1 Stalling vs. Forwarding

In class, we've seen how some types of **data hazards** can be resolved with **forwarding** (or bypassing). The idea behind forwarding is simple: if a register `$r` has not yet been updated with its "new value" by the time a later instruction needs the value of `$r`, the data can be *forwarded* if it is available **at** or **before** the time it is needed. This involves extra hardware in the form of wires, hazard-detection, additional muxes, etc. As we shall see, forwarding **cannot always** eliminate data hazards.

A simple way of eliminating data hazards that is **guaranteed** to work is *stalling*. Once again, the idea behind stalling is simple: if an instruction i_1 writes to a register `$r` and a later instruction i_2 needs to read the value of `$r`, instruction i_2 is *stalled* (i.e. delayed) until i_1 finishes updating `$r`. More specifically, i_2 must be stalled by as many cycles as are necessary to ensure that i_1's WB stage occurs **at** or **before** the the cycle in which i_2's ID stage occurs.

Problems

1. **Single-cycle vs. pipelined implementation.**

   Suppose the latencies of the key components of a *single-cycle* processor are as follows:

   Instruction/data memory: 100ps, register file read/write: 200ps, ALU operation: 200ps

   Assume that the latencies of all other components are negligible. Now suppose that the above processor is pipelined into five stages: IF (uses instruction memory), ID (uses register file), EX (uses ALU), MEM (uses data memory), and WB (uses register file).

   (a) What is the speedup obtained from pipelining?

   (b) If the time for ALU operations can be shortened by 25%, will it affect the speedup obtained from pipelining? If yes, by how much? If no, why not?

   (c) What is the speedup due to pipelining if the ALU operations now takes 25% more time?

2. **Eliminating hazards without forwarding or stalling.**

   StingyMIPS is a 5-stage pipelined implementation of MIPS *without* forwarding. The code below contains *data hazards*.

   Rewrite this code so that it does the same thing on StingyMIPS as on regular MIPS, but runs without *stalls* on StingyMIPS. A stall delays every subsequent instruction by 1 cycle.

   Initial code: Write your code here:

```
add $1, $2, $3
add $4, $1, $3
add $5, $6, $3
add $7, $8, $3
```

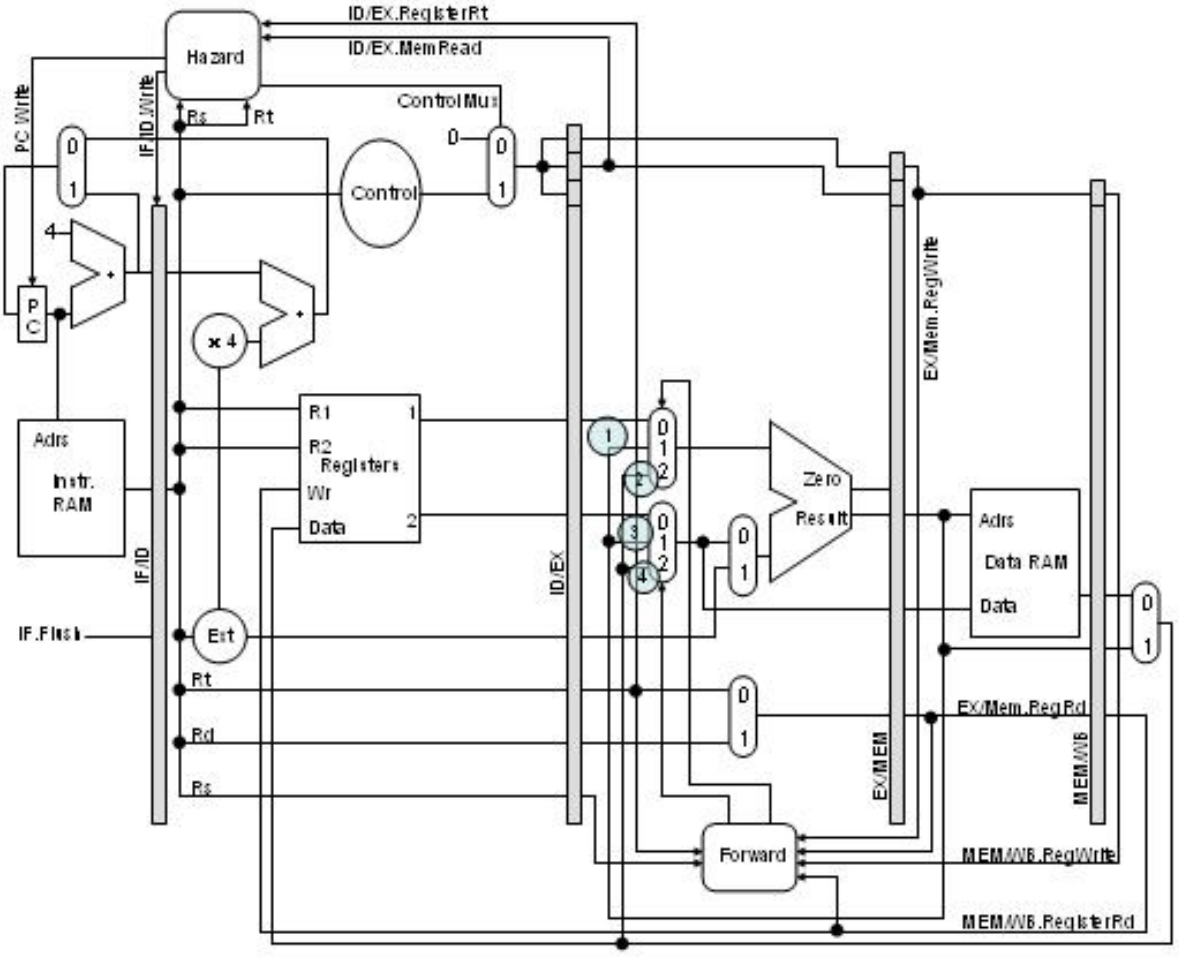
3. **Pipeline diagrams.**

   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,$2,$3
```

   (a) Identify the hazards involved (draw the arrows between *dependencies* that cause data hazards).

   (b) Figure below shows the pipelined datapath with four forwarding inputs. For each dependency identified above specify which numbered forwarding path is used.



(c) Fill out the following pipeline diagram, for the case when forwarding is implemented.

Table 1. Pipeline Diagram with Forwarding

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,$2,$3`															

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

Table 2. Pipeline Diagram without Forwarding

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,$2,$3`															

4. **Pipelined processors.**

A pipelined processor has k pipeline stages. Assuming no stalls, how many cycles are required to execute n instructions?