Lab 3: Pipelined Processor

CPE 333 - S23

In this lab, you will design a pipelined prcoessor. You are required to implement the RV32I instruction set using pipelining techniques discussed in lecture. This handout provides an incomplete specification to get you started with your design. A portion of the lab is left open ended for you to explore design options that interest you. You will begin by creating a basic pipeline that can execute the RV32 instruction set (without hazards). Then you will add support for hazard detection and data forwarding. Bonus points will be given to the group(s) that can execute the Labo benchmarks in the least amount of time and consume the least amount of energy.

Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. Pipelining takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implantation technique used to make fast processors and highly efficient processors that cost less than a dollar.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer assembly line, different steps are completing different parts of the different instructions in parallel. Eahc of these steps is called a stage. The stages are connected one to the next to form a pipe – instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

The throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are connected together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving between moving an instruction one step down the pipeline is a processor cycle. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage. Ideally the pipeline would have balanced pipeline stages, but usually the stages are never perfectly balanced.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has a substantial advantage that, unlike some speedup techniques, it is not visible to the programmer.

Overview Design Diagram

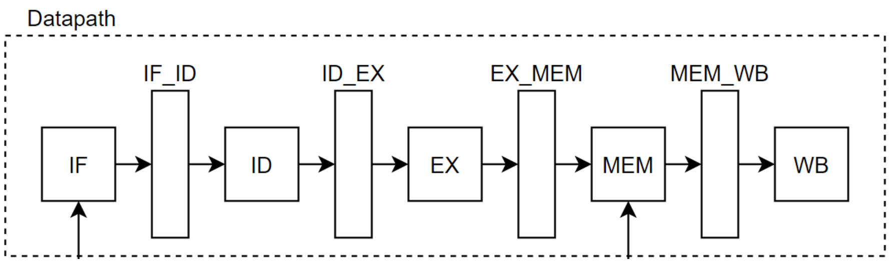


Figure 1: Example 5-stage pipeline

Pipeline Control In your pipeline design, you can reuse the decoder from 233 [1] to generate the control signals for your instructions in the decode stage. You do not need a centralized control unit (FSM) as we used in 233.

[1] We used a hardwired control approach in 233; using a microprogrammed control unit is another approach.

Pipeline Datapath You may reuse modules from 233, however you should create a new top level module for describing your new processor datapath. You may choose the pipeline depth (d) for your processor [2]

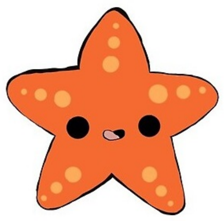
[2] Bonus points for supporting your design choice with a timing analysis of stages in vivado.

Between each pipeline stage, you will need a set of registers to hold the values from the previous stage. You do not need to implement those registers as one large register. You are permitted to break the pipeline registers into many smaller registers each containing one value (e.g., the ALU output, or a control word). Some example implementations include:

- Modular stages with registered outputs. Break the pipeline into individual modules, each with an always_ff block to create flip-flops on the output signals. This option is the most "plug-and-play", allowing a stage's definition to be entirely self-contained.
- Modular stages and modular register "blocks". Each pipeline register is a module consisting of individual flip-flops for the relevant signals.
- Monolithic registers with packed structs. Define a struct for each stage's output and instantiate registers for these structs between the stages. This has the advantages of automatically scoping variable names (ex.opcode vs mem.opcode), allowing easy modification of the interface, and is more succinct.

There are no requirements on how you choose to implement your stages. Pick a style that works best for your group.

Lab Milestones



Basic Pipeline Design

For the first part of lab, you can assume that the programs executed on your pipeline do not contain data hazards (e.g. data dependencies –specifically, RAW dependencies - between instructions in the pipeline at any time) or control hazards (e.g. control flow instructions – branches or jumps). Note that these assumptions mean that your pipeline for this lab can fetch a new instruction every clock cycle. You can find sample test programs on Canvas.

Pipeline + hazards and static branch prediction

By the end of this milestone, you should be able to execute any arbitrary program, including test-all and your lab0 benchmarks. To do this, you will need to implement hazard handling logic (e.g. forwarding, stalls, etc.). As part of this milestone, you should measure the speedup of your pipeline as compared to your results from lab0 with the OTTER.

Pipeline + hazard unit + L1 Caches

For this milestone, you will need to integrate an arbiter, such that both split caches (I-Cache and D-Cache) connect to the arbiter, which interfaces with memory. Since main memory only has a single port, your arbiter determines the priority on which cache request will be served first in the case when both caches miss and need to access memory on the same cycle.

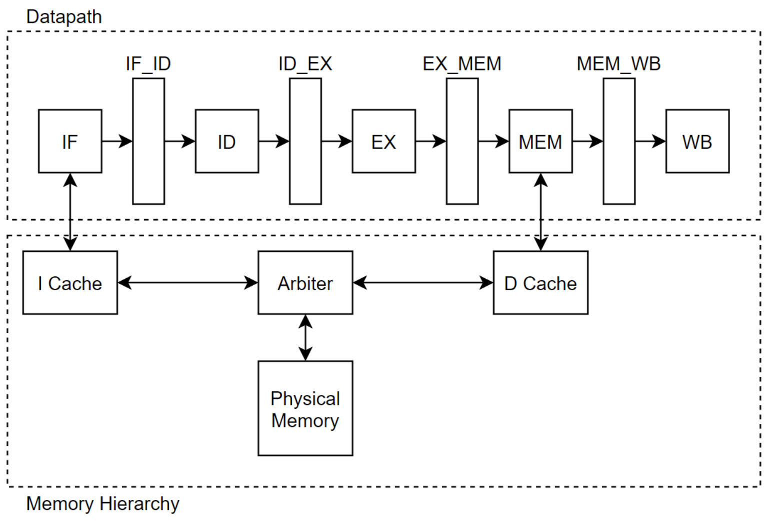


Figure 2: Pipelined Otter + parameterized (variable delay) memory + Cache

Advanced Design Options This is not an exhaustive list. Feel free to propose any feature you think may improve performance. The features in the provided list are designed to improve performance on most test codes based on real-world designs.

In order to decide on exact feature specifications and tune design parameters (e.g., branch history table size, and the size of victim cache), you need information about the performance of your processor on different codes. This information is provided through performance counters. You should at least have counters for hits and misses in each of your caches, for mispredictions and total branches in the branch predictor, and for stalls in the pipeline (one for each class of pipeline stages that get stalled together). You should add counters for any part of your design that you want to measure and use this information to make the design better. The counters may exist as physical registers in your design or as signal monitors in your testbench.

Cache organization and design options:

- L2+ cache system
- 4-way set associative cache
- Parameterized cache
- Alternative replacement policies [3]

[3] `http://old.gem5.org/Replacement_policy.html`

Advanced cache options:

- Eviction write buffer
- Victim cache
- Pipelined L1 caches
- Non-blocking L1 cache
- Banked L1 or L2 cache

Branch prediction options:

- Local branch history table
- Global 2-level branch history table
- Tournament branch predictor
- LTAGE branch predictor
- Alternative branch predictor
- Software branch predictor model
- Branch target buffer, support for jumps
- 4-way set associative or higher BTB
- Return address stack

Prefetch design options

- Basic hardware prefetching
- Advanced hardware prefetching

Other design options

- rv32 M Extension - Multiplication
- rv32 C Extension - Compressed
- rv32 V Extension - Vector
- rv32 A Extension - Atomic
- rv32 F/D Extension - Floating Point
- rv32 J Extension - Dynamically Translated languages
- Privleged/System Instruction Support

Superscalar design options

- Multiple issue
- Register renaming
- Out-of-order (scoreboarding, Tomaulo)