



UNIT 2: PIPELINED PROCESSOR

Estructura de Computadores (Computer Organization)

Course 2018/2019

ETS Ingeniería Informática

Universitat Politècnica de València

Unit goals

- To learn the concept of pipelining as a technique to improve the processor's performance
- To evaluate the throughput of a pipelined processor
- To compare the performance of pipelined vs. non-pipelined processors
- To understand the new, specific problems introduced by pipelining and ways to mitigate them

Unit contents

- 1 An overview of pipelining
- 2 Measuring performance
- 3 Pipelined datapath and control
- 4 Data hazards
- 5 Control hazards
- 6 Beyond pipelining

Bibliography

- D. Patterson, J. Hennessy. *Computer organization and design. The hardware/software interface*. 4th edition. 2009. Elsevier
 - Chapter 4, sections 4.5 to 4.8 and 4.15

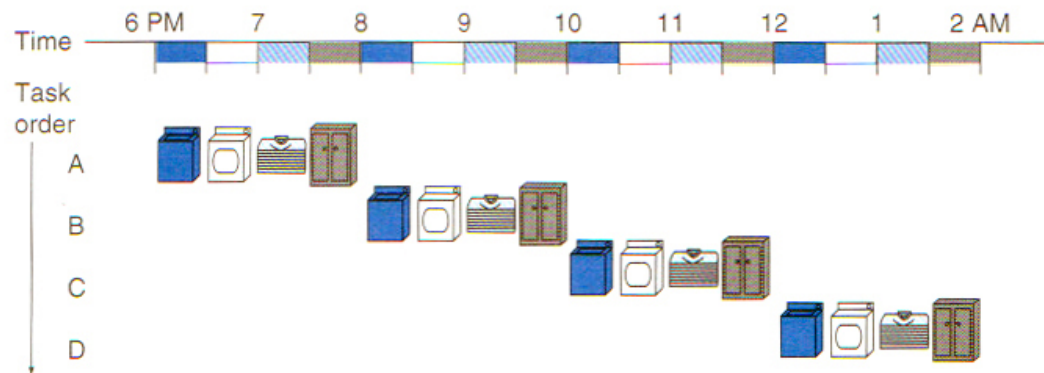
An overview of pipelining

- Processor pipelining is an implementation technique in which execution of multiple instructions is **overlapped**
- Pipelining requires the processor to be composed of **separate stages**, each dealing with a particular phase of execution
 - At a given time, each stage is working on a different instruction
- There are many examples of pipelining in other contexts
 - Manufacturing – each operator works on a particular process stage
 - Laundry – washing, drying, ironing, folding, storing

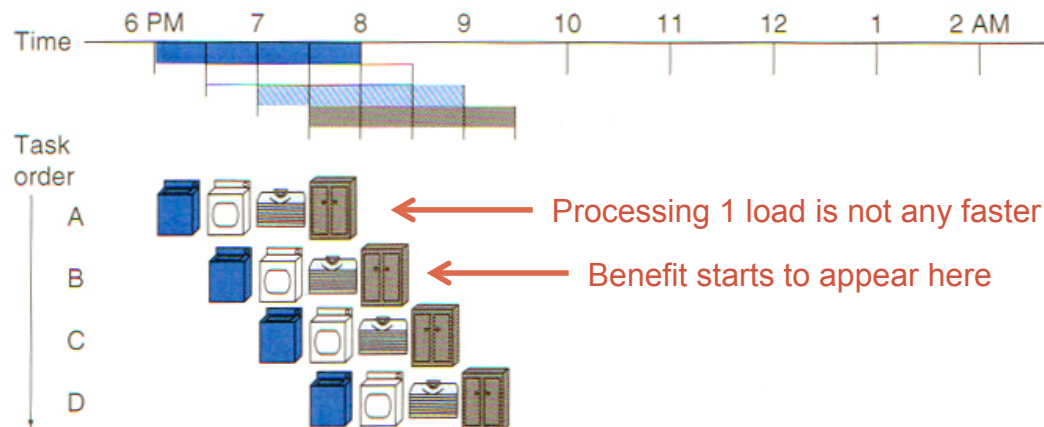


The laundry analogy

Four stages, each taking 30': washing, drying, folding, storing



The *sequential laundry* takes 8 hours to process 4 tasks
 $[4 \text{ tasks} \times 4 \text{ stages} \times 30']$



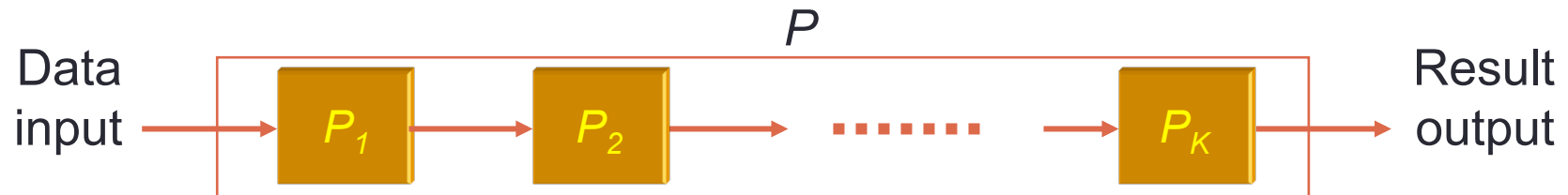
The *pipelined laundry* takes only 3.5 hours to process the same 4 tasks
 $[4 \text{ tasks} + (4-1) \text{ stages}] \times 30'$

Same work done in just 43.75% of the time!

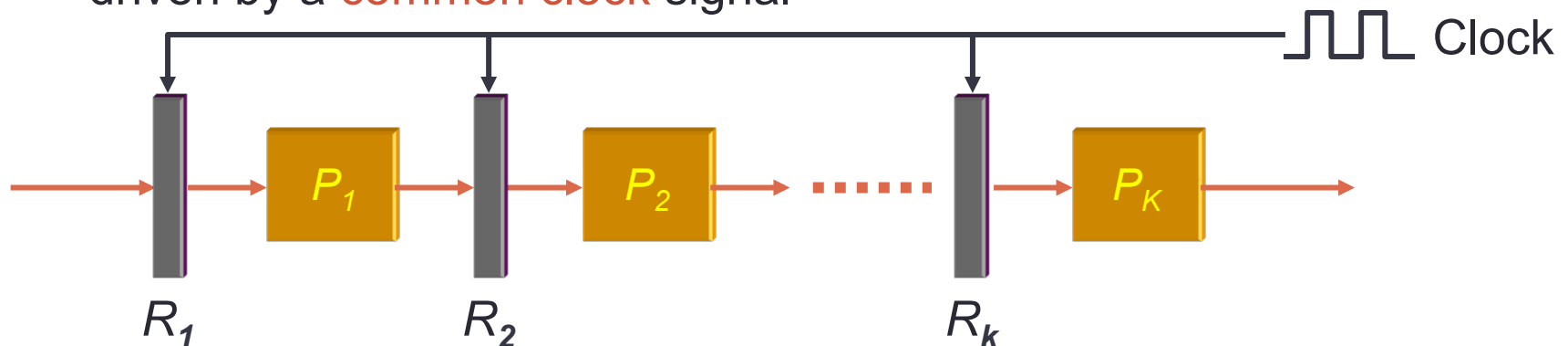
All resources used simultaneously

Pipelining a non-pipelined circuit

- Starting at...
 - non-pipelined logical operator P formed by K sequential stages



- ...arriving at
 - pipelined operator with K stages separated by **staging latches** driven by a **common clock** signal



Timing considerations

- Non-pipelined circuit
 - Each stage of the sequential process P_i takes a time τ_i
 - The whole sequential process takes $T = \tau_1 + \tau_2 + \dots + \tau_K$
 - Processing n items requires $n \times T$ seconds
- Pipelined circuit
 - Each stage has an added t_R delay due to its pipeline latch
 - The clock period must be set to a minimum of $\tau = \max(\tau_i) + t_R$
 - Processing n items requires $(K + n - 1) \times \tau$ seconds
- Discussion
 - Pipelining penalizes the processing of a single data item
 - It is only beneficial for processing a large number of items (n)
 - It is important to have balanced stages of a similar delay τ_i
 - To minimize wasted time in *fast* stages

2. Measuring performance

- Productivity (or throughput, P)

- For non-pipelined circuits

$$P_{np}(n) = \frac{n}{nT} = \frac{1}{T}$$

- Where n = nr. of data items processed, or instructions executed, or clothes washed... and T is the delay to process one item

- Pipeline productivity

$$P_p(n) = \frac{n}{(K + n - 1)\tau}$$

$$\lim_{n \rightarrow \infty} P_p(n) = \frac{1}{\tau} = f_{CLK}$$

- Where K = nr. of stages and τ = clock period
- With sufficient input data, $P_p \rightarrow f_{CLK}$

- Speedup

- Compares non-pipelined vs. pipelined

$$S(n) = \frac{P_p(n)}{P_{np}(n)} = \frac{t_{np}(n)}{t_p(n)} = \frac{nT}{(K + n - 1)\tau}$$

- Speedup grows with n and the clock frequency

Measuring performance

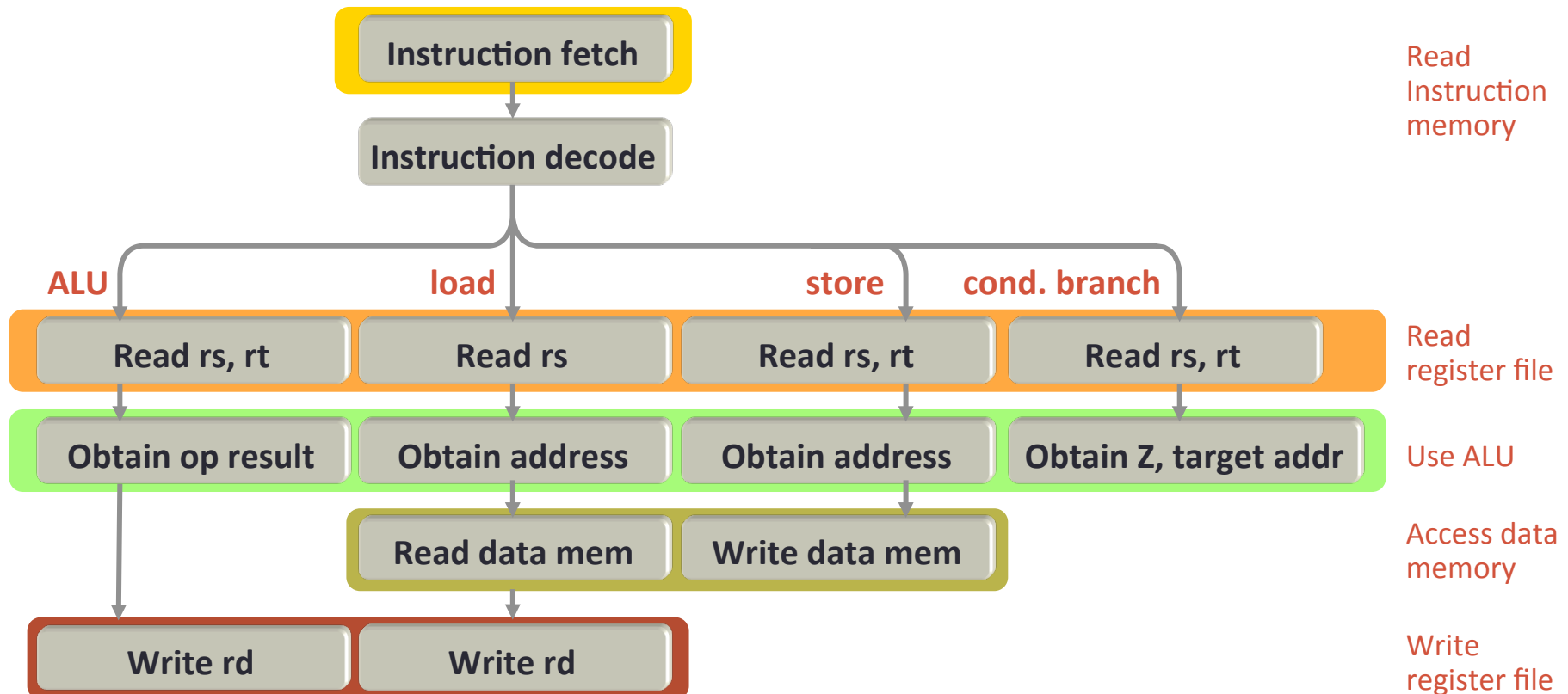
- Conclusion

$$S(n) = \frac{P_p(n)}{P_{np}(n)} = \frac{t_{np}(n)}{t_p(n)} = \frac{nT}{(K + n - 1)\tau}$$

- Since $T \leq K \times \tau$, there's no effective speedup when $n=1$
- The larger n , the higher S
 - When $n \rightarrow \infty$, then $S \rightarrow T/\tau \leq K$ (maximum theoretical speedup)
- If $T = K \times \tau$ (ideal case), then $S = K$
 - This is only possible when all stages have the same delay and $t_R = 0$
- In practice, with sufficient n , the lower τ the higher S
- In order to increase productivity, since $\tau = \max(\tau_i) + t_R$, we must try to reduce both τ_i (short, balanced stages) and t_R

3. Pipelined datapath and control

- Identify stages and how instructions will use them
- Execution stages for a reduced instruction subset (unit 1):



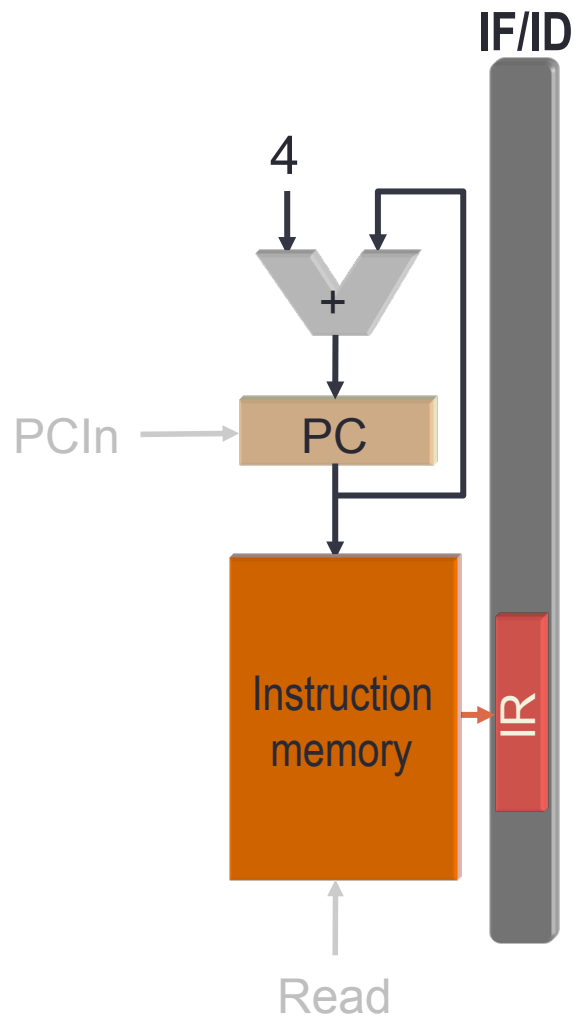
Definition of stages

- Common stages (to all instructions)
 - **IF**: Instruction Fetch (includes PC increment)
 - **ID**: Instruction decode (includes reading 2 source registers)
- Instruction-dependent stages
 - **EX**: Execution
 - ALU instructions use the ALU
 - Load/store calculate memory address in ALU
 - Conditional branch (beq) calculates the branch condition in ALU
 - **M**: Data memory stage
 - Load/store access memory for reading/writing respectively
 - ALU and beq instructions do nothing in this stage
 - **WB**: Write back
 - ALU and load instructions write the result back to the register file
 - Store and beq do nothing in this stage

Pipeline latches

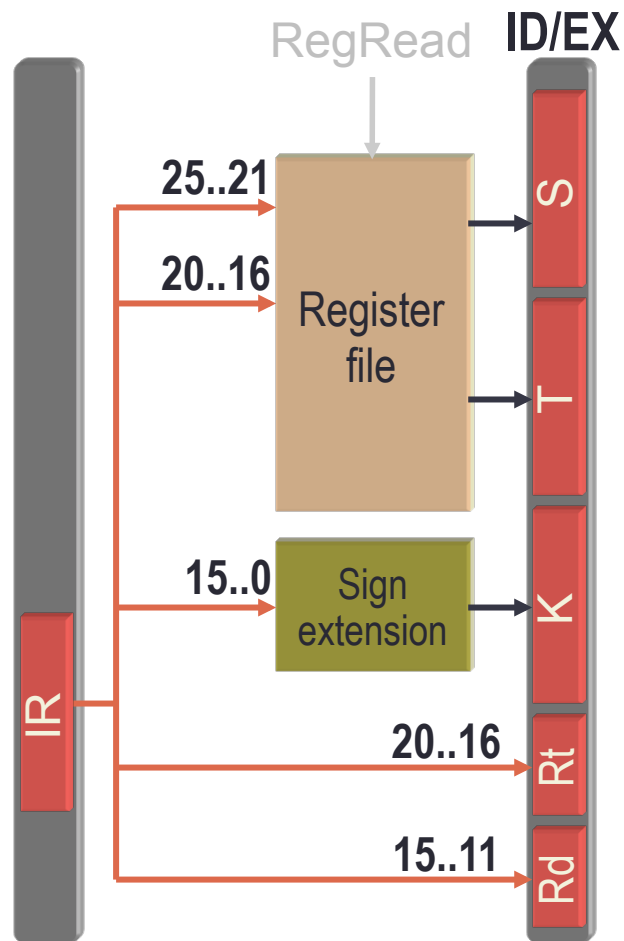
- We need only four, not five!
 - Note that instructions (input) are already “latched” in I-mem
 - Each pipeline latch is structured in fields of information
- Notation
 - A latch connecting the IF and ID stages is called IF/ID
 - So we have IF/ID, ID/EX, EX/M, and M/WB pipeline latches
 - Register fields are identified with dot notation
 - For example, IF/ID.IR refers to the Instruction Register in IF/ID
 - Bit ranges within sub-registers are identified with either sub-indexes or meaningful names
 - E.g., the opcode of the last instruction fetched resides in
 $\text{IF/ID.IR}_{31..26}$ or $\text{IF/ID.IR}_{[\text{opcode}]}$

IF stage



- Functions
 - Read new instruction from memory
 - Increment PC
- Control signals
 - I-mem is always read
 - PC is always updated (incremented by 4 or modified by branch)
 - So there is no need for calculating these control signals (always active)

ID stage



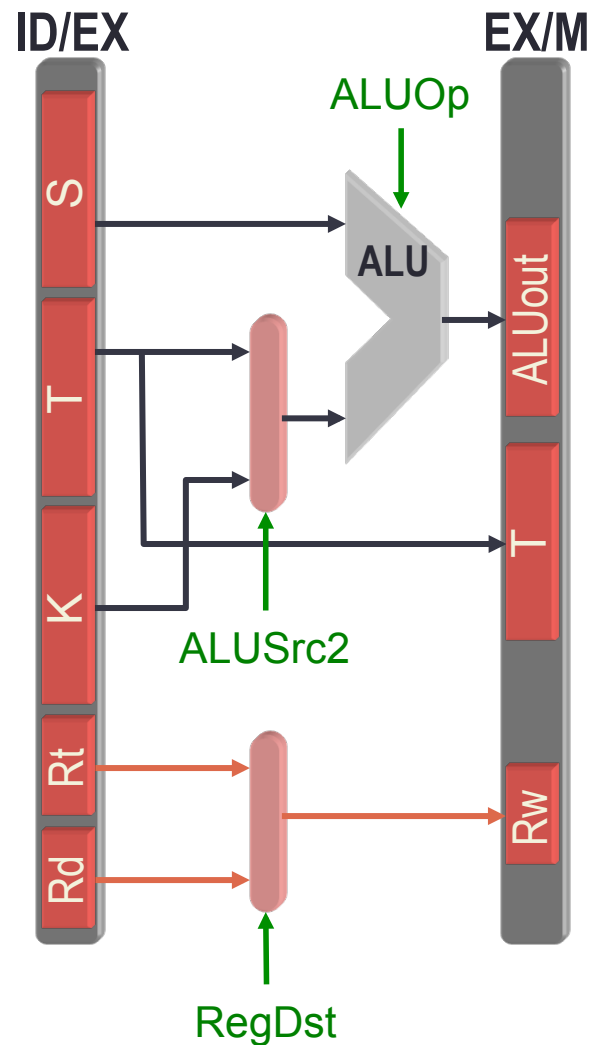
- Functions

- Decode instruction and calculate control signals for all remaining stages
- Read two registers
- Process I-format sign extension of offset/immediate

- Control signals

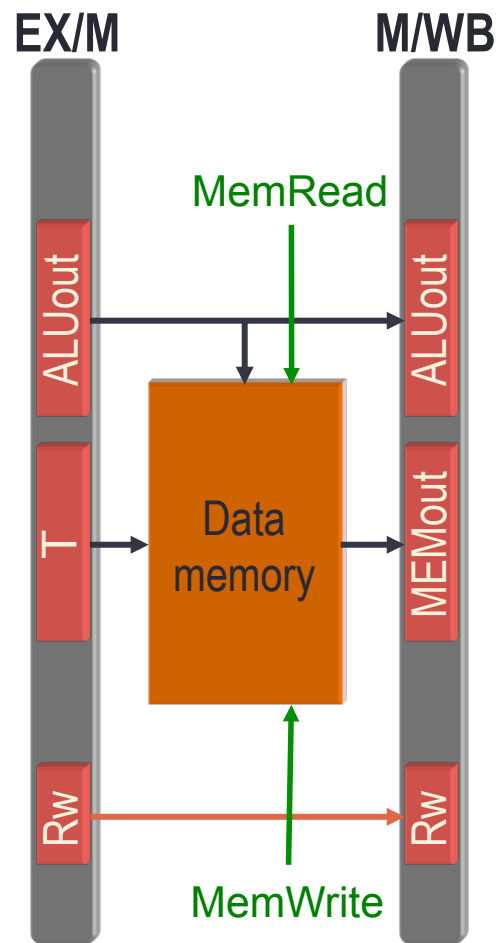
- Register file is always read, so there is no need for calculating this control signal (always active)

EX stage



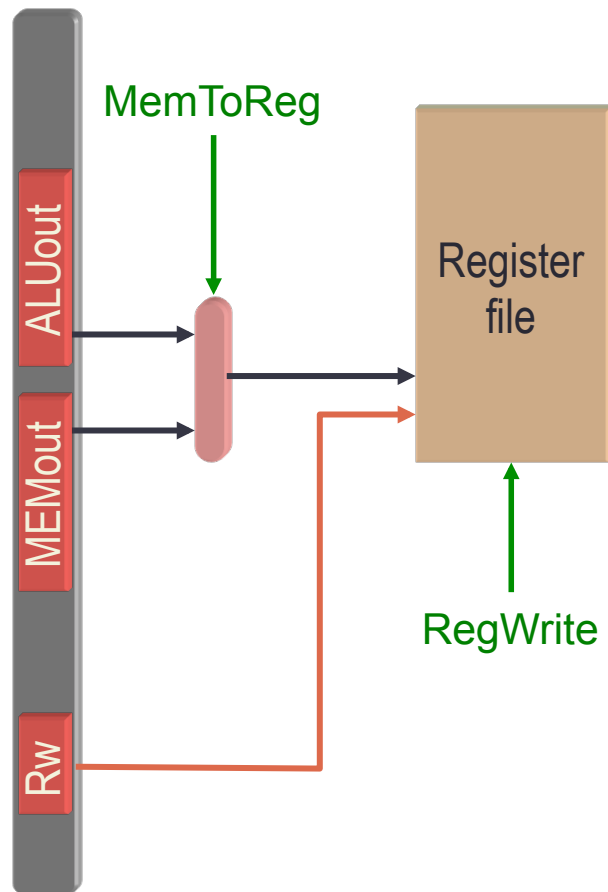
- Functions
 - A/L instructions: calculate result
 - Load/store: calculate address
- Control signals
 - **ALUOp**: selects the operation to perform in the ALU (3 bits)
 - **ALUSrc2**: selects 2nd operand for ALU
 - **RegDst**: selects the number of destination register to be potentially written

M stage



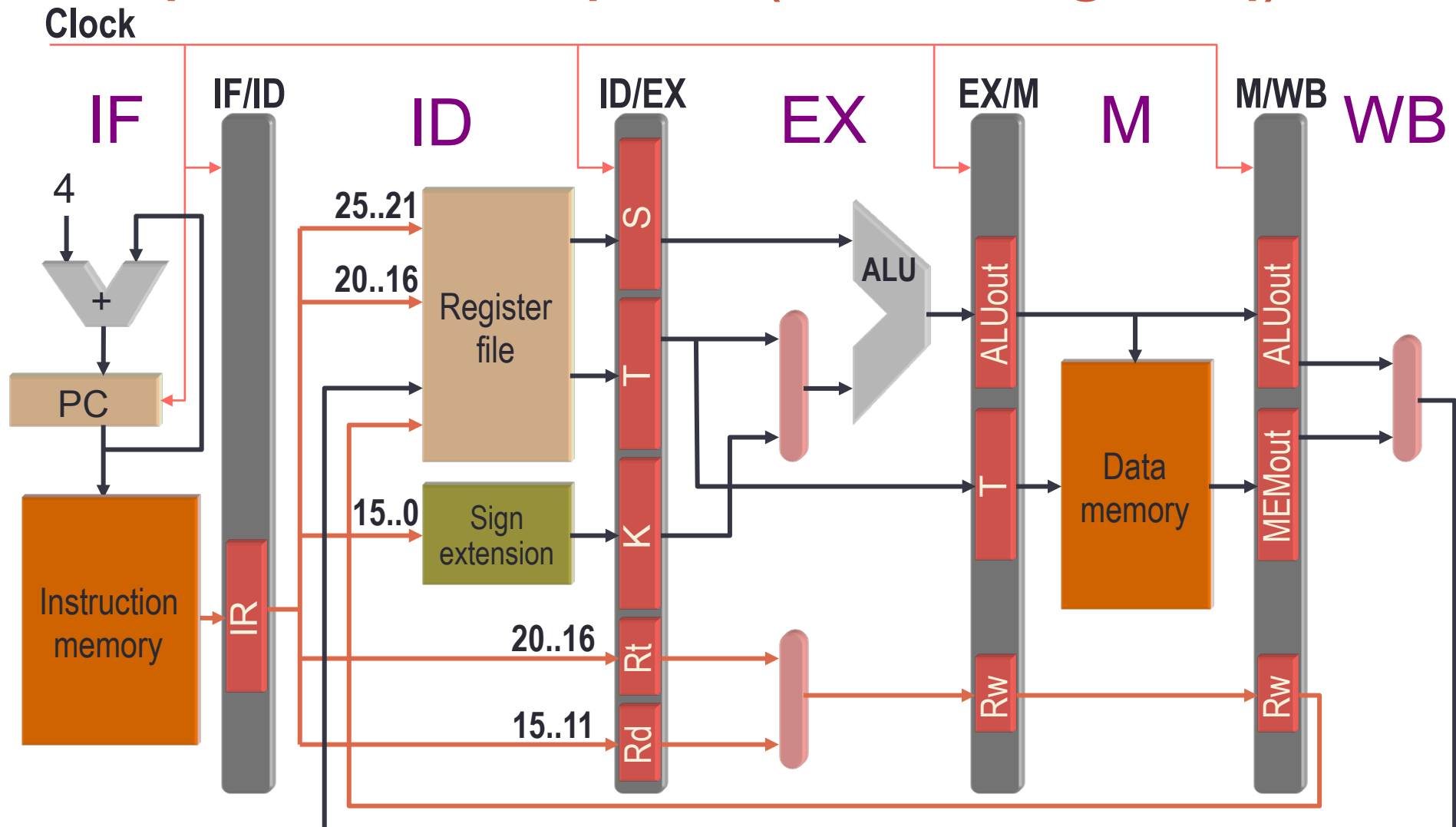
- Function
 - A/L instructions: nothing
 - Load: Read data from memory
 - Store: Write data to memory
- Control signals
 - MemRead
 - MemWrite

WB stage



- **Function**
 - A/L instructions: write result to destination register
 - Load: write data read to destination register
 - Store: nothing
- **Control signals**
 - **MemToReg**: selects memory or ALU output as data to be written
 - **RegWrite**: enables writing to the register file

Pipelined datapath (excluding beq)



Note about stages with memory

- With a memory hierarchy, memory accesses have a varying delay
 - Instruction- and data-memory are actually cache memories (unit 6)
 - A *cache hit* is a fast access
 - A *cache miss* imposes a larger and variable access time
- Delay for stages IF and M
 - We will assume the best case: access within one clock cycle
 - Adapting the pipelined processor to a memory hierarchy requires additional mechanisms

Selecting the clock frequency

- The clock cycle cannot be smaller than
 - $\tau = \max(\tau_i) + t_R$
- Example
 - Assume $\tau_{IF} = \tau_M = 15 \text{ ns}$; $\tau_{ID} = \tau_{WB} = 10 \text{ ns}$; $\tau_{EX} = 12 \text{ ns}$; $t_R = 5 \text{ ns}$
 - Clock period: $\tau = \max(15, 10, 12) + 5 = 20 \text{ ns}$
 - Clock frequency: $f_{CLK} = 1/20 \text{ ns} = 50 \text{ MHz}$
- What's the speedup when $n \rightarrow \infty$?
 - Hint: take the slowest instruction to calculate the delay of the non-pipelined version and relate it to the clock cycle of the pipelined version

Execution diagrams

instruction/time diagram

Inst\Cycle	1	2	3	4	5	6	7	8	9
lw \$1, 4(\$1)	IF	ID	EX	M	WB				
add \$3, \$2, \$0		IF	ID	EX	M	WB			
sub \$4, \$2, \$0			IF	ID	EX	M	WB		
and \$5, \$2, \$0				IF	ID	EX	M	WB	
or \$6, \$2, \$3					IF	ID	EX	M	WB

time/stage diagram

Cyc\Stg	IF	ID	EX	M	WB
1	lw	?	?	?	?
2	add	lw	?	?	?
3	sub	add	lw	?	?
4	and	sub	add	lw	?
5	or	and	sub	add	lw
6	?	or	and	sub	add
7	?	?	or	and	sub
8	?	?	?	or	and

Situation on cycle 5

Performance: CPI

- Cycles Per Instruction (CPI) is a common measure of average processor performance
- As the name suggests, it is the ratio between the number of cycles to execute a particular program and the number of instructions executed by that program
- The first instruction requires $K-1$ cycles to arrive at the last stage. We will subtract those “filling” cycles to calculate the CPI in the “steady” state (large n)
- In our case, $K=5$ hence

$$CPI = \frac{cycles - 4}{I}$$

Note: in section 4, we'll see why the CPI may go beyond 1

Performance

- A program's execution time can be calculated as

$$T = I \cdot CPI \cdot \tau$$

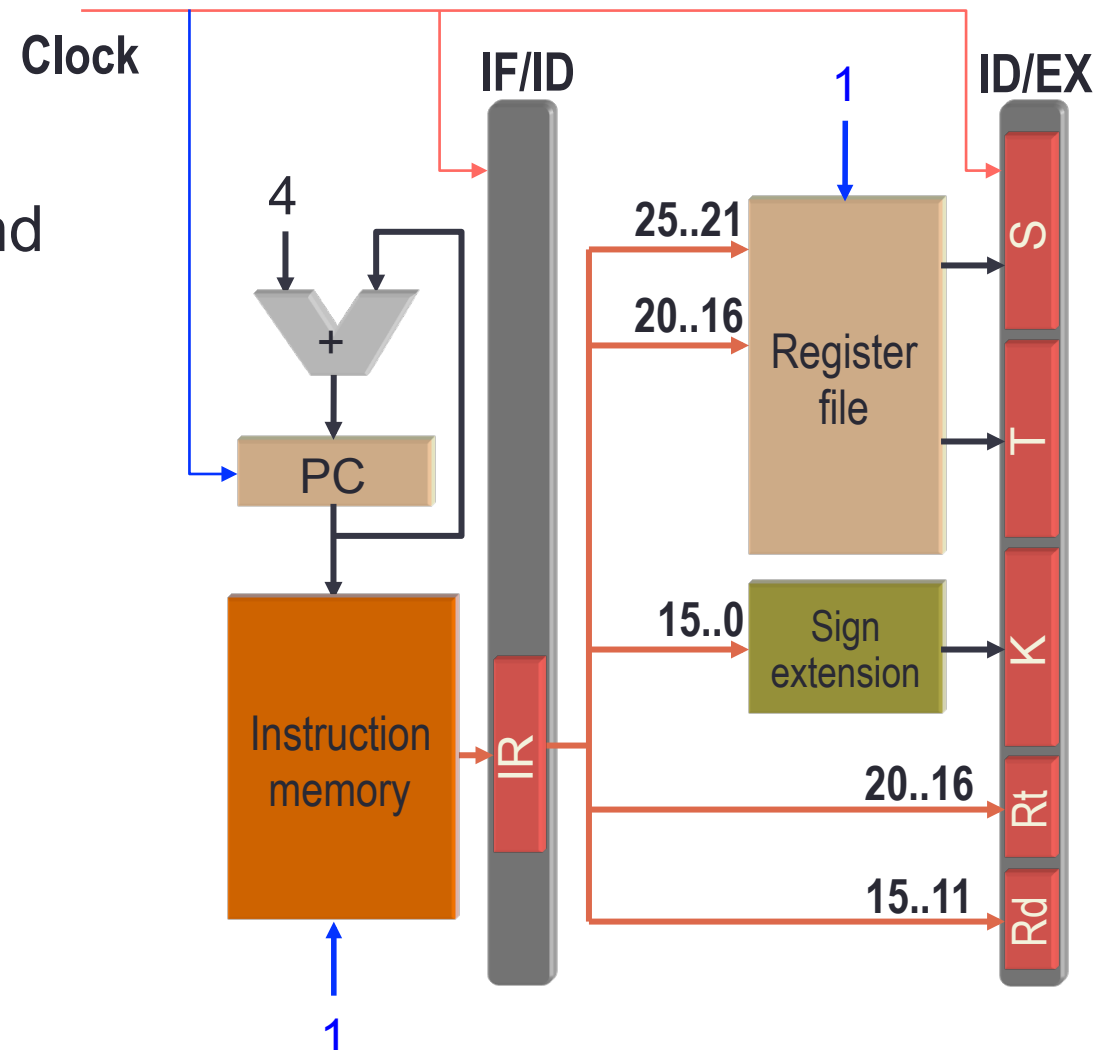
- Where:
 - T is the time taken to execute the program
 - I is the number of instructions executed
 - CPI is the average cycles per instruction
 - τ is the clock period
- In general we want to minimize I, CPI and τ
 - Here we'll focus on I and CPI

Pipelined control

- Goal: to have all stages properly collaborating in executing the instructions they are working for
- Observations
 - Stages work autonomously
 - IF and ID are processing non-decoded instructions
 - Hence their control signals remain constant for all cycles
 - Always read from I-mem, write PC, and read from the register file
 - ID is the stage in charge of calculating the control signals for the current instruction in ID, to be applied in later stages
 - These control signals need to traverse the datapath with the instruction
 - Control signals to apply in EX, M and WB are latched in ID/EX
 - Control signals to apply in M and WB are latched in EX/M
 - Control signals to apply in WB are latched in M/WB
 - Control signals will accompany instructions through all stages

Control for IF and ID

- These stages are independent of the instruction's opcode and parameters
- Always
 - Update PC
 - Read from I-mem
 - Read from register file

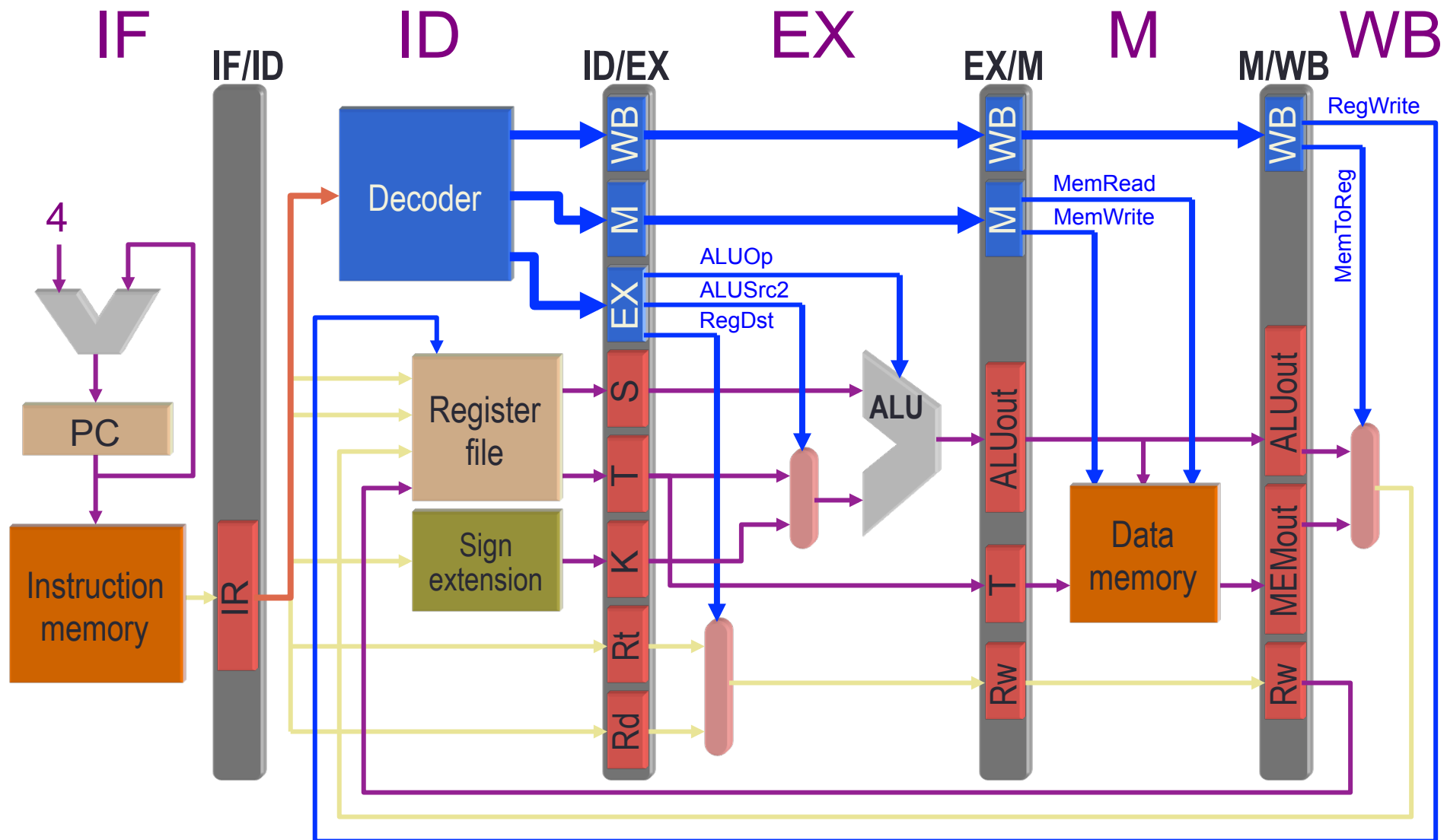


Control for EX, M and WB

- The control unit specification coincides with the control of the single-cycle datapath studied in unit 1

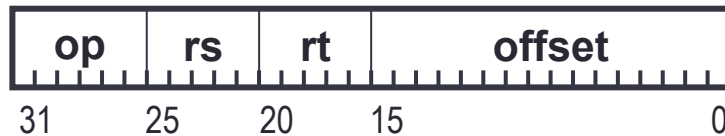
Contents of IF/ID.IR				Signals for EX			Signals for M		Signals for WB	
Instr.	Format	opcode	func (R)	ALUSrc2	ALUOp	RegDst	MemRead	MemWrite	MemToReg	RegWrite
add	R	000000	100000	0	0 1 0	1	0	0	0	1
addi	I	001000		1	0 1 0	0	0	0	0	1
sub	R	000000	100010	0	1 1 0	1	0	0	0	1
and	R	000000	100100	0	0 0 0	1	0	0	0	1
andi	I	001100		1	0 0 0	0	0	0	0	1
or	R	000000	100101	0	0 0 1	1	0	0	0	1
slt	R	000000	101010	0	1 1 1	1	0	0	0	1
lw	I	100011		1	0 1 0	0	1	0	1	1
sw	I	101011		1	0 1 0	X	0	1	X	0
beq	I	000100		0	1 1 0	X	0	0	X	0

Datapath and control (beq excluded)



Support for conditional branches

- MIPS R2000 includes six I-format conditional branch instructions



instruction	branch condition
beq <i>rs,rt,target</i>	$rs = rt$
bne <i>rs,rt,target</i>	$rs \neq rt$
bgez <i>rs,target</i>	$rs \geq 0$
bgtz <i>rs,target</i>	$rs > 0$
blez <i>rs,target</i>	$rs \leq 0$
bltz <i>rs,target</i>	$rs < 0$

- offset* is calculated by the assembler:

$$\text{offset} = (\text{target address} - (\text{PC} + 4)) / 4$$

Where “PC” is the address of the branch instruction itself

Implementation of conditional branch

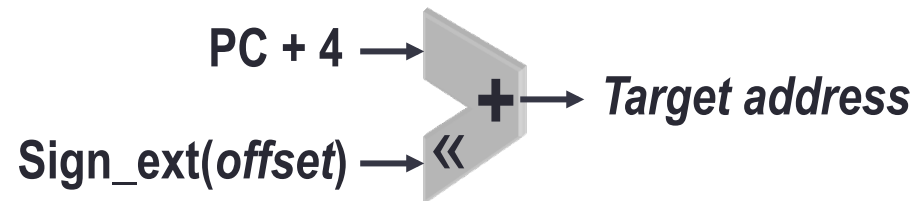
- Evaluating the condition
 - We will add an *identity* operation in the ALU for rs
 - Identity(A) \rightarrow ALUOut = A
 - Two ALU flags will cover all possible conditions
 - Z: Asserted when ALUOut = 0
 - S: Asserted when ALUOut < 0 (i.e., sign bit of ALUOut)

instruction	condition	ALU op.	COND
beq	a=b	sub	Z
bne	a≠b	sub	\overline{Z}
bgez	a≥0	identity	\overline{S}
bgtz	a>0	identity	$\overline{S} \cdot \overline{Z} = \overline{S + Z}$
blez	a≤0	identity	$S + Z$
bltz	a<0	identity	S

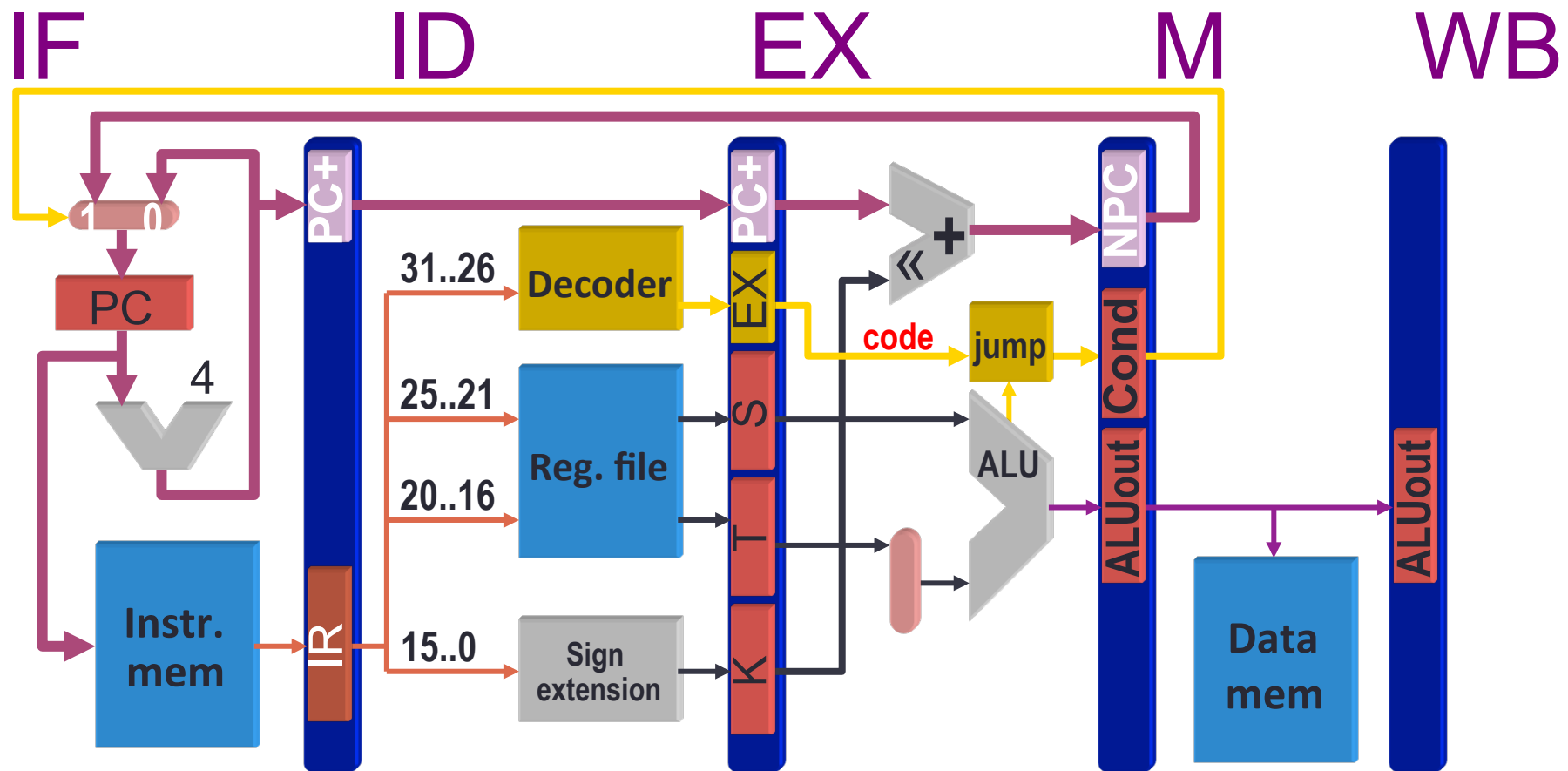
COND = 1 means the branch is effective

Implementation of conditional branch

- Datapath extensions to support branches
 - The branch condition can be evaluated in the ALU during EX
 - The target address will also be calculated during EX
 - For obtaining the target address we will use a new, dedicated adder, with one input shifted left two bits
 - Remember: Target address = $(PC+4) + \text{sign_ext}(\text{offset}) * 4$
 - If the branch condition holds, the new PC will be available at the beginning of the M stage, and enforced when the branch instruction enters the WB stage



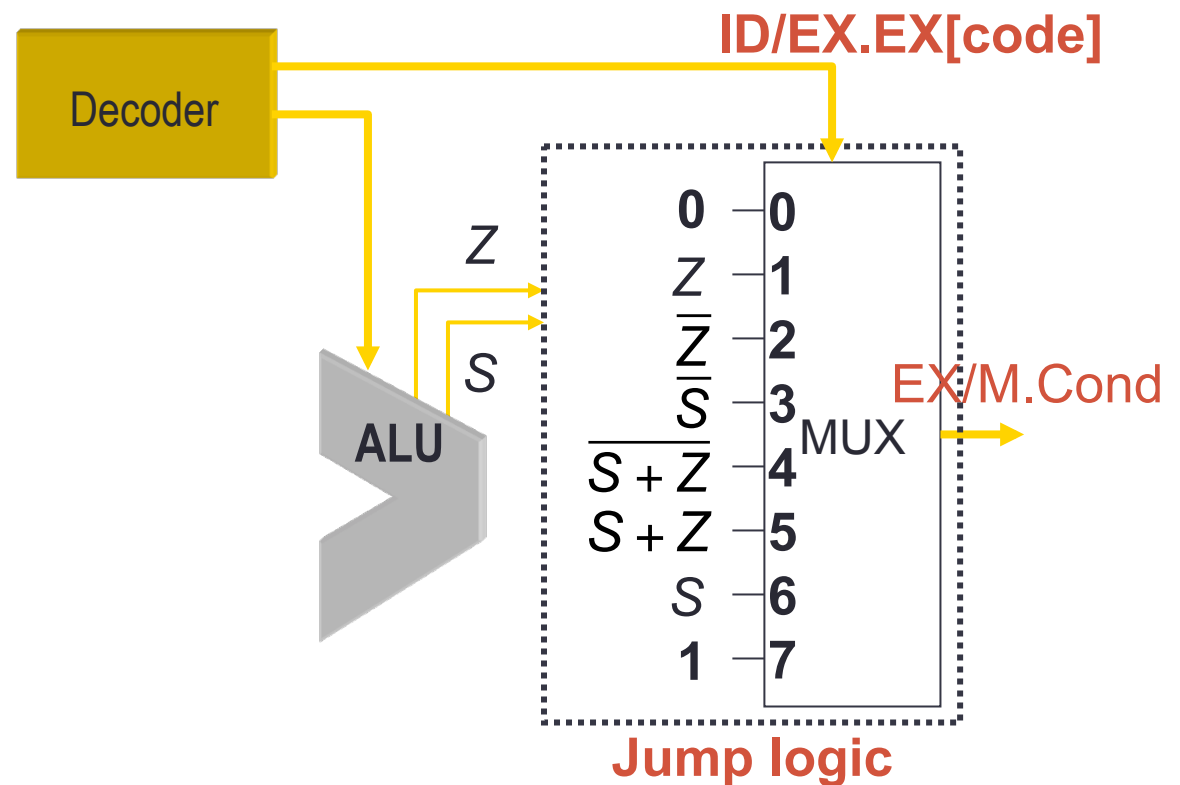
Implementation of conditional branch



Detail of the *jump* logic

- The EX/M.Cond bit indicates the PC source at next cycle
- A multiplexer and some additional logic can obtain this bit

Instruction	code
<i>A/L, l/s</i>	0
beq	1
bne	2
bgez	3
bgtz	4
blez	5
bltz	6
j	7

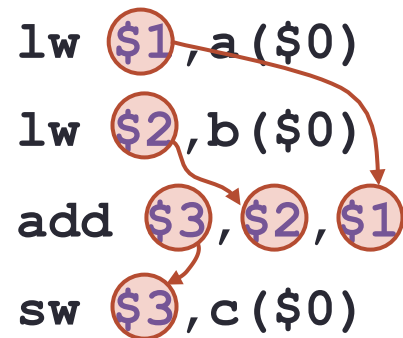


4. Data hazards

- Sometimes an instruction in the pipeline cannot execute in the following clock cycle(s). This situation is called a **hazard** and there are three possible causes for them:
 - **Structural hazards**: two instructions need the same resource at the same time
 - Laundry: a washer-dryer cannot be used for both functions at a time
 - Processor: a single memory for data and instructions cannot be accessed in IF and M at the same time
 - **Control hazards**: AKA branch hazards, occur in the presence of branch instructions, with an impact on the first stage(s) of the datapath
 - **Data hazards**: occur when one instruction needs the result from a previous one, and that result has not yet been written to the register file

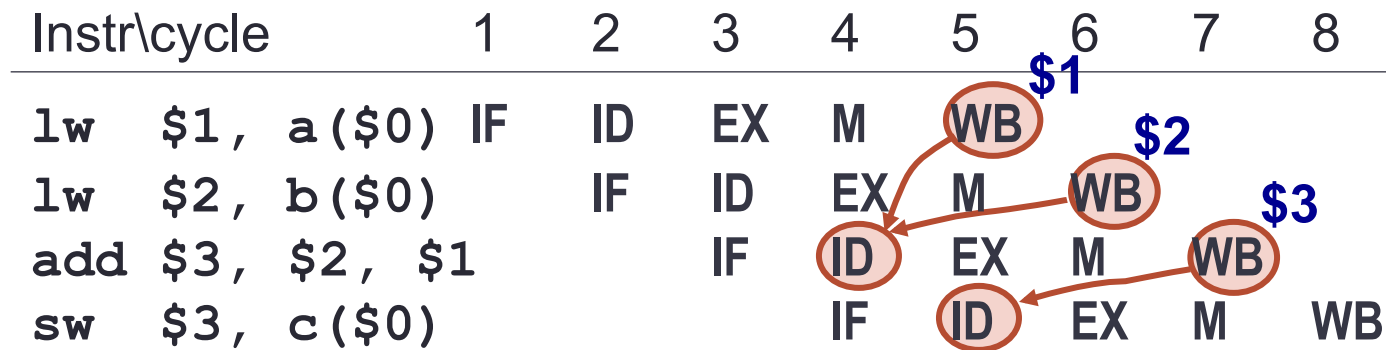
Data hazards

- Instructions need not be independent. Actually, they often show dependency from previous instructions
- Instructions can be seen as producers/consumers of data
- A data hazard occurs when one instruction consumes data produced by a previous, uncompleted instruction
 - For example: assembly of assignment $c \leftarrow a + b$



Data hazards

- Data hazards have an impact when an instruction needs to read data that is not yet written to the register file



- \$1 is written back by **lw** on cycle 5, but needed by **add** on cycle 4
- \$2 is written back by **lw** on cycle 6, but needed by **add** on cycle 4
- \$3 is written back by **add** on cycle 7, but needed by **sw** on cycle 5

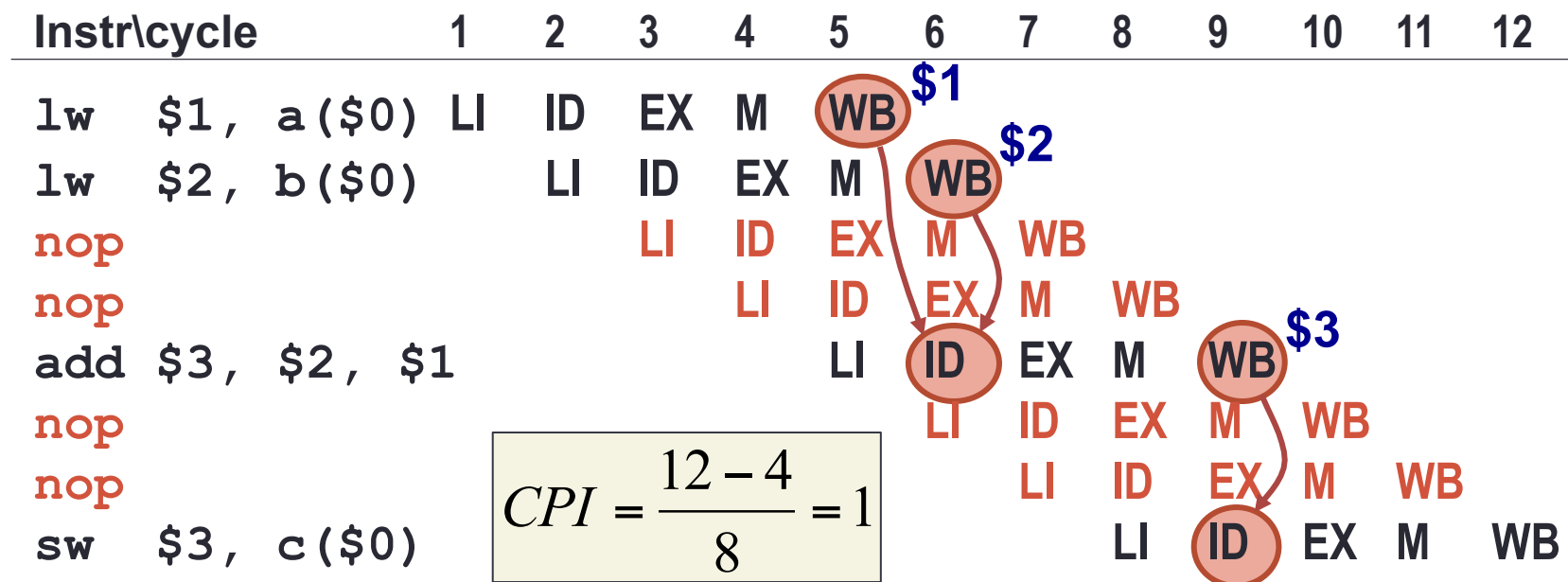
Solving techniques

- Basic techniques
 - Software: the compiler inserts **nop** instructions to separate dependencies – **nop** stands for *no operation*
 - Hardware: the control unit inserts **stall cycles** to give time for the producer to write the critical data
- Smart compilation*
 - Instruction reordering: the compiler reorders instructions to minimize data hazards
- Advanced techniques*
 - Forwarding, or bypassing – passes fresh data as soon as it is available somewhere in the datapath
 - Out-of-order execution – run-time instruction reordering

(*) *These techniques will be covered next year*

Insertion of `nop` instructions

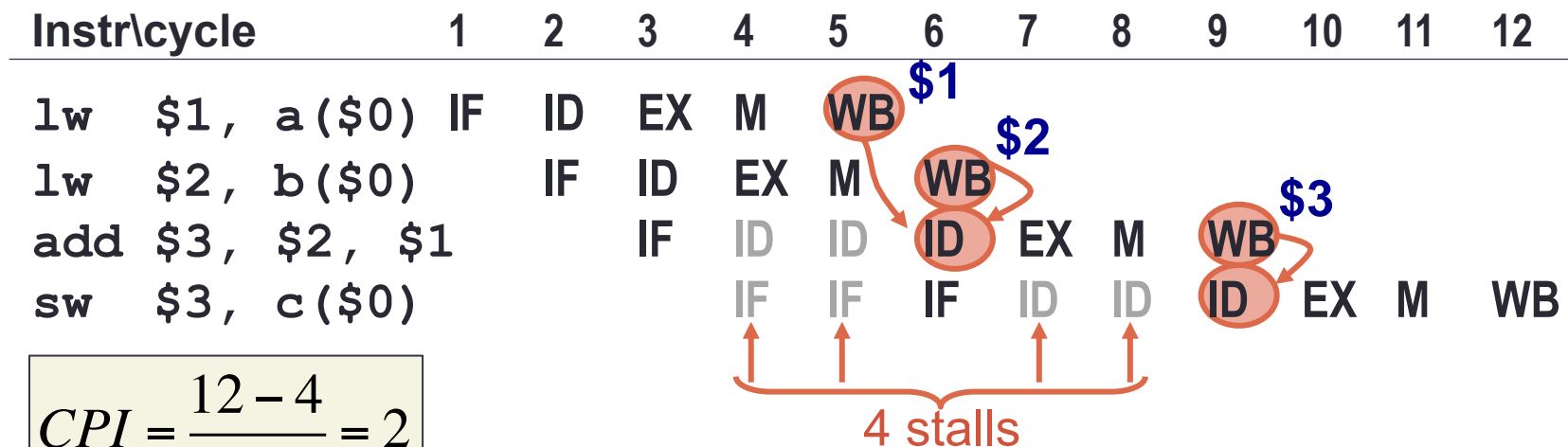
- The **compiler** detects hazards and inserts **`nop`** instructions to eliminate the conflicts
- In our previous example:



- Note that a register in the register file can be written and then read in the same cycle

Insertion of stall cycles

- The control unit can detect data hazards and stall the pipeline if needed



$$CPI = \frac{12 - 4}{4} = 2$$

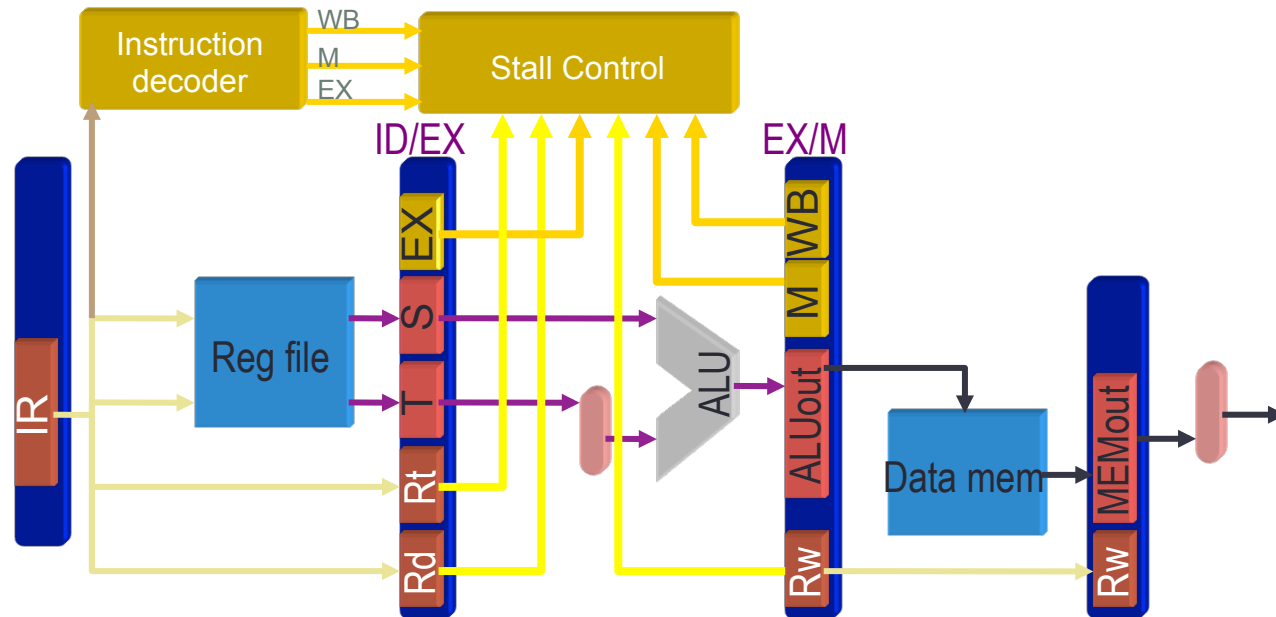
- Given P = number of stalls, the CPI can also be obtained as:

$$CPI = \frac{I + P}{I} = 1 + \frac{P}{I} = 1 + \frac{4}{4} = 2$$

Note that, in both cases, the program needs 12 cycles

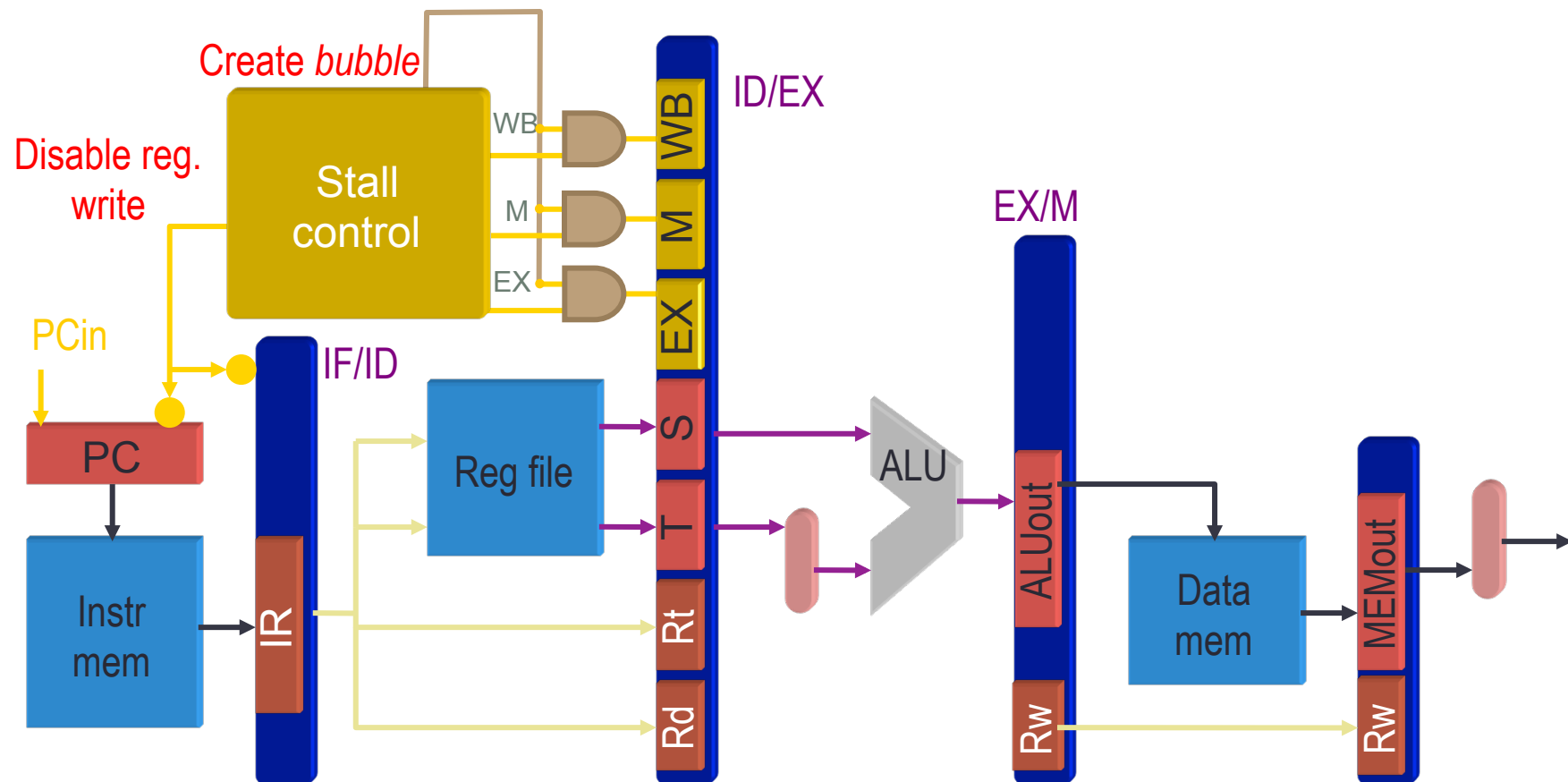
Insertion of stall cycles

- Detecting data hazard conditions (some examples)
 - $EX[MemRead]=0 \ \& \ EX[MemWrite]=0$ (*i.e., instruction in EX is not 'load' or 'store'*)
 - $ID/EX.Rd=IR.Rs + (ID/EX[ALUSrc2] = 1 \ \& \ ID/EX.Rt=IR.Rs)$
 - *In other words, R-format Instruction in EX writes to register Rs of instruction in ID OR I-type instruction in EX writes to register Rs of instruction in ID*
 - $ID/EX.Rd=IR.Rt + (ID/EX[ALUSrc2] = 1 \ \& \ ID/EX.Rt=IR.Rt)$
 - $EX/M.Rw=IR.Rs + (ID/EX[ALUSrc2] = 0 \ \& \ EX/M.Rw=IR.Rt)$



Insertion of stall cycles

- IF and ID must repeat instruction: *Disable update of PC and IR*
- Cancel effects of instruction in ID: *MemRead, MemWrite, RegWrite* $\leftarrow 0$



Evaluation: `nops` vs. stalls

- Similarities
 - In both cases, the number of cycles is the same (12 in the example)
 - A `nop` at assembly time corresponds to a stall at run time
- Differences
 - Inserting `nops` increases the number of instructions, `I`
 - Inserting stalls increases the average cycles per instruction, `CPI`
 - Inserting stalls provides binary compatibility of the pipelined processor vs. a non-pipelined implementation
 - In both cases, the processor executes the same code with identical results
 - But... the circuitry for detecting dependencies may have an impact on the minimum clock period
 - Hardware solutions need not always be faster than software solutions

Control hazards

- Control hazards are caused by branch instructions
 - Branches break the PC+4 assumption
- Branch instructions include
 - Unconditional branches (`j`, `jal` and `jr`)
 - Conditional branches (`beq`, `bne`, etc.)
- Statistically, branch instructions are 10~20% of the code
 - Sequential chunks of code are limited to 8~10 instructions, on average
- For simplicity, we'll focus on conditional branches
 - Note that `beq $0, $0, target` is, in effect, an unconditional branch to `target`

Branch instructions at work

```
if (x>y)
/*then*/ z=x
else
    z=y;
```



```
if:      lw $t0,x($0)
         lw $t1,y($0)
         sub $t2,$t1,$t0
         bgez $t2,else
then:    sw $t0,z($0)
         j endif
else:    sw $t1,z($0)
endif:
```

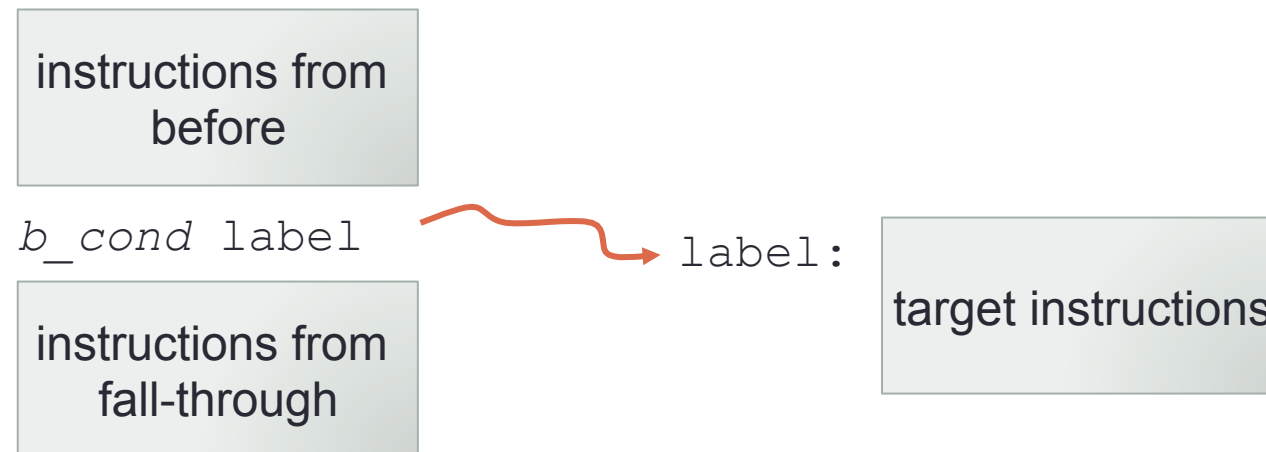
```
z=0;
do      z=z+y;
        x=x-1;
while  (x!=0)
```



```
add $t2,$0,$0
lw $t0,x($0)
lw $t1,y($0)
do:    add $t2,$t2,$t1
        addi $t0,$t0,-1
while: bne $t0,$0,do
enddw: sw $t2,z($0)
```

Branch vocabulary

- Target instruction: the instruction at the branch target address
- When the branch condition holds, the branch is taken. Otherwise we say the branch is not taken
- A conditional branch relates three kinds of instructions:



Analysis of control hazards

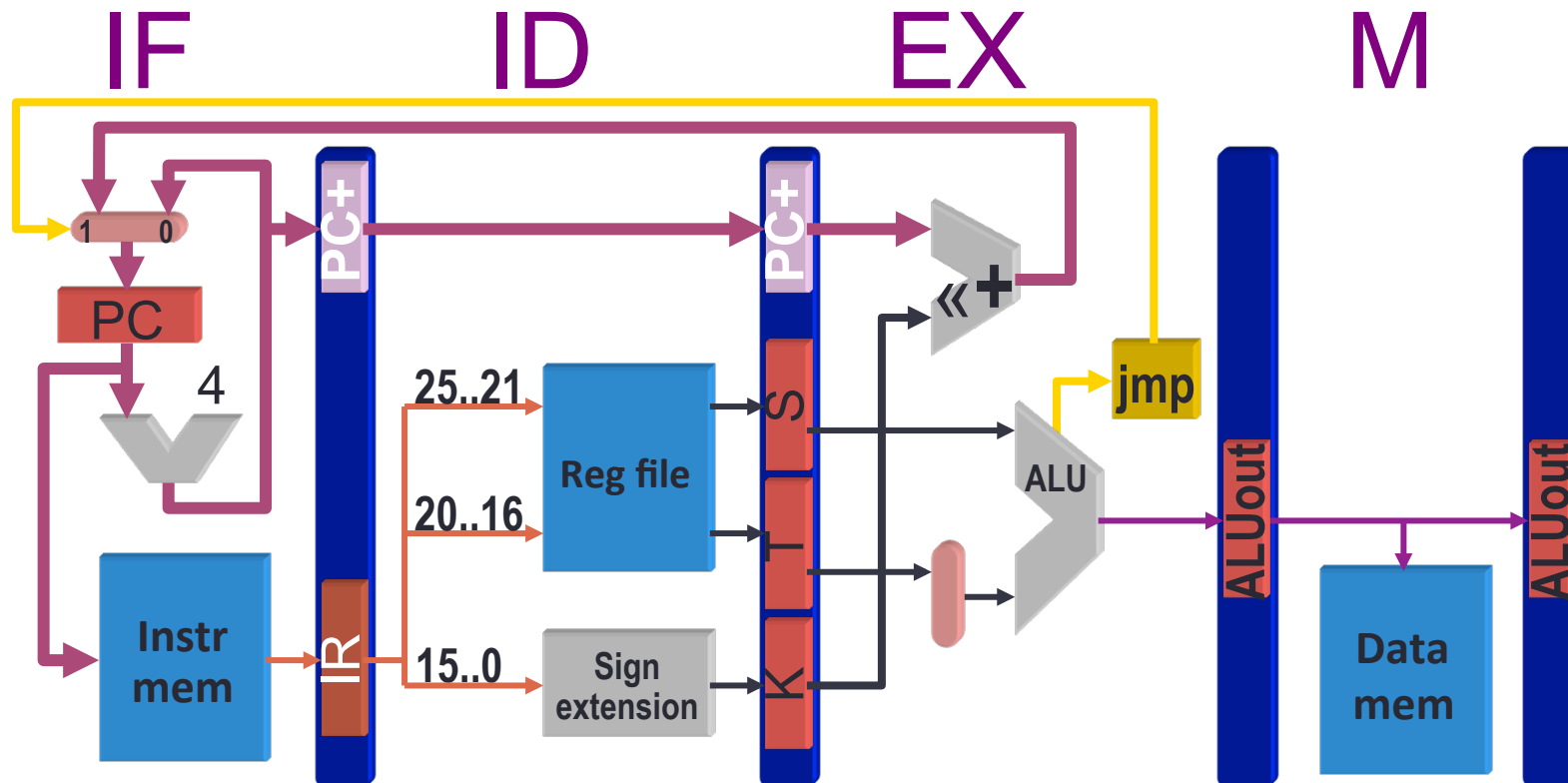
- The target PC is only updated when branch is in WB
- Three *wrong* instructions are in the pipeline
 - The **branch latency** is said to be = 3

Instr\cycle	1	2	3	4	5	6	7	8	9
previous	IF	ID	EX	M	WB				
branch		IF	ID	EX	M	WB			
next 1			IF	ID	EX	M	WB		
next 2				IF	ID	EX	M	WB	
next 3					IF	ID	EX	M	WB
target						IF	ID	EX	M

Cycle\stage	IF	ID	EX	M	WB
4	next 2	next 1	branch	previous	?
5	next 3	next 2	next 1	branch	previous
6	target	next 3	next 2	next 1	branch

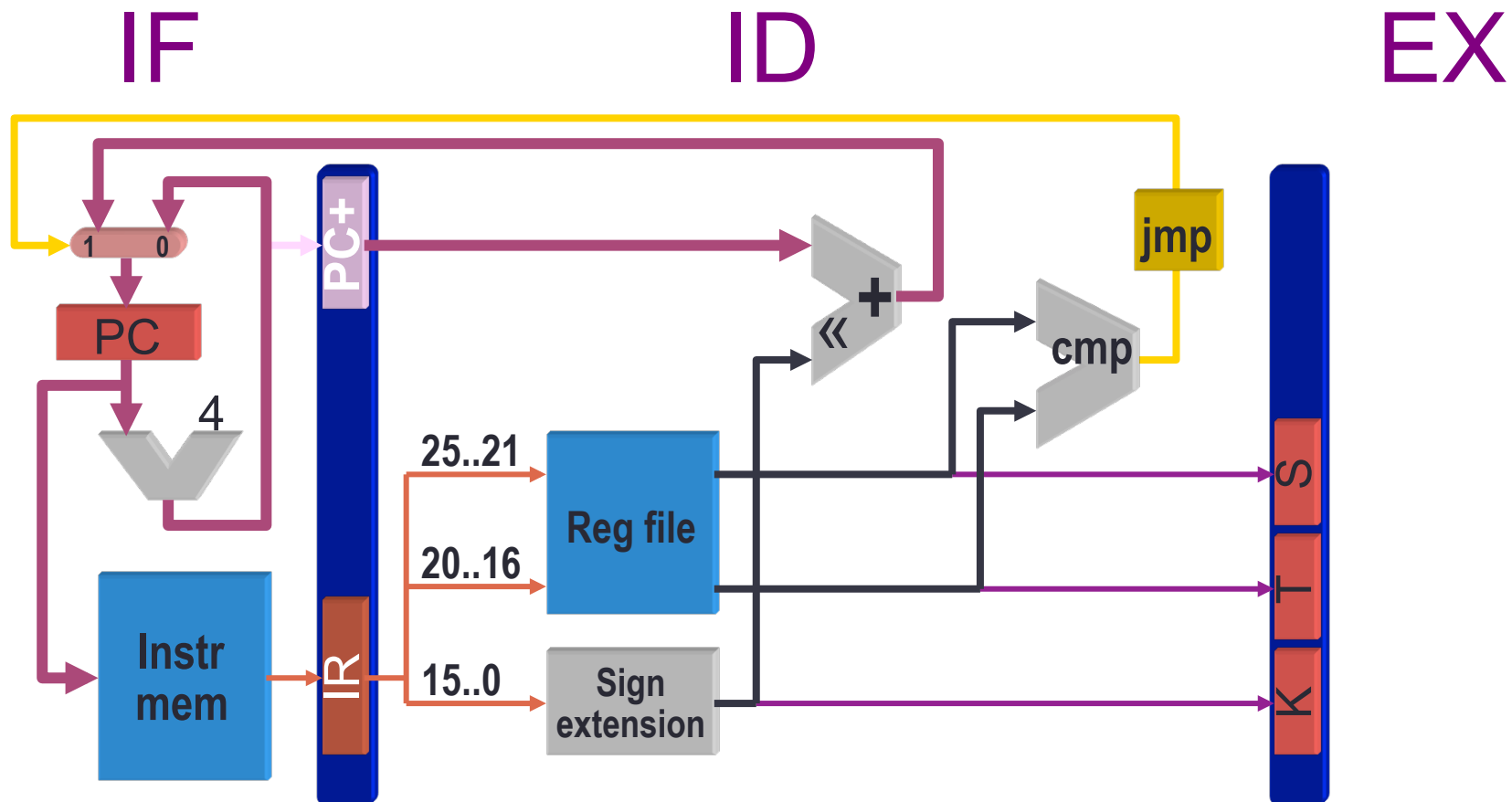
Reducing branch latency

- The earlier the PC is updated, the shorter the branch latency
- A datapath with latency = 2 (compare with page 32)



Reducing branch latency

- A datapath with latency = 1



Reducing branch latency: analysis

- With latency = 2
 - Increased time for EX, which may affect the minimum clock period
- With latency = 1
 - Largely increased time for ID
 - The jump logic depends on control signals

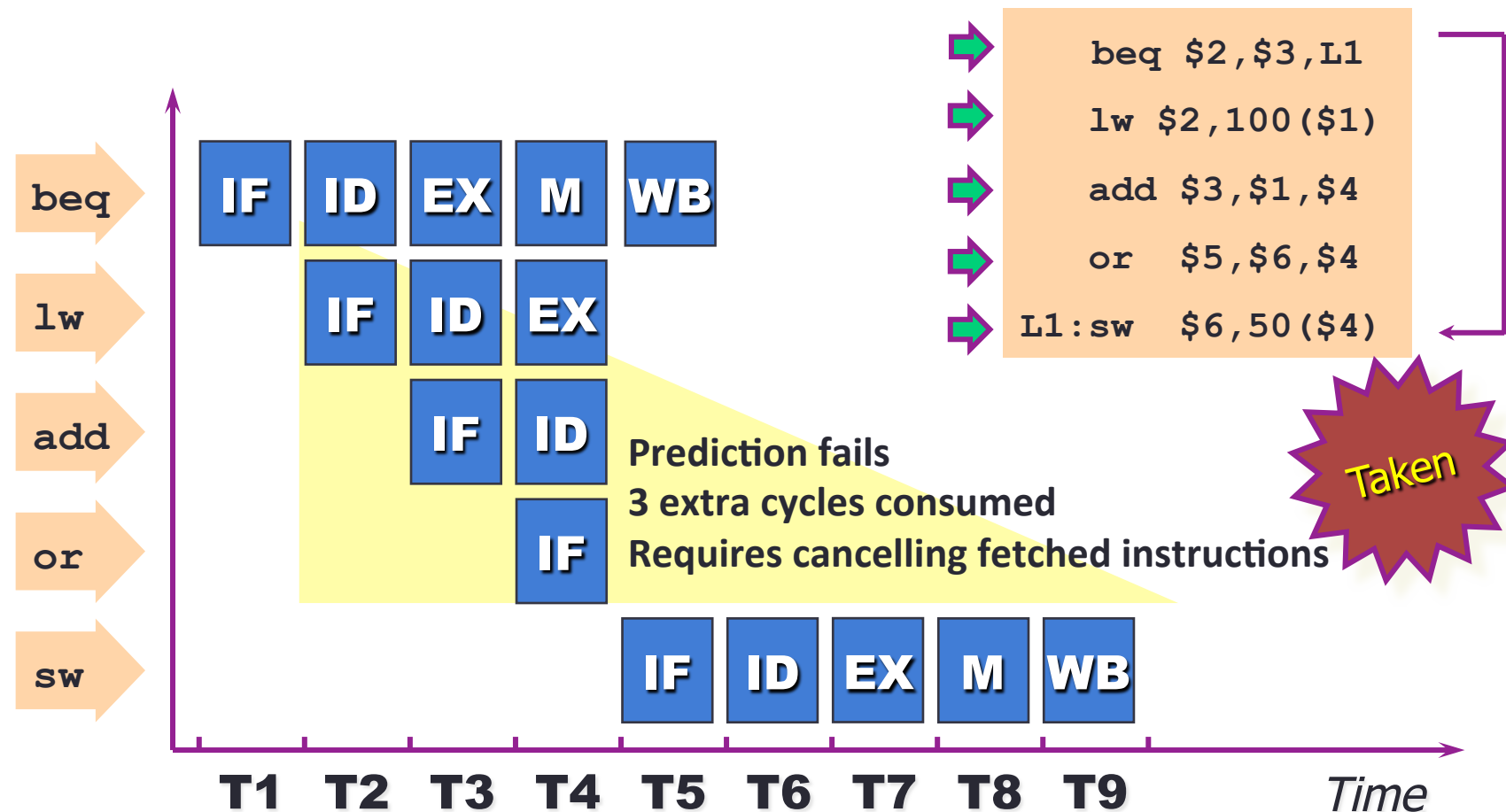
Solving control hazards

- Solving control hazards means avoiding the fetching of wrong instructions, or cancelling their effects
- *Emergency* solutions
 - Hardware – the control unit inserts stalls after branch instructions
 - Software – the assembler inserts nop instructions after branches
- More advanced (higher performance) solutions
 - Fixed branch prediction
 - Predict not-taken (eg, early SPARC and MIPS, i486)
 - Predict taken (eg, Sun SuperSparc)
 - *Static branch prediction
 - Backward taken, forward not-taken (eg, HP PA-7X00)
 - *Dynamic prediction, based on the instruction's recent behaviour
 - 1-bit (eg, AMD K5, Alpha 21064)
 - 2-bit (eg, MIPS R10000, PowerPC 604)
 - *Delayed branch: always execute instructions in the branch delay slot

(*) Next year

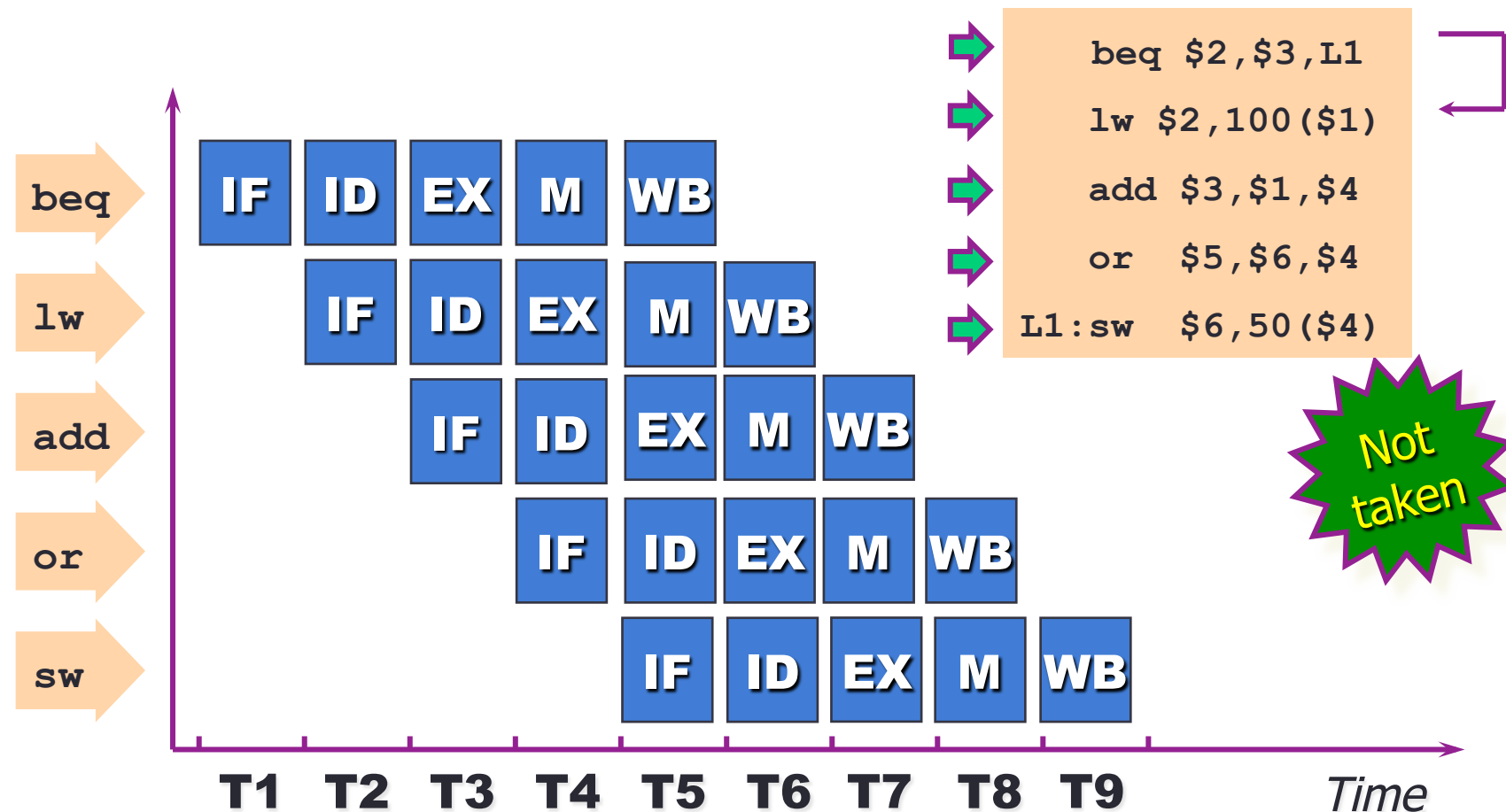
Fixed branch prediction – *predict not-taken*

- Hardware always assumes branch is not taken
 - Failed prediction:



Fixed branch prediction – *predict not-taken*

- Hardware always assumes branch is not taken
 - Correct prediction:

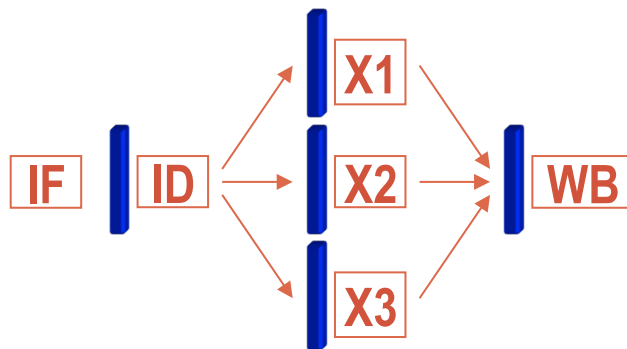


Beyond pipelining

- The four/five-stage model is the basis for most modern processors (except x86 Intel family)
 - The Intel 80486 is said to be pipelined in 5 stages, but quite differently
- Some examples are:
 - MIPS R2000
 - Sun Sparc V7
 - Hewlett Packard – Precision Architecture (HP-PA)
- Further refinements include
 - Multi-cycle stages
 - Superpipelining
 - Superscalar processors
 - Prediction, multiple instruction issue, instruction scheduling, speculative execution... (next year!)

Multi-cycle stages

- Large stages may be split into several smaller stages to favour balance in the execution of *heavy* instructions
- Units have different latencies (L) and repetition rates (R), eg.:
 - Memory ($L \geq 1$, $R = 1$); Integer arithmetic ($L = 1$, $R = 1$), FP arithmetic ($L > 1$, $R \geq 1$)



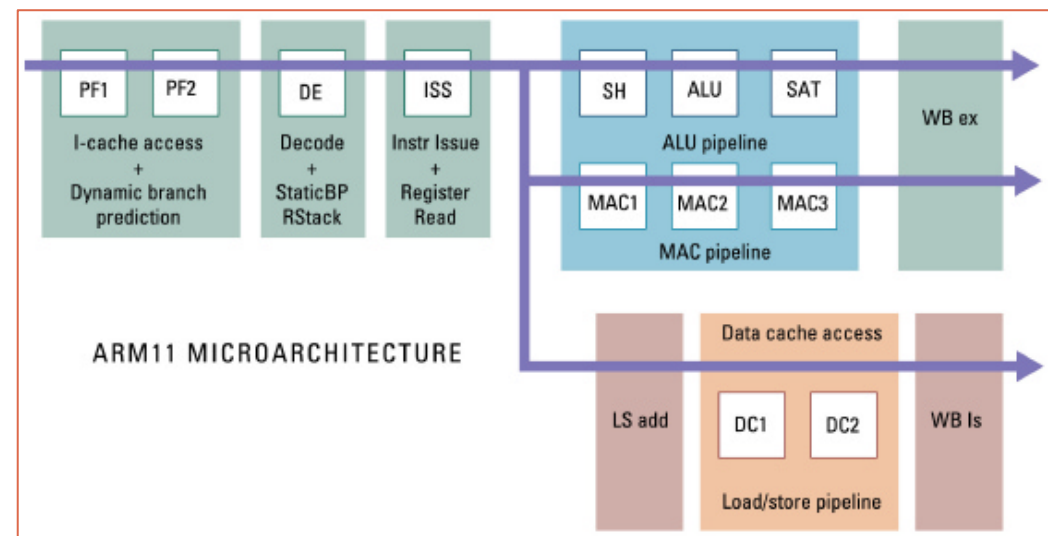
Unit	L	R
X1 (mem)	2	1
X2 (integer)	1	1
X3 (FP)	3	3

- An *issue logic* forwards instructions to appropriate units
- Instructions may not complete in the order they were fetched
- Structural conflicts may arise – eg. two FP instructions in a row

Instr\cycle	1	2	3	4	5	6	7
lw	IF	ID	X1	X1	WB		
add.s (FP)		IF	ID	X3	X3	X3	WB
add (integer)			IF	ID	X2	WB	

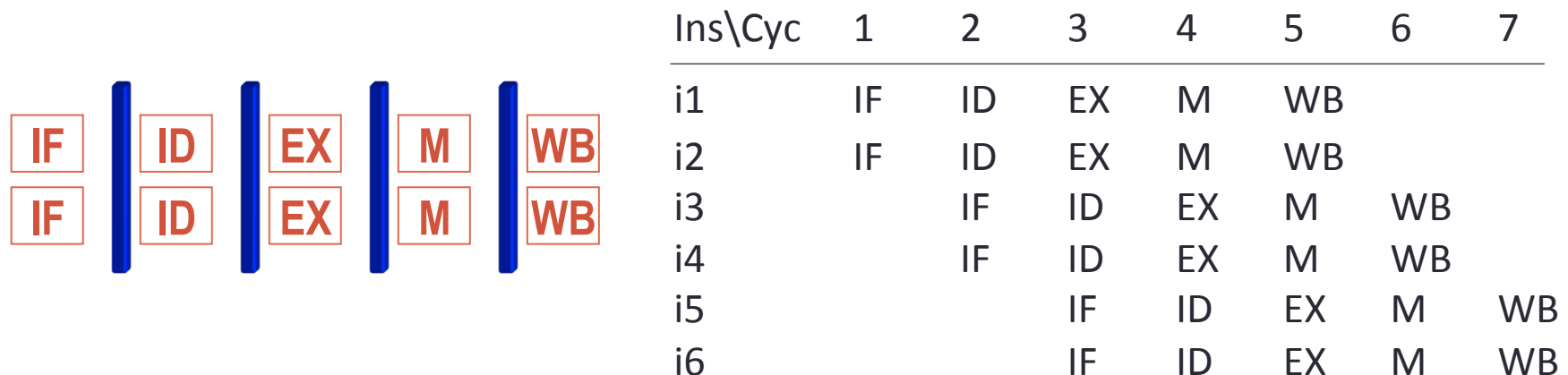
Superpipelining

- The term refers to processors that include *more stages than usual* – one *basic function* split into several stages
- The goal is to increase the clock frequency (shorter stages) at a CPI close to 1
- For example,
 - MIPS R4000: IF1, IF2, ID, EX, M1, M2, M3, WB
 - Alpha 21064 (7 stages)
 - ARM11 (8 stages)



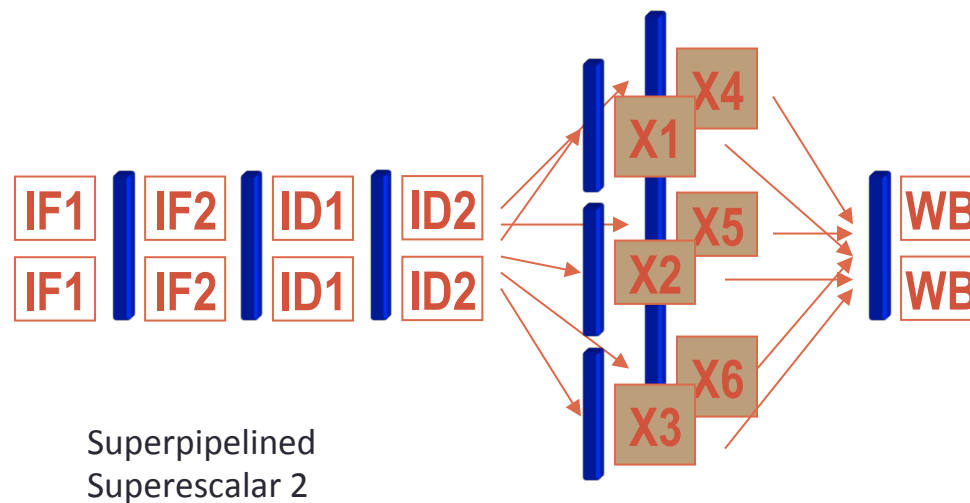
Superscalar processors

- Superscalarity is about increasing parallelism
 - Each stage can be working on more than one instruction
- The goal is to reduce the CPI, while maintaining the clock frequency
 - Instead of CPI, the *Instructions Per Cycle* indicator (IPC) is used (IPC = 1/CPI)
- Example with n=2 (max CPI = 0.5, IPC = 2)



All together...

- Current processors use combinations of these techniques
- Parameters:
 - Number of stages
 - Scalarity: scalar (1), superscalar n
 - Operators and their parameters (L,R)



Unit	L	R
X1 (l/s mem)	3	1
X2 (+/- int)	1	1
X3 (x/÷ int)	4	3
X4 (x/÷ FP)	5	3
X5 (+/- FP)	15	10
X6 (multimedia)	3	1

Historical perspective – CISC vs. RISC

CISC: Complex Instruction Set Computer	RISC: Reduced Instruction Set Computer
Large, complex instruction set	Reduced, simpler instruction set
Many instruction formats with varying instruction size	Few formats, fixed instruction size
Many, complex addressing modes	Few, simple addressing modes
Few registers	Many registers
Many instructions have memory operands	Load-store architecture
Complex, microprogrammed control unit	Simpler, hardwired control unit
Goal: complex instructions, reduce <i>semantic gap</i>	Goal: execute instructions in a single clock cycle
Less complex compilers	More complex compilers
Less amenable to pipelining	More amenable to pipelining
Typical architecture in the 70's-80's	Born in the 80's
Memory was relatively expensive	Memory became cheaper

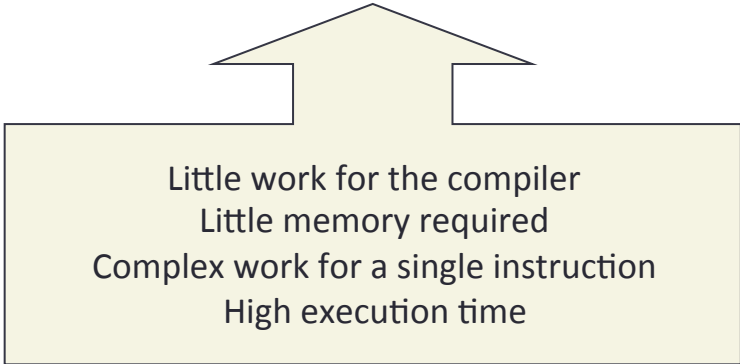
There's no clear difference between RISC and CISC in today's processors. For example, Intel processors are often referred to as CISC-in RISC-out

Historical perspective – CISC vs. RISC

- Example: multiply two variables in memory, leaving result in memory
 - Addr1: address for operand 1 and result
 - Addr2: address for operand 2

CISC

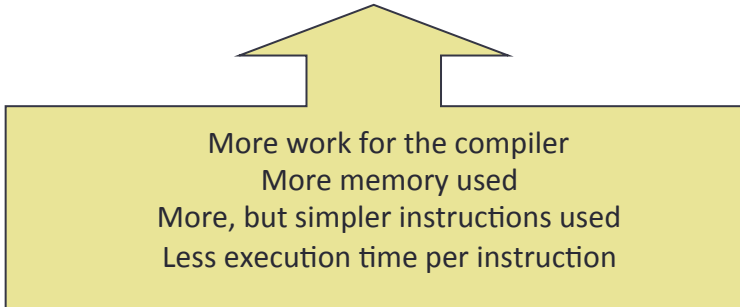
```
MULT Addr1, Addr2
```



Little work for the compiler
Little memory required
Complex work for a single instruction
High execution time

RISC

```
la $t0, dir1  
la $t1, dir2  
lw $t3, 0($t0)  
lw $t4, 0($t1)  
mul $t3, $t4  
mflo $t5  
sw $t5, 0($t0)
```



More work for the compiler
More memory used
More, but simpler instructions used
Less execution time per instruction

Historical perspective – CISC vs. RISC

- Studies carried out in the 80's revealed:
 - CISC processors used 80% of the time to execute a subset of 20% of their instructions
 - Some instructions were never executed, others very few times
- By reducing the instruction set and unifying formats
 - Instructions execute faster
 - Control units become simpler, faster and take less chip area
 - The gained area can then be used for more cache and registers
 - Efficient pipelining is more feasible
- Advances in compiler technology make it possible to efficiently exploit RISC designs