UiT

THE ARCTIC
UNIVERSITY
OF NORWAY

Mandatory Assignment 2

INF-2200 (fall 2016)
Department of Computer Science
University of Tromsø

In a sentence

Implement a pipelined binary code simulator
for a subset of the MIPS architecture.

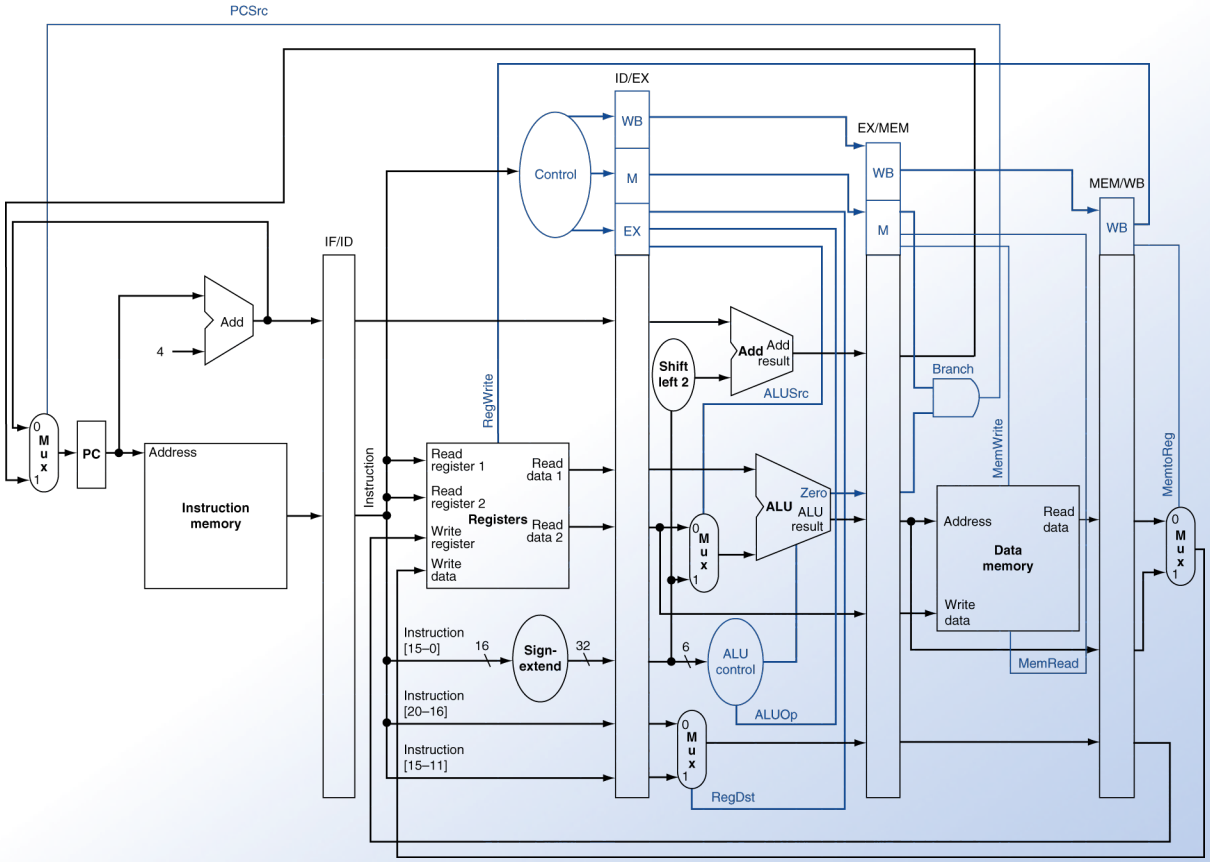
Details

- Pipelined datapath from textbook
- Subset of MIPS instruction set (17 instructions)
- Should handle data hazards and control hazards
- Written in Python

First

- Read and understand chapter 4.1–4.8 (up to, but not including dynamic branch prediction) Chapters 1-3 are also useful.
- Make sure you understand the concept of pipelining before you begin!
- Understand the MIPS instruction format.
- Understand binary operations and hex.
- Understand hazards and how they can be avoided.

Understand figures like this!



Instruction subset that you will implement

Instr.	Description
`j`	Jump
`beq`	Branch equal
`bne`	Branch not equal
`lui`	Load upper immediate
`slt`	Set less than
`lw`	Load word
`sw`	Store word
`add, addu`	Add, Add unsigned
`addi, addiu`	Add immediate, Add immediate unsigned
`sub, subu`	Subtract, Subtract unsigned
`and`	Binary AND
`or`	Binary OR
`nor`	Binary NOR
`break`	Break execution

- **What is the difference between add and addu?**

Hazards

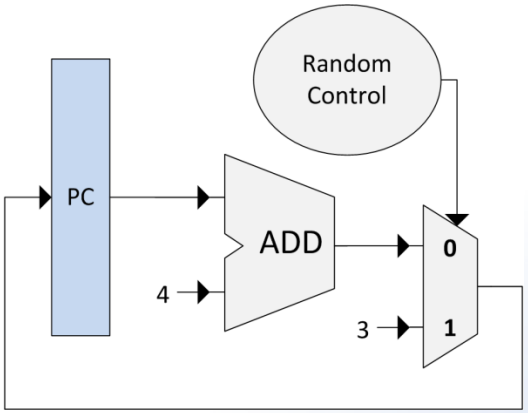
- Data hazards
  - Read-after-write
  - Load-use
  - Others?
- Control hazards
  - How to handle branching?
- Select an approach to handle them and implement it!

Code

- You will implement and connect CPU Elements, where each type of element is implemented as a subclass of a `CPUElement` base class (provided in pre-code).
- You must instantiate and connect the CPU elements in a `MIPSSimulator` class (provided by the pre-code).
- The simulator instantiated from the `MIPSSimulator` class shall run one clock cycle when the `tick()` method is invoked.
- Make sure that the `break` instruction ends the simulation.

Pre-code

- Pre-code implements the following simple control and data path:



- Already implements the following CPU elements:
  - Adder
  - Constant
  - Mux
  - PC (must probably be modified when handling hazards)
  - Random Control (useless)
  - Test Element (used for testing other elements, see how it's used in mux.py)

Code

- You MUST use the top-level API provided by the `MIPSSimulator` class. This means:
  - Don't remove or change the semantics of the functions `clockCycles`, `dataMemory`, `registerFile`, `printDataMemory`, `printRegisterFile`, or `tick`.
  - You can still change the implementation of these functions, as long as the functionality is the same (and you must, naturally, reimplement `tick`).
  - Use the dictionary `memory` in the `Memory` class!
  - Use the dictionary `register` in the `RegisterFile` class!

Testing

- Verify that the simulator runs correctly by running code that causes data and control hazards.
- You must write this code—first in MIPS assembly, then convert it to binary (by hand) to run it
- You need to write a test for each CPU element you implement, to ensure that it works when connecting the datapath!
- You have to work test-driven
  - When implementing a new CPU element, write a unit test first (it should fail, naturally), then implement the actual functionality of the CPU element.
  - When the unit test succeeds, the CPU element is (hopefully) implemented correctly.
  - Look at mux.py for an example.
  - All unit tests should be invoked by functest.py
  - The unit test for <elem>.py should be invoked by running
  - “python <elem>.py” (try running *python mux.py*).

Hints

- Read Chapter 4.6! Read Chapter 4.6! Read Chapter 4.6! Read Chapter 4.6! Read Chapter 4.6! Read chap…
- Keep the green page from the textbook close!
- Look at powerpoint slides for text book, illustrations in text book.
- Mary Jane Irwin's slides are very useful!!!
- Start working NOW!!!
  - This is one of the largest assignments during the CS bachelor!
  - It is not super-duper-difficult, but requires A LOT of time and work!
- Read and understand the pre-code as soon as possible (especially the base class for the CPU elements).
- If something is unclear, contact one of the TAs ASAP!

Report

- Summary of the data path and controllers you are simulating.
- Description of data hazard handling approach, including a discussion about the cost, complexity, and performance of the approach.
- Control hazard handling approach.
- Description of the simulator implementation, including the functions you have implemented.
- Detailed description of the test cases you have designed (first and foremost of all the machine code instructions that causes hazards, but you could also write about per-element unit tests).
- Summary of known bugs and problems.
- A figure of the data path is not formally required, but is a plus!

Deadline

- October 12th @ 12:00 PM (noon)
- Hard deadline!

- This means you have 4 weeks!
- But don’t slack off! You’ll need the time!

A few remarks

- The actual MIPS ISA uses delayed branching (see Chapter 4.3). You should not use delayed branching in your implementation!
- You should not implement dynamic branch prediction (Chapter 4.8) nor exception handling (Chapter 4.9).
- Don’t worry in case your final implementation turns out not to be 100% correct.
- But the report is important —demonstrate that you have truly understood the concepts!