CPE142: Computer Organization

December Fourth 2014

Term Project: Phase Two

Authors:
Ben Smith 55%
Devin Moore 45 %

Instructor:
Dr. Behnam Arad

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CSc/CPE 142 Fall 2014
Term Project Status Report

Complete this form by typing the requested information and include the completed form in your report after TOC. Gray cells will be filled by the instructor.

Name	*% Contribution*	*Grade*
Ben Smith	*50%*	
Devin Moore	*50%*	

Please do not write in the first table

Project Report/Presentation 20%	/200
Functionality of the individual components 40%	/400
Functionality of the overall design 25%	/250
Design Approach 5%	/50
Total points	/900

A: List all the instructions that were implemented correctly and verified by the assembly program on your system:

Instructions	State any issue regarding the instruction.
Signed addition	None
Signed subtraction	None
bitwise and	None
bitwise or	None
signed multiplication	None
signed division	None
Logical shift left	None
Logical shift right	None
rotate left	None
rotate right	None
load	None
store	None
branch on less than	None

Instructions	State any issue regarding the instruction.
branch on greater than	None
branch on equal	None
jump	None
Halt	None

B: Fill out the next table:

Individual Components	Does your system have this component	Does it work ?	List problems with the component, if any.
ALU	yes	yes	None
ALU control unit	yes	yes	None
Memory Unit	yes	yes	None
Register File	yes	yes	None
PC	yes	yes	None
IR	yes	yes	None
Other registers	yes	yes	None
Multiplexors	yes	yes	None
exception handler 1. Unknown opcode 2. Arith. Overflow	yes	yes	None, but the CPU does not transfer control to an ISR upon fault, in both cases the system halts.
Control Units 1. main 2. forwarding 3. lw hazard detection	yes	yes	None

How many stages do you have in your pipeline? Three

C: **State any issue regarding the overall operation of the datapath? Be Specific.**
CPU performs the expected instructions, including the full test program. Validations cases were generated for all instructions at the system level, this gives the team a high degree of confidence in functionality.

I. Introduction

THIS document details the design process of the CSUS CPE 142 Computer Organization course's term project. We have been asked to designed a pipelined datapath which implements an instruction set that is similar to MIPS in architecture. This project exercises a number of design principals from the course material, particularly design considerations for hazard detection and mitigation. This project started with several design specifications, the CPU had to be pipelined, hazards must be dealt with and the supported instruction set was given. Other than the mentioned guidelines the students were asked to make design decisions, the depth of the pipeline, which stage to put various components, and how to mitigate potential hazards.

This document will first introduce the instruction set as specified in the project specification. This will present an opportunity to begin the discussion about the components that will be required to implement the functionality described by the instructions. From this high level view of the architectue we will begin to look at the functionality of the individual components of the system and what functionality they perform. After the individual blocks are described the processes of connecting them together and the dangers that must be mitgated are discussed.

II. Instruction Set Architecture

INSTRUCTION set architecture describes the fundamental elements of a processor's ability to provide a service for software. This is also a sort of *contract* between hardware and software developers. As hardware developers we are saying this is what we promise to provide, our hardware can perform these operations for you. As the instruction set is the focus of the hardware we are designing, it makes sense to begin the design process with a through understanding of what hardware is to perform.

A. Supported Instruction Set Types

There are four instruction types in the prescribed instruction set, each of these types will support several different operations. Most instructions will add to the hardware that must be implemented as they ask for more functionality. Our processor will start with the most basic components, program memory, program counter and the hardware that's required to increment the program counter.

Instruction Format A: provides support for several arithmetic operations. All type A instructions carry an all zero opcode, the type of arithmetic operation is always decided by the four bit "funct code" field of the instruction. The organization of the instruction allows the func field to be supplied directly to the hardware which will perform the arithmetic without increasing the complexity of the main control logic. A full listing of supported hardware can be found in Table I.

This instruction type introduces a need for the first two components of this processor, the Arithmetic and Logic Unit, or ALU, and a register file for providing input and recording the output of the ALU.

4- bit opcode	4-bit operand 1	4-bit operand 2	4-bit funct code

Fig. 1: A Type Instruction Format

Instruction Format B: provides a way to load and store information from main memory. This greatly expands the capability of the ALU by removing the storage limitation of the register file. The new instructions require the implementation of some sort of addressable memory hardware to access. The two B type instructions, load word and store word, use indirect addressing schemes they will require the use of the ALU to calculate the physical address that is to be read or written to. The offset used in indirect addressing is signed and only 4 bits it necessitates new hardware to handle the sign extension out to the 16 bit width required by the ALU.

4- bit opcode	4-bit operand 1	4-bit operand 2	4-bit offset

Fig. 2: B Type Instruction Format

Instruction Format C: Allows the CPU to change the program counter based on logical outcomes. The instruction supplies an offset and a number to compare to a specific register, R0. The instructions jump when the instruction's operand 1 field is greater, equal, or less than R0 depending on the instruction. This comparison operation requires either the ALU or specialized hardware to provide these comparisons. There must also be additional hardware which will allow the instruction to effect the program counter. The jump range of these instructions is increased by shifting the 8-bit offset field left on the natural word boundary of memory. This can be done because all instructions are the same width.

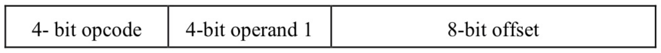


Fig. 3: C Type Instruction Format

Instruction Format D: allows the program counter to be set to almost anywhere in the program memory space. It does this by carrying a relatively large 13-bit effective jump offset. Although the opcode only allows for a 12-bit offset filed, the number is shifted to the left because instructions only start at even memory locations.

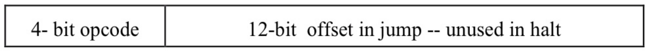


Fig. 4: D Type Instruction Format

III. Memory and Register Design

MEMORY is so crucial to the operation of the system it was the first system block to undergo design. The project has a few requirements with regard to memory. These requirements dictate how the memory can be accessed and it's total capacity. The register file will require several custom logic functions to allow the ALU access to a special register for divide and multiplication operations that produce 32 bits of output. The following subsections detail the high level functionality of our processor's memory organization at an abstract level.

A. Main Memory

The system is based on a 16 bit architecture, the memory will make full use of the addressing lines and provide 2^{16} total bytes of memory. The memory is byte addressable but will always return a 16 bit word, the byte at the address port and the following byte.

TABLE II
Main Memory Module Ports

Signal	Type	Operation
write_enable	logic	write data into memory at the next positive edge clock
write_address	logic[16]	the address data will be written to if write is to take place
write_data	logic[16]	the address data will be written to if write is to take place
data_out	logic[16]	the 16 bit word at location write_address will be made available

B. Program Memory

The program memory will be a combinatorial element which will output the instruction at a given address. There will be no synthasizable mechanism for loading this memory, it will be loaded by the system's testbench at simulation time.

TABLE III
Program Memory Module Ports

Signal	Type	Operation
in	logic[16]	address from program counter, memory will return content at the specified location
out	logic[16]	Instruction from address supplied at input port

C. Register File

The register file's basic function is to provide the contents of a register when an address is supplied to it's address port. The register file has has two address ports and two data ports. Data will be produced on the output ports as soon as it is ready, not waiting for a clock. The write procedure is sequential and the data will be written on the rising edge of the system clock. The register will also implement two custom functions based around the R0 register. R0 will be accessible through the register ports like all of the other registers, but in addition to this it will respond to R0_en and R0_read.

TABLE I
FULL SET OF SUPPORTED INSTRUCTIONS

Function	syntax	opcode	op1	op2	f. Code	type	Operation
Signed addition	add op1, op2	0000	reg	reg	1111	A	op1 = op1 + op2
Signed subtraction	sub op1, op2	0000	reg	reg	1110	A	op1 = op1 - op2
bitwise and	and op1, op2	0000	reg	reg	1101	A	op1 = op1 & op2
bitwise or	or op1, op2	0000	reg	reg	1100	A	op1 = op1 | op2
signed multiplication	mul op1, op2	0000	reg	reg	0001	A	op1 = op1 $*$ op2
							op1: Product (lower half)
							R0: Product (upper half)
signed division	div op1, op2	0000	reg	reg	0010	A	op1: 16-bit quotient
							R0: 16-bit remainder
Logical shift left	sll op1, op2	0000	reg	immd	1010	A	shift op1 to the left by op2 bits
Logical shift right	slr op1, op2	0000	reg	immd	1011	A	shift op1 to the right by op2 bits with sign extension
rotate left	rol op1, op2	0000	reg	immd	1000	A	rotate left op1 by op2 bits
rotate right	ror op1, op2	0000	reg	immd	1001	A	rotate right op1 by op2 bits
load	lw op1, immd (op2)	1000	reg	reg	N/A	B	op1 = Mem [immd + op2]
							(sign extend immd)
Store	sw op1, immd (op2)	1011	reg	reg	N/A	B	Mem [immd + op2] = op1
							(sign extend immd)
branch on less than	blt op1, op2	0100	reg	immd.	N/A	C	if (op1 $<$ R0) then
							PC = PC + op2
							(sign extend op2 & shift left)
branch on grater than	bgt op1, op2	0101	reg	immd.	N/A	C	if(op1 $>$ R0) then
							PC=PC+ op2
							(sign extend op2 & shift left)
branch on equal	beq op1, op2	0110	reg	immd.	N/A	C	if (op1 = R0) then
							PC = PC + op2
							(sign extend op2 & shift left)
jump	jmp op1	1100	off	——	N/A	D	pc = pc + op1
							(S.E. op1 and left shift)
halt	Halt	1111	—-	——	N/A	D	halt program execution

TABLE IV
REGISTER FILE CONTROL SIGNALS

Signal	type	Operation
RA1	logic[4]	read address for port 1
RA2	logic[4]	read address for port 2
RD1	logic[16]	the 16 bit word at location write_address will be made available
RD2	logic[16]	the 16 bit word at location write_address will be made available
write_enable	logic	when asserted data from write_data is captured on the falling edge of the clock
write_address	logic[4]	the address data will be written to if write is to take place
write_data	logic[16]	the data to be written at the positive edge of the clock

IV. DATA PATH ORGANIZATION

A. Number of Pipe Stages

The number of pipe stages the the primary design challenge of the first phase. A great deal of the difficulty surrounded assumptions that had to be made in the selection of the number of pipe stages. Pipelined designs are used to split combinational work across stages using flip flops to allow for higher global clock frequencies. We had to choose the number of pipe stages, guessing the longest path in the design. Given our understanding of digital logic we estimate that the ALU's signed divider circuit will require the most time by a wide margin. Because we do not intend to design a pipelined divider this operation is an atomic unit for us.

Because the ALU is assumed to require the longest time there is no logic between the inputs, outputs and the pipe flops ensuring highest possible operating frequency for the system. All of the control logic is implemented in the first stage and is assumed to require less time than the divisor circuit.

B. Hazard detection and mitigation

The hazard detection unit is used to detect and handle any potential hazards that may occur due to pipelining. With the current three stage design, its outputs will be controlling register forwarding and stalling branch instructions for one cycle when a hazard is detected. The most common hazard with this design is a data hazard. This occurs when an instruction is dependent on data from a previous instruction that has not yet been written back to the register file. When this occurs, the hazard detection unit will decide which control signals must be high in order to forward the correct data to where it will be used. These conditions can be found in Table VI

TABLE V
HAZARD DETECTION UNIT INPUTS

Input	Output is high when the following conditions are met
r0_en	This bit comes from the ALU control in the first stage. It is high for a MULTIPLY or DIVIDE instruction
instr[15:12]	This is the OPCODE from the first stage
S2.instr[15:12]	This is the OPCODE from the second stage
S3.instr[15:12]	This is the OPCODE from the third stage
instr[11:8]	This is R1, typically the destination register address for instruction in first stage
instr[7:4]	This is R2, typically the source register address for instruction in first stage
S2.instr[11:8]	This is R1 of the second stage, typically the destination register
S3.instr[11:8]	This is R1 of the third stage, typically the destination register

TABLE VI
HAZARD DETECTION UNIT CONTROL LOGIC

Signal	Output is high when the following conditions are met
haz0	Arithmetic or load followed two instructions later another arithmetic(Or STORE) using same destination register for R1.
haz1	Arithmetic or load directly followed by an arithmetic op with the R1 as the first destination.
haz2	Arithmetic or load directly followed by an arithmetic op with the R2 as the first destination.
haz3	Arithmetic or load followed 2 instructions later by an arithmetic op with the R2 as the first destination
haz4	An Arithmetic operation is followed directly by a branch instruction
haz5	LOAD is followed directly, or second instruction, by a branch instruction using the dest register for compare. Also if an arithmetic op was followed 2 instructions later by a branch instruction using it's dest register
haz6	Multiply or divide is followed directly by a branch instruction(What registers they specify does not matter. This is for R0 which is implicitly used by all 3 types)
haz7	Multiply or divide is followed 2 instructions later by a branch instruction(What registers they specify does not matter. This is for R0 which is implicitly used by all 3 types)
haz8	LOAD is followed directly by a STORE instruction using same reg for dest(load)/src(store)
haz9	LOAD is followed 2 instructions later by a STORE instruction using same reg for dest(load)/src(store)
haz10	Arithmetic instruction followed directly by a STORE instruction using same reg for dest/src
stall	LOAD is followed directly by a branch instruction using the dest register for compare

C. Stage 1

The first stage of this design contains nearly all of the control logic for processor. Since the path through the 16-bit signed divider of stage two is so long, a lot of control logic can be implemented without effecting the maximum frequency of the CPU. Much of this logic will be in parallel. The main control unit, the hazard detection unit, and most of the jump unit are all independent of each other and control separate signals. Most of the logic will be in the hazard detection unit, and will require inputs from all three stages before the outputs can drive anything.

1) Main Control Unit and Exception Handling: The main control unit is responsible for decoding the opcode of the current instruction and controlling the data path. The truth table for this logic can be found in Table VII. The exception handling logic is omitted from this table due to size and complexity.

Since the control unit is already handling the control signals for the halt operation, it also contains the logic to handle exceptions. There are three types of exceptions that are being handled; divide by zero, overflow, and unknown opcode.

The ALU will be in charge of detecting a divide by zero, or an overflow. If the operation is to be a 16 bit signed division, it will check the divisor for a 0 and assert a div0 signal there is is an attempt to divide by zero. If an overflow is detected, it will assert the overflow flag. Both of these signals are sent to the main control unit, where it will halt the system by gating all of the clock inputs to the flops. It will do the same for the halt instruction, or any opcode that is unknown.

2) Sign Extender: The sign extender in the first stage must be able to handle 4, 8, and 16 bit inputs from the different types of instructions. The main control unit will provide the control signals to let the sign extender know which bits to extend. The logic can be seen in Table X.

3) ALU Control Unit: The ALU control unit directly controls the operations of the ALU. It receives an ALUop bit from the main control unit to signal the use of the instruction's function code. If this bit is low, the ALU control signals will be determined by the function code, if it is high, the operation will be addition for the case of store and load instructions. For branching instructions, these signals don't matter because the main ALU results are not used. It would be more energy efficient to use another bit for those operations to completely shut off the ALU, but there are no energy constraints on our design.

There are 5 output signals from the ALU control unit, four of which are input signals to the main ALU that determine its operations. The fifth output bit, imm_b, is used to bring the immediate value from the instruction into the ALU for the shift and rotate functions. Since there are separate function for rotating left and right, there is no need to treat the immediate as a signed number and it is not sign extended.

4) Branching and Jump Control Unit: The jump control unit decides whether to take PC + 2 or PC + offset during a branch or jump instruction. This unit will be part of the critical path for this particular stage. Using the opcode input, it will determine what instruction is being performed and how to drive the jmp output based off the result from the comparator if necessary. The comparator results will be valid after the register file has been indexed, any hazards have been dealt with, and the register contents have propagated through the comparator. The truth table can be seen in Table IX.

All of the branching and jumping logic is handled within this first stage thanks to the large division path of the second stage. The comparator will output either a 00 for equal, 01 for R1 ¿ R0, and a 10 for R1 ¡ R0. The jump control unit will compare those results to the opcode to determine whether or not a branch will be taken. If the opcode is for JMP, it will assert the jmp control bit no matter what the comparator says.

TABLE X
SIGN EXTENTION LOGIC TABLE

Input		Output
offset_sel[1]	offset_sel[0]	Action
0	0	nothing
0	1	extend 4 bits to 16
1	0	extend 8 bits to 16
1	1	extend 12 bits to 16

D. Stage 2

The entire second stage of this design belongs to the Arithmetic Logic Unit as seen in Figure **??**. This ALU supports all of the operations listed in Table VIII. Its function is determined by the ALU_control in the first stage. The 16 bit signed division will be our longest combinatorial path between stages, so there is very little logic between the ALU and the flip-flops.

E. Stage 3

The third stage of this pipelined processor handles memory references and writes back to the register file as seen in Figure **??**. The main logic of this stage is contained in the main memory unit which is described in section two of this document.

TABLE VII
CONTROL LOGIC TRUTH TABLE

	Input				Output						
Instruction	instr[15]	instr[15]	instr[15]	instr[15]	ALUop	offset_sel[1:0]	mem2r	memwr	R0_read	reg_wr	se_imm_a
Type A	0	0	0	0	0	00	0	0	0	1	1
Load	1	0	0	0	1	10	1	0	0	1	0
store	1	0	1	1	1	10	0	1	0	0	0
BLT	0	1	0	0	0	10	0	0	1	0	1
BGT	0	1	0	1	0	10	0	0	1	0	1
BE	0	1	1	0	0	10	0	0	1	0	1
JMP	1	1	0	0	0	11	0	0	0	0	1
Halt	1	1	1	1	0	00	0	0	0	0	1

TABLE VIII
ALU CONTROL LOGIC TRUTH TABLE

Input					Output				
Instruction	ALUop	instr[3]	instr[2]	instr[1]	instr[0]	alu_ctrl[3]	alu_ctrl[2]	alu_ctrl[1]	alu_ctrl[0]	imm_b
SW/LW	1	X	X	X	X	0	0	0	0	0
Add	0	1	1	1	1	0	0	0	0	0
sub	0	1	1	1	0	0	0	0	1	0
AND	0	1	1	0	1	0	0	1	0	0
OR	0	1	1	0	0	0	0	1	1	0
MULT	0	0	0	0	1	0	1	0	0	0
DIV	0	0	0	1	0	0	1	0	1	0
SHL	0	1	0	1	0	0	1	1	0	1
SHR	0	1	0	1	1	0	1	1	1	1
ROL	0	1	0	0	0	1	0	0	0	1
ROR	0	1	0	0	1	1	0	0	1	1

TABLE IX
JUMP CONTROL LOGIC

	Input						Output
Instruction	instr[15]	instr[14]	instr[13]	instr[12]	cmp_result[1]	cmp_result[0]	jmp
BLT	0	1	0	0	0	0	0
BLT	0	1	0	0	0	1	1
BLT	0	1	0	0	1	0	0
BLT	0	1	0	0	1	1	0
BGT	0	1	0	1	0	0	1
BGT	0	1	0	1	0	1	0
BGT	0	1	0	1	1	0	0
BGT	0	1	0	1	1	1	0
BE	0	1	1	0	0	0	0
BE	0	1	1	0	0	1	0
BE	0	1	1	0	1	0	1
BE	0	1	1	0	1	1	0
JMP	1	1	0	0	0	0	1
JMP	1	1	0	0	0	1	1
JMP	1	1	0	0	1	0	1
JMP	1	1	0	0	1	1	1

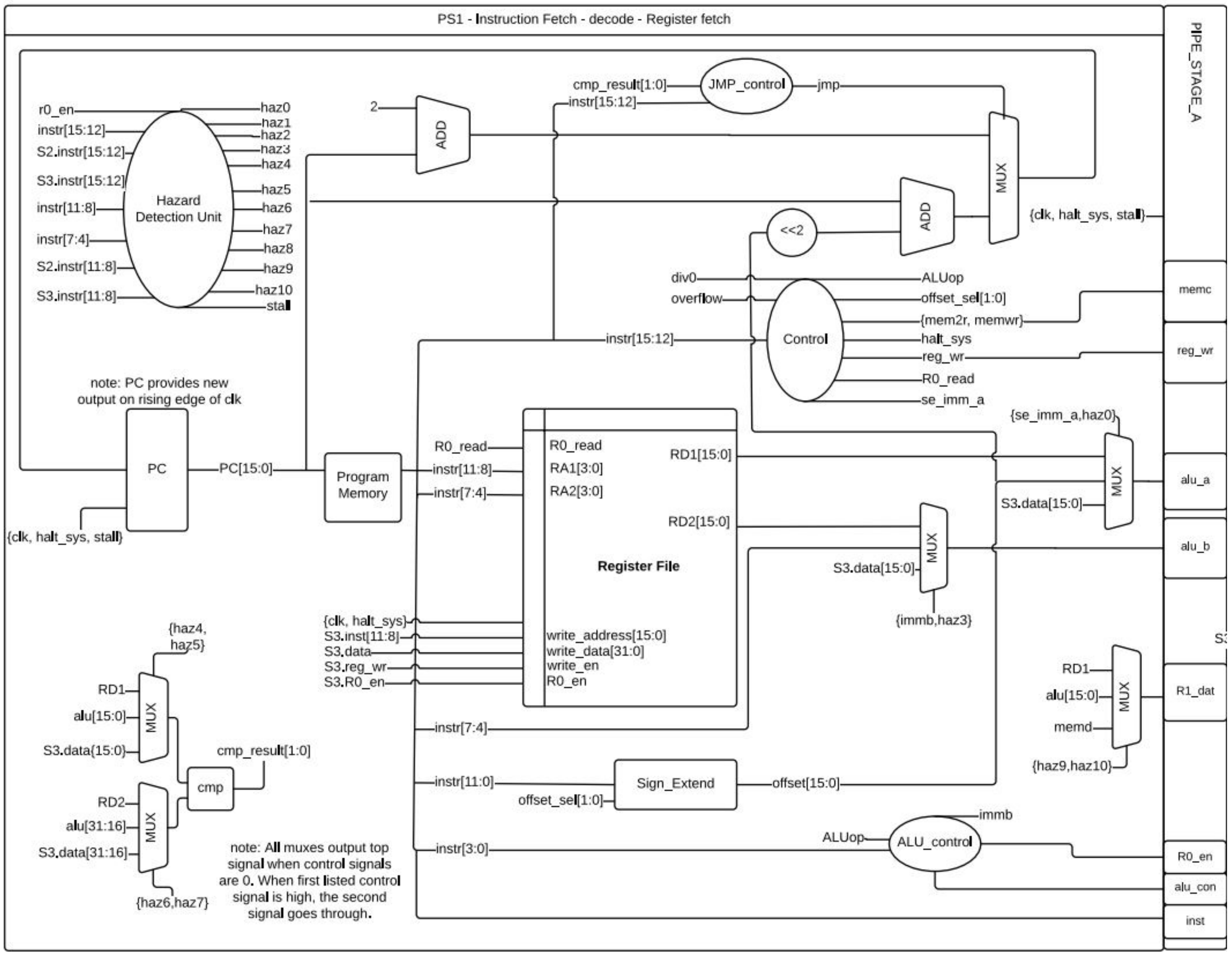
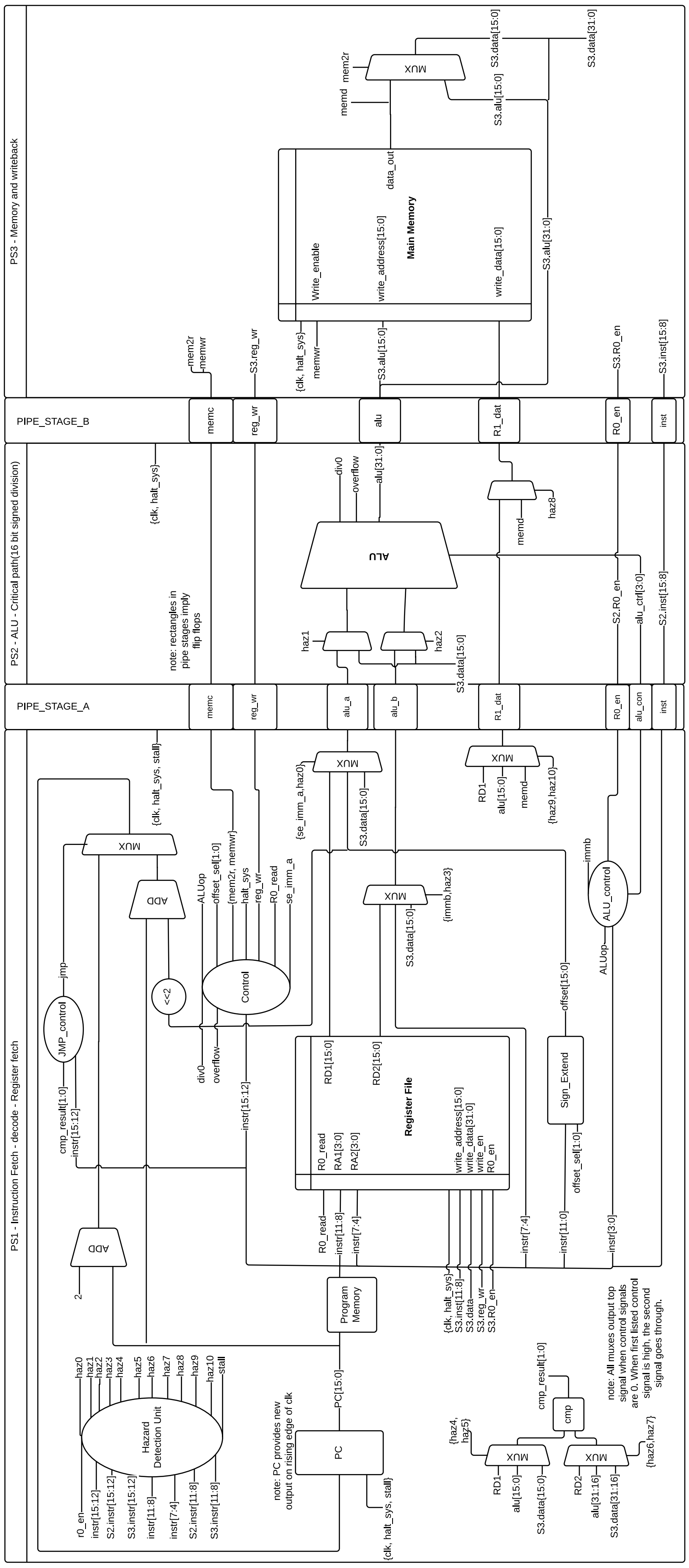


Fig. 5: Pipeline Organization Stage One


PS1 - Instruction Fetch - decode - Register fetch
PIPE_STAGE_A
PS2 - ALU - Critical path(16 bit signed division)
PIPE_STAGE_B
PS3 - Memory and writeback
PC
Program Memory
ADD
MUX
JMP_control
Control
<<2
Hazard Detection Unit
Register File
Sign_Extend
ALU_control
ALU
Main Memory
cmp
note: PC provides new output on rising edge of clk
note: rectangles in pipe stages imply flip flops
note: All muxes output top signal when control signals are 0. When first listed control signal is high, the second signal goes through.
memc
reg_wr
alu_a
alu_b
R1_dat
R0_en
alu_con
inst
alu
PC[15:0]
{clk, halt_sys, stall}
{clk, halt_sys}
cmp_result[1:0]
instr[15:12]
jmp
ALUop
offset_sel[1:0]
{mem2r, memwr}
halt_sys
reg_wr
R0_read
se_imm_a
div0
overflow
RD1[15:0]
RD2[15:0]
RA1[3:0]
RA2[3:0]
write_address[15:0]
write_data[31:0]
write_en
R0_en
instr[11:8]
instr[7:4]
instr[11:0]
instr[3:0]
offset[15:0]
immb
{immb,haz3}
{se_imm_a,haz0}
S3.data[15:0]
alu[15:0]
memd
{haz9,haz10}
haz1
haz2
haz8
alu[31:0]
S2.R0_en
alu_ctrl[3:0]
S2.inst[15:8]
mem2r
memwr
S3.reg_wr
S3.alu[15:0]
S3.alu[31:0]
S3.R0_en
S3.inst[15:8]
Write_enable
write_address[15:0]
write_data[15:0]
data_out
S3.data[31:0]
S3.inst[11:8]
S3.data
r0_en
S2.instr[15:12]
S3.instr[15:12]
S2.instr[11:8]
S3.instr[11:8]
haz0
haz4
haz5
haz6
haz7
haz9
haz10
stall
{haz4, haz5}
{haz6,haz7}
RD1
RD2
alu[31:16]
S3.data[31:16]
2

V. Design Source Code

A. Top Level

```
module top (
    input wire clk,
    input wire rst
);

    import types_pkg::*;
    import alu_pkg::*;

    //| Stage One
    memc_t          memc;

    reg             s1_R0_en;

    types_pkg::memc_t s1_memc;
    reg             s1_reg_wr;
    reg             halt_sys;
    reg             stall;
    in_t            s1_alu_inputs;
    uword           s1_R1_data;
    uword           s1_instruction;
    uword           s2_instruction;
    control_e       s1_alu_control;

    //| Stage Two
    memc_t          s2_memc;
    wire            s2_reg_wr;
    status_t        stat;

    //| stage 3
    wire            s3_reg_wr;
    memc_t          s3_memc;
    uword           s3_R1_data;
    uword           s3_instruction;
    wire    [31:0]  s3_data;
    reg             s3_R0_en;
    reg             s1_haz2;
    reg             s1_haz1;
    reg             s1_haz8;
    wire    [31:0]  s2_alu_result;
    uword           s2_R1_data;
    integer         s3_alu;


    stage_one st1(
        .clk(clk),
        .rst(rst),
        .s3_data(s3_data),
        .s3_instruction(s3_instruction),
        .s2_R0_en(s1_R0_en),
        .s3_R0_en(s3_R0_en),
        .s2_alu(s2_alu_result),
        .memc(memc),
        .div0(stat.div0),
        .overflow(stat.overflow),
        .s3_reg_wr(s3_reg_wr),
        .s3_mem2r(s3_memc.mem2r),

        //outputs
        .out_memc(s1_memc),
        .out_R1_data(s1_R1_data),
        .out_reg_wr(s2_reg_wr),
        .halt_sys(halt_sys),
        .stall(stall),
        .out_alu(s1_alu_inputs),

        .out_haz1(s1_haz1),
        .out_haz2(s1_haz2),
        .out_haz8(s1_haz8),
        .out_R0_en(s1_R0_en),
        .out_alu_ctrl(s1_alu_control),
        .out_instr(s1_instruction)
    );

    stage_two st2(
        .rst(rst),
        .clk(clk),

        .halt_sys(halt_sys),
        .stall(stall),

        .in_alu(s1_alu_inputs),
        .in_R1_data(s1_R1_data),
        .in_R0_en(s1_R0_en),
        .in_instr(s1_instruction),
        .in_memc(s1_memc),
        .in_reg_wr(s2_reg_wr),
        .haz1(s1_haz1),
        .haz2(s1_haz2),
        .haz8(s1_haz8),
        .s3_data(s3_data),
        .alu_control(s1_alu_control),

        .out_reg_wr(s3_reg_wr),
        .out_memc(s2_memc),
        .out_alu_result(s2_alu_result),  // for reg forwarding
        .out_alu(s3_alu),
        .out_alu_stat(stat),
        .out_R1_data(s2_R1_data),
        .out_R0_en(s3_R0_en),
        .out_instr(s2_instruction)
    );

    stage_three st3(
        .clk(clk),
        .rst(rst),
        .memc(s2_memc),
        .instruction(s2_instruction),
        .r1_data(s2_R1_data),

        .halt_sys(halt_sys),
        .alu(s3_alu),
        .out_memc(s3_memc),
        .r0_en(s3_R0_en),

        .instruction_out(s3_instruction),
        .out_r0_en(),
        .r1_data_out(s3_R1_data),
        .data(s3_data)
    );
endmodule
```

../source/Design/top.sv

B. Stage One

```
module alu(
    input alu_pkg::in_t       in,
    input alu_pkg::control_e  control,

    output alu_pkg::status_t   stat,
    output integer   out
);
    import alu_pkg::*;

    logic carry;
    logic signed [17:0] arith;

    always_comb begin
        case(control)
            OR  : out = {16'b0,in.a | in.b};
            AND : out = {16'b0,in.a & in.b};
            MULT: out = in.a * in.b;
            ROL : out = {16'b0,({in.a, in.a} << in.b%16)};
            ROR : out = {16'b0,({in.a, in.a} >> in.b%16)};
            SHL : out = {16'b0,in.a <<< in.b};
            SHR : out = {16'b0,in.a >>> in.b};
            SUB : begin
                arith = in.a - in.b;
                out = {16'b0, arith[15:0]};
            end
            ADD : begin
                arith = in.a + in.b;
                out = {16'b0, arith[15:0]};
            end
            DIV : begin
                if(in.b != 0) begin
                    out[15:0] = in.a / in.b;
                    out[31:16] = in.a % in.b;
                end
                else begin
                    out = 32'b0;
                    assert(0);
                end
            end
         endcase
    end

    always_comb begin:flag_logic
        stat.zero = !(|out);
        stat.div0 = ((control == DIV)&&(in.b == 32'd0)) ? 1'b1 : 1'b0;

        stat.overflow = (control == ADD || control == SUB) ? arith[17]^arith[16] : 1'b0;

        if(control == MULT) stat.sign = out[31];
        else                stat.sign = out[15];
    end
endmodule
```

../source/Design/alu.sv

```
module adder(
    input logic     [15:0]  pc,
    input logic     [15:0]  offset,
    output logic    [15:0]  sum
);

    logic overflow; // If there is an overflow, that is bad!

    assign {overflow, sum} = pc + offset;
endmodule
```

../source/Design/adder.sv

```
module alu(
    input alu_pkg::in_t       in,
    input alu_pkg::control_e  control,

    output alu_pkg::status_t   stat,
    output integer   out
);
    import alu_pkg::*;

    logic carry;
    logic signed [17:0] arith;

    always_comb begin
        case(control)
            OR  : out = {16'b0,in.a | in.b};
            AND : out = {16'b0,in.a & in.b};
            MULT: out = in.a * in.b;
            ROL : out = {16'b0,({in.a, in.a} << in.b%16)};
            ROR : out = {16'b0,({in.a, in.a} >> in.b%16)};
            SHL : out = {16'b0,in.a <<< in.b};
            SHR : out = {16'b0,in.a >>> in.b};
            SUB : begin
                arith = in.a - in.b;
                out = {16'b0, arith[15:0]};
            end
            ADD : begin
                arith = in.a + in.b;
                out = {16'b0, arith[15:0]};
            end
            DIV : begin
                if(in.b != 0) begin
                    out[15:0] = in.a / in.b;
                    out[31:16] = in.a % in.b;
                end
                else begin
                    out = 32'b0;
                    assert(0);
                end
            end
         endcase
    end

    always_comb begin:flag_logic
        stat.zero = !(|out);
        stat.div0 = ((control == DIV)&&(in.b == 32'd0)) ? 1'b1 : 1'b0;

        stat.overflow = (control == ADD || control == SUB) ? arith[17]^arith[16] : 1'b0;

        if(control == MULT) stat.sign = out[31];
        else                stat.sign = out[15];
    end
endmodule
```

../source/Design/alu.sv

```
package alu_pkg;

    typedef logic signed [15:0] word_16;

    typedef enum logic[3:0]{
        MULT= 4'h1,
        DIV = 4'h2,
        ROL = 4'h8,
        ROR = 4'h9,
        SHL = 4'hA,
        SHR = 4'hB,
        OR  = 4'hC,
        AND = 4'hD,
        SUB = 4'hE,
        ADD = 4'hF
    } control_e;

    // Status flags for ALU
    // sign asserted when positive
    typedef struct{
        logic   sign;
        logic   overflow;
        logic   zero;
        logic   div0;
    } status_t;

    // Status flags for ALU
    // sign asserted when positive
    typedef struct{
        word_16 a;
        word_16 b;
    } in_t;

endpackage
```

../source/Design/alu_pkg.sv

```
module comparator(
    input wire      [15:0]  in1,
    input wire      [15:0]  in2,

    output types_pkg::result_t  cmp_result
);
    import types_pkg::*;

    assign cmp_result = (in1 > in2) ? GREATER   :
                        (in1 < in2) ? LESS      :
                        (in1 == in2)? EQUAL     :
                                      UNKNOWN;

endmodule
```

../source/Design/comparator.sv

```
module control_alu(
    input alu_pkg::control_e    func,
    input wire                  ALUop,

    output alu_pkg::control_e   alu_ctrl,
    output logic                immb,
    output logic                R0_en
);
    import alu_pkg::*;

    assign immb = ((func == ROR)||(func == ROL)||(func == SHR)||(func == SHL));
    assign R0_en = ((func == MULT)||(func == DIV));
    assign alu_ctrl = (ALUop) ? ADD : func;

endmodule
```

../source/Design/control_alu.sv

C. Stage Two

```
module control_hazard_unit(

    input wire              s2_R0_en,
    input wire              s3_R0_en,
    input types_pkg::opcode_t           opcode,
    input types_pkg::opcode_t           s2_opcode,
    input types_pkg::opcode_t           s3_opcode,

    input wire      [3:0]   r1,
    input wire      [3:0]   r2,
    input wire      [3:0]   s2_r1,
    input wire      [3:0]   s3_r1,

    output logic    [10:0]  haz,
    output logic            stall
);

  import alu_pkg::*;
  import types_pkg::*;

  logic stall_logic;


  logic haz0, haz1, haz2, haz3, haz4, haz5, haz6, haz7, haz8,
   haz9, haz10;

  // Arithmetic or load followed two instructions later
  // another arithmetic(Or STORE) using same destination
  // register for R1.
  assign haz0 = ((opcode == ARITHM))
          &&((s3_opcode == ARITHM))
          &&((r1 == s3_r1));
  assign haz[0] = (haz0) ? 1'b1 : 1'b0 ;

  // Arithmetic or load directly followed by an arithmetic
  // op with the R1 as the first destination.
  assign haz1 = ((opcode == ARITHM))
          &&((s2_opcode == ARITHM)||(s2_opcode == LW))
          &&((r1 == s2_r1));
  assign haz[1] = (haz1) ? 1'b1: 1'b0;

  // Arithmetic or load directly followed by an arithmetic
  // op with the R2 as the first destination.
  assign haz2 = ((opcode == ARITHM))
```

```
          &&((s2_opcode == ARITHM)||(s2_opcode == LW))
          &&((r2 == s2_r1));
  assign haz[2] = (haz2) ? 1'b1: 1'b0;

  // Arithmetic or load followed 2 instructions later by an
  // arithmetic op with the R2 as the first destination
  assign haz3 = ((opcode == ARITHM))
          &&((s3_opcode == ARITHM)||(s3_opcode == LW))
          &&((r2 == s3_r1));
  assign haz[3] = (haz3) ? 1'b1: 1'b0;

  // An Arithmetic operation is followed directly by a
  // branch instruction
  assign haz4 = ((opcode == BE)||(opcode == BLT)||(opcode == BGT))
          &&((s2_opcode == ARITHM))
          &&(r1 == s2_r1);
  assign haz[4] = (haz4) ? 1'b1: 1'b0;

  // LOAD is followed directly, or second instruction, by a
  // branch instruction using the dest register for compare.
  // Also if an arithmetic op was followed 2 instructions
  // later by a branch instruction using its dest register
  assign haz5 = ((opcode == BE)||(opcode == BLT)||(opcode == BGT))
          &&((s3_opcode == LW)||(s2_opcode == LW)||(s3_opcode == ARITHM))
          &&((r1 == s2_r1)||(r1 == s3_r1)&&!(s2_opcode == LW));
  assign haz[5] = (haz5 && !haz4) ? 1'b1: 1'b0;

  // Multiply or divide is followed directly by a branch
  // instruction(What registers they specify does not matter.
  // This is for R0 which is implicitly used by all 3 types)
  assign haz6 = ((opcode == BE)||(opcode == BLT)||(opcode == BGT))
          &&((s2_R0_en)); // Only if MULT or DIV
  assign haz[6] = (haz6) ? 1'b1: 1'b0;

  // Multiply or divide is followed 2 instructions later by
  // a branch instruction(What registers they specify does
  // not matter. This is for R0 which is implicitly used by
  // all 3 types)
  assign haz7 = ((opcode == BE)||(opcode == BLT)||(opcode == BGT))
          &&((s3_R0_en)); // Only if MULT or DIV
  assign haz[7] = (haz7) ? 1'b1: 1'b0;


    // ==================================================
    // These LW/SW things might be checking the wrong registers
    // Check here if problems occur
    //======================================================

    // LOAD is followed directly by a STORE instruction
    // using same reg for dest(load)/src(store)
    assign haz8 = ((opcode == SW))
                    &&((s2_opcode == LW))
                    &&((r1 == s2_r1)||(r2 == s2_r1));
    assign haz[8] = (haz8) ? 1'b1: 1'b0;

    // LOAD is followed 2 instructions later by a STORE
    // instruction using same reg for dest(load)/src(store)
    assign haz9 = ((opcode == SW))
                    &&((s3_opcode == LW))
                    &&((r1 == s3_r1)||(r2 == s3_r1));
    assign haz[9] = (haz9 && !haz10) ? 1'b1: 1'b0;
    // Arithmetic instruction followed directly by a STORE
    // instruction using same reg for dest/src
    assign haz10 = ((opcode == SW))
                    &&((s2_opcode == ARITHM))
                    &&((r1 == s2_r1)||(r2 == s2_r1));
    assign haz[10] = (haz10) ? 1'b1: 1'b0;


    // LOAD is followed directly by a branch instruction
    // using the dest register for compare
    assign stall_logic = ((opcode == BE)||(opcode == BLT)||(opcode == BGT))
                    &&((s2_opcode == LW))
                    &&((r1 == s2_r1)||(r2 == s2_r1));
    assign stall = (stall_logic) ? 1'b1: 1'b0;
endmodule
```

../source/Design/control_hazard_unit.sv

```
module control_jump(
    input types_pkg::result_t   cmp_result,
    input types_pkg::opcode_t   opcode,

    output logic    jmp
);

import types_pkg::*;

always_comb begin
    case(opcode)
        BLT:
            if(cmp_result == LESS)
                jmp = 1'b1;
            else
                jmp = 1'b0;
        BGT:
            if(cmp_result == GREATER)
                jmp = 1'b1;
            else
                jmp = 1'b0;
        BE:
            if(cmp_result == EQUAL)
                jmp = 1'b1;
            else
                jmp = 1'b0;
        JMP:
            jmp = 1'b1;
        default:
            jmp = 1'b0;
    endcase
end

endmodule
```

../source/Design/control_jump.sv

```
module control_main(
    input types_pkg::opcode_t   opcode,
    input alu_pkg::control_e    func,
    input wire                  div0,
    input wire                  overflow,

    output logic                ALUop,
    output types_pkg::sel_t     offset_sel,
    output logic                mem2r,
    output logic                memwr,
    output logic                halt_sys,
    output logic                reg_wr,
    output logic                R0_read,
    output logic                se_imm_a
);

    import types_pkg::*;
    import alu_pkg::*;

    always_comb begin
        if (div0 || overflow) begin // Exception
            ALUop = 1'b0;
            offset_sel = NONE;
            mem2r = 1'b0;
            memwr = 1'b0;
            halt_sys = 1'b1;
            reg_wr = 1'b0;
            R0_read = 1'b0;
            se_imm_a = 1'b0;
        end
        else begin
            case (opcode)
                ARITHM: begin
                    ALUop = 1'b0;
                    if((func == ROR)||(func == ROL)||(func == SHL)||(func == SHR))
                        offset_sel = FOURBIT;
                    else
                        offset_sel = NONE;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b1;
                    R0_read = 1'b0;
                    se_imm_a = 1'b0; // Not soooo sure bout this one
                end
                LW: begin
                    ALUop = 1'b1;
                    offset_sel = FOURBIT;
                    mem2r = 1'b1;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b1;
                    R0_read = 1'b0;
                    se_imm_a = 1'b1;
                end
                SW: begin
                    ALUop = 1'b1;
                    offset_sel = FOURBIT;
                    mem2r = 1'b0;
                    memwr = 1'b1;
                    halt_sys = 1'b0;
                    reg_wr = 1'b0;
                    R0_read = 1'b0;
                    se_imm_a = 1'b1;
                end
                BLT: begin
                    ALUop = 1'b1;
                    offset_sel = EIGHTBIT;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b0;
                    R0_read = 1'b1;
                    se_imm_a = 1'b1;
                end
                BGT: begin
```

```
                    ALUop = 1'b1;
                    offset_sel = EIGHTBIT;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b0;
                    R0_read = 1'b1;
                    se_imm_a = 1'b1;
                end
                BE: begin
                    ALUop = 1'b1;
                    offset_sel = EIGHTBIT;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b0;
                    R0_read = 1'b1;
                    se_imm_a = 1'b1;
                end
                JMP: begin
                    ALUop = 1'b1;
                    offset_sel = TWELVEBIT;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b0;
                    reg_wr = 1'b0;
                    R0_read = 1'b0;
                    se_imm_a = 1'b1;
                end
                HALT: begin
                    ALUop = 1'b0;
                    offset_sel = NONE;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b1;
                    reg_wr = 1'b0;
                    R0_read = 1'b0;
                    se_imm_a = 1'b1;
                end

                default: begin      // Exception
                    ALUop = 1'b0;
                    offset_sel = NONE;
                    mem2r = 1'b0;
                    memwr = 1'b0;
                    halt_sys = 1'b1;
                    reg_wr = 1'b0;
                    R0_read = 1'b0;
                    se_imm_a = 1'b0;
                end
            endcase
        end // if(exception)
    end
endmodule
```

../source/Design/control_main.sv

```
//============================================================
// Main memory block
// Word addressable (16-bit)
//
//============================================================

module mem_main(
    input wire          rst,
    input wire          clk,
    input wire          halt_sys,


    input wire          write_en,
    input wire  [15:0]  address,
    input wire  [15:0]  write_data,

    output logic[15:0]  data_out
);
    logic clockg;

    logic   [7:0] memory[65536:0];  // Memory block. 16 bit address with 16 bit data
    logic   [7:0] shadow_memory[65536:0] = '{default:0};

    always_comb begin: clock_gating
     clockg = (halt_sys == 1'b1|| write_en == 1'b0)?1'b0:clk; //flop clock gated
    end

    always_comb begin: memory_read_logic
        data_out = {memory[address], memory[address +1]};   // Always read the data from the address
    end

    always_ff@(posedge clockg ,posedge rst) begin: memory_rst_and_write
        if(rst == 1'b1) memory <=  shadow_memory;// If rst is asserted, we want to clear the flops
        else if (write_en)          {memory[address], memory[address +1]} <= write_data;    // Flop the input
    end
endmodule
```

../source/Design/mem_main.sv

```
module mem_program(
    input wire  [15:0]  address,
    output logic[15:0]  data_out
);

    logic rst = 0;

    logic  [7:0] memory[100:0] = '{default:8'b0};  // Memory block. 16 bit address with 16 bit data

    assign data_out = {memory[address], memory[address+1]}; // Always read the data from the address

endmodule
```

../source/Design/mem_program.sv

```
module mem_register(
    input wire          rst,
    input wire          clk,
    input wire          halt_sys,

    input wire          R0_read,
    input wire  [3:0]   ra1,
    input wire  [3:0]   ra2,

    input wire          write_en,
    input wire          R0_en,
    input wire  [3:0]   write_address,
    input wire  [31:0]  write_data,

    output logic[15:0]  rd1,
    output logic[15:0]  rd2
);

    reg     [15:0]      write_data_high;
    reg     [15:0]      write_data_low;
    reg                 clockg;

    logic   [15:0]      registers[31:0];    // Memory block. 4 bit address with 16 bit data
    logic   [15:0]      zregisters[31:0] ='{default:0};

    assign  {write_data_high, write_data_low} = write_data;     // Split the input data into two words

    //generates clock that will only allow writes when they are supposed to.
    always_comb begin: clock_gating
     clockg = (halt_sys == 1'b1|| write_en == 1'b0)?1'b0:clk; //flop clock gated
    end

    //Combinatorial read logic
    always_comb begin: memory_read_logic
        rd1 = registers[ra1];   // Always read the data from the address
        rd2 = (R0_read) ? registers[0] : registers[ra2];        // if R0_read is high then R0 contents are output at r2
    end

    //sequential write logic.
    always_ff@(posedge clockg, posedge rst) begin: mem_reg_flop
        if (rst == 1'b1) begin
            registers <= zregisters;// If rst is asserted, we want to clear the flops
        end
        else begin// Write data to reg, and write top 16 bits to R0 if R0_en is high
            if (R0_en) registers[0] <= write_data_high;
            registers[write_address] <= write_data_low;     // Flop the input
        end
    end
endmodule
```

../source/Design/mem_register.sv

```
module mux
    #(parameter SIZE = 16, parameter IS3WAY = 1)(
    input wire [IS3WAY:0]       sel,

    input wire [(SIZE - 1):0]   in1,
    input wire [(SIZE - 1):0]   in2,
    input wire [(SIZE - 1):0]   in3,

    output logic [(SIZE - 1):0] out
);

    always_comb begin
        if(IS3WAY) begin        //3 to one mux
            case (sel)
                2'b00:
                    out = in1;
                2'b10:
                    out = in2;
                2'b01:
                    out = in3;
                2'b11: begin
                    out = 32'bX;
                end
            endcase
        end
        else begin
```

```
            case (sel)          // 2 to one mux
                1'b0:
                    out = in1;
                1'b1:
                    out = in2;
            endcase
        end
    end
endmodule
```

../source/Design/mux.sv

```
module reg_program_counter(
    input wire clk,
    input wire rst,

    input wire halt_sys,    // Control signal from main control to halt cpu
    input wire stall,       // Control signal from hazard unit to stall for one cycle

    input wire [15:0] in_address,   // Next PC address

    output logic [15:0] out_address // Current PC address
);

    always_ff@ (posedge clk or posedge rst) begin: program_counter_flop
        if (rst) begin
            out_address <= 16'd0;
        end
        else begin
            if(halt_sys || stall)
                out_address <= out_address; // Stay the same value. System is halted.
            else
                out_address <= in_address;  // Flop the input
        end
    end
endmodule
```

../source/Design/reg_program_counter.sv

```
module shift_one(
    input wire   [15:0] in,

    output logic [15:0] out
);

    assign out = {in << 1};

endmodule
```

../source/Design/shift_one.sv

D. Stage Three

```
module sign_extender(
    input types_pkg::sel_t           offset_sel,
    input wire               [11:0]  input_value,

    output logic             [15:0]  se_value
);
    import types_pkg::*;

    always_comb begin
        case (offset_sel)
            NONE:
                se_value = {4'h0, input_value};
            FOURBIT:
                if (input_value[3]) // Might not be sign extending these correctly
                    se_value = {12'hfff, input_value[3:0]};
                else
                    se_value = {12'h000, input_value[3:0]};

            EIGHTBIT:
                if (input_value[7]) // Might not be sign extending these correctly
                    se_value = {8'hff, input_value[7:0]};
                else
                    se_value = {8'h00, input_value[7:0]};

            TWELVEBIT:
                if (input_value[11]) // Might not be sign extending these correctly
                    se_value = {4'hf, input_value[11:0]};
                else
                    se_value = {4'h0, input_value[11:0]};
        endcase
    end
endmodule
```

../source/Design/sign_extender.sv

```
module stage_one(
      input wire          clk,
      input wire          rst,

        input wire [15:0]   s3_instruction,

        input wire          s2_R0_en,
        input wire          s3_R0_en,

        input wire [31:0]   s2_alu,
        input wire [31:0]   s3_data,
        input wire          div0,
        input wire          overflow,

        input wire          s3_reg_wr,
        input wire          s3_mem2r,

        //flopped outputs
        output reg          stall,
        output reg          halt_sys,
        output types_pkg::memc_t  out_memc,
        output reg          out_reg_wr,
        output alu_pkg::in_t         out_alu,
        output reg          out_haz1,
        output reg          out_haz2,
        output reg          out_haz8,
        output reg          out_R0_en,
        output alu_pkg::control_e    out_alu_ctrl,
        output types_pkg::uword     out_instr,
        output types_pkg::uword     out_R1_data,

        output types_pkg::memc_t memc
    );

    import types_pkg::*;
  import alu_pkg::*;

    //| Local logic instantiations
    //| ========================================================================
    uword PC_address;

    logic [15:0] instruction;

    opcode_t        opcode;
    control_e       func_code;

    sel_t           offset_sel;
    wire    [15:0]  offset_se;
    wire    [15:0]  offset_shifted;

    wire    [15:0]  cmp_a;
    wire    [15:0]  cmp_b;
    result_t        cmp_result;

    wire    [15:0]  PC_no_jump;
    wire    [15:0]  PC_jump;
    wire    [15:0]  PC_next;

    uword       R1_data;
    uword     R1_data_muxed;
    wire    [15:0]  r2_data;

    wire    [10:0]  haz;
  wire      R0_en;

  reg     R0_read;
  memc_t    s3_memc;
  reg     ALUop;
  reg     reg_wr;
  reg     se_imm_a;
  control_e   alucontrol;
  reg     immb;
  reg     jmp;
  in_t    alu_muxed;

    assign opcode = opcode_t’(instruction[15:12]);
    assign func_code = control_e’(instruction[3:0]);

    //| Stage 1 Flip-Flop
    //| ========================================================================
    always_ff@ (posedge clk or posedge rst) begin: stage_A_flop
        if (rst) begin
            out_memc        <= memc_t’(2’d0);
            out_reg_wr      <= 1’d0;
            out_alu.a       <= 16’d0;
            out_alu.b       <= 16’d0;
            out_R1_data     <= 16’d0;
            out_haz1        <= 1’b0;
            out_haz2        <= 1’b0;
            out_haz8        <= 1’b0;
            out_R0_en       <= 1’d0;
            out_alu_ctrl    <= ADD;
            out_instr       <= 8’d0;    // Top 8 bits of instruction // If rst is asserted, we want to clear the flops

        end
        else begin
            if(halt_sys || stall) begin
```

```
                // Stay the same value. System is halted.
            end
            else                // Flop the input
                out_memc        <= memc;
                out_reg_wr      <= reg_wr;
                out_alu         <= alu_muxed;
                out_R1_data     <= R1_data_muxed;
                out_haz1        <= haz[1];
                out_haz2        <= haz[2];
                out_haz8        <= haz[8];
                out_R0_en       <= R0_en;
                out_alu_ctrl    <= alucontrol;
                out_instr       <= instruction;
        end
    end

    //| PC adder instantiation
    //| ==========================================================================
    adder pc_adder(
        .pc(PC_address),
        .offset(16'd2),
        .sum(PC_no_jump)
    );

    //| Jump adder instantiation
    //| ==========================================================================
    adder jump_adder(
        .pc(PC_no_jump),
        .offset(offset_shifted),
        .sum(PC_jump)
    );

    //| Memory Instantiations
    //| ==========================================================================
    mem_program program_memory(
        .address(PC_address),
        .data_out(instruction)
    );

    reg_program_counter pc_reg(
        .clk(clk),
        .rst(rst),

        .halt_sys(halt_sys),    // Control signal from main control to halt cpu
        .stall(stall),          // Control signal from hazard unit to stall for one cycle

        .in_address(PC_next),   // Next PC address
        .out_address(PC_address)// Current PC address
    );

    mem_register register_file (
        .rst(rst),
        .clk(clk),
        .halt_sys(halt_sys),

        .R0_read(R0_read),
        .ra1(instruction[11:8]),
        .ra2(instruction[7:4]),

        .write_en(s3_reg_wr || s3_mem2r),
        .R0_en(s3_R0_en),
        .write_address(s3_instruction[11:8]), // r1 address
        .write_data(s3_data),

        .rd1(R1_data),
        .rd2(r2_data)
    );


    //| Main Control Unit
    //| ==========================================================================
    control_main Control_unit(
        .opcode(opcode),
        .func(func_code),
        .div0(div0),
        .overflow(overflow),

        .ALUop(ALUop),
        .offset_sel(offset_sel),

        .mem2r(memc.mem2r),
        .memwr(memc.memwr),
        .halt_sys(halt_sys),
        .reg_wr(reg_wr),
        .R0_read(R0_read),
        .se_imm_a(se_imm_a)
    );

    control_alu alu_control(
        .func(func_code),
        .ALUop(ALUop),

        .alu_ctrl(alucontrol),
        .immb(immb),
        .R0_en(R0_en)
    );

    control_jump jump_unit(
        .cmp_result(cmp_result),
        .opcode(opcode),

        .jmp(jmp)
    );

    //| Hazard Detection Unit
    //| ============================================================================
    control_hazard_unit HDU(
        .s2_R0_en(s2_R0_en),
        .s3_R0_en(s3_R0_en),
        .opcode(opcode),
        .s2_opcode(opcode_t'(out_instr[15:12])), // s2 and s3 instructions hold
        .s3_opcode(opcode_t'(s3_instruction[15:12])), // top 8 bits of that instr

        .r1(instruction[11:8]),
        .r2(instruction[7:4]),
        .s2_r1(out_instr[11:8]),
        .s3_r1(s3_instruction[11:8]),

        .haz(haz),
        .stall(stall)
    );


    //| Sign Extending unitw
    //| ============================================================================
    sign_extender sign_extend(
        .offset_sel(offset_sel),
        .input_value(instruction[11:0]),    // 11:0 to handle all 3 different sized offsets.

        .se_value(offset_se)
    );

    //| Shift Left Unit
    //| ============================================================================
    shift_one shift1(
        .in(offset_se),
        .out(offset_shifted)
    );


    //| Comparator
    //| ============================================================================
    comparator cmp(
        .in1(cmp_a),
        .in2(cmp_b),

        .cmp_result(cmp_result)
    );

    //| Mux
    //| ============================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )Mux0(
        .sel(jmp),
        .in1(PC_no_jump),
        .in2(PC_jump),
      .in3(16'b0),

        .out(PC_next)
    );

    //| Mux before comparator with R1
    //| ============================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )mux1(
        .sel({haz[4], haz[5]}),

        .in1(R1_data),
        .in2(s2_alu[15:0]),
        .in3(s3_data[15:0]),

        .out(cmp_a)
    );

    //| Mux before comparator with R2
    //| ============================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )mux2(
        .sel({haz[6], haz[7]}),

        .in1(r2_data),
        .in2(s2_alu[31:16]),
        .in3(s3_data[31:16]),

        .out(cmp_b)
    );

    //| Mux for R1_data
    //| ============================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )mux3(
        .sel({haz[10], haz[9]}),    // mem2r

        .in1(R1_data),
        .in2(s2_alu[15:0]),
        .in3(s3_data[15:0]),

        .out(R1_data_muxed)
    );

    //| Mux for ALU_a
    //| ==========================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )mux4(
        .sel({haz[0], se_imm_a}),
        .in1(R1_data),
        .in2(s3_data[15:0]),
      .in3(offset_se),

        .out(alu_muxed.a)
    );

    //| Mux for ALU_B
    //| ==========================================================================
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )mux5(
        .sel({immb, haz[3]}),
        .in1(r2_data),
        .in2({12'd0, instruction[7:4]}),
        .in3(s3_data[15:0]),

        .out(alu_muxed.b)
    );

endmodule
```

../source/Design/stage_one.sv

```
module stage_three(
    input   wire                clk,
    input   wire                rst,
    input   types_pkg::uword    instruction,

    input   reg     [31:0]      alu,
    input   types_pkg::memc_t   memc,
    input   types_pkg::uword    r1_data,
    input   wire                r0_en,
    input   wire                halt_sys,

    output  reg     [31:0]      data,
    output  types_pkg::uword    r1_data_out,
    output  types_pkg::memc_t   out_memc,
    output  reg                 out_r0_en,
    output  types_pkg::uword    instruction_out
);
    import types_pkg::*;

    logic [15:0]    data_muxed;
    uword           mem_data;
    opcode_t        opcode;

    assign data            = {alu[31:16], data_muxed[15:0]};
    assign out_r0_en       = r0_en;
    assign out_memc        = memc;
    assign instruction_out = instruction;
    assign r1_data_out     = r1_data;
    assign opcode          = opcode_t'(instruction[15:12]);

    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )mux9(
        .sel(memc.mem2r),
        .in1(alu[15:0]),
        .in2(mem_data),
        .in3(16'b0),

        .out(data_muxed[15:0])
    );

    //| Main Memory
    //| ==========================================================================
    mem_main main_memory(
        .rst(rst),
        .clk(clk),
        .halt_sys(halt_sys),

        .write_en(memc.memwr),
        .address(alu[15:0]),
        .write_data(r1_data),
        .data_out(mem_data)
    );

endmodule
```

../source/Design/stage_three.sv

```
module stage_two(
        input rst,
        input clk,

        input reg halt_sys,
        input reg stall,

        input types_pkg::memc_t in_memc,
        input alu_pkg::in_t in_alu,
        input alu_pkg::control_e alu_control,
        input [15:0] in_R1_data,
        input in_R0_en,
        input wire [15:0] in_instr,
        input wire haz1,
        input wire haz2,
        input wire haz8,
        input wire [31:0] s3_data,
        input wire in_reg_wr,
        output reg out_reg_wr,
        output types_pkg::memc_t out_memc,
        output reg [31:0] out_alu,
        output reg [31:0] out_alu_result,
        output reg [15:0] out_R1_data,
        output reg out_R0_en,
        output reg [15:0] out_instr,
        output alu_pkg::status_t out_alu_stat
    );

    import alu_pkg::*;
    import types_pkg::*;

    control_e alucontrol;
    integer aluout;
    reg     [15:0]  in_R1_data_muxed;

    in_t alu_muxed;

    assign out_alu_result = aluout;
    //| Stage B flip flop
    //| =======================================================
    always_ff@ (posedge clk or posedge rst) begin: stage_B_flop
        if (rst) begin
            out_memc        <= memc_t'(2'd0);
            out_alu         <= 32'd0;
            out_R1_data     <= 16'd0;
            out_R0_en       <= 1'd0;
            out_instr       <= 8'd0;    // Top 8 bits of instruction // If rst is asserted, we want to clear the flops
            out_reg_wr      <= 1'd0;
        end
        else begin
            if(halt_sys || stall) begin
                // Stay the same value. System is halted.
            end
            else                // Flop the input
                out_memc        <= in_memc;
                out_alu         <= aluout;
                out_R1_data     <= in_R1_data_muxed;
                out_R0_en       <= in_R0_en;
                out_instr       <= in_instr;
                out_reg_wr      <= in_reg_wr;
        end
    end

    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )muxa(
        .sel(haz1),
        .in1(in_alu.a),
        .in2(s3_data[15:0]),
        .in3(16'b0),

        .out(alu_muxed.a)
    );

    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )muxb(
        .sel(haz2),
        .in1(in_alu.b),
        .in2(s3_data[15:0]),
        .in3(16'b0),

        .out(alu_muxed.b)
    );

    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )muxc(
        .sel(haz8),
        .in1(in_R1_data),
        .in2(s3_data[15:0]),
        .in3(16'b0),

        .out(in_R1_data_muxed)
    );

    //| ALU instantiation
    //| ===========================================================================
    alu main_alu(
        .in      (alu_muxed),
        .control(alu_control),
        .stat    (out_alu_stat),
        .out     (aluout)
    );


endmodule
```

../source/Design/stage_two.sv

```
package types_pkg;
    import alu_pkg::*;

    typedef logic [15:0] uword;

    typedef enum logic[1:0]{
            GREATER         = 2'b00,
            LESS            = 2'b01,
            EQUAL           = 2'b10,
            UNKNOWN         = 2'b11
    } result_t;

    typedef enum logic[3:0]{
        ARITHM      = 4'b0000,
        LW          = 4'b1000,
        SW          = 4'b1011,
        BLT         = 4'b0100,
        BGT         = 4'b0101,
        BE          = 4'b0110,
        JMP         = 4'b1100,
        HALT        = 4'b1111
    } opcode_t;

    typedef enum logic[1:0]{
        NONE        = 2'b00,
        FOURBIT     = 2'b01,
        EIGHTBIT    = 2'b10,
        TWELVEBIT   = 2'b11
    } sel_t;

    // Status flags for ALU
    // sign asserted when positive
    typedef struct{
        logic memwr;
        logic mem2r;
    } memc_t;

endpackage
```

../source/Design/types_pkg.sv

VI. Verification Source Code

```
// ALU test bench
//
// This module generates random stiumlus for ALU both data and control lines
// through testing is ensured by simulation coverage metrics collected by VCS
//

import alu_pkg::*;
import types_pkg::*;
//`define VERBOSE
//`define BOUNDED_INPUTS

//called by check_alu_outputs to print debug updates
task static print_alu_state(string ident, integer result, control_e control, in_t in, integer out, status_t stat, reg ov);
    // different print formats for different functions
    case(control)
        MULT: begin
            $display("%s -- time %4d - op: %s", ident, $time(), control.name);
            $display("s:%b o:%b, z:%b -- Expected: s:%b o:%b, z:%b", stat.sign, stat.overflow, stat.zero, result[31], ov, !(|result));
            $display("%11d - %b", in.a,in.a);
            $display("%11d - %b", in.b,in.b);
            $display("================================");
            $display("%11d - %b <-- result", out[31:0], out[31:0]);
            $display("%11d - %b <-- expected \n", result[31:0], result[31:0]);
        end

        DIV: begin
            $display("%s -- time %4d - op: %s", ident, $time(), control.name);
            $display("s:%b o:%b, z:%b -- Expected: s:%b o:%b, z:%b", stat.sign, stat.overflow, stat.zero, result[15], ov, !(|result));
            $display("%11d - %b", in.a,in.a);
            $display("%11d - %b", in.b,in.b);
            $display("================================");
            $display("%11d - %b <-- result", out[15:0], out[15:0]);
            $display("%11d - %b <-- expected \n", result[15:0], result[15:0]);
        end

        default :begin
            $display("%s -- time %4d - op: %s", ident, $time(), control.name);
            $display("s:%b o:%b, z:%b -- Expected: s:%b o:%b, z:%b", stat.sign, stat.overflow, stat.zero, result[15], ov, !(|result));
            $display("%11d - %b", in.a,in.a);
            $display("%11d - %b", in.b,in.b);
            $display("================================");
            $display("%11d - %b <-- result", signed'(out[15:0]), out[15:0]);
            $display("%11d - %b <-- expected \n", signed'(result[15:0]), result[15:0]);
        end
    endcase
endtask

// returns number of errors for a given test cycle
function automatic check_alu_outputs(
    status_t stat,
    control_e control,
    in_t     in,
    integer  out
);

    reg                 ov; //expected overflow
    reg                 simsign; //expected simulation sign
    integer             result; //ALU result
    reg signed [15:0]   tarith; //dummy logic for overflow detector
    integer             failure_count = 0; //total number of failures across all checks

    //calculate expected result
    case(control)
        OR  : result = {16'b0,in.a | in.b};
        AND : result = {16'b0,in.a & in.b};
        MULT: result = in.a * in.b;
        ROL : result = {16'b0,({in.a, in.a} <<< in.b)};
        ROR : result = {16'b0,({in.a, in.a} >>> in.b)};
        SHL : result = {16'b0,in.a <<< in.b};
        SHR : result = {16'b0,in.a >>> in.b};
        SUB : result = {16'b0,in.a - in.b};
        ADD : result = {16'b0,in.a + in.b};
        DIV : begin
            if(in.b != 0) begin
                result[15:0] = in.a / in.b;
                result[31:16] = in.a % in.b;
            end
            else begin
                result = 32'b0;
            end
        end
     endcase

    //expected overflow flag calculation
    case(control)
        OR  : ov = 0;
        AND : ov = 0;
        MULT: ov = 0;
        ROL : ov = 0;
        ROR : ov = 0;
        SHL : ov = 0;
        SHR : ov = 0;
        SUB : {ov,tarith} = {in.a - in.b};
        ADD : {ov,tarith} = {in.a + in.b};
        DIV : ov = 0;
     endcase

    //Sign flag test
    simsign = (control == MULT) ? result[31] : result[15];
    if((stat.sign != simsign) && !stat.overflow) begin
        print_alu_state("Sign Flag FAILURE", result, control, in, out, stat, ov);
        failure_count++;
    end
    `ifdef VERBOSE
    else
        print_alu_state("Sign Flag SUCCESS", result, control, in, out, stat, ov);
    `endif

    //Overflow flag test
    if((stat.overflow != ov) && !control[1]) begin
        print_alu_state("Overflow Flag FAILURE",result, control, in, out, stat, ov);
        failure_count++;
    end
    `ifdef VERBOSE
    else
        print_alu_state("Overflow Flag SUCCESS", result, control, in, out, stat, ov);
    `endif

    //Zero flag test
    if((stat.zero && |out)&& !stat.overflow) begin
        print_alu_state("Zero Flag FAILURE", result, control, in, out, stat, ov);
        failure_count++;
    end
    `ifdef VERBOSE
    else
        print_alu_state("Zero Flag SUCCESS", result, control, in, out, stat, ov);
    `endif

    //ALU result flag test
    if((result != out)&& (!stat.overflow)) begin
        print_alu_state("ALU FAILURE", result, control, in, out, stat, ov);
        failure_count++;
    end
    `ifdef VERBOSE
    else
        print_alu_state("ALU SUCCESS", result, control, in, out, stat, ov);
    `endif

    return failure_count;
endfunction

//Class to utilize system verilog's random generation capabilityseald b
class alu_stim;
    rand alu_pkg::control_e control;
    rand word_16            a;
    rand word_16            b;

    //simple numbers for human inspection
    `ifdef BOUNDED_INPUTS
    constraint limits{
        a <= 2;
        a >= -2;

        b <= 2;
        b >= -2;
        b != 0;
    }
    `endif

    //randomize wrapper incase more random features needed
    function r();
        randomize();
    endfunction
endclass

//Main module instanciates classes, modules and wiring
module alu_tb();
    import alu_pkg::*;

    //ALU  I/O lines
    alu_pkg::control_e  control;
    status_t            stat;
    in_t                alu_input;
    integer             alu_output;
    integer             errors = 0;
    integer             testiterations = 10000;
    integer             successes = 0;

    //instantiate ALU module
    alu DUT(
        .in     (alu_input),
        .control(control),
        .stat   (stat),
        .out    (alu_output)
    );

    initial begin
        alu_stim as = new;

        //apply test stimulus and check output
        for(int i = 0; i < testiterations; i++) begin
            //randomize inputs
            as.r();
            //drive DUT
            control = as.control;
            alu_input.a = as.a;
            alu_input.b = as.b;

            //wait and check
            #1 errors += check_alu_outputs(stat, control, alu_input, alu_output);
        end

        //print completion rate
        successes = testiterations - errors;
        $display("\n");
        $display("================Test Statistics=================");
        $display("Pass - %5d Passes", successes);
        $display("Pass - %5d Failures", errors);
        $display("  Percentage Pass: %3d", (successes/testiterations)*100);
        $display("================================================");
    end
endmodule
```

../source/Verif/alu_tb.sv

```
import alu_pkg::*;
import types_pkg::*;
```

```
//'define VERBOSE

class reg_stim;

    rand logic   [15:0]  memory_test_data;
    rand logic           halt;
    rand logic   [3:0]  address;

    function r();
        randomize();
    endfunction

    function [15:0] get_data();
        return memory_test_data;
    endfunction

    function get_halt();
        return halt;
    endfunction

    function [3:0] get_address();
        return address;
    endfunction
endclass

module register_tb();
    import alu_pkg::*;

    integer          errors;
    integer          testiterations = 10000;
    integer          successes;

    logic   [15:0]   test_reg[31:0];

    logic            rst;
    logic            clk;
    logic            halt_sys;

    logic            R0_read;
    logic    [3:0]   ra1;
    logic    [3:0]   ra2;

    logic            write_en;
    logic            R0_en;
    logic    [3:0]   write_address;
    logic    [31:0]  write_data;

    logic   [15:0]  rd1;
    logic   [15:0]  rd2;

    logic   [15:0]  test_data[31:0];

    mem_register dut(.*);
    reg_stim as = new;

    initial forever clk = #1 !clk;

    initial begin
        test_reg = '{default:0};
        rst = '0;
        clk = '0;
        halt_sys = '0;
        R0_read = '0;
        ra1 = '0;
        ra2 = '0;
        write_en = '0;
        R0_en = '0;
        write_address = '{default:0};
        write_data = '0;
        test_data = '{default:0};
        $readmemh("source/Verif/register_memory_blank.hex", dut.zregisters);

        #2 rst = 0;
        #2 rst = 1;
        #2 rst = 0;

        write_en = 1;

        // load memory with test data
        for(int i = 0; i < 16; i++) begin
               as.r();
               test_data[i] = as.get_data();
               write_data = as.get_data();
               write_address = i;
               #4 ;
        end
        write_en = 0;
        #2 ;
        for(int i = 0; i < 16; i++) begin
               if(test_data[i] != dut.registers[i])
                $display("Fail Write! Address: %d -- data ex: %h rec: %h", i, test_data[i], dut.registers[i]);
        end

        //|check stalling mechanism
        halt_sys = 1;
        // try to overwrite data with 1s
        for(int i = 0; i < 16; i++) begin
               write_data = 16'b1;
```

```
                write_address = i;
                #4 ;
        end
        halt_sys = 0;

        // read back test data
        for(int i = 0; i < 16; i++) begin
            #2
            if(test_data[i] != rd1)
               $display("Fail RD1! Address: %d -- data ex: %h rec: %h", i, test_data[i], rd1);
            else
                $display("Read Success! RD1! Address: %d -- data ex: %h rec: %h", i, test_data[i], rd1);

            if(test_data[i] != rd2)
               $display("Fail RD2! Address: %d -- data ex: %h rec: %h", i, test_data[i], rd1);
            else
                $display("Read Success! RD1! Address: %d -- data ex: %h rec: %h", i, test_data[i], rd1);

            ra1 = i + 1;
            ra2 = i + 1;
        end

        //check r0 write
        write_address = 10;
        write_data = 32'h555555;
        write_en = 1;
        R0_en = 1;
        #4
        if(dut.registers[0] != 16'h55)
            $display("Fail R0! Address: %d -- data ex: %h rec: %h", 0, 16'h55, dut.registers[0]);

        //check r0 read
        R0_read = 1;
        #1 ;
        if(rd1 != 16'h55)
            $display("Fail read R0! Address: %d -- data ex: %h rec: %h", 0, 16'h55, rd1);
        #1 R0_read = 0;
        #2 ;
    $finish;
    end
endmodule
```

../source/Verif/register_tb.sv

```
// `define VERBOSE //Prints information about test success, otherwise only
                   //failing checks will print information

// `define BOUNDED_INPUTS   //limits magnitutde of ALU inputs

typedef enum{RESET, IDLE, HAZARD, FULLTEST, HAZ0, HAZ1, HAZ2, HAZ3, HAZ4, HAZ5, HAZ6, HAZ7, HAZ8, HAZ9, HAZ10, STALL} SimPhase_e;

module system_tb();
    import alu_pkg::*;
    import types_pkg::*;
    import tb_utils_pkg::*;
    import tb_class_def::*;

    integer             testiteration = 0;
    integer             failure_count = 0;

    reg     [15:0]register_temp[4:0];

    logic clock = 0;
    logic reset = 0;

    uword memcheck;
    uword memcheck2;

    top dut(
        .clk(clock),
        .rst(reset)
    );

    SimPhase_e SimPhase;
    initial #4 forever #1 clock = ~clock;

    initial begin
        //| system wide reset
        //| =============================================================
        $xzcheckoff;
        $vcdpluson; //make that dve database
        $vcdplusmemon;
        #1 SimPhase = RESET;
           reset = 1;
        #1 $xzcheckon;
        #8 reset = 0;


        $readmemh("source/Verif/program_memory_blank.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        #1 SimPhase = IDLE;
           reset = 1;
        #1 reset = 0;

        #19
        $readmemh("source/Verif/haz0.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ0;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz1.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ1;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz2.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ2;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz3.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ3;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz4.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ4;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz5.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ5;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz6.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ6;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz7.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ7;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz8.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ8;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz9.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ9;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/haz10.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = HAZ10;
           reset = 1;
        #1 reset = 0;

        #19

        $readmemh("source/Verif/stall.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory_blank.hex", dut.st1.register_file.zregisters);
        SimPhase = STALL;
           reset = 1;
        #1 reset = 0;

        #19

        //| official test program
        SimPhase = FULLTEST;
        $readmemh("source/Verif/program_memory.hex", dut.st1.program_memory.memory);
        $readmemh("source/Verif/register_memory.hex", dut.st1.register_file.zregisters);
        $readmemh("source/Verif/main_memory.hex", dut.st3.main_memory.shadow_memory);
        #1 reset = 0;
        #1 reset = 1;
        #1 reset = 0;

        #60
        memcheck = {dut.st3.main_memory.memory[0], dut.st3.main_memory.memory[1]};
        memcheck2 = {dut.st3.main_memory.memory[2], dut.st3.main_memory.memory[3]};

        if (memcheck != 32'h2bcd) $display("check FAILED! memory[0] value = %h expected 2BCD", memcheck);
        else  $display("Memory check Passed! memory[0] value = %h expected 2bcd", memcheck);

        if (memcheck2 != 32'h579A) $display("check FAILED! memory[2] value = %h expected 579A", memcheck2);
        else $display("Memory check Passed! memory[2] value = %h expected 579a", memcheck2);
        $finish;
    end

  integer cycle = 0;
    always @ (negedge clock) begin
        if(SimPhase == FULLTEST) begin
          cycle++;
            //PC, ADDERS, MEMORY, REGISTER FILE, ALU, and pipeline buffers (inputs and output)
            $display("\n\n");
            $display("Current CPU State ====Cycle: %2d===================================", cycle);
            $display("Pipe Stage One ----------------------------------------------------");
            $display("stall        :%b"     , dut.st1.stall);
            $display("halt_sys     :%b"     , dut.st1.halt_sys);

            $display("out_memc     :%b"     , dut.st1.out_memc.mem2r);
            $display("out_reg_wr   :%b"     , dut.st1.out_reg_wr);
            $display("out_alu      :%b"     , integer'(dut.st1.out_alu));
            $display("out_haz1     :%b"     , dut.st1.out_haz1);
            $display("out_haz2     :%b"     , dut.st1.out_haz2);
            $display("out_haz8     :%b"     , dut.st1.out_haz8);
            $display("out_R0_en    :%b"     , dut.st1.out_R0_en);
            $display("out_alu_ctrl :%b"     , dut.st1.out_alu_ctrl);
            $display("out_instr    :%b"     , dut.st1.out_instr);
            $display("out_R1_data  :%b"     , dut.st1.out_R1_data);
            $display("memc         :%b"     , dut.st1.memc.mem2r);

            $display("instruction - %b"     , dut.st1.instruction );
            $display("PC_address - %b"      , dut.st1.PC_address );

            $display("opcode - %b"          , dut.st1.opcode );
            $display("func_code - %b"       , dut.st1.func_code );

            $display("offset_sel - %b"      , dut.st1.offset_sel );
            $display("offset_se - %b"       , dut.st1.offset_se );
            $display("offset_shifted - %b"  , dut.st1.offset_shifted );

            $display("cmp_a - %b"           , dut.st1.cmp_a );
            $display("cmp_b - %b"           , dut.st1.cmp_b );
            $display("cmp_result - %b"      , dut.st1.cmp_result );

            $display("PC_no_jump - %b"      , dut.st1.PC_no_jump );
            $display("PC_jump - %b"         , dut.st1.PC_jump );
            $display("PC_next - %b"         , dut.st1.PC_next );

            $display("R1_data - %b"         , dut.st1.R1_data );
            $display("R1_data_muxed - %b"   , dut.st1.R1_data_muxed );
            $display("r2_data - %b"         , dut.st1.r2_data );

            $display("haz - %b"             , dut.st1.haz );
            $display("R0_en - %b"           , dut.st1.R0_en );

            $display("Pipe Stage Two ----------------------------------------------------");
            $display("in_memc - %b "        , dut.st2.out_memc.mem2r);
            $display("in_alu a - %b "       , dut.st2.in_alu.a);
            $display("in_alu b- %b "        , dut.st2.in_alu.b);
            $display("alu_control - %b "    , dut.st2.alu_control);
            $display("in_R1_data - %b "     , dut.st2.in_R1_data);
            $display("in_R0_en - %b "       , dut.st2.in_R0_en);
            $display("in_instr - %b "       , dut.st2.in_instr);
            $display("haz1 - %b "           , dut.st2.haz1);
            $display("haz2 - %b "           , dut.st2.haz2);
            $display("haz8 - %b "           , dut.st2.haz8);
            $display("s3_data - %b "        , dut.st2.s3_data);
            $display("in_reg_wr - %b "      , dut.st2.in_reg_wr);

            $display("alucontrol - %s"      , dut.st2.alucontrol);
            $display("alu overflow - %b"    , dut.st2.alustat.sign);
            $display("alu sign - %b"        , dut.st2.alustat.overflow);
      $display("alu zero- %b"         , dut.st2.alustat.zero);

            $display("");
            $display("out_reg_wr - %b "     , dut.st2.out_reg_wr);
            $display("out_memc - %b "       , dut.st2.out_memc.mem2r);
            $display("out_alu - %b "        , dut.st2.out_alu);
            $display("out_alu_result - %b " , dut.st2.out_alu_result);
            $display("out_R1_data - %b "    , dut.st2.out_R1_data);
            $display("out_R0_en - %b "      , dut.st2.out_R0_en);
            $display("out_instr - %b "      , dut.st2.out_instr);

            $display("Pipe Stage Three --------------------------------------------------");
            $display("instruction - %b"     , dut.st3.instruction);
            $display("alu - %b"             , dut.st3.alu);
            $display("memc - %b"            , dut.st3.memc.mem2r);
            $display("r1_data - %b"         , dut.st3.r1_data);
            $display("r0_en - %b"           , dut.st3.r0_en);
            $display("halt_sys - %b"        , dut.st3.halt_sys);

            $display("");
            $display("data - %b"            , dut.st3.data);
            $display("r1_data_out - %b"     , dut.st3.r1_data_out);
            $display("out_memc - %b"        , dut.st3.out_memc.mem2r);
            $display("out_r0_en - %b"       , dut.st3.out_r0_en);
            $display("instruction_out - %b" , dut.st3.instruction_out);
            $display("===================================================================");
        end
    end
endmodule
```

../source/Verif/system_tb.sv

```


typedef enum{
    A_SEL,      // sel == 00
    C_SEL,      // sel == 01
    B_SEL       // sel == 10
} sele_t;


module mux_tb();
    import alu_pkg::*;
    import types_pkg::*;


    integer             testiteration = 0;
    integer             failure_counta = 0;
    integer             failure_countb = 0;
    integer             failure_countc = 0;

    logic               is3way;
    integer             size;
    sele_t               sel;
    logic [1:0]        select;
    logic [15:0]     input_a;
    logic [15:0]     input_b;
    logic [15:0]     input_c;
    wire  [15:0]     output_a;
    wire  [15:0]     output_b;
    wire  [1:0]     output_c;



    // 16-bit mux with 3 inputs
    mux #(
        .SIZE(16),
        .IS3WAY(1)
    )duta(
        .sel(select),
        .in1(input_a),
        .in2(input_b),
        .in3(input_c),

        .out(output_a)
    );

    // 16bit mux with 2 inputs
    mux #(
        .SIZE(16),
        .IS3WAY(0)
    )dutb(
        .sel(select[0]),
        .in1(input_a),
        .in2(input_b),
        .in3(input_c),

        .out(output_b)
    );

    // 2 bit mux with 2 inputs
    mux #(
        .SIZE(2),
        .IS3WAY(0)
    )dutc(
        .sel(select[0]),
        .in1(input_a[1:0]),
        .in2(input_b[1:0]),
        .in3(input_c[1:0]),

        .out(output_c)
    );
    assign select = sel;
    initial begin
        //| system wide reset
        //| ============================================================
        $xzcheckoff;
        $vcdpluson; //make that dve database
        $vcdplusmemon;
        size = 32'd16;
        is3way = 1'b1;
        sel = A_SEL;
        input_a = 16'h000f;
        input_b = 16'h00f1;
        input_c = 16'h0f02;
        #1
            // input_a should be on the output
            if(output_a != input_a) failure_counta++;
            if(output_b != input_a) failure_countb++;
            if(output_c != input_a[1:0]) failure_countc++;

        #5  size = 32'd16;
            is3way = 1'b1;
            sel = B_SEL;
        #1
            // input_b should be on the output
            if(output_a != input_b) failure_counta++;


        #5  size = 32'd16;
            is3way = 1'b1;
            sel = C_SEL;
        #1
            // input_c should be on the output
            if(output_a != input_c) failure_counta++;
            if(output_b != input_b) failure_countb++;
            if(output_c != input_b[1:0]) failure_countc++;
        #5
        $display("Number of unexpected results for a: %d", failure_counta);
        $display("Number of unexpected results for b: %d", failure_countb);
        $display("Number of unexpected results for c: %d", failure_countc);
        $finish;
    end
endmodule
```

../source/Verif/mux_tb.sv

VII. Build Scripts and utilities

```
#! /bin/bash
# $runsim [test bench]
# where acceptable selections of test benches are alu_tb, register_tb
# system_tb, mux_tb
# examlple $> ./runtim.sh system_tb
#
# Three step VCS flow as described in Synopsys user guide. Uses implicit configuration
# which allows unknown modules to be automatically resolved. See individual command
# comments for details. An important caveat of implicit configuration is packages and
# interfaces are not resolved by the search algorithm and file names must match .
#
# coverage analysis is enabled. Results can be viewed by running: dve -cov -dir simv.vdb/
# Command to run DVE: dve -vpd vcdplus.vpd
#
#

export VCS_LIC_EXPIRE_WARNING=1 #removes license expiry warning
mkdir logs lib #VCS will not create it's output directories if they don't exist

echo
echo
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo              RUNNING Vlogan
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo

vlogan -f vlogan_args.list

if [ $? -ne 0 ]; then
    echo "Vlogan analysis failed"
    exit 1;
fi

echo
echo
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo              RUNNING VCS
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo

vcs -file VCS_args.list $1

if [ $? -ne 0 ]; then
    echo "VCS elaboration failed"
    exit 1;
fi

echo
echo
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo              RUNNING Simulation
echo ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
echo
#
# Explanation of Command Line Flags:
```

```
# -cm fsm+line+tgl+branch record coverage metrics
# -cg_coverage_control=1 coverage data collection for all the coverage groups (not yet in code)
# -l log file directory
simv -l $PWD/logs/simv.log -cm fsm+line+tgl+branch -cg_coverage_control=1
```

../runsim.sh

```
//configuration
//=====================================================================================
//Because everyone knows it's the bext verilog
-sverilog

// -nc               : suppress Synopsys copyright message at beginning of log
-nc

+v2k

// +lint=all         : display all lint checks for code quality (noVCDE suppress messages about compiler directives)
+lint=all,noVCDE

// +warn=all         : always pay attention to warnings, they're there for a reason.
+warn=all

// -l <path>         : vlogan will direct it's output messages to this file
-l $PWD/logs/vlogan.log

//part of VCS implicit configuration. The top level file is the only
//module reqired to be imported (packages and interfaces wont be resolved)
//VCS will then search the -y directory for missing modules in file names
//that have the module name with one of the libext extentions
+libext+.sv+.v

//library directories VCS will search when looking for unresolved modules
//for implicit configuration Module name must match file name!!!!
-y $PWD/source/Design
-y $PWD/source/Verif

//packages that must be explicitly compiled(VCS implicit config isnt smart enough yet)
//=====================================================================================
//Design packages
$PWD/source/Design/alu_pkg.sv
$PWD/source/Design/types_pkg.sv

//Top level files
$PWD/source/Verif/system_tb.sv
$PWD/source/Verif/alu_tb.sv
$PWD/source/Verif/register_tb.sv
```

../vlogan_args.list

```
//
// VCS configuration file
//

//enables post process debug utilities
-PP

//VCS will build interactive debug capability into the simv executable
-debug_all

// Enables coverage metrics which tells what parts of the code have been exercised
// FSM - Which states of finite state machines have been used
// line - which lines of code have been used by test run
// tgl - records which signals have been toggled in test rub
// branch - which parts of if branches have been taken (superfluous with line?)
-cm fsm+line+tgl+branch

//initialize all memory elements with random data at sim start
+vcs+initreg+random

//enables system verilog
-sverilog

//REALLY verbose warning messages
-notice

//check
-xzcheck nofalseneg

//Always listen to lint... always
+lint=all

//check for race conditions in TB assignments.. we like our sims nice and deterministic
-race

//suppress synopsys copywright message
-q

//put log file in log folder because we're civilzed here.
-l $PWD/logs/VCS.log
```

../VCS_args.list