Chapter 4

Code Generation

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

4.1 S-code

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

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

Notation

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

stack notation: (before -- after)

Zero argument instructions

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

One argument instructions

`lit n`	Push *n* (-- n)
`inc v`	Increment local variable, SS[FP+v]++
`dec v`	Decrement local variable, SS[FP+v]--
`ld v`	Push DS[v]. (-- DS[v])
`st v`	Take a value *x* and store to DS[*v*] = *x*. (x --)
`get v`	Get local variable *v*. (-- SS[FP+v])
`put v`	Store a value *x* to local variable *v*. (x --)
`call f`	Create a new activation record, goto *f* in CS
`ret n`	Return from a function call, *n* is the size of activation record. Remove the current activation record. Return a value if function returns a value.
`fun n`	Function header, *n* is the number of local variables
`jmp n`	Jump to PC+n in CS
`jt n`	Jump PC+n if top of stack = 1, pop
`jf n`	Jump PC+n if top of stack = 0, pop
`sys n`	Call a system function *n*, for interfacing to external functions, the arguments are in the stack, the number of arguments can vary.

4.2 S-code format

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

The "`end`" is a pseudo instruction. It does not existed in a real processor. It is used to stop the processor simulation. S-code supports high-level function call directly similar to N-code. The run-time data structure must be understood. An

Encoding

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

activation record stored a computation state. It is resided in the stack segment. The computation state consists of: PC (return address), FP, all local variables. SP needs not be stored as it will be recovered properly when return. The necessary information, the size of the activation record, is stored as the argument of "*ret*" instruction. The following diagram shows the layout of an activation record in the stack segment (notice that it is exactly the same as the activation record of N-code).

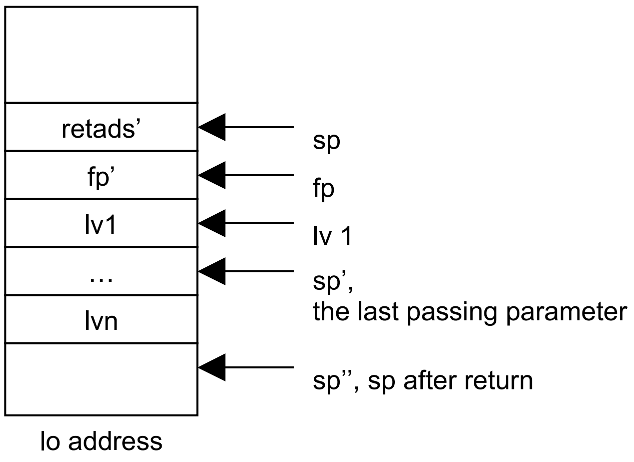


Figure 4.1 The activation record to support S-code

A function call creates a new activation record. The new FP is *SP + k.* The value k is the argument of "*fun k*", *k = n - arity + 1.* The new activation record overlaps the evaluation stack such that the passing parameters become the local variables of the new activation record. A local variable is indexed by an offset from the current FP. The numbering of the local variables causes the first passing parameter to be the *n-th* local variable and so on. This fact is handled by the compiler (Chapter 3). A function call does the following:

```
1  Decode k at function header
2  Create new activation record, save old FP
3  Set new SP
4  Save return address
5  Goto body of function
```

When returning, the return instruction, "`ret m`", supplies a value *m* to be used to restore SP. *m* is size of activation record + 1. When restoring SP (not considering the return value yet):

```
SP'' = FP - m
```

A return does the following:

```
1  Restore PC
2  If there is a return value
3     Restore SP and FP
4     Push the return value
5  Else
6     Restore SP and FP
```

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

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

A `case` does:

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

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

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

```
a = a + 1

     get a, lit 1, add, put a

a = b[i]

     get b, get i, ldx, put a

d[i] = b

     ld d, get i, get b, stx

e = add2(a,b)

     get a, get b, call add2, st e

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

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

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

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

is compiled into N-code object,

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

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

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

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

It means the following:

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

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

Table 4.1 Mapping between N-code and S-code

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

4.3 How the code generator work?

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

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

```
eval e [gen 224]               ; S-code generator
...                            ; e1 is the argument list
switch op
ADD
 eval head e1
 eval arg2 e1
 out icAdd
LIT
 out icLit arg
GET
 out icGet arg
FUN
 insertLab ads XP              ; ads is N-code, XP is S-code
 lv = arg & 255
 arity = arg >> 8              ; decode a.v
 out icFun (lv-arity+1)
 eval head e1
 out icRet (lv+1)
CALL
 while e1 not empty            ; generate all arguments
   eval head e1
   e1 = tail e1
 out icCall (assoc arg)        ; map address to S-code
...
else
 error "unknown op"
```

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

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

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

The pattern for code generation is:

```
(xIF e1 e2 e3)

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

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

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

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

```
(xWHILE e1 e2)

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

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

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

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

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

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

```
; e = (cond true false)

(def genif e (ads e3)   [gen 186]
  (do
  (eval (head e))                  ; gen cond
  (outa icJf 0)
  (set ads (- XP 1))
  (eval (arg2 e))                  ; gen if-true
  (set e3 (arg3 e))
  (if (= e3 NIL)
    (patch ads (- XP ads))
    (do                            ; else
    (outa icJmp 0)
    (patch ads (- XP ads))
    (set ads (- XP 1))
    (eval e3)                              ; gen else
  (patch ads (- XP ads))))))

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

; change arg, preserve op

(def patch (ads v) ()   [gen 167]
  (setv XS ads (+ (<< v 8) (& (vec XS ads) 255))))
```

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

```
; n1 is the label, n2 is the address

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

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

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

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

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

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

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

4.4 Three-address code generation

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

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

S2 description

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

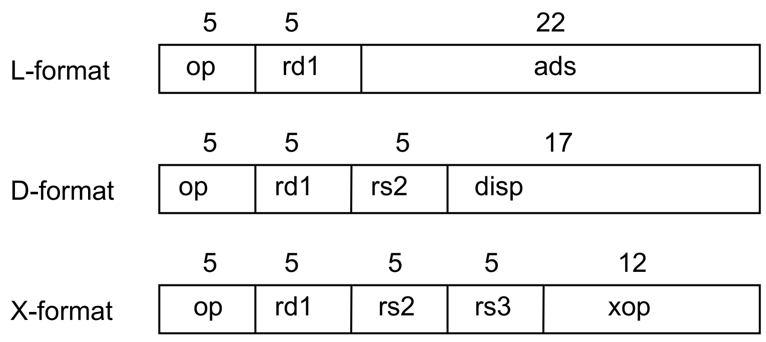
S2 addressing mode

S2 has four addressing modes: absolute, displacement, index, and immediate. The *absolute* mode has 22-bit range (0..4M). The *displacement* mode uses one register and a 17-bit value (0..128K). The *index* mode employs two registers. Lastly, the *immediate* mode uses the literal value in the instruction. Depend on what instruction the literal is 22-bit (load/store) or 17-bit (arithmetic). For example, to load a value from memory into a register, all four addressing modes are as follows:

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

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

S2 instruction format



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

Figure 4.2 S2 instruction format

Opcode encoding

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

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

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

Meaning

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

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

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

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

The arithmetic operations are two-complement integer arithmetic.

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

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

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

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

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

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

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

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

How an expression be transformed into sequence of instructions

An instruction in machine language composed of an operator and operands. The number of operands varies from zero (stack instruction), one, two, and three. Our hypothetical S2 processor is a 3-operand machine. Each instruction has the form "`op r1 r2 r3`" and all operands are registers. Having three operands means each operation can take two inputs from two operands and stores the result in the third operand. It is suitable for binary operations such as; `add`, `sub` etc. To translate an expression into S2 instructions, each result of a binary operation needs to store in a temporary register. For example, the following expression in transformed into a sequence of simple 3-operands instructions using `t1`, `t2`, `t3` as temporary registers.

```
a * b + c - d

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

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

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

In fact, at most two temporary variables are needed for any arbitrary long sequence of binary operations (not nested) as the temporary value can be accumulated using just one register and another register is used to hold one operand of the binary operator.

Let use only `t1`, the previous expression becomes,

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

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

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

This expression is transformed into:

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

Another example,

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

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

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

Access simple scalar

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

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

can be translated into the following sequence.

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

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

Access an array

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

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

can be translated into the following sequence.

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

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

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

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

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

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

Using jump for conditional branching

```
jmp cond ads
```

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

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

Using jump to do if-then-else

For an "if-then-else" expression, S2 instructions are generated using conditional jumps. The following example shows the skeleton of the generated code.

```
(if (> a b) e1 e2)
```

Let `r1` = `a`, `r2` = `b`,

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

Generate code for a simple while loop

The "while" expression can be translated to S2 code as follows.

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

Let `r1` = `s`, `r2` = `i`

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

Function call

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

```
jal rx ads
```

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

```
jr rx
```

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

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

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

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

The above program can be translated into S2 code as follows.
Let `r1` = `a`, `r2` = `b`, `r3` = `x`, `r4` = `link`, `r5` be return value.

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

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

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

To push a register "*x*",

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

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

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

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

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

And similarly for popping multiple values.

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

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

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

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

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

This program can be translated into S2 code as follows.

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

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

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

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

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

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

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

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

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

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

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

4.5 Lab session

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

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

And use Nut-compiler to compile a program, let it be "`t3.txt`", the output goes to "`t3.obj`":

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

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

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

Use it to generate the final S-code object:

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

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

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

Run it.

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

4.6 Summary

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

To illustrate the method of generating object code for a conventional processor, a register-based processor, S2, is demonstrated. One important aspect of generating code for a register-based instruction set is that of register allocation.

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

References

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

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

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

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

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

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

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

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

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

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

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

Exercises

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

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

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

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

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

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

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