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. S1 contains all the important elements of a real processor. It 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. 1024 16-bit words. It is a load/store architecture. It has 8 general purpose registers (R0..R7).

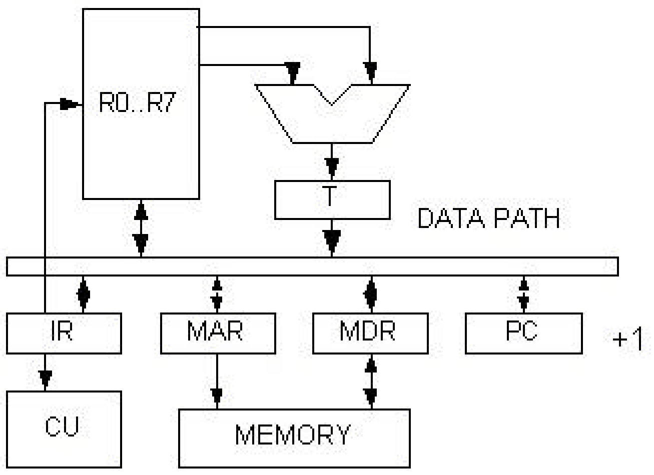

Figure 5.1 S1 microarchitecture

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". In S-format, the operands are registers, only 9 bits are used (op:3, r1:3, r2:3), 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`

```
| op:3 | r0:3 |  ads:10                  |
15..13 12..10    9..0                     bit position
```

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

```
| op:3 | xop:4 | r1:3 | r2:3 |  u:3  |
15..13  12..9   8..6   5..3    2..0   bit position
```

Instruction set

opcode	mnemonics	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 conditions: unconditional, equal, not equal, less than, less than or equal, greater than or equal, greater than, which 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 represent 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".

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).

TIMING of S1 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.

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.

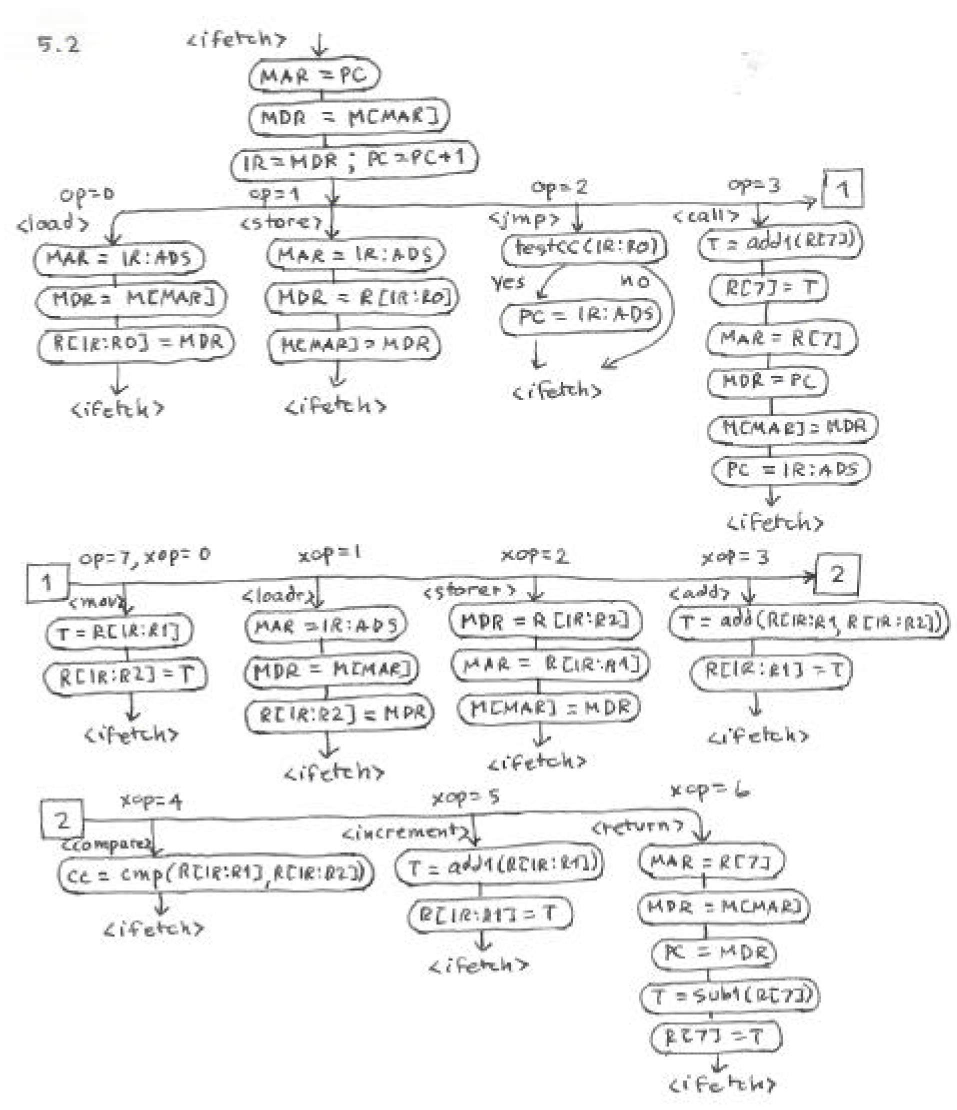

Figure 5.2 State diagram of S1 hardwired control unit

```
   <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. 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
```

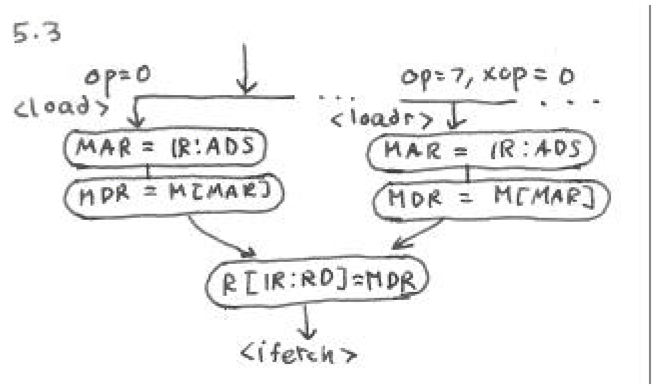

Figure 5.3 states of <load> and <loadr> after sharing

```
   <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.

Microprogram 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 }

Dest	Src	SelR	ALU	Mclt	Misc	Cond	Goto

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 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	

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
```

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

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.

Dest:5	Src:6	SelR:4	ALU:4	Mclt:2	Misc:1	Cond:4	Goto:5

a) unencoded microword (31 bits)

Dest:3	Src:3	SelR:3	ALU:3	Mclt:2	Misc:1	Cond:4	Goto:3

b) encoded microword (21 bits)

Figure 5.6 Comparing unencoded and encoded microword for S1

Calculating CPI

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 Turbo C v2.0. 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 xxxxxxxxxxxxxxxxxxx
....
```

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 0001001000000000000000000000000
1 0000000000000000000100010000001
2 0010000000100000000001000100000
3 0001000100000000000000000000000
4 0000000000000000000100010000100
5 1000000001010000000000100000000
6 0001000100000000000000000000000
....
27 0100000100000000000000100000000
28 0100010000000100000000100000000
```

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

<Web material>