EECS 370 - Lecture 14

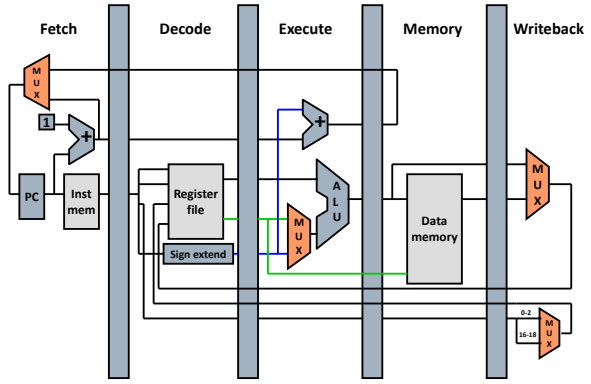
Pipelining and Data Hazards II

Live Poll + Q&A: slido.com #eecs370 *Poll and Q&A Link*

Review: Pipelining

- Goal:
  - Achieve low clock period of multi-cycle processor…
  - … while maintaining low cycles-per-instruction (CPI) of single cycle processor (close to 1)
  - Achieve this by overlapping execution of multiple instructions simultaneously

Review: Our new pipelined datapath



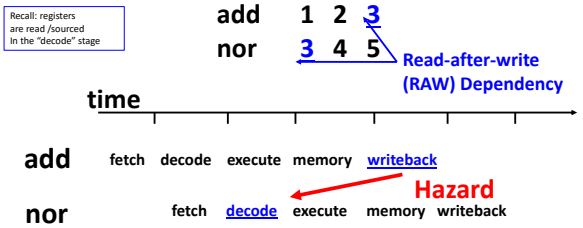
Review: Sample Code (Simple)

Let's run the following code on pipelined LC2K:

```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```

Data Hazards

Recall: registers are read /sourced In the "decode" stage



If not careful, nor will read a stale value of register 3

Three approaches to handling data hazards

- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

Problems with detect and stall

- CPI increases every time a hazard is detected!
- Is that necessary? Not always!
  - Re-route the result of the **add** to the **nor**
    - **nor** no longer needs to read R3 from reg file
    - It can get the data later (when it is ready)
    - This lets us complete the decode this cycle
    - But we need more control logic

Handling data hazards III: Detect and forward

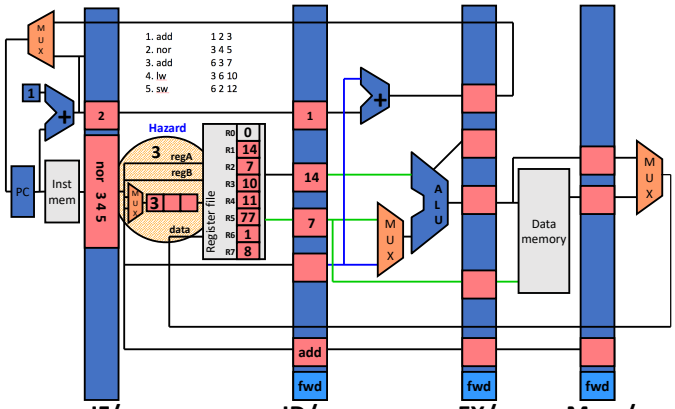
- Detect: same as detect and stall
  - Except that all 4 hazards have to be treated differently
    - i.e., you can't logical-OR the 4 hazard signals
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still be required even in the presence of forwarding

Forwarding example

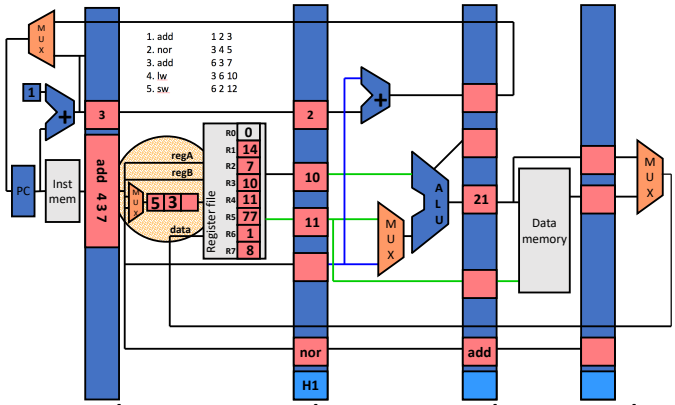
- We will use this program for the next example (same as last pipeline diagram example)

1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

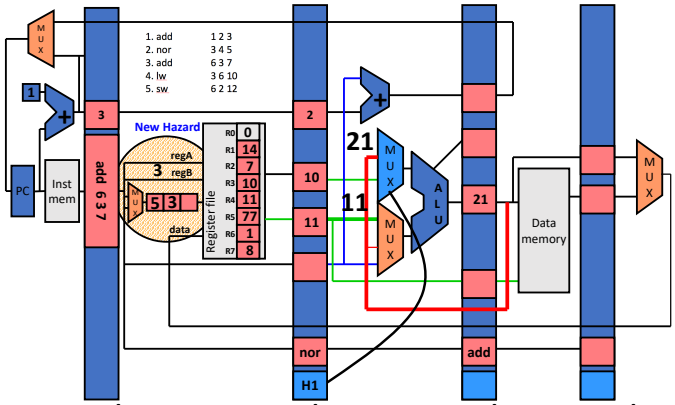
First half of cycle 3



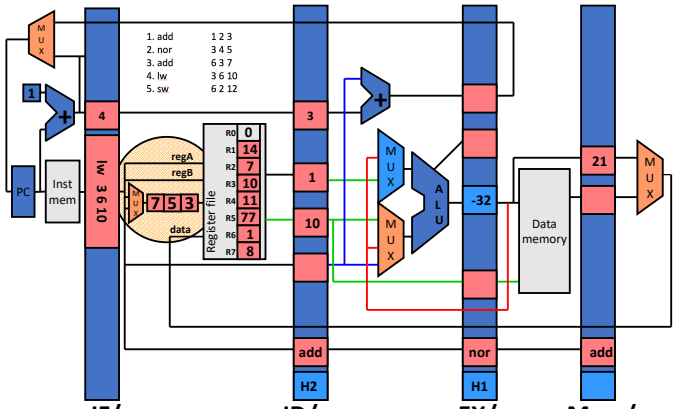
End of cycle 3



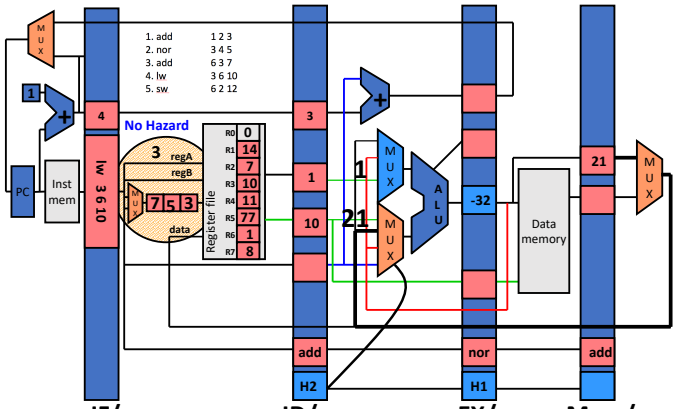
First half of cycle 4



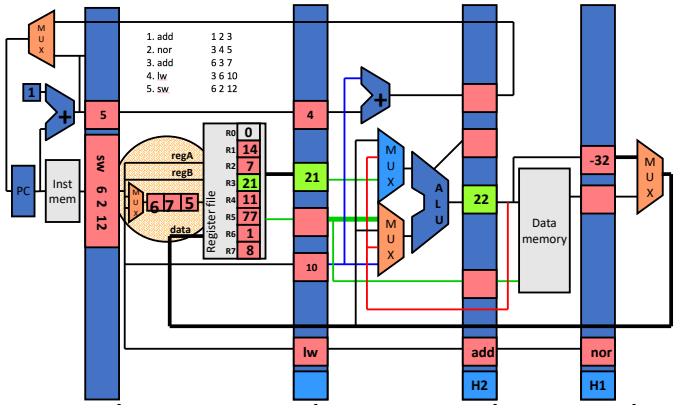
End of cycle 4



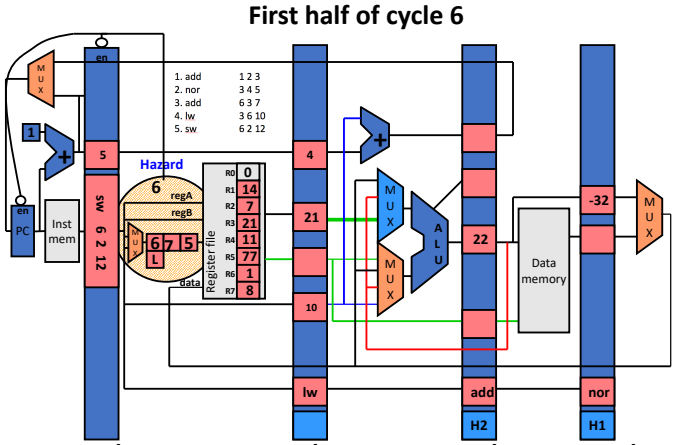
First half of cycle 5



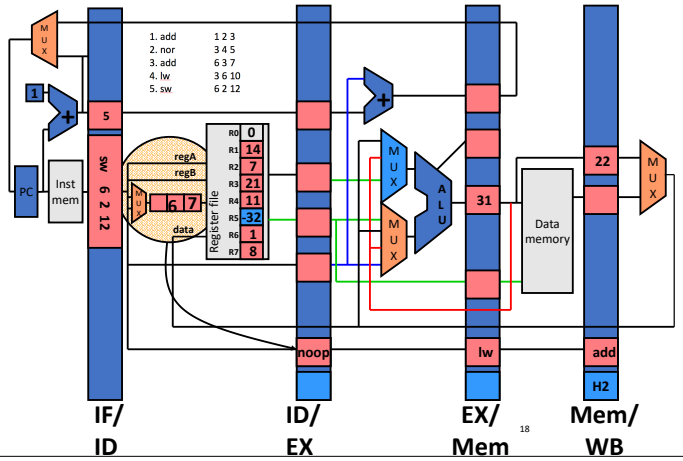
End of cycle 5



First half of cycle 6

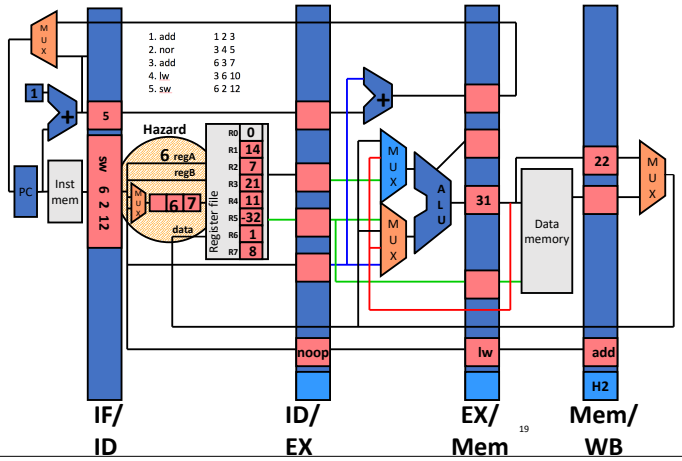


End of cycle 6



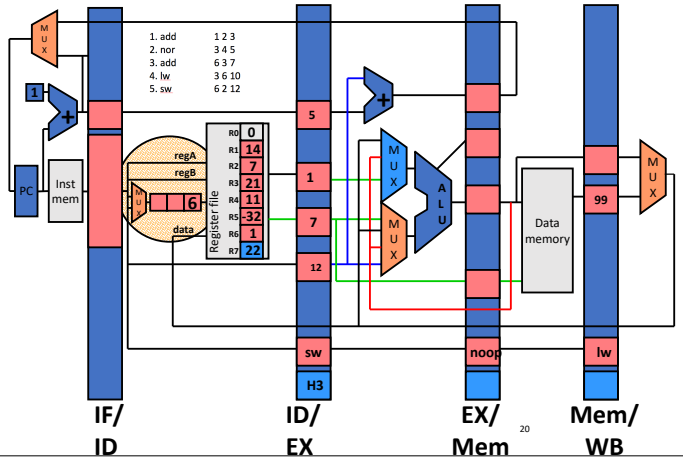
18

First half of cycle 7



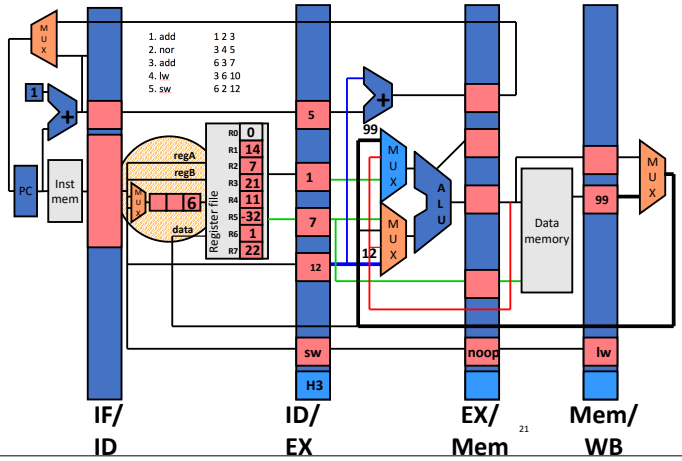
19

End of cycle 7



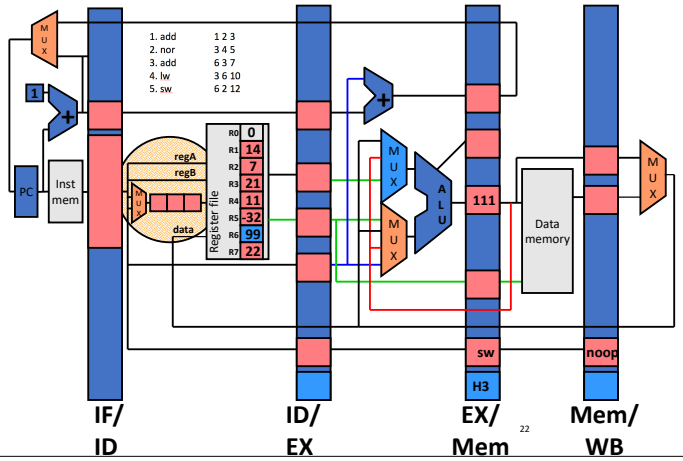
20

First half of cycle 8



21

End of cycle 8



22

Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID	EX	ME	WB							
add 6 3 7			IF	ID	EX	ME	WB						
lw 3 6 10				IF	ID	EX	ME	WB					
sw 6 2 12					IF	ID*	ID	EX	ME	WB			

24

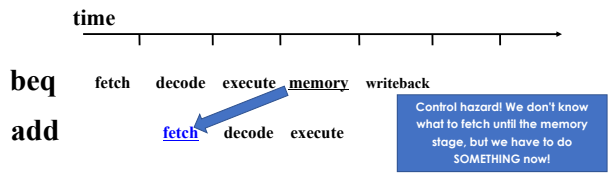
Other issues

- What other instruction(s) have we been ignoring so far??
- Branches!! (Let's not worry about jumps yet)
- Sequence for BEQ:
  - Fetch: read instruction from memory
  - Decode: read source operands from registers
  - Execute: calculate target address and test for equality
  - Memory: Send target to PC if test is equal
  - Writeback: nothing
  - Branch Outcomes
    - Not Taken
      - PC = PC + 1
    - Taken
      - PC = Branch Target Address

25

Control Hazards

```
beq  1  1  10
add  3  4  5
```



26

Approaches to handling control hazards

- 3 strategies – similar to handling data hazards

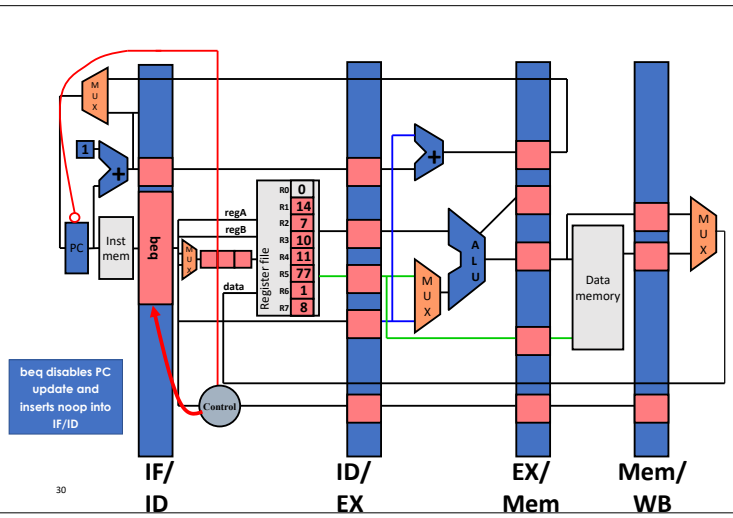
1. Avoid
   - Make sure there are no hazards in code
2. Detect and stall
   - Delay fetch until branch resolved
3. Speculate and squash-if-wrong
   - Guess outcome of branch
   - Fetch instructions assuming we're right
   - Stop them if they shouldn't have been executed

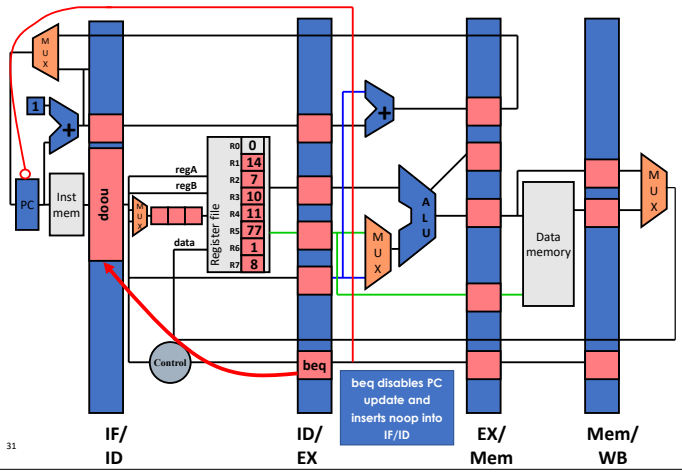
Avoiding Control Hazards

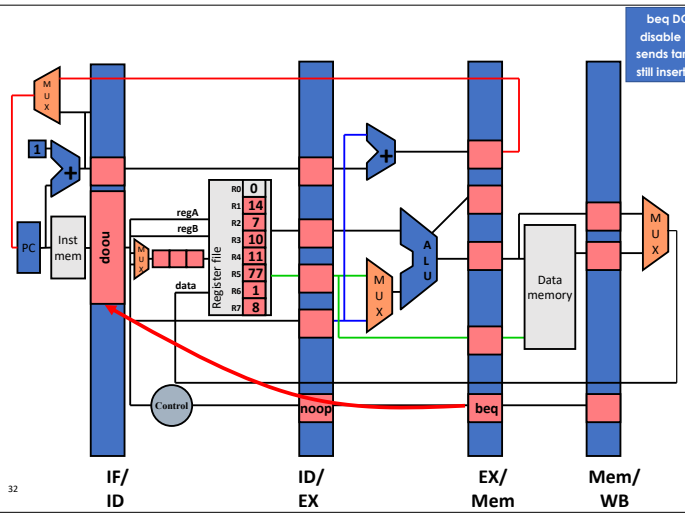
- Don't have branch instructions!
  - Possible, but not practical
  - ARM offers **predicated** instructions (instructions that throw away result if some condition is not met)
    - Allows replacement of if/else conditions
    - Hard to use for everything
    - Not covered more in this class

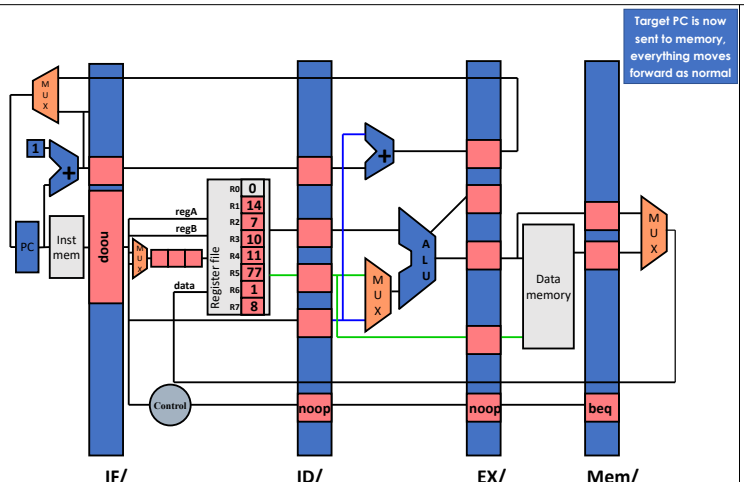
Detect and Stall

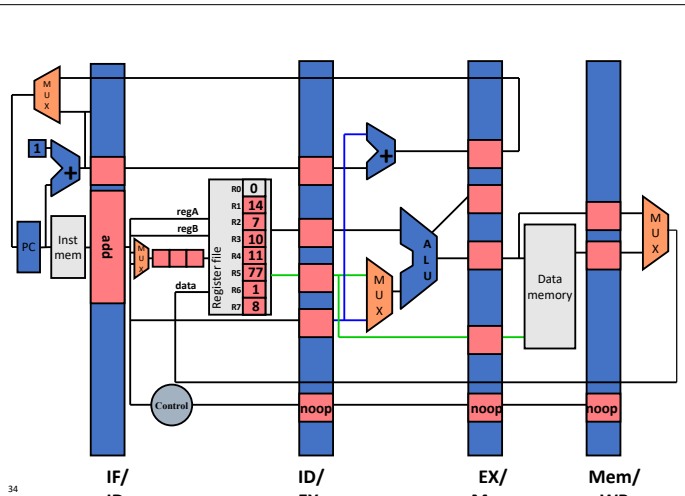
- Detection
  - Wait until decode
  - Check if opcode == beq or jalr
- Stall
  - Keep current instruction in fetch
  - Insert noops
  - Pass noop to decode stage, not execute!





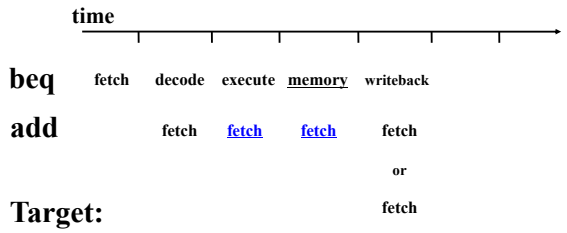






Control Hazards

```
beq   1  1  10
add   3  4  5
```



Problems with Detect and Stall

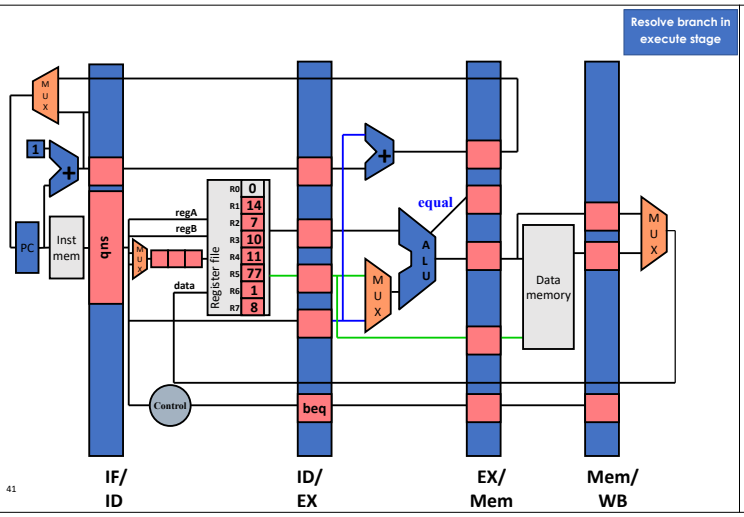
- CPI increases every time a branch is detected!
- Is that necessary? Not always!
  - Branch not always taken
  - Let's assume it is NOT taken…
    - In this case, we can ignore the beq (treat it like a noop)
    - Keep fetching PC + 1
  - What if we're wrong?
  - OK, as long as we do not COMPLETE any instruction we mistakenly execute
  - I.e. DON'T write values to register file or memory

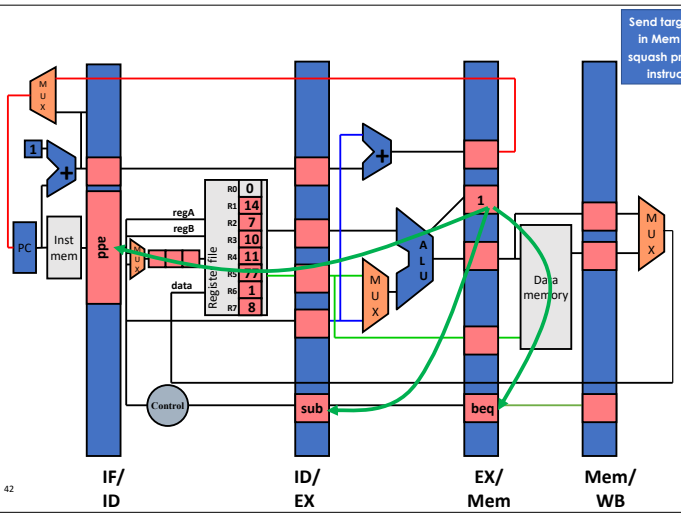
Agenda

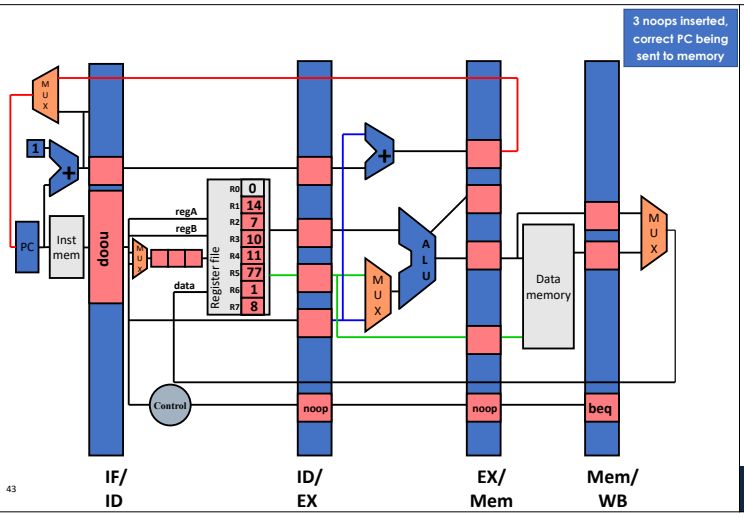
- Control Hazards and Basic Approaches
- Detect-and-Stall
- **Speculate-and-Squash**
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Speculate and Squash

- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory
  - Sent target address to PC







Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- **Exceptions**
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

What can go wrong?

- Data hazards: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if is about to be written.
- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- Exceptions: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Exceptions

- Exception: when something unexpected happens during program execution.
  - Example: divide by zero.
  - The situation is more complex than the hardware can handle
    - So the hardware branches to a function, an "exception handler" which is code to try to deal with the problem.
- The exact way to set up such an exception handler will vary by ISA.
  - With C on x86 you would use <signal.h> functions to handle the "SIGFPE" signal.
  - There is a pretty good Hackaday article on this if you want to learn more.

Exceptions and Pipelining

- The hardware branches to the "exception handler"
  - This means that any instruction which can "throw" an exception *could* be a branch.
  - Throwing an exception should be rare ("exceptional")
- So we would treat it much like a branch we predicted as "not taken"
  - Squash instructions behind it and then branch
  - It will introduce stalls, but since it should be rare, we don't worry about it.
    - "Make the common case fast".

Classic performance problem

- Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%

- Speculate "always not-taken" and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time per instruction if clock frequency is 100MHz?

CPI = 1 + 0.10 (loads) * 0.20 (load use stall)*1
+ 0.25 (branch) * 0.20 (miss rate)*3
CPI = 1 + 0.02 + 0.15 = 1.17
Time = 1.17 * 10ns =11.7ns per instruction

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*1
+ 0.25 (branch) * 0.20 (miss rate)*2
CPI = 1 + 0.02 + 0.1 = 1.12

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*4 + 0.25 (branch) * 0.20 (N stalls)*6
CPI = 1 + 0.08 + 0.30 = 1.38
Time = 1.38 * 5ns = 6.9 ns per instruction