

## 12.1 Microprogrammed Sequencers

Wednesday March 7, 2001

- Directly equivalent to finite state machines
- Consider the 'divide by 5' FSM
  - One input - One output - State is the remainder
  - Next State is
    - Previous State \* 2
    - + Input
    - Mod 5

Old	New			
State	Input	State	Output	Op
0	0	0	0	2*0+0=0
0	1	1	0	2*0+1=1
1	0	2	0	2*1+0=2
1	1	3	0	2*1+1=3
2	0	4	0	2*2+0=4
2	1	0	1	2*2+1=5+0
3	0	1	1	2*3+0=6+1
3	1	2	1	2*3+1=7+2
4	0	3	1	2*4+0=8+3
4	1	4	1	2*4+1=9+4

## 12.2 Implementation in VHDL

```

library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity by5 is port (
  x, clk : in std_logic;
  y       : out std_logic);
end by5;

architecture state_machine of by5 is
  type StateType is (state0, state1,
                    state2, state3, state4);
  signal p_s, n_s : StateType;
begin
  state_clocked:process(clk)
  begin
    if rising_edge(clk) then p_s <= n_s;
    end if;
  end process state_clocked;

  -- More architecture is on the next slide.

```

## 12.3 More Architecture of Divide by 5 FSM

```

fsm:process(p_s, x) - combinational
begin -- case
  case p_s is
    when state0 => y <= '0';
      if x = '1' then n_s <= state1;
      end if;
    when state1 => y <= '0';
      if (x = '1') then n_s <= state3;
      else n_s <= state2;
      end if;
    when state2 =>
      if (x = '1') then n_s <= state0;
      y <= '1';
      else n_s <= state4; y <= '0';
      end if;
    when state3 => y <= '1';
      if (x = '1') then n_s <= state2;
      else n_s <= state1;
      end if;
    when state4 => y <= '1';
      if (x = '0') then n_s <= state3;
      end if;
    when others => n_s <= state0;
  end case;
end process fsm;
end architecture state_machine;

```

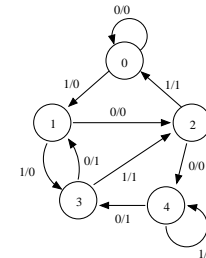
## 12.4 Implemented in Simple Code

Implement this in simple code.

```

S0:   if /x then {z=0: goto S0;} else {z=0; goto S1}
S1:   if /x then {z=0: goto S2;} else {z=0; goto S3}
S2:   if /x then {z=0: goto S4;} else {z=1; goto S0}
S3:   if /x then {z=1: goto S1;} else {z=1; goto S2}
S4:   if /x then {z=1: goto S3;} else {z=1; goto S4}

```



## 12.5 Single Instruction Machine

□ Fast: One cycle per instruction - "Horizontal"

○ Is this a microcoded machine or FSM?

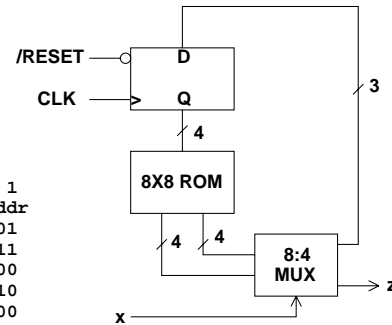
Single Instruction Format  
(for one condition)

addr 1	z1	addr0	z0
D7 D6 D5	D4	D3 D2 D1	D0
cond 1		cond 0	

Width =  $2^{(\text{Number of Conditions})}$

ROM Contents

Instr	x = 0	x = 1	z	adr	z	addr
000	0	000	0	001		
001	0	010	0	011		
010	0	100	1	000		
011	1	001	1	010		
100	1	011	1	100		



## 12.6 Another Single Instruction Machine

□ Fast: One cycle per instruction - "Horizontal"

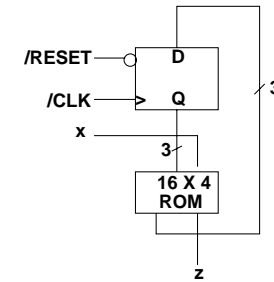
○ Is this a microcoded machine or FSM?

Single Instruction Format  
(for one condition)

cond	addr	z
0	D3 D2 D1	D0
1	D3 D2 D1	D0

Height =  $2^{(\text{Number of Conditions})}$

s	Instr	X	Address	z
			D3 D2 D1 D0	
s0	000	0	0 0 0 0	0
	000	1	0 0 1 0	0
s1	001	0	0 1 0 0	0
	001	1	0 1 1 0	0
s2	010	0	1 0 0 0	0
	010	1	0 0 0 1	1
s3	011	0	0 0 1 1	1
	011	1	0 1 0 1	1
s4	100	0	0 1 1 1	1
	100	1	1 0 0 1	1



## 12.7 Microprogrammed Sequencers

□ Single instruction - "Horizontal"

- Simple to implement
- Fast
- Wide word if many inputs (conditions)

```
if COND then jmp ADDR1 and assert X1
else jmp ADDR2 and assert X2
```

□ Single Instruction Machines

- Use a lot of memory (lots of ROM bits)
- Particularly if there are many conditions
- And for instructions that are independent of conditions

## 12.8 Two-Instruction Machine

□ Separates signal assertion from control - "Vertical"

- Reduces speed - Need two instructions if all instructions test a condition.
- Reduces memory usage (fewer ROM bits).
- Slight increase of sequencer hardware.

□ Instruction set for a simple, two-instruction machine

- Allows 16 addresses ( $2^4$ ).
- Allows 8 conditions ( $2^3$ ) - But one must be 'true'.
- Use a wider word to get more conditions and/or more addresses.

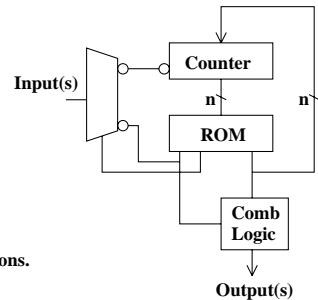
```
If COND then jmp ADDR      0 C C C A A A A
assert OUTPUT              1 S S S S S S S
```

## 12.9 Two-Instruction Machine

### Implementation

Use the mux to select the condition to test. One condition must be TRUE to implement an unconditional jump.

Output signals are a function of state. Outputs are asserted only when executing an "assert!" instruction. Use combinational logic to decode instructions. Beware, outputs are glitchy!

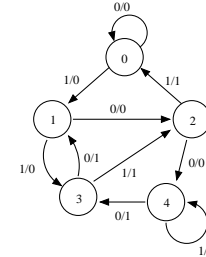


	17	16 15 14	13 12 11 10
if COND jmp ADDR	0	COND	ADDR
assert SIGNAL	1	SIGNAL	

## 12.10 Here is an Example

### Divide by 5 in 2-instruction machine:

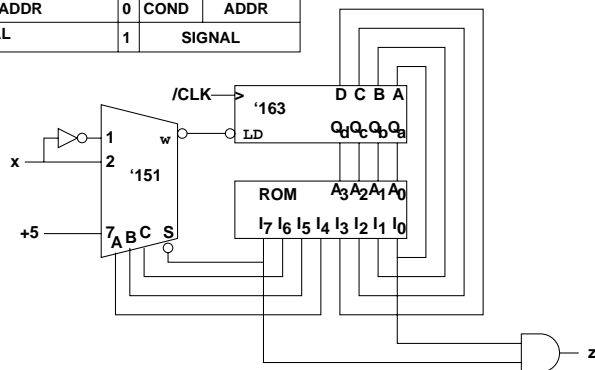
	Location	Contents	Condition
S0:	if /x goto S0;	0000 0 001 0000	001 /x
S1:	if x goto S3;	0001 0 010 0111	010 x
S2:	if /x goto S4;	0010 0 001 0101	001 /x
	assert z;	0011 1 --- ---1	
	if true goto S0;	0100 0 111 0000	111 true
S4:	assert z;	0101 1 --- ---1	
	if x goto S4;	0110 0 010 0101	010 x
S3:	assert z;	0111 1 --- ---1	
	if x goto S2;	1000 0 010 0010	010 x
	if true goto S1;	1001 0 111 0001	111 true



## 12.11 Hardware Implementation

### Two-instruction microsequencer implemented with SSI

	17	16 15 14	13 12 11 10
if COND jmp ADDR	0	COND	ADDR
assert SIGNAL	1	SIGNAL	



## 12.12 Programming PROMs

### PROMs (Programmable Read-Only Memory)

○ Include EPROM, EEPROM, Flash, etc.

### xxx.dat format is a number separated by white space.

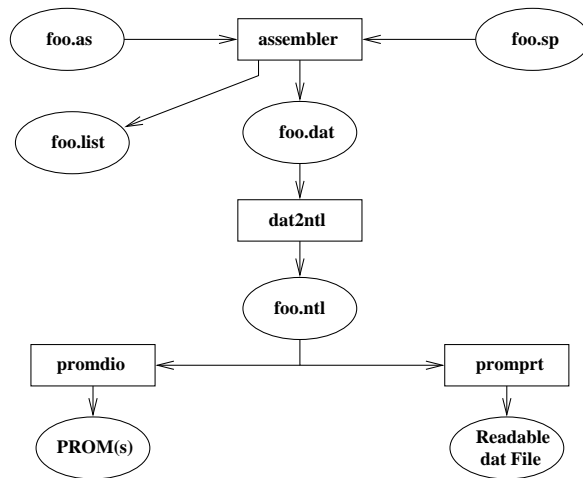
○ Try 'man dat2nt!' (after setup 6.111, of course).

○ Command statements can specify the base and starting address in the PROM.

○ They include

```
# SET_ADDRESS = integer;      Interpreted by assembler as well.
# LOAD_ADDRESS = integer;     Only interpreted by dat2nt1.
# RIGHT_SHIFT = integer;
# MASK_COUNT = integer;       Number of bits to output.
# PACK_BYTES = integer;
# BASE = 'HEX' | 'DECIMAL' | 'BINARY';
```

## 12.13 Programming PROMs



## 12.14 assembler

- Takes input from
  - assembly file has instructions (foo.as)
  - specification file defines instructions (foo.sp)
- Output is a series of numbers (foo.dat).
  - Destined to be the contents of the ROM.
- dat2ntl produces ntl hex format for programming ROM.
  - Splits words into bytes.
  - Puts in format for programmers.
- Scripts have been set up to automate operation.
  - Here is assem16to8.

```

assembler <$1.as >/tmp/$$$.dat
dat2ntl </tmp/$$$.dat | tr [a-z] [A-Z] >byt0$1.ntl
dat2ntl 8 </tmp/$$$.dat | tr [a-z] [A-Z] >byt1$1.ntl
rm /tmp/$$$.dat
  
```

- Assumes 16-bit wide numbers as input.
- Generates two 8-bit wide output files.

## 12.15 Specification File

- Extract of /mit/6.111/prom/examples/encr.sp

```

op<7:0>;          /*defines instruction length and location*/
address op<4:0>;  /*defines bit field the address*/
value op<4:0>;    /*allows integer values in ASSEM_FILE*/
if op<7> = 0;     /*used in making conditional jumps*/
do op<7:6> = 3;   /*this is used for making assertions*/
/*definitions of conditional signals*/
go op<6:5> = 0;
encrypt op<6:5> = 1;
a_b op<6:5> = 2;
test op<6:5> = 3;
uncond op<7:5> = 4; /*this is for an unconditional jump*/
/*definitions of assertions*/
assert_low op<5> op<4>; /*makes default for field <5:4> high*/
select_b op<5> = 0;
select_a op<4> = 0; /*Note that they are asserted low*/
load op<3> = 1;
select_157 op<2:0> = %b001;
incmar op<2:0> = %b010;
clr op<2:0> = %b011;
shift op<2:0> = %b110;
bit0 op<0> = %b1;
select_257 op<2> = 1 op<1> = %b1 bit0;
  
```

## 12.16 Assembly File (Instructions)

- Extract of /mit/6.111/prom/examples/encr.as

```

# SPEC_FILE = encr.sp; /*gives the specification filename*/
# LIST_FILE = encr.list; /*specifies the list filename*/
# SET_ADDRESS = 0;     /*start assembling here*/
start      : if go start; /*The semicolon delimits one
                instruction from the next semicolon. Everything between
                semicolons is put in the same instruction */
rdata      : do load clr select_a select_b;
encheck1   : if encrypt memcheck;
aplyalg    : do load select_b shift;
switch     : do load select_a select_b select_257;
              do load select_b;
donecheck  : if encrypt ready1;
memcheck   : if a_b multrep;
              do incmar;
lastloc    : if test memcheck;
              if encrypt aplyalg;
              do 14; /*Note the use of integer values. a value*/
              do 11; /*field was specified in the spec file */
ready1     : do ready;
              uncond start;
multrep    : if encrypt a_multrep;
              uncond ready1;
  
```

## 12.17 List File

□ Extract of /mit/6.111/prom/examples/encr.list

```
ADR DAT CODE
# SPEC_FILE = encr.sp; /*specification filename*/
# LIST_FILE = encr.list; /*list filename*/
# SET_ADDRESS = 0; /*where to start*/
0 0 start : if go start;
1 cb rdata : do load clr select_a select_b;
2 2e encheck1 : if encrypt memcheck;
3 de applyalg : do load select_b shift;
b cf switch : do load select_a select_b select_257;
c d8 : do load select_b;
d 34 donecheck : if encrypt ready1;
e 56 memcheck : if a_b multrep;
f f2 : do incmar;
10 6e lastloc : if test memcheck;
11 23 : if encrypt applyalg;
12 ee : do 14; /*Note the use of integer values.*/
13 eb : do 11; /*A value field was specified.*/
14 f4 ready1 : do ready;
15 80 : uncond start;
16 39 multrep : if encrypt a_multrep;
18 94 : uncond ready1;
```

## 12.18 Numbers for the PROM

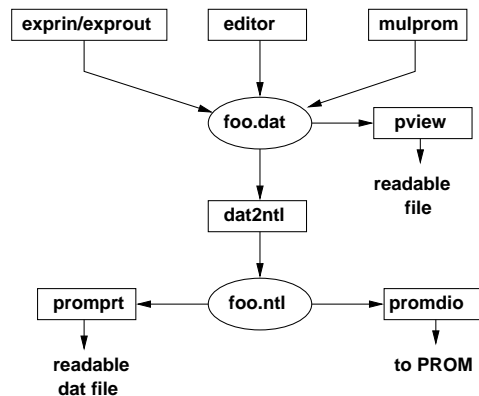
□ assem output

```
:0800000000CB2EDEEEF6EEF659
:08000800EEF6EECFD83456F2FB
:080010006E23EEEBF48039FDD4
:0300180094F983D5
:00000001FF
```

□ To make that more readable:

```
00 cb 2e de ee f6 ee f6
ee f6 ee cf d8 34 56 f2
6e 23 ee eb f4 80 39 fd
94 f9 83
```

## 12.19 Other Sources of Data



## 12.20 Functions

□ To generate a lookup table:

```
generator.mit.edu% exprin > log.in
```

```
If you need help at any time type ?
If you want to redo the last command type redo or r
If you want to quit before completion type q
```

```
enter expression : 64*log(input)
enter number of steps : 16
enter starting address in HEX: 0
enter starting value for INPUT: 2
enter step increment : 2
```

## 12.21 Output Files

```
generator.mit.edu% exprout < log.in > log.dat
generator.mit.edu% cat log.dat
2c
59
73
85
93
9f
a9
b1
b9
c0
c6
cb
d1
d5
da
de
generator.mit.edu% dat2ntl < log.dat > log.ntl
generator.mit.edu% cat log.ntl
:080000002C597385939FA9B1EF
:08000800B9C0C6CBD1D5DADE88
:00000001FF
```

## 12.22 Special Characters

□ This file was built with a text editor.

```
generator.mit.edu% cat arrows.dat
00000000
00000000
00001000
00011000
00110000
01111110
01111110
00110000
00011000
00001000
00000000
00000000
00000000
00010000
00011000
00001100
01111110
01111110
00001100
00011000
00010000
00000000
```

## 12.23 To See Special Characters

□ The ones can be used to light up pixels.

```
pview < arrows.dat
```

```
 1
 11
 11
111111
111111
 11
 11
 1
```

```
 1
 11
 11
111111
111111
 11
 11
 1
```

## 12.24 Mulprom

Mulprom is a program to make up multiplication tables:

```
mulprom -n 8 -b -c 8 -a 6 -i 3 -s 3 -t > exb.dat
```

Control arguments to mulprom are:

- n # prom word width
- b radix is binary
- c # number of columns to use for output
- a # number of address bits
- i # number of multiplier bits
- s # number of multiplicand bits
- t number is two's complement

Here is the data file produced by the invocation above

```
#BASE = BINARY;
#SET_ADDRESS = 0000;
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000001 00000010 00000011 11111100 11111101 11111110 11111111
00000000 00000010 00000100 00000110 11111000 11111010 11111100 11111110
00000000 00000011 00000110 00001001 11110100 11110111 11111010 11111101
00000000 11111100 11111000 11110100 00010000 00001100 00001000 00000100
00000000 11111101 11111010 11110111 00001100 00001001 00000110 00000011
00000000 11111110 11111100 11111010 00001000 00000110 00000100 00000010
00000000 11111111 11111110 11111101 00000100 00000011 00000010 00000001
```

## 12.25 Mulprom Output

```
#BASE = BINARY;
#SET_ADDRESS = 0000;
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000001 00000010 00000011 11111100 11111101 11111110 11111111
00000000 00000010 00000100 00000110 11111000 11111010 11111100 11111110
00000000 00000011 00000110 00001001 11110100 11110111 11111010 11111101
00000000 11111100 11111000 11110100 00010000 00001100 00001000 00000100
00000000 11111101 11111010 11110111 00001100 00001001 00000110 00000011
00000000 11111110 11111100 11111010 00001000 00000110 00000100 00000010
00000000 11111111 11111110 11111101 00001000 00000011 00000010 00000001
```

**Here is the .hex file**

```
:0800000000000000000000000000F8
:0800080000010203FCFDFEFFF4
:0800100000020406F8FAFCFEF0
:0800180000030609F4F7FAFDEC
:0800200000FCF8F4100C0804C8
:0800280000FDFAF70C090603C4
:0800300000FEFCFA08060402C0
:0800380000FFFEFD04030201BC
:00000001FF
```

**This is the file produced  
by promprt**

```
#SET_ADDRESS = 00000000;
00 00 00 00 00 00 00 00
00 01 02 03 fc fd fe ff
00 02 04 06 f8 fa fc fe
00 03 06 09 f4 f7 fa fd
00 fc f8 f4 10 0c 08 04
00 fd fa f7 0c 09 06 03
00 fe fc fa 08 06 04 02
00 ff fe fd 04 03 02 01
```