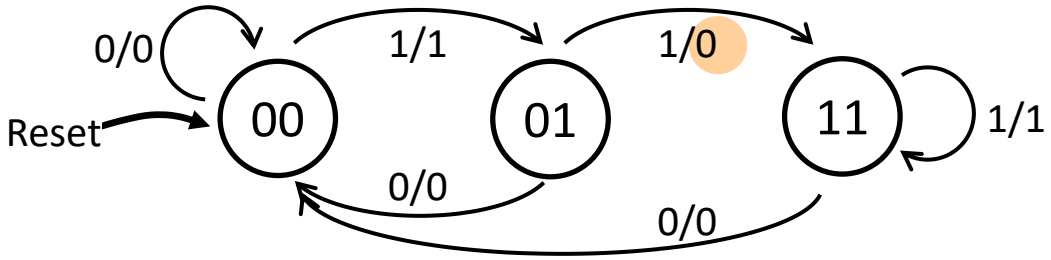
# FSM Example: String Manipulator

- Takes in a stream of inputs and removes the *second* 1 from every consecutive string of 1's.




input: 0 1 1 0 0 1 1 1 0 0 1

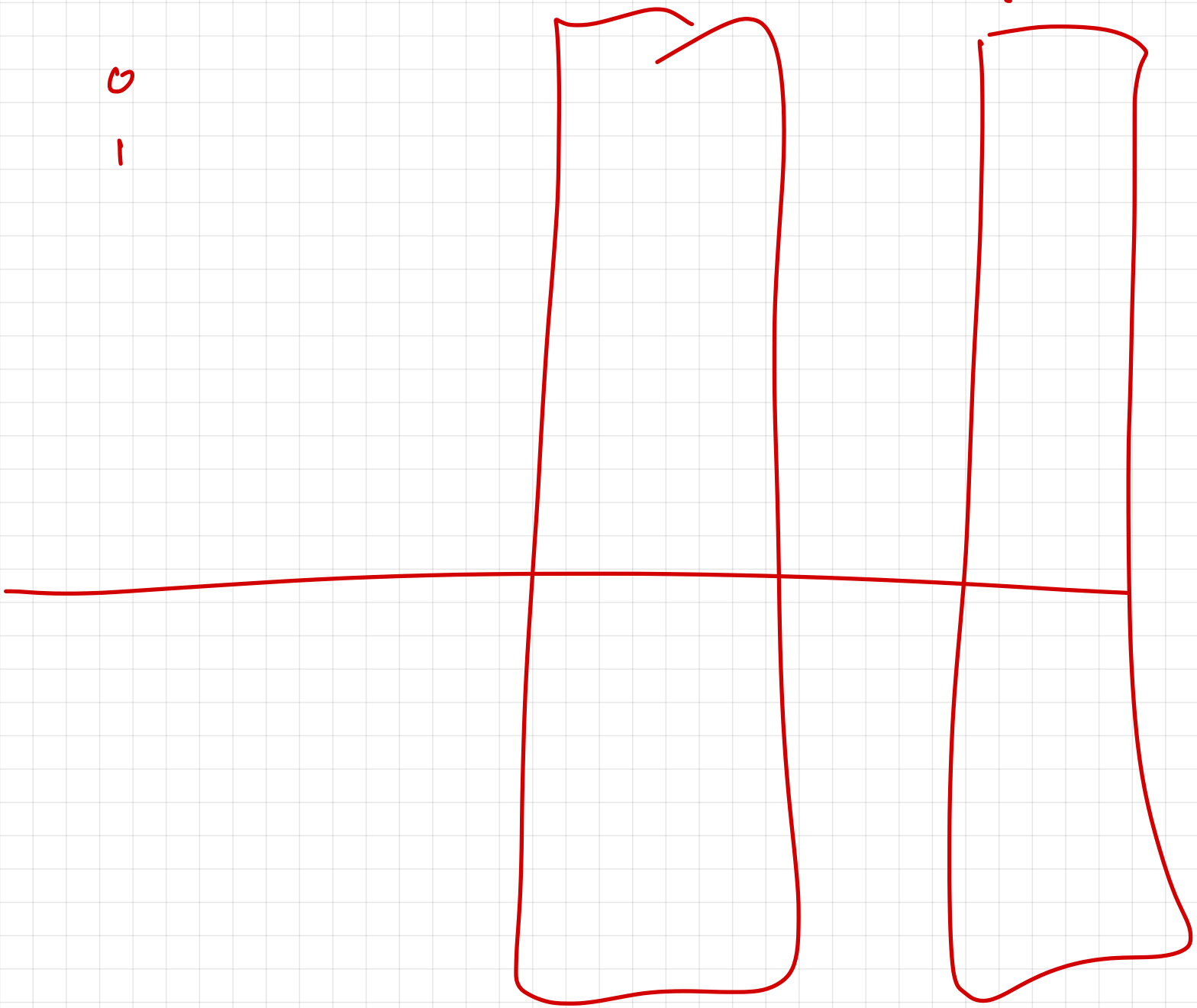
out: 0 1 0 0 0 1 0 1 1 0 0 1

# String Manipulator FSM

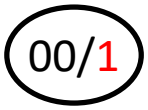
```
module fsm (input  logic clk, reset, in,
            output logic out);

   // present and next state
   enum {S0, S1, S3} ps, ns;

   // next state logic
   always_comb
      case (ps)
          S0: if (in) ns = S1;
              else    ns = S0;
          S1: if (in) ns = S3;
              else    ns = S0;
          S3: if (in) ns = S3;
              else    ns = S0;
      endcase

   // output logic
   assign out = in & (ps[1] | ~ps[0]);

   ...
```

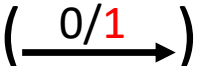
```
   ...

   // sequential logic (DFFs)
   // synchronous reset
   always_ff @(posedge clk)
      if (reset)
         ps <= S0;  // reset state
      else
         ps <= ns;

endmodule  // fsm
```

| PS | input | ns | output |
|---|---|---|---|
| 00 | 0 | | |
| | 1 | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |



# Moore vs. Mealy

- **Moore** machines define their outputs based on states ( 

) and **Mealy** machines define outputs based on transitions ( 

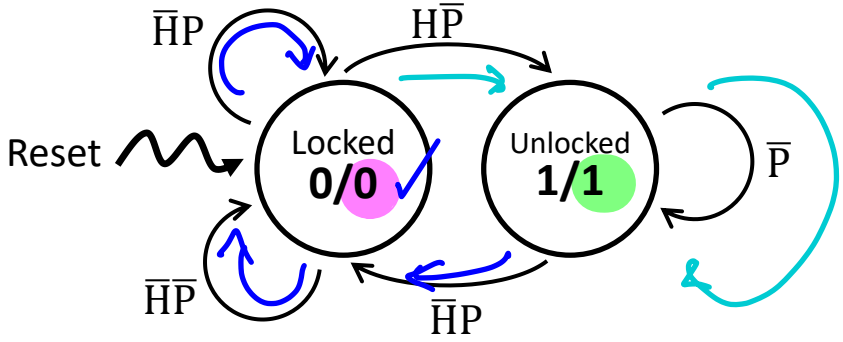
)
  - Mealy machines are more *flexible*
    - Moore outputs are function of state; Mealy outputs are function of state *and inputs*
  - All FSMs can be expressed in either form, but some systems are more naturally expressed one way versus the other
    - Feel free to use either in this class if not specified
    - However, there *are* implementation differences!

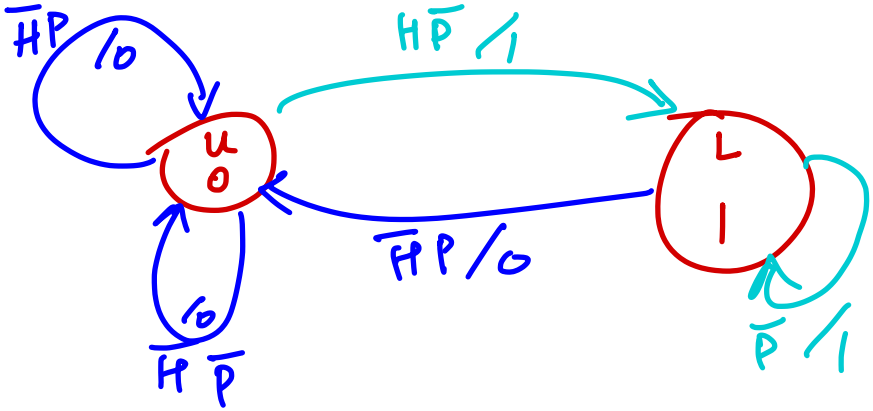
# Mealy ↔ Moore Conversions

Not testable material

- **Moore → Mealy:** copy the state output to every transition *entering* the state

- Example: FSM for a *turnstile*, which is locked until someone swipes their Husky ID (input H) and then locks once you push through (input P) the unlocked gate. Outputs a light that glows red (0) or green (1).



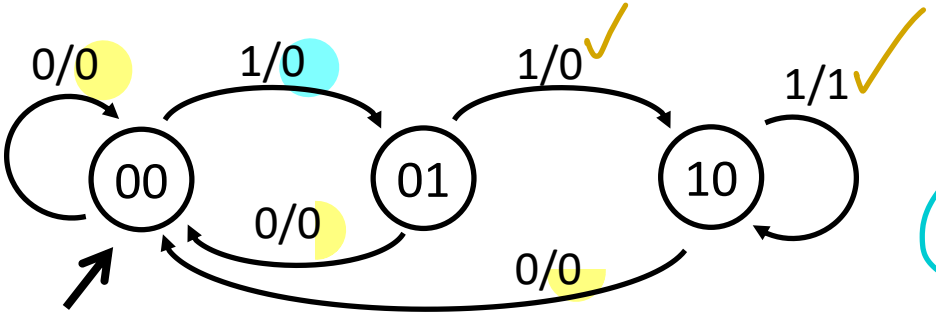



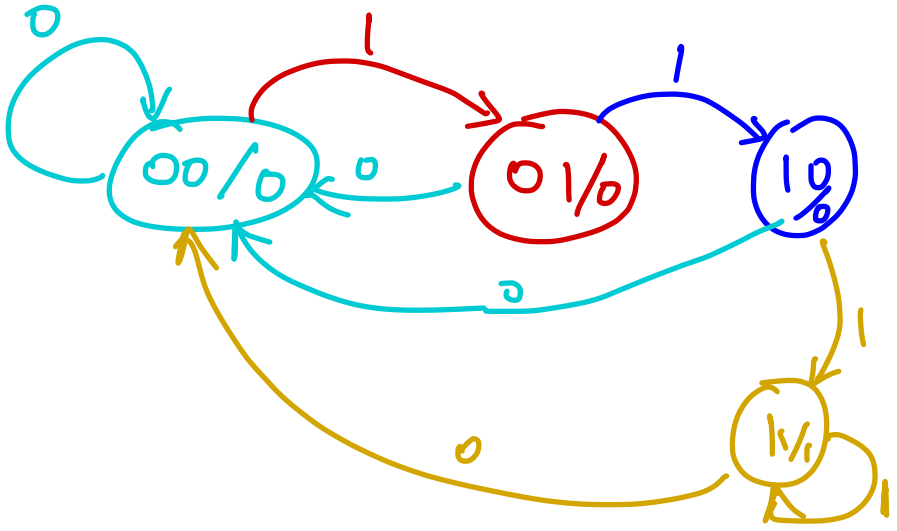
# Mealy ↔ Moore Conversions

Not testable material

- **Mealy → Moore**: more complicated process; if incoming transitions differ in output, may need to "split" the state

- Example: the `threeOnes` FSM from Lecture 1

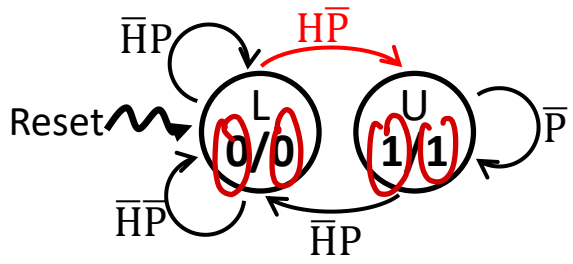


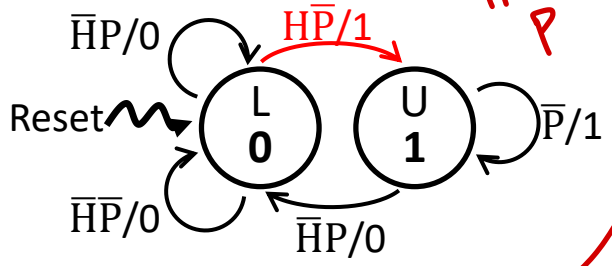
# Moore vs. Mealy Outputs

- Compare a Moore and Mealy FSM for the turnstile. Complete the statements and waveform below, assuming no delays:

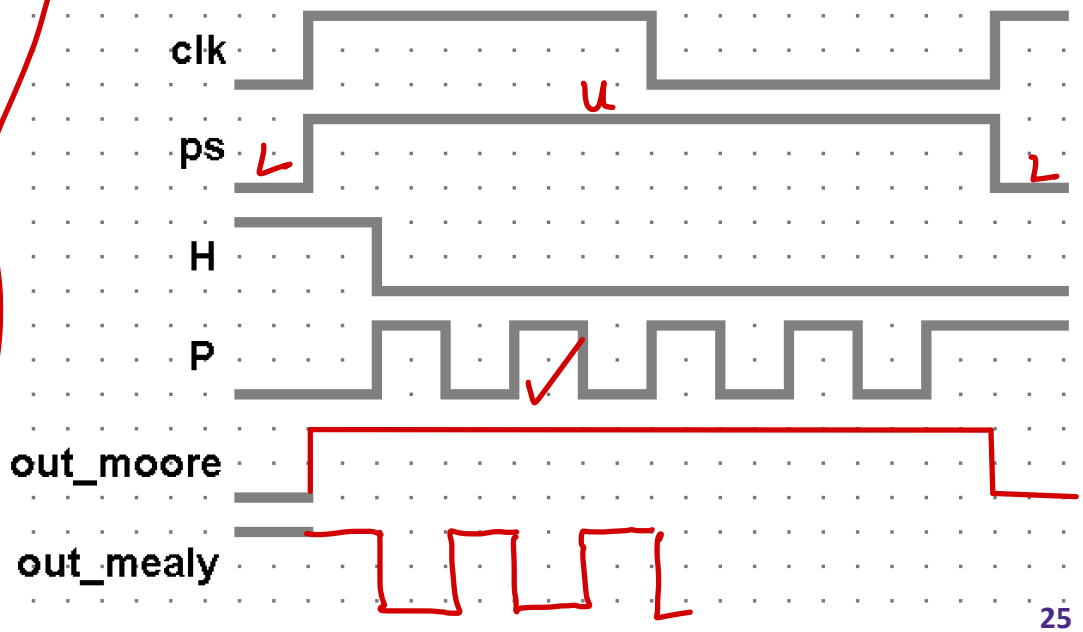
**Moore:**




**Mealy:**



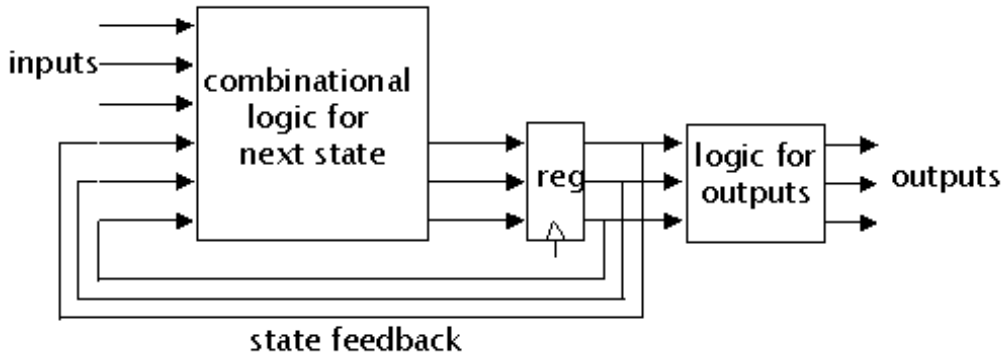

```
assign out_moore = (ps==U);
assign out_mealy = H&~P&(ps==L)
                 | (ps==U)&~P;
```




$H \cdot \bar{P} \cdot \overline{ps} + ps \cdot \bar{P}$

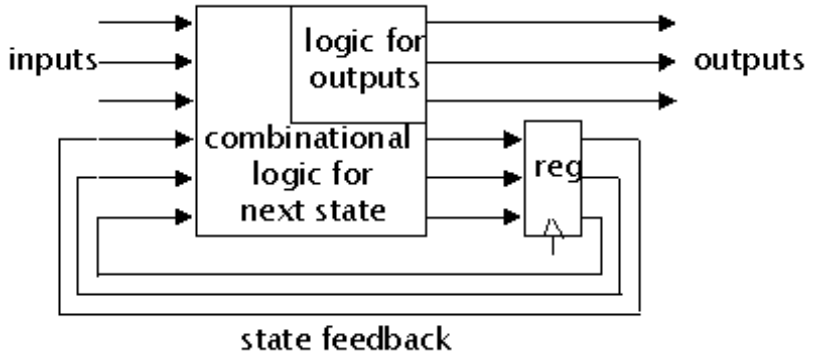
# Moore vs. Mealy Outputs

- Moore:




  - Outputs change synchronously with state changes

- Mealy:




  - Input changes can cause *immediate* output changes

# Lecture Outline

- SystemVerilog Review & Tips (Cont.)
- Finite State Machine Design
- **Testbenches**

# Testbenches

- Special modules *needed for simulation only!*
  - Software constraint to mimic hardware
- ModelSim runs entirely on your computer
  - Tries to simulate your FPGA environment without actually using hardware – no physical signals available
  - Must create fake inputs for FPGA's physical connections
    - *e.g.*, `LEDR`, `HEX`, `KEY`, `SW`, `CLOCK_50`
  - Unnecessary when code is loaded onto FPGA
- Need to define both input signal combinations as well as their *timing*

# Testbench Timing Controls

#(period/2);

#10

- Delay: #<time>
  - Delays by a specific amount of simulation time
- Edge-sensitive: @(<pos/neg>edge <signal>)
  - Delays next statement until specified transition on signal
- Level-sensitive Event: wait(<expression>)
  - Waits until <expression> evaluates to TRUE
- Stop simulation: $stop;

$finish.

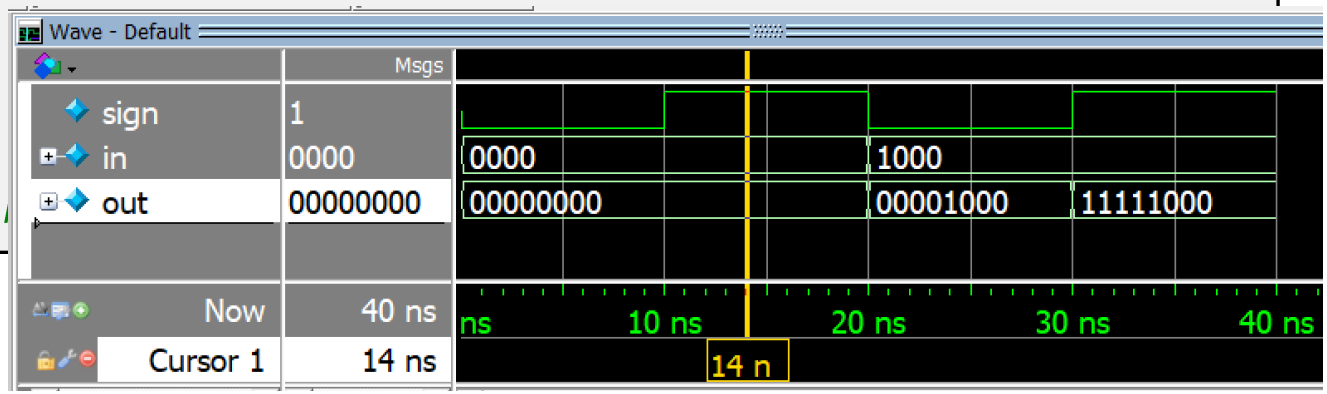
- Timescale: `timescale <time unit> / <precision>
  - *e.g.*, `timescale 1 ns / 1 ps

# Extender Testbench

Zero sign

MSB 0/1

```
`timescale 1 ns / 1 ns
module extender_tb();

   parameter M = 4, N = 8;
   logic [M-1:0] in;
   logic [N-1:0] out;
   logic sign;

   extender #(M, N) dut (.*);

   int i;
   initial begin
      for (i = 0; i < 2**2; i++) begin
         sign = i[0]; in = {i[1], {(M-1){1'b0}}}; #10;
      end  // for
      $stop;
   end  // initial

endmodule  // exte
```



# FSM Testbench Notes

- Your main goal is to test *every transition* that we care about – may take extra clock cycles
- For simulation, you need to generate a clock signal
  - Assume we have `parameter clock_period;`

**Explicit Edges:**

```
initial
    clk = 0;

always_comb begin
    #(clock_period/2) clk <= 1;
    #(clock_period/2) clk <= 0;
end
```

**Toggle:**

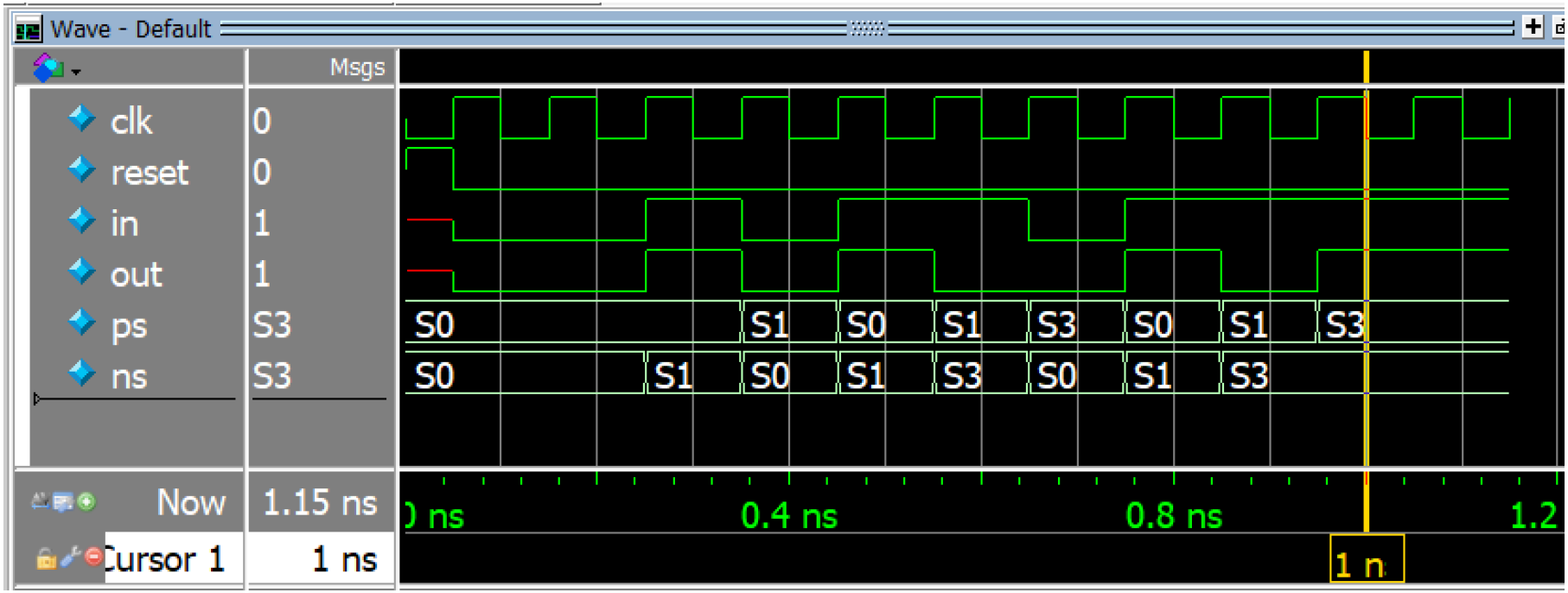
```
initial begin
    clk <= 0;
    forever #(clock_period/2) clk <= ~clk;
end
```

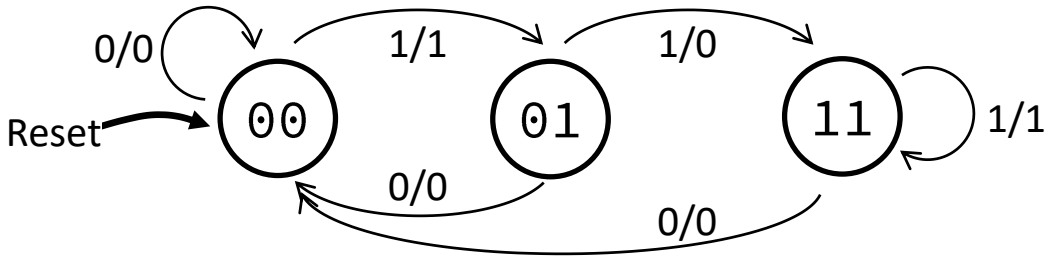
# String Manipulator Testbench

```
module fsm_tb();

   logic clk, reset, in, out;

   fsm dut (.*);

   // simulated clock
   parameter period = 100;
   initial begin
      clk <= 0;
      forever
         #(period/2)
         clk <= ~clk;
   end  // initial clock

   ...
```

```
   ...

   initial begin
      reset <= 1;              @(posedge clk);
      reset <= 0; in <= 0;     @(posedge clk);
                  in <= 0;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                  in <= 0;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                  in <= 0;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                  in <= 1;     @(posedge clk);
                               @(posedge clk);
      $stop;  // end simulation
   end  // initial signals

endmodule  // fsm_tb
```

# String Manipulator Waveforms



# Checking Responses

- Visually checking simulated waveforms quickly becomes impractical for large designs simulated over thousands of clock cycles
  - Displaying and explaining your waveforms for labs can be tedious

- There are simulator-independent system tasks to write messages to the user/tester!
  - Look similar to `printf()` in C
    - `$<system_task>(<format_string>, <sig_1>, <sig_2>, …)`
  - Will look at `$display` today and others later on

# Checking Responses: `$display`

- Triggers once when encountered, prints the given format string and adds a new line:

```
// define test inputs
int i;
initial begin
   for (i = 0; i < 2**2; i++) begin
      sign = i[0]; in = {i[1], {(M-1){1'b0}}}; #10;
      $display("t = %0t, %b %s %b",
               $time, in, sign ? "-+->" : "-0->", out);
   end  // for
   $stop;
end  // initial
```

Transcript

```
# t = 10, 0000 -0-> 00000000
# t = 20, 0000 -+-> 00000000
# t = 30, 1000 -0-> 00001000
# t = 40, 1000 -+-> 11111000
```

# Format Specifiers

Table 5.7: Format Specifiers.

| Specifier | Meaning |
|---|---|
| %h | Hexadecimal format |
| %d | Decimal format |
| %o | Octal format |
| %b | Binary format |
| %c | ASCII character format |
| %v | Net signalstrength |
| %m | Hierarchical name of current scope |
| %s | String |
| %t | Time |
| %e | Real in exponential format |
| %f | Real in decimal format |
| %g | Real in exponential or decimal format |

Table 5.8: Special characters.

| Symbol | Meaning |
|---|---|
| \n | New line |
| \t | Tab |
| \\ | \character |
| \" | " character |
| \xyz | Where xyz is are octal digits<br>- the character given by that octal code |
| %% | % character |

- Warning: these differ from the specifiers for `printf`
- The *minimum* field width is specified by numbers between the '%' and specifier letter
  - *e.g.*, `%3d` will pad out to 3 digits if necessary,
    `%0d` will show just the minimum number of digits needed

# EE/CSE 371
# Design of Digital Circuits and Systems

## Lecture 3: Memory I

# Relevant Course Information

1. HW 1 and Lab 1 due this week

2. Demos are in progress

3. ~~2~~. HW 2 and Lab 2 will be posted on Wednesday October 4th

4. Online lectures next week.

# Review Question

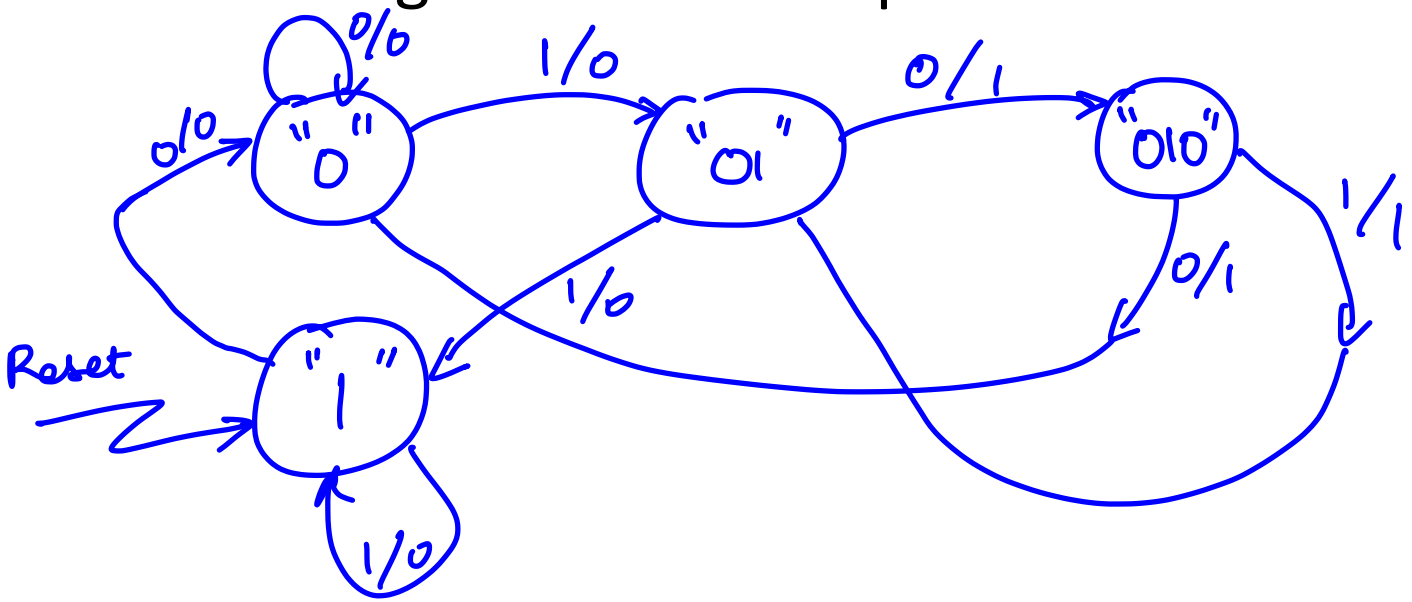
- Design a FSM that will output two consecutive 1's any time it sees the string "010" and outputs 0 otherwise
  - 1 input bit and 1 output bit
  - How many state bits do we need?
  - Which state should be your initial/reset state?

- If you have time, design both a **Moore** machine and **Mealy** machine "from scratch" (*i.e.*, don't convert between the two)
  - Which seems easier to implement? Can you name specific ways that the SystemVerilog implementation will be "easier"?

# Review Solution

- Design a FSM that will output two consecutive 1's any time it sees the string "010" and outputs 0 otherwise

Mealy




Moore

Do after Class

# Aside: Powers of 2 and Prefixes

- Here focusing on large numbers (*i.e.*, exponents > 0)
- SI prefixes are *ambiguous* if base 10 or 2
  - Note that $10^3 \approx 2^{10}$
- IEC prefixes are *unambiguously* base 2

**SIZE PREFIXES ($10^x$ for Disk, Communication; $2^x$ for Memory)**

| SI Size | Prefix | Symbol | IEC Size | Prefix | Symbol |
|---|---|---|---|---|---|
| $10^3$ | Kilo- | K | $2^{10}$ | Kibi- | Ki |
| $10^6$ | Mega- | M | $2^{20}$ | Mebi- | Mi |
| $10^9$ | Giga- | G | $2^{30}$ | Gibi- | Gi |
| $10^{12}$ | Tera- | T | $2^{40}$ | Tebi- | Ti |
| $10^{15}$ | Peta- | P | $2^{50}$ | Pebi- | Pi |
| $10^{18}$ | Exa- | E | $2^{60}$ | Exbi- | Ei |
| $10^{21}$ | Zetta- | Z | $2^{70}$ | Zebi- | Zi |
| $10^{24}$ | Yotta- | Y | $2^{80}$ | Yobi- | Yi |

| |
|---|
| $2^0 = 1$ |
| $2^1 = 2$ |
| $2^2 = 4$ |
| $2^3 = 8$ |
| $2^4 = 16$ |
| $2^5 = 32$ |
| $2^6 = 64$ |
| $2^7 = 128$ |
| $2^8 = 256$ |
| $2^9 = 512$ |
| $2^{10} = 1024$ |

# Memory

- Several forms of memory are available, which include:
  - ~~Secondary memory (*e.g.*, hard disk, flash drive)~~
  - Read-only memory (ROM)
  - Random-access memory (RAM)
  - Register files
    - Small, fast, fixed-sized memory that hold CPU data state
  - First in, first out (FIFO) buffers

# Embedded FPGA Memory

- An FPGA contains prefabricated memory modules
  - Intended for small or intermediate-sized storage
  - Contents of memory blocks can be configured via memory initialization files (`.mif`)
- The DE1-SoC's Cyclone V FPGA (Cyclone V SE A5) has:
  - 31k Adaptive Logic Modules (ALMs)
  - 4.45 Mbits of memory organized as 397 memory blocks, each with 10 kbits of storage (M10K)
    - Flexible, configurable memory storage available to the designer
    - Each M10K can act as single-port memory, dual-port RAM, shift register, ROM, or a FIFO buffer
  - More info in "*Cyclone V Device Handbook Vol 1.pdf*"

# DE1-SoC Memory

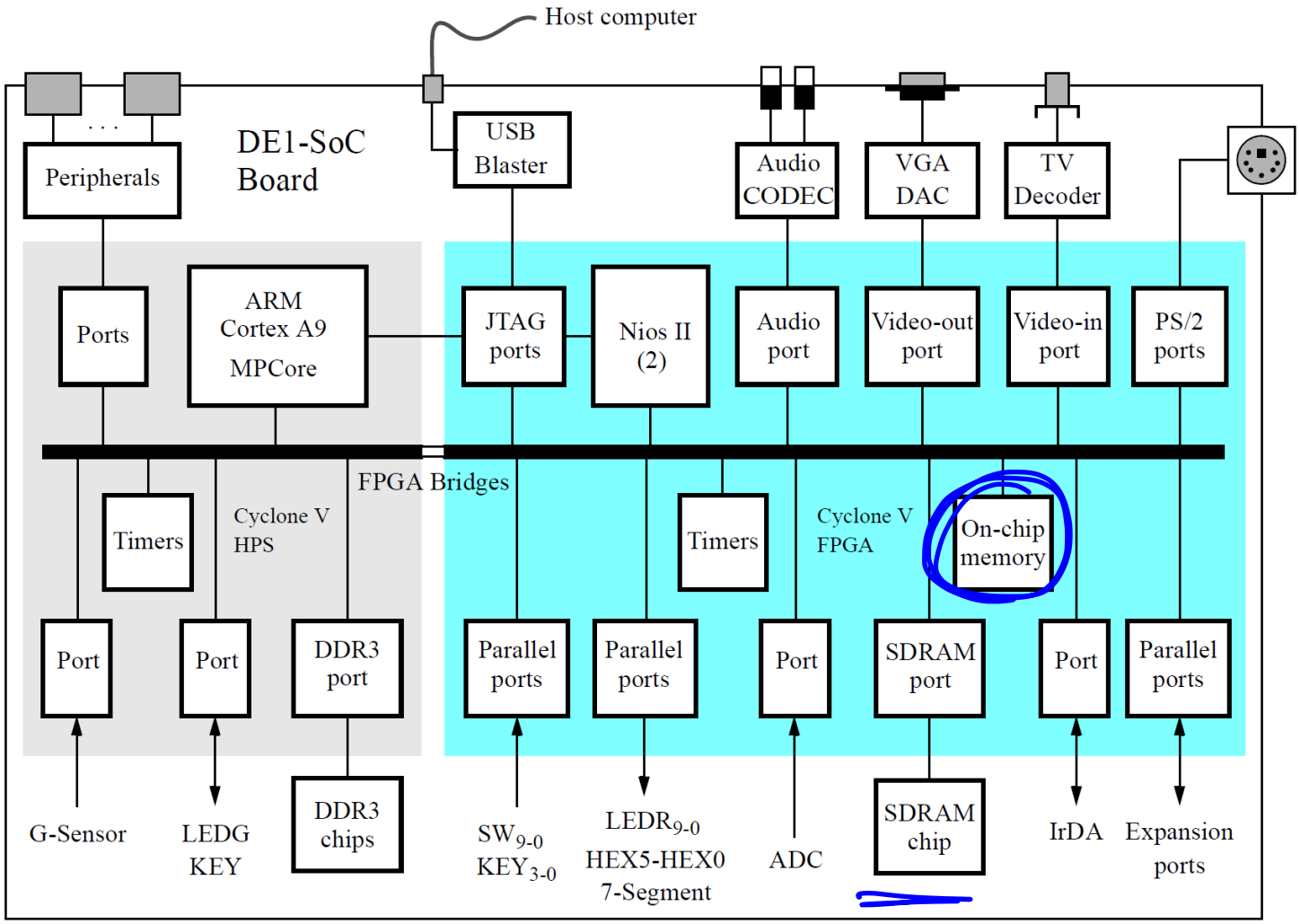
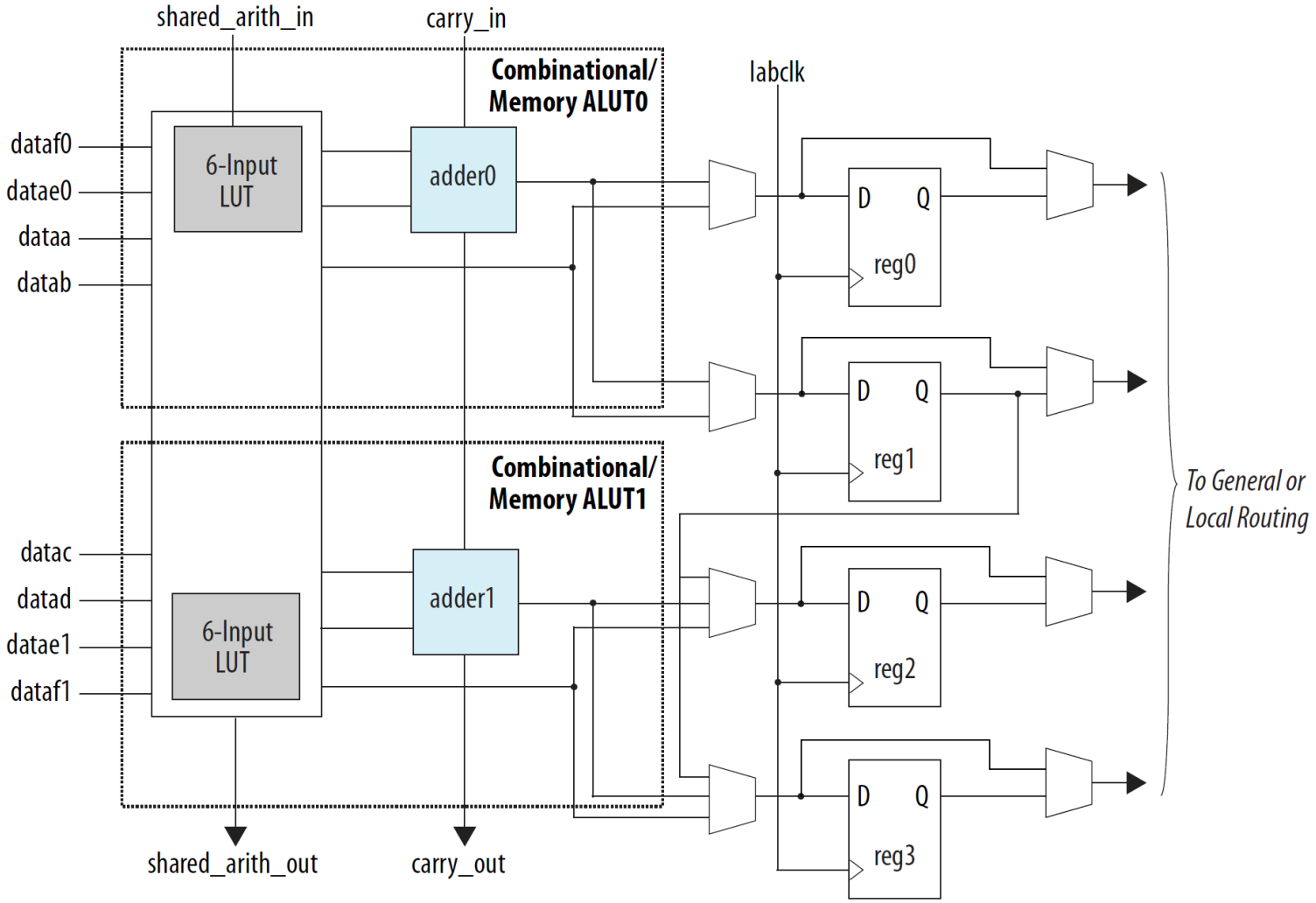



Figure 1. Block diagram of the DE1-SoC Computer.

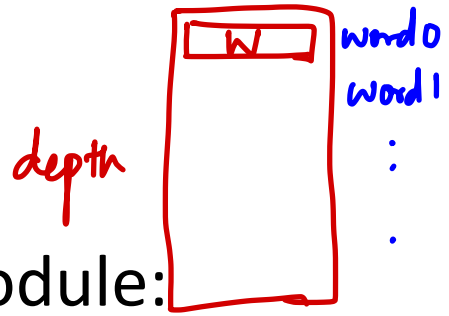
# Cyclone V Adaptive Logic Modules

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v2.pdf

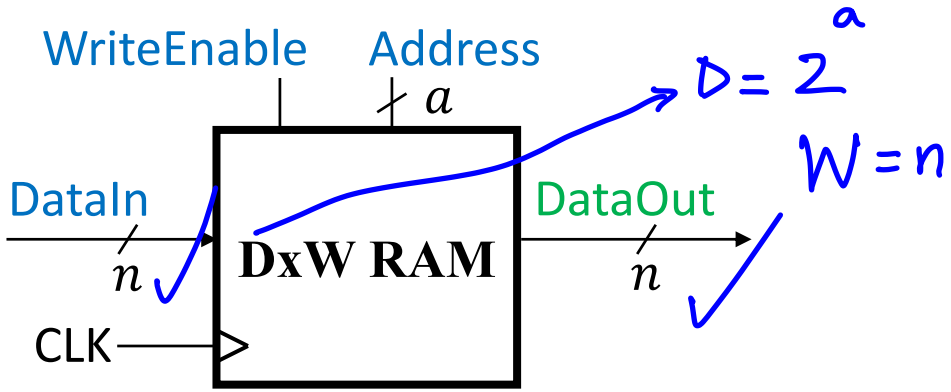
**Figure 1-5: ALM High-Level Block Diagram for Cyclone V Devices**



# Memory Modules




- Key characteristics of a simple memory module:
  - Width – the number of bits in a word
  - Depth – the number words in the module
  - Number of ports
  - Synchronicity of port access – whether or not accesses are controlled by a clock signal
  - Simultaneous address access – can the same address be used for both read and write operations
- Example:

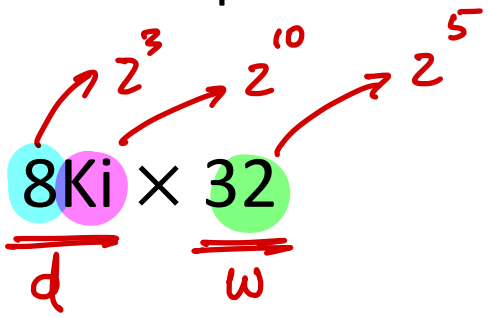


# Memory Characteristics

- Memory units are specified as **depth** × **width**.
- For the following memory units, answer:
  1) Memory capacity (in bits and bytes using IEC prefixes)
  2) Width of address bus
  3) Width of data output bus

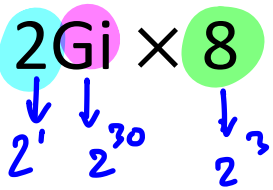
1 byte = 8 bits
$= 2^3$ bits

- Memory 1: 8Ki × 32




1. $2^{18}$ bits $= 2^{15}$ bytes
   $= 32$ KiB
2. 13 bits.
3. 32 bits.

- Memory 2: 2Gi × 8



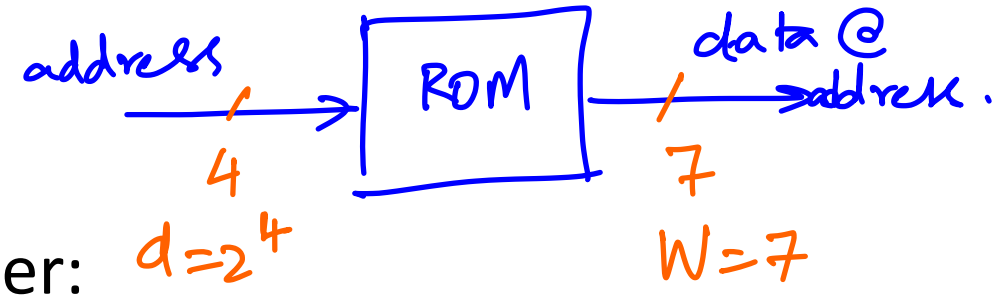

1. $2^{34}$ bits $= 2^{31}$ bytes
   $= 2$ GiB
2. 31 bits
3. 8 bits.

# Technology Break

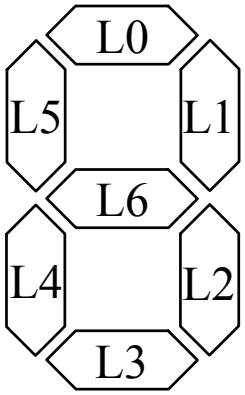
# Memory Type #1: ROM

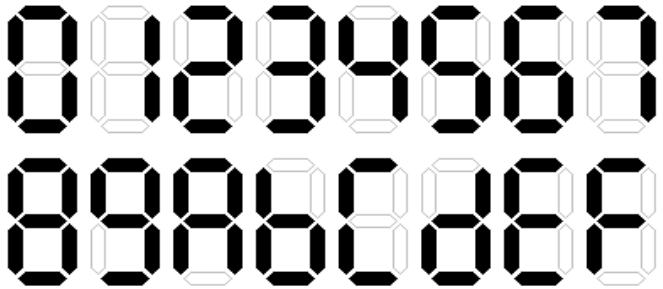
- **R**ead-**O**nly **M**emory
  - A purely *combinational* circuit (no internal state)
  - Output is determined solely by address input
- In an FPGA:
  - No actual embedded ROM, but can be emulated by a combinational circuit or a RAM with the write operation disabled
  - Only practical for small tables
- In SystemVerilog:
  - Define the ROM content as a 2-dimensional constant
  - Oftentimes a selected assignment or case statement

# Example ROM




- Hex-to-7seg LED decoder:
  - Segments:
    - Active low



| B3 | B2 | B1 | B0 | L0 | L1 | L2 | L3 | L4 | L5 | L6 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| ⋮ | | | | | | | ⋮ | | | |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

- ROM size?

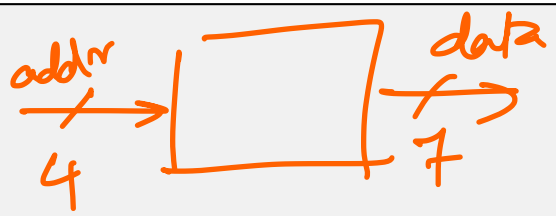
16 x 7

# Example ROM (SystemVerilog)

```
module ROM_case(addr, data);

endmodule
```

# Example ROM (SystemVerilog)

```
module ROM_case(addr, data);

   input  logic [3:0] addr;
   output logic [6:0] data;

   always_comb
      case (addr)
         // 0000          L6543210
         4'h0: data = 7'b1000000;
         4'h1: data = 7'b1111001;
         4'h2: data = 7'b0100100;
         // ...
         4'hD: data = 7'b0100001;
         4'hE: data = 7'b0000110;
         4'hF: data = 7'b0001110;
      endcase

endmodule
```



# Reading Data from a File

- Hard-coded values in your SV files can be tedious to format, debug, and swap out!
- Verilog provides *system tasks* to read data from a text file into an array: `$readmemb()` and `$readmemh()`
  - Basic usage: `$readmemb("file.txt", my_array);`
  - Reads in numerals (in **b**inary or **h**ex) from file separated by *whitespace* (*i.e.*, spaces, tabs, newlines)
  - Can avoid some recompilation
  - File can have any extension
  - Be very careful with dimensions of array and ordering of data in file!

# Example ROM (SystemVerilog)

```
module ROM_file(addr, data);

   input  logic [3:0] addr;
   output logic [6:0] data;

   logic [6:0] ROM [0:15];  // (!) unpacked dimension

   initial
      // reads binary values from file into array
      $readmemb("seg7decode.txt", ROM);

   assign data = ROM[addr];

endmodule
```

seg7decode.txt:

```
1000000 1111001 0100100 0110000
0011001 0010010 0000010 1111000
0000000 0010000 0001000 0000011
1000110 0100001 0000110 0001110
```

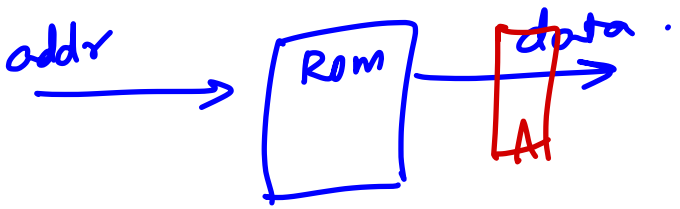
# Dynamic Array Indexing Operation

- A signal can be used as an index to access an element in the array:

```
assign data = ROM[addr];
```


```
always_comb
   case (addr)
      4'h0: data = ROM[0];
      4'h1: data = ROM[1];
      // ...
      4'hE: data = ROM[14];
      4'hF: data = ROM[15];
   endcase
```

# Synchronicity Revisited




- To synchronize either the address input or ROM output, we can add a register as appropriate
  - Sometimes called “registering” the input or output to make your module “synchronous”
  - In SystemVerilog, can be done inside or outside of the module:

```
initial
   $readmemb("seg7decode.txt", ROM);

// internally-registered ROM module
always_ff @(posedge clk)
   data <= ROM[addr];
```