Lecture 13-14: Pipelines Hazards

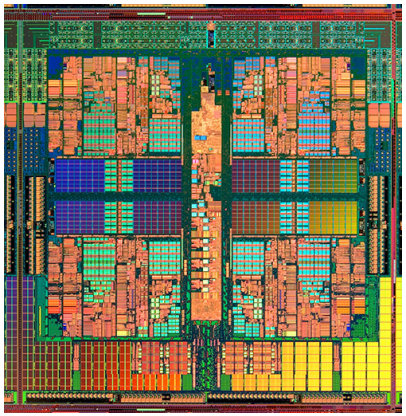
Suggested reading:

(HP Chapter 4.5—4.7)

CSE 30321

Processor components

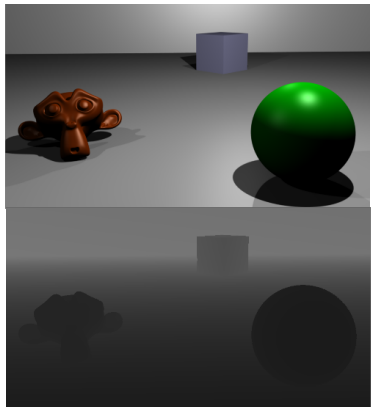
Multicore processors and programming



Processor comparison

AMD Athlon 64 vs. intel Pentium Dual-Core inside

Writing more efficient code



The right HW for the right application

HLL code translation

```
for i=0; i<5; i++ {
        a = (a*b) + c;
}
```

```
MULT r1,r2,r3    # r1 ← r2*r3
ADD r2,r1,r4     # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

Fundamental lesson(s)

- **Pipelining changes the timing as to when the result(s) of an instruction are produced**

- **Additional HW is needed to ensure that the correct program results are produced while maintaining the speedups offered from the introduction of pipelining**

- **We must also account for the efficient pipelining of control instructions (e.g. beq) to preserve performance gains *and* program correctness**

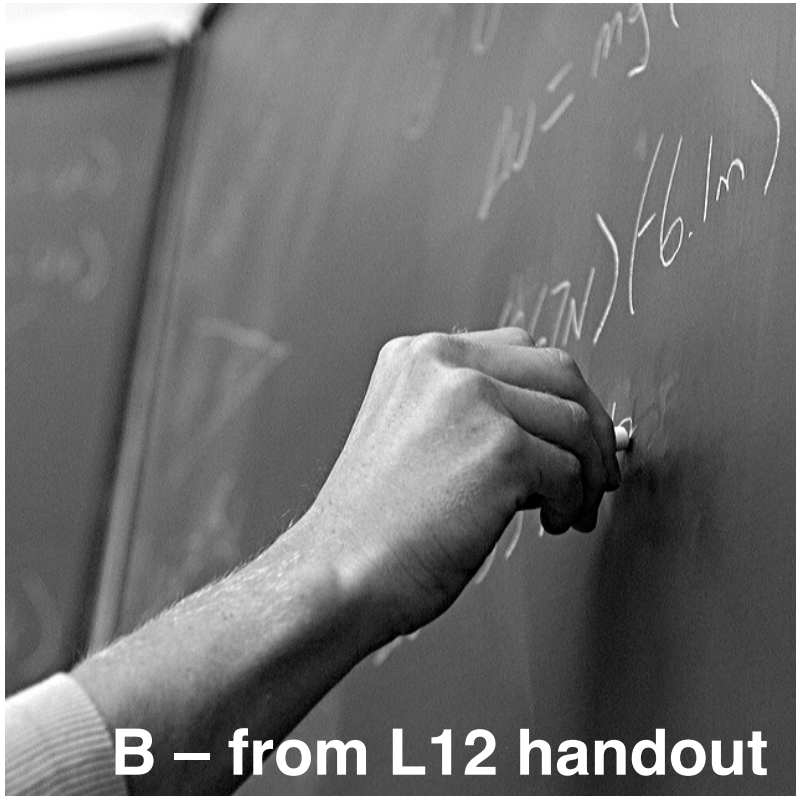
Why it’s important…

- If you're a hardware designer OR a complier writer, you need to be aware of how HLL is mapped to assembly instructions AND what HW is used to execute a sequence of assembly instructions

- Otherwise, it is quite possible to always have built in inefficiencies

On the board…

- Let's look at hazards…
  - …and how they (generally) impact performance.



Pipelining hazards

- **Pipeline hazards prevent next instruction from executing during designated clock cycle**

- **There are 3 classes of hazards:**
  - **Structural Hazards:**
    - **Arise from resource conflicts**
    - **HW cannot support all possible combinations of instructions**
  - **Data Hazards:**
    - **Occur when given instruction depends on data from an instruction ahead of it in pipeline**
  - **Control Hazards:**
    - **Result from branch, other instructions that change flow of program (i.e. change PC)**

How do we deal with hazards?

- Often, pipeline must be stalled
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.

- A note on terminology:
  - If we say an instruction was "issued later than instruction x", we mean that it was issued after instruction x and is not as far along in the pipeline
  - If we say an instruction was "issued earlier than instruction x", we mean that it was issued before instruction x and is further along in the pipeline

Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle

- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…

- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction

Stalls and performance

- **Ignoring overhead and assuming stages are balanced:**

$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

- **If no stalls, speedup equal to # of pipeline stages in ideal case**

STRUCTURAL HAZARDS

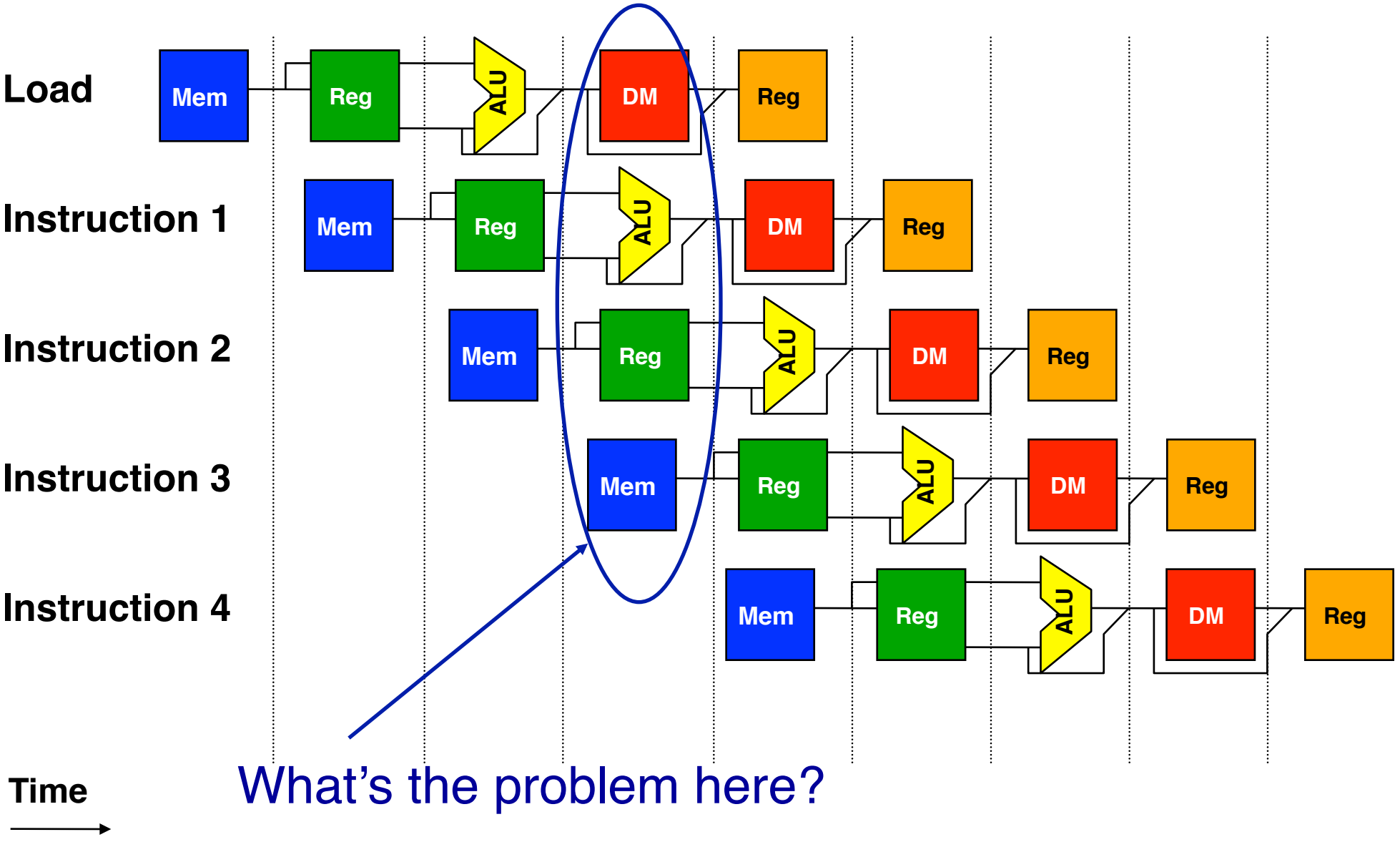
Structural hazards

- Avoid structural hazards by duplicating resources
  - e.g. an ALU to perform an arithmetic operation and an adder to increment PC

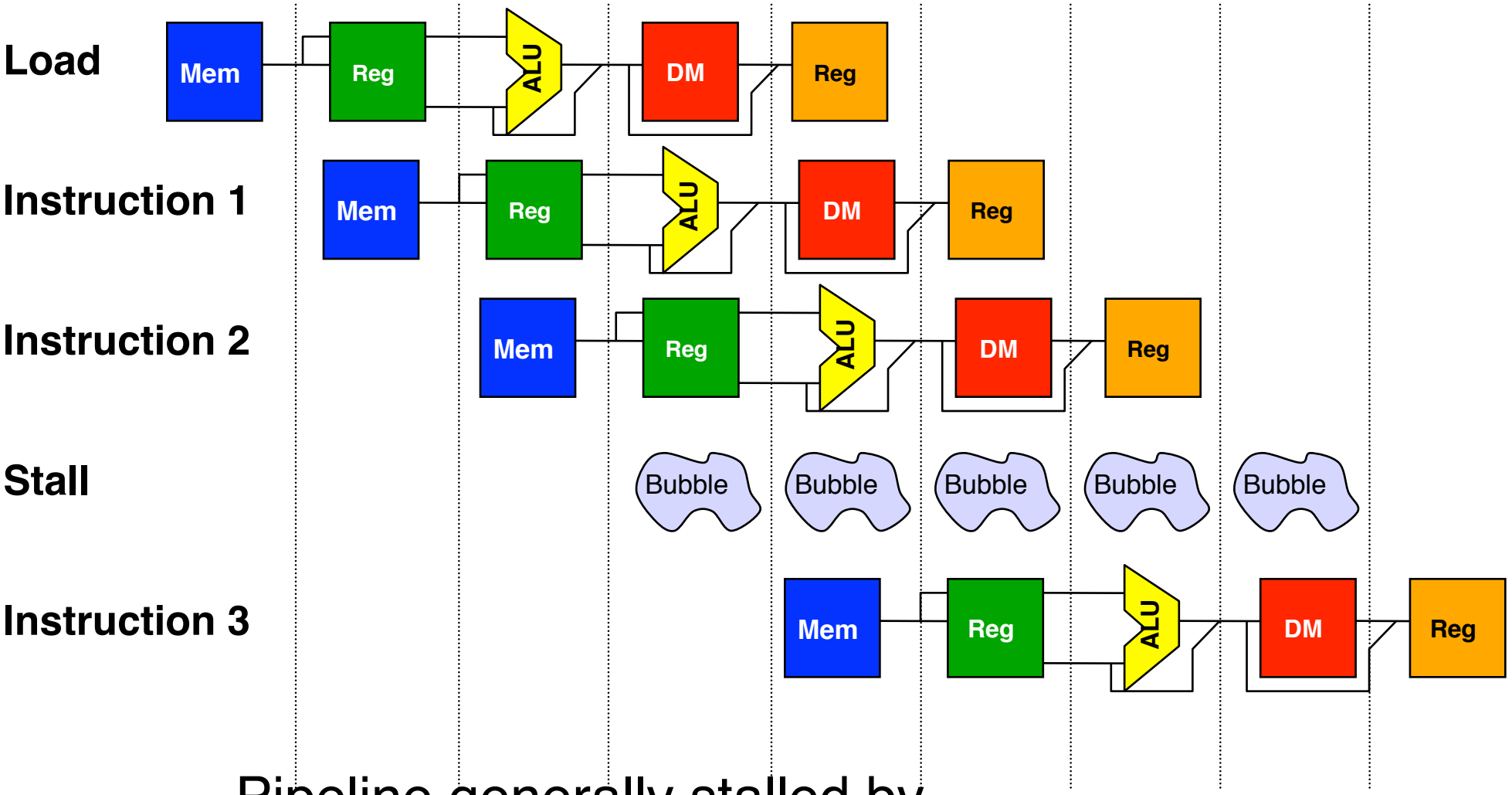
- If not all possible combinations of instructions can be executed, structural hazards occur

- Pipelines stall result of hazards, CPI increased from the usual “1”

An example of a structural hazard



How is it resolved?



Pipeline generally stalled by inserting a “bubble” or NOP

Or alternatively…

Clock Number →

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction “steals” an instruction fetch cycle which will cause the pipeline to stall.
- Thus, no instruction completes on clock cycle 8

What's the realistic solution?

- **Answer: Add more hardware.**
  - **(especially for the memory access example – i.e. the common case)**
    - **CPI degrades quickly from our ideal '1' for even the simplest of cases…**

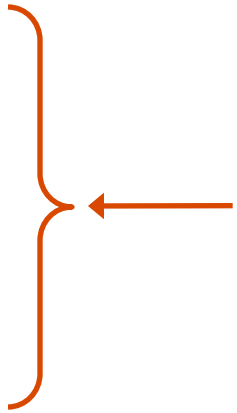
DATA HAZARDS

Data hazards

- These exist because of pipelining

- Why do they exist???
  - Pipelining changes when data operands are read, written
  - Order differs from order seen by sequentially executing instructions on un-pipelined machine

- Consider this example:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

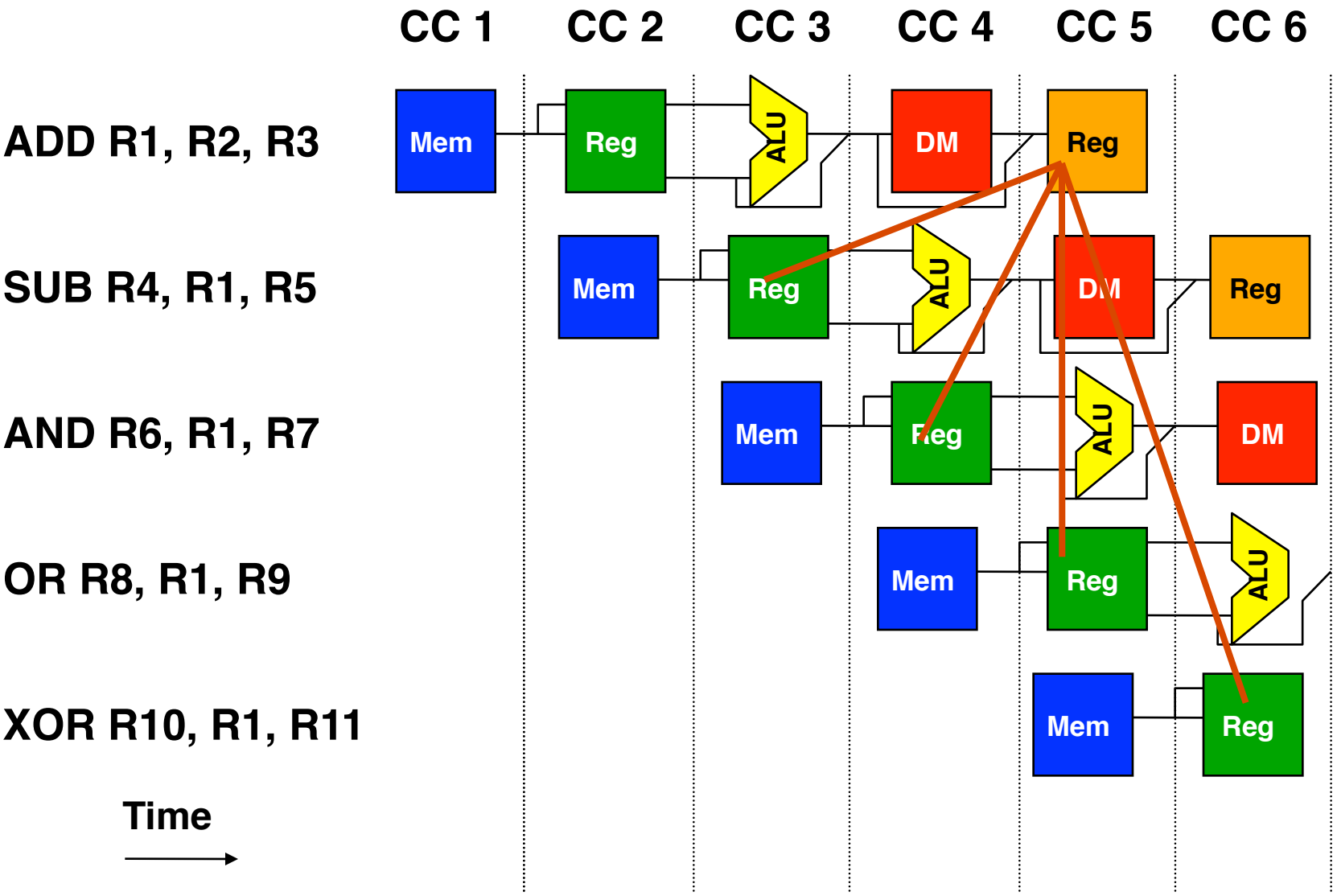


All instructions after ADD use result of ADD

ADD writes the register in WB but SUB needs it in ID.

This is a data hazard

Illustrating a data hazard

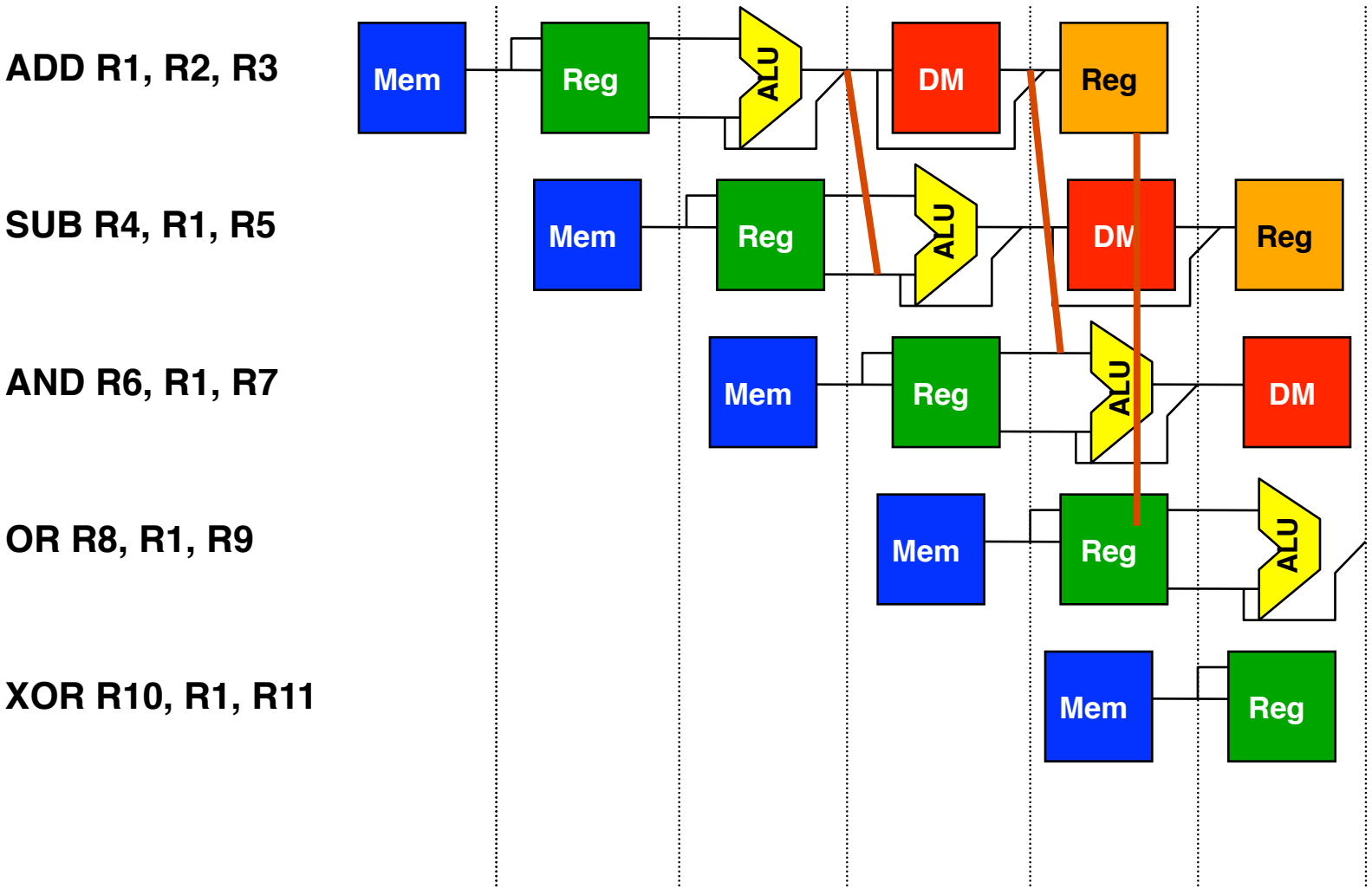


ADD instruction causes a hazard in next 3 instructions b/c register not written until after those 3 read it.

Forwarding

- Problem illustrated on previous slide can actually be solved relatively easily – with forwarding
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes!
- Generally speaking:
  - Forwarding occurs when a result is passed directly to functional unit that requires it.
  - Result goes from output of one unit to input of another

When can we forward?



SUB gets info. from EX/MEM pipe register

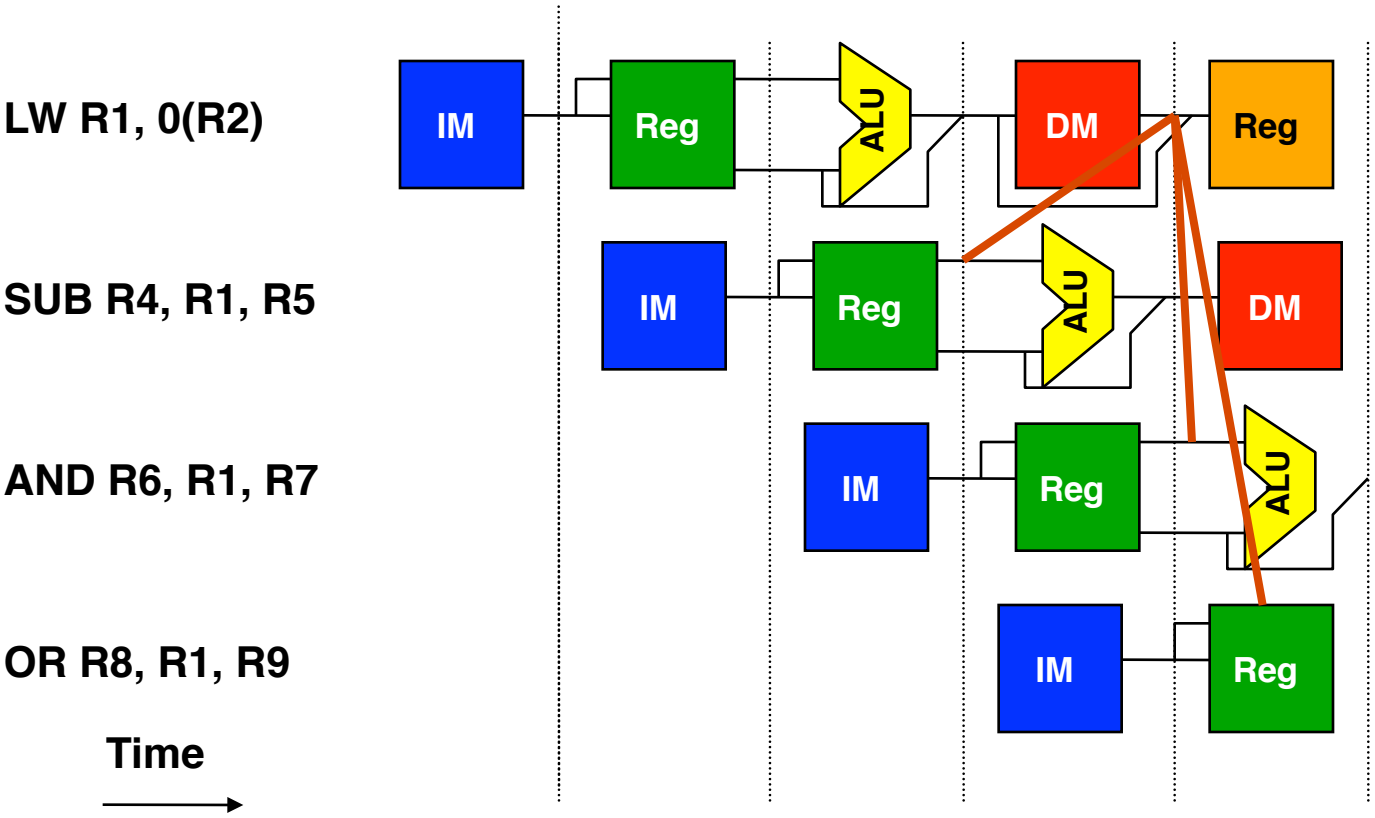
AND gets info. from MEM/WB pipe register

OR gets info. by forwarding from register file

Rule of thumb:

If line goes "forward" you can do forwarding.
If its drawn backward, it's physically impossible.

Forwarding doesn't always work

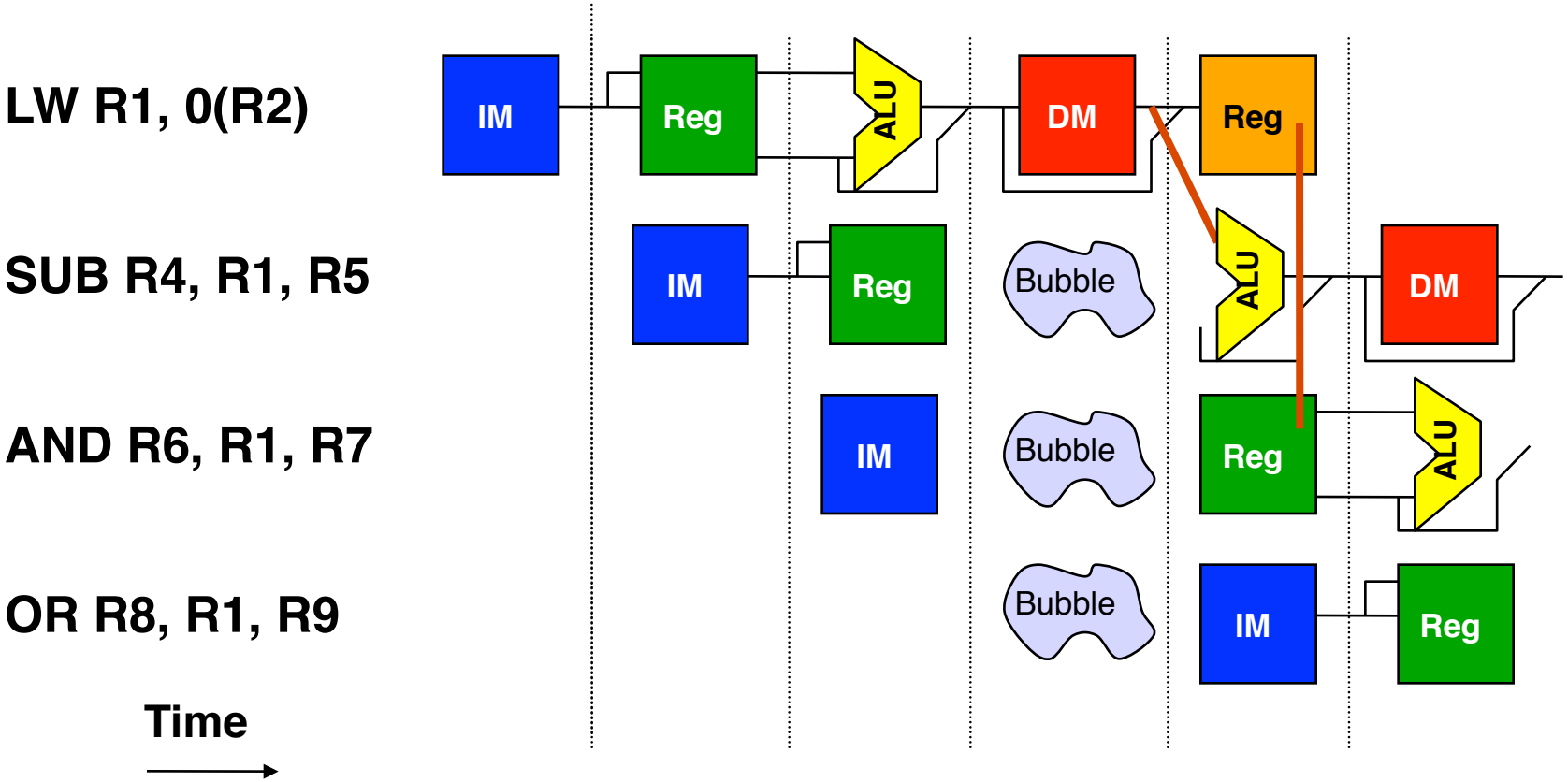


Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

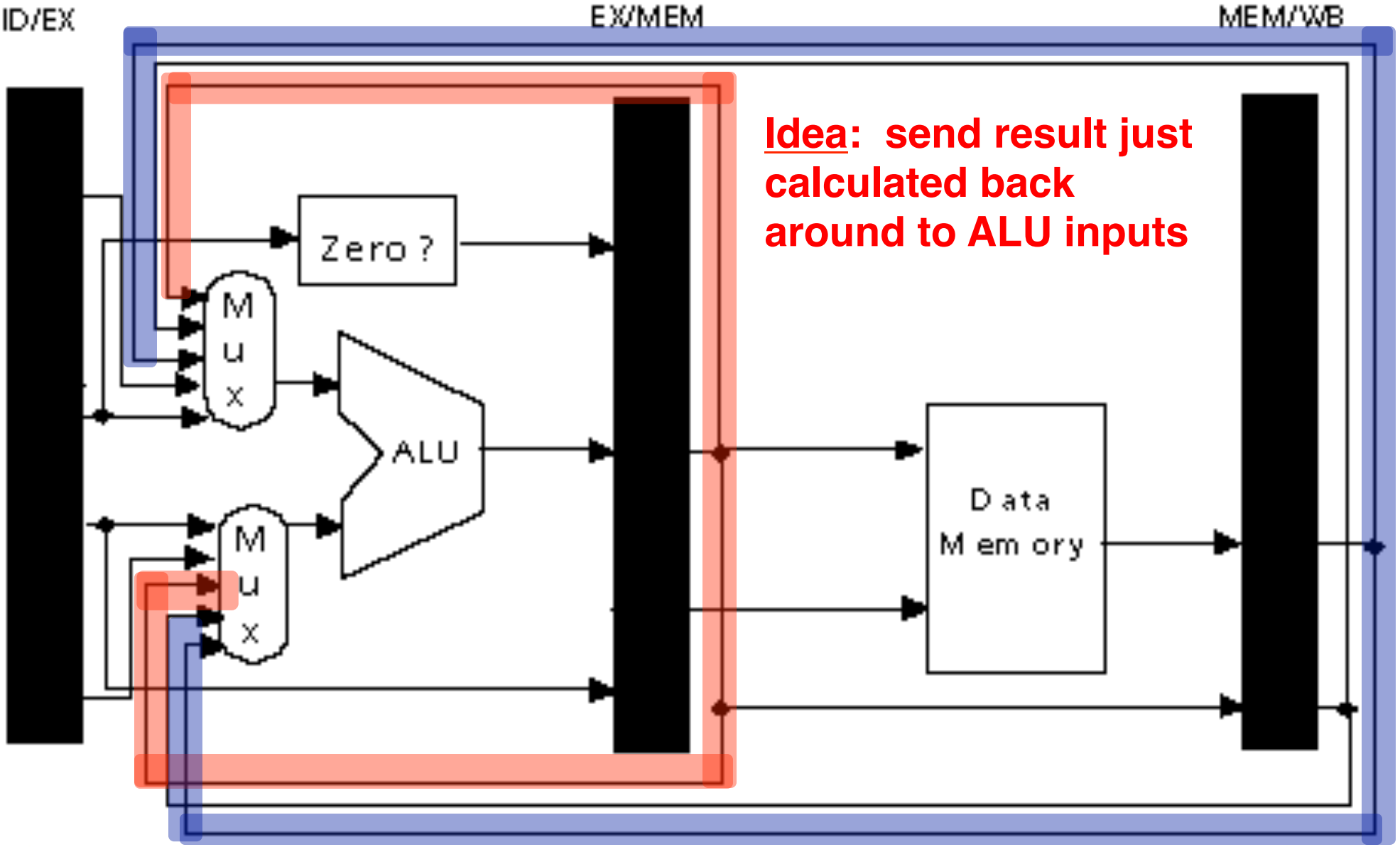
Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

The solution pictorially



Insertion of bubble causes # of cycles to complete this sequence to grow by 1

HW Change for Forwarding



Send output of memory back to ALU too…

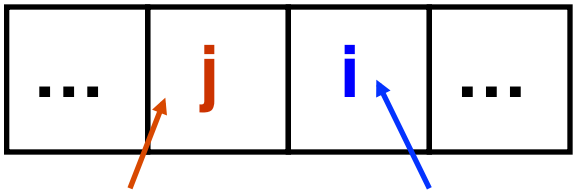
Data hazard specifics

- There are actually 3 different kinds of data hazards:
  - Read After Write (RAW)
  - Write After Write (WAW)
  - Write After Read (WAR)

- With an in-order issue/in-order completion machine, we're not as concerned with WAW, WAR

Read after write (RAW) hazards

- With RAW hazard, instruction j tries to read a source operand before instruction i writes it.
- Thus, j would incorrectly receive old or incorrect value
- Graphically/Example:



Instruction j is a read instruction issued after i

Instruction i is a write instruction issued before j

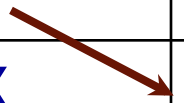
i: ADD R1, R2, R3
j: SUB R4, R1, R6

- Can use stalling or forwarding to resolve this hazard

Memory Data Hazards

- Seen register hazards, can also have memory hazards
  - RAW:
    - store R1, 0(SP)
    - load R4, 0(SP)

	1	2	3	4	5	6
Store R1, 0(SP)	F	D	EX	M	WB	
Load R1, 0(SP)		F	D	EX	M	WB



  - In simple pipeline, memory hazards are easy
    - In order, one at a time, read & write in same stage
  - In general though, more difficult than register hazards

Data hazards and the compiler

- **Compiler should be able to help eliminate some stalls caused by data hazards**

- **i.e. compiler could not generate a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register.**

What about control logic?

- For MIPS integer pipeline, all data hazards can be checked during ID phase of pipeline
- If data hazard, instruction stalled before its issued
- Whether forwarding is needed can also be determined at this stage, controls signals set
- If hazard detected, control unit of pipeline must stall pipeline and prevent instructions in IF, ID from advancing

Some example situations

Situation	Example	Action

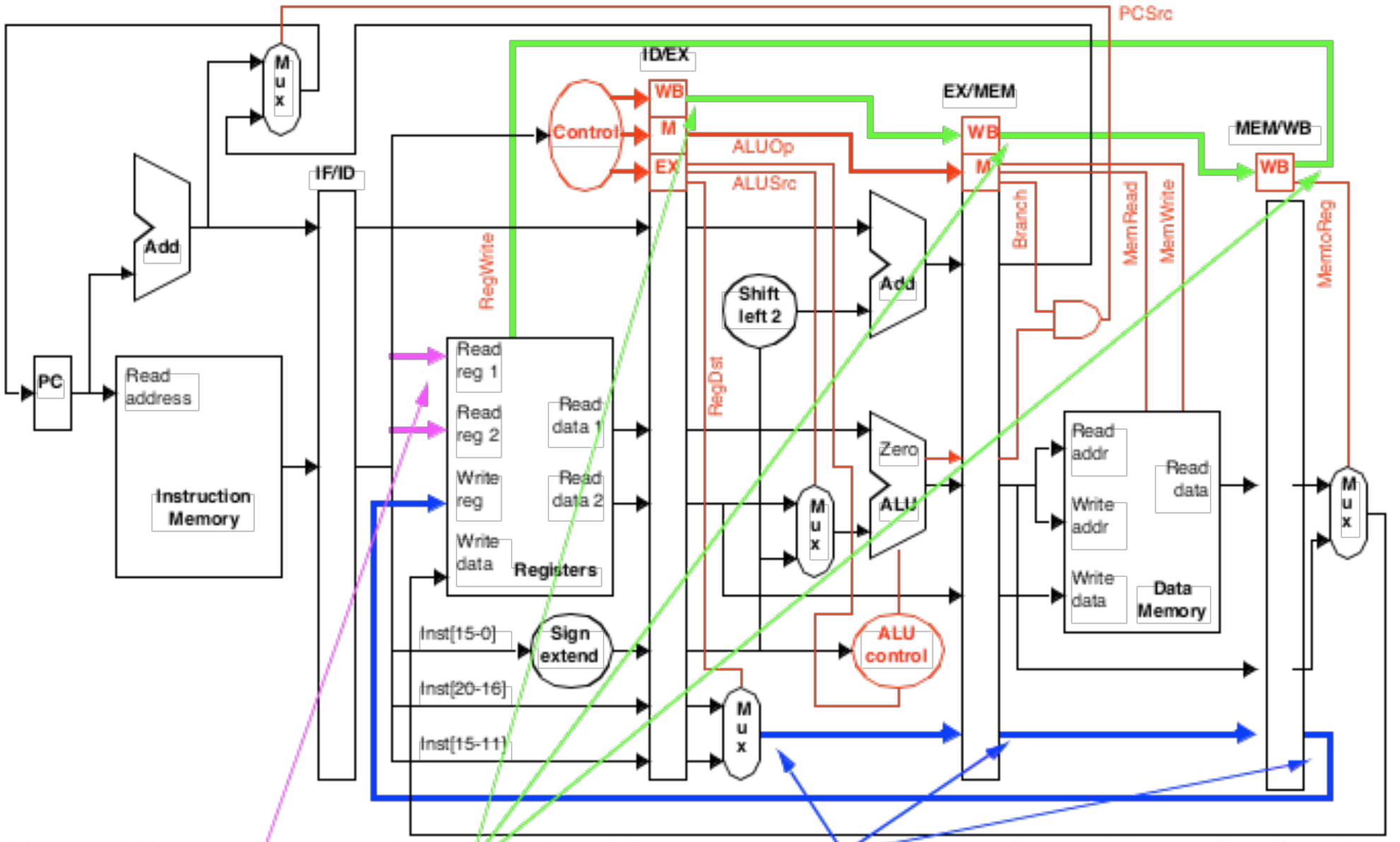
No Dependence	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	LW R1, 45(R2)
	ADD R5, R1, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R1, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX

Dependence with accesses in order	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R1, R7
		No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Detecting Data Hazards



Hazard **if a current register read address = any register write address in pipeline and RegWrite is asserted in that pipeline stage**

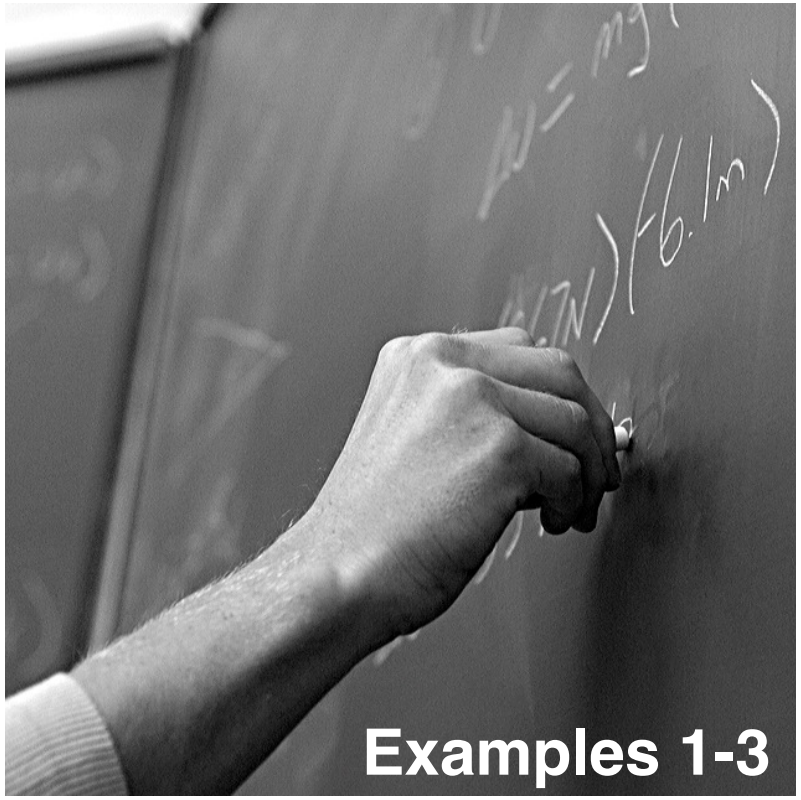
Hazards vs. Dependencies

- dependence: fixed property of instruction stream
  - (i.e., program)

- hazard: property of program and processor organization
  - implies potential for executing things in wrong order
    - potential only exists if instructions can be simultaneously "in-flight"
    - property of dynamic distance between instructions vs. pipeline depth

- For example, can have RAW dependence with or without hazard
  - depends on pipeline

Examples…



CONTROL HAZARDS

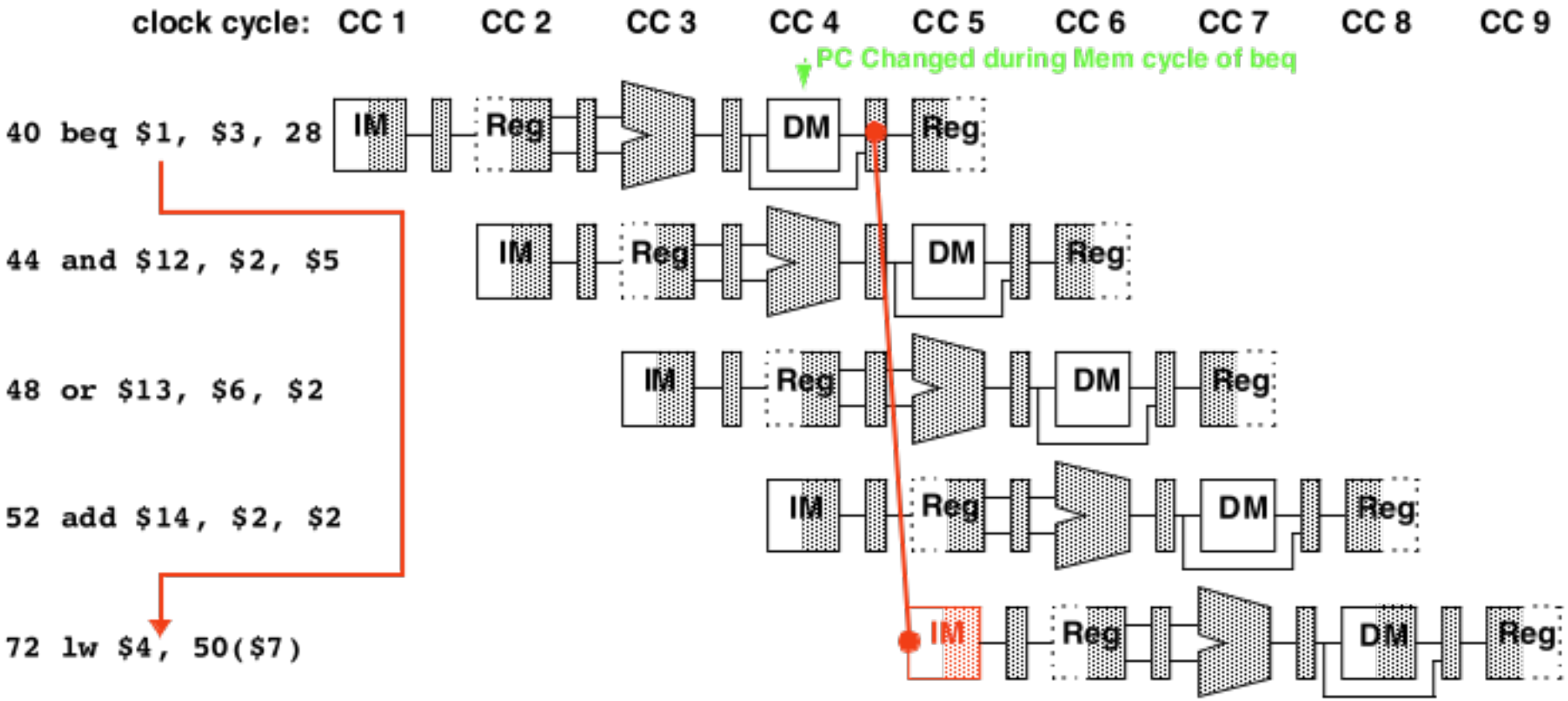
Branch / Control Hazards

- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers

- Also need to consider hazards involving branches:
  - Example:
    - 40: beq $1, $3, 28 # (28 leads to address 72)
    - 44: and $12, $2, $5
    - 48: or $13, $6, $2
    - 52: add $14, $2, $2
    - 72: lw $4, 50($7)

- How long will it take before the branch decision takes effect?
  - What happens in the meantime?

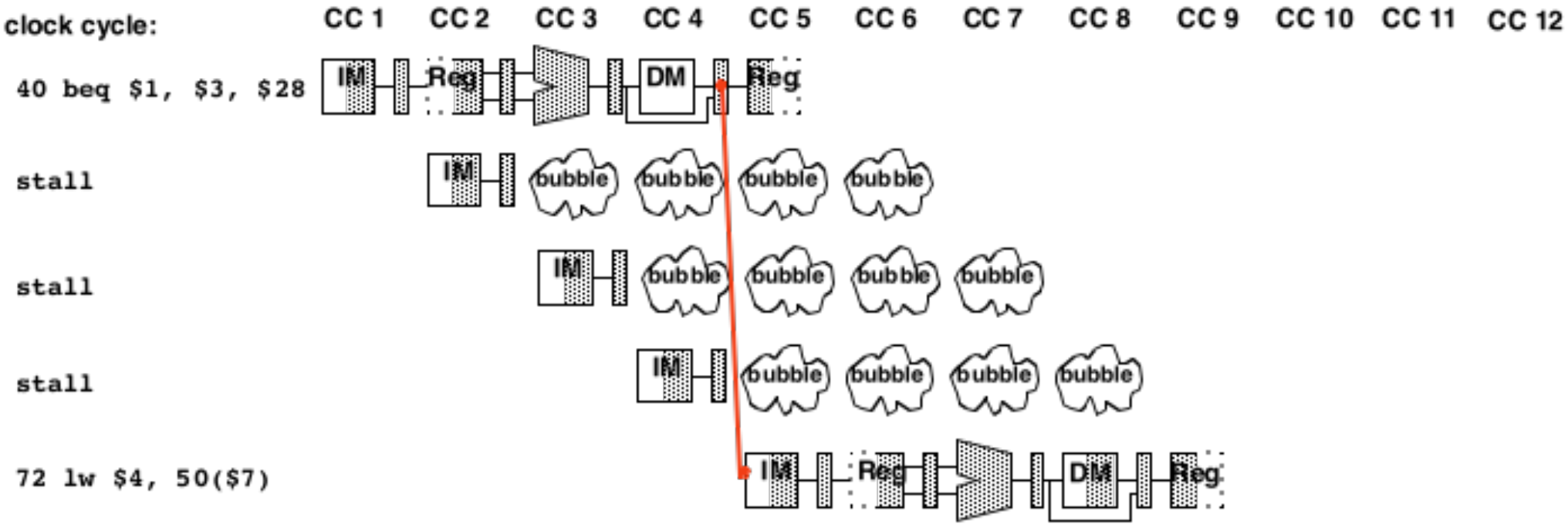
How branches impact pipelined instructions



- If branch condition true, must skip 44, 48, 52
  - But, these have already started down the pipeline
  - They will complete unless we do something about it
- How do we deal with this?
  - We'll consider 2 possibilities

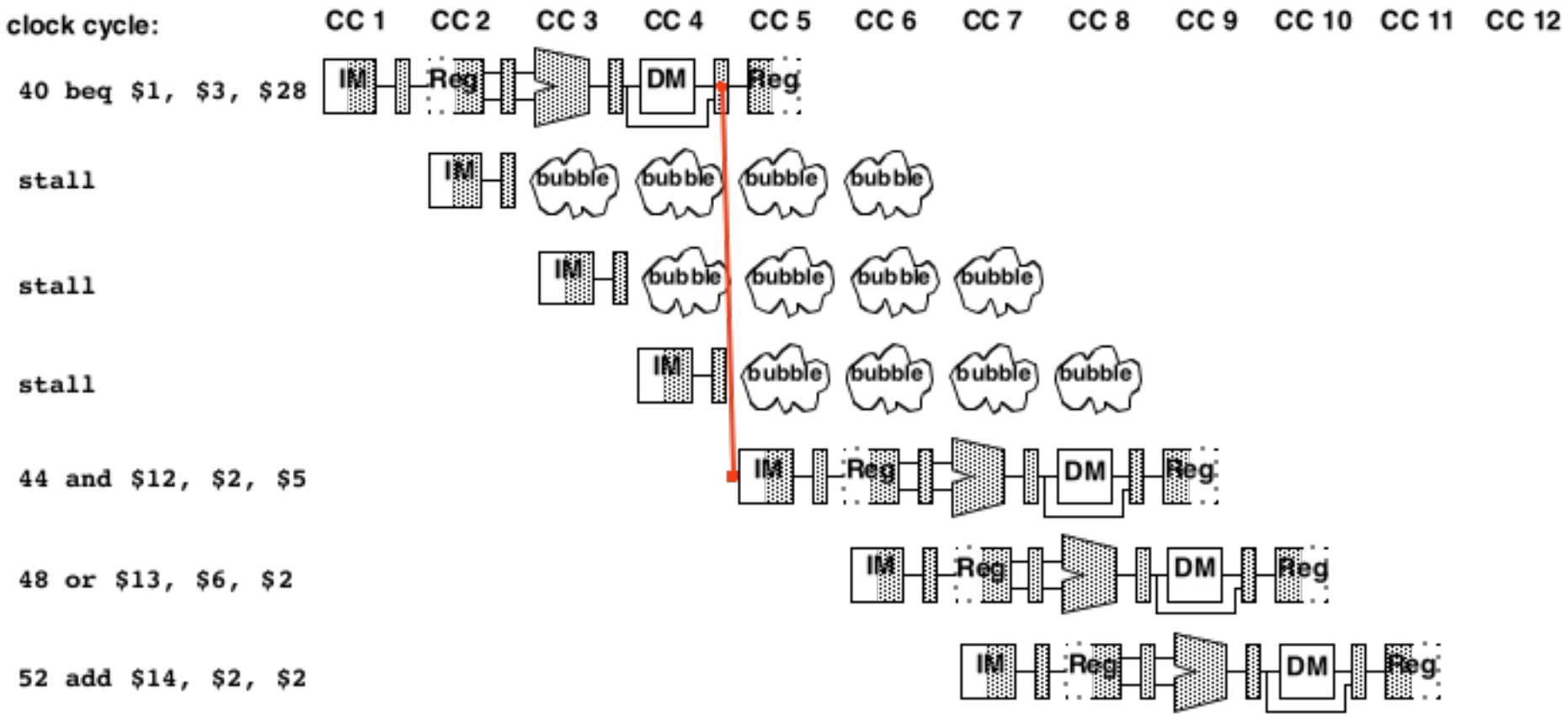
Dealing w/branch hazards: always stall

- **Branch taken**
  - **Wait 3 cycles**
  - **No proper instructions in the pipeline**
  - **Same delay as without stalls (no time lost)**



Dealing w/branch hazards: always stall

- Branch not taken
  - Still must wait 3 cycles
  - Time lost
  - Could have spent CCs fetching, decoding next instructions



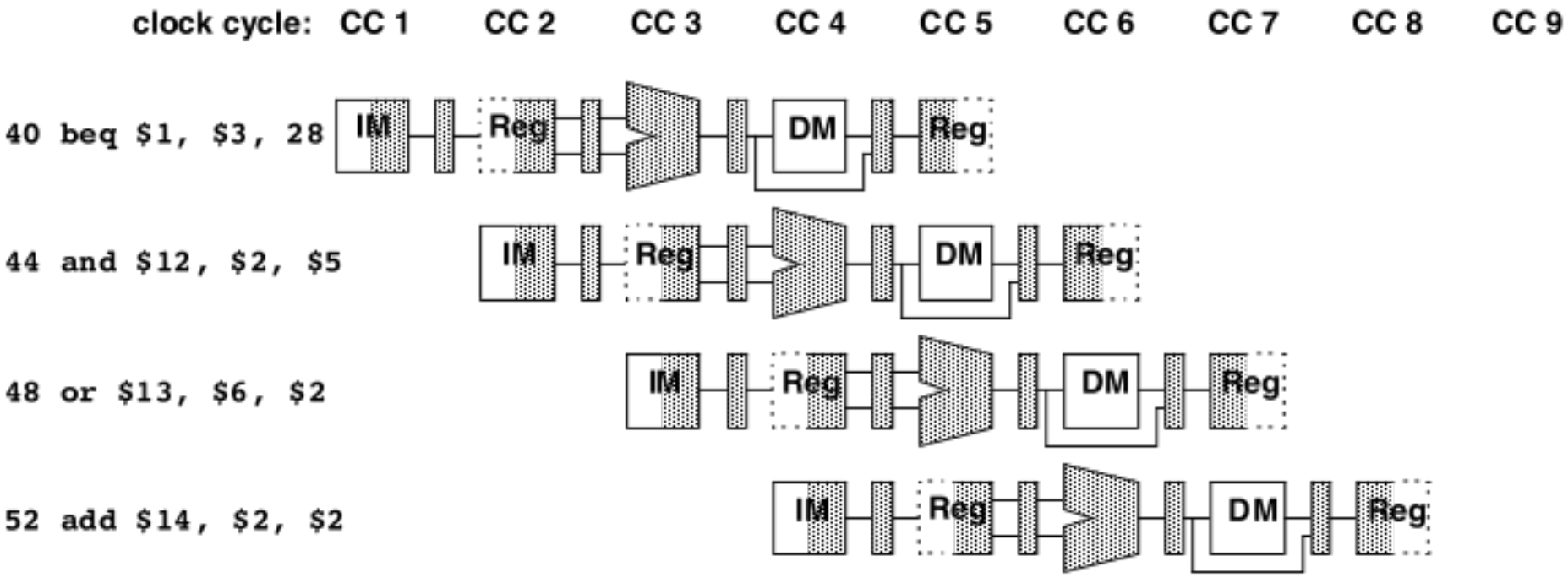
Dealing w/branch hazards

- On average, branches are taken ½ the time
  - If branch not taken…
    - Continue normal processing
  - Else, if branch is taken…
    - Need to flush improper instruction from pipeline

- One approach:
  - Always assume branch will NOT be taken
    - Cuts overall time for branch processing in ½
  - If prediction is incorrect, just flush the pipeline

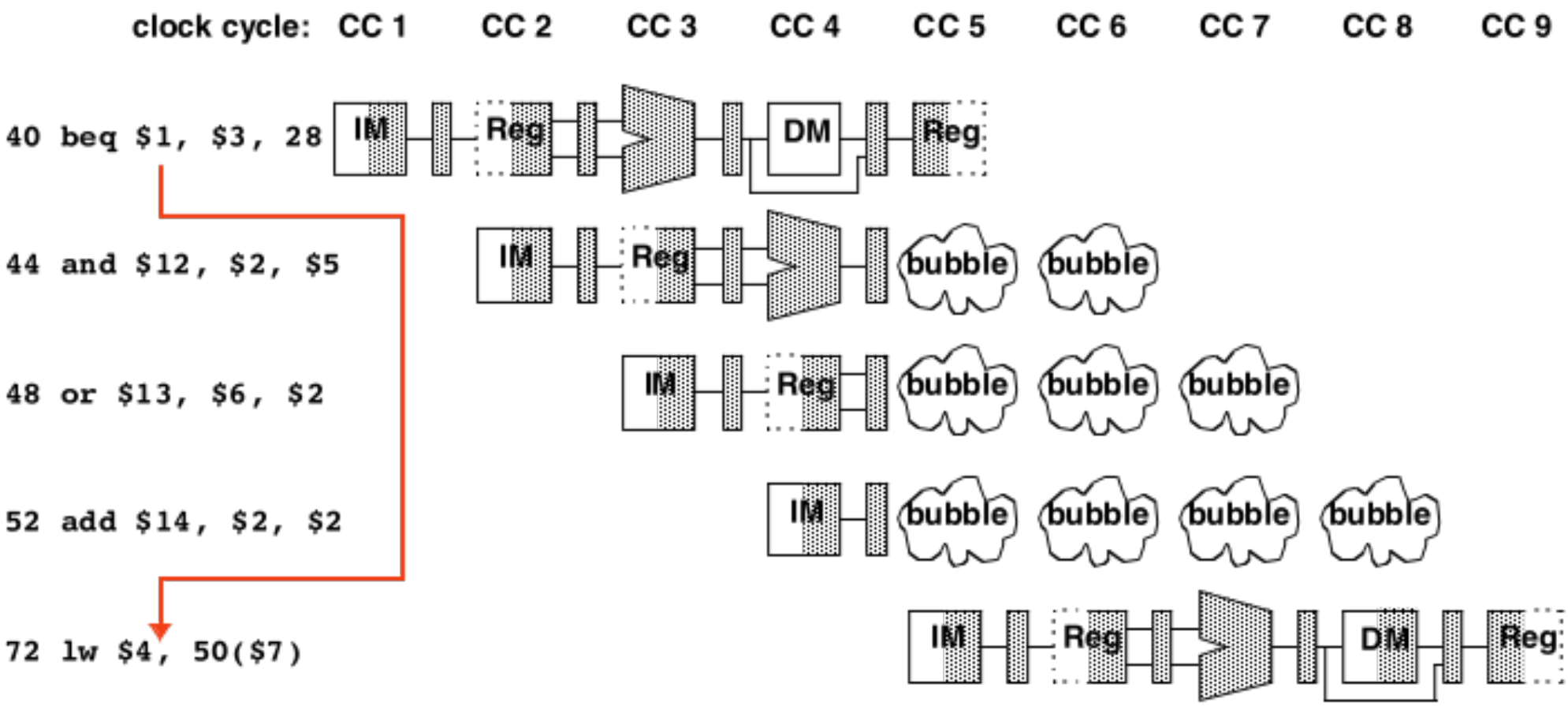
Impact of “predict not taken”

- **Execution proceeds normally – no penalty**

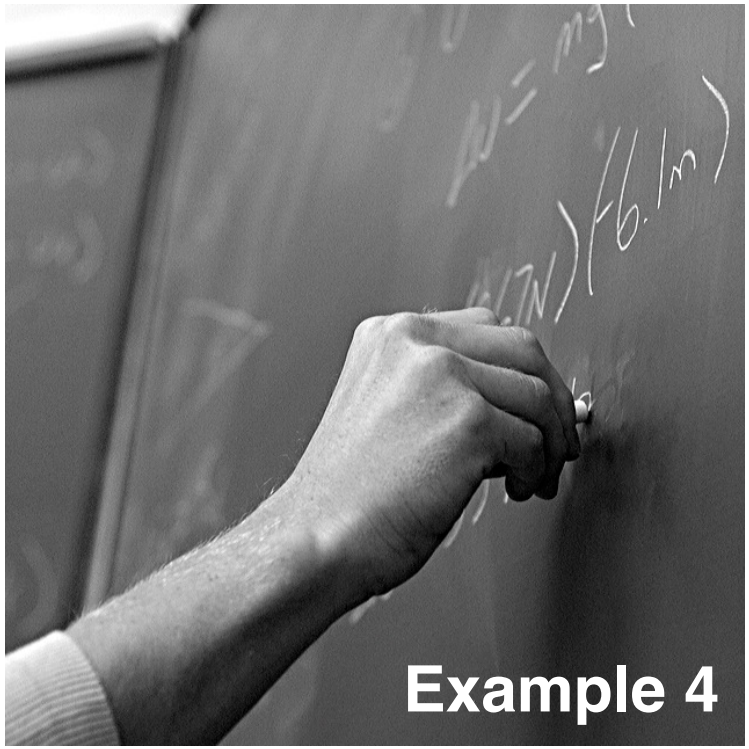


Impact of "predict not taken"

- **Bubbles injected into 3 stages during cycle 5**



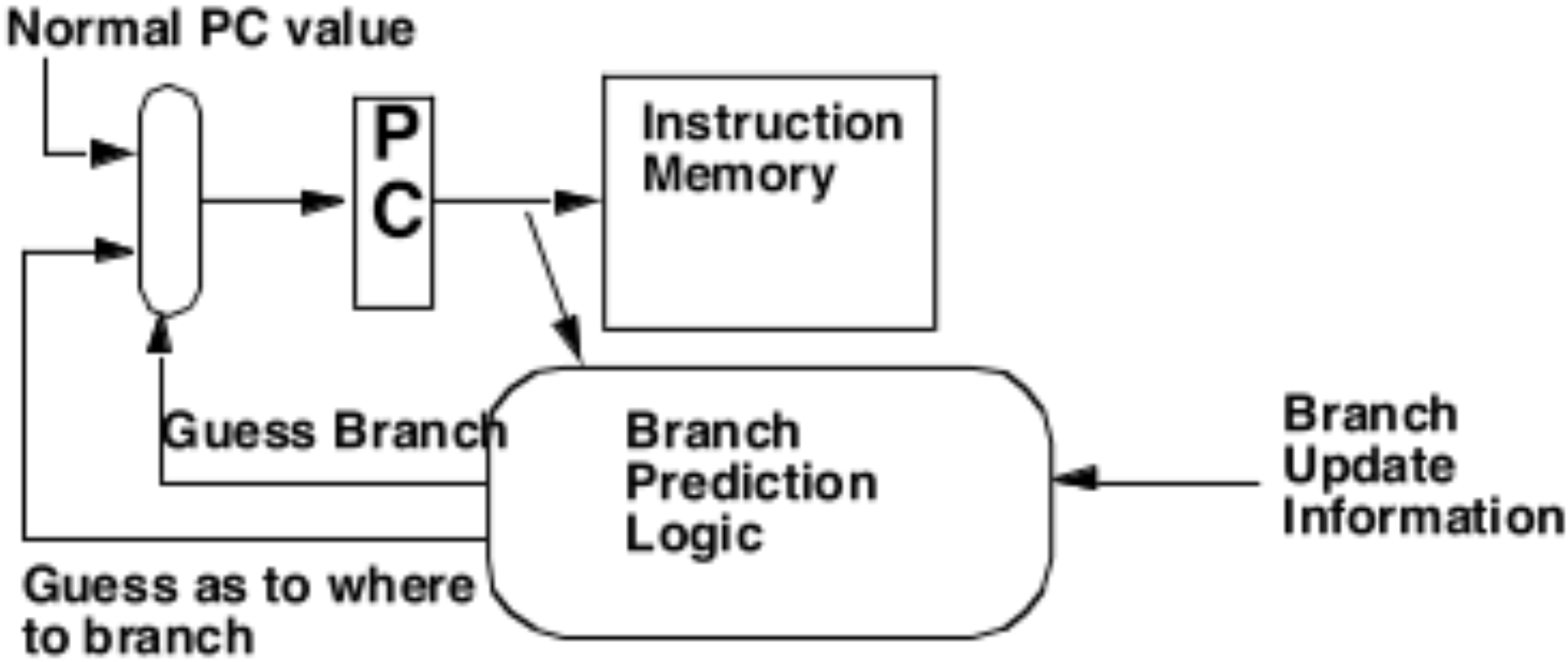
Branch Penalty Impact



Branch Prediction

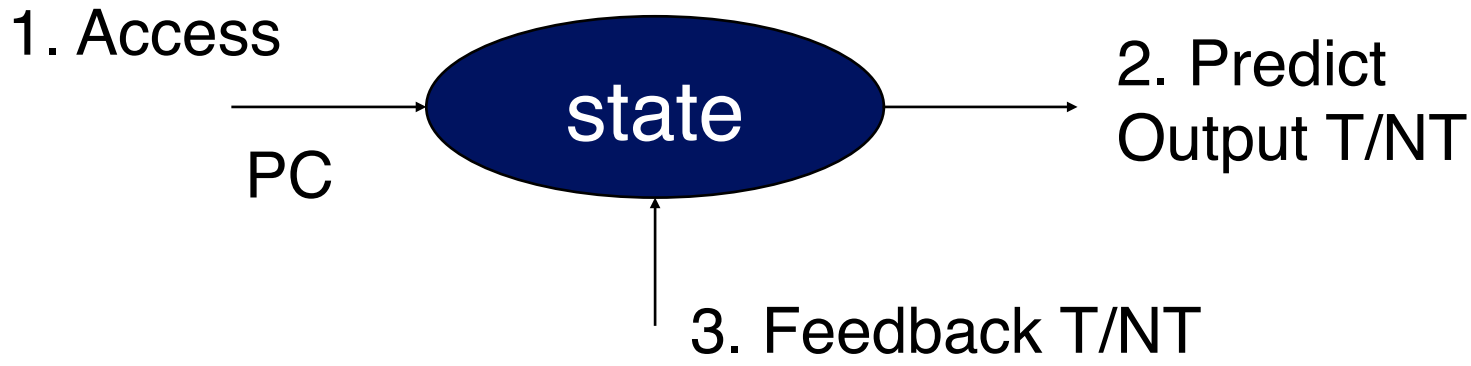
- Prior solutions are “ugly”
- Better (& more common): guess possible outcome
  - Technique is called “branch predicting”; needs 2 parts:
    - “Predictor” to guess where / if instruction will branch
      - (and to where)
    - “Recovery Mechanism”:
      - i.e. a way to fix your mistake
  - Prior strategy:
    - Predictor: always guess branch never taken
    - Recovery: flush instructions if branch taken
  - Alternative: accumulate info. in IF stage as to…
    - Whether or not for any particular PC value a branch was taken next
    - To where it is taken
    - How to update with information from later stages

A Branch Predictor

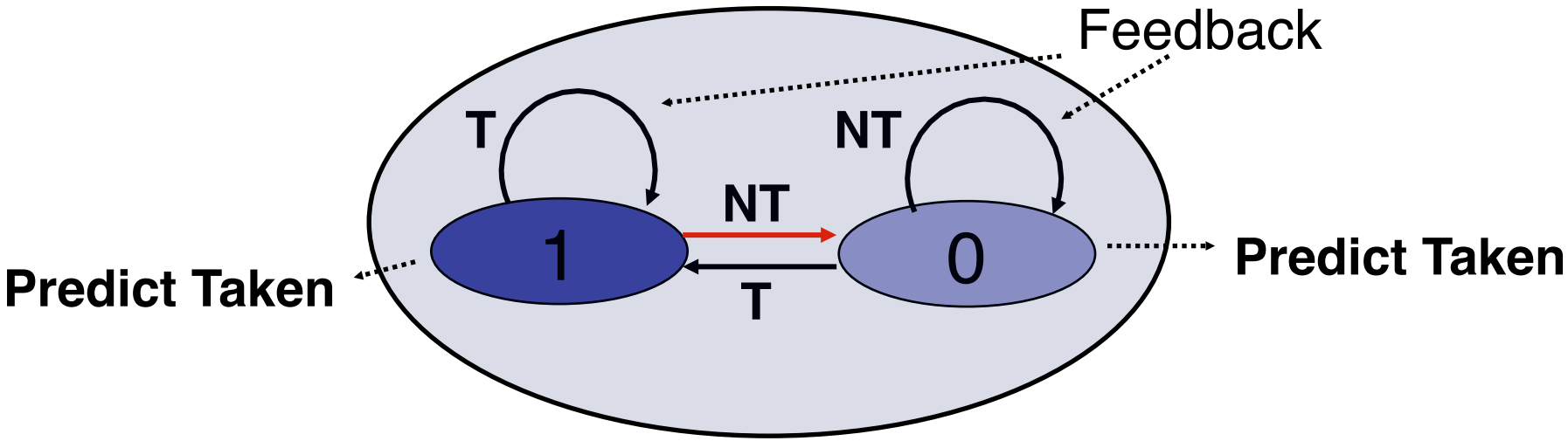


Predictor for a Single Branch

General Form



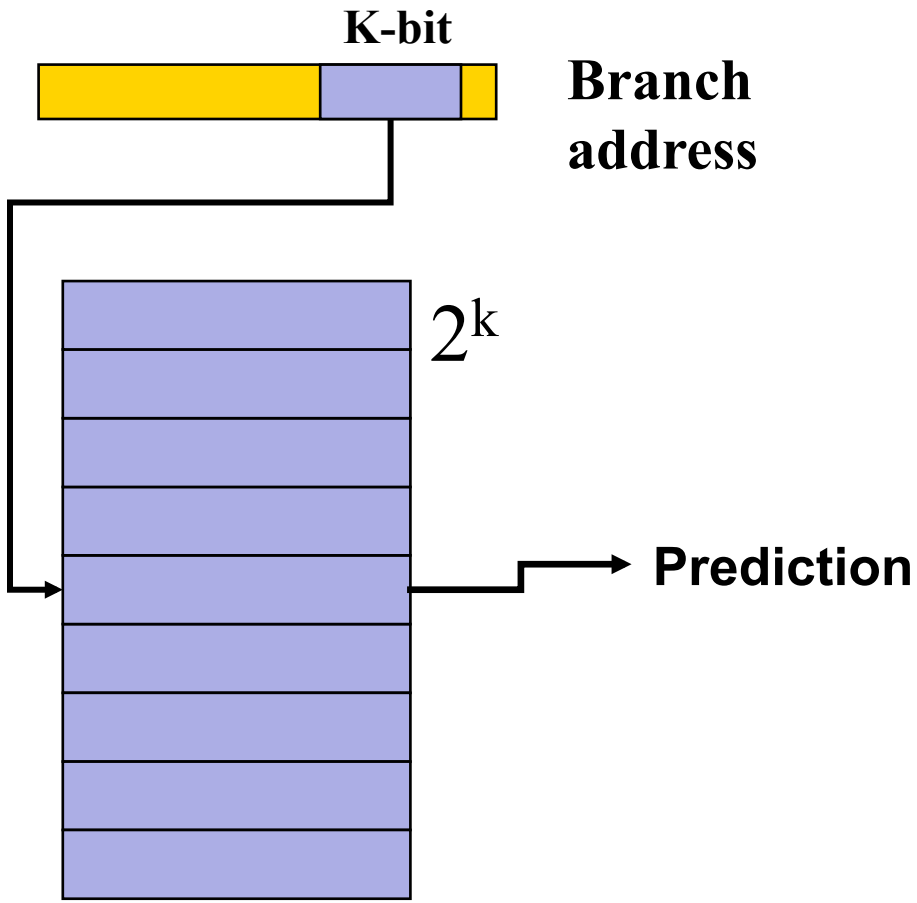
1-bit prediction



Branch History Table of 1-bit Predictor

BHT also called branch prediction buffer

- Can use only one 1-bit predictor, but accuracy is low
- BHT: use a table of simple predictors, indexed by bits from PC
- More entries, more cost, but less conflicts, higher accuracy
- BHT can contain complex predictors



1 bit weakness

- **Example:**
  - in a loop, 1-bit BHT will cause 2 mispredictions

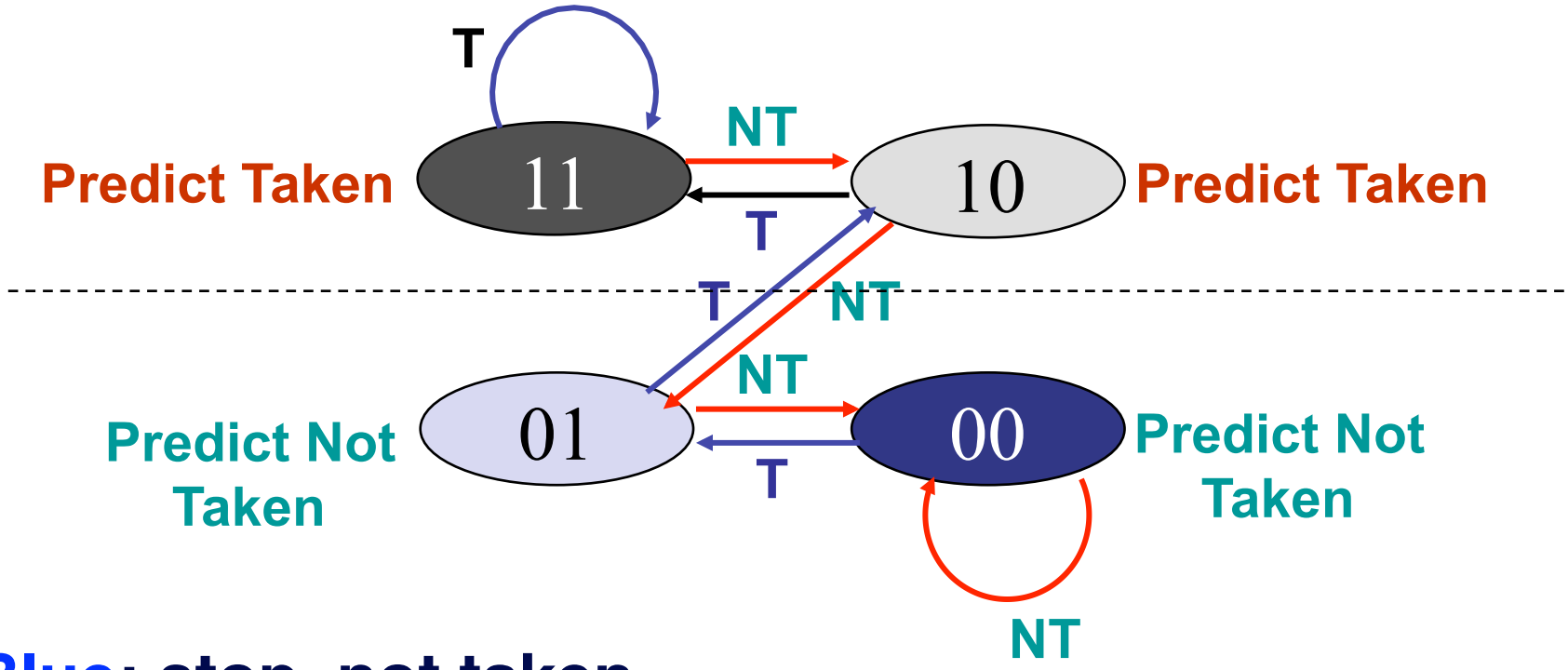
- **Consider a loop of 9 iterations before exit:**

```
for (…){
   for (i=0; i<9; i++)
      a[i] = a[i] * 2.0;
}
```

  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Only 80% accuracy even if loop 90% of the time

2-bit saturating counter

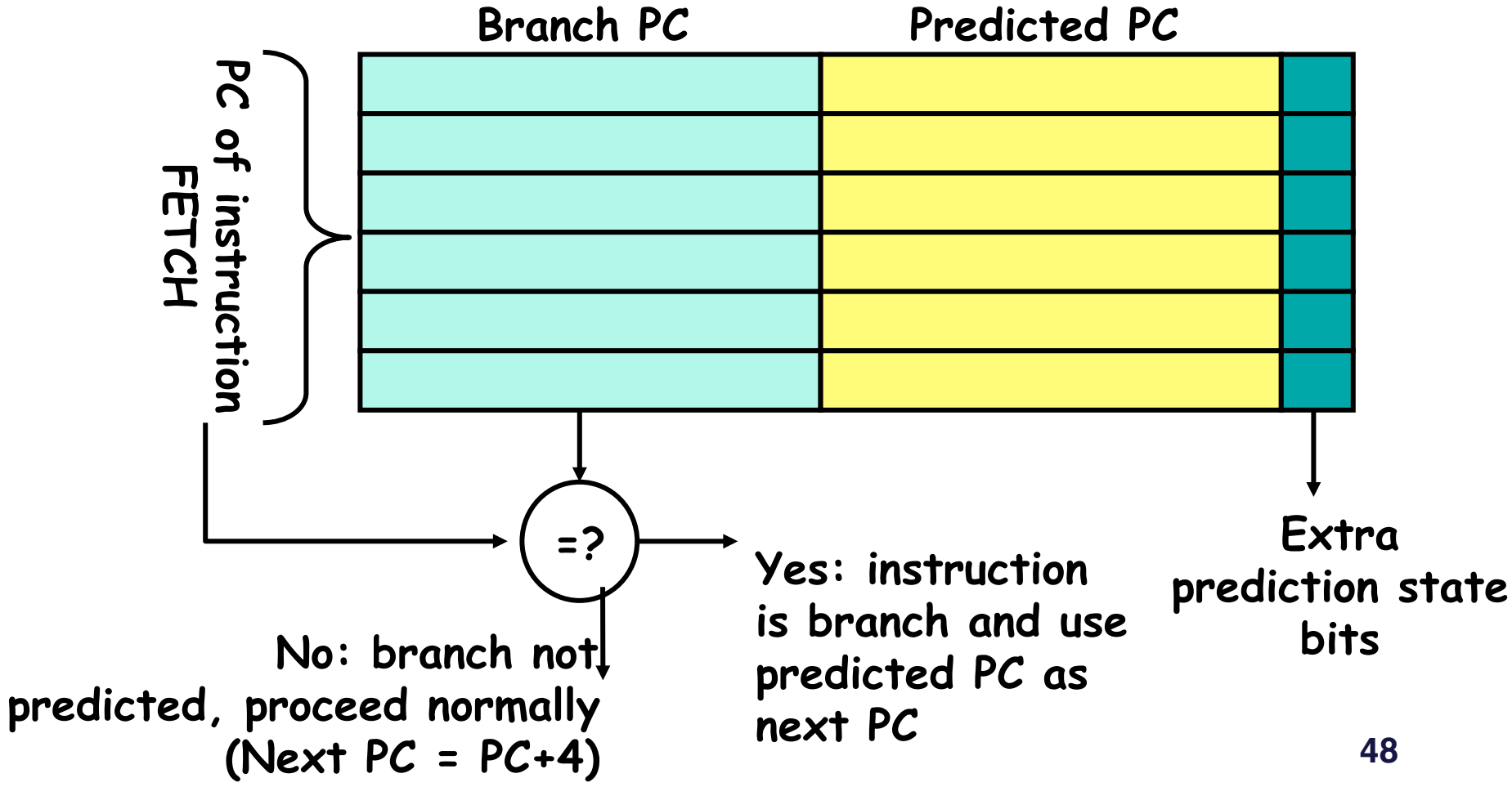
- Solution: 2-bit scheme where change prediction only if get misprediction *twice:*



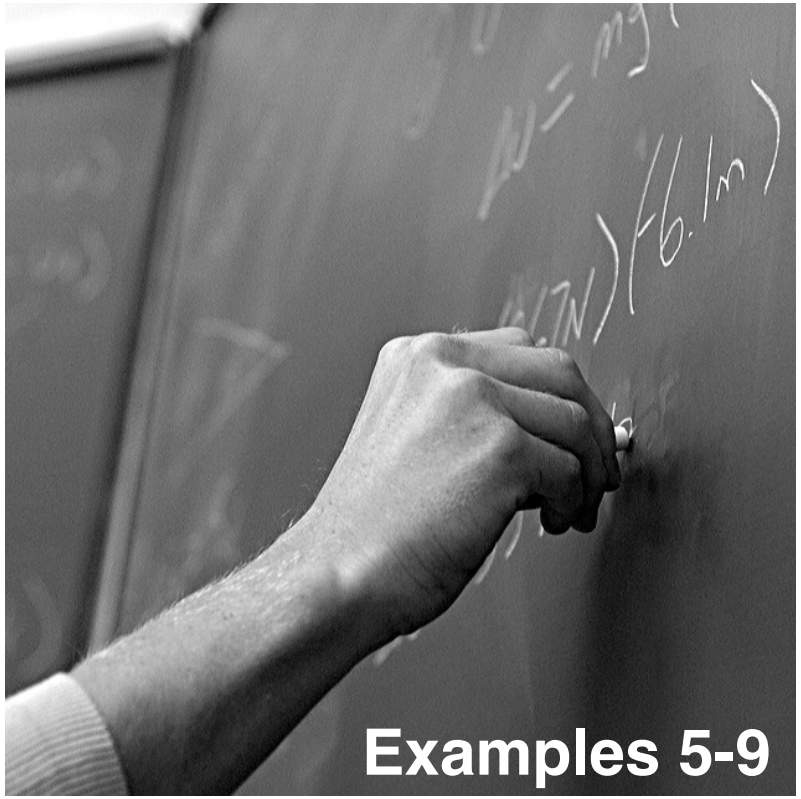
- Blue: stop, not taken
- Gray: go, taken
- Adds *hysteresis* to decision making process

Branch target buffer

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address
- Example: BTB combined with BHT



Examples…

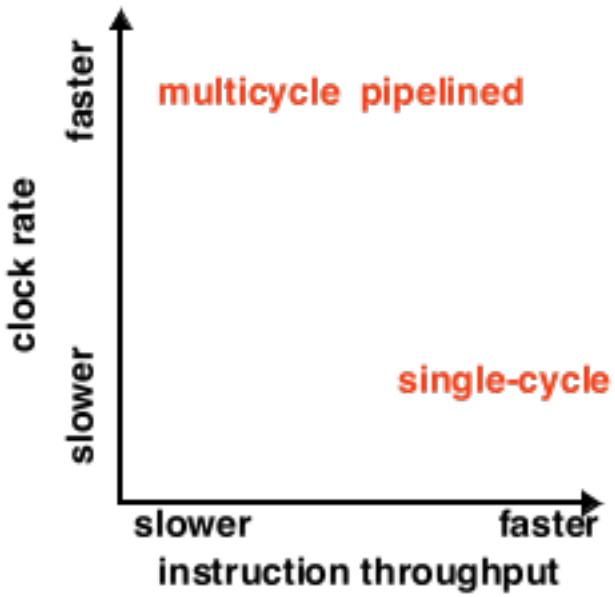


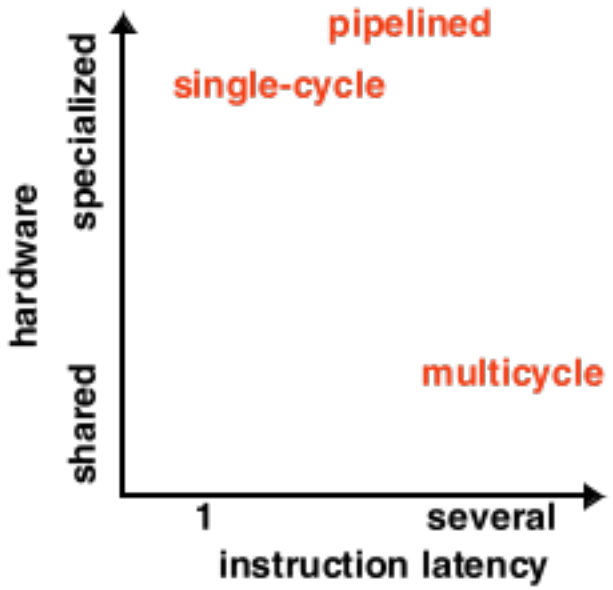
Discussion

- How does instruction set design impact pipelining?

- Does increasing the depth of pipelining always increase performance?

Comparative performance





- Throughput: instructions per clock cycle = 1/cpi
  - Pipeline has fast throughput and fast clock rate
- Latency: inherent execution time, in cycles
  - High latency for pipelining causes problems
    - Increased time to resolve hazards