CS2200
Systems and Networks
Spring 2023

Lecture 12: Pipeline Hazards, pt 2

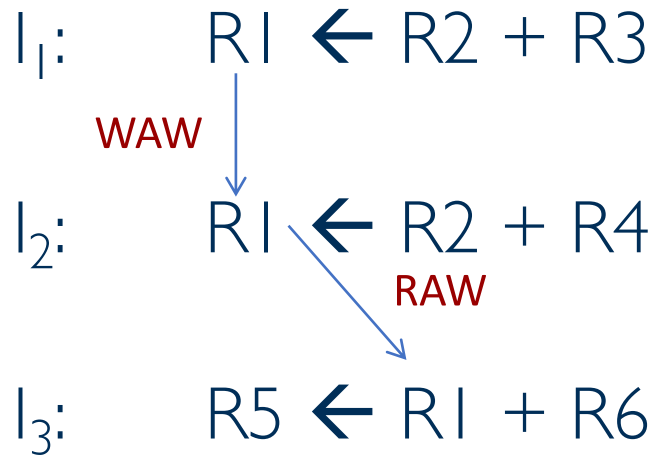
Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

Pipeline Hazards

- Structural hazards
  - Datapath/resource limitations
- Data hazards
  - Program limitations
- Control hazards
  - Program limitations

What hazards?



Register file

	B	**RP**
	B	**RP**
	B	**RP**
	B	**RP**
	B	**RP**
	B	**RP**
	B	**RP**
	B	**RP**

Must ensure that I_3 gets the correct value of R1 (from I_2 instead of I_1)

- Without forwarding, need to make sure I_1 does not clear the busy bit – I_2 should do that
  - i.e., I3 *cannot* move past ID/RR before I2 is also done with WB
- With forwarding, need to make sure R1 value is forwarded from EX stage (where I_2 is), not from MEM (where I_1 is)

Control hazard

	IF*	ID/RR	EX	MEM	WB
1.BEQ					
2.ADD					
3.NAND					
4.LW					

* We do not actually know what the instruction is in the Fetch Stage before we decode it

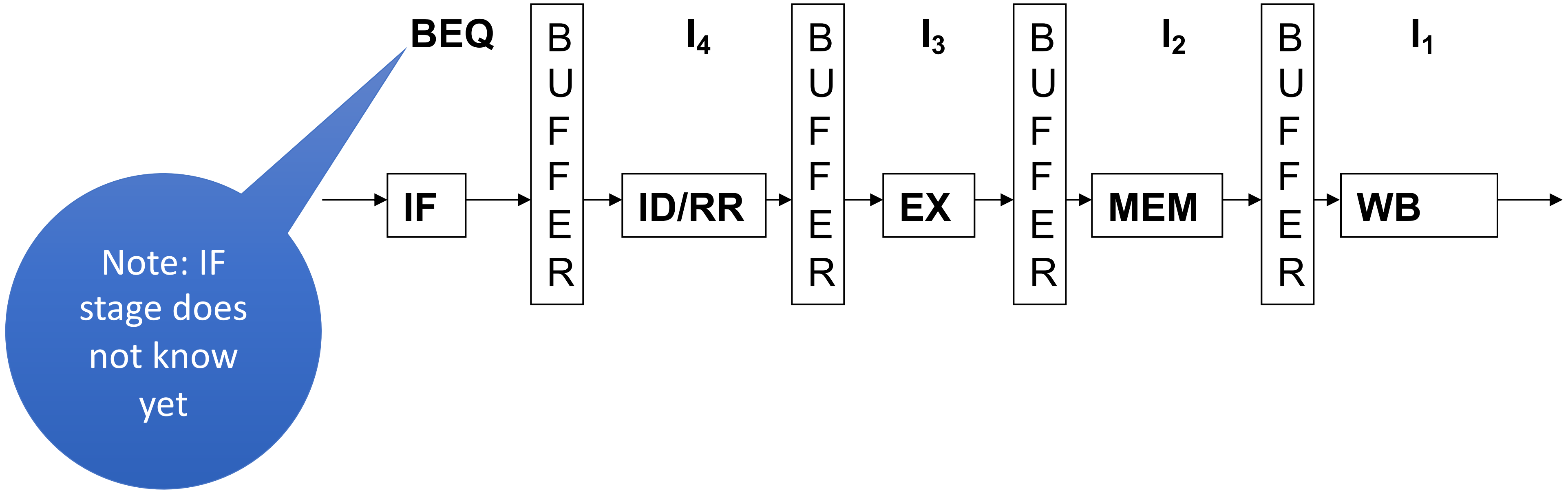
Is there any analog in the sandwich pipeline?

Conservative handling of branches

- Wait until branch outcome is known
  - Stall the pipeline ➔ send NOPs from IF state
- Resume normal execution when branch outcome is known
  - Result of comparison (A-B) is known
  - If there is need to branch, PC gets the target address of the branch
- Two cases
  - Branch TAKEN (i.e. A==B)
  - Branch NOT TAKEN (i.e. A!=B)

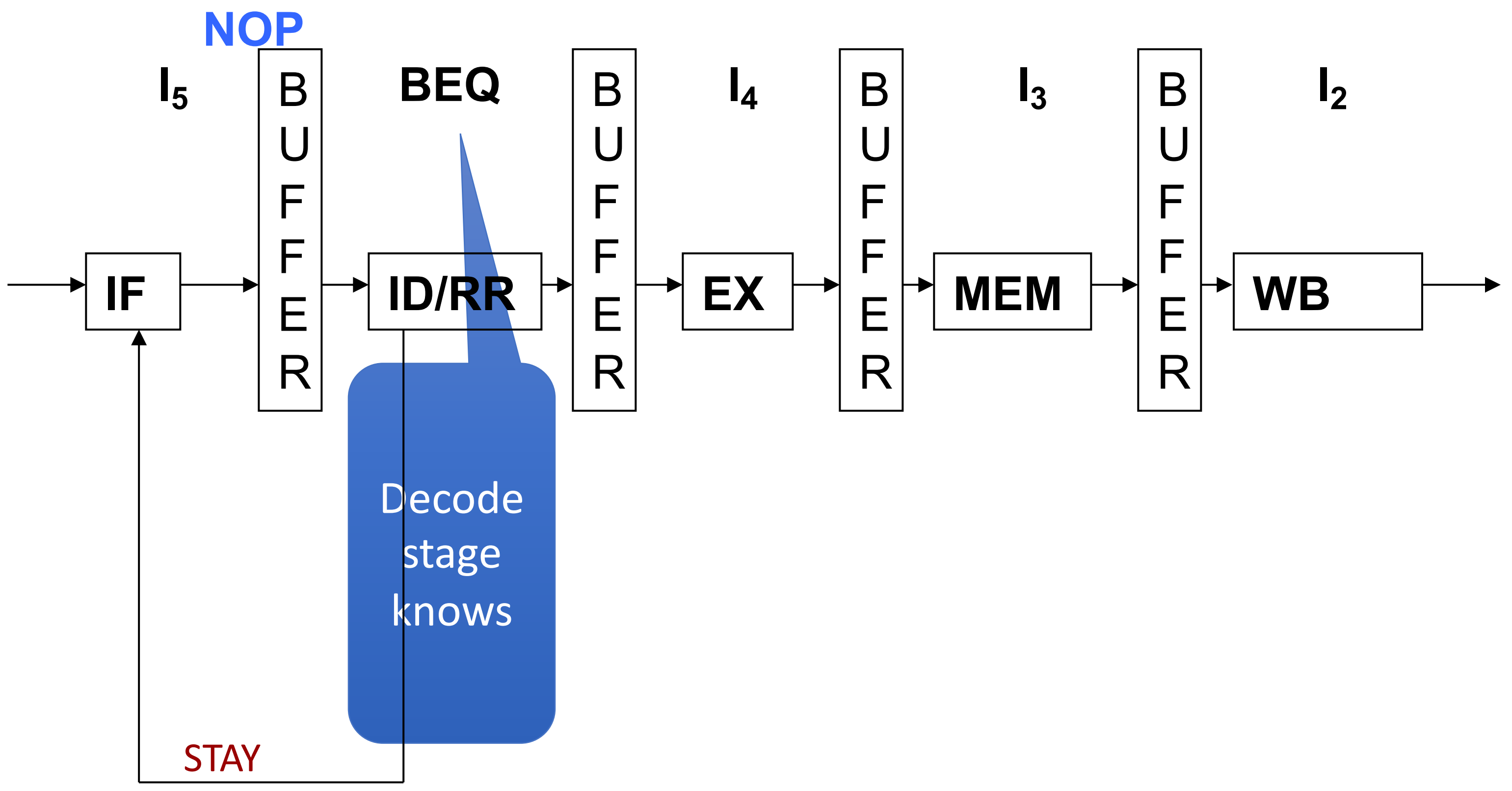
Conservative branch handling

Cycle 1



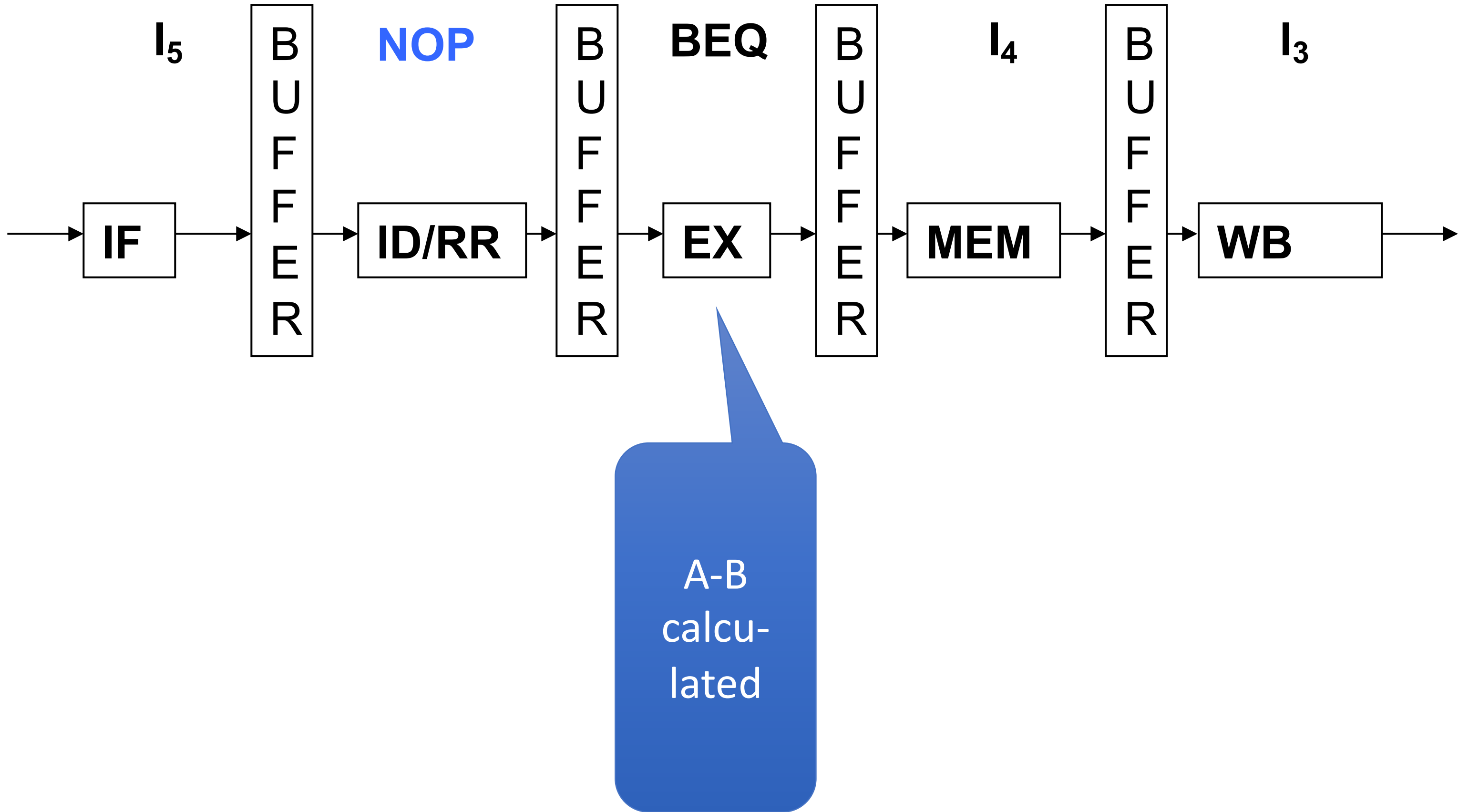
Branch handling

Cycle 2



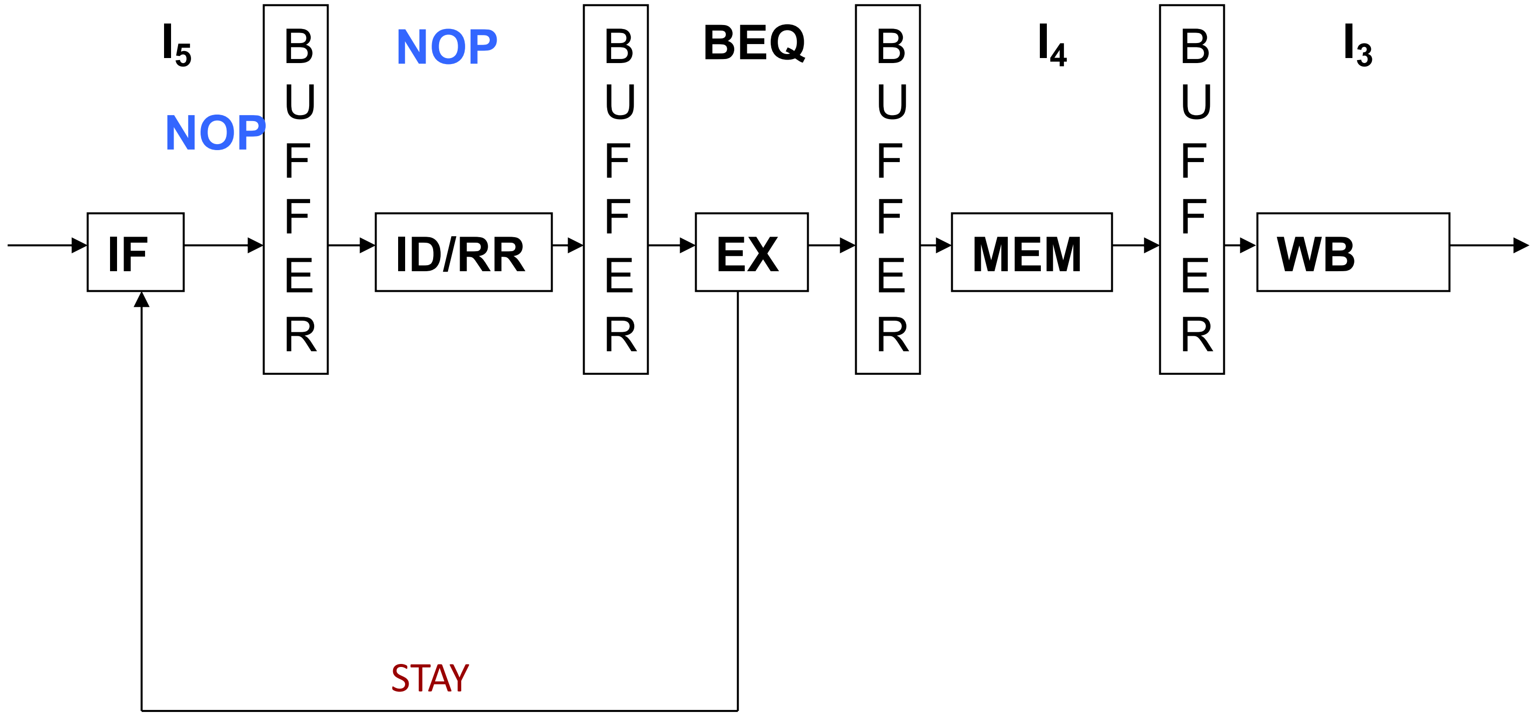
Branch handling

Cycle 3



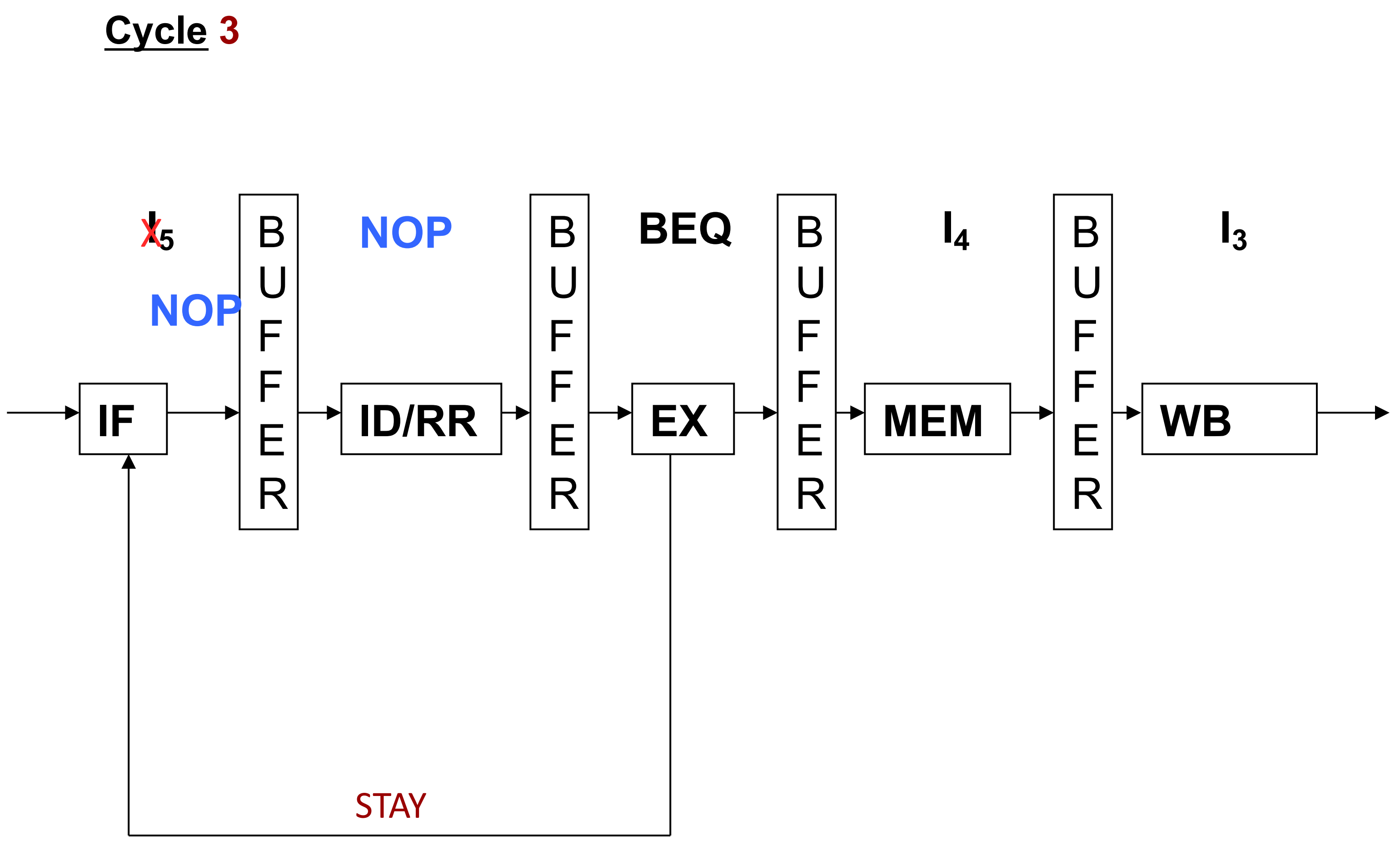
Branch handling (taken)

Cycle 3



A-B is calculated as zero and branch is going to be taken
PC+offset is being calculated

Branch handling (taken)

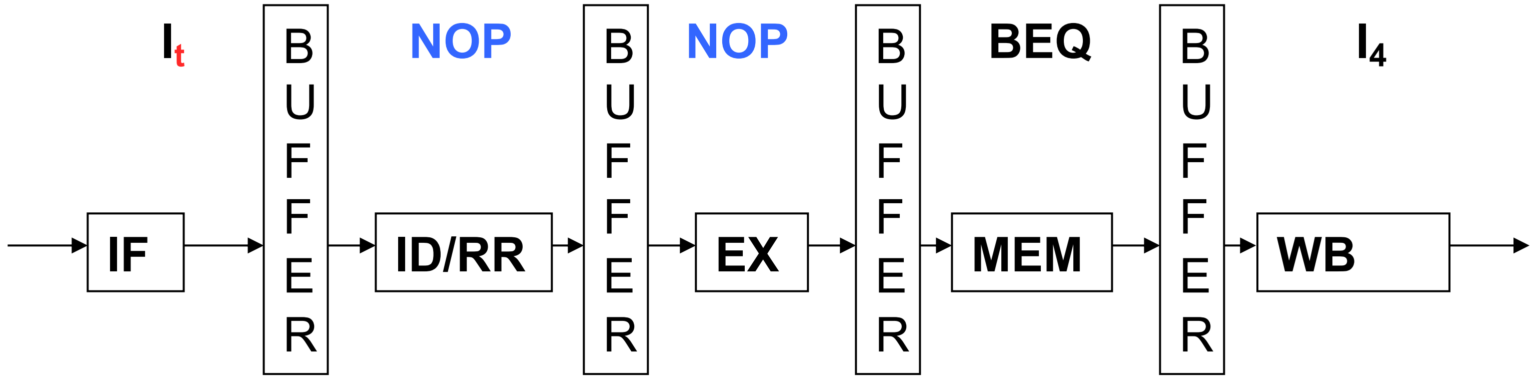


A-B is calculated as zero and branch is going to be taken

PC+offset is being calculated ➔ value is clocked into PC at end of EX

Branch handling (taken)

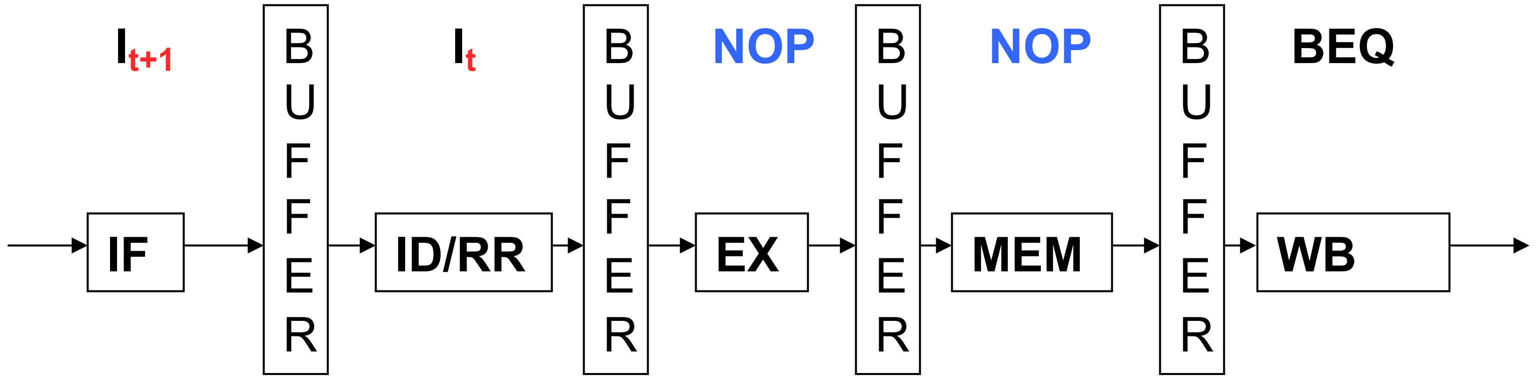
Cycle 4



We used updated PC to fetch the target of branch

Branch handling (taken)

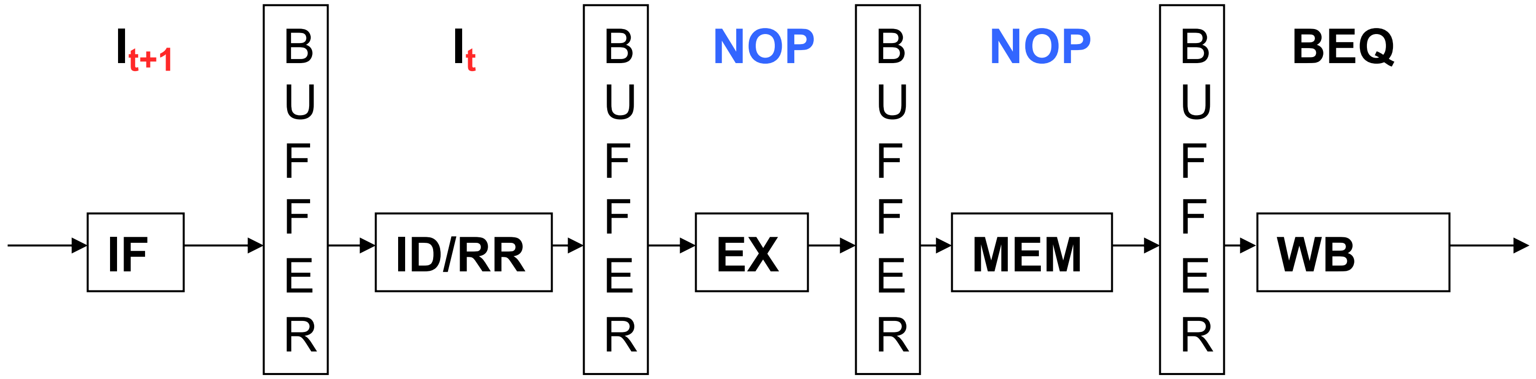
Cycle 5



Normal pipeline operation resumed

Branch handling (taken)

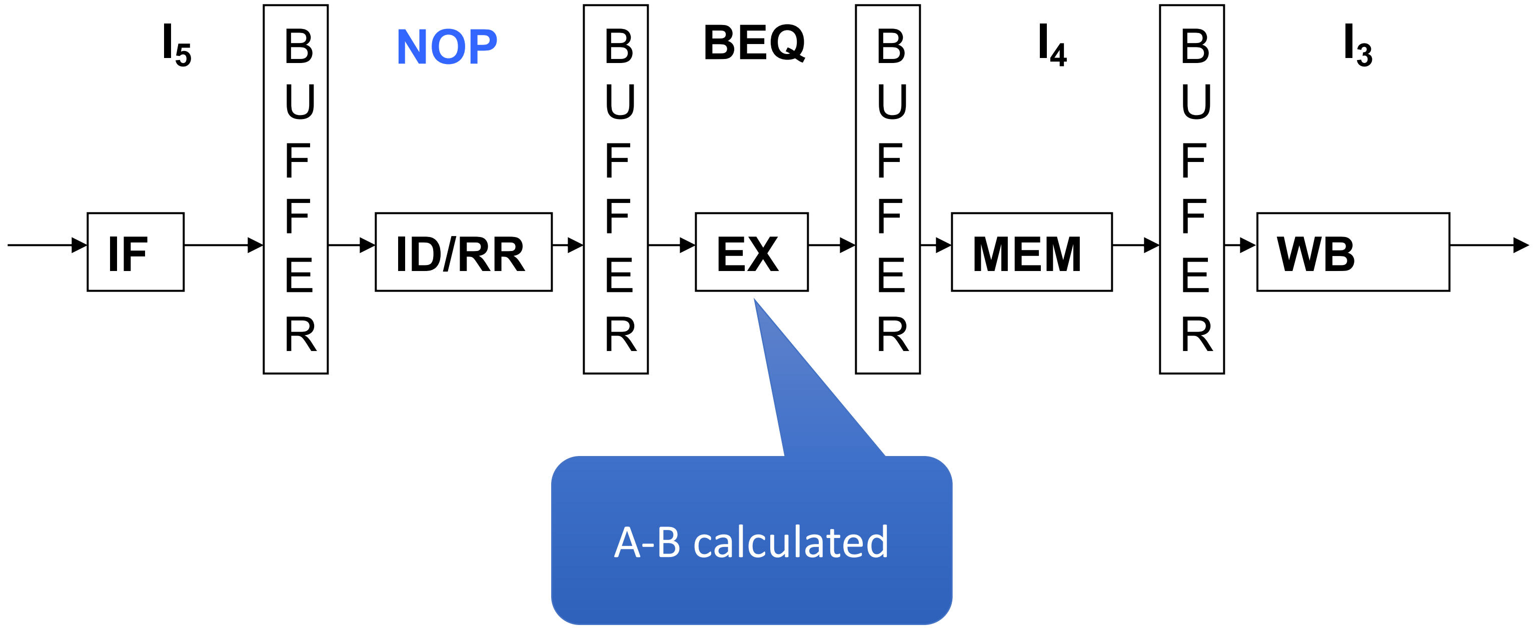
Cycle 5



Outcome: Two pipeline bubbles

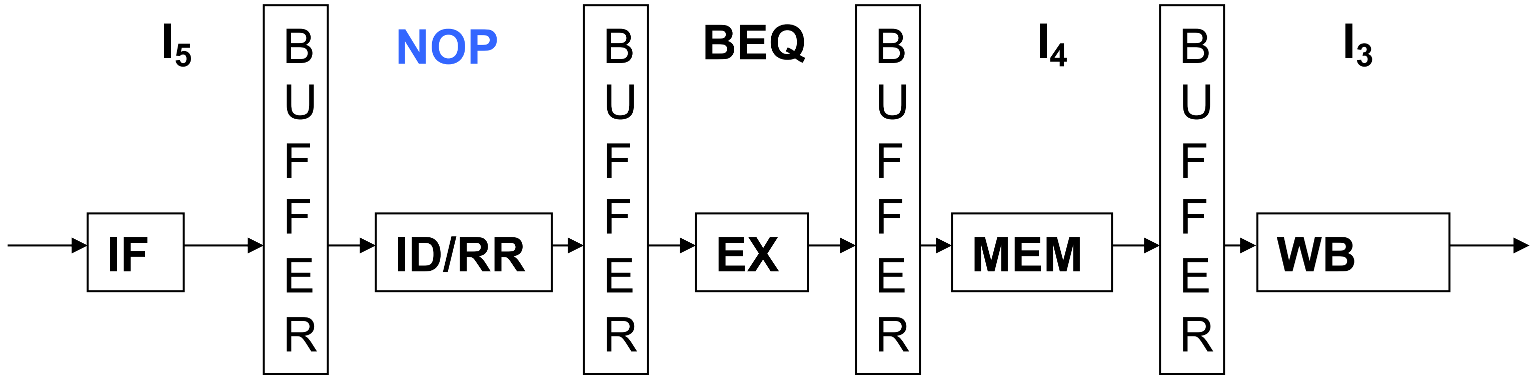
Branch handling

Cycle 3



Branch handling (NOT taken)

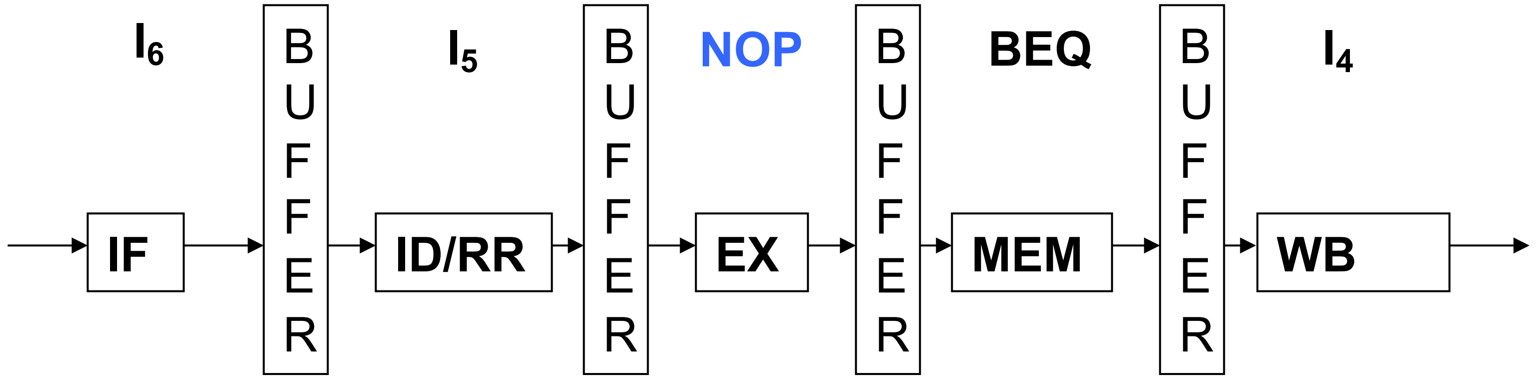
Cycle 3



A-B calculated as non-zero; branch not taken
PC is already pointing to the sequential path

Branch handling (NOT taken)

Cycle 4

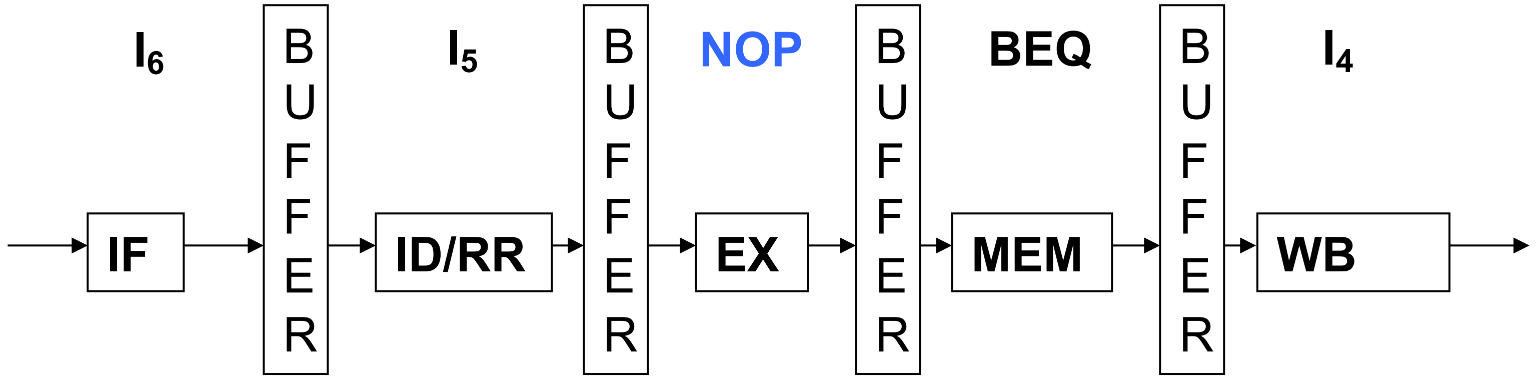


A-B calculated as non-zero; branch not taken
Normal pipeline operation resumed

Incoming

Branch handling (NOT taken)

<u>Cycle</u> 4



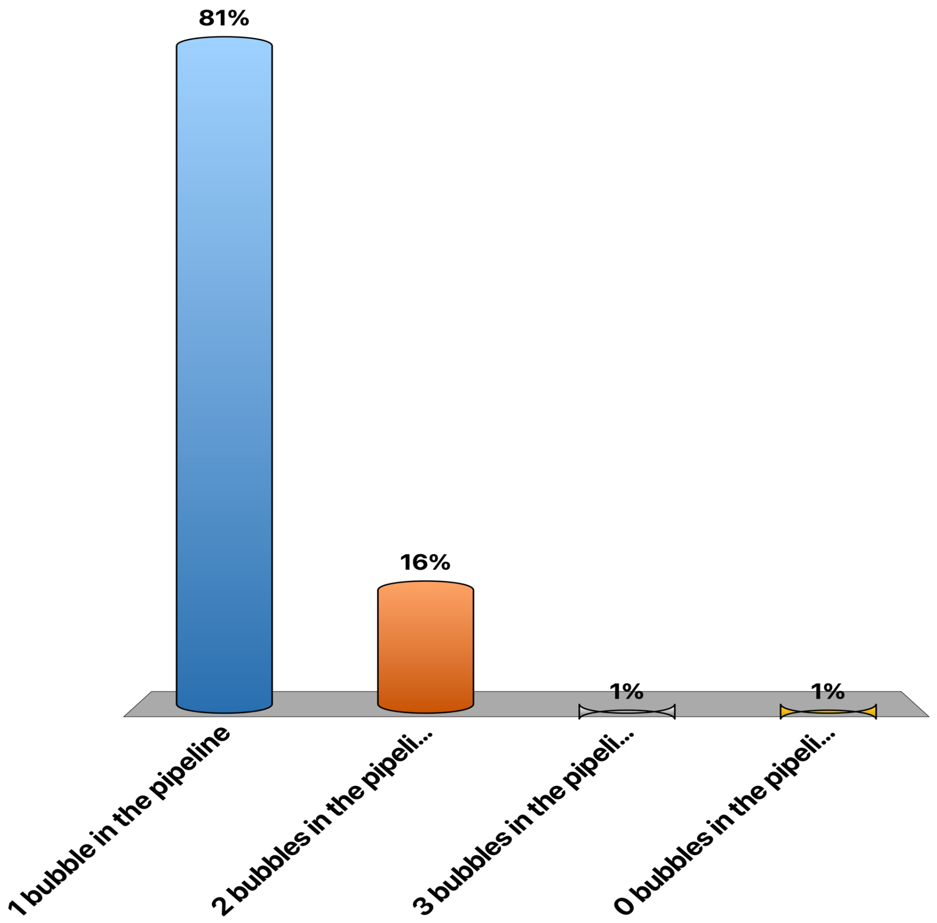
Outcome: one pipeline bubble

Session ID: dag2200

Question

Conservative handling of branches introduces at least…

A. 1 bubble in the pipeline
B. 2 bubbles in the pipeline
C. 3 bubbles in the pipeline
D. 0 bubbles in the pipeline



Code for our examples

Assume the following sequence of instructions:

```
          BEQ       L1
          ADD
          LW
          ….
L1        NAND
          SW
```

The instruction executed after BEQ will be ADD or NAND, depending on BEQ’s outcome

Perfect pipeline

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ		-	-
3	NAND	ADD	BEQ	-	-
4	SW	NAND	ADD	BEQ	-
5	-	SW	NAND	ADD	BEQ
6	-	-	SW	NAND	ADD
7			-	SW	NAND
8				-	SW
9					

Unfortunately stalls ruin a perfect pipeline

Conservative Branch Handling

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

Cycle 2: Still Don’t Know If We Branch

Only at the end of EX stage is the target address of BEQ known

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

What should cycle 2 do?

- ➔ ID/RR tells IF to stall after fetching
- ➔ IF sends NOP to ID/RR at end of cycle

Cycle 3: Branch taken…

IF sends another NOP into ID/RR at the end of cycle 3

- (ADD is on the wrong path and gets squashed)

EX sets PC to PC + offset, so NAND is fetched

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

A – B

PC ← PC + offset

Assume BR is taken

Cycle 4: Branch Taken

Only at the end of the EX stage was the target addr of BEQ known

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

Assume BR is taken

We have these two NOPs introduced by BEQ stalling the pipeline

Cycle 9: Finished

Cycles 5-9, the pipeline runs normally (i.e. no new stalls)

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

Finish here with stalls

Cycle 4: Branch Not Taken

EX on cycle 3 allows IF and ID/RR to proceed normally on cycle 4

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	LW	ADD	NOP	BEQ	-
5		LW	ADD	NOP	BEQ
6	-		LW	ADD	NOP
7	-	-		LW	ADD
8			-		LW
9				-	

Finish here with stalls

What's the CPI difference?

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	(ADD)	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

Branch taken: 9 total cycles / 3 instructions = 3 CPI

Not taken: 8 total cycles / 3 instructions = 2.66 CPI

Strategies for improving branches

- Branch prediction
- Delayed branch

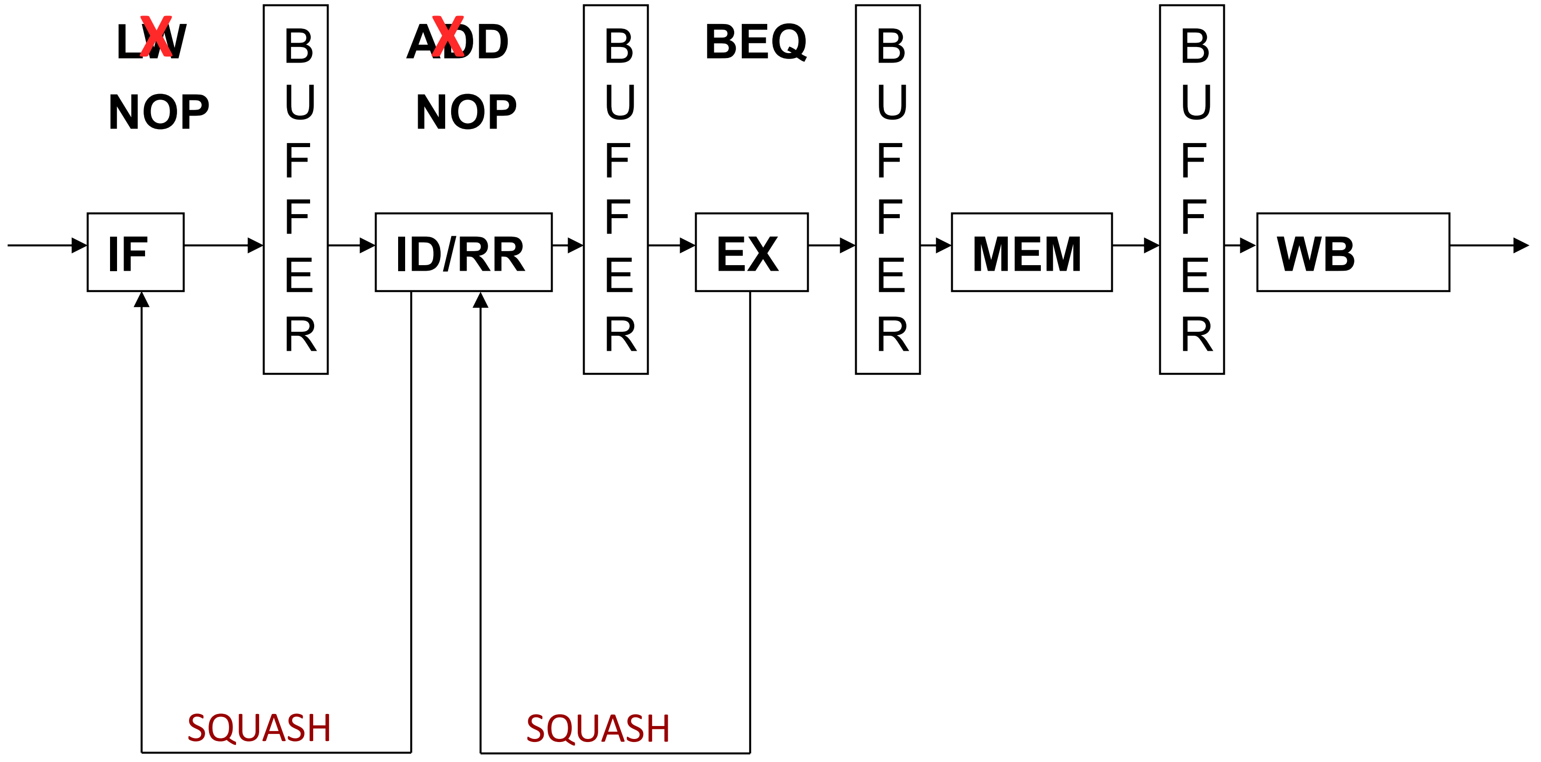
Branch Prediction

Assume branch is not taken

Cycle	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	LW	ADD	BEQ		

* We do not actually know what the instruction is in the Fetch Stage, before it's decoded

Branch Prediction



Our prediction was wrong! Branch will be taken
Squash the instructions at IF and ID/RR

Static Branch "not taken" prediction

Given the following sequence of instructions:

```
                BEQ             L1
                ADD
                LW
                ….
L1              NAND
                SW
```

Using the branch prediction approach, what is the observed CPI for the 3 instructions (BEQ, ADD, LW) or (BEQ, NAND, SW) in each case?

Happy path ☺ – prediction correct!

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	-	LW	ADD	BEQ	-
5	-	-	LW	ADD	BEQ
6	-	-	-	LW	ADD
7	-	-	-	-	LW

7 cycles / 3 instructions – 2.33 CPI

What if prediction turns out false?

Sad path ☹ – Branch prediction failed

Squash ADD and LW and replace with NOPs;
basically we degrade to the conservative approach

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

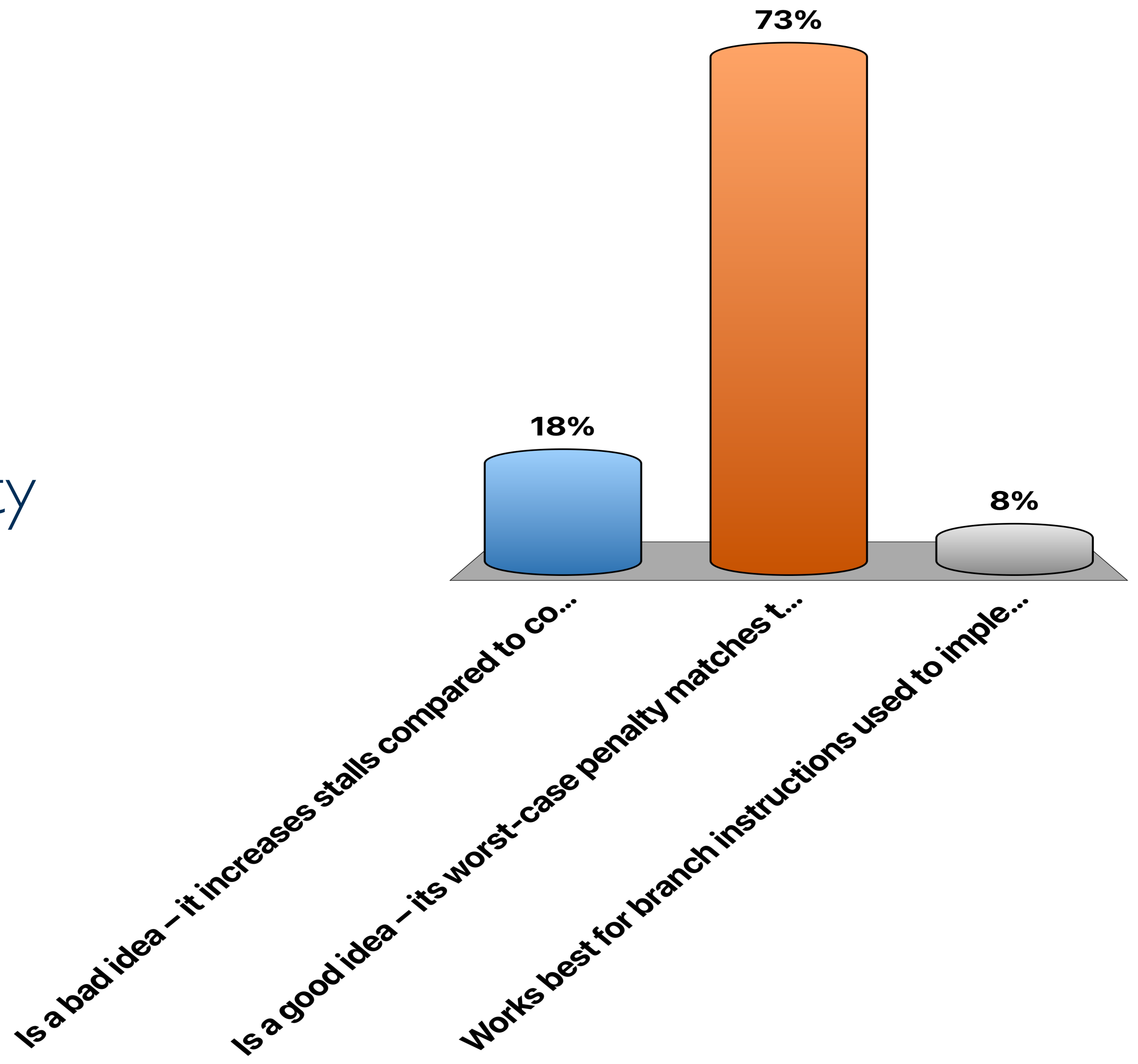
9 cycles / 3 instructions – 3 CPI

Static "not taken" prediction…

A. Is a bad idea – it increases stalls compared to conservative branch handling when most branches are actually taken

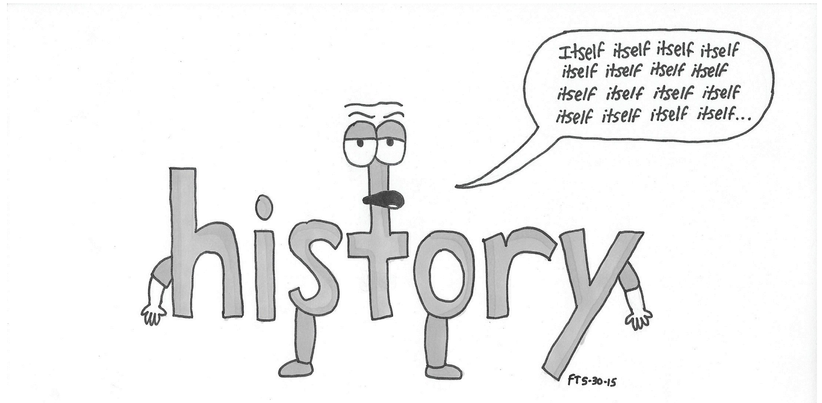
B. Is a good idea – its worst-case penalty matches that of conservative branch handling

C. Works best for branch instructions used to implement loops

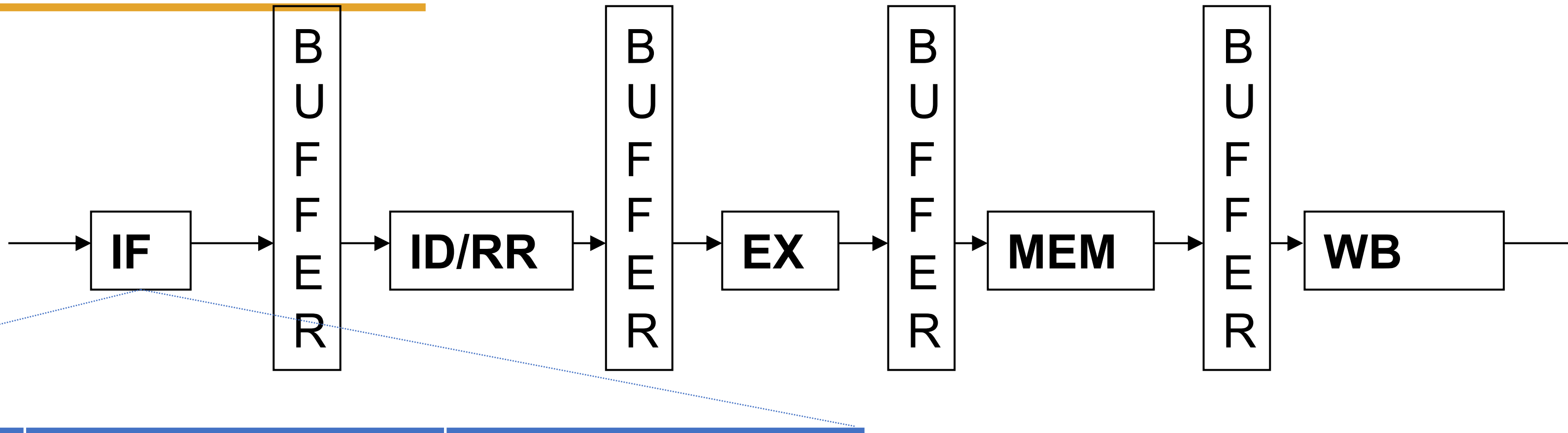


Branch prediction heuristics

- How do we (quickly) guess which way a branch will go?
- Can often tell by comparing target with current PC
  - Loops usually branch backwards (target < PC)
    - Guess the branch will be taken
  - Can construct conditionals to usually branch forwards (target > PC)
    - Guess the branch won't be taken
- Some ISAs have 2 different families of branch instructions so that the compiler can signal whether it thinks the branch will be taken or not taken
- Or we could keep a history…


Itself itself itself itself itself itself itself itself itself itself itself itself itself itself itself itself…
history
FTS-30-15

Branch History Table and Branch Target Buffer



PC	Instr
1000	BEQ
…	
1300	BEQ
…	
1500	BEQ
…	
1600	BEQ
…	
1800	BEQ
…	

PC of branch instruction	Taken/Not Taken	PC of branch target
1000	T	1200
1300	N	xxxx
1500	T	1100
1600	N	xxxx
1800	N	xxxx

Past history

Delayed branching

- Ignore the problem in hardware (i.e. no automatic generation of NOPs in fetch stage)
- Give the NOP instructions to the compiler
  - E.g. the instruction after a branch is *always* executed
- Let the compiler worry about it
  - Compile in a NOP instruction after every branch instruction
- Why could this be a good idea?
  - The compiler may be able to optimize…

Example using MIPS-style branch delays

```
; Add 7 to each element of a ten-element array
; whose address is in a0
        addi  $t1, $a0, 40 ; 40 = 10 elements * 4 bytes
  loop:
        beq   $a0, $t1, done ; When a0=t1 we are done
        nop                 ; branch delay slot
        lw    $t0, 0($a0)
        addi  $t0, t0, 7
        sw    $t0, 0($a0)
        addi  $a0, $a0, 4
        beq   $zero, $zero, loop
        nop                 ; branch delay slot
 done: halt
```

Let's assume one delay slot

Assume pipeline determines if branch is taken in its second stage

Delayed branch – before optimization

```
; Add 7 to each element of a ten-element array
; whose address is in a0
     addi  $t1, $a0, 40
 loop:
     beq   $a0, $t1, done
     nop                     ; branch delay slot
     lw    $t0, 0($a0)
     addi  $t0, t0, 7
     sw    $t0, 0($a0)
     addi  $a0, $a0, 4
     beq   $zero, $zero, loop
     nop                     ; branch delay slot
 done: halt
```

Delayed branch – during optimization

```
; Add 7 to each element of a ten-element array
; whose address is in a0
      addi  $t1, $a0, 40
  loop:
      beq   $a0, $t1, done
      nop                   ; branch delay slot
      lw    $t0, 0($a0)
      addi  $t0, $t0, 7
      sw    $t0, 0($a0)
      addi  $a0, $a0, 4
      beq   $zero, $zero, loop
      nop                   ; branch delay slot
 done: halt
```

OK if LW is redundantly executed on loop exit

Delayed branch – during optimization

```
; Add 7 to each element of a ten-element array
; whose address is in a0
       addi  $t1, $a0, 40
  loop:
       beq   $a0, $t1, done
       nop                      ; branch delay slot
       lw    $t0, 0($a0)
       addi  $t0, $t0, 7
       sw    $t0, 0($a0)
       addi  $a0, $a0, 4
       beq   $zero, $zero, loop
       nop                      ; branch delay slot
  done: halt
```

Delayed branch – after opt

```
; Add 7 to each element of a ten-element array
; whose address is in a0
       addi  t1, a0, 40
  loop:
       beq   $a0, $t1, done
       lw    $t0, 0($a0) ; branch delay slot

       addi  $t0, $t0, 7
       sw    $t0, 0($a0)

       beq   $zero, $zero, loop
       addi  $a0, $a0, 4 ; branch delay slot
  done: halt
```

Delayed branching

- Ignore the problem in hardware (i.e. no automatic generation of NOPs in fetch stage)
- Give the NOP instructions to the compiler
  - E.g. the instruction after a branch is *always* executed
- Let the compiler worry about it
  - Compile in a NOP instruction after every branch instruction
- Why could this be a good idea?
  - The compiler may be able to optimize…
- Why is this mostly a bad idea?
  - Exposes microarchitectural detail to programmer – breaks abstraction!
  - Dynamic branch prediction proved to be very effective

Name	Pros	Cons	Use cases
Stall the pipeline	Simple strategy, no hardware needed for squashing instructions	Loss of performance	Early pipelined machines such as IBM 360 series
Branch Prediction (branch not taken)	Results in good performance with virtually no additional hardware since the instruction is anyhow being fetched from the sequential path already in IF stage	Needs ability to squash instructions in partial execution in the pipeline	Processors such as Intel Pentium, AMD Athlon, and PowerPC use this technique; typically they also employ sophisticated branch target buffers; MIPS R4000 uses a combination 1-delay slot plus a 2-cycle branch-not-taken prediction
Dynamic History-based Branch Prediction	Results in good performance but requires slightly more elaborate hardware design	Since the new PC value that points to the target of the branch is not available until the branch instruction is in EX stage, this technique requires more elaborate hardware assist to be practical	-
Delayed Branch	No need for any additional hardware for either stalling or squashing instructions; It involves the compiler by exposing the pipeline delay slots and takes its help to achieve good performance	With increase in depth of pipelines of modern processors, it becomes increasingly difficult to fill the delay slots by the compiler Exposes microarchitectural details to ISA level Assembly harder to understand	Older RISC architectures such as MIPS, PA-RISC, SPARC

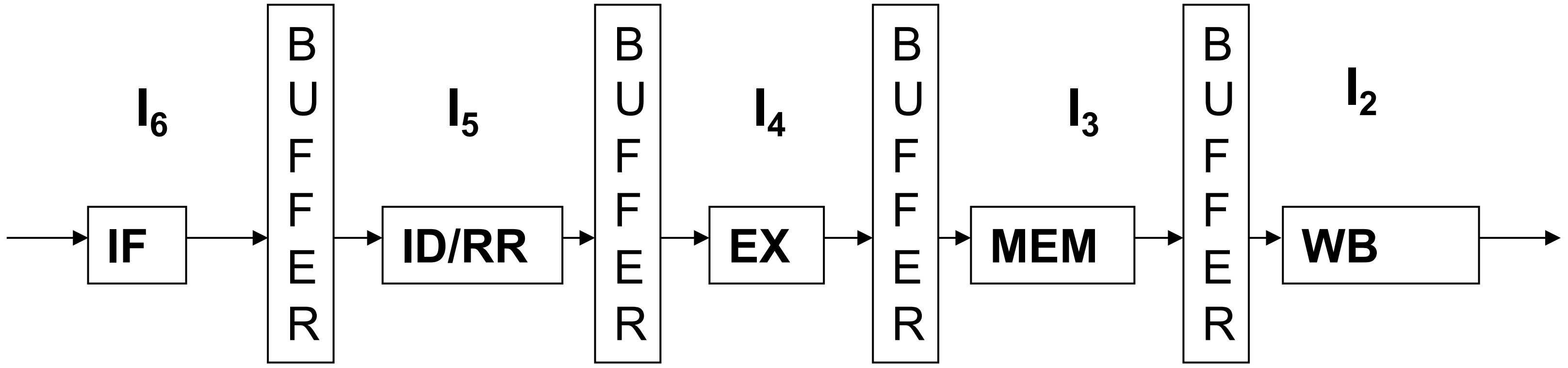
Summary of hazards

Instruction	Type of Hazard	Potential Stalls	With Data forwarding	With branch prediction (branch not taken)
ADD, NAND	Data	0, 1, 2, or 3	0	Not Applicable
LW	Data	0, 1, 2, or 3	0 or 1	Not Applicable
BEQ	Control	1 or 2	Not Applicable	0 (success) or 2 (mispredict)

* Structural hazards can always be eliminated by replicating bottleneck hardware resource

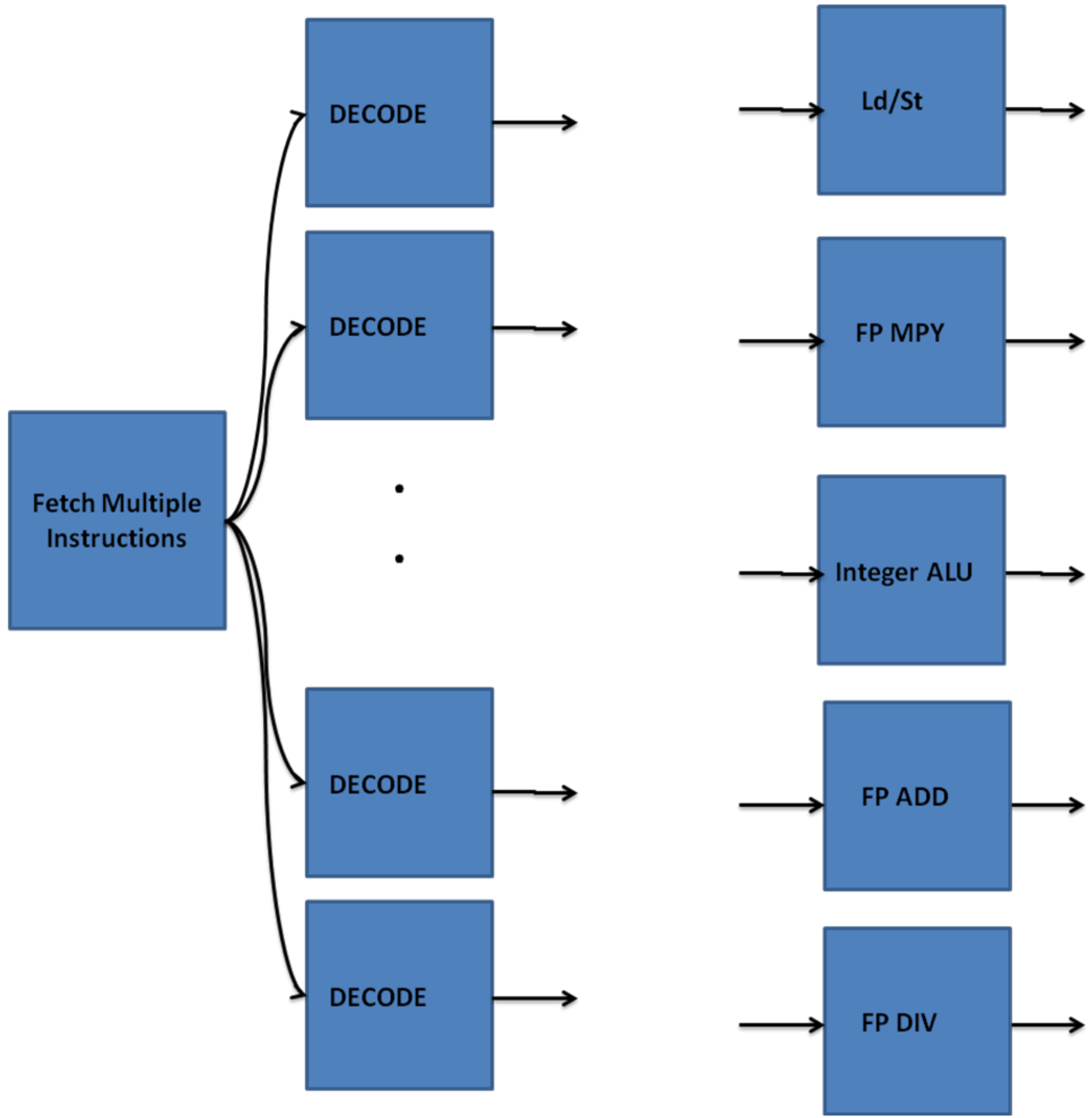
** These penalties refer to the *specific 5-stage pipeline* we saw. Different penalties for different pipelines.

Program discontinuities

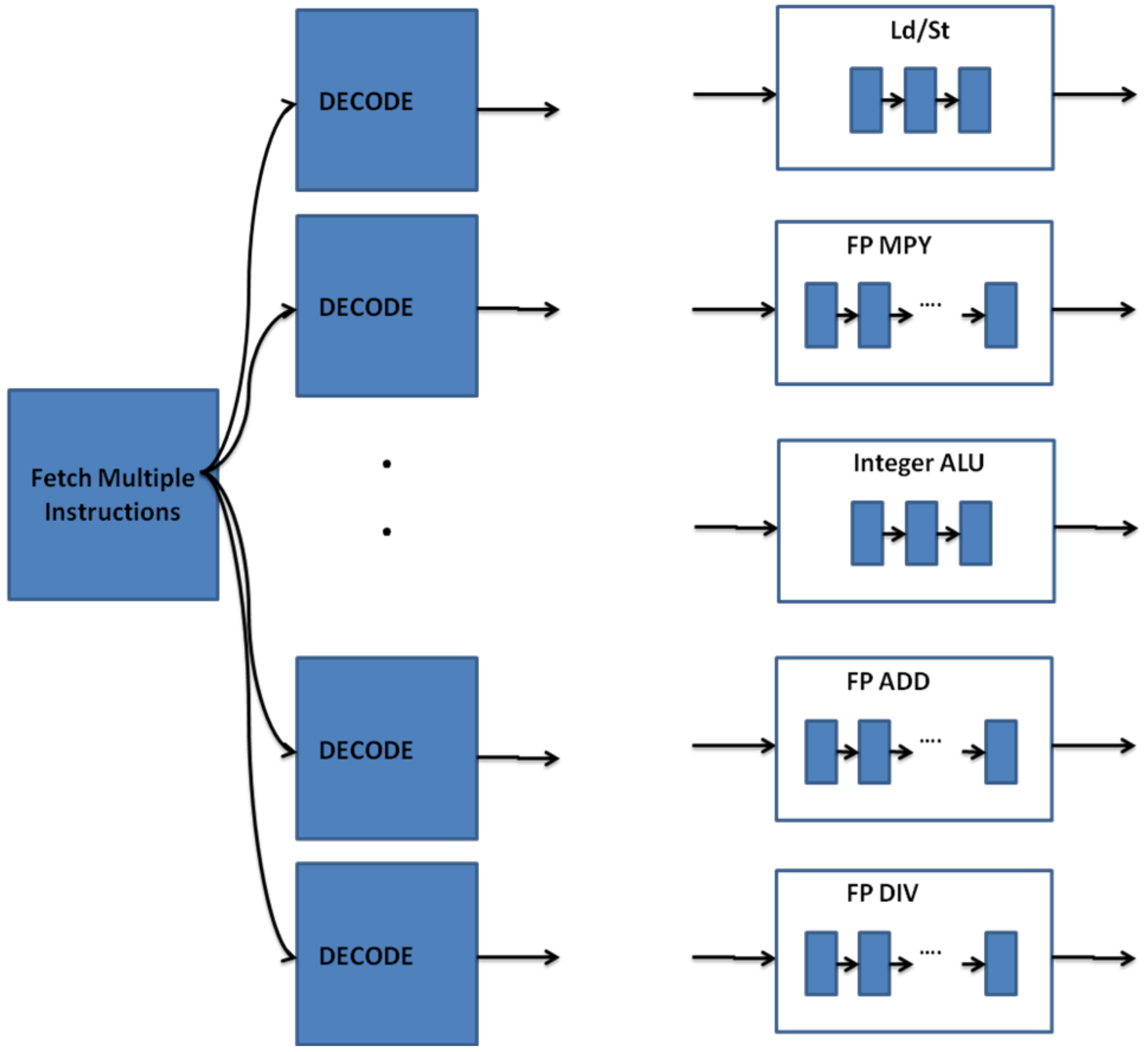


- External interrupt can occur at any time
- When do we take the interrupt?
- Any one of the 5 instructions may cause an exception or trap
- As soon as an instruction raising trap/exception reaches WB: STOP, FLUSH, go to INT state
  - Slow to respond to external events

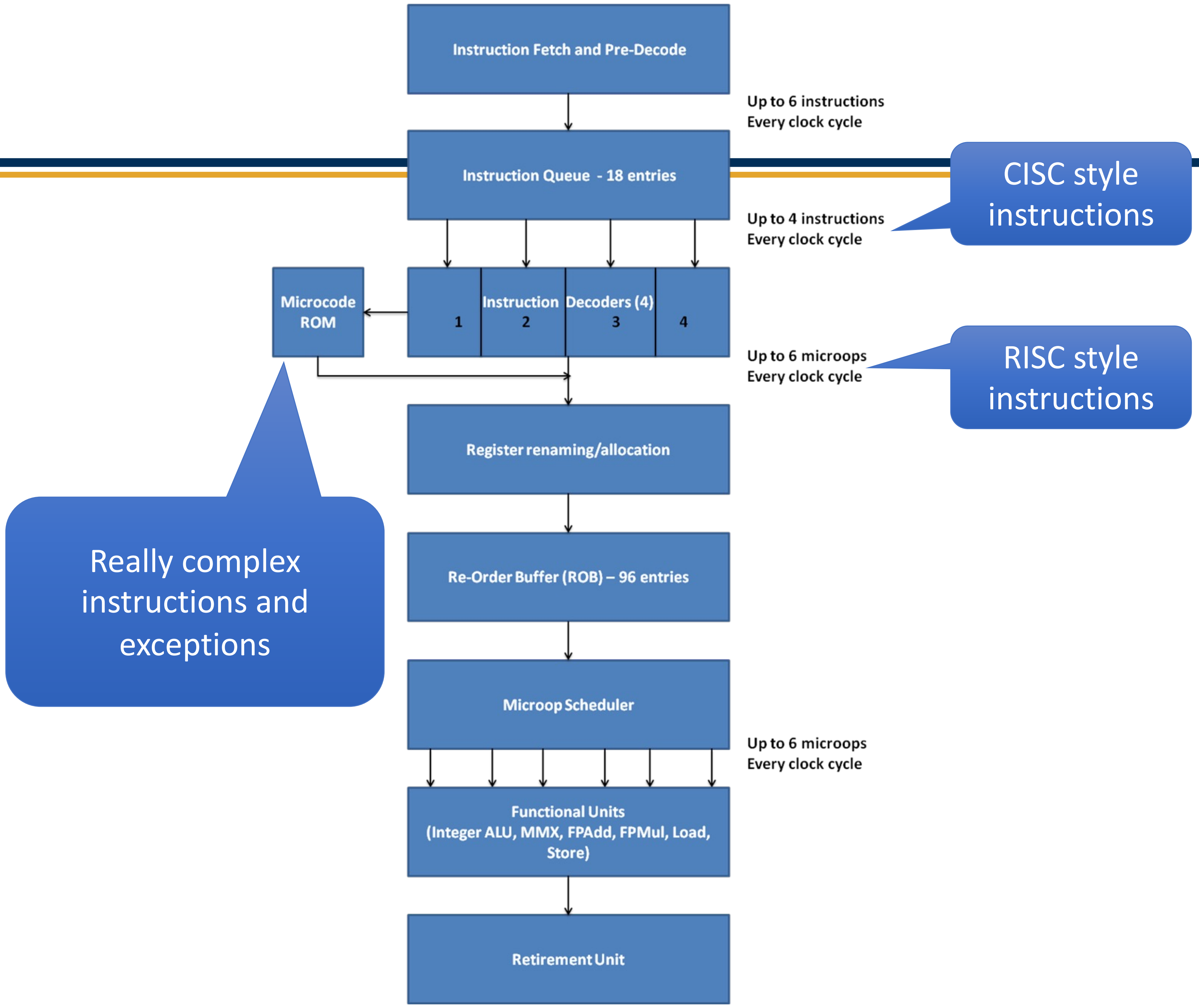
Superscalar pipelines



Different Pipeline Depths



Intel Core Microarchitecture: An example pipeline


Instruction Fetch and Pre-Decode
Up to 6 instructions Every clock cycle
Instruction Queue - 18 entries
Up to 4 instructions Every clock cycle
CISC style instructions
Microcode ROM
Instruction Decoders (4)
1
2
3
4
Up to 6 microops Every clock cycle
RISC style instructions
Really complex instructions and exceptions
Register renaming/allocation
Re-Order Buffer (ROB) – 96 entries
Microop Scheduler
Up to 6 microops Every clock cycle
Functional Units (Integer ALU, MMX, FPAdd, FPMul, Load, Store)
Retirement Unit