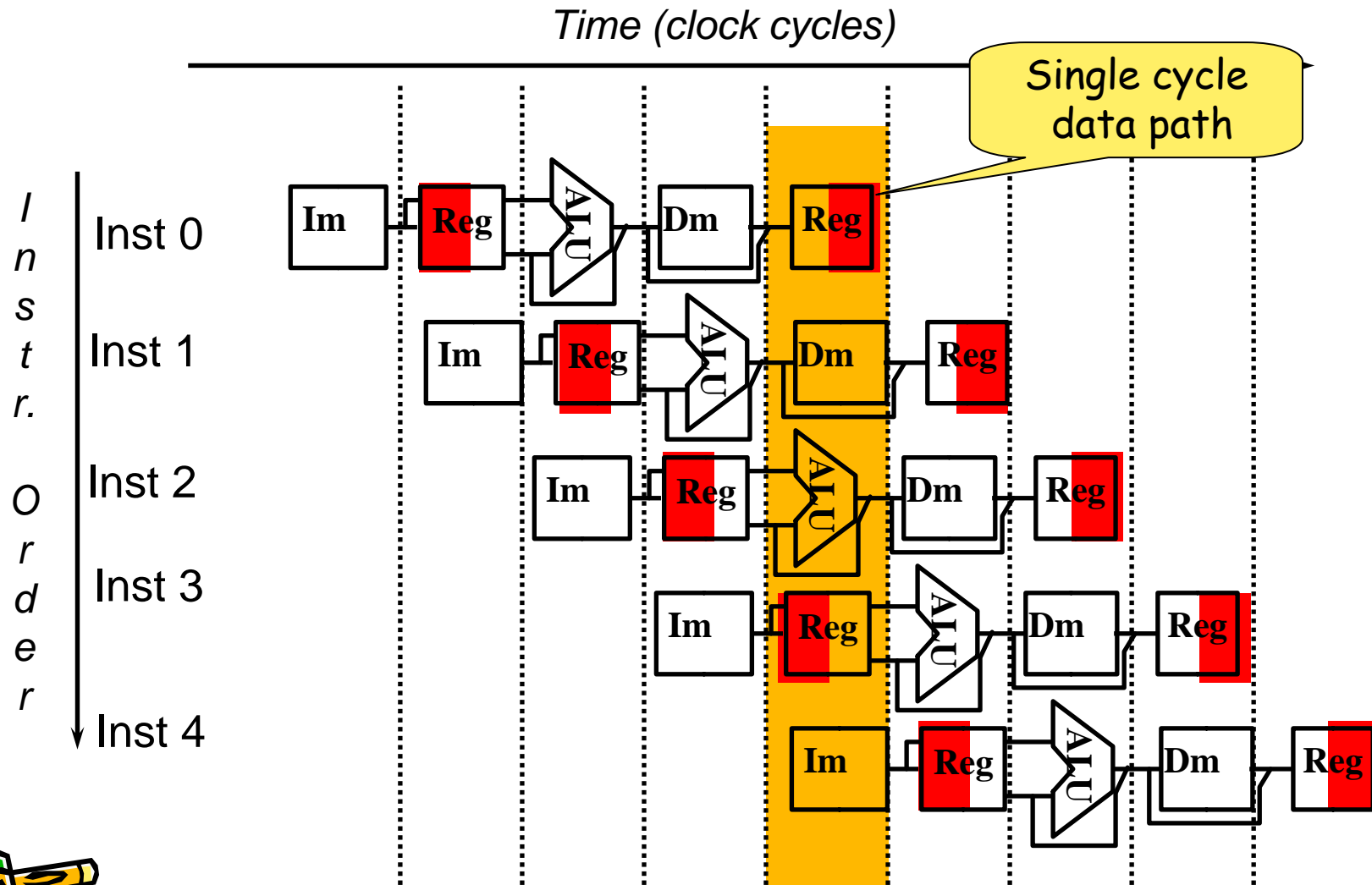


5008: Computer Architecture

Appendix A - Pipelining



Why Pipeline? Because the resources are there!





Pipeline Review

- A pipeline is like an hooked assembly line.
- Pipelining, in general, is not visible to the programmer (vs ILP)
- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest pipeline stage**
- **Multiple tasks** operating simultaneously **using different resources**
- Potential speedup = **Number pipe stages**, if perfectly balanced stage.
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup
- **Stall for Dependences**



Outline

- MIPS – An ISA example for pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion



A "Typical" RISC ISA



- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
base + displacement
 - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

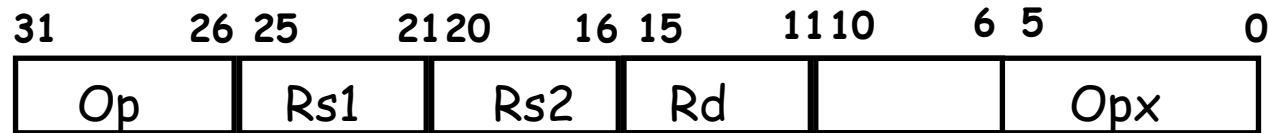




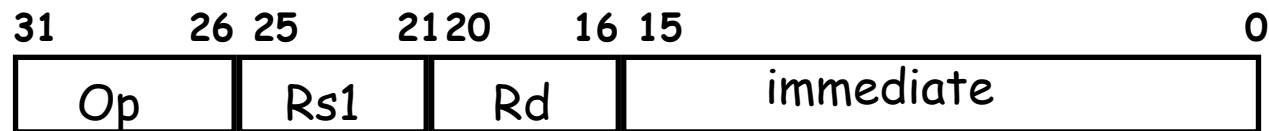
Example: MIPS (- MIPS)



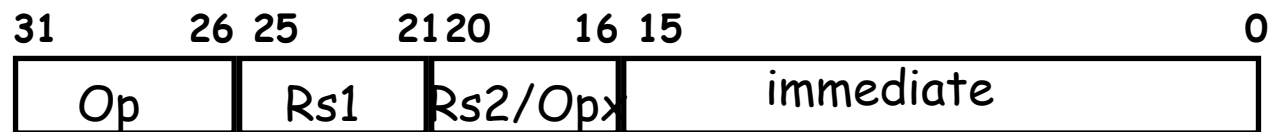
Register-Register



Register-Immediate



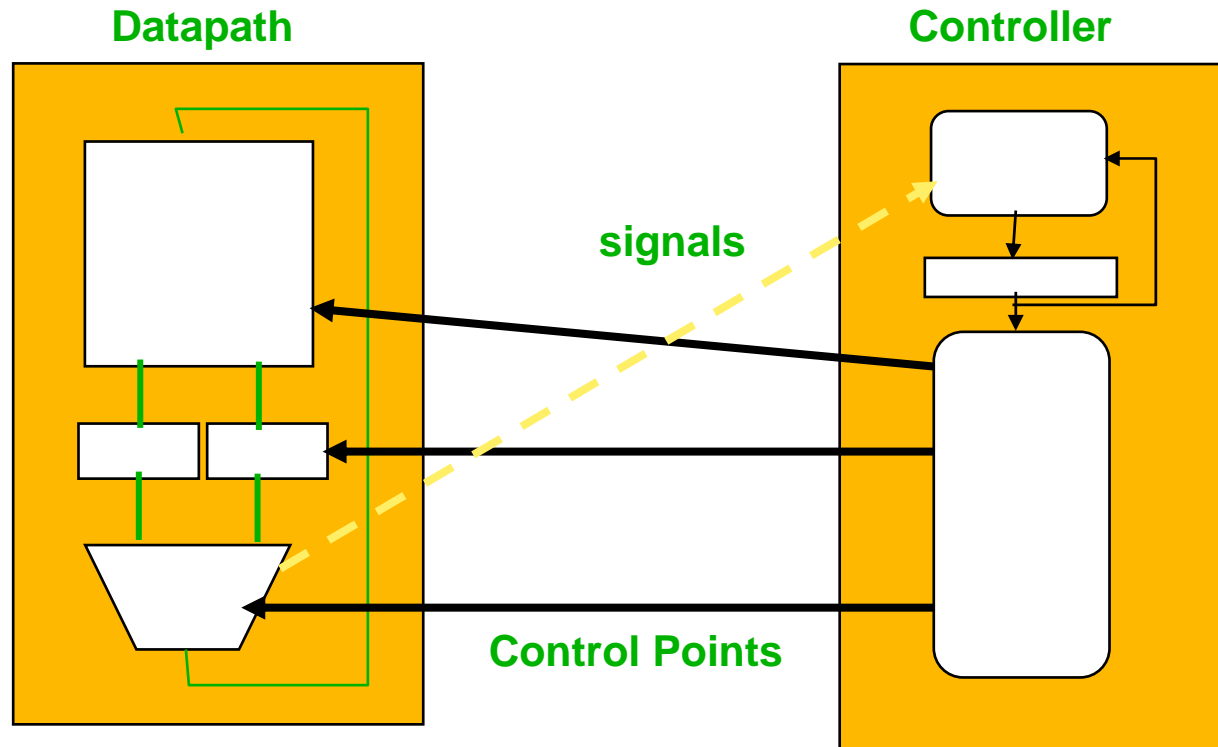
Branch



Jump / Call



Datapath vs Control



- Datapath: Storage, FU, interconnect sufficient to perform the desired functions
 - Inputs are Control Points
 - Outputs are signals
- Controller: State machine to orchestrate operation on the data path
 - Based on desired function and signals



Approaching an ISA



- Instruction Set Architecture
 - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
 - Architected storage mapped to actual storage
 - Function units to do all the required operations
 - Possible additional storage (eg. MAR, MBR, ...)
 - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller



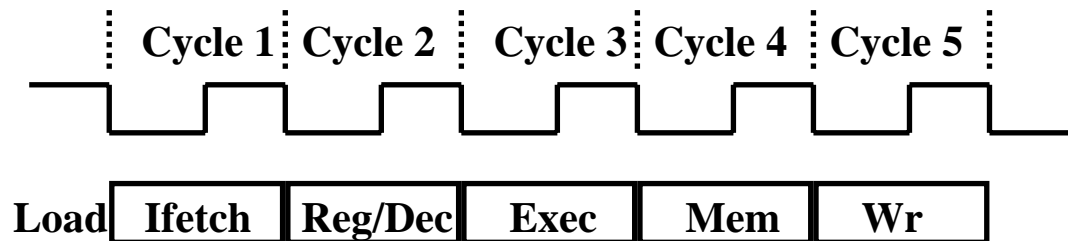
Outline

- MIPS – An ISA example for pipelining -- Read Appendix B
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion





The Five Steps of the Load Instruction



- Every instruction can be implemented in **at most 5 clock cycle**
- **Ifetch**: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Execution and calculate the memory address
- **Mem**: Read the data from the Data Memory
- **Wr**: Write the data back to the register file

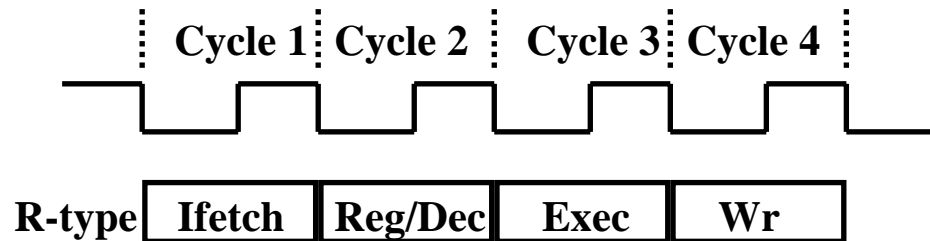
Branch requires ? cycles, Store requires ? cycles, others require ? cycles





The Four Steps of R-type Instruction

does not access data memory...



- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
 - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
 - ALU operates on the two register operands
- Wr: Write the ALU output back to the register file

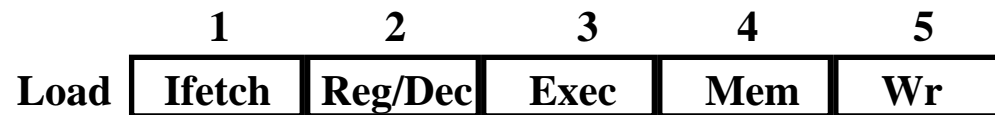




Important Observation

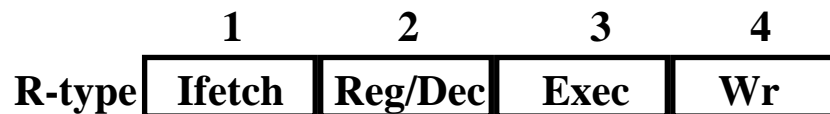
- Each functional unit can only be used **once** per instruction
- Each functional unit must be used at the **same** step for all instructions:

– Load uses Register File's Write Port during its **5th** step



This's what caused the problem

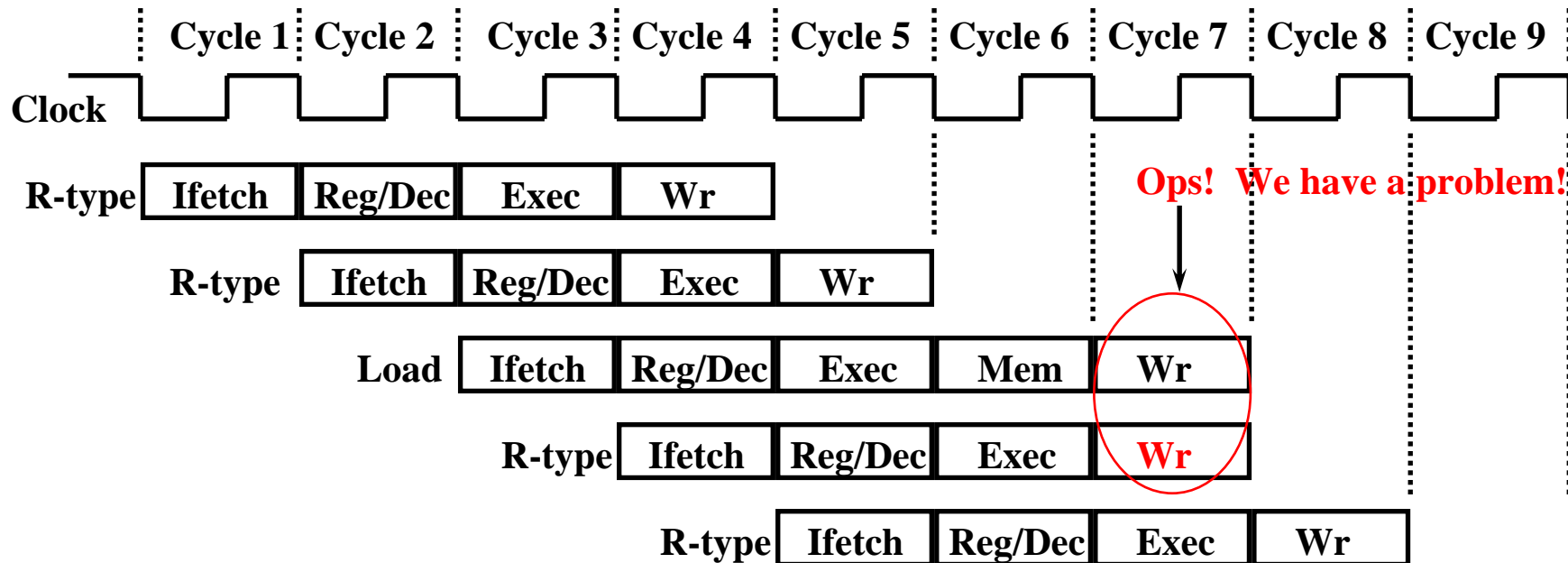
– R-type uses Register File's Write Port during its **4th** step



Structural hazard !!



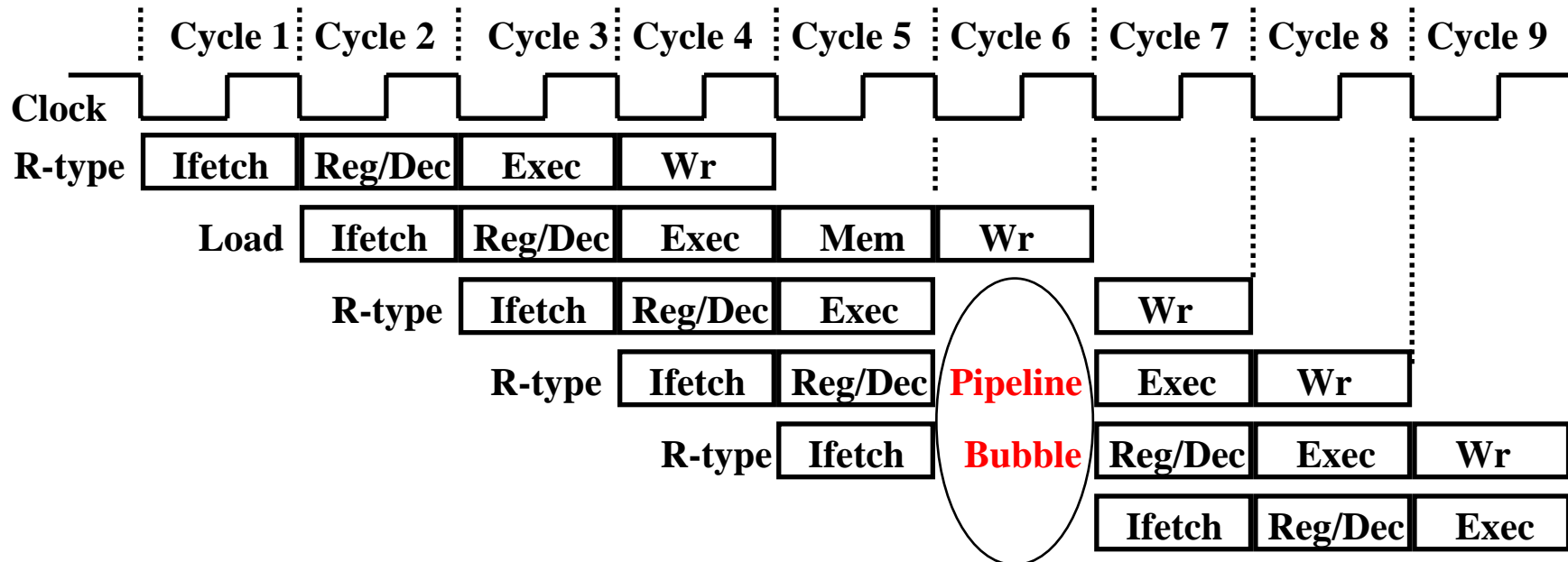
Pipelining the R-type and Load Instruction



- We have pipeline conflict or structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

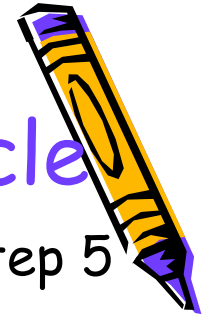


Sol 1: Insert "Bubble" into the Pipeline



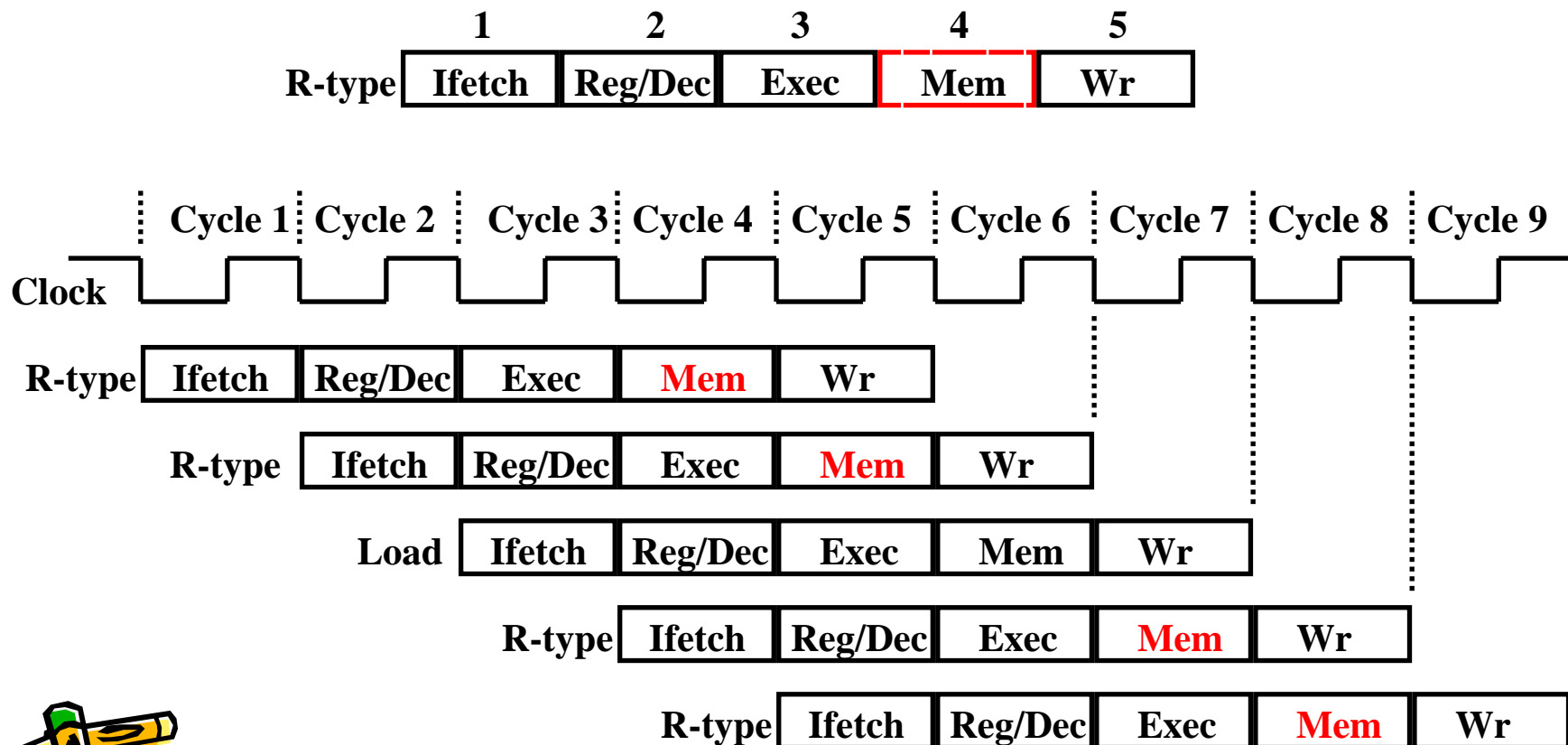
- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex.
 - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!



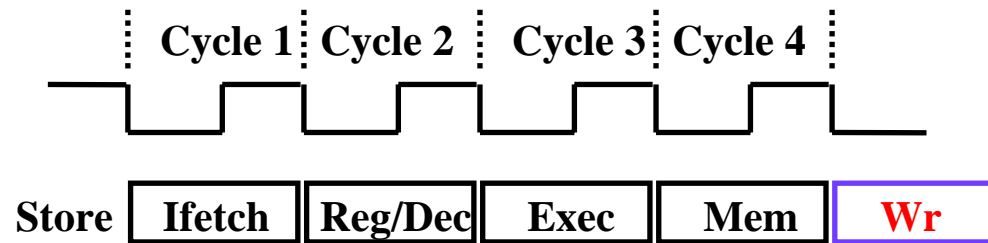


Sol 2: Delay R-type's Write by One Cycle

- Now R-type instructions also use Reg File's write port at Step 5
- Mem step for R-type inst. is a **NOOP** : nothing is being done.



Similarly, the Four Steps of Store

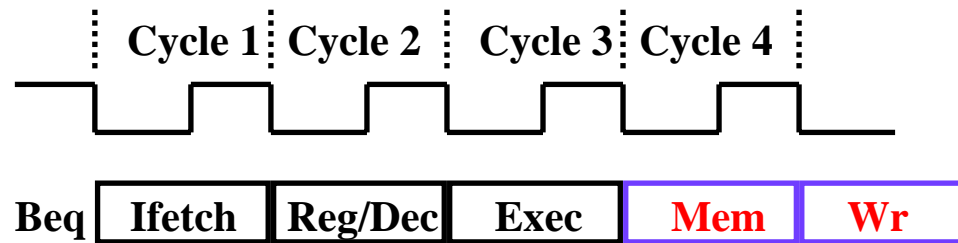


In order to keep our pipeline uniform

- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
 - Update PC
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory



The Three Steps of Beq



- Ifetch: Instruction Fetch
 - Fetch the instruction from the Instruction Memory
- Reg/Dec:
 - Registers Fetch and Instruction Decode
- Exec:
 - compares the two register operand,
 - select correct branch target address
 - latch into PC



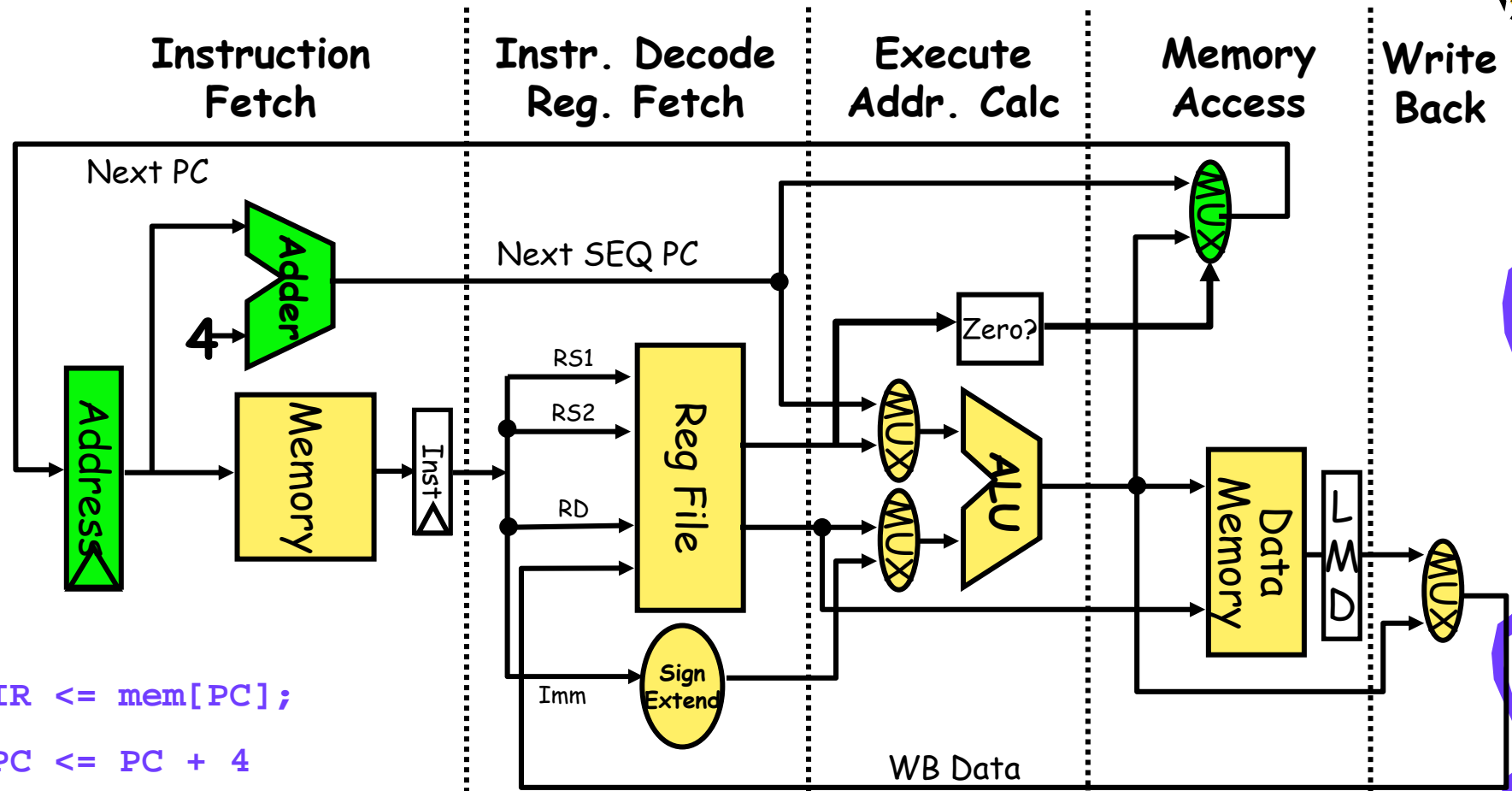


Designing a Pipelined Processor

- Examine the datapath and control diagram
 - Starting with single- or multi-cycle datapath?
 - Single- or multi-cycle control?
- Partition datapath into steps
- Insert pipeline registers between successive steps
- Associate resources with steps
- Ensure that flows do not conflict, or figure out how to resolve
- Assert control in appropriate stage



5 Steps of MIPS Datapath



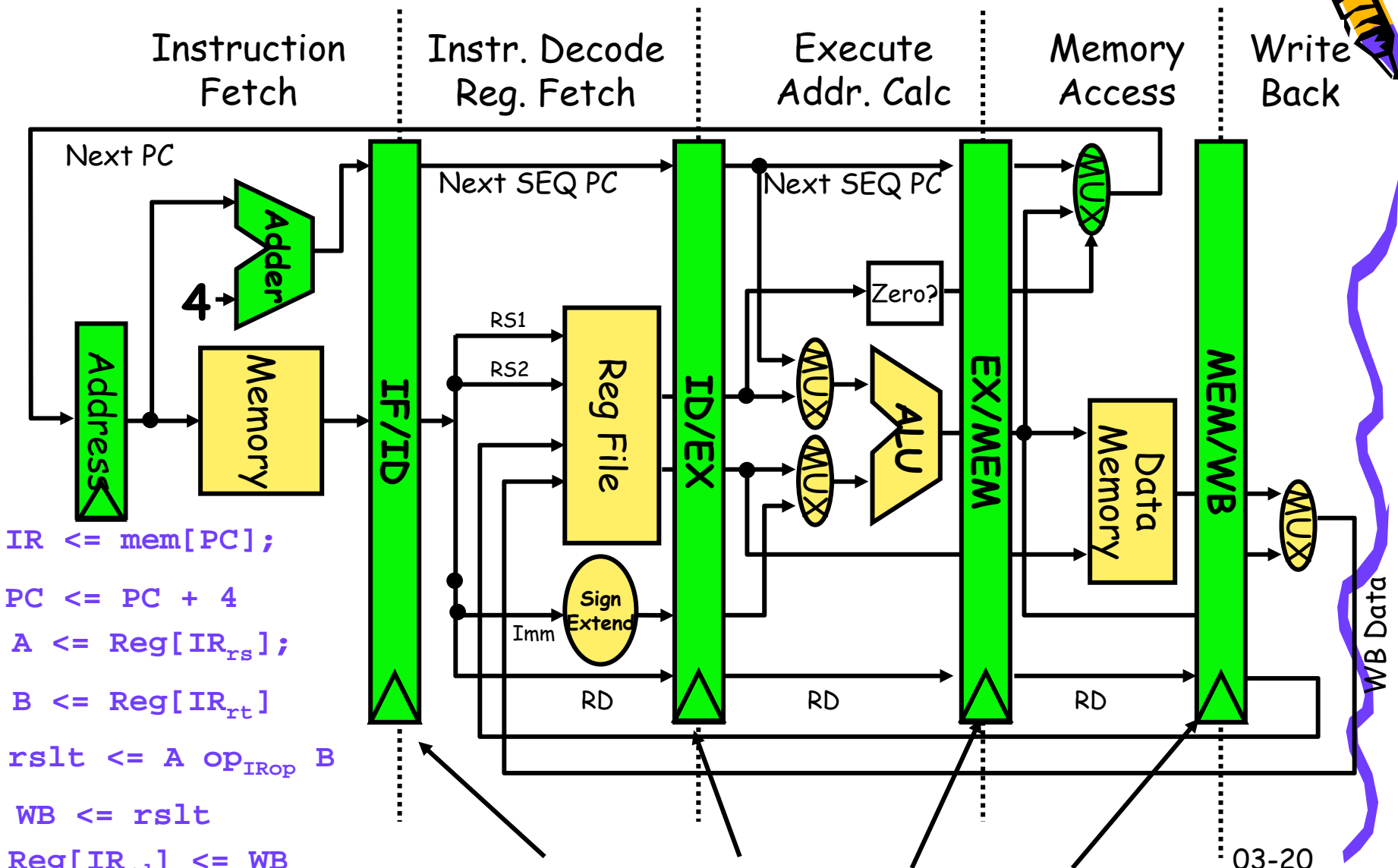
$IR \leftarrow mem[PC];$

$PC \leftarrow PC + 4$

$Reg[IR_{rd}] \leftarrow Reg[IR_{rs}] \text{ op}_{IRop} Reg[IR_{rt}]$



5 Steps of MIPS Datapath



```

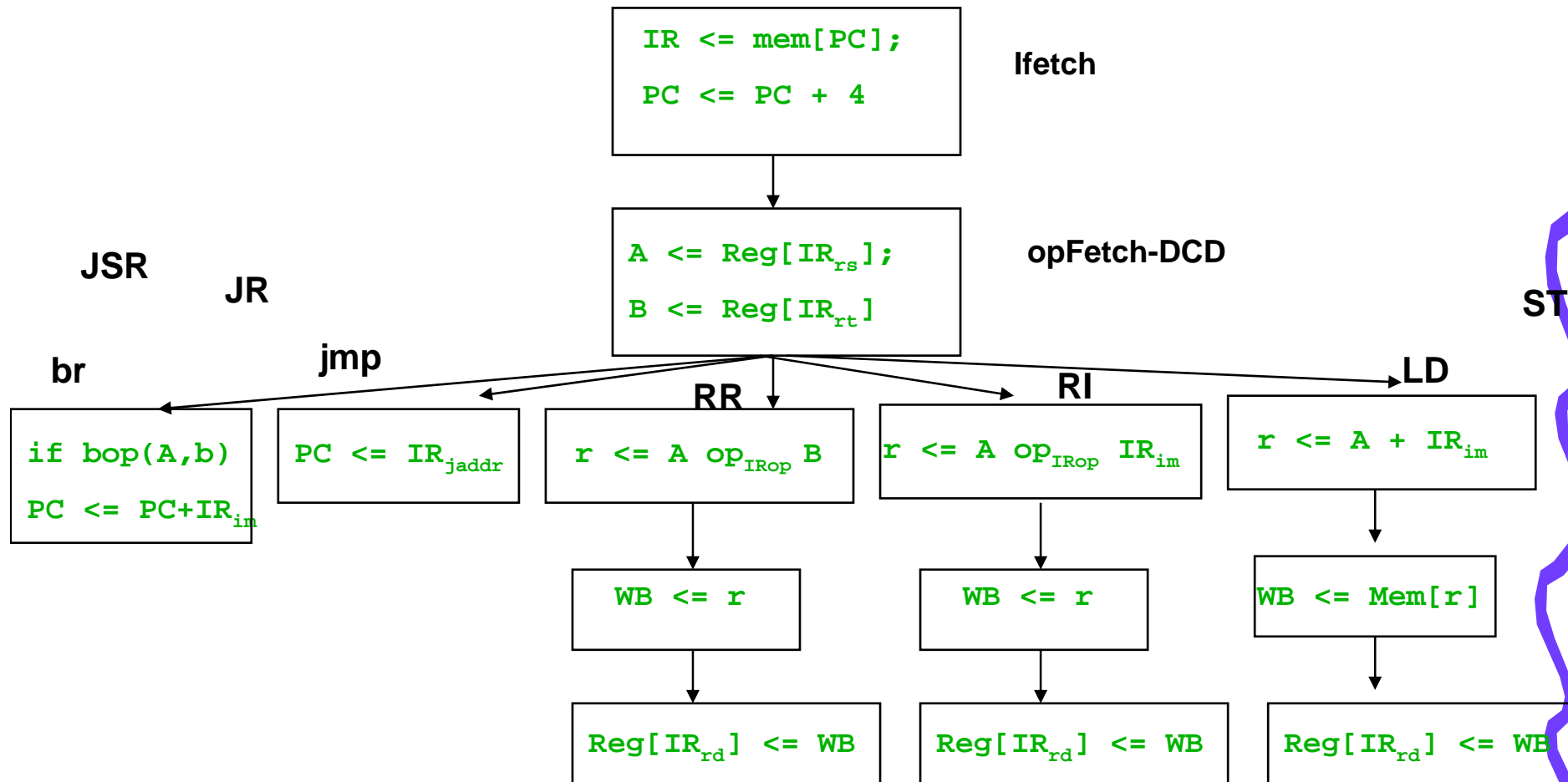
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IRrs];
B <= Reg[IRrt];
rslt <= A opIRop B
WB <= rslt
Reg[IRrd] <= WB
    
```

Pipeline registers (latches)





Inst. Set Processor Controller



ST

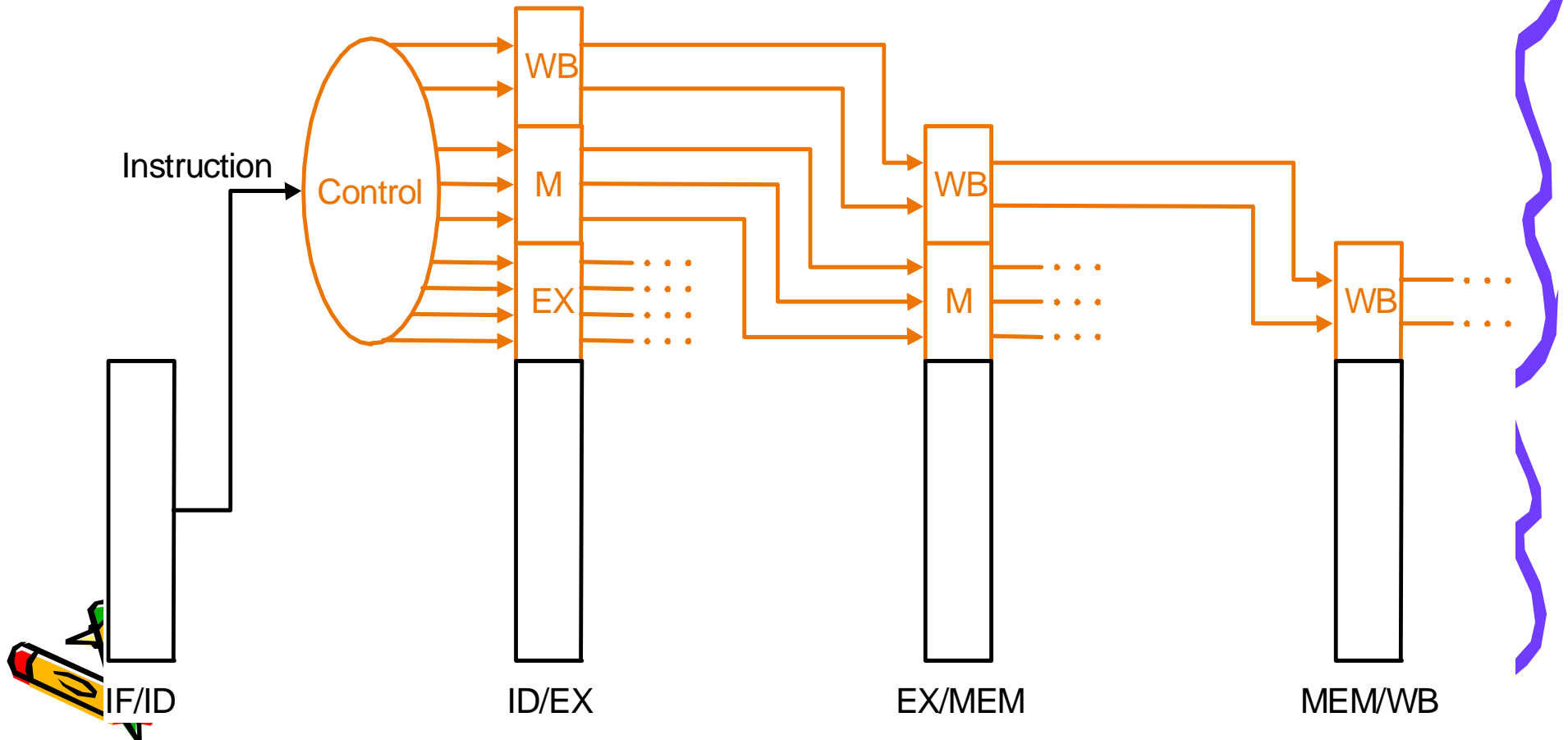


- Use data stationary control
 - local decode for each instruction phase / pipeline stage



Data Stationary Control

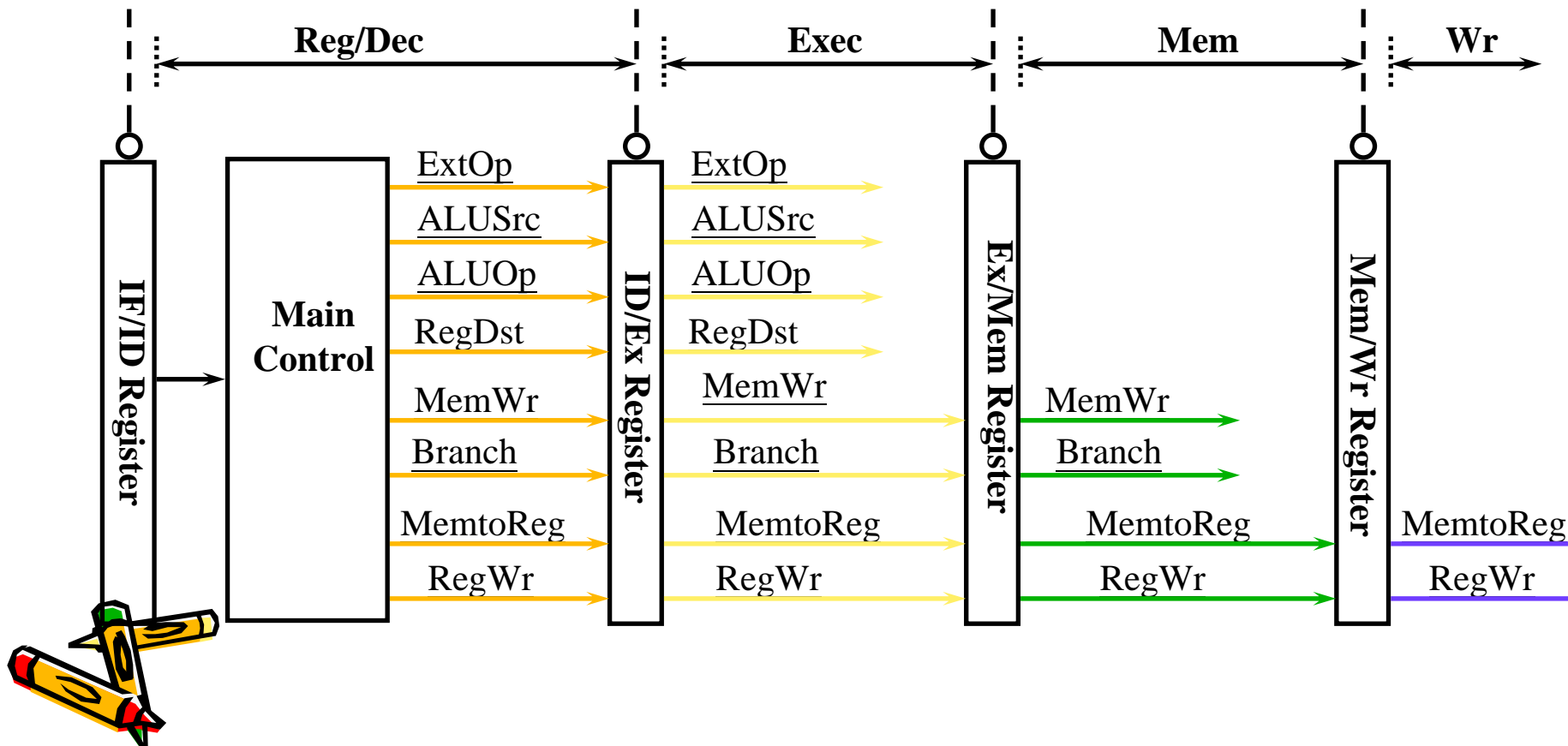
- Pass control signals along just like the data
 - Main control generates control signals during ID



Use of “Data Stationary Control”



- The Main Control generates the control signals during **Reg/Dec**
 - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
 - Control signals for Mem (MemWr Branch) are used 2 cycles later
 - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later

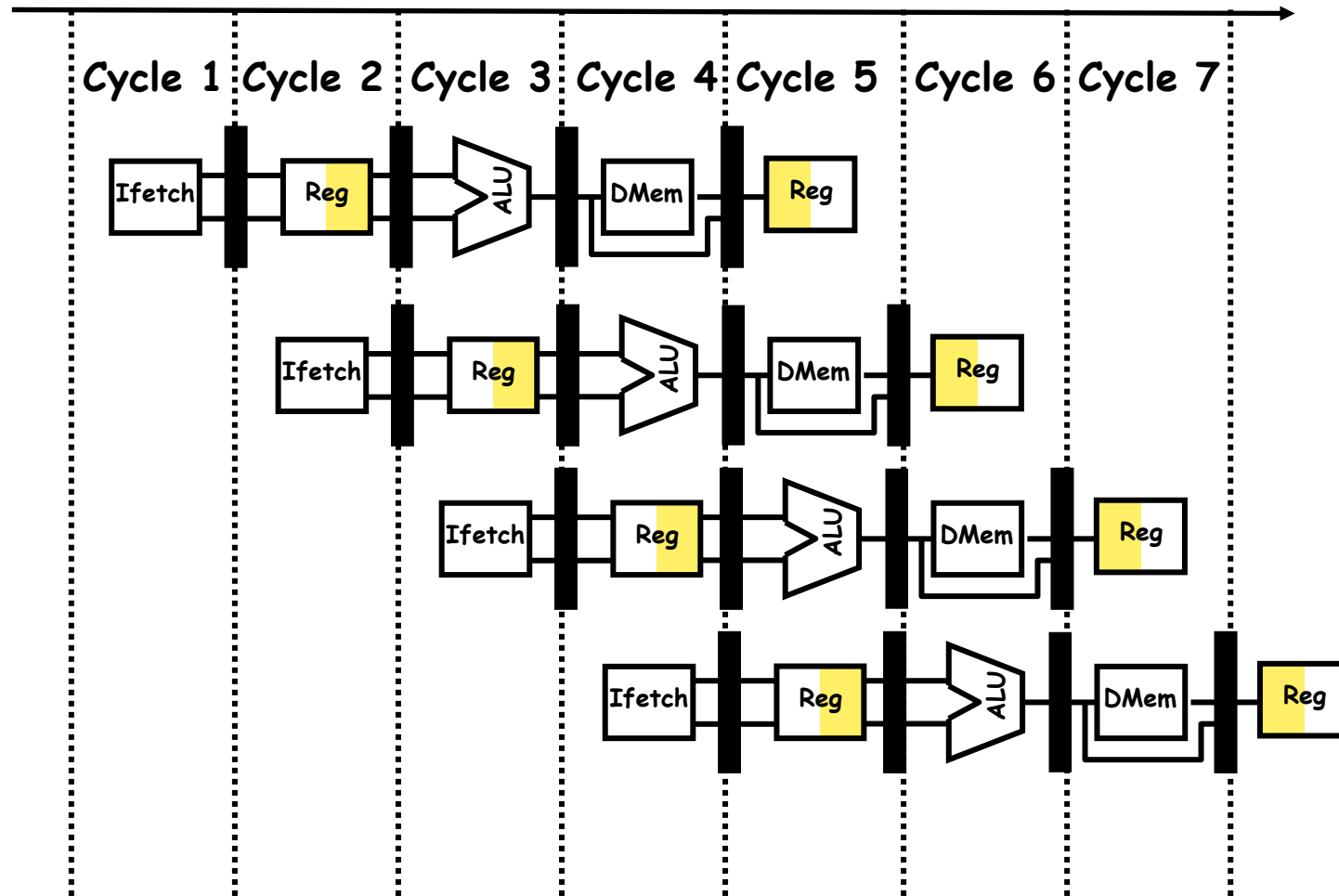


Visualizing Pipelining

Figure A.2, Page A-8

Time (clock cycles)

I
n
s
t
r.
O
r
d
e
r



Outline

- MIPS – An ISA example for pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion



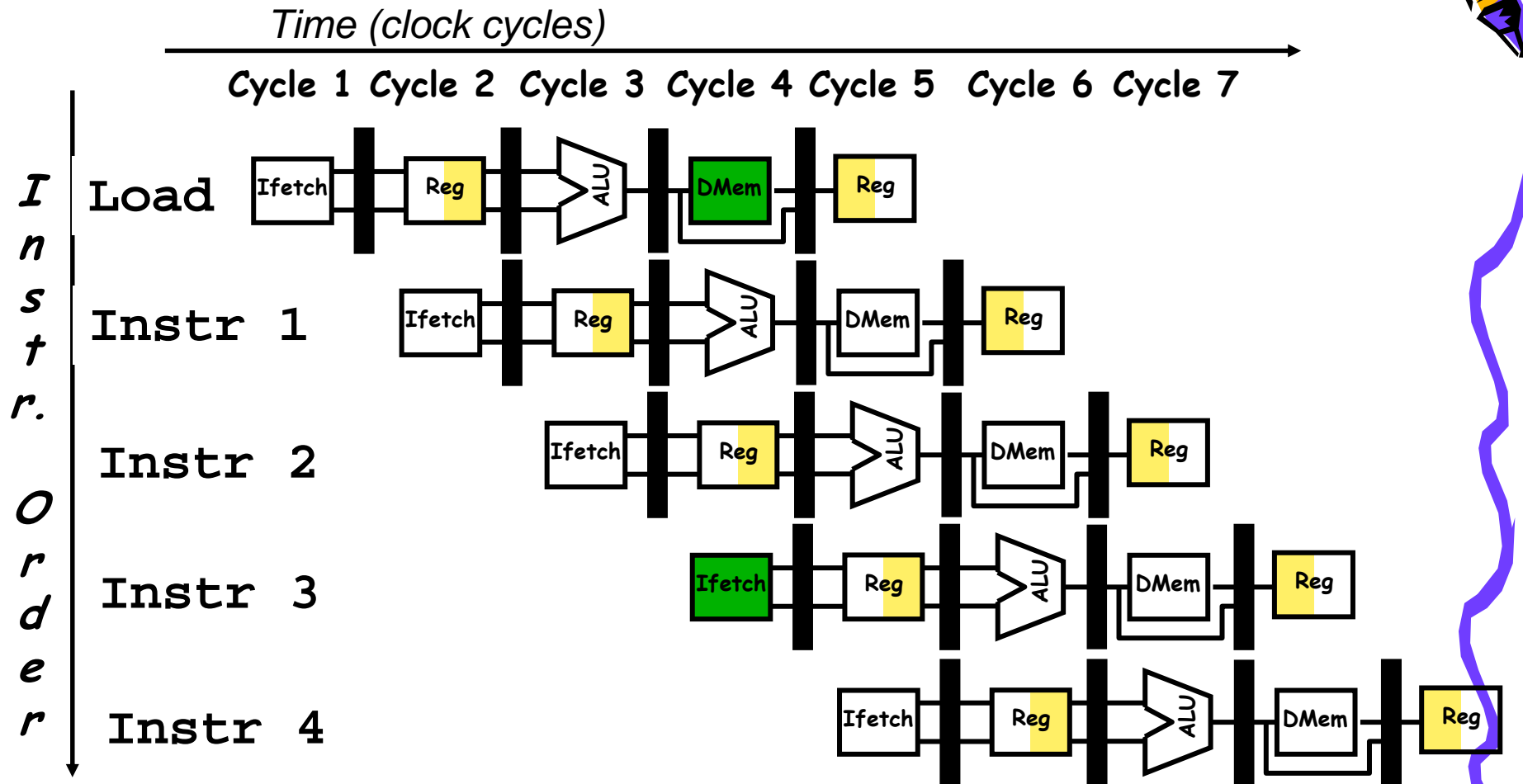


Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).



One Memory Port/Structural Hazards



Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards Limit Performance



- Why? The primary reason is to reduce cost of the unit
- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
 - average CPI = 1.3
 - otherwise resource is more than 100% utilized
- Solution 1: Use separate instruction and data memories
- Solution 2: Allow memory to read and write more than one word per cycle
- Solution 3: Stall





Speed Up Equations for Pipelining

$$\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{CPI_{\text{unpipelined}}}{CPI_{\text{pipelined}}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

for balanced pipelining

For simple RISC pipeline, $CPI = 1$:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$





Example: One or Two Memory Ports?

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth}\end{aligned}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$$

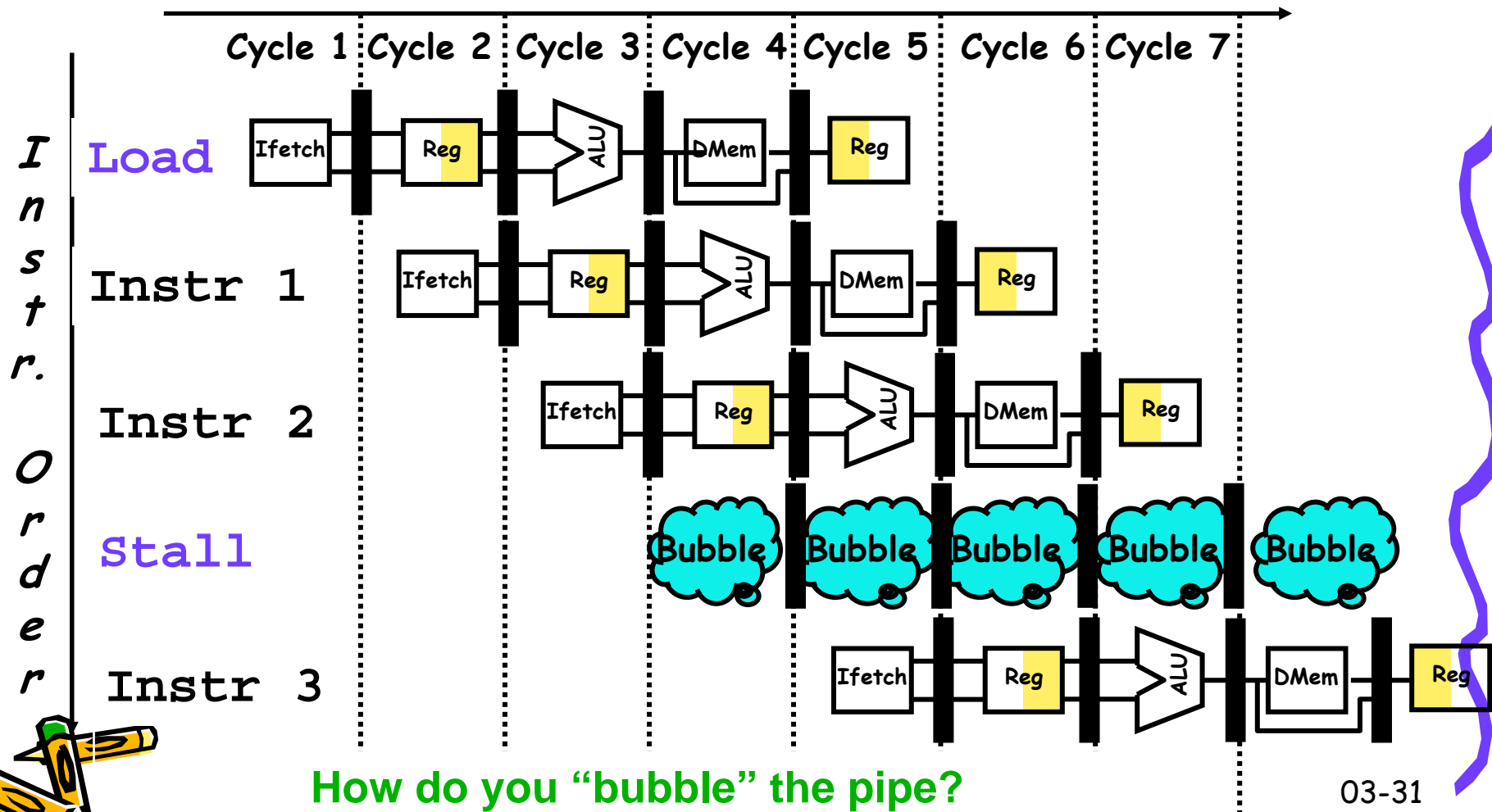
- Machine A is 1.33 times faster





One Memory Port Structural Hazards

Time (clock cycles)



Handling Stalls



- How to stall?
 - Stall instruction in IF and ID: **not change PC and IF/ID**
=> the stages re-execute the instructions
 - What to move into EX: **insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0**
 - as control signals propagate, all control signals to EX, MEM, WB are de-asserted and no registers or memories are written



Data Hazard Problem

- Due to the overlapped instructions.

Example: r1 cannot be read by other instructions before it is written by the add.

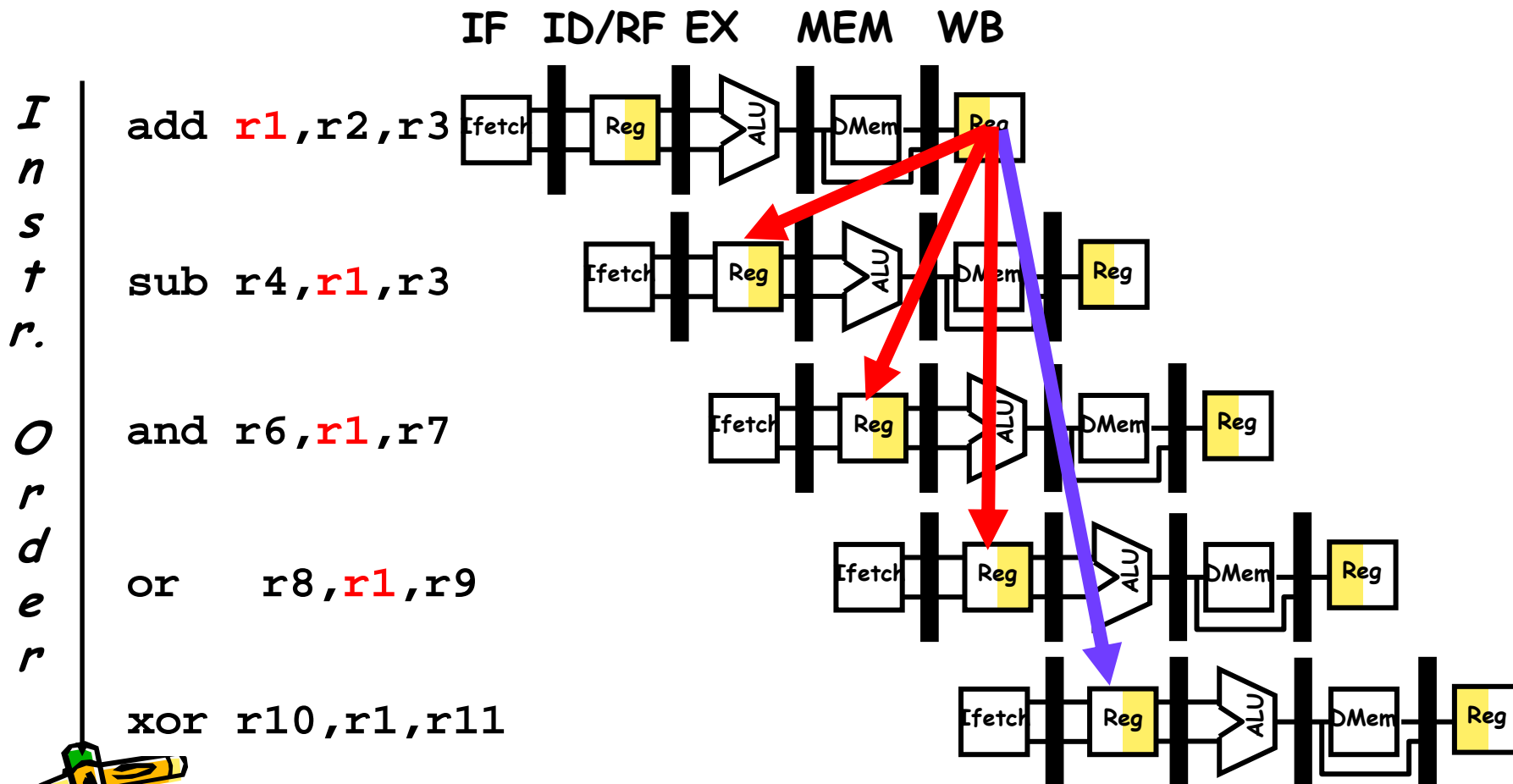
```
add   r1, r2, r3
sub   r4, r1, r3
and   r6, r1, r7
or    r8, r1, r9
xor   r10, r1, r11
```



RAW Hazards on R1



Time (clock cycles) Dependencies backwards in time are hazards





Types of Data Hazards

Three types: (inst. i1 followed by inst. i2)

- **RAW (read after write):** dependence
i2 tries to read operand before i1 writes it
- **WAR (write after read):** anti-dependence
i2 tries to write operand before i1 reads it
 - Gets wrong operand, e.g., auto-increment addr.
 - Can't happen in MIPS 5-stage pipeline because:
 - All instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5
- **WAW (write after write):** output dependence
i2 tries to write operand before i1 writes it
 - Leaves wrong result (i1's not i2's); occur only in pipelines that write in more than one stage
 - Can't happen in MIPS 5-stage pipeline because:
 - All instructions take 5 stages, and writes are always in stage 5
 - Out of order executions may suffer this data dependence



WAR Data Hazard



- Write After Read (WAR)
Instr_J writes operand before Instr_I reads it

⤵
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7

- Called an “anti-dependence” by compiler writers.
This results from reuse of the name “r1”.
- Can’t happen in MIPS 5 stage pipeline because:
- Can happen in between a shorter (Int) pipeline and a longer (FP) pipeline
- WAR hazards can happen if instructions execute out of order or access data late



No WAR Hazards on r1

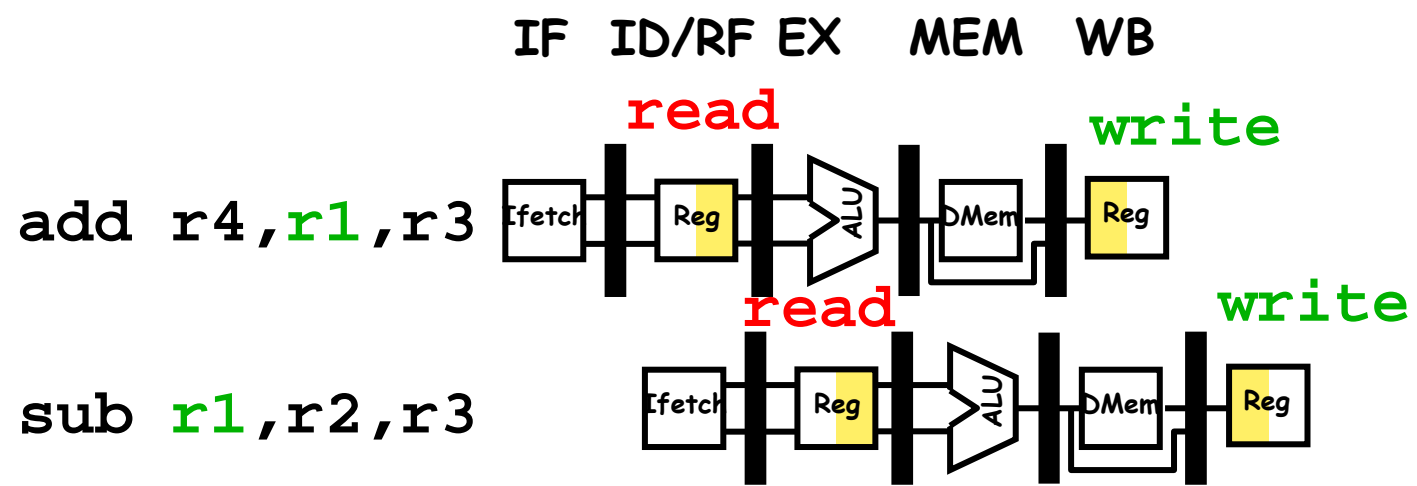


Time (clock cycles)



I
n
s
t
r.

O
r
d
e
r

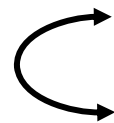


WAW Data Hazard



Write After Write (WAW)

Instr_J writes operand before Instr_I writes it.

 I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “r1”.
- Can't happen in 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes





No WAW Hazards on R1



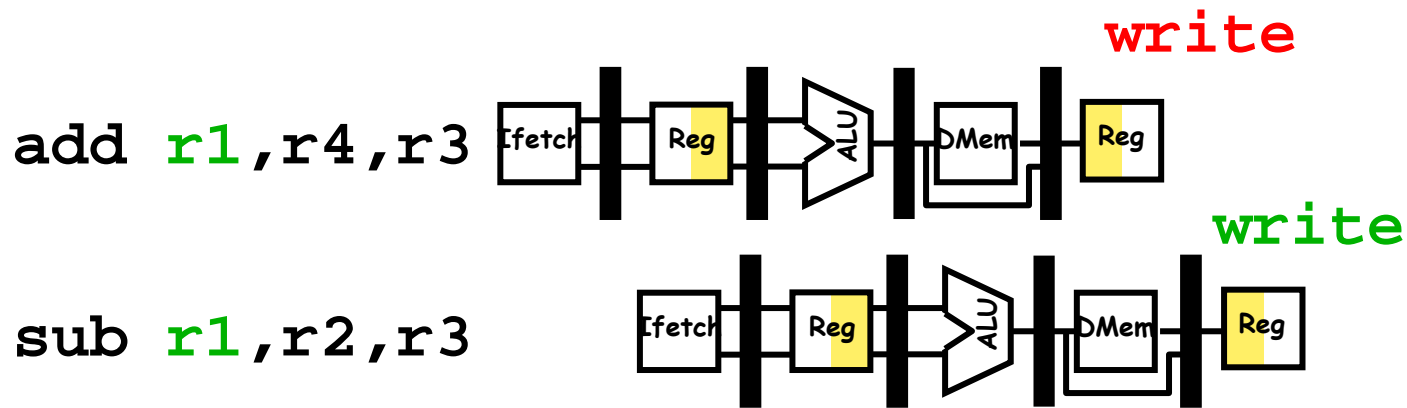
Time (clock cycles)



I
n
s
t
r.

O
r
d
e
r

IF ID/RF EX MEM WB





Data Forwarding to Avoid Data Hazard

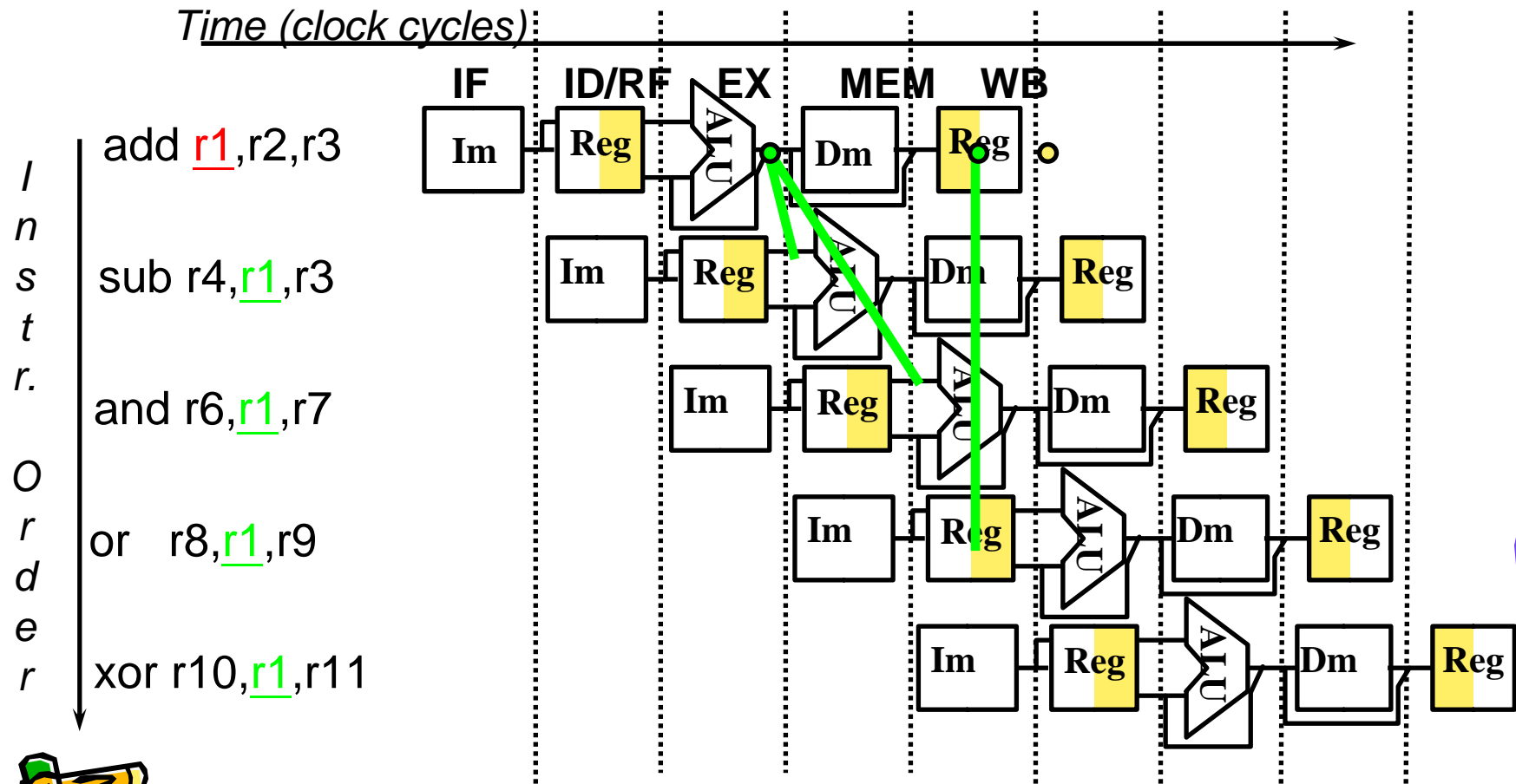
- With **data forwarding** (also called **bypassing** or **short-circuiting**), data is transferred back to earlier pipeline stages before it is written into the register file.
 - Instr i: add **r1**,r2,r3 (result ready after EX stage)
 -
 - Instr j: sub r4,**r1**,r5 (result needed in EX stage)
- This either eliminates or reduces the penalty of RAW hazards.
- To support data forwarding, **additional hardware is required**.
 - Multiplexors to allow data to be transferred back
 - Control logic for the multiplexors





Data Hazard Solution

“Forward” result from one stage to another

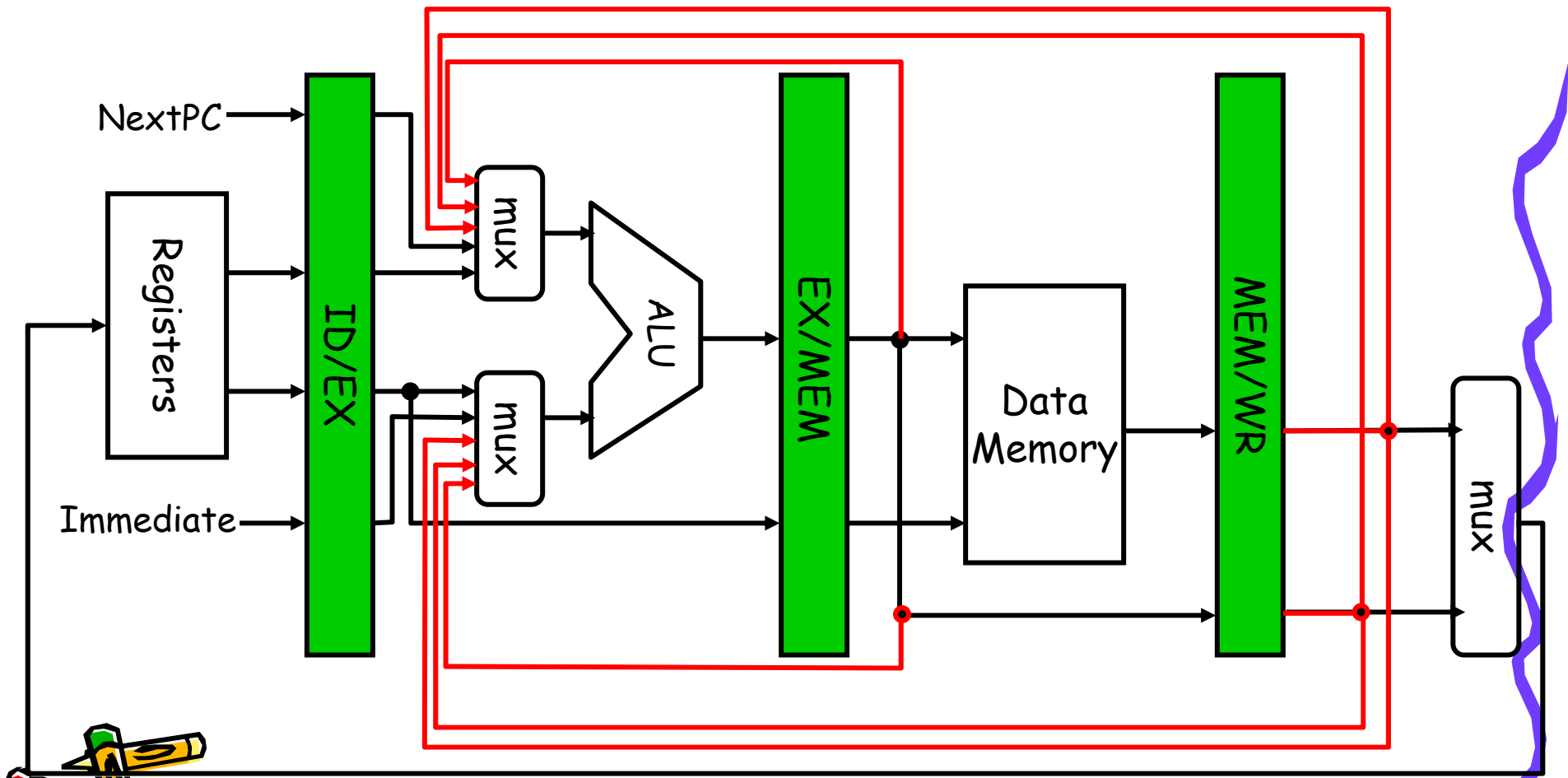


“or” OK if define read/write properly





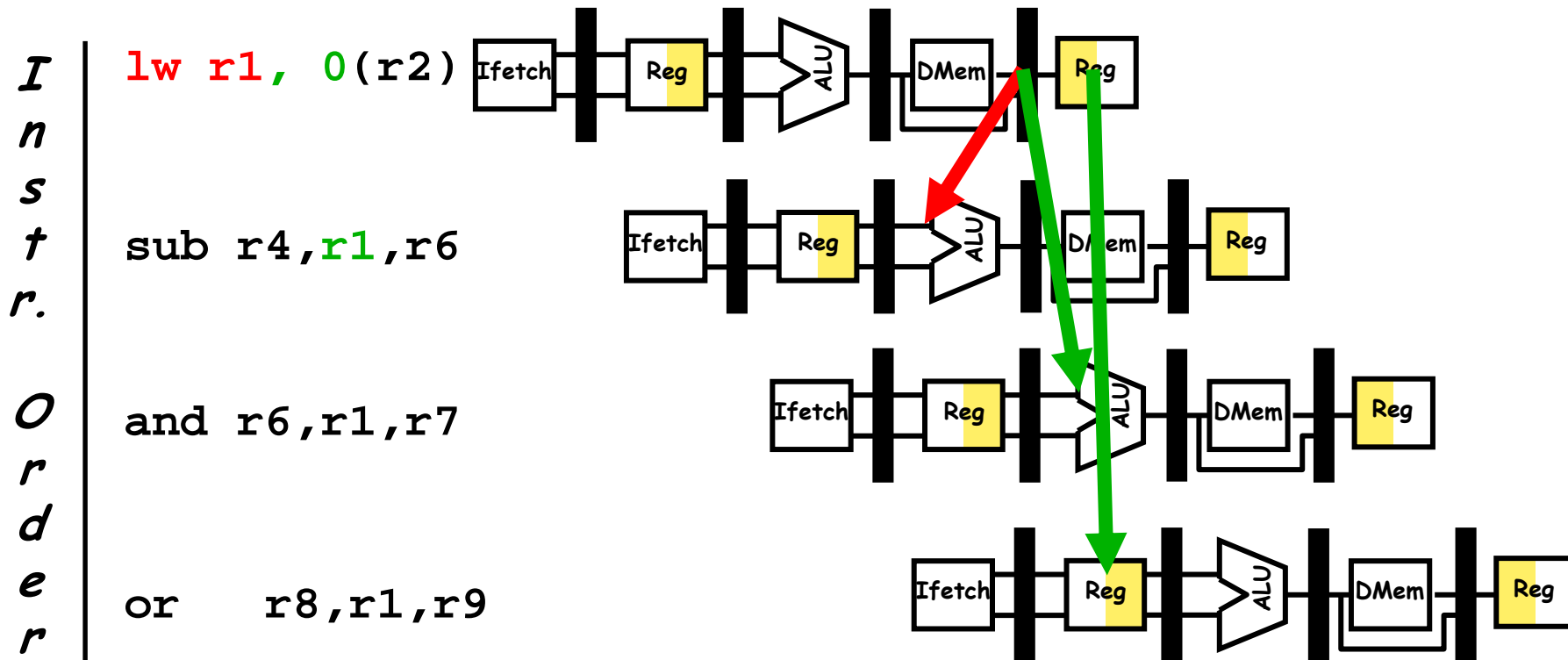
HW Change for Forwarding





Data Hazard Even with Forwarding

Time (clock cycles)



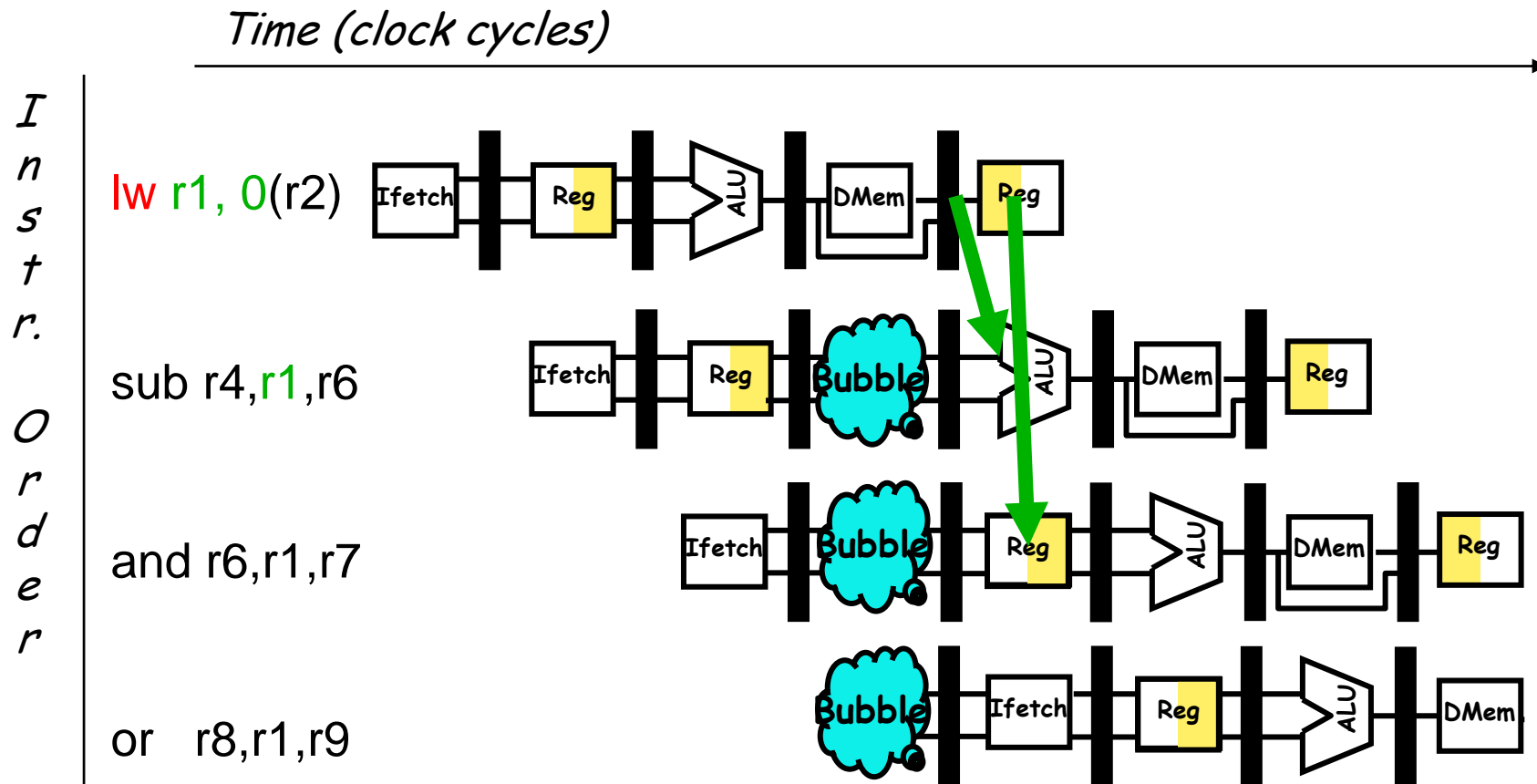
Can't solve with forwarding

Must delay/stall instruction dependent on loads





Pipeline Interlock Solution for Load Stall



How is this detected?

Pipeline interlock





Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

$$d = e - f;$$

assuming a, b, c, d, e, and f in memory.

Slow code:

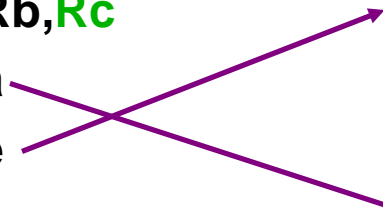
```

LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
    
```

Fast code:

```

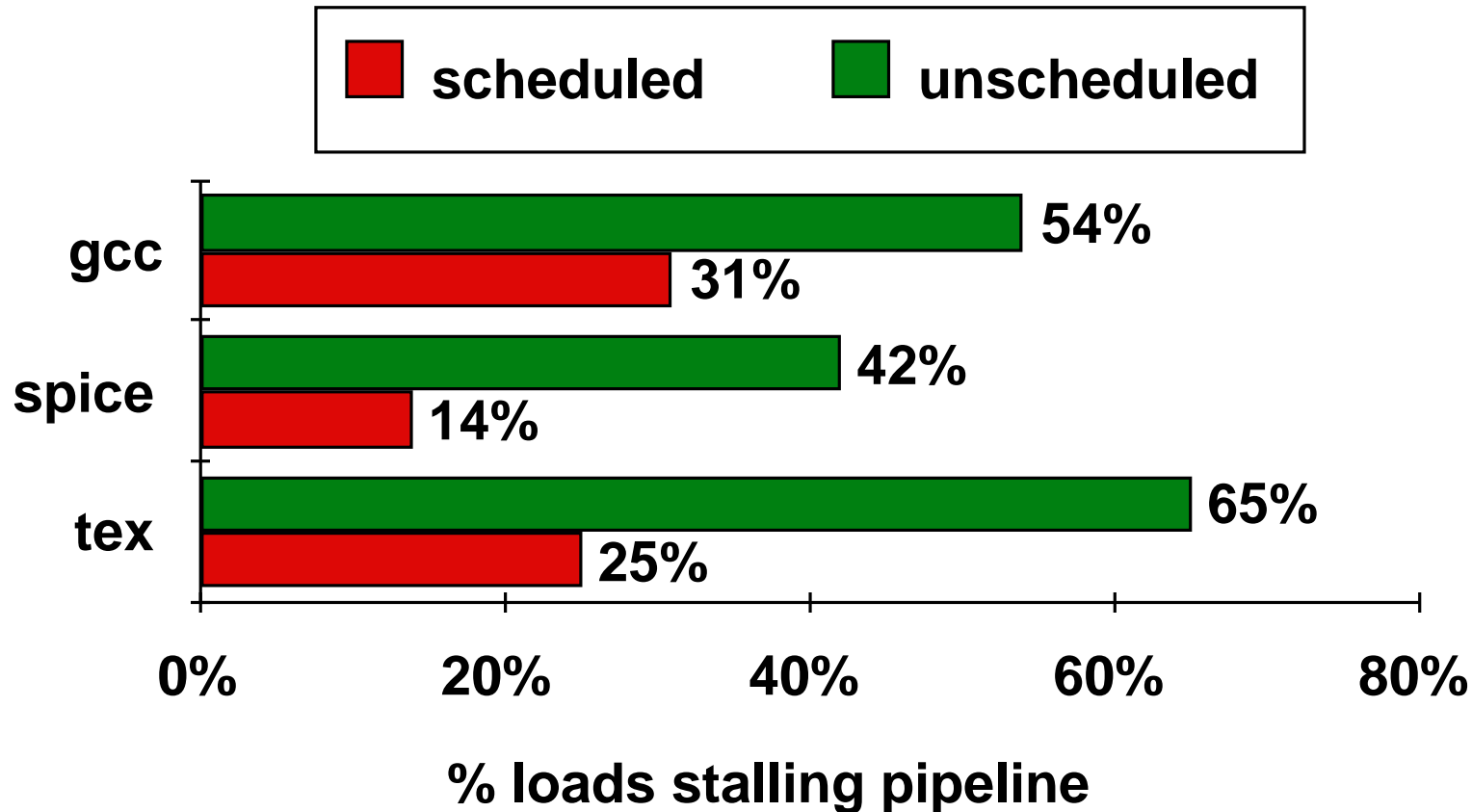
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
    
```



Compiler optimizes for performance. Hardware checks for safety.



Compiler Avoiding Load Stalls



Compilers reduce the number of load stalls, but do not completely eliminate them.

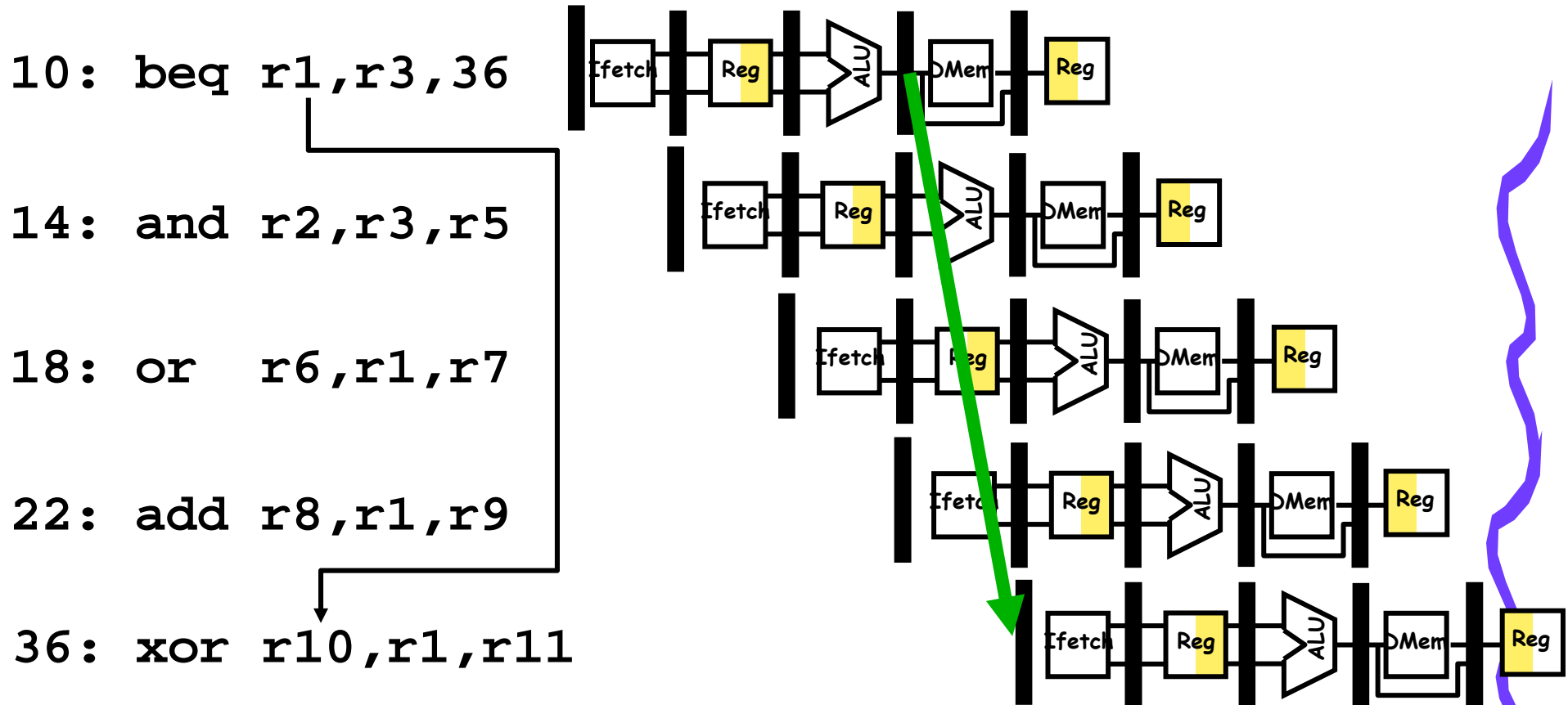
Outline

- MIPS – An ISA example for pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion



Control Hazard on Branches

Three Stages Stall



What do you do with the 3 instructions in between?





Control/Branch Hazards

- Control hazards, which occur due to **instructions changing the PC**, can result in a large performance loss.
- A branch is either
 - **Taken**: $PC \leftarrow PC + 4 + Imm$; branch target address
 - **Not Taken**: $PC \leftarrow PC + 4$
- The simplest solution is **to stall the pipeline as soon as a branch instruction is detected**.
 - Detect the branch in the ID stage
 - Don't know if the branch is taken until the EX stage
 - If the branch is taken, we need to repeat the IF and ID stages
 - **New PC is not changed until the end of the MEM stage**, after determining if the branch is taken and the new PC value



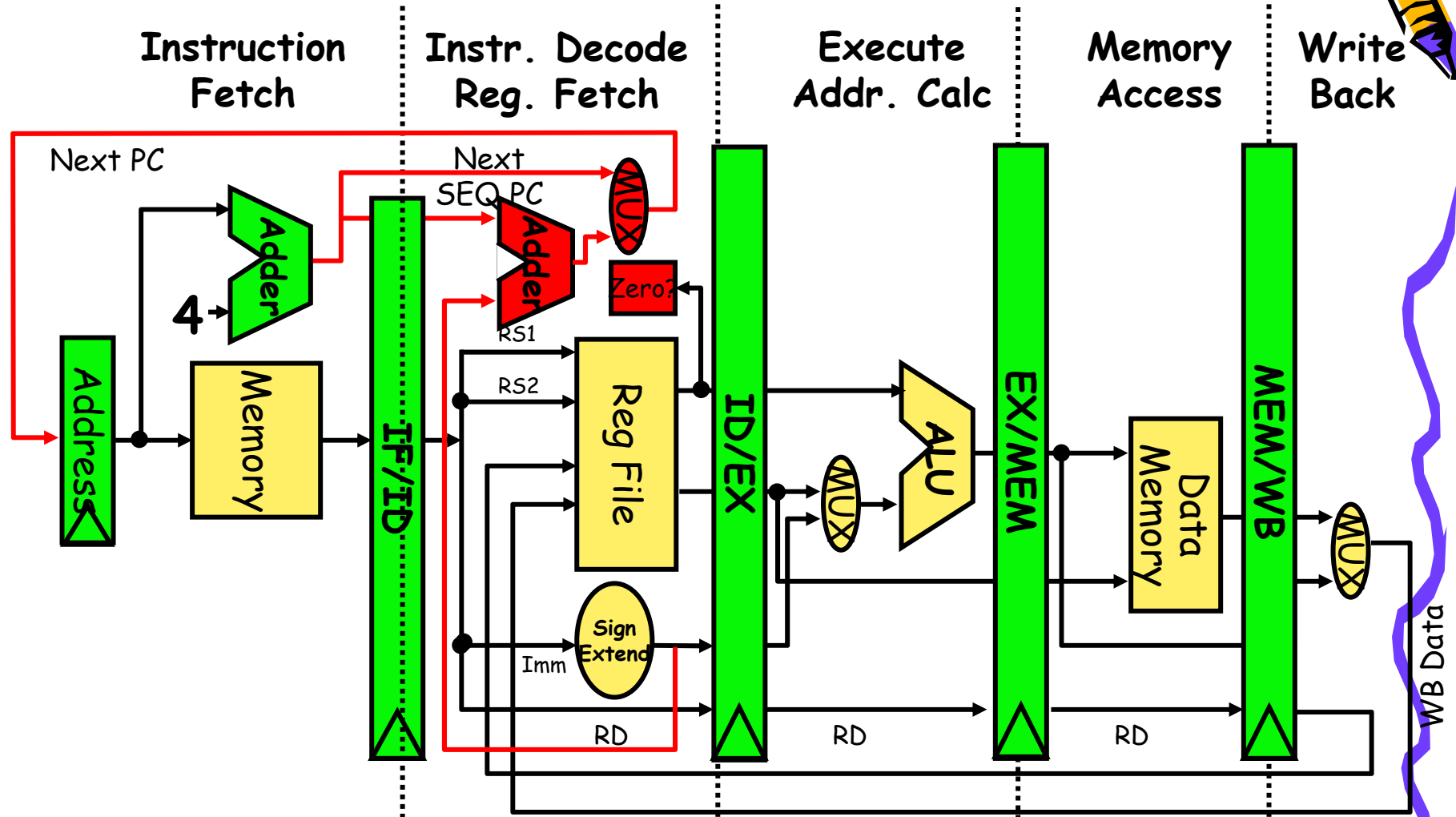
Branch Stall Impact



- If $CPI = 1$, 30% branch,
Stall 3 cycles \Rightarrow new $CPI = 1.9$!!
- Two part solution:
 - Determine branch taken or not sooner, AND
 - Compute taken branch address earlier
- MIPS branch tests if register = 0 or $\neq 0$
- MIPS Solution:
 - Move Zero test to ID/RF stage
 - Adder to calculate new PC in ID/RF stage
 - 1 clock cycle penalty for branch versus 3



Pipelined MIPS Datapath



• Interplay of instruction set design and cycle time.





Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
 - MIPS still incurs 1 cycle branch penalty
 - Other machines: branch target known before outcome





Four Branch Hazard Alternatives

#4: Delayed Branch -- make the stall cycle useful

- Define branch to take place **AFTER** a following instruction

branch instruction

sequential successor₁

sequential successor₂

.....

sequential successor_n

branch target if taken

e.g. Branch delay slot
of length n

These insts. are executed !!

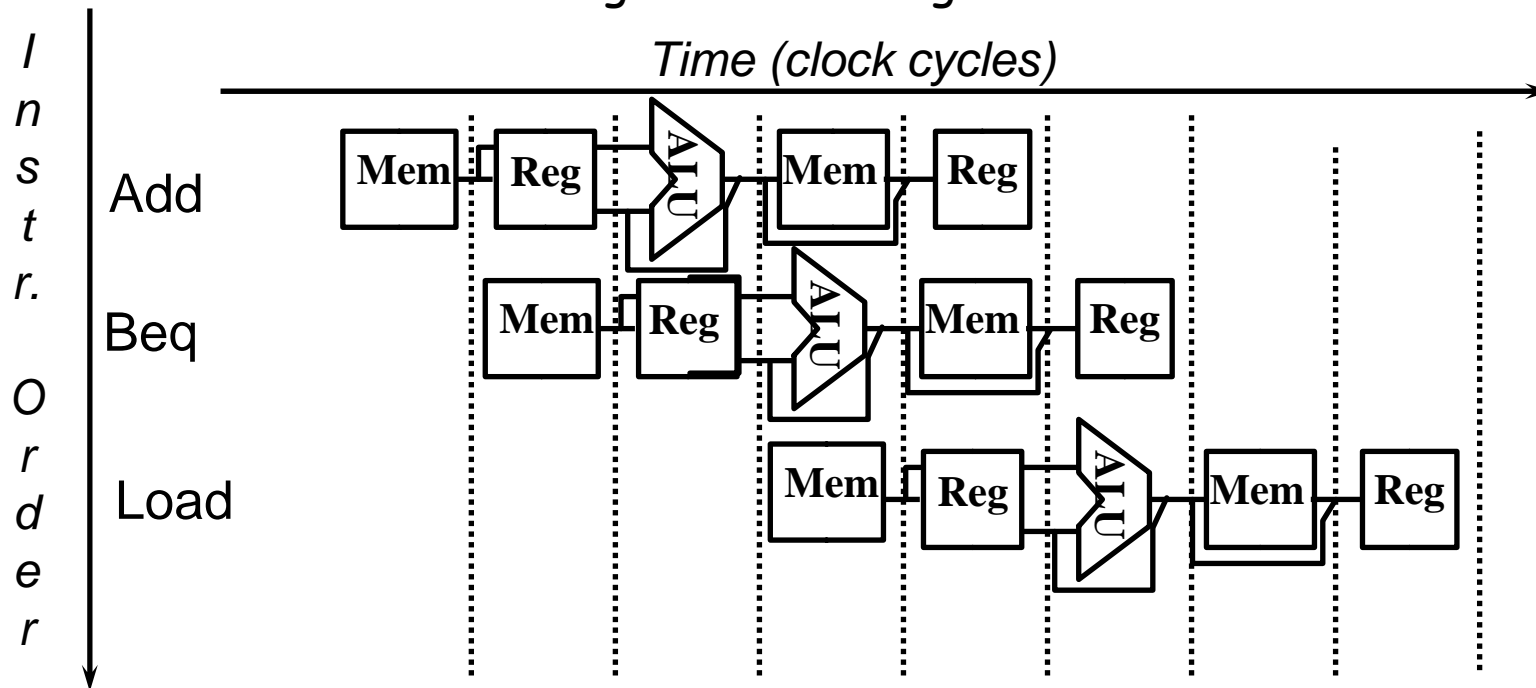
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this





Stall -- Control Hazard Solution

- Stall: **wait until decision is clear**
 - It's possible to move up decision to **2nd stage** by adding hardware to check registers as being read



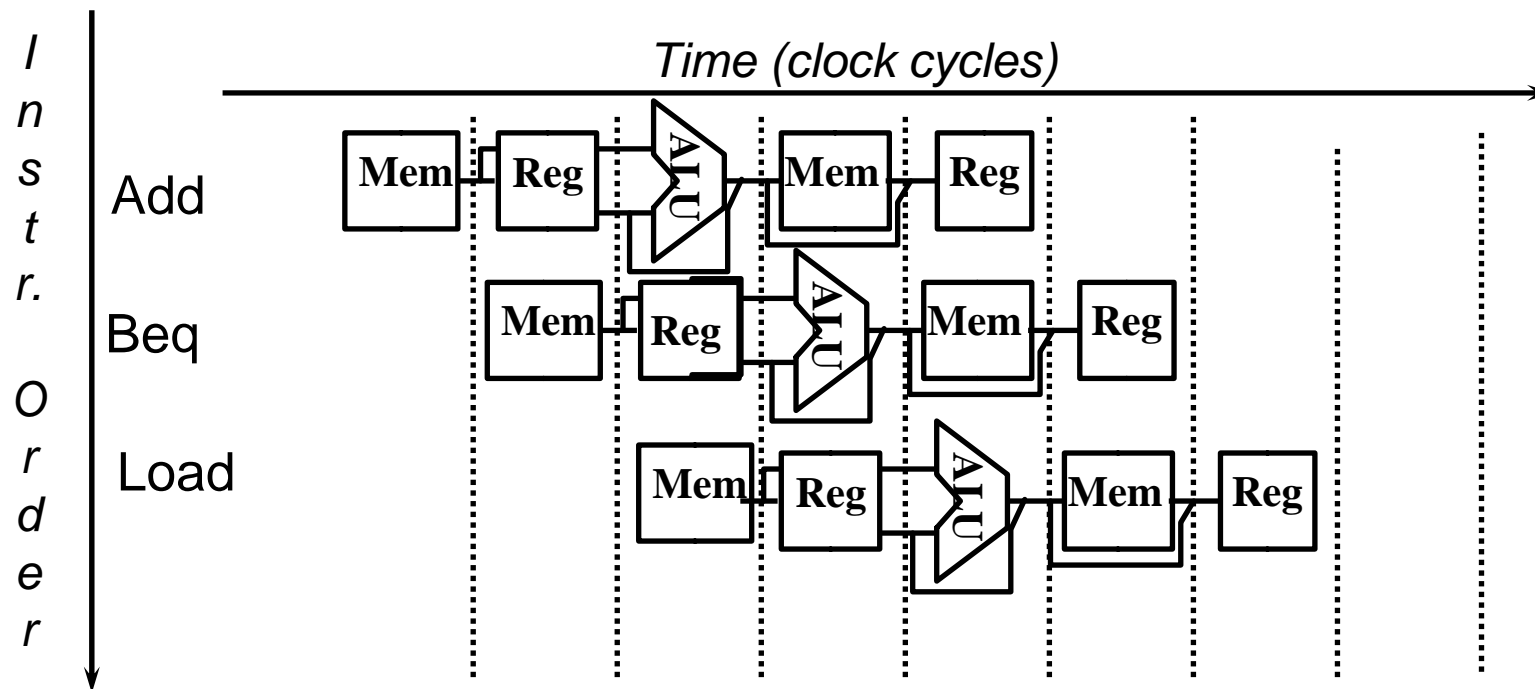
- Impact: 2 cycles (or 1 cycle penalty) per branch instruction
=> slow



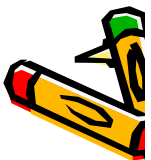


Predict-- Control Hazard Solution

- Predict: **guess one direction then back up if wrong**
 - Predict not taken, for example



- Impact: 1 clock cycle per branch instruction if right, 2 if wrong





Predict-Not-Taken Example

Untaken branch instruction	IF	ID	EX	MEM	WB		
Instruction $i + 1$		IF	ID	EX	MEM	WB	
Instruction $i + 2$			IF	ID	EX	MEM	WB
Instruction $i + 3$				IF	ID	EX	MEM
Instruction $i + 4$					IF	ID	EX

A Stall indeed

Taken branch instruction	IF	ID	EX	MEM	WB		
Instruction $i + 1$		IF	idle	idle	idle	idle	
Branch target			IF	ID	EX	MEM	WB
Branch target + 1				IF	ID	EX	MEM
Branch target + 2					IF	ID	EX

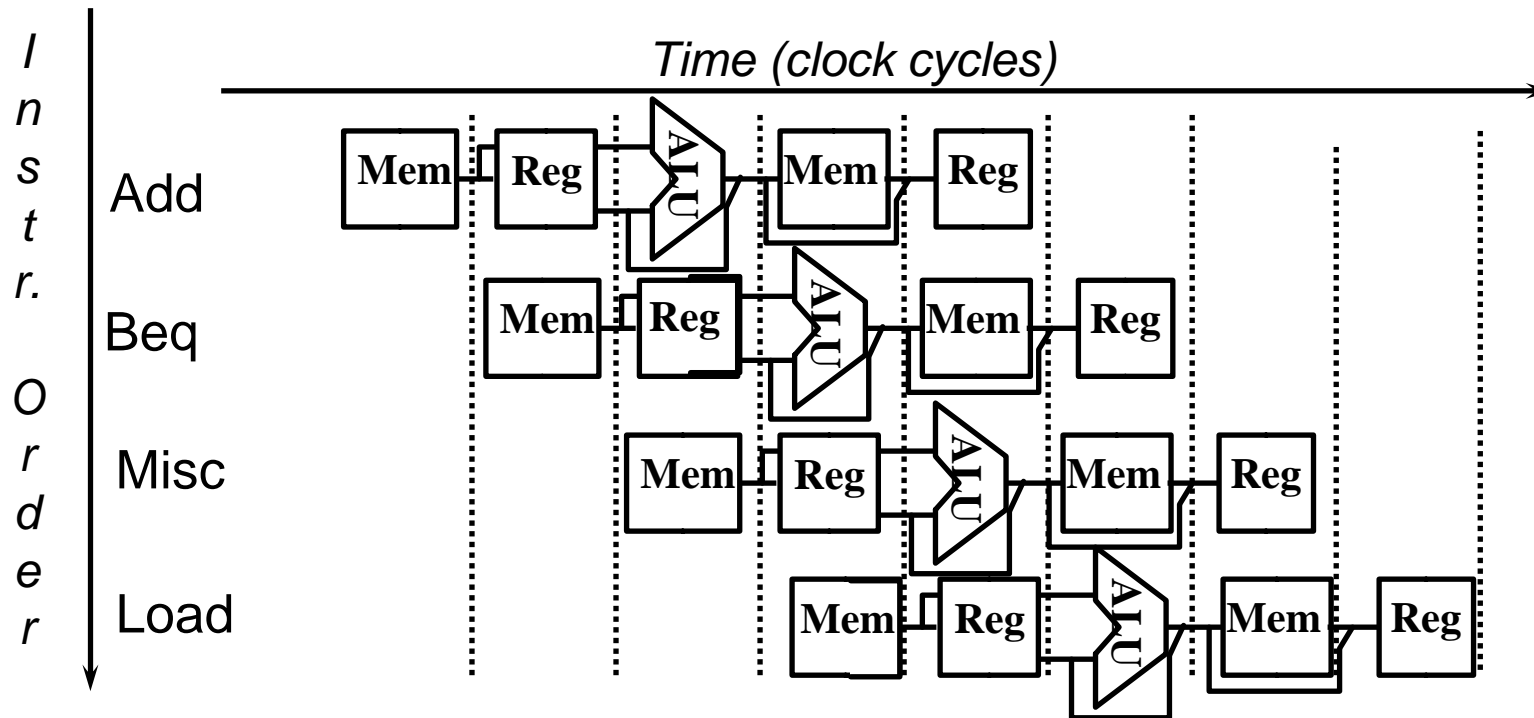


1 clock cycle per branch instruction if right, 2 if wrong

Delayed Branch-- Control Hazard Solution



- Redefine branch behavior (takes place after next instruction)
"delayed branch"



- Impact: 1 clock cycles per branch instruction if can find instruction to put in "slot"



Delayed Branch



- Delayed branch → **make the stall cycle useful**
 - Add delay slots = branch penalty = length of branch delay
 - 1 slot for 5-stage DLX/MIPS
 - Instructions in the delay slot are executed whether or not the branch is taken
 - **See if the compiler can schedule something useful in these slots**
 - When the slots cannot be scheduled, they are filled with the **no-op instruction** (indeed, stall!!)
 - Hope that filled slots actually help advance the computation

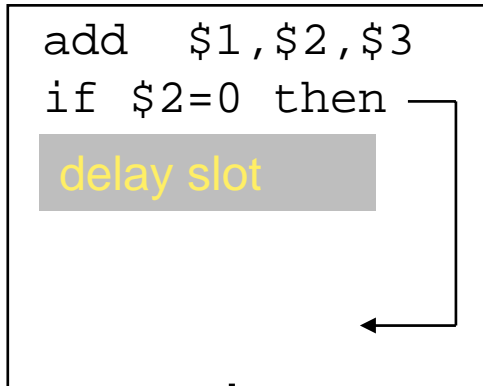




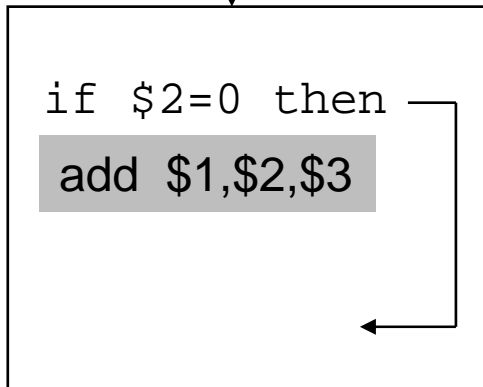
Scheduling Branch Delay Slots



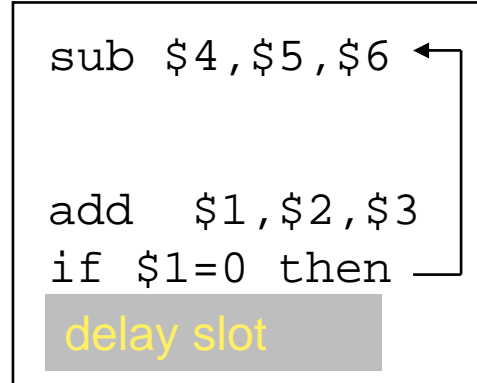
A. From before branch



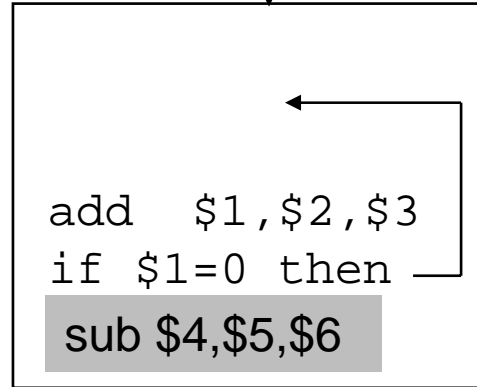
becomes



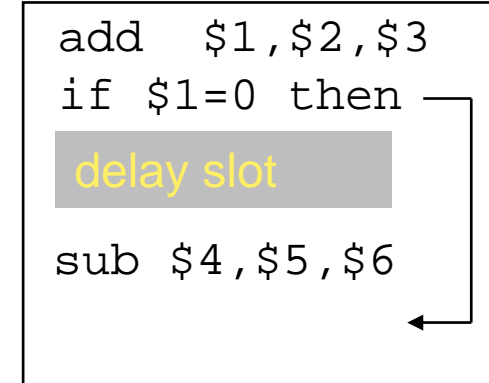
B. From branch target



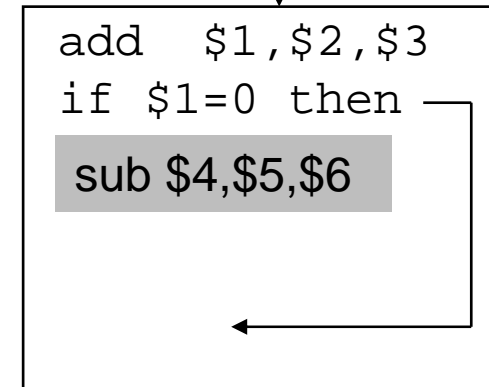
becomes



C. From fall through



becomes



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails





Delay-Branch Scheduling Schemes and Their Requirements



Scheduling Strategy	Requirements	Improve Performance When?
From before	Branch must not depend on the rescheduled instructions	Always
From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions	When branch is taken. May enlarge program if instructions are duplicated
From fall through	Must be OK to execute instructions if branch is taken	When branch is not taken.



Delayed Branch Summary



- Compiler effectiveness for single branch delay slot:
 - Fills about 60% of branch delay slots
 - About 80% of instructions executed in branch delay slots useful in computation
 - About 50% (60% x 80%) of slots usefully filled
- As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slots
 - Delayed branching (the static way) has lost popularity compared to more expensive but more flexible dynamic approaches
 - Growth in available transistors has made dynamic approaches relatively cheaper





Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

- Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<i>Branch scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45





Deeper Pipeline Example

- For a deeper pipeline, e.g. MIPS R4K, it takes at least 3 pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparisons
 - Assuming an ideal CPI of 1,

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$





Branch Penalties

- Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken
- Branch penalties

<i>Branch scheme</i>	<i>Penalty unconditional.</i>	<i>Penalty untaken</i>	<i>Penalty taken</i>
Flush pipeline	2	3	3
Predict taken	2	3	2
Predict not taken	2	0	3

Try to find the CPI penalties for 3 branch schemes (Fig. A.16)



Problems with Pipelining



- **Exception:** An unusual event happens to an instruction during its execution
 - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
 - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **Problem:** It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
 - The effect of all instructions up to and including I_i is totalling complete
 - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}





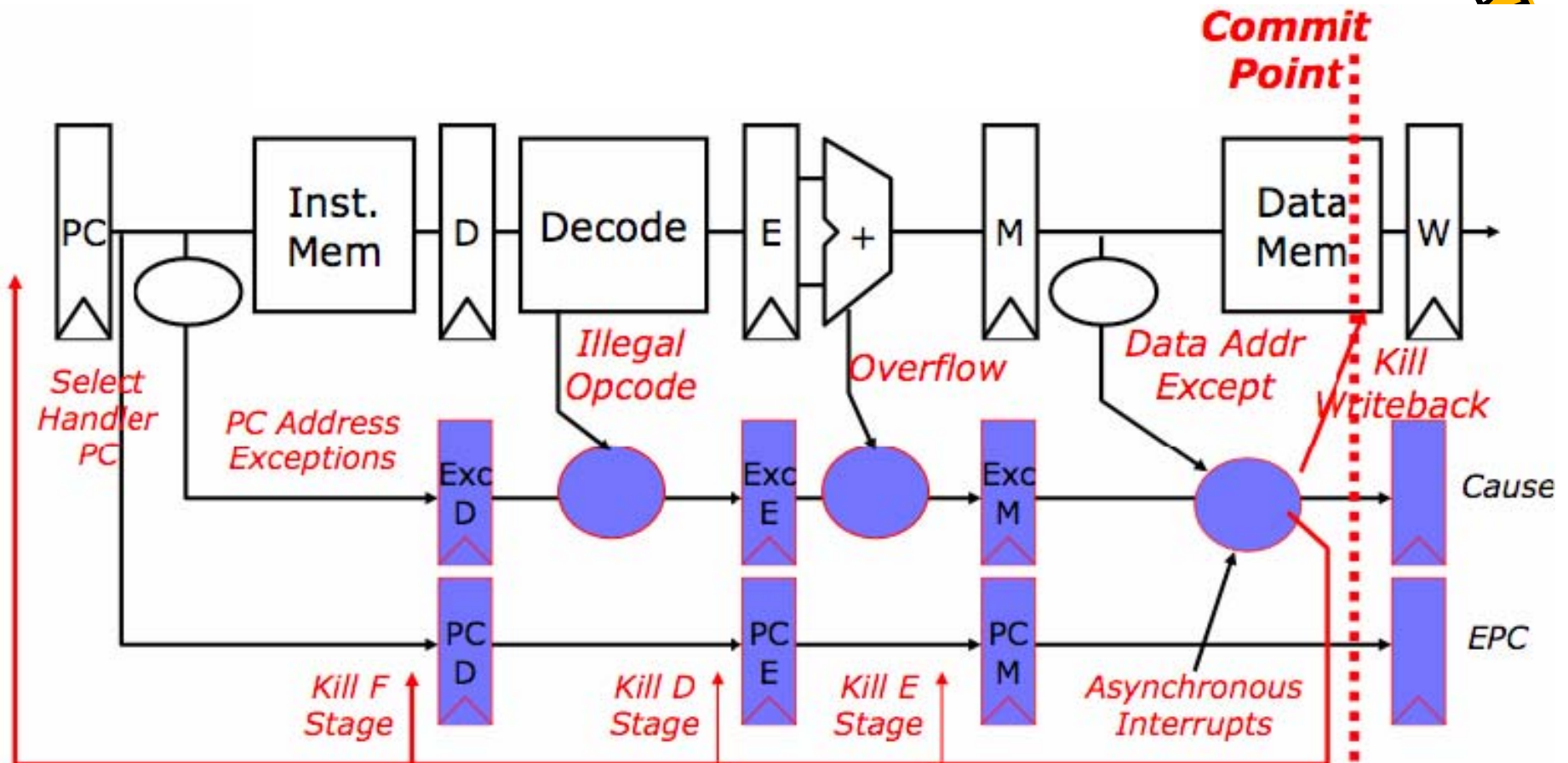
Exceptions in MIPS

Pipeline stage	Problem exceptions 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
WB	Non

Note: Multiple exceptions may occur in the same clock cycle in pipelining architecture



Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.



And In Conclusion: Control & Pipelining



- Control VIA **State Machines** and **Microprogramming**
- Just overlap tasks; easy if tasks are independent
- Speed Up \leq Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW, WAR, WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity

