**Pràctica 1**

**Introducció / Objectius**

En aquesta primera pràctica volem començar a familiaritzar-nos amb la Màquina Ripes, el seu simulador (amb les dues versions: RSCP i R5SP) i les comandes rudimentàries d’aquesta. Veurem com crear un nou fitxer, com compilar-lo, quins errors de compilació poden aparèixer i com podem seguir el flux del programa. Comprendrem també l’analogia entre llenguatge màquina i el llenguatge assemblador. A continuació un resum de les tasques que realitzarem durant la pràctica:

* Exercicis guiats resolts a classe de laboratori amb ajuda del professor.
* Observarem alguns errors de compilació d’un codi sense una funcionalitat concreta.
* Escriurem un petit programa i esbrinarem què fa.
* Executarem un programa donat pel professor i esbrinarem què fa.
* Compararem aquests dos últims programes.
* Veurem les diferències entre RSCP i R5SP.

**Exercicis guiats**

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

|  |  |  |
| --- | --- | --- |
| Instruccions | Correcte/Incorrecte | Motiu |
| LW a1(R0), a1 | Incorrecte | La forma correcta seria **LW a1, 0(a1)**. Davant del registre es posa un offset, no un altre registre. |
| SW a1, a1(3) | Incorrecte | La forma correcte seria **SW a1, 3(a1)**. El registre va entre parèntesi i l’offset al davant. |
| BNE 6(t1) | Incorrecte | La forma correcte seria **BNE t1, t2, etiqueta**. Compara dos registres (t1, t2) si són iguals salta a la posició on tenim l’etiqueta. |
| ADDI t2, #11, 5(t3) | Incorrecte | La forma correcte seria **ADDI t2, t3, 11**. t2 és registre destí, t3 és origen. Al contingut de t3 li sumem un 11 per guardar-lo a t2. |
| SUB zero, a2, a3 | Correcte | La instrucció és correcta, restarà a2-a3 però no podrà guardar-ho. |
| LOAD 3(a0), a1 | Incorrecte | La forma correcta seria **LOAD a1, 3(a0)**. Sempre primer el registre on volem fer la càrrega. |

**Exercici 2:**

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

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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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 | A5EE | **------** | **------** | +4 |
| E | **SUB zero, a2, a3** | 0 | **------** | **------** | **------** | **------** | **------** | +4 |
| f | **SRLI a1, a1, 1** | 0 | 7A2D | **------** | **------** | **------** | **------** | +4 |

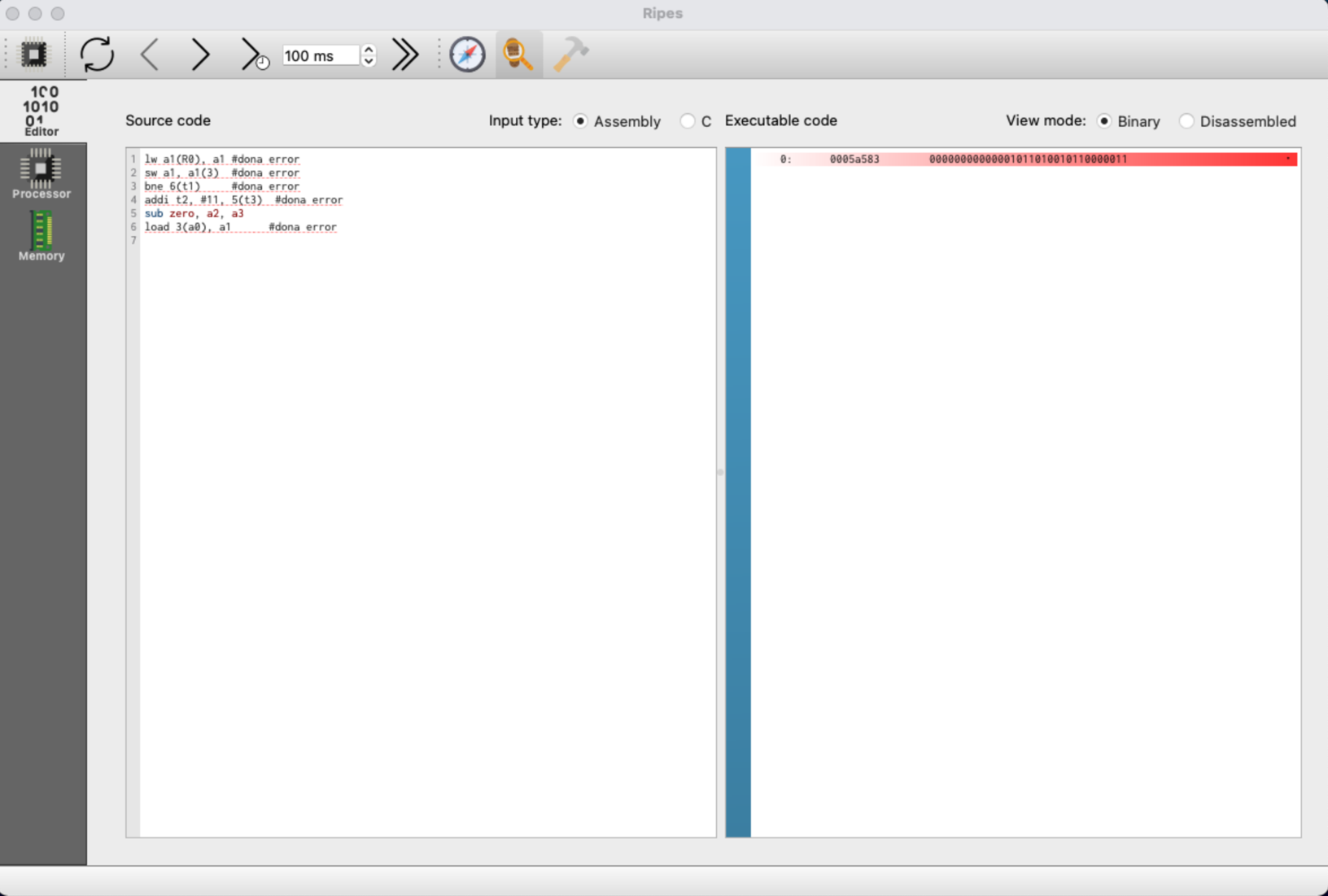
Breu descripció del que fa cada instrucció:

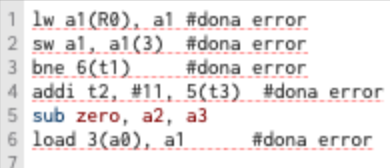
1. **a:** Guarda en a1 el contingut apuntat per [a4] + 0.
2. **b:** Guarda en la posició de memòria apuntada per [a5] + 0 el contingut de a1.
3. **c:** Salta a la posició on es troba l’etiqueta.
4. **d:** Suma d’immediat, contingut de a2 + 11 i ho guarda en a3.
5. **e:** Restem el contingut de a2 – el contingut de a3 i ho guardem en zero. Com que el contingut de zero sempre és 0, no podem guardar res.
6. **f:** Suma lògica. Desplacem el contingut de a1 un bit a la dreta.

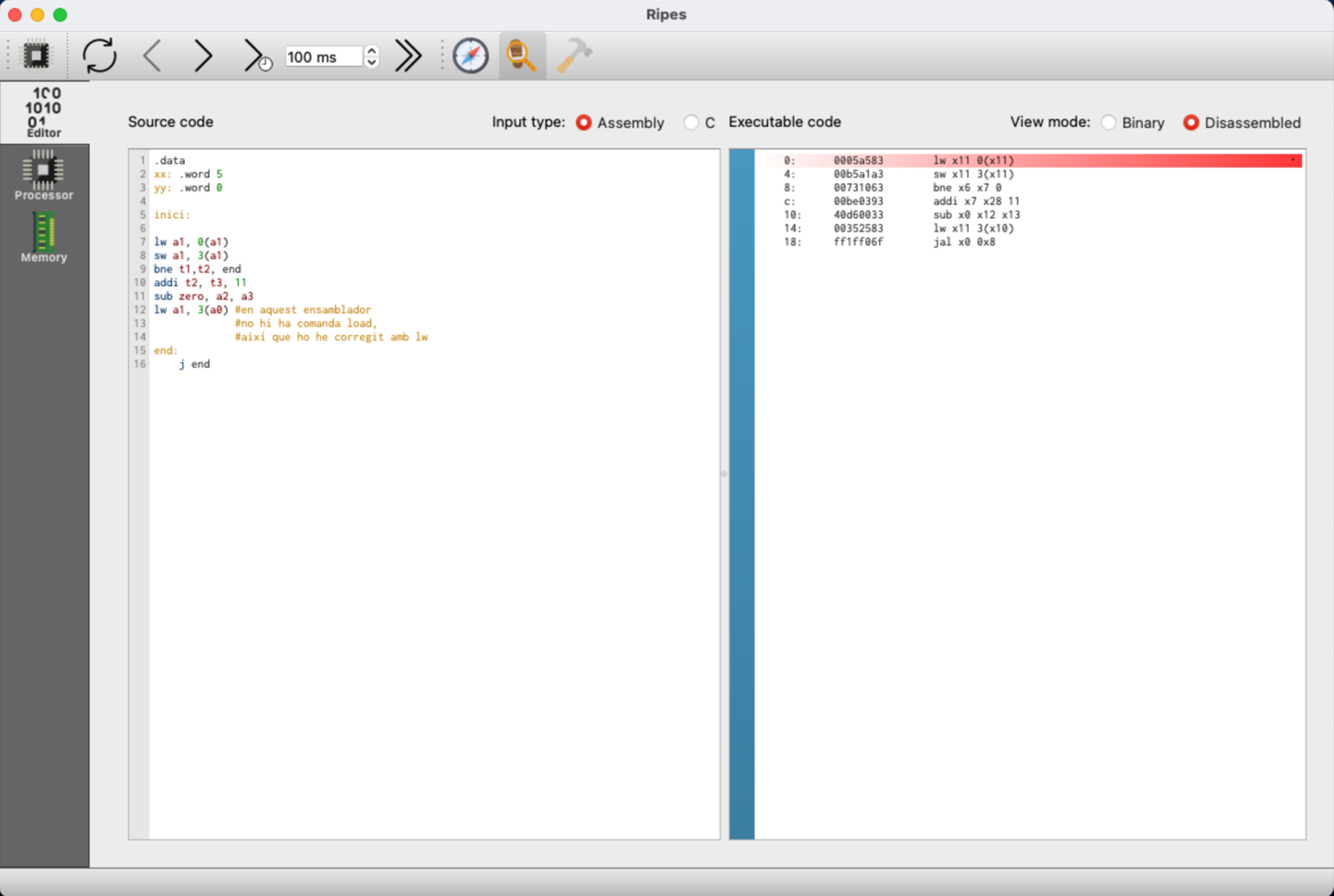
**Exercici 3:**

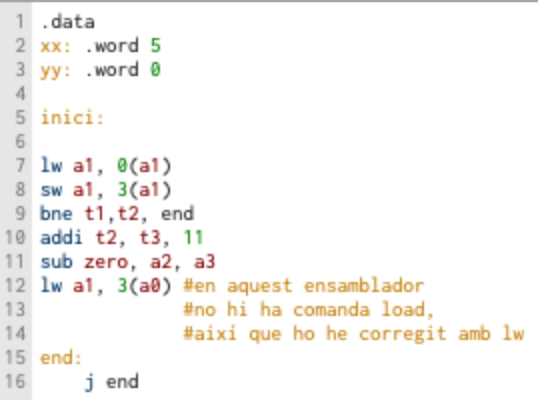
|  |  |  |  |
| --- | --- | --- | --- |
| @ | M[@] | LLENGUATGE MÀQUINA | LLENGUATE MÀQUINA (hex) |
| .data |  |  |  |
| xx: | .word 3 |  |  |
| yy: | .word 5 |  |  |
| oo: | .word 2 |  |  |
| zz: | .word 8 |  |  |
| .text |  |  |  |
| 00h | LA a0, xx | ﻿00010000000000000000010100010111 | ﻿10000517 & ﻿00050513 |
| 08h | SUB a1, a1, a1 | ﻿01000000101101011000010110110011 | ﻿40b585b3 |
| 0ch | ADD a2, zero, zero | ﻿00000000000000000000011000110011 | ﻿00000633 |
| 10h | Loop: ADDI a7, a1, -4 | ﻿11111111110001011000100010010011 | ﻿ffc58893 |
| 14h | BEQZ a7, final | ﻿00000000000010001000110001100011 | ﻿00088c63 |
| 18h | LW a3, 0(a0) | ﻿00000000000001010010011010000011 | ﻿00052683 |
| 1ch | ADD a2, a2, a3 | ﻿00000000110101100000011000110011 | ﻿00d60633 |
| 20h | ADDI a0, a0, 4 | ﻿00000000010001010000010100010011 | ﻿00450513 |
| 24h | ADDI a1, a1, 1 | ﻿00000000000101011000010110010011 | ﻿00158593 |
| 28h | j loop | ﻿11111110100111111111000001101111 | ﻿fe9ff06f |
| 2ch | Final: SW a2, 4(a0) | ﻿00000000110001010010001000100011 | ﻿00c52223 |

**Realització pràctica**

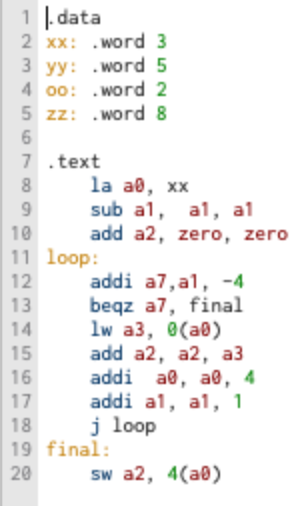
****Exercici 1:** Implementarem el codi de l’exercici 1.



Correcció:



**Exercici 2:** Implementarem el següent programa:



1. **En quina posició de memòria escriu aquest programa el resultat?**

Això ho veiem en la última línia de codi, la instrucció **sw a2, 4(a0)** guarda el contingut del registre a2 en 4 + a0. Concretament el desa a la posició de memòria amb adreça: ﻿0x10000014.

1. **Què fa el programa?**

Suma tots els elements que hi ha sota de .data (xx, yy, oo, zz) i els guarda a la posició de memòria 0Ch. El contingut de a1 es va incrementant d’1 en 1. I, en la operació **lw a3, 0(a0)** anem carregant en a3 cada cop el contingut d’una posició de memòria una unitat més enllà. Fins que sortim del loop i acabem la iteració (la suma de tots els elements que havíem declarat).

1. **Quina seqüència de control de flux (en llenguatge d’alt nivell) implementa el programa?**

Estem implementant el que és equivalent a un while o un bucle for. Dins de l’etiqueta loop tenim una condició beqz que ens farà sortir del loop quan a7 sigui equal to zero. El valor de a7 ve donat per **addi a7, a1, -4**, és a dir que quan a1 - 4 sigui zero, s’acabarà el bucle. Això en llenguatge d’alt nivell, podria correspondre’s a l’estructura **while i <= 4** (tenint en compte que a1 funciona com a i); equivalentment podríem fer una analogia amb un **for(i = 0; i <= 4; i++)** que també és una estructura molt utilitzada (sobretot per sumar elements d’una llista que ja sabem quants elements té).

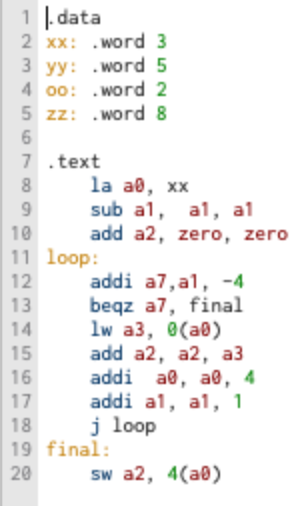
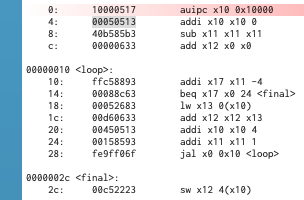
1. **Quina és la funció d’a1 al programa? I la d’a0?**

a1 -> actua com a comptador i control. Ens permet anar iterant sobre les posicions de memòria a les que accedirem per sumar tots els elementsque hi ha sota de .data (xx, yy, 00, zz); també ens serveix per sortir del bucle, com hem explicat abans, de restar-li 4 al contingut d’aquest registre (a1) i guardar-ho en el registre a7 per posteriorment fer la comprovació amb el beqz.

a0 -> ens serveix d’una banda per inicialitzar valors (registre a3) i d’altra per accedir a posicions de memòria. El valor de a0 es carrega a CPU a cada nou cicle.

**Exercici 3:**

1. **Establiu la relació entre el codi que hi ha a la finestra ‘source code’ i el que apareix a la finestra ‘Executable code’.**



A la part esquerra, és possible escriure un programa de muntatge escrit amb els conjunts d’instruccions RISC-V RV32. Sempre que es realitzin edicions en aquest programa de muntatge (i no es trobin errors de sintaxi), el codi de muntatge s’assemblarà i s’inserirà automàticament al simulador.

A la part dreta es mostra una segona vista del codi. Aquesta és una vista no interactiva del programa actual en el seu estat assemblat, denotat com el visor del programa. Podem veure el programa muntat com a instruccions RISC-V desmuntades o com a codi binari. Es pot fer clic a la barra lateral blava de la vista de la dreta per establir un punt d’interrupció a l’adreça desitjada.

1. **Observeu que inicialment PC = 0h (el PC apunta a la següent instrucció executable del programa). Esbrineu com ho fa. Perquè no apunta a la primera instrucció?**

RISC-V s’ocupa de tot en qüestió d’adreces de bytes. Per tant, quan s’utilitzen adreces de bytes, té sentit pensar que el PC augmenta en 4, ja que cadascuna de les instruccions RV32 són de 4 bytes.

Quan diem que RISC-V incrementa el PC en 4, vol dir que per a qualsevol adreça de byte X (per exemple) d’una instrucció RISC-V de 32 bits, la següent instrucció començarà a la memòria a l’adreça X + 4.

Segons el manual d’especificacions de RISC-V el PC conté l’adreça de la instrucció actual. A cada cicle es suma 4 al PC just després d’executar-se la instrucció que referenciava el PC anterior.

1. **Executeu el programa, instrucció per instrucció, observant el resultat d’executar cada una de les instruccions:**

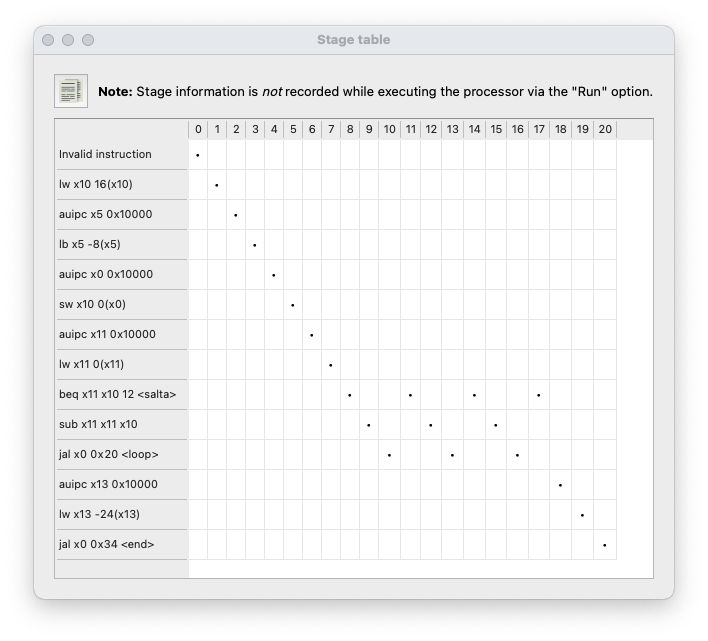
* **LA a0, xx:** el registre x10(a0) s’inicialitza a 0x10000000
* **SUB a1, a1, a1:** restarà el contingut de a1 a a1 i ho guardarà en a1. a1, llavors, quedarà amb el valor 0. En aquest cas el registre ja tenia el valor 0, així que no hi haurà cap canvi.
* **ADD a2, zero, zero:** sumarà a zero el contingut de zero (per tant, és 0) i ho guardarà en a2. Com que ja tenia aquest valor, no hi haurà cap canvi.
* **Loop:**
* **ADDI a7, a1, -4:** restarà 4 al valor del registre a1 i ho posarà en el registre a7. Aquest val al principi 0xfffffffc.
* **BEQZ a7, final:** comprova si el registre a7 és igual a 0. Veurem un canvi quan la condició es compleixi, llavors saltarà a la posició marcada per final.
* **LW a3, 0(a0):** carregarà a la posició de memòria segons el valor que tingui a0 (veurem d’aquí dos instruccions que augmenta fent una suma amb l’immediat 4). Es carrega aquest valor a a3.
* **ADD a2, a2, a3:** suma el contingut de a2 al de a3 i el posa a a2.
* **ADDI a0, a0, 4:** suma el contingut del registre a0 amb l’immediat 4 i el guarda a la posició a0. Afectarà a la instrucció **lw a3, 0(a0)**.
* **ADDI a1, a1, 1:** incrementa en 1 el valor que tenim a a1. Varia llavors el valor d’aquest registre.
* **j loop:** tornem a la posició de memòria marcada per loop.
* **Final:**
* **SW a2, 4(a0):** en el cas que es complís la condició de **beqz a7, final**, carregarà el contingut de a2 en la posició 4 + a0, que té l’adreça: 0x10000014.

**Treball a fer a casa**

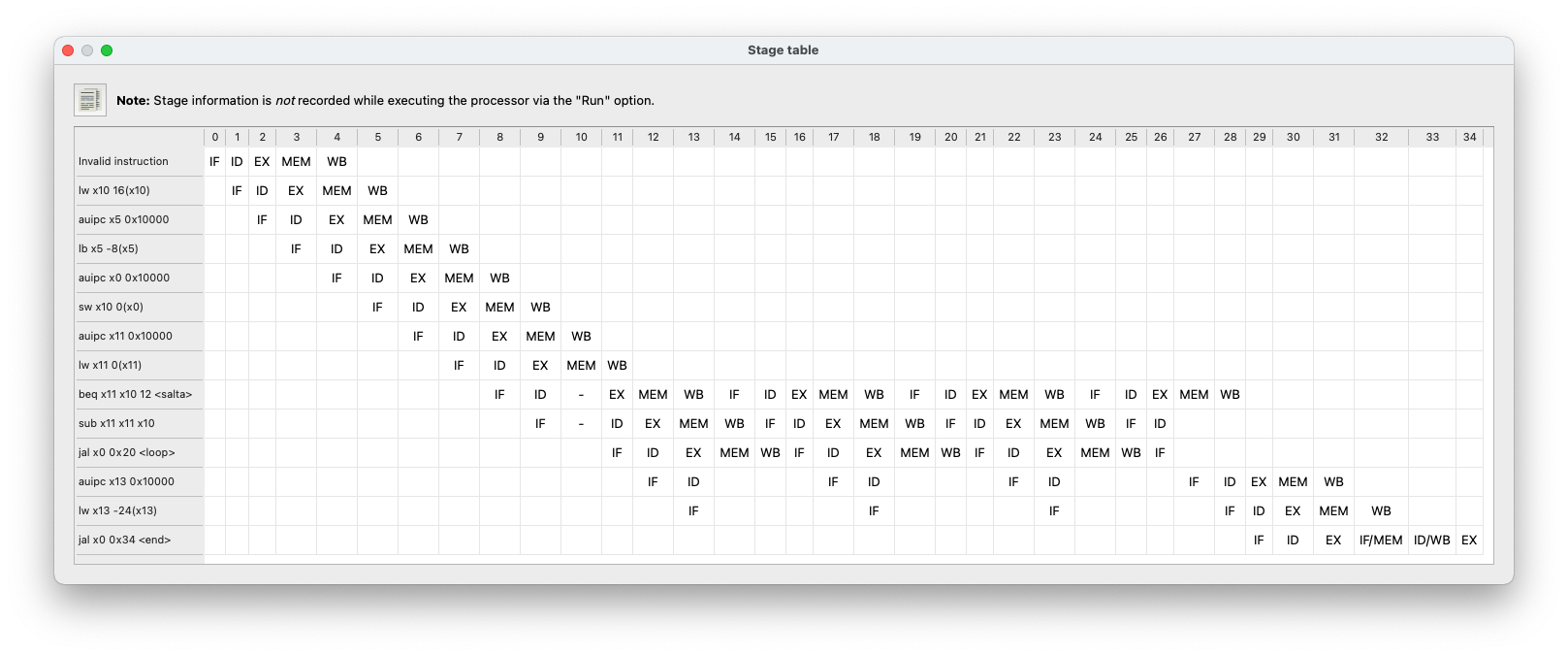
1. **Quan triguen a executar el programa ? Per què?**

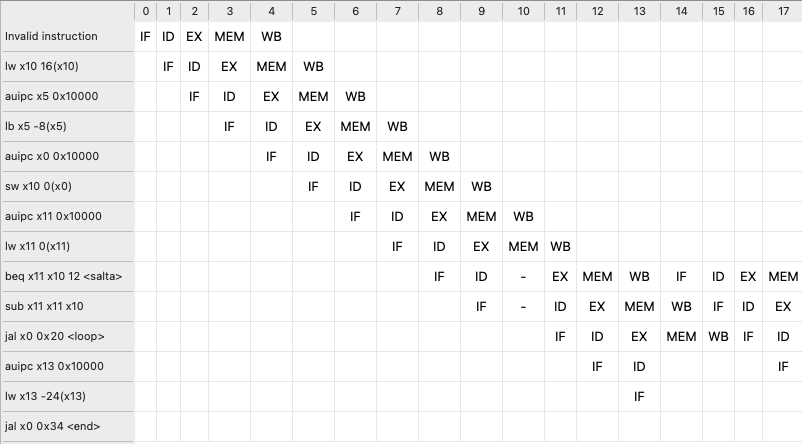
En el Single Cycle Processor triguem a executar el programa 20 cicles, però degut a que quan s’executa la instrucció **j end** ja no podem sortir de l’etiqueta end, el temps d’execució no està ben definit.

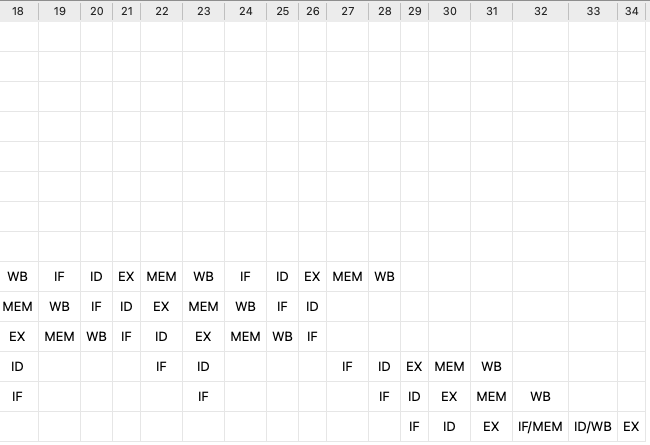
En el Processador multicicle amb un pipeline de 5 estats (R5SP) triguem a executar el programa 34 cicles, i pel mateix argument que abans, el programa no tindrà un temps definit d’execució.

1. **Quin és el diagrama d’execució?**

Single Cycle Processor (RSCP)

Processador multicicle amb un pipeline de 5 estats (R5SP)





1. **Quin és el màxim nombre d’instruccions que s’han executat simultàniament?**

En el Single Cycle Processor s’executa només una instrucció alhora.

Amb Processador multicicle amb un pipeline de 5 estats (R5SP) el nombre màxim d’instruccions que s’han executat simultàniament és de 5. Té sentit al ser un processador de 5 estats.

Imatge del programa realitzant 5 instruccions simultàniament:

**Conclusions**

En aquesta pràctica ens hem introduït al simulador de Ripes (RISC-V) i ens hem familiaritzat amb les seves comandes i amb el llenguatge assemblador. Hem realitzat els exercicis guiats que vam fer a classe i la pràctica guiada. Hem pogut observar els diferents comportaments dels dos tipus de processadors que hem fet servir: el primer un Single Cycle Processor i el segon un Processador multicicle amb un pipeline de 5 estats.