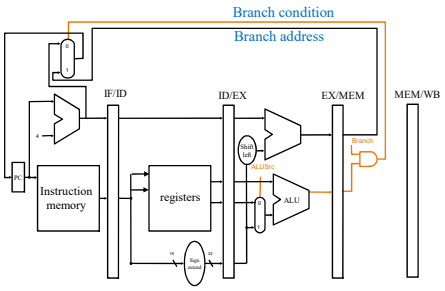
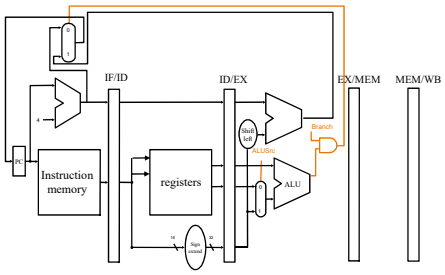
Control Hazards

Where are branch conditions and target addresses resolved (when is the PC overwritten with the branch target address)?





If resolved when the branch instruction is in MEM stage:

- the 3 instructions following the branch are already in the pipeline

If resolved when the branch instruction is in EX stage:

- the 2 instructions following the branch are already in the pipeline
- How does this affect the cycle time?

Control Hazards

Assume that "branches" are resolved in the EX stage. Hence, when a branch decision is made two instructions are already in the pipe (started execution).

Example: consider the execution of the following code segment:

```
add $4, $5, $6
beq $1, $2, 10
lw $3, 300($0)
sub $7, $8, $9
sw $10, 4($8)
o
o
o
and $3, $2, $1
```

10 instructions

	IF stage	ID stage	EX stage	MEM stage	WB stage
Cycle 1	add $4, $5, $6				
Cycle 2	beq $1, $2, 10	add $4, $5, $6			
Cycle 3	lw $3, 300($0)	beq $1, $2, 10	add $4, $5, $6		
Cycle 4	sub $7, $8, $9	lw $3, 300($0)	beq $1, $2, 10	add $4, $5, $6	
Cycle 5	sw $10, 4($8) or and $3, $2, $1	**sub $7, $8, $9**	**lw $3, 300($0)**	beq $1, $2, 10	add $4, $5, $6

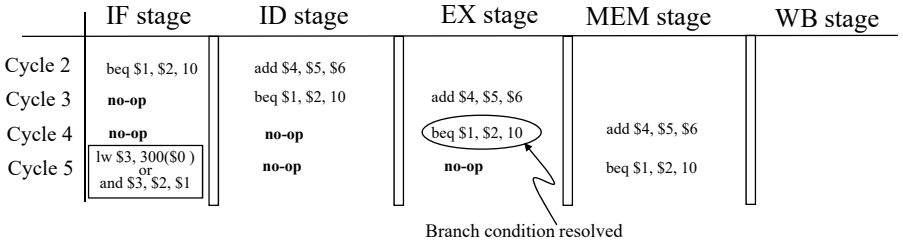
Branch condition resolved

What is wrong and what can be done?

Adding no-ops (a software solution)

Make the compiler add no-ops after the branch instruction.
- Why is that not ideal??

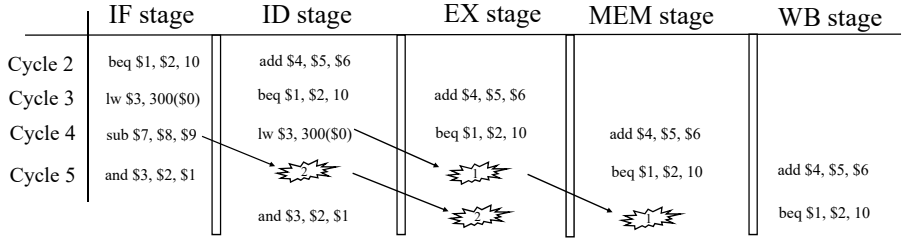
```
add $4, $5, $6
beq $1, $2, 10
no-op
no-op
lw $3, 300($0)
sub $7, $8, $9
sw $10, 4($8)
°
°
°
and $3, $2, $1
```

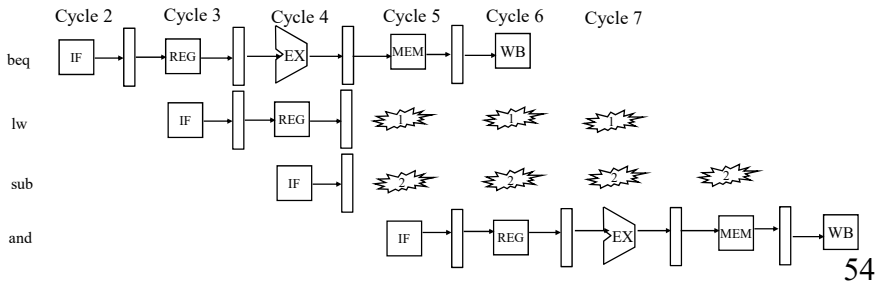


The branch condition will be resolved in cycle 4 and the correct instruction will enter the pipe in cycle 5

A hardware solution

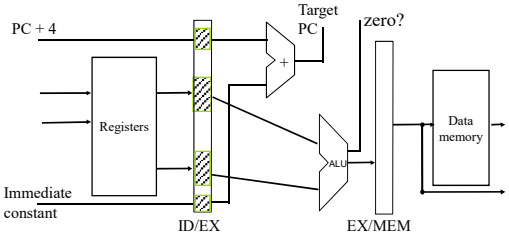
Introduce a bubble (a no-op introduced by the hardware) to abort unwanted instructions.





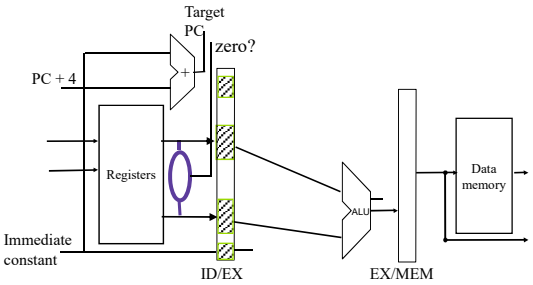
Reducing the number of aborted instructions

Branch condition and address resolved in the EX stage

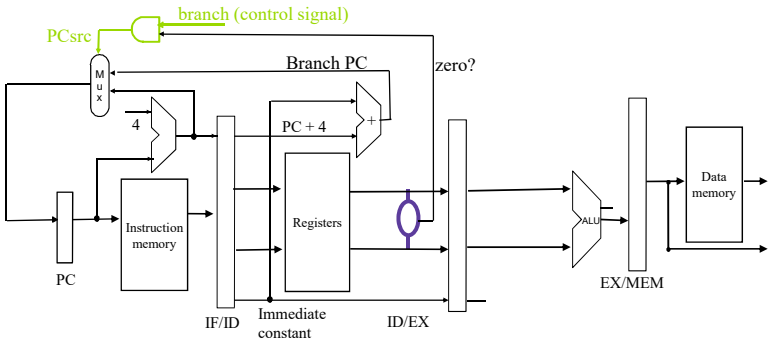


Branch address can be resolved in ID stage.
Branch condition can also be resolved in the ID stage if we use a comparator.

- why would this help?
- Does this have any effect on the cycle time?



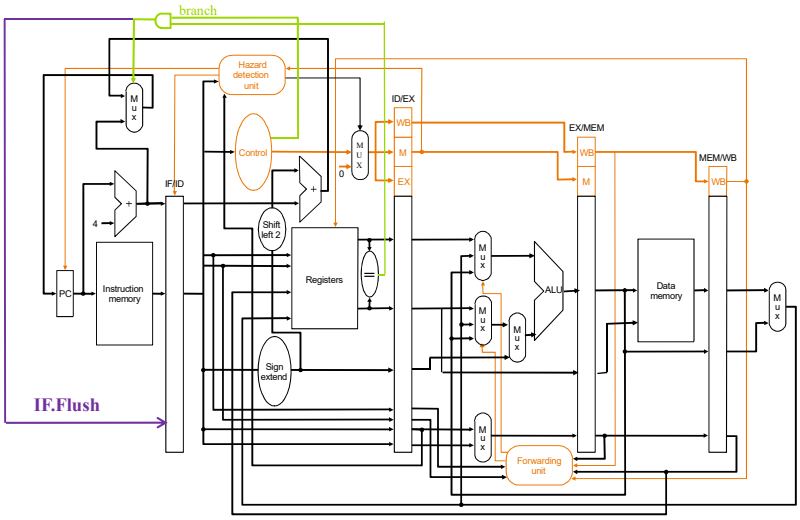
Resolving the branches



The mux at the input of PC selects the branch PC when
- the control indicates that the instruction in the ID stage is a *beq*
- the zero output of the comparator is true

Insert a no-op to the IF/ID buffer whenever the mux selects the branch PC (PCsrc=1).

Flushing Instructions (creating a bubble)



The effect of control hazard on throughput

- Assume that when the branch is resolved, K instructions following the branch are already in the pipeline.
- If control hazards are dynamically resolved, then each taken branch introduces K bubbles in the pipeline
- Hence, the average number of clock cycles to execute an instruction is:

$$CPI = CPI_{nch} + \alpha * \pi * K$$

where

CPI_{nch} is the CPI with no control hazard
α is the fraction of branch instructions in the instruction mix
π is the probability a branch is actually taken

Example: if branches are dynamically resolved in the EX stage, 10% of the instructions are branches and the probability that a branch is taken is 40%, then

$CPI = 1 + 2 * 0.1 * 0.4 = 1.08$ cycles per instruction (assuming $CPI_{nch} = 1$)

Hence, **the average execution time** of an instruction is 1.08 * clock cycle time

- For the software solution, where the compiler adds K no-ops after each branch,

$$CPI = CPI_{nch} + \alpha * K$$

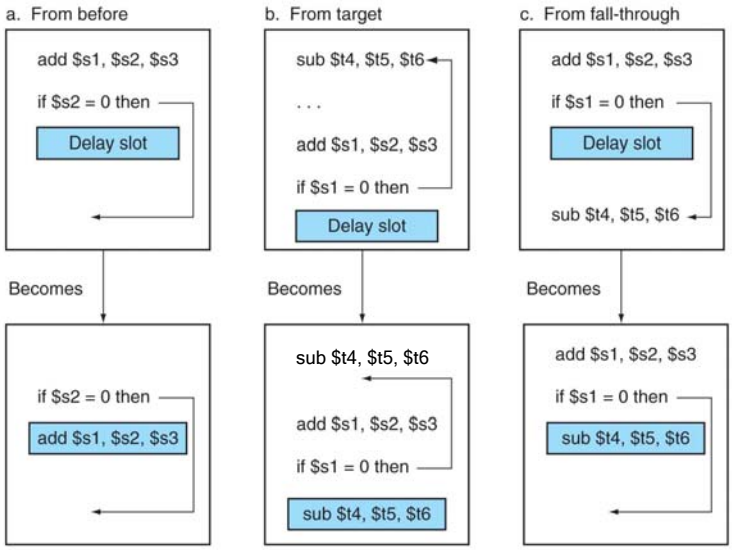
Pipeline depth vs. branch penalty

- Today's processors employ a deep pipeline (possibly more than 20 stages!) to increase the clock rate
  - Many stages means smaller amount of work per stage ⇒ shorter time needed per stage ⇒ higher clock rate!
- But what about branch penalty?
  - Penalty depends on the pipeline length!
  - Branches represent 15~20% of all instructions executed
- Situation is compounded by the increased issue bandwidth (will discuss when we talk about superscalar processors)

Delayed branching

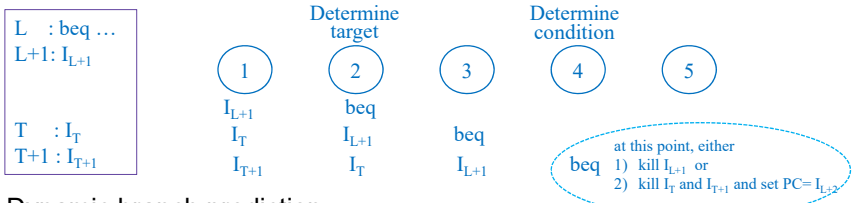
- Another approach to avoid control hazards
- Change the branch semantics such that the "N" instructions after a branch are always executed before branching takes place.
  - The N instructions are executed regardless of the branch outcome
  - N is set to be the number of cycles needed to resolve the branch
- The compiler will try to fill the N slots with useful work
  - If can't find instructions to fill the slots, use NOPs.
- Simplifies hardware (eliminates hazards by changing the semantics)
- Possibly good performance – if slots are filled with useful instructions
- Code size will increase??

Finding instructions to fill the delay slots



Different branch handling strategies

- Stall until branch direction is know (compiler can add no-ops)
- Delayed branches
- Assume branch is “NOT TAKEN” and take corrective action if wrong
  - Execute fall-off instructions that follow the branch (at PC+4, PC+8, …)
  - (PC+4) is computed every cycle, so use it to get the fall-off instructions
  - Squash (or cancel) instructions in pipeline if branch is actually taken
- Assume branch is “TAKEN” and take corrective action
  - Motivated by the observation that 67% of branches are taken, on average
  - Start fetching from the branch target as soon as it is available
  - useful if target address is computed earlier than the branch condition.



- Dynamic branch prediction