Introduction to Computer Architecture 2021
Lab 3 — Pipelining a MIPS Processor

Introduction

In this lab you will do the following things:

1. Identify different stages though the execution of a MIPS instruction.
2. Add registers between stages to hold temporal values of data and control signals.
3. Hookup the key parts to make a pipelined working datapath.
4. Verify that your design works by running several programs.
5. Add Operand Forwarding to your design in order to handle some data hazards.

Examination

When you have completed the lab you should enter a Zoom meeting with one of the TAs in the course on the designated time. Through screen sharing you should demonstrate that your solution works and explain it. You are expected to be able to answer the questions in this document but the TA may ask you other questions as well. More information on how to set up the Zoom meeting is available on the lab page on Studium. The labs will be examined in English.

Getting Started

Read the background material

Before you get started you should have completed the lecture on pipelining and hazards. (We recommend that you read Chapter 4.6 (Pipelined Datapath and Control) in the 4th Edition of the book as well.) This will explain step by step the foundations behind pipelining, showing diagrams and pictures of what you are to build.

Files and programs

You are provided with the following files in the *mips-pipeline-lab.zip* archive:

mips-pipeline-lab.circ A skeleton of the pipelined processor, missing most of the important parts responsible for making the pipelining feasible.

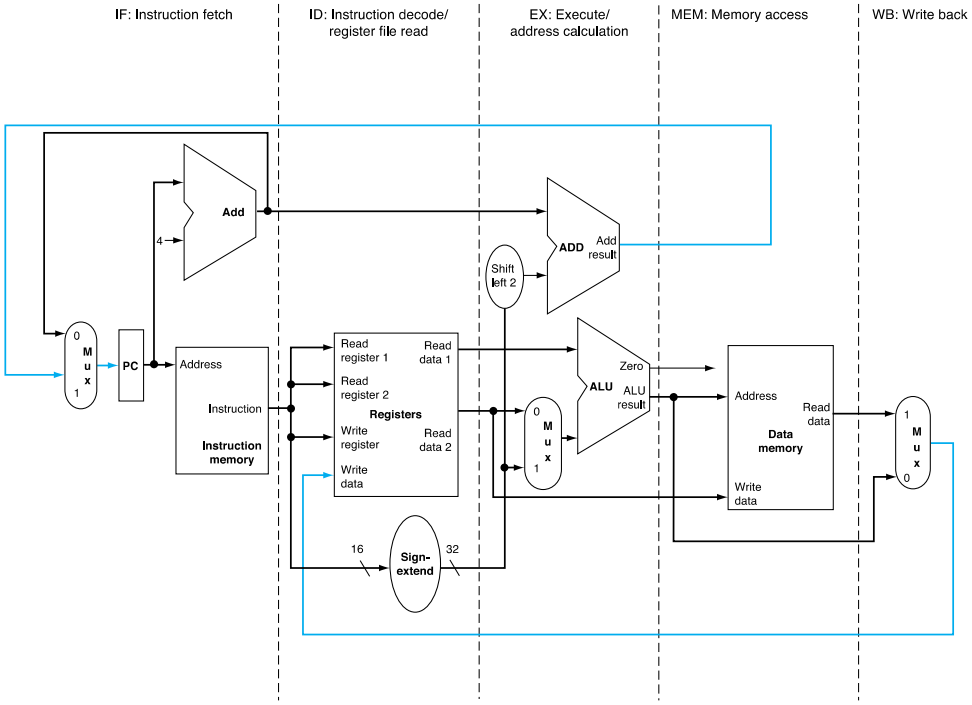
1234.mem A hex memory file that contains the values 1, 2, 3, 4 in the first 4 words of memory.

test-code-1.mem**,** ***test-code-2.mem*****, and** ***test-code-3.mem*** Files with different simple test programs for verifying your design.

For this lab, we assume you are now a skilled user of Logisim from the previous lab.

Step 1: Adding Pipeline Registers

As you have seen in class, there are 5 stages in the MIPS pipeline. The following figure depicts each of these stages.



We will refer to them as **IF** (*Instruction Fetch*), **ID** (*Instruction Decode*), **EX** (*Execute*), **MEM** (*Data memory access*), and **WB** (*Write Back*).

In order to make these pipelined stages to work, it is necessary to put *pipeline registers* between them. The pipeline registers hold the instructions and any temporary values for the instructions as they move through the pipeline on each cycle. (E.g., if you read register values or decode the instruction, you will need to store this information in the pipeline registers so each stage thereafter has access to them.) The values and control signals are moved from one stage to the next one with each clock cycle. Your first task is to add these pipeline registers. You will need to figure out where to put them (the above figure is a good hint) and what to store at each stage (the wires that cross the stages in the above figure are a good hint). Make sure you understand what data you need to store and where in the above figure before you proceed!

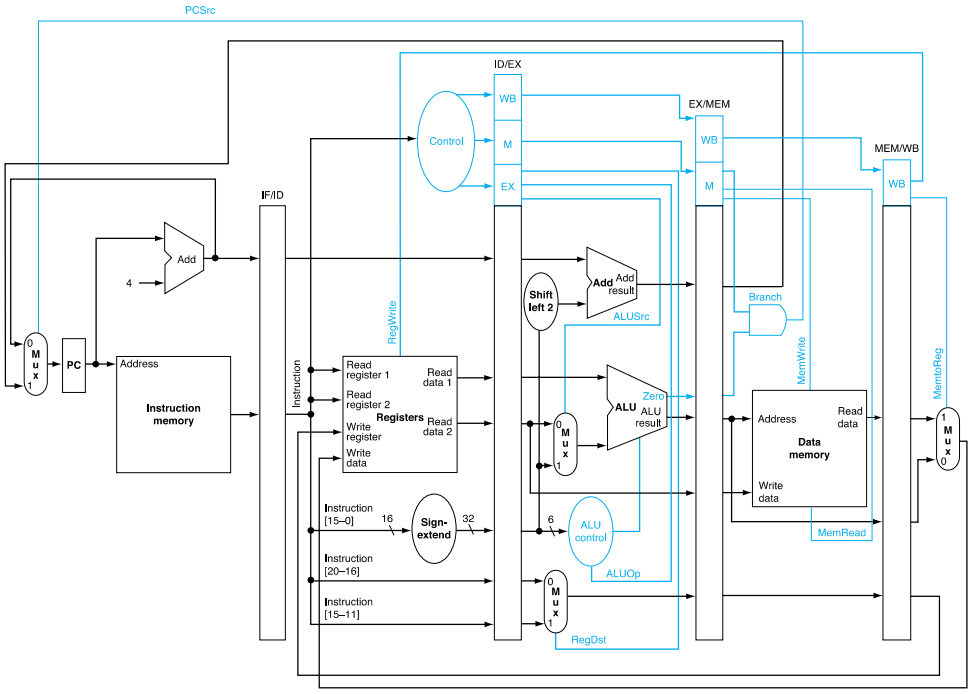
The next figure shows a high-level view of the processor with the registers added. The black rectangular boxes situated between stages include all the registers for the data around the datapath, while the blue ones includes the registers for the control signals, organized by execution stage.

IF/ID Registers

Let's start with an easy one. For this section we only need two registers: one for the PC+4 value and one for the instruction read from memory. We have already put one of these in for you as an example in the starter file. In this case, both registers are 32-bits wide, and there are no control signals to save. You will need to add the second register aligned next to the instruction memory output in your design.

ID/EX

Here is when things start getting a little messier. We have to pay special attention to data/control signals to understand what is happening, but don't worry, if you do it the right way, the rest will be very straightforward! (Think carefully about what data you need to generate in the ID stage and send along to the later stages before you start building.)



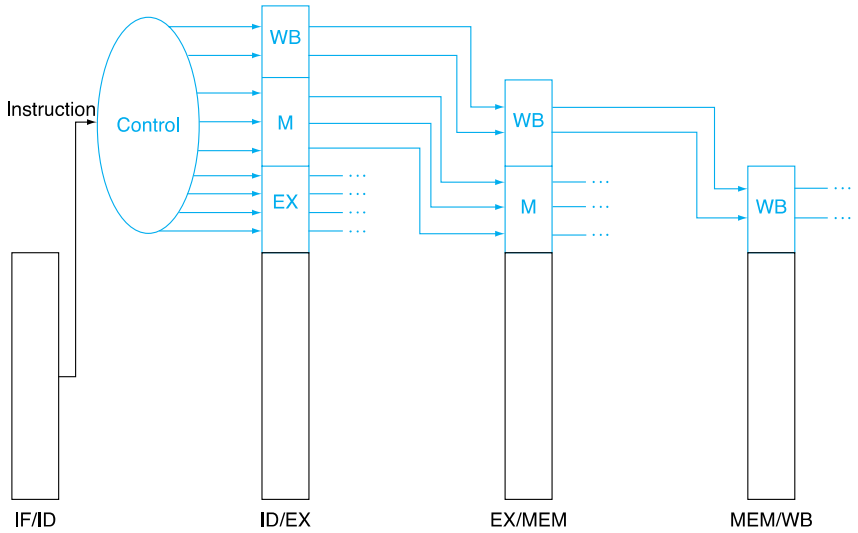
ID/EX: Pipeline (Data) Registers

Let's first look at the dataflow. We can see from the picture that we need to save the *Next PC* value (from the IF stage), *Read Data 1* and *Read Data 2* from the register file, as well as the output of the sign-extension module. The procedure here is very similar to the previous part: just add some registers aligned with the outputs. However, we also need to save some bits of the original instruction in different registers, such as bits [20-16] and [15-11]. Add some registers here as well, but be careful, since you'll have to adjust its width, since you no longer need 32-bits like for the other ones.

> **Question 1:** *Why are we saving bits 20-16 and 15-11 of the instruction? How are they used, and why do we not save the whole instruction?*

ID/EX: Pipeline (Control) Registers

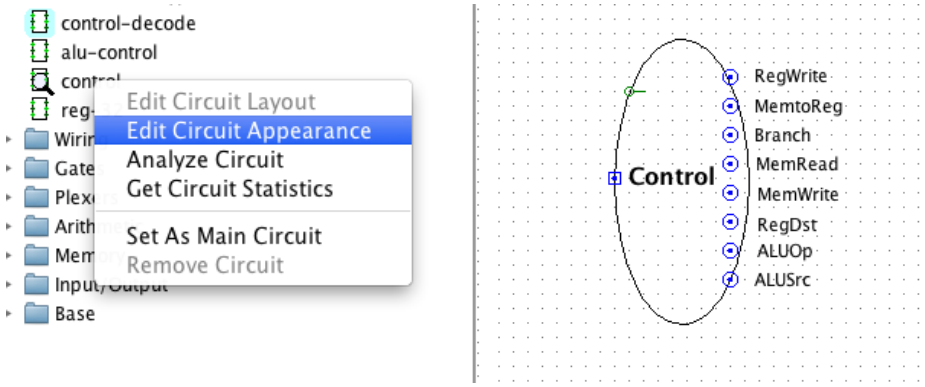
Here is the first place where we need to save the control signals for later stages of the execution. If you take a closer look at the book, you'll see the following figure:



In this figure the control signals are split up by which stage uses them.

The control unit supplied is the same as on the previous lab, but be careful! The order of these signals does not correspond to the order in the figure above. This means that the first two signals of the control unit you have implemented are not necessarily the same two first control signals described in the picture.

Your first job is to change the control circuit so the signals come out in the order in which you will use them, as in the figure above. One way of doing this is to right-click the control unit, and select the "Edit Circuit Appearance" option from the menu.



This will let you re-order the output pins of the control unit as you need. The second alternative is to reorder the registers that you need for the control signals, but we do not recommend this, since it creates a lot of unaesthetic wire-mess.

> **Question 2:** *For each of the signals from the control circuit, explain why is it a write-back (WB), memory (MEM) or execution signal (EX).*

After re-organizing your control circuit, you need to add pipeline registers for the signals as well. As the signals are only 1 or 2 bit width, is basically the same as adding just flip-flops. Take a look at the one that is already placed, and try to replicate it aligned to the corresponding outputs.

EX/MEM and MEM/WB

Now that you are gaining experience with adding registers, you can just repeat the procedure for the next two stages on the pipeline. Remember to identify which control and data signals are needed in each stage and size the registers appropriately. Notice that the control signals are not always on top of the data registers, as in the case of the *Zero* signal.

Step 2: Hooking Up the Pipelined Datapath

Now that you have all the registers aligned and placed, it's time to hook up the datapath. Use the figure from the book as a reference. You should not delete anything we've put in there. Notice that we kept the debugging "probes" we had, and we added some extras.

Please don't forget to connect the clock to all the registers! This is very important, as otherwise nothing will work. (Remember, the input of the pipeline registers is only transferred to the output on the rising edge of the clock signal. If you don't have a clock then the data will never go anywhere!) Double check that all the wires are connected to the right place, as this could cause a lot of debugging headaches!

Step 3: Testing it with Code

To test your processor you were provided with three test programs (*test-code-1.mem*, *test-code-2.mem*, and *test-code-3.mem*).

Test code 1

```
lw $t1, 0($zero)
sw $t2, 0($zero)
```

Test code 2

```
lw $t1, 0($zero)
sw $t1, 12($zero)
```

Test code 3

```
lw $t1, 0($zero)
lw $t2, 4($zero)
add $t3, $t1, $t2
add $t3, $t1, $t3
sw $t3, 12($zero)
```

For each program, answer the following questions:
Question 3: *What does the program do?*
Question 4: *If you execute the code on your pipelined processor, does it work? (Does it produce the expected result?)*
Question 5: *If the answer to Question 4 is NO, explain why?*
Question 6: *If the answer to Question 4 is NO, change the code so it will work. (Hint: you can use the online MIPS assembler[a] to compile new code and save it in your own .mem file to test.)*

[a] `http://alanhogan.com/asu/assembler.php`

That's it

Show your solution to a TA in a Zoom meeting (more info on Studium). Be prepared to show what you have done to complete the pipelining and to answer questions. A clean solution will be easier on everyone.

BONUS TASK: Forwarding

> Note: You will pass the lab without solving the bonus task. The bonus task is here for you to explore how forwarding is implemented.

By now, you should have realized that some of the programs we gave you in the previous section will not function properly without stalling the pipeline.

Let's look at a simple example in order to understand why:

```
add $t1, $t2, $t3
add $t4, $t5, $t1
```

The first instruction stores the result of the addition into register `$t1`, and the second instruction reads from `$t1` and `$t5` and stores the result into `$t4`. The first instruction writes back the result into the register file at the WB stage. Meanwhile, the second instruction is already in the MEM stage (and thus has already read its input from the register file). This is called a *data hazard*.

In the previous section, you worked around this issue by introducing pipeline stalls (or "bubbles"). Here, we will see how we can avoid stalling the pipeline, thus improving performance, by using *operand forwarding*.

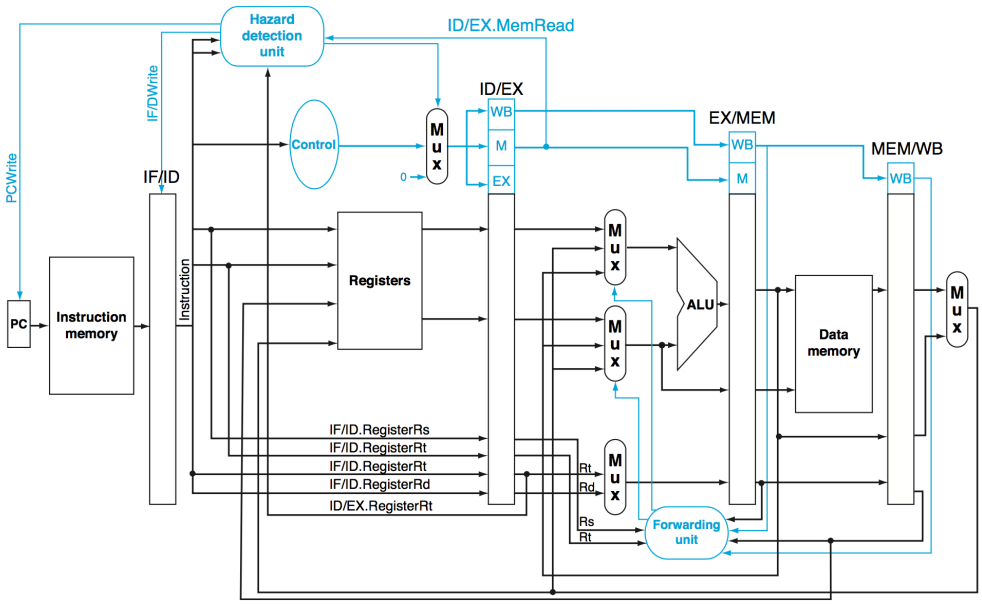
Operand forwarding consists in feeding the output of a stage directly into a previous stage. In the example we just looked at, we can see that the result of the first instruction is computed during the EX stage. Instead of stalling the pipeline until the result gets written back to the register file, we can directly forward it to the previous stage, which can then use it immediately.

Operand forwarding can be implemented between different stages. We will focus on implementing it between the MEM and EX stages, so that the result from the ALU can be fed directly back into the ALU at the next clock cycle.

By looking at the datapath, we can see that the ALU so far has two inputs (let's call them A and B):

1. A comes from the register file (ReadData1).
2. B comes either from the register file (ReadData2), or from an immediate encoded in the instruction. A multiplexer, driven by the control unit, chooses one or the other based on the instruction type.

By introducing forwarding, one of the ALU inputs could also come from the next pipeline stage. Choosing the right source for the ALU inputs will be done, as you have probably guessed, using two multiplexers.



The Forwarding Unit

The operand must be forwarded only when a data hazard is detected. In order to drive these two multiplexers choosing the ALU inputs, we will add yet another control unit, that we call *forwarding unit*. Its job is twofold:

1. Detect when a data hazard occurs.
2. Control the two ALU multiplexers.

In order to control the two multiplexers, we introduce two new signals: *ForwardA* and *ForwardB*, indicating respectively where the data for inputs A and B comes from:

- Input A comes from the EX/MEM pipeline register if the previous instruction's writes into a register (RegWrite=1) and the destination register is the same as the current instruction's `rs` register.
- Input B comes from the EX/MEM pipeline register if the previous instruction's writes into a register (RegWrite=1) and the destination register is the same as the current instruction's `rt` register.

Your job for this step is to:

1. Add/modify the two multiplexers.
2. Add the forwarding unit and the hazard detection unit.
3. Wire everything properly.

Question 7: *With our forwarding added, have we now eliminated all need for pipeline stalls in the given test programs? If not, explain why, and how the pipeline could be further improved.*

Revisions

2013-10-26 — version 1.0 — Fall 2013 — Germán Ceballos, Ricardo Alves
2013-10-31 — version 1.1 — Fall 2013 — David Black-Schaffer
2013-12-16 — version 1.2 — Fall 2013 — Germán Ceballos, Ricardo Alves
2014-10-01 — version 2.0 — Fall 2014 — Germán Ceballos
2016-09-01 — version 2.1 — Fall 2016 — Moncef Mechri (Added forwarding)
2017-09-13 — version 2.2 — Fall 2017 — Ricardo Alves, Germán Ceballos
2018-10-01 — version 2.3 — Fall 2018 — Ricardo Alves, Germán Ceballos
2020-04-27 — version 2.4 — Spring 2020 — Per Ekemark, Christos Sakalis
2021-04-23 — version 2.5 — Spring 2021 — Muhammad Hassan