

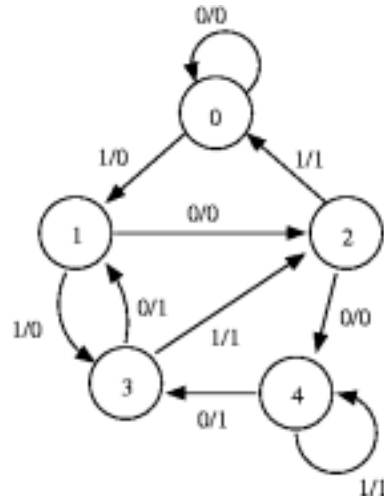
L12.1 Microprogrammed Sequencers Wed. March 6, 2002

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



L12.2 Divide By 5 in VHDL Wed. March 6, 2002

We already know how to express this in VHDL
 (actually, there are a number of ways).
 Here is a way with state variables assigned.

```
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
    signal p_s, n_s : std_logic_vector(2 downto 0);
    constant state0 : std_logic_vector(2 downto 0) := "000";
    constant state1 : std_logic_vector(2 downto 0) := "001";
    constant state2 : std_logic_vector(2 downto 0) := "010";
    constant state3 : std_logic_vector(2 downto 0) := "011";
    constant state4 : std_logic_vector(2 downto 0) := "100";
```

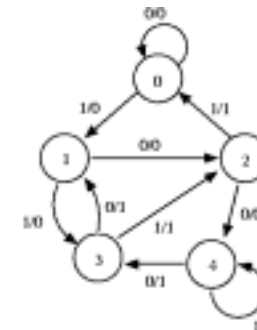
L12.3 VHDL Architecture Wed. March 6, 2002

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

L12.4 Wed. March 6, 2002

Change the syntax a little but don't change the meaning.

```
state0: if /x then {z=0: goto S0;} else {z=0: goto S1}
state1: if /x then {z=0: goto S2;} else {z=0: goto S3}
state2: if /x then {z=0: goto S4;} else {z=1: goto S0}
state3: if /x then {z=1: goto S1;} else {z=1: goto S2}
state4: if /x then {z=1: goto S3;} else {z=1: goto S4}
```



L12.5 Single Instruction Machine Wed. March 6, 2002

Or is this an FSM? Could it be ANY FSM you want it to be?

How about any FSM with up to 8 states, one input, and one output?

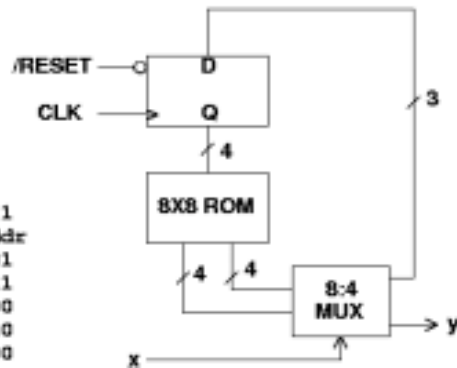
Single Instruction Format
 (for one condition)

addr 1	y1	addr0	y0
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
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



L12.6 Optimizing a Bit Wed. March 6, 2002

This is still fast. One "instruction" per clock cycle.

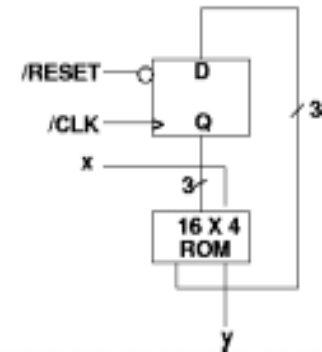
Each pair of PROM words represents a single "instruction".

Single Instruction Format
 (for one condition)

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

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

Instr	X	Address	y
		D3 D2 D1 D0	
S0	000	0 0 0 0	0
	000	1 0 0 1	0
S1	001	0 0 1 0	0
	001	1 0 1 1	0
S2	010	0 1 0 0	0
	010	1 0 0 0	1
S3	011	0 0 0 1	1
	011	1 0 1 0	1
S4	100	0 0 1 1	1
	100	1 1 0 0	1



L12.7 Two-Instruction Machine Wed. March 6, 2002

Separates signal assertion from control

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'.

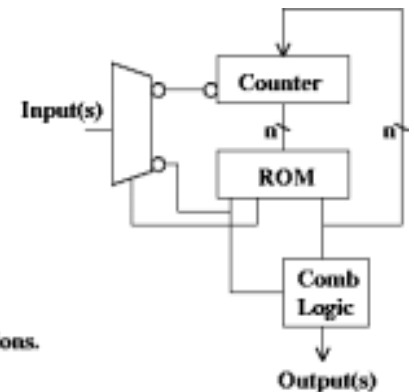
Use a wider word to get more conditions and/or more addresses.

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

L12.8 Block Diagram Wed. March 6, 2002

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

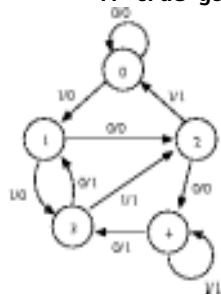
Output signals are a function of state.
 Outputs are asserted only when executing an "assert" instruction.
 Use combinational logic to decode instructions.
 Beware, outputs are glitchy!



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

L12.9 An Example Program Wed. March 6, 2002

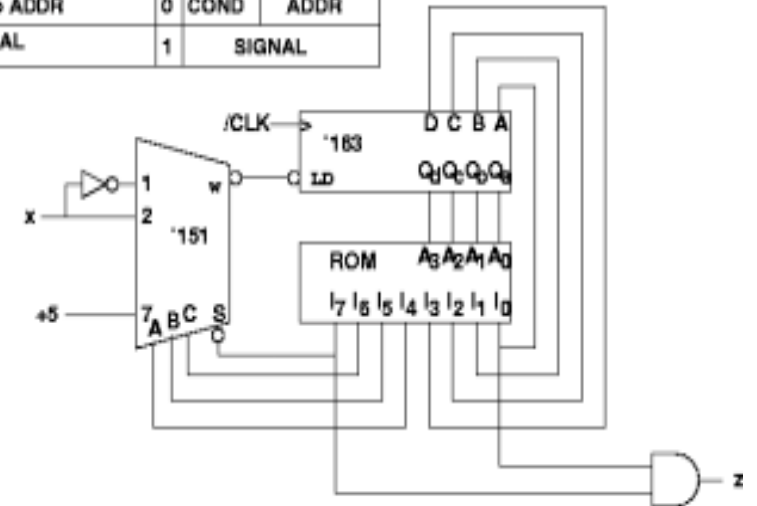
		Location	Contents	Condition
state0:	if /x goto S0;	0000	0 001 0000	001 /x
state1:	if x goto S3;	0001	0 010 0111	010 x
state2:	if /x goto S4;	0010	0 001 0101	001 /x
	assert z;	0011	1 --- ---1	
	if true goto S0;	0100	0 111 0000	111 true
state4:	assert z;	0101	1 --- ---1	
	if x goto S4;	0110	0 010 0101	010 x
state3:	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



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

L12.10 MSI Implementation Wed. March 6, 2002

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



L12.11 Programming PROMs Wed. March 6, 2002

PROMs (Programmable Read-Only Memory)

Include EPROM, EEPROM, Flash, etc.

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

Try 'man dat2ntl' (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 dat2ntl.
- # RIGHT_SHIFT = integer;
- # MASK_COUNT = integer; Number of bits to output.
- # PACK_BYTES = integer;
- # BASE = 'HEX' | 'DECIMAL' | 'BINARY';

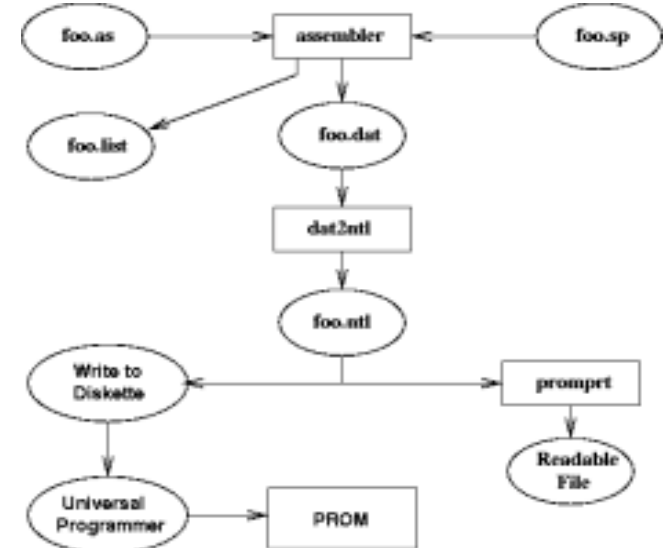
L12.12 Programming PROMs Wed. March 6, 2002

Here is the process of putting an assembly program into a ROM.

The assembler puts together the pieces of the program. The result is a list of numbers.

The formatter turns that list into the file used to program the ROM.

Right now, the intel hex file (foo.ntl) is written to a floppy and transferred to the programmer where iot is used to program the ROM.



L12.13 assembler Wed. March 6, 2002

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.

L12.14 Specification File Wed. March 6, 2002

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;
```

L12.15 Assembly File (Instructions) Wed. March 6, 2002

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;
applyalg : 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 applyalg;
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;
```

L12.16 Wed. March 6, 2002

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;
```

L12.17 Numbers for the PROM Wed. March 6, 2002

The assembler reads xxx.as to find out the name of the xxx.sp file.
 It then reads the xxx.sp file to define all the symbols that will be used in the xxx.as file.
 It makes two passes through the xxx.as file.
 It evaluates everything that it can (all except forward label references) on the first pass. and produces the output numbers .
 It then takes a second pass through the xxx.as file to resolve the forward references.
 For more details, type 'man assembler' (after setup 6.111, of course).

The output of assem is in intel hex format.

```
:0800000000CB2EDEF6EEF659
:08000800EEF6EECFD83456F2FB
:080010006E23EEBF48039FDD4
:0300180094F983D5
:00000001FF
```

Use promprt to make that more readable.

```
#SET_ADDRESS = 00000000;
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
```

L12.18 Other Sources of Data Wed. March 6, 2002

PROM contents can come from the assembler, from a text editor, or from other sources. There are a variety of tools to help with this.



L12.19 Functions Wed. March 6, 2002

Use exprin to generate input for exprout and exprout to generate xxx.dat.

```
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
```

```
generator.mit.edu% exprout < log.in > log.dat
generator.mit.edu% cat log.dat
# set_address = 0 ;
```

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

L12.20 Special Characters Wed. March 6, 2002

arrows.dat was built with a text editor.

Use pview to "display" spaces instead of zeros.

```
generator.mit.edu% cat arrows.dat      pview < arrows.dat
#SET_ADDRESS=0;                        #SET_ADDRESS= ;
#BASE=BINARY;                          #BASE=BINARY;
00001000                                1
00011000                                11
00110000                                11
01111110                                111111
01111110                                111111
00110000                                11
00011000                                11
00001000                                1
00000000
00000000
00000000
00010000                                1
00011000                                11
00001100                                11
01111110                                111111
01111110                                111111
00001100                                11
00011000                                11
00010000                                1
00010000                                1
```

L12.21 mulprom Wed. March 6, 2002

Type 'man mulprom' to see all the details.

mulprom is a program to make up multiplication tables:

```
mulprom -n 8 -b -c 8 -s 6 -i 3 -a 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
```