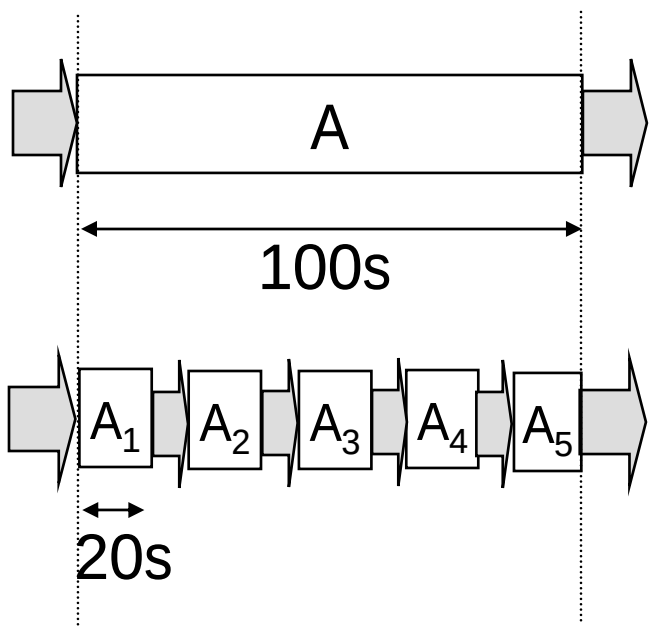
Pipelining

Outline

- Pipelining basics
- The Basic Pipeline for DLX & MIPS
- Pipeline hazards
  - Structural Hazards
  - Data Hazards
  - Control Hazards
- Handling exceptions
- Multi-cycle operations

Pipelining basics

- Basic idea: exploit concurrency of independent operations
  - Split one operation into independent sub-operations

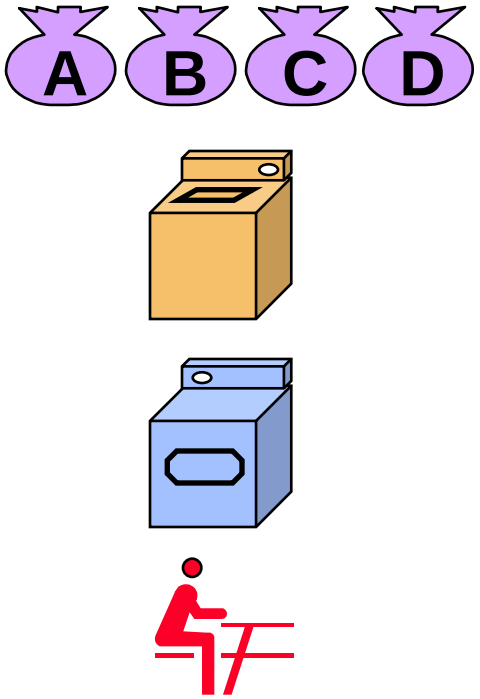


One computation is launched every 100 s

One computation is launched every 20 s

Pipelining: example

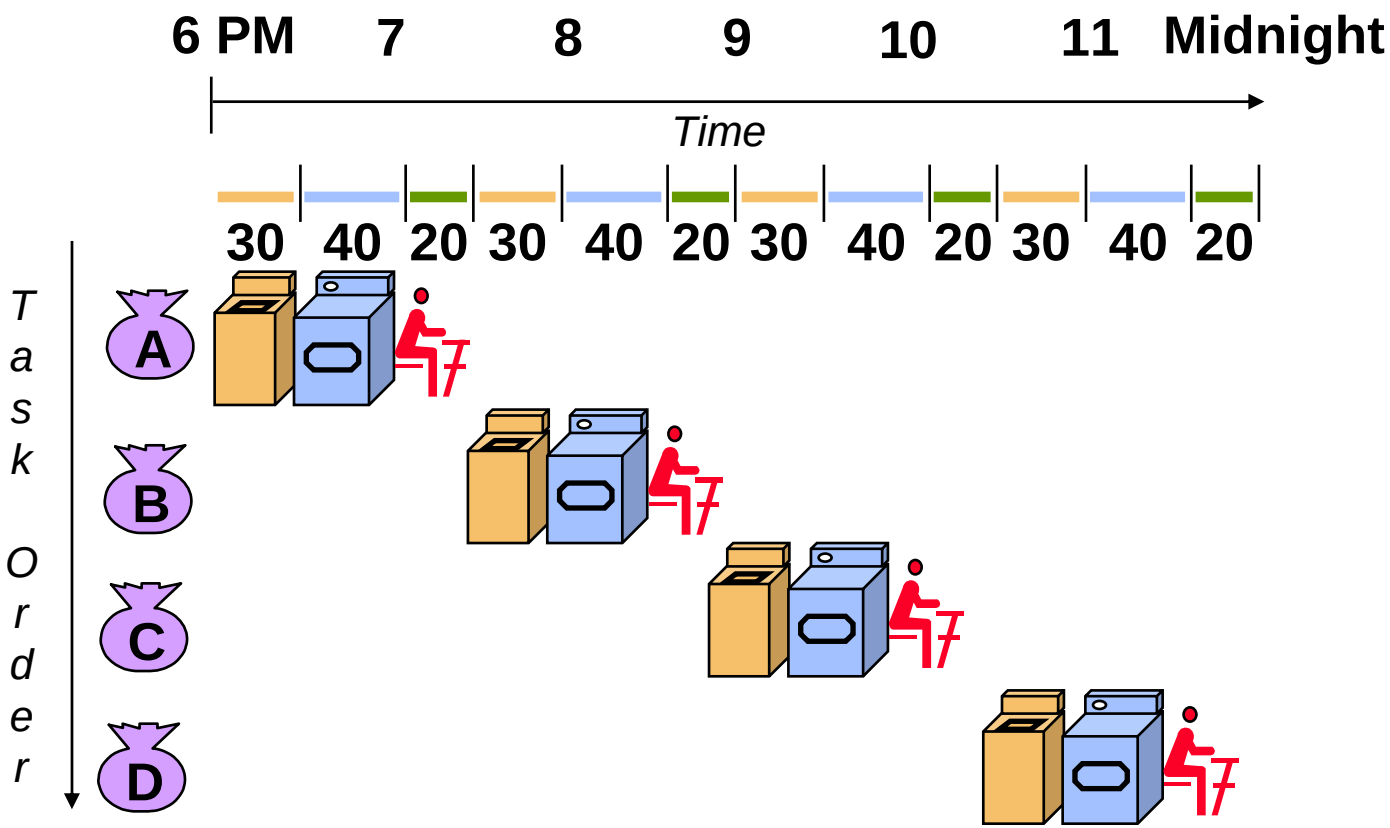
- Laundry Example
  - A, B, C, D each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes



1 operation = wash+dry+fold = 90 min.

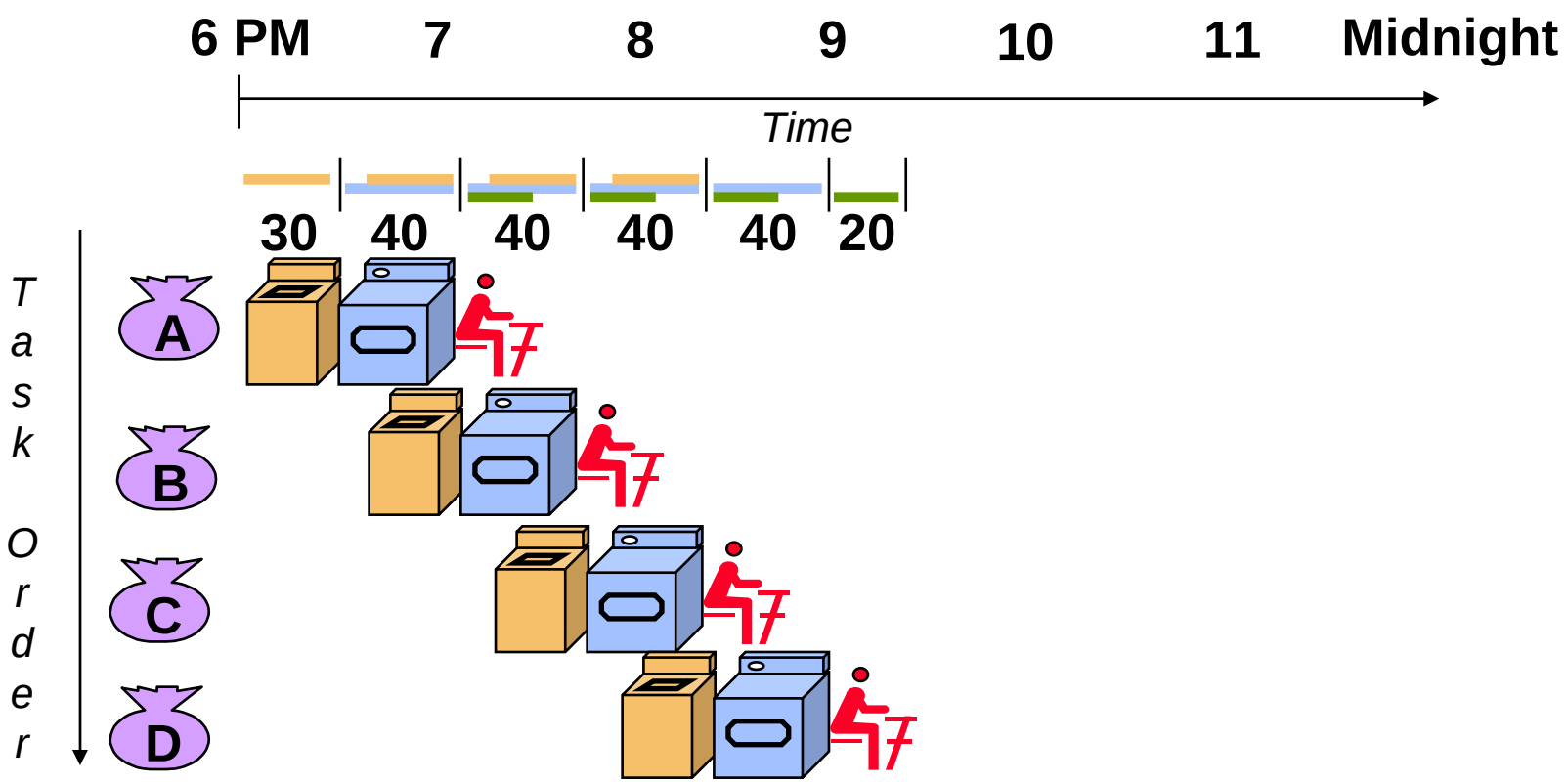
Pipelining: example (2)

- Sequential laundry takes 6 hours for 4 loads



Pipelining: example (3)

- Pipelined laundry takes 3.5 hours for 4 loads

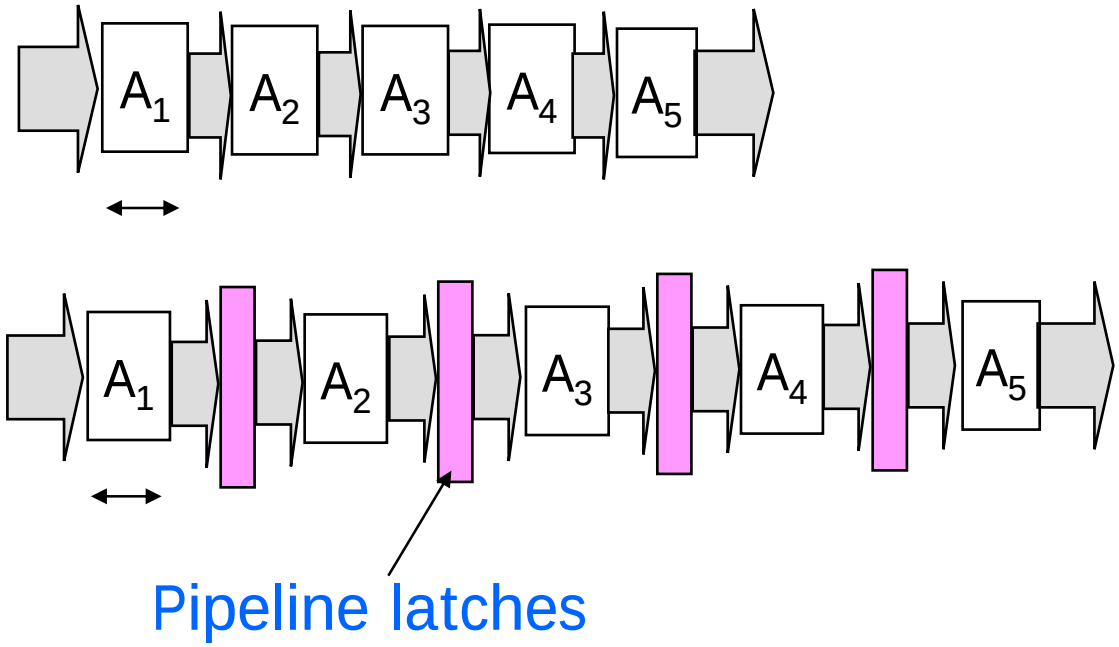


Pipelining Lessons

- Pipelining **doesn't help latency** of single task
  - It helps throughput of entire workload =>
    CPI is decreased !
- Pipeline rate **limited by slowest pipeline stage**
- Multiple tasks operating simultaneously
- Potential speedup = Number of pipe stages
  - Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

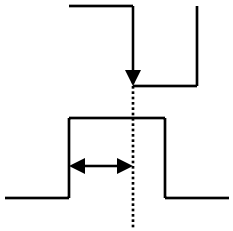
Applying pipelining to hardware

- Implementation of pipelining requires a way to store intermediate results
  - In hardware, the output of each stage must be stored using latches (flip-flops)



Applying pipelining to hardware (2)

- What prevents us from just doing too many pipe stages?
  - Some computations just won't divide into any shorter logical implementations
  - Ultimately, it comes down to circuit design issues
    - Latches have delays!!!
      - Time for a signal to be stable before clock edge
      - Time for a signal to be stable after clock edge



- In practice:
  - Modern pipelines: **10-20 stages** (e.g. Pentium4, Xeon)
  - More than 20 not beneficial!

Pipeline performance

- T_{mono} = clock period of non-pipelined computation
- τ_{mono} = exec time of overall (non-pipelined) computation
- τ_p = exec time of overall (pipelined) computation
- τ_i = exec time of i-th pipeline stage
- τ_l = latency of latches
- k = # of pipeline stages
- T_p = clock period of pipelined computation

$$T_p = max_{i=1,\ldots,k} \{\tau_i\} + \tau_l \qquad \tau_p = k\, T_p$$

- **μ = execution rate (# of instructions for time unit)**

$$\mu_{mono} = 1/T_{mono} \qquad \mu_p = 1/T_p$$

T_{mono}, T_p = average instruction execution times

Pipeline performance (2)

- **Average execution rate**:
  - To complete n instructions starting from an empty pipe [k+(n-1)] clock cycles are needed

$$\overline{\mu_p} = \frac{n}{kT_p + (n-1)T_p}$$

For $n \to \infty$, $\mu_p \to 1/T_p$

- **Efficiency** (utilization): % of time in which the CPU is busy
  - = ratio of average rate and ideal rate

$$\eta = \frac{\overline{\mu_p}}{\mu_p} = \frac{\frac{n}{kT_p+(n-1)T_p}}{\frac{1}{T_p}} = \frac{n}{k+(n-1)}$$

For $n \to \infty$, $\eta \to 1$

Pipeline performance (3)

- **Speedup**:
  - Ration between the speed of pipelined and non-pipelined

$$\alpha = \mu_p / \mu_{mono} = T_{mono} / T_p$$

- Example:
  - 5 stages (50ns, 50ns, 60ns, 50ns, 50ns)
  - Latch delay = 5ns
    - $T_{mono} = \tau_{mono} = 50+50+60+50+50 = 260ns$
    - $T_p = 60 + 5 = 65ns$ $\tau_p = 5*65ns = 325\ ns > \tau_{mono}$
    - $\mu_{mono} = 1/T_{mono}$ $\mu_p = 1/T_p$
    - **α = 260/65 = 4.0 speedup**

Two views of pipelining

- W.r.t. single-cycle implementation
  - Reduces T_{cycle}
    - ~ 1/k
  - Improves average instruction execution time
- W.r.t. multi-cycle implementation
  - Reduces CPI
    - ~ 1/k
  - Improves average instruction execution time

The MIPS pipeline

MIPS pipeline stages

- Pipelining execution = split execution into stages
- What and how many stages?
  - **Stage 1: Instruction Fetch (IF)**
  - **Stage 2: Instruction Decode (ID)**
  - **Stage 3: Execute (EX)**
  - **Stage 4: Memory Access (ME)**
  - **Stage 5: Write Back (to register file) (WB)**

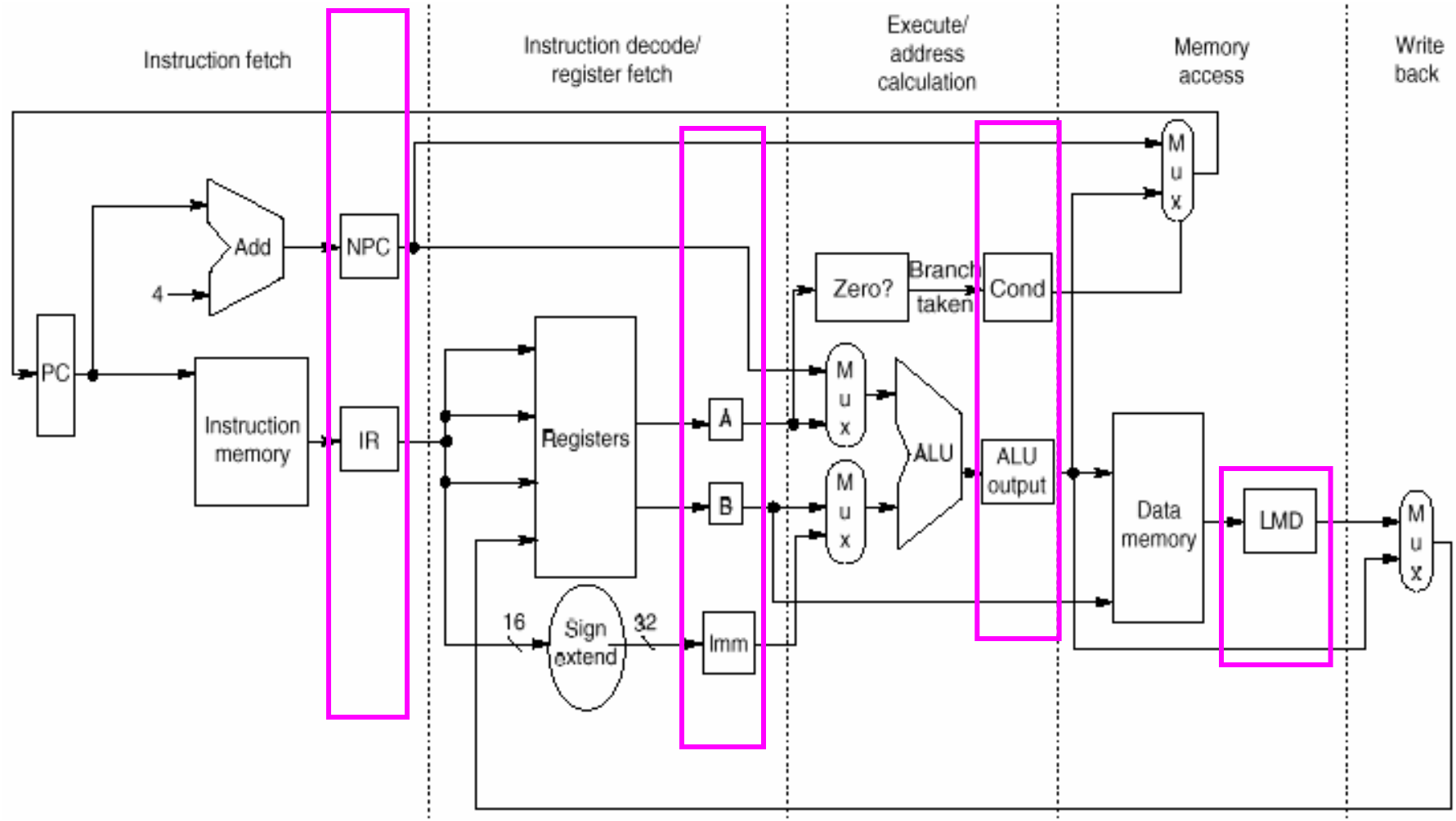
MIPS ISA summary

- 32 registers
  - $0,...,$31
- 2^{30} flat memory addressing
- 3 instruction formats
  - Fixed size = 32 bit

Name	Fields						Comments
FieldSize	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op 0-5	rs 6-10	rt 11-15	rd 16-20	shmt 20-24	funct 25-31	Arithmetic instruction format
I-format	op	rs	rt	address/immediate 16-31			Transfer (load/store), branch immediate format
J-format	op	target address 6-31					Jump instruction format

  - Will see FP instructions later

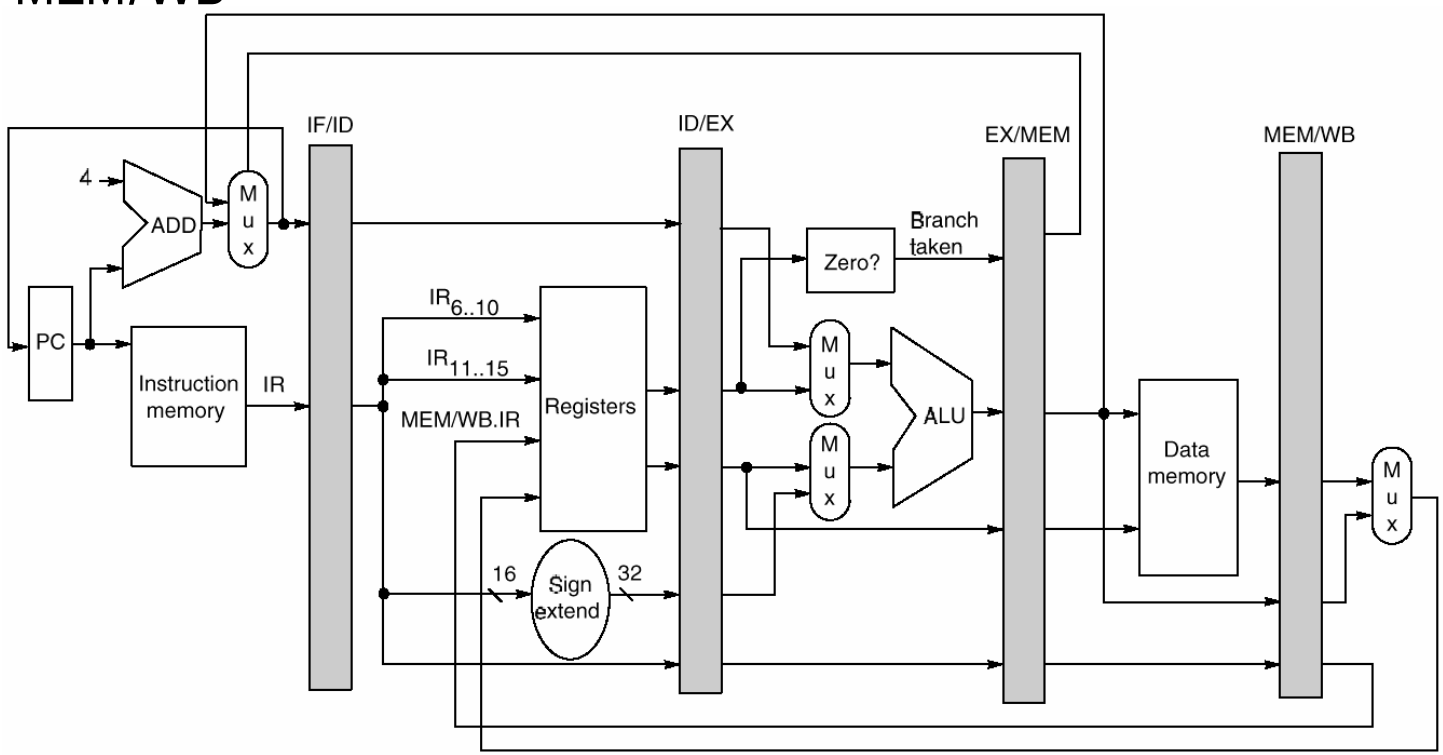
MIPS without pipelining



Control logic not shown!

The Basic Pipeline For MIPS

- Latch names use boundary unit names
  - IF/ID
  - ID/EX
  - EX/MEM
  - MEM/WB

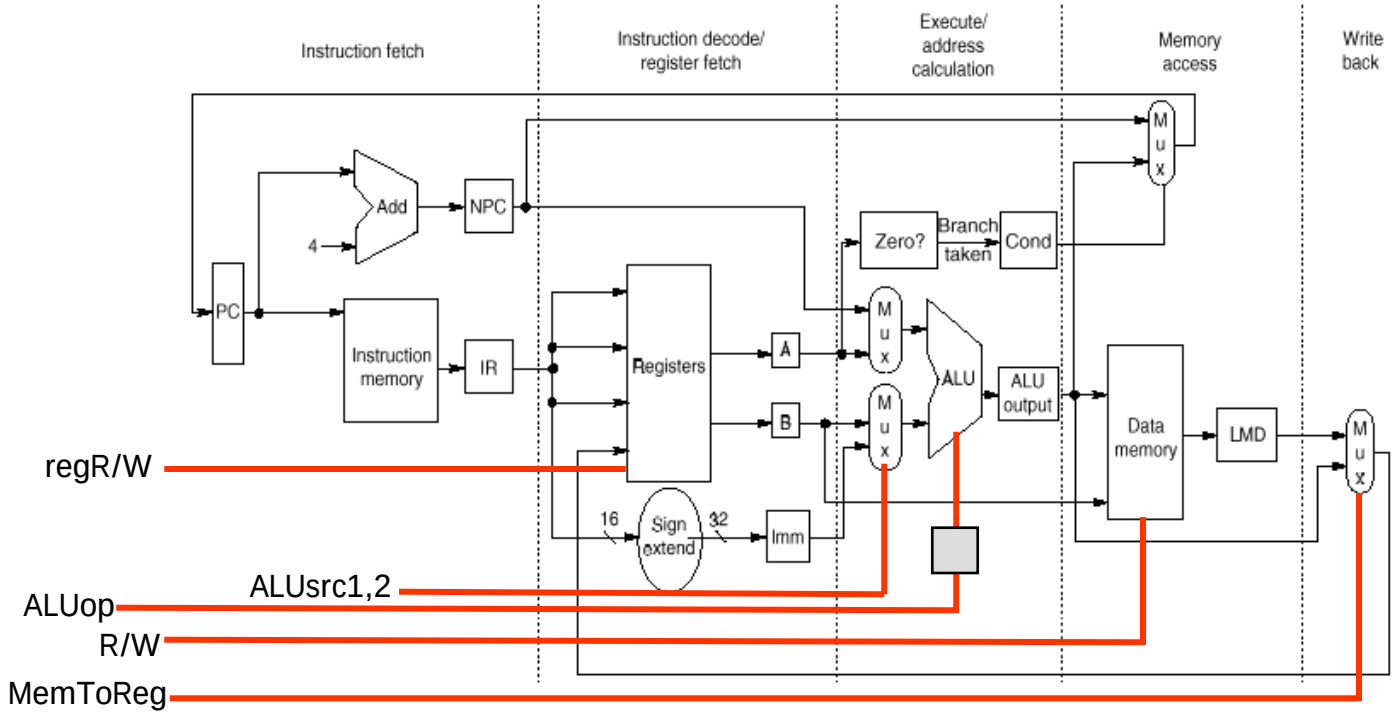


Pipeline features

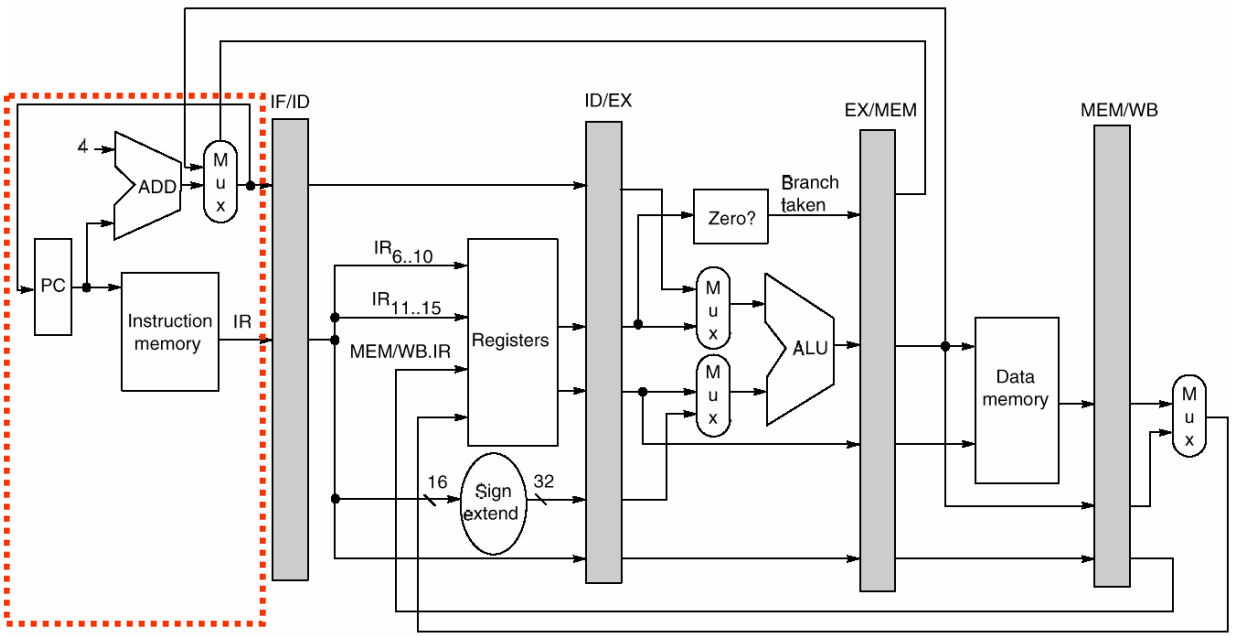
- Execution is based on separate data and instruction memory
  - Typically implemented as separate I- and D- caches
- Register file is used in ID and in WB
  - What if a read and write are to the same register?
- PC assignment done in IF
  - But branches may modify it later...

Pipeline analysis (2)

- Control overhead
  - Need extra logic to control execution
  - Limited to the proper assignment of the various multiplexers and control signals



MIPS pipeline functions (1)

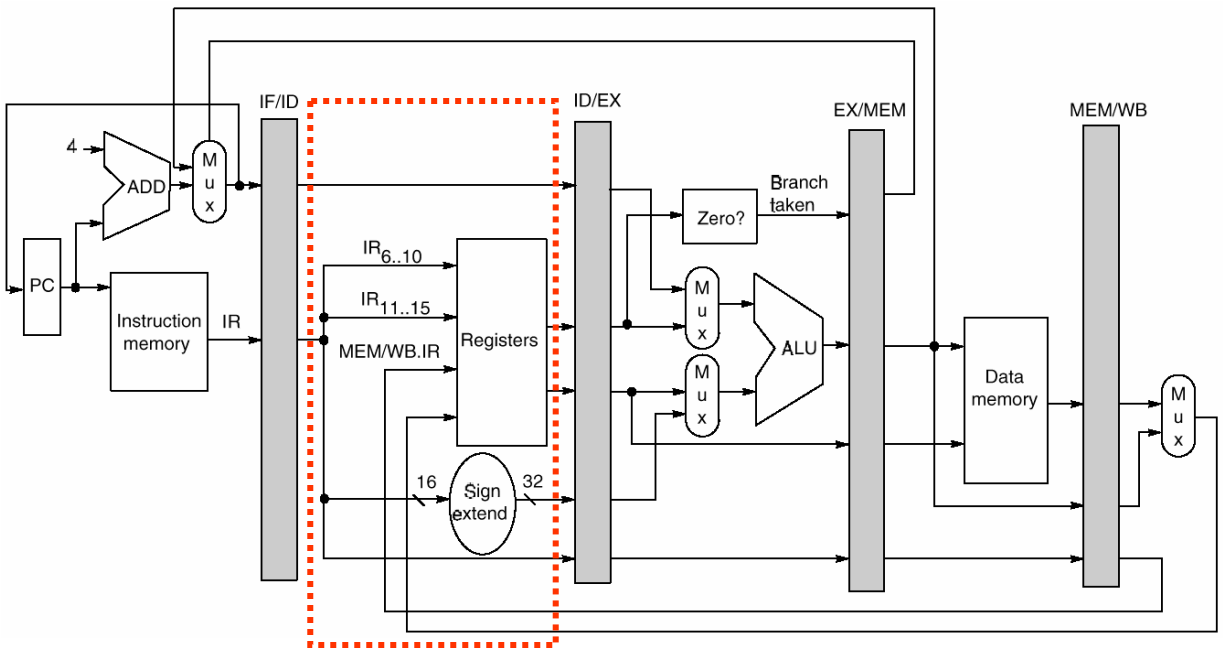


Passed To Next Stage

```
IR <- Mem[PC]
NPC <- PC + 4
```

- **Instruction Fetch** (IF):
  - Send out the PC and fetch the instruction from memory into the instruction register (IR)
  - Increment the PC by 4 to address the next sequential instruction.
- IR holds the instruction that will be used in the next stage.
- NPC holds the value of the next PC.

MIPS pipeline functions (2)

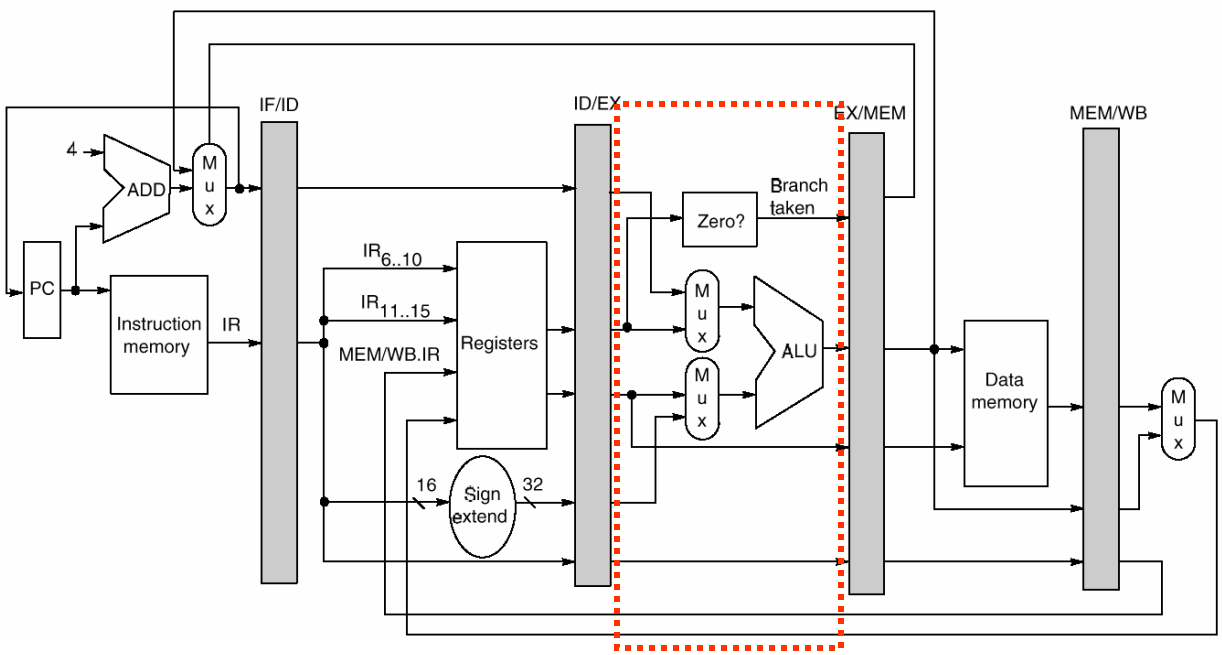


Passed To Next Stage

```
A <- Regs[IR6…10];
B <- Regs[IR11…15];
Imm <-((IR16)## IR16-31)
```

- **Instruction Decode**/Register Fetch Cycle (ID):
  - Decode instruction and access the register file to read the registers.
  - The outputs of the general purpose registers are read into two temporary registers (A & B) for use in later clock cycles.
  - We extend the sign of the lower 16 bits of the Instruction Register

MIPS pipeline functions (3)

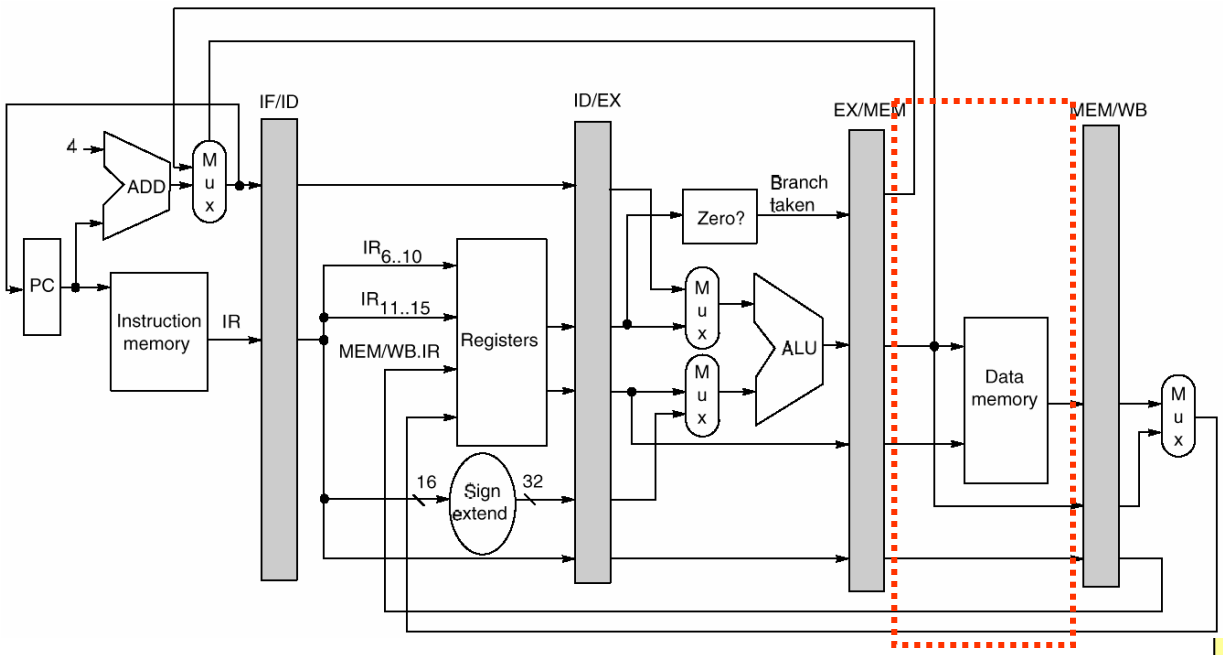


Passed To Next Stage

- Memory reference
  - `ALUOutput <= A + Imm`
- ALU op (reg-reg)
  - `ALUOutput <= A op B`
- ALU op (reg-imm)
  - `ALUOutput <= A op Imm`
- Branch
  - `ALUOutput <= PC +Imm`
  - `Cond <= A op 0`

- **Execute** Address Calculation (EX):
  - Perform an operation (for an ALU) or an address calculation (if a load or a Branch).
    - If an ALU, actually do the operation
    - If an address calculation, figure out how to obtain the address and stash away the location of that address for the next cycle

MIPS pipeline functions (4)



Passed To Next Stage

- Memory reference

```
LMD<= Mem(ALUOutput)
```

or

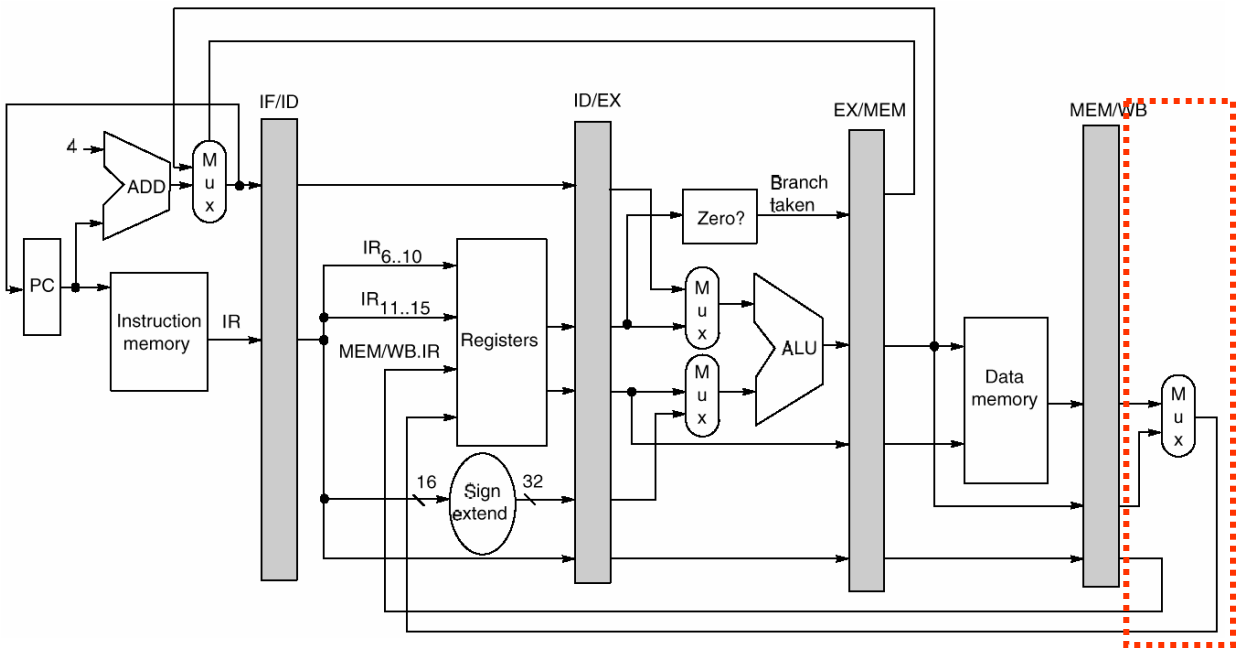
```
Mem(ALUOutput)<= B
```

- Branch

```
if (cond)
   PC <= ALUOutput
else
   PC <= NPC
```

- **Memory** access (MEM):
  - If this is an ALU op, do nothing.
  - If a load or store, then access memory.

MIPS pipeline functions (5)



Passed To Next Stage

- ALU op (reg-reg)

  Regs(IR$_{16…20}$) <= ALUOutput

- ALU op (reg-imm)

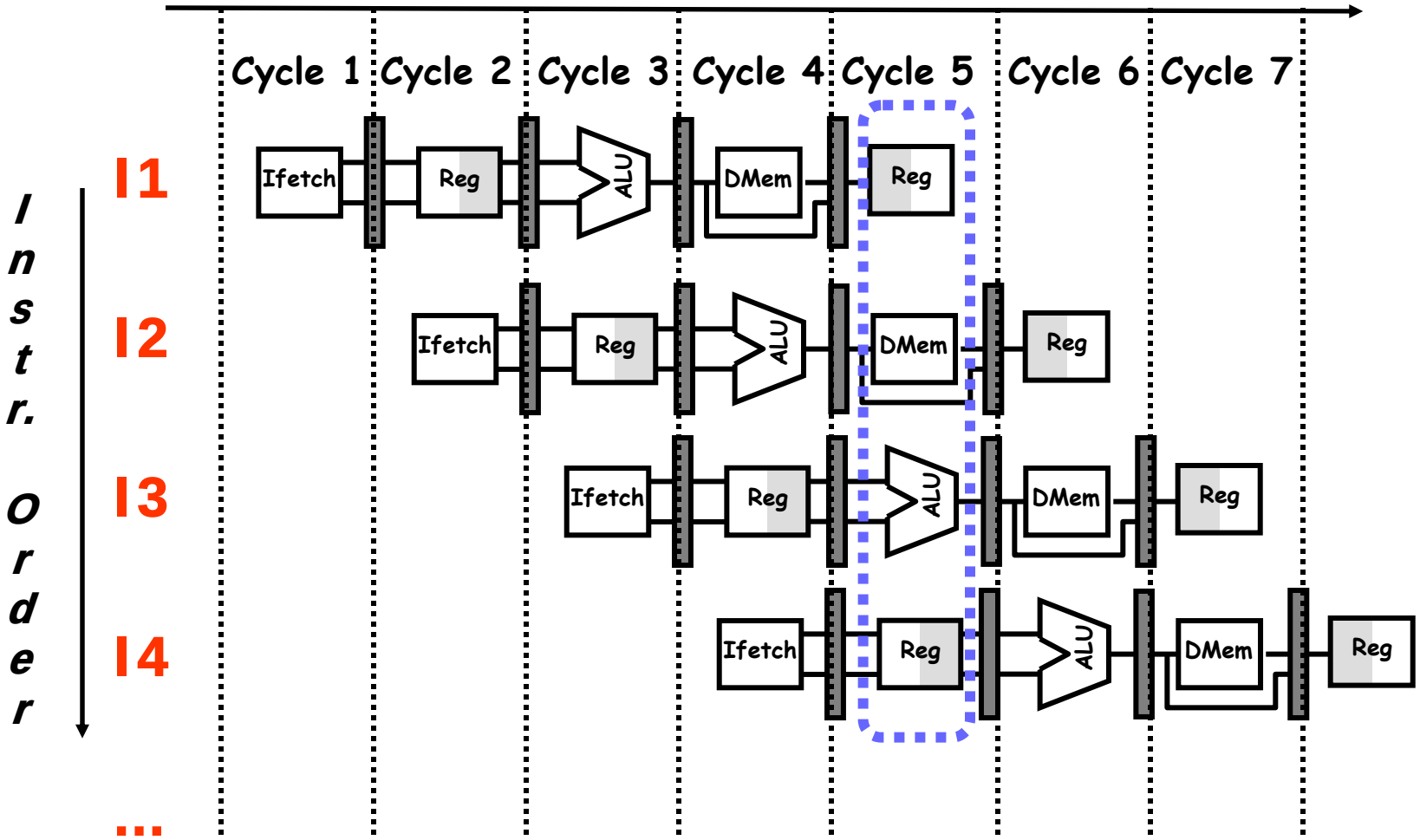
  Regs(IR$_{11…15}$) <= ALUOutput

- Load

  Regs(IR$_{11…15}$) <= ALUOutput

- **Write back** (WB):
  - Update the registers from either the ALU or from the data loaded.

The Basic Pipeline For MIPS (2)



Pipeline stages and functions

- Summary:

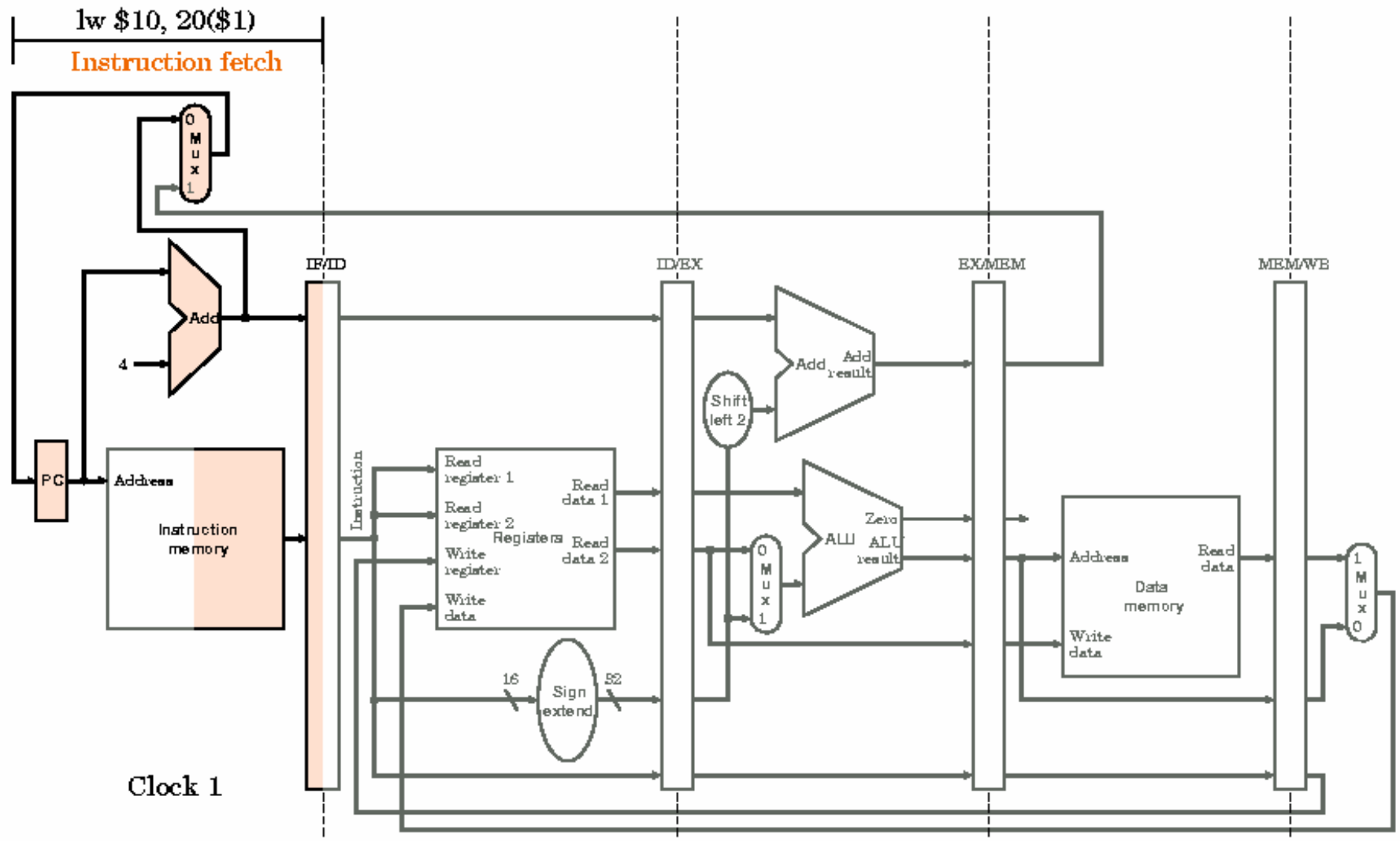
	Reg-Reg ALU	*Reg-immed ALU*	*Load*	*Store*	*Branch*	*Jump*
IF	IR_{ID}=IMem[PC_{IF}]; PC_{ID}=PC_{IF}=PC_{IF}+4;					
ID	A_{EX}=Regs[IR_{ID}[rs]]; B_{EX}=Regs[IR_{ID}[rt]]; IR_{EX}=IR_{ID}; PC_{EX}=PC_{ID}; IM_{EX}=IR_{ID} [15]^{16}##IR_{ID} [14..0];					
EX	ALU_M= A_{EX}op B_{EX}; IR_M=IR_{EX}; PC_M=PC_{EX};	ALU_M= A_{EX} op IM_{EX}; IR_M=IR_{EX}; PC_M=PC_{EX};	ALU_M =A_{EX} + IM_{EX}; IR_M=IR_{EX}; PC_M=PC_{EX}; MD_M=B_{EX};		ALU_M =PC_{EX}+IM_{EX}; CO_M= A_{EX} op 0; IR_M=IR_{EX}; PC_M=PC_{EX};	ALU_M = PC_{EX}+IM_{EX}; IR_M=IR_{EX}; PC_M=PC_{EX};
MEM	IR_{WB}=IR_M; PC_{WB}=PC_M;	IR_{WB}=IR_M; PC_{WB}=PC_M;	WB_{WB} = DMem[ALU_M] ;	DMem[ALU_M] = MD_M;	IR_{WB}=IR_M; PC_{WB}=PC_M; if (CO_M) PC_{IF}=ALU_M;	IR_{WB}=IR_M; PC_{WB}=PC_M; PC_{IF}=ALU_M;
WB	Regs[IR_{WB}[rd]] = WB_{WB};	Regs[IR_{WB}[rt]] = WB_{WB};	Regs[IR_{WB}[rt]] = WB_{WB};			

MIPS pipeline: Example (1)

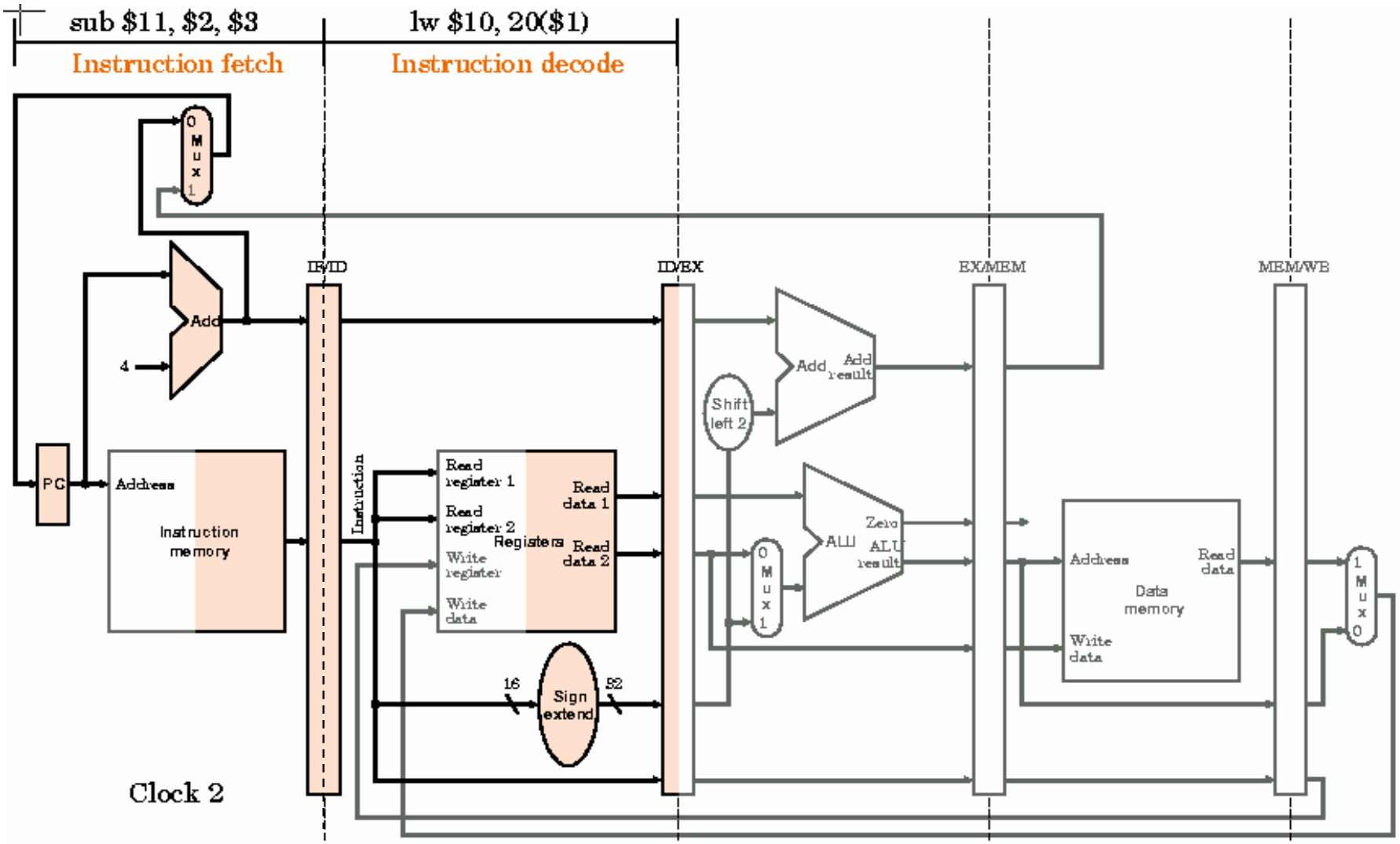
- Operation sequence:

```
lw      $10, 20($1)   // mem[$1+20] -> $10
sub     $11, $2, $3
```

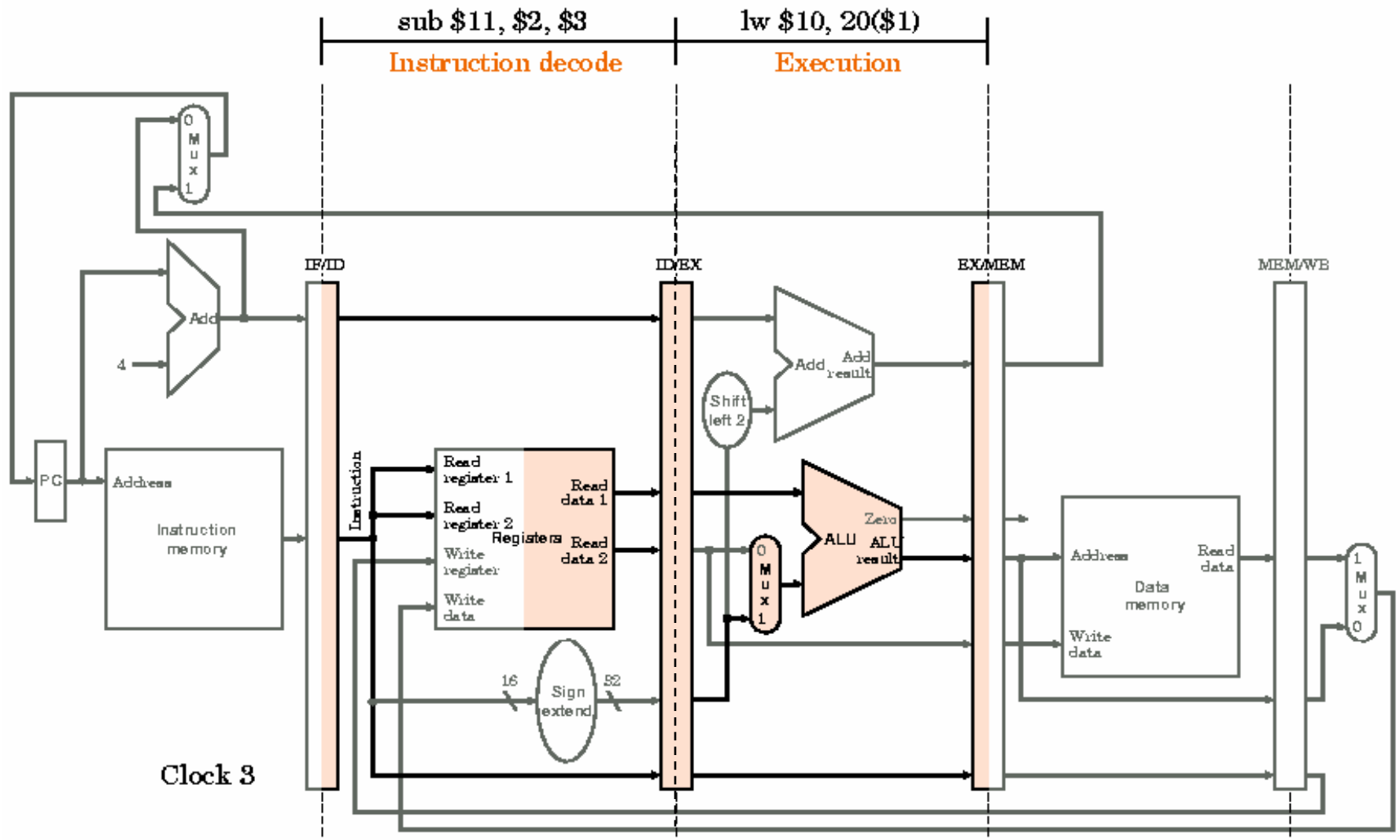
MIPS pipeline: Example (2)



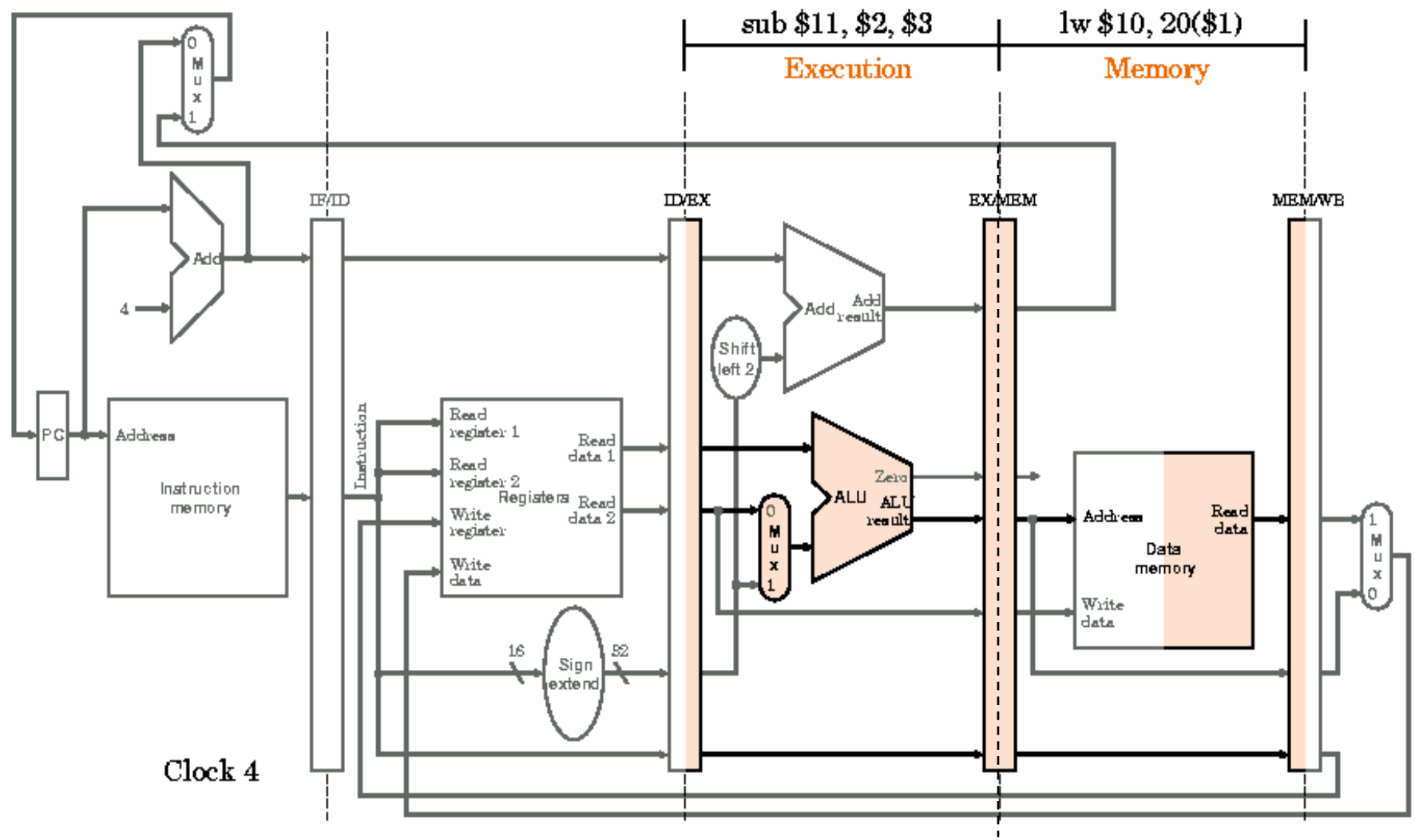
MIPS pipeline: Example (3)



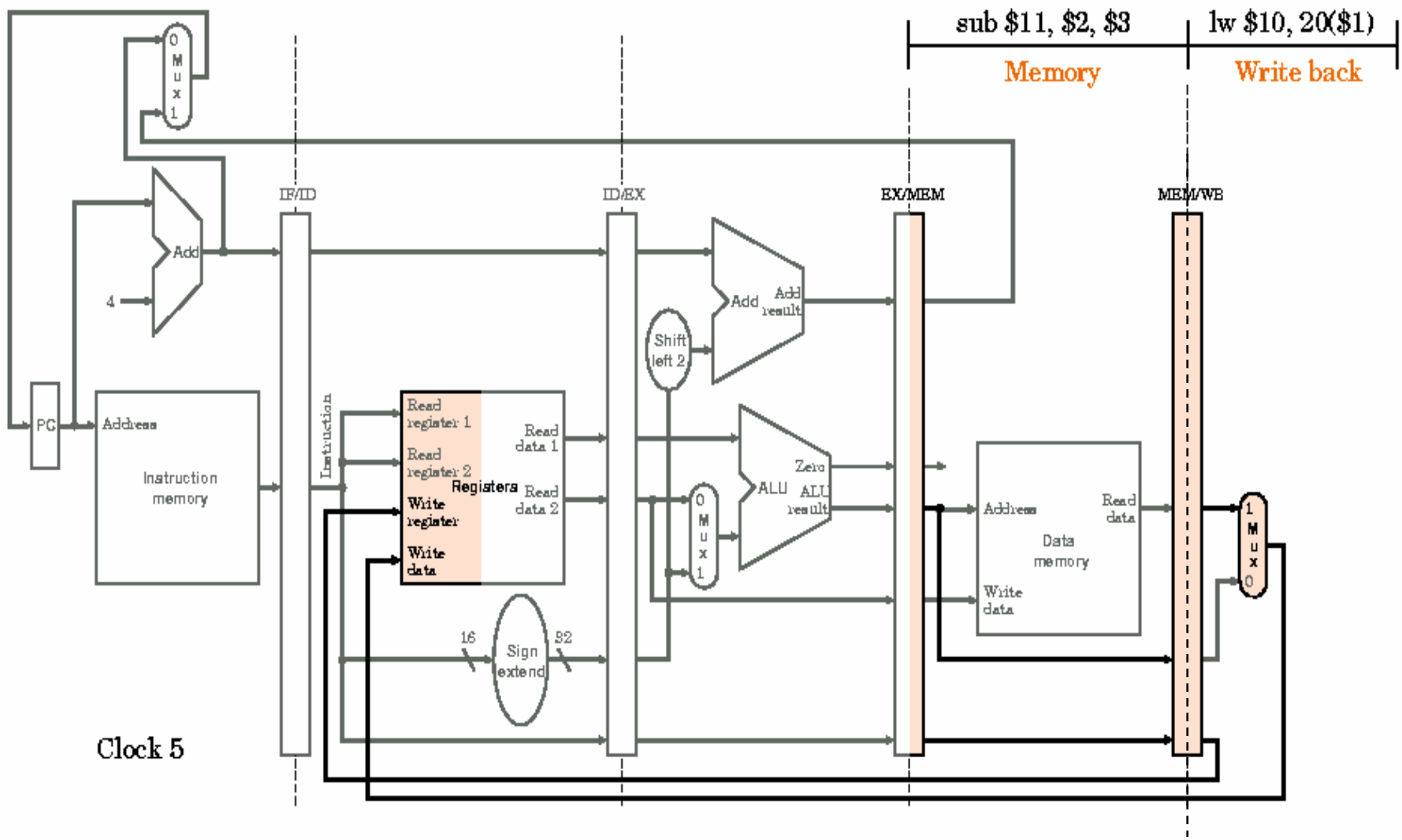
MIPS pipeline: Example (4)



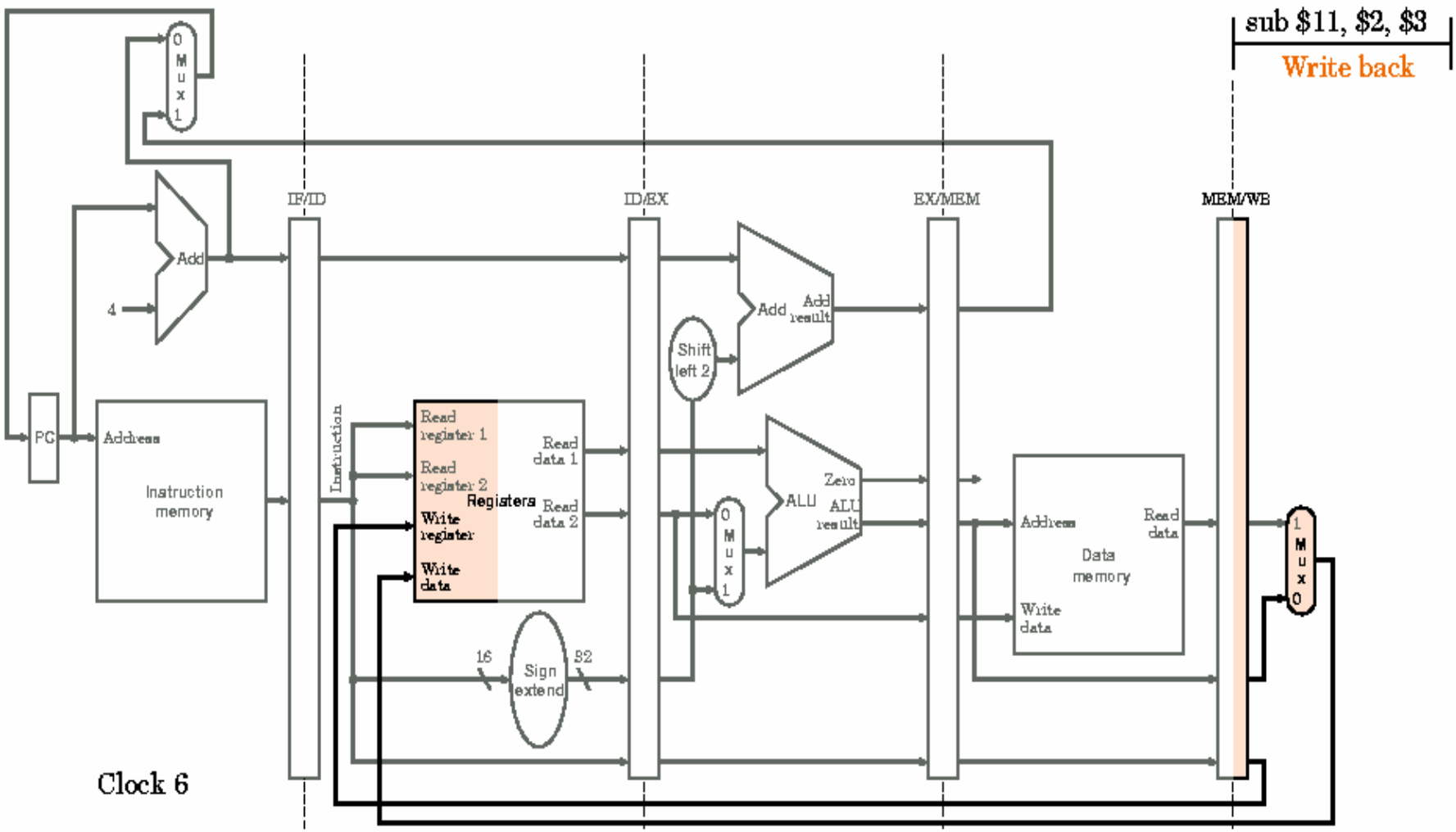
MIPS pipeline: Example (5)



MIPS pipeline: Example (6)



MIPS pipeline: Example (7)



Pipeline hazards

Pipeline Hazards

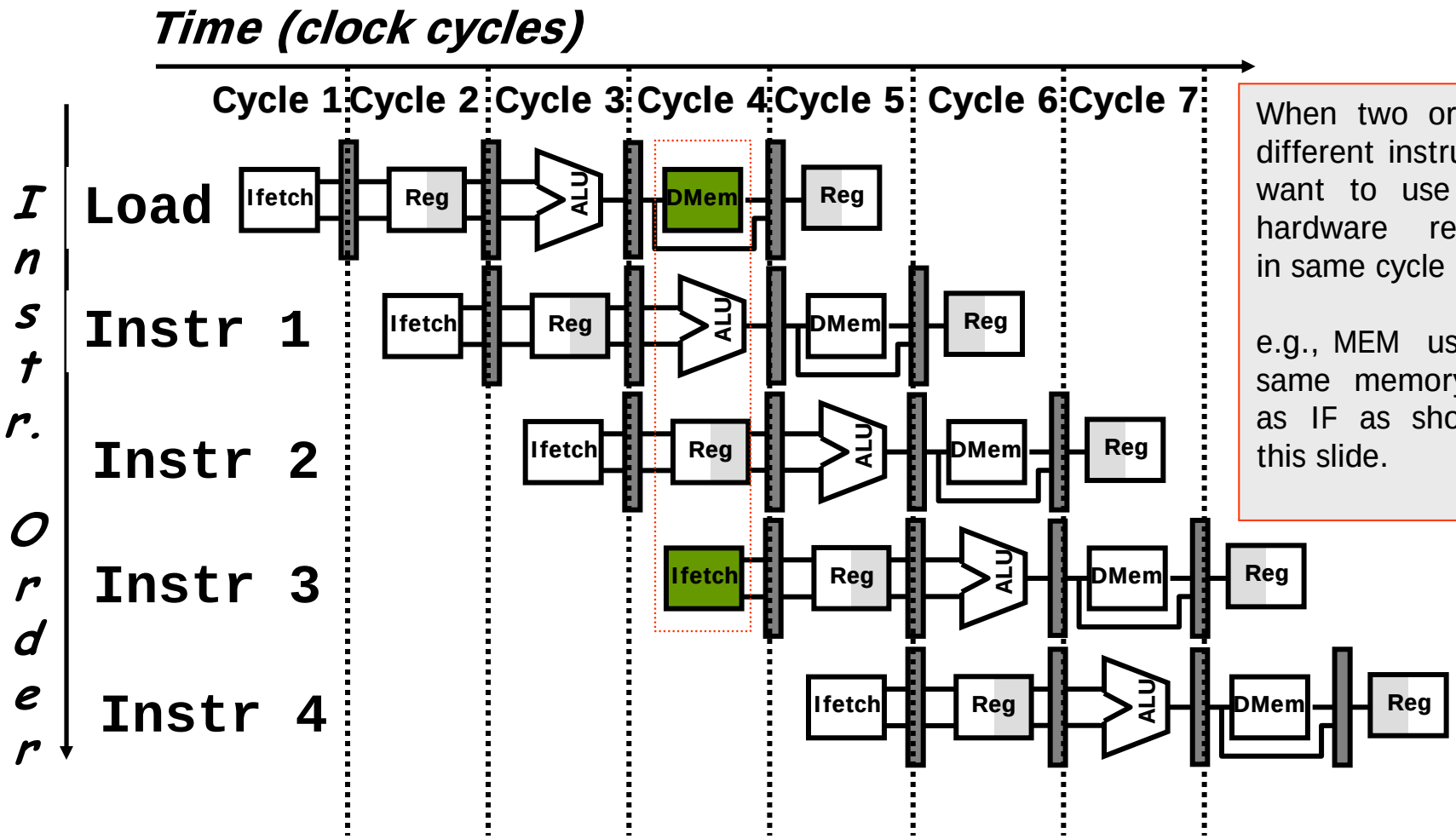
- Hazards: conditions that lead to incorrect behavior if not fixed
- Hazards are due to dependencies:
  - Dependencies are a property of a program:
    - **Data** dependencies
      - Instruction *j* uses the result of instruction *i*
    - **Control** dependencies
      - The execution of instruction *j* depends on the result of instruction *i*
  - Hazards = how dependencies manifest in the pipeline

Types of hazards

- **Structural** hazards
  - Two different instructions use same resource in the same cycle
- **Data** hazards
  - Two different instructions use same storage
  - Must appear as if the instructions execute in correct order
- **Control** hazards
  - One instruction affects which instruction is next
- Solution:
  - Specific pipeline interlock logic detects hazards and fixes them
    - Simple solution: **stall** the pipeline
      - increases CPI, decreases performance
    - More complex solutions available

Structural hazards

Structural Hazards: example



When two or more different instructions want to use same hardware resource in same cycle

e.g., MEM uses the same memory port as IF as shown in this slide.

Tackling structural hazards (1)

- **Stall**
  - low cost, simple
    - Block PC increment on hazard & fill pipe registers with 0's
  - Increases CPI
    - Used for rare cases since stalling has performance effect
- **Pipeline hardware resource**
  - useful for multi-cycle resources
  - good performance
    - sometimes complex e.g., RAM
- **Replicate resource**
  - good performance
  - increases cost (+ maybe interconnect delay)
    - useful for cheap or divisible resources

Tackling structural hazards (2)

- Structural hazards are **reduced** with these rules:
  - Each instruction uses a resource at most once
  - Always use the resource in the same pipeline stage
  - Use the resource for one cycle only
  - Many RISC ISAs are designed with this in mind
    - Sometimes very complex to do this
- Some common structural hazards:
  - Memory instructions (load/stores)
  - Floating point instructions
    - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.

Tackling structural hazards (3)

- Load/Store hazards can be removed by:
  - Using separate instruction and data memories
    - Usually in the form of I- and D-caches
    - Do not solve the issue completely!
  - Using dual- (or multi-) port memories
    - Two or more simultaneous read/writes are possible!

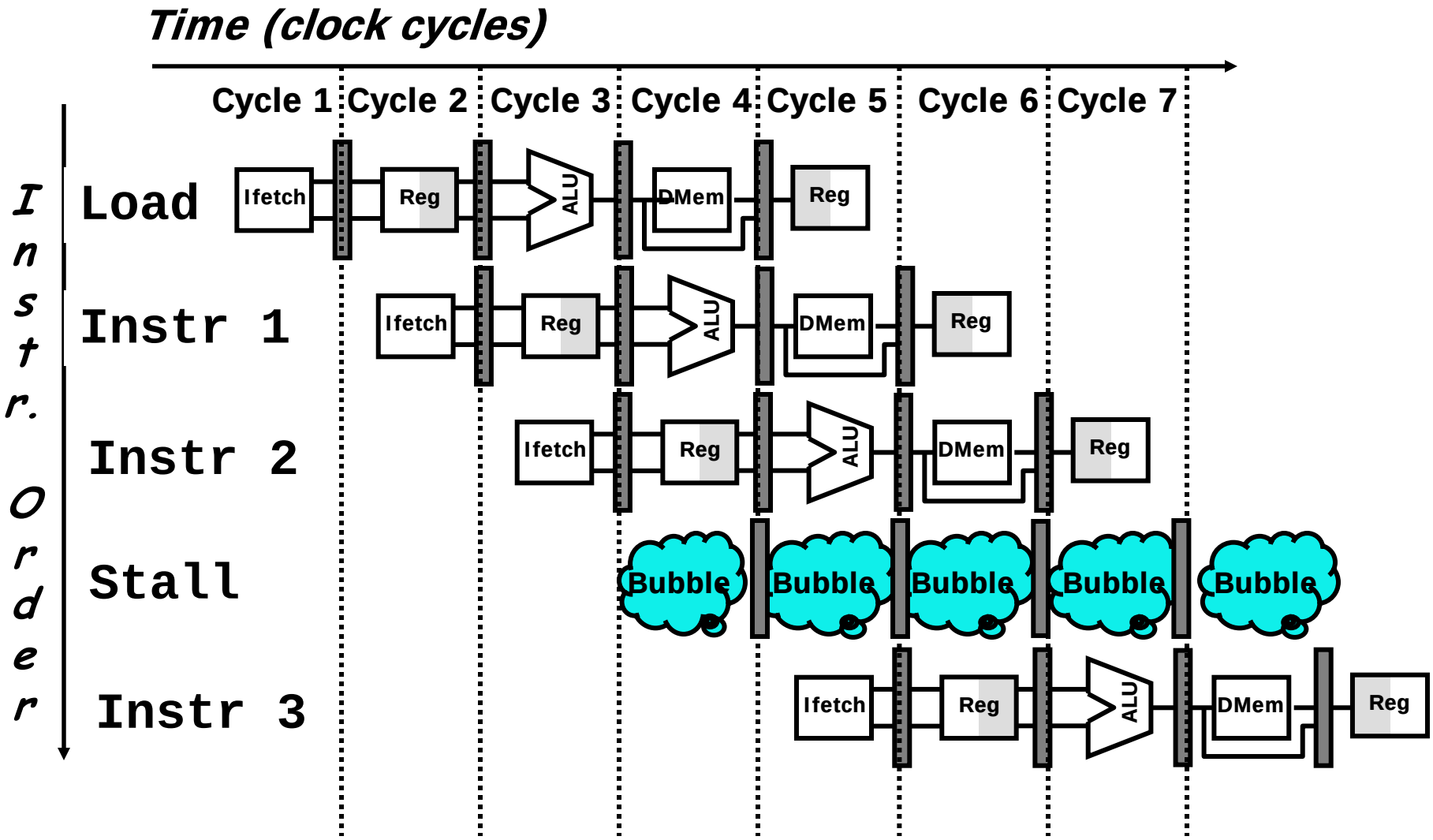
Pipeline stalls (1)

- Stalling the pipeline is the simplest possible solution
  - Stalling is implemented by inserting one or more "bubbles" in the pipeline
- Clearly, the amount of stalling impacts the performance speedup
  - Approximate analysis:
    - $\alpha = T_p / T_{orig}$
    - Assuming β = % of stall cycles

$$\alpha' = T_p / (T_{orig} * (1+ \beta))$$

  - Example:
    - $\alpha = 5$, $\beta = 15\%$ => $\alpha' = 5/1.15 =$ **4.34**

Pipeline stalls (2)



Pipeline stalls (3)

- Another view of stalling

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction j + 1		IF	ID	EX	MEM	WB				
Instruction j + 2			IF	ID	EX	MEM	WB			
Instruction j + 3				stall	IF	ID	EX	MEM	WB	
Instruction j + 4						IF	ID	EX	MEM	WB
Instruction j + 5							IF	ID	EX	MEM
Instruction j + 6								IF	ID	EX

Structural hazard: comments

- Removing a structural hazard depends on its "importance"
  - Example:
    - Hazards due to memory have a significant impact
      - Memory accesses are "popular"
    - Hazards due to FP operations don't
      - FP operations are not very frequent
  - Example 2:
    - FP multiplication not pipelined in MIPS (5 cycles)
    - Impact on CPI depends on:
      - Frequency of FP multiplication (%)
      - Distribution of FP multiplication (clustered or not)
    - Average case:
      - With uniform distribution of FP mult., we can tolerate 1 FP mult each 5 instructions (20%) with negligible penalty

Data hazards

Data Hazards

- Data hazards occur when there are instructions that need to access the same data (memory or register) locations.
- Typical situation:
  - instruction A precedes instruction B
  - *and B manipulates (reads or writes) data* ***before A does****.*
  - Violation of the instruction order
    - The architecture implies that A completes entirely before B!

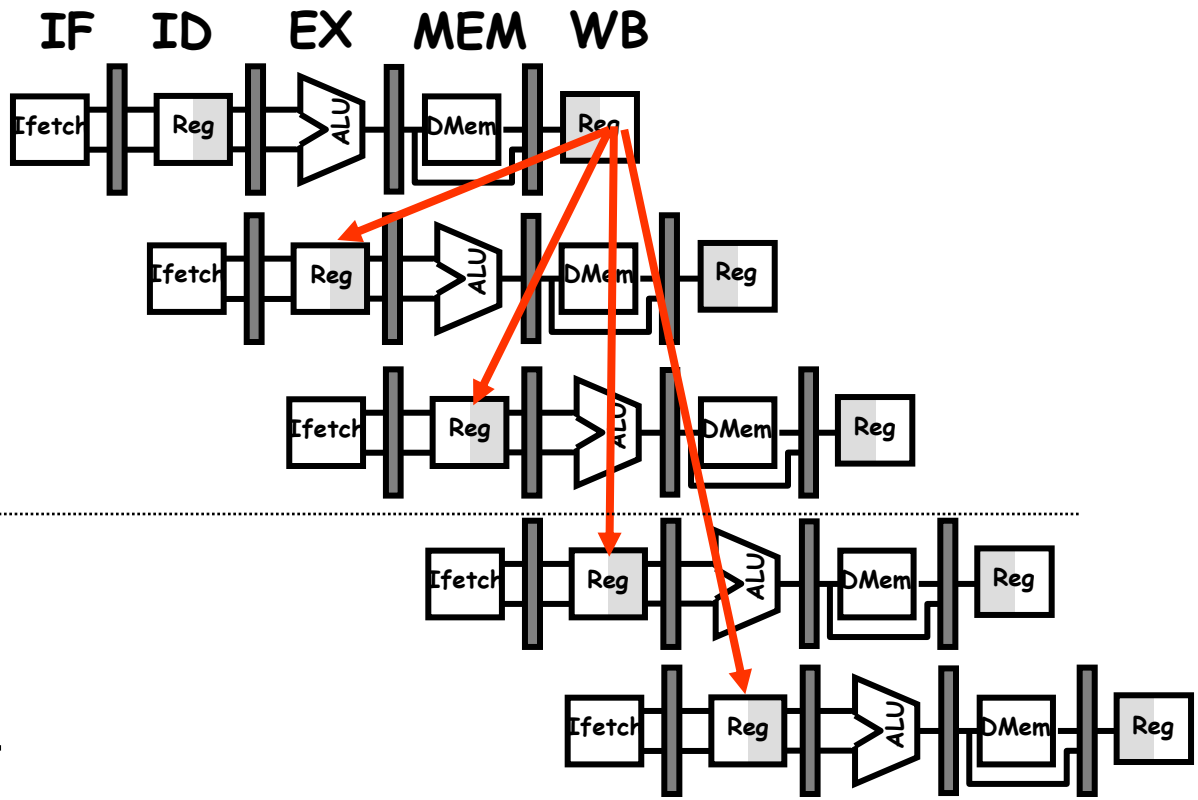
Data hazards: example

- Instruction sequence:
  - **ADD R1, R2, R3**
  - **SUB R4, R5, R1**
  - **AND R6, R1, R7**
  - **OR R8, R1, R9**
  - **XOR R10, R1, R11**
- All instructions use the result of the ADD (R1)
  - Problem:
    - SUB will read the wrong value of R1!!!
    - 2nd AND as well

Data hazards: example (2)

- Visualization:

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

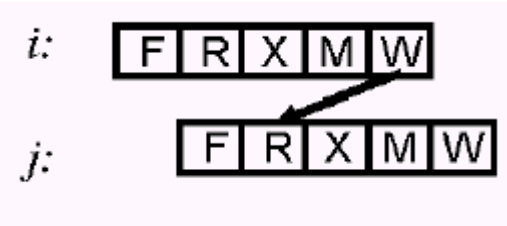


Data hazards classification (1)

- Read After Write (RAW)

  Instr *j* tries to read operand before Instr *i* writes it

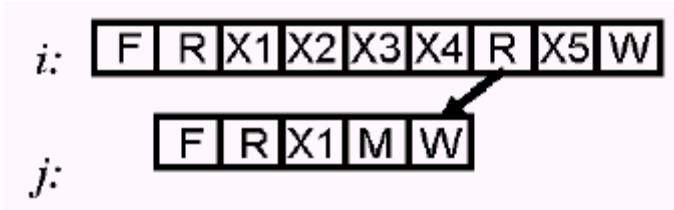
```
I: add r1,r2,r3
J: sub r4,r1,r3
```



Data hazards classification (2)

- Write After Read (WAR)
  Instr *j* tries to write operand before Instr *i* reads it
  - Cannot happen in our pipeline
    - All instructions take 5 stages
    - Reads are always in stage 2
    - Writes are always in stage 5
    - Only for pipelines with “late” read

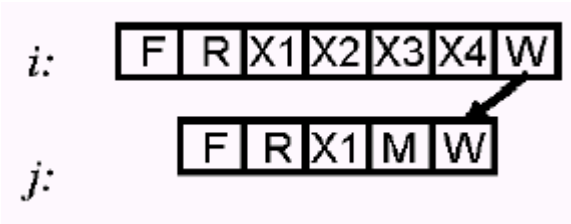
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```



Data hazards classification (2)

- Write After Write (WAW)
  Instr *j* tries to write operand before Instr *i* writes it
  - Leaves wrong result (instr i)
  - Cannot happen in our pipeline
    - All instruction take 5 stages
    - Writes are always in stage 5
    - Only for pipelines with variable length pipelines

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



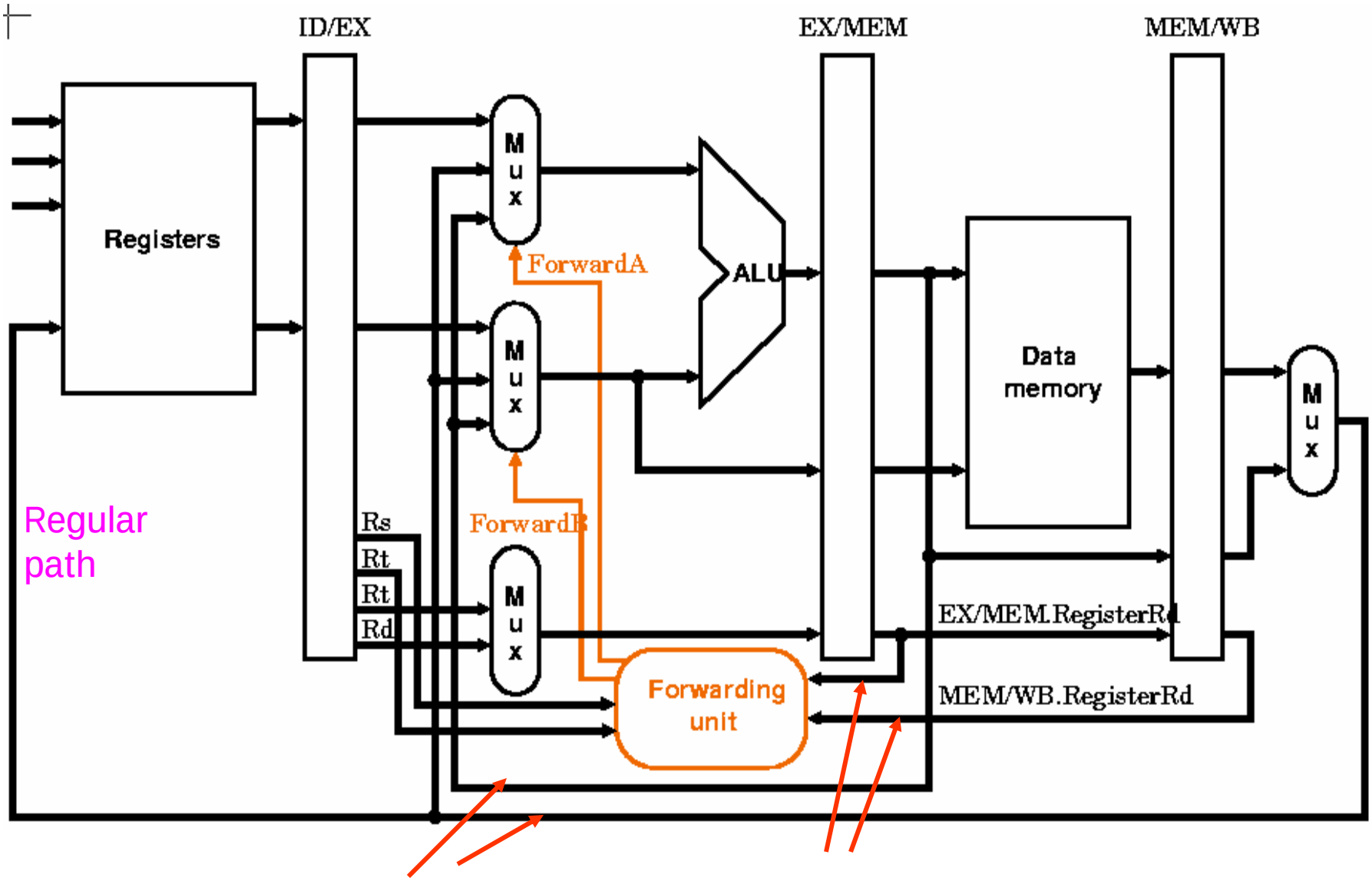
Data hazards removal

- Simple Solution to RAW
  - Hardware detects RAW and stalls
    - + low cost to implement, simple
    - - reduces IPC
  - Not enough: *Should try to minimize stalls*
- Minimizing RAW stalls
  - **Forward** (bypass, short-circuit)
  - **Instruction scheduling**

Forwarding

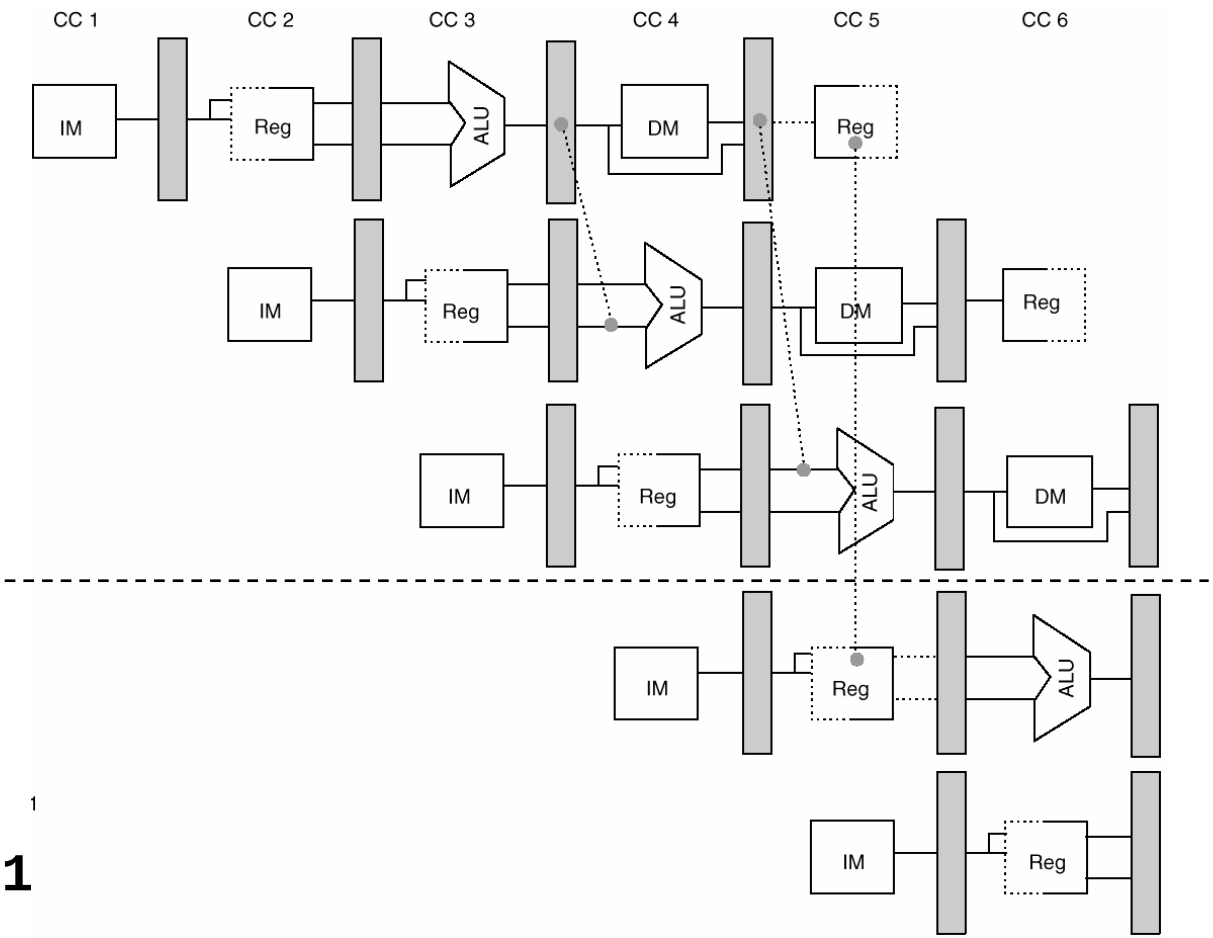
- Forwarding is the concept of making data available to the input of the ALU for subsequent instructions
  - Even if the generating instruction has not arrived yet to WB
- Concept can be extended:
  - Forward = **passing a result to the functional unit that requires it**
- Implementation:
  - Specific forwarding logic required (*detection* logic)
  - Impact on control unit

Forwarding unit



Forwarding

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



Forwarding & stalls

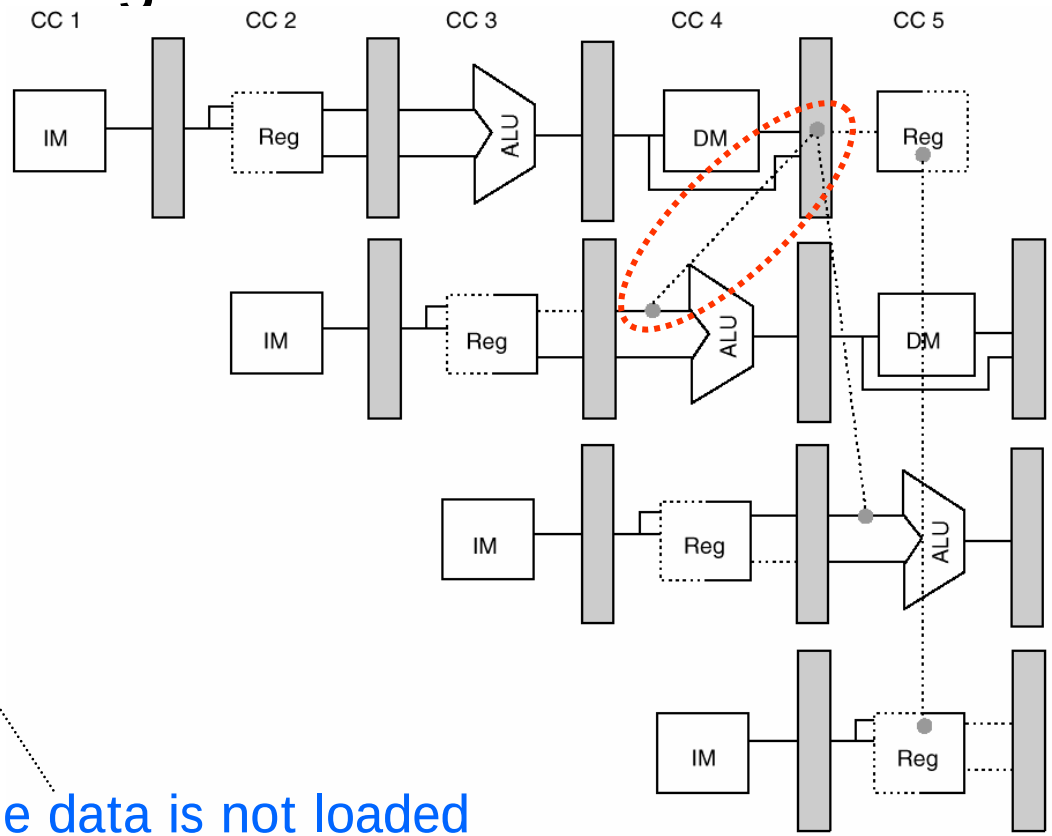
- There are some instances where hazards occur, even with forwarding.

```
lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
```

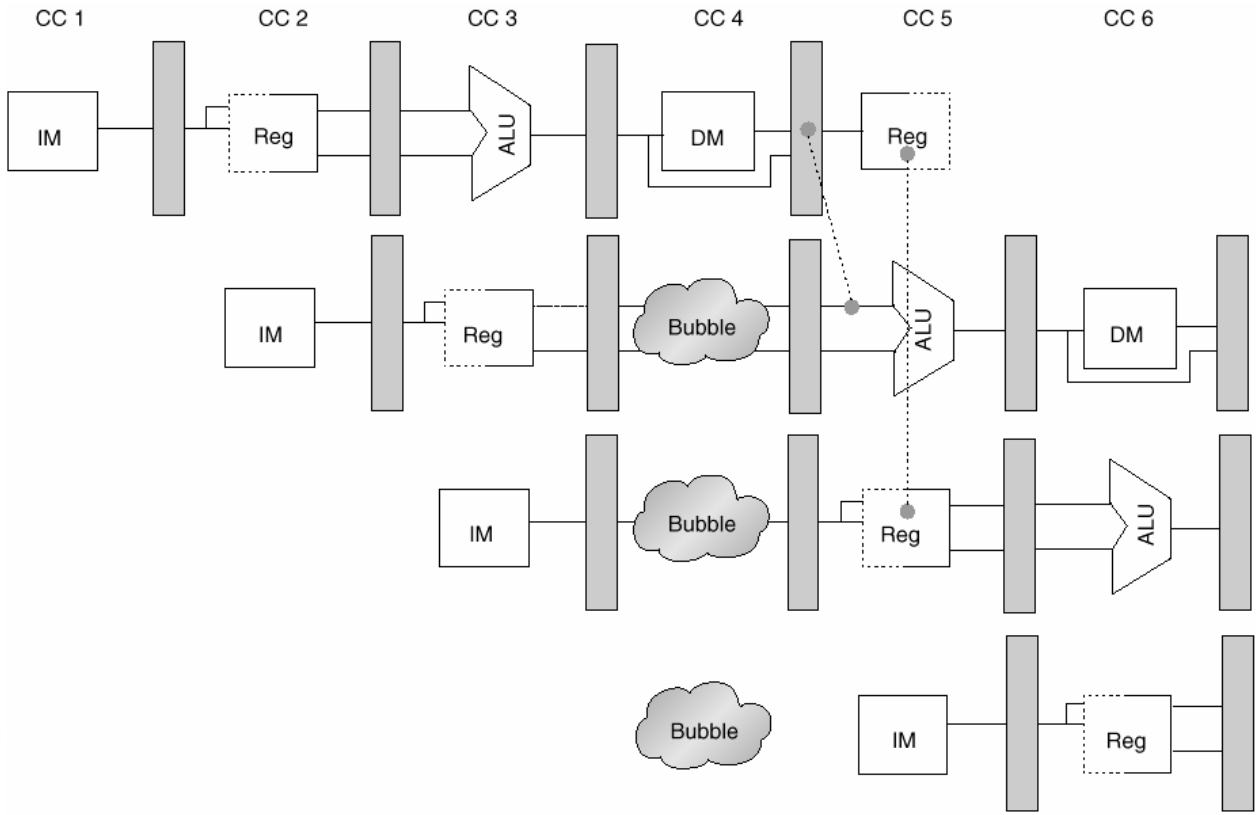


The data is not loaded until after the MEM stage.

Forwarding & stalls (2)

- Actual execution:

```
lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or   r8,r1,r9
```



Compiler scheduling for data hazards

- Many stalls are frequent:
  - Example: Code for **A=B+C** causes a stall for load of **B**

```
LW r1,B
LW r2,C
ADD r3,r1,r2
SW A,r3
```

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	Stall	EX	MEM	WB	
			IF	Stall	ID	EX	MEM	WB

- Rather than just stall, the compiler can schedule instructions so as to avoid the hazard
  - *Pipeline scheduling* (or *instruction scheduling*)
    - Static scheme

Compiler scheduling for data hazards (2)

- Example:

```
a = b + c
d = e – f
```

- Slow code:

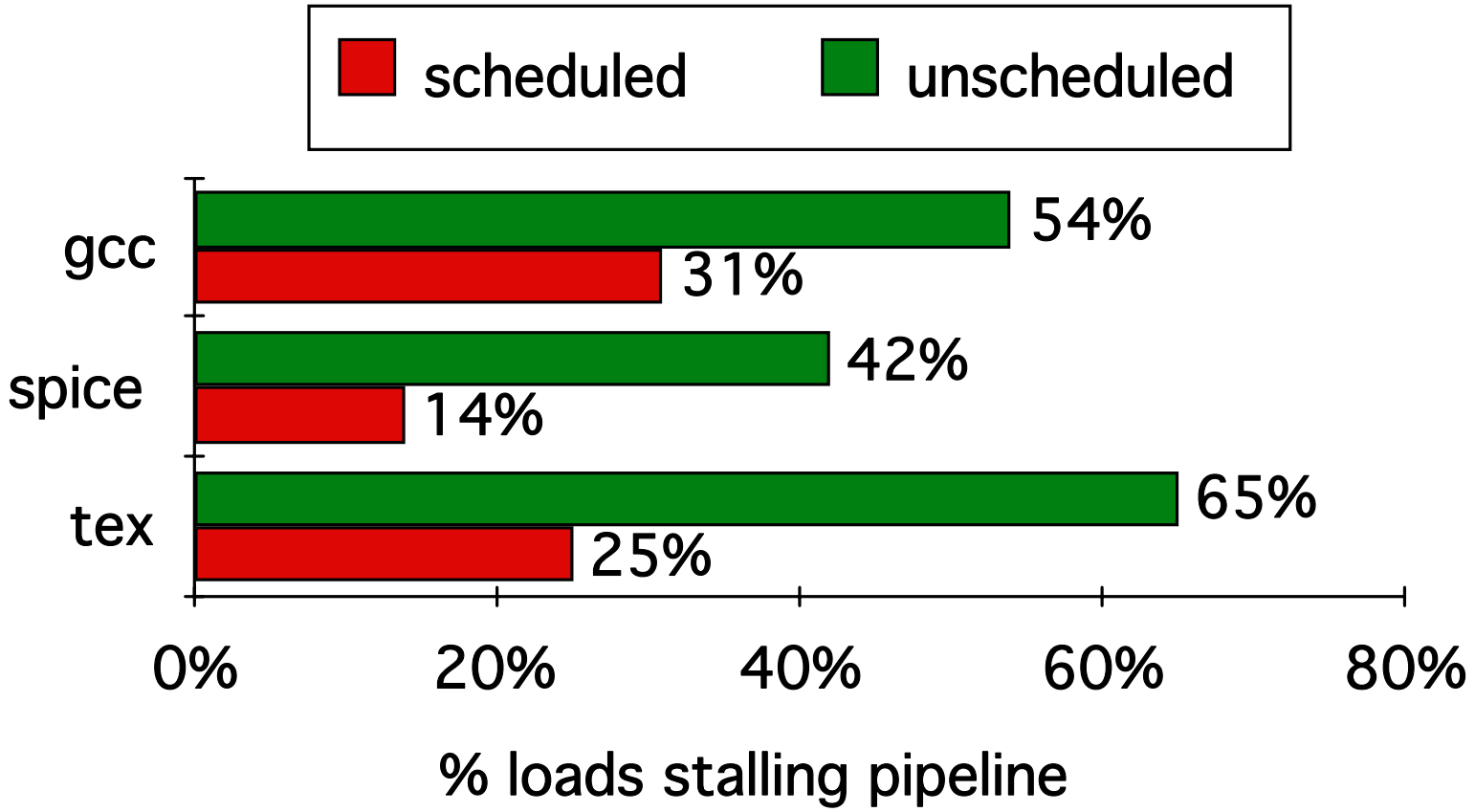
```
LW      Rb,b
LW      Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW      Re,e
LW      Rf,f
SUB     Rd,Re,Rf
SW      d,Rd
```

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
```

Data Hazards

- With pipeline scheduling



Control hazards

Control Hazards

- Control hazards occur when executing branch (or jump) instructions
  - Cannot fetch any new instructions until we know the branch destination (i.e., end of MEM stage)
- Example:

```
40 sub $10, $4, $8
44 add $1, $10, $11
48 beq $1, $3, 20        // jumps to 48+4+20 = 72 if cond.
52 add $1, $2, $3
…
72 lw  $4, 0($1)
```

New PC is known only after the result of the comparison $1=$3 is known:

a) PC = PC+4

b) PC = 72

Branch Stall Impact

- Branches are critical: each branch causes 3 stall cycles

Branch inst.	IF	ID	EX	MEM	WB				
Branch successor		IF	Stall	Stall	IF	ID	EX	MEM	WB
Branch successor+1						IF	ID	EX	MEM
...							IF	ID	EX

- Example:
  - If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9 !!!
- Solution must deal with:
  - Determination of branch taken or not **earlier**

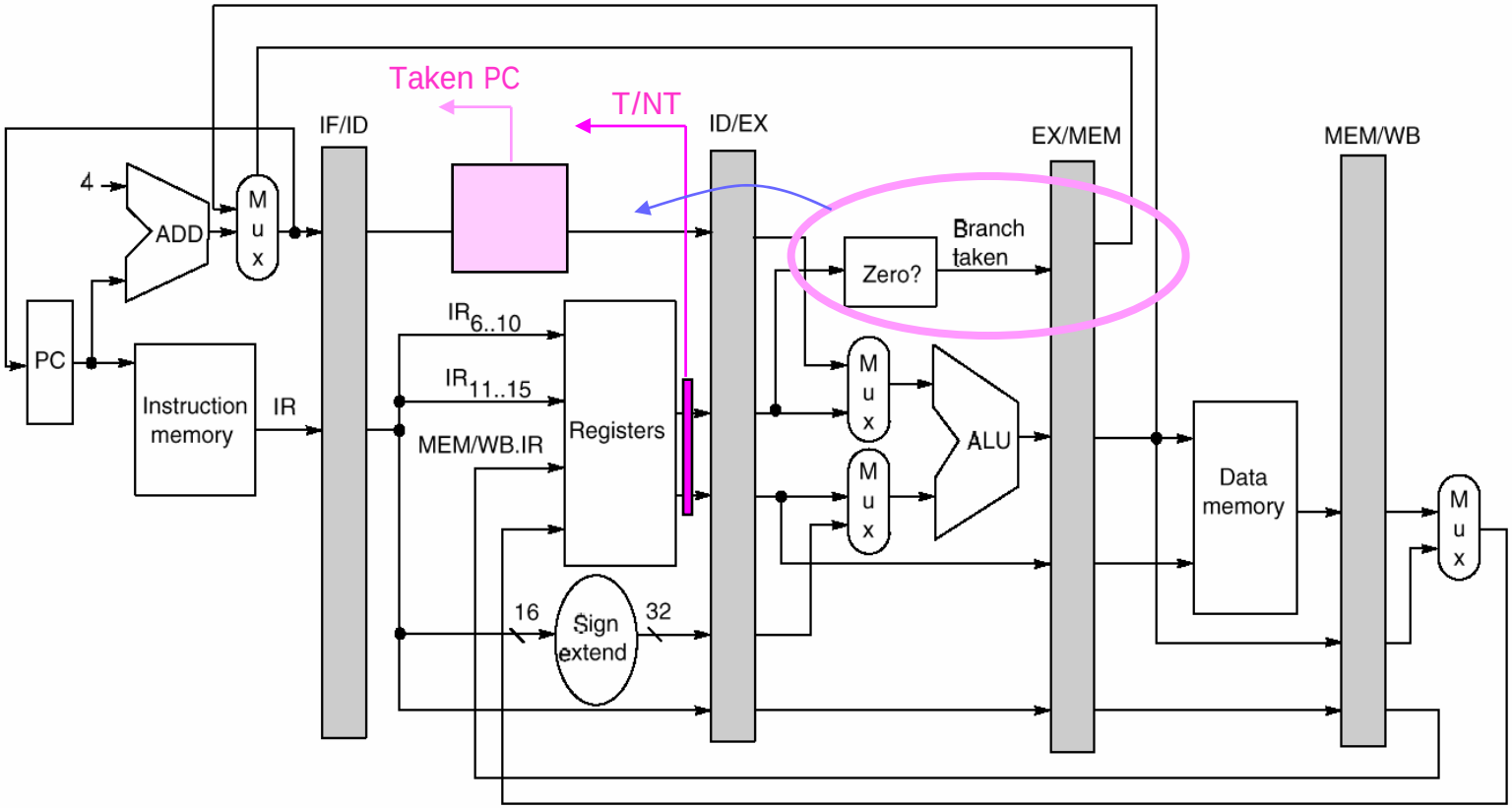
    AND
  - Computation of taken branch address **earlier**

Handling branches

- MIPS pipeline:
  - Result of branch tests is explicitly tested (normally in MEM stage)
- First solution:
  - Move test to ID stage
    - Must be fast
      - compares with 0 are simple
      - >=, <=, >, < must OR all bits
      - more general tests need ALU
  - Add an adder to calculate new PC in ID stage
    - Both taken and not-taken PC are calculated
  - (**Always**) 1 clock cycle penalty for branch (instead of 3)

Handling branches (2)

- Anticipating branch target calculation

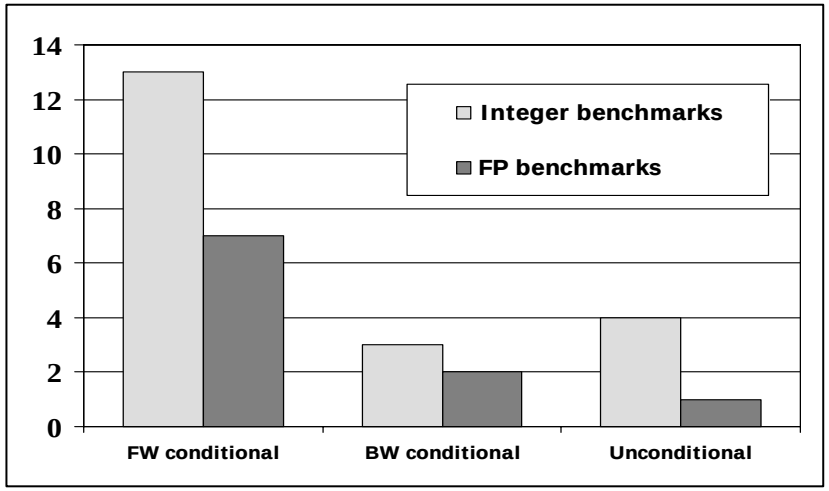


Branch analysis

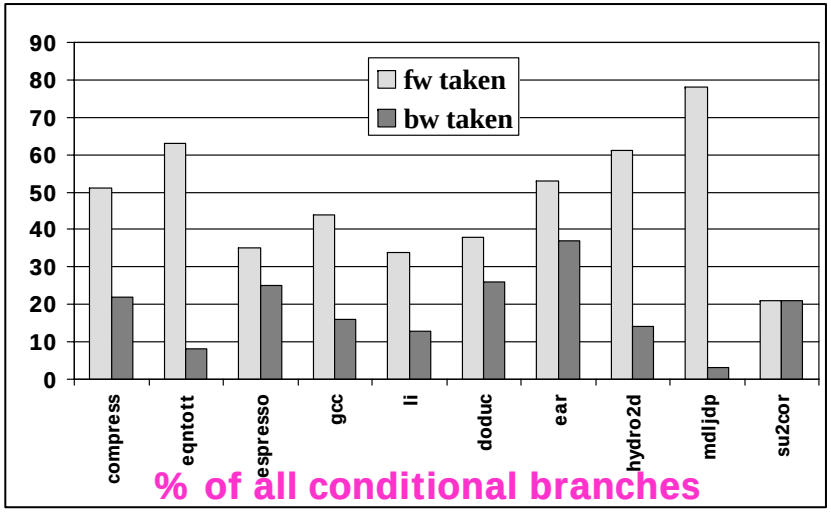
- Classification
  - Conditional branches
    - Forward branches
    - Backward branches
  - Unconditional
    - Jumps
- Taken/not taken distribution:

  67% of all conditional branches are taken

SPEC95 distribution



% of executed instructions



% of all conditional branches

Branch prediction techniques

- Stalls branch hazards can be almost completely eliminated
- Solutions:
  - **Static** predictions on the result of a conditional branch (taken/not taken)
    - Compiler-driven
    1. Predict Branch Not Taken
    2. Predict Branch Taken
    3. Delayed Branch
  - **Dynamic** predictions
    - Hardware-driven

Predict Branch Not Taken (1)

- Execute successor instructions in sequence (as if branch were not executed)
  - In ID actual condition is evaluated:
    - If not taken, OK (**no penalty**!)
    - If taken, we must:
      - Replace current instruction with a NOP
      - One stall cycle in pipeline if branch actually taken
  - Care must be taken not to change the machine state until the branch outcome is definitely known.
  - Problem: only 33% of branches are untaken...

Predict Branch Not Taken (2)

- Example

Untaken Branch Instr	IF	ID	EX	MEM	WB		
Instr i+1		IF	ID	EX	MEM	WB	
Instr i+2			IF	ID	EX	MEM	WB

Taken Branch Instr	IF	ID	EX	MEM	WB			
Instr i+1		IF	*idle*	*idle*	*idle*	*idle*		
Branch target			IF	ID	EX	MEM	WB	
Branch target+1				IF	ID	EX	MEM	WB

Predict Branch Taken

- Predict Branch Taken
  - 67% branches taken on average
  - Execute instruction corresponding to branch target address
  - No advantage (but the higher probability):
    - branch target address in MIPS is known no earlier than branch result (regardless of anticipation)
    - Still one cycle branch penalty
      - On other machines: branch target may be known before outcome

Delayed branch (1)

- Generic structure:

```
branch instruction
sequential successor1
sequential successor2
........
sequential successorn
branch target if taken
```

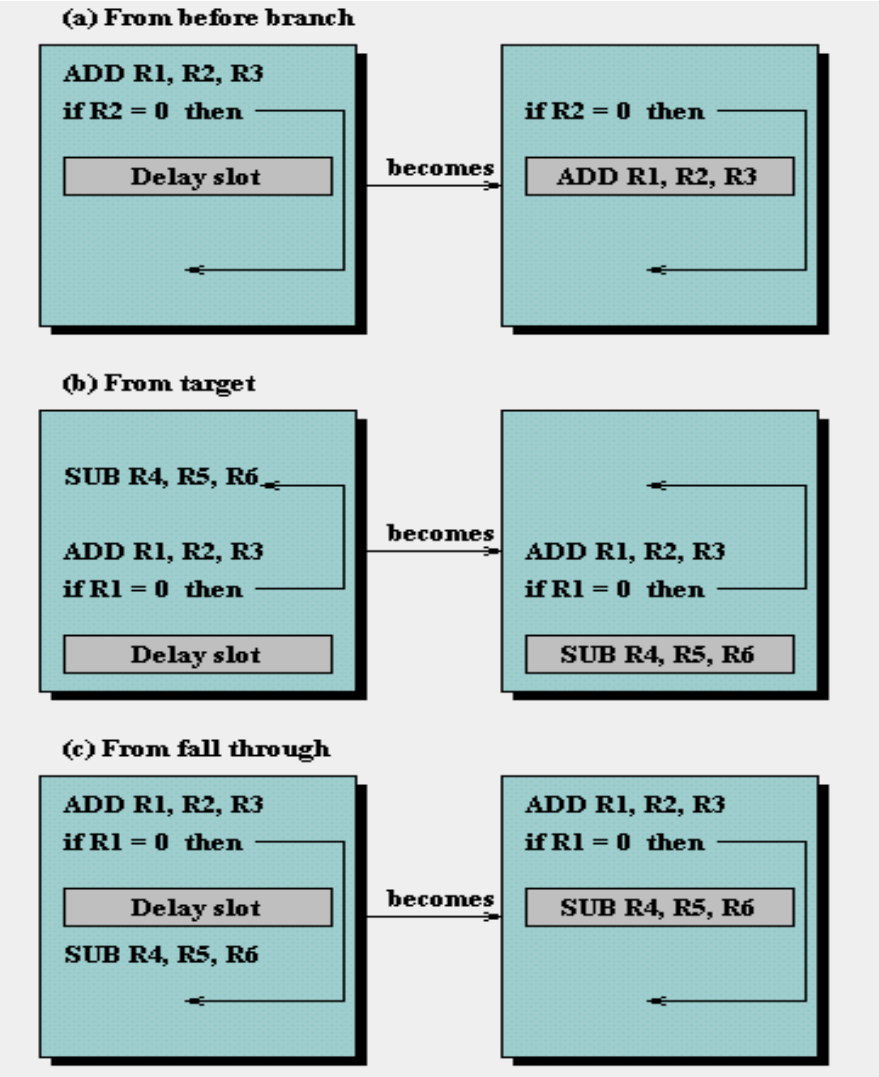
Fall-through: instructions that could be executed while determining the result of the branch test

  - **Branch delay slot (BDS) =** the number of cycles required to resolve branch
    - In MIPS, BDS = **1**
- *In practice, execute the instruction(s) in the BDS regardless of the branch result*

Delayed Branch (2)

- What instructions are used to fill BDS?
  - Three options:
    - From before the branch
    - From the target address
    - From fall through
- Who fills BDS?
  - Typically done by the compiler!
  - Could be the programmer

Delayed Branch (3)



a) From before:
- Best solution, used when possible
- Branch must not depend on the rescheduled instructions

b) From target:
- Sub-optimal
- Usually the target instruction will need to be copied because it can be reached by another path
- *Effective for highly-taken branches*

c) From fall through
- *Effective for highly-not-taken branches*

- To make this optimization legal for (b) and (c), it must be OK to execute the SUB instruction when the branch goes in the unexpected direction.
  - That is, work might be wasted but the program will still execute correctly.

Canceling branch

- To improve the ability of the compiler to fill branch delay slots, most machines with conditional branches have a **cancelling branch:**
  - If the branch behaves as predicted, the instruction in the branch delay slot is executed as in a delayed branch
  - If the branch is incorrectly predicted, the instruction in the delay slot is turned into a NOP
- Result:
  - Requirements on the instruction placed in the delay slot are removed
    - Solutions b) and c) are now usable

Canceling branch (2)

- Example:

Executed anyway, but made a NOP

Untaken branch instr	IF	ID	EX	MEM	WB				
Branch delay instr(i+1)		IF	ID	***idle***	***idle***	***idle***			
Instr i+2			IF	ID	EX	MEM	WB		
Instr i+3				IF	ID	EX	MEM	WB	
Instr i+4					IF	ID	EX	MEM	WB

Taken branch instr	IF	ID	EX	MEM	WB				
Branch delay instr(i+1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target+1				IF	ID	EX	MEM	WB	
Branch target+2					IF	ID	EX	MEM	WB

Predicted-taken canceling branch

Evaluating Branch Alternatives

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

$$CPI_{pipelined} = CPI_{ideal} + \#\ of\ stall\ cycles\ per\ instruction =$$
$$= 1 + \#\ of\ stall\ cycles\ per\ instruction$$

Scheduling scheme	Branch penalty	CPI	speedup v. unpipelined	Speedup v. stall
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1 (0.33)	1.14	4.4	1.26
Predict not taken	1 (0.67)	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Conditional & unconditional branches= 14%, 65% change PC

NOTE: (ex.: 1.42 = 1 + 3*0.14)

Filling branch delay slots

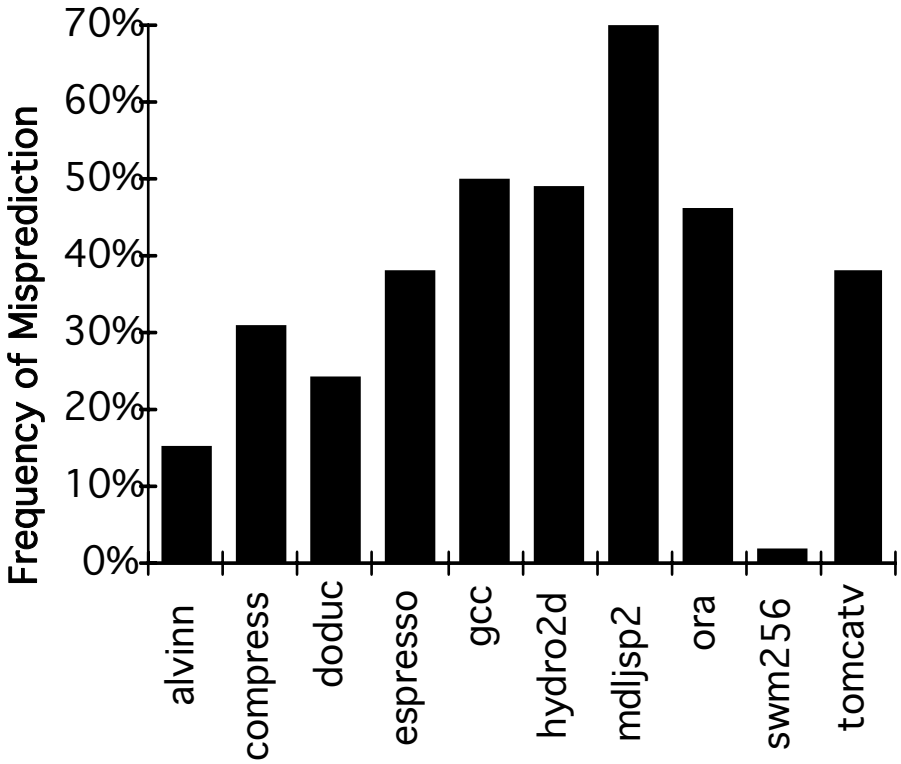
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Not very used anymore
  - Availability of HW resources allows **dynamic** (HW) branch prediction

Compiler-driven branch prediction

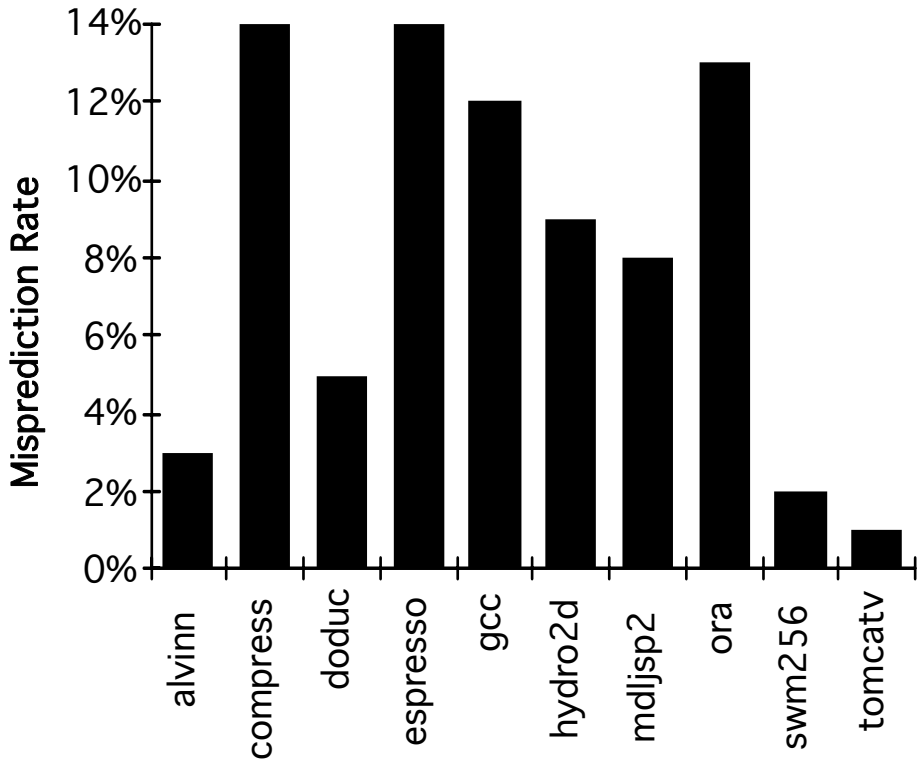
- Branch prediction could be done during compilation
  - Still a static prediction
  - Can help compiler to decide how to fill BDSs
- Two strategies:
  - **Static analysis of program behavior**
    - Backward branch predict taken, forward branch not taken
    - Based on statistics
  - **Using profile** (i.e., run time) **information**
    - Record branch behavior, predict branch based on prior run

Compiler-driven branch prediction (2)

- Prediction from static analysis



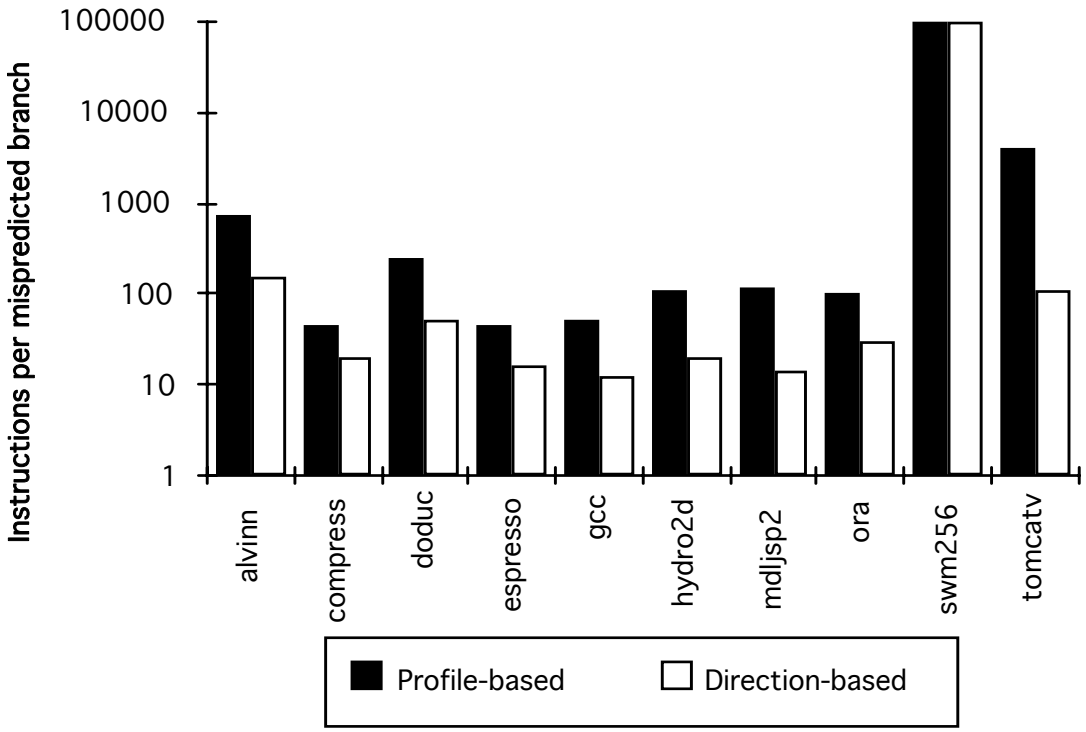
Always taken



Taken backwards
Not Taken Forwards

Compiler-driven branch prediction (3)

- Misprediction rate ignores frequency of branch
  - How “critical” are these branches?
- Better metric:
  - “Instructions between mispredicted branches”



Dynamic branch prediction

Dynamic branch prediction

- Dynamic = decision changes over time based on past history
- **Done by hardware**
- Conceptually: $F(x_1, x_2, \ldots, x_n)$
  - F : function expressing the result of a branch prediction
  - $x_1, x_2, \ldots, x_n$: parameters that affect F
    - related to prediction history
  - If $F > 0.5$ branch taken, otherwise not taken
- Example:
  - $F(X) = X$ (X = result of last branch)
    - Poor, all predictions treated the same way, regardless of their individual probabilities

Branch History Table (BHT)

- Simplest solution:
  - Use a table that stores branch history
  - During IF, access BHT to predict branch outcome
  - During ID, check if this is a branch
- Implemented as a (fully associative) cache:
  - LRU replacement

Branch PC	stats
...	...
0x00ff3d0f	011

  - To save space, only some (5÷6) LS bits of PC are stored
  - History = n bits

Branch History Table : example

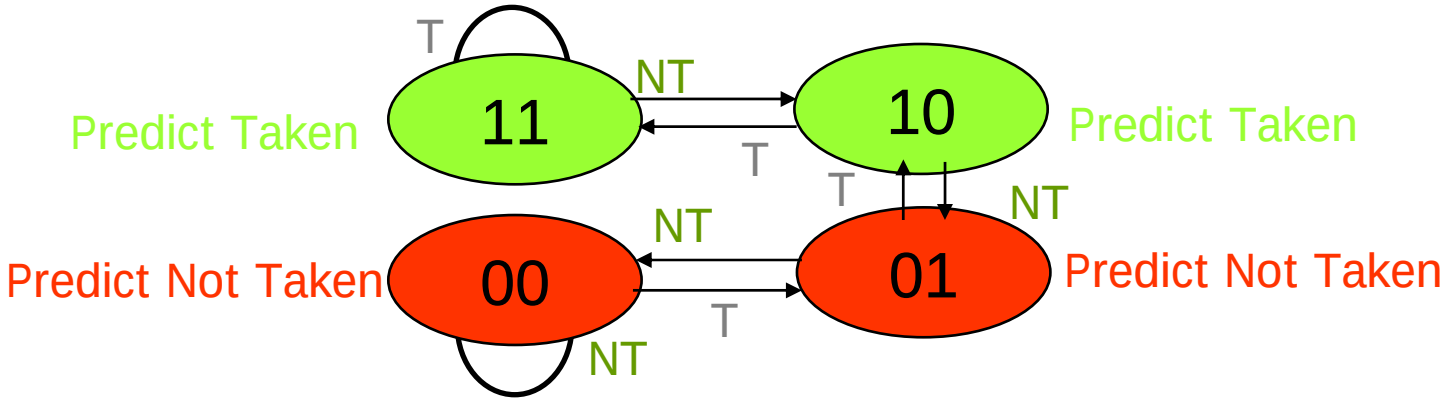
- 1-bit history:

```
sub   r1,r1,r1          ;r1:= 0
      add   r1,r1,10    ;r1:= 10
loop:
      subi r1,r1,1      ;r1--
      bnez r1,loop
```

  - First 9 times, branch is taken (history bit =1)
  - 10th time: branch not taken (history bit =0)
  - Next time, branch will be not taken (**error**)
- Branch actually taken 90%
  - Prediction only 80%

Branch History Table

- Typical solution uses a 2-bit prediction
  - Change only if mispredicted twice
  - Updated as 2-bit saturating up-down counter



  - Experimental data (SPEC95)
    - P(NN) = 0.11
    - P(NT) = 0.54 Success probability
    - P(TN) = 0.61
    - P(TT) = 0.97

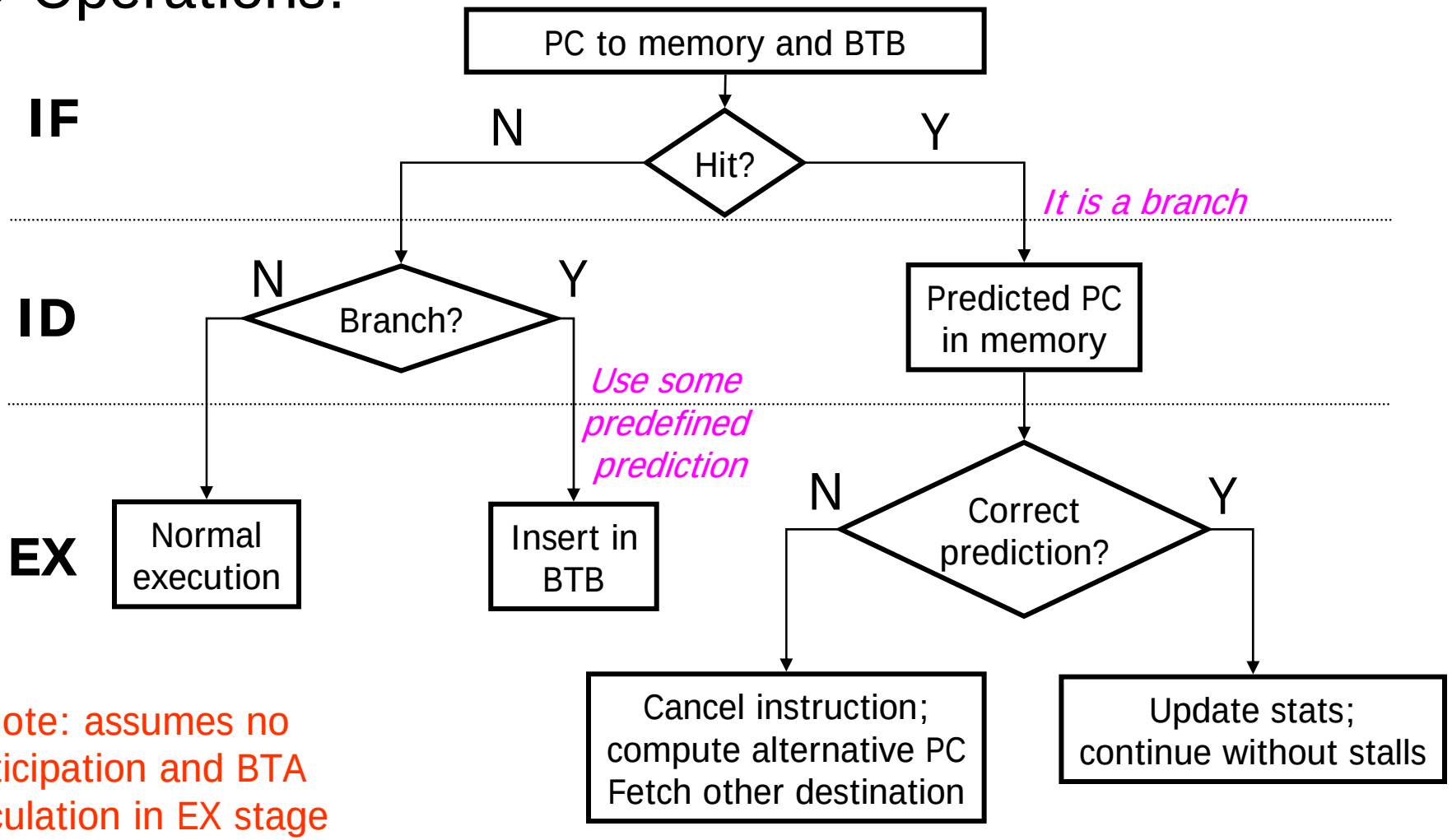
Branch target buffer (BTB)

- With BHT we only save **time** for the computation of the branch condition
- Can we predict the branch target address?
  - Include in BHT branch targets!
  - **Branch Target Buffer**

Branch PC	Branch target address	stats

Branch target buffer (2)

- Operations:



Note: assumes no anticipation and BTA calculation in EX stage

BTB performance

- Cases 1 & 3:
  - BTB hit and correct prediction: 0 penalty
- Case 2:
  - BTB hit and wrong prediction
  - 1 cycle only (get PC+4)
- Case 4:
  - BTB hit and wrong prediction
  - 2 cycles (must wait for computation of correct address) (**Assuming target address calculation in EX)**
- Case 5 & 6:
  - BTB miss
  - NT default prediction => 2 cycle for T result

Case	BTB hit	Prediction	Result	Penalty cycles
1	Y	T	T	0
2	Y	T	NT	1
3	Y	NT	NT	0
4	Y	NT	T	2
5	N	(NT)	T	2
6	N	(NT)	NT	0

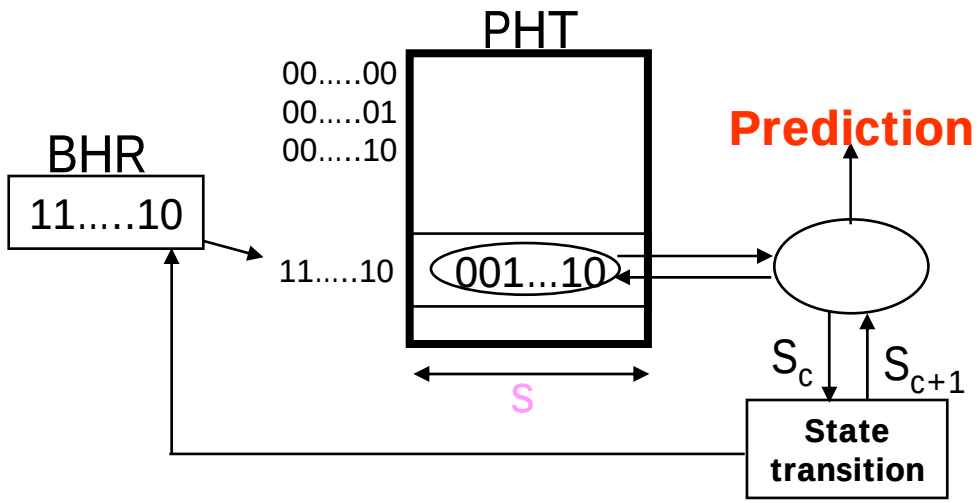
(NT prediction for BTB miss)

Two-level predictors

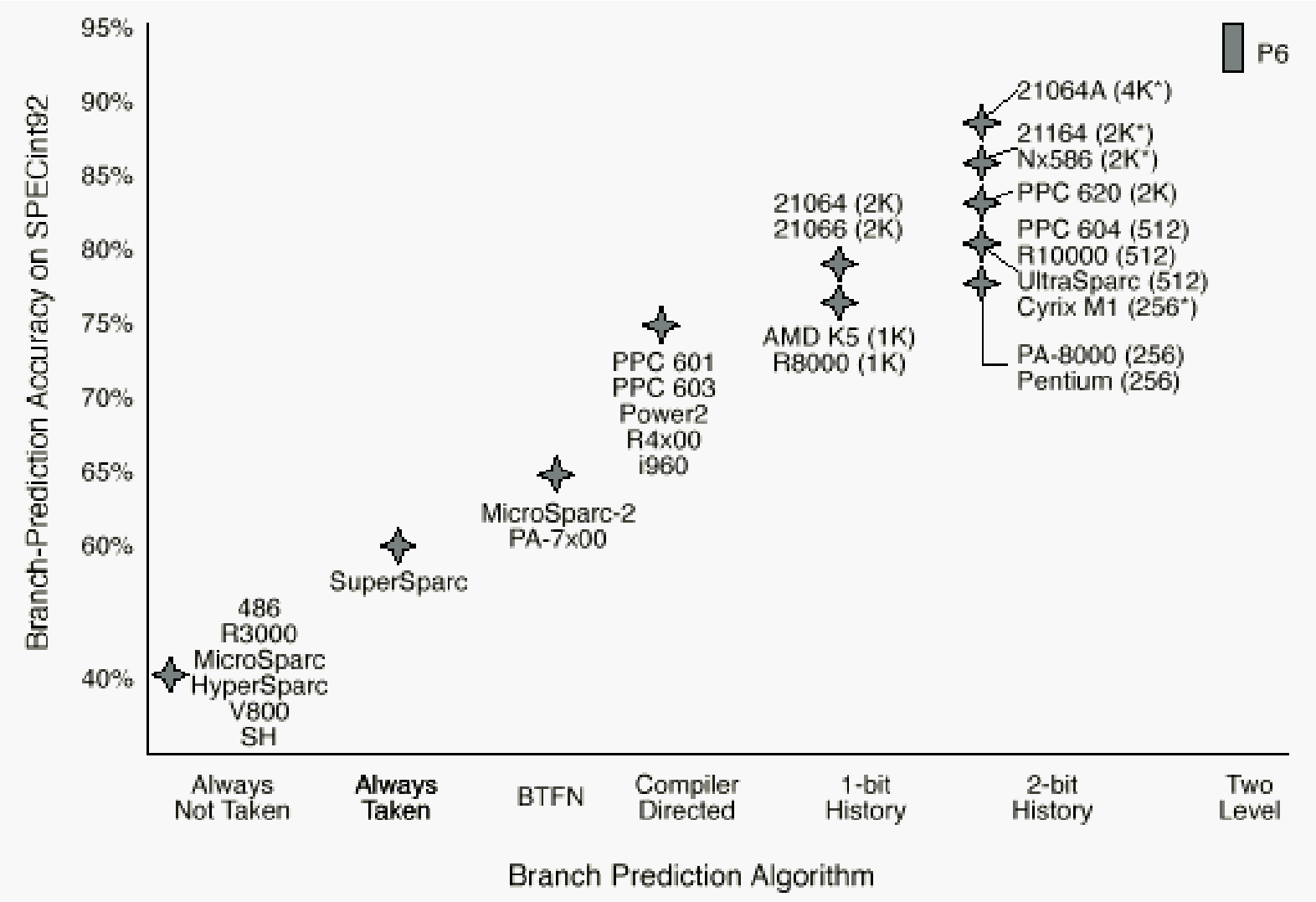
- To improve prediction accuracy use a two-level mechanism
  - First level: use the history of last *k* branches
  - Second level: branch result for last *s* times it was preceded by that history
- Example: k=8, s=6
  - Last k branches yielded 11100110 (1=T, 0=NT)
  - Last s times this pattern appeared result was 101010
  - Decision is 1 (=T)

Two-level predictors: implementation

- Two tables:
  - ***Branch History Register (BHR)***
    - K-bit shift register (contains history of last k branches)
    - Used as index in Pattern History Table
  - ***Pattern History Table (PHT)***
    - 2^k entries
    - Each entry contains s bits
- Control:
  - FSM:



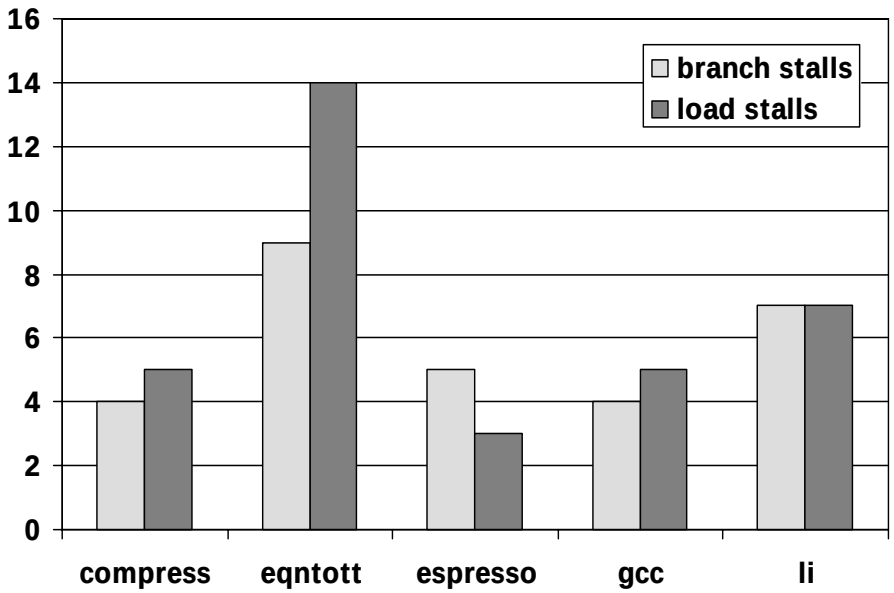
Branch prediction accuracy



From Microprocessor Report

Impact of hazards: summary

- Results for some SPEC benchmarks:
  - % of stalled instructions
  - Average:
    - 6% branch stalls
    - 5% load stalls
- Resulting **CPI = 1.11**
  - Assuming:
    - Perfect memory system
    - No clock overhead



Summary

- Pipelining helps instruction bandwidth, not latency
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
- Hardware can improve that