

8-by-16 Bit-wide Register File with 1 Write and 2 Read Ports

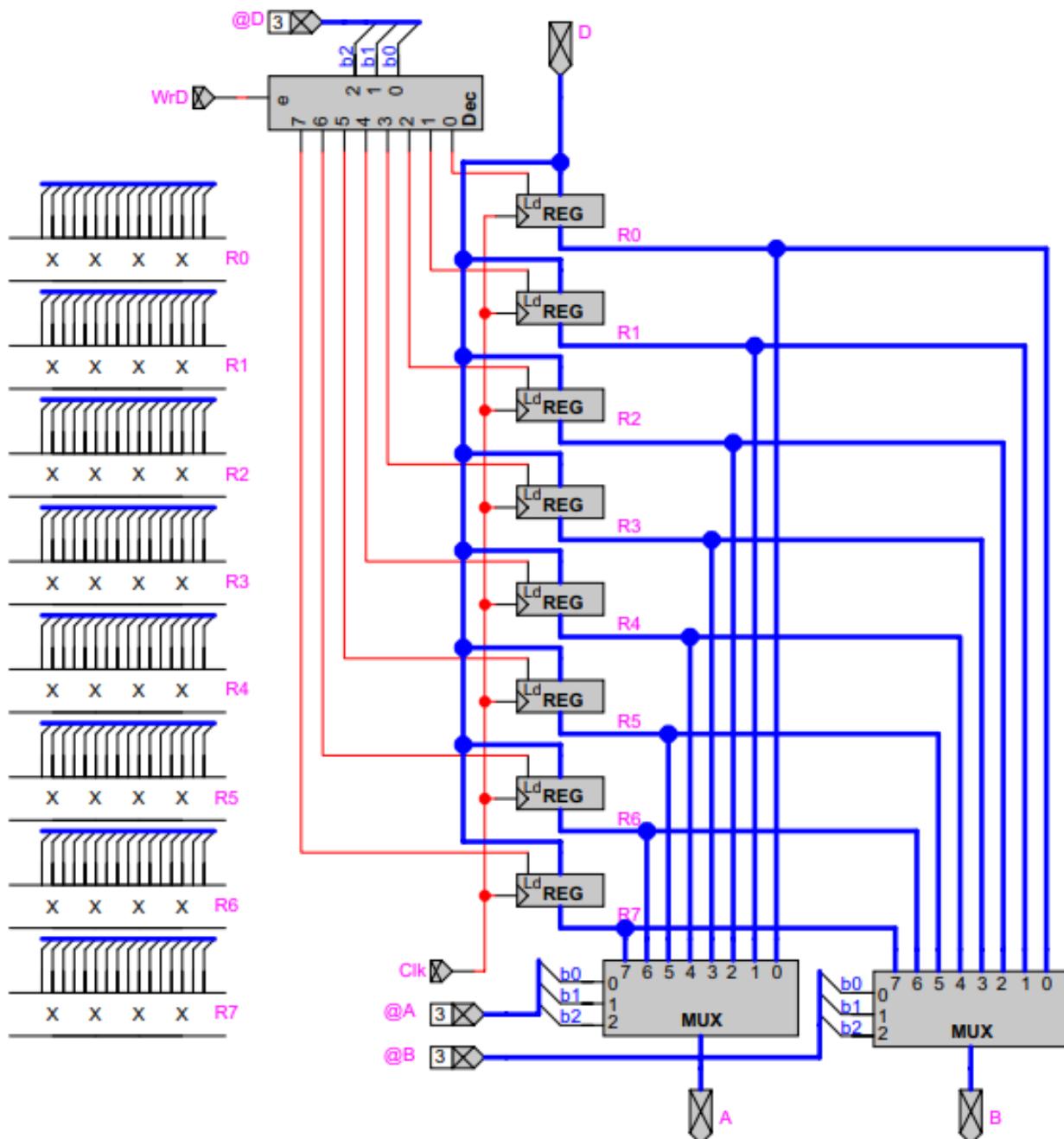
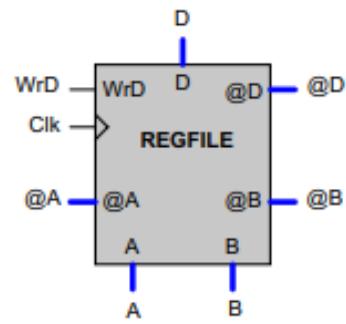
REGFILE

Description:

Register file with 8 16 Bit-wide registers with 2 read ports, A and B, and 1 write port D. @A, @B and @D are the 3-bit addresses of ports A, B and D respectively. If WrD = 1 the port D is written into Register @D when the Clk binary input changes from 0 to 1.

A and B are the contents of Registers @A and @B.

For simulation purposes, a display of the contents of each register is available.



Mnemotécnicos	Palabra de Control de 33 bits								
	@A	@B	Rb/N	OP	F	In/Alu	@D	WrD	N (hexa)
ADD R6, R3, R5	0 1 1	1 0 1	1	0 0	1 0 0	0	1 1 0	1	X X X X
CMPLEU R3, R1, R5	0 0 1	1 0 1	1	0 1	1 0 1	0	0 1 1	1	X X X X
ADDI R7, R1, -1	0 0 1	x x x	0	0 0	1 0 0	0	1 1 1	1	F F F F
ANDI R2, R3, 0xFF00	0 1 1	x x x	0	0 0	0 0 0	0	0 1 0	1	F F 0 0
NOT R4, R2	0 1 0	x x x	x	0 0	0 1 1	0	1 0 0	1	X X X X
MOVE R1, R5	1 0 1	x x x	x	1 0	0 0 0	0	0 0 1	1	X X X X
MOVEI R3, 0xFA02	x x x	x x x	0	1 0	0 0 1	0	0 1 1	1	F A 0 2
IN R2	x x x	x x x	x	x x	x x x	x	1 0 1 0	1	X X X X
OUT R4	1 0 0	x x x	x	x x	x x x	x	x x x	0	X X X X
ANDI -, R3, 0x8000	0 1 1	x x x	0	0 0	0 0 0	x	x x x	0	8 0 0 0
NOP	x x x	x x x	x	x x	x x x	x	x x x	0	X X X X
IN R2 // OUT R4	1 0 0	x x x	x	x x	x x x	x	1 0 1 0	1	X X X X

Fig. 8.9 Ejemplos de acciones que se pueden hacer en la UP en un ciclo y las palabras de control de 33 bits asociadas a cada acción

@A y @B son XXX cuando es un IN o un MOVEI

Rb/N és X → cuando el registro **@B** es XXX

Rb/N és 0 → cuando el **registro N** se está utilizando (ADDI, MOVI)

Rb/N és 1 → cuando se utilizan todos los registros.

OP y F són XXX cuando es un IN, o un OUT.

In/Alu és X → cuando es un OUT o una función que no registra (ANDI -, R3, 0x8000)

In/Alu és 1 → cuando es un IN

@D és X → cuando es un OUT o una función que no registra (ANDI -, R3, 0x8000)

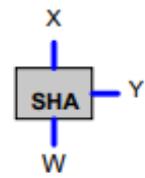
WRD és 0 → cuando es un OUT o una función que no registra (ANDI -, R3, 0x8000)

16 Bit-wide Arithmetic Shifter

SHA

Description:

The 16-bit output W is the result of shifting the 16-bit input X the number of bits coded by bits 0 to 4 of the 16-bit input Y. Y(b4:b0) represents a signed integer in two's-complement. The number of shifting bits is the absolute value of the number represented in Y. If this value is positive, the shifting is to the left and if it is negative the shifting is to the right. When SHA shifts to the left the less significant bits of W are set to 0 and the most significant bits of X are lost. When SHA shifts to the right bit X(15) is sign-extended into the most significant bits of W and the less significant bits of X are lost.



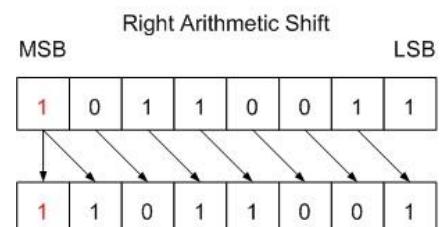
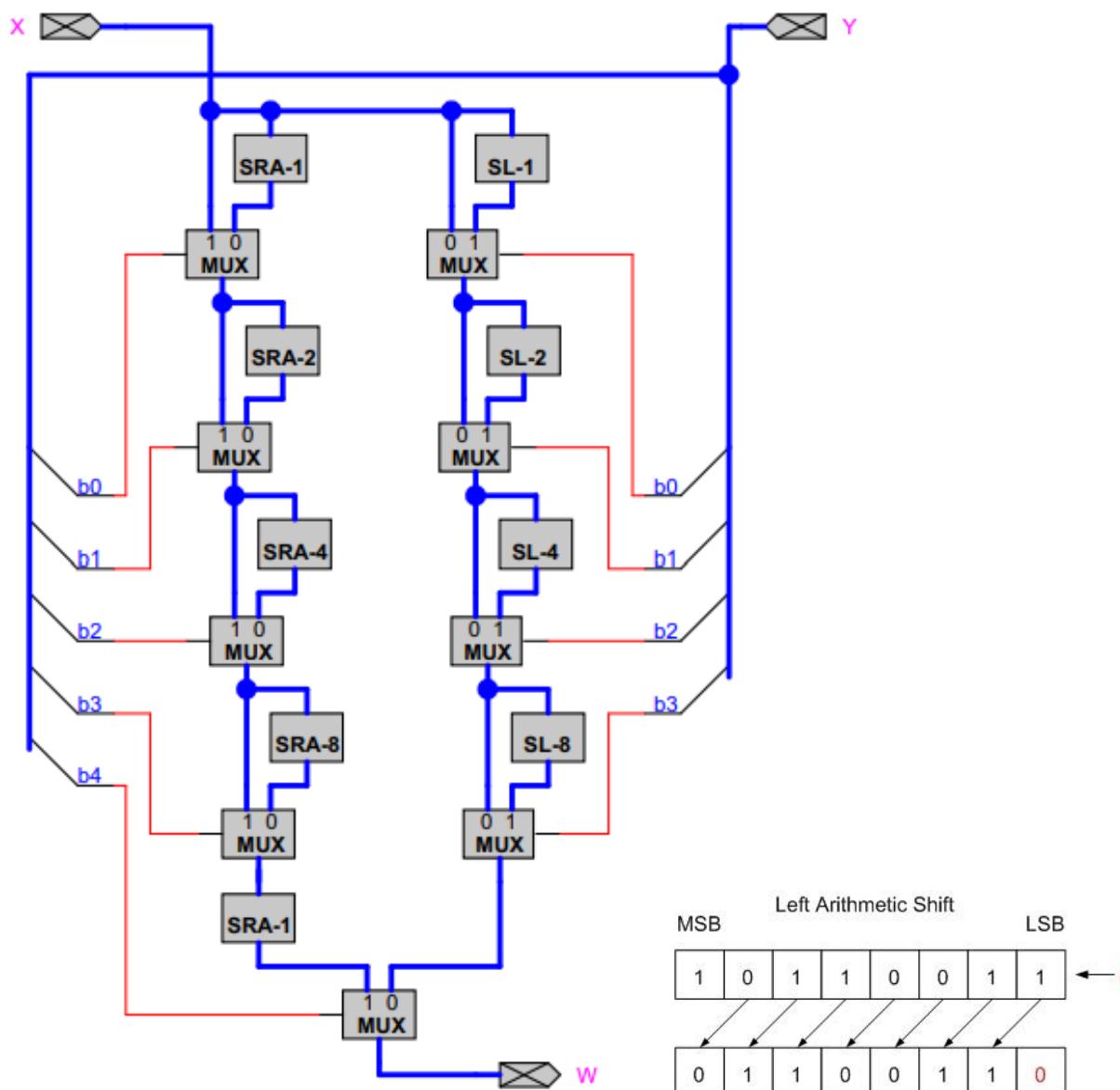
Arithmetic Operation:

Interpreting X, Y(b4:b0) and W as signed integers,
if $(-2^{15}) \leq (X_s * (2^{b4}Y_{b4})) \leq (2^{15}-1)$ then $W_s = X_s * (2^{b4}Y_{b4})$
(The integer division of powers of 2 (when $Y(b4:b0)_s$ is negative) is done with positive remainder)

Bit-level Logical Operation:

```

if (Y(b4:b0)_s >= 0) then
    W(bi) = 0 for i = 0 to Y(b4:b0)_s - 1
    W(bi) = X(bi-Y(b4:b0)_s) for i = Y(b4:b0)_s to 15
if (Y(b4:b0)_s < 0) then
    W(bi) = X(bi-Y(b4:b0)_s) for i = 0 to -Y(b4:b0)_s - 1
    W(bi) = X(b15) for i = -Y(b4:b0)_s to 15
  
```

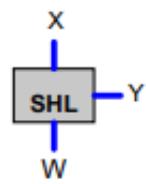


16 Bit-wide Logic Shifter

SHL

Description:

The 16-bit output W is the result of shifting the 16-bit input X the number of bits coded by bits 0 to 4 of the 16-bit input Y. Y(b0:b4) represents a signed integer in two's-complement. The number of shifting bits is the absolute value of the number represented in Y. If this value is positive, the shifting is to the left and if it is negative the shifting is to the right. When SHL shifts to the left the less significant bits of W are set to 0 and the most significant bits of X are lost. When SHL shifts to the right the most significant bits of W are set to 0 and the less significant bits of X are lost.



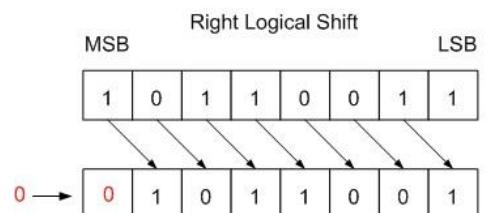
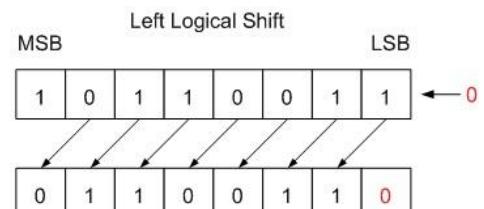
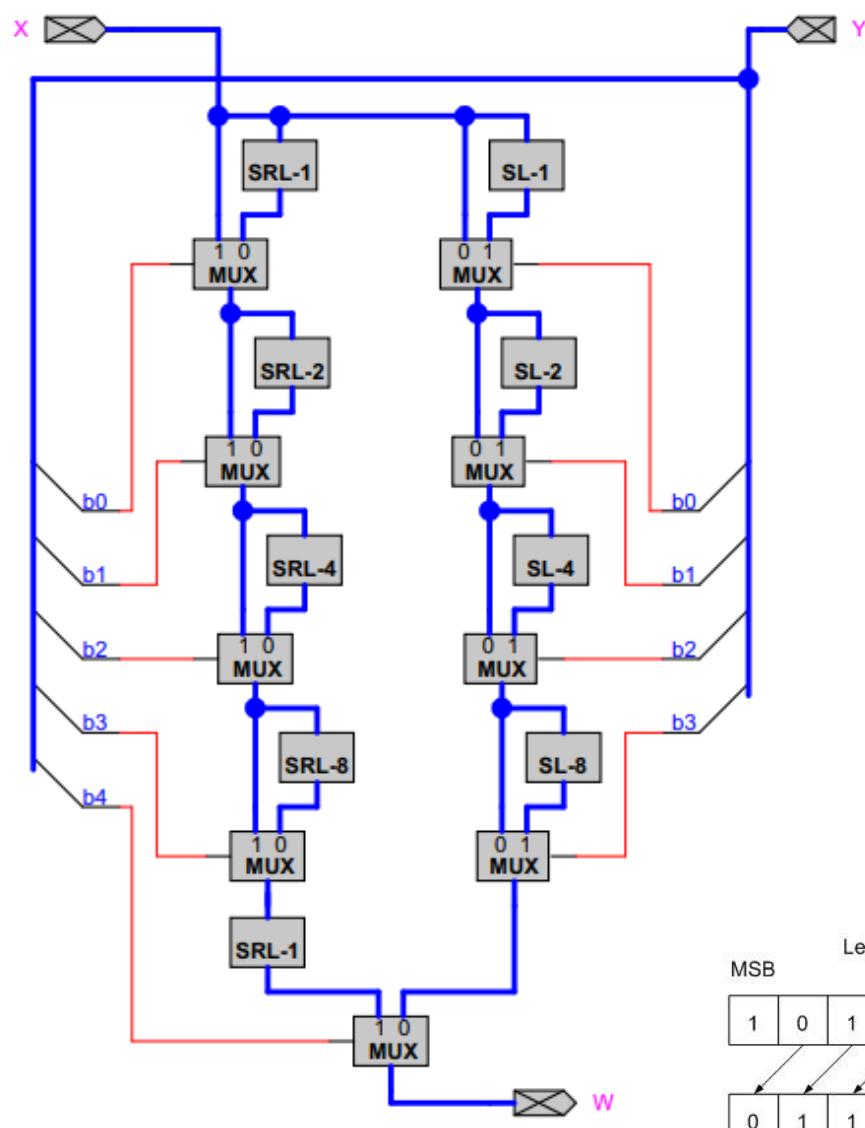
Arithmetic Operation:

Interpreting X and W as unsigned integers and Y as signed integer
 $\text{if } ((\text{X}_u * (2^{16} - 1)) \leq (\text{Y}_u * (2^{16} - 1))) \text{ then } \text{W}_u = \text{X}_u * (2^{16} - 1)$

Bit-level Logical Operation:

```

if (Y(b0:b4)s >= 0) then
    W(bi) = 0 for i = 0 to Y(b0:b4)s - 1
    W(bi) = X(bi - Y(b0:b4)s) for i = Y(b0:b4)s to 15
if (Y(b0:b4)s < 0) then
    W(bi) = X(bi - Y(b0:b4)s) for i = 0 to -Y(b0:b4)s - 1
    W(bi) = 0 for i = -Y(b0:b4)s to 15
  
```

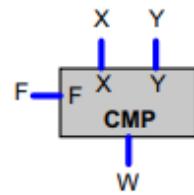


16 Bit-wide Signed and Unsigned Comparator

CMP

Description:

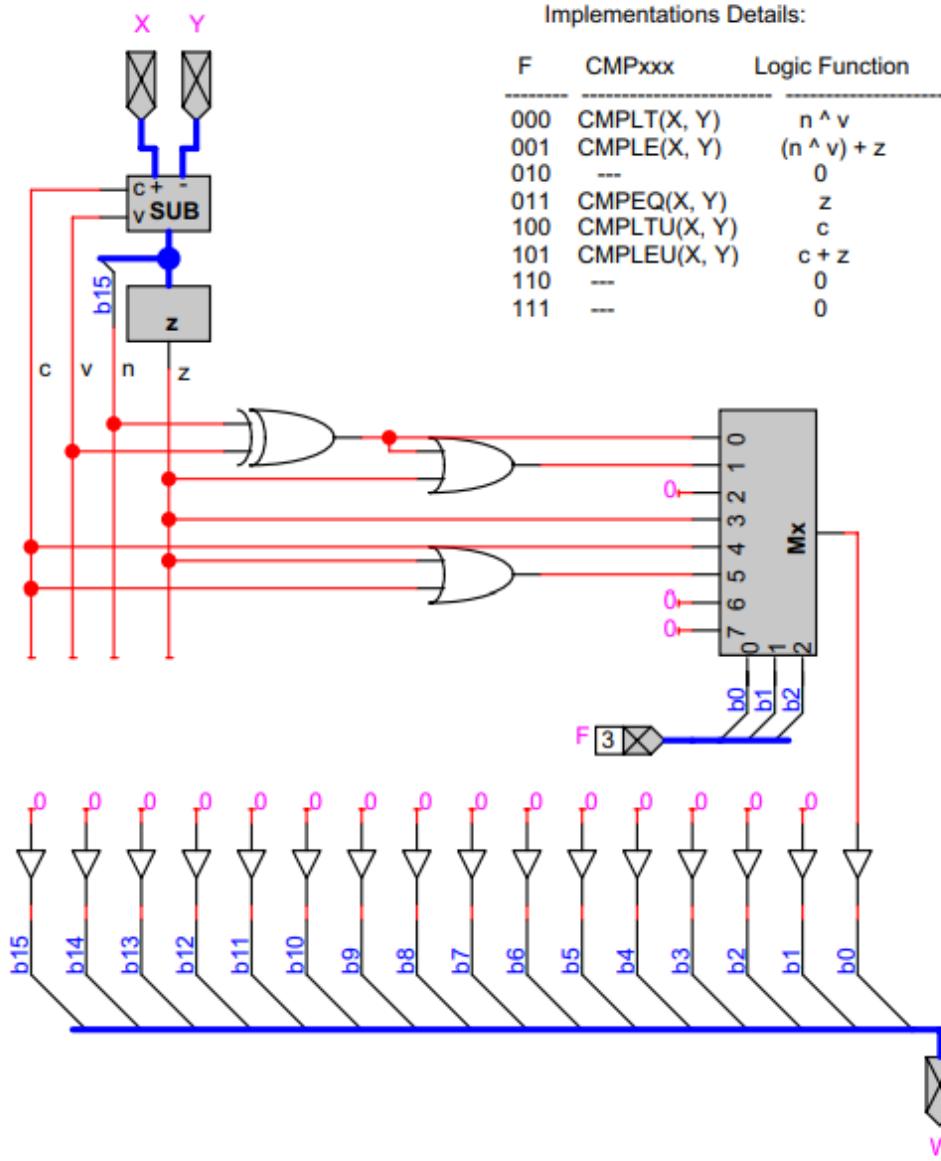
The 16-bit output W has only two possible values as the result of a comparison: true or false. True is coded as $W(b_0) = 1$ and false as $W(b_0) = 0$.
 For all the cases $W(b_i) = 0$ for $i = 1$ to 15 . The type of comparison and the consideration of the 16-bit inputs X and Y as signed or as unsigned integers is chosen by the 3-bit selection input F according to the next table:



F	W	CMPxx(X, Y)	Name
000	$X_s < Y_s$	CMPLT(X, Y)	Less Than (Signed)
001	$X_s \leq Y_s$	CMPLE(X, Y)	Less than or Equal (Signed)
010	---	---	
011	$X == Y$	CMPEQ(X, Y)	Equal
100	$X_u < Y_u$	CMPLTU(X, Y)	Less Than Unsigned
101	$X_u \leq Y_u$	CMPLEU(X, Y)	Less than or Equal Unsigned
110	---	---	
111	---	---	

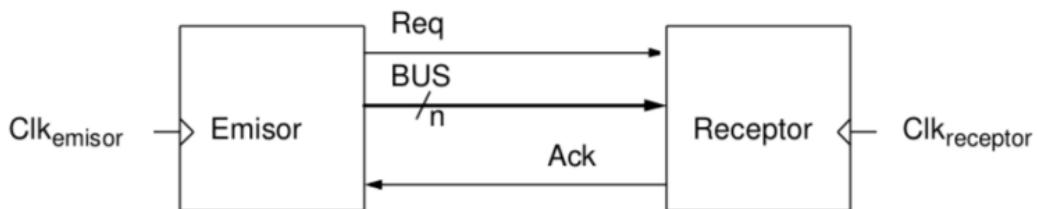
Implementations Details:

F	CMPxxx	Logic Function
000	CMPLT(X, Y)	$n \wedge v$
001	CMPLE(X, Y)	$(n \wedge v) + z$
010	---	0
011	CMPEQ(X, Y)	z
100	CMPLTU(X, Y)	c
101	CMPLEU(X, Y)	$c + z$
110	---	0
111	---	0



- Connectarem perifèrics al nostre computador
 - Dispositius mitjançant els quals es comunica amb l'exterior
 - teclat, impressora,
 - Els perifèrics realitzen operacions d'entrada/sortida (E/S)
 - Input/Output (I/O)
- Categorització dels perifèrics:
 - Els perifèrics d'entrada en enviaran dades a través del bus RD-IN
 - Exemple: teclat
 - Els perifèrics de sortida rebran les dades a través del bus WR-OUT
 - Exemple: impressora
- La comunicació entre UPG i perifèrics serà **asíncrona**
 - Els perifèrics **no** poden compartir senyal de rellotge amb la UPG
 - Diferents velocitats de procés
 - Distància entre UPG i perifèric
- Caldrà identificar quin *port* es vol llegir/escriure
 - Cada *port* tindrà associat un nombre natural de 8 bits
 - $2^8 = 256$ *ports* d'entrada i 256 *ports* de sortida
 - Aquest identificador l'anomenarem ADDR-IO
 - ADDR-IO formarà part de la paraula de control generada per la UC
- Reformulació acció IN
 - Llegeix un dels 256 *ports* d'entrada i, al final del cicle, escriu el seu valor a un registre
 - Exemple:
 - IN R1, 5
 - Llegeix el *port* d'entrada número 5 i guarda el seu contingut a R1
- Reformulació acció OUT
 - Escriu a un dels 256 *ports* de sortida el valor d'un dels registres del banc de registres de la UPG
 - Exemple:
 - OUT 4, R2
 - Escriu el valor de R2 al *port* de sortida número 4
 - La UC generarà un nou bit de control: Wr-Out
 - Pels busos WR-OUT i ADDR-IO sempre hi ha algun valor
 - Wr-Out indica si en aquest cicle realment s'està fent una acció OUT
 - És a dir, si aquest cicle el valor dels busos WR-OUT i ADDR-IO és vàlid
 - Anàleg al bit de control WrD del banc de registres

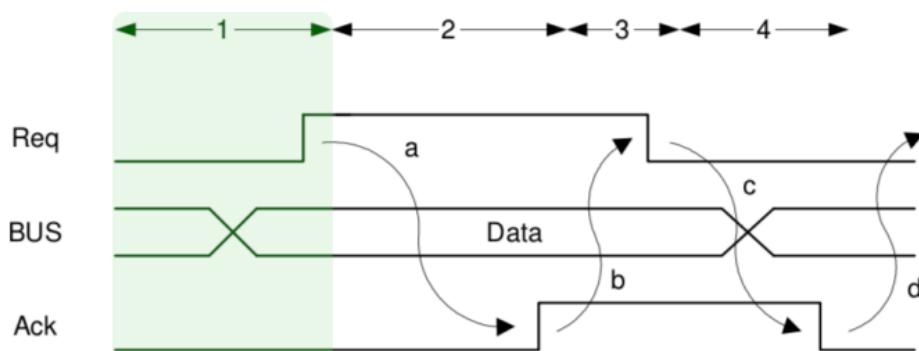
- Comunicar un emissor i un receptor sense compartir senyal de rellotge
 - Cal utilitzar un protocol de comunicació asíncron
 - **Four-phase handshake**
- Senyals que hi intervendran:
 - De control: *request* i *acknowledgment* d'un bit cadascun
 - Estat inicial: senyals *request* i *acknowledgment* a "0"
 - De dades: BUS: bus de n bits



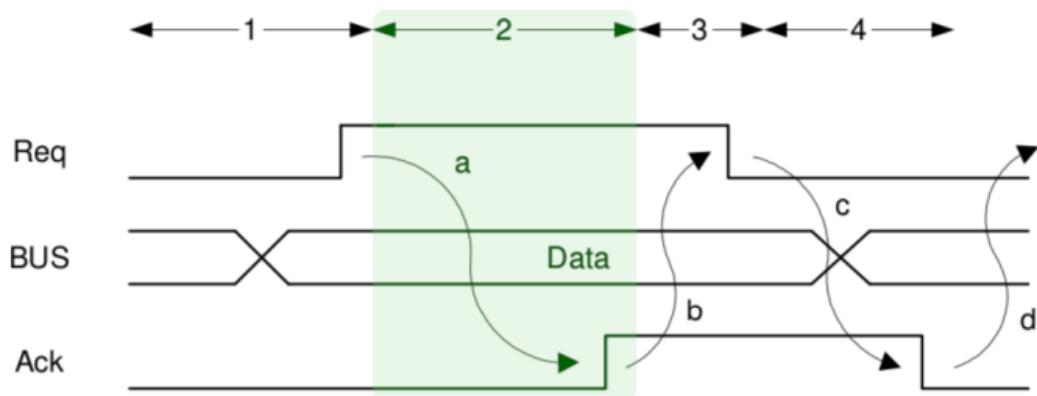
Protocol *four-phase handshake*: fase 1



- Des de l'estat inicial ($request = acknowledgment = "0"$)
 - L'emissor posa la dada a transmetre al bus de dades i la manté estable
 - L'emissor indica al receptor que hi ha una dada disponible
 - Posa el senyal *request* a "1" i el manté estable
- És clau l'ordre d'activació dels senyals
 - Altrament, el receptor podria llegir una dada que encara no és estable

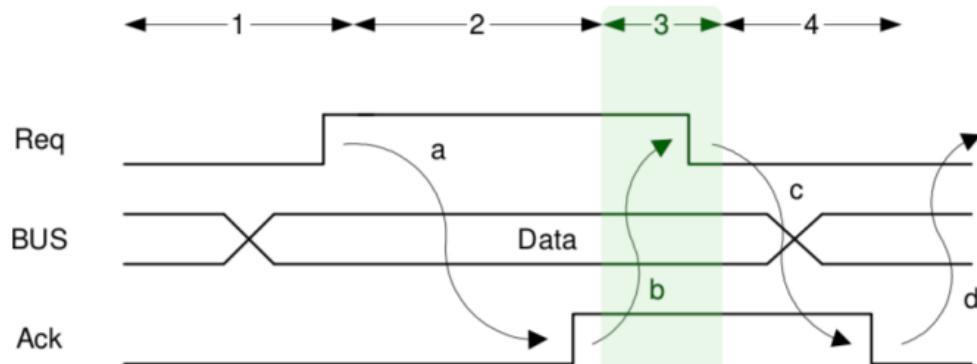


- Quan el receptor observa el canvi al senyal *request*
 - El receptor llegeix la dada del bus de dades
 - El receptor indica a l'emissor que ja l'ha llegida
 - Posa a el senyal *acknowledgment* a "1" i el manté estable
 - A partir d'ara, l'emissor pot retirar la dada del bus
- És clau l'ordre d'activació dels senyals
 - Altrament, l'emissor podria retirar la dada abans de ser llegida

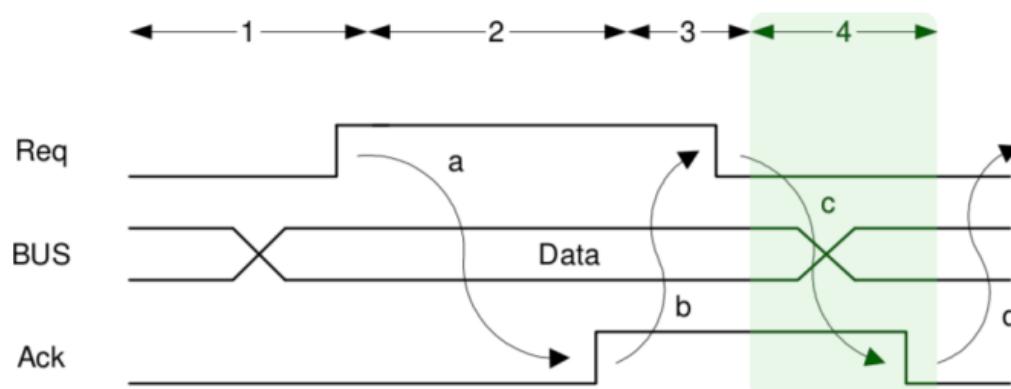


Protocol four-phase handshake: fase 3

- Quan l'emissor observa el canvi al senyal *acknowledgment*
 - L'emissor torna al seu estat inicial
 - El senyal *request* torna a valer "0" i el manté estable
 - Return to zero* (RZ)
 - Ja no cal mantenir la dada estable al bus

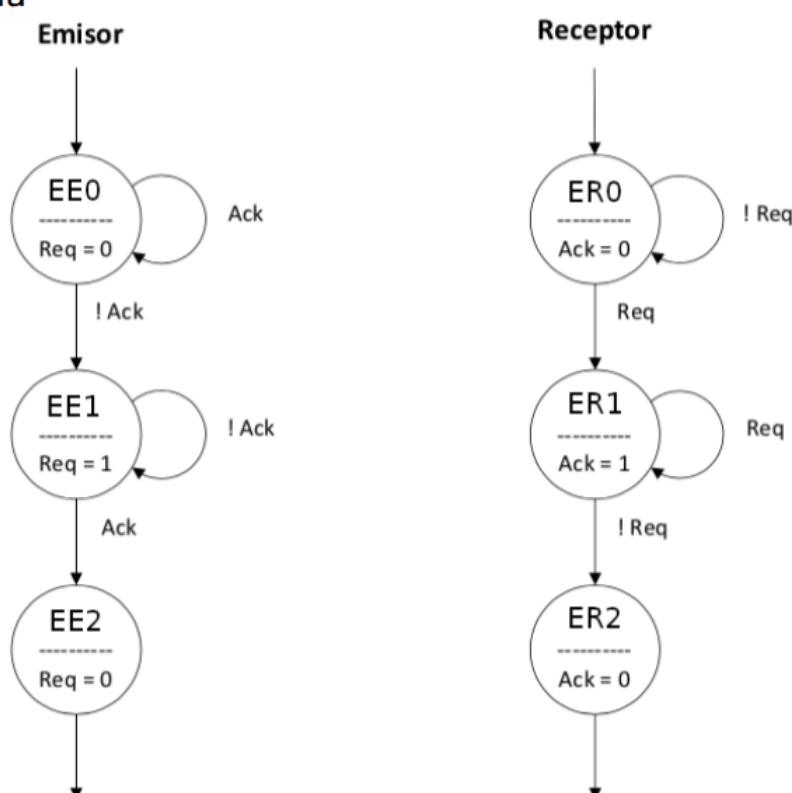


- Quan el receptor observa el canvi al senyal *request*
 - El receptor torna al seu estat inicial
 - El senyal *acknowledgment* torna a valer "0" i el manté estable
 - Return to zero* (RZ)
- A partir d'ara, l'emissor pot iniciar una nova transferència

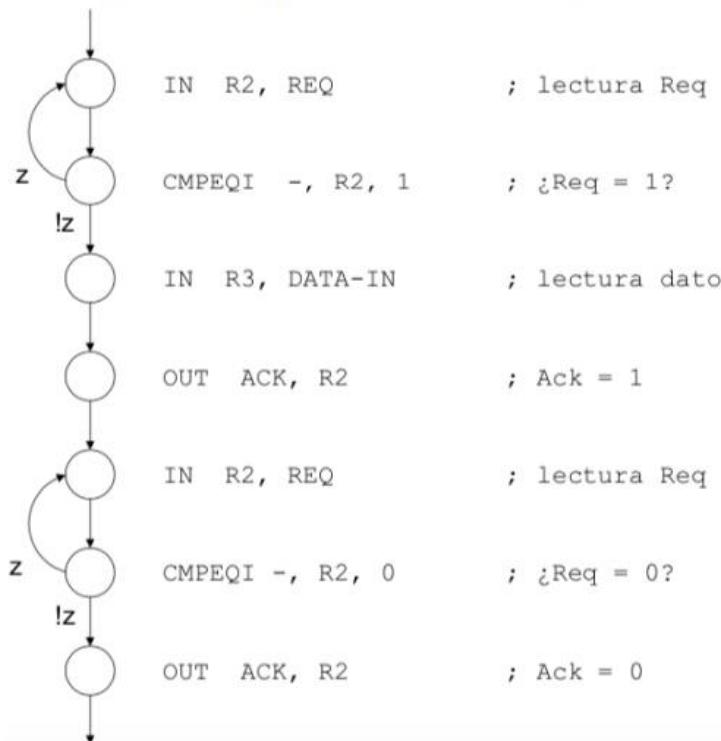


Protocol 4 fases: diagrama d'estats

- Fragments dels grafs d'estats de l'emissor i del receptor per fer una transferència

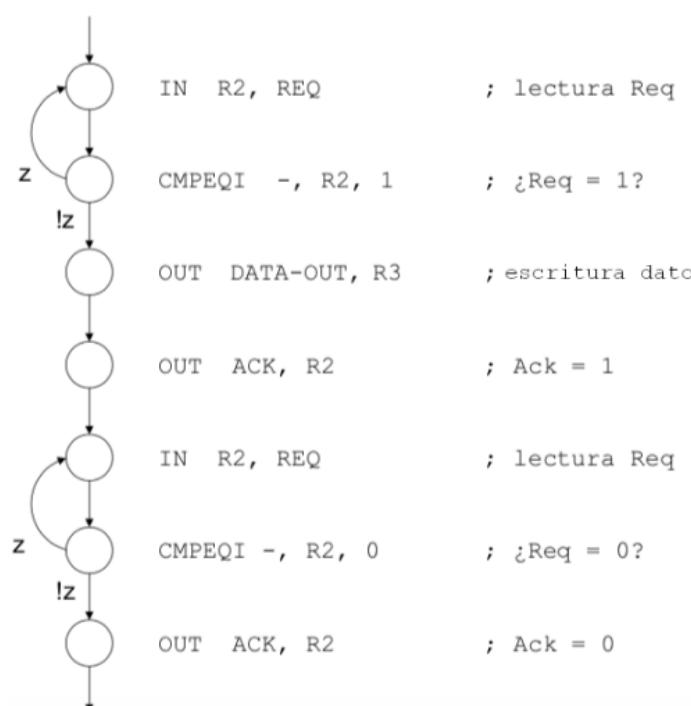


- REQ, DATA-IN i ACK són els identificadors dels *ports* que emmagatzemen senyal de *request* (*port* d'estat), busos de dades i senyal de *acknowledgment* (*port* de control)

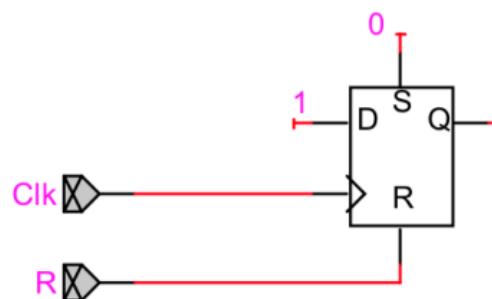


Protocol 4 fases a la UPG: escriptura

- El perifèric és el receptor i la UPG l'emissor
- DATA-OUT és l'identificador del *port* que emmagatzema bus de dades

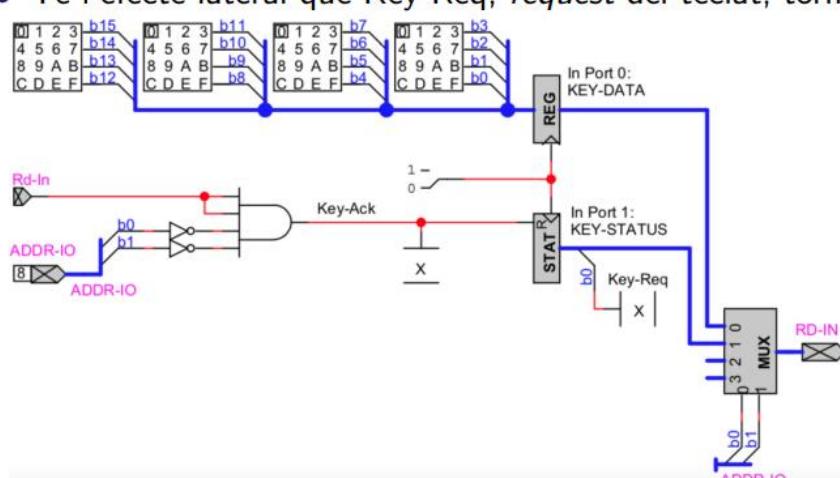


- Utilitza biestable D activat per flanc ascendent i amb entrades asíncrones de posada a "0" (Reset) i a "1" (Set)
 - Quan es produueixi flanc ascendent a Clk, el valor a D passa a Q
 - Si S val 1, Q es posa a "1" immediatament
 - Si R val 1, Q es posa a "0" immediatamente
- Implementació del port de control
 - Bit 0:
 - Aquest biestable definirà el bit 0 del port de control
 - Quan hi hagi flanc ascendent a Clk, Q valdrà "1" (*request = "1"*)
 - L'entrada R el posarà a "0" asíncronament (*request = "0"*)**
 - Mai el posarem a "1" asíncronament
 - La resta de bits del port valdran "0"

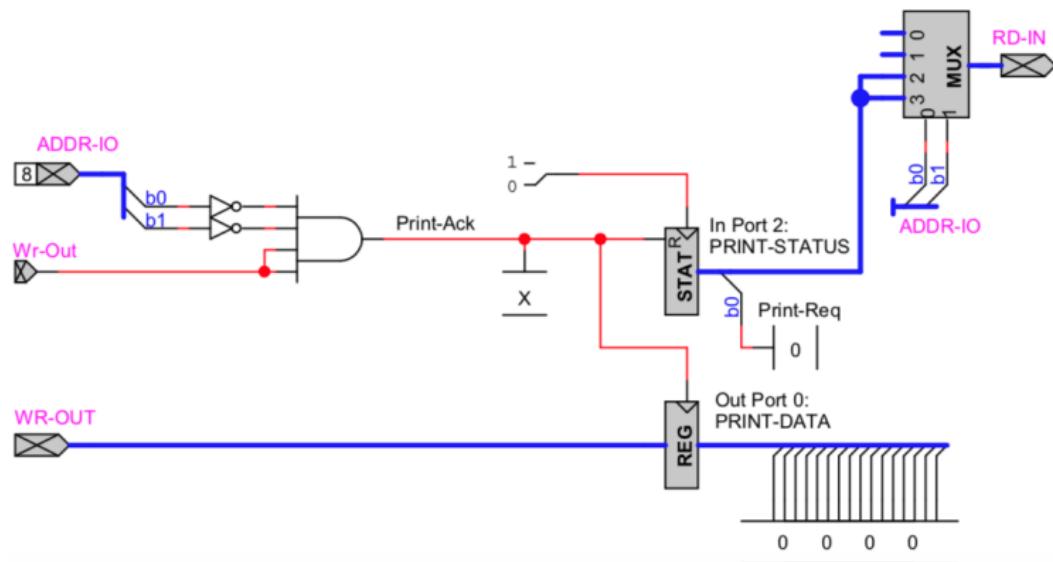


Efecte lateral llegint port dades teclat

- Key-Req es posa a "1" amb el *Binary switch* que fa de Clk als ports
- Al fer IN sobre el port KEY-DATA
 - El valor de KEY-DATA surt per RD-IN
 - La AND-4 fa que Key-Ack valgui "1"
 - Té l'efecte lateral que Key-Req, request del teclat, torni a "0"



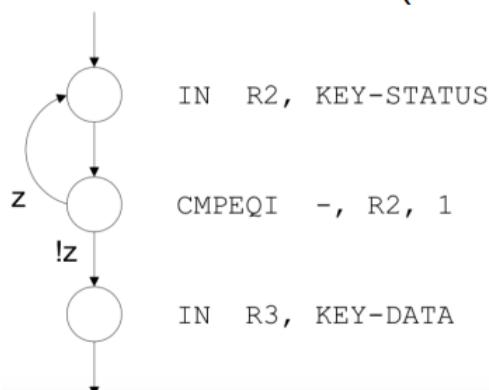
- Print-Req es posa a "1" amb el *Binary switch* que fa de Clk als *ports*
- Al fer OUT sobre el *port* de PRINT-DATA
 - S'escriu el valor del bus WR-OUT al *port* PRINT-DATA
 - La AND-4 fa que Print-Ack val "1"
 - Té l'efecte lateral que Print-Req, *request* de la impressora, torni a "0"



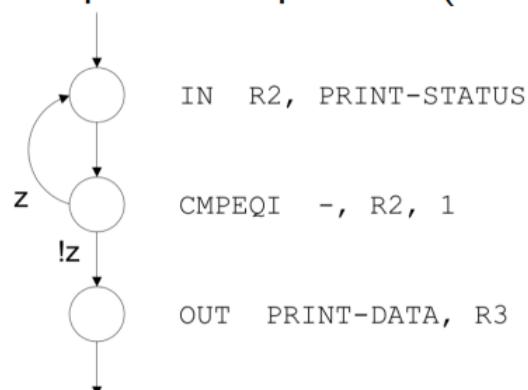
Efecte lateral: accions i graf d'estats UC

- Fragments graf d'estats UC
 - KEY-DATA i PRINT-DATA són constants que representen els identificadors dels *ports* de dades de teclat i impressora
 - KEY-STATUS i PRINT-STATUS són constants que representen els identificadors dels *ports* d'estat del teclat i impressora, senyal *request* és al bit de menys pes del *port* (i resta de bits del *port* a "0")

Lectura de teclat (a R3)



Escriptura a impressora (de R3)

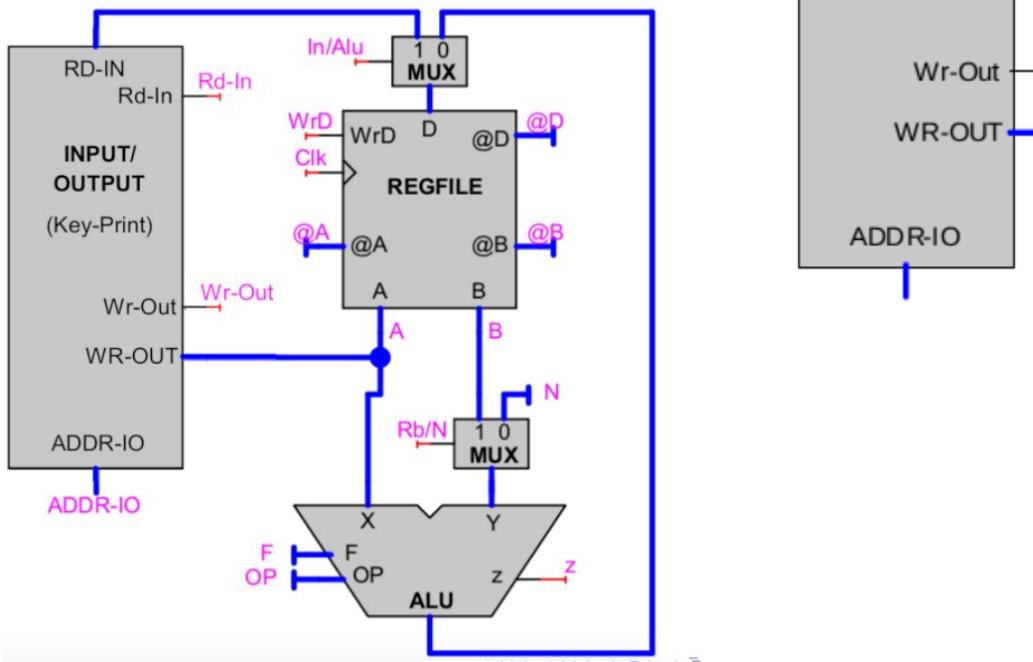


- Senyals d'entrada:

- Senyals de control Rd-In i Wr-Out
- Bus d'adreça ADDR-IO
- Bus de dades WR-OUT

- Senyals de sortida:

- Bus de dades RD-IN

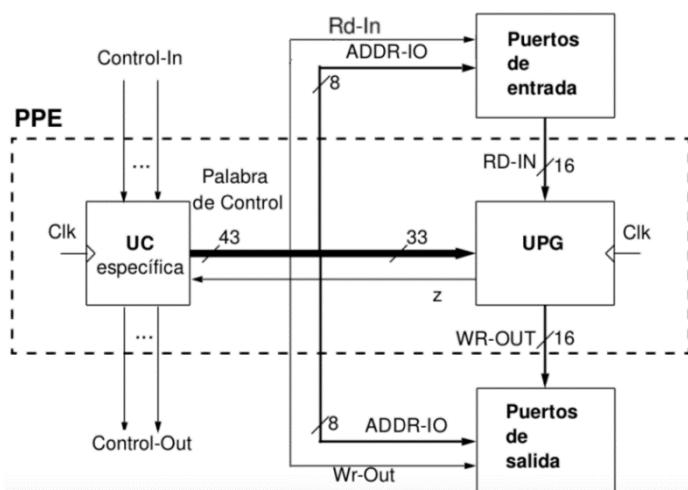


- S'afegeixen 10 bits a la paraula de control del tema anterior

- ADDR-IO: bus de 8 bits amb l'identificador de *port*
- Wr-Out: senyal binari que indica si en aquest cicle es fa l'acció OUT
- Rd-In: senyal binari que indica si en aquest cicle es fa l'acció IN

- La nova paraula de control té 43 bits:

	@A	@B	Rb/N	OP	F	In/Alu	@D	WrD	Wr-Out	Rd-In	N (hexa)	ADDR-IO (hexa)
IN R2, 33	x x x	x x x	x	x x x	x x x	1 0 1 0	1	0 1	1 0 1	X X X X	2 1	
OUT 0x50, R1	0 0 1	x x x	x	x x x	x x x			0 1 0	X X X X	5 0		
ADD R1, R2, R3	0 1 0	0 1 1	1 1 0	0 0 1	0 0 0		0 0 0 0 1 1 0 0	0 0 0 0 1 1 0 0	X X X X X X X X			



- La memòria utilitzada a la UC pot ser una ROM o una RAM
 - ROM: *Read Only Memory*
 - RAM:
 - El seu contingut es pot modificar per a adaptar-se a un nou problema
- Sigui ROM o RAM, d'aquesta memòria en direm I-MEM
 - *Instruction Memory*
 - Cada paraula de la I-MEM és una **instrucció de llenguatge màquina**
 - A la ROM combinada, cada instrucció de LM ocupa 75 bits!
- Del conjunt d'instruccions que implementen el graf d'estats de la UC en direm **programa en llenguatge màquina**
- El registre d'estat de la UC ...
 - Conté un valor natural entre 0 i $2^k - 1$
 - Identifica la fila de la I-MEM on es troba la instrucció de llenguatge màquina que s'executa al cicle actual
 - **D'aquest registre en direm PC (*Program Counter*)**
 - També IP (*Instruction Pointer*) o IAR (*Instruction Address Register*)

• En funció del propòsit del processador

- Processadors de propòsit específic (*empotrats, embedded*)
 - Acostumen a seguir model Harvard
 - El programa està a una memòria ROM i sempre serà el mateix
- Processadors de propòsit general
 - Segueixen model von Neumann
 - Una memòria RAM conté programa i dades
 - El programa es pot canviar quan ho considerem oportú

Objectiu del tema

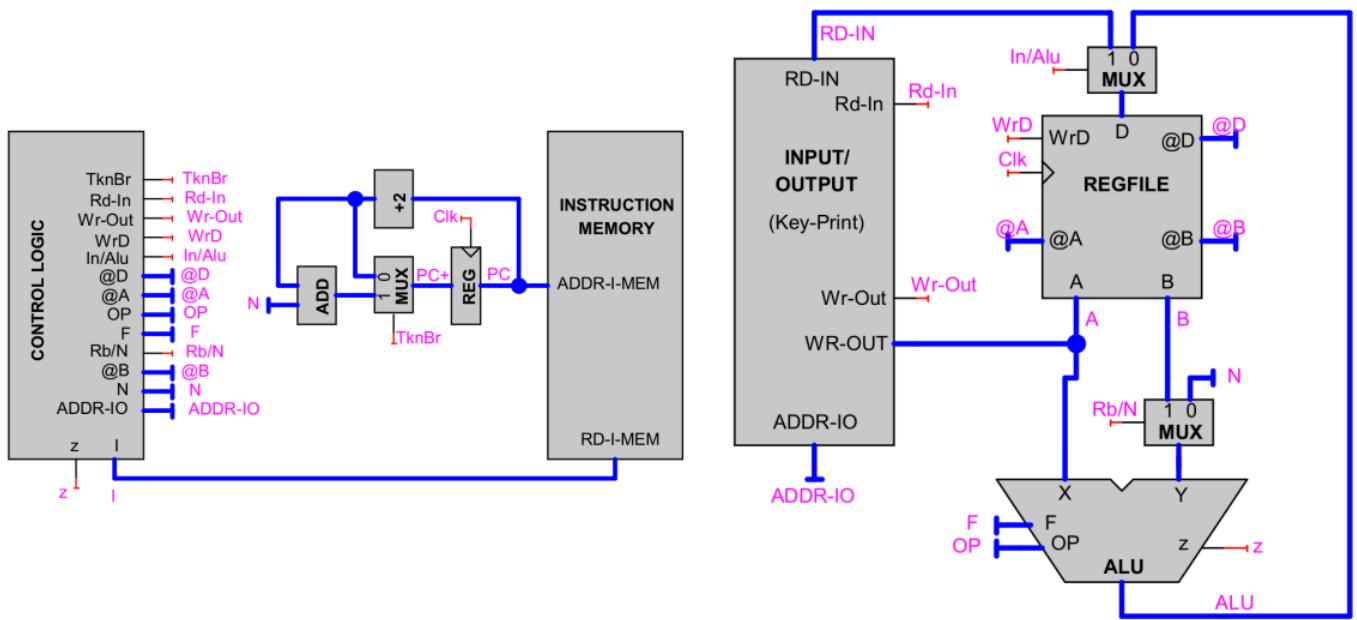


- Reduir la mida de les paraules de la I-MEM
 - Passarem de 75 bits/instrucció a 16 bits/instrucció
 - Crearem el llenguatge màquina SISA
 - *Simple Instruction Set Architecture*
 - Cada instrucció SISA ocuparà 16 bits
 - Per legibilitat, crearem llenguatge *assembler*
 - Semblant als mnemotècnics de la UPG
 - Convertirem entre *assembler* i llenguatge màquina i a l'inrevés
 - Al registre d'estat de la UC en direm PC (*Program Counter*)
 - Adreça de la I-MEM on es troba la instrucció a executar aquest cicle

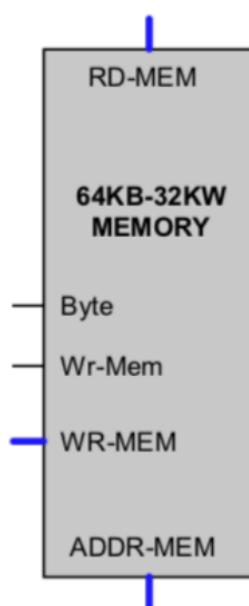
- Aritmètic-Lògiques (AL):
 - **AND, OR, XOR, NOT, ADD, SUB, SHA, SHL**
 - Llevat **NOT**, operen amb dos registres font i actualitzen el registre destí
- Comparació (CMP):
 - **CMPLT, CMPLE, CMPEQ, CMPLTU, CMPLEU**
 - Anàleg a les Aritmètic-Lògiques
- De salt:
 - **BZ, BNZ**
- Entrada/Sortida:
 - **IN, OUT**
 - Lectura/Escriptura d'un *port* d'entrada/sortida
- Càrrega de valors immediats:
 - **MOVI, MOVHI**
 - Càrrega part baixa (amb extensió de signe)/part alta
- Suma amb valor immediat de fins a 6 bits:
 - **ADDI**
 - Útil per actualitzar comptadors sense haver de carregar l'increment a un registre



Grafo de estados para la UPG		Ensamblador SISA
a)		ADD R3, R4, R5 <div style="background-color: #f0f0f0; padding: 5px;">ADD R3, R4, R5</div>
b)		SHAI R4, R3, -3 <div style="background-color: #f0f0f0; padding: 5px;">MOVI R7, -3</div> <div style="background-color: #f0f0f0; padding: 5px;">SHA R4, R3, R7</div>
c)		IN R0, 1 // OUT 1, R6 <div style="background-color: #f0f0f0; padding: 5px;">IN R0, 1</div> <div style="background-color: #f0f0f0; padding: 5px;">OUT 1, R6</div>
d)		MOVEI R5, 0x3AF6 <div style="background-color: #f0f0f0; padding: 5px;">MOVI R5, 0xF6</div> <div style="background-color: #f0f0f0; padding: 5px;">MOVHI R5, 0x3A</div>
e)		MOVEI R3, -16 <div style="background-color: #f0f0f0; padding: 5px;">MOVI R3, 0xF0</div>



- Afegir un mòdul de memòria RAM al nostre computador
 - RAM = *Random Access Memory*
 - Volàtil (necessita subministrament elèctric per mantenir el contingut)
 - Hi podrem emmagatzemar "molta" informació amb un cost reduït
 - És més barata i gran que el banc de registres però més lenta
 - La RAM té un seguit de posicions
 - Cada posició s'identifica amb una adreça (un natural entre 0 i $2^n - 1$)
 - A cada posició s'hi emmagatzema el mateix nombre de bits (8, un *byte*)
 - El temps d'accés és independent de la posició accedita
 - S'hi podran fer lectures i escriptures de *byte* i de *word* (16 bits)
 - Asíncron
- El mòdul haurà de permetre una lectura/escriptura a *byte/word* cada cicle de la UPG
- El construirem amb mida 2^{16} bytes, equivalent a 2^{15} words
 - MEM-64KB-32KW
- Cada posició del mòdul de memòria té una adreça (@) de 16 bits
 - Des de l'adreça 0x0000 fins a la 0xFFFF
 - 2^{16} adreces diferents
- Adreçament a nivell de *byte*
 - El mòdul emmagatzema 2^{16} bytes
 - Des de l'adreça 0x0000 a la 0xFFFF
- Adreçament a nivell de *word*
 - El mòdul emmagatzema 2^{15} words
 - Adreces "parells" 0x0000, 0x0002, ... 0xFFFF
 - El *word* està format per dos *bytes* amb adreces consecutives
 - Els indicats per l'adreça i per adreça + 1
 - El *byte* de menys pes del *word* és el l'adreça "parell"
 - Emmagatzemament *Little endian*
- Entrades:
 - ADDR-MEM: bus de 16 bits amb l'adreça a accedir
 - Byte: senyal d'un bit que indica que si l'accés és a *byte* ("1") o a *word* ("0")
 - Wr-Mem: senyal d'un bit que indica si es fa una escriptura ("1") o una lectura ("0")
 - WR-MEM: bus de 16 bits amb la dada a escriure
- Sortides:
 - RD-MEM: bus de 16 bits amb la dada llegida
- Compte amb la temporalitat!
 - A les escriptures, Wr-Mem hauria de passar a "1" quan els busos ADDR-MEM i WR-MEM tinguin un valor estable

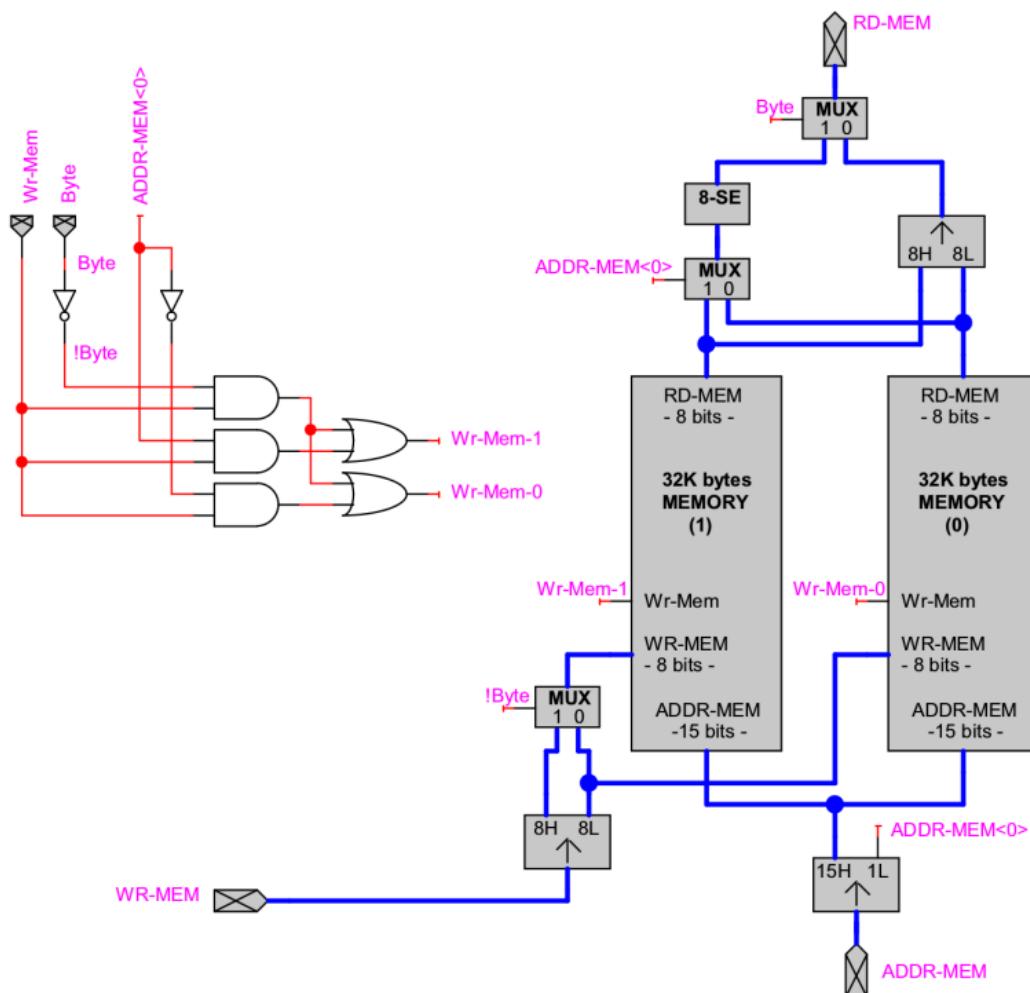


- *Load Byte*: **LDB** Rd , N6(Ra)
 - $Rd \leftarrow SE(MEM_b[Ra + SE(N6)])$; $PC \leftarrow PC + 2$;
 - Fa extensió de signe a 16 bits del byte llegit de memòria
- *Store Byte*: **STB** N6(Ra) , Rb
 - $MEM_b[Ra + SE(N6)] \leftarrow Rb < 7..0 >$; $PC \leftarrow PC + 2$;
 - Només s'emmagatzema a memòria el byte baix de Rb
- *Load Word*: **LD** Rd , N6(Ra)
 - $Rd \leftarrow MEM_w[(Ra + SE(N6)) \& \sim 1]$; $PC \leftarrow PC + 2$;
 - $\sim 1 = 0xFFFF$
 - $\&$ és l'operació AND bit a bit
 - El bit de menys pes del resultat de la suma es posa a 0 perquè, com accedim a word, l'adreça ha de ser un nombre parell
- *Store Word*: **ST** N6(Ra) , Rb
 - $MEM_w[(Ra + SE(N6)) \& \sim 1] \leftarrow Rb$; $PC \leftarrow PC + 2$;

Lògica de control: paraula de control

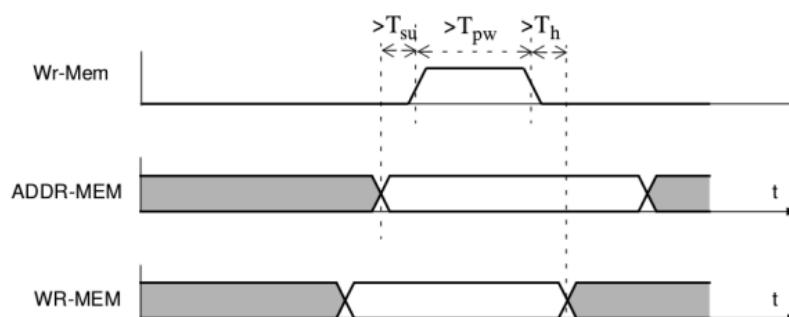


- Passarà a tenir 47 bits
 - Afegim senyals Wr-Mem i Byte
 - Wr-Mem mai podrà valer x, valdrà "1" per ST*, "0" per a la resta
 - Byte valdrà "1" per STB i LDB, "0" per ST i LD, x per a la resta
 - Canviem senyal In/Alu (1 bit) per bus -/i/l/a (2 bits)
- | | | | | | | | | | | | | | | |
|----|----|------|----|---|---------|----|-----|--------|-------|--------|------|-------|----------|----------------|
| @A | @B | Rb/N | OP | F | -/i/l/a | @D | WrD | Wr-Out | Rd-In | Wr-Mem | Byte | TknBr | N (hexa) | ADDR-IO (hexa) |
| | | | | | | | | | | | | | | |
- Els nous senyals els generarà la ROM_CTRL_LOGIC a partir del codi d'operació de la instrucció en execució

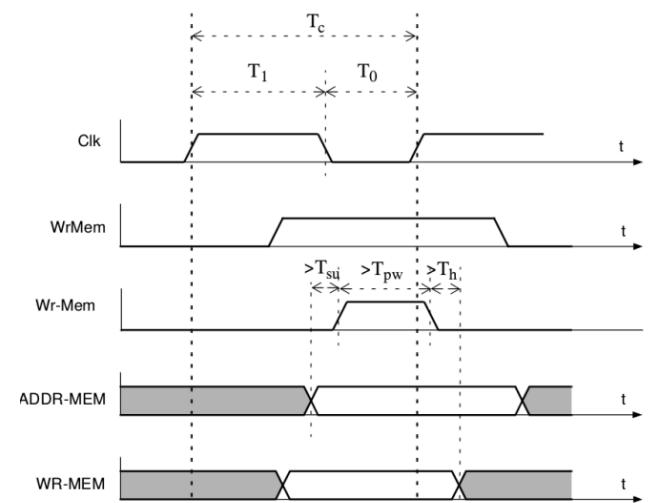
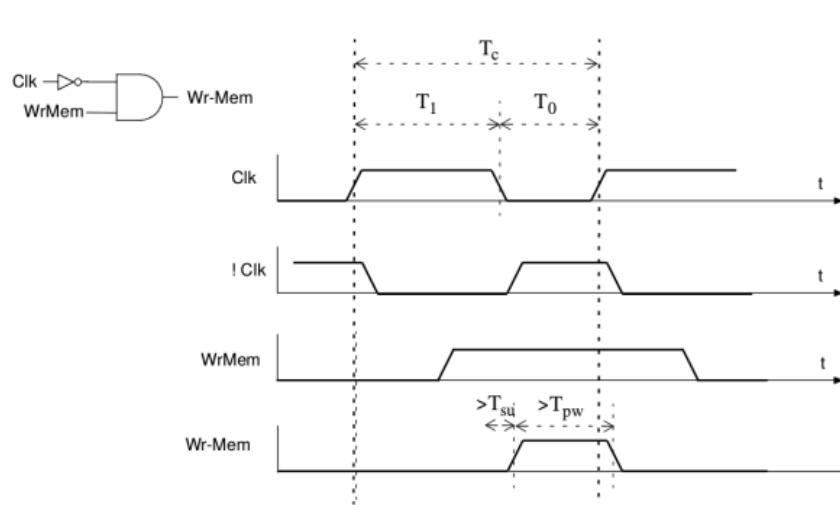


Senyal Wr-Mem

- Determina si aquest cicle es farà una escriptura a la RAM
 - Escriptura asíncrona (per nivell del senyal Wr-Mem)
- Cal garantir que quan Wr-Mem passa a "1", ADDR-MEM i WR-MEM ja siguin estables
- A més, el mòdul de memòria imposa que els senyals siguin estables un cert temps abans que Wr-Mem passi a "1", mentre val "1" i després que passi a "0"
 - T_{su} (setup), T_{pw} (pulse width), T_h (hold)
- Cronograma d'una escriptura correcta a memòria:



- Diferenciarem WrMem i Wr-Mem
 - WrMem: sortida de la ROM_CTRL_LOGIC
 - Wr-Mem: WrMem amb restriccions temporals
- Per implementar Wr-Mem modificarem la forma del senyal de rellotge
 - Ja no serà simètrica
 - Fins ara, el flanc descendent de rellotge ha estat irrelevante
 - Determinarem T_0 de forma que ja sigui segur posar Wr-Mem a "1"



T_c del computador Harvard unicicle

- Identificarem el camí crític del computador Harvard i el seu T_p
- Necessitem saber el T_p dels components bàsics:
 - $T_p(\text{NOT}) = 10 \text{ u.t.}$
 - $T_p(\text{And-2}) = T_p(\text{Or-2}) = 20 \text{ u.t.}$
 - $T_p(\text{FF}) = 100 \text{ u.t.}$
 - $T_p(\text{ROM_CTRL_LOGIC}) = 60 \text{ u.t.}$
 - $T_{acc}(\text{I-MEM}) = 800 \text{ u.t.}$
 - $T_{acc}(32\text{KB-RAM}) = 800 \text{ u.t.}$
 - Temps d'accés (lectura) a un mòdul de memòria RAM
- De totes les instruccions SISA, el camí crític ve donat per LDB
 - El camí comença al registre PC
 - Finalitza al REGFILE (registre destí del LDB)

- Com a mínim, T_c ha de ser 2.950 u.t.
 - L'arrodonim a 3.000 u.t.
- El senyal de rellotge pot ser simètric?
 - $T_1 = T_0 = 3.000/2 = 1.500$ u.t. ?
 - **No**, perquè ADDR-MEM triga 1.950 en estabilitzar-se
 - El senyal Wr-Mem passaria a "1" massa d'hora
- Alternatives:
 - $T_1 = T_0 = 2.000$ u.t. $\Rightarrow T_c = 4.000$ u.t.
 - $T_1 = \mathbf{2000}$ u.t., $T_0 = \mathbf{1.000}$ u.t. $\Rightarrow T_c = \mathbf{3.000}$ u.t.
- A una màquina unicicle
 - $T_{execució} = N_{instruccions} \cdot T_c$
- Calcularem el temps d'execució per a quatre versions d'un programa
- Especificació del programa:
 - Copiar un vector de 1.000 bytes emmagatzemat a la RAM
 - Origen: a partir de l'adreça 0x5000, a posicions consecutives
 - Destí: a partir de l'adreça 0x8000, a posicions consecutives

Instruccions lentes vs. ràpides

- De l'estudi del camí crític constatem diferències significatives al T_p de les instruccions de LM SISA
 - Instruccions "lentes":
 - Instruccions d'accés a memòria: **LD**, **LDB**, **ST** i **STB**
 - Era d'esperar perquè totes elles, després d'utilitzar l'ALU per sumar Ra i N6, accedeixen a la RAM
 - El T_p de totes elles és proper a 3.000 u.t.
 - Instruccions "ràpides":
 - Totes les altres
 - La més lenta de les ràpides és **CMPLE** amb $T_p=2.210$ u.t.
 - Si $T_c=3.000$ u.t., a l'executar cada instrucció ràpida al computador Harvard unicicle es malbarata un mínim de 790 u.t.
- Intentarem no malbaratar temps del processador quan executa instruccions "lentes"
 - Totes les instruccions no tindran el mateix temps d'execució

- Relotge amb "taquicàrdia"
 - T_c variable en funció del tipus d'instrucció
- **Processador multicicle**
 - Introduir un T_c de durada inferior al temps necessari per executar qualsevol instrucció ("minicicle")
 - L'execució de cada instrucció requerirà varis "minicicles"
 - Les instruccions "lentes" trigaran més "minicicles" que les "ràpides"
- Introduceix els "minicicles", de durada inferior al temps necessari per a executar qualsevol instrucció
 - Les instruccions necessitaran varis "minicicles" per a executar-se
 - Al nostre cas, els "minicicles" seran de 750 u.t.
 - Les instruccions ràpides trigaran 3 "minicicles" ($3 \times 750 = 2.250$ u.t.)
 - Les instruccions lentes trigaran 4 "minicicles" ($4 \times 750 = 3.000$ u.t.)
- La paraula de control es mantindrà invariable al llarg de tots els "minicicles" d'execució de cada instrucció
 - Llevat els senyals de modificació d'estat: WrD, Wr-Mem, Rd-In Wr-Out
 - Els registres no s'actualitzaran al final de cada "minicicle" sinó al final del darrer "minicicle" d'execució de la instrucció
 - REGFILE, PC
 - Haurem de tornar a estudiar com generar els senyals per autoritzar la modificació de memòria RAM i ports d'E/S
- Per implementar el processador Harvard multicicle només farem modificacions mínimes a la Lògica de Control i a la UPG

- Harvard unicicle:
 - Actualitza PC al final de cada cicle
 - Actualitza REGFILE al final dels cicles on WrD="1"
- Harvard multicicle:
 - **No pot actualitzar-los al final de cada "minicicle"**
 - Només al final del darrer "minicicle" d'execució de cada instrucció
 - Cal afegir senyal de càrrega al registre PC
 - LdPC: Nou senyal de sortida a la lògica de control
 - Els senyals WrD i LdPC només podran valer "1" el darrer minicicle d'execució de cada instrucció
 - La lògica de control haurà de ser conscient de s'està executant una instrucció lenta o ràpida i de en quin minicicle d'execució es troba

- Harvard unicicle:

- Utilitza un senyal de rellotge asimètric
- La segona fase indica quan és segur modificar memòria i *ports*
- Els senyals d'autorització de modificació de memòria i *ports* només poden valer "1" a aquesta segona fase del cicle

- Harvard multicicle:

- Donades les latències del mòdul de memòria i dels dispositius d'E/S, els senyals Wr-Mem, Rd-In i Wr-Out només cal que estiguin a "1" un temps inferior a la durada d'un minicicle
- Farem que aquests senyals només pugin valdre "1" el darrer minicicle d'execució de **ST**, **STB**, **IN** i **OUT**
- Mateixa solució que a l'actualització de registres

- Tindrà una part seqüencial i una altra combinacional

- Combinacional: SISC Harvard Multicycle CONTROL LOGIC

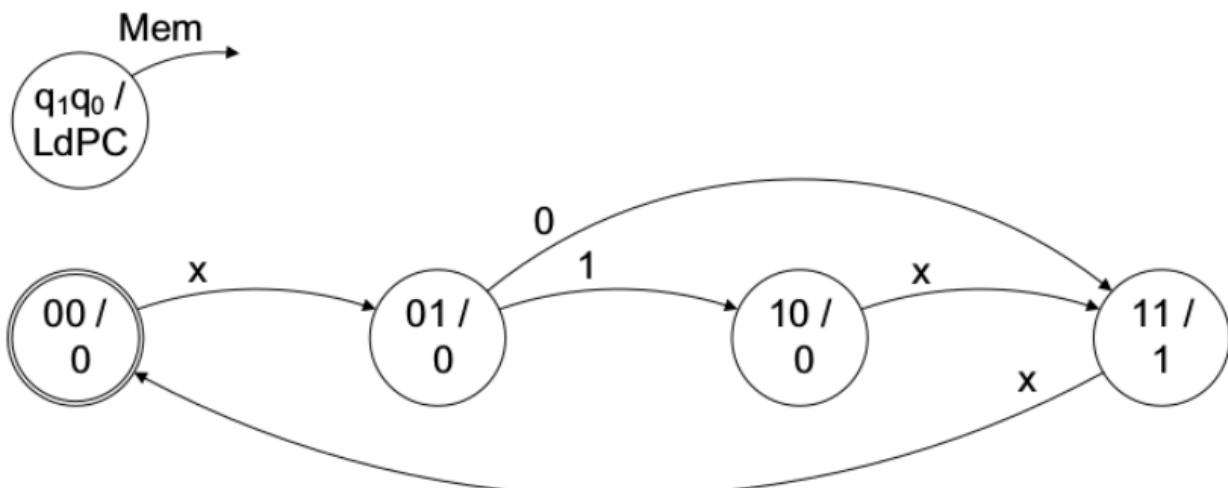
- La ROM generarà un nou senyal anomenat Mem
 - Valdrà "1" si la instrucció és lenta (**ST**, **STB**, **LD**, **LDB**); altrament "0"
 - El senyal WrD passa a dir-se WrD1
 - No farà la And-2 amb el senyal de rellotge negat

- Seqüencial: SISC Harvard Multicycle CONTROL

- Calcularà la sortida LdPC que indica quan actualitzar el registre PC
 - Valdrà "1" el darrer cicle d'execució de cada instrucció
 - LdPC també s'utilitzarà per restringir quan poden valer "1" els senals Wr-Mem, Rd-In i Wr-Out

- Graf d'estats del CLS que calcula LdPC

- Compta 3 o 4 minicicles en funció de si és una instrucció ràpida (Mem="0") o lenta (Mem="1")



- Computador Harvard unicicle
 - Cada instrucció triga 3.000 u.t.
 - $T_{execució} = N_{instruccions} \times 3.000 \text{ u.t.}$
- Computador Harvard multicicle
 - Les instruccions lentes triguen $4 \times 750 = 3.000 \text{ u.t.}$
 - Les instruccions ràpides triguen $3 \times 750 = 2.250 \text{ u.t.}$
 - $T_{execució} = N_{instruccions \ ràpides} \times 2.250 + N_{instruccions \ lentes} \times 3.000 \text{ u.t.}$
- Llevat que el programa només contingui instruccions lentes, el computador Harvard multicicle l'executarà en menys temps que el computador Harvard unicicle
- A qualsevol processador:
 - $T_{execució} = N_{instruccions} \times T_{mig \ per \ instrucció}$
- Computador Harvard unicicle
 - $T_{mig \ per \ instrucció} = 3.000 \text{ u.t./instrucció}$
- Computador Harvard multicicle
 - Dependrà de la proporció d'instruccions "lentes" i "ràpides"
 - Cal fer una **mitjana ponderada**
 - Exemple:
 - Temps mig per instrucció d'un programa que executa un 30% instruccions lentes i un 70% de ràpides?
 - $T_{mig \ per \ instrucció} = (0,3 \times 4 + 0,7 \times 3) \times 750 = 2.475 \text{ u.t/instrucció}$

Correspondència instruccions SISA i nodes



- Totes les instruccions passen pels nodes *Fetch* i *Decode*
- Nodes de càcul i actualització de l'estat

Instrucció SISA	Node(s) al graf
AND, OR, XOR, NOT, ADD, SUB, SHA, SHL	Al
CMPLT, CMPLE, CMPEQ, CMPLTU, CMPLEU	Cmp
ADDI	Addi
BZ	Bz
BNZ	Bnz
MOVI	Movi
MOVHI	Movhi
IN	In
OUT	Out
LD	Addr i Ld
LDB	Addr i Ldb
ST	Addr i St
STB	Addr i Stb

- Nous registres

- IR : *Instruction Register*

- Amb senyal de càrrega (LdIr)
- Guarda instrucció SISA

- RX, RY

- Operands llegits del REGFILE

- R@

- Adreça de memòria o nou PC

- Invisibles al programador

- Nous multiplexors

- Pc/Rx

- L'ALU calcularà el nou PC

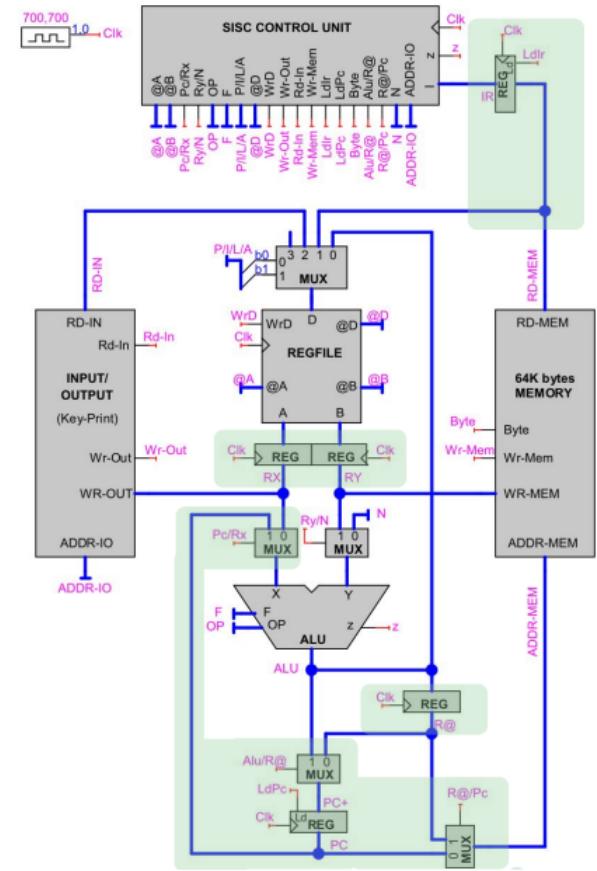
- R@/PC

- Memòria conté codi i dades

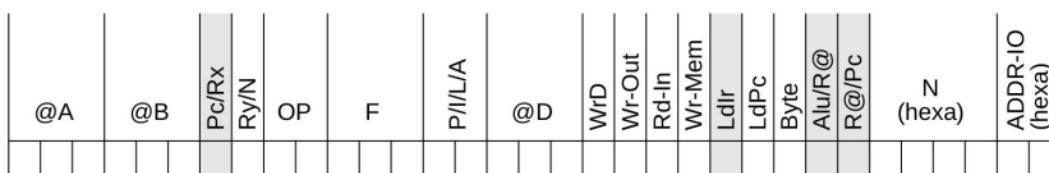
- Alu/R@

- Actualització del PC

- Elimina sumador i incrementador PC



- 51 bits



- LdIr: senyal de càrrega del registre IR
- Pc/Rx, Alu/R@, R@/Pc: senyals de control dels nous multiplexors
- LdPc: senyal de càrrega del registre PC
 - Eliminem el senyal TknBr

Paraula de control node F (*Fetch*)

- Fase comuna per a totes les instruccions

- Llegeix instrucció SISA de memòria i la carrega al registre IR

- La ALU actualitza registre PC amb PC+2

- Avancem feina perquè la ALU no faria res aquesta fase
- Si la instrucció és un BZ/BNZ que salta, tornarem a actualitzar el PC

Estado	Acciones	Palabra de control compactada
0 F	Búsqueda de la Instr.: IR \leftarrow Mem _w [PC] // Incremento del PC: PC \leftarrow PC + 2	R@/Pc=0, Byte=0, LdIr=1, Pc/Rx=1, N=0x0002, Ry/N=0, OP=00, F=100, Alu/R@=1, LdPc=1.

- Fase comuna per a totes les instruccions
- De forma especulativa...
 - La ALU calcula quin serà el valor del PC en cas que aquesta instrucció resulti ser **BZ** o **BNZ** i es compleixi la condició de salt
 - Es guarda al registre R0
 - Llegeix del REGFILE els valors dels que haurien de ser els registres font Ra i Rb i es carreguen als registres RX i RY

Estado	Acciones	Palabra de control compactada
1 D	Decodificación. Calculo @ salto tomado: $R@ \leftarrow PC + SE(N8)*2 //$ Lectura de registros. $(RX \leftarrow Ra) // (RY \leftarrow Rb)$	La decodificación no requiere ninguna acción en la UPG por lo que se usa la UPG en este ciclo para adelantar trabajo que pueda ser útil. Este cálculo solo será útil si la instrucción es BZ o BNZ. $N=SE(IR<7..0>)*2$, $Pc/Rx=1$, $Ry/N=0$, $OP=00$, $F=100$. Estas acciones se realizan sin tener que especificar nada en la palabra de control ya que RX y RY no tienen señal de permiso de escritura y @A y @B se generan directamente de los campos de bits del registro de instrucción IR<11..9> e IR<8..6>, respectivamente, en cada ciclo del grafo. Por ello las especificamos aquí entre paréntesis y ya no las especificaremos de ninguna forma en el resto de nodos.

- Fase de càlcul comuna per a totes les instruccions aritmèticò-lògiques
- La ALU fa el càlcul corresponent a la instrucció
- El resultat s'escriu al REGFILE

Estado	Acciones	Palabra de control compactada
2 Al	$Rd \leftarrow RX AL RY$	$Pc/Rx=0$, $Ry/N=1$, $OP=00$, $F=IR<2..0>$, $P/I/L/A=00$, $WrD=1$, $@D=IR<5..3>$.

- El node Cmp és anàleg a aquest
 - A la paraula de control només varia el valor del bus OP (valdrà 01)
- Fase de càlcul per a la instrucció **ADDI**
- La ALU fa la suma de RX amb SE(N6)
- El resultat s'escriu al REGFILE

Estado	Acciones	Palabra de control compactada
4 Addi	$Rd \leftarrow RX + SE(N6)$	$N=SE(IR<5..0>)$, $Pc/Rx=0$, $Ry/N=0$, $OP=00$, $F=100$, $P/I/L/A=00$, $WrD=1$, $@D=IR<8..6>$.

- Node comú per a les instruccions de memòria LD, LDB, ST, STB
- La ALU calcula la direcció de memòria a accedir RX + SE(N6)
 - El resultat es guarda al registre R@

Estado	Acciones	Palabra de control compactada
5	Addr $R@ \leftarrow RX + SE(N6)$	N=SE(IR<5..0>), Pc/Rx=0, Ry/N=0, OP=00, F=100.

- Com a *Decode*, al final del cicle RX i RY es carregaran amb Ra i Rb
- Un node per a cada instrucció
- Es determinen senyals de control Wr-Mem Byte i WrD
- Es fa l'accés a memòria (a l'adreça emmagatzemada al registre R@)
 - LD, LDB guarden la dada llegida al REGFILE
 - ST, STB guarden RY a memòria

Estado	Acciones	Palabra de control compactada
6	Ld $Rd \leftarrow Mem_w[R@]$	R@/Pc=1, Byte=0, P/I/L/A=01, WrD=1, @D=IR<8..6>.
7	St $Mem_w[R@] \leftarrow RY$	R@/Pc=1, Byte=0, Wr-Mem=1.
8	Ldb $Rd \leftarrow Mem_b[R@]$	R@/Pc=1, Byte=1, P/I/L/A=01, WrD=1, @D=IR<8..6>.
9	Stb $Mem_b[R@] \leftarrow RY<7..0>$	R@/Pc=1, Byte=1, Wr-Mem=1.

- La ALU deixa passar RX i actualitza el bit z
- Si $z=1$, es trenca el seqüenciament implícit
 - Cal posar LdPc=1
 - R@ (calculat a *Decode*) es carrega al registre PC

Estado	Acciones	Palabra de control compactada
11	Bz if (RX == 0) PC $\leftarrow R@$	Pc/Rx=0, OP=10, F=000, Alu/R@=0, LdPc=z.

- El node Bnz és anàleg a aquest
 - Només varia el valor del senyal LdPC (serà !z)

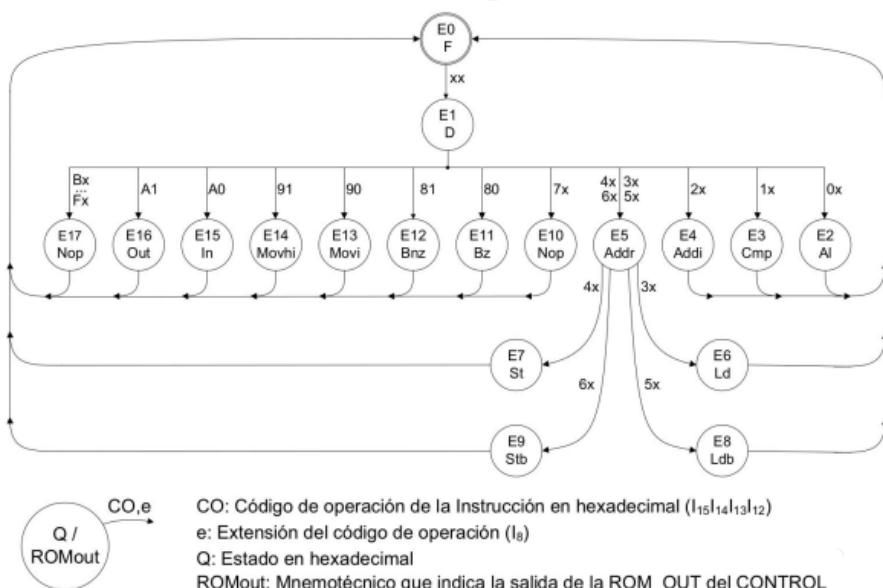
- Un node per a cada instrucció
- Varen en el senyals F i Pc/Rx
- Guarda el resultat al REGFILE

Estado		Acciones	Palabra de control compactada
13	Movi	Rd \leftarrow SE(N8)	N=SE(IR<7..0>, Ry/N=0, OP=10, F=001, P/I/L/A=00, WrD=1, @D=IR<11..9>.
14	Movhi	Rd \leftarrow (N*(2^8)) RX<7..0>	N=SE(IR<7..0>, Pc/Rx=0, Ry/N=0, OP=10, F=010, P/I/L/A=00, WrD=1, @D=IR<11..9>.

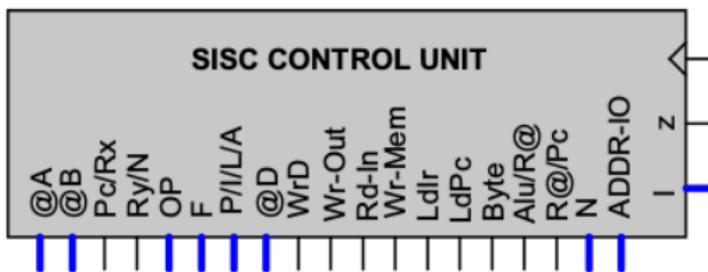
- Un node per a cada instrucció
 - IN guarda el contingut d'un port al REGFILE
 - OUT escriu RX sobre un port

Estado		Acciones	Palabra de control compactada
15	In	Rd \leftarrow Input[N8]	ADDR-IO=IR<7..0>, Rd-In=1, P/I/L/A=10, WrD=1, @D=IR<11..9>
16	Out	Output[N8] \leftarrow RX	ADDR-IO=IR<7..0>, Wr-Out=1.

- Implementarem la UC del computador Von Neumann amb un CLS
 - $n = 5$ bits d'entrada, $k = 5$ bits d'estat i $m = 24$ bits de sortida
 - ROM Q+: Determinació de l'estat següent de la UC



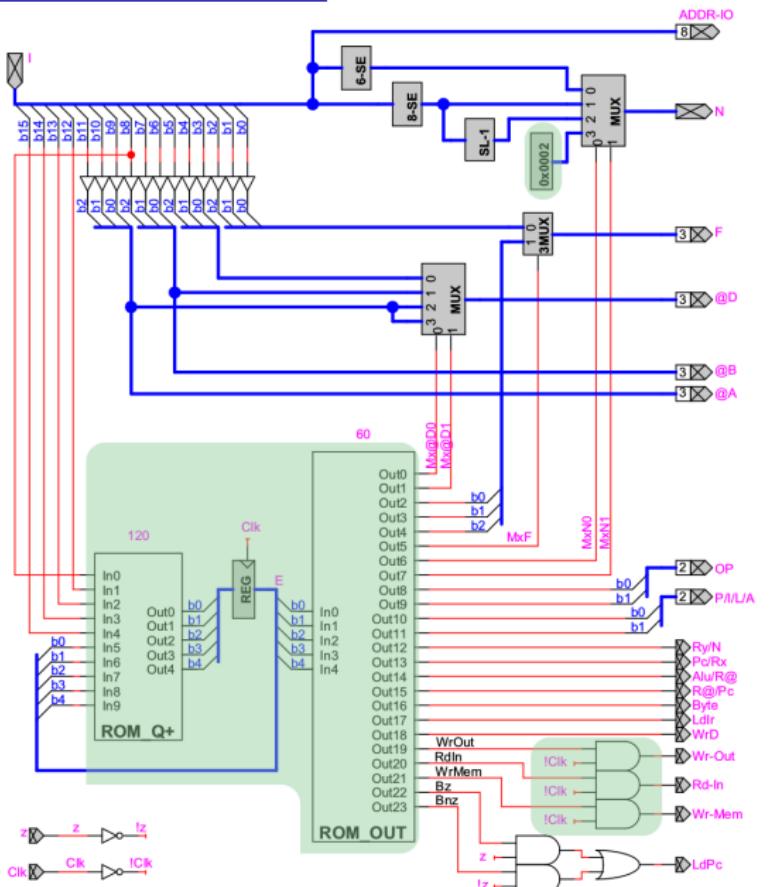
- ROM OUT: tindrà 24 bits de sortida
 - Combinant aquests 24 bits, els 16 de la instrucció (IR) i el bit z es generaran els 51 bits de la paraula de control pròpia de cada estat



- 17 bits d'entrada
 - Instrucció SISA (I) i bit z
- 51 bits de sortida
 - Addicions respecte a la del computador Harvard multicicle:
 - PC/Rx, Alu/R@, R@/PC: Senyals de control dels nous multiplexors
 - LdIr: Senyal de càrrega del registre IR
 - Canvis de nom:
 - Ry/N: abans era Rb/N
 - P/I/L/A: abans era -/I/L/A (acabarem utilitzant les 4 entrades)
 - LdPc: abans era TknBr

Esquema lògic SISC CONTROL UNIT

- CLS intern
 - $n = 5$ bits d'entrada
 - $k = 5$ bits d'estat
 - $m = 24$ bits de sortida
- Cal generar 0x0002
 - $PC \leftarrow PC + 2$
- Wr-Out, Rd-In, Wr-Mem
 - AND-2 amb $\neg Clk$
 - Com a Harvard unicicle
- Genera 51 bits a partir de:
 - 16 bits codificació SISA (I)
 - 24 bits ROM_OUT
 - bit z
 - senyal Clk



- Cal saber deduir el seu contingut a partir dels esquemes lògics
 - $2^k \text{ paraules} \times m \text{ bits/paraula} = 2^5 \text{ paraules} \times 24 \text{ bits/paraula} = 768 \text{ bits}$
 - Poseu x's on sigui possible

@ROM	Bnz	Bz	WrMem	Rdin	WrOut	WrD	LdIr	Byte	R@/Pc	Alu/R@	Pc/Rx	Ry/N	P/I/L/A1	P/I/L/A0	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@/D1
0	1 1	0 0 0	0	0 1	0 0 1	0 1	x	x x x x	0 1	1 0	x x	x x	0 0	1 1	1	1	1 0 0	x x					
1	0 0	0 0 0	0	0 0	x x x x	1 0	x x x x	0 0 1	x x x x	0 0 1	0 1	0 0 1	1 0 1	1 1 0 0	x x								
2	0 0	0 0 0	1	x x	x x x x	0 1	x x x x	0 1 0	0 0 0 0	0 0 0 0	x x x x	0 x x x	0 0 0 0	x x x x	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
3	0 0	0 0 0	1	x x	x x x x	0 1	x x x x	0 1 0	0 0 0 0	0 0 0 0	x x x x	0 x x x	0 0 0 1	x x x x	0 x x x	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
4	0 0	0 0 0	1	x x	x x x x	0 0	x x x x	0 0 0	0 0 0 0	0 0 0 0	x x x x	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 0	0 0 1	1 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
5	0 0	0 0 0	0	0 0	x x x x	0 0	x x x x	0 0 0	x x x x	0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 0	0 0 1	1 1 0 0	0 0 0 0	x x			
6	0 0	0 0 0	0	1 x	x x x x	0 1	x x x x	0 1 x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	0 1	
7	0 0	1 0 0	0	x 0	x x x x	0 1	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	
8	0 0	0 0 0	1	x x	1 1 x x x x	0 1	x x x x	0 1 x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	0 1	
9	0 0	1 0 0	0	x x	1 1 x x x x	0 1	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	
10	1 1	0 0 0	1	x x	x x x x	1 0 x	1 1 1 0	x x x x	1 1 1 0	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	1 0 1 1	0 1	1 0 1 1	0 1	1 0 1 1	0 1	
11	0 1	0 0 0	0	x x	x x x x	0 0	x x x x	1 0 x x	x x x x	1 0 x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	1 0 0 0	0 0	x x				
12	1 0	0 0 0	0	x x	x x x x	0 0	x x x x	1 0 x x	x x x x	1 0 x x	x x x x	x x x x	x x x x	x x x x	x x x x	x x x x	1 0 0 0	0 0	x x				
13	0 0	0 0 0	1	x x	x x x x	0 0	x x x x	0 0 0 0	1 0 x x	0 1 0 0	0 1 0 0	0 1 0 0	1 0 0 1	1 1 0 0	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	
14	0 0	0 0 0	1	x x	x x x x	0 0	x x x x	0 0 0 0	0 0 0 0	0 0 0 0	1 0 1 0	0 1 0 0	0 1 0 0	0 1 0 0	0 1 0 0	0 1 0 0	1 1 0 1	0 1 0	1 1 0 1	0 1 0	1 1 0 1	0 1 0	
15	0 0	0 1 0	1	x x	x x x x x x	1 0	x x x x x x	1 0 x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	1 0	
16	0 0	0 0 1	0	x x	x x x x x x	0 x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	
17..31	0 0	0 0 0	0	x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	x x x x x x	

ROM Q+: contingut estats Fetch i Decode

- Tres possibles representacions del contingut
 - Simbòlica, taula de veritat compacta, "Logic Works"
 - La codificació de l'estat de la UC (5 bits) ocupa dos díigits hexadecimals
- Paraules corresponents als estats Fetch i Decode

Q	I	Q ⁺	q ₄ q ₃ q ₂ q ₁ q ₀	I ₁₅ I ₁₄ I ₁₃ I ₁₂ I ₈	Q ⁺ (Hexa)	#veces	Q ⁺ (Hexa)
F	x	D	0 0 0 0 0	x x x x x	01	32	01
D	AL	Al	0 0 0 0 1	0 0 0 0 x	02	2	02
D	CMP	Cmp	0 0 0 0 1	0 0 0 1 x	03	2	03
D	ADDI	Addi	0 0 0 0 1	0 0 1 0 x	04	2	04
D	LD	Addr	0 0 0 0 1	0 0 1 1 x	05	2	05
D	ST	Addr	0 0 0 0 1	0 1 0 0 x	05	2	05
D	LDB	Addr	0 0 0 0 1	0 1 0 1 x	05	2	05
D	STB	Addr	0 0 0 0 1	0 1 1 0 x	05	2	05
D	JALR	Jalr	0 0 0 0 1	0 1 1 1 x	0A	2	0A
D	BZ	Bz	0 0 0 0 1	1 0 0 0 0	0B	1	0B
D	BNZ	Bnz	0 0 0 0 1	1 0 0 0 1	0C	1	0C
D	MOVI	Movi	0 0 0 0 1	1 0 0 1 0	0D	1	0D
D	MOVHI	Movhi	0 0 0 0 1	1 0 0 1 1	0E	1	0E
D	IN	In	0 0 0 0 1	1 0 1 0 0	0F	1	0F
D	OUT	Out	0 0 0 0 1	1 0 1 0 1	10	1	10
D	illegal	Nop	0 0 0 0 1	1 0 1 1 x	11	2	11
			0 0 0 0 1	1 1 x x x	11	8	11

- Paraules corresponents als estats de càlcul i modificació de l'estat

Q	I	Q⁺	q₄q₃q₂q₁q₀	I₁₅I₁₄I₁₃I₁₂I₈	Q⁺ (Hexa)	# veces	Q⁺ (Hexa)
Al	x	F	0 0 0 1 0	xxxxx	00	32	00
Cmp	x	F	0 0 0 1 1	xxxxx	00	32	00
Addi	x	F	0 0 1 0 0	xxxxx	00	32	00
Addr	! (LD+ST+ LDB+STB)	x	0 0 1 0 1	0 0 0 0 x	xx	2	00
Addr	LD	Ld	0 0 1 0 1	0 0 0 1 x	xx	2	00
Addr	ST	St	0 0 1 0 1	0 0 1 0 x	xx	2	00
Addr	LDB	Ldb	0 0 1 0 1	0 0 1 1 x	06	2	06
Addr	STB	Stb	0 0 1 0 1	0 1 0 0 x	07	2	07
Addr	! (LD+ST+ LDB+STB)	x	0 0 1 0 1	0 1 0 1 x	08	2	08
Ld	x	F	0 0 1 0 1	0 1 1 0 x	09	2	09
St	x	F	0 0 1 0 1	0 1 1 1 x	xx	2	00
Ldb	x	F	0 0 1 0 1	1 xxxx	xx	16	00
Stb	x	F	0 0 1 1 0	xxxxx	00	32	00
Jalr	x	F	0 0 1 1 1	xxxxx	00	32	00
Bz	x	F	0 1 0 0 0	xxxxx	00	32	00
Bnz	x	F	0 1 0 0 1	xxxxx	00	32	00
Movi	x	F	0 1 0 1 0	xxxxx	00	32	00
Movhi	x	F	0 1 0 1 1	xxxxx	00	32	00
In	x	F	0 1 1 0 0	xxxxx	00	32	00
Out	x	F	0 1 1 0 1	xxxxx	00	32	00
			0 1 1 1 0	xxxxx	00	32	00
			0 1 1 1 1	xxxxx	00	32	00
			1 0 0 0 0	xxxxx	00	32	00

Instrucció JALR



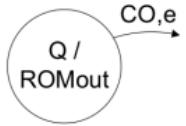
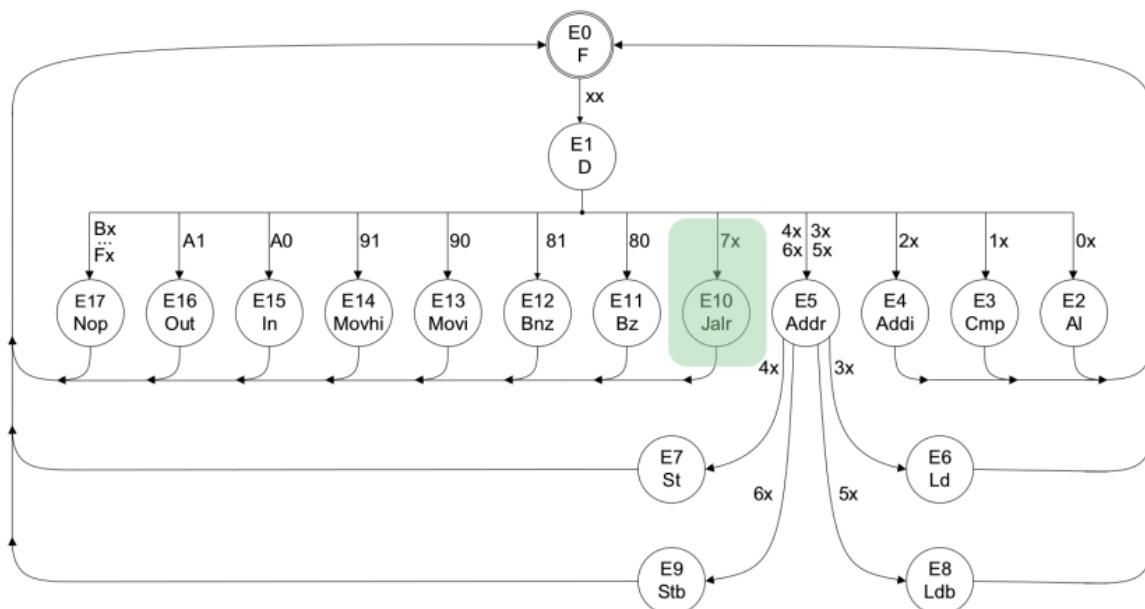
- Afegirem la darrera instrucció de LM al repetori SISA
- JALR** = *Jump Address and Link Register*
 - Imprescindible per a expressar en LM crides/retorns a routines
 - Guarda el valor actual del PC a un registre i assigna al PC un nou valor
 - Permet saltar incondicionalment a una adreça arbitrària
- Sintaxi assemblador:
 - JALR Rd , Ra**
 - Format 2-R
- Semàntica:
 - PC = PC + 2; tmp = Ra&(~1); Rd = PC; PC = tmp;
 - Variable tmp per si el registre font i el destí són el mateix
 - Amb Ra&(~1) força que el nou valor del PC sigui parell
- Codificació en llenguatge màquina:
 - 0111 aaa ddd xxxxxxxx
 - Els 6 bits baixos de la codificació són irrelevants

- Utilitzarà 3 nodes: *Fetch*, *Decode* i un de càclul
 - Jalr: carrega al PC el valor de Ra&(~ 1) i a Rd el valor de PC

Estado	Acciones	Palabra de control compactada
F	IR \leftarrow Memw[PC] // PC \leftarrow PC + 2	R@/Pc=0, Byte=0, LdIr=1, Pc/Rx=1, N=0x0002, Ry/N=0, F=100, OP=00, Alu/R@=1, LdPc=1.
D	R@ \leftarrow PC + SE(N8)*2 // (RX \leftarrow Ra) // (RY \leftarrow Rb)	N=SE(IR<7..0>)*2, Pc/Rx=1, Ry/N=0, F=100, OP=00.
Jalr	PC \leftarrow RX & (~ 1) // Rd \leftarrow PC	Pc/Rx=0, OP=10, F=011, Alu/R@=1, LdPc=1, P/I/L/A=11, WrD=1, @D=IR<8..6>.

- És la implementació que utilitzarem
- Paraula de control per a l'estat Jalr de la instrucció **JALR R7, R0**
 - Senyals imprescindibles (@A, @B i ADDR-IO són irrelevants en aquest estat, tot i que podríem saber-ne el valor)

@A	@B	Pc/Rx	Ry/N	OP	F	P/I/L/A	@D	WrD	Wr-Out	Rd-In	Wr-Mem	LdIr	LdPc	Byte	Alu/R@	R@/Pc	N (hexa)	ADDR-IO (hexa)
x x x x x x 0 x 1 0 0 1 1 1 1 1 1 1 x x x x x x x	x x x x x x 0 x 1 0 0 1 1 1 1 1 1 1 x x x x x x																	



CO: Código de operación de la Instrucción en hexadecimal ($I_{15}I_{14}I_{13}I_{12}$)

e: Extensión del código de operación (I_8)

Q: Estado en hexadecimal

ROMout: Mnemotécnico que indica la salida de la ROM_OUT del CONTROL

@ROM	Bnz	Bz	WrMem	RdIn	WrOut	WrD	LdIr	Byte	R@/Pc	Alu/R@	Pc/Rx	Ry/N	P/I/L/A1	P/I/L/A0	OP1	OP0	MxN1	MxN0	MxF	F2	F1	F0	Mx@D1	Mx@D0
10	1	1	0	0	0	1	x	x	x	1	0	x	1	1	1	0	x	x	1	0	1	1	0	1

- Com sempre actualitza el PC, tant Bz com Bnz han de valer "1"
 - Anàleg a l'estat de *Fetch*
- El bus P/I/L/A ha de valer "11" perquè és l'entrada que encamina el valor del PC a l'entrada del REGFILE
- La funció de la ALU és OP=10 i F=011 (calcular X&(~1))

Estudi del temps de cicle mínim

- Estudiarem el comportament dels estats potencialment més lents:
 - *Fetch*, *Decode*, *Ldb* i *Cmp* (en particular, el cas **CMPLE**)
- Els T_p dels blocs seran els mateixos que quan varem estudiar el computador Harvard
 - $T_p(\text{NOT}) = 10 \text{ u.t.}$
 - $T_p(\text{And-2}) = T_p(\text{Or-2}) = 20 \text{ u.t.}$
 - $T_p(\text{FF}) = 100 \text{ u.t.}$
 - $T_{acc}(32\text{KB-RAM}) = 800 \text{ u.t.}$
 - Temps d'accés (lectura) a un mòdul de memòria RAM
- Alguns casos:
 - $T_p(\text{ROM_Q+}) = 120 \text{ u.t.}$ i $T_p(\text{ROM_OUT}) = 60 \text{ u.t.}$
 - $T_p(\text{MUX-2-1-Sel}) = 50 \text{ u.t.}$ i $T_p(\text{MUX-2-1-Data}) = 40 \text{ u.t.}$
 - $T_p(\text{ALU}_{ADD}) = 860 \text{ u.t.}$
 - $T_p(\text{MEM}_{Ldb}) = 880 \text{ u.t.}$ i $T_p(\text{MEM}_{Ld}) = 840 \text{ u.t.}$

- Lectura instrucció:
 - $REG_Q \rightarrow R@/PC \rightarrow ADDR_MEM \rightarrow RDGMEM \rightarrow IR$
 - $T_p = 100 (\text{REG_Q}) + 60 (\text{ROM OUT}) + 50 (\text{mux } R@/\text{PC sel}) + 840 (\text{Memòria Ld}) + 40 (\text{dada registre càrrega IR}) = 1.090 \text{ u.t.}$
- $\text{PC} = \text{PC} + 2$
 - $REG_Q \rightarrow MxN \rightarrow N \rightarrow Ry/N \rightarrow ALU \rightarrow Alu/R@ \rightarrow PC$
 - $T_p = 100 (\text{REG_Q}) + 60 (\text{ROM OUT}) + 90 (\text{MUX}_{4-1} \text{ MxN selecció}) + 40 (\text{Mux Ry/N Dada}) + 860 (\text{ALU ADD}) + 40 (\text{MUX}_{2-1} \text{ Alu/R@ dada}) + 40 (\text{dada registre càrrega PC}) = 1.230 \text{ u.t.}$
- Conclusió, l'etapa de *Fetch* imposa que $T_c \geq 1.230 \text{ u.t.}$

T_p a l'estat de Decode

- Càlcul següent estat UC:
 - $REG_Q \rightarrow Q^+ \rightarrow REG_Q$
 - $T_p = 100 (\text{REG Q}) + 120 (\text{ROM Q}+) = 220 \text{ u.t.}$
- Lectura de registres
 - $IR \rightarrow REGFILE \rightarrow RX/RY$
 - $T_p = 100 (\text{IR}) + 130 (\text{REGFILE MUX}_{8-1} \text{ selecció}) = 230 \text{ u.t.}$
- Càlcul adreça destí del salt
 - $REG_Q \rightarrow MxN \rightarrow N \rightarrow Ry/N \rightarrow ALU \rightarrow R@$
 - $T_p = 100 (\text{REG_Q}) + 60 (\text{ROMOUT}) + 90 (\text{MUX}_{4-1} \text{ MxN selecció}) + 40 (\text{Mux Ry/N Dada}) + 860 (\text{ALU ADD}) = 1.150 \text{ u.t.}$
- Conclusions:
 - L'estat *Decode* imposa que $T_c \geq 1.150 \text{ u.t.}$
 - És menys restrictiu que *Fetch*

Temps mig per instrucció

- Sigui r el ratio d'instruccions lentes a l'execució d'un programa

$$\bullet \quad r = \frac{N_{\text{instruccions lentes}}}{N_{\text{instruccions lentes}} + N_{\text{instruccions ràpides}}} \quad 0 \leq r \leq 1$$

Computador	Temps mig per instrucció
Hardvard unicicle	3.000 u.t./inst.
Harvard multicicle	$(3 + r) \cdot 750 \text{ u.t./inst.}$
Von Neumann	$(3 + r) \cdot 1.400 \text{ u.t./inst.}$

- Observacions:

- Von Neumann és el més lent
- Harvard multicicle és el més ràpid
 - Empatat amb el Hardvard unicicle en el cas extrem $r=1$

- Ldb: Accés a memòria i escriptura a REGFILE:
 - $REG_Q \rightarrow R@/PC \rightarrow ADDR_MEM \rightarrow RD_MEM \rightarrow P/I/L/A \rightarrow REGFILE$
 - $T_p = 100 (\text{REG_Q}) + 60 (\text{ROMOUT}) + 50 (\text{MUX}_{2-1} \text{ R@/PC selecció}) + 880 (\text{RAM Ldb}) + 80 (\text{MUX}_{4-1} \text{ P/I/L/A}) + 40 (\text{dada registre càrrega Rd}) = 1.210 \text{ u.t.}$
- Cmp (instrucció CMPLE)
 - $REG_Q \rightarrow P_C/Rx, Ry/N \rightarrow X, Y \rightarrow ALU \rightarrow P/I/L/A \rightarrow REGFILE$
 - $T_p = 100 (\text{REG_Q}) + 60 (\text{ROMOUT}) + 50 (\text{PC/Rx, Ry/N MUX}_{2-1} \text{ selecció}) + 1.020 (\text{ALU CMPLE}) + 80 (\text{MUX}_{4-1} \text{ P/I/L/A dada}) + 40 (\text{dada registre càrrega Rd}) = 1.350 \text{ u.t.}$
- Conclusions:
 - L'estat més restrictiu és Cmp (cas CMPLE)
 - Cmp imposa $T_c \geq 1.350 \text{ u.t.}$
 - Arrodonim i determinem $T_c = 1.400 \text{ u.t.}$

Comparació de rendiments: temps



- Quant **trigarà més** (en %) el Von Neumann que el Harvard multicicle
 - Regla de 3 amb el temps mig per instrucció

$750 \cdot (3+r) \text{ u.t./inst.}$	-	100
$1.400 \cdot (3+r) \text{ u.t./inst.}$	-	x
 - $x = 100 \cdot (1.400 \cdot (3+r)) / (750 \cdot (3+r)) = 100 \cdot 1.400 / 750 = 186,67$
 \Rightarrow triga un 86,67% més
- Quant **trigarà més** (en %) el Von Neumann que Harvard unicicle?
 - Regla de 3 amb el temps mig per instrucció

3.000 u.t./inst.	-	100
$1.440 \cdot (3+r) \text{ u.t./inst.}$	-	x
 - $x = 100 \cdot (1.440 \cdot (3+r)) / 3.000 = 140 \cdot (3+r) / 3$
 - Si $r=0 \Rightarrow x=140 \Rightarrow$ Triga un 40% més
 - Si $r=0,2 \Rightarrow x=149,33 \Rightarrow$ Triga un 49,33% més
 - Si $r=1 \Rightarrow x=186,67 \Rightarrow$ Triga un 86,67% més

- A un codi font *assembler* trobarem instruccions i dades
 - Les instruccions s'agrupen en una o vàries seccions de codi
 - Contenen les instruccions SISA
 - Comencen amb la directiva .text
 - Les dades s'agrupen en una o vàries seccions de dades
 - Contenen la reserva d'espai de memòria i la seva inicialització
 - Comencen amb la directiva .data
- El codi font *assembler* conté una seqüència de seccions
 - L'inici d'una secció comporta la finalització de la secció anterior
- La directiva .end indica el final de la darrera secció i del codi font
 - El contingut posterior del fitxer font és ignorat

Etiquetes

- És una cadena alfanumèrica seguida del caràcter :
- Poden aparèixer tant a seccions de codi com de dades
- Permet identificar de forma simbòlica una adreça de memòria
 - Podrem fer referència a adreces de memòria que encara no són conegudes
- Exemple:

```
LD    R1 , 0(R3)
BZ    R1 , et1      ; Saltem a etiqueta et1
ADD   R2 , R0 , R1
ADD   R3 , R1 , R2
et1: AND  R1 , R2 , R3 ; et1 representa adreça de la instr
```

- L'*assembler* calcularà el desplaçament corresponent al BZ
 - En aquest cas, +2
- El programador es despreocuparà de fer aquest càcul
- Codi font més lleible i fàcil de mantenir
 - Si modifiquem el programa i augmenta la distància entre el BZ i el destí, l'*assembler* recalcularà el desplaçament

- El llenguatge *assembler* ofereix directives per a dimensionar i inicialitzar les seccions de dades
 - **.space size, fill**
Reserva *size bytes* consecutius i els inicialitza amb el *byte fill*. El paràmetre *fill* és opcional; si no hi és, s'inicialitzarà amb el *byte 0*
 - **.byte fill-1, fill-2, ..., fill-n**
Reserva i inicialitza *n bytes* consecutius amb els valors *fill-1, fill-2, ..., fill-n*
 - **.word fill-1, fill-2, ..., fill-n**
Reserva i inicialitza *n words* consecutius amb els valors *fill-1, fill-2, ..., fill-n* (*byte de menys pes a l'adreça parell*)
 - **.even**
Si volem tenir la garantia que la següent sentència/directiva del codi font *assembler* s'ubiqui a una adreça parell, aquesta directiva insereix, si cal, una directiva **.byte**

- Exemple (assumint **.data** comença a adreça parell):

```
.data
v: .space 15, 20          ; 15 bytes inicialitzats a 20
    .even                 ; Inserta un byte
w: .word 0x178A
    .byte 12, 0xFA, -1   ; 3 bytes init. a 12, 0xFA i -1
    .even                 ; Inserta un byte
z: .word 0xABCD
```

- El primer **.even** insereix un *byte* per garantir que **.word** estigui ben alineat a una adreça parell
- El segon **.even** també perquè hem declarat 3 *bytes* després del **word**
- Si les dades es carreguem a partir de 0x3000 llavors *v*=0x3000, *w*=0x3010, *z*=0x3016
 - Mateix resultat si la primera directiva fos **.space 16, 20**

- Dues possibles sintaxis:

- nom_constant = valor
- .set nom_constant, valor

- Exemple:

```
Mida = 100
.data
vector: .space Mida ; Reserva Mida bytes
```

- Associa al símbol Mida el valor 100
- A partir d'aquest moment, l'*assembler* substituirà totes les aparicions del símbol Mida pel valor 100
 - La definició d'una constant no ocuparà espai a memòria
 - Seria equivalent a un "Buscar i reemplaçar totes" al codi font o a un #define de C/C++

- Avantatges:

- Codi més lleible
- Facilita el manteniment de codi
 - Si el programador canvia el valor de la constant, l'*assembler* propagarà el canvi a tots els llocs on es referencia

- Permeten obtenir la part alta/baixa d'una dada de 16 bits
 - Siguin adreces de memòria, etiquetes, o constants numèriques
- Estalvia al programador haver de fer el càlcul
- Exemple:

- Assumim que la secció .data es carrega a partir de 0xCAFE

```
N = 24225 ; 24225 = 0x5EA1
.data
    .space 2 ; 0xCAFE
vector: .space 100, 0xFF ; 0xCB00 (0xCAFE+2)

.text
    MOVI R0, lo(N) ; lo(N) = 0xA1
    MOVHI R0, hi(N) ; hi(N) = 0x5E

    MOVI R1, lo(vector) ; lo(vector) = 0x00
    MOVHI R1, hi(vector) ; hi(vector) = 0xCB
```