Superscalar Instruction Dispatch
With Exception Handling

Design and Testing Report

Joseph McMahan

December 2013
Contents

1 Design 3
  1.1 Overview ................................................. 3
    1.1.1 The Tomasulo Algorithm ............................... 4
    1.1.2 Two-Phase Clocking .................................. 4
    1.1.3 Resolving Data Hazards ................................ 5
    1.1.4 Software Interface .................................... 7
  1.2 High Level Design .......................................... 10
  1.3 Low Level Design ........................................... 12
    1.3.1 Buses ............................................... 12
    1.3.2 Functional Units ...................................... 13
    1.3.3 Completion File ....................................... 14
    1.3.4 Bus Cycles ........................................... 17
    1.3.5 Instruction Decode and Dispatch ........................ 18

2 Testing 23
  2.1 Tests .................................................. 23
    2.1.1 Load Data, Add ....................................... 23
    2.1.2 Resource Hazard: Only One Station Available .......... 24
    2.1.3 Multiply .............................................. 26
    2.1.4 Resource Hazard: Two Instructions of Same Type ....... 26
    2.1.5 Data Hazard: RAW ..................................... 29
    2.1.6 Data Hazard: WAW ..................................... 31
    2.1.7 Data Hazard: WAR ..................................... 31
    2.1.8 Memory Hazard: WAR ................................... 35
    2.1.9 Resource Hazard: No Stations Available ................ 35
    2.1.10 Stall on Full Completion File .......................... 38
    2.1.11 Handle Exception .................................... 40
## 3 Code

<table>
<thead>
<tr>
<th>3.1 Modules</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1 rstation.v</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2 reg.v</td>
<td>49</td>
</tr>
<tr>
<td>3.1.3 fetch.v</td>
<td>53</td>
</tr>
<tr>
<td>3.1.4 store.v</td>
<td>59</td>
</tr>
<tr>
<td>3.1.5 mem-unit.v</td>
<td>63</td>
</tr>
<tr>
<td>3.1.6 ID.v</td>
<td>70</td>
</tr>
<tr>
<td>3.1.7 alu.v</td>
<td>84</td>
</tr>
<tr>
<td>3.1.8 alu-unit.v</td>
<td>86</td>
</tr>
<tr>
<td>3.1.9 mult.v</td>
<td>90</td>
</tr>
<tr>
<td>3.1.10 mp-unit.v</td>
<td>92</td>
</tr>
<tr>
<td>3.1.11 cf.v</td>
<td>96</td>
</tr>
<tr>
<td>3.1.12 IMem.v</td>
<td>107</td>
</tr>
<tr>
<td>3.1.13 cpu.v</td>
<td>110</td>
</tr>
<tr>
<td>3.1.14 assemble.py</td>
<td>114</td>
</tr>
</tbody>
</table>
Chapter 1

Design

1.1 Overview

This report covers the design, implementation, and testing of a basic superscalar CPU with exception handling. The CPU has a peak throughput of two instructions per cycle.

The design uses the Tomasulo algorithm for parallel execution (see section 1.1.1). This calls for an instruction dispatch that dynamically allocates resources and allows for localized control logic. There is no state machine governing the CPU actions; each functional unit operates under its own control. As long as the required resources are available, the instruction decode unit will dispatch two instructions every cycle, allowing the functional units to handle the execution.

Naturally, the instruction decode will have to resolve the data hazards that arise in dispatching two instructions every cycle. The process for resolving Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR) hazards for both the register names and memory accesses is covered in section 1.1.3.

The functional units consist of an adder, a multiplier, a fetch unit, and a store unit. A completion file is used to track all instructions that have been issued but not yet retired into the retirement registers. Section 1.3.3 covers the implementation and purpose of the completion file in the system.
writes are issued to the memory hierarchy.

WAW

The problem of subsequent writes at the same address is handled by the memory interface always issuing writes in sequential order.

RAW

When a read command is issued by the instruction dispatch and goes out on an instruction bus, the allocated fetch reservation station checks if the requested address exists in any of the store stations (meaning it belongs to a pending write operation). If there are no matches, it latches the address from the instruction bus and signals to the memory interface that it has a read request ready for the cache. If there is a match, it instead latches the data directly from the store unit, circumventing the memory access entirely.

This method not only solves the issues that might arise when one gives strict priority to reads over writes, it actually eliminates the delay for memory access in situations with close read-after-write operations.

1.1.4 Software Interface

<table>
<thead>
<tr>
<th>op</th>
<th>src/dst</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 1.1: Number of bits devoted to each field for a load or store command.

<table>
<thead>
<tr>
<th>op</th>
<th>dst</th>
<th>src1</th>
<th>src2</th>
<th>(empty)</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 1.2: Number of bits devoted to each field for all ALU and multiply commands.

The CPU takes 32-bit machine instructions. Tables 1.1.4 and 1.1.4 show how the 32-bit instructions are divided for load/store instructions (1.1.4) and all other instructions (1.1.4).
<table>
<thead>
<tr>
<th>operation</th>
<th>assembly</th>
<th>type</th>
<th>resource</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>NOR</td>
<td>nor</td>
<td>0</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>shift left logical</td>
<td>sll</td>
<td>0</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>shift right logical</td>
<td>srl</td>
<td>0</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>multiply</td>
<td>mult</td>
<td>0</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>load lower immediate</td>
<td>li</td>
<td>0</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>lui</td>
<td>0</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>store word</td>
<td>sw</td>
<td>1</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>load word</td>
<td>lw</td>
<td>1</td>
<td>1</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 1.3: Instruction opcodes. Note that load and store instructions use 2-bit opcodes, while all others use 5-bit opcodes.

Load and store commands use a separate instruction format from all other commands; these commands use a 2-bit opcode, a 2-bit register number, and a 28-bit word address. (The word address is later padded with two zeros to form a 30-bit byte address.)

All other instructions use a 5-bit opcode, 2-bits each for a destination register and two source registers, and a 16-bit immediate field. The immediate field allows for values specified in a program to be loaded into registers.

Table 1.1.4 shows how all opcodes are divided. The first bit indicates the instruction type: memory or non-memory. Variable-length opcodes are used and the first bit indicates how many bits to expect for the instruction. 1 is used for memory operations, 0 for all others. For memory operations, the second bit indicates a load or a store operation.

For non-memory operations (all instructions except load-word and store-word), the remaining four bits give the rest of the instruction. The second and third bit make up the resource identifier; an instruction can possibly use the ALU, the multiplier, or the register file. A designated field for resource type makes it easier for the instruction decode to know if a resource hazard exists between two instructions. The final two bits give the function code, which specifies which operation the functional unit is to perform. The multiplier has only one possible operation. The register file has two: load lower immediate (load the immediate field into the lower 16 bits) and load
1.2 High Level Design

The design of the CPU is broken into three major components: the instruction decode/dispatch unit (ID), the completion file (CF), and a set of functional units (FU). The system contains two instruction buses and five data buses. Each bus has a single owner with write privileges. Figure 1.2 shows the blocks and buses in the high-level design.

The two instruction buses are output by the ID. At peak throughput, each has a new instruction every cycle. Each register in the RF monitors the “dst” (destination) field of the instruction buses; if a register finds that its tag matches an instruction’s destination field, then it latches the allocated station for that instruction, which tells it what data to wait for. When data shows up with that tag attached, the register latches it. The registers also monitor the “src” (source) fields of each instruction. If a tag match is found (i.e., a register is used as an operand), the register signals to the RF controller that it needs to output its contents onto a bus.

Every functional unit monitors the instruction buses. The reservation stations in each function unit compare their tags to the tag in the “allocated station” field of each instruction. If it matches, the station latches the two source tags (which tell it what data to wait for) and the function tag. When all operands have been acquired, the functional unit executes the instruction, and puts it out on a data bus.

The ALU, multiplier, and fetch unit all have ownership of one data bus. The completion file has two (since multiple registers frequently need to output their contents at the same time). The store unit never has data to output, as it simple sends store requests to the memory controller.

Every functional unit (as well as the RF and ID) monitor the tags on all five data buses. Every bus is taken as an input into every block in the design.

Localized control in each FU handles how to execute each instruction, meaning that the system-wide control is minimal. No state machine is used for the CPU. The ID dispatches instructions every cycle, and the functional units handle the execution.
Figure 1.1: High-level design of the superscalar CPU. Only the connections controlling bus contents are drawn here; however, every unit in the design takes as input every bus.
1.3.2 Functional Units

The CPU has four functional units: a four-function ALU, a multiplier, a fetch unit (for memory reads), and a store unit (for memory writes).

At the head of each functional unit is a reservation station of two entries. This station is connected to all data and instruction buses. The station reads from the instruction bus on the positive clock edge. If the “allocated station” field on one of the instruction buses matches the station’s tag, then the station latches the tags for the two source operands. A given station slot can only be allocated on one of the two instruction buses; the instruction decode units have guards in place to prevent such resource allocation hazards.

When an instruction has been latched, the reservation station compares its two operand tags to the tag field on each instruction bus, every cycle. On a match, the station latches the corresponding data and marks the operand as ready. When both operands have been latched, it signals to the controller for the functional unit that its data is ready. The controller will then pass the data to the functional unit at
Figure 1.3: The Completion File tracks all instructions that have been dispatched, but not committed to the retirement registers. One pointer into the file marks that all entries behind the marker have been completed (the DONE marker). Another pointer tracks which station the next issued instruction should allocate. The two move through the CF in a circular fashion. If they ever overlap, it either indicates that the completion file is empty or full.
1.3.4 Bus Cycles

The two instruction buses run negative-clock-edge to negative-clock-clock edge. This means that all writes to the instructions buses occur on the negative edge. During the positive clock edge, while the two instruction decode units are preparing the next instructions to be issued, all of the functional units are reading from the instruction buses. Each instruction lives on the bus for one cycle; it is guaranteed to have fully stabilized by the rising clock edge, and will remain there for the full positive half-cycle. Figure 1.3.4 shows where IBus operations fall with respect to the clock.

The two data buses run positive-edge to positive-edge. The two bus types are intentionally set up out of phase to allow for the shortest possible clock cycle; that is, in a given half cycle, a unit in the design has to compare and potentially latch
Figure 1.5: Data bus cycles. The DBuses run positive-to-positive. Each word of data and its source tag stay on the bus for one cycle. All writes to a DBus happen on the positive edge, and all reads occur on the negative edge. The DBus and IBus cycles are out of phase to allow for quick reuse of tags without the danger of overlapping usage. The ID monitors functional unit execution pipelines to know when a used tag is about to be released onto a data bus. The ID marks the tag as available one half cycle before it actually appears. This means the tag may be re-used in an instruction to be dispatched on the following negative edge, so the earliest a functional unit will know to look for the new usage of a tag is the first cycle after its previous usage was on the bus. The system is designed so that there is no down time in tag re-use, but also no danger of overlapping tag use.
Chapter 2

Testing

2.1 Tests

Each of the following test descriptions contains an assembly version of the instruction executed for the test. The actual instructions are in binary machine code, hard-coded into the instruction memory before the test execution.

2.1.1 Load Data, Add

# Load data in registers
li R0 250
li R1 1

# Two add operations
add R2 R0 R1
add R3 R0 R1

We first see the two instructions (in hexadecimal) in the instr1 and instr2 fields, which carry them from the instruction memory to the dispatch units, on the positive clock edge. One the following negative clock edge, the decoded instructions are both dispatched on the two instruction buses. Following that, on the next positive edge,
Figure 2.1: Two load-immediates followed by two adds.
Figure 2.6 shows the results of the second instruction appearing on DB1 a few cycles later, with the tag for A1. R1 latches the data and clears its waiting tag.

2.1.6 Data Hazard: WAW

\[
\text{add R1 R2 R2} \\
\text{add R1 R3 R3}
\]

Issuing two instructions that write to the same register, we must ensure that only the later value persists. Figure 2.7 illustrates the test. The first add is issued to station A0, the second one cycle later to A1. We see that R1 latches tag 0x05 (A0) as its waiting tag. When the second instruction is issued, R1 latches the tag 0x06 (A1), overwriting its previous tag. In the end, it holds the correct data value of 0x2BC0, which is R3 + R3.

The system correctly avoids WAW hazards by registers always taking the most recently issued values, even if previous values have not yet resolved. Notably, the system still executes both instructions, even though the results of the first are never used. This is desirable behavior; the unused instruction may cause an exception, which must be handled.

2.1.7 Data Hazard: WAR

\[
\text{sw R2 250} \\
\text{add R2 R0 R1}
\]

With a write-after-read hazard, the system must ensure that the earlier value of the data (before the write takes place) is used in the first operation. Here we issue a store word command to write the value of R2 out to memory, and then overwrite R2 with an add. Figure 2.8 shows the two instructions being dispatched simultaneously. We see that R2 takes the tag 0x06 (for A1), waiting for the results of the add.
Figure 2.9: A memory read is issued directly after a memory write at the same address.
Here we give three instructions that all require the same resource (the ALU). When no resources are available to dispatch an instruction, the ID should stall until it becomes available. In figure 2.10, we see a no-op and the first add loaded into the ID. The add instruction is decoded and dispatched. The next cycle, two more add instructions are loaded in. We see that only the first of two is dispatched on the instruction buses; the second ID stalls. The cycle after, the stalled instruction is now handled by the first ID unit (since it’s the next sequential instruction in program order). However, since the two ALU reservation station slots are both still occupied, both ID units are forced to stall for several cycles until one clears. We see that the loaded instruction stays in the ID unit, but is not dispatched until station A0 becomes available.

2.1.10 Stall on Full Completion File

```
# fill completion file
lw R0 1000
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
add R0 R1 R2
```

The goal now is to fill every completion file entry (the system uses a total of 8) to ensure that the instruction dispatch does, in fact, stall. The instruction sequence used consists of a load word and then eight adds. The pseudo-memory interface has been configured to take 200 cycles for the memory fetch (simulating misses in
Figure 2.10: When all stations of a needed type are unavailable, the dispatch unit stalls until one becomes available.
3.1.6 ID.v

ID.v

// Joseph McMahan, 2013

/*
Instruction Dispatch Unit for superscalar CPU. Dispatches a maximum of two
instructions per cycle.
*/
module ID(

    // Input Declarations
    input wire clk,
    input wire rst,
    input wire [31:0] instr1,
    input wire [31:0] instr2,
    input wire store1d,
    input wire store2d,
    input wire [4:0] alutag, // snoop output tags for each datapath element
    input wire [4:0] multtag,
    input wire [4:0] fetchtag,

    // Output Declarations
    output reg [38:0] IB1, // two instruction buses
    output reg [38:0] IB2,
    output reg [29:0] addr1,
    output reg [29:0] addr2,
    output reg [1:0] incr
);

//
// Internal Registers and Connections
//
70
```verilog
reg stationfree [7:0]; // table storing which res stations are free
    // 000 – A0, 001 – A1;
    // 010 – M0, 011 – M1;
    // 100 – F0, 101 – F1;
    // 110 – S0, 111 – S1;
reg stationfreenext [7:0];
reg [5:0] regwaiting [3:0]; // table storing what each reg is waiting for
    // 1 bit indiciates waiting or not; 5 bits indicate tag
reg [4:0] alloc1, alloc2, dst1, dst2, src11, src12, src21, src22;
reg [2:0] funct1, funct2;
reg [15:0] imm1, imm2;
reg dispatch1, dispatch2;
reg [29:0] addr1n, addr2n;

always @(*) begin

    // On Positive Clock

if (clk) begin
if (rst) begin
    stationfree [0] = 1;
    stationfree [1] = 1;
    stationfree [2] = 1;
    stationfree [3] = 1;
    stationfree [4] = 1;
    stationfree [5] = 1;
    stationfree [6] = 1;
    stationfree [7] = 1;
end
end

71
```
regwaiting[0] = 0;
regwaiting[1] = 0;
regwaiting[2] = 0;
regwaiting[3] = 0;
end

stationfreenext[0] <= 1;
stationfreenext[1] <= 1;
stationfreenext[2] <= 1;
stationfreenext[3] <= 1;
stationfreenext[4] <= 1;
stationfreenext[5] <= 1;
stationfreenext[6] <= 1;
stationfreenext[7] <= 1;

// Instructi on Dispatch Unit 1

if (instr1[31]) begin
  // fetch or store
  // Assign default signals
  alloc1 = 5’d0;
dst1 = 5’b0;
src11 = 5’b0;
src21 = 5’b0;
functl = 3’b0;
imml = 16’b0;
addrln = {instr1[27:0], 2’b00};

  case (instr1[30])

    0:
      begin
        // store word
        if (stationfree[3'b110] || stationfree[3'b111]) begin
          dispatch1 = 1;
        end
      end
if (stationfree[3'b110]) begin
    alloc1 = 5'd11;
    stationfreenext[3'b110] <= 0;
    src11 = {3'b000, instr1[29:28]} + 5'b1;
end
else if (stationfree[3'b111]) begin
    alloc1 = 5'd12;
    stationfreenext[3'b111] <= 0;
    src11 = {3'b000, instr1[29:28]} + 5'b1;
end

// handle data forwarding for register source
if (regwaiting[instr1[29:28]][5])
    src11 = regwaiting[instr1[29:28]][4:0];
else
    dispatch1 = 0;
end

1: begin

// load word
if (stationfree[3'b100] || stationfree[3'b101]) begin
    dispatch1 = 1;
    if (stationfree[3'b100]) begin
        alloc1 = 5'd9;
        stationfreenext[3'b100] <= 0;
        dst1 = {3'b000, instr1[29:28]} + 5'b1;
    end
    else if (stationfree[3'b101]) begin
        alloc1 = 5'd10;
        stationfreenext[3'b101] <= 0;
        dst1 = {3'b000, instr1[29:28]} + 5'b1;
    end
    else
        dispatch1 = 0;
end
end
endcase
end

// All other instructions
else begin

// Assign default signals
alloc1 = 5’d0;
dst1 = {3'b000, instr1[26:25]} + 5'b1;
src11 = {3'b000, instr1[24:23]} + 5'b1;
src12 = {3'b000, instr1[22:21]} + 5'b1;
funct1 = {1'b0, instr1[28:27]};
imm1 = instr1[15:0];
addr1n = 30'b0;

// handle data forwarding for register operands
if (regwaiting[instr1[24:23]][5])
  src11 = regwaiting[instr1[24:23]][4:0];
if (regwaiting[instr1[22:21]][5])
  src12 = regwaiting[instr1[22:21]][4:0];

// decide if instruction can be dispatched and
// allocate reservation station
case (instr1[30:29])

  2'b00: begin // ALU op
    if (stationfree[3'b000] || stationfree[3'b001]) begin
      dispatch1 = 1;
    end
    if (stationfree[3'b000]) begin
      alloc1 = 5’d5;
      stationfreenext[3'b000] <= 0;
      regwaiting[instr1[26:25]] <= {1'b1, alloc1};
    end
    else if (stationfree[3'b001]) begin
      alloc1 = 5’d6;
      stationfreenext[3'b001] <= 0;
      regwaiting[instr1[26:25]] <= {1'b1, alloc1};
  end
endcase
end

else

dispatch1 = 0;

end

2'b01: begin // mult op
if (stationfree[3'b010] || stationfree[3'b011]) begin
    dispatch1 = 1;
if (stationfree[3'b010]) begin
    alloc1 = 5'd7;
    stationfreenext[3'b010] <= 0;
    regwaiting[instr1[26:25]] <= {1'b1, alloc1};
end
else if (stationfree[3'b011]) begin
    alloc1 = 5'd8;
    stationfreenext[3'b011] <= 0;
    regwaiting[instr1[26:25]] <= {1'b1, alloc1};
end
else dispatch1 = 0;
end

2'b00: begin // register op
    dispatch1 = 1;
alloc1 = 5'b0;
    regwaiting[instr1[26:25]] <= {1'b0, 5'b0};
end

2'b11: begin // no-op
    dst1 = {3'b000, instr1[26:25]};
    src11 = {3'b000, instr1[24:23]};
    src12 = {3'b000, instr1[22:21]};
    dispatch1 = 1;
end
if (instr2[31]) begin
  // fetch or store
  // Assign default signals
  alloc2 = 5'd0;
dst2 = 5'b0;
src21 = 5'b0;
funct2 = 3'b0;
imm2 = 16'b0;
addr2n = {instr2[27:0], 2'b00};
case (instr2[30])

  0:
  // store word
  // If 1st instr is same, must have two stations free
  // always take second station (so 1st instr can take 1st)
  if (((instr1[31:30] != 2'b10) &&
    (stationfree[3'b110] || stationfree[3'b111])) ||
    (stationfree[3'b110] && stationfree[3'b111]))
    begin
    dispatch2 = 1;
    if (stationfree[3'b111]) begin
      alloc2 = 5'd12;
      stationfreenext[3'b111] <= 0;
      src21 = {3'b000, instr2[29:28]} + 5'b1;
    end
    else if (stationfree[3'b110]) begin
      alloc2 = 5'd11;
  endcase
end
stationfreenext[3'b110] <= 0;
src21 = {3'b000, instr2[29:28]} + 5'b1;
end

// handle data forwarding for register source
if (regwaiting[instr2[29:28]][5])
src21 = regwaiting[instr2[29:28]][4:0];
end
else

dispatch2 = 0;
end

1:
begin
// load word
// If 1st instr is same, must have two stations free
// always take second station (so 1st instr can take 1st)
if (((instr1[31:30] != 2'b11) &&
    (stationfree[3'b100] || stationfree[3'b101]) ||
    (stationfree[3'b100] && stationfree[3'b101]))
begin
    dispatch2 = 1;
    if (stationfree[3'b101])
    begin
        alloc2 = 5'd10;
        stationfreenext[3'b101] <= 0;
        dst2 = {3'b000, instr2[29:28]} + 5'b1;
    end
    else if (stationfree[3'b100])
    begin
        alloc2 = 5'd9;
        stationfreenext[3'b100] <= 0;
        dst2 = {3'b000, instr2[29:28]} + 5'b1;
    end
end
else
    dispatch2 = 1;
end
endcase
end

// All other instructions
else begin
  // Assign default signals
  alloc2 = 5'd0;
dst2 = {3'b000, instr2[26:25]} + 5'b1;
src21 = {3'b000, instr2[24:23]} + 5'b1;
src22 = {3'b000, instr2[22:21]} + 5'b1;
funct2 = {1'b0, instr2[28:27]};
imm2 = instr2[15:0];
addr2n = 30'b0;

  // handle data forwarding for register operands
  if (regwaiting[instr2[24:23]][5])
      src21 = regwaiting[instr2[24:23]][4:0];
  if (regwaiting[instr2[22:21]][5])
      src22 = regwaiting[instr2[22:21]][4:0];

  // decide if instruction can be dispatched and
  // allocate reservation station
  case (instr2[30:29])
    2'b00: begin
      // If 1st instr is same, must have two stations free
      if (((instr1[31:29] != 3'b000) &&
           (stationfree[3'b000] || stationfree[3'b001])) ||
           (stationfree[3'b000] && stationfree[3'b001]))
        begin
          dispatch2 = 1;
        end
      if (stationfree[3'b001])
        begin
          alloc2 = 5'd6;
          stationfreenext[3'b001] <= 0;
          regwaiting[instr2[26:25]] <= {1'b1, alloc2};
        end
    end
else if (stationfree[3'b000]) begin
    alloc2 = 5'd5;
    stationfreenext[3'b000] <= 0;
    regwaiting[instr2[26:25]] <= {1'b1, alloc2};
end
    end
else
    dispatch2 = 0;
end

2'b01: begin // mult op
    // If 1st instr is same, must have two stations free
    // always take second station (so 1st instr can take 1st)
    if (((instr[31:29] != 3'b001) &&
        (stationfree[3'b010] || stationfree[3'b011]) ||
        (stationfree[3'b010] && stationfree[3'b011]))
    begin
        dispatch2 = 1;
        if (stationfree[3'b011]) begin
            alloc2 = 5'd8;
            stationfreenext[3'b011] <= 0;
            regwaiting[instr2[26:25]] <= {1'b1, alloc2};
        end
        else if (stationfree[3'b010]) begin
            alloc2 = 5'd7;
            stationfreenext[3'b010] <= 0;
            regwaiting[instr2[26:25]] <= {1'b1, alloc2};
        end
        end
else
    dispatch2 = 0;
end

2'b10: begin // register op
    alloc2 = 5'b0;
    dispatch2 = 1;
    regwaiting[instr2[26:25]] <= {1'b0, 5'b0};
end

2'b11: begin // no-op
dst2 = {3'b000, instr2[26:25]};
src21 = {3'b000, instr2[24:23]};
src22 = {3'b000, instr2[22:21]};
dispatch2 = 1;
end

endcase

// Check for RAW hazards between instructions
// If dest of 1st equals a source of second, use allocated station
// as source operand instead
if (instr2[26:25] == instr1[24:23])
src21 = alloc1;
if (instr2[26:25] == instr1[22:21])
src22 = alloc1;
end

//----------------------------------
// On Negative Clock
//----------------------------------

if (~clk) begin

// On negative clock, output dispatched instructions onto buses
if (dispatch1)
  IB1 <= {alloc1, dst1, src11, src12, funct1, imm1};
else
  IB1 <= 39'b0;
if (dispatch1 && dispatch2)
  IB2 <= {alloc2, dst2, src21, src22, funct2, imm2};
else

end
IB2 <= 39'b0;

// output addresses
addr1 <= addr1n;
addr2 <= addr2n;

// Increment instruction memory
if (dispatch1 && dispatch2)
  incr = 2'd2;
else if (dispatch1 && ~dispatch2)
  incr = 2'd1;
else
  incr = 2'd0;

// Update station free table based on dispatched instructions
if (¬stationfreenext [0]) begin
  stationfree [0] <= 0;
end
if (¬stationfreenext [1]) begin
  stationfree [1] <= 0;
end
if (¬stationfreenext [2]) begin
  stationfree [2] <= 0;
end
if (¬stationfreenext [3]) begin
  stationfree [3] <= 0;
end
if (¬stationfreenext [4]) begin
  stationfree [4] <= 0;
end
if (¬stationfreenext [5]) begin
  stationfree [5] <= 0;
end
if (¬stationfreenext [6]) begin
  stationfree [6] <= 0;
end
if (¬stationfreenext [7]) begin

81
stationfree[7] <= 0;
end

//
// Update Internal Tables based on Data buses
//

// Mark reservations stations as free when their tag shows up
if (alutag == 5'd5) stationfree[3'b000] <= 1;
if (alutag == 5'd6) stationfree[3'b001] <= 1;
if (multtag == 5'd7) stationfree[3'b010] <= 1;
if (multtag == 5'd8) stationfree[3'b011] <= 1;
if (fetchtag == 5'd9) stationfree[3'b100] <= 1;
if (fetchtag == 5'd10) stationfree[3'b101] <= 1;
if (store1d) stationfree[3'b110] <= 1;
if (store2d) stationfree[3'b111] <= 1;

// If register is waiting on tag and tag shows up, clear waiting bit
if (regwaiting[2'b00][4:0] == alutag) regwaiting[2'b00][5] <= 0;
if (regwaiting[2'b01][4:0] == alutag) regwaiting[2'b01][5] <= 0;
if (regwaiting[2'b10][4:0] == alutag) regwaiting[2'b10][5] <= 0;
if (regwaiting[2'b11][4:0] == alutag) regwaiting[2'b11][5] <= 0;

if (regwaiting[2'b00][4:0] == multtag) regwaiting[2'b00][5] <= 0;
if (regwaiting[2'b01][4:0] == multtag) regwaiting[2'b01][5] <= 0;
if (regwaiting[2'b10][4:0] == multtag) regwaiting[2'b10][5] <= 0;
if (regwaiting[2'b11][4:0] == multtag) regwaiting[2'b11][5] <= 0;

if (regwaiting[2'b00][4:0] == fetchtag) regwaiting[2'b00][5] <= 0;
if (regwaiting[2'b01][4:0] == fetchtag) regwaiting[2'b01][5] <= 0;
if (regwaiting[2'b10][4:0] == fetchtag) regwaiting[2'b10][5] <= 0;
if (regwaiting[2'b11][4:0] == fetchtag) regwaiting[2'b11][5] <= 0;
end

end
endmodule