

An 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

Table 1.7 MIPS instruction set (subset supported)

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

Instruction Encoding

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

format	example	encoding					
		6	5	5	5	5	6
R	add \$rd, \$ra, \$rb	0	ra	rb	rd	0	funct
		6	5	5	16		
I	beq \$ra, \$rb, imm	op	ra	rb	imm		
		6	26				
J	j dest	op	dest				

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)

- ◆ 1st statement: `int n = 8;`
- ◆ How do we translate this to assembly?
 - Decide which register should hold its value
 - load an immediate value into that register
 - But, there's no "load immediate" instruction...
 - But, there is an `addi` instruction, and there's a convenient register that's always pinned to 0
- ◆ `addi $3, $0, 8 ; load 0+8 into register 3`

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)

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

format	example	encoding					
		6	5	5	5	5	6
R	add \$rd, \$ra, \$rb	0	ra	rb	rd	0	funct
		6	5	5	16		
I	beq \$ra, \$rb, imm	op	ra	rb	imm		
		6	26				
J	j dest	op	dest				

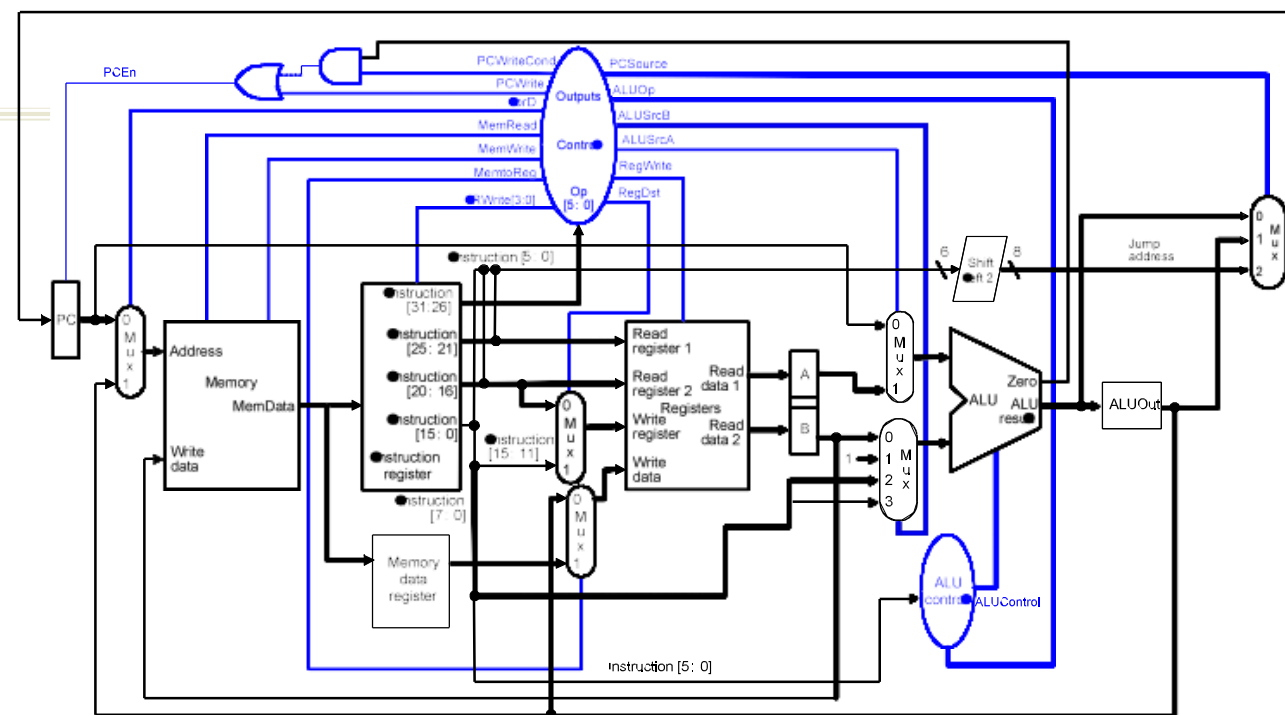
Fibonacci (Binary)

◆ Machine language program

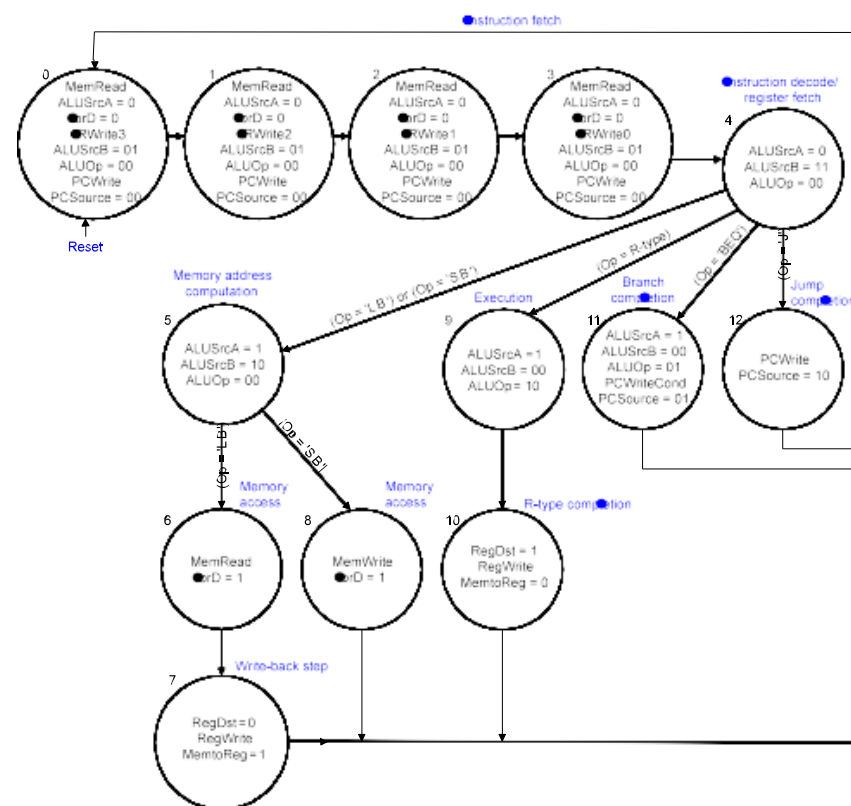
Instruction	Binary Encoding	Hexadecimal Encoding
addi \$3, \$0, 8	001000 00000 00011 00000000000001000	20030008
addi \$4, \$0, 1	001000 00000 00100 00000000000000001	20040001
addi \$5, \$0, -1	001000 00000 00101 11111111111111111	2005ffff
beq \$3, \$0, end	000100 00011 00000 00000000000000101	10600005
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 11111111111111111	2063ffff
j loop	000010 00000000000000000000000000011	08000003
sb \$4, 255(\$0)	110000 00000 00100 00000000111111111	a00400ff

MIPS Microarchitecture

- ◆ Multicycle μarchitecture from Patterson & Hennessy

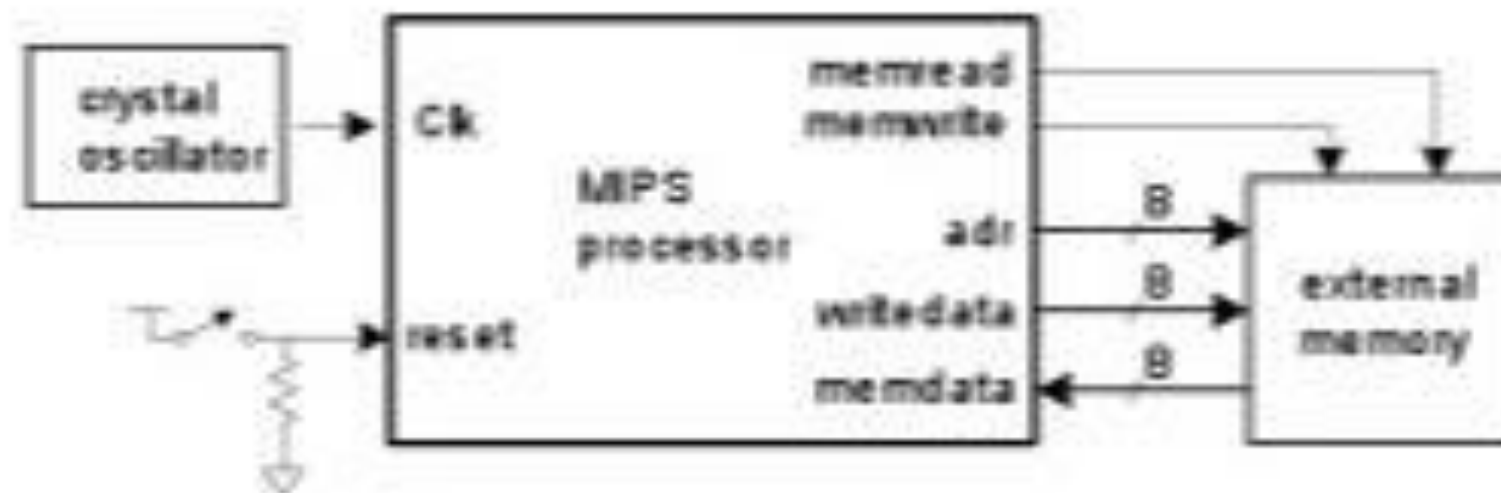


Multicycle Controller



Logic Design

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

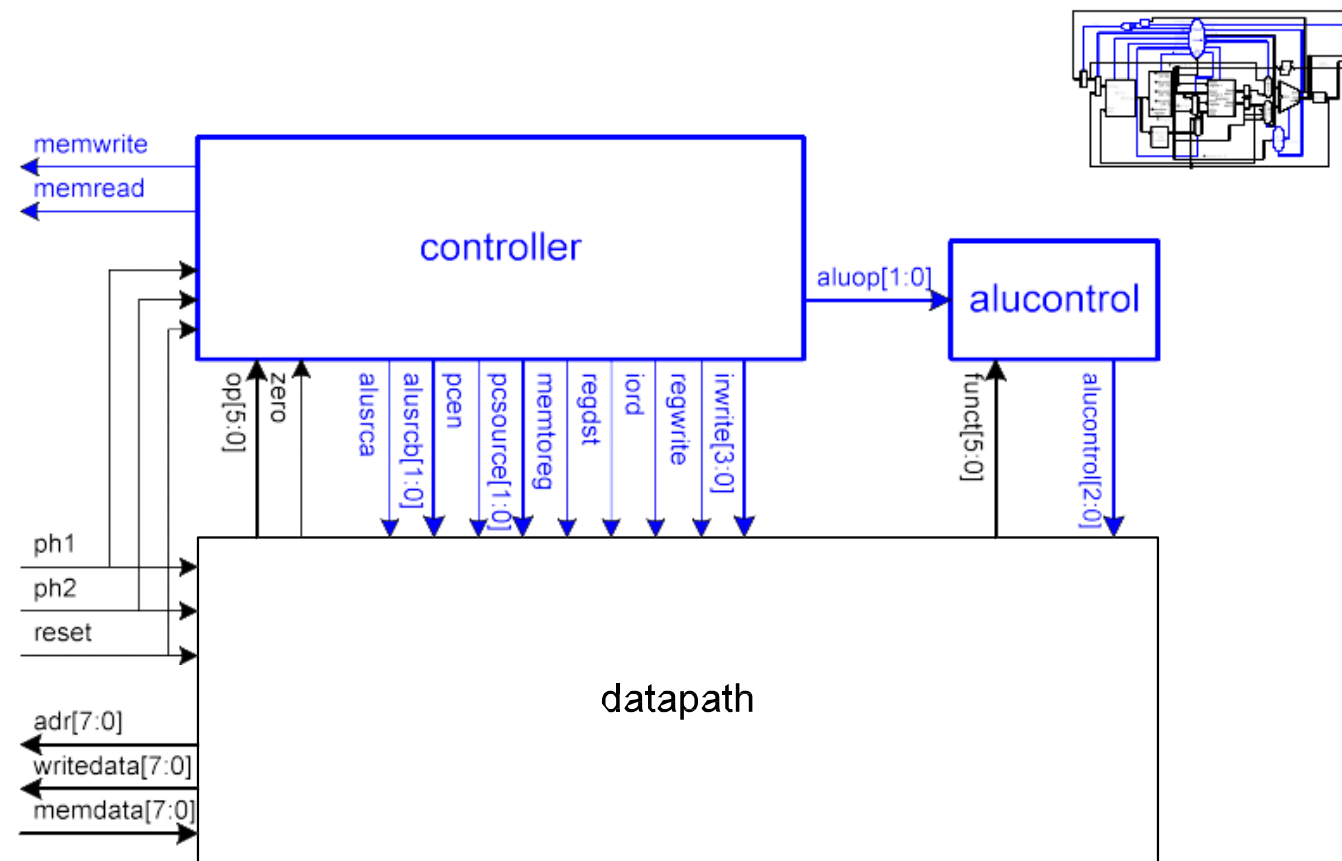


Verilog Code

```
// top level design includes both mips processor and memory
module mips_mem #(parameter WIDTH = 8, REGBITS = 3)(clk, reset);
    input clk, reset;
    wire      memread, memwrite;
    wire [WIDTH-1:0] adr, writedata;
    wire [WIDTH-1:0] memdata;

    // instantiate the mips processor
    mips #(WIDTH,REGBITS) mips(clk, reset, memdata, memread,
        memwrite, adr, writedata);
    // instantiate memory for code and data
    exmem #(WIDTH) exmem(clk, memwrite, adr, writedata, memdata);
endmodule
```

Block Diagram



II 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
```

Top-level code

```
module controller(input clk, reset,
    input [5:0] op,
    input zero,
    output reg memread, memwrite, alusrca, memtoreg, iord,
    output pcen,
    output reg regwrite, regdst,
    output reg [1:0] pcsource, alusrcb, aluop,
    output reg [3:0] irwrite);

    parameter FETCH1 = 4'b0001;
    parameter FETCH2 = 4'b0010;
    parameter FETCH3 = 4'b0011;
    parameter FETCH4 = 4'b0100;
    parameter DECODE = 4'b0101;
    parameter MEMADR = 4'b0110;
    parameter LBRD = 4'b0111;
    parameter LBWR = 4'b1000;
    parameter SBWR = 4'b1001;
    parameter RTYPEEX = 4'b1010;
    parameter RTYPEWR = 4'b1011;
    parameter BEQEX = 4'b1100;
    parameter JEX = 4'b1101;
    parameter ADDIWR = 4'b1110; // II added for ADDI

    parameter LB = 6'b100000;
    parameter SB = 6'b101000;
    parameter RTYPE = 6'b0;
    parameter BEQ = 6'b000100;
    parameter J = 6'b000010;
    parameter ADDI = 6'b001000; // III added for ADDI

    reg [3:0] state, nextstate;
    reg pcwrite, pcwritecond;
```

State Encodings...

Controller Parameters

Opcodes...

Local reg variables...

Main state machine – NS logic

II state register

```
always @(posedge clk)
  if(reset) state <= FETCH1;
  else state <= nextstate;
```

II next state logic (combinational)

```
always @(*)
  begin
    case(state)
      FETCH1: nextstate <= FETCH2;
      FETCH2: nextstate <= FETCH3;
      FETCH3: nextstate <= FETCH4;
      FETCH4: nextstate <= DECODE;
      DECODE: case(op)
        LB:   nextstate <= MEMADR;
        SB:   nextstate <= MEMADR;
        ADDI: nextstate <= MEMADR;
        RTYPE: nextstate <= RTYPEEX;
        BEQ:  nextstate <= BEQEX;
        J:    nextstate <= JEX;
        II should never happen
        default: nextstate <= FETCH1;
      endcase
    endcase
```

```
MEMADR: case(op)
  LB:   nextstate <= LBRD;
  SB:   nextstate <= SBWR;
  ADDI: nextstate <= ADDIWR;
  II should never happen
  default: nextstate <= FETCH1;
endcase
LBRD:  nextstate <= LBWR;
LBWR:  nextstate <= FETCH1;
SBWR:  nextstate <= FETCH1;
RTYPEEX: nextstate <= RTYPEWR;
RTYPEWR: nextstate <= FETCH1;
BEQEX:  nextstate <= FETCH1;
JEX:    nextstate <= FETCH1;
ADDIWR: nextstate <= FETCH1;
II should never happen
default: nextstate <= FETCH1;
endcase
end
```

Setting Control Signal Outputs

```
always @(*)
begin
    // set all outputs to zero, then
    // conditionally assert just the
    // appropriate ones
    irwrite <= 4'b0000;
    pcwrite <= 0; pcwritecond <= 0;
    regwrite <= 0; regdst <= 0;
    memread <= 0; memwrite <= 0;
    alusrca <= 0; alusrcb <= 2'b00;
    aluop <= 2'b00; pcsource <= 2'b00;
    iord <= 0; memtoreg <= 0;
    case(state)
        FETCH1:
            begin
                memread <= 1;
                irwrite <= 4'b0001;
                alusrcb <= 2'b01;
                pcwrite <= 1;
            end
        FETCH2:
            begin
                memread <= 1;
                irwrite <= 4'b0010;
                alusrcb <= 2'b01;
                pcwrite <= 1;
            end
        FETCH3:
            begin
                memread <= 1;
                irwrite <= 4'b0100;
                alusrcb <= 2'b01;
                pcwrite <= 1;
            end
        FETCH4:
            begin
                memread <= 1;
                irwrite <= 4'b1000;
                alusrcb <= 2'b01;
                pcwrite <= 1;
            end
        DECODE: alusrcb <= 2'b11;
    endcase
end
endmodule
```

Verilog: alucontrol

```
module alucontrol(input [1:0] aluop,  
                 input [5:0] funct,  
                 output reg [2:0] alucont);  
  
    always @(*)  
        case(aluop)  
            2'b00: alucont <= 3'b010; // add for lblsbladdi  
            2'b01: alucont <= 3'b110; // sub (for beq)  
            default: case(funct) // R-Type instructions  
                6'b100000: alucont <= 3'b010; // add (for add)  
                6'b100010: alucont <= 3'b110; // subtract (for sub)  
                6'b100100: alucont <= 3'b000; // logical and (for and)  
                6'b100101: alucont <= 3'b001; // logical or (for or)  
                6'b101010: alucont <= 3'b111; // set on less (for slt)  
                default: alucont <= 3'b101; // should never happen  
            endcase  
        endcase  
    endcase  
endmodule
```

Verilog: 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 b2 = alucont[2] ? -b:b;
    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
```

Verilog: regfile

```
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 (combinational)
// write third port on rising edge of clock
// register 0 is hardwired to 0

```
    always @(posedge clk)
        if (regwrite) RAM[wa] <= wd;
```

```
    assign rd1 = ra1 ? RAM[ra1] : 0;
```

```
    assign rd2 = ra2 ? RAM[ra2] : 0;
```

```
endmodule
```

Verlog: Other stuff

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

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

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

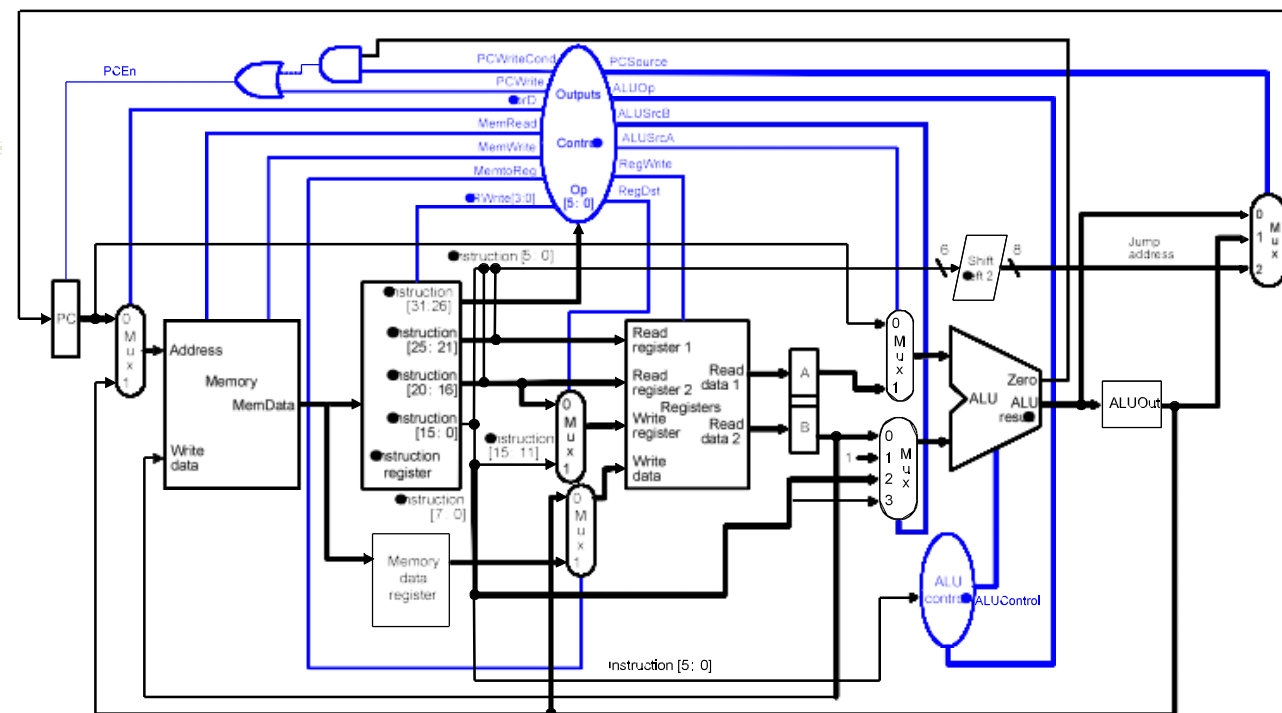
```
module flopenr #(parameter WIDTH = 8)
    (input clk, reset, en,
     input [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);
    always @(posedge clk)
        if (reset) q <= 0;
        else if (en) q <= d;
endmodule
```

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

```
module mux4 #(parameter WIDTH = 8)
    (input [WIDTH-1:0] d0, d1, d2, d3,
     input [1:0] s,
     output reg [WIDTH-1:0] y);
    always @(*)
        case(s)
            2'b00: y <= d0;
            2'b01: y <= d1;
            2'b10: y <= d2;
            2'b11: y <= d3;
        endcase
endmodule
```

MIPS Microarchitecture

- ◆ Multicycle μ architecture from Patterson & Hennessy



```

module datapath #(parameter WIDTH = 8, REGBITS = 3)
    (input      clk, reset,
     input [WIDTH-1:0] memdata,
     input      alusrca, memtoreg, iord, pcen, regwrite, regdst,
     input [1:0]  pcsource, alusrcb,
     input [3:0]  irwrite,
     input [2:0]  alucont,
     output      zero,
     output [31:0] instr,
     output [WIDTH-1:0] adr, writedata);

```

II the size of the parameters must be changed to match the WIDTH parameter

```
localparam CONST_ZERO = 8'b0;
```

```
localparam CONST_ONE = 8'b1;
```

```
wire [REGBITS-1:0] ra1, ra2, wa;
```

```
wire [WIDTH-1:0] pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
                aluout, constx4;
```

II shift left constant field by 2

```
assign constx4 = {instr[WIDTH-3:0],2'b00};
```

II register file address fields

```
assign ra1 = instr[REGBITS+20:21];
```

```
assign ra2 = instr[REGBITS+15:16];
```

```
mux2    #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11], regdst, wa);
```

Verilog: Datapath 1

II independent of bit width, load instruction into four 8-bit registers over four cycles

```
flopen #(8) ir0(clk, irwrite[0], memdata[?:0], instr[?:0]);  
flopen #(8) ir1(clk, irwrite[1], memdata[?:0], instr[15:8]);  
flopen #(8) ir2(clk, irwrite[2], memdata[?:0], instr[23:16]);  
flopen #(8) ir3(clk, irwrite[3], memdata[?:0], instr[31:24]);
```

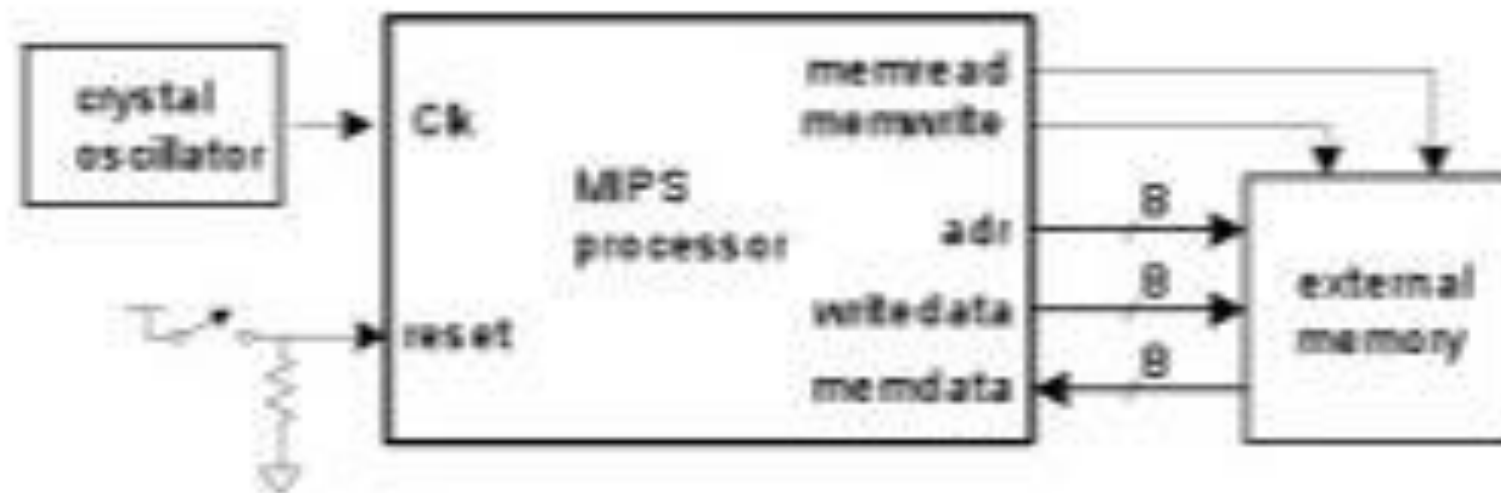
II datapath

```
flopenr #(WIDTH) pcreg(clk, reset, pcen, 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) adrmux(pc, aluout, iord, adr);  
mux2 #(WIDTH) src1mux(pc, a, alusrca, src1);  
mux4 #(WIDTH) src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],  
                      constx4, alusrcb, src2);  
mux4 #(WIDTH) pcmux(aluresult, aluout, constx4, 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

II 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
```

II read and write bytes from 32-bit word

```
    always @(posedge clk)
    if(memwrite)
        case (adr[1:0])
            2'b00: RAM[adr>>2][?: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[?:0];
        2'b01: memdata <= word[15:8];
        2'b10: memdata <= word[23:16];
        2'b11: memdata <= word[31:24];
    endcase
```

```
endmodule
```

Synthesized memory?

- ◆ If you synthesize the Verilog, you'll get a memory
 - But – it will be huge!
 - It will be made of your DFF cells
 - plus synthesized address decoders
- Custom memory is much smaller
- but much trickier to get right
- ... see details in VGA slides ...

Verilog: exmemory

// external memory accessed by MIPS

```
module exmem #(parameter WIDTH = 8)
    (clk, memwrite, adr, writedata,
     memdata);
```

```
    input          clk;
```

```
    input          memwrite
```

```
    input          ;
```

```
    output [WIDTH-1:0] adr, writedata;
           [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),
```

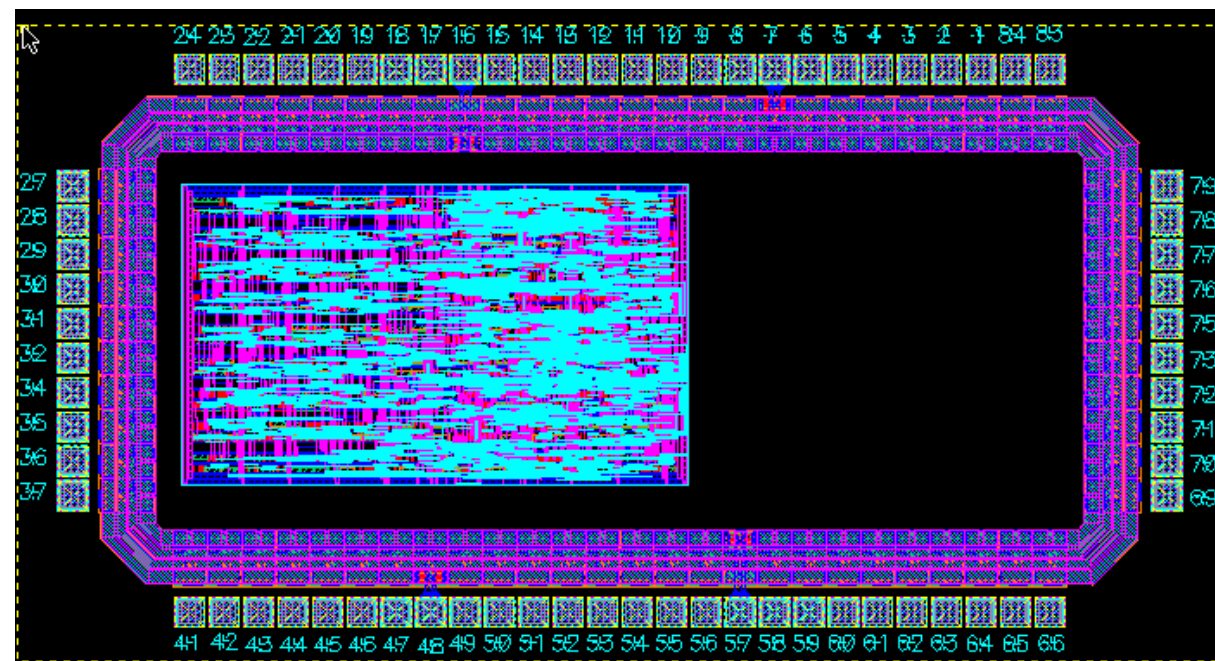
```
        ADR(adr),
```

```
        DI(writedata),
```

```
        DOUT(memdata));
```

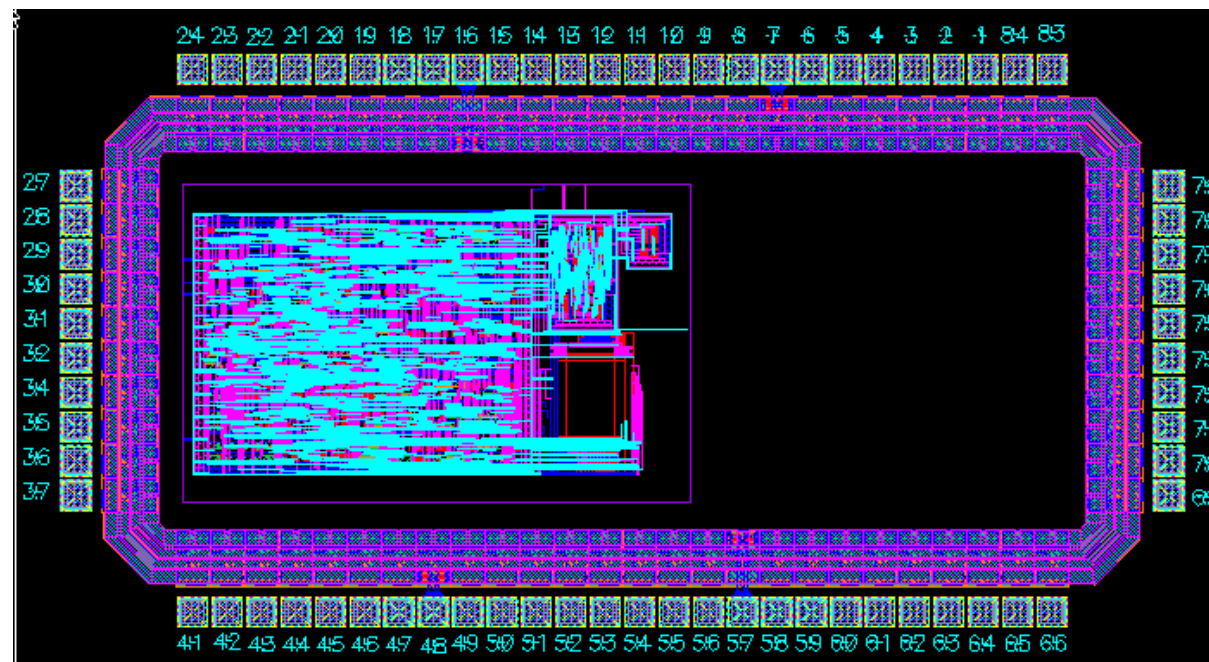
```
endmodule
```

MIPS (8-bit) size comparison



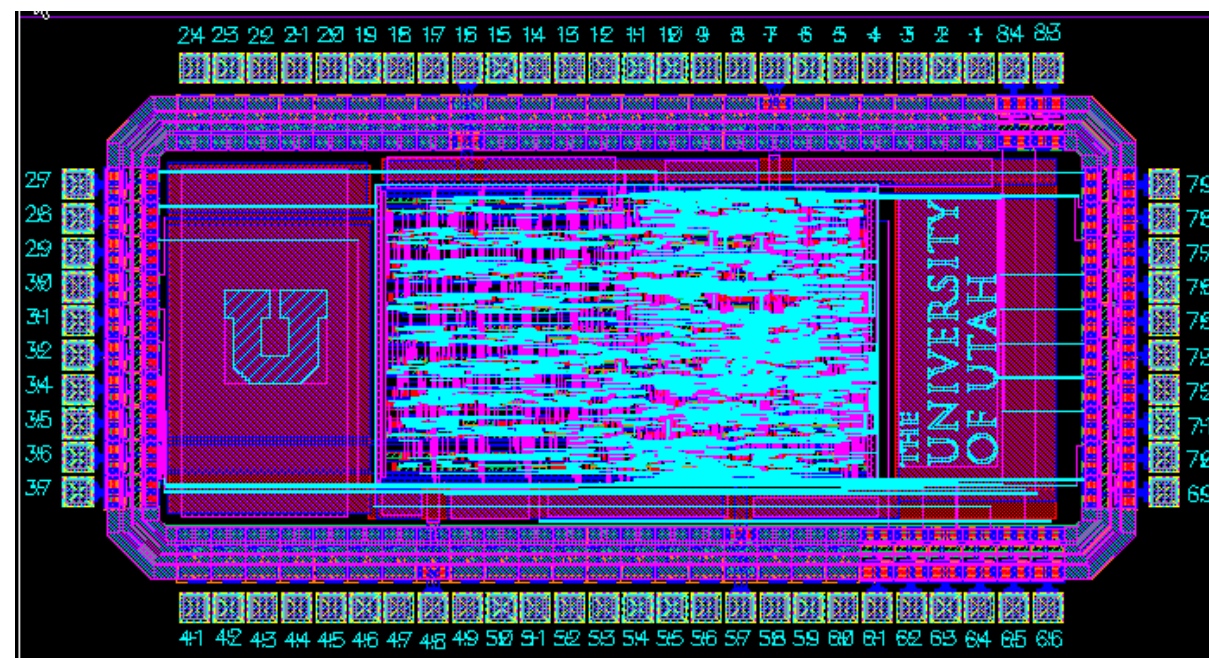
One big EDI run of the whole thing
With some work, could probably get this in a single TCU...

MIPS (8-bit) size comparison



Separate EDI for controller, alucontrol, datapath
and custom RF, assembled with ccar

MIPS (8-bit) whole chip



Includes poly/m1/m2/m3 fill, and logos
Routed to the pads using ccar