Overview

•The overall design, `mips`, has four modules implementing the five logical pipeline stages: `ifetch`, `id`, `ex`, `mem`.

•The fifth stage, `wb`, has been folded into the register fetch operations of the `id` and so is not explicitly represented by a module.

•Additionally, there is a set of auxiliary modules used by the four stages.

•The operation of the pipeline is achieved by transferring the contents of the pipeline registers to the next stage at the clock edge.

•There are two memories, `imem` and `dmem`, as well as the general purpose registers, `regfile`.

•The clock period in the base design is 11 ticks.

The `ifetch` module

Inputs: `nextpc` from `MEM` in case of branch

Operation: fetch the instruction from the `imem`
increment the pc by 4 to reference the next instruction.

Outputs to instruction decode: `pc+4`, `ir`

Increments the PC
Fetches the instruction

```
module ifetch;
   wire [31:0] pc, pc4;
   wire [31:0] ir;
   cpa cpa_0( pc, 32'b100, pc4 );
   imem imem_0( pc, ir );
endmodule
```

cpa, imem

The increment / The offset

```
module cpa( a, b, r);
   input [31:0] a, b;
   output [31:0] r;
   reg [31:0] r;
   always @(a or b)
      r <= #10 a + b;
endmodule
```

Change state at the end of the cycle

Create a word address

```
module imem( raddress, rdata);
   input [31:0] raddress;
   output [31:0] rdata;
   reg [31:0] rdata;
   reg [31:0] a [0:63];
   always @(raddress)
      rdata <= #10 a[raddress>>2];
endmodule
```

The `id` module

Inputs: `pc+4, ir, windex, final`

Operation: decode the instruction
fetch the register values
store the register for the wb stage.

Outputs: pc+4 (passed as is)
`rdata1, rdata2` (generated values)
`offset, Rt, Rd` (fields from the instruction)
`ALUOp, RegDst, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, RegWrite` (generated control signals)

The id module

The instruction

```
wire [31:0] ir;
   wire [5:0]      Op = ir[31:26],
                   Funct = ir[5:0];
   wire [4:0]      Rs = ir [25:21],
                   Rt  = ir[20:16];
                   Rd = ir[15:11];
   wire [15:0]     Immed = ir[15:0],
   wire [25:0]     Ja = ir[25:0],
```

Sign extension

```
wire [31:0] offset = {{16{Immed[15]}},Immed};
```

The id module

```
always @(Op)
  begin
    RegDst   = (Op == 6'b000000);
    MemRead  = (Op == 6'b100011);
    MemWrite = (Op == 6'b101011);
    Branch   = (Op == 6'b000100);
    ALUSrc = MemRead | MemWrite;
    MemToReg = MemRead;
    RegWrite = RegDst | MemRead;
    ALUOp = {RegDst,Branch};
  end

regfile rf( mips.clk, mips.mem.RegWrite,
            Rs, Rt, mips.mem.windex,
            rdata1, rdata2, mips.mem.final );
```

Rtype (Rd)
load
store
beq
immediate used
register value from memory
write a register
set the ALUop

regfile

```
module regfile( clk, we2, index0, index1, index2, value0, value1, value2);
  input clk, we2;
  input [4:0] index0, index1, index2;
  output [31:0] value0, value1;
  reg [31:0] value0, value1, value2, rf [0:31];

  initial rf[0] = 0;

  always @(posedge clk)
    if ( we2 && index2 != 0 )
      rf[index2] <= #5 value2;

  always @(index0 or rf[index0])
    value0 <= #5 rf[index0];

  always @(index1 or rf[index1])
    value1 <= #5 rf[index1];

endmodule
```

The `id` module

Set the flush control signal if the **instruction is a branch** and the **condition is true**.

Do this in the **mem stage**.

```
always @(mips.mem.Branch or mips.mem.Zero)
   flush = mips.mem.Branch & mips.mem.Zero;
```

The id module

Determine the **destination register**.

Do the hazard control logic to determine if there is a data hazard in both the **ex** and **mem** stages. If there is, **stall**. **Load** is a special case

```
always @(mips.ex.RegDst or mips.ex.Rt or mips.ex.Rd or
        mips.ex.RegWrite or
        Op or Rs or Rt or
        mips.ex.RegWrite or
        mips.ex.windex)
    begin
        tmpindex = mips.ex.RegDst == 0) ? mips.ex.Rt : mips.ex.Rd;
        Stall = mips.ex.RegWrite & tmpindex != 0 &
                    (tmpindex == Rs |
                    (tmpindex == Rt & Op != 6'b100011)) |
                  mips.mem.RegWrite & mips.mem.windex != 0 &
                    (mips.mem.windex == Rs |
                    (mips.mem.windex == Rt & Op != 6'b100011));
    end
```

The ex module

Inputs: `pc+4, rdata1, rdata2, offset,`
`Rt, Rd,`
`ALUOp, RegDst, ALUSrc, Branch,`
`MemRead, MemWrite, MemToReg, RegWrite`

Operation: perform the operation, which might be computing an address or the branch PC
determine the destination register address

Outputs: `windex` (calculated control)
`branchpc` (calculated value)
`result, Zero` (calculated value & control)
`wdata, nextpc` (passed as is)
`MemRead, MemWrite, MemToReg` (passed as is)

The ex module

```
mux #(5) wmux_0( RegDst, Rt, Rd, windex );
cpa cpa_0( pc4, {offset[29:0],2'b00}, branchpc );
mux #(32) wmux_1( ALUSrc, rdata2, offset, rdataB );
alu alu_0( rdata1, rdataB, result, Zero, Overflow, ALUControl );

alucontrol alucontrol_0( offset[5:0], ALUOp, ALUControl );
```

Calculate **destination register**, **branch target address**, **ALU sourceB**, and **ALUControl**.

Do the arithmetic operation.

mux

```
module mux( cntl, in0, in1, out);
   parameter width=1;
   input  cntl;
   input  [width-1:0] in0, in1;
   output [width-1:0] out;
   reg [width-1:0] out;

   always @(cntl or in0 or in1)
     if      ( cntl == 1)
        out = in1;
     else if ( cntl == 0)
        out = in0;
     else
        out = 'bx;
endmodule
```

alucontrol

```
module alucontrol( f, a, Op);
    input [5:0] f;
    input [1:0] a;
    output [2:0] Op;
    reg [2:0] Op;
  always @(f or a)
    begin
        casex ({a,f})
            8'b00_xxxxxx:  Op = 3'b010;
            8'b01_xxxxxx:  Op = 3'b110;
            8'b10_xx0000:  Op = 3'b010;
            8'b10_xx0010:  Op = 3'b110;
            8'b10_xx0100:  Op = 3'b000;
            8'b10_xx0101:  Op = 3'b001;
            8'b10_xx1010:  Op = 3'b111;
            default:       Op = 3'bxxx;
        endcase
    end
endmodule
```

ALUOp func

ld/st: add
beq : subt

RType func: do the function

alu

```
module alu( a, b, r, z, v, aluop );
  input [31:0] a, b;
  output [31:0] r;
  reg [31:0] r;
  output z, v;
  reg z, v, CarryOut;
  input [2:0] aluop;

  always @(aluop or a or b)
  begin
    case (aluop)
        3'b000: {CarryOut,r} <=  #10 {1'b0,a&b};
        3'b001: {CarryOut,r} <=  #10 {1'b0,a|b};
        3'b010: {CarryOut,r} <=  #10 a+b;
        3'b110: {CarryOut,r} <=  #10 a-b;
        3'b111: begin :slt
            integer ia, ib;
            ia = a; ib = b;
            {CarryOut,r} <= #10 {32'b0,ia<ib};
                end
        default: {CarryOut,r} <= #10 33'bx;
    endcase
  end
```

alu

```
    always @(r)
        z = ~(| r);

    always @( CarryOut or r[31] or a[31] or b[31] )
        v = CarryOut ^ r[31] ^ a[31] ^ b[31];

end module
```

The MEM module

```
Inputs:   result, Zero,
          wdata,
          windex,
          branchpc, nextpc,
          MemRead, MemWrite, MemToReg
```

Operation: fetch data on a read or store data on a write
decide the next instruction's address
set up to do the register write-back

Outputs: `windex, final, nextpc`

The MEM module

```
dmem dmem_0( mips.clk, MemRead, MemWrite, result, result, rdata, wdata );

mux #(32) mux_0( Branch & Zero, mips.ifetch.pc4, branchpc, nextpc );

mux #(32) wmux_0( MemToReg, result, rdata, final );
```

Reading from or writing to memory

Choosing between incremented PC and target address

Choosing between ALU calculated result & loaded data

dmem

```
module dmem( clk, MemRead, MemWrite, raddress, waddress,
                 rdata, wdata );
    input clk, MemRead, MemWrite;
    input [31:0] raddress, waddress, wdata;
    output [31:0] rdata;
    reg [31:0] rdata;
    reg [31:0] a [0:63];
    reg [31:0] wdata_delayed, waddress_delayed;

    always @(raddress or a[raddress>>2])
        rdata <= #10 a[raddress>>2];

    always @(wdata)
        wdata_delayed <= #10 wdata;

    always @(waddress)
        waddress_delayed <= #10 waddress;

    always @(posedge clk)
        if ( MemWrite )
            a[waddress_delayed>>2] = wdata_delayed;
endmodule
```

mux

```
module mux( cntl, in0, in1, out);
   parameter width=1;
   input  cntl;
   input  [width-1:0] in0, in1;
   output [width-1:0] out;
   reg [width-1:0] out;

   always @(cntl or in0 or in1)
      if      ( cntl == 1)
         out = in1;
      else if ( cntl == 0)
         out = in0;
      else
         out = 'bx;
endmodule
```