Digital System Design Final Project

Pipelined RISC-V Processor

1. Architecture of 5-stage Pipelined RISC-V Processor

In this final project, we successfully designed a pipelined RISC-V processor with synchronous active low reset, along with an instruction cache and a data cache, named RISCV_Pipeline, I_cache, D_cache, respectively. Building on HW2, which featured a single-cycle processor, we extended the design to a 5-stage pipelined architecture. Additionally, since J-type instructions weren't considered in the previous homework, they have been included in this work. Pipeline hazards, which arise when transitioning from a single-cycle to a pipelined CPU, are addressed using forwarding and pipeline stalling techniques.

The overall RISC-V design resembles the one given in the problem description. Below, we list some details that differ from the given design:

- Signals are selected based on whether they need to proceed to the EX stage. This is due to our expectation that the ALU calculation in the EX stage will form the critical path in the overall design.
- Register reads in the ID stage would access the value from `r_w` instead of `r_r`. This modification could eliminate hazards when the instruction in the ID stage needs the same value that is being written back in the WB stage.

Regarding the caches, we take the 2-way version from HW3, and also made some modifications compared to the HW3 version:

- Our caches work fine without a Finite State Machine by taking use of existing flip-flops.
- Additional flip-flops are added to the input and output interfaces to buffer signals coming from the slow memories, preventing potential timing violations caused by excessively long data paths.

2. Hazard Detection and Resolution

2.1. Data Hazard and Forwarding Unit

In a pipelined processor, data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its write-back to the register file. In our 5-stage pipelined RISCV, the result is only written to the register file in the end of WB stage, but subsequent instructions might need the value in the ID or EX stage. If this dependency is not detected and handled correctly, the processor would use outdated data, leading to incorrect execution results.

To resolve this issue and avoid unnecessary pipeline stalls, we implemented a forwarding unit in the **ID stage**. This unit dynamically compares the destination register (rd) of instructions in the EX/MEM and MEM/WB pipeline stages with the source registers (rs1, rs2) of the instruction currently in the ID stage. The conditions are displayed as:

1a. EX/MEM. Rd = ID/EX. Rs1
1b. EX/MEM. Rd = ID/EX. Rs2
2a. MEM/WB. Rd = ID/EX. Rs1
2b. MEM/WB. Rd = ID/EX. Rs2

In ID stage, rs1/rs2 address are decoded and their original data is obtained. For each source register, the forwarding unit determines whether a more recent value is available in later pipeline stages:

- If the register was written by an instruction in EX/MEM, the value is forwarded from C_w (the EX result).
- If the value is not available in EX/MEM but is ready in MEM/WB, it is forwarded from rd_value_w (the value from memory and ready to write back to rd).

The logic is implemented as follows:

```
assign rs1_data = (rs1 == 5'b0) ? 32'b0 : r_w[rs1];
assign rs2_data = (rs2 == 5'b0) ? 32'b0 : r_w[rs2];

assign for1_from_EXE = (rd_Exe2Mem_w != 5'd0 && rd_Exe2Mem_w == rs1);
assign for1_from_MEM = (rd_Mem2WB_w  != 5'd0 && rd_Mem2WB_w  == rs1);
assign rs1_data_for = for1_from_EXE ? C_w : (for1_from_MEM ? rd_value_w : rs1_data);

assign for2_from_EXE = (rd_Exe2Mem_w != 5'd0 && rd_Exe2Mem_w == rs2);
assign for2_from_MEM = (rd_Mem2WB_w  != 5'd0 && rd_Mem2WB_w  == rs2);
assign rs2_data_for = for2_from_EXE ? C_w : (for2_from_MEM ? rd_value_w : rs2_data);
```

Here, `rs1_data_for` and `rs2_data_for` are the final selected values that feed into ALU. These are determined through a mux, with the selection signal derived from the forwarding condition.

2.2. Load-Use Hazard and Stall/NOP Injection

While most data hazards can be resolved via forwarding, load-use hazards require special treatment. This is because in our 5-stage pipeline, the result of a LW instruction is not ready until the MEM stage, but the next instruction may already be entering the EX stage and expecting the result. Here, forwarding can't help since x5 won't be loaded from memory until after EX.

We detect the load-use hazard in **EX stage** by checking if the previous instruction is a LW (i.e., `read_Dec2Exe_r` is high), and if its destination register(`rd_Dec2Exe_r`) is used as either rs1 or rs2 of the current instruction.

This condition is accomplished by the following logic:

```
lw_hazard_w = (read_Dec2Exe_r && (
              (rd_Dec2Exe_r == rs1 && rs1 != 0) || (rd_Dec2Exe_r == rs2 && rs2 != 0))
               && !lw_hazard_r);
```

The extra condition `!lw_hazard_r` ensures that the stall is only active for one cycle, avoiding repeated stalling. Once `lw_hazard_w` is asserted,

- The IF and ID stages are frozen (i.e., they retain the current instruction and PC).
- The ID/EX stage is cleared, injecting a NOP to delay execution.

LW x14, 4(x11)	IF	ID	EX lw hazard	ME	WB			
SLT x15, x13, x14		IF	ID	ID	EX	ME	WB	
BEQ x15, x0, Exit			IF	IF	ID	EX	ME	WB

This effectively inserts a bubble into the pipeline and lets the LW result arrive before the dependent instruction resumes execution.

2.3. Control Hazard and Unconditional Jump Handling

JAL and JALR instructions unconditionally change the control flow by setting PC to target address. Owing to the pipelined structure, subsequent instructions are also fetched and processed before jumping. This leads to incorrect execution or wasted cycles.

Our jump decision is made at **EX stage**. In ID stage, JAL/JALR instructions are identified and signaled through `jal_Dec2Exe_w` and jalr_Dec2Exe_w, which are latched and carried to the EX stage.

```
assign jal_taken  = jal_Dec2Exe_r;

assign jalr_taken = jalr_Dec2Exe_r;
```

Once either `jal_taken` or `jalr_taken` is detected, the IF stage updates the next PC with the computed jump target C_w, which is the result of A + B from the ALU in EX stage:

```
else if (jal_taken || jalr_taken) begin

    pc_w = C_w;

end
```

Simultaneously, ID stage is flushed into the execution path to avoid unintended execution.

3. Extension Features

3.1. Branch Prediction

Since the outcome of the branch (taken or not taken) is only determined after EX stage, the processor risks fetching the wrong instruction in the IF stage and wasting cycles. To reduce such penalties, branch prediction is implemented. In this section, we introduce the Branch Prediction Unit (BPU) we designed and compare its performance with a always-not-taken strategy.

Our BPU adopts a 2-level global history predictor architecture. It consists of two main components, 2-bit Global History Register (GHR) and a 4-entry Pattern History Table (PHT). GHR (2 bits) records the outcomes of the most recent branches, while PHT is an array of 2-bit finite state machines indexed by GHR.

Global History Register: NT

Pattern History Table

History	Pattern
NN (00)	strongly NT (01)
NT (01)	weakly T (10)
TN (10)	weakly NT (00)
TT (11)	strongly T (11)

state pattern

00: weakly not taken

01: strongly not taken

10: weakly taken

11: strongly taken

Our BPU predict branch direction in IF stage, and corrects mispredictions in EX stage if needed. In the **IF stage**, we determine if the current instruction is a branch via decoding ICACHE_rdata. If the instruction is a branch, branch prediction is generated by BPU. The prediction result is latched and propagated to the decode and execution stage.

Once the actual branch condition is resolved in **EX stage**, we determine if the earlier prediction was wrong. If the prediction is wrong, the following correction logic is triggered in BPU:

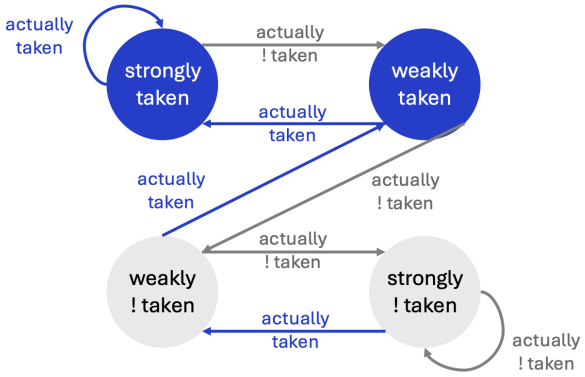
- Update GHR by shifting in the actual result
- Update PHT state based on standard 2-bit FSM transitions

Global History Register: ~~NT~~ → NN

Pattern History Table

History	Pattern
NN	strongly NT
NT	~~weakly T~~
TN	weakly NT
TT	strongly T

weakly NT



At the same time, PC in IF is updated and ID stage is flushed to discard any incorrectly fetched instructions.

To evaluate the effectiveness of our 2-level predictor in the test case, we compare its performance against a simpler always-not-taken baseline. In this strategy, the processor always assumes that conditional branches are not taken, and only corrects the PC when the branch is actually taken in EX stage.

	Area (um2)	Qsort time (ns)	Conv time (ns)	A*T*T
2-level global	477188	403040	58727	1.129e+16
always not taken	449639	403012	58727	1.064e+16

The result shows that the miss rate and execution time of the two strategy is quite similar. However, by removing the BPU module in always-not-taken, the overall area dropped from 477,188 μm² to 449639 μm², giving an evident area and A*T*T reduction.

3.2. C-instructions

Our RISC-V processor is also capable of handling C-instructions, also known as Compressed instructions. The following are all the instructions that we support:

C.ADD	C.ANDI	C.LW	C.J
C.MV	C.SLLI	C.SW	C.JAL
C.ADDI	C.SRLI	C.BEQZ	C.JR
C.NOP	C.SRAI	C.BNEZ	C.JALR

We revise the logic related to the program counter(pc), which prevents us from using a FSM to record the current state and a data buffer to stall pre-read instruction pieces. Only one flip-flop, `half_instr`, is added to manage pc control. The function works as follows: The value of the flip-flop `half_instr` indicates whether the current input includes the second half of a 32-bit instruction. This value is derived from the opcode of a compressed instruction, specifically the [1:0] slice of the instruction. If this slice is not 11, then `half_instr` is set to be 1.

In the following description, we use `half_instr_w` to represent the current value and `half_instr_r` for the value from the previous cycle. If pc[1] is True, It means we are reading the least significant 16 bits of the input. In this case, we store these bits in instr, regardless of

whether it is a half-read instruction. If `half_instr_r` is True, we are dealing with the second half of a 32-bit instruction. Therefore, we combine the first half, which was stored in the instr previously, with the second half, which is the input for the current cycle. This approach eliminates the need for additional flip-flops to buffer the half-read instructions.

As for the Dec stage, we do not build a separate decoder or translator for compressed instructions. Instead, a multiplexer (MUX) is applied to each signal in advance to avoid large arear overhead in the design. Below are some examples:

```
assign rs2_norm  = cmp ? instr_r[30:26] : {instr_r[0],  instr_r[15:12]};
assign rs2_prime =  instr_r[28:26];
assign rs2 = (cmp_rs) ? {2'b01, rs2_prime} : rs2_norm;
```

Here, `_norm` represents the normal 5-bit format, `_prime` the 3-bit compressed format, and the one without suffix is the final selected result.

The cases for compressed instructions are integrated directly into the conditions of the case statement that decode the original instructions, rather than being handled by a separate decoding block. This avoids duplication and keeps the logic compact. Additionally, question marks (?) are also used in the case statement to enable synthesis tools like Design Compiler to better to better optimize the logic by treating "don't care" bits flexibly. For example:

```
opcode_lw, 7'b??01000: begin
            A_w = rs1_data_for;
            B_w = imm_lw;
            rd_Dec2Exe_w = rd;
            ALU_op_w = func3_add;
            read_Dec2Exe_w = 1'b1;

end
```

In this case, we can see that compressed and normal instructions formats are merged into a single statement, and the use of question marks increases flexibility for synthesis optimization.

3.3. Multiplication

To implement the instruction MUL, a multiplier is essential. Therefore, we implemented a 32-bit Dadda multiplier, applying the concepts we learned in class. However, directly code a 32-bit Dadda multiplier can be challenging due to the numerous iterations required and the complex adding rules. To address this issue, we wrote a piece of Python code to automatically generate the correct architecture for the Dadda module. The detailed steps are provided in the following.

Step1. Partial Product Generation

- Compute all bitwise products between multiplicand and multiplier (e.g., Aiii & Bjjj)
- Only handle the lower 33 columns (bit positions 0–32)
- Each dot s_i_k_l represents a partial product at column k = i + j – 1, on layer l

Step2. Initial Column Heights

- Simulate initial number of bits (dots) in each column using an array
- Example: initial = [1, 2, 3, 4, 5, 6, 7, 8, 7, ..., 1]
- Determine Dadda height per layer: [2, 3, 4, 6, 9, 13, 19, 28]

Step3. Dadda Compression Using Adders

- In each layer, reduce column height to allowed maximum d using:
  Full adders (3 → 2 compression)
  Half adders (2 → 2 compression)
- Apply half adders only when height − d = 1
- Carry outputs are shifted to the next column

While adding the partial sums together, a special distribution rule is applied. In order to shorten the critical path, we place the newly generated sum or carry-out in the back of the queue, and always pick 2 bits (for a Half Adder) or 3 bits (for a Full Adder) from the front of the queue in the next iteration. This approach helps avoid situations where a bit is constantly used in every iteration, which would otherwise cause an excessively long critical path. As we can see from the result below, for inputs A and B, and output C, the critical path is from A[18] to C[29], rather than from A[1] to C[31], which would be the worst case scenario.

```
-----------------------------------------------------------
clock CLK (rise edge)                    0.00       0.00
clock network delay (ideal)              0.50       0.50
A_r_reg[18]/CK (DFFHQX4)                 0.00       0.50 r
A_r_reg[18]/Q (DFFHQX4)                  0.14       0.64 f
U1908/Y (BUFX8)                          0.12       0.76 f
U1774/Y (NAND2X2)                        0.15       0.91 r
U1530/Y (OAI2BB1XL)                      0.24       1.15 r
U3828/Y (OAI21X2)                        0.09       1.24 f
U2135/S (ADDFX2)                         0.39       1.63 f
U1441/Y (OAI21X1)                        0.18       1.82 r
U3873/Y (OAI21X1)                        0.12       1.93 f
U4054/S (ADDFX2)                         0.39       2.32 f
U1412/Y (XNOR2X2)                        0.13       2.45 r
U4072/Y (XNOR2X4)                        0.14       2.59 f
U4075/Y (NAND2X2)                        0.13       2.72 r
U4092/Y (NAND2X4)                        0.05       2.77 f
U4093/Y (XNOR2X4)                        0.09       2.86 f
U1374/Y (INVX2)                          0.10       2.96 r
U4097/Y (NAND3X6)                        0.09       3.06 f
U1364/Y (NAND2X1)                        0.15       3.20 r
U1995/Y (XNOR2X2)                        0.12       3.32 r
U1994/Y (OR2X4)                          0.13       3.46 r
U2091/Y (INVX3)                          0.06       3.52 f
U4284/Y (NOR2X2)                         0.11       3.63 r
U4285/Y (XNOR2X4)                        0.13       3.76 f
U4286/Y (NOR2X4)                         0.06       3.82 r
C_r_reg[29]/D (DFFHQX4)                  0.00       3.82 r
data arrival time                                   3.82
```

One unexpected outcome is that multiplication, after implementing the Dadda multiplier, can be performed in a single cycle. However, the original goal of implementing the multiplier ourselves was enable further pipelining. Given this, we can consider simplifying the process by using the "*" operator in the Verilog code and delegating the task of optimizing to the Design Complier in the future.

4. Cache Design and Optimization

To reduce area and miss rate, we have already applied 2-way set-associative strategy for I-cahce and D-cache. I-cache is read-only type. The details of cache structures are listed below.

I-cahce:
Read-only (no write logic required)
2-way set associativity

Block size: 128 bits = 16 bytes = 4 instructions

Replacement strategy: LRU by shifting way 1 to way 0 on new block fetch

D-cache:

2-way set associativity

Block size: 128 bits = 16 bytes = 4 data words

Write policy: Write-allocate + Write-back

Replacement strategy: pseudo-LRU (similar to I-cache)

For our official submission, we used a configuration with **4-block I-cache** and **8-block D-cache**. However, to further evaluate the effect of cache capacity on area and time, we experiment with different numbers of blocks for both I-cache and D-cache, while keeping the block size and associativity fixed. Specifically, we compare configurations with 4, 8, and 16 sets. A comparison table summarizing how these changes affect execution time, synthesis area, and the resulting A*T*T is provided below.

Cache blocks combination	Area (um2)	Cycle time (ns)	Qsort time (ns)	Conv time (ns)	A*T*T
I-cache 16 D-cache 16	816367	3.1	270471	38247	8.445e+15
I-cache 8 D-cache 8	513499	3.1	343681	49352	8.709e+15
I-cache 4 D-cache 4	371134	2.8	428968	75975	1.209e+16
I-cache 4 D-cache 8	477188	2.8	403040	58727	1.129e+16
I-cache 8 D-cache 16	664421	3.1	321807	40089	8.571e+15

As the result, instead of the one we submitted, the combination of 16-block I-cache and 16-block D-cache shows the best performance. Additionally, during our experiments, we found that Conv program prefers larger D-cache, while Qsort perform better at larger I-cache.

5. Testbench Timing Adjustment to Resolve Post-Synthesis Reset Issue

During post-synthesis gate-level simulation, we encountered a critical issue that our post-sim result failed despite of met timing during synthesis. Upon investigation, we realized that the cause was a mismatch between the reset timing in the testbench and the timing constraints (.sdc) applied during synthesis. This led to functional errors because some flip-flops were not properly reset.

Specifically, our .sdc file defines different input delays for regular signals and for the reset signal `rst_n`:

```
set t_long [expr {$cycle / 2.0 + 0.1}]
set t_short  0.1
set_input_delay $t_long -clock CLK [all_inputs except rst_n]
set_input_delay $t_short -clock CLK [get_ports rst_n]
```

This indicates that the synthesis tool expects `rst_n` to arrive 0.1 ns before the active clock edge. However, in our original testbench, the reset was asserted relatively late:

```
#(`CYCLE*1.6) rst_n = 1'b0;
```

This caused the reset signal to arrive after the expected time, leading to flip-flops ignoring it in post-simulation. To resolve this, we adjusted the reset delay to raise `rst_n` earlier, just after the first full clock cycle and with a 0.1 ns delay to align with the .sdc:

```
#(`CYCLE*1)
#(0.1) rst_n = 1'b0;
```

This change ensures that the reset signal reaches flip-flops within the expected timing, making it effective in gate-level simulation.

6. Conclusion and Future Work

In our submission, we implemented a 2-level global history branch predictor, a Dadda multiplier, a 4-block I-cache, and an 8-block D-cache, both using 2-way set associativity. The result is presented as below.

Area (um2)	Qsort time (ns)	Conv time (ns)	A*T*T
477188	403040	58727	1.129e+16

Due to time constraints, we were unable to fully realize all aspects of our ideal design. Several key adjustments, such as branch prediction and cache block size, have been analyzed in previous sections. There are several directions we can explore to further enhance processor design:

- Synthesis Optimization: We plan to explore more advanced features of the synthesis tools, such as leveraging built-in multipliers. Additionally, we can experiment with different synthesis commands to improve area-time performance.
- Memory Timing Behavior: A deeper understanding of the slow memory could help us better tune timing parameters. In this particular case, the fixed stall duration is crucial.
- Cache Enhancement: We aim to expand the preload function of caches and redesign the stall handling mechanism to minimize memory wait time.

Through these improvements, we hope to further optimize our processor design in the future.