Chapter 6

Sx Processor

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

6.1 Data path

Sx has seven special purpose registers (no visible user registers): `TS`, `FP`, `SP`, `NX`, `FF`, `IR` and `PC`. `TS` caches the top of stack value (Fig. 6.1).

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

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

`PC+arg` or `tbus`. The ALU has two ports: `p1`, `p2` and can perform many functions as shown in Table 6.1. There are two flags: Zero, and Sign.

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

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

The instruction register, `IR`, has the operation code at the right-most 8-bit and the argument at the left-most 24-bit. The argument field is signed extended to 32 bits. When the instruction requires no argument, the argument field is zero.

24	8
arg	op

Using 2-phase clock enables read-modify-write of registers in one cycle. Reading from registers and memory will be on the positive edge and writing to registers will be on the negative edge. The basic cycles in the control unit are:

- read-modify-write registers
- register transfer
- memory read
- memory write

Memory access

Before going into details of each control cycle, one important consideration is how the memory is accessed (read/write) in each control cycle. A memory access is assumed to take a full cycle. The memory access time is assumed to be half of the processor cycle. This is not a realistic assumption. Usually the memory cycle time is much longer than the processor cycle time, as much as ten

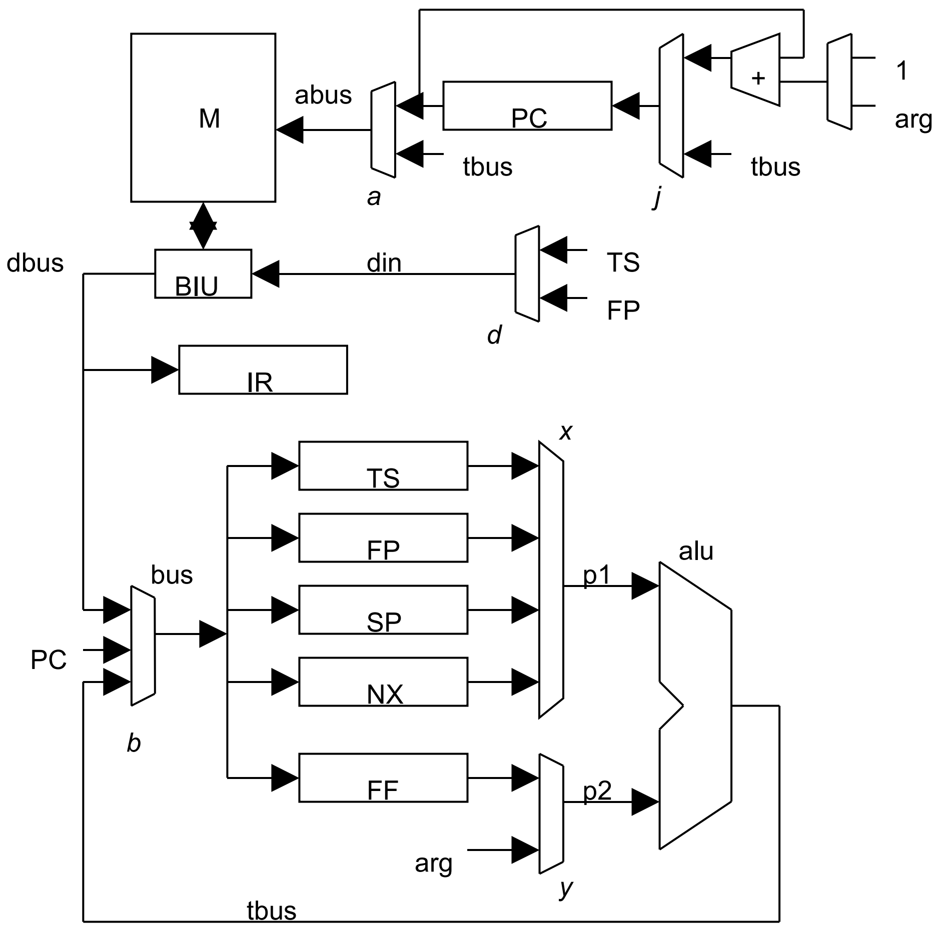

Figure 6.1 The Sx data path

times. This is called "processor memory speed gap". However, we make this assumption as it simplifies our control cycle greatly.

A memory access is initiated by setting the address through `abus`, for a read, a memory read signal is asserted (*mR*). The data from the memory is ready at the middle of the cycle. The data from `dbus` is latched to a register in the middle of the cycle, at the negative edge of the clock.

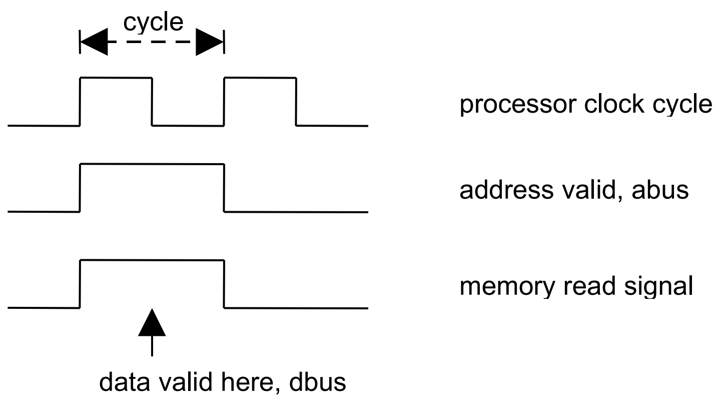

Figure 6.2 A memory read cycle

A memory write cycle is similar. The address and data are asserted at the beginning of the cycle. The memory write signal is asserted (*mW*). The data will be written in the memory in the middle of the cycle, at the negative edge of the clock.

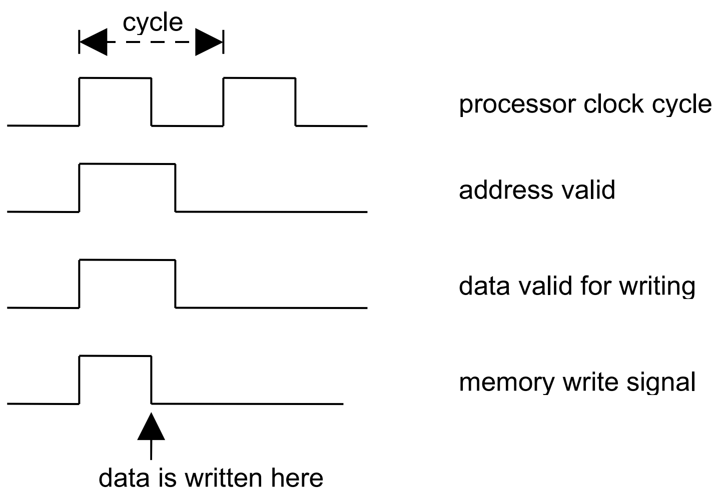

Figure 6.3 A memory write cycle

Register access

The basic read-modify-write starts at the positive edge of the clock. The data are read from the registers into the ALU ports through the multiplexor *x* and *y*. The ALU outputs the result to `tbus`. At the negative edge, `tbus` is fed back to the input of registers, `bus`, through the multiplexor *b* and is latched into the designated register.

Read-modify-write a register

```
pos edge: R -> alu -> tbus
neg edge: tbus -> R
```

Register transfer

```
pos edge: R1 -> tbus
neg edge: tbus -> R2
```

6.2 Execution cycle

The processor begins its execution cycle with fetching an instruction from the memory. It is complete in the first half of the cycle. The instruction in stored in the instruction register (IR). It is decoded through a read-only-memory, called micro-ROM, that stored the address of the microprogram control. The control step then transfers to the appropriate microprogram step. At the end of microprogram step of the instruction, the control is transferred back to fetch the next instruction. A register transfer language (RTL) is used to describe these steps of execution. RTL notation mainly describes the transfer between two registers, `dest = source`. In our notation, RTL does not specify the actual concurrent operation beyond what that can be written as `dest = source`. We will fully specify the concurrent operations in the control unit using the microprogram notation later.

Execution cycle in RTL

The registers in the data path are *IR*, *TS*, *FP*, *SP*, *NX*, *FF*, and *PC*. In some operation that there are a number of arguments, the picture of the data in the evaluation stack will be shown in this notation, {.. top of stack}. Each operation is labeled as <*op*>. `M[.]` is the memory.

A shorthand notation is used to describe two often used stack operations: `push` and `pop`.

```
[push x]
sp = sp + 1
M[sp] = x

[pop x]
x = M[sp]
sp = sp - 1
```

The instruction fetch cycle is,

```
ir = M[pc].
```

An operation on the ALU is specified by the operation code field. The opcode bits determine the ALU function. The binary operations are: `add`, `sub`, `mul`, `div`, `band`, `bor`, `bxor`, `shl`, `shr`, `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `inc`, `dec`. In a binary operation, the second argument is in the top of stack; the first argument is in the evaluation stack pointed to by `SP`. Please note the order of argument. The second argument is popped to `FF`, and then two arguments are fed to the ALU. The result is stored back to `TS`.

```
<bop>
pop ff
ts = ts op ff
```

The unary operation affects only the `TS`.

```
<uop>
ts = op ts
```

The access operations to local variables are "`get`" and "`put`". "`get`" must pushes `TS` first to make room for the new data that will be taken from the activation record, `M[FP-arg]`. "`put`" stores `TS` to the activation record then it pops the evaluation stack to `TS` (caching the top of stack).

```
<get>
push ts
ts = M[fp-arg]

<put>
M[fp-arg] = ts
pop ts
```

"`ld`" and "`st`" are similar to "`get`" and "`put`" but access to the memory instead of the activation record.

```
<ld>
push ts
ts = M[arg]

<st>    {data}
M[arg] = ts
pop ts
```

The "`ldx`" and "`stx`" are a bit more complicate as they have a number of arguments. "`ldx`" takes the `base` from the stack, using `FF` to store it. "`stx`" takes two arguments from the stack, the first one is `idx`, and the second one is `base`. The effective address is calculated using the ALU.

```
<ldx>              {base idx}
pop ff             base
ts = M[ff+ts]

<stx>              {base idx data}
pop nx             idx
pop ff             base
M[ff+nx] = ts
pop ts
```

The literal instruction is simply pushing the argument to `TS`.

```
<lit>
push ts
ts = arg
```

The control transfer operations are: unconditional jump, conditional jump, call and return. "`jmp`" is straightforward. "`jt`" and "`jf`" inspect the zero flag, which reflected the value of *TS*, and transfer the control step accordingly. The evaluation stack is popped to get rid of the old *TS*.

```
<jmp>
pc = pc + arg

<jt>
if ts != 0
 pc = pc + arg
else
 pc = pc + 1
pop ts

<jf>
if ts == 0
 pc = pc + arg
else
 pc = pc + 1
pop ts
```

The "*call*" is perhaps the most complex instruction in this instruction set. It creates a new activation record and transfers the control step to the called function. The new activation record is created on top of the current evaluation stack, overlapping the evaluation stack by the amount of the arity of the called function to pass the parameters. Hence, the new *FP* is offset from the current *SP*. This offset is computed by the compiler and it becomes the argument of the function header, the "*fun*" instruction. "*call*" fetches the function header to get the offset, then uses the offset to set up a new *FP* location and saves the current *FP* there. The *FP* and *SP* are updated to the new location. Next, it pushes the return address and finally jumps to the function body.

```
<call ads>
push ts                 flush eval stack
ts = pc + 1             save ret ads to ts
nx = arg                save call ads to nx
ir = M[arg]             fetch at ads
M[fp+arg] = fp          save old fp
fp = sp = fp + arg      new fp, sp
push ts                 save ret ads
pc = nx + 1             jump to body
```

The "`ret`" instruction sets `PC` to the return address, restores the old `SP`, and restores to the previous activation record. As it is different between returning and not returning a value, it is necessary to decide whether there is a return value or not. The condition `SP` = `FP` indicates that the *net effect* of the evaluation stack (the state of stack after many operations) is that the stack is back to its initial state, there is no value to return. The argument of "`ret`" is the offset to set `SP` back.

```
<ret>
pc = M[fp+1]          restore ret ads
if sp == fp           no return value
 sp = fp - arg       restore sp
 pop ts              cache top of stack
 fp = M[fp]          restore fp
else                  return a value
 sp = fp - arg
 fp = M[fp]
```

If the *net effect* cannot be assumed (because some anomaly in the stack manipulation), then an alternative is to do flow analysis at the compile time to decide whether a function returns a value or not. The "`ret`" instruction must be spilt into two instructions, one without a return value and one with it. Let it be "`ret`" and "`retv`", then the following steps are their execution cycles.

```
<ret>
pc = M[fp+1]
sp = fp - arg
fp = M[fp]

<retv>
pc = M[fp+1]
sp = fp - arg
pop ts
fp = M[fp]
```

For simplicity, we assume the net effect is proper. This assumption let us avoid the flow analysis in the compiler.

Microprogram

Next, we describe the actual microprogram level. The whole microprogram on Sx processor is presented in the appendix H. The difference between RTL and microprogram is that microprogram specifies the concurrent operations on the data path, including the signals asserted on the multiplexor and ALU. The microprogram level exposed more details that are necessary to realise on actual circuits. A control signal in the microprogram can be regarded as an event that occurs in the data path, such events are latching a data to a register, selecting a multiplexor, memory read, memory write, etc. The notation used in writing microprogram is as follows.

```
src->dest
```

denotes the event that transfer data from a source to a destination where source and destination can be a wire or a register. A wire represents a connection or the input/output of a component.

```
alu(p1 op p2)->dest
```

denotes the ALU performing the "*op*" on its two input ports, `p1` and `p2`, and its output is connected to `dest`, where `dest` can be a wire or a register.

```
mR(ads)->dest
src->mW(ads)
```

`mR` denotes memory read with the address from the source `ads`, the data is transferred to `dest`. `mW` denotes memory write with the address sets to the source and the address is `ads`. `src` and `dest` can be a wire or a register. The concurrent events are specified in the microprogram by writing them on the same line. Each event is separated from other event by ",". The order of events in the same line is unimportant because they occur in the same clock cycle. However, some event occurs on the positive edge of the clock, some event occurs on the negative edge of the clock. Reading from registers and memory will be on positive edge and writing to registers will be on negative edge.

```
src->dest, mR(ads)->dest, ...
```

The "jump" of the microprogram is achieved by loading the "next microaddress" bit to the microprogram counter. It can be unconditional or conditional. The *next address* is written as `<label>`. There are three "jump" events in Sx data path.

`ifT`	jump if ts is not zero
`ifF`	jump if ts is zero
`decode`	multiway branch according to opcode

`PC` has special events.

```
pc+1      is increment PC by 1
pc+arg    is increment PC by arg
```

We have a shorthand notation for `SP`.

```
sp-1    is alu(sp-1)->sp
sp+1    is alu(sp+1)->sp
```

The microprogram for Sx is followed from its RTL description. We begin with the instruction fetch.

```
<fetch>   [micro 47]
mR(pc)->ir, decode
```

Where `decode` is a control signal to look up the microprogram address according to the opcode field on the instruction register, `IR`.

Next is the binary operation.

```
<bop>   [micro 49]
mR(sp)->ff
sp-1
alu(ts op ff)->ts, pc+1, <fetch>
```

Please note that the `PC` is incremented at the end of the instruction cycle and then the microprogram is jumped back to the instruction fetch at the beginning. The unary operation changes the value on `TS`.

```
<uop>   [micro 53]
alu(ts op ?)->ts, pc+1, <fetch>

<get>   [micro 55]
sp+1
ts->mW(sp)
alu(fp-arg)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

"`get`" pushes `TS` and loads `M[FP-arg]` using ALU to do the effective address calculation. The address is presented to `tbus` (and then to `abus`) and the memory read signal is asserted. The data is ready and is latched to `TS`.

```
<put>   [micro 59]
alu(fp-arg)->tbus, ts->mW(tbus)
mR(sp)->ts
sp-1, pc+1, <fetch>
```

"*put*" writes *TS* to *M[fp-arg]* then pops a value to *TS*. The *SP*-1 and *PC*+1 can be concurrent because *SP*-1 uses the ALU while *PC*+1 does not use ALU. *PC* has its own adder.

```
<ld>   [micro 64]
sp+1
ts->mW(sp)
mR(arg)->ts, pc+1, <fetch>

<st>   [micro 68]
ts->mW(arg)
mR(sp)->ts
sp-1, pc+1, <fetch>
```

"*ld*" and "*st*" are similar to "*get*" and "*put*" but "*ld*" and "*st*" access the memory using direct address from the argument of the instruction.

```
<ldx> [micro 70]                {ads idx}
mR(sp)->ff
sp-1
alu(ff+ts)->tbus, mR(tbus)->ts, pc+1, <fetch>
```

"*ldx*" gets the base address to *FF*. The *index* is in *TS*. The effective address is calculated using ALU and the value is fetched from the memory.

```
<stx>  [micro 74]               {ads idx val}
mR(sp)->nx                      pop idx to nx
sp-1                            pop ads to ff
mR(sp)->ff
alu(nx+ff)->tbus, ts->mW(tbus)
sp-1
mR(sp)->ts                      cache ts
sp-1, pc+1, <fetch>
```

"*stx*" has three arguments. It gets the *index* to *NX*, and the base address to *FF*. The effective address is calculated using ALU. The value in *TS* is stored to that address. Finally, the top of stack is cached to *TS*.

```
<lit>   [micro 80]
sp+1
ts->mW(sp)
arg->ts, pc+1, <fetch>

<jmp>  [micro 84]
pc+arg, <fetch>

<jt>   [micro 86]
alu(ts=0), ifT <j3>       if true, don't jump
<j2>                      jump
pc+arg, mR(sp)->ts
sp-1, <fetch>

<jf>   [micro 92]
alu(ts=0), ifT <j2>       if true, jump
<j3>                      don't jump
pc+1, mR(sp)->ts
sp-1, <fetch>
```

The "*jt*" and "*jf*" use the event "*ifT*" to do conditional branching. The branching is the "goto" style of programming which is quite natural in a microprogram. It saves the microprogram space.

```
<call>   [micro 98]
sp+1
ts->mW(sp), pc+1                      flush stack
pc->ts
arg->tbus->nx, mR(tbus)->ir           fetch fun, nx=ads
alu(sp+arg)->tbus, fp->mW(tbus)       save old fp
alu(sp+arg)->fp->sp                   new fp, sp
alu(nx+1)->pc, <fetch>
```

The event "*arg->tbus->nx*" uses ALU to pass *arg* through. This event saves the address of the called function to *NX*. To get the offset, the "*fun*" instruction is fetched to *IR* then its argument is used. There are two concurrent register writes in the event "*alu(sp+arg)->fp->sp*". The address of the body of the function, *NX*+1, is updated to *PC*.

```
<ret>   [micro 106]
sp->ff
alu(fp=ff), ifF <r2>
ts->pc                               do ret
alu(fp-arg)->sp
mR(sp)->ts
sp-1
mR(fp)->fp, <fetch>                  restore fp
<r2>                                 do retv
alu(fp+1)->tbus, mR(tbus)->ff        ret ads
ff->pc
alu(fp-arg)->sp
mR(fp)->fp, <fetch>
```

The "`ret`" tests the condition `FP` = `SP` to decide whether there is a value to return or not. To do the test, `SP` is moved to `FF` to use ALU operation. When there is no value to return, the return address is in `TS` but when there is a value to return, the return address is in `M[FP+1]`. `FF` is used to pass the value through `PC`.

```
<sys>   [micro 119]
<array>
<end>
trap, pc+1, <fetch>
```

The instructions "`sys`", "`array`" and "`end`" have no implementation on the real processor. They are used in the simulator. The event "`trap`" is used by the simulator to handle these instructions.

After the microprogram is completely specified, the number of cycle taken by each instruction is known. They are shown in the table below.

Table 6.2 The number of cycle for each instruction

bop 4	uop 3	get 4	put 4
ld 4	st 4	ldx 4	stx 8
lit 4	jmp 2	jt 4	jf 4
call 8	ret 8	retv 7	

6.3 Performance

A number of benchmark programs are compiled and then run on the Sx processor simulator. Table 6.3 reports the number of instructions (noi), the number of cycles (cycle) and the cycle-per-instruction number (cpi) for each program.

"bubble" is a bubble sort program sorting an array of 20 integers, initially the value in the array is in descending order and sort to ascending order. "hanoi" is a program to solve Hanoi problem with 6 disks. "matmul" is a matrix multiplication program; the input is two matrices of the size 4 × 4. "perm" is a program to do all permutation of {0,1,2,3}. "queen" is a program to find all configurations of 8-queen problem. "quick" is a quicksort program with a similar input to "bubble". "sieve" is a program to find prime numbers less than 1000 using "Sieve of Eratosthenes" algorithm. "aes" is a program to do AES (Advanced Encryption Standard) block cipher (128, 128) bit key. The average cycle-per-instruction number of Sx processor is 4.3. This is quite good comparing to the stack-based processor of an earlier design [BUR04c], a 16-bit processor runs the same "aes" in 284108 cycles. Sx processor completed it in 131560 cycles, twice as fast at the same clock frequency.

Table 6.3 The performance of Sx processor

program	noi	cycle	cpi
bubble	10068	44214	4.39
hanoi	2312	10092	4.37
matmul	3043	12880	4.23
perm	4868	20932	4.30
queen	618665	2576210	4.16
quick	3172	13539	4.27
sieve	28026	124338	4.44
aes	30579	131560	4.29

6.4 Sx processor simulator

The design of Sx processor with the detailed design of the data path and its control unit using microprogram is complete enough to be realised on real silicon using either FPGA (Field Programmable Gate Array) technology or ASIC (Application Specific Integrated Circuit). However, it is much easier to study it using a simulator. The Sx processor simulator performs cycle-accurate simulation of Sx processor executing programs. The simulator executes step-by-step microprogram of Sx. It is used to validate the microprogram and to collect the performance statistics.

Data path

The data path consists of registers, multiplexors, combinational circuits such as ALU and wires. The registers and wires are simulated as variables of type integer capable of holding 32-bit values. The multiplexors are simulated as if-then statements to update the output wires. The simulated ALU performs the expected operations on its input ports and updates the flags. The combinational circuits can be simulated by statements to update the output wires.

Control unit

A straightforward way to simulate the microprogram control unit is to regard the microprogram as a ROM, a two-dimensional array of bits (Fig. 6.4). Each address is called a microprogram word. One microprogram word is executed in one cycle. Each word contains event-control bits where each bit represents an event in the data path. The event that is active is 1, otherwise it is 0. Many events can occur simultaneously in one cycle. Each event has its symbolic name. The simulation is run as event-driven. The main simulation loop looks at each microprogram word and scans the event bits to find the active one then performs the action for that event. This includes the control transfer of microprogram address which updates the microprogram counter. The simulation loop continues until the "*end*" instruction throws a trap with the event "*trap*". The "*trap*" events are system specific. They implement the input/output and other useful functions.

```
62
 0 00000000000010000000000001000000100010000001
 1 00100001000100000000000100000010100000000010
 2 00100010000000000001000000001000000000000011
 3 10001010000000100000000100010000100000000000
 4 10000010000000100000000100010000100000000000
. . .
55 00010010000000100000000010010000100000000000
56 10000010000000000000000100000100000000111001
57 01000101000100000100000000100000100000111010
58 10000010000000000001000000100000000000111011
59 01000100001100000100000000000000010000111000
60 00010010000000100000000010010000100000000000
61 00000000000001000000000000000001000001000000
```

Figure 6.4 The microprogram ROM. The first line shows the length of the microprogram. The first column of each line is the address of the microword.

The events are defined as follows.

multiplexor *x* selects {*ts, fs, sp, nx*}
multiplexor *y* selects {*ff, arg*}
multiplexor *b* selects {*tbus, dbus, pc*}
multiplexor *d* selects {*fp, ts*}
multiplexor *a* selects {*pc, tbus*}
multiplexor *j* selects {*pc+1, pc+arg, tbus*}
ALU events are {*add, sub, inc, dec, z, eq, op, p1, p2*}
load registers events are {*ir, ts, fp, sp, nx, ff, pc*}
memory events are {*mR, mW*}
next micro-address events are {*ifT, ifF, decode, trap*}

We use the naming convention as follows. The multiplexor has its name as a prefix followed by its choice, for example, mux *x* selects *TS* is written as *x.ts*. The ALU is similar, ALU performs *inc* is written as *alu.inc*. The load register is written with a prefix "*l*" followed by the name of the register, *lpc* is load *PC*.

The registers are *IR*, *PC*, *TS*, *FP*, *SP*, *NX*, *FF*. *Z* is the zero flag. The wires are *p1, p2, tbus, abus, dbus, bus, pcin*. The functions *IRarg()*, *IRop()* decode the *op* and *arg* field of *IR*. *alu()* performs ALU operations. *udecode()* returns the microaddress corresponded to the current opcode. `m2` is the next microaddress, specified as the next address field in the microprogram word.

Let `mx[mpc][bit]` be the microprogram ROM. The main simulation loop is.

```
while (running)
 m2 = next micro address field
 for i = 0 to microwidth-1
  s = scan for active event in mx[mpc][i]
  do s
 mpc = m2
```

For each event in a microprogram word. Let *s* be the event that is active.

```
switch(s){  [sx 120]
  case x.ts:     p1 = TS
  case x.fp:     p1 = FP
  case x.sp:     p1 = SP
  case x.nx:     p1 = NX
  case y.ff:          p2 = FF
  case y.arg:         p2 = IRarg()
  case alu.add:  tbus = alu(icAdd,p1,p2)
  case alu.sub:  tbus = alu(FSUB,p1,p2)
  case alu.inc:  tbus = alu(icInc,p1,p2)
  case alu.dec:  tbus = alu(icDec,p1,p2)
  case alu.z:    tbus = alu(FZ,p1,p2)
  case alu.eq:   tbus = alu(icEq,p1,p2)
  case alu.p1:   tbus = alu(FP1,p1,p2)
  case alu.p2:   tbus = alu(FP2,p1,p2)
  case alu.op:   tbus = alu(IRop(),p1,p2)
  case a.pc:          abus = PC
  case a.tbus:   abus = tbus
  case d.ts:     dbus = TS
  case d.fp:     dbus = FP
  case mR:       dbus = M[abus]
  case mW:       M[abus] = dbus
  case b.tbus:   bus = tbus
  case b.dbus:   bus = dbus
  case b.pc:     bus = PC
  case j.pc1:    pcin = PC + 1
  case j.pcarg:  pcin = PC + IRarg()
  case j.tbus:   pcin = tbus
  case lpc:      PC = pcin
  case lir:      IR = dbus
  case lts:       TS = bus
  case lfp:       FP = bus
  case lsp:       SP = bus
  case lnx:       NX = bus
  case lff:       FF = bus
  case ifT:       m2 = (Z == 0) ? m2 : mpc+1
  case ifF:       m2 = (Z == 1) ? m2 : mpc+1
  case decode:    m2 = udecode()
  case trap:      trap(IRop(),IRarg())
}
```

The simulator is sequential, that is, it simulates each event one by one. Therefore the order of scanning the event (the bits in a microprogram word) is important to get the correct result. All the positive-edge events must be updated before the negative-edge events. Within the same group the input side is updated to the output side. For example the read-modify-write loop of a register, the read side must be performed, then goes through the modify operation from input to output, finally the write is performed to that register. With these rules the order of events are:

```
mux x, mux y, alu,
mux a, mux d, mR, mW,
mux b, mux j,
load registers,
ifT, ifF, decode, trap.
```

6.5 Lab session

A tool is provided to write a microprogram. The “mgen” tool takes the input file as a microprogram specification and outputs the microprogram ROM as shown in Fig 6.4. The microprogram must be written in the following form. The specification composed of two sections, the first section is the signal definition and the second section is the microprogram. The signal definition lists all the events, the order of the event is important as a simple implementation of the simulator will simulate each event according to this order (see Exercise 6.7). This can be relaxed in the alternative implementation. Here is an example of the signal definition, the section starts with “*.s*”. The line started with “..” is the comment line.

```
.. sx microprogram v 1.0   [micro 1]
..
.s
x.ts
x.fp
x.sp
...
alu.add
alu.sub
...
.. load registers
lir
lts
lfp
lsp
lnx
lff
lpc
mR
mW
.. next micro ads
ifT
ifF
decode
trap
```

After the signal definition the next section is the microprogram section. Each line consists of,

```
[:label]  event* [/label] ;
```

A line starts with a label "`:label`", follows by events and the next micro-address label "`/label`", and ends with "`;`". The starting label and the micro-address label are optional. The microprogram section starts with "`.m`" and ends with "`.e`". Here is an example.

```
.m          [micro 45]
:fetch
 a.pc mR lir decode ;
:bop
 x.sp alu.p1 a.tbus mR b.dbus lff ;
 x.sp alu.dec b.tbus lsp ;
 x.ts y.ff alu.op b.tbus lts j.pc1 lpc /fetch ;
...
```

```
:jmp
 j.pcarg lpc /fetch ;
:jt
 x.ts alu.z ifT /j3 ;
:j2
 j.pcarg lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
 x.sp alu.dec b.tbus lsp /fetch ;
:jf
 x.ts alu.z ifT /j2 ;
:j3
 j.pc1 lpc x.sp alu.p1 a.tbus mR b.dbus lts ;
 x.sp alu.dec b.tbus lsp /fetch ;
:end
 trap j.pc1 lpc /fetch ;
.e
```

The microprogram word for "*fetch*" can be read as, mux *a* selects *PC* (to be the address of the memory operation), memory read, load *IR*, jump to the corresponding micro-address. The "*bop*" reads as *SP* goes through ALU to *tbus*, mux *a* selects *tbus* (to be the address of the memory operation), memory read, mux *b* selects *dbus* (to be the input of registers), load register *FF*. Then, *SP* goes through ALU to do −1 and back to bus to write to *SP*. Then, mux *x* selects *TS*, mux *y* selects *FF*, ALU performs a function according to the opcode, mux *b* selects *tbus* (to be the input of registers), load register *TS*, at the same time, *PC* is updated +1, then jump to "*fetch*", the instruction fetch.

How to microprogram Sx

To write microprogram for Sx, the microprogram specification is in the file "mspec.txt". "mgen" transforms the specification to a ROM file. Store it in the name "mpgm.txt". The source for "mgen" can be found in sx0.zip package.

```
c:> mgen < mspec.txt > mpgm.txt
```

Here is what "mpgm.txt" looked like.

```
62
 0 00000000000010000000000001000000100010000001
 1 00100001000100000000000100000010100000000010
 2 00100010000000000001000000001000000000000011
 3 10001010000001000000001000100001000000000000
 4 10000010000001000000001000100001000000000000
. . .
55 00010010000001000000000100100001000000000000
56 10000010000000000000000100000100000000111001
57 01000101000100000100000000100000100000111010
58 10000010000000000001000000100000000000111011
59 01000100000110000100000000000000010000111100
60 00010010000001000000000100100001000000000000
61 00000000000001000000000000000001000001000000
```

A few right most bits are the next microprogram address. “mgen” also generates binding of symbolic names to numeric values which are used in the simulator, “mspec.h”. This is it:

```
#define s_x_ts 0
#define s_x_fp 1
#define s_x_sp 2
#define s_x_nx 3
#define s_y_ff 4
. . .
#define a_fetch 0
#define a_bop 1
#define a_uop 4
#define a_get 5
#define a_put 8
#define a_popts 9
#define a_ld 11
. . .
#define a_end 61
#define MCWIDTH 38
#define MAWIDTH 6
#define MLEN 62
```

The event names prefixed “s_” are the signal events, prefixed “a_” are the address of the label of microprogram. The MCWIDTH is the number of the control bits. The MAWIDTH is the number of bit of the microprogram address field. The MLEN is the number of microprogram word. “mgen” also generates a listing file “mlist.txt”. It is used for debugging.

The next step is to convert this micro-ROM into an event-list. “sxgen” combines “mgen” and conversion to event-list. (If you are using sx1, sx2 simulator, sxgen is inside the simulator, you don’t have to do it explicitly). “sxgen” reads the files "mpgm.txt" and "mspec.h" then generates "sxbit.h" which must be compiled with the simulator.

```
c:> sxgen < mspec.txt
```

The "sxbit.h" contains the binding, the op-decoder-rom (`udop[]`), the pointer to event-list (`mw[]`), the event-list itself (`mx[]`) and finally the next-address-rom (`nxt[]`).

```
#define s_x_ts 0
#define s_x_fp 1
. . .
#define a_sys 51
#define a_array 51
#define a_end 51
#define MCWIDTH 38
#define MAWIDTH 6
#define MLEN 52

int udop[] = {
0, 1, 1, 1, 1, 1, 1, 1, 4, 1,
1, 1, 1, 1, 1, 1, 1, 0, 15, 18,
40, 0, 51, 51, 5, 8, 11, 14, 26, 27,
30, 23, 33, 0, 0, 0, 51, 0, 0, 0, 0 };

int mw[] = {
0, 5, 12, 17, 25, 32, 37, 43, 53, 60,
67, 74, 79, 85, 94, 100, 107, 112, 122, 129,
134, 141, 148, 153, 158, 164, 171, 174, 178, 187,
192, 196, 205, 210, 215, 223, 226, 234, 241, 248,
253, 258, 263, 268, 274, 281, 286, 293, 300, 305,
311, 318, 0 };

int mx[] = {
12, 32, 25, 36, -1,
2, 23, 11, 32, 7, 30, -1,
2, 19, 6, 28, -1,
0, 4, 22, 6, 13, 31, 26, -1,
. . .
1, 5, 17, 6, 28, -1,
1, 23, 11, 32, 7, 27, -1,
13, 31, 37, -1,
0 };
```

```
int nxt[] = {
1, 2, 3, 0, 0, 6, 7, 0, 9, 10,
0, 12, 13, 0, 9, 16, 17, 0, 19, 20,
21, 22, 9, 24, 25, 0, 0, 31, 29, 0,
28, 32, 0, 34, 35, 36, 37, 38, 39, 0,
41, 47, 43, 44, 45, 46, 0, 48, 49, 50,
0, 0, 0 };
```

You must recompile the simulator to include your new signal definitions, or new instruction labels. The processor simulator takes a proper object file as input and run it. If it is `sx0`, the processor is run in a batch mode. For `sx1` and `sx2` they run in interactive mode. You can ask for help by typing "h". The following session is `sx0` running quicksort.

```
D:\sx\test>sx qs.obj
load program, last address 203
DP 1000
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4820 instructions, 20231 clocks, CPI 4.20

D:\sx\test>
```

6.6 Summary

In this chapter we have learnt the detailed design of a stack-based processor that can execute S-code instructions directly. This processor is the basic component of our hardware system. The data path is quite simple. The processor is aimed for clarity. It is simple enough to be studied and to be modified without due complexity by students. At the same time, it retained the flavour of reality that the design is complete enough to be realised as a real processor. The control unit has been detailed down to the cycle-by-cycle execution of an instruction. Microprogram represents the specification of the execution behaviour. Writing a microprogram starts with a RTL description which concerns only the registers transfer. Then, the specification is refined to a microprogram level which also concerns concurrency of operations. With microprogram fully specified, the number of cycle taken by each instruction can be calculated. The algorithm to simulate the processor has been presented. The processor simulator is event-driven. The microprogram is regarded as events occur in the data path.

In the laboratory session, we learn how to write a concrete microprogram. Tools are given to convert this concrete specification into a data structure suitable for simulation. The processor simulator itself has been described in details enough that students can modify it to include other instructions and/or additional features. We shall see in the next chapter how to improve the performance of this Sx processor.

6.7 Further reading

Stack-based processors have been a popular architecture in the past due to their simplicity and their compatibility with *structured programming* paradigm around 1970-1980. Burroughs has developed many commercial machines based on this type of architecture (Burroughs B5500) [BUR68]. For more recent discussion of stack-based processors see Koopman [KOO89] which discussed the strength of this architecture including a comprehensive survey of many stack-based processors. The weakness of stack architecture lies in its performance. As RISC type of processors [PAT82] [PAT85] [HEN84] [STA88] becomes popular during 1980-1990, it dominates all the market with its high performance. Nowadays, all processors are register-based. We shall discuss the use of registers in the next chapter.

References

[BUR68] Burroughs B5500 Electronic Information Processing System: Operation manual. Burroughs Corp. Detroit, 1968.

[BUR04a] Burutarchanai, A., and Chongstitvatana, P., “Design of a two-phased clocked control unit for performance enhancement of a stack processor”, National Computer Science and Engineering Conference, Thailand, 21-22 Sept. 2004, pp.114-119.

[BUR04b] Burutarchanai, A., Kotrajaras, V. and Chongstitvatana, P., “A fast instruction fetch unit for an embedded stack processor”, Proc. of Int. Conf.

on Information and Communication Technologies, Thailand, 18-19 November 2004.

[BUR04c] Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., “A stack-based processor for resource efficient embedded systems”, Proc. of IEEE TENCON, Thailand, 21-24 November 2004.

[HEN84] Hennessy, J., “VLSI Processor Architecture”, IEEE Trans. on Computers, December 1984.

[KOO89] Koopman, J., Stack Computers: the new wave, Ellis Horwood, 1989.

[PAT82] Patterson, D., and Sequin, C., “A VLSI RISC”, Computer, Septermber, 1982.

[PAT85] Patterson, D., “Reduced Instruction Set Computers”, Communications of the ACM, January, 1985.

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

Exercises

6.1 Compile and run Sx-kit

6.2 Write three benchmark programs in nut: hanoi, matmul, (bubble, quick already existed in Sx-kit) run it under sx-simulator to test the correctness.

6.3 Modify the simulator to count the frequency of each instruction. Run the above benchmarks to find out the dynamic instruction count.

6.4 Write additional microprogram for the instruction “`inc`” and “`dec`”.

6.5 The combination of test-and-jump occurs frequently in the program. Write new instructions, such as, `jump-if-less-than`, `jump-if-equal` etc. Modify the code generator to output the new instructions. Measure their effectiveness.

6.6 Suggest some way to improve the speed of Sx processor.

6.7 A simple implementation of the processor simulator is to scan the microprogram ROM and execute the active event directly. The next micro-address is also converted into an integer. This makes the processor simulator 5 to 10 times slower than the implementation illustrated in the section laboratory session in this chapter. Do it and compare the speed.