Cell SPU Project Final Report

Dongyun Lee (112794190)

Contents of Table

1 Project Description

This project is to build a dual-issue, 11-stage pipelined processor based on the Cell SPU. Supports in-order execution with static branch prediction (predict-not-taken). Implements structural, data, and control hazard handling with full forwarding. The design runs a subset of SPU instructions including Fixed Point, Single Precision, Byte, Permute, Local Store, and Branch. Everything is cycle-accurate and written in Verilog. Instruction memory and register file are preloaded before execution. Full source codes are provided in Appendix.

2 SPU ISA

The SPU uses six 32-bit fixed-length instruction formats to encode all operations. Implemented instructions follows the original instructions of Sony Cell SPU ISA. 88 instructions are implemented in the project including all 8 units with basic sub-set of general-purpose RSIC architecture operations. The instruction table is shown in appendix A.

There is total 6 formats of instructions. Below table shows the breakdown:

Format	Field Names	Bit Ranges	Description
RR	OP, RB, RA, RT	OP: 0–10, RB: 11–17, RA: 18–24, RT: 25–31	2 source registers (RA, RB), 1 destination register (RT)
RRR	OP, RT, RB, RA, RC	OP: 0–3, RT: 4–10, RB: 11–17, RA: 18–24, RC: 25–31	3 source registers (RA, RB, RC), 1 destination (RT)
RI7	OP, I7, RA, RT	OP: 0–10, I7: 11–17, RA: 18–24, RT: 25–31	7-bit immediate with 2 registers
RI10	OP, I10, RA, RT	OP: 0–7, I10: 8–17, RA: 18–24, RT: 25–31	10-bit immediate with 2 registers
RI16	OP, I16, RT	OP: 0–8, I16: 9–24, RT: 25–31	16-bit immediate with 1 register
RI18	OP, I18, RT	OP: 0–6, I18: 7–24, RT: 25–31	18-bit immediate with 1 register

3 Assembly Parser

This program reads SPU-lite assembly code, parses each instruction, and generates the corresponding 32-bit machine code. C++ is used to write this program. It supports all instruction formats used in the SPU-lite processor and handles label resolution and basic syntax validation. Input file is input_assembly.txt and output file is output_binary.txt.

The program uses a combination of modern C++ features such as unordered_map, vector, string, and bit-level manipulation with bitset. The lookup table (unordered_map<string, InstructionInfo> instructionMap) that maps instruction mnemonics (like addx, and, rotqby, etc.) to their corresponding instruction format (FormatType) and binary opcode. Each instruction format (e.g., RR, RRR, RI10, RI16, etc.) is defined with its bit layout, and there are dedicated functions like assembleRR, assembleRRR, assembleRI10, and so on, to convert operands into a final 32-bit binary string.

The assembler first reads the entire input file, filters out labels and comments, and builds a labelMap that tracks instruction addresses for label resolution (crucial for PC-relative branches). Then, each remaining line is passed to the processInstruction() function, which looks up the mnemonic, extracts operands, chooses the right assembler function based on the format, and encodes the instruction.

To handle branching and label resolution, the assembler calculates relative or absolute offsets using the instruction's position (pc) and a lookup in the labelMap. This allows it to encode targets correctly without hardcoding addresses manually.

3.1 Features

- **Supports Six Instruction Formats** :
  The parser correctly encodes instructions in RR, RRR, RI7, RI10, RI16, and RI18 formats based on the SPU specification.
- **Instruction Lookup Table** :
  Mnemonics are mapped to their format and opcode using a predefined `unordered_map`. The opcodes are stored as integers and translated to binary as needed.
- **Operand Parsing** :
  Supports register (`$X`), decimal, and hexadecimal (`hFF`) operands. For memory-style instructions like `lqd`, D-form syntax is parsed to extract immediate and base register.
- **Negative Value** :
  Converts negative decimal into two's complement binary for immediate field.
- **Label Support** :
  Labels ending with a colon (e.g., `loop:`) are tracked and used to resolve branch targets, particularly for RI16 instructions.

- **Comment Handling :**
  Any line starting with // is treated as a comment and skipped during parsing. This allows readable and documented assembly code without affecting output.
- **Binary Output :**
  The assembler writes one 32-bit binary string per instruction to output_binary.txt. This file can be directly loaded into the instruction memory of the SPU-lite testbench.
- **Error Feedback :**
  The program detects and reports unknown instructions, wrong operand counts, malformed labels, and incorrect field lengths.

```
31  LOOP: // can also have comments
32  and $1, $2, $3
33  rotqby $4, $2, $3
34  and $5, $2, $3
35  rotqby $6, $2, $3
36  nop
37  bra LOOP
```

```
1  00011000001000001100000100000001
2  00111011100000001100000100000100
3  00011000001000001100000100000101
4  00111011100000001100000100000110
5  01000000001000000000000000000000
6  00110000000000000000000100000000
```

4 Module Description

4.1 Top Level

The top_level module connects all components of the SPU-lite dual-issue processor. It integrates the instruction fetch unit, decode and hazard unit, register file and forwarding unit, and both execution pipeline units. It controls the flow of instructions and data through the 11-stage pipeline.

4.1.2 Key Features

```
module top_level(
  input clk,
  input rst,
  input load_en,
  input [0:31] instruction_in,
  input [0:9] instr_load_addr,
  input preload_en,
  input [0:9] preload_addr,
  input [0:127] preload_values,
  input preload_LS_en,
  input [0:6] preload_LS_addr,
  input [0:127] preload_LS_data
);
```

Inputs

- clk, rst: Clock and asynchronous reset signals.

- instruction_in, instr_load_addr, load_en: Used to preload instructions into instruction memory.
- preload_en, preload_addr, preload_values: Used to preload data into the register file.
- preload_LS_en, preload_LS_addr, preload_LS_data: Used to preload data into the local store (LS), typically for memory-related instructions.

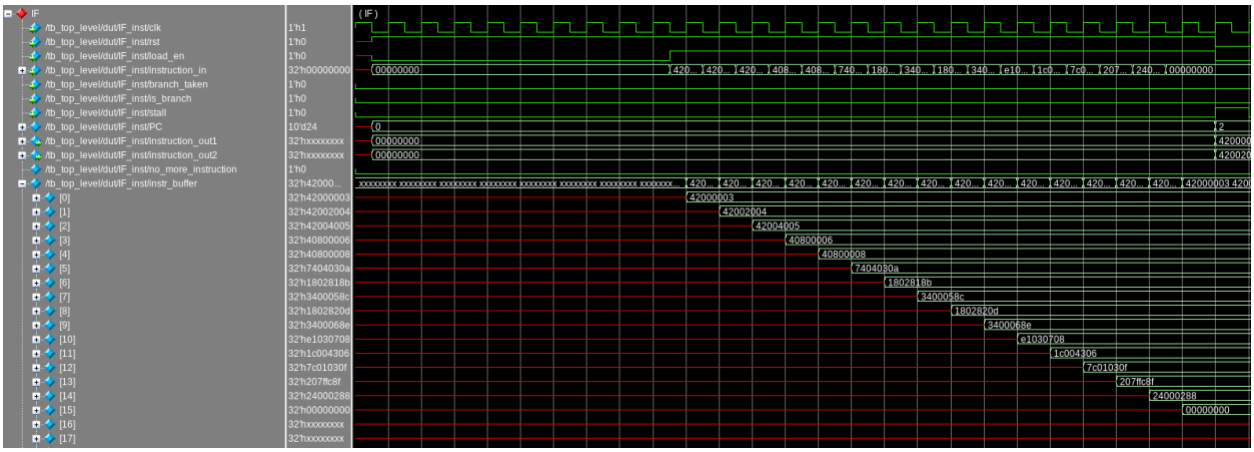
Structure and Module Connections

1. **IF_wrapper (Instruction Fetch Unit)**
   Handles program counter (PC) updates, branch target resolution, and dual-instruction fetch. Outputs instruction_out1 and instruction_out2 each cycle. Supports stall and flush signals for control hazards.
2. **ID_HU_wrapper (Instruction Decode + Hazard Unit)**
   Parses instruction formats and fields, determines operand addresses, and detects hazards. It also decides whether to flush or stall based on branch and data hazards. Outputs decoded control/data signals to the next stage.
3. **RF_FU_wrapper (Register File + Forwarding Unit)**
   Reads register operands, applies forwarding if necessary, and resolves RAW hazards. It also updates the register file using results from later pipeline stages and handles all immediate formats. Preload support is built in.
4. **Even_Pipe and Odd_Pipe (Execution Pipelines)**
   These modules represent two independent 11-stage pipelines. They execute the instructions issued to them, perform ALU operations, memory access, and write back results to the register file. The odd pipe also handles branch instructions and PC redirection.

4.2 IF stage

The IF_wrapper module is responsible for fetching two instructions per cycle from instruction memory. It supports dual-issue fetching, handles stalls, and redirects the program counter (PC) on taken branches.

The output binary .txt file loads into instruction memory. The waveform below shows that the instructions are loaded.



4.2.1 Key Features

```
module IF_wrapper(
    input  clk,
    input  rst,
    input  load_en,
    input  [0:31] instruction_in,
    input [0:8] instr_load_addr,
    input [0:8] PC_br_target,
    input  branch_taken,
    input is_branch,
    input  stall,

    output reg [0:8] PC_current_out,
    output reg [0:31] instruction_out1,
    output reg [0:31] instruction_out2,
    output reg find_nop
);
```

Inputs

- clk, rst: Clock and asynchronous reset signals.
- instruction_in, instr_load_addr, load_en: Used to preload instructions into internal instruction memory.
- PC_br_target: Branch target address.
- branch_taken, is_branch: Control signals indicating when to redirect the PC.
- stall: Freezes the PC and reissues the previous instructions.

Outputs

- PC_current_out: Current program counter.
- instruction_out1, instruction_out2: Two fetched 32-bit instructions.
- find_nop: Signals the decode stage to insert a NOP when only one valid instruction is fetched (due to misalignment).

A 512-word (2 KB) instruction buffer stores 32-bit instructions. This buffer is byte-addressable by default, but the design simplifies it to word-addressing (i.e., one instruction per address index). This means the PC increments by 2 (instead of 8) each cycle to fetch two 32-bit instructions at once.

```
// Instruction buffer memory (2KB = 512 x 32-bit instructions)
reg [0:31] instr_buffer [0:LINE_LENGTH-1];
```

During a branch, if the target address is misaligned (i.e., odd), only one instruction is fetched, and the second slot is filled with zero. In this case, the find_nop signal is asserted so that the decode stage inserts a NOP. If the target address is aligned, both instructions are fetched normally. During a stall, the current PC is held, and the same instructions are re-issued. Otherwise, the PC increments by 2 to fetch the next pair of instructions. During reset, if load_en is high, the specified instruction is written into the given address of the instruction buffer.

```
            if (is_branch && branch_taken) begin // branch taken
                if (PC_br_target[8] == 1'b1) begin // misaligned target
                    instruction_out1 <= 32'b0;
                    instruction_out2 <= instr_buffer[PC_br_target - 2];
                    PC <= PC_br_target - 1;
                    find_nop <= 1'b1; // telling ID stage to find which nop is needed
                end
                else begin // aligned target
                    instruction_out1 <= instr_buffer[PC_br_target - 2];
                    instruction_out2 <= instr_buffer[PC_br_target - 1];
                    PC <= PC_br_target;
                    find_nop <= 1'b0;
                end
            end
```

Originally, the SPU's instruction memory is byte-addressable and the PC increments by 8 each cycle to fetch two 32-bit instructions (since each instruction is 4 bytes). In this implementation, the addressing is simplified to word-based (one index per 32-bit instruction), so the PC increments by 2 instead. This avoids unnecessary byte-level address handling while preserving correctness for aligned instruction access.

4.3 ID/HU stage

The ID_HU_wrapper module serves as the decode and hazard detection stage in a dual-issue SPU-like processor pipeline. Instruction_Decode module is a combinational module that decodes two fetched instructions per cycle, determines their type (even or odd), and assigns them to the appropriate pipeline. At the same time, Hazard_Unit checks for structural, data, and control hazards using a dedicated hazard unit. When hazards are detected, it stalls or flushes instructions as needed and inserts NOPs accordingly. It also handles branch misalignment correction using the find_nop signal. Decoded outputs, such as destination registers, immediate values, unit IDs, and register addresses, are forwarded to the RF stage along with control signals.

4.3.1 Key Features

Inputs

- clk, rst: Clock and asynchronous reset signals.

Pipeline Control

- is_branch, branch_taken: From Odd pipe, signals whether a control hazard occurred and notifying that branch has taken.
- flush_instr2_even: Tells this stage to flush the second instruction (even) due to control hazards.
- find_nop: Raised when IF stage fetches only one valid instruction due to PC misalignment.

Instruction Inputs

- PC_pass_in: PC associated with the fetched instructions (passed forward).
- instruction_in1, instruction_in2: Dual instructions fetched from IF.

Pipeline Feedback

From RF and EX stages:

- Past destination register info: RF_reg_dst_even, RF_reg_dst_odd
- Write enable: RF_reg_wr_even, RF_reg_wr_odd
- Latency: RF_latency_even, RF_latency_odd
- Packed pipeline info (packed_2stage_* to packed_6stage_*): Encodes instruction metadata useful for hazard detection.

Outputs

Decoded Control & Operands

- For each pipe (even and odd):
    - instr_id_*, reg_dst_*, unit_id_*, latency_*, reg_wr_*
    - Immediate values: imme7_*, imme10_*, imme16_*, imme18_*
    - Source register addresses: ra_addr_*, rb_addr_*, rc_addr_*

Pipeline Control

- instr1_branch: Signals Odd_Pipe that the first instruction is a branch.
- stall: Asserted when a stall is required (from Hazard_Unit).
- flush: Tells RF to flush both instructions.
- flush_4stage: Additional flush signal used for corner case.
- PC_pass_out: Passes the associated PC forward to RF.

Core Control Protocols

Stall Logic

- Generated when hazard conditions exist (via Hazard_Unit).
- If temp_stall or temp_dependent_stall is high, the module inserts NOPs (instr_ID_nop and instr_ID_lnop) as needed to protect data consistency.

Dependency Protocols

Two internal registers manage complex pipeline rescheduling:

- instr_dependent_protocol: Used for resolving instruction-type mismatches or ordering dependencies.
- data_dependent_protocol: Used when one instruction must be delayed due to unresolved data hazards.

Each uses a 2-bit flag:

- 2'b01: even instruction should proceed
- 2'b10: odd instruction should proceed
- 2'b00: no active protocol

These enable *half-slot issue recovery*, letting one instruction proceed while the other slot is padded with a NOP.

Branch Misalignment (NOP Insertion)

If find_nop is asserted (from IF stage), this means the PC was misaligned. So only one valid instruction is available. Based on the instruction's type (instr2_type), the module assigns either:

- instr_ID_nop to the even pipe
- or instr_ID_lnop to the odd pipe

Special Control Signal

- instr1_branch: If the first decoded instruction has a unit_id of 3'b111 (indicating a branch), this flag is raised to inform the Odd_Pipe for branch tracking.
- flush_4stage and flush: Forwarded from flush_instr2_even and Hazard_Unit, respectively. These control flushing deeper in the pipeline.

4.3.2 Instruction Decode Unit

The Instruction_Decode module takes in two 32-bit SPU instructions (instruction_in1, instruction_in2) and decodes each to extract fields such as opcode, register operands, immediate values, destination register, unit ID, execution latency, and control signals like reg_wr. It converts opcode into fixed 7 bit instruction ID which are declared in opcode_package.vh. It supports multiple instruction formats (RR, RRR, RI7, RI10, RI16, RI18) based on the SPU ISA. The module distinguishes between even and odd pipeline slots using instr1_type and instr2_type flags, where 1 denotes even and 0 denotes odd. The decoding is performed in two parallel always blocks, one for each instruction input. Each instruction is mapped to its corresponding instr_id and other metadata by matching its opcode to known constants defined in opcode_package.vh. Below is a part of Instruction decode function.

```
// ------------- R18 -----------------
else if (temp_opcode1[0:6] == `op_ila) begin
  instr_id_1 = `instr_ID_ila;
  reg_dst_1 = instruction_in1[25:31];
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  imme18_1 = instruction_in1[7:24];
  instr1_type = 1'b1;
end
```

For instructions that uses data from RT, I have utilized RC port for reading RT data. This can be done since no instructions uses all 3 source register and RT data. Below is how I treated RC port as reading RT data.

```
  imme7_1 = instruction_in1[8:17];
  reg_dst_1 = instruction_in1[25:31];
  ra_addr_1 = instruction_in1[18:24];
```

```
rc_addr_1 = instruction_in1[25:31]; // for rt data
```

4.3.3 Hazard Unit

The Hazard_Unit is a combinational module responsible for detecting hazards in a dual-issue SPU-like pipelined processor. This module is integrated inside ID_HU_wrapper. It takes information about two decoded instructions (even and odd slots) and checks for control hazards, data hazards, and issue dependencies, producing signals to stall or flush the pipeline, or to allow only one instruction to proceed.

4.3.3.1 Key Features

Control Hazard Detection (Flush)

- If a branch is mispredicted (e.g., predict-not-taken but actually taken), and branch_taken && is_branch is true, then flush is asserted.
- This logic assumes static prediction but can be extended later to 2-bit predictors.

Data Hazard Detection (Stall)

- Checks whether current instruction operands (ra, rb, rc) are waiting on results from previous pipeline stages.
- For even pipe, it checks ra_addr_even, rb_addr_even, and rc_addr_even (only if instr_needs_rc_even is asserted, indicating that the instruction is of a type that uses the rc operand).

```
•   else if ((ra_addr_even == packed_2stage_even[131:137] || rb_addr_even == packed_2stage_even[131:137]
    || (instr_needs_rc_even && rc_addr_even == packed_2stage_even[131:137])) && packed_2stage_even[142])
    begin
•     if(packed_2stage_even[138:141] > 4'd3) begin
•       stall = 1'b1;
•     end
•   end
```

- For odd pipe, similar logic is applied using ra_addr_odd, rb_addr_odd, and rc_addr_odd, and instr_needs_rc_odd is used to guard rc-based checks:

```
•   else if ((ra_addr_odd == packed_2stage_odd[131:137] || rb_addr_odd == packed_2stage_odd[131:137] ||
    (instr_needs_rc_odd && rc_addr_odd == packed_2stage_even[131:137])) && packed_2stage_odd[142])
    begin
•     if(packed_2stage_odd[138:141] > 4'd3) begin
•       stall = 1'b1;
•     end
```

```
    end
```

- If any operand has a match and the producer hasn't completed yet, stall is asserted.
- Latency thresholds increase per stage: e.g., 2 for RF, 3 for 2nd stage, …, 7 for 6th stage.

Instruction Pair Dependency Detection (dependent_stall)

E. g.)

```
always @(*) begin
  dependent_stall = 1'b0;
  if (instr_dependent_protocol==2'b00
      && data_dependent_protocol==2'b00
      && both_valid
      && (instr1_type==instr2_type
          || reg_dst_even==reg_dst_odd)) begin
    dependent_stall = 1'b1;
  end
end
```

- Separate stall signal, it is asserted only when both instructions are same unit (using same pipe) or two instructions have same register destination which is WAW hazard.
- If instr_dependent_protocol and data_dependent_protocol are both 2'b00 (i.e., no override), and both instructions are valid, then check dependent stall.

Instruction Format-Aware Logic

- RRR-format instructions (like fma, fms, fnms, mpya, selb) use three source operands (ra, rb, rc). The unit uses is_RRR_instr_even to gate rc_addr_even checks accordingly.

4.4 RF/FU Stage

The RF_FU_wrapper module serves combining register file access and operand forwarding logic to supply source operands (ra, rb, rc) for both even and odd instruction. It interfaces with a 6-port register file to read operand values and supports dual write-back paths. To resolve read-after-write hazards, it integrates a forwarding unit that dynamically selects operand values from later pipeline stages when a data dependency is detected, with priority given to the most recent and same-lane results. The module also handles instruction metadata propagation, pipeline flushing on mispredicted branches, and testbench-driven register preloading for verification.

4.4.1 Register File

The Reg_file module implements a 128-entry, 128-bit-wide register file with six asynchronous read ports and two synchronous write ports. It supports dual write-back paths. Each read output uses bypass logic to immediately reflect newly written values in the same cycle, effectively resolving read-after-write (RAW) hazards internally. The module also includes a preload_en feature for verification, allowing one register to be initialized with a given value at reset. On a positive clock edge or reset, the register file either clears all entries or preloads a specific address, ensuring deterministic simulation and testbench behavior.

Register file memory is declared as

```
// 128 registers, 128 bit width
reg [0:127] reg_file [0:127];
```

Read operation example is

```
assign reg_read_data_1 =
   (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_1) ? reg_write_data_1 :
   (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_1) ? reg_write_data_2 :
   reg_file[reg_read_addr_1];
```

Write operation always block is

```
// write operations (synchronous)
always @(posedge clk or posedge rst) begin
 if (rst) begin
  if (preload_en) begin
   // Load register every cycle
   reg_file[preload_addr] <= preload_values;
  end else begin
   for(i=0;i<128;i=i+1) begin
    reg_file[i] <= 128'b0;
   end
  end
 end
 else begin
  if (reg_write_en_1) begin
   reg_file[reg_write_addr_1] <= reg_write_data_1;
```

```
    end
    if (reg_write_en_2) begin
      reg_file[reg_write_addr_2] <= reg_write_data_2;
    end
  end
end
```

4.4.2 Forwarding Unit

The Forwarding_Unit module detects and enables data forwarding in a dual-issue pipelined processor to resolve read-after-write (RAW) hazards without stalling. It receives source register addresses for both even and odd pipeline instructions (ra, rb, and rc), and monitors the destination register, write-enable, and latency of all pipeline stages (from 2 to 7) through packed_*stage_* signals. Each packed signal contains the destination register address, register write-enable bit, and result latency. Based on these, the module computes whether data should be forwarded from any stage in the pipeline to any of the three source operands for both even and odd pipes. It also supports cross-pipe forwarding (e.g., forwarding from an odd-stage result to an even-pipe instruction) by asserting corresponding *_fw_en_* signals, ensuring correct and efficient data propagation across the pipeline. Example code shown below:

```
  assign ra_fw_en_2stage_even = (reg_ra_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);
  assign rb_fw_en_2stage_even = (reg_rb_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);
  assign rc_fw_en_2stage_even = (reg_rc_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);
```

4.5 Even Pipe

The Even_Pipe module implements the datapath and pipeline register handling for the even pipeline of a dual-issue processor. It takes as input the instruction information (instr_id, reg_dst, unit_id, latency, and reg_wr), operand values from the register file (ra_data, rb_data, rc_data), and various immediate values. Based on the unit_id, it routes the operands and immediates to one of four functional units: FX1_ALU, FX2_ALU, SP_ALU, or BYTE_ALU, each of which produces a 128-bit result. The selected result is packed along with metadata (unit ID, destination register, latency, and write-enable) into a 143-bit vector called packed_1stage.

This packed result propagates through seven pipeline registers (packed_2stage through packed_7stage) on every clock cycle, with flush and flush_4stage allowing selective flushing of the pipeline. The module outputs these packed stages to enable forwarding, and in the write-back stage, extracts the destination register, result data, and write-enable signal from packed_7stage for register file writing. This modular structure facilitates easy pipeline control, forwarding, and execution of multiple SIMD units in parallel.

4.6 Odd Pipe

The Odd_Pipe module implements the pipeline datapath for the odd-issue slot in a dual-issue superscalar processor. It executes instructions from three units: PERM for permute operations, LS for load/store memory access via LocalStore, and BRANCH for control-flow decisions. Like the Even_Pipe, it collects operand and immediate inputs, selects a functional unit based on the unit_id, computes a 128-bit result, and packs it into a 143-bit vector packed_1stage that includes metadata like unit ID, destination register, latency, and write-enable signal. This packed result propagates through a 7-stage pipeline (packed_2stage to packed_7stage), providing outputs for forwarding and write-back.

Additionally, Odd_Pipe handles memory writes via the LS_ALU and LocalStore, and computes branch decisions using BRANCH_ALU, generating a new_PC, asserting branch_taken, and signaling whether the instruction is a branch. If the branch is taken and it's in the second issue slot, it raises flush_instr2_even to cancel the simultaneously issued instruction in the even pipe. The module also supports preloading data into the local store for verification. The design maintains precise control over execution and resolution of control and data hazards, crucial for supporting a dual-issue superscalar architecture.

4.6.1 Local Store

The Local Store module is inplemented inside Odd pipe since only odd unit instructions deal with Local Store memory. The memory size is 32KB byte-addressing. The memory is declared as reg [0:7] LS_mem [0:32767], meaning it stores 32,768 individual bytes (8 bits each), and each address index corresponds to one byte. This structure allows the module to support precise memory access at the byte level while externally handling 128-bit (16-byte) blocks.

When performing a read, the module takes a 15-bit byte-aligned address LS_addr and reads 16 consecutive bytes starting from that address. It assembles these 16 bytes into a single 128-bit word by extracting each 8-bit chunk and concatenating them into LS_data_out. This is done using a loop:

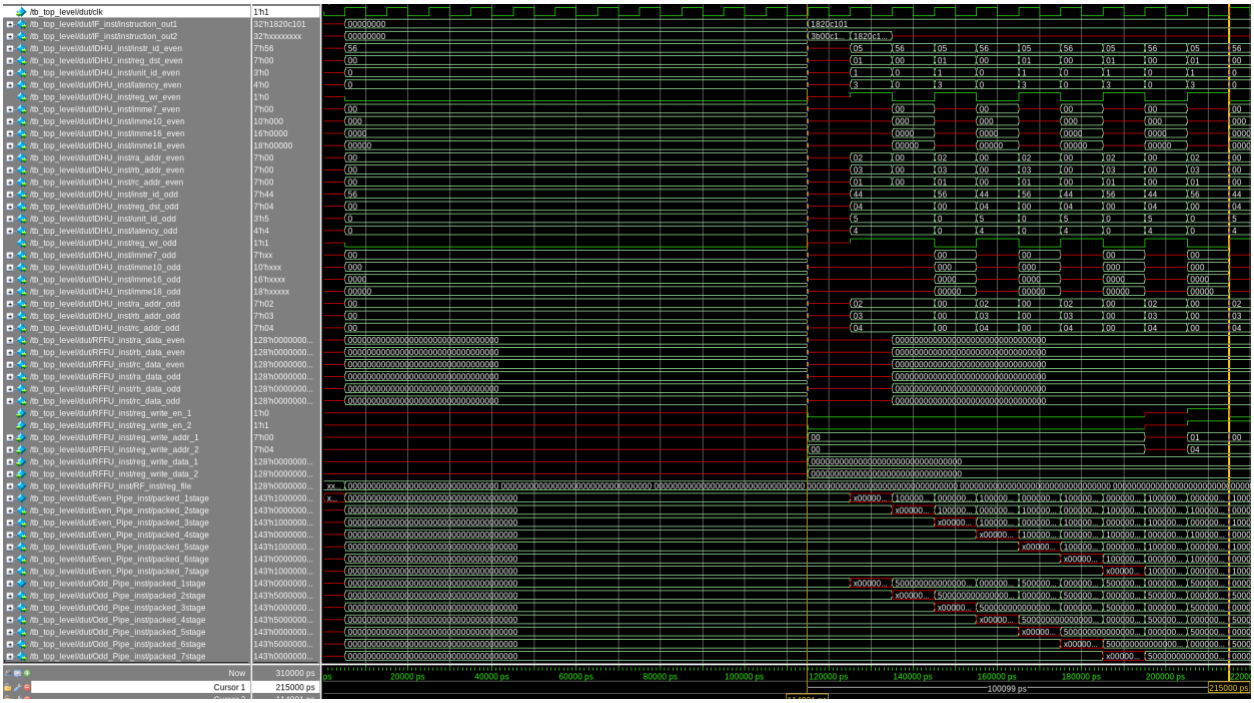
```
// Asynchronous read
```

```
always @(LS_addr) begin
    for (i = 0; i < 16; i = i + 1) begin
        LS_data_out[i*8 +: 8] = LS_mem[LS_addr + i];
    end
end
```

This expression i*8 +: 8 specifies an 8-bit slice of LS_data_out, effectively packing bytes back into a word in little-endian order.

For writes, a similar loop writes each 8-bit segment of the 128-bit LS_data_in value into consecutive byte addresses of LS_mem, starting at LS_addr. This again ensures that the memory is updated in a byte-by-byte manner, preserving correct alignment and supporting finer-grained access patterns.

5 11 Stage Pipeline



This shows the full pipeline shows 11 stages working correctly. The first cursor is when Instruction fetched and second cursour is when result was written on register file.

6 Structural hazard

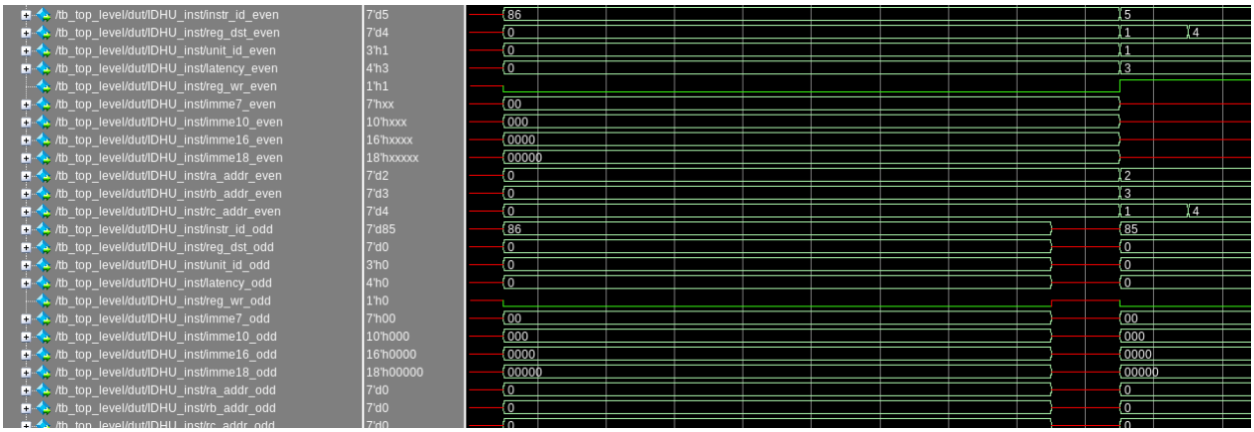
6.1 Both Instructions Same Unit

When both instructions are Even

Input code :

1 and $1, $2, $3

2 and $4, $2, $3



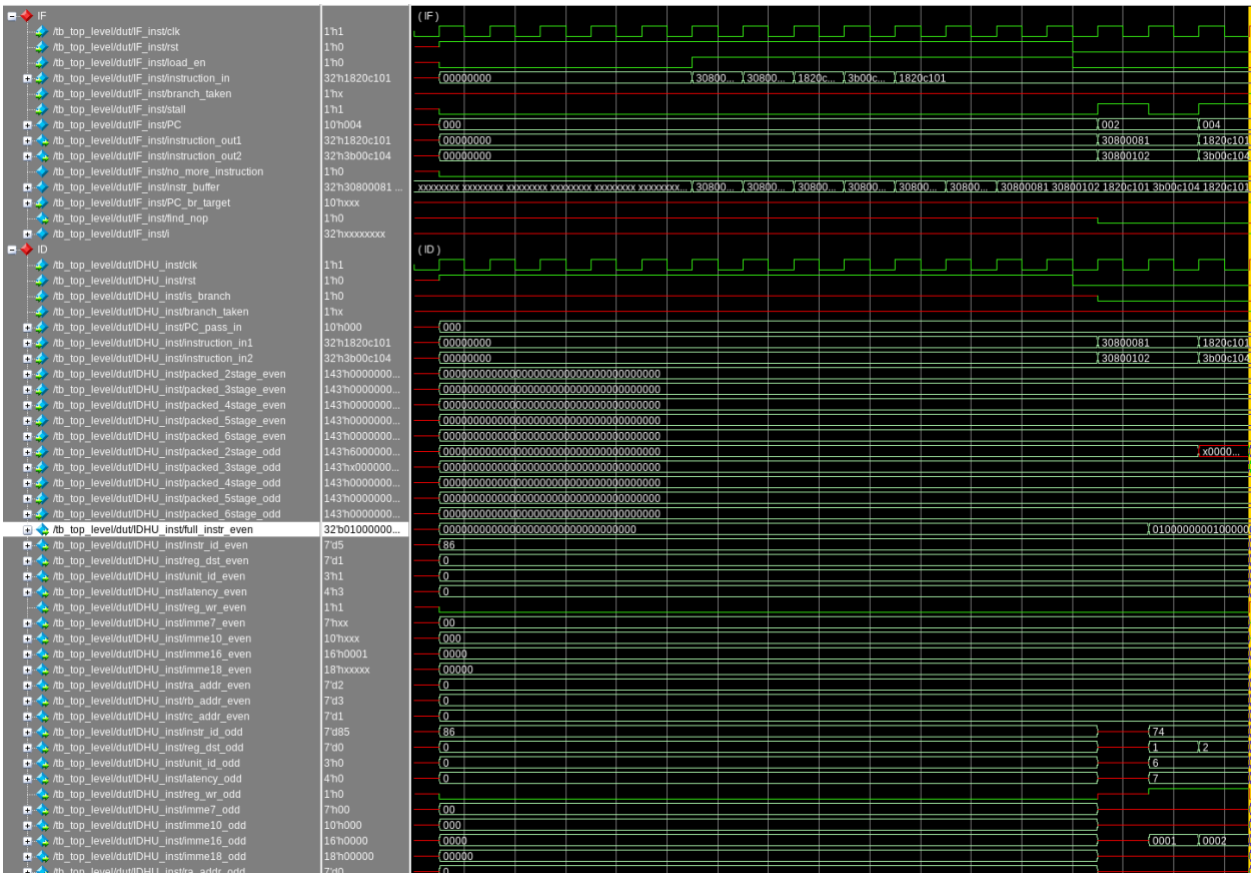
We can distinguish the first instruction and second instruction by looking at the register destination. It is shown that the first instruction goes to next stage with lnop for odd pipe and the next clock cycle, the second instruction goes to next stage with lnop for odd pipe again.

When both instructions are Odd

Input code :

1 lqa $1, h1

2 lqa $2, h2



Shows that when both instructions are odd (lqa) it stalls and send nop for even pipe and send odd instruction one by one

7 Data Hazard

7.1 No Stall

Data hazard resolving by forwarding with no stall :

Input code :

1 lqa $1, h1

2 lqa $2, h2

3 (nops, wait until load is done)

4 or $3, $1, $2

5 lnop

6 nop

7 lnop

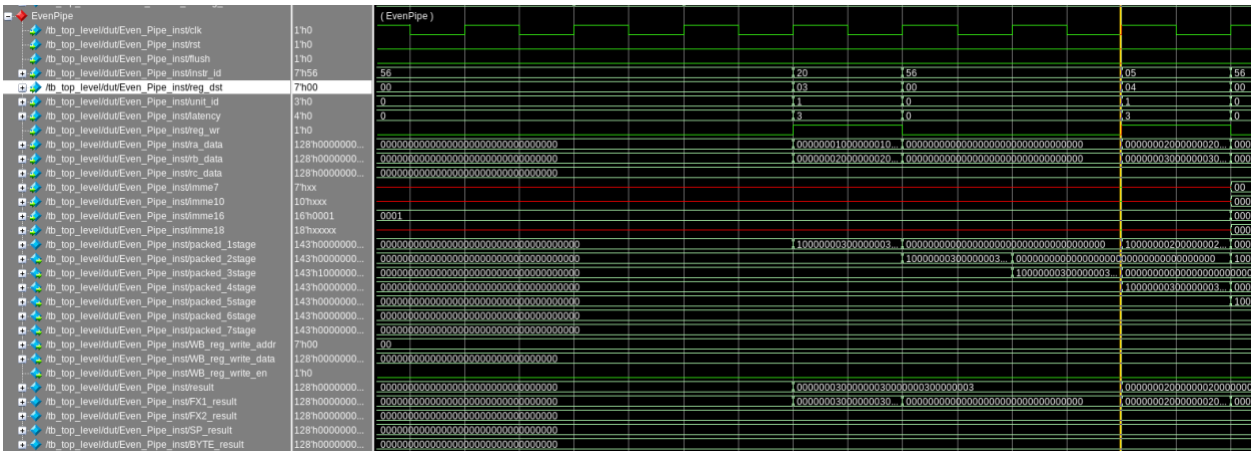
8 nop

9 lnop

10 and $4, $2, $3

11 lnop

$3 should be forwarded with value 0h00000003 each word.



Where cursor starts is showing the Even pipe when “and” instructions comes in with the forwarded value 0h00000003 each word. This waveform shows that forwarding works correctly on Even pipe.

7.2 Stall

However, I have purposely filled the hazard gaps with nop. This time, we will see when “and” instruction is able to go after 3 clock cycles of stalls as well as forwarded value.

Input code:

1 lqa $1, h1

2 lqa $2, h2

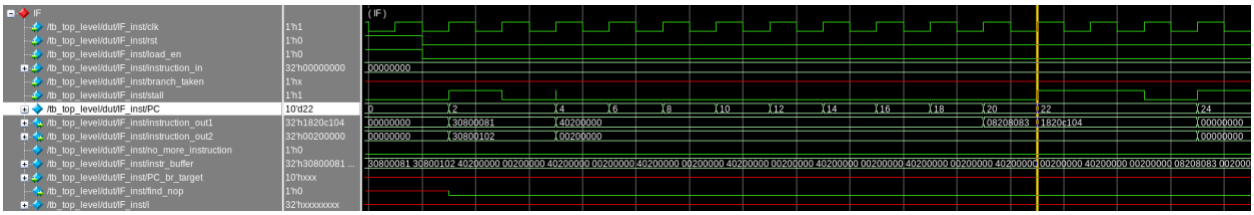
3 // (nops, wait until load is done)

4 or $3, $1, $2 // no nops between

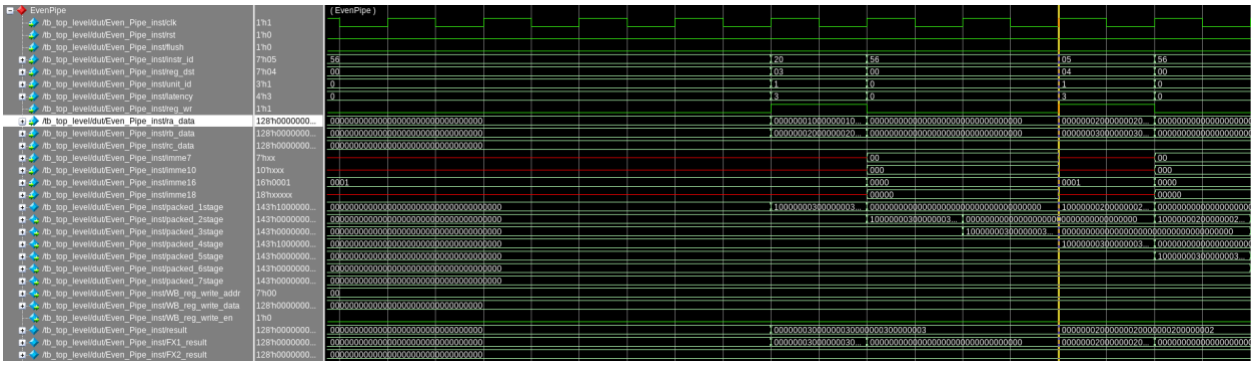
5 lnop

6 and $4, $2, $3

7 lnop



The waveforms shows IF stage and at the cursor, stall asserted due to data hazard until “or” instruction’s result is able to forward which is 3 cycles later.



This waveform shows the Even Pipe and at the cursor, RB data is shown as 0h00000003 which is forwarded from “or” instruction’s result. This shows a correct forwarding with stall.

Input code :

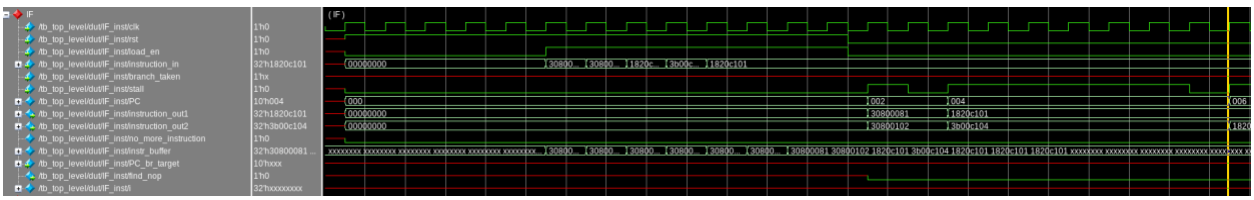
1 lqa $1, h1

2 lqa $2, h2

3 and $1, $2, $3

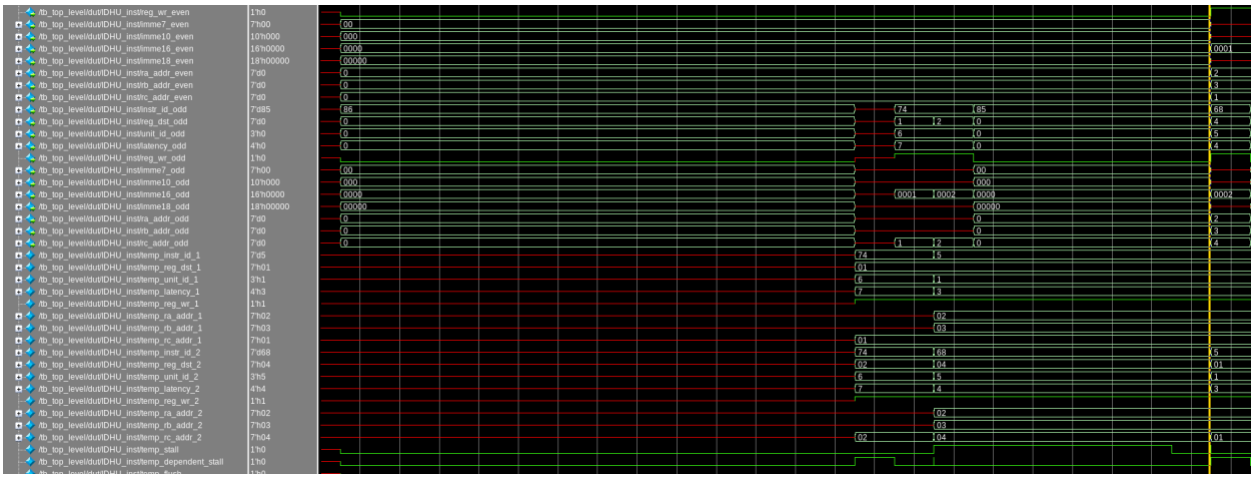
4 rotqbi $4, $2, $3

“and” instruction and “rotqbi” instruction must stall until load instruction is done loading $2 value.



IF stage shows that it is stalled and freeze PC and proceed when load_instruction is ready.

(stall signal deasserts when it is ready)



This is from ID stage, it shows that ID stage correctly outputs “and” and “rotqbi” instructions at the cursor.

8 Control Hazard

8.1 Branch

Input code :

1 LOOP:

2 and $1, $2, $3

3 rotqby $4, $2, $3

4 and $5, $2, $3

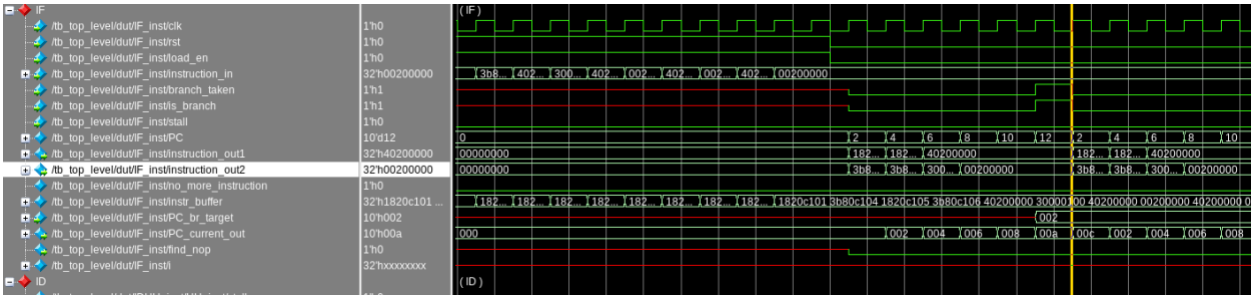
5 rotqby $6, $2, $3

6 nop

7 bra LOOP

(nops)

Branch target address is PC <= 0 and should go back to the first “and” instruction.



This waveform shows that at the cursor, branch taken to the first instruction.

8.2 Branch with flush

We will see if the flush is working correctly.

Input code :

1 LOOP:

2 and $1, $2, $3

3 rotqby $4, $2, $3

4 and $5, $2, $3

5 rotqby $6, $2, $3

6 and $7, $2, $3

7 bra LOOP

8 xor $10, $11, $12

9 shlqbi $13, $14, $15

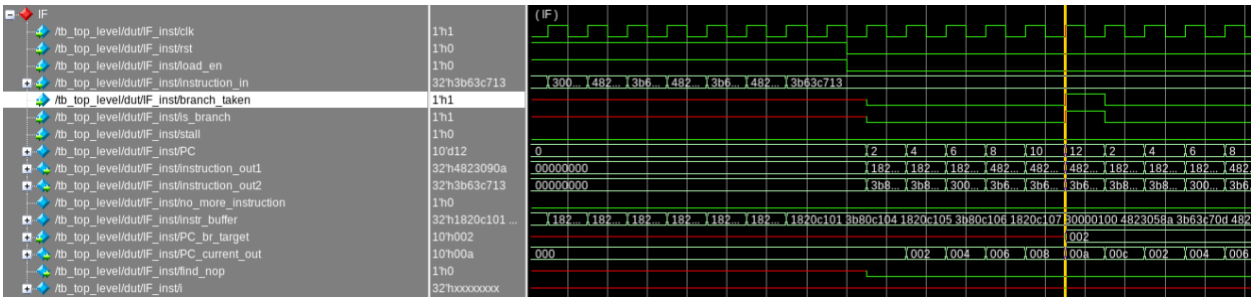
10 xor $16, $11, $12

11 shlqbi $17, $14, $15

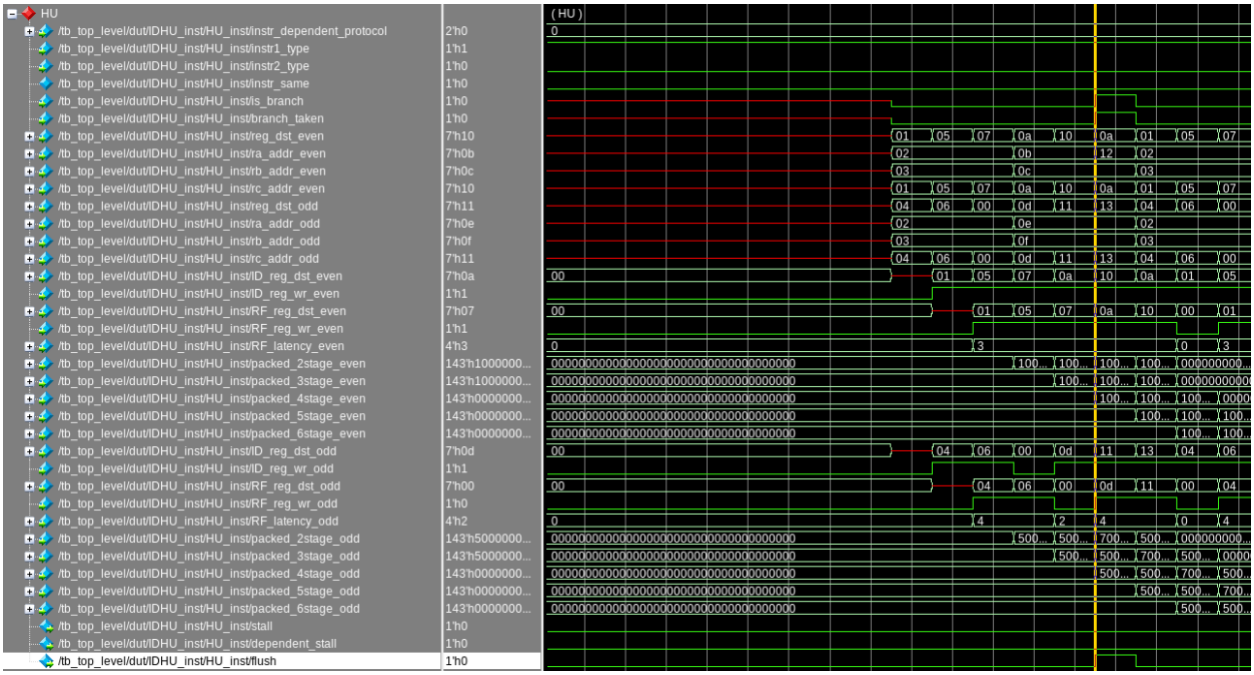
12 xor $10, $18, $12

13 shlqbi $19, $14, $15

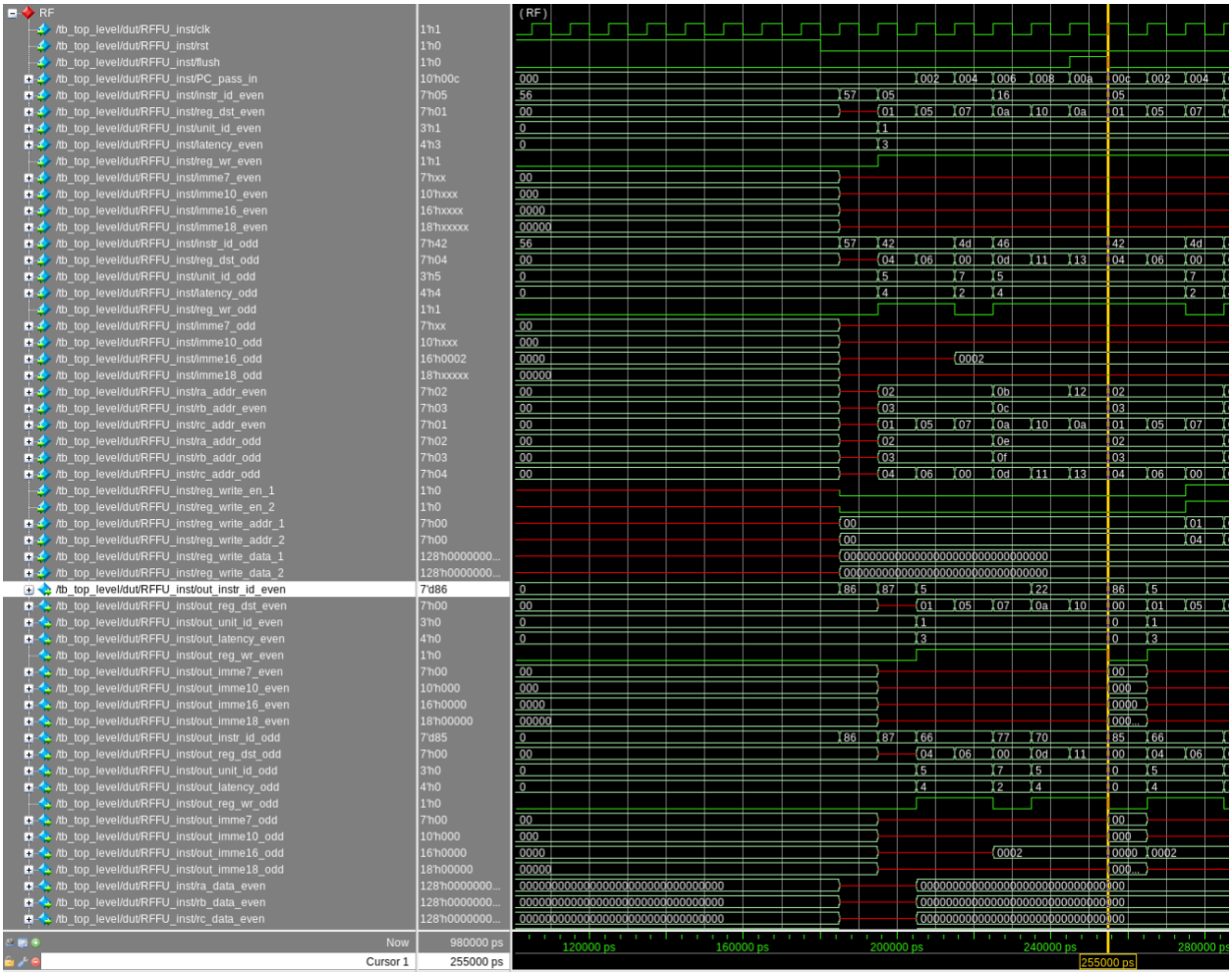
Expected result is that the last 6 instructions (“xor” and “shlqbi”) instructions must be flushed when brank is taken.



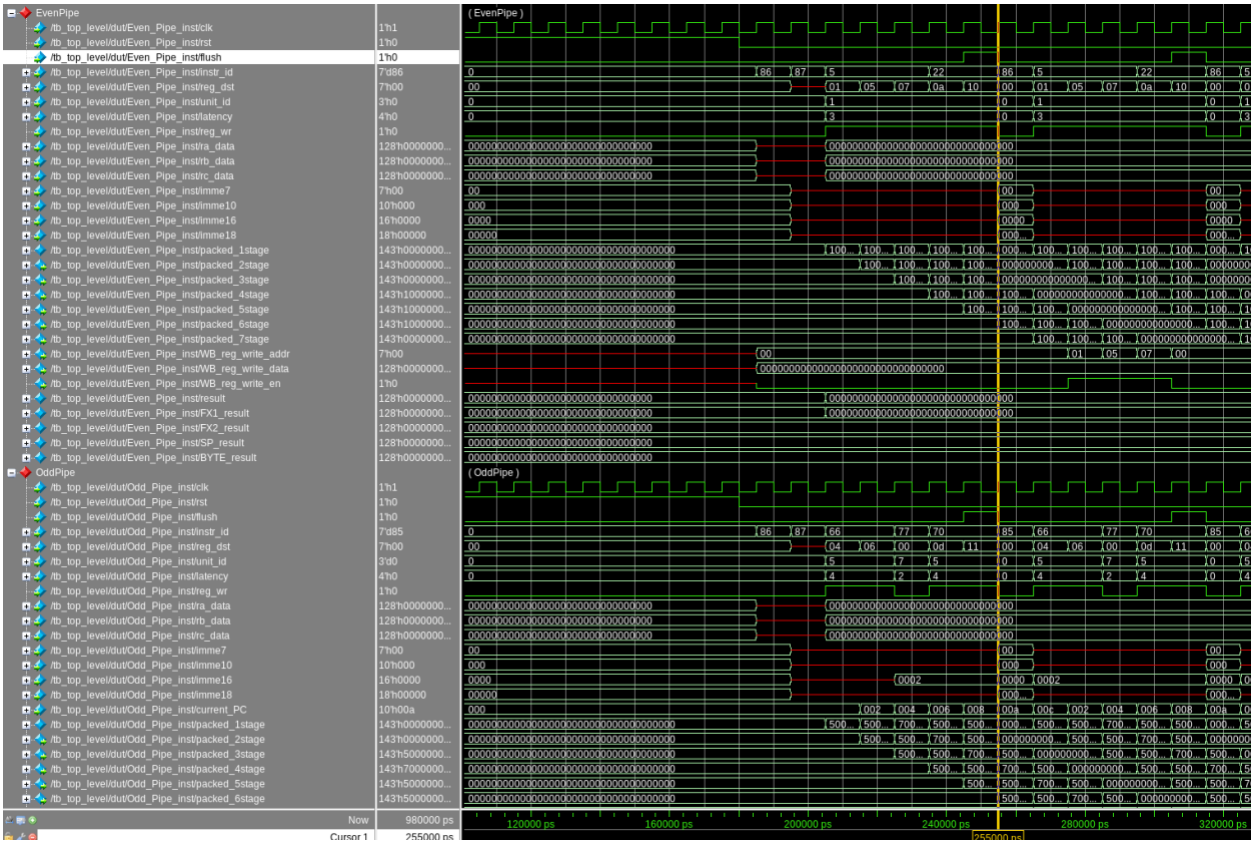
This waveform is IF stage and shows that IF stage prefetches PC = 8, 10 , and 12. When branch is taken, PC goes back to 0.



This waveform shows Hazard Unit. When is_branch signal and branch_taken signal is both 1, flush signal asserts high '1'. This flush signal will be sent to RF stage, Even stage, and Odd stage to make prefetched instructions to be flushed.



This waveform is RF stage, when flush signal is asserted, it outputs nop for Even pipe and lnop for Odd pipe. The cursor shows that RF stage correctly outputs nops for both pipes.



This waveform shows both Even pipe at the top and Odd pipe at the bottom. When flush signal asserts, it is shown that stage 2, and stage 3 get flushed with all 0s for both pipe. The cursor shows that stage 1 which was flushed from RF stage with all 0s and both second stage and third stage are flushed.

The simulation successfully reflects that flush when branck prediction is wrong works correctly.

8.3 Branch target at odd PC

We will test when branch target address is odd. This scenario brings up a complexity that if the target PC instruction is even unit, then it needs to send lnop with it and if the instruction is odd unit, then it needs to send nop with it because Cell SPU always has to send two even and odd instructions at every clock cycle.

Input code :

1 bra LOOP

2 and $1, $2, $3

3 rotqby $4, $2, $3

4 and $5, $2, $3

5 rotqby $6, $2, $3

6 and $7, $2, $3

7 rotqby $8, $2, $3

8 and $9, $2, $3

9 xor $10, $18, $12

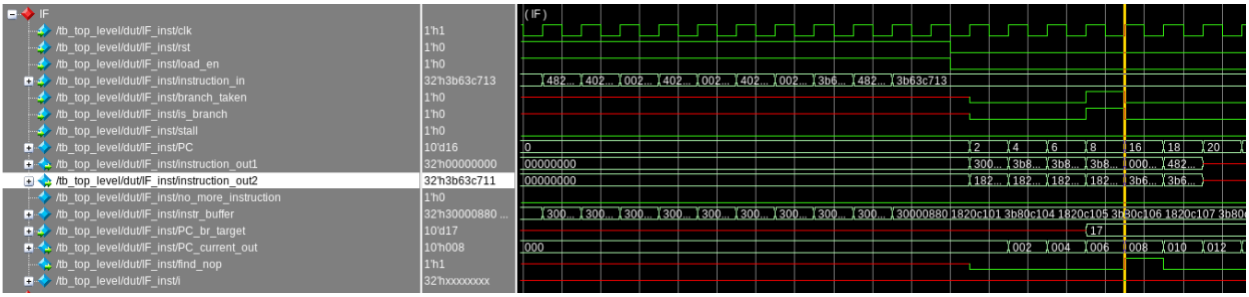
10 LOOP:

11 shlqbi $17, $14, $15

12 xor $11, $18, $12

13 shlqbi $19, $14, $15

Expected result should be that IF stage sends “shlqbi” with nop when branch is taken as well as correcting the PC to even number.



The waveform shows IF stage. At the cursor, it shows that when branch is taken, it sends the odd instruction with nop for even pipe. This shows that branch target at odd PC scenario is taken correctly with proper control.

8.4 Branch Instruction Before Even Instruction

This case is when branch instruction comes before even instruction that it will be executed together with the branch instruction. The correct result of this case is that the even instruction must be flushed which menas two instruction that will be go to pipe should be branch instruction and nop instruction.

Input code :

1 LOOP:

2 and $1, $2, $3

3 rotqby $4, $2, $3

4 and $5, $2, $3

5 rotqby $6, $2, $3

6 and $7, $2, $3

7 rotqby $8, $2, $3

8 bra LOOP

9 and $9, $2, $3

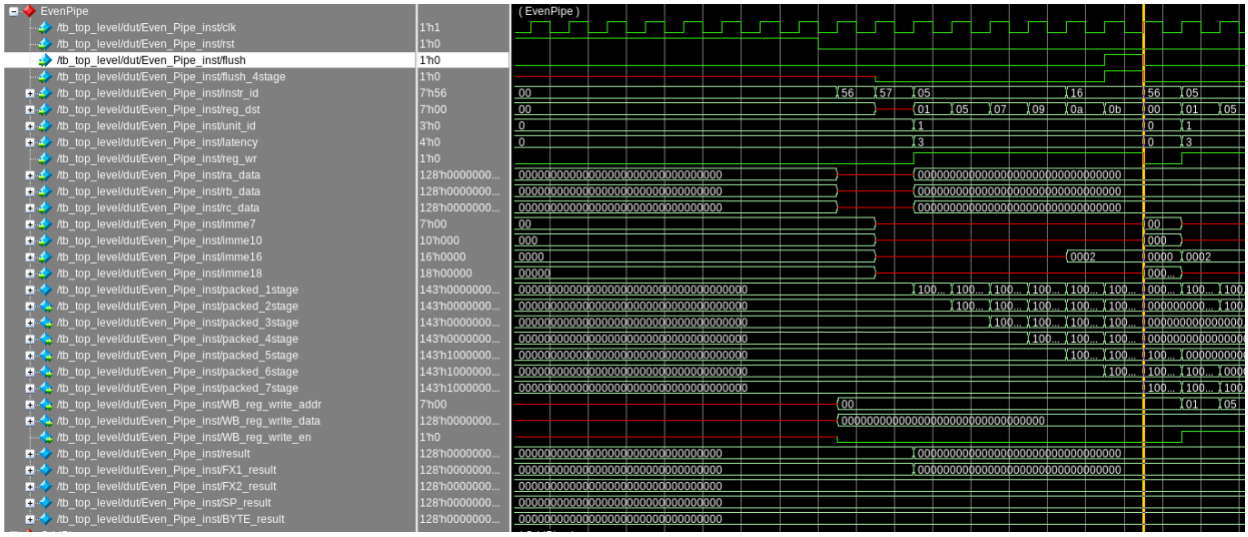
10 xor $10, $18, $12

11 shlqbi $17, $14, $15

12 xor $11, $18, $12

13 shlqbi $19, $14, $15

bra LOOP comes first than “and” instruction. The correct flush is that 4th stage of Evenpipe should also be flushed. Extra signal “flush_4stage” from Hazard Unit (ID stage) will be asserted when branch instruction comes first than even unit instruction and branch is taken.



This waveform show Even pipe. At the cursor, it shows that from first stage to fourth stage get flushed. Fourth stage is the “add” instruction in blod from input code. This shows that branch instruction before even instruction case is handled correctly.

9 4x4 SP FP Matrix Multiplication

This program performs 4x4 single precision floating point matrix multiplication. The mathematical representation of this program is

$$\begin{bmatrix} 0.0 & 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 & 7.0 \\ 8.0 & 9.0 & 10.0 & 11.0 \\ 12.0 & 13.0 & 14.0 & 15.0 \end{bmatrix} \times \begin{bmatrix} 16.0 & 17.0 & 18.0 & 19.0 \\ 20.0 & 21.0 & 22.0 & 23.0 \\ 24.0 & 25.0 & 26.0 & 27.0 \\ 28.0 & 29.0 & 30.0 & 31.0 \end{bmatrix} = \begin{bmatrix} 152.0 & 158.0 & 164.0 & 170.0 \\ 504.0 & 526.0 & 548.0 & 570.0 \\ 856.0 & 894.0 & 932.0 & 970.0 \\ 1208.0 & 1262.0 & 1316.0 & 1370.0 \end{bmatrix}$$

We will call the first matrix, matrix A and second matrix, matrix B. For example, from the result matrix element [1,1] is calculated as

$$C_{[1,1]} = A_{[1,1]} * B_{[1,1]} + A_{[1,2]} * B_{[2,1]} + A_{[1,3]} * B_{[3,1]} + A_{[1,4]} * B_{[4,1]}$$

This can be represented in pseudo-code as :

```
il $C[1,1] , 0 // initialize with zero
Fma $C[1,1], $A[1,1], $B[1,1], $C[1,1]
Fma $C[1,1], $A[1,2], $B[2,1], $C[1,1]
Fma $C[1,1], $A[1,3], $B[3,1], $C[1,1]
Fma $C[1,1], $A[1,4], $B[4,1], $C[1,1]
```

For full matrix multiplication can be expressed in pseudo-code as :

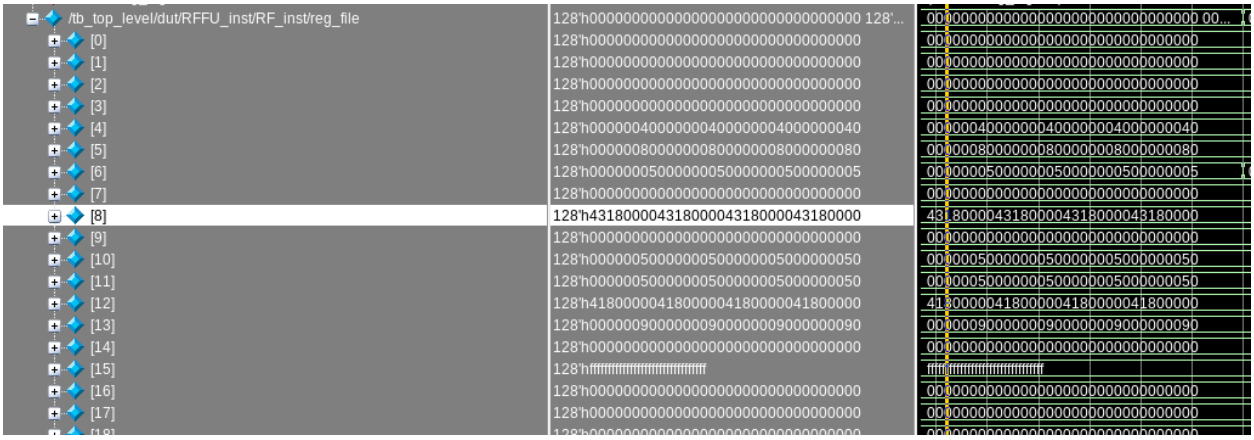
```
# Assume all C[i][j] initialized to 0
for i = 1 to 4:        # Row of A
  for j = 1 to 4:      # Column of B
    for k = 1 to 4:    # Shared dimension
      Fma $C[i][j], $A[i][k], $B[k][j], $C[i][j]
```

We will first examine if C[1][1] is correctly calculated. Then we could expand the code with two more loops to calculate all the elements in matrix C. Below is a SPU ISA assembly code for computing only C[1][1]

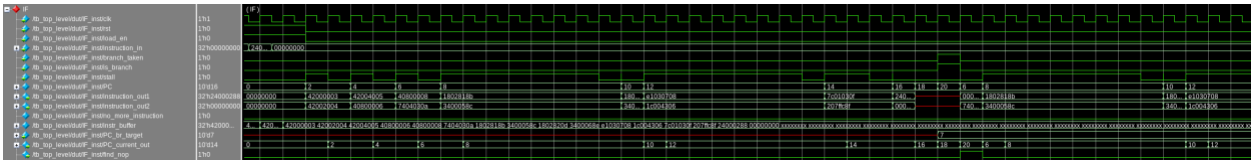
```
// initialization
ila $3, 0 // A base addr
ila $4, 64 // B base addr
ila $5, 128 // C base addr
il $6, 0 // counter k
il $8, 0 // accumulator
LOOP:
```

```
mpyi $10, $6, 16 // $10 = k * 16 byte offset
a $11, $3, $10 // $11 = addr of A[0][k]
lqd $12, 0($11) // $12 = A[0][k]
a $13, $4, $10 // $13 = addr of B[k][0]
lqd $14, 0($13) // $14 = B[k][0]
fma $8, $12, $14, $8 // $8 = A[0][k] * B[k][0] + $8
ai $6, $6, 1
ceqi $15, $6, 4 // if $6 = 4 then $15 = 1
brz $15, LOOP  // if $15 != 0, branch taken
stqd $8, 0($5)
stop
```

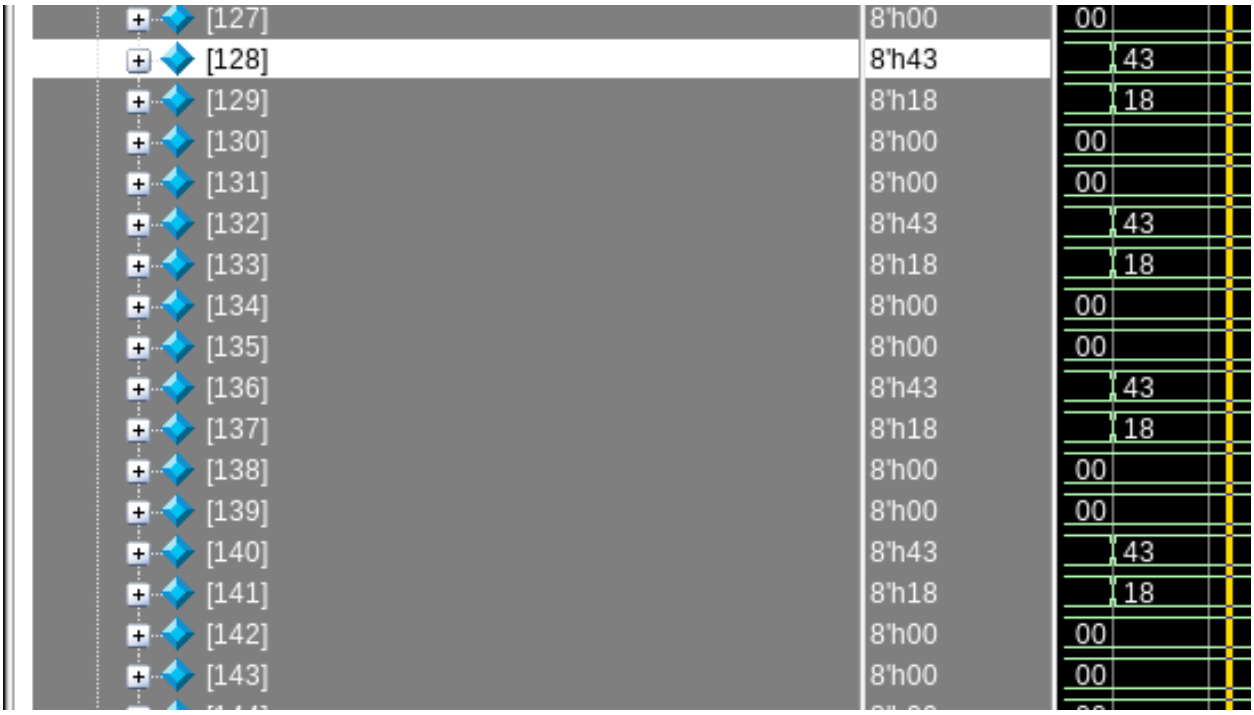
Matrix A and matrix B are preloaded to LocalStore through testbench. Matrix B is pre-transposed when it is preloaded to LocalStore. Expected C[0][0] in $8 should have 152.0 which is 0h43180000 in SP FP.



This waveform shows Register File. $8 is highlighted where the result will be written on. As it is shown, the expected result 0h43180000 is written on $8.



Above waveform is IF stage and it shows that instructions are correctly process in a loop for 4 iterations.



Above waveform is LocalStore memory. From address 128, the result has been written.

Appendix

Appendix A (SPU ISA Table)

No	Name	Mnemonic	RTL Description	Exec Unit (Unit ID)	Exec Pipe	Instruction Latency
1	Add Extended	addx rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) + RB^(4j:4j+3) + RT_(32j+31)	FX1(1)	Even	3
2	Add Halfword	ah rt, ra, rb	for j=0 to 7 RT^(2j:2j+1) ← RA^(2j:2j+1) + RB^(2j:2j+1)	FX1(1)	Even	3
3	Add Halfword Immediate	ahi rt, ra, value	s ← RepLeftBit(I10, 16) for j=0 to 7 RT^(2j:2j+1) ← RA^(2j:2j+1) + s	FX1(1)	Even	3
4	Add Word	a rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) + RB^(4j:4j+3)	FX1(1)	Even	3
5	Add Word Immediate	ai rt, ra, value	t ← RepLeftBit(I10, 32) for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) + t	FX1(1)	Even	3
6	And	and rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) & RB^(4j:4j+3)	FX1(1)	Even	3
7	And Halfword Immediate	andhi rt,ra, value	t ← RepLeftBit(I10, 16) for j=0 to 7 RT^(2j:2j+1) ← RA^(2j:2j+1) + t	FX1(1)	Even	3
8	And Word Immediate	andi rt, ra, value	t ← RepLeftBit(I10, 32) for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) & t	FX1(1)	Even	3
9	Borrow Generate	bg rt, ra, rb	for j=0 to 15 by 4 if (RB^(j::4) ≥^u RT^(j::4) ← 1 else RT^(j::4) ← 0 end	FX1(1)	Even	3

10	Borrow Generate Extended	bgx rt, ra, rb	for j=0 to 15 by 4
			if(RT_(j*8+31) then
			if(RB^(j::4) ≥^u RA^(j::4))
			then RT^(j::4) ← 1
			else RT^(j::4) ← 0
			else
			if(RB^(j::4) >^u RA^(j::4))
			then RT^(j::4) ← 1
			else RT^(j::4) ← 0
			end
				FX1(1)	Even	3

11	Carry Generate	cg rt, ra, rb	for j=0 to 15 by 4 t_(0:32) ← ((0 || RA^(j::4) +	FX1(1)	Even	3

			(0 || RB^(j::4)) RT^(j::4) ← 31_^0 || t_0			
12	Carry Generate Extended	cgx rt, ra, rb	for j=0 to 15 by 4 t_(0:32) ← (0||RA^(j::4) + (0 ||RB^(j::4) + (32_^0 || RT_j * _8+31) RT^(j::4) ← 31_^0 || t_0	FX1(1)	Even	3
13	Compare Equal Halfword	ceqh rt, ra, rb	for j=0 to 15 by 2 IF RA^(j::2) = RB^(j::2) then RT^(j::2) ← 0xFFFF else RT^(j::2) ← 0x0000 end	FX1(1)	Even	3
14	Compare Equal Halfword Immediate	ceqhi rt, ra, value	for j=0 to 15 by 2 IF RA^(j::2) = RepLeftBit(I10, 16) then RT^(j::2) ← 0xFFFF else RT^(j::2) ← 0x0000 end	FX1(1)	Even	3
15	Compare Equal Word	ceq rt, ra, rb	for j=0 to 15 by 4 IF RA^(j::4) = RB^(j::4) then RT^(j::4) ← 0xFFFFFFFF else RT^(j::4) ← 0x00000000 end	FX1(1)	Even	3
16	Compare Equal Word Immediate	ceqi rt, ra, value	for j=0 to 15 by 4 IF RA^(j::4) = RepLeftBit(I10, 32) then RT^(j::4) ← 0xFFFFFFFF else RT^(j::4) ← 0x00000000 end	FX1(1)	Even	3
17	Compare Greater Than Halfword	cgth rt, ra, rb	for j=0 to 15 by 2 IF RA^(j::2) > RB^(j::2) then RT^(j::2) ← 0xFFFF else RT^(j::2) ← 0x0000 end	FX1(1)	Even	3
18	Compare Greater Than Halfword Immediate	cgthi rt, ra, value	for j=0 to 15 by 2 IF RA^(j::2) > RepLeftBit(I10, 16) then RT^(j::2) ← 0xFFFF else RT^(j::2) ← 0x0000 end	FX1(1)	Even	3
19	Compare Greater Than Word	cgt rt, ra, rb	for j=0 to 15 by 4 IF RA^(j::4) > RB^(j::4) then RT^(j::4) ← 0xFFFFFFFF else RT^(j::4) ← 0x00000000 end	FX1(1)	Even	3
20	Compare Greater Than Word Immediate	cgti rt, ra, value	for j=0 to 15 by 4 IF RA^(j::4) > RepLeftBit(I10, 32) then RT^(j::4) ← 0xFFFFFFFF	FX1(1)	Even	3

			else RT^(j::4) ← 0x00000000 end			
21	Count Leading Zeros	clz rt, ra	for j=0 to 15 by 4 t ← 0 u ← RA^(j::4) For m = 0 to 31 if u_m = 1 then leave t ← t + 1 end RT^(j::4) ← t end	FX1(1)	Even	3
22	Equivalent	eqv rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) ⊕ ~RB^(4j:4j+3)	FX1(1)	Even	3
23	Exclusive Or	xor rt,ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) ⊕ RB^(4j:4j+3)	FX1(1)	Even	3
24	Exclusive Or Halfword Immediate	xorhi rt, ra, value	t ← RepLeftBit(I10, 16) for j=0 to 7 RT^(2j:2j+1) ← RA^(2j:2j+1) ⊕ t	FX1(1)	Even	3
25	Exclusive Or Word Immediate	xori rt, ra, value	t ← RepLeftBit(I10, 32) for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) ⊕ t	FX1(1)	Even	3
26	Immediate Load Address	ila rt, symbol	t ← 14_0 || I18 for j=0 to 3 RT^(4j:4j+3) ← t	FX1 (1)	Even	3
27	Immediate Load Halfword	ilh rt, symbol	s ← I16 for j=0 to 7 RT^(2j:2j+1) ← s	FX1 (1)	Even	3
28	Immediate Load Halfword Upper	ilhu rt, symbol	t ← I16 || 0x0000 for j=0 to 3 RT^(4j:4j+3) ← t	FX1 (1)	Even	3
29	Immediate Load Word	il rt, symbol	t ← RepLeftBit(I16, 32) for j=0 to 3 RT^(4j:4j+3) ← t	FX1 (1)	Even	3
30	Immediate Or Halfword Lower	iohl rt, symbol	t ← 0x0000 || I16 for j=0 to 3 RT^(4j:4j+3) ← RT^(4j:4j+3) | t	FX1(1)	Even	3
31	Nand	nand rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← ~(RA^(4j:4j+3) & RB^(4j:4j+3))	FX1(1)	Even	3
32	Nor	nor rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← ~(RA^(4j:4j+3) | RB^(4j:4j+3))	FX1(1)	Even	3

33	Or	or rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) & RB^(4j:4j+3)	FX1(1)	Even	3
34	Or Halfword Immediate	orhi rt, ra, value	t ← RepLeftBit(I10, 16) for j=0 to 7 RT^(2j:2j+1) ← RA^(2j:2j+1) | t	FX1(1)	Even	3
35	Or Word Immediate	ori rt, ra, value	t ← RepLeftBit(I10, 32) for j=0 to 3 RT^(4j:4j+3) ← RA^(4j:4j+3) | t	FX1(1)	Even	3
36	Select Bits	selb rt, ra, rb, rc	RT^(0:15) ← RC^(0:15) & RB^(0:15) | (~RC^(0:15)) & RA^(0:15)	FX1(1)	Even	3
37	Subtract from Extended	sfx rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RB^(4j:4j+3) + ~RA^(4j:4j+3) + RT_(32j+31)	FX1(1)	Even	3
38	Subtract from Halfword	sfh rt, ra, rb	for j=0 to 7 RT^(2j:2j+1) ← RB^(2j:2j+1) + ~RA^(2j:2j+1) + 1	FX1(1)	Even	3
39	Subtract from Halfword Immediate	sfhi rt, ra, value	t ← RepLeftBit(I10, 16) for j=0 to 7 RT^(2j:2j+1) ← t + ~RA^(2j:2j+1) + 1	FX1(1)	Even	3
40	Subtract from Word	sf rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RB^(4j:4j+3) + ~RA(4j:4j+3) + 1	FX1(1)	Even	3
41	Subtract from Word Immediate	sfi rt, ra, value	t ← RepLeftBit(I10, 32) for j=0 to 4 RT^(4j:4j+3) ← t + ~RA^(4j:4j+3) + 1	FX1(1)	Even	3
42	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(2)	Even	4
43	Rotate Halfword Immediate	rothi rt, ra, value	s ← RepLeftBit(I7, 16) & 0x000F for j=0 to 15 by 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	FX2(2)	Even	4

			RT^(j::2) ← r end			
44	Rotate Word	rot rt, ra, rb	for j=0 to 15 by 4 s ← RB^(j::4) & 0x0000001F 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(2)	Even	4
45	Rotate Word Immediate	roti rt, ra, value	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(2)	Even	4
46	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(2)	Even	4
47	Shift Left Halfword Immediate	shlhi rt, ra, value	s ← RepLeftBit(I7, 16) & 0x001F for j=0 to 15 by 2 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(2)	Even	4
48	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	FX2(2)	Even	4

			RT^(j::4) ← r end			
49	Shift Left Word Immediate	shli rt, ra, value	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(2)	Even	4
50	Floating Add	fa rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) = RA^(4j;4j+3) + RB^(4j;4j+3)	SP(3)	Even	7
51	Floating Multiply	fm rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) = RA^(4j;4j+3) * RB^(4j;4j+3)	SP(3)	Even	7
52	Floating Multiply and Add	fma rt, ra, rb, rc	for j=0 to 3 RT^(4j:4j+3) = RA^(4j;4j+3) * RB^(4j;4j+3) + RC^(4j;4j+3)	SP(3)	Even	7
53	Floating Multiply and Subtract	fms rt, ra, rb, rc	for j=0 to 3 RT^(4j:4j+3) = RA^(4j;4j+3) * RB^(4j;4j+3) - RC^(4j;4j+3)	SP(3)	Even	7
54	Floating Negative Multiply and Subtract	fnms rt, ra, rb, rc	for j=0 to 3 RT^(4j:4j+3) = RC^(4j;4j+3) - RA^(4j;4j+3) * RB^(4j;4j+3)	SP(3)	Even	7
55	Floating Subtract	fs rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) = RA^(4j;4j+3) - RB^(4j;4j+3)	SP(3)	Even	7
56	Multiply	mpy rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(2j+2:2j+3) * RB^(2j+2:2j+3)	SP(3)	Even	8
57	Multiply and Add	mpya rt, ra, rb, rc	for j=0 to 3 t(j) ← RA^(4j+2:4j+3) * RB^(4j+2:4j+3) RT^(4j:4j+3) ← t(j) + RC^(4j:4j+3)	SP(3)	Even	8
58	Multiply Immediate	mpyi rt, ra, value	t ← RepLeftBit(I10, 16) for j=0 to 3 RT^(4j:4j+3) ← RA^(2j+2:2j+3) * t	SP(3)	Even	8
59	Multiply Unsigned	mpyu rt, ra, rb	for j=0 to 3 RT^(4j:4j+3) ← RA^(2j+2:2j+3) |*| RB^(2j+2:2j+3)	SP(3)	Even	8

60	Multiply Unsigned Immediate	mpyui rt, ra, value	t ← RepLeftBit(I10, 16) for j=0 to 3 RT^(4j:4j+3) ← RA^(2j+2:2j+3) |*| t	SP(3)	Even	8
61	Multply High	mpyh rt, ra, rb	for j=0 to 3 t(j) ← RT^(4j:4j+1) * RB^(4j+2:2j+3) RT^(4j:4j+3) ← t(j)^(2:3) || 0x0000	SP(3)	Even	8
62	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 ← c end	BYTE(4)	Even	4
63	Absolute Differences of Bytes	absdb rt, ra, rb	for j=0 to 15 if(RB^j >^u RA^j) then RT^j ← RB^j - RA^j else RT^j ← RA^j - RB^j end	BYTE(4)	Even	4
64	Sum Bytes into Halfwords	sumb rt, ra, rb	RT^(0:1) ← RB^0 + RB^1+ RB^2 + RB^3 RT^(2:3) ← RA^0 + RA^1+ RA^2 + RA^3 RT^(4:5) ← RB^4 + RB^5+ RB^6 + RB^7 RT^(6:7) ← RA^4 + RA^5+ RA^6 + RA^7 RT^(8:9) ← RB^8 + RB^9+ RB^10 + RB^11 RT^(10:11) ← RA^8 + RA^9+ RA^10 + RA^11 RT^(12:13) ← RB^12 + RB^13+ RB^14 + RB^15 RT^(14:15) ← RA^12 + RA^13+ RA^14 + RA^15	BYTE(4)	Even	4
65	Average Bytes	avgb rt, ra, rb	for j=0 to 15 RT^j ← ((0x00 || RA^j) + (0x00 || RB^j) + 1)_(7:14)	BYTE(4)	Even	4
66	Rotate Quadword by Bytes Immediate	rotqbyi rt, ra, value	s ← I7_(14:17) for b=0 to 15 if b + s < 16 then r^b ← RA^(b + s) r^b ← RA^(b + s - 16) end RT ← r	PERM(5)	Odd	4
67	Rotate Quadword by Bytes	rotqby rt, ra, rb	s ← RB_(23:31) for b=0 to 15 if b + s < 16 then r^b ← RA^(b + s)	PERM(5)	Odd	4

			else r^b ← RA^(b + s - 16) end RT ← r			
68	Rotate Quadword by Bits Immediate	rotqbii rt, ra, value	s ← I7_(4:6) for b=0 to 127 if b + s < 128 then r_b ← RA_(b + s) else r_b ← RA_(b + s - 128) end RT ← r	PERM(5)	Odd	4
69	Rotate Quadword by Bits	rotqbi rt, ra, rb	s ← RB_(29:31) for b=0 to 127 if b + s < 128 then r_b ← RA_(b + s) else r_b ← RA_(b + s - 128) end RT ← r	PERM(5)	Odd	4
70	Shift Left Quadword by Bits Immediate	shlqbii rt, ra, value	s ← I7 & 0x07 for b=0 to 127 if b+s < 128 then r_b ← RA_(b+s) else r_b ← 0 end RT ← r	PERM(5)	Odd	4
71	Shift Left Quadword by Bits	shlqbi rt, ra, rb	s ← RB_(29:31) for b=0 to 127 if b+s < 128 then r_b ← RA_(b+s) else r_b ← 0 end RT ← r	PERM(5)	Odd	4
72	Shift Left Quadword by Bytes Immediate	shlqbyi rt, ra, value	s ← I7 & 0x1F for b=0 to 15 if b+s < 16 then r^b ← RA^(b+s) else r^b ← 0 end RT ← r	PERM(5)	Odd	4
73	Shift Left Quadword by Bytes	shlqby rt, ra, rb	s ← RB_(27:31) for b=0 to 15 if b+s < 16 then r^b ← RA^(b+s) else r^b ← 0 end RT ← r	PERM(5)	Odd	4
74	Load Quadword (d-form)	lqd rt, symbol(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + RA^(0:3)) & LSLR & 0xFFFFFFF0 RT ← LocStor(LSA, 16)	LS (6)	Odd	7
75	Load Quadword (a-form)	lqa rt, symbol	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR &	LS (6)	Odd	7

			0xFFFFFFF0 RT ← LocStor(LSA,16)			
76	Store Quadword (a-form)	stqa rt, symbol	LSA ← RepLeftBit(I16 || 0b00,32) & LSLR & 0xFFFFFFF0 LocStor(LSA, 16) ← RT	LS (6)	Odd	7
77	Store Quadword (d-form)	stqd rt, symbol(ra)	LSA ← (RepLeftBit(I10 || 0b0000,32) + RA^(0:3)) & LSLR & 0xFFFFFFF0 LocStor(LSA, 16) ← RT	LS (6)	Odd	7
78	Branch Absolute	bra symbol	PC ← RepLeftBit(I16 || 0b00, 32) & LSLR	BRANCH(7)	Odd	2

79	Branch If Not Zero Halfword	brhnz rt, symbol	If RT^(2:3) ≠ 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				BRANCH(7)	Odd	2

80	Branch If Zero Word	brz rt, symbol	If RT^(0:3) = 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				BRANCH(7)	Odd	2

81	Branch If Not Zero Word	brnz rt, symbol	If RT^(0:3) ≠ 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				BRANCH(7)	Odd	2

82	Branch Absolute and Set Link	brasl rt, symbol	RT^(0:3) ← (PC + 4) & LSLR RT^(4:15) ← 0 PC ← (PC + RepLeftBit(I16 || 0b00, 32)) & LSLR	BRANCH(7)	Odd	2
83	Branch Relative and Set Link	brsl rt, symbol	RT^(0:3) ← (PC + 4) & LSLR RT^(4:15) ← 0 PC ← (PC + RepLeftBit(I16 || 0b00, 32)) & LSLR	BRANCH(7)	Odd	2
84	Branch Relative	br symbol	PC ← (PC + RepLeftBit(I16 || 0b00, 32)) & LSLR	BRANCH(7)	Odd	2

85	Branch If Zero Halfword	brhz rt, symbol	If RT^(2:3) = 0 then
			PC ← (PC + RepLeftBit(I16 || 0b00)) & LSLR & 0xFFFFFFFC
			else
			PC ← (PC + 4) & LSLR
				BRANCH(7)	Odd	2

86	No Operation (Load)	lnop		nop(0)	Odd	
87	No Operation (Execute)	nop		nop(0)	Even	
88	Stop and Signal	stop	PC ← PC + 4 & LSLR precise stop	nop(0)		

Appendix B (Top Level.v)

```
module top_level(
  input clk,
  input rst,
  input load_en,
  input [0:31] instruction_in,
  input [0:9] instr_load_addr,
  input preload_en,
  input [0:6] preload_addr,
  input [0:127] preload_values,
  input preload_LS_en,
  input [0:14] preload_LS_addr,
  input [0:127] preload_LS_data

);

wire [0:31] instruction_out1, instruction_out2;

wire [0:31] full_instr_even, full_instr_odd, full_instr_even_to_pipe, full_instr_odd_to_pipe;
wire [0:6] instr_id_even, instr_id_odd, instr_id_even_to_pipe, instr_id_odd_to_pipe;
wire [0:6] reg_dst_even, reg_dst_odd, reg_dst_even_to_pipe, reg_dst_odd_to_pipe;
wire [0:2] unit_id_even, unit_id_odd, unit_id_even_to_pipe, unit_id_odd_to_pipe;
wire [0:3] latency_even, latency_odd, latency_even_to_pipe, latency_odd_to_pipe;
wire reg_wr_even, reg_wr_odd, reg_wr_even_to_pipe, reg_wr_odd_to_pipe;

wire [0:6] imme7_even, imme7_odd, imme7_even_to_pipe, imme7_odd_to_pipe;
wire [0:9] imme10_even, imme10_odd, imme10_even_to_pipe, imme10_odd_to_pipe;
wire [0:15] imme16_even, imme16_odd, imme16_even_to_pipe, imme16_odd_to_pipe;
wire [0:17] imme18_even, imme18_odd, imme18_even_to_pipe, imme18_odd_to_pipe;

wire [0:6] ra_addr_even, rb_addr_even, rc_addr_even,
           ra_addr_odd, rb_addr_odd, rc_addr_odd;
wire [0:127]  ra_data_even_to_pipe, rb_data_even_to_pipe, rc_data_even_to_pipe,
            ra_data_odd_to_pipe, rb_data_odd_to_pipe, rc_data_odd_to_pipe;
```

```
wire [0:142] packed_1stage_even, packed_2stage_even, packed_3stage_even, packed_4stage_even,
             packed_5stage_even, packed_6stage_even, packed_7stage_even,
             packed_1stage_odd, packed_2stage_odd, packed_3stage_odd, packed_4stage_odd,
             packed_5stage_odd, packed_6stage_odd, packed_7stage_odd;

wire [0:6] WB_reg_write_addr_even, WB_reg_write_addr_odd;
wire [0:127] WB_reg_write_data_even, WB_reg_write_data_odd;
wire WB_reg_write_en_even, WB_reg_write_en_odd;

wire [0:9] PC_br_target, PC_pass2ID, PC_pass2RF, PC_pass2odd;

wire stall, flush, branch_taken, is_branch, find_nop, instr1_branch_pass2RF, instr1_branch_pass2odd,
flush_instr2_even, flush_4stage;

IF_wrapper IF_inst(
  .clk(clk),
  .rst(rst),
  .load_en(load_en),
  .instruction_in(instruction_in),
  .instr_load_addr(instr_load_addr),
  .PC_br_target(PC_br_target),
  .is_branch(is_branch),
  .branch_taken(branch_taken),
  .stall(stall),

  .PC_current_out(PC_pass2ID),
  .instruction_out1(instruction_out1),
  .instruction_out2(instruction_out2),
  .find_nop(find_nop)
);

ID_HU_wrapper IDHU_inst(
  .clk(clk),
  .rst(rst),
  .is_branch(is_branch),
  .branch_taken(branch_taken),
  .flush_instr2_even(flush_instr2_even),
```

```
.PC_pass_in(PC_pass2ID),
.instruction_in1(instruction_out1),
.instruction_in2(instruction_out2),
.find_nop(find_nop),

.RF_reg_dst_even(reg_dst_even_to_pipe),
.RF_reg_wr_even(reg_wr_even_to_pipe),
.RF_latency_even(latency_even_to_pipe),
.packed_2stage_even(packed_2stage_even),
.packed_3stage_even(packed_3stage_even),
.packed_4stage_even(packed_4stage_even),
.packed_5stage_even(packed_5stage_even),
.packed_6stage_even(packed_6stage_even),

.RF_reg_dst_odd(reg_dst_odd_to_pipe),
.RF_reg_wr_odd(reg_wr_odd_to_pipe),
.RF_latency_odd(latency_odd_to_pipe),
.packed_2stage_odd(packed_2stage_odd),
.packed_3stage_odd(packed_3stage_odd),
.packed_4stage_odd(packed_4stage_odd),
.packed_5stage_odd(packed_5stage_odd),
.packed_6stage_odd(packed_6stage_odd),

.instr_id_even(instr_id_even),
.reg_dst_even(reg_dst_even),
.unit_id_even(unit_id_even),
.latency_even(latency_even),
.reg_wr_even(reg_wr_even),
.imme7_even(imme7_even),
.imme10_even(imme10_even),
.imme16_even(imme16_even),
.imme18_even(imme18_even),

.ra_addr_even(ra_addr_even),
.rb_addr_even(rb_addr_even),
.rc_addr_even(rc_addr_even),
```

```
  .instr_id_odd(instr_id_odd),
  .reg_dst_odd(reg_dst_odd),
  .unit_id_odd(unit_id_odd),
  .latency_odd(latency_odd),
  .reg_wr_odd(reg_wr_odd),
  .imme7_odd(imme7_odd),
  .imme10_odd(imme10_odd),
  .imme16_odd(imme16_odd),
  .imme18_odd(imme18_odd),

  .ra_addr_odd(ra_addr_odd),
  .rb_addr_odd(rb_addr_odd),
  .rc_addr_odd(rc_addr_odd),

  .instr1_branch(instr1_branch_pass2RF),
  .stall(stall),
  .flush(flush),
  .flush_4stage(flush_4stage),
  .PC_pass_out(PC_pass2RF)
);

RF_FU_wrapper RFFU_inst(
  .clk(clk),
  .rst(rst),
  .flush(flush),
  .PC_pass_in(PC_pass2RF),
  .instr1_branch(instr1_branch_pass2RF),
  // from ID
  .instr_id_even(instr_id_even),
  .reg_dst_even(reg_dst_even),
  .unit_id_even(unit_id_even),
  .latency_even(latency_even),
  .reg_wr_even(reg_wr_even),
  .imme7_even(imme7_even),
  .imme10_even(imme10_even),
  .imme16_even(imme16_even),
  .imme18_even(imme18_even),
```

```
.instr_id_odd(instr_id_odd),
.reg_dst_odd(reg_dst_odd),
.unit_id_odd(unit_id_odd),
.latency_odd(latency_odd),
.reg_wr_odd(reg_wr_odd),
.imme7_odd(imme7_odd),
.imme10_odd(imme10_odd),
.imme16_odd(imme16_odd),
.imme18_odd(imme18_odd),

// register file access
.ra_addr_even(ra_addr_even),
.rb_addr_even(rb_addr_even),
.rc_addr_even(rc_addr_even),
.ra_addr_odd(ra_addr_odd),
.rb_addr_odd(rb_addr_odd),
.rc_addr_odd(rc_addr_odd),
.reg_write_en_1(WB_reg_write_en_even),
.reg_write_en_2(WB_reg_write_en_odd),
.reg_write_addr_1(WB_reg_write_addr_even),
.reg_write_addr_2(WB_reg_write_addr_odd),
.reg_write_data_1(WB_reg_write_data_even),
.reg_write_data_2(WB_reg_write_data_odd),

// inputs for forwarding unit
.packed_2stage_even(packed_2stage_even),
.packed_3stage_even(packed_3stage_even),
.packed_4stage_even(packed_4stage_even),
.packed_5stage_even(packed_5stage_even),
.packed_6stage_even(packed_6stage_even),
.packed_7stage_even(packed_7stage_even),

.packed_2stage_odd(packed_2stage_odd),
.packed_3stage_odd(packed_3stage_odd),
.packed_4stage_odd(packed_4stage_odd),
.packed_5stage_odd(packed_5stage_odd),
```

```
.packed_6stage_odd(packed_6stage_odd),
.packed_7stage_odd(packed_7stage_odd),

// output
.out_instr_id_even(instr_id_even_to_pipe),
.out_reg_dst_even(reg_dst_even_to_pipe),
.out_unit_id_even(unit_id_even_to_pipe),
.out_latency_even(latency_even_to_pipe),
.out_reg_wr_even(reg_wr_even_to_pipe),
.out_imme7_even(imme7_even_to_pipe),
.out_imme10_even(imme10_even_to_pipe),
.out_imme16_even(imme16_even_to_pipe),
.out_imme18_even(imme18_even_to_pipe),

.out_instr_id_odd(instr_id_odd_to_pipe),
.out_reg_dst_odd(reg_dst_odd_to_pipe),
.out_unit_id_odd(unit_id_odd_to_pipe),
.out_latency_odd(latency_odd_to_pipe),
.out_reg_wr_odd(reg_wr_odd_to_pipe),
.out_imme7_odd(imme7_odd_to_pipe),
.out_imme10_odd(imme10_odd_to_pipe),
.out_imme16_odd(imme16_odd_to_pipe),
.out_imme18_odd(imme18_odd_to_pipe),

.ra_data_even(ra_data_even_to_pipe),
.rb_data_even(rb_data_even_to_pipe),
.rc_data_even(rc_data_even_to_pipe),
.ra_data_odd(ra_data_odd_to_pipe),
.rb_data_odd(rb_data_odd_to_pipe),
.rc_data_odd(rc_data_odd_to_pipe),

.instr1_branch_out(instr1_branch_pass2odd),
.PC_pass_out(PC_pass2odd),

.preload_en(preload_en),
.preload_addr(preload_addr),
.preload_values(preload_values)
```

```
);

Even_Pipe Even_Pipe_inst (
  .clk(clk),
  .rst(rst),
  .flush(flush),
  .flush_4stage(flush_4stage),
  .instr_id(instr_id_even_to_pipe),
  .reg_dst(reg_dst_even_to_pipe),
  .unit_id(unit_id_even_to_pipe),
  .latency(latency_even_to_pipe),
  .reg_wr(reg_wr_even_to_pipe),
  .ra_data(ra_data_even_to_pipe),
  .rb_data(rb_data_even_to_pipe),
  .rc_data(rc_data_even_to_pipe),
  .imme7(imme7_even_to_pipe),
  .imme10(imme10_even_to_pipe),
  .imme16(imme16_even_to_pipe),
  .imme18(imme18_even_to_pipe),
  .packed_2stage(packed_2stage_even),
  .packed_3stage(packed_3stage_even),
  .packed_4stage(packed_4stage_even),
  .packed_5stage(packed_5stage_even),
  .packed_6stage(packed_6stage_even),
  .packed_7stage(packed_7stage_even),
  .WB_reg_write_addr(WB_reg_write_addr_even),
  .WB_reg_write_data(WB_reg_write_data_even),
  .WB_reg_write_en(WB_reg_write_en_even)
);

Odd_Pipe Odd_Pipe_inst(
  .clk(clk),
  .rst(rst),
  .flush(flush),
  .instr_id(instr_id_odd_to_pipe),
  .reg_dst(reg_dst_odd_to_pipe),
  .unit_id(unit_id_odd_to_pipe),
```

```
  .latency(latency_odd_to_pipe),
  .reg_wr(reg_wr_odd_to_pipe),
  .ra_data(ra_data_odd_to_pipe),
  .rb_data(rb_data_odd_to_pipe),
  .rc_data(rc_data_odd_to_pipe),
  .imme7(imme7_odd_to_pipe),
  .imme10(imme10_odd_to_pipe),
  .imme16(imme16_odd_to_pipe),
  .imme18(imme18_odd_to_pipe),
  .current_PC(PC_pass2odd),
  .instr1_branch(instr1_branch_pass2odd),
  .packed_2stage(packed_2stage_odd),
  .packed_3stage(packed_3stage_odd),
  .packed_4stage(packed_4stage_odd),
  .packed_5stage(packed_5stage_odd),
  .packed_6stage(packed_6stage_odd),
  .packed_7stage(packed_7stage_odd),
  .WB_reg_write_addr(WB_reg_write_addr_odd),
  .WB_reg_write_data(WB_reg_write_data_odd),
  .WB_reg_write_en(WB_reg_write_en_odd),
  .new_PC(PC_br_target),
  .branch_taken(branch_taken),
  .is_branch(is_branch),
  .flush_instr2_even(flush_instr2_even),
  .preload_LS_en(preload_LS_en),
  .preload_LS_addr(preload_LS_addr),
  .preload_LS_data(preload_LS_data)
);

endmodule
```

Appendix C (IF_wrapper.v)

```
module IF_wrapper(
    input  clk,
    input  rst,
    input  load_en,
```

```
    input  [0:31] instruction_in,
    input [0:8] instr_load_addr,
    input [0:8] PC_br_target,
    input  branch_taken,
    input is_branch,
    input  stall,

    output reg [0:8] PC_current_out,
    output reg [0:31] instruction_out1,
    output reg [0:31] instruction_out2,
    output reg find_nop
);
    localparam LINE_LENGTH = 512;  // 512 instructions (2KB total)

    // Instruction buffer memory (2KB = 512 x 32-bit instructions)
    reg [0:31] instr_buffer [0:LINE_LENGTH-1];

    // Program Counter
    reg no_more_instruction;
    reg [0:8] PC;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            PC <= 10'b0;  // Reset PC to 0
            instruction_out1 <= 32'b0;
            instruction_out2 <= 32'b0;
            PC_current_out <= 10'b0;  // Reset output PC
        end
        else begin
            if (is_branch && branch_taken) begin // branch taken
                if (PC_br_target[9] == 1'b1) begin // misaligned target
                    instruction_out1 <= 32'b0;
                    instruction_out2 <= instr_buffer[PC_br_target - 2];
                    PC <= PC_br_target - 1;
                    find_nop <= 1'b1; // telling ID stage to find which nop is needed
                end
                else begin // aligned target
```

```
                instruction_out1 <= instr_buffer[PC_br_target - 2];
                instruction_out2 <= instr_buffer[PC_br_target - 1];
                PC <= PC_br_target;
                find_nop <= 1'b0;
            end
        end
        else if (stall) begin
            // hold PC
            instruction_out1 <= instr_buffer[PC - 2];
            instruction_out2 <= instr_buffer[PC - 1];
            PC <= PC;
            find_nop <= 1'b0;
        end
        else begin
            // Normal operation, increment PC
            instruction_out1 <= instr_buffer[PC];
            instruction_out2 <= instr_buffer[PC + 1];
            PC <= PC + 2;
            find_nop <= 1'b0;
        end
        PC_current_out <= PC;  // Update output PC
    end

end

integer i;
    // Process to load and output instructions
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            if (load_en) begin
                instr_buffer[instr_load_addr] <= instruction_in;  // Load instruction
            end
                // for (i = 0; i < LINE_LENGTH; i = i + 1)
                //     instr_buffer[i] <= 32'b0;
                no_more_instruction <= 1'b0;
        end else begin
            if (PC == LINE_LENGTH-1) begin
```

```
            no_more_instruction <= 1'b1;
        end else begin
            no_more_instruction <= 1'b0;
        end
    end
  end
endmodule
```

Appendix D (ID_HU_wrapper.v)

```
module ID_HU_wrapper(
  input clk,
  input rst,

  input is_branch,
  input branch_taken,
  input flush_instr2_even,
  input [0:8] PC_pass_in,
  input [0:31] instruction_in1,
  input [0:31] instruction_in2,
  input find_nop,

  input [0:6] RF_reg_dst_even,
  input RF_reg_wr_even,
  input [0:3] RF_latency_even,
  input [0:142] packed_2stage_even,
  input [0:142] packed_3stage_even,
  input [0:142] packed_4stage_even,
  input [0:142] packed_5stage_even,
  input [0:142] packed_6stage_even,

  input [0:6] RF_reg_dst_odd,
  input RF_reg_wr_odd,
  input [0:3] RF_latency_odd,
  input [0:142] packed_2stage_odd,
  input [0:142] packed_3stage_odd,
```

```
input [0:142] packed_4stage_odd,
input [0:142] packed_5stage_odd,
input [0:142] packed_6stage_odd,

output reg [0:6] instr_id_even,
output reg [0:6] reg_dst_even,
output reg [0:2] unit_id_even,
output reg [0:3] latency_even,
output reg reg_wr_even,
output reg [0:6] imme7_even,
output reg [0:9] imme10_even,
output reg [0:15] imme16_even,
output reg [0:17] imme18_even,

output reg [0:6] ra_addr_even,
output reg [0:6] rb_addr_even,
output reg [0:6] rc_addr_even,

output reg [0:6] instr_id_odd,
output reg [0:6] reg_dst_odd,
output reg [0:2] unit_id_odd,
output reg [0:3] latency_odd,
output reg reg_wr_odd,
output reg [0:6] imme7_odd,
output reg [0:9] imme10_odd,
output reg [0:15] imme16_odd,
output reg [0:17] imme18_odd,

output reg [0:6] ra_addr_odd,
output reg [0:6] rb_addr_odd,
output reg [0:6] rc_addr_odd,

output reg instr1_branch,
output wire stall,
output reg flush,
output reg flush_4stage,
output reg [0:9] PC_pass_out
```

```
);

`include "opcode_package.vh"

wire [0:142] packed_IDstage_even, packed_IDstage_odd;
reg [0:142]   packed_RFFUstage_even, packed_1stage_even, packed_RFFUstage_odd, packed_1stage_odd;

wire [0:6] temp_instr_id_1, temp_instr_id_2;
wire [0:6] temp_reg_dst_1, temp_reg_dst_2;
wire [0:2] temp_unit_id_1, temp_unit_id_2;
wire [0:3] temp_latency_1, temp_latency_2;
wire temp_reg_wr_1, temp_reg_wr_2;
wire [0:6] temp_imme7_1, temp_imme7_2;
wire [0:9] temp_imme10_1, temp_imme10_2;
wire [0:15] temp_imme16_1, temp_imme16_2;
wire [0:17] temp_imme18_1, temp_imme18_2;
wire [0:6] temp_ra_addr_1, temp_ra_addr_2;
wire [0:6] temp_rb_addr_1, temp_rb_addr_2;
wire [0:6] temp_rc_addr_1, temp_rc_addr_2;

wire [0:6] temp_reg_dst_even, temp_reg_dst_odd, temp_ra_addr_even, temp_ra_addr_odd,
           temp_rb_addr_even, temp_rb_addr_odd, temp_rc_addr_even, temp_rc_addr_odd;

wire temp_stall, temp_dependent_stall, temp_flush, temp_instr1_type, temp_instr2_type;

wire [0:6] temp_instr_id_even, temp_instr_id_odd;
reg [0:1] instr_dependent_protocol; // 01: even, 10: odd 00: reset
reg [0:1] data_dependent_protocol; // 01: even, 10: odd 00: reset

Instruction_Decode ID_inst(
  .instruction_in1(instruction_in1),
  .instruction_in2(instruction_in2),

  .instr_id_1(temp_instr_id_1),
  .reg_dst_1(temp_reg_dst_1),
  .unit_id_1(temp_unit_id_1),
  .latency_1(temp_latency_1),
```

```
  .reg_wr_1(temp_reg_wr_1),
  .imme7_1(temp_imme7_1),
  .imme10_1(temp_imme10_1),
  .imme16_1(temp_imme16_1),
  .imme18_1(temp_imme18_1),

  .ra_addr_1(temp_ra_addr_1),
  .rb_addr_1(temp_rb_addr_1),
  .rc_addr_1(temp_rc_addr_1),

  .instr_id_2(temp_instr_id_2),
  .reg_dst_2(temp_reg_dst_2),
  .unit_id_2(temp_unit_id_2),
  .latency_2(temp_latency_2),
  .reg_wr_2(temp_reg_wr_2),
  .imme7_2(temp_imme7_2),
  .imme10_2(temp_imme10_2),
  .imme16_2(temp_imme16_2),
  .imme18_2(temp_imme18_2),

  .ra_addr_2(temp_ra_addr_2),
  .rb_addr_2(temp_rb_addr_2),
  .rc_addr_2(temp_rc_addr_2),

  .instr1_type(temp_instr1_type),
  .instr2_type(temp_instr2_type)
);

assign temp_reg_dst_even = temp_instr1_type ? temp_reg_dst_1 : temp_reg_dst_2;
assign temp_ra_addr_even = temp_instr1_type ? temp_ra_addr_1 : temp_ra_addr_2;
assign temp_rb_addr_even = temp_instr1_type ? temp_rb_addr_1 : temp_rb_addr_2;
assign temp_rc_addr_even = temp_instr1_type ? temp_rc_addr_1 : temp_rc_addr_2;
assign temp_reg_dst_odd = temp_instr2_type ? temp_reg_dst_1 : temp_reg_dst_2;
assign temp_ra_addr_odd = temp_instr2_type ? temp_ra_addr_1 : temp_ra_addr_2;
assign temp_rb_addr_odd = temp_instr2_type ? temp_rb_addr_1 : temp_rb_addr_2;
assign temp_rc_addr_odd = temp_instr2_type ? temp_rc_addr_1 : temp_rc_addr_2;
assign temp_instr_id_even = temp_instr1_type ? temp_instr_id_1 : temp_instr_id_2;
```

```
assign temp_instr_id_odd = temp_instr2_type ? temp_instr_id_1 : temp_instr_id_2;

Hazard_Unit HU_inst(
  .instr_dependent_protocol(instr_dependent_protocol),
  .data_dependent_protocol(data_dependent_protocol),
  .instr1_type(temp_instr1_type),
  .instr2_type(temp_instr2_type),
  .is_branch(is_branch),
  .branch_taken(branch_taken),

  .instr_id_even(temp_instr_id_even),
  .reg_dst_even(temp_reg_dst_even),
  .ra_addr_even(temp_ra_addr_even),
  .rb_addr_even(temp_rb_addr_even),
  .rc_addr_even(temp_rc_addr_even),
  .instr_id_odd(temp_instr_id_odd),
  .reg_dst_odd(temp_reg_dst_odd),
  .ra_addr_odd(temp_ra_addr_odd),
  .rb_addr_odd(temp_rb_addr_odd),
  .rc_addr_odd(temp_rc_addr_odd),

  .ID_reg_dst_even(reg_dst_even),
  .ID_reg_wr_even(reg_wr_even),
  .RF_reg_dst_even(RF_reg_dst_even),
  .RF_reg_wr_even(RF_reg_wr_even),
  .RF_latency_even(RF_latency_even),
  .packed_2stage_even(packed_2stage_even),
  .packed_3stage_even(packed_3stage_even),
  .packed_4stage_even(packed_4stage_even),
  .packed_5stage_even(packed_5stage_even),
  .packed_6stage_even(packed_6stage_even),

  .ID_reg_dst_odd(reg_dst_odd),
  .ID_reg_wr_odd(reg_wr_odd),
  .RF_reg_dst_odd(RF_reg_dst_odd),
  .RF_reg_wr_odd(RF_reg_wr_odd),
  .RF_latency_odd(RF_latency_odd),
```

```
  .packed_2stage_odd(packed_2stage_odd),
  .packed_3stage_odd(packed_3stage_odd),
  .packed_4stage_odd(packed_4stage_odd),
  .packed_5stage_odd(packed_5stage_odd),
  .packed_6stage_odd(packed_6stage_odd),

  .stall(temp_stall),
  .dependent_stall(temp_dependent_stall),
  .flush(temp_flush)
);

assign packed_IDstage_even = {unit_id_even, 128'd0, reg_dst_even, latency_even, reg_wr_even};
assign packed_IDstage_odd = {unit_id_odd, 128'd0, reg_dst_odd, latency_odd, reg_wr_odd};
assign stall = temp_stall | temp_dependent_stall;


always @(posedge clk or posedge rst) begin
instr1_branch <= 1'b0;
  if (rst) begin
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
    latency_odd <= 4'b0;
```

```
  reg_wr_odd <= 1'b0;
  imme7_odd <= 7'b0;
  imme10_odd <= 10'b0;
  imme16_odd <= 16'b0;
  imme18_odd <= 18'b0;

  ra_addr_odd <= 7'b0;
  rb_addr_odd <= 7'b0;
  rc_addr_odd <= 7'b0;

  packed_RFFUstage_even <= 142'd0;
  packed_RFFUstage_odd <= 142'd0;

  instr1_branch <= 1'b0;
  flush <= 1'b0;
  PC_pass_out <= 10'b0;
  instr_dependent_protocol <= 2'b00;
  data_dependent_protocol <= 2'b00;
end
else if (temp_dependent_stall) begin // when dependency stall happens, allow first instruction go to first
  if(temp_instr1_type == 1'b1 && temp_instr2_type == 1'b1) begin // if both instrs are even
    instr_id_even <= temp_instr_id_1;
    reg_dst_even <= temp_reg_dst_1;
    unit_id_even <= temp_unit_id_1;
    latency_even <= temp_latency_1;
    reg_wr_even <= temp_reg_wr_1;
    imme7_even <= temp_imme7_1;
    imme10_even <= temp_imme10_1;
    imme16_even <= temp_imme16_1;
    imme18_even <= temp_imme18_1;
    ra_addr_even <= temp_ra_addr_1;
    rb_addr_even <= temp_rb_addr_1;
    rc_addr_even <= temp_rc_addr_1;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
```

```
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;

    instr_dependent_protocol <= 2'b01;

  end
  else if (temp_instr1_type == 1'b0 && temp_instr2_type == 1'b0) begin // if both instrs are odd
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_1;
    reg_dst_odd <= temp_reg_dst_1;
    unit_id_odd <= temp_unit_id_1;
    latency_odd <= temp_latency_1;
    reg_wr_odd <= temp_reg_wr_1;
    imme7_odd <= temp_imme7_1;
    imme10_odd <= temp_imme10_1;
    imme16_odd <= temp_imme16_1;
```

```
    imme18_odd <= temp_imme18_1;

    ra_addr_odd <= temp_ra_addr_1;
    rb_addr_odd <= temp_rb_addr_1;
    rc_addr_odd <= temp_rc_addr_1;

    instr_dependent_protocol <= 2'b10;
  end
  // data dependent stall
  else if (temp_instr1_type == 1'b1) begin  // instr1 go first and it's even
    instr_id_even <= temp_instr_id_1;
    reg_dst_even <= temp_reg_dst_1;
    unit_id_even <= temp_unit_id_1;
    latency_even <= temp_latency_1;
    reg_wr_even <= temp_reg_wr_1;
    imme7_even <= temp_imme7_1;
    imme10_even <= temp_imme10_1;
    imme16_even <= temp_imme16_1;
    imme18_even <= temp_imme18_1;

    ra_addr_even <= temp_ra_addr_1;
    rb_addr_even <= temp_rb_addr_1;
    rc_addr_even <= temp_rc_addr_1;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;
```

```
    data_dependent_protocol <= 2'b01;
  end
  else if (temp_instr1_type == 1'b0) begin // instr1 go first and it's odd
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_1;
    reg_dst_odd <= temp_reg_dst_1;
    unit_id_odd <= temp_unit_id_1;
    latency_odd <= temp_latency_1;
    reg_wr_odd <= temp_reg_wr_1;
    imme7_odd <= temp_imme7_1;
    imme10_odd <= temp_imme10_1;
    imme16_odd <= temp_imme16_1;
    imme18_odd <= temp_imme18_1;

    ra_addr_odd <= temp_ra_addr_1;
    rb_addr_odd <= temp_rb_addr_1;
    rc_addr_odd <= temp_rc_addr_1;

    data_dependent_protocol <= 2'b10;
  end
end
else if (temp_stall) begin
  // feed nop to both pipes
```

```
  instr_id_even <= `instr_ID_nop;
  reg_dst_even <= 7'b0;
  unit_id_even <= 3'b0;
  latency_even <= 4'b0;
  reg_wr_even <= 1'b0;
  imme7_even <= 7'b0;
  imme10_even <= 10'b0;
  imme16_even <= 16'b0;
  imme18_even <= 18'b0;

  ra_addr_even <= 7'b0;
  rb_addr_even <= 7'b0;
  rc_addr_even <= 7'b0;

  instr_id_odd <= `instr_ID_lnop;
  reg_dst_odd <= 7'b0;
  unit_id_odd <= 3'b0;
  latency_odd <= 4'b0;
  reg_wr_odd <= 1'b0;
  imme7_odd <= 7'b0;
  imme10_odd <= 10'b0;
  imme16_odd <= 16'b0;
  imme18_odd <= 18'b0;

  ra_addr_odd <= 7'b0;
  rb_addr_odd <= 7'b0;
  rc_addr_odd <= 7'b0;

  // feed 0s to internal packed stages
  packed_RFFUstage_even <= 142'b0;
  packed_RFFUstage_odd <= 142'b0;
  packed_1stage_even <= packed_RFFUstage_even;
  packed_1stage_odd <= packed_RFFUstage_odd;
end
else if (instr_dependent_protocol != 2'b00) begin // when dependency stall, allow the next instruction to go with nop
  if (instr_dependent_protocol == 2'b01) begin // next even instr go
    instr_id_even <= temp_instr_id_2;
```

```
    reg_dst_even <= temp_reg_dst_2;
    unit_id_even <= temp_unit_id_2;
    latency_even <= temp_latency_2;
    reg_wr_even <= temp_reg_wr_2;
    imme7_even <= temp_imme7_2;
    imme10_even <= temp_imme10_2;
    imme16_even <= temp_imme16_2;
    imme18_even <= temp_imme18_2;

    ra_addr_even <= temp_ra_addr_2;
    rb_addr_even <= temp_rb_addr_2;
    rc_addr_even <= temp_rc_addr_2;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;

    instr_dependent_protocol <= 2'b00;
  end
  else if (instr_dependent_protocol == 2'b10) begin // next odd instr go
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
```

```
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_2;
    reg_dst_odd <= temp_reg_dst_2;
    unit_id_odd <= temp_unit_id_2;
    latency_odd <= temp_latency_2;
    reg_wr_odd <= temp_reg_wr_2;
    imme7_odd <= temp_imme7_2;
    imme10_odd <= temp_imme10_2;
    imme16_odd <= temp_imme16_2;
    imme18_odd <= temp_imme18_2;

    ra_addr_odd <= temp_ra_addr_2;
    rb_addr_odd <= temp_rb_addr_2;
    rc_addr_odd <= temp_rc_addr_2;

    instr_dependent_protocol <= 2'b00;
  end
end
else if (data_dependent_protocol != 2'b00) begin
  if (data_dependent_protocol == 2'b01) begin // next odd instr2 go when instr1 was even
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
```

```
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_2;
    reg_dst_odd <= temp_reg_dst_2;
    unit_id_odd <= temp_unit_id_2;
    latency_odd <= temp_latency_2;
    reg_wr_odd <= temp_reg_wr_2;
    imme7_odd <= temp_imme7_2;
    imme10_odd <= temp_imme10_2;
    imme16_odd <= temp_imme16_2;
    imme18_odd <= temp_imme18_2;

    ra_addr_odd <= temp_ra_addr_2;
    rb_addr_odd <= temp_rb_addr_2;
    rc_addr_odd <= temp_rc_addr_2;

    data_dependent_protocol <= 2'b00;
  end
  else if (data_dependent_protocol == 2'b10) begin // next even instr2 go when instr1 was odd
    instr_id_even <= temp_instr_id_2;
    reg_dst_even <= temp_reg_dst_2;
    unit_id_even <= temp_unit_id_2;
    latency_even <= temp_latency_2;
    reg_wr_even <= temp_reg_wr_2;
    imme7_even <= temp_imme7_2;
    imme10_even <= temp_imme10_2;
    imme16_even <= temp_imme16_2;
    imme18_even <= temp_imme18_2;

    ra_addr_even <= temp_ra_addr_2;
    rb_addr_even <= temp_rb_addr_2;
    rc_addr_even <= temp_rc_addr_2;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
```

```
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;

    data_dependent_protocol <= 2'b00;
  end
end
else if (find_nop == 1'b1) begin
  if (temp_instr2_type == 1'b1) begin // if instr is even, lnop asserted
    instr_id_even <= temp_instr_id_2;
    reg_dst_even <= temp_reg_dst_2;
    unit_id_even <= temp_unit_id_2;
    latency_even <= temp_latency_2;
    reg_wr_even <= temp_reg_wr_2;
    imme7_even <= temp_imme7_2;
    imme10_even <= temp_imme10_2;
    imme16_even <= temp_imme16_2;
    imme18_even <= temp_imme18_2;

    ra_addr_even <= temp_ra_addr_2;
    rb_addr_even <= temp_rb_addr_2;
    rc_addr_even <= temp_rc_addr_2;

    instr_id_odd <= `instr_ID_lnop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
```

```
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;
  end
  else if (temp_instr2_type == 1'b0) begin // if instr is odd, nop asserted
    instr_id_even <= `instr_ID_nop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_2;
    reg_dst_odd <= temp_reg_dst_2;
    unit_id_odd <= temp_unit_id_2;
    latency_odd <= temp_latency_2;
    reg_wr_odd <= temp_reg_wr_2;
    imme7_odd <= temp_imme7_2;
    imme10_odd <= temp_imme10_2;
    imme16_odd <= temp_imme16_2;
    imme18_odd <= temp_imme18_2;

    ra_addr_odd <= temp_ra_addr_2;
    rb_addr_odd <= temp_rb_addr_2;
    rc_addr_odd <= temp_rc_addr_2;
  end
end
```

```
else if (temp_instr_id_1 == `instr_ID_stop || temp_instr_id_2 == `instr_ID_stop) begin // when one of instr is stop
 if(temp_instr_id_1 != `instr_ID_stop && temp_instr1_type == 1'b1) begin
  instr_id_even <= temp_instr_id_1;
  reg_dst_even <= temp_reg_dst_1;
  unit_id_even <= temp_unit_id_1;
  latency_even <= temp_latency_1;
  reg_wr_even <= temp_reg_wr_1;
  imme7_even <= temp_imme7_1;
  imme10_even <= temp_imme10_1;
  imme16_even <= temp_imme16_1;
  imme18_even <= temp_imme18_1;
  ra_addr_even <= temp_ra_addr_1;
  rb_addr_even <= temp_rb_addr_1;
  rc_addr_even <= temp_rc_addr_1;

  instr_id_odd <= `instr_ID_stop;
  reg_dst_odd <= 7'b0;
  unit_id_odd <= 3'b0;
  latency_odd <= 4'b0;
  reg_wr_odd <= 1'b0;
  imme7_odd <= 7'b0;
  imme10_odd <= 10'b0;
  imme16_odd <= 16'b0;
  imme18_odd <= 18'b0;

  ra_addr_odd <= 7'b0;
  rb_addr_odd <= 7'b0;
  rc_addr_odd <= 7'b0;
 end
 else if (temp_instr_id_1 != `instr_ID_stop && temp_instr1_type == 1'b0) begin
  instr_id_even <= `instr_ID_stop;
  reg_dst_even <= 7'b0;
  unit_id_even <= 3'b0;
  latency_even <= 4'b0;
  reg_wr_even <= 1'b0;
  imme7_even <= 7'b0;
  imme10_even <= 10'b0;
```

```
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_1;
    reg_dst_odd <= temp_reg_dst_1;
    unit_id_odd <= temp_unit_id_1;
    latency_odd <= temp_latency_1;
    reg_wr_odd <= temp_reg_wr_1;
    imme7_odd <= temp_imme7_1;
    imme10_odd <= temp_imme10_1;
    imme16_odd <= temp_imme16_1;
    imme18_odd <= temp_imme18_1;

    ra_addr_odd <= temp_ra_addr_1;
    rb_addr_odd <= temp_rb_addr_1;
    rc_addr_odd <= temp_rc_addr_1;

  end
  else if(temp_instr_id_2 != `instr_ID_stop && temp_instr2_type == 1'b1) begin
    instr_id_even <= temp_instr_id_2;
    reg_dst_even <= temp_reg_dst_2;
    unit_id_even <= temp_unit_id_2;
    latency_even <= temp_latency_2;
    reg_wr_even <= temp_reg_wr_2;
    imme7_even <= temp_imme7_2;
    imme10_even <= temp_imme10_2;
    imme16_even <= temp_imme16_2;
    imme18_even <= temp_imme18_2;

    ra_addr_even <= temp_ra_addr_2;
    rb_addr_even <= temp_rb_addr_2;
    rc_addr_even <= temp_rc_addr_2;
```

```
    instr_id_odd <= `instr_ID_stop;
    reg_dst_odd <= 7'b0;
    unit_id_odd <= 3'b0;
    latency_odd <= 4'b0;
    reg_wr_odd <= 1'b0;
    imme7_odd <= 7'b0;
    imme10_odd <= 10'b0;
    imme16_odd <= 16'b0;
    imme18_odd <= 18'b0;

    ra_addr_odd <= 7'b0;
    rb_addr_odd <= 7'b0;
    rc_addr_odd <= 7'b0;

  end
  else if(temp_instr_id_2 != `instr_ID_stop && temp_instr2_type == 1'b0) begin
    instr_id_even <= `instr_ID_stop;
    reg_dst_even <= 7'b0;
    unit_id_even <= 3'b0;
    latency_even <= 4'b0;
    reg_wr_even <= 1'b0;
    imme7_even <= 7'b0;
    imme10_even <= 10'b0;
    imme16_even <= 16'b0;
    imme18_even <= 18'b0;

    ra_addr_even <= 7'b0;
    rb_addr_even <= 7'b0;
    rc_addr_even <= 7'b0;

    instr_id_odd <= temp_instr_id_2;
    reg_dst_odd <= temp_reg_dst_2;
    unit_id_odd <= temp_unit_id_2;
    latency_odd <= temp_latency_2;
    reg_wr_odd <= temp_reg_wr_2;
    imme7_odd <= temp_imme7_2;
    imme10_odd <= temp_imme10_2;
```

```
    imme16_odd <= temp_imme16_2;
    imme18_odd <= temp_imme18_2;

    ra_addr_odd <= temp_ra_addr_2;
    rb_addr_odd <= temp_rb_addr_2;
    rc_addr_odd <= temp_rc_addr_2;

   end
   else if(temp_instr_id_2 != `instr_ID_stop && temp_instr2_type ) begin
    instr_id_even <= temp_instr_id_1;
    reg_dst_even <= temp_reg_dst_1;
    unit_id_even <= temp_unit_id_1;
    latency_even <= temp_latency_1;
    reg_wr_even <= temp_reg_wr_1;
    imme7_even <= temp_imme7_1;
    imme10_even <= temp_imme10_1;
    imme16_even <= temp_imme16_1;
    imme18_even <= temp_imme18_1;

    ra_addr_even <= temp_ra_addr_1;
    rb_addr_even <= temp_rb_addr_1;
    rc_addr_even <= temp_rc_addr_1;
   end
 end
 else begin
   // feed the decoded instruction to the next stage RF
   if(temp_instr1_type == 1'b1) begin
    instr_id_even <= temp_instr_id_1;
    reg_dst_even <= temp_reg_dst_1;
    unit_id_even <= temp_unit_id_1;
    latency_even <= temp_latency_1;
    reg_wr_even <= temp_reg_wr_1;
    imme7_even <= temp_imme7_1;
    imme10_even <= temp_imme10_1;
    imme16_even <= temp_imme16_1;
    imme18_even <= temp_imme18_1;
    ra_addr_even <= temp_ra_addr_1;
```

```
  rb_addr_even <= temp_rb_addr_1;
  rc_addr_even <= temp_rc_addr_1;
end
else begin
  instr_id_odd <= temp_instr_id_1;
  reg_dst_odd <= temp_reg_dst_1;
  unit_id_odd <= temp_unit_id_1;
  latency_odd <= temp_latency_1;
  reg_wr_odd <= temp_reg_wr_1;
  imme7_odd <= temp_imme7_1;
  imme10_odd <= temp_imme10_1;
  imme16_odd <= temp_imme16_1;
  imme18_odd <= temp_imme18_1;
  ra_addr_odd <= temp_ra_addr_1;
  rb_addr_odd <= temp_rb_addr_1;
  rc_addr_odd <= temp_rc_addr_1;
end

if(temp_instr2_type == 1'b1) begin
  instr_id_even <= temp_instr_id_2;
  reg_dst_even <= temp_reg_dst_2;
  unit_id_even <= temp_unit_id_2;
  latency_even <= temp_latency_2;
  reg_wr_even <= temp_reg_wr_2;
  imme7_even <= temp_imme7_2;
  imme10_even <= temp_imme10_2;
  imme16_even <= temp_imme16_2;
  imme18_even <= temp_imme18_2;

  ra_addr_even <= temp_ra_addr_2;
  rb_addr_even <= temp_rb_addr_2;
  rc_addr_even <= temp_rc_addr_2;

end
else begin
  instr_id_odd <= temp_instr_id_2;
  reg_dst_odd <= temp_reg_dst_2;
```

```
    unit_id_odd <= temp_unit_id_2;
    latency_odd <= temp_latency_2;
    reg_wr_odd <= temp_reg_wr_2;
    imme7_odd <= temp_imme7_2;
    imme10_odd <= temp_imme10_2;
    imme16_odd <= temp_imme16_2;
    imme18_odd <= temp_imme18_2;

    ra_addr_odd <= temp_ra_addr_2;
    rb_addr_odd <= temp_rb_addr_2;
    rc_addr_odd <= temp_rc_addr_2;
   end

 end
 if (temp_unit_id_1 == 3'b111) begin // send signal to Odd pipe that instr1 is branch
  instr1_branch <= 1'b1;
 end
 flush_4stage <= flush_instr2_even;
 flush <= temp_flush;
 PC_pass_out <= PC_pass_in;
end

endmodule
```

Appendix E (Instruction_Decode.v)

```
module Instruction_Decode(
 input [0:31] instruction_in1,
 input [0:31] instruction_in2,

 output reg [0:6] instr_id_1,
 output reg [0:6] reg_dst_1,
 output reg [0:2] unit_id_1,
 output reg [0:3] latency_1,
 output reg reg_wr_1,
 output reg [0:6] imme7_1,
 output reg [0:9] imme10_1,
```

```
  output reg [0:15] imme16_1,
  output reg [0:17] imme18_1,

  output reg [0:6] ra_addr_1,
  output reg [0:6] rb_addr_1,
  output reg [0:6] rc_addr_1,

  output reg [0:6] instr_id_2,
  output reg [0:6] reg_dst_2,
  output reg [0:2] unit_id_2,
  output reg [0:3] latency_2,
  output reg reg_wr_2,
  output reg [0:6] imme7_2,
  output reg [0:9] imme10_2,
  output reg [0:15] imme16_2,
  output reg [0:17] imme18_2,

  output reg [0:6] ra_addr_2,
  output reg [0:6] rb_addr_2,
  output reg [0:6] rc_addr_2,

  output reg instr1_type, // 1 for even, 0 for odd
  output reg instr2_type
);

`include "opcode_package.vh"

localparam RRR = 3'b000;
localparam RR = 3'b001;
localparam RI7 = 3'b010;
localparam RI10 = 3'b011;
localparam RI16 = 3'b100;
localparam RI18 = 3'b101;


wire [0:10] temp_opcode1;
wire [0:10] temp_opcode2;
```

```
assign temp_opcode1 = instruction_in1[0:10];
assign temp_opcode2 = instruction_in2[0:10];


// ------------ for instruction 1 -------------------
always @(*) begin

  // ------------- RRR ------------------
  if (temp_opcode1[0:3] == `op_fma || temp_opcode1[0:3] == `op_fms ||
      temp_opcode1[0:3] == `op_fnms || temp_opcode1[0:3] == `op_mpya ||
      temp_opcode1[0:3] == `op_selb) begin

    if (temp_opcode1[0:3] == `op_fma) begin
      instr_id_1 = `instr_ID_fma;
      unit_id_1 = 3'b011;
      latency_1 = 4'd7;
      reg_wr_1 = 1'b1;
      instr1_type = 1'b1;
    end

    else if (temp_opcode1[0:3] == `op_fms) begin
      instr_id_1 = `instr_ID_fms;
      unit_id_1 = 3'b011;
      latency_1 = 4'd7;
      reg_wr_1 = 1'b1;
      instr1_type = 1'b1;
    end

    else if (temp_opcode1[0:3] == `op_fnms) begin
      instr_id_1 = `instr_ID_fnms;
      unit_id_1 = 3'b011;
      latency_1 = 4'd7;
      reg_wr_1 = 1'b1;
      instr1_type = 1;
    end
```

```
  else if (temp_opcode1[0:3] == `op_mpya) begin
    instr_id_1 = `instr_ID_mpya;
    unit_id_1 = 3'b011;
    latency_1 = 4'd8;
    reg_wr_1 = 1;
    instr1_type = 1;
  end
  else if (temp_opcode1[0:3] == `op_selb) begin
    instr_id_1 = `instr_ID_selb;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
  end

  reg_dst_1 = instruction_in1[4:10];
  ra_addr_1 = instruction_in1[18:24];
  rb_addr_1 = instruction_in1[11:17];
  rc_addr_1 = instruction_in1[25:31];
end

// ------------- R18 ------------------
else if (temp_opcode1[0:6] == `op_ila) begin
  instr_id_1 = `instr_ID_ila;
  reg_dst_1 = instruction_in1[25:31];
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  imme18_1 = instruction_in1[7:24];
  instr1_type = 1'b1;
end

// ------------- RI10 --------------------
else if (temp_opcode1[0:7] == `op_ahi || temp_opcode1[0:7] == `op_ai ||
        temp_opcode1[0:7] == `op_andhi || temp_opcode1[0:7] == `op_andi ||
        temp_opcode1[0:7] == `op_ceqhi || temp_opcode1[0:7] == `op_ceqi ||
        temp_opcode1[0:7] == `op_cgthi || temp_opcode1[0:7] == `op_cgti ||
```

```
      temp_opcode1[0:7] == `op_xorhi || temp_opcode1[0:7] == `op_xori ||
      temp_opcode1[0:7] == `op_orhi || temp_opcode1[0:7] == `op_ori ||
      temp_opcode1[0:7] == `op_sfhi || temp_opcode1[0:7] == `op_sfi ||
      temp_opcode1[0:7] == `op_mpyi || temp_opcode1[0:7] == `op_mpyui ||
     temp_opcode1[0:7] == `op_lqd || temp_opcode1[0:7] == `op_stqd) begin

// i dont think below two are necessary
// instr_format_even = RI10;
// full_instr_even = instruction_in1;


if (temp_opcode1[0:7] == `op_ahi) begin
  instr_id_1 = `instr_ID_ahi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_ai) begin
  instr_id_1 = `instr_ID_ai;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_andhi) begin
  instr_id_1 = `instr_ID_ahi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_andi) begin
  instr_id_1 = `instr_ID_andi;
```

```
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_ceqhi) begin
  instr_id_1 = `instr_ID_ceqhi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_ceqi) begin
  instr_id_1 = `instr_ID_ceqi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_cgthi) begin
  instr_id_1 = `instr_ID_cgthi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_cgti) begin
  instr_id_1 = `instr_ID_cgti;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end
```

```
else if (temp_opcode1[0:7] == `op_xorhi) begin
  instr_id_1 = `instr_ID_xorhi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_xori) begin
  instr_id_1 = `instr_ID_xori;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_orhi) begin
  instr_id_1 = `instr_ID_orhi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_ori) begin
  instr_id_1 = `instr_ID_ori;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_sfhi) begin
  instr_id_1 = `instr_ID_sfhi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
```

```
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_sfi) begin
  instr_id_1 = `instr_ID_sfi;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_mpyi) begin
  instr_id_1 = `instr_ID_mpyi;
  unit_id_1 = 3'd3;
  latency_1 = 4'd8;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_mpyui) begin
  instr_id_1 = `instr_ID_mpyui;
  unit_id_1 = 3'd3;
  latency_1 = 4'd8;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1[0:7] == `op_lqd) begin
  instr_id_1 = `instr_ID_lqd;
  unit_id_1 = 3'b110;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if (temp_opcode1[0:7] == `op_stqd) begin
```

```
    instr_id_1 = `instr_ID_stqd;
    unit_id_1 = 3'b110;
    latency_1 = 4'd7;
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
   end

   imme10_1 = instruction_in1[8:17];
   reg_dst_1 = instruction_in1[25:31];
   ra_addr_1 = instruction_in1[18:24];
   rc_addr_1 = instruction_in1[25:31]; // for rt data

 end

 // ------------- RI16 ----------------------

 else if (temp_opcode1[0:8] == `op_ilh || temp_opcode1[0:8] == `op_ilhu || temp_opcode1[0:8] == `op_il ||
          temp_opcode1[0:8] == `op_iohl || temp_opcode1[0:8] == `op_lqa || temp_opcode1[0:8] == `op_stqa ||
          temp_opcode1[0:8] == `op_bra || temp_opcode1[0:8] == `op_brhnz || temp_opcode1[0:8] == `op_brz ||
          temp_opcode1[0:8] == `op_brnz || temp_opcode1[0:8] == `op_brasl || temp_opcode1[0:8] == `op_brsl ||
          temp_opcode1[0:8] == `op_br || temp_opcode1[0:8] == `op_brhz) begin

   if(temp_opcode1[0:8] == `op_ilh) begin
    instr_id_1 = `instr_ID_ilh;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
   end

   else if(temp_opcode1[0:8] == `op_ilhu) begin
    instr_id_1 = `instr_ID_ilhu;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
   end
```

```
else if(temp_opcode1[0:8] == `op_il) begin
  instr_id_1 = `instr_ID_il;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if(temp_opcode1[0:8] == `op_iohl) begin
  instr_id_1 = `instr_ID_iohl;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if(temp_opcode1[0:8] == `op_lqa) begin
  instr_id_1 = `instr_ID_lqa;
  unit_id_1 = 3'b110;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_stqa) begin
  instr_id_1 = `instr_ID_stqa;
  unit_id_1 = 3'b110;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b0;
  instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_bra) begin
    instr_id_1 = `instr_ID_bra;
    unit_id_1 = 3'b111;
    latency_1 = 4'd2;
```

```
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_brhnz) begin
    instr_id_1 = `instr_ID_brhnz;
    unit_id_1 = 3'b111;
    latency_1 = 4'd2;
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_brz) begin
    instr_id_1 = `instr_ID_brz;
    unit_id_1 = 3'b111;
    latency_1 = 4'd2;
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_brnz) begin
    instr_id_1 = `instr_ID_brnz;
    unit_id_1 = 3'b111;
    latency_1 = 4'd2;
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_brasl) begin
    instr_id_1 = `instr_ID_brasl;
    unit_id_1 = 3'b111;
    latency_1 = 4'd2;
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
end

else if(temp_opcode1[0:8] == `op_brsl) begin
```

```
      instr_id_1 = `instr_ID_brsl;
      unit_id_1 = 3'b111;
      latency_1 = 4'd2;
      reg_wr_1 = 1'b0;
      instr1_type = 1'b0;
    end

    else if(temp_opcode1[0:8] == `op_br) begin
      instr_id_1 = `instr_ID_br;
      unit_id_1 = 3'b111;
      latency_1 = 4'd2;
      reg_wr_1 = 1'b0;
      instr1_type = 1'b0;
    end

    else if(temp_opcode1[0:8] == `op_brhz) begin
      instr_id_1 = `instr_ID_brhz;
      unit_id_1 = 3'b001;
      latency_1 = 4'd2;
      reg_wr_1 = 1'b1;
      instr1_type = 1'b0;
    end


    imme16_1 = instruction_in1[9:24];
    reg_dst_1 = instruction_in1[25:31];
    rc_addr_1 = instruction_in1[25:31]; // for rt data
  end

  // ------------- RI7 ---------------------
  else if (temp_opcode1 == `op_rothi || temp_opcode1 == `op_roti ||
          temp_opcode1 == `op_shlhi || temp_opcode1 == `op_shli ||
          temp_opcode1 == `op_rotqbyi || temp_opcode1 == `op_rotqbii ||
          temp_opcode1 == `op_shlqbii || temp_opcode1 == `op_shlqbyi) begin

    if (temp_opcode1 == `op_rothi) begin
      instr_id_1 = `instr_ID_rothi;
```

```
  unit_id_1 = 3'd2;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_roti) begin
  instr_id_1 = `instr_ID_roti;
  unit_id_1 = 3'd2;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_shlhi) begin
  instr_id_1 = `instr_ID_shlhi;
  unit_id_1 = 3'd2;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_shli) begin
  instr_id_1 = `instr_ID_shli;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_rotqbyi) begin
  instr_id_1 = `instr_ID_rotqbyi;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end
```

```
  else if (temp_opcode1 == `op_rotqbii) begin
    instr_id_1 = `instr_ID_rotqbii;
    unit_id_1 = 3'd5;
    latency_1 = 4'd4;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b0;
  end

  else if (temp_opcode1 == `op_shlqbii) begin
    instr_id_1 = `instr_ID_shlqbii;
    unit_id_1 = 3'd5;
    latency_1 = 4'd4;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b0;
  end

  else if (temp_opcode1 == `op_shlqbyi) begin
    instr_id_1 = `instr_ID_shlqbyi;
    unit_id_1 = 3'd5;
    latency_1 = 4'd4;
    instr1_type = 1'b0;
  end

  imme7_1 = instruction_in1[11:17];
  reg_dst_1 = instruction_in1[25:31];
  ra_addr_1 = instruction_in1[18:24];
  rc_addr_1 = instruction_in1[25:31];
end

// ------------- RR ---------------------
else if (temp_opcode1 == `op_addx || temp_opcode1 == `op_ah || temp_opcode1 == `op_a ||
         temp_opcode1 == `op_and || temp_opcode1 == `op_bg || temp_opcode1 == `op_bgx ||
         temp_opcode1 == `op_cg || temp_opcode1 == `op_cgx || temp_opcode1 == `op_ceqh ||
         temp_opcode1 == `op_ceq || temp_opcode1 == `op_cgth || temp_opcode1 == `op_cgt ||
         temp_opcode1 == `op_clz || temp_opcode1 == `op_eqv || temp_opcode1 == `op_xor ||
         temp_opcode1 == `op_nand || temp_opcode1 == `op_nor || temp_opcode1 == `op_or ||
```

```
        temp_opcode1 == `op_sfx || temp_opcode1 == `op_sfh || temp_opcode1 == `op_sf ||
        temp_opcode1 == `op_roth || temp_opcode1 == `op_rot || temp_opcode1 == `op_shlh ||
        temp_opcode1 == `op_shl || temp_opcode1 == `op_fa || temp_opcode1 == `op_fm ||
        temp_opcode1 == `op_fs || temp_opcode1 == `op_mpy || temp_opcode1 == `op_mpyu ||
        temp_opcode1 == `op_mpyh || temp_opcode1 == `op_cntb || temp_opcode1 == `op_absdb ||
        temp_opcode1 == `op_sumb || temp_opcode1 == `op_avgb || temp_opcode1 == `op_rotqby ||
        temp_opcode1 == `op_rotqbi || temp_opcode1 == `op_shlqbi || temp_opcode1 == `op_shlqby ||
        temp_opcode1 == `op_nop || temp_opcode1 == `op_lnop) begin

if (temp_opcode1 == `op_addx) begin
  instr_id_1 = `instr_ID_addx;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_ah) begin
  instr_id_1 = `instr_ID_ah;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_a) begin
  instr_id_1 = `instr_ID_a;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_and) begin
  instr_id_1 = `instr_ID_and;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
```

```
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_bg) begin
  instr_id_1 = `instr_ID_bg;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_bgx) begin
  instr_id_1 = `instr_ID_bgx;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_cg) begin
  instr_id_1 = `instr_ID_cg;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b0;
  instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_cgx) begin
  instr_id_1 = `instr_ID_cgx;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_ceqh) begin
```

```
    instr_id_1 = `instr_ID_ceqh;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_ceq) begin
    instr_id_1 = `instr_ID_ceq;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_cgth) begin
    instr_id_1 = `instr_ID_cgth;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_cgt) begin
    instr_id_1 = `instr_ID_cgt;
    unit_id_1 = 3'b001;
    latency_1 = 4'd3;
    reg_wr_1 = 1'b1;
    instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_clz) begin
  instr_id_1 = `instr_ID_clz;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
```

```
end

else if (temp_opcode1 == `op_eqv) begin
  instr_id_1 = `instr_ID_eqv;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_xor) begin
  instr_id_1 = `instr_ID_xor;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_nand) begin
  instr_id_1 = `instr_ID_nand;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_nor) begin
  instr_id_1 = `instr_ID_nor;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_or) begin
  instr_id_1 = `instr_ID_or;
  unit_id_1 = 3'b001;
```

```
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_sfx) begin
  instr_id_1 = `instr_ID_sfx;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_sfh) begin
  instr_id_1 = `instr_ID_sfh;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_sf) begin
  instr_id_1 = `instr_ID_sf;
  unit_id_1 = 3'b001;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_roth) begin
  instr_id_1 = `instr_ID_roth;
  unit_id_1 = 3'b010;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end
```

```
else if (temp_opcode1 == `op_rot) begin
  instr_id_1 = `instr_ID_rot;
  unit_id_1 = 3'b010;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_shlh) begin
  instr_id_1 = `instr_ID_shlh;
  unit_id_1 = 3'b010;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_shl) begin
  instr_id_1 = `instr_ID_shl;
  unit_id_1 = 3'b010;
  latency_1 = 4'd3;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_fa) begin
  instr_id_1 = `instr_ID_fa;
  unit_id_1 = 3'b011;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_fm) begin
  instr_id_1 = `instr_ID_fm;
  unit_id_1 = 3'b011;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b1;
```

```
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_fs) begin
  instr_id_1 = `instr_ID_fs;
  unit_id_1 = 3'b011;
  latency_1 = 4'd7;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_mpy) begin
  instr_id_1 = `instr_ID_mpy;
  unit_id_1 = 3'b011;
  latency_1 = 4'd8;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_mpyu) begin
  instr_id_1 = `instr_ID_mpyu;
  unit_id_1 = 3'b011;
  latency_1 = 4'd8;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_mpyh) begin
  instr_id_1 = `instr_ID_mpyh;
  unit_id_1 = 3'b011;
  latency_1 = 4'd8;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_cntb) begin
  instr_id_1 = `instr_ID_cntb;
```

```
  unit_id_1 = 3'd4;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_absdb) begin
  instr_id_1 = `instr_ID_absdb;
  unit_id_1 = 3'd4;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_sumb) begin
  instr_id_1 = `instr_ID_sumb;
  unit_id_1 = 3'd4;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_avgb) begin
  instr_id_1 = `instr_ID_avgb;
  unit_id_1 = 3'd4;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_rotqby) begin
  instr_id_1 = `instr_ID_rotqby;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end
```

```
else if (temp_opcode1 == `op_rotqbi) begin
  instr_id_1 = `instr_ID_rotqbi;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_shlqbi) begin
  instr_id_1 = `instr_ID_shlqbi;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_shlqby) begin
  instr_id_1 = `instr_ID_shlqby;
  unit_id_1 = 3'd5;
  latency_1 = 4'd4;
  reg_wr_1 = 1'b1;
  instr1_type = 1'b0;
end

else if (temp_opcode1 == `op_nop) begin
  instr_id_1 = `instr_ID_nop;
  unit_id_1 = 3'b000;
  latency_1 = 4'd0;
  reg_wr_1 = 1'b0;
  instr1_type = 1'b1;
end

else if (temp_opcode1 == `op_lnop) begin
  instr_id_1 = `instr_ID_lnop;
  unit_id_1 = 3'b000;
  latency_1 = 4'd0;
```

```
    reg_wr_1 = 1'b0;
    instr1_type = 1'b0;
   end


   reg_dst_1 = instruction_in1[25:31];
   ra_addr_1 = instruction_in1[18:24];
   rb_addr_1 = instruction_in1[11:17];
   rc_addr_1 = instruction_in1[25:31]; // for rt data
  end
  else if (temp_opcode1 == `op_stop) begin
   instr_id_1 = `instr_ID_stop;
   unit_id_1 = 3'b000;
   latency_1 = 4'd0;
   reg_wr_1 = 1'b0;
   instr1_type = 1'b1;
  end
end

// ------------------------------------------------
// ------------ for instruction 2 -----------------
// ------------------------------------------------
always @(*) begin

  // ------------ RRR -----------------
  if (temp_opcode2[0:3] == `op_fma || temp_opcode2[0:3] == `op_fms ||
     temp_opcode2[0:3] == `op_fnms || temp_opcode2[0:3] == `op_mpya ||
     temp_opcode2[0:3] == `op_selb) begin

   if (temp_opcode2[0:3] == `op_fma) begin
    instr_id_2 = `instr_ID_fma;
    unit_id_2 = 3'b011;
    latency_2 = 4'd7;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
   end
```

```
  else if (temp_opcode2[0:3] == `op_fms) begin
    instr_id_2 = `instr_ID_fms;
    unit_id_2 = 3'b011;
    latency_2 = 4'd7;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2[0:3] == `op_fnms) begin
    instr_id_2 = `instr_ID_fnms;
    unit_id_2 = 3'b011;
    latency_2 = 4'd7;
    reg_wr_2 = 1'b1;
    instr2_type = 1;
  end

  else if (temp_opcode2[0:3] == `op_mpya) begin
    instr_id_2 = `instr_ID_mpya;
    unit_id_2 = 3'b011;
    latency_2 = 4'd8;
    reg_wr_2 = 1;
    instr2_type = 1;
  end
  else if (temp_opcode2[0:3] == `op_selb) begin
    instr_id_2 = `instr_ID_selb;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  reg_dst_2 = instruction_in2[4:10];
  ra_addr_2 = instruction_in2[18:24];
  rb_addr_2 = instruction_in2[11:17];
  rc_addr_2 = instruction_in2[25:31];
end
// ------------- R18 -----------------
```

```
else if (temp_opcode2[0:6] == `op_ila) begin
  instr_id_2 = `instr_ID_ila;
  reg_dst_2 = instruction_in2[25:31];
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  imme18_2 = instruction_in2[7:24];
  instr2_type = 1'b1;
end

// ---------- RI10 ----------
else if (temp_opcode2[0:7] == `op_ahi || temp_opcode2[0:7] == `op_ai ||
        temp_opcode2[0:7] == `op_andhi || temp_opcode2[0:7] == `op_andi ||
        temp_opcode2[0:7] == `op_ceqhi || temp_opcode2[0:7] == `op_ceqi ||
        temp_opcode2[0:7] == `op_cgthi || temp_opcode2[0:7] == `op_cgti ||
        temp_opcode2[0:7] == `op_xorhi || temp_opcode2[0:7] == `op_xori ||
        temp_opcode2[0:7] == `op_orhi || temp_opcode2[0:7] == `op_ori ||
        temp_opcode2[0:7] == `op_sfhi || temp_opcode2[0:7] == `op_sfi ||
        temp_opcode2[0:7] == `op_mpyi || temp_opcode2[0:7] == `op_mpyui ||
        temp_opcode2[0:7] == `op_lqd || temp_opcode2[0:7] == `op_stqd) begin

  if (temp_opcode2[0:7] == `op_ahi) begin
    instr_id_2 = `instr_ID_ahi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_ai) begin
    instr_id_2 = `instr_ID_ai;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_andhi) begin
    instr_id_2 = `instr_ID_ahi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
```

```
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_andi) begin
    instr_id_2 = `instr_ID_andi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_ceqhi) begin
    instr_id_2 = `instr_ID_ceqhi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_ceqi) begin
    instr_id_2 = `instr_ID_ceqi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_cgthi) begin
    instr_id_2 = `instr_ID_cgthi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_cgti) begin
    instr_id_2 = `instr_ID_cgti;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_xorhi) begin
    instr_id_2 = `instr_ID_xorhi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
```

```
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_xori) begin
    instr_id_2 = `instr_ID_xori;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_orhi) begin
    instr_id_2 = `instr_ID_orhi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_ori) begin
    instr_id_2 = `instr_ID_ori;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_sfhi) begin
    instr_id_2 = `instr_ID_sfhi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_sfi) begin
    instr_id_2 = `instr_ID_sfi;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end else if (temp_opcode2[0:7] == `op_mpyi) begin
    instr_id_2 = `instr_ID_mpyi;
    unit_id_2 = 3'd3;
    latency_2 = 4'd8;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
```

```
    end else if (temp_opcode2[0:7] == `op_mpyui) begin
      instr_id_2 = `instr_ID_mpyui;
      unit_id_2 = 3'd3;
      latency_2 = 4'd8;
      reg_wr_2 = 1'b1;
      instr2_type = 1'b1;
    end else if (temp_opcode2[0:7] == `op_lqd) begin
      instr_id_2 = `instr_ID_lqd;
      unit_id_2 = 3'b110;
      latency_2 = 4'd7;
      reg_wr_2 = 1'b1;
      instr2_type = 1'b0;
    end else if (temp_opcode2[0:7] == `op_stqd) begin
      instr_id_2 = `instr_ID_stqd;
      unit_id_2 = 3'b110;
      latency_2 = 4'd7;
      reg_wr_2 = 1'b0;
      instr2_type = 1'b0;
    end

    imme10_2 = instruction_in2[8:17];
    reg_dst_2 = instruction_in2[25:31];
    ra_addr_2 = instruction_in2[18:24];
    rc_addr_2 = instruction_in2[25:31];
  end

  // ------------- RI16 ---------------------

else if (temp_opcode2[0:8] == `op_ilh || temp_opcode2[0:8] == `op_ilhu || temp_opcode2[0:8] == `op_il ||
        temp_opcode2[0:8] == `op_iohl || temp_opcode2[0:8] == `op_lqa || temp_opcode2[0:8] == `op_stqa ||
        temp_opcode2[0:8] == `op_bra || temp_opcode2[0:8] == `op_brhnz || temp_opcode2[0:8] == `op_brz ||
        temp_opcode2[0:8] == `op_brnz || temp_opcode2[0:8] == `op_brasl || temp_opcode2[0:8] == `op_brsl ||
        temp_opcode2[0:8] == `op_br || temp_opcode2[0:8] == `op_brhz) begin

  if(temp_opcode2[0:8] == `op_ilh) begin
    instr_id_2 = `instr_ID_ilh;
    unit_id_2 = 3'b001;
```

```
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if(temp_opcode2[0:8] == `op_ilhu) begin
  instr_id_2 = `instr_ID_ilhu;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if(temp_opcode2[0:8] == `op_il) begin
  instr_id_2 = `instr_ID_il;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if(temp_opcode2[0:8] == `op_iohl) begin
  instr_id_2 = `instr_ID_iohl;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if(temp_opcode2[0:8] == `op_lqa) begin
  instr_id_2 = `instr_ID_lqa;
  unit_id_2 = 3'b110;
  latency_2 = 4'd7;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end
```

```
else if(temp_opcode2[0:8] == `op_stqa) begin
  instr_id_2 = `instr_ID_stqa;
  unit_id_2 = 3'b110;
  latency_2 = 4'd7;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_bra) begin
  instr_id_2 = `instr_ID_bra;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brhnz) begin
  instr_id_2 = `instr_ID_brhnz;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brz) begin
  instr_id_2 = `instr_ID_brz;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brnz) begin
  instr_id_2 = `instr_ID_brnz;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
```

```
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brasl) begin
  instr_id_2 = `instr_ID_brasl;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brsl) begin
  instr_id_2 = `instr_ID_brsl;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_br) begin
  instr_id_2 = `instr_ID_br;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if(temp_opcode2[0:8] == `op_brhz) begin
  instr_id_2 = `instr_ID_brhz;
  unit_id_2 = 3'b111;
  latency_2 = 4'd2;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

imme16_2 = instruction_in2[9:24];
reg_dst_2 = instruction_in2[25:31];
```

```
  rc_addr_2 = instruction_in2[25:31]; // for rt data
end

// ------------- RI7 ----------------------
else if (temp_opcode2 == `op_rothi || temp_opcode2 == `op_roti ||
        temp_opcode2 == `op_shlhi || temp_opcode2 == `op_shli ||
        temp_opcode2 == `op_rotqbyi || temp_opcode2 == `op_rotqbii ||
        temp_opcode2 == `op_shlqbii || temp_opcode2 == `op_shlqbyi) begin

  if (temp_opcode2 == `op_rothi) begin
    instr_id_2 = `instr_ID_rothi;
    unit_id_2 = 3'd2;
    latency_2 = 4'd4;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_roti) begin
    instr_id_2 = `instr_ID_roti;
    unit_id_2 = 3'd2;
    latency_2 = 4'd4;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_shlhi) begin
    instr_id_2 = `instr_ID_shlhi;
    unit_id_2 = 3'd2;
    latency_2 = 4'd4;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_shli) begin
    instr_id_2 = `instr_ID_shli;
    unit_id_2 = 3'd5;
    latency_2 = 4'd4;
```

```
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_rotqbyi) begin
  instr_id_2 = `instr_ID_rotqbyi;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_rotqbii) begin
  instr_id_2 = `instr_ID_rotqbii;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_shlqbii) begin
  instr_id_2 = `instr_ID_shlqbii;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_shlqbyi) begin
  instr_id_2 = `instr_ID_shlqbyi;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

imme7_2 = instruction_in2[11:17];
```

```
 reg_dst_2 = instruction_in2[25:31];
 ra_addr_2 = instruction_in2[18:24];
 rc_addr_2 = instruction_in2[25:31]; // for rt data
end


 // ------------- RR ---------------------
 else if (temp_opcode2 == `op_addx || temp_opcode2 == `op_ah || temp_opcode2 == `op_a ||
         temp_opcode2 == `op_and || temp_opcode2 == `op_bg || temp_opcode2 == `op_bgx ||
         temp_opcode2 == `op_cg || temp_opcode2 == `op_cgx || temp_opcode2 == `op_ceqh ||
         temp_opcode2 == `op_ceq || temp_opcode2 == `op_cgth || temp_opcode2 == `op_cgt ||
         temp_opcode2 == `op_clz || temp_opcode2 == `op_eqv || temp_opcode2 == `op_xor ||
         temp_opcode2 == `op_nand || temp_opcode2 == `op_nor || temp_opcode2 == `op_or ||
         temp_opcode2 == `op_sfx || temp_opcode2 == `op_sfh || temp_opcode2 == `op_sf ||
         temp_opcode2 == `op_roth || temp_opcode2 == `op_rot || temp_opcode2 == `op_shlh ||
         temp_opcode2 == `op_shl || temp_opcode2 == `op_fa || temp_opcode2 == `op_fm ||
         temp_opcode2 == `op_fs || temp_opcode2 == `op_mpy || temp_opcode2 == `op_mpyu ||
         temp_opcode2 == `op_mpyh || temp_opcode2 == `op_cntb || temp_opcode2 == `op_absdb ||
         temp_opcode2 == `op_sumb || temp_opcode2 == `op_avgb || temp_opcode2 == `op_rotqby ||
         temp_opcode2 == `op_rotqbi || temp_opcode2 == `op_shlqbi || temp_opcode2 == `op_shlqby ||
         temp_opcode2 == `op_nop || temp_opcode2 == `op_lnop) begin

  if (temp_opcode2 == `op_addx) begin
    instr_id_2 = `instr_ID_addx;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_ah) begin
    instr_id_2 = `instr_ID_ah;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end
```

```
else if (temp_opcode2 == `op_a) begin
  instr_id_2 = `instr_ID_a;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_and) begin
  instr_id_2 = `instr_ID_and;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_bg) begin
  instr_id_2 = `instr_ID_bg;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_bgx) begin
  instr_id_2 = `instr_ID_bgx;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_cg) begin
  instr_id_2 = `instr_ID_cg;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
```

```
  reg_wr_2 = 1'b0;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_cgx) begin
  instr_id_2 = `instr_ID_cgx;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_ceqh) begin
  instr_id_2 = `instr_ID_ceqh;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_ceq) begin
    instr_id_2 = `instr_ID_ceq;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_cgth) begin
    instr_id_2 = `instr_ID_cgth;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_cgt) begin
```

```
    instr_id_2 = `instr_ID_cgt;
    unit_id_2 = 3'b001;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_clz) begin
  instr_id_2 = `instr_ID_clz;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_eqv) begin
  instr_id_2 = `instr_ID_eqv;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_xor) begin
  instr_id_2 = `instr_ID_xor;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_nand) begin
  instr_id_2 = `instr_ID_nand;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
```

```
end

else if (temp_opcode2 == `op_nor) begin
  instr_id_2 = `instr_ID_nor;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_or) begin
  instr_id_2 = `instr_ID_or;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_sfx) begin
  instr_id_2 = `instr_ID_sfx;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_sfh) begin
  instr_id_2 = `instr_ID_sfh;
  unit_id_2 = 3'b001;
  latency_2 = 4'd3;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_sf) begin
  instr_id_2 = `instr_ID_sf;
  unit_id_2 = 3'b001;
```

```
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_roth) begin
    instr_id_2 = `instr_ID_roth;
    unit_id_2 = 3'b010;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_rot) begin
    instr_id_2 = `instr_ID_rot;
    unit_id_2 = 3'b010;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end

  else if (temp_opcode2 == `op_shlh) begin
    instr_id_2 = `instr_ID_shlh;
    unit_id_2 = 3'b010;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
end

  else if (temp_opcode2 == `op_shl) begin
    instr_id_2 = `instr_ID_shl;
    unit_id_2 = 3'b010;
    latency_2 = 4'd3;
    reg_wr_2 = 1'b1;
    instr2_type = 1'b1;
  end
```

```
else if (temp_opcode2 == `op_fa) begin
  instr_id_2 = `instr_ID_fa;
  unit_id_2 = 3'b011;
  latency_2 = 4'd7;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_fm) begin
  instr_id_2 = `instr_ID_fm;
  unit_id_2 = 3'b011;
  latency_2 = 4'd7;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_fs) begin
  instr_id_2 = `instr_ID_fs;
  unit_id_2 = 3'b011;
  latency_2 = 4'd7;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_mpy) begin
  instr_id_2 = `instr_ID_mpy;
  unit_id_2 = 3'b011;
  latency_2 = 4'd8;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_mpyu) begin
  instr_id_2 = `instr_ID_mpyu;
  unit_id_2 = 3'b011;
  latency_2 = 4'd8;
  reg_wr_2 = 1'b1;
```

```
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_mpyh) begin
  instr_id_2 = `instr_ID_mpyh;
  unit_id_2 = 3'b011;
  latency_2 = 4'd8;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_cntb) begin
  instr_id_2 = `instr_ID_cntb;
  unit_id_2 = 3'd4;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_absdb) begin
  instr_id_2 = `instr_ID_absdb;
  unit_id_2 = 3'd4;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_sumb) begin
  instr_id_2 = `instr_ID_sumb;
  unit_id_2 = 3'd4;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_avgb) begin
  instr_id_2 = `instr_ID_avgb;
```

```
  unit_id_2 = 3'd4;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b1;
end

else if (temp_opcode2 == `op_rotqby) begin
  instr_id_2 = `instr_ID_rotqby;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_rotqbi) begin
  instr_id_2 = `instr_ID_rotqbi;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_shlqbi) begin
  instr_id_2 = `instr_ID_shlqbi;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end

else if (temp_opcode2 == `op_shlqby) begin
  instr_id_2 = `instr_ID_shlqby;
  unit_id_2 = 3'd5;
  latency_2 = 4'd4;
  reg_wr_2 = 1'b1;
  instr2_type = 1'b0;
end
```

```
    else if (temp_opcode2 == `op_nop) begin
      instr_id_2 = `instr_ID_nop;
      unit_id_2 = 3'b000;
      latency_2 = 4'd0;
      reg_wr_2 = 1'b0;
      instr2_type = 1'b1;
    end

    else if (temp_opcode2 == `op_lnop) begin
      instr_id_2 = `instr_ID_lnop;
      unit_id_2 = 3'b000;
      latency_2 = 4'd0;
      reg_wr_2 = 1'b0;
      instr2_type = 1'b0;
    end

    reg_dst_2 = instruction_in2[25:31];
    ra_addr_2 = instruction_in2[18:24];
    rb_addr_2 = instruction_in2[11:17];
    rc_addr_2 = instruction_in2[25:31]; // for rt data
  end
  else if (temp_opcode2 == `op_stop) begin
    instr_id_2 = `instr_ID_stop;
    unit_id_2 = 3'b000;
    latency_2 = 4'd0;
    reg_wr_2 = 1'b0;
    instr2_type = 1'b0;
  end
end

endmodule
```

Appendix F (Hazard_Unit.v)

```
module Hazard_Unit(
  input [0:1] instr_dependent_protocol,
```

```
input [0:2] data_dependent_protocol,
input instr1_type,
input instr2_type,
input is_branch,
input branch_taken,

input [0:6] instr_id_even,
input [0:6] reg_dst_even,
input [0:6] ra_addr_even,
input [0:6] rb_addr_even,
input [0:6] rc_addr_even,
input [0:6] instr_id_odd,
input [0:6] reg_dst_odd,
input [0:6] ra_addr_odd,
input [0:6] rb_addr_odd,
input [0:6] rc_addr_odd,

input [0:6] ID_reg_dst_even,
input ID_reg_wr_even,
input [0:6] RF_reg_dst_even, // 얘가 사실상 stage 1 이 가지고 잇는 정보야
input RF_reg_wr_even,
input [0:3] RF_latency_even,
input [0:142] packed_2stage_even,
input [0:142] packed_3stage_even,
input [0:142] packed_4stage_even,
input [0:142] packed_5stage_even,
input [0:142] packed_6stage_even,

input [0:6] ID_reg_dst_odd,
input ID_reg_wr_odd,
input [0:6] RF_reg_dst_odd,
input RF_reg_wr_odd,
input [0:3] RF_latency_odd,
input [0:142] packed_2stage_odd,
input [0:142] packed_3stage_odd,
input [0:142] packed_4stage_odd,
input [0:142] packed_5stage_odd,
```

```
  input [0:142] packed_6stage_odd,

  output reg stall,
  output reg dependent_stall,
  output reg flush
);

`include "opcode_package.vh"

wire instr_same;

assign instr_same = (instr1_type == instr2_type) ? 1'b1 : 1'b0;
// detect when two instructions must not issue together
wire both_valid = (instr_id_even != `instr_ID_nop
                && instr_id_odd != `instr_ID_lnop
                && instr_id_even != `instr_ID_stop
                && instr_id_odd != `instr_ID_stop);

wire instr_needs_rc_even =
  (instr_id_even == `instr_ID_addx || instr_id_even == `instr_ID_bg || instr_id_even ==`instr_ID_bgx ||
    instr_id_even == `instr_ID_cgx || instr_id_even == `instr_ID_iohl || instr_id_even == `instr_ID_sfx ||
    instr_id_even == `instr_ID_fma || instr_id_even == `instr_ID_fms || instr_id_even == `instr_ID_fnms ||
    instr_id_even == `instr_ID_mpya || instr_id_even == `instr_ID_selb || instr_id_even == `instr_ID_mpyh);

wire instr_needs_rc_odd =
  (instr_id_even == `instr_ID_brhnz || instr_id_even == `instr_ID_brz || instr_id_even == `instr_ID_brnz ||
    instr_id_even == `instr_ID_brhz);

// dependent stall only when both valid and protocols allow
always @(*) begin
  dependent_stall = 1'b0;
  if (instr_dependent_protocol==2'b00
      && data_dependent_protocol==2'b00
      && both_valid
      && (instr1_type==instr2_type
          || reg_dst_even==reg_dst_odd)) begin
    dependent_stall = 1'b1;
```

```
  end
end

always @(*) begin
  stall = 1'b0;
  flush = 1'b0;

  if (branch_taken == 1'b1 && is_branch == 1'b1) begin // for now it's always predict-not-taken, later should be
replaced by 2bit branch prediction signal
    flush = 1'b1;
  end

  else if ((ra_addr_even == ID_reg_dst_even || rb_addr_even == ID_reg_dst_even || (instr_needs_rc_even &&
rc_addr_even == ID_reg_dst_even)) && ID_reg_wr_even) begin
    stall = 1'b1;
  end
  else if ((ra_addr_even == RF_reg_dst_even || rb_addr_even == RF_reg_dst_even || (instr_needs_rc_even &&
rc_addr_even == RF_reg_dst_even)) && RF_reg_wr_even) begin
    if(RF_latency_even > 4'd2) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_2stage_even[131:137] || rb_addr_even == packed_2stage_even[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_2stage_even[131:137])) && packed_2stage_even[142]) begin
    if(packed_2stage_even[138:141] > 4'd3) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_3stage_even[131:137] || rb_addr_even == packed_3stage_even[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_3stage_even[131:137])) && packed_3stage_even[142]) begin
    if(packed_3stage_even[138:141] > 4'd4) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_4stage_even[131:137] || rb_addr_even == packed_4stage_even[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_4stage_even[131:137])) && packed_4stage_even[142]) begin
    if(packed_4stage_even[138:141] > 4'd5) begin
```

```
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_5stage_even[131:137] || rb_addr_even == packed_5stage_even[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_5stage_even[131:137])) && packed_5stage_even[142]) begin
    if(packed_5stage_even[138:141] > 4'd6) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_6stage_even[131:137] || rb_addr_even == packed_6stage_even[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_6stage_even[131:137])) && packed_6stage_even[142]) begin
    if(packed_6stage_even[138:141] > 4'd7) begin
      stall = 1'b1;
    end
  end

  else if ((ra_addr_odd == ID_reg_dst_odd || rb_addr_odd == ID_reg_dst_odd || (instr_needs_rc_odd && rc_addr_odd
== ID_reg_dst_odd)) && ID_reg_wr_odd) begin
    stall = 1'b1;
  end
  else if ((ra_addr_odd == RF_reg_dst_odd || rb_addr_odd == RF_reg_dst_odd || (instr_needs_rc_odd && rc_addr_odd
== RF_reg_dst_odd)) && RF_reg_wr_odd) begin
    if(RF_latency_odd > 4'd2) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_2stage_odd[131:137] || rb_addr_odd == packed_2stage_odd[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_2stage_even[131:137])) && packed_2stage_odd[142]) begin
    if(packed_2stage_odd[138:141] > 4'd3) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_3stage_odd[131:137] || rb_addr_odd == packed_3stage_odd[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_3stage_even[131:137])) && packed_3stage_odd[142]) begin
    if(packed_3stage_odd[138:141] > 4'd4) begin
      stall = 1'b1;
    end
```

```
  end
  else if ((ra_addr_odd == packed_4stage_odd[131:137] || rb_addr_odd == packed_4stage_odd[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_4stage_even[131:137])) && packed_4stage_odd[142]) begin
    if(packed_4stage_odd[138:141] > 4'd5) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_5stage_odd[131:137] || rb_addr_odd == packed_5stage_odd[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_5stage_even[131:137])) && packed_5stage_odd[142]) begin
    if(packed_5stage_odd[138:141] > 4'd6) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_6stage_odd[131:137] || rb_addr_odd == packed_6stage_odd[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_6stage_even[131:137])) && packed_6stage_odd[142]) begin
    if(packed_6stage_odd[138:141] > 4'd7) begin
      stall = 1'b1;
    end
  end

  // cross stage hazard detection
  else if ((ra_addr_even == ID_reg_dst_odd || rb_addr_even == ID_reg_dst_odd || (instr_needs_rc_even &&
rc_addr_even == ID_reg_dst_odd)) && ID_reg_wr_odd) begin
    stall = 1'b1;
  end
  else if ((ra_addr_even == RF_reg_dst_odd || rb_addr_even == RF_reg_dst_odd || (instr_needs_rc_even &&
rc_addr_even == RF_reg_dst_odd)) && RF_reg_wr_odd) begin
    if(RF_latency_odd > 4'd2) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_2stage_odd[131:137] || rb_addr_even == packed_2stage_odd[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_2stage_odd[131:137])) && packed_2stage_odd[142]) begin
    if(packed_2stage_odd[138:141] > 4'd3) begin
      stall = 1'b1;
    end
  end
```

```
  else if ((ra_addr_even == packed_3stage_odd[131:137] || rb_addr_even == packed_3stage_odd[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_3stage_odd[131:137])) && packed_3stage_odd[142]) begin
    if(packed_3stage_odd[138:141] > 4'd4) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_4stage_odd[131:137] || rb_addr_even == packed_4stage_odd[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_4stage_odd[131:137])) && packed_4stage_odd[142]) begin
    if(packed_4stage_odd[138:141] > 4'd5) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_5stage_odd[131:137] || rb_addr_even == packed_5stage_odd[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_5stage_odd[131:137])) && packed_5stage_odd[142]) begin
    if(packed_5stage_odd[138:141] > 4'd6) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_even == packed_6stage_odd[131:137] || rb_addr_even == packed_6stage_odd[131:137] ||
(instr_needs_rc_even && rc_addr_even == packed_6stage_odd[131:137])) && packed_6stage_odd[142]) begin
    if(packed_6stage_odd[138:141] > 4'd7) begin
      stall = 1'b1;
    end
  end

  else if ((ra_addr_odd == ID_reg_dst_even || rb_addr_odd == ID_reg_dst_even  || (instr_needs_rc_odd &&
rc_addr_odd == ID_reg_dst_even)) && ID_reg_wr_even) begin
    stall = 1'b1;
  end
  else if ((ra_addr_odd == RF_reg_dst_even || rb_addr_odd == RF_reg_dst_even || (instr_needs_rc_odd &&
rc_addr_odd == RF_reg_dst_even)) && RF_reg_wr_even) begin
    if(RF_latency_even > 4'd2) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_2stage_even[131:137] || rb_addr_odd == packed_2stage_even[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_2stage_even[131:137])) && packed_2stage_even[142]) begin
```

```
    if(packed_2stage_even[138:141] > 4'd3) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_3stage_even[131:137] || rb_addr_odd == packed_3stage_even[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_3stage_even[131:137])) && packed_3stage_even[142]) begin
    if(packed_3stage_even[138:141] > 4'd4) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_4stage_even[131:137] || rb_addr_odd == packed_4stage_even[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_4stage_even[131:137])) && packed_4stage_even[142]) begin
    if(packed_4stage_even[138:141] > 4'd5) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_5stage_even[131:137] || rb_addr_odd == packed_5stage_even[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_5stage_even[131:137])) && packed_5stage_even[142]) begin
    if(packed_5stage_even[138:141] > 4'd6) begin
      stall = 1'b1;
    end
  end
  else if ((ra_addr_odd == packed_6stage_even[131:137] || rb_addr_odd == packed_6stage_even[131:137] ||
(instr_needs_rc_odd && rc_addr_odd == packed_6stage_even[131:137])) && packed_6stage_even[142]) begin
    if(packed_6stage_even[138:141] > 4'd7) begin
      stall = 1'b1;
    end
  end
  else begin
    stall = 1'b0;
    // dependent_stall = 1'b0;
    flush = 1'b0;
  end
end

endmodule
```

Appendix G (RF_FU_wrapper.v)

```
module RF_FU_wrapper(
  input clk,
  input rst,
  input flush,
  input [0:8] PC_pass_in,
  input instr1_branch,

  // inputs for instruction
  input [0:6] instr_id_even,
  input [0:6] reg_dst_even,
  input [0:2] unit_id_even,
  input [0:3] latency_even,
  input reg_wr_even,
  input [0:6] imme7_even,
  input [0:9] imme10_even,
  input [0:15] imme16_even,
  input [0:17] imme18_even,

  input [0:6] instr_id_odd,
  input [0:6] reg_dst_odd,
  input [0:2] unit_id_odd,
  input [0:3] latency_odd,
  input reg_wr_odd,
  input [0:6] imme7_odd,
  input [0:9] imme10_odd,
  input [0:15] imme16_odd,
  input [0:17] imme18_odd,

  // inputs for register file
  input [0:6] ra_addr_even,   // these three addresses are also used for forwarding unit
  input [0:6] rb_addr_even,
  input [0:6] rc_addr_even,

  input [0:6] ra_addr_odd,
  input [0:6] rb_addr_odd,
```

```
input [0:6] rc_addr_odd,

input reg_write_en_1,
input reg_write_en_2,

input [0:6] reg_write_addr_1,
input [0:6] reg_write_addr_2,
input [0:127] reg_write_data_1,
input [0:127] reg_write_data_2,

// inputs for forwarding unit
input [0:142] packed_2stage_even,
input [0:142] packed_3stage_even,
input [0:142] packed_4stage_even,
input [0:142] packed_5stage_even,
input [0:142] packed_6stage_even,
input [0:142] packed_7stage_even,

input [0:142] packed_2stage_odd,
input [0:142] packed_3stage_odd,
input [0:142] packed_4stage_odd,
input [0:142] packed_5stage_odd,
input [0:142] packed_6stage_odd,
input [0:142] packed_7stage_odd,

// OUTPUT
output reg [0:6] out_instr_id_even,
output reg [0:6] out_reg_dst_even,
output reg [0:2] out_unit_id_even,
output reg [0:3] out_latency_even,
output reg out_reg_wr_even,
output reg [0:6] out_imme7_even,
output reg [0:9] out_imme10_even,
output reg [0:15] out_imme16_even,
output reg [0:17] out_imme18_even,

output reg [0:6] out_instr_id_odd,
```

```
  output reg [0:6] out_reg_dst_odd,
  output reg [0:2] out_unit_id_odd,
  output reg [0:3] out_latency_odd,
  output reg out_reg_wr_odd,
  output reg [0:6] out_imme7_odd,
  output reg [0:9] out_imme10_odd,
  output reg [0:15] out_imme16_odd,
  output reg [0:17] out_imme18_odd,

  output reg [0:127] ra_data_even,
  output reg [0:127] rb_data_even,
  output reg [0:127] rc_data_even,

  output reg [0:127] ra_data_odd,
  output reg [0:127] rb_data_odd,
  output reg [0:127] rc_data_odd,

  output reg [0:8] PC_pass_out,
  output reg instr1_branch_out,

  // Preload RF. Verification purpose only
  input preload_en,
  input [0:6] preload_addr,
  input [0:127] preload_values

);
wire [0:127] regfile_out_data_1;
wire [0:127] regfile_out_data_2;
wire [0:127] regfile_out_data_3;
wire [0:127] regfile_out_data_4;
wire [0:127] regfile_out_data_5;
wire [0:127] regfile_out_data_6;
// need fw en for odd pipe
wire ra_fw_en_2stage_even;
wire rb_fw_en_2stage_even;
wire rc_fw_en_2stage_even;
wire ra_fw_en_3stage_even;
```

```
wire rb_fw_en_3stage_even;
wire rc_fw_en_3stage_even;
wire ra_fw_en_4stage_even;
wire rb_fw_en_4stage_even;
wire rc_fw_en_4stage_even;
wire ra_fw_en_5stage_even;
wire rb_fw_en_5stage_even;
wire rc_fw_en_5stage_even;
wire ra_fw_en_6stage_even;
wire rb_fw_en_6stage_even;
wire rc_fw_en_6stage_even;
wire ra_fw_en_7stage_even;
wire rb_fw_en_7stage_even;
wire rc_fw_en_7stage_even;

wire ra_fw_en_2stage_odd;
wire rb_fw_en_2stage_odd;
wire rc_fw_en_2stage_odd;
wire ra_fw_en_3stage_odd;
wire rb_fw_en_3stage_odd;
wire rc_fw_en_3stage_odd;
wire ra_fw_en_4stage_odd;
wire rb_fw_en_4stage_odd;
wire rc_fw_en_4stage_odd;
wire ra_fw_en_5stage_odd;
wire rb_fw_en_5stage_odd;
wire rc_fw_en_5stage_odd;
wire ra_fw_en_6stage_odd;
wire rb_fw_en_6stage_odd;
wire rc_fw_en_6stage_odd;
wire ra_fw_en_7stage_odd;
wire rb_fw_en_7stage_odd;
wire rc_fw_en_7stage_odd;

wire ra_fw_en_2stage_from_odd_to_even;
wire rb_fw_en_2stage_from_odd_to_even;
wire rc_fw_en_2stage_from_odd_to_even;
```

```
wire ra_fw_en_3stage_from_odd_to_even;
wire rb_fw_en_3stage_from_odd_to_even;
wire rc_fw_en_3stage_from_odd_to_even;
wire ra_fw_en_4stage_from_odd_to_even;
wire rb_fw_en_4stage_from_odd_to_even;
wire rc_fw_en_4stage_from_odd_to_even;
wire ra_fw_en_5stage_from_odd_to_even;
wire rb_fw_en_5stage_from_odd_to_even;
wire rc_fw_en_5stage_from_odd_to_even;
wire ra_fw_en_6stage_from_odd_to_even;
wire rb_fw_en_6stage_from_odd_to_even;
wire rc_fw_en_6stage_from_odd_to_even;
wire ra_fw_en_7stage_from_odd_to_even;
wire rb_fw_en_7stage_from_odd_to_even;
wire rc_fw_en_7stage_from_odd_to_even;

wire ra_fw_en_2stage_from_even_to_odd;
wire rb_fw_en_2stage_from_even_to_odd;
wire rc_fw_en_2stage_from_even_to_odd;
wire ra_fw_en_3stage_from_even_to_odd;
wire rb_fw_en_3stage_from_even_to_odd;
wire rc_fw_en_3stage_from_even_to_odd;
wire ra_fw_en_4stage_from_even_to_odd;
wire rb_fw_en_4stage_from_even_to_odd;
wire rc_fw_en_4stage_from_even_to_odd;
wire ra_fw_en_5stage_from_even_to_odd;
wire rb_fw_en_5stage_from_even_to_odd;
wire rc_fw_en_5stage_from_even_to_odd;
wire ra_fw_en_6stage_from_even_to_odd;
wire rb_fw_en_6stage_from_even_to_odd;
wire rc_fw_en_6stage_from_even_to_odd;
wire ra_fw_en_7stage_from_even_to_odd;
wire rb_fw_en_7stage_from_even_to_odd;
wire rc_fw_en_7stage_from_even_to_odd;

Reg_file RF_inst(
  .clk(clk),
```

```
  .rst(rst),
  // write operations
  .reg_write_en_1(reg_write_en_1),
  .reg_write_en_2(reg_write_en_2),
  .reg_write_addr_1(reg_write_addr_1),
  .reg_write_addr_2(reg_write_addr_2),
  .reg_write_data_1(reg_write_data_1),
  .reg_write_data_2(reg_write_data_2),

  // read operations
  .reg_read_addr_1(ra_addr_even),
  .reg_read_addr_2(rb_addr_even),
  .reg_read_addr_3(rc_addr_even),
  .reg_read_addr_4(ra_addr_odd),
  .reg_read_addr_5(rb_addr_odd),
  .reg_read_addr_6(rc_addr_odd),

  .reg_read_data_1(regfile_out_data_1),
  .reg_read_data_2(regfile_out_data_2),
  .reg_read_data_3(regfile_out_data_3),
  .reg_read_data_4(regfile_out_data_4),
  .reg_read_data_5(regfile_out_data_5),
  .reg_read_data_6(regfile_out_data_6),

  .preload_en(preload_en),
  .preload_addr(preload_addr),
  .preload_values(preload_values)
);

Forwarding_Unit FU_inst(
  .reg_ra_src_even(ra_addr_even),
  .reg_rb_src_even(rb_addr_even),
  .reg_rc_src_even(rc_addr_even),

  .reg_ra_src_odd(ra_addr_odd),
  .reg_rb_src_odd(rb_addr_odd),
  .reg_rc_src_odd(rc_addr_odd),
```

```
.packed_2stage_even(packed_2stage_even),
.packed_3stage_even(packed_3stage_even),
.packed_4stage_even(packed_4stage_even),
.packed_5stage_even(packed_5stage_even),
.packed_6stage_even(packed_6stage_even),
.packed_7stage_even(packed_7stage_even),

.packed_2stage_odd(packed_2stage_odd),
.packed_3stage_odd(packed_3stage_odd),
.packed_4stage_odd(packed_4stage_odd),
.packed_5stage_odd(packed_5stage_odd),
.packed_6stage_odd(packed_6stage_odd),
.packed_7stage_odd(packed_7stage_odd),

// OUTPUT
.ra_fw_en_2stage_even(ra_fw_en_2stage_even),
.rb_fw_en_2stage_even(rb_fw_en_2stage_even),
.rc_fw_en_2stage_even(rc_fw_en_2stage_even),
.ra_fw_en_3stage_even(ra_fw_en_3stage_even),
.rb_fw_en_3stage_even(rb_fw_en_3stage_even),
.rc_fw_en_3stage_even(rc_fw_en_3stage_even),
.ra_fw_en_4stage_even(ra_fw_en_4stage_even),
.rb_fw_en_4stage_even(rb_fw_en_4stage_even),
.rc_fw_en_4stage_even(rc_fw_en_4stage_even),
.ra_fw_en_5stage_even(ra_fw_en_5stage_even),
.rb_fw_en_5stage_even(rb_fw_en_5stage_even),
.rc_fw_en_5stage_even(rc_fw_en_5stage_even),
.ra_fw_en_6stage_even(ra_fw_en_6stage_even),
.rb_fw_en_6stage_even(rb_fw_en_6stage_even),
.rc_fw_en_6stage_even(rc_fw_en_6stage_even),
.ra_fw_en_7stage_even(ra_fw_en_7stage_even),
.rb_fw_en_7stage_even(rb_fw_en_7stage_even),
.rc_fw_en_7stage_even(rc_fw_en_7stage_even),

.ra_fw_en_2stage_odd(ra_fw_en_2stage_odd),
.rb_fw_en_2stage_odd(rb_fw_en_2stage_odd),
```

```
.rc_fw_en_2stage_odd(rc_fw_en_2stage_odd),
.ra_fw_en_3stage_odd(ra_fw_en_3stage_odd),
.rb_fw_en_3stage_odd(rb_fw_en_3stage_odd),
.rc_fw_en_3stage_odd(rc_fw_en_3stage_odd),
.ra_fw_en_4stage_odd(ra_fw_en_4stage_odd),
.rb_fw_en_4stage_odd(rb_fw_en_4stage_odd),
.rc_fw_en_4stage_odd(rc_fw_en_4stage_odd),
.ra_fw_en_5stage_odd(ra_fw_en_5stage_odd),
.rb_fw_en_5stage_odd(rb_fw_en_5stage_odd),
.rc_fw_en_5stage_odd(rc_fw_en_5stage_odd),
.ra_fw_en_6stage_odd(ra_fw_en_6stage_odd),
.rb_fw_en_6stage_odd(rb_fw_en_6stage_odd),
.rc_fw_en_6stage_odd(rc_fw_en_6stage_odd),
.ra_fw_en_7stage_odd(ra_fw_en_7stage_odd),
.rb_fw_en_7stage_odd(rb_fw_en_7stage_odd),
.rc_fw_en_7stage_odd(rc_fw_en_7stage_odd),

.ra_fw_en_2stage_from_odd_to_even(ra_fw_en_2stage_from_odd_to_even),
.rb_fw_en_2stage_from_odd_to_even(rb_fw_en_2stage_from_odd_to_even),
.rc_fw_en_2stage_from_odd_to_even(rc_fw_en_2stage_from_odd_to_even),
.ra_fw_en_3stage_from_odd_to_even(ra_fw_en_3stage_from_odd_to_even),
.rb_fw_en_3stage_from_odd_to_even(rb_fw_en_3stage_from_odd_to_even),
.rc_fw_en_3stage_from_odd_to_even(rc_fw_en_3stage_from_odd_to_even),
.ra_fw_en_4stage_from_odd_to_even(ra_fw_en_4stage_from_odd_to_even),
.rb_fw_en_4stage_from_odd_to_even(rb_fw_en_4stage_from_odd_to_even),
.rc_fw_en_4stage_from_odd_to_even(rc_fw_en_4stage_from_odd_to_even),
.ra_fw_en_5stage_from_odd_to_even(ra_fw_en_5stage_from_odd_to_even),
.rb_fw_en_5stage_from_odd_to_even(rb_fw_en_5stage_from_odd_to_even),
.rc_fw_en_5stage_from_odd_to_even(rc_fw_en_5stage_from_odd_to_even),
.ra_fw_en_6stage_from_odd_to_even(ra_fw_en_6stage_from_odd_to_even),
.rb_fw_en_6stage_from_odd_to_even(rb_fw_en_6stage_from_odd_to_even),
.rc_fw_en_6stage_from_odd_to_even(rc_fw_en_6stage_from_odd_to_even),
.ra_fw_en_7stage_from_odd_to_even(ra_fw_en_7stage_from_odd_to_even),
.rb_fw_en_7stage_from_odd_to_even(rb_fw_en_7stage_from_odd_to_even),
.rc_fw_en_7stage_from_odd_to_even(rc_fw_en_7stage_from_odd_to_even),

.ra_fw_en_2stage_from_even_to_odd(ra_fw_en_2stage_from_even_to_odd),
```

```
  .rb_fw_en_2stage_from_even_to_odd(rb_fw_en_2stage_from_even_to_odd),
  .rc_fw_en_2stage_from_even_to_odd(rc_fw_en_2stage_from_even_to_odd),
  .ra_fw_en_3stage_from_even_to_odd(ra_fw_en_3stage_from_even_to_odd),
  .rb_fw_en_3stage_from_even_to_odd(rb_fw_en_3stage_from_even_to_odd),
  .rc_fw_en_3stage_from_even_to_odd(rc_fw_en_3stage_from_even_to_odd),
  .ra_fw_en_4stage_from_even_to_odd(ra_fw_en_4stage_from_even_to_odd),
  .rb_fw_en_4stage_from_even_to_odd(rb_fw_en_4stage_from_even_to_odd),
  .rc_fw_en_4stage_from_even_to_odd(rc_fw_en_4stage_from_even_to_odd),
  .ra_fw_en_5stage_from_even_to_odd(ra_fw_en_5stage_from_even_to_odd),
  .rb_fw_en_5stage_from_even_to_odd(rb_fw_en_5stage_from_even_to_odd),
  .rc_fw_en_5stage_from_even_to_odd(rc_fw_en_5stage_from_even_to_odd),
  .ra_fw_en_6stage_from_even_to_odd(ra_fw_en_6stage_from_even_to_odd),
  .rb_fw_en_6stage_from_even_to_odd(rb_fw_en_6stage_from_even_to_odd),
  .rc_fw_en_6stage_from_even_to_odd(rc_fw_en_6stage_from_even_to_odd),
  .ra_fw_en_7stage_from_even_to_odd(ra_fw_en_7stage_from_even_to_odd),
  .rb_fw_en_7stage_from_even_to_odd(rb_fw_en_7stage_from_even_to_odd),
  .rc_fw_en_7stage_from_even_to_odd(rc_fw_en_7stage_from_even_to_odd)
);

wire [0:6] temp_instr_id_even;
wire [0:6] temp_reg_dst_even;
wire [0:2] temp_unit_id_even;
wire [0:3] temp_latency_even;
wire temp_reg_wr_even;
wire [0:6] temp_imme7_even;
wire [0:9] temp_imme10_even;
wire [0:15] temp_imme16_even;
wire [0:17] temp_imme18_even;

wire [0:6] temp_instr_id_odd;
wire [0:6] temp_reg_dst_odd;
wire [0:2] temp_unit_id_odd;
wire [0:3] temp_latency_odd;
wire temp_reg_wr_odd;
wire [0:6] temp_imme7_odd;
wire [0:9] temp_imme10_odd;
wire [0:15] temp_imme16_odd;
```

```
wire [0:17] temp_imme18_odd;

assign temp_instr_id_even = instr_id_even;
assign temp_reg_dst_even = reg_dst_even;
assign temp_unit_id_even = unit_id_even;
assign temp_latency_even = latency_even;
assign temp_reg_wr_even = reg_wr_even;
assign temp_imme7_even = imme7_even;
assign temp_imme10_even = imme10_even;
assign temp_imme16_even = imme16_even;
assign temp_imme18_even = imme18_even;

assign temp_instr_id_odd = instr_id_odd;
assign temp_reg_dst_odd = reg_dst_odd;
assign temp_unit_id_odd = unit_id_odd;
assign temp_latency_odd = latency_odd;
assign temp_reg_wr_odd = reg_wr_odd;
assign temp_imme7_odd = imme7_odd;
assign temp_imme10_odd = imme10_odd;
assign temp_imme16_odd = imme16_odd;
assign temp_imme18_odd = imme18_odd;

// for forwarding reg dst result extracted from packed value
wire [0:127] reg_dst_result_2stage_even;
wire [0:127] reg_dst_result_3stage_even;
wire [0:127] reg_dst_result_4stage_even;
wire [0:127] reg_dst_result_5stage_even;
wire [0:127] reg_dst_result_6stage_even;
wire [0:127] reg_dst_result_7stage_even;

wire [0:127] reg_dst_result_2stage_odd;
wire [0:127] reg_dst_result_3stage_odd;
wire [0:127] reg_dst_result_4stage_odd;
wire [0:127] reg_dst_result_5stage_odd;
wire [0:127] reg_dst_result_6stage_odd;
wire [0:127] reg_dst_result_7stage_odd;
```

```
assign reg_dst_result_2stage_even = packed_2stage_even[3:130];
assign reg_dst_result_3stage_even = packed_3stage_even[3:130];
assign reg_dst_result_4stage_even = packed_4stage_even[3:130];
assign reg_dst_result_5stage_even = packed_5stage_even[3:130];
assign reg_dst_result_6stage_even = packed_6stage_even[3:130];
assign reg_dst_result_7stage_even = packed_7stage_even[3:130];

assign reg_dst_result_2stage_odd = packed_2stage_odd[3:130];
assign reg_dst_result_3stage_odd = packed_3stage_odd[3:130];
assign reg_dst_result_4stage_odd = packed_4stage_odd[3:130];
assign reg_dst_result_5stage_odd = packed_5stage_odd[3:130];
assign reg_dst_result_6stage_odd = packed_6stage_odd[3:130];
assign reg_dst_result_7stage_odd = packed_7stage_odd[3:130];

always @(posedge clk or posedge rst) begin
  if(rst) begin
    out_instr_id_even <= 7'b0;
    out_reg_dst_even <= 7'b0;
    out_unit_id_even <= 3'b0;
    out_latency_even <= 4'b0;
    out_reg_wr_even <= 1'b0;
    out_imme7_even <= 7'b0;
    out_imme10_even <= 10'b0;
    out_imme16_even <= 16'b0;
    out_imme18_even <= 18'b0;

    out_instr_id_odd <= 7'b0;
    out_reg_dst_odd <= 7'b0;
    out_unit_id_odd <= 3'b0;
    out_latency_odd <= 4'b0;
    out_reg_wr_odd <= 1'b0;
    out_imme7_odd <= 7'b0;
    out_imme10_odd <= 10'b0;
    out_imme16_odd <= 16'b0;
    out_imme18_odd <= 18'b0;

    ra_data_even <= 128'b0;
```

```
    rb_data_even <= 128'b0;
    rc_data_even <= 128'b0;

    ra_data_odd <= 128'b0;
    rb_data_odd <= 128'b0;
    rc_data_odd <= 128'b0;
  end
  else if (flush) begin
    out_instr_id_even <= 7'd86;
    out_reg_dst_even <= 7'b0;
    out_unit_id_even <= 3'b0;
    out_latency_even <= 4'b0;
    out_reg_wr_even <= 1'b0;
    out_imme7_even <= 7'b0;
    out_imme10_even <= 10'b0;
    out_imme16_even <= 16'b0;
    out_imme18_even <= 18'b0;

    ra_data_even <= 128'b0;
    rb_data_even <= 128'b0;
    rc_data_even <= 128'b0;

    out_instr_id_odd <= 7'd85;
    out_reg_dst_odd <= 7'b0;
    out_unit_id_odd <= 3'b0;
    out_latency_odd <= 4'b0;
    out_reg_wr_odd <= 1'b0;
    out_imme7_odd <= 7'b0;
    out_imme10_odd <= 10'b0;
    out_imme16_odd <= 16'b0;
    out_imme18_odd <= 18'b0;

    ra_data_odd <= 128'b0;
    rb_data_odd <= 128'b0;
    rc_data_odd <= 128'b0;
  end
  else begin
```

```
    out_instr_id_even <= temp_instr_id_even;
    out_reg_dst_even <= temp_reg_dst_even;
    out_unit_id_even <= temp_unit_id_even;
    out_latency_even <= temp_latency_even;
    out_reg_wr_even <= temp_reg_wr_even;
    out_imme7_even <= temp_imme7_even;
    out_imme10_even <= temp_imme10_even;
    out_imme16_even <= temp_imme16_even;
    out_imme18_even <= temp_imme18_even;

    out_instr_id_odd <= temp_instr_id_odd;
    out_reg_dst_odd <= temp_reg_dst_odd;
    out_unit_id_odd <= temp_unit_id_odd;
    out_latency_odd <= temp_latency_odd;
    out_reg_wr_odd <= temp_reg_wr_odd;
    out_imme7_odd <= temp_imme7_odd;
    out_imme10_odd <= temp_imme10_odd;
    out_imme16_odd <= temp_imme16_odd;
    out_imme18_odd <= temp_imme18_odd;

    // when result has to be forwarded to ra (even), recent stages has priority for forwarding ALSO same pipe has
priority
    ra_data_even <=         (ra_fw_en_2stage_even) ? reg_dst_result_2stage_even :
                (ra_fw_en_2stage_from_odd_to_even) ? reg_dst_result_2stage_odd :
                        (ra_fw_en_3stage_even) ? reg_dst_result_3stage_even :
                (ra_fw_en_3stage_from_odd_to_even) ? reg_dst_result_3stage_odd :
                        (ra_fw_en_4stage_even) ? reg_dst_result_4stage_even :
                (ra_fw_en_4stage_from_odd_to_even) ? reg_dst_result_4stage_odd :
                        (ra_fw_en_5stage_even) ? reg_dst_result_5stage_even :
                (ra_fw_en_5stage_from_odd_to_even) ? reg_dst_result_5stage_odd :
                        (ra_fw_en_6stage_even) ? reg_dst_result_6stage_even :
                (ra_fw_en_6stage_from_odd_to_even) ? reg_dst_result_6stage_odd :
                        (ra_fw_en_7stage_even) ? reg_dst_result_7stage_even :
                (ra_fw_en_7stage_from_odd_to_even) ? reg_dst_result_7stage_odd :
                                        regfile_out_data_1;

    rb_data_even <=         (rb_fw_en_2stage_even) ? reg_dst_result_2stage_even :
```

```
                (rb_fw_en_2stage_from_odd_to_even) ? reg_dst_result_2stage_odd :
                        (rb_fw_en_3stage_even) ? reg_dst_result_3stage_even :
                (rb_fw_en_3stage_from_odd_to_even) ? reg_dst_result_3stage_odd :
                        (rb_fw_en_4stage_even) ? reg_dst_result_4stage_even :
                (rb_fw_en_4stage_from_odd_to_even) ? reg_dst_result_4stage_odd :
                        (rb_fw_en_5stage_even) ? reg_dst_result_5stage_even :
                (rb_fw_en_5stage_from_odd_to_even) ? reg_dst_result_5stage_odd :
                        (rb_fw_en_6stage_even) ? reg_dst_result_6stage_even :
                (rb_fw_en_6stage_from_odd_to_even) ? reg_dst_result_6stage_odd :
                        (rb_fw_en_7stage_even) ? reg_dst_result_7stage_even :
                (rb_fw_en_7stage_from_odd_to_even) ? reg_dst_result_7stage_odd :
                                        regfile_out_data_2;

rc_data_even <=         (rc_fw_en_2stage_even) ? reg_dst_result_2stage_even :
                (rc_fw_en_2stage_from_odd_to_even) ? reg_dst_result_2stage_odd :
                        (rc_fw_en_3stage_even) ? reg_dst_result_3stage_even :
                (rc_fw_en_3stage_from_odd_to_even) ? reg_dst_result_3stage_odd :
                        (rc_fw_en_4stage_even) ? reg_dst_result_4stage_even :
                (rc_fw_en_4stage_from_odd_to_even) ? reg_dst_result_4stage_odd :
                        (rc_fw_en_5stage_even) ? reg_dst_result_5stage_even :
                (rc_fw_en_5stage_from_odd_to_even) ? reg_dst_result_5stage_odd :
                        (rc_fw_en_6stage_even) ? reg_dst_result_6stage_even :
                (rc_fw_en_6stage_from_odd_to_even) ? reg_dst_result_6stage_odd :
                        (rc_fw_en_7stage_even) ? reg_dst_result_7stage_even :
                (rc_fw_en_7stage_from_odd_to_even) ? reg_dst_result_7stage_odd :
                                        regfile_out_data_3;

ra_data_odd <=          (ra_fw_en_2stage_odd) ? reg_dst_result_2stage_odd :
                (ra_fw_en_2stage_from_even_to_odd) ? reg_dst_result_2stage_even :
                        (ra_fw_en_3stage_odd) ? reg_dst_result_3stage_odd :
                (ra_fw_en_3stage_from_even_to_odd) ? reg_dst_result_3stage_even :
                        (ra_fw_en_4stage_odd) ? reg_dst_result_4stage_odd :
                (ra_fw_en_4stage_from_even_to_odd) ? reg_dst_result_4stage_even :
                        (ra_fw_en_5stage_odd) ? reg_dst_result_5stage_odd :
                (ra_fw_en_5stage_from_even_to_odd) ? reg_dst_result_5stage_even :
                        (ra_fw_en_6stage_odd) ? reg_dst_result_6stage_odd :
                (ra_fw_en_6stage_from_even_to_odd) ? reg_dst_result_6stage_even :
```

```
                    (ra_fw_en_7stage_odd) ? reg_dst_result_7stage_odd :
                (ra_fw_en_7stage_from_even_to_odd) ? reg_dst_result_7stage_even :
                                        regfile_out_data_4;

 rb_data_odd <=             (rb_fw_en_2stage_odd) ? reg_dst_result_2stage_odd :
                (rb_fw_en_2stage_from_even_to_odd) ? reg_dst_result_2stage_even :
                    (rb_fw_en_3stage_odd) ? reg_dst_result_3stage_odd :
                (rb_fw_en_3stage_from_even_to_odd) ? reg_dst_result_3stage_even :
                    (rb_fw_en_4stage_odd) ? reg_dst_result_4stage_odd :
                (rb_fw_en_4stage_from_even_to_odd) ? reg_dst_result_4stage_even :
                    (rb_fw_en_5stage_odd) ? reg_dst_result_5stage_odd :
                (rb_fw_en_5stage_from_even_to_odd) ? reg_dst_result_5stage_even :
                    (rb_fw_en_6stage_odd) ? reg_dst_result_6stage_odd :
                (rb_fw_en_6stage_from_even_to_odd) ? reg_dst_result_6stage_even :
                    (rb_fw_en_7stage_odd) ? reg_dst_result_7stage_odd :
                (rb_fw_en_7stage_from_even_to_odd) ? reg_dst_result_7stage_even :
                                        regfile_out_data_5;

 rc_data_odd <=             (rc_fw_en_2stage_odd) ? reg_dst_result_2stage_odd :
                (rc_fw_en_2stage_from_even_to_odd) ? reg_dst_result_2stage_even :
                    (rc_fw_en_3stage_odd) ? reg_dst_result_3stage_odd :
                (rc_fw_en_3stage_from_even_to_odd) ? reg_dst_result_3stage_even :
                    (rc_fw_en_4stage_odd) ? reg_dst_result_4stage_odd :
                (rc_fw_en_4stage_from_even_to_odd) ? reg_dst_result_4stage_even :
                    (rc_fw_en_5stage_odd) ? reg_dst_result_5stage_odd :
                (rc_fw_en_5stage_from_even_to_odd) ? reg_dst_result_5stage_even :
                    (rc_fw_en_6stage_odd) ? reg_dst_result_6stage_odd :
                (rc_fw_en_6stage_from_even_to_odd) ? reg_dst_result_6stage_even :
                    (rc_fw_en_7stage_odd) ? reg_dst_result_7stage_odd :
                (rc_fw_en_7stage_from_even_to_odd) ? reg_dst_result_7stage_even :
                                        regfile_out_data_6;



end
instr1_branch_out <= instr1_branch;
PC_pass_out <= PC_pass_in;
```

```
end

endmodule
```

Appendix H (Reg_File.v)

```
module Reg_file(
  input clk,
  input rst,

  input reg_write_en_1,
  input reg_write_en_2,

  input [0:6] reg_write_addr_1,
  input [0:6] reg_write_addr_2,

  input [0:127] reg_write_data_1,
  input [0:127] reg_write_data_2,

  input [0:6] reg_read_addr_1,
  input [0:6] reg_read_addr_2,
  input [0:6] reg_read_addr_3,
  input [0:6] reg_read_addr_4,
  input [0:6] reg_read_addr_5,
  input [0:6] reg_read_addr_6,

  output [0:127] reg_read_data_1,
  output [0:127] reg_read_data_2,
  output [0:127] reg_read_data_3,
  output [0:127] reg_read_data_4,
  output [0:127] reg_read_data_5,
  output [0:127] reg_read_data_6,

  // Preload inputs for verification purpose only
  input preload_en,
  input [0:6] preload_addr,
  input [0:127] preload_values // array input does not support from veilog, only support in systemverilog
```

```
);
// 128 registers, 128 bit width
reg [0:127] reg_file [0:127];

// read operations (asynchronous) also resolving RAW hazards
// write enable 1 has higher priority than write enable 2
assign reg_read_data_1 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_1) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_1) ? reg_write_data_2 :
    reg_file[reg_read_addr_1];

assign reg_read_data_2 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_2) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_2) ? reg_write_data_2 :
    reg_file[reg_read_addr_2];

assign reg_read_data_3 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_3) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_3) ? reg_write_data_2 :
    reg_file[reg_read_addr_3];

assign reg_read_data_4 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_4) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_4) ? reg_write_data_2 :
    reg_file[reg_read_addr_4];

assign reg_read_data_5 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_5) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_5) ? reg_write_data_2 :
    reg_file[reg_read_addr_5];

assign reg_read_data_6 =
    (reg_write_en_1 && reg_write_addr_1 == reg_read_addr_6) ? reg_write_data_1 :
    (reg_write_en_2 && reg_write_addr_2 == reg_read_addr_6) ? reg_write_data_2 :
    reg_file[reg_read_addr_6];
integer i;
```

```
// write operations (synchronous)
always @(posedge clk or posedge rst) begin
  if (rst) begin
    if (preload_en) begin
      // Load register every cycle
      reg_file[preload_addr] <= preload_values;
    end else begin
      for(i=0;i<128;i=i+1) begin
        reg_file[i] <= 128'b0;
      end
    end
  end
  else begin
    if (reg_write_en_1) begin
      reg_file[reg_write_addr_1] <= reg_write_data_1;
    end
    if (reg_write_en_2) begin
      reg_file[reg_write_addr_2] <= reg_write_data_2;
    end
  end
end

endmodule
```

Appendix I (Forwarding_Unit.v)

```
module Forwarding_Unit(
  // INPUT
  input [0:6] reg_ra_src_even, // even pipe register source
  input [0:6] reg_rb_src_even,
  input [0:6] reg_rc_src_even,

  input [0:6] reg_ra_src_odd, // odd pipe register source
  input [0:6] reg_rb_src_odd,
  input [0:6] reg_rc_src_odd,

  // input from 7 stage pipes
  input [0:142] packed_2stage_even,
```

```
  input [0:142] packed_3stage_even,
  input [0:142] packed_4stage_even,
  input [0:142] packed_5stage_even,
  input [0:142] packed_6stage_even,
  input [0:142] packed_7stage_even,

  input [0:142] packed_2stage_odd,
  input [0:142] packed_3stage_odd,
  input [0:142] packed_4stage_odd,
  input [0:142] packed_5stage_odd,
  input [0:142] packed_6stage_odd,
  input [0:142] packed_7stage_odd,
// OUTPUT
// even pipe forwarding enable ------------------
  output ra_fw_en_2stage_even,
  output rb_fw_en_2stage_even,
  output rc_fw_en_2stage_even,

  output ra_fw_en_3stage_even,
  output rb_fw_en_3stage_even,
  output rc_fw_en_3stage_even,

  output ra_fw_en_4stage_even,
  output rb_fw_en_4stage_even,
  output rc_fw_en_4stage_even,

  output ra_fw_en_5stage_even,
  output rb_fw_en_5stage_even,
  output rc_fw_en_5stage_even,

  output ra_fw_en_6stage_even,
  output rb_fw_en_6stage_even,
  output rc_fw_en_6stage_even,

  output ra_fw_en_7stage_even,
  output rb_fw_en_7stage_even,
  output rc_fw_en_7stage_even,
```

```
// ---------------------------------------------

// odd pipe forwarding enable -------------------
  output ra_fw_en_2stage_odd,
  output rb_fw_en_2stage_odd,
  output rc_fw_en_2stage_odd,

  output ra_fw_en_3stage_odd,
  output rb_fw_en_3stage_odd,
  output rc_fw_en_3stage_odd,

  output ra_fw_en_4stage_odd,
  output rb_fw_en_4stage_odd,
  output rc_fw_en_4stage_odd,

  output ra_fw_en_5stage_odd,
  output rb_fw_en_5stage_odd,
  output rc_fw_en_5stage_odd,

  output ra_fw_en_6stage_odd,
  output rb_fw_en_6stage_odd,
  output rc_fw_en_6stage_odd,

  output ra_fw_en_7stage_odd,
  output rb_fw_en_7stage_odd,
  output rc_fw_en_7stage_odd,
  // ---------------------------------------------

  // cross pipe forwarding enable -------------------
  output ra_fw_en_2stage_from_odd_to_even,
  output rb_fw_en_2stage_from_odd_to_even,
  output rc_fw_en_2stage_from_odd_to_even,

  output ra_fw_en_3stage_from_odd_to_even,
  output rb_fw_en_3stage_from_odd_to_even,
  output rc_fw_en_3stage_from_odd_to_even,
```

```
output ra_fw_en_4stage_from_odd_to_even,
output rb_fw_en_4stage_from_odd_to_even,
output rc_fw_en_4stage_from_odd_to_even,

output ra_fw_en_5stage_from_odd_to_even,
output rb_fw_en_5stage_from_odd_to_even,
output rc_fw_en_5stage_from_odd_to_even,

output ra_fw_en_6stage_from_odd_to_even,
output rb_fw_en_6stage_from_odd_to_even,
output rc_fw_en_6stage_from_odd_to_even,

output ra_fw_en_7stage_from_odd_to_even,
output rb_fw_en_7stage_from_odd_to_even,
output rc_fw_en_7stage_from_odd_to_even,

output ra_fw_en_2stage_from_even_to_odd,
output rb_fw_en_2stage_from_even_to_odd,
output rc_fw_en_2stage_from_even_to_odd,

output ra_fw_en_3stage_from_even_to_odd,
output rb_fw_en_3stage_from_even_to_odd,
output rc_fw_en_3stage_from_even_to_odd,

output ra_fw_en_4stage_from_even_to_odd,
output rb_fw_en_4stage_from_even_to_odd,
output rc_fw_en_4stage_from_even_to_odd,

output ra_fw_en_5stage_from_even_to_odd,
output rb_fw_en_5stage_from_even_to_odd,
output rc_fw_en_5stage_from_even_to_odd,

output ra_fw_en_6stage_from_even_to_odd,
output rb_fw_en_6stage_from_even_to_odd,
output rc_fw_en_6stage_from_even_to_odd,

output ra_fw_en_7stage_from_even_to_odd,
```

```
  output rb_fw_en_7stage_from_even_to_odd,
  output rc_fw_en_7stage_from_even_to_odd
);
  // NOTE: forwarding actually only works from 2 stage to 6 stage
  // even pipe ------------------
  wire [0:6] reg_dst_2stage_even; // even pipe register destination
  wire [0:6] reg_dst_3stage_even;
  wire [0:6] reg_dst_4stage_even;
  wire [0:6] reg_dst_5stage_even;
  wire [0:6] reg_dst_6stage_even;
  wire [0:6] reg_dst_7stage_even;

  wire reg_wr_2stage_even;
  wire reg_wr_3stage_even;
  wire reg_wr_4stage_even;
  wire reg_wr_5stage_even;
  wire reg_wr_6stage_even;
  wire reg_wr_7stage_even;

  wire [0:3] latency_2stage_even;
  wire [0:3] latency_3stage_even;
  wire [0:3] latency_4stage_even;
  wire [0:3] latency_5stage_even;
  wire [0:3] latency_6stage_even;
  wire [0:3] latency_7stage_even;

  // unpacking packed data
  assign reg_dst_2stage_even = packed_2stage_even[131:137];
  assign reg_dst_3stage_even = packed_3stage_even[131:137];
  assign reg_dst_4stage_even = packed_4stage_even[131:137];
  assign reg_dst_5stage_even = packed_5stage_even[131:137];
  assign reg_dst_6stage_even = packed_6stage_even[131:137];
  assign reg_dst_7stage_even = packed_7stage_even[131:137];

  assign reg_wr_2stage_even = packed_2stage_even[142];
  assign reg_wr_3stage_even = packed_3stage_even[142];
  assign reg_wr_4stage_even = packed_4stage_even[142];
```

```
  assign reg_wr_5stage_even = packed_5stage_even[142];
  assign reg_wr_6stage_even = packed_6stage_even[142];
  assign reg_wr_7stage_even = packed_7stage_even[142];

  assign latency_2stage_even = packed_2stage_even[138:141];
  assign latency_3stage_even = packed_3stage_even[138:141];
  assign latency_4stage_even = packed_4stage_even[138:141];
  assign latency_5stage_even = packed_5stage_even[138:141];
  assign latency_6stage_even = packed_6stage_even[138:141];
  assign latency_7stage_even = packed_7stage_even[138:141];
// odd pipe ------------------

  wire [0:6] reg_dst_1stage_odd;
  wire [0:6] reg_dst_2stage_odd;
  wire [0:6] reg_dst_3stage_odd;
  wire [0:6] reg_dst_4stage_odd;
  wire [0:6] reg_dst_5stage_odd;
  wire [0:6] reg_dst_6stage_odd;
  wire [0:6] reg_dst_7stage_odd;

  wire reg_wr_1stage_odd;
  wire reg_wr_2stage_odd;
  wire reg_wr_3stage_odd;
  wire reg_wr_4stage_odd;
  wire reg_wr_5stage_odd;
  wire reg_wr_6stage_odd;
  wire reg_wr_7stage_odd;

  wire [0:3] latency_1stage_odd;
  wire [0:3] latency_2stage_odd;
  wire [0:3] latency_3stage_odd;
  wire [0:3] latency_4stage_odd;
  wire [0:3] latency_5stage_odd;
  wire [0:3] latency_6stage_odd;
  wire [0:3] latency_7stage_odd;

  // unpacking packed data
```

```
  assign reg_dst_2stage_odd = packed_2stage_odd[131:137];
  assign reg_dst_3stage_odd = packed_3stage_odd[131:137];
  assign reg_dst_4stage_odd = packed_4stage_odd[131:137];
  assign reg_dst_5stage_odd = packed_5stage_odd[131:137];
  assign reg_dst_6stage_odd = packed_6stage_odd[131:137];
  assign reg_dst_7stage_odd = packed_7stage_odd[131:137];

  assign reg_wr_2stage_odd = packed_2stage_odd[142];
  assign reg_wr_3stage_odd = packed_3stage_odd[142];
  assign reg_wr_4stage_odd = packed_4stage_odd[142];
  assign reg_wr_5stage_odd = packed_5stage_odd[142];
  assign reg_wr_6stage_odd = packed_6stage_odd[142];
  assign reg_wr_7stage_odd = packed_7stage_odd[142];

  assign latency_2stage_odd = packed_2stage_odd[138:141];
  assign latency_3stage_odd = packed_3stage_odd[138:141];
  assign latency_4stage_odd = packed_4stage_odd[138:141];
  assign latency_5stage_odd = packed_5stage_odd[138:141];
  assign latency_6stage_odd = packed_6stage_odd[138:141];
  assign latency_7stage_odd = packed_7stage_odd[138:141];

  // forwarding enable

  assign ra_fw_en_2stage_even = (reg_ra_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);
  assign rb_fw_en_2stage_even = (reg_rb_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);
  assign rc_fw_en_2stage_even = (reg_rc_src_even == reg_dst_2stage_even) && reg_wr_2stage_even &&
(latency_2stage_even <= 2);

  assign ra_fw_en_3stage_even = (reg_ra_src_even == reg_dst_3stage_even) && reg_wr_3stage_even &&
(latency_3stage_even <= 3);
  assign rb_fw_en_3stage_even = (reg_rb_src_even == reg_dst_3stage_even) && reg_wr_3stage_even &&
(latency_3stage_even <= 3);
  assign rc_fw_en_3stage_even = (reg_rc_src_even == reg_dst_3stage_even) && reg_wr_3stage_even &&
(latency_3stage_even <= 3);
```

```
  assign ra_fw_en_4stage_even = (reg_ra_src_even == reg_dst_4stage_even) && reg_wr_4stage_even &&
(latency_4stage_even <= 4);
  assign rb_fw_en_4stage_even = (reg_rb_src_even == reg_dst_4stage_even) && reg_wr_4stage_even &&
(latency_4stage_even <= 4);
  assign rc_fw_en_4stage_even = (reg_rc_src_even == reg_dst_4stage_even) && reg_wr_4stage_even &&
(latency_4stage_even <= 4);

  assign ra_fw_en_5stage_even = (reg_ra_src_even == reg_dst_5stage_even) && reg_wr_5stage_even &&
(latency_5stage_even <= 5);
  assign rb_fw_en_5stage_even = (reg_rb_src_even == reg_dst_5stage_even) && reg_wr_5stage_even &&
(latency_5stage_even <= 5);
  assign rc_fw_en_5stage_even = (reg_rc_src_even == reg_dst_5stage_even) && reg_wr_5stage_even &&
(latency_5stage_even <= 5);

  assign ra_fw_en_6stage_even = (reg_ra_src_even == reg_dst_6stage_even) && reg_wr_6stage_even &&
(latency_6stage_even <= 6);
  assign rb_fw_en_6stage_even = (reg_rb_src_even == reg_dst_6stage_even) && reg_wr_6stage_even &&
(latency_6stage_even <= 6);
  assign rc_fw_en_6stage_even = (reg_rc_src_even == reg_dst_6stage_even) && reg_wr_6stage_even &&
(latency_6stage_even <= 6);

  assign ra_fw_en_7stage_even = (reg_ra_src_even == reg_dst_7stage_even) && reg_wr_7stage_even &&
(latency_7stage_even <= 7);
  assign rb_fw_en_7stage_even = (reg_rb_src_even == reg_dst_7stage_even) && reg_wr_7stage_even &&
(latency_7stage_even <= 7);
  assign rc_fw_en_7stage_even = (reg_rc_src_even == reg_dst_7stage_even) && reg_wr_7stage_even &&
(latency_7stage_even <= 7);

  assign ra_fw_en_2stage_odd = (reg_ra_src_odd == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);
  assign rb_fw_en_2stage_odd = (reg_rb_src_odd == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);
  assign rc_fw_en_2stage_odd = (reg_rc_src_odd == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);

  assign ra_fw_en_3stage_odd = (reg_ra_src_odd == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);
```

```
  assign rb_fw_en_3stage_odd = (reg_rb_src_odd == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);
  assign rc_fw_en_3stage_odd = (reg_rc_src_odd == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);

  assign ra_fw_en_4stage_odd = (reg_ra_src_odd == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);
  assign rb_fw_en_4stage_odd = (reg_rb_src_odd == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);
  assign rc_fw_en_4stage_odd = (reg_rc_src_odd == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);

  assign ra_fw_en_5stage_odd = (reg_ra_src_odd == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);
  assign rb_fw_en_5stage_odd = (reg_rb_src_odd == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);
  assign rc_fw_en_5stage_odd = (reg_rc_src_odd == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);

  assign ra_fw_en_6stage_odd = (reg_ra_src_odd == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);
  assign rb_fw_en_6stage_odd = (reg_rb_src_odd == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);
  assign rc_fw_en_6stage_odd = (reg_rc_src_odd == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);

  assign ra_fw_en_7stage_odd = (reg_ra_src_odd == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);
  assign rb_fw_en_7stage_odd = (reg_rb_src_odd == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);
  assign rc_fw_en_7stage_odd = (reg_rc_src_odd == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);

  // cross pipe forwarding enable

  assign ra_fw_en_2stage_from_odd_to_even = (reg_ra_src_even == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);
```

```
  assign rb_fw_en_2stage_from_odd_to_even = (reg_rb_src_even == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);
  assign rc_fw_en_2stage_from_odd_to_even = (reg_rc_src_even == reg_dst_2stage_odd) && reg_wr_2stage_odd &&
(latency_2stage_odd <= 2);

  assign ra_fw_en_3stage_from_odd_to_even = (reg_ra_src_even == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);
  assign rb_fw_en_3stage_from_odd_to_even = (reg_rb_src_even == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);
  assign rc_fw_en_3stage_from_odd_to_even = (reg_rc_src_even == reg_dst_3stage_odd) && reg_wr_3stage_odd &&
(latency_3stage_odd <= 3);

  assign ra_fw_en_4stage_from_odd_to_even = (reg_ra_src_even == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);
  assign rb_fw_en_4stage_from_odd_to_even = (reg_rb_src_even == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);
  assign rc_fw_en_4stage_from_odd_to_even = (reg_rc_src_even == reg_dst_4stage_odd) && reg_wr_4stage_odd &&
(latency_4stage_odd <= 4);

  assign ra_fw_en_5stage_from_odd_to_even = (reg_ra_src_even == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);
  assign rb_fw_en_5stage_from_odd_to_even = (reg_rb_src_even == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);
  assign rc_fw_en_5stage_from_odd_to_even = (reg_rc_src_even == reg_dst_5stage_odd) && reg_wr_5stage_odd &&
(latency_5stage_odd <= 5);

  assign ra_fw_en_6stage_from_odd_to_even = (reg_ra_src_even == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);
  assign rb_fw_en_6stage_from_odd_to_even = (reg_rb_src_even == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);
  assign rc_fw_en_6stage_from_odd_to_even = (reg_rc_src_even == reg_dst_6stage_odd) && reg_wr_6stage_odd &&
(latency_6stage_odd <= 6);

  assign ra_fw_en_7stage_from_odd_to_even = (reg_ra_src_even == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);
  assign rb_fw_en_7stage_from_odd_to_even = (reg_rb_src_even == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);
```

```
  assign rc_fw_en_7stage_from_odd_to_even = (reg_rc_src_even == reg_dst_7stage_odd) && reg_wr_7stage_odd &&
(latency_7stage_odd <= 7);

  assign ra_fw_en_2stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_2stage_even) && reg_wr_2stage_even
&& (latency_2stage_even <= 2);
  assign rb_fw_en_2stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_2stage_even) && reg_wr_2stage_even
&& (latency_2stage_even <= 2);
  assign rc_fw_en_2stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_2stage_even) && reg_wr_2stage_even
&& (latency_2stage_even <= 2);

  assign ra_fw_en_3stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_3stage_even) && reg_wr_3stage_even
&& (latency_3stage_even <= 3);
  assign rb_fw_en_3stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_3stage_even) && reg_wr_3stage_even
&& (latency_3stage_even <= 3);
  assign rc_fw_en_3stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_3stage_even) && reg_wr_3stage_even
&& (latency_3stage_even <= 3);

  assign ra_fw_en_4stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_4stage_even) && reg_wr_4stage_even
&& (latency_4stage_even <= 4);
  assign rb_fw_en_4stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_4stage_even) && reg_wr_4stage_even
&& (latency_4stage_even <= 4);
  assign rc_fw_en_4stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_4stage_even) && reg_wr_4stage_even
&& (latency_4stage_even <= 4);

  assign ra_fw_en_5stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_5stage_even) && reg_wr_5stage_even
&& (latency_5stage_even <= 5);
  assign rb_fw_en_5stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_5stage_even) && reg_wr_5stage_even
&& (latency_5stage_even <= 5);
  assign rc_fw_en_5stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_5stage_even) && reg_wr_5stage_even
&& (latency_5stage_even <= 5);

  assign ra_fw_en_6stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_6stage_even) && reg_wr_6stage_even
&& (latency_6stage_even <= 6);
  assign rb_fw_en_6stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_6stage_even) && reg_wr_6stage_even
&& (latency_6stage_even <= 6);
  assign rc_fw_en_6stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_6stage_even) && reg_wr_6stage_even
&& (latency_6stage_even <= 6);
```

```
  assign ra_fw_en_7stage_from_even_to_odd = (reg_ra_src_odd == reg_dst_7stage_even) && reg_wr_7stage_even
&& (latency_7stage_even <= 7);
  assign rb_fw_en_7stage_from_even_to_odd = (reg_rb_src_odd == reg_dst_7stage_even) && reg_wr_7stage_even
&& (latency_7stage_even <= 7);
  assign rc_fw_en_7stage_from_even_to_odd = (reg_rc_src_odd == reg_dst_7stage_even) && reg_wr_7stage_even
&& (latency_7stage_even <= 7);

  endmodule
```

Appendix J (Even_Pipe.v)

```
module Even_Pipe(
  input clk,
  input rst,
  input flush,
  input flush_4stage,

  input [0:6] instr_id,
  input [0:6] reg_dst,
  input [0:2] unit_id,
  input [0:3] latency,
  input reg_wr,

  input [0:127] ra_data,
  input [0:127] rb_data,
  input [0:127] rc_data, // rc port is also used for instructions that need rt data as an operand
  input [0:6] imme7,
  input [0:9] imme10,
  input [0:15] imme16,
  input [0:17] imme18,

  // output for forwarding unit
  output reg [0:142] packed_2stage,
  output reg [0:142] packed_3stage,
  output reg [0:142] packed_4stage,
  output reg [0:142] packed_5stage,
```

```
  output reg [0:142] packed_6stage,
  output reg [0:142] packed_7stage,

  // Write back stage
  output reg [0:6] WB_reg_write_addr,
  output reg [0:127] WB_reg_write_data,
  output reg WB_reg_write_en

);
// [0:2] unit ID, [3:130] 128-bit result, [131:137] reg_dst, [138:141] latency, [142] RegWr
reg [0:142] packed_1stage;

reg [0:127] result;

wire [0:127] FX1_result, FX2_result, SP_result, BYTE_result;


FX1_ALU fx1_inst (
  .instr_id(instr_id),
  .ra_data(ra_data),
  .rb_data(rb_data),
  .rc_data(rc_data),
  .imme7(imme7),
  .imme10(imme10),
  .imme16(imme16),
  .imme18(imme18),
  .result(FX1_result)
);

// FX2 unit
FX2_ALU FX2_inst(
  .instr_id(instr_id),
  .ra_data(ra_data),
  .rb_data(rb_data),
  .rc_data(rc_data),
  .imme7(imme7),
  .imme10(imme10),
```

```
  .imme16(imme16),
  .imme18(imme18),
  .result(FX2_result)
);

// SP unit
SP_ALU SP_inst(
  .instr_id(instr_id),
  .ra_data(ra_data),
  .rb_data(rb_data),
  .rc_data(rc_data),
  .imme7(imme7),
  .imme10(imme10),
  .imme16(imme16),
  .imme18(imme18),
  .result(SP_result)
);

// BYTE unit
BYTE_ALU BYTE_inst(
  .instr_id(instr_id),
  .ra_data(ra_data),
  .rb_data(rb_data),
  .rc_data(rc_data),
  .imme7(imme7),
  .imme10(imme10),
  .imme16(imme16),
  .imme18(imme18),
  .result(BYTE_result)
);

always @(*) begin
  if (instr_id == 7'd86 | instr_id == 7'd87) begin
    packed_1stage = 143'b0;
  end else begin
    case (unit_id)
      3'b001: result = FX1_result;
```

```
      3'b010: result = FX2_result;
      3'b011: result = SP_result;
      3'b100: result = BYTE_result;
      default: result = 128'd0;
    endcase
    packed_1stage = {unit_id, result, reg_dst, latency, reg_wr};
  end
end

always @(posedge clk or posedge rst) begin
  if (rst) begin
    packed_1stage <= 0;
    packed_2stage <= 0;
    packed_3stage <= 0;
    packed_4stage <= 0;
    packed_5stage <= 0;
    packed_6stage <= 0;
    packed_7stage <= 0;
  end
  else if (flush) begin
    packed_2stage <= 0;
    packed_3stage <= 0;
    if(flush_4stage) begin
      packed_4stage <= 0;
    end
    else begin
      packed_4stage <= packed_3stage;
    end
    packed_5stage <= packed_4stage;
    packed_6stage <= packed_5stage;
    packed_7stage <= packed_6stage;
    WB_reg_write_addr <= packed_7stage[131:137];
    WB_reg_write_data <= packed_7stage[3:130];
    WB_reg_write_en <= packed_7stage[142];
  end
  else begin
    packed_2stage <= packed_1stage;
```

```
    packed_3stage <= packed_2stage;
    packed_4stage <= packed_3stage;
    packed_5stage <= packed_4stage;
    packed_6stage <= packed_5stage;
    packed_7stage <= packed_6stage;
    WB_reg_write_addr <= packed_7stage[131:137];
    WB_reg_write_data <= packed_7stage[3:130];
    WB_reg_write_en <= packed_7stage[142];
  end
end
endmodule
```

Appendix K (Odd_Pipe.v)

```
module Odd_Pipe(
  input clk,
  input rst,
  input flush,

  input [0:6] instr_id,
  input [0:6] reg_dst,
  input [0:2] unit_id,
  input [0:3] latency,
  input reg_wr,

  input [0:127] ra_data,
  input [0:127] rb_data,
  input [0:127] rc_data, // rc port is also used for instructions that need rt data as an operand
  input [0:6] imme7,
  input [0:9] imme10,
  input [0:15] imme16,
  input [0:17] imme18,

  input [0:8] current_PC,
  input instr1_branch,

  // output for forwarding unit
```

```
  output reg [0:142] packed_2stage,
  output reg [0:142] packed_3stage,
  output reg [0:142] packed_4stage,
  output reg [0:142] packed_5stage,
  output reg [0:142] packed_6stage,
  output reg [0:142] packed_7stage,

  // Write back stage also used for load instruction
  output reg [0:6] WB_reg_write_addr,
  output reg [0:127] WB_reg_write_data,
  output reg WB_reg_write_en,

  // branch new PC
  output reg [0:8] new_PC,
  output reg branch_taken,
  output reg is_branch,
  output reg flush_instr2_even, // goes to Even pipe

  // preload signals
  input preload_LS_en,
  input [0:14] preload_LS_addr,
  input [0:127] preload_LS_data
);

`include "opcode_package.vh"

// [0:2] unit ID, [3:130] 128-bit result, [131:137] reg_dst, [138:141] latency, [142] RegWr
reg [0:142] packed_1stage;
wire [0:8] new_PC_result; // new PC from branch unit
wire [0:8] temp_in_PC = current_PC + 2; // used for branch taken
reg [0:127] result; // used for permute or branch

wire [0:127] PERM_result, branch_rt_result, LS_data_result;
wire [0:14] addr_result;

reg LS_write_en;
wire branch_taken_1stage;
```

```
// Permute
PERM_ALU PERM_inst(
  .instr_id(instr_id),
  .ra_data(ra_data),
  .rb_data(rb_data),
  .rc_data(rc_data),
  .imme7(imme7),
  .imme10(imme10),
  .imme16(imme16),
  .imme18(imme18),
  .result(PERM_result)
);

// Branch
BRANCH_ALU BRANCH_inst(
  .instr_id(instr_id),
  .rc_data(rc_data),
  .imme16(imme16),
  .in_PC(current_PC),
  .PC_result(new_PC_result),
  .rt_result(branch_rt_result),
  .branch_taken(branch_taken_1stage)
);

// LS
LS_ALU LS_inst(
  .instr_id(instr_id),
  .ra_data(ra_data),
  .imme10(imme10),
  .imme16(imme16),
  .addr_result(addr_result)
);

always @(*) begin
  case (instr_id)
    `instr_ID_stqa: LS_write_en = 1'b1;
```

```
    `instr_ID_stqd: LS_write_en = 1'b1;
    default: LS_write_en = 1'b0;
  endcase
end

// LocalStore
LocalStore LSmem_inst(
  .clk(clk),
  .rst(rst),
  .LS_write_en(LS_write_en),
  .LS_addr(addr_result),
  .LS_data_in(rc_data), // from ID stage, it should place rt data here
  .LS_data_out(LS_data_result),
  // preload
  .preload_LS_en(preload_LS_en),
  .preload_LS_addr(preload_LS_addr),
  .preload_LS_data(preload_LS_data)
);

always @(*) begin
  if (instr_id == 7'd85 | instr_id == 7'd87) begin
    packed_1stage = 143'b0;
  end else begin
    case (unit_id)
      3'b101: result = PERM_result; // permute result
      3'b110: result = LS_data_result;  // load result (from ls)
      3'b111: result = branch_rt_result;
      default: result = 128'd0;
    endcase
    packed_1stage = {unit_id, result, reg_dst, latency, reg_wr};
  end
end

always @(posedge clk or posedge rst) begin
  if (rst) begin
    packed_1stage <= 0;
    packed_2stage <= 0;
```

```
  packed_3stage <= 0;
  packed_4stage <= 0;
  packed_5stage <= 0;
  packed_6stage <= 0;
  packed_7stage <= 0;
end
else if (flush) begin
  is_branch <= 1'b0;
  branch_taken <= 1'b0;

  packed_2stage <= 0;
  packed_3stage <= 0;
  packed_4stage <= packed_3stage;
  packed_5stage <= packed_4stage;
  packed_6stage <= packed_5stage;
  packed_7stage <= packed_6stage;
  WB_reg_write_addr <= packed_7stage[131:137];
  WB_reg_write_data <= packed_7stage[3:130];
  WB_reg_write_en <= packed_7stage[142];
end
else begin
flush_instr2_even <= 1'b0; // default value
  if(unit_id == 3'b111) begin
    new_PC <= new_PC_result - 2;
    branch_taken <= branch_taken_1stage;
    is_branch <= 1'b1;
    if(instr1_branch && branch_taken_1stage) begin
      flush_instr2_even <= 1'b1;
    end
  end else begin
    is_branch <= 1'b0;
    branch_taken <= 1'b0;
  end

  packed_2stage <= packed_1stage;
  packed_3stage <= packed_2stage;
  packed_4stage <= packed_3stage;
```

```
    packed_5stage <= packed_4stage;
    packed_6stage <= packed_5stage;
    packed_7stage <= packed_6stage;
    WB_reg_write_addr <= packed_7stage[131:137];
    WB_reg_write_data <= packed_7stage[3:130];
    WB_reg_write_en <= packed_7stage[142];
  end
end

endmodule
```

Appendix L (LocalStore.v)

```
module LocalStore(
  input clk,
  input rst,
  input LS_write_en,

  input [0:14] LS_addr,
  input [0:127] LS_data_in,

  output reg [0:127] LS_data_out,

  input preload_LS_en,
  input [0:14] preload_LS_addr,
  input [0:127] preload_LS_data
);
reg [0:7] LS_mem [0:32767];


integer i;
  // Asynchronous read
  always @(*) begin
      for (i = 0; i < 16; i = i + 1) begin
          LS_data_out[i*8 +: 8] = LS_mem[LS_addr + i];
      end
  end
```

```
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      if (preload_LS_en) begin
          for (i = 0; i < 16; i = i + 1) begin
            LS_mem[preload_LS_addr + i] <= preload_LS_data[i*8 +: 8];
          end
      end else begin
        for (i = 0; i < 32768; i = i + 1) begin
            LS_mem[i] <= 8'b0;
        end
      end
    end
    else begin
      if (LS_write_en) begin
        for (i = 0; i < 16; i = i + 1) begin
          LS_mem[LS_addr + i] <= LS_data_in[i*8 +: 8];
        end
      end
    end
  end


endmodule
```

Appendix M (parser.cpp)

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <bitset>
#include <unordered_map>
#include <vector>
#include <cctype>
#include <iomanip>
```

```cpp
using namespace std;

/*
 * Convert an integer 'value' to a binary string of length 'bits'.
 * Zero-pads (or truncates) on the left as needed.
 */
// value may be negative. bits is the field width you need (7,10,16,18,…)
string toBinary(int value, int bits) {
    unsigned int u;
    if (value >= 0) {
        u = static_cast<unsigned int>(value);
    } else {
        // map negative value into its two s complement in bits width
        u = (1u << bits) + value;
    }
    string out;
    out.reserve(bits);
    for (int i = bits - 1; i >= 0; --i) {
        out.push_back(((u >> i) & 1) ? '1' : '0');
    }
    return out;
}

/*
 * 6 instruction formats:
 *   1) RR
 *   2) RRR
 *   3) RI7
 *   4) RI10
 *   5) RI16
 *   6) RI18
 */
enum class FormatType {
    RR,
    RRR,
    RI7,
    RI10,
```

```
    RI16,
    RI18
};

/*
 * For each instruction, store:
 *   - which format it uses,
 *   - its opcode (integer).
 * The opcode size depends on the format (e.g., 4 bits for RRR, 11 bits for RR/RI7, etc.).
 * But we'll store it as an int and rely on the assembler function to take the correct bits.
 */
struct InstructionInfo {
    FormatType format;
    int opcode;  // e.g. 11-bit opcode stored as decimal
};

/*
 * 1) RR format (32 bits total)
 *    Bits [0..10]   = opcode (11 bits)
 *    Bits [11..17]  = RB     (7 bits)
 *    Bits [18..24]  = RA     (7 bits)
 *    Bits [25..31]  = RT     (7 bits)
 */
string assembleRR(int opcode, int rt, int ra, int rb) {
    // According to your layout: [OP(11)][RB(7)][RA(7)][RT(7)]
    // If your assembly is "RR $RT, $RA, $RB", then parse them in that order,
    // but place them in the bit string as specified.
    string opBin = toBinary(opcode, 11);
    string rbBin = toBinary(rb, 7);
    string raBin = toBinary(ra, 7);
    string rtBin = toBinary(rt, 7);
    return opBin + rbBin + raBin + rtBin; // 11 + 7 + 7 + 7 = 32
}

/*
 * 2) RRR format (32 bits)
 *    Bits [0..3]   = opcode (4 bits)
```

```
 *    Bits [4..10]  = RT     (7 bits)
 *    Bits [11..17] = RB     (7 bits)
 *    Bits [18..24] = RA     (7 bits)
 *    Bits [25..31] = RC     (7 bits)
 */
string assembleRRR(int opcode, int rt, int rb, int ra, int rc) {
    // [OP(4)][RT(7)][RB(7)][RA(7)][RC(7)]
    string opBin = toBinary(opcode, 4);
    string rtBin = toBinary(rt, 7);
    string rbBin = toBinary(rb, 7);
    string raBin = toBinary(ra, 7);
    string rcBin = toBinary(rc, 7);
    return opBin + rtBin + rbBin + raBin + rcBin; // 4 + 7 + 7 + 7 + 7 = 32
}

/*
 * 3) RI7 format (32 bits)
 *    Bits [0..10]   = opcode (11 bits)
 *    Bits [11..17]  = I7     (7 bits)
 *    Bits [18..24]  = RA     (7 bits)
 *    Bits [25..31]  = RT     (7 bits)
 */
string assembleRI7(int opcode, int imm7, int ra, int rt) {
    // [OP(11)][I7(7)][RA(7)][RT(7)]
    string opBin = toBinary(opcode, 11);
    string i7Bin = toBinary(imm7, 7);
    string raBin = toBinary(ra, 7);
    string rtBin = toBinary(rt, 7);
    return opBin + i7Bin + raBin + rtBin; // 11 + 7 + 7 + 7 = 32
}

/*
 * 4) RI10 format (32 bits)
 *    Bits [0..7]   = opcode (8 bits)
 *    Bits [8..17]  = I10    (10 bits)
 *    Bits [18..24] = RA     (7 bits)
 *    Bits [25..31] = RT     (7 bits)
```

```
 */
string assembleRI10(int opcode, int imm10, int ra, int rt) {
    // [OP(8)][I10(10)][RA(7)][RT(7)]
    string opBin  = toBinary(opcode, 8);
    string i10Bin = toBinary(imm10, 10);
    string raBin  = toBinary(ra, 7);
    string rtBin  = toBinary(rt, 7);
    return opBin + i10Bin + raBin + rtBin; // 8 + 10 + 7 + 7 = 32
}

/*
 * 5) RI16 format (32 bits)
 *    Bits [0..8]   = opcode (9 bits)
 *    Bits [9..24]  = I16    (16 bits)
 *    Bits [25..31] = RT     (7 bits)
 *
 * (We assume your doc meant bits [25..31] = 7 bits for RT, so total = 32 bits.)
 */
string assembleRI16(int opcode, int imm16, int rt) {
    // [OP(9)][I16(16)][RT(7)]
    string opBin  = toBinary(opcode, 9);
    string i16Bin = toBinary(imm16, 16);
    string rtBin  = toBinary(rt, 7);
    return opBin + i16Bin + rtBin; // 9 + 16 + 7 = 32
}

/*
 * 6) RI18 format (32 bits)
 *    Bits [0..6]   = opcode (7 bits)
 *    Bits [7..24]  = I18    (18 bits)
 *    Bits [25..31] = RT     (7 bits)
 *
 * (We assume your doc meant bits [25..31] = 7 bits for RT, so total = 32 bits.)
 */
string assembleRI18(int opcode, int imm18, int rt) {
    // [OP(7)][I18(18)][RT(7)]
    string opBin   = toBinary(opcode, 7);
```

```
    string i18Bin  = toBinary(imm18, 18);
    string rtBin   = toBinary(rt, 7);
    return opBin + i18Bin + rtBin; // 7 + 18 + 7 = 32
}

/*
 * Example lookup table from mnemonic -> {FormatType, opcode}.
 * You will fill this with real instructions (e.g. "AND", "OR", "ADD", "ADDI", etc.)
 * and the correct opcode bits (in decimal).
 */
unordered_map<string, InstructionInfo> instructionMap = {
    // Example: "AND" uses RR format with an 11-bit opcode of 0x123 (binary 100100011).
    // Just a placeholder. Convert your real opcode bits to decimal and put them here.
    {"addx", {FormatType::RR, 0b01101000000}},
    {"ah", {FormatType::RR, 0b00011001000}},
    {"ahi", {FormatType::RI10, 0b00011101}},
    {"a", {FormatType::RR, 0b00011000000}},
    {"ai", {FormatType::RI10, 0b00011100}},
    {"and", {FormatType::RR, 0b00011000001}},
    {"andhi", {FormatType::RI10, 0b00010101}},
    {"andi", {FormatType::RI10, 0b00010100}},
    {"bg", {FormatType::RR, 0b00001000010}},
    {"bgx", {FormatType::RR, 0b01101000011}},
    {"cg", {FormatType::RR, 0b00011000010}},
    {"cgx", {FormatType::RR, 0b01101000010}},
    {"ceqh", {FormatType::RR, 0b01111001000}},
    {"ceqhi", {FormatType::RI10, 0b01111101}},
    {"ceq", {FormatType::RR, 0b01111000000}},
    {"ceqi", {FormatType::RI10, 0b01111100}},
    {"cgth", {FormatType::RR, 0b01001001000}},
    {"cgthi", {FormatType::RI10, 0b01001101}},
    {"cgt", {FormatType::RR, 0b01001000000}},
    {"cgti", {FormatType::RI10, 0b01001100}},
    {"clz", {FormatType::RR, 0b01010100101}},
    {"eqv", {FormatType::RR, 0b01001001001}},
    {"xor", {FormatType::RR, 0b01001000001}},
    {"xorhi", {FormatType::RI10, 0b01000101}},
```

```
{"xori", {FormatType::RI10, 0b01000100}},
{"ila", {FormatType::RI18, 0b0100001}},
{"ilh", {FormatType::RI16, 0b010000011}},
{"ilhu", {FormatType::RI16, 0b010000010}},
{"il", {FormatType::RI16, 0b010000001}},
{"iohl", {FormatType::RI16, 0b011000001}},
{"nand", {FormatType::RR, 0b00011001001}},
{"nor", {FormatType::RR, 0b00001001001}},
{"or", {FormatType::RR, 0b00001000001}},
{"orhi", {FormatType::RI10, 0b00000101}},
{"ori", {FormatType::RI10, 0b00000100}},
{"selb", {FormatType::RRR, 0b1000}},
{"sfx", {FormatType::RR, 0b01101000001}},
{"sfh", {FormatType::RR, 0b00001001000}},
{"sfhi", {FormatType::RI10, 0b00001101}},
{"sf", {FormatType::RR, 0b00001000000}},
{"sfi", {FormatType::RI10, 0b00001100}},
{"roth", {FormatType::RR, 0b00001011100}},
{"rothi", {FormatType::RI7, 0b00001111100}},
{"rot", {FormatType::RR, 0b00001011000}},
{"roti", {FormatType::RI7, 0b00001111000}},
{"shlh", {FormatType::RR, 0b00001011111}},
{"shlhi", {FormatType::RI7, 0b00001111111}},
{"shl", {FormatType::RR, 0b00001011011}},
{"shli", {FormatType::RI7, 0b00001111011}},
{"fa", {FormatType::RR, 0b01011000100}},
{"fm", {FormatType::RR, 0b01011000110}},
{"fma", {FormatType::RRR, 0b1110}},
{"fms", {FormatType::RRR, 0b1111}},
{"fnms", {FormatType::RRR, 0b1101}},
{"fs", {FormatType::RR, 0b01011000101}},
{"mpy", {FormatType::RR, 0b01111000100}},
{"mpya", {FormatType::RRR, 0b1100}},
{"mpyi", {FormatType::RI10, 0b01110100}},
{"mpyu", {FormatType::RR, 0b01111001100}},
{"mpyui", {FormatType::RI10, 0b01110101}},
{"mpyh", {FormatType::RR, 0b01111000101}},
```

```
    {"cntb", {FormatType::RR, 0b01010110100}},
    {"absdb", {FormatType::RR, 0b00001010011}},
    {"sumb", {FormatType::RR, 0b01001010011}},
    {"avgb", {FormatType::RR, 0b00011010011}},
    {"rotqbyi", {FormatType::RI7, 0b00111111100}},
    {"rotqby", {FormatType::RR, 0b00111011100}},
    {"rotqbii", {FormatType::RI7, 0b00111111000}},
    {"rotqbi", {FormatType::RR, 0b00111011000}},
    {"shlqbii", {FormatType::RI7, 0b00111111011}},
    {"shlqbi", {FormatType::RR, 0b00111011011}},
    {"shlqbyi", {FormatType::RI7, 0b00111111111}},
    {"shlqby", {FormatType::RR, 0b00111011111}},
    {"lqd", {FormatType::RI10, 0b00110100}},
    {"lqa", {FormatType::RI16, 0b001100001}},
    {"stqa", {FormatType::RI16, 0b001000001}},
    {"stqd", {FormatType::RI10, 0b00100100}},
    {"bra", {FormatType::RI16, 0b001100000}},
    {"brhnz", {FormatType::RI16, 0b001000110}},
    {"brz", {FormatType::RI16, 0b001000000}},
    {"brnz", {FormatType::RI16, 0b001000010}},
    {"brasl", {FormatType::RI16, 0b001100010}},
    {"brsl", {FormatType::RI16, 0b001100110}},
    {"br", {FormatType::RI16, 0b001100100}},
    {"brhz", {FormatType::RI16, 0b001000100}},
    {"lnop", {FormatType::RR, 0b00000000001}},
    {"nop", {FormatType::RR, 0b01000000001}},
    {"stop", {FormatType::RR, 0b00000000000}},

    // Add more instructions...
};

/*
 * Helper to parse a single operand:
 *   - If it starts with '$', interpret as register (e.g. "$3" => 3).
 *   - If it starts with 'h'/'H', interpret as hex immediate (e.g. "hFF" => 255).
 *   - Otherwise, interpret as decimal (e.g. "42" => 42).
 */
```

```
int parseOperand(const string &operand) {
    if (operand.empty()) return 0;

    // Register?
    if (operand[0] == '$') {
        return stoi(operand.substr(1)); // skip '$'
    }
    // Hex immediate?
    if (operand[0] == 'h' || operand[0] == 'H') {
        return stoi(operand.substr(1), nullptr, 16);
    }
    // Decimal immediate
    return stoi(operand);
}

// Parses "imm($reg)" syntax for D-form
pair<int, int> parseDFormOperand(const string& operand) {
    size_t openParen = operand.find('(');
    size_t closeParen = operand.find(')');
    if (openParen == string::npos || closeParen == string::npos || closeParen <= openParen)
        return {0, 0}; // Default fallback

    string immStr = operand.substr(0, openParen);
    string regStr = operand.substr(openParen + 1, closeParen - openParen - 1);

    int imm = parseOperand(immStr);
    int ra = parseOperand(regStr);
    return {imm, ra};
}

/*
 * Process a single line of assembly: "MNEMONIC operand1, operand2, ..."
 *  - Lookup the mnemonic in instructionMap
 *  - Depending on its FormatType, parse the correct # of operands
 *  - Call the corresponding assemble function
 *  - Return a 32-bit binary string (or empty if error)
 */
```

```
string processInstruction(const string &line, const unordered_map<string, int> &labelMap, int pc) {
    // 1) Extract mnemonic
    istringstream iss(line);
    string mnemonic;
    iss >> mnemonic;
    if (mnemonic.empty()) {
        return ""; // skip empty or invalid line
    }

    // 2) Lookup
    auto it = instructionMap.find(mnemonic);
    if (it == instructionMap.end()) {
        cerr << "Error: Unknown instruction '" << mnemonic << "'\n";
        return "";
    }
    InstructionInfo info = it->second;

    // 3) Collect comma-separated operands
    vector<string> operands;
    string opToken;
    while (getline(iss, opToken, ',')) {
        // Trim whitespace
        size_t start = opToken.find_first_not_of(" \t");
        size_t end   = opToken.find_last_not_of(" \t");
        if (start != string::npos && end != string::npos) {
            operands.push_back(opToken.substr(start, end - start + 1));
        }
    }

    // 4) Dispatch based on FormatType
    switch (info.format) {
    case FormatType::RR: {
        // Expect assembly: e.g. "AND $rt, $ra, $rb"
        // special case for zero-operand instructions
        if (mnemonic=="nop" ||
            mnemonic=="lnop"||
            mnemonic=="stop") {
```

```
        // emit opcode plus all zero registers
        return assembleRR(info.opcode, 0, 0, 0);
    }
    if (operands.size() < 3) {
        cerr << "Error: RR format expects 3 registers.\n";
        return "";
    }
    int rt = parseOperand(operands[0]);
    int ra = parseOperand(operands[1]);
    int rb = parseOperand(operands[2]);
    return assembleRR(info.opcode, rt, ra, rb);
}
case FormatType::RRR: {
    // e.g. "MUL $rt, $rb, $ra, $rc"
    if (operands.size() < 4) {
        cerr << "Error: RRR format expects 4 registers.\n";
        return "";
    }
    int rt = parseOperand(operands[0]);
    int rb = parseOperand(operands[1]);
    int ra = parseOperand(operands[2]);
    int rc = parseOperand(operands[3]);
    return assembleRRR(info.opcode, rt, rb, ra, rc);
}
case FormatType::RI7: {
    // e.g. "MNEMONIC $rt, $ra, imm7"
    if (operands.size() < 3) {
        cerr << "Error: RI7 expects 3 operands (RT, RA, I7).\n";
        return "";
    }
    int rt  = parseOperand(operands[0]);
    int ra  = parseOperand(operands[1]);
    int imm = parseOperand(operands[2]);
    return assembleRI7(info.opcode, imm, ra, rt);
}
case FormatType::RI10: {
    if (operands.size() < 2) {
```

```
            cerr << "Error: RI10 expects 2 operands (RT, I10(ra)).\n";
            return "";
        }
        int rt = parseOperand(operands[0]);

        // Try parsing the D-form syntax
        int imm = 0, ra = 0;
        if (operands[1].find('(') != string::npos) {
            tie(imm, ra) = parseDFormOperand(operands[1]);
        } else {
            if (operands.size() < 3) {
                cerr << "Error: RI10 expects 3 operands if not using D-form.\n";
                return "";
            }
            ra = parseOperand(operands[1]);
            imm = parseOperand(operands[2]);
        }

        return assembleRI10(info.opcode, imm, ra, rt);
    }
    case FormatType::RI16: {
        int rt   = 0;
        int imm  = 0;
        string tok;

        // pick up rt and label/immediate token
        if (operands.size() > 1) {
            rt  = parseOperand(operands[0]);
            tok = operands[1];
        } else {
            tok = operands[0];
        }

        // see if token is a label
        auto itL = labelMap.find(tok);
        if (itL != labelMap.end()) {
            // labelMap value is instruction index
```

```
            int targetIndex = itL->second;
            // original code added 2 to get absolute PC
            int targetPC    = targetIndex + 2;

            if (mnemonic == "bra") {
                // absolute branch
                imm = targetPC;
            } else {
                // PC-relative branch: offset = targetPC - nextPC
                // nextPC is (pc + 1)
                imm = targetPC - (pc + 1);
            }
        } else {
            // not a label, parse as numeric immediate
            imm = parseOperand(tok);
        }

        return assembleRI16(info.opcode, imm, rt);
    }
    case FormatType::RI18: {
        // e.g. "MNEMONIC $rt, imm18"
        if (operands.size() < 2) {
            cerr << "Error: RI18 expects 2 operands (RT, I18).\n";
            return "";
        }
        int rt   = parseOperand(operands[0]);
        int imm  = parseOperand(operands[1]);
        return assembleRI18(info.opcode, imm, rt);
    }
    }

    // Should not reach here
    return "";
}

int main() {
```

```
// read all lines into memory
vector<string> allLines;
{
    ifstream fin("input_assembly.txt");
    if (!fin) { cerr << "Error opening input file\n"; return 1; }
    string raw;
    while (getline(fin, raw))
        allLines.push_back(raw);
}

// map label name to instruction index
unordered_map<string,int> labelMap;
// filtered list of real instructions
vector<string> instLines;

for (auto &raw : allLines) {
    // trim leading plus trailing whitespace
    size_t start = raw.find_first_not_of(" \t");
    if (start == string::npos) continue;
    size_t end = raw.find_last_not_of(" \t");
    string t = raw.substr(start, end - start + 1);

    // skip full-line comments
    if (t.rfind("//", 0) == 0) continue;

    // label if ends with colon
    if (t.back() == ':') {
        string name = t.substr(0, t.size() - 1);
        labelMap[name] = instLines.size();
    }
    else {
        instLines.push_back(t);
    }
}

ofstream fout("output_binary.txt");
if (!fout) { cerr << "Error opening output file\n"; return 1; }
```

```
    for (int i = 0; i < (int)instLines.size(); ++i) {
        string bin32 = processInstruction(instLines[i], labelMap, i);
        if (bin32.size() == 32)
            fout << bin32 << "\n";
        else
            cerr << "line " << i
                 << " gave " << bin32.size()
                 << " bits: " << instLines[i] << "\n";
    }


    return 0;
}
```