Mini-MIPS

From Weste/Harris
CMOS VLSI Design

Based on MIPS

- In fact, it’s based on the multi-cycle MIPS from Patterson and Hennessy
  - Your CS/EE 3810 book...
- 8-bit version
  - 8-bit data and address
  - 32-bit instruction format
  - 8 registers numbered $0$-$7
    - $0$ is hardwired to the value 0
### Instruction Set

#### Table 1.7  
MIPS instruction set (subset supported)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>Encoding</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1, $2, $3</td>
<td>addition: $1 &lt;= $2 + $3</td>
<td>R</td>
<td>000000</td>
<td>100000</td>
</tr>
<tr>
<td>sub $1, $2, $3</td>
<td>subtraction: $1 &lt;= $2 - $3</td>
<td>R</td>
<td>000000</td>
<td>100010</td>
</tr>
<tr>
<td>and $1, $2, $3</td>
<td>bitwise and: $1 &lt;= $2 and $3</td>
<td>R</td>
<td>000000</td>
<td>100100</td>
</tr>
<tr>
<td>or $1, $2, $3</td>
<td>bitwise or: $1 &lt;= $2 or $3</td>
<td>R</td>
<td>000000</td>
<td>100101</td>
</tr>
<tr>
<td>slt $1, $2, $3</td>
<td>set less than: $1 &lt;= 1 if $2 &lt; $3</td>
<td>R</td>
<td>000000</td>
<td>101010</td>
</tr>
</tbody>
</table>

$1 <= 0 otherwise

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>Encoding</th>
<th>op</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $1, $2, imm</td>
<td>add immediate: $1 &lt;= $2 + imm</td>
<td>X</td>
<td>001000</td>
<td>n/a</td>
</tr>
<tr>
<td>beq $1, $2, imm</td>
<td>branch if equal: PC &lt;= PC + imm</td>
<td>X</td>
<td>000100</td>
<td>n/a</td>
</tr>
<tr>
<td>j destination</td>
<td>jump: PC &lt;= destination</td>
<td>J</td>
<td>000010</td>
<td>n/a</td>
</tr>
<tr>
<td>lb $1, imm($2)</td>
<td>load byte: $1 &lt;= mem[$2 + imm]</td>
<td>X</td>
<td>100000</td>
<td>n/a</td>
</tr>
<tr>
<td>sb $1, imm($2)</td>
<td>store byte: mem[$2 + imm] &lt;- $1</td>
<td>X</td>
<td>101000</td>
<td>n/a</td>
</tr>
</tbody>
</table>

*Technically, MIPS addresses specify bytes. Instructions require a four-byte word and must begin at addresses that are a multiple of four. To most effectively use instruction bits in the full 32-bit MIPS architecture, branch and jump constants are specified in words and must be multiplied by four (shifted left two bits) to be converted to byte addresses.*

CS/EE 3710

---

### Instruction Encoding

#### Figure 1.48  
Instruction encoding formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>add $rd, $ra, $rb</td>
<td>6 5 5 5 0 funct</td>
</tr>
<tr>
<td>I</td>
<td>beq $ra, $rb, imm</td>
<td>6 5 5 16</td>
</tr>
<tr>
<td>J</td>
<td>j dest</td>
<td>6 26</td>
</tr>
</tbody>
</table>

CS/EE 3710
Fibonacci C-Code

```c
int fib(void)
{
    int n = 8;        /* compute nth Fibonacci number */
    int f1 = 1, f2 = -1;    /* last two Fibonacci numbers */

    while (n != 0)    /* count down to n = 0 */
    {
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

FIG 1.50 C code for Fibonacci program

Cycle 1: \( f1 = 1 + (-1) = 0, f2 = 0 - (-1) = 1 \)
Cycle 2: \( f1 = 0 + 1 = 1, f2 = 1 - 1 = 0 \)
Cycle 3: \( f1 = 1 + 0 = 1, f2 = 1 - 0 = 1 \)
Cycle 4: \( f1 = 1 + 1 = 2, f2 = 2 - 1 = 1 \)
Cycle 5: \( f1 = 2 + 1 = 3, f2 = 3 - 1 = 2 \)
Cycle 6: \( f1 = 3 + 2 = 5, f2 = 5 - 2 = 3 \)
Fibonacci Assembly Code

```assembly
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:  addi $3, $0, 8       # initialize n=8
      addi $4, $0, 1       # initialize f1 = 1
      addi $5, $0, -1      # initialize f2 = -1
loop: beq $3, $0, end     # Done with loop if n = 0
      add $4, $4, $5      # f1 = f1 + f2
      sub $5, $4, $5      # f2 = f1 - f2
      addi $3, $3, -1     # n = n - 1
      j loop             # repeat until done
end:  sb $4, 255($0)      # store result in address 255
```

**FIG 1.51** Assembly language code for Fibonacci program

Compute 8th Fibonacci number (8’d13 or 8’h0D)
Store that number in memory location 255

---

Fibonacci Machine Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Encoding</th>
<th>Hexadecimal Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $3, $0, 8</td>
<td>001000 00000 00011 00000000000100</td>
<td>20030008</td>
</tr>
<tr>
<td>add $4, $0, 1</td>
<td>001000 00000 00100 0000000000000001</td>
<td>20040001</td>
</tr>
<tr>
<td>add $5, $0, -1</td>
<td>001000 00000 00101 1111111111111111</td>
<td>2005ffffff</td>
</tr>
<tr>
<td>beq $3, $0, end</td>
<td>000100 00011 00000 0000000000000101</td>
<td>19600000 4</td>
</tr>
<tr>
<td>add $4, $4, $5</td>
<td>000000 00100 00101 00100 0000 1000000</td>
<td>00852020</td>
</tr>
<tr>
<td>sub $5, $4, $5</td>
<td>000000 00100 00101 00101 0000 1000000</td>
<td>00852822</td>
</tr>
<tr>
<td>add $3, $3, -1</td>
<td>001000 00011 00011 1111111111111111</td>
<td>2063ffffff</td>
</tr>
<tr>
<td>j loop</td>
<td>000010 00000000000000000000000000000001</td>
<td>08000003</td>
</tr>
<tr>
<td>sb $4, 255($0)</td>
<td>101000 00000 00100 0000000000000111111</td>
<td>004000ff</td>
</tr>
</tbody>
</table>

**FIG 1.52** Machine language code for Fibonacci program

Assembly Code  

Machine Code
Architecture

Table 1.8 ALUControl determination

<table>
<thead>
<tr>
<th>aluop</th>
<th>funct</th>
<th>alucontrol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>110</td>
<td>SUB</td>
</tr>
<tr>
<td>10</td>
<td>10000</td>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>10</td>
<td>10010</td>
<td>110</td>
<td>SUB</td>
</tr>
<tr>
<td>10</td>
<td>10100</td>
<td>000</td>
<td>AND</td>
</tr>
<tr>
<td>10</td>
<td>10101</td>
<td>001</td>
<td>OR</td>
</tr>
<tr>
<td>10</td>
<td>10110</td>
<td>111</td>
<td>SLT</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>x</td>
<td>undefined</td>
</tr>
</tbody>
</table>
Connection to External Memory

<table>
<thead>
<tr>
<th>Table 1.9 Top-level inputs and outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td>ph1</td>
</tr>
<tr>
<td>ph2</td>
</tr>
<tr>
<td>reset</td>
</tr>
<tr>
<td>memdata[7:0]</td>
</tr>
</tbody>
</table>

// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
  (input                  clk,
   input memwrite,
   input      [WIDTH-1:0] adr, writedata,
   output reg [WIDTH-1:0] memdata);

  reg [31:0] RAM [{1<WIDTH-2}:1:0];
  wire [31:0] word;

  // Initialize memory with program
  initial $readmemh("memfile.dat",RAM);

  // read and write bytes from 32-bit word
  always @(posedge clk)
    if(memwrite)
      case (adr[1:0])
        2'b00: RAM[adr>>2][7:0] <= writedata;
        2'b01: RAM[adr>>2][15:8] <= writedata;
        2'b10: RAM[adr>>2][23:16] <= writedata;
        2'b11: RAM[adr>>2][31:24] <= writedata;
      endcase
    assign word = RAM[adr>>2];
    always @(*)
      case (adr[1:0])
        2'b00: memdata <= word[7:0];
        2'b01: memdata <= word[15:8];
        2'b10: memdata <= word[23:16];
        2'b11: memdata <= word[31:24];
      endcase
endmodule

Notes:
- Endianness is fixed here
- Writes are on posedge clk
- Reads are asynchronous
- This is a 32-bit wide RAM
- With 64 locations
- But with an 8-bit interface...
module exmem #(parameter WIDTH = 8, RAM_ADDR_BITS = 8)
(input clk, en,
input memwrite,
input [RAM_ADDR_BITS-1:0] adr,
input [WIDTH-1:0] writedata,
output reg [WIDTH-1:0] memdata);

reg [WIDTH-1:0] mips_ram [(2**RAM_ADDR_BITS)-1:0];
initial $readmemb("fib.dat", mips_ram);
always @(posedge clk)
 if (en) begin
   if (memwrite)
     mips_ram[adr] <= writedata;
   memdata <= mips_ram[adr];
end
endmodule

• This is synthesized to a Block RAM on the Spartan3e FPGA
• It’s 8-bits wide
• With 256 locations
• Both writes and reads are clocked

Note clock!
Byte-wide Block RAM is really 9-bits – parity bit...

(Actually dual ported too!)

<table>
<thead>
<tr>
<th>Write Mode</th>
<th>Effect on Same Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_FIRST</td>
<td>Data on DI, DIP inputs written into specified RAM location and simultaneously appears on DO, DOP outputs.</td>
</tr>
<tr>
<td>READ_FIRST</td>
<td>Data from specified RAM location appears on DO, DOP outputs.</td>
</tr>
<tr>
<td>NO_CHANGE</td>
<td>Data on DI, DIP inputs written into specified location.</td>
</tr>
</tbody>
</table>

Read-first or Write-first?

always @(posedge clk)
if (en) begin
  if (memwrite)
    mips_ram[adr] <= writedata;
    memdata <= mips_ram[adr];
end
Read_First Template

```verilog
parameter RAM_WIDTH = <ram_width>;  
parameter RAM_ADDR_BITS = <ram_addr_bits>;  

reg [RAM_WIDTH-1:0] <ram_name> [2**RAM_ADDR_BITS-1:0];  
reg [RAM_WIDTH-1:0] <output_data>;  

// The following code is only necessary if you wish to initialize the RAM  
// contents via an external file (use readmem for binary data)  
initial  
$readmem("<data_file_name>", <ram_name>, <begin_address>, <end_address>);  

always @(posedge <clock>)  
if |<ram_enable> begin  
if |<write_enable> <ram_name>[<address>] <= <input_data>;  
<output_data> <= <ram_name>[<address>];  
end
```

Write_First Template

```verilog
parameter RAM_WIDTH = <ram_width>;  
parameter RAM_ADDR_BITS = <ram_addr_bits>;  

reg [RAM_WIDTH-1:0] <ram_name> [2**RAM_ADDR_BITS-1:0];  
reg [RAM_WIDTH-1:0] <output_data>;  

// The following code is only necessary if you wish to initialize the RAM  
// contents via an external file (use readmem for binary data)  
initial  
$readmem("<data_file_name>", <ram_name>, <begin_address>, <end_address>);  

always @(posedge <clock>)  
if |<ram_enable> begin  
if |<write_enable> begin  
<ram_name>[<address>] <= <input_data>;  
<output_data> <= <input_data>;  
end  
else  
<output_data> <= <ram_name>[<address>];  
end
```
Read_First waveforms

Data_in ➔ Internal Memory ➔ DO ➔ Prior stored data

CLK
WE
DI
ADDR
DO
EN
DISABLED ➔ READ ➔ WRITE MEM[111] ➔ WRITE MEM[2222] ➔ READ

Figure 34: Waveforms of Block RAM Data Operations with READ_FIRST Selected

Write_First Waveforms

Data_in ➔ DI ➔ Internal Memory ➔ DO ➔ Data_out = Data_in

CLK
WE
DI
ADDR
DO
EN
DISABLED ➔ READ ➔ WRITE MEM[111] ➔ WRITE MEM[2222] ➔ READ

Figure 33: Waveforms of Block RAM Data Operations with WRITE_FIRST Selected
Block RAM Organization

Each block is 18k bits...

Recall – Overall System

<table>
<thead>
<tr>
<th>Device</th>
<th>Total Number of RAM Blocks</th>
<th>Total Addressable Locations (bits)</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC3S100E</td>
<td>4</td>
<td>73,728</td>
<td>1</td>
</tr>
<tr>
<td>XC3S250E</td>
<td>12</td>
<td>221,184</td>
<td>2</td>
</tr>
<tr>
<td>XC3S500E</td>
<td>20</td>
<td>368,640</td>
<td>2</td>
</tr>
<tr>
<td>XC3S1000E</td>
<td>28</td>
<td>516,096</td>
<td>2</td>
</tr>
<tr>
<td>XC3S1600E</td>
<td>36</td>
<td>663,552</td>
<td>2</td>
</tr>
</tbody>
</table>
Recall – Overall System

So, what are the implications of using a RAM that has both clocked reads and writes instead of clocked writes and async reads? (we’ll come back to this question...)

mips Block Diagram

---

13
// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
  (input clk, reset,
   input [WIDTH-1:0] memdata,
   output memread, memwrite,
   output [WIDTH-1:0] adr, writedata);
wire [31:0] instr;
wire zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
wire [1:0] aluop, pcsource, alusrcb;
wire [3:0] irwrite;
wire [2:0] alucont;
controller cont(clk, reset, instr[31:26], zero, memread, memwrite,
              alusrca, memtoreg, iord, pcen, regwrite, regdst,
              pcsource, alusrcb, aluop, irwrite);
alucontrol ac(aluop, instr[5:0], alucont);
datapath #(WIDTH, REGBITS)
dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
   regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
   zero, instr, adr, writedata);
endmodule

Controller

module controller(input clk, reset,
  input [31:0] rs1, rs2,
  input zero,
  input instr,
  output reg memread, memwrite, alusrca, memtoreg, iord, pcen,
  output reg irwrite, regdst,
  output reg [1:0] pcsource, alusrcb, aluop,
  output reg [2:0] alucont);
  // State Codes
  // Useful constants to compare against

// State Register
always @(posedge clk) 
  $if(!reset)$ state <= FETCH; 
  $else$ state <= nextstate; 
endcase

// State Code Definitions
parameter FETCH = 4'000000;
parameter INSTR = 4'000000;
parameter MEM = 4'000000;
parameter BR = 4'000000;
parameter RTYPE = 4'000000;
parameter S = 6'000000;
parameter B = 6'000000;

reg [3:0] state, nextstate;
reg psr, pc, cpc;
Control FSM

Next State Logic

```plaintext
// next state logic
always R*??
begn
  case (state)
  |  FETCH1: nextstate <= FETCH1;
  |  FETCH2: nextstate <= FETCH2;
  |  FETCH3: nextstate <= FETCH3;
  |  FETCH4: nextstate <= DECIDE;
  |  DECODE: case top)
  |  |  LB: nextstate <= MEMWR;
  |  |  SB: nextstate <= MEMWR;
  |  |  TYPE1: nextstate <= TYPE1;
  |  |  JMP: nextstate <= DECIDE;
  |  |  Jo: nextstate <= JUMP;
  |  |  default: nextstate <= FETCH1; // should never happen
  |  endcase
  MEMWR: case top)
  |  |  LB: nextstate <= LBWR;
  |  |  SB: nextstate <= SBWR;
  |  |  default: nextstate <= FETCH1; // should never happen
  endcase
  LBWR: nextstate <= LBWR;
  SBWR: nextstate <= SBWR;
  TYPE2: nextstate <= TYPE2;
  JMPWR: nextstate <= FETCH1;
  BEQX: nextstate <= FETCH1;
  JUMP: nextstate <= FETCH1;
  default: nextstate <= FETCH1; // should never happen
  endcase
end
```
Output Logic

// get all outputs to zero then conditionally assert
// just the appropriate ones
ivrite <= 4'b0000;
write <= 0; pewritecond <= 0;
regwrite <= 0; regdest <= 0;
newread <= 0; newwrite <= 0;
always (0) begin
  alucond <= 2'b00; aluop <= 2'b00;
  ressource <= 2'b00;
  iord <= 0;  seboreg <= 0;
  casenestate;
end

FETCH1:
begin
  newread <= 1;
  ivrite <= 4'b1000; // changed to reflect new memory and
  write <= 2'b00;  // get the IR bits in the right spots
  pewrite <= 1;    // FETCH 2,3,4 also changed...
end

FETCH2:
begin
  newread <= 1;
  ivrite <= 4'b0001;
  write <= 2'b00;
  pewrite <= 1;
end

FETCH3:
begin
  newread <= 1;
  ivrite <= 4'b1000;
  write <= 2'b00;
  pewrite <= 1;
end

FETCH4:
begin
  newread <= 1;
  ivrite <= 4'b1000;
  write <= 2'b00;
  pewrite <= 1;
end

Continued for the other states...

Output Logic

Why AND these two?

Two places to update the PC

pewrite on jump
pewritecond on BEQ
ALU Control

module alucontrol(input [11:0] aluop, input [5:0] funct, output reg [23:0] alucont);
always @(*)
case(aluop)
  2'b00: alucont <= 3'b100; // add for lb/ld/addi
  2'b01: alucont <= 3'b110; // sub (for beq)
  default: case(funct); // F-Type instructions
  8'b100000: alucont <= 3'b100; // add (for addi)
  8'b100001: alucont <= 3'b110; // subtract (for sub)
  8'b100010: alucont <= 3'b000; // logical and (for and)
  8'b100011: alucont <= 3'b010; // logical or (for or)
  8'b101010: alucont <= 3'b011; // set on less (for slt)
  default: alucont <= 3'b100; // should never happen
endcase
endmodule

CS/EE 3710

ALU

module alu #(parameter WIDTH = 8)
  (input [WIDTH-1:0] a, b,
  input [2:0] alucont,
  output reg [WIDTH-1:0] result);
wire [WIDTH-1:0] b2, sum, slt;
assign sum = a + b2 + alucont[2]; // slt should be 1 if most significant bit of sum is 1
assign slt = sum[WIDTH-1]?

always@(*)
case(alucont[1:0])
  2'b00: result <= a + b;
  2'b01: result <= a + b;
  2'b10: result <= sum;
  2'b11: result <= slt;
endcase
endmodule

CS/EE 3710

Table 1.8 ALUControl determination

<table>
<thead>
<tr>
<th>aluop</th>
<th>funct</th>
<th>aluControl</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>110</td>
<td>SUB</td>
</tr>
<tr>
<td>10</td>
<td>100000</td>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>10</td>
<td>100010</td>
<td>110</td>
<td>SUB</td>
</tr>
<tr>
<td>10</td>
<td>100100</td>
<td>000</td>
<td>ADD</td>
</tr>
<tr>
<td>10</td>
<td>101001</td>
<td>001</td>
<td>OR</td>
</tr>
<tr>
<td>10</td>
<td>101101</td>
<td>111</td>
<td>SLT</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>x</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Invert b if subtract...
add is a + b
sub is a + ~b + 1
subtract on slt
then check if answer is negative
zerodetect

module zerodetect #(parameter WIDTH = 8)
  (input [WIDTH-1:0] a,
   output [WIDTH-1:0] y);

  assign y = (a==0);
endmodule

Register File

module regfile #(parameter WIDTH = 8, REGBITS = 3)
  (input clk,
   input regwrite,
   input [REGBITS-1:0] ra1, ra2, wa,
   input [WIDTH-1:0] wd,
   output [WIDTH-1:0] rd1, rd2);

  reg [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 0 hardwired to 0
  always @(posedge clk)
    if (!regwrite) RAM[wa] <= wd;

  assign rd1 = ra1 ? RAM[ra1] : 0;
  assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule

What is this synthesized into?
Synthesis Report

Macro Statistics
# RAMs : 3
256x8-bit single-port RAM : 1
64x8-bit dual-port RAM : 2
# Adders/Subtractors : 1
8-bit adder carry in : 1
# Registers : 10
8-bit register : 10
# Multiplexers : 3
8-bit 4-to-1 multiplexer : 3

 Synthesizing (advanced) Unit <exmem>.
INFO:Net - The RAM <ram_mips_ram> will be implemented as a BLOCK RAM, absorbing

<table>
<thead>
<tr>
<th>ram_type</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port 1</td>
<td></td>
</tr>
<tr>
<td>aspect ratio</td>
<td>256-word x 8-bit</td>
</tr>
<tr>
<td>mode</td>
<td>read-first</td>
</tr>
<tr>
<td>clk</td>
<td>connected to signal &lt;clk&gt;</td>
</tr>
<tr>
<td>ena</td>
<td>connected to signal &lt;ena&gt;</td>
</tr>
<tr>
<td>ena</td>
<td>connected to signal &lt;ena&gt;</td>
</tr>
<tr>
<td>add1</td>
<td>connected to signal &lt;add&gt;</td>
</tr>
<tr>
<td>clk</td>
<td>connected to signal &lt;clk&gt;</td>
</tr>
<tr>
<td>d1a</td>
<td>connected to signal &lt;write_data&gt;</td>
</tr>
<tr>
<td>doa</td>
<td>connected to signal &lt;read_data&gt;</td>
</tr>
<tr>
<td>optimization</td>
<td>speed</td>
</tr>
</tbody>
</table>

CS/EE 3710
Synthesis Report

Two register files? Why?

Datapath

Fairly complex...

Not really, but it does have lots of registers instantiated directly

It also instantiates muxes...

Instruction Register
Datapath continued

Flops and MUXes

module Flop #parameter WIDTH = 8#
  (input clk, en, input [WIDTH-1:0] d, output reg [WIDTH-1:0] q);
always @posedge clk)
  q <= d;
endmodule

module Flope #parameter WIDTH = 8#
  (input clk, en, input [WIDTH-1:0] d, output reg [WIDTH-1:0] q);
always @posedge clk)
  q <= d;
endmodule

module Flop2 #parameter WIDTH = 8#
  (input clk, en, input [WIDTH-1:0] d, output reg [WIDTH-1:0] q);
always @posedge clk)
  q <= d;
endmodule

RF and ALU

Flops and muxes...
Back to the Memory Question

- What are the implications of using RAM that is clocked on both write and read?
  - Book version was async read
  - So, let’s look at the sequence of events that happen to read the instruction
  - Four steps – read four bytes and put them in four slots in the 32-bit instruction register (IR)

Instruction Fetch
• Memread, irwrite, addr, etc are set up just after clk edge
• Data comes back sometime after that (async)
• Data is captured in ir0 – ir3 on the next rising clk edge
• How does this change if reads are clocked?
One of those rare cases where using both edges of the clock is useful!

Memory Mapped I/O

- Break memory space into pieces (ranges)
  - For some of those pieces: regular memory
  - For some of those pieces: I/O
    - That is, reading from an address in that range results in getting data from an I/O device
    - Writing to an address in that range results in data going to an I/O device
### Mini-MIPS Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111 FF</td>
<td>8-bit addresses</td>
</tr>
<tr>
<td>1100 0000 C0</td>
<td>256 bytes total!</td>
</tr>
<tr>
<td>1011 1111 BF</td>
<td>Top two address bits define regions</td>
</tr>
<tr>
<td>1000 0000 80</td>
<td></td>
</tr>
<tr>
<td>0111 1111 7F</td>
<td></td>
</tr>
<tr>
<td>0100 0000 40</td>
<td></td>
</tr>
<tr>
<td>0011 1111 3F</td>
<td></td>
</tr>
<tr>
<td>0000 0000 00</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

### Enabled Devices

Only write to that device (i.e. enable it) if you’re in the appropriate memory range.

Check top two address bits!

```verilog
module flop_en # (parameter WIDTH = 8) (en, clk, d, q);
input clk, en;
input [WIDTH-1:0] d;
output reg [WIDTH-1:0] q;

always @ (posedge clk)
  if (en) q <= d;

endmodule
```
### MUXes for Return Data

```verilog
module mux2 #(parameter WIDTH = 8)
  (input [WIDTH-1:0] d0, d1,
   input s,
   output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

Use MUX to decide if data is coming from memory or from I/O

Check address bits!

---

### Lab2 in a Nutshell

- Understand and simulate mips/exmem
  - Add ADDI instruction
  - Fibonacci program – correct if 8’0d is written to memory location 255
- Augment the system
  - Add memory mapped I/O to switches/LEDs
  - Write new Fibonacci program
  - Simulate in ISE
  - Demonstrate on your board
My Initial Testbench...

```verilog
// My Initial Testbench...

module my_testbench #(parameter n = 10);
  // Inputs
  reg clk;
  reg reset;
  // Output
  wire result;
  // Enumerated types
  enum {0, 1, 2, 3, 4, 5} dir;

  // Instantiate the DUT
  my_dut dut (.clk(clk),
               .reset(reset),
               .result(result));

  always begin
    cli = 0;
    #100
    result
  end
endmodule
```

My Initial Results

![Simulation Results](image)

- Current Simulation Time: 5000 ns
- Xilinx ISE Simulator
- Verilog Design Results
  - All signals are within expected ranges
  - No assertions failed

This is a lite version of Xilinx ISE Simulator. Simulator is doing circuit initialization process. Finished circuit initialization process. Try 4000 ns or run 4000 ns and check for successful Fibonacci completion.