"An excellent plumber is infinitely more admirable than an incompetent philosopher. The society that scorns excellence in plumbing because plumbing is a humble activity and tolerates shoddiness in philosophy because it is exalted activity will have neither good plumbing or good philosophy. Neither its pipes or its theories will hold water." – J. Gardner

Learning Objectives

1. Learn the fundamentals of pipelining implementation, including **data forwarding**, **stalling**, and **control hazards**.

Work that needs to be handed in on GitHub

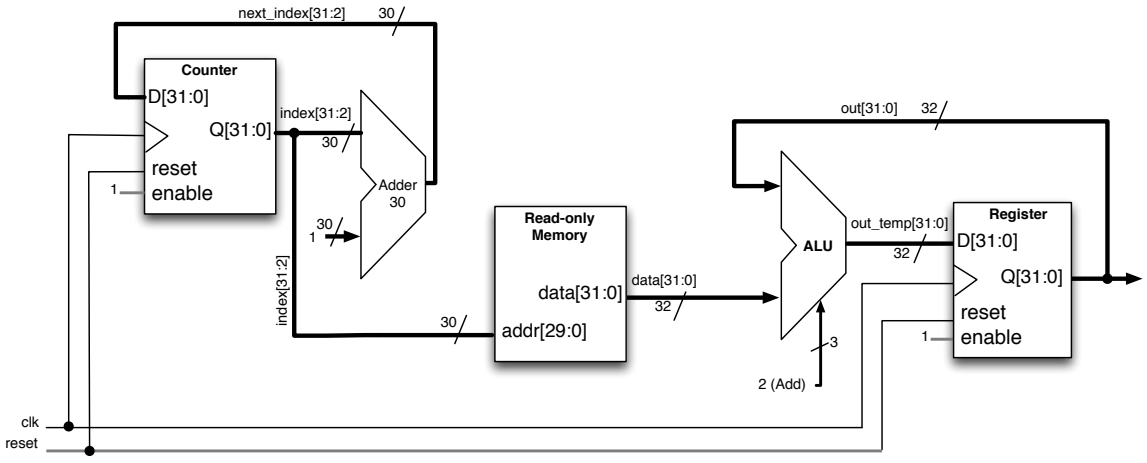
1. **First Deadline:** `pipelined_adding_machine.v`, A 2-stage pipelined implementation of a circuit that adds a bunch of numbers stored in memory.
2. **Second Deadline:** `pipelined_machine.v`, A 3-stage pipelined implementation of a MIPS ISA subset machine.

Guidelines

- As always, make sure to read over the handout carefully to understand the circuit before attempting to implement it. Take special note of the diagrams provided, as they may give clues into necessary additions/modifications
- Alongside the handout diagrams, note that the starter files implement an unpiped version of their respective circuits. Your code will need to interact and build off the existing code; use it to help in accomplishing your goals.
- Unlike the previous lab, partners are once again allowed for this lab.

First Deadline: A 2-stage pipelined adding machine

Below is a circuit diagram of a simple adding machine. The values to be added are loaded into the memory (process not shown) and the machine is run. The counter iterates through the memory addresses, one per cycle. As each memory value comes out, it is accumulated into the register.



Implementation Details

1. Compile and run this machine using `make pipelined_adding_machine`. You will be given the correct output for the machine. This can be confirmed by looking at the verilog and the `things_to_add.data.dat` file.

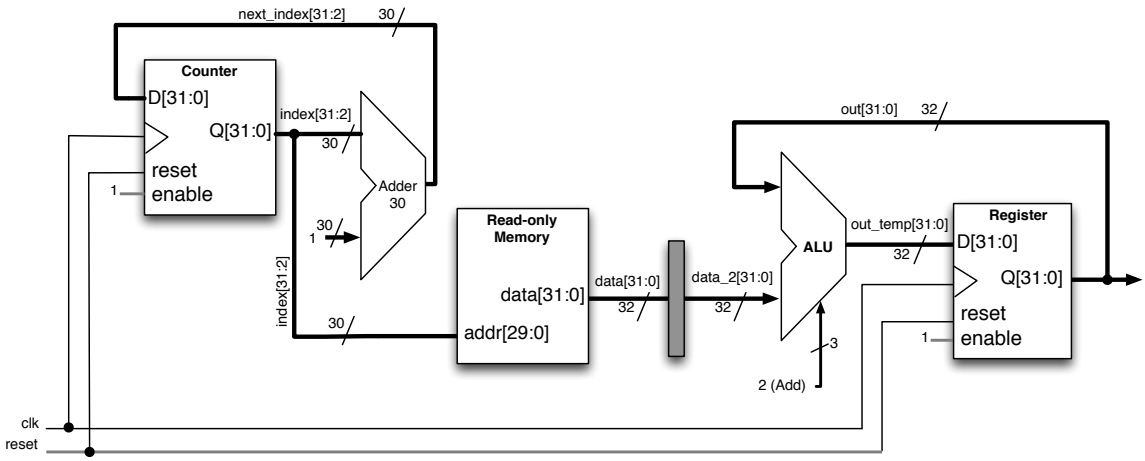
2. Because of the delays of the components, this machine can't run at faster than 4 time units per clock cycle. This is seen at our original clock period of 2. **In the `pipelined_adding_machine_tb.v` testbench, change `"#2"` to `"#1"` so it looks like:**

   ```
   always #1 clk = !clk;
   ```

   (this new statement will cause each phase of the clock to be 1 time unit, so the clock period will be 2 time units.)

   Re-compile/run and see that the output to see that the output is different. We've clocked our circuit too fast, so it is giving incorrect results.

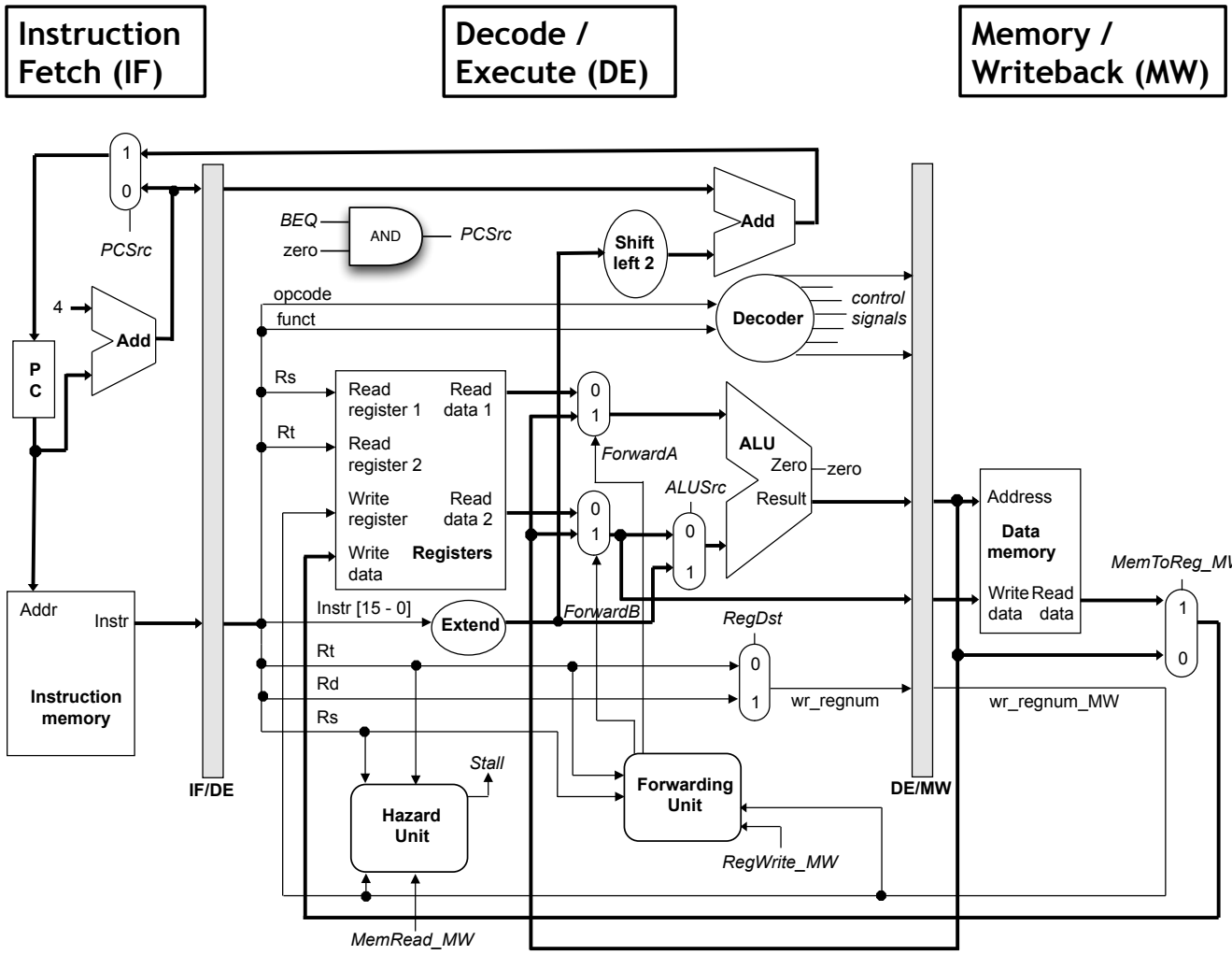
3. We can make the circuit work at this higher clock frequency by pipelining it, as shown below. Be sure to name the two copies of the `data` bus differently, and connect them properly. Make sure that your pipelined version behaves the same as the original version, but does so at the higher clock frequency as well.



Notes for First Deadline

- Remember that each stage of the circuit will execute in parallel, separate from one another. Once each stage receives the correct inputs and passes on the correct outputs, the entire pipeline will act as intended.

Second Deadline: A 3-stage pipelined MIPS machine



Using the provided Verilog code for a single-cycle machine, implement the 3-stage pipeline shown above in Verilog.

Forwarding, stalling and flushing

Forwarding, stalling, and flushing all work similarly to the 5 stage pipeline from lecture, but we will have to make some logical modifications for a 3 stage pipline.

For forwarding, unlike the 5 stage pipeline from lecture, we don't implement register file writes in the first half and reads in the second half. (Doing so would require the clock cycle to be significantly extended). Therefore, we need a forwarding path from the DE/MW pipeline register providing inputs to the ALU. Additionally, with only one forwarding path, our forwarding unit will consist of only the boolean expressions for control signals `ForwardA` and `ForwardB`. **Make sure your expressions guard against forwarding `$0`.**

Forwarding, stalling and flushing, cont.

For stalling, we'll stall in the same situations that the 5-stage pipeline from class does. In these cases, we'll need to repeat the IF and DE stages in the next cycle and convert the MW stage of the next cycle to a NOP. Our register modules have enable and synchronous reset inputs, which will come in handy for repeating and NOPing stages, respectively. The hazard unit will consist solely of the boolean expression for the `Stall` signal. **Remember to check for Rt being a source in your expression and guard against stalling for `$0`.**

For flushing, we just need to convert the DE stage of the next cycle to a NOP, since branches are resolved in the DE stage. Once again, the synchronous reset input of our register modules will come in handy.

Implementation Details

1. We suggest that you implement this assignment in four steps, and we've provided inputs to facilitate testing each of these steps. Feel free to modify any of the assembly files to add your own test cases, and they'll get reassembled the next time you run the appropriate `make` command (see the next section). If you're on your own machine, you'll need to download `spim-vasm` from the wiki for the assembling.
   (a) Pipeline the datapath ignoring data dependences, stalls, and branches. Don't worry about adding the forwarding or stalling logic yet and don't worry about flushing for the moment (since the first input doesn't have any hazards or branches). Just implement the pipeline registers and make sure your code works for the `no_hazards_or_flushing.s` input. Once this is working, commit this, so if you have a later bug you can always diff/revert to this version.
   (b) Implement forwarding. You can test this code using `forwarding.s`. Commit.
   (c) Implement stalling. You can test it using `stalling.s`. Commit.
   (d) Implement flushing. Test this code using `flushing.s`. Again, commit.
   (e) Finally, test using `pipeline_test.s`, which includes both data hazards and taken branches.
2. Although the pipeline registers are drawn as monolithic entities, we'd suggest breaking this into a collection of smaller registers, one for each signal.
3. Since some signals will exist in multiple pipeline stages (*e.g.*, `wr_regnum`) you should develop a naming scheme that allows you to distinguish the names of the signals. One suggestion is to append `_MW` to the end of all the signals in the MW stage.
4. To test if your pipelining is working correctly, **reduce the clock period in the test bench from 12 to 4** and check if your code is still producing the right outputs; an incorrectly pipelined machine will not. (Something to think about: why does the DE stage only need 4 time units and not 5?)

Notes for Second Deadline

- The Verilog provided implements a single-cycle MIPS machine which supports the following instructions: ADD, SUB, AND, OR, SLT, LW, SW, BEQ
- There are some differences between this machine and the one that we implemented in Lab 6, including:
  - We're providing a totally different ALU, which implements SLT internally and has a different encoding for its control inputs.
  - We've provided discrete adders so that we didn't have to use ALUs for computing the next PC.
  - The registers have a **synchronous** reset. This can be useful for implementing stalling and flushing.
  - When reset, the registers (except register 0) all start with the value `0x10010000`, which is the base address of the `.data` segment.
  - We've provided a working decoder; this code uses some goofy Verilog syntax, so don't be distressed if it doesn't make sense to you. You don't need to change it.
  - The control for the next PC mux (`PCSrc`) is provided outside the decoder.
- Also, we've added delays to all of the major components, using Verilog's delay notation: #N means the output should be updated N time units after its input changes. The latencies are as follows: instruction and data memories (3), adders/ALUs (2), register file read/write (1), decoder (2). Thus the baseline machine needs a clock period $\geq$ 11 time units. Hence, in `pipelined_machine_tb.v`, the clock alternates every 6 time units, giving a 12 time unit clock period.

Compiling, Running and Testing

We have given you a Makefile for setting up the test cases we have provided and compiling and running your machine.

Before you compile your code, you need to set up the test case you want to use. To do that, use one of the following:

- `make no_hazards_or_flushing`
- `make forwarding`
- `make stalling`
- `make flushing`
- `make pipeline_test`

Then, compile and run your code using the command `make pipelined_machine`. Notice that the assembly programs that we provide contain comments that specify the behavior of the program. You can use these comments to verify that your pipelined machine has indeed the expected behavior.