The processor

pipelining

Introduction

- the performance of a computer is determined by three key: instruction count, clock cycle time, and clock cycles per instruction (CPI).
- the compiler and the instruction set architecture determine the instruction count required for a given program.
- the implementation of the processor determines both the clock-cycle time and the number of clock cycles per instruction.

In this chapter, we construct the datapath and control unit. Explanation of the principles and techniques used in implementing a processor.

Building a Datapath

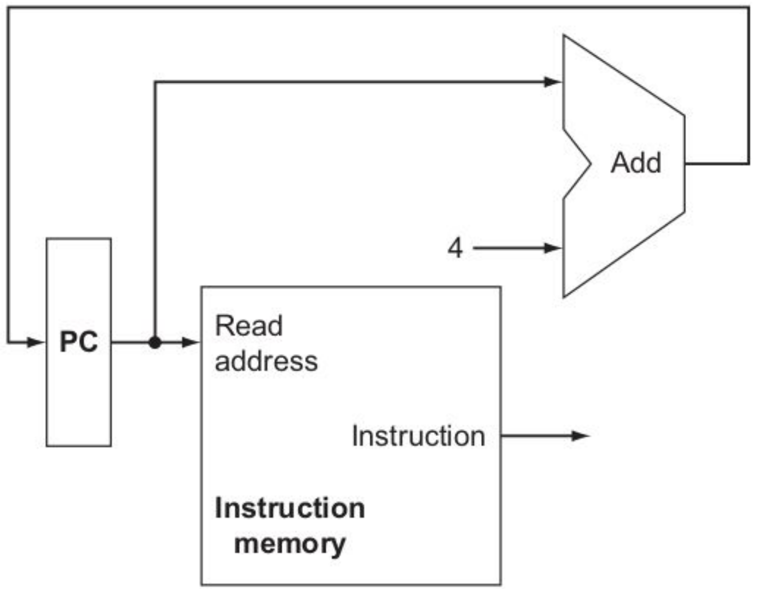
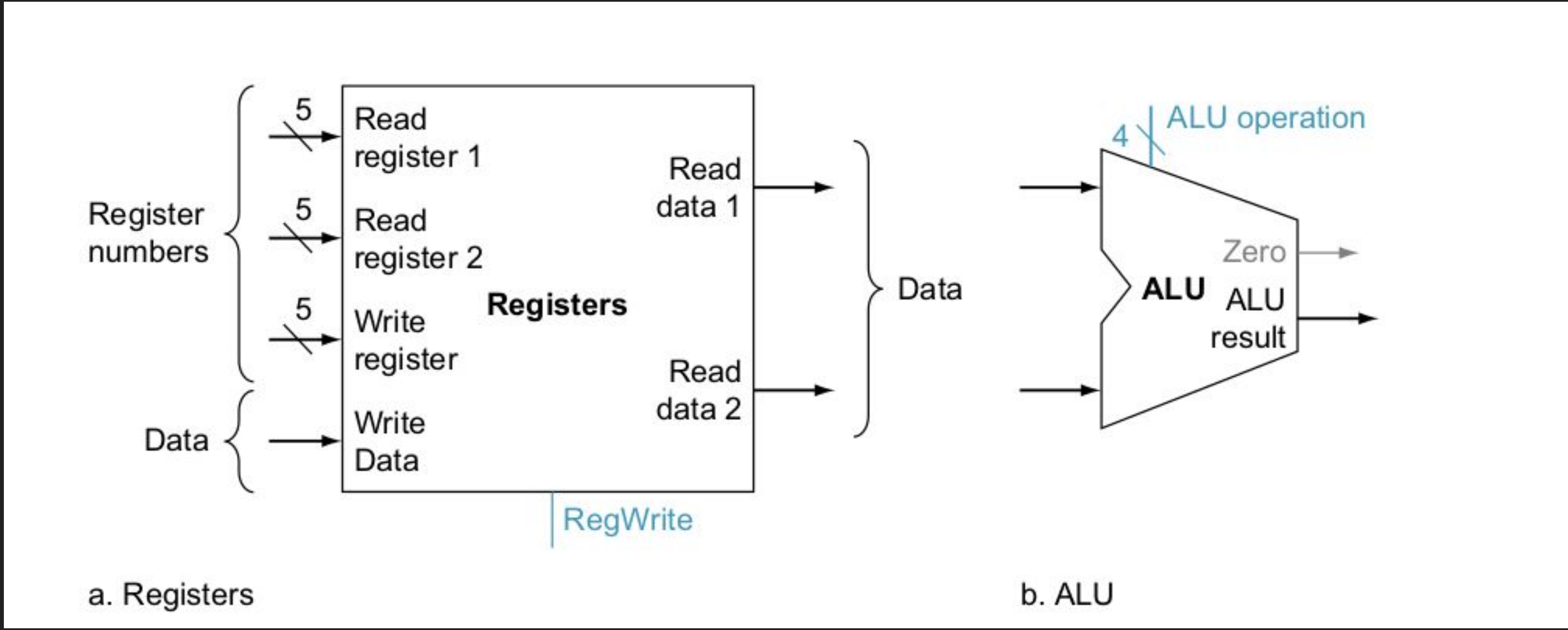


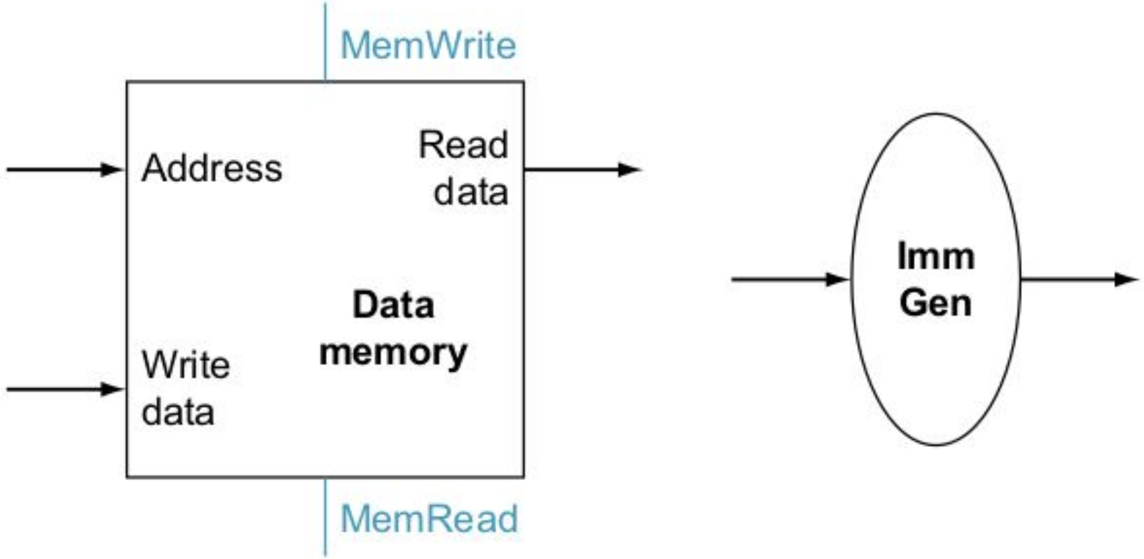
FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

R-type



Register numbers
5
5
5
Read register 1
Read register 2
Write register
Write Data
Data
Registers
Read data 1
Read data 2
Data
RegWrite
a. Registers
4
ALU operation
ALU
Zero
ALU result
b. ALU

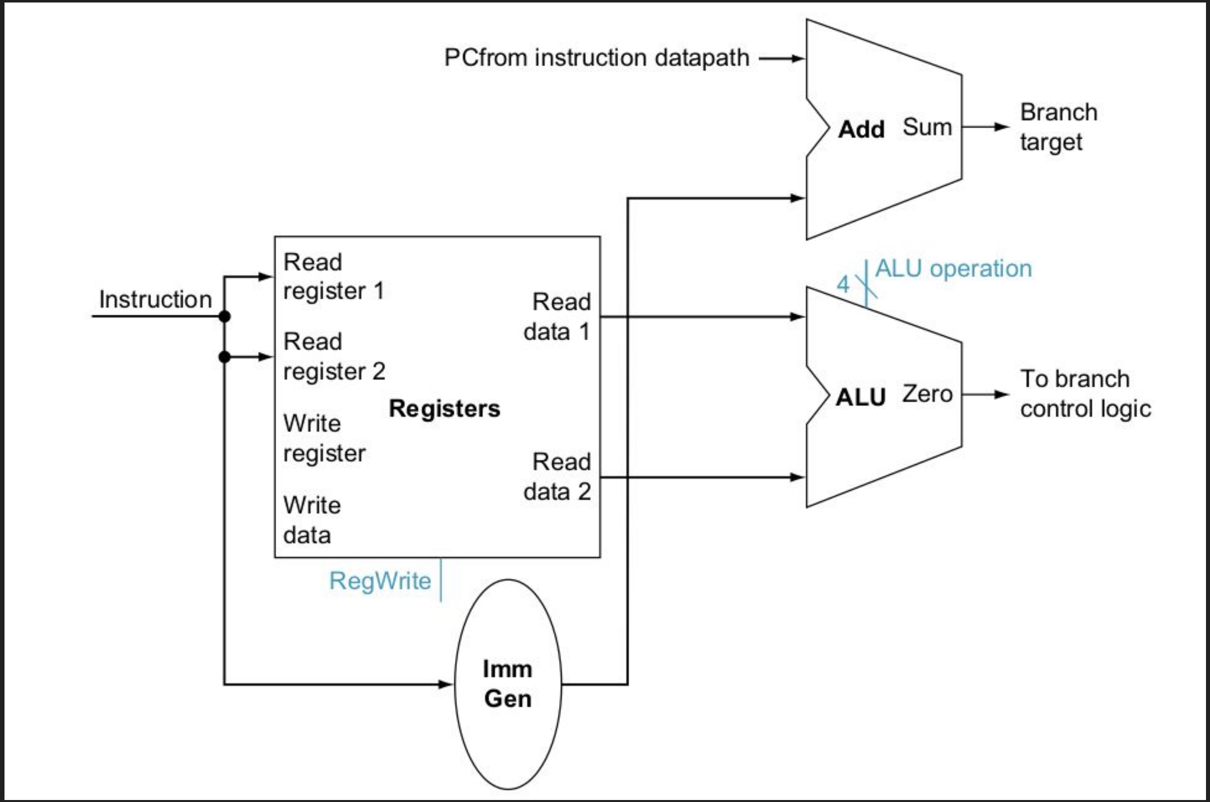
Load - Store



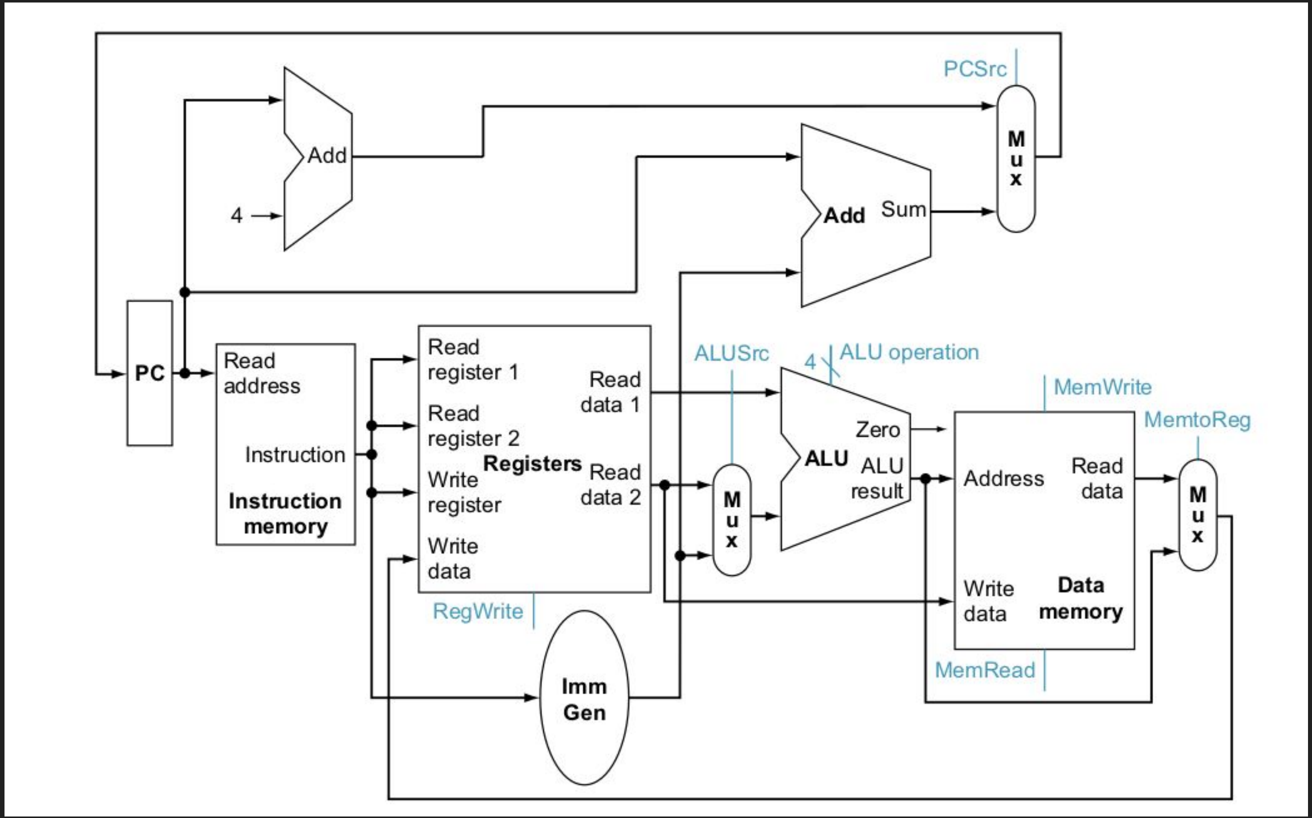
a. Data memory unit

b. Immediate generation unit

Branch


PCfrom instruction datapath
Add
Sum
Branch target
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Instruction
RegWrite
Imm Gen
4
ALU operation
ALU
Zero
To branch control logic

Simple Datapath for the core RISC-V



Simple Implementation scheme

We build this simple implementation using the datapath of the last section and adding a simple control function. This simple implementation covers:

- load word (lw)
- store word (sw)
- branch if equal (beq)
- arithmetic-logical instructions add, sub, and, and or.

Control

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

ALU control

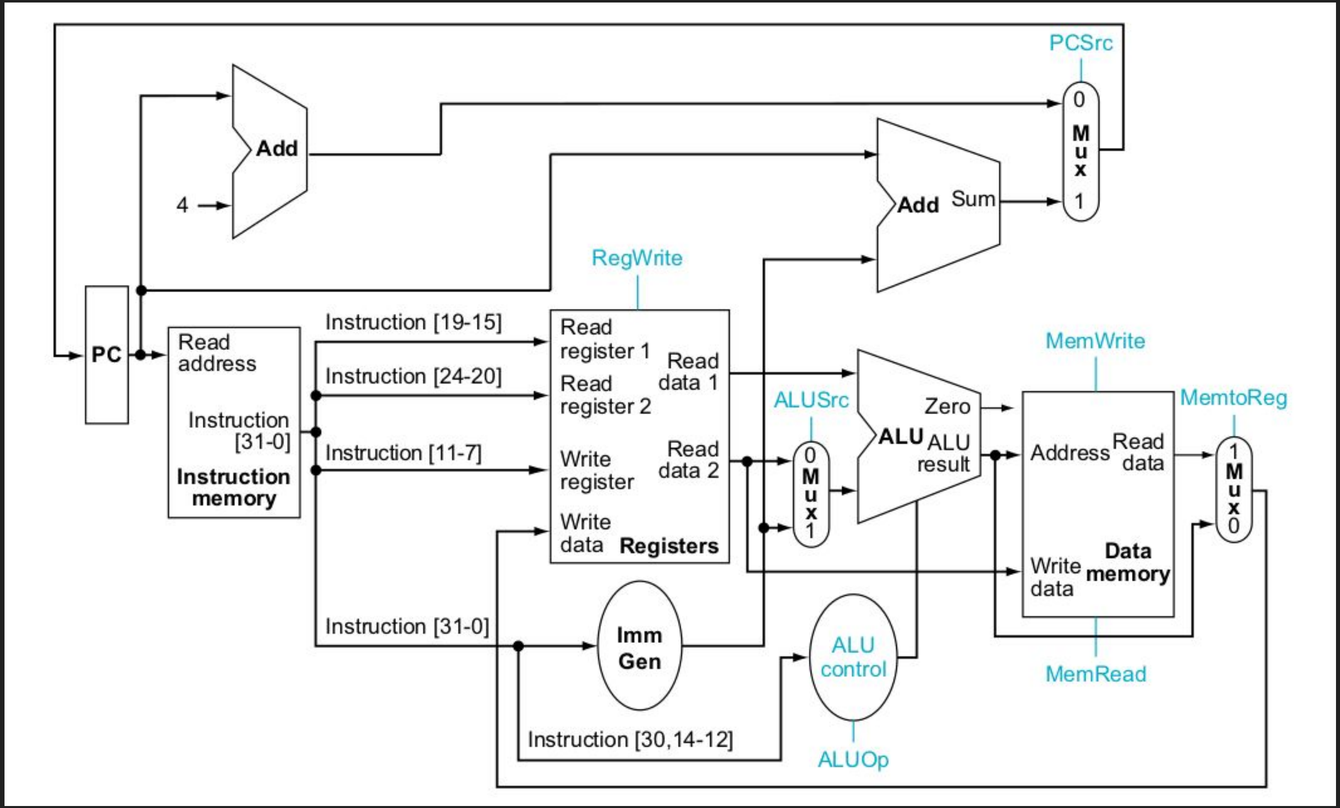
- load/store: Compute mem address
- R-type: AND, OR, add or subtract
- branch: subtract and tests zero

We can generate the 4-bit ALU control input using a small control unit that has as inputs the funct7 and funct3 fields of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, subtract and test if zero (01) for beq, or be determined by the operation encoded in the funct7 and funct3 fields (10).

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALU control



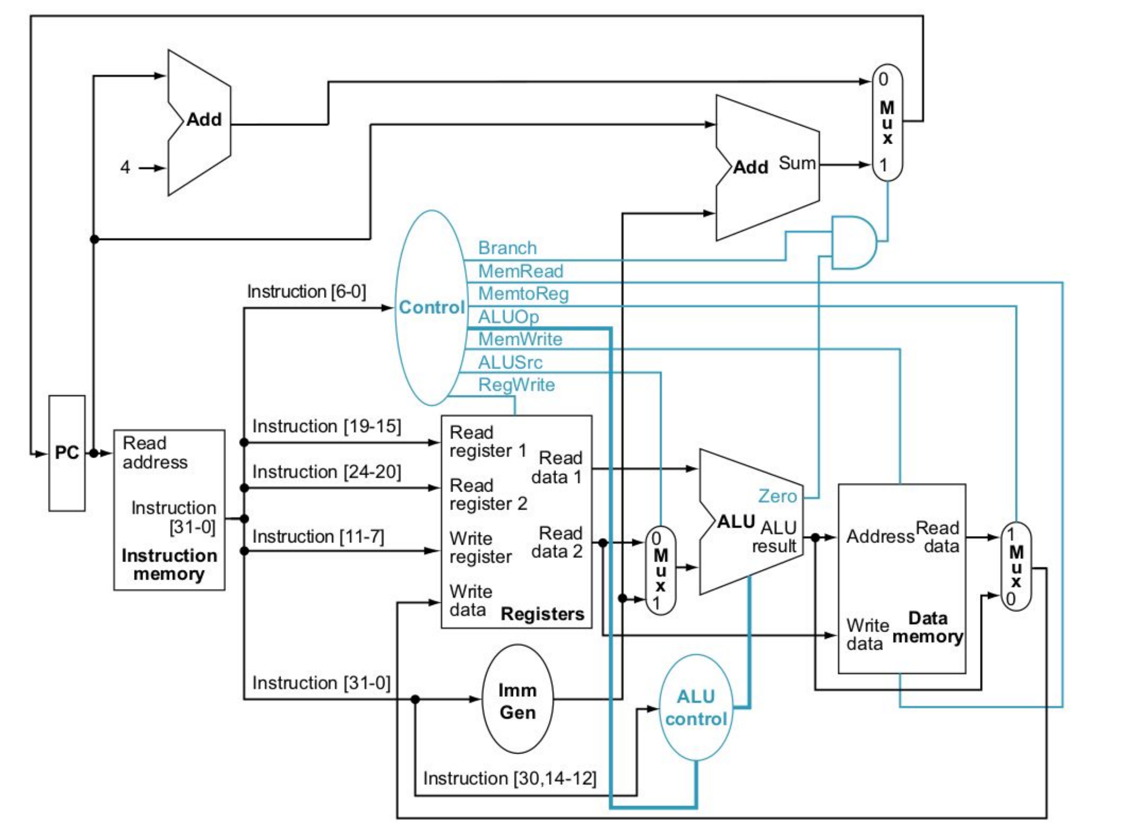
PCSrc
0
Mux
1
Add
4
Add
Sum
RegWrite
PC
Read address
Instruction [31-0]
Instruction memory
Instruction [19-15]
Instruction [24-20]
Instruction [11-7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALUSrc
0
Mux
1
ALU
Zero
ALU result
MemWrite
Address
Read data
Write data
Data memory
MemtoReg
1
Mux
0
MemRead
Instruction [31-0]
Imm Gen
ALU control
ALUOp
Instruction [30,14-12]

Control unit

- All signals (but PCSrc) can be obtained from opcode and funct.
- PCSrc: branch AND zero.

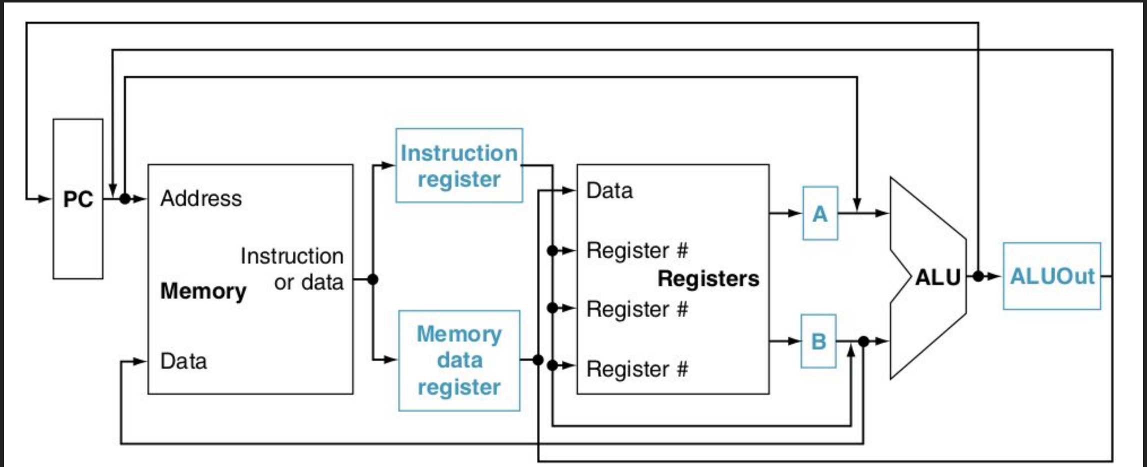
Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

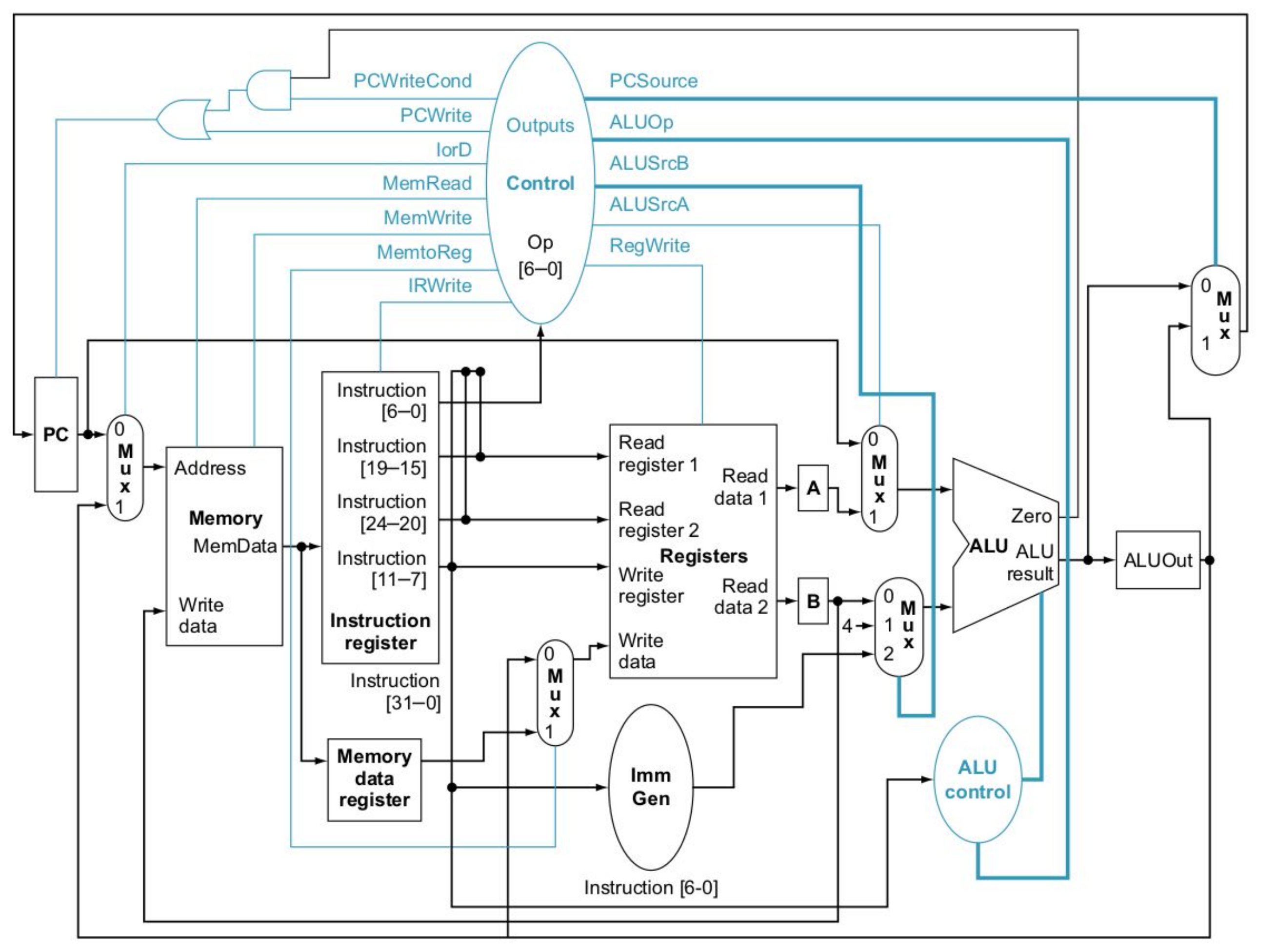
high-level view of a RISC-V implementation



Multicycle Implementation

- Single-cycle is too inefficient.
- The longest possible path determines the clock cycle (Single-cycle).
- Each step in the execution take 1 clock cycle.
- Allows functional units to be used more than once per instruction.
- Single memory unit
- Single ALU
- Registers to hold the output.




PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[6–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
PC
Mux
Address
Memory
MemData
Write data
Instruction [6–0]
Instruction [19–15]
Instruction [24–20]
Instruction [11–7]
Instruction register
Instruction [31–0]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
Imm Gen
Instruction [6-0]
Zero
ALU
ALU result
ALUOut
ALU control

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by the value on the Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the immediate generated from the IR.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

Breaking the execution into clock cycles

- goal in breaking the execution into clock cycles should be to maximize performance.
- breaking the execution of any instruction into a series of steps, each taking one clock cycle, attempting to keep the amount of work per cycle roughly equal.
- Recall that at the end of every clock cycle any data values that will be needed on a subsequent cycle must be stored into a register

1. Instruction fetch step

Fetch the instruction from memory and compute the address of the next sequential instruction:

```
IR <= Memory[PC];

PC <= PC + 4;
```

2. Instruction decode and register fetch step

```
A <= Reg[IR[19:15]];

B <= Reg[IR[24:20]];

ALUOut <= PC + immediate;
```

3. Execution, memory address computation, or branch completion

Memory reference: immediate

```
Immediate
```

Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

Branch:

```
if (A == B) PC <= ALUOut;
```

4. Mem access or R-type completion

Memory reference:

MDR <= Memory [ALUOut];

Memory [ALUOut] <= B;

Arithmetic-logical instruction (R-type):

Reg[IR[11:7]] <= ALUOut;

5. Mem read completion

Load:

```
Reg[IR[11:7]] <= MDR;
```

Step name	Action for R-type instructions	Action for memory reference instructions	Action for branches
Instruction fetch	IR <= Memory[PC] PC <= PC + 4		
Instruction decode/register fetch	A<= Reg [IR[19:15]] B <= Reg [IR[24:20]] ALUOut <= PC + immediate		
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + immediate	if (A == B) PC <= ALUOut
Memory access or R-type completion	Reg [IR[11:7]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B	
Memory read completion		Load: Reg[IR[11:7]] <= MDR	

CPI in multicycle

- Using the SPECINT2006 instruction mix
- 20% loads, 8% stores, 10% branches, and 62% ALU
- clock cycles for each instruction
  - loads: 5
  - stores: 4
  - ALU: 4
  - branch: 3

CPI = $\sum$ Instruction count i × CPI i / Instruction count

CPI = $0.20 \times 5 + 0.08 \times 4 + 0.62 \times 4 + 0.10 \times 3 =$ **4.10**

Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, pipelining is nearly universal.
- it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- The time required between moving an instruction one step down the pipeline is a processor cycle. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage

La idea es obtener un CPI de 1, por cada ciclo saco una instrucción
Como hago esto, superponiendo varias instrucciones en ejecucion

Para ello debo hacer que cada etapa dure lo mismo, lo cual la va a determinar la etapa mas lenta then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

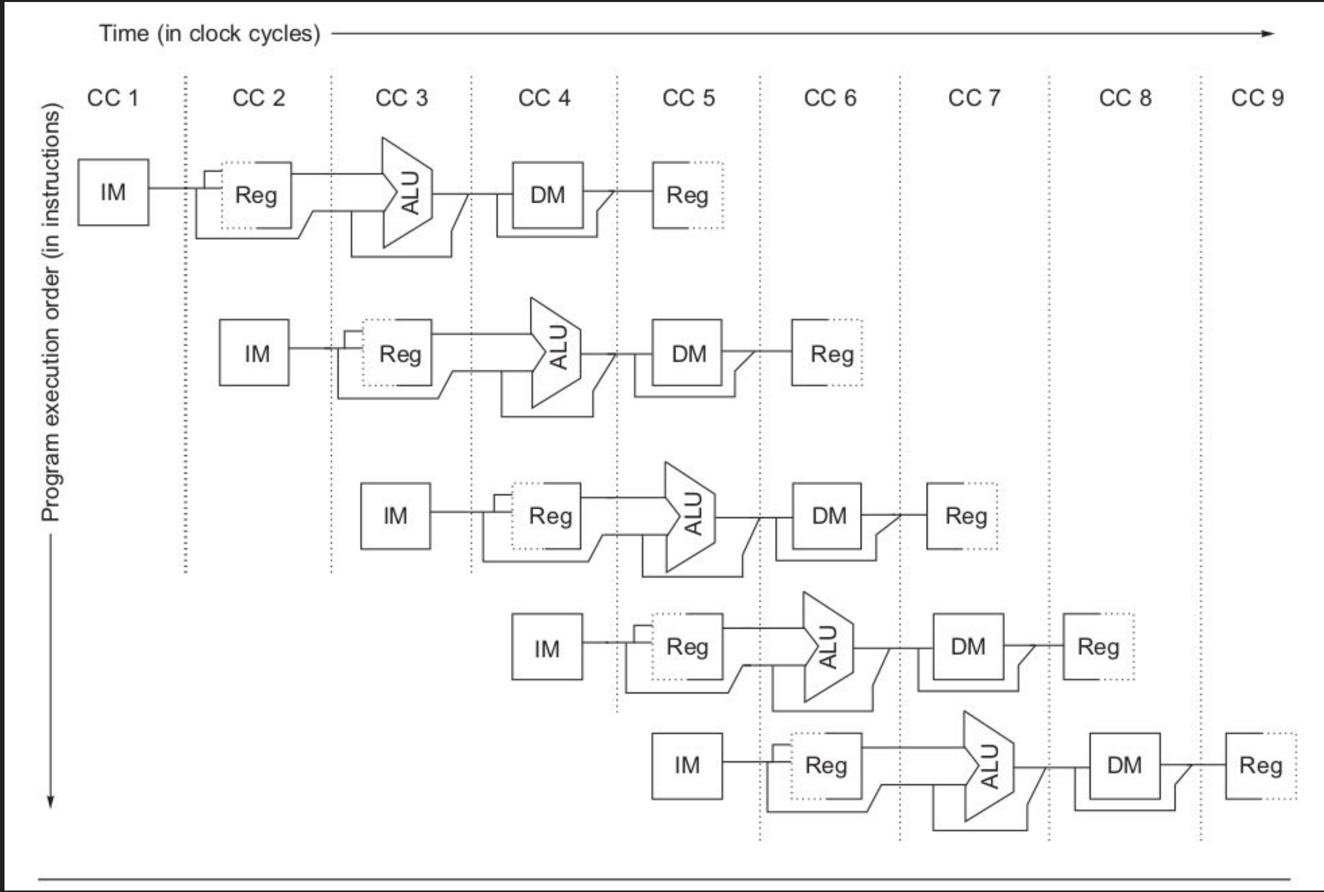
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

the speedup from pipelining equals the number of pipe stages, just as an assembly

The Classic Five-Stage Pipeline for a RISC Processor

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Aqui en el 5to ciclo ya entra en régimen, tengo la pipeline llena
y de aqui saco una intstruccion por ciclo


Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
IM
Reg
ALU
DM
Reg

single cycle vs pipeline

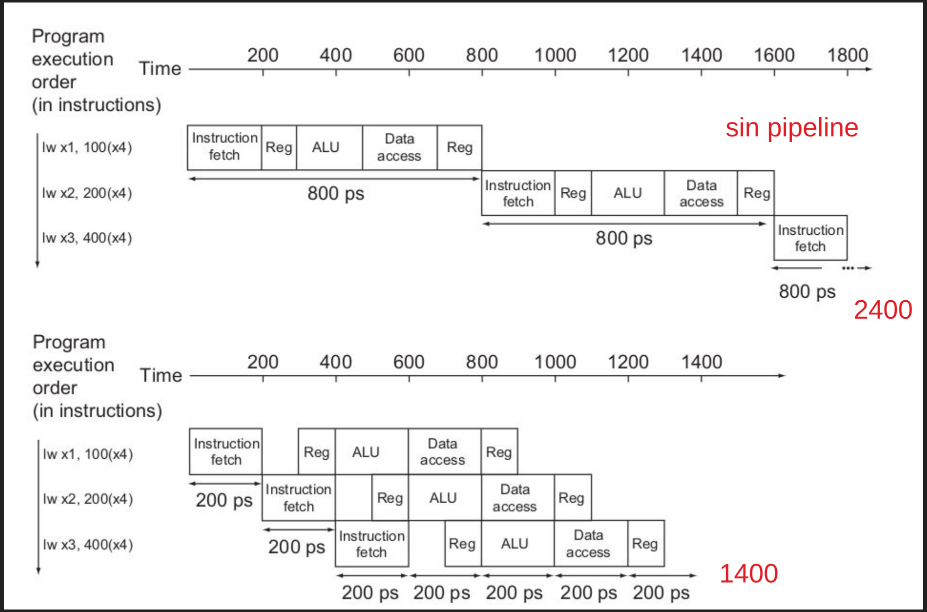
todas tienen que tener el mismo tiempo por ciclo y también todas tienen que tener la misma cantidad de ciclos

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

no necesito usar registros

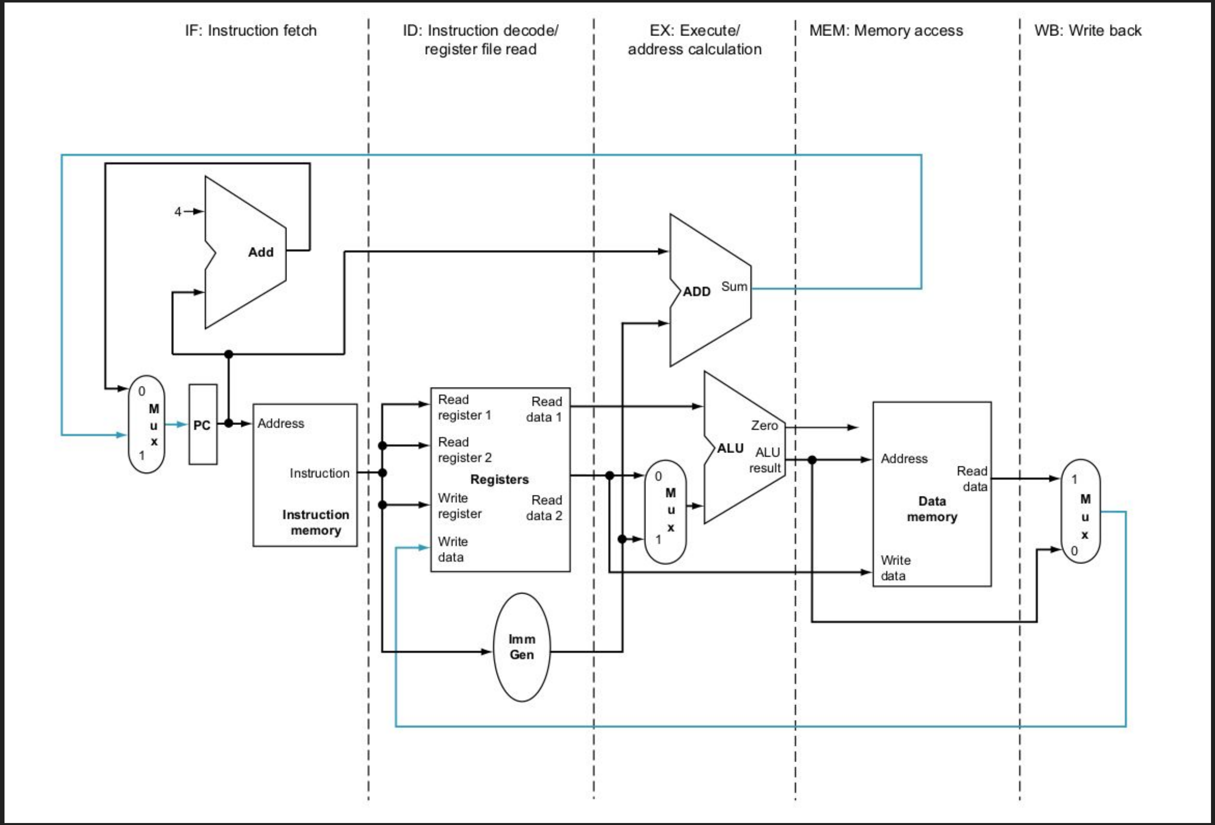
no acceden a memoria

Pipelining improves performance by increasing instruction throughput, in contrast to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions.



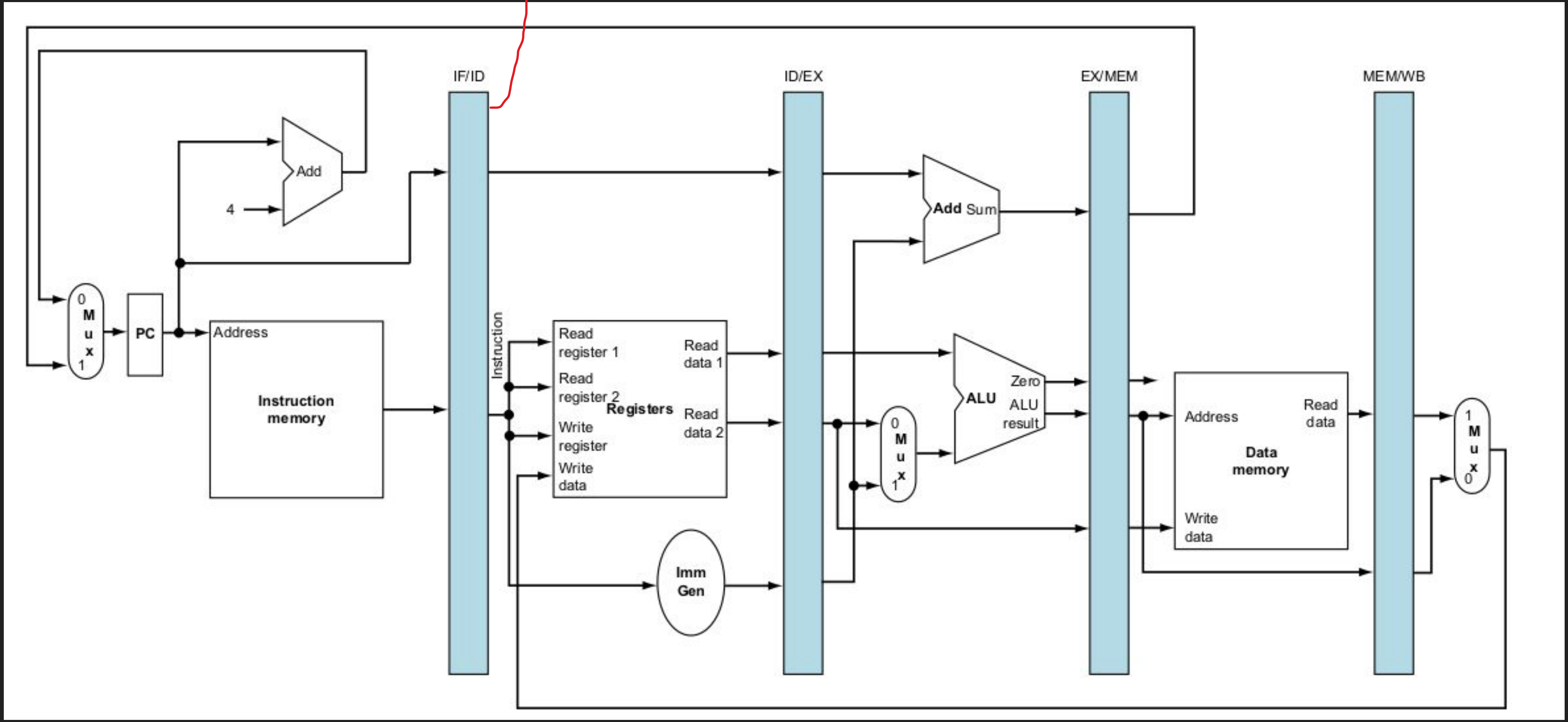
se aumenta el tiempo de ejecucion por instruccion,
porque todas las etapas tienen que ser iguales

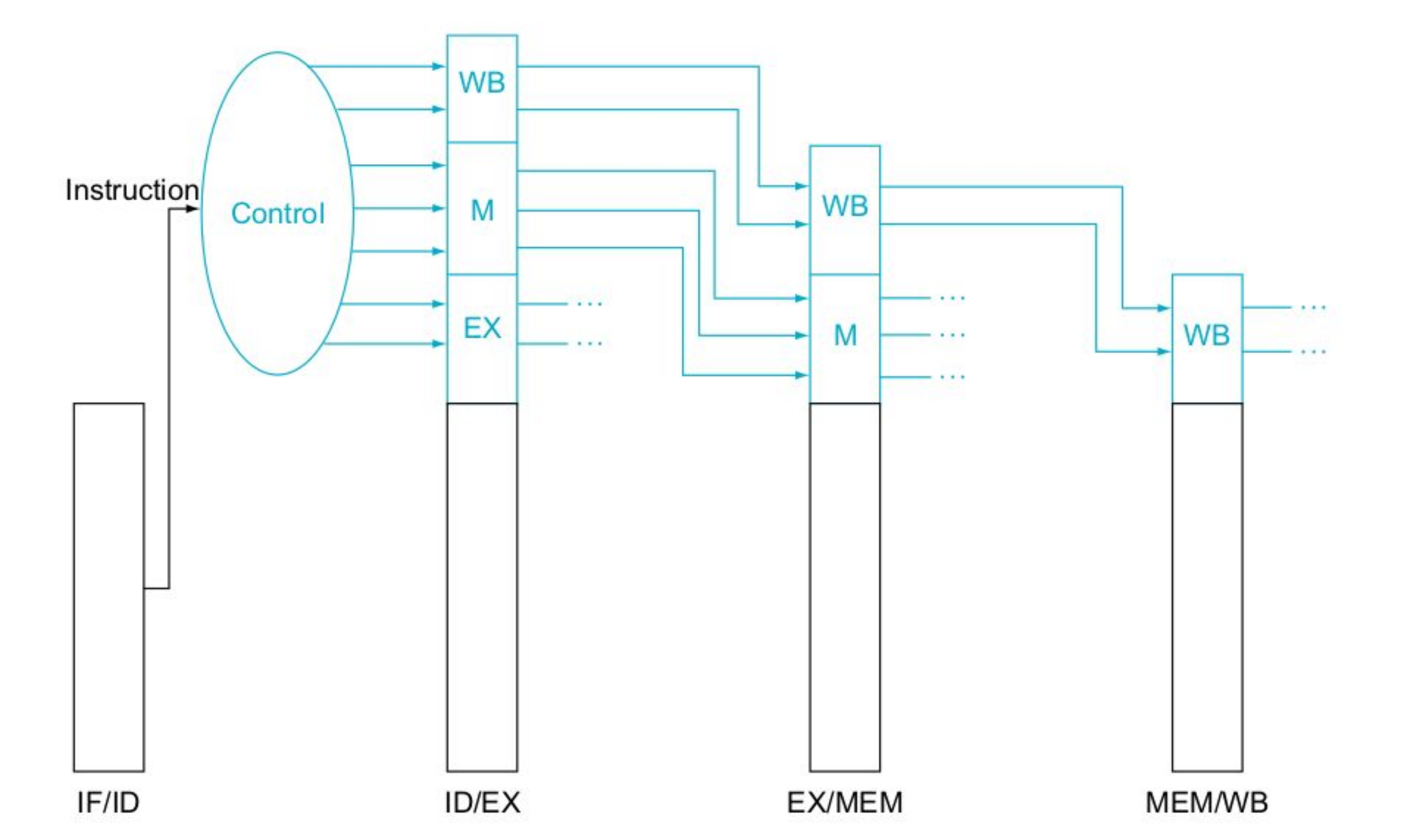
Pipelined Datapath and Control



- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages.
- The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

cantidad de latches dependiendo de cantidad de bits que pasan de etapa a etapa





las señales de control siguen en cada etapa

Pipeline Hazards

nos impiden tener el pipeline lleno

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards, and there are three different types.

- Structural hazards
- Data hazards
- Control hazards

Performance of Pipelines With Stalls

Hazards in pipelines can make it necessary to stall the pipeline.

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

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

Structural Hazards

recursos de hardware que no puedo reutilizar al mismo tiempo

the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

- Memory: IF - MEM
- Registers: ID - WB
- ALU: Exec

Separamos la memoria --> caché de instrucciones y caché de datos

leer los registros y escribirlos, si tenemos distintas instrucciones, una en cada una de estas etapas. van a intentar acceder a los registros al mismo tiempo. Lo que hacemos es hacer cada una de estas operaciones en distintos flancos del clock

Instruccion de salto y Calcular la instruccion, entonces lo que hacemos es poner 2 ALUs (3 en total mas la que suma una al PC)

Data hazards

- Arise from the dependence of one instruction on an earlier one that is still in the pipeline. que quieren escribir o leer el mismo registro
- Assume instruction i occurs in program order before instruction j and both instructions use register x, then there are three different types of hazards that can occur between i and j:
  - Read After Write (RAW) en el pipeline se escribe en una de las ultimas etapas y leo en la segunda
  - Write After Read (WAR) -> anti dependence por ahora no molesta
  - Write After Write (WAW) -> out dependence

- WAW (write after write)—j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.
- WAR (write after read)—j tries to write a destination before it is read by i, so i incorrectly gets the new value. WAR hazards cannot occur in most static issue pipelines, because all reads are early. A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline, or when instructions are reordered.

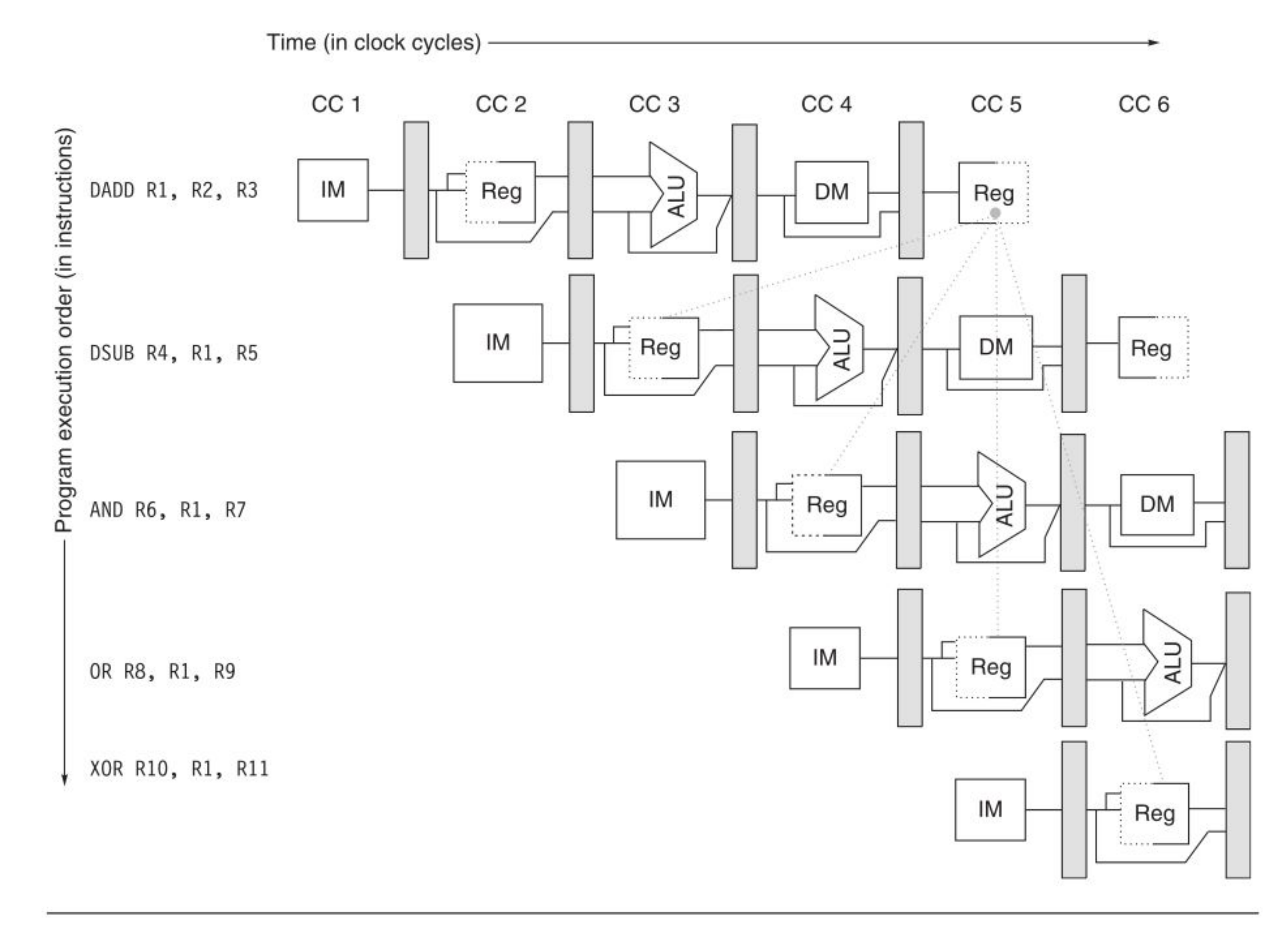
RAW hazards

Consider the pipelined execution of these instructions:

todas del tipo R y usan x1

```
add x1,x2,x3
sub x4,x1,x5
and x6,x1,x7
or x8,x1,x9
xor x10,x1,x11
```

en los tipo R la etapa 4 no sirve



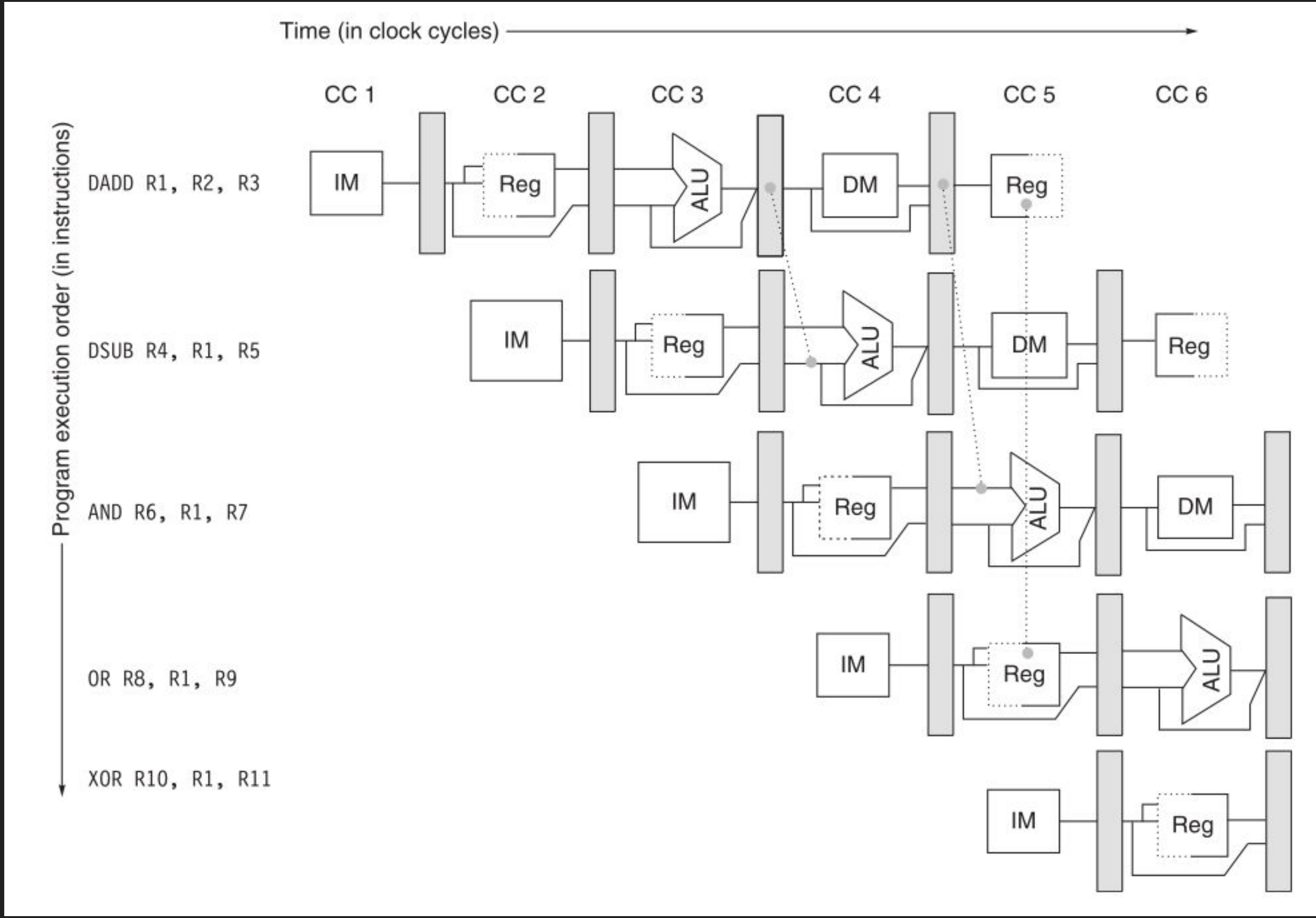
Minimizing Data Hazard Stalls by Forwarding

the result is not really needed by the sub until after the add actually produces it. If the result can be moved from the pipeline register where the add stores it to where the sub needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

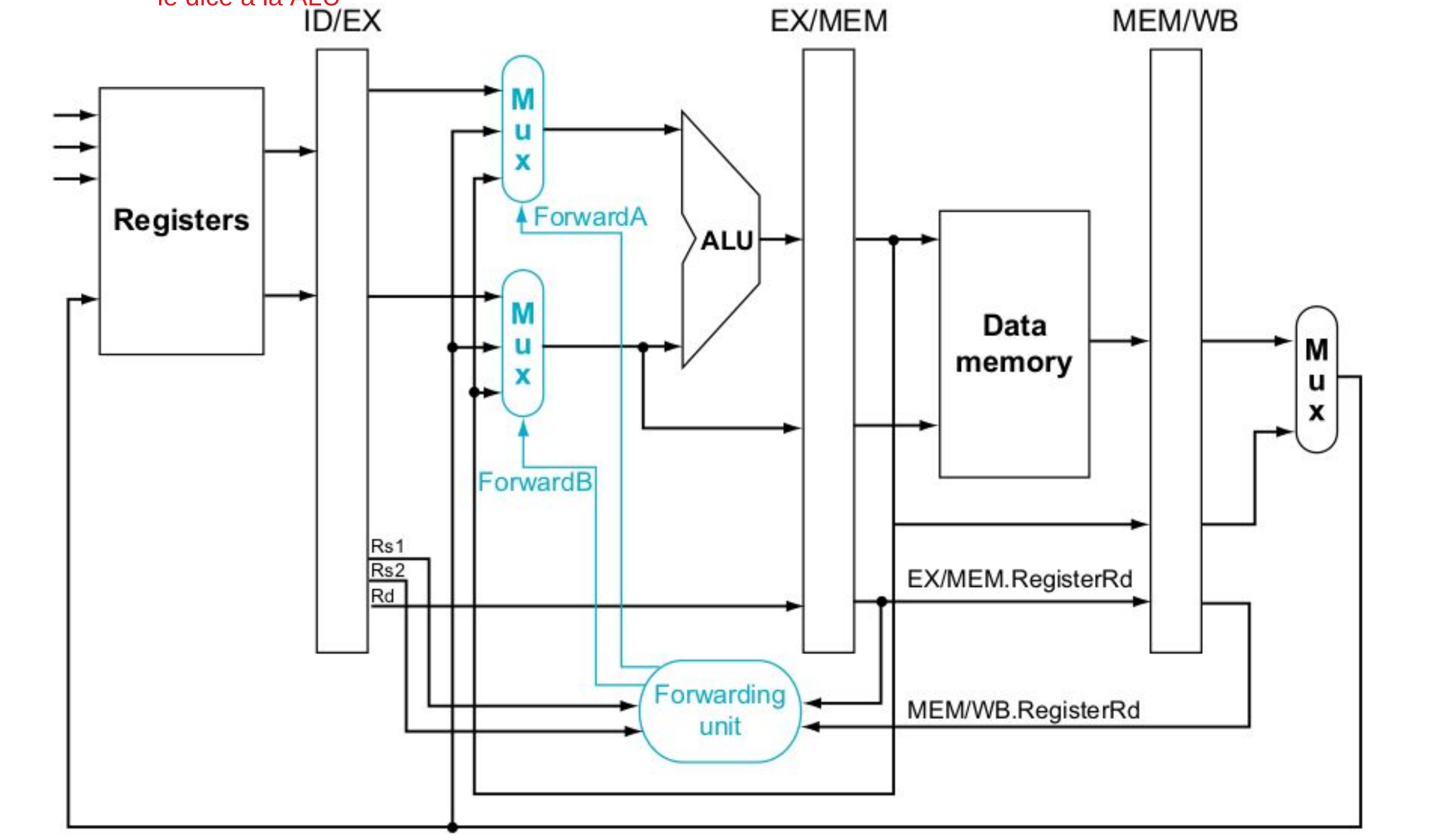
forwarding



Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

Si la forwarding unit detecta que tiene que hacer forwarding, activa el MUX le da a la ALU, directamente la salida de la ALU de la instrucción anterior

le dice a la ALU



b. With forwarding

1. EX hazard:

if (EX/MEM.RegWrite

and (EX/MEM.RegisterRd ≠ 0)

Esto significa que va a escribir un registro,
Tiene que ser una operacion de r/w si o si (tipo r y load)

and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

Que el registro que quiero escribir sea uno de los source de la siguiente instrucción

if (EX/MEM.RegWrite

and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10

2.MEM hazard:

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

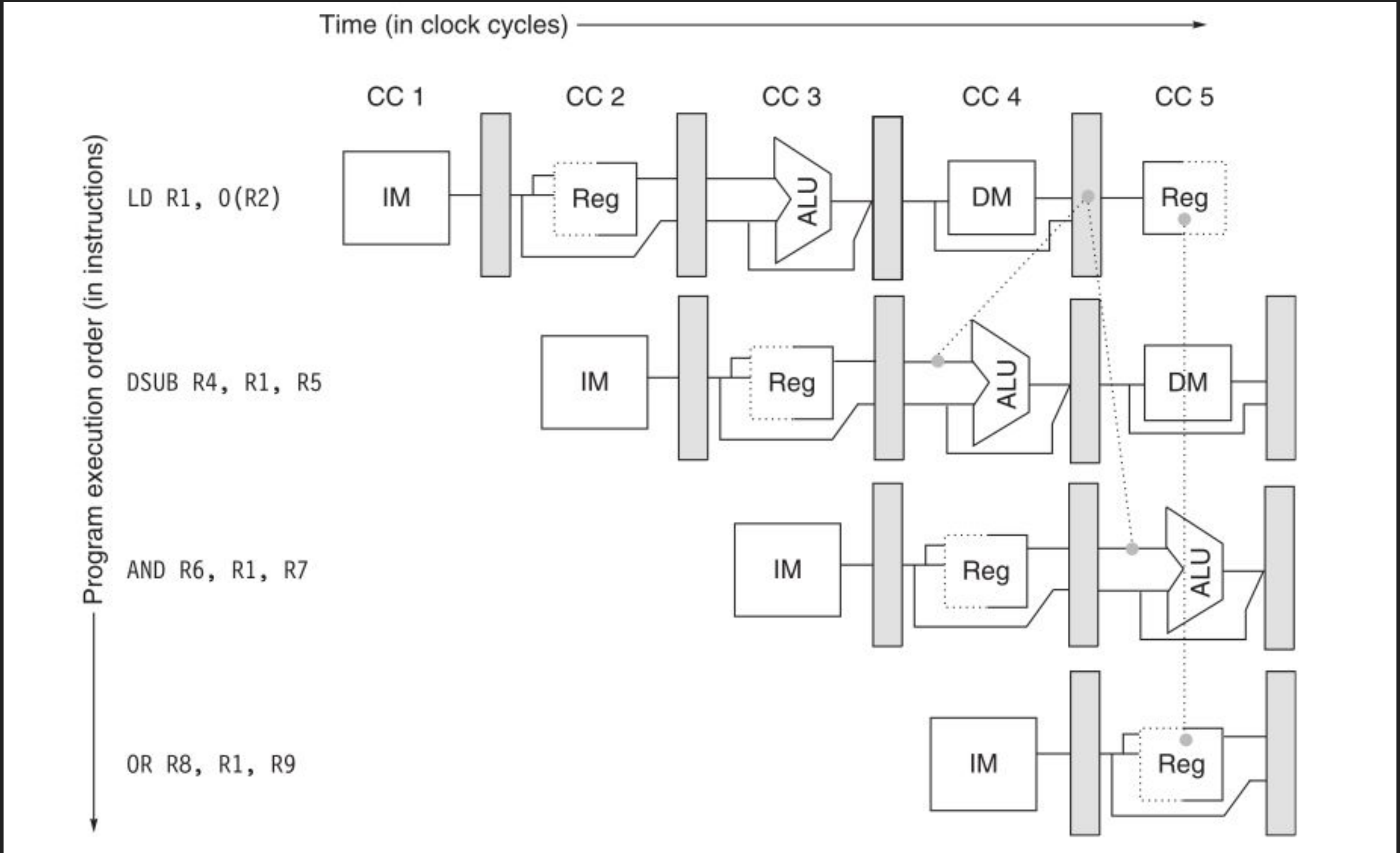
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
ld x1,0(x2)
sub x4,x1,x5
and x6,x1,x7
or x8,x1,x9
```

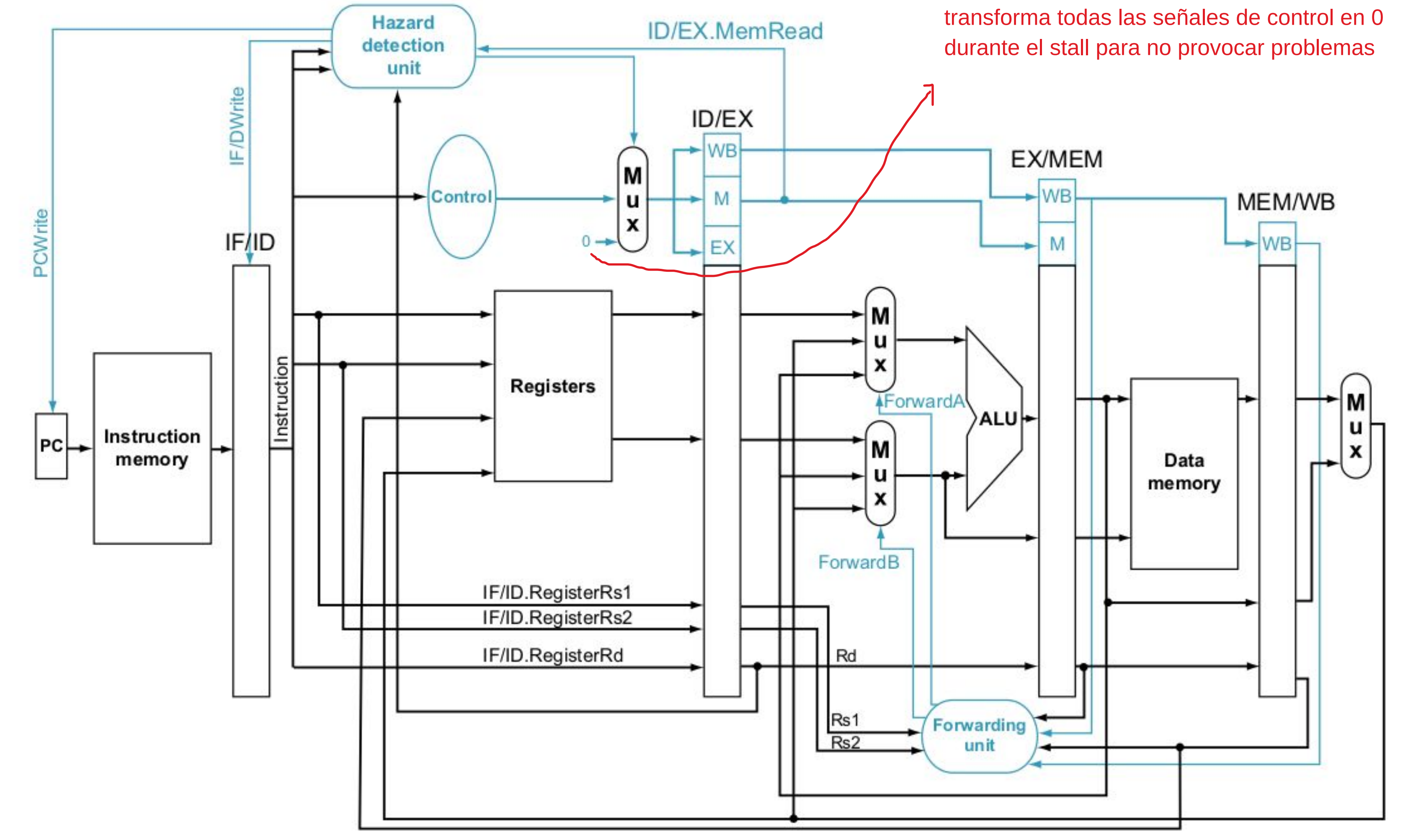
el load se hace al final de la etapa 4 y la siguiente instruccion necesita ese registro para su etapa 3, por lo que no podemos hacer un forwarding linea punteada hacia la izq es imposible porque estariamos viajando en el tiempo



- The load instruction has a delay or latency that cannot be eliminated by forwarding alone.
- we need to add hardware, called a pipeline interlock, to preserve the correct execution pattern.
- a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.
- This pipeline interlock introduces a stall or bubble.
- The CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case).

`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	EX	MEM	WB			
`and x6,x1,x7`			IF	ID	EX	MEM	WB		
`or x8,x1,x9`				IF	ID	EX	MEM	WB	
`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	Stall	EX	MEM	WB		
`and x6,x1,x7`			IF	Stall	ID	EX	MEM	WB	
`or x8,x1,x9`				Stall	IF	ID	EX	MEM	WB

paramos por un ciclo esa instruccion y todas las que siguen transforma todas las señales de control en 0 durante el stall para no provocar problemas



Control hazards

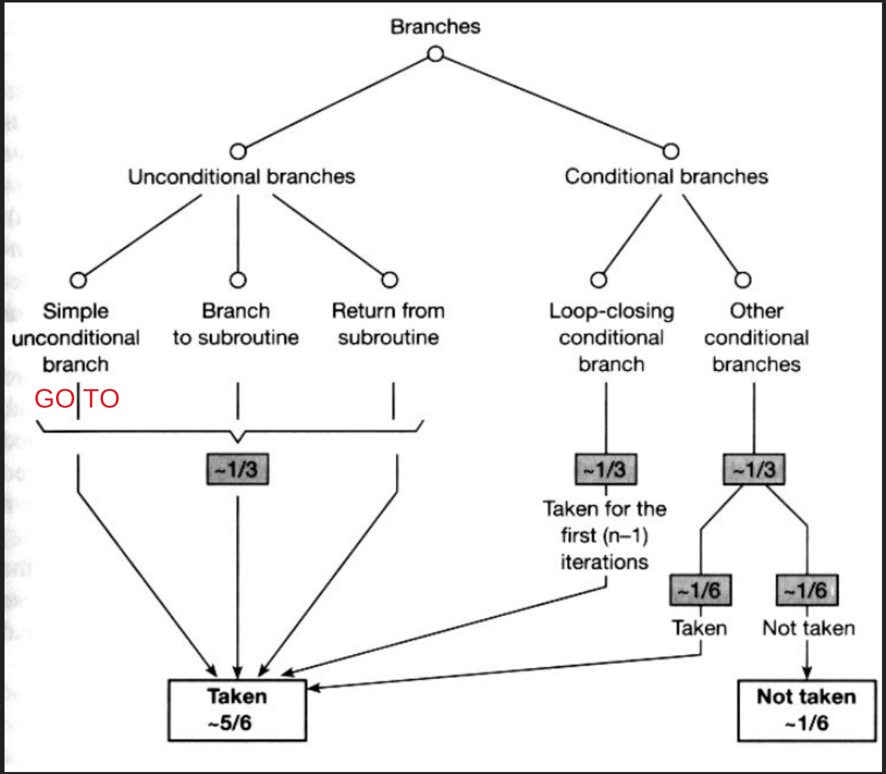
primero tengo que evaluar si tengo que saltar y en caso de saltar tengo que calcular la instrucción

- Control hazards can cause a greater performance loss for our RISC V pipeline than do data hazards.
- When a branch is executed, it may or may not change the PC to something other than its current value plus 4.
- If instruction i is a taken branch, then the PC is usually not changed until the end of ID, after the completion of the address calculation and comparison.

branch types

Unconditional branches:
No molestan porque ya se que voy a saltar

Condicional branches:
No sabemos si vamos a saltar o no



la mayoria de los saltos son tomados

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay

- four simple compile time schemes
- hardware-based schemes that dynamically predict branch behavior
- speculation.

the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID.

repito el IF de la instruccion que sigue despues de la del branch, porque en ese momento ya voy a saber si tomo el salto o no

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency

Entonces tenemos un stall por cada instrucción de salto lo que significa una pérdida del 10 al 30 %de rendimiento predicted-not-taken or predicted-untaken

- treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed.
- implemented by continuing to fetch instructions as if the branch were a normal instruction. If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target $+1$				IF	ID	EX	MEM	WB	
Branch target $+2$					IF	ID	EX	MEM	WB

predicted-taken

- treat every branch as taken.
- As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target.
- This buys us a one-cycle improvement when the branch is actually taken, because we know the target address at the end of ID, one cycle before we know whether the branch condition is satisfied in the ALU stage.

pierdo un ciclo cuando me equivoco

delayed branch

ponemos una instruccion independiente del salto a continuacion del salto y la que depende del salto la tiramos mas adelante

The sequential successor is in the branch delay slot. This instruction is executed whether or not the branch is taken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction $(i+1)$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction $(i+1)$		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

Exceptions

- I/O device request
- Invoking an operating system service from a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

procesador -> sincronas

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Exceptions in RISC V

	Pipeline stage	Problem exceptions occurring	
Instrucrion Fetch	IF	Page fault on instruction fetch; misaligned memory access; memory protection violation	
Instruction decode	ID	Undefined or illegal opcode	
Execution	EX	Arithmetic exception	division por 0, overflow
Memoria	MEM	Page fault on data fetch; misaligned memory access; memory protection violation	
	WB	None	

- the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state.
- Restarting is usually implemented by saving the PC of the instruction at which to restart.
- When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:
  - 1. Force a trap instruction into the pipeline on the next IF.
  - 2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline; this can be done by placing zeros into the pipeline latches of all instructions in the pipeline
  - 3. After the exception-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

Paro los writes de esa instruccion que tiro excepcion y de las que siguen

Precise exceptions

deje terminar las instrucciones previas antes de manejar las excepcion, pero las que siguen no pueden seguir, es decir, no pueden escribir ni los registros ni las memorias

- If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have precise exceptions.
- Exceptions may occur out of order; that is, an instruction may cause an exception before an earlier instruction causes one
- The pipeline cannot simply handle an exception when it occurs in time, because that will lead to exceptions occurring out of the unpipelined order.

Precise exceptions

- the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction.
- The exception status vector is carried along as the instruction goes down the pipeline.
- Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off (this includes both register writes and memory writes).
- When an instruction enters WB (or is about to leave MEM), the exception status vector is checked. If any exceptions are posted, they are handled in the order in which they would occur in time on an unpipelined processor

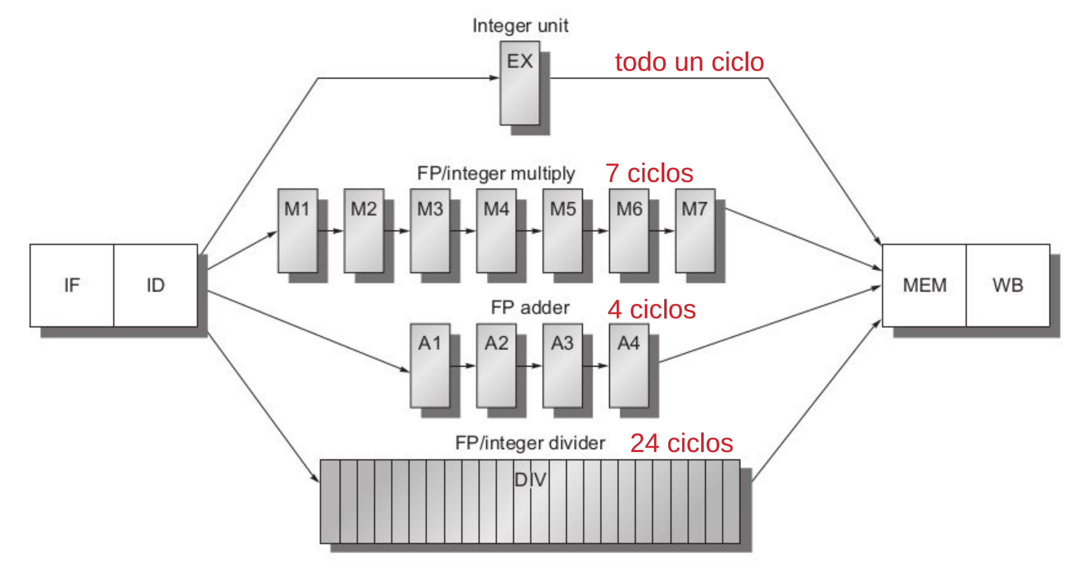
cada instruccion teien un vectorsito de 4 bits que son las flags de excepcion de cada etapa. Cuando esa instruccion llega a la ultima etapa WB, se chequea ese registro y se atiende la instruccion, entonces con esta lógica se atienden las mas viejas

Extending the RISC V Integer Pipeline to Handle Multicycle Operations

- FP pipeline will allow for a longer latency for operations.
- the EX cycle may be repeated as many times as needed to complete the operation
- the number of repetitions can vary for different operations.

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

las instrucciones de suma y multiplicacion son segmentadas, es decir la puedo meter en el pipeline, puedo seguir sacando varias instrucciones por ciclo, en cambio, la division no



Hazards and Forwarding in Longer Latency Pipelines

las instrucciones pueden terminar fuera de orden

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
fadd.d			IF	ID	*EX*	**MEM**	WB				
fsd				IF	ID	*EX*	**MEM**	WB			

- Because the divide unit is not fully pipelined, structural hazards can occur.
- varying running times, the number of register writes required in a cycle can be larger than 1. la solucion son stalls
- Write after write (WAW) hazards are possible y WAR
- nstructions can complete in a different order than they were issued, causing
- problems with exceptions

también podemos ver que hay instrucciones imprecisas, porque la excepcion se pudede dar en la instuccion mas vieja y ya las siguientes pueden haber terminado