Computer Architecture
-----A Quantitative Approach

陈文智

浙江大学计算机学院

chenwz@zju.edu.cn

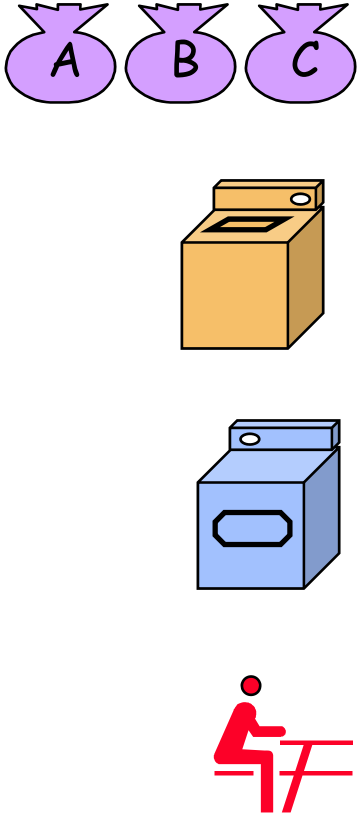
What is Pipelining ?

- Pipelining:
  - *"A technique designed into some computers to increase speed by starting the execution of one instruction before completing the previous one."*

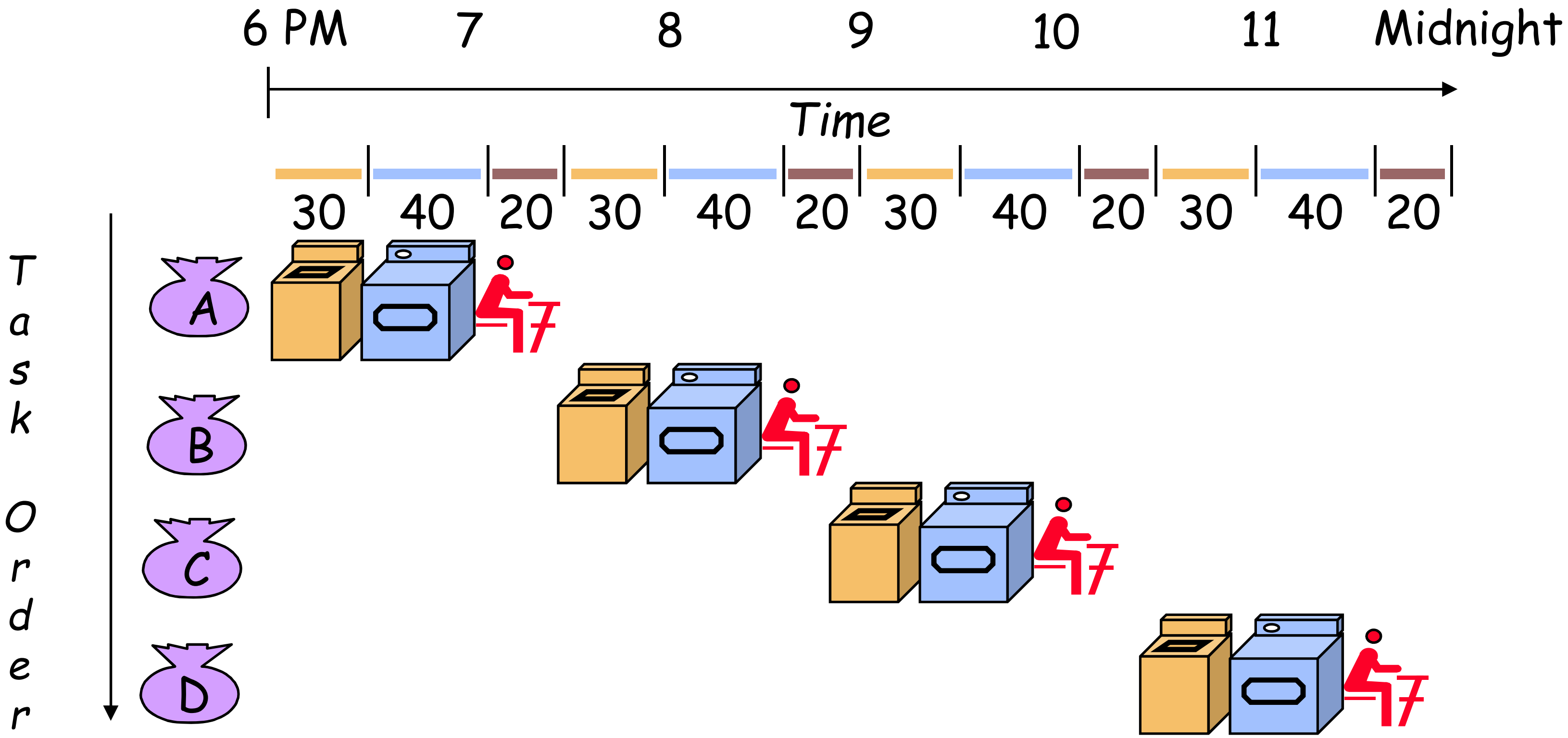
    ----Modern English-Chinese Dictionary
  - implementation technique whereby different instructions are overlapped in execution at the same time.
  - implementation technique to make fast CPUs

Why Pipelining: Its Natural

- Laundry
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

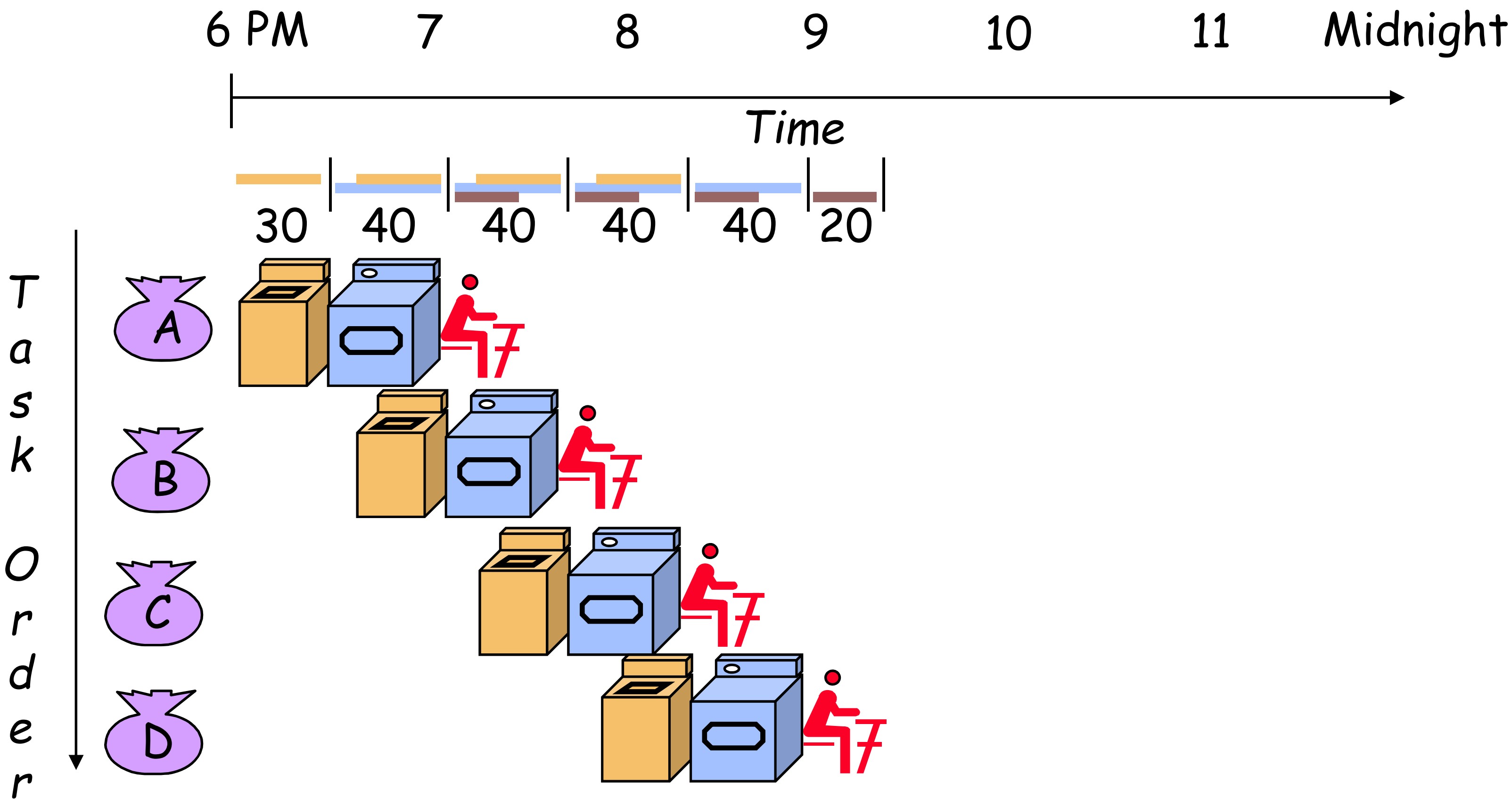


Sequential Laundry



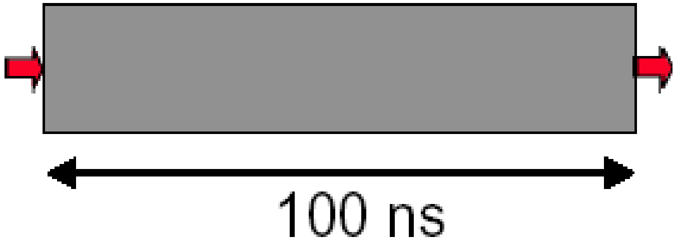
- Sequential laundry takes 6 hours for 4 loads
- **If they learned pipelining, how long would laundry take?**

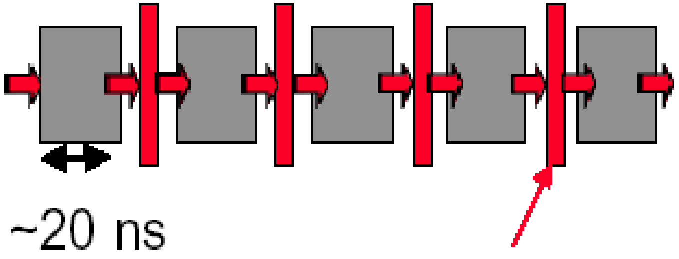
Pipelined Laundry----Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Why pipelining : overlapped

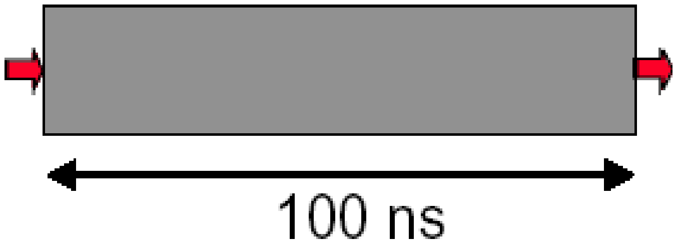




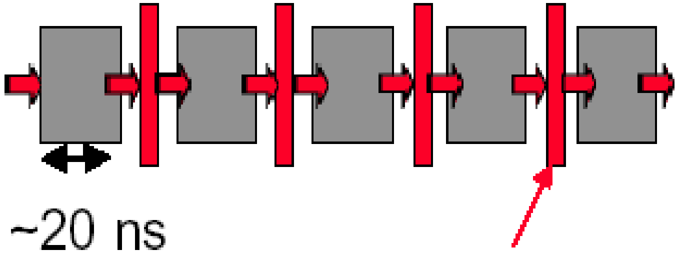
- Only deal one task each time.
- This task takes “ such a long time”

- Latches, called pipeline registers’ break up computation into 5 stages
- Deal 5 tasks at the same time.

Why pipelining: more faster



- Can "launch" a new computation every 100ns in this structure
- Can finish 10^7 computations per second



- Can launch a new computation every 20ns in pipelined structure
- Can finish 5×10^7 computations per second

Why pipelining : conclusion

- The key implementation technique used to Make fast CPU: decrease CPUtime.
- Improving of Throughput (rather than individual execution time)
- Improving of efficiency for resources (functional unit)

What is a pipeline ?

- A pipeline is like an auto assemble line
- A pipeline has many stages
- Each stage carries out a different part of instruction or operation
- The stages, which cooperates at a synchronized clock, are connected to form a pipe
- An instruction or operation enters through one end and progresses through the stages and exit through the other end
- Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream

Ideal Performance for Pipelining

- If the stages are perfectly balanced, The time per instruction on the pipelined processor equal to:

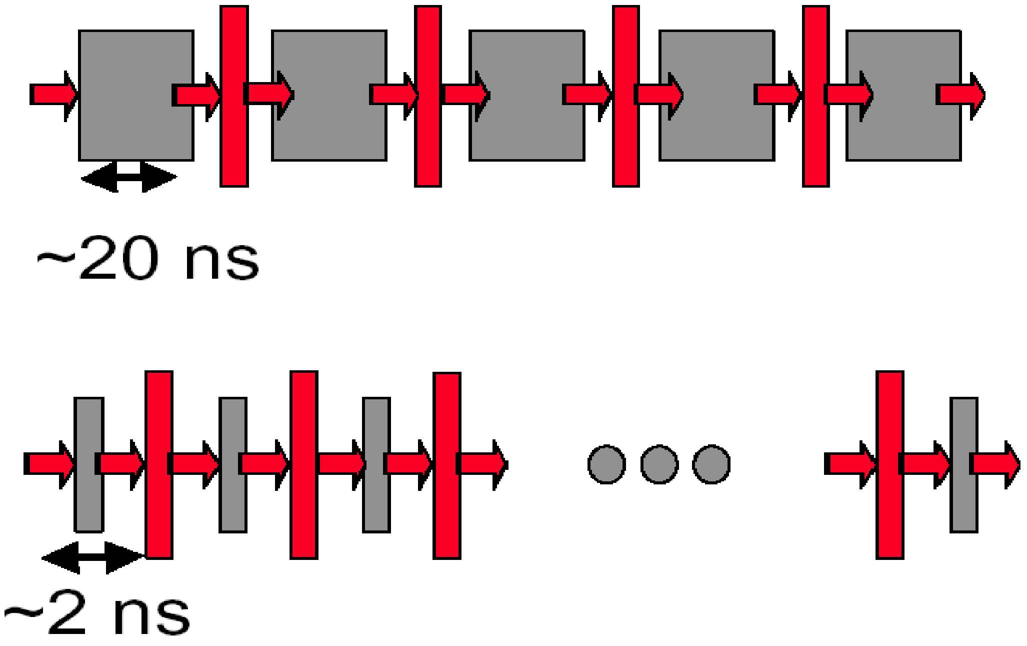
$$\frac{\textit{Time per instruction on unpipelined machine}}{\textit{Number of pipe stages}}$$

- So, **Ideal speedup equal to**

  Number of pipe stages.

Why not just make a 50-stage pipeline ?

- Some computations just won't divide into any finer (shorter in time) logical implementation.

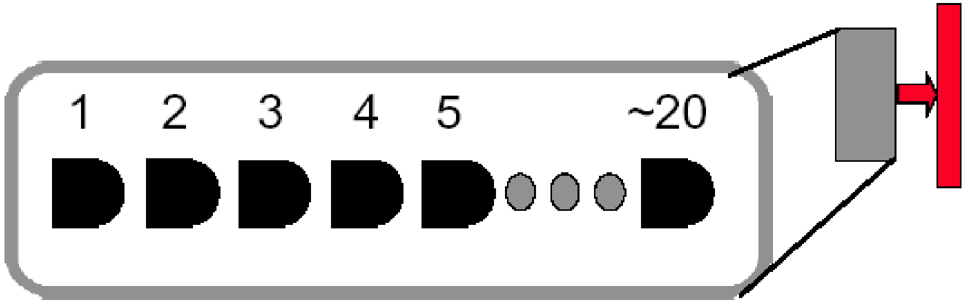


5 stages OK

50 stages NO. Sorry!

Why not just make a 50-stage pipeline ?

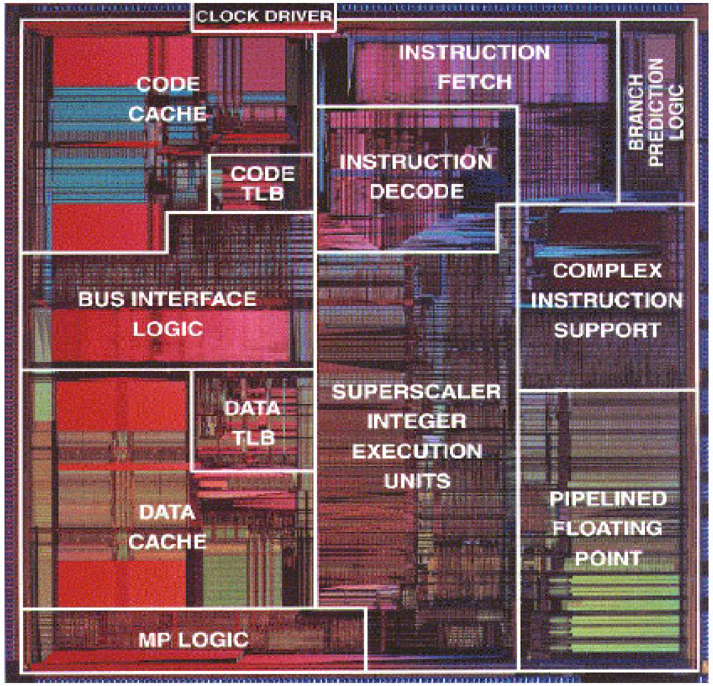
- Those latches are **NOT free**, they take up **area**, and there is a real **delay** to go THRU the latch itself.
  - Machine cycle > latch latency + clock skew
- In modern, deep pipeline (10-20 stages), this is a real effect
- Typically see logic “depths” in one pipe stage of 10-20 “gates”.



At these speeds, and with this few levels of logic, latch delay is important

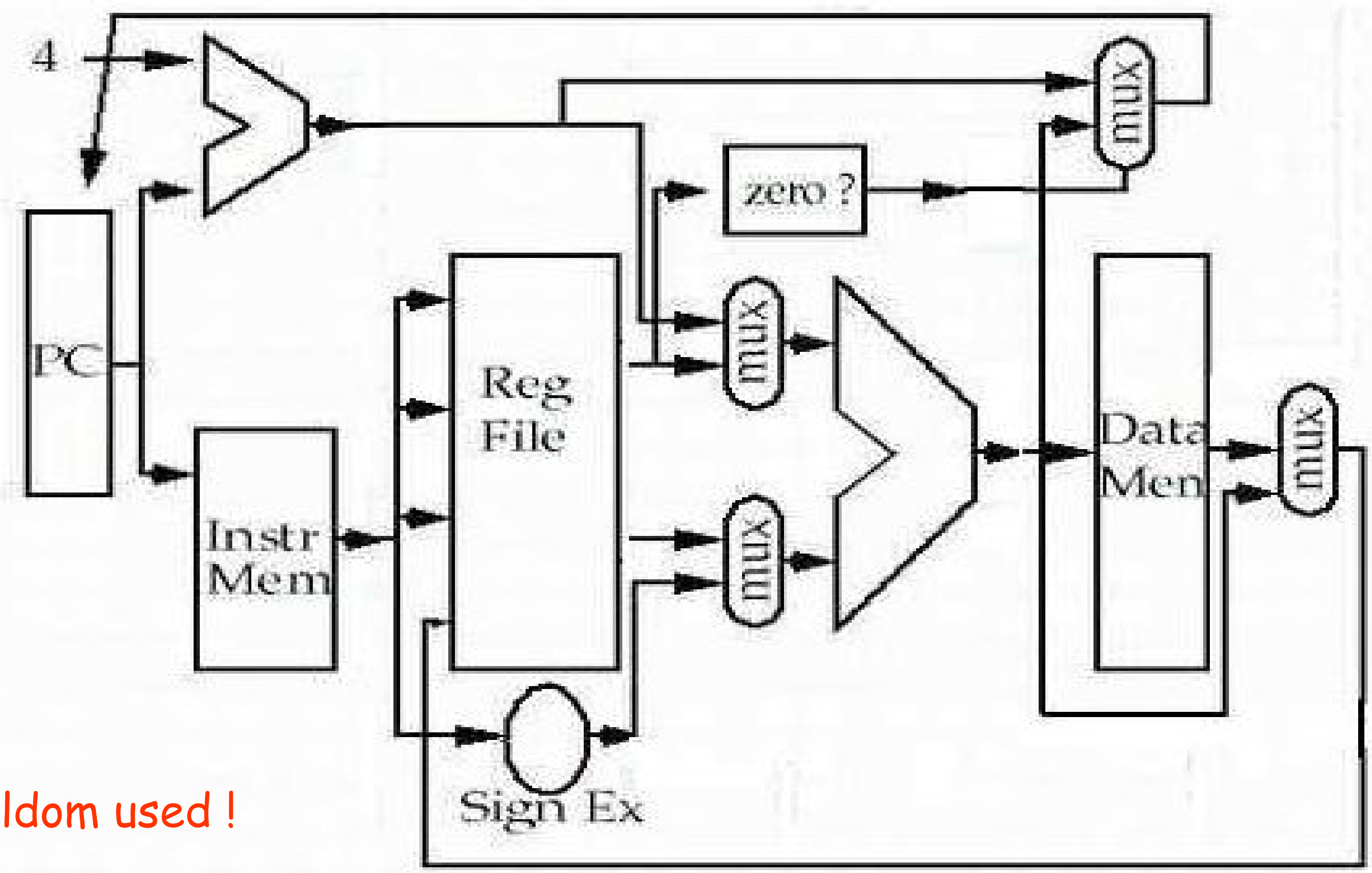
How Many Pipeline Stages?

- E.g., Intel
  - Pentium III, Pentium 4: 20+ stages
  - More than 20 instructions in flight
  - High clock frequency (>1GHz)
  - High IPC



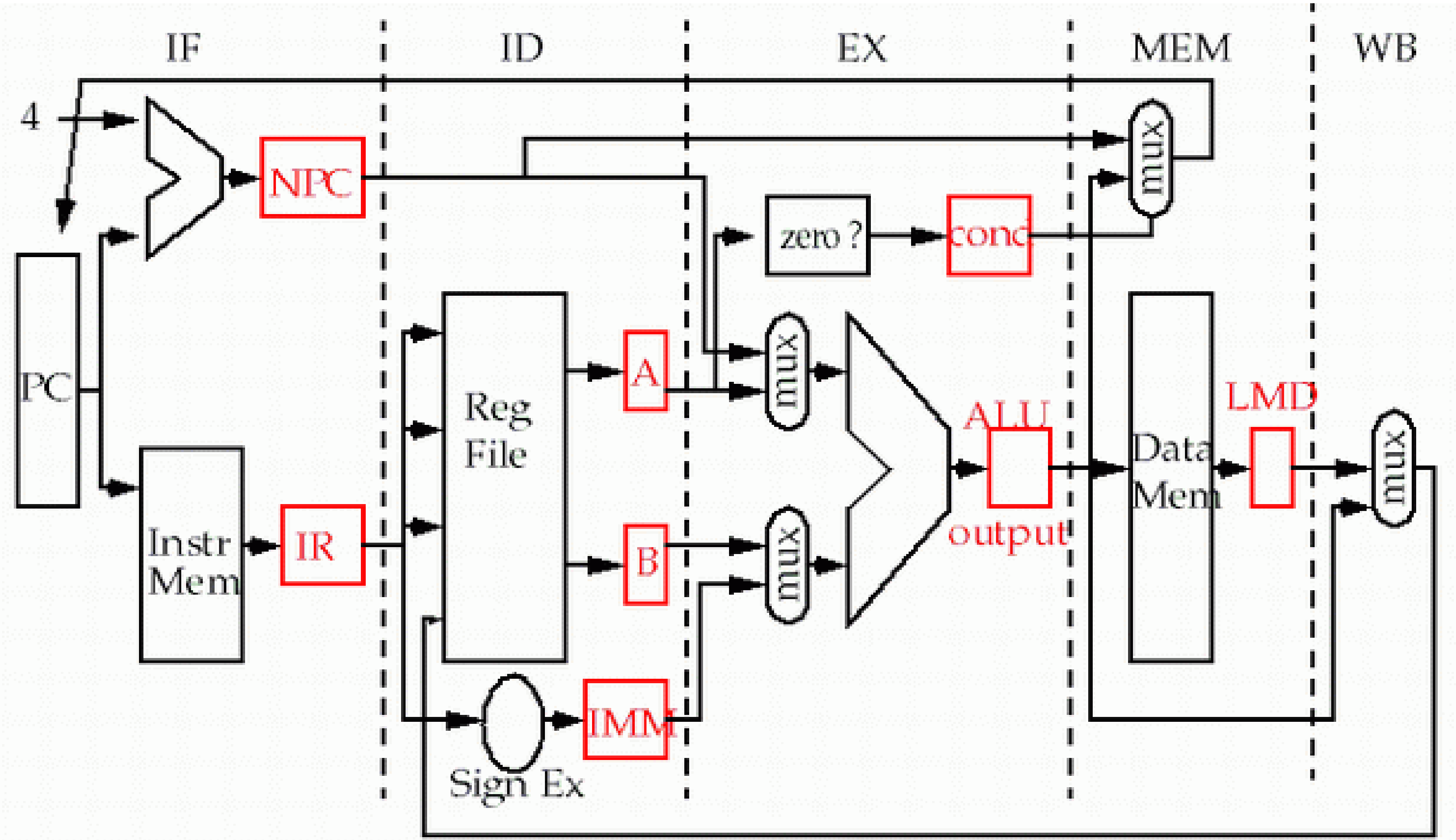
- Too many stages:
  - Lots of complications
  - Should take care of possible dependencies among in-flight instructions
  - Control logic is huge

Single-cycle implementation

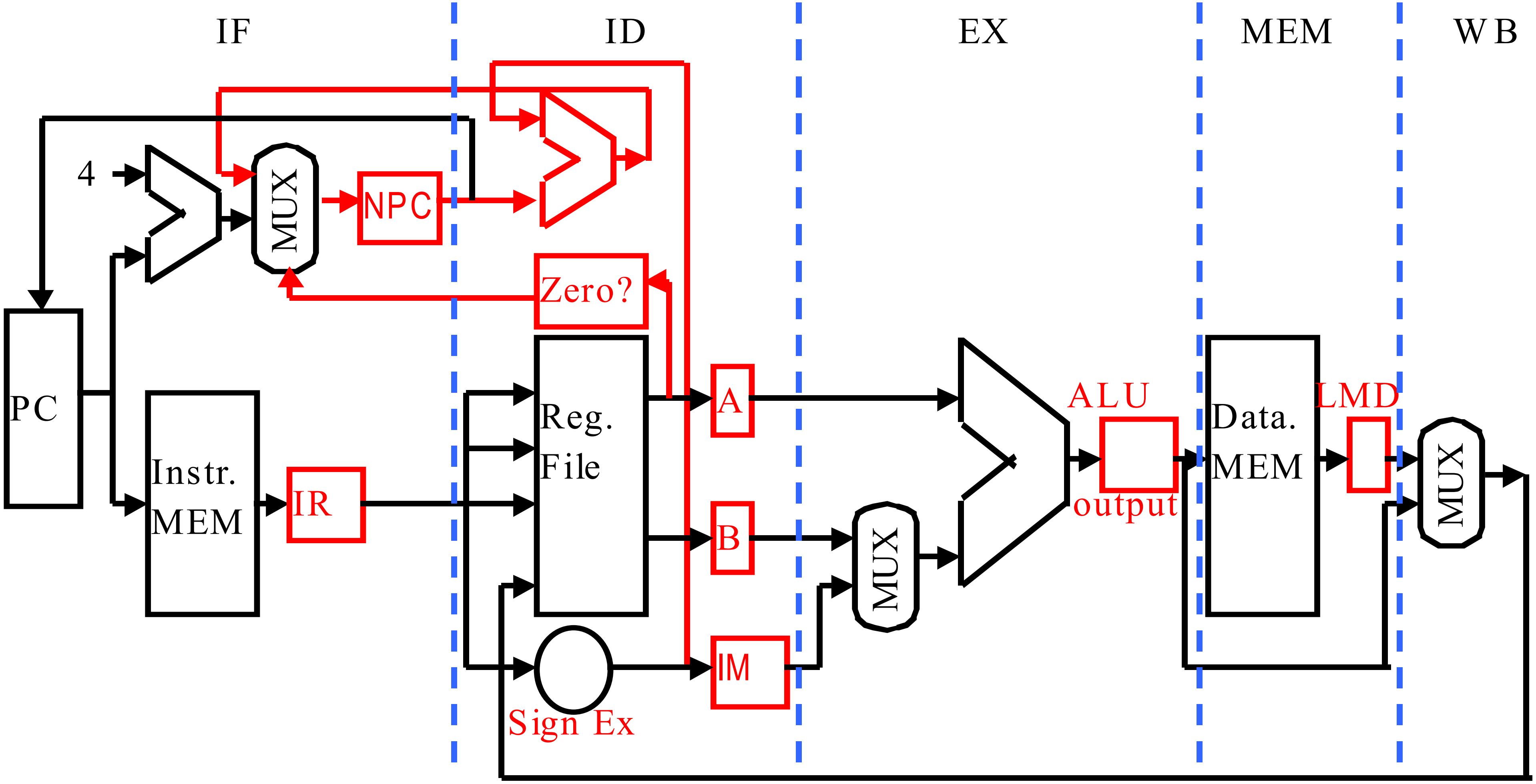


seldom used !

Multi-cycle implementation

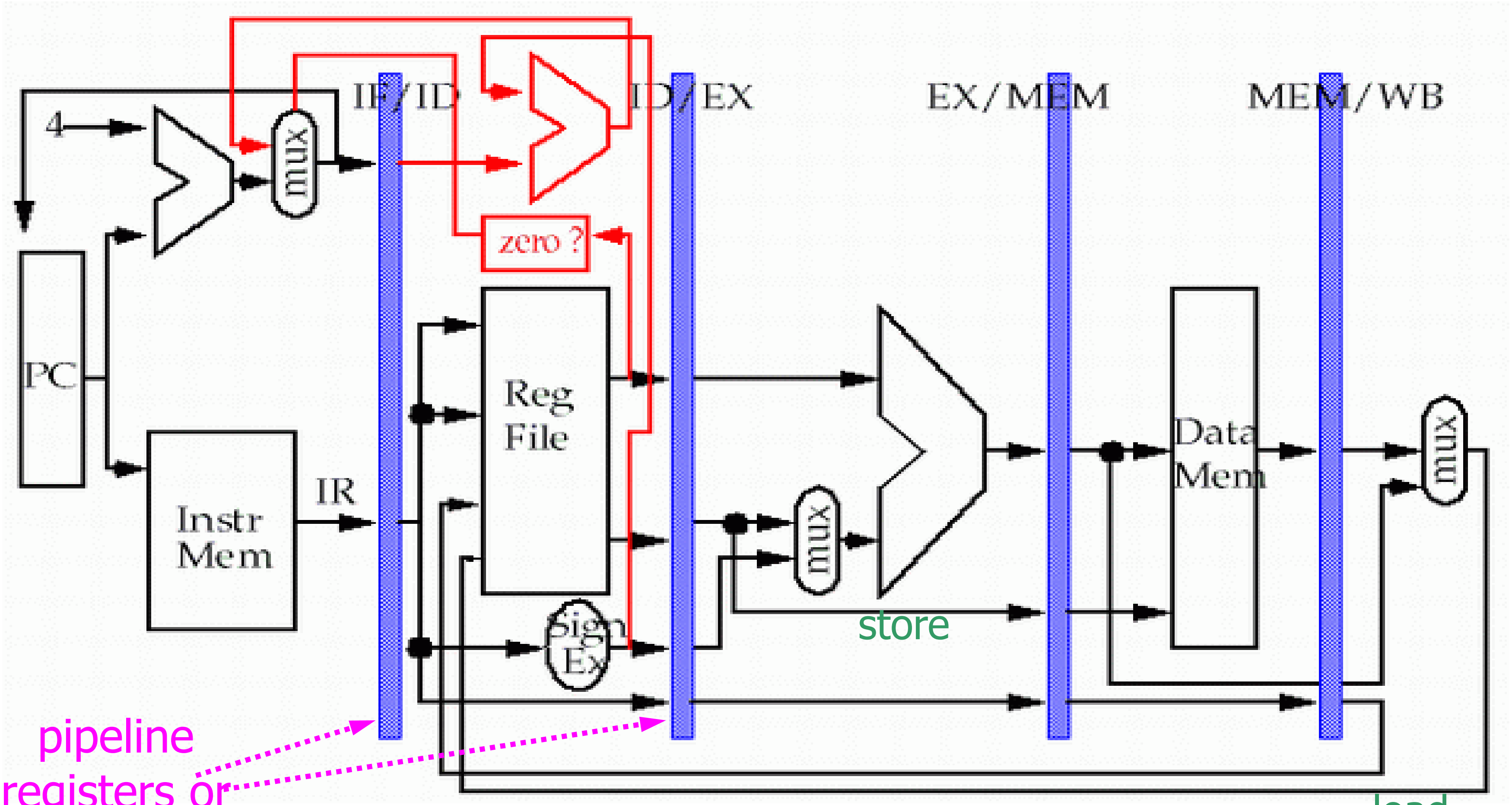


Optimized Multi-cycle implementation



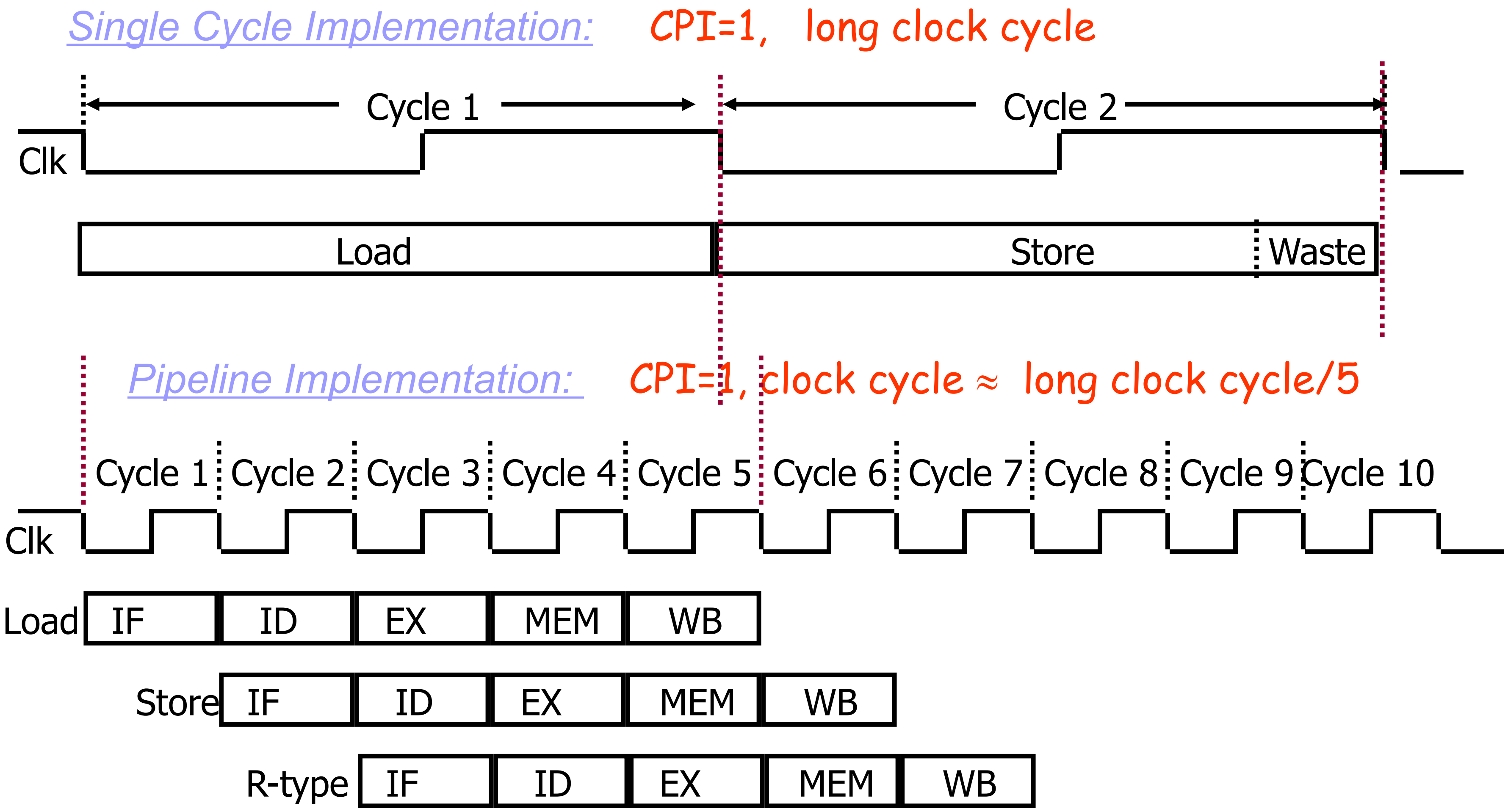
🟥 Temporary storage locations

5-stage Version of MIPS Datapath

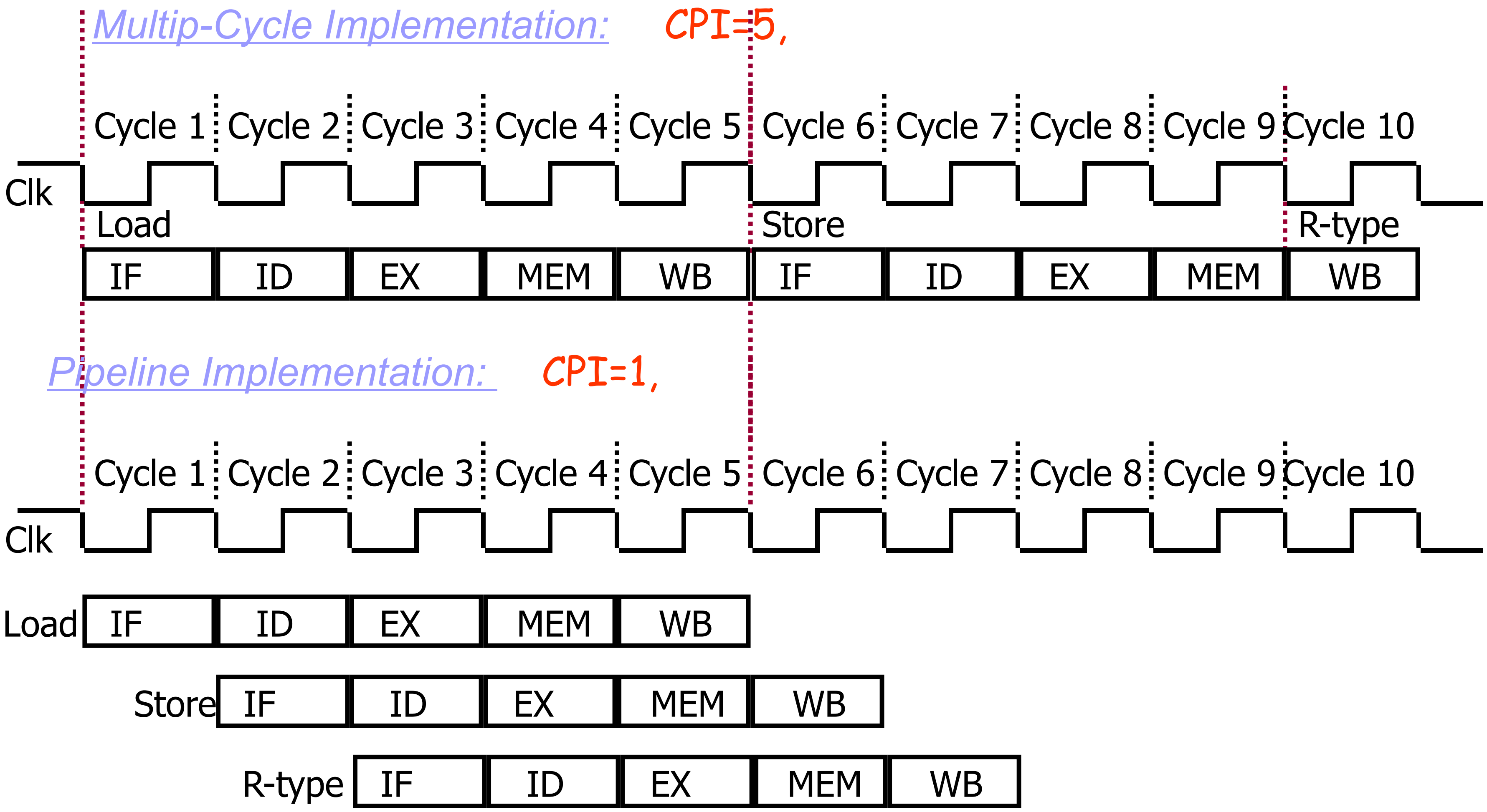


pipeline registers or latches

Single-cycle implementation vs. pipelining



Multi-cycle implementation vs. pipelining



Multip-Cycle Implementation: CPI=5,

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10

Clk

Load: IF ID EX MEM WB

Store: IF ID EX MEM

R-type: WB

Pipeline Implementation: CPI=1,

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10

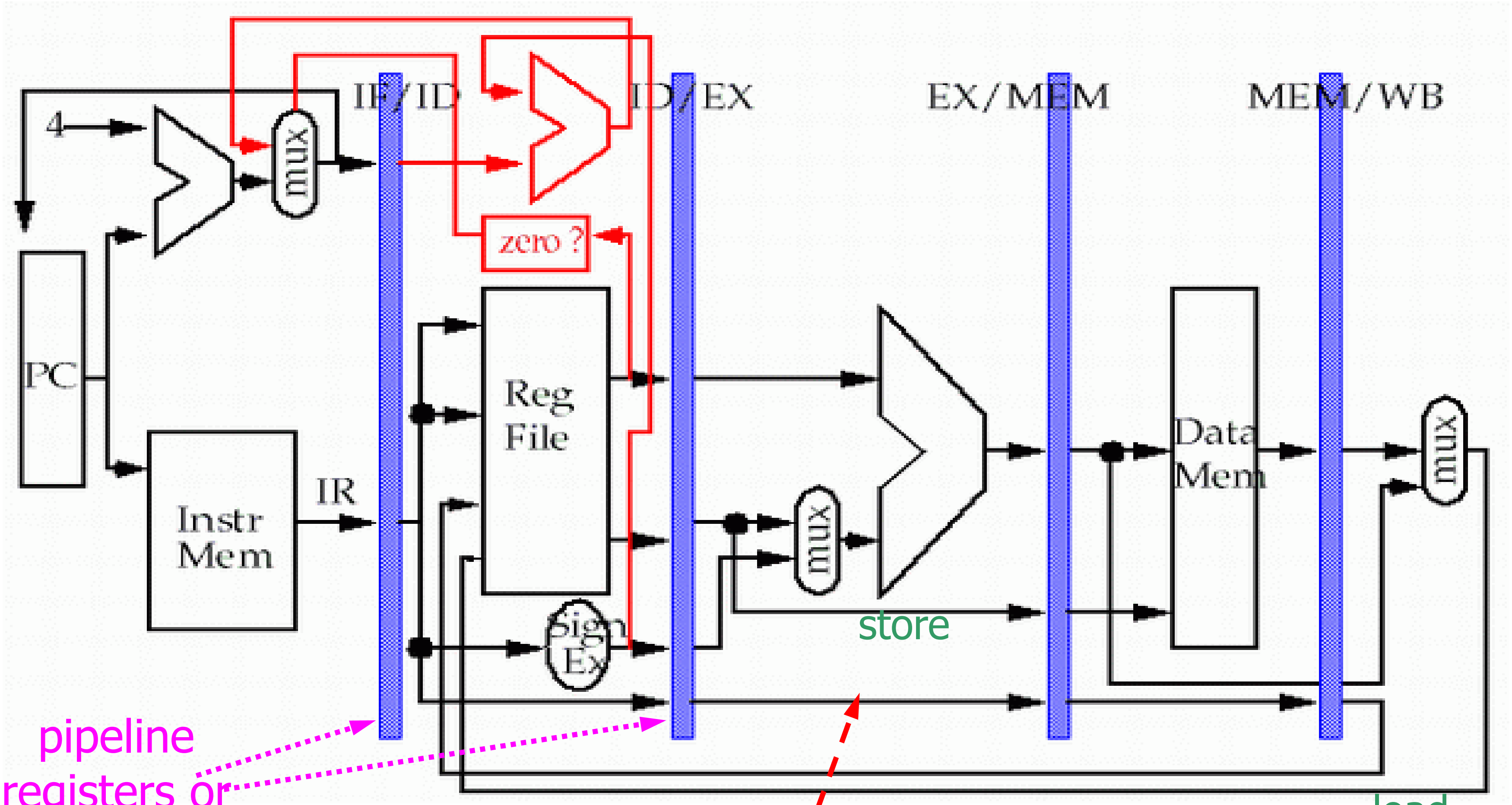
Clk

Load IF ID EX MEM WB

Store IF ID EX MEM WB

R-type IF ID EX MEM WB

How simple as this ! Really ?



pipeline registers or latches

Why need to add this line?

Pipeline hazard: the major hurdle

- A **hazard** is a condition that prevents an instruction in the pipe from executing its next scheduled pipe stage
- Taxonomy of hazard
  - **Structural hazards**
    - These are conflicts over hardware resources.
  - **Data hazards**
    - Instruction depends on result of prior computation which is not ready (computed or stored) yet
  - **Control hazards**
    - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

Hazards can always be resolved by Stall

- The simplest way to "fix" hazards is to **stall** the pipeline.
- Stall means suspending the pipeline for some instructions by one or more clock cycles.
- The **stall** delays **all instructions issued after** the instruction that was stalled, while other instructions in the pipeline go on proceeding.
- A pipeline stall is also called a **pipeline bubble** or simply **bubble**.
- No new instructions are fetched during a stall .

Performance of pipeline with stalls

- Pipeline stalls decrease performance from the ideal
- Recall the speedup formula:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instuction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \boxed{\frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}}$$

Case of multi-cycle implementation

- The ideal CPI on a pipelined processor is almost always 1. (may less than or greater that)
- So

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clk cycles per instuction}\\ &= 1 + \text{Pipeline stall clk cycles per instruction}\end{aligned}$$

- Ignore the overhead of pipelining clock cycle.
- Pipe stages are ideal balanced.

Case of multi-cycle implementation

- So: Clock cycle unpipelined = Clock cycle pipelining

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

CPI unpipelined = pipeline depth

$$\boxed{\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}}$$

Case of single-cycle implementation

- CPI unpipelined = 1

- $\text{Clock cycle pipelined} = \dfrac{\text{Clock cycle unpipelined}}{\text{pipeline depth}}$

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline Stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

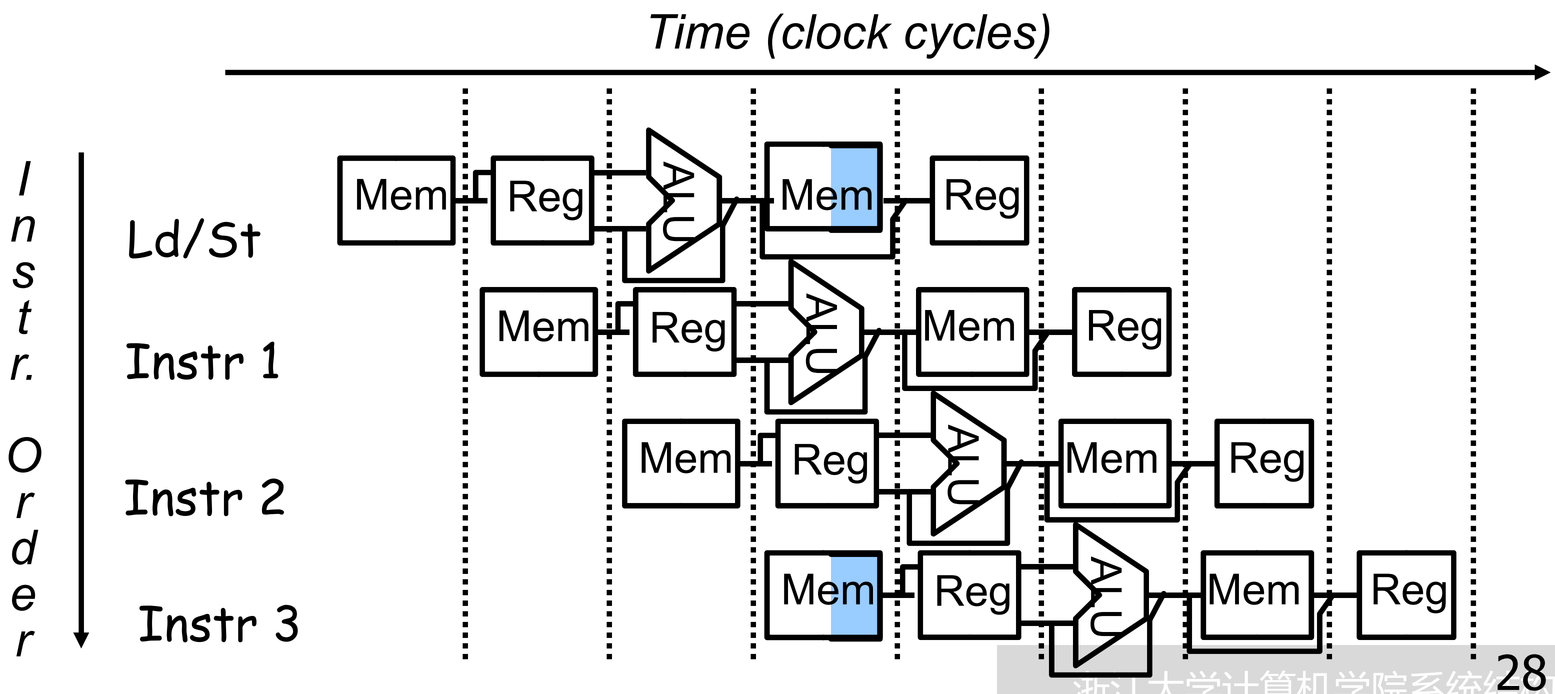
$$\boxed{\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}}$$

Structural hazard: Pipe Stage Contention

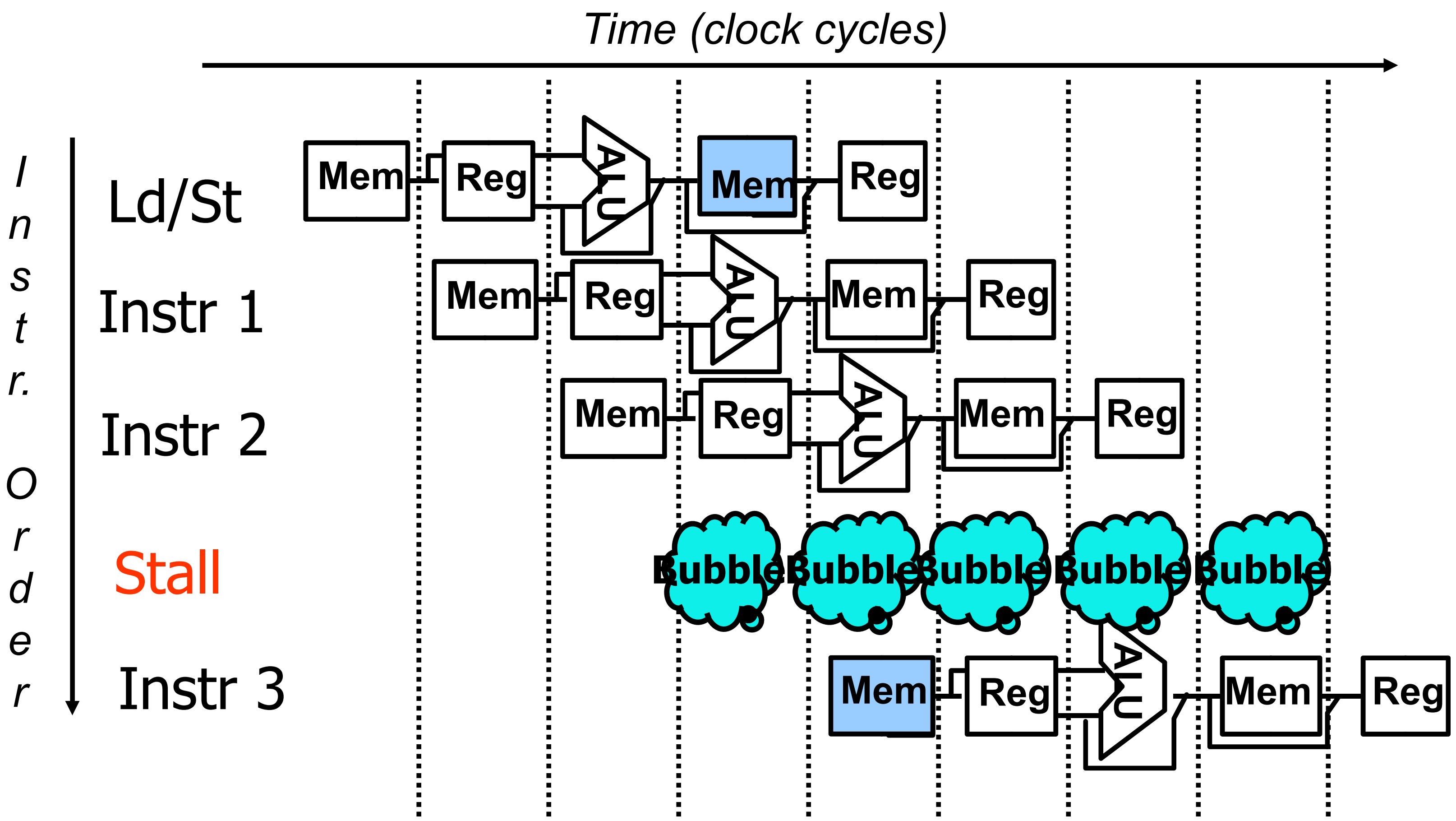
- **Structural hazards**
  - Occurs when two or more instructions want to use the same hardware resource in the same cycle
  - Causes bubble (stall) in pipelined machines
  - Overcome by replicating hardware resources
    - Multiple accesses to the register file
    - Multiple accesses to memory
    - some functional unit is not fully pipelined.
    - Not pipelined functional units

Problems that pipelining introduces

- Focus: different operations with the same data path resource on the same clock cycle is not possible. (structure hazard)
- There is conflict about the memory !

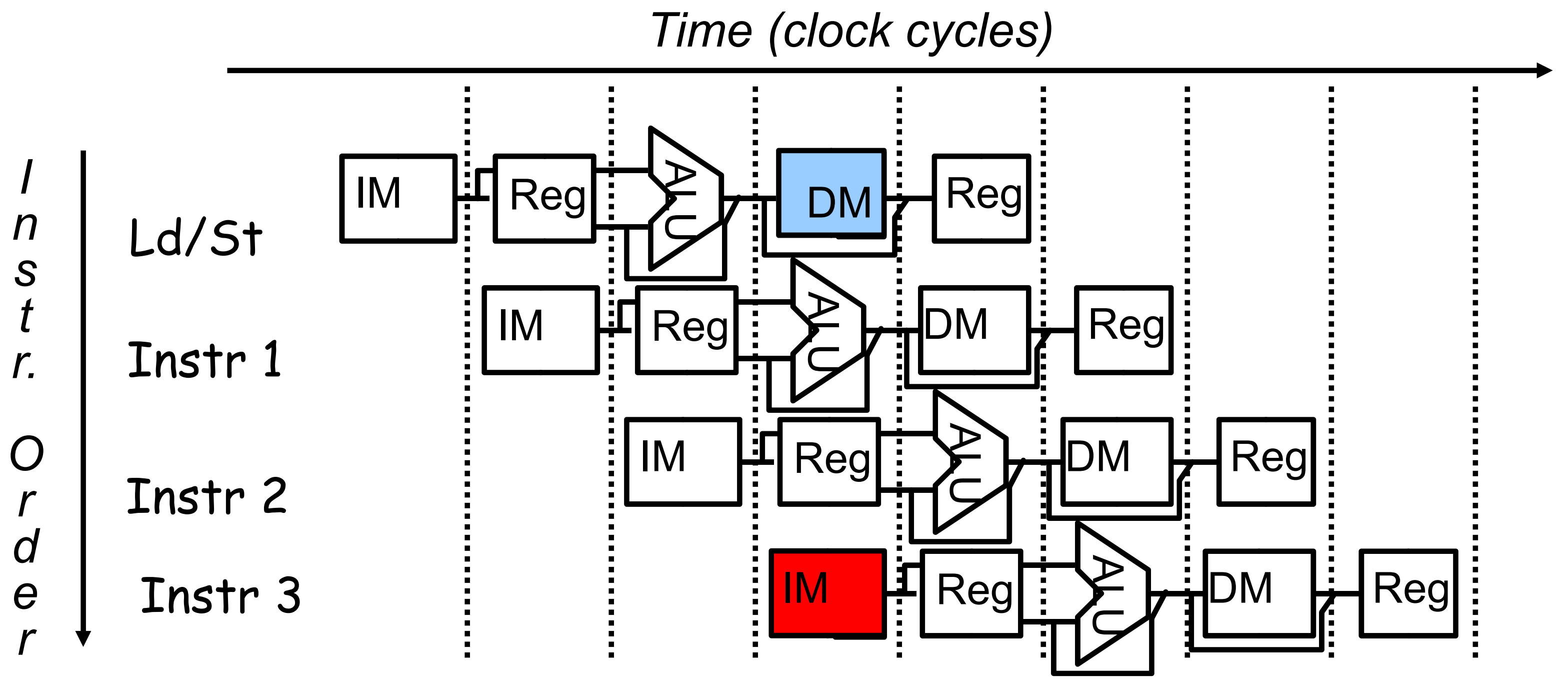


Insert Stall



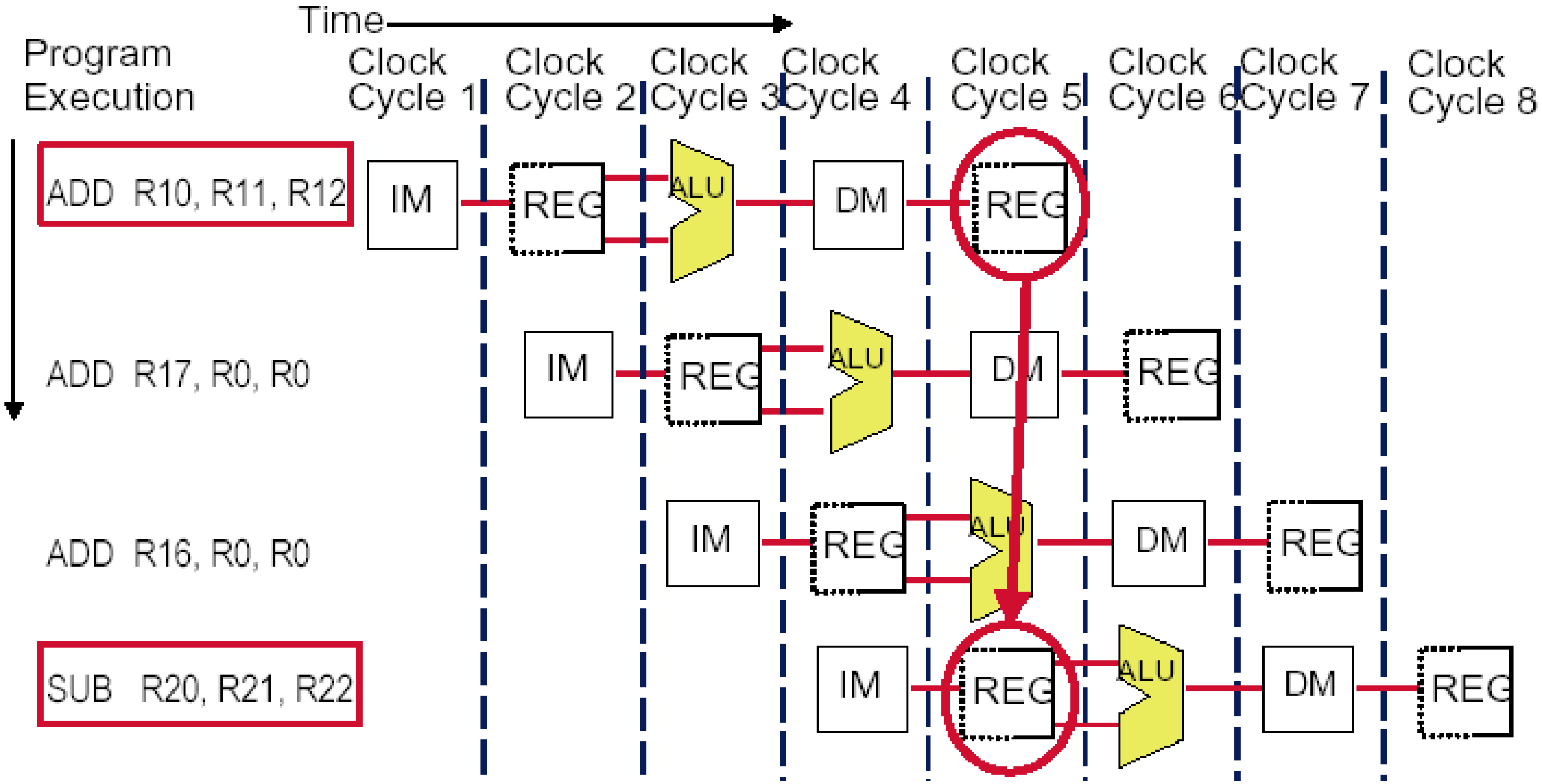
Separate instruction and data memories

- Let's split instruction and data cache



- the memory system must deliver 5 times the bandwidth over the unpipelined version.

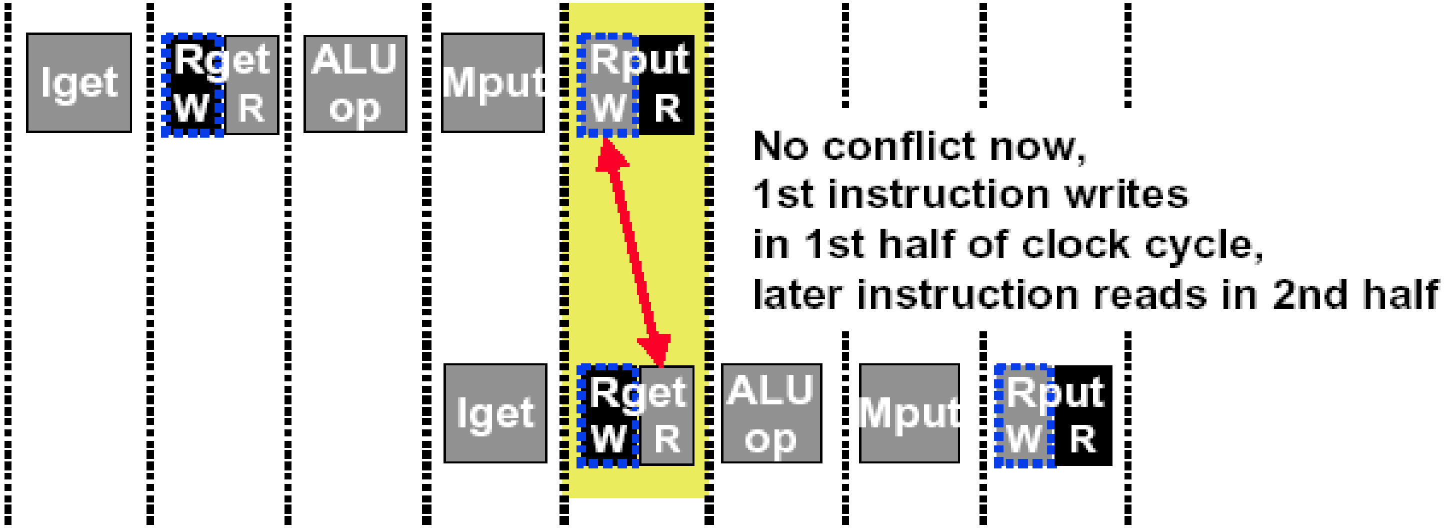
Multi access to the register file



- Simply insert a stall， speedup will be decreased.
- We have resolved it with “ double bump”

Sometimes we can redesign the resource

- allow WRITE-then-READ in one clock cycle (double pump)



No conflict now,
1st instruction writes
in 1st half of clock cycle,
later instruction reads in 2nd half

  - Two reads and one write required per clock.
  - Need to provide two read port and one write port.
- What happens when a read and a write occur to the same register ? (Data hazard)

Not fully pipelined function unit :

Unpipelined Float Adder

<table>
<tr><td>ADDD</td><td>IF</td><td>ID</td><td colspan="6">ADDD</td><td>WB</td><td></td></tr>
<tr><td>ADDD</td><td></td><td>IF</td><td>ID</td><td>stall</td><td>stall</td><td>stall</td><td>stall</td><td>stall</td><td colspan="2">ADDD</td></tr>
</table>

Not fully pipelined Adder

<table>
<tr><td>ADDD</td><td>IF</td><td>ID</td><td colspan="2">A1</td><td colspan="2">A2</td><td colspan="2">A3</td><td>WB</td><td></td></tr>
<tr><td>ADDD</td><td></td><td>IF</td><td>ID</td><td>stall</td><td colspan="2">A1</td><td colspan="2">A2</td><td colspan="2">A3</td></tr>
</table>

Fully pipelined Adder

ADDD	IF	ID	A1	A2	A3	A4	A5	A6	WB	
ADDD		IF	ID	A1	A2	A3	A4	A5	A6	WB

Or multiple unpipelined Float Adder

<table>
<tr><td>ADDD</td><td>IF</td><td>ID</td><td colspan="6">ADDD1</td><td>WB</td><td></td></tr>
<tr><td>ADDD</td><td></td><td>IF</td><td>ID</td><td colspan="6">ADDD2</td><td>WB</td></tr>
</table>

Machine without structural hazards will always have a lower CPI

- Example (pA-14)
  - Data reference constitute 40% of the mix
  - Ideal CPI ignoring the structural hazard is 1
  - The processor with the structural hazard has a clock rate that is 1.05 times higher than that of a processor without structural hazard.
- Answer
  - Average instruction time = CPI×Clock cycle time
  - $=(1+0.4\times 1)\times CC_{ideal}/1.05$
  - $= 1.3 \times CC_{ideal}$
  - Clearly, **the processor without the structural hazard is faster**.

Why allow machine with structural hazard ?

- **To reduce cost.**
  - i.e. adding split caches, requires twice the memory bandwidth.
  - also fully pipelined floating point units costs lots of gates.
  - It is not worth the cost if the hazard does not occur very often.
- **To reduce latency of the unit.**
  - Making functional units pipelined adds delay
  - (pipeline overhead -> registers.)
  - An unpipelined version may require fewer clocks per operation.
  - Reducing latency has other performance benefits, as we will see.

Example: impact of structural hazard to performance

- Example
  - Many machines have unpipelined float-point multiplier.
  - The function unit time of FP multiplier is 6 clock cycles
  - FP multiply has a frequency of 14% in a SPECfp benchmark
  - Will the structural hzard have a large performance impact on the SPECfp benchmark?

Answer to the example

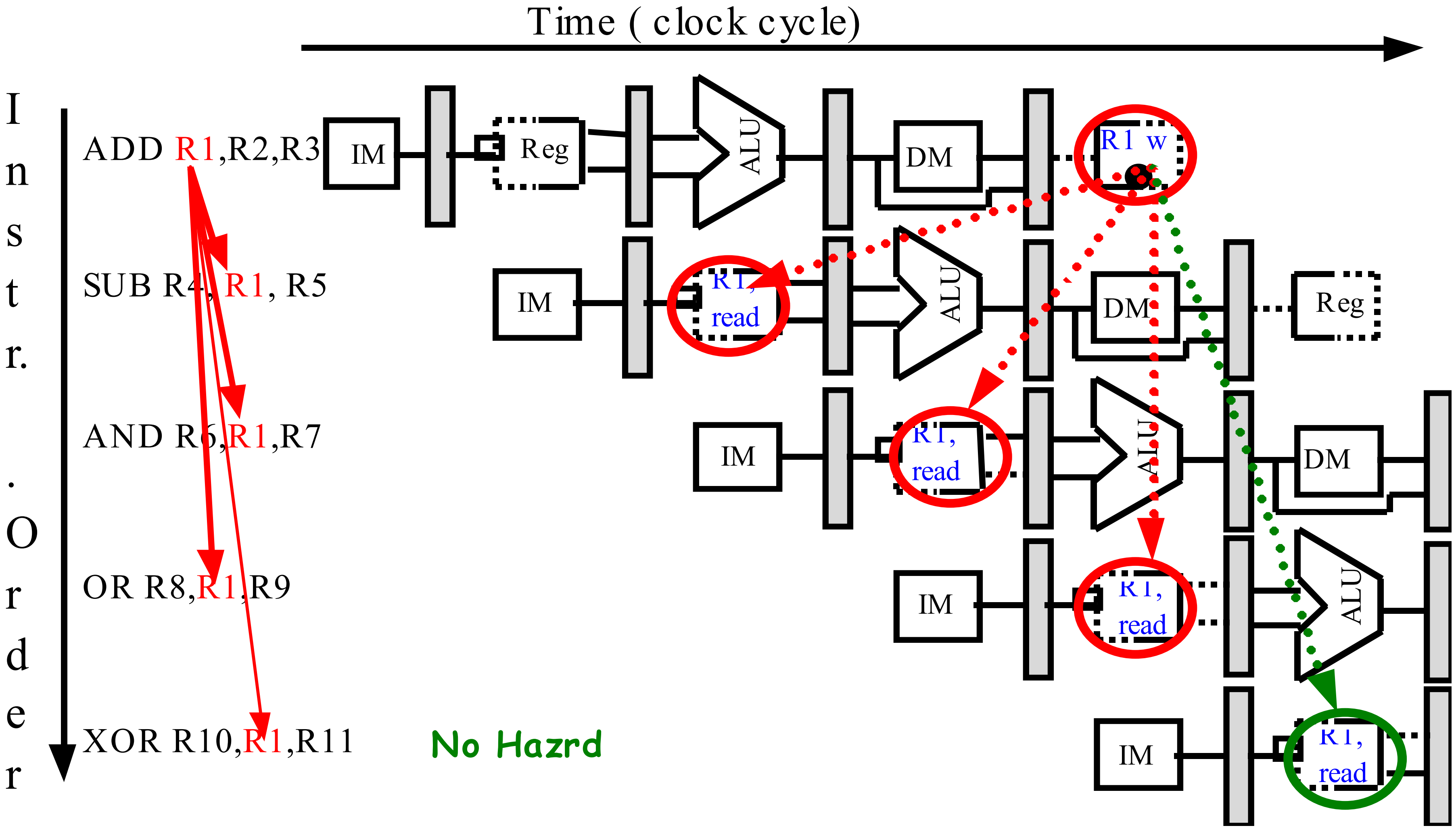
- **In the best case**: FP multiplies are distributed uniformly.
  - There is one multiply in every 7 clock.   1/14%
  - Then there will be no structural hazard, then there is no performance penalty at all.
- **In the worst case**: the multiplies are all clustered with no intervening instructions.
  - Then every multiply instruction have to stall 5 clock cycles to wait for the multiplier be released.
  - The CPI will increase 70% to 1.7, if the ideal CPI is 1.
- Experiment result:
  - This structural hazard increase execution time by less than **3%**.

Data hazard

- Data hazards occur when the pipeline changes the order of read/write accesses to operands comparing with that in sequential executing .
- Let’s see an Example

```
DADD R1,   R1, R3
DSUB R4,   R1, R5
AND   R6,   R1, R7
OR      R8,   R1, R9
XOR    R10, R1, R11
```

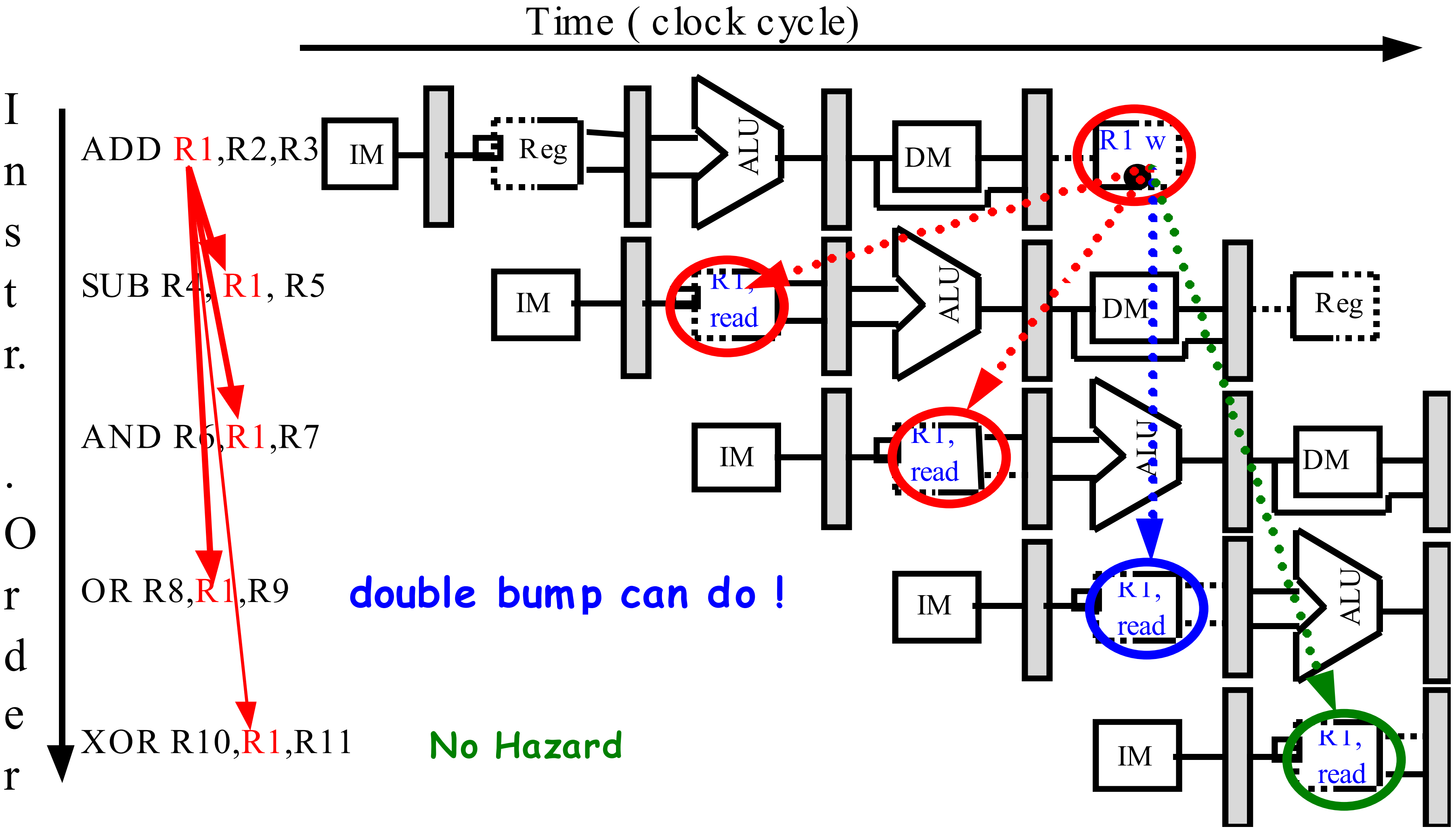
Coping with data hazards:example



Data hazard

- **Basic structure**
  - An instruction in flight wants to use a data value that's **not** "**done**" yet
  - "**Done**" means "it's been computed" and "it's located where I would normally expect to go look in the pipe hardware to find it"
- **Basic cause**
  - You are used to assuming a purely sequential model of instruction execution
  - Instruction N finishes before instruction N+k, for k >= 1
  - There are **dependencies now between "nearby" instructions** ("near" in sequential order of fetch from memory)
- **Consequence**
  - Data hazards -- instructions want data values that are not done yet, or in the right place yet

Somecases “Double Bump” can do !



Proposed solution

- **Proposed solution**
  - **Don’t** let them **overlap** like this…?
- **Mechanics**
  - Don’t let the instruction flow through the pipe
  - In particular, don’t let it **WRITE** any bits anywhere in the pipe hardware that represents **REAL** CPU state (e.g., register file, memory)
  - **Let the instruction wait until the hazard resolved.**
  - Name for this operation: **PIPELINE STALL**
  - **SLOW!**

Forwarding: reduce data hazard stalls

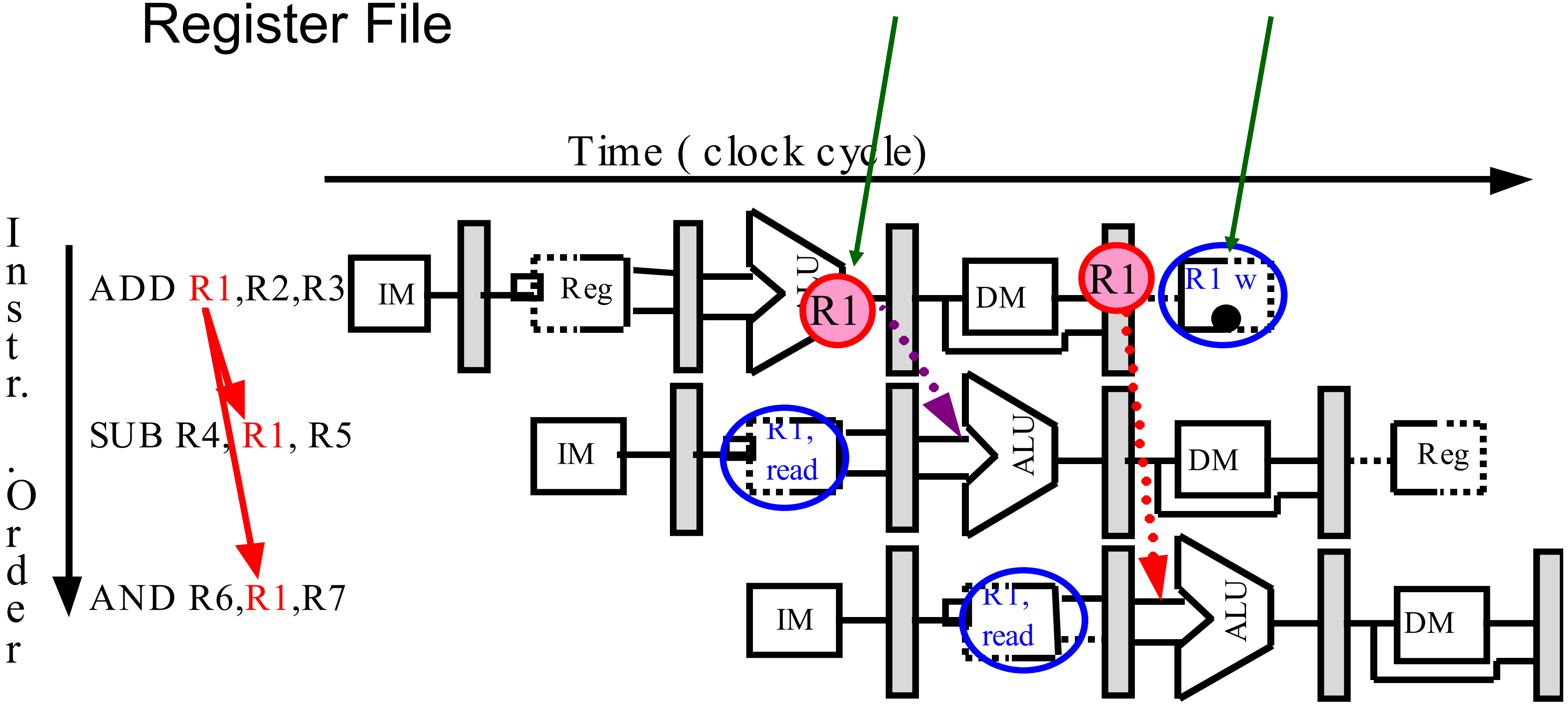
- If the result you need does not exist AT ALL yet,
  - you are out of luck, sorry.
- But, what if the result exists, but is not stored back yet?
  - Instead of stalling until the result is stored back in its “natural” home…
  - **grab the result “on the fly” from “inside” the pipe, and send it to the other instruction (another pipe stage) that wants to use it**

Forwarding

- Generic name: **forwarding (bypass, short-circuiting)**
  - Instead of waiting to store the result, we forward it immediately (more or less) to the instruction that wants it
  - Mechanically, we add buses to the datapath to move these values around, and these **buses** always “point backwards” in the datapath, **from later stages to earlier stages**

Forwarding: reduce data hazard stalls

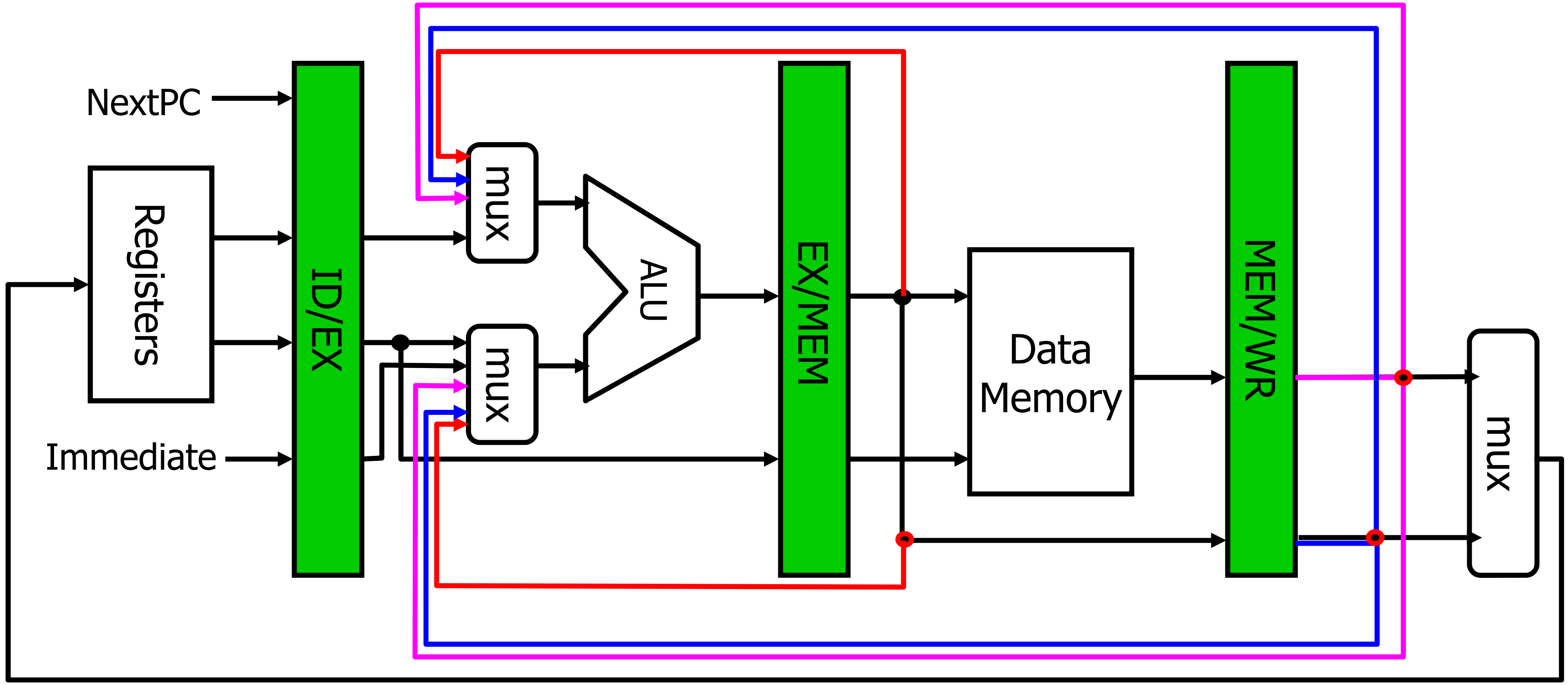
- Data may be already computed - just not in the Register File



EX/MEM.ALUoutput → ALU input port

MEM/WB.ALUoutput → ALU input port

Hardware Change for Forwarding

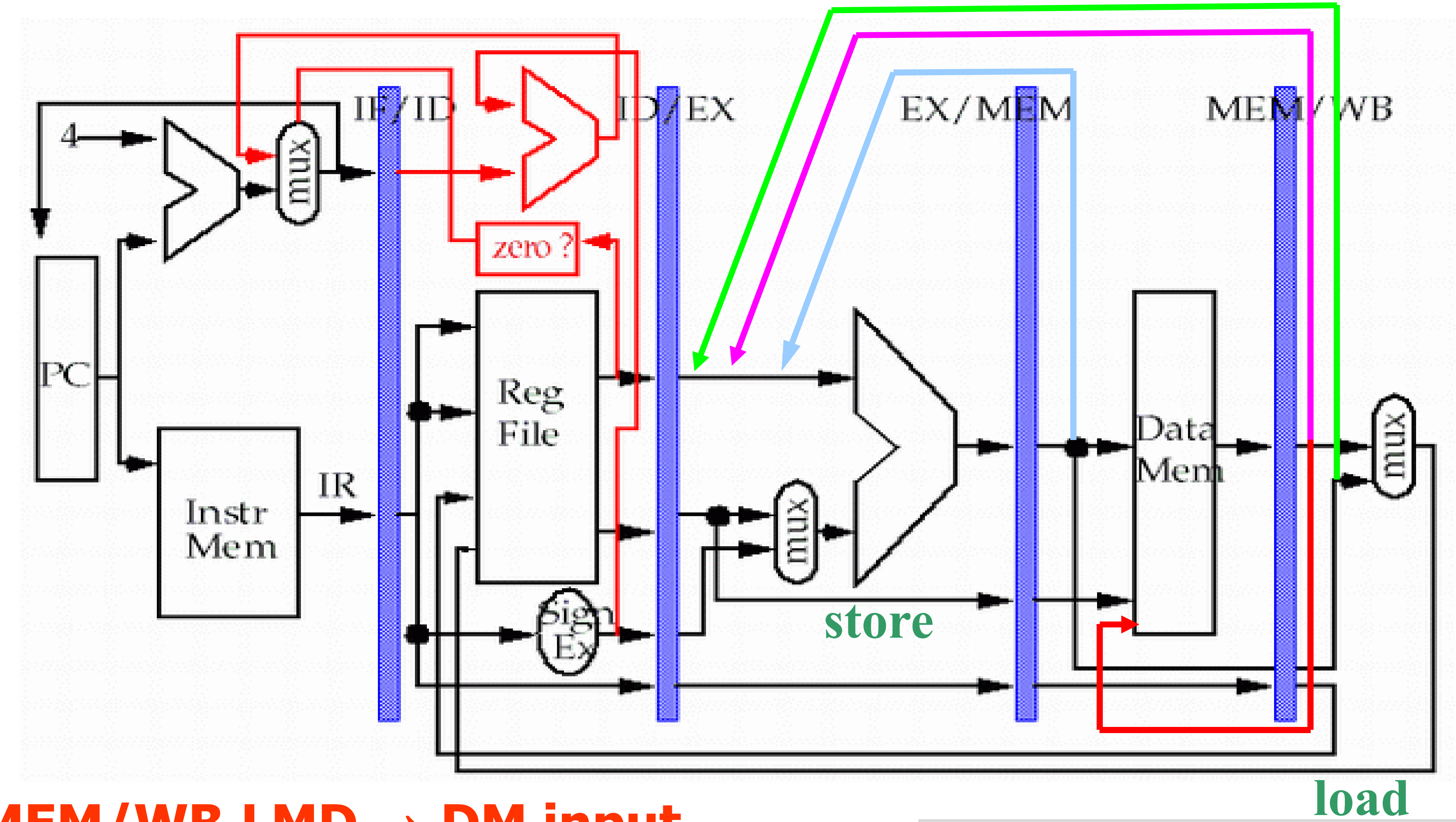


EX/Mem.ALUoutput → ALU input

MEM/WB.ALUoutput → ALU input

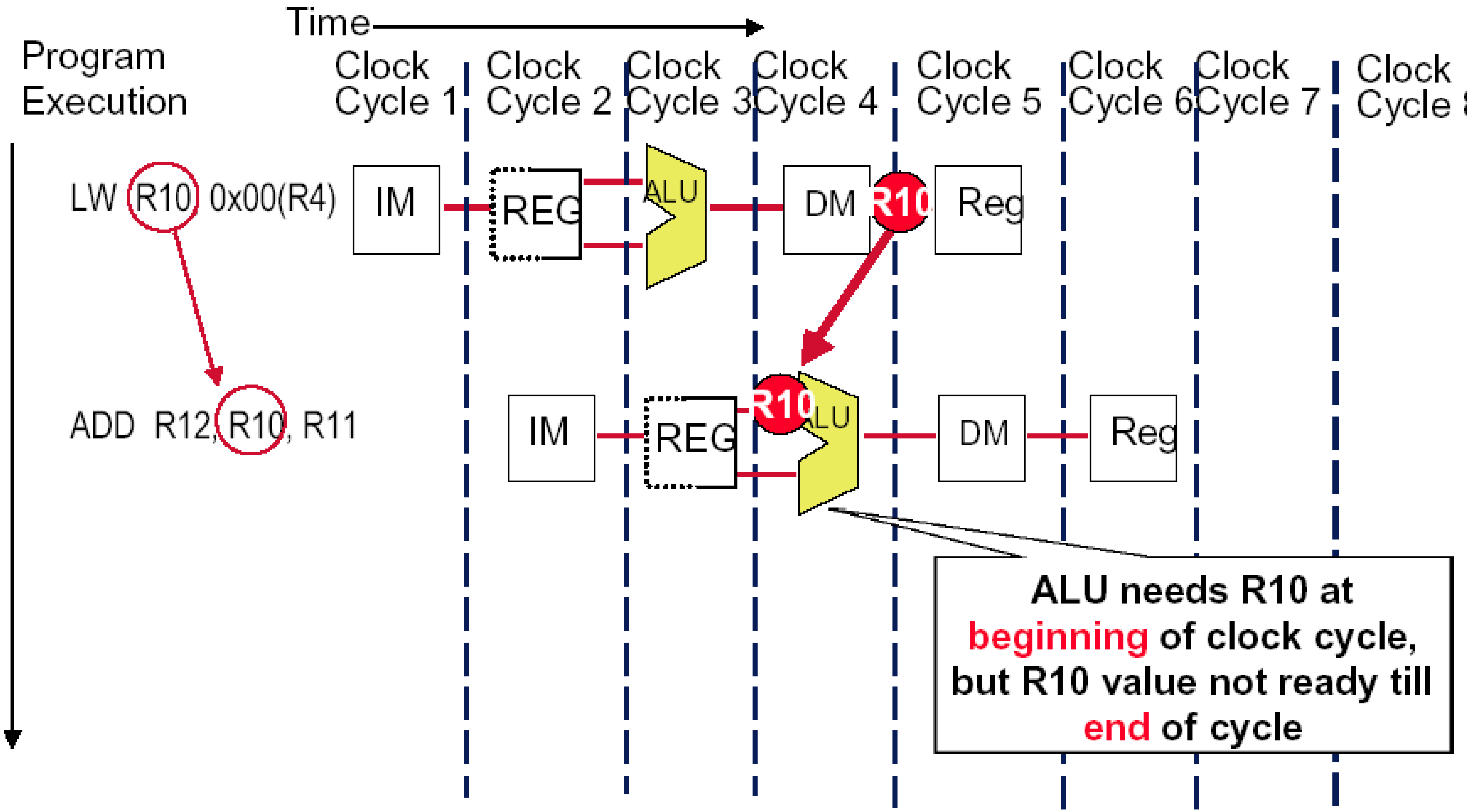
MEM/WB.LMD → ALU input

Forwarding path to other input entry



MEM/WB.LMD → DM input

Forwarding Doesn’t Always Work



ALU needs R10 at beginning of clock cycle, but R10 value not ready till end of cycle

So we have to insert stall: Load stall

Time (clock cycles)

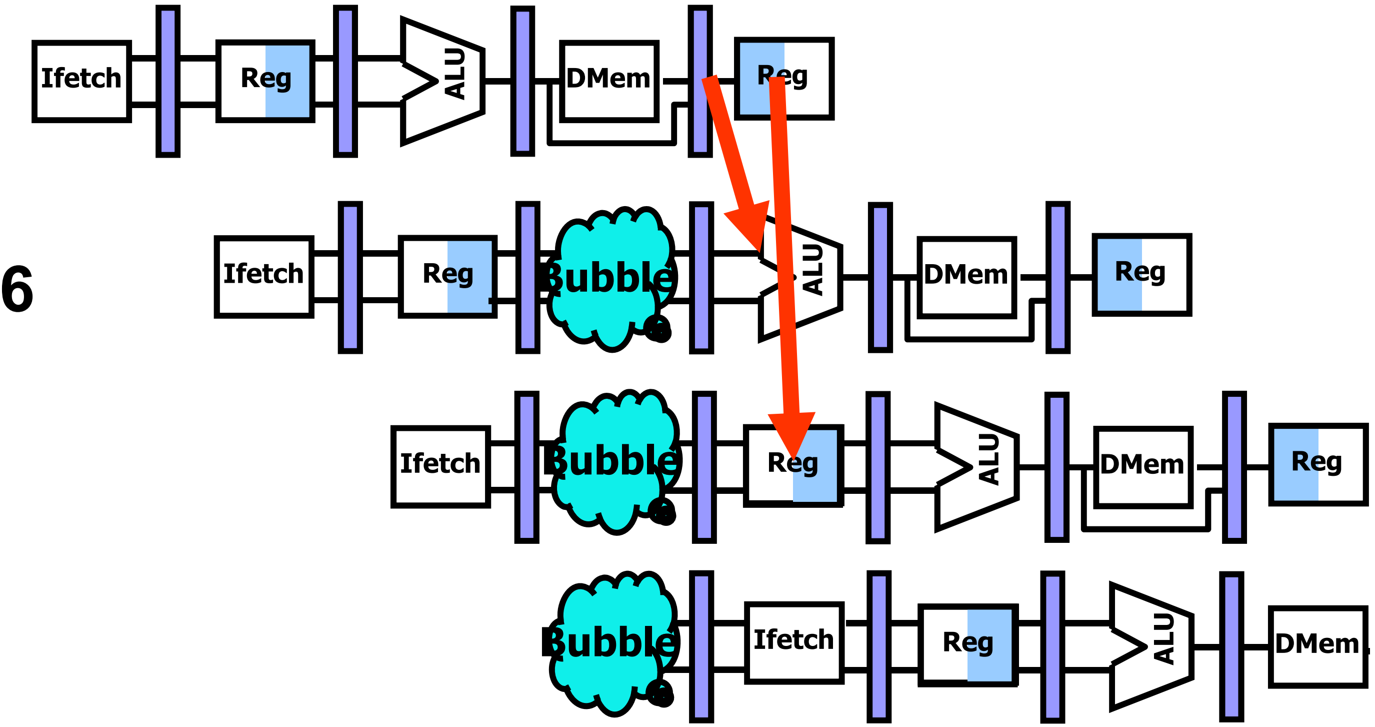
Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or r8,r1,r9



How to implement Load Interlock

- **Detect** when should use Load Interlock

situation	Example code sequence	Action
No dependence	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because of no dependence
Dependence requiring stall	LD R1, 45(R2) DADD R5,R1,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX
Dependence overcome by forwarding	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R1,R7 OR R9,R6,R7	Comparators detect the use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX
Dependence with accesses in order	LD R1, 45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R1,R7	No action required because 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.

The logic to detect for Load interlock

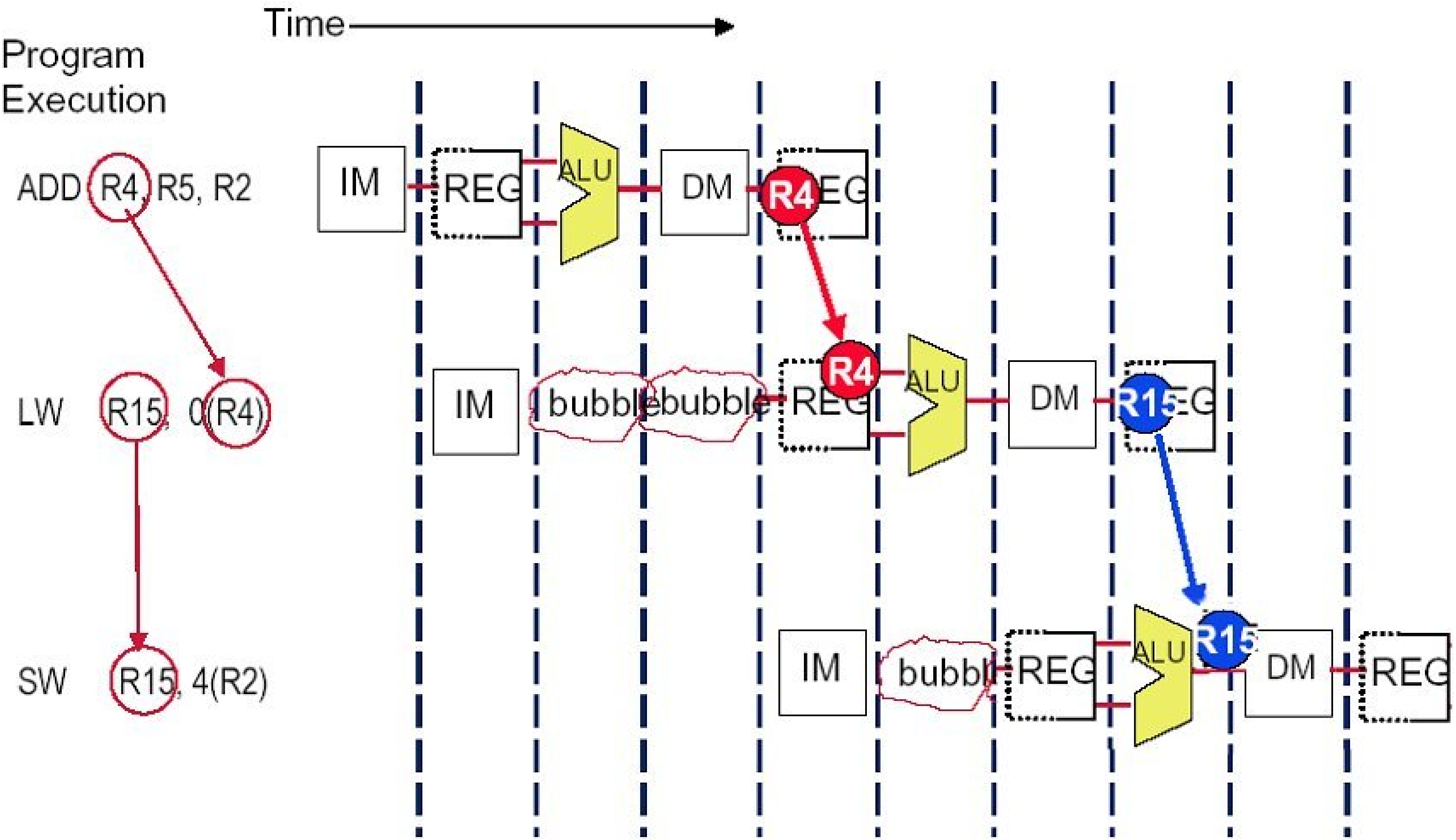
Opcode field of ID/EX	Opcode Field of IF/ID	Matching operand fields
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rs]
Load	Reg-Reg ALU	ID/EX.IR[rt]==IF/ID.IR[rt]
Load	Load,store, ALU immediate, branch	ID/EX.IR[rt]==IF/ID.IR[rs]

Example of Forwarding and Load Delay

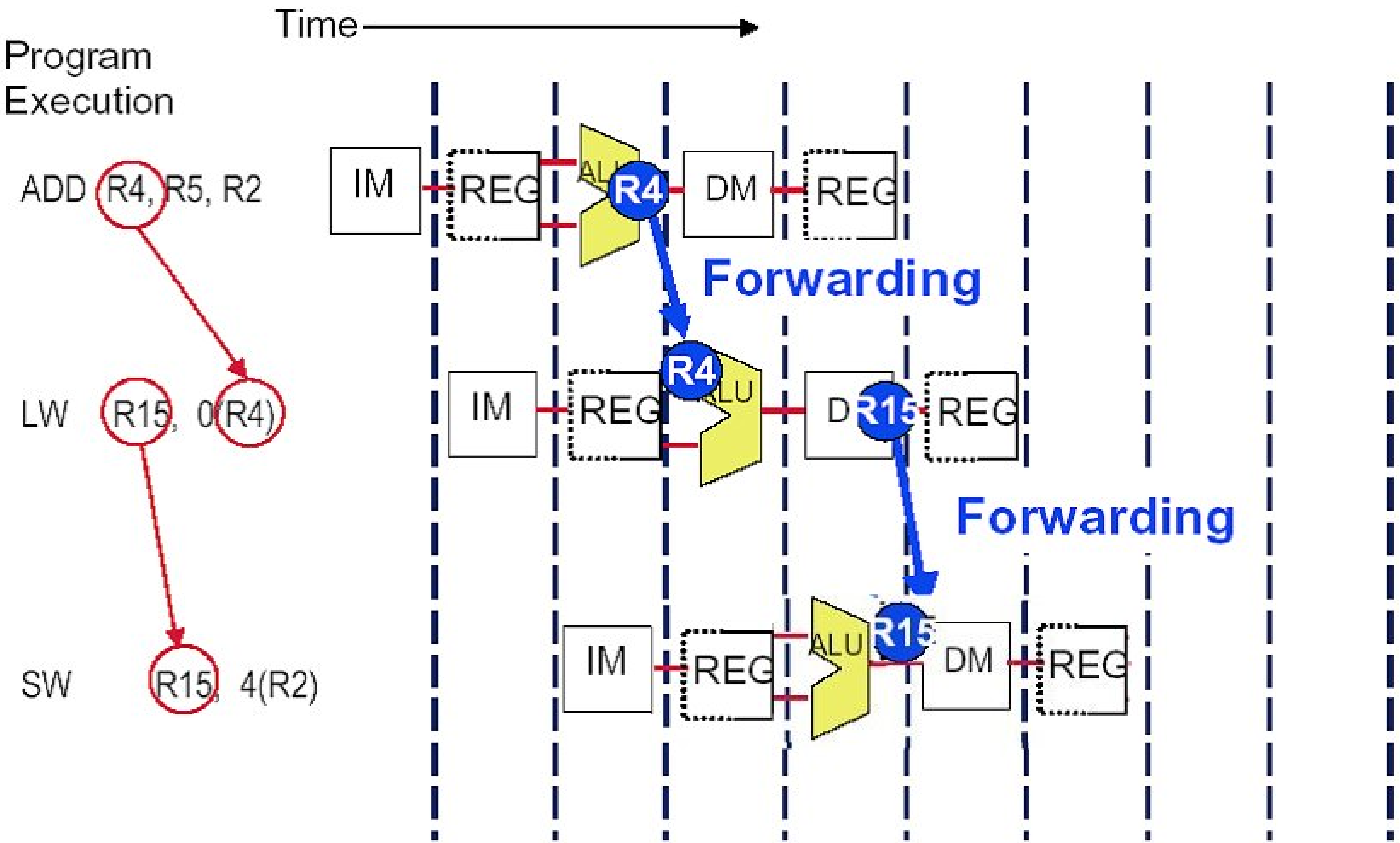
- Why forwarding?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)

- Why load delay?
  - ADD R4, R5, R2
  - LW R15, 0(R4)
  - SW R15, 4(R2)

Solution (without forwarding)



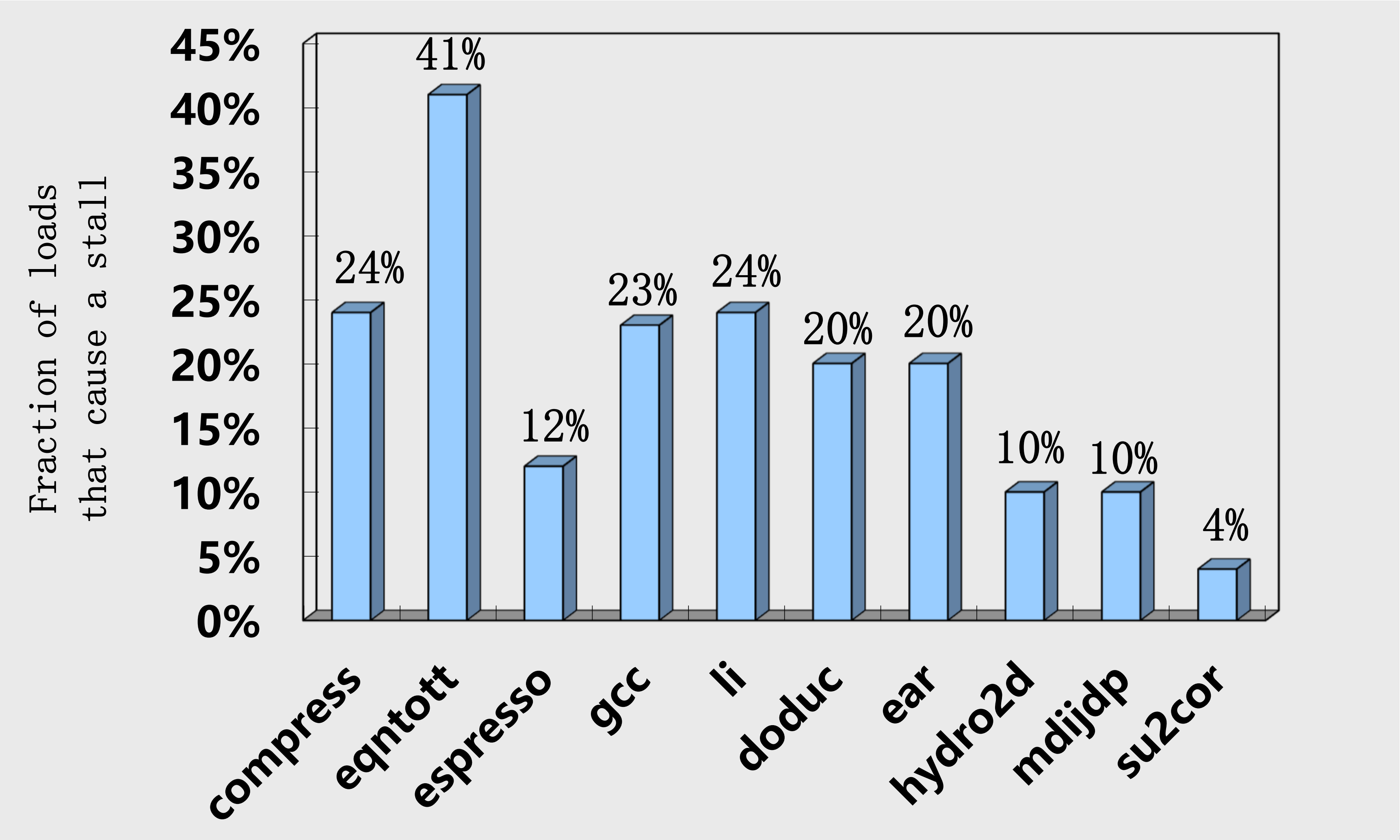
Solution (with forwarding)



The performance influence of load stall

- Example
  - Assume 30% of the instructions are loads.
  - Half the time, instruction following a load instruction depends on the result of the load.
  - If hazard causes a single cycle delay, how much faster is the ideal pipeline ?
- Answer
  - CPI = 1+30%×50% ×1=1.15
  - The performance decrease about **15%** due to load stall.

Fraction of load that cause a stall



Instruction reordering by compiler to avoid load stall

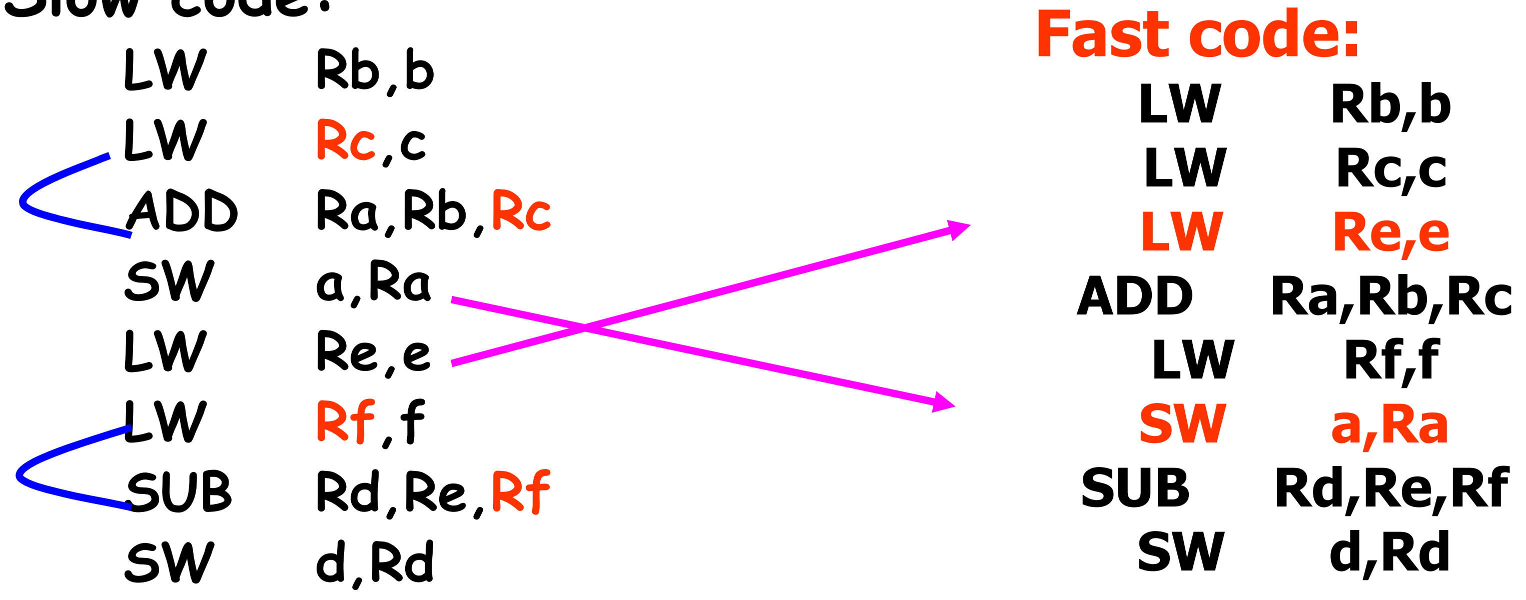
- Try producing fast code for

  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:

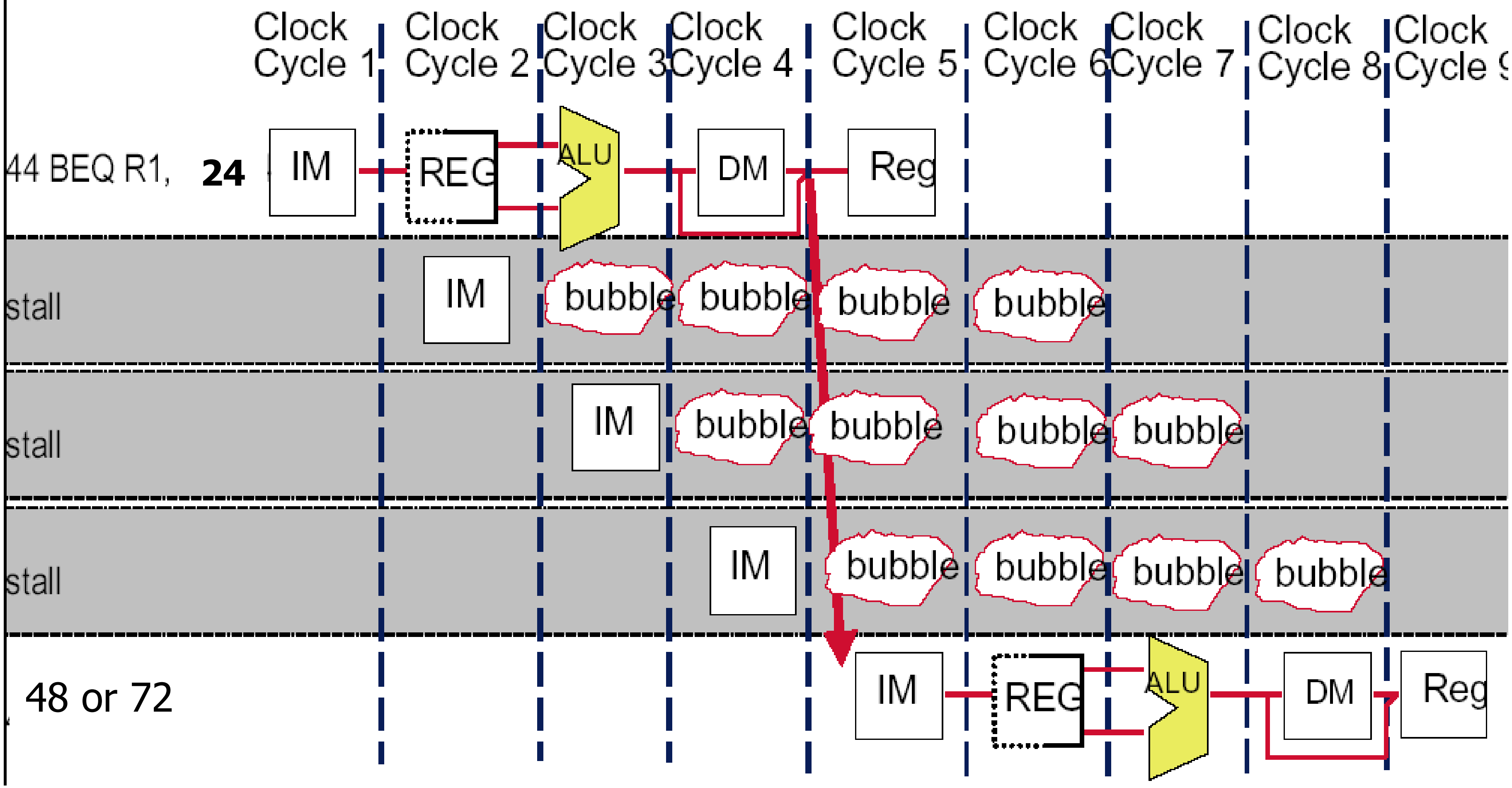
```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```

The Control hazard

- Cause
  - branch condition and the branch PC are not available in time to fetch an instruction on the next clock
  - The next PC takes time to compute
  - For conditional branches, the branch direction takes time to compute.
- **Control hazards can cause a greater performance loss for MIPS pipeline than do data hazards.**

Recall: solve the hazard by inserting stalls

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



The pipeline status

Branch instruction	IF	ID	EX	MEM	WB		
Branch Successor		IF	stall	stall	idle	idle	
Branch successor+1					IF	ID	EX
Branch successor+2						IF	ID
Branch successor+3							IF

Flushing the pipeline

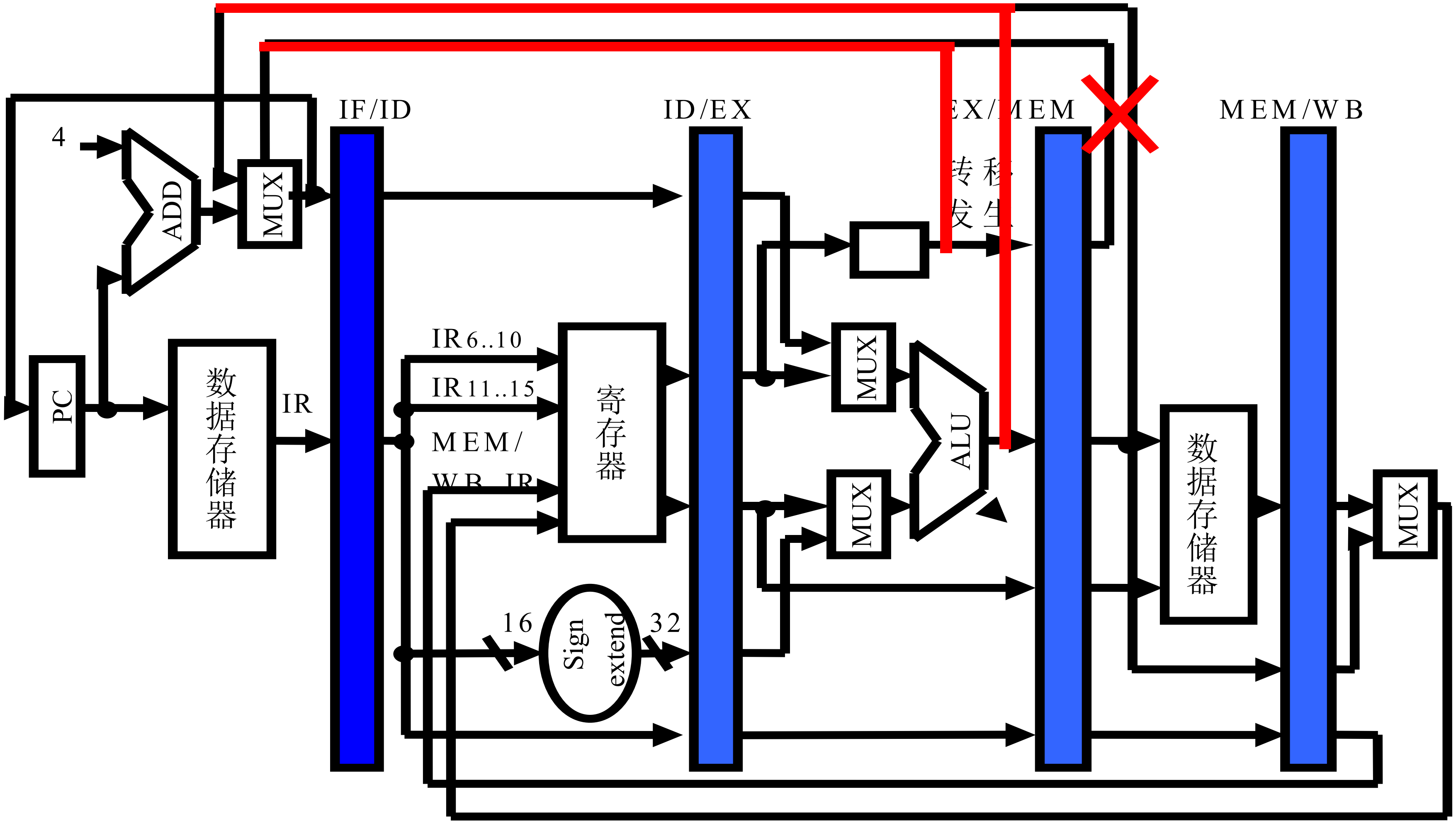
- Simplest hardware:
  - Holding or deleting any instruction after branch until the branch destination is know.
  - Penalty is fixed.
  - Can not be reduced by software.

Stalls greatly hurt the performance

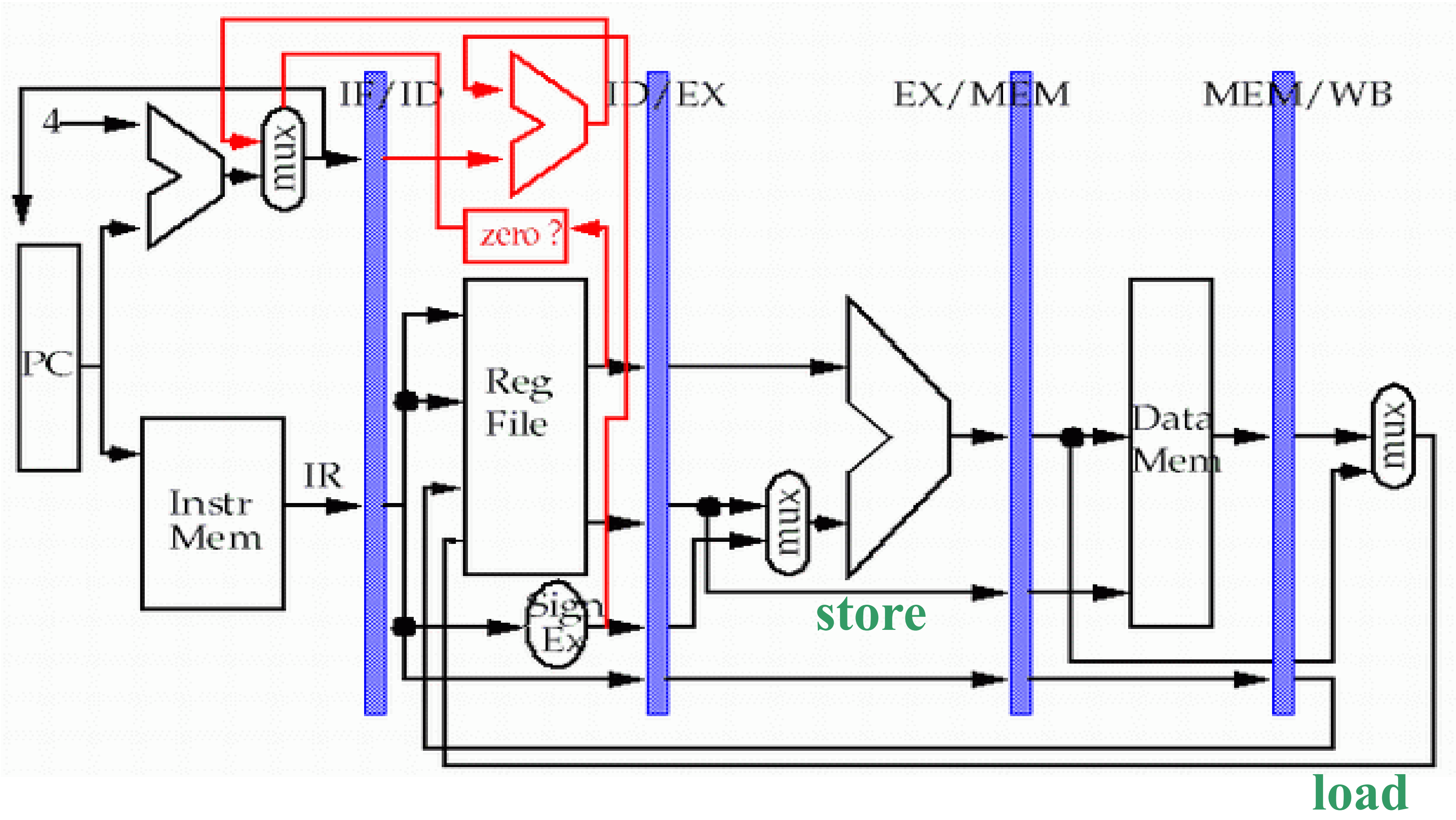
- Problem:
  - With a 30% branch frequency and an ideal CPI of 1, how much the performace is by inserting stalls ?

- Answer:
  - CPI = $1+30\%\times 3 = 1.9$
  - this simple solution achieves only about half of the ideal performance.

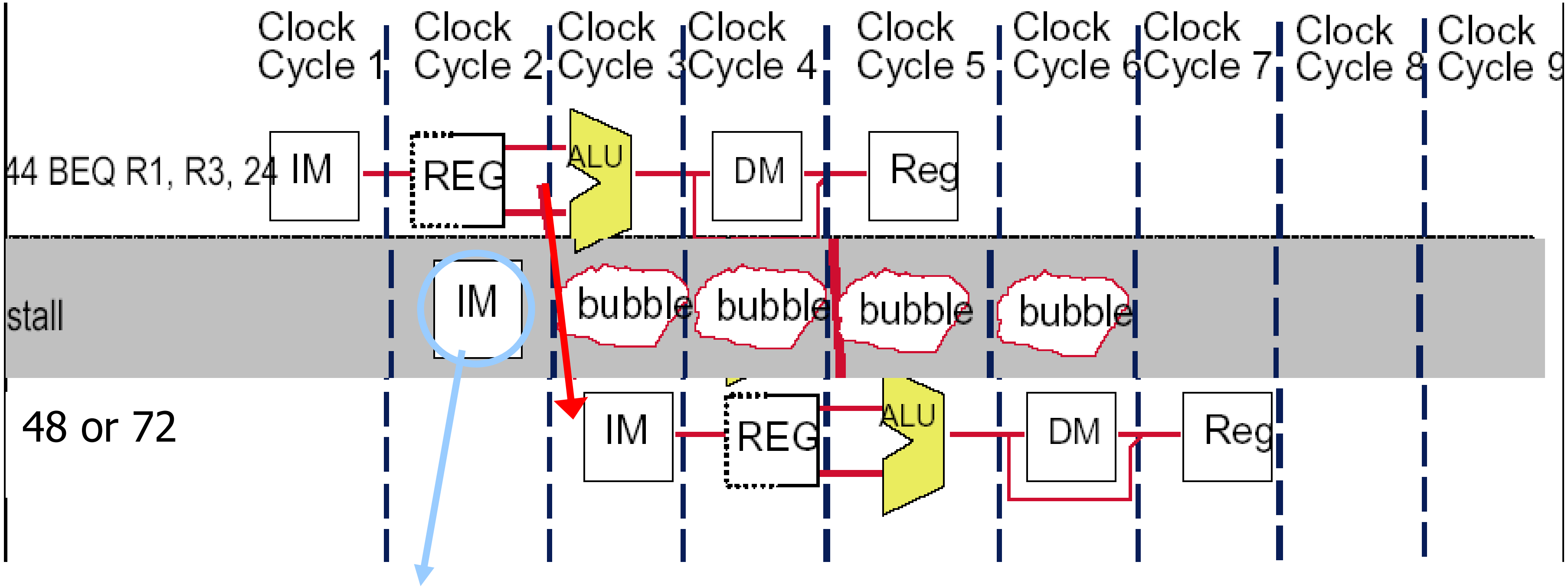
Move the Branch Computation Forward



Move the Branch Computation more Forward



Why “waste” the fetched instruction ?



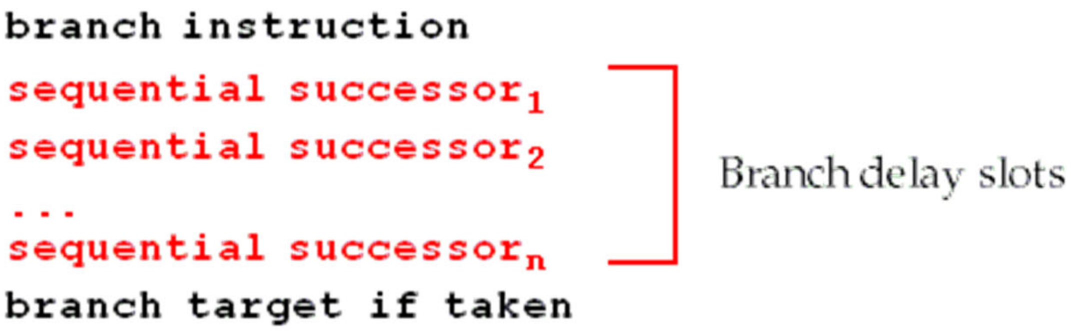
- We have fetched the instruction 48, why we fetch the second time if the branch not taken at last ?

Delayed branch

- 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
  - And on MIPS, **this instruction always executes, no matter whether the branch taken or not taken. (hardware scheme)**

Branch delay slot

- Hence the name: **branch delay slot**

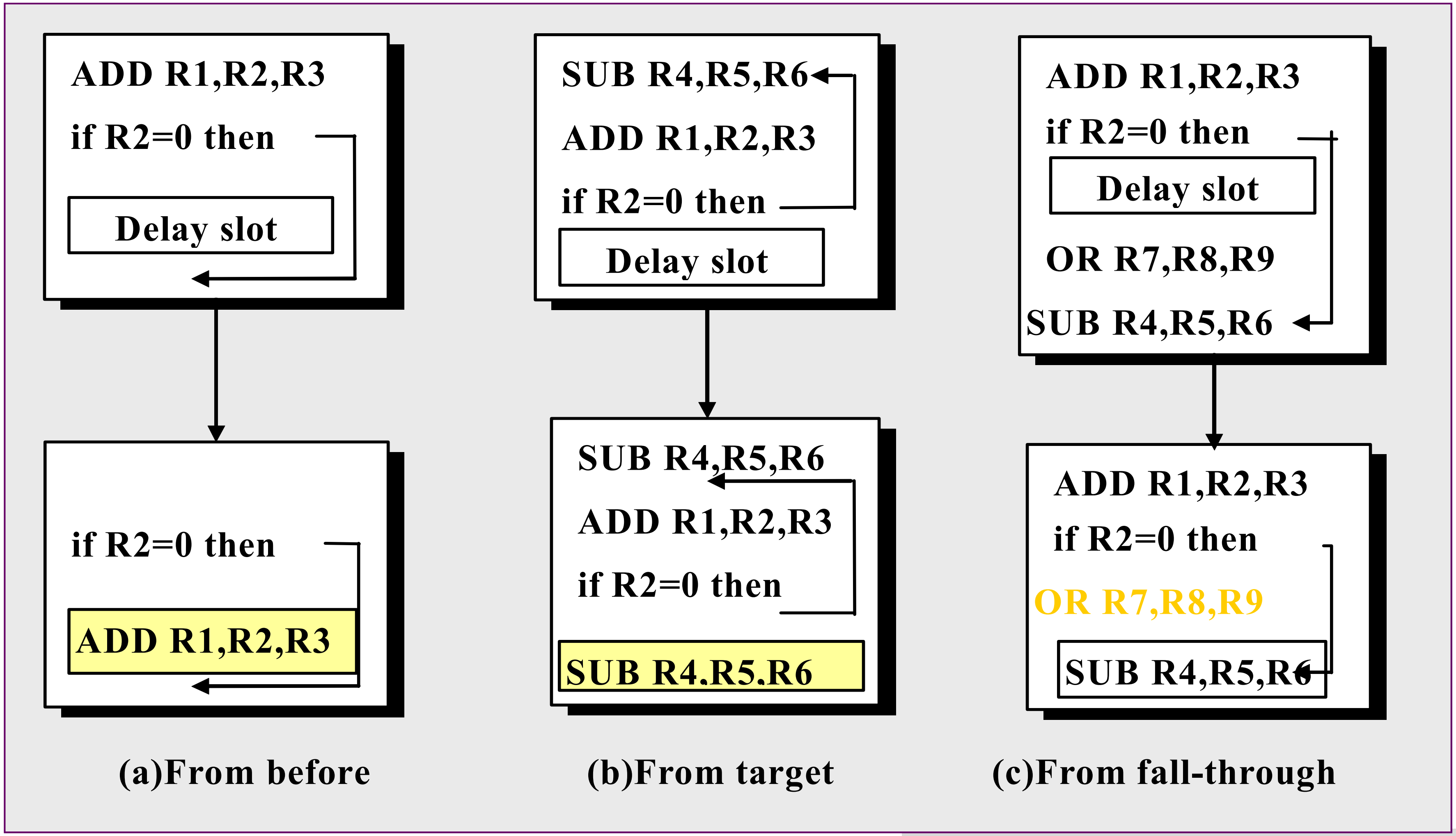


branch instruction
sequential successor$_1$
sequential successor$_2$
...
sequential successor$_n$
branch target if taken

  - The instruction cycle after the branch is used for address calculation , 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 scheme)

How to adjust the codes?



(a)From before

(b)From target

(c)From fall-through

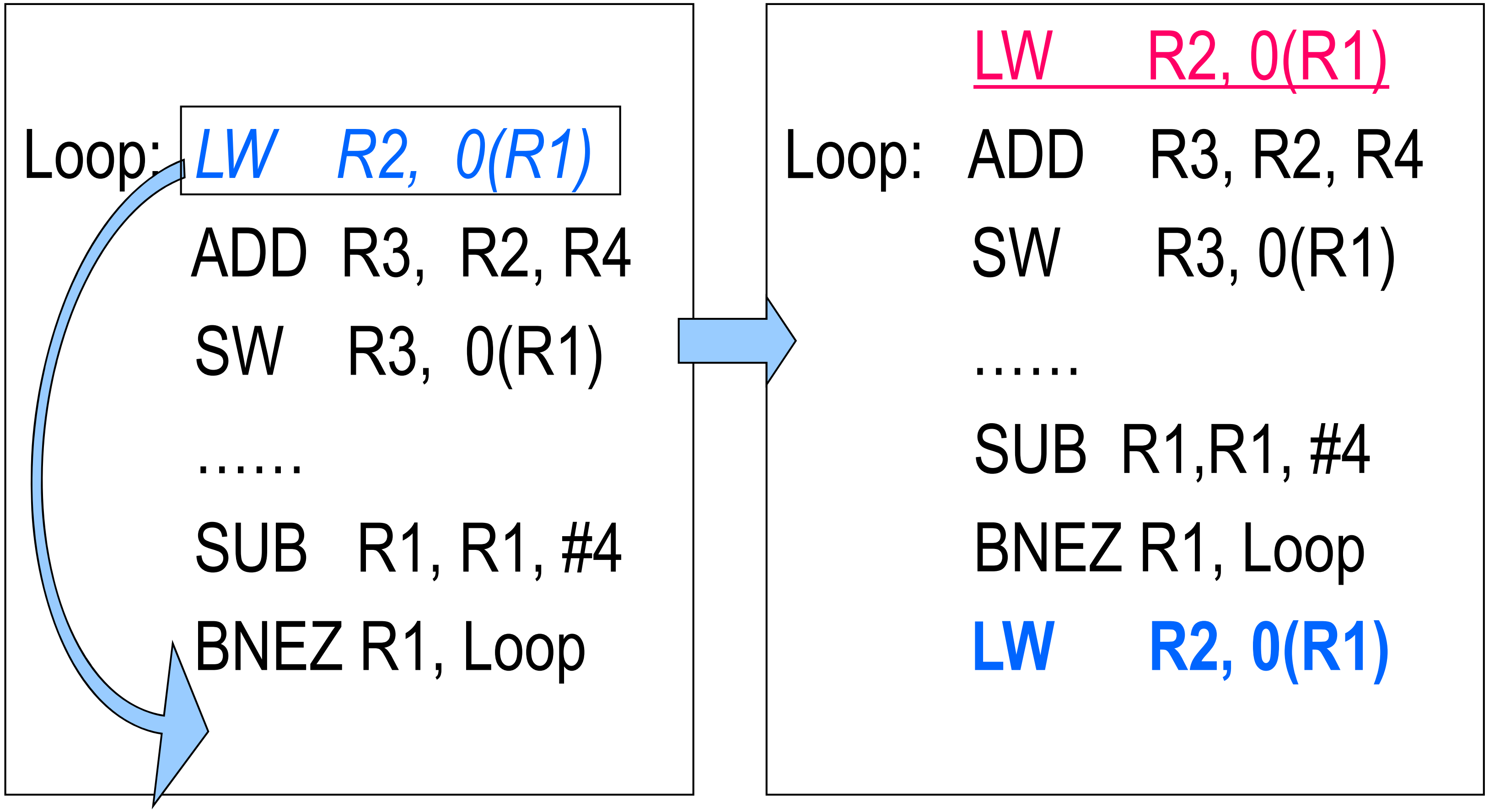
Example: rewrite the code (a)

- ❑ Without Branch Delay Slot — With Branch Delay Slot

Address	Instruction	Address	Instruction
36	NOP	36	NOP
40	ADD R30,R30,R30	40	BEQ R1, R3, 28
44	BEQ R1, 24	44	ADD R30, R30, R30
48	AND R12, R2, R5	48	AND R12, R2, R5
52	OR R13, R6, R2	52	OR R13, R6, R2
56	ADD R14, R2, R2	56	ADD R14, R2, R2
60	...	60	...
64	...	64	...
68	...	68	...
72	LW R4, 50(R7)	72	LW R4, 50(R7)
76	...	76	...

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

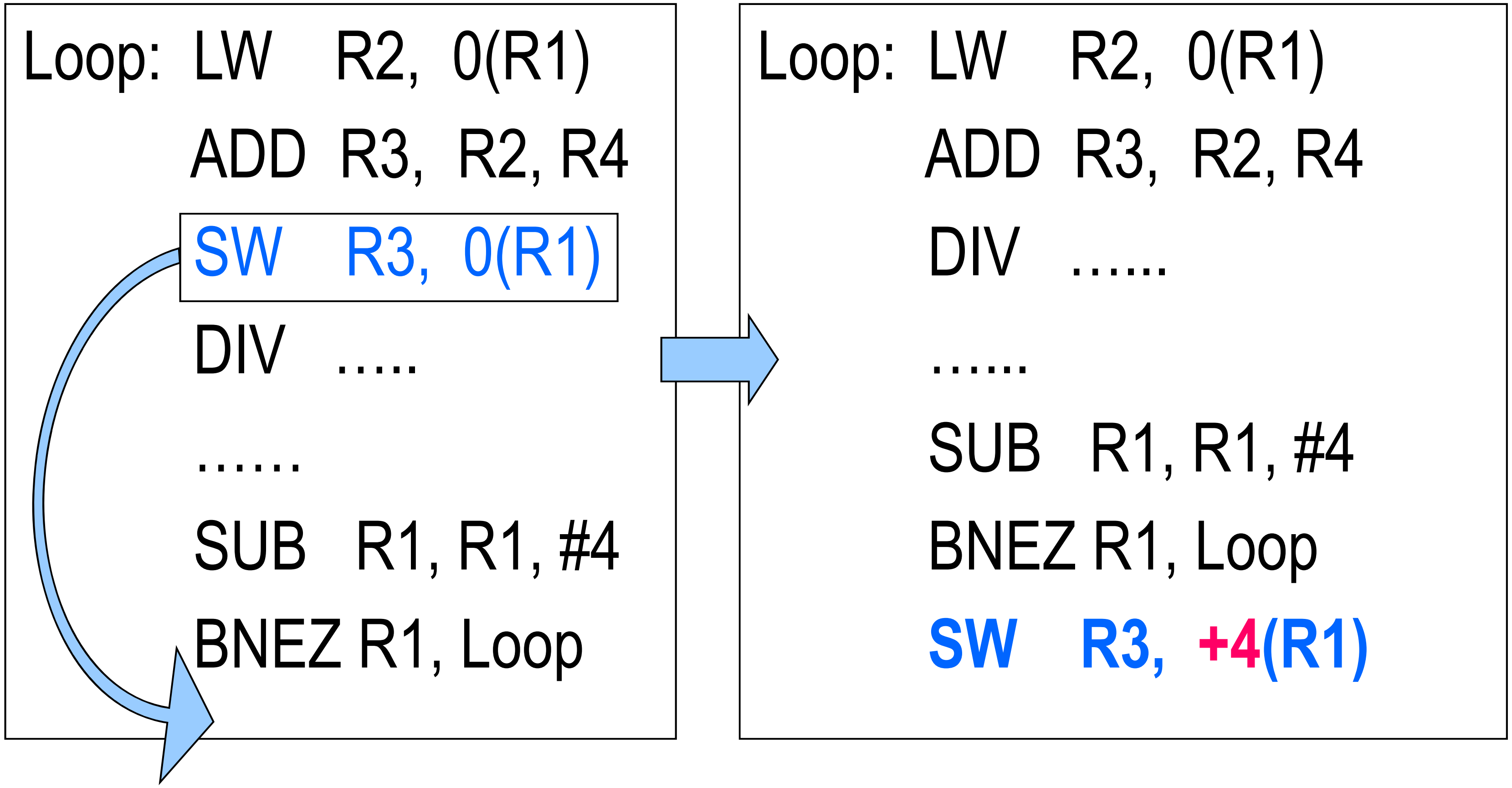
Example: rewrite the code (b-1)



```
Loop: LW   R2,  0(R1)
      ADD  R3,  R2, R4
      SW   R3,  0(R1)
      ......
      SUB  R1, R1, #4
      BNEZ R1, Loop
```

```
      LW    R2, 0(R1)
Loop: ADD   R3, R2, R4
      SW    R3, 0(R1)
      ......
      SUB  R1,R1, #4
      BNEZ R1, Loop
      LW    R2, 0(R1)
```

Example: rewrite the code (b-2)



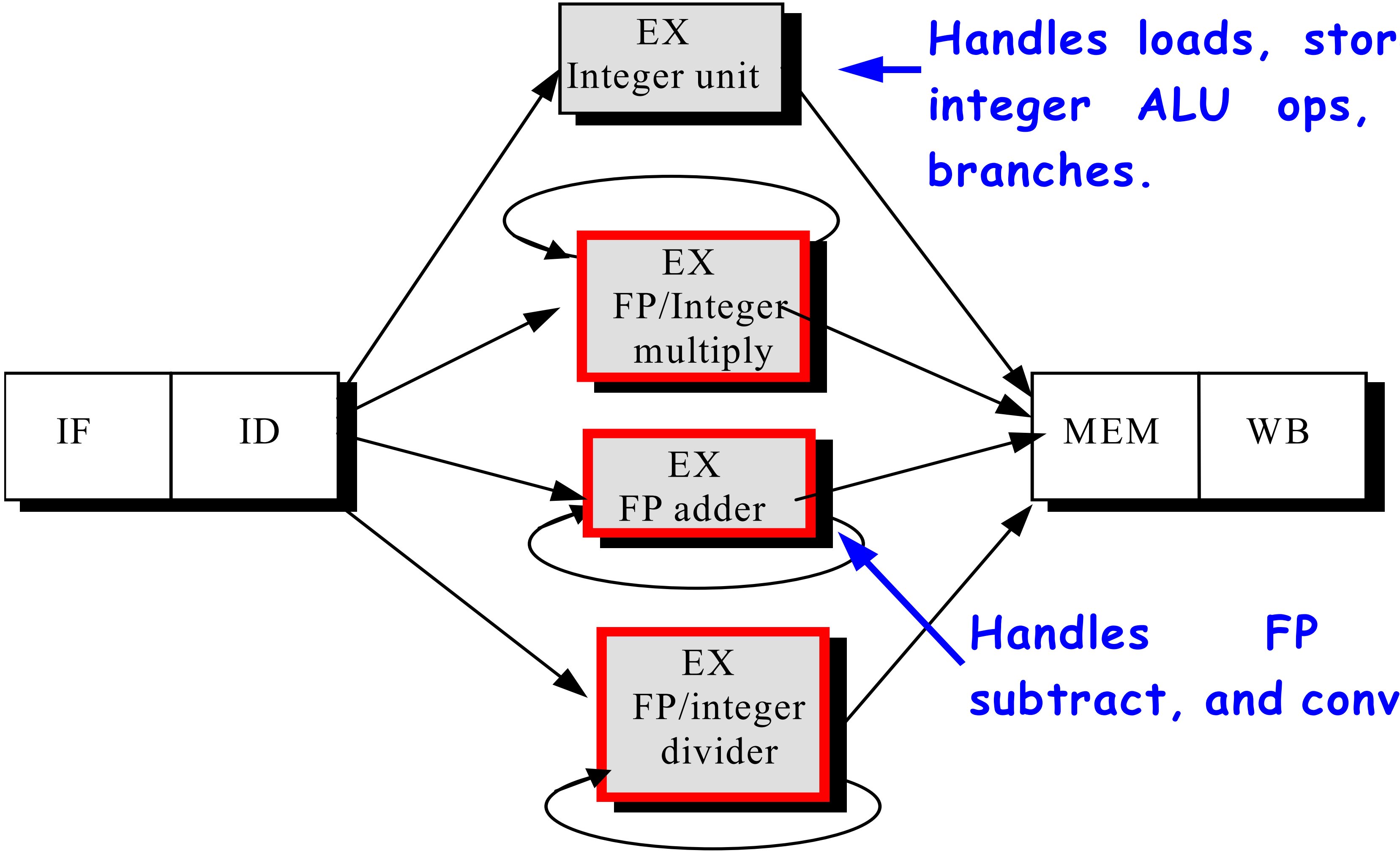
```
Loop:  LW    R2,  0(R1)
       ADD  R3,  R2, R4
       SW    R3,  0(R1)
       DIV   …..
       ……
       SUB   R1, R1, #4
       BNEZ R1, Loop
```

```
Loop:  LW    R2,  0(R1)
       ADD  R3,  R2, R4
       DIV   ……
       ……
       SUB   R1, R1, #4
       BNEZ R1, Loop
       SW    R3,  +4(R1)
```

Extending the MIPS pipeline to handle MultiCycle Operations

- Alternative resolutions to handle floating-point operations
  - Complete operation in 1 or 2 clock cycles,
    - Which means using a slow clock,
    - or/and using enormous amounts of logic in FP units.
  - **Allow for a longer latency for operations**
    - The **EX** cycle may be repeated as many times as needed to complete the operation
    - There may be multiple FP units

MIPS pipeline with FP units



Handles loads, stores , integer ALU ops, and branches.

Handles FP add, subtract, and conversion

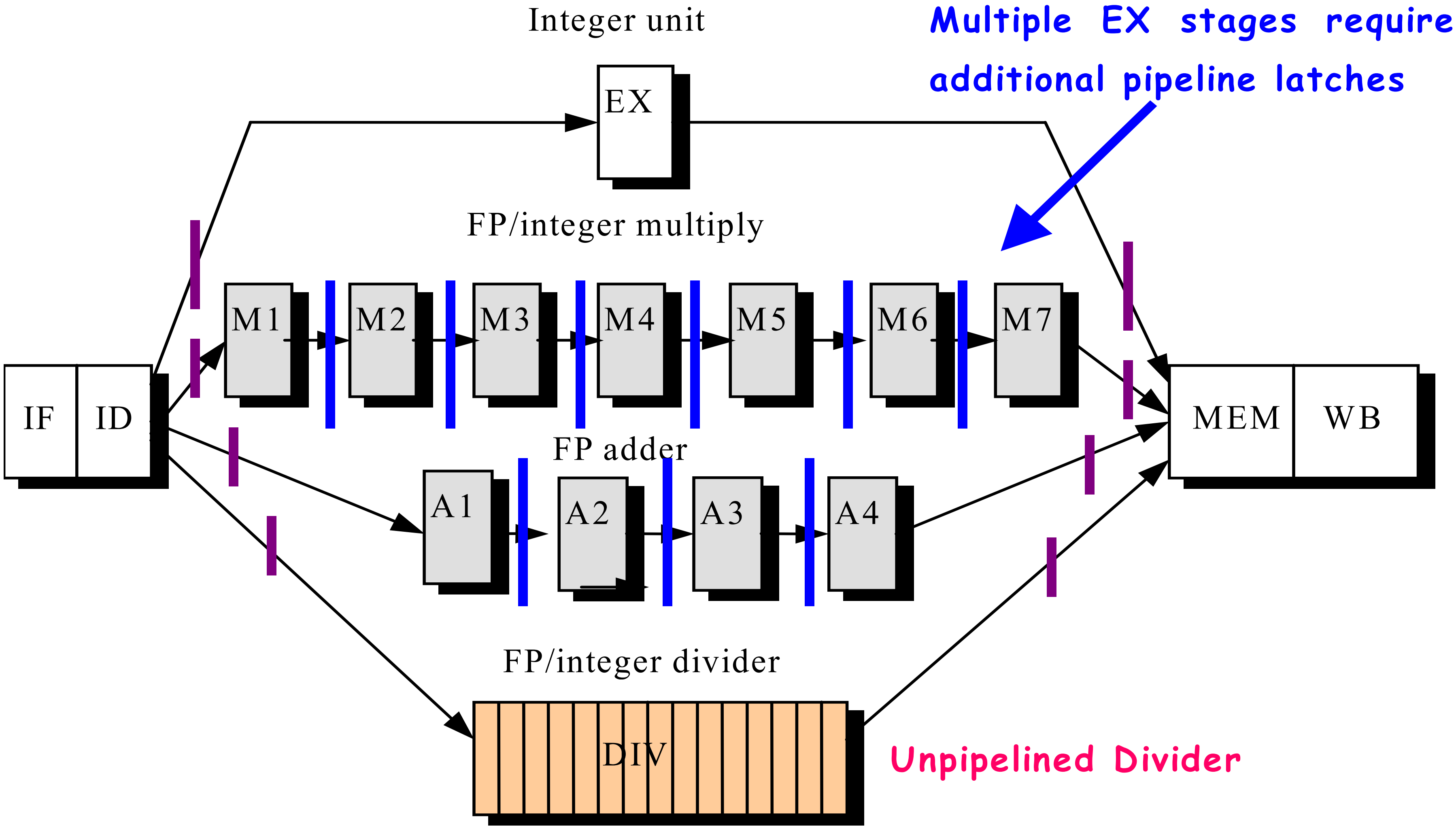
Pipelining some of the FP units

- Two terminologies
  - **Latency**---the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
  - **Initiation interval**---the number of cycles that must elapse between instructions issue to the same unit.
    - For full pipelined units, initiation interval is 1
    - For unpipelined units, initiation interval is always the latency plus 1.

Latencies and initiation intervals for functional units

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory(integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Pipeline supports multiple outstanding FP operations



Multiple EX stages require additional pipeline latches

Unpipelined Divider

Specifications

- Memory bandwidth: double words/one cycle
- New pipeline latches are required:
  - M1/M2, M2/M3, M3/M4, M4/M5, M5/M6, M6/M7
  - A1/A2, A2/A3, A3/A4
- New connection registers are required:
  - ID/EX, ID/M1, ID/A1, ID/DIV
  - EX/MEM, M7/MEM, A4/MEM, DIV/MEM
- Because the divider unit is unpipelined, structural hazards can occur.
- Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1
- New data hazards: WAW is possible due to disorder WBs
- Due to longer latency of operations, stalls for RAW hazards will be more frequent.
- Problems with exceptions resulting from disorder completion

Issuing in order and completion out of order

Instruction	1	2	3	4	5	6	7	8	9	10	11		
MUL.D	IF	ID	***M1***	M2	M3	M4	M5	M6	**M7**	MEM	WB		
ADD. D		IF	ID	***A1***	A2	A3	**A4**	MEM	WB				
MUL.D			IF	ID	***M1***	M2	M3	M4	M5	M6	**M7**	MEM	WB
LD.D				IF	ID	***EX***	**MEM**	WB					
SD.D					IF	ID	***EX***	***MEM***	WB				

Structural Hazards for the FP register write port

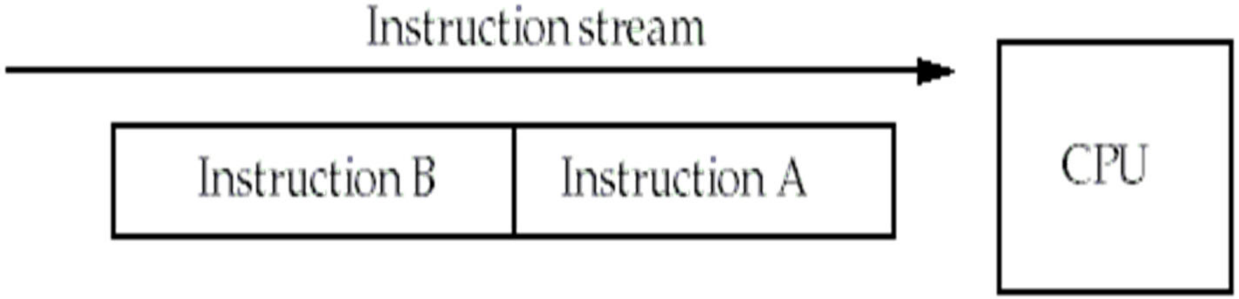
Instruction	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EXi	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	WB

How to solve the write port conflict ?

- Increase the number of write ports
  - Unattractive at all !
  - No worthy since steady state usage is close to 1.
- Detect and insert stalls by serializing the writes
  - Track the use of the write port in the ID stage and to stall an instruction before it issues
    - Additional Hardware: a shift register+ write conflict logic
    - The shift register tracks when already-issued instructions will use the register file, and right shift 1 bit each clock.
    - The stalls might *aggravate* the data hazards
    - All interlock detection and stall insertion occurs in ID stage
  - To stall a conflicting instruction when it tries to enter the MEM or WB stage.
    - Easy to detect the conflict at this point
    - Complicates pipeline control since stalls can now occur in two places.

Types of data hazards

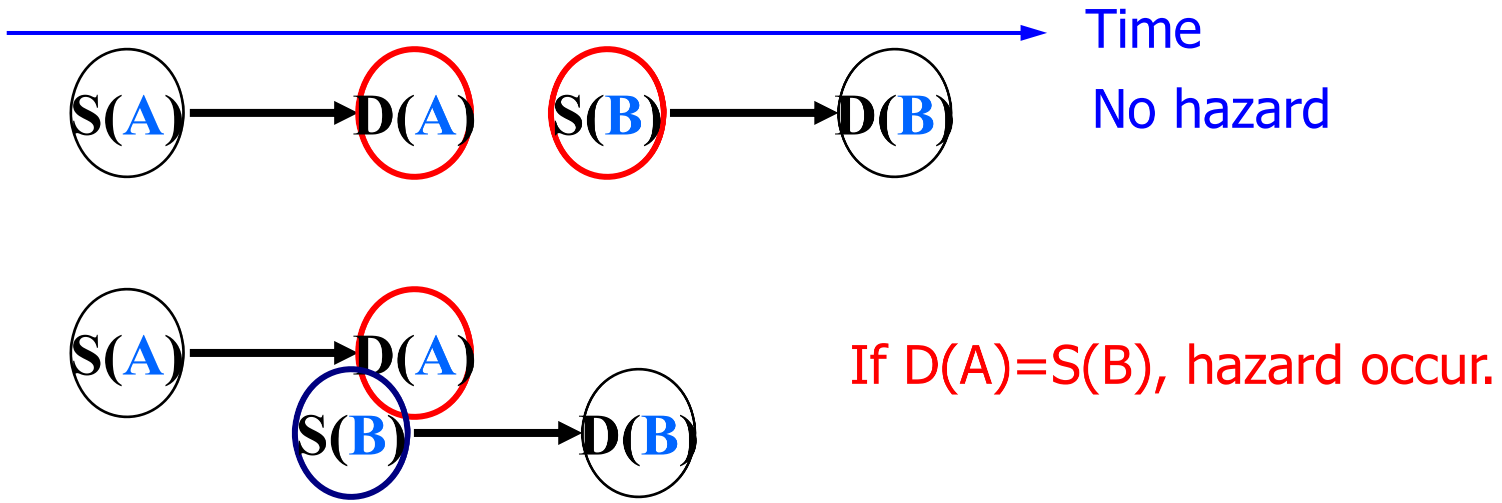
- Consider two instructions, A and B. A occurs before B.



- **RAW(Read after write) true dependence**
  - Instruction A writes Rx，instruction B reads Rx
- **WAW(Write after write) output dependence**
  - Instruction A writes Rx，instruction B writes Rx
- **WAR(Write after read) anti-denpendence**
  - Instruction A reads Rx，instruction B writes Rx
- Hazards are named according to the ordering **that MUST be preserved by the pipeline**

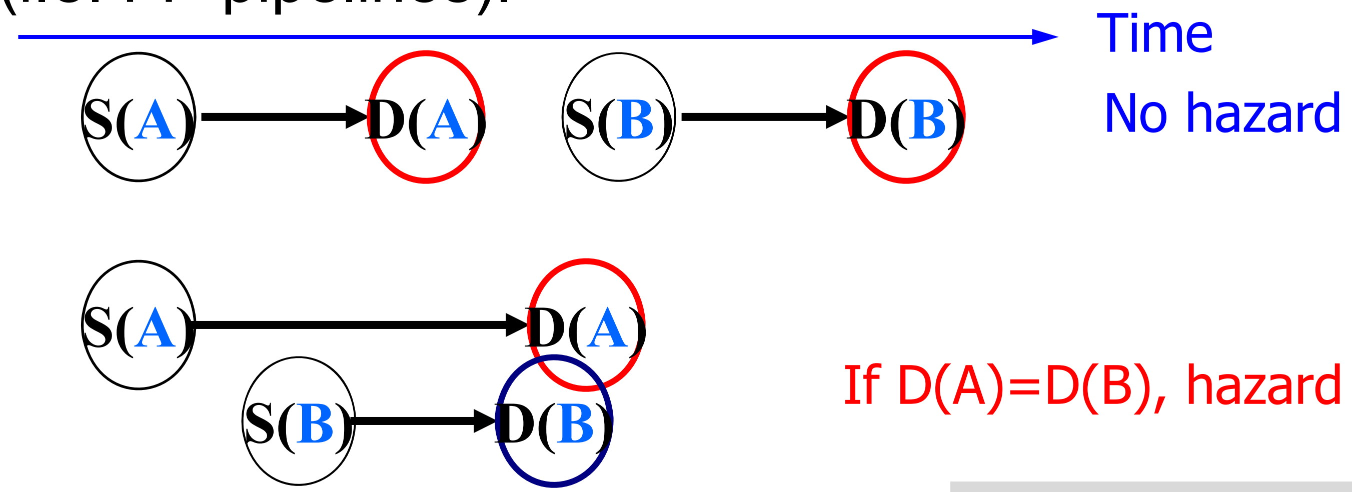
RAW dependence

- B tries to read a register before A has written it and gets the old value.
- This is common, and forwarding helps to solve it.



WAW dependence

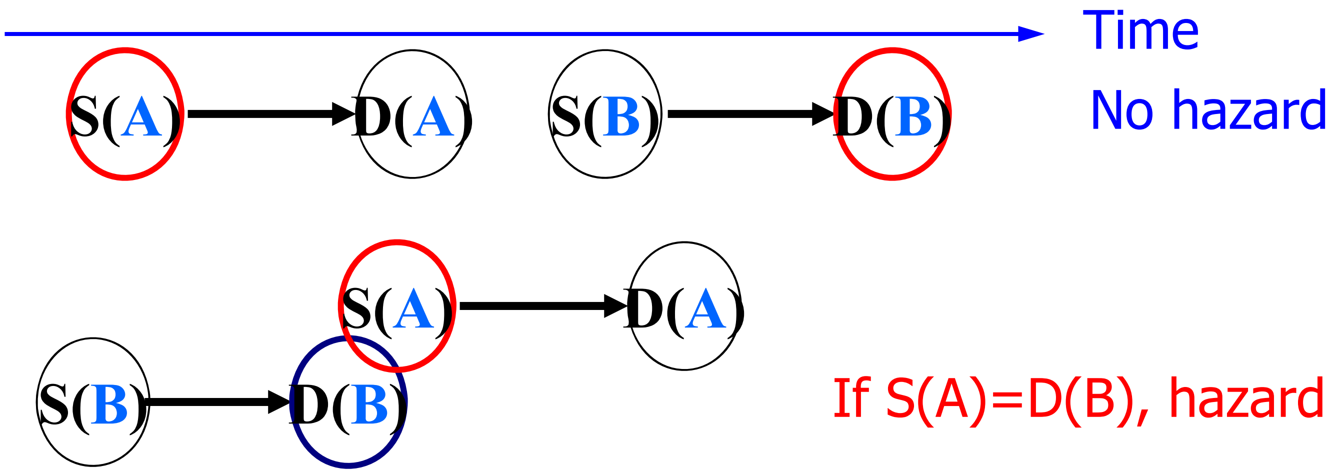
- B tries to write an operand before A has written it.
- After instruction B has executed, the value of the register should be B's result, but A's result is stored instead.
- This can only happen with pipelines that write values in more than one stage, or in variable-length pipelines (i.e. FP pipelines).



If D(A)=D(B), hazard occur.

WAR dependence

- B tries to write a register before A has read it.
- In this case, A uses the new (incorrect) value.
- This type of hazard is rare because most pipelines read values early and write results late.
- However, it might happen for a CPU that had complex addressing modes. i.e. autoincrement.



If S(A)=D(B), hazard occur.

Stalls arising from RAW hazards

Instruction	1	2	3	4	5	6	7	8	9	10	11						
LD F4, 0(R2)	IF	ID	EX	MEM	WB												
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
SD 0(R2) , F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

The WAW hazards

Instruction	1	2	3	4	5	6	7	8	9	10	11		
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB		
…		IF	ID	EXi	MEM	WB							
…			IF	ID	EX	MEM	WB						
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	S	MEM	WB	
…					IF	ID	EX	MEM	WB				
LD F2, 0(R2)						IF	ID	EX	MEM	WB			
LD F8, 0(R2)							IF	ID	EX	S	S	MEM	WB

Solving the WAW hazard

- **Stall an instruction** that would "pass" another until after the earlier instruction reaches the MEM phase.
- **Cancel the WB phase of the earlier instruction**
- Both of these can be done in ID, i.e. when LD is about to issue.
- Since pure WAW hazards are not common, either method works.
- Pick the one that simplest to implement.
- The simplest solution for the MIPS pipeline is to hold the instruction in ID if it writes the same register as an instruction already issued.

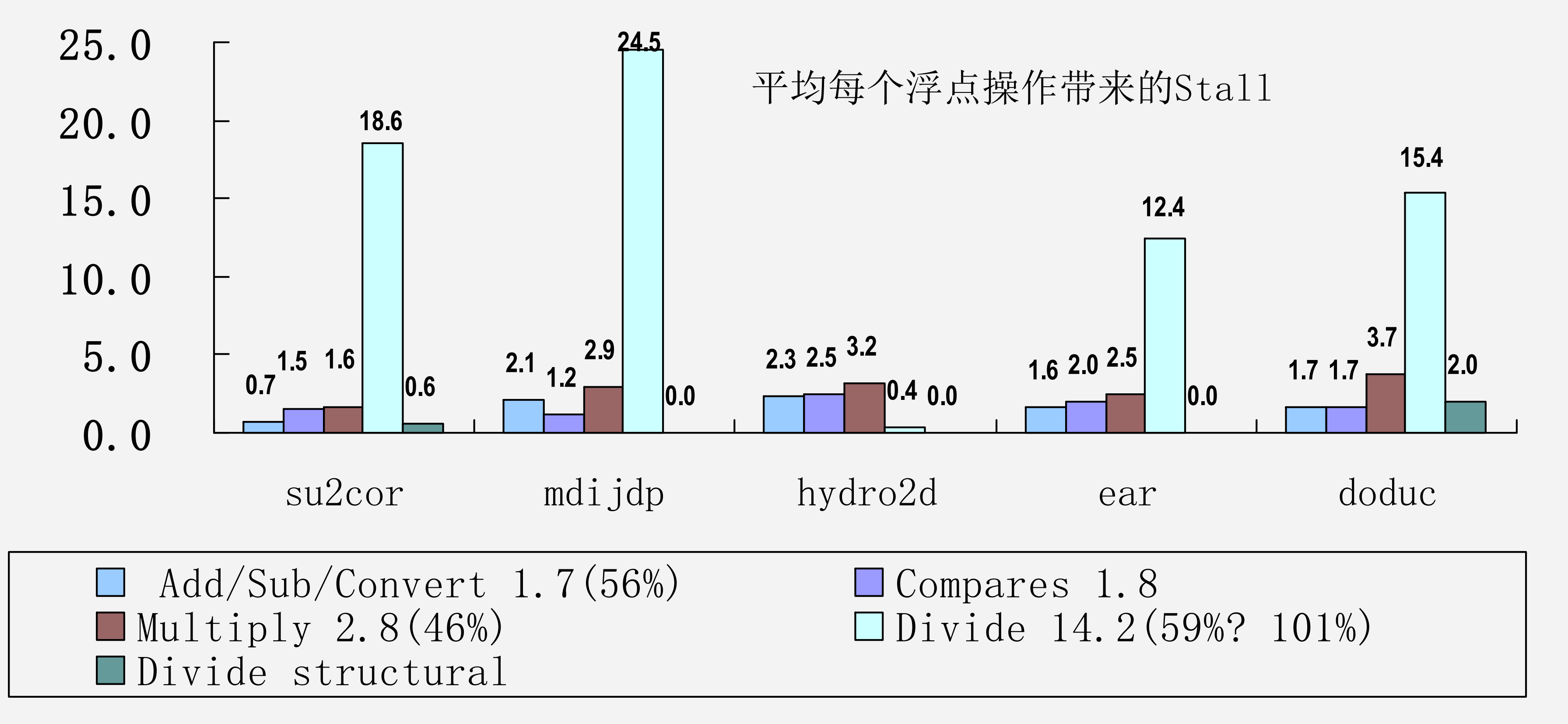
What other hazards are possible ?

- Hazards among FP instructions.
- Hazards between an FP instruction and an integer instruction.
  - Since two register files exist, only FP loads and stores and FP register moves to integer registers involve hazards.

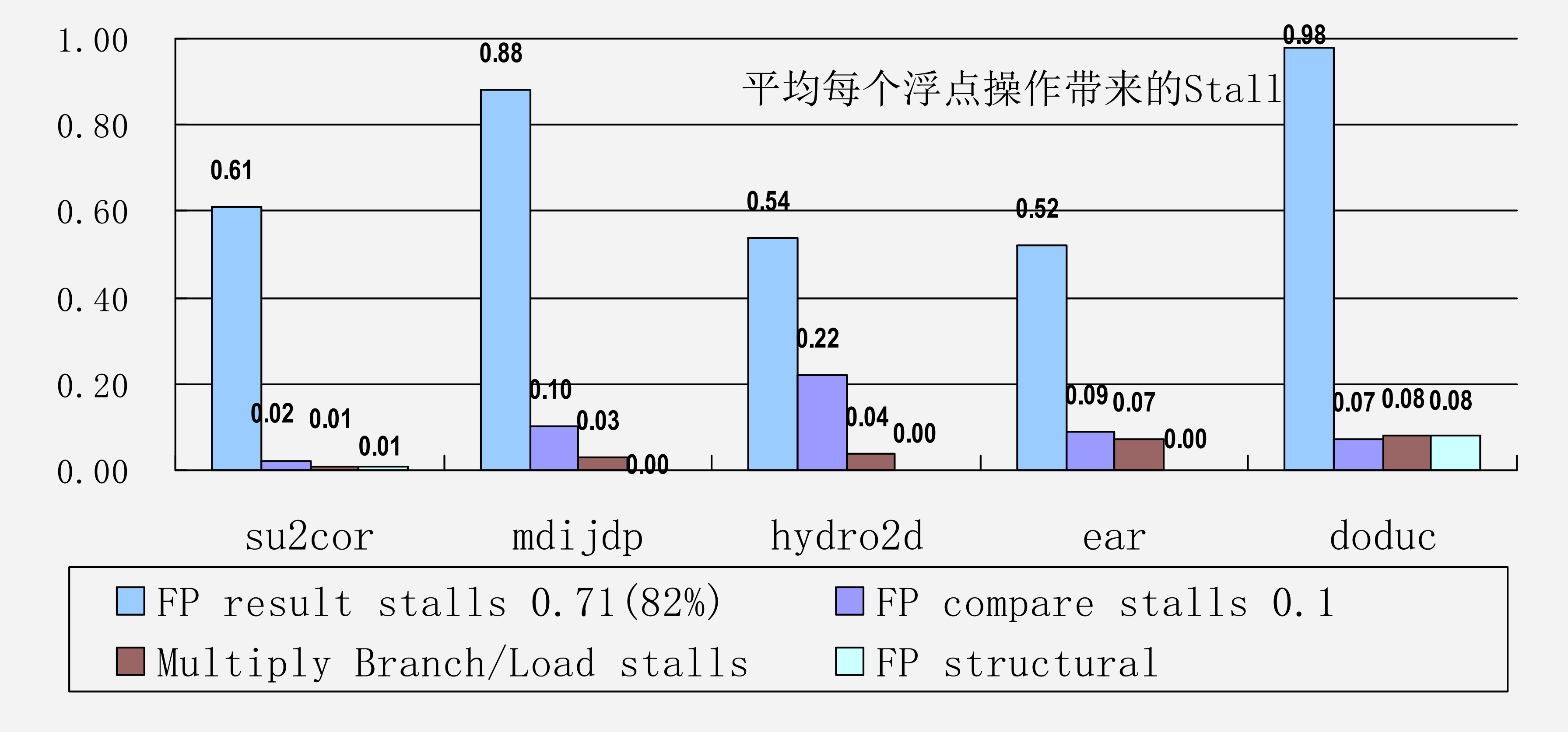
Checks are required in ID

- Check for **structural hazards**
  - The divide unit and Register write port.
- Check for **RAW hazards**
  - The CPU simply stalls the instruction at ID stage until:
    - Its **source registers are no longer listed as destinations** in any of the execution pipeline registers (registers between stages of M and A) OR
    - Its **source registers are no longer listed as the destination of a load** in the EX/MEM register.
- Check for **WAW hazards**
  - Check instructions in A1, ..., A4, Divide, or M1, ...,M7 for the same destination register (check pipeline registers.)
  - Stall instruction in ID if necessary.

Performance of MIPS FP pipeline



Performance of MIPS FP pipeline



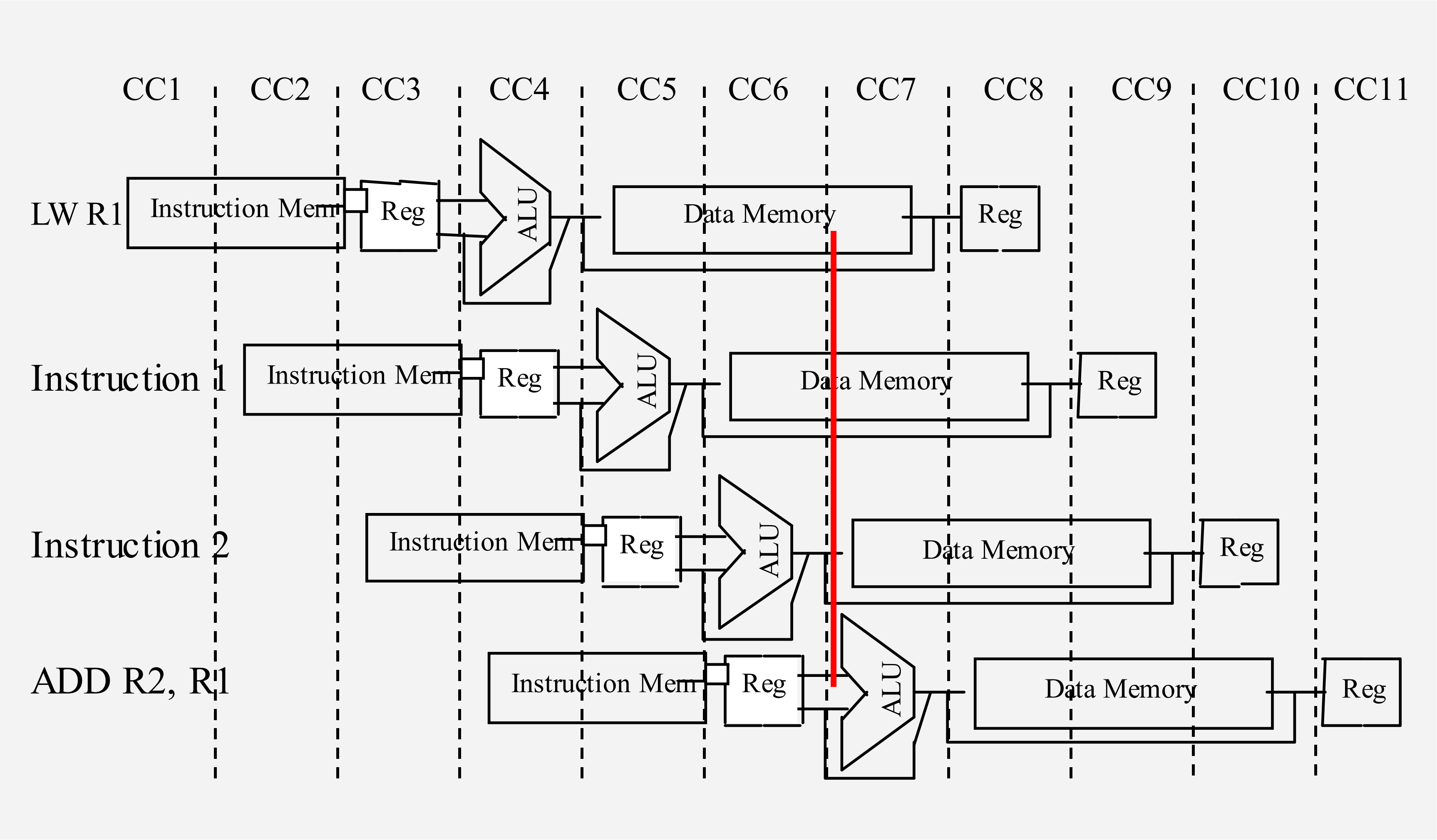
The MIPS R4000 pipeline

- IF—First half of instruction fetch. PC selection occurs. Cache access is initiated.
- IS—Second half of instruction fetch.
  - —This allows the cache access to take two cycles.
- RF—Decode and register fetch, hazard checking, I-cache hit detection.
- EX—Execution: address calculation, ALU Ops, branch target calculation and condition evaluation.
- DF/DS/TC
  - — Data fetched from cache in the first two cycles.
  - — The third cycle involves checking a tag check to determine if the cache access was a hit.
- WB—Write back result for loads and R-R operations.

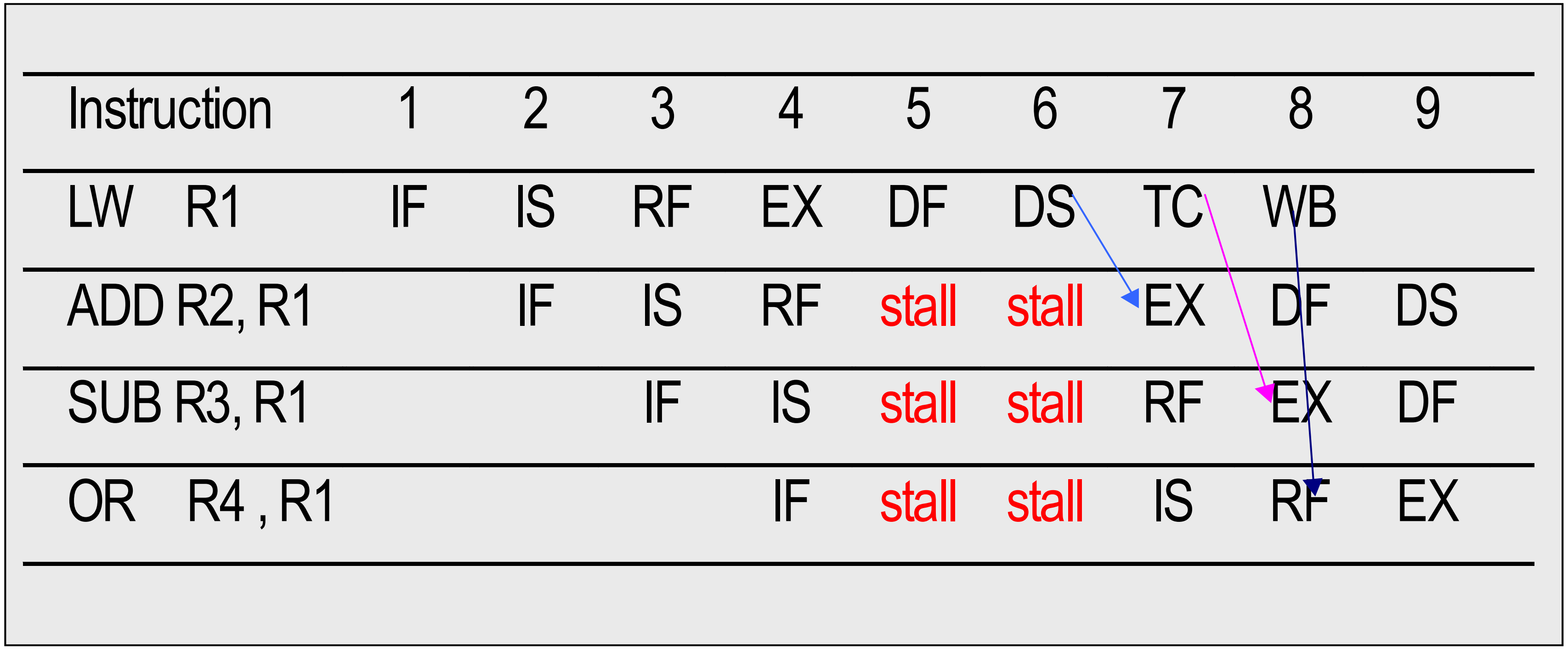
Possible stalls and delays

- Load delay: two cycles
  - The delay might seem to be three cycles, since the tag isn't checked until the end of the TC cycle.
  - However, if TC indicates a miss, the data must be fetched from main memory and the pipeline is backed up to get the real value.

Load stalls


CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
CC9
CC10
CC11
LW R1
Instruction Mem
Reg
ALU
Data Memory
Reg
Instruction 1
Instruction Mem
Reg
ALU
Data Memory
Reg
Instruction 2
Instruction Mem
Reg
ALU
Data Memory
Reg
ADD R2, R1
Instruction Mem
Reg
ALU
Data Memory
Reg

Example：load stalls

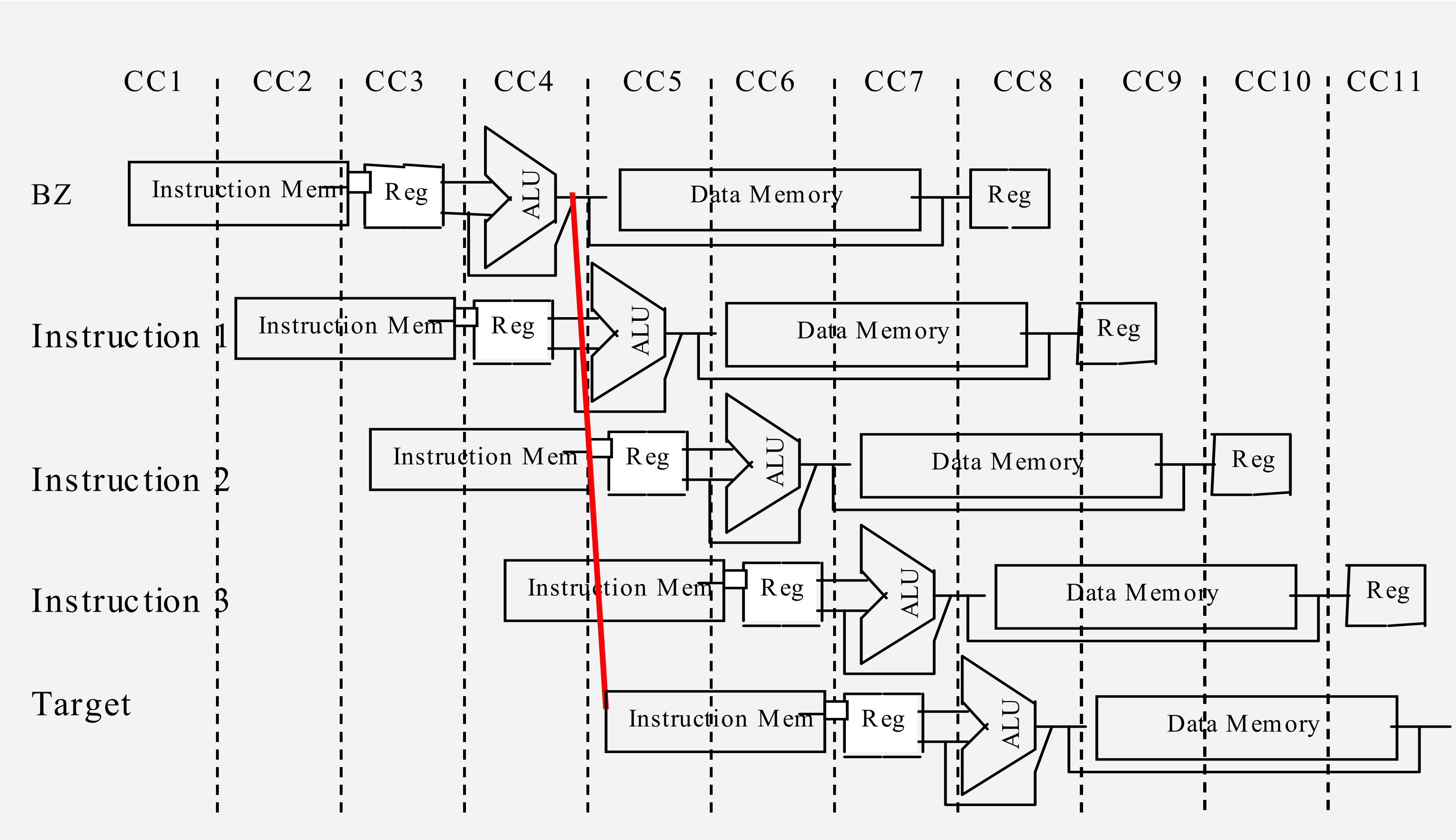


Instruction	1	2	3	4	5	6	7	8	9
LW R1	IF	IS	RF	EX	DF	DS	TC	WB	
ADD R2, R1		IF	IS	RF	stall	stall	EX	DF	DS
SUB R3, R1			IF	IS	stall	stall	RF	EX	DF
OR R4 , R1				IF	stall	stall	IS	RF	EX

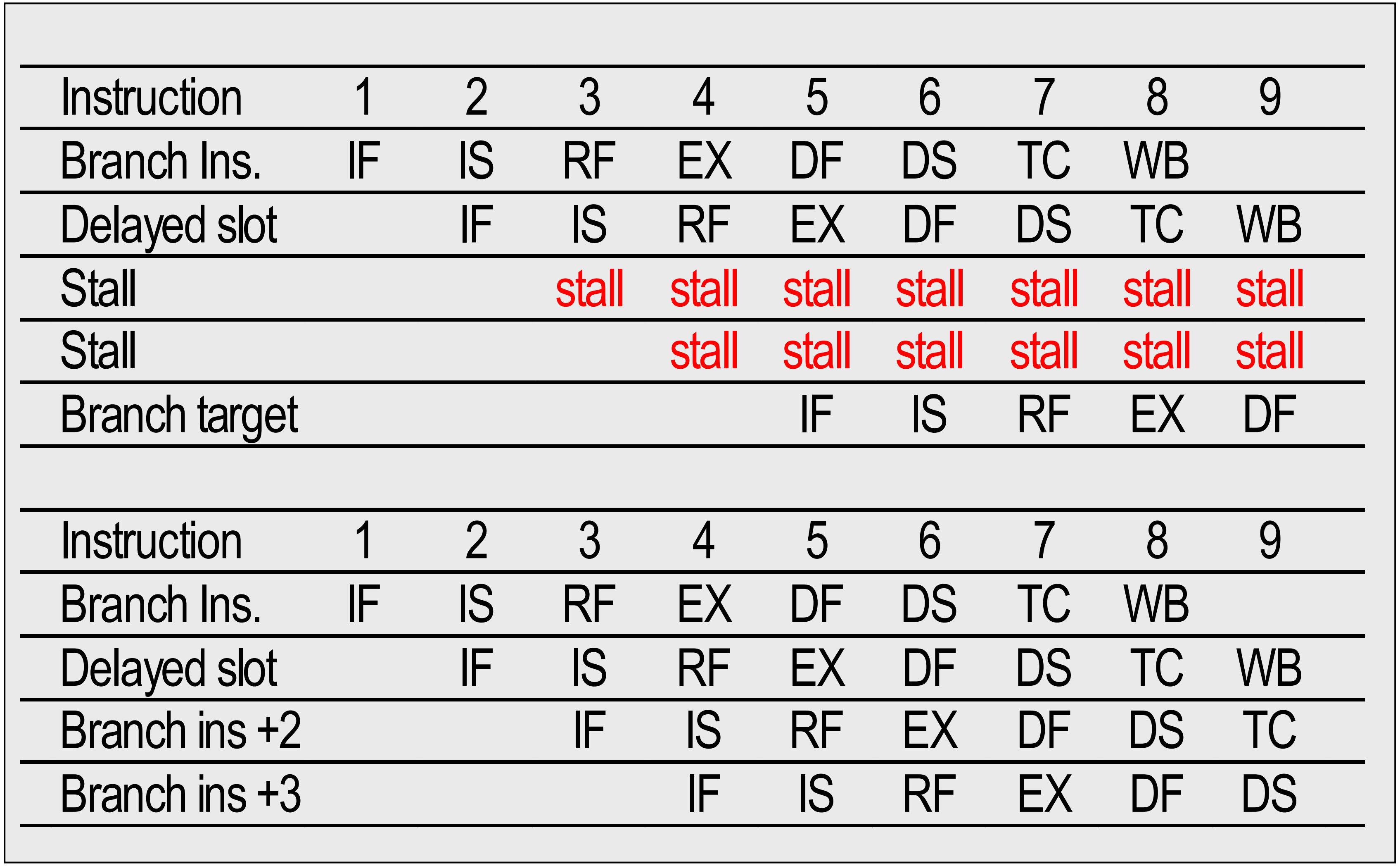
Branch delay: three cycles

- Branch delay: three cycles (including one branch delay slot)
  - The branch is resolved during EX, giving a 3 cycle delay.
  - The first cycle may be a regular branch delay slot (instruction always executed) or a branch-likely slot (instruction cancelled if branch not taken).
  - MIPS uses a predict-not-taken method presumably because it requires the least hardware.

Branch Delays： 3 stalls



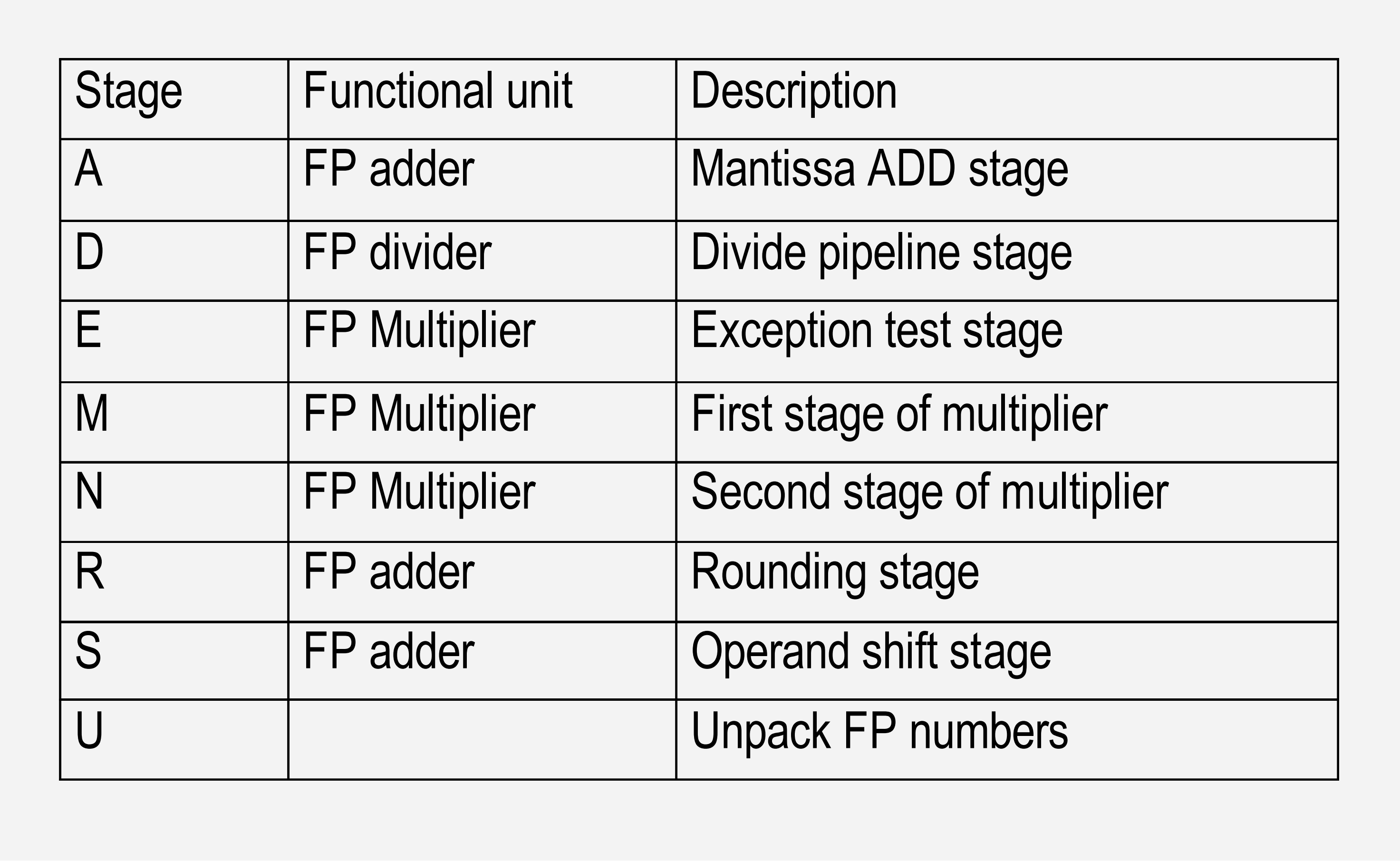
Pipeline status for branch latency



Instruction	1	2	3	4	5	6	7	8	9
Branch Ins.	IF	IS	RF	EX	DF	DS	TC	WB	
Delayed slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			stall	stall	stall	stall	stall	stall	stall
Stall				stall	stall	stall	stall	stall	stall
Branch target					IF	IS	RF	EX	DF

Instruction	1	2	3	4	5	6	7	8	9
Branch Ins.	IF	IS	RF	EX	DF	DS	TC	WB	
Delayed slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch ins +2			IF	IS	RF	EX	DF	DS	TC
Branch ins +3				IF	IS	RF	EX	DF	DS

The FP 8-stage operational pipeline



Stage	Functional unit	Description
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP Multiplier	Exception test stage
M	FP Multiplier	First stage of multiplier
N	FP Multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

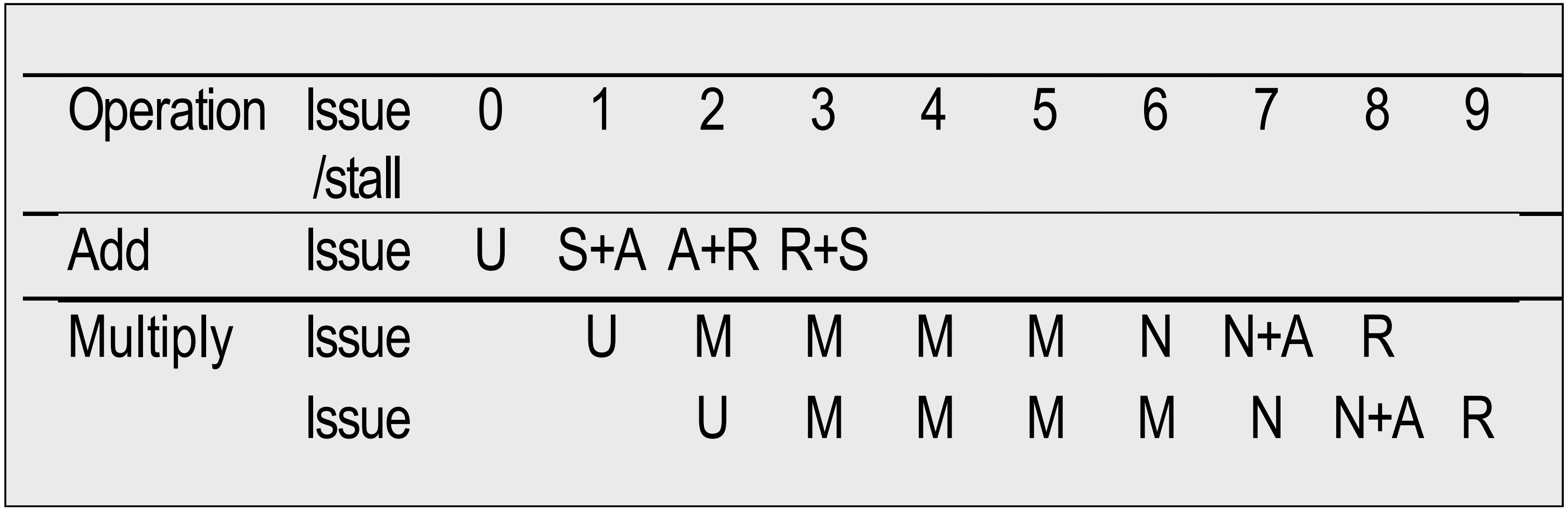
Latency and initiation intervals

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U,E+M,M,M,M,N,N+A,R
Divide	36	35	U,A,R,D^{27},D+A,D+R,D+A, D+R, A, R
Square root	112	111	U, E, $(A+R)^{108}$, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Structural hazards-1

Operation	Issue /stall	0	1	2	3	4	5	6	7	8	9
Multiply	Issue	U	M	M	M	M	N	N+A	R		
Add	Issue		U	S+A	A+R	R+S					
	Issue			U	S+A	A+R	R+S				
	Issue				U	S+A	A+R	R+S			
	Stall					U	S+A	A+R	R+S		
	Stall						U	S+A	A+R	R+S	
	Issue							U	S+A	A+R	R+S
	Issue								U	S+A	A+R

Structural hazards-2



Operation	Issue /stall	0	1	2	3	4	5	6	7	8	9
Add	Issue	U	S+A	A+R	R+S						
Multiply	Issue		U	M	M	M	M	N	N+A	R	
	Issue			U	M	M	M	M	N	N+A	R

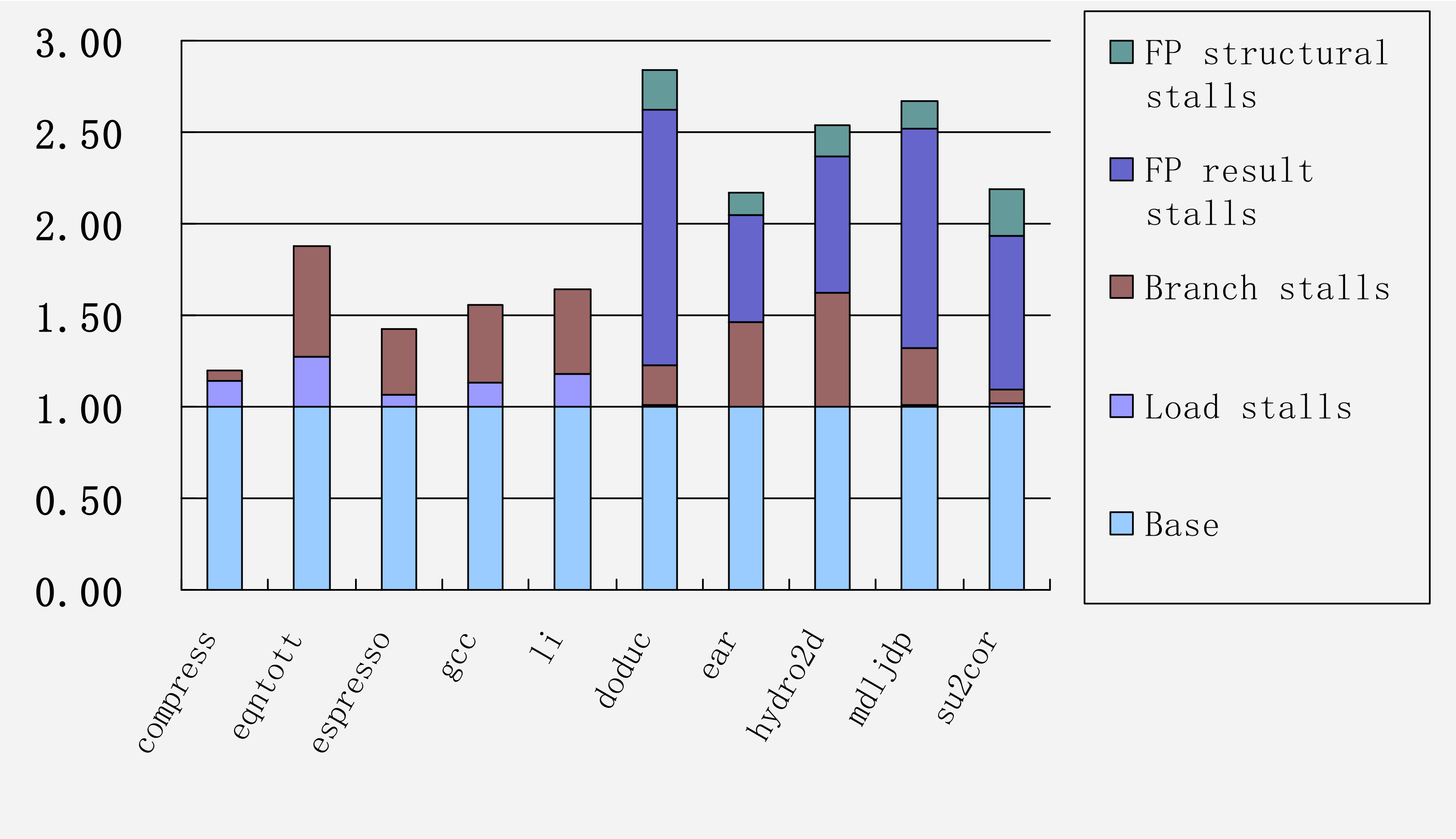
Structural hazards-3

Operation	Issue /stall	25	26	27	28	29	30	31	32	33	34	35
Divide	Issue in cycle 0	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R
Add	Issue		U	S+A	A+R	R+S						
	Issue			U	S+A	A+R	R+S					
	Stall				U	S+A	A+R	R+S				
	Stall					U	S+A	A+R	R+S			
	Stall						U	S+A	A+R	R+S		
	Stall							U	S+A	A+R	R+S	
	Stall								U	S+A	A+R	R+S
	Stall									U	S+A	A+R
	Issue										U	S+A
	Issue											U

Structural hazards-4

Operation	Issue	0	1	2	3	4	5	6	7	8	9
Add	Issue	U	S+A	A+R	R+S						
Divide	stall		U	A	R	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D

Performance loss measurements



THANK YOU