CS 333: Pipelining Class 2

I. Pipelining Review

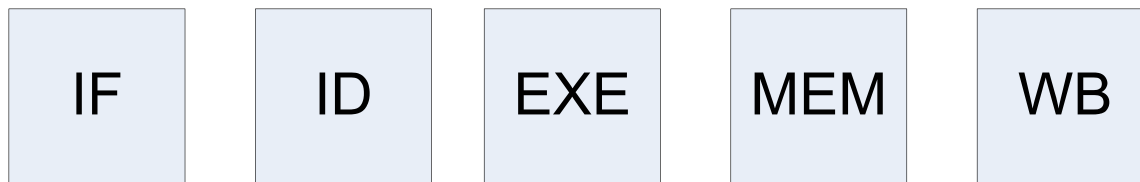
A. Pipelining

1) Implementation technique that allows multiple instructions to be **overlapped in execution**
2) Basic idea
   a) Subdivide instruction handling and execution into subtasks which can be separate **stages**
      i) Each stage must take equal time (controlled by clock). Limited by time for **slowest** stage.
   b) Each stage operates concurrently on a different instruction (works as long as each stage has separate resources.
3) Pipelining:
   a) **Throughput** – improves the completion rate
   b) **Latency** – (execution time of a single instruction) is not helped
   c) Has better effect when the number of insts is larger
      i) In other words, if you can keep the pipeline full longer, the more benefit. There is **overhead** (time when resources are not fully utilized) when **filling** and **draining** the pipeline.
4) Single-cycle vs. pipelined performance. If stages are perfectly balanced, under ideal conditions (no hazards) and large number of instructions:

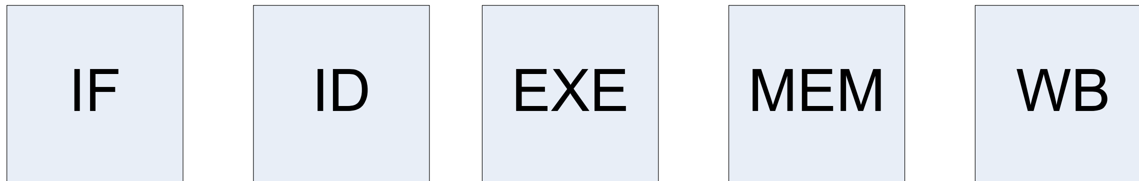
$$TimeBetweenInstructions_{pipelined} = \frac{TimeBetweenInsts_{nonpipelined}}{NumberOfPipeStages}$$

5) Subdividing instruction fetch and instruction execute into stages: Example:
   a) **Instruction Fetch (IF)** – fetch instruction from memory to IR
   b) **Read registers, decode instruction (ID)** – read registers and decode IR fields
   c) **Execute operation or calculate address (EXE)** – ALU operation or address calculation for load or store
   d) **Access operand in memory (MEM)** – access memory
   e) **Write results into a register (WB)**

II. Introduction to pipeline diagramming conventions



First half of clock cycle: **Writes**
Second half of clock cycle: **Reads**
W R W R W R W R



III. Pipeline Hazards

A. Structural Hazards

1) hardware cannot support the instruction combination (not enough resources)

B. Data Hazards

1) pipeline needs to be stalled because one step must wait for another to complete

2) Types of hazards

a) RAW (Read after Write)

add r0, r1, r2
sub r4, r3, r0

b) WAW (Write after Write)

i) (matters if instructions allowed to execute out of order)

ii) Can be avoided with register renaming

add r0, r1, r2
sub r0, r4, r5

c) WAR (Write after Read)

i) (matters if instructions allowed to execute out of order)

ii) Can be avoided with register renaming

add r2, r1, r0
sub r0, r3, r4

3) Example 1: **Read after Write (RAW)**

add **r0,** r1, r2
sub r3, **r0**, r4

Draw diagram of these 2 insts going through 5 stage pipeline:

add r0, r1, r2
IF: IR ← add r0, r1, r2
ID: decode: r0, r1, r2, **read:** r1, r2
EXE: r1 + r2
MEM: not needed
WB: r0 ← r1+r2

sub r3, r0, r4
IF: IR ← sub r3, r0, r4
ID: decode: r3, r0, r4, **read:** r0, r4

EXE: r0+r4
MEM: not needed
WB: r3 ←r0-r4

Resolution technique #1: Stalling (inserting bubbles)
Resolution technique #2: Forwarding/Bypassing
pass result of EXE back to EXE

4) Example 2: Result forwarding does not work in all cases

ld r0, 0(r31)
sub r2, r0, r3

ld r0, 0(r31)
IF: IR = ld r0, 0(r31)
ID: **decode:** r0, 0, r31; **read:** r31
EXE: 0(r31)
MEM: read mem[0(r31)]
WB: r0 = mem[0(r31)]

sub r2, r0, r3
IF: IR = sub r2, r0, r3
ID: **decode:** r2, r0, r3, **read:** r0, r3
EXE: r0-r3
MEM: not needed
WB: r2 = r0-r3

Without forwarding result from Mem, need to stall 2 cycles.
Try forwarding MEM result, still have to stall for one cycle.

Resolution technique #3: Code reordering
Example: Assume we have forwarding from EXE and MEM

ld r1, 0(r31)
ld r2, 4(r31)
add r3, r1, r2
st r3, 12(r31)
ld r4, 8(r31)
add r5, r1, r4
st r5, 16(r31)

What are the hazards? both adds
Solution? Reorder the code to eliminate hazards (finishes 2 fewer cycles)

ld r1, 0(r31)
ld r2, 4(r31)
ld r4, 8(r31)
add r3, r1, r2
st r3, r1, r2
add r5, r1, r4
st r5, 16(r31)

C. Control Hazards

1) Need to make a decision based on the results of one instruction while others are executing
2) Example

add r4, r5, r6
brzr r1, r2
or r7, r8, r9

Draw diagram to demonstrate the hazard (IF needs result of condition of branch, but it's not ready for next inst)

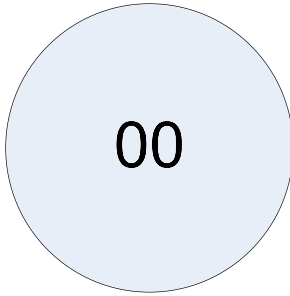
One solution: stall:
Another solution: static branch prediction
Branch directions: taken (go to target address)
not taken (go to PC + 4)

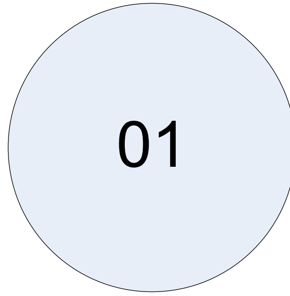
If correct, proceed full speed
If incorrect, stall the pipeline

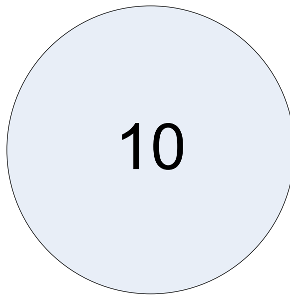
(Pipeline needs 1) a way to check correctness of prediction and 2) a way to ensure that wrongly guessed branch direction won't have an effect, 3) a way to restart pipeline at the correct address)

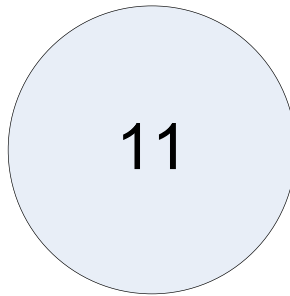
More advanced solution: Dynamic hardware branch prediction

bimodal branch prediction


00


01


10


11

Try these sequences:

1. lw r0, 0(r0)
   add r1, r0, r0

2. add r1, r0, r0
   addi r2, r0, 5
   addi r4, r1, 5

3. addi r1, r0, 1
   addi r2, r0, 2
   addi r3, r0, 2
   addi r3, r0, 4
   addi r5, r0, 5