Project title: Dual-Issue Cell SPU-Lite Multimedia Processor

Team members: Ahib Alghazali (111705816) Prince Kumar Maurya (114354075)

	SPU-lite Model Code	Parser code	Loading of the instruction memory with instruction code from the file created by the parser	Dual-instruction fetch_decode_RF read_ execute (7 stages)_WB (no hazards)	Structural hazard resolution	Data hazard resolution by forwarding (no stall)	Data hazard resolution by stalling & forwarding	Control hazard resolution for branches	4x4 SP FP matrix multiply using multiply-add instructions (as a reference, look at the code for the 2x2 DP FP matrix multiply for Intel SSE in the lecture on DLP and SIMD) Initialize elements of the first matrix with values like 0.0, 1.0, 2.0, 3.0, and so on, and the second matrix with values of 16.0, 17.0, 18.0, 19.0 and so on. Both matrices need to be in memory. **Hint: Initialize these matrices at the locations starting from address 0 in the Local Store before the start of simulation <u>without any use of Cell SPU instructions.</u>**
Page #s for Code & Testing Results	32	99	33	13 (Sample Code 107)	11 (Sample Code 107)	13 (Sample Code 107)	12 (Sample Code 107)	12 (Sample Code 107)	14 (Sample Code 108)

Comments by Design Team		Written in Python as assembler.py, with instruction formats in following .lst file. Writes to “out” file shown in sample programs section.	tb_Instruction Fetch handles reading from “out” output from parser. PC aligns to each instr as (line number - 1).	Lines 21-30 of input1.asm; Even and Odd instructions issue in pairs if there are no conflicts between them and both instr of pair issue to different pipes.	Lines 1-15 of input1.asm; If two instr of pair issue to the same pipe, the first of the two will issue and the second will stall for 1 cycle.	Lines 21-30 of input1.asm, emphasis on 21 & 30; If consumer instr is dependent on output of producer instr and reaches decode on (latency - 1) cycle of prod., cons. will issue to RF/FWD and read forwarded output value from (latency) cyc of cons.	Lines 18-20 of input1.asm; If consumer instr is dependent on output of producer instr before prod. can reach (latency - 1), cons. will stall until prod. reaches said cycle, and then issue on (latency) cyc following rules of previous case.	Lines 18-21 of input1.asm; After branch taken, IF and Decode must both stall while PC is overwritten and new instr can be read from Instr Mem. All instr in pipe behind the branch must be flushed. If BR instr is first of pair, all instr in same stage will be flushed as well.	input_mp.asm Performs matrix-multiplication on vectors stored in localstore by masking and shifting/rotating row-major Matrix B to be column-major, thus multiplicable with row-major Matrix A. Branch instr are used to loop between iterations of masks and shifts and conserve program size at cost of some control hazard compensation. Once aligned, matrix-multiplication is performed, result Matrix C is shifted for output alignment and stored in LS mem.
Comments by Instructor									

Dual-Issue Cell SPU-Lite Multimedia Processor

Ahib Alghazali **Prince Kumar Maurya**

111705816 **114354075**

ESE 545: Computer Architecture

May 1st, 2022

Table of Contents

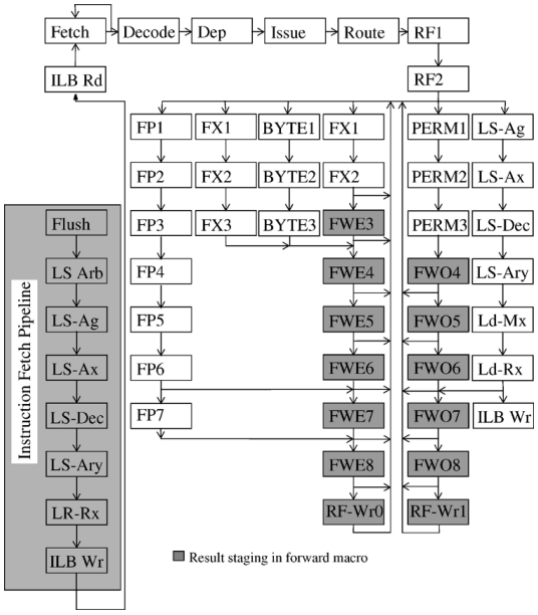
Introduction

The goal of this project is to learn about some of the complexities of modern microprocessor architecture, such as multiple-issue, multiple-input-multiple-data, and handling of data, structural, and control hazards. A key feature of the Sony Cell Synergistic Processing Unit (SPU) is that each processing element (SPE) is capable of issuing two instructions in a single cycle, but despite the fact that all instructions don't execute in the same length of time, it still has in-order behavior thanks to a forwarding pipeline that effectively idles either pipeline if its respective instruction has completed. The design of this project is a model-based simulation of the Cell SPE, capable of approximating most of the original unit's functionality and timing without needing to worry about intricacies such as instruction cache or the exact timing of memory accesses.

The design begins with a Python assembler that translates Cell-like assembly code into binary machine code that can be directly read by the unit. This code, from within the instruction memory, is read by the instruction fetch stage in pairs and then sent to the decode stage, which is responsible for sorting out structural and data hazards between these instructions. Compatible pairs of instructions are then dispatched to the register file/forwarding stage, where source values are read from registers or forwarded from later completed steps in the pipeline. These instructions and values are then sent to their respective dual-issue pipe to process a result. The execution, or even, pipe consists of four units focused more on arithmetic operations and the byte data type, while the load, or odd, pipe consists of three units specializing in memory access and program counter manipulation, as well as operating on quadword values. After completion and propagation through all forwarding stages, these results are written back to their respective registers, if necessary. Simplifications to the overall pipeline include fewer cycles through the first three single-pipe stages, reduced branch calculation time and penalty, static predict-not-taken branch prediction, considerably simpler instruction fetch, no use of data caches, and overall simpler memory management.

Data Flow

Figure 1: Original SPE Pipeline Diagram



The original Cell SPE (synergistic processing element) pipeline consisted of 16 stages that performed instructions in 16 cycles, from instruction fetch to register writeback (not accounting for the pipeline behind the instruction fetch itself). This pipeline is demonstrated in Figure 1. This modified Cell SPU-lite processor uses a simplified model of the hardware to perform the same instructions much faster and in a more straightforward manner, though without any concern for synthesis or real hardware operation. The simplified unit has 11 stages in its pipeline, also a single cycle each. The entire instruction fetch pipeline was shortened to a single stage, as were the Decode through Route stages. The register access was shortened to a single stage as well, while the eighth forwarding stage of the processing unit was combined with the register writeback stage.

The datapath of the SPU-lite unit starts by fetching two instructions in the first stage, then sending these instructions to the decode stage to handle decoding, checking for structural hazards (such as two instructions going to the same pipe) and data hazards (such as read-after-write and write-after-write), and issuing the pair of instructions, if possible. The instructions are issued to the register access stage, where the register table is read with the needed input values and data forwarding checks the two pipes for any more recent values ready to be used. Once updated, the values and relevant control signals are sent to the two pipes to be processed and forwarded to the final forwarding stage, which writes the results to their respective destination registers.

Instruction Fetch

The Instruction Fetch stage of the Cell SPU-lite consists of the instruction memory construct as well as the unit itself that reads the memory's values. The 1KB instruction memory is word-addressed, consisting of 256 (2^8-1) memory slots with 32-bit data per slot, each occupied by a single instruction. Each cycle, the IF unit reads two instructions from the instruction memory to then be processed by the Decode unit. The addresses used are based on the program counter, which typically increments by 2 every cycle, though it can also be manipulated by the Decode unit in case of a branch or stall. The unit uses static predict-not-taken branch prediction, so all branches will incur a branch miss penalty of about three cycles.

Decode

The Decode stage of the processor is responsible for decoding the inbound instructions from the Instruction Fetch unit and dealing with a variety of hazards associated with this process. It first will decode the operation, functional unit, pipe, and associated registers/values of the two instructions. It then will check the first instruction for any read-after-write (RAW) data hazards with existing instructions in the pipeline that haven't computed their results yet. Next, the unit will check the second instruction against the first in case they are both destined for the same instruction pipe (structural hazard) or if they would both write to the same destination register resulting in a write-after-write (WAW) data hazard. The same RAW hazard checking would be done for the second instruction while also checking for a RAW hazard against the first instruction (if the second instruction reads from the first instruction's destination register), and then the instructions are issued accordingly if all goes well. If a hazard is detected against only the second instruction, the first instruction will issue as normal with a no-op down the alternate pipe while the second will be stalled for a cycle. If a hazard is detected against the first instruction, both instructions will be held back to preserve in-order operation.

In the case of a control stall, instructions will be queued into a register for issue in the next available cycle, and the program counter will be held back as well. If a branch is detected to be taken in the Branch unit, the new program counter will be sent to the Instruction Fetch unit to replace the existing count, and both the IF and Decode units will stall for a cycle to flush outgoing instructions and prepare for the new instructions to be fetched..

Processing Unit

For this report, all stages following the Decode unit will be collectively known as the Processing unit. This consists of the shared Register File/Forward stage and the seven execution units of the even and odd pipes. The former stage takes one cycle to complete, whereas the execution units have latencies of up to 7 cycles, with an eighth cycle for writing back to the register table. This is a noteworthy point in the pipeline because from here on out, all pipeline hazards will be handled passively through forwarding as opposed through stalling and manipulating the program counter as the previous units did (the Branch unit does not directly do any of this either, it simply sends signals up for the Decode unit to handle accordingly).

Register Table

The register table used in this processor holds 128 registers of 128-bit width. This is notable in that it allows a single register to hold 4 32-bit words at once, or 8 16bit halfwords, which is very useful in some simpler computations that operate on those data types. The table has 6 read ports (one for each register ra, rb, and rc of both pipes) and two write ports (one each for the result of both pipes' instructions). After the respective instructions are pushed to the register access stage, their source register addresses are decoded and the contents of these registers are read asynchronously. After being read, the register table checks if there are any inbound writes being performed to these source registers on the same cycle (representing the writeback stage of these older instructions). If so, the register table instead forwards these values to the next stage rather than the register contents, as these writeback values are more recent.

The write operation of the register table is simpler. During the writeback stage of an instruction, three result signals are sent to the register table: the 128-bit result value, the destination register address, and the reg_write signal letting the unit know if it should be updated with the new result value. Despite having two write ports, two results cannot be written to the same register at the same time as that would create a structural hazard via collision. The Decode stage of the pipeline will not issue a pair instructions in one cycle if they both write to the same destination register, instead opting to only issue the first instruction and stall the latter until the next cycle.

Data Forwarding

The data forwarding unit consists entirely of combinational logic, with no use of registers. It is in the same pipeline stage as register access, but uses signals from the register table after being read. It is used to prevent RAW (read-after-write) hazards by forwarding readied results from the pipe staging registers directly to the input buses of the execution units. For each of the six possible register table read port outputs, it checks every stage of both pipes' forwarding registers for a destination register address matching a source register. If a matching address is found, and that stage's reg_write signal is asserted, the result from that stage is used to update the source value of the respective source register, allowing the most recently updated value to be used in the new instruction. Lower forwarding stages (with more recently issued instructions) are prioritized, and there is no need to account for both pipes having the same destination address as this structural hazard is prevented by the decode stage, as mentioned in the above section.

Even Pipe

The even pipe, or execution pipe, of the Processing Unit consists of four units focused more on arithmetic operations and the byte data type. They are the Single Precision (FP) unit, the Simple Fixed 2 (FX2) unit, the Byte unit, and the Simple Fixed 1 (FX1) unit.

The Single Precision unit handles two types of instructions: floating point arithmetic and integer multiplication. In the true SPE hardware, floating point operations take six stages to be calculated, while integer operations take one more stage to perform. While calculations in the SPU-lite simulation model all occur in the first execution cycle and propagate internally before reaching the respective staging register, this difference in latency is simulated by having an internal shift register depth of seven stages and adding a control signal called int. Quite simply, if the int control signal is not asserted for a given result, it will be released from the unit after six stages; if int is asserted, it will be released after the seventh stage.

The Simple Fixed 2 unit performs shift and rotate operations. In the original hardware, the unit used three stages to operate, but needed four cycles to complete an instruction before it could be forwarded. To simulate this, the SPU-lite model behaves as if the unit has four delay stages rather than three, but otherwise follows specifications.

The Byte unit performs four operations on the byte data type exclusively. It has the same pipeline behavior as the Simple Fixed 2 unit, with regard to latency and delay staging, resulting in an effective latency of four cycles.

The Simple Fixed 1 unit performs most arithmetic and logical operations, as well as compares. It performs, by far, the most unique instructions of any unit, yet only has a pipeline depth of two stages with the same latency, meaning results are ready after two full cycles of operation.

Odd Pipe

The odd pipe, or load pipe, of the Processing Unit consists of three units specializing in memory access and program counter manipulation, as well as operating on quadword values. They are the Permute unit, the Local Store (LS) unit, and the Branch unit.

The Permute unit primarily performs quadword operations, meaning it can shift or rotate an entire register's contents as one value, while also doing gather operations. It has the same pipeline behavior as the Simple Fixed 2 unit, with regard to latency and delay staging, resulting in an effective latency of four cycles.

The Local Store unit performs all memory accesses for the SPE. There is a 32KB local memory that the unit directly interfaces with for all of its instructions. This memory is byte-addressed, consisting of 32,767 (2^{15}-1) memory slots with 8-bit data per slot. Memory accesses are always to quadwords, which is the full size of the 128-bit registers. Memory access is done via one of three addressing modes: indexed addressing, base/indexed addressing, and direct addressing. All memory accesses actually occur on the first execution stage, with the results from a load instruction being ready for forwarding after six cycles. This means there are no RAW hazards within the Local Store unit as reads always obtain the most recent data.

The Branch unit handles all changes to the program counter, both unconditional and conditional. In other ISAs, unconditional branches may be known as jump instructions, but in Cell SPU, all are referred to as branch instructions. The 1KB instruction memory is word-addressed, consisting of 256 (2^{8}-1) memory slots with 32-bit data per slot, each occupied by a single instruction. Each branch instruction takes only a single cycle to execute, and rather than be forwarded to the odd pipe's delay stages, the new program counter and a branch_taken control signal are sent directly to the Instruction Fetch and Decode units, which then handle the next fetch based on the branch_taken flag. A signal is also sent to all other functional units to kill the instructions that have been issued after branch to ensure in-order operation is kept and all control hazards are accounted for.

Testing/Results

Figure 2: Sample Program for Testing Hazard Handling (Binary, Hex, and Assembly)

```
1  00011100000000110000000000000000    0x1C030000
2  01000001111111111111100010000100    0x41FFF884
3  00011100000000000100000000000001    0x1C004001
4  01000001111111111111100100000101    0x41FFF905
5  00011101000000111000000100000010    0x1D038102
6  01000001111111111111100110000110    0x41FFF986
7  00011101000000000100000100000011    0x1D004103
8  00100100000000000000000010000001    0x24000081
9  00100100000000000100000010000001    0x24004081
10 00100100000000001000000010000001    0x24008081
11 00100100000000001100000010000001    0x2400C081
12 00110000100000000000011010000100    0x30800684
13 00110000100000000000011100000101    0x30800705
14 00110000100000000000011110000110    0x30800786
15 00110000100000000000100000000111    0x30800807
16 01000010000000000000000110011001    0x42000199
17 01000010000000000000000000011010    0x4200001A
18 00011100111111111100110010011001    0x1CFFCC99
19 00011100000000000100110100011010    0x1C004D1A
20 00100001011111111111111110011001    0x217FFF19
21 00011010011000000100000000000010    0x1A604002
22 00111011000000010100001000001100    0x3B01420C
23 01111000100000010000000110000101    0x78810185
24 00111011000000000000001000001101    0x3B00020D
25 01111000100000011100001100001000    0x7881C308
26 00111011000000000100001000001110    0x3B00420E
27 01111000100000101000010010001011    0x7882848B
28 00111011000000001100001000001111    0x3B00C20F
29 01111000100001101000110010010001    0x78868C91
30 00111011000000001000001000001111    0x3B00820F
```

```
1  ai 0 0 12       //Even pipe structural hazard
2  ilh 4 -15
3  ai 1 0 1
4  ilh 5 -14
5  ahi 2 2 14
6  ilh 6 -13
7  ahi 3 2 1
8  stqd 1 0(1)     //Odd pipe structural hazard, plus memory ref
9  stqd 1 1(1)
10 stqd 1 2(1)
11 stqd 1 3(1)
12 lqa 4 13
13 lqa 5 14
14 lqa 6 15
15 lqa 7 16
16 ila 25 3        //Set r25 to 3 and decrement
17 ila 26 0
18 ai 25 25 -1     //Keep decrementing until r25==0
19 ai 26 26 1
20 brnz 25 -2      //Branch hazard resolution, odd destination addr;
21 avgb 2 6-2      //Dual issue example
22 rotqbi 12 4 5
23 mpy 5 3 4
24 rotqbi 13 4 0
25 mpy 8 6 7
26 rotqbi 14 4 1
27 mpy 11 9 10
28 rotqbi 15 4 3
29 mpy 17 25 26
30 rotqbi 15 4 2   //Forwarding without stalling
31 stop            //Stop instr ignores all following instr and ends
32 mpy 8 6 7
33 rotqbi 14 4 1
34 mpy 11 9 10
35 rotqbi 15 4 3
36 mpy 14 22 23
```

In order to test proper operation of the Cell SPU-lite processor, the program shown in Figure 2 was drafted to highlight a number of operating conditions. These are pairs of instructions fetched for the same pipe, RAW hazards resolved via stalling and forwarding, branch control hazards, optimal dual-issue operation, and RAW hazard resolved via forwarding without stalling.

The first conditions tested were pairs of instructions going to the same functional pipe. Lines 1-7 of Figure 2 show 7 instructions being issued to the Register File/Forwarding unit to go to the even pipe. This can be seen in the pipeline in Figure 3, where the even pipe is shown to be populated for 7 consecutive cycles rather than issuing twice per cycle. The stall signal used to modulate this is also shown, though note that this signal is used to route instructions in the cycle following the pulse. Figure 4 shows something similar, but with instructions 9-15 in the odd pipe.

Figure 3: Even Pipe Populated with 7 Consecutive Lone Instructions

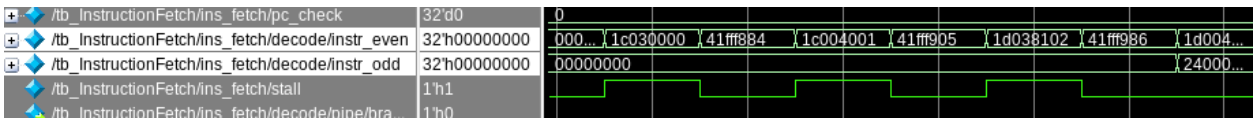
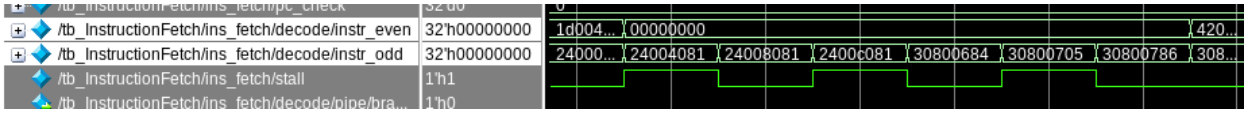
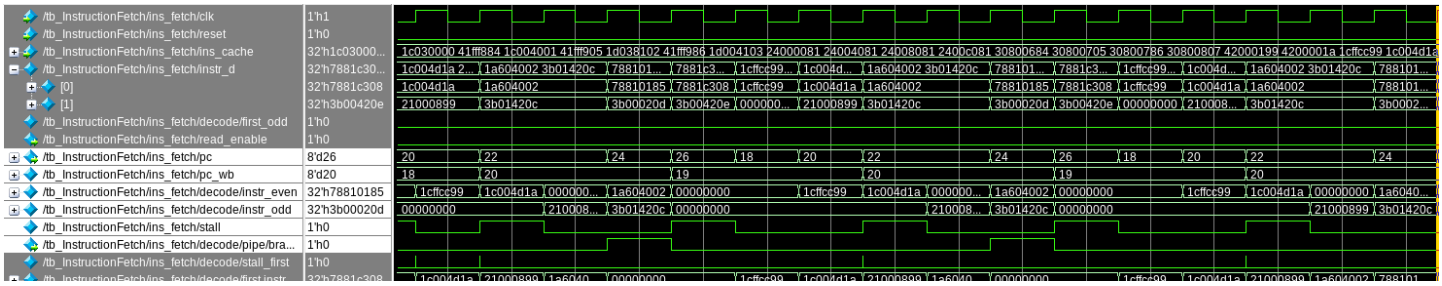


Figure 4: Odd Pipe Populated with 7 Consecutive Lone Instructions



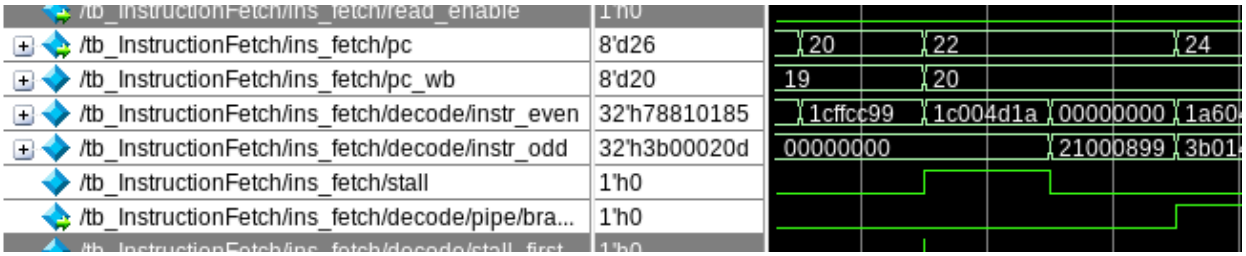
Following the two stalled pipeline sequences in a very typical case that approximates a for loop from high level programming. Line 16 sets register 25 to 3 in the preferred slot, while Line 18 is equivalent to a “dec r25” command. Following that in line 20 is the BRNZ instruction, which sets up the following loop: for (r25 = 3; r25 > 0; r25--). In other words, the register will decrement every cycle, allowing us to perform other tasks in the loop; in this case, that task is simply incrementing r26 to show the number of loops. Note that because the registers are assigned and decremented before the branch conditional is reached, the loop is effectively starting at r25 = 2. The whole loop dynamic is shown in Figure 5 below.

Figure 5: Branch Control Hazard Resolution with RAW Stall



Of course, another thing to note about the loop above is that it demonstrates a RAW stall as well. Line 20 (BRNZ) is an odd-pipe instruction, while line 19 (r26++) is an even-pipe instruction. The reason these two don’t issue together in spite of this is because Line 20 has a dependency on line 18 (r25--), so it must wait until the FX2 unit’s data is available for reading before it can check for branch taken. FX2 has a latency of two cycles, which would suggest 3 cycles are necessary to read the data from it due to the RF stage adding one, but results are ready to forward to the RF/FWD stage once FX2 hits its final stage. This means that once data is in the first cycle of FX2, Decode is free to issue any depending instructions; by the next cycle they’ll reach the FWD stage and read the result of FX2-2. This is shown in Figure 6. Line 18 is the leftmost instruction in the even pipe, while line 20 is the first non-zero instruction in the odd.

Figure 6: Zoomed View of Branch Control Hazard Resolution with RAW Stall



The stall signal is pulsed in alignment with line 19 as that cycle was originally due to issue both the (inc r26) instruction and the BNEZ, but the latter still needed to be delayed one stage in order to safely read the result of FX2 and the (dec r25) instruction.

The next shown condition of the processor's operation is the ideal dual-issue case. Lines 21-30 are pairs of even and odd instructions with no dependencies among one another, except for line 30. As one can see from Figure 7 below, they are able to issue in pairs with no stalling. One can see from looking at the code though that line 30 (rotqbi 15 4 2; 0x3B00820F) has a dependency on line 21 (avgb 2 0 1; 0x1A604002), with the first instruction writing to register 2 and the latter one using register 2 as a data source. Yet as seen in Figure 7, there are no stalls seen or active hazard prevention used. This is because AVGB is a Byte unit instruction, which means there is a latency of four cycles after issue before it can be read from. The cycle immediately before the yellow dividing line is the cycle in which AVGB is in the RF/FWD stage. The following three cycles have no hazards between the instructions, though on that third cycle after the line, AVGB's result is full calculated and will be sent to the Forwarding unit on the next cycle so that instructions in the decode stage (such as ROTQBI seen a few lines above in the waveform) see the data is available. On the next clock edge, Decode has determined that ROTQBI has all necessary values available for issue and dispatches it with its even pairmate rather than stalling it for a cycle, allowing it to issue exactly when it is allowed to without hazard.

Figure 7: Dual-Issue with No Stalls

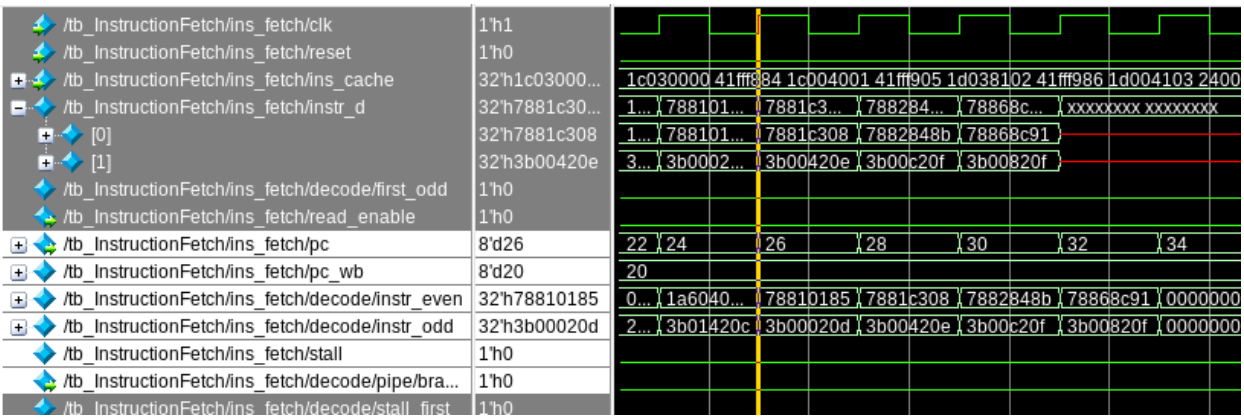


Figure 8: Sample Program for Matrix-Multiplication

```
1   ila 0 0             // store 0 in reg0
2   ila 1 4             // shift=4;
3   ilh 10 -1           // used to create mask FFFFFFFF
4   ilh 30 4            // use to shift by 12byte
5   ilh 31 8            // use to shift by 8 byte
6   ilh 32 12           // use to shift by 8 bytes
7   shlqby 10 10 32     // generate FFFFFFFF ins first word
8   ila 11 4            // i = 4// START OUTER LOOP
9   ai 11 11 -1         // i--
10  ila 12 4            // j = 4 START INNER LOOP
11  ila 42 0            // load_index=0
12  ila 43 16           // rotate=16
13  ila 20 0            // clearing reg 20
14  ai 12 12 -1         // j--;
15  ila 15 0            // clearing reg 15
16  lqd 5 4(42)         // load row(2)= 4+0,+41,4+2,4+3
17  and 15 5 10         // extract the first word of the quadword
18  rotqby 15 15 43     // rotate the word to align in the slot
19  xor 20 15 20        // reg 15 will have elemnt for column 0 matrix2
20  shlqby 5 5 1        // shift row by 4
21  ai 43 43 -4         // rotate-=4
22  stqd 5 4(42)        // store the shifted reg back to memory
23  ai 42 42 1          // load_index+=1
24  brnz 12 -10         // if reg11==0 break else goto ai 12 12 -1
25  stqd 20 10(11)      // store 20 in 13 to 10
26  brnz 11 -17         // if reg11==0 break else goto ai 11 11 -1
27  ila 51 4            // i = 4// START OUTER LOOP
28  ai 51 51 -1         // i--
29  ila 80 0            // reset the quad word
30  lqd 21 0(51)        // load maxtrix 1 rows
31  ila 12 4            // j = 4 START INNER LOOP
32  ila 43 16           // rotate=16s
33  ai 12 12 -1         // j--;
34  lqd 22 10(12)       // load transposed matrix2 rows
35  mpya 23 22 21 23    // multiply slot wise both rows and store qword in 23
36  ila 81 0            // reset the cell
37  ila 45 4            // k=0
38  ai 45 45 -1         // k--
39  and 70 23 10        // using mask to extract the left most word
40  shlqby 23 23 30     // shift left by 4 bytes
41  a 81 81 70          // computing sum of the product cell wise
42  brnz 45 -4          // jump to ai 45 45 -1
43  and 81 81 10        // clearing any other bits
44  rotqby 81 81 43     // rotate reg81 to place word in correct slot of quad
45  ai 43 43 -4         // rotate-=4
46  or 80 80 81         // build the quadword using or with rotated word
47  brnz 12 -14         // jump to ai 12 12 -1
48  stqd 80 20(51)      // store the quad word row for the matrix3
49  brnz 51 -21         //Jump to ai 51 51 -1
50  stop
```

A practical, more complex application of the MIMD format of the Cell SPU-lite processor is the ability to perform multiply-add operations on four 32-bit words of data simultaneously, which simplifies the process or matrix multiplication. Figure 8 shows a program designed to do just that. It extracts preloaded matrices A and B from memory (the values of which are shown

below in Figure 9) and performs masks and masks on matrix B to allow it to be column-aligned, allowing for multiplication with the row-aligned matrix A. The two matrices are then multiplied through consecutive multiply-add operations, and the results are written to memory as matrix C after some more formatting and alignment. Figure 10 shows the results of this operation in memory locations 20-23. The displayed elements, which convert into the decimal values shown in Figure 9, are: [(0x98, 0x9E, 0xA4, 0xAA),
(0x1F8, 0x20E, 0x224, 0x23A),
(0x358, 0x37E, 0x3A4, 0x3CA),
(0x4B8, 0x4EE, 0x524, 0x55A)].

Figure 9: Example Values and Results to be Produced in Matrix-Multiply Program

Matrix A

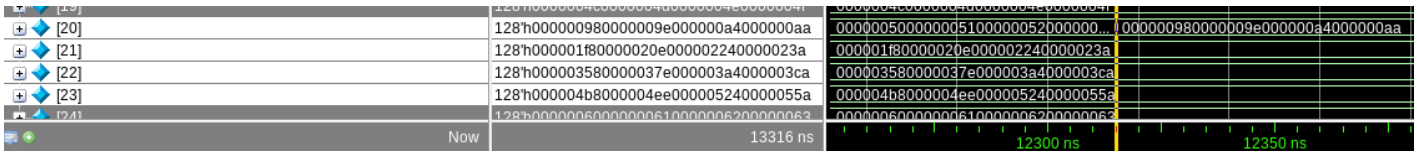
	A_1	A_2	A_3	A_4
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11
4	12	13	14	15

Matrix B

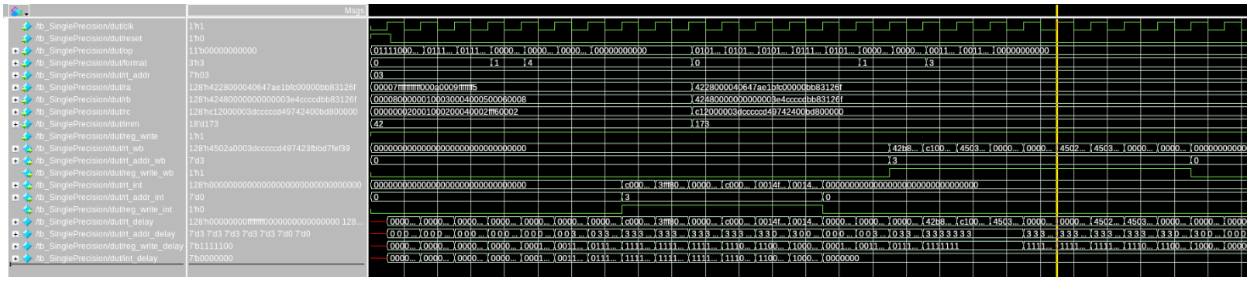
	B_1	B_2	B_3	B_4
1	16	17	18	19
2	20	21	22	23
3	24	25	26	27
4	28	29	30	31

	C_1	C_2	C_3	C_4
1	152	158	164	170
2	504	526	548	570
3	856	894	932	970
4	1208	1262	1316	1370

Figure 10: Results of Matrix-Multiplication in Memory (Hex, Word-Indexed)



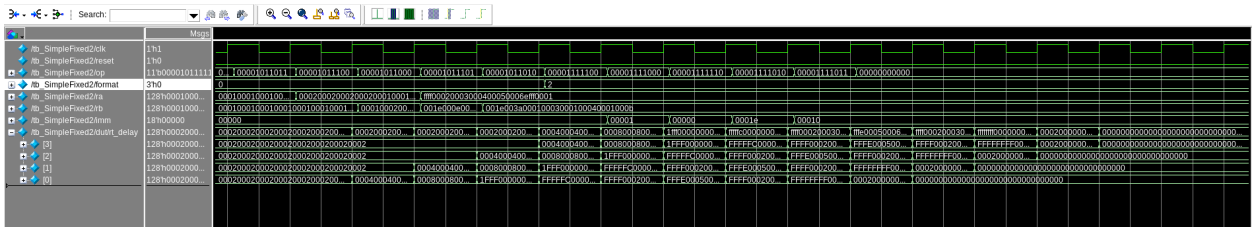
Single Precision (FP)



Instructions tested in above screenshot:

1. mpy: Multiply
2. mpyu: Multiply Unsigned
3. mpyh: Multiply High
4. mpya: Multiply and Add
5. mpyi: Multiply Immediate
6. mpyui: Multiply Unsigned Immediate
7. fa: Floating Add
8. fs: Floating Subtract
9. fm: Floating Multiply
10. fceq: Floating Compare Equal
11. fcgt: Floating Compare Greater Than
12. fma: Floating Multiply and Add
13. fms: Floating Multiply and Subtract
14. cflts: Convert Floating Point to Signed Integer
15. cfltu: Convert Floating Point to Unsigned Integer

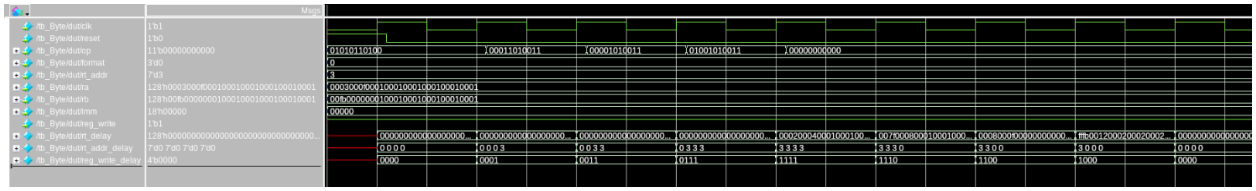
Simple Fixed 2 (FX2)



Instructions tested in above screenshot:

1. shl: Shift Left Word
2. roth: Rotate Halfword
3. rot: Rotate Word
4. rothm: Rotate and Mask Halfword
5. rotma: Rotate and Mask Algebraic Word
6. rothi: Rotate Halfword Immediate
7. rotmai: Rotate and Mask Algebraic Word Immediate
8. shli: Shift Left Word Immediate

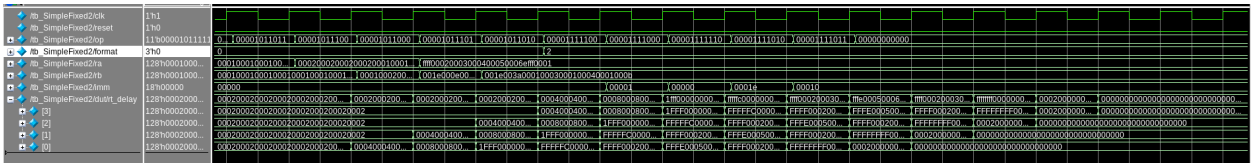
Byte



Instructions tested in above screenshot:

1. cntb: Count Ones in Byte
2. avgb: Average Bytes
3. absdb: Absolute Difference of Bytes
4. sumb: Sum Bytes into Halfwords

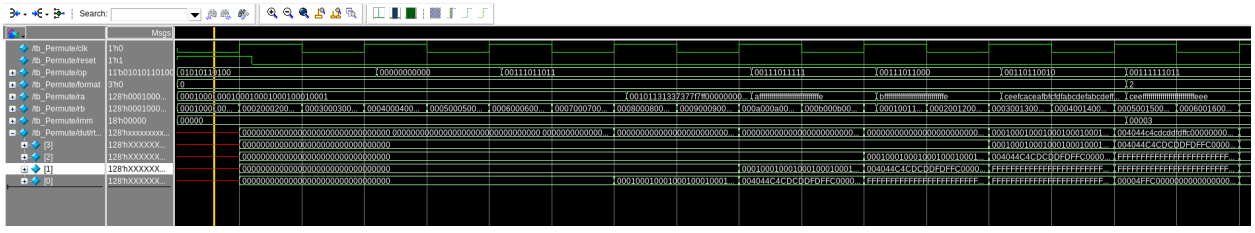
Simple Fixed 1 (FX1)



Instructions tested in above screenshot:

1. ah: Add Halfword
2. sfh: Subtract from Halfword
3. sfhi: Subtract from Halfword Immediate
4. ai: Add Word Immediate
5. and: And
6. or: Or
7. ceqb: Compare Equal Byte
8. cgt: Compare Greater Than Word
9. clgtb: Compare Logical Greater Than Byte
10. andbi: And Byte Immediate
11. xorbi: Exclusive Or Byte Immediate
12. clgti: Compare Logical Greater Than Word Immediate
13. ilh: Immediate Load Halfword
14. ila: Immediate Load Address

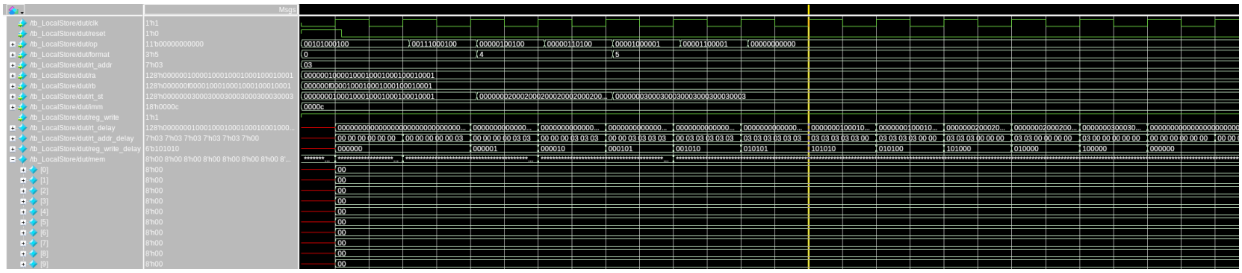
Permute



Instructions tested in above screenshot:

1. shlqbi: Shift Left Quadword by Bits
2. rotqbi: Rotate Quadword by Bits
3. gbb: Gather Bits from Bytes
4. shlqbii: Shift Left Quadword by Bits Immediate
5. rotqbyi: Rotate Quadword by Bytes Immediate

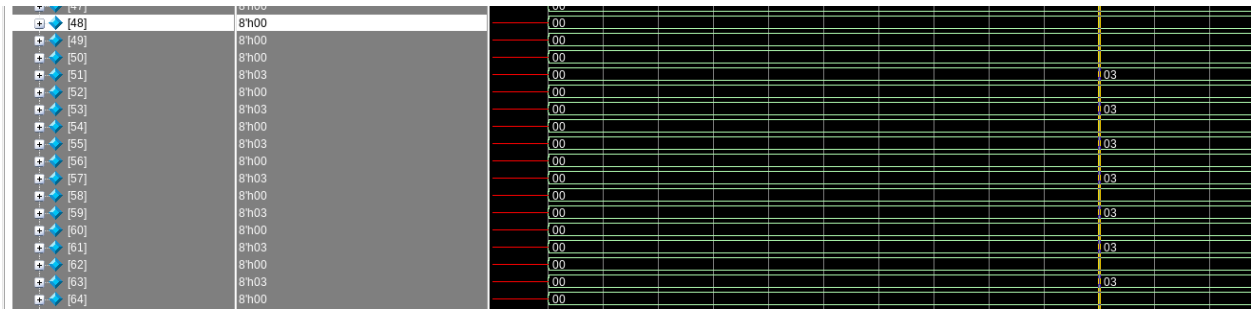
Local Store



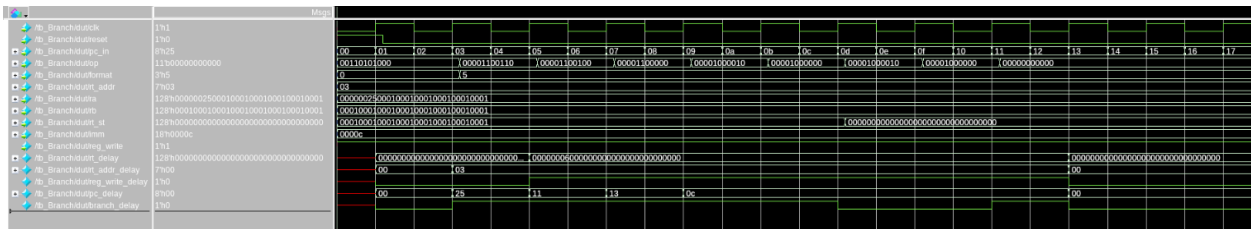
Instructions tested in above screenshot:

1. stqx : Store Quadword (x-form)
2. lqx : Load Quadword (x-form)
3. stqd : Store Quadword (d-form)
4. lqd : Load Quadword (d-form)
5. stqa : Store Quadword (a-form)
6. lqa : Load Quadword (a-form)

An example of writing to memory location 48 is shown below:



Branch



Instructions tested in above screenshot:

1. bi : Branch Indirect
2. brsl: Branch Relative and Set Link
3. br: Branch Relative
4. bra: Branch Absolute
5. brnz: Branch If Not Zero Word
6. brz: Branch If Zero Word

Instruction Set

Name	Mnemonic	RTL Description	Exec Unit	Exec Pipe	Lat-ency
Multiply	mpy rt, ra, rb	$RT^{0:3} \leftarrow RA^{2:3} * RB^{2:3}$ $RT^{4:7} \leftarrow RA^{6:7} * RB^{6:7}$ $RT^{8:11} \leftarrow RA^{10:11} * RB^{10:11}$ $RT^{12:15} \leftarrow RA^{14:15} * RB^{14:15}$	FP1	EVEN	7
Multiply Unsigned	mpyu rt, ra, rb	$RT^{0:3} \leftarrow RA^{2:3} \mid * \mid RB^{2:3}$ $RT^{4:7} \leftarrow RA^{6:7} \mid * \mid RB^{6:7}$ $RT^{8:11} \leftarrow RA^{10:11} \mid * \mid RB^{10:11}$ $RT^{12:15} \leftarrow RA^{14:15} \mid * \mid RB^{14:15}$	FP1	EVEN	7
Multiply Immediate	mpyi rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:3} \leftarrow RA^{2:3} * t$ $RT^{4:7} \leftarrow RA^{6:7} * t$ $RT^{8:11} \leftarrow RA^{10:11} * t$ $RT^{12:15} \leftarrow RA^{14:15} * t$	FP1	EVEN	6
Multiply Unsigned Immediate	mpyui rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:3} \leftarrow RA^{2:3} \mid * \mid t$ $RT^{4:7} \leftarrow RA^{6:7} \mid * \mid t$ $RT^{8:11} \leftarrow RA^{10:11} \mid * \mid t$ $RT^{12:15} \leftarrow RA^{14:15} \mid * \mid t$	FP1	EVEN	6
Multiply and Add	mpya rt, ra, rb, rc	$t0 \leftarrow RA^{2:3} * RB^{2:3}$ $t1 \leftarrow RA^{6:7} * RB^{6:7}$ $t2 \leftarrow RA^{10:11} * RB^{10:11}$ $t3 \leftarrow RA^{14:15} * RB^{14:15}$ $RT^{0:3} \leftarrow t0 + RC^{0:3}$ $RT^{4:7} \leftarrow t1 + RC^{4:7}$ $RT^{8:11} \leftarrow t2 + RC^{8:11}$ $RT^{12:15} \leftarrow t3 + RC^{12:15}$	FP1	EVEN	6
Multiply High	mpyh rt, ra, rb	$t0 \leftarrow RA^{0:1} * RB^{2:3}$ $t1 \leftarrow RA^{4:5} * RB^{6:7}$ $t2 \leftarrow RA^{8:9} * RB^{10:11}$ $t3 \leftarrow RA^{12:13} * RB^{14:15}$ $RT^{0:3} \leftarrow t0^{2:3} \mid\mid 0x0000$ $RT^{4:7} \leftarrow t1^{2:3} \mid\mid 0x0000$ $RT^{8:11} \leftarrow t2^{2:3} \mid\mid 0x0000$ $RT^{12:15} \leftarrow t3^{2:3} \mid\mid 0x0000$	FP1	EVEN	6
Floating Add	fa rt, ra, rb		FP1	EVEN	6
Floating Subtract	fs rt, ra, rb		FP1	EVEN	6
Floating Multiply	fm rt, ra, rb		FP1	EVEN	6
Floating Multiply and Add	fma rt, ra, rb, rc		FP1	EVEN	6
Floating Multiply and	fms rt, ra,		FP1	EVEN	6

Subtract	rb, rc				
Convert Floating to Signed Integer	cflts rt, ra, imm8		FP1	EVEN	4
Convert Floating to Unsigned Integer	cfltu rt, ra, imm8		FP1	EVEN	4
Floating Compare Equal	fceq rt, ra, rb		FP1	EVEN	4
Floating Compare Greater Than	fcgt rt, ra, rb		FP1	EVEN	4

Shift Left Halfword	shlh rt, ra, rb	for j = 0 to 15 by 2
		  s ← $RB^{j::2}$ & 0x001F
		  t ← $RA^{j::2}$
		  for b = 0 to 15
		  if b + s < 16 then r_b ← t_{b+s}
		  else r_b ← 0
		  end
		  $RT^{j::2}$ ← r
		end
			FX2	EVEN	2

Shift Left Word	shl rt, ra, rb	for j = 0 to 15 by 4
		  s ← $RB^{j::4}$ & 0x0000003F
		  t ← $RA^{j::4}$
		  for b = 0 to 31
		  if b + s < 32 then r_b ← t_{b+s}
		  else r_b ← 0
		  end
		  $RT^{j::4}$ ← r
		end
			FX2	EVEN	2

Shift Left Word Immediate	shli rt, ra, imm7	s ← RepLeftBit(I7,32) & 0x0000003F
		for j = 0 to 15 by 4
		  t ← $RA^{j::4}$
		  for b = 0 to 31
		  if b + s < 32 then r_b ← t_{b+s}
		  else r_b ← 0
		  end
		  $RT^{j::4}$ ← r
		end
			FX2	EVEN	2

Rotate Halfword	roth rt, ra, rb	for j = 0 to 15 by 2
		  s ← $RB^{j::2}$ & 0x000F
		  t ← $RA^{j::2}$
		  for b = 0 to 15
		  if b + s < 16 then r_b ← t_{b+s}
		  else r_b ← t_{b+s-16}
		  end
		  $RT^{j::2}$ ← r; end
			FX2	EVEN	2

Rotate Halfword Immediate	rothi rt, ra, imm7	s ← RepLeftBit(I7,16) & 0x000F for j = 0 to 15 by 2	FX2	EVEN	2

		t ← $RA^{j:2}$ for b = 0 to 15 if b + s < 16 then r_b ← t_{b+s} else r_b ← t_{b+s-16} end $RT^{j:2}$ ← r end			
Rotate Word	rot rt, ra, rb	for j = 0 to 15 by 4 s ← $RB^{j:4}$ & 0x00000001F t ← $RA^{j:4}$ for b = 0 to 31 if b + s < 32 then r_b ← t_{b+s} else r_b ← t_{b+s-32} end $RT^{j:4}$ ← r end	FX2	EVEN	2
Rotate Word Immediate	roti rt, ra, imm7	s ← RepLeftBit(I7,32) & 0x0000001F for j = 0 to 15 by 4 t ← $RA^{j:4}$ for b = 0 to 31 if b + s < 32 then r_b ← t_{b+s} else r_b ← t_{b+s-32} end $RT^{j:4}$ ← r end	FX2	EVEN	2
Rotate and Mask Halfword	rothm rt, ra, rb	for j = 0 to 15 by 2 s ← (0 - $RB^{j:2}$) & 0x001F t ← $RA^{j:2}$ for b = 0 to 15 if b ≥ s then r_b ← t_{b-s} else r_b ← 0 end $RT^{j:2}$ ← r end	FX2	EVEN	2
Rotate and Mask Word	rotm rt, ra, rb	for j = 0 to 15 by 4 s ← (0 - $RB^{j:4}$) & 0x0000003F t ← $RA^{j:4}$ for b = 0 to 32 if b ≥ s then r_b ← t_{b-s} else r_b ← 0 end $RT^{j:4}$ ← r end	FX2	EVEN	2
Rotate and Mask Algebraic Halfword	rotmah rt, ra, rb	for j = 0 to 15 by 2 s ← (0 - $RB^{j:2}$) & 0x001F t ← $RA^{j:2}$	FX2	EVEN	2

		for b = 0 to 15
		  if b ≥ s then $r_b \leftarrow t_{b-s}$
		  else $r_b \leftarrow t_0$
		 end
		 $RT^{j::2} \leftarrow r$
		end			

Rotate and Mask Algebraic Halfword Immediate	rotmahi rt, ra, imm7	s ← (0 - RepLeftBit(I7,16)) & 0x001F
		for j = 0 to 15 by 2
		 t ← $RA^{j::2}$
		 for b = 0 to 15
		  if b ≥ s then $r_b \leftarrow t_{b-s}$
		  else $r_b \leftarrow t_0$
		 end
		 $RT^{j::2} \leftarrow r$
		end
			FX2	EVEN	2

Rotate and Mask Algebraic Word	rotma rt, ra, rb	for j = 0 to 15 by 4
		 s ← (0 - $RB^{j::4}$) & 0x001F
		 t ← $RA^{j::4}$
		 for b = 0 to 31
		  if b ≥ s then $r_b \leftarrow t_{b-s}$
		  else $r_b \leftarrow t_0$
		 end
		 $RT^{j::4} \leftarrow r$
		end
			FX2	EVEN	2

Rotate and Mask Algebraic Word Immediate	rotmai rt, ra, imm7	s ← (0 - RepLeftBit(I7,32)) & 0x0000003F
		for j = 0 to 15 by 4
		 t ← $RA^{j::4}$
		 for b = 0 to 31
		  if b ≥ s then $r_b \leftarrow t_{b-s}$
		  else $r_b \leftarrow t_0$
		 end
		 $RT^{j::4} \leftarrow r$
		end
			FX2	EVEN	2

Count Ones in Bytes	cntb rt, ra	for j = 0 to 15 c = 0 b ← RA^j for m = 0 to 7 If b_m = 1 then c ← c + 1 end $RT^j \leftarrow c$ end	BYTE	EVEN	7
Average Bytes	avgb rt, ra, rb	for j = 0 to 15 $RT^j \leftarrow ((0x00 \,\|\|\, RA^j)+(0x00 \,\|\|\, RB^j)+1)_{7:14}$	BYTE	EVEN	7
Absolute Differences of Bytes	absdb rt, ra, rb	for j = 0 to 15 if ($RB^j >^u RA^j$) then $RT^j \leftarrow RB^j - RA^j$ else $RT^j \leftarrow RA^j - RB^j$ end	BYTE	EVEN	7

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	7
Immediate Load Halfword	ilh rt, imm16	$s \leftarrow I16$ $RT^{0:1} \leftarrow s$ $RT^{2:3} \leftarrow s$ $RT^{4:5} \leftarrow s$ $RT^{6:7} \leftarrow s$ $RT^{8:9} \leftarrow s$ $RT^{10:11} \leftarrow s$ $RT^{12:13} \leftarrow s$ $RT^{14:15} \leftarrow s$	FX1	EVEN	4
Immediate Load Halfword Upper	ilhu rt, imm16	$t \leftarrow I16 \,\|\|\, 0x0000$ $RT^{0:3} \leftarrow t$ $RT^{4:7} \leftarrow t$ $RT^{8:11} \leftarrow t$ $RT^{12:15} \leftarrow t$	FX1	EVEN	4
Immediate Load Word	il rt, imm16	$s \leftarrow RepLeftBit(I16,32)$ $RT^{0:3} \leftarrow s$ $RT^{4:7} \leftarrow s$ $RT^{8:11} \leftarrow s$ $RT^{12:15} \leftarrow s$	FX1	EVEN	4
Immediate Load Address	ila rt, imm18	$t \leftarrow {}_{14}0 \,\|\|\, I18$ $RT^{0:3} \leftarrow t$ $RT^{4:7} \leftarrow t$ $RT^{8:11} \leftarrow t$ $RT^{12:15} \leftarrow t$	FX1	EVEN	4
Immediate Or Halfword Lower	iohl rt, imm16	$t \leftarrow 0x0000 \,\|\|\, I16$ $RT^{0:3} \leftarrow RT^{0:3} \mid t$ $RT^{4:7} \leftarrow RT^{4:7} \mid t$ $RT^{8:11} \leftarrow RT^{8:11} \mid t$ $RT^{12:15} \leftarrow RT^{12:15} \mid t$	FX1	EVEN	4
Add Halfword	ah rt, ra, rb	$RT^{0:1} \leftarrow RA^{0:1} + RB^{0:1}$ $RT^{2:3} \leftarrow RA^{2:3} + RB^{2:3}$ $RT^{4:5} \leftarrow RA^{4:5} + RB^{4:5}$ $RT^{6:7} \leftarrow RA^{6:7} + RB^{6:7}$ $RT^{8:9} \leftarrow RA^{8:9} + RB^{8:9}$ $RT^{10:11} \leftarrow RA^{10:11} + RB^{10:11}$ $RT^{12:13} \leftarrow RA^{12:13} + RB^{12:13}$ $RT^{14:15} \leftarrow RA^{14:15} + RB^{14:15}$	FX1	EVEN	4

Add Halfword Immediate	ahi rt, ra, imm10	s ← RepLeftBit(I10,16) $RT^{0:1}$← $RA^{0:1}$ + s $RT^{2:3}$← $RA^{2:3}$ + s $RT^{4:5}$← $RA^{4:5}$ + s $RT^{6:7}$← $RA^{6:7}$ + s $RT^{8:9}$← $RA^{8:9}$ + s $RT^{10:11}$← $RA^{10:11}$ + s $RT^{12:13}$← $RA^{12:13}$ + s $RT^{14:15}$← $RA^{14:15}$ + s	FX1	EVEN	4
Add Word	a rt, ra, rb	$RT^{0:3}$← $RA^{0:3}$ + $RB^{0:3}$ $RT^{4:7}$← $RA^{4:7}$ + $RB^{4:7}$ $RT^{8:11}$← $RA^{8:11}$ + $RB^{8:11}$ $RT^{12:15}$← $RA^{12:15}$ + $RB^{12:15}$	FX1	EVEN	4
Add Word Immediate	ai rt, ra, imm10	t ← RepLeftBit(I10,32) $RT^{0:3}$← $RA^{0:3}$ + t $RT^{4:7}$← $RA^{4:7}$ + t $RT^{8:11}$← $RA^{8:11}$ + t $RT^{12:15}$← $RA^{12:15}$ + t	FX1	EVEN	4
Subtract from Halfword	sfh rt, ra, rb	$RT^{0:1}$← $RB^{0:1}$ + ¬$RA^{0:1}$ + 1 $RT^{2:3}$← $RB^{2:3}$ + ¬$RA^{2:3}$ + 1 $RT^{4:5}$← $RB^{4:5}$ + ¬$RA^{4:5}$ + 1 $RT^{6:7}$← $RB^{6:7}$ + ¬$RA^{6:7}$ + 1 $RT^{8:9}$← $RB^{8:9}$ + ¬$RA^{8:9}$ + 1 $RT^{10:11}$← $RB^{10:11}$ + ¬$RA^{10:11}$ + 1 $RT^{12:13}$← $RB^{12:13}$ + ¬$RA^{12:13}$ + 1 $RT^{14:15}$← $RB^{14:15}$ + ¬$RA^{14:15}$ + 1	FX1	EVEN	4
Subtract from Halfword Immediate	sfhi rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:1}$← t + ¬$RA^{0:1}$ + 1 $RT^{2:3}$← t + ¬$RA^{2:3}$ + 1 $RT^{4:5}$← t + ¬$RA^{4:5}$ + 1 $RT^{6:7}$← t + ¬$RA^{6:7}$ + 1 $RT^{8:9}$← t + ¬$RA^{8:9}$ + 1 $RT^{10:11}$← t + ¬$RA^{10:11}$ + 1 $RT^{12:13}$← t + ¬$RA^{12:13}$ + 1 $RT^{14:15}$← t + ¬$RA^{14:15}$ + 1	FX1	EVEN	4
Subtract from Word	sf rt, ra, rb	$RT^{0:3}$← $RB^{0:3}$ + ¬$RA^{0:3}$ + 1 $RT^{4:7}$← $RB^{4:7}$ + ¬$RA^{4:7}$ + 1 $RT^{8:11}$← $RB^{8:11}$ + ¬$RA^{8:11}$ + 1 $RT^{12:15}$← $RB^{12:15}$ + ¬$RA^{12:15}$ + 1	FX1	EVEN	4
Subtract from Word Immediate	sfi rt, ra, imm10	t ← RepLeftBit(I10,32) RT^(0:3) ← t + (¬RA^(0:3)) + 1 RT^(4:7) ← t + (¬RA^(4:7)) + 1 RT^(8:11) ← t + (¬RA^(8:11)) + 1 RT^(12:15) ← t + (¬RA^(12:15)) + 1	FX1	EVEN	4
And	and rt,ra,rb	RT← RA & RB	FX1	EVEN	4

And Byte Immediate	andbi rt, ra, imm10	b ← I10 & 0x00FF bbbb ← b || b || b || b $RT^{0:3}$← $RA^{0:3}$ & bbbb $RT^{4:7}$← $RA^{4:7}$ & bbbb $RT^{8:11}$← $RA^{8:11}$ & bbbb $RT^{12:15}$← $RA^{12:15}$ & bbbb	FX1	EVEN	2
And Halfword Immediate	andhi rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:1}$← $RA^{0:1}$ & t $RT^{2:3}$← $RA^{2:3}$ & t $RT^{4:5}$← $RA^{4:5}$ & t $RT^{6:7}$← $RA^{6:7}$ & t $RT^{8:9}$← $RA^{8:9}$ & t $RT^{10:11}$← $RA^{10:11}$ & t $RT^{12:13}$← $RA^{12:13}$ & t $RT^{14:15}$← $RA^{14:15}$ & t	FX1	EVEN	2
And Word Immediate	andi rt, ra, imm10	t ← RepLeftBit(I10,32) $RT^{0:3}$← $RA^{0:3}$ & t $RT^{4:7}$← $RA^{4:7}$ & t $RT^{8:11}$← $RA^{8:11}$ & t $RT^{12:15}$← $RA^{12:15}$ & t	FX1	EVEN	2
Or	or rt, ra, rb	RT← RA | RB	FX1	EVEN	2
Or Byte Immediate	orbi rt, ra, imm10	b ← I10 & 0x00FF bbbb ← b || b || b || b $RT^{0:3}$← $RA^{0:3}$ | bbbb $RT^{4:7}$← $RA^{4:7}$ | bbbb $RT^{8:11}$← $RA^{8:11}$ | bbbb $RT^{12:15}$← $RA^{12:15}$ | bbbb	FX1	EVEN	2
Or Halfword Immediate	orhi rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:1}$← $RA^{0:1}$ | t $RT^{2:3}$← $RA^{2:3}$ | t $RT^{4:5}$← $RA^{4:5}$ | t $RT^{6:7}$← $RA^{6:7}$ | t $RT^{8:9}$← $RA^{8:9}$ | t $RT^{10:11}$← $RA^{10:11}$ | t $RT^{12:13}$← $RA^{12:13}$ | t $RT^{14:15}$← $RA^{14:15}$ | t	FX1	EVEN	2
Or Word Immediate	ori rt, ra, imm10	t ← RepLeftBit(I10,32) $RT^{0:3}$← $RA^{0:3}$ | t $RT^{4:7}$← $RA^{4:7}$ | t $RT^{8:11}$← $RA^{8:11}$ | t $RT^{12:15}$← $RA^{12:15}$ | t	FX1	EVEN	2
Xor	xor rt,ra,rb	RT← RA ⊕ RB	FX1	EVEN	2
Xor Byte Immediate	xorbi rt, ra, imm10	b ← I10 & 0x00FF bbbb ← b || b || b || b $RT^{0:3}$← $RA^{0:3}$ ⊕ bbbb $RT^{4:7}$← $RA^{4:7}$ ⊕ bbbb	FX1	EVEN	2

		$RT^{8:11} \leftarrow RA^{8:11} \oplus bbbb$ $RT^{12:15} \leftarrow RA^{12:15} \oplus bbbb$			
Xor Halfword Immediate	xorhi rt, ra, imm10	t ← RepLeftBit(I10,16) $RT^{0:1} \leftarrow RA^{0:1} \oplus t$ $RT^{2:3} \leftarrow RA^{2:3} \oplus t$ $RT^{4:5} \leftarrow RA^{4:5} \oplus t$ $RT^{6:7} \leftarrow RA^{6:7} \oplus t$ $RT^{8:9} \leftarrow RA^{8:9} \oplus t$ $RT^{10:11} \leftarrow RA^{10:11} \oplus t$ $RT^{12:13} \leftarrow RA^{12:13} \oplus t$ $RT^{14:15} \leftarrow RA^{14:15} \oplus t$	FX1	EVEN	2
Xor Word Immediate	xori rt, ra, imm10	t ← RepLeftBit(I10,32) $RT^{0:3} \leftarrow RA^{0:3} \oplus t$ $RT^{4:7} \leftarrow RA^{4:7} \oplus t$ $RT^{8:11} \leftarrow RA^{8:11} \oplus t$ $RT^{12:15} \leftarrow RA^{12:15} \oplus t$	FX1	EVEN	2
Nand	nand rt, ra, rb	RT← ¬(RA & RB)	FX1	EVEN	2
Compare Equal Byte	ceqb rt, ra, rb	for i = 0 to 15 If $RA^i = RB^i$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	EVEN	2
Compare Equal Byte Immediate	ceqbi rt, ra, imm10	for i = 0 to 15 If $RA^i = I10_{2:9}$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	EVEN	2
Compare Equal Halfword	ceqh rt, ra, rb	for i = 0 to 15 by 2 If $RA^{i::2} = RB^{i::2}$ then $RT^{i::2} \leftarrow$ 0xFFFF else $RT^{i::2} \leftarrow$ 0x0000 end	FX1	EVEN	2
Compare Equal Halfword Immediate	ceqhi rt, ra, imm10	for i = 0 to 15 by 2 If $RA^{i::2}$ = RepLeftBit(I10,16) then $RT^{i::2} \leftarrow$ 0xFFFF else $RT^{i::2} \leftarrow$ 0x0000 end	FX1	EVEN	2
Compare Equal Word	ceq rt, ra, rb	for i = 0 to 15 by 4 If $RA^{i::4} = RB^{i::4}$ then $RT^{i::4} \leftarrow$ 0xFFFFFFFF else $RT^{i::4} \leftarrow$ 0x00000000 end	FX1	EVEN	2
Compare Equal Word Immediate	ceqi rt, ra, imm10	for i = 0 to 15 by 4 If $RA^{i::4}$ = RepLeftBit(I10,32) then $RT^{i::4} \leftarrow$ 0xFFFFFFFF else $RT^{i::4} \leftarrow$ 0x00000000 end	FX1	EVEN	2

Compare Greater Than Byte	cgtb rt, ra, rb	for i = 0 to 15 If $RA^i > RB^i$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	EVEN	2
Compare Greater Than Byte Immediate	cgtbi rt, ra, imm10	for i = 0 to 15 If $RA^i > I10_{2:9}$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	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 $RT^{i:2} \leftarrow$ 0xFFFF else $RT^{i:2} \leftarrow$ 0x0000 end	FX1	EVEN	2
Compare Greater Than Halfword Immediate	cgthi rt, ra, imm10	for i = 0 to 15 by 2 If $RA^{i:2}$ > RepLeftBit(I10,16) then $RT^{i:2} \leftarrow$ 0xFFFF else $RT^{i:2} \leftarrow$ 0x0000 end	FX1	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 $RT^{i:4} \leftarrow$ 0xFFFFFFFF else $RT^{i:4} \leftarrow$ 0x00000000 end	FX1	EVEN	2
Compare Greater Than Word Immediate	cgti rt, ra, imm10	for i = 0 to 15 by 4 If $RA^{i:4}$ > RepLeftBit(I10,32) then $RT^{i:4} \leftarrow$ 0xFFFFFFFF else $RT^{i:4} \leftarrow$ 0x00000000 end	FX1	EVEN	2
Compare Logical Greater Than Byte	clgtb rt, ra, rb	for i = 0 to 15 If $RA^i >^u RB^i$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	EVEN	2
Compare Logical Greater Than Byte Immediate	clgtbi rt, ra, imm10	for i = 0 to 15 If $RA^i >^u I10_{2:9}$ then $RT^i \leftarrow$ 0xFF else $RT^i \leftarrow$ 0x00 end	FX1	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 $RT^{i:2} \leftarrow$ 0xFFFF else $RT^{i:2} \leftarrow$ 0x0000 end	FX1	EVEN	2
Compare Logical Greater Than Halfword Immediate	clgthi rt, ra, imm10	for i = 0 to 15 by 2 If $RA^{i:2} >^u$ RepLeftBit(I10,16) then $RT^{i:2} \leftarrow$ 0xFFFF else $RT^{i:2} \leftarrow$ 0x0000 end	FX1	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 $RT^{i::4} \leftarrow$ 0xFFFFFFFF else $RT^{i::4} \leftarrow$ 0x00000000 end	FX1	EVEN	2
Compare Logical Greater Than Word Immediate	clgti rt, ra, imm10	for i = 0 to 15 by 4 If $RA^{i::4} >^{u}$ RepLeftBit(I10,32) then $RT^{i::4} \leftarrow$ 0xFFFFFFFF else $RT^{i::4} \leftarrow$ 0x00000000 end	FX1	EVEN	2
Shift Left Quadword by Bits	shlqbi rt, ra, rb	$s \leftarrow RB_{29:31}$ for b = 0 to 127 if b + s < 128 then $r_b \leftarrow RA_{b+s}$ else $r_b \leftarrow 0$ end RT ← r	PERM	ODD	6
Shift Left Quadword by Bits Immediate	shlqbii rt, ra, imm7	$s \leftarrow I7 \,\&\, 0x07$ for b = 0 to 127 if b + s < 128 then $r_b \leftarrow RA_{b+s}$ else $r_b \leftarrow 0$ end RT ← r	PERM	ODD	6
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 ← r	PERM	ODD	6
Shift Left Quadword by Bytes Immediate	shlqbyi rt, ra, imm7	$s \leftarrow I7 \,\&\, 0x1F$ for b = 0 to 15 if b + s < 16 then $r^b \leftarrow RA^{b+s}$ else $r^b \leftarrow 0$ end RT ← r	PERM	ODD	6
Rotate Quadword by Bits	rotqbi rt, ra, rb	$s \leftarrow RB_{29:31}$ for b = 0 to 127 if b + s < 128 then $r_b \leftarrow RA_{b+s}$ else $r_b \leftarrow RA_{b+s-128}$ end RT ← r	PERM	ODD	6
Rotate Quadword by Bits Immediate	rotqbii rt, ra, imm7	$s \leftarrow I7_{4:6}$ for b = 0 to 127 if b + s < 128 then $r_b \leftarrow RA_{b+s}$ else $r_b \leftarrow RA_{b+s-128}$ end RT ← r	PERM	ODD	1

Rotate Quadword by Bytes	rotqby rt, ra, rb	$s \leftarrow RB_{28:31}$
		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 ← r
			PERM	ODD	1

Rotate Quadword by Bytes Immediate	rotqby rt, ra, rb	s ← I7 & 0x1F
		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 ← r
			PERM	ODD	1

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 || s
		$RT^{4:7} \leftarrow 0$
		$RT^{8:11} \leftarrow 0$
		$RT^{12:15} \leftarrow 0$
			PERM	ODD	1

Gather Bits from Halfwords	gbh rt, ra	k = 0
		s = 0x00
		for j = 15 to 128 by 16
		  $s_k \leftarrow RA_j$
		  k = k + 1
		end
		$RT^{0:3} \leftarrow$ 0x000000 || s
		$RT^{4:7} \leftarrow 0$
		$RT^{8:11} \leftarrow 0$
		$RT^{12:15} \leftarrow 0$
			PERM	ODD	1

Gather Bits from Words	gb rt, ra	k = 0
		s = 0x0
		for j = 31 to 128 by 132
		  $s_k \leftarrow RA_j$
		  k = k + 1
		end
		$RT^{0:3} \leftarrow$ 0x00000000 || s
		$RT^{4:15} \leftarrow 0$
			PERM	ODD	1

Load Quadword (d-form)	lqd rt, imm10(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA, 16)	LS	ODD	4
Load Quadword (x-form)	lqx rt, ra, rb	LSA ← ($RA^{0:3}$ + $RB^{0:3}$) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA,16)	LS	ODD	4

Load Quadword (a-form)	lqa rt, imm16	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA,16)	LS	ODD	4
Store Quadword (d-form)	stqd rt, imm10(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + $RA^{0:3}$) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	ODD	4
Store Quadword (x-form)	stqx rt, ra, rb	LSA ← ($RA^{0:3}$ + $RB^{0:3}$) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	ODD	4
Store Quadword (a-form)	stqa rt, imm16	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 LocStor(LSA,16) ← RT	LS	ODD	6
Branch Relative	br imm16	PC ← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR	BR1	ODD	4
Branch Absolute	bra imm16	PC ← RepLeftBit(I16 || 0b00,32) & LSLR	BR1	ODD	4
Branch Relative and Set Link	brsl rt, imm16	$RT^{0:3}$ ← (PC + 4) & LSLR $RT^{4:15}$ ← 0 PC ← (PC + RepLeftBit(I16 || 0b00,32)) & LSLR	BR1	ODD	4
Branch Indirect	bi ra	PC ← $RA^{0:3}$ & LSLR & 0xFFFFFFFC	BR1	ODD	4

Branch If Not Zero Word	brnz rt, imm16	If $RT^{0:3}$ ≠ 0 then
		  PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
		else
		  PC ← (PC+4) & LSLR
			BR1	ODD	4

Branch if Zero Word	brz rt, imm16	If $RT^{0:3}$ = 0 then
		  PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
		else
		  PC ← (PC+4) & LSLR
			BR1	ODD	4

SystemVerilog Codebase

```
module tb_InstructionFetch();
    logic           clk, reset;
    logic[0:31] ins_mem[0:255];
    logic[0:31] instr[0:255];
    logic read_enable,stall;
    logic[7:0]   pc;


    IF ins_fetch(clk, reset, instr,pc, read_enable);


    initial
        clk = 0;

    always begin
        #5 clk = ~clk;
    end
    initial begin
        $readmemb("./compiler/out", ins_mem);
        for(integer i=0;i<256;i++) begin
            $display("PC %d %b", i, ins_mem[i]);
        end
        #1; reset =1;
        @(posedge clk); #1; reset = 1;
        @(posedge clk); #1; reset = 0;



        #300;$stop;
    end
    always @(posedge clk) begin
        if(read_enable==1) begin
            $display($time,"TB: pc %d ",pc);
            instr[0:255] = ins_mem[(pc)+:256];
            for(integer i=0;i<256;i++) begin
                $display("PC %d %b", i, ins_mem[i+pc]);
            end
        end
    end

endmodule
```

```
module IF(clk, reset, ins_cache, pc, read_enable);

    input logic clk;
    input logic reset;

    input logic[0:31] ins_cache[0:255];  // Instruction line buffer , stores 64B instruction


    logic[0:31] instr_d[0:1];           // 2 instruction sent to DECODE stage
    logic stall;                        // incase of stall Instruction fetch
                                        //should stop fetching new instruction

    logic branch_taken;

    output logic read_enable;           // Used to signal that IF is ready to read next set of 64B instruction

    output logic[7:0]   pc;

    logic[7:0]  pc_wb;                  // pc_wb access as reset PC signal for IF to start new instr read position
    integer pc_check;                // used for checkpointing, to adjust the pc while reading from ins_cache

    localparam [0:31] NOP = 32'b01000000001000000000000000000000;
    localparam [0:31] LNOP = 32'b00000000001000000000000000000000;

    Decode decode(clk, reset, instr_d, pc, pc_wb, stall, branch_taken);

    always_comb begin : pc_counter
        if(reset == 1) begin
            pc_check = 0;
            read_enable =1;
        end
        else begin
                read_enable = 0;
        end
    end

    always_ff @(posedge clk) begin : fetch_instruction

        $display($time," IF: stall %d pc %d pc_wb %d  read_enable %d ",stall, pc, pc_wb, read_enable);

        if(reset == 1 ) begin
            pc<=0;

            instr_d[0]<=32'h0000;
            instr_d[1]<=32'h0000;

        end
        else begin


            // Use of stall is to stop IF fetching new instruction
            // This can be used in case of dual issue conflicts where decode
            // inserts no-op instruction
            // we use pc_wb to continue fetch new stream of instruction then onwards

            if( stall==0 ) begin
                // stall<=0;
                instr_d[0]<=ins_cache[pc];
                instr_d[1]<=ins_cache[pc+1];
                $display($time," IF: ins %b ins %b pc %d pc_wb %d read_enable %d ",ins_cache[pc], ins_cache[pc+1],pc,pc_wb, read_enable);
                $display($time," IF: ins %h ins %h pc %d pc_wb %d read_enable %d ",ins_cache[pc], ins_cache[pc+1],pc,pc_wb, read_enable);

```

```
63
64                  $display($time," IF: ins %b ins %b pc %d pc_wb %d read_enable %d
                    ",instr_d[0], instr_d[1],pc,pc_wb, read_enable);
65                  $display($time," IF: ins %h ins %h pc %d pc_wb %d read_enable %d
                    ",instr_d[0], instr_d[1],pc,pc_wb, read_enable);
66
67                  if (pc < 254)
68                      pc <= pc+2;
69
70
71              end
72              else begin
73                  if (branch_taken == 0) begin
74                      $display($time,"IF: pc update to pc_wb %d pc %d" ,pc_wb, pc);
75                      instr_d[0]<=ins_cache[pc_wb];
76                      instr_d[1]<=ins_cache[pc_wb+1];
77                  end
78                  else begin
79                      pc <= pc_wb & ~1;
80                      $display($time,"IF: pc update to pc_wb %d pc %d" ,pc_wb, pc);
81                      if (pc_wb & 256'h1) begin                // If branching to an odd
                        number instr
82                          instr_d[0]<=ins_cache[pc_wb-2];
83                          instr_d[1]<=0;
84                      end
85                      else begin
86                          instr_d[0]<=ins_cache[pc_wb-2];
87                          instr_d[1]<=ins_cache[pc_wb-1];
88                      end
89                  end
90              end
91          end
92      end
93
94  endmodule
```

```
1   module Decode(clk, reset, instr, pc, stall_pc, stall, branch_taken_reg);
2       input logic         clk, reset;
3       input logic [0:31]  instr[0:1];
4       logic [0:31]        instr_next[0:1], instr_dec[0:1], instr_next_reg[0:1];
5
6       logic [0:31]    instr_even, instr_odd, instr_odd_issue, instr_even_issue;   //Instr
        from decoder
7
8       //Signals for handling branches
9       logic [7:0]         pc_wb;                                  //New program counter
        for branch
10      output logic [7:0]  stall_pc;
11      output logic        stall;
12      output logic        branch_taken_reg;                           //Was branch taken?
13      logic               branch_taken;                               //Was branch taken?
14      logic               first_odd, first_odd_out;               //1 if odd instr is
        first in pair, 0 else; Used for branch flushing
15      logic               stall_var;
16      logic [7:0]         stall_pc_var;
17
18      input logic[7:0] pc;                            // PC for the current state
19
20      //Nets from decode logic
21      logic [2:0]     format_even, format_odd;                //Format of instr
22      logic [0:10]    op_even, op_odd;                        //Opcode of instr (used
        with format)
23      logic [1:0]     unit_even, unit_odd;                    //Destination unit of
        instr; Order of: FP, FX2, Byte, FX1 (Even); Perm, LS, Br (Odd)
24      logic [0:6]     rt_addr_even, rt_addr_odd;              //Destination register
        addresses
25      logic [0:17]    imm_even, imm_odd;                      //Full possible immediate
        value (used with format)
26      logic           reg_write_even, reg_write_odd;          //1 if instr will write to
        rt, else 0
27
28      logic           first_cyc;                              //Due to how finished is
        detected, workaround is needed to prevent flag after reset
29
30      logic [7:0]     pc_dec, pc_pipe;                                    //New program
        counter for stalls
31
32      localparam [0:31] NOP = 32'b01000000001000000000000000000000;
33      localparam [0:31] LNOP = 32'b00000000001000000000000000000000;
34
35      //Internal Signals for Handling RAW Hazards
36      logic [0:6]         rt_addr_delay_even, rt_addr_delay_odd;      //Destination
        register for rt_wb
37      logic               reg_write_delay_even, reg_write_delay_odd;  //Will rt_wb write
        to RegTable
38      logic               stall_first_raw, stall_second_raw;          // 1 if respective
        signal is to be stalled due to RAW hazard
39      logic               stall_first, stall_second;          // 1 if respective signal
        is to be stalled due to RAW or structural hazard
40
41      logic [6:0][0:6]    rt_addr_delay_fp1;      //Destination register for rt_wb
42      logic [6:0]         reg_write_delay_fp1;    //Will rt_wb write to RegTable
43      logic [6:0]         int_delay_fp1;          //Will fp1 write an int result
44
45      logic [3:0][0:6]    rt_addr_delay_fx2;      //Destination register for rt_wb
46      logic [3:0]         reg_write_delay_fx2;    //Will rt_wb write to RegTable
47
48      logic [3:0][0:6]    rt_addr_delay_b1;       //Destination register for rt_wb
49      logic [3:0]         reg_write_delay_b1;     //Will rt_wb write to RegTable
50
51      logic [1:0][0:6]    rt_addr_delay_fx1;      //Destination register for rt_wb
52      logic [1:0]         reg_write_delay_fx1;    //Will rt_wb write to RegTable
53
54      logic [3:0][0:6]    rt_addr_delay_p1;       //Destination register for rt_wb
55      logic [3:0]         reg_write_delay_p1;     //Will rt_wb write to RegTable
```

```
56
57      logic [5:0][0:6]    rt_addr_delay_ls1;      //Destination register for rt_wb
58      logic [5:0]         reg_write_delay_ls1;    //Will rt_wb write to RegTable
59
60      typedef struct {
61          logic [0:31]    instr;
62          logic [0:10]    op;
63          logic           reg_write;
64          logic           even_valid, odd_valid;          //Is even/odd instr a valid
            instr?
65          logic [0:17]    imm;
66          logic [0:6]     rt_addr;
67          logic [1:0]     unit;
68          logic [2:0]     format;
69          logic [0:6]     ra_addr, rb_addr, rc_addr;
70          logic           ra_valid, rb_valid, rc_valid;   //Is ra/rb/rc read in this instr?
71      } op_code;
72
73      op_code first, second;
74
75      Pipes pipe(.clk(clk), .reset(reset), .pc(pc_pipe),
76          .instr_even(instr_even), .instr_odd(instr_odd),
77          .pc_wb(pc_wb), .branch_taken(branch_taken),
78          .op_even(op_even), .op_odd(op_odd),
79          .unit_even(unit_even), .unit_odd(unit_odd),
80          .rt_addr_even(rt_addr_even), .rt_addr_odd(rt_addr_odd),
81          .format_even(format_even), .format_odd(format_odd),
82          .imm_even(imm_even), .imm_odd(imm_odd),
83          .reg_write_even(reg_write_even), .reg_write_odd(reg_write_odd),
            .first_odd(first_odd_out),
84          .rt_addr_delay_even(rt_addr_delay_even),
            .reg_write_delay_even(reg_write_delay_even),
            .rt_addr_delay_odd(rt_addr_delay_odd), .reg_write_delay_odd(reg_write_delay_odd),
85          .rt_addr_delay_fp1(rt_addr_delay_fp1),
            .reg_write_delay_fp1(reg_write_delay_fp1), .int_delay_fp1(int_delay_fp1),
            .rt_addr_delay_fx2(rt_addr_delay_fx2), .reg_write_delay_fx2(reg_write_delay_fx2),
86          .rt_addr_delay_b1(rt_addr_delay_b1), .reg_write_delay_b1(reg_write_delay_b1),
            .rt_addr_delay_fx1(rt_addr_delay_fx1), .reg_write_delay_fx1(reg_write_delay_fx1),
87          .rt_addr_delay_p1(rt_addr_delay_p1), .reg_write_delay_p1(reg_write_delay_p1),
            .rt_addr_delay_ls1(rt_addr_delay_ls1), .reg_write_delay_ls1(reg_write_delay_ls1)
88          );
89
90      always_ff @( posedge clk ) begin : decode_op
91
92          if (first.even_valid) begin
93              instr_even <= first.instr;
94              op_even <= first.op;
95              reg_write_even <= first.reg_write;
96              imm_even <= first.imm;
97              rt_addr_even <= first.rt_addr;
98              unit_even <= first.unit;
99              format_even <= first.format;
100             first_odd_out <= 0;
101         end
102         else if (second.even_valid) begin
103             instr_even <= second.instr;
104             op_even <= second.op;
105             reg_write_even <= second.reg_write;
106             imm_even <= second.imm;
107             rt_addr_even <= second.rt_addr;
108             unit_even <= second.unit;
109             format_even <= second.format;
110         end
111         else begin
112             instr_even <= 0;
113             op_even <= 0;
114             reg_write_even <= 0;
115             imm_even <= 0;
116             rt_addr_even <= 0;
```

```
117             unit_even <= 0;
118             format_even <= 0;
119         end
120 
121         if (first.odd_valid) begin
122             instr_odd <= first.instr;
123             op_odd <= first.op;
124             reg_write_odd <= first.reg_write;
125             imm_odd <= first.imm;
126             rt_addr_odd <= first.rt_addr;
127             unit_odd <= first.unit;
128             format_odd <= first.format;
129             first_odd_out <= 1;
130         end
131         else if (second.odd_valid) begin
132             instr_odd <= second.instr;
133             op_odd <= second.op;
134             reg_write_odd <= second.reg_write;
135             imm_odd <= second.imm;
136             rt_addr_odd <= second.rt_addr;
137             unit_odd <= second.unit;
138             format_odd <= second.format;
139             first_odd_out <= 0;
140         end
141         else begin
142             instr_odd <= 0;
143             op_odd <= 0;
144             reg_write_odd <= 0;
145             imm_odd <= 0;
146             rt_addr_odd <= 0;
147             unit_odd <= 0;
148             format_odd <= 0;
149         end
150 
151         instr_next_reg <= instr_next;
152         stall <= stall_var;
153         stall_pc <= stall_pc_var;
154         branch_taken_reg <= branch_taken;
155         pc_pipe <= pc_dec;
156 
157         first_cyc <= reset;     //flag is always high after reset and low otherwise
158     end
159 
160 
161     //Decode logic
162     always_comb begin
163 
164         stall_first = 0;
165         stall_second = 0;
166 
167         $display("=======================================");
168         $display($time," New Decode: Sensitivity list");
169         $display("pc: %d, reset: %b, branch_taken: %b, finished: %b, stall: %b", pc,
        reset, branch_taken, 0, stall);
170         $display("instr_next[0]:        %b, instr_next[1]:      %b, ", instr_next[0],
        instr_next[1]);
171         $display("instr_next_reg[0]:    %b, instr_next_reg[1]:  %b, ",
        instr_next_reg[0], instr_next_reg[1]);
172         $display("instr[0]:             %b, instr[1]:           %b", instr[0], instr[1]);
173         $display("=======================================");
174 
175         if (reset == 1) begin
176             stall_var = 0;
177             stall_pc_var = 0;
178             first_odd = 0;
179             first = check_one(0);
180             second = check_one(0);
181             instr_next[0] = 0;
182             instr_next[1] = 0;
```

```
183             end
184             else begin
185                 if (branch_taken == 1) begin
186                     $display($time," New Decode: Branch taken, jumping to addr: %d", pc_wb);
187                     stall_pc_var = pc_wb+2;
188                     stall_var = 1;
189                     instr_next[0] = 0;
190                     instr_next[1] = 0;
191                     first = check_one(0);
192                     second = check_one(0);
193                 end
194                 else begin
195                     if (stall == 1) begin
196                         instr_dec[0] = instr_next_reg[0];
197                         instr_dec[1] = instr_next_reg[1];
198                         stall_var = 0;
199                         pc_dec = stall_pc;
200                         $display($time," New Decode: Choosing queued instr");
201                     end
202                     else begin
203                         instr_dec[0] = instr[0];
204                         instr_dec[1] = instr[1];
205                         pc_dec = pc;
206                         $display($time," New Decode: Choosing new instr");
207                     end
208 
209                     if ((instr_dec[0] != 0)) begin
210                         first = check_one(instr_dec[0]);    //Checking first instr
211 
212                         if (first.ra_valid) begin
213                             for (int i = 0; i <= 4; i++) begin
214                                 if ((first.ra_addr == rt_addr_delay_fp1[i]) &&
                                    (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
215                                     stall_first = 1;
216                                 end
217 
218                                 if ((i < 4) &&
219                                     (((first.ra_addr == rt_addr_delay_ls1[i]) &&
                                     reg_write_delay_ls1[i]) ||
220                                     ((first.ra_addr == rt_addr_delay_fp1[i]) &&
                                     reg_write_delay_fp1[i]))) begin
221                                     stall_first = 1;
222                                 end
223 
224                                 if ((i < 2) &&
225                                     (((first.ra_addr == rt_addr_delay_fx2[i]) &&
                                     reg_write_delay_fx2[i]) ||
226                                     ((first.ra_addr == rt_addr_delay_b1[i]) &&
                                     reg_write_delay_b1[i]) ||
227                                     ((first.ra_addr == rt_addr_delay_p1[i]) &&
                                     reg_write_delay_p1[i]))) begin
228                                     stall_first = 1;
229                                 end
230                             end
231                             if ((first.ra_addr == rt_addr_delay_even) &&
                            reg_write_delay_even) begin
232                                 stall_first = 1;
233                             end
234                             if ((first.ra_addr == rt_addr_delay_odd) &&
                            reg_write_delay_odd) begin
235                                 stall_first = 1;
236                             end
237                         end
238                         if (first.rb_valid) begin
239                             for (int i = 0; i <= 4; i++) begin
240                                 if ((first.rb_addr == rt_addr_delay_fp1[i]) &&
                                    (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
241                                     stall_first = 1;
242                                 end
```

```
243
244                         if ((i < 4) &&
245                             (((first.rb_addr == rt_addr_delay_ls1[i]) &&
                                reg_write_delay_ls1[i]) ||
246                             ((first.rb_addr == rt_addr_delay_fp1[i]) &&
                                reg_write_delay_fp1[i]))) begin
247                             stall_first = 1;
248                         end
249
250                         if ((i < 2) &&
251                             (((first.rb_addr == rt_addr_delay_fx2[i]) &&
                                reg_write_delay_fx2[i]) ||
252                             ((first.rb_addr == rt_addr_delay_b1[i]) &&
                                reg_write_delay_b1[i]) ||
253                             ((first.rb_addr == rt_addr_delay_p1[i]) &&
                                reg_write_delay_p1[i]))) begin
254                             stall_first = 1;
255                         end
256                     end
257                     if ((first.rb_addr == rt_addr_delay_even) &&
                        reg_write_delay_even) begin
258                         stall_first = 1;
259                     end
260                     if ((first.rb_addr == rt_addr_delay_odd) &&
                        reg_write_delay_odd) begin
261                         stall_first = 1;
262                     end
263                 end
264                 if (first.rc_valid) begin
265                     for (int i = 0; i <= 4; i++) begin
266                         if ((first.rc_addr == rt_addr_delay_fp1[i]) &&
                            (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
267                             stall_first = 1;
268                         end
269
270                         if ((i < 4) &&
271                             (((first.rc_addr == rt_addr_delay_ls1[i]) &&
                                reg_write_delay_ls1[i]) ||
272                             ((first.rc_addr == rt_addr_delay_fp1[i]) &&
                                reg_write_delay_fp1[i]))) begin
273                             stall_first = 1;
274                         end
275
276                         if ((i < 2) &&
277                             (((first.rc_addr == rt_addr_delay_fx2[i]) &&
                                reg_write_delay_fx2[i]) ||
278                             ((first.rc_addr == rt_addr_delay_b1[i]) &&
                                reg_write_delay_b1[i]) ||
279                             ((first.rc_addr == rt_addr_delay_p1[i]) &&
                                reg_write_delay_p1[i]))) begin
280                             stall_first = 1;
281                         end
282                     end
283                     if ((first.rc_addr == rt_addr_delay_even) &&
                        reg_write_delay_even) begin
284                         stall_first = 1;
285                     end
286                     if ((first.rc_addr == rt_addr_delay_odd) &&
                        reg_write_delay_odd) begin
287                         stall_first = 1;
288                     end
289                 end
290             end
291             else begin
292                 first = check_one(0);
293             end
294
295             if ((instr_dec[1] != 0)) begin
296                 second = check_one(instr_dec[1]);
```

```
297
298                     if ((second.even_valid && first.even_valid) || (second.odd_valid &&
                        first.odd_valid)) begin     //Same pipe, structural hazard
299                         stall_second = 1;
300                         $display($time," New Decode: Same pipe hazard found for second
                            instr");
301                     end
302                     else if (first.reg_write && second.reg_write && (first.rt_addr ==
                        second.rt_addr)) begin            //Same dest, data hazard (WAW)
303                         stall_second = 1;
304                         $display($time," New Decode: Same destination hazard found for
                            second instr");
305                     end
306                     else begin
307                         if (second.ra_valid) begin
308                             for (int i = 0; i <= 4; i++) begin
309                                 if ((second.ra_addr == rt_addr_delay_fp1[i]) &&
                                    (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
310                                     stall_second = 1;
311                                     $display($time," New Decode: RAW hazard found for
                                        second instr ra");
312                                 end
313
314                                 if ((i < 4) &&
315                                     (((second.ra_addr == rt_addr_delay_ls1[i]) &&
                                        reg_write_delay_ls1[i]) ||
316                                     ((second.ra_addr == rt_addr_delay_fp1[i]) &&
                                        reg_write_delay_fp1[i]))) begin
317                                     stall_second = 1;
318                                     $display($time," New Decode: RAW hazard found for
                                        second instr ra");
319                                 end
320
321                                 if ((i < 2) &&
322                                     (((second.ra_addr == rt_addr_delay_fx2[i]) &&
                                        reg_write_delay_fx2[i]) ||
323                                     ((second.ra_addr == rt_addr_delay_b1[i]) &&
                                        reg_write_delay_b1[i]) ||
324                                     ((second.ra_addr == rt_addr_delay_p1[i]) &&
                                        reg_write_delay_p1[i]))) begin
325                                     stall_second = 1;
326                                     $display($time," New Decode: RAW hazard found for
                                        second instr ra");
327                                 end
328                             end
329                             if ((second.ra_addr == rt_addr_delay_even) &&
                                reg_write_delay_even) begin
330                                 stall_second = 1;
331                                 $display($time," New Decode: RAW hazard found for
                                    second instr ra");
332                             end
333                             if ((second.ra_addr == rt_addr_delay_odd) &&
                                reg_write_delay_odd) begin
334                                 stall_second = 1;
335                                 $display($time," New Decode: RAW hazard found for
                                    second instr ra");
336                             end
337                             if ((second.ra_addr == first.rt_addr) && first.reg_write)
                                begin
338                                 stall_second = 1;
339                                 $display($time," New Decode: RAW hazard found for
                                    second instr ra");
340                             end
341                         end
342                         if (second.rb_valid) begin
343                             for (int i = 0; i <= 4; i++) begin
344                                 if ((second.rb_addr == rt_addr_delay_fp1[i]) &&
                                    (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
345                                     stall_second = 1;
```

```
346                             $display($time," New Decode: RAW hazard found for
                                second instr rb");
347                         end
348
349                         if ((i < 4) &&
350                             (((second.rb_addr == rt_addr_delay_ls1[i]) &&
                                reg_write_delay_ls1[i]) ||
351                             ((second.rb_addr == rt_addr_delay_fp1[i]) &&
                                reg_write_delay_fp1[i]))) begin
352                             stall_second = 1;
353                             $display($time," New Decode: RAW hazard found for
                                second instr rb");
354                         end
355
356                         if ((i < 2) &&
357                             (((second.rb_addr == rt_addr_delay_fx2[i]) &&
                                reg_write_delay_fx2[i]) ||
358                             ((second.rb_addr == rt_addr_delay_b1[i]) &&
                                reg_write_delay_b1[i]) ||
359                             ((second.rb_addr == rt_addr_delay_p1[i]) &&
                                reg_write_delay_p1[i]))) begin
360                             stall_second = 1;
361                             $display($time," New Decode: RAW hazard found for
                                second instr rb");
362                         end
363                     end
364                     if ((second.rb_addr == rt_addr_delay_even) &&
                        reg_write_delay_even) begin
365                         stall_second = 1;
366                         $display($time," New Decode: RAW hazard found for
                            second instr rb");
367                     end
368                     if ((second.rb_addr == rt_addr_delay_odd) &&
                        reg_write_delay_odd) begin
369                         stall_second = 1;
370                         $display($time," New Decode: RAW hazard found for
                            second instr rb");
371                     end
372                     if ((second.rb_addr == first.rt_addr) && first.reg_write)
                        begin
373                         stall_second = 1;
374                         $display($time," New Decode: RAW hazard found for
                            second instr rb");
375                     end
376                 end
377                 if (second.rc_valid) begin
378                     for (int i = 0; i <= 4; i++) begin
379                         if ((second.rc_addr == rt_addr_delay_fp1[i]) &&
                            (reg_write_delay_fp1[i] && int_delay_fp1[i])) begin
380                             stall_second = 1;
381                             $display($time," New Decode: RAW hazard found for
                                second instr rc");
382                         end
383
384                         if ((i < 4) &&
385                             (((second.rc_addr == rt_addr_delay_ls1[i]) &&
                                reg_write_delay_ls1[i]) ||
386                             ((second.rc_addr == rt_addr_delay_fp1[i]) &&
                                reg_write_delay_fp1[i]))) begin
387                             stall_second = 1;
388                             $display($time," New Decode: RAW hazard found for
                                second instr rc");
389                         end
390
391                         if ((i < 2) &&
392                             (((second.rc_addr == rt_addr_delay_fx2[i]) &&
                                reg_write_delay_fx2[i]) ||
393                             ((second.rc_addr == rt_addr_delay_b1[i]) &&
                                reg_write_delay_b1[i]) ||
```

```
394                                     ((second.rc_addr == rt_addr_delay_p1[i]) &&
                                        reg_write_delay_p1[i]))) begin
395                                     stall_second = 1;
396                                     $display($time," New Decode: RAW hazard found for
                                        second instr rc");
397                                 end
398                             end
399                             if ((second.rc_addr == rt_addr_delay_even) &&
                                reg_write_delay_even) begin
400                                 stall_second = 1;
401                                 $display($time," New Decode: RAW hazard found for
                                    second instr rc");
402                             end
403                             if ((second.rc_addr == rt_addr_delay_odd) &&
                                reg_write_delay_odd) begin
404                                 stall_second = 1;
405                                 $display($time," New Decode: RAW hazard found for
                                    second instr rc");
406                             end
407                             if ((second.rc_addr == first.rt_addr) && first.reg_write)
                                begin
408                                 stall_second = 1;
409                                 $display($time," New Decode: RAW hazard found for
                                    second instr rc");
410                             end
411                         end
412                     end
413                 end
414                 else begin
415                     second = check_one(0);
416                 end
417 
418                 if (stall_first) begin
419                     $display($time," New Decode: RAW hazard found for first instr");
420                     stall_pc_var = pc_dec;
421                     stall_var = 1;
422                     instr_next[0] = instr_dec[0];
423                     instr_next[1] = instr_dec[1];
424                     first = check_one(0);
425                     second = check_one(0);
426                 end
427                 else if (stall_second) begin
428                     stall_pc_var = pc_dec;
429                     stall_var = 1;
430                     instr_next[0] = 0;
431                     instr_next[1] = instr_dec[1];
432                     second = check_one(0);
433                 end
434                 else begin
435                     stall_var = 0;
436                 end
437             end
438 
439             $display($time, "finished_var: %b, stall_var: %b", 0, stall_var);
440         end
441 
442 
443     end
444 
445     function op_code check_one (input logic[0:31] instr);
446 
447         if(reset==1) begin
448             check_one.op = 0;
449             check_one.even_valid = 1;
450             check_one.odd_valid = 1;
451         end
452         $display($time," instr %b %b",instr, instr[0:10]);
453                                                             //Even decoding
454         check_one.reg_write = 1;
```

```
455             check_one.rt_addr = instr[25:31];
456             check_one.ra_addr = instr[18:24];
457             check_one.rb_addr = instr[11:17];
458             check_one.rc_addr = instr[25:31];
459             check_one.instr = instr;
460             check_one.even_valid = 1;
461             if (instr == 0) begin                               //alternate nop
462                 check_one.format = 0;
463                 check_one.ra_valid = 0;
464                 check_one.rb_valid = 0;
465                 check_one.rc_valid = 0;
466                 check_one.op = 0;
467                 check_one.unit = 0;
468                 check_one.rt_addr = 0;
469                 check_one.imm = 0;
470                 check_one.reg_write = 0;
471                 check_one.even_valid = 0;
472             end                                                     //RRR-type
473             else if (instr[0:3] == 4'b1100) begin               //mpya
474                 check_one.format = 1;
475                 check_one.ra_valid = 1;
476                 check_one.rb_valid = 1;
477                 check_one.rc_valid = 1;
478                 check_one.op = 4'b1100;
479                 check_one.unit = 0;
480                 check_one.rt_addr = instr[4:10];
481             end
482             else if (instr[0:3] == 4'b1110) begin               //fma
483                 check_one.format = 1;
484                 check_one.ra_valid = 1;
485                 check_one.rb_valid = 1;
486                 check_one.rc_valid = 1;
487                 check_one.op = 4'b1110;
488                 check_one.unit = 0;
489                 check_one.rt_addr = instr[4:10];
490             end
491             else if (instr[0:3] == 4'b1111) begin               //fms
492                 check_one.format = 1;
493                 check_one.ra_valid = 1;
494                 check_one.rb_valid = 1;
495                 check_one.rc_valid = 1;
496                 check_one.op = 4'b1111;
497                 check_one.unit = 0;
498                 check_one.rt_addr = instr[4:10];
499             end                                                     //RI18-type
500             else if (instr[0:6] == 7'b0100001) begin            //ila
501                 check_one.format = 6;
502                 check_one.ra_valid = 0;
503                 check_one.rb_valid = 0;
504                 check_one.rc_valid = 0;
505                 check_one.op = 7'b0100001;
506                 check_one.unit = 3;
507                 check_one.imm = $signed(instr[7:24]);
508             end                                                     //RI8-type
509             else if (instr[0:9] == 10'b0111011000) begin        //cflts
510                 check_one.format = 3;
511                 check_one.ra_valid = 1;
512                 check_one.rb_valid = 0;
513                 check_one.rc_valid = 0;
514                 check_one.op = 10'b0111011000;
515                 check_one.unit = 0;
516                 check_one.imm = $signed(instr[10:17]);
517             end
518             else if (instr[0:9] == 10'b0111011001) begin        //cfltu
519                 check_one.format = 3;
520                 check_one.ra_valid = 1;
521                 check_one.rb_valid = 0;
522                 check_one.rc_valid = 0;
523                 check_one.op = 10'b0111011001;
```

```
524             check_one.unit = 0;
525             check_one.imm = $signed(instr[10:17]);
526         end                                                 //RI16-type
527         else if (instr[0:8] == 9'b010000011) begin      //ilh
528             check_one.format = 5;
529             check_one.ra_valid = 0;
530             check_one.rb_valid = 0;
531             check_one.rc_valid = 0;
532             check_one.op = 9'b010000011;
533             check_one.unit = 3;
534             check_one.imm = $signed(instr[9:24]);
535         end
536         else if (instr[0:8] == 9'b010000001) begin      //il
537             check_one.format = 5;
538             check_one.ra_valid = 0;
539             check_one.rb_valid = 0;
540             check_one.rc_valid = 0;
541             check_one.op = 9'b010000001;
542             check_one.unit = 3;
543             check_one.imm = $signed(instr[9:24]);
544         end
545         else if (instr[0:8] == 9'b010000010) begin      //ilhu
546             check_one.format = 5;
547             check_one.ra_valid = 0;
548             check_one.rb_valid = 0;
549             check_one.rc_valid = 0;
550             check_one.op = 9'b010000010;
551             check_one.unit = 3;
552             check_one.imm = $signed(instr[9:24]);
553         end
554         else if (instr[0:8] == 9'b011000001) begin      //iohl
555             check_one.format = 5;
556             check_one.ra_valid = 0;
557             check_one.rb_valid = 0;
558             check_one.rc_valid = 0;
559             check_one.op = 9'b011000001;
560             check_one.unit = 3;
561             check_one.imm = $signed(instr[9:24]);
562         end
563         else begin
564             check_one.format = 4;               //RI10-type
565             check_one.ra_valid = 1;
566             check_one.rb_valid = 0;
567             check_one.rc_valid = 0;
568             check_one.imm = $signed(instr[8:17]);
569             case(instr[0:7])
570                 8'b01110100 : begin         //mpyi
571                     check_one.op = 8'b01110100;
572                     check_one.unit = 0;
573                 end
574                 8'b01110101 : begin         //mpyui
575                     check_one.op = 8'b01110101;
576                     check_one.unit = 0;
577                 end
578                 8'b00011101 : begin         //ahi
579                     check_one.op = 8'b00011101;
580                     check_one.unit = 3;
581                 end
582                 8'b00011100 : begin         //ai
583                     check_one.op = 8'b00011100;
584                     check_one.unit = 3;
585                 end
586                 8'b00001101 : begin         //sfhi
587                     check_one.op = 8'b00001101;
588                     check_one.unit = 3;
589                 end
590                 8'b00001100 : begin         //sfi
591                     check_one.op = 8'b00001100;
592                     check_one.unit = 3;
```

```
593                 end
594                 8'b00010110 : begin         //andbi
595                     check_one.op = 8'b00010110;
596                     check_one.unit = 3;
597                 end
598                 8'b00010101 : begin         //andhi
599                     check_one.op = 8'b00010101;
600                     check_one.unit = 3;
601                 end
602                 8'b00010100 : begin         //andi
603                     check_one.op = 8'b00010100;
604                     check_one.unit = 3;
605                 end
606                 8'b00000110 : begin         //orbi
607                     check_one.op = 8'b00000110;
608                     check_one.unit = 3;
609                 end
610                 8'b00000101 : begin         //orhi
611                     check_one.op = 8'b00000101;
612                     check_one.unit = 3;
613                 end
614                 8'b00000100 : begin         //ori
615                     check_one.op = 8'b00000100;
616                     check_one.unit = 3;
617                 end
618                 8'b01000110 : begin         //xorbi
619                     check_one.op = 8'b01000110;
620                     check_one.unit = 3;
621                 end
622                 8'b01000101 : begin         //xorhi
623                     check_one.op = 8'b01000101;
624                     check_one.unit = 3;
625                 end
626                 8'b01000100 : begin         //xori
627                     check_one.op = 8'b01000100;
628                     check_one.unit = 3;
629                 end
630                 8'b01111110 : begin         //ceqbi
631                     check_one.op = 8'b01111110;
632                     check_one.unit = 3;
633                 end
634                 8'b01111101 : begin         //ceqhi
635                     check_one.op = 8'b01111101;
636                     check_one.unit = 3;
637                 end
638                 8'b01111100 : begin         //ceqi
639                     check_one.op = 8'b01111100;
640                     check_one.unit = 3;
641                 end
642                 8'b01001110 : begin         //cgtbi
643                     check_one.op = 8'b01001110;
644                     check_one.unit = 3;
645                 end
646                 8'b01001101 : begin         //cgthi
647                     check_one.op = 8'b01001101;
648                     check_one.unit = 3;
649                 end
650                 8'b01001100 : begin         //cgti
651                     check_one.op = 8'b01001100;
652                     check_one.unit = 3;
653                 end
654                 8'b01011110 : begin         //clgtbi
655                     check_one.op = 8'b01011110;
656                     check_one.unit = 3;
657                 end
658                 8'b01011101 : begin         //clgthi
659                     check_one.op = 8'b01011101;
660                     check_one.unit = 3;
661                 end
```

```
                8'b01011100 : begin         //clgti
                    check_one.op = 8'b01011100;
                    check_one.unit = 3;
                end
                default : check_one.format = 7;
            endcase
            if (check_one.format == 7) begin
                check_one.format = 0;                   //RR-type
                check_one.ra_valid = 1;
                check_one.rb_valid = 1;
                check_one.rc_valid = 0;
                case(instr[0:10])
                    11'b01111000100 : begin         //mpy
                        check_one.op = 11'b01111000100;
                        check_one.unit = 0;
                    end
                    11'b01111001100 : begin         //mpyu
                        check_one.op = 11'b01111001100;
                        check_one.unit = 0;
                    end
                    11'b01111000101 : begin         //mpyh
                        check_one.op = 11'b01111000101;
                        check_one.unit = 0;
                    end
                    11'b01011000100 : begin         //fa
                        check_one.op = 11'b01011000100;
                        check_one.unit = 0;
                    end
                    11'b01011000101 : begin         //fs
                        check_one.op = 11'b01011000101;
                        check_one.unit = 0;
                    end
                    11'b01011000110 : begin         //fm
                        check_one.op = 11'b01011000110;
                        check_one.unit = 0;
                    end
                    11'b01111000010 : begin         //fceq
                        check_one.op = 11'b01111000010;
                        check_one.unit = 0;
                    end
                    11'b01011000010 : begin         //fcgt
                        check_one.op = 11'b01011000010;
                        check_one.unit = 0;
                    end
                    11'b00001011111 : begin         //shlh
                        check_one.op = 11'b00001011111;
                        check_one.unit = 1;
                    end
                    11'b00001011011 : begin         //shl
                        check_one.op = 11'b00001011011;
                        check_one.unit = 1;
                    end
                    11'b00001011100 : begin         //roth
                        check_one.op = 11'b00001011100;
                        check_one.unit = 1;
                    end
                    11'b00001011000 : begin         //rot
                        check_one.op = 11'b00001011000;
                        check_one.unit = 1;
                    end
                    11'b00001011101 : begin         //rothm
                        check_one.op = 11'b00001011101;
                        check_one.unit = 1;
                    end
                    11'b00001011001 : begin         //rotm
                        check_one.op = 11'b00001011001;
                        check_one.unit = 1;
                    end
                    11'b00001011110 : begin         //rotmah
```

```
731                         check_one.op = 11'b00001011110;
732                         check_one.unit = 1;
733                     end
734                     11'b00001011010 : begin         //rotma
735                         check_one.op = 11'b00001011010;
736                         check_one.unit = 1;
737                     end
738                     11'b01010110100 : begin         //cntb
739                         check_one.op = 11'b01010110100;
740                         check_one.unit = 2;
741                         check_one.rb_valid = 0;
742                     end
743                     11'b00011010011 : begin         //avgb
744                         check_one.op = 11'b00011010011;
745                         check_one.unit = 2;
746                     end
747                     11'b00001010011 : begin         //absdb
748                         check_one.op = 11'b00001010011;
749                         check_one.unit = 2;
750                     end
751                     11'b01001010011 : begin         //sumb
752                         $display("found  sumb");
753                         check_one.op = 11'b01001010011;
754                         check_one.unit = 2;
755                     end
756                     11'b00011001000 : begin         //ah
757                         check_one.op = 11'b00011001000;
758                         check_one.unit = 3;
759                     end
760                     11'b00011000000 : begin         //a
761                         check_one.op = 11'b00011000000;
762                         check_one.unit = 3;
763                     end
764                     11'b00001001000 : begin         //sfh
765                         check_one.op = 11'b00001001000;
766                         check_one.unit = 3;
767                     end
768                     11'b00001000000 : begin         //sf
769                         check_one.op = 11'b00001000000;
770                         check_one.unit = 3;
771                     end
772                     11'b00011000001 : begin         //and
773                         check_one.op = 11'b00011000001;
774                         check_one.unit = 3;
775                     end
776                     11'b00001000001 : begin         //or
777                         check_one.op = 11'b00001000001;
778                         check_one.unit = 3;
779                     end
780                     11'b01001000001 : begin         //xor
781                         check_one.op = 11'b01001000001;
782                         check_one.unit = 3;
783                     end
784                     11'b00011001001 : begin         //nand
785                         check_one.op = 11'b00011001001;
786                         check_one.unit = 3;
787                     end
788                     11'b01111010000 : begin         //ceqb
789                         check_one.op = 11'b01111010000;
790                         check_one.unit = 3;
791                     end
792                     11'b01111001000 : begin         //ceqh
793                         check_one.op = 11'b01111001000;
794                         check_one.unit = 3;
795                     end
796                     11'b01111000000 : begin         //ceq
797                         check_one.op = 11'b01111000000;
798                         check_one.unit = 3;
799                     end
```

```
                    11'b01001010000 : begin         //cgtb
                        check_one.op = 11'b01001010000;
                        check_one.unit = 3;
                    end
                    11'b01001001000 : begin         //cgth
                        check_one.op = 11'b01001001000;
                        check_one.unit = 3;
                    end
                    11'b01001000000 : begin         //cgt
                        check_one.op = 11'b01001000000;
                        check_one.unit = 3;
                    end
                    11'b01011010000 : begin         //clgtb
                        check_one.op = 11'b01011010000;
                        check_one.unit = 3;
                    end
                    11'b01011001000 : begin         //clgth
                        check_one.op = 11'b01011001000;
                        check_one.unit = 3;
                    end
                    11'b01011000000 : begin         //clgt
                        check_one.op = 11'b01011000000;
                        check_one.unit = 3;
                    end
                    11'b01000000001 : begin         //nop
                        check_one.op = 11'b01000000001;
                        check_one.unit = 0;
                        check_one.reg_write = 0;
                    end
                    default : check_one.format = 7;
                endcase
                if (check_one.format == 7) begin
                    check_one.format = 2;                   //RI7-type
                    check_one.ra_valid = 1;
                    check_one.rb_valid = 0;
                    check_one.rc_valid = 0;
                    check_one.imm = $signed(instr[11:17]);
                    case(instr[0:10])
                        11'b00001111011 : begin         //shli
                            check_one.op = 11'b00001111011;
                            check_one.unit = 1;
                        end
                        11'b00001111100 : begin         //rothi
                            check_one.op = 11'b00001111100;
                            check_one.unit = 1;
                        end
                        11'b00001111000 : begin         //roti
                            check_one.op = 11'b00001111000;
                            check_one.unit = 1;
                        end
                        11'b00001111110 : begin         //rotmahi
                            check_one.op = 11'b00001111110;
                            check_one.unit = 1;
                        end
                        11'b00001111010 : begin         //rotmai
                            check_one.op = 11'b00001111010;
                            check_one.unit = 1;
                        end
                        default begin
                            check_one.format = 0;
                            check_one.ra_valid = 0;
                            check_one.rb_valid = 0;
                            check_one.rc_valid = 0;
                            check_one.op = 0;
                            check_one.unit = 0;
                            check_one.rt_addr = 0;
                            check_one.imm = 0;
                            check_one.even_valid = 0;
                        end
```

```
                    endcase
                end
            end
        end

        if (check_one.even_valid == 0)
        begin                                                   //odd decoding
            check_one.rt_addr = instr[25:31];
            check_one.ra_addr = instr[18:24];
            check_one.rb_addr = instr[11:17];
            check_one.rc_addr = instr[25:31];
            check_one.reg_write = 1;
            check_one.odd_valid = 1;
            if (instr == 0) begin                               //alternate lnop
                check_one.format = 0;
                check_one.ra_valid = 0;
                check_one.rb_valid = 0;
                check_one.rc_valid = 0;
                check_one.op = 0;
                check_one.unit = 0;
                check_one.rt_addr = 0;
                check_one.imm = 0;
                check_one.reg_write = 0;
                check_one.odd_valid = 0;
            end                                                 //RI10-type
            else if (instr[0:7] == 8'b00110100) begin       //lqd
                check_one.format = 4;
                check_one.ra_valid = 1;
                check_one.rb_valid = 0;
                check_one.rc_valid = 0;
                check_one.op = 8'b00110100;
                check_one.unit = 1;
                check_one.imm = $signed(instr[8:17]);
            end
            else if (instr[0:7] == 8'b00100100) begin       //stqd
                check_one.format = 4;
                check_one.ra_valid = 1;
                check_one.rb_valid = 0;
                check_one.rc_valid = 1;
                check_one.op = 8'b00100100;
                check_one.unit = 1;
                check_one.imm = $signed(instr[8:17]);
                check_one.reg_write = 0;
            end
            else begin
                check_one.format = 5;                   //RI16-type
                check_one.ra_valid = 0;
                check_one.rb_valid = 0;
                check_one.rc_valid = 0;
                check_one.imm = $signed(instr[9:24]);
                case(instr[0:8])
                    9'b001100001 : begin        //lqa
                        check_one.op = 9'b001100001;
                        check_one.unit = 1;
                    end
                    9'b001000001 : begin        //stqa
                        check_one.op = 9'b001000001;
                        check_one.unit = 1;
                        check_one.reg_write = 0;
                        check_one.rc_valid = 1;
                    end
                    9'b001100100 : begin        //br
                        check_one.op = 9'b001100100;
                        check_one.unit = 2;
                        check_one.reg_write = 0;
                    end
                    9'b001100000 : begin        //bra
                        check_one.op = 9'b001100000;
                        check_one.unit = 2;
```

```
                        check_one.reg_write = 0;
                    end
                    9'b001100110 : begin        //brsl
                        check_one.op = 9'b001100110;
                        check_one.unit = 2;
                    end
                    9'b001000010 : begin        //brnz
                        check_one.op = 9'b001000010;
                        check_one.unit = 2;
                        check_one.reg_write = 0;
                        check_one.rc_valid = 1;
                    end
                    9'b001000000 : begin        //brz
                        check_one.op = 9'b001000000;
                        check_one.unit = 2;
                        check_one.reg_write = 0;
                        check_one.rc_valid = 1;
                    end
                    default : check_one.format = 7;
                endcase
                if (check_one.format == 7) begin
                    check_one.format = 0;                   //RR-type
                    check_one.ra_valid = 1;
                    check_one.rb_valid = 1;
                    check_one.rc_valid = 0;
                    //$display("check: instr[0:10] %b ",instr[0:10]);
                    case(instr[0:10])
                        11'b00111011011 : begin     //shlqbi
                            $display("shlqbi ");
                            check_one.op = 11'b00111011011;
                            check_one.unit = 0;
                        end
                        11'b00111011111 : begin     //shlqby
                            check_one.op = 11'b00111011111;
                            check_one.unit = 0;
                        end
                        11'b00111011000 : begin     //rotqbi
                            check_one.op = 11'b00111011000;
                            check_one.unit = 0;
                        end
                        11'b00111011100 : begin     //rotqby
                            check_one.op = 11'b00111011100;
                            check_one.unit = 0;
                        end
                        11'b00110110010 : begin     //gbb
                            check_one.op = 11'b00110110010;
                            check_one.unit = 0;
                            check_one.rb_valid = 0;
                        end
                        11'b00110110001 : begin     //gbh
                            check_one.op = 11'b00110110001;
                            check_one.unit = 0;
                            check_one.rb_valid = 0;
                        end
                        11'b00110110000 : begin     //gb
                            check_one.op = 11'b00110110000;
                            check_one.unit = 0;
                            check_one.rb_valid = 0;
                        end
                        11'b00111000100 : begin     //lqx
                            check_one.op = 11'b00111000100;
                            check_one.unit = 1;
                        end
                        11'b00101000100 : begin     //stqx
                            check_one.op = 11'b00101000100;
                            check_one.unit = 1;
                            check_one.reg_write = 0;
                            check_one.rc_valid = 1;
                        end
```

```
                        11'b00110101000 : begin     //bi
                            check_one.op = 11'b00110101000;
                            check_one.unit = 2;
                            check_one.reg_write = 0;
                            check_one.rb_valid = 0;
                        end
                        11'b00000000001 : begin     //lnop
                            check_one.op = 11'b00000000001;
                            check_one.unit = 0;
                            check_one.reg_write = 0;
                        end
                        default : check_one.format = 7;
                    endcase
                    if (check_one.format == 7) begin
                        check_one.format = 2;                   //RI7-type
                        check_one.ra_valid = 1;
                        check_one.rb_valid = 0;
                        check_one.rc_valid = 0;
                        check_one.imm = $signed(instr[11:17]);
                        case(instr[0:10])
                            11'b00111111011 : begin     //shlqbii
                                check_one.op = 11'b00111111011;
                                check_one.unit = 0;
                            end
                            11'b00111111111 : begin     //shlqbyi
                                check_one.op = 11'b00111111111;
                                check_one.unit = 0;
                            end
                            11'b00111111000 : begin     //rotqbii
                                check_one.op = 11'b00111111000;
                                check_one.unit = 0;
                            end
                            11'b00111111100 : begin     //rotqbyi
                                check_one.op = 11'b00111111100;
                                check_one.unit = 0;
                            end
                            default begin
                                check_one.format = 0;
                                check_one.ra_valid = 0;
                                check_one.rb_valid = 0;
                                check_one.rc_valid = 0;
                                check_one.op = 0;
                                check_one.unit = 0;
                                check_one.rt_addr = 0;
                                check_one.imm = 0;
                                check_one.odd_valid = 0;
                            end
                        endcase
                    end
                end
            end
        end
        else check_one.odd_valid = 0;
    endfunction

endmodule
```

```
1  module Pipes(clk, reset, instr_even, instr_odd, pc, pc_wb, branch_taken, op_even,
   op_odd, unit_even, unit_odd, rt_addr_even, rt_addr_odd, format_even, format_odd,
2      imm_even, imm_odd, reg_write_even, reg_write_odd, first_odd, rt_addr_delay_even,
       reg_write_delay_even, rt_addr_delay_odd, reg_write_delay_odd,
3      rt_addr_delay_fp1, reg_write_delay_fp1, int_delay_fp1, rt_addr_delay_fx2,
       reg_write_delay_fx2, rt_addr_delay_b1, reg_write_delay_b1, rt_addr_delay_fx1,
       reg_write_delay_fx1,
4      rt_addr_delay_p1, reg_write_delay_p1, rt_addr_delay_ls1, reg_write_delay_ls1);
5      input logic clk, reset;
6      input logic[0:31]   instr_even, instr_odd;              //Instr from decoder
7      input logic [7:0]   pc;                                 //Program counter from IF
       stage
8
9      //Nets from decode logic
10     input logic [2:0]   format_even, format_odd;            //Format of instr
11     input logic [0:10]  op_even, op_odd;                    //Opcode of instr (used
       with format)
12     input logic [1:0]   unit_even, unit_odd;                //Destination unit of
       instr; Order of: FP, FX2, Byte, FX1 (Even); Perm, LS, Br (Odd)
13     input logic [0:6]   rt_addr_even, rt_addr_odd;          //Destination register
       addresses
14     logic [0:127]       ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd;
       //Register values from RegTable
15     input logic [0:17]  imm_even, imm_odd;                  //Full possible immediate
       value (used with format)
16     input logic         reg_write_even, reg_write_odd;      //1 if instr will write to
       rt, else 0
17     input               first_odd;                          //1 if odd instr is first
       in pair, 0 else; Used for branch flushing
18
19     //Signals for writing back to RegTable
20     logic [0:127]   rt_even_wb, rt_odd_wb;                  //Values to be written back
       to RegTable
21     logic [0:6]     rt_addr_even_wb, rt_addr_odd_wb;        //Destination register
       addresses
22     logic           reg_write_even_wb, reg_write_odd_wb;    //1 if instr will write to
       rt, else 0
23
24     //Signals for forwarding logic
25     logic [6:0][0:127]  fw_even_wb, fw_odd_wb;              //Pipe shift registers of
       values ready to be forwarded
26     logic [6:0][0:6]    fw_addr_even_wb, fw_addr_odd_wb;    //Destinations of values to
       be forwarded
27     logic [6:0]         fw_write_even_wb, fw_write_odd_wb;  //Will forwarded values be
       writted to reg?
28     logic [0:127]       ra_even_fwd, rb_even_fwd, rc_even_fwd, ra_odd_fwd, rb_odd_fwd,
       rt_st_odd_fwd;   //Updated input values
29
30     //Signals for handling branches
31     output logic [7:0]  pc_wb;                              //New program counter for
       branch
32     output logic        branch_taken;                       //Was branch taken?
33     logic               branch_kill;                        //If branch is taken and
       branch instr is first in pair, kill twin instr
34     logic [2:0]         format_even_live;                   //Format of instr, only
       valid if branch not taken by first instr
35     logic [0:10]        op_even_live;                       //Opcode of instr, only
       valid if branch not taken by first instr
36
37     //Internal Signals for Handling RAW Hazards
38     output logic [0:6]  rt_addr_delay_even, rt_addr_delay_odd;          //Destination
       register for rt_wb
39     output logic        reg_write_delay_even, reg_write_delay_odd;      //Will rt_wb
       write to RegTable
40
41     output logic [6:0][0:6] rt_addr_delay_fp1;      //Destination register for rt_wb
42     output logic [6:0]      reg_write_delay_fp1;    //Will rt_wb write to RegTable
43     output logic [6:0]      int_delay_fp1;          //Will fp1 write an int result
44
```

```
45      output logic [3:0][0:6] rt_addr_delay_fx2;      //Destination register for rt_wb
46      output logic [3:0]      reg_write_delay_fx2;    //Will rt_wb write to RegTable
47
48      output logic [3:0][0:6] rt_addr_delay_b1;       //Destination register for rt_wb
49      output logic [3:0]      reg_write_delay_b1;     //Will rt_wb write to RegTable
50
51      output logic [1:0][0:6] rt_addr_delay_fx1;      //Destination register for rt_wb
52      output logic [1:0]      reg_write_delay_fx1;    //Will rt_wb write to RegTable
53
54      output logic [3:0][0:6] rt_addr_delay_p1;       //Destination register for rt_wb
55      output logic [3:0]      reg_write_delay_p1;     //Will rt_wb write to RegTable
56
57      output logic [5:0][0:6] rt_addr_delay_ls1;      //Destination register for rt_wb
58      output logic [5:0]      reg_write_delay_ls1;    //Will rt_wb write to RegTable
59
60      RegisterTable rf(.clk(clk), .reset(reset), .instr_even(instr_even),
        .instr_odd(instr_odd),
61          .ra_even(ra_even), .rb_even(rb_even), .rc_even(rc_even), .ra_odd(ra_odd),
            .rb_odd(rb_odd), .rt_st_odd(rt_st_odd), .rt_addr_even(rt_addr_even_wb),
62          .rt_addr_odd(rt_addr_odd_wb), .rt_even(rt_even_wb), .rt_odd(rt_odd_wb),
            .reg_write_even(reg_write_even_wb), .reg_write_odd(reg_write_odd_wb));
63
64      EvenPipe ev(.clk(clk), .reset(reset), .op(op_even_live), .format(format_even_live),
        .unit(unit_even), .rt_addr(rt_addr_even), .ra(ra_even_fwd), .rb(rb_even_fwd),
        .rc(rc_even_fwd),
65          .imm(imm_even), .reg_write(reg_write_even), .rt_wb(rt_even_wb),
            .rt_addr_wb(rt_addr_even_wb), .reg_write_wb(reg_write_even_wb),
            .branch_taken(branch_taken),
66          .fw_wb(fw_even_wb), .fw_addr_wb(fw_addr_even_wb), .fw_write_wb(fw_write_even_wb),
67          .rt_addr_delay_fp1(rt_addr_delay_fp1),
            .reg_write_delay_fp1(reg_write_delay_fp1), .int_delay_fp1(int_delay_fp1),
68          .rt_addr_delay_fx2(rt_addr_delay_fx2),
            .reg_write_delay_fx2(reg_write_delay_fx2), .rt_addr_delay_b1(rt_addr_delay_b1),
69          .reg_write_delay_b1(reg_write_delay_b1), .rt_addr_delay_fx1(rt_addr_delay_fx1),
            .reg_write_delay_fx1(reg_write_delay_fx1));
70
71      OddPipe od(.clk(clk), .reset(reset), .op(op_odd), .format(format_odd),
        .unit(unit_odd), .rt_addr(rt_addr_odd), .ra(ra_odd_fwd), .rb(rb_odd_fwd),
72          .rt_st(rt_st_odd_fwd), .imm(imm_odd), .reg_write(reg_write_odd), .pc_in(pc),
            .rt_wb(rt_odd_wb), .rt_addr_wb(rt_addr_odd_wb), .reg_write_wb(reg_write_odd_wb),
73          .pc_wb(pc_wb), .branch_taken(branch_taken), .fw_wb(fw_odd_wb),
            .fw_addr_wb(fw_addr_odd_wb), .fw_write_wb(fw_write_odd_wb), .first(first_odd),
            .branch_kill(branch_kill),
74          .rt_addr_delay_p1(rt_addr_delay_p1), .reg_write_delay_p1(reg_write_delay_p1),
75          .rt_addr_delay_ls1(rt_addr_delay_ls1),
            .reg_write_delay_ls1(reg_write_delay_ls1));
76
77      Forward fwd(.clk(clk), .reset(reset), .instr_even(instr_even),
        .instr_odd(instr_odd), .ra_even(ra_even), .rb_even(rb_even), .rc_even(rc_even),
78          .ra_odd(ra_odd), .rb_odd(rb_odd), .rt_st_odd(rt_st_odd),
            .ra_even_fwd(ra_even_fwd), .rb_even_fwd(rb_even_fwd), .rc_even_fwd(rc_even_fwd),
79          .ra_odd_fwd(ra_odd_fwd), .rb_odd_fwd(rb_odd_fwd),
            .rt_st_odd_fwd(rt_st_odd_fwd), .fw_even_wb(fw_even_wb),
            .fw_addr_even_wb(fw_addr_even_wb),
80          .fw_write_even_wb(fw_write_even_wb), .fw_odd_wb(fw_odd_wb),
            .fw_addr_odd_wb(fw_addr_odd_wb), .fw_write_odd_wb(fw_write_odd_wb));
81
82
83      always_comb begin
84          if (branch_kill == 0) begin         //If branch is taken and branch is first
            instr in pair, kill corresponding even instr
85              format_even_live = format_even;
86              op_even_live = op_even;
87          end
88          else begin
89              format_even_live = 0;
90              op_even_live = 0;
91          end
92
```

```
93            rt_addr_delay_even = rt_addr_even;
94            reg_write_delay_even = reg_write_even;
95            rt_addr_delay_odd = rt_addr_odd;
96            reg_write_delay_odd = reg_write_odd;
97        end
98
99    endmodule
```

```
1   module RegisterTable(clk, reset, instr_even, instr_odd, ra_even, rb_even, rc_even,
    ra_odd, rb_odd,
2           rt_st_odd, rt_addr_even, rt_addr_odd, rt_even, rt_odd, reg_write_even,
            reg_write_odd);
3       input                   clk, reset;
4
5       //RF/FWD Stage
6       input [0:31]            instr_even, instr_odd;          //Instructions to read from
        decoder
7       output logic [0:127]    ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd;
        //Set all possible register values regardless of format
8
9       //WB Stage
10      input [0:6]             rt_addr_even, rt_addr_odd;      //Destination registers to
        write to
11      input [0:127]           rt_even, rt_odd;                //Values to write to
        destination registers
12      input                   reg_write_even, reg_write_odd;  //1 if instr will write to
        rt, else 0
13
14      logic [0:127] registers [0:127];                        //RegFile
15      logic [7:0]             i;                              //8-bit counter for reset
        loop
16
17      always_comb begin
18          //All source register addresses are in the same location in all instr, so read
            no matter what
19          rc_even = registers[instr_even[25:31]];
20          ra_even = registers[instr_even[18:24]];
21          rb_even = registers[instr_even[11:17]];
22
23          ra_odd = registers[instr_odd[18:24]];
24          rb_odd = registers[instr_odd[11:17]];
25          rt_st_odd = registers[instr_odd[25:31]];
26
27          //Forwarding in case of WAR hazard
28          if (reg_write_even == 1) begin
29              if (instr_even[25:31] == rt_addr_even)
30                  rc_even = rt_even;
31              if (instr_even[18:24] == rt_addr_even)
32                  ra_even = rt_even;
33              if (instr_even[11:17] == rt_addr_even)
34                  rb_even = rt_even;
35              if (instr_odd[25:31] == rt_addr_even)
36                  rt_st_odd = rt_even;
37              if (instr_odd[18:24] == rt_addr_even)
38                  ra_odd = rt_even;
39              if (instr_odd[11:17] == rt_addr_even)
40                  rb_odd = rt_even;
41          end
42          if (reg_write_odd == 1) begin
43              if (instr_even[25:31] == rt_addr_odd)
44                  rc_even = rt_odd;
45              if (instr_even[18:24] == rt_addr_odd)
46                  ra_even = rt_odd;
47              if (instr_even[11:17] == rt_addr_odd)
48                  rb_even = rt_odd;
49              if (instr_odd[25:31] == rt_addr_odd)
50                  rt_st_odd = rt_odd;
51              if (instr_odd[18:24] == rt_addr_odd)
52                  ra_odd = rt_odd;
53              if (instr_odd[11:17] == rt_addr_odd)
54                  rb_odd = rt_odd;
55          end
56      end
57
58      always_ff @(posedge clk) begin
59          if(reset == 1)begin
60              registers[127] = 0;
```

```
61              for (i=0; i<127; i=i+1) begin
62                  registers[i] <= 0;
63              end
64          end
65          else begin
66              if (reg_write_even == 1)
67                  registers[rt_addr_even] <= rt_even;
68              if (reg_write_odd == 1)
69                  registers[rt_addr_odd] <= rt_odd;
70          end
71      end
72  endmodule
```

```
1   module Forward(clk, reset, instr_even, instr_odd, ra_even, rb_even, rc_even, ra_odd,
    rb_odd, rt_st_odd, ra_even_fwd, rb_even_fwd, rc_even_fwd, ra_odd_fwd, rb_odd_fwd,
    rt_st_odd_fwd, fw_even_wb, fw_addr_even_wb, fw_write_even_wb, fw_odd_wb,
    fw_addr_odd_wb, fw_write_odd_wb);
2       input           clk, reset;
3       input [0:31]    instr_even, instr_odd;                  //Instr from decoder
4
5       //Nets from decode logic (Note: Will be placed in decode/hazard unit in final
        submission)
6       input [0:127]   ra_even, rb_even, rc_even, ra_odd, rb_odd, rt_st_odd;   //Register
        values from RegTable
7
8       //Signals for forwarding logic
9       input [6:0][0:127]  fw_even_wb, fw_odd_wb;              //Pipe shift registers of
        values ready to be forwarded
10      input [6:0][0:6]    fw_addr_even_wb, fw_addr_odd_wb;    //Destinations of values to
        be forwarded
11      input [6:0]         fw_write_even_wb, fw_write_odd_wb;  //Will forwarded values be
        writted to reg?
12      output logic [0:127]    ra_even_fwd, rb_even_fwd, rc_even_fwd, ra_odd_fwd,
        rb_odd_fwd, rt_st_odd_fwd;   //Updated input values
13
14      always_comb begin
15          //Checking inputs for even instr. Compared against FWE0 to FWE6, breaking when
            match found
16          //ra
17          if (instr_even[18:24] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
18              ra_even_fwd = fw_even_wb[0];
19          else if (instr_even[18:24] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
20              ra_even_fwd = fw_odd_wb[0];
21          else if (instr_even[18:24] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
22              ra_even_fwd = fw_even_wb[1];
23          else if (instr_even[18:24] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
24              ra_even_fwd = fw_odd_wb[1];
25          else if (instr_even[18:24] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
26              ra_even_fwd = fw_even_wb[2];
27          else if (instr_even[18:24] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
28              ra_even_fwd = fw_odd_wb[2];
29          else if (instr_even[18:24] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
30              ra_even_fwd = fw_even_wb[3];
31          else if (instr_even[18:24] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
32              ra_even_fwd = fw_odd_wb[3];
33          else if (instr_even[18:24] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
34              ra_even_fwd = fw_even_wb[4];
35          else if (instr_even[18:24] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
36              ra_even_fwd = fw_odd_wb[4];
37          else if (instr_even[18:24] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
38              ra_even_fwd = fw_even_wb[5];
39          else if (instr_even[18:24] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
40              ra_even_fwd = fw_odd_wb[5];
41          else if (instr_even[18:24] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
42              ra_even_fwd = fw_even_wb[6];
43          else if (instr_even[18:24] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
44              ra_even_fwd = fw_odd_wb[6];
45          else
46              ra_even_fwd = ra_even;
47
48          //rb
49          if (instr_even[11:17] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
50              rb_even_fwd = fw_even_wb[0];
51          else if (instr_even[11:17] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
52              rb_even_fwd = fw_odd_wb[0];
53          else if (instr_even[11:17] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
54              rb_even_fwd = fw_even_wb[1];
55          else if (instr_even[11:17] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
56              rb_even_fwd = fw_odd_wb[1];
57          else if (instr_even[11:17] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
58              rb_even_fwd = fw_even_wb[2];
59          else if (instr_even[11:17] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
```

```
60              rb_even_fwd = fw_odd_wb[2];
61          else if (instr_even[11:17] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
62              rb_even_fwd = fw_even_wb[3];
63          else if (instr_even[11:17] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
64              rb_even_fwd = fw_odd_wb[3];
65          else if (instr_even[11:17] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
66              rb_even_fwd = fw_even_wb[4];
67          else if (instr_even[11:17] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
68              rb_even_fwd = fw_odd_wb[4];
69          else if (instr_even[11:17] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
70              rb_even_fwd = fw_even_wb[5];
71          else if (instr_even[11:17] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
72              rb_even_fwd = fw_odd_wb[5];
73          else if (instr_even[11:17] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
74              rb_even_fwd = fw_even_wb[6];
75          else if (instr_even[11:17] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
76              rb_even_fwd = fw_odd_wb[6];
77          else
78              rb_even_fwd = rb_even;
79
80          //rc
81          if (instr_even[25:31] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
82              rc_even_fwd = fw_even_wb[0];
83          else if (instr_even[25:31] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
84              rc_even_fwd = fw_odd_wb[0];
85          else if (instr_even[25:31] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
86              rc_even_fwd = fw_even_wb[1];
87          else if (instr_even[25:31] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
88              rc_even_fwd = fw_odd_wb[1];
89          else if (instr_even[25:31] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
90              rc_even_fwd = fw_even_wb[2];
91          else if (instr_even[25:31] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
92              rc_even_fwd = fw_odd_wb[2];
93          else if (instr_even[25:31] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
94              rc_even_fwd = fw_even_wb[3];
95          else if (instr_even[25:31] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
96              rc_even_fwd = fw_odd_wb[3];
97          else if (instr_even[25:31] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
98              rc_even_fwd = fw_even_wb[4];
99          else if (instr_even[25:31] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
100             rc_even_fwd = fw_odd_wb[4];
101         else if (instr_even[25:31] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
102             rc_even_fwd = fw_even_wb[5];
103         else if (instr_even[25:31] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
104             rc_even_fwd = fw_odd_wb[5];
105         else if (instr_even[25:31] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
106             rc_even_fwd = fw_even_wb[6];
107         else if (instr_even[25:31] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
108             rc_even_fwd = fw_odd_wb[6];
109         else
110             rc_even_fwd = rc_even;
111
112         //Checking inputs for odd instr. Compared against FWO0 to FWO6, breaking when
            match found
113         //ra
114         if (instr_odd[18:24] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
115             ra_odd_fwd = fw_even_wb[0];
116         else if (instr_odd[18:24] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
117             ra_odd_fwd = fw_odd_wb[0];
118         else if (instr_odd[18:24] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
119             ra_odd_fwd = fw_even_wb[1];
120         else if (instr_odd[18:24] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
121             ra_odd_fwd = fw_odd_wb[1];
122         else if (instr_odd[18:24] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
123             ra_odd_fwd = fw_even_wb[2];
124         else if (instr_odd[18:24] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
125             ra_odd_fwd = fw_odd_wb[2];
126         else if (instr_odd[18:24] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
127             ra_odd_fwd = fw_even_wb[3];
```

```
128             else if (instr_odd[18:24] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
129                 ra_odd_fwd = fw_odd_wb[3];
130             else if (instr_odd[18:24] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
131                 ra_odd_fwd = fw_even_wb[4];
132             else if (instr_odd[18:24] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
133                 ra_odd_fwd = fw_odd_wb[4];
134             else if (instr_odd[18:24] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
135                 ra_odd_fwd = fw_even_wb[5];
136             else if (instr_odd[18:24] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
137                 ra_odd_fwd = fw_odd_wb[5];
138             else if (instr_odd[18:24] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
139                 ra_odd_fwd = fw_even_wb[6];
140             else if (instr_odd[18:24] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
141                 ra_odd_fwd = fw_odd_wb[6];
142             else
143                 ra_odd_fwd = ra_odd;
144 
145             //rb
146             if (instr_odd[11:17] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
147                 rb_odd_fwd = fw_even_wb[0];
148             else if (instr_odd[11:17] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
149                 rb_odd_fwd = fw_odd_wb[0];
150             else if (instr_odd[11:17] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
151                 rb_odd_fwd = fw_even_wb[1];
152             else if (instr_odd[11:17] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
153                 rb_odd_fwd = fw_odd_wb[1];
154             else if (instr_odd[11:17] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
155                 rb_odd_fwd = fw_even_wb[2];
156             else if (instr_odd[11:17] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
157                 rb_odd_fwd = fw_odd_wb[2];
158             else if (instr_odd[11:17] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
159                 rb_odd_fwd = fw_even_wb[3];
160             else if (instr_odd[11:17] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
161                 rb_odd_fwd = fw_odd_wb[3];
162             else if (instr_odd[11:17] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
163                 rb_odd_fwd = fw_even_wb[4];
164             else if (instr_odd[11:17] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
165                 rb_odd_fwd = fw_odd_wb[4];
166             else if (instr_odd[11:17] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
167                 rb_odd_fwd = fw_even_wb[5];
168             else if (instr_odd[11:17] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
169                 rb_odd_fwd = fw_odd_wb[5];
170             else if (instr_odd[11:17] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
171                 rb_odd_fwd = fw_even_wb[6];
172             else if (instr_odd[11:17] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
173                 rb_odd_fwd = fw_odd_wb[6];
174             else
175                 rb_odd_fwd = rb_odd;
176 
177             //rt_st
178             if (instr_odd[25:31] == fw_addr_even_wb[0] && fw_write_even_wb[0] == 1)
179                 rt_st_odd_fwd = fw_even_wb[0];
180             else if (instr_odd[25:31] == fw_addr_odd_wb[0] && fw_write_odd_wb[0] == 1)
181                 rt_st_odd_fwd = fw_odd_wb[0];
182             else if (instr_odd[25:31] == fw_addr_even_wb[1] && fw_write_even_wb[1] == 1)
183                 rt_st_odd_fwd = fw_even_wb[1];
184             else if (instr_odd[25:31] == fw_addr_odd_wb[1] && fw_write_odd_wb[1] == 1)
185                 rt_st_odd_fwd = fw_odd_wb[1];
186             else if (instr_odd[25:31] == fw_addr_even_wb[2] && fw_write_even_wb[2] == 1)
187                 rt_st_odd_fwd = fw_even_wb[2];
188             else if (instr_odd[25:31] == fw_addr_odd_wb[2] && fw_write_odd_wb[2] == 1)
189                 rt_st_odd_fwd = fw_odd_wb[2];
190             else if (instr_odd[25:31] == fw_addr_even_wb[3] && fw_write_even_wb[3] == 1)
191                 rt_st_odd_fwd = fw_even_wb[3];
192             else if (instr_odd[25:31] == fw_addr_odd_wb[3] && fw_write_odd_wb[3] == 1)
193                 rt_st_odd_fwd = fw_odd_wb[3];
194             else if (instr_odd[25:31] == fw_addr_even_wb[4] && fw_write_even_wb[4] == 1)
195                 rt_st_odd_fwd = fw_even_wb[4];
196             else if (instr_odd[25:31] == fw_addr_odd_wb[4] && fw_write_odd_wb[4] == 1)
```

```
197             rt_st_odd_fwd = fw_odd_wb[4];
198         else if (instr_odd[25:31] == fw_addr_even_wb[5] && fw_write_even_wb[5] == 1)
199             rt_st_odd_fwd = fw_even_wb[5];
200         else if (instr_odd[25:31] == fw_addr_odd_wb[5] && fw_write_odd_wb[5] == 1)
201             rt_st_odd_fwd = fw_odd_wb[5];
202         else if (instr_odd[25:31] == fw_addr_even_wb[6] && fw_write_even_wb[6] == 1)
203             rt_st_odd_fwd = fw_even_wb[6];
204         else if (instr_odd[25:31] == fw_addr_odd_wb[6] && fw_write_odd_wb[6] == 1)
205             rt_st_odd_fwd = fw_odd_wb[6];
206         else
207             rt_st_odd_fwd = rt_st_odd;
208     end
209
210 endmodule
```

```
1   module EvenPipe(clk, reset, op, format, unit, rt_addr, ra, rb, rc, imm, reg_write,
    rt_wb, rt_addr_wb, reg_write_wb, branch_taken, fw_wb, fw_addr_wb, fw_write_wb,
2       rt_addr_delay_fp1, reg_write_delay_fp1, int_delay_fp1, rt_addr_delay_fx2,
        reg_write_delay_fx2, rt_addr_delay_b1, reg_write_delay_b1, rt_addr_delay_fx1,
        reg_write_delay_fx1);
3       input           clk, reset;
4
5       //RF/FWD Stage
6       input [0:10]    op;             //Decoded opcode, truncated based on format
7       input [2:0]     format;         //Format of instr, used with op and imm
8       input [1:0]     unit;           //Execution unit of instr (0: FP, 1: FX2, 2: Byte,
        3: FX1)
9       input [0:6]     rt_addr;        //Destination register address
10      input [0:127]   ra, rb, rc;     //Values of source registers
11      input [0:17]    imm;            //Immediate value, truncated based on format
12      input           reg_write;      //Will current instr write to RegTable
13      input           branch_taken;   //Was branch taken?
14
15      //WB Stage
16      output logic [0:127]    rt_wb;          //Output value of Stage 7
17      output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
18      output logic            reg_write_wb;   //Will rt_wb write to RegTable
19
20      //Internal Signals
21      output logic [6:0][0:127]   fw_wb;          //Staging register for forwarded values
22      output logic [6:0][0:6]     fw_addr_wb;     //Destination register for rt_wb
23      output logic [6:0]          fw_write_wb;    //Will rt_wb write to RegTable
24
25      logic [0:10]        fp1_op;         //Multiplexed opcode
26      logic [2:0]         fp1_format;     //Multiplexed format
27      logic               fp1_reg_write;  //Multiplexed reg_write
28      logic [0:127]       fp1_out;        //Output value of fp1 Stage 6
29      logic [0:6]         fp1_addr_out;   //Destination register for rt_wb
30      logic               fp1_write_out;  //Will rt_wb write to RegTable
31      logic [0:127]       fp1_int;        //Output value of fp1 Stage 7
32      logic [0:6]         fp1_addr_int;   //Destination register for rt_wb
33      logic               fp1_write_int;  //Will rt_wb write to RegTable
34
35      logic [0:10]        fx2_op;         //Multiplexed opcode
36      logic [2:0]         fx2_format;     //Multiplexed format
37      logic               fx2_reg_write;  //Multiplexed reg_write
38      logic [0:127]       fx2_out;        //Output value of fx2 Stage 4
39      logic [0:6]         fx2_addr_out;   //Destination register for rt_wb
40      logic               fx2_write_out;  //Will rt_wb write to RegTable
41
42      logic [0:10]        b1_op;          //Multiplexed opcode
43      logic [2:0]         b1_format;      //Multiplexed format
44      logic               b1_reg_write;   //Multiplexed reg_write
45      logic [0:127]       b1_out;         //Output value of b1 Stage 4
46      logic [0:6]         b1_addr_out;    //Destination register for rt_wb
47      logic               b1_write_out;   //Will rt_wb write to RegTable
48
49      logic [0:10]        fx1_op;         //Multiplexed opcode
50      logic [2:0]         fx1_format;     //Multiplexed format
51      logic               fx1_reg_write;  //Multiplexed reg_write
52      logic [0:127]       fx1_out;        //Output value of fx1 Stage 2
53      logic [0:6]         fx1_addr_out;   //Destination register for rt_wb
54      logic               fx1_write_out;  //Will rt_wb write to RegTable
55
56      //Internal Signals for Handling RAW Hazards
57      output logic [6:0][0:6] rt_addr_delay_fp1;      //Destination register for rt_wb
58      output logic [6:0]      reg_write_delay_fp1;    //Will rt_wb write to RegTable
59      output logic [6:0]      int_delay_fp1;          //Will fp1 write an int result
60
61      output logic [3:0][0:6] rt_addr_delay_fx2;      //Destination register for rt_wb
62      output logic [3:0]      reg_write_delay_fx2;    //Will rt_wb write to RegTable
63
64      output logic [3:0][0:6] rt_addr_delay_b1;       //Destination register for rt_wb
65      output logic [3:0]      reg_write_delay_b1;     //Will rt_wb write to RegTable
```

```
66
67      output logic [1:0][0:6] rt_addr_delay_fx1;      //Destination register for rt_wb
68      output logic [1:0]      reg_write_delay_fx1;    //Will rt_wb write to RegTable
69
70
71      SinglePrecision fp1(.clk(clk), .reset(reset), .op(fp1_op), .format(fp1_format),
        .rt_addr(rt_addr), .ra(ra), .rb(rb), .rc(rc), .imm(imm), .reg_write(fp1_reg_write),
72          .rt_wb(fp1_out), .rt_addr_wb(fp1_addr_out), .reg_write_wb(fp1_write_out),
            .rt_int(fp1_int), .rt_addr_int(fp1_addr_int), .reg_write_int(fp1_write_int),
73          .branch_taken(branch_taken), .rt_addr_delay(rt_addr_delay_fp1),
            .reg_write_delay(reg_write_delay_fp1), .int_delay(int_delay_fp1));
74
75      SimpleFixed2 fx2(.clk(clk), .reset(reset), .op(fx2_op), .format(fx2_format),
        .rt_addr(rt_addr), .ra(ra), .rb(rb), .imm(imm), .reg_write(fx2_reg_write),
        .rt_wb(fx2_out),
76          .rt_addr_wb(fx2_addr_out), .reg_write_wb(fx2_write_out),
            .branch_taken(branch_taken),
77          .rt_addr_delay(rt_addr_delay_fx2), .reg_write_delay(reg_write_delay_fx2));
78
79      Byte b1(.clk(clk), .reset(reset), .op(b1_op), .format(b1_format),
        .rt_addr(rt_addr), .ra(ra), .rb(rb), .imm(imm), .reg_write(b1_reg_write),
        .rt_wb(b1_out),
80          .rt_addr_wb(b1_addr_out), .reg_write_wb(b1_write_out),
            .branch_taken(branch_taken),
81          .rt_addr_delay(rt_addr_delay_b1), .reg_write_delay(reg_write_delay_b1));
82
83      SimpleFixed1 fx1(.clk(clk), .reset(reset), .op(fx1_op), .format(fx1_format),
        .rt_addr(rt_addr), .ra(ra), .rb(rb), .rt_st(rc), .imm(imm),
        .reg_write(fx1_reg_write), .rt_wb(fx1_out),
84          .rt_addr_wb(fx1_addr_out), .reg_write_wb(fx1_write_out),
            .branch_taken(branch_taken),
85          .rt_addr_delay(rt_addr_delay_fx1), .reg_write_delay(reg_write_delay_fx1));
86
87
88      always_comb begin
89          fp1_op = 0;
90          fp1_format = 0;
91          fp1_reg_write = 0;
92
93          fx2_op = 0;
94          fx2_format = 0;
95          fx2_reg_write = 0;
96
97          b1_op = 0;
98          b1_format = 0;
99          b1_reg_write = 0;
100
101         fx1_op = 0;
102         fx1_format = 0;
103         fx1_reg_write = 0;
104
105
106         case (unit)                                 //Mux to determine which unit will
            take the instr
107             2'b00 : begin                           //Instr going to fp1
108                 fp1_op = op;
109                 fp1_format = format;
110                 fp1_reg_write = reg_write;
111             end
112             2'b01 : begin                           //Instr going to fx2
113                 fx2_op = op;
114                 fx2_format = format;
115                 fx2_reg_write = reg_write;
116             end
117             2'b10 : begin                           //Instr going to b1
118                 b1_op = op;
119                 b1_format = format;
120                 b1_reg_write = reg_write;
121             end
```

```
            2'b11 : begin                               //Instr going to fx1
                fx1_op = op;
                fx1_format = format;
                fx1_reg_write = reg_write;
            end
        endcase
    end

    always_ff @(posedge clk) begin
        fw_wb[0] <= 0;                                  //fw0 and fw1 don't exist, just use 0
        fw_addr_wb[0] <= 0;
        fw_write_wb[0] <= 0;

        if (reset == 1) begin
            rt_wb <= 0;
            rt_addr_wb <= 0;
            reg_write_wb <= 0;
            for (int i=6; i>0; i=i-1) begin
                fw_wb [i] <= 0;
                fw_addr_wb [i] <= 0;
                fw_write_wb [i] <= 0;
            end
        end
        else begin
            rt_wb <= fw_wb[6];
            rt_addr_wb <= fw_addr_wb[6];
            reg_write_wb <= fw_write_wb[6];

            if (fp1_write_int == 1) begin               //Replace fw6 with fp1 if possible
                fw_wb[6] <= fp1_int;
                fw_addr_wb[6] <= fp1_addr_int;
                fw_write_wb[6] <= fp1_write_int;
            end
            else begin
                fw_wb[6] <= fw_wb[5];
                fw_addr_wb[6] <= fw_addr_wb[5];
                fw_write_wb[6] <= fw_write_wb[5];
            end

            if (fp1_write_out == 1) begin               //Replace fw6 with fp1 if possible
                fw_wb[5] <= fp1_out;
                fw_addr_wb[5] <= fp1_addr_out;
                fw_write_wb[5] <= fp1_write_out;
            end
            else begin
                fw_wb[5] <= fw_wb[4];
                fw_addr_wb[5] <= fw_addr_wb[4];
                fw_write_wb[5] <= fw_write_wb[4];
            end

            fw_wb[4] <= fw_wb[3];
            fw_addr_wb[4] <= fw_addr_wb[3];
            fw_write_wb[4] <= fw_write_wb[3];

            if (fx2_write_out == 1) begin               //Replace fw4 with fx2 if possible
                fw_wb[3] <= fx2_out;
                fw_addr_wb[3] <= fx2_addr_out;
                fw_write_wb[3] <= fx2_write_out;
            end
            else if (b1_write_out == 1) begin           //Replace fw4 with b1 if possible
                fw_wb[3] <= b1_out;
                fw_addr_wb[3] <= b1_addr_out;
                fw_write_wb[3] <= b1_write_out;
            end
            else begin
                fw_wb[3] <= fw_wb[2];
                fw_addr_wb[3] <= fw_addr_wb[2];
                fw_write_wb[3] <= fw_write_wb[2];
            end
```

```
191
192                 fw_wb[2] <= fw_wb[1];
193                 fw_addr_wb[2] <= fw_addr_wb[1];
194                 fw_write_wb[2] <= fw_write_wb[1];
195
196                 if (fx1_write_out == 1) begin           //Replace fw2 with fx1 if possible
197                     fw_wb[1] <= fx1_out;
198                     fw_addr_wb[1] <= fx1_addr_out;
199                     fw_write_wb[1] <= fx1_write_out;
200                 end
201                 else begin
202                     fw_wb[1] <= fw_wb[0];
203                     fw_addr_wb[1] <= fw_addr_wb[0];
204                     fw_write_wb[1] <= fw_write_wb[0];
205                 end
206             end
207         end
208
209     endmodule
```

```
1   module OddPipe(clk, reset, op, format, unit, rt_addr, ra, rb, rt_st, imm, reg_write,
    pc_in, rt_wb, rt_addr_wb, reg_write_wb, pc_wb, branch_taken,
2       fw_wb, fw_addr_wb, fw_write_wb, first, branch_kill, rt_addr_delay_ls1,
        reg_write_delay_ls1, rt_addr_delay_p1, reg_write_delay_p1);
3       input           clk, reset;
4
5       //RF/FWD Stage
6       input [0:10]    op;             //Decoded opcode, truncated based on format
7       input [2:0]     format;         //Format of instr, used with op and imm
8       input [1:0]     unit;           //Execution unit of instr (0: Perm, 1: LS, 2: Br,
        3: Undefined)
9       input [0:6]     rt_addr;        //Destination register address
10      input [0:127]   ra, rb, rt_st;  //Values of source registers
11      input [0:17]    imm;            //Immediate value, truncated based on format
12      input           reg_write;      //Will current instr write to RegTable
13      input [7:0]     pc_in;          //Program counter from IF stage
14      input           first;          //1 if first instr in pair; used for determining
        order of branch
15
16      //WB Stage
17      output logic [0:127]    rt_wb;          //Output value of Stage 7
18      output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
19      output logic            reg_write_wb;   //Will rt_wb write to RegTable
20      output logic [7:0]      pc_wb;          //New program counter for branch
21      output logic            branch_taken;   //Was branch taken?
22      output logic            branch_kill;    //If branch is taken and branch instr is
        first in pair, kill twin instr
23
24      //Internal Signals
25      output logic [6:0][0:127]   fw_wb;          //Staging register for forwarded values
26      output logic [6:0][0:6]     fw_addr_wb;     //Destination register for rt_wb
27      output logic [6:0]          fw_write_wb;    //Will rt_wb write to RegTable
28
29      logic [0:10]        p1_op;          //Multiplexed opcode
30      logic [2:0]         p1_format;      //Multiplexed format
31      logic               p1_reg_write;   //Multiplexed reg_write
32      logic [0:127]       p1_out;         //Output value of p1 Stage 4
33      logic [0:6]         p1_addr_out;    //Destination register for rt_wb
34      logic               p1_write_out;   //Will rt_wb write to RegTable
35
36      logic [0:10]        ls1_op;         //Multiplexed opcode
37      logic [2:0]         ls1_format;     //Multiplexed format
38      logic               ls1_reg_write;  //Multiplexed reg_write
39      logic [0:127]       ls1_out;        //Output value of ls Stage 6
40      logic [0:6]         ls1_addr_out;   //Destination register for rt_wb
41      logic               ls1_write_out;  //Will rt_wb write to RegTable
42
43      logic [0:10]        br1_op;         //Multiplexed opcode
44      logic [2:0]         br1_format;     //Multiplexed format
45      logic               br1_reg_write;  //Multiplexed reg_write
46      logic [0:127]       br1_out;        //Output value of br1 Stage 1
47      logic [0:6]         br1_addr_out;   //Destination register for rt_wb
48      logic               br1_write_out;  //Will rt_wb write to RegTable
49
50      //Internal Signals for Handling RAW Hazards
51      output logic [5:0][0:6] rt_addr_delay_ls1;      //Destination register for rt_wb
52      output logic [5:0]          reg_write_delay_ls1;    //Will rt_wb write to RegTable
53
54      output logic [3:0][0:6] rt_addr_delay_p1;       //Destination register for rt_wb
55      output logic [3:0]          reg_write_delay_p1;     //Will rt_wb write to RegTable
56
57      Permute p1(.clk(clk), .reset(reset), .op(p1_op), .format(p1_format),
        .rt_addr(rt_addr), .ra(ra), .rb(rb), .imm(imm), .reg_write(p1_reg_write),
        .rt_wb(p1_out),
58          .rt_addr_wb(p1_addr_out), .reg_write_wb(p1_write_out),
            .branch_taken(branch_taken), .rt_addr_delay(rt_addr_delay_p1),
            .reg_write_delay(reg_write_delay_p1));
59
60      LocalStore ls1(.clk(clk), .reset(reset), .op(ls1_op), .format(ls1_format),
```

```
    .rt_addr(rt_addr), .ra(ra), .rb(rb), .rt_st(rt_st), .imm(imm),
    .reg_write(ls1_reg_write), .rt_wb(ls1_out),
61      .rt_addr_wb(ls1_addr_out), .reg_write_wb(ls1_write_out),
        .branch_taken(branch_taken), .rt_addr_delay(rt_addr_delay_ls1),
        .reg_write_delay(reg_write_delay_ls1));
62
63  Branch br1(.clk(clk), .reset(reset), .op(br1_op), .format(br1_format),
    .rt_addr(rt_addr), .ra(ra), .rb(rb), .rt_st(rt_st), .imm(imm),
    .reg_write(br1_reg_write), .pc_in(pc_in), .rt_wb(br1_out),
64      .rt_addr_wb(br1_addr_out), .reg_write_wb(br1_write_out), .pc_wb(pc_wb),
        .branch_taken(branch_taken), .first(first), .branch_kill(branch_kill));
65
66  always_comb begin
67      p1_op = 0;
68      p1_format = 0;
69      p1_reg_write = 0;
70
71      ls1_op = 0;
72      ls1_format = 0;
73      ls1_reg_write = 0;
74
75      br1_op = 0;
76      br1_format = 0;
77      br1_reg_write = 0;
78
79      case (unit)                                 //Mux to determine which unit will
        take the instr
80          2'b00 : begin                           //Instr going to p1
81              p1_op = op;
82              p1_format = format;
83              p1_reg_write = reg_write;
84          end
85          2'b01 : begin                           //Instr going to ls1
86              ls1_op = op;
87              ls1_format = format;
88              ls1_reg_write = reg_write;
89          end
90          2'b10 : begin                           //Instr going to br1
91              br1_op = op;
92              br1_format = format;
93              br1_reg_write = reg_write;
94          end
95          default begin                           //Instr going to p1
96              p1_op = op;
97              p1_format = format;
98              p1_reg_write = reg_write;
99          end
100     endcase
101 end
102
103 always_ff @(posedge clk) begin
104
105     if (reset == 1) begin
106         rt_wb = 0;
107         rt_addr_wb = 0;
108         reg_write_wb = 0;
109         for (int i=6; i>0; i=i-1) begin
110             fw_wb [i] <= 0;
111             fw_addr_wb [i] <= 0;
112             fw_write_wb [i] <= 0;
113         end
114     end
115     else begin
116         rt_wb <= fw_wb[6];
117         rt_addr_wb <= fw_addr_wb[6];
118         reg_write_wb <= fw_write_wb[6];
119
120         fw_wb[6] <= fw_wb[5];
121         fw_addr_wb[6] <= fw_addr_wb[5];
```

```
122                 fw_write_wb[6] <= fw_write_wb[5];
123
124                 if (ls1_write_out == 1) begin           //Replace fw6 with ls1 if possible
125                     fw_wb[5] <= ls1_out;
126                     fw_addr_wb[5] <= ls1_addr_out;
127                     fw_write_wb[5] <= ls1_write_out;
128                 end
129                 else begin
130                     fw_wb[5] <= fw_wb[4];
131                     fw_addr_wb[5] <= fw_addr_wb[4];
132                     fw_write_wb[5] <= fw_write_wb[4];
133                 end
134
135                 fw_wb[4] <= fw_wb[3];
136                 fw_addr_wb[4] <= fw_addr_wb[3];
137                 fw_write_wb[4] <= fw_write_wb[3];
138
139                 if (p1_write_out == 1) begin            //Replace fw4 with p1 if possible
140                     fw_wb[3] <= p1_out;
141                     fw_addr_wb[3] <= p1_addr_out;
142                     fw_write_wb[3] <= p1_write_out;
143                 end
144                 else begin
145                     fw_wb[3] <= fw_wb[2];
146                     fw_addr_wb[3] <= fw_addr_wb[2];
147                     fw_write_wb[3] <= fw_write_wb[2];
148                 end
149
150                 fw_wb[2] <= fw_wb[1];
151                 fw_addr_wb[2] <= fw_addr_wb[1];
152                 fw_write_wb[2] <= fw_write_wb[1];
153
154                 fw_wb[1] <= fw_wb[0];
155                 fw_addr_wb[1] <= fw_addr_wb[0];
156                 fw_write_wb[1] <= fw_write_wb[0];
157
158                 if (br1_write_out == 1) begin           //Replace fw1 with p1 if possible
                    (??????)
159                     fw_wb[0] <= br1_out;
160                     fw_addr_wb[0] <= br1_addr_out;
161                     fw_write_wb[0] <= br1_write_out;
162                 end
163                 else begin
164                     fw_wb[0] <= 0;
165                     fw_addr_wb[0] <= 0;
166                     fw_write_wb[0] <= 0;
167                 end
168             end
169         end
170     endmodule
```

```
module SinglePrecision(clk, reset, op, format, rt_addr, ra, rb, rc, imm, reg_write,
rt_wb, rt_addr_wb, reg_write_wb, rt_int, rt_addr_int, reg_write_int, branch_taken,
    rt_addr_delay, reg_write_delay, int_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb, rc;     //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 6
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    output logic [0:127]    rt_int;         //Output value of Stage 7
    output logic [0:6]      rt_addr_int;    //Destination register for rt_wb
    output logic            reg_write_int;  //Will rt_wb write to RegTable

    //Internal Signals
    logic [6:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [6:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [6:0]      reg_write_delay;    //Will rt_wb write to RegTable
    output logic [6:0]      int_delay;          //1 if int op, 0 if else

    always_comb begin
        if (int_delay[5] == 1) begin            //FP7 writeback (only for int ops)
            rt_int = rt_delay[5];
            rt_addr_int = rt_addr_delay[5];
            reg_write_int = reg_write_delay[5];
        end
        else begin
            rt_int = 0;
            rt_addr_int = 0;
            reg_write_int = 0;
        end

        if (int_delay[4] == 0) begin            //FP6 writeback
            rt_wb = rt_delay[4];
            rt_addr_wb = rt_addr_delay[4];
            reg_write_wb = reg_write_delay[4];
        end
        else begin
            rt_wb = 0;
            rt_addr_wb = 0;
            reg_write_wb = 0;
        end
    end

    always_ff @(posedge clk) begin
        integer scale;
        shortreal tempfp;
        logic [0:15] temp16;

        if (reset == 1) begin
            rt_delay[6] <= 0;
            rt_addr_delay[6] <= 0;
            reg_write_delay[6] <= 0;
            int_delay[6] <= 0;
            for (int i=0; i<6; i=i+1) begin
                rt_delay[i] <= 0;
                rt_addr_delay[i] <= 0;
                reg_write_delay[i] <= 0;
                int_delay[i] <= 0;
            end
```

```
69          end
70          else begin
71              rt_delay[6] <= rt_delay[5];
72              rt_addr_delay[6] <= rt_addr_delay[5];
73              reg_write_delay[6] <= reg_write_delay[5];
74              int_delay[6] <= int_delay[5];
75              for (int i=0; i<5; i=i+1) begin
76                  rt_delay[i+1] <= rt_delay[i];
77                  rt_addr_delay[i+1] <= rt_addr_delay[i];
78                  reg_write_delay[i+1] <= reg_write_delay[i];
79                  int_delay[i+1] <= int_delay[i];
80              end
81
82              if (format == 0 && op[0:9] == 0000000000) begin     //nop : No Operation
                (Execute)
83                  rt_delay[0] <= 0;
84                  rt_addr_delay[0] <= 0;
85                  reg_write_delay[0] <= 0;
86                  int_delay[0] <= 0;
87              end
88              else begin
89                  rt_addr_delay[0] <= rt_addr;
90                  reg_write_delay[0] <= reg_write;
91                  if (branch_taken) begin
92                      int_delay[0] <= 0;
93                      rt_delay[0] <= 0;
94                      rt_addr_delay[0] <= 0;
95                      reg_write_delay[0] <= 0;
96                  end
97                  else if (format == 0) begin
98                      case (op)
99                          11'b01111000100 : begin                 //mpy : Multiply
100                             int_delay[0] <= 1;
101                             for (int i=0; i<4; i=i+1)
102                                 rt_delay[0][(i*32) +: 32] <= $signed(ra[(i*32)+16 +:
                                    16]) * $signed(rb[(i*32)+16 +: 16]);
103                         end
104                         11'b01111001100 : begin                 //mpyu : Multiply
                            Unsigned
105                             int_delay[0] <= 1;
106                             for (int i=0; i<4; i=i+1)
107                                 rt_delay[0][(i*32) +: 32] <= $unsigned(ra[(i*32)+16 +:
                                    16]) * $unsigned(rb[(i*32)+16 +: 16]);
108                         end
109                         11'b01111000101 : begin                 //mpyh : Multiply High
110                             int_delay[0] <= 1;
111                             for (int i=0; i<4; i=i+1)
112                                 rt_delay[0][(i*32) +: 32] <= ($signed(ra[(i*32) +: 16])
                                    * $signed(rb[(i*32)+16 +: 16])) << 16;
113                         end
114                         11'b01011000100 : begin                 //fa : Floating Add
115                             int_delay[0] <= 0;
116                             for (int i=0; i<4; i=i+1) begin
117                                 if (($bitstoshortreal(ra[(i*32) +: 32]) +
                                    $bitstoshortreal(rb[(i*32) +: 32])) >=
                                    $bitstoshortreal(32'h7F7FFFFF))
118                                     rt_delay[0][(i*32) +: 32] <= 32'h7F7FFFFF;
119                                 else if (($bitstoshortreal(ra[(i*32) +: 32]) +
                                    $bitstoshortreal(rb[(i*32) +: 32])) <=
                                    $bitstoshortreal(32'hFF7FFFFF))
120                                     rt_delay[0][(i*32) +: 32] <= 32'hFF7FFFFF;
121                                 else if (($bitstoshortreal(ra[(i*32) +: 32]) +
                                    $bitstoshortreal(rb[(i*32) +: 32])) <=
                                    $bitstoshortreal(32'h00000001)
122                                     && ($bitstoshortreal(ra[(i*32) +: 32]) +
                                    $bitstoshortreal(rb[(i*32) +: 32])) > 0)
123                                     rt_delay[0][(i*32) +: 32] <= 32'h00000001;
124                                 else if (($bitstoshortreal(ra[(i*32) +: 32]) +
                                    $bitstoshortreal(rb[(i*32) +: 32])) >=
```

```
                        $bitstoshortreal(32'h80000001)
125                         && ($bitstoshortreal(ra[(i*32) +: 32]) +
                            $bitstoshortreal(rb[(i*32) +: 32])) < 0)
126                         rt_delay[0][(i*32) +: 32] <= 32'h80000001;
127                     else
128                         rt_delay[0][(i*32) +: 32] <=
                            $shortrealtobits($bitstoshortreal(ra[(i*32) +: 32]) +
                            $bitstoshortreal(rb[(i*32) +: 32]));
129                 end
130             end
131             11'b01011000101 : begin                 //fs : Floating Subtract
132                 int_delay[0] <= 0;
133                 for (int i=0; i<4; i=i+1) begin
134                     if (($bitstoshortreal(ra[(i*32) +: 32]) -
                        $bitstoshortreal(rb[(i*32) +: 32])) >=
                        $bitstoshortreal(32'h7F7FFFFF))
135                         rt_delay[0][(i*32) +: 32] <= 32'h7F7FFFFF;
136                     else if (($bitstoshortreal(ra[(i*32) +: 32]) -
                        $bitstoshortreal(rb[(i*32) +: 32])) <=
                        $bitstoshortreal(32'hFF7FFFFF))
137                         rt_delay[0][(i*32) +: 32] <= 32'hFF7FFFFF;
138                     else if (($bitstoshortreal(ra[(i*32) +: 32]) -
                        $bitstoshortreal(rb[(i*32) +: 32])) <=
                        $bitstoshortreal(32'h00000001)
139                         && ($bitstoshortreal(ra[(i*32) +: 32]) -
                            $bitstoshortreal(rb[(i*32) +: 32])) > 0)
140                         rt_delay[0][(i*32) +: 32] <= 32'h00000001;
141                     else if (($bitstoshortreal(ra[(i*32) +: 32]) -
                        $bitstoshortreal(rb[(i*32) +: 32])) >=
                        $bitstoshortreal(32'h80000001)
142                         && ($bitstoshortreal(ra[(i*32) +: 32]) -
                            $bitstoshortreal(rb[(i*32) +: 32])) < 0)
143                         rt_delay[0][(i*32) +: 32] <= 32'h80000001;
144                     else
145                         rt_delay[0][(i*32) +: 32] <=
                            $shortrealtobits($bitstoshortreal(ra[(i*32) +: 32]) -
                            $bitstoshortreal(rb[(i*32) +: 32]));
146                 end
147             end
148             11'b01011000110 : begin                 //fm : Floating Multiply
149                 int_delay[0] <= 0;
150                 for (int i=0; i<4; i=i+1) begin
151                     if (($bitstoshortreal(ra[(i*32) +: 32]) *
                        $bitstoshortreal(rb[(i*32) +: 32])) >=
                        $bitstoshortreal(32'h7F7FFFFF))
152                         rt_delay[0][(i*32) +: 32] <= 32'h7F7FFFFF;
153                     else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                        $bitstoshortreal(rb[(i*32) +: 32])) <=
                        $bitstoshortreal(32'hFF7FFFFF))
154                         rt_delay[0][(i*32) +: 32] <= 32'hFF7FFFFF;
155                     else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                        $bitstoshortreal(rb[(i*32) +: 32])) <=
                        $bitstoshortreal(32'h00000001)
156                         && ($bitstoshortreal(ra[(i*32) +: 32]) *
                            $bitstoshortreal(rb[(i*32) +: 32])) > 0)
157                         rt_delay[0][(i*32) +: 32] <= 32'h00000001;
158                     else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                        $bitstoshortreal(rb[(i*32) +: 32])) >=
                        $bitstoshortreal(32'h80000001)
159                         && ($bitstoshortreal(ra[(i*32) +: 32]) *
                            $bitstoshortreal(rb[(i*32) +: 32])) < 0)
160                         rt_delay[0][(i*32) +: 32] <= 32'h80000001;
161                     else
162                         rt_delay[0][(i*32) +: 32] <=
                            $shortrealtobits($bitstoshortreal(ra[(i*32) +: 32]) *
                            $bitstoshortreal(rb[(i*32) +: 32]));
163                 end
164             end
165             11'b01111000010 : begin                 //fceq : Floating
```

```
                        Compare Equal
166                         int_delay[0] <= 0;
167                         for (int i=0; i<4; i=i+1) begin
168                             if ($bitstoshortreal(ra[(i*32) +: 32]) ==
                                $bitstoshortreal(rb[(i*32) +: 32]))
169                                 rt_delay[0][(i*32) +: 32] <= 32'hFFFFFFFF;
170                             else
171                                 rt_delay[0][(i*32) +: 32] <= 0;
172                         end
173                     end
174                     11'b01011000010 : begin                 //fcgt : Floating
                        Compare Greater Than
175                         int_delay[0] <= 0;
176                         for (int i=0; i<4; i=i+1) begin
177                             if ($bitstoshortreal(ra[(i*32) +: 32]) >
                                $bitstoshortreal(rb[(i*32) +: 32]))
178                                 rt_delay[0][(i*32) +: 32] <= 32'hFFFFFFFF;
179                             else
180                                 rt_delay[0][(i*32) +: 32] <= 0;
181                         end
182                     end
183                     default begin
184                         int_delay[0] <= 0;
185                         rt_delay[0] <= 0;
186                         rt_addr_delay[0] <= 0;
187                         reg_write_delay[0] <= 0;
188                     end
189                 endcase
190             end
191             else if (format == 1) begin
192                 case (op[7:10])
193                     4'b1100 : begin                 //mpya : Multiply and Add
194                         int_delay[0] <= 1;
195                         for (int i=0; i<4; i=i+1)
196                             rt_delay[0][(i*32) +: 32] <= ($signed(ra[(i*32)+16 +:
                                16]) * $signed(rb[(i*32)+16 +: 16])) +
                                $signed(rc[(i*32) +: 32]);
197                     end
198                     4'b1110 : begin                 //fma : Floating Multiply and Add
199                         int_delay[0] <= 0;
200                         for (int i=0; i<4; i=i+1) begin
201                             if ((($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) +
                                $bitstoshortreal(rc[(i*32) +: 32])) >=
                                $bitstoshortreal(32'h7F7FFFFF))
202                                 rt_delay[0][(i*32) +: 32] <= 32'h7F7FFFFF;
203                             else if ((($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) +
                                $bitstoshortreal(rc[(i*32) +: 32])) <=
                                $bitstoshortreal(32'hFF7FFFFF))
204                                 rt_delay[0][(i*32) +: 32] <= 32'hFF7FFFFF;
205                             else if ((($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) +
                                $bitstoshortreal(rc[(i*32) +: 32])) <=
                                $bitstoshortreal(32'h00000001)
206                                 && (($bitstoshortreal(ra[(i*32) +: 32]) *
                                    $bitstoshortreal(rb[(i*32) +: 32])) +
                                    $bitstoshortreal(rc[(i*32) +: 32])) > 0)
207                                 rt_delay[0][(i*32) +: 32] <= 32'h00000001;
208                             else if ((($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) +
                                $bitstoshortreal(rc[(i*32) +: 32])) >=
                                $bitstoshortreal(32'h80000001)
209                                 && (($bitstoshortreal(ra[(i*32) +: 32]) *
                                    $bitstoshortreal(rb[(i*32) +: 32])) +
                                    $bitstoshortreal(rc[(i*32) +: 32])) < 0)
210                                 rt_delay[0][(i*32) +: 32] <= 32'h80000001;
211                             else
212                                 rt_delay[0][(i*32) +: 32] <=
```

```
                                $shortrealtobits(($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) +
                                $bitstoshortreal(rc[(i*32) +: 32]));
213                         end
214                     end
215                     4'b1111 : begin                 //fms : Floating Multiply and
                        Subtract
216                         int_delay[0] <= 0;
217                         for (int i=0; i<4; i=i+1) begin
218                             if (($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) -
                                $bitstoshortreal(rc[(i*32) +: 32])) >=
                                $bitstoshortreal(32'h7F7FFFFF))
219                                 rt_delay[0][(i*32) +: 32] <= 32'h7F7FFFFF;
220                             else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) -
                                $bitstoshortreal(rc[(i*32) +: 32])) <=
                                $bitstoshortreal(32'hFF7FFFFF))
221                                 rt_delay[0][(i*32) +: 32] <= 32'hFF7FFFFF;
222                             else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) -
                                $bitstoshortreal(rc[(i*32) +: 32])) <=
                                $bitstoshortreal(32'h00000001)
223                                 && (($bitstoshortreal(ra[(i*32) +: 32]) *
                                    $bitstoshortreal(rb[(i*32) +: 32])) -
                                    $bitstoshortreal(rc[(i*32) +: 32])) > 0)
224                                 rt_delay[0][(i*32) +: 32] <= 32'h00000001;
225                             else if (($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) -
                                $bitstoshortreal(rc[(i*32) +: 32])) >=
                                $bitstoshortreal(32'h80000001)
226                                 && (($bitstoshortreal(ra[(i*32) +: 32]) *
                                    $bitstoshortreal(rb[(i*32) +: 32])) -
                                    $bitstoshortreal(rc[(i*32) +: 32])) < 0)
227                                 rt_delay[0][(i*32) +: 32] <= 32'h80000001;
228                             else
229                                 rt_delay[0][(i*32) +: 32] <=
                                $shortrealtobits(($bitstoshortreal(ra[(i*32) +: 32]) *
                                $bitstoshortreal(rb[(i*32) +: 32])) -
                                $bitstoshortreal(rc[(i*32) +: 32]));
230                         end
231                     end
232                     default begin
233                         int_delay[0] <= 0;
234                         rt_delay[0] <= 0;
235                         rt_addr_delay[0] <= 0;
236                         reg_write_delay[0] <= 0;
237                     end
238                 endcase
239             end
240             else if (format == 3) begin
241                 case (op[1:10])
242                     10'b0111011000 : begin                  //cflts : Convert
                        Floating to Signed Integer
243                         int_delay[0] <= 0;
244                         for (int i=0; i<4; i=i+1) begin
245                             scale = 173 - $unsigned(imm[10:17]);
246                             if (scale > 127)
247                                 scale = 127;
248                             else if (scale < 0)
249                                 scale = 0;
250                             tempfp = $bitstoshortreal(ra[(i*32) +: 32]) * (2**scale);
251                             if (tempfp > (2**31 - 1))
252                                 rt_delay[0][(i*32) +: 32] <= 32'h7FFFFFFF;
253                             else if (tempfp < -(2**31))
254                                 rt_delay[0][(i*32) +: 32] <= 32'h80000000;
255                             else
256                                 rt_delay[0][(i*32) +: 32] <= int'(tempfp);
257                         end
```

```
                        end
                        10'b0111011001 : begin                    //cfltu : Convert
                        Floating to Unsigned Integer
                            int_delay[0] <= 0;
                            for (int i=0; i<4; i=i+1) begin
                                scale = 173 - $unsigned(imm[10:17]);
                                if (scale > 127)
                                    scale = 127;
                                else if (scale < 0)
                                    scale = 0;
                                tempfp = $bitstoshortreal(ra[(i*32) +: 32]) *
                                (2**scale);// << scale;
                                if (tempfp > (2**31 - 1))
                                    rt_delay[0][(i*32) +: 32] <= 32'h7FFFFFFF;
                                else if (tempfp < 0)
                                    rt_delay[0][(i*32) +: 32] <= 0;
                                else
                                    rt_delay[0][(i*32) +: 32] <= int'(tempfp);
                            end
                        end
                        default begin
                            int_delay[0] <= 0;
                            rt_delay[0] <= 0;
                            rt_addr_delay[0] <= 0;
                            reg_write_delay[0] <= 0;
                        end
                    endcase
                end
                else if (format == 4) begin
                    case (op[3:10])
                        8'b01110100 : begin                   //mpyi : Multiply Immediate
                            int_delay[0] <= 1;
                            for (int i=0; i<4; i=i+1)
                                rt_delay[0][(i*32) +: 32] <= $signed(ra[(i*32)+16 +:
                                16]) * $signed(imm[8:17]);
                        end
                        8'b01110101 : begin                   //mpyui : Multiply Unsigned
                        Immediate
                            int_delay[0] <= 1;
                            temp16 = $signed(imm[8:17]);
                            for (int i=0; i<4; i=i+1)
                                rt_delay[0][(i*32) +: 32] <= $unsigned(ra[(i*32)+16 +:
                                16]) * $unsigned(temp16);
                        end
                        default begin
                            int_delay[0] <= 0;
                            rt_delay[0] <= 0;
                            rt_addr_delay[0] <= 0;
                            reg_write_delay[0] <= 0;
                        end
                    endcase
                end
            end
        end
    end
endmodule
```

```
module SimpleFixed2(clk, reset, op, format, rt_addr, ra, rb, imm, reg_write, rt_wb, rt_addr_wb, reg_write_wb, branch_taken, rt_addr_delay, reg_write_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb;         //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    //Internal Signals
    logic [3:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [3:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [3:0]      reg_write_delay;    //Will rt_wb write to RegTable

    logic [6:0]         i;                  //7-bit counter for loops
    logic [0:127] tmp,s;

    always_comb begin
        rt_wb = rt_delay[2];
        rt_addr_wb = rt_addr_delay[2];
        reg_write_wb = reg_write_delay[2];
    end

    always_ff @(posedge clk) begin
        if (reset == 1) begin
            rt_delay[3] <= 0;
            rt_addr_delay[3] <= 0;
            reg_write_delay[3] <= 0;
            for (i=0; i<3; i=i+1) begin
                rt_delay[i] <= 0;
                rt_addr_delay[i] <= 0;
                reg_write_delay[i] <= 0;
            end
        end
        else begin
            rt_delay[3] <= rt_delay[2];
            rt_addr_delay[3] <= rt_addr_delay[2];
            reg_write_delay[3] <= reg_write_delay[2];

            rt_delay[2] <= rt_delay[1];
            rt_addr_delay[2] <= rt_addr_delay[1];
            reg_write_delay[2] <= reg_write_delay[1];

            rt_delay[1] <= rt_delay[0];
            rt_addr_delay[1] <= rt_addr_delay[0];
            reg_write_delay[1] <= reg_write_delay[0];

            if (format == 0 && op == 0) begin                   //nop : No Operation (Execute)
                rt_delay[0] <= 0;
                rt_addr_delay[0] <= 0;
                reg_write_delay[0] <= 0;
            end
            else begin
                rt_addr_delay[0] <= rt_addr;
                reg_write_delay[0] <= reg_write;
                if (branch_taken) begin
                    rt_delay[0] <= 0;
                    rt_addr_delay[0] <= 0;
                    reg_write_delay[0] <= 0;
```

```
68                  end
69                  else if (format == 0) begin
70                      case (op)
71                          11'b00001011111 : begin                 //shlh : Shift Left
                            Halfword
72                              for (i=0; i<8; i=i+1) begin
73                                  if ((rb[(i*16) +: 16] & 16'h001F) < 16) begin
74                                      rt_delay[0][(i*16) +: 16] <= ra[(i*16) +: 16] <<
                                        (rb[(i*16) +: 16] & 16'h001F);
75                                  end
76                                  else
77                                      rt_delay[0][(i*16) +: 16] <= 0;
78                              end
79                          end
80                          11'b00001011011 : begin                 //shl rt, ra, rb :
                            Shift Left Word
81                              for (i=0; i<16; i=i+4) begin
82                                  if ((rb[(i*8) +: 32] & 32'h0000003F) < 32) begin
83                                      rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] <<
                                        (rb[(i*8) +: 32] & 32'h0000003F);
84                                  end
85                                  else begin
86                                      rt_delay[0][(i*8) +: 32] = 0;
87                                  end
88                              end
89                          end
90                          11'b00001011100 : begin                 //roth rt, ra, rb :
                            Rotate Halfword
91                              for(i = 0;i<=15;i=i+2) begin
92                                  tmp[0:15] = ra[(i*8) +: 16];
93                                  for(int b = 0;b<16;b=b+1) begin
94                                      if( ( b+(rb[(i*8) +: 16] & 16'h000F)) < 16 ) begin
95                                          rt_delay[0][(i*8)+b] = tmp[b+(rb[(i*8) +: 16] &
                                            16'h000F)];
96                                      end
97                                      else begin
98                                          rt_delay[0][(i*8)+b] = tmp[b+(rb[(i*8) +: 16] &
                                            16'h000F)-16];
99                                      end
100                                 end
101                             end
102                         end
103                         11'b00001011000 : begin                 //rot rt, ra, rb :
                            Rotate Word
104
105                             for(i = 0;i<=15;i=i+4) begin
106                                 tmp[0:31] = ra[(i*8) +: 32];
107                                 for(int b = 0;b<32;b=b+1) begin
108                                     if( ( b+(rb[(i*8) +: 32] & 32'h0000001F)) < 32 )
                                        begin
109                                         rt_delay[0][(i*8)+b] = tmp[b+(rb[(i*8) +: 32] &
                                            32'h0000001F)];
110                                     end
111                                     else begin
112                                         rt_delay[0][(i*8)+b] = tmp[b+(rb[(i*8) +: 32] &
                                            32'h0000001F)-32];
113                                     end
114                                 end
115                             end
116                         end
117                         11'b00001011101  : begin                    //rothm rt, ra, rb
                            : Rotate and Mask Halfword
118
119                             for(i = 0;i<=15;i=i+2) begin
120                                 tmp[0:15] = ra[(i*8) +: 16];
121                                 for(int b = 0;b<16;b=b+1) begin
122                                     if( b > (( 0- rb[(i*8) +: 16] ) & 16'h001F)) begin
123                                         rt_delay[0][(i*8)+b] = tmp[b-(( 0- rb[(i*8) +:
                                            16] ) & 16'h001F)];
```

```
124                                 end
125                                 else begin
126                                     rt_delay[0][(i*8)+b] =0;
127                                 end
128                             end
129                         end
130                     end
131                     11'b00001011001  : begin                    //rotm rt, ra, rb :
                    Rotate and Mask Word
132
133                         for(i = 0;i<=15;i=i+4) begin
134                             tmp[0:31] = ra[(i*8) +: 32];
135                             for(int b = 0;b<32;b=b+1) begin
136                                 if( b > (( 0- rb[(i*8) +: 32] ) & 32'h0000003F))
                                    begin
137                                     rt_delay[0][(i*8)+b] = tmp[b-(( 0- rb[(i*8) +:
                                        32] ) & 32'h0000003F)];
138                                 end
139                                 else begin
140                                     rt_delay[0][(i*8)+b] =0;
141                                 end
142                             end
143                         end
144                     end
145                     11'b00001011110  : begin                    //rotmah rt, ra, rb
                    : Rotate and Mask Algebraic Halfword
146
147                         for(i = 0;i<=15;i=i+2) begin
148                             tmp[0:15] = ra[(i*8) +: 16];
149                             for(int b = 0;b<16;b=b+1) begin
150                                 if( b >= (( 0- rb[(i*8) +: 16] ) & 16'h001F)) begin
151                                     rt_delay[0][(i*8)+b] = tmp[b-(( 0- rb[(i*8) +:
                                        16] ) & 16'h001F)];
152                                 end
153                                 else begin
154                                     rt_delay[0][(i*8)+b] =tmp[0];
155                                 end
156                             end
157                         end
158                     end
159                     11'b00001011010  : begin                    //rotma rt, ra, rb
                    : Rotate and Mask Algebraic Word
160
161                         for(i = 0;i<=15;i=i+4) begin
162                             tmp[0:31] = ra[(i*8) +: 32];
163                             for(int b = 0;b<32;b=b+1) begin
164                                 if( b > (( 0- rb[(i*8) +: 32] ) & 32'h0000003F))
                                    begin
165                                     rt_delay[0][(i*8)+b] = tmp[b-(( 0- rb[(i*8) +:
                                        32] ) & 32'h0000003F)];
166                                 end
167                                 else begin
168                                     rt_delay[0][(i*8)+b] =tmp[0];
169                                 end
170                             end
171                         end
172                     end
173                     default begin
174                         rt_delay[0] <= 0;
175                         rt_addr_delay[0] <= 0;
176                         reg_write_delay[0] <= 0;
177                     end
178                 endcase
179             end
180             else if (format == 2) begin
181                 case (op)
182                     11'b00001111011 : begin //shli rt, ra, imm7 : Shift Left Word
                    Immediate
183                         for(int i=0;i<26;i=i+1) begin
```

```
184                             s[i] =  imm[11];
185                         end
186                         s[26:31] = imm[11:17];
187                         for(i = 0;i<=15;i=i+4) begin
188                             tmp[0:31] = ra[(i*8) +: 32];
189                             for(int b = 0;b<32;b=b+1) begin
190                                 if( b+(s[0:31] & 32'h0000003F) < 32 ) begin
191                                     rt_delay[0][(i*8)+b] = tmp[b+(s[0:31] &
                                        32'h0000003F)];
192                                 end
193                                 else begin
194                                     rt_delay[0][(i*8)+b] = 0;
195                                 end
196                             end
197                         end
198                     end
199
200                     11'b00001111100: begin // rothi rt, ra, imm7 :   Rotate
                    Halfword Immediate
201                         for(int i=0;i<9;i=i+1) begin
202                             s[i] =  imm[11];
203                         end
204                         s[9:15] = imm[11:17];
205
206                         for(i = 0;i<=15;i=i+2) begin
207                             tmp[0:15] = ra[(i*8) +: 16];
208                             for(int b = 0;b<16;b=b+1) begin
209                                 if( ( b+(s[0:15] & 16'h000F)) < 16 ) begin
210                                     rt_delay[0][(i*8)+b] = tmp[b+(s[0:15] &
                                        16'h000F)];
211                                 end
212                                 else begin
213                                     rt_delay[0][(i*8)+b] = tmp[b+(s[0:15] &
                                        16'h000F)-16];
214                                 end
215                             end
216                         end
217                     end
218                     11'b00001111000 : begin         //roti rt, ra, imm7 : Rotate
                    Word Immediate
219                         for(int i=0;i<26;i=i+1) begin
220                             s[i] =  imm[11];
221                         end
222                         s[26:31] = imm[11:17];
223                         for(i = 0;i<=15;i=i+4) begin
224                             tmp[0:31] = ra[(i*8) +: 32];
225                             for(int b = 0;b<32;b=b+1) begin
226                                 if( ( b+(s[0:31] & 32'h0000001F)) < 32 ) begin
227                                     rt_delay[0][(i*8)+b] = tmp[b+(s[0:31] &
                                        32'h0000001F)];
228                                 end
229                                 else begin
230                                     rt_delay[0][(i*8)+b] = tmp[b+(s[0:31]  &
                                        32'h0000001F)-32];
231                                 end
232                             end
233                         end
234                     end
235                     11'b00001111110  : begin    //rotmahi rt, ra, imm7 : Rotate and
                    Mask Algebraic Halfword Immediate
236
237                         for(int i=0;i<9;i=i+1) begin
238                             s[i] =  imm[11];
239                         end
240                         s[9:15] = imm[11:17];
241
242                         for(i = 0;i<=15;i=i+2) begin
243                             tmp[0:15] = ra[(i*8) +: 16];
244                             for(int b = 0;b<16;b=b+1) begin
```

```
245                                     if( b >= ((0-s[0:15]) & 16'h001F)) begin
246                                         rt_delay[0][(i*8)+b] = tmp[b-((0-s[0:15]) &
                                            16'h001F)];
247                                     end
248                                     else begin
249                                         rt_delay[0][(i*8)+b] =tmp[0];
250                                     end
251                                 end
252                             end
253                         end
254                         11'b00001111010 : begin //rotmai rt, ra, imm7 : Rotate and Mask
                            Algebraic Word Immediate
255                             for(int i=0;i<26;i=i+1) begin
256                                 s[i] =  imm[11];
257                             end
258                             s[26:31] = imm[11:17];
259                             for(i = 0;i<=15;i=i+4) begin
260                                 tmp[0:31] = ra[(i*8) +: 32];
261                                 for(int b = 0;b<32;b=b+1) begin
262                                     if( b>=((0-s[0:31]) & 32'h0000003F)) begin
263                                         rt_delay[0][(i*8)+b] = tmp[b-((0-s[0:31]) &
                                            32'h0000003F)];
264                                     end
265                                     else begin
266                                         rt_delay[0][(i*8)+b] = tmp[0];
267                                     end
268                                 end
269                             end
270                         end
271                         default begin
272                             rt_delay[0] = 0;
273                             rt_addr_delay[0] = 0;
274                             reg_write_delay[0] = 0;
275                         end
276                     endcase
277                 end
278             end
279         end
280     end
281
282 endmodule
```

```
module Byte(clk, reset, op, format, rt_addr, ra, rb, imm, reg_write, rt_wb, rt_addr_wb,
reg_write_wb, branch_taken, rt_addr_delay, reg_write_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb;         //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    //Internal Signals
    logic [3:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [3:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [3:0]      reg_write_delay;    //Will rt_wb write to RegTable

    logic [3:0]         temp;               //Incrementing counter

    always_comb begin
        rt_wb = rt_delay[2];
        rt_addr_wb = rt_addr_delay[2];
        reg_write_wb = reg_write_delay[2];
    end

    always_ff @(posedge clk) begin
        if (reset == 1) begin

            rt_delay[3] <= 0;
            rt_addr_delay[3] <= 0;
            reg_write_delay[3] <= 0;
            for (int i=0; i<3; i=i+1) begin
                rt_delay[i] <= 0;
                rt_addr_delay[i] <= 0;
                reg_write_delay[i] <= 0;
            end
        end
        else begin
            rt_delay[3] <= rt_delay[2];
            rt_addr_delay[3] <= rt_addr_delay[2];
            reg_write_delay[3] <= reg_write_delay[2];
            rt_delay[2] <= rt_delay[1];
            rt_addr_delay[2] <= rt_addr_delay[1];
            reg_write_delay[2] <= reg_write_delay[1];
            rt_delay[1] <= rt_delay[0];
            rt_addr_delay[1] <= rt_addr_delay[0];
            reg_write_delay[1] <= reg_write_delay[0];

            if (format == 0 && op == 0) begin                   //nop : No Operation
            (Execute)
                rt_delay[0] <= 0;
                rt_addr_delay[0] <= 0;
                reg_write_delay[0] <= 0;
            end
            else begin
                rt_addr_delay[0] <= rt_addr;
                reg_write_delay[0] <= reg_write;
                if (branch_taken) begin
                    rt_delay[0] <= 0;
                    rt_addr_delay[0] <= 0;
                    reg_write_delay[0] <= 0;
                end
                else if (format == 0) begin
```

```
68                      case (op)
69                          11'b01010110100 : begin                 //cntb : Count Ones in
                            Bytes
70                              for (int i=0; i<16; i=i+1) begin
71                                  temp = 0;
72                                  for (int j=0; j<8; j=j+1) begin
73                                      if (ra[(i*8)+j] == 1'b1)
74                                          temp = temp + 1;
75                                  end
76                                  rt_delay[0][(i*8) +: 8] <= temp;
77                              end
78                          end
79                          11'b00011010011 : begin                 //avgb : Average Bytes
80                              for (int i=0; i<16; i=i+1) begin
81                                  rt_delay[0][(i*8) +: 8] <= ({2'b00, ra[(i*8) +: 8]} +
                                    {2'b00, rb[(i*8) +: 8]} + 1) >> 1;
82                              end
83                          end
84                          11'b00001010011 : begin                 //absdb : Absolute
                            Difference of Bytes
85                              for (int i=0; i<16; i=i+1) begin
86                                  if ($signed(ra[(i*8) +: 8]) > $signed(rb[(i*8) +: 8]))
87                                      rt_delay[0][(i*8) +: 8] <= $signed(ra[(i*8) +: 8])
                                        - $signed(rb[(i*8) +: 8]);
88                                  else
89                                      rt_delay[0][(i*8) +: 8] <= $signed(rb[(i*8) +: 8])
                                        - $signed(ra[(i*8) +: 8]);
90                              end
91                          end
92                          11'b01001010011 : begin                 //sumb : Sum Bytes into
                            Halfwords
93                              for (int i=0; i<4; i=i+1) begin
94                                  rt_delay[0][(i*32) +: 16] <= $signed(rb[(i*32) +: 8]) +
                                    $signed(rb[(i*32)+8 +: 8]) + $signed(rb[(i*32)+16 +:
                                    8]) + $signed(rb[(i*32)+24 +: 8]);
95                                  rt_delay[0][(i*32)+16 +: 16] <= $signed(ra[(i*32) +:
                                    8]) + $signed(ra[(i*32)+8 +: 8]) + $signed(ra[(i*32)+16
                                    +: 8]) + $signed(ra[(i*32)+24 +: 8]);
96                              end
97                          end
98                          default begin
99                              rt_delay[0] <= 0;
100                             rt_addr_delay[0] <= 0;
101                             reg_write_delay[0] <= 0;
102                         end
103                     endcase
104                 end
105                 else begin
106                     rt_delay[0] <= 0;
107                     rt_addr_delay[0] <= 0;
108                     reg_write_delay[0] <= 0;
109                 end
110             end
111         end
112     end
113
114 endmodule
```

```
module SimpleFixed1(clk, reset, op, format, rt_addr, ra, rb, rt_st, imm, reg_write, rt_wb, rt_addr_wb, reg_write_wb, branch_taken, rt_addr_delay, reg_write_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb, rt_st;      //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    //Internal Signals
    logic [1:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [1:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [1:0]      reg_write_delay;    //Will rt_wb write to RegTable

    logic [6:0]     i;              //7-bit counter for loops
    logic signed [31:0] max_value_32 = 32'h7FFFFFFF;
    logic signed [31:0] min_value_32 = 32'h80000000;

    logic signed [15:0] max_value_16 = 16'h7FFF;
    logic signed [15:0] min_value_16 = 16'h8000;
    logic signed [0:31] mask = 1 << 31;

    logic [0:128] tmp;

    always_comb begin
        rt_wb = rt_delay[0];
        rt_addr_wb = rt_addr_delay[0];
        reg_write_wb = reg_write_delay[0];
    end

    always_ff @(posedge clk) begin
        if (reset == 1) begin
            rt_delay[1] <= 0;
            rt_addr_delay[1] <= 0;
            reg_write_delay[1] <= 0;
            rt_delay[0] <= 0;
            rt_addr_delay[0] <= 0;
            reg_write_delay[0] <= 0;
            tmp = 0;
        end
        else begin
            rt_delay[1] <= rt_delay[0];
            rt_addr_delay[1] <= rt_addr_delay[0];
            reg_write_delay[1] <= reg_write_delay[0];

            if (format == 0 && op == 0) begin                   //nop : No Operation (Execute)
                rt_delay[0] <= 0;
                rt_addr_delay[0] <= 0;
                reg_write_delay[0] <= 0;
            end
            else begin
                rt_addr_delay[0] <= rt_addr;
                reg_write_delay[0] <= reg_write;
                if (branch_taken) begin
                    rt_delay[0] = 0;
                    rt_addr_delay[0] = 0;
                    reg_write_delay[0] = 0;
                end
                else if (format == 0) begin
```

```
68                      case (op)
69                          11'b00011001000 : begin                 //ah : Add Halfword
70                              for (i=0; i<16; i=i+1) begin
71                                  if (($signed(ra[(i * 16) +:16]) + $signed(rb[(i * 16)
                                    +:16])) >= max_value_16)
72                                      rt_delay[0][(i * 16) +:16] = max_value_16;
73                                  else if (($signed(ra[(i * 16) +:16]) + $signed(rb[(i *
                                    16) +:16])) <= min_value_16)
74                                      rt_delay[0][(i * 16) +:16] = min_value_16;
75                                  else
76                                      rt_delay[0][(i * 16) +:16] = $signed(ra[(i * 16)
                                        +:16]) + $signed(rb[(i * 16) +:16]);
77                              end
78                          end
79                          11'b00011000000 : begin                 //ah : Add Word
80                              for (i=0; i<32; i=i+1) begin
81                                  if (($signed(ra[(i * 32) +:32]) + $signed(rb[(i * 32)
                                    +:32])) >= max_value_32)
82                                      rt_delay[0][(i * 32) +:32] = max_value_32;
83                                  else if (($signed(ra[(i * 32) +:32]) + $signed(rb[(i *
                                    32) +:32])) <= min_value_32)
84                                      rt_delay[0][(i * 32) +:32] = min_value_32;
85                                  else
86                                      rt_delay[0][(i * 32) +:32] = $signed(ra[(i * 32)
                                        +:32]) + $signed(rb[(i * 32) +:32]);
87                              end
88                          end
89                          11'b00001001000 : begin     //sfh rt, ra, rb : Subtract from
                            Halfword
90                              for (i=0; i<16; i=i+1) begin
91                                  if (($signed(rb[(i * 16) +:16]) - $signed(ra[(i * 16)
                                    +:16])) >= max_value_16)
92                                      rt_delay[0][(i * 16) +:16] = max_value_16;
93                                  else if ($signed(rb[(i * 16) +:16]) - $signed(ra[(i *
                                    16) +:16]) <= min_value_16)
94                                      rt_delay[0][(i * 16) +:16] = min_value_16;
95                                  else
96                                      rt_delay[0][(i * 16) +:16] = $signed(rb[(i * 16)
                                        +:16]) - $signed(ra[(i * 16) +:16]);
97                              end
98                          end
99                          11'b00001000000 : begin     //sf rt, ra, rb : Subtract from Word
100                             for (i=0; i<32; i=i+1) begin
101                                 if ($signed(rb[(i * 32) +:32]) - $signed(ra[(i * 32)
                                    +:32]) >= max_value_32)
102                                     rt_delay[0][(i * 32) +:32] = max_value_32;
103                                 else if ($signed(rb[(i * 32) +:32]) - $signed(ra[(i *
                                    32) +:32]) <= min_value_32)
104                                     rt_delay[0][(i * 32) +:32] = min_value_32;
105                                 else
106                                     rt_delay[0][(i * 32) +:32] = $signed(rb[(i * 32)
                                        +:32]) - $signed(ra[(i * 32) +:32]);
107                             end
108                         end
109                         11'b00011000001 : begin // and
110                             rt_delay[0] =  ra & rb;
111                         end
112                         11'b00001000001 : begin // or
113                             rt_delay[0] =  ra | rb;
114                         end
115                         11'b01001000001 : begin // xor
116                             rt_delay[0] =  ra ^ rb;
117                         end
118                         11'b00011001001 : begin // nand
119                             rt_delay[0] =  ~(ra & rb);
120                         end
121                         11'b01111010000 : begin // ceqb rt, ra, rb Compare Equal Byte
122                             if(ra==rb) begin
123                                 rt_delay[0]=128'hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF;
```

```
124                         end
125                         else begin
126                             rt_delay[0]=128'h00000000000000000000000000000000;
127                         end
128                     end
129                     11'b01111001000 : begin // ceqh rt, ra, rb Compare Equal Halfword
130                         for(int i = 0;i<16;i=i+2) begin
131                             if(ra[(i*8) +: 16]==rb[(i*8) +: 16]) begin
132                                 rt_delay[0][(i*8) +: 16] = 16'hFFFF;
133                             end
134                             else begin
135                                 rt_delay[0][(i*8) +: 16] = 16'h0000;
136                             end
137                         end
138 
139                     end
140                     11'b01111000000 : begin // ceq rt, ra, rb Compare Equal Word
141                         for(int i = 0;i<16;i=i+4) begin
142                             if(ra[(i*8) +: 32]==rb[(i*8) +: 32]) begin
143                                 rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
144                             end
145                             else begin
146                                 rt_delay[0][(i*8) +: 32] = 32'h00000000;
147                             end
148                         end
149                     end
150                     11'b01001010000 : begin // cgtb rt, ra, rb Compare Greater Than
                    Byte
151                         for(int i = 0;i<16;i=i+1) begin
152                             if($signed(ra[(i*8) +: 8])>$signed(rb[(i*8) +: 8])) begin
153                                 rt_delay[0][(i*8) +: 8] = 8'hFF;
154                             end
155                             else begin
156                                 rt_delay[0][(i*8) +: 8] = 8'h00;
157                             end
158                         end
159                     end
160                     11'b01001001000 : begin // cgth rt, ra, rb Compare Greater Than
                    Halfword
161                         for(int i = 0;i<16;i=i+2) begin
162                             if($signed(ra[(i*8) +: 16])>$signed(rb[(i*8) +: 16]))
                            begin
163                                 rt_delay[0][(i*8) +: 16] = 16'hFFFF;
164                             end
165                             else begin
166                                 rt_delay[0][(i*8) +: 16] = 16'h0000;
167                             end
168                         end
169                     end
170                     11'b01001000000 : begin // cgt rt, ra, rb Compare Greater Than
                    Word
171                         for(int i = 0;i<16;i=i+4) begin
172                             if($signed(ra[(i*8) +: 32])>$signed(rb[(i*8) +: 32]))
                            begin
173                                 rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
174                             end
175                             else begin
176                                 rt_delay[0][(i*8) +: 32] = 32'h00000000;
177                             end
178                         end
179                     end
180                     11'b01011010000 : begin // clgtb rt, ra, rb Compare Logical
                    Greater Than Byte
181                         for(int i = 0;i<16;i=i+1) begin
182                             if(ra[(i*8) +: 8]>rb[(i*8) +: 8]) begin
183                                 rt_delay[0][(i*8) +: 8] = 8'hFF;
184                             end
185                             else begin
186                                 rt_delay[0][(i*8) +: 8] = 8'h00;
```

```
                        end
                    end
                end
                11'b01011001000 : begin // clgth rt, ra, rb Compare Logical
                Greater Than Halfword
                    for(int i = 0;i<16;i=i+2) begin
                        if(ra[(i*8) +: 16]>rb[(i*8) +: 16]) begin
                            rt_delay[0][(i*8) +: 16] = 16'hFFFF;
                        end
                        else begin
                            rt_delay[0][(i*8) +: 16] = 16'h0000;
                        end
                    end
                end
                11'b01011000000 : begin // clgt rt, ra, rb Compare Logical
                Greater Than Word
                    for(int i = 0;i<16;i=i+4) begin
                        if(ra[(i*8) +: 32]>rb[(i*8) +: 32]) begin
                            rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
                        end
                        else begin
                            rt_delay[0][(i*8) +: 32] = 32'h00000000;
                        end
                    end
                end
                default begin
                    rt_delay[0] <= 0;
                    rt_addr_delay[0] <= 0;
                    reg_write_delay[0] <= 0;
                end
            endcase
        end
        else if (format == 4) begin //RI10-type
            case (op)
                8'b00011101 : begin                 //ahi rt, ra, imm10 : Add
                Halfword Immediate

                    for (i=0; i<16; i=i+1) begin

                        if (($signed(ra[(i * 16) +:16]) + $signed(imm[8:17]))
                        >= max_value_16)
                            rt_delay[0][(i * 16) +:16] = max_value_16;
                        else if (($signed(ra[(i * 16) +:16]) +
                        $signed(imm[8:17])) <= min_value_16)
                            rt_delay[0][(i * 16) +:16] = min_value_16;
                        else
                            rt_delay[0][(i * 16) +:16] = $signed(ra[(i * 16)
                            +:16]) + $signed(imm[8:17]);
                    end
                end
                8'b00011100 : begin                 //ai rt, ra, imm10 : Add
                Word Immediate
                    for (i=0; i<4; i=i+1) begin
                        if (($signed(ra[(i * 32) +:32]) + $signed(imm[8:17]))
                        >= max_value_32)
                            rt_delay[0][(i * 32) +:32] = max_value_32;
                        else if (($signed(ra[(i * 32) +:32]) +
                        $signed(imm[8:17])) <= min_value_32)
                            rt_delay[0][(i * 32) +:32] = min_value_32;
                        else
                            rt_delay[0][(i * 32) +:32] = $signed(ra[(i * 32)
                            +:32]) + $signed(imm[8:17]);


                    end
                end
                8'b00001101 : begin     //sfhi rt, ra, imm10 : Subtract from
                Halfword Immediate
                    for (i=0; i<16; i=i+1) begin
```

```
245                             if (($signed(imm[8:17]) - $signed(ra[(i * 16) +:16]))
                                >= max_value_16)
246                                 rt_delay[0][(i * 16) +:16] = max_value_16;
247                             else if (($signed(imm[8:17]) - $signed(ra[(i * 16)
                                +:16])) <= min_value_16)
248                                 rt_delay[0][(i * 16) +:16] = min_value_16;
249                             else
250                                 rt_delay[0][(i * 16) +:16] = $signed(imm[8:17]) -
                                    $signed(ra[(i * 16) +:16]);
251                         end
252                     end
253                     8'b00001100 : begin     //sfi rt, ra, imm10 : Subtract from
                    Word Immediate
254                         for (i=0; i<32; i=i+1) begin
255                             if (($signed(imm[8:17]) - $signed(ra[(i * 32) +:32]))
                                >= max_value_32)
256                                 rt_delay[0][(i * 32) +:32] = max_value_32;
257                             else if (($signed(imm[8:17]) - $signed(ra[(i * 32)
                                +:32])) <= min_value_32)
258                                 rt_delay[0][(i * 32) +:32] = min_value_32;
259                             else
260                                 rt_delay[0][(i * 32) +:32] = $signed(imm[8:17]) -
                                    $signed(ra[(i * 32) +:32]);
261                         end
262                     end
263                     8'b00010110 : begin // andbi rt, ra, imm10 And Byte Immediate
264                         for(int i=0;i<15;i=i+2) begin
265                             rt_delay[0][(i*8) +: 16] = ra[(i*8) +: 16] &
                                ((imm[10:17] & 8'hFF) | (imm[9:17] & 8'hFF) <<16) ;
266                         end
267                     end
268
269                     8'b00010101 : begin // andhi rt, ra, imm10 And Halfword Immediate
270                         for(int i=0;i<6;i=i+1) begin
271                             tmp[i] =  imm[8];
272                         end
273                         tmp[6:15] = imm[8:17];
274                         for(int i=0;i<15;i=i+2) begin
275                             rt_delay[0][(i*8) +: 16] = ra[(i*8) +: 16] & tmp[0:15];
276                         end
277                     end
278                     8'b00010100 : begin // andi rt, ra, imm10 And Word Immediate
279                         for(int i=0;i<21;i=i+1) begin
280                             tmp[i] =  imm[8];
281                         end
282                         tmp[21:31] = imm[8:17];
283                         for(int i=0;i<15;i=i+4) begin
284                             rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] & tmp[0:31];
285                         end
286                     end
287
288                     8'b00000110 : begin // orbi rt, ra, imm10 Or Byte Immediate
289                         for(int i=0;i<15;i=i+4) begin
290                             rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] |
                                ({(imm[10:17] & 8'hFF) ,(imm[10:17] &
                                8'hFF),(imm[10:17] & 8'hFF),(imm[10:17] & 8'hFF)}) ;
291                         end
292                     end
293
294                     8'b00000101 : begin // orhi rt, ra, imm10 Or Halfword Immediate
295                         for(int i=0;i<6;i=i+1) begin
296                             tmp[i] =  imm[8];
297                         end
298                         tmp[6:15] = imm[8:17];
299                         for(int i=0;i<15;i=i+2) begin
300                             rt_delay[0][(i*8) +: 16] = ra[(i*8) +: 16] | tmp[0:15];
301                         end
302                     end
303                     8'b00000100 : begin // ori rt, ra, imm10 Or Word Immediate
```

```
304                             for(int i=0;i<21;i=i+1) begin
305                                 tmp[i] =  imm[8];
306                             end
307                             tmp[21:31] = imm[8:17];
308                             for(int i=0;i<15;i=i+4) begin
309                                 rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] | tmp[0:31];
310                             end
311                         end
312
313
314                         8'b01000110 : begin // xorbi rt, ra, imm10 Exclusive Or Byte
                            Immediate
315                             for(int i=0;i<15;i=i+4) begin
316                                 rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] ^
                                    ({(imm[10:17] & 8'hFF) ,(imm[10:17] &
                                    8'hFF),(imm[10:17] & 8'hFF),(imm[10:17] & 8'hFF)}) ;
317                             end
318                         end
319
320                         8'b01000101 : begin // xorhi rt, ra, imm10 Xor Halfword Immediate
321                             for(int i=0;i<6;i=i+1) begin
322                                 tmp[i] =  imm[8];
323                             end
324                             tmp[6:15] = imm[8:17];
325                             for(int i=0;i<15;i=i+2) begin
326                                 rt_delay[0][(i*8) +: 16] = ra[(i*8) +: 16] ^ tmp[0:15];
327                             end
328                         end
329                         8'b01000100 : begin // xori rt, ra, imm10 Xor Word Immediate
330                             for(int i=0;i<21;i=i+1) begin
331                                 tmp[i] =  imm[8];
332                             end
333                             tmp[21:31] = imm[8:17];
334                             for(int i=0;i<15;i=i+4) begin
335                                 rt_delay[0][(i*8) +: 32] = ra[(i*8) +: 32] ^ tmp[0:31];
336                             end
337                         end
338
339                         8'b01111110 : begin // ceqbi rt, ra, imm10 Compare Equal Byte
                            Immediate
340                             for(int i=0;i<16;i=i+1) begin
341                                 if(ra[(i*8) +: 8] == imm[10:17]) begin
342                                     rt_delay[0][(i*8) +: 8] = 8'hFF;
343                                 end
344                                 else begin
345                                     rt_delay[0][(i*8) +: 8] = 8'h00;
346                                 end
347                             end
348                         end
349
350
351                         8'b01111101 : begin // ceqhi rt, ra, imm10 Compare Equal
                            Halfword Immediate
352                             for(int i=0;i<6;i=i+1) begin
353                                 tmp[i] =  imm[8];
354                             end
355                             tmp[6:15] = imm[8:17];
356
357                             for(int i=0;i<16;i=i+2) begin
358                                 if(ra[(i*8) +: 16] == tmp[0:15]) begin
359                                     rt_delay[0][(i*8) +: 16] = 16'hFFFF;
360                                 end
361                                 else begin
362                                     rt_delay[0][(i*8) +: 16] = 16'h0000;
363                                 end
364                             end
365                         end
366
367                         8'b01111100 : begin // ceqi rt, ra, imm10 Compare Equal Word
```

```
                    Immediate
368                     for(int i=0;i<22;i=i+1) begin
369                         tmp[i] =  imm[8];
370                     end
371                     tmp[22:31] = imm[8:17];
372                     for(int i=0;i<16;i=i+4) begin
373                         if(ra[(i*8) +: 32] == tmp[0:31]) begin
374                             rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
375                         end
376                         else begin
377                             rt_delay[0][(i*8) +: 32] = 32'h00000000;
378                         end
379                     end
380                 end
381
382                 8'b01001110 : begin // cgtbi rt, ra, imm10 Compare Greater Than
                    Byte Immediate
383                     for(int i=0;i<16;i=i+1) begin
384                         if($signed(ra[(i*8) +: 8]) > $signed(imm[10:17])) begin
385                             rt_delay[0][(i*8) +: 8] = 8'hFF;
386                         end
387                         else begin
388                             rt_delay[0][(i*8) +: 8] = 8'h00;
389                         end
390                     end
391                 end
392
393
394                 8'b01001101 : begin // cgthi rt, ra, imm10 Compare Greater Than
                    Halfword Immediate
395                     for(int i=0;i<6;i=i+1) begin
396                         tmp[i] =  imm[8];
397                     end
398                     tmp[6:15] = imm[8:17];
399                     for(int i=0;i<16;i=i+2) begin
400                         if($signed(ra[(i*8) +: 16]) > $signed(tmp[0:15])) begin
401                             rt_delay[0][(i*8) +: 16] = 16'hFFFF;
402                         end
403                         else begin
404                             rt_delay[0][(i*8) +: 16] = 16'h0000;
405                         end
406                     end
407                 end
408
409                 8'b01001100 : begin // cgti rt, ra, imm10 Compare Greater Than
                    Word Immediate
410                     for(int i=0;i<22;i=i+1) begin
411                         tmp[i] =  imm[8];
412                     end
413                     tmp[22:31] = imm[8:17];
414
415                     for(int i=0;i<16;i=i+4) begin
416                         if($signed(ra[(i*8) +: 32]) > $signed(tmp[0:31])) begin
417                             rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
418                         end
419                         else begin
420                             rt_delay[0][(i*8) +: 32] = 32'h00000000;
421                         end
422                     end
423                 end
424
425
426                 8'b01011110 : begin //clgtbi rt, ra, imm10 Compare Logical
                    Greater Than Byte Immediate
427                     for(int i=0;i<16;i=i+1) begin
428                         if($unsigned(ra[(i*8) +: 8]) > $unsigned(imm[10:17]))
                        begin
429                             rt_delay[0][(i*8) +: 8] = 8'hFF;
430                         end
```

```
                            else begin
                                rt_delay[0][(i*8) +: 8] = 8'h00;
                            end
                        end
                    end

                    8'b01011101 : begin // clgthi rt, ra, imm10 Compare Logical
                    Greater Than Halfword Immediate
                        for(int i=0;i<6;i=i+1) begin
                            tmp[i] =  imm[8];
                        end
                        tmp[6:15] = imm[8:17];
                        for(int i=0;i<16;i=i+2) begin
                            if($unsigned(ra[(i*8) +: 16]) > $unsigned(tmp[0:15]))
                            begin
                                rt_delay[0][(i*8) +: 16] = 16'hFFFF;
                            end
                            else begin
                                rt_delay[0][(i*8) +: 16] = 16'h0000;
                            end
                        end
                    end

                    8'b01011100 : begin // clgti rt, ra, imm10 Compare Logical
                    Greater Than Word Immediate

                        for(int i=0;i<22;i=i+1) begin
                            tmp[i] =  imm[8];
                        end
                        tmp[22:31] = imm[8:17];

                        for(int i=0;i<16;i=i+4) begin
                            if($unsigned(ra[(i*8) +: 32]) > $unsigned(tmp[0:31]))
                            begin
                                rt_delay[0][(i*8) +: 32] = 32'hFFFFFFFF;
                            end
                            else begin
                                rt_delay[0][(i*8) +: 32] = 32'h00000000;
                            end
                        end

                    end
                    default begin
                        rt_delay[0] = 0;
                        rt_addr_delay[0] = 0;
                        reg_write_delay[0] = 0;
                        tmp=0;
                    end
                endcase
            end
            else if (format == 5) begin
                case (op)
                    9'b010000011: begin // ilh rt, imm16 Immediate Load Halfword
                        for(int i=0;i<16;i=i+2) begin
                            rt_delay[0][(i*8) +: 16] = imm[2:17];
                        end
                    end
                    9'b010000010: begin // ilhu rt, imm16 Immediate Load Halfword
                    Upper
                        for(int i=0;i<16;i=i+4) begin
                            rt_delay[0][(i*8) +: 32] = {imm[2:17],16'h0000};
                        end
                    end
                    9'b010000001: begin // il rt, imm16 Immediate Load Word
                        for(int i=0;i<4;i++) begin
                            rt_delay[0][(i*32) +: 32] = $signed(imm[2:17]);
                        end
                    end
                    9'b011000001: begin // iohl rt, imm16 Immediate Or Halfword Lower
```

```
                            for(int i=0;i<16;i=i+4) begin
                                rt_delay[0][(i*8) +: 32] = rt_st[(i*8) +: 32] |
                                {16'h0000,imm[2:17]};
                            end
                        end

                        default begin
                            rt_delay[0] = 0;
                            rt_addr_delay[0] = 0;
                            reg_write_delay[0] = 0;
                            tmp=0;
                        end
                    endcase

                end
                else if (format == 6) begin
                    case (op)
                        7'b0100001: begin // ila rt, imm18 Immediate Load Address
                            for(int i=0;i<16;i=i+4) begin
                                rt_delay[0][(i*8) +: 32] =  {14'h0000,imm[0:17]};
                            end
                        end

                        default begin
                            rt_delay[0] = 0;
                            rt_addr_delay[0] = 0;
                            reg_write_delay[0] = 0;
                            tmp=0;
                        end
                    endcase
                end
            end
        end
    end
endmodule
```

```
module Permute(clk, reset, op, format, rt_addr, ra, rb, imm, reg_write, rt_wb, rt_addr_wb, reg_write_wb, branch_taken, rt_addr_delay, reg_write_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb;         //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    //Internal Signals
    logic [3:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [3:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [3:0]      reg_write_delay;    //Will rt_wb write to RegTable

    logic [6:0]         i;                  //7-bit counter for loops

    // Temp variables
    logic [0:127] tmp;

    always_comb begin
        rt_wb = rt_delay[2];
        rt_addr_wb = rt_addr_delay[2];
        reg_write_wb = reg_write_delay[2];
    end

    always_ff @(posedge clk) begin
        if (reset == 1) begin
            rt_delay[3] <= 0;
            rt_addr_delay[3] <= 0;
            reg_write_delay[3] <= 0;
            for (i=0; i<3; i=i+1) begin
                rt_delay[i] <= 0;
                rt_addr_delay[i] <= 0;
                reg_write_delay[i] <= 0;
            end
            tmp <= 0;

        end
        else begin
            rt_delay[3] <= rt_delay[2];
            rt_addr_delay[3] <= rt_addr_delay[2];
            reg_write_delay[3] <= reg_write_delay[2];

            rt_delay[2] <= rt_delay[1];
            rt_addr_delay[2] <= rt_addr_delay[1];
            reg_write_delay[2] <= reg_write_delay[1];

            rt_delay[1] <= rt_delay[0];
            rt_addr_delay[1] <= rt_addr_delay[0];
            reg_write_delay[1] <= reg_write_delay[0];

            if (format == 0 && op == 0) begin                   //nop : No Operation (Load)
                rt_delay[0] <= 0;
                rt_addr_delay[0] <= 0;
                reg_write_delay[0] <= 0;
            end
            else begin
                rt_addr_delay[0] <= rt_addr;
                reg_write_delay[0] <= reg_write;
```

```
68                  if (branch_taken) begin
69                      rt_delay[0] <= 0;
70                      rt_addr_delay[0] <= 0;
71                      reg_write_delay[0] <= 0;
72                  end
73                  else if (format == 0) begin
74                      case (op)
75                          11'b00111011011 : begin                 //shlqbi : Shift Left
                          Quadword by Bits
76                              rt_delay[0] <= ra << rb[29:31];
77                          end
78                          11'b00111011111 : begin                 //shlqby rt, ra, rb :
                          Shift Left Quadword by Bytes
79                              rt_delay[0] <= ra << (rb[27:31] * 8);
80                          end
81                          11'b00111011000 : begin                 //rotqbi rt, ra, rb :
                          Rotate Quadword by Bits
82                              tmp = rb[29:31];
83                              for(int b=0;b<128;b++) begin
84                                  if(b+tmp < 128 ) begin
85                                      rt_delay[0][b] = ra[b+tmp];
86                                  end
87                                  else begin
88                                      rt_delay[0][b]=ra[b+tmp-128];
89                                  end
90                              end
91                          end
92                          11'b00111011100 : begin                 //rotqby rt,ra,rb
                          Rotate Quadword by Bytes
93                              tmp = rb[28:31];
94
95                              for(int b=0;b<=15;b++) begin
96                                  if(b+tmp < 16 ) begin
97                                      for(int i = b*8;i<(b*8+8);i++) begin
98                                          rt_delay[0][i] = ra[i+tmp*8];
99                                      end
100
101                                 end
102                                 else begin
103                                     for(int i = b*8;i<(b*8+8);i++) begin
104                                         rt_delay[0][i] = ra[i+tmp*8-16*8];
105                                     end
106
107                                 end
108                             end
109                         end
110                         11'b00110110010 : begin                 //gbb rt,ra Gather Bits
                          from Bytes
111                             tmp = 0;
112                             for(int j=7,k=0;j<128;j=j+8,k++) begin
113                                 tmp[k+16] = ra[j];
114                             end
115
116                             for(int i = 32;i<128;i++) begin
117                                 rt_delay[0][i]=0;
118                             end
119                             rt_delay[0][0:31] = 16'h0000 | tmp[0:31];
120                         end
121                         11'b00110110001 : begin                 //gbb rt,ra Gather Bits
                          from halfwords
122                             tmp = 0;
123
124                             for(int j=15,k=0;j<128;j=j+16,k++) begin
125                                 tmp[k+24] = ra[j];
126                             end
127
128                             for(int i = 0;i<128;i++) begin
129                                 rt_delay[0][i]=0;
130                             end
```

```
131                         rt_delay[0][24:31] = 8'h00 | tmp[24:31];
132                     end
133                     11'b00110110000 : begin                 //gbb rt,ra Gather Bits
                    from halfwords
134                         tmp = 0;
135
136                         for(int j=31,k=0;j<128;j=j+32,k++) begin
137                             tmp[k+28] = ra[j];
138                         end
139
140                         for(int i = 0;i<128;i++) begin
141                             rt_delay[0][i]=0;
142                         end
143                         rt_delay[0][28:31] = 4'h0 | tmp[28:31];
144                     end
145                     default begin
146                         rt_delay[0] <= 0;
147                         rt_addr_delay[0] <= 0;
148                         reg_write_delay[0] <= 0;
149                     end
150                 endcase
151             end
152             else if (format == 2) begin
153                 case (op)
154                     11'b00111111011 : begin                 //shlqbi : Shift Left
                    Quadword by Bits
155                         tmp[0:6]  = imm & 7'b0000111;
156                         for(int b=0;b<128;b++) begin
157                             if(b+tmp[0:6] < 128 ) begin
158                                 rt_delay[0][b] = ra[b+tmp[0:6]];
159                             end
160                             else begin
161                                 rt_delay[0][b] = 0;
162                             end
163                         end
164                     end
165                     11'b00111111111 : begin                 //shlqbyi rt,ra,value
                    Shift Left Quadword by Bytes Immediate
166                         tmp[0:6]  = imm & 7'b0001111;
167                         for(int b=0;b<=15;b++) begin
168                             if(b+tmp[0:6] < 16 ) begin
169                                 for(int i = b*8;i<(b*8+8);i++) begin
170                                     rt_delay[0][i] = ra[i+tmp[0:6]*8];
171                                 end
172                             end
173                             else begin
174                                 for(int i = b*8;i<(b*8+8);i++) begin
175                                     rt_delay[0][i] = 0;
176                                 end
177                             end
178                             $display(" ");
179                         end
180                     end
181                     11'b00111111000 : begin                 //rotqbii rt,ra,value
                    Rotate Quadword by Bits Immediate
182                         tmp[0:6]  = imm & 7'b0000111;
183
184                         for(int b=0;b<128;b++) begin
185                             if(b+tmp[0:6] < 128 ) begin
186                                 rt_delay[0][b] = ra[b+tmp[0:6]];
187                             end
188                             else begin
189                                 rt_delay[0][b]=ra[b+tmp[0:6]-128];
190                             end
191                         end
192                     end
193                     11'b00111111100 : begin                 //rotqbyi rt, ra, imm7
                    Rotate Quadword by Bytes Immediate
194                         tmp[0:6]  = imm & 7'b0001111;
```

```
                                for(int b=0;b<=15;b++) begin
                                    if(b+tmp[0:6] < 16 ) begin
                                        for(int i = b*8;i<(b*8+8);i++) begin
                                            rt_delay[0][i] = ra[i+tmp[0:6]*8];
                                        end
                                    end
                                    else begin
                                        for(int i = b*8;i<(b*8+8);i++) begin
                                            rt_delay[0][i] = ra[i+tmp[0:6]*8-16*8];
                                        end
                                    end
                                end
                            end
                            default begin
                                rt_delay[0] = 0;
                                rt_addr_delay[0] = 0;
                                reg_write_delay[0] = 0;
                            end
                        endcase
                    end
                end
            end
        end

endmodule
```

```
module LocalStore(clk, reset, op, format, rt_addr, ra, rb, rt_st, imm, reg_write, rt_wb, rt_addr_wb, reg_write_wb, branch_taken, rt_addr_delay, reg_write_delay);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb, rt_st;  //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input           branch_taken;   //Was branch taken?

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable

    //Internal Signals
    logic [5:0][0:127]  rt_delay;           //Staging register for calculated values
    output logic [5:0][0:6] rt_addr_delay;      //Destination register for rt_wb
    output logic [5:0]      reg_write_delay;    //Will rt_wb write to RegTable

    logic [0:127] mem [0:2047];             //32KB local memory

    always_comb begin
        rt_wb = rt_delay[4];
        rt_addr_wb = rt_addr_delay[4];
        reg_write_wb = reg_write_delay[4];
    end

    always_ff @(posedge clk) begin
        if (reset == 1) begin
            rt_delay[5] <= 0;
            rt_addr_delay[5] <= 0;
            reg_write_delay[5] <= 0;
            for (int i=0; i<5; i=i+1) begin
                rt_delay[i] <= 0;
                rt_addr_delay[i] <= 0;
                reg_write_delay[i] <= 0;
            end
            for (logic [0:11] i=0; i<2048; i=i+1) begin
                //mem[i] <= 0;
                mem[i] <= {i*4, (i*4 + 1), (i*4 + 2), (i*4 + 3)};
            end
        end
        else begin
            rt_delay[5] <= rt_delay[4];
            rt_addr_delay[5] <= rt_addr_delay[4];
            reg_write_delay[5] <= reg_write_delay[4];

            for (int i=0; i<4; i=i+1) begin
                rt_delay[i+1] <= rt_delay[i];
                rt_addr_delay[i+1] <= rt_addr_delay[i];
                reg_write_delay[i+1] <= reg_write_delay[i];
            end

            if (format == 0 && op == 0) begin                   //nop : No Operation (Load)
                rt_delay[0] <= 0;
                rt_addr_delay[0] <= 0;
                reg_write_delay[0] <= 0;
            end
            else begin
                rt_addr_delay[0] <= rt_addr;
                reg_write_delay[0] <= reg_write;
                if (branch_taken) begin                     // If branch taken last cyc, cancel last instr
                    rt_delay[0] <= 0;
```

```
67                      rt_addr_delay[0] <= 0;
68                      reg_write_delay[0] <= 0;
69                  end
70                  else if (format == 0) begin
71                      case (op)
72                          11'b00111000100 : begin                 //lqx : Load Quadword
                            (x-form)
73                              rt_delay[0] <= mem[$signed(ra[0:31]) + $signed(rb[0:31])];
74                          end
75                          11'b00101000100 : begin                 //stqx : Store Quadword
                            (x-form)
76                              mem[$signed(ra[0:31]) + $signed(rb[0:31])] <= rt_st;
77                              reg_write_delay[0] <= 0;
78                          end
79                          default begin
80                              rt_delay[0] <= 0;
81                              rt_addr_delay[0] <= 0;
82                              reg_write_delay[0] <= 0;
83                          end
84                      endcase
85                  end
86                  else if (format == 4) begin
87                      case (op[3:10])
88                          8'b00110100 : begin                 //lqd : Load Quadword
                            (d-form)
89                              rt_delay[0] <= mem[$signed((ra[0:31]) + $signed(imm[8:17]))];
90                          end
91                          8'b00100100 : begin                 //stqd : Store Quadword
                            (d-form)
92                              mem[($signed(ra[0:31]) + $signed(imm[8:17]))] <= rt_st;
93                              reg_write_delay[0] <= 0;
94                          end
95                          default begin
96                              rt_delay[0] <= 0;
97                              rt_addr_delay[0] <= 0;
98                              reg_write_delay[0] <= 0;
99                          end
100                     endcase
101                 end
102                 else if (format == 5) begin
103                     case (op[2:10])
104                         9'b001100001 : begin                //lqa : Load Quadword
                            (a-form)
105                             rt_delay[0] <= mem[$signed(imm[2:17])];
106                         end
107                         9'b001000001 : begin                //stqa : Store Quadword
                            (a-form)
108                             mem[$signed(imm[2:17])] <= rt_st;
109                             reg_write_delay[0] <= 0;
110                         end
111                         default begin
112                             rt_delay[0] <= 0;
113                             rt_addr_delay[0] <= 0;
114                             reg_write_delay[0] <= 0;
115                         end
116                     endcase
117                 end
118                 else begin
119                     rt_delay[0] <= 0;
120                     rt_addr_delay[0] <= 0;
121                     reg_write_delay[0] <= 0;
122                 end
123             end
124         end
125     end
126
127 endmodule
```

```
module Branch(clk, reset, op, format, rt_addr, ra, rb, rt_st, imm, reg_write, pc_in, rt_wb, rt_addr_wb, reg_write_wb, pc_wb, branch_taken, first, branch_kill);
    input           clk, reset;

    //RF/FWD Stage
    input [0:10]    op;             //Decoded opcode, truncated based on format
    input [2:0]     format;         //Format of instr, used with op and imm
    input [0:6]     rt_addr;        //Destination register address
    input [0:127]   ra, rb, rt_st;          //Values of source registers
    input [0:17]    imm;            //Immediate value, truncated based on format
    input           reg_write;      //Will current instr write to RegTable
    input [7:0]     pc_in;          //Program counter from IF stage
    input           first;          //1 if first instr in pair; used for determining order of branch

    //WB Stage
    output logic [0:127]    rt_wb;          //Output value of Stage 3
    output logic [0:6]      rt_addr_wb;     //Destination register for rt_wb
    output logic            reg_write_wb;   //Will rt_wb write to RegTable
    output logic [7:0]      pc_wb;          //New program counter for branch
    output logic            branch_taken;   //Was branch taken?
    output logic            branch_kill;        //If branch is taken and branch instr is first in pair, kill twin instr

    //Internal Signals
    logic [0:127]   rt_delay;           //Staging register for calculated values
    logic [0:6]     rt_addr_delay;      //Destination register for rt_wb
    logic           reg_write_delay;    //Will rt_wb write to RegTable
    logic [7:0]     pc_delay;           //Staging register for PC
    logic           branch_delay;       //Was branch taken?

    always_ff @(posedge clk) begin
        if (reset == 1) begin
            rt_wb <= 0;
            rt_addr_wb <= 0;
            reg_write_wb <= 0;
            pc_wb <= 0;
            branch_taken <= 0;
        end
        else begin
            rt_wb <= rt_delay;
            rt_addr_wb <= rt_addr_delay;
            reg_write_wb <= reg_write_delay;
            pc_wb <= pc_delay;
            branch_taken <= branch_delay;
        end
    end

    always_comb begin
        if (format == 0 && op == 0) begin                   //nop : No Operation (Load)
            rt_delay = 0;
            rt_addr_delay = 0;
            reg_write_delay = 0;
            pc_delay = 0;
            branch_delay = 0;
        end
        else begin
            rt_addr_delay = rt_addr;
            reg_write_delay = reg_write;
            if (branch_taken) begin
                rt_delay = 0;
                rt_addr_delay = 0;
                reg_write_delay = 0;
                pc_delay = 0;
                branch_delay = 0;
            end
            else if (format == 0) begin
                case (op)
                    11'b00110101000 : begin             //bi : Branch Indirect
```

```
67                          pc_delay = ra[0:31];
68                          reg_write_delay = 0;
69                          branch_delay = 1;
70                      end
71                      default begin
72                          rt_delay = 0;
73                          rt_addr_delay = 0;
74                          reg_write_delay = 0;
75                          pc_delay = 0;
76                          branch_delay = 0;
77                      end
78                  endcase
79              end
80              else if (format == 5) begin
81                  case (op[2:10])
82                      9'b001100110 : begin                //brsl: Branch Relative and Set
                        Link
83                          rt_delay[0:31] = pc_in + 1;
84                          rt_delay[32:127] = 0;
85                          pc_delay = imm[2:17] + pc_in;
86                          branch_delay = 1;
87                      end
88                      9'b001100100 : begin                //br: Branch Relative
89                          pc_delay = imm[2:17] + pc_in;
90                          branch_delay = 1;
91                      end
92                      9'b001100000 : begin                //bra: Branch Absolute
93                          pc_delay = imm[2:17];
94                          branch_delay = 1;
95                      end
96                      9'b001000010 : begin                //brnz: Branch If Not Zero Word
97                          if (first)
98                              pc_delay = pc_in - 2 + $signed(imm[2:17]);
99                          else
100                             pc_delay = pc_in - 1 + $signed(imm[2:17]);
101                         branch_delay = (rt_st[0:31] == 32'h0000) ? 0 : 1;
102                     end
103                     9'b001000000 : begin                //brz: Branch If Zero Word
104                         if (first)
105                             pc_delay = pc_in - 2 + $signed(imm[2:17]);
106                         else
107                             pc_delay = pc_in - 1 + $signed(imm[2:17]);
108                         branch_delay = (rt_st[0:31] == 32'h0000) ? 1 : 0;
109                     end
110                     default begin
111                         rt_delay = 0;
112                         rt_addr_delay = 0;
113                         reg_write_delay = 0;
114                         pc_delay = 0;
115                         branch_delay = 0;
116                     end
117                 endcase
118             end
119             else begin
120                 rt_delay = 0;
121                 rt_addr_delay = 0;
122                 reg_write_delay = 0;
123                 pc_delay = 0;
124                 branch_delay = 0;
125             end
126         end
127
128         if (branch_delay == 1 && first == 1)
129             branch_kill = 1;
130         else
131             branch_kill = 0;
132     end
133 endmodule
```

Assembler Code

```
from distutils.log import debug
import sys
class Assembler:
    def __init__(self,ins_list_name,input_file,output_file,debug) -> None:
        self.ins_list_name =  ins_list_name
        self.ins_opcode_mapping = dict()
        self.opcode_ins_mapping = dict()
        self.output_file = output_file
        self.input_file =  input_file
        self.debug_level =debug
        self.load_mapping()


    def load_mapping(self):

        with open(self.ins_list_name,'r') as ins_lst:
            for line in ins_lst.readlines():
                # print(line)
                ins,opcode =  line.split("\t")
                memonic = ins.split(" ")[0]
                opcode = opcode.strip()
                self.ins_opcode_mapping[memonic]=opcode
                if "rt, ra, rb, rc" in ins:
                    self.opcode_ins_mapping[opcode]=[1,ins]
                elif "rt, ra, rb" in ins:
                    self.opcode_ins_mapping[opcode]=[0,ins]
                elif "rt, ra, imm7" in ins:
                    self.opcode_ins_mapping[opcode]=[2,ins]
                elif "rt, ra, imm8" in ins:
                    self.opcode_ins_mapping[opcode]=[3,ins]
                elif "rt, ra, imm10" in ins or "rt, imm10(ra)" in ins :
                    self.opcode_ins_mapping[opcode]=[4,ins]
                elif "rt, imm16" in ins or "imm16" in ins :
                    self.opcode_ins_mapping[opcode]=[5,ins]
                elif "rt, imm18" in ins:
                    self.opcode_ins_mapping[opcode]=[6,ins]
                else:
                    if "nop" in ins:
                        print("nop")
                        self.opcode_ins_mapping[opcode]=[7,ins]
                    elif "stop" in ins:
                        self.opcode_ins_mapping[opcode]=[7,ins]

                # #print(ins,opcode)
        # #print(self.opcode_ins_mapping)
        # #print(self.ins_opcode_mapping)

    def parse_line(self,line):
        ins = line.strip()
        ins =ins.split("//")
        ins = ins[0].strip().split(" ")
        return ins

    def parse_input(self):

        with open(self.input_file,'r') as ins_lst, open(self.output_file,'w') as object:
            for line in ins_lst.readlines():
                ins = self.parse_line(line)
                memonic = ins[0]
                if "stop" in memonic:
                    break
                #print(memonic)
                opcode = self.ins_opcode_mapping[memonic]
                #print(opcode)
                format = self.opcode_ins_mapping[opcode]
                # print("format ",format)
                binary = self.compute(format,opcode,ins)
```

```
68                  object.write(binary+"\n")
69
70      def parse_input_1(self):
71          with open(self.input_file,'r') as ins_lst, open(self.output_file,'w') as
            object,open("debug.out",'w') as d_object:
72              for line in ins_lst.readlines():
73
74                  ins = self.parse_line(line)
75                  memonic = ins[0]
76                  if "stop" in memonic:
77                      break
78                  #print(memonic)
79                  opcode = self.ins_opcode_mapping[memonic]
80                  print(opcode)
81                  format = self.opcode_ins_mapping[opcode]
82                  # print("format ",format)
83                  binary = self.compute(format,opcode,ins)
84                  object.write(binary+"\n")
85                  if self.debug_level==2:
86                      o_hex = '0x{0:0{1}X}'.format(int(binary,2),8)
87                      d_object.write(str(binary)+"\t"+o_hex+"\t"+line)
88                  else:
89                      d_object.write(str(binary)+"\t"+line)
90
91
92      def compute(self,format,opcode,ins):
93          ins_binary = "".zfill(32)
94          print(format,ins_binary,len(ins_binary))
95          print("ins {}".format(ins))
96          if format[0] == 0:
97              # opcode rt ra rb
98              # op[0-10]rb[11-17]ra[18-24]rt[25-31]
99              rt = bin(int(ins[1])).replace("0b","")
100             ra = bin(int(ins[2])).replace("0b","")
101             rb = bin(int(ins[3])).replace("0b","")
102             #print(rt,ra,rb)
103             rt = self.fill(rt,7)
104             ra = self.fill(ra,7)
105             rb = self.fill(rb,7)
106             #print(rt,ra,rb)
107             ins_binary = opcode+rb+ra+rt
108
109         elif format[0] == 1:
110             # opcode rt,ra,rb,rc
111             # op[0-3]rt[4-10]rb[11-17]ra[18-24]rc[25-31]
112             rt = bin(int(ins[1])).replace("0b","")
113             ra = bin(int(ins[2])).replace("0b","")
114             rb = bin(int(ins[3])).replace("0b","")
115             rc = bin(int(ins[4])).replace("0b","")
116             #print(rt,ra,rb)
117             rt = self.fill(rt,7)
118             ra = self.fill(ra,7)
119             rb = self.fill(rb,7)
120             rc = self.fill(rc,7)
121             # print(rt,ra,rb,rc)
122             ins_binary = opcode+rt+rb+ra+rc
123         elif format[0] == 2:
124             # opcode rt,ra,imm7
125             # op[0-10]imm7[11-17]ra[18-24]rt[25-31]
126             mask = 0b1111111
127             rt = bin(int(ins[1])).replace("0b","")
128             ra = bin(int(ins[2])).replace("0b","")
129             imm7 = bin(int(ins[3]) & mask).replace("0b","")
130
131             rt = self.fill(rt,7)
132             ra = self.fill(ra,7)
133             imm7 = self.fill(imm7,7)
```

```
            ins_binary = opcode+imm7+ra+rt

        elif format[0] == 3:
            # opcode rt,ra,imm8
            # op[0-9]imm8[10-17]ra[18-24]rt[25-31]
            mask = 0b11111111
            rt = bin(int(ins[1])).replace("0b","")
            ra = bin(int(ins[2])).replace("0b","")
            imm8 = bin(int(ins[3]) & mask).replace("0b","")
            rt = self.fill(rt,7)
            ra = self.fill(ra,7)
            imm8 = self.fill(imm8,8)
            ins_binary = opcode+imm8+ra+rt


        elif format[0] == 4:
            # opcode rt,ra,value
            # op[0-7]imm10[8-17]ra[18-24]rt[25-31]
            rt = bin(int(ins[1])).replace("0b","")
            rt = self.fill(rt,7)
            mask = 0b1111111111

            if '(' in format[1]:
                # opcode rt, symbol(ra)
                imm10 = ins[2].split('(')[0]
                ra = ins[2].split('(')[1][:-1]
            else:
                # opcode rt,ra,value
                imm10 = ins[3]
                ra = ins[2]

            # print(imm10,ra)
            imm10 = bin(int(imm10)  & mask).replace("0b","")
            imm10 = self.fill(imm10,10)

            ra = bin(int(ra)).replace("0b","")
            ra = self.fill(ra,7)
            ins_binary = opcode+imm10+ra+rt
        elif format[0] == 5:
            print(ins)
            mask = 0b1111111111111111
            if len(ins)==3:
                rt = bin(int(ins[1])).replace("0b","")
                imm16 = bin(int(ins[2]) & mask).replace("0b","")
            else:
                rt="0"
                imm16 = bin(int(ins[1]) & mask ).replace("0b","")

            print(rt,imm16)
            rt = self.fill(rt,7)
            imm16 = self.fill(imm16,16)
            print(rt,imm16)
            ins_binary = opcode+imm16+rt
        elif format[0] == 6:
            # opcode rt,value
            # op[0-8]imm16[9-24]rt[25-31]
            mask = 0b11111111111111111
            rt = bin(int(ins[1]) & mask).replace("0b","")
            rt = self.fill(rt,7)

            imm18 = bin(int(ins[2]) ).replace("0b","")
            imm18 = self.fill(imm18 ,18)
            ins_binary = opcode+imm18+rt
        else:
            ins_binary = opcode + self.fill("",32-len(opcode))


```

```
201
202             #print(ins_binary,len(ins_binary))
203             return ins_binary
204         def fill(self,seq,width):
205             neg = False
206             if len(seq) > 0 and seq[0]=='-':
207                 neg = True
208             seq =  seq.replace("-","")
209             wide_seq = seq.zfill(width)
210
211             if neg:
212                 wide_seq=str(1)+wide_seq[1:]
213             return wide_seq
214
215
216
217     if len(sys.argv) ==1:
218         print("Please mention the asm file: python assembler <input>.asm -o <out>")
219         sys.exit(0)
220     input_file_name = sys.argv[1]
221     output_file_name = "out"
222     debug = 0
223     print(sys.argv)
224
225     print(sys.argv)
226     for i in range(1,len(sys.argv)):
227         print("test ",sys.argv[i])
228         if sys.argv[i] =='-o':
229             print("afdasd")
230             output_file_name = sys.argv[i+1]
231             i=i+1
232
233         if sys.argv[i] =='-d':
234             debug=int(sys.argv[i+1])
235             i=i+1
236
237     print(" fda ",input_file_name)
238     print("dfd ",output_file_name)
239     asm = Assembler(ins_list_name="instructions.lst", input_file=input_file_name,
240                     output_file=output_file_name,debug=debug)
241
242     print(debug)
243     if debug==0:
244         asm.parse_input()
245     elif debug>=1:
246         print("debug ")
247         asm.parse_input_1()
```

```
mpy rt, ra, rb  01111000100
mpyu rt, ra, rb 01111001100
mpyh rt, ra, rb 01111000101
fa rt, ra, rb   01011000100
fs rt, ra, rb   01011000101
fm rt, ra, rb   01011000110
fceq rt, ra, rb 01111000010
fcgt rt, ra, rb 01011000010
shlh rt, ra, rb 00001011111
shl rt, ra, rb  00001011011
roth rt, ra, rb 00001011100
rot rt, ra, rb  00001011000
rothm rt, ra, rb    00001011101
rotm rt, ra, rb 00001011001
rotmah rt, ra, rb   00001011110
rotma rt, ra, rb    00001011010
cntb rt, ra 01010110100
avgb rt, ra, rb 00011010011
absdb rt, ra, rb    00001010011
sumb rt, ra, rb 01001010011
ah rt, ra, rb   00011001000
a rt, ra, rb    00011000000
sfh rt, ra, rb  00001001000
sf rt, ra, rb   00001000000
and rt, ra, rb  00011000001
or rt, ra, rb   00001000001
xor rt, ra, rb  01001000001
nand rt, ra, rb 00011001001
ceqb rt, ra, rb 01111010000
ceqh rt, ra, rb 01111001000
ceq rt, ra, rb  01111000000
cgtb rt, ra, rb 01001010000
cgth rt, ra, rb 01001001000
cgt rt, ra, rb  01001000000
clgtb rt, ra, rb    01011010000
clgth rt, ra, rb    01011001000
clgt rt, ra, rb 01011000000
nop 01000000001
mpya rt, ra, rb, rc 1100
fma rt, ra, rb, rc  1110
fms rt, ra, rb, rc  1111
shli rt, ra, imm7   00001111011
rothi rt, ra, imm7  00001111100
roti rt, ra, imm7   00001111000
rotmahi rt, ra, imm7    00001111110
rotmai rt, ra, imm7 00001111010
cflts rt, ra, imm8  0111011000
cfltu rt, ra, imm8  0111011001
mpyi rt, ra, imm10  01110100
mpyui rt, ra, imm10 01110101
ahi rt, ra, imm10   00011101
ai rt, ra, imm10    00011100
sfhi rt, ra, imm10  00001101
sfi rt, ra, imm10   00001100
andbi rt, ra, imm10 00010110
andhi rt, ra, imm10 00010101
andi rt, ra, imm10  00010100
orbi rt, ra, imm10  00000110
orhi rt, ra, imm10  00000101
ori rt, ra, imm10   00000100
xorbi rt, ra, imm10 01000110
xorhi rt, ra, imm10 01000101
xori rt, ra, imm10  01000100
ceqbi rt, ra, imm10 01111110
ceqhi rt, ra, imm10 01111101
ceqi rt, ra, imm10  01111100
cgtbi rt, ra, imm10 01001110
cgthi rt, ra, imm10 01001101
cgti rt, ra, imm10  01001100
```

```
clgtbi rt, ra, imm10    01011110
clgthi rt, ra, imm10    01011101
clgti rt, ra, imm10 01011100
ilh rt, imm16   010000011
ilhu rt, imm16  010000010
il rt, imm16    010000001
iohl rt, imm16  011000001
ila rt, imm18   0100001
shlqbi rt, ra, rb   00111011011
shlqby rt, ra, rb   00111011111
rotqbi rt, ra, rb   00111011000
rotqby rt, ra, rb   00111011100
gbb rt, ra  00110110010
gbh rt, ra  00110110001
gb rt, ra   00110110000
lqx rt, ra, rb  00111000100
stqx rt, ra, rb 00101000100
bi ra   00110101000
lnop    00000000001
shlqbii rt, ra, imm7    00111111011
shlqbyi rt, ra, imm7    00111111111
rotqbii rt, ra, imm7    00111111000
rotqbyi rt, ra, imm7    00111111100
lqd rt, imm10(ra)   00110100
stqd rt, imm10(ra)  00100100
lqa rt, imm16   001100001
stqa rt, imm16  001000001
br imm16    001100100
bra imm16   001100000
brsl rt, imm16  001100110
brnz rt, imm16  001000010
brz rt, imm16   001000000
stop    00000000000
```

Sample Programs

```
ai 0 0 12       //Even pipe structural hazard
ilh 4 -15
ai 1 0 1
ilh 5 -14
ahi 2 2 14
ilh 6 -13
ahi 3 2 1
stqd 1 0(1)     //Odd pipe structural hazard, plus memory ref
stqd 1 1(1)
stqd 1 2(1)
stqd 1 3(1)
lqa 4 13
lqa 5 14
lqa 6 15
lqa 7 16
ila 25 3        //Set r25 to 3 and decrement
ila 26 0
ai 25 25 -1     //Keep decrementing until r25==0
ai 26 26 1
brnz 25 -2      //Branch hazard resolution, odd destination addr; Also RAW hazard, brnz is stalled 1 cyc until dec r25 finishes
avgb 2 0 1      //Dual issue example
rotqbi 12 4 5
mpy 5 3 4
rotqbi 13 4 0
mpy 8 6 7
rotqbi 14 4 1
mpy 11 9 10
rotqbi 15 4 3
mpy 17 25 26
rotqbi 15 4 2   //Forwarding without stalling
stop            //Stop instr ignores all following instr and ends program
mpy 8 6 7
rotqbi 14 4 1
mpy 11 9 10
rotqbi 15 4 3
mpy 14 22 23
```

```
ila 0 0              // store 0 in reg0
ila 1 4              // shift=4;
ilh 10 -1            // used to create mask FFFFFFFF
ilh 30 4             // use to shift by 12byte
ilh 31 8             // use to shift by 8 byte
ilh 32 12            // use to shift by 8 bytes
shlqby 10 10 32      // generate FFFFFFFF ins first word
ila 11 4             // i = 4// START OUTER LOOP
ai 11 11 -1          // i--
ila 12 4             // j = 4 START INNER LOOP
ila 42 0             // load_index=0
ila 43 16            // rotate=16
ila 20 0             // clearing reg 20
ai 12 12 -1          // j--;
ila 15 0             // clearing reg 15
lqd 5 4(42)          // load row(2)= 4+0,+41,4+2,4+3
and 15 5 10          // extract the first word of the quadword
rotqby 15 15 43      // rotate the word to align in the slot
xor 20 15 20         // reg 15 will have elemnt for column 0 matrix2
shlqby 5 5 1         // shift row by 4
ai 43 43 -4          // rotate-=4
stqd 5 4(42)         // store the shifted reg back to memory
ai 42 42 1           // load_index+=1
brnz 12 -10          // if reg11==0 break else goto ai 12 12 -1
stqd 20 10(11)       // store 20 in 13 to 10
brnz 11 -17          // if reg11==0 break else goto ai 11 11 -1
ila 51 4             // i = 4// START OUTER LOOP
ai 51 51 -1          // i--
ila 80 0             // reset the quad word
lqd 21 0(51)         // load maxtrix 1 rows
ila 12 4             // j = 4 START INNER LOOP
ila 43 16            // rotate=16s
ai 12 12 -1          // j--;
lqd 22 10(12)        // load transposed matrix2 rows
mpya 23 22 21 23     // multiply slot wise both rows and store qword in 23
ila 81 0             // reset the cell
ila 45 4             // k=0
ai 45 45 -1          // k--
and 70 23 10         // using mask to extract the left most word
shlqby 23 23 30      // shift left by 4 bytes
a 81 81 70           // computing sum of the product cell wise
brnz 45 -4           // jump to ai 45 45 -1
and 81 81 10         // clearing any other bits
rotqby 81 81 43      // rotate reg81 to place word in correct slot of quad
ai 43 43 -4          // rotate-=4
or 80 80 81          // build the quadword using or with rotated word
brnz 12 -14          // jump to ai 12 12 -1
stqd 80 20(51)       // store the quad word row for the matrix3
brnz 51 -21          //Jump to ai 51 51 -1
stop
```