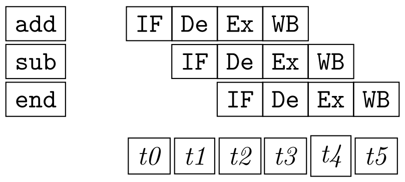
In this lab, we will first go over the "draining" portion of Homework 1 to provide a greater intuition for what this portion of the assignment refers to. Afterwards, we will turn our attention to gem5, and examine the various pipeline schemes that this processor uses. This will provide an initial intuition and motivation for the tool, before we warm up to Homework 2 in next week's lab.

Note, if you are working on the VM, start by downloading and building a clean version of gem5! This will take a while, so you should do this in the background while we talk about draining!

1 Pipeline Emulator Draining

Consider the following pipeline diagram.



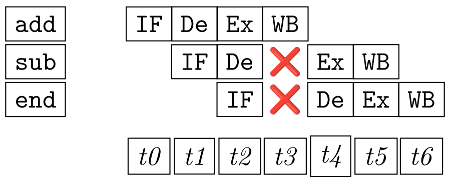
Think about what happens here. At *t0*, `add` enters the pipeline, and the register state is updated by the end of *t3*. Similarly, `sub` enters the pipeline at *t1* and updates the register state at *t4*.

This begs the question, if we receive an `end` instruction at clock cycle *t2*, at what point should we exit the simulation? From the Instruction Fetch (`IF`), it may be the case that our simulator could check the state, and we could exit upon finding this instruction at that point. But if we do this, we get an undesired behavior! The simulator would exit, but we still have instructions in the pipeline that haven't finished executing yet, so our register state will be incorrect on exit.

Given this, let's think about the intended behavior of what we would like our pipeline to do. At what cycle is all of the meaningful work complete? Given this, our processor emulator needs to have some notion of exiting at the right time so that the register file reflects all of the meaningful work completed when the simulator exits.

Things to consider. Note, there likely needs to be some mechanism in your processor emulator that allows the executing program to see an arbitrary ending of the program.

Thinking about our discussion of pipeline hazards from class today, suppose one of the source registers in `sub` is the destination register from `add`. That is, our program suffers from a data hazard. If our emulated processor addresses this hazard by "bubbling," then we would get the following pipeline diagram.



We don't just need to worry about single dependencies between instructions! It is possible that we have a long chain of dependent instructions that cause long stalls in the pipeline. Therefore, we need to have a way to appropriately call `end` and exit at the time that makes sense given the current state of the pipeline. This behavior of needing to exit when the pipeline has finished flushing its current contents is called **draining**, and this behavior should be part of your implementation!

There are several mechanisms to implement draining. We will be sure to spend the first part of lab going over various methodologies to model this behavior. If you are still stuck after lab, be sure to come by office hours!

2 Intro to `gem5`

Let's start using `gem5`! As you will have noticed, our course Docker container comes with a pre-built version of `gem5`. This is (hopefully) helpful, because gem5 is *veeeery* slow to compile. Fortunately, the compilation tool (`scons`) is clever and only recompiles files impacted by the most recent modifications. Even still, the binary is large enough that the linking step of compilation takes a while!

From here, see the course resource for using gem5 to get setup! In lab, we will go through the various pieces of the project more explicitly.

3 Pipelining in gem5

The processor source files can be found in the `src/cpu` directory. In this directory, we will find several subdirectories, including `simple`, `minor`, and `o3`. These directories build on the parent `BaseCPU` class defined in the central `src/cpu` directory.

Your job in this lab is to go through the various processor models in gem5 and try to deduce the pipeline for each processor! In doing so, your task is to:

1. determine the construction of the data path and pipeline for that processor model,
2. test the extent to which the order of execution steps is modifiable, and
3. run a microbenchmark (a small program) through each of the processor models and analyze the performance of each!

This exercise will both encourage you to think more deeply about pipeline construction, and it acts as an initial exposure to the gem5 tool and its outputs!