
Lecture 13

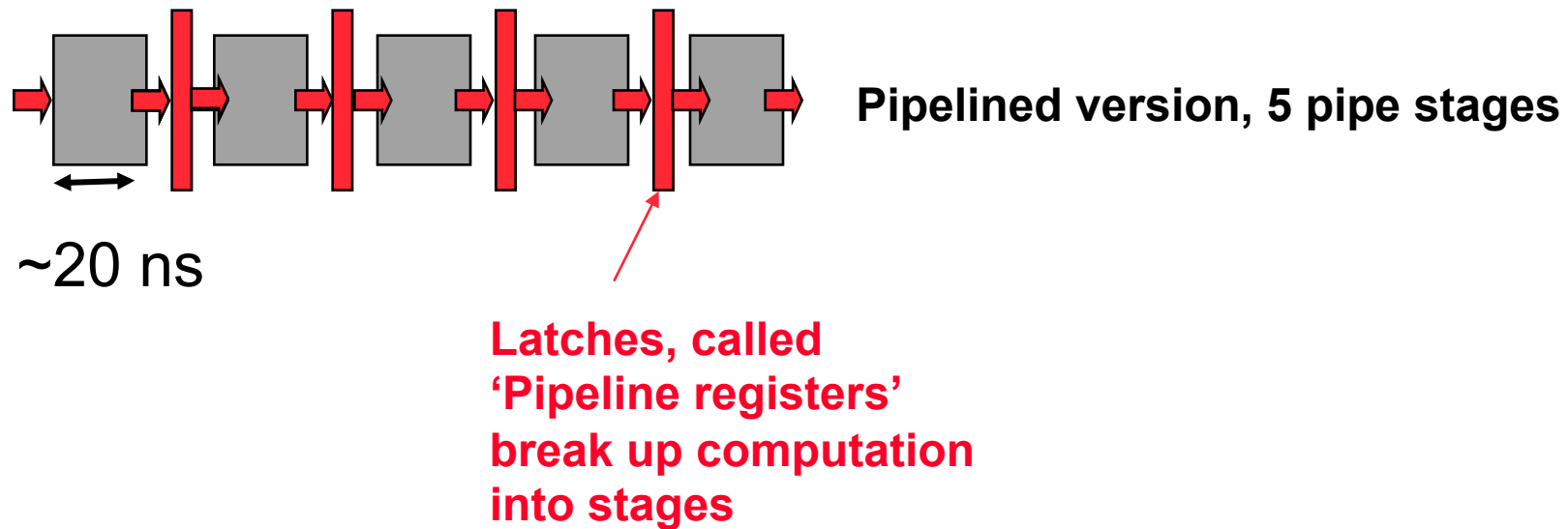
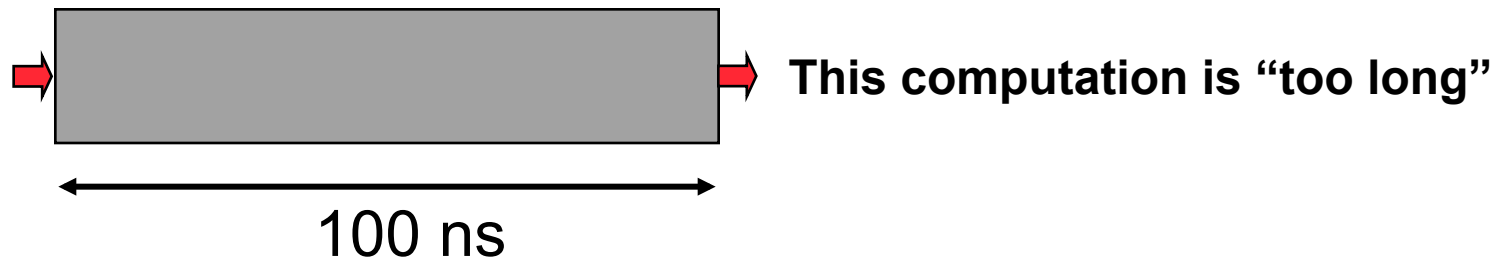
Pipeline Implementation Pragmatics:

Branch Prediction, Exception Handling

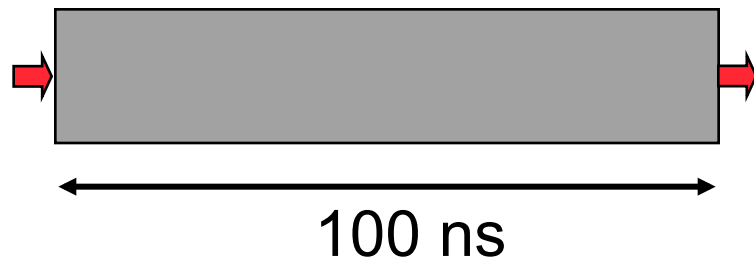


Adapted from slides developed by Profs. Hardavellas, Falsafi, Hill, Marculescu, Patterson, Rutenbar and Vijaykumar of Northwestern, Carnegie Mellon, Purdue, Berkeley, UWisconsin

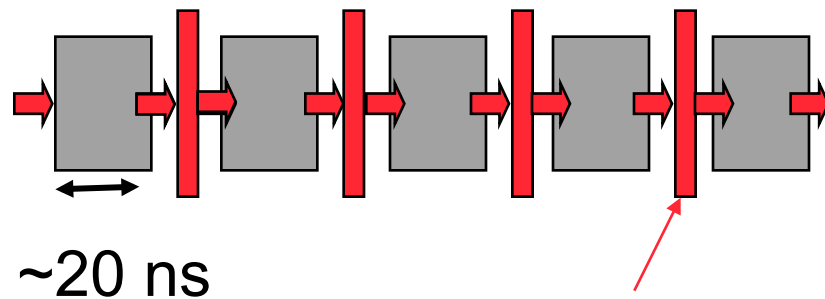
Big Idea: Pipeline Concurrency



Big Idea: It's Faster



I can “launch” a new computation every **100ns** in this structure



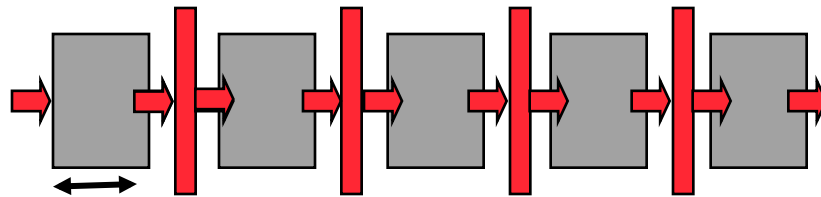
Pipelined version, 5 pipe stages:
I can launch a new computation every **20ns** in pipelined structure

Latches, called
‘Pipeline registers’
break up computation
into stages

Pipelining: Implementation Issues

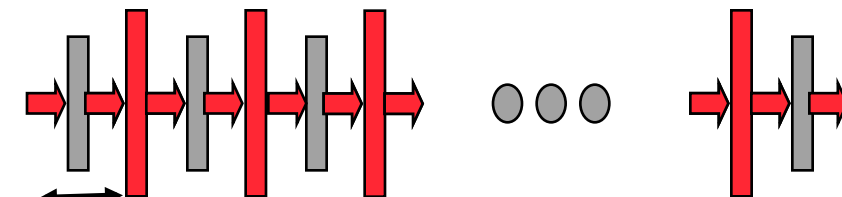
► What prevents us from just doing a zillion pipe stages?

- ▷ Some computations just won't divide into any **finer** (shorter in time) logical implementations
- ▷ Ultimately, often comes down to **circuit** design issues



5 stages: OK

~20 ns



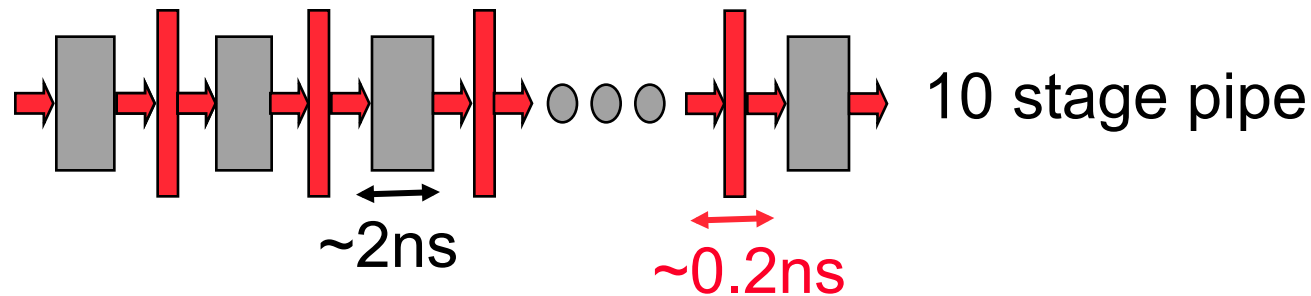
50 stages: nope, sorry

~2 ns

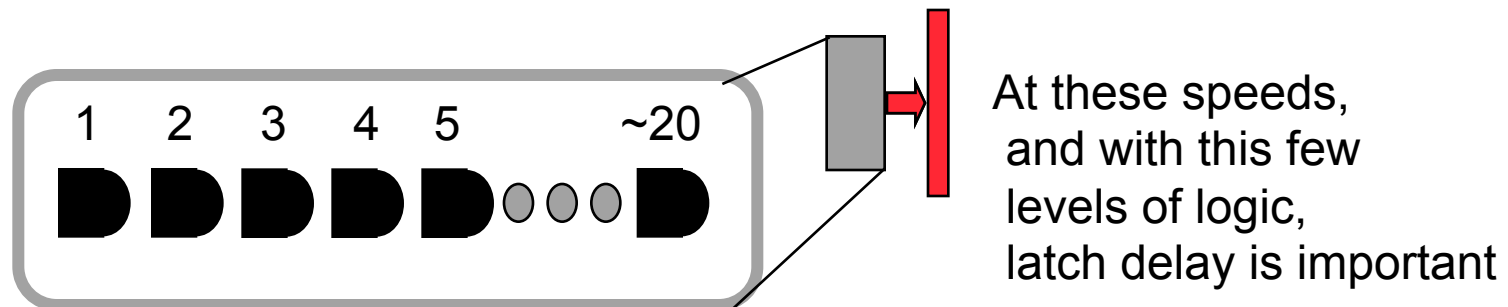
Pipelining: Implementation Issues

► What prevents us from just doing a zillion pipe stages?

- ▷ Those latches are **NOT** free, they take up **area**, and there is a real **delay** to go THROUGH the latch itself



- ▷ In modern, deep pipelines (10-20 stages), this is a real effect
- ▷ Typical logic “depths” in one stage of 10-20 “gates”; optimal delay rumored to be $\sim 6\text{-}8\text{ FO4}$



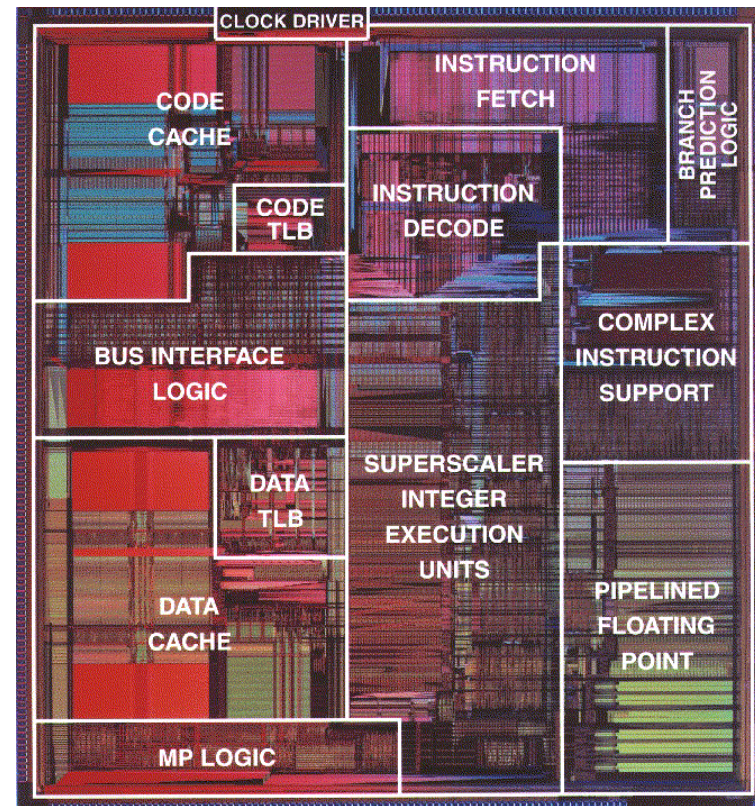
How Many Pipeline Stages?

► E.g., Intel

- ▷ Pentium 4 (2002): 20 stages
- ▷ High clock frequency (>1GHz)
- ▷ High IPC (Instructions per Cycle)
- ▷ Prescott(2004)&Cedar Mill(2006):31 stages !!!
- ▷ Core i7 (2011): 14 stages
- ▷ Xeon E7 (2015): 14 – 19 stages
 - ▷ 224 instructions in flight

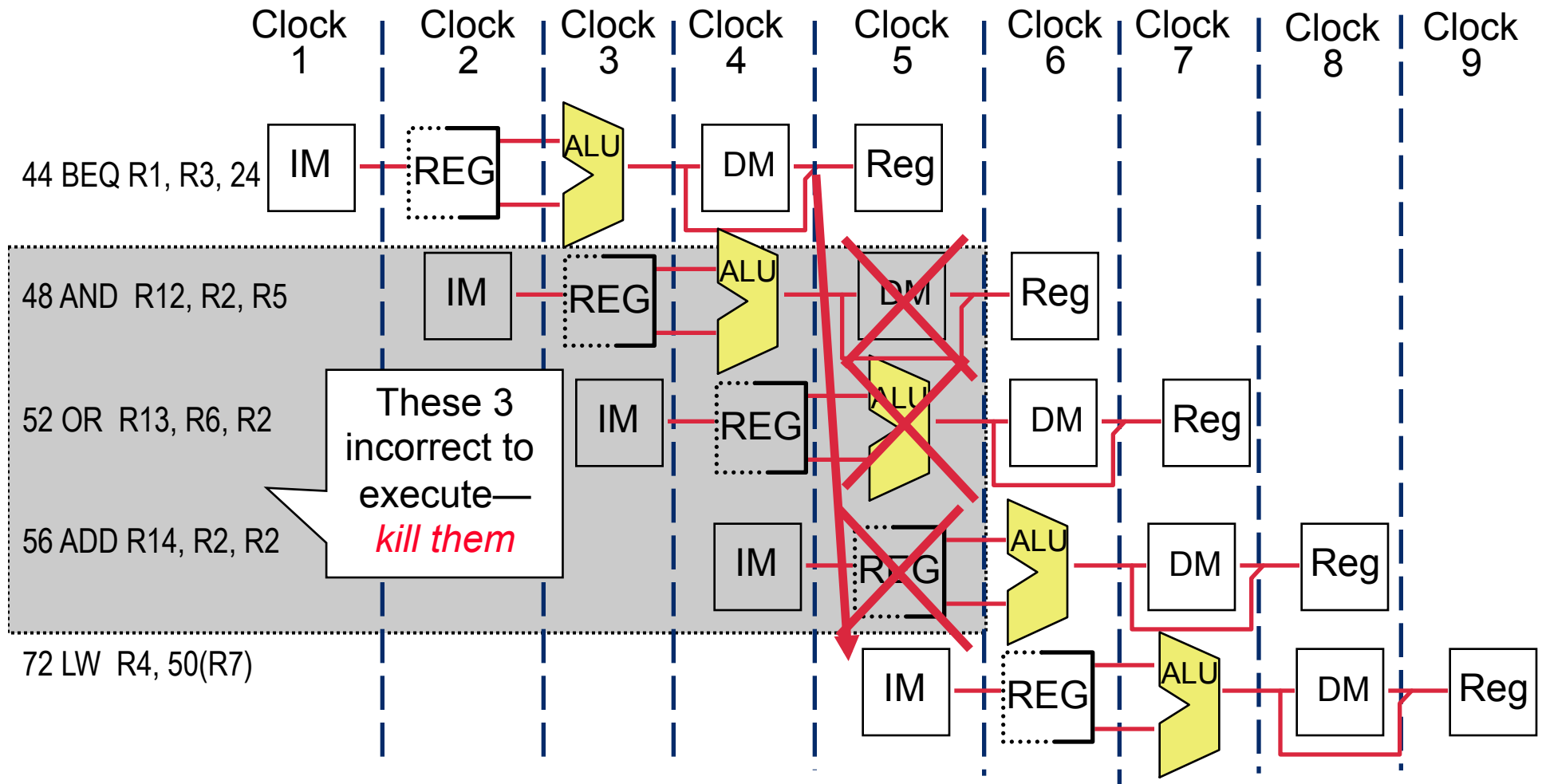
► Too many stages:

- ▷ Lots of complications
- ▷ Should take care of possible dependencies among in-flight instructions
- ▷ Control logic is huge
- ▷ Control hazards also a big problem



What Happens When the Branch IS Taken

Flow of instructions if branch is taken: 36, 40, 44, 72, ...



Effect of Control Hazards in Pipelines

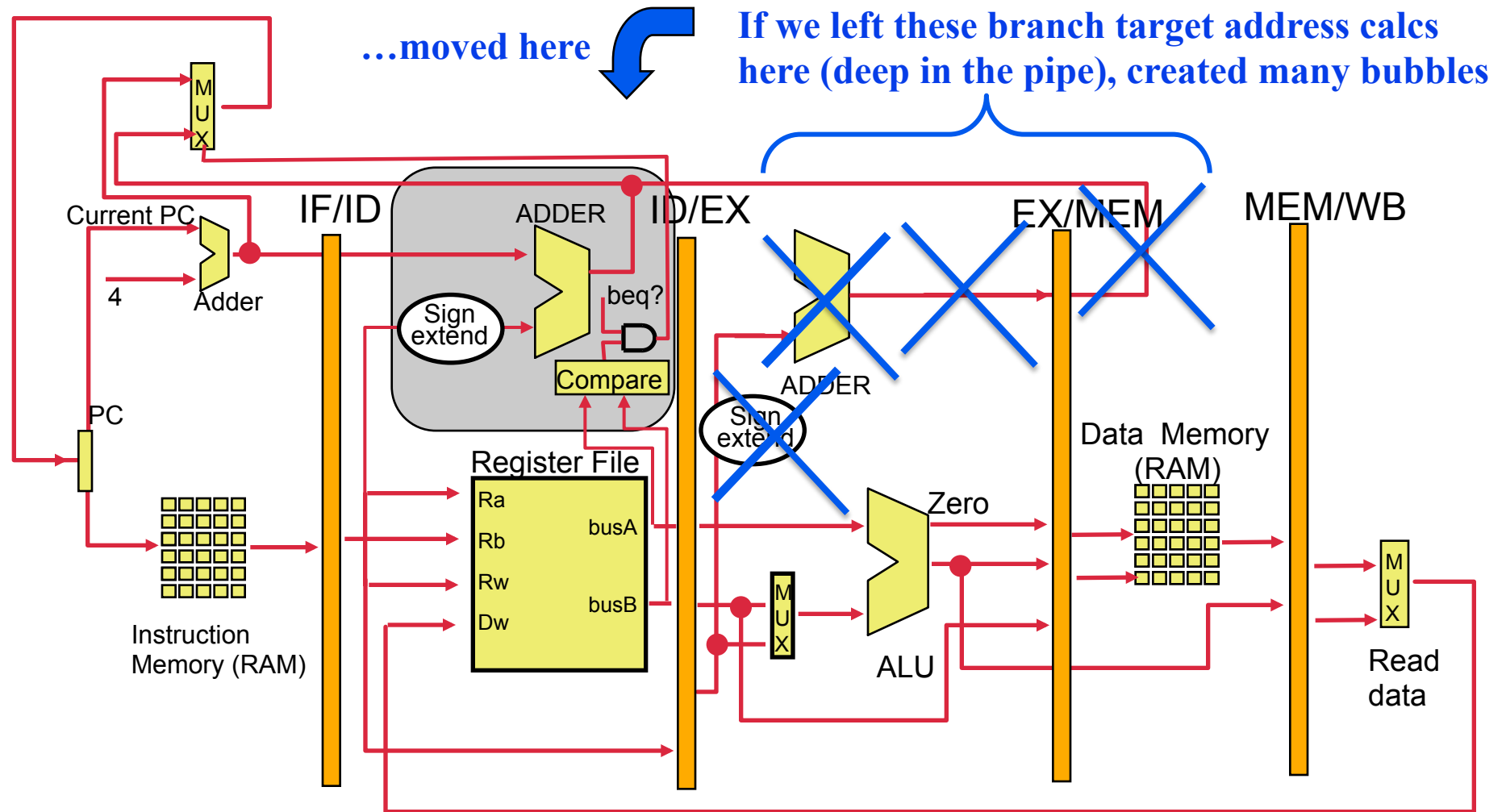
► We mistakenly start executing the wrong instructions

- ▷ To avoid getting wrong answer, must make sure that they DO NOT really execute
 - ▷ Must ensure they do not incorrectly corrupt machine state
 - ▷ How? Make sure they produce no output (NO reg or mem writes) – set controls LOW
“I didn’t do it; nobody saw me doing it!”

► Terminology

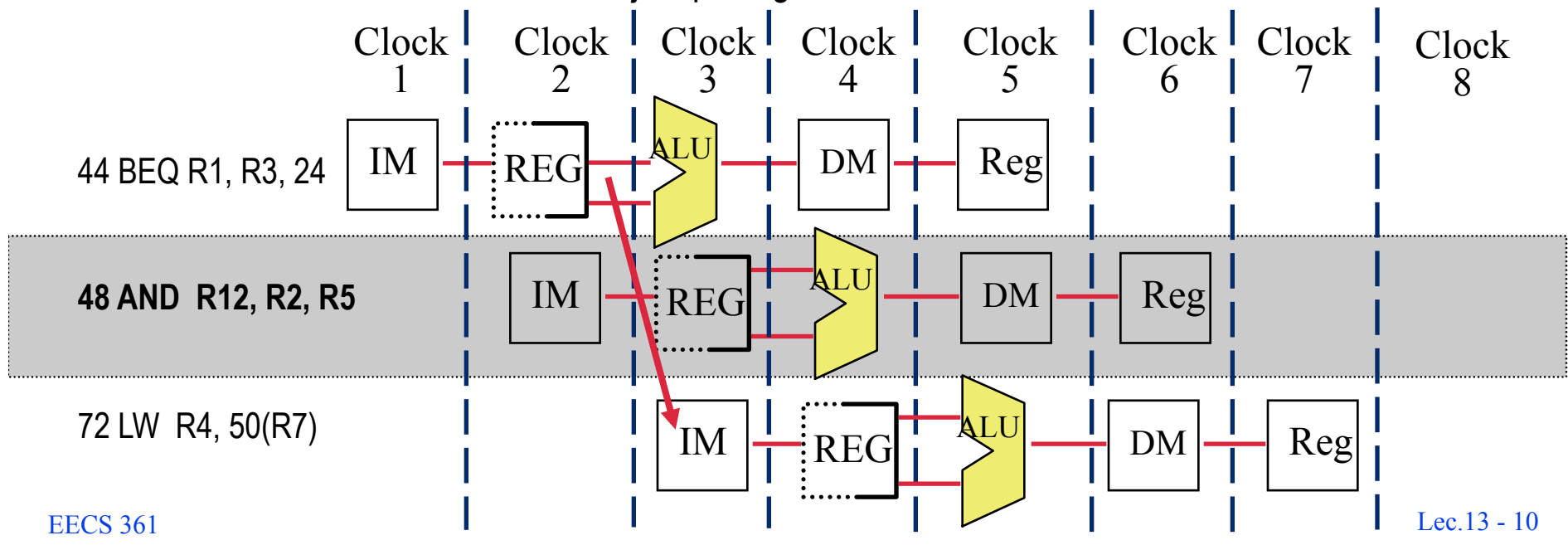
- ▷ We “squash” the instructions
- ▷ Why? Because we insist on “sequential execution semantics”,
i.e., “*the program behaves like the instructions execute sequentially, in program order*”
- ▷ We say the machine **executes the instructions sequentially**
- ▷ Also say “**instructions are RETIRED in sequential order**”
- ▷ Image is: instruction is born (fetched), grows up (reg access, ALU ops), then finally finishes and commits correct machine state. This “finally finishes” is “retiring” the instruction
- ▷ In deep pipelines and complex machines, even knowing WHEN your instruction retires takes a lot of complex logic

Move the Branch Computation Further Forward



The Branch Problem

- ▶ Need just 1 extra cycle after the BEQ branch to know the right PC address
- ▶ Branch is detected and handled in cycle 2
- ▶ Allows branch destination to start in cycle 3
- ▶ What about the instruction fetched in cycle 2? (in our example, **AND**)
 - ▷ Some architectures stall and lose performance
 - ▷ MIPS uses a “branch delay slot”: you (or your compiler) need to **adjust your code** to put some useful work in that “slot”, since just putting in a NOP is wasteful



Today's Menu

- ▶ Hazards and the role of the compiler
- ▶ Branch Prediction
- ▶ Exceptions
- ▶ Examples


Rewriting the Code for Delay Slots and Data Hazards

- ▶ Stalls due to data and control hazards make performance less than one instruction per cycle

- ▶ Compiler is critical in determining overall performance

- ▷ Compiler generates code that avoids stalls

- ▶ **Example**



```
lw R15, 0x00(R2)
add R14, R15, R15
lw R16, 0x04(R2)
```

- ▷ Might become:

```
lw R15, 0x00(R2)
lw R16, 0x04(R2)
add R14, R15, R15
```

- ▶ How to find enough instructions to mitigate data/control hazards?

Loop Unrolling

- ▶ Reduce stall cycles by loop unrolling and code reordering

Loop:	lw	R2, 0(R1)	}	Loop body
	lw	R3, 100(R1)		
	add	R2, R2, R3		
	sw	R2, 200(R1)		
	addi	R1, R1, 4	}	Calculate next iteration's index
	beq	R1, R6, loop	}	Go execute the next loop iteration

- ▶ The 2nd lw instruction needs a stall cycle
- ▶ The beq instruction has an unused delayed branch slot

Loop Unrolling

- Unroll the loop once (duplicate body using more registers):

Loop: **lw R2, 0(R1)**

lw R3, 100(R1)

add R2, R2, R3

sw R2, 200(R1)

lw R4, 4(R1)

lw R5, 104(R1)

add R4, R4, R5

sw R4, 204(R1)

addi R1, R1, 8

beq R1, R6, loop

Loop body for iteration k

Loop body for iteration k+1

Calculate iteration's k+2 index

Go execute the next iteration k+2 and k+3

Loop Unrolling

► Reorder the code so that no extra cycles are needed

- ▷ Resolve the data hazards, fill the branch delay slot

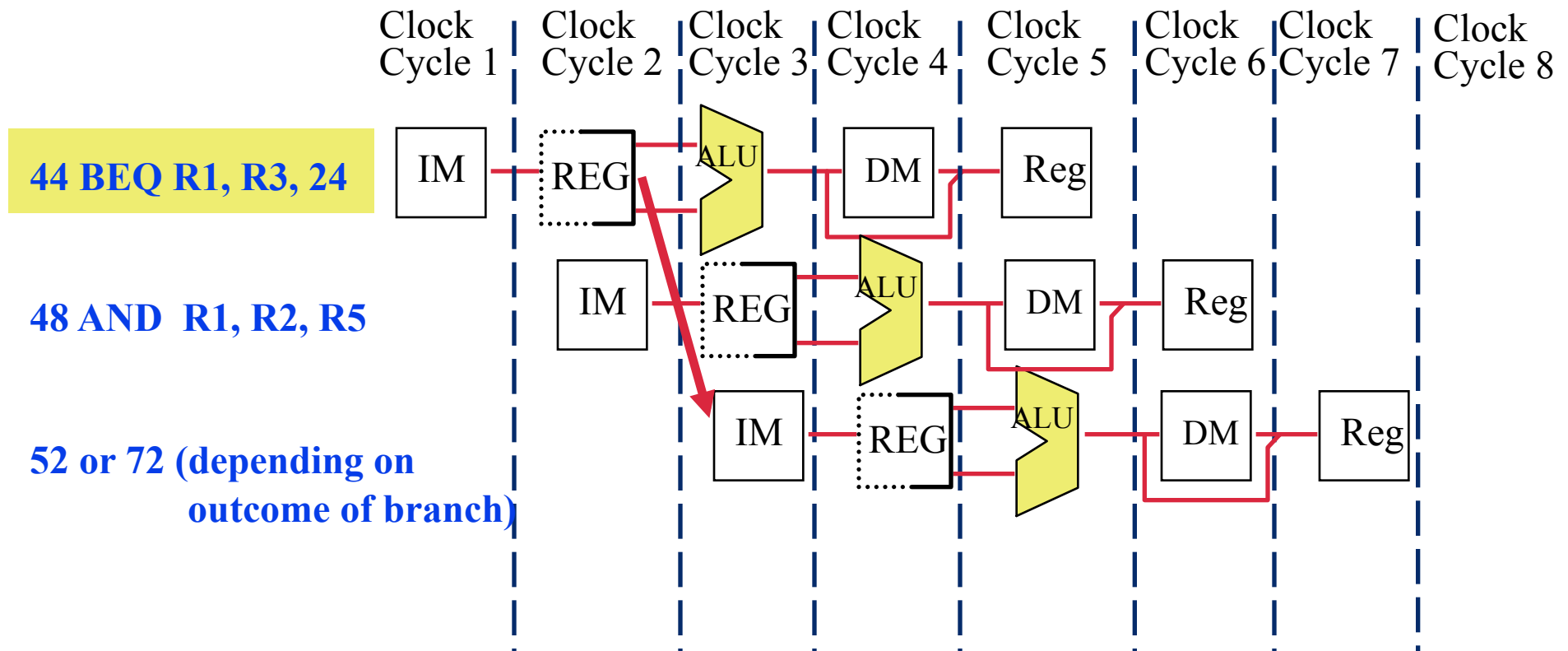
Loop: **lw R2, 0(R1)**
lw R3, 100(R1)
lw R4, 4(R1)
lw R5, 104(R1)
add R2, R2, R3
add R4, R4, R5
sw R2, 200(R1)
addi R1, R1, 8
beq R1, R6, loop
sw R4, 196(R1)

The Branch Delay Slot

- ▶ **In retrospect, the branch delay slot is probably a mistake**
- ▶ **This solution only works for some pipelines**
 - ▷ Deeper pipelines mean higher performance
 - ▷ They also mean more cycles between when you fetch an instruction and when you know for sure the address of the next instruction
 - ▷ # of branch delay slots would have to grow
- ▶ **Breaks the atomic instruction principle**
- ▶ **Compilers don't always find a way to fill the slot**
- ▶ **Forget about it for a minute...**
- ▶ **Let's try a better way: make an educated guess early (prediction)**

Branch Prediction: A better solution?

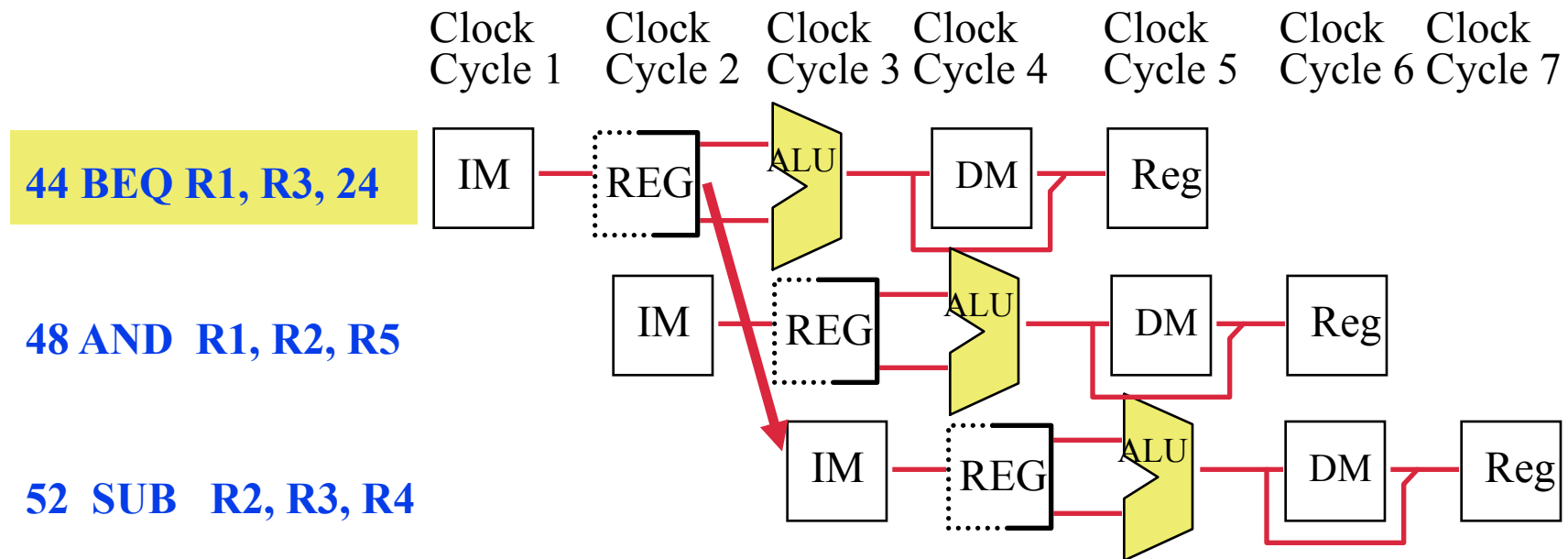
- Assume branch **not taken**; so just start **AND** instruction



- This is a form of **Branch Prediction**

Predict Branch Not Taken

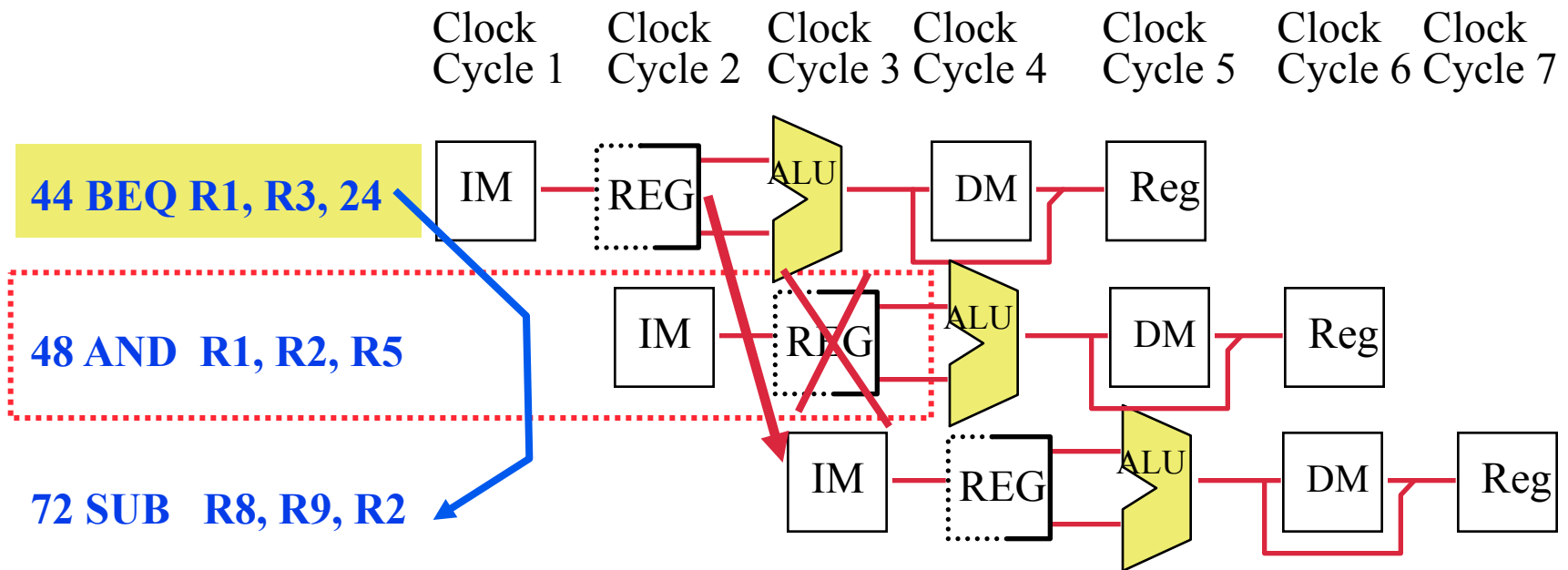
- ▶ Instead of a branch delay slot or stalling, we just **assume** that the branch will **not** happen
 - ▷ If you're right, great!
 - ▷ If your wrong, cancel the instructions that should not have executed
- ▶ **Example:**
 - ▷ Assume **"not taken"** when the branch is **not** taken



Branch Misprediction

► Example:

- ▷ Assume “**not taken**” when the branch is **taken**
- ▷ **Squash** instruction 48 (**AND**) because it should not have issued



How Can We Do Better? *Branch Prediction*

- ▶ **There are many different schemes**

- ▷ Assume taken
- ▷ Assume not taken
- ▷ 1-bit Branch Prediction
- ▷ 2-bit Branch Prediction
- ▷ N-bit Branch Prediction
- ▷ Table-based Branch Prediction
- ▷

- ▶ **Assume taken or not taken is called static branch prediction**

- ▶ **Using hardware to dynamically predict is called dynamic branch prediction**

Static Prediction Problems

- ▶ **Some branches: 99% of the time they are taken**
 - ▷ Example: “are we finished with the loop? if not start at the top again...”
- ▶ **Some branches: 99% of the time they are not taken**
 - ▷ Example: “is this an error? If so branch to error routine”
- ▶ **Compilers have trouble re-writing to make static behavior the right behavior**

Dynamic Branch Prediction

- ▶ **Dynamic branch prediction uses the previous outcome of a branch to determine future outcomes**
 - ▷ What happened in the past, will happen in the future (history repeats)
 - ▷ Does this make sense?

- ▶ **Yes, sometimes. Consider the following code fragment**

```
for (k = 0; k < 100000000; k++) {  
    /* do something */  
}
```

**Most of the time,
the “last” branch
decision is same as
next branch decision**

- ▶ **But what about**

```
if (a == b) { /* do something */ }  
else { /* do something else */ }
```

**Hmmm...
Not so clear here, eh?**

1-bit Branch Prediction

► Hardware has a table of single bits

- ▷ Each entry in the table corresponds to a branch in the program
- ▷ If a bit is **set**, the branch is predicted **taken**
- ▷ If the bit is **not** set (0), the branch is predicted **not** taken

Branch Prediction Table

0	0	L1	ADD	R1, R2, R3
1	1		SUB	R3, R4, R2
2	1		BEQ	R1, R3, L1
3	0	L2	LUI	R2, 0x1234
4	0		BNE	R3, R4, L2
5	0		J	L1

- ▷ How do the branch table bits get set?
 - ▷ The hardware determined the real outcome of a branch and uses that outcome (history) to set (or unset) a bit
- ▷ How do the branch instructions get mapped to entries in the table?
 - ▷ *Magic for now... Lots of custom logic, basically (e.g., hashing on the PC)*

1-bit Branch Prediction (cont.)

Flow of instructions

```
ADD  R1, R2, R3
SUB  R3, R4, R2
BEQ  R1, R3, L1
LUI  R2, 0x1234
```

;table predicts not taken

Branch Prediction Table at start of program

0	0	L1	ADD	R1, R2, R3
1	1		SUB	R3, R4, R2
2	1		BEQ	R1, R3, L1
3	0	L2	LUI	R2, 0x1234
4	0		BNE	R3, R4, L2
5	0		J	L1

; if the branch was taken, then squash LUI

let's assume the branch was taken

;then the hardware will **update** the table

;next, we fetch the destination of the branch

```
ADD  R1, R2, R3
SUB  R3, R4, R2
BEQ  R1, R3, L1
ADD  R1, R2, R3
```

; table predicts taken

Branch Predict Table after first branch resolved

0	1	L1	ADD	R1, R2, R3
1	1		SUB	R3, R4, R2
2	1		BEQ	R1, R3, L1
3	0	L2	LUI	R2, 0x1234
4	0		BNE	R3, R4, L2
5	0		J	L1

Problem with one-bit predictors

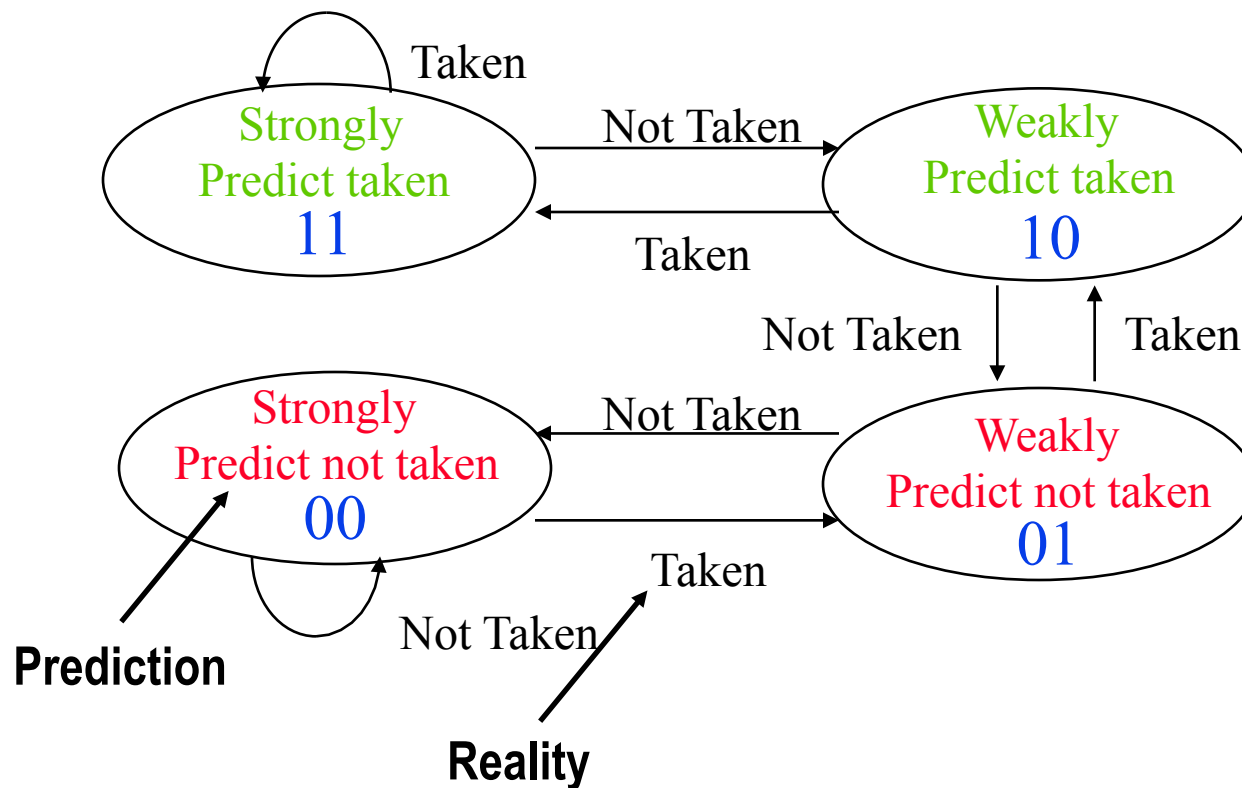
- Consider this:

```
for (j = 0; j < 100000000; j++) {  
    for (k = 0; k < 10; k++) {  
        /* do something */  
    }  
}
```

- How often do we mispredict the k-loop branch?

2-bit Branch Prediction

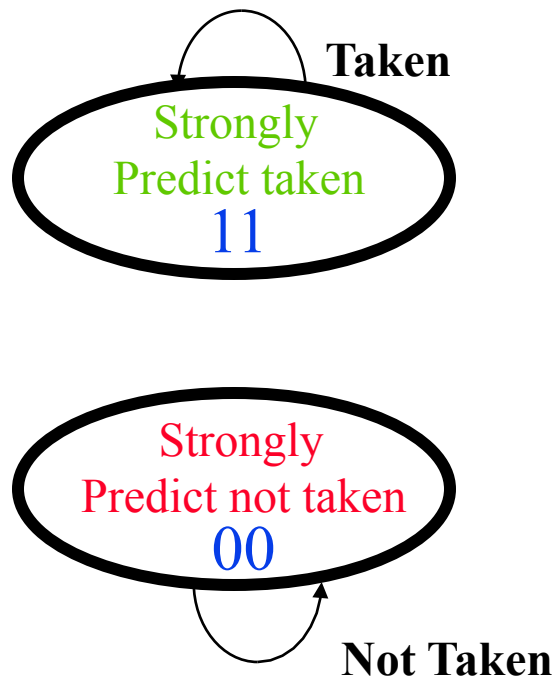
- ▶ Table has 2 bits instead of one bit
- ▶ Creates a history--more “memory” of behavior saved
 - ▷ Use the entries to determine the outcome of a branch as follows



2-bit Branch Prediction: Mechanics

► If you're right--you're right

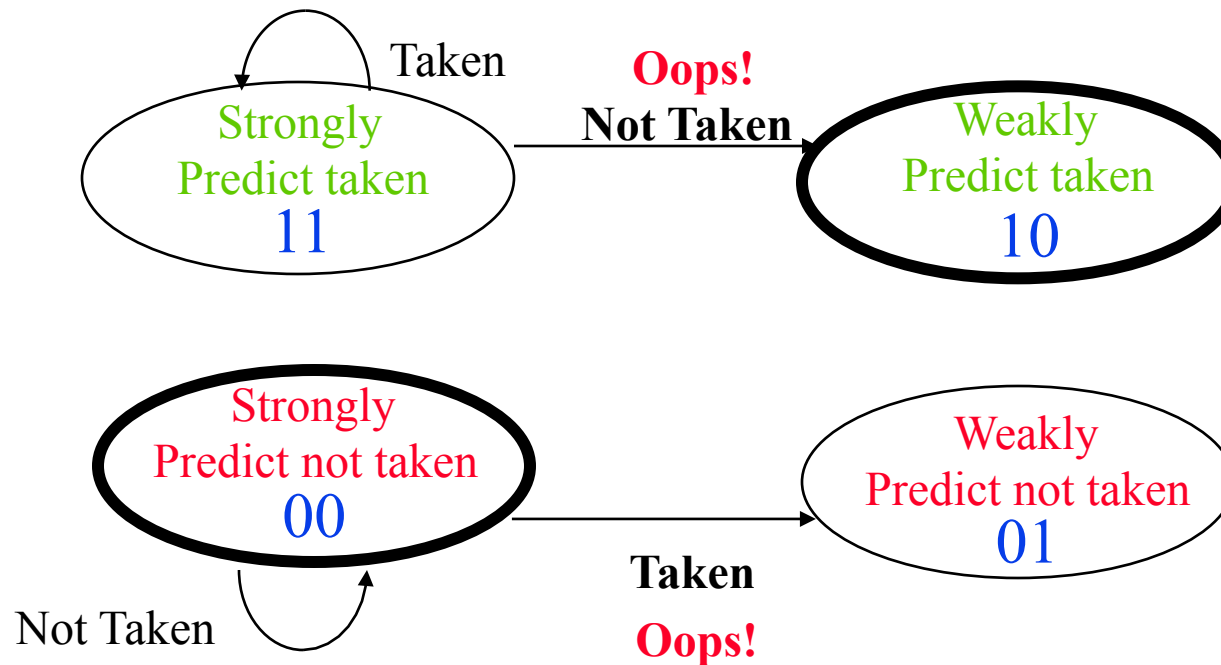
- ▷ Don't change your prediction if things are going OK with it...



2-bit Branch Prediction: Mechanics

► Oops--first wrong, mis-predict.

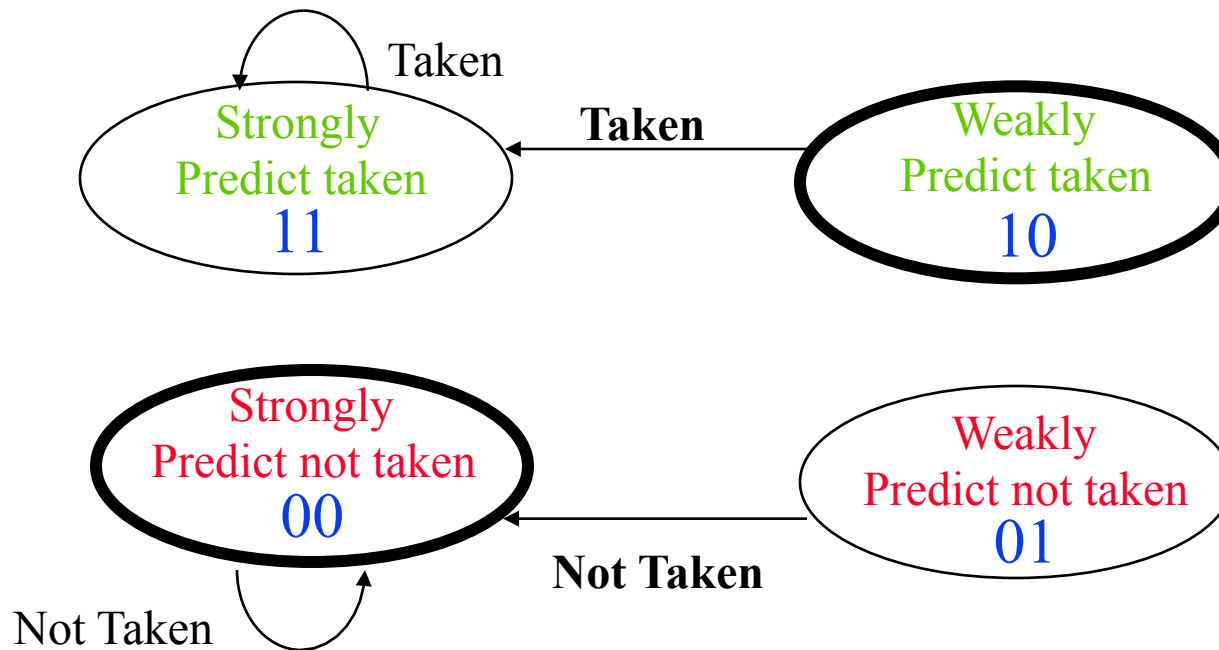
- ▷ Remember that this was first--it's a new state
- ▷ BUT--don't change your prediction about the branch direction



2-bit Branch Prediction: Mechanics

► If we're lucky, that mispredict was a fluke...

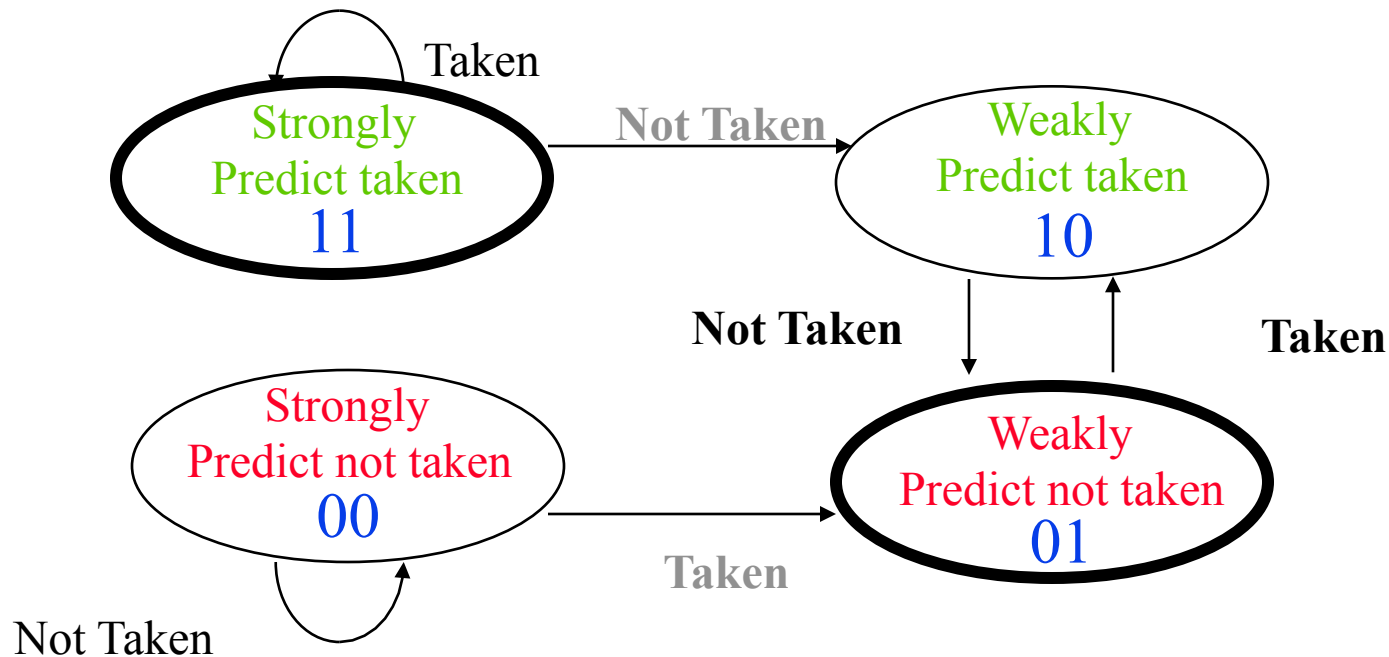
- ▷ The way we WERE predicting it before was OK, that one previous branch was just wrong. NEXT one, we're right again.



2-bit Branch Prediction: Mechanics

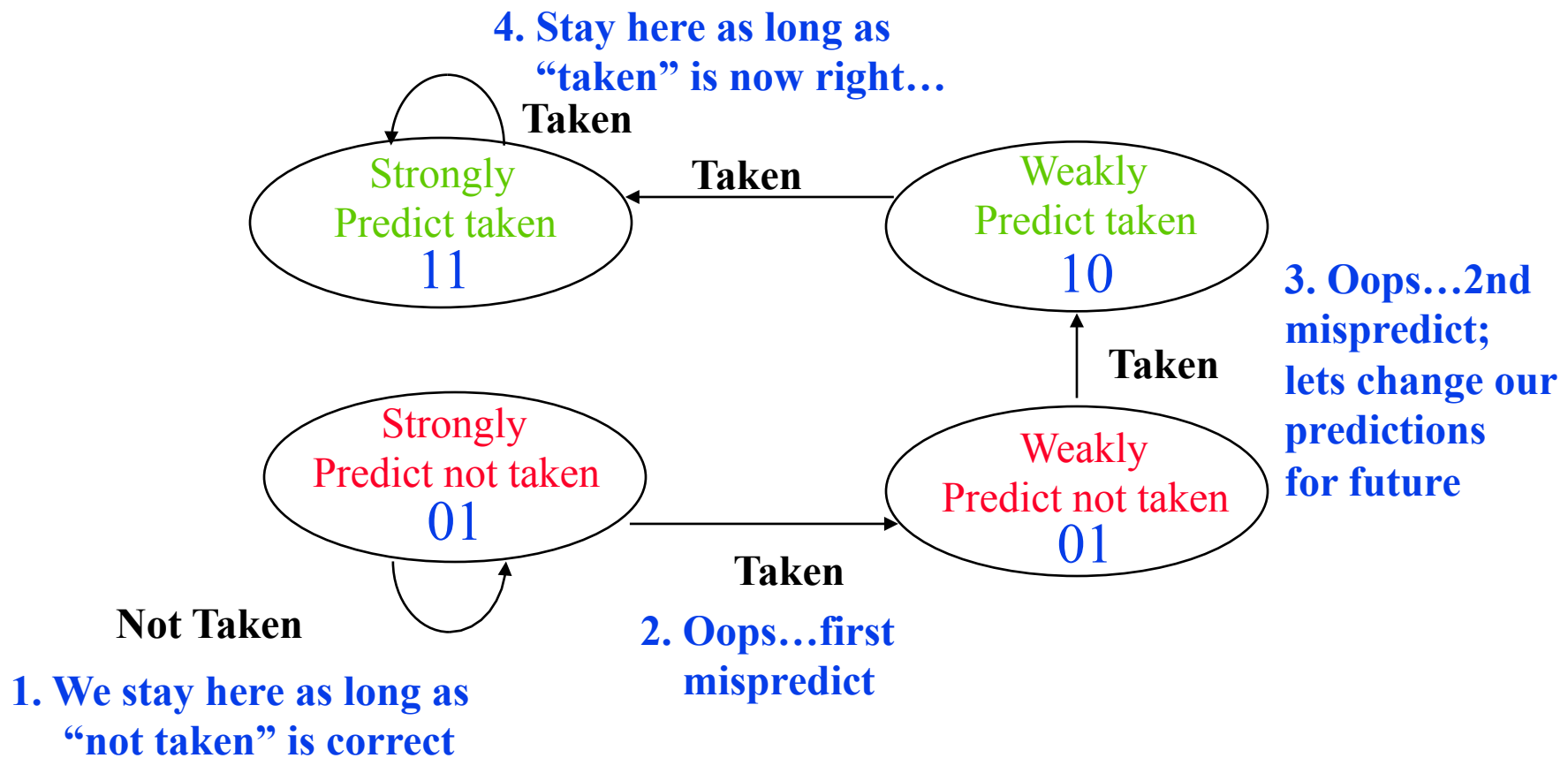
► Nope--we're really wrong.

- ▷ Now, the branch really wants to go the other way, twice in a row.
- ▷ So, Alter the prediction.



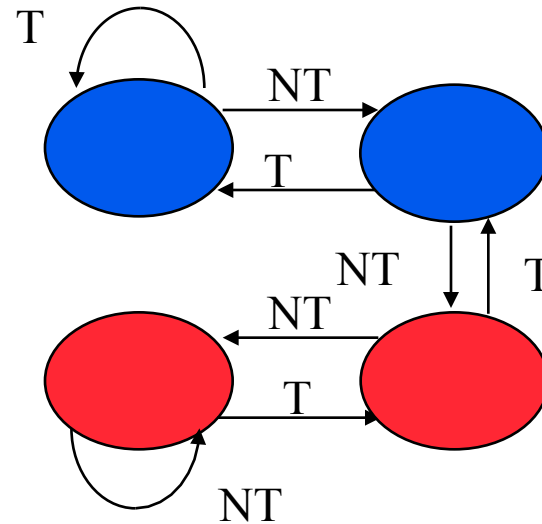
2-bit Branch Prediction: Example

- Consider a few branch predictions in sequence

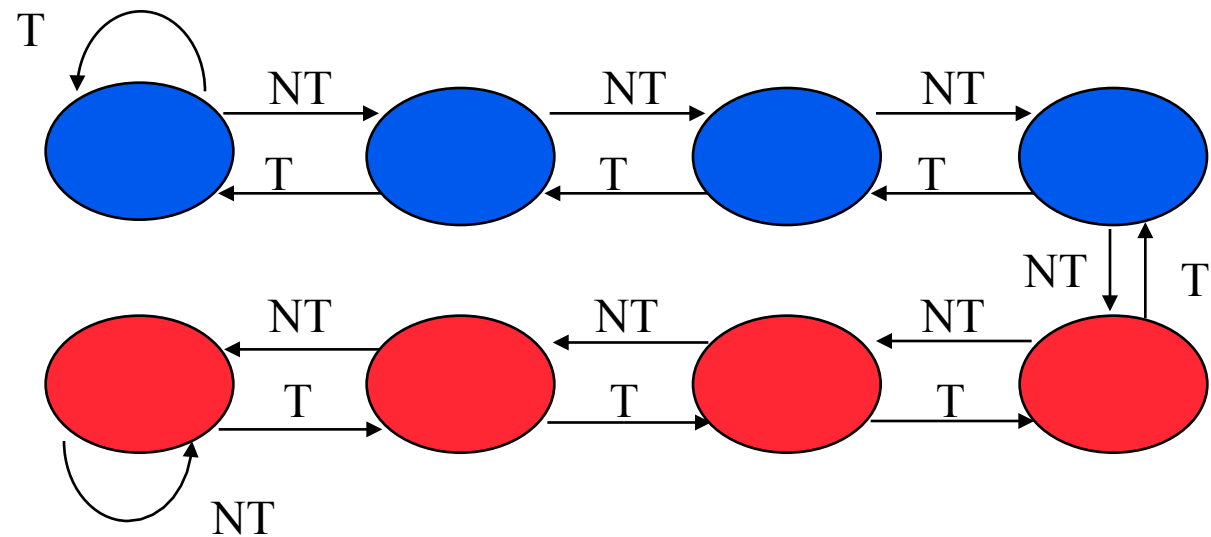


Generalization: 3-bit Prediction

2-bit Prediction

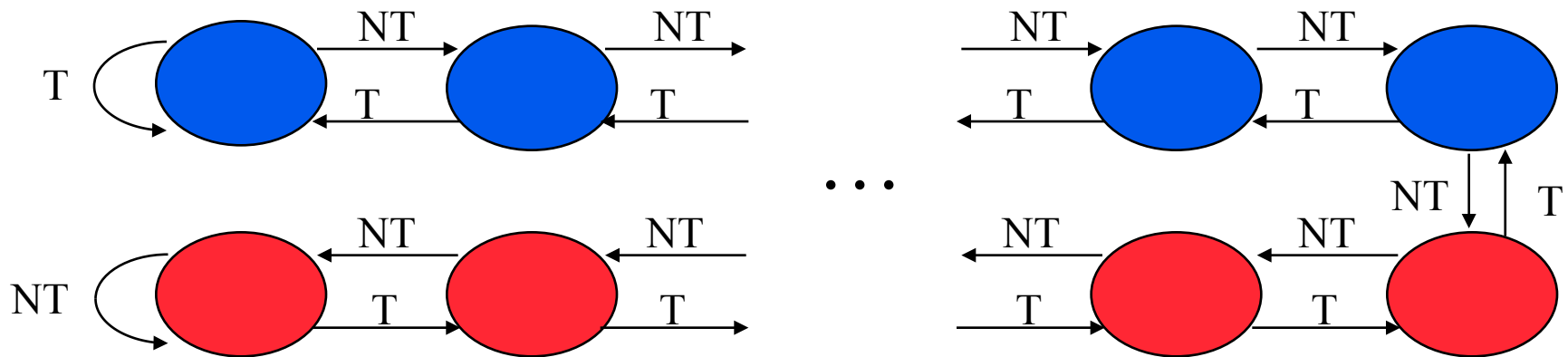


3-bit Prediction



Generalization: N-bit Prediction

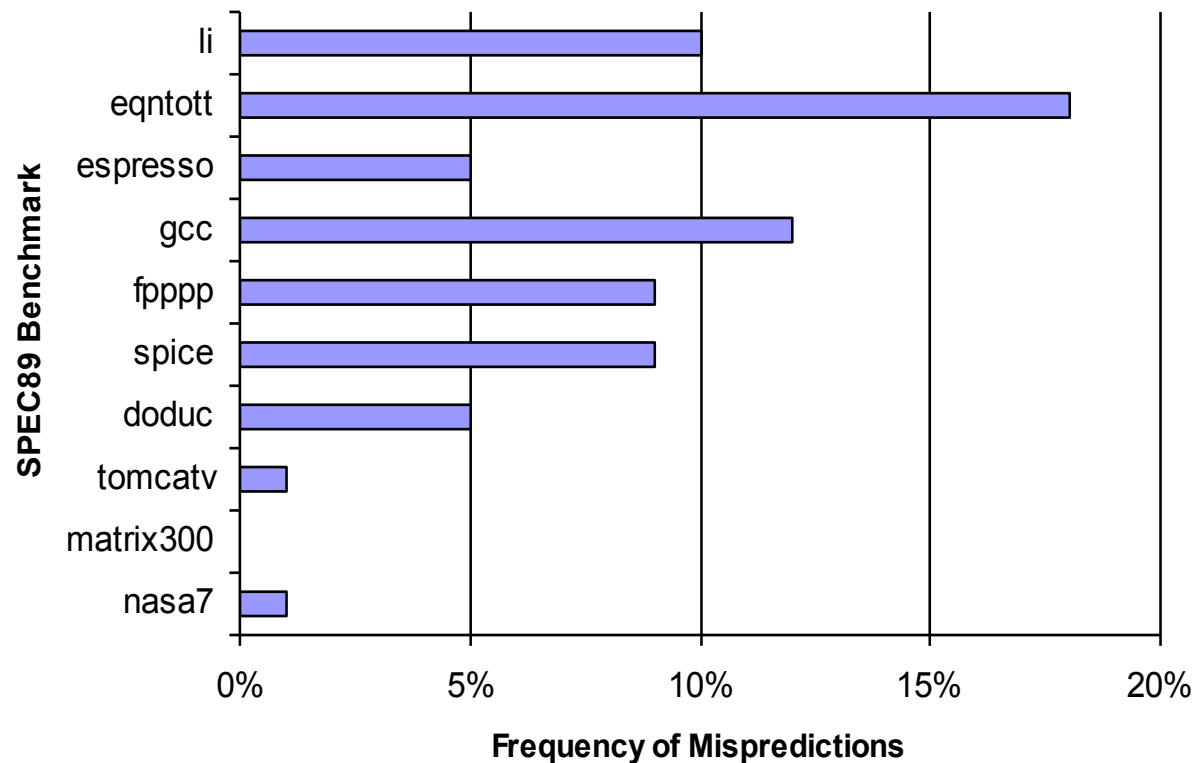
► Saturating N-bit counter



► See whether a majority of the last 2^{N-1} branches were taken or not taken

How well does this work?

► Really good for loop behavior!



4096 entries

2 bits per
entry

When does this break?

- ▶ Doesn't deal with data-dependent branches (not much we can do here)
- ▶ Doesn't deal with correlated behavior:

```
L1:  bne $s1, $0, L2    # B1
      ...
L2:  bne $s2, $0, L3    # B2
      ...
L3:  bne $s1, $s2, L4   # B3
      ...
L4:  ...
```

- ▶ Note that if B1 not-taken, and B2 not-taken, then B3 is not-taken
- ▶ There is a lot of correlated behavior like this in real programs

Small Example:

```
L1:  bne  $s1,  $0, L2    # B1: if (d != 0) goto L2;
      addi $s1,  $0, 1    #      d++;
L2:  subi $s2,  $s1, 1    # L2: d--;
      bne  $s2,  $0, L3    # B2: if (d != 0) goto L3;
      ...
L3:
```

- ▶ If B1 is not taken, B2 will be not-taken
- ▶ If B1 is taken, B2 will most likely be take
- ▶ How does a standard 1-bit predictor work with this?
 - ▷ Assume \$s1 alternates between 2 and 0
 - ▷ When \$1==2, B1 and B2 taken
 - ▷ When \$1==0, B1 and B2 not taken

Small Example:

\$s1 = ?	B1 predict	B1 action	New B1 predict	B2 predict	B2 action	New B2 predict
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

► We ALWAYS mispredict!!!!

Correlating Branch Predictors

- ▶ Idea: keep 2 (or more) predictors
- ▶ One is used/updated if **last branch** was taken (T)
- ▶ One is used/updated if **last branch** was not taken (NT)
- ▶ Each predictor could be N bits (we'll assume one bit)

B1: bne xxxxx
 ...
B2: beq xxxxxxx
 ...

Use this predictor for B2
if last time B1 = NT

Use this predictor for B2
if last time B1 = T

Entry for branch B2:

...	...
Predictor 1	Predictor 2
...	...

Previous Example

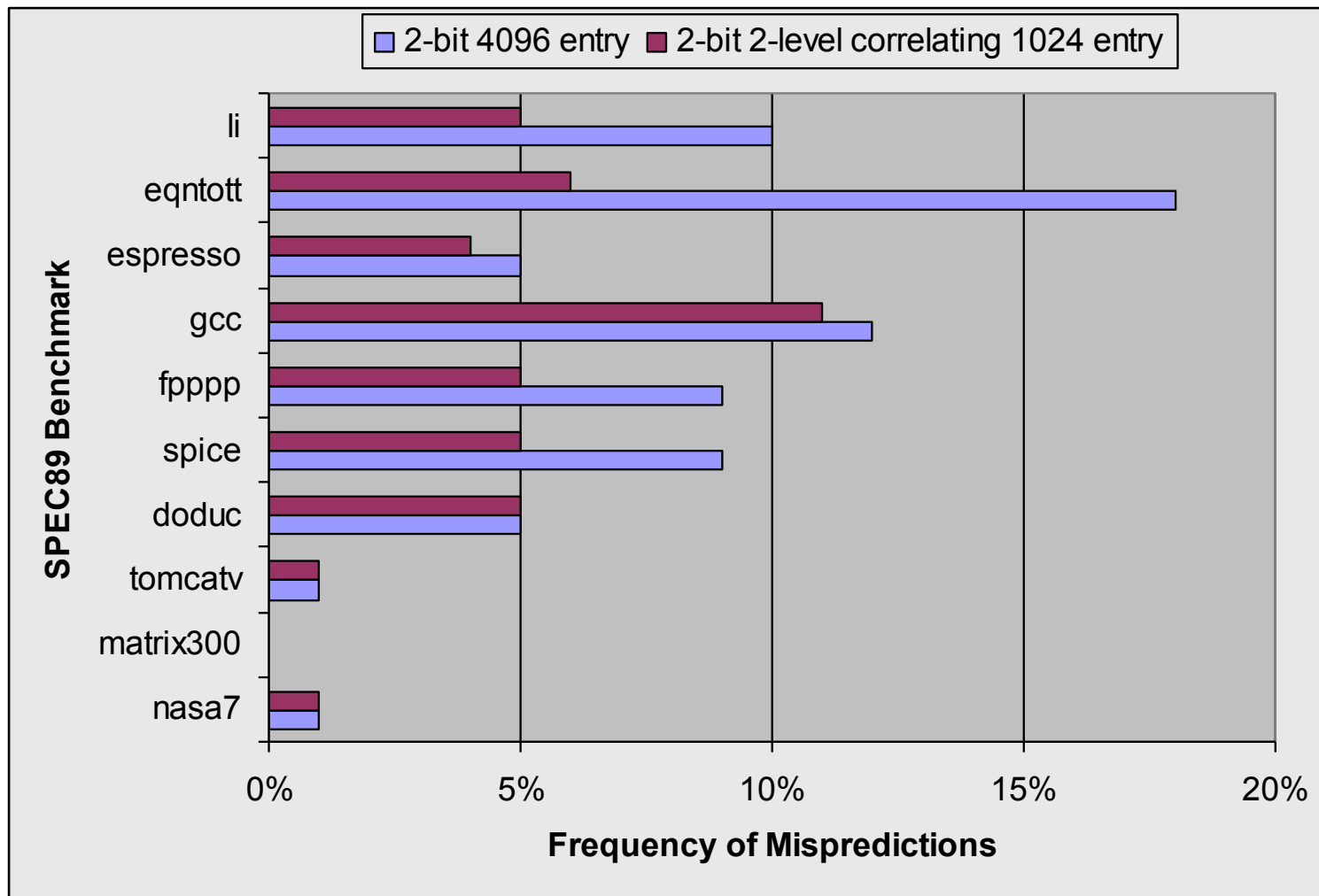
\$s1 = ?	B1 predict	B1 action	New B1 predict	B2 predict	B2 action	New B2 predict
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

Update predictor based on
B1 action

- Initialized to NT/NT
- Only one misprediction of B2!

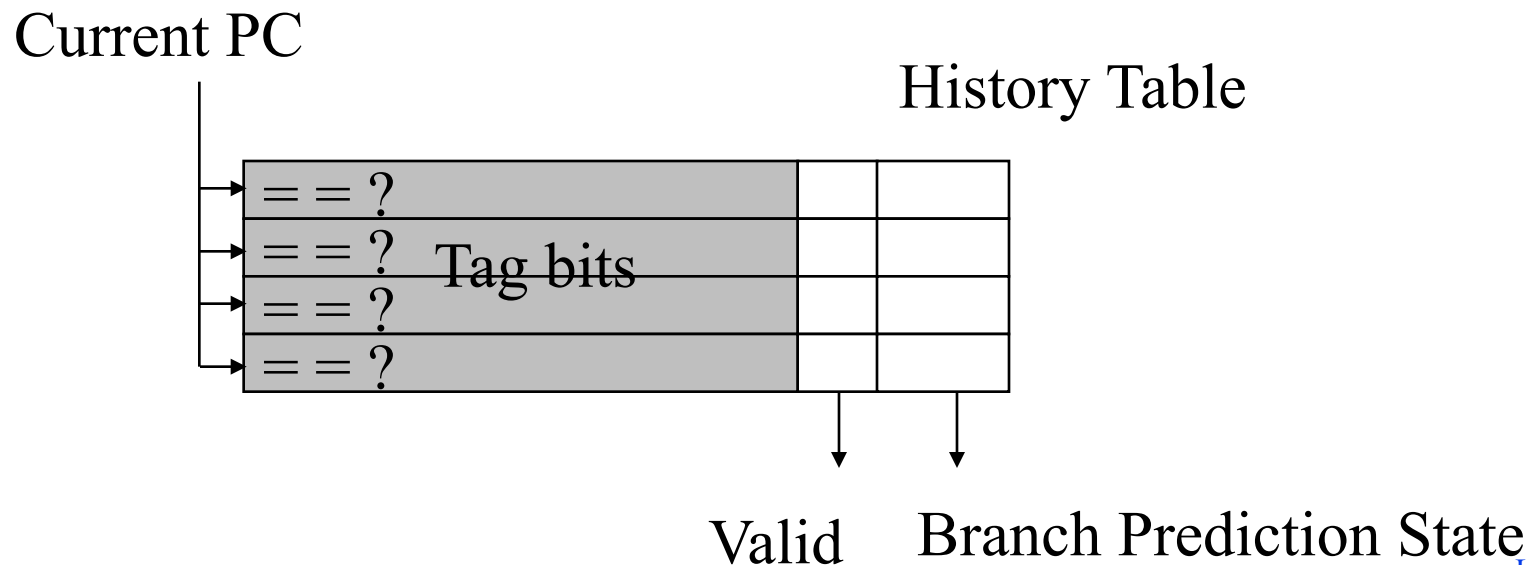
Use different predictor for B2
based on whether B1 was taken
or not.

Performance of Correlating Branch Predictors



How to keep the branch prediction data

- ▶ Keep a table of addresses of branch instructions with the current state of the branch predictor for that branch (history table)
- ▶ A valid field to indicate that this address is a branch
- ▶ Check history table when you fetch instruction
- ▶ Update bits when you know whether the branch is taken or not



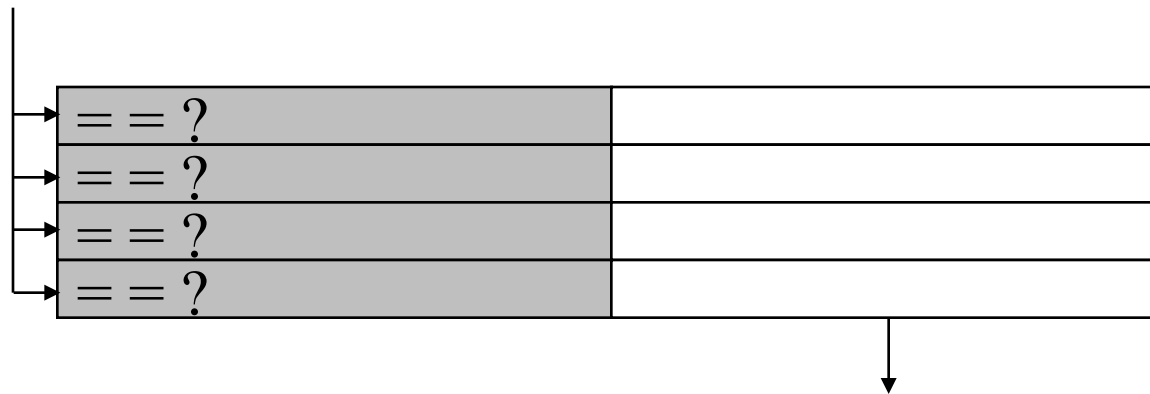
Branch Targets

- ▶ **OK, so we now predict whether we are going to take the branch or not**
- ▶ **Doesn't help if we don't know where a taken branch goes**
- ▶ **MIPS: branch delay slot (BDS)**
 - ▷ solves both the prediction and target address problem
 - ▷ End of ID stage, we know whether we are taken a branch and where we are going
- ▶ **Without the BDS, or with deeper pipelining, this doesn't work**
- ▶ **Fortunately, conditional branches, when taken, go the same place every time!**
 - ▷ Remember the target address in the table
- ▶ **Branch Target Buffer (BTB): Table for branch target**

Branch Target Buffers

- ▶ **A table for branches that are predicted as taken**
 - ▷ Don't have to compute branch targets for not-taken branches
- ▶ **Easy to add to the structure that stores the state of the predictor**
- ▶ **Useful for jumps (we know it is always taken, but we don't know where)**

Current PC



Next PC

What Makes Pipelines Hard to Implement?

- ▶ **Detecting and resolving hazards**
- ▶ **Instruction Set Architecture**
 - ▷ Very complex multicycle instructions are difficult to pipeline
 - ▷ Example:
 - ▷ stringMov from 0x1234, to 0x4000, 0x1000 bytes
- ▶ **Exceptions and Interrupts**

What Makes Pipelines Hard to Implement?

- ▶ Detecting and resolving hazards
- ▶ Instruction Set Architecture
 - ▷ Very complex multicycle instructions are difficult to pipeline
 - ▷ Example:
 - ▷ stringMov from 0x1234, to 0x4000, 0x1000 bytes
- ▶ **Exceptions and Interrupts**

Exceptions and Interrupts

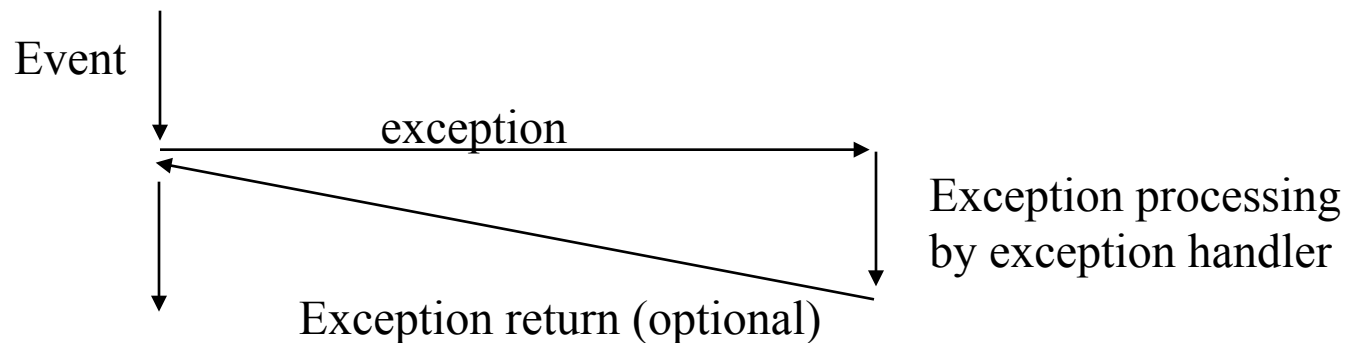
- ▶ **Exceptions are *exceptional* events that *disrupt* the normal flow of a program**
- ▶ **Terminology varies between different machines**
- ▶ **Examples of *Interrupts***
 - ▷ User hitting the keyboard
 - ▷ Disk drive asking for attention
 - ▷ Arrival of a network packet
- ▶ **Examples of *Exceptions***
 - ▷ Divide by zero
 - ▷ Overflow
 - ▷ Page fault

Exception Flow

- ▶ When an exception (or interrupt) occurs, control is transferred to the OS

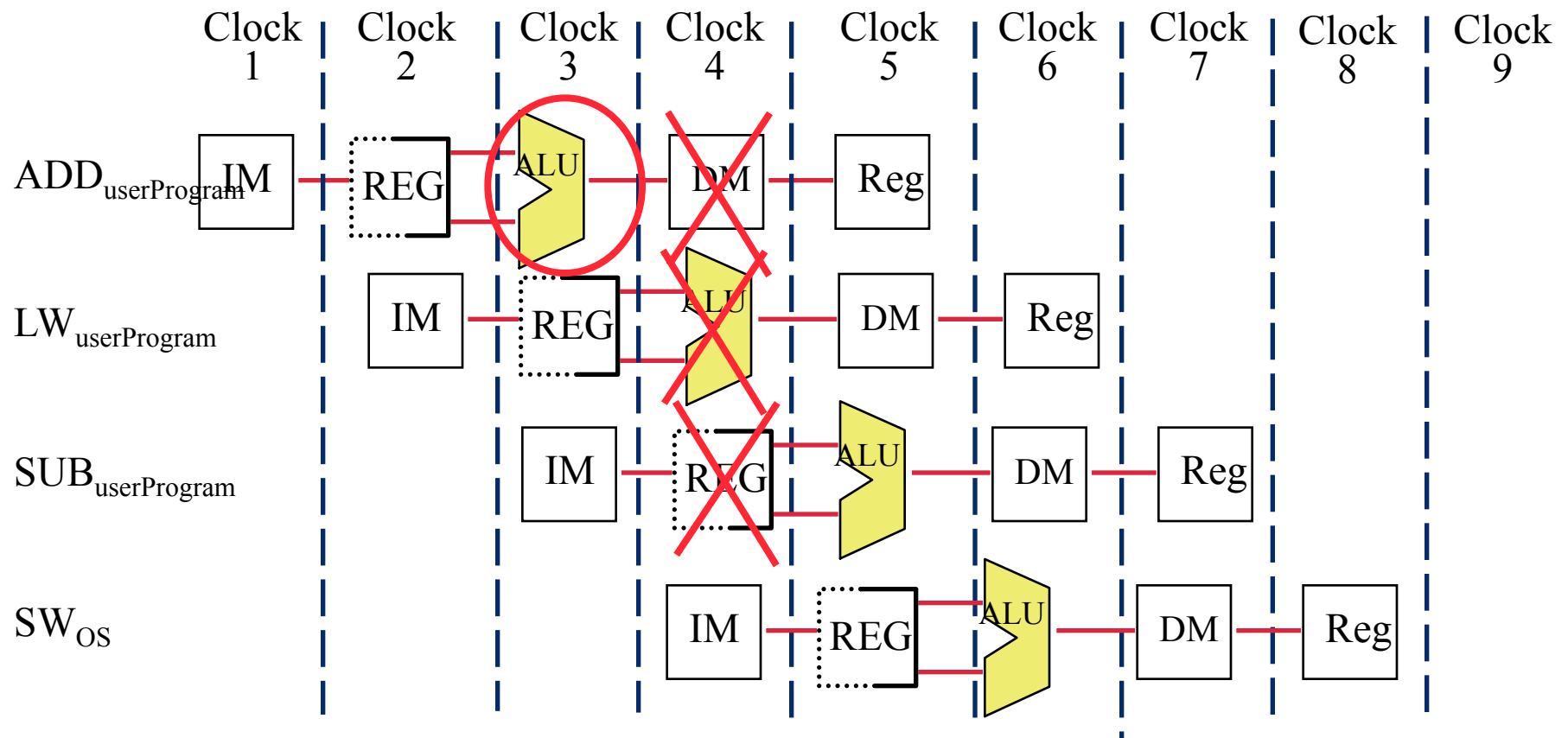
User Process

Operating System



Flow of Instructions During Exception

- Example: Add instruction **overflows** in clock cycle 3



Characterizing Exceptions and Interrupts

▶ Synchronous vs. asynchronous events

- ▷ Synchronous events occur at the same place every time a program executes
- ▷ Asynchronous events are caused by external devices such as a keyboard, disk drive or mouse

▶ User requested vs. coerced

- ▷ If a user asks for it, it is user requested
- ▷ Coerced are hardware events not under user control

▶ User maskable vs. nonmaskable

- ▷ Can a user disable an exception from being detected?

▶ Within vs. between instructions

- ▷ Does the event prevent the current instruction from completing?

▶ Resume vs. terminate

- ▷ Can the event be handled (corrected) or must the program be terminated?

Types of Exceptions

<u>Exception</u>	<u>Syn/Asynch</u>	<u>User request?</u>	<u>User maskable?</u>	<u>Within?</u>	<u>Resume?</u>
I/O device	asynch	coerced	nonmaskable	between	resume
invoke OS	synch	user req.	nonmaskable	between	resume
tracing instr. execution	synch	user req.	user maskable	between	resume
breakpoint	synch	user req.	user maskable	between	resume
int overflow	synch	coerced	user maskable	within	resume
fp overflow	synch	coerced	user maskable	within	resume
page fault	synch	coerced	nonmaskable	within	resume
misaligned mem access	synch	coerced	user maskable	within	resume
mem-protection violation	synch	coerced	nonmaskable	within	term.
undef. instr	synch	coerced	nonmaskable	within	term.
hardware malf.	asynch	coerced	nonmaskable	within	term.

Stopping and Restarting Execution

- ▶ **Exception occurs while many instructions are in flight**
 - ▷ **Ex:** a page fault on a load instruction will occur in stage 4 of the MIPS pipe
 - ▷ Pipeline must be safely shutdown when exception occurs and then **restarted at the offending instruction**
- ▶ **How to handle this? This is done by:**
 - ▷ Force a trap instruction into the pipeline
 - ▷ Until the trap is taken, turn off all writes for the faulting instruction and any instruction that issued after the faulting instruction
 - ▷ This prevents instructions from changing the state of the machine
 - ▷ When the trap is taken, invoking the OS, the OS saves the PC of the offending instruction
 - ▷ The OS fixes the exception (if possible) and then restarts the machine
 - ▷ Restarting usually means setting PC \leftarrow offending instruction address
 - ▷ Replays instruction(s)

Precise vs. Imprecise Exceptions

- ▶ If the pipeline can be stopped so that the instructions issued before the faulting instruction complete, then the pipeline is said to implement **precise exceptions**

- ▷ Gives the illusion that the machine executes one instruction at a time
- ▷ Difficult to do when some instructions take multiple cycles to complete
 - ▷ Some instructions may complete before an exception is detected
 - ▷ Example

Multiply	r1, r2, r3	; multiply takes 10 cycles
Add	r10, r11, r12	; takes 5 cycles

Add will complete before multiply is done. If multiply overflows, then an exception will be raised AFTER the add has updated the value in R10. This is an imprecise exception.

- ▷ Some machines implement both modes: imprecise and precise exceptions
 - ▷ Special software instructions to guarantee precise exceptions
 - ▷ Machine runs slower when one needs precise exceptions

Exceptions and the MIPS Architecture

► Which stage can exceptions occur in?

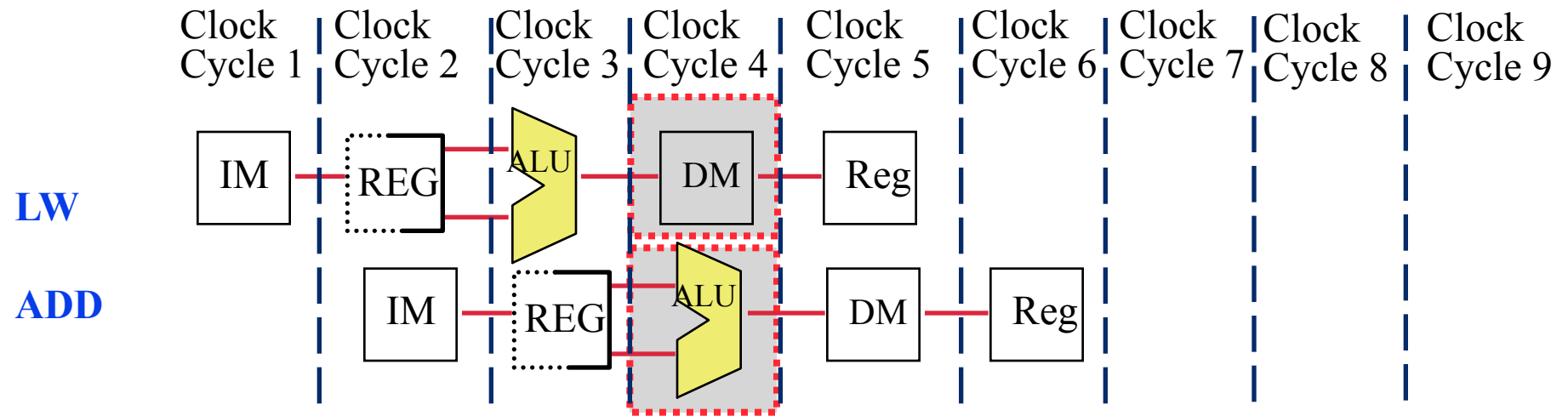
<u>Stage</u>	<u>Problem exceptions occurring</u>
IF	page fault on instruction fetch; misaligned memory access; memory protection violation
ID	undefined or illegal opcode
EX	arithmetic exception
MEM	page fault on data fetch; misaligned memory access; memory-protection violation
WB	none

Multiple Exceptions

► Multiple exceptions can happen in the same cycle

▷ Example

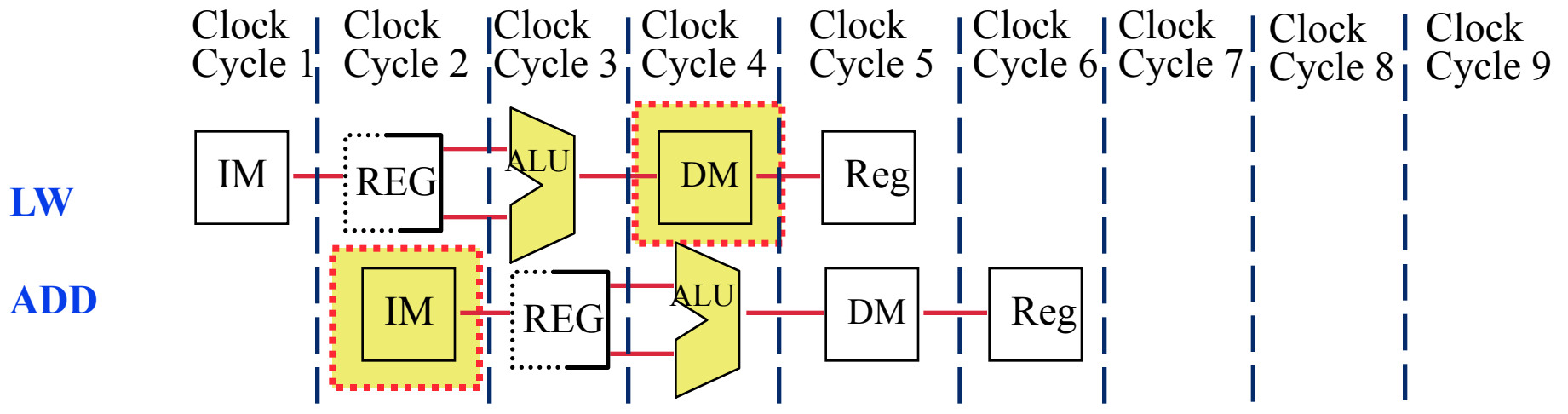
- ▷ In Clock Cycle 4, **LW** can have a data page fault while the **ADD** has an arithmetic exception
- ▷ Handled by servicing the page fault and then restarting the **LW** instruction
 - ▷ The **ADD's** arithmetic exception will occur again because the **ADD** instruction is restarted after the exception is handled



Multiple Exceptions (cont.)

► Multiple exceptions can be difficult to manage

- ▷ Can occur **out-of-order**
- ▷ Example
 - ▷ **ADD** causes an exception in the instruction fetch stage while **LW** causes an exception in the memory access stage
 - ▷ If we implement precise exceptions, **LW** exception must be handled first
 - ▷ This is done by having hardware post exceptions by order of instruction



About Exceptions

► One of the single messiest parts of designing a modern CPU

- ▷ It isn't pretty, it's easy to get wrong
- ▷ It's often not too elegant
- ▷ It usually takes huge wads of special logic
- ▷ It causes architects to age prematurely

► Further complicated by modern CPU mechanisms

- ▷ Deep pipelines
- ▷ Superscalar --lots of instructions in flight in parallel
- ▷ Out-of-order execution -- time order of exceptions != program order of the instructions on which the exceptions happened
- ▷ Maintaining illusion of “sequential instruction execution” gets really complicated

Summary

- ▶ **Super-pipelining is not always good; the optimal number of stages is relatively small**
- ▶ **Deep pipelines hurt more by control and data hazards**
- ▶ **Compiler may resolve data hazards and fill branch delay stalls**
 - ▷ Loop unrolling, code reordering
- ▶ **But, compiler is restricted. Better to resolve branches early with Branch Prediction**
 - ▷ Static branch prediction
 - ▷ Dynamic branch prediction
 - ▷ N-bit predictors, correlating predictors
 - ▷ Squash instructions if prediction incorrect
- ▶ **Exceptions and Interrupts**
 - ▷ Precise vs. Imprecise
 - ▷ Multiple exceptions can be difficult to manage