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>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>Encoding</th>
<th>op</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1$, $t2$, $t3$</td>
<td>$t1 := t2 + t3$</td>
<td>$000000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>sub $t1$, $t2$, $t3$</td>
<td>$t1 := t2 - t3$</td>
<td>$000000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>and $t1$, $t2$, $t3$</td>
<td>$t1 := t2 &amp; t3$</td>
<td>$000000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>or $t1$, $t2$, $t3$</td>
<td>$t1 := t2</td>
<td>t3$</td>
<td>$000000$</td>
<td>10000</td>
</tr>
<tr>
<td>xor $t1$, $t2$, $t3$</td>
<td>$t1 := t2 \oplus t3$</td>
<td>$000000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>slt $t1$, $t2$, $t3$</td>
<td>if $t2 &lt; t3$ then $t1 := 1$ else $t1 := 0$</td>
<td>$000000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>addi $t1$, $t2$, imm</td>
<td>$t1 := t2 + \text{imm}$</td>
<td>$010000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>beq $t1$, $t2$, imm</td>
<td>branch if $t1 = t2$</td>
<td>$010000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>j destination</td>
<td>jump to code specified by <code>dest</code></td>
<td>$000010$</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>lb $t1$, imm($t2$)</td>
<td>load byte at $t2 + \text{imm}$</td>
<td>$100000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
<tr>
<td>sb $t1$, imm($t2$)</td>
<td>store byte at $t2 + \text{imm}$</td>
<td>$100000$</td>
<td>10000</td>
<td>n/a</td>
</tr>
</tbody>
</table>

*Technically, MIPS allows special forms. Instructions require a four-byte word and must begin at addresses that are a multiple of four. This is to reflect the fact that instruction words are placed in words and must be multiplied by four. (Integers left justified to be contained in four addresses.)

Fibonacci C-Code

```c
int fib(int n) {
    if (n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}
```

C-Code for Fibonacci Program

### Fibonacci C-Code

```c
int fib(void) {
    int n = 8;
    int f1 = 1, f2 = 1;
    while (n > 0) {
        if (n == 0) {
            return f1;
        } else {
            f1 = f1 + f2;
            f2 = f1 - f2;
            n = n - 1;
        }
    }
    return 0;
}
```

Fibonacci C-Code

```c
int fib(void) {
    int n = 8;
    int f1 = 1, f2 = 1;
    while (n > 0) {
        if (n == 0) {
            return f1;
        } else {
            f1 = f1 + f2;
            f2 = f1 - f2;
            n = n - 1;
        }
    }
    return 0;
}
```

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: $f1 = 2 + 1 = 3$, $f2 = 3 - 2 = 1$ |
| Cycle 6: $f1 = 3 + 2 = 5$, $f2 = 5 - 2 = 3$ |
Fibonacci Assembly Code

```
# fib.asm
# register usage: S3: x S4: f1 S5: f2
# return value written to address 255
fib: addi $4, 0, 0      # initialize n=0
   addi $4, $4, 1      # initialize f1 = 1
   addi $5, 0, 0       # initialize f2 = -1
loop: beq $5, 0, end   # done with loop if n = 0
   add $4, $4, $5     # f1 = f1 + f2
   sub $5, $5, $4     # f2 = f1 - f2
   addi $5, $5, -1    # n = n - 1, loop = loop
end: sb $4, 255($0)    # store result in address 255
```

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

Fibonacci Machine Code

```
addi $3, 0, 0
add,$4,$5,1
beq,$5,0,end
add,$4,$4,$5
add,$5,$5,$4
sub,$5,$5,$4
jeep
sb,$4,255($0)
```

Another View

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory Encoding</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, 3, 0, 0</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>add, 4, 0, 1</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>beq, 5, 0, end</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>add, 4, 4, 5</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>sub, 5, 5, 4</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>jeep</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
<tr>
<td>sb, 4, 255($0)</td>
<td>0010000000000000</td>
<td>0010000000000000</td>
</tr>
</tbody>
</table>

Architecture

Control FSM
Connection to External Memory

### Table 1.9 Top-level inputs and outputs

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>φn</td>
<td>addr[7:0]</td>
</tr>
<tr>
<td>φo</td>
<td>writedata[7:0]</td>
</tr>
<tr>
<td>reset</td>
<td>memread</td>
</tr>
<tr>
<td>memw</td>
<td>memwrite</td>
</tr>
</tbody>
</table>

---

Exmem.v

```vhdl
module exmem #(parameter WIDTH = 8, RAM_ADDR_BITS = 8)
  (input clk, en,
   input memwrite,
   input [RAM_ADDR_BITS-1:0] adr,
   input [WIDTH-1] writedata,
   output reg [WIDTH-1] memdata);
  reg [WIDTH-1] 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

Note clock!

---

External Memory from Book

```vhdl
module exmem #(parameter WIDTH = 8)
  (input clk,
   input memwrite,
   input [WIDTH-1] writedata,
   output reg [WIDTH-1] memdata);
  reg [31:0] RAM[(2^WIDTH-1)-1:0];
  initial $readmemh("memfile.dat",RAM);
  assign word = RAM[adr>>2];
  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
  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...

---

Block RAM

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

(Actually dual ported too?)

---

Our Block Ram

- Read-first or Write-first?

```vhdl
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

---

Block RAM

- Read-first or Write-first?

```vhdl
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
Block RAM Organization

Block RAM is Single or Dual ported

Each block is 18k bits...

Recall – Overall System

Clock

MIPS processor

MMU

External memory

Main memory

Data transfer from Port A to Port B

Reset

External memory

MIPS computer system
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...)

Controller

State Codes
Useful constants to compare against

State Register

Controller Block Diagram

Control FSM

Next State Logic
Output Logic

Very common way to deal with default values in combinational
Always blocks

Continued for the other states...

ALU

Why AND these two?

Two places to update the PC
pcwrite on jump
pcwritecond on BEQ

zerodetect

What is this synthesized into?
Synthesis Report

Datapath

Two register files? Why?

Datapath continued

Flops and MUXes
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

mips + exmem

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

- **8-bit addresses**
- **256 bytes total!**
- **Top two address bits define regions**
- **64 bytes**

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

Enabled Devices

- Only write to that device (i.e. enable it) if you’re in the appropriate memory range.
- Check top two address bits!

My Initial Testbench...

```
module flipflop #parameter WIDTH = #16)
   input [WIDTH-1:0] d,
   input [WIDTH-1:0] q,
   output reg [WIDTH-1:0] q;
always @ (posedge clk)
   if (len) q <= d;
endmodule
```

My Initial Results

```
use MUX to decide if data is coming from memory or from I/O
- Check address bits!
```