

PRÀCTIQUES DE L'ASSIGNATURA

INTRODUCCIÓ ALS ORDINADORS



UNIVERSITAT DE BARCELONA



Índice de contenido

PRÀCTIQUES DE L'ASSIGNATURA.....	1
INTRODUCCIÓ ALS ORDINADORS.....	1
1. Normes de les Pràctiques.....	3
Avaluació.....	3
Sessions Pràctiques.....	3
Normativa.....	3
INTRODUCCIÓ TEÒRICA. ESTRUCTURA D'UN ORDINADOR.....	4
DIAGRAMA DE BLOCS D'UN ORDINADOR.....	4
ESTRUCTURA BÀSICA DE LA CPU.....	6
Pràctica 1: Introducció a la Màquina Ripes.....	8
Objectius.....	8
Introducció Teòrica.....	8
El programa el guardem seguint les indicacions del simulador.....	10
Instruccions.....	10
Exercicis guiats.....	12
Realització Pràctica Guiada.....	13
Informe.....	17
Pràctica 2: Exercicis amb el simulador Ripes.....	18
Objectius.....	18
Exercici 1 (Guiat).....	18
Exercici 2 (Guiat).....	19
Exercici 3 (a fer per l'alumne).....	19
Informe.....	20
Pràctica 3: Operacions i bucles amb RISC-V.....	21
Objectius.....	21
Estructura de Ripes V.....	21
Instruccions de salt.....	22
Problema 1.....	23
Problema 2.....	24
Problema 3.....	25
Informe.....	26
Exercici 1.....	26
Exercici 2.....	26
Exercici 3.....	26
Pràctica 4: Microinstruccions en Ripes.....	27
Objectiu.....	27
Introducció.....	27
Exercici guiat.....	28
Realització de la pràctica.....	29
Informe.....	29

1. Normes de les Pràctiques

Avaluació

Les pràctiques d'Introducció als Ordinadors són una part integrant de l'assignatura. Aquestes pràctiques es realitzen a l'aula IE de la facultat de Matemàtiques i comporten la realització de 10 pràctiques avaluable, un miniprojecte opcional i un examen pràctic final. El resultat de l'avaluació d'aquestes pràctiques constitueix el 50% de la nota de l'assignatura. Per poder aprovar l'assignatura la nota mitjana de pràctiques ha de ser superior o igual a 5 i caldrà presentar-se a totes les sessions i entregar tota la documentació requerida.

Sessions Pràctiques

L'assignatura consta de un total de 10 pràctiques dividides en 10 sessions i un miniprojecte com a treball a realitzar per l'alumne a casa. S'establiran períodes d'entrega a través del Campus Virtual de l'assignatura. Si l'entrega no es realitza en el període establert, la pràctica constarà com suspesa.

Normativa

–Els alumnes treballaran a l'aula individualment (si hi ha ordinadors suficients). En cas de que es disposi d'un ordinador portàtil propi, pot portar-se a l'aula per a la realització de les pràctiques si el alumne així ho vol.

–Les pràctiques es realitzaran utilitzant el sistema operatiu Linux. La contrasenya d'entrada és l'assignada per accedir als ordinadors de la facultat

–Queda totalment prohibit instal·lar o utilitzar programes propis als ordinadors de l'aula

–No està permesa la utilització dels ordinadors per realitzar treballs d'altres assignatures durant les pràctiques.

INTRODUCCIÓ TEÒRICA. ESTRUCTURA D'UN ORDINADOR

DIAGRAMA DE BLOCS D'UN ORDINADOR

Un ordinador és un dispositiu electrònic basat en un processador, un sistema de memòria, i uns dispositius d'entrada sortida que permeten la interacció amb l'usuari. Aquesta màquina processa les dades a partir de un grup d'instruccions, el programa. És, en definitiva, una dualitat entre hardware (part física) i software (part lògica) que interactuen entre si per tal d'assolir una funcionalitat determinada.

L'ordinador segueix una arquitectura comú. És a dir, té uns determinats atributs que són visibles al programador i que li permeten assolir unes determinades tasques. En funció de l'arquitectura tindrem un conjunt d'instruccions, el nombre de bits per realitzar una determinada operació, les tècniques d'adreçar les dades a memòria, etc. La forma en com s'implementa això internament és el que es coneix com organització de l'ordinador.

- **Arquitectura:** consisteix en aquells atributs que són visibles al programador
 - Conjunt d'instruccions
 - Nombre de bits usat per representació de dades
 - Mecanismes Entrada/Sortida
 - Tècniques d'adreça
 - P.ex.: Hi ha instrucció de multiplicar?
- **Organització:** consisteix en com s'han implementat, típicament amagats al programador
 - Senyals de control, interfícies, tecnologia de memòria
 - P.Ex.: Hi ha unitat de multiplicació per HW o es fa per addició repetida (algorisme)?

L'estructura típica d'un processador és la proposada per Von Neumann. Els ordinadors que implementen aquesta arquitectura consten de cinc parts: La unitat aritmètic-lògica o ALU, la unitat de control, la memòria, els dispositius d'entrada-sortida i els busos de interconnexió, que proporcionen un medi de transport de les dades entre les diferents parts que constitueixen el computador. Un ordinador amb aquesta arquitectura realitza o emula els següents passos de manera seqüencial:

1. Accedeix a la primera instrucció del programa, carregant l'adreça en el comptador de programa (PC). S'habilita l'escriptura del registre d'instruccions i es permet la lectura de memòria. El contingut apuntat per PC, la instrucció a executar, es guarda en el registre d'instruccions.
2. Augmenta el registre contador de programa, per apuntar a la següent instrucció.
3. Descodifica la instrucció mitjançant la unitat de control. Aquesta s'encarrega de coordinar la resta de components de l'ordinador per realitzar les operacions necessàries per tal d'executar la instrucció
4. S'executa la instrucció.

La figura 1 mostra aquest tipus d'arquitectura. A la memòria principal tenim el programa i la memòria assignada a les dades associades a l'execució del programa.

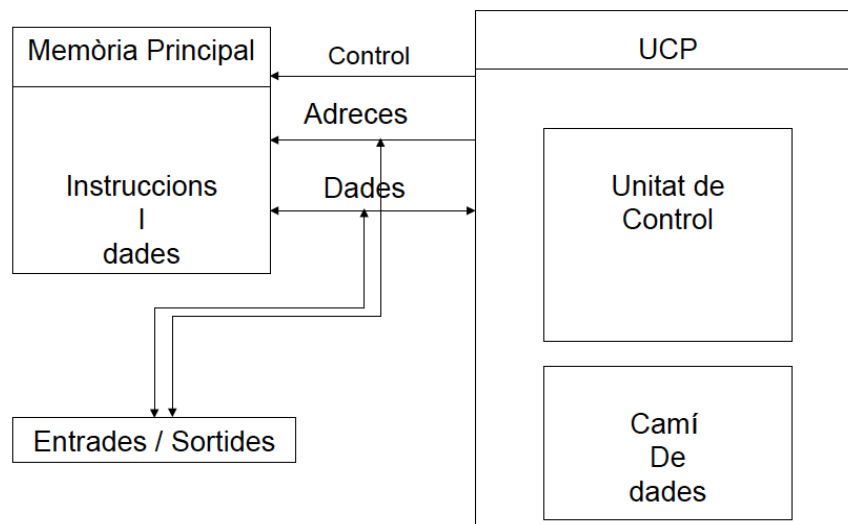


Figura 1. Estructura simplificada de l'arquitectura de Von Neumann

Tal i com s'observa a la figura 1, el bus d'adreces es unidireccional, ja que els generadors d'adreces apunten a la memòria principal i als dispositius d'entrada-sortida. A la CPU, no hi han adreces que adreçar per als altres dispositius. El bus de dades és bidireccional, per tal de permetre de llegir dades e instruccions per part de la CPU i escriure dades en la memòria o en els dispositius d'entrada-sortida. El bus de control ens serveix per identificar l'acció a executar. Per exemple, si el que volem és fer una escriptura a la memòria, des del bus de control seleccionarem la memòria i habilitarem el bit d'escriptura des del bus de control, a través del bus d'adreces indicarem la posició on volem guardar la dada i aquesta enviarà pel bus de dades.

Un exemple d'arquitectura von Neumann és la utilitzada per Intel en les seves famílies. Per exemple, a la figura 2 mostrem l'arquitectura del 8086.

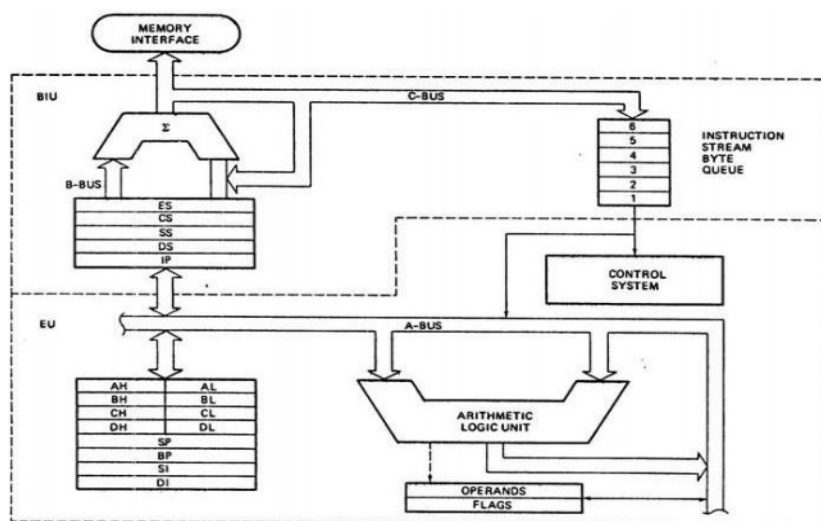


Figura 2. Arquitectura Intel 8086

Com s'ha comentat, aquesta arquitectura, tot i ser la més utilitzada, no és la única. Existeixen nombrosos processadors que implementen una arquitectura diferent, on el que tenim són dos memòries: una per dades i l'altra associada a la memòria de programa. Aquesta estructura es coneix com arquitectura Harvard. Originalment, el terme arquitectura Harvard feia referència a les architectures de computadors que utilitzaven dispositius emmagatzemament físicament separats per a les instruccions i les dades, en oposició a l'arquitectura de von Neumann. Aquest terme prové de la computadora Harvard Mark I, que emmagatzema les instruccions en cintes perforades i les dades en interruptors. La figura 3 mostra el diagrama de blocs simplificat de una arquitectura Harvard.

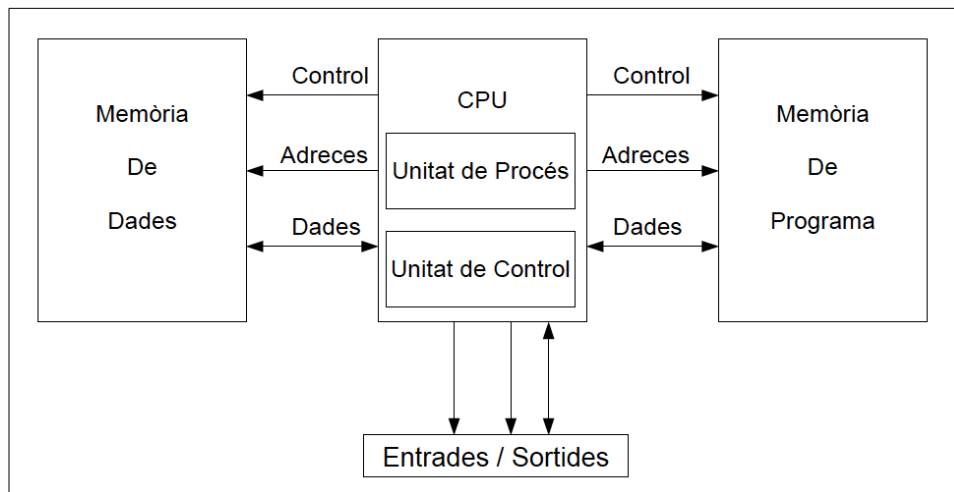


Figura 3. Diagrama de blocs simplificat de una arquitectura Harvard.

La figura 4 mostra un exemple de processador implementant aquest tipus d'arquitectura: El microprocessador de la casa Microchip de 8 bits 16F84A. La memòria de programa es una memòria Flash de múltiples lectures i escriptures, mentre que la memòria de dades és una memòria EEPROM. En aquest cas, les dues memòries s'integren en el mateix xip.

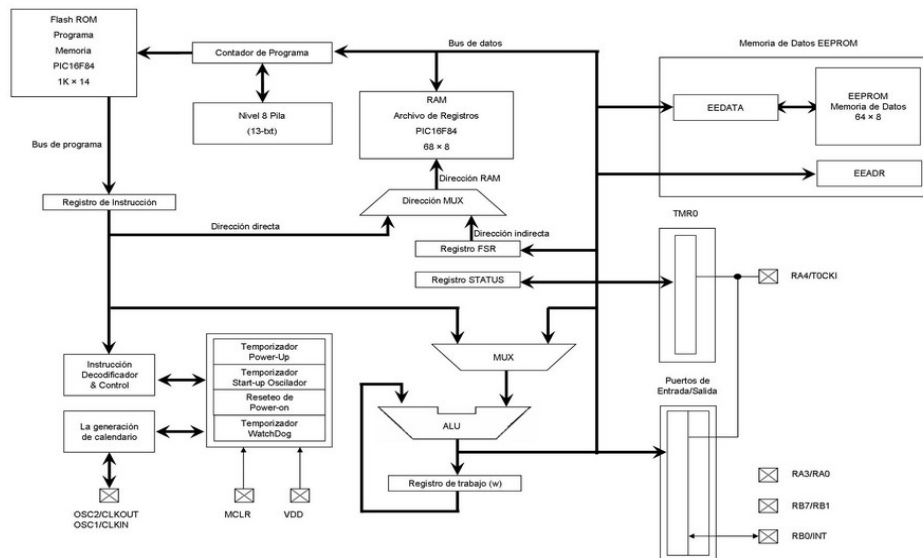


Figura 4. Diagrama de blocs arquitectura Harvard.

ESTRUCTURA BÀSICA DE LA CPU

Un processador consta dels següents blocs:

1. El camí de dades, que inclou, la Unitat Aritmètica Lògica (ALU), els registres de propòsit general, el registre acumulador, normalment connectat a la sortida de la ALU i les línies de interconnexió
2. La unitat de control, que s'encarrega de gestionar tot el funcionament del processador així com l'accés a la memòria principal
3. El conjunt de registres de propòsit específic, que es troben majoritàriament al camí de dades, però que poden estar associats a les altres unitats
4. Els busos d'interconnexió, entre els que tenim el bus de dades, el bus d'adreces i el bus de control
5. Els blocs de memòria cache, on tindrem les instruccions que s'han d'executar i les dades més importants.

El propòsit d'aquestes pràctiques inicials és justament entendre el funcionament de la CPU i la seva connexió amb la memòria principal. Per assolir aquest objectiu treballarem amb el simulador Ripes_RiscV. Per la mateixa arquitectura, Ripes ens permet treballar amb diferents versions, cada una amb diferents compromisos entre velocitat, complexitat, cost i forma de treballar. En particular treballarem amb la forma més bàsica associada a aquesta arquitectura, un Processador de un sol cicle (**Single cycle processor**) i un **processador multicicle**. Tots dos tenen instruccions de mida de 32 bits ja que tenen la mateixa arquitectura. En canvi, l'estructura de la CPU (microarquitectura) és lleugerament diferent. Tots dos tenen una estructura Harvard, amb la memòria de dades per una banda i la memòria de programa per l'altre. **Ripes Single Cycle Processor (RSCP)** presenta una microarquitectura simple, que permet executar instruccions en un sol cicle de rellotge. Això fa que el temps de cicle del rellotge hagi de tenir present el pitjor cas, és a dir, que ajustarem el temps a aquella instrucció que més triga en executar-se. Per altra banda el mateix simulador ens permet treballar amb un Ripes amb instruccions multicicle i amb execució simultània de varies instruccions, el que es coneix com pipeline, **Ripes 5-Stage Processor (R5SP)**. Això vol dir que mentre estàs executant una instrucció, pots anar carregant una altra instrucció i pots anar decodificant una altra de manera simultània, ja que aquests processos fan servir diferents recursos. Això ens permetrà analitzar i veure en la pràctica conceptes com el 'Pipeline' o execució pseudo-paral·lela de les instruccions. Per entendre el funcionament posarem l'exemple d'una bugaderia.

RSCP treballa de forma totalment seqüencial. La roba (el programa) està a la cistella inicial. Agafo la primera peça de roba i la fico a la rentadora, un cop acabada, fico la peça a l'assecadora, verifico que està neta i seca i la guardo en una segona cistella. Un cop acabat agafo la segona peça i la fico a la rentadora, després passo a l'assecadora, torno a verificar que estigui neta i seca i la guardo a la segona cistella... Aquest és el mode de treball sense pipeline. A més, això ho faig de una sola tirada, sense parar, a la velocitat que em limiti el pitjor cas (els llençols per exemple).



Figura 5. Exemple de treball de un processador sense pipeline ni execució multicicle simulat com una bugaderia.

R5SP treballa seguint una execució multicicle i amb pipeline. Seguint amb l'exemple, primer agafa la primera peça de roba bruta i la fica en la rentadora, un cop acabada aquesta, fico la peça a l'assecadora, i com que la rentadora està buida, fico la segona peça de roba. Un cop acabat el procés de rentar i assecat, inspecciono visualment la peça rentada i secada, mentrestant, fico la segona peça ja rentada a l'assecadora i introdueixo una tercera peça a la rentadora... Aquest procés, implica fer servir més recursos del sistema en el mateix interval de temps, minimitzant el temps d'execució i millorant les prestacions de l'ordinador. Avui dia tots els ordinadors treballen d'aquesta manera.

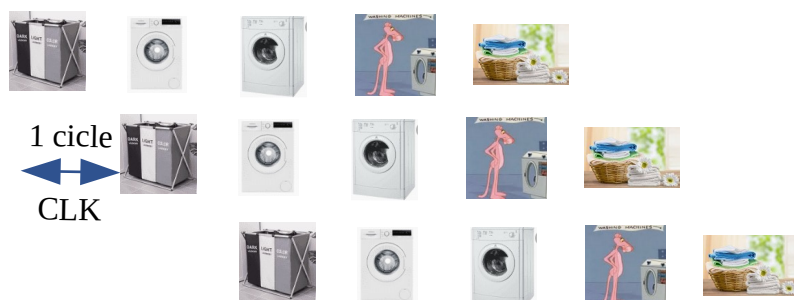


Figura 6. Exemple de treball de un processador amb execució multicicle i amb pipeline simulat com una bugaderia que treballa optimitzant els seus recursos.

Pràctica 1: Introducció a la Màquina Ripes

(1 sessió)

Objectius

Familiaritzar-se amb el simulador Ripes en les seves dues versions amb que treballarem: RSCP i R5SP.

Entendre què fan les instruccions

Comprendre l'analogia entre llenguatge màquina i el llenguatge ensamblador

Introducció Teòrica

Ripes és un simulador que integra diverses microarquitectures de menor a major dificultat i que permet aprendre els conceptes bàsics sobre estructura i arquitectura de processadors. L'arquitectura del processador es tan senzilla com elegant, amb 32 registres de propòsit general, un que sempre val zero i on no es permet l'escriptura de cap valor (x0) i la resta dels registres on es poden fer lectures, escriptures i desplaçaments. Com a registres específics tenim (i no tenim):

1. Punters a memòria: Stack Pointer (SP), correspon amb el registre x2, Global Pointer (GP) que correspon amb el registre x3 i el Thread Pointer (TP) que correspon amb el registre x3.
2. No hi ha registre d'estat
3. No hi ha registre d'instruccions (IR)
4. El registre comptador de programa (PC) per adreçar la memòria
5. 7 registres temporals t0 – t6 i 8 registres que es fan servir com argument de funció a0 – a7
6. 12 registres per guardar dades globals s0 – s11
7. Un registre específic de retorn de funció ra, que correspon al x1

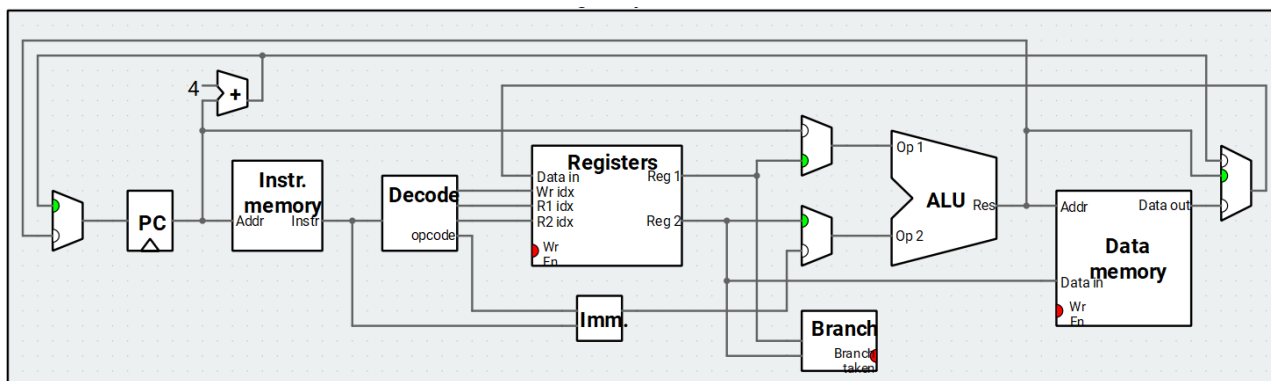


Figura 1. Diagrama del simulador Ripes RSCP

Tenim a més un banc de memòria on es carrega el programa a executar (Instruction Memory) i un banc per guardar les dades (Data Memory), una sèrie de multiplexors per seleccionar els registres, entrada en ALU i sortida cap a memòria i una unitat de control que, tot i no veure's implícitament està associada al bus de control, que correspon a totes les línies que governen els diferents dispositius.

El professor de pràctiques us explicarà detalladament el funcionament d'aquest simulador.

A nivell de software tenim parts ben diferenciades: (i) les directives de programació, (ii) les etiquetes, (iii) els comentaris i, finalment (iv) el set d'instruccions.

i) **Directives** de programació: Les directives, també anomenades pseudo-operacions o pseudo-ops són instruccions que s'executen en temps d'ensamblat i no per la CPU en temps d'execució. Permeten fer l'ensamblat del programa depenent de paràmetres introduïts pel programador, de tal manera que el codi pugui ser ensamblat de diferents formes tenint en compte diferents aplicacions.

Indiquen per tant a l'ensamblador detalls necessaris per portar a bon terme el procés d'ensamblatge. Per exemple, una directiva pot indicar a l'ensamblador la posició de memòria on ha d'estar localitzat el programa. Tenim les següents directives

1. Directiva **.data** -> Serveix per assignar a una determinada posició de memòria un determinat contingut. Caldrà dir l'espai que assignem:

- .byte Reserva un byte de memòria. Exemple .byte 12
- .half Reserva 16 bits de memòria. Exemple .half 1234
- .word Reserva 32 bits de memòria. Exemple .word 456423
- .string Reserva una seqüència de caràcters: Exemple .string 'hola mon'

2.Directiva .text -> Indica on comença la memòria de programa.

ii)**Etiquetes**. Serveixen per apuntar de forma més amigable a una determinada posició del programa.

Exemple: **etiqueta**: .word 7 -> Indica que en l'adreça de memòria etiqueta tinc un 7 guardat en un espai de memòria de 32 bits.

iii)Comentaris. A diferència de altres llenguatges de programació com C o Java, en aquest simulador els comentaris s'indiquen amb un “#”

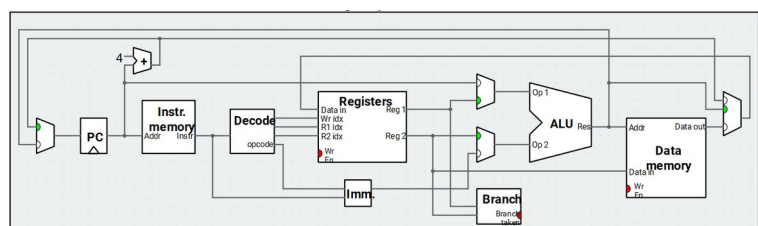
iv)El conjunt d'instruccions: Una instrucció en ensamblador és una representació simbòlica de codi màquina binari necessària per realitzar una sèrie de operacions per part de la CPU.

ADD a2,t0,a1 => 0b00000000110001011000011010110011 => t0 + a1 => a2

El conjunt d'instruccions d'un processador ens reflexa directament la seva arquitectura. La implementació d'un algorisme per tal de solucionar un problema implica la realització d'un programa que es realitzarà tenint en compte les instruccions que pot interpretar l'ensamblador. La majoria dels processadors tenen el mateix tipus de grups d'instruccions, tot i que no necessàriament han de tenir el mateix format ni el mateix nombre d'instruccions per a cada grup. De fet, cada arquitectura té el seu propi llenguatge màquina i en conseqüència el seu propi conjunt d'instruccions.

Exemple de tot plegat:

```
1 # Primer programa en ensamblador
2 # Autor: Alumne
3
4 # memòria de dades
5 .data
6 xx: .word 3
7 yy: .word 5
8 res: .word 0
9
10 # memòria d'instruccions
11 .text
12 la a0, xx
13 lw a1, 0(a0)
14 lw a2, 4(a0)
15 add a3, a1, a2
16 end: nop
17 j end
18
```



El programa el guardem seguint les indicacions del simulador.

Instruccions

Dividim el conjunt d'instruccions en els següents subgrups:

1. Instruccions aritmètic – lògiques

2. Desplaçaments

3. Instruccions d'accés a memòria

4. Instruccions de salt

Instruccions Aritmètic – lògiques

ADD Rd, Rf1, Rf2	Suma el contingut de Registre font 1 + el contingut de R font 2 i ho guarda en R destí
SUB Rd,Rf1,Rf2	Resta el contingut de R font 1 – el contingut de R font 2 i ho guarda en R destí
ADDI Rd, Rf, Imm	Suma el contingut de R font 1 + el nombre Immediat i guarda el resultat en R destí. L'immediat pot ser un valor positiu o negatiu.
AND Rd, Rf1, Rf2	Fa una AND dels continguts de Rf1 i Rf2 i ho guarda en Rd
XOR Rd, Rf1, Rf2	Fa una XOR dels continguts de Rf1 i Rf2 i ho guarda en Rd
OR Rd, Rf1, Rf2	Fa una OR dels continguts de Rf1 i Rf2 i ho guarda en Rd
ANDI Rd, Rf1, Imm	Fa una AND dels continguts de Rf1 i el nombre Immediat Imm i ho guarda en Rd
XORI Rd, Rf1, Imm	Fa una XOR dels continguts de Rf1 i el nombre Immediat Imm i ho guarda en Rd
ORI Rd, Rf1, Imm	Fa una OR dels continguts de Rf1 i el nombre Immediat Imm i ho guarda en Rd

Desplaçaments

SLL Rd,Rf1,Rf2	Fa un desplaçament a l'esquerra del contingut que hi ha a Rf1 en nombre de bits que hi ha a Rf2, guardant el resultat en Rd
SRL Rd,Rf1,Rf2	Fa un desplaçament a la dreta del contingut que hi ha a Rf1 en nombre de bits que hi ha a Rf2, guardant el resultat en Rd
SLLI Rd,Rf1,Imm	Fa un desplaçament a l'esquerra del contingut que hi ha a Rf1 en nombre de bits que indica el valor immediat Imm, guardant el resultat en Rd
SRLI Rd,Rf1,Imm	Fa un desplaçament a la dreta del contingut que hi ha a Rf1 en nombre de bits que indica el valor immediat Imm, guardant el resultat en Rd
SRA Rd,Rf1,Rf2	Fa un desplaçament aritmètic cap a la dreta del contingut que hi ha a Rf1 en nombre de bits que hi ha a Rf2, guardant el resultat en Rd
SRAI Rd,Rf1,Imm	Fa un desplaçament aritmètic a la dreta del contingut que hi ha a Rf1 en nombre de bits que indica el valor immediat Imm, guardant el resultat en Rd

Instruccions d'accés a memòria

on valor és un nombre o una etiqueta. Per exemple el cas

LOAD Rd, Imm(Rf)	Desa el contingut que hi ha a la posició de memòria Imm + contingut de Rf en el registre Rd. LA, pseudoinstrucció per assignar una adreça a un registre. Podem parlar de carregar un word LW , carregar 16 bits LH o carregar un byte LB
STORE Rf1, valor(Rf2)	Guarda el contingut de Rf1 en la posició de memòria Imm + contingut de Rf2. Igual que en el cas anterior les instruccions poden treballar amb 8 bits SB , 16 bits SH o 32 bits SW

Exemples:

```
1 # Primer programa en ensamblador
2 # Autor: Alumne
3
4 # memòria de dades
5 .data
6 xx: .word 3
7 yy: .word 5
8 res: .word 0
9
10 # memòria d'instruccions
11 .text
12 #permet guardar l'adreça associada a xx
13 #en el registre a0
14 la a0, xx
15 # observa com l'immediat augmenta de 4
16 # en 4...|
17 lw a1, 0(a0)
18 lw a2, 4(a0)
19 add a3, a1, a2
20 sll a3, a3, a1
21 sw a3, 8(a0)
22 end: nop
23 j end
```

la primera posició de memòria de dades és la 1000 0000h
aniran incrementant de 32b en 32b

Instruccions de salt

j posicioMemoria	salt incondicional a la posició de memòria posicioMemoria
BEQ Rf1, Rf2, posicioMemoria	si el contingut de Rf1 és igual al de Rf2 salta a la posició de memòria posicioMemoria
BEQZ Rf, posicioMemoria	salt a la posició de memòria posicioMemoria si Rf és igual a zero.
BGT Rf1, Rf2, posicioMemoria	si Rf1 > Rf2, salta a la posició de memòria posicioMemoria
BGE Rf1, Rf2, posicioMemoria	si Rf1 >= Rf2, salta a la posició de memòria posicioMemoria
BGTZ Rf1, posicioMemoria	si Rf1 > 0, salta a la posició de memòria posicioMemoria
BLT Rf1, Rf2, posicioMemoria	si Rf1 < Rf2, salta a la posició de memòria posicioMemoria
BLE Rf1, Rf2, posicioMemoria	si Rf1 <= Rf2, salta a la posició de memòria posicioMemoria
BLTZ Rf1, posicioMemoria	si Rf1 < 0, salta a la posició de memòria posicioMemoria
BLEZ Rf1, posicioMemoria	si Rf1 <= 0, salta a la posició de memòria posicioMemoria
BNE Rf1, Rf2, posicioMemoria	si Rf1 != Rf2 salta a la posició de memòria posicioMemoria

BNEZ Rf1, posicioMemoria

si Rf1!= 0 salta a la posició de memòria posicioMemoria

Exercicis guiats

1. Indiqueu quines de les següents instruccions són incorrectes segons les especificacions anteriorment indicades. Expliqueu el perquè en cada cas:

	Instruccions	Correcte/Incorrecte	Motiu
a	LW a1(R0), a1	incorrecte	Lw a1, 0(a1)
b	SW a1,a1(3)	incorrecte	Sw a1, 3(a1)
c	BNE 6(t1)	incorrecte	BNE t1, t2, etiqueta
d	ADDI t2, #11, 5(t3)	incorrecte	ADDI t2, t3, 11
e	SUB zero ,a2,a3	correcte	
f	LOAD 3(a0),a1	incorrecte	LOAD a1, 3(a0)

2. Supposeu que el simulador Ripes té els següents valors en alguns registres i posicions de memòria (recordeu que amb la lletra “h” s’indica que el número està en format hexadecimal):

-Registres: zero=0000h, a1=0002h, a2=A5E3h, a3=F412h, a4 = address1, a5 = address2

-Memòria: M[address1]=F45Ah i M[address2]=0033h, etiqueta => 10h

Indiqueu què fa cadascuna de les instruccions següents, cal explicitar totes les alteracions que es produeixen a la memòria, bits de condició, registres i valor final. Per cada una de les instruccions sempre partiu de les condicions inicials descrites més a dalt.

	Instruccions	x0	a1	a2	a3	M[address1]	M[address2]	PC
a	lw a1, 0(a4)	0	F45A	-----	-----	-----	-----	+4
b	sw a1,0(a5)	0	-----	-----	-----	-----	F45A	+4
c	j etiqueta	0	-----	-----	-----	-----	-----	et+4
d	addi a3,a2, 11	0	-----	A5E3	A5f4	-----	-----	+4
e	SUB zero,a2,a3	0	-----	-----	-----	-----	-----	+4
f	srli a1,a1,1		7A25	-----	-----	-----	-----	+4

Breu descripció del que fa cada instrucció:

a. guarda en a1 el contingut apuntat per [a4]+0

b.

c.

d.

e.

f.

3. **RISC V** té totes les instruccions de la mateixa mida, 32 bits, i les defineix segons el tipus:

Format R. Són les que tenen tres registres: add, sub, and, or...

Format I. Són les que tenen un immediat: addi, lw, jalr, slli

Format S. Instruccions de 'store': sw, sb, sh

Format SB. Instruccions de salt: BEQ, BGE,...

Format U.

Format UJ

Escriviu en llenguatge màquina el següent programa, feu-ho en hexadecimal i binari:

@	M[@]	LLENGUATGE MÀQUINA	LENG. MAQ. (hex)
.data			
xx:	.word 3		
yy:	.word 5		
oo:	.word 2		
zz:	.word 8		
.text			
00h	LA a0, xx		
08h	SUB a1,a1,a1		
0ch	ADD a2,zero,zero		
10h	Loop: ADDI a7,a1,-4		
14h	BEQZ a7,final		
18h	LW a3,0(a0)		
1ch	ADD a2,a2,a3		
20h	ADDI a0,a0,4		
24h	ADDI a1, a1, 1		
28h	j loop		
2ch	final: SW a2,4(a0)		

Realització Pràctica (entregable)

1. Escriure les instruccions de l'apartat 1 de l'estudi previ en un programa al simulador Ripes (no cal que implementi cap funcionalitat) i intenteu compilar-lo. Podreu comprovar com les instruccions incorrectes us donen error alhora de compilar. Feu les modificacions adequades per a que no us doni cap error en la compilació (recordeu que no es demana cap funcionalitat).

La forma de crear un programa amb el simulador Ripes és el següent:

1. Executar el programa Ripes.exe

2. Cliqueu en la icona



3. Escriure el programa en llenguatge Assemblador. Noteu que a la finestra de la dreta apareix el codi en assemblador pur, sense mnemotècnics.

4. Executeu el programa clicant



5. Alhora de desar el programa donar-li l'extensió *.asm

6. Podeu executar el programa veient com va executant el camí de dades. Cliqueu a



7. Seleccionem inicialment Single Cycle Processor

Un exemple de com escriure les instruccions anteriors en un programa és el següent:

```
.data
xx: .word 5
yy: .word 0
inici:
    la a0, xx
    lw a1, 0(a0)
    sw a1, 4(a0)
    addi a1, a1, -1
    beqz a1, inici
    sub zero, a2, a1
end:
    j end
```

2. Pel programa de l'apartat 3 de l'estudi guiat, les posicions de memòria que van des de la xxh a la zzh, ambdues incloses, contenen una seqüència del quatre números 3, 5, 2, 8, emmagatzemats de forma consecutiva.

Partint d'aquesta descripció responeu a les següents preguntes:

- a) En quina posició de memòria escriu aquest programa el resultat?
- b) Què fa el programa?
- c) Quina sentència de control de flux (en llenguatge d'alt nivell) implementa el programa?
- d) Quina és la funció d'a1 al programa? I la d'a0?

3. Realitzeu les següents accions sobre el Simulador i expliqueu-ne els resultats:

- e) Establiu la relació entre el codi que hi ha a la finestra 'source code' i el que apareix a la finestra 'Executable code'.
- f) Observeu que inicialment PC=0h (el PC apunta a la següent instrucció executable del programa). Esbrineu com ho fa.
- g) Executeu el programa, instrucció per instrucció, observant el resultat d'executar cada una de les instruccions:
 - Abans d'executar cada instrucció prediu les variacions que es produiran al banc de registres i a la memòria.
 - Comproveu que el resultat de l'execució del programa coincideix amb el que havíeu previst.
 - Per cada instruccions anoteu tots els canvis.

Treball a fer a casa (entregable)

1.- Seleccionem inicialment Single Cycle Processor. Obre l'editor de text predefinit i escriu el següent programa:

```

.data
#arguments i valors...
dataByte0: .byte 4
dataByte1: .byte 55
dataByte2: .byte 255
dataByte3: .byte 31
data1: .word 1
data2: .word 0
data3: .word 4

.text
main:
    lw a0, data1
    lb t0, dataByte0
    sw a0, data2(zero)
    lw a1, data3

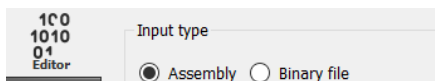
loop:
    beq a1, a0, salta
    sub a1, a1, a0
    j loop

salta:
    lw a3, data2

end:
    j end

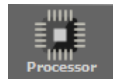
```

2.- Guarda el programa amb el nom Practica1_riscv.asm (sobre tot és important l'extensió). Obre el simulador Ripes. Posiciona a la part de l'editor.



3.- Clica File i selecciona l'opció Load Assembly File. Veureu el vostre codi a la part esquerra de la pantalla (source code) i el codi desensamblat a la part dreta.

4.- Clica a la part del processador

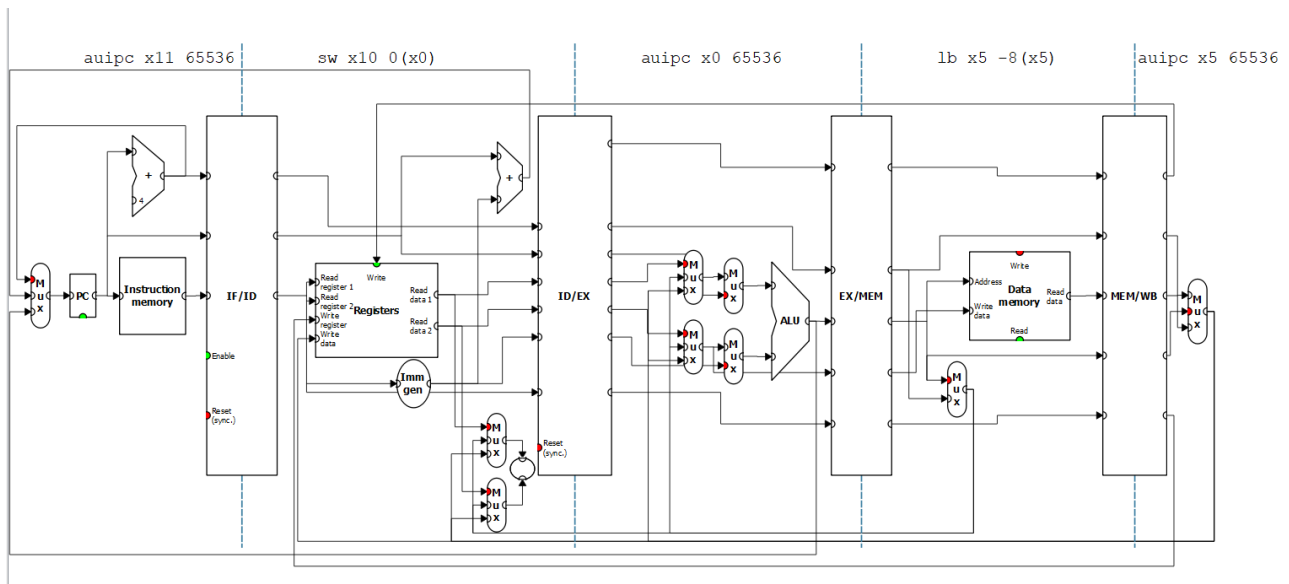


```

auipc x10 65536
lw x10 16(x10)
auipc x5 65536
lb x5 -8(x5)
auipc x0 65536
sw x10 0(x0)
auipc x11 65536
lw x11 0(x11)
beq x11 x10 12
sub x11 x11 x10
jal x0 32
auipc x13 65536
lw x13 -24(x13)
jal x0 52


```

5.- Executa el programa pas a pas. El programa que s'executarà és el desensamblat. El que tens a la figura superior. La primera instrucció el que fa es incrementar el registre PC per carregar la instrucció des de la memòria de programa. Executa inicialment el programa amb el processador que hem fet servir des de l'inici: el RSCP. Tot seguit canvia de processador i posa el R5SP. El processador amb el que esteu simulant ara és un processador multicicle amb un pipeline de 5 estats. Això vol dir que, en un cicle de rellotge està executant 5 fases de 5 instruccions diferents.



Analitza el comportament del programa per tots dos processadors.

h) Quan triguen a executar el programa? Per què?

i) Quin és el diagrama d'execució (botó )?

j) Quin és el màxim nombre d'instruccions que s'han executat simultàniament?

Recorda que el simulador Ripes té 32 registres, entre els que tenim el primer x0, anomenat zero i que només és de lectura i té el valor 0. Tota la resta són accessibles tant per lectura com per escriptura.

La següent figura mostra el conjunt de registres i les instruccions .

Registro	Nombre ABI	Descripción
x0	zero	Alambrado a cero
x1	ra	Dirección de retorno
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Link register temporal/alternativo
x6-7	t1-2	Temporales
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Argumentos de función/valores de retorno
x12-17	a2-7	Argumentos de función
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporales
f0-7	ft0-7	Temporales, FP
f8-9	fs0-1	Saved registers, FP
f10-11	fa0-1	Argumentos/valores de retorno, FP
f12-17	fa2-7	Argumentos, FP
f18-27	fs2-11	Saved registers, FP
f28-31	ft8-11	Temporales, FP

Tarjeta de Referencia para RISC-V Abierto

Instrucciones Base para Enteros: RV32I y RV64I					Instrucciones Privilegiadas RV																										
Categoría	Nombre	Fmt	RV32I Base	+RV64I	Categoría	Nombre	Fmt	Mnemónico RV																							
Shifts	Shift Left Logical	R	SLL rd, rd1, rd2	SLLW rd, rd1, rd2	Excep. Mach-mode trap return	R	MRET																								
	Shift Left Log. Imm.	I	SLLI rd, rd1, shamt	SLLIW rd, rd1, shamt	Supervisor-mode trap return	R	SRET																								
	Shift Right Logical	R	SRL rd, rd1, rd2	SRLW rd, rd1, rd2	Interruptions Wait for Interrupt	R	WFI																								
	Shift Right Log. Imm.	I	SRLI rd, rd1, shamt	SRLIW rd, rd1, shamt	MMU Virtual Memory FENCE	R	SFENCE.VMA rd1, rd2																								
	Shift Right Arithmetic	R	SRA rd, rd1, rd2	SRAW rd, rd1, rd2	Ejemplos de las 60 Pseudoinstrucciones RV																										
Shift Right Arith. Imm.	I	SRAI rd, rd1, shamt	SRAIW rd, rd1, shamt	Branch = 0 (BEQ rs, x0, imm)	J	BEQ rs, imm																									
Aritmética	ADD	R	ADD rd, rd1, rd2	ADDW rd, rd1, rd2	Jump (uses JAL x0, imm)	J	J imm																								
	ADD Immediate	I	ADDI rd, rd1, imm	ADDIW rd, rd1, imm	MoVe (uses ADDI rd, rs, 0)	R	MV rd, rs																								
	SUBtract	R	SUB rd, rd1, rd2	SUBW rd, rd1, rd2	RETurn (uses JALR x0, 0, ra)	I	RET																								
	Load Upper Imm	U	LUI rd, imm		Extensión Opcional de Instrucciones Comprimidas (16b): RV32C																										
	Add Upper Imm to PC	U	AUIPC rd, imm		Categoría	Nombre	Fmt	RVC	Equivalente RISC-V																						
Lógica	XOR	R	XOR rd, rd1, rd2		Loads	Load Word	CL	C.LW rd', rd1', imm	LW rd', rd1', imm*4																						
	XOR Immediate	I	XORI rd, rd1, imm			Load Word SP	CI	C.LWSP rd, imm	LW rd, sp, imm*4																						
	OR	R	OR rd, rd1, rd2			Float Load Word SP	CL	C.FLW rd', rd1', imm	FLW rd', rd1', imm*8																						
	OR Immediate	I	ORI rd, rd1, imm			Float Load Word	CI	C.FLWSP rd, imm	FLW rd, sp, imm*8																						
	AND	R	AND rd, rd1, rd2			Float Load Double	CL	C.FLD rd', rd1', imm	FLD rd', rd1', imm*16																						
AND Immediate	I	ANDI rd, rd1, imm			Float Load Double SP	CI	C.FLDSP rd, imm	FLD rd, sp, imm*16																							
Comparación	Set <	R	SLT rd, rd1, rd2		Stores	Store Word	CS	C.SW rd', rd1', imm	SW rd', rd1', imm*4																						
	Set < Immediate	I	SLTI rd, rd1, imm			Store Word SP	CSS	C.SWSP rd2, imm	SW rd2, sp, imm*4																						
	Set < Unsigned	R	SLTU rd, rd1, rd2			Float Store Word	CS	C.FSW rd', rd1', imm	FSW rd', rd1', imm*8																						
	Set < Imm Unsigned	R	SLTIU rd, rd1, imm			Float Store Double	CS	C.FSD rd', rd1', imm	FSD rd', rd1', imm*16																						
						Float Store Double SP	CSS	C.FSDSP rd2, imm	FSD rd2, sp, imm*16																						
Branches	Branch =	B	BEQ rd1, rd2, imm		Aritmética	ADD	CR	C.ADD rd, rd1	ADD rd, rd1																						
	Branch ≠	B	BNE rd1, rd2, imm			ADD Immediate	CI	C.ADDI rd, imm	ADDI rd, rd1, imm																						
	Branch <	B	BLT rd1, rd2, imm			ADD SP Imm * 16	CI	C.ADDI16SP x0, imm	ADDI sp, sp, imm*16																						
	Branch ≥	B	BGE rd1, rd2, imm			ADD SP Imm * 4	CIW	C.ADDI4SPW rd', imm	ADDI rd', sp, imm*4																						
	Branch < Unsigned	B	BLTU rd1, rd2, imm			SUB	CR	C.SUB rd, rd1	SUB rd, rd1																						
Jump & Link	J&L	J	JAL rd, imm			AND	CR	C.AND rd, rd1	AND rd, rd1																						
	Jump & Link Register	J	JALR rd, rd1, imm			AND Immediate	CI	C.ANDI rd, imm	ANDI rd, rd1, imm																						
Sinc.	Synch thread	I	FENCE			OR	CR	C.OR rd, rd1	OR rd, rd1																						
	Synch Instr & Data	I	FENCE.I			eXclusive OR	CR	C.XOR rd, rd1	XOR rd, rd1																						
Ambiente	CALL	I	ECALL			MoVe	CR	C.MV rd, rd1	MV rd, rd1																						
	BREAK	I	EBREAK			Load Immediate	CI	C.LI rd, imm	LD rd, rd1, imm																						
Control Status Register (CSR)						Load Upper Imm	CI	C.LUI rd, imm	LUI rd, imm																						
	Read/Write	I	CSRWR rd, oer, rd1		Shifts	Shift Left Imm	CI	C.SLLI rd, imm	SLLI rd, rd1, imm																						
	Read & Set Bit	I	CSRRS rd, oer, rd1			Shift Right Ari. Imm.	CI	C.SRAI rd, imm	SRAI rd, rd1, imm																						
	Read & Clear Bit	I	CSRRC rd, oer, rd1			Shift Right Log. Imm.	CI	C.SRLI rd, imm	SRLI rd, rd1, imm																						
	Read/Write Imm	I	CSRRWI rd, oer, imm		Branches	Branch=0	CB	C.BEQ rd', rd1', imm	BEQ rd', rd1', x0, imm																						
Read & Set Bit Imm	I	CSRRSI rd, oer, imm			Branch≠0	CB	C.BNE rd', rd1', imm	BNE rd', rd1', x0, imm																							
Read & Clear Bit Imm	I	CSRRCI rd, oer, imm		Jump	Jump	CJ	C.J imm	JAL x0, imm																							
Loads	Load Byte	I	LB rd, rd1, imm			Jump Register	CR	C.JR rd, rd1	JALR x0, rd1, 0																						
	Load Halfword	I	LH rd, rd1, imm		Jump & Link	J&L	CJ	C.JAL imm	JAL ra, imm																						
	Load Byte Unsigned	I	LBU rd, rd1, imm			Jump & Link Register	CR	C.JALR rd1	JALR ra, rd1, 0																						
	Load Half Unsigned	I	LHU rd, rd1, imm		Sistema Env. BREAK	CI	C.EBREAK	EBREAK																							
	Load Word	I	LW rd, rd1, imm		+RV64I																										
Stores	Store Byte	S	SB rd, rd1, rd2, imm		LWU	rd, rd1, imm		Extensión Opcional Comprimida: RV64C																							
	Store Halfword	S	SH rd, rd1, rd2, imm		LD	rd, rd1, imm		Todo RV32C (excepto C.JAL, 4 word loads, 4 word stores) más:																							
	Store Word	S	SW rd, rd1, rd2, imm					ADD Word (C.ADDW)	Load Doubleword (C.LD)																						
								ADD Imm. Word (C.ADDIW)	Load Doubleword SP (C.LDSI)																						
								SUBtract Word (C.SUBW)	Store Doubleword (C.SD)																						
								Store Doubleword SP (C.SDSI)																							
Formatos de Instrucciones de 32 bits					Formatos de Instrucciones de 16 bits (RV32C)																										
31	27	26	25	24	20	19	15	14	13	11	7	6	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
R	func7					rs2		rs1		func3		rd		opcode	CR	func4					rd/rs1		rs2								op
I	imm[11:0]							rs1		func3		rd		opcode	CI	func3				imm		rd/rs1		imm							op
S	imm[11:5]					rs2		rs1		func3		imm[4:0]		opcode	CSS	func3					imm			rs2						op	
B	imm[12:10:5]						rs2		rs1		func3		imm[4:1:11]		opcode	CIW	func3						imm			rd'				op	
U													rd		opcode	CL	func3				imm		rs1'		imm		rd'			op	
J													rd		opcode	CS	func3				imm		rs1'		imm		rs2'			op	
																CB	func3				offset		rs1'			offset				op	
																CJ	func3								jump target					op	

RISC-V Base-Enteros (RV32I/64I), privilegiado, y RV32C/64C opcional. Registros x1-x31 y el PC son de 32 bits en RV32I y 64 en RV64I (x0=0). RV64I agrega 12 insts. para los datos anchos. Toda instrucción de 16 bits RVC se mapea a una instrucción RV existente de 32 bits.

Informe

Realitzeu un informe de la pràctica explicant el funcionament del simulador utilitzat.

1. Captura de pantalla del simulador RIPES amb les modificacions adequades a les instruccions incorrectes (2 punt).
2. Responen les preguntes a), b), c) i d) (2 punt).
3. Responen les preguntes e), f) i g) (2 punt).
4. Responen les preguntes h), i) i j) (3 punts).

L'informe ha de contenir un apartat d'objectius i un de conclusions (1 punt presentació i redacció).

Pràctica 2: Exercicis amb el simulador Ripes

Aquesta pràctica és novament una sessió guiada pel professor.

El professor recordarà el conjunt d'instruccions, el funcionament del simulador i l'adreçament a memòria.

Objectius

L'objectiu d'aquesta pràctica és familiaritzar-nos amb el funcionament dels registres que hi ha a la CPU. Per tal d'assolir aquest objectiu, farem servir el simulador Ripes.

Com s'ha explicat a teoria, dividim el conjunt de registres fonamentalment en dos tipus:

–Registres de propòsit general

–Registres específics

Els registres de propòsit general es troben al banc de registres. Són 32 registres [x0 : x31] de propòsit general, el registre x0 sempre val 0, per tots dos casos, mentre que la resta, el seu contingut és variable. Així una bona forma d'inicialitzar un registre (posar a 0 el seu contingut) serà per exemple:

ADD a0, zero, zero => Suma el contingut de x0 + el contingut de x0 i es desa en a0

Un registre de propòsit específic és per exemple el Program Counter (PC), on tenim l'adreça de la següent instrucció a executar. RISC-V té altres registres de propòsit específic (per gestionar el temps, per fer entrada/sortida amb el hardware, etc.) però el simulador Ripes no ho implementa.

Exercici 1 (Guiat)

Carrega el següent programa en el simulador Ripes.

```
.data
Dades: .word 9, 3, 4, 1      # vector de 4 enters
Resultat: .word 0, 0        # vector de 2 enters
.text                        # directiva d'inici de programa
main:
    la a0, Dades             # a0 actuarà com a punter
    lw a1, 0(a0)             # 0(a0) = Dada[0]
    lw a2, 4(a0)             # 4(a0) = Dada[1]
    lw a3, 8(a0)             # 8(a0) = Dada[2]
    lw a4, 12(a0)            # 12(a0) = Dada[3]
loop:
    add a5, a1, a2
    add a6, a4, a2
    addi a3, a3, -1
    bgez a3, loop
    sw a5, 16(a0)            # 16(a0) = Resultat[0]
    sw a6, 20(a0)            # 20(a0) = Resultat[1]
end:    nop                  #final de programa
```

Quina operació fa aquest codi?

Exercici 2 (Guiat)

Feu un programa similar als de l'exercici 1. Inicialitzeu primer el contingut de A0. Carregueu al A1 el valor 0000110000001011b. Carregueu al registre A2 el valor 0000000000010001b. Feu l'operació $A0 \leftarrow A0 + A1$ i decrementeu el valor de A2. Feu això en un bucle fins que el valor contingut en A2 sigui 0.

Exercici 3 (a fer per l'alumne)

Ens demanen calcular un algoritme que ens faci el següent: Donades dues entrades emmagatzemades en les posicions de memòria A i B fer la comparació. Si $A > B$ calculem la suma. Si $B > A$ fem la diferència ($B - A$) i si són iguals que multipliqui el seu valor. Carregueu diferents valors en memòria per tal de comprovar el funcionament del programa. Quina instrucció feu servir per fer la multiplicació? Què és més eficient, una instrucció directament que faci aquesta operació o el càlcul iteratiu? Raoneu la resposta.

Informe

Expliqueu-ne la feina realitzada en aquesta pràctica.

A l'exercici 1 (2 punts):

- Un cop acabat el programa quant val el contingut de A5?
- Un cop acabat el programa quant val el contingut de A6?
- Un cop acabat el programa quant val el contingut de A3?
- Quines instruccions reals s'utilitzen per a les pseudoinstruccions nop i bgez? (mireu el codi màquina que es genera i consulteu la taula d'instruccions, utilitzant els camps opcode i func3/7 si cal)
- En quina posició de memòria es troben els valors de Resultat? En quina posició de memòria es guarda el contingut del registre A6?
- Quan executem bgez, quin registre es veu afectat?

A l'exercici 2 (2 punts):

- Quina operació matemàtica està realitzant aquest codi?
- Quant val el contingut d'A0, A1 i A2 quan acaba el programa?
- Quantes iteracions del bucle executa el codi?

A l'exercici 3 (6 punts):

- Expliqueu el funcionament del programa que heu implementat.
- Feu que el resultat es guardi a la posició de memòria 24h bytes després de l'inici de la secció .data.

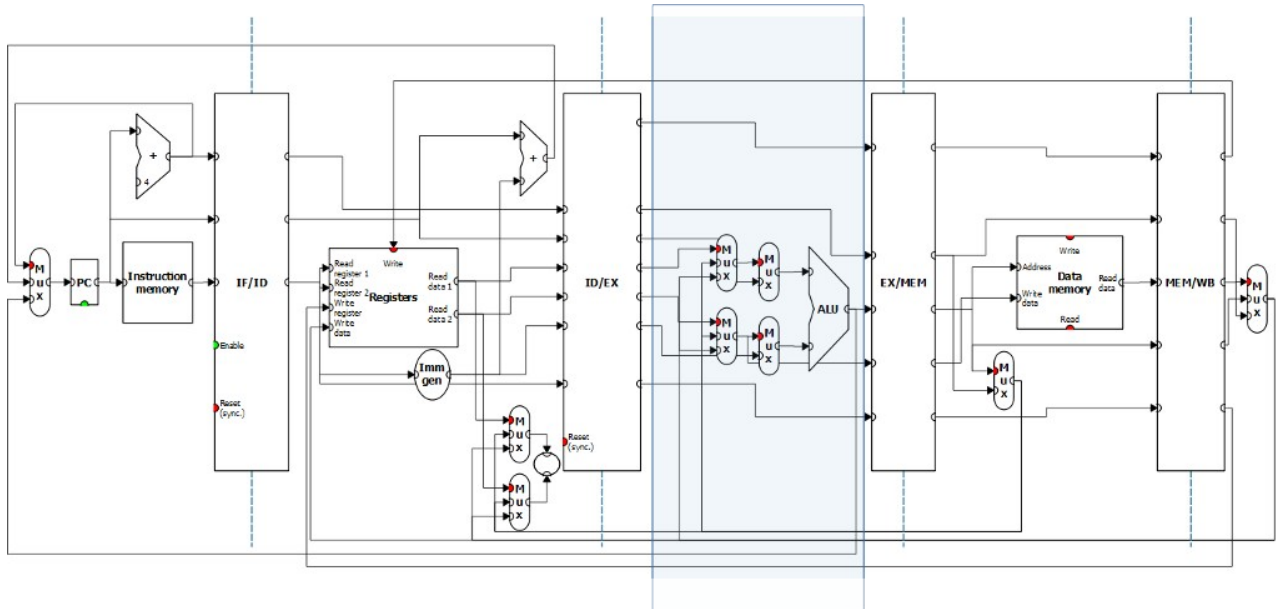
Recordeu que l'informe ha de tenir una secció inicial d'introducció i objectius, i una secció final de conclusions.

Pràctica 3: Operacions i bucles amb RISC-V

Objectius

Aquesta pràctica té com a principal objectiu la utilització de sentències de control de flux, generació de bucles i salts condicionals i incondicionals. Per altra banda, un altre objectiu és solidificar els coneixements adquirits per part de l'alumne en les pràctiques anteriors.

Estructura de Ripes V



La zona marcada és el lloc on es realitza la comprovació per decidir si es produeix un salt o no. Les diferents opcions de salt venen donades per la taula presentada en el següent punt, i on el que es fa és comparar entre dos entrades, associades a dos registres.

Instruccions de salt

Instrucció	Operació	Equivalent
beq rs, rt, destí	Salta a “destí” si el registre $rs = rt$	
bne rs, rt, destí	Salta a “destí” si el registre $rs \neq rt$	
blt rs, rt, destí	Salta a “destí” si el registre $rs < rt$	
bge rs, rt, destí	Salta a “destí” si el registre $rs \geq rt$	
bltu rs, rt, destí	Salta a “destí” si el registre $rs < rt$ (*)	
bgeu rs, rt, destí	Salta a “destí” si el registre $rs \geq rt$ (*)	
beqz rs, destí	Salta a “destí” si el registre $rs = 0$	beq rs, zero, destí
bnez rs, destí	Salta a “destí” si el registre $rs \neq 0$	bne rs, zero, destí
blez rs, destí	Salta a “destí” si el registre $rs \leq 0$	bge zero, rt, destí
bgez rs, destí	Salta a “destí” si el registre $rs \geq 0$	bge rs, zero, destí
bltz rs, destí	Salta a “destí” si el registre $rs < 0$	blt rs, zero, destí
bgtz rs, destí	Salta a “destí” si el registre $rs > 0$	blt zero, rs, destí
bgt rs, rt, destí	Salta a “destí” si el registre $rs > rt$	blt rt, rs, destí
ble rs, rt, destí	Salta a “destí” si el registre $rs \leq rt$	bge rt, rs, destí
bgtu rs, rt, destí	Salta a “destí” si el registre $rs > rt$ (*)	bltu rt, rs, destí
bleu rs, rt, destí	Salta a “destí” si el registre $rs \leq rt$ (*)	bgeu rt, rs, destí
j destí	Salta a “destí” incondicionalment	

* Comparacions sense complement a dos, considerant només números positius.

Problema 1

Els salts ens permeten executar diferents blocs de codi i així podem implementar estructures de control de flux.

Exemple **if**: distància entre dos números enters

Per calcular la distància entre dos números enters, farem la resta i llavors el valor absolut.

$$|a - b|$$

Exemples:

$$|5 - (-6)| = 11$$

o

$$|-6 - 5| = 11$$

El programa següent en C calcula aquesta distància. Com es faria en RISC V?

Alt nivell (C)	RISC V
<pre>int a = 5; int b = -6; int resultat; if (a >= b) resultat = a - b; else resultat = b - a;</pre>	<pre>.data a: .word 5 b: .word -6 resultat: .word 0 .text la a0, a lw a1, 0(a0) lw a2, 4(a0) # condició cert: # branca certa fals: # branca falsa end: sw a3, 8(a0)</pre>

Nota important: Fixa't que la condició per al codi d'alt nivell ($a \geq b$) és la contrària que fem servir en el codi en llenguatge ensamblador.

Això passa perquè en el codi d'alt nivell la condició indica que s'executa la branca certa ($a \geq b$).

En canvi, en el codi màquina de l'exemple anterior indica que saltem a la branca falsa ($a < b$).

També és important recordar que després d'executar la branca certa cal saltar-se el bloc de la branca falsa.

Problema 2

Exemple **while**: Fibonacci

La successió de Fibonacci és una successió matemàtica de nombres naturals tal que cada un dels seus termes és igual a la suma dels dos anteriors.

$$F_n = F_{n-1} + F_{n-2}$$

sempre es parteix dels valors inicials

$$F_0 = 0, F_1 = 1$$

i es a partir del elements F_0 i F_1 que es genera la resta d'elements. Exemples:

$$F_2 = 1$$

$$F_3 = 2$$

$$F_4 = 3$$

...

$$F_9 = 34$$

$$F_{10} = 55$$

El programa següent calcula el terme de la sèrie de Fibonacci indicat per la variable “comptador” ($F_{\text{comptador}}$), i el desa a la variable “resultat”. **Com es faria en RISC V?**

Alt nivell (C)	RISC V
<pre>int comptador = 10; int resultat; int a = 0; int b = 1; while (comptador > 0) { int t = a + b; a = b; b = t; comptador--; } resultat = a;</pre>	<pre>.data comptador: .word 10 resultat: .word 0 .text la a0, comptador lw a0, 0(a0) addi a1, zero, 0 addi a2, zero, 1 loop: # condició # cos del bucle end: la a0, resultat sw a1, 0(a0)</pre>

Nota important: De nou, la condició per al codi d'alt nivell ($\text{comptador} > 0$) és la contrària que fem servir en el codi en llenguatge ensamblador.

En aquest cas, el codi d'alt nivell avalua si es manté en el bucle ($\text{comptador} > 0$), mentre que el codi en llenguatge ensamblador salta si surt del bucle ($\text{comptador} = 0$).

A més, al final del bucle hem de tornar al principi.

Problema 3

Per fer a casa: implementa les parts que falten del programa ensamblador.

Exemple **while** amb **if**: màxim comú divisor

El màxim comú divisor (mcd) de dos o més nombres enters positius és el major divisor possible de tots ells.

$\text{mcd}(a,b)$

Exemples:

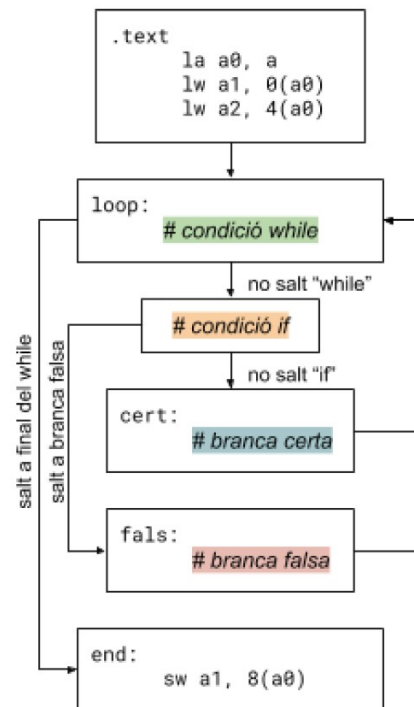
$\text{mcd}(252,105) = 21$

$\text{mcd}(13,31) = 1$

A continuació hi ha la implementació en C usant l'algorisme d'Euclides. Dins d'un bucle "while", a cada iteració es fa una comparació en un "if". **Com es faria en RISC V?**

Tingues en compte el flux del programa ensamblador, i el que hem explicat en els exercicis anteriors sobre com avaluem de forma diferent les condicions en llenguatge d'alt nivell i en llenguatge màquina, i que algunes estructures de flux necessiten també salts incondicionals.

Alt nivell (C)	RISC V
<pre>int a = 252; int b = 105; int resultat; while (a != b) { if (a > b) a = a - b; else b = b - a; } resultat = a;</pre>	<pre>.data a: .word 252 b: .word 105 resultat: .word 0 .text la a0, a lw a1, 0(a0) lw a2, 4(a0) loop: # condició while # condició if cert: # branca certa fals: # branca falsa end: sw a1, 8(a0)</pre>



Informe

Exercici 1

1. Quines instruccions de salt condicional hem fet servir? En quin cas salten?
2. Què fan les instruccions de salt condicional quan la condició no es compleix?
3. Per què hem utilitzat les instruccions de salt incondicional?

Exercici 2

1. Quin tipus d'estructura de control de flux indica normalment un salt cap enrere?
2. Omple la següent taula executant el programa.

Valor inicial	Valor a "resultat"	# cicles	# instruccions
F ₀			
F ₁			
F ₂			
F ₃			
F ₄			
F ₅			
F ₆			
F ₂₅			
F ₄₆			
F ₄₇			

3. Què passa amb el resultat F₄₇?

Exercici 3

Describeu el programa que has implementat. Quins salts has usat? Quins són condicionals i quins són incondicionals, i per què?

Pràctica 4: Microinstruccions en Ripes

(1 sessió)

Objectiu

L'objectiu de la pràctica és visualitzar els diferents passos que ha de fer un microprocessador per tal d'executar una instrucció.

Introducció

El número de cicles dividit per la freqüència de funcionament del processador ens defineix el temps d'execució de la instrucció. Les diferents accions que es realitzen en cada cicle forma el que es coneixen com microinstruccions.

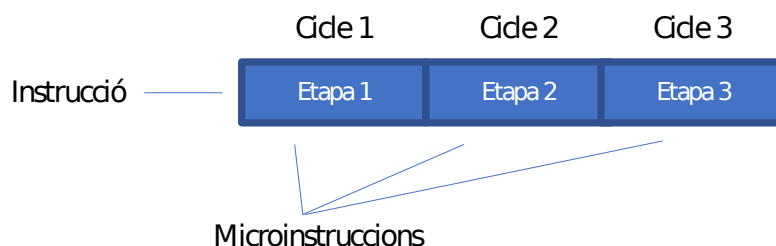
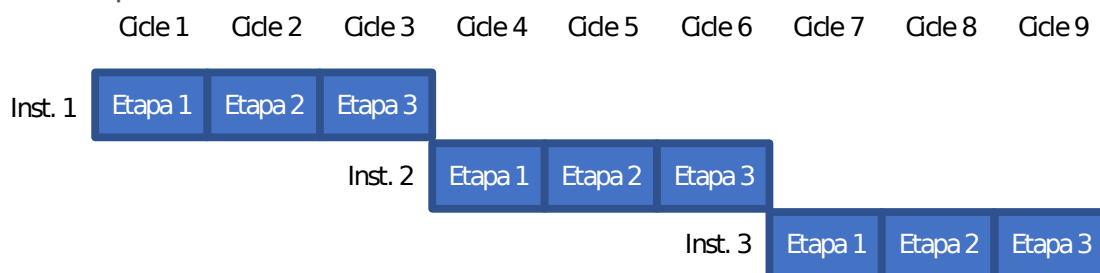


Figura 1. Concepte de microinstrucció

Com ja vàrem veure a l'apartat *Estructura bàsica de la CPU* d'aquest guió de pràctiques de l'assignatura, hi ha processadors que han d'executar totes les etapes d'una instrucció abans d'executar la següent, però també n'hi ha d'altres que a costa de multiplicar recursos de hardware poden executar diferents etapes d'una instrucció com a una cadena de muntatge, o *pipeline*.

a) Sense Pipeline



b) Amb Pipeline

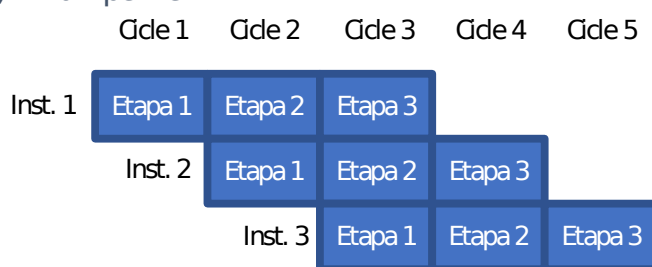


Figura 2. Cronograma d'execució d'instruccions en un processador a) sense *pipeline* i b) amb *pipeline*. El temps avança seguint un eix imaginari horitzontal de esquerra a dreta. Pel que fa a les micro-instruccions, aquestes s'incrementen seguint un eix imaginari vertical que va de dalt a baix.

En ciència de computadors, el *pipeline* RISC es una solució de micro-arquitectura que té com a propòsit la execució d'una instrucció per circle de rellotge. Aquest *pipeline* executa una instrucció en cinc etapes: Instruction Fetch, Instruction Decode, Execute, Memory Access i Writeback. A continuació s'expliquen les accions que realitza el processador en cadascuna d'aquestes etapes:

- **Instruction Fetch (IF):** Llegeix la següent instrucció de memòria i incrementa el comptador de programa.
- **Instruction Decode (ID):** Descodifica la instrucció. Llegeix els registres d'operands i computa direccions de salt (si escau).
- **Execute(EX):** Executa una operació aritmètic-lògica o realitza un salt.
- **Memory Access (MEM):** Realitza accessos a memòria de lectura o escriptura.
- **Writeback(WB):** Escriu els resultats de les operacions a un registre.

Alguns dels processadors que comparteixen aquesta microarquitectura són: MIPS (Microprocessor without Interlocked Pipelined Stages), SPARC (Scalable Processor Architecture), Motorola 88000 (o m88k) i, es clar, el RISC-V (Reduced Instruction Set Computer). A la següent figura podem veure el típic pipeline de cinc etapes d'una màquina RISC.

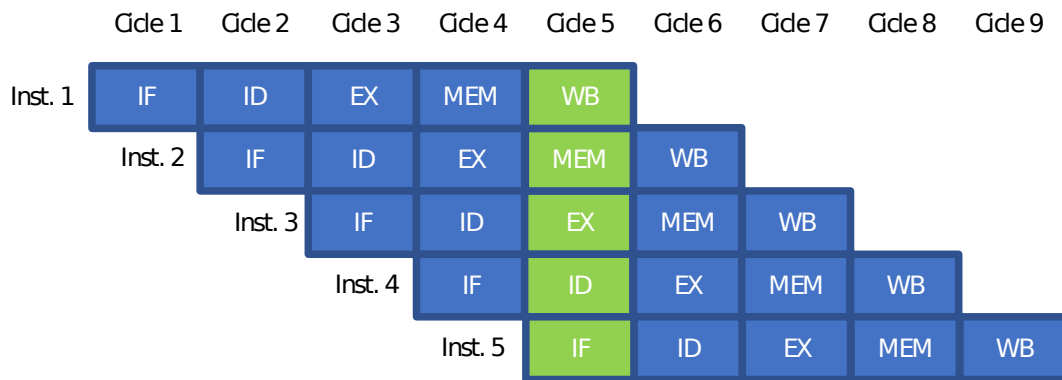


Figura 3. Pipeline de cinc etapes bàsic a una màquina RISC. Veiem que al cinquè cicle de rellotge (columna verda) la primera instrucció s'acaba d'executar, la segona instrucció es troba a l'etapa IF, la tercera a EX, la quarta a ID, i la cinquena a IF. Noteu que si no hi haguèss esperes (stall(s) deguts a un hazard), a partir del cinquè cicle cada instrucció s'executaria en un cicle de rellotge.

Exercici guiat


Haurem d'executar els següents codis, cicle a cicle (microinstrucció a microinstrucció)

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
bgt a7, zero, salta
salta:
la a0, resultat
sw a3, 0(a0)
```

```
.data
resultat: .word 0
.text
main:
add a3, zero, zero
add a7, zero, zero
addi a2, zero, 4
add a3, a2, a3
addi a7, a7, -1
la a0, resultat
sw a3, 0(a0)
```

- 1) Abans d'executar els codis tracta d'esbrinar la seva funcionalitat. Faran el mateix? Creus que trigaran el mateix nombre de cicles en executar-se?
- 2) Ves a la finestra del Ripes V dedicada al processador (Processor). Busca entre les opcions del Simulator control, la Pipeline table.

- 3) Executa els dos codis cicle a cicle fixant-te com les instruccions van passant per les diferents etapes del pipeline. Compta els nombre de cicles que es necessiten per executar cadascun dels codis i compara les Pipeline tables.
- 4) Què signifiquen el signes '-' que apareixen a les Pipeline tables?
- 5) Com afectaria al nombre total de cicles d'execució el següent canvi en el codi:

<pre>.data Resultat: .word 0 .text main: add a3, zero, zero add a7, zero, zero addi a2, zero, 4 add a3, a2, a3 addi a7, a7, -1 bgt a7, zero, salta salta: la a0, Resultat sw a3, 0(a0)</pre>		<pre>.data Resultat: .word 0 .text main: la a0, Resultat add a3, zero, zero add a7, zero, zero addi a2, zero, 4 add a3, a2, a3 addi a7, a7, -1 sw a3, 0(a0)</pre>
--	---	---

Realització de la pràctica

Ripes: Executa el següent programa microinstrucció a microinstrucció:

```
.data
valorDada: .word 2
guardaResultat: .word 0
.text
main:
lw a7, valorDada
addi a2, zero, 9
add a3, zero, zero
loop:
add a3, a2, a3
addi a2, zero, -1
bgt a7, zero, loop
la a0, guardaResultat
sw a3, 0(a0)
```

Informe

Explica detalladament la pràctica realitzada. Fes els diagrames necessaris per entendre i mostrar el cicle d'execució dels diferents tipus d'instruccions als dos simuladors.

Preguntes sobre el simulador RIPES:

Per resoldre aquestes qüestions es necessari mirar l'estat del pipeline o utilitzar la *Pipeline table*:

- 1) Quin es l'estat de cadascuna de les cinc etapes del pipeline al cicle 6? I al 8?
- 2) Quins senyals de control s'activen en el cicle 4? A quines instruccions del codi corresponen?
- 3) Quins són els valors a les sortides dels multiplexors assenyalats a la figura al cicle 7:

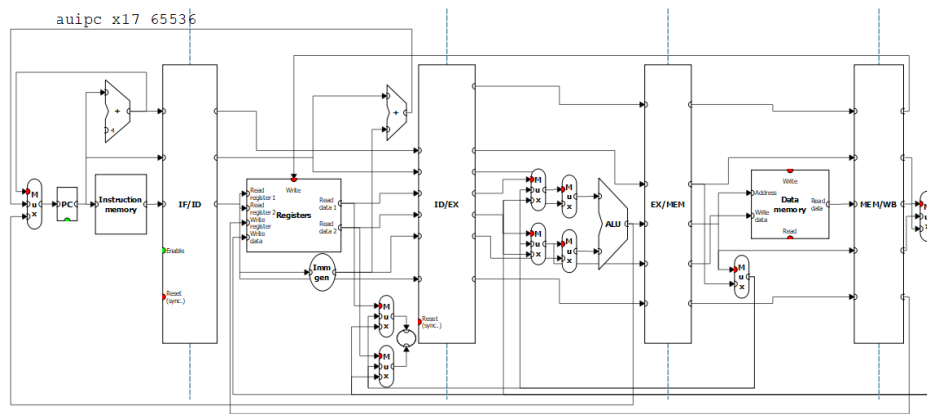


Figura 4. Detall de les etapes del RISC-V en el simulador RIPES.
L'el·lipse vermella assenyalava els multiplexors que s'utilitzen a la qüestió 3.

- 4) Perquè els valors apareixen en aquest ordre?
- 5) Llegeix amb cura la part del codi amb que s'implementa el loop. Tenint en compte el que has vist a la qüestió 3), justifica perquè el pipeline al cicle 9 presenta aquest estat:

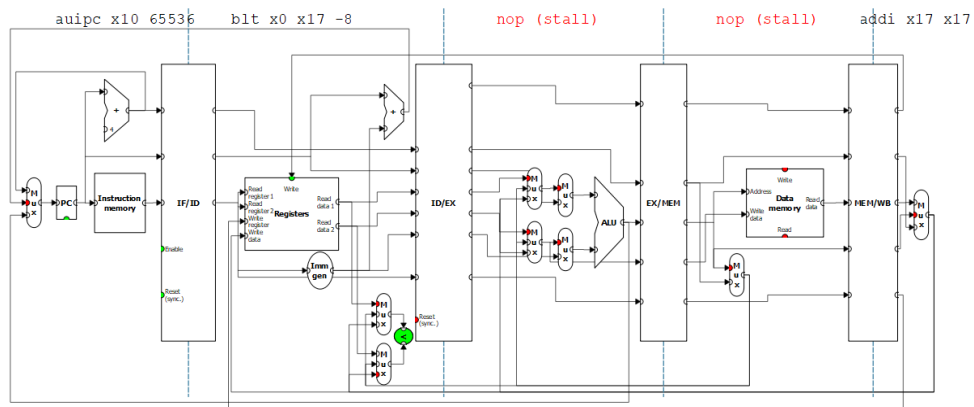


Figura 5. Estat del pipeline al cicle 9.

- 6) Quan NO es produeix el salt, quants cicles triga en executar-se la instrucció **bgt a7, zero**, loop?
- 7) Quan s'està executant la instrucció de salt, però el salt NO es produeix, a quina posició apunta la memòria d'instruccions?
- 8) Quan es produeix el salt, quants cicles triga en executar-se la instrucció **bgt a7, zero**, loop?
- 9) Quan s'està executant la instrucció de salt, i el salt es produeix, a quina posició apunta la memòria d'instruccions?