Nick Mikstas

COEN 210

Final Project

Spring 2017

1. Instruction Set Architecture

1.1 Instruction Formats

There is a total of 4 different types of instruction formats and they are listed below:

```
R instruction format:
op   rd  rs  rt  N/A
0000_000_000_000_XXX

I instruction format:
op    rd  rs  signed_lit
0000__000_000_000000

B/S instruction format:
op    rt  rs  signed_lit
0000__000_000_000000

J instruction format:
op      unsinged_lit
0000____000000000000
```

op – Opcode
rd – Destination register
rs – Source register 1
rt – Source register 2
signed_lit – Signed literal
unsigned_lit – Unsigned literal

The underscores are inserted into the instructions to make them more readable. There is a total of 8 registers in the register file. Register 0 is a general-purpose register and can be read and written like any other register.

1.2 Instruction Definitions

There is a total of 10 instructions (including NOP). They are as follows:

Opcode	Name	Description	Format
0000	NOP	No change in state	All zeros
0001	ADD	$rd = $rs + $rt	R format
0010	Addi	$rd = $rs + lit	I format
0011	SUB	$rd = $rs - $rt	R format
0100	SUBi	$rd = $rs - lit	I format
0101	LWr	$rd = mem($rs + $rt)	R format
0110	LW	$rd = mem($rs + lit)	I format
0111	SW	mem($rs + lit) = $rt	B/S format
1000	JMP	PC = unsigned_lit	J format
1001	BEQ	if $rs == $rt, PC = PC + 2*signed_lit	B/S format

2. Application

2.1 First Function

2.1.1 Memory and Register Description

```
////////////////////////////////////////////////////////////////////////
//  Accumulate all the values in an array.                            //
//                                                                    //
//  Data memory:                                                      //
//  0x00 = Array base address.                                        //
//  0x04 = Array length in elements.                                  //
//  0x0C = Start of array.                                            //
//  0x78 = 7 segment display - memory mapped I/O.                     //
//                                                                    //
//  Registers:                                                        //
//  r0 = Zero reference.                                              //
//  r1 = Array base address.                                          //
//  r2 = Array pointer.                                               //
//  r3 = Accumulator.                                                 //
//  r4 = Retreived value from memory.                                 //
////////////////////////////////////////////////////////////////////////
```

2.1.2 Machine/Assembly Code

```
//Address 0
0101_001_000_000_000 //LWr  $r1, $r0($r0) <- Get array start index.
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
//Address 8
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Subtract 4 to find start of last
word.
//Address 10
0011_011_011_011_000 //SUB  $r3, $r3, $r3 <- Zero out accumulator.

//AccumLoop:

//Address 12
0101_100_001_010_000 //LWr  $r4, $r2($r1) <- Get value from array.
//Address 14
0001_011_011_100_000 //ADD  $r3, $r3, $r4 <- Add value to the accumulator.
//Address 16
1001__010_000_000010 //BEQ  $r2, $r0, 2   <- Exit if last index reached.
//Address 18
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Decrement to next array index.
//Address 20
1000____000000001100 //JMP  AccumLoop     <- More work to do, loop.

//LoopFinished:

//Address 22
0010__010_000_011110 //ADDi $r2, $r0, 30  <- Load register with 30.
//Address 24
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 26
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Now contains address 120.
//Address 28
0111__011_010_000000 //SW   $r3, 0($r2)   <-Write results to 7 segment
display.

//Spinlock:

//Address 30
1000____000000011110 //JMP  Spinlock      <- Done.
```

2.2 Second Function

2.2.1 Memory and Register Description

```
////////////////////////////////////////////////////////////////////////////
//  Find the last occurrence of a value in an array.                      //
//                                                                        //
//  Data memory:                                                          //
//  0x00 = Array base address.                                            //
//  0x04 = Array length in elements.                                      //
//  0x08 = Element value to search for.                                   //
//  0x0C = Start of array.                                                //
//  0x78 = 7 segment display - memory mapped I/O.                         //
//                                                                        //
//  Registers:                                                            //
//  r0 = Zero reference.                                                  //
//  r1 = Array base address.                                              //
//  r2 = Array pointer.                                                   //
//  r3 = Element index.                                                   //
//  r4 = Retreived value from memory\comparison results.                  //
//  r5 = Value to find.                                                   //
//  r6 = Current array element number.                                    //
////////////////////////////////////////////////////////////////////////////
```

2.2.2 Machine/Assembly Code

```
//Address 0
0101_001_000_000_000 //LWr  $r1, $r0($r0) <- Get array start index.
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_110_010_000_000 //ADD $r6, $r2, $r0  <- Store number of array elements.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 8
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
//Address 10
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Sub 4 to find start of last word.
//Address 12
0011_011_011_011_000 //SUB  $r3, $r3, $r3 <- Zero element index.
//Address 14
0100__011_011_000001 //SUBi $r3, $r3, 1   <- Set index to -1 (not found).
//Address 16
0110__101_000_001000 //LW   $r5, 8($r0)   <- Get value to find.

//FindLoop:

//Address 18
0101_100_001_010_000 //LWr  $r4, $r2($r1) <- Get value from array.
//Address 20
0011_100_100_101_000 //SUB  $r4, $r4, $r5 <- Compare with value to find.
//Address 22
1001__100_000_000100 //BEQ  $r4, $r0, 4   <- Branch if match found.
//Address 24
1001__010_000_000100 //BEQ  $r2, $r0, 4   <- Exit if last index reached.


//Address 26
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Decrement to next array index.
//Address 28
0100__110_110_000001 //SUBi $r6, $r6, 1   <-Decrement current array element.
//Address 30
1000____000000010010 //JMP  FindLoop      <- More work to do, loop.

//MatchFound:

//Address 32
0100_011_110_000001 //SUBi $r3, $r6, 1    <- Match. Save index-1 (from 0).

//LoopFinished:
```

```
//Address 34
0010__010_000_011110 //ADDi $r2, $r0, 30  <- Load register with 30.
//Address 36
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 38
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Now contains address 120.
//Address 40
0111__011_010_000000 //SW   $r3, 0($r2)   <-Write results to 7 seg display.

//Spinlock:

//Address 42
1000____000000101110 //JMP  Spinlock      <- Done.
```

2.3 Third Function

2.3.1 Memory and Register Description

```
////////////////////////////////////////////////////////////////////////
//  Count the number of occurrences of a value in an array.           //
//                                                                    //
//  Data memory:                                                      //
//  0x00 = Array base address.                                        //
//  0x04 = Array length in elements.                                  //
//  0x08 = Element value to search for and count.                     //
//  0x0C = Start of array.                                            //
//  0x78 = 7 segment display - memory mapped I/O.                     //
//                                                                    //
//  Registers:                                                        //
//  r0 = Zero reference.                                              //
//  r1 = Array base address.                                          //
//  r2 = Array pointer.                                               //
//  r3 = Element counter.                                             //
//  r4 = Retreived value from memory\comparison results.              //
//  r5 = Value to find.                                               //
////////////////////////////////////////////////////////////////////////
```

2.3.2 Machine/Assembly Code

```
//Address 0
0101_001_000_000_000 //LWr  $r1, $r0($r0) <- Get array start index.
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
//Address 8
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Sub 4 to find start of last word.
//Address 10
0011_011_011_011_000 //SUB  $r3, $r3, $r3 <- Zero out counter.
//Address 12
0110__101_000_001000 //LW   $r5, 8($r0)   <- Get value to find.

//CountLoop:

//Address 14
0101_100_001_010_000 //LWr  $r4, $r2($r1) <- Get value from array.
//Address 16
0011_100_100_101_000 //SUB  $r4, $r4, $r5 <- Compare with value to find.

//Address 18
1001__100_000_000011 //BEQ  $r4, $r0, 3   <- Branch if match found.
//Address 20
1001__010_000_000100 //BEQ  $r2, $r0, 4   <- Exit if last index reached.
//Address 22
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Decrement to next array index.
//Address 24
1000____000000001110 //JMP  CountLoop     <- More work to do, loop.

//MatchFound:
```

```
//Address 26
0010__011_011_000001 //ADDi $r3, $r3, 1   <- Match found. Increment counter.

//Address 28
1000____000000010100 //JMP  Address 20    <- Jump to continue processing.

//LoopFinished:

//Address 30
0010__010_000_011110 //ADDi $r2, $r0, 30  <- Load register with 30.
//Address 32
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 34
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Now contains address 120.
//Address 36
0111__011_010_000000 //SW   $r3, 0($r2)   <-Write results to 7 seg display.

//Spinlock:

//Address 38
1000____000000100110 //JMP  Spinlock      <- Done.
```

2.4 Fourth Function

2.4.1 Memory and Register Description

```
////////////////////////////////////////////////////////////////////////
//  Create a new array with the reverse contents of the original.     //
//                                                                    //
//  Data memory:                                                      //
//  0x00 = First array base address.                                  //
//  0x04 = First array length in elements.                            //
//  0x08 = Second array base address.                                 //
//  0x0C = Start of first array.                                      //
//  0x3C = Start of second array.                                     //
//                                                                    //
//  Registers:                                                        //
//  r0 = Zero reference.                                              //
//  r1 = First array base address.                                    //
//  r2 = Pointer into first array.                                    //
//  r3 = Pointer into second array.                                   //
//  r4 = Retreived value from first array                             //
////////////////////////////////////////////////////////////////////////
```

2.4.2 Machine/Assembly Code

```
//Address 0
0101_001_000_000_000 //LWr  $r1, $r0($r0) <- Get first array base address.
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
//Address 8
0100__010_010_000100 //SUBi $r2, 4($r2)   <- Sub 4 to find start of last word.
//Address 10
0110__011_000_001000 //LW   $r3, 8($r0)   <- Get second array base address.

//CopyLoop:

//Address 12
0101_100_001_010_000 //LWr  $r4, $r2($r1) <- Get value from first array.
//Address 14
0111__100_011_000000 //SW   $r4, 0(r3)    <- Copy value into the second array.
//Address 16
1001__010_000_000011 //BEQ  $r2, $r0, 3   <- Exit if last index reached.
//Address 18
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Decrement first array index.
//Address 20
0010__011_011_000100 //ADDi $r3, $r3, 4   <- Increment second array index.
//Address 22
1000____000000001100 //JMP  AccumLoop     <- More work to do, loop.
```

```
//Spinlock:

//Address 24
1000____000000011000 //JMP  Spinlock      <- Done.
```

2.5 Fifth Function

2.5.1 Memory and Register Description

```
////////////////////////////////////////////////////////////////////////
//  Add a value to each element in an array.                          //
//                                                                    //
//  Data memory:                                                      //
//  0x00 = Array base address.                                        //
//  0x04 = Array length in elements.                                  //
//  0x08 = Value to add to array elements.                            //
//  0x0C = Start of array.                                            //
//                                                                    //
//  Registers:                                                        //
//  r0 = Zero reference.                                              //
//  r1 = Array base address.                                          //
//  r2 = Array pointer.                                               //
//  r3 = Value to add to element.                                     //
//  r4 = Retreived value from memory.                                 //
//  r5 = Start of array + array pointer.                              //
////////////////////////////////////////////////////////////////////////
```

2.5.2 Machine/Assembly Code

```
//Address 0
0101_001_000_000_000 //LWr  $r1, $r0($r0) <- Get array base address.
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
//Address 8
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Sub 4 to find start of last word.
//Address 10
0110__011_000_001000 //LW   $r3, 8($r0)   <- Get val to add to array elements.

//AddLoop:

//Address 12
0101_100_001_010_000 //LWr  $r4, $r2($r1) <- Get value from array.
//Address 14
0001_100_100_011_000 //ADD  $r4, $r4, $r3 <- Add value to array element.
//Address 16
0001_101_001_010_000 //ADD  $r5, $r1, $r2 <- Calc storage point for new value.
//Address 18
0111__100_101_000000 //SW   $r4, 0($r5)   <- Store new value back in array.
//Address 20
1001__010_000_000010 //BEQ  $r2, $r0, 2   <- Exit if last index reached.
//Address 22
0100__010_010_000100 //SUBi $r2, $r2, 4   <- Decrement to next array index.
//Address 24
1000____000000001100 //JMP  AccumLoop     <- More work to do, loop.

//Spinlock:


//Address 26
1000____000000011010 //JMP  Spinlock      <- Done.
```

3. Data Control

3.1 Changes to Datapath Hardware

Some minor changes were made to the datapath hardware to accommodate the compressed instruction set. Below are the changes made.

MUX for rd, rt swap with BEQ instruction: An additional MUX has been placed on the front end of the register file on the rt port. This MUX replaces the rt field with the rd field in the event a BEQ instruction is executed.

Unsigned extension module: Unsigned extension logic has been added to support the JMP instruction. The JMP instruction jumps to an absolute value in the address space. The address is limited to 12 bits. These 12 bits are in the instruction and are extended to 32 bits by the unsigned extension module and replaces the current program counter value.

3.2 Instruction Decode

3.2.1 Definitions of inputs

The inputs for the instruction decode is the 4-bit opcode. The valid opcodes are listed in the table in Section 1.2. Any invalid opcode is treated as a NOP.

3.2.2 Definitions of outputs

There is a total of 7 output control bits. The table below summarizes the control bits and their function:

Control Bit Name	Description
dat_mem	0 = Use ALU output in writeback stage. 1 = Use data memory output in writeback stage.
ld	0 = Do not update register file. 1 = Update register file.
str	0 = Do not update data memory. 1 = Update data memory.
brnch	0 = Branch instruction not issued. 1 = Branch instruction not issued.
jmp	0 = Jump instruction not issued. 1 = Jump instruction issued.
reg_lit	0 = Use register value as second operand. 1 = Use literal value as second operand.
add_sub	0 = Perform ALU subtraction. 1 = Perform ALU addition.

3.2.3 Logic equations for outputs as a function of inputs

The logic equations for the outputs are derived from a case structure within the control module. Below is the Verilog code used to implement the logic decoding:

```
`timescale 1ns / 1ps

module Control(
    input [3:0]opcode, //4-bit opcode.
    output dat_mem,    //Data memory read.
    output ld,         //Load register.
    output str,        //Data memory store.
    output brnch,      //Branch.
    output jmp,        //Jump.
    output reg_lit,    //Register/literal.
    output add_sub     //Addition/subtraction.
    );

    reg [6:0]control_out = 7'h00;

    assign {dat_mem, ld, str, brnch, jmp, reg_lit, add_sub} = control_out[6:0];

    always @(*) begin
        case(opcode)                //dm  ld  str br  jmp lit add
            4'b0001: control_out = 7'b0___1___0___0___0___0___1; //ADD  (Same as MIPS).
            4'b0010: control_out = 7'b0___1___0___0___0___1___1; //ADDi (Same as MIPS).
            4'b0011: control_out = 7'b0___1___0___0___0___0___0; //SUB  (Same as MIPS).
            4'b0100: control_out = 7'b0___1___0___0___0___1___0; //SUBi (Same as MIPS).
            4'b0101: control_out = 7'b1___1___0___0___0___0___1; //LWr  (Not MIPS...Bonus instruction! Useful for arrays).
            4'b0110: control_out = 7'b1___1___0___0___0___1___1; //LW   (Same as MIPS).
            4'b0111: control_out = 7'b0___0___1___0___0___1___1; //SW   (Same as MIPS).
            4'b1000: control_out = 7'b0___0___0___0___1___0___0; //JMP  (Similar to MIPS but upper bits are 0).
            4'b1001: control_out = 7'b0___0___0___1___0___1___0; //BEQ  (Same as MIPS but r1, r2 swapped for simplicity).
            default: control_out = 7'b0___0___0___0___0___0___0; //NOP  (Same as MIPS).
        endcase
    end
endmodule
```

3.3 Hazard Detection

Hazard detection is broken down into two parts: flush control and stall control.

3.3.1 Definitions of inputs

The following are tables of the input signals and their descriptions:

Stall Control	
Input Signal Name	Description
branch	Indicates the instruction in the decode stage is a BEQ instruction.
ID_EX_load	Indicates the instruction in the execute stage is a register loading instruction.
ID_EX_dat_mem	Indicates the instruction in the execute stage is a data memory reading instruction.
EX_MEM_dat_mem	Indicates the instruction in the memory stage is a data memory reading instruction.
IF_ID_r1	rs address of instruction in the decode stage.
IF_ID_r2	rt address of instruction in the decode stage.
ID_EX_r2	rt address of instruction in the execute stage.

Flush Control	
Input Signal Name	Description
branch	Indicates a branch is going to occur.
jump	Indicates a jump is going to occur.
stall	Indicates the pipeline is currently stalled.

3.3.2 Definitions of outputs

The following are tables of the output signals and their descriptions:

Stall Control	
Output Signal Name	Description
stall_out	Indicates a stall is in progress.
PC_en	Program counter enable signal.
IF_ID_write	IF_ID pipeline registers enable.

Flush Control	
Output Signal Name	Description
flush	Indicates a flush is in progress.

3.3.3 Logic equations for outputs as a function of inputs

Below is the Verilog code for the stall control module:

```
`timescale 1ns / 1ps

module Stall_Control(
    input branch,         //Branch instruction signal.
    input ID_EX_load,     //Execute stage load register signal.
    input ID_EX_dat_mem,  //Execute stage data memory signal.
    input EX_MEM_dat_mem, //Memory stage data memory signal.
    input [2:0]IF_ID_r1,  //Decode stage rs address.
    input [2:0]IF_ID_r2,  //Decode stage rt address.
    input [2:0]ID_EX_r2,  //Execute stage rt address.
    output stall_out,     //Stall signal out.
    output PC_en,         //Program counter enable.
    output IF_ID_write    //IF_ID piipeline register enable.
    );

    reg stall = 1'b0;

    assign stall_out = stall ? 1'b1 : 1'b0;
    assign PC_en = stall ? 1'b0 : 1'b1;
    assign IF_ID_write = stall ? 1'b0 : 1'b1;

    always @(*) begin
        //Branch stall.
        if(branch && (ID_EX_load || EX_MEM_dat_mem)) begin
            stall = 1'b1;
        end

        //Data memory read stall.
        else if(ID_EX_dat_mem && ((ID_EX_r2 == IF_ID_r1) || (ID_EX_r2 == IF_ID_r2))) begin
            stall = 1'b1;
        end
        else begin
            stall = 1'b0;
        end
    end
endmodule
```

Below is the Verilog code for the flush module:

```
`timescale 1ns / 1ps

module Flush_Control(
    input branch, //Branch indicator.
    input jump,   //Jump indicator.
    input stall,  //Stall indicator.
    output flush  //Flush signal.
    );

    //Only flush if not stalled.
    assign flush = stall ? 1'b0 : jump | branch;
endmodule
```

3.4 Forwarding

3.4.1 Definitions of inputs

The following is a table of the input signals and their descriptions:

Input Signal Name	Description
WB_load	Write back stage register load bit.
WB_dm	Write back stage data memory bit.
MEM_load	Memory stage register load bit.
MEM_store	Memory stage memory store bit.
EX_reg_lit	Execute stage literal or register selection bit.
EX_store	Execute stage memory store bit.
IF_ID_r1	Decode stage rs address.
IF_ID_r2	Decode stage rt address.
EX_r1	Execute stage rs address.
EX_r2	Execute stage rt address.
EX_r2_store	Execute stage rt address (pre-branch MUX).
WB_rd	Write back stage rd address.
MEM_r2_store	Memory stage rt address (pre-branch MUX).
]MEM_rd	Memory stage rd address.

3.4.2 Definitions of outputs

The following is a table of the input signals and their descriptions:

Output Signal Name	Description
a	ALU input 1 forwarding MUX control.
b	ALU input 2 forwarding MUX control.
c	Comparator input 1 forwarding MUX control.
d	Comparator input 2 forwarding MUX control.
e	Memory address input MUX control.
f	Memory data input MUX control.
rf_a	Register file output 1 MUX control.
rf_b	Register file output 2 MUX control.

3.4.3 Logic equations for outputs as a function of inputs

Below is the Verilog code for the forwarding module:

```
`timescale 1ns / 1ps

module Forwarding_Logic(
    input WB_load,              //Write back stage register load bit.
    input WB_dm,                //Write back stage data memory bit.
    input MEM_load,             //Memory stage register load bit.
    input MEM_store,            //Memory stage memory store bit.
    input EX_reg_lit,           //Literal or register selection bit.
    input EX_store,             //Execute stage memory store bit.
    input [2:0]IF_ID_r1,        //Decode stage rs address.
    input [2:0]IF_ID_r2,        //Decode stage rt address.
    input [2:0]EX_r1,           //Execute stage rs address.
    input [2:0]EX_r2,           //Execute stage rt address.
    input [2:0]EX_r2_store,     //Execute stage original rt address.
    input [2:0]WB_rd,           //Write back stage rd address.
    input [2:0]MEM_r2_store,    //Memory stage original rt address.
    input [2:0]MEM_rd,          //Memory stage rd address.
    output reg [1:0]a = 2'b00, //ALU input 1 forwarding MUX control.
    output reg [1:0]b = 2'b00, //ALU input 2 forwarding MUX control.
    output reg [1:0]c = 2'b00, //Comparator input 1 forwarding MUX control.
    output reg [1:0]d = 2'b00, //Comparator input 2 forwarding MUX control.
    output reg [1:0]e = 2'b00, //Memory address input MUX control.
    output reg f = 1'b0,        //Memory data input MUX control.
    output reg rf_a = 1'b0,     //Register file output 1 MUX control.
    output reg rf_b = 1'b0      //Register file output 2 MUX control.
    );

    always @(*) begin
        /**********Data memory stage register write hazard***********/
        //Forward r1 from memory stage.
        a[1] = (MEM_load && (MEM_rd == EX_r1)) ? 1'b1 : 1'b0;

        //Forward r2 from memory stage. !EX_reg_lit ensures no false hazards detected.
        b[1] = (MEM_load && !EX_reg_lit && (MEM_rd == EX_r2)) ? 1'b1 : 1'b0;

        /***********Write back stage register write hazard***********/
        //Forward r1 from write back stage.
        if(WB_load && !a[1] && (WB_rd == EX_r1)) begin
            a[0] = 1'b1;
        end
        else begin
            a[0] = 1'b0;
        end

        //Forward r2 from write back stage. !EX_reg_lit ensures no false hazards detected.
        if(WB_load && !EX_reg_lit && !b[1] && (WB_rd == EX_r2)) begin
            b[0] = 1'b1;
        end
        else begin
            b[0] = 1'b0;
        end
        /************Decode stage register write hazard************/
        //Forward WB data to output of register file ports.
        if(WB_load && (WB_rd == IF_ID_r1)) begin
            rf_a = 1'b1;
        end
        else begin
            rf_a = 1'b0;
        end
```

```
        if(WB_load && (WB_rd == IF_ID_r2)) begin
            rf_b = 1'b1;
        end
        else begin
            rf_b = 1'b0;
        end

        /***************Data memory stage branch hazard**************/
        //Forward r1 from memory stage.
        c[1] = (MEM_load && (MEM_rd == IF_ID_r1)) ? 1'b1 : 1'b0;

                //Forward r2 from memory stage.
        d[1] = (MEM_load && (MEM_rd == IF_ID_r2)) ? 1'b1 : 1'b0;

        /***************Write back stage branch hazard***************/
        //Forward r1 from write back stage.
        if(WB_load && !c[1] && (WB_rd == IF_ID_r1)) begin
            c[0] = 1'b1;
        end
        else begin
            c[0] = 1'b0;
        end

        //Forward r2 from write back stage.
        if(WB_load && !d[1] && (WB_rd == IF_ID_r2)) begin
            d[0] = 1'b1;
        end
        else begin
            d[0] = 1'b0;
        end

        /*****************Data memory store hazard*****************/
        //Forward from memory stage.
        e[1] = (MEM_load && (MEM_rd == EX_r2_store)) ? 1'b1 : 1'b0;

        /*****************Write back store hazard******************/
        //Forward from write back stage.
        if(WB_load && !e[1] && (WB_rd == EX_r2_store)) begin
            e[0] = 1'b1;
        end
        else begin
            e[0] = 1'b0;
        end


        /***********STR immediately following an LDR hazard**********/
        if(WB_dm && MEM_store && (MEM_r2_store == WB_rd)) begin
            f = 1'b1;
        end
        else begin
            f = 1'b0;
        end
    end
endmodule
```

4. Hazards

4.1 Control Hazards

4.1.1 Code from section 2 that must flush instructions due to a control hazard

Any time a branch is taken or a jump instruction is executed, a flush is performed.

The following two lines of code from the second example will cause a flush for each instruction if the branch is taken:

```
//Address 22
1001__100_000_000100 //BEQ  $r4, $r0, 4   <- Branch if match found.
//Address 24
1001__010_000_000100 //BEQ  $r2, $r0, 4   <- Exit if last index reached.
```

Also, at the end of every sample program in this project, the processor goes into an infinite loop with the execution of a jump instruction that jumps to itself. A flush is performed every time this instruction is called. Below is a sample infinite loop from example 2:

```
//Spinlock:

//Address 42
1000____000000101110 //JMP  Spinlock      <- Done.
```

4.1.2 How the hazard detection logic causes the flush and how many cycles are flushed

The flush control is simple and requires only a single line of Verilog code:

```
assign flush = stall ? 1'b0 : jump | branch;
```

The processor will flush if a branch instruction is in the decode stage and is going to branch OR a jump instruction is in the decode stage. Also, the flush will happen only after any current stalls have been resolved.

The flush will happen for only a single clock cycle per BEQ (if branch taken) or JMP instruction.

4.2 Data Hazard – Stall

4.2.1 Code that must stall due to a data hazard

The following code from example 3 must stall for a single cycle because of a data hazard:

```
//Address 2
0110__010_000_000100 //LW   $r2, 4($r0)   <- Get array length in words.
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
```

4.2.2 How the hazard detection logic causes the stall

A stall will only occur for two reasons: a branch instruction needs a value from the execute stage to evaluate equality between registers and a data memory read is followed by a register write instruction with a dependency on the value read.

Below is the Verilog code from the stall controller:

```
//Data memory read stall.
else if(ID_EX_dat_mem && ((ID_EX_r2 == IF_ID_r1) || (ID_EX_r2 == IF_ID_r2))) begin
    stall = 1'b1;
end
else begin
    stall = 1'b0;
end
```

The pipeline will stall if the instruction in the execute stage is a memory read instruction AND the destination address is the same as The second OR third operand in the following instruction.

4.3 Data Hazard – Forwarding

4.3.1 Code where data is forwarded

The following code from the third example will cause data to be forwarded:

```
//Address 4
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2.
//Address 6
0001_010_010_010_000 //ADD  $r2, $r2, $r2 <- *2, Array length in bytes.
```

4.3.2 How the forwarding logic allows this to happen

The forwarding logic is the most complicated logic in the processor and has many parts. The following is the Verilog code from the forwarding logic that is exercised by the assembly code above:

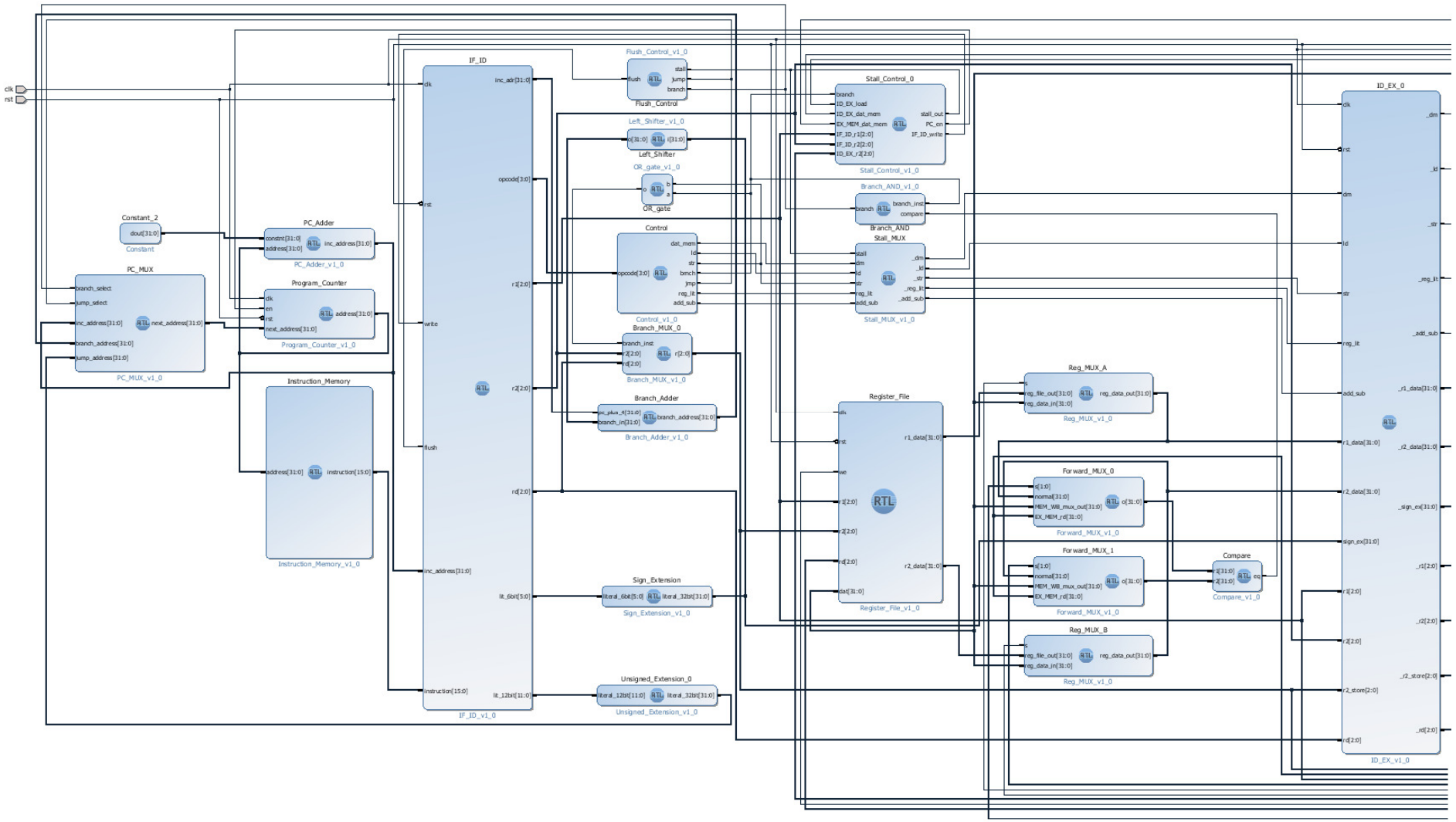
```
//Forward r1 from memory stage.
a[1] = (MEM_load && (MEM_rd == EX_r1)) ? 1'b1 : 1'b0;

//Forward r2 from memory stage. !EX_reg_lit ensures no false hazards detected.
b[1] = (MEM_load && !EX_reg_lit && (MEM_rd == EX_r2)) ? 1'b1 : 1'b0;
```

This is actually a dual data hazard as both the second and third operands need to be forwarded in the second instruction. Note in the second line of the Verilog code that the hardware is actively checking to make sure a literal value is not being used in the execution stage of the pipeline. This check is necessary as a false hazard can be detected if the literal pattern matches the destination register in the memory stage. The literal overlaps the rt register field because the instruction width is so small and bit fields need to be reused in different instruction formats.

5. Extras

5.1 Processor Block Diagram

Fetch/Decode Stages:



Execute/Memory/Write Back Stages: