

ESE 545 Cell SPU-Lite:

Dual Issue (MIMD)

Multimedia Processor

Table Of Contents

Acknowledgements	3
Introduction	4
Instruction Set Architecture	5
Design Implementation	27
Instruction Fetch stage	27
Decode	27
Register File	28
Data Forwarding	28
Dual-Execution Pipeline	29
Testing	29
Execution Units Functionality	30
Data Forwarding	33
Cache hit and miss penalty	35
RAW Hazard Detections	35
Simultaneous instruction pipeline/dependency detection	36
Branch mispredictions and loops	37
4x4 Matrix multiplication Example	39
Appendix: Code	42
Top Module	42
Instruction Fetch Stage	43
Instruction Cache	45
Branch Target Buffer	47
Main Memory For Instructions	50
Decode Stage	51
Processing Unit	74
Register File	77
Data Forwarding Unit	79
Even Pipe Module	85
Odd Pipe Module	87
Byte Unit	89
Simple Fixed 1 Unit	91
Simple Fixed 2 Unit	98
Single Precision Unit	103
Branch Unit	108
Permute Unit	110
Localstore Unit	112
Parser Code	115

Acknowledgements

At this juncture, we would like to thank Dr. Mikhail Dorojevets and Stony Brook University who in collaboration have provided us with this incredible opportunity to work on such an interesting project. We would like to thank Dr. Mikhail Dorojevets for all the guidance and support throughout the semester which was vital for us not only to complete the project on time but also in order to make this project a very interesting one.

This project has turned out to be an incredible learning opportunity for us. It has certainly helped us appreciate the field of Computer Architecture and the Sony Cell SPU engineers. Through this project, we were able to learn about the various aspects of designing a microprocessor starting from picking an instruction set to work and ending with Branch prediction and cache memory structures.

We would also like to thank Sony Enterprises and Dr. Mikhail Dorojevets for making the Cell SPU ISA available as a starting base for our project. The project would have been impossible without the help of the ISA. The thorough, well-structured and expansive ISA of Cell SPU provided us with ample opportunities to prune the ISA while retaining maximum possible functionality in the processor.

We would also like to take this opportunity to thank Modelsim software for providing us the platform to test our codes and ascertain the functionality of our processor. Modelsim's support for SystemVerilog and its simulation framework has helped us in a huge way in achieving our goal.

Introduction

The goal of this project is to learn modern multimedia processor design using Sony Cell SPU processor architecture and its implementation as an example. Our own version of a dual issue Multiple Instruction Multiple Data (MIMD) multimedia processor was implemented in System Verilog. We have picked a subset of instructions to implement from the original Cell SPU instruction set. Some complicated aspects of the Sony Cell SPU were either dropped or modified for simplicity and expectation to reach project deadline.

Our implementation consists of an Instruction Fetch stage, Decode stage, and a processing unit consisting of a single cycle Register File stage, a dual-pipelined 7 pipe Execution stage and a write back stage. As opposed to the original Cell SPU architecture this implementation does not have an Instruction Fetch pipeline, Channel execution unit, or Routing stage, and only has 1 Register File stage (opposed to 2), a combined Decode and Issue stage, a 1 cycle Branch execution unit, 7 pipes in execution stage instead of 8, and simplified memory management.

In the following sections, we outline our instruction set selection, descriptions of the instructions in the logical sense. Then, we follow it with brief explanations of each of the building blocks of our design and the basic intuition behind some of our key design choices. Finally, we illustrate the testing results for various units, hazard detection and safe execution of parsed Cell SPU machine code in our design.

Instruction Set Architecture

Instruction Name	Instruction Pneumonic	RTL Description	Unit Name (ID)	Pipe	Latency (Cycles)
Load Quadword (d-form)	lqd rt, symbol(ra)	$LSA \leftarrow (\text{RepLeftBit}(I10 \parallel 0b0000,32) + RA0:3) \& LSLR \& 0xFFFFFFFF0,$ $RT \leftarrow \text{LocStor}(LSA, 16)$	Local Store	Odd	6
Load Quadword (a-form)	lqa rt, symbol	$LSA \leftarrow \text{RepLeftBit}(I16 \parallel 0b00,32) \& LSLR \& 0xFFFFFFFF0,$ $RT \leftarrow \text{LocStor}(LSA,16)$	Local Store	Odd	6
Load Quadword IR (a-form)	lqr rt, symbol	$LSA \leftarrow (\text{RepLeftBit}(I16 \parallel 0b00,32) + PC) \& LSLR \& 0xFFFFFFFF0,$ $RT \leftarrow \text{LocStor}(LSA,16)$	Local Store	Odd	6
Store Quadword (d-form)	stqd rt, symbol(ra)	$LSA \leftarrow (\text{RepLeftBit}(I10 \parallel 0b0000,32) + RA0:3) \& LSLR \& 0xFFFFFFFF0,$ $\text{LocStor}(LSA,16) \leftarrow RT$	Local Store	Odd	6
Store Quadword (a-form)	stqa rt, symbol	$LSA \leftarrow \text{RepLeftBit}(I16 \parallel 0b00,32) \& LSLR \& 0xFFFFFFFF0,$ $\text{LocStor}(LSA,16) \leftarrow RT$	Local Store	Odd	6
Store Quadword IR (a-form)	stqr rt, symbol	$LSA \leftarrow (\text{RepLeftBit}(I16 \parallel 0b00,32) + PC) \& LSLR \& 0xFFFFFFFF0,$ $\text{LocStor}(LSA,16) \leftarrow RT$	Local Store	Odd	6
Immediate Load Halfword	ilh rt, symbol	$s \leftarrow I16, RT^{0:1} \leftarrow s, RT^{2:3} \leftarrow s, RT^{4:5} \leftarrow s, RT^{6:7} \leftarrow s, RT^{8:9} \leftarrow s, RT^{10:11} \leftarrow s, RT^{12:13} \leftarrow s, RT^{14:15} \leftarrow s$	Simple Fixed 1	Even	2

Immediate Load Halfword Upper	ilhu rt, symbol	$t \leftarrow l16 \mid \mid 0x0000, RT^{0:3} \leftarrow t, RT^{4:7} \leftarrow t, RT^{8:11} \leftarrow t, RT^{12:15} \leftarrow t$	Simple Fixed 1	Even	2
Immediate Load Halfword Lower	iohl rt, symbol	$t \leftarrow 0x0000 \mid \mid l16, RT^{0:3} \leftarrow RT^{0:3} \mid t, RT^{4:7} \leftarrow RT^{4:7} \mid t, RT^{8:11} \leftarrow RT^{8:11} \mid t, RT^{12:15} \leftarrow RT^{12:15} \mid t$	Simple Fixed 1	Even	2
Immediate Load Word	il rt, symbol	$t \leftarrow \text{RepLeftBit}(l16,32),$ $RT^{0:3} \leftarrow s, RT^{4:7} \leftarrow s, RT^{8:11} \leftarrow s, RT^{12:15} \leftarrow s$	Simple Fixed 1	Even	2
Immediate Load Address	ila rt, symbol	$t \leftarrow l40 \mid \mid l18,$ $RT^{0:3} \leftarrow s, RT^{4:7} \leftarrow s, RT^{8:11} \leftarrow s, RT^{12:15} \leftarrow s$	Simple Fixed 1	Even	2
Form Select Mask for Bytes Immediate	fsmbi rt, symbol	$s \leftarrow l16,$ for j = 0 to 15 if $s_j = 0$ then $r^j \leftarrow 0x00$ else $r^j \leftarrow 0xFF$ end $RT \leftarrow r$	Simple Fixed 1	Even	2
Add Halfword	ah rt, ra, rb	$RT^{0:1} \leftarrow RA^{0:1} + RB^{0:1}, RT^{2:3} \leftarrow RA^{2:3} + RB^{2:3}, RT^{4:5} \leftarrow RA^{4:5} + RB^{4:5},$ $RT^{6:7} \leftarrow RA^{6:7} + RB^{6:7}, RT^{8:9} \leftarrow RA^{8:9} + RB^{8:9}, RT^{10:11} \leftarrow RA^{10:11} + RB^{10:11},$ $RT^{12:13} \leftarrow RA^{12:13} + RB^{12:13}, RT^{14:15} \leftarrow RA^{14:15} + RB^{14:15}$	Simple Fixed 1	Even	2
Add Halfword Immediate	ahi rt, ra, value	$s \leftarrow \text{RepLeftBit}(l10,16), RT^{0:1} \leftarrow RA^{0:1} + s, RT^{2:3} \leftarrow RA^{2:3} + s, RT^{4:5} \leftarrow$ $RA^{4:5} + s, RT^{6:7} \leftarrow RA^{6:7} + s, RT^{8:9} \leftarrow RA^{8:9} + s, RT^{10:11} \leftarrow RA^{10:11} + s,$ $RT^{12:13} \leftarrow RA^{12:13} + s, RT^{14:15} \leftarrow RA^{14:15} + s$	Simple Fixed 1	Even	2
Add word	a rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} + RB^{0:3}, RT^{4:7} \leftarrow RA^{4:7} + RB^{4:7}, RT^{8:11} \leftarrow RA^{8:11} + RB^{8:11},$ $RT^{12:15} \leftarrow RA^{12:15} + RB^{12:15}$	Simple Fixed 1	Even	2
Add Word Immediate	ai rt, ra, value	$t \leftarrow \text{RepLeftBit}(l10,32),$ $RT^{0:3} \leftarrow RA^{0:3} + t, RT^{4:7} \leftarrow RA^{4:7} + t, RT^{8:11} \leftarrow RA^{8:11} + t, RT^{12:15} \leftarrow$ $RA^{12:15} + t$	Simple Fixed 1	Even	2

Subtract from Halfword	sfh rt, ra, rb	$RT^{0:1} \leftarrow RB^{0:1} + (\neg RA^{0:1}) + 1, RT^{2:3} \leftarrow RB^{2:3} + (\neg RA^{2:3}) + 1, RT^{4:5} \leftarrow RB^{4:5} + (\neg RA^{4:5}) + 1, RT^{6:7} \leftarrow RB^{6:7} + (\neg RA^{6:7}) + 1, RT^{8:9} \leftarrow RB^{8:9} + (\neg RA^{8:9}) + 1, RT^{10:11} \leftarrow RB^{10:11} + (\neg RA^{10:11}) + 1, RT^{12:13} \leftarrow RB^{12:13} + (\neg RA^{12:13}) + 1, RT^{14:15} \leftarrow RB^{14:15} + (\neg RA^{14:15}) + 1$	Simple Fixed 1	Even	2
Subtract from Halfword immediate	sfhi rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,16), RT^{0:1} \leftarrow t + (\neg RA^{0:1}) + 1, RT^{2:3} \leftarrow t + (\neg RA^{2:3}) + 1, RT^{4:5} \leftarrow t + (\neg RA^{4:5}) + 1, RT^{6:7} \leftarrow t + (\neg RA^{6:7}) + 1, RT^{8:9} \leftarrow t + (\neg RA^{8:9}) + 1, RT^{10:11} \leftarrow t + (\neg RA^{10:11}) + 1, RT^{12:13} \leftarrow t + (\neg RA^{12:13}) + 1, RT^{14:15} \leftarrow t + (\neg RA^{14:15}) + 1$	Simple Fixed 1	Even	2
Subtract from word	sf rt, ra, rb	$RT^{0:3} \leftarrow RB^{0:3} + (\neg RA^{0:3}) + 1, RT^{4:7} \leftarrow RB^{4:7} + (\neg RA^{4:7}) + 1, RT^{8:11} \leftarrow RB^{8:11} + (\neg RA^{8:11}) + 1, RT^{12:15} \leftarrow RB^{12:15} + (\neg RA^{12:15}) + 1$	Simple Fixed 1	Even	2
Subtract from word immediate	sfi rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,32), RT^{0:3} \leftarrow t + (\neg RA^{0:3}) + 1, RT^{4:7} \leftarrow t + (\neg RA^{4:7}) + 1, RT^{8:11} \leftarrow t + (\neg RA^{8:11}) + 1, RT^{12:15} \leftarrow t + (\neg RA^{12:15}) + 1$	Simple Fixed 1	Even	2
Multiply	mpy rt, ra, rb	$RT^{0:3} \leftarrow RA^{2:3} * RB^{2:3}, RT^{4:7} \leftarrow RA^{6:7} * RB^{6:7}, RT^{8:11} \leftarrow RA^{10:11} * RB^{10:11}, RT^{12:15} \leftarrow RA^{14:15} * RB^{14:15}$	Single Precision 1	Even	7
Multiply Unsigned	mpyu rt, ra, rb	$RT^{0:3} \leftarrow RA^{2:3} * RB^{2:3}, RT^{4:7} \leftarrow RA^{6:7} * RB^{6:7}, RT^{8:11} \leftarrow RA^{10:11} * RB^{10:11}, RT^{12:15} \leftarrow RA^{14:15} * RB^{14:15}$	Single Precision 1	Even	7
Multiply immediate	mypi rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,16), RT^{0:3} \leftarrow RA^{2:3} * t, RT^{4:7} \leftarrow RA^{6:7} * t, RT^{8:11} \leftarrow RA^{10:11} * t, RT^{12:15} \leftarrow RA^{14:15} * t$	Single Precision 1	Even	7
Multiply unsigned immediate	mpyui rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,16), RT^{0:3} \leftarrow RA^{2:3} * t, RT^{4:7} \leftarrow RA^{6:7} * t, RT^{8:11} \leftarrow RA^{10:11} * t, RT^{12:15} \leftarrow RA^{14:15} * t$	Single Precision 1	Even	7
Multiply and add	mpya rt, ra, rb, rc	$t0 \leftarrow RA^{2:3} * RB^{2:3}, t1 \leftarrow RA^{6:7} * RB^{6:7}, t2 \leftarrow RA^{10:11} * RB^{10:11}, t3 \leftarrow RA^{14:15} * RB^{14:15}, RT^{0:3} \leftarrow t0 + RC^{0:3}, RT^{4:7} \leftarrow t1 + RC^{4:7}, RT^{8:11} \leftarrow t2 + RC^{8:11}, RT^{12:15} \leftarrow t3 + RC^{12:15}$	Single Precision 1	Even	7

Multiply high	mpyh rt, ra, rb	$t0 \leftarrow RA^{0:1} * RB^{2:3}, t1 \leftarrow RA^{4:5} * RB^{6:7}, t2 \leftarrow RA^{8:9} * RB^{10:11}, t3 \leftarrow RA^{12:13} * RB^{14:15},$ $RT^{0:3} \leftarrow t0^{2:3} 0x0000, RT^{4:7} \leftarrow t1^{2:3} 0x0000, RT^{8:11} \leftarrow t2^{2:3} 0x0000, RT^{12:15} \leftarrow t3^{2:3} 0x0000$	Single Precision 1	Even	7
Count leading Zeroes	clz rt, ra	for j = 0 to 15 by 4 $t \leftarrow 0$ $u \leftarrow RA^{j:4}$ For m = 0 to 31 if $u_m = 1$ then leave $t \leftarrow t + 1$ end $RT^{j:4} \leftarrow t$ end	Simple Fixed 1	Even	2
Count ones in bytes	cntb rt, ra	for j = 0 to 15 $c = 0$ $b \leftarrow RA_j$ for m = 0 to 7 if $b_m = 1$ then $c \leftarrow c + 1$ end $RT_j \leftarrow c$ end	Byte	Even	4

Form select mask for bytes	fsmb rt, ra	$s \leftarrow RA^{2:3}$ for j = 0 to 15 If $s_j = 0$ then $r_j \leftarrow 0x00$ else $r_j \leftarrow 0xFF$ end $RT \leftarrow r$	Simple Fixed 1	Even	2
Form Select mask for halfwords	fsmh rt, ra	$s \leftarrow RA^3$ for j = 0 to 15 If $s_j = 0$ then $r_j \leftarrow 0x00$ else $r_j \leftarrow 0xFF$ end $RT \leftarrow r$			
Form select mask for words	fsm rt, ra	$s \leftarrow RA^{28:31}$ $k = 0$ for j = 0 to 3 If $s_j = 0$ then $r^{k::4} \leftarrow 0x00000000$ else $r^{k::4} \leftarrow 0xFFFFFFFF$ $k = k + 4$ end $RT \leftarrow r$	Simple Fixed 1	Even	2

Gather bits from bytes	gbb rt, ra	$k = 0$ $s = 0$ for j = 7 to 128 by 8 $s_k \leftarrow RA_j$ $k = k + 1$ end $RT^{0:3} \leftarrow 0x0000 \parallel s$ $RT^{4:7} \leftarrow 0$ $RT^{8:11} \leftarrow 0$ $RT^{12:15} \leftarrow 0$	Permute	Odd	4
Gather bits from words	gbw rt, ra	$k = 0$ $s = 0x0$ for j = 31 to 128 by 32 $s_k \leftarrow RA_j$ $k \leftarrow k + 1$ end $RT^{0:3} \leftarrow 0x0000000 \parallel$ $RT^{4:7} \leftarrow 0$ $RT^{8:11} \leftarrow 0$ $RT^{12:15} \leftarrow 0$	Permute	Odd	4
Average Bytes	avgb rt, ra, rb	for j = 0 to 15 $RT_j \leftarrow ((0x00 \parallel RA_j) + (0x00 \parallel RB_j) + 1)_{7:14}$ end	Byte	Even	4

Absolute difference of bytes	absdb rt, ra, rb	for j = 0 to 15 if (RB ^j > ^u RA ^j) then RT ^j ← RB ^j - RA ^j else RT ^j ← RA ^j - RB ^j end	Byte	Even	4
Sum Bytes into Halfwords	sumb rt, ra, rb	$RT^{0:1} \leftarrow RB^0 + RB^1 + RB^2 + RB^3$, $RT^{2:3} \leftarrow RA^0 + RA^1 + RA^2 + RA^3$, $RT^{4:5} \leftarrow RB^4 + RB^5 + RB^6 + RB^7$, $RT^{6:7} \leftarrow RA^4 + RA^5 + RA^6 + RA^7$, $RT^{8:9} \leftarrow RB^8 + RB^9 + RB^{10} + RB^{11}$, $RT^{10:11} \leftarrow RA^8 + RA^9 + RA^{10} + RA^{11}$, $RT^{12:13} \leftarrow RB^{12} + RB^{13} + RB^{14} + RB^{15}$, $RT^{14:15} \leftarrow RA^{12} + RA^{13} + RA^{14} + RA^{15}$	Byte	Even	4
And	and rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} \& RB^{0:3}$, $RT^{4:7} \leftarrow RA^{4:7} \& RB^{4:7}$, $RT^{8:11} \leftarrow RA^{8:11} \& RB^{8:11}$, $RT^{12:15} \leftarrow RA^{12:15} \& RB^{12:15}$	Simple Fixed 1	Even	2
And byte immediate	andbi rt, ra, rb	$b \leftarrow I10 \& 0x00FF$, $bbbb \leftarrow b b b b$, $RT^{0:3} \leftarrow RA^{0:3} \& bbbb$, $RT^{4:7} \leftarrow RA^{4:7} \& bbbb$, $RT^{8:11} \leftarrow RA^{8:11} \& bbbb$, $RT^{12:15} \leftarrow RA^{12:15} \& bbbb$	Simple Fixed 1	Even	2
And Halfword Immediate		$t \leftarrow \text{RepLeftBit}(I10,16)$, $RT^{0:1} \leftarrow RA^{0:1} \& t$, $RT^{2:3} \leftarrow RA^{2:3} \& t$, $RT^{4:5} \leftarrow RA^{4:5} \& t$, $RT^{6:7} \leftarrow RA^{6:7} \& t$, $RT^{8:9} \leftarrow RA^{8:9} \& t$, $RT^{10:11} \leftarrow RA^{10:11} \& t$, $RT^{12:13} \leftarrow RA^{12:13} \& t$, $RT^{14:15} \leftarrow RA^{14:15} \& t$	Simple Fixed 1	Even	2
And word immediate	andi rt, ra, rb	$t \leftarrow \text{RepLeftBit}(I10,32)$, $RT^{0:3} \leftarrow RA^{0:3} \& t$, $RT^{4:7} \leftarrow RA^{4:7} \& t$, $RT^{8:11} \leftarrow RA^{8:11} \& t$, $RT^{12:15} \leftarrow RA^{12:15} \& t$	Simple Fixed 1	Even	2
Or	or rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} RB^{0:3}$, $RT^{4:7} \leftarrow RA^{4:7} RB^{4:7}$, $RT^{8:11} \leftarrow RA^{8:11} RB^{8:11}$, $RT^{12:15} \leftarrow RA^{12:15} RB^{12:15}$	Simple Fixed 1	Even	2
Or byte immediate	orbi rt, ra, value	$b \leftarrow I10 \& 0x00FF$, $bbbb \leftarrow b b b b$, $RT^{0:3} \leftarrow RA^{0:3} bbbb$, $RT^{4:7} \leftarrow RA^{4:7} bbbb$, $RT^{8:11} \leftarrow RA^{8:11} bbbb$, $RT^{12:15} \leftarrow RA^{12:15} bbbb$	Simple Fixed 1	Even	2
Or Halfword Immediate		$t \leftarrow \text{RepLeftBit}(I10,16)$, $RT^{0:1} \leftarrow RA^{0:1} t$, $RT^{2:3} \leftarrow RA^{2:3} t$, $RT^{4:5} \leftarrow RA^{4:5} t$, $RT^{6:7} \leftarrow RA^{6:7} t$, $RT^{8:9} \leftarrow RA^{8:9} t$, $RT^{10:11} \leftarrow RA^{10:11} t$, $RT^{12:13} \leftarrow RA^{12:13} t$, $RT^{14:15} \leftarrow RA^{14:15} t$	Simple Fixed 1	Even	2
Or word immediate	ori rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,32)$,	Simple Fixed 1	Even	2

		$RT^{0:3} \leftarrow RA^{0:3} \mid t, RT^{4:7} \leftarrow RA^{4:7} \mid t, RT^{8:11} \leftarrow RA^{8:11} \mid t, RT^{12:15} \leftarrow RA^{12:15} \mid t$			
Or across	orx rt, ra	$RT^{0:3} \leftarrow RA^{0:3} \mid RA^{4:7} \mid RA^{8:11} \mid RA^{12:15}$ $RT^{4:15} \leftarrow 0$	Simple Fixed 1	Even	2
Exclusive or	xor rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} \oplus RB^{0:3}, RT^{4:7} \leftarrow RA^{4:7} \oplus RB^{4:7}, RT^{8:11} \leftarrow RA^{8:11} \oplus RB^{8:11}, RT^{12:15} \leftarrow RA^{12:15} \oplus RB^{12:15}$	Simple Fixed 1	Even	2
Exclusive or byte immediate	xorbi rt, ra, value	$b \leftarrow I10 \& 0x00FF, bbbb \leftarrow b \mid \mid b \mid \mid b,$ $RT^{0:3} \leftarrow RA^{0:3} \oplus bbbb, RT^{4:7} \leftarrow RA^{4:7} \oplus bbbb, RT^{8:11} \leftarrow RA^{8:11} \oplus bbbb, RT^{12:15} \leftarrow RA^{12:15} \oplus bbbb$	Simple Fixed 1	Even	2
Exclusive Or Halfword Immediate	xorhi rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,16), RT^{0:1} \leftarrow RA^{0:1} \oplus t, RT^{2:3} \leftarrow RA^{2:3} \oplus t, RT^{4:5} \leftarrow RA^{4:5} \oplus t, RT^{6:7} \leftarrow RA^{6:7} \oplus t, RT^{8:9} \leftarrow RA^{8:9} \oplus t, RT^{10:11} \leftarrow RA^{10:11} \oplus t, RT^{12:13} \leftarrow RA^{12:13} \oplus t, RT^{14:15} \leftarrow RA^{14:15} \oplus t$	Simple Fixed 1	Even	2
Exclusive or word immediate	xori rt, ra, value	$t \leftarrow \text{RepLeftBit}(I10,32),$ $RT^{0:3} \leftarrow RA^{0:3} \oplus t, RT^{4:7} \leftarrow RA^{4:7} \oplus t, RT^{8:11} \leftarrow RA^{8:11} \oplus t, RT^{12:15} \leftarrow RA^{12:15} \oplus t$	Simple Fixed 1	Even	2
Shift Left Halfword	shlh rt, ra, rb	for j = 0 to 15 by 2 $s \leftarrow RB^{j:2} \& 0x001F$ $t \leftarrow RA^{j:2}$ for b = 0 to 15 if $b + s < 16$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow 0$ end $RT^{j:2} \leftarrow r$ End	Simple Fixed 2	Even	4

Shift Left Halfword Immediate	shlhi rt, ra, value	$s \leftarrow \text{RepLeftBit}(17,16) \& 0x001F$ for j = 0 to 15 by 2 $t \leftarrow RA^{j:2}$ for b = 0 to 15 if $b + s < 16$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow 0$ end $RT^{j:2} \leftarrow r$ End	Simple Fixed 2	Even	4
Shift left word	shl rt, ra, rb	for j = 0 to 15 by 4 $s \leftarrow RB^{j:4} \& 0x0000003F$ $t \leftarrow RA^{j:4}$ for b = 0 to 31 if $b + s < 32$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow 0$ end $RT^{j:4} \leftarrow r$ End	Simple Fixed 2	Even	4

Shift left word immediate	shli rt, ra, value	$s \leftarrow \text{RepLeftBit}(17,32) \ \& \ 0x0000003F$ for j = 0 to 15 by 4 $t \leftarrow RA^{j::4}$ for b = 0 to 31 if $b + s < 32$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow 0$ end $RT^{j::4} \leftarrow r$ End	Simple Fixed 2	Even	4
Shift left quadword by bytes	shlqby rt, ra, rb	$s \leftarrow RB_{27:31}$ for b = 0 to 15 if $b + s < 16$ then $r^b \leftarrow RA^{b+s}$ else $r^b \leftarrow 0$ end $RT \leftarrow r$	Permute	Odd	4
Shift left Quadword by bytes immediate	shlqbyi rt, ra, value	$s \leftarrow I7 \ \& \ 0x1F$ for b = 0 to 15 if $b + s < 16$ then $r^b \leftarrow RA^{b+s}$ else $r^b \leftarrow 0$ end $RT \leftarrow r$	Permute	Odd	4

Rotate Halfword	roth rt, ra, rb	for j = 0 to 15 by 2 $s \leftarrow RB^{j:2} \& 0x001F$ $t \leftarrow RA^{j:2}$ for b = 0 to 15 if b + s < 16 then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow t_{b+s-16}$ end $RT^{j:2} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate Halfword Immediate	rothi rt, ra, value	$s \leftarrow \text{RepLeftBit}(17,16) \& 0x000F$ for j = 0 to 15 by 2 $t \leftarrow RA^{j:2}$ for b = 0 to 15 if b + s < 16 then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow t_{b+s-16}$ end $RT^{j:2} \leftarrow r$ end	Simple Fixed 2	Even	4

Rotate Word	rot rt, ra, rb	for j = 0 to 15 by 4 $s \leftarrow \text{RB}^{j::4} \& 0x0000001F$ $t \leftarrow \text{RA}^{j::4}$ for b = 0 to 31 if $b + s < 32$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow t_{b+s-32}$ end $\text{RT}^{j::4} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate Word immediate	roti rt, ra, value	$s \leftarrow \text{RepLeftBit}(17,32) \& 0x0000001F$ for j = 0 to 15 by 4 $t \leftarrow \text{RA}^{j::4}$ for b = 0 to 31 if $b + s < 32$ then $r_b \leftarrow t_{b+s}$ else $r_b \leftarrow t_{b+s-32}$ end $\text{RT}^{j::4} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate quadwords by bytes	rotqby rt, ra, rb	$s \leftarrow \text{RB}^{28:31}$ for b = 0 to 15 if $b + s < 16$ then $r^b \leftarrow \text{RA}^{b+s}$ else $r^b \leftarrow \text{RA}^{b+s-16}$ end $\text{RT} \leftarrow r$	Permute	Odd	4

Rotate quadwords by bytes immediate	rotqybi rt, ra, value	$s \leftarrow I7_{14:17}$ for b = 0 to 15 if $b + s < 16$ then $r^b \leftarrow RA^{b+s}$ else $r^b \leftarrow RA^{b+s-16}$ end $RT \leftarrow r$	Permute	Odd	4
Rotate and mask halfword	rothm rt, ra, rb	for j = 0 to 15 by 2 $s \leftarrow (0 - RB_{j:2}) \& 0x001F$ $t \leftarrow RA_{j:2}$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0$ end $RT_{j:2} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate and mask halfword immediate		$s \leftarrow (0 - RB_{27:31}) \& 0x1F$ for j = 0 to 15 by 2 $t \leftarrow RA_{j:4}$ for b = 0 to 31 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0$ end $RT_{j:4} \leftarrow r$ end	Simple Fixed 2	Even	4

Rotate and mask word	rotm rt, ra, rb	for j = 0 to 15 by 4 $s \leftarrow (0 - RB_{j:4}) \& 0x0000003F$ $t \leftarrow RA_{j:4}$ for b = 0 to 31 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0$ end $RT_{j:4} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate and mask word immediate	rotmi rt, ra, value	$s \leftarrow (0 - \text{RepLeftBit}(17,32)) \& 0x0000001F$ for j = 0 to 15 by 2 $t \leftarrow RA_{j:2}$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0$ end $RT_{j:2} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate and mask Quadword by bytes	rotqmb by rt, ra, rb	$s \leftarrow (0 - RB_{27:31}) \& 0x1F$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0x00$ end $RT \leftarrow r$	Permute	Odd	4

Rotate and mask quadwords by bytes immediate	rotqmbyi rt, ra, value	$s \leftarrow (0 - 17) \& 0x1F$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow 0x00$ end $RT \leftarrow r$	Permute	Odd	4
Rotate and mask algebraic Halfword	rotmah rt, ra, rb	for j = 0 to 15 by 2 $s \leftarrow (0 - RB^{j:2}) \& 0x001F$ $t \leftarrow RA^{j:2}$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow t_0$ end $RT^{j:2} \leftarrow r$	Simple Fixed 2	Even	4
Rotate and mask algebraic Halfword immediate	rotmahi rt, ra, value	$s \leftarrow (0 - \text{RepLeftBit}(17,16)) \& 0x001F$ for j = 0 to 15 by 2 $t \leftarrow RA^{j:2}$ for b = 0 to 15 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow t_0$ end $RT^{j:2} \leftarrow r$	Simple Fixed 2	Even	4

Rotate and mask algebraic word	rotma rt, ra, rb	for j = 0 to 15 by 4 $s \leftarrow (0 - RB^{j:4}) \& 0x0000003F$ $t \leftarrow RA^{j:4}$ for b = 0 to 31 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow t_0$ end $RT^{j:4} \leftarrow r$ end	Simple Fixed 2	Even	4
Rotate and mask algebraic word immediate	rotmai rt, ra, value	$s \leftarrow (0 - \text{RepLeftBit}(17,32)) \& 0x0000003F$ for j = 0 to 15 by 4 $t \leftarrow RA^{j:4}$ for b = 0 to 31 if $b \geq s$ then $r_b \leftarrow t_{b-s}$ else $r_b \leftarrow t_0$ end $RT^{j:4} \leftarrow r$ end	Simple Fixed 2	Even	4
Compare equal byte	ceqb rt, ra, rb	for i = 0 to 15 If $RA^i = RB^i$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare equal byte immediate	ceqbi rt, ra, value	for i = 0 to 15 If $RA^i = I10_{2:9}$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2

Compare Equal Halfword	ceqh rt, ra, rb	for i = 0 to 15 by 2 If $RA^{i:2} = RB^{i:2}$ then $RT^{i:2} \leftarrow 0xF$ else $RT^{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Equal Halfword Immediate	ceghi rt, ra, value	for i = 0 to 15 by 2 If $RA^{i:2} = \text{RepLeftBit}(I10,16)$ then $RT^{i:2} \leftarrow 0xF$ else $RT^{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare equal word	ceq rt, ra, rb	for i = 0 to 15 by 4 If $RA^{i:4} = RB^{i:4}$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare equal word immediate	ceqi rt, ra, value	for i = 0 to 15 by 4 If $RA^{i:4} = \text{RepLeftBit}(I10,32)$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare greater than byte	cgtb rt, ra, rb	for i = 0 to 15 If $RA^i > RB^i$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare greater than byte immediate	cgtbi rt, ra, value	for i = 0 to 15 If $RA^i > I10_{2:9}$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2

Compare greater than Halfword	cgth rt, ra, rb	for i = 0 to 15 by 2 If $RA^{i:2} > RB^{i:2}$ then $RT^{i:2} \leftarrow 0xF$ else $RT^{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare greater than Halfword immediate	cgthi rt, ra, value	for i = 0 to 15 by 2 If $RA^{i:2} > \text{RepLeftBit}(I10,16)$ then $RT^{i:2} \leftarrow 0xF$ else $RT^{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare greater than word	cgt rt, ra, rb	for i = 0 to 15 by 4 If $RA^{i:4} > RB^{i:4}$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare greater than word immediate	cgti rt, ra, value	for i = 0 to 15 by 4 If $RA^{i:4} > \text{RepLeftBit}(I10,32)$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Logical greater than byte	clgtb rt, ra, rb	for i = 0 to 15 If $RA^i >^u RB^i$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Logical greater than byte immediate	clgti rt, ra, value	for i = 0 to 15 If $RA^i >^u I10_{2:9}$ then $RT^i \leftarrow 0xF$ else $RT^i \leftarrow 0x00$ end	Simple Fixed 1	Even	2

Compare Logical Greater than Halfword	clgth rt, ra, rb	for i = 0 to 15 by 2 If $RA_{i:2} >^u RB_{i:2}$ then $RT_{i:2} \leftarrow 0xF$ else $RT_{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Logical greater than Halfword immediate	clgthi rt, ra, value	for i = 0 to 15 by 2 If $RA_{i:2} >^u \text{RepLeftBit}(I10,16)$ then $RT_{i:2} \leftarrow 0xF$ else $RT_{i:2} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Logical Greater than word	clgt rt, ra, rb	for i = 0 to 15 by 4 If $RA_{i:4} >^u RB_{i:4}$ then $RT_{i:4} \leftarrow 0xF$ else $RT_{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Compare Logical greater than word immediate	clgti rt, ra, value	for i = 0 to 15 by 4 If $RA_{i:4} >^u \text{RepLeftBit}(I10,32)$ then $RT_{i:4} \leftarrow 0xF$ else $RT_{i:4} \leftarrow 0x00$ end	Simple Fixed 1	Even	2
Branch relative	br symbol	$PC \leftarrow (PC + \text{RepLeftBit}(I16 \parallel 0b00,32)) \& \text{LSLR}$	Branch	Odd	4
Branch relative and set link	brsl rt, symbol	$RT_{0:3} \leftarrow (PC + 4) \& \text{LSLR}$, $RT_{4:15} \leftarrow 0$, $PC \leftarrow (PC + \text{RepLeftBit}(I16 \parallel 0b00,32)) \& \text{LSLR}$	Branch	Odd	4
Branch if zero word	brz rt, symbol	If $RT_{0:3} = 0$ then $PC \leftarrow (PC + \text{RepLeftBit}(I16 \parallel 0b00)) \& \text{LSLR} \& 0xFFFFFFFFC$ else $PC \leftarrow (PC+4) \& \text{LSLR}$	Branch	Odd	4
Branch if not zero word	brnz rt, symbol	If $RT_{0:3} \neq 0$ then $PC \leftarrow (PC + \text{RepLeftBit}(I16 \parallel 0b00)) \& \text{LSLR} \& 0xFFFFFFFFC$ else $PC \leftarrow (PC+4) \& \text{LSLR}$	Branch	Odd	4

Floating add	fa rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} + RB^{0:3}, RT^{4:7} \leftarrow RA^{4:7} + RB^{4:7}, RT^{8:11} \leftarrow RA^{8:11} + RB^{8:11},$ $RT^{12:15} \leftarrow RA^{12:15} + RB^{12:15}$	Single Precision 1	Even	6
Floating subtract	fs rt, ra, rb	$RT^{0:3} \leftarrow RA^{0:3} - RB^{0:3}, RT^{4:7} \leftarrow RA^{4:7} - RB^{4:7}, RT^{8:11} \leftarrow RA^{8:11} - RB^{8:11},$ $RT^{12:15} \leftarrow RA^{12:15} - RB^{12:15}$	Single Precision 1	Even	6
Floating multiply	fm rt, ra, rb	for i = 0 to 15 by 4 $t \leftarrow RA^{i:4} * RB^{i:4}$ if $t > S_{max}$ then $RT^{i:4} \leftarrow S_{max}$ else if $t < 0$ then $RT^{i:4} \leftarrow 0$ else $RT^{i:4} \leftarrow t$ end	Single Precision 1	Even	6
Floating multiply and add	fma rt, ra, rb, rc	for i = 0 to 15 by 4 $s \leftarrow RA^{i:4} * RB^{i:4}$ $t \leftarrow s + RC^{i:4}$ if $t > S_{max}$ then $RT^{i:4} \leftarrow S_{max}$ else if $t < 0$ then $RT^{i:4} \leftarrow 0$ else $RT^{i:4} \leftarrow t$ end	Single Precision 1	Even	6
Floating multiply and subtract	fms rt, ra, rb, rc	for i = 0 to 15 by 4 $s \leftarrow RA^{i:4} * RB^{i:4}$ $t \leftarrow s - RC^{i:4}$ if $t > S_{max}$ then $RT^{i:4} \leftarrow S_{max}$ else if $t < 0$ then $RT^{i:4} \leftarrow 0$ else $RT^{i:4} \leftarrow t$ end	Single Precision 1	Even	6

Convert signed integer to floating point	csflt rt, ra, scale	$S \leftarrow 155 - I8$ for i = 0 to 15 by 4 $RT^{i:4} \leftarrow \text{shortreal}(\$signed(RA^{i:4})) / \text{shortreal}(2^S)$	Single Precision 1	Even	6
Convert floating point to signed integer	cflts rt, ra, scale	$S \leftarrow 155 - I8$ for i = 0 to 15 by 4 $RT^{i:4} \leftarrow \text{int}[\text{shortreal}(\$signed(RA^{i:4})) * \text{shortreal}(2^S)]$	Single Precision 1	Even	6
Convert unsigned integer to floating point	cufit rt, ra, scale	$S \leftarrow 155 - I8$ for i = 0 to 15 by 4 $RT^{i:4} \leftarrow \text{shortreal}(RA^{i:4}) / \text{shortreal}(2^S)$	Single Precision 1	Even	6
Convert floating point to unsigned integer	cfltu rt, ra, scale	$S \leftarrow 155 - I8$ for i = 0 to 15 by 4 $RT^{i:4} \leftarrow \text{int}[\text{shortreal}(RA^{i:4}) * \text{shortreal}(2^S)]$	Single Precision 1	Even	6
Floating compare equal	fceq rt, ra, rb	for i = 0 to 15 by 4 If $RA^{i:4} = RB^{i:4}$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Single Precision 1	Even	6
Floating compare magnitude equal	fcmeq rt, ra, rb	for i = 0 to 15 by 4 If $ RA^{i:4} = RB^{i:4} $ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Single Precision 1	Even	6
Floating compare greater than	fcgt rt, ra, rb	for i = 0 to 15 by 4 If $RA^{i:4} > RB^{i:4}$ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Single Precision 1	Even	6

Floating compare magnitude greater than	fcmgt rt, ra, rb	for i = 0 to 15 by 4 If $ RA^{i:4} > RB^{i:4} $ then $RT^{i:4} \leftarrow 0xF$ else $RT^{i:4} \leftarrow 0x00$ end	Single Precision 1	Even	6
Stop and signal	stop	Stop	Decode	Even/O dd	
No operation load	Inop	No operation		Odd	
No operation execute	nop	No operation		Even	

Design Implementation

Instruction Fetch stage

This stage consists of a basic 1-level cache that is initially empty. The cache is 8KB directly mapped, with 64 entries, each containing 32 words, a valid bit, and a 19-bit tag. Since 2 instructions are fetched at a time, normal operation (without any branches) will yield 16 cycles without a miss. A simple memory module has been used to simulate the hard disk memory, in which the cache must fetch 32 words on a miss. The penalty for a miss was set to a 5 cycle delay. Control logic was added to the Instruction Fetch stage so that when a miss occurs, or a branch has been miss-predicted, the program counter (PC) will stall and the stage will instead supply NOPs to the subsequent stages. Also a Branch Target Buffer (BTB) with 2-bit history was implemented to dynamically predict branches. The BTB is a small cache containing 8 entries, each with a 27-bit tag, 32-bit target address, and a 2-bit branch history. The PC is incremented by 8 (when not interrupted) so that 2 instructions have been accounted for (since 1 instruction is addressed every 4 bytes).

Decode

For simplicity, we implement a single cycle decode and issue stage in our project instead of the multiple stage decode and issue designed in the original cell SPU processor. This stage is responsible for checking all hazards in the incoming instructions, safely issuing instructions in a manner that they do not cause structural or data hazards down the pipeline. The instructions are read from the output of the Instruction Fetch stage and then decoded based on the opcode. The decode stage places a 3 bit value designating the unit ID to which the instructions will be issued. We assign IDs to each execution unit as shown in the table.

Unit Name	Unit ID
Simple Fixed 1	0
Simple Fixed 2	1
Single Precision	2
Byte	3
Permute	4
Local Store	5
Branch	6

The decode stage then evaluates the hazards in each of the instructions fetched and pushes the instruction after evaluating hazards. If any of the instructions cannot be passed in the current cycle, then a no-op instruction specific to the pipe is sent ahead and the instruction in the current cycle is looped back to be read again for decode in the next cycle. Additionally, in such a case the decode stage also stalls the Instruction Fetch stage so that the PC does not keep moving ahead. This state is maintained till both the instructions fetched have not been pushed by the decode stage. After that, the processor proceeds with the next pair of instructions to decode.

The decode stage gives a priority to the first instruction in a pair. If it causes a hazard, then none of the instructions are passed to the processing unit. If the first instruction is safe, then it will be pushed and then the hazard will be evaluated for the second instruction. The processor also stays in a stalled state unless both instructions are safely pushed to the processing unit. These two design choices guarantee the in order execution of the whole program.

Register File

There are 128 registers each with 4 words of data. Two instructions are evaluated at a time, each receiving upto 3 registers at a time (ra, rb, rc). Register data for the registers given by the Decode stage are fetched, also registers given by the execution stage for writeback are updated every cycle. The address' of chosen registers are concatenated with their respective register data. On the occurrence that the writeback address is the same as a read address for any of the registers given by the Decode stage, operation will resume but the conflicting register will instead read what is to be written from the writeback, instead of what is currently stored in the register file. The writeback accounts for forwarded data as well (coming from completed execution units).

Data Forwarding

The processing unit of our microprocessor is a 7 stage unit. Therefore, any instruction which is dependent on an earlier instruction will have to be either placed a minimum of 7 cycles down the line or it would have to wait till the earlier instruction completes execution. This strategy will induce stalls in the processor and decrease the speed of the processor. But when we look more closely at the execution units, we find out that many of the units in fact do not execute for all 7 stages. Therefore, we implement chains of forwarding registers all through the execution pipes to store the interim results of the units which have finished execution but are waiting to write back to the registers.

Whenever an instruction is about to enter the execution pipe, we check whether the operand required by that instruction is available in one of the forward registers. If so, then it means the current instruction actually fetched an outdated copy of the operand. In such a scenario, we replace the contents of the operands with the contents from the respective forward register instead of the value from the register file.

We append register write back address, rite enable and the cycle at which the data in the forwarding register is ready to be forwarded to each result. In the data forwarding unit, we check these bits to determine whether the data needs to be forwarded to the current instruction's operands.

Dual-Execution Pipeline

There are two execution pipelines; Even and Odd. Both have 7 pipeline stages, their own forwarding stages, and writeback stages. The 2 pipelines execute simultaneously and are able to accept one new instruction each cycle.

- **Even Pipe:** The Even Pipe contains 4 execution units, 5 forwarding stages, and 1 writeback stage. The 4 execution units are Single Precision, Simple Fixed 1, Simple Fixed 2, and Byte. The units execute in 6/7, 2, 3, and 3 cycles respectively. Since the pipe must only writeback after 7 cycles, data from finished units passes through forwarding stages until passed to the writeback stage. There are forwarding stages starting at the 3rd cycle of execution, ending at the 7th cycle of execution.
- **Odd Pipe:** The Odd Pipe contains 3 execution units, 6 forwarding stages, and 1 writeback stage. The 3 execution units are Branch, Local-store, and Permute. The units execute in 1, 6, and 3 cycles respectively. There are forwarding stages starting at the 2nd cycle of execution, ending at the 7th cycle of execution. Branch execution has been shortened to 1 cycle for this implementation, so that the penalty for a misprediction is less harsh (not having to flush operations that have entered the pipeline)

Testing

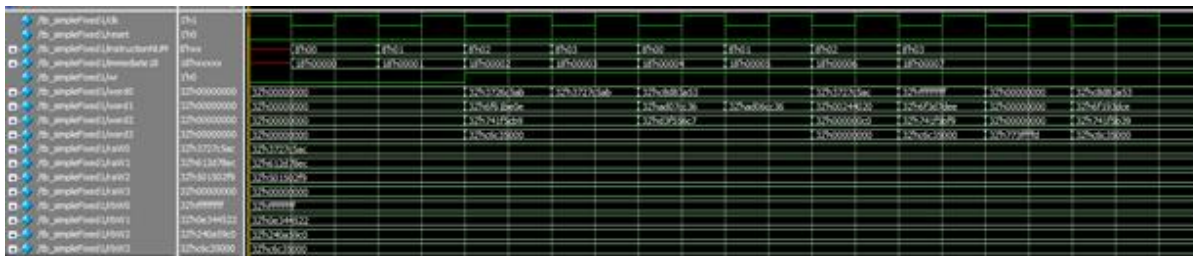
The following features and test conditions have been tested and will be demonstrated in the report:

- Execution units functionality
- Data Forwarding functionality
- Cache hit and miss penalty
- RAW hazard detection
- Simultaneous instruction pipeline detection (2 Even or 2 Odd instructions fetched)
- Simultaneous instruction dependency detection (RAW between 2 fetched instructions)
- Branch mispredictions and loops

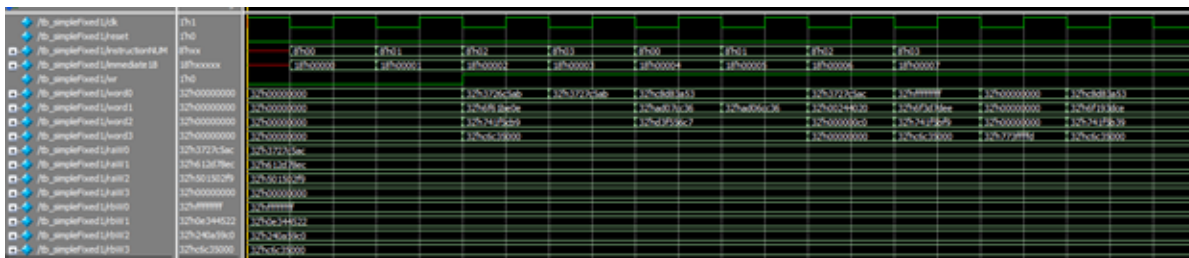
Execution Units Functionality

SimpleFixed1: For this unit, instructions Add Halfword, Add Word, Subtract from Halfword, Subtract form Word, AND, OR OR Across, and XOR were tested and saved. Refer to the tb_simpleFixed1.sv file for the sequence of instructions and module parameters.

The results of executing the testbench are outlined in the next page. The values of Ra and Rb are static for all instructions, and “word” and “hw” signals are the output for an instruction 2 cycles after being received.

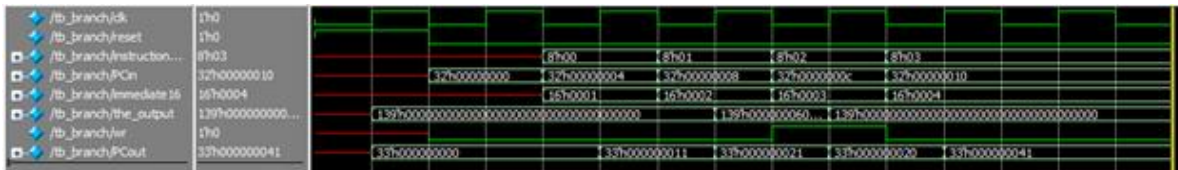


SimpleFixed2: For this unit, instructions Shift Left Halfword, Shift Left Word, Rotate Halfword, Rotate Word, Rotate and Mask Halfword, Rotate and Mask Word, Shift Left Halfword Immediate, and Shift Left Word Immediate. Refer to the tb_simpleFixed2.sv file for the sequence of instructions and module parameters.



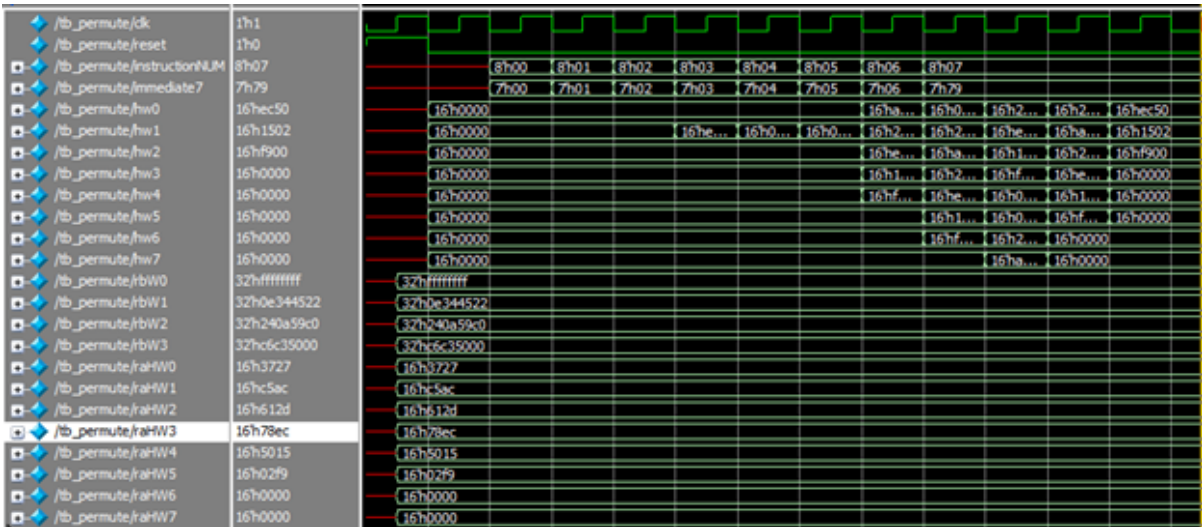
These are the results of executing the testbench. The values of Ra and Rb are static for all instructions, and “word” and “hw” signals are the output for an instruction 3 cycles after being received.

SinglePrecision: For this unit, instructions Multiply, Multiply Unsigned, Multiply High, Multiply Immediate, and Multiply Add were tested and saved. Refer to the tb_singlePrec.sv file for the sequence of instructions and module parameters.



These are the results of executing the testbench. The values of Ra, and Rb are static for all instructions, and “the_output” and “PCout” signals are the output for an instruction 1 cycles after being received. Note that the “wr” signal is only asserted for 1 cycle execution, which is when Branch Relative and Set Link is executed. The “wr” signal refers to the writeback status of an instruction. And only for this instruction (in the branch unit) a value is to be written back to the register file.

Permute: For this unit, instructions Gather Bits from Bytes, Gather Bits from Word, Shift Left Quadword by Bytes, Shift Left Quadword by Bytes Immediate, Rotate Quadword by Bytes, Rotate Quadword by Bytes Immediate, Rotate and Mask Quadword by Bytes, and Rotate and Mask Quadwords by Bytes Immediate were tested and saved. Refer to the tb_permute.sv file for the sequence of instructions and module parameters.



These are the results of executing the testbench. The values of Ra, and Rb are static for all instructions, and “word” and “hw” signals are the output for an instruction 3 cycles after being received.

Localstore: For this unit, instructions Load Quadword (A and D forms), Store Quadword (A and D forms), Load Quadword Instruction Relative (A form), and Store Quadword Instruction Relative (A form) were tested and saved. Refer to the tb_localstore.sv file for the sequence of instructions and module parameters.

In this test, 3 RAWs were implemented within a total of 16 instructions (8 cycles). The first RAW is not detected by the forwarding logic since the instruction that is to do the writeback has not finished execution yet (needs 3 cycles to execute). The second RAW is detected by the forwarding logic and the data from the finished instruction is used for the incoming instruction, instead of the value read from the register file. On the next immediate cycle, a RAW is detected again and the data is forwarded successfully again. This sequence of forwarded data is illustrated in the waveform below.

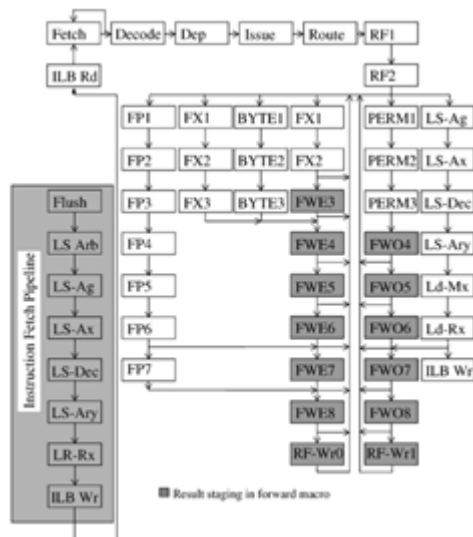


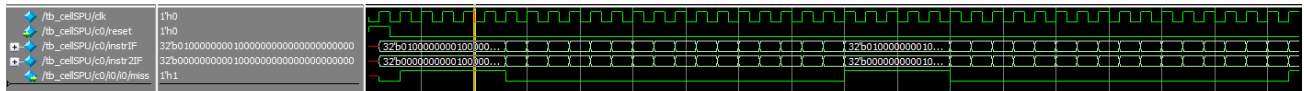
Figure 2

Here the “fw_out” signals refer to the data found in the different forwarding registers at each cycle, for both even and odd pipes, and the “data_out” signals refer to the data fetched for the incoming instruction for either Ra, Rb, or Rc for the even or odd pipes. The data inside the register file unit was made so that on reset, the value at each address location is the value of the address itself (ex: address 0 contains 128’d0, address 1 contains 128’d1, etc...). This was done so that tracking the data through the pipeline would be easier. The 2nd RAW is clearly detected (where the yellow line intersects) as “data_out_ra_1” does not take an expected low value from the register file, instead it gets a much higher value that is going to be written at the given address later in the pipeline. Also “data_out_rb_0” does not take on a higher value as expected (since the addresses values of each instruction are increasing every cycle), but instead takes in a low forwarded value such as 3. And then on the next cycle a RAW hazard is detected again for the same addresses as earlier. The same data forwarded to the previous 2 registers are now forwarded to ra_0 and ra_1.



Cache hit and miss penalty

Since the cache is empty at the start of a program, a cache miss penalty will be illustrated immediately. For this demonstration, the PC will be incrementing at a normal rate (no branches) until there is a miss, and more instructions need to be fetched.



Every cache index contains 32 words, or 32 instructions. The system fetches 2 instructions at a time, which suggests that the cache will hit 16 times without interruption under normal PC increments. Followed by a 5 cycle penalty when missing.

RAW Hazard Detections

Read after write (RAW) hazards are likely to occur, where instructions need data from registers whose data will change from other instructions executing in the pipeline. The Decode stage prevents these hazards by stalling the PC and pushing NOPs until the instruction that will update the suspect register has finished executing (and from there data forwarding takes care of the rest). The following waveform demonstrates a RAW between two ‘Even’ instructions in different pipes, a RAW between an ‘Even’ and an ‘Odd’ instruction at different pipes, and a dummy RAW that is not detected since an instruction has no dependency on the register that is conflicted (a RAW on an instruction’s value for rb register, although the instruction does not actually use rb).

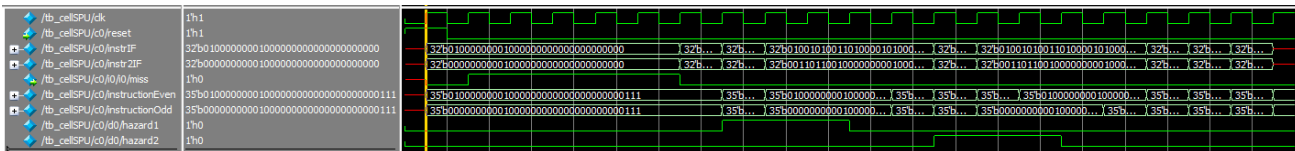


Figure 3

By analyzing the assembly code in Figure 4, it is apparent that a hazard will occur on instructions 3 and 8. This can be seen on the waveform in Figure 3, where at 2 separate instances the hazard signals are asserted, and NOPs are pushed from the Decoder to the execution unit ('InstructionEven' and 'InstructionOdd'). Notice that on the second hazard, the 'Even' instruction found during the hazard still gets pushed. This is because the 'Even' instruction was before the conflicting instruction ('Odd' instruction) in the instruction memory, and therefore is allowed to continue since it is still in order and has no hazard. Also, at instruction 14, the rt is register \$0. The instruction at 16 does not use an rb, but due to this, the value at the instruction's rb location is zero. This could of possibly lead to a hazard being falsely detected, but the hazard control logic in the Decode stage accounts for this. Which is why there are only 2 visible hazards being avoided in the waveform.

```
sumb $0, $32, $33
gbb $1, $32
avgb $2, $0, $33
gbb $3, $32
sumb $4, $32, $33
gbb $5, $32
avgb $6, $32, $33
gbb $7, $4
sumb $8, $32, $33
gbb $9, $32
avgb $10, $32, $33
gbb $11, $32
sumb $12, $32, $33
gbb $0, $32
avgb $14, $32, $33
gbb $15, $32
```

Figure 4

Simultaneous instruction pipeline/dependency detection

Another RAW hazard can occur between 2 simultaneous instructions. Specifically when the latter of the 2 fetched instructions is reading from the rt address of the former instruction. Also the system accounts for structural hazards when 2 instructions of the same type ('Even' or 'Odd') have been fetched. Under this condition, each of the 2 instructions must execute in order rather than simultaneously while the vacant pipeline is pushed with NOPs. These 2 hazard detections can be seen from the waveform at Figure 5.

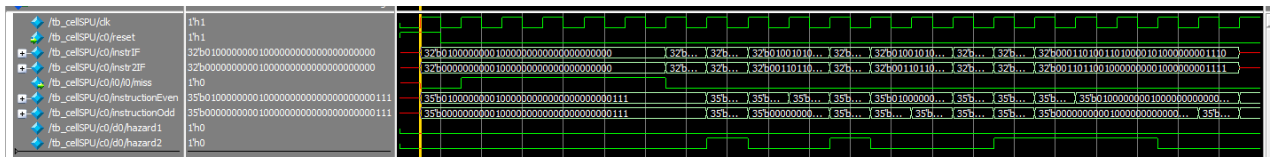


Figure 5

```

sumb $0, $32, $33
gbb $1, $32
avgb $2, $32, $33
sumb $3, $32, $33
sumb $4, $32, $33
gbb $5, $32
gbb $6, $32
gbb $7, $32
sumb $8, $32, $33
gbb $9, $32
avgb $10, $32, $33
gbb $11, $32
sumb $12, $32, $33
gbb $13, $12
avgb $14, $32, $33
gbb $15, $32

```

Figure 6

As seen in Figure 6, there are 2 fetched 'Even' instructions at lines 3 and 4 (2nd cycle), and 2 'Odd' instructions fetched from lines 7 and 8 (4th cycle). And then at lines 13 and 14 (7th cycle), the former 'Odd' instruction has a read dependency on the rt of the prior (but simultaneously fetched) 'Even' instruction. These three hazards are detected and dealt with as seen by the NOPs being pushed in 'InstructionEven' and 'InstructionOdd', and the hazard signals being asserted in Figure 5.

Branch mispredictions and loops

Two branch instructions will be used to demonstrate the effectiveness of the branch predictions, and how the system responds to a miss prediction; Branch relative and branch if not zero word. For branch relative, the program will execute 4 instructions, hit the branch instruction, make a misprediction of not taken, then eventually loop back to the branch target address (address 0), and loop infinitely while predicting taken. This description is based on the program in Figure 9.

As seen in the waveform in Figure 7, the rt1 and r2 values that are fetched every cycle, eventually pass the branch instruction. This is because the first prediction of any new branch is always predicted as not taken. But this is processed

to be a misprediction (as seen by PCBranch[65] signal), and is corrected. The PC jumps to the target address (0 or the beginning). The subsequent branches are predicted taken as seen by the taken signal, and the rt values keep repeating. Also it is clear in this waveform that the instructions that were in the pipeline during the misprediction were not executed, since the registers 4, 6, 7, and 8 never changed their values (hence were never executed).

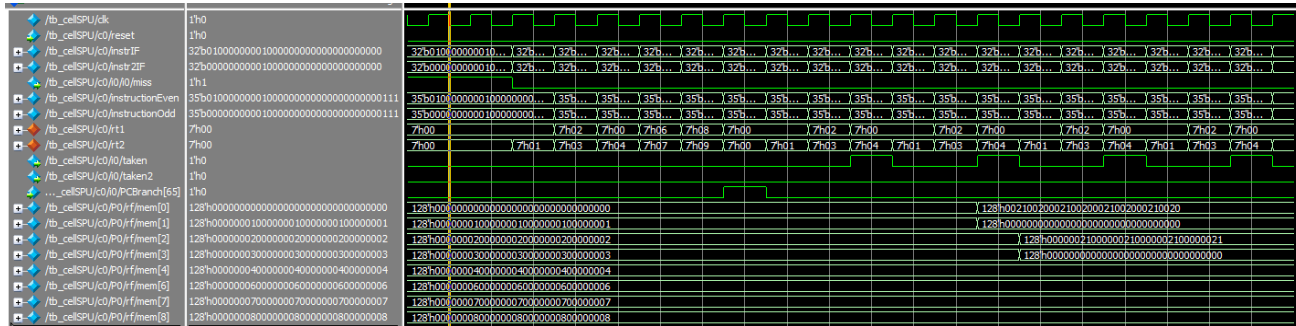


Figure 7

For branch if not zero word, a simple loop will be formed, using values initialized in the register file (to make this demonstration simpler). Every register in the register file will have a value of its address (r0=0, r1=1, etc..). A subtract from word (sf) instruction will be used every time the branch loops, and once the result of sf is zero, the branch will no longer loop. Figure 10 has this implemented in assembly.

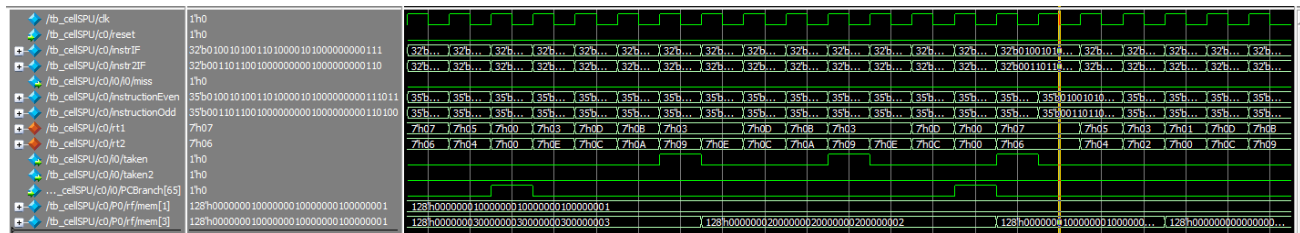


Figure 8

```

sf $3, $1, $3
gbb $14, $32
avgb $13, $32, $33
gbb $12, $32
sumb $11, $32, $33
gbb $10, $32
brnz $3, -6
avgb $9, $32, $33
sumb $7, $32, $33
gbb $6, $32
avgb $5, $32, $33
gbb $4, $32
sumb $3, $32, $33
gbb $2, $32
avgb $1, $32, $33
gbb $0, $32
avgb $13, $32, $33
gbb $12, $32
sumb $11, $32, $33
avgb $9, $32, $33
sumb $7, $32, $33
gbb $6, $32
avgb $5, $32, $33
gbb $4, $32
sumb $3, $32, $33
gbb $2, $32
avgb $1, $32, $33
gbb $0, $32

```

Figure 10

```

sumb $0, $32, $33
gbb $1, $32
avgb $2, $32, $33
gbb $3, $32
br -4
sumb $4, $32, $33
avgb $6, $32, $33
gbb $7, $32
sumb $8, $32, $33
gbb $9, $32
avgb $10, $32, $33
gbb $11, $32
sumb $12, $32, $33
gbb $13, $32
avgb $14, $32, $33
gbb $15, $32

```

Figure 9

In the waveform at Figure 8, it is clear that register 3 is not zero, and the branch is taken. The branch was initially mispredicted on the first iteration, as seen by the PCBranch[65] bit. The next 3 cycles the branch is predicted taken. Register 3 is being subtracted from register 1 and then rewritten to register 3. This is essentially a loop equivalent to `for(i=3;i>0;i--)`. On the third iteration, the branch is predicted taken, but the branch will no longer branch since register 3 is equal to 0. The misprediction flag at PCBranch[65] is set again, and the program continues after the branch, after the misprediction penalty.

Matrix multiplication Example

Using the instructions at our disposal, we have successfully implemented a 4x4 matrix multiplier. For this demonstration, fixed point arithmetic was used to make the accuracy of the results clear. The commands issues are below:

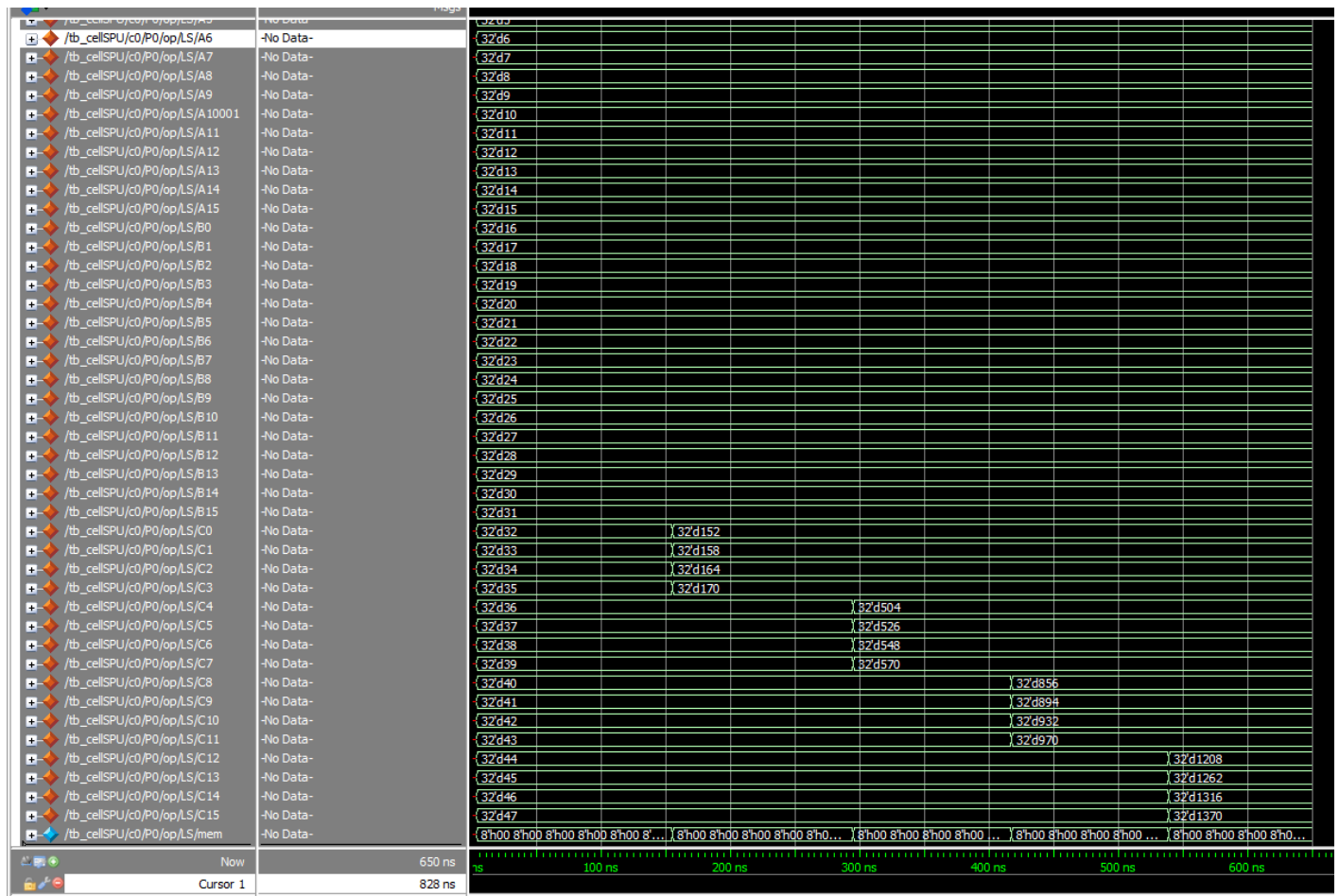

```

ila $0, 0
lqa $3, 16
ila $2, 128
lqa $4, 20
il $8, 8
lqa $5, 24
il $1, 4
lqa $6, 28
fsm $9, $8
lnop
lqd $7, 0($0)
ai $1, $1, -1
shlqbyi $10, $7, 4
ai $0, $0, 16
shlqbyi $11, $7, 8
and $13, $7, $9
shlqbyi $12, $7, 12
and $14, $10, $9
rotqmbiyi $17, $13, -4
and $15, $11, $9
rotqmbiyi $18, $13, -8
and $16, $12, $9
rotqmbiyi $19, $13, -12
or $13, $13, $17
rotqmbiyi $20, $14, -4
or $13, $13, $18
rotqmbiyi $21, $14, -8
or $13, $13, $19
rotqmbiyi $22, $14, -12
or $14, $14, $20
rotqmbiyi $23, $15, -4
or $14, $14, $21
rotqmbiyi $24, $15, -8
or $14, $14, $22
rotqmbiyi $25, $15, -12
or $15, $15, $23
rotqmbiyi $26, $16, -4
or $15, $15, $24
rotqmbiyi $27, $16, -8
or $15, $15, $25
rotqmbiyi $28, $16, -12
or $16, $16, $26
il $29, 0
or $16, $16, $27
or $16, $16, $28
mpya $29, $3, $13, $29
mpya $29, $4, $14, $29
mpya $29, $5, $15, $29
mpya $29, $6, $16, $29
stqd $29, 0($2)
ai $2, $2, 16
brnz $1, -41
stop

```

Figure 11

Simple values were used in this demonstration. Values ranging from 0 to 32 were placed in memory (where matrices A and B are located) during reset. From there the program stores the result in matrix C, which will also be stored in memory. The waveform at Figure 12 demonstrates correct 4x4 matrix multiplication.



Appendix: Code

Top Module

```
module cellSPU(clk, reset);
    input          clk, reset;

    logic          flush, enable;
    logic [0:31]    instrIF, instr2IF;
    logic          stall, miss, predictIF2, predictIF, predictOutD;
    logic [0:31]    PCin, PCoutIF, PCoutD, predictPCoutD,
                    predictedPCIF2, predictedPCIF;
    logic [0:65]    PCoutBranch;
    logic [0:34]    instructionEven, instructionOdd;

    IF i0(.clk(clk), .reset(reset), .instr(instrIF), .instr2(instr2IF), .stall(stall),
        .predictedPC(predictedPCIF), .fb_PC(PCoutBranch[33:64]), .fb_taken(PCoutBranch[32]),
        .fb_predictedPC(PCoutBranch[0:31]), .fb_en(enable), .pc_reg(PCoutIF),
        .taken(predictIF), .taken2(predictIF2), .predictedPC2(predictedPCIF2) ,
        .fb_mispredict(PCoutBranch[65]));

    DecodeUnit d0(.clk(clk), .reset(reset), .predictIn0(predictIF), .predictIn1(predictIF2),
        .PCin(PCoutIF), .predictPCin0(predictedPCIF), .predictPCin1(predictedPCIF2),
        .IR({instrIF, instr2IF}), .PCSelect(stall), .predictOut(predictOutD),
        .instructionOdd(instructionOdd), .instructionEven(instructionEven),
        .PCOut(PCoutD), .predictPCOut(predictPCoutD), .flush(flush), .taken(PCoutBranch[65]));

    ProcessingUnit P0(
        .instructionOdd (instructionOdd),
        .instructionEven (instructionEven),
        .PCout (PCoutBranch),
        .PCin (PCoutD),
        .predictPCin(predictPCoutD),
        .predictIn(predictOutD),
        .reset (reset),
        .clk (clk),
        .flush(flush));

    assign enable = (PCoutBranch != 0);

endmodule
```

Instruction Fetch Stage

```
module IF(clk, reset, stall, predictedPC, fb_PC, fb_predictedPC, fb_en, taken, fb_taken, fb_mispredict,
predictedPC2, taken2,
        instr, instr2, pc_reg);
    input                clk, reset, fb_taken, fb_en, fb_mispredict, stall;
    input [0:31]         fb_PC, fb_predictedPC;
    output logic         taken, taken2;
    output logic [0:31]  predictedPC, predictedPC2, instr, instr2, pc_reg;

    logic                miss, stall_reg;
    logic [0:31]         pc2, pc, im_rd_addr, instr1_out, instr2_out;
    integer i;

    BTB b0(.clk(clk), .reset(reset), .currentPC(im_rd_addr), .currentPC2(im_rd_addr + 4), .fb_PC(fb_PC),
.fb_taken(fb_taken),
        .fb_en(fb_en), .fb_predictedPC(fb_predictedPC), .taken(taken), .taken2(taken2),
.predictedPC(predictedPC),
        .predictedPC2(predictedPC2));

    icache i0(.clk(clk), .reset(reset), .pc(im_rd_addr), .instr(instr1_out), .instr2(instr2_out), .miss(miss));

    always_ff @(posedge clk) begin
        if(reset) begin
            pc<=0;
            pc_reg <= 0;
            stall_reg <= 0;
        end else begin
            pc<=im_rd_addr+8;
            pc_reg <= im_rd_addr;
            stall_reg <= stall;
        end
    end

    end

    always_comb begin
        //taken=0; taken2=0; predictedPC=0; predictedPC2=0;

        pc2=pc+4;
        im_rd_addr = (fb_mispredict) ? fb_predictedPC : (stall_reg | miss) ? pc_reg : (taken) ?
predictedPC : (taken2) ? predictedPC2 : pc;
        instr = fb_mispredict | miss ? 32'b0100_0000_0010_0000_0000_0000_0000_0000 :
instr1_out;
```

```
instr2 = fb_mispredict | miss ? 32'b0000_0000_0010_0000_0000_0000_0000_0000 :  
instr2_out;  
end  
  
endmodule
```

Instruction Cache

```
module icache(clk, reset, pc, instr, instr2, miss);
    input                clk, reset;
    input [0:31]         pc;//first 2 bits aren't used, next 3 bits for word offset, next 6 for cache index, last 21
    for tag
    output logic         miss;
    output logic [0:31]  instr, instr2;

    logic [0:63][0:1043] cache;//1 valid bit, 19 bit tag, 32 32bit words
    logic [0:31]         missedPC;
    logic                validPCs;
    logic [0:31][0:31]   instr32;
    integer i, j;

    hardDisk h0(.clk(clk), .reset(reset), .missedPC(missedPC),
        .miss(miss), .instr32(instr32), .valid(validPCs));

    always_ff @(posedge clk) begin
        if(reset) begin
            instr<=32'b0100_0000_0010_0000_0000_0000_0000_0000;
            instr2<=32'b0000_0000_0010_0000_0000_0000_0000_0000;
            miss<=0;
        end else begin
            //instruction 1
            if(cache[pc[19:24]][0]==1) begin//check for valid bit
                if(cache[pc[19:24]][1:19]==pc[0:18]) begin//check for tag
                    instr<=cache[pc[19:24]][20+(pc[25:29] * 32) +:32];
                    instr2<=cache[pc[19:24]][20+((pc[25:29] + 1) * 32) +:32];
                    miss <= 0;
                    missedPC <= 0;
                end else begin
                    instr<=32'b0100_0000_0010_0000_0000_0000_0000_0000;
                    instr2<=32'b0000_0000_0010_0000_0000_0000_0000_0000;
                    miss <= 1;
                    missedPC <= pc;
                end
            end else begin
                instr<=32'b0100_0000_0010_0000_0000_0000_0000_0000;
                instr2<=32'b0000_0000_0010_0000_0000_0000_0000_0000;
                miss <= 1;
                missedPC <= pc;
            end
        end
    end
end
```

```

        if(validPCs) begin
            cache[pc[19:24]][0]<=1;
            cache[pc[19:24]][1:19]<=pc[0:18];
            for(i=0; i<32; i++)
                cache[pc[19:24]][20+(32*i) +:32]<=instr32[i];//writing 8 instructions to
cache
        end
    end
end
endmodule

```

Branch Target Buffer

```
module BTB(clk, reset, currentPC, predictedPC, fb_PC, fb_predictedPC, fb_en, taken, fb_taken,
currentPC2, predictedPC2, taken2);
    input                clk, reset, fb_taken, fb_en;
    input [0:31]         currentPC, currentPC2, fb_PC, fb_predictedPC;
    output logic         taken, taken2;
    output logic [0:31]  predictedPC, predictedPC2;

    logic [0:7][0:60]    bufferTable;//no valid bit, may be redundant, have to settle for misprediction entering
a for-loop                                                    //but will still be fine when returning to previous

    for-loops
    logic [0:1]          takenState;
    integer i;

    always_ff @(posedge clk) begin
        if(reset) begin
            taken <= 0;
            predictedPC <= 0;
            taken2 <= 0;
            predictedPC2 <= 0;
            for(i=0; i<8; i++) begin
                bufferTable[i][0:58]=0;
                bufferTable[i][59]=0;//bit 0 of takenState
                bufferTable[i][60]=0;//bit 1 of takenState
            end
        end else begin
            //feed-back from branch pipe
            if(fb_en) begin
                bufferTable[fb_PC[0:2]][0:26]<=fb_PC[3:29];
                bufferTable[fb_PC[0:2]][27:58]<=fb_predictedPC;
                if(bufferTable[fb_PC[0:2]][0:26]==fb_PC[3:29])
                    bufferTable[fb_PC[0:2]][59:60]<=takenState;
                else begin
                    if(fb_taken)
                        bufferTable[fb_PC[0:2]][59:60]<=2'b11;//pretend as if we
assumed taken previously, which is the desired behavior of 2-bit history
                    else
                        bufferTable[fb_PC[0:2]][59:60]<=2'b10;//same assumption
                end
            end
        end

        //PC1
    end
```

```

        if(bufferTable[currentPC[0:2]][0:26]==currentPC[3:29]) begin
            if(fb_en && bufferTable[currentPC[0:2]][0:26]==fb_PC[3:29])
                predictedPC<=fb_predictedPC;
            else
                predictedPC<=bufferTable[currentPC[0:2]][27:58];
            taken<=bufferTable[currentPC[0:2]][59]; //left bit (of the 2) pretty much defines
prediction
        end else
            taken<=0;

        //PC2
        if(bufferTable[currentPC2[0:2]][0:26]==currentPC2[3:29]) begin
            if(fb_en && bufferTable[currentPC2[0:2]][0:26]==fb_PC[3:29])
                predictedPC2<=fb_predictedPC;
            else
                predictedPC2<=bufferTable[currentPC2[0:2]][27:58];
            taken2<=bufferTable[currentPC2[0:2]][59]; //left bit (of the 2) pretty much
defines prediction
        end else
            taken2<=0;

    end
end

always_comb begin
    //prediction state transition
    case(bufferTable[fb_PC[0:2]][59:60])
        2'b00: begin
            if(fb_taken)
                takenState=2'b01;
            else
                takenState=2'b00;
        end
        2'b01: begin
            if(fb_taken)
                takenState=2'b11;
            else
                takenState=2'b00;
        end
        2'b11: begin
            if(fb_taken)
                takenState=2'b11;
            else
                takenState=2'b10;
        end
    end
end

```



```
        end
        2'b10: begin
            if(fb_taken)
                takenState=2'b11;
            else
                takenState=2'b00;
            end
        default: takenState=2'b10;
    endcase
end
endmodule
```

Main Memory For Instructions

```
module hardDisk(clk, reset, missedPC, instr32, miss, valid);
    input                                clk, reset, miss;
    input [0:31]                        missedPC;//first 2 bits aren't used
    output logic                        valid;
    output logic [0:31][0:31]          instr32;//8 word outputs

    reg [0:31] mem [0:127];//128 instructions, 32 bit wide
    logic [0:2]                        penalty;
    integer i;
    initial $readmemb("C:/Users/Celia/Documents/ese545/instructions.txt", mem);//initializes instructions with
values from a txt file

    always_ff @(posedge clk) begin
        if(reset) begin
            penalty<=0;
        end else begin
            if(penalty<4) begin//4 cycle delay
                if(miss || penalty!=0)
                    penalty<=penalty+1;
                else
                    penalty<=0;
            end else
                penalty<=0;
        end
    end

    always_comb begin
        if(penalty==3) begin//4 cycle delay
            for(i=0; i<32; i++)
                instr32[i]=mem[missedPC[0:29]+i];
            valid=1;
        end else begin
            valid=0;
        end
    end
endmodule
```

Decode Stage

/* Decode module for decoding instructions, detecting hazards and issuing instructions in order after resolving all hazards */

```
module DecodeUnit(
    input                                clk, reset, predictIn0, predictIn1, taken,
    input [0:31]                        PCin, predictPCin0, predictPCin1,
    input [0:63]                        IR,
    output logic [0:31]                 PCOut, predictPCOut,
    output logic                        PCSelect, predictOut, flush,
    output logic [0:34]                 instructionEven, instructionOdd);

    logic [0:31] nextInstr1, nextInstr2, nextInstr1Reg, nextInstr2Reg, oldPC, PCRead,
oldPredictPC_0, predictPCRead_0, oldPredictPC_1, predictPCRead_1;
    logic [0:31] oldPC_reg, oldPredictPC_0_reg, oldPredictPC_1_reg;
    logic [0:34] instr1, instr2, instrEvenReg, instrOddReg;
    logic [0:34] ex1_even, ex2_even, ex3_even, ex4_even, ex5_even, ex6_even, ex7_even,
wb_even;
    logic [0:34] ex1_odd, ex2_odd, ex3_odd, ex4_odd, ex5_odd, ex6_odd, ex7_odd, wb_odd;
    logic [0:6]  addr_ra, addr_rb, addr_rc;
    logic                firstSrc, secondSrc, thirdSrc, hazard1, hazard2, PCSelectReg, firstBranch,
predictRead_0, oldPredict_0, predictRead_1, oldPredict_1;
    logic                oldPredict_0_reg, oldPredict_1_reg;

    always_ff @(posedge clk) begin
        if(reset | taken) begin
            instructionEven <= 35'b0100_0000_0010_0000_0000_0000_0000_111;
            instructionOdd <= 35'b0000_0000_0010_0000_0000_0000_0000_111;
        end
        else begin
            /* Outputs containing instructions to be passed to the RF module for operand fetch.
            Also functions as a register which remembers the instructions currently going
            through RF stage.*/
            instructionEven <= instrEvenReg;
            instructionOdd <= instrOddReg;
        end

        // Registers to remember all instructions currently executing in the even pipe
        ex1_even <= instructionEven;
        ex2_even <= ex1_even;
        ex3_even <= ex2_even;
        ex4_even <= ex3_even;
        ex5_even <= ex4_even;
        ex6_even <= ex5_even;
    end
end
```

```

ex7_even <= ex6_even;
wb_even <= ex7_even;

// Registers to remember all instructions currently executing in the odd pipe
ex1_odd <= instructionOdd;
ex2_odd <= ex1_odd;
ex3_odd <= ex2_odd;
ex4_odd <= ex3_odd;
ex5_odd <= ex4_odd;
ex6_odd <= ex5_odd;
ex7_odd <= ex6_odd;
wb_odd <= ex7_odd;

// PC and prediction forwarded
if(firstBranch) begin
    PCOut <= PCRead;
    predictOut <= predictRead_0;
    predictPCOut <= predictPCRead_0;
    flush <= 1;
end
else begin
    PCOut <= PCRead + 4;
    predictOut <= predictRead_1;
    predictPCOut <= predictPCRead_1;
    flush <= 0;
end

// Register to store next instructions in case all instructions in the current set were not pushed
nextInstr1Reg <= nextInstr1;
nextInstr2Reg <= nextInstr2;
oldPC_reg <= oldPC;
oldPredict_0_reg <= oldPredict_0;
oldPredict_1_reg <= oldPredict_1;
oldPredictPC_0_reg <= oldPredictPC_0;
oldPredictPC_1_reg <= oldPredictPC_1;

//PC Select signal forwarded
PCSelectReg <= PCSelect;
end

always_comb begin

    // Read two instructions from the IR each cycle
    instr1[0:31] = (PCSelectReg) ? nextInstr1Reg : IR[0:31];

```

```

instr2[0:31] = (PCSelectReg) ? nextInstr2Reg : IR[32:63];
PCRead = (PCSelectReg) ? oldPC_reg : PCin;
predictRead_0 = (PCSelectReg) ? oldPredict_0_reg : predictIn0;
predictRead_1 = (PCSelectReg) ? oldPredict_1_reg : predictIn1;
predictPCRead_0 = (PCSelectReg) ? oldPredictPC_0_reg : predictPCin0;
predictPCRead_1 = (PCSelectReg) ? oldPredictPC_1_reg : predictPCin1;

// Decode instructions starting from the first of two instructions fetched
// Default value of '111' for unit ID for no-op instructions
instr1[32:34] = 3'b111;
instr2[32:34] = 3'b111;
instrEvenReg = 35'b0100_0000_0010_0000_0000_0000_0000_111;
instrOddReg = 35'b0000_0000_0010_0000_0000_0000_0000_111;
// Set Hazard flag to zero before evaluating hazards
PCSelect = 0;
hazard1 = 0;
hazard2 = 0;
firstBranch = 0;
// If data not ready for this instruction, need to set this to 1 and stall the pipeline

// Start decoding first instruction only if it is not a no-op
if(instr1[0:10] != 11'b0100_0000_001 & instr1[0:10] != 11'b0000_0000_001 & instr1[0:10] !=
11'b0000_0000_000) begin
    firstSrc = 0;
    secondSrc = 0;
    thirdSrc = 0;
    // instr1 going to simple fixed 1 unit
    if(instr1[0:10] == 11'b00011001000 | instr1[0:10] == 11'b00011000000 | instr1[0:10]
== 11'b00001001000 | instr1[0:10] == 11'b00001000000
    | instr1[0:10] == 11'b01010100101 | instr1[0:10] == 11'b00110110110 | instr1[0:10]
== 11'b00110110101 | instr1[0:10] == 11'b00110110100
    | instr1[0:10] == 11'b00011000001 | instr1[0:10] == 11'b00001000001 | instr1[0:10]
== 11'b00111110000 | instr1[0:10] == 11'b01001000001
    | instr1[0:10] == 11'b01111010000 | instr1[0:10] == 11'b01111001000 | instr1[0:10]
== 11'b01111000000 | instr1[0:10] == 11'b01001010000
    | instr1[0:10] == 11'b01001001000 | instr1[0:10] == 11'b01001000000 | instr1[0:10]
== 11'b01011010000 | instr1[0:10] == 11'b01011001000
    | instr1[0:10] == 11'b01011000000 | instr1[0:8] == 9'b010000011 | instr1[0:8] ==
9'b010000010 | instr1[0:8] == 9'b010000001
    | instr1[0:8] == 9'b011000001 | instr1[0:8] == 9'b001100101 | instr1[0:7] ==
8'b00011101 | instr1[0:7] == 8'b00011100
    | instr1[0:7] == 8'b00001101 | instr1[0:7] == 8'b00001100 | instr1[0:7] ==
8'b00010110 | instr1[0:7] == 8'b00010101

```

```

        | instr1[0:7] == 8'b00010100 | instr1[0:7] == 8'b00000110 | instr1[0:7] ==
8'b00000101 | instr1[0:7] == 8'b00000100
        | instr1[0:7] == 8'b01000110 | instr1[0:7] == 8'b01000101 | instr1[0:7] ==
8'b01000100 | instr1[0:7] == 8'b01111110
        | instr1[0:7] == 8'b01111101 | instr1[0:7] == 8'b01111100 | instr1[0:7] ==
8'b01001110 | instr1[0:7] == 8'b01001101
        | instr1[0:7] == 8'b01001100 | instr1[0:7] == 8'b01011110 | instr1[0:7] ==
8'b01011101 | instr1[0:7] == 8'b01011100
        | instr1[0:6] == 7'b0100001) begin
            instr1[32:34] = 3'b000;
            addr_ra = instr1[18:24];
            addr_rb = instr1[11:17];
            if(instr1[0:8] != 9'b010000011 & instr1[0:8] != 9'b010000010 & instr1[0:8] !=
9'b010000001 & instr1[0:8] != 9'b011000001
                & instr1[0:8] != 9'b001100101 & instr1[0:6] != 7'b0100001) firstSrc = 1;
            if(instr1[0:10] == 11'b00011001000 | instr1[0:10] == 11'b00011000000 |
instr1[0:10] == 11'b00001001000 | instr1[0:10] == 11'b00001000000
                | instr1[0:10] == 11'b00011000001 | instr1[0:10] == 11'b00001000001 |
instr1[0:10] == 11'b01001000001 | instr1[0:10] == 11'b01111010000
                | instr1[0:10] == 11'b01111001000 | instr1[0:10] == 11'b01111000000 |
instr1[0:10] == 11'b01001010000 | instr1[0:10] == 11'b01001001000
                | instr1[0:10] == 11'b01001000000 | instr1[0:10] == 11'b01011010000 |
instr1[0:10] == 11'b01011001000 | instr1[0:10] == 11'b01011000000)
                secondSrc = 1;
        end
        // instr1 going to simple fixed 2 unit
        else if(instr1[0:10] == 11'b00001011111 | instr1[0:10] == 11'b00001011011 |
instr1[0:10] == 11'b00001011100 | instr1[0:10] == 11'b00001011000
            | instr1[0:10] == 11'b00001011101 | instr1[0:10] == 11'b00001011001 |
instr1[0:10] == 11'b00001011110 | instr1[0:10] == 11'b00001011010
            | instr1[0:10] == 11'b00001111111 | instr1[0:10] == 11'b00001111011 |
instr1[0:10] == 11'b00001111100 | instr1[0:10] == 11'b00001111000
            | instr1[0:10] == 11'b00001111101 | instr1[0:10] == 11'b00001111001 |
instr1[0:10] == 11'b00001111110 | instr1[0:10] == 11'b00001111010) begin
            instr1[32:34] = 3'b001;
            addr_ra = instr1[18:24];
            addr_rb = instr1[11:17];
            firstSrc = 1;
            if(instr1[0:10] == 11'b00001011111 | instr1[0:10] == 11'b00001011011 |
instr1[0:10] == 11'b00001011100 | instr1[0:10] == 11'b00001011000
                | instr1[0:10] == 11'b00001011101 | instr1[0:10] == 11'b00001011001 |
instr1[0:10] == 11'b00001011110 | instr1[0:10] == 11'b00001011010)
                secondSrc = 1;
        end
end

```

```

        // instr1 going to single precision unit
        else if(instr1[0:10] == 11'b01111000100 | instr1[0:10] == 11'b01111001100 |
instr1[0:10] == 11'b01111000101 | instr1[0:10] == 11'b01011000100
        | instr1[0:10] == 11'b01011000101 | instr1[0:10] == 11'b01011000110 |
instr1[0:10] == 11'b01111000010 | instr1[0:10] == 11'b01111001010
        | instr1[0:10] == 11'b01011000010 | instr1[0:10] == 11'b01011001010 |
instr1[0:9] == 10'b0111011010 | instr1[0:9] == 10'b0111011000
        | instr1[0:9] == 10'b0111011011 | instr1[0:9] == 10'b0111011001 | instr1[0:7]
== 8'b01110100 | instr1[0:7] == 8'b01110101
        | instr1[0:3] == 4'b1100 | instr1[0:3] == 4'b1110 | instr1[0:3] == 4'b1111) begin
instr1[32:34] = 3'b010;
addr_ra = instr1[18:24];
addr_rb = instr1[11:17];
addr_rc = instr1[25:31];
firstSrc = 1;
        if(instr1[0:10] == 11'b01111000100 | instr1[0:10] == 11'b01111001100 |
instr1[0:10] == 11'b01111000101 | instr1[0:10] == 11'b01011000100
        | instr1[0:10] == 11'b01011000101 | instr1[0:10] == 11'b01011000110 |
instr1[0:10] == 11'b01111000010 | instr1[0:10] == 11'b01111001010
        | instr1[0:10] == 11'b01011000010 | instr1[0:10] == 11'b01011001010 |
instr1[0:3] == 4'b1100 | instr1[0:3] == 4'b1110 | instr1[0:3] == 4'b1111)
        secondSrc = 1;
        if(instr1[0:3] == 4'b1100 | instr1[0:3] == 4'b1110 | instr1[0:3] == 4'b1111)
thirdSrc = 1;
        end
        // instr1 going to byte unit
        else if(instr1[0:10] == 11'b01010110100 | instr1[0:10] == 11'b00011010011 |
instr1[0:10] == 11'b00001010011 | instr1[0:10] == 11'b01001010011) begin
instr1[32:34] = 3'b011;
addr_ra = instr1[18:24];
addr_rb = instr1[11:17];
firstSrc = 1;
        if(instr1[0:10] == 11'b00011010011 | instr1[0:10] == 11'b00001010011 |
instr1[0:10] == 11'b01001010011) secondSrc = 1;
        end
        // instr1 going to permute unit
        else if(instr1[0:10] == 11'b00110110010 | instr1[0:10] == 11'b00110110000 |
instr1[0:10] == 11'b00111011111 | instr1[0:10] == 11'b00111111111
        | instr1[0:10] == 11'b00111011100 | instr1[0:10] == 11'b00111111100 | instr1[0:10]
== 11'b00111011101 | instr1[0:10] == 11'b00111111101) begin
instr1[32:34] = 3'b100;
addr_ra = instr1[18:24];
addr_rb = instr1[11:17];
firstSrc = 1;

```

```

        if(instr1[0:10] == 11'b00111011111 | instr1[0:10] == 11'b00111011100 |
instr1[0:10] == 11'b00111011101) secondSrc = 1;
        end
        // instr1 going to local store unit
        else if(instr1[0:8] == 9'b001100001 | instr1[0:8] == 9'b001100111 | instr1[0:8] ==
9'b001000001 | instr1[0:8] == 9'b001000111
            | instr1[0:7] == 8'b00110100 | instr1[0:7] == 8'b00100100) begin
            instr1[32:34] = 3'b101;
            addr_ra = instr1[18:24];
            addr_rb = instr1[11:17];
            addr_rc = instr1[25:31];
            if(instr1[0:7] == 8'b00110100 | instr1[0:7] == 8'b00100100) firstSrc = 1;
            if(instr1[0:8] == 9'b001000001 | instr1[0:8] == 9'b001000111 | instr1[0:7] ==
8'b00100100) thirdSrc = 1;
            end
            // instr1 going to branch unit
            else if(instr1[0:8] == 9'b001100100 | instr1[0:8] == 9'b001100110 | instr1[0:8] ==
9'b001000000 | instr1[0:8] == 9'b001000010) begin
            instr1[32:34] = 3'b110;
            addr_rc = instr1[25:31];
            if(instr1[0:8] == 9'b001000000 | instr1[0:8] == 9'b001000010) thirdSrc = 1;
            end

        // Ra Rb or Rc slot dependency on a previous instruction

        // Instruction in Execution Stage 5 should not be a No-op. Instructions from Odd Pipe
is always ready after 6 stages.
        if(ex5_even[0:10] != 11'b0100_0000_001) begin
            if(ex5_even[32:34] == 3'b010) begin // Dependency with integer mul and madd
instructions
                if(ex5_even[0:10] == 11'b01111000100 | ex5_even[0:10] ==
11'b01111001100 | ex5_even[0:10] == 11'b01111000101 | ex5_even[0:7] == 8'b01110100
                | ex5_even[0:7] == 8'b01110101 | ex5_even[0:3] == 4'b1100) begin
                    if(ex5_even[0:3] != 4'b1100) begin
                        if((firstSrc & ex5_even[25:31] == addr_ra) | (secondSrc
& ex5_even[25:31] == addr_rb) | (thirdSrc & ex5_even[25:31] == addr_rc))
                            hazard1 = 1;
                        end
                    else begin
                        if((firstSrc & ex5_even[4:10] == addr_ra) | (secondSrc &
ex5_even[4:10] == addr_rb) | (thirdSrc & ex5_even[4:10] == addr_rc))
                            hazard1 = 1;
                        end
                    end
                end
            end
        end
    end
end

```



```

end
end // Only need to consider the Even Pipe since only FP instructions can go upto 7
stages.

```

```

// Instruction in Execution Stage 4 should not be a No-op. Need to consider both
Pipes for instructions which take 6 stages or more.
if(ex4_even[0:10] != 11'b0100_0000_001) begin
    if(ex4_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
        if(ex4_even[0:3] != 4'b1100 & ex4_even[0:3] != 4'b1110 &
ex4_even[0:3] != 4'b1111) begin
            if((firstSrc & ex4_even[25:31] == addr_ra) | (secondSrc &
ex4_even[25:31] == addr_rb) | (thirdSrc & ex4_even[25:31] == addr_rc))
                hazard1 = 1;
            end
        else begin
            if((firstSrc & ex4_even[4:10] == addr_ra) | (secondSrc &
ex4_even[4:10] == addr_rb) | (thirdSrc & ex4_even[4:10] == addr_rc))
                hazard1 = 1;
            end
        end
    end
end
if(ex4_odd[0:10] != 11'b0000_0000_001) begin
    if(ex4_odd[32:34] == 3'b101) begin // Dependency with load instructions
        if(ex4_odd[0:8] == 9'b001100001 | ex4_odd[0:8] == 9'b001100111 |
ex4_odd[0:7] == 8'b00110100) begin
            if((firstSrc & ex4_odd[25:31] == addr_ra) | (secondSrc &
ex4_odd[25:31] == addr_rb) | (thirdSrc & ex4_odd[25:31] == addr_rc))
                hazard1 = 1;
            end
        end
    end
end
end
end

```

```

// Instruction in Execution Stage 3 should not be a No-op. Need to consider both
Pipes for instructions which take 5 stages or more.
if(ex3_even[0:10] != 11'b0100_0000_001) begin
    if(ex3_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
        if(ex3_even[0:3] != 4'b1100 & ex3_even[0:3] != 4'b1110 &
ex3_even[0:3] != 4'b1111) begin
            if((firstSrc & ex3_even[25:31] == addr_ra) | (secondSrc &
ex3_even[25:31] == addr_rb) | (thirdSrc & ex3_even[25:31] == addr_rc))
                hazard1 = 1;
            end
        end
    end
end
end

```

```

        else begin
            if((firstSrc & ex3_even[4:10] == addr_ra) | (secondSrc &
ex3_even[4:10] == addr_rb) | (thirdSrc & ex3_even[4:10] == addr_rc))
                hazard1 = 1;
            end
        end
    end
    if(ex3_odd[0:10] != 11'b0000_0000_001) begin
        if(ex3_odd[32:34] == 3'b101) begin // Dependency with load instructions
            if(ex3_odd[0:8] == 9'b001100001 | ex3_odd[0:8] == 9'b001100111 |
ex3_odd[0:7] == 8'b00110100) begin
                if((firstSrc & ex3_odd[25:31] == addr_ra) | (secondSrc &
ex3_odd[25:31] == addr_rb) | (thirdSrc & ex3_odd[25:31] == addr_rc))
                    hazard1 = 1;
                end
            end
        end
    end

    // Instruction in Execution Stage 2 should not be a No-op. Need to consider both
    Pipes for instructions which take 4 stages or more.
    if(ex2_even[0:10] != 11'b0100_0000_001) begin
        if(ex2_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
            if(ex2_even[0:3] != 4'b1100 & ex2_even[0:3] != 4'b1110 &
ex2_even[0:3] != 4'b1111) begin
                if((firstSrc & ex2_even[25:31] == addr_ra) | (secondSrc &
ex2_even[25:31] == addr_rb) | (thirdSrc & ex2_even[25:31] == addr_rc))
                    hazard1 = 1;
                end
            end
        else begin
            if((firstSrc & ex2_even[4:10] == addr_ra) | (secondSrc &
ex2_even[4:10] == addr_rb) | (thirdSrc & ex2_even[4:10] == addr_rc))
                hazard1 = 1;
            end
        end
    end
    if(ex2_even[32:34] == 3'b001 | ex2_even[32:34] == 3'b011) begin //
Dependency with Simple Fixed 2 unit or byte unit instructions
        if((firstSrc & ex2_even[25:31] == addr_ra) | (secondSrc &
ex2_even[25:31] == addr_rb) | (thirdSrc & ex2_even[25:31] == addr_rc))
            hazard1 = 1;
        end
    end
    if(ex2_odd[0:10] != 11'b0000_0000_001) begin
        if(ex2_odd[32:34] == 3'b101) begin // Dependency with load instructions

```

```

        if(ex2_odd[0:8] == 9'b001100001 | ex2_odd[0:8] == 9'b001100111 |
ex2_odd[0:7] == 8'b00110100) begin
            if((firstSrc & ex2_odd[25:31] == addr_ra) | (secondSrc &
ex2_odd[25:31] == addr_rb) | (thirdSrc & ex2_odd[25:31] == addr_rc))
                hazard1 = 1;
            end
        end
    end
    if(ex2_odd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
        if((firstSrc & ex2_odd[25:31] == addr_ra) | (secondSrc &
ex2_odd[25:31] == addr_rb) | (thirdSrc & ex2_odd[25:31] == addr_rc))
            hazard1 = 1;
        end
    end
end

// Instruction in Execution Stage 1 should not be a No-op. Need to consider both
Pipes for instructions which take 3 stages or more.
    if(ex1_even[0:10] != 11'b0100_0000_001) begin
        if(ex1_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
            if(ex1_even[0:3] != 4'b1100 & ex1_even[0:3] != 4'b1110 &
ex1_even[0:3] != 4'b1111) begin
                if((firstSrc & ex1_even[25:31] == addr_ra) | (secondSrc &
ex1_even[25:31] == addr_rb) | (thirdSrc & ex1_even[25:31] == addr_rc))
                    hazard1 = 1;
                end
            end
        else begin
            if((firstSrc & ex1_even[4:10] == addr_ra) | (secondSrc &
ex1_even[4:10] == addr_rb) | (thirdSrc & ex1_even[4:10] == addr_rc))
                hazard1 = 1;
            end
        end
    end
    if(ex1_even[32:34] == 3'b001 | ex1_even[32:34] == 3'b011) begin //
Dependency with Simple Fixed 2 unit or byte unit instructions
        if((firstSrc & ex1_even[25:31] == addr_ra) | (secondSrc &
ex1_even[25:31] == addr_rb) | (thirdSrc & ex1_even[25:31] == addr_rc))
            hazard1 = 1;
        end
    end
end
    if(ex1_odd[0:10] != 11'b0000_0000_001) begin
        if(ex1_odd[32:34] == 3'b101) begin
            if(ex1_odd[0:8] == 9'b001100001 | ex1_odd[0:8] == 9'b001100111 |
ex1_odd[0:7] == 8'b00110100) begin

```

```

                                if((firstSrc & ex1_odd[25:31] == addr_ra) | (secondSrc &
ex1_odd[25:31] == addr_rb) | (thirdSrc & ex1_odd[25:31] == addr_rc))
                                    hazard1 = 1;
                                end
                            end
                            if(ex1_odd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
                                if((firstSrc & ex1_odd[25:31] == addr_ra) | (secondSrc &
ex1_odd[25:31] == addr_rb) | (thirdSrc & ex1_odd[25:31] == addr_rc))
                                    hazard1 = 1;
                                end
                            end

// Instruction in RF Stage should not be a No-op. Need to consider both Pipes for
instructions which take 2 stages or more.
                            if(instructionEven[0:10] != 11'b0100_0000_001) begin
                                if(instructionEven[32:34] == 3'b010) begin // Dependency with all single
precision unit instructions
                                    if(instructionEven[0:3] != 4'b1100 & instructionEven[0:3] != 4'b1110 &
instructionEven[0:3] != 4'b1111) begin
                                        if((firstSrc & instructionEven[25:31] == addr_ra) | (secondSrc &
instructionEven[25:31] == addr_rb) | (thirdSrc & instructionEven[25:31] == addr_rc))
                                            hazard1 = 1;
                                        end
                                    else begin
                                        if((firstSrc & instructionEven[4:10] == addr_ra) | (secondSrc &
instructionEven[4:10] == addr_rb) | (thirdSrc & instructionEven[4:10] == addr_rc))
                                            hazard1 = 1;
                                        end
                                    end
                                end
                                if(instructionEven[32:34] == 3'b001 | instructionEven[32:34] == 3'b011 |
instructionEven[32:34] == 3'b000) begin
                                    // Dependency with Simple Fixed 2 unit, byte unit or Simple Fixed 1
instructions
                                        if((firstSrc & instructionEven[25:31] == addr_ra) | (secondSrc &
instructionEven[25:31] == addr_rb) | (thirdSrc & instructionEven[25:31] == addr_rc))
                                            hazard1 = 1;
                                        end
                                    end
                                end
                                if(instructionOdd[0:10] != 11'b0000_0000_001) begin
                                    if(instructionOdd[32:34] == 3'b101) begin // Dependency with local store
instructions
                                        if(instructionOdd[0:8] == 9'b001100001 | instructionOdd[0:8] ==
9'b001100111 | instructionOdd[0:7] == 8'b00110100) begin

```

```

                                if((firstSrc & instructionOdd[25:31] == addr_ra) | (secondSrc &
instructionOdd[25:31] == addr_rb) | (thirdSrc & instructionOdd[25:31] == addr_rc))
                                    hazard1 = 1;
                                end
                            end
                        end
                    if(instructionOdd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
                                if((firstSrc & instructionOdd[25:31] == addr_ra) | (secondSrc &
instructionOdd[25:31] == addr_rb) | (thirdSrc & instructionOdd[25:31] == addr_rc))
                                    hazard1 = 1;
                                end
                            end
                        end

                    end

                    // Start decoding second instruction only if it is not a no-op
                    if(instr2[0:10] != 11'b0100_0000_001 & instr2[0:10] != 11'b0000_0000_001 & instr2[0:10] !=
11'b0000_0000_000) begin
                        firstSrc = 0;
                        secondSrc = 0;
                        thirdSrc = 0;
                        // instr2 going to simple fixed 1 unit
                        if(instr2[0:10] == 11'b00011001000 | instr2[0:10] == 11'b00011000000 | instr2[0:10]
== 11'b00001001000 | instr2[0:10] == 11'b00001000000
                            | instr2[0:10] == 11'b01010100101 | instr2[0:10] == 11'b00110110110 | instr2[0:10]
== 11'b00110110101 | instr2[0:10] == 11'b00110110100
                            | instr2[0:10] == 11'b00011000001 | instr2[0:10] == 11'b00001000001 | instr2[0:10]
== 11'b00111110000 | instr2[0:10] == 11'b01001000001
                            | instr2[0:10] == 11'b01111010000 | instr2[0:10] == 11'b01111001000 | instr2[0:10]
== 11'b01111000000 | instr2[0:10] == 11'b01001001000
                            | instr2[0:10] == 11'b01001001000 | instr2[0:10] == 11'b01001000000 | instr2[0:10]
== 11'b01011010000 | instr2[0:10] == 11'b01011001000
                            | instr2[0:10] == 11'b01011000000 | instr2[0:8] == 9'b010000011 | instr2[0:8] ==
9'b010000010 | instr2[0:8] == 9'b010000001
                            | instr2[0:8] == 9'b011000001 | instr2[0:8] == 9'b001100101 | instr2[0:7] ==
8'b00011101 | instr2[0:7] == 8'b00011100
                            | instr2[0:7] == 8'b00001101 | instr2[0:7] == 8'b00001100 | instr2[0:7] ==
8'b00010110 | instr2[0:7] == 8'b00010101
                            | instr2[0:7] == 8'b00010100 | instr2[0:7] == 8'b00000110 | instr2[0:7] ==
8'b00000101 | instr2[0:7] == 8'b00000100
                            | instr2[0:7] == 8'b01000110 | instr2[0:7] == 8'b01000101 | instr2[0:7] ==
8'b01000100 | instr2[0:7] == 8'b01111110
                            | instr2[0:7] == 8'b01111101 | instr2[0:7] == 8'b01111100 | instr2[0:7] ==
8'b01001110 | instr2[0:7] == 8'b01001101

```

```

| instr2[0:7] == 8'b01001100 | instr2[0:7] == 8'b01011110 | instr2[0:7] ==
8'b01011101 | instr2[0:7] == 8'b01011100
| instr2[0:6] == 7'b0100001) begin
    instr2[32:34] = 3'b000;
    addr_ra = instr2[18:24];
    addr_rb = instr2[11:17];
    firstSrc = 1;
    if(instr2[0:8] != 9'b010000011 & instr2[0:8] != 9'b010000010 & instr2[0:8] !=
9'b010000001 & instr2[0:8] != 9'b011000001
    & instr2[0:8] != 9'b001100101 & instr2[0:6] != 7'b0100001) firstSrc = 1;
    if(instr2[0:10] == 11'b00011001000 | instr2[0:10] == 11'b00011000000 |
instr2[0:10] == 11'b00001001000 | instr2[0:10] == 11'b00001000000
    | instr2[0:10] == 11'b00011000001 | instr2[0:10] == 11'b00001000001 |
instr2[0:10] == 11'b01001000001 | instr2[0:10] == 11'b01111010000
    | instr2[0:10] == 11'b01111001000 | instr2[0:10] == 11'b01111000000 |
instr2[0:10] == 11'b01001010000 | instr2[0:10] == 11'b01001001000
    | instr2[0:10] == 11'b01001000000 | instr2[0:10] == 11'b01011010000 |
instr2[0:10] == 11'b01011001000 | instr2[0:10] == 11'b01011000000)
        secondSrc = 1;
    end
    // instr2 going to simple fixed 2 unit
    else if(instr2[0:10] == 11'b00001011111 | instr2[0:10] == 11'b00001011011 |
instr2[0:10] == 11'b00001011100 | instr2[0:10] == 11'b00001011000
    | instr2[0:10] == 11'b00001011101 | instr2[0:10] == 11'b00001011001 |
instr2[0:10] == 11'b00001011110 | instr2[0:10] == 11'b00001011010
    | instr2[0:10] == 11'b00001111111 | instr2[0:10] == 11'b00001111011 |
instr2[0:10] == 11'b00001111100 | instr2[0:10] == 11'b00001111000
    | instr2[0:10] == 11'b00001111101 | instr2[0:10] == 11'b00001111001 |
instr2[0:10] == 11'b00001111110 | instr2[0:10] == 11'b00001111010) begin
        instr2[32:34] = 3'b001;
        addr_ra = instr2[18:24];
        addr_rb = instr2[11:17];
        firstSrc = 1;
        if(instr2[0:10] == 11'b00001011111 | instr2[0:10] == 11'b00001011011 |
instr2[0:10] == 11'b00001011100 | instr2[0:10] == 11'b00001011000
    | instr2[0:10] == 11'b00001011101 | instr2[0:10] == 11'b00001011001 |
instr2[0:10] == 11'b00001011110 | instr2[0:10] == 11'b00001011010)
            secondSrc = 1;
        end
        // instr2 going to single precision unit
        else if(instr2[0:10] == 11'b01111000100 | instr2[0:10] == 11'b01111001100 |
instr2[0:10] == 11'b01111000101 | instr2[0:10] == 11'b01011000100
    | instr2[0:10] == 11'b01011000101 | instr2[0:10] == 11'b01011000110 |
instr2[0:10] == 11'b01111000010 | instr2[0:10] == 11'b01111001010

```

```

| instr2[0:10] == 11'b01011000010 | instr2[0:10] == 11'b01011001010 |
instr2[0:9] == 10'b0111011010 | instr2[0:9] == 10'b0111011000
| instr2[0:9] == 10'b0111011011 | instr2[0:9] == 10'b0111011001 | instr2[0:7]
== 8'b01110100 | instr2[0:7] == 8'b01110101
| instr2[0:3] == 4'b1100 | instr2[0:3] == 4'b1110 | instr2[0:3] == 4'b1111) begin
instr2[32:34] = 3'b010;
addr_ra = instr2[18:24];
addr_rb = instr2[11:17];
addr_rc = instr2[25:31];
firstSrc = 1;
if(instr2[0:10] == 11'b01111000100 | instr2[0:10] == 11'b01111001100 |
instr2[0:10] == 11'b01111000101 | instr2[0:10] == 11'b01011000100
| instr2[0:10] == 11'b01011000101 | instr2[0:10] == 11'b01011000110 |
instr2[0:10] == 11'b01111000010 | instr2[0:10] == 11'b01111001010
| instr2[0:10] == 11'b01011000010 | instr2[0:10] == 11'b01011001010 |
instr2[0:3] == 4'b1100 | instr2[0:3] == 4'b1110 | instr2[0:3] == 4'b1111)
secondSrc = 1;
if(instr2[0:3] == 4'b1100 | instr2[0:3] == 4'b1110 | instr2[0:3] == 4'b1111)
thirdSrc = 1;
end
// instr2 going to byte unit
else if(instr2[0:10] == 11'b01010110100 | instr2[0:10] == 11'b00011010011 |
instr2[0:10] == 11'b00001010011 | instr2[0:10] == 11'b01001010011) begin
instr2[32:34] = 3'b011;
addr_ra = instr2[18:24];
addr_rb = instr2[11:17];
firstSrc = 1;
if(instr2[0:10] == 11'b00011010011 | instr2[0:10] == 11'b00001010011 |
instr2[0:10] == 11'b01001010011) secondSrc = 1;
end
// instr2 going to permute unit
else if(instr2[0:10] == 11'b00110110010 | instr2[0:10] == 11'b00110110000 |
instr2[0:10] == 11'b00111011111 | instr2[0:10] == 11'b00111111111
| instr2[0:10] == 11'b00111011100 | instr2[0:10] == 11'b00111111100 | instr2[0:10]
== 11'b00111011101 | instr2[0:10] == 11'b00111111101) begin
instr2[32:34] = 3'b100;
addr_ra = instr2[18:24];
addr_rb = instr2[11:17];
firstSrc = 1;
if(instr1[0:10] == 11'b00111011111 | instr1[0:10] == 11'b00111011100 |
instr1[0:10] == 11'b00111011101) secondSrc = 1;
end
// instr2 going to local store unit

```

```

        else if(instr2[0:8] == 9'b001100001 | instr2[0:8] == 9'b001100111 | instr2[0:8] ==
9'b001000001 | instr2[0:8] == 9'b001000111
            | instr2[0:7] == 8'b00110100 | instr2[0:7] == 8'b00100100) begin
            instr2[32:34] = 3'b101;
            addr_ra = instr2[18:24];
            addr_rb = instr2[11:17];
            if(instr2[0:7] == 8'b00110100 | instr2[0:7] == 8'b00100100) firstSrc = 1;
            if(instr2[0:8] == 9'b001000001 | instr2[0:8] == 9'b001000111 | instr2[0:7] ==
8'b00100100) thirdSrc = 1;
            end
            // instr2 going to branch unit
        else if(instr2[0:8] == 9'b001100100 | instr2[0:8] == 9'b001100110 | instr2[0:8] ==
9'b001000000 | instr2[0:8] == 9'b001000010) begin
            instr2[32:34] = 3'b110;
            addr_rc = instr2[25:31];
            if(instr2[0:8] == 9'b001000000 | instr2[0:8] == 9'b001000010) thirdSrc = 1;
            end

        // Ra Rb or Rc slot dependency on a previous instruction

        // Instruction in Execution Stage 5 should not be a No-op. Instructions from Odd Pipe
is always ready after 6 stages.
        if(ex5_even[0:10] != 11'b0100_0000_001) begin
            if(ex5_even[32:34] == 3'b010) begin // Dependency with integer mul and madd
instructions
                if(ex5_even[0:10] == 11'b01111000100 | ex5_even[0:10] ==
11'b01111001100 | ex5_even[0:10] == 11'b01111000101 | ex5_even[0:7] == 8'b01110100
                | ex5_even[0:7] == 8'b01110101 | ex5_even[0:3] == 4'b1100) begin
                    if(ex5_even[0:3] != 4'b1100) begin
                        if((firstSrc & ex5_even[25:31] == addr_ra) | (secondSrc &
& ex5_even[25:31] == addr_rb) | (thirdSrc & ex5_even[25:31] == addr_rc))
                            hazard2 = 1;
                        end
                    end
                else begin
                    if((firstSrc & ex5_even[4:10] == addr_ra) | (secondSrc &
ex5_even[4:10] == addr_rb) | (thirdSrc & ex5_even[4:10] == addr_rc))
                        hazard2 = 1;
                    end
                end
            end
        end
    end
end // Only need to consider the Even Pipe since only FP instructions can go upto 7
stages.

```



```

// Instruction in Execution Stage 4 should not be a No-op. Need to consider both
Pipes for instructions which take 6 stages or more.
    if(ex4_even[0:10] != 11'b0100_0000_001) begin
        if(ex4_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
            if(ex4_even[0:3] != 4'b1100 & ex4_even[0:3] != 4'b1110 &
ex4_even[0:3] != 4'b1111) begin
                if((firstSrc & ex4_even[25:31] == addr_ra) | (secondSrc &
ex4_even[25:31] == addr_rb) | (thirdSrc & ex4_even[25:31] == addr_rc))
                    hazard2 = 1;
                end
            else begin
                if((firstSrc & ex4_even[4:10] == addr_ra) | (secondSrc &
ex4_even[4:10] == addr_rb) | (thirdSrc & ex4_even[4:10] == addr_rc))
                    hazard2 = 1;
                end
            end
        end
    end
    if(ex4_odd[0:10] != 11'b0000_0000_001) begin
        if(ex4_odd[32:34] == 3'b101) begin // Dependency with load instructions
            if(ex4_odd[0:8] == 9'b001100001 | ex4_odd[0:8] == 9'b001100111 |
ex4_odd[0:7] == 8'b00110100) begin
                if((firstSrc & ex4_odd[25:31] == addr_ra) | (secondSrc &
ex4_odd[25:31] == addr_rb) | (thirdSrc & ex4_odd[25:31] == addr_rc))
                    hazard2 = 1;
                end
            end
        end
    end
end

// Instruction in Execution Stage 3 should not be a No-op. Need to consider both
Pipes for instructions which take 5 stages or more.
    if(ex3_even[0:10] != 11'b0100_0000_001) begin
        if(ex3_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
            if(ex3_even[0:3] != 4'b1100 & ex3_even[0:3] != 4'b1110 &
ex3_even[0:3] != 4'b1111) begin
                if((firstSrc & ex3_even[25:31] == addr_ra) | (secondSrc &
ex3_even[25:31] == addr_rb) | (thirdSrc & ex3_even[25:31] == addr_rc))
                    hazard2 = 1;
                end
            else begin
                if((firstSrc & ex3_even[4:10] == addr_ra) | (secondSrc &
ex3_even[4:10] == addr_rb) | (thirdSrc & ex3_even[4:10] == addr_rc))
                    hazard2 = 1;
                end
            end
        end
    end
end

```

```

        end
    end
end
if(ex3_odd[0:10] != 11'b0000_0000_001) begin
    if(ex3_odd[32:34] == 3'b101) begin // Dependency with load instructions
        if(ex3_odd[0:8] == 9'b001100001 | ex3_odd[0:8] == 9'b001100111 |
ex3_odd[0:7] == 8'b00110100) begin
            if((firstSrc & ex3_odd[25:31] == addr_ra) | (secondSrc &
ex3_odd[25:31] == addr_rb) | (thirdSrc & ex3_odd[25:31] == addr_rc))
                hazard2 = 1;
            end
        end
    end
end

// Instruction in Execution Stage 2 should not be a No-op. Need to consider both
Pipes for instructions which take 4 stages or more.
if(ex2_even[0:10] != 11'b0100_0000_001) begin
    if(ex2_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
        if(ex2_even[0:3] != 4'b1100 & ex2_even[0:3] != 4'b1110 &
ex2_even[0:3] != 4'b1111) begin
            if((firstSrc & ex2_even[25:31] == addr_ra) | (secondSrc &
ex2_even[25:31] == addr_rb) | (thirdSrc & ex2_even[25:31] == addr_rc))
                hazard2 = 1;
            end
        else begin
            if((firstSrc & ex2_even[4:10] == addr_ra) | (secondSrc &
ex2_even[4:10] == addr_rb) | (thirdSrc & ex2_even[4:10] == addr_rc))
                hazard2 = 1;
            end
        end
    end
    if(ex2_even[32:34] == 3'b001 | ex2_even[32:34] == 3'b011) begin //
Dependency with Simple Fixed 2 unit or byte unit instructions
        if((firstSrc & ex2_even[25:31] == addr_ra) | (secondSrc &
ex2_even[25:31] == addr_rb) | (thirdSrc & ex2_even[25:31] == addr_rc))
            hazard2 = 1;
        end
    end
end
if(ex2_odd[0:10] != 11'b0000_0000_001) begin
    if(ex2_odd[32:34] == 3'b101) begin // Dependency with load instructions
        if(ex2_odd[0:8] == 9'b001100001 | ex2_odd[0:8] == 9'b001100111 |
ex2_odd[0:7] == 8'b00110100) begin
            if((firstSrc & ex2_odd[25:31] == addr_ra) | (secondSrc &
ex2_odd[25:31] == addr_rb) | (thirdSrc & ex2_odd[25:31] == addr_rc))

```

```

                                hazard2 = 1;
                                end
                                end
                                end
                                if(ex2_odd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
                                if((firstSrc & ex2_odd[25:31] == addr_ra) | (secondSrc &
ex2_odd[25:31] == addr_rb) | (thirdSrc & ex2_odd[25:31] == addr_rc))
                                hazard2 = 1;
                                end
                                end
                                end

                                // Instruction in Execution Stage 1 should not be a No-op. Need to consider both
                                Pipes for instructions which take 3 stages or more.
                                if(ex1_even[0:10] != 11'b0100_0000_001) begin
                                if(ex1_even[32:34] == 3'b010) begin // Dependency with all single precision
unit instructions
                                if(ex1_even[0:3] != 4'b1100 & ex1_even[0:3] != 4'b1110 &
ex1_even[0:3] != 4'b1111) begin
                                if((firstSrc & ex1_even[25:31] == addr_ra) | (secondSrc &
ex1_even[25:31] == addr_rb) | (thirdSrc & ex1_even[25:31] == addr_rc))
                                hazard2 = 1;
                                end
                                else begin
                                if((firstSrc & ex1_even[4:10] == addr_ra) | (secondSrc &
ex1_even[4:10] == addr_rb) | (thirdSrc & ex1_even[4:10] == addr_rc))
                                hazard2 = 1;
                                end
                                end
                                end
                                if(ex1_even[32:34] == 3'b001 | ex1_even[32:34] == 3'b011) begin //
Dependency with Simple Fixed 2 unit or byte unit instructions
                                if((firstSrc & ex1_even[25:31] == addr_ra) | (secondSrc &
ex1_even[25:31] == addr_rb) | (thirdSrc & ex1_even[25:31] == addr_rc))
                                hazard2 = 1;
                                end
                                end
                                end
                                if(ex1_odd[0:10] != 11'b0000_0000_001) begin
                                if(ex1_odd[32:34] == 3'b101) begin
                                if(ex1_odd[0:8] == 9'b001100001 | ex1_odd[0:8] == 9'b001100111 |
ex1_odd[0:7] == 8'b00110100) begin
                                if((firstSrc & ex1_odd[25:31] == addr_ra) | (secondSrc &
ex1_odd[25:31] == addr_rb) | (thirdSrc & ex1_odd[25:31] == addr_rc))
                                hazard2 = 1;
                                end
                                end
                                end
                                end
                                end

```

```

        if(ex1_odd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
            if((firstSrc & ex1_odd[25:31] == addr_ra) | (secondSrc &
ex1_odd[25:31] == addr_rb) | (thirdSrc & ex1_odd[25:31] == addr_rc))
                hazard2 = 1;
            end
        end

        // Instruction in RF Stage should not be a No-op. Need to consider both Pipes for
instructions which take 2 stages or more.
        if(instructionEven[0:10] != 11'b0100_0000_001) begin
            if(instructionEven[32:34] == 3'b010) begin // Dependency with all single
precision unit instructions
                if(instructionEven[0:3] != 4'b1100 & instructionEven[0:3] != 4'b1110 &
instructionEven[0:3] != 4'b1111) begin
                    if((firstSrc & instructionEven[25:31] == addr_ra) | (secondSrc &
instructionEven[25:31] == addr_rb) | (thirdSrc & instructionEven[25:31] == addr_rc))
                        hazard2 = 1;
                    end
                else begin
                    if((firstSrc & instructionEven[4:10] == addr_ra) | (secondSrc &
instructionEven[4:10] == addr_rb) | (thirdSrc & instructionEven[4:10] == addr_rc))
                        hazard2 = 1;
                    end
                end
            end
        end
        if(instructionEven[32:34] == 3'b001 | instructionEven[32:34] == 3'b011 |
instructionEven[32:34] == 3'b000) begin
            // Dependency with Simple Fixed 2 unit, byte unit or Simple Fixed 1
instructions
                if((firstSrc & instructionEven[25:31] == addr_ra) | (secondSrc &
instructionEven[25:31] == addr_rb) | (thirdSrc & instructionEven[25:31] == addr_rc))
                    hazard2 = 1;
                end
            end
        if(instructionOdd[0:10] != 11'b0000_0000_001) begin
            if(instructionOdd[32:34] == 3'b101) begin // Dependency with local store
instructions
                if(instructionOdd[0:8] == 9'b001100001 | instructionOdd[0:8] ==
9'b001100111 | instructionOdd[0:7] == 8'b00110100) begin
                    if((firstSrc & instructionOdd[25:31] == addr_ra) | (secondSrc &
instructionOdd[25:31] == addr_rb) | (thirdSrc & instructionOdd[25:31] == addr_rc))
                        hazard2 = 1;
                    end
                end
            end
        end
    end
end

```

```

        if(instructionOdd[32:34] == 3'b100) begin // Dependency with permute unit
instructions
            if((firstSrc & instructionOdd[25:31] == addr_ra) | (secondSrc &
instructionOdd[25:31] == addr_rb) | (thirdSrc & instructionOdd[25:31] == addr_rc))
                hazard2 = 1;
            end
        end

        // Dependency with first instruction of current set
        if(instr1[0:10] != 11'b0100_0000_001 & instr1[0:10] != 11'b0000_0000_001) begin
            if(instr1[32:34] == 3'b010) begin // Dependency with all single precision unit
instructions
                if(instr1[0:3] != 4'b1100 & instr1[0:3] != 4'b1110 & instr1[0:3] !=
4'b1111) begin
                    if((firstSrc & instr1[25:31] == addr_ra) | (secondSrc &
instr1[25:31] == addr_rb)
                        | (thirdSrc & instr1[25:31] == addr_rc))
                        hazard2 = 1;
                    end
                else begin
                    if((firstSrc & instr1[4:10] == addr_ra) | (secondSrc & instr1[4:10]
== addr_rb)
                        | (thirdSrc & instr1[4:10] == addr_rc))
                        hazard2 = 1;
                    end
                end
            end
        end
        if(instr1[32:34] == 3'b001 | instr1[32:34] == 3'b011 | instr1[32:34] == 3'b000)
begin
            // Dependency with Simple Fixed 2 unit, byte unit or Simple Fixed 1
instructions
            if((firstSrc & instr1[25:31] == addr_ra) | (secondSrc & instr1[25:31] ==
addr_rb)
                | (thirdSrc & instr1[25:31] == addr_rc))
                hazard2 = 1;
            end
        if(instr1[32:34] == 3'b101) begin // Dependency with local store instructions
            if(instr1[0:8] == 9'b001100001 | instr1[0:8] == 9'b001100111 |
instr1[0:7] == 8'b00110100) begin
                if((firstSrc & instr1[25:31] == addr_ra) | (secondSrc &
instr1[25:31] == addr_rb)
                    | (thirdSrc & instr1[25:31] == addr_rc))
                    hazard2 = 1;
                end
            end
        end
    end
end

```

```

        if(instr1[32:34] == 3'b100) begin
            // Dependency with permute unit instructions
            if((firstSrc & instr1[25:31] == addr_ra) | (secondSrc & instr1[25:31] ==
addr_rb)
                | (thirdSrc & instr1[25:31] == addr_rc))
                hazard2 = 1;
        end
        if(instr1[32:34] == 3'b110) begin
            // Dependency with branch unit instructions
            if(instr1[0:8] == 9'b001000000) begin
                if((firstSrc & instr1[25:31] == addr_ra) | (secondSrc &
instr1[25:31] == addr_rb)
                    | (thirdSrc & instr1[25:31] == addr_rc))
                    hazard2 = 1;
            end
        end
    end
end
end

if(hazard1 == 1) begin // Data hazard in first instruction, cannot push any instruction further
    nextInstr1 = instr1[0:31];
    nextInstr2 = instr2[0:31];
    PCSelect = 1;
end
else if(instr1[0:10] == 11'b0000_0000_000) begin // First instruction is a stop instruction, need
to push no-ops to fill both pipes
    if(wb_even[0:10] == 11'b0100_0000_001 & ex7_even[0:10] == 11'b0100_0000_001 &
ex6_even[0:10] == 11'b0100_0000_001 & ex5_even[0:10] == 11'b0100_0000_001
        & ex4_even[0:10] == 11'b0100_0000_001 & ex3_even[0:10] == 11'b0100_0000_001
        & ex2_even[0:10] == 11'b0100_0000_001 & ex1_even[0:10] == 11'b0100_0000_001
        & instructionEven[0:10] == 11'b0100_0000_001 & wb_odd[0:10] ==
11'b0000_0000_001 & ex7_odd[0:10] == 11'b0000_0000_001 & ex6_odd[0:10] == 11'b0000_0000_001
        & ex5_odd[0:10] == 11'b0000_0000_001 & ex4_odd[0:10] == 11'b0000_0000_001 &
ex3_odd[0:10] == 11'b0000_0000_001 & ex2_odd[0:10] == 11'b0000_0000_001
        & ex1_odd[0:10] == 11'b0000_0000_001 & instructionOdd[0:10] ==
11'b0000_0000_001) begin
        nextInstr1 = 32'b0100_0000_0010_0000_0000_0000_0000_0000;
        nextInstr2 = 32'b0000_0000_0010_0000_0000_0000_0000_0000;
    end
    else begin
        nextInstr1 = instr1;
        nextInstr2 = 32'b0000_0000_0010_0000_0000_0000_0000_0000;
    end
    PCSelect = 1;
end

```

```

        end
        else begin // No data hazards in first instruction
            if(instr1[32:34] != 3'b111 & instr1[32:34] < 3'b100) begin // First instruction goign to
even pipe
                instrEvenReg = instr1;
                if(instr2[32:34] != 3'b111 & instr2[32:34] < 3'b100) // Second instruction also
going to even pipe, HAZARD
                    hazard2 = 1;
                else if(instr2[32:34] != 3'b111 & instr2[32:34] >= 3'b100) begin // If second
instruction going to odd pipe
                    if(instr2[0:8] != 9'b001100100 & instr2[0:8] != 9'b001000000 &
instr2[0:8] != 9'b001000010 & instr2[0:8] != 9'b001000001
                        & instr2[0:8] != 9'b001000111 & instr2[0:7] != 8'b00100100) begin //
Check Rt Rt structural hazard
                        if(((instr1[0:3] == 4'b1100 | instr1[0:3] == 4'b1110 | instr1[0:3]
== 4'b1111) & (instr2[25:31] == instr1[4:10]))
                            | (((instr1[0:3] != 4'b1100 & instr1[0:3] != 4'b1110 & instr1[0:3] !=
4'b1111) & (instr2[25:31] == instr1[25:31])))
                                hazard2 = 1;
                        end
                    end
                end
            end
            else if(instr1[32:34] != 3'b111 & instr1[32:34] >= 3'b100) begin // First instruction going
to odd pipe
                instrOddReg = instr1;
                firstBranch = (instr1[32:34] == 3'b110);
                if(instr2[32:34] != 3'b111 & instr2[32:34] >= 3'b100) // Second instruction also
going to odd pipe, HAZARD
                    hazard2 = 1;
                else if(instr2[32:34] != 3'b111 & instr2[32:34] < 3'b100) begin // If second
instruction going to even pipe
                    if(((instr2[0:3] == 4'b1100 | instr2[0:3] == 4'b1110 | instr2[0:3] ==
4'b1111) & (instr1[25:31] == instr2[4:10]))
                        | (((instr2[0:3] != 4'b1100 & instr2[0:3] != 4'b1110 & instr2[0:3] !=
4'b1111) & (instr1[25:31] == instr2[25:31])))
                            // Check Rt Rt structural hazard
                            if(instr1[0:8] != 9'b001100100 & instr1[0:8] != 9'b001000000 &
instr1[0:8] != 9'b001000010 & instr1[0:8] != 9'b001000001
                                & instr1[0:8] != 9'b001000111 & instr1[0:7] != 8'b00100100)
                                    begin
                                        hazard2 = 1;
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```

```

        nextInstr1 = 32'b0100_0000_0010_0000_0000_0000_0000_0000;
        if(hazard2 == 1 | hazard2 == 1) begin // Second instruction has a hazard detected,
cannot push it further
            PCSelect = 1;
            nextInstr2 = instr2[0:31];
        end
        else if(instr2[0:10] == 11'b0000_0000_000) begin // Second instruction is a stop
instruction, need to push no-ops to fill both pipes
            if(wb_even[0:10] == 11'b0100_0000_001 & ex7_even[0:10] ==
11'b0100_0000_001 & ex6_even[0:10] == 11'b0100_0000_001 & ex5_even[0:10] == 11'b0100_0000_001
            & ex4_even[0:10] == 11'b0100_0000_001 & ex3_even[0:10] ==
11'b0100_0000_001 & ex2_even[0:10] == 11'b0100_0000_001 & ex1_even[0:10] == 11'b0100_0000_001
            & instructionEven[0:10] == 11'b0100_0000_001 & wb_odd[0:10] ==
11'b0000_0000_001 & ex7_odd[0:10] == 11'b0000_0000_001 & ex6_odd[0:10] == 11'b0000_0000_001
            & ex5_odd[0:10] == 11'b0000_0000_001 & ex4_odd[0:10] ==
11'b0000_0000_001 & ex3_odd[0:10] == 11'b0000_0000_001 & ex2_odd[0:10] == 11'b0000_0000_001
            & ex1_odd[0:10] == 11'b0000_0000_001 & instructionOdd[0:10] ==
11'b0000_0000_001) begin
                nextInstr1 = 32'b0100_0000_0010_0000_0000_0000_0000_0000;
                nextInstr2 = 32'b0000_0000_0010_0000_0000_0000_0000_0000;
            end
            else begin
                nextInstr2 = instr2;
                nextInstr1 = 32'b0100_0000_0010_0000_0000_0000_0000_0000;
            end
            PCSelect = 1;
        end
    else begin // Safe to push second instruction
        if(instr2[32:34] != 3'b111 & instr2[32:34] < 3'b100)
            instrEvenReg = instr2;
        else if(instr2[32:34] != 3'b111 & instr2[32:34] >= 3'b100)
            instrOddReg = instr2;
        nextInstr2 = 32'b0000_0000_0010_0000_0000_0000_0000_0000;
    end
end

if(PCSelect) begin
    oldPC = PCRead;
    oldPredict_0 = predictRead_0;
    oldPredict_1 = predictRead_1;
    oldPredictPC_0 = predictPCRead_0;
    oldPredictPC_1 = predictPCRead_1;
end

```


end

endmodule

Processing Unit

/* Top Processing Unit module for Cell-SPU-Lite */

```

module ProcessingUnit(
    input                                clk, reset, flush, predictIn,
    input [0:34]                        instructionEven, instructionOdd,
    input [0:31]                        PCin, predictPCin,
    output logic [0:65]                 PCout);

    logic [0:134]                        data_ra_0, data_rb_0, data_rc_0, data_ra_1, data_rb_1, data_rc_1;
    logic [0:138]                        fwe2_out, fwe3_out, fwe4_out, fwe5_out, fwe6_out, fwe7_out, rf_wbe_out;
    logic [0:138]                        fwo1_out, fwo2_out, fwo3_out, fwo4_out, fwo5_out, fwo6_out, fwo7_out,
rf_wbo_out;
    logic [0:127]                        data_out_ra_0, data_out_rb_0, data_out_rc_0, data_out_ra_1, data_out_rb_1,
data_out_rc_1; //);
    logic [0:10]                        opcode_0, opcode_1, opcode_reg_0, opcode_reg_1;
    logic [0:17]                        immediate_reg_0, immediate_reg_1;
    logic [0:6]                         addr_rt_0, addr_rt_reg_0, addr_rt_reg_1;
    logic                               reset_reg;
    logic [0:31]                        PC_in_reg, predictPCin_reg, predictPCin_reg_1;
    logic                               flushInstr, flush_reg, flush_reg_1, predictIn_reg, predictIn_reg_1;

    registerFile rf(.clk(clk), .reset(reset), .wr_en_0(rf_wbe_out[131]), .wr_en_1(rf_wbo_out[131]),
.addr_rb_0(instructionEven[11:17]), .addr_ra_0(instructionEven[18:24]),
                                .addr_rc_0(instructionEven[25:31]), .addr_rb_1(instructionOdd[11:17]),
.addr_ra_1(instructionOdd[18:24]), .addr_rc_1(instructionOdd[25:31]),
                                .wr_addr_0(rf_wbe_out[132:138]), .wr_addr_1(rf_wbo_out[132:138]),
.wr_data_0(rf_wbe_out[0:127]), .wr_data_1(rf_wbo_out[0:127]),
                                .data_ra_0(data_ra_0), .data_rb_0(data_rb_0), .data_rc_0(data_rc_0),
.data_ra_1(data_ra_1), .data_rb_1(data_rb_1), .data_rc_1(data_rc_1));

    DataForward df(.data_ra_0(data_ra_0), .data_rb_0(data_rb_0), .data_rc_0(data_rc_0),
.data_ra_1(data_ra_1), .data_rb_1(data_rb_1), .data_rc_1(data_rc_1),
                                .fwe2_out(fwe2_out), .fwe3_out(fwe3_out), .fwe4_out(fwe4_out),
.fwe5_out(fwe5_out), .fwe6_out(fwe6_out), .fwe7_out(fwe7_out),
                                .rf_wbe_out(rf_wbe_out), .fwo1_out(fwo1_out), .fwo2_out(fwo2_out),
.fwo3_out(fwo3_out), .fwo4_out(fwo4_out), .fwo5_out(fwo5_out),
                                .fwo6_out(fwo6_out), .fwo7_out(fwo7_out), .rf_wbo_out(rf_wbo_out),
.data_out_ra_0(data_out_ra_0), .data_out_rb_0(data_out_rb_0),
                                .data_out_rc_0(data_out_rc_0), .data_out_ra_1(data_out_ra_1),
.data_out_rb_1(data_out_rb_1), .data_out_rc_1(data_out_rc_1));

    EvenPipe ep(.clk(clk), .reset(reset_reg), .data_ra(data_out_ra_0), .data_rb(data_out_rb_0),
.data_rc(data_out_rc_0), .immediate(immediate_reg_0), .opcode(opcode_0),

```

```

        .addr_rt(addr_rt_reg_0), .fwe2_out(fwe2_out), .fwe3_out(fwe3_out),
.fwe4_out(fwe4_out), .fwe5_out(fwe5_out), .fwe6_out(fwe6_out),
        .fwe7_out(fwe7_out), .rf_wbe_out(rf_wbe_out), .flush(flushInstr));

```

```

    oddPipe op(.clk(clk), .reset(reset_reg), .data_ra(data_out_ra_1), .data_rb(data_out_rb_1),
.data_rc(data_out_rc_1), .immediate(immediate_reg_1), .opcode(opcode_1),
        .addr_rt(addr_rt_reg_1), .PCin(PC_in_reg), .fwo1_out(fwo1_out),
.fwo2_out(fwo2_out), .fwo3_out(fwo3_out), .fwo4_out(fwo4_out), .fwo5_out(fwo5_out),
        .fwo6_out(fwo6_out), .fwo7_out(fwo7_out), .rf_wbo_out(rf_wbo_out),
.PCout(PCout), .predictIn(predictIn_reg), .predictPCin(predictPCin_reg));

```

```

always_ff @(posedge clk) begin

```

```

    reset_reg <= reset;
    PC_in_reg <= PCin;
    predictPCin_reg <= predictPCin;
    predictPCin_reg_1 <= predictPCin_reg;
    predictIn_reg <= predictIn;
    predictIn_reg_1 <= predictIn_reg;
    flush_reg <= flush;
    flush_reg_1 <= flush_reg;

```

```

    if(reset_reg | PCout[65]) begin

```

```

        opcode_reg_0 <= 11'b010000000001;
        addr_rt_reg_0 <= 0;
        immediate_reg_0 <= 0;

```

```

        opcode_reg_1 <= 11'b000000000001;
        addr_rt_reg_1 <= 0;
        immediate_reg_1 <= 0;

```

```

    end else begin

```

```

        opcode_reg_0 <= instructionEven[0:10];
        addr_rt_reg_0 <= addr_rt_0;
        immediate_reg_0 <= instructionEven[7:24];

```

```

        opcode_reg_1 <= instructionOdd[0:10];
        addr_rt_reg_1 <= instructionOdd[25:31];
        immediate_reg_1 <= instructionOdd[7:24];

```

```

    end

```

```

end

always_comb begin

    addr_rt_0 = (instructionEven[0:3] == 4'b1100 | instructionEven[0:3] == 4'b1110 |
instructionEven[0:3] == 4'b1111)
                ? instructionEven[4:10] : instructionEven[25:31];
    flushInstr = flush_reg_1 & PCout[32];
    opcode_0 = (PCout[65]) ? 11'b0100_0000_001 : opcode_reg_0;
    opcode_1 = (PCout[65]) ? 11'b0000_0000_001 : opcode_reg_1;
end

endmodule

```

Register File

```
/* Module for register file:
 * 128 registers each of 128 bits width
 * Total 6 read ports to read operands and 2 write ports to write data each cycle.
 * If the read address on any of the read ports is same as the write address in the same cycle then the input
data is forwarded to the output bus thereby simulating
 * the ability to read and write from a location in the same cycle.
 */
module registerFile(
    input                clk, reset, wr_en_0, wr_en_1,
    input [0:6]          addr_ra_0, addr_rb_0, addr_rc_0,
    input [0:6]          addr_ra_1, addr_rb_1, addr_rc_1,
    input [0:6]          wr_addr_0, wr_addr_1,
    input [0:127]        wr_data_0, wr_data_1,
    output reg [0:134]    data_ra_0, data_rb_0, data_rc_0,
    output reg [0:134]    data_ra_1, data_rb_1, data_rc_1);

    logic [0:127] [0:127] mem;
    int i;

    always_ff @(posedge clk) begin

        // Reading data for the even pipe and appending source register addresses for data forwarding
        data_ra_0[0:127] <= (wr_en_0 & (addr_ra_0 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_ra_0
== wr_addr_1)) ? wr_data_1 : mem[addr_ra_0]);
        data_ra_0[128:134] <= addr_ra_0;
        data_rb_0[0:127] <= (wr_en_0 & (addr_rb_0 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_rb_0
== wr_addr_1)) ? wr_data_1 : mem[addr_rb_0]);
        data_rb_0[128:134] <= addr_rb_0;
        data_rc_0[0:127] <= (wr_en_0 & (addr_rc_0 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_rc_0
== wr_addr_1)) ? wr_data_1 : mem[addr_rc_0]);
        data_rc_0[128:134] <= addr_rc_0;

        // Reading data for the odd pipe and appending source register addresses for data forwarding
        data_ra_1[0:127] <= (wr_en_0 & (addr_ra_1 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_ra_1
== wr_addr_1)) ? wr_data_1 : mem[addr_ra_1]);
        data_ra_1[128:134] <= addr_ra_1;
        data_rb_1[0:127] <= (wr_en_0 & (addr_rb_1 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_rb_1
== wr_addr_1)) ? wr_data_1 : mem[addr_rb_1]);
        data_rb_1[128:134] <= addr_rb_1;
        data_rc_1[0:127] <= (wr_en_0 & (addr_rc_1 == wr_addr_0)) ? wr_data_0 : ((wr_en_1 & (addr_rc_1
== wr_addr_1)) ? wr_data_1 : mem[addr_rc_1]);
        data_rc_1[128:134] <= addr_rc_1;
```

```

// Writing data from the even pipe
if(wr_en_0)
    mem[wr_addr_0] <= wr_data_0;

// Writing data from odd pipe
if(wr_en_1)
    mem[wr_addr_1] <= wr_data_1;

if(reset) begin
    for (i=0; i<128; i=i+1) begin
        mem[i] <= 0;
    end
end

end

endmodule

```

Data Forwarding Unit

```
/* Module implementing Data forwarding from the execution pipes
 * Purely combinational block and comes right after the register file to make sure that correct values are only
 * pushed as operands to the execution pipes
 */
module DataForward(
    input          [0:134]      data_ra_0, data_rb_0, data_rc_0,
    input          [0:134]      data_ra_1, data_rb_1, data_rc_1,
    input          [0:138]      fwe2_out, fwe3_out, fwe4_out, fwe5_out, fwe6_out, fwe7_out,
    rf_wbe_out,
    input          [0:138]      fwo1_out, fwo2_out, fwo3_out, fwo4_out, fwo5_out, fwo6_out,
    fwo7_out, rf_wbo_out,
    output logic    [0:127]      data_out_ra_0, data_out_rb_0, data_out_rc_0, data_out_ra_1,
    data_out_rb_1, data_out_rc_1);

    always_comb begin

        data_out_ra_0 = data_ra_0[0:127];
        data_out_rb_0 = data_rb_0[0:127];
        data_out_rc_0 = data_rc_0[0:127];
        data_out_ra_1 = data_ra_1[0:127];
        data_out_rb_1 = data_rb_1[0:127];
        data_out_rc_1 = data_rc_1[0:127];

        // Data Forwarding data_ra for even pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_ra_0[128:134]) data_out_ra_0 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_ra_0[128:134]) data_out_ra_0 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo7_out[0:127];
        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo5_out[0:127];
```

```

        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo4_out[0:127];
        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo2_out[0:127];
        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_ra_0[128:134]) data_out_ra_0 = fwo1_out[0:127];

// Data Forwarding data_rb for even pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_rb_0[128:134]) data_out_rb_0 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_rb_0[128:134]) data_out_rb_0 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo7_out[0:127];
        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo5_out[0:127];
        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo4_out[0:127];
        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo2_out[0:127];

```



```

        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_rb_0[128:134]) data_out_rb_0 = fwo1_out[0:127];

        // Data Forwarding data_rc for even pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_rc_0[128:134]) data_out_rc_0 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_rc_0[128:134]) data_out_rc_0 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo7_out[0:127];
        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo5_out[0:127];
        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo4_out[0:127];
        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo2_out[0:127];
        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_rc_0[128:134]) data_out_rc_0 = fwo1_out[0:127];

        // Data Forwarding data_ra for odd pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_ra_1[128:134]) data_out_ra_1 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_ra_1[128:134]) data_out_ra_1 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo7_out[0:127];

```

```

        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo5_out[0:127];
        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo4_out[0:127];
        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo2_out[0:127];
        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_ra_1[128:134]) data_out_ra_1 = fwo1_out[0:127];

        // Data Forwarding data_rb for odd pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_rb_1[128:134]) data_out_rb_1 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_rb_1[128:134]) data_out_rb_1 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo7_out[0:127];
        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo5_out[0:127];
        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo4_out[0:127];

```

```

        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo2_out[0:127];
        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_rb_1[128:134]) data_out_rb_1 = fwo1_out[0:127];

        // Data Forwarding data_rc for odd pipe
        if(rf_wbe_out[131] == 1 & rf_wbe_out[132:138] == data_rc_1[128:134]) data_out_rc_1 =
rf_wbe_out[0:127];
        if(rf_wbo_out[131] == 1 & rf_wbo_out[132:138] == data_rc_1[128:134]) data_out_rc_1 =
rf_wbo_out[0:127];
        if(fwe7_out[131] == 1 & fwe7_out[128:130] <= 3'd7 & fwe7_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe7_out[0:127];
        if(fwo7_out[131] == 1 & fwo7_out[128:130] <= 3'd7 & fwo7_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo7_out[0:127];
        if(fwe6_out[131] == 1 & fwe6_out[128:130] <= 3'd6 & fwe6_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe6_out[0:127];
        if(fwo6_out[131] == 1 & fwo6_out[128:130] <= 3'd6 & fwo6_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo6_out[0:127];
        if(fwe5_out[131] == 1 & fwe5_out[128:130] <= 3'd5 & fwe5_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe5_out[0:127];
        if(fwo5_out[131] == 1 & fwo5_out[128:130] <= 3'd5 & fwo5_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo5_out[0:127];
        if(fwe4_out[131] == 1 & fwe4_out[128:130] <= 3'd4 & fwe4_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe4_out[0:127];
        if(fwo4_out[131] == 1 & fwo4_out[128:130] <= 3'd4 & fwo4_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo4_out[0:127];
        if(fwe3_out[131] == 1 & fwe3_out[128:130] <= 3'd3 & fwe3_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe3_out[0:127];
        if(fwo3_out[131] == 1 & fwo3_out[128:130] <= 3'd3 & fwo3_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo3_out[0:127];
        if(fwe2_out[131] == 1 & fwe2_out[128:130] <= 3'd2 & fwe2_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwe2_out[0:127];
        if(fwo2_out[131] == 1 & fwo2_out[128:130] <= 3'd2 & fwo2_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo2_out[0:127];
        if(fwo1_out[131] == 1 & fwo1_out[128:130] <= 3'd1 & fwo1_out[132:138] ==
data_rc_1[128:134]) data_out_rc_1 = fwo1_out[0:127];

end

```

endmodule

Even Pipe Module

```
/* Even Pipe module containing SimpleFixed1, SimpleFixed2, Byte and Single Precision units
 * It also hosts all the forwarding registers through which the outputs from the units are shifted.
 */
module EvenPipe(
    input                                clk, reset, flush,
    input [0:127]                        data_ra, data_rb, data_rc,
    input [0:17]                         immediate,
    input [0:6]                         addr_rt,
    input [0:10]                        opcode,
    output logic [0:138]                 fwe2_out, fwe3_out, fwe4_out, fwe5_out, fwe6_out, fwe7_out, rf_wbe_out);

    logic [0:138] sp2_out, b1_out, fp1_out, fp2_out, fwe3_reg, fwe6_reg, fwe7_reg;

    SimpleFixed1 sp1(.clk(clk), .reset(reset), .data_ra(data_ra), .data_rb(data_rb), .immediate(immediate),
        .addr_rt(addr_rt), .opcode(opcode), .flush(flush)
        , .out_data(fwe2_out));

    SimpleFixed2 sp2(.clk(clk), .reset(reset), .data_ra(data_ra), .data_rb(data_rb),
        .immediate(immediate[4:10]), .addr_rt(addr_rt), .flush(flush)
        , .opcode(opcode), .out_data(sp2_out));

    Byte b1(.clk(clk), .reset(reset), .data_ra(data_ra), .data_rb(data_rb), .addr_rt(addr_rt), .opcode(opcode),
        .flush(flush), .out_data(b1_out));

    singlePrec fp1(.clk(clk), .reset(reset), .ra(data_ra), .rb(data_rb), .rc(data_rc), .opcode11(opcode),
        .opcode10(opcode[0:9]), .opcode8(opcode[0:7])
        , .opcode4(opcode[0:3]), .immediate10(immediate[1:10]),
        .immediate8(immediate[3:10]), .pipe6out(fp1_out), .pipe7out(fp2_out), .addr_rt(addr_rt)
        , .flush(flush));

    always_ff @(posedge clk) begin

        fwe3_reg <= fwe2_out;
        fwe4_out <= fwe3_out;
        fwe5_out <= fwe4_out;
        fwe6_reg <= fwe5_out;
        fwe7_reg <= fwe6_out;
        rf_wbe_out <= fwe7_out;

    end

    always_comb begin

        if(sp2_out[131] == 1'b1) fwe3_out = sp2_out;
        else if(b1_out[131] == 1'b1) fwe3_out = b1_out;
```

```
else fwe3_out = fwe3_reg;
```

```
if(fp1_out[131] == 1'b1) fwe6_out = fp1_out;  
else fwe6_out = fwe6_reg;
```

```
if(fp2_out[131] == 1'b1) fwe7_out = fp2_out;  
else fwe7_out = fwe7_reg;
```

```
end
```

```
endmodule
```

Odd Pipe Module

/* Odd Pipe contains the branch, local store, and permute units.

* It also hosts all the forwarding registers through which the outputs from the units are shifted.

```
*/
module oddPipe(
    input                                clk, reset, predictIn,
    input [0:127]                        data_ra, data_rb, data_rc,
    input [0:17]                         immediate,
    input [0:6]                          addr_rt,
    input [0:10]                         opcode,
    input [0:31]                         PCin, predictPCin,
    output logic [0:65]                  PCout,
    output logic [0:138]                 fwo1_out, fwo2_out, fwo3_out, fwo4_out, fwo5_out, fwo6_out, fwo7_out,
    rf_wbo_out);

    logic [0:138] perm_out, LS_out, br_out, fwo3_reg, fwo6_reg;

    permute perm(.clk(clk), .reset(reset), .ra(data_ra), .rb(data_rb), .opcode11(opcode),
        .immediate7(immediate[4:10]), .WBpipe3(perm_out)
        , .addr_rt(addr_rt));

    localstore LS(.clk(clk), .reset(reset), .ra(data_ra), .rt(data_rc), .opcode9(opcode[0:8]),
        .opcode8(opcode[0:7])
        , .immediate16(immediate[2:17]), .immediate10(immediate[1:10]), .FWpipe6(LS_out)
        , .addr_rt(addr_rt), .PCin(PCin));

    branch br(.clk(clk), .reset(reset), .addr_rt(addr_rt), .opcode9(opcode[0:8]), .immediate16(immediate[2:17])
        , .pipe(br_out), .PCpipe(PCout), .PCin(PCin), .predictPCin(predictPCin), .predictIn(predictIn),
        .rt(data_rc));

    always_ff @(posedge clk) begin

        fwo2_out <= fwo1_out;
        fwo3_reg <= fwo2_out;
        fwo4_out <= fwo3_out;
        fwo5_out <= fwo4_out;
        fwo6_reg <= fwo5_out;
        fwo7_out <= fwo6_out;
        rf_wbo_out <= fwo7_out;

    end
```

```
always_comb begin

    fwo1_out = br_out;

    if(perm_out[131] == 1'b1) fwo3_out = perm_out;
    else fwo3_out = fwo3_reg;

    if(LS_out[131] == 1'b1) fwo6_out = LS_out;
    else fwo6_out = fwo6_reg;

end

endmodule
```


Byte Unit

/* Module for Byte unit containing all instruction coded

* Imagine all the calculations for all the instructions would be done in the first cycle and then just shifted through

* Things to Note:-

* 1. "result", "result_reg" and "result_reg_1" are internal logic since the pipeline depth is 2. "out_data" is output going to

* forwarding registers.

*/

module Byte(

```
input          clk, reset, flush,
input [0:127]  data_ra, data_rb,
input [0:6]    addr_rt,
input [0:10]   opcode,
output logic [0:138] out_data);
```

```
logic [0:138] result, result_reg, result_reg_1;
logic [0:9]   temp;
```

```
always_ff @(posedge clk) begin
```

```
  if(reset) begin
    result_reg <= 0;
    result_reg_1 <= 0;
    out_data <= 0;
  end
  else begin
    result_reg <= result;
    result_reg_1 <= (flush) ? 0 : result_reg;
    out_data <= result_reg_1;
  end
end
```

```
end
```

```
always_comb begin
```

```
  integer i,j;
  result[128:130] = 3'd4;
  result[131] = 1'b1;
  result[132:138] = addr_rt;
  case(opcode)
    11'b01010110100: begin // Count Ones in bytes
      for(i = 0; i < 16; i++) begin
        temp = 0;
        for(j = 0; j < 8; j++) begin
          if(data_ra[(i * 8) + j] == 1'b1) temp = temp + 1;
        end
      end
    end
  end
```

```

        end
        result[(i * 8) +:8] = temp[2:9];
    end
end
11'b00011010011: begin // Average Bytes
    for(i = 0; i < 16; i++) begin
        temp = {{2'b00}, data_ra[(i * 8) +:8]} + {{2'b00}, data_rb[(i * 8) +:8]} + 1;
        temp = temp >> 1;
        result[(i * 8) +:8] = temp[2:9];
    end
end
11'b00001010011: begin // Absolute Differences of bytes
    for(i = 0; i < 16; i++) result[(i * 8) +:8] = (data_rb[(i * 8) +:8] > data_ra[(i * 8) +:8]) ?
(data_rb[(i * 8) +:8] - data_ra[(i * 8) +:8])
: (data_ra[(i * 8) +:8] - data_rb[(i * 8)
+:8]);
    end
    11'b01001010011: begin // Sum Bytes into Halfwords
        j = 0;
        for(i = 0; i < 4; i++) begin
            result[(i * 32) +:16] = data_rb[(j * 32) +:8] + data_rb[((j * 32) + 8) +:8] + data_rb[((j *
32)+16) +:8] + data_rb[((j * 32) + 24) +:8];
            result[((i * 32) + 16) +:16] = data_ra[(j * 32) +:8] + data_ra[((j * 32) + 8) +:8] +
data_ra[((j * 32)+16) +:8] + data_ra[((j * 32) + 24) +:8];
            j = j + 1;
        end
    end
end

    default: result = 0;
endcase
end

endmodule

```

Simple Fixed 1 Unit

/* Module for Simple Fixed 1 unit containing all instruction coded

* Imagine all the calculations for all the instructions would be done in the first cycle and then just shifted through

* Things to Note:-

* 1. "result" and "result_reg" are internal logic since the pipeline depth is 2. "out_data" is output going to

* forwarding registers.

*/

module SimpleFixed1(

input clk, reset, flush,

input [0:127] data_ra, data_rb,

input [0:17] immediate,

input [0:6] addr_rt,

input [0:10] opcode,

output logic [0:138] out_data);

logic [0:138] result, result_reg;

logic [0:15] temp16a, temp16b, temp16c, temp16d;

logic [0:7] temp8a, temp8b, temp8c, temp8d;

logic [0:3] temp4a, temp4b, temp4c, temp4d;

always_ff @(posedge clk) begin

if(reset) begin

result_reg <= 0;

out_data <= 0;

end

else begin

result_reg <= result;

out_data <= (flush) ? 0 : result_reg;

end

end

always_comb begin

integer i, j;

result[128:130] = 3'd2;

result[131] = 1'b1;

result[132:138] = addr_rt;

case(opcode) // RR format with 11 bit opcode

11'b00011001000: begin // Add Halfword

for(i = 0; i < 8 ; i++) result[(i * 16) +:16] = \$signed(data_ra[(i * 16) +:16]) + \$signed(data_rb[(i * 16) +:16]);

end

```

11'b00011000000: begin // Add word
    for(i = 0; i < 4 ; i++) result[(i * 32) +:32] = $signed(data_ra[(i * 32) +:32]) + $signed(data_rb[(i * 32)
+:32]);
    end
11'b00001001000: begin // Subtract from halfword
    for(i = 0; i < 8 ; i++) result[(i * 16) +:16] = $signed(data_rb[(i * 16) +:16]) - $signed(data_ra[(i * 16)
+:16]);
    end
11'b00001000000: begin // Subtract from word
    for(i = 0; i < 4 ; i++) result[(i * 32) +:32] = $signed(data_rb[(i * 32) +:32]) - $signed(data_ra[(i * 32)
+:32]);
    end
11'b01010100101: begin // Count Leading Zeroes
    result[0:127] = 128'd0;
    for(i = 0; i < 4; i++) begin
        for(j = 0; j < 32; j++) begin
            if(data_ra[(i * 32) + j] == 0) result[(i * 32) +:32] = result[(i * 32) +:32] + 1;
            else break;
        end
    end
end
11'b00110110110: begin // Form Select mask for bytes
    for(i = 0; i < 16 ; i++) begin
        if(data_ra[16 + i] == 0)
            result[(i * 8) +:8] = 8'h00;
        else
            result[(i * 8) +:8] = 8'hff;
        end
    end
11'b00110110101: begin // Form Select mask for halfword
    for(i = 0; i < 8 ; i++) begin
        if(data_ra[24 + i] == 0)
            result[(i * 16) +:16] = 16'h0000;
        else
            result[(i * 16) +:16] = 16'hffff;
        end
    end
11'b00110110100: begin // Form Select mask for word
    for(i = 0; i < 4 ; i++) begin
        if(data_ra[28 + i] == 0)
            result[(i * 32) +:32] = 32'h0000_0000;
        else
            result[(i * 32) +:32] = 32'hffff_ffff;
        end
    end
end

```

```

end
11'b00011000001: begin // and
    result[0:127] = data_ra & data_rb;
end
11'b00001000001: begin // or
    result[0:127] = data_ra | data_rb;
end
11'b00111110000: begin // or across
    result[96:127] = data_ra[0:31] | data_ra[32:63] | data_ra[64:95] | data_ra[96:127];
    result[0:95] = 0;
end
11'b01001000001: begin // xor
    result[0:127] = data_ra ^ data_rb;
end
11'b01111010000: begin // Compare Equal Byte
    for(i = 0; i < 16; i++) result[(i * 8) +:8] = (data_ra[(i * 8) +:8] == data_rb[(i * 8) +:8]) ? 8'hff :
8'h00;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01111001000: begin // Compare Equal Halfword
    for(i = 0; i < 8; i++) result[(i * 16) +:16] = (data_ra[(i * 16) +:16] == data_rb[(i * 16) +:16]) ?
16'hffff : 16'h0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01111000000: begin // Compare Equal Word
    for(i = 0; i < 4; i++) result[(i * 32) +:32] = (data_ra[(i * 32) +:32] == data_rb[(i * 32) +:32]) ?
32'hffff_ffff : 32'h0000_0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01001010000: begin // Compare Greater than Byte
    for(i = 0; i < 16; i++) result[(i * 8) +:8] = ($signed(data_ra[(i * 8) +:8]) > $signed(data_rb[(i *
8) +:8])) ? 8'hff : 8'h00;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01001001000: begin // Compare Greater than Halfword
    for(i = 0; i < 8; i++) result[(i * 16) +:16] = ($signed(data_ra[(i * 16) +:16]) > $signed(data_rb[(i *
* 16) +:16])) ? 16'hffff : 16'h0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
end

```

```

11'b01001000000: begin // Compare Greater than Word
    for(i = 0; i < 4; i++) result[(i * 32) +:32] = ($signed(data_ra[(i * 32) +:32]) > $signed(data_rb[(i
* 32) +:32])) ? 32'hffff_ffff : 32'h0000_0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01011010000: begin // Compare Logical Greater than Byte
    for(i = 0; i < 16; i++) result[(i * 8) +:8] = (data_ra[(i * 8) +:8] > data_rb[(i * 8) +:8]) ? 8'hff :
8'h00;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01011001000: begin // Compare Logical Greater than Halfword
    for(i = 0; i < 8; i++) result[(i * 16) +:16] = (data_ra[(i * 16) +:16] > data_rb[(i * 16) +:16]) ?
16'hffff : 16'h0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
11'b01011000000: begin // Compare Logical Greater than Word
    for(i = 0; i < 4; i++) result[(i * 32) +:32] = (data_ra[(i * 32) +:32] > data_rb[(i * 32) +:32]) ?
32'hffff_ffff : 32'h0000_0000;
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
default: begin

case(opcode[0:8]) // RI16 format with 9 bit opcode
9'b010000011: begin // Immediate Load Half word
    for(i = 0; i < 8; i++) result[(i * 16) +:16] = immediate[2:17];
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
9'b010000010: begin // Immediate Load halfword upper
    for(i = 0; i < 4; i++) result[(i * 32) +:32] = {immediate[2:17], 16'b0000};
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
9'b010000001: begin // Immediate Load word
    for(i = 0; i < 4; i++) result[(i * 32) +:32] = {{8{immediate[2]}}, immediate[2:17]};
    result[128:130] = 3'd2;
    result[131] = 1'b1;
end
9'b011000001: begin // Immediate Load halfword lower
    for(i = 0; i < 8; i++) result[(i * 16) +:16] = immediate[2:17];

```

```

        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end

    9'b001100101: begin // Form select mask for bytes immediate
        for(i = 0; i < 8; i++) result[(i * 16) +:16] = immediate[2:17];
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end

    default: begin

        case(opcode[0:7]) // RI10 format with 8 bit opcode
            8'b00011101: begin // Add Halfword immediate
                for(i = 0; i < 8 ; i++) result[(i * 16) +:16] = $signed(data_ra[(i * 16) +:16]) +
$signed(immediate[1:10]);
                result[128:130] = 3'd2;
                result[131] = 1'b1;
            end
            8'b00011100: begin // Add word immediate
                for(i = 0; i < 4 ; i++) result[(i * 32) +:32] = $signed(data_ra[(i * 32) +:32]) +
$signed(immediate[1:10]);
                result[128:130] = 3'd2;
                result[131] = 1'b1;
            end
            8'b00001101: begin // Subtract from Halfword immediate
                for(i = 0; i < 8 ; i++) result[(i * 16) +:16] = $signed(immediate[1:10]) -
$signed(data_ra[(i * 16) +:16]);
                result[128:130] = 3'd2;
                result[131] = 1'b1;
            end
            8'b00001100: begin // Subtract from word immediate
                for(i = 0; i < 4 ; i++) result[(i * 32) +:32] = $signed(immediate[1:10]) -
$signed(data_ra[(i * 32) +:32]);
                result[128:130] = 3'd2;
                result[131] = 1'b1;
            end
            8'b00010110: begin // and byte immediate
                for(i = 0; i < 16; i++) result[(i * 8) +:8] = data_ra[(i * 8) +:8] & (immediate[1:10]
& 16'h00ff);

                result[128:130] = 3'd2;
                result[131] = 1'b1;
            end
            8'b00010101: begin // and halfword immediate
                for(i = 0; i < 8; i++) result[(i * 16) +:16] = data_ra[(i * 16) +:16] &
{{6{immediate[1]}}, immediate[1:10]};

```

```

        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b00010100: begin // and word immediate
        for(i = 0; i < 4; i++) result[(i * 32) +:32] = data_ra[(i * 32) +:32] &
{{22{immediate[1]}}, immediate[1:10]};
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b00000110: begin // or byte immediate
        for(i = 0; i < 16; i++) result[(i * 8) +:8] = data_ra[(i * 8) +:8] | (immediate[1:10]
& 16'h00ff);

        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b00000101: begin // or halfword immediate
        for(i = 0; i < 8; i++) result[(i * 16) +:16] = data_ra[(i * 16) +:16] |
{{6{immediate[1]}}, immediate[1:10]};
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b00000100: begin // or word immediate
        for(i = 0; i < 4; i++) result[(i * 32) +:32] = data_ra[(i * 32) +:32] |
{{22{immediate[1]}}, immediate[1:10]};
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01000110: begin // xor byte immediate
        for(i = 0; i < 16; i++) result[(i * 8) +:8] = data_ra[(i * 8) +:8] ^ (immediate[1:10]
& 16'h00ff);

        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01000101: begin // xor halfword immediate
        for(i = 0; i < 8; i++) begin
            result[(i * 16) +:16] = data_ra[(i * 16) +:16] ^ {{6{immediate[1]}},
immediate[1:10]};

        end
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01000100: begin // xor word immediate
        for(i = 0; i < 4; i++) result[(i * 32) +:32] = data_ra[(i * 32) +:32] ^
{{22{immediate[1]}}, immediate[1:10]};

```



```

        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01111110: begin // Compare Equal Byte immediate
        for(i = 0; i < 16; i++) result[(i * 8) +:8] = (data_ra[(i * 8) +:8] ==
immediate[3:10]) ? 8'hff : 8'h00;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01111101: begin // Compare Equal Halfword immediate
        for(i = 0; i < 8; i++) result[(i * 16) +:16] = (data_ra[(i * 16) +:16] ==
{{6{immediate[1]}}, immediate[1:10]}) ? 16'hffff : 16'h0000;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01111100: begin // Compare Equal Word immediate
        for(i = 0; i < 4; i++) result[(i * 32) +:32] = (data_ra[(i * 32) +:32] ==
{{22{immediate[1]}}, immediate[1:10]}) ? 32'hffff_ffff : 32'h0000_0000;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01001110: begin // Compare Greater than Byte immediate
        for(i = 0; i < 16; i++) result[(i * 8) +:8] = ($signed(data_ra[(i * 8) +:8]) >
immediate[3:10]) ? 8'hff : 8'h00;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01001101: begin // Compare Greater than Halfword immediate
        for(i = 0; i < 8; i++) result[(i * 16) +:16] = ($signed(data_ra[(i * 16) +:16]) >
{{6{immediate[1]}}, immediate[1:10]}) ? 16'hffff : 16'h0000;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01001100: begin // Compare Greater than Word immediate
        for(i = 0; i < 4; i++) result[(i * 32) +:32] = ($signed(data_ra[(i * 32) +:32]) >
{{22{immediate[1]}}, immediate[1:10]}) ? 32'hffff_ffff : 32'h0000_0000;
        result[128:130] = 3'd2;
        result[131] = 1'b1;
    end
    8'b01011110: begin // Compare Logical Greater than Byte immediate
        for(i = 0; i < 16; i++) result[(i * 8) +:8] = (data_ra[(i * 8) +:8] > immediate[3:10])
? 8'hff : 8'h00;

        result[128:130] = 3'd2;
        result[131] = 1'b1;

```

```

        end
        8'b01011101: begin // Compare Logical Greater than Halfword immediate
            for(i = 0; i < 8; i++) result[(i * 16) +:16] = (data_ra[(i * 16) +:16] >
{{6{immediate[1]}}, immediate[1:10]}) ? 16'hffff : 16'h0000;
            result[128:130] = 3'd2;
            result[131] = 1'b1;
        end
        8'b01011100: begin // Compare Logical Greater then Word immediate
            for(i = 0; i < 4; i++) result[(i * 32) +:32] = (data_ra[(i * 32) +:32] >
{{22{immediate[1]}}, immediate[1:10]}) ? 32'hffff_ffff : 32'h0000_0000;
            result[128:130] = 3'd2;
            result[131] = 1'b1;
        end
        default:begin // RI18 format - Immediate load Address with 7 bit opcode

            case(opcode[0:6])
                7'b0100001: begin
                    for(i = 0; i < 4; i++) result[(i * 32) +:32] = {{14'd0}, immediate};
                    result[128:130] = 3'd2;
                    result[131] = 1'b1;
                end
                default: result = 0;
            endcase
        end
    endcase

end

endcase

end
endmodule

```

Simple Fixed 2 Unit

/* Module for Simple Fixed 2 unit containing all instruction coded

* Imagine all the calculations for all the instructions would be done in the first cycle and then just shifted through

* Things to Note:-

* 1. "result", "result_reg" and "result_reg_1" are internal logic since the pipeline depth is 2. "out_data" is output going to

* forwarding registers.

*/

```

module SimpleFixed2(
    input          clk, reset, flush,
    input [0:127]  data_ra, data_rb,
    input [0:6]    immediate,
    input [0:6]    addr_rt,
    input [0:10]   opcode,
    output logic [0:138] out_data);

    logic [0:138] result, result_reg, result_reg_1;
    logic [0:15]  temp16, temp16_reg;
    logic [0:31]  temp32, temp32_reg;

    always_ff @(posedge clk) begin
        if(reset) begin
            result_reg <= 0;
            result_reg_1 <= 0;
            out_data <= 0;
        end
        else begin
            result_reg <= result;
            result_reg_1 <= (flush) ? 0 : result_reg;
            out_data <= result_reg_1;
        end
    end

    always_comb begin
        integer i, j;
        result[128:130] = 3'd4;
        result[131] = 1'b1;
        result[132:138] = addr_rt;
        case(opcode)
            // RR type instructions
            11'b000010111111: begin // Shift left halfword
                for(i = 0; i < 8 ; i++) begin
                    temp16 = data_rb[(i * 16) +:16] & 16'h001f;
                    temp16_reg = data_ra[(i * 16) +:16];
                    result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg << temp16 : 16'h0000;
                end
            end
            11'b000010110111: begin // Shift left word
                for(i = 0; i < 4 ; i++) begin
                    temp32 = data_rb[(i * 32) +:32] & 32'h0000_003f;
                    temp32_reg = data_ra[(i * 32) +:32];
                    result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg << temp32 : 32'h0000_0000;
                end
            end
        endcase
    end
end

```

```

    end
end
11'b00001011100: begin // Rotate halfword
    for(i = 0; i < 8 ; i++) begin
        temp16 = data_rb[(i * 16) +:16] & 16'h000f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16_reg << temp16) | (temp16_reg >> (16 - temp16));
    end
end
11'b00001011000: begin // Rotate word
    for(i = 0; i < 4 ; i++) begin
        temp32 = data_rb[(i * 32) +:32] & 32'h0000_001f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32_reg << temp32) | (temp32_reg >> (32 - temp32));
    end
end
11'b00001011101: begin // Rotate and mask halfword
    for(i = 0; i < 8 ; i++) begin
        temp16 = (0 - data_rb[(i * 16) +:16]) & 16'h001f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg >> temp16 : 16'h0000;
    end
end
11'b00001011001: begin // Rotate and mask word
    for(i = 0; i < 4 ; i++) begin
        temp32 = (0 - data_rb[(i * 32) +:32]) & 32'h0000_003f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg >> temp32 : 32'h0000_0000;
    end
end
11'b00001011110: begin // Rotate and mask algebraic halfword
    for(i = 0; i < 8 ; i++) begin
        temp16 = (0 - data_rb[(i * 16) +:16]) & 16'h001f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg >>> temp16 : 16'h0000;
    end
end
11'b00001011010: begin // Rotate an mask algebraic word
    for(i = 0; i < 4 ; i++) begin
        temp32 = data_rb[(i * 32) +:32] & 32'h0000_003f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg >>> temp32 : 32'h0000_0000;
    end
end

```

```

// RI7 type instructions
11'b00001111111: begin // Shift left halfword immediate
    for(i = 0; i < 8 ; i++) begin
        temp16 = {{9{immediate[0]}}, immediate} & 16'h001f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg << temp16 : 16'h0000;
    end
end
11'b00001111011: begin // Shift left word immediate
    for(i = 0; i < 4 ; i++) begin
        temp32 = {{25{immediate[0]}}, immediate} & 32'h0000_003f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg << temp32 : 32'h0000_0000;
    end
end
11'b00001111100: begin // Rotate halfword immediate
    for(i = 0; i < 8 ; i++) begin
        temp16 = {{9{immediate[0]}}, immediate} & 16'h000f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16_reg << temp16) | (temp16_reg >> (16 - temp16));;
    end
end
11'b00001111000: begin // Rotate word immediate
    for(i = 0; i < 4 ; i++) begin
        temp32 = {{25{immediate[0]}}, immediate} & 32'h0000_001f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32_reg << temp32) | (temp32_reg >> (32 - temp32));;
    end
end
11'b00001111101: begin // Rotate and mask halfword immediate
    for(i = 0; i < 8 ; i++) begin
        temp16 = (0 - {{9{immediate[0]}}, immediate}) & 16'h001f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg >> temp16 : 16'h0000;
    end
end
11'b00001111001: begin // Rotate and mask word immediate
    for(i = 0; i < 4 ; i++) begin
        temp32 = (0 - {{25{immediate[0]}}, immediate}) & 32'h0000_003f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg >> temp32 : 32'h0000_0000;
    end
end
11'b00001111110: begin // Rotate and mask algebraic halfword immediate

```

```

    for(i = 0; i < 8 ; i++) begin
        temp16 = (0 - {{9{immediate[0]}}, immediate}) & 16'h001f;
        temp16_reg = data_ra[(i * 16) +:16];
        result[(i * 16) +:16] = (temp16 < 16) ? temp16_reg >>> temp16 : 16'h0000;
    end
end
11'b00001111010: begin // Rotate and mask algebraic word immediate
    for(i = 0; i < 4 ; i++) begin
        temp32 = (0 - {{25{immediate[0]}}, immediate}) & 32'h0000_003f;
        temp32_reg = data_ra[(i * 32) +:32];
        result[(i * 32) +:32] = (temp32 < 32) ? temp32_reg >>> temp32 : 32'h0000_0000;
    end
end
default: result = 0;
endcase
end

endmodule

```

Single Precision Unit

```
module singlePrec(clk, ra, rb, rc, opcode11, opcode10, opcode8, opcode4, immediate10, immediate8,  
pipe6out, pipe7out, addr_rt, reset, flush);
```

```
    input                clk, reset, flush;  
    input [0:127]        ra, rb, rc;  
    output logic [0:138]  pipe6out, pipe7out;  
    logic [0:138]        pipe1, pipe2, pipe3, pipe4, pipe5, pipe6, result;  
    input [0:10]         opcode11;  
    input [0:9]          opcode10;  
    input [0:7]          opcode8;  
    input [0:3]          opcode4;  
    input [0:9]          immediate10;  
    logic [0:7]          scale1, scale2;  
    input [0:7]          immediate8;  
    input [0:6]          addr_rt;  
    integer              i;
```

```
always_ff @(posedge clk) begin  
    if(reset)begin  
        pipe1 <= 0;  
        pipe2 <= 0;  
        pipe3 <= 0;  
        pipe4 <= 0;  
        pipe5 <= 0;  
        pipe6 <= 0;  
        pipe6out <= 0;  
        pipe7out <= 0;  
    end else begin  
        pipe1 <= result;  
        pipe2 <= (flush) ? 0 : pipe1;  
        pipe3 <= pipe2;  
        pipe4 <= pipe3;  
        pipe5 <= pipe4;  
        if(pipe5[128:130] == 3'd6)  
            pipe6out <= pipe5;  
        else  
            pipe6out <= 0;  
        pipe6 <= pipe5;  
        if(pipe6[128:130] == 3'd7)  
            pipe7out <= pipe6;  
        else  
            pipe7out <= 0;  
    end  
end
```

```

end

always_comb begin
    scale1 = 155 - immediate8;
    scale2 = 173 - immediate8;

    result[128:130] = 3'd6;
    result[131] = 1'b1;
    result[132:138] = addr_rt;

    case(opcode11)
        11'b01111000100:begin//multiply                                good
            for (i=0; i<4; i=i+1) result[i*32 +:32] = $signed(ra[((2*i)+1)*16 +:16]) *
$signed(rb[((2*i)+1)*16 +:16]);
            result[128:130] = 3'd7;
            end
        11'b01111001100:begin//multiply unsigned                        good
            for (i=0; i<4; i=i+1) result[i*32 +:32] = ra[((2*i)+1)*16 +:16] * rb[((2*i)+1)*16 +:16];
            result[128:130] = 3'd7;
            end
        11'b01111000101:begin//multiply high                            good
            for (i=0; i<4; i=i+1) result[i*32 +:32] = ($signed(ra[2*i*16 +:16]) *
$signed(rb[((2*i)+1)*16 +:16])) << 16;
            result[128:130] = 3'd7;
            end
        11'b01011000100://floating add
            for (i=0; i<4; i=i+1) result[i*32 +:32] = shortreal'(ra[i*32 +:32]) + shortreal'(rb[i*32
+:32]);
        11'b01011000101://floating subtract
            for (i=0; i<4; i=i+1) result[i*32 +:32] = shortreal'(ra[i*32 +:32]) - shortreal'(rb[i*32
+:32]);
        11'b01011000110://floating mult
            for (i=0; i<4; i=i+1) result[i*32 +:32] = shortreal'(ra[i*32 +:32]) * shortreal'(rb[i*32
+:32]);
        11'b01111000010:begin//Floating Compare Equal
            for (i=0; i<4; i=i+1) begin
                if(shortreal'(ra[i*32 +:32]) == shortreal'(ra[i*32 +:32]))
                    result[i*32 +:32] = 32'hFFFFFFFF;
                else
                    result[i*32 +:32] = 32'h00000000;
            end
        end
        11'b01111001010:begin//Floating Compare Magnitude Equal

```



```

        for (i=0; i<4; i=i+1) begin
            if(shortreal'(ra[i*32 +:31]) == shortreal'(rb[i*32 +:31]))
                result[i*32 +:32] = 32'hFFFFFFFF;
            else
                result[i*32 +:32] = 32'h00000000;
            end
        end
11'b01011000010:begin//Floating Compare Greater Than
    for (i=0; i<4; i=i+1) begin
        if(shortreal'(ra[i*32 +:32]) > shortreal'(rb[i*32 +:32]))
            result[i*32 +:32] = 32'hFFFFFFFF;
        else
            result[i*32 +:32] = 32'h00000000;
        end
    end
11'b01011001010:begin//Floating Compare Magnitude Greater Than
    for (i=0; i<4; i=i+1) begin
        if(shortreal'(ra[i*32 +:31]) > shortreal'(rb[i*32 +:31]))
            result[i*32 +:32] = 32'hFFFFFFFF;
        else
            result[i*32 +:32] = 32'h00000000;
        end
    end

    default:begin
        case(opcode10)
            10'b0111011010://Convert Signed Integer to Floating    //need to add some
kind of saturation
                for (i=0; i<4; i=i+1) begin
                    if($signed(scale1) < 0)
                        result[i*32 +:32] = 32'hxxxxxxxx;
                    else
                        result[i*32 +:32] = shortreal'($signed(ra[i*32 +:32])) / shortreal'(2 **
scale1);
                    end
                10'b0111011000://Convert Floating to Signed Integer    //need to add some
kind of saturation
                for (i=0; i<4; i=i+1) begin
                    if($signed(scale2) < 0)
                        result[i*32 +:32] = 32'hxxxxxxxx;
                    else
                        result[i*32 +:32] = $signed(int'(shortreal'(ra[i*32 +:32]) * shortreal'(2
** scale2)));
                    end
        end
    end
end

```

```

10'b0111011011://Convert Unsigned Integer to Floating    //need to add some
kind of saturation
    for (i=0; i<4; i=i+1) begin
        if($signed(scale1) < 0)
            result[i*32+:32] = 32'hxxxxxxxx;
        else
            result[i*32+:32] = shortreal'(ra[i*32+:32]) / shortreal'(2 ** scale1);
        end
10'b0111011001://Convert Floating to Unsigned Integer    //need to add some
kind of saturation
    for (i=0; i<4; i=i+1) begin
        if($signed(scale2) < 0)
            result[i*32+:32] = 32'hxxxxxxxx;
        else
            result[i*32+:32] = int'(shortreal'(ra[i*32+:32]) * shortreal'(2 **
scale2));
        end
default:begin
    case(opcode8)
        8'b01110100:begin//multiply immediate    good
            for (i=0; i<4; i=i+1) result[i*32+:32] =
$signed(ra[((2*i)+1)*16+:16]) * $signed(immediate10);
            result[128:130] = 3'd7;
            end
        8'b01110101:begin//multiply immediate    good
            for (i=0; i<4; i=i+1) result[i*32+:32] =
$signed(ra[((2*i)+1)*16+:16]) * $signed(immediate10);
            result[128:130] = 3'd7;
            end
        default:begin case(opcode4)
            4'b1100:begin//multiply add
                for (i=0; i<4; i=i+1) result[i*32
+:32] = ($signed(ra[((2*i)+1)*16+:16]) * $signed(rb[((2*i)+1)*16+:16])) + $signed(rc[i*32+:32]);
                result[128:130] = 3'd7;
                end
            4'b1110://floating multiply add
                //need to add some kind of saturation
                for (i=0; i<4; i=i+1) result[i*32
+:32] = (shortreal'(ra[i*32+:32]) * shortreal'(rb[i*32+:32])) + shortreal'(rc[i*32+:32]);
            4'b1111://floating multiply subtract
                //need to add some kind of saturation
                for (i=0; i<4; i=i+1) result[i*32
+:32] = (shortreal'(ra[i*32+:32]) * shortreal'(rb[i*32+:32])) - shortreal'(rc[i*32+:32]);
            default:

```

```
result = 0;
endcase
end
endcase
end
endcase
end
endcase
end
endmodule
```

Branch Unit

```
module branch(clk, rt, reset, addr_rt, opcode9, pipe, PCin, predictPCin, predictIn, PCpipe, immediate16);
    input                clk, reset, predictIn;
    input [0:31]         PCin, predictPCin;
    input [0:127]        rt;
    input [0:6]          addr_rt;
    output logic [0:138]  pipe;
    output logic [0:65]   PCpipe;
    input [0:8]          opcode9;
    input [0:15]         immediate16;
    logic [0:138]        result;
    logic [0:65]         PCout;

    always_ff @(posedge clk) begin
        if(reset) begin
            pipe <= 0;
            PCpipe <= 0;
        end else begin
            pipe <= result;
            PCpipe <= PCout;
        end
    end

    always_comb begin
        result[128:130] = 3'd1;
        result[131] = 1'b1;
        result[132:138] = addr_rt;

        PCout[32] = 1;
        PCout[33:64] = PCin;
        PCout[65] = 0;

        case(opcode9)
            9'b001100100:begin//branch relative GOOD
                result = 0;//need to assign a value
                PCout[0:31] = PCin + $signed({{14{immediate16[0]}}, immediate16, 2'b00});
                PCout[65] = (predictIn == 0 | (predictPCin != PCout[0:31])) ? 1 : 0;
            end
            9'b001100110:begin//branch relative and set link
                result[32:127] = 0;
                result[0:31] = PCin + 4;
                PCout[0:31] = PCin + $signed({{14{immediate16[0]}}, immediate16, 2'b00});
            end
        endcase
    end
end
```

```

        PCout[65] = (predictIn == 0 | (predictPCin != PCout[0:31])) ? 1 : 0;
    end
    9'b001000000:begin//branch if zero word GOOD
        if(rt[0:31] == 0)
            PCout[0:31] = PCin + $signed({{14{immediate16[0]}},{immediate16,
2'b00}});
        else begin
            PCout[0:31] = (PCin + 8) &
(32'b1111_1111_1111_1111_1111_1111_1000);
            PCout[32] = 0;
        end
        PCout[65] = ((predictIn != PCout[32]) | (predictPCin != PCout[0:31])) ? 1 : 0;
        result = 0;//must assign value
    end
    9'b001000010:begin//branch if not zero word GOOD
        if(rt[0:31] != 0)
            PCout[0:31] = PCin + $signed({{14{immediate16[0]}},{immediate16,
2'b00}});
        else begin
            PCout[0:31] = (PCin + 8) &
(32'b1111_1111_1111_1111_1111_1111_1000);
            PCout[32] = 0;
        end
        PCout[65] = ((predictIn != PCout[32]) | (predictPCin != PCout[0:31])) ? 1 : 0;
        result = 0;//must assign value
    end
    default: begin
        result = 0;
        PCout = 0;
    end
end
endcase
end
endmodule

```

Permute Unit

```
module permute(clk, ra, rb, opcode11, immediate7, WBpipe3, addr_rt, reset);
    input                clk, reset;
    input [0:6]          addr_rt;
    input [0:127]        ra, rb;
    output logic [0:138]  WBpipe3;
    logic [0:138]        result, pipe1, pipe2;
    input [0:10]          opcode11;
    input [0:6]           immediate7;
    integer i;

    always_ff @(posedge clk) begin
        if(reset) begin
            pipe1 <= 0;
            pipe2 <= 0;
            WBpipe3 <= 0;
        end else begin
            pipe1 <= result;
            pipe2 <= pipe1;
            WBpipe3 <= pipe2;
        end
    end

    always_comb begin

        result[128:130] = 3'd3;
        result[131] = 1'b1;
        result[132:138] = addr_rt;

        case(opcode11)
            11'b00110110010:begin//gather bits from bytes GOOD
                result[0:15] = 0;
                result[32:127] = 0;
                result[16:31] = {ra[7], ra[15], ra[23], ra[31], ra[39], ra[47], ra[55], ra[63], ra[71],
ra[79], ra[87], ra[95], ra[103], ra[111], ra[119], ra[127]};
            end
            11'b00110110000:begin//gather bits from word GOOD
                result[0:27] = 0;
                result[32:127] = 0;
                result[28:31] = {ra[31], ra[63], ra[95], ra[127]};
            end
            11'b00111011111://shift left quadword by bytes GOOD
                if(rb[27:31] > 15)

```

```

        result[0:127] = 0;
    else
        result[0:127] = ra << 8*rb[27:31];
11'b0011111111111111://shift left quadword by bytes immediate GOOD
        if(immediate7[2:6] > 15)
            result[0:127] = 0;
        else
            result[0:127] = ra << 8*immediate7[2:6];
11'b00111011100://rotate quadword by bytes GOOD
            result[0:127] = (ra << 8*rb[28:31]) | (ra >> 8*(16-rb[28:31]));
11'b001111111111100://rotate quadword by bytes immediate GOOD
            result[0:127] = (ra << 8*immediate7[2:6]) | (ra >> 8*(16-immediate7[2:6]));
11'b00111011101://rotate and mask quadword by bytes GOOD
            if((0-rb[0:31])%32 < 16)
                result[0:127] = ra >> ((0-rb[0:31])%32)*8;
            else
                result[0:127] = 0;
11'b001111111111101://rotate and mask quadword by bytes immediate GOOD
            if((0-immediate7)%32 < 16)
                result[0:127] = ra >> ((0-immediate7)%32)*8;
            else
                result[0:127] = 0;
    default:
        result = 0;

endcase

end
endmodule

```

Localstore Unit

```
module localstore(clk, ra, rt, addr_rt, opcode9, opcode8, immediate16, immediate10, FWpipe6, PCin, reset);
    input                clk, reset;
    input [0:6]          addr_rt;
    input [0:31]         PCin;
    input [0:127]        ra, rt;
    output logic [0:138]  FWpipe6;
    logic [0:1]          wr_en, wr_en_pipe1;
    logic [0:127]        rt_pipe;
    logic [0:138]        memOut, pipe1, pipe2, pipe3, pipe4, pipe5, result;
    logic [0:31]         memAddr, memAddr_pipe1;
    input [0:8]          opcode9;
    input [0:7]          opcode8;
    input [0:15]         immediate16;
    input [0:9]          immediate10;
    // Double Dimension array should suffice as memory, do not require another module
    logic [0:255][0:7] mem;
    // All memory reads and writes will be in always_ff block
    always_ff @(posedge clk) begin
        integer i;
        if(wr_en==2) begin
            for(i = 0; i < 16; i++) mem[memAddr[24:31] + i] <= rt[(i * 8) +:8];
            result <= 0;
        end
        else begin if(wr_en==1) begin
            if(wr_en_pipe1 == 2 & memAddr_pipe1 == memAddr) result[0:127] <= rt_pipe;
            else
                for(i = 0; i < 16; i++) result[(i * 8) +:8] <= mem[memAddr[24:31] + i];
            result[128:130] <= 3'd6;
            result[131] <= 1'b1;
            result[132:138] <= addr_rt;
        end else
            result <= 0;
        end

        if(reset)begin
            pipe2 <= 0;
            pipe3 <= 0;
            pipe4 <= 0;
            pipe5 <= 0;
            FWpipe6 <= 0;
            wr_en_pipe1 <= 0;
            memAddr_pipe1 <= 0;
        end
    end
endmodule
```



```

        rt_pipe <= 0;
        for (i=0; i<64; i=i+1) begin
            mem[4*i] <= 0;
            mem[(4*i)+1] <= 0;
            mem[(4*i)+2] <= 0;
            mem[(4*i)+3] <= i;
        end
    end else begin
        pipe2 <= result;
        pipe3 <= pipe2;
        pipe4 <= pipe3;
        pipe5 <= pipe4;
        FWpipe6 <= pipe5;
        wr_en_pipe1 <= wr_en;
        memAddr_pipe1 <= memAddr;
        rt_pipe <= rt;
    end
end

always_comb begin

    case(opcode9)
        9'b001100001:begin//load quad word A-form
            memAddr = {{14{immediate16[0]}},{immediate16, 2'b00}} & 32'hFFFFFFF0;
            wr_en = 1;
        end
        9'b001100111:begin//load quad word instruction relative A-form
            memAddr = (PCin + {immediate16, 2'b00}) & 32'hFFFFFFF0;
            wr_en = 1;
        end
        9'b001000001:begin//store quad word A-form
            memAddr = {{14{immediate16[0]}},{immediate16, 2'b00}} & 32'hFFFFFFF0;
            wr_en = 2;
        end
        9'b001000111:begin//store quad word instruction relative A-form
            memAddr = (PCin + {{14{immediate16[0]}},{immediate16, 2'b00}}) &
32'hFFFFFFF0;
            wr_en = 2;
        end
        default:begin case(opcode8)
            8'b00110100:begin//load quad word D-form
                memAddr = ({18{immediate10[0]}}, {immediate10, 4'b0000}) +
ra[0:31]) & 32'hFFFFFFF0;
                wr_en = 1;
            end
        end
    end
end

```

```

                                end
                        8'b00100100:begin//store quad word D-form
                                memAddr = ({18{immediate10[0]}}, {immediate10, 4'b0000}) +
ra[0:31]) & 32'hFFFFFFF0;
                                wr_en = 2;
                                end
                        default: wr_en = 0;//output 0 when unit not called
                endcase
        end
    endcase
end
endmodule

```

Parser Code

```
from bitstring import Bits
if __name__ == "__main__":
    inFile = "commandFormats.txt"
    inFile2 = "commands.txt"
    inFile3 = "ops.csv"
    outFile = "instructions.txt"

    commandDict={
        'lqd': ['rt', 'symbol(ra)'], 'lqa': ['rt', 'symbol'], 'lqr': ['rt', 'symbol'],
        'stqd': ['rt', 'symbol(ra)'], 'stqa': ['rt', 'symbol'], 'stqr': ['rt', 'symbol'],
        'ilh': ['rt', 'symbol'], 'ilhu': ['rt', 'symbol'], 'iohl': ['rt', 'symbol'],
        'il': ['rt', 'symbol'], 'ila': ['rt', 'symbol'], 'fsmbi': ['rt', 'symbol'],
        'ah': ['rt', 'ra', 'rb'], 'ahi': ['rt', 'ra', 'value'], 'a': ['rt', 'ra', 'rb'],
        'ai': ['rt', 'ra', 'value'], 'sfh': ['rt', 'ra', 'rb'], 'sfhi': ['rt', 'ra', 'value'],
        'sf': ['rt', 'ra', 'rb'], 'sfi': ['rt', 'ra', 'value'], 'mpy': ['rt', 'ra', 'rb'],
        'mpyu': ['rt', 'ra', 'rb'], 'mypi': ['rt', 'ra', 'value'], 'mpyui': ['rt', 'ra', 'value'],
        'mpya': ['rt', 'ra', 'rb', 'rc'], 'mpyh': ['rt', 'ra', 'rb'], 'clz': ['rt', 'ra'],
        'cntb': ['rt', 'ra'], 'fsmb': ['rt', 'ra'], 'fsmh': ['rt', 'ra'], 'fsm': ['rt', 'ra'],
        'gbb': ['rt', 'ra'], 'gb': ['rt', 'ra'], 'avgb': ['rt', 'ra', 'rb'],
        'absdb': ['rt', 'ra', 'rb'], 'sumb': ['rt', 'ra', 'rb'], 'and': ['rt', 'ra', 'rb'],
        'andbi': ['rt', 'ra', 'rb'], 'andi': ['rt', 'ra', 'rb'], 'or': ['rt', 'ra', 'rb'],
        'orbi': ['rt', 'ra', 'value'], 'ori': ['rt', 'ra', 'value'], 'orx': ['rt', 'ra'],
        'xor': ['rt', 'ra', 'rb'], 'xorbi': ['rt', 'ra', 'value'], 'xorhi': ['rt', 'ra', 'value'],
        'xori': ['rt', 'ra', 'value'], 'shlh': ['rt', 'ra', 'rb'], 'shlhi': ['rt', 'ra', 'value'],
        'shl': ['rt', 'ra', 'rb'], 'shli': ['rt', 'ra', 'value'], 'shlqby': ['rt', 'ra', 'rb'],
        'shlqbyi': ['rt', 'ra', 'value'], 'roth': ['rt', 'ra', 'rb'], 'rothi': ['rt', 'ra', 'value'],
        'rot': ['rt', 'ra', 'rb'], 'roti': ['rt', 'ra', 'value'], 'rotqby': ['rt', 'ra', 'rb'],
        'rotqbyi': ['rt', 'ra', 'value'], 'rothm': ['rt', 'ra', 'rb'], 'rotm': ['rt', 'ra', 'rb'],
        'rotmi': ['rt', 'ra', 'value'], 'rotqmbi': ['rt', 'ra', 'rb'],
        'rotqmbyi': ['rt', 'ra', 'value'], 'rotmah': ['rt', 'ra', 'rb'],
        'rotmahi': ['rt', 'ra', 'value'], 'rotma': ['rt', 'ra', 'rb'],
        'rotmai': ['rt', 'ra', 'value'], 'ceqb': ['rt', 'ra', 'rb'],
        'ceqbi': ['rt', 'ra', 'value'], 'ceqh': ['rt', 'ra', 'rb'],
        'ceqhi': ['rt', 'ra', 'value'], 'ceq': ['rt', 'ra', 'rb'],
        'ceqi': ['rt', 'ra', 'value'], 'cgtb': ['rt', 'ra', 'rb'],
        'cgtbi': ['rt', 'ra', 'value'], 'cgth': ['rt', 'ra', 'rb'],
        'cgthi': ['rt', 'ra', 'value'], 'cgt': ['rt', 'ra', 'rb'],
        'cgti': ['rt', 'ra', 'value'], 'clgtb': ['rt', 'ra', 'rb'],
        'clgtbi': ['rt', 'ra', 'value'], 'clgth': ['rt', 'ra', 'rb'],
        'clgthi': ['rt', 'ra', 'value'], 'clgt': ['rt', 'ra', 'rb'], 'clgti': ['rt', 'ra', 'value'],
        'br': ['symbol'], 'brsl': ['rt', 'symbol'], 'brz': ['rt', 'symbol'],
        'brnz': ['rt', 'symbol'], 'fa': ['rt', 'ra', 'rb'], 'fs': ['rt', 'ra', 'rb'],
        'fm': ['rt', 'ra', 'rb'], 'fma': ['rt', 'ra', 'rb', 'rc'], 'fms': ['rt', 'ra', 'rb', 'rc'],
```

```

'csflt': ['rt', 'ra', 'scale'], 'cflts': ['rt', 'ra', 'scale'],
'cufilt': ['rt', 'ra', 'scale'], 'cfltu': ['rt', 'ra', 'scale'], 'fceq': ['rt', 'ra', 'rb'],
'fcmeq': ['rt', 'ra', 'rb'], 'fcgt': ['rt', 'ra', 'rb'], 'fcmgt': ['rt', 'ra', 'rb'],
'stop': [], 'lnop': [], 'nop': []}

```

```

opsDict={}
fstream1 = open(inFile3, "r")
for line in fstream1:
    the_line = line.strip("\n").strip(' ').split(" ")
    opsDict[the_line[0]]=the_line[1]
#print(opsDict)
fstream1.close()

fstream1 = open(inFile2, "r")
outF = open(outFile, "w")
fillzero='0000000'
for line in fstream1:
    bits0_6=None
    bits7_13=None
    bits14_20=None
    bits21_27=None
    value=None#7 or 10 bits depending on opcode size (11bit op or 8bit op respectively) after op
    symbol=None#7, 10, 16 or 18 bits depending on opcode size and instance of ra
    scale=None#always 8bit after op
    opcode=None#4 to 11 bits

    count=0
    the_line = line.strip("\n").replace('$', "").split(" ")
    for elem in the_line:
        elem=elem.replace(',', '')
        the_line[count]=elem
        count+=1
    #print(the_line)
    cmdFormat=commandDict[the_line[0]]
    #print(cmdFormat)
    if(the_line[0] in opsDict):
        opcode=opsDict[the_line[0]]

    #fills in fields appropriate for the command
    if('rc' in cmdFormat):
        bits0_6=Bits(int=int(the_line[cmdFormat.index('rc')+1]), length=7).bin
        bits21_27=Bits(int=int(the_line[cmdFormat.index('rt')+1]), length=7).bin
    elif('rt' in cmdFormat):
        bits0_6=Bits(int=int(the_line[cmdFormat.index('rt')+1]), length=7).bin

```

```

if('ra' in cmdFormat):
    bits7_13=Bits(int=int(the_line[cmdFormat.index('ra')+1]), length=7).bin
if('rb' in cmdFormat):
    bits14_20=Bits(int=int(the_line[cmdFormat.index('rb')+1]), length=7).bin
if('value' in cmdFormat):
    if(len(opcode)==11):
        immSize=7
    elif(len(opcode)==8):
        immSize=10
    value=Bits(int=int(the_line[cmdFormat.index('value')+1]), length=immSize).bin
if('scale' in cmdFormat):
    scale=Bits(int=int(the_line[cmdFormat.index('scale')+1]), length=8).bin
if('symbol' in cmdFormat):
    if(bits7_13!=None):
        if(len(opcode)==8):
            symSize=10
        elif(len(opcode)==11):
            symSize=7
    else:
        if(len(opcode)==7):
            symSize=18
        elif(len(opcode)==9):
            symSize=16
    symbol=Bits(int=int(the_line[cmdFormat.index('symbol')+1]), length=symSize).bin
if('symbol(ra)' in cmdFormat):
    symRa=the_line[cmdFormat.index('symbol(ra)')+1].strip(' ').split(' ')
    sym=symRa[0]
    ra=symRa[1]
    symbol=Bits(int=int(sym), length=10).bin
    bits7_13=Bits(int=int(ra), length=7).bin

#appends none empty fields in order
allbits=[]
allbits.append(opcode)
if(bits21_27!=None):
    allbits.append(bits21_27)#rc/rt
if(bits14_20!=None):
    allbits.append(bits14_20)#or rb
if(value!=None):
    allbits.append(value)#or imm
if(scale!=None):
    allbits.append(scale)#or imm
if(symbol!=None):
    allbits.append(symbol)#or imm

```

```

if(len(allbits)<2):
    allbits.append(fillzero)#fills dont care values if nothing has been appended
if(bits7_13!=None):
    allbits.append(bits7_13)#ra
if(bits0_6!=None):
    allbits.append(bits0_6)#rt/doesn't matter
if(len(allbits)<3):
    allbits.append(fillzero)#fills dont care values if nothing has been appended

#need case if stop, nop, or Inop
if(the_line[0]=='nop' or the_line[0]=='Inop' or the_line[0]=='stop'):
    allbits=[]
    allbits.append(opcode)
    while(len(allbits)<4):
        allbits.append(fillzero)

#print(allbits)
#print(opcode)
outString=""
for bits in allbits:
    outString=outString+bits
#print(outString)
outF.write(outString)
outF.write("\n")
fstream1.close()
outF.close()

```