Processor Looper

Team *Loopers*

Fan Zhu

Haoyan Jia (Team Leader)

Jing Tu

Junjue Wang

Yuewen Lei

Zheng Ling

Zhexuan Liu

ECE554/901 Final Report

University of Wisconsin-Madison

Spring 2014

Table of Contents

1. Introduction

In this project, we designed an out-of-order superscalar processor called Looper. Looper is a 4-issue superscalar, out-of-order execution, 7-stage pipelined processor, with special designs inspired by Mitchell Hayenga's Revolver architecture, including frontend loop detection, training and dispatch, and multiple backend supporting schemes in issue queue, re-order buffer and load/store queue.

The system runs with a 16-bit specifically constructed ISA, 16 16-bit logical registers, 64 16-bit physical registers, and separate instruction memory and data memory with 16K entries and 64-bit in width. It is also supported with a dynamic branch predictor, a loop dispatch unit, and loop-aware modifications in matrix scheduler, load store queue and commit logic. We also implemented a Memory Manage Unit to communicate with the processor and the host machine, evaluate the overall performance and show the results for demo.

This is the first time in this course that the students work on an out-of-order processor with a timely research topic. This is also the first time that the students are so ambitious to implement a 4-issue superscalar with so many specially designed features. All the team members knew that this would be a tough challenge to all of us, but we accepted the challenge without hesitation. However, the workload of the whole project is not fully analyzed. Only the normal features of 4-issue and multiple branch prediction are already complex enough for a three months project. Therefore at the end of the semester, we accomplished to simulate all the functionality correctly in normal mode, but could not get enough time to fully test the loop mode. We had already included most of the design for loop mode in the code, especially in ID, IS, and WB stage, but we just had no time to test it out.

Anyway, we are proud of what we have accomplished through this semester. And we sincerely appreciate all the help we got from Prof. Lipasti, Ashish, Zhenhong and Vignyan, without whom we could never achieve what we have done today. And thankfully, we consolidated our knowledge in computer architecture through this valuable experience, and gained a priceless friendship between all the team members.

2. Instruction Set Architecture

The 16 bit instruction set architecture is listed below:

Instruction Format	Operation	Syntax	Semantics
0000 0000 0000 0000	NOP	nop	No operation
0001 dddd ssss tttt	ADD	add rd, rs, rt	rd <- rs + rt
0010 dddd ssss tttt	SUB	sub rd, rs, rt	rd <- rs - rt
0011 dddd ssss tttt	AND	and rd, rs, rt	rd <- rs & rt
0100 dddd ssss tttt	OR	or rd, rs, rt	rd <- rs | rt
0101 dddd ssss tttt	XOR	xor rd, rs, rt	rd <- rs ^ rt
0110 dddd ssss 0000	NOT	not rd, rs	rd <- ~rs
0111 dddd ssss tttt	SRA	sra rd, rs, rt	rd <- rs >> rt (sign extension)
1000 dddd ssss tttt	MUL	mult rd, rs, rt	rd <- rs * rt
1001 ssss iiiiiiii	BEQZ	beqz rs, immediate	if (rs == 0) PC <- PC + 1 + (sign extension) Imm
1010 ssss iiiiiiii	BLTZ	bltz rs, immediate	if (rs < 0) PC <- PC + 1 + (sign extension) Imm
1011 ssss iiiiiiii	BGTZ	bgtz rs, immediate	if (rs > 0) PC <- PC + 1 + (sign extension) Imm
1100 dddd iiiiiiii	LDI	ldi rd, immediate	rd <- Imm (sign extension)
1101 dddd ssss iiii	STR	str rd, rs, immediate	mem[rs + immediate] <- rd
1110 dddd ssss iiii	LDR	ld rd, rs, immediate	rd <- mem[rs + immediate]
1111 iiiiiiiiii 00	J	j displacement	PC <- PC + 1 + Imm (sign exten)
1111 ssss iiiiii 01	JR	jr, rs, immediate	PC <- rs + Imm(sign exten)
1111 iiiiiiiiii 10	JAL	jal displacement	R15 <- PC + 1; PC <- PC + 1 + I(sign ext.)

2.1. Addressing Modes

Our processor supports four addressing modes: Register direct, PC-relative, Immediate and Base-offset.

All arithmetic and logical instructions including ADD, SUB, AND, OR, XOR, NOT, SRA and MUL use register direct addressing mode.

All branch instruction and some jump instructions like J and JAL use PC-relative addressing mode.

LDI instruction uses immediate addressing mode.

LDR/STR instruction and the other jump instructions like JR and JALR use Base-offset addressing mode.

2.1.1. Register direct addressing mode

This is the basic addressing mode we used. We use bits[15:12] as our opcode, followed by 4 bits representing for destination register and the other 8 bits used to represent two source register.

Syntax: Opcode[15:12] Rd, Rs, Rt
Example: ADD Rd, Rs,Rt
Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Rd				Rs				Rt			

2.1.2. PC-relative addressing mode

This addressing mode is mostly used for branch instructions. Also there are two jump instructions use this addressing mode. For branch instruction, bits[7:0] represent for the immediate value. For jump instruction, bits[11:2] represent for the immediate value. All immediate value will be sign extended to 16 bits.

Case 1: Branch Instruction
Syntax: Opecode[15:12], Rd, Immediate[7:0]
Example: BEQ Rd, Immediate
Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Rs				Immediate							

Case 2: Jump Instrcution
Syntax: Opecode[15:12], immediate[11:2], mode[1:0]
Example: Jump Immediate 00
Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Immediate										mode	

2.1.3. Immediate addressing mode

Only LDI instruct will use this addressing mode. Bits [7:0] represent for the immediate value. The immediate value will be sign extended.

Syntax: Opcode[15:12] Rd, Immediate
Example: LDI Rd, Immediate
Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Rs				Immediate							

2.1.4. Base-offset addressing mode

Both load and store instructions will use this addressing mode. There are also two other jump instructions use this addressing mode. For load/store instructions, bits[3:0] represent for immediate value. For jump instructions, bit [7:2] represent for immediate value. All immediate value will be sign extended.

Case 1: load/store instructions
Syntax: Opecode[15:12] Rd, Rs, Immediate

Example: STR Rd, Rs, Immediate

Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Rd				Rs				Immediate			

Case 2: jump instructions

Syntax: Opcode[15:12] Rs, Immediate, mode[1:0]

Example: Jump Rs, Immediate, 01

Encoding:

Bits	11	10	9	8	7	6	5	4	3	2	1	0
Contents	Rs				Immediate						mode	

2.2. Instruction Descriptions

2.2.1. NOP

Syntax:

NOP

Pseudo Code:

No operation

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Usage and Examples:

No operation will be done by this instruction. It is used for flushing and stall to substitute original instruction.

2.2.2. ADD

Syntax:

ADD <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs + Rt

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	0	1	Rd				Rs				Rt			

Usage and Examples:

Arithmetic add instruction. Load value from register Rs and Rt, do the addition Rs plus Rt and store the result into Rd. Immediate value addition can be done by first call LDI to load immediate value into register and then do an addition.

2.2.3. SUBTRACT

Syntax:

SUB <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs - Rt

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	1	0	Rd				Rs				Rt			

Usage and Examples:

Arithmetic subtract instruction. Load value from register Rs and Rt, do the subtraction Rs minus Rt and store the result into Rd. Immediate value subtraction can be done by first call LDI to load immediate value into register and then do an subtraction.

2.2.4. AND

Syntax:

AND <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs & Rt

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	0	1	1	Rd				Rs				Rt			

Usage and Examples:

Logical AND instruction. Load value from register Rs and Rt, do the Rs AND Rt operation and store the result into Rd. Immediate value AND can be done by first call LDI to load immediate value into register and then do AND operation.

2.2.5. OR

Syntax:

OR <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs | Rt

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	1	0	0	Rd				Rs				Rt			

Usage and Examples:

Logical OR instruction. Load value from register Rs and Rt, do the Rs OR Rt operation and store the result into Rd. Immediate value OR can be done by first call LDI to load immediate value into register and then do OR operation.

2.2.6. XOR

Syntax:

XOR <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs ^ Rt

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	1	0	1	Rd				Rs				Rt			

Usage and Examples:

Logical XOR instruction. Load value from register Rs and Rt, do the Rs XOR Rt operation and store the result into Rd. Immediate value XOR can be done by first call LDI to load immediate value into register and then do XOR operation.

2.2.7. NOT

Syntax:

NOT <Rd>, <Rs>

Pseudo Code:

Rd = ~Rs

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	1	1	0	Rd				Rs				0	0	0	0

Usage and Examples:

Logical NOT instruction. Load value from register Rs, invert the value in Rs and store the result into Rd. Immediate value NOT can be done by first call LDI to load immediate value into register and then do NOT operation.

2.2.8. SHIFT RIGHT ARITHMETIC

Syntax:

SRA <Rd>, <Rs>, <Rt>

Pseudo Code:

Rd = Rs >> Rt (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	0	1	1	1	Rd				Rs				Rt			

Usage and Examples:

Arithmetic shift register instruction. Value need to be shifted is load from Rs and number of bits to shift is load from Rt. The result is store into Rd. Immediate value shift can be done by first call LDI to load immediate value into register and then do SRA operation.

2.2.9. MULTIPLY

Syntax:

MULT <Rd>, <Rs>, <Rt>

Pseudo Code:

*Rd = Rs * Rt*

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	0	0	0	Rd				Rs				Rt			

Usage and Examples:

Arithmetic multiply instruction. Load value from register Rs and Rt, do the multiplication Rs times Rt and store the result into Rd. (Note: If the result is more than 16 bit, only 16 least significant bits will be stored into Rd). Immediate value subtraction can be done by first call LDI to load immediate value into register and then do an subtraction.

2.2.10. BRANCH EQUAL TO ZERO

Syntax:

BEQZ <Rs>, <Immediate>

Pseudo Code:

If(Rs == 0) then PC = PC + 1 + Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	0	0	1	Rs				Immediate							

Usage and Examples:

Conditional branch instruction. If the value store in Rs is equal to 0, the processor will do a branch and set PC value to PC + 1 + (sign extension) Immediate. (Note: The reason we make our branch instruction compare to 0 is to save more bits for immediate value. Otherwise we have only 4 bits to represent for immediate value so that we can only reach address from -8 to +7. The region is too small).

2.2.11. BRANCH LESS THAN ZERO

Syntax:

BLTZ <Rs>, <Immediate>

Pseudo Code:

If(Rs < 0) then PC = PC + 1 + Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	0	1	0	Rs				Immediate							

Usage and Examples:

Conditional branch instruction. If the value store in Rs is less than 0, the processor will do a branch and set PC value to PC + 1 + (sign extension) Immediate.

2.2.12. BRANCH GREAT THAN ZERO

Syntax:

BGTZ <Rs>, <Immediate>

Pseudo Code:

If(Rs > 0) then PC = PC + 1 + Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	0	1	1	Rs				Immediate							

Usage and Examples:

Conditional branch instruction. If the value store in Rs is larger than 0, the processor will do a branch and set PC value to PC + 1 + (sign exten) Immediate.

2.2.13. LOAD IMMEDIATE

Syntax:

LDI <Rd>, <Immediate>

Pseudo Code:

Rd = Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	0	0	Rd				Immediate							

Usage and Examples:

Load immediate instruction. Load the sign extended immediate value into register Rd.

2.2.14. STORE

Syntax:

STR <Rd>, <Rs>, <Immediate>

Pseudo Code:

Mem[Rs + Immediate (sign extension)] = Rd

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	0	1	Rd				Rs				Immediate			

Usage and Examples:

Store instruction. Store the value in register Rd into memory. Memory address is the value stored in register Rs add sign extended immediate value.

2.2.15. LOAD

Syntax:

LDR <Rd>, <Rs>, <Immediate>

Pseudo Code:

Rd = Mem[Rs + Immediate (sign extension)]

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	1	0	Rd				Rs				Immediate			

Usage and Examples:

Load instruction. Load the value into register Rd from memory. Memory address is the value stored in register Rs add sign extended immediate value.

2.2.16. JUMP

Syntax:

J <Immediate>

Pseudo Code:

PC = PC + 1 + Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	1	1	Immediate										0	0

Usage and Examples:

Jump instruction, this is jump instruction use PC-relative addressing mode. It will set the PC address to PC + 1 + (sign extension) immediate

2.2.17. JUMP BASE REGISTER

Syntax:

JR <Rs>, <Immediate>

Pseudo Code:

PC = Rs + Immediate (sign extension)

Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	1	1	Rs				Immediate						0	1

Usage and Examples:

Jump instruction, this is jump instruction use base-offset addressing mode. It will set the PC address to Rs + (sign extension) immediate.

2.2.18. JUMP FUNCTION CALL

Syntax:

JAL <Immediate>

Pseudo Code:

R15 = PC + 1

PC = PC + 1 + Immediate (sign extension)

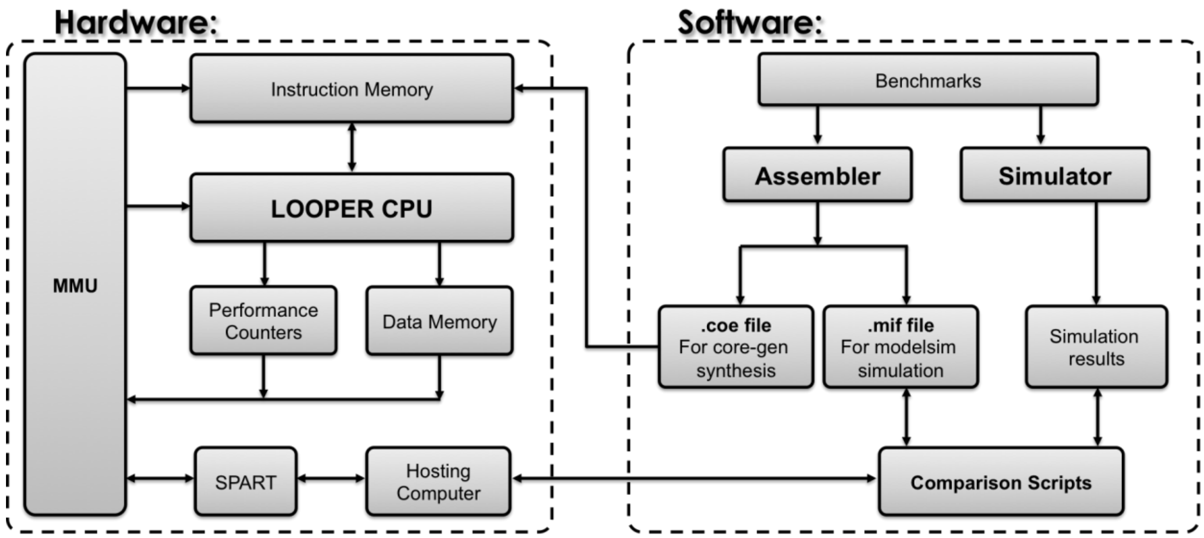
Encoding:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Contents	1	1	1	1	Immediate										1	0

Usage and Examples:

Jump instruction, this is jump instruction use PC-relative addressing mode and it will set return address into R15. This jump instruction can be used for function call. It will set the PC address to PC + 1 + (sign extension) immediate.

3. System Overview



The whole Looper project system is composed of the hardware part and the software part, which we will talk about with more detail in the later chapters.

Here we want to emphasize the Memory Manage Unit, which enables Looper CPU to communicate with host machine more easily. It comprises a driver, a full-duplex SPART and a DVI controller. Below is a list of functions of MMU:

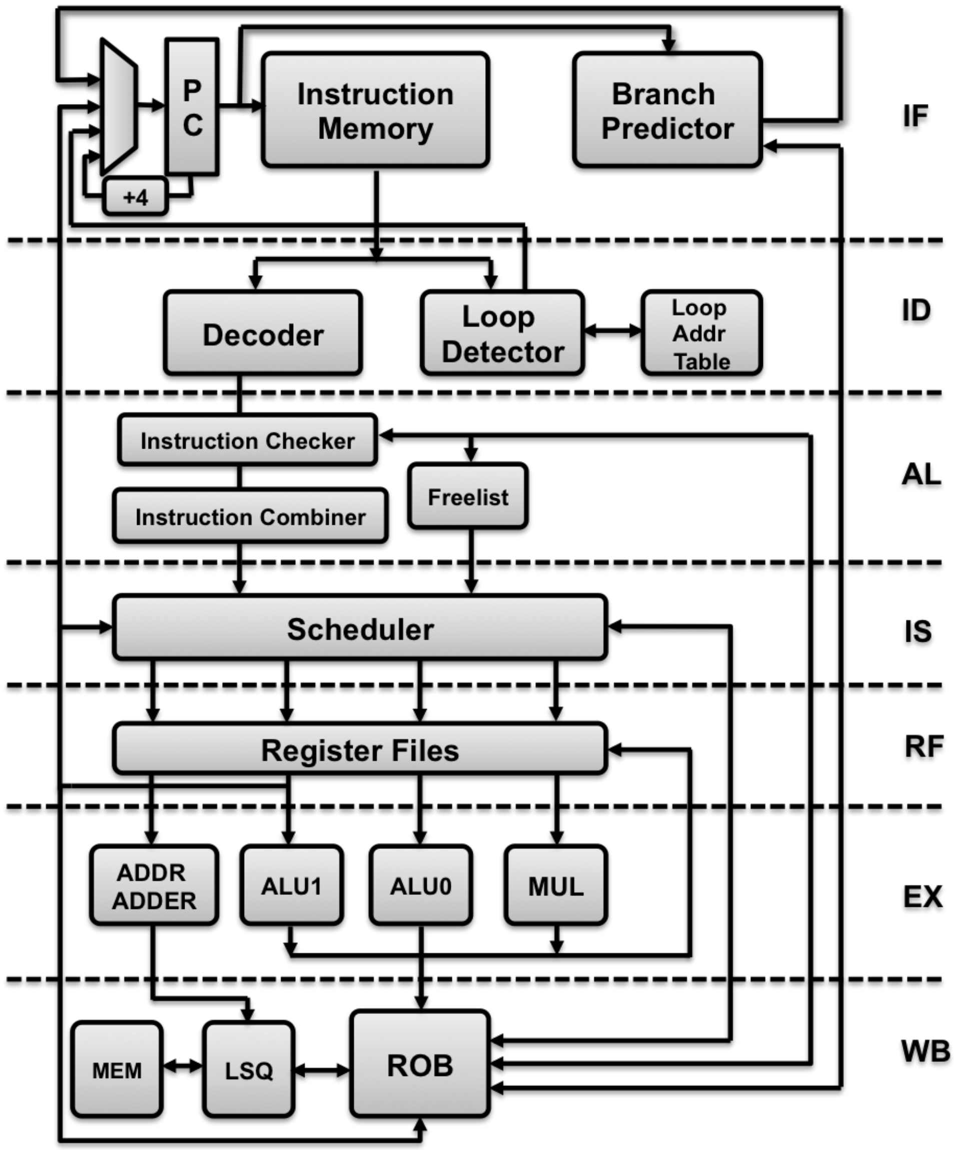
1. Receive commands using serial interface from host machine.
2. Access CPU data memory and transmit interested region back to host machine.
3. Change program counter's value to force Looper CPU to start executing specific benchmarks.
4. Display predefined region in CPU data memory through DVI interface.
5. Record total number of cycles executed, instructions sent by FETCH and ISSUE stage, mispredictions and total number of branch instructions executed.

These functions of MMU empowers designers to run individual benchmark, log and verify results at host machine, display image processing results and evaluate performances.

As to the benchmark switching function, since the benchmarks that we are going to use are all quite smaller than the total instruction memory size, we plan to put all the benchmarks into the IM together, and then keep the start address of each of them. Also, we leave the instruction at address 0 as a jump to address 0 instruction, so that when the CPU starts and when any benchmark finishes, we can be in an idle looping state. Therefore, when we want to start any of the benchmarks, the MMU will change the PC to the start address of that benchmark. Also, when a benchmark finishes, the last instruction will be jumping to address 0, i.e. into the idle loop. Then, several cycles later, when the re-order buffer is empty, which means the designated benchmark is totally finished, we can read the performance counters and computation results.

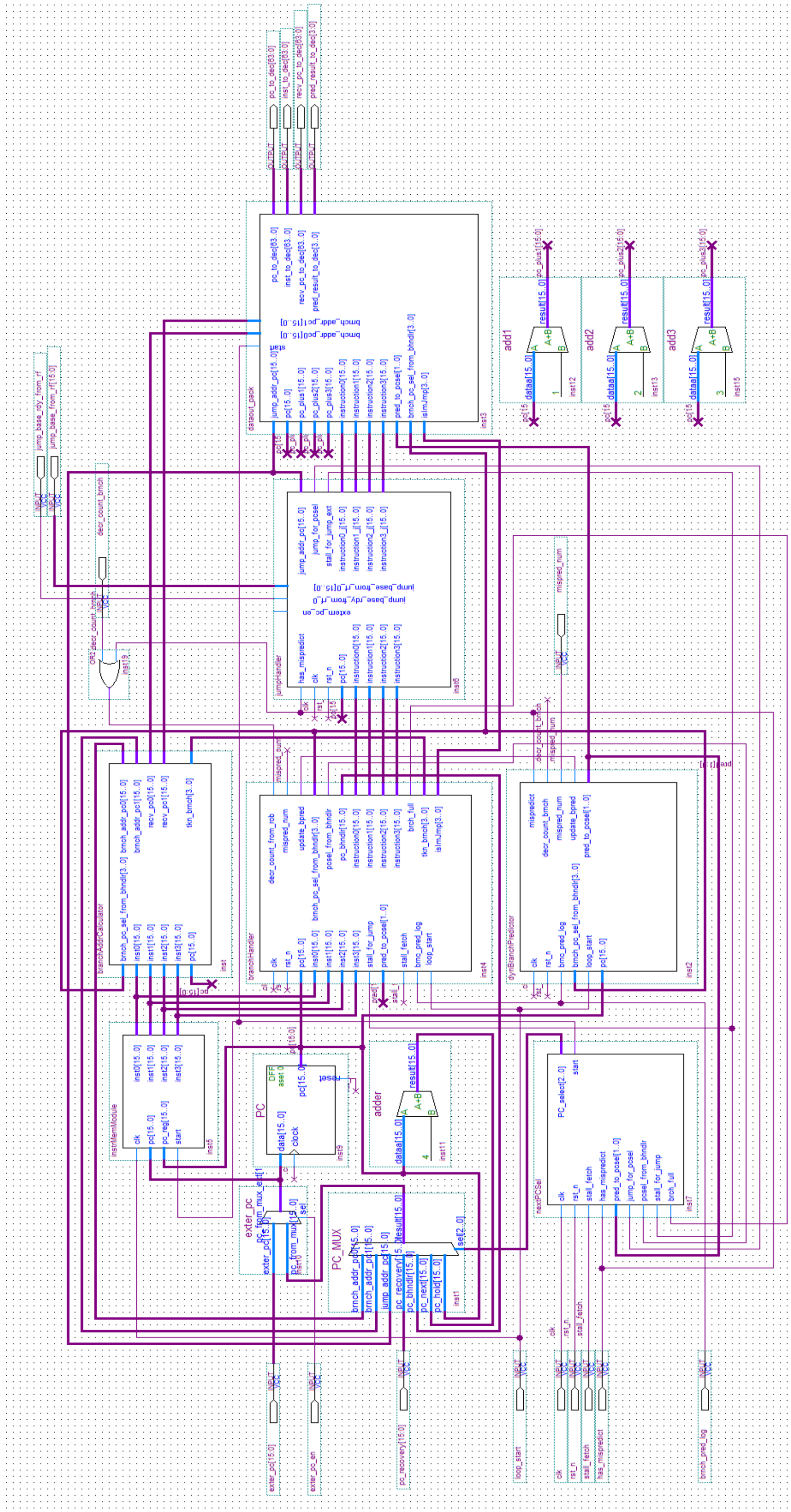
Also, the functionality of the MMU will be controlled by the host computer through the SPART port. And the performance and results will also be evaluated on the host computer.

4. Hardware Components



The Looper will be a seven-stage pipeline out-of-order superscalar machine, including instruction fetch(IF), instruction decode(ID), allocation(AL), issue(IS), register file(RF), execution(EX) and write back(WB) stages.

4.1. Instruction Fetch (IF)



The instruction fetch stage is in charge of fetching four consecutive instructions per clock cycle and detecting and calculating branch or jump address if there is a branch or jump. The interface table for instruction fetch is shown below.

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
Rst_n	I	1		Reset pc counter
stall_fetch	I	1	LAT	When in loop mode LAT unit in decoder stage output stall_fetch=1 to halt input, and PC hold its value until this signal becomes 0.
loop_start	I	1	LAT	Loop_start informs fetch stage to disable branch counter and ensure branch predictor always predicts taken
decr_count_brnch	I	1	Reorder Buffer	When a branch instruction commits from ROB, this signal is set to inform fetch stage
has_mispredict	I	1	Reorder Buffer (PR stage)	At physical register stage, when the reorder buffer detects branch prediction, it outputs this signal flush=1 to reset PC
mispred_num	I	1	Reorder Buffer	
brnc_pred_log	I	1	Reorder Buffer	
pc_recovery	I	16	Reorder Buffer (commit stage)	At commit stage, when the reorder buffer detects branch prediction, it send the recovery PC address to PC-MUX
jump_base_from_rf	I	16	RF stage	Jump addresses are calculated at RF stage and sent back to jump handler in fetch stage to calculate jump address
Jump_base_rdy_from_rf	I	1	RF stage	
exter_pc	I	16	MMU	MMU uses this port to control the program counter jump to specific address
exter_pc_en	I	1	MMU	input port for program start address
pc_to_dec	O	64	Decoder	64 bits of all four PCs
inst_to_dec	O	64	Decoder	64 bits of all four instructions
recv_pc_to_dec	O	64	Decoder	64 bits of recovery pc addresses for branch and jump instructions, if not branch/jump give all zero
pred_result_to_dec	O	4	Decoder	4b bits of prediction result for branch instructions

In the Instruction fetch stage, there are mainly seven function blocks: PC mux, instruction memory, branch handler, branch predictor and address calculator, jump handler, next PC selector and dataout_pack module.

4.1.1. Function Blocks

4.1.1.1. PC MUX

Signal Name	I/O	Width	Source/Target (blocks)	Description
pc_hold	I	16	PC register	current pc
pc_recovery	I	16	ROB	At commit stage, when the reorder buffer detects branch prediction, it send the recovery PC address to PC-MUX
brnch_addr_pc0	I	16	Branch Address Calculator	Branch address is selected to be next pc when branch is predicted taken
brnch_addr_pc1	I	16	Branch Address Calculator	Branch address is selected to be next pc when branch is predicted taken
jump_addr_pc	I	16	jump handler	Jump address is stored in jump address buffer inside jump handler
pc_next	I	16	IM	Update pc to fetch the next instruction
pc_bhndlr	I	16	Branch Handler	Select signal for the mux
PC_select	I	3	next PC Selector	
pc	O	16	pc register	Output to pc register, then passed to following modules
Signal Name	I/O	Width	Source/Target (blocks)	Description
pc_hold	I	16	PC register	current pc
pc_recovery	I	16	ROB	At commit stage, when the reorder buffer detects branch prediction, it send the recovery PC address to PC-MUX
brnch_addr_pc0	I	16	Branch Address Calculator	Branch address is selected to be next pc when branch is predicted taken
brnch_addr_pc1	I	16	Branch Address Calculator	Branch address is selected to be next pc when branch is predicted taken
jump_addr_pc	I	16	jump handler	Jump address is stored in jump address buffer inside jump handler
pc_next	I	16	IM	Update pc to fetch the next instruction
pc_bhndlr	I	16	Branch Handler	Select signal for the mux
PC_select	I	3	next PC Selector	
pc	O	16	pc register	Output to pc register, then passed to following modules

PC MUX is a 7-to-1 multiplexer with inputs as possible next PC addresses. The PC select signal is given by next PC selector to choose from these PC values to update the PC register at each clock posedge.

4.1.1.2. Instruction Memory

Signal Name	I/O	Width	Source/Target (blocks)	Description

Clk	I	1		
pc	I	16	PC MUX	Input to instruction memory, use this because there is one clock cycle delay for instruction memory
pc_reg	I	16	PC register	input to instruction memory module to select four instructions out of the two memory lines
start	I	1	next PC selector	start signal to enable reading data from instruction memory when reset is done
inst0	O	16	Branch Handler	
inst1	O	16	Branch Handler	
inst2	O	16	Branch Handler	
inst3	O	16	Branch Handler	

Instruction Memory module includes instruction memory and other hardware so that four consecutive instructions can be fetched and output in order. The instruction memory is designed to have 4 set. At each pc input, it will output 64 bits of memory data entry which is 4 instructions together. To avoid misaligned instruction fetch, we have designed to fetch two memory lines each time and use a mux tree to select 4 out of eight instructions.

4.1.1.3. Branch Handler

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
rst_n	I	1		
inst0	I	16	IM	
inst1	I	16	IM	
inst2	I	16	IM	
inst3	I	16	IM	
Stall_for_jump	I	1	Jump handler	If register based jump, need to stall fetching for certain cycles
loop_start	I	1	decode stage	inform branch handler to disable branch counter
stall_fetch	I	1	decode stage	NOP instructions in this module so that instructions are not passed to next stage
pred_to_pcsel	I	2	Branch Predictor	
decr_count_from_rob	I	1	Reorder Buffer	inform branch handler to decrement branch counter. This signal is set high when either a branch commits or a misprediction occurs.
mispred_num	I	1	Reorder Buffer	indicate which branch is mispredicted, and corresponding number should be deducted from branch counter.

brnc_pred_log	I	1	Reorder Buffer	On misprediction, remind branch handler the wrong prediction value
update_bpred	O	1	Branch Predictor	enables branch predictor to take in instructions
brnch_pc_sel_from_bhndlr	O	4	Branch Predictor	pick branch instructions from the four instructions
pc_bhndlr	O	16	PC MUX	output to PC the next instruction to fetch if it got replaced by nop
pcsel_from_bhndlr	O	1	next PC Selector	output to next PC selector
instruction0	O	16	jump handler&dataout_pack	instruction after branch hander
instruction1	O	16	jump handler&dataout_pack	instruction after branch hander
instruction2	O	16	jump handler&dataout_pack	instruction after branch hander
instruction3	O	16	jump handler&dataout_pack	instruction after branch hander
brnch_inst0	O	16	branch address calculator	one of the branch instructions
brnch_inst1	O	16	branch address calculator	one of the branch instructions
brch_full	O	1	next PC selector	to inform PC selector stop fetching new pc because branch counter is full
tkn_brnch	O	4	branchAddrCalculator	Indicate which branch need to predict for
islmJmp	O	4	dataout_pack	NOP immediate jump instruction at output

This module is uniquely designed for our process for controlling of number of branch instructions fetched each time. Too many branch instructions will exert burden on our branch predictor and create difficulty for updating the predictor pattern on misprediction. Because of this, we designed a Branch Handler to limit the number of branch instructions fetched per clock cycle within a number of two. If there are already two branch instructions fetched, we replace the third branch and subsequent instructions with NOP instruction and update the next PC to the PC of the third branch instruction.

4.1.1.4. Branch Predictor and address calculator

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
rst_n	I	1		
decr_count_brnch	I	1	ROB	when a branch is committed update predictor counter
mispredict	I	1	ROB	indicate a misprediction took place

mispred_num	I	1	ROB	indicate which prediction was wrong
brnc_pred_log	I	1	ROB	use this value to decide how to invert predictor counter
brnch_pc_sel_from_bhndlr	I	4	Branch Handler	choose which of the 4 instructions are branch instruction
update_bpred	I	1	Branch Handler	indicate there exist branch instr, the bpred need to be updated
pc	I	16	IM	current pc to use in predictor for generating addr in PHT
pc_plus1	I	16	IM	current pc+1 to use in predictor for generating addr in PHT
pc_plus2	I	16	IM	current pc+2 to use in predictor for generating addr in PHT
pc_plus3	I	16	IM	current pc+3 to use in predictor for generating addr in PHT
pred_to_pcsel	O	2	next PC Selector	indicate the branch taken/not taken for two branches

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
brnch_pc_sel_from_bhndlr	I	4	Branch Handler	choose which pc of the instructions to calculate addr for
tkn_brnch	I	4	Branch Handler	Indicate which branch instruction to calculate branch address for
pc	I	16	PC register	current PC
brnch_addr_pc0	O	16	PC MUX & dataout_pack	branch address for inst 0, pass to next pc and the next stages
brnch_addr_pc1	O	16	PC MUX & dataout_pack	branch address for inst 1,pass to next pc and next stages
recv_pc0	O	16	dataout_pack	record recovery pc for the branch in case of misprediction
recv_pc0	O	16	dataout_pack	record recovery pc for the branch in case of misprediction

Branch predictor and branch address calculator cooperate with each other to calculator the branch target address for two or less instructions and output them to PC MUX. They also generate selection signal to next PC selector to choose which branch target address to jump to according to prediction result.

4.1.1.5. Jump Handler

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		

rst_n	I	1		
has_mispredict	I	1	Reorder buffer	Clear wait for jump signal when mispredict occurs
Instruction1	I	16	Branch Handler	choose which instruction to calculate addr for
Instruction2	I	16	Branch Handler	
Instruction3	I	16	Branch Handler	
Instruction4	I	16	Branch Handler	
extern_pc_en	I	1	MMU	enable external PC, disable current jumps
jump _base_from_rf	I	16	RF stage	from RF stage, address is calculated in fetch stage
Jump_base_rdy_from_rf	I	1	RF stage	From RF stage
Jump_addr_pc	O	16	PC MUX	Can be obtained from rf stage and computed in fetch stage or from instruction immediate value
jump_for_pcsel				
	O	1	Next PC Selector	
Stall_for_jump_ext	O	1	Branch Handler&next PC selector	If register based jump, need to stall fetching for certain cycles
instruction1_j	O	16	dataout_pack	
instruction2_j	O	16	dataout_pack	
instruction3_j	O	16	dataout_pack	
instruction4_j	O	16	dataout_pack	

Jump handler is responsible for processing jump instructions. When there is a jump instruction, it first checks if it is register-based jump or immediate jump. If it is register-based jump, stall_for_jump signal should be set to 1 and keep the value until jump_base_rdy_from_rf signal is high. Jump_base_rdy high indicate that the value of jump base register has been read from register file stage. Meanwhile, stall_for_jump signal is sent to Branch Handler module. On receiving the signal, Branch Handler perform stall by output NOPs on 4 instructions.

4.1.1.6. Next PC Selector

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
rst_n	I	1		
stall fetch	I	1	DECODE stage	from decode stage, if in loop mode
has_mispreditct	I	1	ROB	from RF stage, if branch mispredicted
pred_to_pcsel	I	2	Branch Predictor	
jump_for_pcsel	I	1	jump handler	
pcsel_from_bhndlr	I	1	Branch Handler	If instruction got replaced by NOP
stall_for_jump	I	1	jumpHandler	If waiting for a register-based jump,

				stall the next PC
brch_full	I	1	branchHandler	When a third branch is detected, this signal is sent to stop fetching new PC
PC_select	O	1	PC MUX	
start	O	1	instrMemModule & dataout_pack	prevent instructions being read and send to the next stage when pc data is not ready

4.1.1.7. dataout_pack module

Signal Name	I/O	Width	Source/Target (blocks)	Description
start	I	1	next PC selector	start=1 indicate the currently fetch instructions are not valid
pc	I	16	current pc	
pc_plus1	I	16		
pc_plus2	I	16		
pc_plus3	I	16		
instruction0	I	16	Jump Handler	instruction after jump hander
instruction1	I	16	Jump Handler	instruction after jump hander
instruction2	I	16	Jump Handler	instruction after jump hander
instruction3	I	16	Jump Handler	instruction after jump hander
pred_to_pcsel	I	2	Branch Predictor	
jump_addr_pc	I	16	jump handler	
brnch_addr_pc0	I	16	Branch Address Calculator	
brnch_addr_pc1	I	16	Branch Address Calculator	
brnch_pc_sel_from_bhndlr	I	4	Branch Handler	
isImJmp	I	4	Branch Handler	tell dataout_pack to NOP instruction output to next stage
pc_to_dec	O	64	DECODE stage	
inst_to_dec	O	64	DECODE stage	
recv_pc_to_dec	O	64	DECODE stage	recovery address for 4 brnch/jump intructions. if dont have =0
pred_result_to_dec	O	4	DECODE stage	prediction result for 4 instructions, if not branch =0

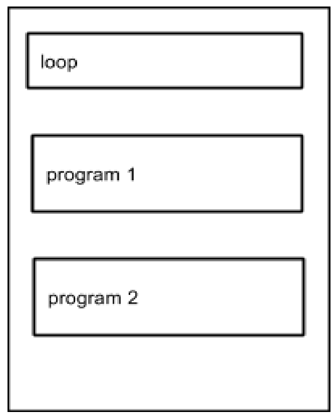
This module assembles important data needed by the subsequent stages in a certain format. These data include all the PC values, instruction values, PC values for recovering from branch/jump and branch prediction results which will be used by reorder buffer to check if the prediction is correct.

4.1.2. Normal Mode

As can be seen from the block diagram, PC value is decided by the output of PC multiplexer which is by default 16’b0. Then this PC value is sent to pc register and instruction memory module simultaneously. Because the instruction memory has registered inputs, there will be one clock cycle delay. By making the

PC register in parallel, the delay will be eliminated. Another thing worth noticing is that, at the start of the operation, reset button will be pushed and instructions from the first line of memory will be fetched twice and sent to the next stage by mistake. To deal with this problem we designed a start signal will will not be cleared until reset is triggered. When start is high, instructions will not be read from instruction memory and passed to the following stages.

To make the behavior of processor more controllable, we introduced a port to set the PC to specific value externally. In the instruction memory, the first couple of lines will be an infinite loop. If user does



not specify the start PC of programs stored in the instruction memory, the processor will run in the loop endlessly. Once an external PC value is set, the program counter will jump the the given address and begin fetching the instructions of the benchmarks.

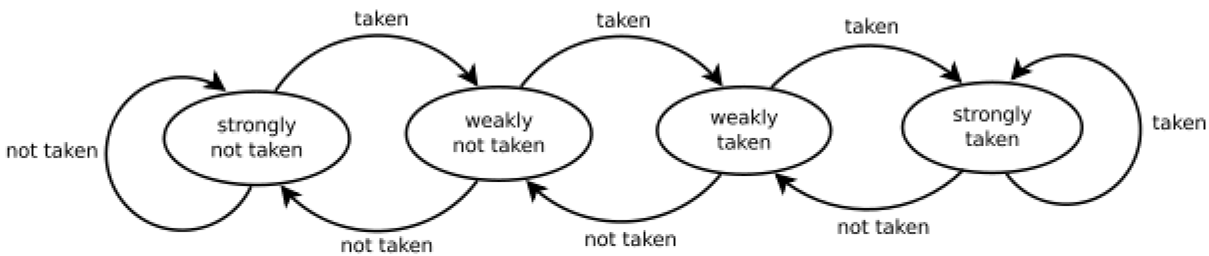
After four instructions are read from instruction memory, they are send to the branch handler which is used to detect number of branches and keep count of them. When a branch is detected, the branch counter will be increment by one and the predictor will be enabled to give a prediction. If it is taken, the branch subsequent instructions fetched in this cycle will be replaced by NOP instructions. As multiple branch prediction will exert much burden on predictor when misprediction occurs, we limit the number of uncommitted branches in the processor to be less or equal than two. When the counter is already two and a third branch is detected, the PC will hold at the third branch's PC and NOPs will be send until the branch counter is decremented. There are two cases when the counter will be deducted. One is when a branch is committed from reorder buffer and a decr_count_brnch signal is sent to fetch stage. And the other circumstance is on misprediction which will be discussed in the later section.

After passing the branch handler, instructions are fed to a jump handler. When there are multiple jumps in the four instructions, only the first one will be processed for calculation of jump address and the ones following the jump will all be replaced by NOP instructions.If it is immediate jump, the pc will jump to the address the next cycle and nothing is send to the next stage for this instruction in this cycle. If the instruction is register based jump, the PC will hold at its pc the next cycle and output NOP instructions. This state will not change until a jump_base_rdy signal is received. The jump base will be stored in a register right away and the processor will wait one cycle for computing the jump address. In this way the critical path will be reduced. After one cycle the PC will jump to target address and restart fetch next instructions. On exception is when misprediction occurs, the waiting for jump ready state will be stopped and jump handler will go back to initial state.

At the output, the instructions to next stage will be assembled into four packages. 64 bit of PCs, 64 bits of four instructions, 64 bits of recovery PCs and 4 bits of prediction results. The recovery PC is the branch address if it is not taken and it is the next PC when branch taken.

4.1.3. Misprediction

On receiving the misprediction signal from reorder buffer will send three signals to fetch stage: one indicating that there is a midprediction; one show which of the branch is mispredicted; and the last send back the original wrong prediction result. Based on these information, the saturation counter predictor will decide to decrease one or two steps. Below is the graph for the counter.



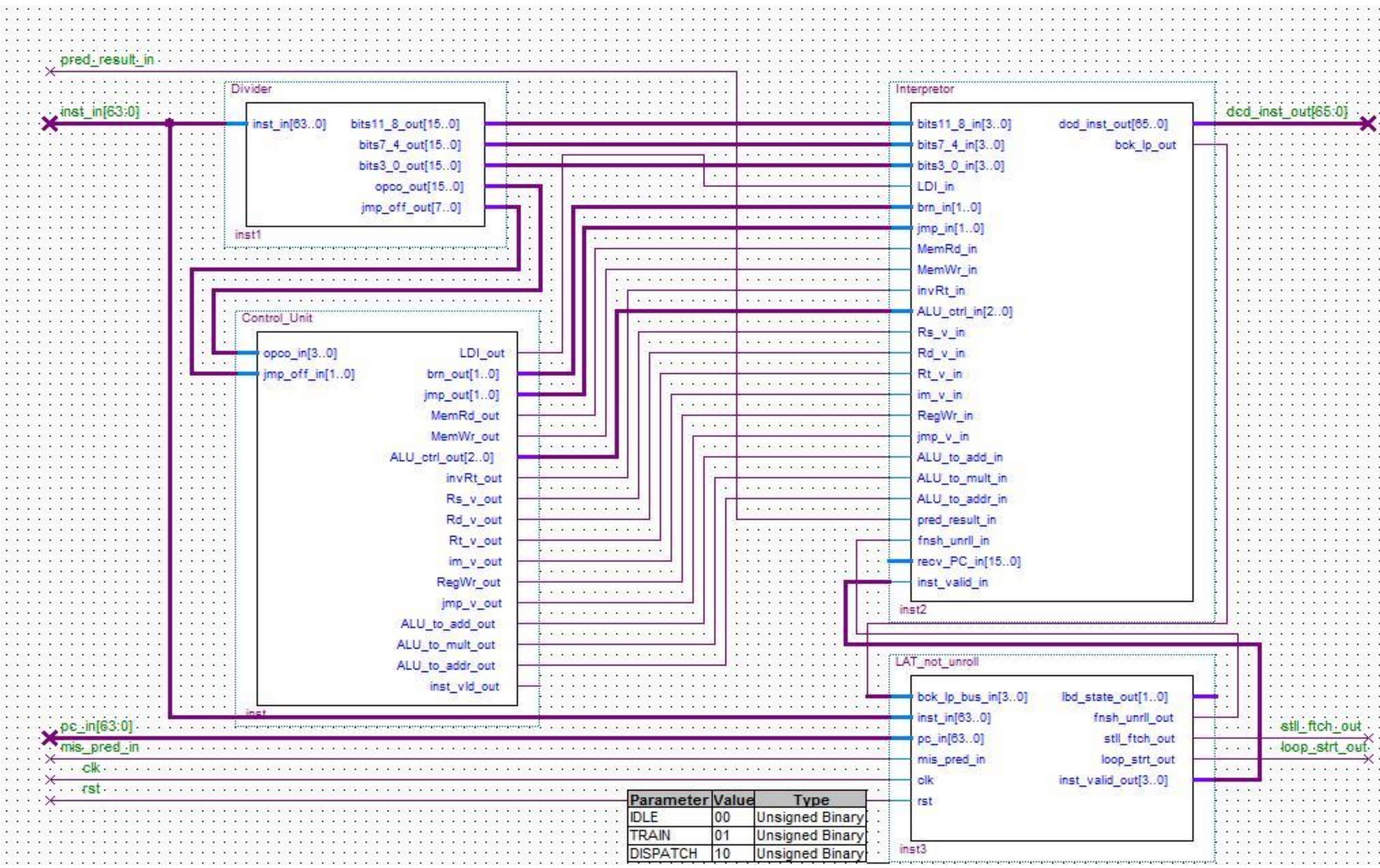
If the misprediction occurs when jump handler is stalling the PC to wait for jump base ready signal, the wait signal will be cleared and jump address register will be reset.

4.1.4. Loop Mode

During loop mode, when LAT first detects a loop, it will send a loo_start signal to fetch. On receiving the signal, the branch predictor will only output predict taken and the branch counter will be disabled so that the instructions can be continuously fetched until a stall fetch is received.

4.2. Instruction Decode (ID)

The main purpose of this stage is to accept instructions from **Fetch** stage and correctly decode them. Here decode means to determine what registers each instruction will use, to decide what control signals should be generated for each instruction and to reproduce a new line for each instruction such that contains all necessary information for the following stages. Also, this stage will achieve the loop detection functionality that **detects loops**, **buffers loops, unrolls loops and enters loop modes**.



4.2.1. Interface Description

4.2.1.1. Interface Description

This module extracts the four instructions from inst_in_frm_IF bus and processes them respectively: divide instruction bits based on opcode, Rs, Rd, Rt and jump_offset.

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
Inst_in	I	64	IF	[63:0] 4 instructions in a bus from fetch
opco_out	O	16	Control Unit	opcodes of each instructions.
jmp_off_out	O	8	Control Unit	the 1st to the 0th bits of each instruction
bits11_8_out	O	16	Interpreter	the 11th to the 8th bits of each instruction
bits7_4_out	O	16	Interpreter	the 7th to the 4th bits of each instruction

bits3_0_out	O	16	Interpreter	the 3rd to the 0th bits of each instruction

4.2.1.2. Control Unit

This module accepts opcode and jump offset of each instruction, and outputs necessary control signal for both the **Interpreter** and **following stages.**

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
opco_in	I	16	Divider	opcodes of each instructions. (comes from Divider.opcodes)
LDI_out	O	4	Interpreter	whether each instruction is LDI or not

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
jmp_off_in	I	18	Divider	offset that determines jump's type. (comes from Divider.1_0bits)
brn_out	O	8	Interpreter	branch's type of each instruction
jmp_out	O	12	Interpreter	jump type of each instruction
MemRd_out	O	4	Interpreter	whether each instruction will do memory read or not
MemWr_out	O	4	Interpreter	whether each instruction will do memory write or not
ALU_ctrl_out	O	16	Interpreter	ALU control signal that controls ALU behaviors
invRt_out	O	4	Interpreter	whether each instruction will need to invert Rt or not
Rs_v_out	O	4	Interpreter	Rs valid bits
Rd_v_out	O	4	Interpreter	Rd valid bits
Rt_v_out	O	4	Interpreter	Rt valid bits
im_v_out	O	4	Interpreter	immediate valid bits
RegWr_out	O	4	Interpreter	whether each instruction is writing to registers or not

4.2.1.3. Interpreter

This module accepts the divided bits from **Divider** and control signals correspond to each instruction from **Control Unit**, then it interprets each instruction, deciding what Rs, Rd, Rt or imme it may use, as well as what control signal it will utilize, and package each of them into one line.

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
bits11_8_in	I	16	Divider	[15:0] the 11th to the 8th bits of each instruction. (from Divider.11_8bits)
bits7_4_in	I	16	Divider	[15:0] the 7th to the 4th bits of each instruction. (from Divider.7_4bits)

bits3_0_in	I	16	Divider	[15:0] the 3rd to the 0th bits of each instruction. (from Divider.3_0 bits)
LDI_in	I	4	Control Unit	whether each instruction is LDI or not
brn_in	I	8	Control Unit	branch's type of each instruction
jmp_in	I	12	Control Unit	jump type of each instruction
MemRd_in	I	4	Control Unit	whether each instruction will do memory read or not
MemWr_in	I	4	Control Unit	whether each instruction will do memory write or not
invRt_in	I	4	Control Unit	whether each instruction will need to invert Rt or not
ALU_ctrl_in	I	16	Control Unit	ALU control signal that controls ALU behaviors
Rs_v_in	I	4	Control Unit	Rs valid bits
Rd_v_in	I	4	Control Unit	Rd valid bits
Rt_v_in	I	4	Control Unit	Rt valid bits
im_v_in	I	4	Control Unit	immediate valid bits
RegWr_in	I	4	Control Unit	whether each instruction is writing to registers or not
pred_result_in	I	4	IF	predict result of each instruction
dcd_inst1_out	O	64	AL	Check Decode_instruction_package_format sheet
dcd_inst2_out	O	64	AL	check Decode_instruction_package_format sheet
dcd_inst3_out	O	64	AL	check Decode_instruction_package_format sheet
dcd_inst4_out	O	64	AL	check Decode_instruction_package_format sheet
bck_lp_out	O		LAT	[3:0] indicates whether each signal is a backwards branch or not

The output **decoded_instruction**'s format is shown as follow

package	description	width	range
inst_valid	whether this instruction is valid or not	1	[63]
Rs_valid_bit	valid bit for Rs	1	[62]
Rs	Rs number	4	[61:58]
Rd_valid_bit	valid bit for Rd	1	[57]
Rd	Rd number	4	[56:53]
Rt_valid_bit	valid bit for Rt	1	[52]
Rt	Rt number	4	[51:48]
imm_valid_bit	valid bit for immediate	1	[47]
imme	immediate value	16	[46:31]
LDI	control signal LDI	1	[30]
brn	control signal Branch	2	[29:28]

	00: not branch 01: BEQZ 10: BLTZ 11: BGTZ		
jmp_valid_bit	valid bit for control signal Jump	1	[27]
jump	control signal Jump 00: J 01: JR 10: JAL 11: JALR	2	[26:25]
MemRd	control signal Memory Read	1	[24]
MemWr	control signal Memory Write	1	[23]
ALU_ctrl_valid_bit	valid bit for ALU control	1	[22]
ALU_ctrl	control signal ALU control	3	[21:19]
invtRt	control signal invtRt	1	[18]
RegWr	control signal RegWr	1	[17]
pred_result	predict result	1	[16]
recv_pc	recovery PC of this instruction	16	[15:0]

4.2.1.4. LAT

This module acts as the **Loop Address Table**, which buffers any loop information, and decides whether we should enter loop mode via the **FSM** in it.

Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I	1		
bck_lp_in	I	1	Interpretor	indicates whether each signal is a backwards branch or not. (from Interpretor.backwards_branch)
inst_in	I	64	IF	4 PCs in a bus from fetch
pc_in	I	64	IF	4 PCs in a bus from fetch
mis_pred_in	I	1	ROB	mispredict signal from commit stage
lbd_state_out	O	2	AL	loop_dector state. 00: IDLE 01: TRAIN 10: DISPATCH
fnsh_unrll_out	O	1	AL	signal that indicates we've finished unrollinng
stll_ftch_out	O	1	IF	signal that stalls fetch state

4.2.2. Normal mode

Accepts four instructions, along with four PCs and four recovery PCs from **Fetch** stage. Each of them will be packed into one bus (i.e. instruction bus, PC bus and recovery PC bus), and be processed by three modules: **Divider, Control Unit and Interpreter**.

First, the **instruction bus**, which is 64 bits wide, as it contains four instructions at a time, will enter the **Divider** module. **Divider** will simply recognize opcode, jump offset, the eleventh to the eighth bits

(bits11_8), the seventh to the fourth bits **(bits7_4)** and the third to the zeroth bits **(bits3_0)** for each instruction, respectively. Each of these recognized fields will be outputted as a bus. For example, the **opcode bus** will be a 16 bits wide line, in the format of **{1st instruction's opcode, 2nd instruction's opcode, 3rd instruction's opcode, 4th instruction's opcode}.** Opcode and jump offset will be passed to **Control Unit** in order to generate corresponding control signal, and the others will be passed to **Interpreter** to generate the decoded instruction.

With opcode and jump offset from **Divider**, our **Control Unit** is ready to generate required control signals for both our **interpreter** and following stages. Other than those common control signals (i.e. MemRead, MemWrite, ALU_ctrl, etc), we added six more signals: Rs_valid, Rd_valid, Rt_valid, immediate_valid, jmp_valid and ALU_ctrl_valid outputting to our **Interpreter**. As their names indicate, they represent whether a instruction contains Rs, Rd, Rt or immediate field or not, performs jump or not and utilizes ALU or not, respectively. We will use them to form our decoded instructions inside **Interpreter**. Also, since we receive four instructions at a time, each of these signals will be outputted as a bus, representing a specific control signal for four instructions.

Now **Interpreter's** inputs are all control signal buses from **Control Unit,** bits11_8 bus, bits7_4 bus, bits3_0 bus**,** and recovery PC bus, and it can utilize them to form decoded instructions. A decoded instruction is 64 bits wide instead of 16 bits wide.

Interpreter Basically uses all its input signals to form four decoded instructions. Note that we can observe the four valid bits we mentioned previously (Rs_valid, Rd_valid, Rt_valid,immediate_valid, jmp_valid and ALU_ctrl_valid) inside the decoded instruction. Each of them will mark whether the field right next to it is valid or not. By packing all this information together, **Interpreter** outputs **four** decoded instruction buses, and each of them should contain all necessary information that following stages may use.

4.2.3. Loop mode

The **LAT** module is designed for handling loop mode. There are three states in **LAT**:

IDLE: Monitor each coming PC, if it matches any start address in the Loop Address Table, jump to **DISPATCH** state;
or if **Interpreter** detects a **backwards branch,** jump to **TRAIN** state.

TRAIN: Entered when a **backwards branch** is detected. Once entered, record loop information such as start addr,
fall through addr, # of instructions, profitability and max # of unroll. When it met the same **backwards branch**
again, jump back to **IDLE**. If any known loop is detected during **TRAIN**, jump to **DISPATCH** state.

DISPATCH: Signal to unroll the entire loop as many times as possible in **Issue Queue**, then stalls **Fetch** and wait until
a **mispredict** signal comes from **COMMIT**.

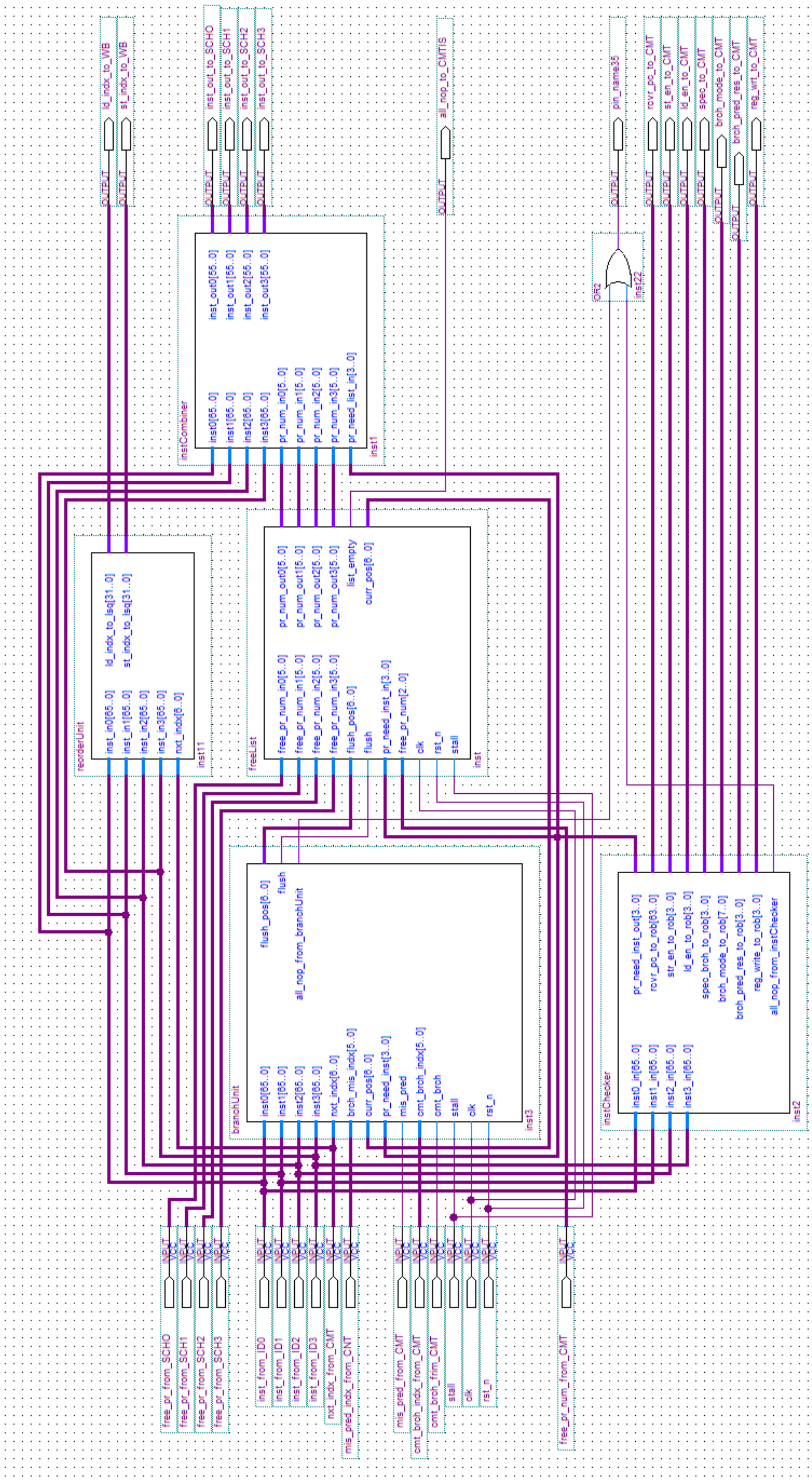
Every time we detect a backwards branch inside **Interpreter**, by looking at the immediate field of branch instruction, **LAT** enters **TRAIN** state in order to buffer loop information. Once all required information is recorded, **LAT** jumps back to **IDLE** state. Note that if an already known loop is detected during **TRAIN**,

LAT will jump to **DISPATCH** state directly. This pattern is designed to handle nested loop. Then, when we detect an already known loop in **IDLE** state, by comparing each **PC address** to those start addresses in our **LAT**, **LAT** jumps to **DISPATCH** state. During **DISPATCH,** we first unroll the entire loop as many times as possible (using the max# of unroll field in **LAT**) and then fill the rest of **Issue Queue** with “Noop” (i.e. 64bits 0s). This is done by outputting a finish_unroll signal from **LAT** to **Interpreter**; once this line is signal by **LAT**, all four decoded instructions will be mux to 64bits 0s. Moreover, **LAT** will keep a counter to track whether it has filled up the **Issue Queue** or not. Once that counter reaches 64, the entire **DECODE** stage is stalled untill **COMMIT** stage signals a mispredict signal, which will move **LAT** back to **IDLE** state.

4.2.4. Misprediction

In **DECODE** stage, if a misprediction occurs and its **LAT** is in **DISPATCH STATE**, it will signal the **FETCH** stage to stop stalling.

4.3. Allocation (AL)



4.3.1. Stage interface:

The allocation stage mainly consists of five blocks: InstChecker, InstCombiner, BranchUnit, ReorderUnit and Freelist.

Overview Interface:

Signal Name	I/O	Width	Source/Target (blocks)	Description
free_pr_num_from_SCH[3:0]	I	6	Schedule Stage	free physical register number need to be freed from schedule stage
inst_from_ID[3:0]	I	66	Decode Stage	Combined packet with parsed register number, pc and recovery pc. Also including the corresponding control signal
nxt_indx_from_CMT	I	7	Commit Stage	next available index from commit stage
clk	I	1	global	clock signal
stall	I	1	global	stall signal
rst	I	1	global	global asynchronize reset
lbd_state_out_from_ID	I	2	Decode Stage	loop_dector state. 00: IDLE 01: TRAIN 10: DISPATCH
mis_pred_from_CMT	I	1	Commit Stage	Indicated there is a mispredict branch
mis_pred_indx_from_CMT	I	6	Commit Stage	Indicate the instruction index of the mispredict branch
cmt_brch_from_CMT	I	1	Commit Stage	Indicate there is a commit branch
cmt_brch_indx_from_CMT	I	6	Commit Stage	Indicate the instruction index of the commit branch
fnsh_unrll_out_from_ID	I	1	Decode Stage	signal that indicates we've finished unrollinng
inst_out_to_SCH[3:0]	O	56	Schedule Stage	add the preg number to the combined packet and send it to the schedule
no_empt_preg_to_IF	O	1	Fetch Stage	signal the fetch stage that there are no more free preg in freelist
rcvr_pc_to_CMT	O	64	Commit Stage	recovery PC address to commit stage
reg_wrt_to_CMT	O	4	Commit Stage	regwrite control to commit
str_en_to_CMT	O	4	Commit Stage	store control to commit
spec_to_CMT	O	4	Commit Stage	speculative control to commit
brch_mode_to_CMT	O	8	Commit Stage	branch mode to commit
brch_pred_res_to_CMT	O	4	Commit Stage	branch prediction result to commit
load_indx_to_WB	O	32	WriteBack Stage	Load index and valid bit to the write back stage
st_indx_to_WB	O	32	WriteBack Stage	Sotre index and valid bit to the write back stage
lbd_state_out_to_SCH	O	2	Schedule Stage	loop_dector state. 00: IDLE

				01: TRAIN 10: DISPATCH
fnsh_unrll_out_to_SCH	O	1	Schedule Stage	signal that indicates we've finished unrollinng

4.3.2. Block Detail

4.3.2.1. InstChecker:

The purpose of instChecker unit is to load the decoded instruction packet from decode stage and check whether the instruction need to be assign a new physical register, i.e. has a reg write. InstChekcer also distribute the control signal to reorder buffer.

InstChecker Interface:

Signal Name	I/O	Width	Source/Target (blocks)	Description
inst_in[3:0]	I	66	Decode Stage	Combined packet with parsed register number, pc and recovery pc. Also including the corresponding control signal
nxt_indx	I	7	Commit Stage	next available index from commit stage
pr_need_inst_out	O	4	FreeList	Let the free list know which inst need a preg
rcvr_pc_to_rob	O	64	Reorder Buffer	recovery PC address to reorder buffer
reg_wrt_to_rob	O	4	Reorder Buffer	regwrite control to reorder buffer
str_en_to_rob	O	4	Reorder Buffer	store control to reorder buffer
spec_to_rob	O	4	Reorder Buffer	speculative control to reorder buffer
brch_mode_to_rob	O	8	Reorder Buffer	branch mode to reorder buffer
brch_pred_res_to_rob	O	4	Reorder Buffer	branch prediction result to reorder buffer
ld_indx_to_lsq	O	32	Load/Store Queue	Load index send to load store queue
st_indx_to_lsq	O	32	Load/Store Queue	Store index send to load store queue

4.3.2.2. Reorder Unit:

The purpose of the reorder unit is to distribute the load store control signal and the index of load store instruction to load store queue. Since the load store queue cannot load nop into it, the reorder unit will also do a compact work so that all load store control signal will be placed together. The basic structure of reorder unit is lots of priority mux and decide the place that an load/store instruction it should be placed.

Reorder Unit interface:

Signal Name	I/O	Width	Source/Target (blocks)	Description
inst_in[3:0]	I	66	Decode Stage	Combined packet with parsed register number, pc and

				recovery pc. Also including the corresponding control signal
nxt_indx	I	7	Commit Stage	next available index from commit stage
ld_indx_to_lsq	O	32	Load/Store Queue	Load index send to load store queue
st_indx_to_lsq	O	32	Load/Store Queue	Store index send to load store queue

4.3.2.3. Freelist:

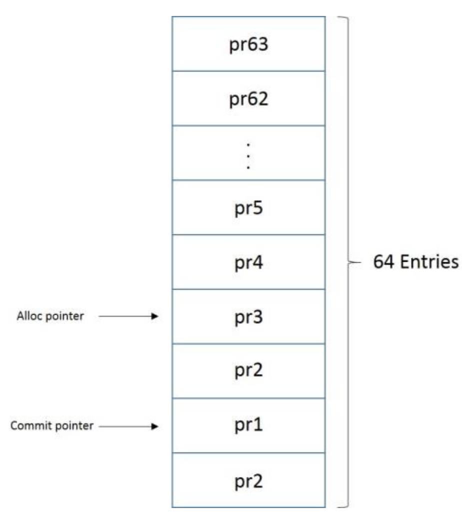
Freelist is the core part of allocation stage. It stored all the free physical register number and keep updating. Freelist will also give out a list empty signal when there is less than three free physical register in the list.

Frelist interface:

Signal Name	I/O	Width	Source/Target (blocks)	Description
free_pr_num_in[3:0]	I	6	Commit Stage	The physical register numbers that need to be freed
free_pr_num	I	3	Commit Stage	The number of physical register will be freed in a cycle
pr_need_inst_in	I	4	Instruction Checker	tell the freelist which instruction need a preg
flush	I	1	Branch Unit	Tell the freelist there is a mispredict branch happen and freelist need to do a flush operation
flush_pos	I	7	Branch Unit	Tell the freelist to move allocation pointer back to the flush position when there is a flush happen
stall	I	1	global	stall signal
clk	I	1	global	clock signal
rst	I	1	global	global asyn reset
pr_num_out[3:0]	O	6	Instruction Combinner	physical register assign to each instruction
list_empty	O	1	Fetch Stage	notify that list is empty
curr_pos	O	7	Branch Unit	Tell the branch unit current allocation pointer position when there is a branch come in

4.3.2.4. Detail of Freelist

Normal operation:



There are 64 entries and two pointers in the freelist. When the unit is reset, number 0 to 63 were written into freelist. Allocation pointer is placed at position index at 16 and commit pointer is placed at position index at 0.

When we need a new freelist, freelist will give out the number that allocation pointer currently pointed at. Then it will move allocation pointer up. During the assignment, freelist will not change the value store in the entries.

When an instruction is committed, reorder buffer will give back the free physical register number. Then freelist will use this number to overwrite the value store in the entry currently pointed by commit point.

When there are less than 3 free physical registers in the list, freelist will output a signal to indicate there are no more physical registers available. In this case, fetch should be stalled.

Freelist do the assignment operation based on the physical register need sequence generated by instChecker. The sequence is a 4 bits bus and 1 represent that the instruction need a physical register. When a freelist is stalled, it will not assign any physical register out to any instruction. However, during the stall, commit operation will keep working.

Misprediction Operation:

When there is a misprediction happen, freelist need to restore its allocation pointer position back to the flush position which is the last position before a speculative branch instruction.

Freelist keeps telling the branchUnit its allocation position at every cycle. When there is a branch come in, branch Unit will record current allocation pointer's position. When a mispredict branch happens, branchUnit will find the record allocation pointer's position. Then freelist will restore its allocation pointer back to that position. This design is safe because reorder buffer will not commit any instruction after a speculative branch which means the commit pointer will not go ahead to allocation pointer.

Loop Mode operation:

During the original loop mode design, freelist will assign two physical registers to an instruction when the processor is in the loop mode. However, because the loop mode was change to an not-unroll design, freelist will work as the same as the normal operation in loop mode.

4.3.2.5. BranchUnit:

BranchUnit is another important part for allocation stage. The purpose of branchUnit is to record the branch instruction so that freelist can be recovered when there is a mispredict branch happened. Since the processor needs to handle dependency of branches, so branchUnit is designed as a FIFO structure. There are two entry for branchUnit since the max number of branch instructions handled by processor in one cycle is two.

BranchUnit Interface:

Signal Name	I/O	Width	Source/Target (blocks)	Description
curr_pos	I	7	freelist	The branch unit need to record the current position of allocation pointer in freelist
inst_in[3:0]	I	66	Decode Stage	Combined packet with parsed register number, pc and recovery pc. Also including the corresponding control

				signal
mis_pred	I	1	Commit Stage	Indicated there is a mispredict branch
mis_pred_indx	I	6	Commit Stage	Indicate the instruction index of the mispredict branch
cmt_brch	I	1	Commit Stage	Indicate there is a commit branch
cmt_brch_indx	I	6	Commit Stage	Indicate the instruction index of the commit branch
nxt_indx	I	7	Commit Stage	next available index from commit stage
stall	I	1	global	stall signal
clk	I	1	global	clock signal
rst	I	1	global	global asyn reset
flush	O	1	Freelist	Tell the freelist there is a mispredict branch happen and freelist need to do a flush operation
flush_pos	O	7	Freelist	Tell the freelist to move allocation pointer back to the flush position when there is a flush happen

4.3.2.6. Detail of BranchUnit:

Branch Unit is a two-entries FIFO. It has a head pointer and a tail pointer. Each entry consists of one bit valid bit, six bits index and seven bits freelist position. When a branch is coming in, it will be added to the entry pointed by tail point. When a branch pointed by head pointer is committed, head pointer will move one entry. When misprediction happed to the branch pointed by head pointer, it means the first branch is wrong and the next branch is useless, so the whole FIFO will be cleared and both pointers are reset back to initial position. In the other case, branchUnit just clean the branch that is committed or mispredicted.

When a misprediction happens, branchUnit will set the flush signal high, give out the flush position and clear the record in branchUnit.

4.3.2.7. InstCombiner:

The purpose of Instruction Combiner is to add the assigned physical register into instruction packet and remove the unnecessary recovery PC out. There is almost all combinational logic in this block.

Signal Name	I/O	Width	Source/Target (blocks)	Description
inst[3:0]	I	64	Decode Stage	instruction packet come from decode
pr_num_in[3:0]	I	6	FreeList	physical register number assign to each instruction
pr_need_inst_in	I	4	Instruction Checker	tell the combinner which instruction need a physical register
inst_out[3:0]	O	54	Schedule Stage	instruction packet go out after combine the preg number in it

4.4. Issue (IS)

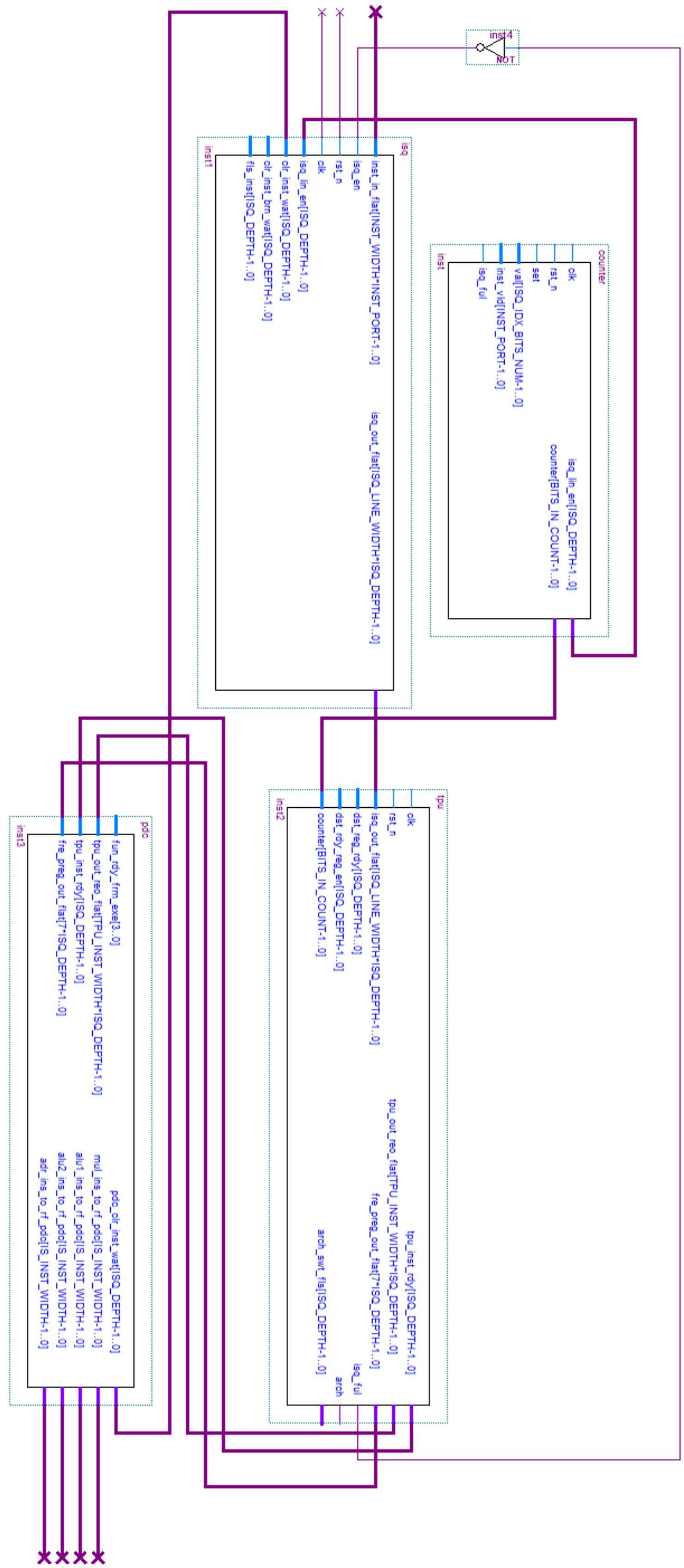
4.4.1. Stage Overview

Starting from Issue stage, the looper CPU starts to execute out of order. Issue stage takes in instructions from front end, buffer at maximum sixty-four instructions, performs register renaming through tag-propagation, determines dependencies and dispatches instructions out of program order to CPU backend. Connecting with the front end, it receives at maximum four instructions at one time from allocation stage along with allocated physical register number if an instruction writes to register file. Bridging the back end, it sends out at maximum four instructions to register file stage after checking the availabilities of each function unit in execution stage.

As shown in the figure 1, Issue stage contains 4 blocks, an issue queue, a counter, a tag propagation unit and a customized priority queue to decide which instruction to send. The overall interface of issue stage is shown in the following table.

4.4.2. Overall interface:

Signal Name	IO	Width	Source/Target	Description
clk	I	1	global	global clk
rst_n	I	1	global	global reset
inst_frm_al	I	64	allocation stage	instruction from front-end
lop_sta	I	1	allocation stage	start signal of loop mode
fls_frm_rob	I	7	rob	flush branch signal from rob
cmt_frm_rob	I	7	rob	commit branch signal from rob
fun_rdy_frm_exe	I	4	execution stage	function unit ready signals from execution
prg_rdy_frm_exe	I	28	execution stage	physical register ready signals from execution
ful_to_al	O	1	allocation stage	issue queue full signal to allocation to stop feeding instructions in
mul_ins_to_rf	O	66	register file	multiply instructions to backend
alu1_ins_to_rf	O	66	register file	instruction utilize alu1 functional unit to backend
alu2_ins_to_rf	O	66	register file	instruction utilize alu2 functional unit to backend
adr_ins_to_rf	O	66	register file	instruction utilize address adder functional unit to backend


isq
inst1
inst_in_flat[INST_WIDTH*INST_PORT-1..0]
isq_en
rst_n
clk
isq_lin_en[ISQ_DEPTH-1..0]
clr_inst_wat[ISQ_DEPTH-1..0]
clr_brn_wat[ISQ_DEPTH-1..0]
fls_inst[ISQ_DEPTH-1..0]
isq_out_flat[ISQ_LINE_WIDTH*ISQ_DEPTH-1..0]
inst4
NOT
counter
inst
clk
rst_n
set
val[ISQ_IDX_BITS_NUM-1..0]
inst_vld[INST_PORT-1..0]
isq_ful
isq_lin_en[ISQ_DEPTH-1..0]
counter[BITS_IN_COUNT-1..0]
tpu
inst2
clk
rst_n
isq_out_flat[ISQ_LINE_WIDTH*ISQ_DEPTH-1..0]
dst_reg_rdy[ISQ_DEPTH-1..0]
dst_rdy_reg_en[ISQ_DEPTH-1..0]
counter[BITS_IN_COUNT-1..0]
tpu_inst_rdy[ISQ_DEPTH-1..0]
tpu_out_reo_flat[TPU_INST_WIDTH*ISQ_DEPTH-1..0]
fre_preg_out_flat[7*ISQ_DEPTH-1..0]
isq_ful
arch
arch_swt_fls[ISQ_DEPTH-1..0]
pdc
inst3
fun_rdy_frm_exe[3..0]
tpu_out_reo_flat[TPU_INST_WIDTH*ISQ_DEPTH-1..0]
tpu_inst_rdy[ISQ_DEPTH-1..0]
fre_preg_out_flat[7*ISQ_DEPTH-1..0]
pdc_clr_inst_wat[ISQ_DEPTH-1..0]
mul_ins_to_rf_pdc[IS_INST_WIDTH-1..0]
alu1_ins_to_rf_pdc[IS_INST_WIDTH-1..0]
alu2_ins_to_rf_pdc[IS_INST_WIDTH-1..0]
adr_ins_to_rf_pdc[IS_INST_WIDTH-1..0]

4.4.3. Operation Overview:

4.4.3.1. Normal Mode:

The issue queue loads instructions coming from allocation stage to next available lines in the queue. The counter determines next available slots in issue queue. Counter's value increments by four each cycle unless the four instructions coming in from the allocation stage are all NOP or the issue queue is full. When all the instructions from allocation stage are NOP, counter's value stay the same and those NOPs are ignored.

Once an instruction is loaded into issue queue, the tag propagation unit starts register renaming and dependency check. For each slot in issue queue, (64 in our implementation), there is a dedicated tag propagation line unit to perform register renaming and dependency check. As discussed in Mitchell Hayenga's thesis, each tag propagation unit get the logical to physical mapping from the line unit logically above it and changes the mapping if an instruction is writing to register file. In this way, all tag-propagation unit logically below can see the updated mappings. Each tag propagation line unit also checks if all operands are ready. It outputs ready signals for each instruction if dependencies are all resolved. Overall, in our implementation, tag propagation has 64 tag propagation line units and 2 segment headers serving as storage elements to record logical to physical register mappings. At one time instance, only one of these two segment headers serve as architecture state while the other is bypassed. When half of instructions in the issue queue between these two segment headers are all dispatched, an architecture switch happens – the other segment header reads in the most updated mappings and serves as architecture state while the original header is bypassed.

The last block in issue stage, the priority queue, determines which instructions to dispatch. It first differentiate instructions by their functions. It has four dedicated output ports for instructions using different function unit. These four dedicated output ports, corresponding to function units in the execution stage are multiply, alu1, alu2 and address adder. All multiply instructions are sent through dedicated "multiply" port. Similarly, some alu instructions such as add, subtract and branch and jump instructions will be sent from alu1 port. Other alu instructions go through alu2 port. Alu1 and alu2 has a distribution of 1:2 for alu instructions. In other words, on average, 1/3 of alu instructions flow through alu1 port while 2/3 of alu instruction flow through alu2 port. Such design choice is made since alu1 port also sends out branch and jump instructions. Load and store instructions, which utilizes address adder go through address adder port.

4.4.3.2. Branch-Misprediction:

If there a branch misprediction happens, every instructions below such branch in program need to be flushed by issue queue. There are two main goals that need to be achieved by flushing:
1) Issue stage should not send any existing instructions in the issue queue below such branch instruction any more.
2) Register renaming needs to recover to the state before the branch instruction.
With current design of tag propagation unit, achieving these two design goals are not hard. When issue stage gets a branch misprediction signal from reorder buffer. The issue queue invalidates instructions that are below the branch instruction. Invalid instructions will not affect logical to physical mapping by design in the tag propagation unit. Hence, mapping is restored. Counter, at the same time, sets its value to the original branch instruction so that new instructions can utilize

free issue queue space after flushing. One final point to notice, to make sure logical to physical mapping can be restored, architecture state cannot happen until branches are resolved, whether mispredicted or committed. Such operation is implemented by associate each issue queue slot with a branch wait flag. Such wait flag is not cleared until the branch is resolved.

4.4.3.3. Loop-Mode:

Lop-sta signal indicates the start of a loop mode. At the beginning of the loop mode, looper CPU frontend will unroll instructions in the loop to 64 instructions. Issue stage then only dispatch existing instructions in the issue queue until a branch mispredication happens. Such loop-mode is implemented by reset the “wait” bit associated with each issue queue slot when an architecture state happens. By resetting the “wait” bit, issue stage will then treat these instructions as newly fetched from front-end and dispatch them for another time. Current implementation of issue stage has only partial support for loop mode. We have not fully tested issue stage yet in loop mode.

4.4.4. Block Description

4.4.4.1. issue queue

As mentioned above, issue queue buffers instructions from allocation stage. Since there are no pipeline registers between allocation and issue stage, issue queue serves as a storage element among these two stages. At normal operation, allocation stages send four instructions at once to be loaded into the issue queue. Issue queue comprises of 64 slots to store the instructions fetched from the front-end. Each issue queue slot has following bit structure:

Valid | branch wait | wait | instructions

Valid and wait bit are set when a new instruction is loaded, meaning that such instruction is valid and waiting to be dispatched. Branch wait is also set if the instruction loaded is a branch instruction. Valid bit will be cleared when architecture state happens in tag propagation unit. Wait bit will be cleared when current instruction is dispatched to the backend. Branch wait will be cleared when the branch is resolved (flushed or committed). Tag propagation unit refer to all three bits of valid, branch wait and wait in each issue queue slot to determine whether an architecture switch can happen.

When branch misprediction happens, valid bit of instructions below such branch instruction is set to 0 so that new instructions can be loaded. In loop mode, wait bit of instructions which are just finished will be set to high again when tag-propagation unit’s architecture state changes so that these instructions will be executed again without any fetching from the front end.

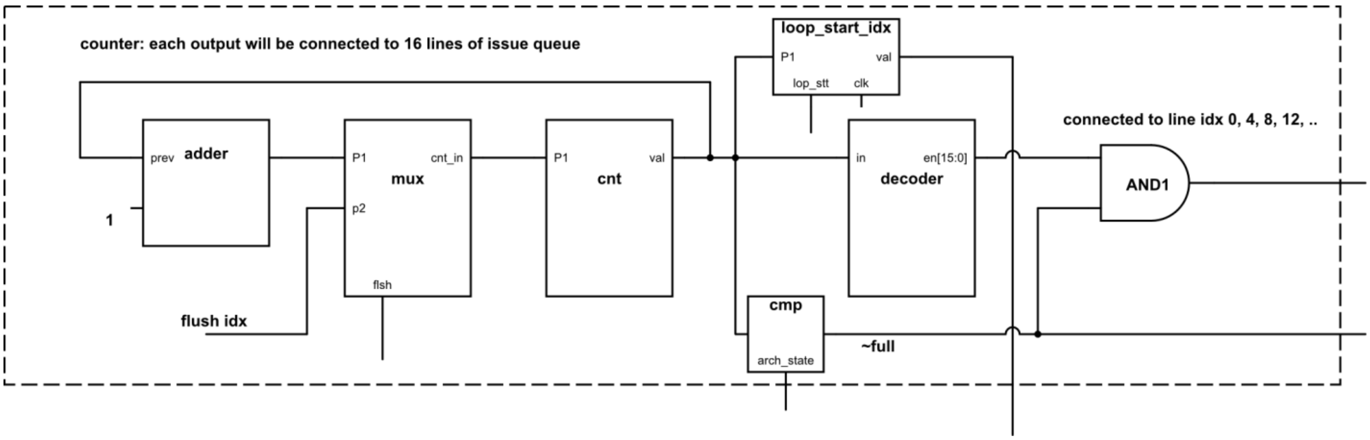
At loop mode, branch wait and wait bit will be reset to 1 when architecture switch happens so that such instruction can be dispatched again by issue stage.

Interface:

Signal Name	IO	Width	Source/Target	Description
clk	I	1	global	global clk
rst_n	I	1	global	global reset
isq_en	I	1	counter	issue queue enable
inst_in_flat	I	224	allocation stage	instructions from frontend
isq_lin_en	I	64	counter	enable signal for each issue queue slot
clr_inst_wat	I	64	pdc	clear instruction wait signal from priority queue
clr_inst_brn_wat	I	64	rob	clear instruction branch wait when such branch is committed
fls_inst	I	64	rob	flush issue queue slot signal when branch misprediction happens
isq_out_flat	O	4096	tpu	output of all 64 instructions to tpu

4.4.4.2. Counter

The counter is used for helping issue queue identify the next available slot so that it can load new instructions correspondingly. It will increment itself by 4 every cycle. When a branch misprediction happens, it sets the value to the multiples of 4 which is closer to the position of branch instruction that causes the misprediction so that new instructions can be loaded to override invalid instructions.
The structure of the counter is following:



In loop mode, the counter will also record when the loop starts so that we will not re-execute instructions immediately before the loop before front-end finish filling issue queue fully with instructions in the loop.

Interface:

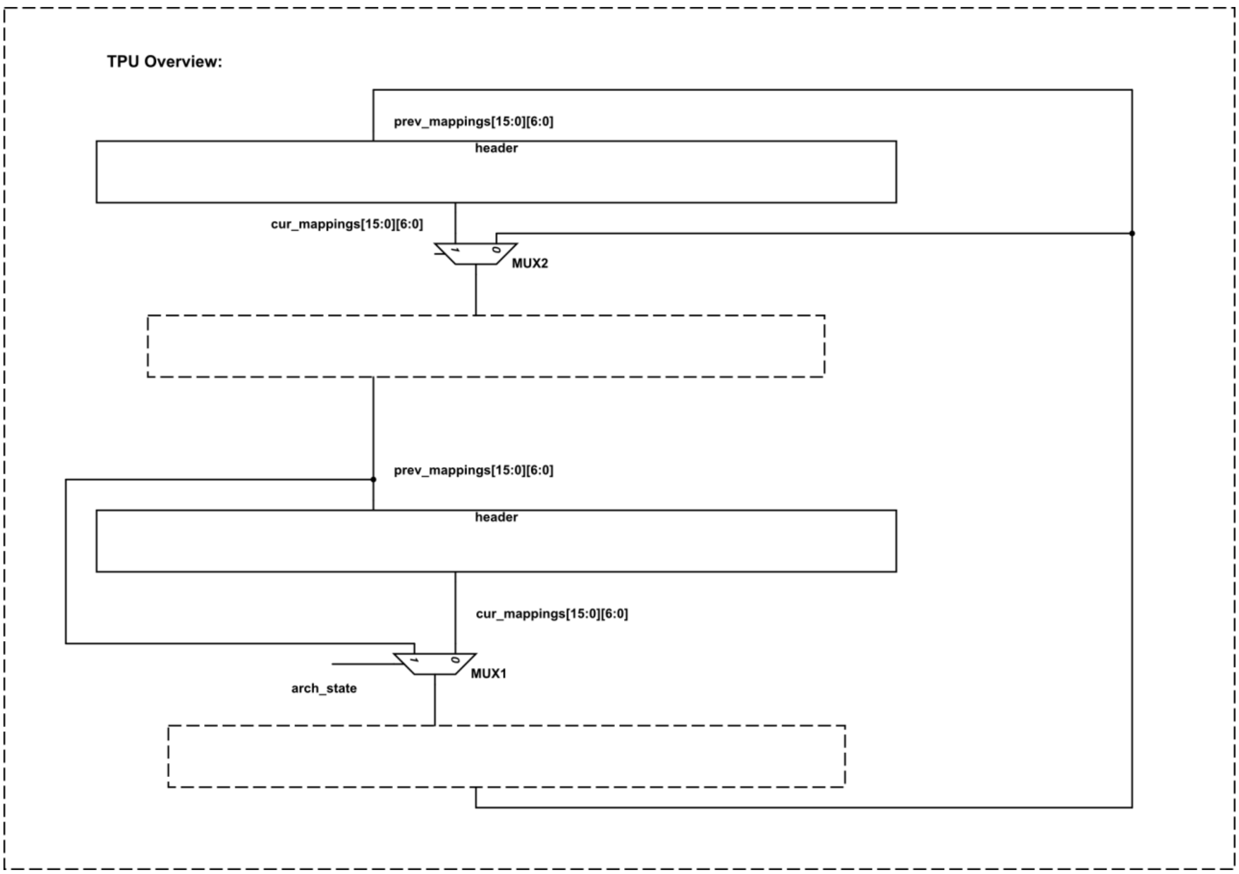
Signal	IO	Width	Source/Tar	Description

Name			get	
clk	I	1	global	global clk
rst_n	I	1	global	global reset
set	I	1	rob	set counter's value to data on val
val	I	6	rob	val to be set when set signal is asserted
inst_vld	I	4	allocation	instruction valid bits from allocation. 1 bit per instruction coming in
isq_ful	I	1	tpu	issue queue full signal
isq_lin_en	O	64	isq	individual issue queue slot enable signal
counter	O	4	tpu	counter's value for tpu to reference to determine if isq is full

4.4.4.3. Tag Propagation Unit

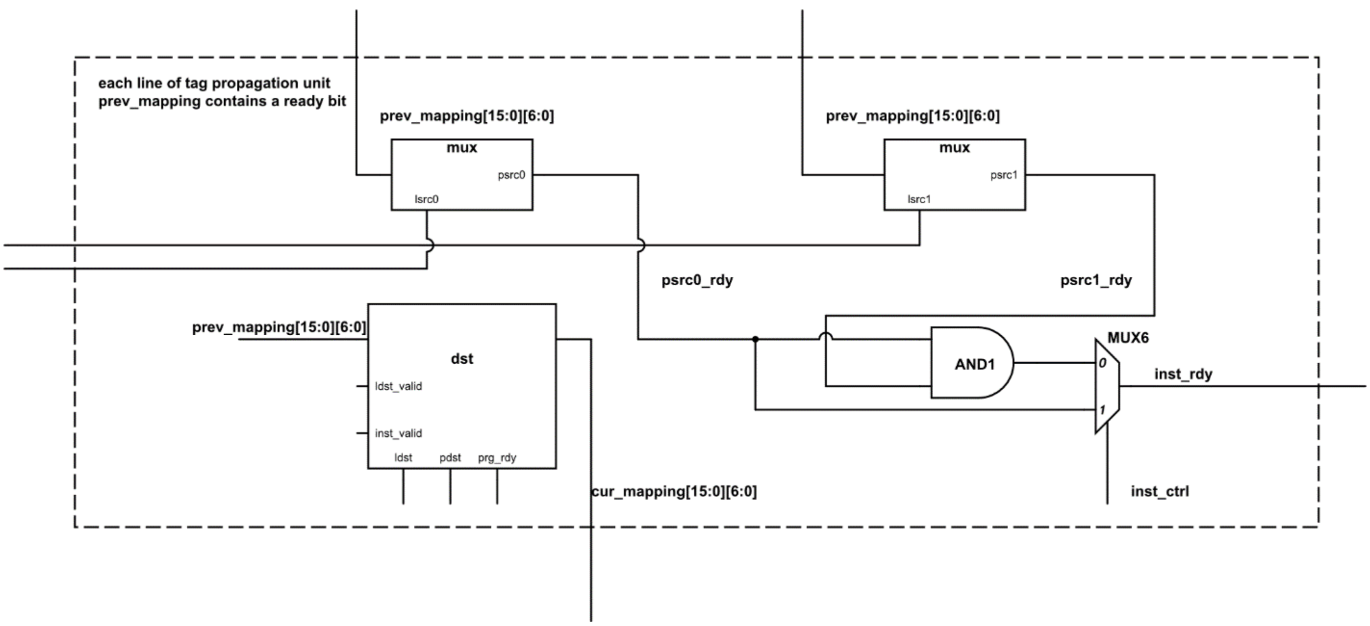
This block is in charge of register renaming and identifying instruction dependencies. The internal structure is implemented according to chapter 4.4 in Mitchell Hayenga's thesis.

The overview of the tag-propagation unit implementation is as follows:



There are two segment headers serving as storage elements for preserving architecture state alternatively. As discussed in the stage overview, at one time instance, only one segment header, serving as architecture state holds register mapping while the other is passed. Hence, all 64 instructions have the potential to be issued as long as their source registers and execution unit are ready. Architecture switch happens when all following criteria is satisfied:
1. Region below current header has all been dispatched (wait=0 for all valid instructions)
2. Counter's value is outside of region below current header
3. All physical registers involved in the logical to physical mapping have been resolved.

In each dotted square, there are 32 tag propagation line units each corresponding to one slot in issue queue. Each line in the tag-propagation unit has following structure shown in the graph. Register renaming and dependency check are performed in each line. When both of the source register becomes ready, the instruction ready bit will output high to the priority queue so that priority queue can know such instruction is ready to be issued. If an instruction writes to a register, it will also change the mappings from logical registers to physical ones so that the following instructions are use updated mappings.



Interface:

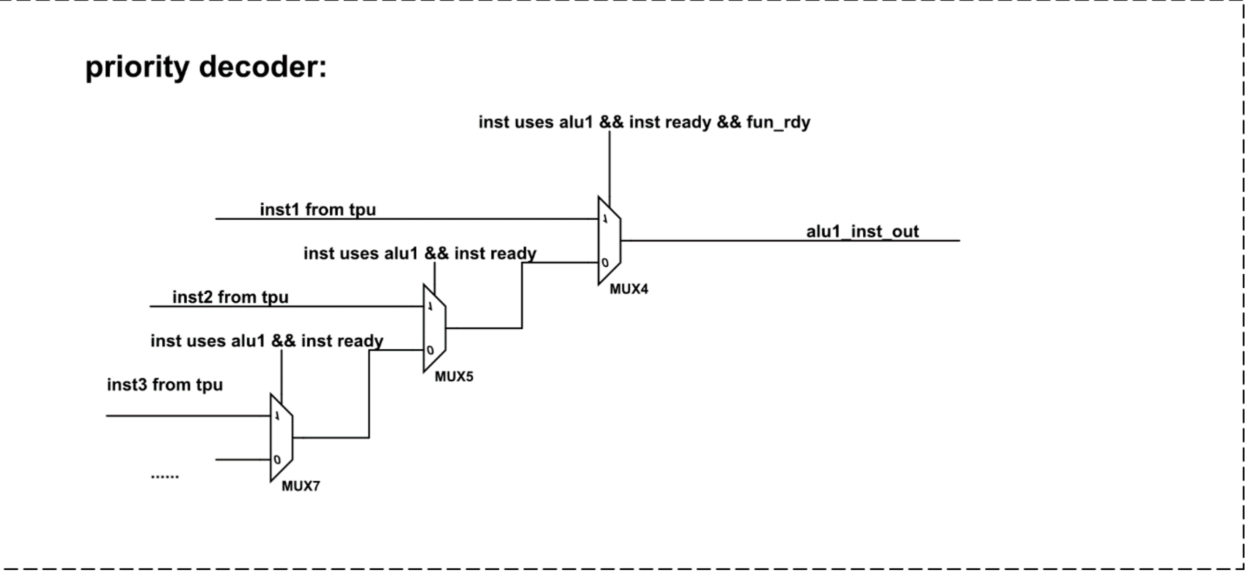
Signal Name	IO	Width	Source/Target	Description
clk	I	1	global	global clk
rst_n	I	1	global	global reset
isq_out_flat	I	4096	isq	instructions from issue queue
dst_reg_rdy	I	64	execution	physical register ready. 1 -- ready, 0 -- not ready

dst_rdy_reg_en	I	64	execution	physical register ready bit enable
counter	I	4	counter	current counter's value
tpu_inst_rdy	O	64	pdc	instruction in issue queue is ready to be sent
tpu_out_reo_flat	O	4032	pdc	tpu instruction output to pdc
fre_preg_out_flat	O	448	rob	physical register to be freed for each instruction
isq_ful	O	1	allocation	issue queue full
arch	O	1	isq	current architecture header
arch_swt_fls	O	64	isq	flush issue queue signal at architecture switch

4.4.4.4. Priority Queue

This customized priority queue takes in instructions after renaming, check whether they are ready to be issued and issue at maximum four instructions in each cycle to four different function units. Each output port of this priority queue is dedicated to one function unit. This priority queue issues ready-to-sent instructions who are upward closest to the architecture header first. When one function unit is occupied, the port dedicated to such function unit will not output new instructions.

Each output port of priority queue has following structure:



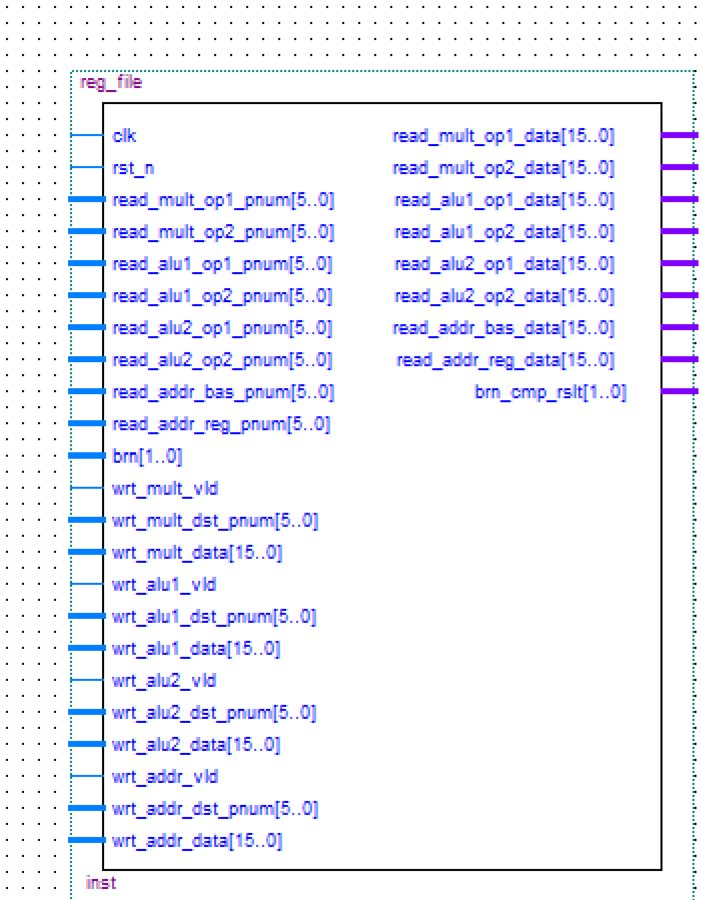
As illustrated in the diagram above, for each execution unit, the priority decoder chooses the top one instruction to send in each cycle. There are two possible priorities depending on which segment header is serving as architecture state. When the segment header on top is the architecture state, physically higher instruction should be given higher priority. When the segment header in the middle is the architecture state, the priority, from higher to lower, should be 33rd line to 64th line, and then 1st line to 32nd line. An array of mux are put in the middle of tag-propagation unit and priority queue to choose from xth line and (x+32)th line according to architecture state so that physically higher line in the priority decoder should always be given higher issue priority.

The priority queue will clear the wait bit in issue queue once an instruction is sent.

Interface:

Signal Name	IO	Width	Source/Target	Description
fun_rdy_frm_exe	I	4	execution	function unit ready signals from execution
tpu_out_reo_flat	I	4032	tpu	tpu instruction output
tpu_inst_rdy	I	64	tpu	instruction ready bit for each line of tpu
fre_preg_out_flat	I	448	tpu	physical registers to be freed for each instruction in tpu
pdc_clr_inst_wat	O	64	isq	clear instruction wait bit
mul_ins_to_rf_pdc	O	66	register file	multiply instruction output port
alu1_ins_to_rf_pdc	O	66	register file	alu, branch and jump instruction output port
alu2_ins_to_rf_pdc	O	66	register file	alu instruction output port
adr_ins_to_rf_pdc	O	66	register file	address adder instructions output port

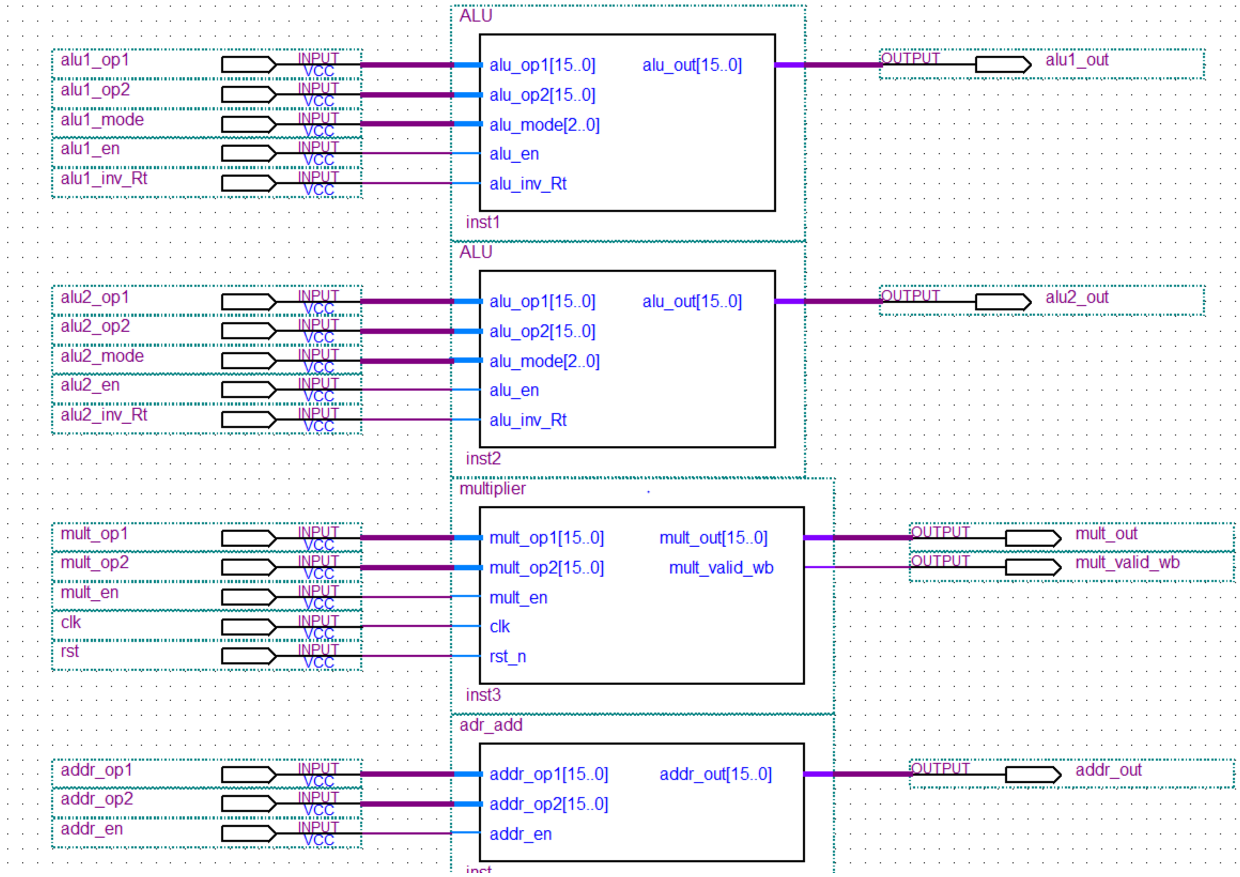
4.5. Register Files (RF)



In this stage we have a 64-entry physical register file. The reading and writing of the register file is designated to each function unit, thusly according to our four issue design, the register file could support 7 ports read (address adder for load and store instructions are only reading from one base register) and 4 ports write concurrently. Also, we decided to allocate all branch and jump instructions to the ALU1 function unit operand1 slot, which will generate corresponding control signals accordingly, like brnc_cmp_rslt or jump_base_rdy. As to the branch base register comparison results, we are going to do the comparison for each register write during the WriteBack stage. Whenever a register write is happening, the WriteBack stage will compare the data with zero and save the comparison result to register file, too. There will be two more bits within each entry of the register file for this functionality. Therefore, when a later branch instruction requires the comparison result of the base register, it is already there.

4.6. Execution (EX)

Four functional units can be used simultaneously, namely, two Arithmetic Logic Units (ALUs), a Multiplier and a Load/Store Unit. In the Load/Store Unit, a separate address adder calculates the memory address without occupying ALU resources. The second part of the unit is placed to the commit stage. A common data bus (CDB) is used to fulfill data forwarding and bypassing. Detailed module specifications are shown below.



4.6.1. Multiplier

Booth multiplication is a technique that allows for smaller, faster multiplication circuits, by recording the numbers that are multiplied. It is the standard technique used in chip design, and provides significant improvements over the "long multiplication" technique.

We planned to use the technique of radix 4 Booth recoding. The basic idea is that, instead of shifting and adding for every column of the multiplier term and multiplying by 1 or 0, we only take every second column, and multiply by ±1, ±2, or 0, to obtain the same results. So, to multiply by 7, we can multiply the partial product aligned against the least significant bit by -1, and multiply the partial product aligned with the third column by 2. The advantage of this method is the halving of the number of partial products. This is important in circuit design as it relates to the propagation delay in the running of the circuit, and the complexity and power consumption of its implementation. The advantage of radix 4 Booth algorithm is that we will reduce our cycles spending on multiplications by half.

Also, the multiplier we designed will have a free signal being set two clock before the computation complete. This is because that there is an extra register file stage between Issue and Execution stage. So we need to send the signal to Issue stage earlier instead of sending it when computation complete.

Basic operation is following:

Block	Partial Product
000	0
001	1 * Multiplicand
010	1 * Multiplicand
011	2 * Multiplicand
100	-2 * Multiplicand
101	-1 * Multiplicand
110	-1 * Multiplicand
111	0

4.6.2. ALUs

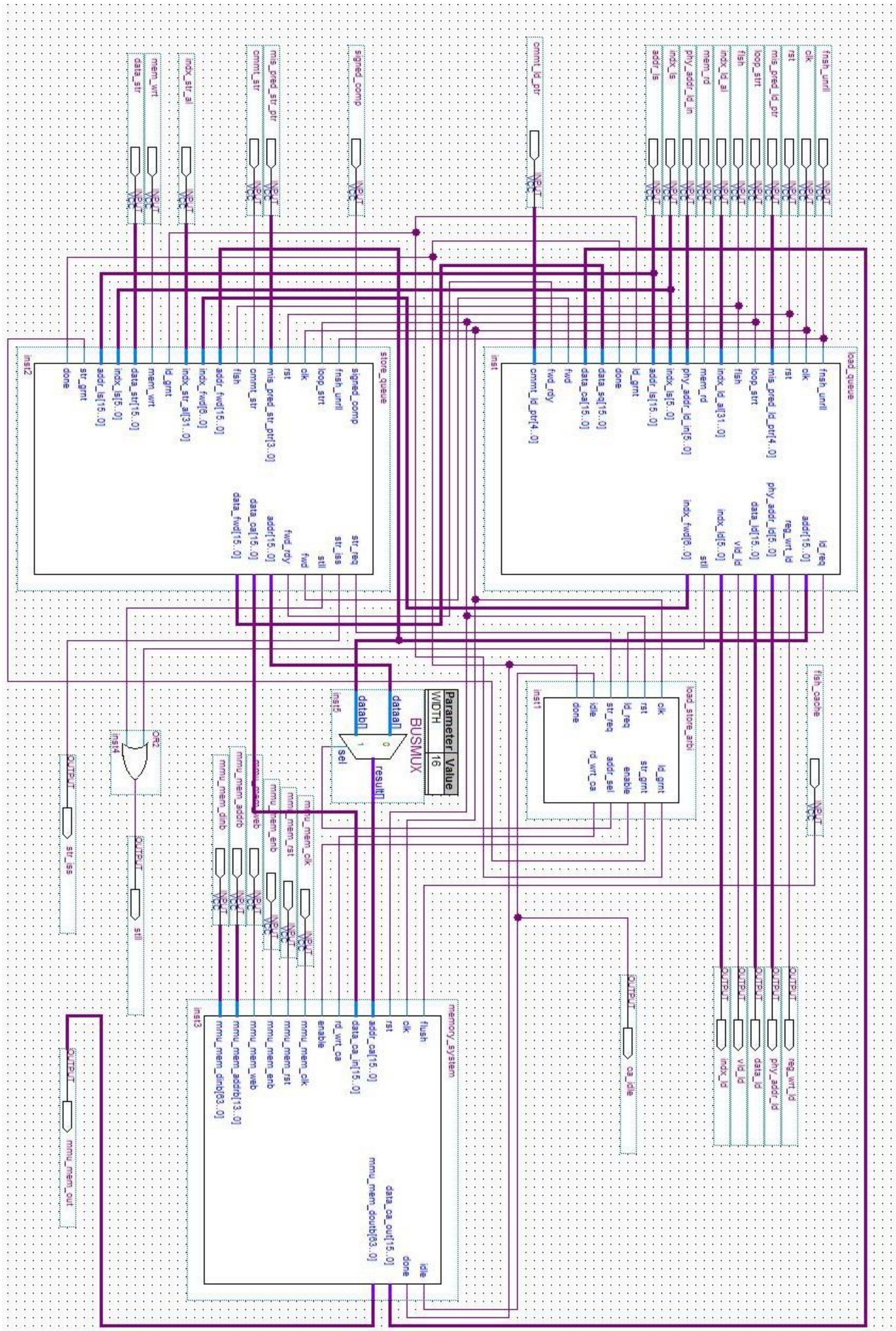
We have two general purpose ALUs in execution stage. Each of the ALU will have two input and an operation select signal. The ALU will be able to perform ADD, XOR, SHIFT, AND, OR operation. Each of the operation will take only one clock cycle. Moreover, we will have a valid signal and free signal indicating that the operation is finished. The valid signal will be sent in a package along with the physical register index.

We also designate ALU1 to handle the branch instructions and jump-reg-base instructions for register file access.

4.6.3. Address Adder

The address adder will only handle the load and store instruction. It is functionally the same with an adder.

4.7. Write Back (WB)



Signal Name	I/O	Width	Source/Target (blocks)	Description
Clk	I			No need to explain if you are using system level signals of standard name
Rst	I			--do--
addr_ls	I	16	EXE/WB pipeline register	the calculated memory address for load/store
phy_addr_ld_in	I	6	EXE/WB pipeline register	the physical register address to write the loaded data to
mem_wrt	I	1	EXE/WB pipeline register	indicate whether to write data to memory
mem_rd	I	1	EXE/WB pipeline register	indicate whether to read data from memory
indx_ls	I	6	EXE/WB pipeline register	the index of the load/store instruction
data_str	I	16	EXE/WB pipeline register	the data read from register file to write to memory
indx_ls_al	I	32	ALLOCATION	indices of load/store instructions to put into load/store queue together with valid bit
cmmt_str	I	1	COMMIT	indicate whether to execute one store in the store queue
cmmt_ld_ptr	I	5	COMMIT	used to direct the head's position
mis_pred_ld_ptr	I	5	COMMIT	the pointer position for the tail in LQ to jump to when misprediction occurs
mis_pred_str_ptr	I	4	COMMIT	the pointer position for the tail in SQ to jump to when misprediction occurs
flsh	I	1	COMMIT	indicate the time to flush speculative instructions in load/store queue
flsh_cache	I	1	MMU	used to signal flushing cache back to data memory
signed_comp	I	1	COMMIT	used to tell store queue whether to do signed comparison of indices when data forwarding is needed
mmu_mem_clk	I	1	MMU	used by MMU to control the behavior of the second port of data memory
mmu_mem_rst	I	1	MMU	used by MMU to control the behavior of the second port of data memory
mmu_mem_enb	I	1	MMU	used by MMU to control the behavior of the second port of data memory
mmu_mem_web	I	1	MMU	used by MMU to control the behavior of the second port of data memory
mmu_mem_dinb	I	64	MMU	used by MMU to write data to data memory through the second port
fnsh_unrll	I	1	decode	indicate whether loop unrolling is finished in order to mark loop end
loop_strt	I	1	decode	indicate whether a loop mode starts in order to mark loop start

data_ld	O	16	RF	the loaded data from memory to write to register file
phy_addr_ld	O	6	RF	the physical register address to write data_ld to
reg_wrt_ld	O	1	RF	indicate whether to write the register file

Above is the overview diagram of this stage together with the interface detailing the input and output signals. The writeback stage has two primary responsibilities. One is to write data back to physical register file and the other is processing memory accesses. The former one includes sending relevant signals received from EX/WB pipeline registers to physical register file and reorder buffer, and generating signals to write loaded data back to physical register file and notifying the reorder buffer of the completion of the load instruction. The latter is realized using separate load queue and store queue, a load store arbitrator together with the memory system, which is elaborated below.

Load queue

Load/store queue is used to handle memory operations, i.e. load and store. Separate load queue and store queue are adopted for the use of convenience. Our load queue has 24 entries. Below is a description of the fields in each entry.

Load Queue Entry

Valid	Addr	Index	phy_addr	Ready	Done
whether this entry is occupied by a load instruction	the memory address to read data from	indicate the program order of the instructions; used to tell whether the calculated address is for the load	the physical register address to write the loaded data to	whether this instruction is ready to execute (has its memory address calculated)	indicate whether this load has already been executed

In this queue, we have five pointers, i.e. head, tail, current, loop_start and loop_end. Head always points to the next load instruction to be committed. Tail always points to the entry for next coming load instruction to occupy. Current points to the load instruction being currently executed. Loop_start is used to mark the start and loop_end is used to mark the end of a loop.

After the Allocation stage, indices of the load instructions are sent to the load queue to occupy entries by setting the valid bit to 1 and storing the indices into the occupied entries. The tail indicates the position where to insert indices. After the Execution stage, the index together with the calculated address and the physical register address for the loaded data is sent to the load queue. Through index comparison, the corresponding entry is updated with Ready bit set to 1.

Load can be executed out-of-order to allow higher efficiency. We have a busy bit register to indicate whether a load can be issued. When the busy bit is 0 which means there is no load being executed, we choose the oldest ready load that has not been executed to execute and a request signal is sent to the load store arbitrator to ask for permission. And the selected load is pointed to by the current pointer. A shifter and a priority decoder is used to locate the position of the current point. A grant signal sent from load store arbitrator acknowledges the execution of the load. The busy bit register is set to 1. Since load is executed in an out-of-order manner, data forwarding from store queue is necessary to guarantee the correctness of loaded data. Each load sends the memory address and its index to the store queue. And the store queue gives feedback signals signifying whether a data forwarding occurs together with possible forwarded data. Besides, the load stalls upon any not-ready store instructions that should occur before the load in program order. After the data is read out from memory and the result of data forwarding is resolved, the done bit is set to 1, the data is written back to the physical register file and the index of the load instruction is sent to reorder buffer to notify the completion of the instruction. The busy bit register is then reset to 0. A finite state machine is used to control the execution behavior of the load.

The update of head pointer is controlled directly by the reorder buffer. Every clock cycle, the reorder buffer sends the position for the head to switch to. When there is a head movement meaning some loads are committed, the valid bit together with other information bits of the committed entries are cleared to make room for new instructions. When the load queue is full, a stall signal is sent out to disable front-end stages.

When a misprediction occurs, a flush signal is sent from reorder buffer to the load queue together with a load pointer. What the load queue needs to do is change the value of the tail pointer to the received load pointer value and flush out the load instructions in the mispredicted path by clearing their entry bits.

When the loop_strt signal goes high, a loop mode is entered. At this time, we record down the current tail value as the loop_start. When a fnsh_unrll signal goes high, loop unrolling is finished. At this time, we record down the next tail value minus one as the loop_end. In loop-mode execution, the tail is fixed at loop_end plus one since no more new instructions will be sent from the already-disabled allocation stage. The working of head and current is similar to that in normal execution with only one exception. When the head is pointed to loop_start again, all valid bits of entries between loop_start and loop_end are set to 1, indicating that they are occupied again by the load instructions in the unrolled loop body. When a misprediction occurs, the tail is set by the reorder buffer and the instructions in the mispredicted path are flushed out by clearing corresponding entry bits.

Store queue

Our store queue has 16 entries and below is a description of the fields in each entry.

Store Queue Entry

Valid	Addr	Data	Index	Ready
whether this entry is occupied by a	the memory address to write data	the data to be written into	indicate the program order of the instructions; used to tell whether the calculated address is for	whether this instruction is ready to execute (has its memory address

store instruction	to	memory	the store	calculated)

In this queue, we have four pointers, i.e. head, tail, loop_start and loop_end. And the meanings of the four pointers are the same as those in load queue.

After the Allocation stage, indices of the store instructions are sent to the store queue to occupy entries by setting the valid bit to 1 and storing the indices into the occupied entries. The tail indicates the position where to insert indices. After the Execution stage, the index together with the calculated address and the data read out from register file is sent to the store queue. Through index comparison, the corresponding entry is updated with Ready bit set to 1.

In our design, store is executed in order and only upon commitment from reorder buffer. When a store instruction is committed by reorder buffer, a commit signal is sent to the store queue to enable its execution and the reorder buffer should wait for an asserted feedback signal str_iss to continue committing. If the address has not been available yet, the commit is stuck until the address is ready and the store is issued to the memory system. Similar to load queue, a request signal is sent to the load store arbitrator to ask for execution permission and a grant signal is sent back for acknowledgement. A finite state machine is used to control this series of behavior. Upon the completion of a store, the information bits in the entry pointed by the head are cleared and then the head increments. When the store queue is full, a stall signal is sent out to disable the front-end stages.

When a misprediction occurs, a flush signal is sent from reorder buffer to the store queue together with a store pointer. What the store queue needs to do is change the value of the tail pointer to the received store pointer value and flush out the store instructions in the mispredicted path by clearing the entry bits.

When the loop_strt signal goes high, a loop mode is entered. At this time, we record down the current tail value as the loop_start. When a fnsh_unrll signal goes high, loop unrolling is finished. At this time, we record down the next tail value minus one as the loop_end. In loop-mode execution, the tail is fixed at loop_end plus one since no more new instructions will be sent from the already-disabled allocation stage. The working of head and current is similar to that in normal execution with only one exception. When the head is pointed to loop_start again, all valid bits of entries between loop_start and loop_end are set to 1, indicating that they are occupied again by the store instructions in the unrolled loop body. When a misprediction occurs, the tail is set by the reorder buffer and the instructions in the mispredicted path are flushed out by clearing corresponding entry bits.

Data forwarding

In this section, we elaborate the data forwarding process.

Since load is executed out of order, simply taking data from memory system can cause incorrect result. Thus, every time a load is issued to memory system as the load grant signal goes high, the store queue starts its index and address comparison to determine whether a data needs to be forwarded from itself to the load queue. There are several issues to be noted applied to different scenarios.

The first one is how to tell whether the store occurs before the load in program order considering simple index comparison is not enough to handle the “wrap-around” situation. For example, the index of the uncompleted store is 56 while the received load index is 3. Simply by index comparison, the load is thought to occur before the store but actually the load occurs after the store because the increasing of the index wraps around [0, 63]. To solve the problem, an extra bit is used to indicate whether “wrap-around” has occurred or not. In this way, the index of the load and store changes between [0000000, 1111111]. When the commit pointer stays at the first page with 0 as its MSB, usual unsigned comparison is used in index comparison. When the commit pointer goes to the second page with 1 as its MSB, signed comparison is used in index comparison. And the commit pointer used here is the head pointer of reorder buffer since the reorder buffer is the only one that has the accurate global execution information.

After the ordering problem is solved, the store queue needs to tell whether the load queue should wait by checking the ready bits of the stores before the load. A forward ready signal is used to control the waiting behavior of load queue. When the signal stays low, the load queue must wait. When the memory addresses of previous stores are resolved, the signal goes high to let the load instruction go.

After all addresses are resolved, whether data forwarding is needed and what data to forward to become the last task. Address comparison is used to determine whether data forwarding is needed. If there is no address match, no data forwarding is needed and this process is terminated for this load. If there exists any match, a forward signal goes high to notify the load queue to select the forwarded data from the store queue rather than the data read from memory. When there are multiple matches, we need to select the stored data of the nearest store to forward which is realized by a shifter and a priority decoder. Finally, the data forwarding is completed and the load queue can operate according to the signals sent from the store queue.

Load Store Arbitrator

Our cache is designed to do only one operation at a time so an arbitrator is used to avoid the contention between a simultaneous load and store. Also, when the memory system is busy handling one read/write operation, it’s the load store arbitrator’s responsibility to stall the execution of the pending store/load. In our design, store has a higher priority over load since store is executed in order while load is executed out of order. The coordination and control of load/store queue behavior is achieved through the generated grant signals. The input signals of the memory system are also controlled by this load store arbitrator. And the memory system feeds done and idle signals back to the arbitrator to inform its current status. A finite state machine is the main body of this arbitrator.

Memory System

The memory system mainly consists of three parts, namely, data cache, cache controller and data memory. The cache is 2-way set-associative with writeback and write-allocate policy. The total number of entries is 8. And only one operation can be done at a time. Separate input and output data buses are used for convenience. The cache directly sends data back if it’s a read hit. It writes data into its corresponding cache line if it’s a write hit with dirty bit set to 1. A cache controller is used to handle cache misses and possible writeback. If a read miss occurs, the address is forwarded to memory to fetch the cache line containing the wanted data and one way of the indexed entry is replaced by the fetched cache line with tag and state bits updated. Writeback is needed if the evicted cache line is dirty. An alternating replacement policy is used for choosing the victim way. The load operation is completed when the data is fetched and the possible writeback is finished. A done signal is sent from cache to the load queue and load store arbitrator. If a read miss occurs, the address is forwarded to memory to fetch the cache line and one way of the indexed entry is replaced by the fetched one with tag and state bits updated. Writeback is needed if the replaced cache line is dirty. The data from store queue is then written into the cache with dirty bit of the corresponding cache line set to 1. The done signal is asserted when the write is finished and the possible writeback is completed. The cache controller is responsible for telling the load store arbitrator whether the memory system is busy through the idle signal. The data memory also uses separate input and output data buses and a done signal is sent out to cache controller when the read/write operation is completed.

For the use convenience of MMU, the cache can flush all its dirty cache lines back to data memory assisted by the cache controller when the flush_cache signal goes high.

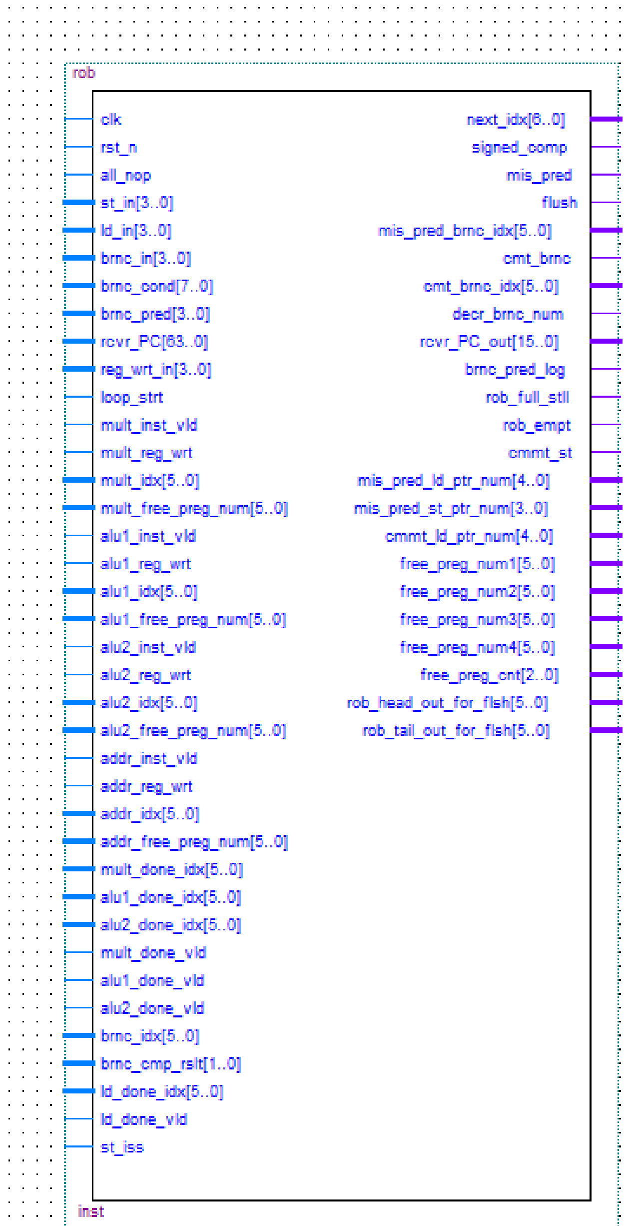
Cache Controller

The cache controller is basically a finite state machine. The input signals include miss_hit, wrt_bck and line_dirty from cache, done_mem from data memory, enable_cache from load store arbitrator and flush_cache from MMU. The output signals include rd_wrt_mem and mem_enable to memory, mem_rdy and one_line_flushed to cache and idle to load store arbitrator.

Memory

We used the CoreGen to generate a block RAM with two read/write ports. One read/write port is wrapped around by a memory interface to fake a data memory used by our processor. Relevant control signals are forwarded from cache and cache controller. The design purpose of the memory interface is to emulate the long-latency of memory accesses. The other read/write port is used by MMU with relevant control signals directly controlled by the unit. In this way, the processor treats the block RAM as a data memory with long-latency access while the MMU treats the block RAM as a simple data storage unit which only takes one clock cycle to read/write data.

4.8. Commit (Re-Order Buffer)



In this stage, we only have one primary module, i.e. reorder buffer (ROB). Below is an overview diagram of this stage with input signals listed on the left and output signals listed on the right. Table X is the The ROB is used to guarantee the correctness of the out-of-order execution which commits each instruction in program order and handles mispredictions. To match the capacity of the issue queue, our ROB is designed to be 64-entry, too. Below is a description of each group of bits in one ROB entry.

Vld
Done
Brnc
BrncPred
BrncCond[1:0]
RcvrPC[15:0]
Store
St_Ptr[3:0]
Ld_Ptr[4:0]
RegWrite
FrPRegNum[5:0]
signed_comp

Every completed instruction marks the Ready bit in its corresponding ROB entry. Most of the information bits are received from allocation stage. To match the amount of implemented functional units, the ROB commits at most four instructions in one clock cycle. There are two pointers used for entry insertion and commitment in this ROB. The tail always points to the next available entry in the ROB for next coming instruction. The head always points to the last committed instruction. So the tail increments upon receiving new instruction information from the Allocation stage. When the entry the tail points to after incrementing is not available, the ROB is full and a stall signal is sent out to disable front-end stages, i.e. fetch, decode and allocation. Every clock cycle, the index the tail points to is also sent to allocation stage for its use. The head increments upon committing instruction in the ROB. One instruction can be committed if its Ready bit is asserted or its Store bit is set. After committing this instruction, the entry it used to occupy is freed by setting its Available bit to 1. Every time a load or store instruction is committed, the corresponding load/store queue will be notified and then increment the head pointer in the load/store queue. If the instruction the head points to cannot be committed, the commit process is stuck until this instruction is ready to commit.

The use of head and tail pointers make misprediction handling very convenient. When a branch is found mispredicted through comparing predicted result and correct result, the entries between head+1 and tail is set to Available and tail is then set to head+1, which flushes out all the instructions along the mispredicted path. A misprediction signal together with the correct PC address to jump to is sent to the fetch stage to resume execution from the correct starting point. A flush signal is sent to issue queue, allocation stage and load/store quedue. The index of the mispredicted branch or jump instruction is sent to issue queue and allocation stage for architectural state recovery. A load pointer is sent to the load queue for its tail to jump to and a store pointer is sent to the store queue for its tail to jump to. In the ROB entry, the two pointers' values always indicate the positions in the load/store queue of the youngest load/store instruction after the instruction occupying this entry.

5. Software Structure

5.1. Assembler & Simulator

In order to provide input instructions to our designs, but not manually type all 1s and 0s, we implemented an assembler (using Java) that could compile instructions included in our ISA, detect any syntax errors, support labeling and produce compiled instruction in different formats. Additionally, we implemented several modes for the assembler and hence our simulator is also integrated with it as one of the modes. We would provide more details of our assembler below.

5.1.1. Usage

In Linux terminal, under assembler's directory (/ECE554_assembler/src), execute the following command:

java Assemble <input file path and name> <output file name> -m <mode>

input file path and name:

This could be any files, as long as the correct path is provided. However, only files with instructions that are in assembly syntax and included in our ISA could be successfully processed. For consistency reason, we named all our assembly files as type **.asm**

output file name:

This is name of the output file **without** type suffix. Plus, path should **not** be specified for this parameter, since all output files would be saved in **/ECE554_assembler/Test**

mode:

Here is assembler's working mode, different mode would produce different type of output files, and that is the reason why we do **not** need type suffix for **output file name**. Followings are details of each mode.

1. coe

In this mode, assembler will produce output file that ends with **_coe.coe,** and it will be stored under **/ECE554_assembler/Test. .coe** files are files with format that is able to be loaded by our real design. Specifically, it would be something like this:

```
memory_initialization_radix=16;

memory_initialization_vector=

c100c200c301c400,
```

c601c71715231234;

Where the first line specifies radix of our instructions, and the following lines specify all instruction in a given program (four instructions per line).

Example:

java Assemble add1.asm add1 -m coe

would produce **add1_coe.coe** under **/ECE554_assembler/Test**.

2. mif

In this mode, assembler will produce output file that ends with **_mif.mif,** and it will be stored under **/ECE554_assembler/Test. .mif** files are files with format that is able to be loaded by ModelSim Simulation. Specifically, it would be something like this:

```
1100000100000000110000100000000011000011000000001110001000000000

11000110000000001110001110001011100010101001000110001001000110100

0001001101010100001001110111011010110111111110110001010101010100
```

Unlike in **.coe** file where radix could be specified, **.mif** only accepts binary form. Here each line represents four instruction in binary format. Plus, assembler would concatenate 20 lines of 0s (a total of 80 **NOPs**) at the end of the program in order to avoid any overload during Modelsim Simulation

Example:

java Assemble add1.asm add1 -m mif

would produce **add1_mif.mif** under **/ECE554_assembler/Test**.

3. lst

In this mode, assembler will produce output file that ends with **_lst.lst**, and it will be stored under **/ECE554_assembler/Test. .lst** files are not able to be loaded by our real design on the board nor by Modelsim Simulation. Instead, this is a file for debug use, as it contains useful information such as **PC, original instructions, binary form and hex form** for each instruction. Specifically, it would be something like this:

```
PC: 0000 Binary: 1100000100001111 Hex: c10f ldi r1, 15    // load 15 to r1

PC: 0001 Binary: 1100001000010100 Hex: c214 ldi r2, 20    // load 20 to r2

PC: 0002 Binary: 0001001100010010 Hex: 1312 add r3, r1, r2// expect r3 = 35
```

As we can see, detailed information of each instruction is generated for a given **.asm** file, and we could therefore use it to check whether our assembler is working well and to trace waveforms in Modelsim Simulation more easily.

Example:

java Assemble add1.asm add1 -m lst

would produce **add1_lst.lst** under **/ECE554_assembler/Test**.

4. sim

In this mode, our assembler works as an simulator. In other words, it will **a)** run the program by itself, **b)** record down registers' status and memory's status in **_sim.sim** file after each instruction is executed, **c)** count total number of executed instruction, and **d)** produce registers' and memory's final status in **sim_reg.dump** and **sim_mem.dump** under project's root directory (/looper). We would explain these one by one

a) We use two arrays to imitate registers and memory, respectively, and therefore we are able to run all instructions (including branches and jumps) purely using java, and monitor all changes in both registers and memory.

b) After each instruction is executed by our java code, the **register array** will be printed out to **_sim.sim** file. However, the **memory array** will only be displayed when an **str** or **ldr** instruction is encountered. Additionally, since the printing out the entire memory would be too many and too tedious to trace, we only print out the modified memory slots.

Example:

java Assemble add1.asm add1 -m sim

would produce **add1_sim.sim** under/**ECE554_assembler/Test**, **sim_reg.dump and sim_mem.dump** under **root**.

5.1.2. Functionality

5.1.2.1. Syntax check

Our assembler would check syntax of the given input file and report any syntax errors. In other words, although we accept any files as input files, only files that are formed by correct syntax could be processed. Several syntax rules are described below

1. Each line should contain only either one instruction or one label

2. Operands in each instruction should be separated by a comma or one space; but opcode and the first operand should be separated only by space

3. For instructions, they should be valid, which means they should be included in our ISA

4. For labels, each label should start with a dot and end with colon. **e.g. .first_label:**

5. Any texts after "//" would be consider as comments, and "//" is only used to do one-line comment

5.1.2.2. Immediate check

Since our ISA includes instructions that contain immediate field, such as **ldr and str**, we should ensure that the input immediate is bounded in a valid range. Our assembler achieves this functionality. For example, for a **ldr** or an **str**, number of immediate bits are 4, so immediate field for these two instruction should range **from -8 to 7**, any value that is outside this range will be reported by our assembler

Example:

ldr r0, r1, 16 **invalid and will be reported**

ldr r0, r1, 2 **valid**

5.1.2.3. Register check

Similar to the immediate check, our assembler also does register check, as we only have sixteen regular registers in our design, we would not process any registers that are not between r0 and r15 (inclusively).

Example:

add r1, r2, r19 **invalid and will be reported** add r1, r2, r3 **valid**

5.2. Benchmarks

We have designed multiple small benchmarks to test each instruction individually, but here we would like to only cover our large benchmarks, which are designed to be "loop intensive" and therefore could test the loop mode in our design

5.2.1. Fibonacci sequence

In this benchmark, we simply use our design to generate the first 23 Fibonacci sequence numbers, and store them in memory, so that we can view results via our MMU. The reason we only generate the first 23 is due to the fact that our registers are 16 bits long, and the 23rd number would the largest number we could generate

5.2.2. Bubble sort

In this benchmark, we first initialize an unordered array in memory, then we perform bubble sorts on that array, and store result back to memory. We have a couple versions of bubble sort, the only difference is array size. This will allow us to first test if sorting five numbers is working, and continue on sorting more numbers.

5.2.3. Fixed point iteration

In this benchmark, we simply perform the fixed point iteration algorithm to calculate the square root of a given number, then store the results in memory. The input of this benchmark is updatable, so we could start from test with small input and step towards tests with larger inputs to see if our design could go through all of them.

5.2.4. Mult sequence generator

In this benchmark, we simply want to do intensive loop and intensive multiplication simultaneously. What we designed here is pretty much similar to the **Fibonacci sequence generator**, but here the **(n+1)th number = (n)th number * 2 + 1**. Similarly, all results will be stored in memory as well.

5.3. Helper Script

In order to compile test codes, simulate in our assembler or in Modelsim and compare result quickly, we implemented several shell scripts that enhance our efficiency.

5.3.1. run script

Usage: **./run <test file name>**

This script will run the vsim compile and simulate tools on our projects, simulate our codes using our assembler, and finally compare results between Modelsim simulation and our assembler's simulation, and prompt out whether our codes work correctly or not

5.3.2. run_java script

Usage: **./run_java -normal**

This script will run our simulator on all of our tests file, and generate .lst, .coe, .mif versions for each of them.

5.3.3. compare script

Usage: **./compare**

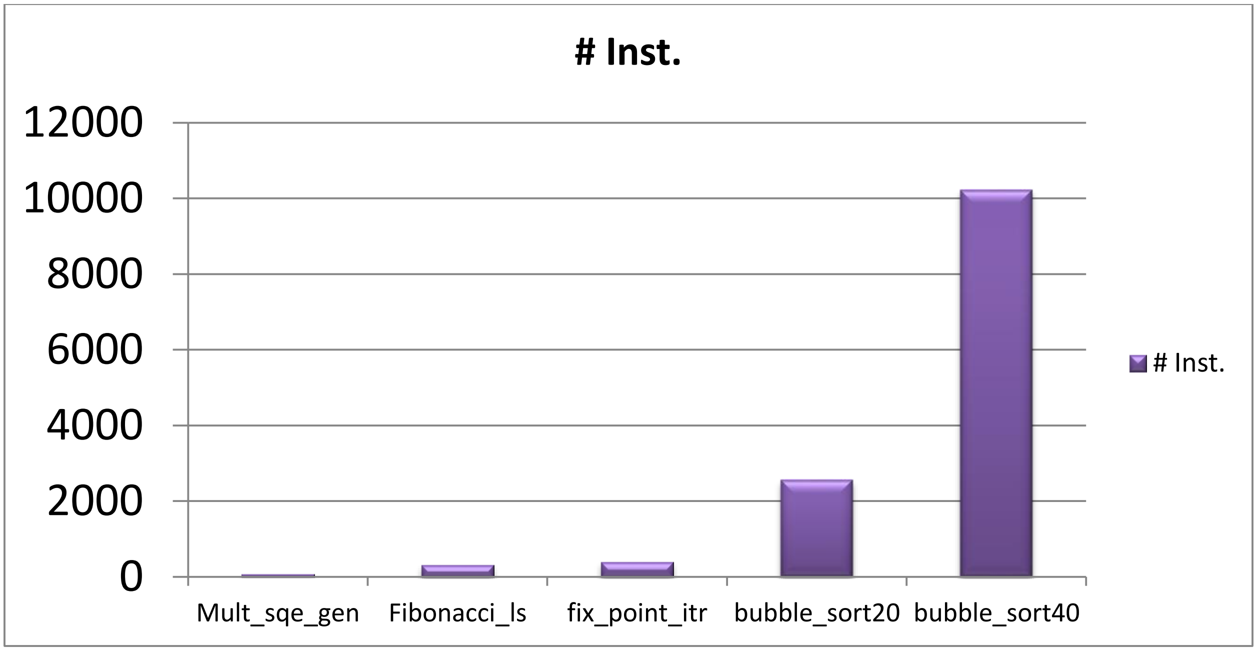
This script will invoke another Java program we implemented, which will read contents in **sim_mem.dump, sim_reg.dump, reg.dump, mem.dump and map.dump**, and compare results between them in order to determine whether our design works correctly or not

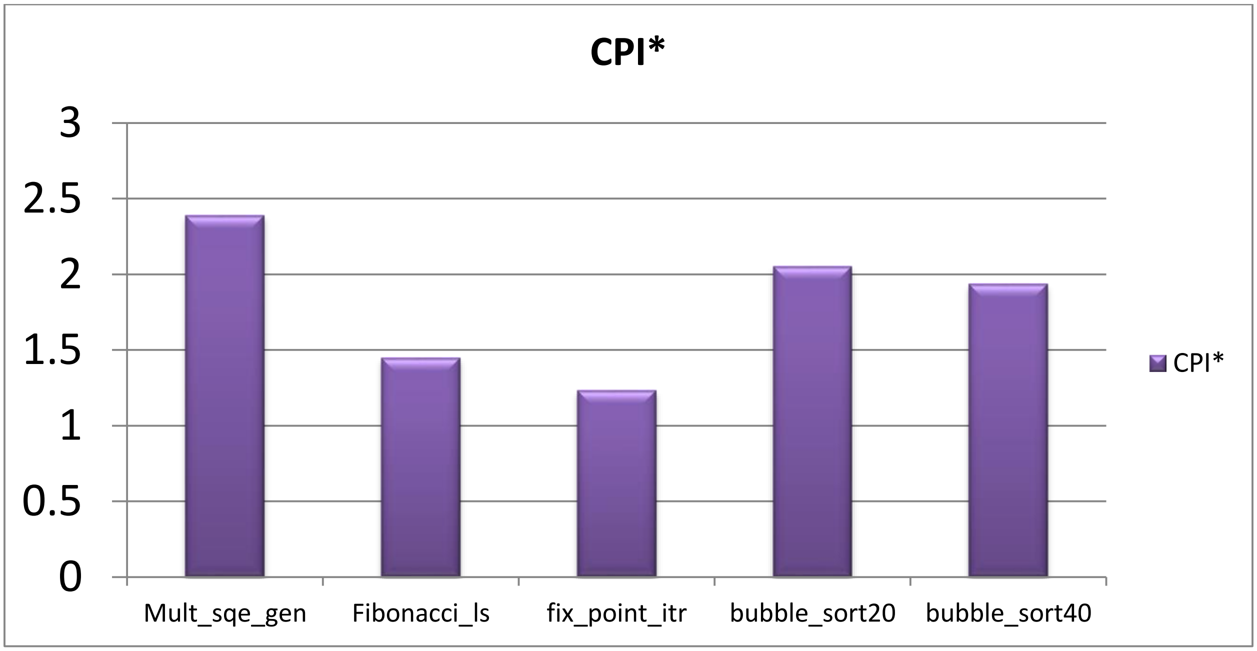
6. Implementation Results

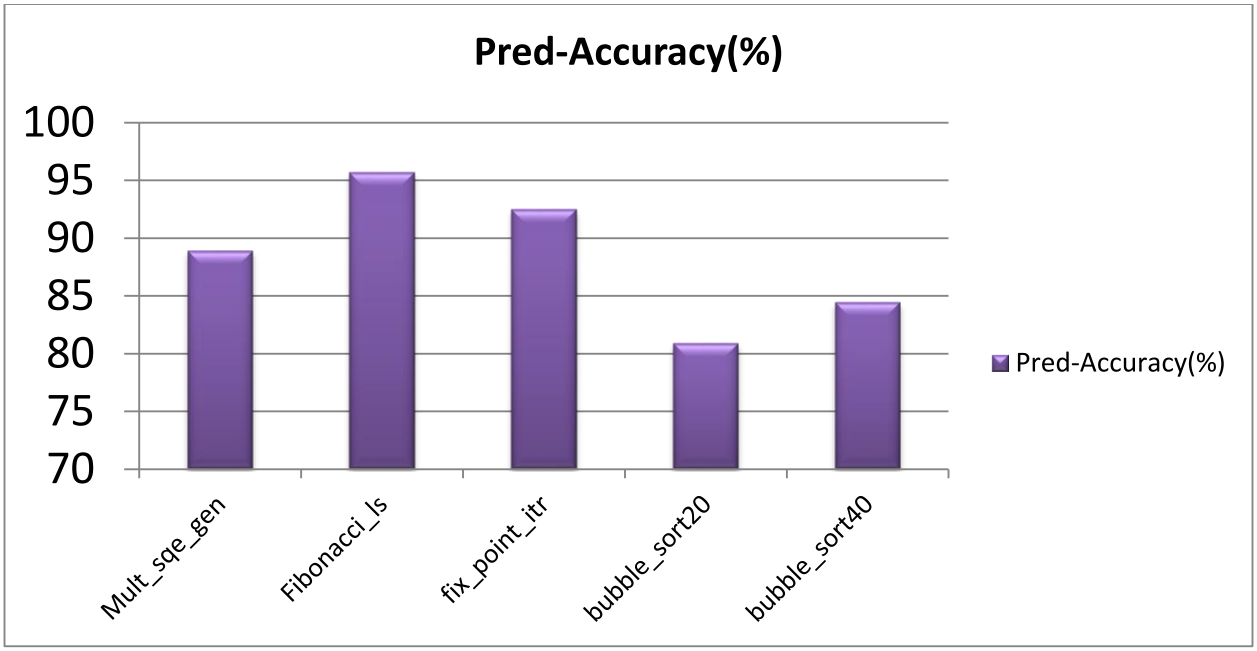
6.1. Synthesis Reports

Minimum period:	7.060ns (Maximum Frequency: 141.633MHz)
Number of Slice LUTS:	55006 out of 69120 ≈ 79%
Number of Slice LUT-Flip Flop pairs:	56159 out of 69120 ≈ 81%

6.2. Evaluation Results







Since we didn't have enough time to implement any more complex branch predictors, our performance got suffered a lot because of that. Also, the multiple branch prediction makes that worse. Anyway, if there could have been a more accurate predictor, we are sure that the performance will increase tremendously.

7. Contribution Report

Fan Zhu: Execution stage design and implementation; Decode stage debugging.

Haoyan Jia: Regfile stage and Re-order Buffer design and implementation; Writeback stage debugging.

Jing Tu: Writeback stage design and implementation; On board synthesis and testing.

Junjue Wang: Issue stage and MMU design and implementation; On board synthesis and testing.

Yuewen Lei: Fetch stage design and implementation. On board synthesis and testing.

Zheng Ling: Allocation stage design and implementation; Fetch stage debugging.

Zhexuan Liu: Decode stage design and implementation; Software construction.

Signature:

朱凡 贾昊龙天 涂靖 王俊珏
雷悦雯 凌峥 刘哲轩