Jon Dis 12 Notes

CS 61C Fall 2018

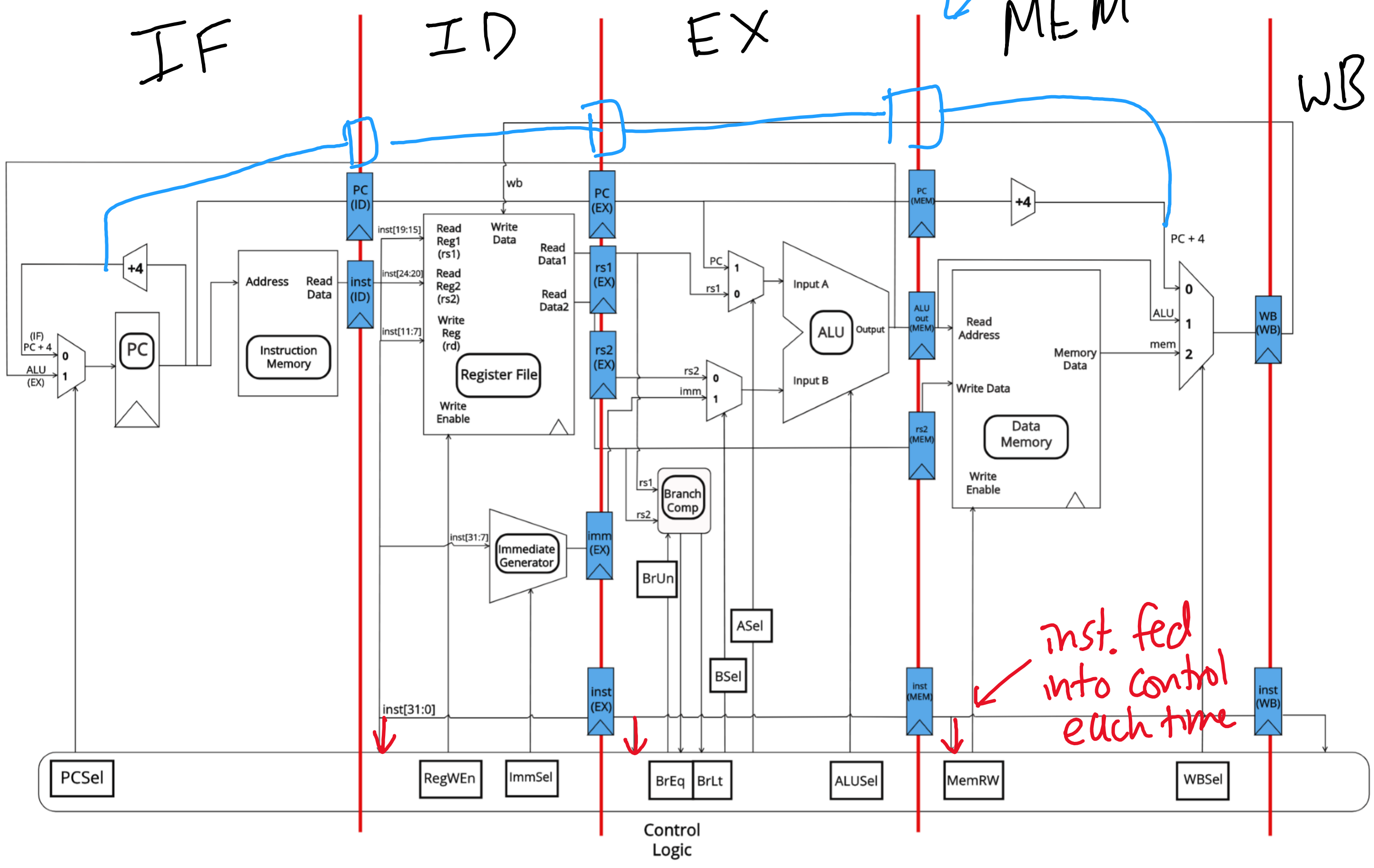
Pipelining

Discussion 12: November 12, 2018

1 Pipelining Registers

In order to pipeline, we add registers between the five datapath stages. Label each of the five stages (IF, ID, EX, MEM, and WB) on the diagram below.

more registers



1.1 What is the purpose of the new registers?

store data & keep consistent @ each stage - "snapshot" of that phase

1.2 Why do we add +4 to the PC again in the memory stage?

don't need extra registers when passing through see ▨

1.3 Why do we need to save the instruction in a register multiple times?

needs correct control signal see ▨

2 Performance Analysis

Register clk-to-q 30 ps	**Branch comp.** 75 ps	**Memory write** 200 ps
Register setup 20 ps	**ALU** 200 ps	**RegFile read** 150 ps
Mux 25 ps	**Memory read** 250 ps	**RegFile setup** 20 ps

2.1 With the delays provided above for each of the datapath components, what would be the fastest possible clock time for a single cycle datapath?

See soln & prev dis WS 950ps → 1.05 GHz

2.2 What is the fastest possible clock time for a pipelined datapath?

find time @ each stage
- same as single cycle, but b/w registers

soln: max time of all stages = 325 ps from MEM

2.3 What is the speedup from the single cycle datapath to the pipelined datapath? Why is the speedup less than 5?

$\frac{950}{325} = 2.9\times$ speedup

3 Hazards

One of the costs of pipelining is that it introduces three types of pipeline hazards: structural hazards, data hazards, and control hazards.

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. There are two main causes of structural hazards:

Register File The register file is accessed both during ID, when it is read, and during WB, when it is written to. We can solve this by having separate read and write ports. To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle, and read from it during in the second half. This is also known as double pumping.

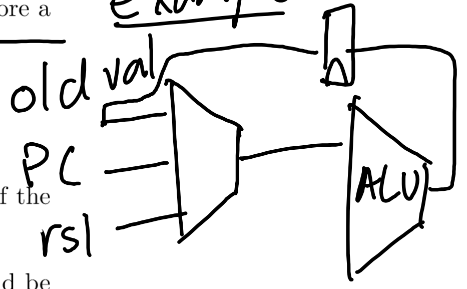
Memory Memory is accessed for both instructions and data. Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards

Data hazards are caused by data dependencies between instructions. In CS 61C, where we will always assume that instructions are always going through the processor in order, we see data hazards when an instruction **reads** a register before a previous instruction has finished **writing** to that register.

example



Forwarding

Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.

3.1 Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

Instruction	C1	C2	C3	C4	C5	C6	C7
1. `addi t0, a0, -1`	IF	ID	EX	MEM	WB		
2. `and s2, t0, a0`		IF	ID	EX	MEM	WB	
3. `sltiu a0, t0, 5`			IF	ID	EX	MEM	WB

forward values (controller muxes between the inputs)

3.2 Imagine you are a hardware designer working on a CPU's forwarding control logic. How many instructions after the first addi instruction above could be affected a potential data hazard created by this addi instruction?

3 instruction

EX MEM WB
1) IF ID ...
2) IF ID...
3) IF ID...
4) IF ID ← ok now

all not ready for ID phase yet

3.3 You have the signals rs1, rs2, RegWEn, and rd for two instructions, instruction n and instruction $n+1$. Write a condition you can check to see if there is a data hazard between the two instructions, in terms of these signals.

if ((rs1(n+1) == rd(n) || rs2(n+1) == rd(n)) && RegWEn(n) == 1)

//forward value

(rs1 uses WB value) (rs2 uses WB val) with write enabled

Stalls

3.4 Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8
1. `addi s0, s0, 1`	IF	ID	EX	MEM	WB			?
2. `addi t0, t0, 4`		IF	ID	EX	MEM	WB		
3. `lw t1, 0(t0)`			IF	ID	EX	MEM	WB	
4. `add t2, t1, x0`				IF	ID	~~EX~~	MEM	WB

Can't forward this - loading from mem
:(:(:(

{4) NOP
{5) add t2 t1 x0 IF ID EX MEM WB
⇒ insert NOP + forward

addi ⟩ forward
lw
addi) forward (addi stalls like NOP, but useful work)
add

3.5 Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?

New order 2 - 3 - 1 - 4
order doesn't matter, don't waste time w/ NOP

Control Hazards

Control hazards are caused by **jump and branch instructions**, because for all jumps and some branches, the next PC is not PC + 4, but the result of the computation completed in the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

3.6 Besides stalling, what can we do to resolve control hazards?

predict where branch goes, if predict wrong, flush pipeline (get rid of all pre computed instructions)

Extra for Experience

3.7 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards from all pairs of instructions.

How many stalls would there need to be in order to fix the data hazard(s)? What about the control hazard(s)?

Data
Data
Data
Control

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9		
1. `sub t1, s0, s1`	IF	ID	EX	MEM	WB						
2. `or s0, t0, t1`		IF	ID	~~EX~~ IF	~~MEM~~ ID	~~WB~~ EX	MEM	WB			
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	~~WB~~ IF	ID	EX	MEM	WB
4. `bgeu s0, s2, 1`				IF	ID	EX	MEM	~~WB~~ IF	ID	EX	...
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB		

2 stalls
2 stalls

good from stalling

Only Stall

if we handle [illegible] w/ Branch predictor → no stalls, otherwise stall 5 by 3 (after Ex of branch)