CPE142: COMPUTER ORGANIZATION

NOVEMBER THIRTEENTH 2014

Term Project: Phase One

Authors:
Ben SMITH 55%
Devin MOORE 45 %

Instructor:
Dr. Behnam ARAD

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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.

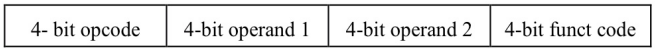


Fig. 1: A Type Instruction Format

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.

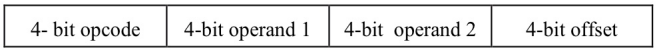


Fig. 2: B Type Instruction Format

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.

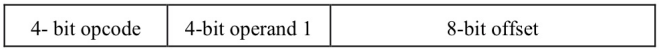


Fig. 3: C Type Instruction Format

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.

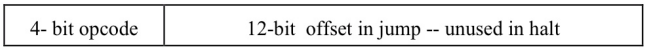


Fig. 4: D Type Instruction Format

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

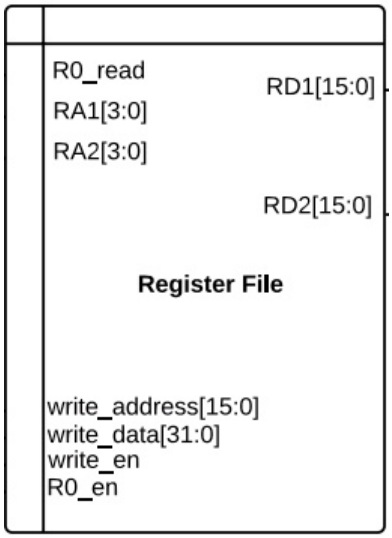


Fig. 5: Register File Block

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

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

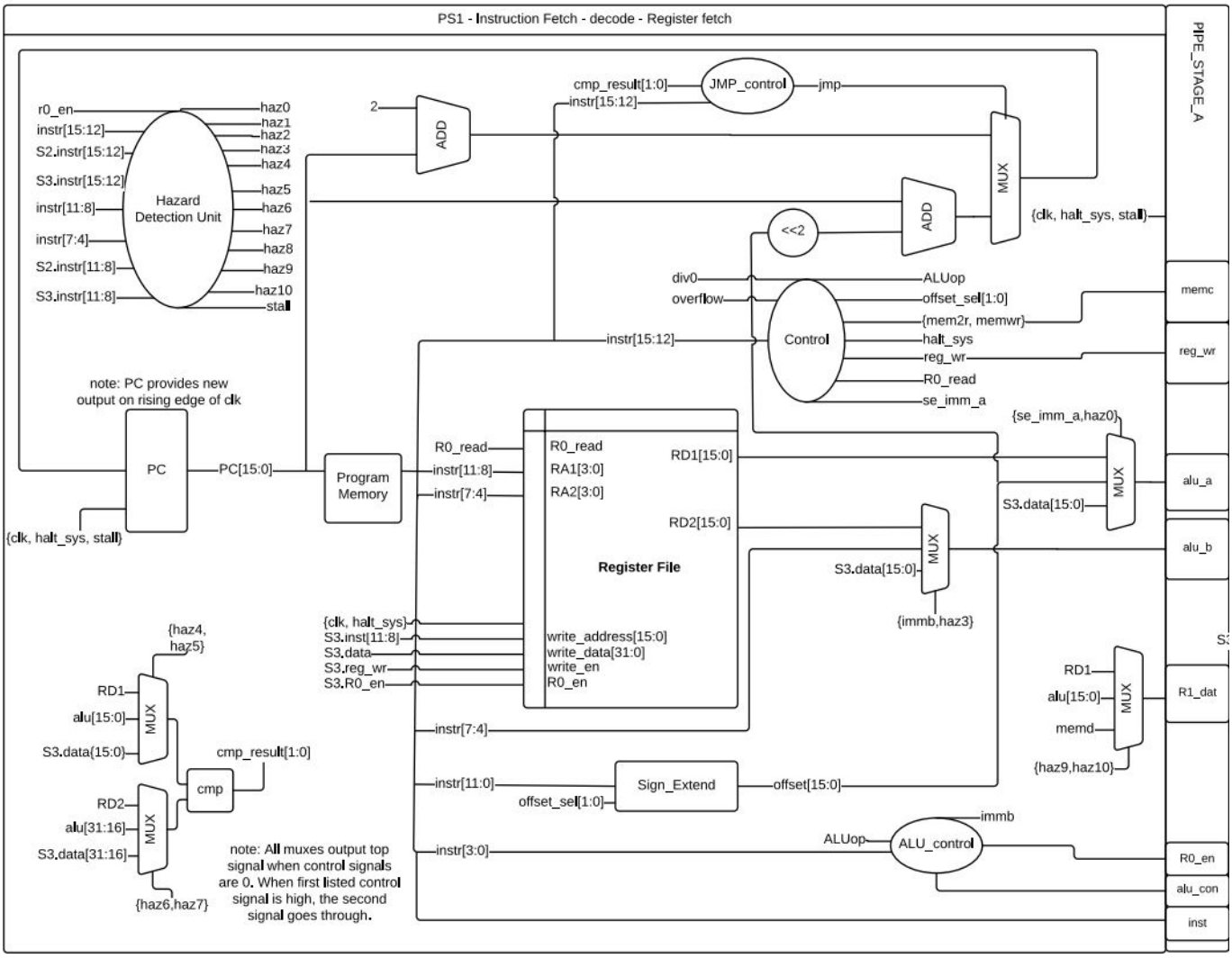


Fig. 6: Pipeline Organization Stage One

D. Stage 2

The entire second stage of this design belongs to the Arithmetic Logic Unit as seen in Figure 7. 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 8. The main logic of this stage is contained in the main memory unit which is described in section two of this document.

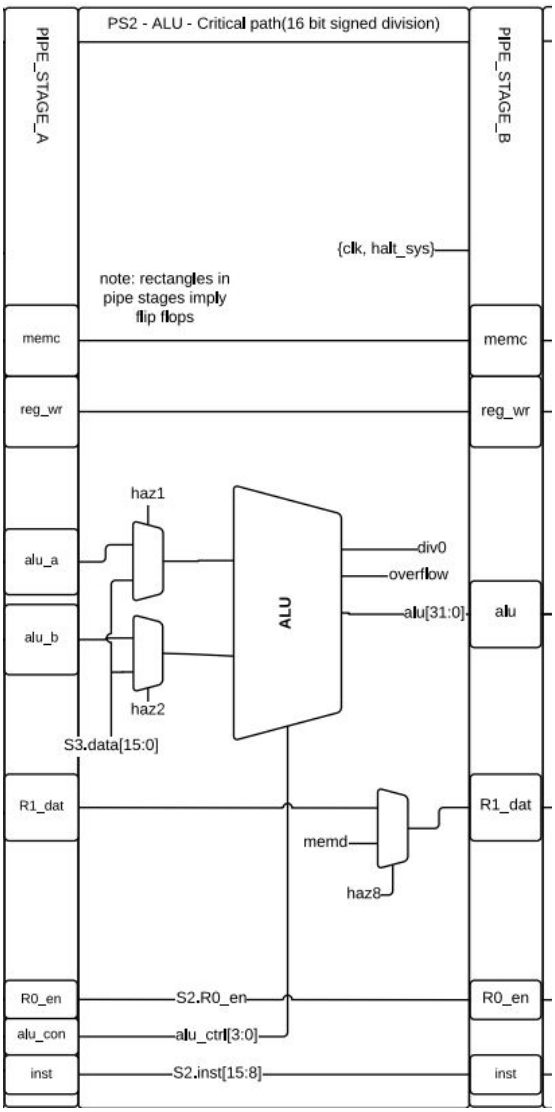


Fig. 7: Pipeline Organization Stage Two

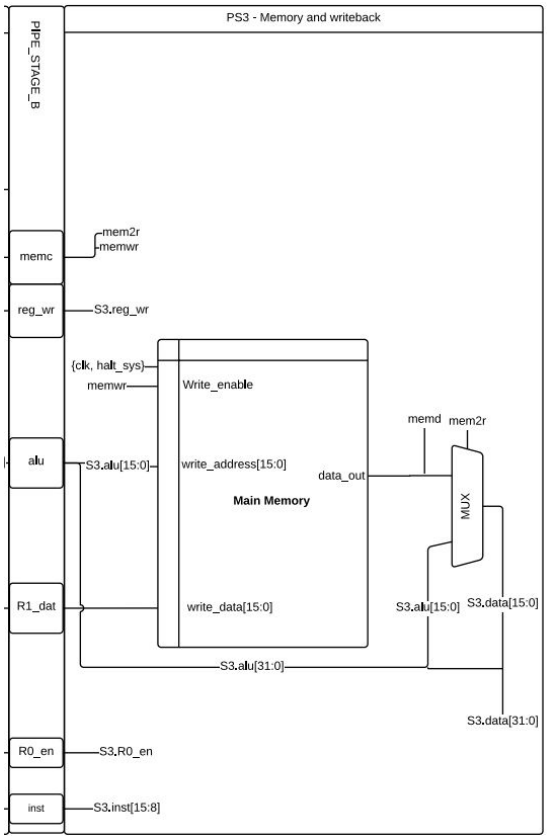
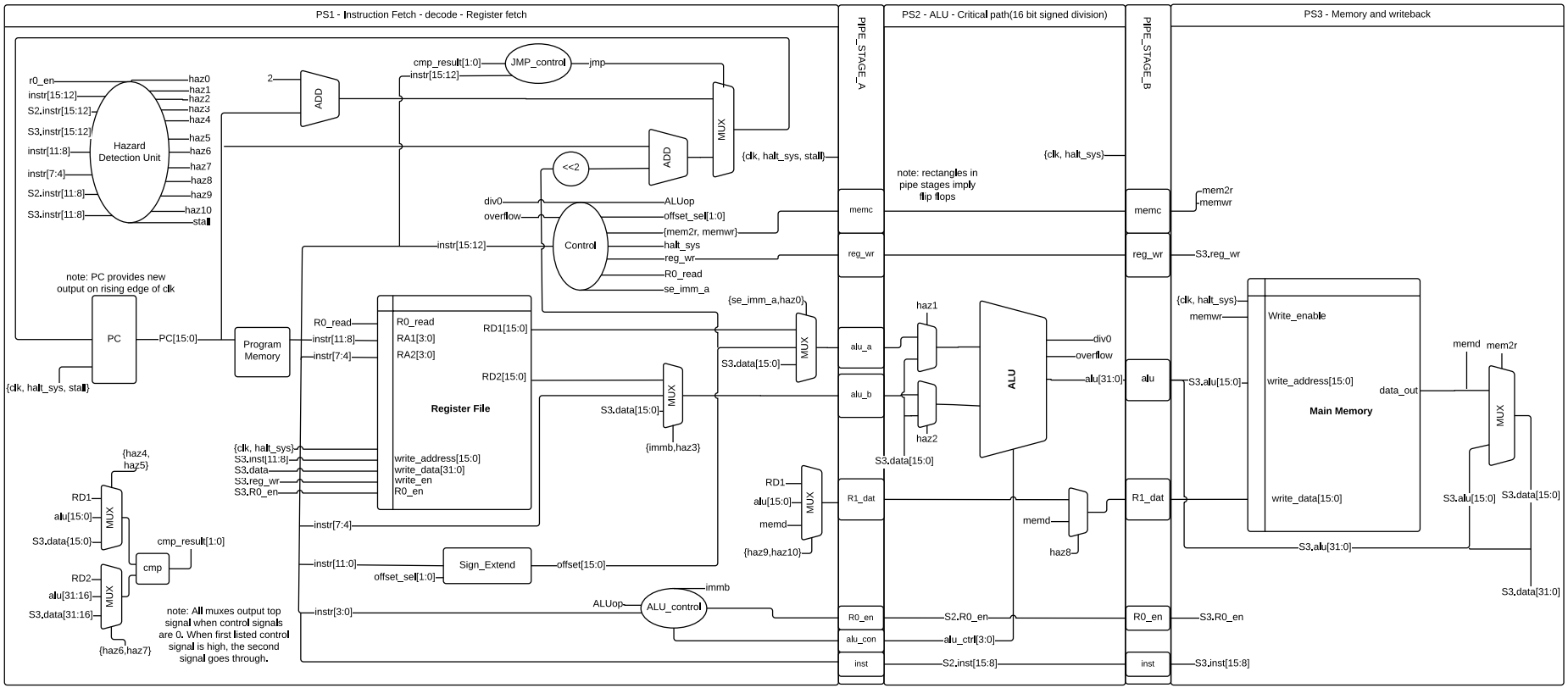


Fig. 8: Pipeline Organization Stage Three


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