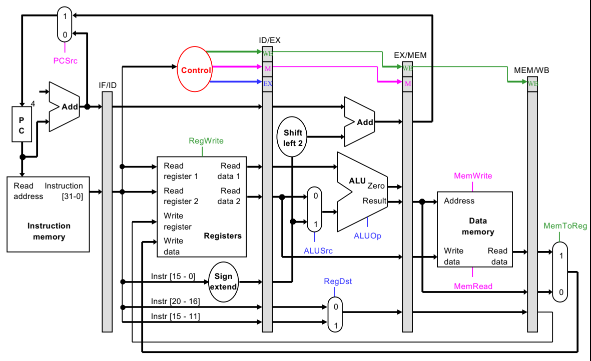
Lecture 13: Pipeline Hazards

(CPEG323: Intro. to Computer System Engineering)

Review

- Pipelining is a BIG idea
- Optimal Pipeline
  - Each instruction requires five stages, and five cycles, to complete.
  - Each stage uses different functional units of the datapath.
  - So we can execute up to five instructions in any clock cycle, with each instruction in a different stage and using different hardware.
- What makes this work well?
  - Similarities between instructions allow us to use same stages for all instructions (generally).
  - Each stage takes about the same amount of time as all others: little wasted time.

The pipelined datapath



Problems for Pipelining CPUs

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Structural hazards: HW cannot support some combination of instructions
  - Control hazards: Pipelining of branches causes later instruction fetches to wait for the result of the branch
- These might result in pipeline stalls or "bubbles" in the pipeline.

Data Hazards

Pipeline diagram review

	Clock cycle 1	2	3	4	5	6	7	8	9
lw $8, 4($29)	IF	ID	EX	MEM	WB				
sub $2, $4, $5		IF	ID	EX	MEM	WB			
and $9, $10, $11			IF	ID	EX	MEM	WB		
or $16, $17, $18				IF	ID	EX	MEM	WB	
add $13, $14, $0					IF	ID	EX	MEM	WB

- This diagram shows the execution of an ideal code fragment.
  - Each instruction needs a total of five cycles for execution.
  - One instruction begins on every clock cycle for the first five cycles.
  - One instruction completes on each cycle from that time on.

Our examples are too simple

- Here is the example instruction sequence used to illustrate pipelining on the previous page.

```
lw  $8, 4($29)
sub $2, $4, $5
and $9, $10, $11
or  $16, $17, $18
add $13, $14, $0
```

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our datapath handles this sequence easily, as we saw last time.
- But most sequences of instructions are *not* independent!

An example with dependencies

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

Data hazards in the pipeline diagram

Clock cycle

	1	2	3	4	5	6	7	8	9
sub $2, $1, $3	IF	ID	EX	MEM	WB				
and $12, $2, $5		IF	ID	EX	MEM	WB			
or $13, $6, $2			IF	ID	EX	MEM	WB		
add $14, $2, $2				IF	ID	EX	MEM	WB	
sw $15, 100($2)					IF	ID	EX	MEM	WB

- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

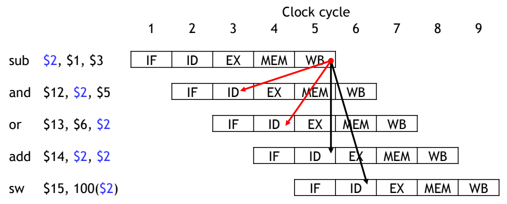
Things that are okay

Clock cycle

	1	2	3	4	5	6	7	8	9
sub $2, $1, $3	IF	ID	EX	MEM	WB				
and $12, $2, $5		IF	ID	EX	MEM	WB			
or $13, $6, $2			IF	ID	EX	MEM	WB		
add $14, $2, $2				IF	ID	EX	MEM	WB	
sw $15, 100($2)					IF	ID	EX	MEM	WB

- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $2 after the SUB finishes.

Dependency arrows


Clock cycle
1 2 3 4 5 6 7 8 9
sub $2, $1, $3 IF ID EX MEM WB
and $12, $2, $5 IF ID EX MEM WB
or $13, $6, $2 IF ID EX MEM WB
add $14, $2, $2 IF ID EX MEM WB
sw $15, 100($2) IF ID EX MEM WB

- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath.

Bypassing the register file

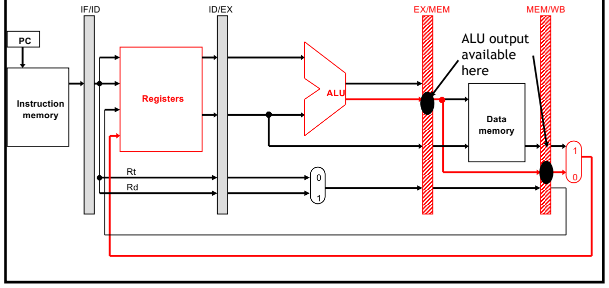
- The actual result $1 – $3 is computed in clock cycle 3, *before* it's needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.

Clock cycle

	1	2	3	4	5	6	7
sub $2, $1, $3	IF	ID	EX	MEM	WB		
and $12, $2, $5		IF	ID	EX	MEM	WB	
or $13, $6, $2			IF	ID	EX	MEM	WB

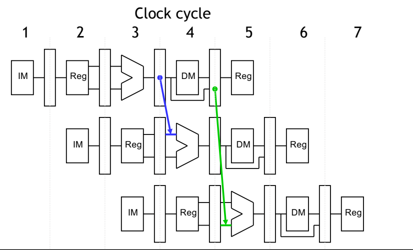
Pipeline Registers to the rescue!

- Pipeline stages communicate through pipeline registers:
  IF/ID ID/EX EX/MEM MEM/WB
- We “forward” data from pipeline registers to later instructions



Forwarding

- The actual result $1 - $3 is computed in clock cycle 3, before it is needed in cycles 4 and 5
- We forward that value to later instructions, to prevent data hazards.
  - In clock cycle 4, AND gets the value $1 − $3 from the EX/MEM
  - In cycle 5, OR gets that same result from the MEM/WB



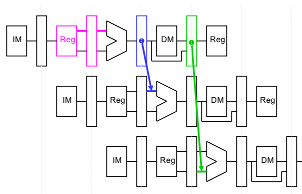
sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

Outline of forwarding hardware

- A forwarding unit selects the correct ALU inputs for the EX stage.
  - No hazard: ALU’s operands comes from the register file, like normal.
  - Data hazard: operands come from either the EX/MEM or MEM/WB pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.

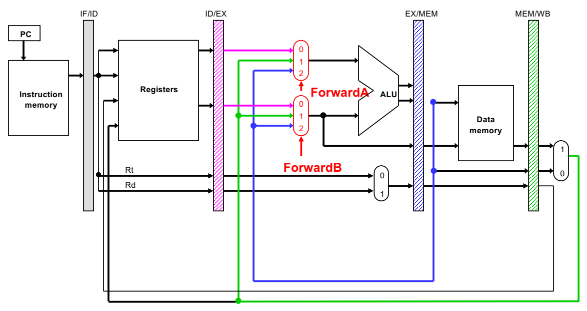


sub $2, $1, $3

and $12, $2, $5

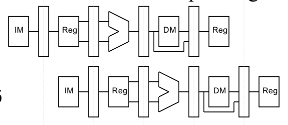
or $13, $6, $2

Simplified datapath with forwarding muxes



Detecting EX/MEM Hazards

- In which stage cycle can we detect an impending hazard?



sub $2, $1, $3

or $12, $2, $5

- Answer: cycle 3, when sub is in EX, or is in ID
  - Hazard because: ID/EX.rd == IF/ID.rs
- An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, *and*
  2. The destination is one of the ALU source registers in the EX stage

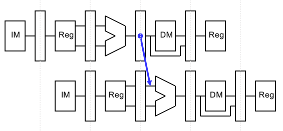
EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.

```
if (EX/MEM.RegWrite = 1
   and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2
```

- The second ALU source is similar.

```
if (EX/MEM.RegWrite = 1
   and EX/MEM.RegisterRd = ID/EX.RegisterRt)
then ForwardB = 2
```



sub $2, $1, $3

and $12, $2, $5

Detecting MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

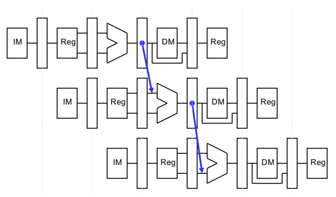
```
add  $1, $2, $3
add  $1, $1, $4
sub  $5, $5, $1
```

- Register $1 is written by *both* of the previous instructions; from which instruction should it receive its value?

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



Detecting MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

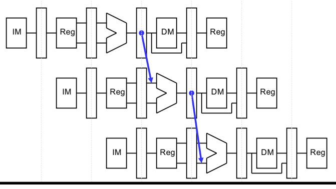
```
add  $1, $2, $3
add  $1, $1, $4
sub  $5, $5, $1
```

- Register $1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



MEM/WB hazard equations

- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

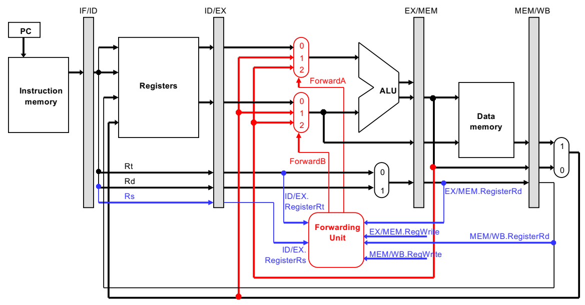
```
if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRs
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
then ForwardA = 1
```

- The second ALU operand is handled similarly.

```
if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRt
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
then ForwardB = 1
```

- Handled by a forwarding unit which uses the control signals stored in pipeline registers to set the values of ForwardA and ForwardB

Simplified datapath with forwarding



The forwarding unit

- The forwarding unit has several control signals as inputs.

ID/EX.RegisterRs	EX/MEM.RegisterRd	MEM/WB.RegisterRd
ID/EX.RegisterRt	EX/MEM.RegWrite	MEM/WB.RegWrite

(The two RegWrite signals are not shown in the diagram, but they come from the control unit.)

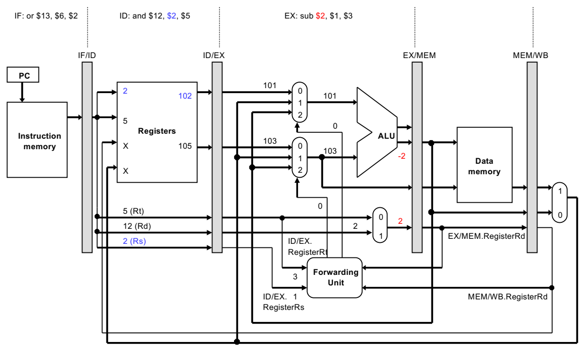
- The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

Example

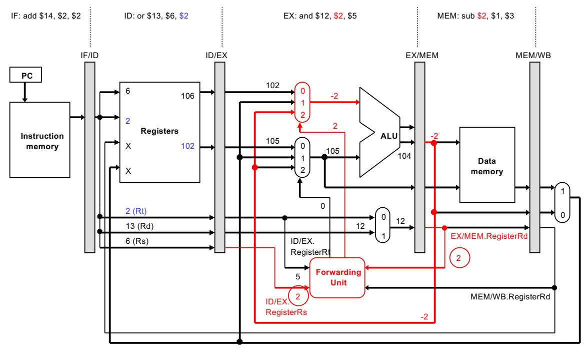
```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain -2 (101 - 103).
  - The other instructions should all use -2 as one of their operands.
- We' ll try to keep the example short.
  - Assume no forwarding is needed except for register $2.
  - We' ll skip the first two cycles, since they' re the same as before.

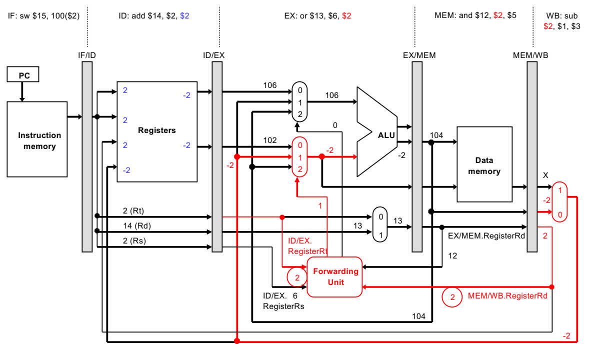
Clock cycle 3



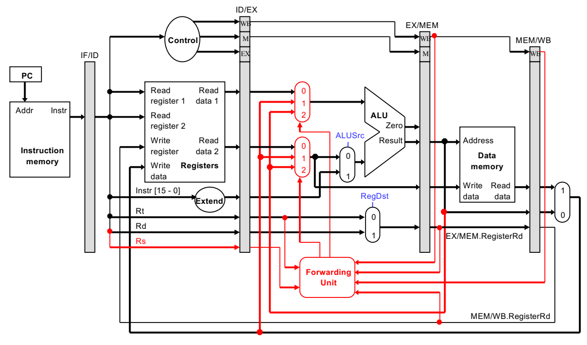
Clock cycle 4: forwarding $2 from EX/MEM



Clock cycle 5: forwarding $2 from MEM/WB

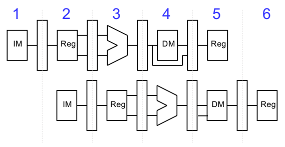


Complete pipelined datapath...so far



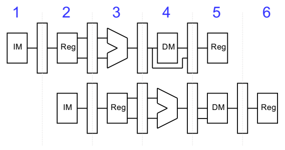
What about stores?

- Two "easy" cases:



add $1, $2, $3

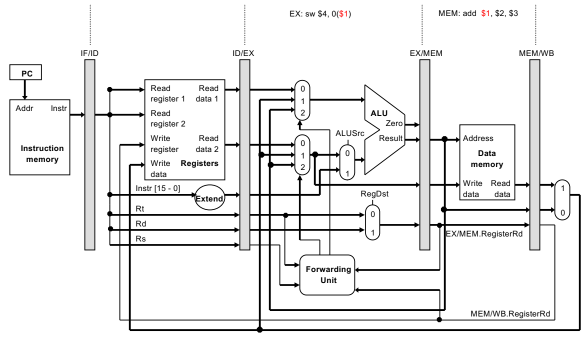
sw $4, 0($1)



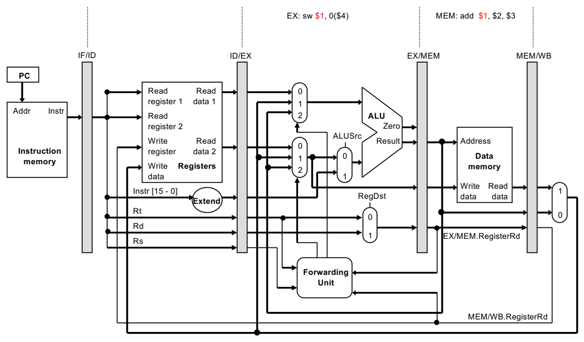
add $1, $2, $3

sw $1, 0($4)

Store Bypassing: Version 1

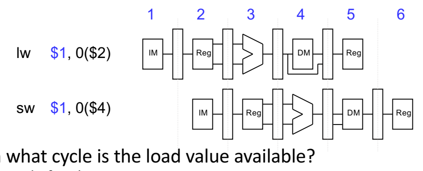


Store Bypassing: Version 2



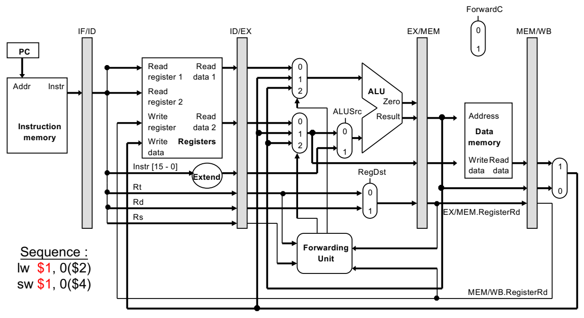
What about stores?

- A harder case:



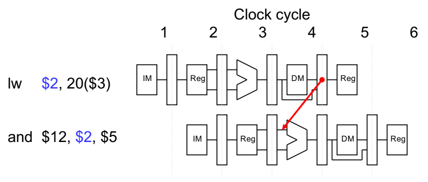
- In what cycle is the load value available?
  - End of cycle 4
- In what cycle is the store value needed?
  - Start of cycle 5
- What do we have to add to the datapath?

Load/Store Bypassing: Extend the Datapath



What about loads?

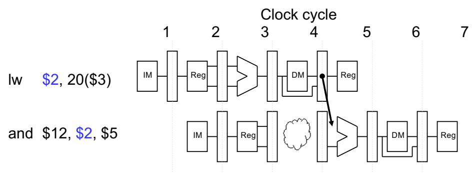
- Consider the instruction sequence shown below:
  - The load data doesn't come from memory until the *end* of cycle 4.
  - But the AND needs that value at the *beginning* of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.



- We call this a load-use hazard.

Stalling

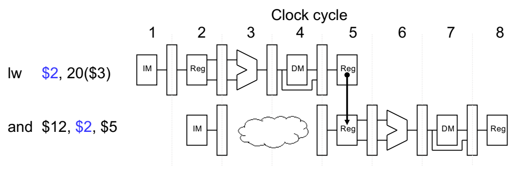
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

Stalling and forwarding

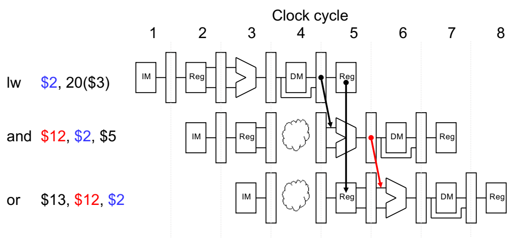
- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.

Stalling delays the entire pipeline

- If we delay the second instruction, we' ll have to delay the third one too.
  - This is necessary to make forwarding work between AND and OR.
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle.



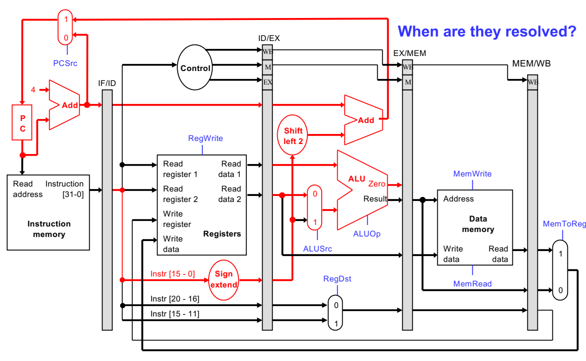
Summary

- In real code, most instructions are dependent upon other ones.
  - This can lead to data hazards in our original pipelined datapath.
  - Instructions can' t write back to the register file soon enough for the next two instructions to read.
- Forwarding eliminates data hazards involving arithmetic instructions.
  - The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction.
  - Hazards are avoided by grabbing results from the pipeline registers *before* they are written back to the register file.
- Stalling
  - Forwarding can' t save us in some cases involving lw.

Structural Hazards

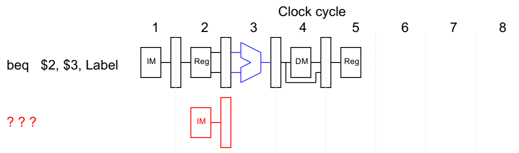
Control Hazards

Branches in the original pipelined datapath



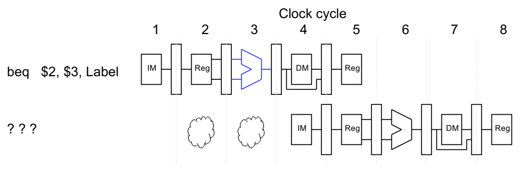
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what' s called a control hazard.



Stalling is one solution

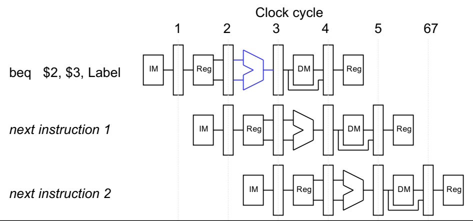
- Again, stalling is always one possible solution.



- Here we just stall until cycle 4, after we do make the branch decision.

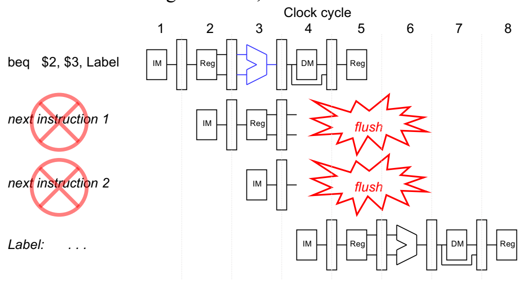
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.



Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for *every* branch.
- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.

Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.
- Number of cycles for N instructions on a k stage pipeline:

  $N + k - 1 + s + f$ where s = #stalls, f = #flushes

Reading

- **5th Edition: 4.7,4.8**

61