"As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now [1972] that we have gigantic computers, programming has become a gigantic problem. As the power of available machines grew by a factor of more than a thousand, society's ambition to apply these new machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between the ends and the means. The increased power of the hardware, together with the perhaps more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them and even worse, he had to transform such dreams into reality! It is no wonder that we found ourselves in a software crisis." – E. Dijkstra

"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

Learn the fundamentals of pipelining implementation, including:

1. Data Forwarding
2. Control Hazards

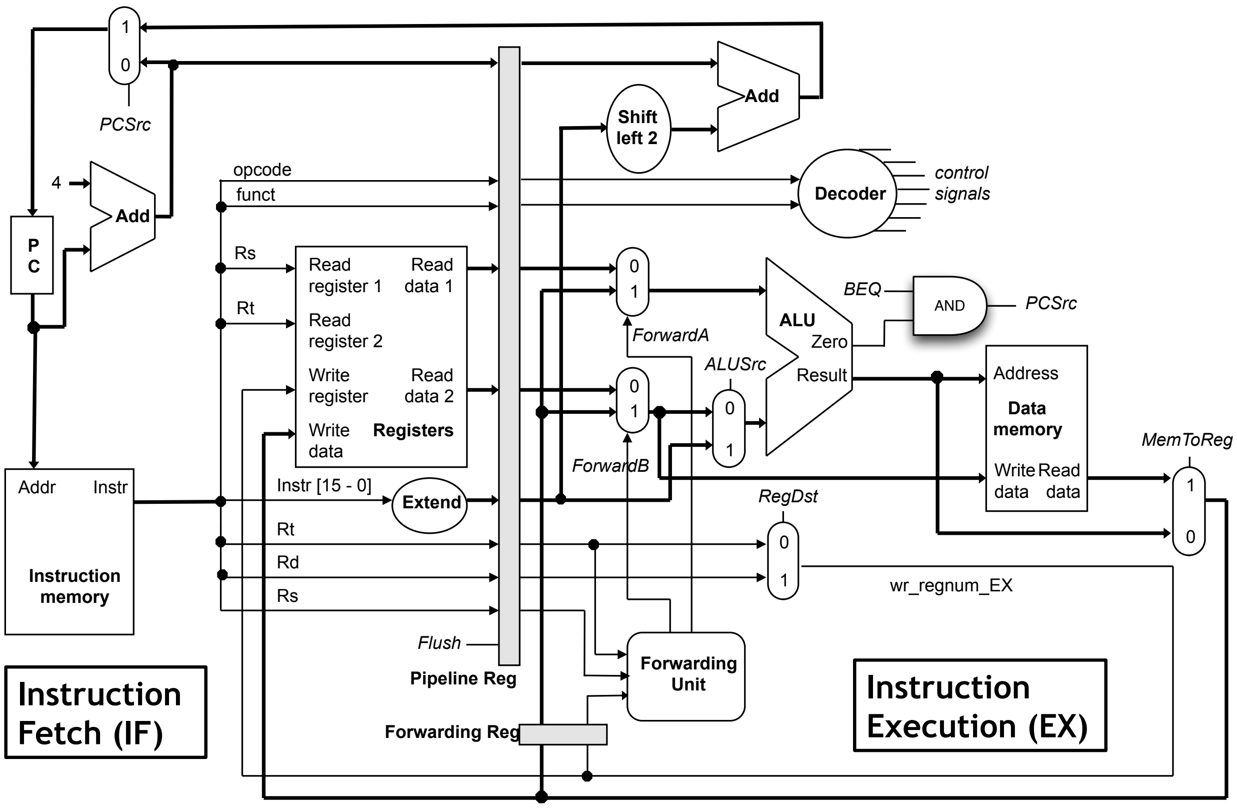
Work that needs to be handed in (via SVN)

1. `pipelined_machine.v`: A 2-stage pipelined implementation of a MIPS ISA subset machine.

Important Information

- We're providing you a Verilog implementation of a single-cycle MIPS machine that implements 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 Lab6, 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 used for flushing the `opcode` and `funct`.
  - 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.
  - 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: memories (2), adders/ALUs(2), register file read/write (1). Thus the baseline machine needs a clock period ≥ 8 time units. Hence, in `pipelined_machine_tb.v`, the clock alternates every 4 time units, giving an 8 time unit clock period.
- You will have an extra week to complete this lab, but we'll also be handing out the SPIMbot lab assignment so don't put this off.

A 2-stage pipeline



Using the provided Verilog code for a single-cycle machine, implement the 2-stage pipeline shown above in Verilog.

Helpful suggestions

1. We suggest that you implement this assignment in 3 steps, and we've provided inputs to facilitate testing each of these steps.
   (a) Pipeline the datapath ignoring data dependences and branches. Don't worry about adding the forwarding logic yet and hardwire PCSrc to 0. Just implement the pipeline registers implemented and make sure your code works for the `no_deps_or_beq.s` input. Once this is working SVN commit this, so if you have a later bug you can always diff/revert to this version.
   (b) Add branches. Get branch resolution working by hooking up PCSrc and implementing flushes. Test this code using `beqs_but_no_deps.s`. Again, commit.
   (c) Add forwarding. You can test this code using `deps_but_no_beqs.s`.
   (d) 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 both pipeline stages (*e.g.*, opcode) you should develop a naming scheme that allows you to distinguish the names of the signals. One suggestion is to append "_EX" to the end of all the signals in the EX stage.
4. Given the component latencies on the front page, what is the minimum clock period of the pipelined machine? (Test it with that clock period! We will!)

Discussion Questions

1. If the functional units had the following latencies, what is the shortest clock period could this pipeline run at?

Structure	**Latency**
Memory	3ns
Register Read	1ns
Adder	1ns
ALU	2ns
Register Write	1ns

2. In what pipeline stage are branches resolved? Does this machine have a taken branch penalty? If so, how many cycles?

3. For the following code, compute the value of the forwarding signals. Assume the branch is not taken.

Instruction	`forwardA`	`forwardB`
`add $8, $5, $5`		
`add $2, $5, $8`		
`sub $3, $8, $4`		
`lw $2, 0($3)`		
`beq $8, $2, target`		
`and $7, $2, $8`		
`sw $7, 0($2)`		
`or $9, $7, $2`		

4. Write a boolean expression for the `forwardA` control signal

   `forwardA` =

5. Write a boolean expression for the `Flush` control signal

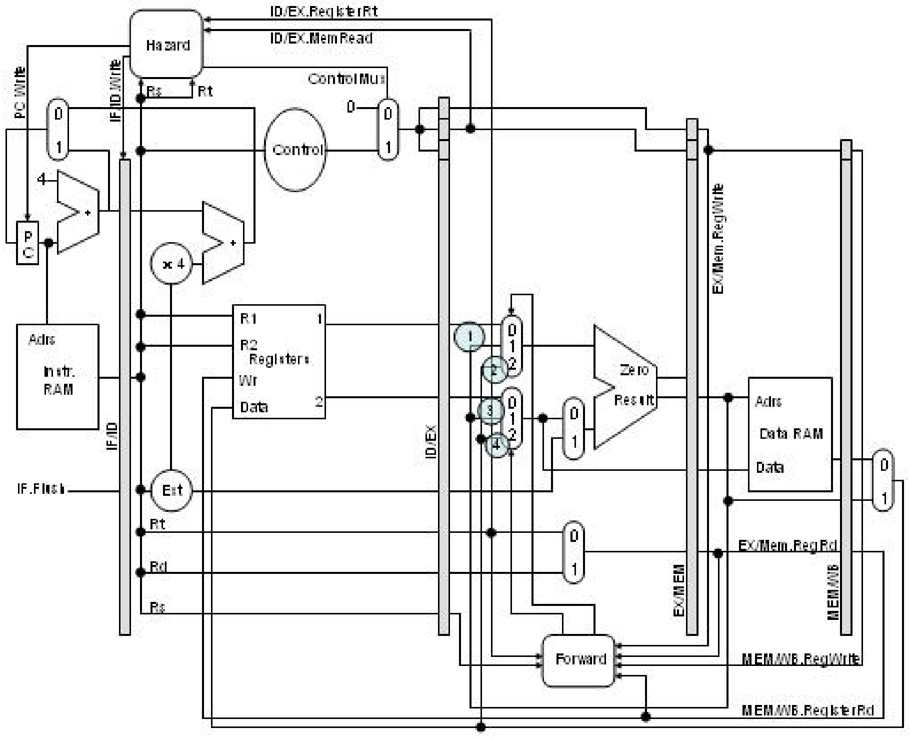
   `Flush` =

1 Warmup: Stalling vs. Forwarding

Suppose we have the following chunk of code containing only R-type instructions.

```
add $8,$5,$5
add $2,$5,$8
sub $3,$8,$4
add $2,$3,$2
```

1. Identify the hazards involved (draw the arrows between *dependencies* that cause data hazards).
2. Figure below shows the pipelined datapath with four forwarding inputs. For each dependency identified above specify which numbered forwarding path is used.



3. Fill out the following pipeline diagram, for the case when forwarding is implemented.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`add $8,$5,$5`															
`add $2,$5,$8`															
`sub $3,$8,$4`															
`add $2,$3,$2`															

4. Fill in the pipeline diagram below, for the case when forwarding is **not** implemented.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`add $8,$5,$5`															
`add $2,$5,$8`															
`sub $3,$8,$4`															
`add $2,$3,$2`															