EE3450:
Computer Architecture
Chapter 4

Part 2: Pipelining-B

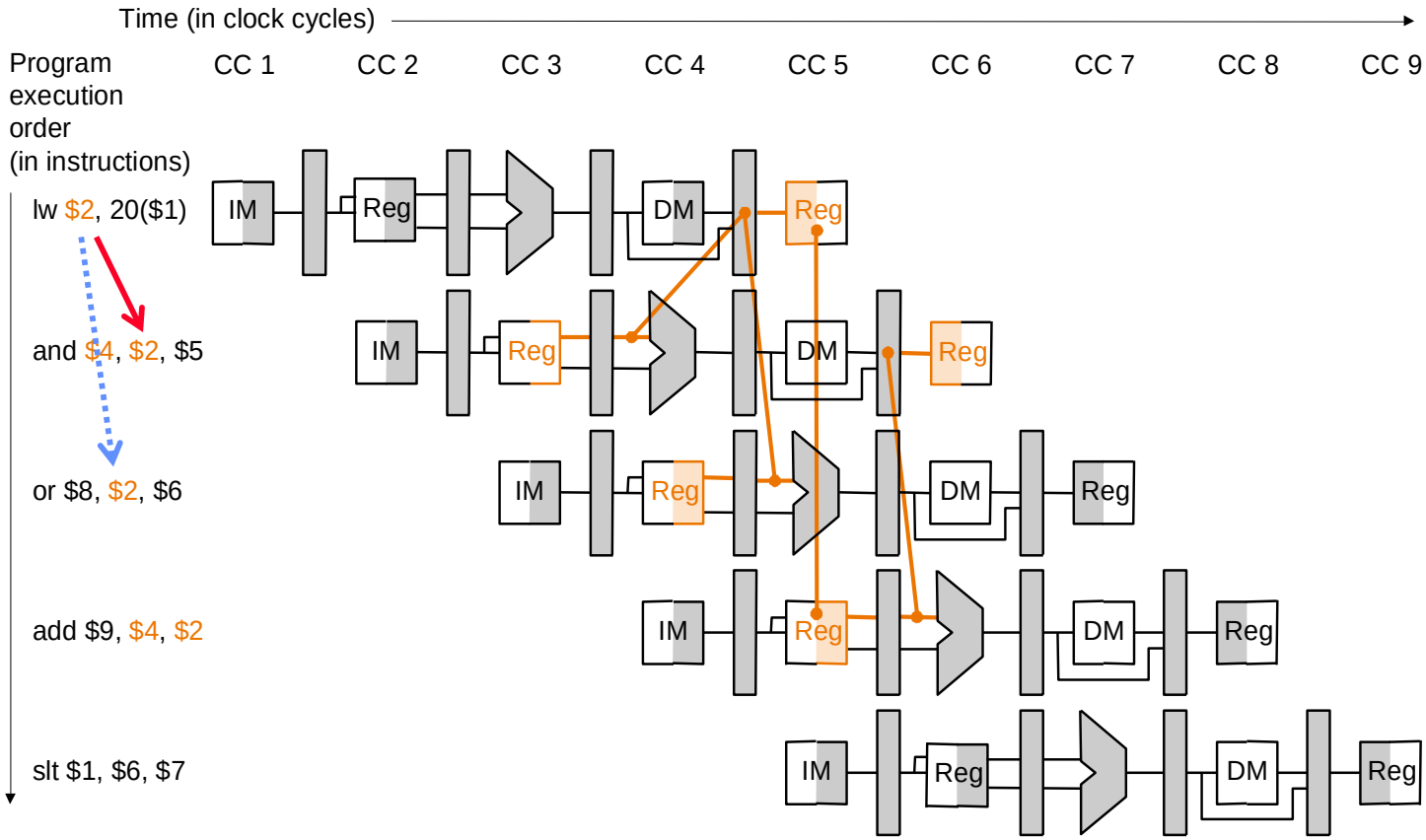
Prof. Yarsun Hsu

EE Department

National Tsing Hua University

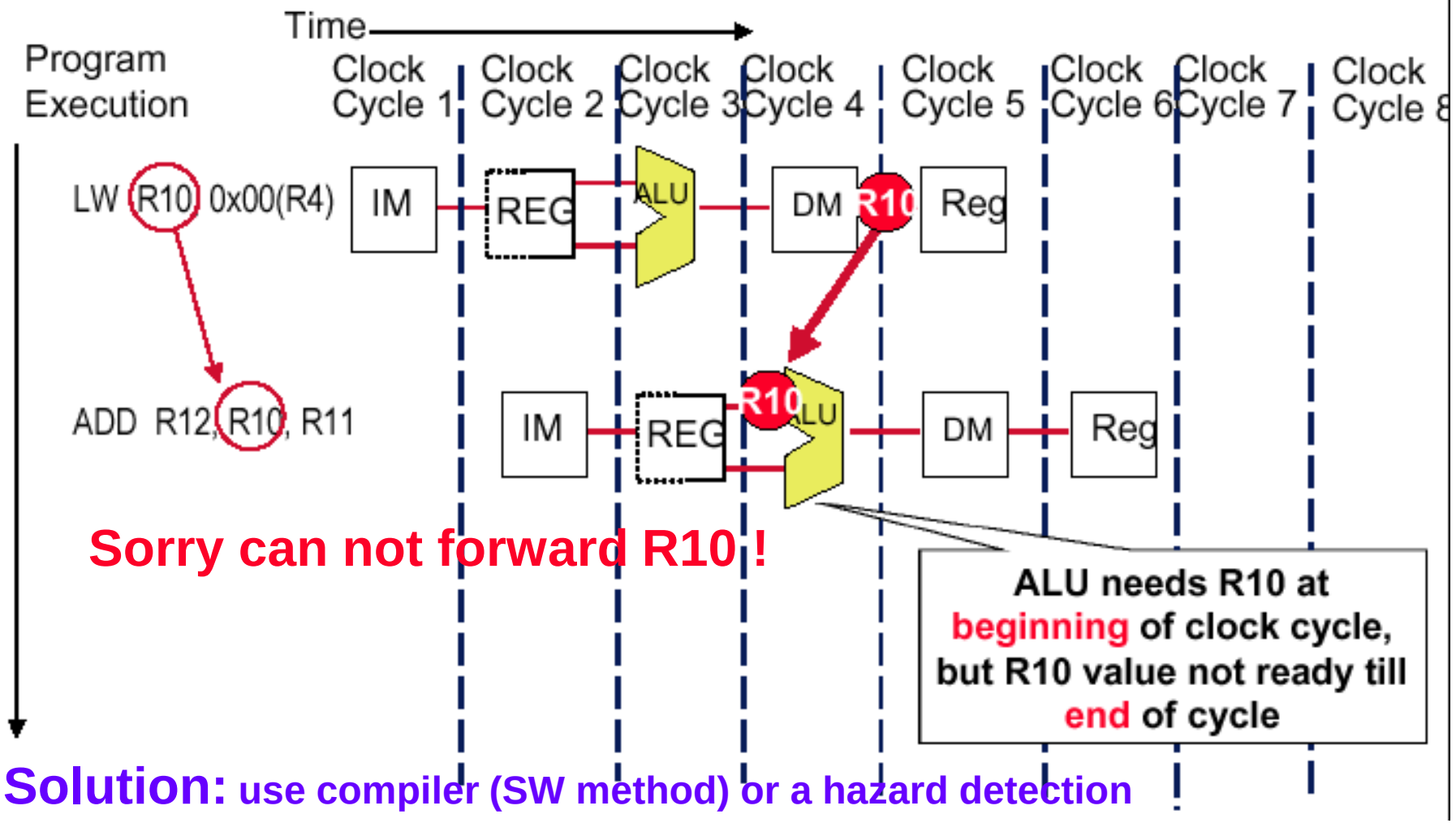
Can't Always Forward

- **Load word can still cause a hazard:**
  - **a load followed by an instr. to read the register just loaded**



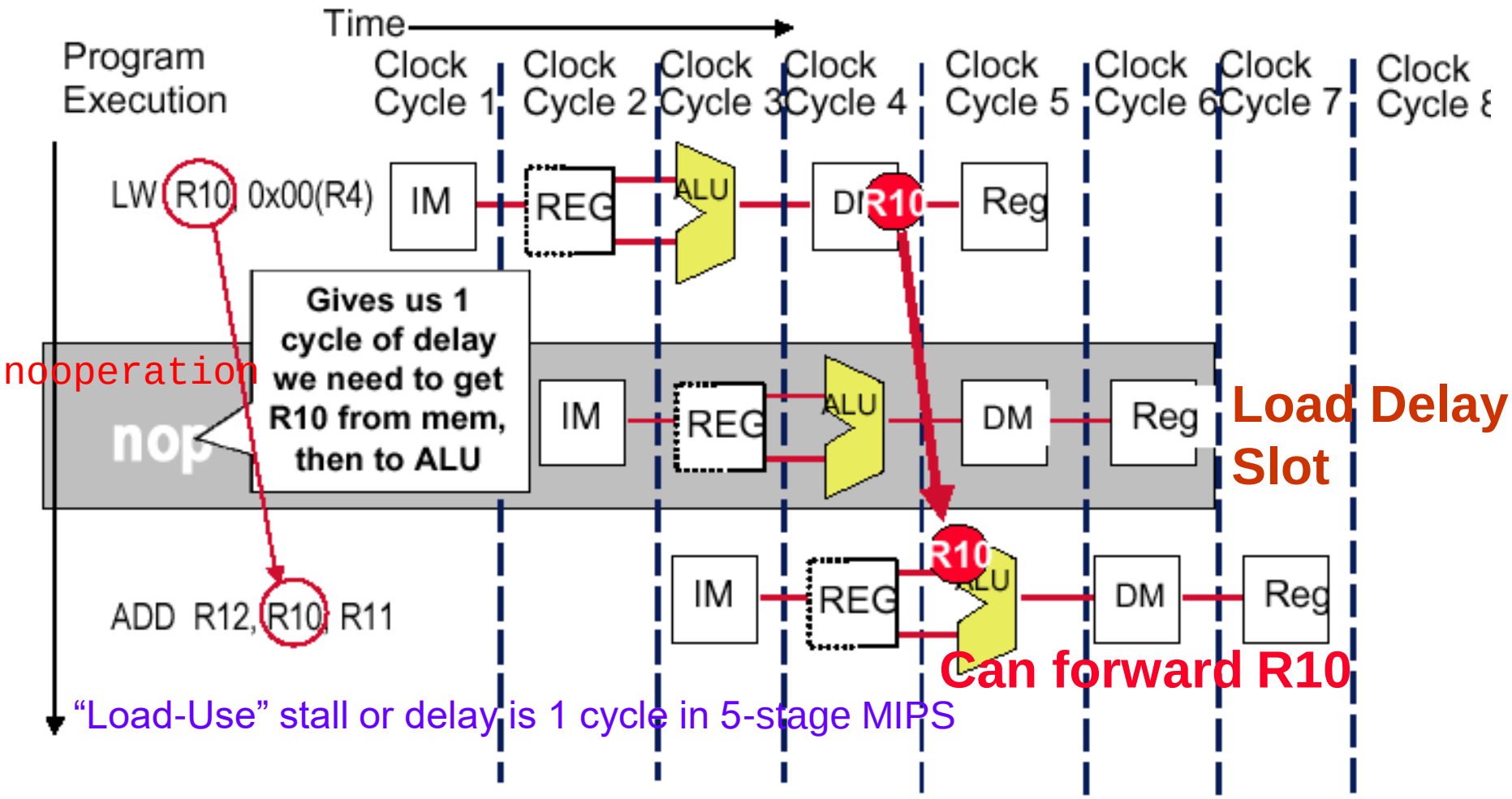
- ***Load-Use* delay between LW and AND via $2**

Forwarding Doesn't Always Work



Solution: use compiler (SW method) or a hazard detection unit to "stall" dependent instruction (HW method)

(A1) Software method: compiler inserts a NOP



NOP: *add $r0, $r0, $r0* or *or $r0, $r0, $r0* behaves like a NOP

(A2) Software method

- **Follow a load with an instn independent of that load**

```
LW   $t0,  0($t1)
LW   $t2,  0($t7)
Add  $t5, $t2, $t3
Add  $t6, $t0, $t4
```

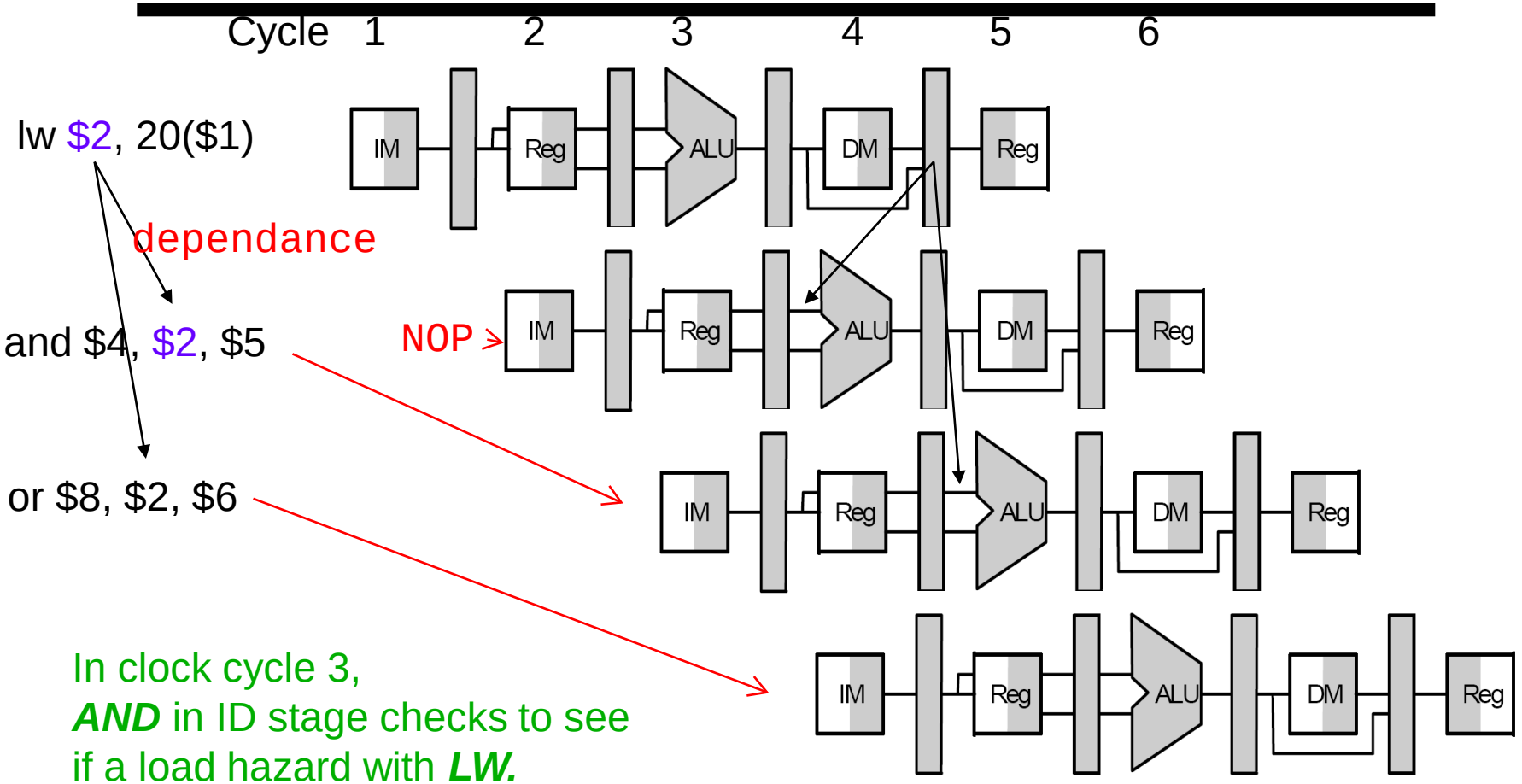
One cycle delay here

Can be re-ordered by compiler to:

```
LW   $t0,  0($t1)
LW   $t2,  0($t7)
Add  $t6, $t0, $t4
Add  $t5, $t2, $t3
```

No delay

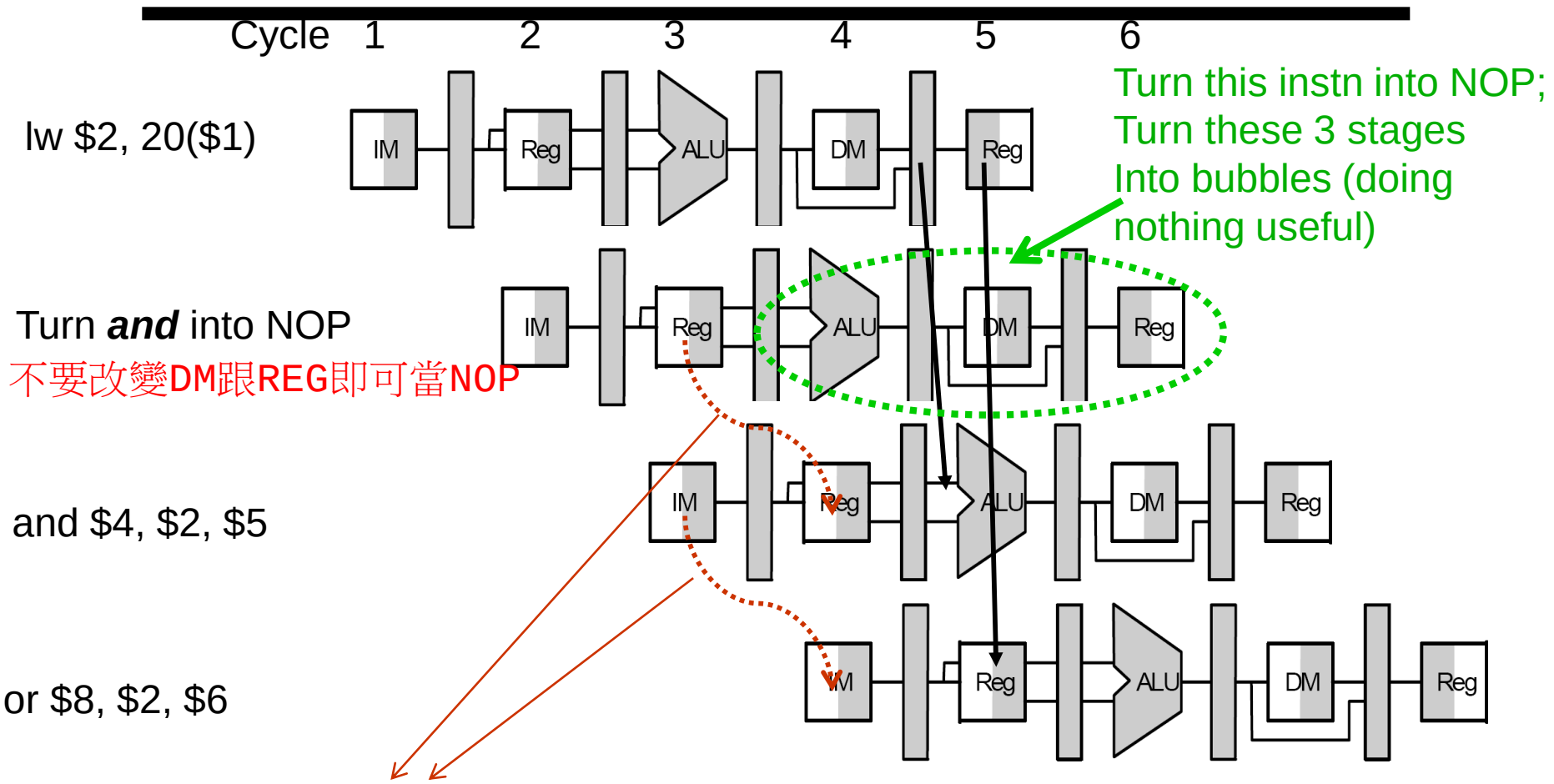
(B) "Load-Use" Delay: HW method



In clock cycle 3,
AND in ID stage checks to see
if a load hazard with ***LW.***

*Notice that **OR** is already in IF stage*

HW method: turn instn into NOP & re-execute instns



Handling Stalls: HW Method

Example: Between LD & AND instns

- **Check if a hazard exists: destination reg (rt) of Load in EXE stage matches any of the two sources (rs, rt) of AND in ID stage?**

```
if (ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.registerRt))
   stall the pipeline for one cycle
```

lw's register

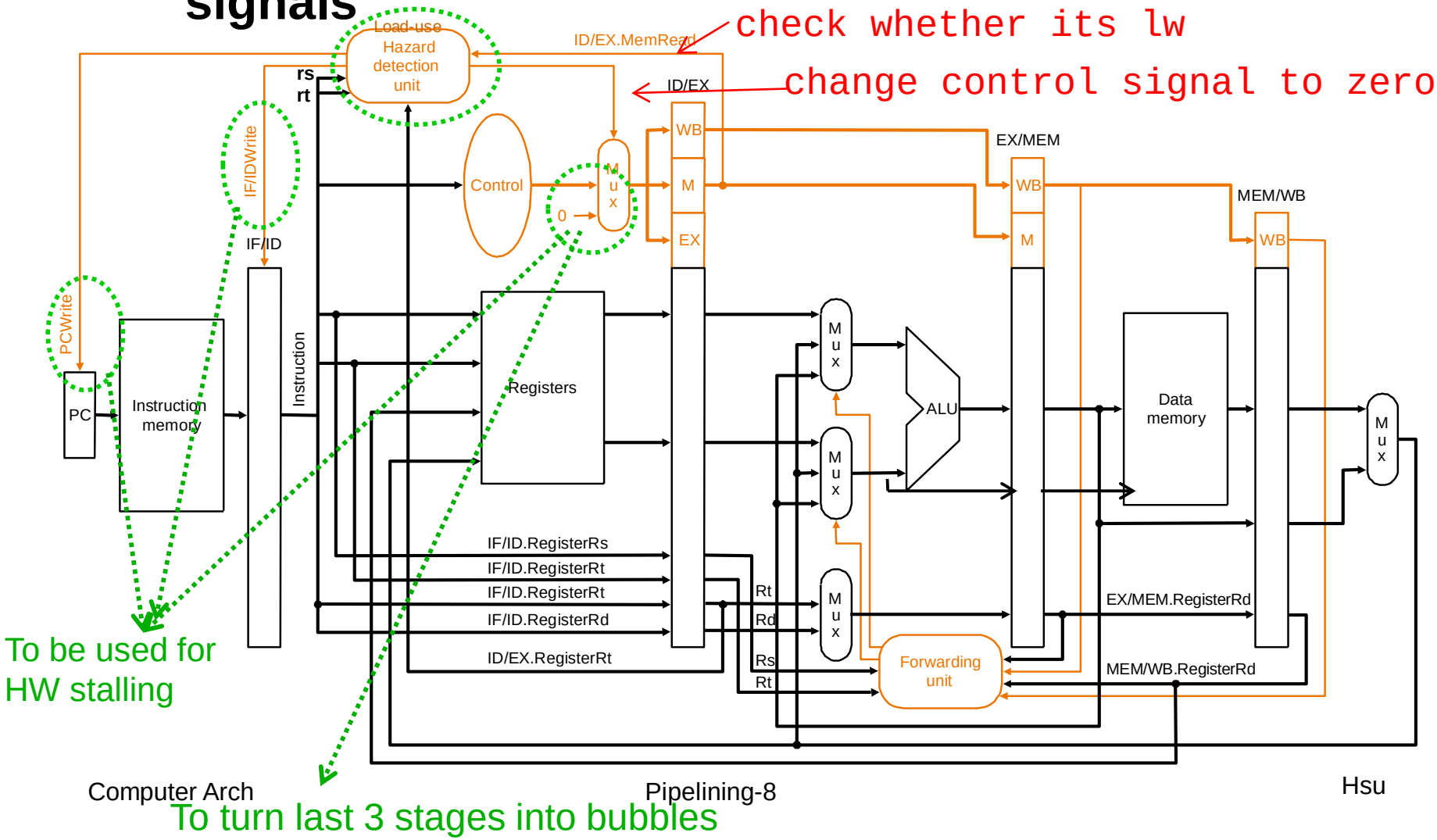
(ID/EX.MemRead = 1 indicates it is a load instruction

ID/EX.RegisterRt is load's destination register)

- **How to stall?**
  - **Stall instructions in IF and ID: not change PC value and IF/ID pipeline reg => the IF & ID stages re-execute the instructions in the next cycle**
    - **IF fetches the same instn using same PC, ID uses same pipe reg**
  - **Turn AND instn into NOP by changing EX, MEM, WB control signals of ID/EX pipeline register to 0: create & move bubble into later stages**
    - **as control signals propagate, all control signals to EX, MEM, WB associated with bubble are deasserted and no registers or memories are written**

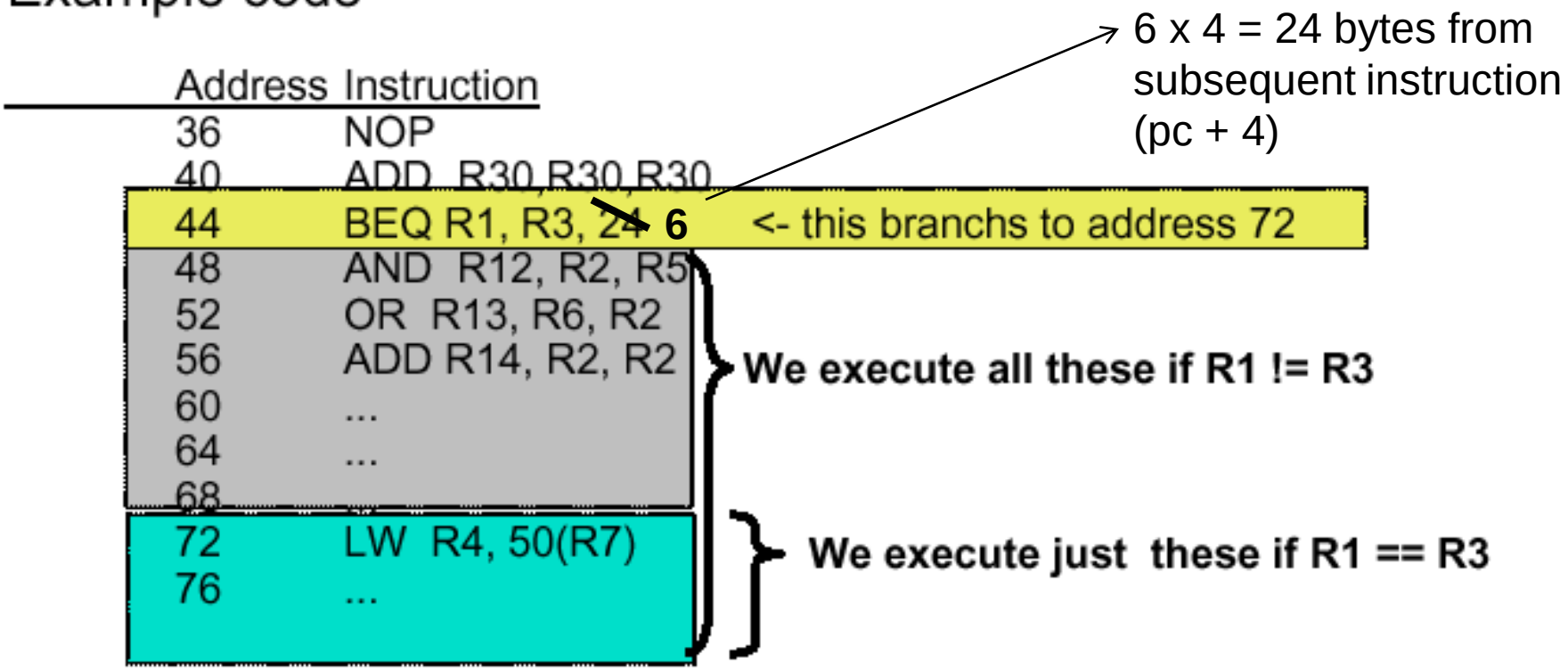
Pipeline with Stalling Unit

- **Hazard detection controls PC, IF/ID and control signals**



Control Hazards - Branches

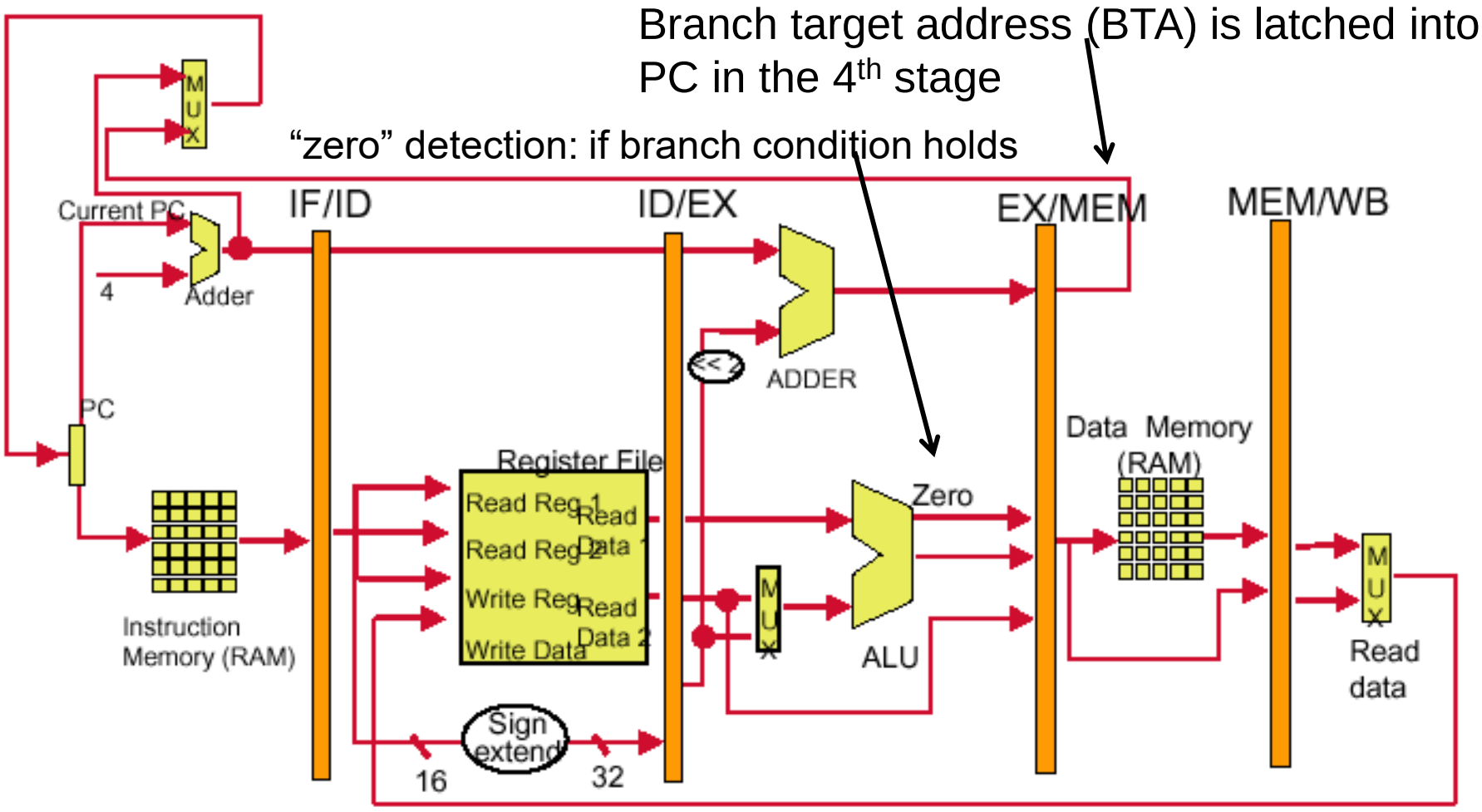
Example code



Flow of instructions if branch is taken: 36, 40, 44, 72, ...
Flow of instructions if branch is not taken: 36, 40, 44, 48, ...

The Problem: already fetch the next instructions by the time branch condition and branch target address are known

Recall: Basic Pipelined Datapath

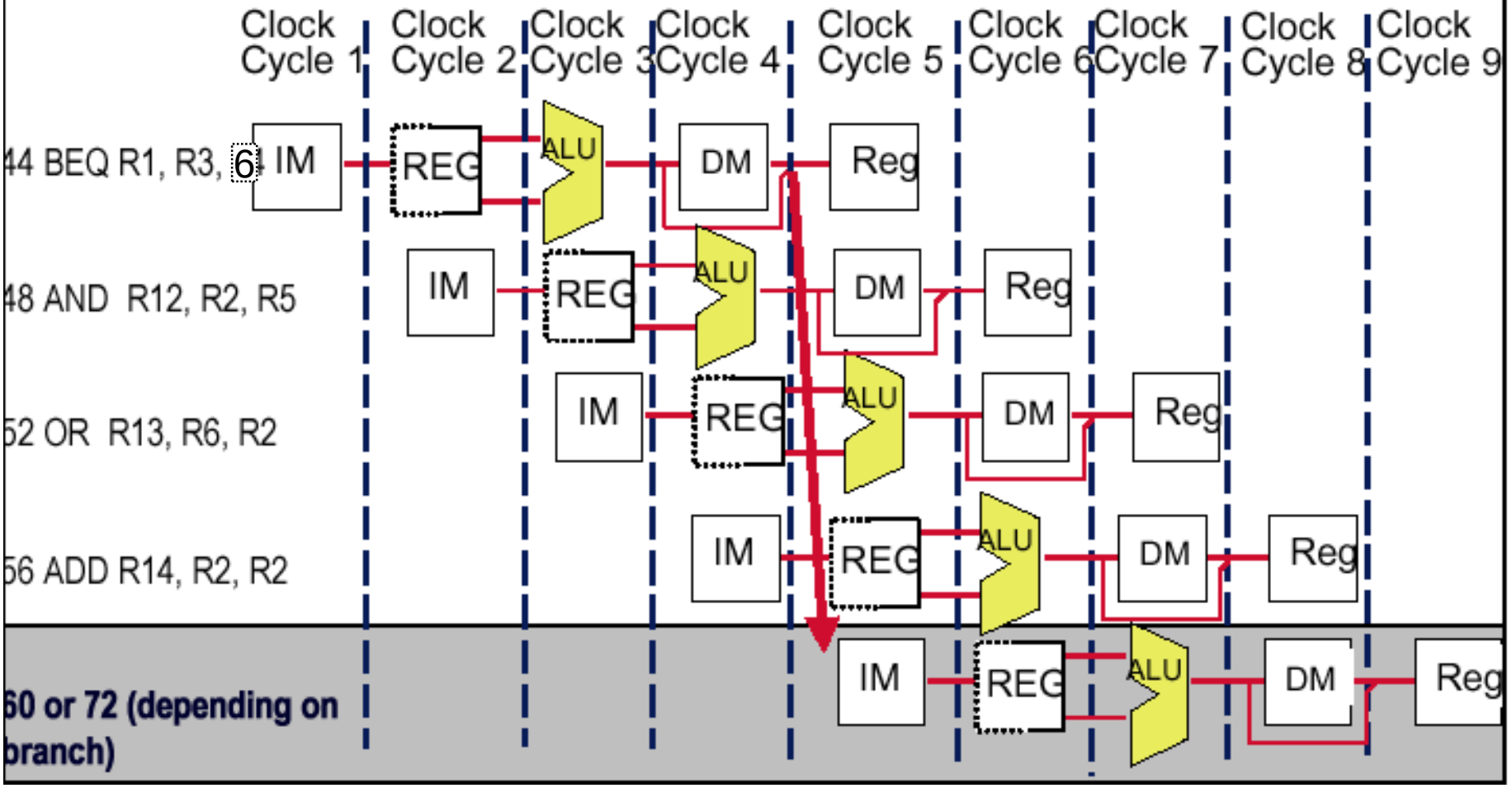


In MEM stage, BTA is latched into PC. However, EXE, ID & IF stages already fill with instns which will not be executed if branch is taken.

Branch Hazards

48, 52, & 56 already in the pipeline before we know the outcome of BEQ

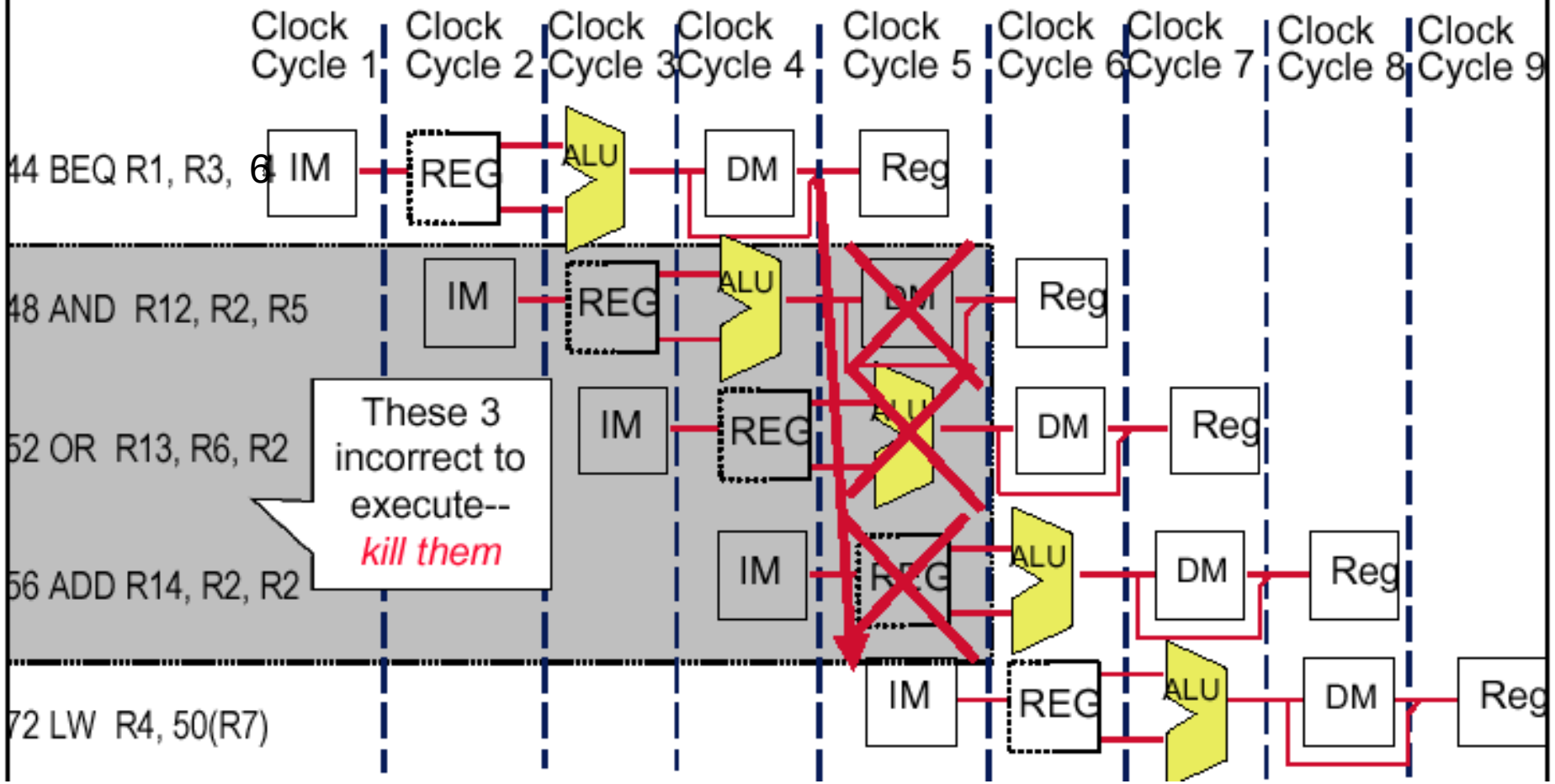
Flow of instructions if branch is taken: 36, 40, 44, 72, ...
Flow of instructions if branch is not taken: 36, 40, 44, 48, ...



If branch is taken, we shouldn't have executed 48, 52, 56 !!!!

What Happens When the Branch IS Taken

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

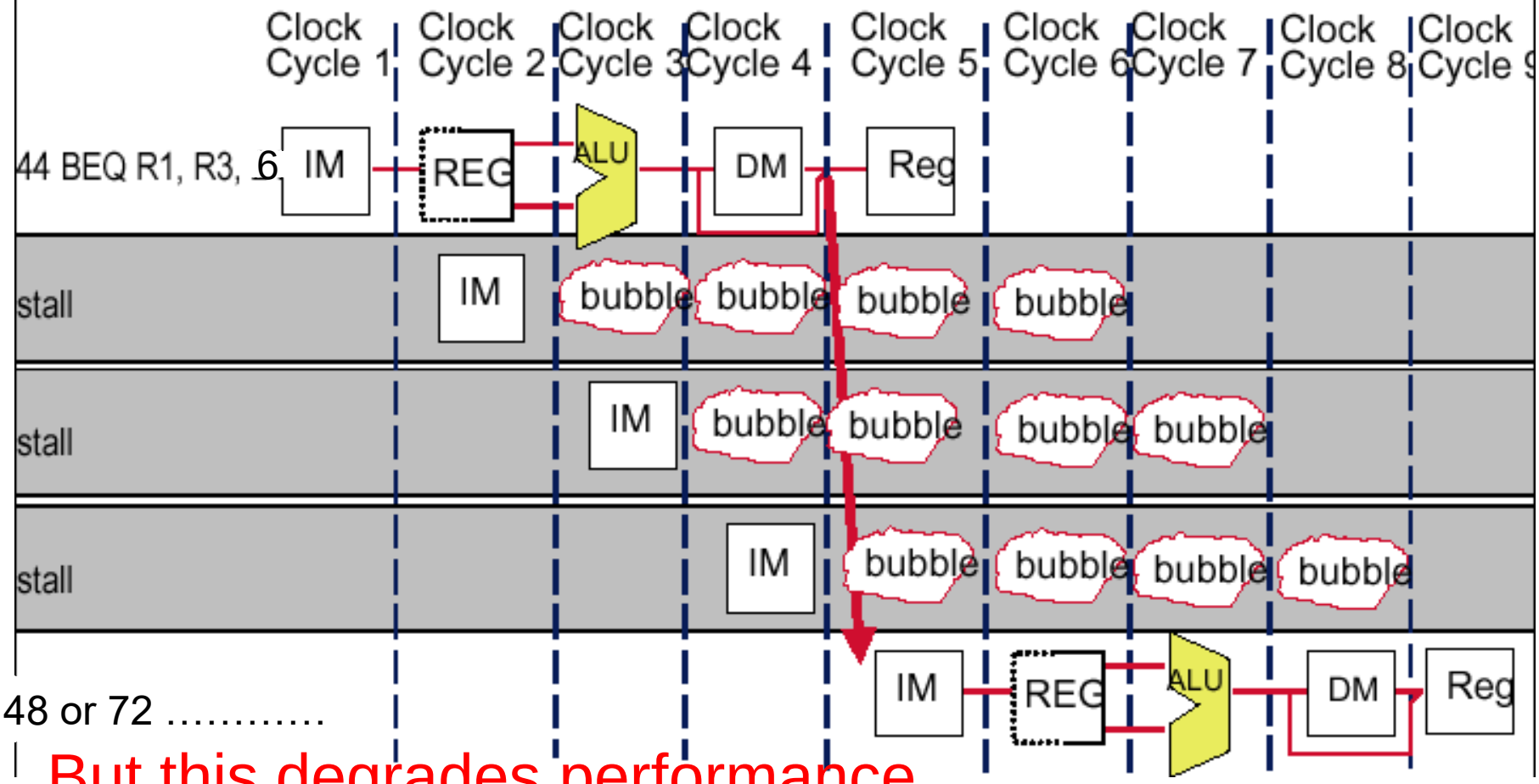


Flush, kill, or discard these three instructions

However, this is not a good solution !!!

Un-exciting way to resolve Branch Hazard: stall until branch outcome is known

For example, compiler inserts 3 NOPs behind BEQ



48 or 72 ……

But this degrades performance

Can we reduce branch stalls?

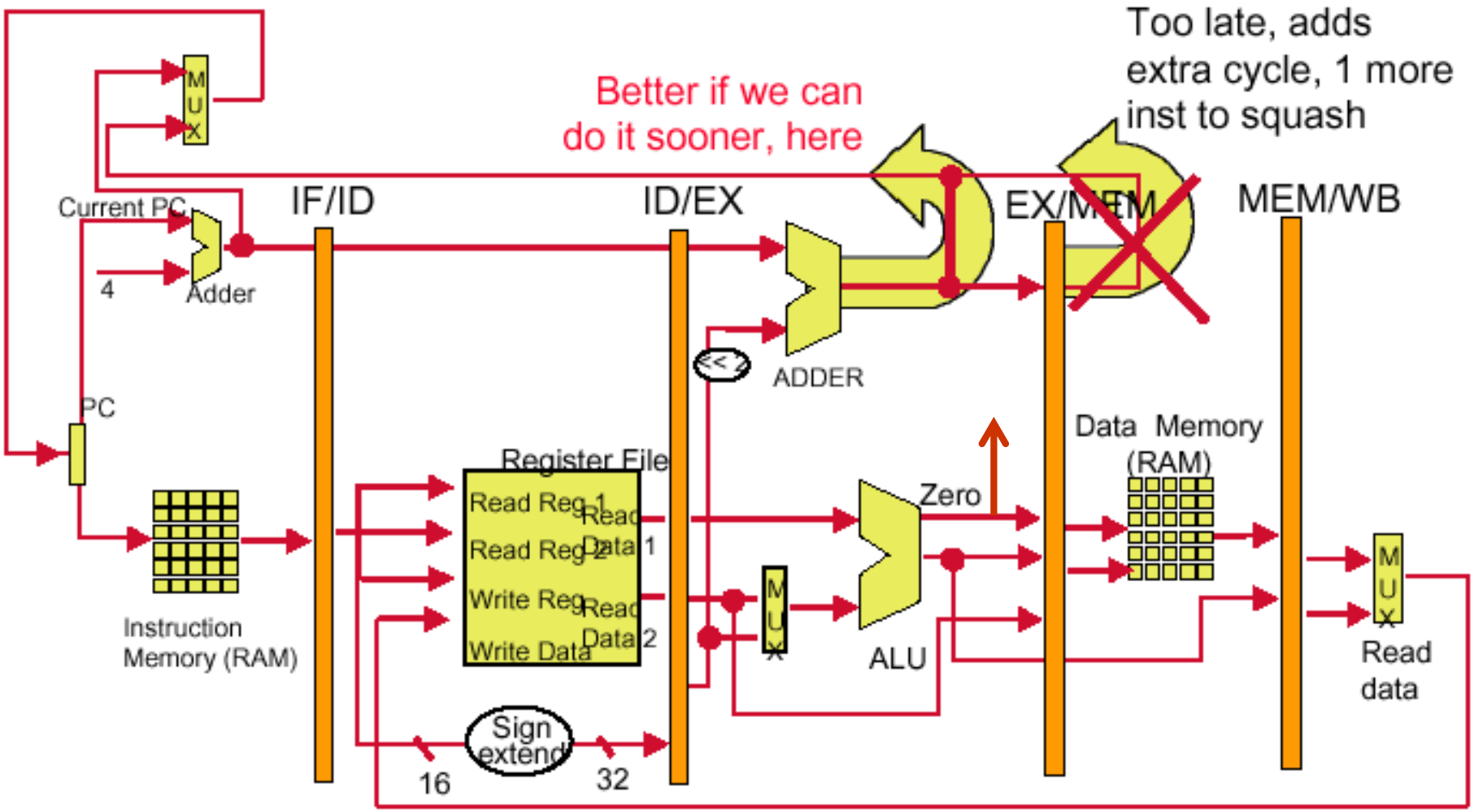
- **Yes, if we can know BTA and branch condition earlier**
- **How can we do that?**

Move branch comparison & branch target computation up to decode / reg. fetch stage.

- **only == test, not > or <**
- **can use *xor* for == test**

Condition Test & Branch Target Address

Move the Branch Computation Forward



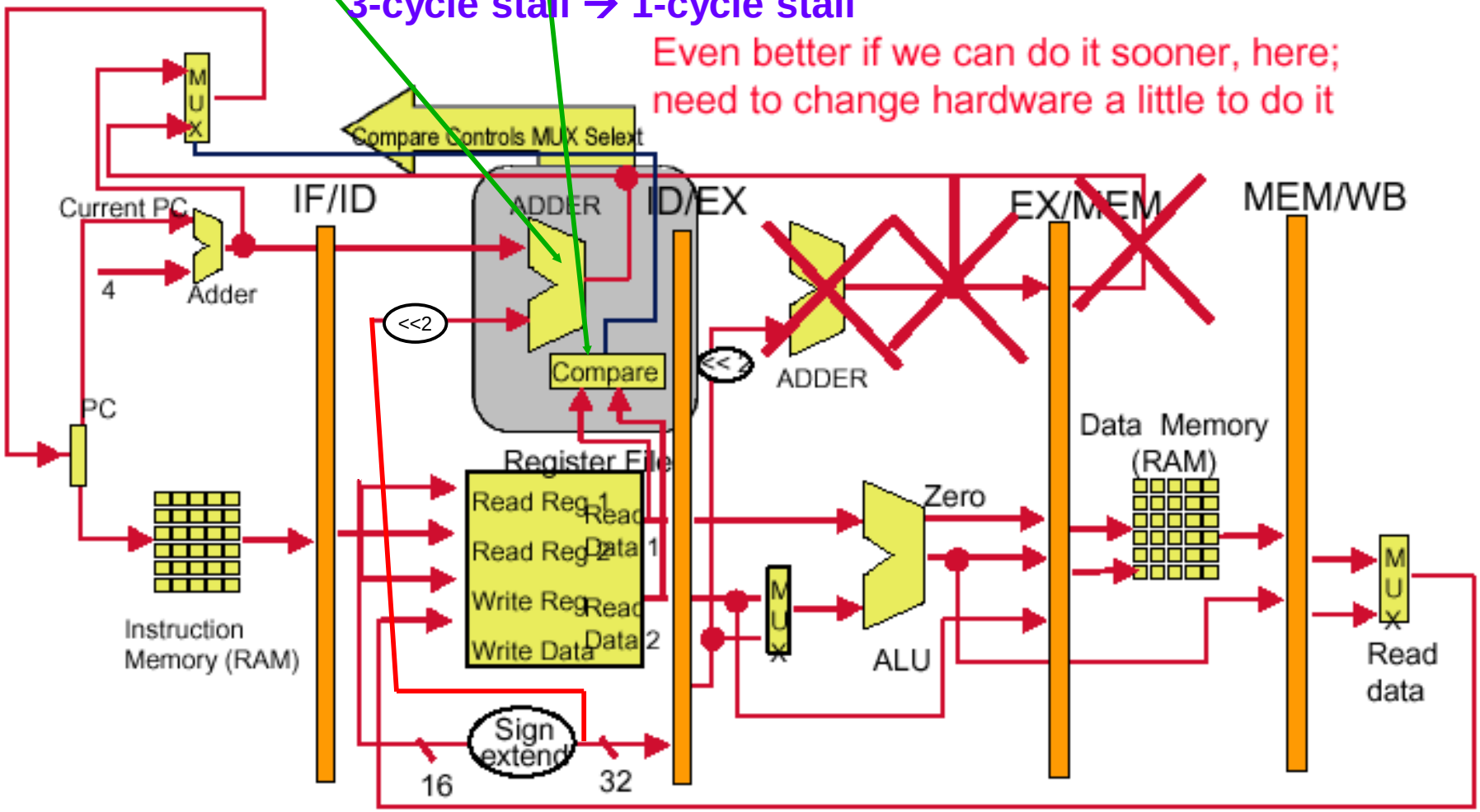
To calculate branch target address

To see if branch condition holds (compare using *xor* to see if *rs=rt*)

Move the Branch Computation Further Forward

3-cycle stall → 1-cycle stall

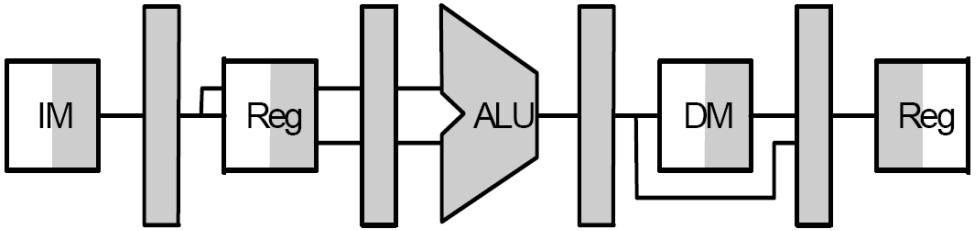
Even better if we can do it sooner, here;
need to change hardware a little to do it



Use XOR for ***BEQ*** comparison

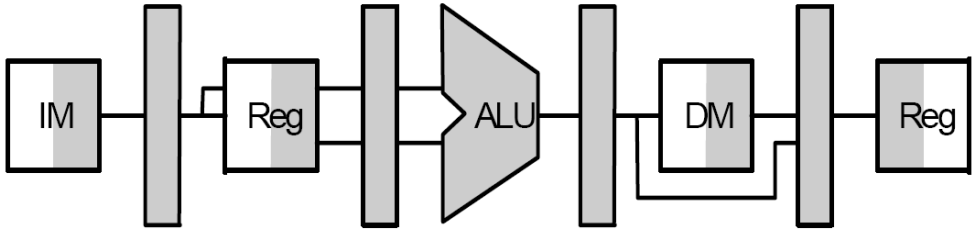
So we've reduced BEQ stalls from 3 to 1

44 BEQ



Stall

48 or 72



. . . . .

Compiler can always insert a NOP behind a branch to account for the 1 cycle stall. But can we do better again?

Can we improve further?

- **Branch occurs frequently so it is desirable to reduce branch stalls further**
- **Possible solutions**
  1. **Branch delay slot**
     - **In MIPS-1, this slot is always executed**
     - **Compiler finds a useful instn for the slot**
     - **Otherwise, inserts NOP**
  2. **Static branch prediction**
  3. **Dynamic branch prediction**

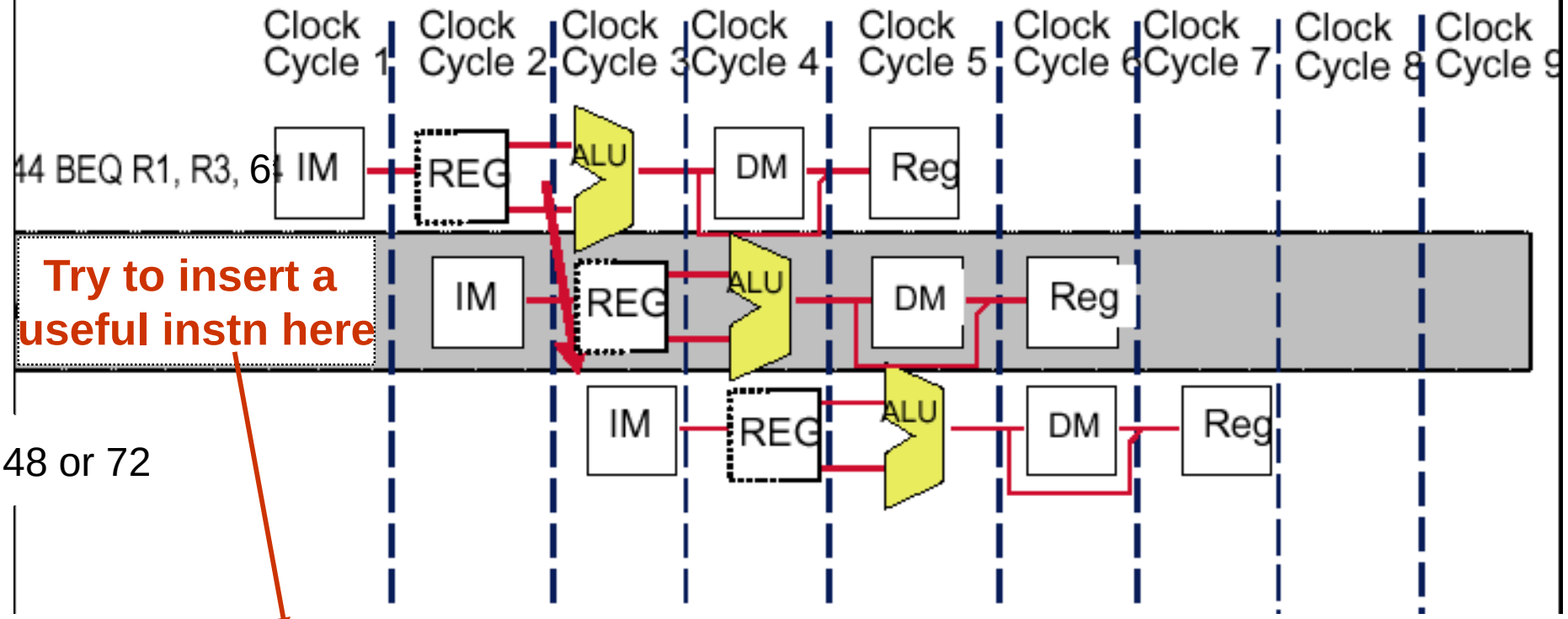
Branch Delay Slot in MIPS-1

- Good news
  - Just 1 cycle to figure out what the right branch address is
  - So, not 2 or 3 cycles of potential NOP or stall
- Strange news
  - OK, it's always 1 cycle, and we always have to wait
  - **But** -on MIPS, this instruction always executes, no matter what
- Hence the name: branch delay slot
  - The instruction cycle after the branch is used for address calc, 1 cycle delay necessary
  - SO…we regard this as a free instruction cycle, and we just DO IT
- Consequence
  - You (or your compiler) will need to adjust your code to put some useful work in that "slot", since just putting in a NOP is wasteful

Compiler can insert a *nop* after *BEQ* if a useful instn cannot be found.

Result: New & Improved MIPS Datapath

- Need just 1 extra cycle after the BEQ branch to know right address
- On MIPS, its called - the branch delay slot



In MIPS-1, this slot is always executed. Compiler tries to find a useful instn and puts it in this delay slot, otherwise inserts a NOP

Rewriting the Code for a Branch Delay Slot

- Without Branch Delay Slot

Address	Instruction
36	NOP
40	ADD R30,R30,R30
44	BEQ R1, R3, ~~24~~ 6
48	AND R12, R2, R5
52	OR R13, R6, R2
56	ADD R14, R2, R2
60	...
64	...
68	...
72	LW R4, 50(R7)
76	...

With Branch Delay Slot

Address	Instruction
36	NOP
~~40~~	BEQ R1, R3, ~~28~~ 7
~~44~~	ADD R30, R30, R30
48	AND R12, R2, R5
52	OR R13, R6, R2
56	ADD R14, R2, R2
60	...
64	...
68	...
72	LW R4, 50(R7)
76	...

Since *ADD R30, R30, R30* is executed anyway & *BEQ* does not depend on its outcome, we can safely move it behind BEQ

Effectiveness of Delayed branch

- **Simplest approach: put NOP after every branch or jump**
  - **20% control transfers**
  - **=> CPI = 1 + 1x0.2 = 1.2**
- **In practice compiler can fill delay slot with a useful instruction about 50% of the time**
  - **independent inst. before the branch**
  - **benign instruction at target**
  - **=> CPI = 1 + 1 x 0.2 x 0.5 = 1.1**
- **Delayed branching is effective for 5-stage MIPS pipeline**
- **However, as processors go to longer pipelines and issuing multiple instns per clock cycle, delayed branching becomes less effective**
  - **Uses dynamic branch prediction instead**

Branch Prediction

- There are many different schemes
  - Assume taken
  - Assume not taken
  - 1-bit Branch Prediction
  - 2-bit Branch Prediction
  - N-bit Branch Prediction
  - 2 level branch prediction
  - …..

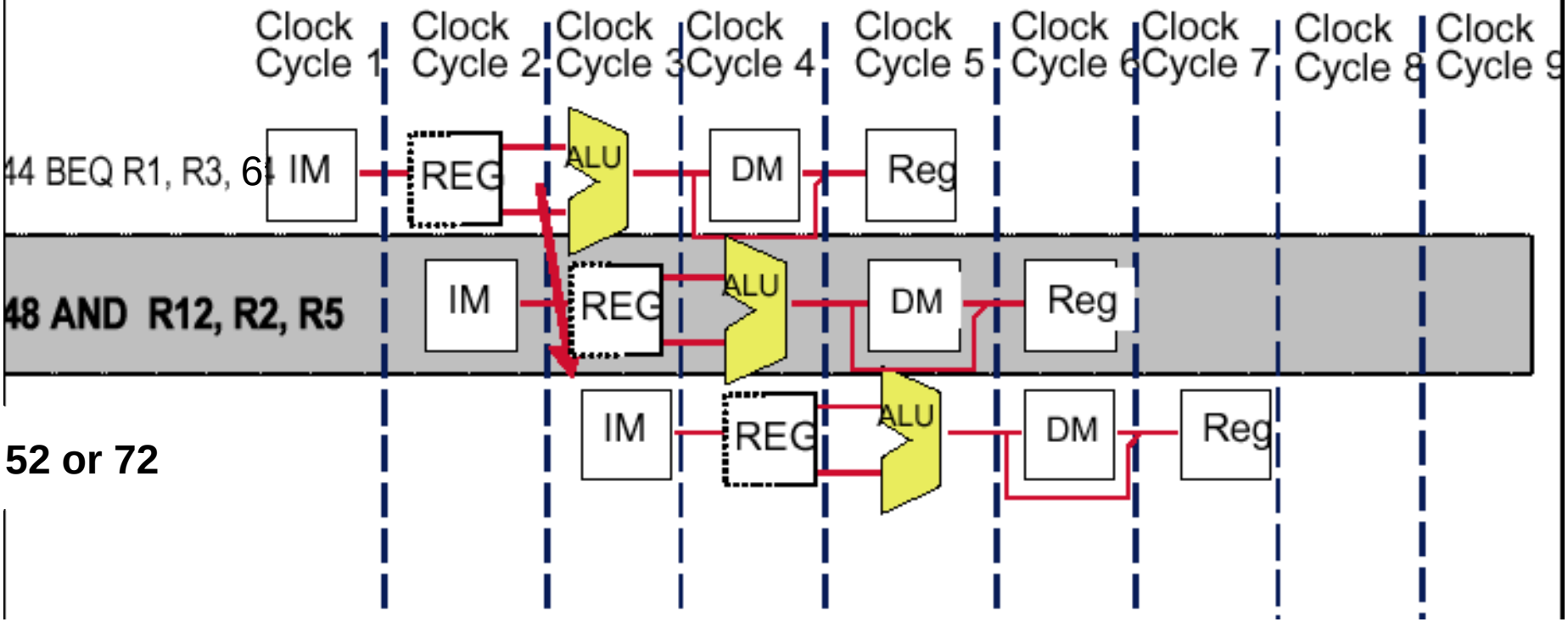
Today, many processors do not use branch delay slot. Instead, they use branch prediction.

- Assume taken or not taken is called static branch prediction
- Using hardware to dynamically predict is called dynamic branch prediction

at run time

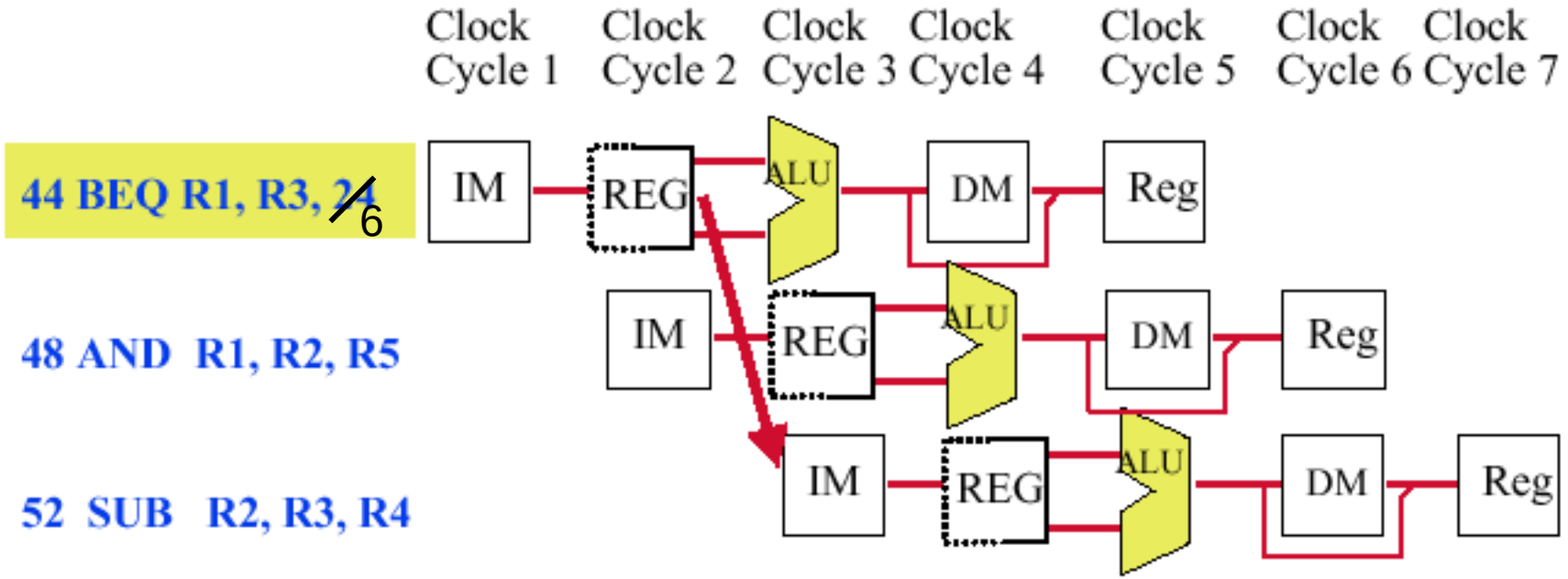
(1) Static Branch Prediction

Make a guess: assume branch is *not taken*



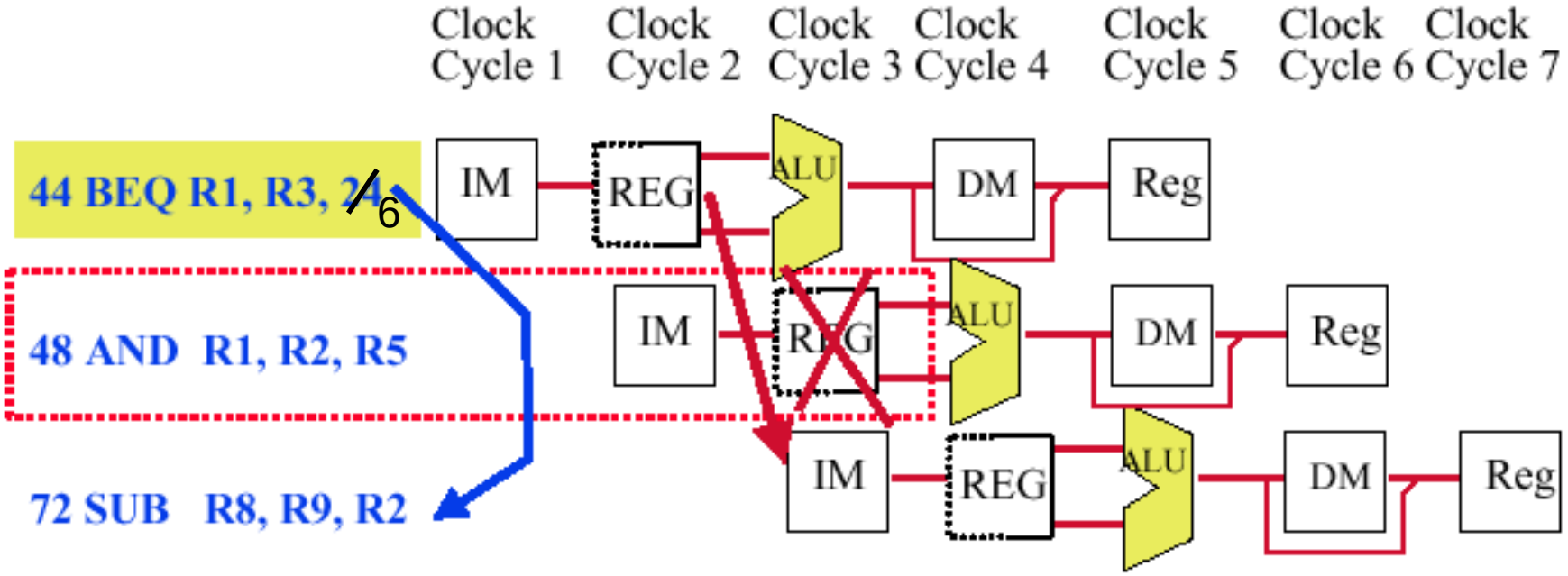
Predict Branch Not Taken

- 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

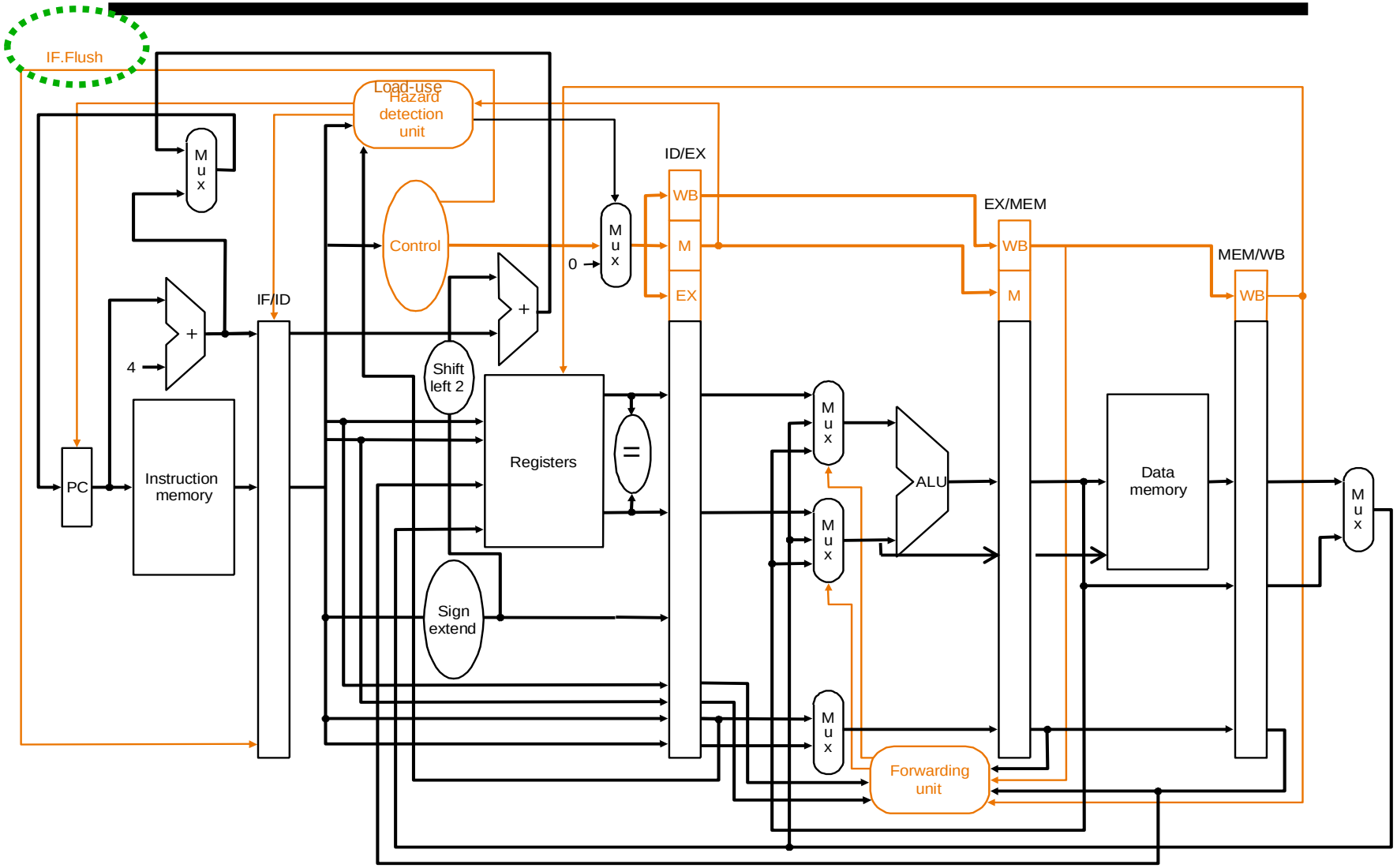
- Example:
  - Assume **"not taken"** when the branch is **taken**
  - **Cancel** instruction 48 (**AND**) because it should not have issued



AND already in IF stage. To cancel/flush it, turn IF/ID pipeline register to 0s, effectively turn it into NOP. "**IF.Flush**" control signal is added.

IF.FLUSH can zero IF/ID pipeline register
Effectively kill the instr after Branch

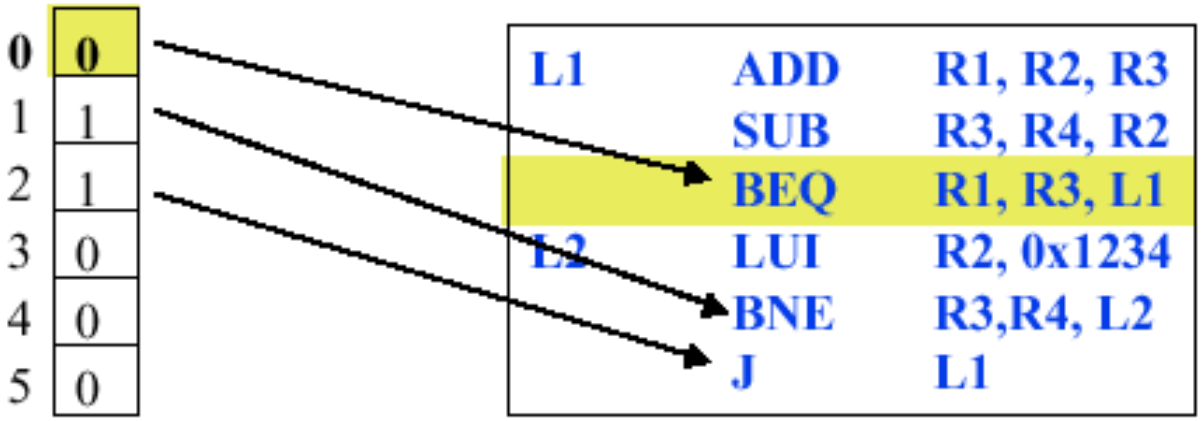
Cancel or Flush Instruction



(2) Dynamic Branch Prediction: 1-bit branch prediction

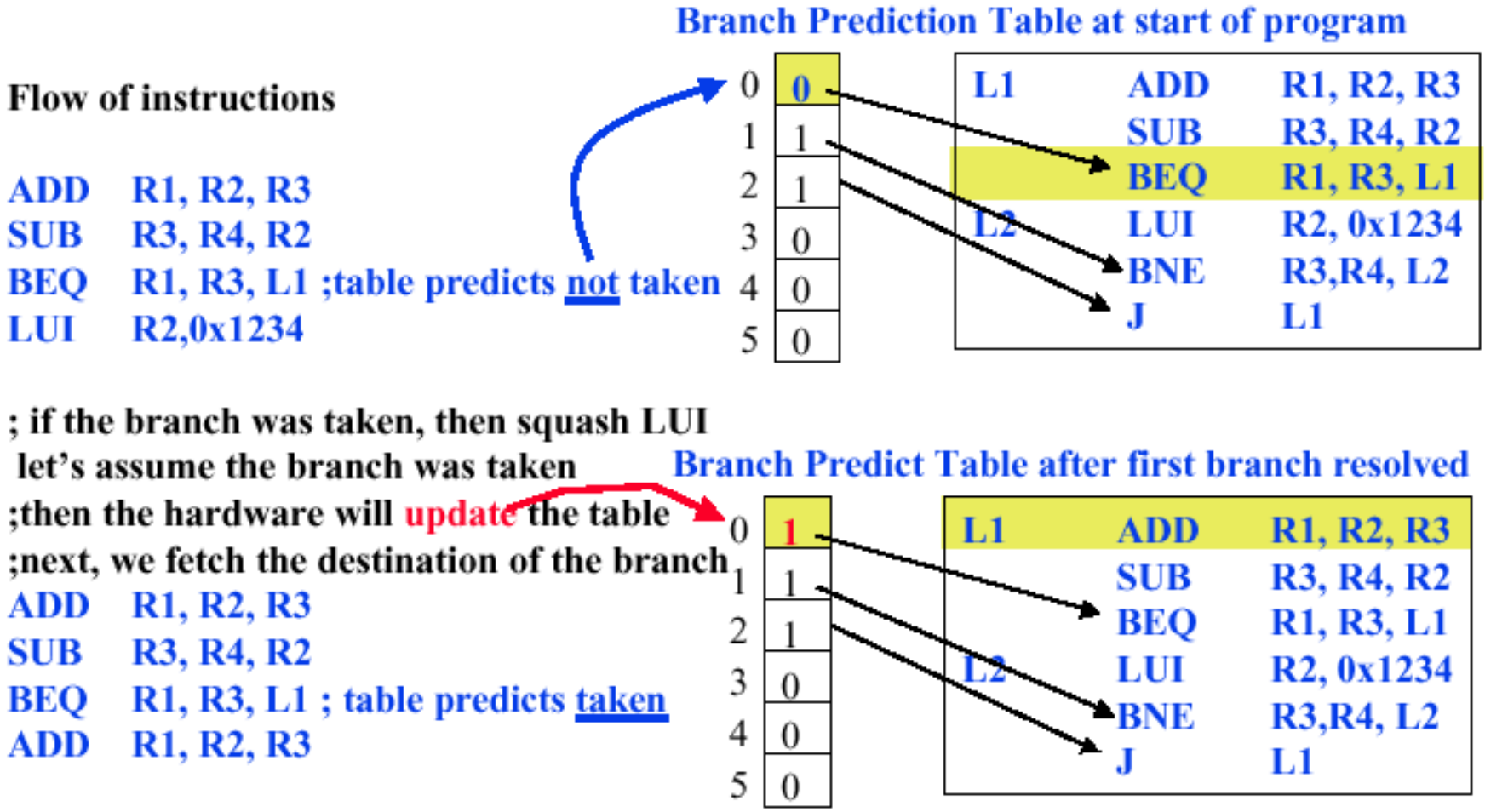
- Hardware has a table of single bits
  - Each entry in the table corresponds to a branch in the program
  - If a bit is **set**, the branch is predicted **taken**
  - If the bit is **not** set (0), the branch is predicted **not** taken

Branch Prediction Table (branch history table)



- 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…)

1-bit Branch Prediction (cont.): simple scheme



Dynamic Branch Prediction

- Dynamic branch prediction uses the previous outcome of a branch to determine future outcomes
  - 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

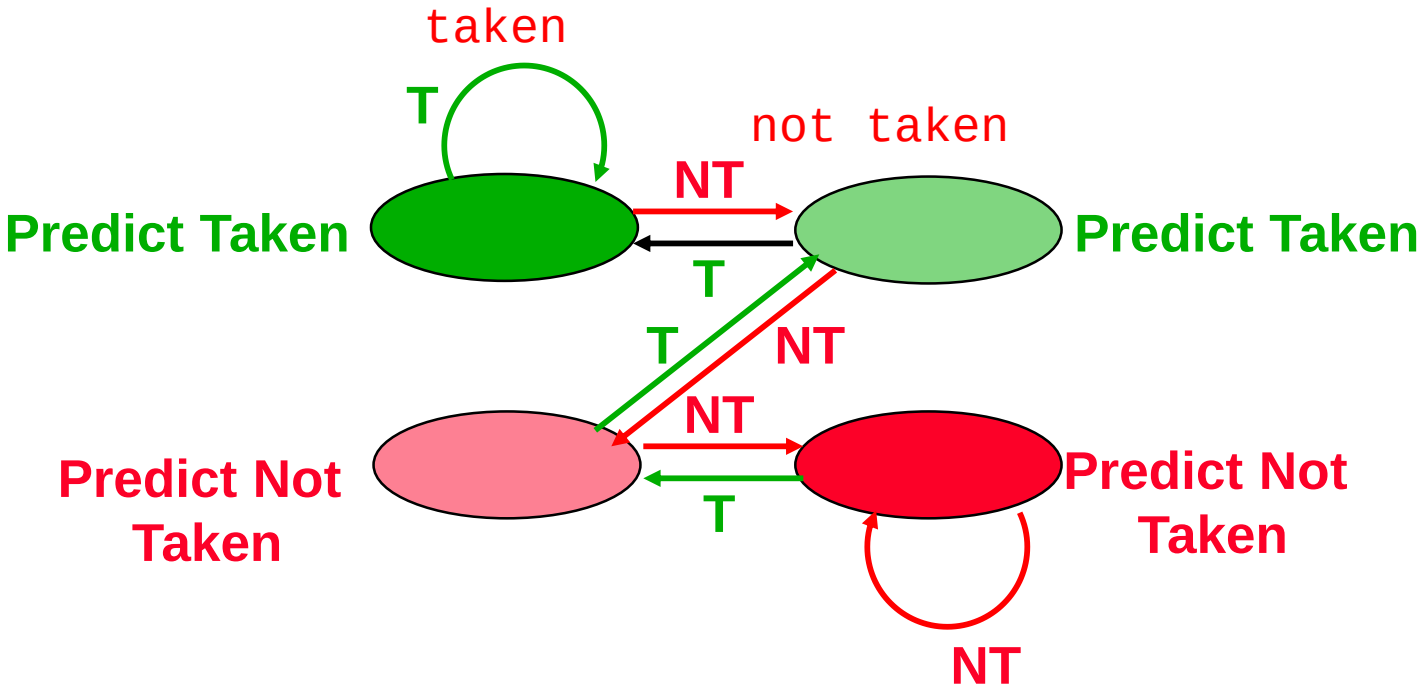
- But what about

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

Hmmm…
Not so clear here, eh?

2-bit Prediction Scheme

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



About Branch Prediction

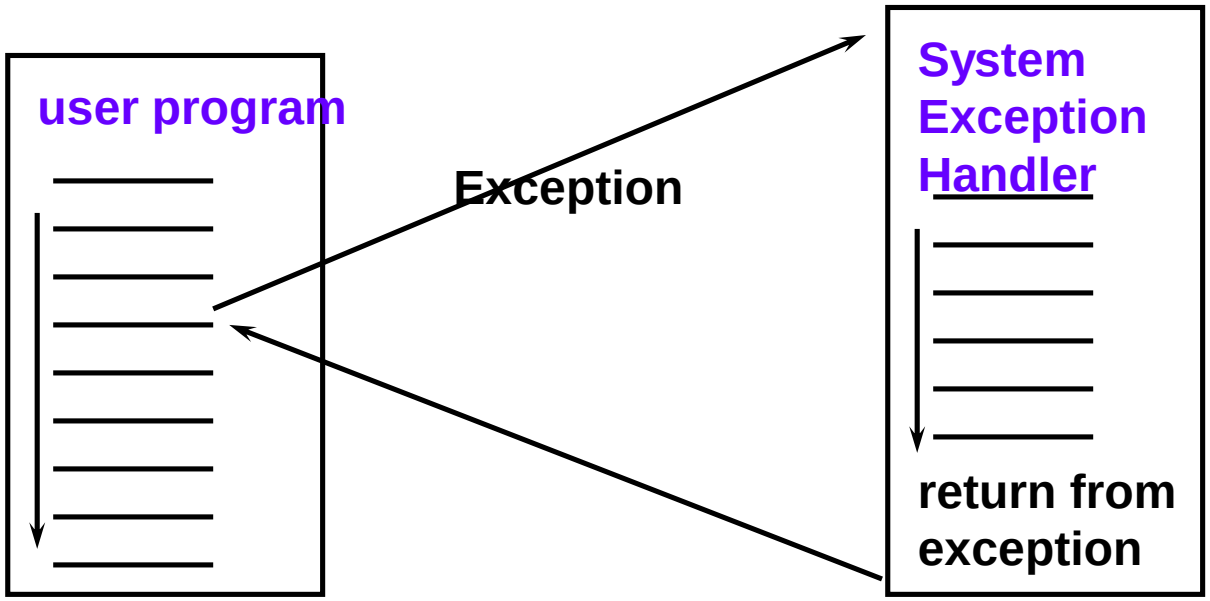
For your eyes only
Details in EE6455

- It's a BIG deal in modern processor design
  - Pipelines are really deep, so stalls are complicated
  - Lots of instructions in flight: may have to squash LOTS if you mispredict
  - May actually have several branch paths executing down the pipe at the same time
- Complicated in superscalar, out-of-order designs
  - Superscalar: lots more instructions in flight in parallel
  - Out-of-order: may rearrange code order to avoid stalls, better utilize free hardware resources in the pipe
- Solutions: Lots of sophisticated prediction hardware
  - But--need to be very careful about cost (size, delay) of this hardware
  - Modern processors speculatively execute a lot of branch paths
  - Modern processors still squash a lot of instructions (it's the only way you can keep ramming stuff down the pipes every cycle; sometimes, you're just wrong)

Summary: Handling Branch Hazard

- **Predict branch always not taken**
  - **Need to add hardware for flushing instructions if wrong**
  - **Branch decision made at MEM => need to flush instructions in IF, ID, EX by changing control values to 0 (flush 3 instns)**
- **Reduce delay of taken branch by moving branch execution earlier in the pipeline**
  - **Move up branch address calculation to ID stage**
  - **Also check branch equality at ID (using XOR or XNOR) by comparing the two registers (rs & rt) read during ID**
  - ➢ **only one instruction to flush if we have to kill it**
  - ➢ **Add a control signal, IF.Flush, to zero the instruction field of IF/ID => turn the instruction behind branch into a NOP**
- **Dynamic branch prediction:**
  - **Predict at IF via branch history using 1 or 2-bit counters in a *branch history table*, indexed by address of branch instr.**
- **Compiler rescheduling, *delay branch***

Exceptions/Interrupts



normal control flow:
sequential, jumps, branches, calls, returns

- **Exception = unprogrammed control transfer**
  - **Operation system takes action to handle exceptions**
    - **must record the address of the offending instruction**
    - **should know the cause and transfer to proper handler**
    - **may save & restore user state if necessary**

Two Types of Exceptions

- **Interrupts:**
  - caused by external events and asynchronous to execution
    => may be handled between instructions
  - simply suspend and resume user program
- **Exceptions:**
  - caused by internal events and synchronous to execution, e.g., exceptional conditions (overflow), errors (parity), faults
  - instruction may be retried and program continued or program may be aborted
- **MIPS convention:**
  - *exception* means any unexpected change in control flow, without distinguishing internal or external
  - use *interrupt* only when the event is externally caused

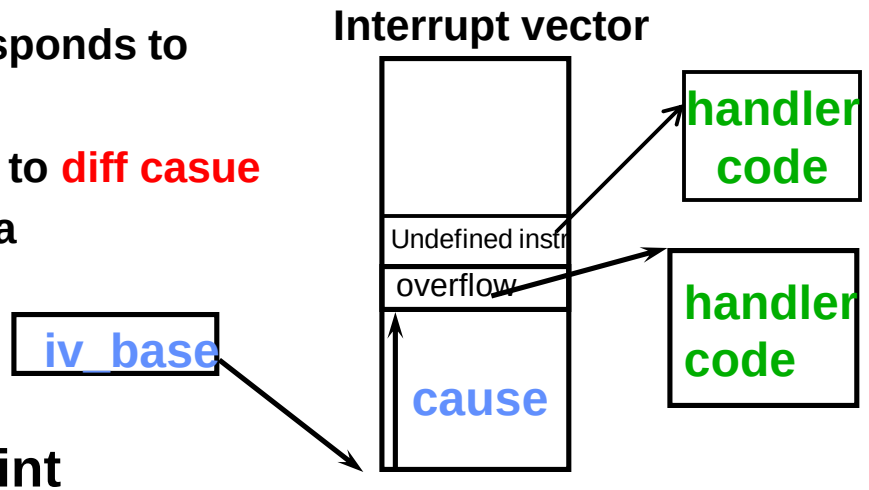
Type of event	*From where?*	*MIPS terminology*
I/O device request	External	Interrupt
Invoke OS from user program	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception

Steps in Executing MIPS => exceptions

1) Ifetch: Fetch Instruction, Increment PC
   - Page fault/Access fault on Instruction fetch? 讀取資料失敗
2) Decode Instruction, Read Registers
   - Undefined Opcode?
3) Execute: Perform operation
   - Overflow? Divided by Zero?
4) Memory: read or write memory
   - Page fault/Access fault on Data access?
5) Write Back: Write Data to Register
   - I/O interrupts?

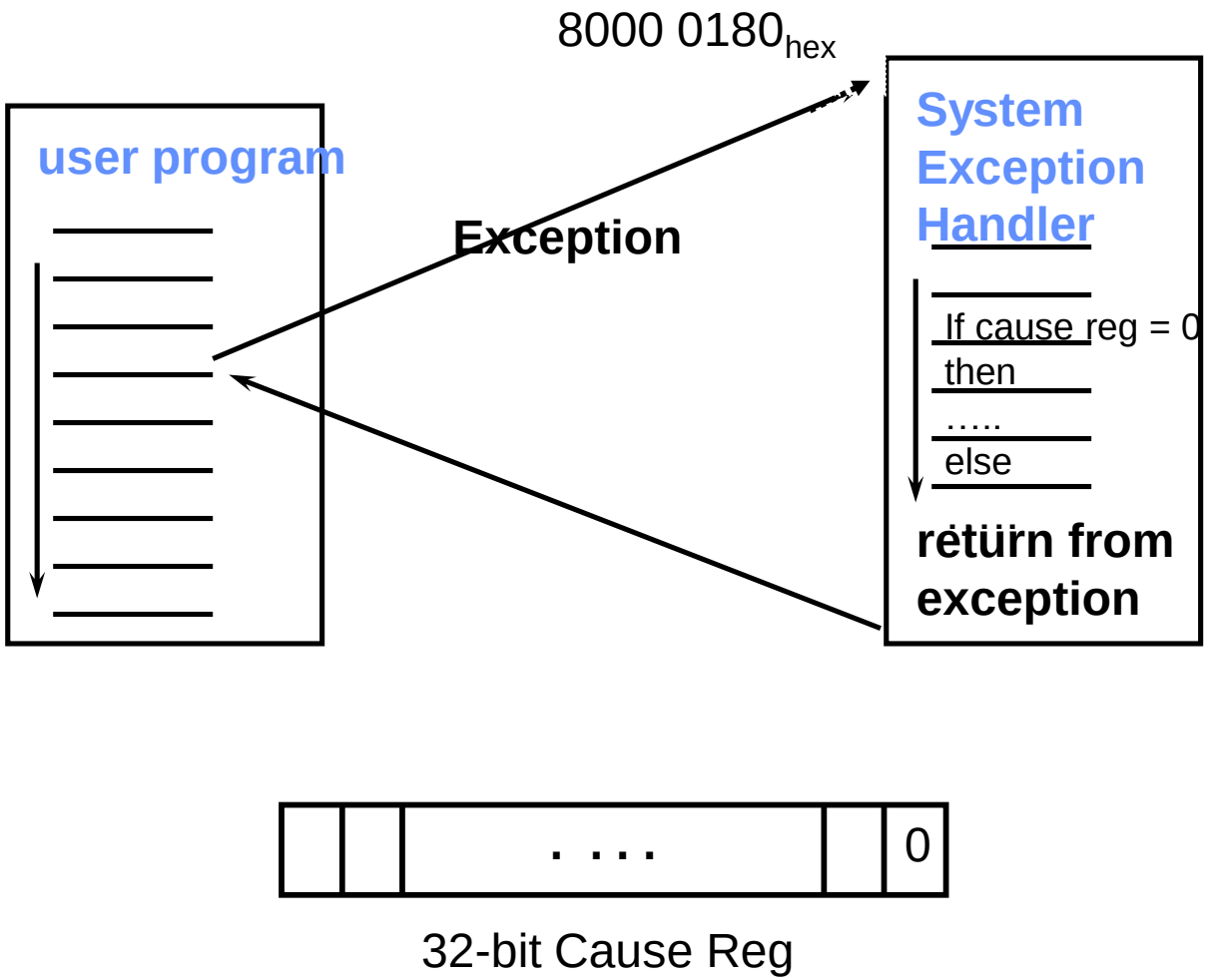
Addressing the Exception Handler

- **Traditional approach: vectored interrupts**
  - **the cause of exception is a vector giving the address of the handler**
  - **Multiple entry points, each corresponds to a cause of exception**
  - **Jump to different addr according to diff casue of exception, each addr points to a separate handler**
  - **370, 68000, Vax, 80x86, . . .**



- **MIPS approach: fixed entry Point**
  - **use a status register (*cause register*) to hold a field to indicate the cause of exception**
  - **All exceptions jump to the same entry point (assuming 8000 0180_{hex}). OS then checks cause reg to determine the cause of exception and take action accordingly**

MIPS Exceptions: fixed entry point



Additions to Our Design

- **EPC (Exception Program Counter): 32-bit register to save address of the affected instruction**
  - **Hold PC that program should jump back to when resuming execution**
- **Cause: 32-bit register to record the cause of exception**
  - **e.x. 10 = *undefined instn* & 12 = *arithmetic overflow***
- **Be able to write *exception address* into PC, assuming at 8000 0180_{hex}**

Handling Exceptions in Our Design

- **Basic actions on exception:**
  - **Save state: save the address of the offending instruction into the *exception program counter* (EPC)**
    - **In reality, PC+4 is saved in EPC**
  - **Transfer control to OS at some specified address**
    - **Assume it is 8000 0180_{hex}**
    - **Then also need to know the *cause* of the exception**
  - **After service, OS can terminate the program or continue its execution using EPC to return (need to move instruction address from EPC to PC to resume execution)**

`move from coprocessor`

"**mfc0 $reg EPC**" to move from EPC to any general purpose reg

`PC<-PC+4-4`

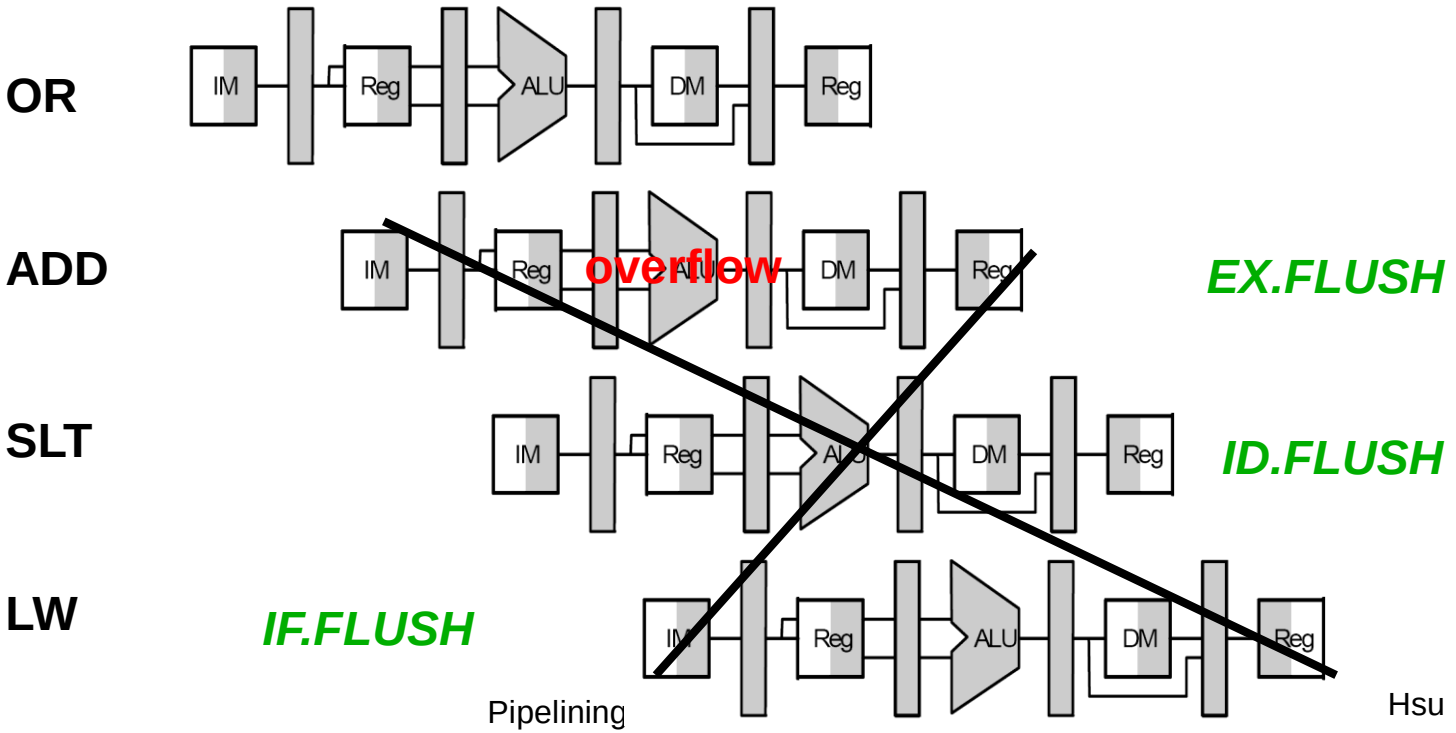
What about Exceptions?

- **5 instructions executing in 5 stage pipeline**
  - **How to stop the pipeline? restart?**
  - **Who caused the interrupt?**
  - **Who to serve first, if multiple interrupts at the same time?**
    - **The earliest instn in pipeline is interrupted first**
- **Important to know in which stage a type of exception can occur => help determine cause**

Stage	*Problem interrupts occurring*
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; memory error**

Handling Exceptions

- *or $13, $2, $6* in MEM stage
  add $1, $2, $1 in EX stage & detect an overflow. Should flush *add, slt, lw*
  slt $15, $6, $7 in ID stage
  lw $16, 50($7) in IF stage
- **Suppose overflow occurs at `add $1,$2,$1` detected at EX stage**



Handling Exceptions

- **Suppose overflow occurs at `add $1,$2,$1` detected at EX stage**
  - **Flush offending instn & instns behind *add* already in the pipeline,**
    - **IF.Flush to zero pipeline register (turn *lw* into *nop*)**
    - **(For *slt* & *add*) ID.Flush & EX.Flush to cause multiplexors to zero control signals (overflow exception detected at EX => flush offending instn. Use EX.FLUSH to prevent it from writing $1 in WB stage)**
  - **Transfer control to exception routine at 8000 0180_{hex} by adding 8000 0180_{hex} to PC input MUX**
  - **Save address of offending instruction in EPC**
    - **Actually save PC + 4 (the one following offending instr)**
    - **Exception routine must first subtract 4 from saved value**

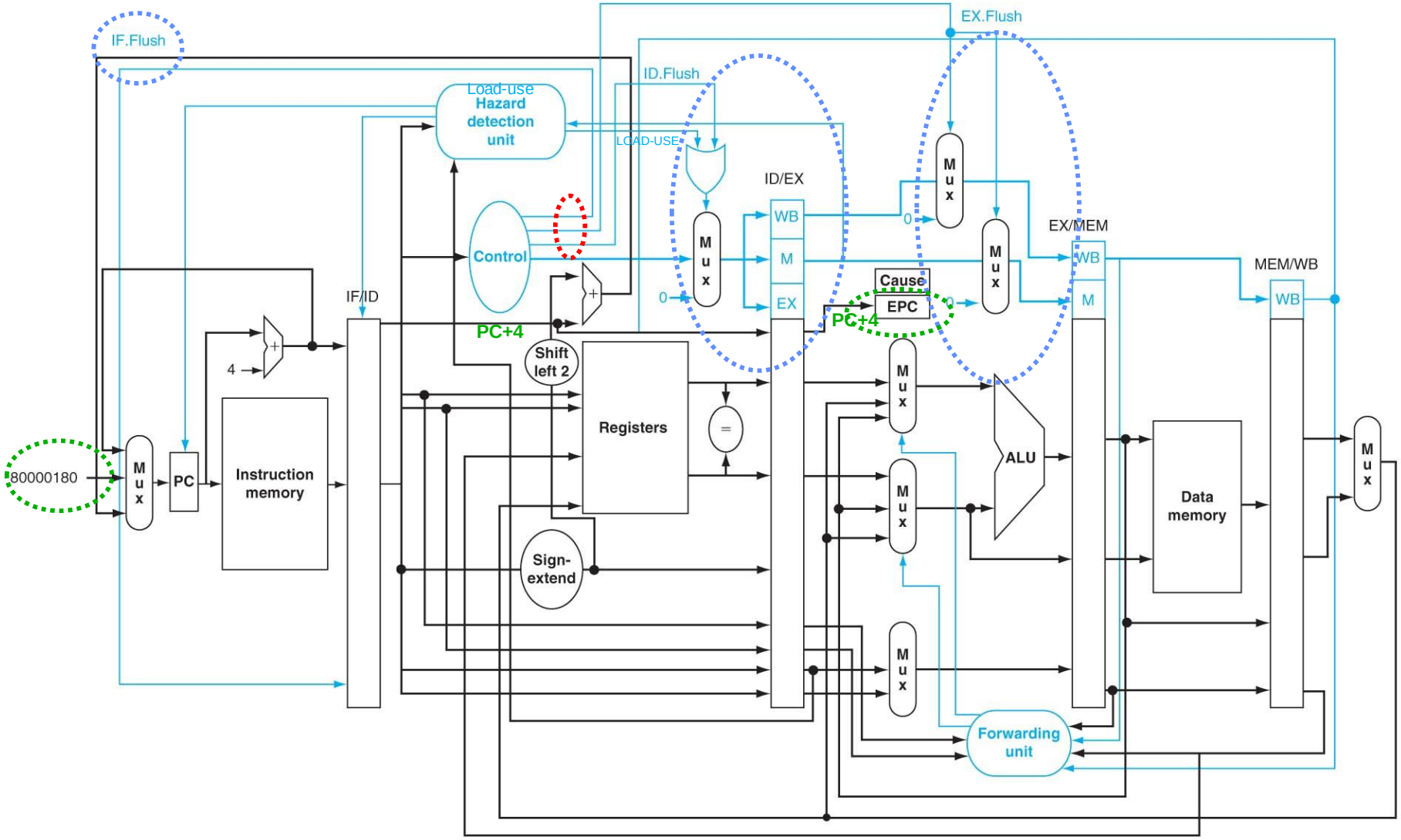


FIG. 4.66 The datapath with controls to handle exceptions. The key additions include a new input with the value 8000 0180hex in the multiplexor that supplies the new PC value; a Cause register to record the cause of the exception; and an Exception PC register to save the address of the instruction that caused the exception. The 8000 0180hex input to the multiplexor is the initial address to begin fetching instructions in the event of an exception. Although not shown, the ALU overflow signal is an input to the control unit.

Summary: Designing a Pipelined Processor

- Go back and examine your datapath and control diagram
  - starting with single-cycle datapath and control
- Partition datapath into stages:
  - IF (instruction fetch), ID (instruction decode and register file read), EX (execution or address calculation), MEM (data memory access), WB (write back)
- Associated resources with states
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage

Summary of Pipeline Basics

- **Pipelining is a fundamental concept**
  - **multiple steps using distinct resources**
- **Utilize capabilities of the datapath by pipelined instruction processing**
  - **start next instruction while working on the current one**
  - **limited by length of longest stage (plus fill/flush)**
  - **detect and resolve hazards**
- **What makes it easy**
  - **all instructions are the same length**
  - **just a few instruction formats**
  - **memory operands appear only in loads and stores**
- **What makes it hard?**
  - **structural hazards: suppose we had only one memory**
  - **control hazards: need to worry about branch instructions**
  - **data hazards: an instruction depends on a previous instruction**