Ampliació PEC:

Unitat vectorial, coma flotant i sistema operatiu

David Cañadas Guillem Nieto Raúl Gilabert Pol Saumell
Roger Ortega Pau Morillas

Projecte d'Enginyeria de Computadors 2024

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Índex

1

Sistema operatiu

En aquest apartat, definirem cadascun dels components del nostre sistema operatiu (SO a partir d'ara) i explicarem el procés pel qual passa el SO d'ençà que s'encén el nostre processador fins que arriba a l'estat `cpu_idle`.

1.1 Kernel

Aquest és el nucli del SO. És el primer codi que s'executarà quan s'iniciï el nostre processador. Es compon de 5 arxius: `kernel.h`, `kernel.c`, `kernel-entry.s`, `kernel.ld` i `kernel-userprog.s`.

1.1.1 `kernel.h`

El header del kernel conté els elements que faran servir els altres arxius que componen aquest (vegeu més endavant). Aquest són:

1. **Adreces de memòria**. Són les adreces de memòria on comencen i acaben les instruccions i dades d'usuari i de kernel.
2. **Codis d'excepcions i d'interrupcions**. Són els codis que defineixen (mitjançant la directiva `#DEFINE`) el tipus d'interrupció o d'excepció a tractar. Són les mateixes que hi ha a la documentació de PEC además de dues noves: `0x2`, que indica un revessament en una instrucció en coma flotant; i `0x3`, que indica que hi ha hagut una divisió per 0 en una instrucció de coma flotant.
3. **Task struct**. Es defineix l'estructura que tindrà un procés. Aquesta tindrà un `struct` amb els 8 registres del banc de registres, el PC i la seva PSW. També tindrà el seu PID, el seu quàntum i el seu `list_head` (vegeu més endavant).
4. **Definició de les funcions del planificador**. Les funcions del fitxer `kernel.c` estan definides en aquest fitxer. Aquestes funcions són `sched_init`, on s'inicialitza el planificador; `sched_run` on, per cada flanc de rellotge, s'actualitza el planificador (i canvia de tasca si és necessari); `sched_schedule`, on se selecciona la tasca a canviar (o se selecciona l'idle task, en cas de no haver-hi cap més).
5. **Altres definicions**. També es defineixen el nombre de tasques totals del sistema operatiu (3), el quàntum per defecte de cada tasca (1) i els valors de la paraula d'estat de l'habilitació d'interrupcions i del mode sistema i usuari.

1.1.2 `kernel.c`

Aquest fitxer C implementa part del sistema de planificació de tasques d'un kernel. El codi gestiona la creació, la planificació i l'execució de les tasques, així com la gestió de cues per a les tasques lliures i preparades.

Aquest fitxer conté les següents declaracions globals:

1. **Funció** `cpu_idle`. Declara un punter a la funció `cpu_idle`, que s'executa quan no hi ha cap altra tasca a la readyqueue (vegeu més endavant).
2. **Struct** `idle_task`. Declara l'struct del procés idle.
3. **Vector d'struct de tasques** `tasks`. Declara un vector de tasques de la mida definit en el fitxer `kernel.c`, es a dir, tres.
4. **Cues** `readyqueue` **i** `freequeue`. Són les cues que utilitzarà el planificador a l'hora de decidir quina tasca s'executa. La `freequeue` és la cua que conté totes les tasques que es poden crear. En el moment que la cua es buida, no es podrà crear cap altra tasca. Per una altra part, la `readyqueue` conté les tasques que estan preparades per ser executades i que el planificador anirà traient a mesura que passa el temps. L'estructura que segueixen les cues s'expliquen més endavant.
5. **PID global**. És un indicador del PID que s'utilitzarà en cas que es generi una nova tasca. Això serveix perquè no hi hagi tasques amb PID diferents.
6. **Quàntum global**. És el quàntum que indica quan la tasca actual s'ha de canviar. En cas que arribi a 0, la tasca canvia per una altra de la `readyqueue`.

Juntament amb les declaracions, les següents funcions es defineixen en aquest fitxer:

1. **Funció** `list_pop_front_task`. Extreu la primera tasca d'una llista.
2. **Funció** `sys_fork`. Crea una nova tasca copiant l'estructura de la tasca actual i assigna un PID nou a la nova tasca. Retorna un 0 en cas del procés pare i el PID nou en cas del procés fill. També afegeix la tasca filla a la `readyqueue`. Retorna un -1 en cas de que no hi hagi tasques lliures a la `freequeue`.
3. **Funcions** `sys_getpid`**,** `sys_getticks`**,** `sys_readkey`. Retornen el PID de la tasca actual, els ticks de rellotge i la tecla llegida, respectivament.
4. **Funcions** `sched_needs_switch`. Determina si cal canviar de tasca en funció del PID actual i el quantum global. En cas que el PID actual sigui 0 (l'`idle_task`) i la `readyqueue` no sigui buida, retorna 1. També retorna 1 en cas que el quàntum sigui 0.
5. **Funció** `sched_needs_switch`. Canvia a una nova tasca actualitzant el quàntum global i el punter a la tasca actual.
6. **Funció** `sched_schedule`. Planifica la pròxima tasca per executar, afegint la tasca actual a la cua i seleccionant la següent tasca de la cua de preparats. Afegeix la tasca actual a la cua que rep com a paràmetre (de la cua ready o free) en cas que no sigui la tasca idle, mentre que assigna una nova tasca (una de la cua ready o la tasca idle).
7. **Funció** `sched_run`. Rutina que s'invoca cada flanc de rellotge. En cas de necessitar un canvi de tasca (mitjançant `sched_needs_switch`), executa `sched_schedule(&readyqueue)`.
8. **Funció** `sched_get_free_pid`. Genera un nou PID incrementant el PID global.
9. **Funció** `sched_init_queues`. Inicialitza les cues de planificació i afegeix totes les tasques a la `freequeue`.
10. **Funció** `sched_init_idle`. Inicialitza la tasca idle assignant-li un PID, un quantum per defecte i configurant el seu PC (amb el valor del punter `cpu_idle` i la paraula d'estat (amb les interrupcions activades i en mode kernel).
11. **Funció** `sched_init_task1`. Inicialitza la primera tasca d'usuari assignant-li un PID d'1, un quàntum per defecte i configurant el seu PC (a l'inici del codi d'usuari) i paraula d'estat (amb les interrupcions activades i en mode usuari), i fa el canvi a aquesta tasca (és a dir, `sched_task_switch(task1)`).

12. **Funció** `sched_init`. Inicialitza el sistema de planificació, configurant les tasques amb un PID per defecte de -1 i zero en els seus registres, inicialitza les cues, la tasca idle i la primera tasca d'usuari (és a dir, s'executa, en aquest ordre, `sched_init_queues()`, `sched_init_idle()` i `sched_init_task1()`.

13. **Funció** `kernel_main`. Aquesta funció s'invoca des de `kernel-entry.s`, que s'explica a continuació, i té com a funcionalitats inicialitzar el hardware executant `hw_init()` (vegeu més endavant), inicialitzar el planificador mitjançant `sched_init()` i saltar al codi d'usuari modificant la paraula d'estat al registre `s0` (amb les interrupcions activades i en mode usuari) i la següent instrucció al registre `s1` amb la primera instrucció del codi d'usuari.

1.1.3 `kernel-entry.s`

Aquest fitxer d'assembler defineix el punt d'entrada del kernel d'un sistema operatiu, així com el maneig d'interrupcions, excepcions i crides al sistema. A continuació, es proporciona una descripció detallada de cada part del codi:

1. **Declaracions globals**. Inclou el fitxer `macros.s`, defineix l'adreça de la pila de kernel i comença la seccio de codi `.text`. També defineix el punt d'entrada global `_start`.

2. `_start`. Assigna l'adreça del gestor de GSR al registre s5, configura la pila del kernel utilitzant el registre r7 i crida a la funció principal del kernel, nomenada com a `kernel_main`. La instrucció halt no s'arriba a executar.

3. `GSR_handler`. Desa l'estat del procés actual, reconfigura la pila del kernel (r7), llegeix l'origen de la interrupció, comprova si és una syscall (codi 14) o una interrupció (codi 15) i salta al gestor corresponent.

4. `exception_handler`. En cas que no hi hagi una interrupció ni una crida a sistema, s'arriba al gestor d'excepcions, on es maneja l'excepció saltant a l'adreça corresponent a la taula d'excepcions.

5. `interrupt_handler`. Arriba del gestor GSR. Maneja interrupcions obtenint l'ID i saltant a l'adreça corresponent a la taula d'interrupcions.

6. `syscall_handler`. Arriba del gestor GSR. Maneja syscalls obtenint el número de servei i saltant a l'adreça corresponent a la taula de syscalls. Desa el valor de retorn o retorna un error si el servei no és vàlid.

7. `GSR_end`. Arriba del gestor d'excepcions, interrupcions i crides a sistema. Restaura el context del procés actual i retorna de la interrupció.

8. **Funció** `cpu_idle`. Bucle infinit per mantenir la CPU inactiva mentre espera una altra tasca.

9. **Gestors per defecte:** `ESR_default_halt`, `ESR_default_resume`. Paren el processador o reprenen l'execució del programa.

10. **Secció** `.rodata`. Aquesta secció de només lectura es troben els vectors per a excepcions, interrupcions i crides a sistema. Tots els gestors utilitzen aquestes taules per obtenir les adreces de memòria on es troben les funcions o rutines.

1.1.4 `kernel.ld`

En aquest fitxer de linker es defineix l'organització de la memòria per a un executable ELF per a l'arquitectura sisa. L'estructura segueix aquest ordre

1. **Regió de Codi i Dades d'Usuari**. Marca l'inici i final del codi i dades (en aquest ordre) d'usuari. Les dades d'usuari comencen en l'adreça de memòria `0x1000`. En el fitxer `user.ld` s'organitza la secció d'usuari en més detall.

2. **Regió de Codi i Dades de Kernel**. Organitza les dades del Kernel i la pila a partir de l'adreça 0x8000, amb les seccions `.data` i `.bss`, i una reserva mínima per a la pila. Així ho fa també amb el codi de kernel, que s'enmagatzema en la direcció de memoria `0xC000`, amb les seccions `.text` i `.rodata` (read-only data).

1.1.5 `kernel-userprog.s`

Aquest fitxer te com a finalitat integrar el codi i dades d'usuari prèviament compilats dins de les seccions especificades en el linker del kernel.

1.2 Hardware

Es tracta del codi que inicialitza les funcions que interactuen directament amb els components hardware del nostre processador. Es compon de 2 arxius: `hardware.h` i `hardware.c`.

1.2.1 `hardware.h`

Aquest fitxer header conté elements que es faran servir des d'altres components de l'SO. Aquest són:

1. **Definicions de constants**. Es defineix la constant `KEY_CBUFFER_SIZE` a 32, que es la mida del buffer circular declarat en aquest mateix header.
2. **Buffer circular**. Un struct encarregat d'emmagatzemar les tecles premudes. Consta d'un array de caràcters amb la mida definida per `KEY_CBUFFER_SIZE`, que emmagatzema les dades del buffer, un índex que apunta al cap del buffer, un altre que apunta a la cua, i un contador dels elements dins del buffer.
3. **Definicions de les funcions**. Es defineixen les funcions `hw_init`, que inicialitza el hardware; `hw_getticks`, que retorna els ticks de rellotge del sistema i `hw_readkey`, que llegeix l'últim element de la cua del buffer circular.

1.2.2 `hardware.c`

Aquest fitxer C conté les funcions definides en el header, apart de les rutines de les interrupcions. Es compon dels següents elements:

1. **Definicions de variables i estructures**. Es defineix la variable `clock_ticks`, que porta el compte dels ticks de rellotge de sistema. També es defineix un buffer circular que servirà per emmagatzemar les tecles llegides.
2. **Funció** `hw_init`. Funció definida al header. Inicialitza el buffer circular i els ticks de rellotge a 0.
3. **Funció** `hw_readkey`. Aquesta funció llegeix una tecla del buffer circular si hi ha tecles disponibles: si si el comptador del buffer és major que 0, es llegeix una tecla del buffer, s'actualitza la cua i es decrementa el comptadorr. Retorna la tecla llegida o 0 si el buffer està buit.
4. **Rutina** `timer_routine`. Aquesta funció incrementa el comptador de ticks del rellotge i crida la funció `sched_run` per gestionar la planificació de tasques.
5. **Rutina** `keyboard_routine`. Aquesta funció gestiona les interrupcions del teclat: si el buffer està ple, la funció retorna sense fer res. Llegeix una tecla del port d'entrada 15 i la guarda en el buffer circular. Actualitza l'índex del cap i incrementa el comptador.

1.3 Libc

La llibreria del SO són els fitxers `libc`, que contenen funcions i constants d'utilitat per a tots els altres components.

1.3.1 libc.h

Aquest header conté definicions de funcions, constants i tipus:

1. **Definicions de constants i funcions amb la directiva `#DEFINE`**. Es defineix els codis de les crides a sistema del fork, del getpid, getticks i readkey. També es defineixen les funcions d'aquestes crides a sistema. Es defineixen altres constants d'utilitat (per exemple, una funció per calcular el nombre d'elements d'una array).
2. **Definicions de tipus**. Es defineixen tipus amb la directiva `typedef` per declarar tipus de dades compatibles amb l'arquitectura de 16 bits del processador.
3. **Definicions de les funcions**. Es defineixen les funcions `memcpy`, que copia bytes d'una adreça de memòria a una altra; `memset`, que omple un nombre de bytes d'un punter amb un valor determinat i `__mulsi3`, que és una funció per a la multiplicació d'enters.
4. **Funció inline `_syscall`** .Aquesta funció inline realitza una crida de sistema. Utilitza codi d'assembler per configurar el registre amb el número del servei que es passa com a paràmetre, realitzar la crida (calls r0) i obtenir el resultat.

1.3.2 libc.c

Aquest fitxer C conté les funcions `memcpy`, `memset`, `__mulsi3`, abans descrites.

1.4 List

És l'estructura bàsica la qual estan formades les cues ready i free, i són àmpliament utilitzades en entorns Linux per la gestió de processos. Aquesta estructura representa un node de la llista doblement enllaçada, que conté dos punters: un al següent node (next) i un a l'anterior (prev). Les seves funcions són conegudes i diverses. Per tant, podem ometre una explicació detallada. Així i tot, cal destacar la funció `list_entry`, el qual ens permet obtenir un punter de l'estructura que conté una llista. És gràcies a aquesta funció que podem obtenir l'struct d'una tasca que està en qualsevol cua de l'SO.

1.5 Macros

Les macros estan definides en el fitxer `macros.s`. Aquest fitxer conté diverses macros utilitzades per facilitar la programació en assemblador SISA. Les macros proporcionen funcionalitats per manipular la pila, gestionar trucades de subrutines, recuperar valors de taules, guardar i restaurar el context del processador, i realitzar comparacions que falten en el conjunt d'instruccions de SISA.

1.6 User

El codi d'usuari és bastant simple, però ens serveix per demostrar la funcionalitat de l'SO. Es compon de 3 fitxers: `user.c`, `user.ld`, `user-entry.s`.

1.6.1 user.c

Aquest fitxer de codi C defineix la funció principal (main) d'un programa que crea un nou procés utilitzant la crida al sistema fork prèviament explicada. El programa bifurca l'execució en dos camins: un per al procés fill i un altre per al procés pare. Cadascun dels processos executa un dels codis proporcionats pel professorat de l'assignatura PEC: `corre_letras.c` i `fibonacci.c`. Quan acaba el programa, l'SO s'encarrega d'executar la tasca idle.

1.6.2 user-entry.s

Aquest codi assemblador inicialitza l'execució del sistema establint l'stack de l'usuari i després cridant la funció main. Després d'això, entra en un bucle infinit.

1.6.3 user.ld

Aquest fitxer és un linker script que especifica més detalladament com s'ha de distribuir el codi i les dades d'usuari en la memòria quan es genera el fitxer executable:

1. `.text`. Aquesta secció comença a `0x1000` i conté el codi executable i les dades de només lectura.
2. `.data`. Inclou les dades del programa, seguides per una reserva per a l'stack de 512 bytes.
3. `_user_stack`: Defineix el punt inicial de l'stack de l'usuari després de la secció `.data`.

1.7 Fluxe de l'SO

A continuació, s'expliquen els passos que realitza l'SO quan el processador s'encén:

1. **Entrada al kernel**. Com ja sabem, el proecssador comença a l'adreça `0xC000`, on li hem ordenat al linker que posi el codi de kernel. Per tant, accedirà al kernel, on inicialitzarà el hardware, el planificador i entrarà en mode usuari mitjançant la tasca 1.
2. **Mode usuari**. S'inicialitza la pila d'usuari i es crida la funció main del fitxer `user.c`.
3. **Crida a sistema fork**. Aquí la tasca 1 fa una crida a sistema perquè vol fer un fork. El processador entra en mode sistema i entra en el gestor GSR, que guarda el context del procés i determina que cal anar al gestor de crides de sistema. Aquest últim consulta la taula de crides de sistema mitjançant el número de servei i accedeix a la funció fork del sistema. En aquesta funció, es genera una nova tasca tal com s'ha explicat i s'afegeix a la cua ready. Es recupera el context de la tasca 1 i es torna a mode usuari.
4. **Finalització del codi d'usuari**. En mode usuari tenim el codi executant-se amb dues tasques que es van intercanviant llocs en l'estat d'execució pel fet que tenen un quàntum limitat, ja que el planificador canvia la tasca cada cop que s'exhaureix el quàntum de la tasca actual, com ja s'ha esmentat. Per tant, tenim dos processos executant intercaladament el codi d'usuari fins que, finalment, retornen.
5. **Bucle infinit**. Aquests dos processos entren en un bucle infinit, i així romandràn fins que s'apagui el processador.

Cal esmentar que, en aquest procés hem rebut múltiples interrupcions de rellotge, que fa entrar al processador en mode sistema per tal que executi la rutina de rellotge (el procés és semblant a la crida a sistema). Aquesta rutina és el que invoca el planificador que fa decrementar el quàntum del procés actual.

2

Unitat vectorial

2.1 Etapes de desenvolupament

Per al desenvolupament de la unitat vectorial hem dividit la feina en 4 etapes:

1. Banc de registres.
2. Instruccions *move*.
3. ALU.
4. Memòria.

Hem decidit finalment fer aquestes etapes en aquest ordre per poder paral·lelitzar millor la feina entre els dos companys que estem fent la unitat vectorial i perquè realment la memòria no és totalment necessària tenir-la des de pràcticament el principi com en la planificació original per poder tenir l'ALU.

2.2 Banc de registres

Com ja es va explicar a la planificació inicial, el banc de registres té 8 registres de 128 bits. Té tres ports de lectura (element a, b i valor actual de l'element d'escriptura) i un port d'escriptura. Compta també amb un permís d'escriptura i 3 entrades de 3 bits per dir quins registres es volen fer servir.

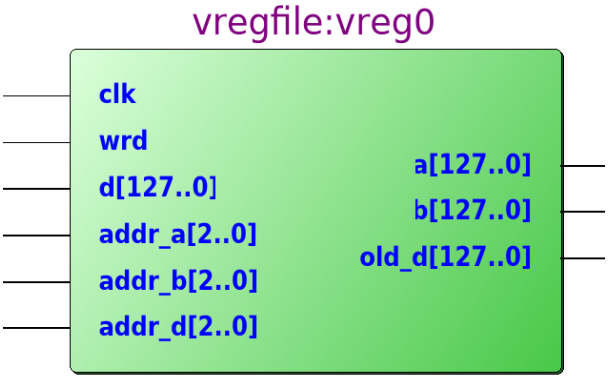


Figura 2.1: Bloc del mòdul dels registres vectorials a Quartus

Per testejar que funcioni no hem pogut fer cap test amb instruccions ja que encara no s'ha implementat res més enllà del propi banc de registres. Per tant, hem utilitzat un simple *test bench* (**??**) amb el compilador GHDL (Apèndix A).

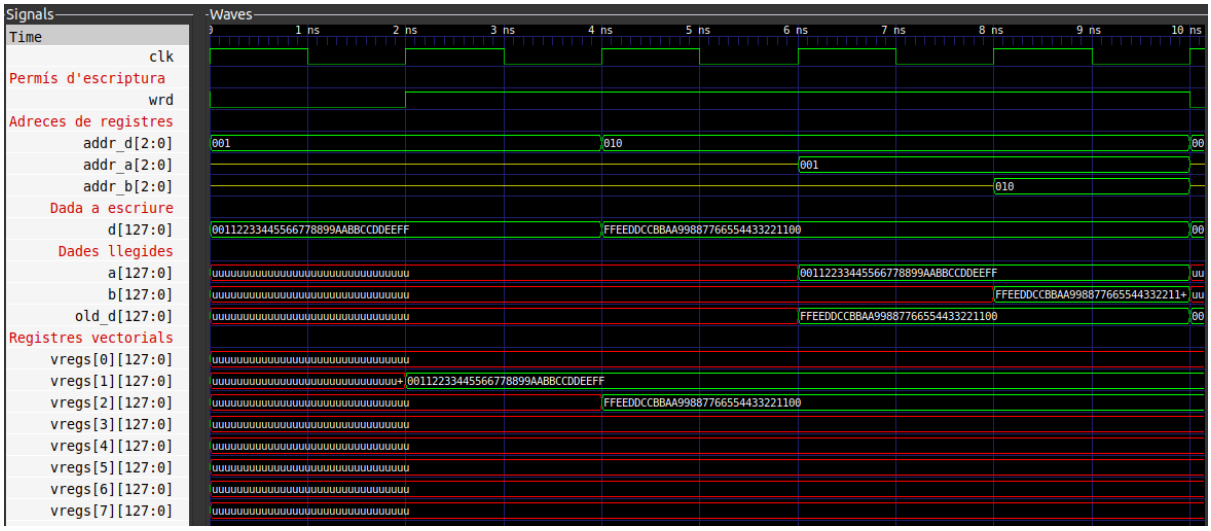
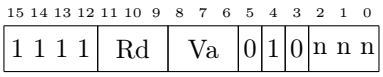


Figura 2.2: Waves generades per l'execució del *test bench* del banc de registres vectorials a Secció B.1

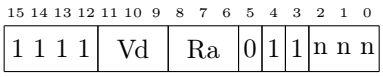
2.3 Instruccions *move*

Les instruccions *move* es mantenen com es van planificar originalment de la següent manera:

MVVR



MVRV



2.3.1 ALU vectorial (*valu*)

L'ALU vectorial compta amb dues entrades vectorials, una entrada de mida escalar, una entrada de 3 bits per l'inmediat i una entrada per el codi intern de la operació. També té una sortida vectorial, una sortida escalar i una sortida per marcar una excepció de divisió entre 0.

Actualment només fa moviment de dades. Per una part es té el moviment per passar una dada d'un registre escalar a un registre vectorial on es llegeix el valor contingut al registre vectorial de destí per un dels ports de lectura vectorial del mòdul i en base a l'inmediat es substitueix un element del vector per el valor de la entrada escalar. Per l'altra banda tenim el moviment on es treu per la sortida escalar el segment corresponent del vector que s'està llegint identificat també per l'inmediat.

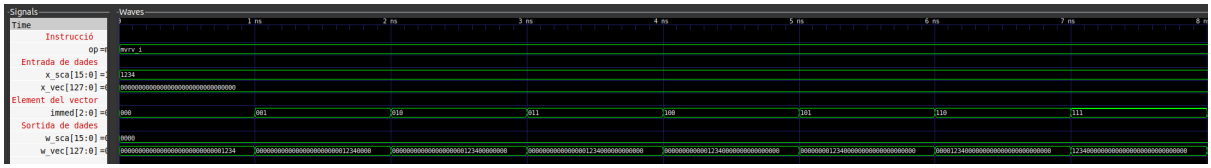


Figura 2.3: Waves generades per l'execució del *test bench* de la instrucció MVRV a Secció B.2

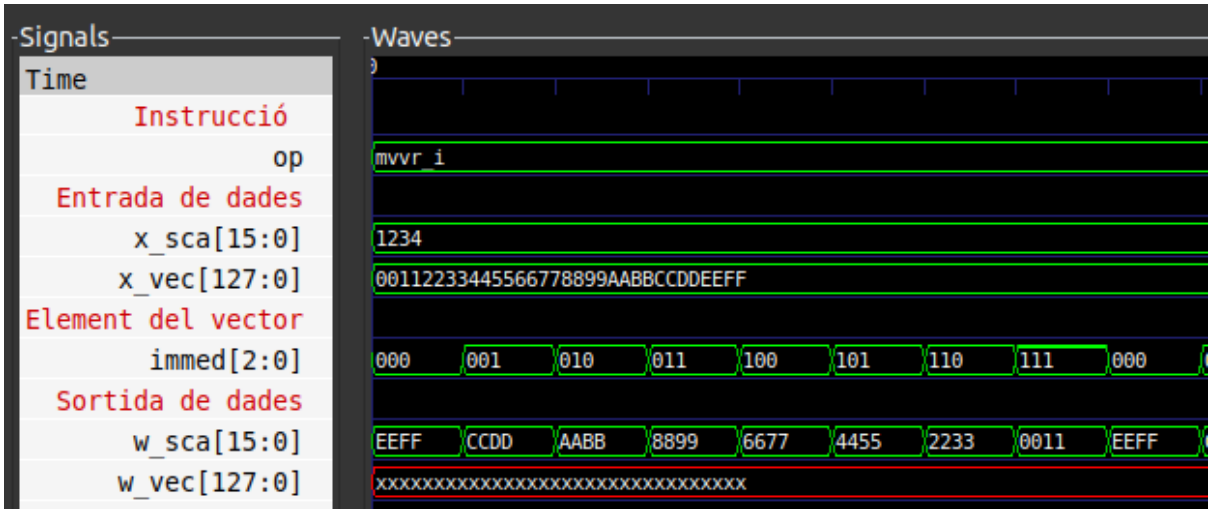


Figura 2.4: Waves generades per l'execució del *test bench* de la instrucció MVVR a Secció B.3

2.3.2 *control_l*

Per implementar aquestes instruccions, hem hagut de modificar la decodificació de la instrucció que es fa a aquest mòdul, incorporant els següents senyals:

- `vwrd`: permís d'escriptura a registre vectorial.
- `va_old_vd`: l'entrada a de l'ALU vectorial ha de tenir el contingut del registre de destí.
- `vec_produce_sca`: l'ALU vectorial produeix un resultat escalar.

Hem inclòs també dos nous senyals internes al mòdul:

- `super_special`: conté la codificació interna de la instrucció decodificada per les instruccions de memòria (per ara desctivades), els *moves* vectorials i les instruccions ja existents amb aquest *opcode*.
- `special_Rb_N`: s'utilitza per assignar que es vol llegir l'inmediat en comptes del registre Rb.

S'ha modificat l'assignació del inmediat de forma que es posen els 3 bits de menor pes de la codificació de la instrucció si l'*opcode* és el de una operació especial.

Finalment, s'ha modificat també l'assignació del permís d'escriptura escalar per que s'activi també al tenir una instrucció MVVR.

2.3.3 *multi*

En aquest mòdul únicament s'ha inclos que arribi al datapath el senyal `vwrd` quan s'està en l'estat DEMW.

2.3.4 *unidad_control*

En aquest mòdul simplement s'han connectat els senyals generats i modificats als dos mòduls que conté (*control_l* i *multi*) cap a l'exterior de la unitat de control. Es connecten, per tant, els senyals `va_old_vd` i `vec_produce_sca` del mòdul `control_l` i el senyal `vwrd` del mòdul *multi*.

2.3.5 *datapath*

Al datapath hem inclòs els senyals noves que s'han creat a la unitat de control ja que totes tenen a veure amb l'execució directament de les instruccions al datapath.

Aquí s'han connectat el banc de registres vectorial i l'ALU vectorial.

Al banc de registres per facilitar les coses s'estan reutilitzant els senyals d'adreça dels registres escalar per les adreces vectorials.

A l'ALU per l'entrada `x` entra o el vector `a` o el valor actual del registre destí en base al senyal `va_old_vd`.

També es modifica el senyal que entra al banc de registres escalar con valor a escriure per escollir entre el valor produit a la ALU escalar o a la vectorial en base al senyal `vec_produce_sca`.

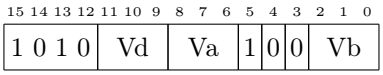
2.3.6 *proc*

Finalment, al mòdul que conté tot el processador es connecten els senyals que s'han comentat abans a la Subsecció 2.3.4 al datapath.

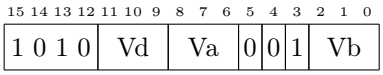
2.4 Instruccions aritmètiques

Hem decidit canviar la proposta inicial ja que el mòdul de divisió pot ser molt costòs en àrea i no és tan necessari com poden ser els shifts. A més també hem vist el problema de que les multiplicacions tal i com estava dissenyat a la proposta només podien tenir la part baixa del resultat. D'aquesta manera les instruccions queden així:

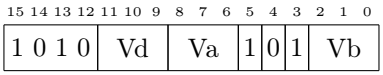
ADDV



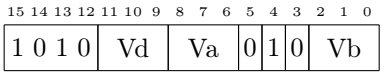
MULV



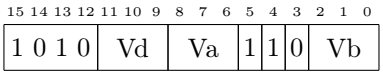
SUBV



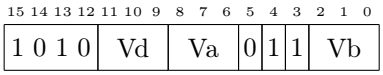
MULHV



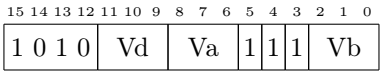
SHAV



MULHUV



SHLV



2.4.1 *valu*, *addsub*, *mul* i *shift*

Al mòdul de l'ALU vectorial s'han instanciat 8 vegades els mòduls de la ALU escalar excepte el mòdul de divisió i el de comparació.

A cada una d'aquests mòduls instanciats se'ls envia com a dades d'entrada i de sortida un element del vector, de forma que tots operin de forma independent al no tenir la capacitat de fer cálculs amb elements de diferent mides per limitacions de la quantitat d'instruccions disponibles al voler fer també una FPU.

A més perque funcionin aquestes noves instruccions s'han modificat els mòduls interns de l'ALU per incoporar les instruccions vectorials a més de les instruccions escalars.

2.4.2 *control_l*

No s'han hagut de fer molts canvis a tot el que és el control ja que moltes coses que s'havien d'activar ja es van activar a la Secció 2.3.

S'ha generat un nou senyal intern per totes les instruccions amb el opcode que originalment era únicament per les instruccions *jump* per poder passar l'instrucció que s'està executant a la ALU. A més

també s'ha modificat l'assignació de `addr_b` perque originalment les instruccions *jump* el tenen mapejat als bits $11-9$ i per les instruccions aritmètiques estan a $2-0$. Finalment s'ha activat també el permís d'escriptura quan es té el opcode d'aquestes instruccions a menys que sigui de salt.

2.4.3 Tests

Per poder testejar la alu vectorial hem decidit fer tests en assembly per totes les instruccions aritmètiques i, al no comptar al moment de fer-les amb les instruccions de memòria, s'han fet amb *moves* amb els registres escalars que ja es van testejar correctament a Secció B.2 i Secció B.3 i fent l'escriptura dels resultats amb instruccions `ST`.

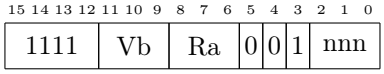
Per poder ensablar fàcilment sense haver de possar totes les instruccions noves per la SIMD hem creat una eina[1] que fa un preprocessament de les instruccions noves per convertir-les directament en el codi hexadecimal de la instrucció ja ensamblada, ho passa per l'ensamblador, i genera els `.hex` per les plaques i el `.rom` per poder simular.

Després a mà hem creat els `.assertions` per poder simular fàcilment amb l'eina de testing[3][1].

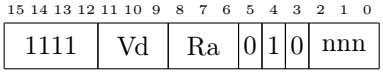
Tots els tests es poden trobar a la Secció C.1 a més de a la carpeta corresponent de la entrega amb els jocs de proves.

2.5 Instrucció `STV` i `LDV` (no implementades [2])

STV



LDV



2.5.1 Controlador de memòria

Per tal d'ajustar-se als temps marcats pel controlador de *SRAM* escriure o llegir a memòria es triguen més d'un cicle, per aquest motiu s'ha afegit l'estat *M*.

En aquest nou estat, quan es vol escriure, es segmenta l'escriptura en 8 parts de 16 bits. Fent-ho d'aquesta manera ens podem cenyir als temps del controlador de *SRAM*.

En el cas de la lectura, es va carregant el registre vectorial 16 bits en cada dos cicles del controlador, també es va decrementant l'adreça per anar llegint de la mateixa manera que s'ha escrit.

Quan s'acaba de l'operació el controlador activa la senyal *done* per indicar que ja ha acabat.

2.5.2 *control_l*

Per implementar aquestes instruccions, hem hagut de modificar la decodificació de la instrucció que es fa a aquest mòdul, incorporant els següents senyals:

- `vec`: indica que s'està executant una instrucció vectorial.

S'ha afegit que en les instruccions de ldv també s'activi el permís d'escriptura al banc de registres vectorial.

2.5.3 *multi*

S'ha afegit l'estat *M* per realitzar operacions de memòria que no es poden realitzar en un cicle. Un cop en aquest estat fins que no es rep el *done* del controlador de memòria no es torna a passar a l'estat de *F* per tornar a executar instruccions.

[1] Aquesta eina només ha funcionat tenint només la SIMD al processador. Al incorporar també la FPU han hagut errors a la simulació que fa amb ModelSim que no deixava executar-los i una vegada unificat tot hem hagut de executar els tests directament al ModelSim de manera gràfica i comprovar a mà que el contingut de memòria fos el correcte.

[2] No hem pogut implentar aquestes instruccions per falta de temps. L'equip encarregat de l'extensió vectorial hem tingut problemes en compaginar la feina que vam comenençar a principis de mes amb les entregues de l'universitat. Tot i així adjuntem el codi del controlador de memòria (Apèndix D).

2.5.4 *datapath*

S'han afegit els senyals necessaris per poder fer arribar les dades de la memòria al banc de registres o viceversa. Les senyals utilitzades en aquest mòdul s'han comentat a la Subsecció 2.5.2.

3

Coma flotant

Per implementar les instruccions de coma flotant que ofereix l'arquitectura SISA, ens ha fet falta implementar una unitat que s'encarregui dels càlculs(FPU) i un banc de registres exclusiu per coma flotant. Els móduls de la FPU(bf16_unit) i del banc de registres(regfile_fpu) han estat instanciats al datapath.

3.1 FPU

Per a l'implementació de la FPU, hem reutilitzat una unitat de coma flotant que hem trobat a un repositori de Github (`https://github.com/mfkiwl/Bfloat16-Floating-Point-Arithmetic-Unit`) i l'hem adaptat al nostre processador base. Aquesta unitat disposa dels mòduls per realitzar sumes, restes, multiplicacions i divisions, i hi hem afegit les modificacions necessaries per poder realitzar també operacions de comparació.

3.1.1 Comparacions

Per implementar les operacions de comparació, primer vam modificar el módul de sumes i restes perquè quan l'operació fos una comparació realitzes una resta. Posteriorment, l'únic que quedava fer era modificar el mòdul de la FPU que s'encarrega de seleccionar el resultat per tal de que, depenent del resultat obtingut a la resta, proporcionés el resultat corresponent.

Afegint-hi les següents senyals

- `less`: indica si la resta és negativa.
- `eq`: indica si la resta és zero.

podem obtenir el resultat de la comparació pertinent:

- `CMPLTF`: resultat = less.
- `CMPLEF`: resultat = (less || eq).
- `CMPEQF`: resultat = eq.

3.1.2 Nous estats

Com totes les operacions realitzades amb la FPU tarden 3 cicles, hem modificat la màquina d'estats afegint-hi 3 cicles extres (FP1, FP2 i FP3) als quals s'accedeix quan a l'estat DEMW es veu que volem realitzar una operació de coma flotant.

3.2 Registres

Hem decidit implementar un banc de registres específic per coma flotant, separat del regfile que ja existia, perquè d'aquesta manera ens facilita la feina a l'hora de connectar les diferents senyals, ja que ens estalviem afegir multiplexors i senyals de control per aquest per decidir entre les dades de coma flotant

3.3 Format de coma flotant

Utilitzem el format de 16 bits que s'utilitza en el disseny en el que ens hem basat:

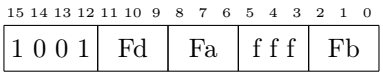
15	14 13 12 11 10 9 8 7	6 5 4 3 2 1 0
Signe	Exponent	Mantissa
1 bit	8 bits	7 bits

3.4 Instruccions

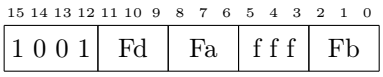
Hem decidit implementar totes les funcions de coma flotant que disposa SISA, utilitzant el seu mateix format.

3.4.1 Artimètiques

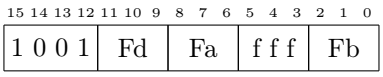
ADDF



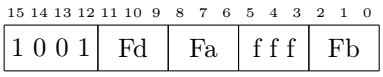
MULF



SUBF

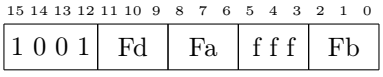


DIVF

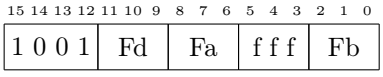


3.4.2 Comparació

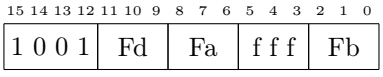
CMPLTF



CMPEQF

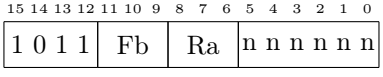


CMPLEF

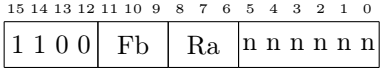


3.4.3 Accés a memòria

LDF



STF



3.5 Excepcions

Les excepcions relacionades amb coma flotant que tractarem són: overflow i divisió entre zero.

3.5.1 Overflow

Com que els mòduls que hem utilitzat ja tenien implementada la detecció d'un possible overflow, el que hem fet ha sigut afegir una senyal(of), que s'activa quan els mòduls detecten que es produeix, i que està connectada amb el mòdul encarregat de les excepcions.
Com que aquesta excepció es pot desactivar utilitzant un bit del registre S7, també afegirem una senyal que prengui el valor d'aquest bit per poder comprovar el seu valor abans de produir una exepció. Concretament, farem realitzarem una AND entre la senyal of i la senyal d'activació i el resultat d'això serà el que anirà connectat al modul d'excepcions.

3.5.2 Divisió entre zero

Aquesta excepció la comprovarem en l'estat DEMW, moment en que ja disposem dels valors dels registres però encara no hem començat l'operació.
Afegirem una senyal anomenada dp_div_zero que s'activarà quan detecti que el valor del divisor(Fb) és 0, i que estarà connectada amb el mòdul que tracta les excepcions.

3.6 Test

Per testejar la FPU com ja estava implementada i només l'hem hagut d'adaptar i no fer de zero no hem fet cap mena de *test bench* ja que ja estaven al propi repositori original. Hem fet un únic test que executa totes les instruccions de la FPU que es troba a la Secció C.2.

Apèndix A

GHDL

El compilador GHDL[2] facilita molt el treball amb VHDL i evita alguns problemes que estàvem tenint amb Quartus, com congelacions constants del programa al compilar el processador, ja que GHDL és només un compilador i podem utilitzar versions actuals per treballar i té compatibilitat nativa amb Linux, la qual cosa assegura un funcionament molt més correcte. A més, compta amb la capacitat de fer servir molt fàcilment tests bench per testejar aïlladament els diferents mòduls i també es pot utilitzar amb diferents frameworks de testing com JUnit per executar directament des de la terminal i molt ràpidament els tests.

En el nostre cas, i per agilitzar el treball, hem utilitzat un programa[3] ja desenvolupat per una companya de classe que incorpora JUnit amb GHDL per testejar el core, al qual li hem hagut de fer pràcticament cap modificació per tal que funcionés amb el nostre.

Apèndix B

Codis *test bench*

B.1 Banc de registres vectorials

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY work;

ENTITY tb_vregfile IS
END tb_vregfile;

ARCHITECTURE behav OF tb_vregfile IS
    COMPONENT vregfile
        PORT(
        clk          : IN  std_logic;
        wrd          : IN  std_logic;
        d            : IN  std_logic_vector(127 downto 0);
        addr_a       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
      addr_b       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
    addr_d       : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        a            : OUT STD_LOGIC_VECTOR(127 downto 0);
        b            : OUT STD_LOGIC_VECTOR(127 downto 0);
        old_d        : OUT STD_LOGIC_VECTOR(127 downto 0)
        );
    END COMPONENT;

    SIGNAL clk:      std_logic;
    SIGNAL wrd:      std_logic;
    SIGNAL a:        std_logic_vector(127 downto 0);
    SIGNAL b:        std_logic_vector(127 downto 0);
    SIGNAL d:        std_logic_vector(127 downto 0);
    SIGNAL old_d:    std_logic_vector(127 downto 0);
    SIGNAL addr_a:   std_logic_vector(2 downto 0);
    SIGNAL addr_b:   std_logic_vector(2 downto 0);
    SIGNAL addr_d:   std_logic_vector(2 downto 0);

BEGIN
    vreg: vregfile
        PORT MAP(
            clk => clk,
            wrd => wrd,
            a => a,
            b => b,
            d => d,
            old_d => old_d,
            addr_a => addr_a,
            addr_b => addr_b,
            addr_d => addr_d
        );

```

```
    PROCESS
    TYPE pattern_type IS RECORD
        clk:        std_logic;
        wrd:        std_logic;
        a:          std_logic_vector(127 downto 0);
        b:          std_logic_vector(127 downto 0);
        d:          std_logic_vector(127 downto 0);
        old_d:      std_logic_vector(127 downto 0);
        addr_a:     std_logic_vector(127 downto 0);
        addr_b:     std_logic_vector(127 downto 0);
        addr_d:     std_logic_vector(127 downto 0);
    END RECORD;

    BEGIN
        addr_a <= "ZZZ";
        addr_b <= "ZZZ";
        wrd <= '0';
        addr_d <= "001";
        d <= x"00112233445566778899AABBCCDDEEFF";
        clk <= '1';
        wait for 1 ns;
        clk <= '0';
        wait for 1 ns;
        clk <= '1';
        wrd <= '1';
        wait for 1 ns;
        clk <= '0';
        wait for 1 ns;
        clk <= '1';
        addr_d <= "010";
        d <= X"FFEEDDCCBBAA99887766554433221100";
        wait for 1 ns;
        clk <= '0';
        wait for 1 ns;
        clk <= '1';
        addr_a <= "001";
        wait for 1 ns;
        clk <= '0';
        wait for 1 ns;
        clk <= '1';
        addr_b <= "010";
        wait for 1 ns;
        clk <= '0';
        wait for 1 ns;

    END PROCESS;


END behav;
```

Listing B.1: *Test bench* per el banc de registres vectorial

B.2 ALU vectorial: instrucció MVRV

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY work;
USE work.renacuajo_pkg.all;

ENTITY tb_valu IS
END tb_valu;


ARCHITECTURE behav OF tb_valu IS
    COMPONENT valu
        PORT (
            x_vec       : IN  STD_LOGIC_VECTOR(127 downto 0);
```

```
            x_sca       : IN  STD_LOGIC_VECTOR(15 downto 0);
            y           : IN  STD_LOGIC_VECTOR(127 downto 0);
            immed       : IN  STD_LOGIC_VECTOR(2 downto 0);
            op          : IN  INST;
            w_vec       : OUT STD_LOGIC_VECTOR(127 downto 0);
            w_sca       : OUT STD_LOGIC_VECTOR(15 downto 0);
            div_zero    : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL x_vec: std_logic_vector(127 downto 0);
    SIGNAL x_sca: std_logic_vector(15 downto 0);
    SIGNAL y: std_logic_vector(127 downto 0);
    SIGNAL immed: std_logic_vector(2 downto 0);
    SIGNAL op: INST;
    SIGNAL w_vec: std_logic_vector(127 downto 0);
    SIGNAL w_sca: std_logic_vector(15 downto 0);
    SIGNAL div_zero: std_logic;

BEGIN
    va: valu
        PORT MAP(
            x_vec => x_vec,
            x_sca => x_sca,
            y => y,
            immed => immed,
            op => op,
            w_vec => w_vec,
            w_sca => w_sca,
            div_zero => div_zero
        );

    PROCESS
    TYPE pattern_type IS RECORD
        x_vec:      std_logic_vector(127 downto 0);
        x_sca:      std_logic_vector(15 downto 0);
        y:          std_logic_vector(127 downto 0);
        immed:      std_logic_vector(2 downto 0);
        op:         INST;
        w_vec:      std_logic_vector(127 downto 0);
        w_sca:      std_logic_vector(15 downto 0);
        div_zero:   std_logic;
    END RECORD;

    BEGIN
        x_vec <= (others => '0');
        x_sca <= x"1234";
        y <= (others => 'X');
        immed <= "000";
        op <= MVRV_I;
        wait for 1 ns;

        immed <= "001";
        wait for 1 ns;

        immed <= "010";
        wait for 1 ns;

        immed <= "011";
        wait for 1 ns;

        immed <= "100";
        wait for 1 ns;

        immed <= "101";
        wait for 1 ns;

        immed <= "110";
        wait for 1 ns;

```

```
        immed <= "111";
        wait for 1 ns;
    END PROCESS;

END behav;
```

Listing B.2: *Test bench* per la instrucció MVRV

B.3 ALU vectorial: instrucció MVVR

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY work;
USE work.renacuajo_pkg.all;

ENTITY tb_valu IS
END tb_valu;


ARCHITECTURE behav OF tb_valu IS
    COMPONENT valu
        PORT (
            x_vec       : IN  STD_LOGIC_VECTOR(127 downto 0);
            x_sca       : IN  STD_LOGIC_VECTOR(15 downto 0);
            y           : IN  STD_LOGIC_VECTOR(127 downto 0);
            immed       : IN  STD_LOGIC_VECTOR(2 downto 0);
            op          : IN  INST;
            w_vec       : OUT STD_LOGIC_VECTOR(127 downto 0);
            w_sca       : OUT STD_LOGIC_VECTOR(15 downto 0);
            div_zero    : OUT STD_LOGIC
        );
    END COMPONENT;

    SIGNAL x_vec: std_logic_vector(127 downto 0);
    SIGNAL x_sca: std_logic_vector(15 downto 0);
    SIGNAL y: std_logic_vector(127 downto 0);
    SIGNAL immed: std_logic_vector(2 downto 0);
    SIGNAL op: INST;
    SIGNAL w_vec: std_logic_vector(127 downto 0);
    SIGNAL w_sca: std_logic_vector(15 downto 0);
    SIGNAL div_zero: std_logic;

BEGIN
    va: valu
        PORT MAP(
            x_vec => x_vec,
            x_sca => x_sca,
            y => y,
            immed => immed,
            op => op,
            w_vec => w_vec,
            w_sca => w_sca,
            div_zero => div_zero
        );

    PROCESS
    TYPE pattern_type IS RECORD
        x_vec:      std_logic_vector(127 downto 0);
        x_sca:      std_logic_vector(15 downto 0);
        y:          std_logic_vector(127 downto 0);
        immed:      std_logic_vector(2 downto 0);
        op:         INST;
        w_vec:      std_logic_vector(127 downto 0);
        w_sca:      std_logic_vector(15 downto 0);
        div_zero:   std_logic;
    END RECORD;

```

```
60      BEGIN
61          x_vec <= x"00112233445566778899AABBCCDDEEFF";
62          x_sca <= x"1234";
63          y <= (others => 'X');
64          immed <= "000";
65          op <= MVVR_I;
66          wait for 1 ns;
67
68          immed <= "001";
69          wait for 1 ns;
70
71          immed <= "010";
72          wait for 1 ns;
73
74          immed <= "011";
75          wait for 1 ns;
76
77          immed <= "100";
78          wait for 1 ns;
79
80          immed <= "101";
81          wait for 1 ns;
82
83          immed <= "110";
84          wait for 1 ns;
85
86          immed <= "111";
87          wait for 1 ns;
88      END PROCESS;
89
90  END behav;
```

Listing B.3: *Test bench* per la instrucció MVVR

Apèndix C

Tests assembly

C.1 Extensió SIMD

C.1.1 Test `ADDV`

Assembly

```
.text
movi r0, 0x00
movhi r0, 0x11
mvrv v0, r0, 0

movi r0, 0x22
movhi r0, 0x33
mvrv v0, r0, 1

movi r0, 0x44
movhi r0, 0x55
mvrv v0, r0, 2

movi r0, 0x66
movhi r0, 0x77
mvrv v0, r0, 3

movi r0, 0x88
movhi r0, 0x99
mvrv v0, r0, 4

movi r0, 0xaa
movhi r0, 0xbb
mvrv v0, r0, 5

movi r0, 0xcc
movhi r0, 0xdd
mvrv v0, r0, 6

movi r0, 0xee
movhi r0, 0xff
mvrv v0, r0, 7

movi r0, 0x00
movhi r0, 0x11
mvrv v1, r0, 0

movi r0, 0x22
movhi r0, 0x33
mvrv v1, r0, 1

movi r0, 0x44
movhi r0, 0x55
mvrv v1, r0, 2

movi r0, 0x66
movhi r0, 0x77
mvrv v1, r0, 3

movi r0, 0x88
movhi r0, 0x99
mvrv v1, r0, 4

movi r0, 0xaa
movhi r0, 0xbb
mvrv v1, r0, 5

movi r0, 0xcc
movhi r0, 0xdd
mvrv v1, r0, 6

movi r0, 0xee
movhi r0, 0xff
mvrv v1, r0, 7

addv v2, v0, v1

movi r1, 0x00
movhi r1, 0x00

mvvr r0, v2, 0
st 0(r1), r0

mvvr r0, v2, 1
st 2(r1), r0

mvvr r0, v2, 2
st 4(r1), r0

mvvr r0, v2, 3
st 6(r1), r0

mvvr r0, v2, 4
st 8(r1), r0

mvvr r0, v2, 5
st 10(r1), r0

mvvr r0, v2, 6
st 12(r1), r0

mvvr r0, v2, 7
st 14(r1), r0

halt
```

Listing C.1: Codi assembly del test per l'instrucció `ADDV`

Assertions

```
Mem[01:00] = 2200
Mem[03:02] = 6644
Mem[05:04] = aa88
Mem[07:06] = eecc
Mem[09:08] = 3310
Mem[0b:0a] = 7754
```

```
Mem[0d:0c] = bb98
Mem[0f:0e] = ffdc
```

Listing C.2: Assertions del test per l'instrucció `ADDV`

C.1.2 Test `SUBV`

Assembly

```
.text
movi r0, 0x00
movhi r0, 0x11
mvrv v0, r0, 0

movi r0, 0x22
movhi r0, 0x33
mvrv v0, r0, 1

movi r0, 0x44
movhi r0, 0x55
mvrv v0, r0, 2

movi r0, 0x66
movhi r0, 0x77
mvrv v0, r0, 3

movi r0, 0x88
movhi r0, 0x99
mvrv v0, r0, 4

movi r0, 0xaa
movhi r0, 0xbb
mvrv v0, r0, 5

movi r0, 0xcc
movhi r0, 0xdd
mvrv v0, r0, 6

movi r0, 0xee
movhi r0, 0xff
mvrv v0, r0, 7

movi r0, 0x00
movhi r0, 0x11
mvrv v1, r0, 0

movi r0, 0x22
movhi r0, 0x33
mvrv v1, r0, 1

movi r0, 0x44
movhi r0, 0x55
mvrv v1, r0, 2

movi r0, 0x66
movhi r0, 0x77
mvrv v1, r0, 3

movi r0, 0x88
movhi r0, 0x99
mvrv v1, r0, 4

movi r0, 0xaa
movhi r0, 0xbb
mvrv v1, r0, 5

movi r0, 0xcc
movhi r0, 0xdd
mvrv v1, r0, 6

movi r0, 0xee
movhi r0, 0xff
mvrv v1, r0, 7

subv v2, v0, v1

movi r1, 0x00
movhi r1, 0x00

mvvr r0, v2, 0
st 0(r1), r0

mvvr r0, v2, 1
st 2(r1), r0

mvvr r0, v2, 2
st 4(r1), r0

mvvr r0, v2, 3
st 6(r1), r0

mvvr r0, v2, 4
st 8(r1), r0

mvvr r0, v2, 5
st 10(r1), r0

mvvr r0, v2, 6
st 12(r1), r0

mvvr r0, v2, 7
st 14(r1), r0

halt
```

Listing C.3: Codi assembly del test per l'instrucció `SUBV`

Assertions

```
Mem[01:00] = 0000
Mem[03:02] = 0000
Mem[05:04] = 0000
Mem[07:06] = 0000
Mem[09:08] = 0000
Mem[0b:0a] = 0000
Mem[0d:0c] = 0000
Mem[0f:0e] = 0000
```

Listing C.4: Assertions del test per l'instrucció `SUBV`

C.1.3 Test multiplicacions vectorials

Assembly

```
.text
movi r0, 0x00
movhi r0, 0x11
mvrv v0, r0, 0

movi r0, 0x22
movhi r0, 0x33
mvrv v0, r0, 1

movi r0, 0x44
movhi r0, 0x55
mvrv v0, r0, 2

movi r0, 0x66
movhi r0, 0x77
mvrv v0, r0, 3

movi r0, 0x88
movhi r0, 0x99
mvrv v0, r0, 4
```

```
movi r0, 0xaa
movhi r0, 0xbb
mvrv v0, r0, 5

movi r0, 0xcc
movhi r0, 0xdd
mvrv v0, r0, 6

movi r0, 0xee
movhi r0, 0xff
mvrv v0, r0, 7

movi r0, 0xff
movhi r0, 0xee
mvrv v1, r0, 0

movi r0, 0xdd
movhi r0, 0xcc
mvrv v1, r0, 1

movi r0, 0xbb
movhi r0, 0xaa
mvrv v1, r0, 2

movi r0, 0x99
movhi r0, 0x88
mvrv v1, r0, 3

movi r0, 0x77
movhi r0, 0x66
mvrv v1, r0, 4

movi r0, 0x55
movhi r0, 0x44
mvrv v1, r0, 5

movi r0, 0x33
movhi r0, 0x22
mvrv v1, r0, 6

movi r0, 0x11
movhi r0, 0x00
mvrv v1, r0, 7

mulv v2, v0, v1
mulhv v3, v0, v1
mulhuv v4, v0, v1

movi r1, 0x00
movhi r1, 0x00

mvvr r0, v2, 0
st 0(r1), r0

mvvr r0, v2, 1
st 2(r1), r0

mvvr r0, v2, 2
st 4(r1), r0

mvvr r0, v2, 3
st 6(r1), r0

mvvr r0, v2, 4
st 8(r1), r0

mvvr r0, v2, 5
st 10(r1), r0

mvvr r0, v2, 6
st 12(r1), r0

mvvr r0, v2, 7
st 14(r1), r0

addi r1, r1, 16

mvvr r0, v3, 0
st 0(r1), r0

mvvr r0, v3, 1
st 2(r1), r0

mvvr r0, v3, 2
st 4(r1), r0

mvvr r0, v3, 3
st 6(r1), r0

mvvr r0, v3, 4
st 8(r1), r0

mvvr r0, v3, 5
st 10(r1), r0

mvvr r0, v3, 6
st 12(r1), r0

mvvr r0, v3, 7
st 14(r1), r0

addi r1, r1, 16

mvvr r0, v4, 0
st 0(r1), r0

mvvr r0, v4, 1
st 2(r1), r0

mvvr r0, v4, 2
st 4(r1), r0

mvvr r0, v4, 3
st 6(r1), r0

mvvr r0, v4, 4
st 8(r1), r0

mvvr r0, v4, 5
st 10(r1), r0

mvvr r0, v4, 6
st 12(r1), r0

mvvr r0, v4, 7
st 14(r1), r0

halt
```

Listing C.5: Codi assembly del test per multiplicacions vectorials

Assertions

```
Mem[01:00] = ef00
Mem[03:02] = 3c5a
Mem[05:04] = 70ac
Mem[07:06] = 8bf6
Mem[09:08] = 8e38
Mem[0b:0a] = 7772
Mem[0d:0c] = 47a4
Mem[0f:0e] = fece
Mem[11:10] = fede
Mem[13:12] = f5c9
Mem[15:14] = e399
Mem[17:16] = c84f
Mem[19:18] = d6fc
Mem[1b:1a] = edc2
Mem[1d:1c] = fb6e
Mem[1f:1e] = ffff
Mem[21:20] = 0fde
Mem[23:22] = 28eb
Mem[25:24] = 38dd
Mem[27:26] = 3fb5
```

```
Mem[29:28] = 3d73
Mem[2b:2a] = 3217
Mem[2d:2c] = 1da1
Mem[2f:2e] = 0010
```

Listing C.6: Assertions del test per multiplicacions vectorials

C.1.4 Test shifts vectorials

Assembly

```
.text
movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 0

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 1

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 2

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 3

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 4

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 5

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 6

movi r0, 0xFF
movhi r0, 0xFF
mvrv v0, r0, 7

movi r0, 0
mvrv v1, r0, 0

movi r0, 3
mvrv v1, r0, 1

movi r0, 8
mvrv v1, r0, 2

movi r0, 15
mvrv v1, r0, 3

movi r0, 15
mvrv v1, r0, 4

movi r0, 16 ; -16
mvrv v1, r0, 5

movi r0, 20 ; -12
mvrv v1, r0, 6

movi r0, 31 ; -1
mvrv v1, r0, 7

shav v2, v0, v1
shlv v3, v0, v1

movi r1, 0x00
movhi r1, 0x00

mvvr r0, v2, 0
st 0(r1), r0

mvvr r0, v2, 1
st 2(r1), r0

mvvr r0, v2, 2
st 4(r1), r0

mvvr r0, v2, 3
st 6(r1), r0

mvvr r0, v2, 4
st 8(r1), r0

mvvr r0, v2, 5
st 10(r1), r0

mvvr r0, v2, 6
st 12(r1), r0

mvvr r0, v2, 7
st 14(r1), r0

addi r1, r1, 16

mvvr r0, v3, 0
st 0(r1), r0

mvvr r0, v3, 1
st 2(r1), r0

mvvr r0, v3, 2
st 4(r1), r0

mvvr r0, v3, 3
st 6(r1), r0

mvvr r0, v3, 4
st 8(r1), r0

mvvr r0, v3, 5
st 10(r1), r0

mvvr r0, v3, 6
st 12(r1), r0

mvvr r0, v3, 7
st 14(r1), r0

halt
```

Listing C.7: Codi assembly del test per shifts vectorials

Assertions

```
Mem[01:00] = FFFF
Mem[03:02] = FFF8
Mem[05:04] = FF00
Mem[07:06] = 8000
Mem[09:08] = 8000
Mem[0b:0a] = FFFF
Mem[0d:0c] = FFFF
Mem[0f:0e] = FFFF
Mem[11:10] = FFFF
Mem[13:12] = FFF8
Mem[15:14] = FF00
Mem[17:16] = 8000
Mem[19:18] = 8000
Mem[1b:1a] = 0000
```

```
Mem[1d:1c] = 000F
Mem[1f:1e] = 7FFF
```

Listing C.8: Assertions del test per shifts vectorials

C.2 Extensió FPU

C.2.1 Test global `FPU`

Assembly

```
.macro $movei p1 imm16

movi    \p1, lo(\imm16)

movhi   \p1, hi(\imm16)

.endm




.text

$MOVEI r1, 0xD0D0  ;-6

$MOVEI r2, 0x0002

st 0(r2), r1

ldf f1, 0(r2)

$MOVEI r1, 0xBE80  ;-0.25

$MOVEI r2, 0x0004

st 0(r2), r1

ldf f2, 0(r2)

addf f0, f1, f2

stf 2(r2), f0

subf f0, f1, f2

stf 4(r2), f0

mulf f0, f1, f2

stf 6(r2), f0

divf f0, f1, f2

stf 8(r2), f0

cmpltf r1, f1, f2

st 0xA(r2), r1

cmplef r1, f1, f2

st 0xC(r2), r1

cmpeqf r1, f1, f2

st 0xE(r2), r1

halt
```

Listing C.9: Codi assembly del test per la FPU

Assertions

```
Mem[03:02] = d0d0
Mem[05:04] = be80
Mem[07:06] = d0d0
Mem[09:08] = d0d0
Mem[0b:0a] = 4fd0
Mem[0d:0c] = 51d0
Mem[0f:0e] = 0001
Mem[11:10] = 0001
Mem[13:12] = 0000
```

Listing C.10: Assertions del test per la FPU

Apèndix D

Codi del controlador de memòria

L'idea del controlador és fer varis accessos per anar guardant en trossets de 16 bits a la memòria o llegir de memòria amb la mateixa idea.

Durant la lectura o l'escriptura el processador entra en l'estat M i no segueix executant instruccions. Quan s'acaba de fer la lectura o escriptura, s'activa el senyal `done` informant a la unitat de control que ja ha acabat i que ja pot seguir executant instruccions.

```
addr_s <= addr when vec = '0' else
            addr_s1;

  addr_s1 <= addr_rd when vec = '1' and we = '0' else
          addr_wr;

  wr_data_s <= wr_data when vec = '0' else
           wr_vec;

  -- control estat store
  PROCESS (CLOCK_50)
  BEGIN
    if write_s = '0' then
      wr_state <= INI;
    elsif rising_edge(CLOCK_50) then
      if vec = '1' and write_s = '1' then
        case wr_state is
          when INI =>
            wr_state <= VEC_WRITE;
          when VEC_WRITE =>
            wr_state <= VEC_WAIT;
       when VEC_WAIT =>
        wr_state <= VEC_WRITE;
          when others =>
            wr_state <= wr_state;
        END case;
      END if;
    END if;
  END PROCESS;

  PROCESS (CLOCK_50)
  BEGIN
      case wr_state is
        when INI =>
          wr_vec <= wr_vec;
       counter_wr <= x"0";
      addr_wr <= addr;
        when VEC_WRITE =>
          wr_vec <= wr_vec;
        when VEC_WAIT =>
          counter_wr <= counter_wr + 1;
          addr_wr <= addr_wr + 2;
          wr_vdata_s <= std_logic_vector(shift_left(unsigned(wr_vdata), 16));
       wr_vec <= wr_vdata_s(15 downto 0);
      END case;
```

```
46
47      wr_done <= counter_wr(3);
48
49  END PROCESS;
50
51
52
53  rd_data <= rd_data_s;
54
55  --load
56
57  PROCESS (CLOCK_50)
58  BEGIN
59    if write_s = '0' then
60      rd_state <= INI;
61    elsif rising_edge(CLOCK_50) then
62      if vec = '1' and write_s = '0' then
63        case rd_state is
64          when INI =>
65            rd_state <= VEC_READ;
66          when VEC_READ =>
67            rd_state <= VEC_WAIT;
68       when VEC_WAIT =>
69        rd_state <= VEC_READ;
70          when others =>
71            rd_state <= rd_state;
72        END case;
73      END if;
74    END if;
75  END PROCESS;
76
77  PROCESS (CLOCK_50)
78  BEGIN
79      case rd_state is
80        when INI =>
81          rd_vec_s <= rd_vec_s;
82       counter_rd <= x"0";
83       addr_rd <= addr + 16;
84        when VEC_READ =>
85          rd_vec_s <= rd_vec_s;
86        when VEC_WAIT =>
87          counter_rd <= counter_rd + 1;
88          addr_rd <= addr_rd - 2;
89       rd_vec_s(15 downto 0) <= rd_data_s;
90       rd_vec_s <= std_logic_vector(shift_right(unsigned(rd_vec_s), 16));
91      END case;
92
93      rd_done <= counter_rd(3);
94
95    if rd_done = '1' then
96      rd_vec <= rd_vec_s;
97    END if;
98
99  END PROCESS;
100
101 done <= rd_done or wr_done;
```

Listing D.1: Codi de les màquines d'estats.

Referències

[1] Raúl Gilabert Gámez. *SISA Custom Instruction Assembler*. 2024. URL: https://github.com/raulgilabert/SISA-Custom-Instruction-Assembler (cons. 19-06-2024).

[2] GHDL. *GHDL documentation*. 2023. URL: https://ghdl.github.io/ghdl/ (cons. 14-06-2024).

[3] Nara Díaz Viñolas. *SISA testing framework*. 2024. URL: https://github.com/rdvdev2/sisa_testing_framework (cons. 14-06-2024).