PIPELINING
basics

- A pipelined architecture for MIPS
- Hurdles in pipelining
- Simple solutions to pipelining hurdles
- Advanced pipelining
- Conclusive remarks

MIPS pipelined architecture

- MIPS simplified architecture can be realized by having each instruction execute in a single clock cycle, approximately as long as the 5 clocks required to complete the 5 phases. Why would this approach be unconvenient?
- We already know one reason:
  the longer cycle would waste time in all instructions that take less to execute (fewer than 5 clocks).

MIPS pipelined architecture

- There is another relevant reason:
- By breaking down into more phases (clock cycles) the execution of each instruction, it is possible to (partially) overlap the execution of more instructions.
- At eack clock cycle, while a section of the datapath takes care of an instruction, *another* section can be used to execute *another* instruction.

MIPS pipelined architecture

- If we start a new instruction at each new clock cycle, each of the 5 *phases* of the multi-cycle MIPS architecture becomes a *stage* in the pipeline, and the pattern of execution of a sequence of instructions looks like this (Hennessy-Patterson, Fig. A.1):

instr. number	1	2	3	4	5	6	7	8	9
instr. i	IF	ID	EX	MEM	WB				
instr. i+1		IF	ID	EX	MEM	WB			
instr. i+2			IF	ID	EX	MEM	WB		
instr. i+3				IF	ID	EX	MEM	WB	
instr. i+4					IF	ID	EX	MEM	WB

MIPS pipelined architecture

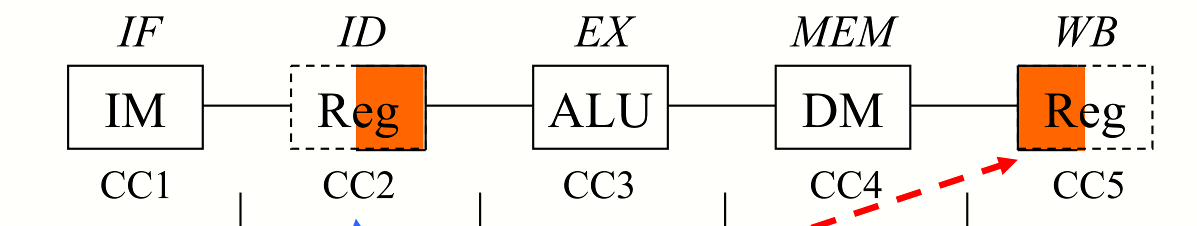
- Pipelining only works is one does not attempt to execute *at the same time* two different operations that use the same datapath resource:
  - as an instance, if the datapath has a single ALU, this cannot compute concurrently the effective address of a load and the subtraction of the operands in another instruction
- Using *reduced* (simple) *instructions* (namely RISC) makes it fairly easy to determine at each time which datapath resources are free and which are busy.
- This is very important to best exploit CPU resources (expecially, when deploying multiple issue: launching more instructions in parallel)

MIPS pipelined architecture

- Why are phases WB and ID highlighted in clock cycle 5? What is peculiar with them?
- Let instruction *i* be a load, and *i+3* be an R-type: in cycle 5 the result from instruction *i* is being written in the register file and the operands for instruction *i+3* are being read from the same file.
- Well, it looks like we are trying to use the *same* resource in the *same* clock cycle for two *different* instructions …

MIPS pipelined architecture

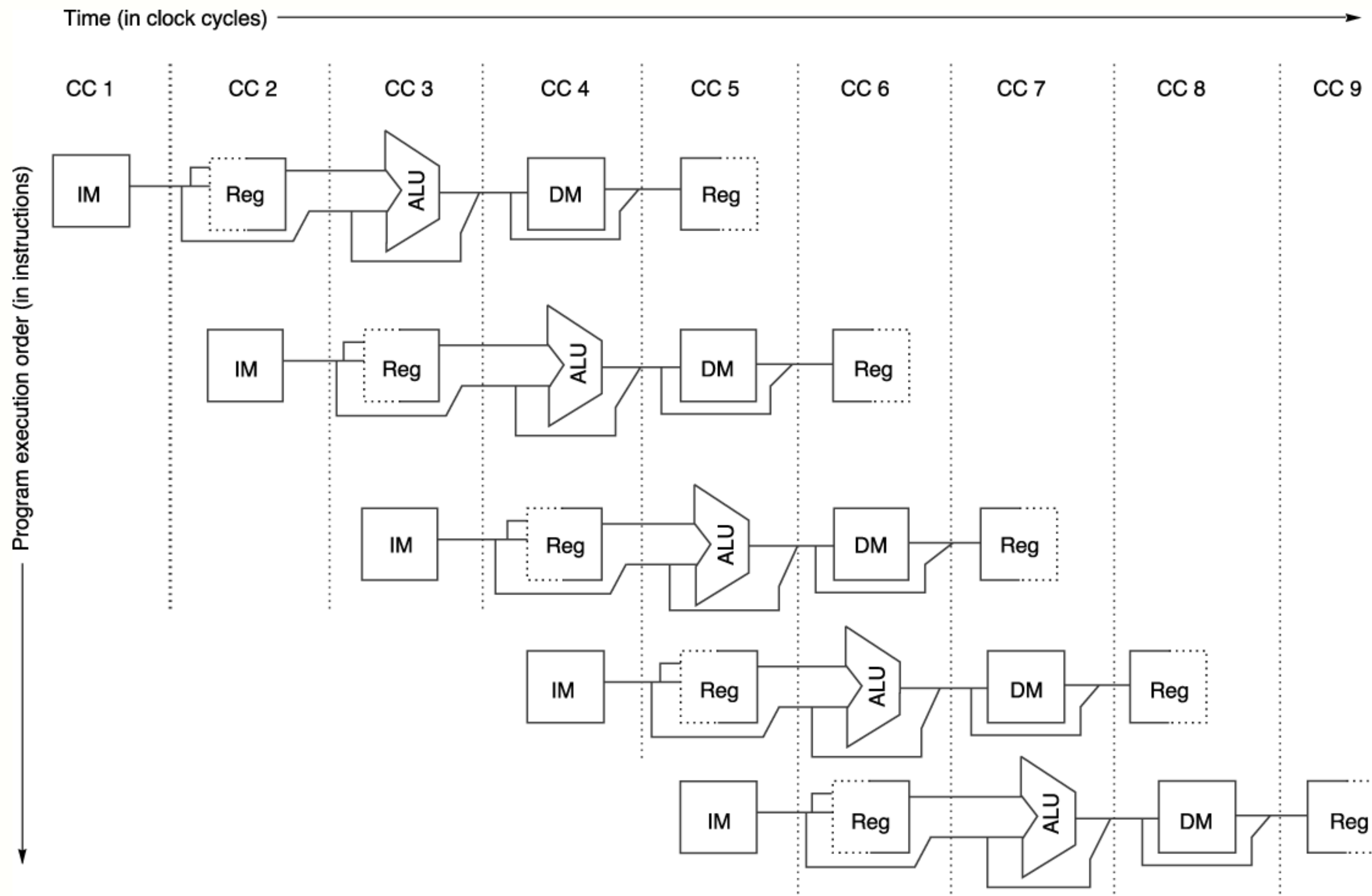
- Actually, this is allowed at those functional units that can be used twice during different phases of the same clock cycle.
- In the following figure depicting the execution pattern of a load instruction, each phase is associated to the unit is uses: the dotted part shows that the unit is not used in that portion of the clock cycle (adapted from Patterson-Henessy, fig. 6.4):



- The register file is written in WB in the first half of the clock cycle, and it is read in ID in the second half of the clock cycle.

MIPS pipelined architecture

The MIPS pipeline can be though of as a series of datapaths shifted in time, each one for each instruction being executed. Please, note the graphic notation (dotted lines) for the register file usage. (Hennessy-Patterson, Fig. A.2):



MIPS pipelined architecture

- The pipeline drawing of the previous chart calls for two observations:

  1. It is mandatory that there exist distinct memory units (caches) for data (DM) and instructions (IM); this prevents conflicts between IF and MEM phases. Both phases access memory, that is used 5 times more than with no pipelining. The memory subsystem must be capable of handling the increased workload.

  2. The drawing does not show the PC, that is incremented at each clock cycle in the IF phase (unless the instruction itself updates the PC in its execution phase, as is the case with JUMP and BRANCH instructions).

Pipeline registers

- In actual implementations, each pipeline stage is separate and connected to the following one by special **pipeline registers** (invisible to the user of the CPU, namely the program in execution).
- At the end of each clock cycle, the results of the operations in a stage are stored in a pipeline register that will be used as a source by the next stage in the next clock cycle.
- These pipeline registers also buffer intermediate results to be used by non-adjacent stages of the pipeline (more details on this later)

Pipeline registers

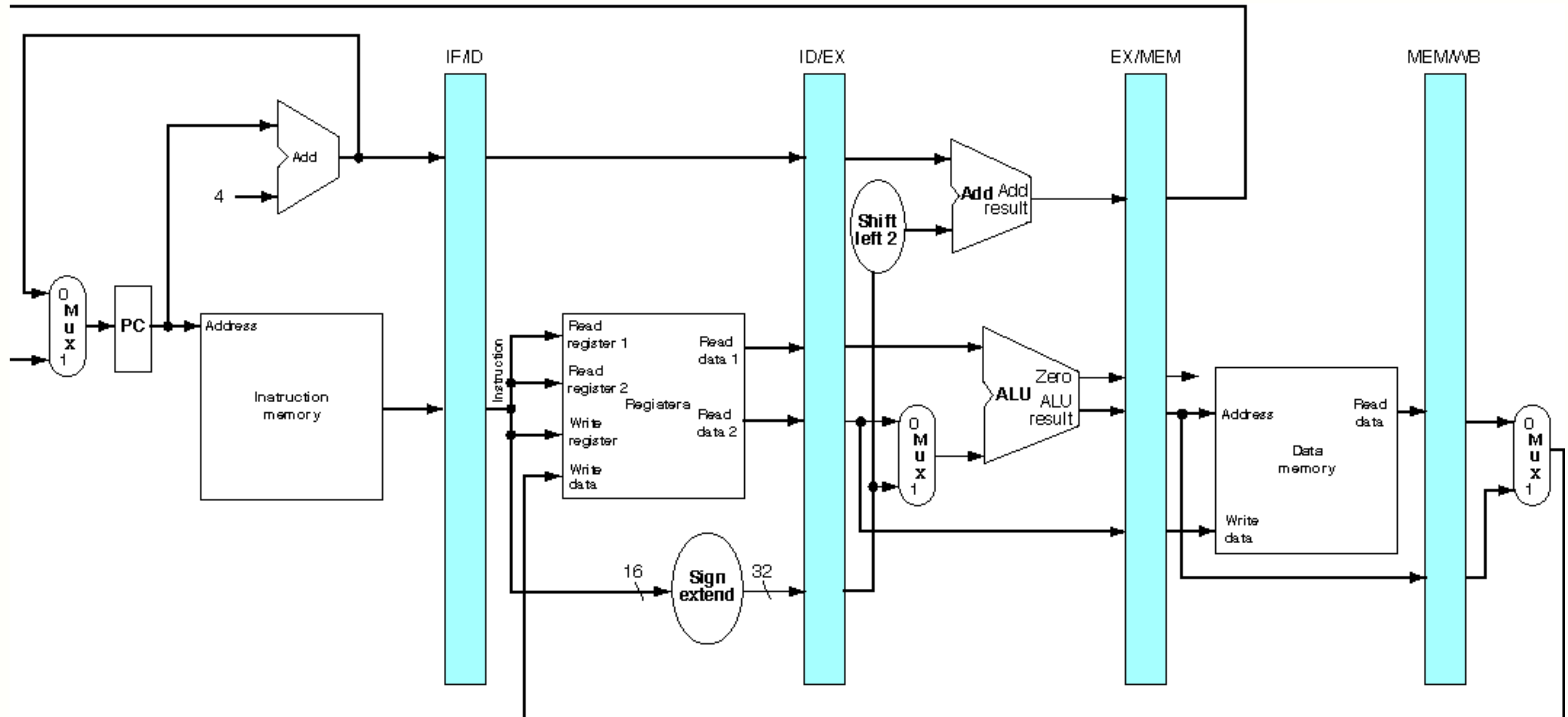
- The pipeline registers have the same role (and actually embed) of the *internal* registers introduced to transform the single-cycle MIPS architecture into the multi-cycle versione: IR, A, B ALUoutput, and MDR.
- They serve the purpose of transferring outputs produced in a phase to the subsequent phase in the multi-cycle implementation.
- An assembly line offers a simple analogy: each worker in a stage completes the work on the current piece, then places it in a *bin*, to be collected by the next worker in the line.

Pipeline registers

- We shall refer to the pipeline registers that are set between two stages with the names of the stages. So, we will have IF/ID, ID/EX, EX/MEM and MEM/WB registers.

- These registers must be large enough to contain all data moving from one phase to the following one (Patterson-Hennessy,fig. 6.11, see next chart. See also Hennessy-Patterson, Fig. A.18).

Pipelined MIPS Architecture

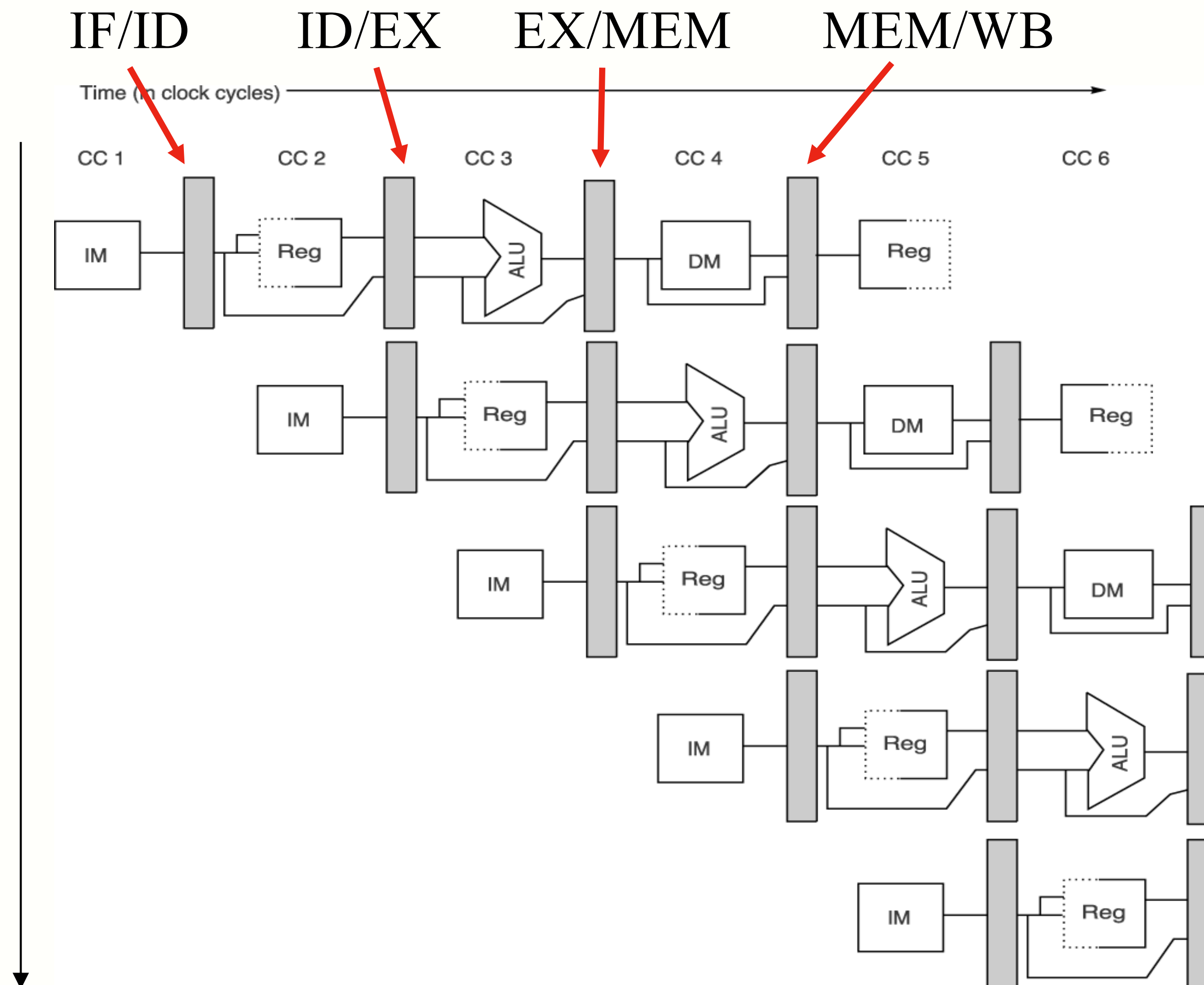


Pipelined MIPS Architecture

- Notably, there is no pipeline register after the WB phase, that is when the result is being written into its final destination.
- Indeed, at the end of this stage all instructions must update some part of the ISA visible processor state: the register file, memory (which one ?) or the PC.
- As an instance, a R-type instruction modifies a register, and the value available in the output register can be stored directly in the destination register specified in the instruction, to be used by subsequent instructions (we will come back on this naive hypothesis in the sequel…)

Pipeline registers

The pipeline datapath with pipeline registers between successive stages (Hennessy-Patterson, fig. A.3):



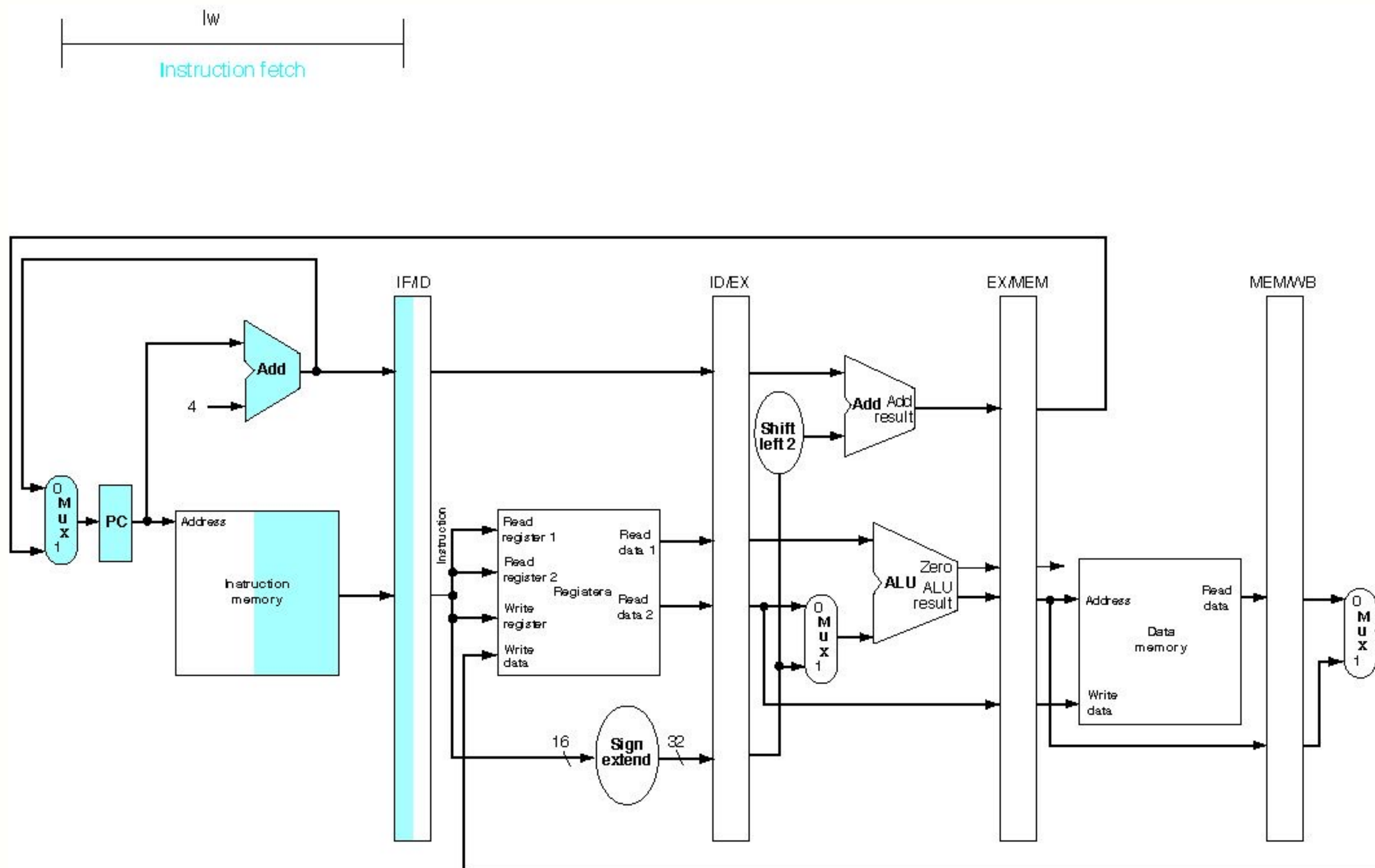
Program execution order of the (instructions)

Instruction flow within the pipelined MIPS

- Let us explain the pipeline operation by tracking the flow of a LOAD instruction.
- In the drawings, blocks representing memory and registers (both pipeline and register file ones) are blue highlighted in their right half when they are read, and in their left half when they are written.
- The blue highlight indicates active functional units in each phase.

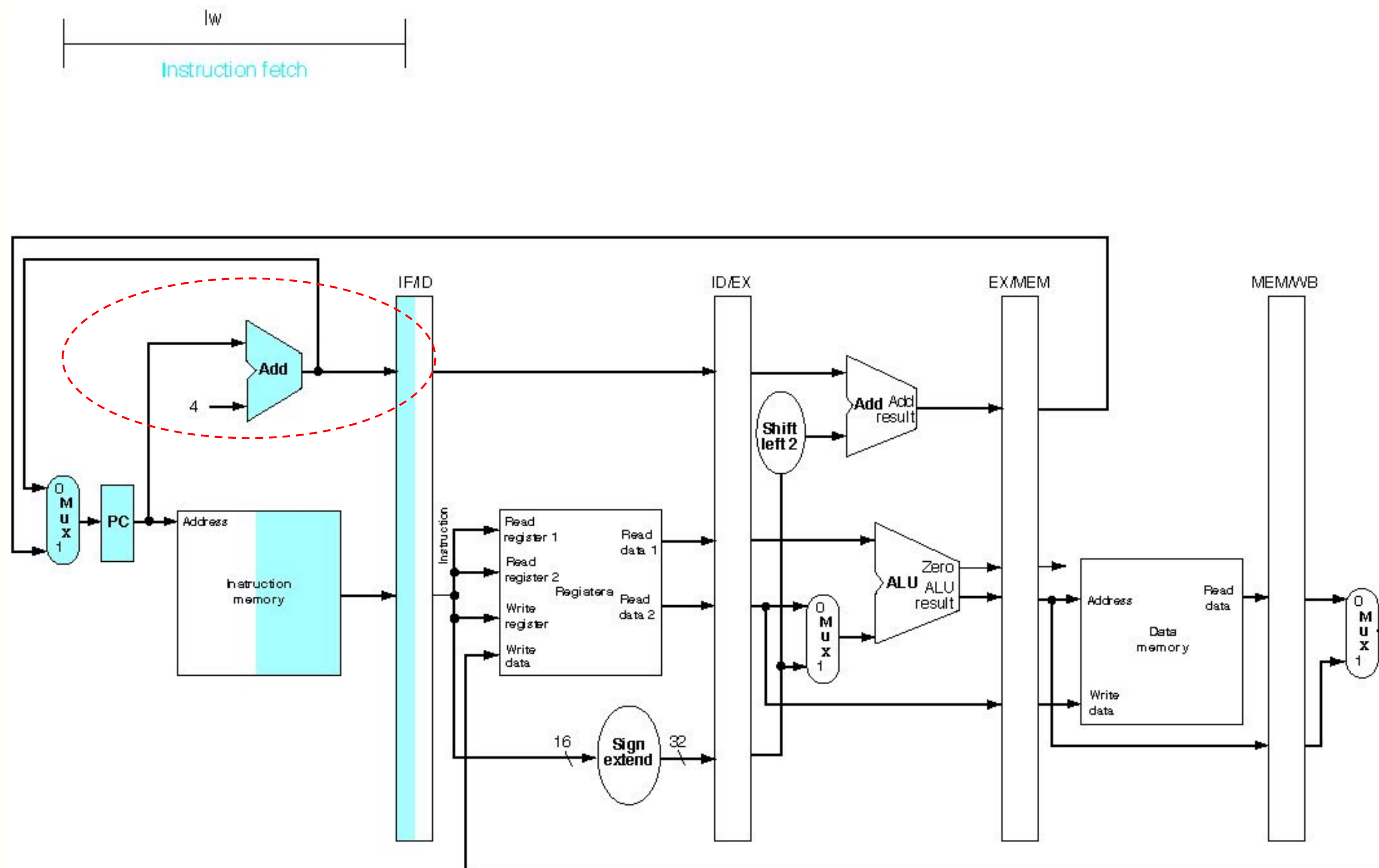
Instruction flow within the pipelined MIPS

- **Instruction Fetch:** the instruction is read from the Instruction Memory at the address in the PC, and it is placed into IF/ID. (Patterson-Hennessy, fig. 6.12a)



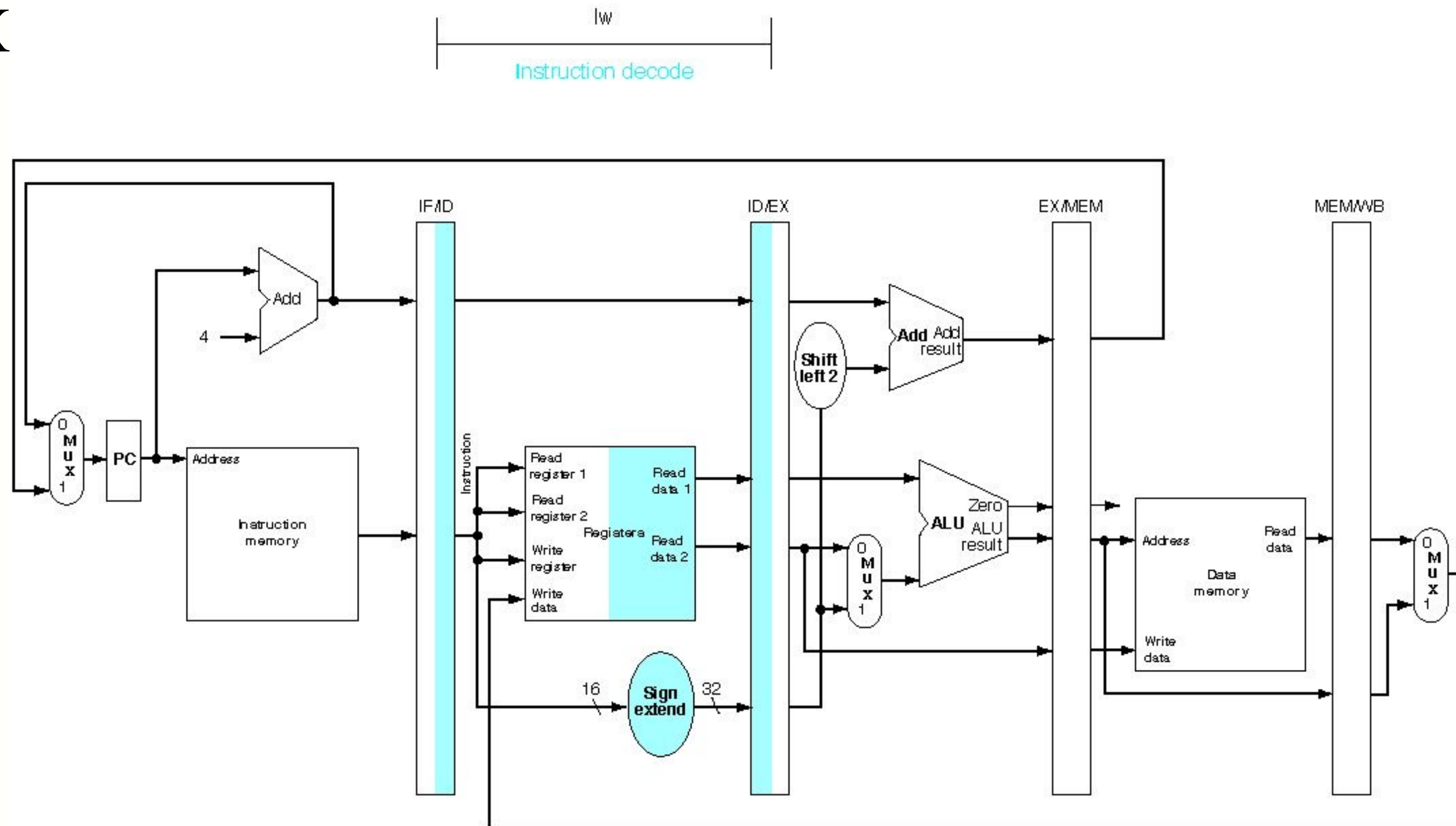
Instruction flow within the pipelined MIPS

- **Instruction Fetch:** The PC is incremented and written back, but is is also stored in IF/ID, since it could be used in a later phase (at this point, the instruction to be executed is not decoded nor fully known).



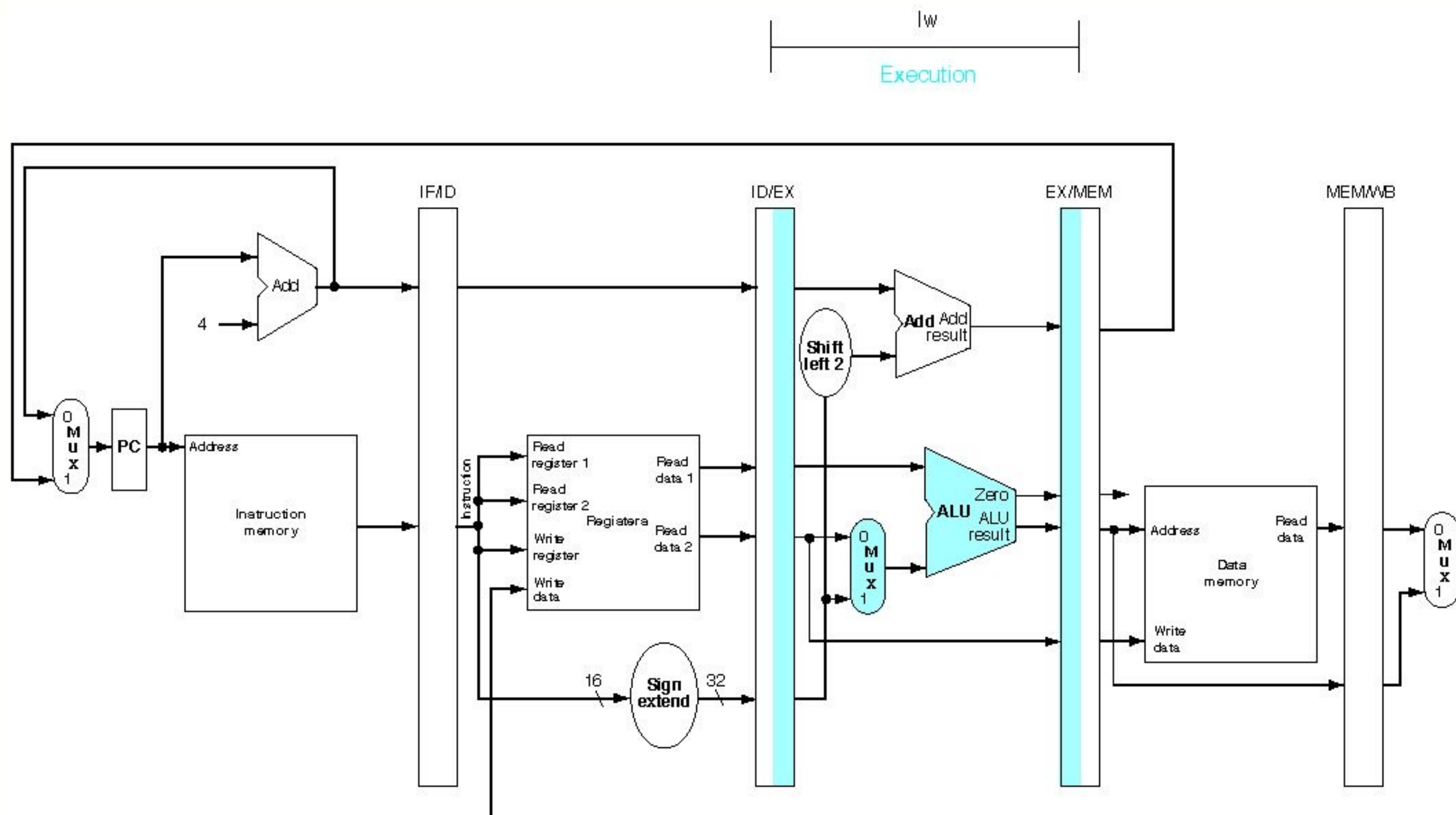
Instruction flow within the pipelined MIPS

- **Instruction Decode**: the IF/ID register is read to address the register file. Both addressed registers are read, even should only one be actually used later. The 16 bits of the immediate operand are converted to a 32-bit, and the three data are stored along with the PC in ID/EX



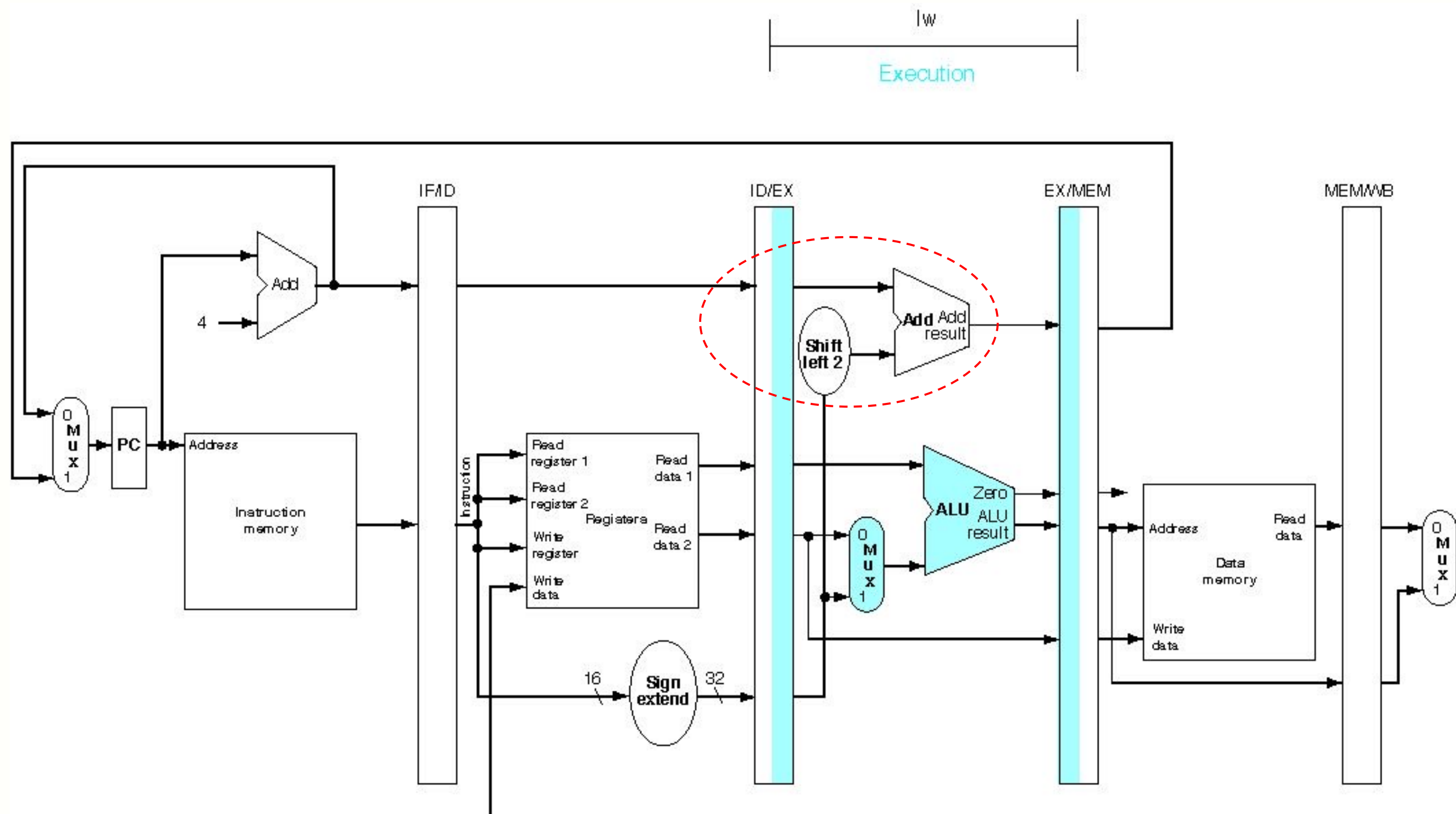
Instruction flow within the pipelined MIPS

- **EXecution**: the LOAD reads in ID/EX the content of register 1 and the immediate, sums them using the ALU and writes the result into EX/MEM. (Patterson-Hennessy, fig. 6.13).



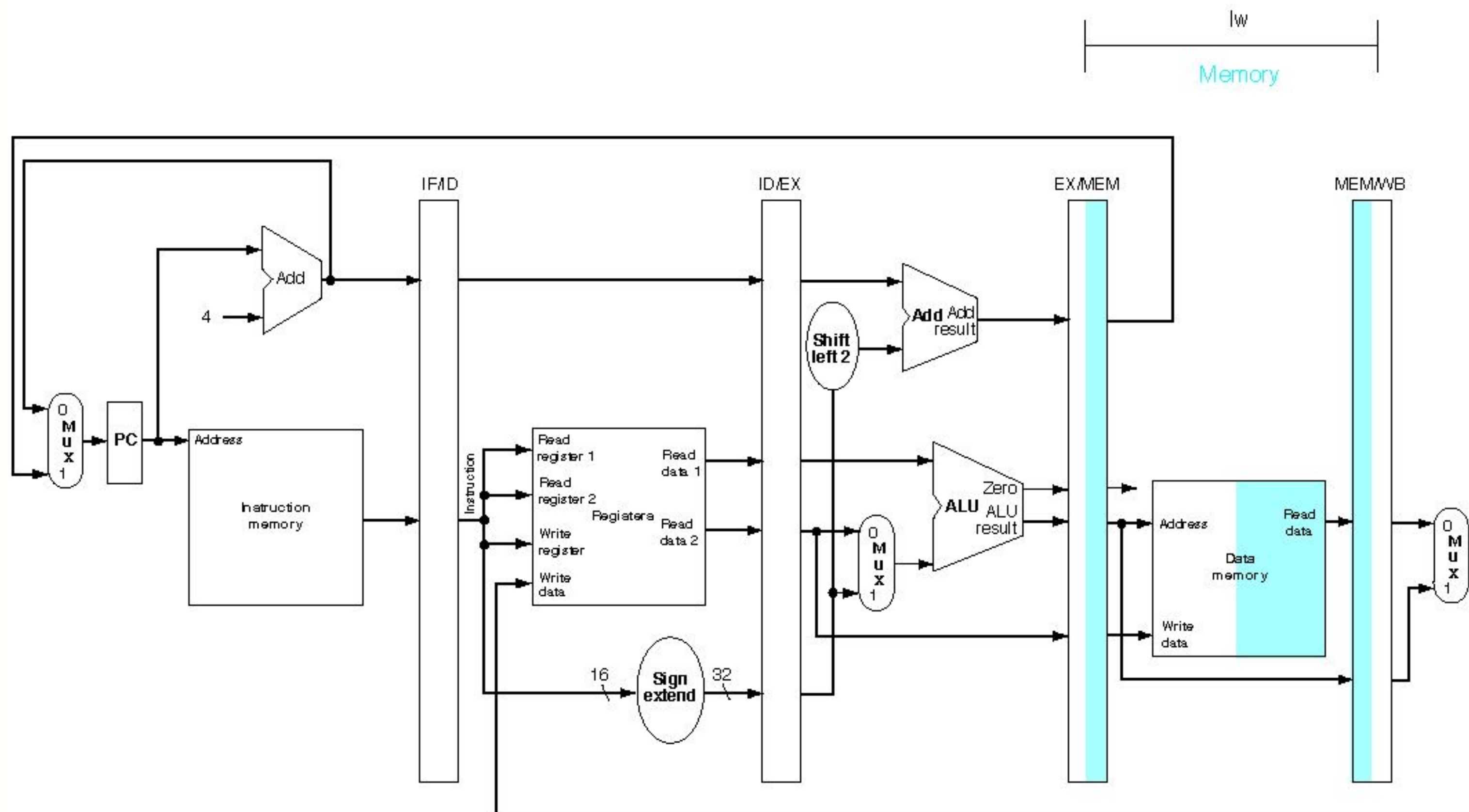
Instruction flow within the pipelined MIPS

- **EXecution**: if the execution is a BRANCH (or a JUMP), the adder uses the PC value (available ID/EX) to compute the new PC value (it will be used only if the BRANCH is taken)



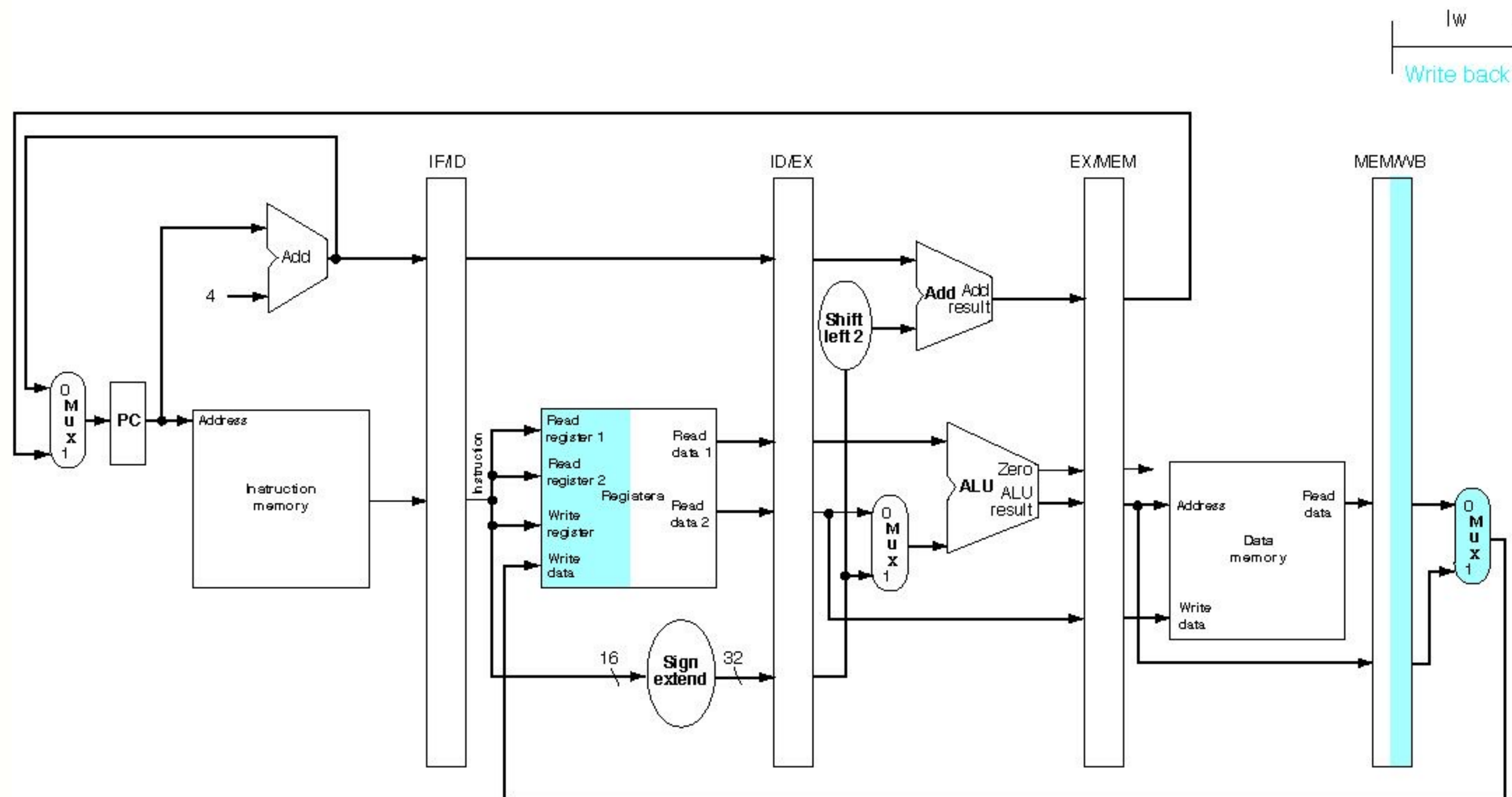
Instruction flow within the pipelined MIPS

- **MEMory**: the data memory is accessed, using the value available in EX/MEM as address. The data read is stored into MEM/WB. (Patterson-Hennessy, fig. 6.14a).



Instruction flow within the pipelined MIPS

- **Write Back**: The LOAD destination register is written with the data read from memory (Patterson-Hennessy, fig. 6.14b).
- Question: where is the *id* of the destination register ?

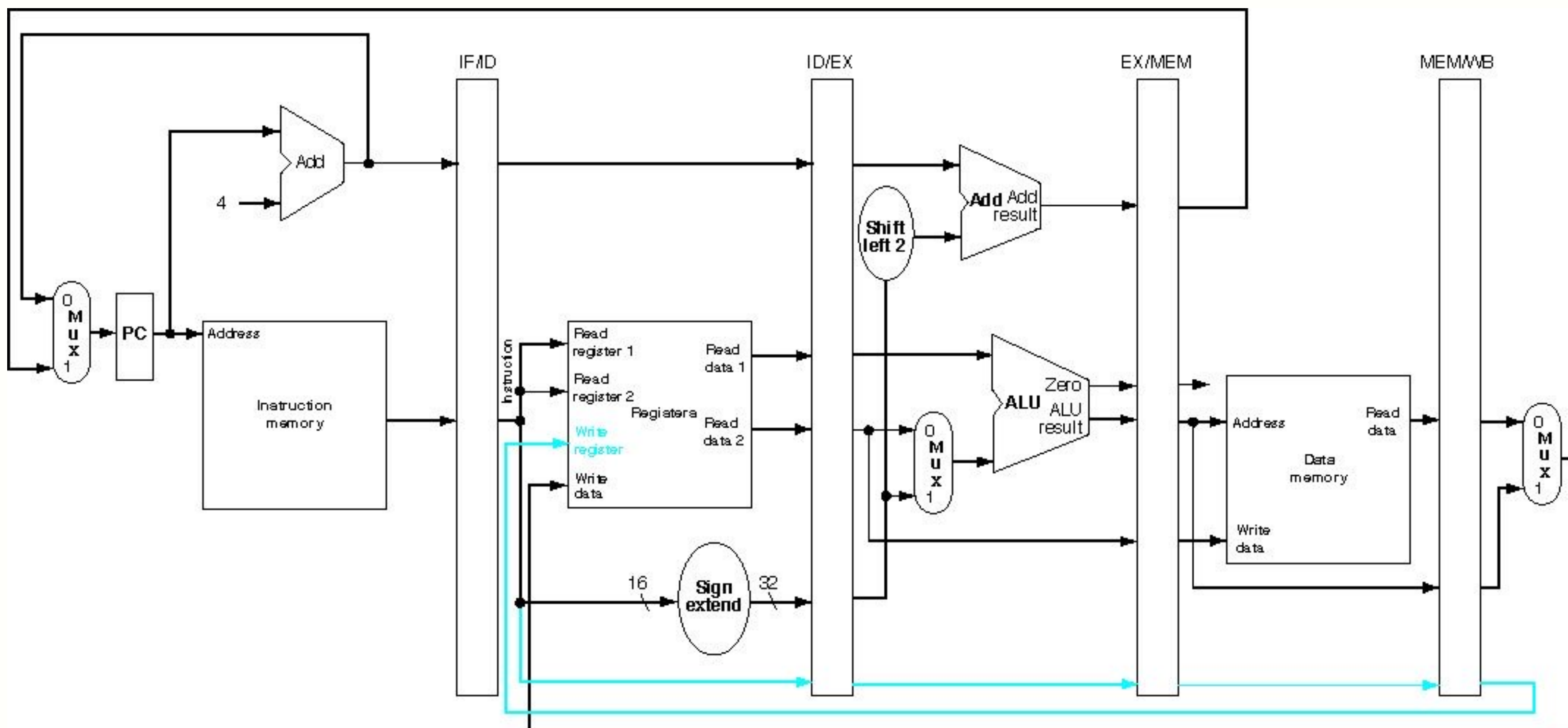


Instruction flow within the pipelined MIPS

- Pipelining is not a simple as it appears so far: in this datapath, when an instruction reaches the WB stage, the IF/ID register already contains the id of the destination register of a successive instruction!
- To avoid loading the data into the wrong register, the LOAD instruction cannot simply store the id of its destination register in IF/ID, because it would be overwritten at the next clock by a new instruction in its IF stage.
- In the various execution steps of the LOAD, the *id* of the destination register must be tranferred from IF/ID to ID/EX, to EX/MEM and eventually to MEM/WB, to be used to address the correct register during WB.

Instruction flow within the pipelined MIPS

- Here is a correct datapath to handle the load: the destination register *id* flows through all pipeline registers to be available during WB. The pipeline registers must be extended to store this value as well (how many more bits?) (Patterson-Hennessy, fig. 6.17)



Performance

- The pipeline (with its increased hardware) makes the complete execution of an instruction longer than its counterpart in a not-pipelined datapath
- But the CPU **throughput** increases, so programs execute faster (how much faster ?)
- Let us consider a CPU with no pipeline and the following characteristics:
  - ALU and branch operations take 4 clock cycles.
  - Memory opearations take 5 clock cycles.
  - ALU op. = 40%; branch = 20%; memory acc. = 40%.
  - clock = 1 ns.

Performance

- What is the average execution time for an instruction?

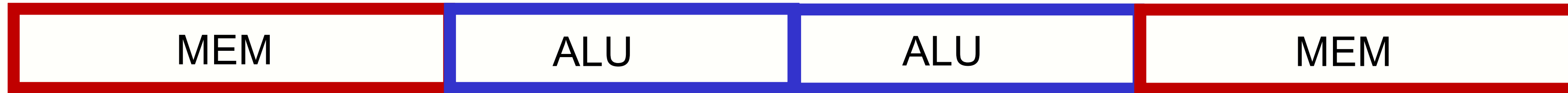
  Avg. exec time = (0,4 + 0,2) x 4ns + 0,4 x 5 ns = 4,4 ns.

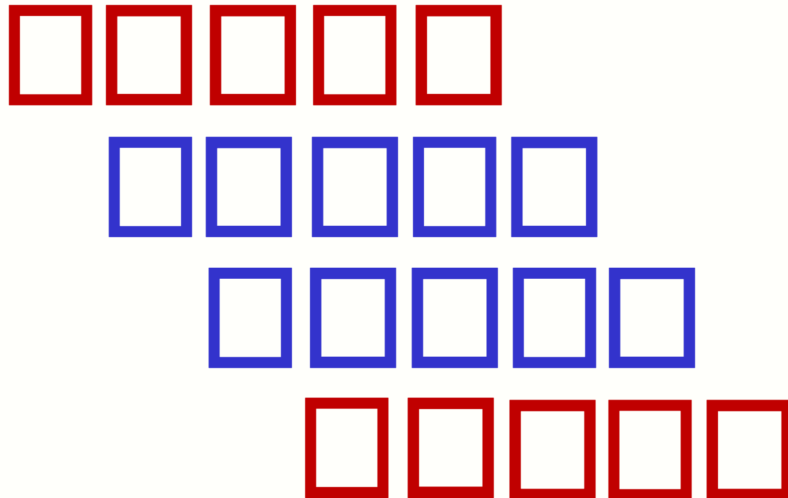
- Let us now assume that the CPU is indeed pipelined, and that the overhead caused by the hardware for pipelining is 0,2 ns (a reasonable value). Then:

  speedup = avg. exec time without pipeline / avg. exec time with pipeline = 4,4 ns / 1,2 ns = 3,7.

- A (theoric) performance speedup of 3,7.

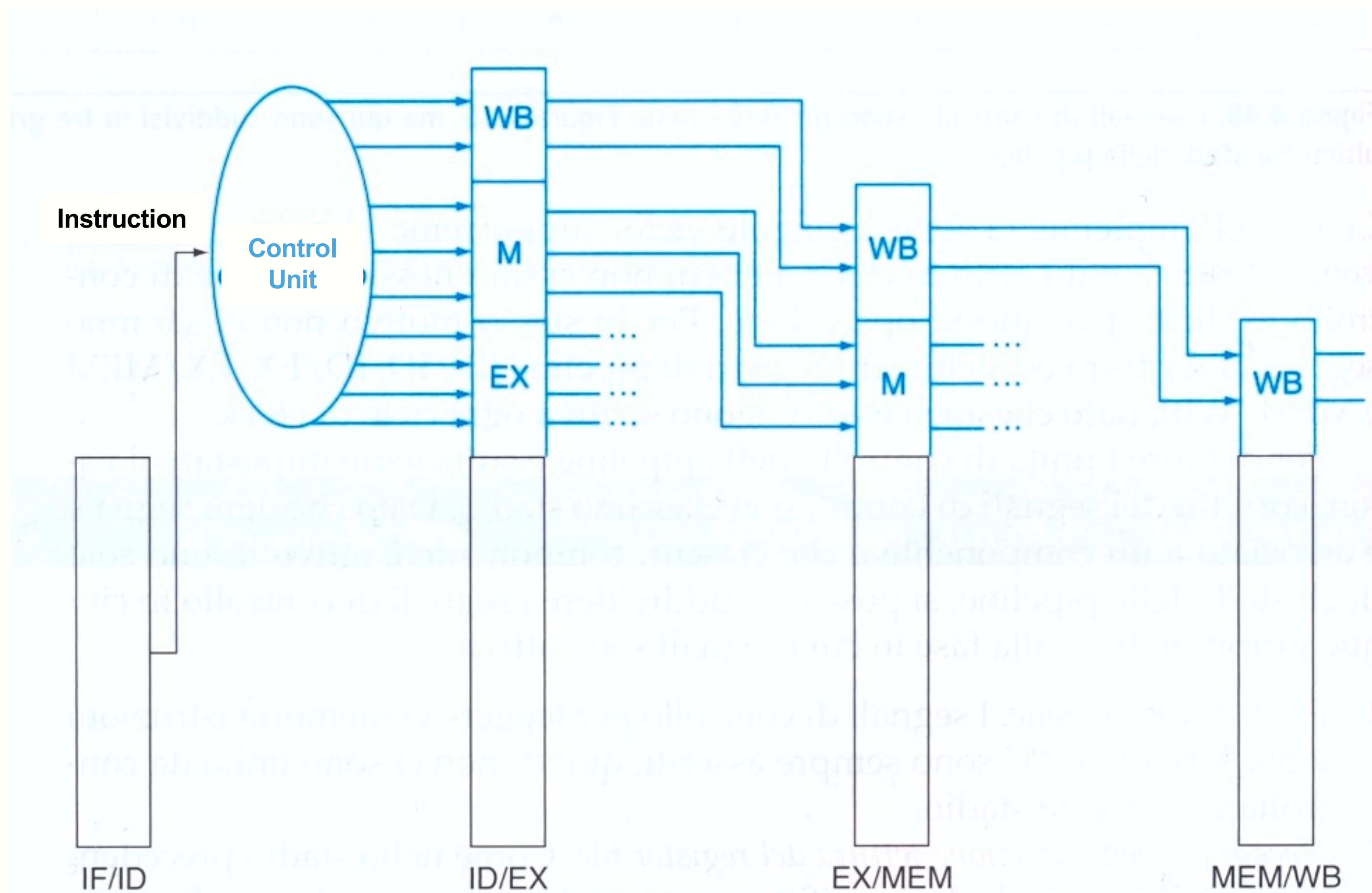
Performance





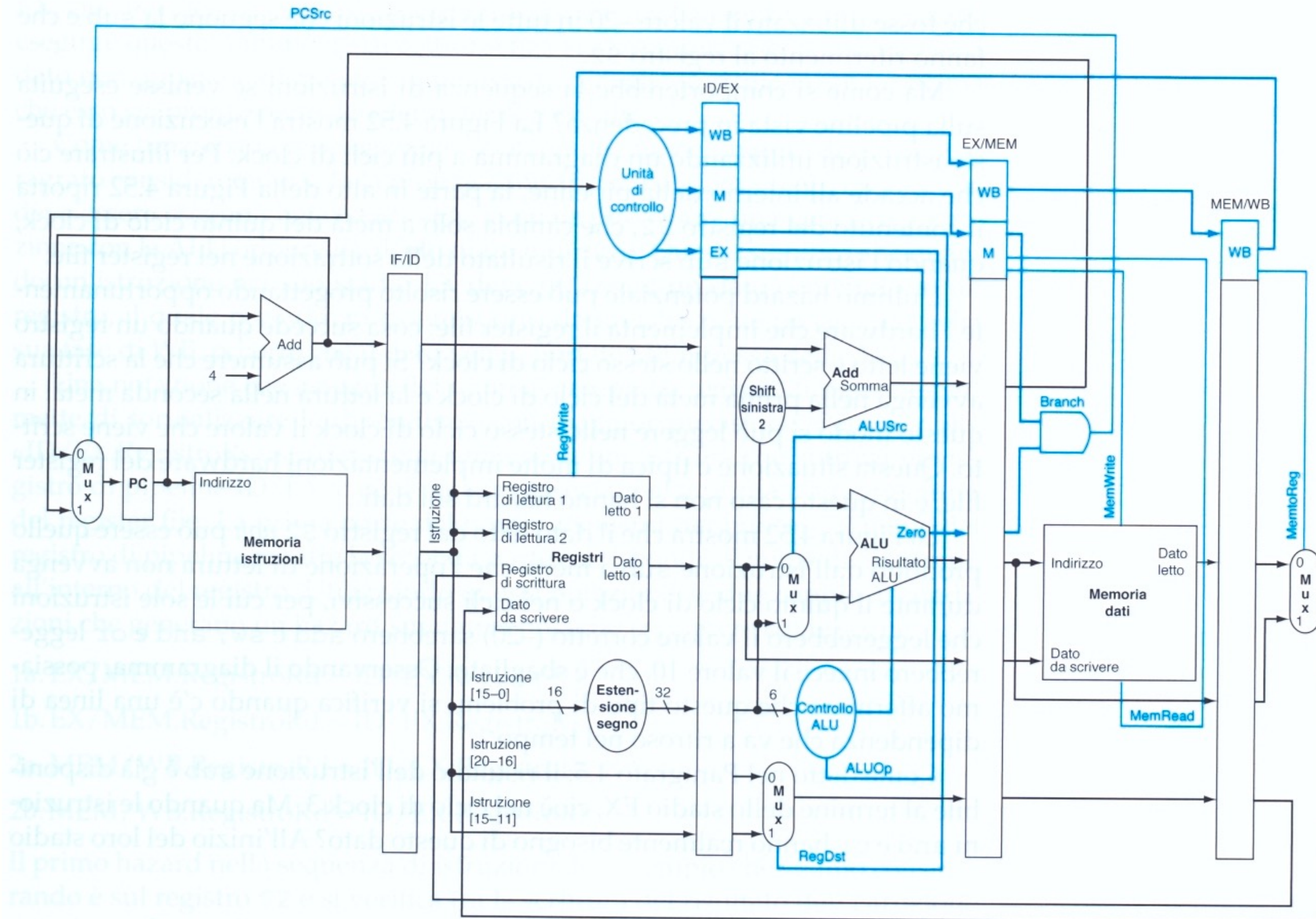
Control in the Pipeline

Pipelined control



Control in the Pipeline

Pipelined control: datapath control points



Pipelining hazards

- Unfortunately, pipelining is not that simple! There are three types of problems (**hazards**) that limit the effectiveness of pipelining:

1. **Structural hazards :** conflicts in datapath *resources*, arising when a specific combination of instructions cannot be executed simultaneously in the pipeline
2. **Data hazards :** conflicts due to a *data dependency*, when an instruction depends on the result (not yet available) of a previous instruction
3. **Control hazards :** pipelined branch execution, that can change the PC and instruction flow

Pipelining hazards

- When an hazard arises, it is necessary to stop (to stall) the pipeline: some instructions can proceed, while other are delayed.
- We can assume that when an instruction *IS* causes an hazard in the pipeline (the simple pipeline we are considering so far):
  - instructions issued *earlier* than *IS* can proceed to their commit,
  - instructions issued *later* than *IS* (thus in pipeline stages preceding *IS*) are stalled as well.
- In the following, we consider some basic solutions to the three types of hazards.

Structural hazards

- Usually, **structural hazards** arise because some hardware resources within the datapath cannot be duplicated (due to their complexity or to excessive production costs), and are requested at the same time by two different instructions in the pipeline.
- As an instance, a *single* L1 cache for data and instructions would continuosly cause structural hazards, and this is why L1 caches are separate.
- Another hazard arises if two instructions cannot use the same ALU during the same clock cycle.

Structural hazards

- In modern CPUs (actually, in less recent ones as well), many functional units (especially combinatorial ones) are replicated many times (we'll consider a few case in the following).
- This requires more complex circuits, with increased costs and complexity.
- A balance must be found among design requirements, performance, power consumption and production costs.

Data hazards

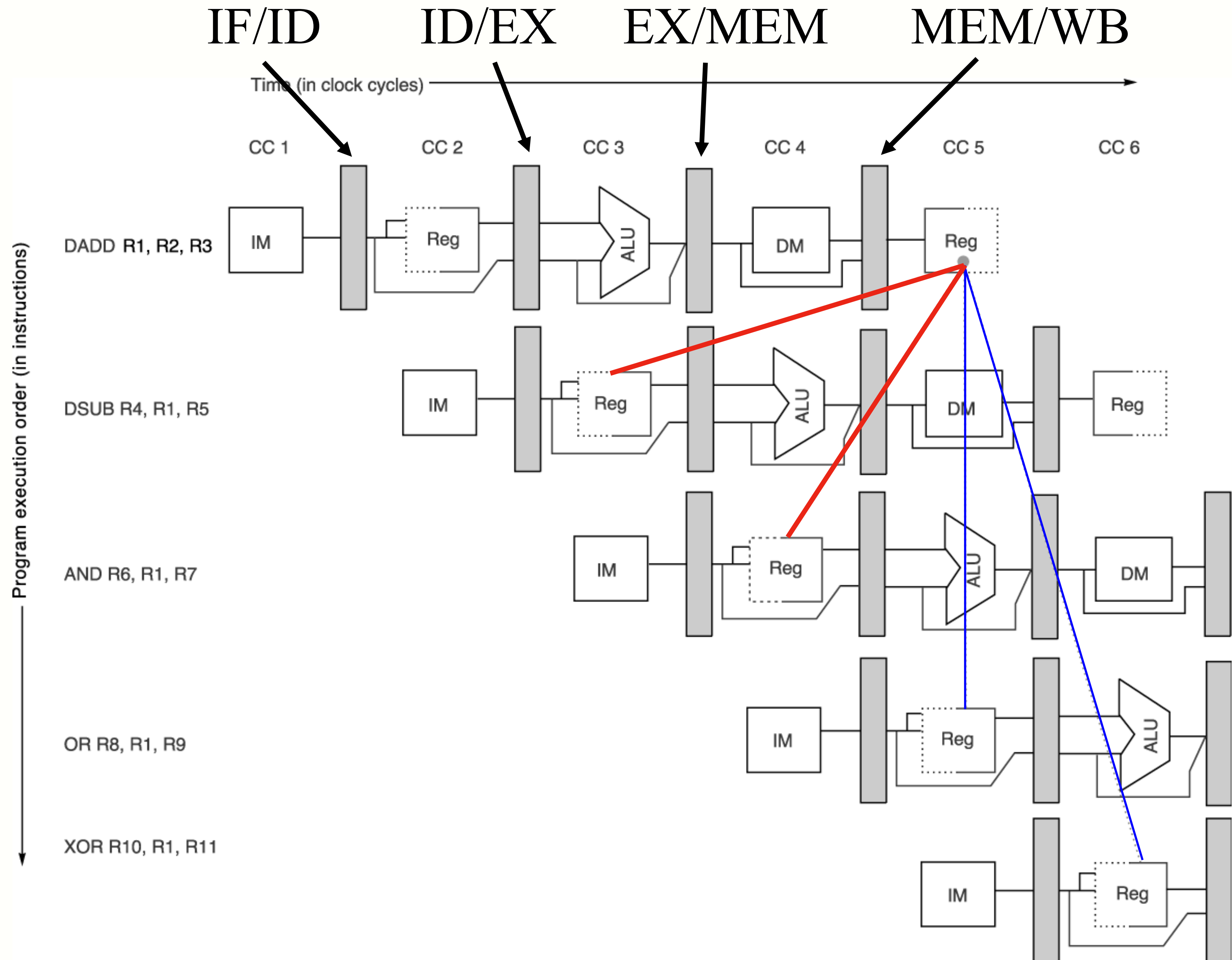
- **Data hazards** arise because instructions are correlated (un-correlated instructions are no algorithm at all!); usually instructions use values produced by preceding ones.
- Let us examine the following sequence of instructions:

```
DADD    R1, R2, R3     // R1 ⟵ [R2] + [R3]
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

- R1 is written by DADD in stage WB, but DSUB should read this value in stage ID, *before it is actually produced!*

Data hazards

The use of DADD result in successive instructions causes a hazard, because the R1 register is written after the instructions try to read it (Hennessy-Patterson, Fig. A.6)



Data Hazards

- Have a look at the AND: R1 is written at the end of clock cycle n. 5, but it is read from the AND in cycle n. 4.
- The OR works with no problem at all, since it reads R1 in the second half of clock cycle n. 5, and the register has just been written in the first half of the same clock cycle.
- Lastly, the XOR works fine as well.

Data hazards: forwarding

- A "simple" technique to solve these hazards is known as **forwarding** (also *bypassing*, or *short-circuiting*)
- The "trick" is to make the results of an instruction (the DADD in the example) available to the following ones *as soon as possible*, even before the WB phase of the DADD.
- The pipeline registers can be used to this purpose; indeed, they store the intermediate results of the various stages in the pipeline.

Data hazards: forwarding

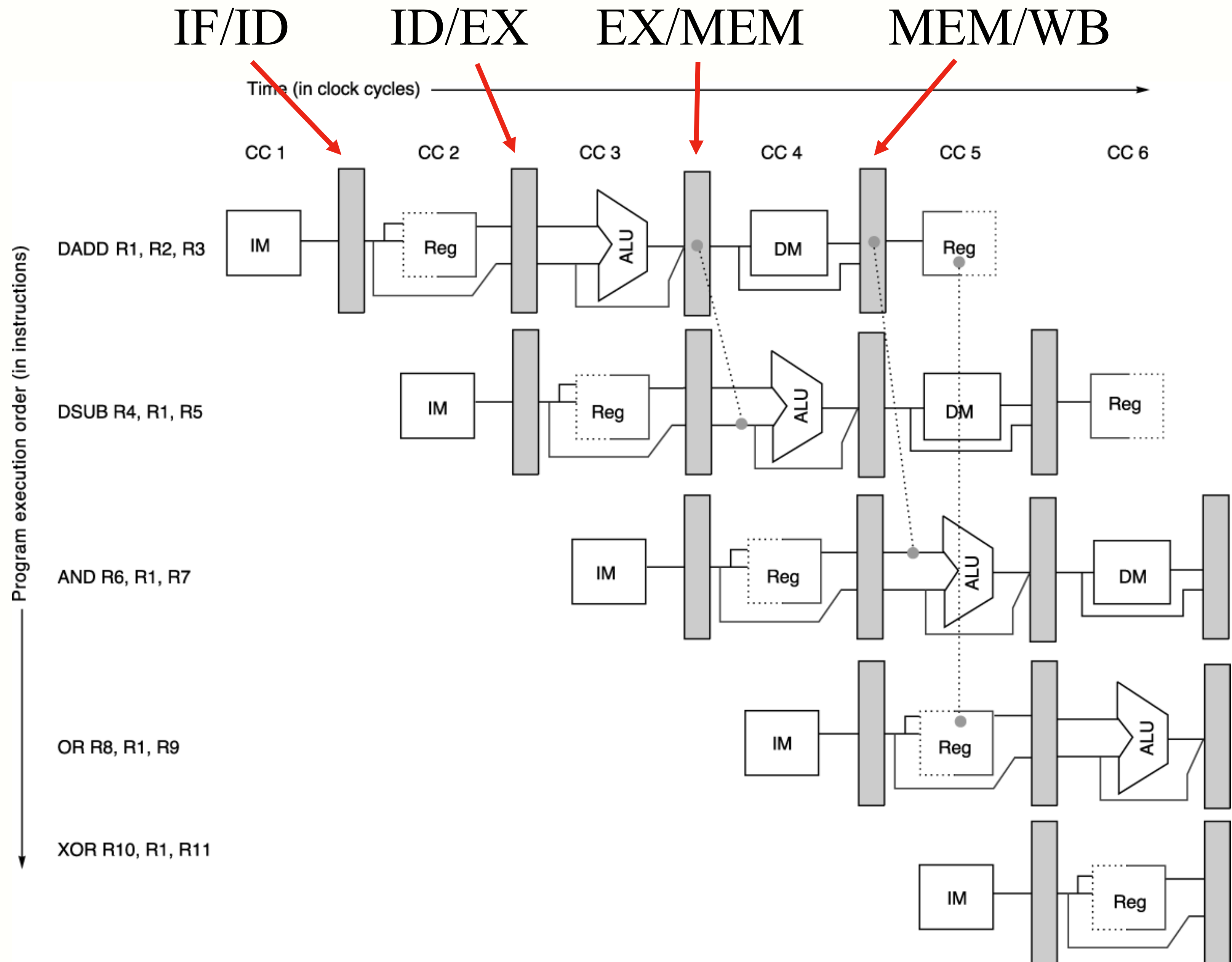
- Please, note that :

1. The output of the ALU produced in the EX stage is stored in the EX/MEM register and passed on to MEM/WB register, and can be re-used as input by the ALU itself in following clock cycles.

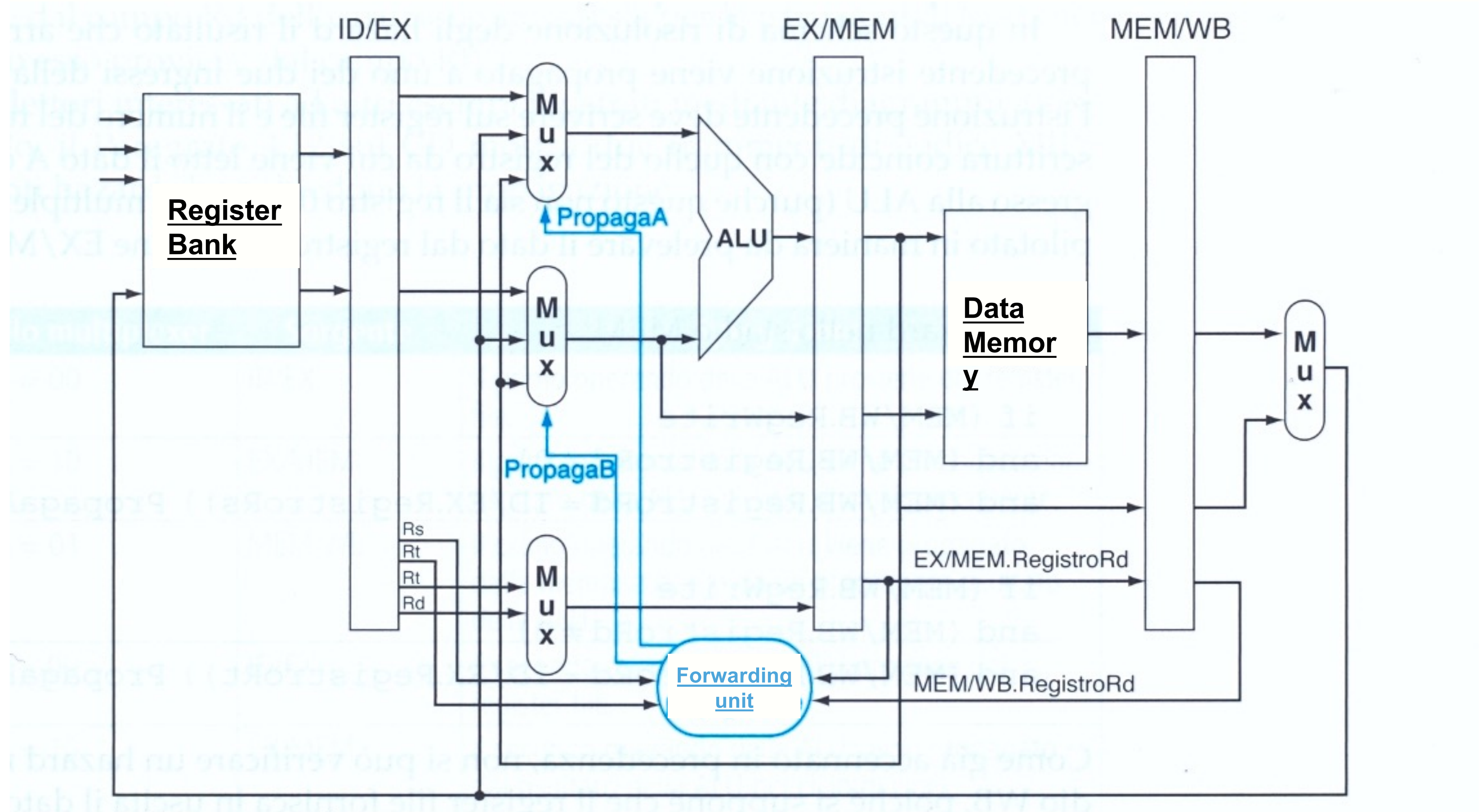
2. If the Control Unit detects that a preceding instruction is writing a result that has to be re-used immediately as input to the ALU, it fetches such value from the proper pipeline register, instead of waiting for it to be available in the destination register.

Data hazards: forwarding

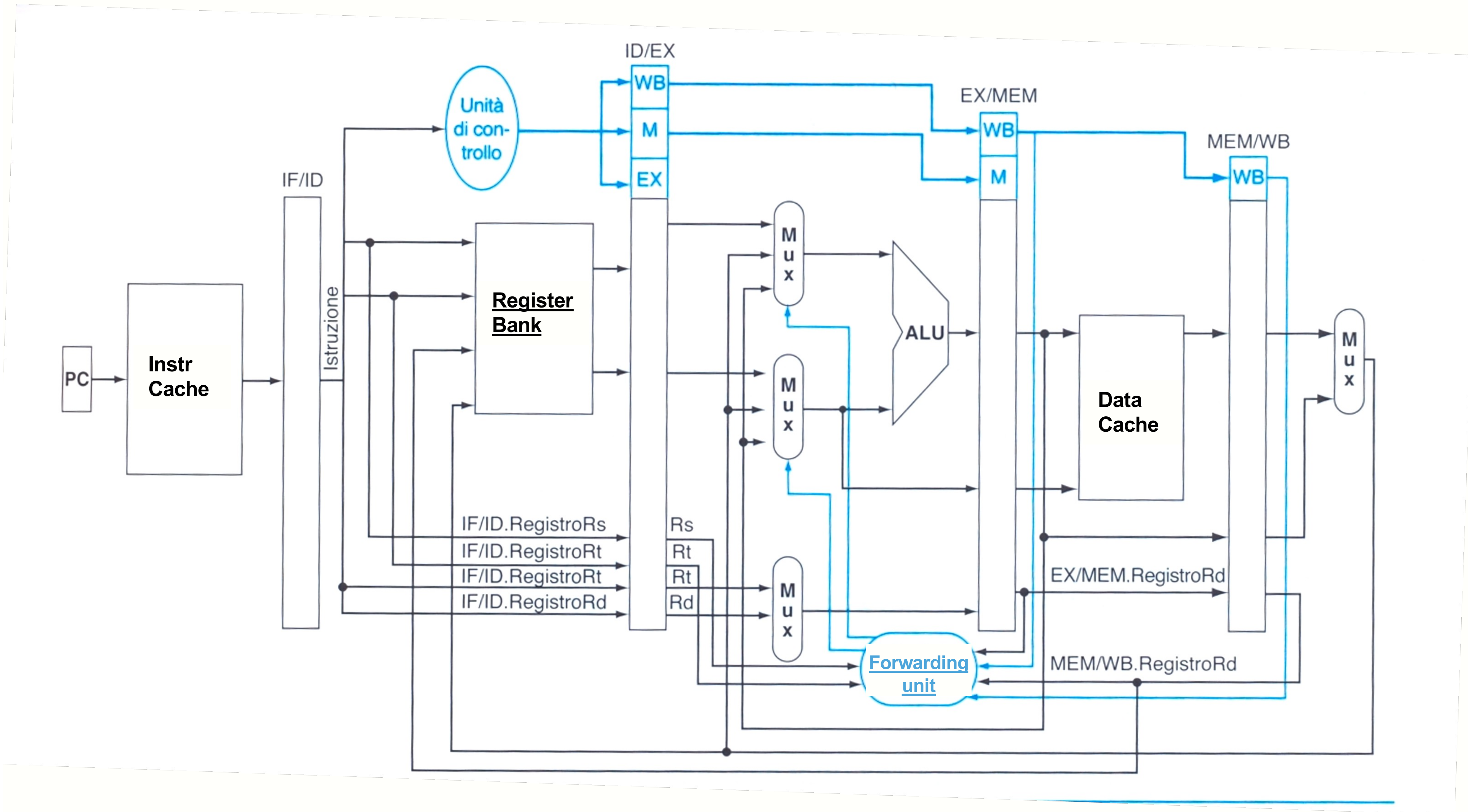
Forwarding allows to prevent *some* data stalls (Hennessy-Patterson, Fig. A.7)



Data hazards: forwarding



Data hazards: forwarding



Data hazard: a stall

- Forwarding cannot solve all possible stalls due to data dependencies. Let us consider the following code fragment:

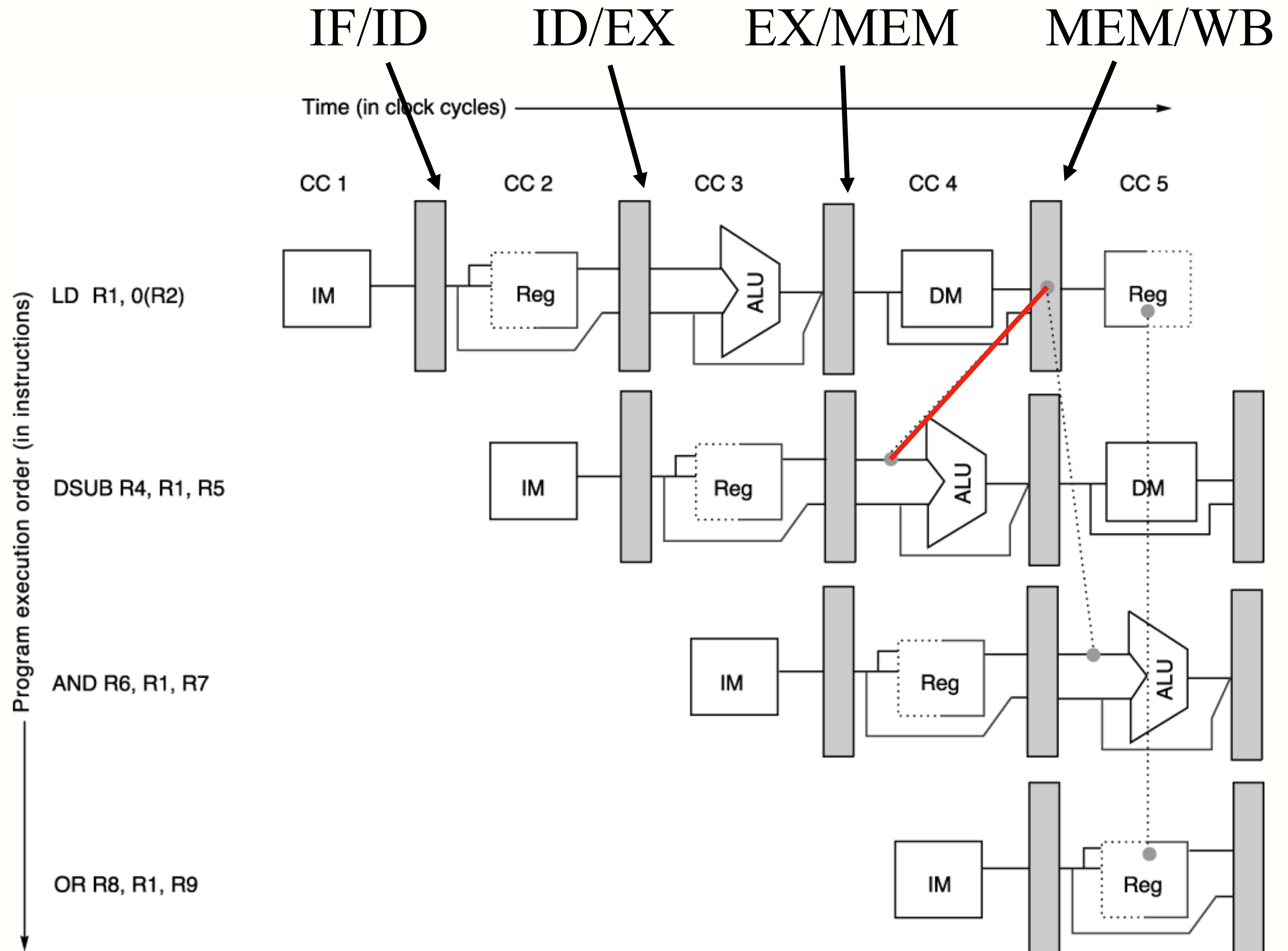
```
LD      R1, 0(R2)    // address ← [R2] + 0
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

Data hazard: a stall

- The data addressed by the LD will be loaded into MEM/WB only at the end of clock cycle 4, while the DSUB requires it at the beginning of the same clock cycle.
- Whenever a load is executing, its targeted data is not available within the CPU until is is fetched from the cache (or, even worse, from RAM), and it cannot be used by any following instruction, and this actually stalls the pipeline.
- The pipeline circuitry detects this situation and stalls subsequent instructions (for a single clock cycle, in the best case) until the value they require is available.

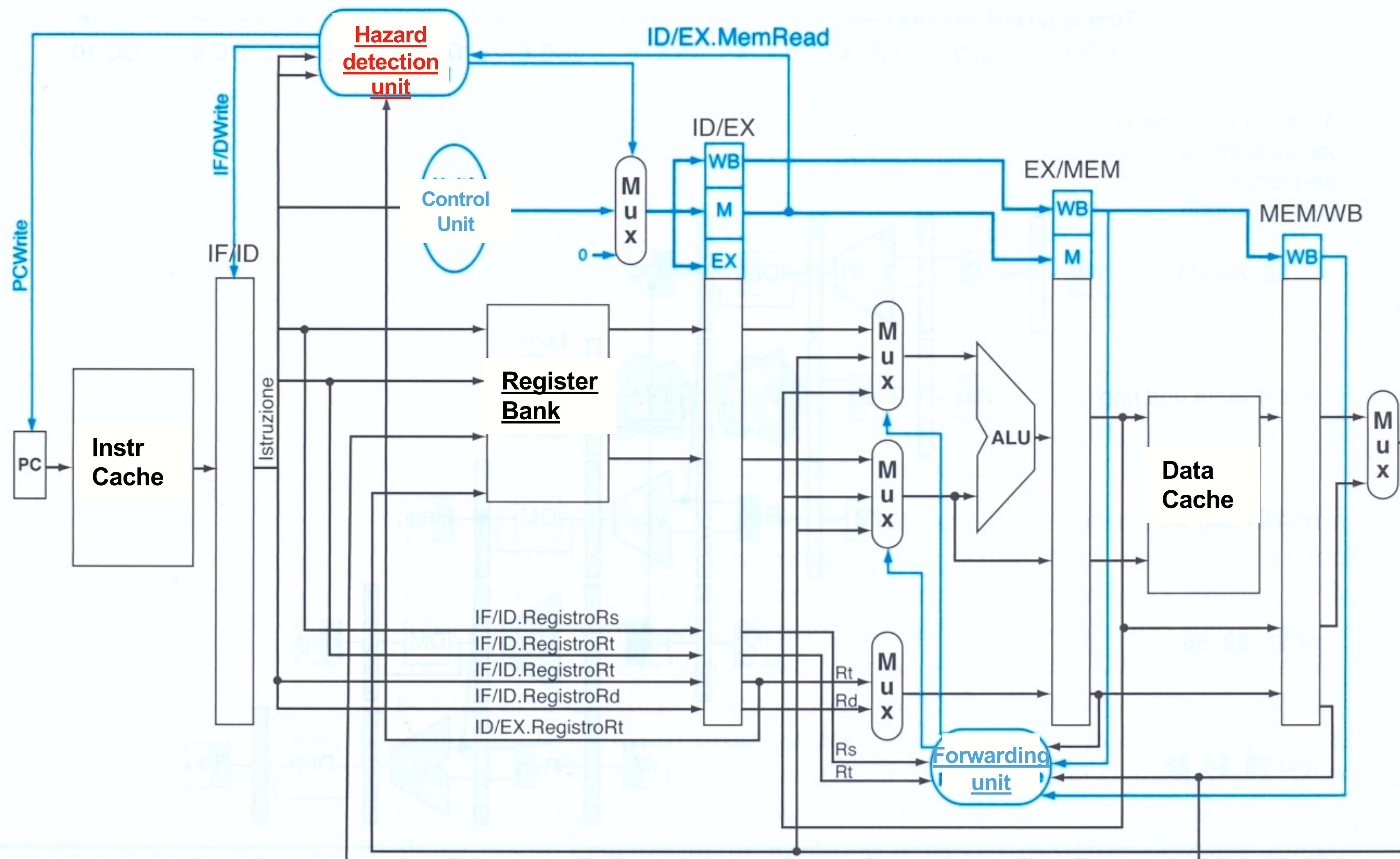
Data hazard: a stall

The LD can forward its data to the AND and to the OR, not to the DSUB (Hennessy-Patterson, Fig. A.9)



Data hazard: a stall

- **Hazards detection unit:** checks for producer-consumers dependencies that cannot be solved through forwarding
- It stalls the pipeline (If and ID phases) and produces a "bubble" (a NOP with proper control signals)



Control hazards

- **Control hazards** arise when a branch is executed, since it can either change the value of the PC or leave it uneffected.
- Even assuming an aggressive implementation which uses two clock cycles for the branch, if the branch is taken, the PC is not updated until the end of the second clock cycle, in the ID stage, after the address to be stored in the PC is actually computed.
- Thus, the instruction that is located immediately "after" the branch (at the address PC+4) has already completed the IF phase. What if it was not the one to issue, since the branch is taken?

Control hazards

- Let us consider the following example:

```
        LD      R1, 0 (R4)
        BNE     R0, R1, else
        DADD    R1, R1, R2
        JMP     next
else:   DSUB    R1, R1, R3
next:   OR      R4, R5, R6
```

- When the branch decision will be known, the DADD has already completed its IF phase.

Control hazards

- A simple approach is to carry out the fetch phase of the instruction that follows the branch (that is, DADD).
- At the end of the ID phase of the branch, it is known if the branch will be taken or not. If not, everything is OK and the DADD proceeds in its ID phase.
- If the branch is taken, the instruction to be issued is the DSUB, which starts with its IF phase: so, a clock clycle is lost, that for the IF phase of the DADD.

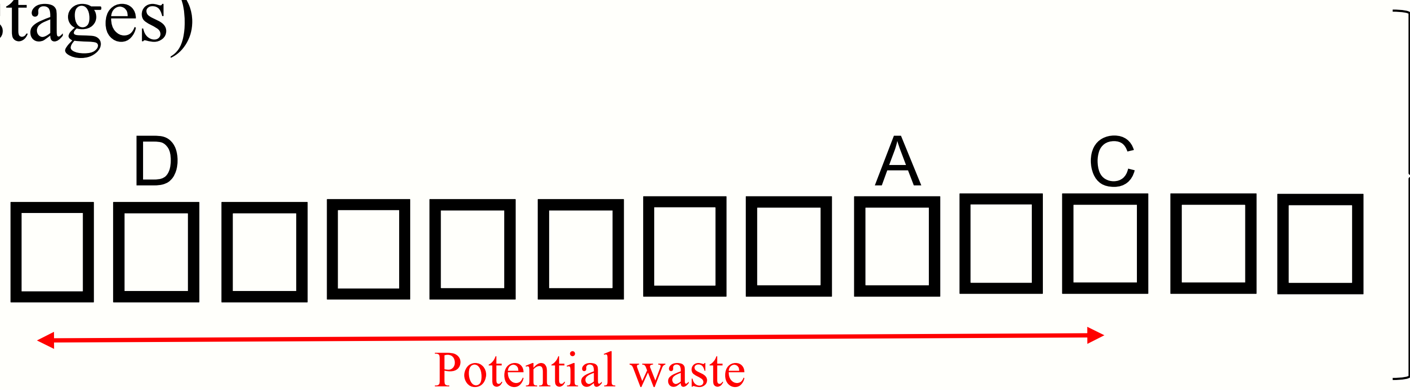
branch instr.	IF	ID	EX	MEM	WB	
branch succ.		IF_{DADD}	IF_{DSUB}	ID	EX	...
branch succ.+1				IF_{OR}	ID	EX

Control hazards

- Obviously, there is a waste in clock cycles, and the amount lost depends on the frequency of branches in a programme, and of the success (or fail) of each specific branch.
- In the example just considered, the waste due to a branch is one clock cycle at the most, since we assume that the datapath is able to complete a branch in two clcok cycles.
- More complex branch instructions can require more clock cycles, and the waste can be larger accordingly. In CISC instructions this was the usual pattern (can you give an example ?)

Control hazards

- Control hazards involve the management of three different actions:
  - instruction deconding (D)
  - nextPC (jump-to) address computation (A)
  - condition evaluation (C)
- Modern processors have superscalar pipelines (with up to 14 stages)



Control hazards

Some statistics

- Jump amount to 2%-8% in integer benchmarks.
- Branch amount to 11%-17%.
- in loops, 90% of branches are taken
- in generic IF, 53% are taken, of these 75% are *forward*

Control hazards

- A different approach is to make an a-priori prediction on the outcome of the branch. This is the so-called *static branch prediction*. As an instance, the pipeline hardware might make the following assumptions:

  1. Backward branches are always taken
  2. Forward branches are never taken

  and continue fetching the instruction following the branch in either case (how can this be realized?)

- However, if the prediction turns out to be wrong, the instruction issued "by mistake" must be turned into a no-op, and it is necessary to fetch the "correct instruction.

Control hazards

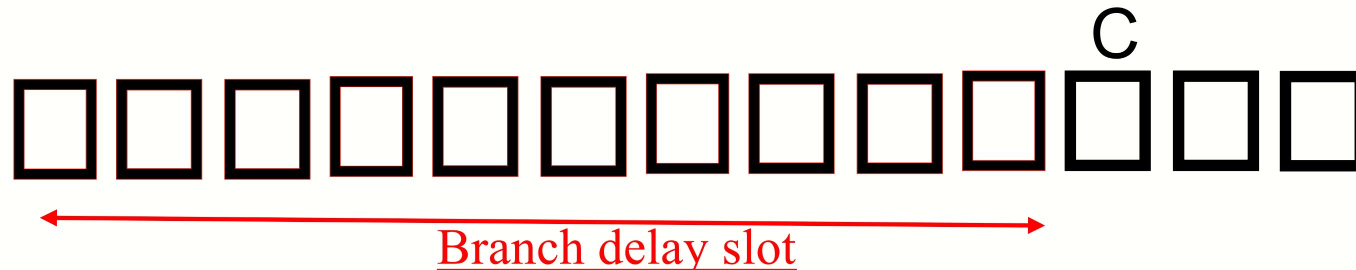
- If most predictions are correct, the waste is minor.
- As an instance, in a loop executed many times, the backward jump is executed most times, and the forward jump is executed only once, when the loop exits.
- Forward jumps are usually used for exceptions, to transfer control to the associated routine. Hopefully, exceptions should be rare...
- This kind of prediction best suites the simple *if then else*… construct

Control hazards

- Another technique, common in the first RISC architectures, is the so-called **delayed branch**.
- It consists of placing, right after the branch, instructions that *do useful work*, **independently of the outcome** of the branch.
- The instructions are executed throughout, and the branch is evaluated (taken or not) and acts accordingly on the PC.
- This technique requires action on the compiler side, and cannot be easily applied in all cases (the compiler plays a key role in improving CPU performance).

Control hazards

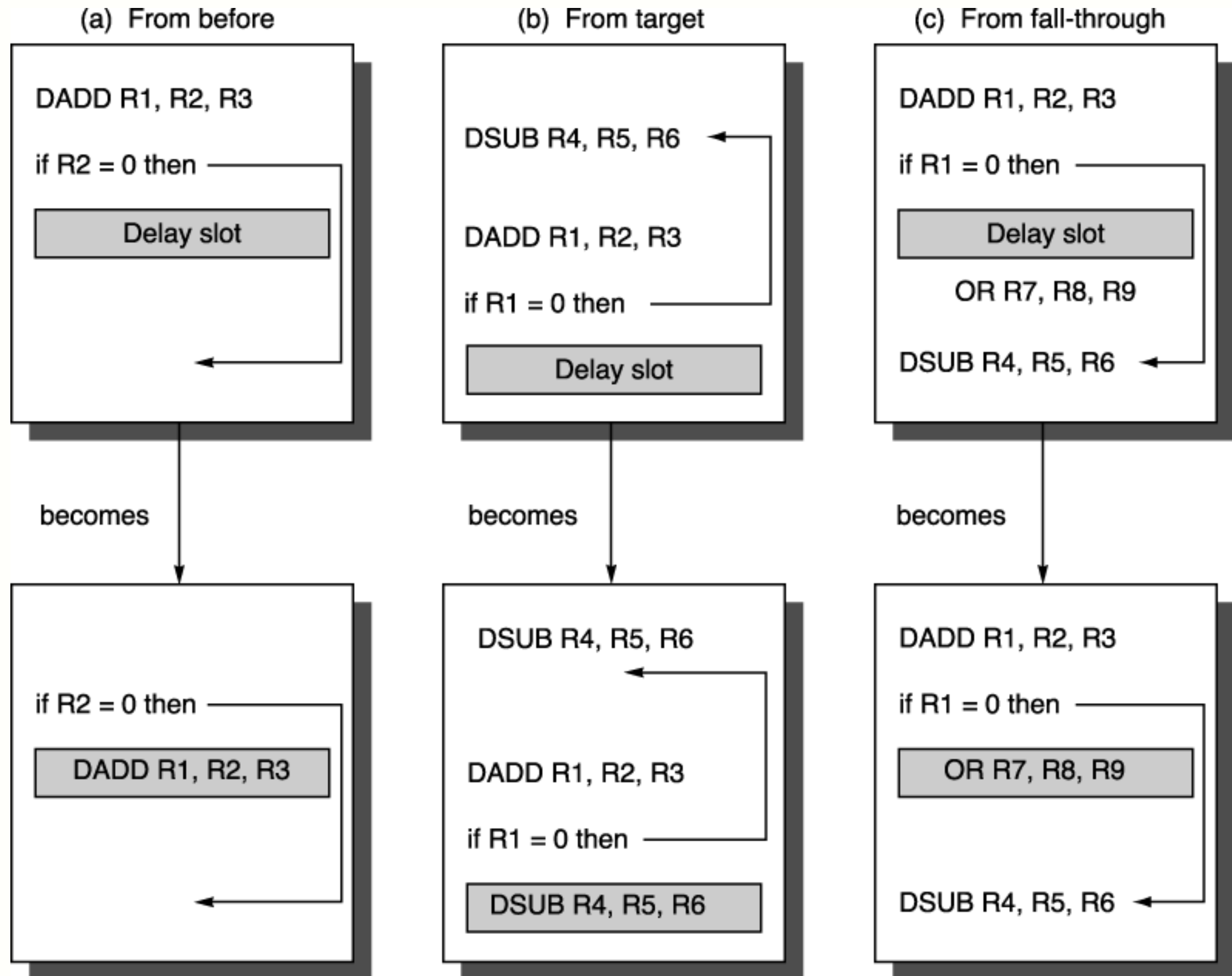
- The **branch delay slot** is the set of instructions that are "conditionally" fetched and sent to the pipeline stages.



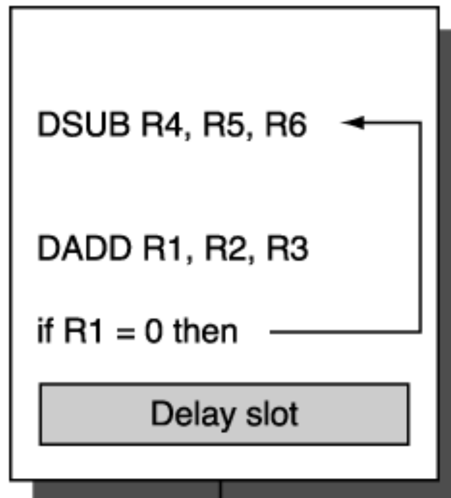
- Scheduling the branch delay slot is not easy, and the probability for the compiler to find "good" instructions to be placed in the delay slot rapidly decreases with the slot length.
- Let us assume (simplified pipeline) that the branch delay slot is *1-instruction* wide.

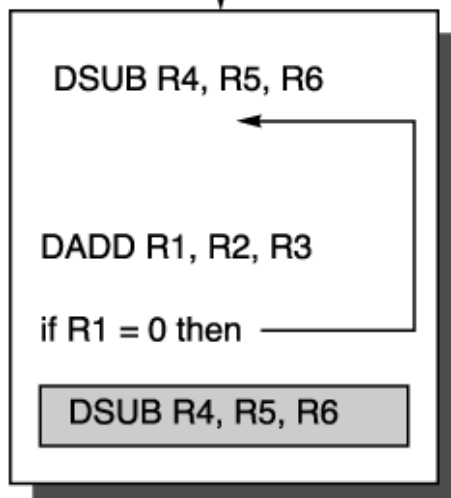
Control hazards

Case (a) is easily handled by the compiler, but as to the others... (Hennessy-Patterson, Fig. A.14).



Control hazards



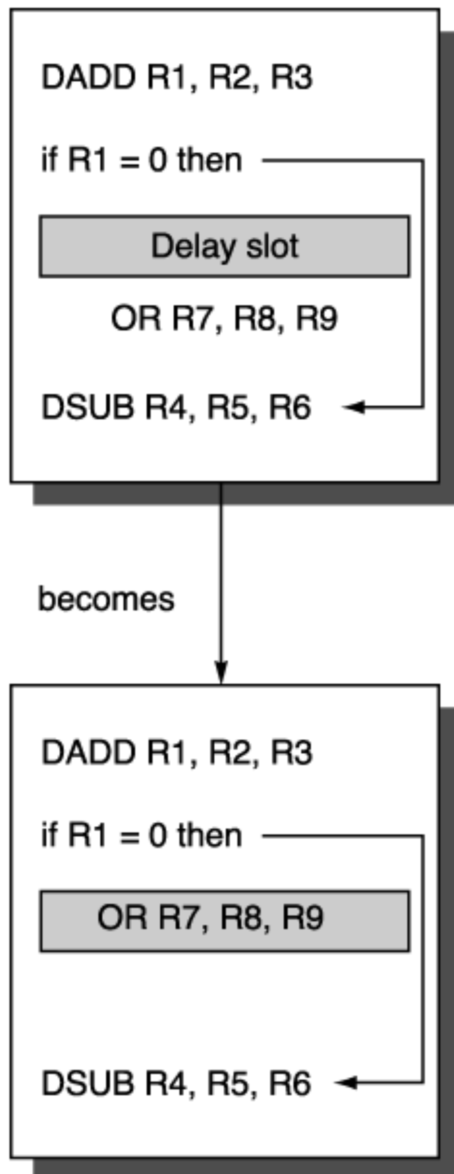


Possible ONLY if the assignements in the delay slot «cause no harm»:
registers assigned to in the delay slot are NOT read before they are assigned AGAIN

```
if R1=0 then go to label
DSUB R4,R5,R6
INSTR A (does not read R4)
INSTR B (does not read R4)
….
ADD R4,R7,R8 OK | SUB R5,R6,R4 WRONG
```

The last execution of DSUB is «wrong» but it is wiped out by the ADD. The SUB instead would read a wrong value.

Control hazards



Possible ONLY if the assignements in the delay slot «cause no harm»:
registers assigned to in the delay slot are NOT read before they are assigned AGAIN

OK

```
OR R7,R8,R9
DSUB R4,R5,R6
ADD R7,R8,R9
```

WRONG

```
OR R7,R8,R9
DSUB R4,R5,R6
ADD R8,R7,R9
```

Exceptions

- Interrupts, traps, faults, other exceptions that alter the flow of instructions make the operations of a pipeline very difficult to manage, since instructions are overlapped in their execution.
- In the most general case, whenever an instruction *IS* raises an exception, the pipeline must be frozen, instructions *after* IS should be allowed to complete execution, and those *before* IS should be restarted from scratch, only after the exception has been taken care of.
- We do not consider the details of the implementations, just keep in mind that this is indeed a very complex task, that increases consistently the hardware complexity in the pipeline.

Exceptions in integer pipeline

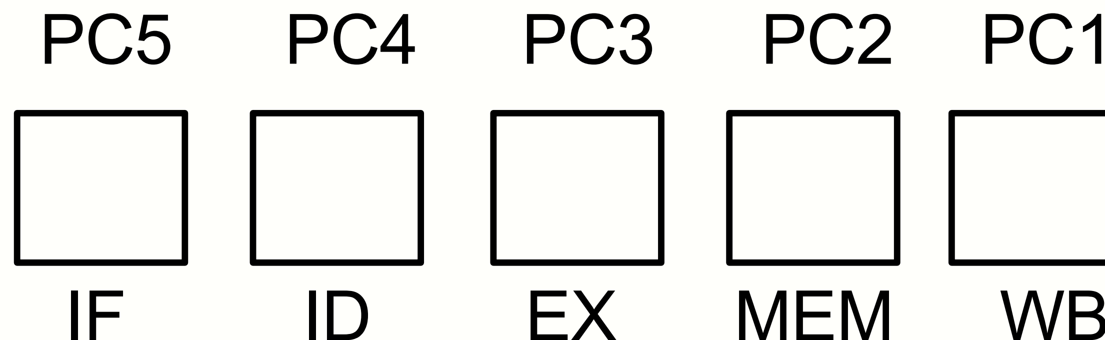
- 5 instructions executing in 5 pipeline stages
  - how is the pipeline stopped?
  - what about restart
  - who actually caused the interrupt?
  - exceptions can be raised in a time sequence that violates the in-order restart mode of instructions.
- Precise exception: the set of instructions in the pipeline is divided into two sub-sets A and B by the instruction I that raises the exception: <A I B> with B entered BEFORE
- B is completed, I is serviced and, if recoverable, I is re-scheduled, and A comes after I

Exceptions in integer pipeline

- types of exceptions – synchronous mode
- 5 instructions executing in 5 pipeline stages
  - IF page fault on instruction fetch; misaligned memory access; memory-protection violation
  - ID undefined/illegal op-code
  - EX arithmetic exception (overflow, divide_by_zero, …)
  - MEM page fault on data fetch, misaligned memory access; memory-protection violation; memory error
  - WB none

Exceptions in integer pipeline

- Exceptions raised out-of-order
  - The sequence of fetched instructions is PC1, PC2, PC3,PC4,PC5
  - PC1 raises the exception in MEM, PC2 in ID
  - The sequence of raised exceptions is therefore PC2, PC1
  - Precise exception management is NOT possible, because during the servicing of the exception raised by PC2 a further exception will be raised by PC1 that is in stage EX when PC2 raises its exception !!



Exceptions in integer pipeline

- Precise exception management
  - Instructions that raise exceptions set a control flag (plus additional info) but are not immediately worked upon
  - Exceptions are serviced ONLY at the MEM/WB transition
  - A single exception of a single instruction is serviced and a single PC is restored (if the exception is recoverable)
  - The assumption is that *the state of the machine is committed only after the MEM/WB transition*
  - This is a general approach is the previous assumption holds true

Enhanced pipelines

- The 5-stage pipeline architecture was used already in the first RISC processors, and it is still in use in low-end processors aimed at embedded applications.
  - nintendo-64 game processor / dreamcast
  - Processor in laser printers, cameras, video camcoders,…
  - routers

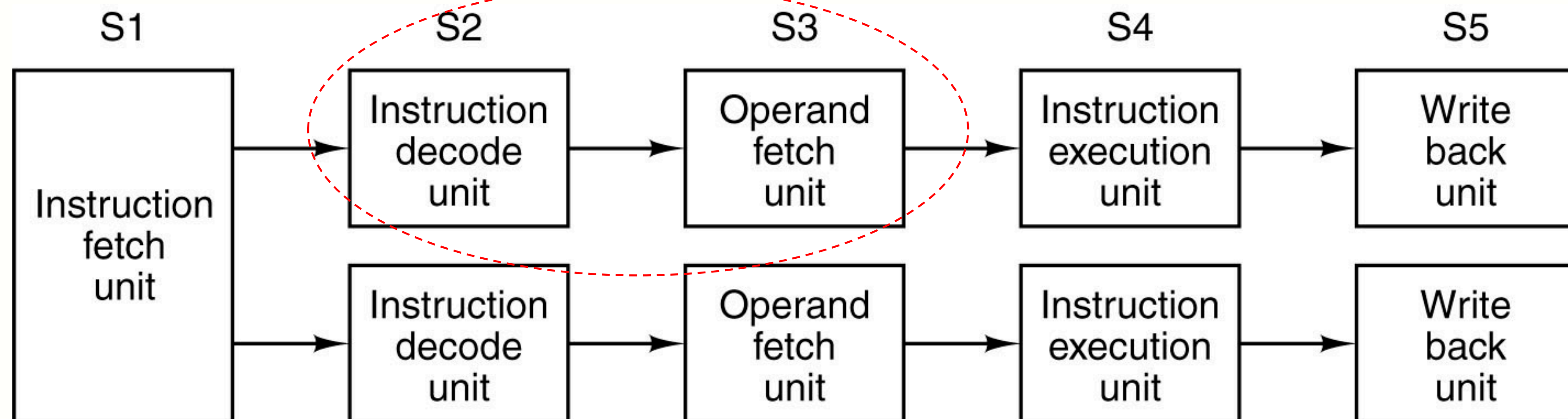
Enhanced pipelines

- So, if pipelining allows for an effective architecture, what about using **two** pipelines? As long as:
  - it is possible to concurrently fetch two instructions from Instruction Memory,
  - the two instructions, issued together, do not conflict in accessing the registers, and
  - they do not depend on each other
- there will be a further gain in computation
- Of course, even in this architecture, hazards must be minimized either at compile time, or at run-time (using techniques to be examined later)

Enhanced pipelines

- A two-pipeline architecture with five stages (approx.) has been used in the Pentium (the INTEL processor just after the 80486). Pipeline **u** could execute any instruction, while pipeline **v** could handle only simple integer-type instructions (Tanenbaum, Fig. 2.5)

BTW, what is "peculiar" in this scheme, with reference to a RISC architecture ?

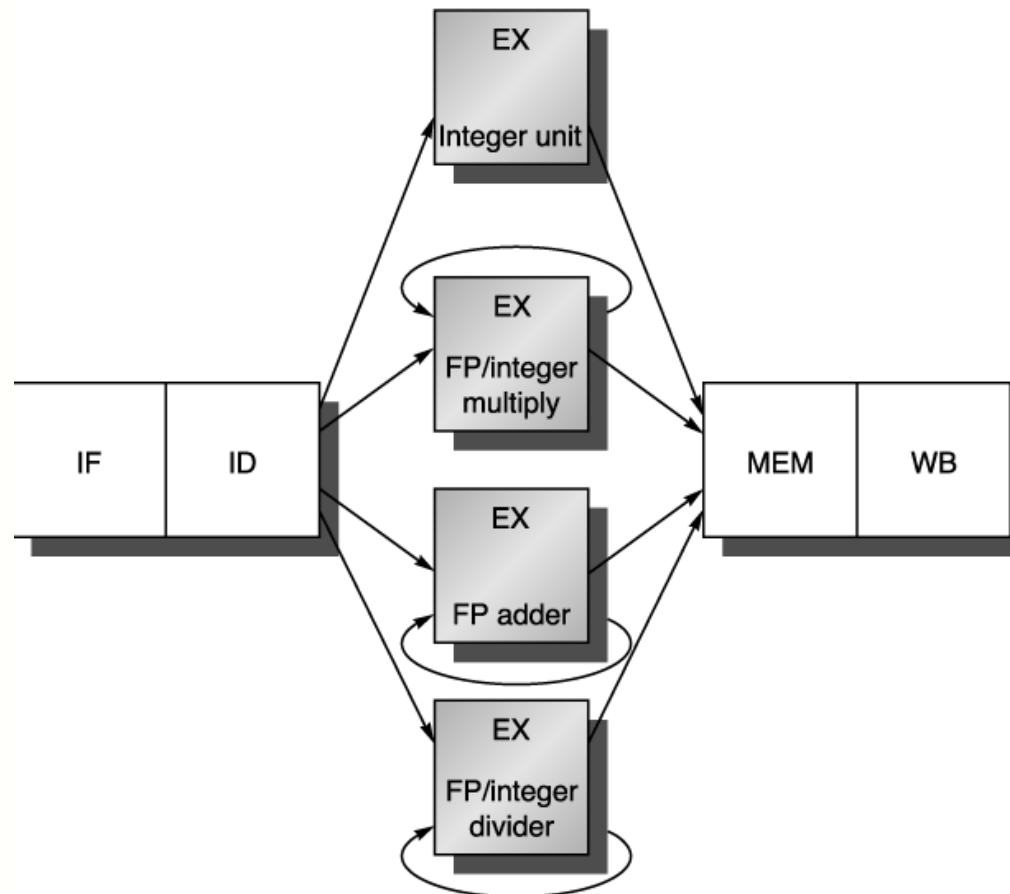


Enhanced pipelines

- Instructions are executed “in-order”, that is in the order specified by the program (well, can you do otherwise ?) and a few fixed rules established if the two instructions are“compatible”, so that they can be executed in parallel.
- If not, only the first instruction is issued, and a coupling is attempted between the second and the third, and so on …
- Specific compilers targeted to the Pentium were capable of producing code with a high number of “coupled instructions”
- A Pentium with an optimizing compiler ran code twice faster than the 486, at the same clock frequency.

Enhanced pipelines

- Of course, one can think of multiple (>2) pipelines operating in parallel, but modern architectures have taken a different route, introducing more functional units in the EX stage (Hennessy-Patterson, Fig. A.29)



Enhanced pipelines

- Actually, in a CPU executing a fairly complete set of instructions, some instructions have an EX phase much longer than the other phases:
  - Add/subtraction of floating point numbers
  - Multiply/division of integer numbers
  - Multiply/division of floating point numbers
- it is possible to carry out these operations in a single, *very long* clock cycle, but this would slow down uselessly the other phases, which can do with a much shorter clock cycle.

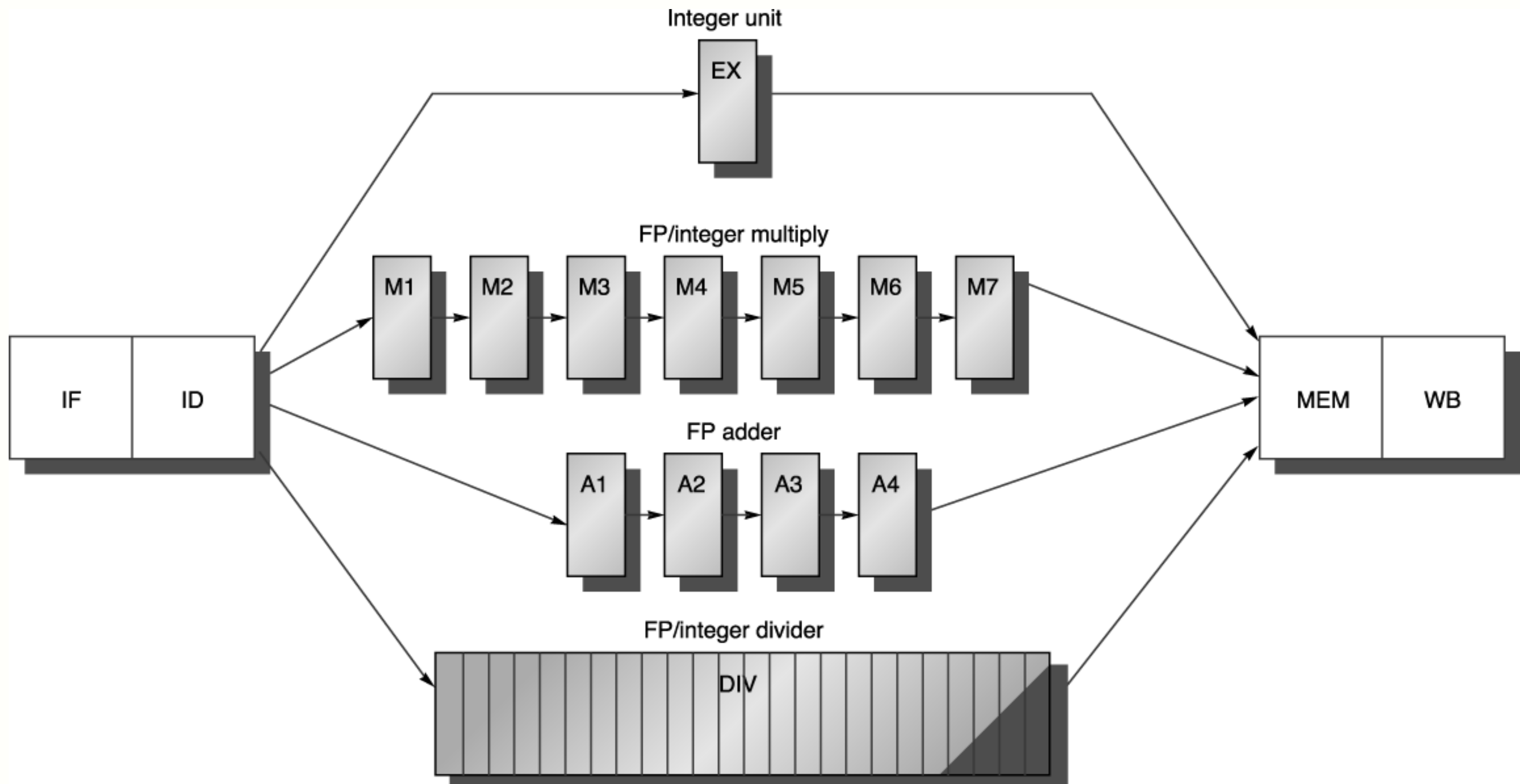
Enhanced pipelines

- The EX phase can use a *short* clock when it carries out an integer operation (add/subtraction), a logical operation (AND, OR, XOR), or a register comparison.
- The best approach is:

1. using a *short* clock cycle, that suites phases IF, ID, MEM, WB, and EX as well, when it executes a simple integer instruction
2. using functional units working with *more* clock cycles or themselves *pipelined* (the usual case) for longer operations, that take more time to execute.

Enhanced pipelines

- Pipeline with multiple functional units for the MIPS architecttture (Hennessy-Patterson, Fig. A.31):



Enhanced pipelines

- The drawing in the previous slide depicts fairly well the high level operation of modern architectures. It highlights some properties shared by all modern CPUs:

1. the length of the pipeline depends on the type of the instruction in execution, with more complex instructions requiring more clock cycles.
2. multiple instructions can be concurrently in their EX phase. In the case instance just considered, up to 4 floating point sum ops, and 7 integer of FP multiplications, all of then active.

- (A technical remark: in many cases, the division functional unit operates in more phases, but it is not pipelined)

Enhanced pipelines

- Floating point instructions in the enhanced pipeline are completed out-of-order:
  - The **POE** (Plane of Execution) prepared by the compiler is:

```
PC1    FDIV   F10,F12,F13
PC2    FMULT  F1,F2,F3
PC3    FADD   F4,F5,F6
PC4    ADD    R1,R2,R3
```

  - The **ROE** (Record of Execution), namely the completions of the instructions carried out by hw is

```
PC4    ADD    R1,R2,R3
PC3    FADD   F4,F5,F6
PC2    FMULT  F1,F2,F3
PC1    FDIV   F10,F12,F13
```

Enhanced pipelines

- Out-of-order completion is not a problem if *data dependencies are preserved*, provided *no exceptions* are raised:
  - The **POE** (Plane of Execution) prepared by the compiler is:

```
PC1    FDIV  F10,F12,F13
PC2    FMULT F1,F10,F3
PC3    FADD  F4,F5,F6
PC4    ADD   R1,R2,R3
```

  - The **ROE** (Record of Execution), namely the completions of the instructions carried out by hw is

```
PC1  FDIV  F10,F12,F13
PC4  ADD   R1,R2,R3
PC3  FADD  F4,F5,F6
PC2  FMULT F1,F10,F3
```

Enhanced pipelines

- **Exceptions** introduce huge complexities. Precise exception management is no longer feasible as it is with integer pipelines, because the MEM/WB transition does not serialize instructions in-order.
  - The **POE** (Plane of Execution) is:

```
PC1    FDIV  F10,F12,F13
PC2    FMULT F1,F2,F3 raises exception
PC3    FADD  F4,F5,F6
PC4    ADD   R1,R2,R3
```

  - The **ROE** (Record of Execution) is

```
PC4    ADD   R1,R2,R3 execution completed !
PC3    FADD  F4,F5,F6 WB
PC2    FMULT F1,F2,F3 exception serviced
PC1    FDIV  F10,F12,F13
```

Enhanced pipelines

- Floating point instructions and exceptions: **two** machine modes/states
  - a) un-precise FP exception: *high-speed* execution
  - b) precise FP exception: *slow-down* of execution
- a) Some applications tolerate un-precise FP management (mostly in graphics): they are compiled with FP libraries that do not enforce the precise management state and are executed by disregarding most exceptions (underflow, n.a.n., even overflow)

Enhanced pipelines

- b) Applications that depend on correct FP values activate the precise FP management state: they are compiled with standard FP libraries and are handled as follows:
  - The pipeline at issue time (ID) sends the FP instruction to the proper functional unit FU
  - The FU establishes in one clock cycle if the instruction will raise an exception – meanwhile the ID stage is frozen, thus effectively wasting one clock cycle (slow down)
  - If no exception will be raised, the ID stage issues the next instruction – otherwise the IF and ID are frozen until the exception has been serviced

Pipeline static scheduling

- The pipeline scheme considered so far assumes that instructions are executed **in-order** (serially, one after another), according to the PC sequence. In case of a structural or data hazard (the latter not being solved by forwarding), the pipeline is stalled. This scheme is called **static scheduling**
- Modern CPUs often adopt some type of **dynamic scheduling** of the pipeline, by changing instructions order of execution, to reduce stalls (we'll cover this subject later)

Superscalar architectures

- Architettures having multiple functional units for the EX phase, are usually called **superscalar architectures**.
- To be precise, an architecture can defined superscalar if it can fetch (and issue) in the same clock cycle multiple instructions in the IF phase. This feature is useful only if there are multiple functional units in the EX stage that can work in parallel (more on this subject when we discuss Instruction Level Parallelism)
- A superscalar architecture requires a larger datapath, capable of transferring more instructions from one stage to the next in the pipeline.

Concluding remarks

- To sum up, why are RISC architectures well matched to pipelining?

1. **All instructions have the same length**: this simplifies fetching from memory (during IF) and decoding (during ID).

- In the 80x86 ISA (also known as IA-32) instructions have variable lengths, from 1 to 17 bytes, so that fetching and decoding are more complex.

- Actually, all current implementations of IA-32 translate the instructions into internal "micro-operations" (uops), similar to the MIPS format. It is the upos that are executed in the pipeline, not native native IA-32 instructions (this will be discussed later)

Concluding remarks

2. The ISA of a typical RISC machine has a **reduced number of instruction formats (types)**, and in every format **source registers are in the same position**.

- This simplified scheme allows reading the register file while at the same time the instruction is being decoded, all of this during ID.

- If this were not possible, it would be necessary to split ID into two stages (Instruction Decode and Operand Fetch), with a longer pipeline ! (go back to figure 2.5 in Tanenbaum…)

Concluding remarks

3. In RISC architectures, **memory operands are allowed only in load and store instructions**: this constraint allows using stage EX to compute the memory address and the succeeding stage (MEM) to actually access memory.

- If memory operands could be used in other instructions (as in CISC architectures), 3 stages would be required: i) to compute the address, ii) to access memeory, iii) to carry out the operation.

- And memory is indeed THE bottleneck in the execution of instructions !