Computer Architecture
Lab 2: Pipelined RISC-V Processor

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

revision: 9-29-2025

In this lab, we will work with a 5-stage pipelined RISC-V processor and expand its functionality. The goal is to fully implement the `rv32i` ISA, add a diagnostic instruction (`CSRW`), and support a subset of `rv32m`, enabling C programs to run without system calls. Detailed instruction specifications are provided in the included `riscv-isa.txt`.

You will begin with a reference processor that has a complete datapath but limited control logic. Your task is to extend the control unit to support the full instruction set, incorporate pipelined controls, and add **bypassing** to reduce stalls and improve performance.

Pipelining increases throughput by overlapping instruction execution, but also introduces hazards. The reference design uses stalling and squashing; in this lab, you will implement bypassing as an optimization and evaluate its trade-offs.

This lab provides experience in:

- Understanding and extending instruction set architectures (ISAs)
- Exploring pipelined processor microarchitecture
- Applying techniques to handle data and control hazards

1 Reminder

- **Submission Deadline:** 10/13 (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
```

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

- `build`: Makefile and compiled code
- `imuldiv`: Integer multiply/divide unit

- `riscvstall`: Pipelined RISC-V processor with stalling
- `riscvbyp`: Pipelined RISC-V processor with bypassing
- `riscvlong`: Pipelined RISC-V processor with bypassing and a pipelined mul/div unit
- `tests`: Assembly test build system
- `ubmark`: Benchmarks for evaluation
- `vc`: Additional Verilog components

The `tests` directory contains a build system that compiles assembly tests into Verilog memory hex (.vmh) files for initializing processor memory. Inside its `riscv` subdirectory are tests that check each ISA instruction, plus a few for pseudo-instructions. These are not implemented in hardware but expanded by the compiler or assembler into actual instructions.

Always test each newly implemented instruction with its corresponding assembly test before moving on. See Section 4 for testing procedures. **Compile the tests as described in Section 4.2 before running them.**

The processor source code is provided in three directories: `riscvstall` (stalling only), `riscvbyp` (with bypassing), and `riscvlong` (with a pipelined mul/div unit).

2.2 Building the Project

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

First, compile the tests as described in Section 4.2:

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

Next, 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` builds the processor simulators. Targets without `-rand-` use synchronous memory with a fixed 1-cycle response. Targets with `-rand-` use memory with random delays. To see how tests are defined, open the Makefile and check the "List of Assembly Tests" section. Add new tests by appending to the `tests` variable.

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 may also run individual tests directly. For instance, to run `riscv-addi`:

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

Use `+stats=1` to display statistics (off by default), and `+vcd=1` to generate a waveform file (`.vcd`) viewable in `gtkwave`. Note: each new run overwrites the file—rename it if you want to keep it.

The Makefile invokes `riscvstall-sim` and `riscvstall-randdelay-sim` to run the assembly tests for `make check-asm-riscvstall` and `make check-asm-rand-riscvstall`, respectively. The `riscvstall` simulators use the stall-based processor, while the `riscvbyp` simulators target the bypass-based processor. At first, both share the same stall-based implementation, but once you add bypassing, the two versions will diverge.

Running commands such as `make check-asm-`, `make check-asm-rand-`, or `make run-bmark-*` produces a `.vcd` waveform dump for each test, named after the test itself. For example, the `riscv-addi.vmh` test generates `riscv-addi.vcd`, while random-delay runs add a `-rand` suffix (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`.** Keep in mind that running the automatic tests for `riscvstall`, `riscvbyp`, and `riscvlong` in sequence will overwrite existing `.vcd` files.

3 Pipelined 5-Stage RISC-V Processor with Bypassing

In this lab you will use a fully implemented RISC-V datapath with stalling and a partially completed control unit. The datapath includes the iterative multiply/divide unit from Lab 1.

The lab has four main objectives:

- Extend the control unit to support more instructions.
- Implement additional M-extension instructions.
- Add bypassing to the datapath and the control unit.
- Integrate a pipelined multiply/divide unit.

Begin with **Objective 1** in the `riscvstall` directory. For **Objective 2**, complete the missing M-extension instructions in the core processor. Once this is done, copy your source files to the `riscvbyp` directory to begin **Objective 3**. Rename the files with the `riscvbyp-` prefix and update any `` `include `` paths accordingly. Finally, for **Objective 4**, move to the `riscvlong` directory, adjust filenames and paths as needed, and integrate the pipelined multiply/divide unit.

3.1 Objective 1: Enhancing the Control Unit

In this objective, you will only modify the control unit (`riscvstall-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: 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`
- Multiply/Divide: `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. **Review how the controls interact with 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 `` `defines `` with the prefix `RISCV_INST_MSG_` to avoid namespace conflicts.

3.2 Example: Adding the lh Instruction

This section walks through adding a new instruction to the reference processor, using `lh` from the **rv32i** ISA as an example.

Since the datapath already supports all **rv32im** instructions, only the control unit needs changes. First, trace how the instruction moves through the pipeline. For subword memory ops like `lb`, use `lw` as a model: `lw` reads operand 0 from the register file and an immediate (operand 1), uses the ALU to form the address, issues the memory request in Execute (X), and writes the response back in Writeback (W).

`lh` matches `lw` except it reads a halfword and sign-extends the result. Implement `lh` by updating the memory request length and the memory-response mux select in the control output table: set length to 2 (`ml_h`) and the mux to the sign-extended halfword (`dmm_h`). The new control entry is:

```
`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 appears on a single line in the source.

Next, add the `lh` test in the lab Makefile under "List of Assembly Tests":

```
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 validate `lh`. For most instructions, adding a row to the control output table as above is sufficient; verify the datapath and set the appropriate control signals accordingly.

3.3 Objective 2: Implementing Remaining RISC-V M Instructions (`mulh`, `mulhu`, `mulhsu`)

So far, only part of the **rv32im** ISA has been implemented in your core. Three multiply instructions are still missing: `mulh`, `mulhu`, and `mulhsu`. Your task is to add support for these instructions and write at least one assembly test for each.

These instructions are variations of `mul` that return the high 32 bits of a 64-bit product:

- `mulh`: signed $\times$ signed
- `mulhu`: unsigned $\times$ unsigned
- `mulhsu`: signed $\times$ unsigned

To implement them, study how multiplication is handled in the existing code. In particular, you will need to extend the iterative multiplier modules: `imuldiv-IntMulIterative.v` and `imuldiv-IntMulDivIterative.v`. Update the datapath and control logic so that the correct high 32-bit result is produced for each case.

3.4 Objective 3: Implementing Bypassing

Once you have added the required control entries in `riscvstall`, copy your sources to `riscvbyp` to begin the bypassing work. In `riscvbyp` run the provided setup script:

```
% cd $LAB2_ROOT/riscvbyp
% ./setup.sh
```

The `setup.sh` script copies files from `../riscvstall`, renames any filenames (and relevant build entries) replacing the `riscvstall` prefix with `riscvbyp`, and performs basic build updates. Verify filenames before proceeding.

To implement bypassing, add bypass multiplexers (muxes) in `riscvbyp-CoreDpath.v`. Refer to Figure 1 to locate the muxes and determine which values should be forwarded. Update the datapath diagram and include the modified figure in your report.

Add new control signals in `riscvbyp-CoreCtrl.v` to drive 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.

Finally, Don't forget to integrate the modified control and datapath in `riscvbyp-Core.v`.

Note that bypassing does not remove all stalls. You must still stall for load-use hazards because a load's memory response must be available before it can be forwarded. Adjust the stall logic so it triggers only on load-use hazards (not on every data hazard).

To help, introduce a pipelined load indicator, `is_load`, with appropriate stage suffixes to indicate whether an instruction is a load. This will be used for the load-use stall signal.

3.5 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.

3.6 Objective 4: Integrating a Pipelined MulDiv Unit

The reference core currently uses an iterative `muldiv` (from Lab 1) and stalls in the **D** and **X** stages until the result is ready. This is simple but inefficient. In this objective you will integrate a pipelined `muldiv` so the processor can continue executing other instructions in parallel and only stall when a true data dependency requires it.

A functional 4-stage pipelined model is provided in `riscvlong`: `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 **add support for additional M instructions**, 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:

```
% cd $LAB2_ROOT/riscvlong
% ./setup.sh
```

Next, replace the iterative `imuldiv_IntMulDivIterative` implementation with the provided pipelined unit by including `riscvlong-CoreDpathPipeMulDiv.v` in `riscvlong-CoreDpath.v`, and update any affected `` `include `` paths. Finally, implement and test the additional stall and bypass signals required to integrate the pipelined unit in `riscvlong-CoreCtrl.v` and `riscvlong-CoreDpath.v`, and wire the updated control and datapath together in `riscvlong-Core.v`.

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. Extending the pipeline will introduce additional latency for all instructions, not just `mul` and `div`, and you should expect this to affect benchmark performance.**

Once the pipeline is extended, add the necessary forwarding, stalling, and bypassing logic to handle data hazards. While more efficient multicycle designs are possible, this simplified pipelined approach is sufficient for the purposes of this lab.

4 Testing Methodology

For this lab, most **RISC-V** assembly tests are provided. However, you must create custom tests for the `mulh`, `mulhu`, and `mulhsu` instructions. In addition, write at least one custom test that either targets a specific bug you 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 demonstrates that bypassing is correctly implemented or that a bug has been resolved.

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 (+disasm=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`.

Level 1: 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.

Level 2: 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.

Level 3: 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 tests into binaries. Since the Verilog processor cannot execute these directly, the `convert` script generates object dumps and produces `.vmh` files, which are placed in the `bin`, `dump`, and `vmh` directories. As long as the required `.vmh` files are in `vmh`, the simulator can load and run them. Remember to list the new `.vmh` filenames in the `Makefile`. Ignore any linker warnings about the missing "_start" label—these are expected due to the custom entry point used by the simulator.

If you add or modify assembly tests, simply re-run `make` and `../convert` in the build directory. The full setup is only necessary if the build directory is deleted. This build process resembles compiling software on Linux: create a build directory, run `configure`, then `make`. Unlike typical projects, however, `make install` 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, and analyze the performance differences. Highlight cases where bypassing improves performance and cases where its effect is limited. Finally, discuss the trade-offs between stalling and bypassing, and explain how each affects the Iron Law of Processor Performance.

6 Submission

6.1 Lab Report

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

- **Introduction/Abstract (max 1 paragraph)**: A concise summary of the lab.
- **Design**: Describe your implementation, justify design decisions, note any deviations from the provided datapath, and explain both what you implemented and why.
- **Testing Methodology**: Explain your testing strategy, including how you verified individual modules and which corner cases you considered.

- **Evaluation**: Present your simulation results and cycle counts.
- **Discussion**: Analyze the benchmark results, comparing the trade-offs of bypassing and 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.

6.2 Modified Files

For this assignment, the following files should be modified:

- `riscvstall-CoreCtrl.v`, `riscvstall-InstMsg.v`
- `imuldiv-IntMulIterative.v`, `imuldiv-IntMulDivIterative.v`, `imuldiv-MulDivReqMsg.v`
- `riscvbyp-CoreCtrl.v`, `riscvbyp-CoreDpath.v`, `riscvbyp-Core.v`
- `riscvlong-CoreCtrl.v`, `riscvlong-CoreDpath.v`, `riscvlong-Core.v`, `riscvlong-CoreDpathPipeMulDiv.v`

6.3 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
- Additional assembly tests of M-extension instructions
- 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

6.4 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.

7 Grading Rubric

- **Report (30%)**
- **Code (70%)**
  - Objective 1 RISCV-stall (25%)
  - Objective 2 RISC-V M Instructions (10%)
  - Objective 3 RISCV-bypass (15%)
  - Objective 4 RISCV-long (20%)

8 Tips

- Use incremental development-never code everything at once and hope it works.
- Use `gtkwave` to debug with waveforms.
- Take advantage of the unit testing framework.
- Sketch the hardware before coding.
- Define control-datapath interactions clearly.
- If your design is incomplete, explain your progress and debugging strategy in the report.

9 Acknowledgments

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

10 Appendix

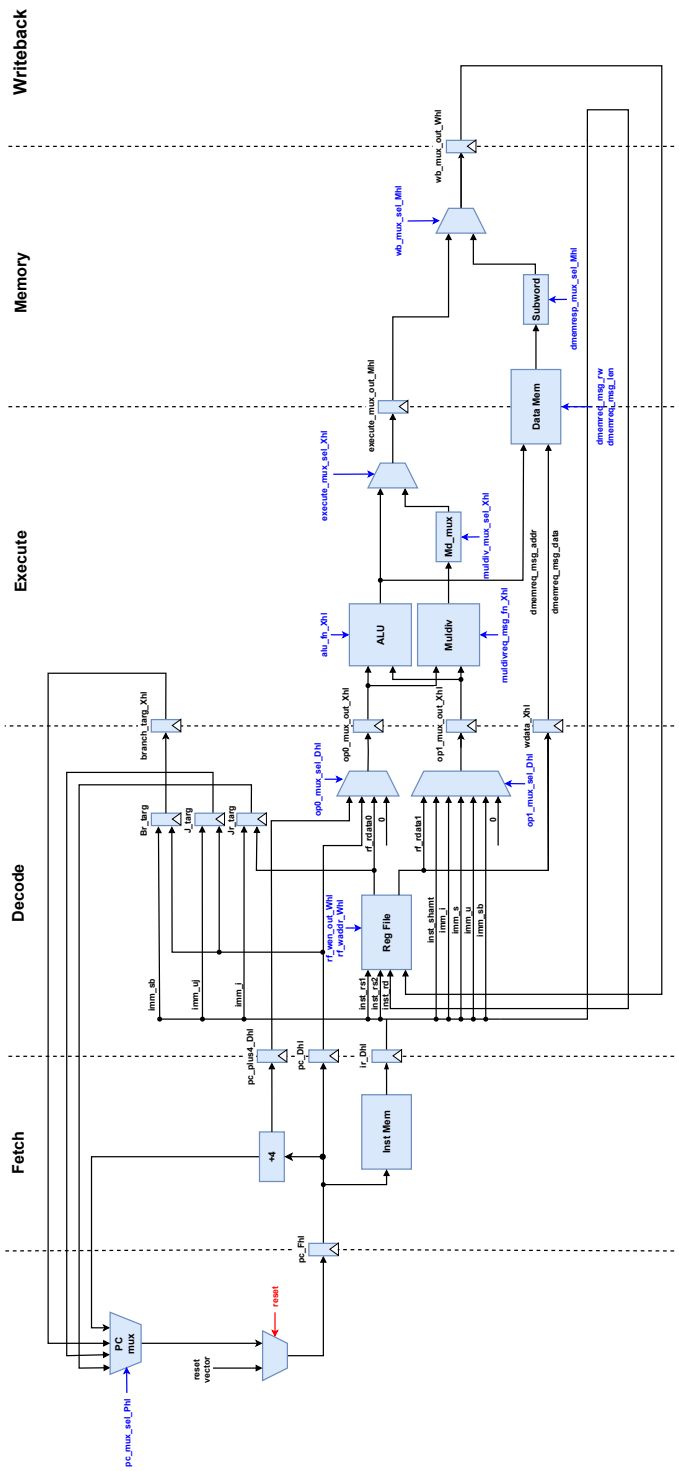

Figure 1: Datapath