A Verilog Behavioral Simulation

The MIPS pipeline, as explained in Hennessy and Patterson is a modest Verilog program.

Understanding its operation will illustrate the timing and data motion in the pipeline.

© 1996 LS All Rights Reserved

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

© 1996 LS All Rights Reserved

The `ifetch` module

Tasks are to fetch the instruction from the `imem`, and increment the `pc` by 4 to reference the next instruction.

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

Operation: Increment `pc`

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

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


© 1996 LS All Rights Reserved

cpa, imem

```
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
```

```
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
```


© 1996 LS All Rights Reserved

The `id` module

- Tasks are to decode instruction, fetch the register values, store the register for the `wb` stage.
- Inputs: `pc+4, ir, windex, final`
- Operation: decode, fetch registers, load fields
- Outputs: `pc+4, rdata1, rdata2, offset, Rt, Rd, ALUOp RegDst, ALUSrc, Branch, MemRead, MemWrite, MemtoReg, RegWrite`
-

```
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);
```


© 1998 LS All Rights Reserved

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;
  reg [31:0] 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
```


© 1998 LS All Rights Reserved

The `ex` module

Tasks are to perform the operation, which might be computing an address, and to compute the branch address.

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

Operation: Compute operation, branch pc, figure write register addr

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

```
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);
```


© 1998 LS All Rights Reserved

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
```


© 1998 LS All Rights Reserved

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
```


© 1998 LS All Rights Reserved

The MEM module

Tasks are to fetch data on a read or store data on a write, figure the next instruction's address, and set up to do the write-back.

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

Operation:

Outputs: `windex, final, nextpc`

```
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);
```


© 1998 LS All Rights Reserved

mux, 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
```

```
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
```


© 1998 LS All Rights Reserved