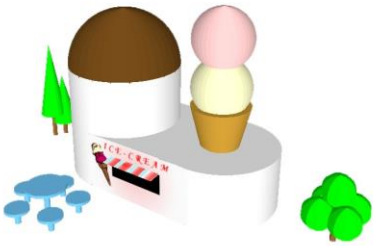
COMPUTER ARCHITECTURE: A SYNTHESIS





Prabhas Chongstitvatana

COMPUTER ARCHITECTURE: A SYNTHESIS

Prabhas Chongstitvatana

CHULALONGKORN UNIVERSITY

To my dearest mother I dedicate this book.

Preface

Computer architecture is an exciting subject. The rapid development of technology increases the speed of a processor by 60 % every year continuously for 20 years. This translates into a factor of more than 1,000 times speed improvement of computer systems from 1980-2000. This is almost beyond any imagination of the creators of the technology. In my life time, I have witnessed the key development of this technology and the important discoveries in this field so often that it almost becomes a part of every life! It is difficult to find the advancement of this scale in other fields. The understanding of this development is important to foresee the future and understand the limitation of the technology.

Teaching computer architecture is extremely rewarding and at the same time can be exhausting as the subject itself evolved at such a rapid rate. I have been teaching students at many levels including bachelor degree, master degree and doctoral degree for a number of years. My experience in teaching the subject is that students learn best by "doing", playing with the design. In the past, performing the experiments with computer design is difficult. However, as our knowledge in computer design grows, we are able to understand and model its behaviour more accurately and more easily. We are able to develop simple tools that can simulate approximate behaviour of various parts of computer systems without too much difficulty. The tools can change the way we learn about computer architecture. These tools are simulators at different levels. At the program-level, a program profiler enable us to see how different parts of the program spend their times. At the instruction-level, the simulator exposes the execution of machine instructions and the frequency of their use. The lower-level down to the level of machine organisation can illustrate the inner working of a processor and let us understand how each component cooperates to achieve performance under different constraints. All these tools enable us to learn with great clarity.

This book explains various parts of modern computer systems. Beginning with the basic organisation and extend it to incorporate performance enhancement features such as pipeline, multiple functional units and vector units. This book compiles many ideas of performance enhancement of modern processors, for example, speculative execution and the revival of the very long instruction word (VLIW) processors. The explanation at each step is accompanied with design

examples and the executable model for students to experiment with. Students can try alternative designs and vary constraints to learn about the effect of different architectural features. The details of these tools are discussed at the end of this preface.

Already there are many excellent books on computer architecture. I will mention three books. The first one is the "Computer structure: reading and example" by Bell and Newell. The second one is "Computer architecture: a quantitative approach" by Hennessy and Patterson. The third one is "Computer architecture" by Blaauw and Brooks. I learn a lot from these books and always enjoy reading them. I strongly recommend students of computer architecture to read them too. I believe that there are many aspects of the subject of computer architecture. One aspect is the lesson learned from the past, which can be read from the history of computers and the development of computer technology. Another aspect is the present day development, the knowledge that can be learned from the current processor design. It is very difficult to write a book that can tracked the ever changing technology (such book will have to be rewritten every 2 years). However, some knowledge that has been distilled from all these materials is valuable and long lasting. It is impossible to include all aspects of the subject in one book. It is the duty of a teacher to select only some topics to be included in his teaching materials. I have chosen to present the view of computer architecture as an evolutionary path towards the ever changing needs of society. I hope students will gain some knowledge reading and experimenting with the design in this book and enjoy it to the extend that they want to learn more by themselves.

The content of this book is divided into three parts. Part 1 is the basic processor design. Part 2 is the performance enhancement architectural features. Part 3 contains the rest of computer systems: memory and magnetic disks, and includes the discussion of future architecture. Totally there are 12 chapters. The chapters of Part 1 are as follows.

Chapter 1 describes the basic concepts of computer architecture and the view that computation is caused by the control of data flow into various functional units. In this chapter, we define performance and its measurement. An interesting history about the origin of computers is briefed at the end of chapter.
Chapter 2 discusses the instruction set design which has an important impact on the performance. The assembly language programming is explained. The end of the chapter includes the discussion of the reduced instruction set computer (RISC) which is a revolutionary idea of instruction set design in 1980.

Chapter 3 explains computer arithmetic. Both integer and floating-point arithmetic are described in details.
Chapter 4 explores the control unit. The elegance of microprogramming is illustrated. The microprogrammed control unit is viewed as a controller made of another small computer. This small computer contains its own program which is an executable code. This program represents the control information and is called "microprogram". The concept of microprogram is very powerful and it is the driving force of the evolution of computer design in 1970.
Chapter 5 integrates all fundamentals in Chapter1-4 to design a hypothetical processor, S1. The detailed design is discussed. Its control unit is implemented in both hardwired and microprogrammed. The instruction-level simulation of S1 and its microprogramming is explained.

Part 2 consists of 4 chapters.
Chapter 6 studies the most fundamental technique for performance enhancement, pipelining. For a concrete design, a case study of instruction pipelining in S1 is discussed.
Chapter 7 discusses many techniques for performance enhancement, for example, superscalar, VLIW, and speculative execution.
Chapter 8 describes supercomputer class of architecture, vector machines. The programming of vector machines and the measurement of their performance are explained.
Chapter 9 describes stack architecture. It was very popular in the past because of its simplicity and suitability for block-structured languages. A case study of one stack processor, R1, is illustrated. Its simulation is studied and the result compared with a register-based architecture.

The rest of the content contains in Part 3.
Chapter 10 studies memory system which is very important and is the most expensive part in modern computer systems. The recent advances in memory technology is included in the end of the chapter.
Chapter 11 explores magnetic disks and its performance. The disk array (RAID) system is discussed. The final chapter,
Chapter 12, looks into the future of computer architecture which one-billion transister device will be possible. Seven proposals for future architecture are examined. These proposals range from evolutionary design to revolutionary design.

An integrated part of this book is the set of tools to explore computer design. These tools enable students to see the detailed working of the design and to try to vary constraints and understand their effect. The source code of simulation is

made available so that students can modify it to try out variation of the design easily. In using this book in a semester-based teaching, I gave out the design assignment around the middle of the course. The problems generally ask students to design a processor that includes a particular architectural feature. Students must work on the simulation of the design and accompany their design with the detailed measurement of the performance. A number of good work are selected to be presented to the class. Near the end of the course, students are assigned to research the additional topics in computer architecture which are not discussed in the class. A list of research papers from the current literature is posted. Students choose their topics from this list and summarise their finding in the written reports. Selected works will be presented to the class by the authors. The detailed example of these assignments can be found in the appendix.

The tools consist of five programs.

1. Learning the assembly language -- Motorola 6800 instruction set is used as a learning tool. The tool includes an assembler, A68, which translates a source program into an executable machine code, and a simulator, SIM68, which enables the 6800 machine code program to be executed. The SIM68 allows students to examine instruction by instruction execution and its effect on registers and flags.
2. Instruction-level simulator of S1 -- Students can write the program for S1 and see it execution. The source code of the simulator is available. Students can understand how S1 microarchitecture works. The simulator can be modified to change the behaviour of the processor such as adding new instructions, changing instruction format etc.
3. Microprogramming tool for S1 -- The simulation of microprogrammed S1 enables students to try out microprogramming. The tool can be modified to run different format of microprogram or change its semantic.
4. Pipeline simulation of S1 -- This simulator shows the mechanism of pipeline and its control (pipeline stall, and interlocking). The tool can be modified to simulate other parallel operations such as scoreboard and Tomasulo.
5. Cache simulation -- The tool simulates three types of cache: fully associative, direct-map and set associative. An address trace is used to compare the performance of different cache configurations and their parameters such as cache size.

These tools are available through the web site of this course at

http://www.cp.eng.chula.ac.th/faculty/pjw/teaching/ca.htm

Also include in the course pages are the additional materials such as information on current processors, the links to other sites in computer architecture. These

pages will be updated from time to time to reflect the current event in my teaching.

Acknowledgements

First and foremost I would like to thank my parents who support me and always encourage me throughout my years. I thank all my teachers who gave me the knowledge and created the foundation that help me to grow and become a teacher myself. I thank Susak Thongthammachart (at Kasetsart University), who always is my mentor. I thank Boonklee Plangsiri who taught me finite automata, Paisal Saguanmoo who gave me my first exposure to computer architecture. I would like to thank all my students who endure my teaching, at the department of electrical engineering, Kasetsart university, at the department of computer science, Mahidol university, at the department of computer engineering, King Mongkut university at Thonburi, and at the department of computer engineering, Chulalongkorn univerity. I thank all students who commented on my course and recommended many improvements which are included in this book. I thank all teaching assistances who bear?? with my tight schedule of the course. I am grateful for all the effort that students put into the study in all my classes. Their creativity and their attention become the source of my enjoyment and my happiness in all my teaching. I thank Somchai Prasitjutrakul and Jaruloj Chongstitvatana who read the draft of this book and recommended a number of improvement of the presentation. I thank the department of computer engineering, Chulalongkorn university that provides me with the best atmosphere to write and supports me to finish this book. The cover of this book is designed by Yodthong Rodkaew, the architectures are the work of students in the class 2110495 Real-time interactive programming at the department of computer engineering in the second semester, year 2000. They are: Vasini Apiwattanakarn, Chaoyut Bovorvongvai and Nic Thippongprapas. Finally, I thank my wife and daughter, Som, who support me and tolerate my neglect during the work on this book, and my little daughter, Nam-tarn, who is the final catalyst of my project.

P. C.
February, 2001

Contents

Part I Basic Processor Design

Chapter 8 Vector machines.....147

Chapter 9 Stack machines.....165

Part III Memory, Disk and Future Architecture

Appendix A Projects in computer architecture 241

Appendix B How to do a paper.....249

Chapter 1

Introduction

This chapter lays the basic knowledge of the subject. We give an overview and a perspective of computer architecture. We look at architecture from the point of view of a computer designer. We describe the components and the organisation of computer systems in many levels of abstraction. We study the "description of an architecture" which specifies a computer system unambiguously. We answer the question "How can a computation is achieved by an architecture". The relationship between architecture and computer languages is important and several issues have been addressed. We discuss the most important aspect of modern computer design, the performance issue. Finally, a brief history of computer, which is a very fascinating subject, is discussed.

Architecture concerns function

Architecture concerns "function" of the system. Function determines *what* the system is capable of. The *how* question is answered by an "implementation" of the system which depends on technology. A computer system consists of many parts. A part can be divided into subparts and forms a hierarchy. Computer architecture 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 consisted of an arithmetic logic unit (ALU), a datapath, and a control unit. The memory system consists of a hierarchical structure: cache memory (high speed memory), main memory, and virtual memory. The input/output system consists of various peripherals such as a visual display unit (VDU), a keyboard, input devices, an interface to the network, various kind of secondary storage, floppy disk, hard disk and so on. The interconnections link every parts together, they are the internal bus, the external bus, I/O channels, and ports.

A computer designer must explore many possibilities of choosing and integrating various "components" of a system to satisfy a set of constraints stated in a requirement. A computer designer must must make decision how to select and integrate various components such as processor, memory, input/output into a computer system. Computer architecture is driven by the advancement of technology. A designer must evaluate an architecture with its technology. The study of computer architecture is the study of method for selection and evaluation. Computer architecture is different from its implementation. Various parts of a computer can be either hardware of software. Hardware and software are interchangeable depending on technology.

One important aspect of computer design is the instruction set design (or instruction set architecture, ISA). A classical view of computer architecture is that the architecture is what the assembly language programmer see, i.e. computer architecture is the instruction set.

A broader view of computer architecture includes the organization of a computer system. An implementation can be regarded as two aspects, one is the organization, and the other is the technology. The organization describes the functional units inside a processor and their relationship. The technology aspect determines how it is possible to build a processor.

Computer system structure

A computer system can be seen at many levels of description, from the applications to the lowest level of electronic circuits. A computer system 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 lowest level is consisted of the actual hardware devices: a central processing unit, a memory, input/output devices and interconnections. These hardware devices can be described at many levels: 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.

Applications
Operating system
Instruction set
Functional units
Finite state machine
Logic gates
Electronics

Figure 1.1 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 an abstraction layer that separates a user program from the underlying system dependent hardware and peripherals.

The level of traditional computer architecture begins at *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 a computer design. The concept of the family of computers was promoted by IBM around 1970. They proposed the concept of one instruction set with different level of performance (with the price differentiation) 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 the 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 (year 2000) 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 circuit 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 tolerant 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.

Computer hardware

The technology of computer is based on the advances of microelectronics. To understand technology one needs to know the fundamental concept of what a computer is made of. The physical components of a computer in the present are based on electronic circuits. The circuits can be regarded as logic gates. The basic elements are logic gates. The complete set of gates is composed of: AND, OR, NOT gates. This is not the only basis, there are several others, for example NAND gate (NOR gate) alone constitutes a complete set because it can perform 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 circuit has no memory, output is the function of input only. To create memory, the output is fed back to input. The resulting circuit is called sequential logic.

A *sequential* logic circuit 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. Sequential circuit has "states". The output depends on both inputs and states. Sequential logic requires clocking. There are two types, synchronous and asynchronous. A s*ynchronous* logic circuit has a common clock. It is a rule of thumb for design engineers to use synchronous logic because it is much simpler to design and to debug. One drawback of synchronous circuits 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 circuit has no central clock, hence it can be much faster than synchronous circuits. It is also advantageous when the clock rate is very high and clock skew becomes a problem. However, asynchronous design is difficult. 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 and 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. 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 (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 computers. These building blocks plus the control unit constitute the basic structure of computer. Basic units to perform arithmetic functions are: adder, multiplier, shifter etc. They reside in an ALU. 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 circuit. 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.

Description of an architecture

Charts and block diagrams can be used to illustrate a "structure" of a computer system with blocks denote functional units (or components) and lines as connections or relations between those units. There are many notational systems such as PMS-ISP [BEL71] instruction set processor, RTL (Register Transfer Language), even APL like notation for behavioural description by Blaauw and Brooks [BLA97]. We shall discuss the PMS-ISP notation because it is well-known and is used in many historical work in computer architecture. We will give our version of the descriptive system that is composed of structural chart, instruction set and behavioural description.

6

PMS and ISP descriptive systems

This descriptive system consists of two levels of description. The PMS describes the total system. The ISP provides the description at the level of the instruction set.

PMS level of description

Digital computer can be viewed as discrete state systems that have three characteristics:

- The state is realised by *information*, stored in memories.
- A computer system consists of a number of subsystems linked together by flows of information. These *components* are called memory, processor etc.
- Each component is associated with *operations* for changing its own state or the state of neighbouring components.

There are seven basic component types in PMS: Memory (M), Link (L), Control (K), Switch (S), Transducer (T), Data-operation (D), and Processor (P). An operation is a transformation of bits from one specific memory to another, M to M'.

Computer model in PMS

We will give an example how to use PMS notation to describe a computer system. A configuration of a computer (C) is

C := Mp – Pc – T – X

where Pc indicates a central processor and Mp a primary memory. T is a transducer connected to the external environment, represented by X (input/output devices such as disks, a console and so on).

The description can be refined to reflect the fact that Pc can be decomposed into a control K and an arithmetic unit or data-operation D and alternatively the control can be connected to a secondary memory.

```
Mp – K – T | Ms – X
     |
     D
```

where "|" expresses alternatives (T "or" Ms, the secondary memory)

ISP level of description

The behaviour of a processor is determined by sequence of its operations. This sequence of operations is determined by a set of bits in Mp, called the program, and a set of interpretation rules that specify how particular bit configurations evoke the operations. ISP (Instruction set processor) provides a scheme to specify any set of operations (instructions) and any rules of interpretation.

An instruction expression has the form:

condition --> action-sequence

The --> is the control action K of evoking an operation. Each action has the form:

memory-expression <-- data-expression

The <-- is the transmit operation of a link (correspond to the assign operation). The left-hand side describes the memory location, the right hand side describes the information pattern.

An ISP example of the DEC PDP8

We give an example how to use ISP notation to describe a part of a classic computer DEC PDP8. The PDP8 is a very simple machine with a small number of instructions. It is the machine that started the market of "minicomputer". This example illustrates the description of a processor state, the primary memory state, the instruction format, the meaning of one instruction and how the machine execute an instruction. Comments are in *italics*.

Processor state Pc

AC<0:11> *the accumulator*

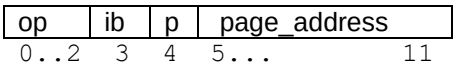
AC is a 12-bit register. AC is a register in the processor.

Primary memory state Mp

Mp[0:7777$_8$]<0:11>

A primary memory consists of 2048 words (the size of memory is expressed in base 8, the convention of this machine). Each word is 12 bits.

PDP8 instruction format can be shown in the diagram:



The width of an instruction is 12 bits. It is defined in ISP as follows:

Instruction format

```
op<0:2> := instruction<0:2>
indirect_bit / ib := instruction<3>
page_0_bit / p := instruction<4>
page_address<0:6> := instruction<5:11>
```

The instruction set

```
and (:= op = 0) --> (AC <-- AC ^ M[z] )
```

This describes that the opcode of the instruction "and" is 0 and its action is to AND AC and a memory location z, where z is an effective address.

An instruction is fetched from the memory and then executed. Next, the *next* instruction is fetched and so on (ignoring the interrupts):

Instruction interpreter

```
Run --> (instruction <-- M[PC]; PC <-- PC + 1; next          fetch
         Instruction_execution)                              execute
```

A state diagram represents the behaviour of the instruction-interpretation process. The K controls the state transitions according to the information in the instruction.

Microarchitecture and behavioural description

We are interested mostly in the microarchitecture, which concerns the processor. Throughout this book, we will describe a computer system using the following notations:

- Structural chart: a diagram of processor organization is used to give a high level view of a processor.
- Instruction set: a description similar to ISP.
- Behavioural description: a RTL is used to describe the operation of each instruction step-by-step. It is called "microsteps" in this book.

Structural chart

A structural chart shows the distinct components of a processor and their connections. For example, the CRAY-1 super computer [RUS78] is composed of main memory (up to 4 M 64-bit words), scalar registers S (8 × 64-bit), backed by a 64-element vector register T, address registers A (8 × 24-bit), backed by a 64-element 24-bit vector register B, vector registers V (8 × 64-bit), vector units, floating-point units, scalar units, and address units as shown in Fig 1.2.

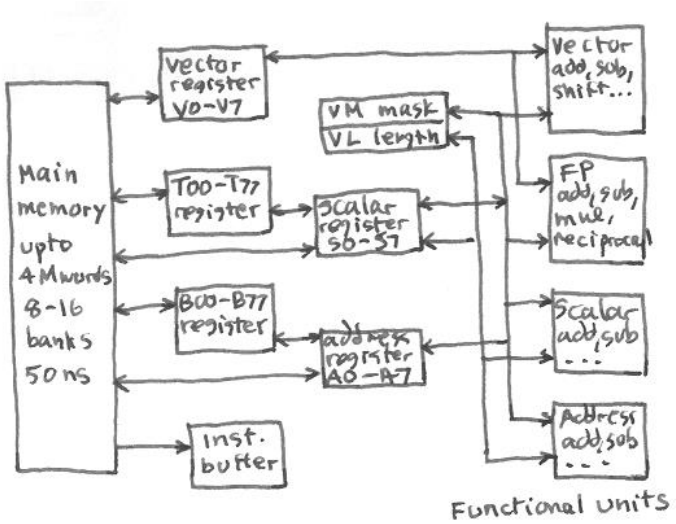
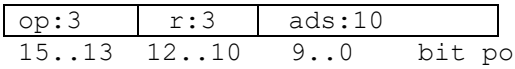


Figure 1.2 Structure chart of CRAY-1

Instruction set

An instruction set is expressed by the instruction formats and the instruction names. For example, the instruction set of S1 (hypothetical processor used in this book) has one format called L-format (for long-format), each field is denoted by the `fieldname:length`.

S1 L-format : `op:3 r:3 ads:10`



Two instructions and their opcodes are: (comments in *italics*)

```
0  ld M, r              M -> r load from memory
1  st r, M              r -> M  store to memory
```

Behavioural description

A register transfer language is used to describe the step-by-step operation of each instruction. The notation of this RTL is as follows:

- Comments start with "`//`" to the end of line
- Data movement from source to destination is denoted by `dest = source`
- The parallel operations of two actions is denoted using ";" such as `e1 ; e2`
- The access to a memory location is denoted by `M[a]`
- The bit field of a register is denoted by register:field such as `IR:a`
- `<name>` denotes the label of sequence of operation of the instruction
- `op( )` denotes the ALU operations: `add, cmp` etc.

Example: S1 behavioural description of the "load" instruction is:

```
<load>
MAR = IR:ADS
MDR = M[MAR]      // memory read
R[IR:R0] = MDR
```

The address field from the instruction (bit `ADS` of Instruction Register, `IR`) is read into Memory Address Register (`MAR`). A memory addressed by `MAR` is read into Memory Data Register (`MDR`). The register indexed by `IR:R0` is written with the value of `MDR`. This sequence of operations takes 3 clocks.

How a processor performs computation

Suppose we want to calculate value of a polynomial function

$$f(x) = ax + bx^2$$

The functional units required to do this computation are multiplier and adder. The desired computation can be performed by directly connect appropriate number of functional units together (Fig 1.3).

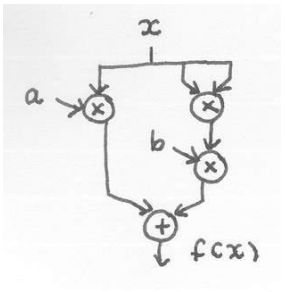


Figure 1.3 a computation graph to evaluate a polynomial

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 sequencing the operations (Fig 1.4)

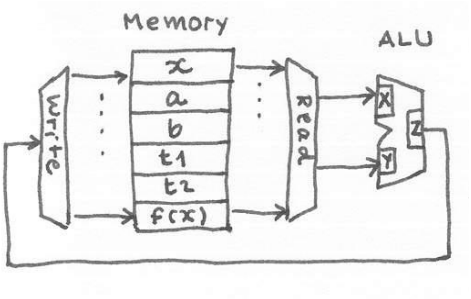


Figure 1.4 a sequential model of computation

The required functional units are memory and a general processing unit. A memory stored all the necessary values: input *x*, constant *a* and *b*, the temporary places to keep intermediate values *t1*, *t2*, and the final result *f(x)*. The memory

can be read and written to. The memory can be read two values at once and feed the data to a general processing unit, so called Arithmetic Logic Unit (ALU). 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*. An ALU performs $Z = f(X,Y)$ where $f = \{ add, sub, mul, increment, \ldots \}$. 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)
compute(mul)
write(t1)
read(x,x)
computer(mul)
write(t2)
read(t2,b)
compute(mul)
write(t2)
read(t1,t2)
compute(add)
write(result)
```

Sequential approach to computation enables functional units to be reused as the computation is performed step-by-step. Intermediate values can be saved in the memory 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 and 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 1.5).

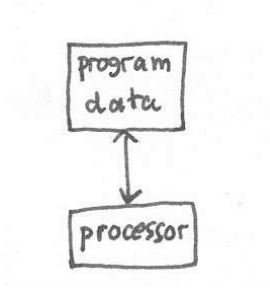


Figure 1.5 Von Neumann architecture

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. 1.6). This configuration increases the memory bandwidth because the processor has separate connections to program and data.

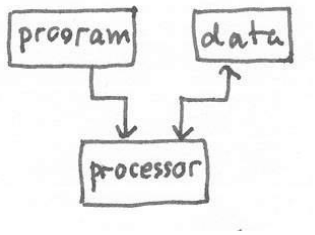


Figure 1.6 Harvard architecture

This organisation is called "Harvard architecture". It is extensively used in high-speed processors for signal processing, which is called Digital Signal Processor (DSP). DSP has many applications. It is used in modems, in sound synthesizer, in graphic generators etc.

Computer languages and architecture

Programming techniques influence the design of computers since the early day of assembly language programming [HOP97]. Most computers today are implemented as sequential machines. They are suitable to be programmed in a class of high level programming language called procedural languages. Examples of procedural languages are C, Pascal, C++ etc. 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 is 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 is not changed by where the function resides. This property is contrasted to procedural programming which compute 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 view computation as composing functions, it maps nicely to the graph model of computation. Many proposals being put forward to build machines which are suitable for this class of programming languages, for example a graph reduction machine [KOO90].

Different programming paradigms lead to different architectures. LISP, the language of artificial intelligence community, requires data tags and dynamic memory reclamation [STE88]. Logic programming paradigm (Prolog programming language and others) requires architecture capable of inferring facts and rules and ability to backtrack efficiently, for example the Edinburgh Prolog virtual machine [PRO]. Japanese proposed and built various types of these machines in the period of their research on the fifth generation computer [FIF]. Presently, object-orientated programming paradigm is becoming the dominated paradigm. The object-oriented programming languages (Java, C++, Smalltalk etc.) require the dynamic allocation and deallocation of objects. They will benefit from machines whose architecture are suitable to implement them.

Performance

This section discusses performance issue. How performance of a computer system is defined and measured. Standard references are 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. The first 25 years the performance improvement came mostly from technology and better computer architectures. Later, the improvement mostly came from the advent of microelectronics. The speed 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 16Kbytes of memory, one floppy disk and no hard disk. The later model offered 5Mbytes hard disk (so called IBM XT). Today (year 2000) a PC is equipped with Pentium 32-bit CPU with 500 MHz clock, 64Mbytes of memory and 10 Gbytes disk. Its performance is around 1000 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 [MOO65], who is a 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 manufacture 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 microarchitecture
- *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 the 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)

The fastest machine of the year 1997 is the ASCI-Red of the department of energy, USA. It is composed of 2048 nodes of Pentium Pro with collective memory of 600 G bytes. Its peak performance is 1.8 Tflops and it has run 630 Gflops on 3400 nodes (running simulation of motion of particles) [KAR98].

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 another machine. The performance is the inverse of the execution time. The following relationships can be derived:

X is n% faster than Y means

execution time Y / execution time X = 1 + n/100

performance = 1/ execution time (or 1/t)

execution time Y / execution time X = 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.

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

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

Therefore the limitation 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 40% of the time. What is the overall speedup?

$$\text{speedup overall} = 1/ ((1 - 0.4) + 0.4/10) = 1.56$$

Please note that Amdalh's law applies only with the problem of fixed size. When problem size can be scaled up to use available resources, Amdalh's law doesn't applied. This is why the massively parallel machine is still possible.

Example: Comparing CPU A and CPU B, A with "compare then branch" instruction sequence, B has special combined "compare&branch". A has 25%

faster clock. For CPU A, 20% of instruction is "branch" and hench another 20% is the accompanied "compare". "Branch" takes 2 clocks and all other instructions take 1 clock. "compare&branch" takes 2 clocks. Both CPUs run the same program. Which is faster?

CPU time A = num. of instruction A × CPI A × cycletime A
= n.o.i A × ((.20 × 2) + (.8 × 1)) × cycletime A
= 1.2 × n.o.i × cycletime A

"compare" are not executed in CPU B so 20% of 80% = 25% of instructions are now branching taking 2 clocks and the rest 75% take 1 clock.

CPI B = .25 × 2 + .75 × 1 = 1.25
CPU time B = .8 × n.o.i A × 1.25 × 1.25 cycletime A
= 1.25 n.o.i A × cycletime A

Therefore A, with shorter cycle time, is faster than B, which executes fewer instructions.

Now if the designer reworks CPU B and reduces the clock cycle time so that now A cycle time is only 10% faster. Which CPU is faster now?

CPU time B = .8 × n.o.i A × 1.25 × 1.1 cycletime A
= 1.1 n.o.i A × cycletime A

So now CPU B is faster.

Calculation of CPI

In order to understand the effect of different instruction set, understanding of assembly language is required. An example of assembly language programming is illustrated as follows.

Suppose a hypothetical machine has the typical instruction set composed of {load, store, compare, increment, jump condition}. It has an index register (x) and a set of general purpose register (r0..r7).

Find max of array[i], i=1..N

```
max = array[1]
```

```
i = 2
while i <= N
  if max < array[i] then max = array[i]
  i = i + 1
end
```

let the array[i] be accessed by `load r0,array,x`

```
max     equ ...
array   equ ...

        load x, #1
        load r0, array, x
        store r0, max          ; max = array[1]
        load x, #2             ; x keeps i
loop    cmp x, #N
        jump GT exit           ; i <= N
        load r0, max
        load r1, array, x
        cmp r0, r1             ; max < array[i]
        jump GE skip
        store r1, max
skip    inc x
        jump loop
exit    END
```

We count the number of instruction being executed to calculate CPI.
let N=3, array[ ] = 1,2,3

	frequency	**clock**
load	3+4	2
store	1+2	2
cmp	4	1
jump	4	2
inc	3	1

Total 21 instructions, 35 clocks. CPI = 35/21 = 1.67

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".

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.

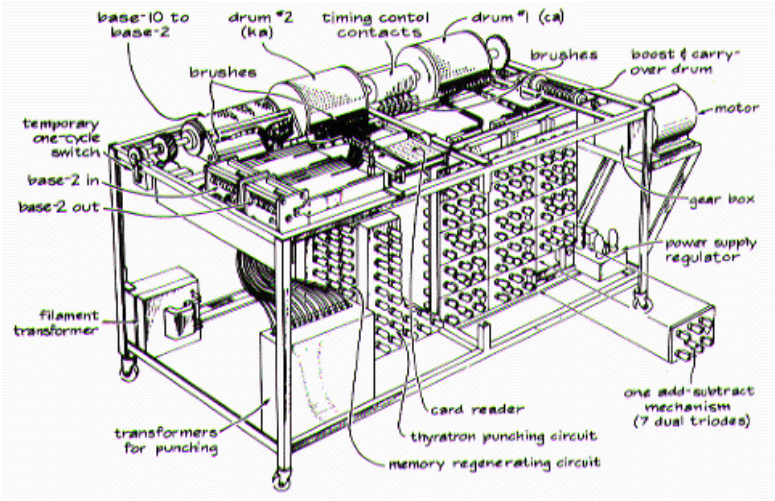


Figure 1.7 the ABC diagram [IOW99]

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's 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 read and write 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.

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).

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).

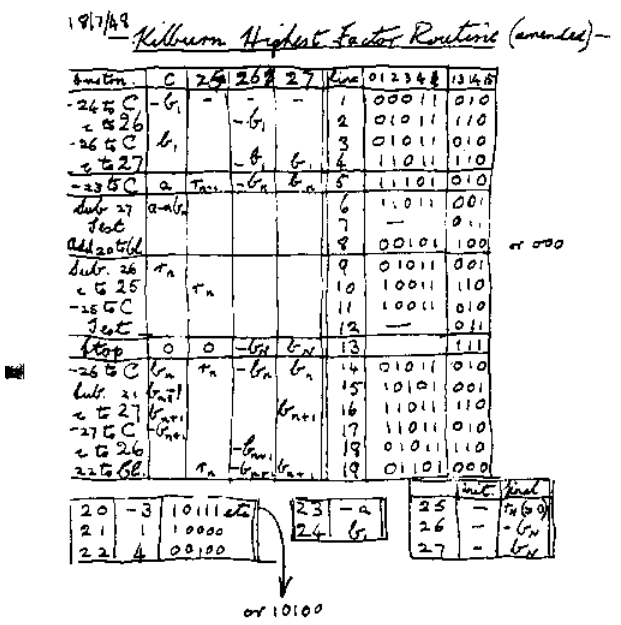


Figure 1.8 the first program [MAN98]

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.

Time line of the history of computer

Mechanical era

1642 Blaise Pascal invented a machine that can add/subtract numbers
1666 Samuel Morland invented a machine that can multiply by repeated addition.
1671 Gottfried Leibniz, an adding and multiplying machine
1820 Charles Babbage, Difference engine
1830 Charles Babbage, Analytical engine (Father of modern computer)

Electro-mechanical era (relays)

1880 Herman Hollerith, punch card machine
1924 Thomas J. Watson founded IBM
1930 Beginning of computer age
Howard H. Aiken, Harvard university (MARK I)
John V. Anatasoff, Iowa State univ.
George R. Stibitz, Bell telephone lab.
Konrad Zuse, Technische Hochschule in Berlin, ZUSE 1
1943 Flowers, Colossus
1946 Eckert & Mauchly, ENIAC

Electronics era

1948 Manchester SSEM
1949 Manchester Mark I
1950 John Von Neumann, EDVAC
1950 Alan Turing, ACE
1951 Forrester (MIT), Whirlwind
1952 Goldstine and Neumann, IAS

Computer industry era

1951 Remington Rand, UNIVAC
1952 IBM 701

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, G. and Newell, A. "Computer structures: readings and examples", McGraw-Hill, 1971.

[BLA97] Blaauw, G., and Brooks, F., Computer Architecture: Concepts and Evolution Addison-Wesley Pub Co., 1997.

[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.

[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.

[HOP97] Hopper, G., Mauchly, J., "Influence of programming techniques on the design of computers", Proceedings of the IEEE Volume 85, no. 3, March 1997, pp. 470-474. Reprint from Proc. of the IRE, vol. 40, no. 10, October 1953, pp. 1250-1254.

[ILI82] Iliffe, J., Advanced computer design, Prentice-Hall, London, 1982.

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

[KAR98] Karp, H., Lusk, E., and Bailey, D., "1997 Gordon Bell prize winners", Computer, January, 1998, pp.86-92.

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

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

[KUC78] Kuck, D., "The structure of computers and computations", Vol 1, John Wiley & Sons, 1978.

[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

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

[MOO65] Moore, G., "Cramming mor components onto integrated circuits", Electronics, April 1965, pp. 483-485.

[RUS78] Russell, R., "The CRAY-1 computer system", CACM 21(1), 63-72. January 1978.

[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.

[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).

[FIF] Japanese Fifth generation computer

[PRO] VM of Edinburgh Prolog

Chapter 2

Instruction Set Architecture

The instruction set design is an important part of computer design. An instruction set is the visible part of a processor where programmers see available resources of the processor such as functional units, registers, flags and the operations that can manipulate those resources.

An instruction set abstracts away the technology dependent part of a processor. For example, the frequency of the master clock, the details of implementation such as the number of pipeline stage and the size of cache memory. An instruction set also defines the architecture of a processor, that is, an ISA defines the function of a processor.

In this chapter we discuss the instruction set design issues. An introduction to assembly language is illustrated using the Motorola 6800. A study of the IBM System360 instruction set is elaborated to illustrate one of the most long-lived ISA. The S/360 ISA defines a family of computers and has a unique position in the computer history. Another approach to the ISA design, the stack-based ISA is discussed. Finally, one of the revolutionalised idea in ISA design of the last decade, the reduced instruction set computer (RISC), is explored.

Design issues

The designer of an instruction set must consider the following issues:

- types of operations
- types of data
- instruction formats
- the number of registers
- the number of addressing modes

Types of operations

An instruction set consists of several types of operations. Most of these types must be present for a general-purpose processor.

1. Arithmetic operations such as add, subtract, increment.
2. Logical operations, such as compare, which return Boolean values or affect flags.
3. Data transfer such as load from memory, store to memory, moving data between registers or input/output.
4. Control transfer such as jump, conditional branch, subroutine call and return. They affect the flow of program execution.
5. Other operations such as disable/enable interrupts and interface to the operating system.

Types of data

The sequence of bit in the memory represents many types of data: addresses, numbers, characters, logical values {True, False}. These data types are interpreted by the instructions. Each instruction requires the correct type of data to produce a meaningful result. The choice of data type in each ISA is heavily influenced by the type of workload, such as binary-packed decimal (BCD) for business applications and floating-point for scientific computing. The difference in design reflects the difference in the intended use.

Example
The Intel Pentium processor has the following data types: byte, word, double word, quadword, integer, unsigned integer, BCD, packed BCD, near pointer, bit field, byte string, floating-point.

The IBM PowerPC processor has the following data types: byte, halfword, word, doubleword, unsigned byte, unsigned halfword, signed halfword, unsigned word, signed word, unsigned doubleword, byte string, single float, double float (IEEE 754).

Endianness (byte ordering, bit ordering)

As the memory is arranged in linear order, the order of bit and byte of data must be specified to have a consistent interpretation. There are two schools of thought: big-endian and little-endian. The big-endian school lays the data in memory

from the most significant to the least significant "digits" and *vice versa* for the little-endian. Neither of which has absolute advantage over the other. In the past, the issue of endianness causes the problem of compatibility when data must be transferred between two machines with different endianness. Presently, the implementation of processors has both endianness built-in which allows software to switch the mode, hence reduces the problem of data translation. The ordering is considered at two levels: bit ordering and byte ordering.

Bit ordering: The ordering refers to whether the least significant bit is the left most or right most bit. This is important when a data is shifted out serially as in the serial communication applications. However, this is not the problem of the architecture as most processor has the instruction to shift both left most bit and right most bit out.

Byte ordering: Suppose a 32-bit value is 12345678 (hex), for a big-endian machine this is represented as 12,34,56,78 (ordering from low address to high address in memory). For a little-endian machine this is represented as 78,56,34,12.

The different processors adopted different endianness, the examples are as follows. The machines with little-endian are Intel 80x86, Pentium, VAX. The machines with big-endian are IBM 370, Motorola 680x0, and most RISC machines. Some machines are bi-endian, the endianness can be set in the processor status bit, they are PowerPC, MIPS.

Example To illustrate the difference between two endianness, consider how the following C structure is mapped in memory.

```
struct {
  int a;      //0x1112_1314                 word
  int pad;
  double b;   //0x2122_2324_2526_2728       doubleword
  char* c;    //0x3132_3334                 word
  char d[7];  //'A','B','C','D','E','F','G' byte array
  short e;    //0x5152                      halfword
  int f;      //0x6162_6364                 word
} s;
```

Big-endian address mapping (byte address)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
11	12	13	14					21	22	23	24	25	26	27	28
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
31	32	33	34	A	B	C	D	E	F	G		51	52		
20	21	22	23												
61	62	63	64												

Little-endian address mapping (byte address)

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
14	13	12	11					28	27	26	25	24	23	22	21
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
34	33	32	31	A	B	C	D	E	D	G		52	51		
20	21	22	23												
64	63	62	61												

Figure 2.1 example of C data structure and its endian maps [IBM94]

Instruction formats

An instruction operates on its "operands". The number of operands varies for each instruction, however many instructions have the same number of operands. The number of operands determines the "format" of an instruction. The instruction format can be classified into 3–, 2–, 1–, and 0–operand instruction.

A 3–operand instruction has the form "op A B C", means A = B op C
A 2–operand instruction has the form "op A B", means A = A op B
A 1–operand instruction has the form "op A", means it operates on A
A 0–operand instruction has the form "op", means it has no operand or the operand is implicit in the stack .

The type of operands can be memory, register or constant values, which will affect:

1. the length of instructions – number of bits required to encode the instruction,
2. the speed of operation – the access time of memory and register are different hence the speed is different for reading and writing operands in memory or register, and

3. the number of instructions required to perform a task – the larger number of operands in an instruction results in fewer instructions to perform a task.

The size of encoding is different between memory and register operand. The number of register in a machine is much smaller than the addressable memory space hence the encoding of register operand is smaller than that of memory locations. The combination of the type of operand gives rise to the difference in category of architecture.

Comparing the register-register format and the memory-memory format. Assume operational code has 8 bits, operand address has 16 bits and each operand size 32 bits. Let I be the size of executed instructions, D be the size of executed data, M be the total memory traffic (in bits). The table below shows the size of instruction for each type of sequence of operations as (I, D, M).

Table 2.1 Comparing register-register and memory-memory instruction formats (I,D,M) – I the size of instruction, D the size of data, M total memory traffic in bits

operations	**register-register**	**memory-memory**
A = B + C	ld rB B ld rC C add rA rB rC st rA A (104 96 200)	add B C A (56 96 152)
A=B+C; B=A+C; D=D-B	add rA rB rC add rB rA rC sub rD rD rB (60 0 60)	add B C A add A C B sub B D D (168 288 456)

The processor design is strongly tied to the instruction set design. There were many diverse computer designs and hence many different instruction set designs in the past. However, as the technology progress, the analysis of the workload – the actual running programs – which affect the instruction set selection leads to the convergence of instruction set architecture. The most common type of instruction set architecture today belong to three classes:

- Load-Store architecture
- Register-Memory architecture
- Register-plus-Memory architecture

Load-Store architecture has 3-address format and mostly 32-bit instruction size. This is the most popular among the current microprocessor design including: HP PA-RISC, IBM RS/6000, SUN Sparc, MIPS R4000, DEC Alpha etc. All data to/from memory must load/store through a register first. The execution (operation) takes operands from registers and the result stored back to a register. This instruction format simplifies the decoding and implementation. Because most operations are performed on registers, they are fast. However, as registers are used extensively the allocation of registers becomes important. Determining which variables to be resided in registers affects the performance of this class of machines and register allocation is done by compilers.

Register-Memory architecture has 2-address format and has 16/32/64 bit instruction size. An instruction can operate both on registers and with one of the operand in the memory. This is the "classical" ISA and is used by one of the longest-lived ISA of today IBM S/360 and Intel x86 family of processors.

Register-plus-Memory architecture is the most flexible in the use of operands. Operands can be registers or memory. This architecture has byte-variable instruction size. This flexibility comes with a price, the complexity in implementation. This type of architecture is typified by VAX family of computer in the era that there was the drive to provide the high level language semantic for the instruction set, so called "close the gap" between high level language and machine language. This architecture combines both operands in memory and registers. It allows flexibility in the use of memory to keep variables and does not need to have a large number of registers to achieve high level of performance.

Addressing modes

The addressing mode refers to the way an instruction calculates addresses of operands. The "effective" address can be computed using the value from the register(s) or the value of some field in the instruction itself. To access an array, the index is necessary. The index is usually stored in a register. The indirect address is used to represent "pointer" type and to access a value via a pointer. Many complicated addressing modes have their use when translating a high level language construct into machine instructions. Table 2.2 shows some of the most frequently used addressing found in most processors.

Table 2.2 various addressing modes

addressing modes	instruction format	instruction meaning
register	add r4,r3	r4 = r4 + r3
immediate	add r4,#3	r4 = r4 + 3
based	add r4,100(r1)	r4 = r4 + M[100+r1]
register indirect	add r4,(r1)	r4 = r4 + M[r1]
indexed	add r3,(r1+r2)	r3 = r3 + M[r1+r2]
direct	add r1,(1001)	r1 = r1 + M[1001]
memory indirect	add r1,@(r3)	r1 = r1 + M[M[r3]]
auto-increment	add r1,(r2)+	r1 = r1 + M[r2] ; r2 = r2 + d
auto-decrement	add r1,–(r2)	r2 = r2 – d; r1 = r1 +M[r2]
scaled	add r1,100(r2)[r3]	r1 = r1 + M[100+r2+r3*d]

Assembly language

In this section we will learn an assembly language. The assembly language is "lingua franca" to talk to the underlying hardware. An example of a real microprocessor assembly language is illustrated in relations with the high level language.

Why assembly language is needed

It is becoming less and less necessary for a programmer to program in an assembly language. High-level languages made programs portable and programming more productive. There are however some situation where an assembly language is necessary such as when programming at very near hardware level. A programmer who creates these types of programs: a compiler, a device driver in an OS, an embedded control program etc. needs to use assembly language. An assembly language is the language that allows a programmer to talk about operations on a bare bone hardware. For a computer architect, an assembly language is the "interface" to the hardware functions. During this course, we will talk about the innards of computers, their organization, how each unit works. All these follow from what kind of assembly language a computer has. It is necessary for a computer architect to be able to write and read assembly language well. All working units inside a computer perform according to some sequence of its instruction.

To study computer architecture, we need to understand assembly language. This introduction will concentrate on principles of assembly language programming. The aim is to enable students to read some subset of assembly language and understand their operational semantics. We will use a real CPU, Motorola 6800, as our example. It was designed more than 20 years ago. It has a simple instruction set and is easy to understand. We use real CPU because it shows the complexity of the real device. We choose only a subset of instruction set that is enough to let us program some small programs.

Instruction set of MC6800

The machine model of Motorola 6800 shows the resource in the view of an assembly language programmer. This microprocessor is composed of two 8-bit accumulators `ACCA` and `ACCB`. It has two 16-bit registers, which can perform indexing: `X` and `SP`. The conditional flags reside in the 8-bit condition code register. The address space is 64K bytes (address 16-bit).

In general, instructions can be grouped into 5 categories:

1. Arithmetic: `ADD, SUB, INC, DEC`
2. Logical operation: `CMP`
3. Data transfer: `LDA, STA` (load, store)
4. Flow of control: `BR, JMP` (branch on condition, jump)
5. Others (such as I/O)

These are instructions that manipulate the index register:

`LDX` load index register
`INX` increment index register
`CPX` compare index registers

Addressing modes are:

- Direct mode, sometimes called "Absolute". The operand is the effective address. `LDA A $100` ($ signify hex)
- Immediate mode, the operand is some constant value to be used. `LDA A #3`
- Indexed mode, the operand is added to the index register to get an effective address. `LDA A $200, X` effective ads = $200 + x
- Relative mode, it is used in jump instructions to get effective address relative to the current `PC`.
- Register mode, the named register is the operand. `TAB, TBA`

Programmer model of 6800

A 8 bits
B 8 bits
X 16 bits index register
SP 16 bits stack pointer
CC 8 bits H,I,N,Z,V,C

Memory model of 6800

64K : 00-FF for short, 0000-FFFF long

Example: P = M + N
let P= $100, M =$101, N= $102

```
ldaa $101
adda $102
staa $100
```

Example: add 1 to 10
In a high level language

```
i = 1; sum = i
while i <= 10
  sum = sum + i
  i = i+ 1
```

in assembly language

let sum =$100, i =101

```
      ldaa #1
      staa $101
      staa $100
loop: ldaa $101
      cmpa #10
      bgt exit    ; while i <= 10
      ldaa $100
      adaa $101
      staa $100   ; sum = sum + i
      inc $101    ; i = i + 1
      jmp loop
exit: ...
```

Example: Find maximum in an array AR[i] i = 1..10
learn how to use index

```
        .org h'100
        ldx i
        ldaa ar,x
        staa max      ; max = ar[0]
loop:   ldaa i+1      ; use 8 bit of i
        cmpa #2
        bgt exit      ; while i <= N
        ldaa ar,x
        cmpa max      ; if max < ar[i]
        ble skip
        staa max      ; then max = ar[i]
skip:   inx           ; i = i+1
        stx i
        jmp loop
exit:
        .org h'10
max:    .db 0         ; max
i:      .dw 0         ; index must be 16 bit
ar:     .db 4,5,6     ; array
        .end
```

Assembler a68

directive `.ORG, .END, .DB` define byte, `.DW` define word
symbolic name `NAME:`
literal `H'100` (hex), `100, #2, #'A'`

In an assembler program, the assembly language directive helps to improve the readability of the assembly language program by providing the use of symbolic names. The directives are special instructions. They are pseudo instructions which do not translated into any actual machine instruction. Mostly they provide the name and the constant value stored in the memory. `ORG` set `PC`, `EQU` define symbol, `DB, DW` reserve storage.

To simplify register allocation, variables are kept in memory (using `DB, DW` or `EQU`). Although sometimes it is laborious to move variables between registers and memory, it is straightforward and easy to understand. Symbolic names can be used to make a program easier to read. From the last example:

```
        .org 0
max     equ  $100
AR      equ  $102
        ldx #1
        ldaa AR,x
        staa max        ; max = AR[1]
        ldx #2          ; i = 2
   ...
```

Tools

An assembler can translate a source file into machine code file (in some file format, such as Motorola S-format). This machine code file can be loaded into a simulator and executed. A simulator allows students to execute and monitor the effect step by step. It shows the value of all registers and can display memory values. The a68 assembler and the simulator, sim68 are available for download from the web page of this book.

IBM System/360 ISA

IBM System/360 [AMD64] is one of the longest-lived instruction set to date, the architecture was introduced in 1964. The goal of this family of computer is to have compatible instruction set but have a performance range of 50. The task of designers is a difficult one. It is aimed to perform both scientific and data processing applications. The scientific applications are dominated by floating point operations. The data processing applications involve movement of long strings. Its long live brought light to a now classical problem in instruction set design: the shortage of addressing space. As applications grow the requirement for address space increase very quickly. A design that has the address space adequate at the time of its introduction quickly find itself lacking address space in just a few years later. To quote from IBM [BEL76]

> "There is only one mistake . . . that is difficult to recover from – not providing enough address bits . . . "

Programmer's model

It is byte addressable, the smallest addressable unit is byte. Addresses are "real" referring to physical location in the main memory. Its successor System/370 [CAS78] introduced a major advanced concept, "virtual" address, where address

does not refer directly to a physical location in the main memory but is mapped to a physical location by a dynamic addressing translation mechanism.

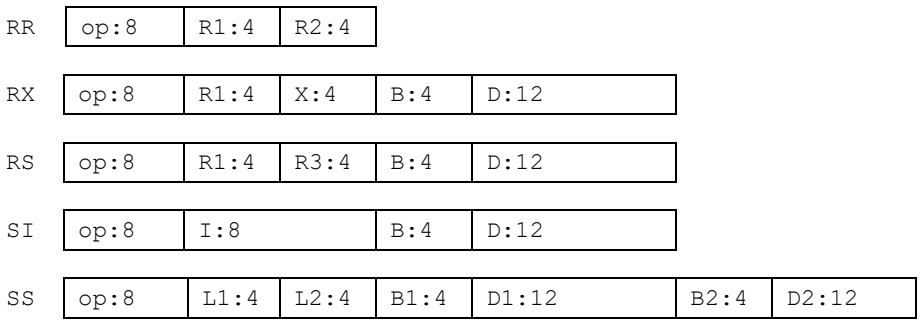
S360 has 16 32-bit registers, R0 to R15. R2 to R12 are general purpose. R0, R1, R13, R14, R15 are special purpose and are used in subroutine linkages (Table 2.3). For floating point number operations the registers are paired into four floating point registers, each 64-bit, numbered : 0, 2, 4, 6.

Table 2.3 S360 special purpose registers

register	caller	callee
R0	return value from the subroutine	return value
R1	send parameters to subroutine	receive parameters
R13	register save area	save and restore registers
R14	return address	return value
R15	the address of subroutine	--

Addressing mode

It has five addressing modes: register-register (RR), register-index (RX), register-storage (RS), storage-index (SI) and storage-storage (SS). The instruction format for each mode is (`field:length` in bit) :



RR register to register R[R1] = R[R1] op R[R2]
RX register to indexed storage R[R1] = R[R1] op M[R[X] + R[B] + D]
RS register to storage R[R1] = M[R[B] + D] op R[R3]
SI storage to immediate M[R[B] + D] op I
SS storage to storage M[R[B1] + D1]:L1 op M[R[B2] + D2]:L2 where L1, L2 are length of operands

Types of data

It is byte-addressable. A full word is 32-bit, a double word is 64-bit. The natural size is 32-bit. For arithmetic data, it has decimal, pack decimal, floating point numbers with single precision 32-bit and double precision 64-bit. It has strings and characters, EBCDIC (extended binary coded decimal interchange code),

Types of operations

The S/360 has been built to accommodate many types of basic functions, with decimal data, binary data and floating-point data and instructions for arithmetic operations for each type of data. The instruction format for decimal addition is not the same as that for binary addition, because the decimal addition does not use registers. Floating-point arithmetic uses its own set of registers, and has special environments in regard to numbering registers. The S/360 has the following classification of its instructions.

- Arithmetic instructions
- Conversion instructions
- Data movement instructions
- Logical instructions
- Branch instructions
- Miscellaneous instructions

load/store

`L` load
`LP` load positive
`LN` load negative
`LC` load complement
`LA` load address
`ST` store

branch

`B` branch
`BC` branch on condition, on condition code (CC bits) using the following mnemonics :
`BZ` branch on zero
`BP` branch on positive
`BM` branch on minus

BNZ branch on not zero
BNP branch on not positive
BNM branch on not minus
BO branch on overflow
BNO branch on not overflow

The addressing mode can be either RR (the destination address is in a register) or RX (the destination address is calculate from base + index + displacement)

doing loop

BCT branch on count, this is auto-decrement the operand (RR-type) and branch when the value is 0.
BXLE branch on index low or equal
BXH branch on index high

calling subroutine

BAL branch and link (RR, RX) the return address is loaded into op1 and branch to the destination address in op2.

to return from subroutine

B r branch register r which store the return address. This is used in pair with BAL r

arithmetic/logic

A add
S subtract
M multiply
D divide
C compare
CL compare logical character

logical operations,

the operands can be RX RR SS SI
N and
O or
X xor
TM test under mask
SL shift left arithmetic/logical

`SR` shift right arithmetic/logical

string operations

`MVC` move characters
`CLC` compare logical characters
`TR` translate and test, string search
`TRT` translate and test table, table look up and character translation

conversion

`CV` convert from packed decimal to binary
`CVD` convert from binary to packed decimal
`PACK` convert from zoned decimal to packed decimal
`UNPACK` convert from packed decimal to zoned decimal
`ED` edit, convert packed decimal to zoned for display
`EDMK` edit and mask, similar to edit but use pattern to insert a currency symbol such as $

Example of a program to perform W = X + Y - Z. Assume W, X, Y, Z are in the memory.

```
PROGRAM
      START 0
      BALR  12,0
      USING *, 12
      L     2, X        R2 = M(X)
      A     2, Y        R2 = R2 + M(Y)
      S     2, Z        R2 = R2 - M(Z)
      ST    2, W        M(W) = R2
      BR    14          STOP

X     DC    F '10'      DEFINE CONST FLOAT 10.0
Y     DC    F '3'       DEFINE CONST FLOAT 3.0
Z     DC    F '4'       DEFINE CONST FLOAT 4.0
W     DS    F           RESERVE STORAGE FLOAT
      END
```

Stack-based instruction set architecture

What is a stack machine?

Contrast to an ordinary processor of contemporary design which uses registers, a stack machine uses stack. A stack is a LIFO (last in first out) storage with two abstract operations: push and pop. Push puts an item into stack at the top. Pop retrieves an item at the top of stack.

Calculation using stack.

Because a stack is LIFO, any operation must access data item from the top. Stack doesn't need "addressing", as it is implicit in the operators, which use stack. Any expression can be transformed into a postfix order and stack can be used to evaluate that expression without the need for explicitly locating any variable. For example,

```
B + C - D  ==>
      B C + D -  (postfix)
      push val B, push val C, add, push val D, sub.
A = B  ==>
      A B =
      push ads A, push val B, store.
```

`add` takes top two items from stack add them and push the result back to stack. Similarly `sub` operators. `Store` takes one value and one address from stack and store value to address.

Let's compare the above expression to the calculation using registers.

```
B + C - D
   load r0, B
   load r1, C
   add r0, r1    ;  r0+r1 -> r0
   load r2, D
   sub r0, r2

A = B
   load r0, ads A
   load r1, val B
   store r1, (r0)  ; r1 --> (r0)
```

One can see that the main difference is that registers must be allocated, for example, `r0` is used to store temporary result while in a stack machine the temporary storage is implicit. ISA based on stack has an advantage over register-based ISA that it is very compact. As most instructions have implicit argument, the size of instruction is very short, usually one byte. Only a few instructions need argument, such as jump, push literal, that required more than one byte.

Example of stack ISA

We will illustrate an ISA that is based on a stack machine. Let us ignore local variables to simplify the presentation (therefore reduce the complication of an activation record). We need load, store, arithmetic operators, call, return and conventional jump and branch for flow of control.

Notation: TOP is the item on top of stack, NEXT is an item below TOP (therefore we can talk about 2 operands on stack by TOP, NEXT), M[ads] value of memory at ads. "pop a" is TOP `-->` a, "pop2" pops two items off stack.

`lit #a`	push the immediate value a.
`load`	pop a, push M[a].
`store`	NEXT -> M[TOP], pop2.
`add`	NEXT + TOP -> a, pop2, push a.
`cmp`	if NEXT > TOP a = 1 else a = 0, pop2, push a.
`call`	pop a, create new activation record, goto a.
`return`	delete current activation record, go back pc'.
`jz #a`	if pop = 0 then goto a.

Please note that except `lit #a` and `jz #a` which has `#a` as argument, all other instructions have argument(s) implicit in the stack. The state of computation consists of a stack pointer and a program counter. If we have two stacks one for computation and one for activation record (called control stack), we need only to store the program counter (return address) in the activation record and there is no need to do anything to computation stack on subroutine calls. Calling a subroutine need just push the current program counter (return address) onto the control stack. Returning is just pop the control stack and restore the previous program counter.

Reduced Instruction Set Computer

As high-level languages became popular and started to replace assembly languages the design of instruction set began to take the central stage. The ISA design of that period (circa 1970) emphasised the support of high-level languages using instructions that perform complex operations such as move block of characters and having various addressing modes to accommodate accessing the data structure of high-level languages. The intention is that with these complex instructions the "level" of assembly languages will be lifted up to be nearer to the high-level languages. (The difference between high-level languages and assembly languages is called "semantic gap" [ILI82]). Thus, simplify the construction of compiler (which was one of the most complex programs of those days). The ISA design also emphasised the small size of the executable code. The reason is that by having a small code size, the program will run faster. One obvious fact is there will be fewer instructions to be fetched from memory.

However, because of their complexity, the complex instructions require many cycles to execute. The control unit was more difficult to design and the technique of "microprogram" became the standard engineering tool to battle this complexity. The complexity of a control unit can be measured by the size of the microprogram (the DEC VAX 11/780 has 5140×96 bits of microprogram, it has one of the most complex ISA [LEV89]). This complexity resulted in the longer cycle time. The other negative aspect of the complex ISA is that the pipeline scheduling is not very effective and the cost of stall is very high.

The study of dynamic execution of instructions of the programs written in high-level languages [PATT82] [LUN77] [HUC83] showed that 1) the most frequently used instructions are the simple instructions 2) compilers do not use much of the complex instructions as it is difficult to match the context (conditions) of statements in the language to specialised instructions, therefore the compiled code contained mostly simple instructions. Table 2.4 shows the result from [PATT82].

Arming with these findings, the movement of the new direction is designing instruction set had begun [PAT82] [PAT85] [STA88]. The ISA design was in the contrast with the earlier ISA, this new ISA emphasised on 1) making the simple instructions run fast 2) making the pipeline efficiency the main concern. This idea led to the effort to make every instruction to run in one cycle. The main technique is to have load/store instruction set and making use of large number of registers to store local values and to pass parameters between call/return. The visible characteristic is that the new ISA has simplified instruction set (this does

not mean the number of instruction is reduced), for example, the number of addressing mode is restricted, the complex instructions which can not be completed in one cycle are abandoned, some complex operations are achieved by using a sequence of simple instructions instead.

Table 2.3 Weighted relative dynamic frequency of high-level languages operation

	Dynamic occurrence		Machine instruction weighted		Memory referenced weighted	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45	38	13	13	14	15
LOOP	5	3	42	32	33	26
CALL	15	12	31	33	44	45
IF	29	43	11	21	7	13
GOTO	–	3	–	–	–	–
OTHER	6	1	3	1	2	1

The other main departure from the previous ISA design is the emphasis on using compilers to schedule efficient codes. Many techniques in the new ISA requires sophistication of the compiler such as the use of delay branch requires compilers to be able to fill in the delay slot. Fortunately, the software technology has been advanced to the stage that writing this sophisticate compiler becomes possible. With simplified instruction set, compilation techniques achieve a good deal of efficiency. It was easier to generate a good code for this simplified ISA than for a complex ISA. The result from this new thinking is that CPI of processor approaches 1.0. The control unit is simplified to the point that the hardwired circuit is practical. The cycle time is reduced.

The complex instruction set was named "Complex Instruction Set Computer" (CISC) in contrast to the simplified instruction set which was then called "Reduced Instruction Set Computer" (RISC). The year 1980-1990 becomes the golden age of the RISC philosophy when the microelectronics industry has matured and it is possible to produce a high performance processor on a chip. The RISC design has dominated the market and becomes synonymous with high performance. Because of the regularity inherent in the RISC design, the computer-aided design (CAD) tools can be applied easily to the design and test process, hence it accelerates the time to market of the new processors. However, the compatibility of the old software keeps the complex instruction set alive, notable the Intel family of microprocessors, the 80x86 and later the Pentium family.

Processor	Decode complexity			Pipelining difficulty			
	No. of Inst. sizes	Max. Inst. size in bytes	No. of addressing modes	indirect addressing	load/store with combined arithmetic	Max. no. of memory operands	unaligned addressing allowed
MIPS R2000	1	4	1	no	no	1	no
SPARC	1	4	2	no	no	1	no
HP PA	1	4	10	no	no	1	no
IBM RS/6000	1	4	4	no	no	1	yes
IBM 3090	4	8	2	no	yes	2	yes
Intel 80486	12	12	15	no	yes	2	yes
MC68040	11	22	44	yes	yes	2	yes
VAX	56	56	22	yes	yes	6	yes

Figure 2.2 Characteristics of some processors

Figure 2.2 shows characteristics of some processors that illustrate the difference between CISC and RISC designs. The first four processors: MIRS R2000, SPARC, HP PA and RS/6000 are RISC. They have one fixed instruction size, small number of addressing modes, has no indirect addressing, no load/store combined with arithmetic instructions and has maximum one memory operand. The other four processors: IBM 3090, Intel 80486, MC68040 and VAX are CISC. This example is chosen to contrast both schools of thought, however, the division between them is not black and white. There are many ISA that fall in between.

The evolution of idea in the ISA design of both generations (CISC and RISC) is the change according to the technological force. The CISC was successful because of microprogramming technique as well as RISC was successful because of the single chip processor technology. The success of both ideas in the past can be a good example how a particular tradeoff is achieved. The lesson learn can be applicable to the future ISA design which definitely will be affected by the technology yet to come (such as DNA computing and nanoelectronics).

The current design uses both ideas in the implementation of a processor [HEN91] [FLY98] [FLY99]. The control is divided in to two parts 1) the execution of basic instructions and 2) the execution of the complex instructions. The basic instructions will be completed in one cycle and multiple issued. The complex instruction will have very deep pipeline, for example the Intel Pentium has 14 stages pipeline in one model. The complex instructions can also be translated at run-time into wide internal micro-operations, which simplify the multicycle pipeline especially for floating-point operations. Flynn said in one of his article [FLY97] that

> "Tradeoffs between computer design cost-performance and programmer accessible functionality are as current a problem today as they were in 1953."

and concerning the debate whether CISC or RISC is better that

> " ... Actual performance differences in instruction set efficiency are slight, but these differences still stir passions among hardware designers. Within the past few years, there has been a continuing (and generally unproductive) debate over the cost-performance benefits of the so-called RISC instruction sets over earlier instruction sets labeled CISC."

No doubt, the instruction set design of the future processor will have another revolutionary idea as much as RISC has over CISC in the past.

References

[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.

[CAS78] Case, R., and A. Padegs, A., "Architecture of the IBM System/370", Communication of the ACM, 21(1978): 73-96.

[FLY97] Flynn, M., "Introduction to :Influence of Programming Techniques on the Design of Computers", Proceedings of the IEEE, Volume 85, no. 3, March 1997, pp. 467-469.

[FLY98] Flynn, M., "Computer engineering 30 years after the IBM Model 91", Computer, Volume 31, no. 4, April 1998, pp. 27 -31.

[FLY99] Flynn, M., Hung, P., Rudd, K., "Deep submicron microprocessor design issues", IEEE Micro, Volume 19, no. 4, July-Aug. 1999, pp. 11-22.

[HEN91] Hennessy, J., Jouppi, N., "Computer technology and architecture: an evolving interaction", Computer, Volume 24, no.9 , Sept. 1991, pp. 18-29.

[HUC83] Huck, T., Comparative analysis of computer architectures, Stanford university technical report no. 83-243, May 1983.

[IBM94] International Business Machines, Inc., The PowerPC architecture: A specification for a new family of RISC processors. San Francisco: CA, Morgan Kaufmann, 1994.

[ILI82] Iliffe, J., Advanced computer design, Prentice-Hall, London, 1982.

[LEV89] Levy M., and Eckhouse, R., Computer programming and architecture: the VAX, Bedford, Mass., Digital Press, 1989

[LUN77] Lunde, A., "Empirical evaluation of some features of instruction set processor architecture", Comm. of the ACM, March 1977.

[PAT82] Patterson, D., and Sequin, C., "A VLSI RISC", Computer, 15, no. 9, September, 1982, pp. 8-21.

[PAT85] Patterson, D., "Reduced instruction set computers", Comm. of the ACM, 28, no.1, January 1985.

[STA88] Stallings, W., "Reduced instruction set computer architecture", Proc. of the IEEE, vol. 76, no. 1, January 1988, pp. 38-55.

Chapter 3

Computer Arithmetic

The arithmetic logic unit (ALU) is the part of the processor that performs calculation both the arithmetic and the logic operations. It composed of functional units and registers including some status bit for storing the result of operations such as zero and overflow. The functional units included adder, multiplier and shifter. As an ALU is realised using logic gates, it relies on the computer arithmetic algorithms to perform calculation by repetition such as using multiple add-shifts to do multiplication. This enables complex calculations such as floating point operations possible on an economical hardware.

Number representation

Decimal system

$$A = 1957_{10}$$
$$A = 1 \times 10^3 + 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

A is expressed in a decimal number. The base is 10. This representation has 10 symbols 0, 1, 2, … 9 which constitutes *digits*.

Binary system

A number is represented as sum of weights that are a power of 2. The base is 2 and there are two symbols 0, 1 called *binary digits* or *bits*.

$$A = 10101_2$$
$$A = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$A = 2^4 + 2^2 + 2^0 \quad = 21_{10}$$

A number can be represented by *n*-bit in many ways. For an integer, there are unsigned, sign-magnitude and two's complement representation.

unsigned integer

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

An unsigned integer ranges over non negative numbers. For *n*-bit integer its range is $0 \ldots 2^n - 1$.

sign-magnitude

The left most bit is sign, the right most $n-1$ bit is magnitude. It has several drawbacks. First addition and subtraction require special treatment of sign and relative magnitudes. Second, the number zero has two representations +0, –0.

two's complement

We have seen how to represent an unsigned integer but how a negative number can be represent without using sign-magnitude? Suppose we have 3-bit binary $a_2a_1a_0$ which can represent $2^3 - 8$ positive numbers for 000 to 111 (0 to 7). The fourth bit can be introduced to associate with the negative weight -2^3. The 4-bit number can represent 1000_2 (-8_{10}) to 0111_2 ($+7_{10}$). The decimal value is

$$A = a_3 \times -2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

The number is negative is $A_3 = 1$. The properties of this representation are

1 Bit A_3 gives the sign of the equivalent decimal number, A_3 =1 negative, $A_3 = 0$ positive.
2 There is one zero and it is positive.
3 A positive decimal number is changed to a negative number of the same absolute value by inverting each bit followed by adding a 1.

a_3	a_2	a_1	a_0	decimal
1	0	0	0	–8
1	0	0	1	–7
1	0	1	0	–6
1	0	1	1	–5
1	1	0	0	–4
1	1	0	1	–3
1	1	1	0	–2
1	1	1	1	–1
0	0	0	0	+0
0	0	0	1	+1
0	0	1	0	+2
0	0	1	1	+3
0	1	0	0	+4
0	1	0	1	+5
0	1	1	0	+6
0	1	1	1	+7

Figure 3.1 4-bit two complement numbers

Example Convert 110_2 ($+6_{10}$) to a negative number -6_{10}.
0010 inverse to 1001, 1001 plus 1 is $1010_2 = -6_{10}$
This number is called *two's complement* of the original number.

The following expression defines the two's complement representation for both positive and negative numbers. if A is positive, the sign bit (a_{n-1}) is zero. The range of positive number is $0 \ldots 2^{n-2}$. The range of negative number is $-1 \ldots -2^{n-1}$.

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Integer arithmetic

Addition and subtraction

Using two's complement representation, subtraction is performed by adding the two's complement. For example, 5 – 3 = 2, (+5) + (–3) = 2, (0101) + (1101) = **1**0010. The left most bit (carry bit) is overflowed. We ignore the overflow and the result is 0010 = 2. On any addition, the result may be larger than can be held in the word size being used. This condition is called *overflow*. When overflow occurs, the ALU signals the condition codes. The overflow rule is: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Subtraction is achieved using addition. We can demonstrate by the following example. If B = –A, then A + B = A + (–A) = 0. For *n*-bit integer, B is a bitwise complement of A plus 1, that is –A. Let a_n' be a complement of a_n .

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$B = -2^{n-1}a_n' + 1 + \sum_{i=0}^{n-2} 2^i a_i'$$

$$A + B = -(a_n + a_n')\,2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i (a_i + a_i')$$

$$= -2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i$$

$$= -2^{n-1} + 2^{n-1} = 0$$

Multiplication

Multiplication is a complex operation. Multiplication firstly generates partial products, one for each digit in the multiplier, then summed them to produce the final product. Each successive partial product is shifted one position to the left relative to the preceding partial product. The multiplication of a binary number 2^n is accomplished by shifting that number to the left by *n* bits. The multiplication of two *n*-bit integers results in a product of up to 2*n* bits in length.

```
          1 0 1 1 ×        Multiplicand
          1 1 0 1          Multiplier
          1 0 1 1
        0 0 0 0            Partial products
      1 0 1 1
    1 0 1 1
-------------------
1 0 0 0 1 1 1 1          Product
```

Figure 3.2 Multiplication of unsigned integers

One of the well-known algorithms for two's complement multiplication is Booth's algorithm [BOO51]. Let Q, M, A be three *n*-bit registers, Q stores multiplier, M multiplicand, the result appears in AQ. A concatenates to Q and when shifting right, the least significant bit of A will go to the most significant bit of Q. There is one bit placed to the right of the least significant bit of Q (Q0), designated Q-. Booth's algorithm is as follows:

```
A = 0, Q- = 0, M = multiplicand, Q = multiplier
repeat n times
    if (Q0, Q-) = 01 then A = A + M
                = 10 then A = A – M
    arithmetic shift right A, Q, Q-    {preserve sign bit}
end
```

Note the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of one addition or subtraction per block.

```
          0 1 1 1 ×
          1 1 0 1 (0)
1 1 1 1 1 0 0 1      1-0
0 0 0 0 1 1 1        0-1
1 1 1 0 0 1          1-0
-------------------
1 1 1 0 1 0 1 1
```

Figure 3.3 example of Booth's algorithm for (7) × (–3) = –21

Why Booth's algorithm work?

Observe that the number to partial product sum can be reduce. Consider a positive multiplier where one contiguous 1s surrounded by 0s. The number of shift-and-add can be reduced by observing that

$$2^{n} + 2^{n-1} + \ldots 2^{n-K} = 2^{n+1} - 2^{n-K}$$

Example

$$M^* (011110) = M^* (2^4 + 2^3 + 2^2 + 2^1) = M * (2^5 - 2^1)$$

The product can be generated by one addition and one subtraction of the multiplicand. Booth's algorithm performs subtraction when first 1 of block is encountered (1-0) and addition when the end of block is encountered (0-1). This scheme extends to any number of blocks of 1s in a multiplier and negative number.

Division

unsigned binary division

The division is based on the long division. It involves repetitive shifting and addition or subtraction. Dividend is examined bit by bit from left to right until it is greater than or equal to the divisor, 0s are placed in the quotient, when it is divisible, 1 is placed in the quotient and the divisor is subtract from the partial dividend. Additional bits from the dividend are appended to the *partial remainder* until the result is greater than or equal to the divisor then the cycle repeat.

```
                          0 0 0 0 1 1 0 1    Quotient
Divisor          1 0 1 1 / 1 0 0 1 0 0 1 1    Dividend
                            1 0 1 1
Partial remainders          0 0 1 1 1 0
                              1 0 1 1
                              0 0 1 1 1 1
                                  1 0 1 1
                                    1 0 0    Remainder
```

Figure 3.4 Division of unsigned binary integers

The algorithm is as follows:

```
{ unsigned integer divide }
A = 0, M = divisor, Q = dividend
repeat n times
        shift left A, Q
        A = A - M
        if A < 0 then Q0 = 0, A = A + M
                else Q0 = 1
end {quotient in Q, remainder in A}
```

two's complement division

This scheme, with some difficulty, can be extended to negative numbers. The divisor must be expressed as 2*n*-bit two's complement number.

```
{ two's complement integer divide }
M = divisor, A Q = dividend
while there are bits in Q
        shift left A Q
        if (M and A have the same sign) then A = A - M
                                        else A = A + M
        if (sign of A not change) or (A = 0 AND Q = 0 ) then Q0 = 1
        if (sign of A change) and (A ≠ 0 OR Q ≠ 0) then Q0 = 0; restore
              the previous A
if (divisor and dividend are not same sign) then two's complement Q
end {quotient is in Q, remainder is in A }
```

```
   A           Q        M = 1101
0 0 0 0     0 1 1 1     Initial value
0 0 0 0     1 1 1 0     Shift
1 1 0 1                 Add
0 0 0 0     1 1 1 0     Restore
0 0 0 1     1 1 0 0     Shift
1 1 1 0                 Add
0 0 0 1     1 1 0 0     Restore
0 0 1 1     1 0 0 0     Shift
0 0 0 0                 Add
0 0 0 0     1 0 0 1     Set Q0 = 1
0 0 0 1     0 0 1 0     Shift
1 1 1 0                 Add
0 0 0 1     0 0 1 0     Restore
```

Figure 3.5 example of two's complement division (7) / (−3)

Floating- Point Numbers

Very large and very small numbers can be represent using scientific notation which separately store *significand* and *exponent*, such as $2.14 * 10^{12}$. This allow a range of very large and very small numbers to be represented using only a few digits. In binary numbers, a number is represent in the form:

$$\pm \text{Significand} \times \text{Base}^{\pm \text{Exponent}}$$

This number can be stored in a binary word using three fields: Sign bit, Significand and Exponent. The base is implicit. The exponent can be stored with bias, i.e. a bias is subtracted from the field to get the true value. An example of 32-bit floating-point format is 1 bit sign, 8 bits biased exponent and 23 bits significand. The bias is 128.

$0.11010001 \times 2^{10100} = 0\ 10010100\ 10100010000000000000000$
$-0.11010001 \times 2^{10100} = 1\ 10010100\ 10100010000000000000000$
$0.11010001 \times 2^{-10100} = 0\ 01101100\ 10100010000000000000000$
$-0.11010001 \times 2^{-10100} = 1\ 01101100\ 10100010000000000000000$

Figure 3.6 an example of 32-bit floating-point format

To simplify the operations on floating-point numbers, it is required that they be *normalized* in the form:

$$0.1\text{bbb}\ldots\text{b} \times 2^{\pm E}$$

Therefore the left most bit of significand is always 1 and is "implicit" (no need to store this bit).

Range of representable numbers

With the above representation the following ranges of numbers are possible:
Negative numbers between $-(1 - 2^{-24}) \times 2^{127}$ and -0.5×2^{-128}
Positive numbers between 0.5×2^{-128} to $(1 - 2^{-24}) \times 2^{127}$

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(1 - 2^{-24}) \times 2^{127}$, called *negative overflow*
- Negative numbers greater than -0.5×2^{-128} , called *negative underflow*

- Zero
- Positive numbers less than 0.5×2^{-128} , called *positive underflow*
- Positive numbers greater than $(1 - 2^{-24}) \times 2^{127}$, called *positive overflow*

Remember that the maximum number of different values that can be represented with 32 bits is still 2^{32}. The numbers represented in floating-point notation are not spaced evenly along the number line. The possible values get closer together near the origin and farther apart as you move away. This is one of the trade-off of floating-point: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

IEEE standard 754

The most important floating-point representation is defined in IEEE Standard 754 [IEE85]. The IEEE standard defines both a 32-bit single and a 64-bit double format. The single format has a sign bit, 8-bit biased exponent, 23-bit significand. The exponent bias is 127. The double format has a sign bit, 11-bit biased exponent, 52-bit significand. The exponent bias is 1023. The implied base is 2. The standard defines two extended formats, single and double, whose exact format is implementation-dependent. The extended formats are to be used for intermediate calculations.

There are some bit patterns that are used to represent special numbers such as zero, plus/minus infinity, NaN (not a number) and denormalized number etc.

numbers	bias exponent	fraction	value
zero	0	0	± 0
infinity	2047	0	$\pm$ infinity
NaN	2047	$\neq 0$	NaN
denormalized	0	$f \neq 0$	$\pm 2^{e-1022} (0.f)$

Figure 3.7 special numbers of IEEE 754 (double precision)

Floating- Point Arithmetic

For addition and subtraction, it is necessary for both operands to have the same exponent. This may require shifting the radix point to achieve the alignment. The multiplication and division are more straightforward. When the significand

is underflow the rounding operation is required. Likewise when it is overflow the realignment (normalized) is required.

Let x, y be two floating-point numbers; xs, ys be the significands; xe, ye be the exponents. Let xe ≤ ye. The floating-point numbers arithmetic operations:

$$x = xs\ B^{xe}$$

$$y = ys\ B^{ye}$$

$$x + y = (xs\ B^{xe - ye} + ys)\ B^{ye}$$

$$x - y = (xs\ B^{xe - ye} - ys)\ B^{ye}$$

$$x \times y = (xs \times ys)\ B^{xe + ye}$$

$$x / y = (xs / ys)\ B^{xe - ye}$$

Addition and Subtraction

There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros
2. Align the significands
3. Add or subtract the significands
4. Normalized the result

Let msd = most significant digit , S = significand, E = exponent

The addition-subtraction algorithm is as follows:

1. made implicit bit explicit
2. check operand 0
3. align by shifting smaller number to the right (increment its E) until two E are equal
4. check 0
5. add signed S
6. check 0
7. check S overflow if so shift right
8. check E overflow if so report error
9. normalize result, shift S left until msd is not zero, decrement E, E may underflow
10. rounded off the result

Multiplication

The multiplication and division are simpler than addition and subtraction. The multiplication algorithm is as follows:

1. check operand 0
2. xe + ye
3. substract bias
4. check E overflow, underflow
5. sign-magnitude multiply S
6. normalized result and rounded (E may underflow)

Division

1. check operand 0
2. xe – ye
3. add bias
4. check E overflow, underflow
5. divide S
6. normalized and rounded result

Precision considerations

Guard bits

For the floating-point operations the significands are loaded into the registers. The length of the register is almost always greater than the length of significand plus an implied bit. The register contains an additional bit, called *guard bits*, the are used to pad out the right end of the significand with 0s. The purpose is to prevent the lost of least significant bit when one operand must be shifted right during floating-point operation. As seen from the following example: a subtraction without and with guard bits.

```
Without guard bit
  1.000 . . . 00 × 2
  0.111 . . . 11 × 2  –
= 0.000 . . . 01 × 2
= 1.000 . . . 00 × 2^-22
```

```
With guard bits
  1.000 . . . 00 0000 × 2
  0.111 . . . 11 1000 × 2  −
= 0.000 . . . 01 1000 × 2
= 1.000 . . . 00 0000 × 2^-23
```

Rounding

The rounding policy affects the precision of the result. IEEE standard lists four approaches:

- Round to nearest – to the nearest representable number
- Round toward positive infinity
- Round toward negative infinity
- Round toward 0 (truncated)

Round to the nearest is the default rounding mode in the standard. The rounding to plus and minus infinity is useful in implementation of interval arithmetic. In the interval arithmetic an upper bound and lower bound on the correct answer are kept. If the range between the upper and lower bounds is sufficiently narrow, it indicates that a sufficiently accurate result is obtained.

Denormalized number

Denormalized numbers are included in IEEE 754 to handle E underflow, the result is denormalized by right-shifting S and increment E until E is within representable range. This method is also referred to as "gradual underflow" [COO81].

References

[BOO51] Booth, A. "A signed binary multiplication technique." Quarterly Journal of Mechanical and Applied Mathematics, vol. 4, pt. 2, 1951.

[COO81] Coonen, J. "Underflow and Denormalized numbers", IEEE Computer, March 1981.

[GOL91] Goldberg, D., "What every computer scientist should know about floating-point arithmetic,", ACM Computing Surveys 23:1, 5-48.

[IEE85] Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.

[KNU81] Knuth, D. The art of computer programming, Volume 2: seminumerical algorithms, Addison-Wesley, 1981.

[OMO94] Omondi, A. Computer Arithmetic Systems: Algorithms, architecture and implementations, Prentice-Hall, 1994.

[SWA90] Swartzlander, E., ed., Computer arithmetic, Volumes I and II, IEEE Computer society press, 1990.

Chapter 4

Control unit

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

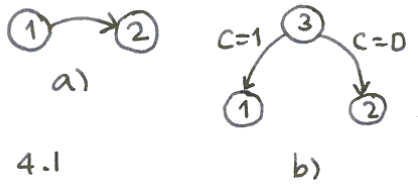
Control unit can be implemented by hardwired or by microprogram. A computer designer strives to optimise three aspects of control unit design:

1. the complexity (hence cost) of the control unit
2. the speed of control unit
3. the engineering cost of the design (time, correctness etc.)

Hardwired control unit

In the past, hardwired control unit is very difficult to design hence its engineering cost is very high. Presently, the emphasis of computer design is the performance therefore hardwired design 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 unit.

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



a) event : go to next state
b) event : go to state 1 or state 2 depends on conditionals

Figure 4.1 several types of states in a state diagram

From the state diagram, a hardware realization can be constructed almost automatically by some CAD tools. The in-depth topic of logic design for sequential circuits and logic minimization can be consulted from many basic textbooks on the subject such as Katz [KAT93]. 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 combinational circuits with feedback. The feedback information is 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 speed 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.

The bound of complexity of control is States × Control inputs × Control outputs

Microprogrammed control unit

Maurice Wilkes invented "microprogram" in 1953 [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. 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.

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.

To realise this idea it required a high speed memory which was not possible at that time. The reason for speed is that as the control unit determines how fast a sequence of operations can be executed, the bottle neck becomes the speed of accessing the microprogram which is stored in a special memory. At IBM, a chief architect of IBM 360 family, Gene Amdahl, has recognised the importance of microprogram and committed to implement it for IBM 360. The in-house development for the high speed memory was pursued. IBM had a great success for her 360 family.

How microprogram work

Like the RAM model, a microprogrammed control unit consists of microPC, micromemory, output buffer and a sequencing unit (Fig 4.2). A micromemory (sometimes called microstore) contains bit patterns that are used to control the datapath. Each word of the micromemory is called "microword". Each word of the micromemory is separated into several fields used for internal control, external control, conditional and specifies the next address. Internal control bits are the signals that control the datapath. External control bits are the signals that control external units such as memory (read, write), interrupt acknowledge etc. Conditionals are the bits that are used to determine the flow of microprogram; loop, branching, next instruction etc. Its input comes from the datapath (usually from the conditional code register). Next address determines the next microword to be executed.

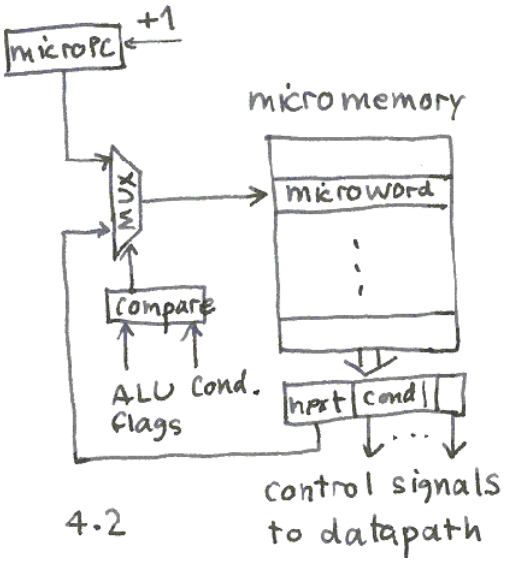


Figure 4.2 a microprogrammed control unit

A microprogram is executed as follow :

1. a word from microprogram at the location specified by the microPC is read out, control bits are latched at the output buffer which is connected to the datapath.
2. if 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 microPC will be incremented (execute the next microword).

What that has been described is called *horizontal microprogram* in which there is a one-to-one relationship between internal/external control bits and the actual control signal of the datapath (hence it is wide or "horizontal"). The microword can have other formats. There are several possibilities :

1. single format – one address, as just described above.
2. single format – two addresses, each microword contains two next addresses field, one for result of test true, the other for result of test false.
3. multiple format, 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 clock.

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 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 datapath. This approach is called *vertical microprogram*. There are many possibilities to compact the micromemory to be as small as possible, sometime trading off speed for space, for example, 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.

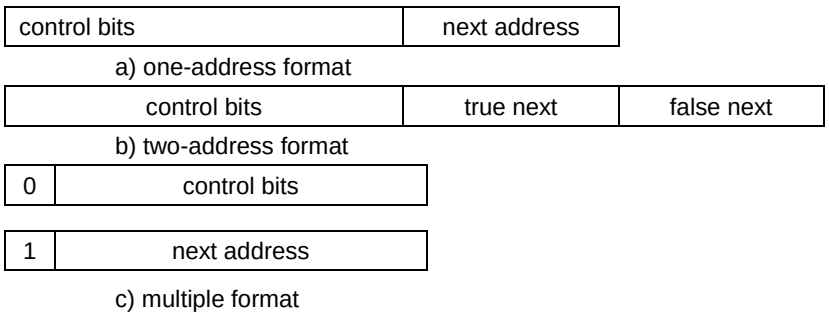


Figure 4.3 several formats of microword

Microprogram becomes obsolete mainly because the present design emphasizes the performance and microprogram is slower than hardwired. The change in instruction set design toward a minimum number of clock 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, hardwired control unit offers a lower engineering cost.

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 is 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 Z is { EQ, LT, GT }. One can write the logic expression of Z as

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.

Z = compare(A,B)

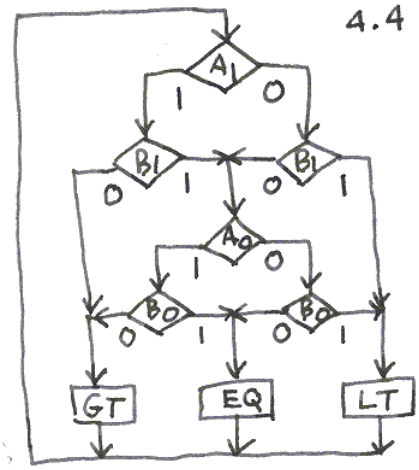


Figure 4.4 diagram of compare

Each primitive can be described as follows:

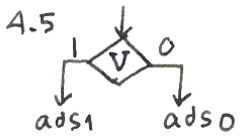


Figure 4.5 test element

test

if V is true then goto ads1 else goto ads0

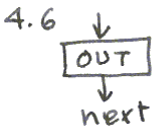


Figure 4.6 assignment element

assignment

output OUT and goto next

The above diagram can be translated into "microprogram" as follows :

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

Next, the microprogram 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)

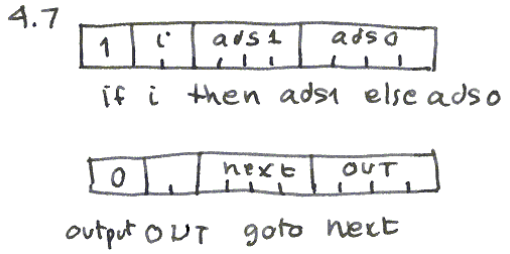


Figure 4.7 microword format for compare

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

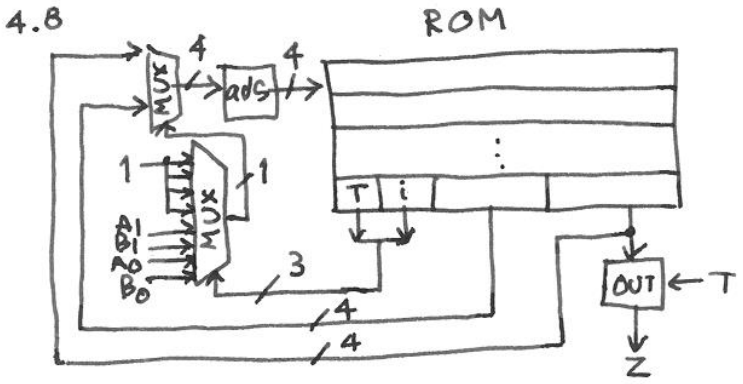


Figure 4.8 microprogrammed unit to realise the function compare

The microprogrammed unit to realise the function compare is shown in Fig. 4.8. How many clocks it takes to evaluate compare (A, B)? Observing the diagram (Fig. 4.4), on the longest path, there are 5 "steps" to traverse the diagram hence it takes 5 clocks to evaluate this function using the microprogrammed unit above.

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

Conclusion

As the history tells us, the microprocessor followed the same trend as earlier computer designs. Because of the limit of resource (the number of transistor in a chip), hardwired control was implemented and the instruction set architecture was toward a simple design. The advantage of simpler design for control unit and ease of change popularised microprogramming. Microprogram made it possible to achieve more complex instruction sets. With a much larger micro memory a machine as elaborate as the VAX [LEV89] is possible. In 1984, DEC wanted to offer a cheaper machine with the same instruction set as VAX. They reduced the instructions interpreted by microprogram by trapping some instructions and performing them in software. They discovered that 20% of VAX instructions occupied 60% of the microprogram, and yet they are used (executed) only 0.2% of the time. The simpler subset of VAX ISA, called MicroVAX-1, implemented 80% of VAX instruction in microprogram, other 20% is trapped to software, has the size of micro-memory reduced from 480K (VAX) to 64K (MicroVAX-1), and perform 90% of the performance of VAX-11/780. This is also an evidence toward a new thinking in instruction set design. The current design sees the revive of the idea of translating between the real executable code into the internal code which is suitable for controlling the functional units [GEP00] [KLA00]. The idea of "code translation" is used to

retain the ISA compatibility for the existing software to be run on the new hardware.

References

[GEP00] Geppert, L. and Perry, T., Transmeta's magic show, IEEE Spectrum, vol 37, no. 5, May 2000, pp.26-33.

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

[KLA00] Klaiber, A., The technology behing Crusoe processors, White paper, Transmeta Corp., January 2000, http:// www.transmeta.com/ crusoe/ download/ pdf/ crusoetechwp.pdf.

[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.

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

Chapter 5

Processor Design: S1 a simple CPU

To illustrate how a processor can be designed, we will describe the design of a simple hypothetical CPU called S1. It contains all the important elements of a real processor. The design is aimed to be as simple as possible so that students can understand it easily. The architectural description of S1: its organization (structure), its instruction set (ISA) and its behaviour (microsteps), is small enough to fit into a few pages. A simulator of S1 at an instruction-level is also provided. Studying how the simulator work will enable students to modify and design their own processors.

S1 is a 16-bit processor. The instruction format is 16-bit fixed length. The address space is 10-bit, i.e. it has 1024 16-bit words. It has 8 general purpose registers (R0..R7).

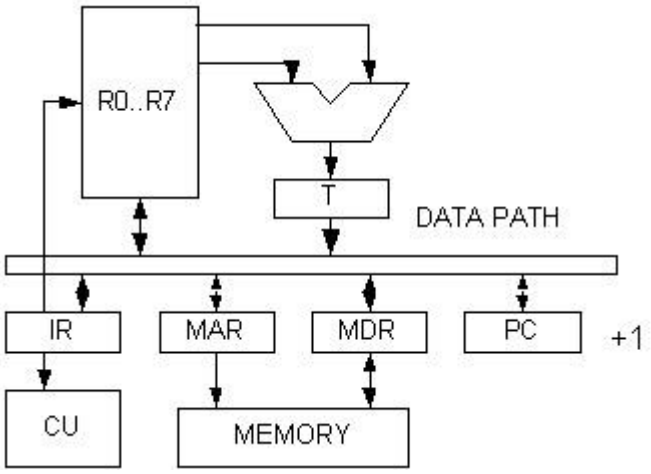


Figure 5.1 S1 microarchitecture

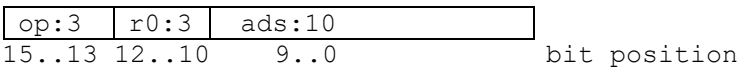
The register bank has one write port, two read ports (2 operands can be read and move to ALU in one cycle). The datapath is 16 bits. The ALU can perform {add, cmp, inc, sub1} and stores the output in a temporary T register. The instruction register IR stores the instruction to be decoded. IR is also connected to the control unit CU. The interface units to the memory consisted of a memory address register (MAR), and a memory data register (MDR). The program counter, PC, stores the current instruction address and can be incremented by 1 for the next instruction.

Instruction format

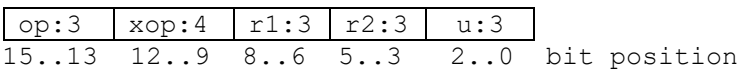
There are one long format (L-format) and one short format (S-format) for instructions. The opcode is 3 bits. This is not enough for all types of operations. One way to increase the number of opcode is to use "extended opcode". One opcode in L-format is used to designate the different format which has the additional field for more opcodes. This second format is the S-format. In S-format, the operands are registers, therefore there are enough room for more bits to encode the extended opcode. One opcode (7) denotes the extension of opcode from L-format to S-format. Another 4 bits is used (xop) to be the extended opcode. This is adequate for this simple machine and still have some room for an extension of its instruction set (such as floating-point operations).

The instruction has two formats. A field in an instruction is denoted `name:length`.

1 L-format : `op r, ads`



2 S-format : `7 xop r1, r2`



Instruction set

```
opcode  mnenomics           meaning

0  ld M, r              M -> r load from memory
1  st r, M              r -> M  store to memory
2  jmp c, ads           jump conditional
3  call  ads            push(PC), goto ads
7  xop

xop
0  mv r1,r2             r1 -> r2 move reg-reg
1  ld (r1),r2           (r1) -> r2 load indirect
2  st r1,(r2)           r1 -> (r2) store indirect
3  add r1,r2            r1 + r2 -> r1
4  cmp r1,r2            compare, affect Z,S
5  inc r1               increment r1
6  ret                  pop(PC)
```

where `r 0..7` , conditional code `c 0..6` is: `0 always, 1 Z, 2 NZ, 3 LT, 4 LE, 5 GE, 6 GT, M` is the address `0..1023`.

The instruction 0..3 use the L-format which has 3-bit opcode (i.e. at most 8 instructions) when the opcode is 7 the instruction use S-format which extend the operational code for another 4 bits (i.e. has maximum 16 extended instructions). There are only two addressing modes: register-register and load/store M to access the memory. There are no immediate or index addressing. (This is left as an exercise to add more addressing mode to S1). The jump instruction has seven conditions: always, equal, not equal, less than, less than or equal, greater than or equal, greater than. The condition is determined by the condition code S sign-bit, and Z zero-bit.

S1 microarchitecture

We study the operation of a hypothetical CPU in details, at the level of events happening every clock cycle when the CPU executes an instruction. Our description is in the form of Register Transfer Language (RTL) which represents the event of data movement inside a processor. Naturally, the description at this level of abstraction involves time. Each line of event happens in one unit of time (clock). We call this description "microstep".

The processor has the following registers: IR instruction register, PC program counter, MAR memory address register, MDR memory data register and general-purpose registers r0..r7. All registers are 16-bit. Two condition codes are Z zero bit and S sign bit. The memory has 1024 of 16-bit words.

Pc state

IR<0:15>
PC<0:15>
MAR<0:15>
MDR<0:15>
R[0:7]<0:15>
Z, S *zero, sign bit*
Run

Mp state

M[0:1023]<0:15>

S1 microsteps

Notation	
`// comment`	
`dest = source`	// data move from source to destination
`e1 ; e2`	// event e1 and e2 occur on the same time
`M[a]`	// memory at the address a
`IR:a`	// bit field specified by a of IR
`<name>`	// label of sequence of operations
`op( )`	// ALU function

```
// running a program
PC = 0
Run --> ( <ifetch>
          <execute> )

<ifetch>
MAR = PC
MDR = M[MAR]              // mem read
IR = MDR ; PC = PC + 1
```

```
<execute> := (              // instruction decoding
(op = 0) --> <load>
(op = 1) --> <store>
(op = 2) --> <jump>
(op = 3) --> <call>
(op = 7) --> <extend>
)

<extend> := (               // extended instruction decoding
(xop = 0) --> <move>
(xop = 1) --> <loadr>
(xop = 2) --> <storer>
(xop = 3) --> <add>
(xop = 4) --> <compare>
(xop = 5) --> <increment>
(xop = 6) --> <return>
)

<load>
MAR = IR:ADS
MDR = M[MAR]
R[IR:R0] = MDR

<store>
MAR = IR:ADS
MDR = R[IR:R0]
M[MAR] = MDR                // mem write

<loadr>
MAR = R[IR:R1]
MDR = M[MAR]
R[IR:R2] = MDR

<storer>
MDR = R[IR:R2]
MAR = R[IR:R1]
M[MAR] = MDR

<move>
T = R[IR:R1]
R[IR:R2] = T

<add>
T = add(R[IR:R1], R[IR:R2])
R[IR:R1] = T
```

```
<compare>
CC = cmp(R[IR:R1], R[IR:R2]) // condition code set

<increment>
T = add1(R[IR:R1])
R[IR:R1] = T

<jump>
if testCC(IR:R0)   // testCC( ) tests the IR:R0 against CC
then PC = IR:ADS

<call>
T = add1(R[7])
R[7] = T
MAR = R[7]         // sp+1 then put to stack
MDR = PC
M[MAR] = MDR
PC = IR:ADS

<return>
MAR = R[7]
MDR = M[MAR]       // get item then sp –1
PC = MDR
T = sub1(R[7])
R[7] = T
```

The instruction fetch can be faster by combining the PC + 1 with reading the instruction from the memory.

```
<ifetch2>
MAR = PC
IR = MDR = M[MAR]; PC = PC + 1
```

We made a number of assumptions here. The register bank is two read ports, one write port, reading and writing must not be on the same clock. Therefore it takes two clocks to move data between registers. The memory access is completed in one clock (assuming it has cache hit).

The timing of S1 in unit clock. Assume the instruction fetch takes 3 clocks and the instruction decode take 0 clock.

Table 5.1 S1 timing

instruction	clock
ld	6
st	6
jmp taken	5
jmp not-taken	4
call	9
mv r r	5
ld (r) r	6
st r (r)	6
add	5
cmp	4
inc	5
ret	8

Call and return take the longest time in the instruction set. Calling a subroutine can be made faster by inventing a new instruction that does not keep the return address in the stack (and hence the memory) but keeping it in a register instead. Jump and link (JAL) just saves the return address in a specified register and jump to the subroutine. Jump Register (JR) then does the reverse. It does the job of the "return" instruction. The register that stored return address must be saved to the memory (i.e. manage by the programmer) if the call to subroutine is nested. This will reduce the clock to 5 for "`jal`" and 4 for "`jr`". This shows that using registers can be much faster than using memory.

```
jal r, ads          store PC in r and jump to ads
jr r                jump back to (r)

<jal>
R[IR:R1] = PC
PC = IR:ADS

<jr>
PC = R[IR:R1]
```

Example of an assembly program for S1. Find sum of an array : sum a[0] .. a[N]

In a high level language

```
      sum = 0
      i = 0
      while ( i < N )
           sum = sum + a[i]
           i = i + 1
```

In S1 assembly language (with the translation to base-10 machine code, each field in an instruction is encoded as a number)

```
      .ORG 0                  // address code
      ld ZERO r0              0     0 0 20
      st r0 SUM               1     1 0 21
      st r0 I                 2     1 0 22
      ld N r1                 3     0 1 23
      ld I r3                 4     0 3 22
loop  cmp r3 r1               5     7 4 3
      jmp GE endw             6     2 5 16
      ld BASE r2              7     0 2 24
      add r2 r3               8     7 3 2 3
      ld (r2) r4              9     7 2 2 4
      ld SUM r5               10    0 5 21
      add r5 r4               11    7 3 5 4
      st r5 SUM               12    1 5 21
      inc r3                  13    7 5 3 0
      st r3 I                 14    1 3 22
      jmp loop                15    2 0 5
endw  ld SUM r0               16    0 0 21
      call print              17    3 0 1001
      call stop               18    3 0 1000

      .ORG  20                // data
ZERO  0                       20    0
SUM   0                       21    0
I     0                       22    0
N     100                     23    100
BASE  25                      24    25
a[0]                          25    a[0]
a[1]                          26    a[1]
...                           ...
```

S1 runs this program in 1110 instruction with 5963 clocks, CPI = 5.37

How to run the S1 simulator

The input file is an object file with the name "in.obj". The simulator will start and load "in.obj" and execute starting from PC=0 until stop with the instruction `call 1000`.

An object file has the following format

`a ads`	set PC to ads
`i op r ads`	instruction op
`i 7 xop r1 r2`	instruction xop
`w data`	set that address to value "data"
`t`	set trace mode on
`d start nbyte`	dump memory n byte
`e`	end of object file

Be careful, the input routine is not robust. A malformed input line can caused unpredictable result. The input loop is limited to 1000 words (to prevent infinite loop).

Control unit of S1

This section shows how to implement the control unit of S1 both hardwired and using microprogram.

Hardwired S1

The state diagram of S1 hardwired control unit (Figure 5.2) simply follows the microsteps. Each line of microstep is a state (assume decoding is done by a combinational circuit and it happens at the end of the instruction fetch without taking extra cycle, this can be achieved using a table lookup in a ROM). The number of cycle for each instruction will in exactly the same as the timing calculated from the microsteps (Table 5.1).

Some improvement can be made to the above design. To increase the speed the number of state for each instruction must be reduced. To reduce the complexity of the circuit, state should be shared wherever possible.

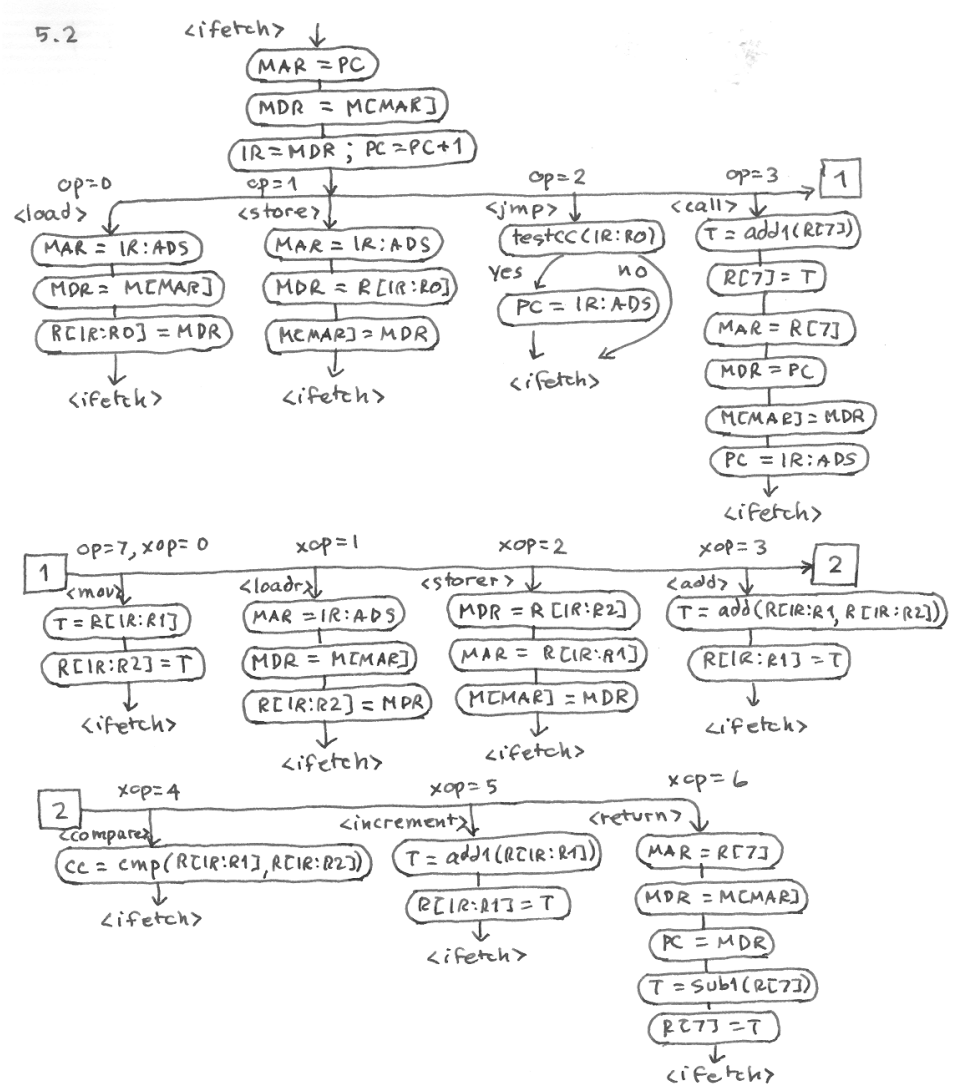


Figure 5.2 State diagram of S1 hardwired control unit

Reduce the number of state

```
   <store>
1. MAR = IR:ADS
2. MDR = R[IR:R0]
3. M[MAR] = MDR

   <storer>
1. MAR = R[IR:R2]
2. MDR = R[IR:R1]
3. M[MAR] = MDR
```

The above states (1 and 2 of both instructions) cannot be merged as both MAR and MDR is on the same internal bus, therefore can not be accessed at the same time. If two internal bus are available then these states can be merged into one (the register bank already has two read ports) and the number of cycle is reduced.

```
<store>
1. MAR = IR:ADS; MDR = R[IR:R0]
2. M[MAR] = MDR

   <storer>
1. MAR = R[IR:R2]; MDR = R[IR:R1]

2. M[MAR] = MDR
```

Share state

```
   <load>
1. MAR = IR:ADS
2. MDR = M[MAR]
3. R[IR:R0] = MDR

   <loadr>
1. MAR = R[IR:R1]
2. MDR = M[MAR]
3. R[IR:R2] = MDR
```

The states 3 of both instructions can be shared if `R0 == R2`. We can do that by changing the opcode format to use *fixed field encoding*. Moving the field R2 to the same field as R0, bit 12–10, and move the field xop to the back (Fig. 5.3).

Charing two states reduces the number of states, which reduces the complexity of the circuits.

L-format

op:3	r0:3	ads:10
15..13	12..10	9..0

bit position

S-format

op:3	r2:3	r1:3	xop:4	u:3
15..13	12..10	9..7	6..3	2..0

bit position

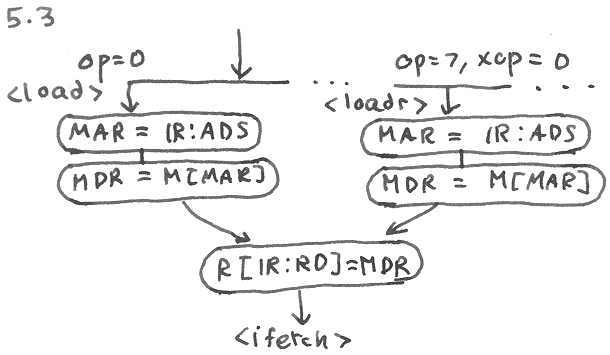


Figure 5.3 states of <load> and <loadr> after sharing

```
   <add>
1. T = add(R[IR:R1], R[IR:R2])
2. R[IR:R1] = T

   <increment>
1. T = add1(R[IR:R1])
2. R[IR:R1] = T
```

Another example of sharing states, for "`add`" and "`inc`", the states 2 of both instructions can be shared.

Microprogrammed control unit for S1

We use a single format microword. The fields are as follows :

Dest, Src :	specify destination and source for internal bus.
SelR :	selecting registers in register file.
Mctl :	memory control for read/write.
ALU :	specify function of ALU and latch the result to T register.
Misc :	other control signal such as PC + 1.
Cond :	for testing condition for jump to other microword.
Goto :	next address.

Dest =	{ MAR, IR, R, MDR, T, PC }
Src =	{ MAR, IR, R, MDR, PC, IR:ADS }
SelR =	{ IR:R0, IR:R1, IR:R2, IR:R12 }
ALU =	{ PASS1, ADD, SUB, ADD1 }
Mctl =	{ RD, WR }
Misc =	{ PC+1 }
Cond =	{ MRDY, Decode, U, testCC }

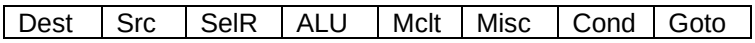


Figure 5.4 The format of a microword

Where MRDY is the memory ready signal (ignore in the simulator, assume no wait), Decode is a combination circuit that set microPC correctly to the appropriate address of the microprogram for the opcode, U is unconditional, testCC checks conditional code against the condition in the opcode (IR:R0) if the condition is false then jump to ifetch. Totally there are 29 microwords to implement the instruction set of S1. (Table 5.2)

The memory read/write step has "wait for memory ready" state. Because the use of cache memory, one can assume 0 clock waiting for memory ready when cache hits and more than 10 clocks for a miss penalty.

Let us go through the execution of one instruction. The instruction fetch starts with

```
0: MAR = PC
```

Table 5.2 S1 microprogram

Loc	Label	Dest	Src	SelR	ALU	Mctl	Misc	Cond	Goto	note
0	ifetch	MAR	PC							
1	w0					RD		MRDY	w0	
2		IR	MDR				PC+1	Decode		
3	load	MAR	IR:ADS							
4	w1					RD		MRDY	w1	
5		R	MDR	IR:R0				U	ifetch	
6	store	MAR	IR:ADS							
7		MDR	R	IR:R0						
8	w2					WR		MRDY	w2	
9								U	ifetch	
10	loadr	MAR	R	IR:R1						
11	w3					RD		MRDY	w3	
12		R	MDR	IR:R2				U	ifetch	
13	storer	MAR	R	IR:R2						
14		MDR	R	IR:R1						
15	w4					WR		MRDY	w4	
16								U	ifetch	
17	mov			IR:R12	PASS1					
18		R	T	IR:R2				U	ifetch	
19	add			IR:R12	ADD					
20		T	T	IR:R1				U	ifetch	
21	cmp			IR:R12	SUB			U	ifetch	set CC
22	inc			IR:R12	ADD1					
23		R	T	IR:R1				U	ifetch	
24	jmp							testCC	ifetch	cc false
25		PC	IR:ADS					U	ifetch	jump
26	jal	R	PC	IR:R0						
27		PC	IR:ADS					U	ifetch	
28	jr	PC	R	IR:R1				U	ifetch	

Dest and Src of the internal bus MAR and PC, then wait for memory to fill in MDR.

```
1: MDR = M[MAR]
```

Memory read (reading the current instruction), after memory cycle has completed,

```
2: IR = MDR ; PC = PC + 1
```

move the instruction to IR, increment PC, then branch to each instruction depends on IR:OP and IR:XOP (we will elaborate on this instruction decoding

mechanism later). Suppose the instruction is "load", the microprogram go to location 2 (load) and the following sequence occurs

```
3: MAR = IR:ADS
```

then waiting for memory then

```
4: MDR = M[MAR]
5: R[IR:R0] = MDR
```

The register is selected by IR:R0 and Dest and Src of internal bus are R and MDR. After completion, the microprogram branches back to instruction fetch (specified by the next address field).

For ALU instruction, for example, "`add`" the following sequence occurs after the instruction fetch, go to location 19 :

```
19: T = ADD(R[IR:R1], R[IR:R2])
```

the registers are selected and read: IR:R1, IR:R2; to ALU and ALU function ADD is activated. The result from ALU is latched to T register. Then the result is written to back to register selected by IR:R1 and the microprogram branches back to the instruction fetch.

```
20: R[IR:R1] = T
```

Totally the microprogram is 29 words. Each microword is in fact composed of the control bits that control the signals in the datapath. We will assign the bits to each field of microword as follows :

bit 0..4	Dest :	5 bits for write to R, PC, IR, MAR, MDR.
bit 5..10	Src :	6 bits for read from R, PC, IR, MAR, MDR, T.
bit 11..14	SelR :	4 bits for selecting IR:R0, IR:R1, IR:R2, IR:R1,R2
bit 15..18	ALU :	4 bits for ALU function : PASS1, ADD, SUB, ADD 1.
bit 19..20	Mclt :	2 bits for Mread, Mwrite
bit 21	Misc :	1 bit for PC + 1.
bit 22..25	Cond :	4 bits for jump control : Uncond, Mrdy, testCC, Decode.
bit 26..30	Goto :	5 bits, micro store has 29 addresses therefore 5 bits to address each of them.

So for the unencoded microword, the microword for S1 is 31-bit long. The instruction decoding, to branch to each microprogram sequence for each instruction, can be achieved by using IR:OP concatenate with IR:XOP (3 bits and 4 bits) to point to a jump table which contain the location of microword in the microprogram.

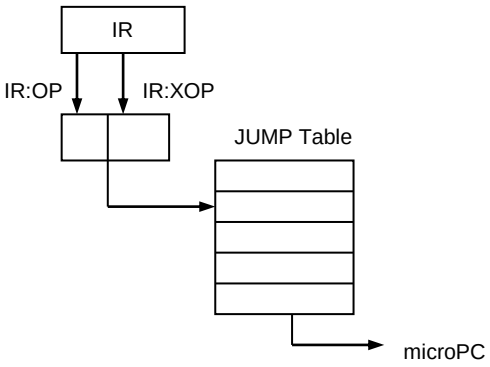


Figure 5.5 Scheme for decoding opcode in ifetch

The timing of microprogrammed S1 can be calculated by counting the number of microsteps required for each instruction. Fetching an instruction takes 3 steps (location 0, 1, 2). Assuming the instruction decoding happens at the same time as jumping to the proper location in the micromemory (takes zero cycle). For example, `ld` instruction takes 3+3 = 6 steps (execute at the location 3, 4, 5) and `st` instruction takes 3+4 = 7 steps (execute at the location 6, 7, 8, 9). The timing is shown in Table 5.3.

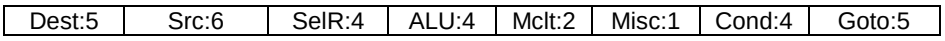
Table 5.3 Timing for microprogrammed S1

instruction	clock
ld	6
st	7
jmp uncond	5
jmp taken	5
jmp not-taken	4
jal	5
mv	5
ld (r) r	6
st r (r)	7
add	5
inc	5
cmp	4
jr	4

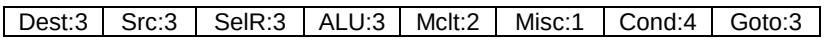
To reduce the width of the microword, each field can be encoded as follows :

Dest : 5 signals, 3 bits.
Src : 6 signals, 3 bits.
SelR : 4 signals, 3 bits (including NONE)
ALU : 4 signals, 3 bits.
Mctl : 2 bits
Misc : 1 bit.
Cond : 4 signals, 3 bits
Goto : only 6 distinct locations to jump to : ifetch, w0, w1, w2, w3, w4 – hence 3 bits.

Totally the encoded or vertical microprogram for S1 is 21-bit long.



a) unencoded microword (31 bits)



b) encoded microword (21 bits)

Figure 5.6 Comparing unencoded and encoded microword for S1

Calculating CPI

We will now illustrate how to calculate the CPI of both hardwired S1 and microprogrammed S1. Using the program benchmark GCC (a C compiler) we record the following instruction mix:

Table 5.4 GCC benchmark instruction mix

load	21%
store	12%
ALU	37%
set	6%
jump (uncond)	2%
jump taken	12%
jump not-taken	10%

CPI for S1 with hardwired control unit will be 5.23
(6 × .21 + 6 × .12 + 5 × .37 + 5 × .06 + 5 × .02 + 5 × .12 + 4 × .10)

CPI for S1 with microprogram control unit will be 5.35
(6 × .21 + 7 × .12 + 5 × .37 + 5 × .06 + 5 × .02 + 5 × .12 + 4 × .10)

Microprogram takes the time longer for "store", therefore its CPI is slightly higher. For the simulation run of "sum.asm" program CPI hardwired = 5.37, and CPI microprogram = 5.46

S1 microprogram simulator package

The package included the simulator of the S1 microprogrammed control unit and the microprogram generator, which takes the readable specification of microprogram and generates bit pattern for the micromemory. It is compiled and tested under Borland Turbo C compiler version 2.0. All the tools and simulators can be found on the web page of this book. The list of files is:

s1m.h, s1m.c, supportm.c	simulator files
mpgm.txt	microprogram file used by s1m.c
in.obj	test machine code
mgen.c, hash.c	microprogram generator
mspec.txt	input microprogram in human readable text
s1mx.txt	explain S1 instruction set and microprogram format.

To generate a microprogram, run mgen.exe, it takes input from mspec.txt and outputs a microprogram in the form that s1m.exe can read. (see mpgm.txt)

S1 microprogram bit position and coding form

bit	field	signal
0	dest	r
1		pc
2		ir
3		mar
4		mdr
5	src	r
6		pc

```
7           ir
8           mar
9           mdr
10          t
11 selr     ir:r0
12          ir:r1
13          ir:r2
14          ir:r1,r2
15 alu      pass1
16          add
17          sub
18          add1
19 mctl     rd
20          wr
21 misc     pc+1
22 cond     u
23          mrdy
24          testcc
25          decode
26 goto     5 bits 26..30
```

How to use mgen.c to generate microprogram

`Mgen` takes input from microprogram specification which is a readable text that a human programmer wrote. `Mgen` is a simple macro processor that substitutes symbolic names with numeric values (set microprogram bits).

The output is in the form :

```
nn
aaaa xxxxxxxxxxxxxxxxxx
....
```

where `nn` is the number of microword, `aaaa` is address and `xxxxx...` is the microprogram bit. `xxx...` begins at the column 5.

Input to `mgen` is in a simple form as follows :

```
.w N                        // width of microword N bits
.a B E                      // bit position of Goto field, B start, E
end
.s                          // start symbolic name section
```

```
name bit                          // "name" is the signal at "bit" position
...
.m                                // start microprogram section
:label name name ... /label ;     // each microword
 ...
.e                                // end of microprogram spec.
```

Within the microprogram section the label begin with "`:`" and the "`name`" is the name of signal (to be translated in to a number). The symbol `/label` destinates the label in Goto field. Each microword (a line of microprogram) must ends with "`;`".

Example The microprogram for S1 from the file "mspec.txt" is illustrated (comment shows here for explanation, no comments are allowed in mspec.txt).

```
.w 31                             // width 31 bits
.a 26 30                          // Goto start at bit 26 end at 30
.s                                // symbol section
dr 0                              // dest R bit 0
dpc 1                             // dest PC bit 1
...
sub 17                            // alu sub bit 17
add1 18
mrd 19                            // memory read bit 19
mwr 20
pc+1 21
u 22                              // Cond uncond bit 22
mrdy 23
testcc 24
decode 25
.m                                // microprogram section
:ifetch dmar spc ;                // <ifetch> MAR = PC
:w0 mrd mrdy /w0 ;                // MDR = M[MAR]; MREAD MRDY w0
dir smdr pc+1 decode ;            // IR = MDR; PC = PC + 1 DECODE
:load dmar sir:ads ;              // <load> MAR = IR:ADS
:w1 mrd mrdy /w1 ;
dr smdr ir:r0 u /ifetch ;
...
.e                                // end
```

This is the output (from mpgm.txt)

```
29
0 00010010000000000000000000000000
1 00000000000000000000100010000001
2 00100000010000000000001000100000
3 00010001000000000000000000000000
4 00000000000000000000100010000100
5 10000000010100000000000100000000
6 00010001000000000000000000000000
....
27 01000001000000000000000100000000
28 01000100000010000000000100000000
```

S1m microprogram simulator reads this microprogram (mpgm.txt) to instantiate its micromemory. S1m runs the same machine code program as S1, such as the program sum in "in.obj" which performs sum(a[0]..a[n]). The "in.obj" executed 1109 instructions 6054 clocks with CPI = 5.46

Chapter 6

Pipeline

The principle of pipeline is to overlap the operation of various functional units therefore reduce their idle time. Pipeline is one of the very first technique to increase the performance invented since the early days of computer. This chapter explores the technique of pipeline operations. The emphasise is on the instruction pipeline. Many techniques to improve the performance of pipeline are introduced. One concrete design based on S1 machine is illustrated.

Instruction pipeline

Consider one important cycle in the working of a processor, executing an instruction. Assume the cycle is broken into 5 stages:

1. instruction fetch IF
2. instruction decode ID
3. operand fetch OF
4. execute EX
5. write back WB

To illustrate the overlapping of operations let see some simple example. A processor executes three instructions i1, i2, i3. Each instruction takes 5 steps:

```
12345,12345,12345
```

The horizontal axis is time, if each stage takes 1 unit time (it is not necessary that each stage takes equal time but for simplicity we assume a fixed cycle pipeline stage) total time is 15 units. If we arrange three instructions such that they can be overlapped:

```
i1: 12345
i2:  12345
i3:   12345
```

The total time is just 7 units. At any time, each stage works on different instructions. The pipeline after 3 clocks can be viewed like this:

```
1  2  3  4  5
i3 i2 i1 -  -
```

Speedup

In an ideal case, the pipeline always be fully used (the number of instruction is large) hence the speedup is

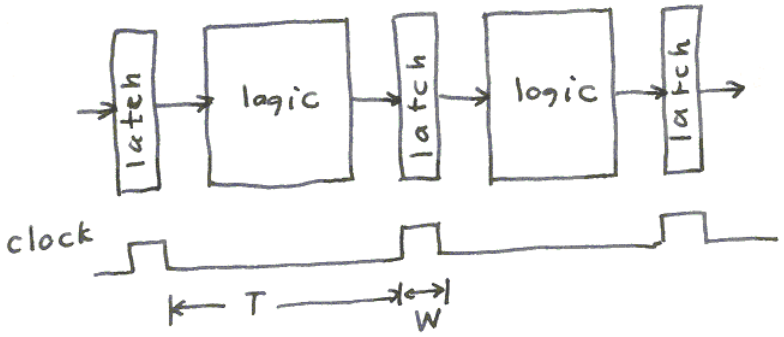
execution time without pipe / execution time with pipe

execution time with pipe = execution time without pipe/ no. of stage in pipe

speedup = no. of stage in pipe

How a pipeline is implemented

Each stage composed of a functional unit follows by a latch. All latches are synchronised.



T = time for logic to computer
W = time for latch

Figure 6.1 basic pipeline clock

As we have seen, the number of stage determines the speedup. What is the limit of the number of stage? When a task is divided into several stages which enable overlap operations total time for executing tasks can be shorten but the delay (caused by the setting time of latch) of each stage remains constant. This delay

time is the limit of speedup. Therefore the latch circuits must be very fast. Amdahl's law can explain this limit.

Stall of pipeline

If there are enough instructions to start execution every cycle and they continue until completion without interruption, the pipeline will be fully used and the speedup attains theoretical maximum. However, a pipeline is not always filled. When the flow of pipeline is interrupted, all stages before interruption are stopped (locked). This is called the pipeline *stall*. The rest of the pipeline can continue to function. There are three causes of stall:

1. structural hazard
2. data hazard
3. control hazard

Structural hazard

It is caused by conflicting or lacking of resources, such as when two stages of a pipeline accessing the same functional unit. Because the required resource is busy the pipeline must be stalled. (more about this in the section pipeline floating-point unit)

Data hazard

It is caused by dependency of data in a sequence of instructions as illustrated by this example.

```
i1: R1 = R2 + R3
i2: R4 = R1 + R5
```

The dependency of data is on R1. R1 in i1 must be updated before R1 in i2 is read. Because of overlapping operation, R1 in i1 is updated at WB stage but R1 in i2 is read in OF stage. The time diagram below shows when this situation occurs. x denotes the stage where hazard occurs.

```
              x
i1: 1 2 3 4 5
            x
i2:   1 2 3 4 5
```

Hence the pipeline must be stalled (i2 wait) until R1 is updated (in i1). There are four possibilities of data hazard:

1. Read After Read (RAR)
2. Read After Write (RAW)
3. Write After Read (WAR)
4. Write After Write (WAW)

The above situation is called **Read After Write (RAW)** hazard.

Write After Read (WAR)

This is the situation where the operand will be updated by the next instruction while it is being read in the current instruction. To avoid the incorrect result, the reading operation of the current instruction must be done before the writing operation of the next instruction. The example below shows the dependency on R2.

```
i1: R1 = R2 + R3
i2: R2 = R4 + R5
```

However, in this pipeline design the WB stage comes later than the OF, there is no WAR hazard as shown is the time diagram below. x denotes when R2 is read, y denotes when R2 is written.

```
          x
i1: 1 2 3 4 5
                y
i2:   1 2 3 4 5
```

Write after Write (WAW)

```
i1: R1 = R2 + R3
i2: R1 = R4 + R5
```

If R1 in i2 is updated before R1 in i1, the result will be in the wrong order. This hazard is presented in pipelines that have write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled, as in the superscalar design, which is called out-of-order execution).

There is no hazard for **Read After Read** dependency (without writing to the register, there is no data hazard).

Hazard detection

The hazard can be detected when two instructions want to use the same resources. Suppose instruction *i* is about to be issued and a previous instruction *j* is in the instruction pipeline. A RAW hazard exists on register *p* if *i* reads *p* ($Rregs_i$) and *p* will be written by *j* ($Wregs_j$). This hazard can be detected using a record of pending writes for all instructions in the pipe and compare with operand registers of the current instruction. When an instruction is issued, reserve its result register. On the completion of the operation, remove its write reservation.

A WAW hazard exists on the register *p* if *p* is-in $Wregs_i$ and $Wregs_j$.

A WAR hazard exists on the register *p* if *p* is-in $Wregs_i$ and $Rregs_j$.

Control hazard

It is caused by the transfer of control (jump). When the transfer occurs, the instruction that has already been fetched and decoded may not be the correct instruction as the destination address may be unknown at the time of the next instruction fetch. The pipeline must be flushed and start fetching the designated instruction. The example below shows there is stall, wasted three instructions that are already in the pipeline. The wasted instructions are in bolded face, x denotes when the destination of jump is known and i* is the designated instruction.

```
            x
i1: 1 2 3 4 5
i2:   1 2 3
i3:     1 2
i4:       1
i*:         1 2 3 4 5
```

Managing pipeline

A stall causes the pipeline performance to degrade. To improve the performance, the stall must be reduced or avoid. The pipeline speedup with stall can be calculated from the following relations:

ideal CPI = CPI without pipeline / pipeline depth

CPI with pipeline = ideal CPI + stall

pipeline speedup = ideal CPI × Pipeline depth / (ideal CPI + stall)

There are several techniques to reduce the stall. For data hazard, the register forwarding is very effective. For control hazard, the branch-prediction, branch target buffer and delay branch are the techniques that are widely used.

Register Forwarding

To reduce stall caused by data hazard, use register forwarding (or bypass) to handle RAW. The bypass unit makes use of the temporary result by forwarding it to the next instruction. This eliminates the stall because the next instruction can access the value without waiting to get it from the register bank.

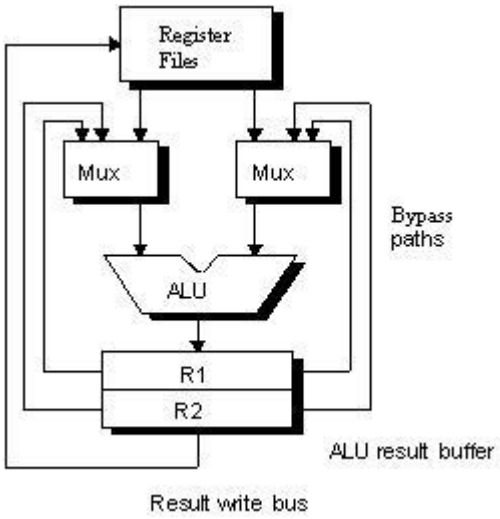


Figure 6.2 the ALU with register forwarding

```
          y x
i1: 1 2 3 4 5
          z
i2:   1 2 3 4 5
```

At stage 5 (WB) of i1 (x), the i1 will write it result back to the register file, while the i2 wants to read its operand at stage 3 OF (z). This caused RAW. Using the bypass unit (see Figure 6.2), the result of i1, which is already available at stage 4 (y) can be used by the next instruction, i2. The bypass unit checks if the result is in the buffer then select to use that result instead of reading from register file. To make it possible for i1 to write and i2 to read at the same cycle, the cycle is divided into 2 parts: the front and the back (see the next diagram). Each occurs at half cycle. The result must be produced by the front half and used by the back half. The i1 EX stage produces a result at the front half of clock cycle (y at 4F) and i2 OF stage reads its operand at the back half (x at 3B)

```
    y
i1: 4F 4B
       x
i2: 3F 3B
```

The next question is how deep the buffer in the bypass unit should be. Consider a sequence of instructions where R1 has data hazard.

```
i1: R1 = R2 + R3
i2: R4 = R1 - R5
i3: R6 = R1 + R7
i4: R8 = R1 - R9
```

```
          y x
i1: 1 2 3 4 5
          x
i2:   1 2 3 4 5
            x
i3:     1 2 3 4 5
i4:       1 2 3 4 5
```

With a bypass unit, i1 only affect i2. There is also hazard between i1 and i3, to forward the result from i1 to i3 required a buffer of depth two. There is no hazard between i1 and i4.

There are several methods to reduce stalls caused by control hazard: prefetch both next and target addresses, use branch prediction, and use delay branch.

Branch prediction

We can predict the current branch (taken/not_taken) one way and back up if the decision is turn out to be wrong, for example, predicting that the branch is always taken. It takes 1 clock per branch instruction if the prediction is right and 2 clocks if the prediction is wrong. The chance of being right is 50%. However, the static scheme always becomes very ineffective in some case. We can use the history of the previous branch to predict the current branch to allow the prediction to be more dynamic. Using 1 bit of history will increase the chance of being right to 90%. The prediction must be made at the decode stage of the current instruction (x) as the next instruction must be fetched.

```
                x
branch i1: 1 2 3 4 5
       i2:   1 2 3 4 5
```

The simplest scheme is the *branch history table.* The BHT contains the history bit (taken/not_taken) and is indexed by the lower bits of the current program counter. With one bit of history the rule to decide is simply *if the previous branch is taken then fetch from this branch target*. With this scheme the prediction will be wrong twice, one when enter the loop and the other when leave the loop. To improve the accuracy of the prediction, the state of history bit can be increased. Using 2 bits history with four states, the rule can be *if wrong prediction twice then change prediction to the other way.* Figure 6.3 shows one possible arrangement for the prediction states with 2 bits.

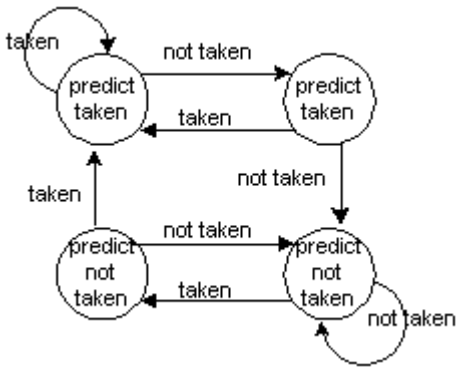


Figure 6.3 Two-bit branch prediction

Branch-target-buffer

The prediction by BHT is not very accurate, as the information stored in BHT is limited. The indexing by the lower bits of PC can be improved. To improve the accuracy of prediction more information about the address of the jump instruction and the address of the destination of jump is required. This information plus the history bit(s) are stored in "branch-target buffer" (BTB). We need to know what address to fetch by the end of IF. That is, we need to know what next PC should be (even the newly fetch instruction is not yet decoded, so we don't even know if that instruction is a branch or not). Therefore we are predicting the next instruction address and will send it out (next instruction fetch) before decoding the instruction. If the instruction is a branch and we know what the next PC is, we can have zero stall on branch. The main idea is to use a cache memory to store PC of instruction fetched and its predicted next PC and history bit. The steps are shown in the figure below.

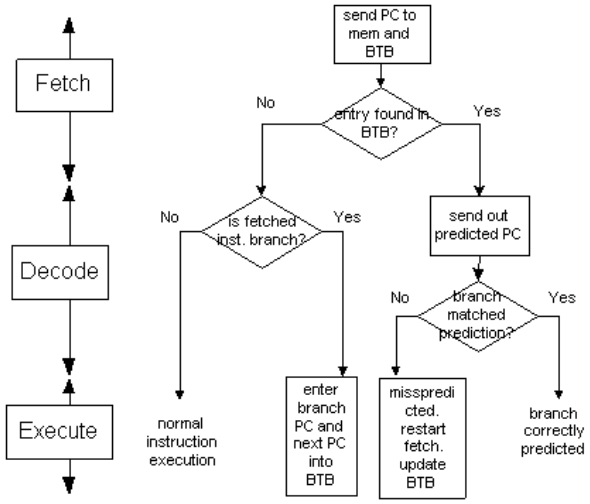


Figure 6.4 Steps in handling branch prediction

Delay branch

Another way to reduce the stall cycle caused by the branch is to change the semantic of the jump instruction. By doing the actual jump in the next cycle (delayed), there will be one free time slot after the jump instruction. The principle of this method is to use the stall cycle to do useful work by moving

some instruction to be executed in the stall cycle. The jump instruction is "delayed" i.e. caused the actual jump by some amount of clock (equal to stall cycle). Some instruction can be scheduled to be executed in that "delay slot". The instruction can be taken from the above, from the target, and from the fall through.

from above

The delay slot is filled by the instruction "above" the branch instruction (the symbol < > denotes the delay slot).

```
i1                  jmp i2
jmp i2     ==>      <i1>
< >                 i2
i2
```

from target

The delay slot is filled by the instruction at the destination. The following example shows the jump backward (usually at the end of loop). The target address (instruction i1) is moved to the delay slot and the jump destination is changed to i2.

```
i1                  i1
i2                  i2
...                 ...
jmp i1     ==>      jmp i2
< >                 <i1>
i3                  i3
```

from fall through

The delay slot is filled with the instruction from the target address, similar to "from target" but the direction of branch is forward. The jump address is changed to the next instruction after the target (i.e. fall through).

```
i1                  i1
jmp i2              jmp i3
< >        ==>      <i2>
i2                  i3
i3
```

There are several considerations which choice of instruction scheduling is suitable. Using *from target*, branch taken should be of high probability. Using

from fall through, branch not taken is of high probability. The instruction in delay slot must be "safe", that is, when the prediction is wrong, executing this instruction should not change the machine state until the outcome of branch is definitely known (and hence knowing whether this instruction must be executed or not). One delay slot can be filled with useful instruction about 50% of the time. The rest can be filled with NOP. It takes 0 clock when an instruction can be found to be filled in the delay slot (this situation is true 50% of the time). However, if more instructions per cycle are issued, the delay slot becomes less useful.

Advanced Pipeline

In our previous discussion, each stage in the pipeline is executed in one cycle. For simple operations such as integer arithmetic this is possible. However, there are several operations especially floating-point arithmetic that takes many cycles to complete. To allow for such operations the depth of the pipeline can be increased. This is possible when all operations have the same number of stages. This may not be possible. It is more economical instead to design a pipeline to have multi-cycle in the execution stage, however the control for such pipeline will be complicated.

Pipeline of the floating-point unit

For floating-point operations the pipeline will required to operate in multiple cycle. We will examine a simplify cases of floating-point multiplication and addition and illustrate how a functional unit for such operation is design and control.

Assume the inputs are two normalized floating-point numbers, A and B. A floating-point multiplication carries out the following steps:

1. Add two exponents
2. Multiply two significands. This may takes several cycles to perform partial products and sum them.
3. Normalized the product
4. Rounding

The pipeline for the FP multiplier is shows in Fig 6.5.

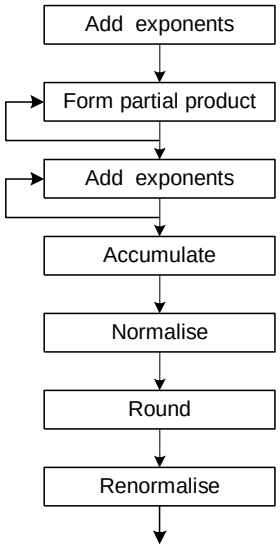


Figure 6.5 the FP multiplier pipeline

Similary for a floating-point addition, the following steps are performed:

1. subtract exponents and swap operands if necessary
2. shift the significand of B to the right
3. add significands
4. renormalized
5. round

The pipeline for a FP adder is shown in Fig 6.6. The FP multiplier and FP adder have several common operations. They can be combined to form a single function unit as shown in Fig. 6.7.

The control of this functional unit is complex because the collision exists if a new operation is admitted into the pipeline while one or more operations are in progress. Davidson [DAV91] developed "reservation table" that gives the timing information of the flow of data through the functional unit. The reservation table

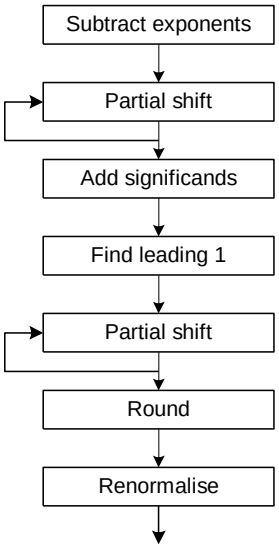


Figure 6.6 the pipeline of a FP adder

is derived directly from the pipeline design. It is used to decide when to launch an operation into the pipeline. Only one operation can use any unit at any time otherwise it is said to have collision.

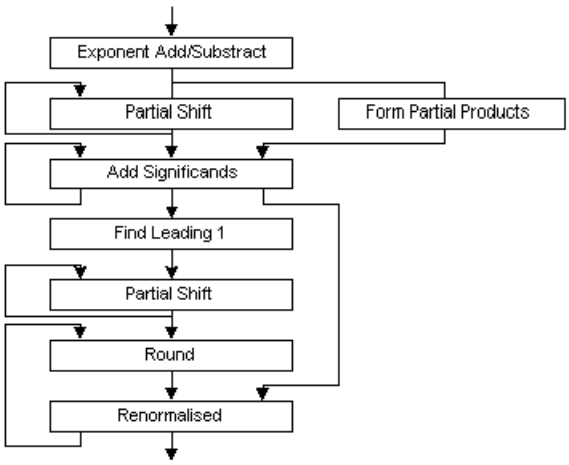


Figure 6.7 a pipelined floating-point combined adder/multiplier unit

Assume the multiplication of significands takes 2 cycles to form partial products and 2 more cycles to add the partial products. The reservation tables for FP multiplier and FP adder are shown in Fig. 6.8.

	1	2	3	4	5	6	7
exp add	x						
mul		x	x				
sig add			x	x			
renorm					x		x
round						x	
shift A							
lead 1							
shift B							

a) FP multiplier

	1	2	3	4	5	6	7	8	9
exp add	x								
mul									
sig add				x					
renorm									x
round								x	
shift A		x	x						
lead 1					x				
shift B						x	x		

b) FP adder

Figure 6.8 the reservation table for a) FP multiplier b) FP adder

If we launched two multiplications, one after another, the reservation table will look like Fig.6.9. x is the first multiplication and y is the second. When x and y occupy the same unit at the same time the collision occurs. Fig 6.9 shows that we cannot launch another multiplication one cycle after the first multiplication. The collision information is represented in a binary vector called "collision vector". Position *i* contains a bit that indicates whether a new operation can be launched *i* cycles after the first operation has been initiated. The collision vector for launching two multiplication in succession is 110000. It indicates that a new multiplication must be launched at least after two cycles after the first multiplication. For two FP addition, the collision vector is 10000000. We can

also determine the collision of a multiplication then addition and vice versa. For more advanced treatment of the pipeline control see Kogge [KOG81].

	1	2	3	4	5	6	7
exp add	x	y					
mul		x	**xy**				
sig add			x	**xy**	y		
renorm					x	y	x
round						x	y
shift A							
lead 1							
shift B							

Figure 6.9 the collision of two multiplications x and y

Pipeline of multiple functional units

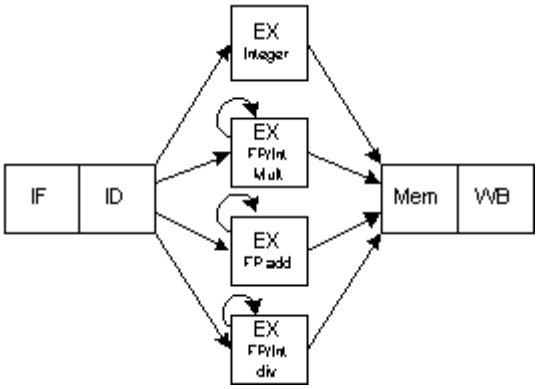


Figure 6.10 an example of multi step pipeline

Assume there are multiple functional units: FP adder, FP multiplier, FP divider, integer unit etc. and two separate register sets: FP and integer. This simplifies the pipeline control as it reduces hazard detection such as overlapping FP and Integer operations, except for load/store FP and movement between FP/Integer registers. Integer unit handles load/store to both register sets. Assume EX stage is repeated many times to do these operations. No other instruction using functional unit may issue until the previous instruction leaves EX. If an instruction cannot proceed to

the EX stage, the entire pipeline behind that instruction will be stalled (to avoid this stall, we need the capability to do out-of-order issue, the topic of next chapter). The following steps are required to issue a new floating-point instruction:

1. Check for structural hazard
2. Check for data hazard
3. Check for forwarding

Because all FP instructions require different execution time, this caused three complications:

1. contention for register access at WB stage,
2. WAR and WAW hazards and
3. interrupts. (we ignore interrupt).

FP load and FP operation can content for FP register on writes. This can be dealt with using priority scheme at the WB stage. The highest priority instruction can get access to register and all other instructions are stalled. A simple heuristic is *to give the longest latency instruction the highest priority***.** If all instructions read their registers at the same time there will be no WAR hazards. WAW hazards occur because the results can be written in different order. The instructions can be completed in a different order from the order in which they are issued.

Example

```
DIVF F0 F2 F4
SUBF F0 F8 F10
```

Assume DIVF takes longer than SUBF to complete. The SUBF will complete first and writes its result before the DIVF. This hazard must be detected and ensure that the result of executing instructions is correct.

S1 pipeline design

To illustrate the design of a pipelined CPU, S1 will be modified to be a pipelined machine. There are many factors that have to be considered: the number of stage of pipeline, the function of each stage of pipeline, and the behaviour of each instruction in the pipeline.

S1-pipe has 5 stages: **Fetch, Decode, Execute, Mem, Writeback.**

The fetch stage (F) reads the instruction from the memory. The decode stage (D) dispatches the instruction for appropriate sequences of operations. Every instructions have the same F,D stages. The execute state (X) performs operations. For ALU instructions, the ALU operations are done in this stage. For load/store instructions, the MAR and MDR are prepared. The memory stage (M) reads or writes the memory. The writeback stage (W) writes the result to the register.

Structure

Assume registers have 2 read ports and one write port. Every clock cycle, one instruction is fetched from the memory while other instruction may accesses the memory in M stage, therefore the memory must have 2 read ports and one write port (usually, the cache memory has this property). There are a number of additional registers that are used to store the information between the stages of pipeline (the state of pipeline): LMDR, SMDR, PCm, PCw, IRx, IRm, IRw and T1. The internal buses become 2 separate buses, R read bus and R write bus to allow concurrent operations to access registers and memory.

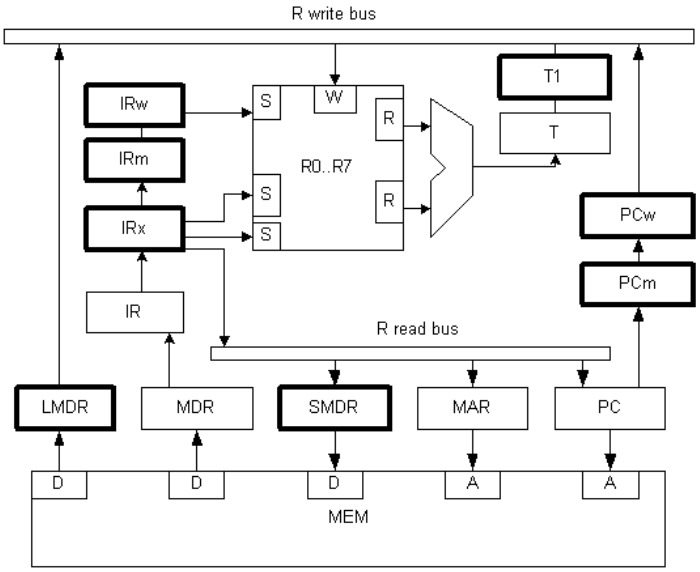


Figure 6.11 the structure of S1 pipeline

ISA

The number of stage of a pipe cannot be smaller than the longest instruction. For a 5-stage pipelined machine, at most an instruction must be completed in 5 clocks. The instructions that are too long (ret, push, pop) will be difficult if not impossible to pipeline therefore they are eliminated. To handle the subroutine call, the "jump and link register" and "jump register" are used. These two instructions can be completed within 5 clocks. Besides these two new instructions, all other instructions must retain their original semantics in the pipelined version (pipeline does not change the meaning of an instruction).

Microstep in the pipeline stages

```
Fstage : Mdr=M[Pc]; Pc=Pc+1
Dstage : Ir=Mdr
```

inst	Xstage	Mstage	Wstage
load	Mar=IRx:ads	Lmdr=M[Mar]	R[IRw:r]=Lmdr
store	Mar=IRx:ads ; Smdr=R[IRx:r]	M[Mar]=Smdr	
loadr	Mar=R[IRx:r1]	Lmdr=M[Mar]	R[IRw:r2]=Lmdr
storer	Mar=R[IRx:r2] ; Smdr=R[IRx:r1]	M[Mar]=Smdr	-
jmp	if CC PC=IRx:ads	-	-
jal	PCm=PC ; PC=IRx:ads	PCw=PCm	R[IRw:r1]=PCw
jr	if CC PC=R[IRx:r1]	-	-
mov	T=R[IRx:r1]	T1=T	R[IRw:r2]=T1
add	T=R[IRx:r1]+ R[IRx:r2]	T1=T	R[IRw:r1]=T1
inc	T=R[IRx:r1]+1	T1=T	R[IRw:r1]=T1
cmp	cmp(R[IRx:r1], R[IRx:r2])	-	-

Design considerations

1. shift register effect
2. conflict of use of resources in different stages

Shift register effect

When some data must be used in the later stage, it must be moved along the pipe and hence there must be intermediate registers (latches). The data moves along the pipe using the same principle as a shift register.

Example If we want to transfer the value of A from stage 1 to B in stage 3, we use an intermediate register, stage 1 to stage 2 and stage 2 to stage 3.

```
stage1   stage2    stage3
T = A,   T1 = T,   B = T1
```

The intermediate register is T1. The pair T = A, T1 = T execute at the same time. T is read (at stage 2) before it is written into (at stage 1).

Conflict of use of resources

All stages execute at the same time, hence the pipe cannot have one resource being used in two different stages. For example, MDR will be used to fetch the next instruction at Fstage all the time therefore MDR can not be used in any other stages.

Another example, LOAD instruction uses MDR = M[MAR] at Mstage and STORE uses MDR = R[IR:R1] at Xstage. MDR is used in two different stages (Mstage and Xstage). Moreover MDR is already been used in Fstage. Therefore, two new registers are assigned to avoid this conflict: Lmdr (load mdr), Smdr (store mdr). There are some situation which still has conflict such as an example below.

```
        Xstage     Mstage
storer  MAR = ...  M[MAR] = ...
```

MAR is on the left hand side (being written into), therefore there is a conflict in writing and reading MAR at the same time, but for a memory chip, we assume

the address can be change during the write memory cycle as long as it was hold for a certain time. PC is also having a conflict. At Fstage, PC = PC+1 but for JMP instruction PC = IR:ads. This situation can be remedied by using a multiplexor circuit for writing into PC and PC = IR:ads when the jump is taken (because it jumps and hence not using the next instruction in sequence). Fig. 6.12 shows the next address circuit according to this scheme.

```
        Fstage    Xstage
jmp     PC=PC+1   PC=IRx:ads
```

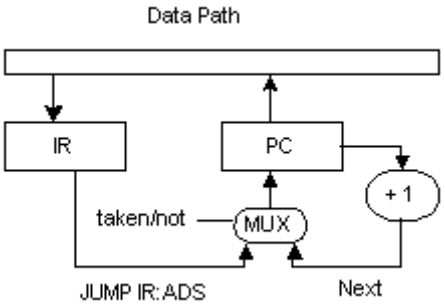


Figure 6.12 the next address circuit

How to assign each microstep into a stage?

The assignment is done according to the guideline that a resource can not appear in two different stages. Let us see the example from S1 pipeline. At Xstage, most operations have the form A = R[..], i.e. reading register file. At Wstage, operations have the form R[..] = A, i.e. writing register file. If we move R[IR:R1] = PCw from Wstage to Xstage there will be conflict on writing a register with R[..] = A of other instruction.

Performance evaluation

Running the sum array benchmark (sum a[0]..a[n] where n = 100) and using the following timing for S1 (Table 6.1).

Table 6.1 the timing of S1 (count ifetch as 2 clocks)

instruction	clock
load	5
store	5
jmp	3
jal	3
mov	4
loadr	5
storer	5
add	4
cmp	3
inc	4

The experiment is carried out to compare s1 and S1-pipe (S1-P) and S1-pipe with register forwarding (S1-PF) and S1-pipe with forwarding and delay branch (S1-PFD). The result is shown in Table 6.2.

Table 6.2 comparison of S1 and pipelined S1

CPU	inst	clock	stall	stall %	CPI	speedup	P:PF	PF:PFD
S1	1110	4642	–	–	4.18	–	–	–
S1-P	1112	3136	2024	64.5	2.82	0.48	–	–
S1-PF	1112	1304	202	15.4	1.18	2.54	1.39	–
S1-PFD	1112	1114	2	0.18	1.002	3.17	1.81	0.18

S1-P is faster than S1 48% and S1-PF is faster than S1 254%. S1-PF is faster than S1-P 139%. The maximum gain is obtained using forwarding. It reduced stall cycle from 64.5% to 15.4%. By using delay branch this can be reduced further but the gain is not as much as using forwarding.

Summary

The pipeline overlaps the operations and reduces the idle time in functional units. This chapter illustrates only one kind of pipelining, the instruction execution pipeline. There are others such as pipeline in accessing memory, pipeline in

cache memory controller [TCH98], pipeline in virtual address calculation. The stall in pipeline caused serious performance degradation. Several techniques to reduce the stall have been demonstrated such as register forwarding and branch prediction. There are many other techniques, several of which will be studied in the next chapter. The reduction of stall caused by branch is still a very much active area of research, for example see [USS97].

References

[DAV71] Davidson, E., "The design and control of pipelined function generators." Proc. of the 1971 Int. Conf. on Systems, Networks, and Computers, Oaxtepec, Mexico, 1971.

[KOG81] Kogge, P., The architecture of pipelined computers, McGraw-Hill, 1981.

[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.

[STO93] Stone, H., High performance Computer architecture, McGraw-Hill, 1993.

[USS97] Uht, A K., Sindagi, V., Somanathan, S., "Branch Effect Reduction Techniques", IEEE COMPUTER, Vol. 30, No. 5: MAY 1997, pp. 71-81.

Chapter 7

Instruction Level Parallelism

The use of simple pipeline for instruction execution and a more complex pipeline for floating-point operations is a way to increase "instruction level parallelism" since the instructions can be evaluated in parallel. In this chapter we will learn more about other techniques to increase performance through instruction level parallelism. The technique that is based on software (a compiler) which rearranges instructions to reduce the number of stall is called *static scheduling* as the scheduling is done at compile-time. The technique that uses hardware to issue many instructions concurrently is called *dynamic scheduling* as the scheduling is done at run-time. We examine three static scheduling techniques: register optimization, register renaming and loop unrolling. We study five architectural features to improve instruction level parallelism: scoreboard, Tomasulo, superscalar, superpipeline, and very-long-instruction-word (VLIW).

Static scheduling

If a compiler knows about architectural features of the target machine, the compiler can analyse a source program and generate sequences of instructions that minimise the number of stall. The technique based on software is powerful as there are plenty of resources available to perform complex analysis. The limitation is that there are many events that are not knowable at compile-time hence it is impossible to schedule many run-time dependent sequences of instructions. *Register optimization* is used to minimise the stall caused by load and store operations. *Register renaming* replaces the registers in conflict to eliminate hazards. *Loop unrolling* is used to schedule instructions across basic blocks. A basic block is defined as a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.

Register optimization

To reduce the number of stalls caused by load and store operations, the compiler keeps the operands for as many computations as possible in register rather than in memory. The approach is as follows. Each program quantity to be kept in a register is assigned a virtual register. The compiler then maps this unlimited number of virtual registers in to a fixed number of real registers. Virtual registers whose usages are not overlap can share the same real register. If there are more quantities than the number of real registers, some quantities are assigned to memory locations. The optimization task is to decide which quantities are to be assigned to registers at any given point in a program. This problem is known as graph coloring. The algorithm to solve graph coloring, which is NP-complete, can be used to allocate registers [CHA82].

Register renaming

This technique assigns idle registers to serve in place of program specified registers in order to avoid conflicts that could stall pipeline. For example, the sequence of instructions:

i1: **R3** = R3 × R5
i2: R4 = **R3** + 1
i3: **R3** = R5 + 1
i4: R7 = **R3** × R4

There are the following data hazards : i1-i2 RAW , i1-i3 WAW, i2-i3 WAR, i3-i4 RAW. By replacing the registers in conflict with different registers the conflict of resources can be avoid. The following example shows how renaming is used.

i1: R3b = R3a × R5a
i2: R4b = R3b + 1
i3: R3c = R5a + 1
i4: R7b = R3c × R4b

The renaming can be achieved by a compiler or by using hardware. The hardware approach will be discussed in the section Tomasulo.

Loop Unrolling

The simplest way to increase instruction parallelism is to exploit the iterative nature of the loop. The technique work by unrolling the loop either by the compiler or by the hardware. Here we examine the loop unrolling by software. To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

Let us use an example to illustrate how a compiler unrolls loops to improve the instruction level parallelism. The example is a program to multiply two matrices. This is a typical triple loop for matrix multiplication. Assume integer numbers. The matrices have size $N \times N$.

```
1  for(i=0; i<N; i++)
2    for(j=0; j<N; j++) {
3      c[i][j] = 0;
4      for(k=0; k<N; k++)
5        c[i][j] += a[i][k] * b[k][j];
6    }
```

Assume we have a 32-bit version of S1. The S1-32 has the element size equal one word (word addressable where a word is 32 bits). S1-32 instructions have the 3-operand format where `op r1 r2 r3` means `r1 = r2 op r3`. Its addressing mode includes register index with displacement in the format `disp(base + index)`. The load/store instructions have 1 clock stall and all other instructions take 1 clock to complete. Assume each element is 32 bits and the matrix is stored in row-major, i.e. the arrangement in the memory is as follows: a_{11}, a_{12}, a_{13}, … a_{1n}, a_{21}, a_{22}, ... , a_{n1}, ..., a_{nn}. The following assembly program will multiply two matrices $C = A \times B$.

```
clk
1      loop: ld1 aik 0(Rba + Rik)
2            stall
3            ld2 bkj 0(Rbb + Rkj)
4            stall
5            mul3 Rt aik bkj
6            add4 cij cij Rt
7            add5 Rik Rik 1
8            add6 Rkj Rkj N
9            cmp7 Rik Rend
10           jnz8 loop
11           st9 0(Rbc + Rij) cij
```

Explanation of variables:
register a_{ik}, b_{kj} hold the current value of a and b.
register R_{ba}, R_{bb}, R_{bc} hold the base address of the matrices A,B,C
register R_{ik}, R_{kj}, R_{ij} hold the index to the a_{ik}, b_{kj}, c_{ij}
register R_t is temporary, R_{end} holds the address of the last element of the matrix A.

Each instruction has suffix to help understanding the rearranging of the instructions. This program takes 10 clocks to computer one element (not counting the last store). Load instructions stall one clock each (ld_1, ld_2).

Schedule loop to reduce stall

Instead of wasting the stall cycle because of `ld` instruction, we can issue other instruction to perform a useful work. We rearrange the program (code scheduling) to fill the stall cycle as follows :

```
1     loop: ld1 aik 0(Rba + Rik)
2           add5 Rik Rik 1
3           ld2 bkj 0(Rbb + Rkj)
4           add6 Rkj Rkj N
5           mul3 Rt aik bkj
6           add4 cij cij Rt
7           cmp7 Rik Rend
8           jnz8 loop
9           st9 0(Rbc + Rij) cij
```

Now the program computes one element in 8 clocks.

Loop unrolling can eliminate branch

To reduce the overhead of control flow stall (increment pointers and jump) the loop can be unrolled. The following example unrolls the loop once, using the offset in `ld/st` instruction to index the variables . Therefore two instructions per loop are eliminated (add_5, add_6).

```
1     loop: ld1 aik 0(Rba + Rik)
2           stall
3           ld2 bkj 0(Rbb + Rkj)
4           stall
```

```
5           mul3 Rt aik bkj
6           add4 cij cij Rt
7           ld10 aik 1(Rba + Rik)
8           add5 Rik Rik 1
9           ld11 bkj 1(Rbb + Rkj)
10          add6 Rkj Rkj N
11          mul12 Rt aik bkj
12          add13 cij cij Rt
13          cmp7 Rik Rend
14          jnz8 loop
15          st9 0(Rbc + Rij) cij
```

This program takes 14/2 = 7 clocks per element.

In our example, unrolling loops improve the speed from 10 clocks per element to 8 clocks to 7 clocks but increase the size of program substantially. To exploit instruction level parallelism it is important to determine which instructions can be executed in parallel (assuming pipeline has sufficient resources). If two instructions are dependent they are not parallel. Instructions that can be reordered are parallel and vice versa. The loop level parallelism is analysed at the source code. The analysis involves determining what dependencies exist among the operands in the loop across the iterations of the loop.

Dynamic scheduling in pipeline

In the previous chapter, the pipeline fetches and issues an instruction unless there is a data dependence between an instruction already in the pipeline and the fetched instruction. When data hazard occurs the pipeline is stalled. This is called *static scheduling*. The stall caused by data hazard can be reduced by forwarding the result using a *bypass unit*. This section introduces more hardware scheme to reduce the stalls, this is called *dynamic scheduling* as it can detect the dependencies at the run-time.

All the previous pipeline technique that we described use *in-order instruction issue*. If an instruction is stalled in the pipeline, no later instructions can proceed. When there are multiple functional units, these units could be idle. For example,

```
DIVF F0 F2 F4
ADDF F10 F0 F8
SUBF F8 F8 F14
```

The `SUBF` cannot be issue because the dependence of `ADDF` on `DIVF` (RAW on F0). Yet, the `SUBF` does not depend on any instruction in the pipeline. If an

instruction can be executed *out-of-order* this stall can be eliminated. We can check data hazards in the ID stage. In order to let an instruction start its execution as soon as its operands is available, the instruction issuing process must be separated from the hazard checking. The pipeline will do *out-of-order execution* which implies *out-of-order completion*. For the simple pipeline in the last chapter, data hazards and structural hazards are checked during the instruction decode stage. To allow out-of-order execution the issue process must be split into two parts: 1) checking and waiting until no structural hazard 2) and then read operands. The distinction must be made between the *beginning of execution* of an instruction and the *completion*. Between these two time, an instruction is in *execution*. Using multiple functional units and pipelines, multiple instructions can be in execution at the same time. We introduce three techniques to do dynamic scheduling: scoreboard, Tomasulo and superscalar.

Scoreboard

Scoreboard is a hardware technique that enables instructions to be out-of-order executed when there are sufficient resources and no data dependencies. It is named after the CDC6600 scoreboard [THO70] (first delivered in 1964 and was considered by many to be the first supercomputer). The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle by executing an instruction as early as possible. This can be achieve only when there are no structural hazard, i.e. there are sufficient number of resources. The scoreboard is responsible for instruction issue and execution. It does all hazard detection.

Every instruction goes through the scoreboard which keeps all information necessary to detect all hazards. The scoreboard determines when an instruction can read its operands and begin execution. The scoreboard controls when an instruction can write its result into the destination register. All hazard detection and resolution is centralised in the scoreboard.

To illustrate the working of scoreboard we will use S1 with scoreboard (S1s). Assume S1s has 2 FP multipliers, one FP divide, one FP add and one integer unit. The integer unit handles all load/store, memory references, branches and integer operations. Each instruction goes through 4 steps, which replaces the pipeline stages, as follows:

1. **Issue**, when there is no structural hazard the instruction is issued. Scoreboard updates its internal data structure. It guarantees that there is no WAW.

2. **Read operands**, the scoreboard monitors source operands. If
   2.1 no active instruction is going to write to it.
   2.2 no active functional unit is writing to the register containing the operands.
   Then when the source operands are available, the scoreboard tells the functional unit to read its operands and begin execution. RAW is resolved and instructions may be executed out-of-order.
3. **Execution**, a functional unit begins execution. It notifies the scoreboard when the result is ready.
4. **Write result**, when the functional unit has completed execution, the scoreboard checks or WAR hazards. An instruction is not allowed to write its results when
   4.1 there is an instruction that has not read its operands.
   4.2 one of the operands is the same register as the result of the completing instruction.
   4.3 the other operand was the result of an earlier instruction.

The scoreboard controls the execution of an instruction by communicating with the functional units.

Example, a scoreboard of S1s controls the execution of the following sequence of instructions:

```
LF F6 34(R2)
LF F2 45(R3)
MULTF F0 F2 F4
SUBF F8 F6 F2
DIVF F10 F0 F6
ADDF F6 F8 F2
```

The scoreboard has three parts:

1. Instruction status
2. Functional unit status, each FU has the following fields
   2.1 Busy
   2.2 Op, instruction to be performed
   2.3 Dest , destination register
   2.4 Src1, Src2, source registers
   2.5 P1, P2, number of units producing Src1, Src2
   2.6 R1, R2, ready flags for Src1, Src2; they are reset when new values are read so the scoreboard knows that the source operand has been read.
3. Register result status, which FU will write register.

Assuming the execution of the floating-point functional units are: add is 2 clocks, multiply is 10 clocks and divide is 40 clocks. Each instruction that has issued, has an entry in the instruction status table. Once the instruction issues, the record of its operands is kept in the functional unit status table. See the figure 7.1, the instruction status says that

1) the first `LF` has completed
2) the second `LF` has completed but has not yet written its result.
3) the `MULTF`, `SUBF` and `DIVF` have issued but are stalled, waiting for their operands.

The functional unit status says that

1) the first multiplier unit is waiting for the integer unit. (RAW on F2).
2) the add unit is waiting for the integer unit. (RAW on F2).
3) the divide unit is waiting for the first multiplier unit. (RAW on F0).
4) the `ADDF` is stalled due to structural hazard (FU Add is in used by `SUBF`).

Instruction status

Instruction	Issue	Read Operands	Ex complete	Write result
LF F6 34(R2)	X	X	X	X
LF F2 45(R3)	X	X	X	
MULTF F0 F2 F4	X			
SUBF F8 F6 F2	X			
DIVF F10 F0 F6	X			
ADDF F6 F8 F2				

Functional unit status

FU no	Name	Busy	Op	Dest	Src1	Src2	P1	P2	R1	R2
1	Integer	Y	loadf	F2	R3				N	N
2	Mult1	Y	multf	F0	F2	F4	1		N	Y
3	Mult2	N								
4	Add	Y	subf	F8	F6	F2		1	Y	N
5	Divide	Y	divf	F10	F0	F6	2		N	Y

Register result status

	F0	F2	F4	F6	F8	F10	F12	---	F30
FU no	2	1			4	5			

Figure 7.1 The scoreboard after issuing the first five instructions.

Now assume the MULF and DIVF are proceeded and ready to write results. There are RAW on

1) the second LF to MULTF and SUBF (on F2)
2) MULTF to DIVF (on F0)
3) SUBF to ADDF (on F8)

There is a WAR between DIVF and ADDF on F6. There is a structural hazard on FU add for ADDF. The DIVF has not yet read its operands. The ADDF has read its operands and is in execution, it was waiting for SUBF (structural hazard). The ADDF cannot write its results because of WAR on F6.

Instruction status

Instruction	Issue	Read Operands	Ex complete	Write result
LF F6 34(R2)	X	X	X	X
LF F2 45(R3)	X	X	X	X
MULTF F0 F2 F4	X	X	X	
SUBF F8 F6 F2	X	X	X	X
DIVF F10 F0 F6	X			
ADDF F6 F8 F2	X	X	X	

Functional unit status

FU no	Name	Busy	Op	Dest	Src1	Src2	P1	P2	R1	R2
1	Integer	N								
2	Mult1	Y	multf	F0	F2	F4			N	N
3	Mult2	N								
4	Add	Y	addf	F6	F8	F2			N	N
5	Divide	Y	divf	F10	F0	F6	2		N	Y

Register result status

	F0	F2	F4	F6	F8	F10	F12	---	F30
FU no	2			4		5			

Figure 7.2 The scoreboard when MULTF and DIVF are ready to write results.

Bookkeeping

The scoreboard records operand specifier information, such as register numbers. For example, it records the source registers when an instruction is issued. Here is the summary of bookkeeping for each step in instruction execution.

Instruction issue

Check functional unit is not busy (functional units status) and dest is not waiting for the result (register result status)

- check busy field in FU = yes
- check op field in FU = opcode
- fill FU : Dest Src1 Src2
- fill P1 P2 with the register result of Src1 Src2
- check R1 R2 = not P1, not P2
- write the name of FU to register result

Read operands

Wait for R1 R2 until ready, read operands

- set R1, R2 = No
- P1, P2 = 0

Execution complete

Wait until functional unit done

Write result

Check WAR hazard

- when other instruction has this instruction Dest as Src1 or Src2
  for all f : Src1(f), Src2(f) != Dest(FU) AND
- when other instruction has written the register R1, R2
  R1 = Yes or R2 = Yes

Wait until no harzard , set ready flag
for all f :

- if P1(f) = FU then R1(f) = Yes
- if P2(f) = FU then R2(f) = Yes
- reset register result
- reset busy field of FU

The next section describes a technique called *register renaming* that eliminates name dependencies so as to avoid WAR and WAW hazards.

Tomasulo

Another dynamic scheduling technique similar to scoreboard is Tomasulo. This technique was invented by Robert Tomasulo in 1967 [TOM67] for the IBM 360/91 floating-point unit [AND67]. The key concept is the *renaming of registers* to avoid WAR and WAW hazards. This function is provided by the *reservation stations*. A reservation station fetches and buffers an operand as soon as it is available, pending instructions designate the reservation station that will provide their input. When successive writes to a register appear, only the last one is actually used to update the register. As instructions are issued, the register name for pending operands are renamed to the names of the reservation station. This is the main difference between scoreboard and Tomasulo's algorithm. There can be more reservation stations than real registers, the technique can eliminate hazards that could not be eliminated by a compiler.

Two other differences between scoreboard and Tomasulo are: first, hazard detection and execution control are *distributed* by each reservation station (in scoreboard it is centralised), second, *results are passed directly* to functional units rather than through registers. A common result bus allows all units waiting for an operand to be loaded simultaneously, this is called the *common data bus* (CDB) (Fig.7.3).

The steps to execute an instruction:

1. **Issue**, Get an instruction from the queue, issue it if there is an empty reservation station, send the operands to the reservation station if they are in the registers. If the operand is a load or store, it can issue if there is an available buffer. If there is no empty reservation station or an empty buffer, then there is a structural hazard. This step also performs the renaming registers.
2. **Execute**, if operands are not yet available, monitor CDB waiting for the registers. When an operand is available, it is placed into the corresponding reservation station. When both operands are available, execute the operation. This step checks RAW hazards.
3. **Write result**, When the result is available, write it on CDB and from there into the registers and any reservation stations waiting for this result.

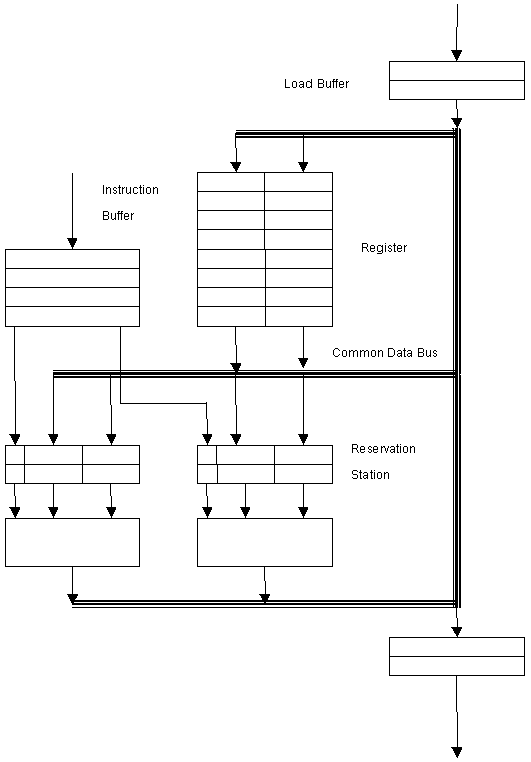


Figure 7.3 A CPU with two floating point functional units each with two reservation stations, and one load one store buffer

Although these steps are similar to those in the scoreboard there are three important differences. First, there is no checking for WAW and WAR hazards – they are eliminated by register renaming. Second, the CDB is used to broadcast the result instead of waiting for the registers. Third, the loads and stores are treated as functional units.

The next two sections examine how to improve the instruction level parallelism further by issuing multiple instructions in one clock cycle.

Superscalar

The term *superscalar* describes a computer implementation that improves performance by concurrent execution of scalar instructions (more than one

instruction per cycle) [JOH90]. Scalar processor is a processor that execute one instruction at a time. A superscalar processor allows concurrent execution of instructions *in the same* pipeline stage. A superscalar processor is a machine that is designed to improve the performance of the execution of scalar instructions as opposed to vector processors that operate on vectors.

The hazard affects a superscalar processor more than in a scalar processor as it prevents a greater amount of resources from being used. For example, when there is no instruction that is not dependent the processor will not execute any new instruction, this is called "zero-issue" cycle. During this cycle, the wide pipeline that can execute more than one instruction at a time is wasted.

The *instruction parallelism* of a program is a measure of the average number of instructions that a processor might be able to execute at the same time (given an unlimited resource). *Machine parallelism* of a processor is a measure of the ability of the processor to take advantage of the instruction-level parallelism. Machine parallelism is determined by the number of instructions that can be fetched and executed at the same time by the mechanisms that the processor uses to find independent instructions. To achieve performance, both machine parallelism and instruction parallelism are required [JOU89].

Instruction-issue refers to the process of initiating instruction execution in the processor's functional units. Instruction-issue policy affects performance because it determines the processor's "lookahead" capability; that is, the ability of the processor to examine instructions beyond the current point of execution in hopes of finding independent instructions to execute. There are three possible policies: in-order issue with in-order completion, in-order issue with out-of-order completion, and out-of-order issue with out-of-order completion. We examine each policy in turn using an example of a superscalar processor.

Assume a superscalar processor capable of fetching and decoding two instructions at a time, having three separate functional units and two writeback stages. There are six instructions being executed. The following constraints occur:

- i1 requires two cycles to execute
- i3 and i4 conflict for the same functional unit.
- i5 depends on the value produced by i4.
- i5 and i6 conflict for a functional unit.

In-order issue with in-order completion

The simplest policy is to issue instructions in exact program order and to write results in the same order. The figure 7.4 shows two instructions being in execution at once. The results are written back in the same order. The instruction issuing stalls where there is a conflict for a functional unit.

Decode		Execution			Writeback		clock
i1	i2						1
i3	i4	i1	i2				2
i3	i4	i1					3
	i4			i3	i1	i2	4
i5	i6			i4			5
	i6		i5		i3	i4	6
			i6				7
					i5	i6	8

Figure 7.4 a superscalar with in-order issue and in-order completion

In-order issue with out-of-order completion

With out-of-order completion, the number of instructions allowed to be in execution in the functional units is up to the total number of pipeline stages in all functional units. Instruction issuing is not stalled when a functional unit takes more than one cycle to compute a result. Therefore instructions may be complete out of order. The figure 7.5 shows i1 is completed out of order. Total time of this sequence is reduced to seven cycles.

Decode		Execution			Writeback		clock
i1	i2						1
i3	i4	i1	i2				2
i3	i4	i1			i2		3
	i4			i3	i1	i3	4
i5	i6			i4	i4		5
	i6		i5		i5		6
			i6		i6		7

Figure 7.5 a superscalar with in-order issue and out-of-order completion

Out-of-order completion yields higher performance than in-order completion, but requires more hardware than in-order completion. Dependency logic is more complex with out-of-order completion, because this logic checks data dependencies between decoded instructions and all instructions in all pipeline stages. Out-of-order completion improves the performance of long latency operations such as loads or floating-point operations.

Out-of-order issue with out-of-order completion

With in-order issue, the processor stops decoding whenever a decoded instruction creates conflict or dependency on the instruction in the pipeline. To be able to look ahead beyond the instruction with conflict or dependency, the processor must isolate the decoder from the execution stage, so that it continues to decode instruction regardless of whether they can be executed immediately. This is accomplished by a buffer between the decode and execute stages, called an *instruction window*. Instructions are issued from the window without regarding their program order but it is required that the program must behaves correctly.

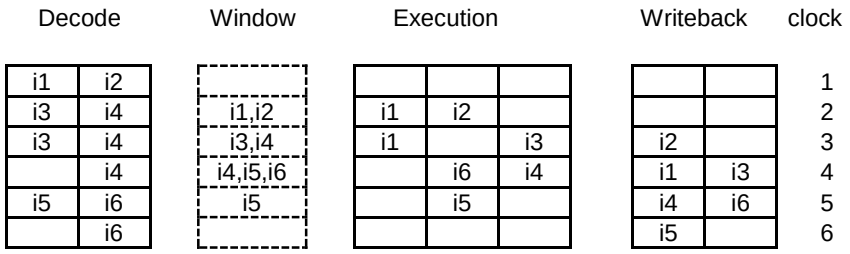


Figure 7.6 a superscalar with out-of-order issue and out-of-order completion

The instruction window is not an addition pipeline stage. It is a buffer that keeps information about instructions to be issued. The figure 7.6 shows that the processor discovers an independent instruction i6 and issues it out-of-order with i4. The total time is reduced to six cycles.

Superpipeline

In a fully-pipeline operation, one result will be produced every clock cycle. Therefore the cycle-per-instruction is one. To divide pipeline into more stages will result in a superpipeline. In a superpipelined processor, the stages are divided into substages [JOU89a]. The substages are clocked at a higher

frequency than the major stages. The processor can initiate an operation at each substage. This effectively reduces the processor cycle time. A superpipeline processor takes longer to generate all results than the superscalar processor for a given set of operations. On the other hand, some simple operations in the superscalar processor take a full cycle whereas the superpipelined processor can complete these operations sooner. For example, the superpipelined processor knows the result of the branch instruction sooner than the superscalar processor, thus reduces the impact of the control hazard. The superpipelining is appropriate when the cost of duplicating resources is high and the ability to control clock skew is good (as it is susceptible to clock skew). It is also appropriate for implementing a very high speed technology. Superpipelining presents no new design problems over pipelined processors.

Very long instruction word

A superscalar processor uses dynamic scheduling, e.g. the hardware controls the issue of instruction dynamically. For static scheduling, the very long instruction word (VLIW) architecture [FIS83] depends on a compiler to schedule concurrent instructions and rearranging them into a long instruction word [ELL87], typically 120-200 bits. A single instruction specifies more than one concurrent operation.

A VLIW processor can be visualised as a processor without instructions, just a processor that directly controls the functional units from its bit-pattern similar to the level of microprogram. A compiler performs scheduling of parallel execution. Since hardware can have multiple functional units we can schedule as many of them to execute concurrently. The limitation is on instruction parallelism. A basic block is defined to contain a sequence of code without branching, i.e. a straight line code. The number of instruction in a basic block is average about 10 lines of assembly. The number of instruction in a basic block must be enough to sustain parallel execution of functional units. One simple technique to increase the number of instruction in a basic block is loop unrolling. More advance technique required inter-block analysis, so called "trace scheduling". Trace scheduling is done by analysing the sequence of instruction executed. Trace scheduling will be discussed in the next section.

Suppose a VLIW processor has one load/store unit, two integer units and one branch unit. Assume the load/store unit can issue the next instruction before the first instruction is completed. The load/store delay is 1 cycle. Figure 7.7 shows scheduling of the code for matrix multiply on the VLIW processor.

clk	load-store	integer1	integer2	branch
1	ld_1 a $_{ik}$ 0(R $_{ba}$ + R $_{ik}$)			
2	ld_2 b $_{kj}$ 0(R $_{bb}$ + R $_{kj}$)			
3	ld_{10} a2 $_{ik}$ 1(R $_{ba}$ + R $_{ik}$)			
4	ld_{11} b2 $_{kj}$ 1(R $_{bb}$ + R $_{kj}$)	mul_3 R $_t$ a $_{ik}$ b $_{kj}$		
5		add_4 c $_{ij}$ c $_{ij}$ R $_t$	add_5 R $_{ik}$ R $_{ik}$ 1	
6		cmp_7 R$_3$ R $_{ik}$ R $_{end}$	mul_{12} R2 $_t$ a2 $_{ik}$ b2 $_{kj}$	
7		add_6 R $_{kj}$ R $_{kj}$ N	add_{13} c $_{ij}$ c $_{ij}$ R2 $_t$	jnz_8 R$_3$
8	st_9 0(R $_{bc}$ + R $_{ij}$) c $_{ij}$			

Figure 7.7 scheduling the matrix multiply code in a VLIW processor

To avoid data hazard, the operands a $_{ik}$, b $_{kj}$ at the instruction ld_{10}, ld_{11}, mul_{12} are renamed to a2 $_{ik}$, b2 $_{kj}$. This program takes 7/2 = 4.5 clocks per element assuming no branch delay. Compare this result to the 7 clocks per element achieved by the software loop unrolling.

Note on the use of flags

We did not use flags for conditional branch as the flags are "global" and in the concurrent issue of instructions it is very difficult predict the effect on flags. The instruction that sets flags and the instruction that tests the flag may not be easy to recognise when instruction scheduling "rearrange" the order of execution. One way to solve this problem is to make the setting and testing "local" by using a register to store the result, for example, the instruction "`cmp r1 r2 r3`" compares r2 and r3 and stores the result {–1, 0, 1} in r1 if r2 < r3, r2 = r3, r2 > r3 respectively. The conditional branch instruction takes the condition from a register, for example, "`jnz r1 ads`" means jump if r1 is not zero. This will avoid the conflict of using global flags by concurrent instructions and facilitate the instruction scheduling.

Trace scheduling

Trace scheduling extends loop unrolling with a technique for finding parallelism across conditional branches. It is consisted of two steps: trace selection and trace compaction. Trace selection finds the sequence, called "trace", to be put together. Loop unrolling is used to generate a long trace. Trace compaction packages the trace into a small number of wide instructions. There are two consideration when perform trace scheduling: 1) data dependency, 2) branch points. Data dependency forces partial order on operations and branch points impose constraint on moving code across the branches. Assume the following code:

```
1 a[i] = ...
2 if cond
3 then b[i] = ...
4 else d ...
5 c[i] = ...
```

Trace selection selects the sequence 1,2,3,5 as the probability of the branch to be taken is higher. The branch point 4 is called branch "out" of the trace and the branch from 4 to 5 is called branch "into" the trace.

Trace compaction tries to move 3 and 5 to the point before the branch 3, so that these operations can be packed into a wide instruction. In moving 3, the code in 4 will be affected if it used `b[]`, since moving 3 will change the value of `b[]`. Therefore, to move 3, 4 must not read 3. To move 5, `c[]` must be moved over 4, as 4 flow "into" the trace. This can be done by copy 5 but the check must be done similar to `b[]` to make sure the code can be moved. If 5 can be moved and the branch "out" is taken, 5 will be performed twice. This incurs penalty.

Loop unrolling, and trace scheduling aim at increasing the amount of instruction level parallelism that can be exploited by a processor issuing more than one instruction on every clock cycle.

Speculative Execution

Another technique to schedule instructions across branch is to execute the instructions "speculatively", i.e. the instructions are issued and executed but may not be "committed" to write their results. The results is committed, i.e. written back to registers, after the outcome of the condition in the branch is known. Speculative execution can improve the performance given that the resources are adequate. See the following example:

```
if cond == true
then a = b + 1
else a = c + 1
```

In normal execution

```
      test cond
      jump if false :2
      a = b + 1
      jump :3
:2    a = c + 1
:3    continue
```

In speculative execution

```
test cond
a = b + 1 || a = c + 1
continue
```

Both instructions; `a = b + 1, a = c + 1`, are executed speculatively if enough resources are available (functional units) but only one of them will be "committed" to write their results. The other result will be ignored depends on the result of the conditional test.

One mechanism that is used in performing speculation is "predicate". Each instruction is tagged with a predicate field. This predicate depends on the conditional test and determines whether the current instruction will be committed or not. For the instructions that are issued concurrently, their predicates will be mutually exclusive, only one of them can be true. From the previous example:

Using predicate

```
test cond => p1, p2
p1: a = b + 1 || p2: a = c + 1
continue
```

`p1, p2` are the predicates which depend on the result of the conditional test. They are mutually exclusive, only one of them can be true. The instructions `a = b + 1, a = c + 1` are tagged with `p1` and `p2` respectively. `a = b + 1` and `b = c + 1` are executed concurrently but only one of them will write their results depends on whether `p1` is true of `p2` is true. Only one of the result will be written back to the register.

For load/store instructions, if execute speculatively may result in an exception. For example, load before the address of operand is known to be valid or a cache miss occurs. To allow load/store instruction to be executed speculatively, their execution are separated into two phases. The first phase load/store is executed without delivering the exception and the second phase when the result is needed the exception is delivered. See the following example:

```
if cond == true then a = b[i]
```

In normal execution

```
            test cond
            jump if false :2
            a = b[i]
      :2    continue
```

`a = b[i]` can be executed speculatively before the condition is known. This can be accomplished by "hoising" the `a = b[i]` up before the test. This will hide the latency of memory access because the instruction is issued much earlier than the result is needed.

In speculative execution

```
            load.s r = b[i]
            test cond => p1
            p1: check.s a = r
```

`load.s` gets `b[i]` without deliver an exception. An exception that will occur in case `i` is invalid or `b[i]` causes a cache miss is suppressed. When the result of load is needed `check.s a = r` is executed and this instruction will delivered the exception (if pending). If the exception occurs the processor will call a trap to operating system to bring in a new page in the virtual memory or a line of cache is refreshed.

Example The following example shows how speculative can improve the performance of a hard to schedule "pointer chasing" code. An associative list is a data structure that stores "key" and "value". To get a value of a key, the list has to be traversed and the key field compared until the required "key" is found or the end of list is encountered (Fig. 7.8).

```
(("house" 1000) ("car" 200) ("table" 30))
```

The program to the value of a key is as follows.

```
1  for ( fp = lenv; fp != NIL ; fp = cdr(fp) )
2    for ( ep = car(fp) ; ep != NIL ; ep = cdr(ep) )
3      if ( sym == car(car(ep)) )
4         return ( cdr(car(ep)) )
```

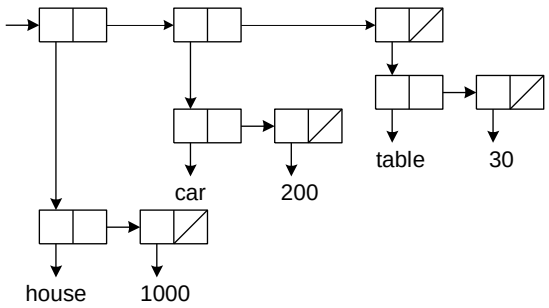


Figure 7.8 the example data structure in dot pair.

The control flow of the above program is as follows.

```
line 1 :1   fp = cdr(fp)
line 2 :2   if ( sym == car(car(ep)) ) goto :3
line 3      if ep != NIL goto :2
line 4      if fp != NIL goto :1
line 5      exit
line 6 :3   return
```

The loop of interest is the inner loop (line 3). Induction variable `ep` is dereferenced twice and compared to the value `sym`. The two loads `x = car(ep)` and `y = car(x)` are in the loop's critical path. Speculation is used to start the loads as soon as possible. A compiler can schedule the loads as soon as `ep` is known, but before the processor determines whether it is a valid pointer.

Suppose a hypothetical machine has five functional units, all the units can execute any instruction. Only two units have access to the first-level data cache which has the one clock latency. This allows two units to issue load and store instructions.

clock	unit 1	unit 2	unit 3
1	ld ep1		
2	ep1 == NIL => p1	ld.s car(ep1)	p1: br nxt_fp
3	check.s	ld x = car(car(ep1))	
4	sym == x => p2	p2: br return	br nxt_ep

Figure 7.9 Schedule of the first iteration

The `ld.s` in cycle 2 is speculative `if(sym == car(car(ep))`, this is before the processor determines whether or not ep1 is a valid pointer. It is known at the cycle 3 when the check corresponding to `ld.s car(ep1)` can be executed. Once the processor loads `car(ep1)` in cycle two, it can load `car(car(ep1))` in cycle three. In cycle four, the processor can check `sym == x`. This schedule shows that one loop iteration can be executed in four cycles without any stall.

To use the resource more efficiently, the loop can be unrolled. The inner loop is unrolled twice.

clk	unit1	unit2	unit3	unit4	unit5
0	ld ep1				
1	ld.s car(ep1)	ep == NIL => p1	ld.s ep2 = cdr(ep1)	p1: br nxt_fp	
2	check.s	ld car(car(ep1))	ld.s car(ep2)	ep2 == NIL => p3	
3	cmp == sym => p2	ld.s car(car(ep2))	p2: br return	p3: br nxt_fp	
4	check.s	ld nxt_ep1 cdr(ep2)	cmp == sym => p4	p4: br return	br nxt_ep

Figure 7.10 Schedule of the two iterations

The loop's second iteration provides an example of the use of control speculation. In cycle one, the processor loads the next value of `ep2` speculatively – that is, before it determines if the first iteration's induction variable is valid. The two loads depend on `ep2` as well as the speculative loads of `car(ep2)` in cycle two and of `car(car(ep2))` in cycle three. The program branches out in cycle three if the `ep2 == NIL`, so all the instructions using `ep2` before that point are speculative. If any speculative load in a chain triggers an exception, the last load will deliver an exception to `check.s`. The processor can execute two loads in parallel in cycle one and two. Two iterations of loop can be executed in 4 cycles (the first `ld ep1` is considered to be in the outer loop). Without control speculation, the static schedule of this code would take six cycles for two iterations of this loop. The loads dereferencing the variable `ep1` can be started before `ep1` has been checked against `NIL`, similarly for `ep2`.

Pipeline in some real machines

PowerPC601

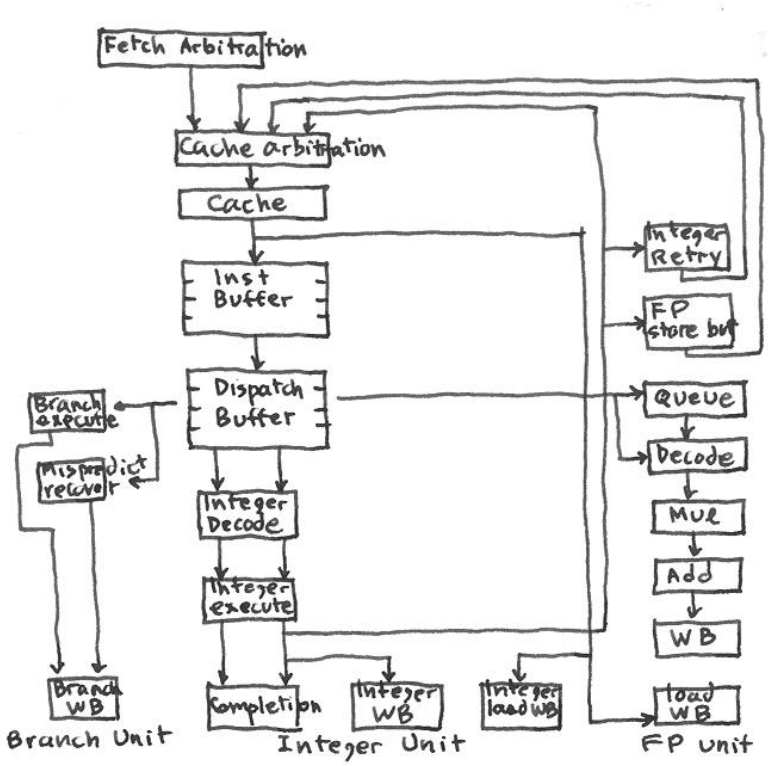


Figure 7.11 PowerPC601 pipeline

PowerPC 601 has many functional units which have different pipeline:

- Dispatch unit holds instruction buffer
- Branch processing unit handles all branch instructions
- Floating-point unit
- Integer unit

Branch inst.	Fetch	Dispatch Decode Execute Predict				
Integer inst.	Fetch	Dispatch Decode	Execute	Writeback		
Load-Store	Fetch	Dispatch Decode	Ads Gen	Cache	Writeback	
FP inst.	Fetch	Dispatch	Decode	Execute1	Execute2	Writeback

PowerPC601 can issue branch and floating-point instructions out of order. Branch processing employs fixed rule to reduce stall cycle as follows.

1. Scan the dispatch buffer (8 deep) for branch instructions. Target address are generated.
2. Determine the outcome of conditional branches :
   a. *will be taken*: for unconditional and for known condition code and indicate branching
   b. *will not be taken*: for unconditional and for known condition code and indicate no branching
   c. *outcome cannot yet be determine*: for backward branch guess taken, for forward branch guess not taken.

The designer did not use branch history for the reason that it will achieve minimum payoff.

Pentium

The Pentium has 5 stages pipeline , two integer units

1. Prefetch
2. Decode stage 1 (instruction pairing)
3. Decode stage 2 (address generation)
4. Execute
5. Writeback

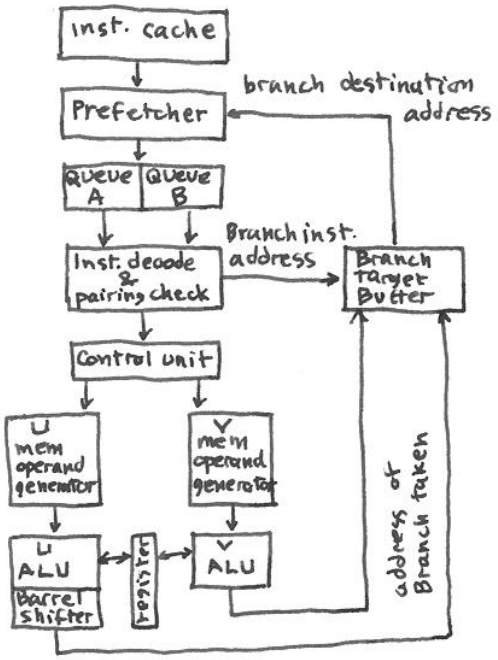


Figure 7.12 Pentium pipeline

Instruction Pairing rule

The instructions that can be issued together must meet the following constraints:

1. both instructions are simple
2. no RAW, WAW
3. no displacement and immediate operand

The simple instructions are instructions that are in the following groups: `mov`, `alu r r`, `shift`, `inc`, `dec`, `pop`, `lea`, `jump`, `call`, `jump conditional near` . Pentium processor uses dynamic branch prediction scheme. A branch target buffer (BTB) stores branch destination address associated with the current branch instruction. Once the instruction is executed the history is updated. The BTB is 4-way set associative cache with 256 lines. Each entry uses the address of branch instruction as a tag. The value field contains the branch destination address for the last time this branch was taken and a two-bit history field.

Further reading

The "pointer chasing" example comes from [DUL98]. For a more detailed description of Intel IA64 use of speculative see [DUL98]. For the impact of compiler technology see [HWU95]. For the use of compredicate in branch prediction see [MAH94]. The historical aspect of VLIW begins from ELI-512 [FIS83].

References

[ACO87] Agerwala, T. and Cocke, J., "High performance reduced instruction set processors," IBM Tech. Rep. , March, 1987.

[AND67] Anderson, D., Sparacio, F., and Tomasulo, R., "The IBM System/360 model 91: Machine philosophy and instruction handling", IBM Jour. of Research and Development 11(1):25-33, 1967.

[CHA82] Chaitin, G., "Register allocation and spilling via graph coloring", Proc. of SIGPLAN symposium on compiler construction, 1982.

[DUL98] Dulong, C., "The IA64 Architecture at work", IEEE Computer, July 1998, pp. 24-32.

[ELL87] Ellis, J., Bulldog: a compiler for VLIW architecture, Cambridge, MA, MIT Press, 1987.

[FIS81] Fisher, J., "Trace scheduling: a technique for global microcode compaction", IEEE Trans. on computers, Vol C-30 (July 1981), pp. 478-490.

[FIS83] Fisher, J., "Very long instruction word architectures and the ELI-512", Proc. of the 10th annual symposium on computer architectures, 1983, pp.140-150.

[HWU95] Hwu, W., Hank, R., Gallagher, D., Mahlke, S., Lavery, D., Haab, G., Gyllenhaal, J., and August, D., "Compiler technology for future microprocessors", Proc. of the IEEE vol.83, no.12, December 1995, pp.1625-1640.

[JOH90] Johnson, M., Superscalar microprocessor design, Prentice-hall, 1990.

[JOU89] Jouppi, N., "The distribution of instruction-level and machine parallelism and its effects on performance", IEEE Trans. on computers, 38(12) (December 1989):1645-1658.

[JOU89a] Jouppi, N, and Wall, D., "Available instruction-level parallelism for superscalar and superpipelined machine", Proc. of 3rd Int. Conf. on Architecutural Support for Programming Languages and Operating Systems, 1989, pp.272-282.

[MAH94] Mahlke, S. et al, "Characterizing the impact of predicated execution on branch prediction", Proc. Annual Int. Symp. on Microarchitecture 27, IEEE CS press, 1994, pp.217-227.

[THO70] Thornton, J., Design of a computer: the Control Data 6600, Glenview, IL: Scott Foreman, 1970.

[TOM67] Tomasulo, R., "An efficient algorithm for exploiting multiple arithmetic units", IBM Jour. of Research and Development 11(5), 1967.

Chapter 8

Vector machines

This chapter discusses one of the most important class of computer architecture, vector machines. A vector machine is a high performance machine suitable for vector computation that is prevalent in numerical problems. It has two key qualities of efficiency and wide applicability. Most vector machines have a pipelined structure and support a streaming mode of data flow through a pipeline. They also support fully parallel operations of multiple pipelines. This chapter describes the general architecture of vector processors and the algorithm to match the architecture to the problems to obtain efficient processing over large classes of computations.

What is a vector machine

What more can be done beside pipelining and multiple issues of instructions to increase a processor performance? There are two factors in performance limitation:

1. *Clock cycle time* – the clock cycle time can be decreased by making the pipelines deeper but very deep pipelining can eventually slow down a processor. A superscalar design needs complex control unit to detect data hazard and to solve control hazard. This also limits the clock cycle time.
2. *Instruction fetch and decode rate* – this prevents fetching and issuing of more than a few instructions per clock cycle.

The cycle time is also limited by the cycle time of the control unit. Scheduling the pipeline and superscalar machines needs complex control units to detect data hazards and to reduce control hazards. It is just as difficult to schedule a pipeline that in *n* times deeper as it is to schedule a machine that issues *n* instructions per clock cycle.

Vector processors provide high-level operations that work on vectors – linear array of numbers. Vector operations have several important properties that solve most of the problems above:

1. The computation of each result is independent of the computation of previous results, allowing a very deep pipeline without generating any data hazards. Essentially, the absence of data hazards was determined by the compiler or programmer.
2. A single vector instruction is equivalent to executing an entire loop. Thus, the instruction bandwidth requirement is reduced.
3. Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the applications domain can use them frequently.

The primary components of a vector processor are: *vector registers* each must have two read ports and one write port, *vector functional units* that can start a new operation on every clock cycle, *vector load/store unit* that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle after an initial latency, a set of *scalar registers* provide data as input to the vector register functional units, as well as compute addresses to pass to the vector load/store unit.
(vector reg-reg CRAY, vector mem-mem CDC)

Vector operations

The basic idea of vector processors is to combine two vectors, element by element, to produce output vector. If A, B, C are vectors with *n* elements, a vector processor can perform as one instruction, C = A + B, which is interpreted as

```
for i = 1 to n
  c(i) = a(i) + b(i)
```

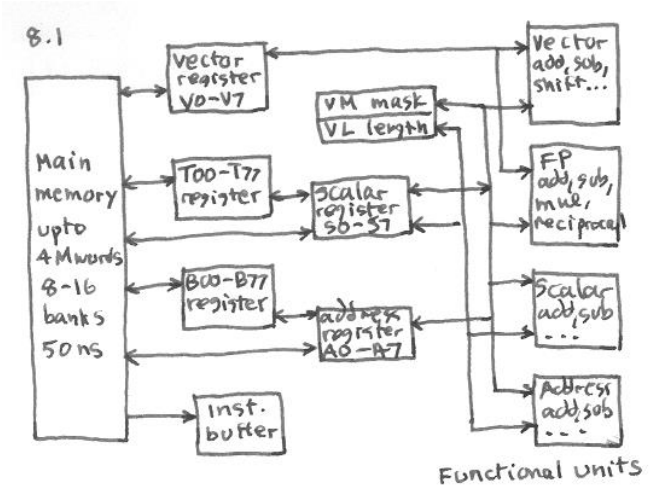


Figure 8.1 CRAY-1 vector computer

Most vector machines have a pipeline structure. Multiple pipelines may also be provided to increase performance. The price performance ratio of vector computers can be one to two order of magnitude increased throughput for vector computations when compared to serial computers of equal cost. But it is limited to the problems that fit the architecture, i.e. the problems that can be structured as a sequence of vector operations.

Memory bandwidth

Vector machines need large memory bandwidth to sustain the high data rate required to feed pipelined functional units. In the above example, the memory system must supply one element of A and B on every clock cycle. The ALU produces one output during each clock cycle. The difficulty is in designing a memory system to sustain a continuous flow of data from memory to ALU and the return flow of results from ALU to memory. Therefore for the C = A + B, the memory system must has at least three times the bandwidth of a conventional memory system. We can ignore the bandwidth for instruction fetches as a single vector operation can initiate a long vector operation. Therefore the bandwidth required for instruction fetches is negligible as compared to the bandwidth of instruction fetches in a conventional machine.

Two major approaches in designing memory system for vector machines:

1. Using independent memory modules in main memory to support concurrent access to independent data. *Interleaving memory* increases the memory bandwidth by parallely access every memory banks.
2. Using intermediate *high-speed memory*.

Interleaving memory increases memory bandwidth by parallel accessing every memory banks. Distribution of data in multi memory modules is important for the performance. Access to the same memory module will cause load/store stall. The figure shows eight memory modules. They provide a system with eight times the bandwidth of a single module. Each of three data streams has an independent path to the memory system so that each stream can be active simultaneously, provided that one module serves only one path at a time.

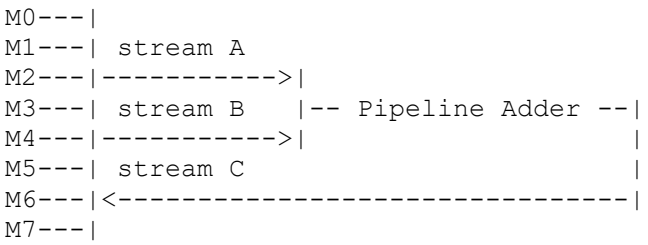


Figure 8.2 Eight 3-port memory modules sustain two reads one write at every clock.

We will illustrate the use of interleaving memory to implement vector arithmetic, C = A + B. Assume a memory cycle takes two clock cycles. The bandwidth required to service the pipeline adder is at least six times the bandwidth of a single memory module. The vectors A, B, and C are laid out in the memory so that they start in modules M0, M2, and M4 respectively (Fig 8.3). Their successive elements lie in successive memories.

```
M0 a0 b6 c4
M1 a1 b7 c5
M2 a2 b0 c6
M3 a3 b1 c7
M4 a4 b2 c0
M5 a5 b3 c1
M6 a6 b4 c2
M7 a7 b5 c3
```

Figure 8.3 physical layout of three vectors in the memory

pipe4					0	1	2	3	4	5	6	7
pipe3				0	1	2	3	4	5	6	7	
pipe2			0	1	2	3	4	5	6	7		
pipe1		0	1	2	3	4	5	6	7			
M7					RB5	RB5	RA7	RA7	W3	W3		
M6				RB4	RB4	RA6	RA6	W2	W2			
M5			RB3	RB3	RA5	RA5	W1	W1				
M4		RB2	RB2	RA4	RA4	W0	W0					
M3		RB1	RB1	RA3	RA3							
M2	RB0	RB0	RA2	RA2								W6
M1		RA1	RA1				RB7	RB7			W5	W5
M0	RA0	RA0				RB6	RB6			W4	W4	
clock	0	1	2	3	4	5	6	7	8	9	10	11	12

Figure 8.4 timing diagram for vector addition in pipeline(4 stages)
R = read, W = write, A,B input vectors

The figure 8.4 shows the vector addition in a vector processor with eight memory modules and 4-stage pipelined arithmetic unit. The timing diagram shows the flow of data through eight memory modules. The memory latency is 2 clocks both reading and writing. Reading A0 of module M0 at clock 0 will enable the data to be available at the pipeline unit at clock 2. Writing the result W0 to the module M4 at clock 6 will finish at clock 8.

Two vectors are allocated to modules so that no conflicts occur. At clock 0, M0 and M2 initiate READs to the first elements of vectors A and B. These elements appear at the pipeline inputs at clock 2, and the corresponding output appears at the end of clock 5. At clock 1, M1 and M3 initiate READs to the second elements of the input vectors. At clock 5, the first output emerges from the ALU pipeline, the next clock, clock 6, M5 and M6 are busy reading A5 and A6. M5 delivers A5 at the beginning of clock 7 and M6 delivers A6 at the beginning of clock 8. At clock 6, M4 initiates WRITE to put away C0, M0 initiates READ of B6. Note how the arrangement of timing enables all operations to proceed without a collision. In reality, the pipeline is never as well behaved as ideal examples are.

At the peak rate of data access, at the clock no. 7, there are two memory modules still free. This indicates that there is surplus memory bandwidth. The number of memory module can be reduced to 6 modules to eliminate this surplus. Now, observe that at the clock 7 all memory bandwidth is used (Fig. 8.5).

```
pipe4                       0   1   2   3   4   5   6   7
pipe3                   0   1   2   3   4   5   6   7
pipe2               0   1   2   3   4   5   6   7
pipe1           0   1   2   3   4   5   6   7
M5                  RB3 RB3 RA5 RA5 W1  W1
M4              RB2 RB2 RA4 RA4 W0  W0                  W6
M3          RB1 RB1 RA3 RA3         RB7 RB7         W5  W5
M2      RB0 RB0 RA2 RA2         RB6 RB6         W4  W4
M1          RA1 RA1         RB5 RB5 RA7 RA7 W3  W3
M0      RA0 RA0         RB4 RB4 RA6 RA6 W2  W2
clock 0   1   2   3   4   5   6   7   8   9   10  11  12
```

Figure 8.5 timing diagram for vector addition with 6 memory modules
R = read, W = write, A,B input vectors

S1 with vector units

To illustrate the performance of a vector processor over a conventional processor, we examine a concrete example. For a vector processor, we extend S1 architecture to include vector facilities. The machine is called S1v. The S1v is a 32-bit processor, it includes a vector load/store unit, 8 vector registers, and usual vector functional units: FP add, FP mul, FP div, plus an integer unit and a logical unit. The vector load/store unit provides vector registers with one input on every clock cycle. Each vector register contains 64 elements. The vector instructions have 3-operand format.

For a conventional processor, we extend the simple S1 to include floating-point facilities. The extended S1 has f0.. f7 as 32-bit floating-point registers, it also has immediate addressing mode and a reasonable instruction set to perform floating-point arithmetic. This machine is called S1x.

Take typical vector problem,

```
Y = a * X + Y
```

where `Y, X`, are vectors `a` is scalar This expression is called SAXPY or DAXPY (single-precision or double-precision A*X Plus Y) loop that forms the inner loop of the Linpack benchmark. Linpack is a collection of linear algebra routines; the Gaussian elimination portion of Linpack is the segment used as benchmark. SAXPY represents a small piece of the SV elimination code, though it takes most of the time in the benchmark.

Assume that the number of elements, or length, of a vector register matches the length of the vector operation we are interested in. Starting addresss of `X` and `Y` are in `rx, ry`.

DAXPY in S1x

notation for the assembly code: `op dest, source`

```
        ld f0, a
        mov r4, rx
        add r4,#512       ; last address to load
loop:   ld f2,(rx)        ; load X(i)
        fmul f2,f0        ; a * X(i)
        ld f4,(ry)        ; load Y(i)
        fadd f4,f2        ; a * X(i) + Y(i)
        st (ry),f4        ; store into Y(i)
        add rx,#8         ; increment index to X
        add ry,#8         ; increment index to Y
        cmp r4,rx         ; compute bound
        jnz loop          ; check if done
```

DAXPY in S1v

```
        ld f0,a           ; load scalar a
        vld v1,rx         ; load vector X
        vmul v2,f0,v1     ; vector-scalar mul v2=f0*v1
        vld v3,ry         ; load vector Y
        vadd v4,v2,v3     ; vector add v4 = v2 + v3
        vst v4,ry         ; store the result
```

Comparing the two code segments it is easy to see that the vector machine greatly reduces the dynamic instruction bandwidth, S1v executing only 6 instructions versus almost 600 for S1x to compute 64 element vectors. Another important difference is the frequency of pipeline interlocks. In S1x every `fadd` must wait for `fmul` and every store must wait for the `fadd`. On the vector machine, each vector instruction operates on all the vector elements independently. Thus, pipeline stalls are required only once per vector operation, rather than once per vector element. In the example, the pipeline stall on S1x will be about 64 times higher than on S1v.

Example calculation of the clock used and the number of instruction fetched and executed for both machines, assume every instruction take 1 clock to execute.

The time taken to calculate DAXPY for 64 elements :

S1x : 3 + (9 × 64) = 579 clocks

S1v : 1 + (5 × 64) = 321 clocks

The number of instruction fetched and executed :

S1x : 579 instructions

S1v : 6 instructions

The execution time of S1v can be greatly reduced by using the technique called "chaining". Instead of waiting for a vector instruction to complete, the instruction in the vector operation of the next instruction can be started as soon as the first element of the previous instruction finished. For this example, assume all vector instructions can be chained, it will take 1 + 5 + 64 = 70 clocks to complete.

How to program a vector machine

To use the vector unit, the program must be "vectorised", i.e. transform the loop into vector operations. If VL is vector length of the machine

```
for i = 1 to VL
  c(i) = a(i) + b(i)
```

can be transformed into one vector operation

```
vadd v0 v1 v2
```

where v0 holds c, v1 holds a, v2 holds b

There are several consideration for programming vector machines such as vector length and vector stride because the data length does not necessary match the vector length of the machine.

Vector length

When the data lenght is not equal to vector length of a machine, the loop can be transformed into loop of vector operations, each of maximum length plus one loop for the rest of element. This is called "strip-mine". Let VLR be the vector length register that holds the number of element in a vector operation, MVL be the maximum length of vector unit (VLR ≤ MVL). The following program can be strip-mined

```
for i = 1 to n
  y(i) = a * x(i) + y(i)
```

to

```
low = 1
VL = ( n mod MVL )
for j = 0 to ( n / MVL )
  for i = low to low + VL -1
    y(i) = a * x(i) + y(i)
  low = low + VL
  VL = MVL
```

The inner loop (`for i = ..`) can be vectorised with length VL. VL is equal to (n mod MVL) the first round through the loop, and becomes MVL for the rest. The last line, VL = MVL, sets VL to the maximum vector length for the second round the loop and the rest. The variable `low` points to the beginning of the data block.

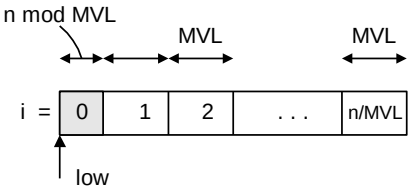


Figure 8.6 strip mining a vector

Vector stride

A vector stride is the distance separate elements that are to be merge into a single vector. If the vector stride is not equal to one, it can caused slow down in the memory access. A vector register is used to load non-unit stride vector and subsequently access to the vector register will have adjacent vector elements. The following program does matrix multiply.

```
for i = 1 to 100
  for j = 1 to 100
    a(i,j) = 0.0
    for k = 1 to 100
      a(i,j) = a(i,j) + b(i,k) * c(k,j)
```

This program can be vectorised to multiply each row of `b` by each column of `c`. The inner loop is strip-mined with `k`. How adjacent elements in `b` and `c` are addressed?

Example A 100 × 100 matrix, for row major order b(i,j) is adjacent to b(i,j+1)
(1,1) (1,2) ... (1,100) (2,1) (2,2) ... (2,100) ... (100,100)
and column major order b(i,j) is adjacent to b(i+1,j)
(1,1) (2,1) ... (100,1) (1,2) (2,2) ... (100,2) ... (100,100)
For row major order `c` has a stride of 100, `b` has a stride of one.

Once a vector is loaded into a vector register it had logically adjacent elements. This enables the machine to handle the non-unit stride such as `c` above. Vector load for non unit stride is complicate and can caused memory bank conflict.

Loop - carried dependency

Not all the loops can be vectorised. The following program is not vectorisable

```
1 for i = 1 to 100
2   a(i+1) = a(i) + b(i)
3   b(i+1) = b(i) + a(i+1)
```

This is because in the loop the computation used the value of an earlier iteration. `a(i+1) = a(i)`. Also line 3 has RAW on line 2 for `a(i+1)`

Improving performance of a vector machine

This section describes the techniques to improve the performance of a vector machine. The first technique, "chaining", improves the speed of running a sequence of vector operations. Other technique introduces transformation to change loops into vector operations.

Chaining

It is invented by Seymour Cray and introduced in CRAY-1. When vector operations are running in sequence, the first result once completed is immediately make available to the next operation. Chaining allows vector operations to run in

parallel. A sustained rate of more than one operations per clock can be achieved even when the operations are dependent. Consider a vector sequence

```
vmul v1 v2 v3
vadd v4 v1 v5
```

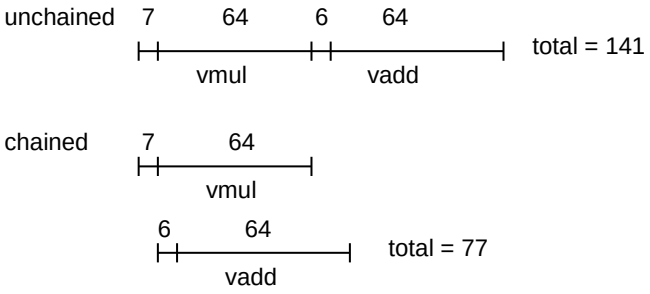


Figure 8.7 Timings for a sequence of dependent vector operations

After the first result of v2 × v3 appears, the vector addition `vadd` can start immediately without waiting for the last result from `vmul`. This allows two vector instructions to proceed concurrently and finish sooner.

Conditional statement

For real programs the level of vectorisation is not very high. The ability to vectorise is influenced by the algorithms used and how the programs were written. It also depends on the ability of the compiler. To get best performance from a vector machine, a significant modification or a rewrite of a program is needed.

When there are *conditionals* inside loops or the matrix is *sparsed*, programs can not be vectorised using the techniques we discussed so far. The sparse matrix is stored in some compacted form which elements are accessed indirectly. The following loop can not be vectorised.

```
for i = 1 to 64
  if (a(i) != 0) then a(i) = a(i) - b(i)
```

The inner loop can be vectorised if we can selectively run it for the element which `a(i) != 0`. This can be achieved using a *vector mask*. The vector mask can be loaded from the vector test instruction and any vector operation will

operate on the element whose the corresponding entry in the vector mask is 1. The above loop can be vectorised using vector mask as follows: `ra, rb` are the starting address of `a, b`.

```
lv v1 ra          ; load vector a into v1
lv v2 rb          ; load vector b
ld f0 #0          ; f0 = 0
vsnes f0 v1       ; set VM to 1 if v1(i) != 0
vsub v1 v1 v2     ; subtract under vector mask
cvm               ; set vector mask to all 1
sv ra v1          ; store the result to a
```

This loop contains sparse matrix

```
for i =1 to n
  a(k(i)) = a(k(i)) + c(m(i))
```

The sum of sparse vector on array `a` and `c` using index vector `k` and `m` to designate the non zero elements of `a` and `c` (`a` and `c` must have the same number of non zero elements -- `n`).

For a sparse matrix, the supporting operation is called "scatter-gather". A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by base_address + offsets in the index vector. After completing the computation, the vector can be store in expanded form using a scatter store with the same index vector. Suppose we have the instruction `lvi` (load vector indexed) and `svi` (store vector indexed), `ra, rc, rk` and `rm` contain the starting address of vector `a, c, k, m`. The inner loop of the above program can be vectorised as follows.

```
lv vk rk            ; load k
lvi va (ra + vk )   ; load a ( k(i) )
lv vm rm            ; load m
lvi vc (rc + vm )   ; load c ( m(i) )
vadd va va vc       ; add
svi (ra + vk ) va   ; store a( k(i) )
```

The load indexed vector instruction is a generalisation of the load indexed scalar by using a vector register as an index register.

Vector reduction

Reduction is a loop that reduces an array to a single value by repeated application of an operation. For example a dot product:

```
dot = 0.0
for i = 1 to 64
  dot = dot + a(i) * b(i)
```

This loop has a loop-carried dependency on `dot` and cannot be vectorised. If we split the loop to separate out the vectorisable part:

```
for i = 1 to 64
  dot(i) = a(i) * b(i)
for i = 2 to 64
  dot(1) = dot(1) + dot(i)
```

The variable `dot` has been expanded into a vector, this is called "scalar expansion".

Another technique is called "recursive doubling". A loop with recurrence is transformed using adding sequences of progressively shorter vectors -- two 32-element vectors and then two 16-element vectors, and so on. It is faster than doing all operations in scalar mode. An example of doing the second loop above with recursive doubling :

```
len = 32
for j = 1 to 6
  for i = 1 to len
    dot(i) = dot(i) + dot( i + len )
  len = len / 2
```

When the loop is done the sum is in `dot(1)`.

Performance of vector machines

The time to complete a vector operation depends on

1. vector start up time
2. initiation rate

Vector start up time is the pipeline latency. It depends on the depth of the pipeline of that functional unit. Initiation rate is the time per result once a vector operation is running, usually one result per clock for a fully pipeline functional unit. The time to complete a vector operation on vector of length *n* is

$$\text{start up time} + n \times \text{initiation rate}$$

Example Start up time for a vector multiply is 10 clocks. Initiation rate is one per clock. What is the number of clock per result for a 64 element vector ?

clock per result = total time / vector length
= (start up time + 64 × initiation rate) / 64
= 1.16 clock per result

What determine the start up time and initiation rate

Let consider register to register operation (therefore ignore memory latency)

start up time = depth of FU pipe, or the time to get the first result

initiation rate = rate that a FU can accept operands, for fully pipe, one per clock.

Pipeline depth is determined by type of operation and clock cycle time.

Example Cray -1 has the following vector unit characteristic:

	start up time
FP add	6
FP mul	7
FP div	20
FP load	12

stall 4 clocks on RAW

Sustained rate is the time per element for a set of vector operations.

Example what is the sustained rate for the following sequence of instruction. Assume a vector of length 64 ?

```
vmul v1 v2 v3
vadd v4 v5 v6
```

We can chart the time line :

	start at	complete at
`vmul`	0	7+64 = 71
`vadd`	1	1+6+64 = 71

Because of independent vector operations, both instruction run concurrently most of the time. Sustained rate is one element per clock, this sequence executes 128 FLOPS in 71 clocks = 1.8 FLOPS per clock.

A simple model of vector performance

The equation for execution time of a vector loop with n elements, T_n. [HEN96]

$$T_n = T_{base} + \lceil n / MVL \rceil \times (T_{loop} + T_{start}) + n \times T_{element}$$

T_n total running time
$T_{element}$ time to process one element
T_{loop} overhead for scalar code to strip-mine
T_{start} vector start up time
T_{base} overhead to compute starting address and set up vector control, occur once for one vector operation.

There are start-up overhead T_{start} and the overhead of executing the strip-mined loop T_{loop}. The strip-mining overhead arises from the need to reinitiate the vector sequence and set the VL. The values T_{start} , T_{loop} and $T_{element}$ are compiler and processor dependent.

Example For CRAY-1 T_{base} 10 clocks, T_{loop} 15 clocks. What is the total running time for DAXPY using CRAY-1 with 64 element vectors?

DAXPY	start at	complete at
`lv v1 rx`	0	12 + 64 = 76
`vmul v2 s1 v1`	12 + 1 = 13	13 + 7 + 64 = 84
`lv v3 ry`	76 + 1 = 77	7 7 + 12 + 64 = 153
`vadd v4 v2 v3`	77 + 1 + 12 = 90	90 + 6 + 64 = 163
`sv ry v4`	160 + 1 + 4 = 165	165 + 12 + 64 = 241

$T_{start} = 241 - 64 \times T_{element} = 241 - 192 = 49$

calculate the other way from the pipeline latency

$$12 + 7 + 12 + 6 + 12 = 49$$

Using MVL = 64, $T_{loop} = 15$, $T_{base} = 10$, $T_{element} = 3$

$$T_n = 10 + \lceil n/64 \rceil \times (15 + 49) + 3n = 4n + 64$$

The sustained rate is over 4 clock cycles per iteration.

The *peak* performance is the performance without the start-up overhead. The peak performance is higher than the sustained performance. We can calculate the peak performance of a vector machine by

$$R^* = \lim_{n \to \infty} (\text{operation per iteration} \times \text{clock rate} / \text{clock cycle per iteration})$$

R* is MFLOPS rate on infinite vector length

$$\text{Peak performance} = \text{number of FLOPS per iteration} * \text{clock rate} / T_{element}$$

$$R_n = \text{number of FLOPS per iteration} * \text{clock rate} / T_n$$

Example The peak performance of CRAY-1 200MHz on DAXPY is

$$R^* = \lim_{n \to \infty} (\text{operation per iteration} \times \text{clock rate} / \text{clock cycle per iteration})$$

The numerator is independent of n

$$R^* = (\text{operation per iteration} \times \text{clock rate}) / \lim_{n \to \infty} (\text{clock cycle per iteration})$$

$$\lim_{n \to \infty} (\text{clock cycle per iteration}) = \lim_{n \to \infty} (T_n / n) = \lim_{n \to \infty} ((4n + 64) / n)) = 4$$

$$R^* = 2 \times 200 \text{ MHz} / 4 = 100 \text{ MFLOPS}$$

Example The Linpack benchmark is a Gaussian elimination on a 100 × 100 matrix. The vector lengths range from 99 to 1. A vector of length k is used k times. Thus the average vector length is 66.3. We obtain an accurate estimate for the performance of DAXPY code, for 66 element vectors.

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 326$$

assume our vector processor has 200 MHz clock

$$R_{66} = 2 \times 66 \times 200 / 326 = 81 \text{ MFLOPS}$$

Final remarks

The first vector machines were the CDC STAR-100 [HIN72] and TI ACS [WAT72], both announced in 1972. Both were memory-memory vector machines. Cray who worked on CDC 6600 and the 7600, founded Cray Research and introduced the CRAY-1 in 1976 [RUS78]. The CRAY-1 used a vector-register architecture to significantly lower start-up overhead. Most importantly, the CRAY-1 was also the fastest scalar machine in the world at that time.

CRAY-1 does not have FP units. It also used T, B register as high speed access to memory, similar to the use of cache memory to increase the memory bandwidth. However, the memory management falls into the hand of programmer. It also has instruction buffer, similar to instruction cache.

CDC-STAR has three data streams feeding pipeline functional units similar to a simple vector machine. It used delay-line to distribute memory access across the memory banks.

Notwithstanding, the vector machines, once called the supercomputer class, have very high scalar performance as they are built for highest possible performance without compromising with cost. In the real-world use for many applications they run very fast even though those applications rarely used vector units.
(something about MMX and Patterson IRAM?)

References

[HEN96] Hennessy, J. and Patterson, D., Computer architecture : quantitative approach, 2nd ed., Morgan Kaufmann Pub. Inc. 1996.

[HIN72] Hintz, R and Tate, D., "Control data STAR-100 processor design", COMPCON, IEEE, September, 1972, pp. 1-4.

[RUS78] Russel, R., "The CRAY-1 processor system", Comm. of the ACM 21:1 (January), 63-72.

[STO93] Stone, H., High performance computer architecture, 3rd ed. Addison Wesley, 1993.

[WAT72] Watson, W., "The TI ASC – A highly modular and flexible super processor architecture", Proc. AFIPS Fall Joint Computer Conf., 1972, pp. 221-228.

Chapter 9

Stack machines

Stack machines are processors that the instruction set operate on implicit data in a stack structure. They were once very popular for implementing "virtual machine" mostly in software. It is very natural to translate a high level language, especially the block-based languages, so called structured languages (PASCAL, C and others), into these virtual machines. Stack machines can be implemented directly into hardware as well. It has the advantage of being quite simple and its executable code is very compact. This chapter explores stack machines in more details.

The use of stacks

A stack is a LIFO (last in first out) storage with two abstract operations: push and pop. *Push* will put an item into stack at the top. *Pop* retrieves an item at the top of stack. Because a stack is LIFO, any operation must access data item from the top. A stack does not need *explicit* addressing as it is *implicit* in the operators which use stack. Any expression can be transformed into a postfix order and a stack can be use to evaluate that expression. In stack machines, the allocation and reclamation of the temporary space is done automatically via the stack.

Calling subroutines

The stack structure also plays an important role in the calling of subroutines (or function calls). When a program transfers the control to another section of program, the current state of computation is saved (composed of a program counter, local variables, and other values). The place where the state of computation is saved is called "activation record". When an execution of a subroutine is completed, the previous state of computation is restored (this action is called "return" from subroutines). Local variables can also be stored as a part of the activation record. Because many of today programming languages are

structured or "block oriented" the creation and deletion of activation records behave like LIFO. A stack is used to store activation records. Two pointers are required to keep track of the thread of control: `fp` – frame pointer which points to the current activation record and `sp` – the current stack pointer which points to the current stack area.

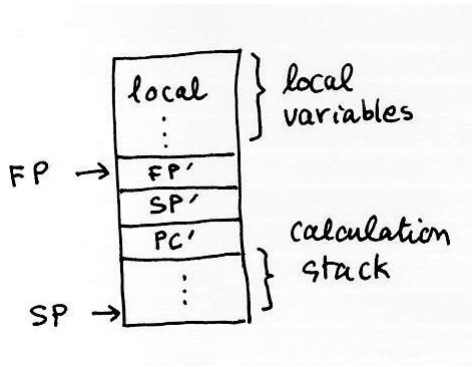


Figure 9.1 an activation record

Local variables can be referenced via offsets from the frame pointer.

Parameter passing by stack

When a function call is made, parameters can be passed from the caller to the callee by pushing them on the stack. The call to a function creates a new activation record. The new activation record can be arranged such that its local area is overlapped with the old stack area, therefore the passing parameters become a part of the local area of the new activation record. The parameter passing occurs without the need to copy them to the new stack. (This scheme is very much like the register window scheme [PAT82] used in SPARC processor family). For example, function `f` calls function `g(A,B)`. `A` and `B` are pushed and the call creates a new activation record (the AR $_{n+1}$). The previous state of computation (`fp', sp', pc'`) is saved in the new activation record. As the new activation record overlapped the old stack, `A` and `B` are in the local area of the new activation record (Fig. 9.2)

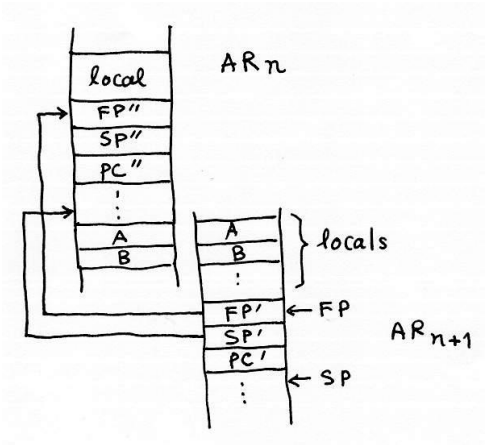


Figure 9.2 the state of stack after a function call occurred

Figure 9.2 shows the state of stack after a funcion call occurrred, `pc` is the current program pointer, `pc'` is the previous `pc`. The frame pointer links activation records together. The stack pointer points to the current stack which is used for storing temporary value of current calculation.

Pure stack machines

Accessing the local area required instructions to have "addressing" to the variables. This is against the spirit of implicit operand. If the local variables are not used, how can operands in the computation stack are accessed? With an exception for the top two items all other items are difficult to get to. We need to be able to reorder and copy a number of items in the stack in order to use them. The following instructions did that:

`dup`	duplicate TOP
`swap`	swap TOP and NEXT
`rot`	1,2,3 -> 3,1,2 get the third item to the top
`over`	copy NEXT to TOP , 1,2 -> 2,1,2

These are just some of the possible instructions. With these instructions, the need to access local variables by addressing them explicitly is minimised. An example of their use.

```
Define f(X,Y) = X*X + Y*Y
```

The function `f` can be evaluated with the following sequence of stack-instructions. Assume `X` and `Y` are the top two items on computation stack when call `f`.

```
dup mul         ; X*X
swap            ; Y, X*X
dup mul add     ; Y*Y + X*X
```

Thinking about rearranging items on stack make it difficult to use pure stack instructions. Having variables avoids the reordering of items on stack because a variable can be accessed by using its name. However, the current compilation techniques can handle the ordering of stack items therefore it frees a programmer from this low level detail.

Microarchitecture of stack machines

A stack machine instruction set can be implemented in many ways. In stack machines, the general purpose registers are not necessary. Two specialised registers are needed to store the state of computation: frame pointer and stack pointer. The stack structure can be either internal of external to a processor. If it is internal the access time is faster. The stack can be regarded, in a way, as data cache. However, the size of stack varies depended on the characteristic of the running program. When the stack is internal, it has a real size limit. Therefore it is necessary to have a mechanism to handle the stack underflow-overflow by pulling and pushing the data between the stack and the main memory (called *stack spilling*). The stack can also reside in the main memory. The number of stack can be more than one Multiple stacks improve the speed of execution. Figure 9.3 shows a typical microarchitecture of a stack machine.

To improve the speed of execution, 2 top elements on the stack can be cached into registers. In Fig. 9.3, the register A and B can be used for this purpose. This will reduce the amount of data movement in the stack because there are a large number of binary operations in the stack which required popping out 2 elements, performs operation and pushing back the result onto the stack.

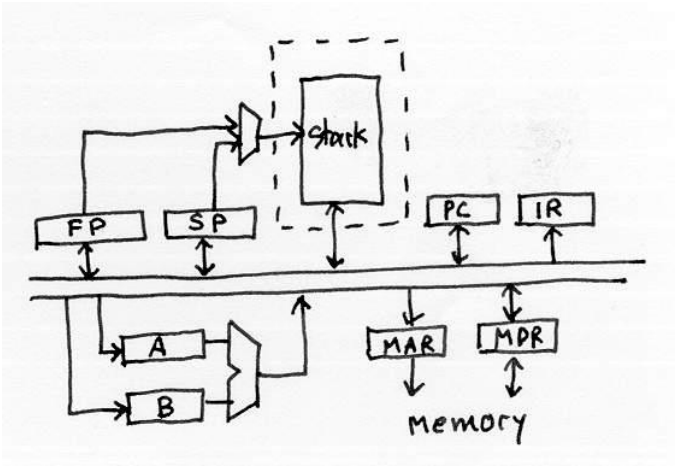


Figure 9.3 a stack machine microarchitecture

Example `result = A op B`, requires 3 data movements in the stack:

```
pop A
pop B
op
push result
```

If the top 2 elements are cached (stored) in the register A and B. The data movement will be reduced to one.

```
A = A op B (access registers)
pop B
```

By caching some elements to registers, the registers are considered a part of the stack. Other operations can be modified to work with this scheme.

Push `x` performs

```
push B
B = A
A = x
```

and Pop `x` performs

```
x = A
A = B
pop B
```

As data movement between registers is faster than accessing the stack, the use of caching will improve the speed of execution for binary operations and for other operations that access 2 top elements.

R1 stack machine

To illustrate a concrete example of a stack machine, the R1 virtual machine will be discussed. The R1 is a stack-based instruction set which is an architectural neutral byte code aims to be portable and reasonably efficient across many platforms [CHO98]. The R1 is intended to be a virtual machine for the purpose of controlling real-time devices. It provides the support for concurrency control and protection of shared resources including real-time facilities such as clock and time-out. The interpreter for R1 is an abstract machine which execute the byte code, providing the multi-task environment for the target machine. The reason for choosing stack-based ISA is the compactness of code size and the ease of implementation. It is well known that many virtual machines are stack-based (example from symbolic computation text).

R1 instruction set

The instructions can be categorised into 6 groups:

1. load/store local
2. load/store
3. control flow
4. arithmetic
5. logical
6. support for real-time and concurrency.

The instructions are as follows.

load/store local :	`lval, rval` (left value or store, right value or load)
load/store :	`lvalg, rvalg, fetch, set`
control flow :	`jmp, jz, call, ret0, ret1, func`
arithmetic :	`add, sub, mul, div, minus, index`
logical :	`not, and, or, lt, le, eq, ne, ge, gt`
others :	`lit` (push literal)

Let us ignore the instructions that support real-time and concurrency. Load/store local instructions access to local variables. Load/store instructions access to data segment. Control flow instructions include jump, conditional jump, call and

return. Table 9.1 below shows the instruction set and its encoding. The opcode is one byte. There are three instruction formats: zero-operand, one-operand and two-operand. All arithmetic and logical instructions are zero-operand. The load/store and control flow instructions are one-operand. Only one instruction that is two-operand. It is the invocation of function, which also create the activation record (`func`). All operands are 16 bits.

Table 9.1 R1 instruction set

	0	1	2	3	4	5	6	7
0	lit	lval	lvalg	rval	rvalg	fetch	set	index
8	jmp	jz	call	func	proc	ret0	ret1	stop
16	add	sub	mul	div	minus	not	and	or
24	le	lt	eq	ne	ge	gt	print	printch
32	send	receive	wait	signal				

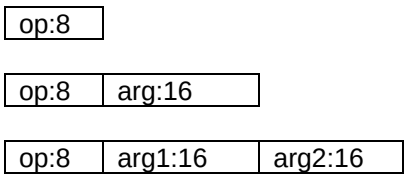


Figure 9.4 R1 instruction formats

Operational semantics of R1 instruction set

The execution model of R1 has one stack. The activation records (thread of control) are stored in the stack (pointed to by Fp). The local variables are stored in the activation record (accessed by the pointer Fp–i where i is the number of the variable) and the calculation is done on top of the current activation record (pointed to by Sp). The following descriptions are the meaning of each instruction of the R1 instruction set. Ip is the instruction pointer.

Notation

CS code segment, DS data segment, SS stack segment, M the memory.
`Aop` arithmetic operators { `add, sub, mul, div, minus` }
`Lop` logical operators `{ not, and, or, le, lt, eq, ne, ge, gt` }

`Uop` unary operators { `print, printch` }

[`Lit n`]	push(n)
[`Lvalg ref`]	push(ref) [1]
[`Lval i`]	push(Fp–i) [1]
[`Rvalg ref`]	push(DS[ref]) [1]
[`Rval i`]	push(SS[Fp–i]) [1]
[`Fetch`]	push(M[pop])
[`Set`]	M[pop1] = pop2
[`Index`]	push(base_ads + index) [2]
[`Jmp ads`]	Ip = ads
[`Jz ads`]	if pop = 0 then Ip = ads [3]
[`Call ads`]	push(Ip), Ip = ads [4]
[`Func np nl`]	save state, new stack frame, pass parameters [5]
[`Proc pid np nl`]	new process descriptor, initialise state, awake
[`Ret0`]	remove stack frame, restore state
[`Ret1`]	remove stack frame, restore state, return a value
[`Stop`]	terminate the process
[`Aop`]	push (pop1 Aop pop2)
[`Lop`]	push (pop1 Lop pop2)
[`Uop`]	push (Uop pop)

1 the variable access
2 the effective address calculation for an array variable
3 if top of stack = 0 jump
4 call to subroutine
5 create new stack frame, invoke a function

Example of a program : bubble sort

This example shows how a high level language program can be translated into ISA of a stack machine. Given `a[n]` an array of integer, the bubble sort program sorts the items in `a[]` in ascending order. Initially, `i=0, j=0`.

```
while( i < n ) {
  while( j < n ) {
    if( a[j] > a[j+1] ) {
      t = a[j];
      a[j] = a[j+1];
      a[j+1] = t;
    }
    j = j+1;
  }
  i = i+1;
}
```

Data segment (word)

```
1: t
2: j
3: i
4: n
5: a[0]
6: ...
```

Code segment (byte)

The code segment shows how the stack-based instruction set is used.

```
68:rval 3 rval 4 ge                       // if not(i<n) exit
75:jz 187
78:rval 2 rval 4 ge                       // if not(j<n) goto 173
85:jz 173
88:lval 5 rval 2 index load               // get a[j]
   lval 5 rval 2 lit 1 add index load gt  // get a[j+1]
109:jz 159                                // if a[j]<= a[j+1] skip
112:lval 1 lval 5 rval 2 index load store // t = a[j]
124:lval 5 rval 2 index                   // get address of a[j]
   lval 5 rval 2 lit 1 add index load store    // a[j] = a[j+1]
144:lval 5 rval 2 lit 1 add index rval 1 store // a[j+1] = t
159:lval 2 rval 2 lit 1 add store         // j = j+1
170:jmp 78                                // loop while(j<n)
173:lval 3 rval 3 lit 1 add store         // i = i+1
184:jmp 68                                // loop while(i<n)
187:
```

Frequency of instruction used

To measure the behaviour of R1 instruction set, the Stanford integer benchmark [HEN] is used. This benchmark is a small (not realistic) suite of programs. It is

composed of seven small programs: hanoi, permutation, quicksort, bubble sort, sieve (prime number), matrix multiplication and 8-queen. The dynamic instruction count are collected and tabulated in Table 9.2. Fig. 9.5 shows the frequency of used of each instruction. This measurement ignores all aspects of multi-task support instructions in R1.

bubble sort	sort 100 integers
hanoi	move 5 disks between 3 poles
matmul	multiply two matrices of size 10 x 10
perm	permute 5 digits
qsort	quick sort 100 integers
queen	find all solutions of 8-queen problem
sieve	find all prime numbers < 100

Table 9.2 the number of dynamic instruction count in the benchmark

bubble	110,611
hanoi	1,300
matmul	41,099
perm	6,901
qsort	88,002
queen	752,804
sieve	57,788

Total number of instruction executed 1,058,520. Fig. 9.5 shows the frequency of each instruction executed. `Rval` is the most frequently used at 267,045. The frequency of used of R1 instruction set running Stanford integer benchmark grouped into category is shown in Table 9.3 below.

Table 9.3 the frequency of used of R1 instruction set

load/store local (lval, rval)	28%
load (rvalg, fetch)	12%
store (lvalg, set)	13%
control flow	10%
arithmetic	20%
logical	11%
others	6%
total	100%

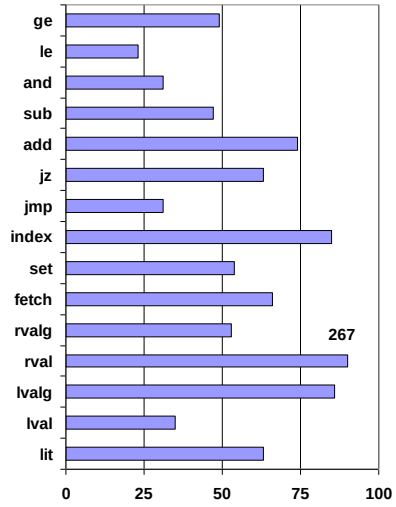


Figure 9.5 the frequency of instructions used ($\times$ 1000)

Comparing this data to 80x86 which is a register machine. The top 10 most frequently used instruction of 80x86 running SPECint92 are [HPA96]:

Table 9.4 the top 10 most frequently used instruction of 80x86

load	22%
conditional branch	20%
compare	16%
store	12%
add	8%
and	6%
sub	5%
move reg-reg	4%
call	1%
ret	1%
total	95%

Table 9.5 compares 80x86 and R1. The difference between the two is that R1 has a large number of load/store local (28%) where as 80x86 uses more load/store (34%) than R1 (25%). Access to local variables in a register machine is to its register set but accessing local variables in a stack machine is to its stack. Therefore for a stack machine load/store local are very frequent.

Table 9.5 Comparison between 80x86 and R1

	80x86	R1
load	22	12
store	12	13
control flow	22	10
arithmetic	13	20
logical	22	11
load/store local	-	28
others	9	6
total	100	100

In a register machine, access to its register set is encoded into each instruction. The R1 accesses its stack very frequently, `rval` is the most frequently used instruction followed by `index` and `lval` which are used to access variables.

Improving the speed of execution

The improvement of R1 ISA is achieved by replacing some long sequence frequently used byte-codes by specialised shorter codes which can be executed faster due to the reduction of stack operations [CHO97]. The sequence are classified into 4 classes :

1. increment, decrement and combined operators (such as "+=" in C language).
2. array access
3. assignment
4. flow control

The first group such as `a = a + 1` can be replaced by the combined operator `a += 1`. The instruction sequence is

```
lval a, rval a, lit 1, add, set.
```

This sequence is replaced by

```
inc a
```

or the sequence `a = a + expression`

```
lval a, rval a, ... exp..., add, set.
```

is replaced by a new instruction

```
addset a.
```

Some flow control instruction such as `while a < b` has the following sequence

```
rval a, rval b, lt, jz $1
```

is replaced by the new instruction

```
jmp_ge a b $1
```

(jump if greater than or equal comparing local variable a and b)

Totally 21 new instructions are added to R1 instruction set in an experiment. These new instructions are used by the compiler when generating optimized code. Running the Stanford integer benchmark shows that these instructions help to speed up 25-120%. The main reason for the speed up is the reduction of the operations on the stack, it has been reduced by 20-80%. Comparing the executable code size, the extended instruction set reduced the size of the executable by 10-34%.

Stack vs register

Stack machines use stacks to store temporary value during calculation and also stored activation records during transfer of control to subroutines. Where as in register machines registers must be allocated explicitly to store temporary values and an explicit LIFO manipulation must be done (via some kind of pointer) to handle activation records.

It is interesting to compare the stack-based machine to the register-based machine. Presently, register-based machines dominate the design in computer industry as they have higher performance. However, the stack-based machines can be much simpler, hence cheaper to produce. The question of performance therefore is important. One work [WON99] uses R1 to experiment with comparing stack-based to register-based by designing a register machine and compare it to R1 at the level of instruction set simulation. We will summarise this work as follows.

The main trust for improving upon a stack-based instruction set is the observation that for a stack-based machine, the performance limit of the interpreter is likely to be the fetch-limit, i.e. the time spending on fetching and decoding an instruction. Hence to improve the performance the number of instruction to be executed should be reduced. This can be achieved by designing an instruction set that each instruction performs as much work as possible.

Based on this assumption, an obvious alternative architecture – a register-based machine is investigated. By comparing two virtual machines: stack-based and register-based using Stanford integer benchmark suite, the result shows that register-based virtual machine interpreter is 1.5 to 2 times faster than the stack-based virtual machine interpreter. Comparing the size of the executable code, they are similar. Although each instruction of the register-based machine is larger (32 bits) there are fewer instruction. Therefore the total size of executable code of both machines are similar.

This work is based on the simulation at the instruction level without concerning concrete implementation at the microarchitecture level. The comparison can illustrate the trend that the reduction of dynamic instruction count in stack machines can speed up its execution. However, it is not possible to make any conclusion about the cycle time, whether the cycle time will increase or decrease. This is still required further investigation.

Conclusion

There are a number of contemporary programming languages that use stack abstraction, for example, Forth, Postscript. Also many languages use "virtual machine" models to implement their executable representations. Pascal has P-system, Smalltalk uses stack for calculation, JAVA has byte-code that use stack model. R1 the real-time concurrent language, uses stack machine as virtual machine.

The stack architecture was very popular and can be dated back quite far, from the Burrough machine with a version of an early multi-tasking operating system. Presently, the RISC architecture dominates the computer design. For a more current discussion about modern stack architecture, the readers are invited to consult Koopman's book. [KOO89]. Presently, one of a commercial CPU that is being designed especially for byte-code interpreting is based on stack architecture [PIC96] [MGH98]. PicoJava is a special CPU which executes Java byte-code, aims for a low power and the embedded applications market, such as Network Computers and hand-held devices. PicoJava is not the only commercially available stack-based processor. There are many others such as Harris RTX etc.

Stack machines are arguable almost the simplest kind of architecture. Its LIFO structure is quite suitable for block-oriented language. The code size for a stack machine can be very compact because most instructions have no operand field. Stack architecture used to be very popular method to implement high level

language machine. Most of modern register machines are faster but there is some renewal effort to improve stack architecture. Notably, Sun's Picojava processor which aims to execute JAVA virtual machine byte-code. Stack architecture may prove to be suitable for the machine in the future.

References

[CHO98] Chongstitvatana, P., A Multi-tasking Environment for Real-time Control, the final report, Research Project Number 132-MRD-2537, Institute of Research and Development, Faculty of engineering, Chulalongkorn universiy, 1998.

[CHO97] Chongstitvatana, P., "Post-processing optimization of byte-code instructions by extension of its virtual machine", Proc. of 20th Conf. of Electrical Engineering, Thailand.

[HEN] Henessy, J. and Nye, P., "Stanford Integer Benchmark", Stanford university.

[HEN96] Hennessy, J. and Patterson, D., Computer architecture: A quantitative approach, 2ed., Morgan Kaufmann Pub., 1996, figure D-15, p. D-19.

[KOO89] Koopman, P., Stack computers: the new wave, Ellis Horwood, 1989.

[MGH98] McGhan, PicoJAVA, IEEE computer 31(10) 1998.

[PAT82] Patterson, D. and Sequin, C., "A VLSI RISC", Computer, September, 1982.

[PIC96] Picojava, 1996, http://java.sun.com.

[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.

Chapter 10

Memory System Design

This chapter discusses the memory system design. We discuss the memory basics, how a memory module is organised. The hierarchical of memory, which is one of the most important aspects of a high performance computer system today, is introduced. The high-speed memory, the cache memory design is explored. The operating system services, which provide logical memory space, have a strong implication on the memory system design. The relationship between processors and operating systems are discussed. We conclude the chapter with a discussion of the memory technology, its rapid changes and the future of technology.

Memory basics

There are many types of memory in a computer system. The major type of semiconductor memory is random-access-memory (RAM). We will discuss the memory technology topic in the later section. A memory module consists of an array of memory cells. A memory cell can store one bit of information. To read or write a memory cell, it must be selected (addressed) and the control signal (read/write) is asserted. The data can be read off or written into the cell via the data line.

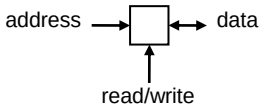


Figure 10.1 a memory cell

A memory module is built on an array of memory cells. The most widely used type of RAM is *dynamic* RAM (DRAM). A DRAM is made with cells that store data as charge on capacitors. It requires periodic refreshing of the cell's content. Fig. 10.2 shows a typical organisation of a 16 Mbit DRAM (4M × 4 bits). The memory array is organised as four arrays of 2048 × 2048 elements. The address lines supply the address A_0 .. A_{11}. They are fed into row and column decoders (2^{11} = 2048). The row and column addresses are multiplexed to reduce the number of pin of the memory package. The row-address-strobe (RAS) and the column-address-strobe (CAS) signals provide the control to the memory chip. The circuits on the chip included refreshing logic and input/output (I/O) lines interface to the external bus. A number of these typical chips are used to build up a larger memory for a computer system.

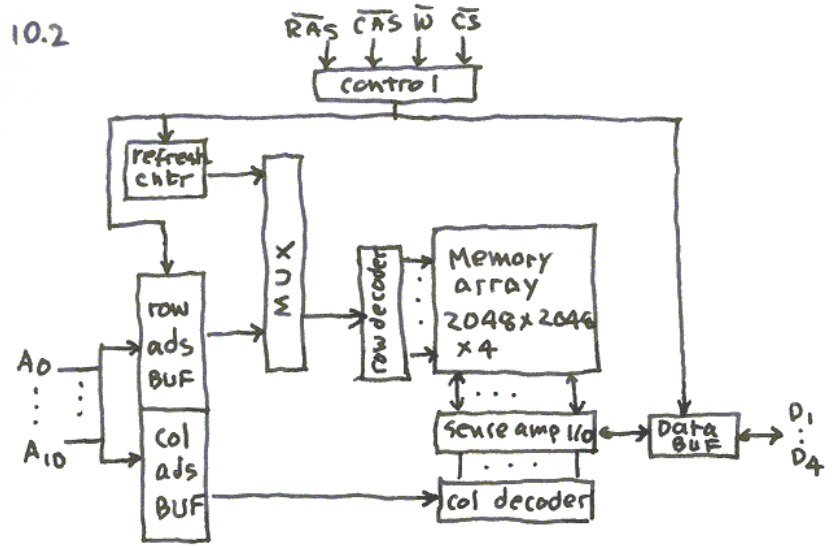


Figure 10.2 a typical 16 Mbit DRAM (4M × 4 bits)

Memory hierarchy

There are many types of memory in a computer system. The range spanned the memory in the control unit, the processor, the high-speed memory (the cache), the main memory, and finally in the secondary storage (the disk cache). We discuss each of them in turn. Fig.10.3 shows a typical memory hierarchy.

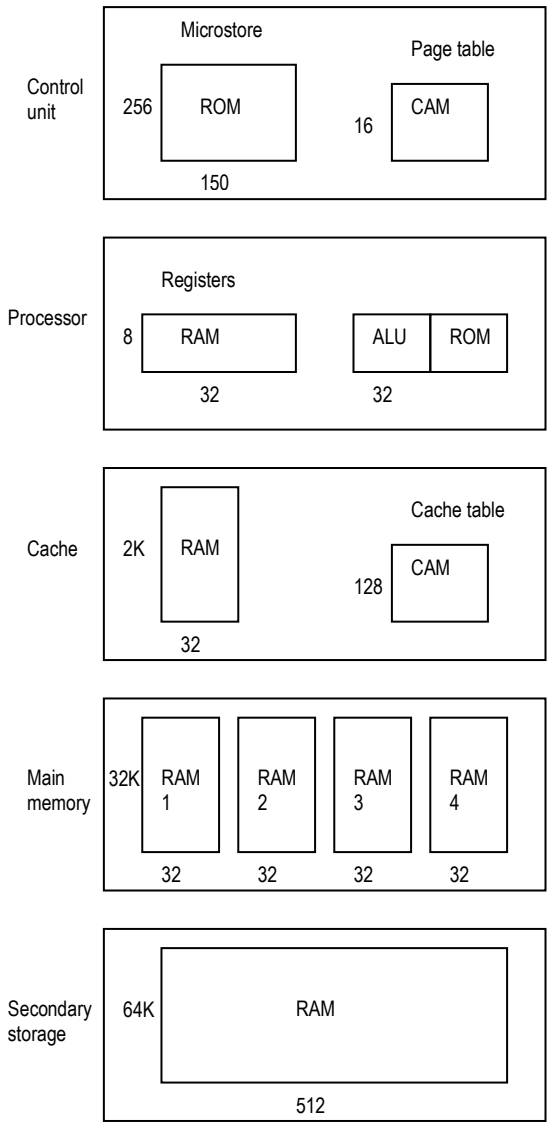


Figure 10.3 a memory hierarchy in a computer system

The control unit contains a micromemory storing the microprogram. It is a read-only-memory (ROM) (sometimes it may be a writable micromemory, in that case, it is a high-speed RAM). The page table, which is used to map virtual address (explained in the later section), contains the content-addressable-memory (CAM). A CAM is a special kind of memory. Rather than retrieving the content by the address lines, the content of a CAM is retrieved by the "association" of the pattern of data. The query such as "Is there any number 105 stored in this memory?" demonstrates the principle of the CAM. A CAM is generally more expensive than a RAM.

Within a processor there are registers. The registers have very fast access time. They are used to store temporary values during computation. The ALU has a ROM storing many constant values used in the floating-point calculations such as rounding operations.

A cache memory is a high-speed memory connecting a processor to a main memory. Because the speed of a processor in much faster than the speed of a main memory (at least by a factor of 10). The cache memory is matched to the processor speed. It is much smaller than the main memory, but using a scheme of storing the recently used data and instructions enables a cache memory to behave as the larger main memory. The cache table is a CAM. Its use will be explained in the later section.

The main memory is usually organised as one large unit. It can also be organised as a number of parallel memory units. Each unit can be addressed independently. The figure shows a four-way parallel main memory with 32-bit words. This improves the memory bandwidth.

The next level in the hierarchy is the secondary storage, usually it is the magnetic disk. The secondary storage has an access time much slower than the main memory (a disk is at least 10,000 times slower than a RAM). Using the principle of a cache memory, the disk cache, a large RAM, is as intermediate backup to the main memory. The goal is to buffer large amount of data to and from the main memory and the secondary storage. This arrangement improves the performance and reduces the system cost. The figure shows 64K of 512-bit words.

The cost of a memory varies with its speed. A multi-level memory hierarchy provides a large amount of memory that is not expensive. If the hierarchy is properly matched to the patterns of addresses generated by programs run on the system, its effective speed can match the processor speed. The address patterns

generated by programs are an important factor in the design of a memory system and will be discussed further in the later section. Figure 10.4 shows a typical parameters of memory hierarchy.

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk
Typical size	< 1 KB	< 4 MB	< 4 GB	> 1 GB
Technology	CMOS BiCMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (ns)	2–5	3–10	80–400	5,000,000
Bandwidth (MB/sec)	4000–32000	800–5000	400–2000	4–32
Managed by	Compiler	Hardware	OS	OS/User
Backed by	Cache	Main memory	Disk	Tape

Figure 10.4 a typical memory hierarchy (from [HEN96] p.41)

Interleaved memory

The constraint of a von Neumann architecture is that a single memory module of conventional design can access no more than one word during each cycle of the memory clock. There are several ways to increase memory bandwidth (bits/bytes per second).

- reduce cycle time
- increase word size
- concurrent access (banking, interleaving)

Reduction of the cycle time can be achieved by using a faster memory, which is more expensive. Increasing the word size will increase the number of connection between memory and processor. In term of the number of pin on the package, increasing the number of wires implies increasing the number of pin. The last alternative, *interleaved memory*, is achieved by arranging multiple memory arrays into parallel units, each of which can be accessed independently. All units are addressed at the same time, hence the consecutive locations can be accessed with zero delay after the first data is available.

Cache

A cache memory is a high-speed memory connecting a processor to a main memory. It is much smaller than the main memory but it is faster. Its advantage is that the average memory access time is nearly equal to the speed of the fastest memory, whereas the average unit cost of the memory system approaches the cost of the cheapest memory.

A cache memory stored the most frequently used values from the memory. This is possible because the principle of locality of references [DEN68]. The patterns of addresses generated by programs run on a system exhibit two kinds of locality:

1. *Temporal*, the values that have been accessed will be accessed again in the near future.
2. *Spatial* (array, vector, code segment), the values near the location of the recently accessed will be accessed in the near future.

Temporal locality

Cache memory stored the most frequently used values from the memory. The access pattern has "temporal locality", the locations in the memory may be spatially separated but the cache memory stored them together when they are accessed.

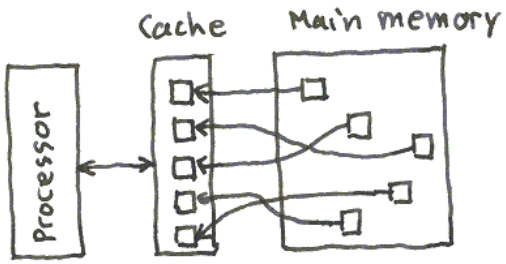


Figure 10.5 cache memory stored values that "temporally related"

Working set of an application is the amount of memory the application required in a period of time. It varies with time.

Cache performance

When the processor attempts to fetch a word from main memory, the word is supplied from the cache if it is resident there. If not the word is supplied from main memory, but the cache is also loaded with a block of words including the one requested. Thus, subsequent accesses to the same word can be handled at the cache speed.

Let *h*, the cache *hit ratio*, be the fraction of the references to main memory that can be found in the cache.

$$T_e = hT_C + (1-h)\,T_m$$

where T_e the effective cycle time, T_c the cycle time of cache memory, T_m the cycle time of main memory. The speed up due to cache is

$$S_C = T_m / T_C$$

Let express the speed up in term of hit ratio

$$S_C = \frac{T_m}{T_C} = \frac{T_m}{hT_C + (1-h)T_m} = \frac{1}{1+h\left(\frac{T_C}{T_m}-1\right)}$$

since $T_C / T_m < 1$ we can write

$$S_C = \frac{1}{1-\left(1-\frac{T_C}{T_m}\right)h}$$

if $h = k/k+1$ then

$$S_C = \frac{1}{1+\frac{k}{k+1}\left(\frac{T_C}{T_m}-1\right)} = \frac{1}{\frac{k}{k+1}\,\frac{T_C}{T_m}+\frac{1}{k+1}} = \frac{(k+1)\,T_m}{T_m + kT_C} < k+1$$

Thus, if h = 1/2 we can not achieve a speed up of more than 2. The figure shows maximum S_c versus *h*.

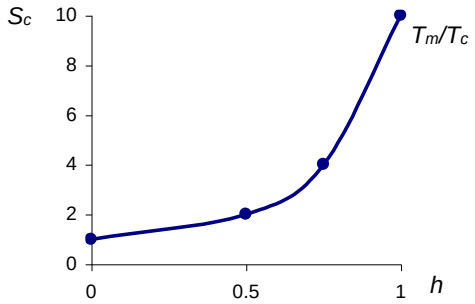


Figure 10.6 maximum possible cache speed up versus hit ratio

Example We want to compare two size of cache memory, 512 K bytes and 1 M bytes. Let the average hit ratio of 512 K bytes cache be 0.93 and the average hit ratio of 1 M bytes cache be 0.97. Let $T_c / T_m = 0.12$.

$$S_c^{512} = 1 / (1 - 0.88 \times 0.93) = 5.5$$

$$S_c^{1M} = 1 / (1 - 0.88 \times 0.97) = 6.85$$

Thus, adding 512 K bytes to the cache achieved a system speed up improvement of

$$\frac{S_c^{1M} - S_c^{512}}{S_c^{512}} = 0.24$$

This 24 percent improvement could be a good return on relatively small investment.

Example We want to assess how much a cache contribute to the performance of a system. Assume a cache miss penalty is 50 clock cycles, all instructions take 2 clock cycles (not take into account the caches miss), the miss rate is 2%. There is an average of 1.33 memory references per instruction. What is the impact on performance when taking cache behaviour into consideration?

We start with the definition of performance

Performance = how fast a processor finishes its job

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

Taking the behaviour of memory into account

execution time = (CPU execution cycle + memory stall cycle) × cycle time

memory stall cycle = number of instruction used × memory reference per instruction × miss rate × miss penalty

Threrefore

execution time = number of instruction used × (cycle per instruction + memory reference per instruction × miss rate × miss penalty) × cycle time

= n.o.i. × (2 + (1.33 × 2% × 50) × cycle time

= n.o.i. × 3.33 × cycle time

Therefore the execution time increases with CPI from 2 (no cache miss) to 3.33 with a cache that can miss. Without a cache at all, CPI would be 2 + 50 × 1.33 = 68.5, a factor of 30 times longer!

Cache organisation

A cache memory stored some part of the main memory. The main memory can be viewed as "blocks". A cache stored a number of these blocks, which are indexed by part of the address bit. The size of the block varies. For any size larger than one, the lower address bits are used as "offset" to indicate the required word within a block. The relationship between the size of the block, the size of the cache, the organisation of the cache and the hit ratio are complex. The larger cache size has a higher hit ratio. The cache contains the memory for storing the address, called "tag", and the memory for storing the blocks.

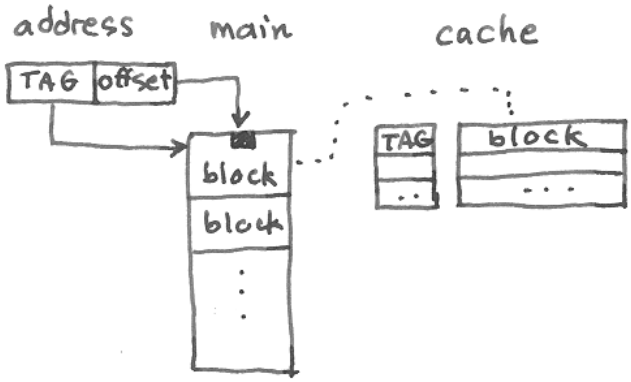


Figure 10.7 the relationship between main memory and cache

The activities in a cache consist of the request for reading and writing. In response to those requests, there are these possible events: load hit, load miss, store hit, and store miss. A cache is classified according to its organisation. We will discuss three major organisations: fully associative cache, direct map cache, and set associative cache. Mostly they differ in the way they response to the request:

1. Where can a block be placed in the cache?
2. How is a block found if it is in the cache?
3. Which block should be replaced on a miss?
4. What happens on a write?

Fully associative

In a fully associative cache, the tag memory is made of CAM, therefore a block can be placed in any slot in the cache. The search for address matching is done with all tags in parallel using the retrieval by association. If a conventional memory (RAM) is used, it will require scanning every address. If there is no ordering among the content, scanning will take O(n). If there is ordering, using binary search will take O(n log n). An associative memory takes just O(1) to find the required content.

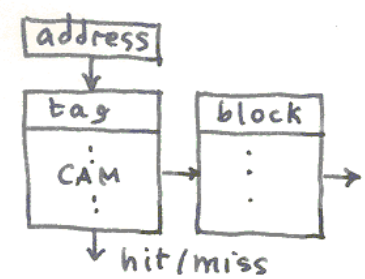


Figure 10.8 a fully associative cache

However, the CAM is expensive and the fully associative cache is used mainly in places where the speed is important and the small cache size is appropriate. To increase the cache size without increasing the tag size (the tag is made of CAM), the block size (made of RAM) can be increased.

Direct map

A direct map cache uses RAM instead of CAM to store tags. The lower bits of address are used to index the block. Therefore a block has a unique place in the cache. The addresses that have the similar higher bits will be placed in the same slot. This reduces the effectiveness of the cache, as some addresses will collide. Due to its simplicity, a direct map cache can be implemented with efficiency (it can be very fast).

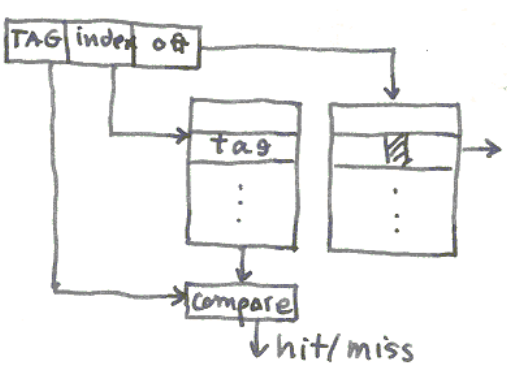


Figure 10.9 a direct map cache

Example to understand how a cache stores the tags, let us follow a simple exercise. Assume a cache has 8 slots with a block size 1. A sequence of memory requests is issued by the processor. Suppose the address sequence is 0, 1, 3, 6, 9, 13, 14, 3, 18, 19, 13. The following diagram shows the state of tags.

address			Tag_{t+3}	Tag_{t+7}	Tag_{t+10}
0	8	16	0	0	0
1	9	17	0	1	1
2	10	18			2
3	11	19	0	0*	2
4	12	20			
5	13	21		1	1*
6	14	22	0	1	1
7	15	23			
address sequence:			0,1,3,6	9,13,14,3*	18,19,13*

Figure 10.10 the state of tags after the sequence of accesses

The tag is calculated by (address / 8) and the index to the slot is (address mod 8). The address marked with * are the cache hits. There are two hits in this example.

Set associative

To improve the performance of a direct map cache, the set associative cache uses a number of direct map caches in parallel, called a set, *k*. The search of the matching address is done on *k* tags in parallel. The mechanism requires only k set of comparators and a multiplexor to select data from *k* sets. This improves the hit ratio as it reduces the chance of collision which different addresses map to the same slot. Having a set of caches introduces another consideration, which set to be replaced when a miss occurs? This question is settled with cache replacement policy.

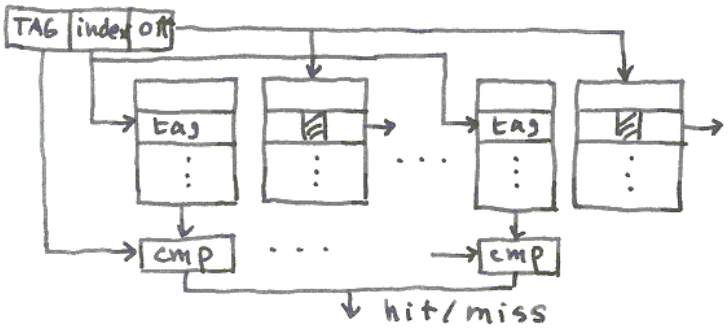


Figure 10.11 a set associative cache

Replacement policy

On a load miss, the value must be read from the main memory and also the whole block must be updated into the cache. There is no choice which block to be replaced in the direct map cache. The selection is done in hardware using the lower address bits to index the slot. For the fully associative cache and the set associative cache there are two major policies:

1. *Random* replacement, the block to be replaced is randomly selected so that it is uniformly distributed in the cache.
2. *Least-Recently-Used* (LRU), this policy replaces the block that has been unused for the longest time.

The random replacement policy is easy to implement in hardware, requiring only a random number generator. The LRU policy must keep track of the number of access to each block hence it is more complicate. Nearly all caches in commercial production use LRU.

Write policy

Reads dominate cache accesses. All instruction accesses are reads. For data accesses, writes constitute 7% of the overall memory traffic and about 25% of data cache traffic. Reads can be made fast. A block can be read at the same time as the tag is read and compared. If it is a miss the value read can be ignored.

It is not the case for writes. Because the tag checking cannot be done in parallel with the memory access (write cannot be undone), writes normally take longer than reads. On a write request, there are two options:

1. *Write through*, the information is written to both the block in the cache and to the block in the main memory.
2. *Write back*, the information is written only the cache. The modified cache block will be written to main memory only when that block is replaced.

There is a bit associated with each block in cache called the *dirty bit*. This bit is set when the content of that block is modified. When the block is going to be replaced, the dirty bit is examined, if it is clean, it is not necessary to write this block to the main memory. This reduces the frequency of writing back to the main memory.

Address Trace

To measure the performance of a cache memory, a miss ratio is measured from an address trace. Some problems are presented:

1. the workload on the trace may not be representative
2. the initialisation transient may grossly affect the evaluation
3. the trace may be too short to obtain accurate measurement.

The length of the trace is important for accuracy. Also the concern about initialisation of cache (should or should not take into account for cache miss). If cache size is 1,000 blocks, assuming 1 byte per block, and the miss ratio is 1%, for a miss to occur once for every block requires the trace length of 100,000 addresses just to initialise the cache. An empirical formula for the trace length is:

$$\text{Trace length} = \text{cache size}^{1.5}$$

This formula implies that for each quadrupling of cache size, the trace length increases by a factor of 8. A typical simulation run to collect the address trace covers hundreds of milliseconds at most.

Improving cache performance

The performance gap between processors and memory is increasing every year. During 1980-1986 the processor performance increased 35% per year and 1987-2000 it increased 55% per year while the memory performance increased during 1980-2000 only 7% per year. In the year 2000, the processor is about 500 times faster than memory. The cache is becoming more and more important to bridge this gap. The average access time of a memory system is.

$$T_{average} = T_{hit} + \text{miss rate} \times \text{miss penalty}$$

All three factors T_{hit}, miss rate, and miss penalty are considered in improving cache performance.

To reduce miss rate the larger block size can be used. The larger block size takes advantage of spatial locality. However, increasing block size will increase the miss penalty. Another way to reduce miss rate is to increase associativity. A rule of thumb for cache is that a direct map cache of size N has about the same miss rate as a 2-way set associative cache of size N/2. An 8-way set associative is as effective in reducing misses as fully associative cache of the same size. Using two separate caches for instruction and data instead of a unified cache can be beneficial. A processor relies on prefetching instructions to fully use the pipeline. Having a separate instruction cache reduces the interference from data access.

The miss penalty can be reduced by using a second-level cache. In designing a cache, it is always a question of whether to make the cache larger or to make it faster. By adding another level of cache between the first cache and the main memory, the first cache can be fast while the second cache can be large. The second cache will reduce the number of traffic to the main memory. It is, in essence, reducing the miss penalty.

The hit time can be reduced in two ways. First, a simpler and smaller hardware is faster, therefore the simple organisation of cache such as a direct map cache can be faster than a complicate cache. Second, use an on-chip cache. Because the advancement of microelectronics, the number of logic on a chip has been

increasing. It has been possible to include a cache on the same die as a processor. The on-chip cache is much faster than an off-chip cache due to shorter paths and smaller delay in signal paths. However the size of an on-chip cache is limited.

For more detailed discussion of high performance cache design, there are numerous research papers on the subject, for example [TAE98] [TAE98a]. The thesis describes the design of a cache controller for a high performance processor. Many other organisations of cache have been introduced to cope with the changing of the workload that emphasis multimedia applications [CHI00].

Example The on-chip cache of Intel 486 is a 4-way set associative, the line size is 16 bytes, total cache size 8Kbytes To simplify the hardware, it used pseudo-LRU having 3 bits: r0, r1, r2. The replacement policy works as follows. Let the set be s0, s1, s2, and s3. The most recently access sets the bit using these rules:

if the set is s0 or s1 set bit r0
if the set is s0 set bit r1
is the set is s2 set bit r2

Then, the replacement follows these rules:

if r0 r1 = 00 replace s0
if r0 r1 = 11 replace s1
if r0 r2 = 10 replace s2
if r0 r2 = 11 replace s3

Virtual Memory

An operating system provides a multi-task capability. It performs three types of scheduling: accepting a number of processes, switching between processes, and handling I/O requests. One important function of an operating system is *memory management*. Many programs, including the operating system itself resided in main memory. The allocation and reclamation of memory for these programs must be done dynamically by the operating system.

The process in memory contains instructions and data. When a process is swapped in and out, the addresses of these instructions and data will be changed. A process is not likely to be loaded into the same place each time it is swapped. The concept of logical address and physical address is used to solve this problem. A *logical address* is expressed as a location relative to the beginning of the program. A *physical address* is an actual location in main memory. A part of

processor hardware is designed to support the mapping between logical to physical addresses.

Paging

To facilitate memory allocation, main memory is divided into small fixed-size blocks, called *pages*. A number of pages are allocated to each process on demand. To translate a logical address to a physical address, the operating system maintains a *page table* for each process. The page table shows the location for each page of the process. A processor uses the page table to produce a physical address.

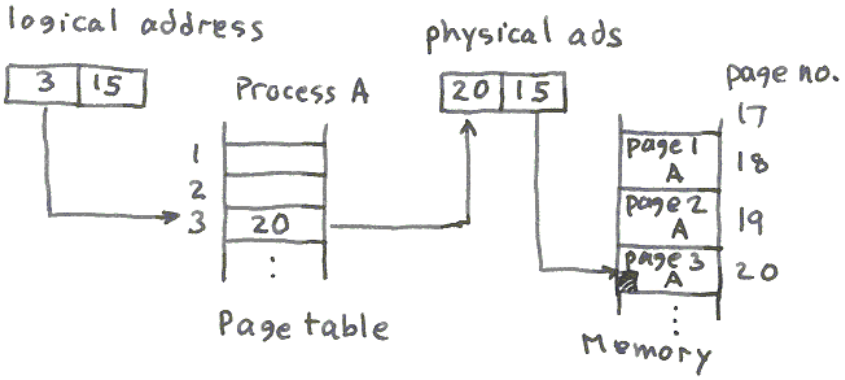


Figure 10.12 translation of logical to physical address

With *demand paging* it is possible for a process to be larger than main memory. A process executes in main memory referred to as *real memory*. A user perceives a much larger memory, the size of disk, referred to as *virtual memory*.

Parameter	First-level cache	Virtual memory
Page size	16–128 bytes	4096–65,536 bytes
Hit time	1–2 clocks	40–100 clocks
Miss penalty (Access time) (Transfer time)	8–100 clocks (6–60 clocks) (2–40 clocks)	700,000–6,000,000 clocks (500,000–4,000,000 clocks) (200,000–2,000,000 clocks)
Miss rate	0.5–10%	0.00001–0.001%
Data memory size	0.016–1MB	16–8192 MB

Figure 10.13 Typical ranges of parameters for caches and virtual memory (from [HEN96] p.441)

Figure 10.13 shows the difference between first-level caches and virtual memory. Comparing the two, the difference in magnitude is about 10× to 100,000×. A cache miss is 8–50× costly as a cache hit but a page fault is 6,000–20,000× as costly as a page hit. A cache miss latency is in the range of hundred of clocks but a page fault latency is in range of millions of clocks. A processor can execute a fair number of instructions during this time.

Address translation

The address translation requires accessing the page table. The page table resides in main memory. If it is large, accessing a page table may cause page fault. The worst case requires two memory accesses to get the data. A special cache, a *translation lookaside buffer* (TLB), caches page table entries, hence improving the speed of address translation. The TLB functions the same way as an ordinary cache and contains page table entries that have been most recently used. The principle of locality of references also applied to TLB, if the accesses have locality, then the address translation of the accesses must also have locality.

The operation of paging and TLB is as follows [FUR87].

Request access to a page
CPU checks TLB
if the page table entry is not in TLB then
  access page table
  if page is not in memory then generate *Page Fault*
    else update TLB
CPU generates physical address

Page Fault
OS call routines to read the page from disk
CPU activates I/O hardware
If memory is full then perform Page Replacement
page table update
restart request to access a page

The virtual memory mechanism (TLB) must interact with the cache system. When a memory access occurs the TLB is looked up, if the matching page is present, the real (physical) address is generated. If TLB does not contain the page, the entry is accessed from the page table. Once the real address is

generated the first-level cache is consulted, if the cache is a hit, the cache supplies the word. If it is a miss, the word is retrieved from main memory.

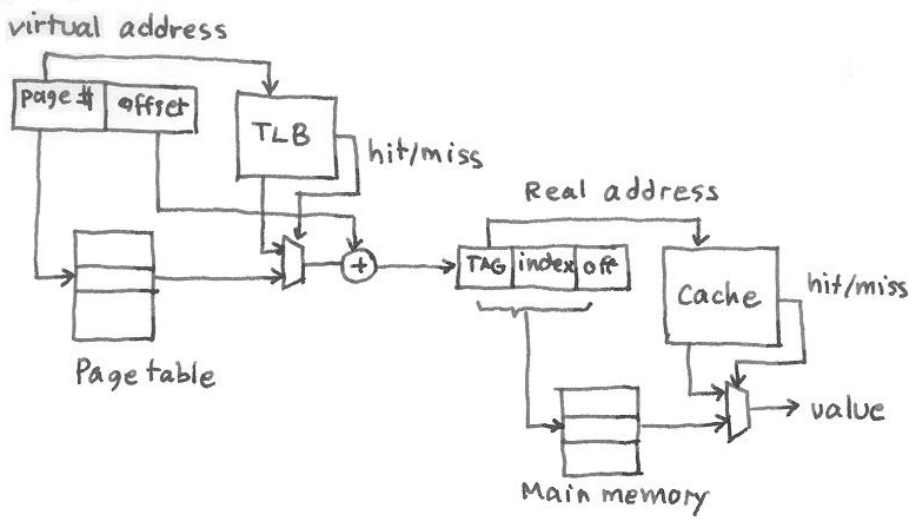


Figure 10.14 Translation lookaside buffer and cache operation

Address translation can easily be on the critical path and hence determining the clock cycle of a processor. The TLB is usually smaller and faster than a first-level cache. The TLB access is sometimes pipelined to improve its speed.

TLB size	32–8192 bytes
Block size	4–8 bytes (1 page table entry)
Hit time	1 clock
Miss penalty	10–30 clocks
Miss rate	0.1%–2%

Figure 10.15 Typical parameters for a translation lookaside buffer

Page Replacement

A page fault is different from a cache miss as the page fault is very costly and the latency is very long. During that time, a processor can perform a significant amount of processing. This processing power can be used to reduce the miss rate. Another difference is that virtual memory supports a multi-task environment. The behaviour of tasks swapping in and out of main memory are

very dynamic compared to the behaviour of a cache miss that occurs in a single thread of execution over a shorter period of time.

When a page fault occurs, a new page must be loaded from the secondary memory, usually a disk. The *trashing* behaviour describes the situation where there are excessively high traffic between main memory and the disk. Trashing causes significant impairment of system performance. The replacement policy is similar to that of a cache, the LRU. However, the question arises on how many pages to be allocated to a process? The answer is to allocate as many pages as the process needs at a given time. This requires the notion of a *working set*, the footprint of a program execution over a short period of time.

How to find working set

W(*t*, *w*) the working set at time *t* for window *w*. It is the set of pages referenced in the last *w* seconds at time *t*. The following steps allocate pages according to the working set of a program:

1. when page fault, add a new page to the working set
2. from set of pages not referenced. within *w*, delete the LRU page otherwise let the set grow
3. if two or more pages not referenced within *w*, delete two LRU pages, *w* is measured by process time.

Page-fault frequency method

This method is based on the observation that a high page fault signals the change in phase to a new working set

1. select a threshold Z
2. when page fault, estimate frequency. $f = 1/(t_1 - t_0)$
3. f > Z assume change phase, add page to the working set
4. f < Z assume stable. add new page, remove the old page (LRU)
5. f < Z over some period, assume stable and dead pages, reduce working set and delete unreferenced. page (LRU)

How to allocate the number of pages to a program

The optimal is to get least page-fault rate. In a muti-task environment there are many processes competing for resources. The allocation must consider all processes. Let $R_i(x_i)$ be fault rate of process *i* with x_i be the memory allocation. The optimality can be achieved in the following sense:

$$\frac{d}{dx}R_i(x_i) = \frac{d}{dx}R_j(x_j)$$

The fault rates for each process for their respective allocation are equal. This implies that an optimum allocation depends on *fault rate derivatives*. Figure 10.16 compare the working set method and the page-fault frequency method.

	working set	page-fault frequency
assumption	immediate future will be like the recent past	transient between two program phases signaled by higher than normal fault rate
implementation	difficult because of sliding window	observable quantities use hardware logging

Figure 10.16 comparison of the working set method and the page-fault frequency method

Memory technology

Memory system design is becoming increasingly important as a computer system performance improved. There are three major reasons for this. First, the new generation of microelectronics technology improves speed and reduces power consumption. However, the data transfer rate on a circuit board is independent of technology scaling and remains at about 1 ns.

Second, the use of parallelism in processor design demands higher data transfer rate but data parallelism cannot be exploited because the number of chip-to-chip connections (the packaging) is limited.

Third, as memory is larger, the time for address decoding is at least logarithmic increased, hence the memory becomes slower. As a result, while processor performance improves exponentially according to Moore's law, memory system performance does not.

History

Before 1960s, computer memory systems consisted of cathode-ray storage tubes, ferrite cores, and thin magnetic films [ECK97]. The first semiconductor memory was a six-transistor SRAM (static random access memory) cell, which is now

used mostly for cache and battery-backup memory. The one-transistor dynamic memory was invented in 1968 [DEN68a] [DEN84]. The DRAM became very successful, because of its low bit cost and high density. The DRAM has dominated the computer main memory market.

DRAM operation

A DRAM is characterised by its access time and its cycle time. An *access* time is the time between the request (the address is presented) and the time when the data is ready. A *cycle* time is the time between two consecutive memory operations. This is longer than the access time due to the electrical characteristics of the memory circuits.

The memory cell consisted of storage capacitor and selection transistor. Its binary state is represented by the amount of charge it holds. The storage capacitor is implemented as a MOS (Metal Oxide Semiconductor) capacitor. The memory array composed of cross-point array of memory cells. Its operational scheme is as follows.

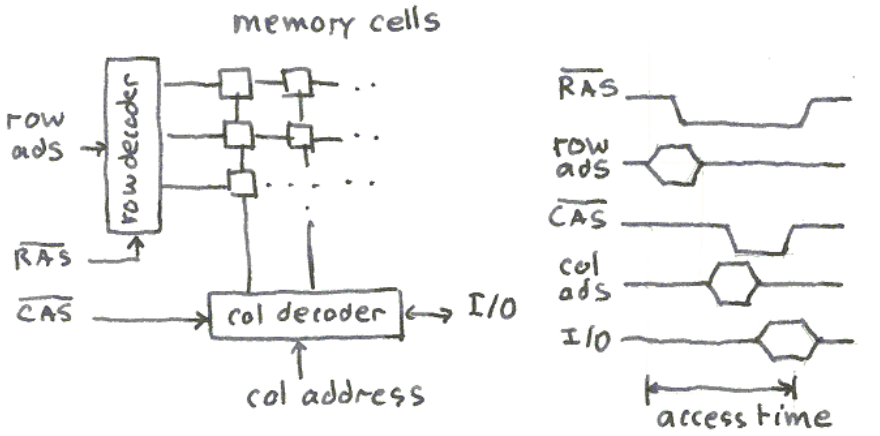


Figure 10.17 read operation of DRAM

For read operation, the RAS (row address strobe) latches the row address and decoding. The CAS (column address strobe) decodes column address and multiplexes data. The row and column address inputs are usually multiplexed to minimize the number of pins in the DRAM package. Since the read operation is

destructive, the data must be written back to the memory cells in parallel with the column access. This causes delay in RAS cycle time. The array then must be precharged for the next memory access operation.

The write operation is similar, only difference is the I/O bit is forced into a new logic state. In other words, writing to a DRAM cell is similar to writing back to the cell with a new logic state.

The limitation in performance of DRAM involves two aspects: the latency and cycle time in the row access and the data rate in the column access. The performance in row access is caused by the resistance capacitance time constant in both charging and discharging the dynamic memory cell and array. The cycle time in column access determines the data rate. Widening the width for the chip-to-chip connections increases the cost as the number of I/O drivers and packing pins increases.

High-speed DRAM development

The research and development in late 1980s results in chips access time in the low 20 ns range and a column access time in the low 10 ns range, using CMOS (Complementary Metal Oxide Semiconductor) technology [LUN89] [TAK90].

For faster data rate, EDO (extended data out) DRAMs improve the data rate for column access by adding an extra pipeline stage in the output buffer. A typical 4-M bit EDO DRAM with × 8 I/Os operating at 33 MHz column access cycle can provide a peak data rate of 266 Mbytes/second per chip.

In the 16-Mbit generation, SDRAM (synchronous DRAM) employed a high-speed synchronous interface. The data rate was improved using pipelining the data path or prefetching data bits over wider data lines. The performance was also improved by interleaving multiple banks of memory arrays (normally two) on the same chip. The peak data rate for a 16-Mbit SDRAM with × 16 I/O operating at 66 MHz is 1.1 Gbps (133 Mbytes/second) per chip. JEDEC (the Joint Electron Device Engineering Council) has standardized both EDO DRAM and SDRAM.

For faster random access, the performance near SRAM level is achieved by integrating a small amount of SRAM as cache or using multiple banks of DRAM. These designs are: EDRAM (enhanced DRAM), CDRAM (cached DRAM), and MDRAM (multibank DRAM).

Rambus DRAM [KUS92] uses a packet-type memory interface to realise a peak data transfer rate of 500 Mbps per data I/O pin (4 Gbps, or 500 Mbytes per chip) with a 250 MHz clock. The improved version, called Concurrent Rambus, realises a peak data rate of 600 Mbps per data I/O pin.

There are many types of memory that designed for specific applications, for example VRAM, WRAM, and 3DRAM. Video RAM (VRAM) realised concurrent dual port access by multiplexing the data in the DRAM array. The internal data rate, with 4,096 full transfer, operating at 10 MHz, is 41 Gbps (5.1 Gbytes/second). It is aimed for video application using video screen refresh data. Window RAM (WRAM) improves graphics operations in a GUI environment. It simplifies VRAM design and it has additional functions such as *bitblt* (bit-block-transfer) and block write for graphics applications. 3DRAM has been especially designed for processing 3D graphics applications. The read-modify-write operation, which occurs frequently in 3D graphics can be achieved with one write operation.

DRAM Trend

The performance of a computer system can be associated with memory system bandwidth and memory system capacity [KAT97].

$$\begin{aligned} \text{performance} &= k_1 \text{ (memory system bandwidth)} \\ &= k_1 \text{ (DRAM data rate) } N_{dram} / N_{bank} \end{aligned}$$

$$\begin{aligned} \text{performance} &= k_2 \text{ (memory system capacity)} \\ &= k_2 \text{ (DRAM density) } N_{dram} \end{aligned}$$

where N_{dram} is the number of DRAMs in the system, N_{bank} is the number of banks sharing the same data bus, and k_1 and k_2 are coefficients. By dividing the two equations.

$$\text{DRAM data rate} = N_{bank} (k_2/k_1) \text{ (DRAM density)}$$

This equation implies that the DRAM needs a higher data rate as its density increases. The coefficient $N_{bank}(k_2/k_1)$ is called *full frequency* [PRZ96], it depends on the application. The empirical numbers are 100 Hz for PC graphics, 10 to 20 Hz for PC main memory, and less than 10 Hz for servers and workstations. The driving forces of DRAM changes are low-end and graphics

applications, where the data rate requirement per DRAM density is higher. The transition to high-speed DRAM occurred not in the high end of the market but in the low end. The high-speed memory DRAM could provide smaller memory granularity for a given bandwidth requirement.

There are three major candidates for the next generation of high-speed DRAMs.

1. SDRAM-DDR (SDRAM double data rate), which uses a synchronous RAS and CAS similar to SDRAM. The data transfer is at both edges of teh clock. A 16-Mbit SDRAM-DDR with ×16 I/O operating at 100 MHz clock (200 MHz data rate) can provide 3.2 Gbps.
2. Rambus DRAM, the data rate can achieved a peak of 13 Gbps per chip due to 400 MHz clock (800 MHz data rate) and 16-bit bus width.
3. SLDRAM (Ramlink IEEE standard 1596.4) [GIL97], It builds on the features of SDRAM-DDR by adding an address/control packet protocol. The internal DRAM address and control paths are decoupled from the data interface to achieve higher bandwidth.

Beyond high-speed DRAM lies the merging between logic and memory, using the processing at the memory cell. Putting RAM and processor together can achieved a very high bandwidth, such as one proposed by Patterson [PAT97].

References

[CHI00] Chiueh, T., and Pradhan, P., "Cache memory design for Internet processors, IEEE Micro, Jan./Feb. 2000.

[DEN68] Denning, P., "The working set model for program behavior", Communications of the ACM, May 1968.

[DEN68a] Dennard, R., Field Effect Transistor Memory, US Patent 3387286, 1968.

[DEN84] Dennard, R., "Evolution of the MOSFET dynamic RAM − A personal view", IEEE Electron Devices, no, 11, Nov. 1984, pp.364-369.

[ECK97] Eckert, J., "A survey of digital computer memory systems", Proc. IEEE, vol. 85, no. 1, Jan. 1997, pp.184-197. (Originally in Proc. Inst. Radio Engineers, vol. 41, Oct. 1953, pp. 1393).

[FUR87] Furht, B., and Milutinovic, V., "A survey of microprocessor architectures for memory management", Computer, March 1987.

[GIL97] Gillingham, P. and Vogley, B., "SLDRAM: High-performance Open-Standard Memory", IEEE Micro, Nov./Dec. 1997, pp. 29-39.

[HEN96] Hennessy, J., and Patterson, D., Computer architecture: A quantitative approach, 2 ed., Morgan Kaufmann, 1996.

[KAT97] Katayama, Y., "Trends in semiconductor memories", IEEE Micro, Nov./Dec. 1997, pp.10-17.

[KUS92] Kushiyama, N. et al., "500 Mbytes/s Data rate 512 Kbits × 9 DRAM using Novel I/O Interface", Dig. Tech. Papers, 1992, Symp. VLSI Circuit, June 1992, pp.66-67. http://www.rambus.com

[LUN89] Lu, N. et al., "A 22-ns 1-Mbit CMOS high speed DRAM with address multiplexing", IEEE J. Solid-state circuits, vol. 24, Oct. 1989, pp.1196-1205.

[PAT97] Patterson, D. et al., "A case for intelligent RAM", IEEE Micro, Mar./Apr. 1997, pp.34-44.

[PRZ96] Przybylski, S. et al., "SDRAMs ready to enter PC mainstream", Microprocessor report, vol. 10, no. 6, May 1996.

[TAK90] Takai, Y. et al., "A 23-ns 1-Mbit BiCMOS DRAM", IEEE J. Solid-state circuits, vol. 25, Oct. 1990, 1102-1111.

[TAE98] Taechashong, Primas, A VLSI design of a load/store unit for a RISC processor, Master of Engineering thesis, Department of computer engineering, Chulalongkorn University, 1998. (in Thai)

[TAE98a] 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, pp. 244-248, Bangkok, March 25-27, 1998.

Chapter 11

Magnetic Disk

This chapter examines the most important type of secondary storage, the magnetic disk. Magnetic disks have dominated the secondary storage market since 1965. Magnetic disks are almost an integral part of all computer systems. It provides long-term, non-volatile storage for files and is used for virtual memory. The use of multiple disks to improve reliability and performance is discussed. Finally, the I/O functions that control data transfer between I/O and main memory is explained.

Disk basics

The read-write mechanism is based on the magnetic field produced by the read-write head and magnetic patterns are recorded on the disk surface. The organisation of data on the platter is a concentric set of rings, called *tracks*. Adjacent tracks are separated by *gaps*. This prevents errors due to misalignment of the head. The density is the amount of bits per inch on each track. Data are recorded in block-size chunk called *sectors*. Adjacent sectors are separated by inter-record gaps. A sector position is identified by its relative position to the control data recorded on the disk. The capacity of a disk is expressed as *areal density* (bit/inch^2)

$$\text{areal density} = \text{track/inch on a disk surface} \times \text{bits/inch on a track}$$

Example A disk format, Seagate ST506 is as follows. Each track contains 30 sectors of 600 bytes each. Each sector stores 512 bytes of data plus control information. The ID field is a unique identifier used to locate a sector. The SYNC byte is a special pattern that delimits the beginning of the field. The ID and data fields contain error-detecting codes (a cycle redundant code, CRC).

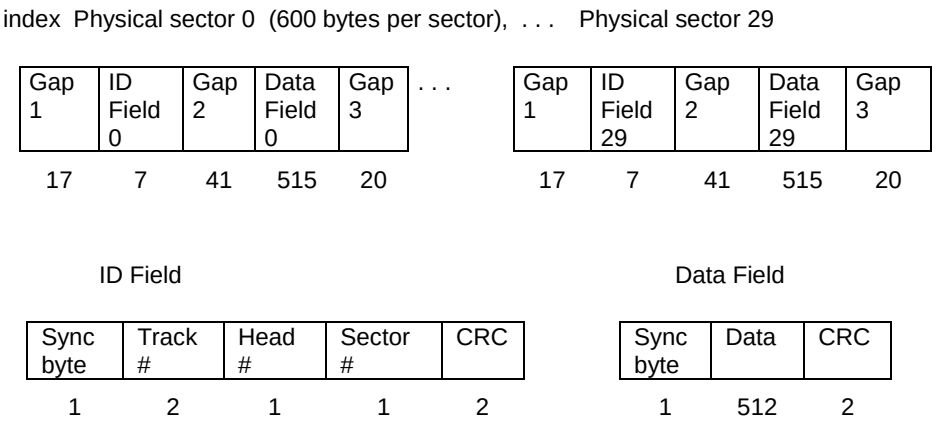


Figure 11.1 Winchester disk format Seagate ST506

Disk access time

The disk is rotating at constant speed. The head must be positioned on the desired track to read or write data. The time to move a head to the track is known as *seek time*. Then, the system waits for the desired sector to line up with the head, this time is known as *rotational latency*.

access time = seek time + rotational latency

After the sector lines up with the head, the transfer can take place. The *transfer time* is the time it takes to transfer a block of bits, typically a sector. It is a function of the block size, rotational speed, recording density and speed of electronics connecting the disk to a computer.

Example What is the average time to read or write a 512-byte sector of a typical disk? The advertised average seek time is 9 ms, the transfer rate is 4MB/sec., it rotates at 7200 RPM, and the controller overhead is 1 ms.

average disk access time = average seek time + average rotational delay + transfer time + controller overhead

9 ms + 0.5/7200 RPM + 0.5 KB/4 MB/s + 1 ms = 14.3 ms.

Disk Performance

The advances in disk technology improve disk performance. These advances include increased rotational speed, faster seek times, and higher data rates. Some other advances such as disk density or total drive capacity also impact the performance [NGS98]. We will discuss these advances.

Disk performance is measured by "total job completion time" for a complex task involving a long sequence of disk I/Os. The time for a disk drive to complete a user request consists of:

- command overhead
- seek time
- rotational latency
- data transfer time

Performance parameters

Command overhead – the time takes for the disk controller to process and handle I/O request – depends on the type of interface (IDE or SCSI), type of command read/write, use of drive's buffer. Typical value is 0.5 ms for buffer miss and 0.1 ms for buffer hit.

Seek time – the time for the head to move from its current cylinder to the target cylinder.

Settling time – the time to position the head over the target track until the correct track identification is confirmed. A typical seek time is 10 ms.

Rotational latency – In the past disk spins at the speed 3,600 rpm. Today the highest speed is 10,000 rpm and typically 5,400 rpm. representing the average latency 5.6 ms.

Data transfer time – It depends on "data rate" and "transfer size". There are two kinds of data rate : media rate and interface rate.

Media rate depends on recording density and rotational speed.

Example, a disk rotating at 5,400 rpm with 111 sectors (512 bytes each) per track will have a media rate 5 Mbytes per second.

Interface rate is how fast data can be transferred between the host and the disk drive over its interface. SCSI drives can do up to 20 Mbytes per sec. over each 8-bit-wide transfer. IDE drives with the Ultra-ATA interface support up to 33.3 Mbytes per sec.

Transfer time equals transfer size divided by data rate. The average media transfer time is 0.8 ms, the average interface transfer time is 0.4 ms.

Example The typical average time to do a random 4-K byte disk I/O is

overhead + seek + latency + transfer
= 0.5 ms + 10 ms + 5.6 ms + 0.8 ms
= 16.9 ms

Locality of access – Most I/Os are not random, the effect is that the real seek time is about one third of random seek time. Taking this into account the above example will be

overhead + seek + latency + transfer
= 0.5 ms + 1/3 * 10 ms + 5.6 ms + 0.8 ms
= 10.2 ms

Caching – With caching the mechanical component, i.e. seek and latency, are eliminated. Data transfer takes place at the interface data rate. Typical time to do 4K I/O becomes

overhead + transfer = 0.1 ms + 0.4 ms = 0.5 ms

Increase recording density

- bits per inch (bpi)
- track per inch (tpi)

Increase BPI

BPI is called "linear density", determines the number of sectors on a track. With "zoned recording", each zone the number of sectors per track is constant. BPI toward the outer diameter of a zone is somewhat lower than the BPI toward the inner diameter of the same zone. Increasing BPI affects a higher media data rate, puts constraint on rpm, has fewer head switches, and a bigger cylinder.

Higher media rate – media data rate = 2 pi × radius × bpi × rotational speed

Constraint on rpm – increasing bpi can push data rate beyond what the drive's data channel can handle. Today's disk electronics can handle up to 25 Mbytes per sec.

Fewer head switches – Switching to the next track on the same cylinder is called "track switch" and switching to the next track on the next cylinder is called "cylinder switch".

$$\text{average switch time} = (\text{request size} - 1/\text{track size}) \times \text{head switch time}$$

Bigger cylinder – When BPI increases, more sectors per track, more sectors per cylinder. When operating within a small range of data, more sectors in a cylinder has 2 effects:

1. The seek distance is reduced
2. The number of seek is reduced

Higher track per inch

Seek time composes of two parts:

1. travel time
2. settling time

$$\text{seek time} = A + B \times \text{sqrt}(\text{ seek distance }) + C \times \log(\text{TPI})$$

where A,B,C are some constants specific to the disk drive.

TPI has two opposing effects on the seek time. Higher TPI means shorter physical seek distance, means shorter travel time. On the other hand, tracks are narrower require longer settling time.

No ID record format

The conventional format, each sector has ID field (or header). To increase capacity, no-ID recording eliminates the ID field, allowing more data sectors on each track. The drive can find the sectors by keeping a table of relations between sectors and embedded servos.

File system – Allocation unit

The size of allocation unit of a file system affects the total capacity of a drive. For example, file allocation table (FAT) of DOS, Windows, the allocation unit is called "cluster", the cluster size is 16 sectors for a drive with capacity 512 – 1,024 M bytes.

For larger files – bigger cluster size is better.

For small files – larger cluster size means larger distance between file, hence longer seek time. With a file occupies only small portion of a cluster, look ahead buffer is less effective. Result: smaller cluster size is better.

RAID

Multiple disks can be organised to use redundancy to improve reliability and performance. Using an array of disks that operate independently and in parallel. Separate I/O requests can be handled in parallel as long as data reside on separate disks. A single I/O request can be executed in parallel if the block of data is distributed across multiple disks.

RAID (Redundant Array of Independent Disks) is proposed to close the gap between processor speed and slow electromagnetic disk drives. The strategy is to use multiple drives and to distribute data to enable simultaneous access, therefore improving I/O performance. The RAID consists of level zero to level five. An excellent survey written by the inventor of the RAID is [CHE94].

The reliability of a disk is stated by the manufacturer in terms of the Mean Time To Failure (MTTF), assuming a constant failure rate – that is, an exponentially distributed time to failure – and that failures are independent.

Example MTTF rated of the disk IBM 3380 (7.5 Gbytes formatted) is 30,000 hours and in practice this figure is 100,000 hours [IBM87].

The arrays are divided into reliability groups, with each group having extra "check" disks containing redundant information. When a disk fails, within a short time, it should be replaced and the information will be reconstructed onto the new disk using the redundant information. This is called Mean Time To Repair (MTTR). The MTTR can be reduced if the system includes extra disks to act as "hot" standby spares; when a disk fails, a replacement is switched in

electronically. Periodically, a human operator replaces all failed disks. There are five different organisations of disk arrays (not counting Level 0), beginning with mirrored disks and progressing through a variety of alternatives with differing performance and reliability.

RAID level 0

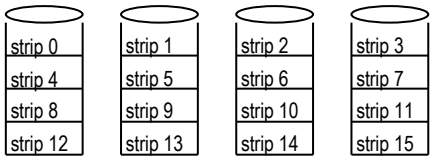
The data is distributed across all disks in the array by stripping. It increases the chance of handling multiple I/O requests in parallel. It also achieves high data transfer if the request contains large amount of contiguous data, compared to the size of a strip. A single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate. RAID 0 can also handle high I/O request rate by balancing I/O load across multiple disks.

RAID level 1

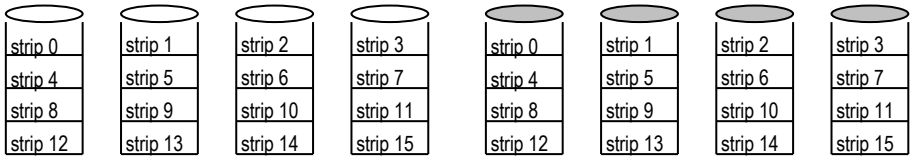
Every disk in the array has a mirror disk that contains the same data. A read request can be serviced by either of two disks. A write request requires both corresponding strips to be updated. When a drive fails, the data may be accessed from the second drive. RAID 1 can achieve high I/O request rates if the bulk of the requests are reads.

RAID level 2

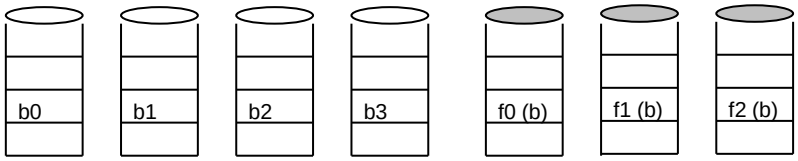
All member of disks participate in executing every I/O request. The spindles of the individual drives are synchronised [KIM86] so that each disk head is in the same position on each disk at any given time. Data striping is RAID 2 is very small as a single byte or word. An error-correcting code is calculated across corresponding bit positions on each data disk. The bits of code are stored in multiple parity disks. Typically, a Hamming code is used. This code is able to correct single-bit errors and detect double-bit errors.



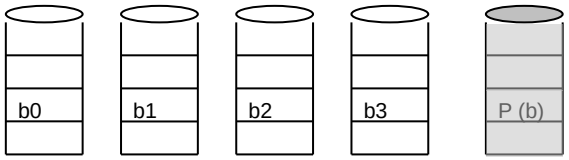
a) RAID 0 (non redundant)



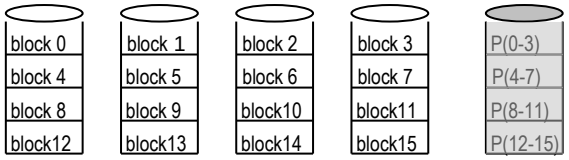
b) RAID 1 (mirrored)



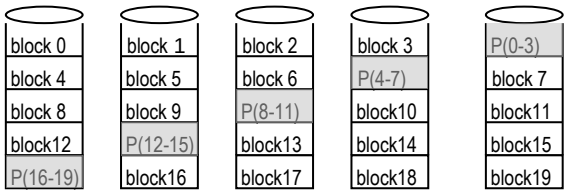
c) RAID 2 (redundancy through Hamming code)



d) RAID 3 (bit-interleaved parity)



e) RAID 4 (block-level parity)



f) RAID 5 (block-level distributed parity)

Figure 11.2 the organisation of RAID

RAID level 3

RAID 3 is similar to RAID 2. The difference is that RAID 3 uses simple parity bit so it requires only a single redundant disk. In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining drives. The contents of any strip of data on any one of the data disks in an array can be regenerated from the contents of the remaining disks in the array. Because the data strip is very small, RAID 3 can achieve very high data transfer rate. However, only one I/O request can be executed at a time.

RAID level 4

Each drive operates independently. Separate I/O requests can be satisfied in parallel. The strips are relatively large. A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the parity disk. Each time a write occurs, two reads and two writes must be performed. One read for the data strip and second read for parity strip to calculate new parity. One write to update data and second write to update the parity strip.

RAID level 5

In RAID 4, every write operation must involve the parity disk, which becomes a bottleneck. RAID 5 is organised similar to RAID 4 but the parity strips are distributed across all disks to avoid the potential I/O bottleneck.

Performance of RAID

To compare all organisations we define several parameters:

D = total number of disks with data (not including extra check disks)
G = number of data disks in a group (not including extra check disks)
C = number of check disks in a group
$n_G = D/G$ = number of groups

The group MTTF is approximately [PAT88]

$$MTTF_{RAID} = \frac{MTTF_{Disk}^{2}}{(D + C \times n_G) \times (G + C - 1) \times MTTR}$$

Since the formula is the same for each level, we assume the following parameters D = 100, G = 10, $MTTF_{Disk}$ = 30,000 hours, $MTTR$ = 1 hour with the check disk per group C determined by the RAID level.

To compare the performance, the following parameters are considered:

- **Reliability overhead cost** – this is the cost of extra check disks**.**
- **Useable storage capacity percentage**
- **Performance** – it is measured by the number of reads and writes per second. It is measured for several types of load:

  For high data rate – Large blocks of data, with large defined as getting at least one sector from each data disk in a group. During large transfers, all disks in a group act as a single unit, each reading or writing a portion of the large data block in parallel.

  For high I/O rate – Small blocks of data, which is read-modify-write sequence of disk accessing. This is a suitable measure for transaction-processing systems which contains many small transfers. During the small transfers, each disk in a group can act independently.

  We measure the effective performance per disk.

One additional factor needs to be considered, the slow down factor *S*. When individual accesses are distributed across multiple disks, average queueing, seek, and rotational delay may differ from the single disk case. When many arms on different disks seek to the same track, the average seek and rotate time will be larger than the average for a single disk. To account for this, the factor S is included, $1 \leq S \leq 2$, when a group of disks work in parallel. With synchronous disks [KIM86], the spindles of all disk in the group are synchronised so that the corresponding sectors of a group of disks pass under the head simultaneously, there is no slow down, S = 1. Figure 11.3 summarises the performance parameters of all RAID levels.

	RAID 1	RAID 2	RAID 3	RAID 4	RAID 5
MTTF	>500 years	>50 years	>90 years	>90 years	>90 years
Total no. of disk	2D	1.4D	1.1D	1.1D	1.1D
Overhead cost	100%	40%	10%	10%	10%
Usable storage capacity	50%	71%	91%	91%	91%
Efficiency per disk (event/sec.)					
Large read	1.00/S	.71/S	.91/S	.91/S	.91/S
Large write	.50/S	.71/S	.91/S	.91/S	.91/S
Large R-M-W	.67/S	.71/S	.91/S	.91/S	.91/S
Small read	1.00	.07/S	.09/S	.91	1.00
Small write	.50	.04/S	.05/S	.05	.25
Small R-M-W	.67	.07/S	.09/S	.09	.50

Figure 11.3 The performance parameters of all RAID levels (from [PAT88])

To achieve reliability and performance, the RAID starts with mirrored disks, and with each succeeding level improving:

- the data rate – characterised by a small number of request per second for massive amounts of sequential data.
- the I/O rate – characterised by a large number of read-modify-write to a small amount of random data.
- the useable storage capacity

Figure 11.4 shows the performance improvement per disk for each level RAID. The highest performance per disk comes from either Level 1 or Level 5. In transaction-processing situations using no more than 50% of storage capacity, then the choice is mirrored disks (Level 1). For a high data rate, Level 5 looks best with high data rate and high useable storage capacity.

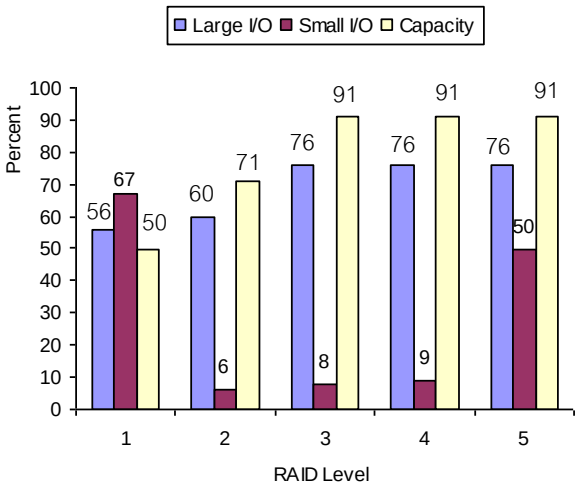


Figure 11.4 The efficiency per disk and usable storage capacity of all RAID levels for D = 100, G = 10, S = 1.3. (from [PAT88])

I/O functions

The processor and memory are connected to input/output devices via I/O modules. Peripherals do not connect directly to the system bus. The data rate of peripherals is much slower than the processor and memory and the data formats are usually different. There will be many I/O devices connected to an I/O controller. Each device has an identifier and it is used to communicate with the I/O controller. There are many activities in an I/O controller:

- control and timing
- CPU communication
- device communication
- data buffering
- error detection

The following scenario illustrates the control of data transfer from an external device to the processor.

1. The CPU interrogates the I/O module to check the status of the device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the CPU requests the transfer of data, by means of a command to the I/O module.

4. The I/O module obtains a unit of data from the external device.
5. The data are transferred from the I/O module to the CPU.

There are three techniques for I/O operations: programmed I/O, interrupt-driven I/O and direct memory access (DMA). With programmed I/O, the CPU directly controls I/O operations, it must wait until the I/O operations are completed. As the CPU speed is much higher than an I/O module, this is wasteful of CPU time. With interrupt driven I/O, the CPU issues an I/O command and continues to execute other task, it is interrupted by I/O module when the data transfer is completed. Both programmed I/O and interrupt-driven I/O, the CPU is responsible to exchange data between main memory and input/output devices.

The programmed and interrupt driven I/O have limitations that the I/O transfer rate is limited by the speed of the CPU and also the CPU is tied up managing I/O transfer. With direct access memory, the I/O module exchanges data with main memory without the CPU involvement.

DMA function

The DMA module takes over the control of the system bus from the CPU in order to transfer data to and from memory. It can forced the CPU to temporarily suspend operation, this is referred to as *cycle stealing*. The DMA module steals a bus cycle. When a CPU wishes to read or write a block of data, it issues command to the DMA module, the following:

- read or write request
- the addresss of I/O device
- the starting location in memory
- the size of data to be read or write

The CPU delegates the I/O operation to the DMA module. The DMA module transfers data, one word at a time, directly to memory. When the transfer is completed, the DMA module sends an interrupt signal to the CPU.

Evolution of I/O Channels

The evolution of I/O functions started from a simple control of CPU to the delegation of the task to the I/O system without the CPU involvement. The following steps show how I/O functions have evolved:

1. the CPU directly control I/O

2. I/O controller, the CPU uses programmed I/O. The CPU is isolated from specific details of devices interface.
3. add interrupt, increase efficiency
4. I/O controller uses DMA
5. I/O controller is enhanced to become a processor, running special I/O instructions.
6. I/O controller has local memory, a large set of I/O devices can be controlled with minimum CPU intervention.

Step 5 and 6, I/O controller is called I/O channel. The term I/O channel is associated with IBM mainframes as IBM is the first to recognise the importance of using direct access devices for external storage and used it in their major operating system OS/360 [PRA89]. Putting a lot of functionality into an I/O controller, so called, the intelligent I/O control, is attractive but in practice the CPU will be advanced in a much higher rate due to the market pressure. A CPU is fabricated using the most advanced process technology for performance reason. Hence, in practice, I/O channel will be one or two generations behind the CPU in terms of speed. Therefore, when the speed of the CPU is much higher than I/O channel it makes sense to let the CPU controls devices directly.

References

[CHE94] Chen, P., Lee, E., Gibson, G., Katz, R., and Patterson, D., "RAID: high-performance, reliable secondary storage", ACM Computing surveys, June 1994.

[IBM87] IBM 3380 Direct Access Storage Introduction, IBM GC 26-4491-0, September 1987.

[KIM87] Kim, M., "Synchronous disk interleaving", IEEE Trans. of Computers, vol. C-35, no. 11, November 1987.

[NGS98] Ng, S., "Advances in disk technology: performance issues", Computer, May 1998, pp.75-81.

[PAT88] Patterson, D., Gibson, G., Kaltz, R., "A case for redundant arrays of inexpensive disks (RAID)", Proc. of the ACM SIGMOD Conf., Chicago, IL., June, 1988.

[PRA89] Prasad, N., IBM mainframes: architecture and design, McGraw-Hill, 1989.

[STE81] Stevens, L., "The evolution of magnetic storage", IBM Jour. of Research and Development, vol. 25, no. 5, September 1981, pp.663-675.

Chapter 12

Future architecture

The future of computer architecture ties closely with the advancement of microelectronics. This chapter reviews the progress of architecture and projects the future based on the fact that one billion-transistor device will be possible (*circa* 2010). There are many alternative proposals ranging from more conservative designs to revolutionary designs. As the future is not likely to be predictable with accuracy, we see only the sketch of what is possible for the future computer architects.

Evolution of computer architecture

The evolution of computer architecture has progressed from simple sequential machines to modern out-of-order execution machines. In the early days, the main challenge has been to design and construct the large and complex systems that required team of engineers. The emphasis in term of computer design has been the instruction set architecture (ISA). As the technology advanced, especially microelectronics industry that can produce a large amount of resources, the number of transistors, on a single chip, it enables the designer to build the most complex part of computer, the processor, on a chip. A new dimension on performance issue brought about many new ideas in computer architecture. We will review this evolution as follows.

Sequential execution

A simple machine performs instruction execution that composed of instruction fetch, decode, execute and writeback in sequence. Each instruction is complete before the next instruction begins.

```
I1, I2, I3
```

Overlapped execution (pipeline)

A pipelined machine achieves a higher performance by using the overlapping of execution. Each instruction execution is divided into a number of steps, which are then executed by independent hardware units. The functional units are used concurrently. The next instruction can begin before the first instruction is finished. In this way, many instructions can be resided in the pipeline at once.

```
Fetch, Decode, Execute
       Fetch,  Decode, Execute
```

Without caches, memory access is much slower than processors. The rate of instruction execution is limited by the speed of fetching instruction from memory. The time spending in fetching an instruction is larger than the time to decode and execute it. To increase performance, designers tried to do "more" in one instruction during execution, the execution phase is multistep.

```
------ Fetch----- Dec  Exe   Exe
                  -----Fetch---- Dec  Exe   Exe
```

The characteristic of the ISA in this era is that the CPI is large. The cycle time is also large because the complex circuits required executing complex instructions. This also increases the chance of having conflict in the pipeline because one instruction stays in the pipeline for long time that it interferes with other instructions.

The invention of cache memory reduces the fetch time greatly. Current design concentrates on reducing CPI and cycle time. By simplifying the execution of one instruction and with appropriate choices of the ISA, the pipeline can be more effective and circuits can be simpler and faster.

Superpipeline

Once the pipeline enables CPI to reach 1.0, the only way to increase speed is to reduce cycle time. To make it possible, the pipeline is divided into finer grain which reduce the clock time for each stage. This technique is called "superpipeline".

```
Fet1, fet2, dec1, dec2, wrt1, wrt2
      Fet1, fet2, dec1, dec2, wrt1, wrt2
```

Superscalar

To increase performance further we need to issue more than one instruction per clock. This is called "superscalar". It relies on the instruction-level-parallelism (ILP). As the number of simulateneous instructions in the pipeline increase, the stall becomes very costly. Several techniques were invented to reduce the number of stall cycles, for example branch prediction, out-of-order execution, and speculative execution. It is becoming more difficult in extracting the instructions that can be executed concurrently from programs.

```
Fetch, decode, execute, writeback
Fetch, decode, execute, writeback
       Fetch,  decode,  execute, writeback
       Fetch,  decode,  execute, writeback
```

Of course, the combination such as superpipe-superscalar is possible.

```
Fet1, fet2, dec1, dec2, wrt1, wrt2
Fet1, fet2, dec1, dec2, wrt1, wrt2
      Fet1, fet2, dec1, dec2, wrt1, wrt2
      Fet1, fet2, dec1, dec2, wrt1, wrt2
```

Summary

The evolution from a non-overlap execution (sequential) machine to an overlapped execution (pipeline) machine was the first step. The pipeline technique can be used for instruction execution or for complex instruction (such as floating-point) execution that required multi-cycle in the pipeline. The CPI for a pipelined machine approaches one. Multiple functional units are used to allow concurrent execution of instructions. Scoreboard and Tomasulo methods are hardware technique that enable dynamic execution in which instructions can be rearranged by hardware to execute according to the resource availability. The superpipeline machine has CPI equals 1.0 with a high clock rate. The superscalar has CPI less than 1.0. Another class of machine is vector machines. Vector machines reduce fetch time and increase effective pipeline using the data-level-parallelism (DLP) but its use is restricted to the class of program that fits to vector computation. With this background in mind, now we start to look into the future.

Driving factors

In 2010 the microelectronics industry will be able to manufacture 800 Million transistors processors with thousands of pins for 1,000 bit-bus and with the clock speed over 200 GHz. The device will consume around 180 W of power. The current trend is that the on-chip wires are becoming much slower than logic gates as the device size getting smaller. It will be impossible to maintain one global clock over the entire chip.

The hardware alone will not be able to extract more parallelism from programs. A new compiler technology is required to extract parallelism from code. The present workloads are becoming more multimedia-centric and will continue to be so in the future. The design and manufacturing of complex devices such as one-billion transistor processor will be a challenging task for engineers. Verifying that the design works correctly and testing each chip now consume 40-50% of Intel chip's design cost. The architecture that simplifies this process will have a great impact.

Multimedia workloads

The multimedia workload is different from the traditional workload of the past [DIE97]. The characteristics of multimedia are as follows. It requires real-time response, the data streams are continuous. It is fine-grained data parallelism, and the workload is multitask. Finally, multimedia workload requires very high memory bandwidth.

All signal processing and graphics applications have inherent data parallelism. Input data streams are large amount of small data elements such as pixels, vertices and signal values. These data frequently need identical processing such as filtering and transformations. Vector units with wide data path would achieve significant speed up for such workloads. Because input data are streams, cache performance will suffered from poor locality of data. Data prefetch and cache bypass schemes become more important.

General purpose processors have been enhanced with special functional units to support multimedia, for example Intel's MMX, Sun's VIS for SPARC, Silicon Graphic's MDMX for MIPS, Digital's MVI for Alpha, and Hewlett-Packard's MAX2 for PA-RISC. The future architecture will provide similar functional units to handle multimedia workloads.

Proposals for future architectures

We review seven proposals that aim to make full use of one billion transistors on a chip. They range from evolutionary designs, which emphasise software compatibility and retaining the current programming model, to revolutionary architecture that break away from current practices. The more traditional designs employ some recent advance concept such as trace cache and multiple branch predictors to improve the performance of instruction issued. The non-conventional designs rely on the combination of compiler and hardware to extract more parallelism from programs.

To deliver a large number of instructions to the execution units, the first three proposals rely on a common technique, the trace cache [ROT96]. The second technique that will have impact on the future architecture is the use of hybrid branch prediction. The current branch predictor is capable of 97% accuracy but the remaining misprediction still incurs a large performance penalty. A multicomponent hybrid predictor such as Multi-Hybrid [EVE96] can achieve nearly 100% prediction accuracy.

Trace cache

The trace cache is an instruction cache. Its main purpose is to fetch pass a taken branch. It stores logically continuous instructions in physically continuous storage. A cache line stores a segment of the dynamic instruction trace – up to an issue width – across multiple taken branches. Instruction fetch hardware unwinds programs into traces, each of which may have 8-32 instructions as well as predicted conditional branches. The traces are placed in a trace cache and the fetch unit subsequently reads traces from the trace cache. A single entry in the trace cache holds an entire trace. The trace cache is access using the starting address of the next block of instructions combined with predicted information returned by the trace predictor. An entire trace consisting of multiple basic blocks is fetched in one cycle. The fill unit attempts to maximise the size of segments by coalescing instructions from multiple cycles. Figure 12.1 shows the organisation of a trace cache.

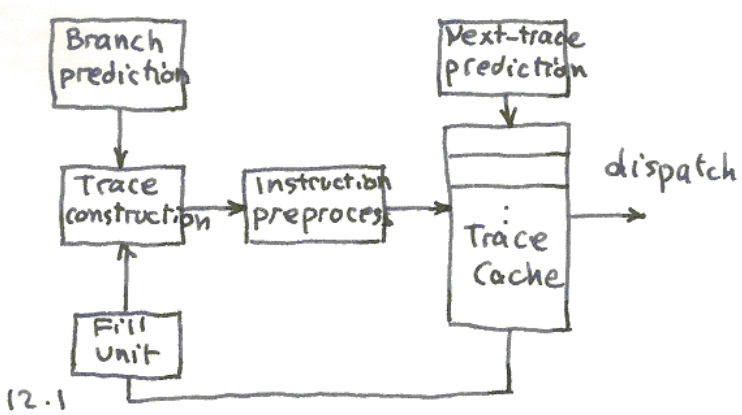


Figure 12.1 the organisation of a trace cache

Figure 12.2 shows the gain of a trace cache versus an instruction cache for three largest applications from SPECint95 benchmarks: go, gcc, and vortex. Assume a 16-wide issue processor with perfect branch prediction. The trace cache is more effective and the gain increased as the size of cache is increased.

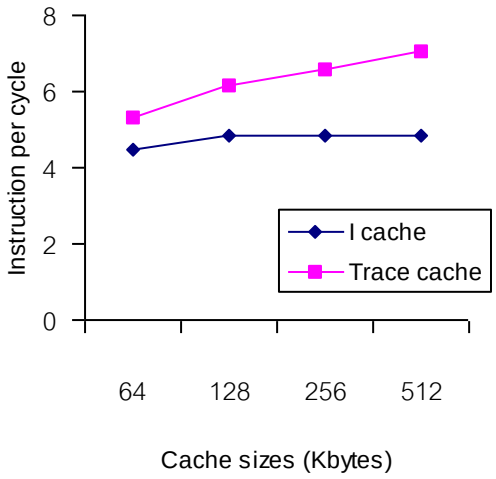


Figure 12.2 the effect of trace cache continue to gain when the size is increased

Hybrid branch prediction

The multi-hybrid branch predictor uses multiple separate branch predictors. Each predictor is tuned to different class of branch. This solves the problem of sensitivity versus accuracy. A large predictor takes more time to react to changes in a program. A small predictor can react quickly but is not very accurate. The Multi-Hybrid [EVE96] uses a set of selection counters for each entry in the branch target buffer keeping track of the predictor currently most accurate for each branch and then using the prediction from that predictor for that branch. Figure 12.3 shows the nearly 100% accuracy of the Multi-Hybrid on SPECint95.

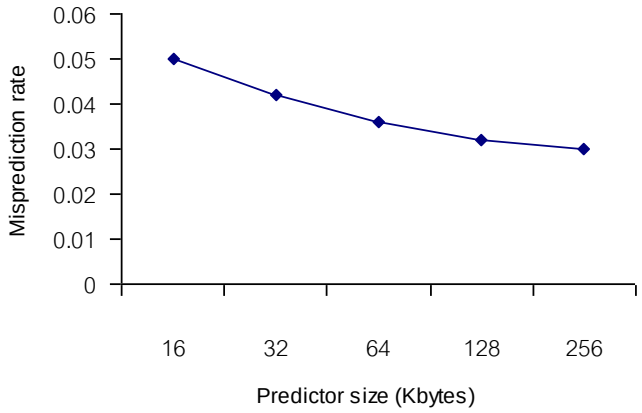


Figure 12.3 Misprediction rate of Multi-Hybrid for SPECint95

Advanced superscalar

An advanced superscalar processor is a scale up of the current design to issue 16-32 instructions per cycle [PAP97]. The first difficulty is in instruction delivery, an advanced superscalar processor uses the trace cache and a hybrid branch predictor to deliver sufficient number of instructions to the execution units.

The second difficulty is the memory bandwidth and latency. A 16-wide issue will need to execute about eight load/stores per cycle. Instead of using a large monolithic, multiported, first-level cache which will have large cycle time, a

number of smaller replicated first-level caches will provide the require ports with fast cycle time. A large second-level cache is also on-chip.

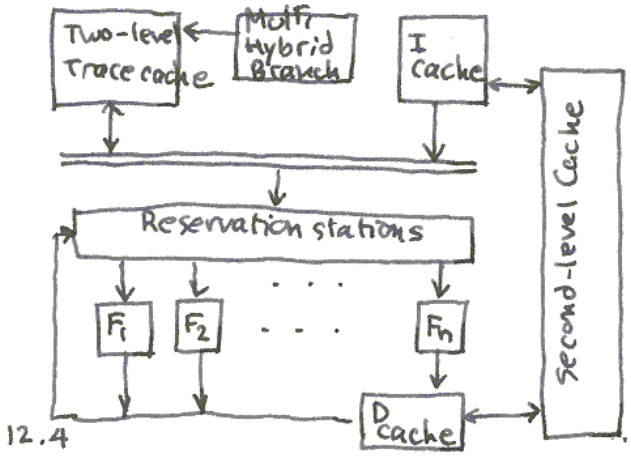


Figure 12.4 an advanced superscalar processor

The execution units comprising 24 to 48 pipelined functional units with large reservation stations having the capacity of 2000 or more instructions will be able to execute 16-32 instruction per cycle. Figure 12.5 shows the available ILP of SPECint95 benchmarks with an instruction window of 2000 instructions while varying issue/execution widths. Assuming a perfect cache and perfect branch prediction.

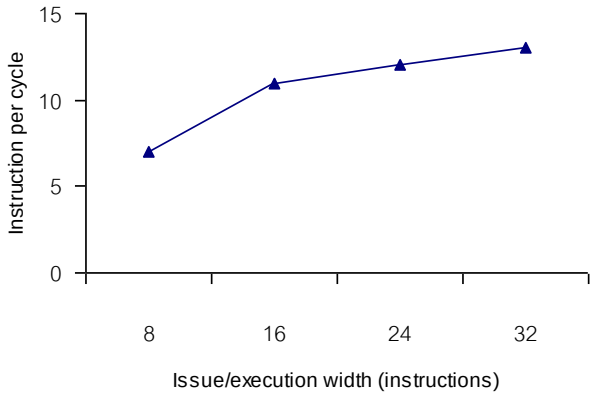


Figure 12.5 the available parallelism with a instruction window of size 2000.

Superspeculative

A superspeculative processor enhances the wide issue superscalar performance using speculative at every point in the pipeline [LIP97]. The instruction execution has four phases: fetch, decode, execute, and commit. The architecture employs a wide range of speculative technique to improve the throughput of instruction flow, register dataflow, and memory dataflow.

To improve the instruction flow, a trace cache is used. The misprediction is reduced using two-phase branch predictor with a local history and a global branch history [MCF93]. Multiple branches are predicted in each cycle.

Register dataflow affects the processing of ALU instructions. The dependence prediction can resolve inter-instruction dependency. This technique predicts the dependence between instructions and speculatively allowing instructions that are predicted to be data ready to execute in parallel with exact dependency checking. The source operand value prediction eliminates true data dependency. This technique uses dynamic-value history, stored per static program instruction, to predict future values of that instruction's source operands.

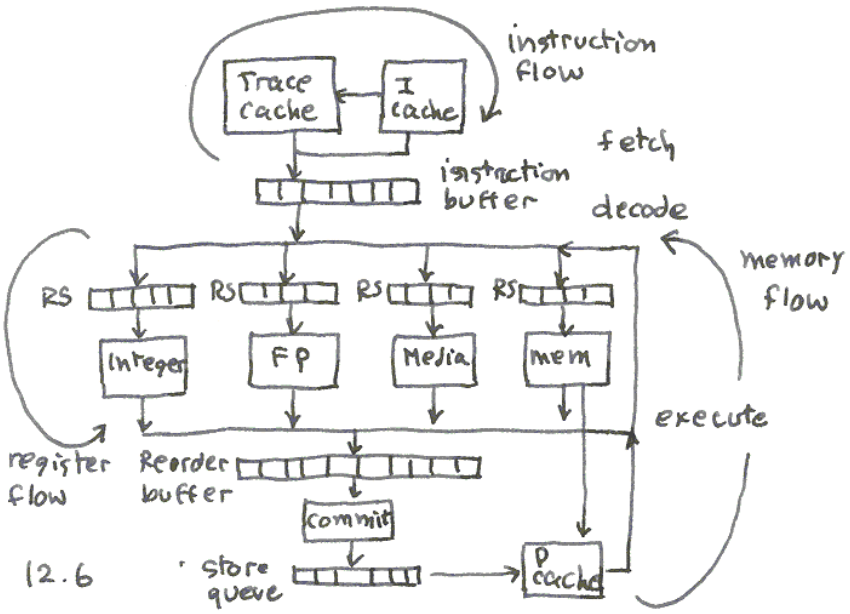


Figure 12.6 an organisation of a superspeculative processor

To improve memory dataflow the average memory latency is minimised using load value prediction. This technique uses per-static-load value history to predict future values.

The performance of a superspeculative processor is evaluated using a processor with 32-issue, 128-entry reorder buffer, 64K byte 4-way set associative D cache and I cache, a perfect second-level cache, and a 128-entry fully associative store queue. Figure 12.7 shows the cumulative gain of the superspeculative IPC (instruction per cycle) beyond a superscalar.

The result demonstrates that superspeculative techniques provide impressive performance, without them a very wide superscalar does not scaled to improve the level of performance.

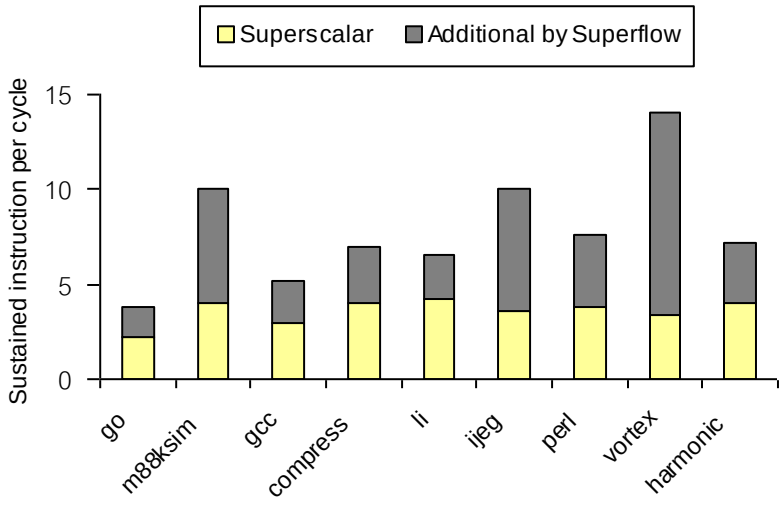


Figure 12.7 additional performance of a superspeculative over a superscalar

Trace processors

A trace processor [SMI97] breaks programs into dynamic sequences of instructions, called *traces*, and uses multiple processing elements to execute multiple traces. A trace processor can execute ordinary serial programs written in a standard language. A high-level control unit partitions the instruction stream

into traces. The processor fetches and executes traces as a unit using a trace cache. Each processing element issues four instruction per cycle, a four-element system can achieve a performance of 16 instructions per cycle. (Fig. 12.8)

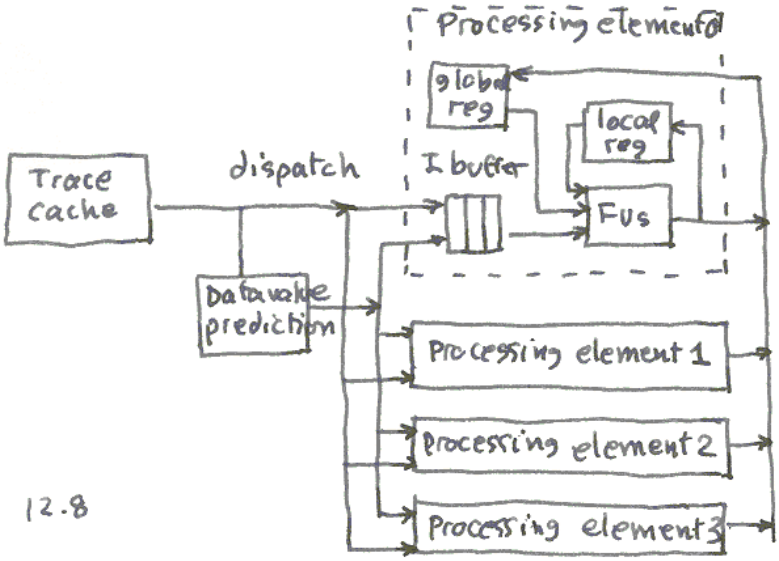


Figure 12.8 a trace processor

Simultaneous multithreading

A simultaneous multithread processor (SMT) [EGG97] exploits all types of parallelism. It consumes both instruction-level (ILP) and thread-level parallelism (TLP). The TLP can come from either multithreaded, parallel programs or individual, programs in a multiprogramming workload. More instructions are extracted from TLP to fill the pipeline. It combines wide issue superscalar processors (similar to MIPS R10000) with multithreading. The processor can hold the hardware state (registers, PC and so on) for several threads at once. It can issue multiple instructions form multiple threads in each cycle.

The fetch unit has eight program counters, one for each thread context. On each cycle, it selects two different threads and fetches eight instructions from each thread. This increases the probability of fetching only useful instructions. It then chooses a subset of these instructions for decoding. This scheme performs 10%

better than fetching from one thread alone. The thread selection uses the instruction count feedback technique, which gives highest priority to the threads with the fewest instruction in the decode, renaming, and queue pipeline stages.

A SMT processor has an eight instruction fetch/decode width, six integer units, four floating-point units, 32-entry integer and floating-point dispatch queues, hardware context for eight threads, 100 integer renaming registers, 100 floating-point renaming registers, and retirement up to 12 instruction per cycle.

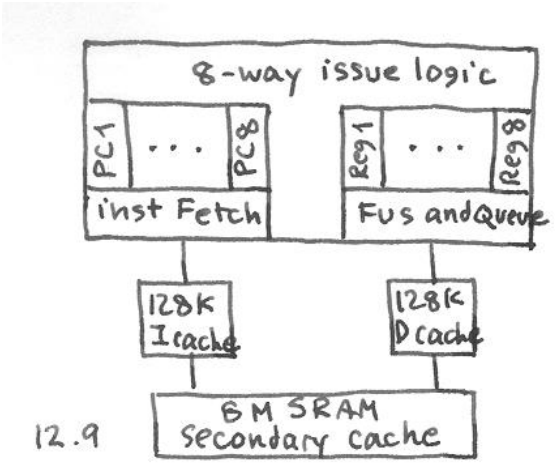


Figure 12.9 a simultaneous multithread processor

To evaluate the performance, the SMT is compared with a four-processor multiprocessor system running parallel workloads. Each processor in the four-processor system contains approximately one-fourth of SMT's chip resources. The benchmark is the parallel applications from SPEC95 and Splash2 suites. The result is shown in Figure 12.10.

The SMT obtained better speedup than the multiprocessors. Speedups of the multiprocessors were hindered by the fixed partitioning of their hardware resources across processors. Using both instruction-level-parallelism and thread-level-parallelism, a simultaneous multithread processor uses functional units more effectively. It achieves greater instruction throughput and programs speed up on multiprogramming and parallel workloads.

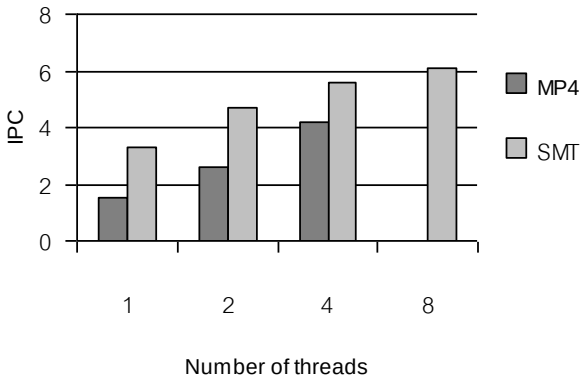


Figure 12.10 instruction throughput of SMT and MP4 on parallel workloads.

Chip multiprocessors (CMP)

A chip multiprocessor (CMP) [HAM97] has a number of duplicated processors (4-16) on a chip and run parallel programs. In addition to loop-level-parallelism and thread-level-parallelism, a CMP exploits a third form of very coarse-grain

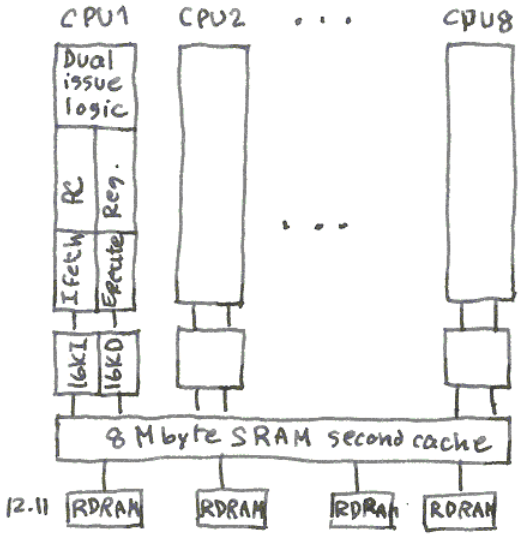


Figure 12.11 a chip multiprocessor

parallelism, process-level-parallelism. The form of parallelism comes from independent applications running in independent processes.

Because of the interconnect delay, the layout of a billion-transistor chip will significantly affect the processor architecture. A CPU will be built out of several small, high-speed logic blocks connected by longer, slower wire that are used infrequently. A CMP processor composed of 8 small 2-issue superscalar processors with 16 16K byte caches. Eight cores are independent. The small cache and tight connection allows single cycle access.

To maximise CMP performance, programmers must find thread-level-parallelism. The CMP has been evaluated against a single 2-issue processor running SPEC95 and multiprograms. The multiprogram is an integer multiprogramming workload. All of them are computation intensive and run as a separate process. This benchmark has a large amount of process-level-parallelism. The result is reported in Figure 12.12.

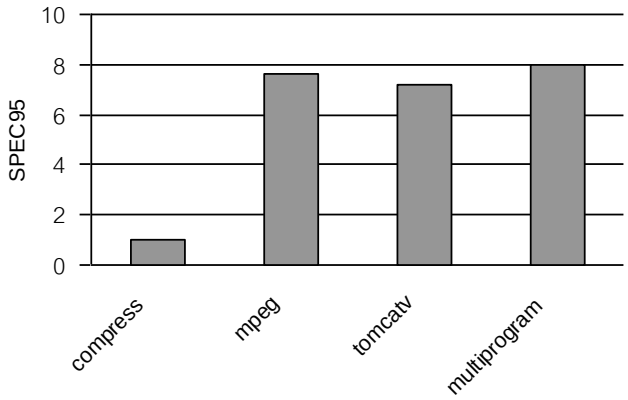


Figure 12.12 CMP performance relative to a superscalar

Among many alternatives, a multiprocessor on a chip will be easiest to implement. A CMP processor offers superior performance using relatively simple hardware. On code that can be parallelised into multiple threads, the CMP core will perform as well as or better than more complicated wide issue superscalar on cycle-per-cycle basis.

Intelligent RAM

The intelligent RAM [PAT97] merges a high-performance processor and DRAM main memory on a single chip to lower memory latency, increase memory bandwidth, and improve energy efficiency. The processor and memory speed gap has been widening steadily as processor performance increasing at the rate of 60% per year while memory latency in improving at only 7% per year. Large amount of chip area is devoted to cache memory to bridge this gap. For example, caches occupy almost half of the die area in Alpha 21164. IRAM approach uses on-chip resources for DRAM.

This on-chip memory can be treated as main memory. It supports high bandwidth and low latency using a wide interface. Using on-chip main memory also reduces the number of pins for memory interface off-chip. An architecture that is a natural match to IRAM is vector processors. The combination of vector units with a scalar processor creates a general-purpose architecture. Vector units have many applications including scientific calculation, multimedia, and databases. Because of the simplicity of their circuits, vector units can operate at higher clock speed and also provide higher energy efficiency.

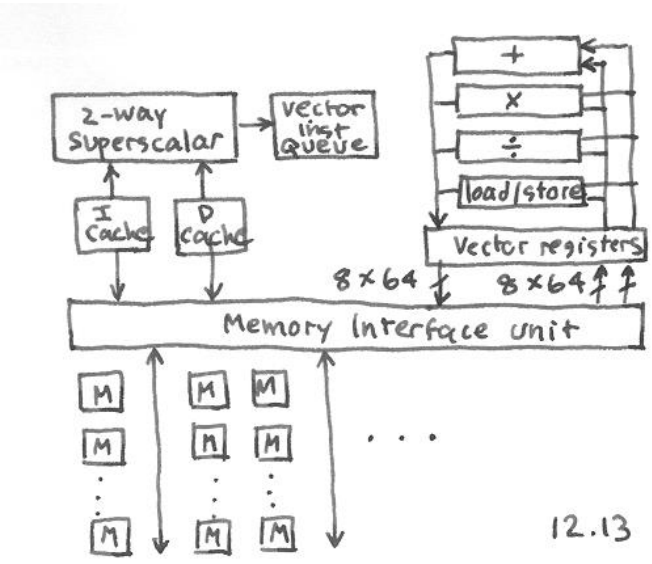


Figure 12.13 organisation of an IRAM vector processor

Assume the feature size of 0.13 um and a die area of 400 mm^2, a full size DRAM die with a quarter of area dedicated to logic. A vector IRAM processor includes

the following: the vector unit consisted of 64-bit floating-point add, multiply, and divide; integer operations; load/store; and multimedia operations, running at 1GHz, 32 64-element vector registers, and sixteen 1024-bit-wide memory ports. The peak performance is 16 GFLOPS at 64-bit per operation. The on-chip memory has a capacity of 96 Mbytes, assuming a pipelined synchronous DRAM interface with 20-ns latency and a 4-ns cycle time, the bandwidth will be adequate for 192 Gbytes/sec to feed vector units.

Merging a microprocessor and DRAM on a chip has the following advantages: a reduction in latency by a factor of 5 to 10, an increase in bandwidth by a factor of 50 to 100, an advantage in energy efficiency of a factor of 2 to 4.

RAW

RAW is a highly parallel architecture consists of hundreds of simple processors connected through a reconfigurable logic [WAI97]. It eliminates the traditional instruction set interface and exposed the simple replicated architecture directly to the compiler. This allows the compiler to customise the hardware to each application.

A RAW processor is a set of interconnected tiles, each of which contains instruction and data memory, an ALU, registers, configurable logic, and a programmable switch for routing the message between tiles. It allows communication with short latencies, similar to register access. Each tile can use configurable logic to construct operations suited to a particular application. Static RAM distributed across tiles eliminates the memory bandwidth bottleneck and provides short latency.

One billion-transistor die could carry 128 tiles, each has 16 K bytes instruction memory, 16 K bytes switch instruction memory, 32 K bytes first-level data memory. The memory is SRAM type and backed by 128 K bytes DRAM. Each tile has 2 M transistors for a pipelined processor (a R2000 equivalent CPU), floating-point units and configurable logic. Interconnect consumes 30% of chip area.

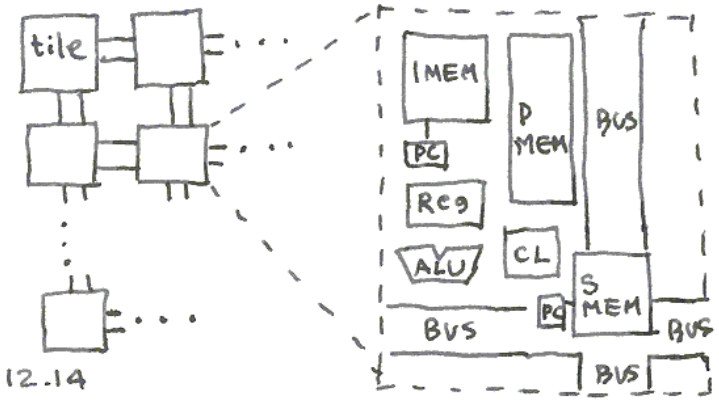


Figure 12.14 organisation of a RAW processor

Software implements operations such as register renaming, instruction scheduling, and dependency checking. Compilers can schedule single word data transfer and exploit ILP. A RAW architecture implements wide-word arithmetic, and multiple-bit or byte-level operations in each tile. Software can select the data path width. The processor can perform bit-level applications such as logic simulators and byte-level applications such as graphics with high degree of parallelism.

The compilation process maps programs to RAW hardware. It composed of partitioning, placement, routing, scheduling and configuration selection. Partitioning aims to find fine-grain ILP. Placement maps threads to physical tiles. Routing and scheduling allocate physical network resources and produce a program for each tile switch. Configuration selection replaces each compound operation by a call to an appropriate custom instruction. Compiler invokes a logic synthesis tool to translate a custom operation into the configurable logic.

A prototype using FPGA technology running at 25 MHz. It uses static schedule and hardwired control. Table xy compares the prototype with all software executing of a 2.82 SPECint95 SparcStation 20/71. The compilation step of the RAW prototype is very expensive requiring several hours on 10 workstations. The prototype achieves 10-1000 speedup over the commercial processor.

Table 12.1 Hardware prototype 25 MHz (Xilinx 4013) compares to software executing on 2.82 SPECint95 SparcStation (Sparc 20/71)

Benchmark	data width (bits)	no. of elements	speed up over sw
binary heap	32	15	1.26
bubble sort	32	64	7
DES encryption	64	4	7
integer FFT	3	4	9
Jacobi 16x16	8	256	230
Jacobi 32 x 64	8	2048	1562
Conway's life 64 x 16	1	1024	597
Conway's life 64 x 64	1	4096	1758
integer matrix multiply	16	16	90
merge sort	32	14	2.6
n queens	1	16	3.96
single-source shortest path	16	16	10
multiplicative shortest path	16	16	14
transitive closure	1	512	398

Conclusion

The rate of progress is very fast. It is interesting to explore the trends that will affect future architectures and the space of these architectures. Future processors will have large on-chip memory. The level-two cache will be the norm. Large amount of on-chip transistors allows virtually anything to be implemented. The limiting factor will likely be the imagination of the architect.

References

[DIE97] Diefendorff, K., and Dubey, P., "How multimedia workloads will change processor design", Computer, September 1997, pp. 43-45.

[EGG97] Eggers, E., Emer, J., Levy, H., Lo, J., Stamm, R., Tullsen, D., "Simultaneous Multithreading: a platform for next-generation processors," IEEE Micro Sept./Oct. 1997, pp.12-19.

[EVE96] Evers, M., Chang, P., and Patt, Y., "Using hybrid branch predictors to improve prediction accuracy in the presence of context switches," Proc. 23rd Ann. Int. Sym. Computer Architecture, ACM Press, NY., 1996, pp.3-11.

[HAM97] Hammond, L., Nayfeh, B., Olukotun, K., "A single chip multiprocessor," Computer, September, 1997, pp.79-85.

[LIP97] Lipasti, M., and Shen, J., "Superspeculative microarchitecture for beyond AD 2000," Computer, September, 1997, pp.59-66.

[MCF93] McFarling, S., "Combining branch predictor, Tech. Rep. TN-36, Digital Equipment Corp., Maynard, Mass., 1993, http:// www. research. digital. com/ wrl/ home.html

[PAT97] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K., "A case for intelligent RAM," IEEE Micro, Mar./Apr., 1997, pp.34-44.

[PAP97] Patt, Y., Patel, S., Evers, M., Friendly, D., and Stark, J., "One billion transistors, one uniprocessor, one chip," Computer, September, 1997, pp.51-57.

[ROT96] Rotenberg, E., Bennett, S., and Smith, J., "Trace cache: a low latency approach to high bandwidth instruction fetching," Proc. 29th Ann. ACM/IEEE Int. Sym. on Microarchitecture, IEEE CS Press, 1996, pp.24-34.

[SMI97] Smith, J., and Vajapeyam, S., "Trace processors: moving to fourth-generation microarchitectures," Computer, September, 1997, pp.68-74.

[WAI97] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P, Barua, R., Babb, J., Amarasinghe, S., and Agarwal, A., "Baring it all to software: RAW machines", Computer, September, 1997, pp. 86-93.

Appendix A

Projects in computer architecture

Problem definition

The objective is to design or modify a machine and run one or two benchmark programs on its simulator and report its performance (CPI). Basically what you have to do is to "design" a machine, i.e. its instruction set and its behaviour (microstep). You must modify or write a simulator and run some benchmark programs chosen from the Stanford integer benchmark suite. If you cannot make the simulation to work you can submit your design and simulate it by hand.

There are 15 problems ranging from creative to mechanistic process. The reward will be proportional to the "quality" of the solution and the "difficulty" of the chosen problem.

Project list

1. Superscalar S1 with 2 ALUs

Add extra ALU to S1. You can use non-pipe or pipe version. Invent a way to issue two instructions at the same time when possible.

2. LIW version of S1

Redesign S1 to have LIW capability. You have to determine what kind of additional functional units you want to add to improve the performance (depend on your benchmark programs).

3. S1 with Scoreboard

Assume S1 has multifunctional units : FPmult1, FPmult2, FPadd, FPdiv, Integer. Simulate Scoreboard running this program :

```
LF F6,34(R2)
LF F2,45(R3)
MULTF F0,F2,F4
SUBF F8,F6,F2
DIVF F10,F0,F6
ADDF F6,F8,F2
```

4. S1 with Tomasulo

Assume S1 has FP adder, FP multiplier, with 3 and 2 reservation stations, Load buffer, Store buffer with 6 and 3 entries. Simulate S1 with Tomasulo running this program:

```
LF F6,34(R2)
LF F2,45(R3)
MULTF F0,F2,F4
SUBF F8,F6,F2
DIVF F10,F0,F6
ADDF F6,F8,F2
```

5. S1p with branch prediction

Add branch prediction capability to S1 pipe. You have to decide the method to do branch prediction, branch-target buffer.

6. S1p with delay branch

Add delay branch capability to S1 pipe. Examine your benchmark programs. How many delay slot can be usefully filled?

7. Stack machine ISA

Design a stack machine, its instruction set must be stack oriented (no register!). Have a look at my research paper which I designed a stack machine

http://www.cp.eng.chula.ac.th/faculty/pjw/r1/R1.htm

at the section "intermediate code specification". You can also have a look at Java chip called PicoJava at http:// www.cp.eng.chula.ac.th/ faculty/ pjw/ teaching/ ca/ JavaVM/ picojava.pdf

8. Minimum instruction set CPU

Design a processor with minimum number of instructions. It must be able to run at least a benchmark program completely (that is, it must have enough instruction to implement a benchmark program). You should not worry too much about the ISA being absolute minimum, however you should try to make its ISA as small as you can.

9. Fastest Matrix Multiplication S1

Modify S1 such that it can run Matmul.c fastest. There are many ways to do this, you can modify the instruction set (add some special instruction) or add functional units or modify organization (such as two pipelines).

10. Comparing S1 with 2, 3, 4 pipeline stages

Design S1 with 2, 3, 4 stages pipeline. Compare its performance with S1p which has 5 stages pipe. Please note that in this case one stage of the pipeline will take several clocks to be completed.

11. S1 microprogram with 2 formats microprogram

Modify S1m to use 2 formats microprogram to shorten the width of a microprogram word. After observing that ALU functions, Memory control, are never activated at the same time as Bus transfer (Dest, Src, SelR), the following formats are suggested :

Format 1 – Bus transfer

First bit is 0 , Dest : 5, Src : 6, SelR : 3, Cond : 4, Goto : 5 , total = 24 bits

Format 2 – ALU and Memory control

First bit is 1, Sel2R : 1, ALU : 4, Mctl : 2, PC+1 : 1, undef : 6, Cond : 4, Goto : 5, total = 24 bits

This format reduces the width from 31 bits to 24 bits without performance penalty. Sel2R is a control bit to select 2 registers for ALU input. All the rest are similar to S1m.

Write a new microprogram using this narrower microprogram word. Write simulator to run it. Run a benchmark program under this new simulator. Remember that the machine code (object code) doesn't change at all running under this new simulator or the original S1.
(Read Stalling's text book for an example of two-format microprogram)

12. Using microprogram as instructions directly.

Consider that there is no "instruction set", no program counter (but microprogram counter), no instruction fetch in the normal sense. Your machine and "program" is **the microprogram** itself. You have to add some fields into microprogram word such as : R0, R1, R2, ADS which hold the appropriate values. Can you pipeline this machine? (pipeline execution of the microprogram).

13. Add Floating point instructions to S1

Add the following FP instruction to S1 : `fadd, fmult, fdiv`. The FP number in your design is a 32 bits word and a set of FP register (32 bits) is needed. In writing the simulator you don't have to do IEEE Floating point arithmetic yourself. You can use data type in C to do it for you, i.e. you can multiply, divide, add the floating point number in C.

Benchmark programs (choose one)

1. running the program to find square root.
   Using Newton−Raphson, or so called "successive approximation" method. Let x be a guess square root of a then

   $$x_{n+1} = 0.5 (x_n + a/x_n)$$

   Iterate this 7 times and the precision will always be better than 24 bits.
2. evaluate sin x

   $$\sin x = x - x^3/3! + x^5/5! - x^7/7! + ...$$

   using only four terms (not very accurate), where x is expressed in radians and maximum is pi/2.

14. Change S1 to 32 bits word

Design new ISA, instruction set, instruction formats. This machine is essentially S1 with 32 bits instruction and data. You should add some new instructions to make your machine run the benchmark program faster. Write its simulator and run benchmark

15. Change S1 instruction set to 3 registers format

You must change ALL S1 instructions that are appropriate and add the "immediate" value to some instruction. These are the example of immediate instructions :

Immediate mode : addi, cmpi, inci, storeri.

```
addi r1,N                r1 = r1 + N
```

Index mode : loadx, storex

```
loadrx (r1),r2,offset    (r1)+ offset -> r2
storex r1,(r2),offset    r1 -> (r2) + offset
```

Write microstep, modify simulator and run benchmark programs.

Benchmark Programs

The benchmark suite is Stanford Integer benchmark. They are a collection of small interger programs supposed to test CPU integer performance. These programs are suitable for students' excercise and are NOT realistic by today standard: qsort.c, queen.c, sieve.c, hanoi.c, matmul.c, bubble.c, perm.c.

Stanford integer benchmark suite

bubble	sort 100..1 to 1..100 global a[100], N=100
hanoi	5 disks from 1 to 3, global num[4], D = 5
matmul	mul 10 x 10 matrix, global a[100], b[100], c[100] a = b × c
perm	permute N, global val[4], id, N = 4
qsort	quick sort 100..1 to 1..100, global a[100], N = 100
queen	soln of all 8-queen, global Q,Z,D, col[8], d45[15], d135[15], queen[8], soln, run find(0)

sieve find prime less than N, global p[1000], N = 1000, original N = 10000 but is too large for 16-bit applications.

How to do the project

1. You have to design an instruction set with enough instruction to execute some benchmark program (no I/O).
2. You have to design "microarchitecture", i.e. the internal structure of a CPU and write its "microstep".
3. A set of benchmark program (Stanford integer benchmark) written in C is provided. To run a benchmark program you have to convert it to an assembly level program (in the instruction set of your own design). You don't have to run the whole program. You must choose some portion of programs to measure your design. The most important portion that determine the runtime of a program is its "innest loop". The benchmark should be taken from several parts of high level program and should be at least 10-20 lines of assembly code in total. Choose the benchmark that will illustrate the capability of your design.
4. To validate (check that the design is correct) and evaluate (measure how fast your processor is) the design, you can either :
   4.1. Write (modify) a simulator and run the benchmark programs to count the number of clocks required to complete the tasks (you will earn extra bonus for doing the simulation) or
   4.2. Estimate the number of clocks by hand. You need to make sure that you count the right thing.
5. You must hand in a report containing the following sections :
   5.1. Motivation behind your design (why you did it that way).
   5.2. Instruction set details : opcode, opcode format, number of clock required by each instruction.
   5.3. Microarchitecture and its microstep.
   5.4. Your benchmark program (in your assembly language) and why you choose this particular part of the program. Programs should be well commented so that I can read and understand what it does.
   5.5. How you validate and evaluate your design.
   5.6. Performance of your design (Cycle Per Instruction)
   5.7. Conclusion, what you learn from this project.

How I evaluate your project

I will look for the "quality" of your work including:

- the innovative idea and/or well thought out solution
- the correct understanding of the concept that you applied
- the completeness of your work, correctness of the result

Any question regarding the project, please contact me promptly.

Appendix B

How to do a paper

In this assignment, students learn how to acquire additional knowledge from the current academic literature. The list of recommended articles are posted and students choose to do one of them. These articles come from the current research work in the conference proceedings or the journals. They contain the advanced information not appear in a normal textbook. Students are expected to read and summarise, then present it to the class and finally submit written reports.

Requirements

You have to submit

1. You report summarising the paper. You can write in either English or Thai. Please write in your own words. I will not accept the style of "cut and paste" from the original paper. I will look for the following point :
   - Can you get the main point of the paper (hint: main point always state in the abstract). If you can, do you explain it correctly? This is asking you "what" about the paper.
   - Can you get the motivation behind the paper. This is asking you "why" this paper.
   - Can you explain "how" they experiment/propose their idea.
2. You must attach a copy of the original paper with your report. I need it to check your writing/ understanding of the paper.

Everyone has to prepare a 8-minute presentation. I will choose around 8 papers for presentation on spot. Please prepare to present your summary using around 4-5 transparencies.

Assessment

I will judge the paper along these aspects :

- Correct understanding of the concept in the paper.
- Quality of your writing : completeness, easy to understand, clarity of the issue etc.
- Quality of your report. (but don't spend too much time in making the report looks "superb", it is the "content" that is important).

Don't just translate English to Thai. Write in your own words! Make the central issue clear. Don't write 40 pages to explain 6 pages of the original English.

Tips how to give a good talk

- Presenting what you understand. Your friends will be more likely to appreciate a talk that they understand which mean the presenter must understand the subject first.
- Don't just summarize the paper. It will not be convincing. You need to present some evidence for what you are claiming, such as, the processor X is very good because it is very fast, now, show me the benchmark result for this processor. Therefore, present some hard fact that included in the full paper.
- Talk about what you think about that paper. This is your opinion. The audience like to hear opinion and give them something to discuss.
- Prepare your slide. A big, easy to see, clear layout, slide will relieve the eyes strain of your friends. Remember they all have to sit through 8 papers!