Resumen Patterson.

Capítulo 6.

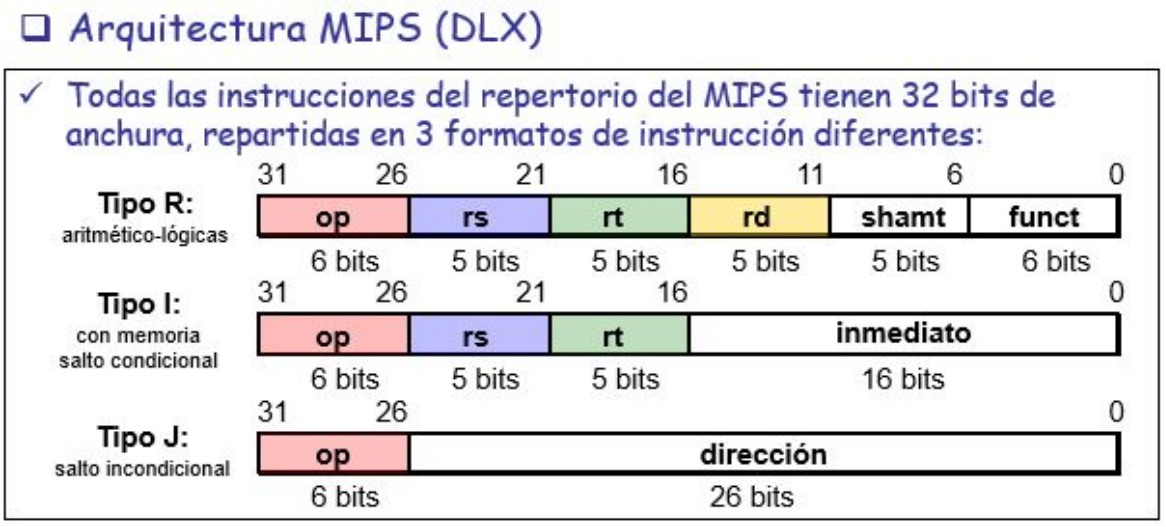
6.1 An Overview of Pipelining

ETAPAS:

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
MIPS instructions classically take five steps:
1. Fetch instruction from memory.
2. Read registers while decoding the instruction. The format of MIPS instructions allows reading and decoding to occur simultaneously
3. Execute the operation or calculate an address.
4. Access an operand in data memory.
5. Write the result into a register

Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction

ISA



Designing Instruction Sets for Pipelining

First, all MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the IA-32, where instructions vary from 1 byte to 17 bytes, pipelining is considerably more challenging. As we saw in Chapter 5, all recent implementations of the IA-32 architecture actually translate IA-32 instructions into simple microoperations that look like MIPS instructions second stage.

Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages. We will shortly see the downside of longer pipelines.

Third, memory operands only appear in loads or stores in MIPS. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage. If we could operate on the operands in memory, as in the IA-32, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Fourth, as discussed in Chapter 2, **operands must be aligned in memory.** Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle

Structural Hazards

The first hazard is called a structural hazard. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.
In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline

add $s0, $t0, $t1
sub $t2, $s0, $t3

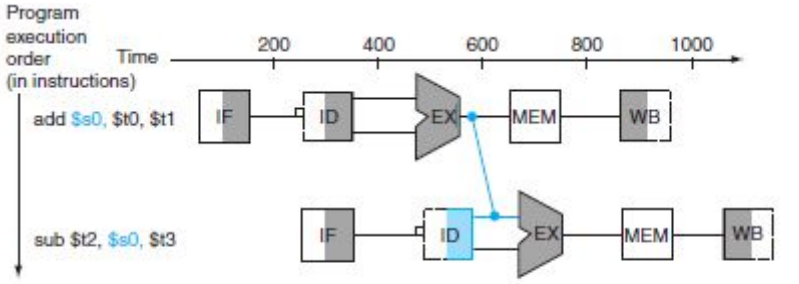


FIGURE 6.5 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Caso Load:

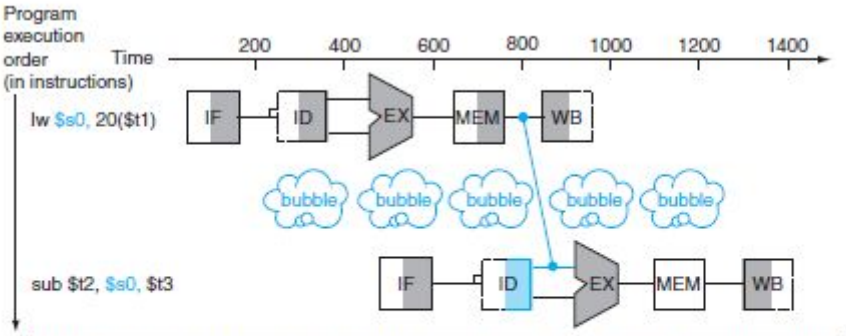
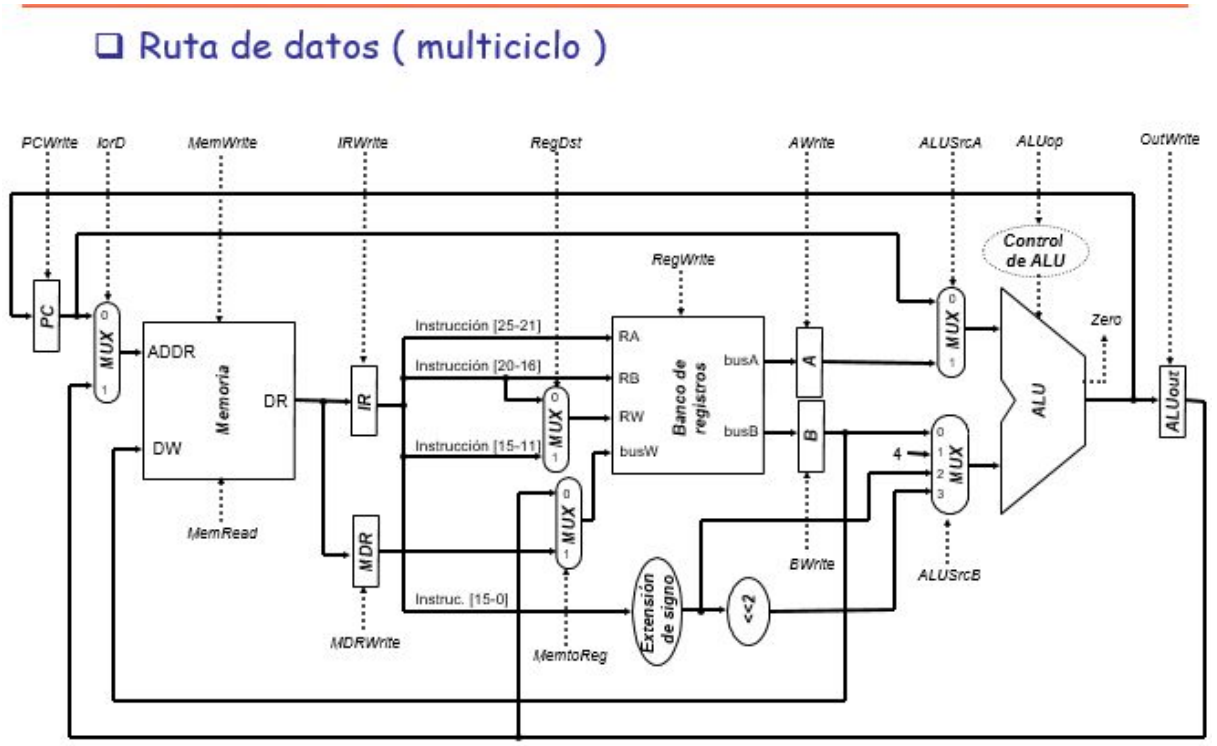


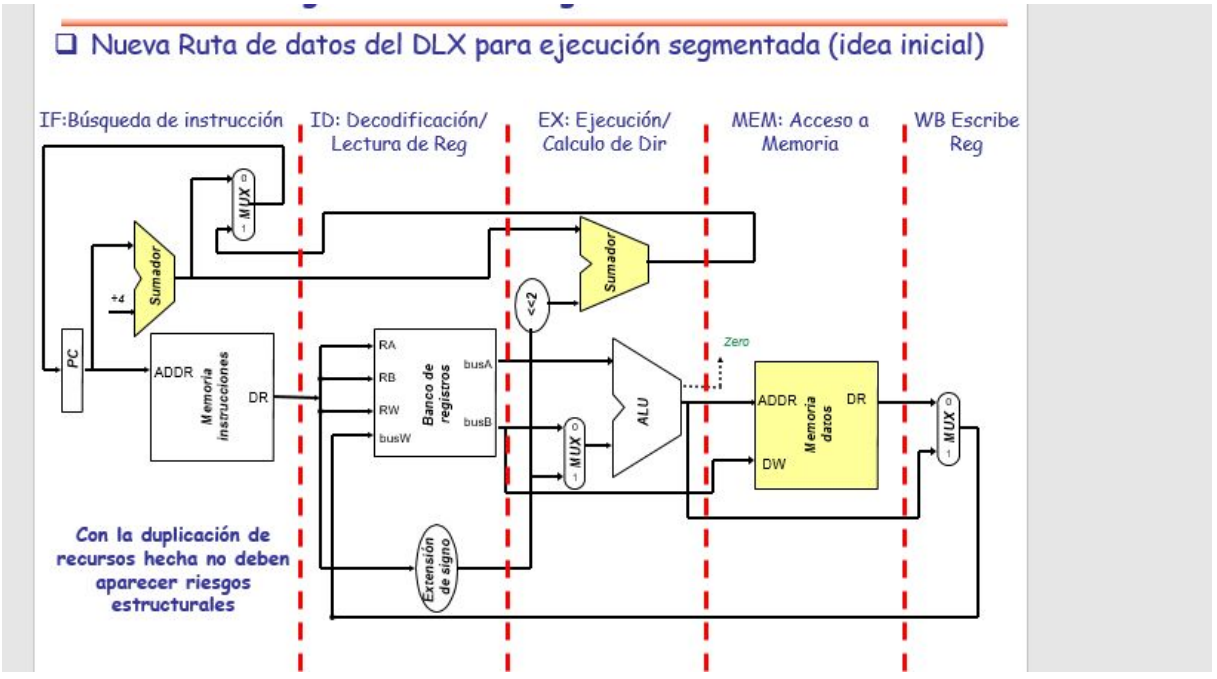
FIGURE 6.6 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 6.5 shows the details of what really happens in the case of a hazard.

Control Hazards

The third type of hazard is called a control hazard, arising from the need to make a decision based on the results of one instruction while others are executing.
The equivalent decision task in a computer is the branch instruction.

Pipeline Datapath.





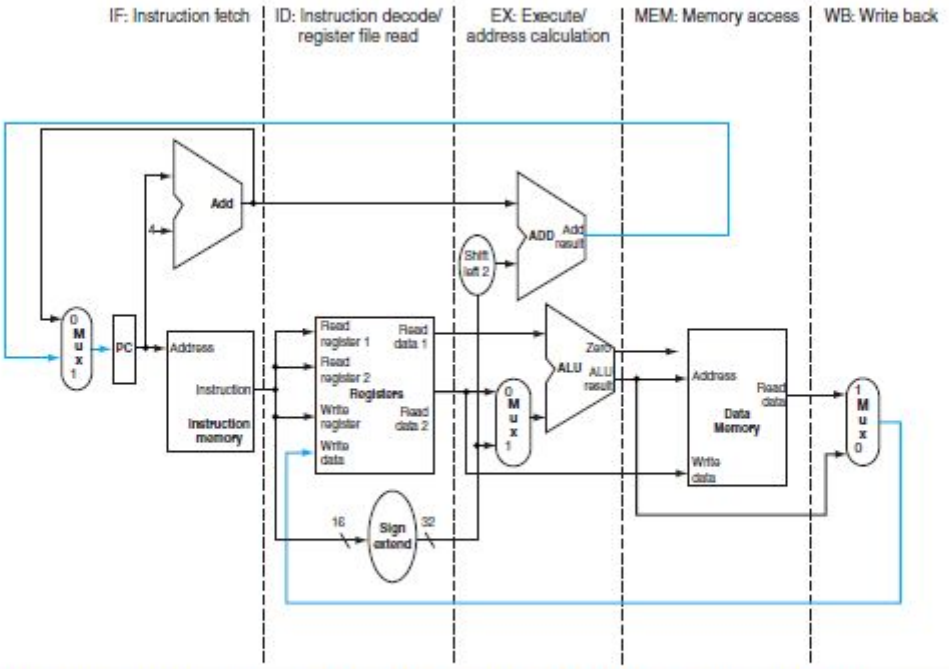
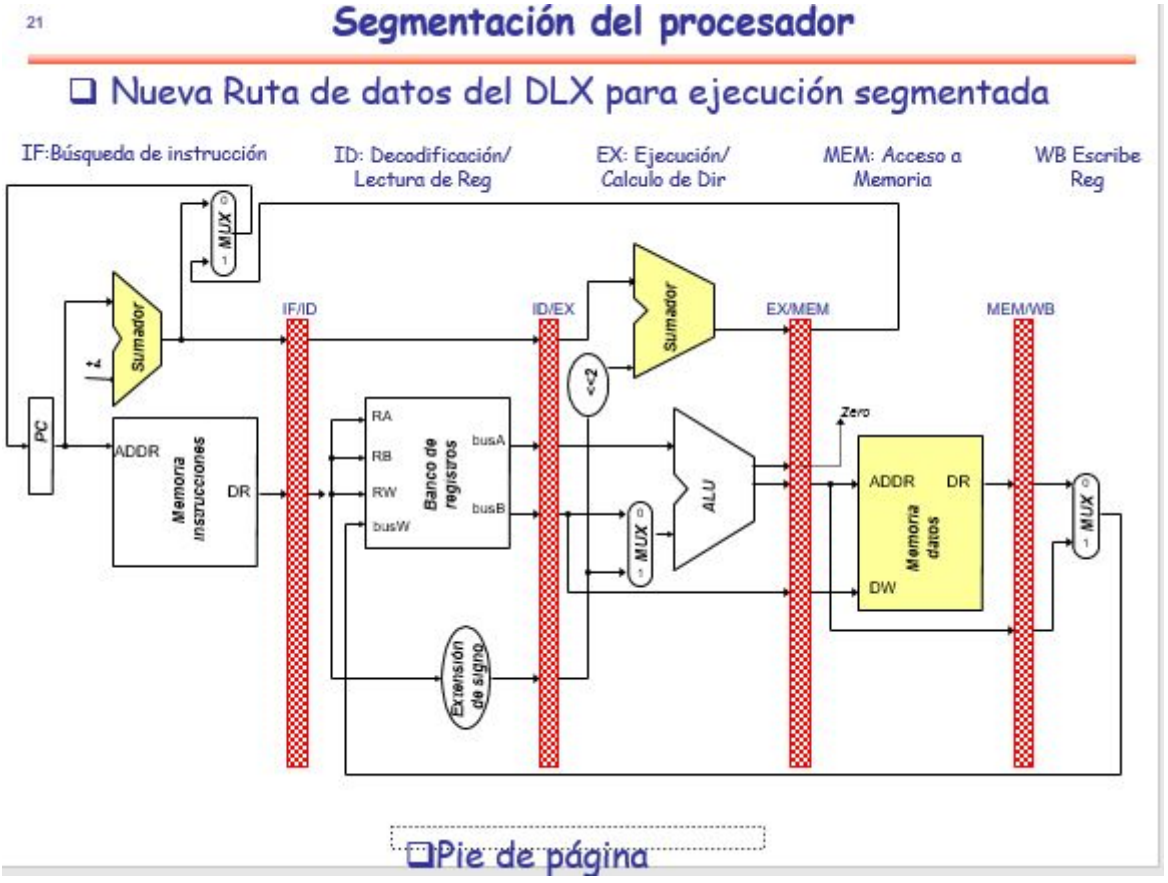


FIGURE 6.9 The single-cycle datapath from Chapter 5 (similar to Figure 5.17 on page 307). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

SEGMENTACION



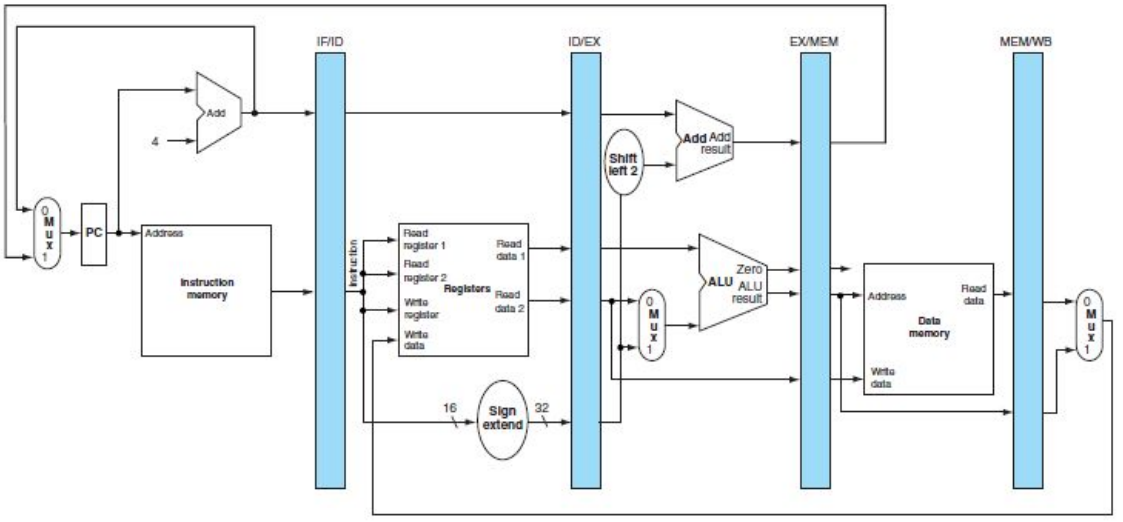


FIGURE 6.11 The pipelined version of the datapath in Figure 6.9. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

1. Instruction fetch: The top portion of Figure 6.12 shows the instruction being read from memory using the address in the PC and then placed in the IF/ID pipeline register. The IF/ID pipeline register is similar to the Instruction register in Figure 5.26 on page 320. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

2. Instruction decode and register file read: The bottom portion of Figure 6.12 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.

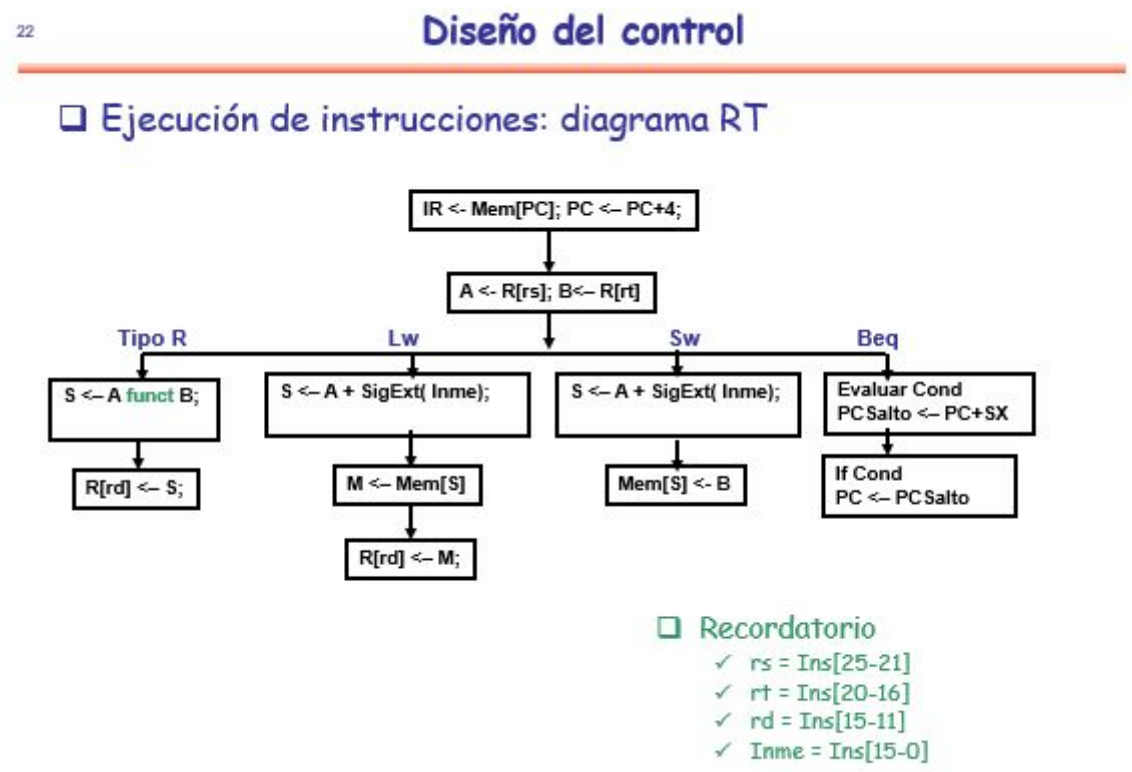
3. Execute or address calculation: Figure 6.13 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in

the EX/MEM pipeline register.

4. Memory access: The top portion of Figure 6.14 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

5. Write back: The bottom portion of Figure 6.14 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

ETAPAS INSTRUCCIONES



CONTROL DATAPATH

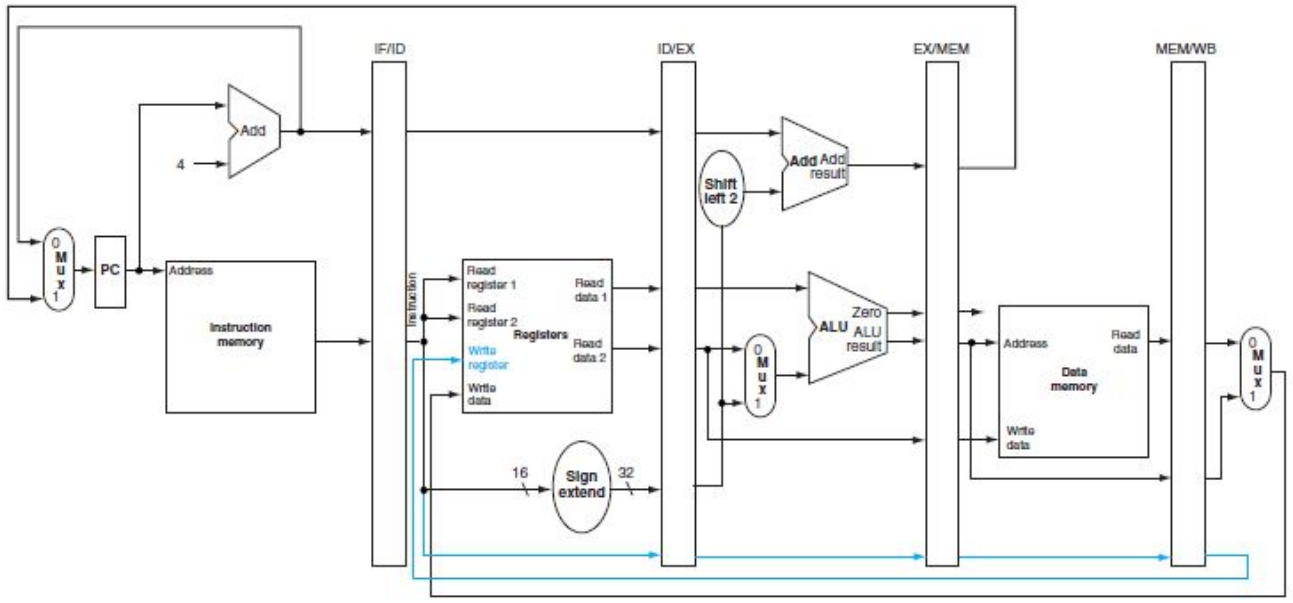
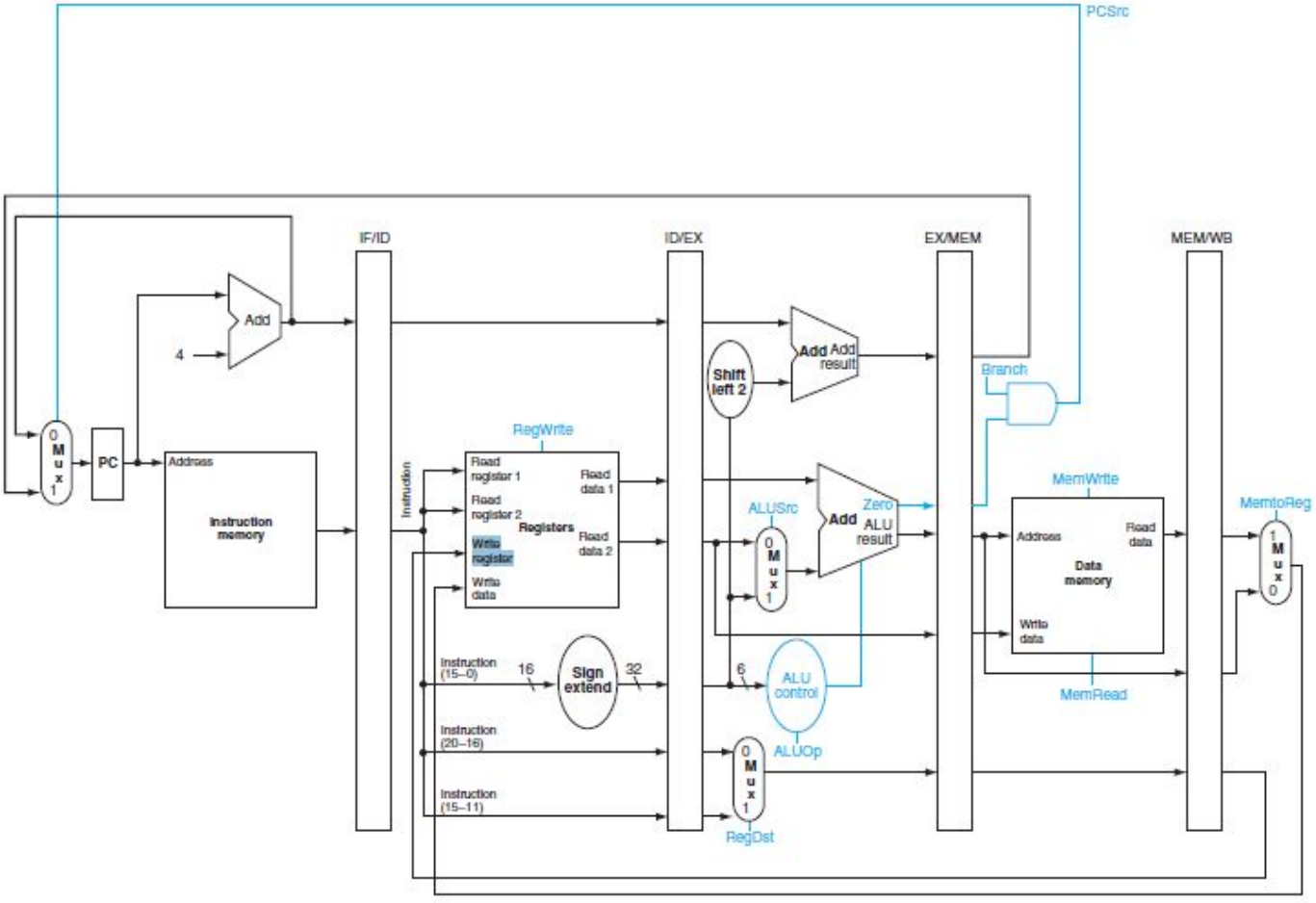


FIGURE 6.17 The corrected pipelined datapath to properly handle the load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding 5 more bits to the last three pipeline registers. This new path is shown in color.



Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 6.23 A copy of Figure 5.12 on page 302. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different function codes for the R-type instruction.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
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, lower 16 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.

FIGURE 6.24 A copy of Figure 5.16 on page 306. The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 6.23. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 6.22. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a `beq` instruction; otherwise, PCSrc is set to 0.

	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

FIGURE 6.25 The values of the control lines are the same as in Figure 5.18 on page 308, but they have been shuffled into three groups corresponding to the last three pipeline stages.

❑ Señales de control

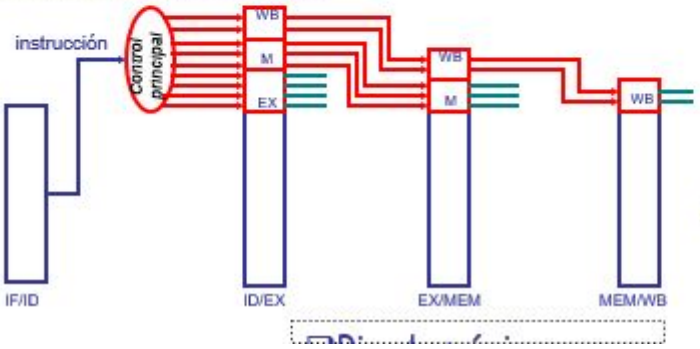
Control de la ALU

op	funct	ALUop	ALUctr
100011 (lw)	XXXXXX	00	010
101011 (sw)		00	010
000100 (beq)		01	110
000000 (tipo-R)	100000 (add)	10	010
	100010 (sub)	10	110
	100100 (and)	10	000
	100101 (or)	10	001
	101010 (slt)	10	111

Control principal

op	RegDst	ALUSrc	ALUop	MemRead	MemWrite	Branch	RegWrite	MemtoReg
100011 (lw)	0	1	00	1	0	0	1	0
101011 (sw)	X	1	00	0	1	0	0	X
000100 (beq)	X	0	01	0	0	1	0	X
000000 (tipo-R)	1	0	10	0	0	0	1	1

El control de la alu se determina por ALUop que depende del tipo de instrucción y el campo de función en las instrucciones de tipo-R



Control segmentado estacionario en los datos

3. Execution/address calculation: The signals to be set are RegDst, ALUOp, and ALUSrc (see Figures 6.23 and 6.24). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

4. Memory access: The control lines set in this stage are Branch, MemRead, and MemWrite. These signals are set by the branch equal, load, and store instructions, respectively. Recall that PCSrc in Figure 6.24 selects the next sequential address unless control asserts Branch and the ALU result was zero.

5. Write back: The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and Reg-Write, which writes the chosen value.

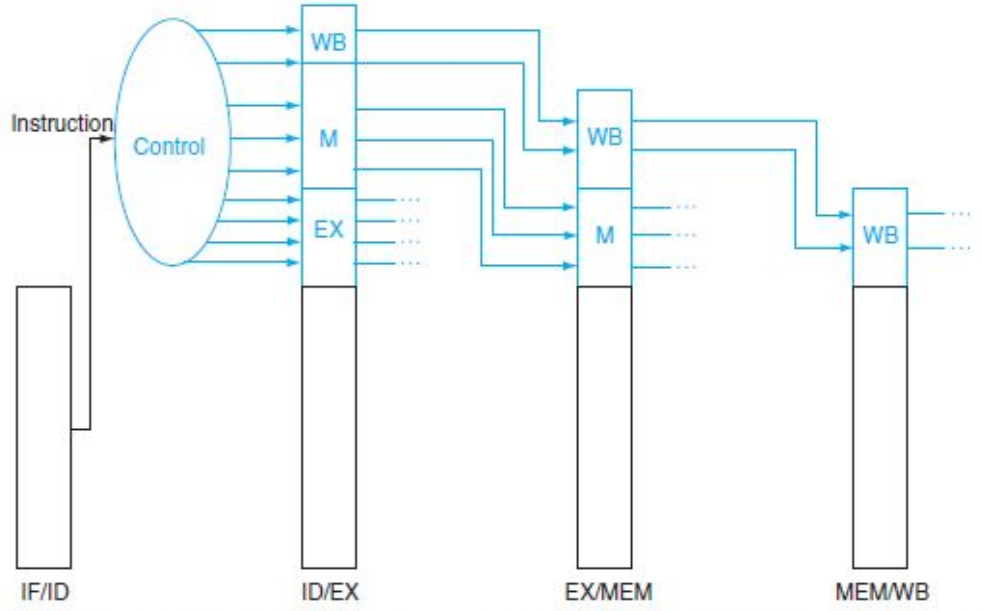
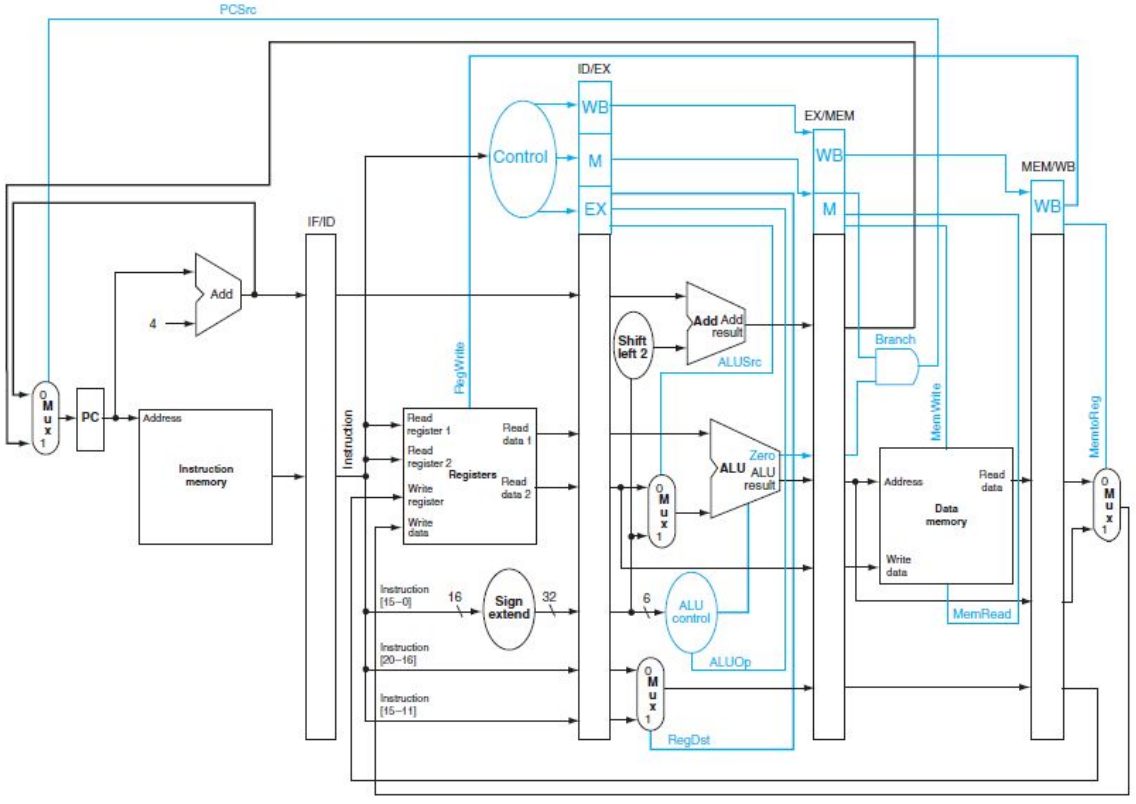
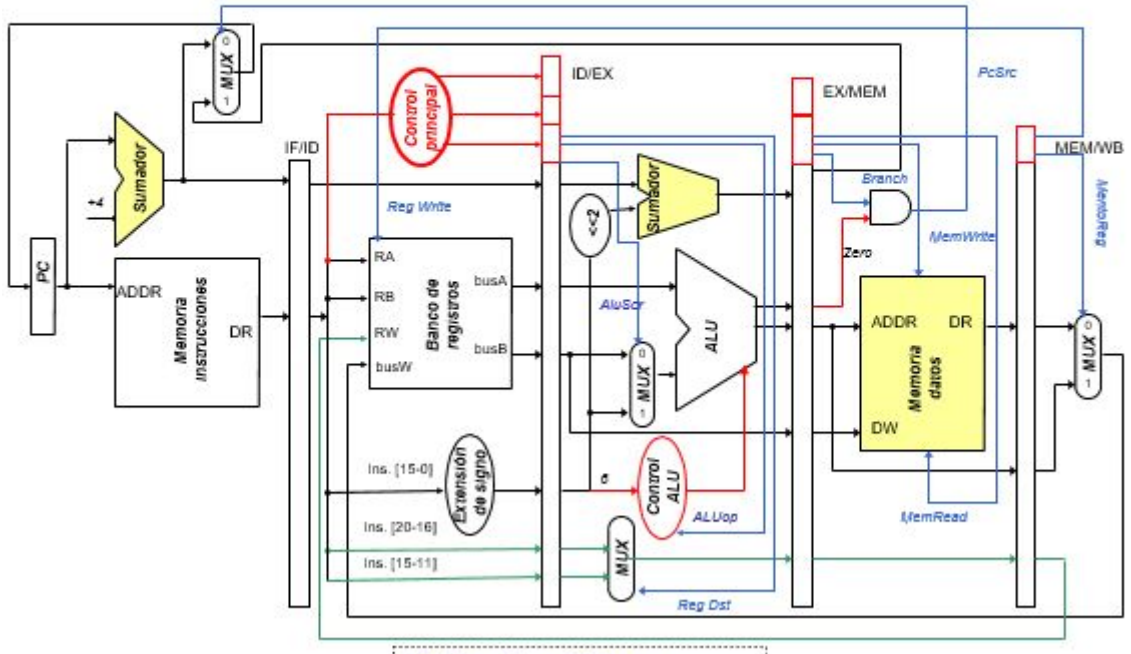


FIGURE 6.26 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.



Diseño del control

- Ruta de datos del DLX con las ordenes de control y control estacionario en los datos



- Pie de página

UNIDAD DE CORTOCIRCUITO

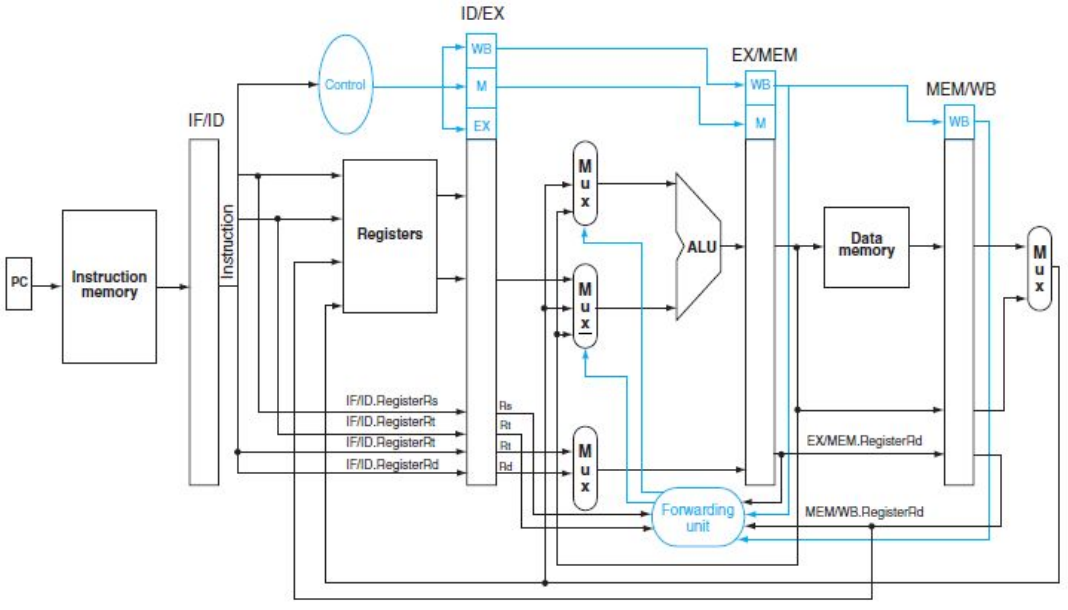
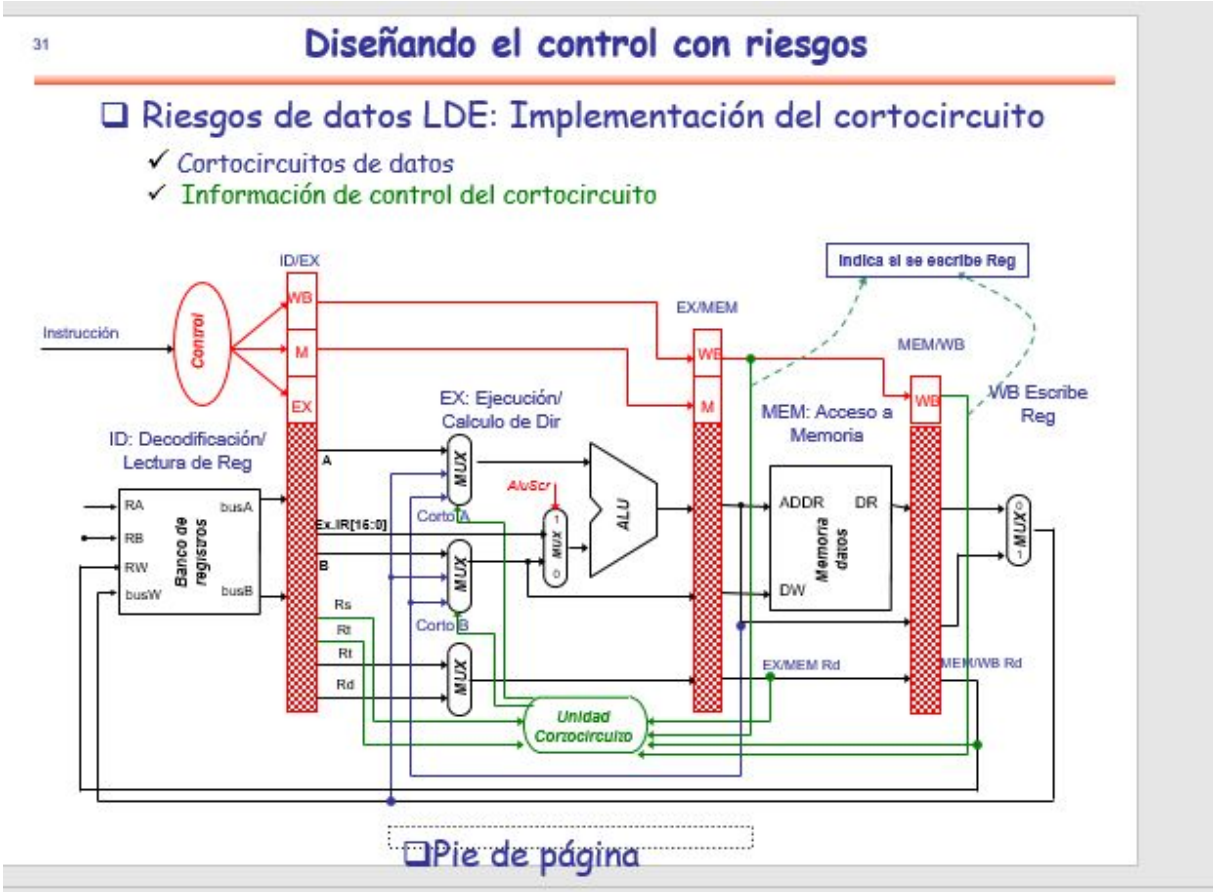


FIGURE 6.32 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 6.27 on page 404, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath such as the branch hardware and the sign extension hardware.



DETECTOR DE RIESGOS (CASO LOAD):

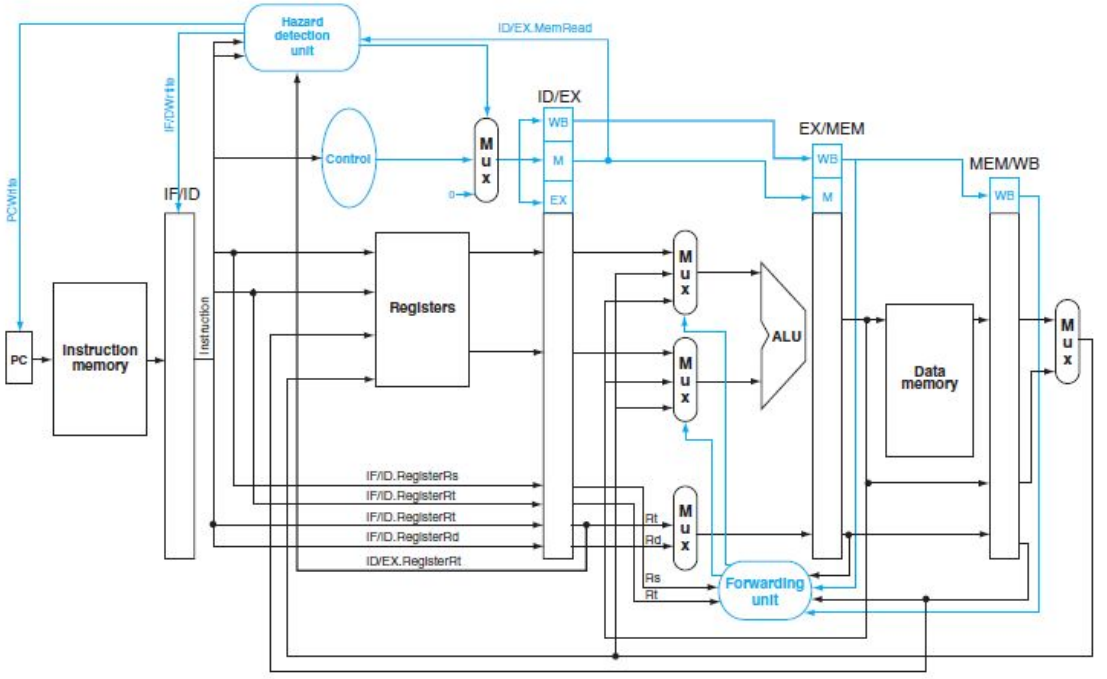
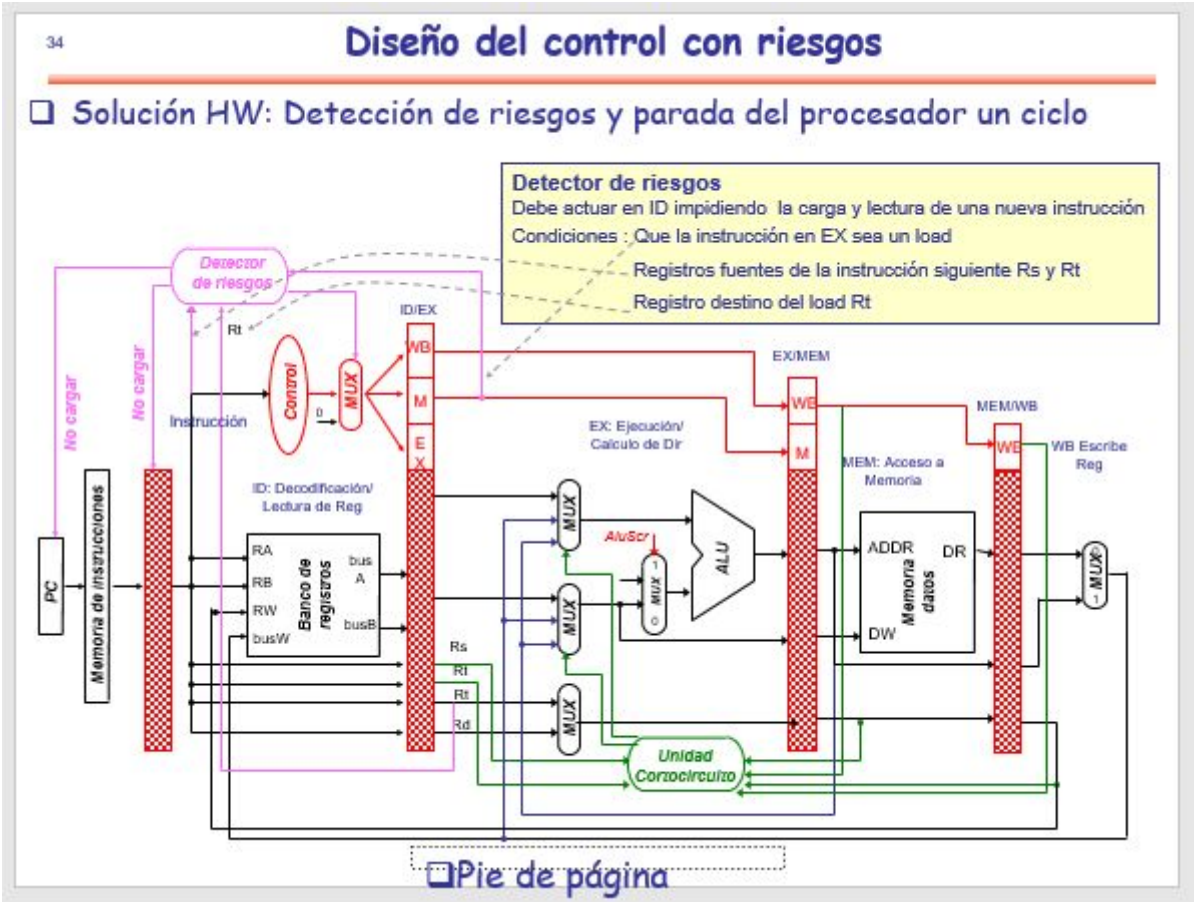
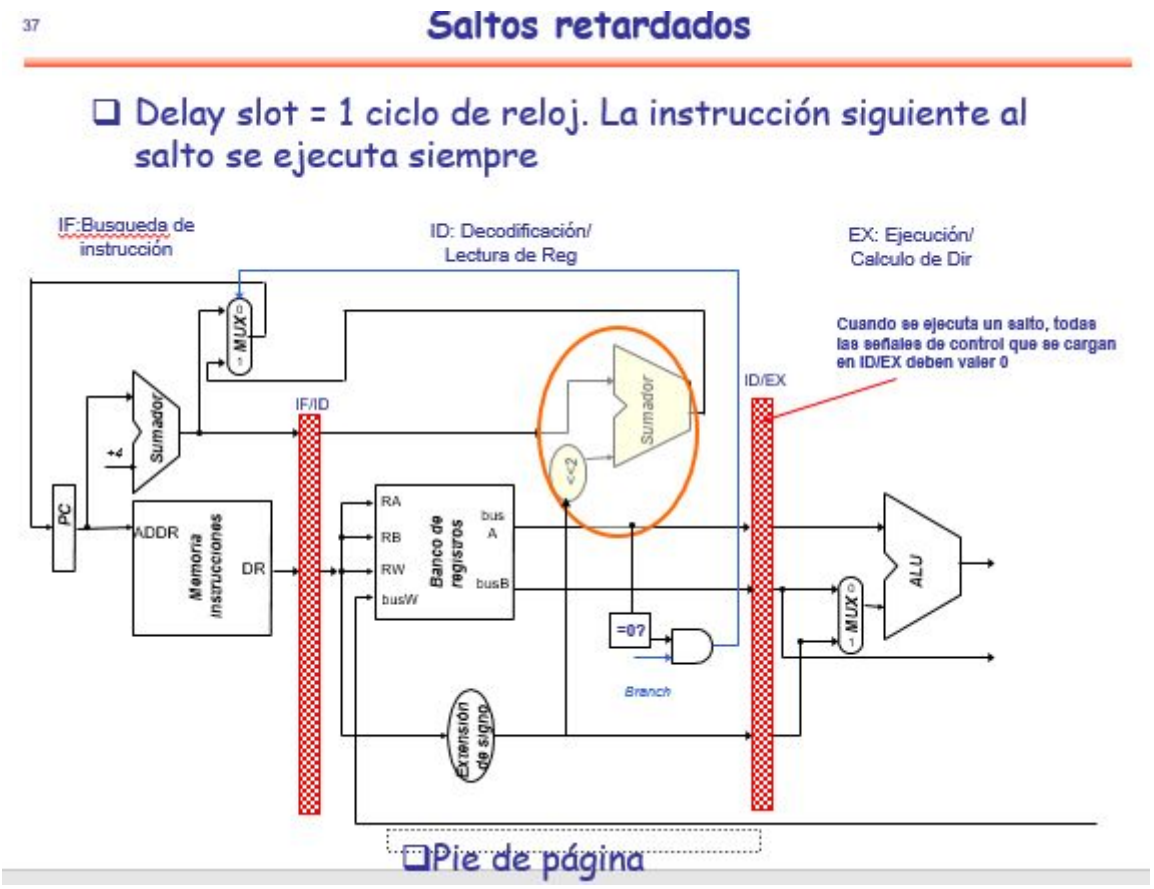
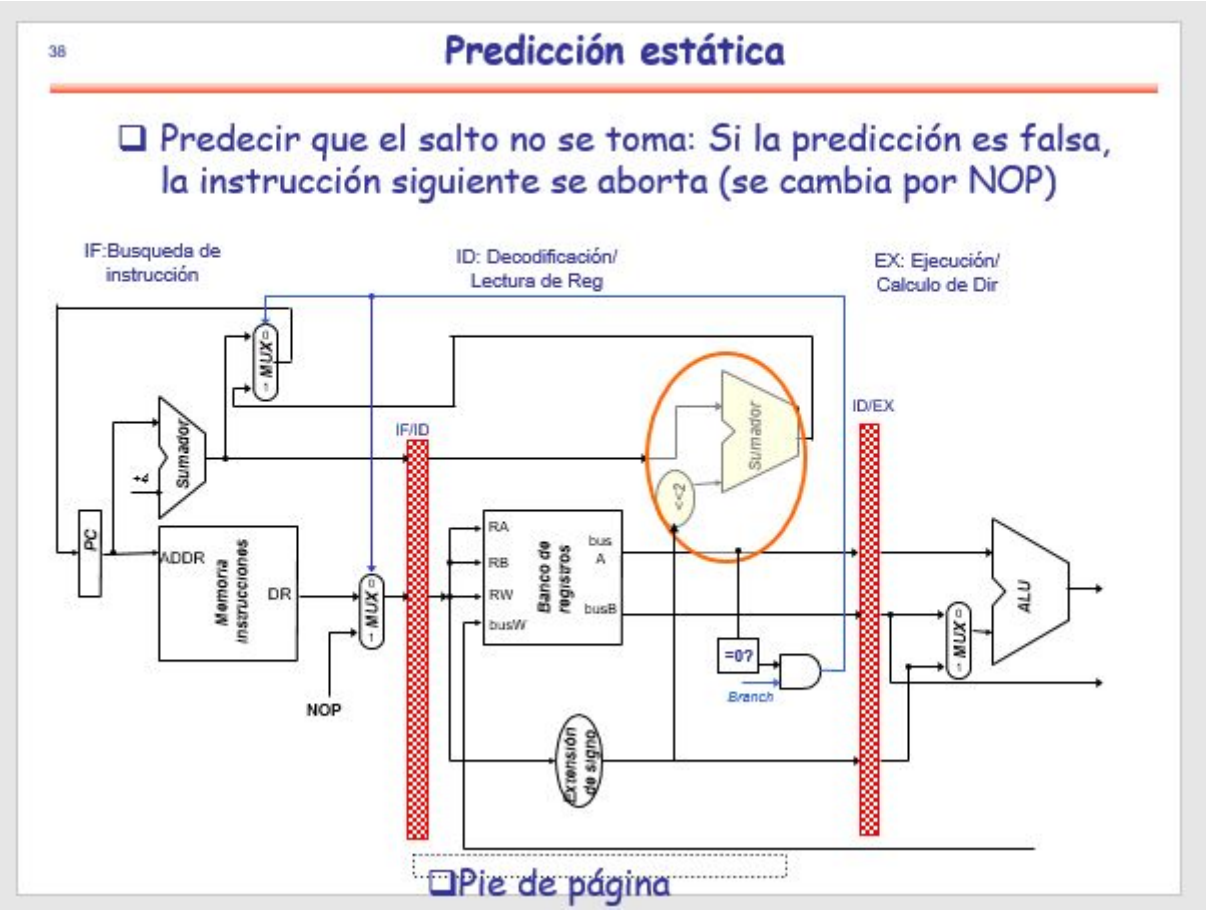


FIGURE 6.36 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.



SOLUCIÓN PARA SALTOS





2. Because the values in a branch comparison are needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces one of the operands for the comparison in the branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch.

Más info

32 registros de 32 bits cada uno.