

## Another Example: MIPS

From the Harris/Weste book  
Based on the MIPS-like processor from  
the Hennessy/Patterson book

## MIPS Architecture

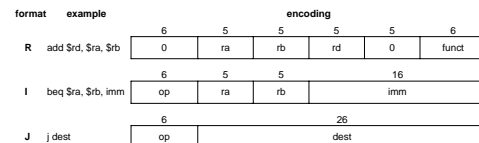
- ◆ Example: subset of MIPS processor architecture
  - Drawn from Patterson & Hennessy
- ◆ MIPS is a 32-bit architecture with 32 registers
  - Consider 8-bit subset using 8-bit datapath
  - Only implement 8 registers (\$0 - \$7)
  - \$0 hardwired to 00000000
  - 8-bit program counter

## Instruction Set

Instruction	Function	Encoding	op	funct
add \$1, \$2, \$3	addition: $S1 \rightarrow S2 + S3$	R	000000	100000
sub \$1, \$2, \$3	subtraction: $S1 \rightarrow S2 - S3$	R	000000	100010
and \$1, \$2, \$3	bitwise and: $S1 \rightarrow S2 \text{ and } S3$	R	000000	100100
or \$1, \$2, \$3	bitwise or: $S1 \rightarrow S2 \text{ or } S3$	R	000000	100101
slt \$1, \$2, \$3	set less than: $S1 \rightarrow 1 \text{ if } S2 < S3$ $S1 \rightarrow 0 \text{ otherwise}$	R	000000	101010
addi \$1, \$2, imm	add immediate: $S1 \rightarrow S2 + \text{imm}$	I	001000	n/a
beq \$1, \$2, imm	branch if equal: $PC \rightarrow PC + \text{imm}^*$	I	000100	n/a
j destination	jump: $PC\_destination^*$	J	000010	n/a
lb \$1, imm(\$2)	load byte: $S1 \rightarrow \text{mem}[S2 + \text{imm}]$	I	100000	n/a
sb \$1, imm(\$2)	store byte: $\text{mem}[S2 + \text{imm}] \rightarrow S1$	I	110000	n/a

## Instruction Encoding

- ◆ 32-bit instruction encoding
  - Requires four cycles to fetch on 8-bit datapath



## Fibonacci (C)

$$f_0 = 1; f_1 = -1$$

$$f_n = f_{n-1} + f_{n-2}$$

$$f = 1, 1, 2, 3, 5, 8, 13, \dots$$

```
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;
}
```

## Fibonacci (Assembly)

- ◆ 1<sup>st</sup> statement:  $n = 8$
- ◆ How do we translate this to assembly?

## Fibonacci (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
```

## Fibonacci (Binary)

- ◆ 1<sup>st</sup> statement: addi \$3, \$0, 8
- ◆ How do we translate this to machine language?
  - Hint: use instruction encodings below

format	example	encoding										
R	addi \$rd, \$ra, \$rb	<table border="1"> <tr> <td>6</td> <td>5</td> <td>5</td> <td>5</td> <td>6</td> </tr> <tr> <td>0</td> <td>ra</td> <td>rb</td> <td>rd</td> <td>0 funct</td> </tr> </table>	6	5	5	5	6	0	ra	rb	rd	0 funct
6	5	5	5	6								
0	ra	rb	rd	0 funct								
I	beq \$ra, \$rb, imm	<table border="1"> <tr> <td>6</td> <td>5</td> <td>5</td> <td>16</td> </tr> <tr> <td>op</td> <td>ra</td> <td>rb</td> <td>imm</td> </tr> </table>	6	5	5	16	op	ra	rb	imm		
6	5	5	16									
op	ra	rb	imm									
J	j dest	<table border="1"> <tr> <td>6</td> <td>26</td> </tr> <tr> <td>op</td> <td>dest</td> </tr> </table>	6	26	op	dest						
6	26											
op	dest											

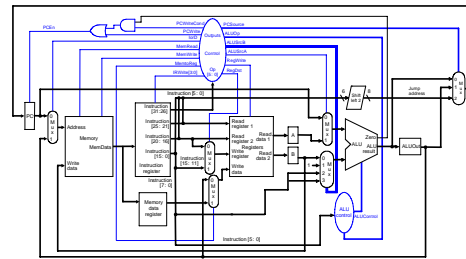
## Fibonacci (Binary)

- ◆ Machine language program

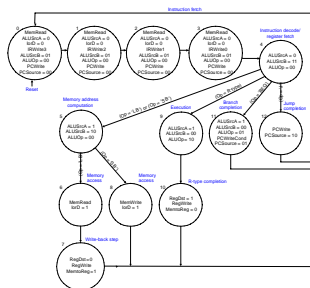
Instruction	Binary Encoding	Hexadecimal Encoding
addi \$3, \$0, 8	001000 00000 00011 0000000000001000	20030008
addi \$4, \$0, 1	001000 00000 00100 0000000000000001	20040001
addi \$5, \$0, -1	001000 00000 00101 1111111111111111	2005ffff
beq \$3, \$0, end	000100 00011 00000 0000000000000101	10600005
add \$4, \$4, \$5	000000 00100 00101 00100 00000 100000	00852020
sub \$5, \$4, \$5	000000 00100 00101 00101 00000 100010	00852822
addi \$3, \$3, -1	001000 00011 00011 1111111111111111	2063ffff
j loop	000010 00000000000000000000000011	08000003
sb \$4, 255(\$0)	110000 00000 00100 0000000011111111	a09400ff

## MIPS Microarchitecture

- ◆ Multicycle architecture from Patterson & Hennessy

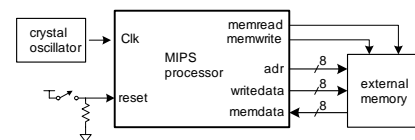


## Multicycle Controller



## Logic Design

- ◆ Start at top level
  - Hierarchically decompose MIPS into units
- ◆ Top-level interface







```
// independent of bit width, load instruction into four 8-bit registers over four cycles
flopnr #(8) lr0(clk, lrwrite[0], memdata[7:0], instr[7:0]);
flopnr #(8) lr1(clk, lrwrite[1], memdata[7:0], instr[15:8]);
flopnr #(8) lr2(clk, lrwrite[2], memdata[7:0], instr[23:16]);
flopnr #(8) lr3(clk, lrwrite[3], memdata[7:0], instr[31:24]);

// datapath
flopnr #(WIDTH) pcreg(clk, reset, pcon, nextpc, pc);
flop #(WIDTH) mdr(clk, memdata, md);
flop #(WIDTH) areg(clk, rd1, a);
flop #(WIDTH) wrd(clk, rd2, writedata);
flop #(WIDTH) res(clk, aluresult, aluout);
mux2 #(WIDTH) admux(pc, aluout, lord, adr);
mux2 #(WIDTH) src1mux(pc, a, alurca, src1);
mux4 #(WIDTH) src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
    constb4, alurcb, src2);
mux4 #(WIDTH) pcmux(aluresult, aluout, constb4, CONST_ZERO, pcsource, nextpc);
mux2 #(WIDTH) wdmux(aluout, md, memtoreg, wd);
regfile #(WIDTH, REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
alu #(WIDTH) alunit(src1, src2, alucont, aluresult);
zerodetect #(WIDTH) zd(aluresult, zero);
endmodule
```

## Verilog: Datapath 2

## Logic Design

- ◆ Start at top level
  - Hierarchically decompose MIPS into units
- ◆ Top-level interface

## Verilog: exmemory

```
// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
    (clk, memwrite, adr, writedata, memdata);

    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;

    initial
    begin
        $readmemh("memfile.dat", RAM);
    end

    // 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])
        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
    endcase
endmodule
```

## Verilog: exmemory

```
// external memory accessed by MIPS
module exmem #(parameter WIDTH = 8)
    (clk, memwrite, adr, writedata,
    memdata);

    input    clk;
    input    memwrite;
    input    [WIDTH-1:0] adr, writedata;
    output    [WIDTH-1:0] memdata;

    wire memwriteB, clkB;

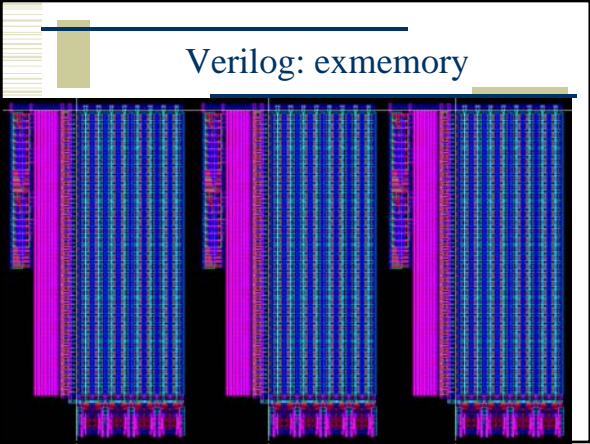
    // UMC RAM has active low write enable...
    not(memwriteB, memwrite);

    // Looks like you need to clock the memory early
    // to make it work with the current control...
    not(clkB, clk);

    // Instantiate the UMC SPRAM module
    UMC130SPRAM_8_8 mips_ram (
        .CK(clkB),
        .CEN(1'b0),
        .WEN(memwriteB),
        .OEN(1'b0),
        .ADR(adr),
        .DI(writedata),
        .DOUT(memdata));
endmodule
```

## Verilog: exmemory

- ◆ Use makemem to generate memory
  - Limited to 64 rows...
  - Can build it out of multiple SRAMs
  - SRAM64x8 (four copies)
  - Approx 450x240microns each...



## Verilog: exmemory

- ◆ Simulation model is the switch-level simulation of the Verilog structural netlist
- ◆ Or you could write a behavioral model...