APÉNDICE C*

ñPIPELININGò: conceptos básicos e intermedios RISC - V

* Computer Architecture , Hennessy & Patterson, 6ta Edición

The load and store instructions in RISC-V

ld x1,80(x2)	Load doubleword	Regs[x1] = Mem[80+Regs[x2]]
lw x1,60(x2)	**Load word**	Regs[x1] <- $_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1] <- $_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte	Regs[x1] <- $_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1] <- $_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word	Regs[x1]<- $_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
flw f0,50(x3)	**Load FP single**	Regs[f0]<- $_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double	Regs[f0]<- $_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double	Mem[400+Regs[x3]] <- $_{64}$ Regs[x2]
sw x3,500(x4)	**Store word**	Mem[500+Regs[x4]]<- $_{32}$ Regs[x3]$_{32..63}$
fsw f0,40(x3)	**Store FP single**	Mem[40+Regs[x3]] <- $_{32}$ Regs[f0]$_{0..31}$
fsd f0,40(x3)	Store FP double	Mem[40+Regs[x3]] <- 64 Regs[f0]
sh x3,502(x2)	Store half	Mem[502+Regs[x2]] <-$_{16}$ Regs[x3]$_{48..63}$
sb x2,41(x3)	Store byte	Mem[41+Regs[x3]] <-$_8$ Regs[x2]$_{56..63}$

The basic ALU instructions in RISC-V

add x1,x2,x3	**Add**	Regs[x1] Regs[x2]+Regs[x3]
addi x1,x2,3	**Add immediate unsigned**	Regs[x1] Regs[x2]+3
lui x1,42	**Load upper immediate**	Regs[x1] = 0^{32}##42##0^{12}
sll x1,x2,5	**Shift left logical**	Regs[x1] = Regs[x2]<<5
slt x1,x2,x3	**Set less than**	if (Regs[x2]<Regs[x3]) Regs[x1] = 1 else Regs[x1] = 0

Typical control flow instructions in RISC-V

jal x1,offset	**Jump and link**	Regs[x1]= PC+4; PC= PC + **(offset<<1)**
jalr x1,x2,offset	**Jump and link register**	Regs[x1]= PC+4; PC = Regs[x2]+offset
beq x3,x4,name	**Branch equal**	if (Regs[x3]==Regs[x4]) PC = PC + **(offset<<1)**
bgt x3,x4,name	**Branch greater than**	if (Regs[x3]>Regs[x4]) PC= PC + **(offset<<1)**

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

Here is the RISC-V code that we would typically generate for this code fragment assuming that `aa` and `bb` are assigned to registers `x1` and `x2`:

```
     addi  x3,x1,-2
     bnez  x3,L1       //branch b1  (aa!=2)
     add   x1,x0,x0    //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2       //branch b2  (bb!=2)
     add   x2,x0,x0    //bb=0
L2:  sub   x3,x1,x2    //x3=aa-bb
     beqz  x3,L3       //branch b3  (aa==bb)
```

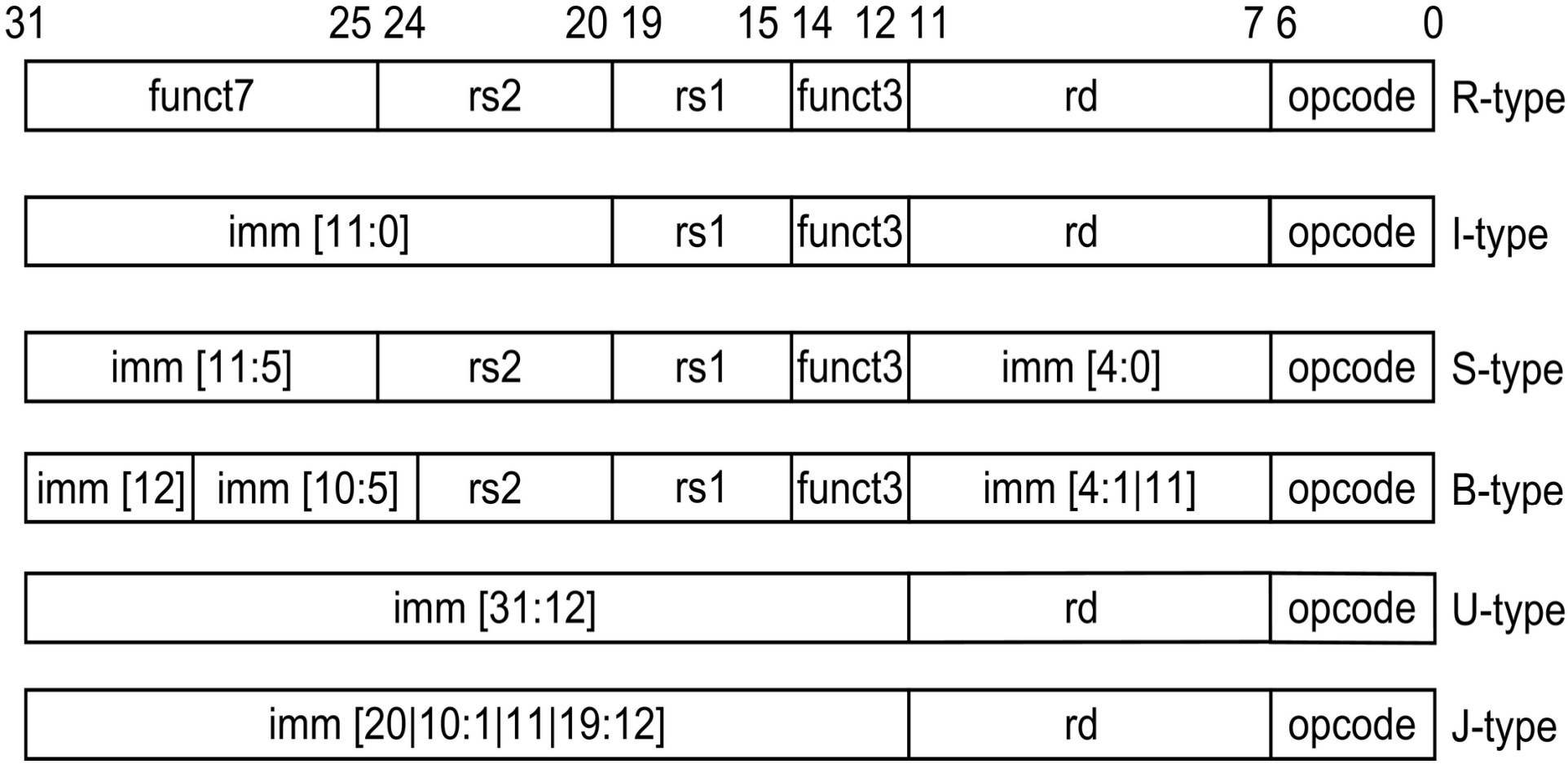
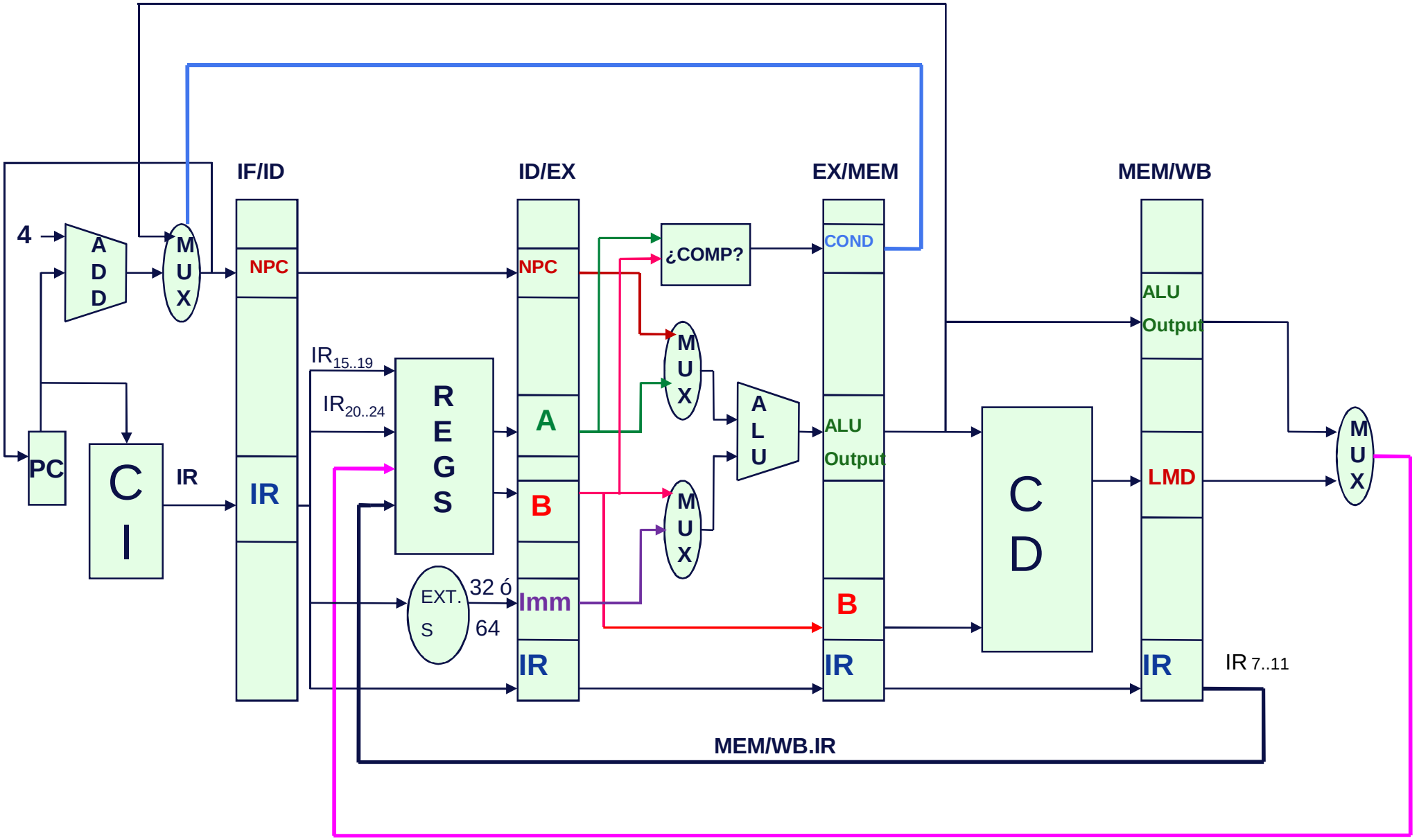


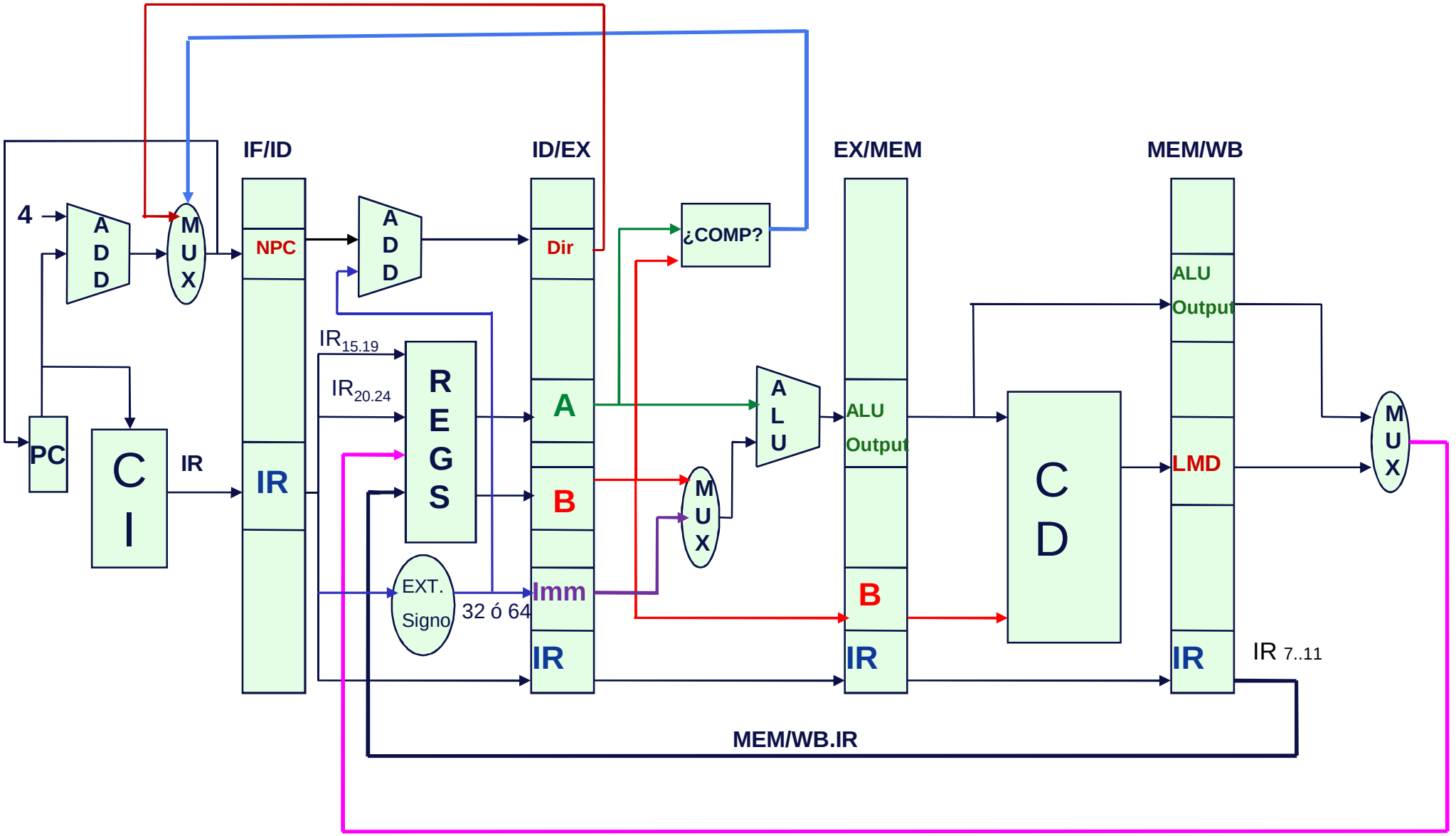
Figura 1.7 Formatos de codificación base para RISC-V Todas son de 32 bits .

Tipo **R**: Para operaciones de enteros R-R
Tipo **I**: Para Loads y operaciones con inmediato.
Tipo **B**: Para Branches
Tipo **J**: Para Jumps and link
Tipo **S**: Para Stores (eso para que los especificadores de los 3 registros estén siempre en la misma posición - (rd, rs1, rs2)
Tipo **U**: Para instrucciones con ñnmediato anchoò(LUI, AUIPC)

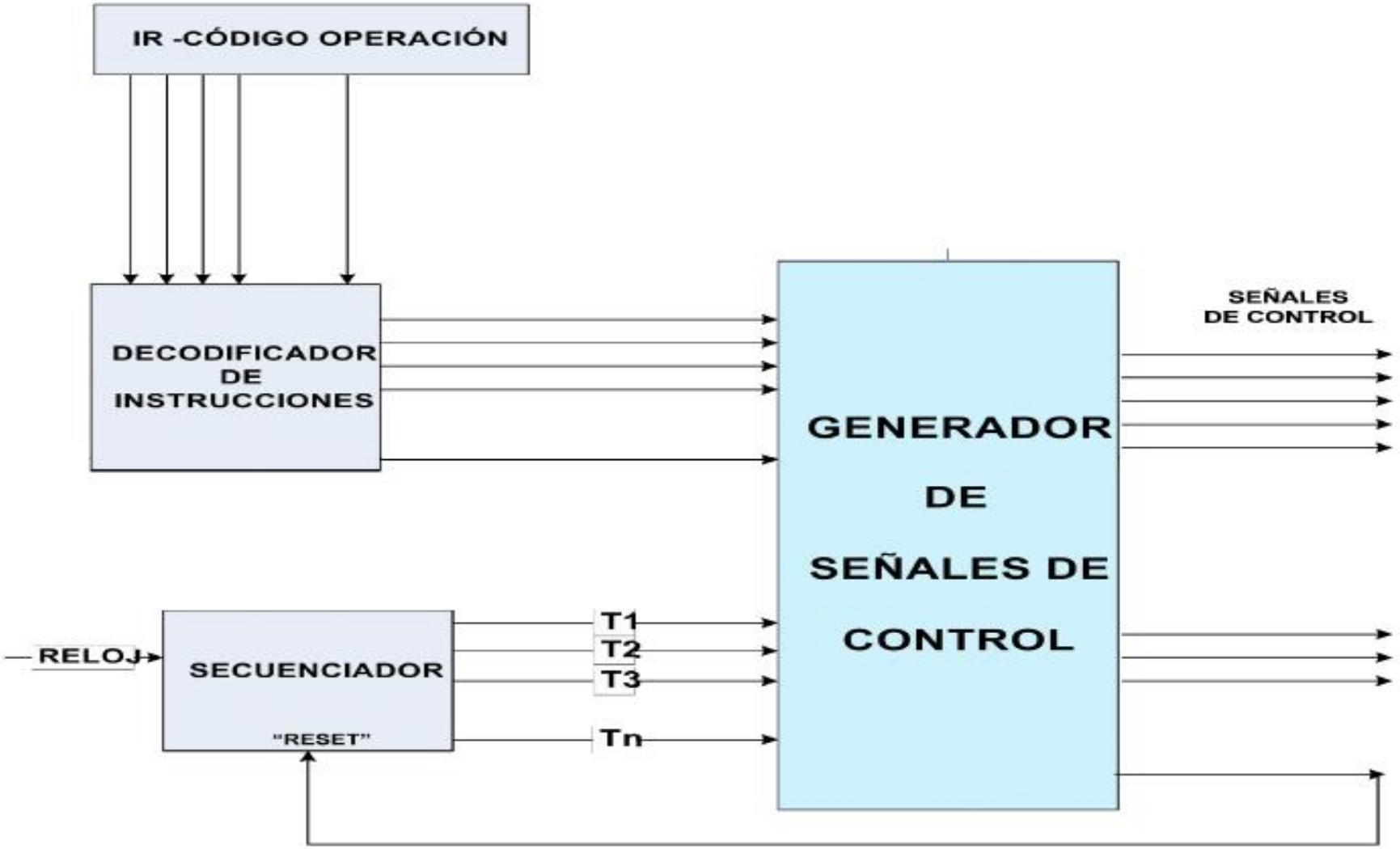
PIPELINE ENTEROS RISC-V



PIPELINE ENTEROS RISC-V MODIFICADO



CONTROL ALAMBRADO
(Hardwired)



EJECUCIÓN EN PIPELINE
TABLA DE TIEMPOS

Explicación	Instrucción	1	2	3	4	5	6	7	8	9
x1=M[60 + x2]	lw x1,60(x2)	IF	ID	EX	M	WB				
x7=x4 + x5	add x7,x4,x5		IF	ID	EX	M	WB			
x8= x9 - x6	sub x8,x9,x6			IF	ID	EX	M	WB		
x3= x10 + 6	addi x3,x10,6				IF	ID	EX	M	WB	
M[500 + x4]=x12	sw x12,500(x4)					IF	ID	EX	M	WB

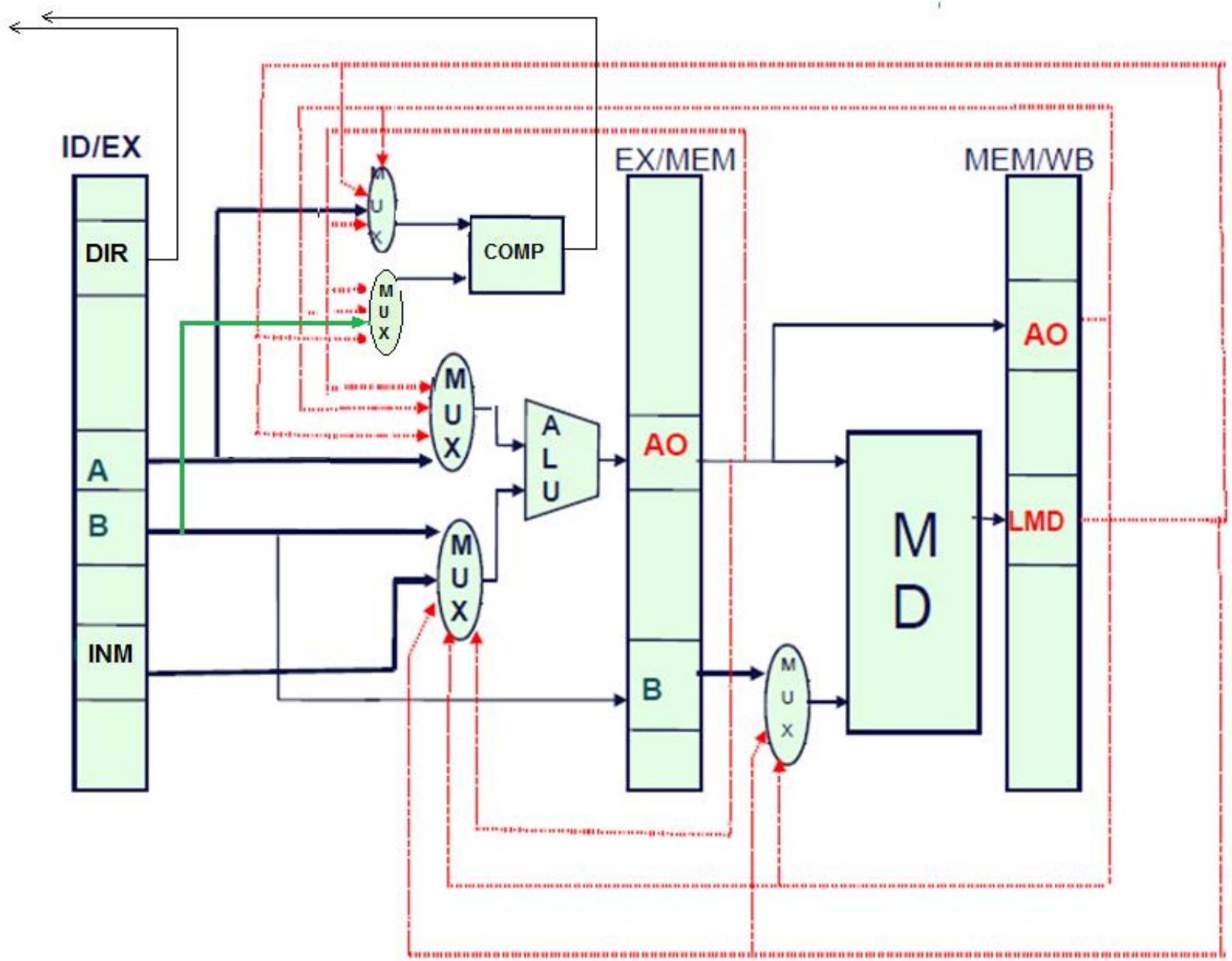
CONFLICTOS DE DATOS
SOLUCIÓN 1- DETENER PIPELINE

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
add x2,x4,x5	IF	ID	EX	M	WB										
lw x1,60(x2)		IF	ID	ID	ID	EX	M	WB							
addi x3,x1,6					IF	ID	ID	ID	EX	M	WB				
sw x3,500(x1)								IF	ID	ID	ID	EX	M	WB	
sub x8,x1,x2											IF	ID	EX	M	WB

CONFLICTOS DE DATOS
SOLUCIÓN 2- FORWARDING

	1	2	3	4	5	6	7	8	9	10	11
add x2,x4,x5	IF	ID	EX	M	WB						
lw x1,60(x2)		IF	ID	EX	M	WB					
addi x3,x1,6			IF	ID	stall	EX	M	WB			
sw x3,500(x1)				IF	stall	ID	EX	M	WB		
sub x8,x1,x2					stall	IF	ID	EX	M	WB	

Forwarding requiere 3 entradas extra en multiplexores del ALU, de COMP. y a mux de MD



Hacer aquí:

Ejemplos y práctica con FORWARDING

CONFLICTOS DE CONTROL

```
        add x1,x2,x3
        beq x12,x0,skip
        sub x4,x5,x6
        add x5,x4,x9
skip:   or  x7,x8,x9
        ETC...
```

SOLUCIÓN 1: DETENER PIPELINE

SIN PREDICCIÓN - FALSO

	1	2	3	4	5	6	7	8	9	10	11	CPI
add x1,x2,x3	IF	ID	EX	M	WB							1
beq x12,x0,skip		IF	ID	EX	M	WB						1
sub x4,x5,x6			~~IF~~									
sub **x4**,x5,x6					IF	ID	EX	M	WB			3
add x5,**x4**,x9						IF	ID	EX	M	WB		1
skip: or x7,x8,x9							IF	ID	EX	M	WB	1

ETC...

SIN PREDICCIÓN - VERDADERO

	1	2	3	4	5	6	7	8	9	10	11	CPI
add x1,x2,x3	IF	ID	EX	M	WB							1
beq x12,x0,skip		IF	ID	EX	M	WB						1
sub x4,x5,x6			~~IF~~									
skip: or x7,x8,x9					IF	ID	EX	M	WB			3

ETC...

SOLUCIÓN 2 (para disminuir ciclos de retraso): PREDICCIÓN NO TOMADO (ñESTÁTICAð)

PREDICCIÓN NO TOMADO - FALSO

	1	2	3	4	5	6	7	8	9	10	11	CPI
add x1,x2,x3	IF	ID	EX	M	WB							1
beq x12,x0,skip		IF	ID	EX	M	WB						1
sub **x4**,x5,x6			IF	ID	EX	M	WB					1
add x5,**x4**,x9				IF	ID	EX	M	WB				1
skip: or x7,x8,x9					IF	ID	EX	M	WB			1
ETC...												

PREDICCIÓN NO TOMADO - VERDADERO

	1	2	3	4	5	6	7	8	9	10	11	CPI
add x1,x2,x3	IF	ID	EX	M	WB							1
beq x12,x0,skip		IF	ID	EX	M	WB						1
sub **x4**,x5,x6			IF	ID								
add x5,**x4**,x9				IF								
skip: or x7,x8,x9					IF	ID	EX	M	WB			3
ETC...												

SOLUCIÓN 3 (dism. retraso): BRANCH RETRASADO (Delayed Branch)

Compilador incluye instrucciones después del branch

Mientras se resuelve el branch, el procesador las comienza a ejecuta, e independientemente del resultado de este, concluye su ejecución

Estas instrucciones:
- se toman del código anterior al branch,
-no dependen del resultado del branch,
- no afectan su condición y
- no tienen conflictos entre ellas

BRANCH RETRASADO - EJEMPLO

X y V dos vectores enteros de longitud 32

X se encuentra en memoria a partir del byte 0 y V a partir del byte 128)

```
for (int i = 0; i < 32, i++) {

V[31-i] = X[i] + V[31-i];
}
```

Se resuelve sin branch retrasado:

x1 tiene dirección del primer elemento de X, es decir, contiene un 0

x2 del último de V, o sea, tiene el valor 252

x3 es 32, la dimensión de X y de V:

```
AQUÍ:  lw   x8, 0(x2)
       lw   x9, 0(x1)
       add  x8, x8, x9
       sw   x8, 0(x2)
       addi x2, x2, - 4
       addi x1, x1, 4
       addi x3, x3, -1
       bnez x3, AQUÍ
```

Se resuelve con branch retrasado se incluyen 2 instrucciones buenas:

Código anterior (original)

```
AQUÍ: lw   x8, 0(x2)
      lw   x9, 0(x1)
      add  x8, x8, x9
      sw   x8, 0(x2)
      addi x2, x2, - 4
      addi x1, x1, 4
      addi x3, x3, -1
      bnez x3, AQUÍ
```

Código nuevo por branch retrasado

```
AQUÍ: lw   x8, 0(x2)
      lw   x9, 0(x1)
      add  x8, x8, x9
      addi x1, x1, 4
      addi x3, x3, -1
      bnez x3, AQUÍ
      sw   x8, 0(x2)
      addi x2, x2, -4
```

SOLUCIÓN 4: PREDICCIÓN DINAMICA

(Predictores y Branch-Prediction Buffers)

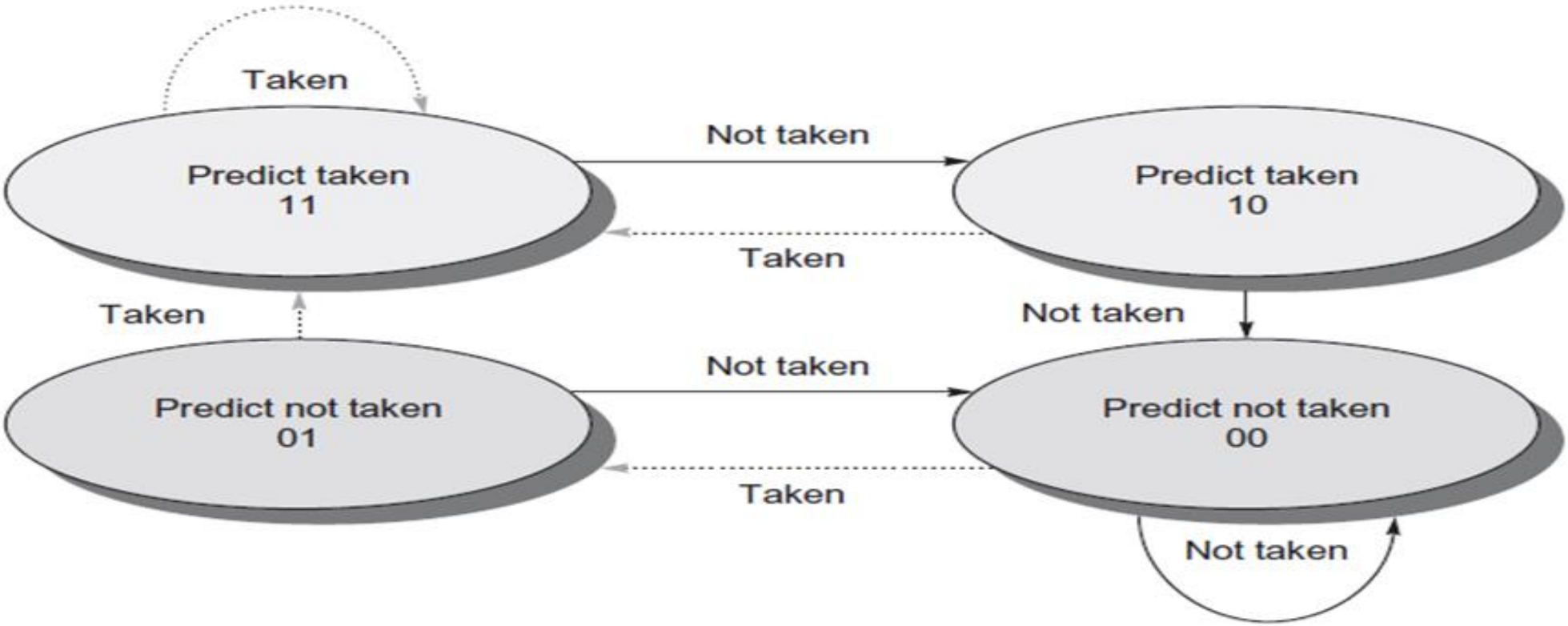
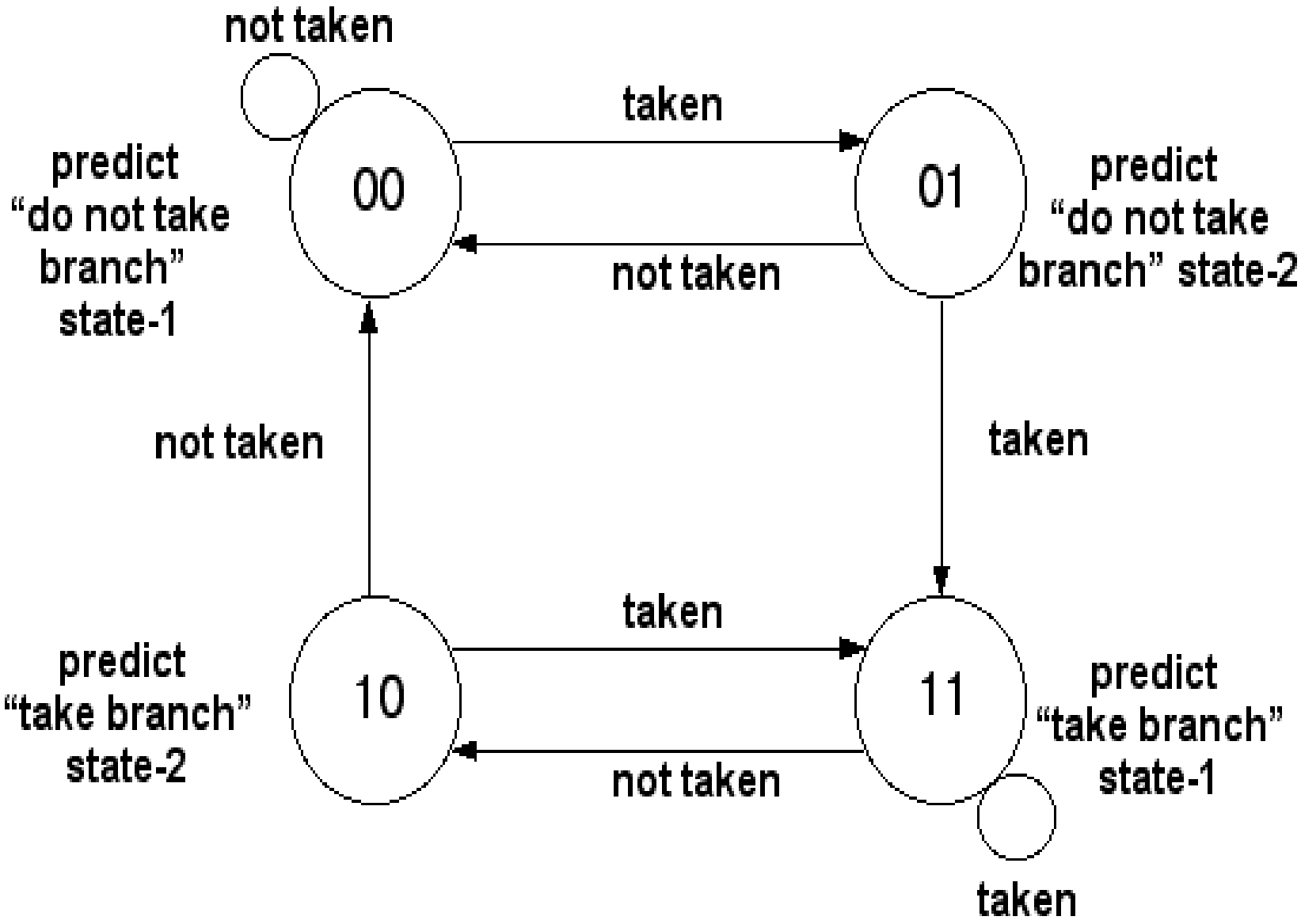
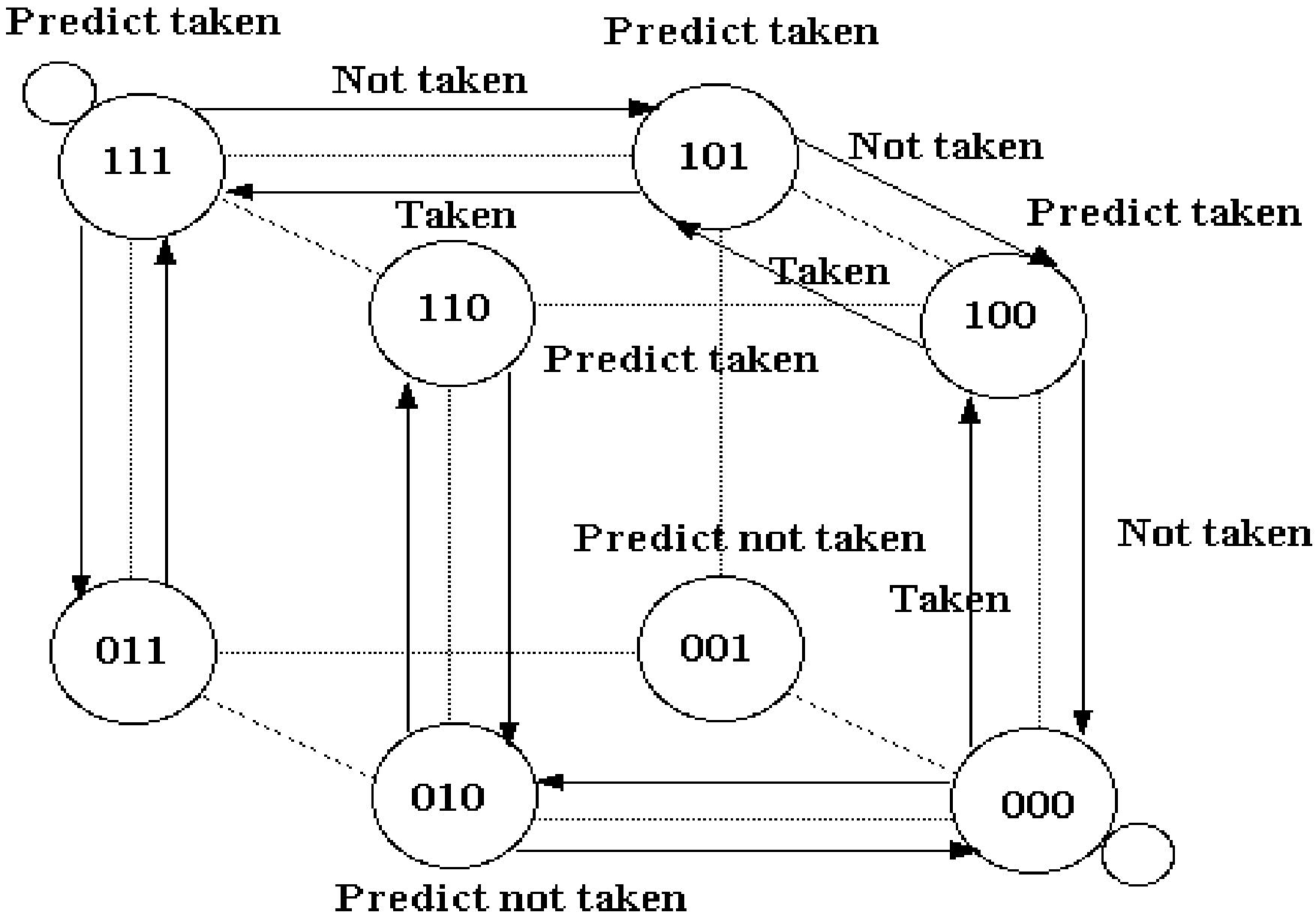


Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of n-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general n-bit predictors.

PREDICTOR DE 2 BITS



PREDICTOR DE 3 BITS



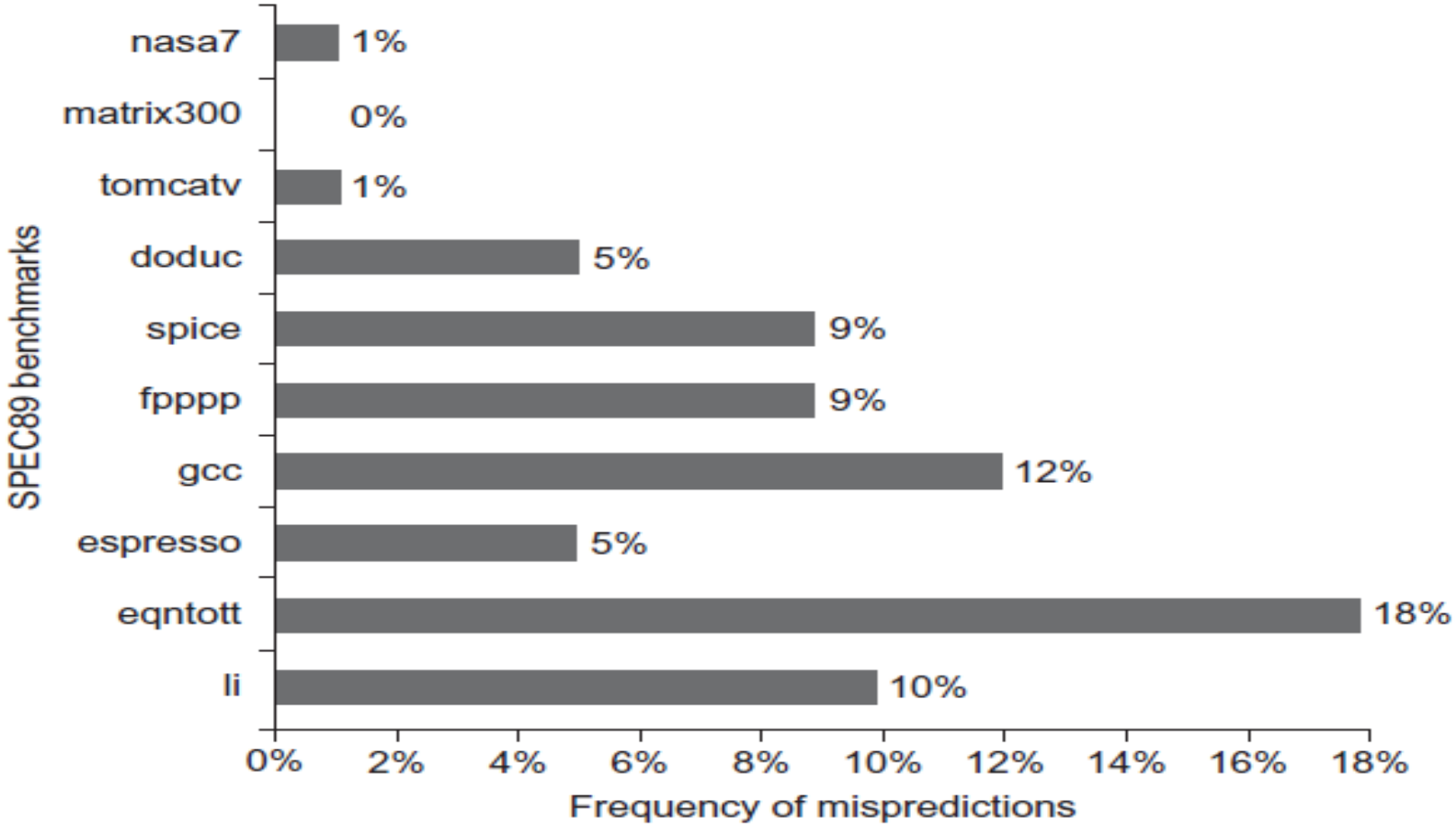
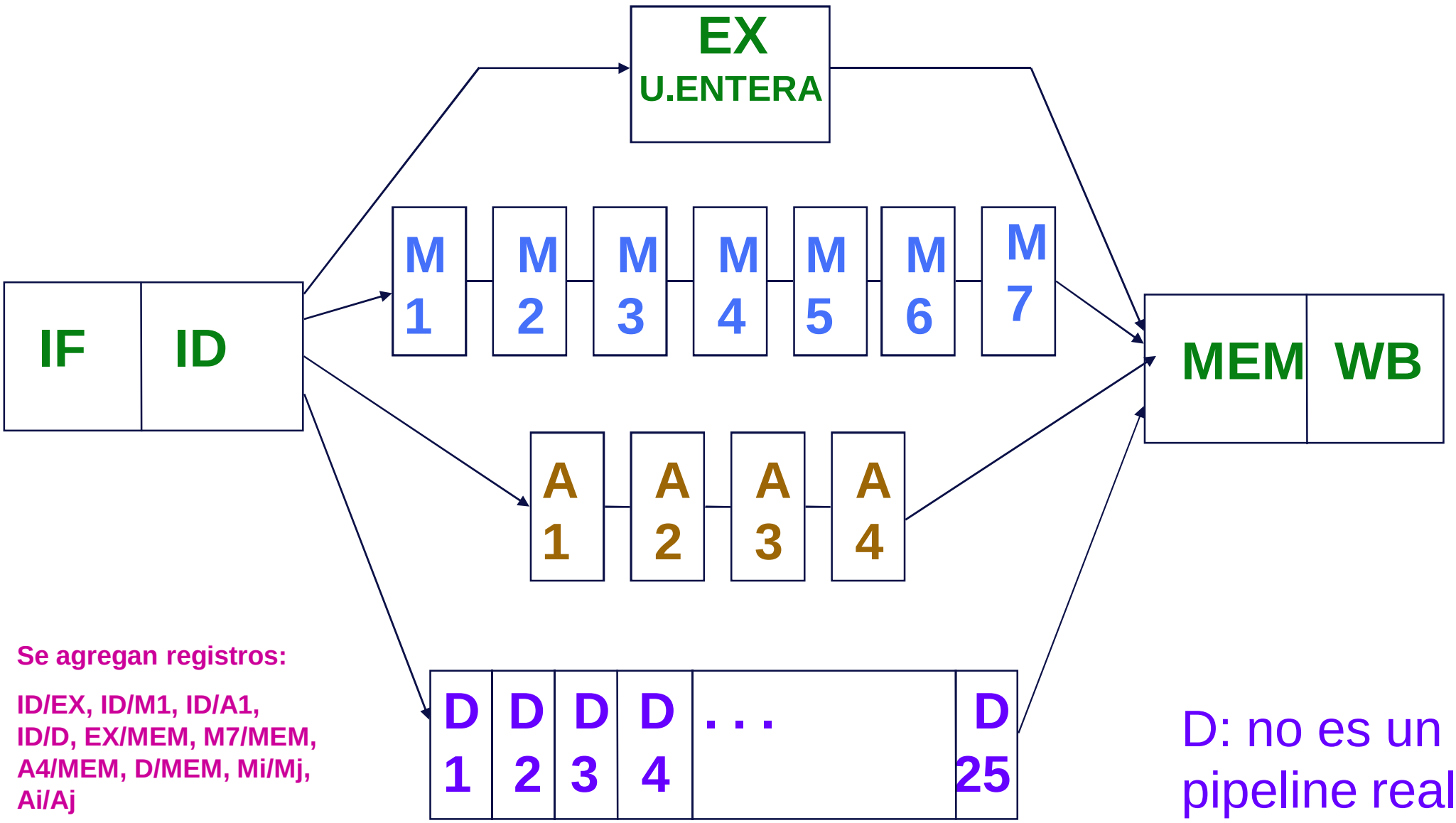


Figure C.16 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. (1992). Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

PIPELINE RISC-V (Fig. C.30)



Se agregan registros:

ID/EX, ID/M1, ID/A1, ID/D, EX/MEM, M7/MEM, A4/MEM, D/MEM, Mi/Mj, Ai/Aj

D: no es un pipeline real

LATENCIA vs. INTERVALO DE REPETICIÓN

Latencia de unidad funcional:. Número de ciclos entre instrucción que usa resultados y la instrucción que los produce.

Intervalo de iniciación/ intervalo de repetición: número de ciclos que deben darse entre la emisión de instrucciones del mismo tipo.

UNIDAD FUNCIONAL	LATENCIA	INTERVALO INIC.
ALU Entera	0	1
Memoria datos (loads ent. y fp, 1 menos para latencia de store)	1	1
Sumador/Rest. FP	3	1
Multiplicador Ent./FP	6	1
Divisor Ent/FP	24	25

Hacer aquí:

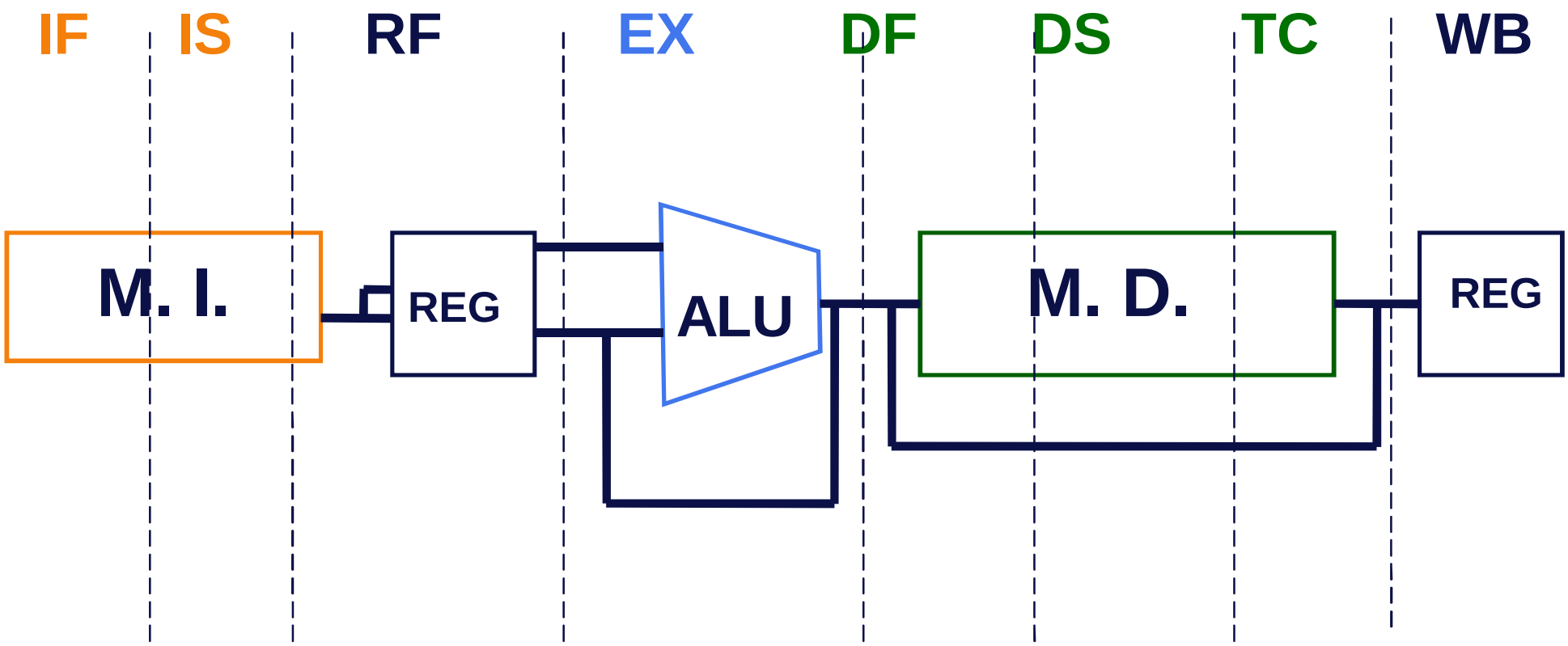
Ejemplos y práctica flotantes

EL PIPELINE DE LA MIPS R4000 (A.6)

Es realmente una máquina de 64 bits

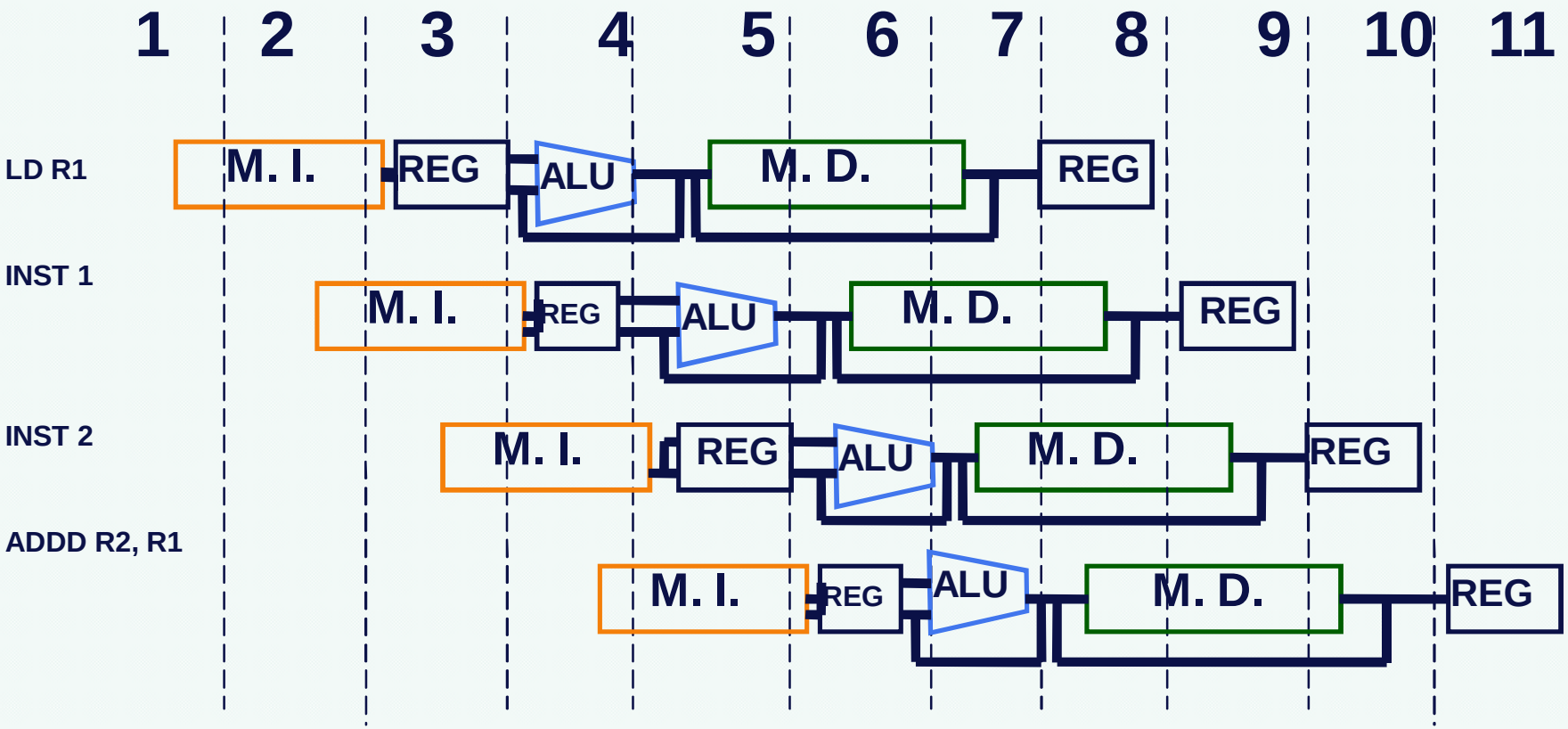
- Pipelinig muy profundo (superpipelining). 8 etapas:
  - **IF** 1er parte fetch instrucc. selección PC, inicia inst. acceso a cache
  - **IS** 2da parte fetch instrucc. Se completa acceso a cache
  - **RF** Decod. Instruc, fetch registros, revisión conflictos, detección de si hubo hit en cache
  - **EX** EFA, ALU op, branch target (dirección nueva)y cód. condición
  - **DF** Fetch de datos, 1era mitad acceso a cache de datos
  - **DS** 2da mitad fetch de datos. Se completa acceso de datos en cache
  - **TC** Revisión de etiqueta (tag), se determina si hubo hit en acceso a cache de datos
  - **WB** ñWrite Backòpara operaciones reg-reg y loads

PIPELINE MIPS R4000 (CONT)



Load tiene 2 ciclos de retraso-(delay slots)

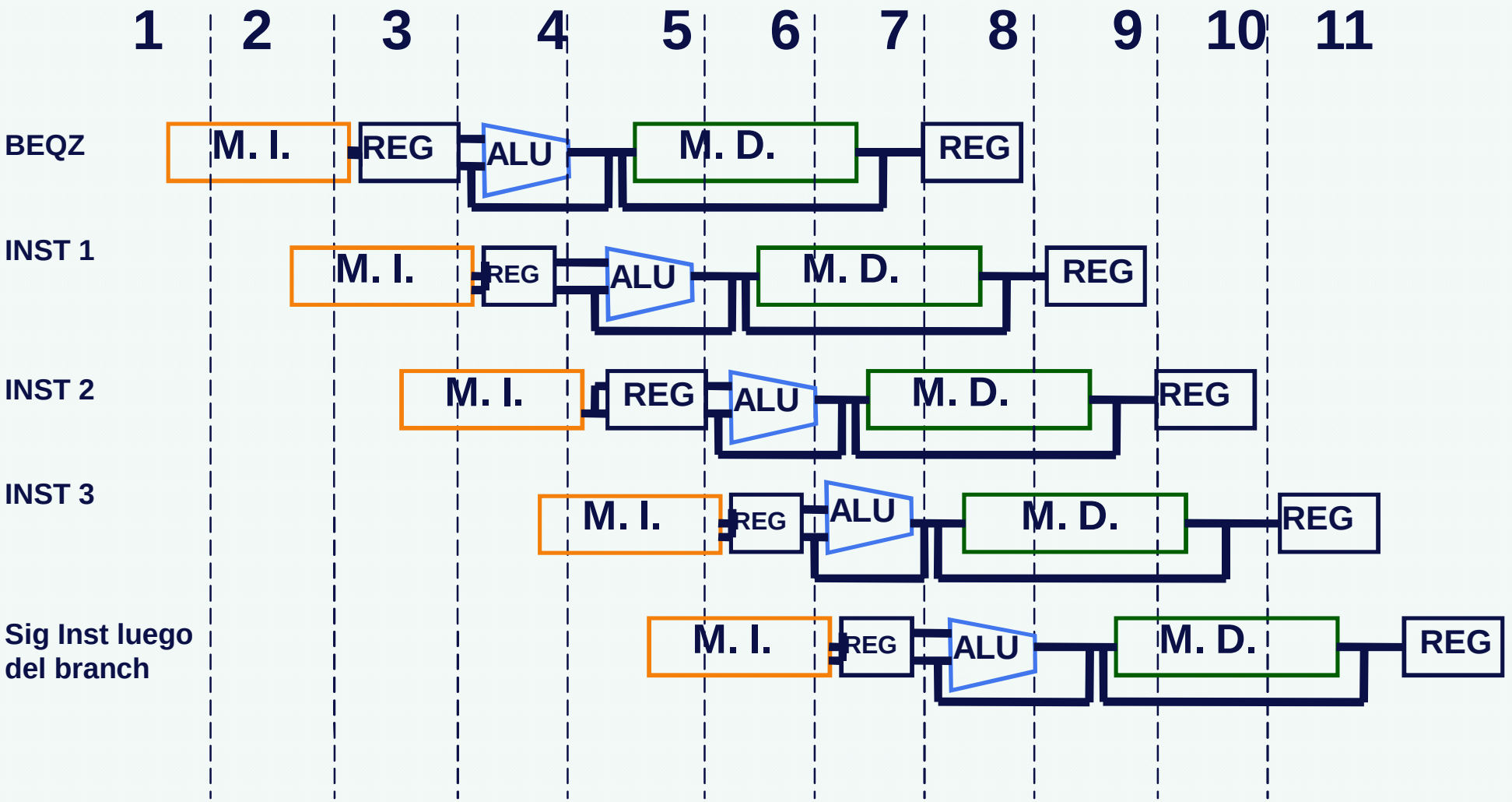
=> debe haber 2 instrucciones o burbujas entre un load y una instrucción dependiente de ésta.



Nótese que el valor está disponible al final de DS del load y se hace un **forwarding** al ALU del ADD=> se usa **ANTES** de la revisión de etiqueta del caché (tag check). Por lo que si hay una falta en cache hay que ñdevolverseòy atrasarse un ciclo. (ocurre menos del 10% de las veces y es fácil detener instrucciones dependientes)

BRANCH TIENE 3 CICLOS DE RETRASO (se resuelve en EX) => CPI = 4 si no hay estrategia para disminuir retraso

(Con branch retrasado se pueden incluir 3 instrucciones luego del branch:)



MIPS SIEMPRE INCLUYE UNA INSTRUCCIÓN POR BRANCH RETRASADO y procesador hace predicción no tomado para las siguientes instrucciones:

(# 1 cuando Branch fue tomado, # 2 cuando Branch no fue tomado)

# 1	1	2	3	4	5	6	7	8	9
BRANCH	IF	IS	RF	EX	DF	DS	TC	WB	
DELAY SLOT		IF	IS	RF	EX	DF	DS	TC	WB
BRANCH + 2			IF	ID					
BRANCH + 3				IF					
B.TARGET					IF	IS	RF	EX	DF

# 2	1	2	3	4	5	6	7	8	9 . . .
BRANCH	IF	IS	RF	EX	DF	DS	TC	WB	
DELAY SLOT		IF	IS	RF	EX	DF	DS	TC	WB
BRANCH + 2			IF	IS	RF	EX	DF	DS	TC . . .
BRANCH + 3				IF	IS	RF	EX	DF	DS . . .
BRANCH + 4					IF IS	RF	EX	DF . . .	

PIPELINE R4000 PARA PUNTO FLOTANTE

8 TIPOS DE ETAPAS

3 unidades funcionales: FP-DIV, FP-MUL, FP-ADD

Un sumador usado en paso final de MUL y DIV

ETAPA	UNIDAD FUNCIONAL	DESCRIPCION
A	FP-ADD	Suma mantisa
D	FP-DIV	Etapa de división
E	FP-MUL	Prueba de Excepción
M	FP-MUL	Primera etapa Multipl.
N	FP-MUL	Segunda etapa Multipl.
R	FP-ADD	Redondeo
S	FP-ADD	ñShiftò de Operando
U	Las 3 unid. Func.	ñUnpackò número FP

OPERACIONES FP EN LA R4000

INST. FP	LATENCIA	INTERVALO INICIACION	SECUENCIA ETAPAS EN EL PIPE
Suma, resta	4	3	U, S + A, A + R, R + S
Multiplicación	8	4	U, E + M, M, M, M, N, N + A, R
División	36	35	U, A, R, D^{27}, D + A, D + R D + R, D + A, D + R, A, R
Raíz Cuadrada	112	111	U, E, $(A + R)^{108}$, A, R
Negación	2	1	U, S
Valor Absoluto	2	1	U, S
Comparación FP	3	2	U, A, R

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 12-bit displacement + contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, load word unsigned, store word (to/from integer registers)
`ld, sd`	Load double word, store double word (to/from integer registers)
`flw, fld, fsw, fsd`	Load SP float, load DP float, store SP float, store DP float
`fmv._.x, fmv.x._`	Copy from/to integer register to/from floating-point register; "__" = S for single-precision, D for double-precision
`csrrw, csrrwi, csrrs, csrrsi, csrrc, csrrci`	Read counters and write status registers, which include counters: clock cycles, time, instructions retired
Arithmetic/logical	*Operations on integer or logical data in GPRs*
`add, addi, addw, addiw`	Add, add immediate (all immediates are 12 bits), add 32-bits only & sign-extend to 64 bits, add immediate 32-bits only
`sub, subw`	Subtract, subtract 32-bits only
`mul, mulw, mulh, mulhsu, mulhu`	Multiply, multiply 32-bits only, multiply upper half, multiply upper half signed-unsigned, multiply upper half unsigned
`div, divu, rem, remu`	Divide, divide unsigned, remainder, remainder unsigned
`divw, divuw, remw, remuw`	Divide and remainder: as previously, but divide only lower 32-bits, producing 32-bit sign-extended result
`and, andi`	And, and immediate
`or, ori, xor, xori`	Or, or immediate, exclusive or, exclusive or immediate
`lui`	Load upper immediate; loads bits 31-12 of register with immediate, then sign-extends
`auipc`	Adds immediate in bits 31–12 with zeros in lower bits to PC; used with `JALR` to transfer control to any 32-bit address
`sll, slli, srl, srli, sra, srai`	Shifts: shift left logical, right logical, right arithmetic; both variable and immediate forms
`sllw, slliw, srlw, srliw, sraw, sraiw`	Shifts: as previously, but shift lower 32-bits, producing 32-bit sign-extended result
`slt, slti, sltu, sltiu`	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch GPR equal/not equal; less than; greater than or equal, signed and unsigned
`jal, jalr`	Jump and link: save PC + 4, target is PC-relative (`JAL`) or a register (`JALR`); if specify `x0` as destination register, then acts as a simple jump
`ecall`	Make a request to the supporting execution environment, which is usually an OS
`ebreak`	Debuggers used to cause control to be transferred back to a debugging environment
`fence, fence.i`	Synchronize threads to guarantee ordering of memory accesses; synchronize instructions and data for stores to instruction memory

Fig 1.5 Algunas instrucciones de RISC-V

Instruction type/opcode	Instruction meaning
Floating point	*FP operations on DP and SP formats*
`fadd.d`, `fadd.s`	Add DP, SP numbers
`fsub.d`, `fsub.s`	Subtract DP, SP numbers
`fmul.d`, `fmul.s`	Multiply DP, SP floating point
`fmadd.d`, `fmadd.s`, `fnmadd.d`, `fnmadd.s`	Multiply-add DP, SP numbers; negative multiply-add DP, SP numbers
`fmsub.d`, `fmsub.s`, `fnmsub.d`, `fnmsub.s`	Multiply-sub DP, SP numbers; negative multiply-sub DP, SP numbers
`fdiv.d`, `fdiv.s`	Divide DP, SP floating point
`fsqrt.d`, `fsqrt.s`	Square root DP, SP floating point
`fmax.d`, `fmax.s`, `fmin.d`, `fmin.s`	Maximum and minimum DP, SP floating point
`fcvt._._`, `fcvt._._u`, `fcvt._u._`	Convert instructions: `FCVT.x.y` converts from type `x` to type `y`, where `x` and `y` are `L` (64-bit integer), `W` (32-bit integer), `D` (DP), or `S` (SP). Integers can be unsigned (`U`)
`feq._`, `flt._`,`fle._`	Floating-point compare between floating-point registers and record the Boolean result in integer register; "__" = `S` for single-precision, `D` for double-precision
`fclass.d`, `fclass.s`	Writes to integer register a 10-bit mask that indicates the class of the floating-point number ($-\infty$, $+\infty$, -0, $+0$, NaN, …)
`fsgnj._`, `fsgnjn._`, `fsgnjx._`	Sign-injection instructions that changes only the sign bit: copy sign bit from other source, the oppositive of sign bit of other source, XOR of the 2 sign bits

Fig 1.6 Algunas operaciones para operandos de punto flotante de RISC-V

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.