Computer Architecture
Lab 2: Pipelined RISC-V Processor

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

revision: 9-17-2024

In this lab, we will explore various implementations of the 32-bit RISC-V Instruction Set Architecture (ISA). The focus will be on implementing the `rv32i` ISA, a diagnostic instruction (`CSRW`), and a subset of the `rv32m` ISA. These components are essential for running a C program without requiring system calls. A detailed description of the instructions to be implemented can be found in the provided `riscv-isa.txt` within the lab tarball.

Your task for this lab involves taking a partially implemented, 5-stage pipelined RISC-V processor with stalling and expanding it into a more advanced version that incorporates bypassing. The provided reference processor comes with a completed RISC-V datapath, but its control unit currently supports only a limited set of instructions. You will be responsible for extending the control unit to fully support the `rv32i` and `rv32m` instruction sets, incorporating pipelined controls and adding bypassing mechanisms to enhance performance.

Pipelining allows for greater performance than a multi-cycle implementation (where one pipeline stage is executed per cycle without pipelining) by reducing the cycles per instruction (CPI) without significantly affecting the clock cycle time. By enabling multiple instructions to be processed simultaneously, pipelining increases overall throughput. However, this introduces challenges related to data and control hazards, which complicate the control logic. While the reference processor uses stalling and squashing techniques to handle these hazards, your task will involve implementing an additional optimization: bypassing. This will further improve performance, though it comes with its own set of trade-offs that you will explore in your report.

This lab will provide you with experience in:

- Understanding and working with instruction set architectures (ISAs).
- Exploring the fundamentals of pipelined processor microarchitecture.
- Applying microarchitectural techniques to handle data and control hazards.

1 Reminder

- **Submission Deadline:** 9/30 (Mon) 11:59 p.m.
- Please refer to Lab 0 for the academic integrity requirements.
- **No sharing or distribution of lab materials is allowed.**

2 Setup

2.1 Getting Started

Once you have the Lab 2 materials, extract them by entering the following commands:

```
% tar -xf lab2.tar
% cd lab2
% export LAB2_ROOT=$PWD
```

Please modify imuldiv-IntDivIterative.v and imuldiv-IntMulIterative.v in lab2/imuldiv by replacing them with your implementations of imuldiv-IntDivIterative.v and imuldiv-IntMulIterative.v from lab1.

Within the lab root directory, you will find eight subdirectories, each serving a specific purpose:

- `build`: Contains the Makefile and compiled code.
- `imuldiv`: Houses all Verilog source code.
- `riscvstall`: Pipelined RISC-V processor with stalling source code
- `riscvbyp`: Pipelined RISC-V processor with bypassing source code
- `riscvlong`: Pipelined RISC-V processor with bypassing and pipelined muldiv unit source code
- `tests`: Assembly test build system
- `ubmark`: Benchmarks for evaluation
- `vc`: Contains additional Verilog components.

The `tests` directory includes a build system designed to compile and convert assembly tests into Verilog memory hex (.vmh) files, which can then be used to initialize the processor's memory. Within the `riscv` subdirectory, you will find assembly tests that verify each instruction in the ISA, as well as a few tests for pseudo-instructions. These pseudo-instructions are not implemented in hardware but are translated by the compiler or assembler into one or more actual instructions.

It is crucial to test each newly implemented instruction using its corresponding assembly test before proceeding to the next one. Refer to Section 4 for detailed testing procedures. **Additionally, ensure that you compile the assembly tests as outlined in Section 4.2 before running them.**

The directories `riscvstall`, `riscvbyp`, and `riscvlong` contain the source code for the pipelined RISC-V processor used in this lab. `riscvstall` is for the processor with stalling only, `riscvbyp` includes bypassing, and `riscvlong` introduces a multicycle, pipelined multiply/divide unit.

2.2 Building the Project

Before building the project, ensure you have completed the environment setup as outlined in Lab 0.

Next, let's compile the tests by following the instructions provided in Section 4.2.

```
% cd $LAB2_ROOT/tests
% mkdir build && cd build
% ../configure --host=riscv32-unknown-elf
% make
% ../convert
```

After compiling the tests, we can proceed to compile the reference processors and run the RISC-V assembly tests:

```
% cd $LAB2_ROOT/build
% make
% make check
% make check-asm-riscvstall
% make check-asm-rand-riscvstall
% make check-asm-riscvbyp
% make check-asm-rand-riscvbyp
% make check-asm-riscvlong
% make check-asm-rand-riscvlong
```

Running `make` will compile the processor simulators. The simulators **without `-rand-`** use synchronous memory with a fixed 1-cycle response time, while those **with `-rand-`** use synchronous memory that introduces random delays. To get a better understanding, open the Makefile in your preferred text editor. Look for the section titled "List of Assembly Tests." Here, the `tests` variable defines the list of assembly tests to be executed. If you have additional tests to run, append them to this list.

To use the simulator without random delays, execute `make check-asm-riscvstall`. Alternatively, to introduce random delays, run `make check-asm-rand-riscvstall`, which utilizes `riscvstall-randdelay-sim` instead of `riscvstall-sim`.

You can also execute individual assembly tests by directly invoking the simulator. For example, to run the `riscv-addi` test alone:

```
% cd $LAB2_ROOT/build
% ./riscvstall-sim +exe=../tests/build/vmh/riscv-addi.vmh +stats=1
```

To display simulation statistics, include the `+stats=1` option (off by default). Additionally, you can generate waveform data by adding the `+vcd=1` option, which produces a `.vcd` file viewable in `gtkwave`. Note that successive simulations will overwrite this file, so rename it if you need to keep the data.

The Makefile uses `riscvstall-sim` and `riscvstall-randdelay-sim` binaries to execute the assembly tests for `make check-asm-riscvstall` and `make check-asm-rand-riscvstall`, respectively. The `riscvstall` simulators utilize the stall-based processor, while the `riscvbyp` versions employ the bypass-based processor. Initially, both simulators use the same stall-based processor, as they share the same source code. Once you implement bypassing, you will be able to run both stall and bypass versions.

Running commands like `make check-asm-*`, `make check-asm-rand-*`, or `make run-bmark-*` will generate a `.vcd` dump for each assembly test, named according to the test. For instance, the `riscv-addi.vmh` test generates a `riscv-addi.vcd` file. Random delay simulations add a `-rand` suffix to the dump file name (e.g., `riscv-addi-rand.vcd`).

All `.vcd` dumps will be located in either `$LAB2_ROOT/tests/build/vmh` or `$LAB2_ROOT/bmark/build/vmh`. Be aware that running the automatic tests for `riscvstall`, `riscvbyp`, and `riscvlong` consecutively will overwrite existing `.vcd` files.

3 Pipelined 5-Stage RISC-V Processor with Bypassing

In this lab, you will work with a fully implemented RISC-V datapath that uses stalling and a partially completed control unit. The datapath incorporates the iterative multiply/divide unit, which functions the same as the one you designed in Lab 1.

The lab has three main objectives:

- Extend the control unit to support more instructions.
- Implement bypassing in both the datapath and control unit.
- Integrate a pipelined multiply/divide unit.

Start with **Objective 1** in the `riscvstall` directory. Once completed, copy your source files to the `riscvbyp` directory to work on **Objective 2**. Rename all files with the `riscvbyp-` prefix and update any `‘include` statements accordingly. For **Objective 3**, move to the `riscvlong` directory, changing filenames and paths as needed. It is crucial to have both the stalling and bypassing processors available for evaluation.

3.1 Objective 1: Enhancing the Control Unit

In this objective, you will only modify the control unit (`riscv-CoreCtrl.v`). All necessary signals from the datapath are already available as inputs, and the required control signals are defined as outputs. Your task is to add entries to the control output table for each RISC-V instruction. The table columns representing control signals are predefined; there is no need to add more. While additional helper signals can be introduced, they are typically only necessary for branch instructions. Carefully review the `localparams` for the control signals, as they will be essential for populating the new entries in the table.

The processor uses a 5-stage pipeline: Fetch (F), Decode (D), Execute (X), Memory (M), and Writeback (W). Most signals in the datapath and control unit are suffixed with `_Fhl`, `_Dhl`, `_Xhl`, `_Mhl`, or `_Whl`, indicating the stage where they are used. The suffix `hl` means the signal is valid for one cycle after the clock edge. We will use flip-flop-based design throughout this lab, so only the `hl` suffix is relevant.

Below are the **rv32i** and **rv32m** instructions to be implemented. Refer to `riscv-isa.txt` for detailed descriptions.

Already Implemented: Minimal Subset for Assembly Tests

- Register-Immediate Arithmetic: `addi, ori, lui, auipc`
- Register-Register Arithmetic: `add`
- Memory: `lw, sw`
- Jump: `jal`
- Branch: `bne, blt`
- Diagnostic: `csrw`

rv32im: Simplest Subset for Running Raw C Code (No Syscalls)

- Register-Immediate Arithmetic: `andi, xori, slli, srli, srai, slti, sltiu`
- Register-Register Arithmetic: `sub`, `slt`, `sltu`, `sll`, `srl`, `sra`, `and`, `or`, `xor`
- Memory: `lb, lbu, lh, lhu, sb, sh`
- Jump: `jalr`
- Branch: `beq, bge, bltu, bgeu`
- Diagnostic: `mul, div, divu, rem, remu`

The control unit used in this lab is different from a traditional multi-cycle processor. Instead of a finite state machine (FSM), we pipeline control signals to the appropriate datapath stages. Each row in the control table represents the set of control signals for a given instruction, not a specific state. A summary of each control signal is provided in Table 1, corresponding to the datapath controls shown in Figure 1. **Understand the interaction between controls and the datapath before adding new instructions.**

INST_VAL	signifies a valid instruction, used for assertions
J_EN	is there a jump that can be determined in decode? (i.e. jal, jr)
BR_SEL	type of branch instruction, used to determine which branch conditions need to be checked to see if branch is taken
PC_SEL	PC mux select
OP0_SEL	operand 0 mux select
RS_EN	is operand 0 being read from the regfile? Used to determine when to stall or bypass
OP1_SEL	operand 1 mux select
RT_EN	is operand 1 being read from the regfile? Used to determine when to stall or bypass
ALU_FN	ALU function
MULDIV_FN	muldiv unit function
MULDIV_EN	muldiv request valid
MULDIV_SEL	muldiv response mux select, used to choose lower or upper 32-bits of result
EX_SEL	execute stage output mux select, used to choose between ALU output or muldiv output
MEM_REQ	type of memory request
MEM_LEN	length of memory request in bytes – use 0 for word, 1 for byte, and 2 for halfword
MEM_SEL	memory response mux select, used to choose unsigned or signed subword memory responses, if necessary
WB_SEL	writeback mux select, used to choose execute stage output or memory response
RF_WEN	regfile write enable
RF_WADDR	regfile write address
CSR_WEN	csr write enable

Table 1: Summary of control signals.

All control signals are set in the **D** (Decode) stage and are pipelined to the stage where they are needed. For example, the signal `alu_fn_Dhl` is set in Decode and pipelined to the Execute stage as `alu_fn_Xhl` to drive the ALU.

Instruction encodings, fields, and control signal fields are defined in the header file `riscv-InstMsg.v`. These parameters are declared as global `‘define`s with the prefix `RISCV_INST_MSG_` to avoid namespace conflicts.

3.2 Example: Adding the LH Instruction

This section will guide you through adding a new instruction to the reference processor. As an example, we will implement the `LH` instruction from the **rv32i** ISA.

Since the datapath already supports all **rv32im** instructions, we only need to modify the control unit to add the new instruction. Begin by examining how the instruction moves through the pipeline. For subword memory operations like `LB`, we can refer to the `LW` instruction as a model. The `LW` instruction reads operands from the register file (operand 0) and a sign-extended immediate (operand 1), using the ALU to calculate the memory address. After the memory request is made in the Execute (X) stage, the response is written to the register file in the Writeback (W) stage.

The `LH` instruction is similar to `LW`, except it reads a halfword and sign-extends the response. To implement `LH`, we only need to modify the memory request length and memory response multiplexer in the control output table. The length should be set to 2 (`ml_h`), and the mux select should be set to the sign-extended halfword (`dmm_h`). Here’s the new control entry for `LH`:

```
`RISCV_INST_MSG_LH : cs={y, n, br_none, pm_p, am_rdat, y, bm_imm_i, n,
alu_add, md_x, n, mdm_x, em_x, ld, ml_h, dmm_h, wm_mem, y, rd, n};
```

Ensure this entry is written on a single line in the code. That's all that's needed to add the LH instruction.

Next, update the assembly tests by navigating to the lab's build directory. In the Makefile, under "List of Assembly Tests", add the new test for LH:

```
tests = \
riscv-addi.vmh \
riscv-ori.vmh \
riscv-lui.vmh \
riscv-add.vmh \
riscv-lw.vmh \
riscv-sw.vmh \
riscv-bne.vmh \
riscv-jal.vmh \
riscv-jr.vmh \
riscv-lh.vmh \ <- This is our new test
```

Now, run make check-asm-riscvstall to test the LH instruction. For most instructions, adding a new entry in the control output table, as we did here, will be sufficient. Review the datapath and set the appropriate control signals for each new instruction.

3.3 Objective 2: Implementing Bypassing

Once you have completed adding the required instructions to the control unit in the riscvstall directory, copy your source files to the riscvbyp directory to begin adding bypassing. You can copy and rename the files using the following commands:

```
% cd $LAB2_ROOT/riscvbyp
% cp ../riscvstall/* .
% for file in *; do mv $file ${file/riscvstall/riscvbyp}; done
```

Remember to update the prefixes in any 'include statements within the source files to riscvbyp. Additionally, search and replace all instances of riscvstall with riscvbyp in riscvbyp.mk (the Linux sed command can be helpful for this task).

To add bypassing, introduce bypass multiplexers (muxes) to the datapath. Examine Figure 1 to identify where these muxes should be placed and which values need to be forwarded. Update the datapath diagram accordingly and include this in your report.

You will need to add new control signals in the control unit to manage the bypass mux selects. Bypassing allows the processor to avoid data hazards by forwarding values needed in the Decode stage before they are written back. **The processor must be fully bypassed, enabling forwarding from the X, M, and W stages.** Consider using helper signals to indicate if values can and should be bypassed from each stage. For example, the signal rs1_X_byp_Dhl could be used to signify that register rs1 should be forwarded from the **X** stage. These helper signals will determine the bypass mux selections.

Implementing bypassing does not eliminate the need for stalling. You still need to stall for load-use hazards, as data memory responses must be received before they can be bypassed. Modify the stall signal to address only load-use hazards rather than any data hazard.

Create a new pipelined signal, is_load, with appropriate stage suffixes to indicate whether an instruction is a load. This will be used for the load-use stall signal.

3.4 Additional Architectural Details

The following information is supplementary but may be of interest.

Bubble bits mark when instructions in a pipeline stage are invalid. The bubble bit for the next stage is set when the current stage is stalled or squashed, and the next stage is not stalling. This mechanism avoids sending nops down the pipeline, reducing energy consumption since pipeline registers retain their values.

You may also notice extra bypass registers connected to the response sides of the instruction and data memories. These registers act as 1-deep queues, implementing a technique known as **skid-buffering**. They decouple the processor from memory, allowing the processor to store a memory response if it arrives while the processor is stalled. This is necessary because, in our synchronous memory model with the `val/rdy` interface, memory responses are valid for only one cycle. Skid buffers store these responses until the processor is ready to use them, preventing data loss.

3.5 Objective 3: Integrating a Pipelined MulDiv Unit

The reference processor uses an iterative `muldiv` unit from Lab 1 and stalls in the **D** and **X** stages while waiting for results. This method is not efficient. A better approach is to integrate a pipelined `muldiv` unit that operates independently, allowing the processor to continue running in parallel, only stalling when there is a data dependency.

In the `riscvlong` directory, you are provided with a functional model of a 4-stage pipelined `muldiv` unit called `riscvlong-CoreDpathPipeMulDiv.v`. This model simplifies the pipelined unit: actual multiplication and division are performed using functional operators (`*, /, %`) in the first stage, with three dummy stages used to pipeline the result. **Note: bypassing within the `muldiv` unit itself is not allowed.** Your goal is to integrate this pipelined unit into the processor and implement the necessary logic to run it in parallel, managing data hazards correctly.

After completing the `riscvbyp` processor, copy your source files to the `riscvlong` directory and rename them as needed, similar to the previous steps. Replace the `imuldiv_IntMulDivIterative muldiv` implementation with the provided pipelined `muldiv` unit. Then, implement and test any additional stall and bypass signals required.

A straightforward solution is to extend the pipeline by two stages. The first two stages of the pipelined `muldiv` unit can overlap with the existing **X** and **M** stages, while the last two stages are inserted before the **W** stage. **Ensure that the `muldiv` unit receives values immediately after the D (Decode) stage instead of the X (Execute) stage, and correctly connect the pipeline stall signals to the multiplier. Note that extending the pipeline increases latency for all instructions except `MUL/DIV`, which will impact overall benchmark performance.**

Once the stages are extended, add the required forwarding, stalling, and bypassing logic. While more efficient ways to implement a multicycle `muldiv` unit exist without extending the entire **X** stage, this simplified solution is sufficient for this lab.

4 Testing Methodology

For this lab, all **RISC-V** assembly tests are provided. However, you must create at least one custom assembly test that either targets specific bugs encountered during the lab or verifies the bypass paths in your completed processor. You may use the macros in `riscv-macros.h` or write raw assembly code. **In your report, explain the design of your custom test and how it confirms that bypassing is correctly implemented or that a bug has been fixed.**

All assembly tests are located in `$LAB2_ROOT/tests/riscv`. Any new tests you add should follow the existing naming conventions for proper compilation. Additionally, you need to include the new tests in the appropriate `.mk` file and update the `tests` variable in the Makefile located in `$LAB2_ROOT/build`.

Open one of the assembly test files, and you will see that most use macros defined in `riscv-macros.h`. This header provides macros that simplify writing assembly tests. For example, the test for `addi` can be found in `$LAB2_ROOT/tests/riscv/riscv-addi.s`.

At the beginning of each test file, include the `riscv-macros.h` header and the ‘`TEST_RISCV_BEGIN` macro to set up the `.text` section. Similarly, each test must end with the ‘`TEST_RISCV_END` macro, which contains the pass and fail routines.

The test body typically uses macros with the ‘`TEST_IMM` prefix for immediate instructions such as `addi`, `ori`, and `lui`. One common macro is:

```
`TEST_IMM_OP(instruction, source value, immediate value, expected result)
```

For example, this macro can test if `0 + 0 = 0`, `1 + 1 = 2`, and so on. The implementation of these macros in assembly can be found in `riscv-macros.h`. Typically, the `csrw` instruction is used to write to a special CSR register, allowing the simulator to track the test status: writing a 1 for a pass or the line number for a failure.

Other commonly used macros include the `SRC0_EQ_X` macros for testing matching source/destination registers, and the `BYP` macros to verify bypassing logic. While the bypass macros are not relevant to this lab, they will be crucial in the next one.

For more information on the available macros and their syntax, refer to `riscv-macros.h`.

4.1 Disassembling Instructions for Debugging

The simulators for this lab include disassembly features that are useful for debugging. There are three disassembly levels, which can be specified using the `+disasm=#` option. For example, to run the `riscvstall` simulator with a disassembly level of 2:

```
% cd $LAB2_ROOT/build
% ./riscvstall-sim +disasm=2 +exe=../tests/build/vmh/riscv-addi.vmh
```

This option can be combined with others, such as `+vcd` or `+stats`.

<u>**Level 1**</u>: Displays the program counter (PC) and the instruction for every executed instruction. Instructions are shown in program order but do not include those that were squashed or stalled.

<u>**Level 2**</u>: Extends Level 1 by including a register file dump for each instruction, showing the register values after execution. Like Level 1, it does not capture stalls or squashed instructions.

<u>**Level 3**</u>: Provides a pipeline trace, showing the flow of instructions through each pipeline stage. The stages are separated by the `'|'` symbol: **F** (Fetch) shows the PC of the fetched instruction, and the other stages show the instruction's name. Each row represents a cycle, with time moving downward. `'#'` symbols preceding an instruction represent a stall, and instructions surrounded by '-' represent a squashed instruction. The '(-_-)' symbols represent bubbles in the pipeline (i.e. an invalid instruction).

Here is an example trace from the `riscv-jr` test:

```
{-00080014-|#jalr   | addi   | (-_-) | (-_-) }
{-00080014-|#jalr   | (-_-) | addi   | (-_-) }
{-00080014-|#jalr   | (-_-) | (-_-) | addi   }
{-00080014-| jalr   | (-_-) | (-_-) | (-_-) }
{ 00080020 | (-_-) | jalr   | (-_-) | (-_-) }
```

In this trace, the processor stalls at PC=00080014 due to a data hazard between jalr and addi in the **D** and **X** stages. It then waits until addi completes and writes back its result. Once the register file is updated, the processor exits the stall, squashes the fetch stage (anticipating a jump), and jumps to PC=00080020, allowing jalr to continue down the pipeline. Note that the jr instruction is an alias for jalr with x0 as the link register.

4.2 Compiling New Assembly Tests

While a set of assembly tests and benchmarks is provided for your RISC-V implementation, you will also need to compile new assembly tests. **First, navigate to $LAB2_ROOT/tests/riscv and open riscv.mk to review the currently compiled tests.**

To compile new tests, go to the tests directory and create a separate build directory. Then, run the configure script to generate a Makefile tailored to your platform. Here's how to do it:

For assembly tests:

```
% cd $LAB2_ROOT/tests
% mkdir build
% cd build
% ../configure --host=riscv32-unknown-elf
```

For benchmarks:

```
% cd $LAB2_ROOT/ubmark
% mkdir build
% cd build
% ../configure --host=riscv32-unknown-elf
```

The -host=riscv32-unknown-elf option specifies using the course's cross-compiler instead of the standard gcc.

Next, compile the assembly tests in the riscv directory and convert them to .vmh files:

```
% make
% ../convert
```

Running make compiles the assembly test sources into binaries. Since our Verilog processor cannot directly interpret this format, the convert script generates object dumps from the binaries and converts them into .vmh files. These files are organized into bin, dump, and vmh directories. As long as the necessary .vmh files are present in the vmh directory, the simulator can access and run them.

Ignore any warnings from the linker about the missing "_start" label; this is due to the custom program entry point required by our Verilog simulator.

If you add or modify assembly tests, you only need to re-run make and ../convert in the build directory. The full setup process is only necessary if the build directory is deleted.

This compilation process is similar to building projects from source on Linux: creating a build directory, running configure, and then make. Installation is usually done with make install, but it is not required for this lab.

5 Evaluation

The evaluation for this lab involves running a set of C benchmarks located in the `$LAB2_ROOT/ubmark` directory. The benchmark suite includes the following four programs:

- `ubmark-vvadd.c`: Vector-vector addition
- `ubmark-cmplx-mult.c`: Complex multiplication
- `ubmark-masked-filt.c`: Masked filtering
- `ubmark-bin-search.c`: Binary search

All benchmarks can be executed in the `build` directory, with statistics automatically saved in their respective `.out` files. The `riscvstall` simulator results use the `-stall.out` suffix, while `riscvbyp` results use the `-byp.out` suffix. For example, the output of `ubmark-vvadd.c` for `riscvstall` will be stored in `ubmark-vvadd-stall.out`. Use the command below to run the benchmarks on `riscvstall` (replace `riscvstall` with `riscvbyp` to run on `riscvbyp`):

```
% cd $LAB2_ROOT/build
% make run-bmark-riscvstall
```

For this lab, you will compare the performance of the `riscvstall`, `riscvbyp`, and `riscvlong` processors. Report the cycle count and IPC for each benchmark on all three processors. In your analysis, discuss the performance differences, highlighting when bypassing improves performance and when its impact is limited. Additionally, examine the trade-offs between bypassing and stalling, and explain which aspects of the Iron Law of Processor Performance are affected.

6 Extensions

If you look up the **RISC-V ISA** specifications online, you will notice that we have only implemented a subset of the **rv32m ISA**. Three instructions are not yet implemented: **MULH**, **MULHU**, and **MULHSU**. Your task is to implement these instructions in your `riscvlong` core and create at least one assembly test for each.

7 Submission

7.1 Lab Report

Along with your source code, you must submit a lab report that includes the following sections:

- **Introduction/Abstract (1 paragraph max)**: A brief summary of the lab.
- **Design**: Describe your implementation, justify design decisions, note any deviations from the provided datapath, and provide a balanced discussion of both what and why you implemented certain choices.
- **Testing Methodology**: Outline your testing strategy, including how you tested individual modules and any corner cases you considered.
- **Evaluation**: Present your simulation results and cycle counts.
- **Discussion**: Analyze benchmark results, comparing the trade-offs of bypassing versus stalling.
- **Figures**: Include updated RISC-V datapath diagrams for both bypassing and the pipelined `muldiv` unit.

Please avoid using scanned or hand-drawn figures; ensure all visuals are clear and legible. The report must be clearly numbered and should not exceed **4 pages** (excluding figures). Exceeding this limit will result in penalties.

7.2 Deliverables

Submit a `.tar.gz` file of your working directory, preserving the original structure. Ensure all source files are in `$LAB2_ROOT/riscvstall`, `$LAB2_ROOT/riscvbyp`, and `$LAB2_ROOT/riscvlong`. Before packaging, remove any generated files by running:

```
% cd $LAB2_ROOT/build
% make clean
% cd $LAB2_ROOT/tests
% rm -rf build
% cd $LAB2_ROOT/ubmark
% rm -rf build
```

To create the tarball, use the following commands:

```
% cd $LAB2_ROOT
% cd ..
% tar -cvzf {student_id}-lab2.tar.gz lab2
```

The following files must be included in your submission:

- `riscvstall` source code
- `riscvbyp` source code
- `riscvlong` source code
- Custom assembly test
- Datapath diagram for the pipelined 5-stage RISC-V processor with bypassing
- Datapath diagram for the pipelined 7-stage RISC-V processor with bypassing and the pipelined `muldiv` unit
- Lab report

7.3 Submission Instructions

- Ensure your code is within the `lab2` folder. If the tarball is not created from this folder, grading will not be possible.
- Submit the tarball and the lab report separately via e3.

8 Grading Rubric

- **Report (30%)**
- **Code (70%)**
  - Objective 1 RISCV-stall (20%)
  - Objective 2 RISCV-bypass (25%)
  - Objective 3 RISCV-long (25%)
- **Extension (Bonus, 10%)**
  - 3 additional instructions

9 Tips

- Use incremental development-never code everything at once and hope it works!
- Utilize gtkwave to view waveforms for effective debugging!
- Use the unit testing framework!
- Always draw the hardware before you start coding!
- Clearly define the interaction between the control logic and the datapath!
- If you can't get everything working, be sure to thoroughly explain how far you did get in the lab report in addition to the debugging strategy you used!

10 Acknowledgments

This lab is adapted from ECE 4750 at Cornell University and ECE 475 at Princeton University.

11 Appendix

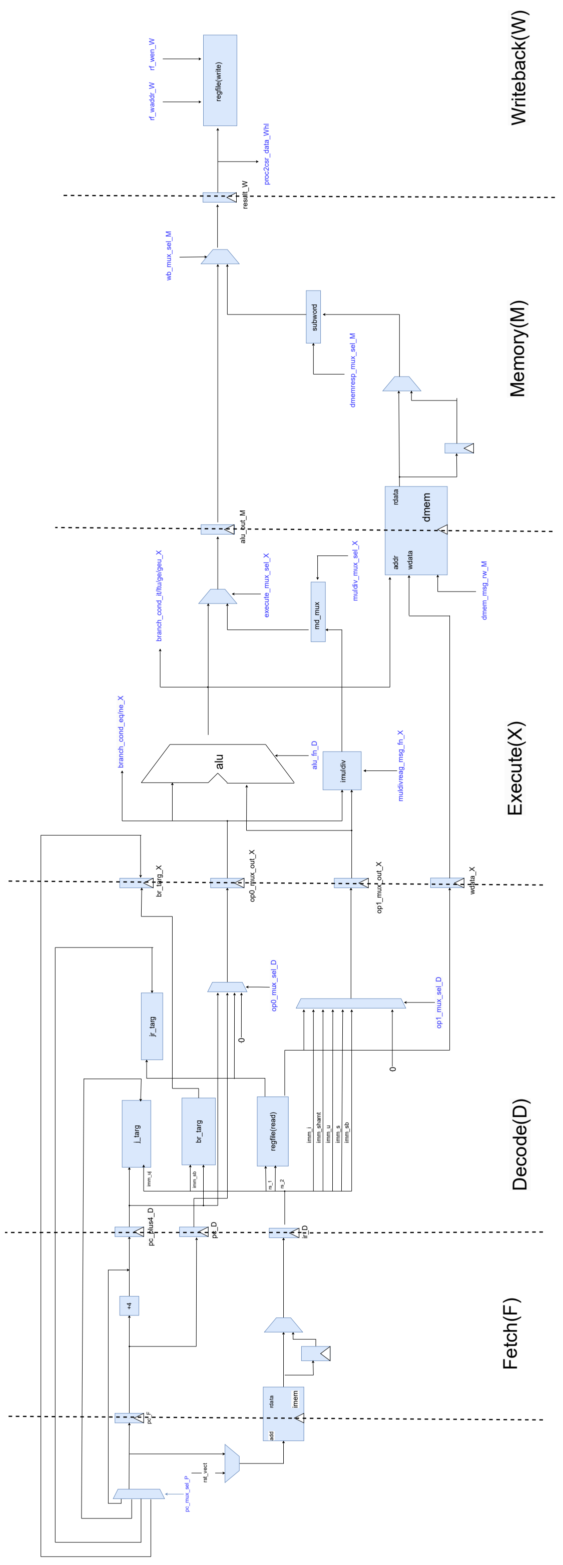

Figure 1: Datapath