



TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DE COMPUTADORES



Simulador de RISC-V empregando SystemC

Estudiante: Hugo Mato Cancela

Dirección: Roberto Rodríguez Osorio

A Coruña, xuño de 2025.

Dedicatoria

Agradecementos

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Resumo

RISC-V é unha nova arquitectura libre para procesadores programables. Está chamada a competir con outras arquitecturas máis establecidas como ARM en moitos ámbitos tecnolóxicos. Ademais de que as especificacións de RISC-V son abertas, a principal vantaxe desta arquitectura é que cubre desde as implementacións máis sinxelas para sistemas embarcados, ata as máis potentes para cálculo científico e multimedia. Todo isto é posible mediante a especificación de moitas das características máis avanzadas como extensións ás arquitecturas básicas.

A implementación dun procesador require previamente dun modelado e simulación que garantan que o funcionamento final vai ser o correcto.

Neste traballo, pártese dun modelo da versión básica RV32I realizada en SystemC. A única extensión implementada é a multiplicación para enteiros. O obxectivo desde proxecto é modelar e simular extensións adicionais ata chegar ao coñecido como nivel G.

Abstract

RISC-V is a new open-source architecture for programmable processors. It is destined to compete with more established architectures like ARM in a lot of technological fields. In addition to the fact that the RISC-V specifications are open, the main advantage of this architecture is that it spans from the simplest implementations for embedded systems to the most powerful ones for scientific computing and multimedia. All of this is possible through the specification of many of the most advanced features as extensions to the basic architectures.

The implementation of a processor requires a previous modeling and simulation that ensures that its final operation will be correct.

This work is based on a model of the basic RV32I implemented in SystemC. The only extension implemented is integer multiplication. The objective for this project is to model and simulate additional extensions up to what is known as the G level.

Palabras chave:

- RISC-V
- Simulador
- SystemC
- Extensións

Keywords:

- RISC-V
- Simulator
- SystemC
- Extensions

Índice Xeral

1	Introdución	1
1.1	Motivación	1
1.2	Obxectivos	1
1.3	Metodoloxía	2
1.3.1	Fases principais	2
1.4	Contida da memoria	3
2	RISC-V	4
2.1	Que é RISC-V?	4
2.2	Por que é importante?	4
3	Modelado e simulación	6
3.1	Por que é importante o modelado e a simulación?	6
3.2	VHDL e Verilog	6
3.3	SystemC	7
3.4	Spike	7
4	Deseño do simulador	8
4.1	RTL	8
4.2	Pipeline de 5 etapas	8
4.3	Módulos do simulador	9
4.4	Modos de operación	9
4.5	Simulación de latencias	9
4.6	Sinais de hazard	10
5	Implementación	11
5.1	Decisións á hora de implementar	11
5.2	Instrucións implementadas	11

5.3	Implementacións dos pipelines	15
5.4	Funcionalidades do simulador	15
5.5	Instrución: tipo de dato	15
5.6	Ferramentas empregadas	15
5.6.1	Segger Embedded Studio for RISC-V	15
5.6.2	Visual Studio 2022	16
5.6.3	GTK Wave	16
5.6.4	Git	16
5.6.5	SystemC	16
6	Probas	18
6.1	Benchmarks	18
6.2	Tests propios	19
6.3	Depuración	19
7	Uso do simulador	20
8	Conclusións	25
8.1	Resultados	25
8.2	Traballo futuro	26
A	Material adicional	29
A.1	Exemplo de código de probas	29
	Relación de Acrónimos	31
	Glosario	32
	Bibliografía	33

Índice de Figuras

7.1	Opcións do proxecto	20
7.2	Elección da extensión correcta	22
7.3	Compilación do proxecto	22
7.4	Captura coas opcións de depuración de Segger	23
7.5	Cambio de parámetros en Config.h	23
7.6	Resultados tras executar o benchmark SPMV	24

Índice de Táboas

2.1	Nome e descripcións das extensións	5
5.1	Extensións e instrucións implementadas	11
6.1	Benchmarks empregados e con que fin	18
8.1	Rendemento do benchmarks SPMV segundo as latencia de distintas operacións.	27

Introdución

ESTE proxecto busca crear un simulador de RISC-V empregando a librería SystemC en C++. Ao longo desta memoria describiranse as etapas de execución, as extensións e os distintos módulos, así como a motivación destes.

REVISAR PREGUNTAS DESPOIS DE GUIA TFG

1.1 Motivación

RISC-V apunta a ser unha das arquitecturas máis empregadas nun futuro, xa que é libre, permitindo aforrar o custo de licenzas. Grazas a que se pode modificar, engadindo ou eliminando funcionalidades, isto permite que abarque múltiples sectores, dende [chips](#) máis sinxelos orientados a [Internet of Things \(IoT\)](#), ata competir con [Advanced RISC Machine \(ARM\)](#) en sistemas embebidos [1, 2]. Co nacemento da nova [Instruction Set Architecture \(ISA\)](#) debido á necesidade dun conxunto de instrucións máis sinxelo e sen custos por licenzas, nace a necesidade de crear un simulador adaptado tanto a esta [ISA](#) como á arquitectura RISC-V. Se ben xa existe un simulador [3], Spike funciona a nivel de instrución, polo que faltarían detalles como pipelines, [hazard](#) ou a posibilidade de ver como cambian os rexistros con cada ciclo.

Durante o proceso de deseño, unha parte clave é a verificación do correcto funcionamento [4, 5]. Se ben é posible crear un chip con cada versión, na práctica, debido aos longos tempos e altos prezos, é inviable. Ahí é onde un simulador toma protagonismo, xa que permite probar de forma rápida, sinxela e barata os deseños creados. Ademais, é unha ferramenta moi interesante para as investigacións da comunidade científica e incluso para entornos educativos.

1.2 Obxectivos

Os obxectivos deste proxecto son modelar e simular, usando SystemC, as seguintes extensións da arquitectura RV32I:

- multiplicación e división por números enteiros (extensión M).
- aritmética en punto flotante de simple (extensión F).
- operacións atómicas (extensión A).
- xestión de rexistros de control e estado (extensión Zicsr).
- sincronización de escritura de instrucións (extensión Zifencei).

O modelado será totalmente parametrizable, permitindo especificar a latencia das diferentes instrucións. Tamén permitirá especificar o número de canles de execución para as unidades de enteiros a punto flotante, e se estes están ou non totalmente segmentados.

Desta maneira, a simulación permitirá comparar o rendemento de, por exemplo, unha implementación na que multiplicador e divisor comparten circuítos, cunha na que ambos son independentes, e tamén comparar un divisor totalmente segmentado con un que non o sexa.

Os resultados do modelado e a simulación son dous: verificar o correcto funcionamento da arquitectura, e comprobar o seu rendemento.

1.3 Metodoloxía

O método de traballo será incremental, dividindo as tarefas en partes independentes que van ser implementadas, simuladas e verificadas por orde de complexidade antes de proceder co seguinte.

O procedemento habitual é unha reunión semanal na que se revisa o feito anteriormente, acompañado de comprobación cos tests correspondentes para esa parte. Despois, decídese cal é o seguinte paso, podendo ser a implementación dunha nova extensión ou modificar un módulo do simulador.

1.3.1 Fases principais

- Estudio da documentación existente sobre RISC-V.
- Familiarización coa implementación base de RV32I en SystemC.
- Modelado e simulación do multiplicador e divisor de enteiros.
- Modelado e simulación das extensións de punto flotante F e D.
- Modelado e simulación de extensións Zicsr e Zifencei.
- Empaquetamento do software.

1.4 Contida da memoria

Nesta sección describírase brevemente os capítulos desta memoria e o seu contido:

- **Capítulo 1: Introducción.** O primeiro capítulo inclúe unha descrición sobre o proxecto, cal foi a motivación deste, os obxectivos propostos para este traballo e a metodoloxía empregada.
- **Capítulo 2: RISC-V.** Aquí falarase sobre a arquitectura, explicando as súas características máis interesantes, as extensións e outros datos relevantes.
- **Capítulo 3: Modelación e simulación** Explicación sobre que é o modelado e a simulación, por qué son útiles e as linguaxes máis empregadas.
- **Capítulo 4: Deseño do simulador.** Neste capítulo tratarase os distintos módulos creados, o por qué e as decisións de deseño detrás destas.
- **Capítulo 5: Implementación.** Explicarase as ferramentas empregadas, como se aplicou a metodoloxía e o proceso de engadir as extensións.
- **Capítulo 6: Probas.** Neste quinto apartado detallase o procedemento para comprobar o correcto funcionamento do simulador, como se elaboraron os tests, unha breve explicación de como funcionan e os programas empregados para a depuración.
- **Capítulo 7: Uso do simulador.** Contén unhas breves indicacións de como empregar o software.
- **Capítulo 8: Conclusións.**
- **Apéndices.**
- **Bibliografía.**

Capítulo 2

RISC-V

Ao longo deste capítulo detallarase en qué consiste RISC-V, a estrutura básica, por qué é interesante e cales foron os motivos de que fose empregado como obxectivo deste proxecto.

2.1 Que é RISC-V?

Co paso do tempo, nacen novas arquitecturas buscando ofrecer algo innovador no mundo tecnolóxico. RISC-V é unha destas novidades, nacida en 2010 na Universidade de Berkeley [6], foi crescendo pouco a pouco, incluso con axuda de voluntarios fóra do ámbito académico. Os puntos fortes desta arquitectura son a súa aposta por unha [Instruction Set Architecture \(ISA\)](#) libre e modificable, permitindo eliminar ou engadir instrucións segundo cada caso. [1]. Non se trata do primeiro proxecto deste tipo, pero sí dun dos máis relevantes.

2.2 Por que é importante?

Unha nova arquitectura acompañada dunha [ISA](#) libre permite reducir custos, polo que o fai un bo candidato para ser empregado en dispositivos [IoT](#). Se ben xa existen [ISAs](#) moito máis populares e amplamente estendidas, como por exemplo a ARMv7 [7], si que existen varios motivos para crear un novo conxunto de instrucións. Un dos principais é que a maioría das xa existentes requiren de licencia para o seu uso. Ademais, é necesaria unha [ISA](#) máis sinxela de cara á implementación e a modificación.

Cada conxunto de instrucións que realizan funcionalidades básicas e que é imprescindible implementar recibe o nome de base. O habitual son as bases que traballan con enteiros de 32 ou 64 bits. Tamén determinan algunhas a codificación, tamaño de rexistros ou instrucións,... As máis típicas son:

- RV32I: Conxunto de instrucións de base enteira de 32-bits.

- RV32E: Conxunto de instrucións de base enteira (embebida, é dicir, con 16 rexistros) de 32-bits.
- RV64I: Conxunto de instrucións de base enteira de 64-bits.
- RV128I: Conxunto de instrucións de base enteira de 128-bits.

Por outra parte, as instrucións similares ou relacionadas agrúpanse habitualmente en extensións. As extensións máis habituais contan cunha versión validada [8], xa que se empregan na inmensa maioría de deseños. Cada extensión traballa sobre unha base determinada, engadindo funcionalidades adicionais, creando un deseño modular. En canto ás extensións:

Nome da extensión	Descrición
<i>M</i>	Extensión estándar para multiplicación de enteiros, divisións e resto
<i>A</i>	Extensión estándar para operación atómicas
<i>F</i>	Extensión estándar para punto flotante de precisión simple
<i>D</i>	Extensión estándar para punto flotante de precisión dobre
<i>G</i>	Abreviatura empregada para o conxunto de extensións "IMAFDZicsr_Zifencei"
<i>L</i>	Extensión estándar para punto flotante decimal
<i>P</i>	Extensión estándar para instrucións de Packed-SIMD
<i>Zicsr</i>	Extensión estándar para a xestión de rexistros de control e estado (Control, Status and Register (CSR) Instructions)
<i>Zifencei</i>	Extensión para instrucións para a sincronización de escritura de instrucións (Fetch e Fence)

Táboa 2.1: Nome e descripcións das extensións

Modelado e simulación

NESTE apartado explicaránse os fundamentos dun simulador, os motivos para crear un e o funcionamento típico. Ademais, indícaránse as linguaxes máis habituais destes casos, as diferenzas e o motivo da elección de SystemC.

3.1 Por que é importante o modelado e a simulación?

Durante o proceso de creación de calquera compoñente electrónico minimamente complexo, é necesario revisar que o deseño realiza as funcións esperadas e de forma correcta. Isto é, que garante os resultados esperados, dentro dun tempo razoable e cun emprego de recursos limitado. Unha opción é encargarse un novo chip cada vez que se crea un deseño que se necesita revisar. Se ben é posible, os longos períodos de tempo de creación e os altos custos son un impedimento enorme. No seu lugar, créase unha versión dixital mediante [software](#). Este é o modelado, mentres que se queremos que o deseño imite o comportamento real para poder ver os erros, é necesario un simulador. En moitos casos, estas ferramentas están xuntas, facendo máis sinxelo o traballo. Grazas á existencia destes programas, o deseño e creación de compoñentes electrónicos é moito máis veloz e barato, permitindo un avance tecnolóxico con menos limitacións.

3.2 VHDL e Verilog

Estas linguaxes son empregadas principalmente para describir circuítos de forma moi precisa, permitindo incluso diferenciar que é unha simple conexión dun rexistro. Ademais, os compiladores para [Hardware Description Language \(HDL\)](#) son capaces de xerar circuítos de alta calidade, imposibles de realizar para un ser humano. Á hora de modelar e simular, son as máis empregadas. Coñecidas por ser o estándar na industria, permiten traballar a baixo nivel. Isto garante unha gran eficiencia e rendemento, ademais de ofrecer flexibilidade. Da

mesma forma que a cercanía ao hardware ofrece algunhas melloras, tamén ten desvantaxes, como a maior complexidade á hora de escribir código, falta de características típicas de [Object Oriented Programming \(OOP\)](#), ...

A elección de SystemC antes que VHDL ou Verilog foi debido a que permite traballar a un nivel máis alto, a simulación é moito máis rápida e ademais, a base do proxecto inicial sobre o que se traballou xa estaba feita empregando esta librería.

3.3 SystemC

Para o modelado e a simulación, a escolla habitual, como se comentaba na sección anterior (3.2), é VHDL ou Verilog. Sen embargo, para este proxecto elixiuse SystemC. Esta [meta-linguaxe](#) creada en C++, é habitualmente empregada para codeseño. O feito de que sexa unha linguaxe de alto nivel, proporciona unha flexibilidade e sinxeleza á hora de traballar que carecen as linguaxes de máis baixo nivel. Engade a posibilidade de ter datos públicos e privados, permite organizar todo en clases, facilitando un deseño modular, gestionar eventos e modelar a varios niveis, como [Register Transfer Level \(RTL\)](#) ou [Transaction-Level Modeling \(TLM\)](#). Ademais, destacar que é moito máis rápido para a simulación que VHDL ou Verilog.

3.4 Spike

A propia organización de RISC-V xa ofrece un simulador [3], pero seguen existindo motivos para crear unha alternativa. Non simula cada ciclo, senón que funciona a [Instruction Set Simulator \(ISS\)](#). Spike é parametrizable, xa que permite cambiar o número de ciclos, núcleos, modificar a memoria, que extensións emprega, Está escrito en C/C++ polo que ofrece unha boa velocidade de simulación. Ademais, trátase dun proxecto open-source, polo que calquera pode colaborar e avanza de forma constante. Funciona coa base RV32I, RV64I, RV32E, RV64E. Tamén a gran maioría de extensións na versión v1.0, e nas últimas versións as extensións I (entero base), M (multiplicación/división), A (atómicas), F/D (punto flotante simple/dobre precisión), C (instrucións comprimidas), V (vectorial). Inclúe soporte para debug, simula diferentes niveis de privilexio e compatibilidade con binarios .elf.

Se ben é un bo simulador cunha ampla oferta de características, este proxecto busca ofrecer unha alternativa que mostre o funcionamento dun programa de forma máis precisa. Spike simula a nivel de instrución, polo que non se poden ver como cambian os valores dos rexistros con cada ciclo, hazards, pipelines ou sinais de comunicación entre módulos.

Deseño do simulador

PREVIAMENTE a crear calquera programa é necesario un deseño. Durante este capítulo explicaranse as distintas decisións tomadas ao longo do traballo, a súa motivación e alternativas. Ademais, falarase sobre características deste, como a parametrización ou o nivel de funcionamento.

4.1 RTL

O nivel de simulación é o seguinte paso despois de decidir que arquitectura modelar. Neste caso, decidiuse que o simulador traballe a [Register Transfer Level \(RTL\)](#) debido ao interese en reflexar todas as operacións, a actualización de valores nos rexistros ou non omitir a implementación da conexión dos módulos (unhas das partes máis interesantes neste traballo) [9].

Se ben unha alternativa interesante sería [Transaction-Level Modeling \(TLM\)](#), que sería o seguinte nivel de deseño electrónico, a abstracción que proporciona neste caso é demasiado alta para os detalles nos que se considera traballar neste proxecto.

Tipicamente, para este nivel tan baixo, o habitual é empregar VHDL ou Verilog. Como se comentou no capítulo 1.1 e en 3.3, SystemC foi elixido por ser máis rápido para simulación, permite traballar a máis nivel e a base do proxecto sobre a que se traballa xa estaba feita cunha linguaxe de alto nivel.

4.2 Pipeline de 5 etapas

A división do pipeline comeza co nacemento dos primeiros ordenadores segmentados en REFES AQUI. A idea é aumentar o rendemento ao permitir que o procesador execute máis dunha instrución por ciclo. Para iso, divídese a execución en 5 etapas, habitualmente Fetch, Decode, Execute, Memory e Write Back.

En Fetch obténse a instrución de memoria. Durante Decode procésase a instrución obtida, analizando que tipo de operación se realizará, cales son os rexistros empregados, se hai algunha dependencia, etc. En Execute realízase a operación determinada, como pode ser un cálculo na *Arithmetic and Logical Unit (ALU)*. En Memory, se é necesario, escríbese ou léese en memoria. Finalmente, en Write Back actualízanse os rexistros.

Ao deseñar o simulador elixiuse un pipeline de 5 etapas debido a súa sinxeleza. A aproximación realizada foi dividir cada etapa en un módulo do simulador, salvando Decod e Write Back que se uniron por comodidade.

4.3 Módulos do simulador

Os módulos principais, como se comentou no apartado anterior, son cada unha das 5 etapas, fusionando Decod e Write Back. No caso da etapa Execute, simplemente se creou unha *ALU*, encargada de realizar operacións de suma, resta e outras operacións lóxicas. Ademais, engadíronse varios módulos ao longo do proxecto. Para as operacións de multiplicación e división da extensión M, creouse un novo módulo. Separar estas funcionalidades permite organizar o traballo, ademais de simplificalo e facerlo máis sinxelo de depurar. Para a extensión F, de forma análoga, existe un compoñente encargado de realizar todas as operacións de punto flotante simple. Estes dous últimos módulos inclúen a posibilidade de parametrizar as súas instrucións.

4.4 Modos de operación

Coa fin de mellorar a calidade da simulación, decidiuse engadir no módulo de multiplicación a posibilidade de elixir entre dous modos de funcionamento. O primeiro limita de forma que, se hai unha multiplicación executándose, non se pode realizar ningunha outra operación no módulo. Isto pretende semellarse a un caso real, no que, por limitacións físicas, se empregan os mesmos circuitos para ambas operacións. O segundo modo permite que se executen todas as multiplicacións necesarias, pero só unha división ao mesmo tempo.

4.5 Simulación de latencias

Á hora de executar código, existen varios axustes que se poden cambiar para simular distintos comportamentos típicos de RISC-V. Pódese modificar a latencia das operacións do módulo de multiplicación, as cales son:

- MUL
- MULH

- MULHU
- MULHSU
- DIV
- DIVU
- REM
- REMU
- FADD.S
- FSUB.S
- FMUL.S

Isto permite unha representación máis realista, xa que por defecto todas as instrucións no simulador teñen unha latencia dun ciclo. Sen embargo, na realidade, operacións máis complexas como as multiplicacións ou divisións levan varios ciclos.

4.6 Sinais de hazard

Como sucede en todas as [arquitectura](#) segmentadas, a execución de instrucións moitas veces vese limitada por dependencias. Isto é, non se pode continuar co programa porque a seguinte instrución emprega algún rexistro que debe ser actualizado previamente, pero aínda non sucedeu porque algunha instrución previa non acabou a súa execución. Para evitar esta situación, en moitos casos engádense burbullas, ciclos nos que non se fai ningún traballo para permitir que o resto de instrucións acaben. O simulador replica este funcionamento, polo que para detectar estas dependencias emprega sinais de [hazard](#).

Chámase hazard a calquera perigo que poidese causar un risco [Read After Write \(RAW\)](#), [Write After Read \(WAR\)](#) ou [Write After Write \(WAW\)](#). Polo que, para evitar un hazard, débese detectar unha dependencia con suficiente antelación. A nosa solución elixida neste caso foi empregar sinais nos módulos de punto-flotante, multiplicación e [ALU](#) conectados co módulo de decodificación. Se foi detectada unha dependencia, o sinal enviará unha alerta ao módulo e este creará burbullas ata que non exista a dependencia.

Implementación

TRAS haber creado o deseño, é necesario realizar a implementación. Neste capítulo, trátanse os problemas afrontados, as solucións elixidas e as ferramentas empregadas.

5.1 Decisións á hora de implementar

Unha vez deseñado o proxecto, o seguinte paso é a implementación. Durante este proceso, buscaranse aproximacións a problemas que non se afrontaron na etapa de deseño. Por exemplo, para a implementación da instrución Fence, da extensión Zifencei, introducíronse sinais no módulo Decod conectadas con todos os módulos. Grazas a isto, pódese saber se había algunha instrución executándose nalgún módulo, o que permite retrasar a execución da seguinte instrución. Así, garántese que todas as instrucións acabaron, simulando a barreira.

5.2 Instrucións implementadas

Como se comentou no capítulo 1.2, neste proxecto implementáronse todas as extensións ata a G. Isto inclúe as funcionalidades da base RV32I, extensión M, F, A, Zifencei, Zicsr. Implementáronse todas as instrucións destas extensións. A continuación, unha lista das instrucións implementadas e a extensión á que pertencen:

Táboa 5.1: Extensións e instrucións implementadas

Nome da operación	Estado da implementación
Extensión M — Multiplicación e división	
<i>Mul</i>	Implementada

.....(continúa na páxina seguinte).....

Táboa 5.1 – (vén da páxina anterior)

Nome da operación	Estado da implementación
<i>Mulh</i>	Implementada
<i>Mulhsu</i>	Implementada
<i>Mulhu</i>	Implementada
<i>Div</i>	Implementada
<i>Divu</i>	Implementada
<i>Rem</i>	Implementada
<i>Remu</i>	Implementada

Extensión F – Punto flotante en simple precisión

<i>Fadd.s</i>	Implementada
<i>Fclass.s</i>	Non implementada
<i>Fcvt.l.s</i>	Non implementada
<i>Fcvt.lu.s</i>	Non implementada
<i>Fcvt.s.l</i>	Non implementada
<i>Fcvt.s.lu</i>	Non implementada
<i>Fcvt.s.w</i>	Implementada
<i>Fcvt.s.wu</i>	Implementada
<i>Fcvt.w.s</i>	Implementada
<i>Fcvt.wu.s</i>	Implementada
<i>Fdiv.s</i>	Non implementada
<i>Feq.s</i>	Non implementada
<i>Fle.s</i>	Non implementada
<i>Flt.s</i>	Non implementada

.....(continúa na páxina seguinte).....

Táboa 5.1 – (vén da páxina anterior)

Nome da operación	Estado da implementación
Flw	Implementada
$Fmadd.s$	Non implementada
$Fmax.s$	Non implementada
$Fmin.s$	Non implementada
$Fmsub.s$	Non implementada
$Fmul.s$	Implementada
$Fmv.w.x$	Implementada
$Fmv.x.w$	Implementada
$Fnmadd.s$	Non implementada
$Fnmsub.s$	Non implementada
$Fsgnj.s$	Non implementada
$Fsgnjn.s$	Non implementada
$Fsgnjx.s$	Non implementada
$Fsqrt.s$	Non implementada
$Fsub.s$	Implementada
Fsw	Implementada
$C.flw$	Non implementada
$C.flwsp$	Non implementada
$C.fsw$	Non implementada
$C.fswsp$	Non implementada
$Fmv.h.x$	Non implementada

Extensión A – Operacións atómicas

.....(continúa na páxina seguinte).....

Táboa 5.1 – (vén da páxina anterior)

Nome da operación	Estado da implementación
<i>Lr.w</i>	Non implementada
<i>Sc.w</i>	Non implementada
<i>Amoswap.w</i>	Non implementada
<i>Amoadd.w</i>	Non implementada
<i>Amoxor.w</i>	Non implementada
<i>Amoand.w</i>	Non implementada
<i>Amoor.w</i>	Non implementada
<i>Amomin.w</i>	Non implementada
<i>Amomax.w</i>	Non implementada
<i>Amominu.w</i>	Non implementada
<i>Amomaxu.w</i>	Non implementada

Extensión Zicsr — Acceso a rexistros CSR

<i>Csrrw</i>	Implementada
<i>Csrrs</i>	Implementada
<i>Csrrc</i>	Implementada
<i>Csrrwi</i>	Implementada
<i>Csrrsi</i>	Implementada
<i>Csrrci</i>	Implementada

Extensión Zifencei — Sincronización de instrucións

<i>Fence.i</i>	Implementada
----------------	--------------

5.3 Implementacións dos pipelines

Neste simulador non se implementaron as unidades funcionais que están segmentadas como tal, polo que á hora de simular o retardo das instrucións, decidiuse empregar arrays para simular o proceso dunha instrución atravesando o pipeline. Os ciclos necesarios para saír do array son a latencia, e unha vez fóra, as instrucións son procesadas. Adicionalmente, se nun ciclo a instrución que saíu é un **No Operation (NOP)**, búscase a anterior para que sexa executada.

5.4 Funcionalidades do simulador

Antes de comezar este proxecto, xa existía un estrutura base deste simulador, implementando todas as funcionalidades básicas recollidas na base RV32I. Isto inclúe todas as instrucións de lectura e escritura de datos en memoria e rexistros, suma e resta (incluso con operandos inmediatos), operacións lóxicas, saltos e ramas. Esta primeira versión podía executar programas relativamente sinxelos.

Unha vez comezado o proxecto, engadíronse o módulo de multiplicación para a extensión M, o módulo de operación de punto flotante simple para a extensión F e algunhas instrucións adicionais para as extensións A, Zicsr e Zifencei. Agora, permite a execución de multiplicacións, divisións e operacións con datos de tipo float.

5.5 Instrución: tipo de dato

Todo programa executado neste simulador está feito a base de instrucións. Por iso, a importancia de traballar correctamente - REV

5.6 Ferramentas empregadas

Durante o proxecto empregáronse 5 ferramentas: Segger, Visual Studio 2022, Git, GTK Wave e SystemC. A continuación, unha breve explicación do seu funcionamento, alternativas dispoñibles e comparativas explicando o porqué desta elección.

5.6.1 Segger Embedded Studio for RISC-V

Segger Embedded Studio for RISC-V é un IDE que permite compilar para RISC-V, incluíndo obxectivos concretos como RV32, producir arquivos .elf e ver o código ensamblador. Foi principalmente empregado á hora de escribir código en C para **test** ou **benchmark**. Ademais, o depurador permite ver código ensamblador coas direccións, polo que foi realmente útil á hora

de encontrar bugs. Se ben existen alternativas populares, como CLion de JetBrains co Toolchain de RISC-V, Visual Studio Code ou Eclipse. No caso de CLion é de pago, polo que é un gran punto en contra. Se ben a universidade ofrece claves, sería necesario engadir o toolchain de RISC-V para poder compilar código para RISC-V, facendo o proceso máis complexo. Visual Studio Code tampouco inclúe ferramentas de base, polo que sería necesario buscar plugins e configurar todo para que sexa apto. Por último, Eclipse cun plugin podería ser apto. Se ben o proceso de instalación non é complexo, non inclúe obxectivos determinados. Todo isto fai que Segger sexa a mellor alternativa, xa que inclúe configuracións xa feitas, todas as ferramentas necesarias sen apenas configuración.

5.6.2 Visual Studio 2022

Á hora de traballar no simulador con C++, o IDE elixido foi Visual Studio 2022. Entre as características máis destacables están: integración con Git, depuración con opcións avanzadas, bo funcionamento con GTK Wave e SystemC, ...Existen infinidades de alternativas, como se mencionou no apartado anterior, este foi o elixido por ser o máis habitual para este tipo de proxectos polo estudante. Ademais, xa fora empregado na asinatura de Codeseño [hardware/software](#) xunto a SystemC.

5.6.3 GTK Wave

Para solventar algúns dos problemas máis complexos, como se mencionou no capítulo REF AQUÍ, foi necesario empregar esta ferramenta. Este software permite, unha vez engadidas trazas no código, rexistrar os cambios de valor de sinais e variables para despois mostralas nun gráfico de ondas. Se ben non é moi popular, xa foi empregada nalgunha asinatura, polo que coñecela previamente foi imprescindible para elixila.

5.6.4 Git

Unha das ferramentas máis empregadas en todos os proxectos é Git. É un sistema de control de versións, polo que mediante repositorios crea un ficheiro onde se almacenan todos os cambios en distintos arquivos. Isto axuda a volver a versións anteriores en caso de erros nas modificacións máis recentes ou evitar perder o traballo en caso de fallo do equipo de traballo.

5.6.5 SystemC

– IGUAL que en modelado simulacion Trátase dunha meta-linguaxe (unha librería e un conxunto de macros) creada en C++ empregada para Codeseño. Contén soporte para data-flow e permite engadir código en C++ sen problema, polo que se pode traballar con clases, facilitando un deseño modular. Ademais engade funcionalidades similares ás de Verilog ou

VHDL. O que fai que sexa unha alternativa a estas dúas linguaxes é que permite misturar deseño **RTL** con código C++ para imprimir por pantalla ou ler arquivos. C tamén podería ser outra opción; sen embargo, a falta de datos públicos e privados, non existe a mesma facilidade para organizar todo en módulos e hai poucos tipos de datos aptos, e crealos implica empregar funcións sempre.

Capítulo 6

Probas

UNHA parte imprescindible de calquera proxecto é o período de probas ou testing, durante o cal se busca atopar bugs e comprobar que o funcionamento é o esperado e correcto. Ao longo deste capítulo explicaránse os distintos exames aos que se someteu o simulador, o seu obxectivo, orixe e diferenzas fundamentais.

6.1 Benchmarks

Unha vez implementada unha nova instrución, ou un pipeline, é necesario comprobar que o funcionamento é o esperado. Para iso, empréganse diferentes métodos. Un deles son os benchmarks, diferentes probas que buscan crear casos habituais e incluso os máis edge cases. A fonte destes benchmarks é o repositorio de RISC-V test [10]. Aquí existen diferentes programas orientados a probar determinadas funcións, como a multiplicación con SPMV. Os benchmarks empregados durante o traballo son os seguintes:

Nome do benchmark	Obxectivo
<i>SPMV</i>	Multiplicacións
<i>Median</i>	Suma, comparacións e desplazamentos de datos
<i>Multiply</i>	Suma, resta, comparacións e desplazamentos de datos
<i>Qsort</i>	Suma e comparacións con operando inmediato e desplazamentos de datos
<i>Rsort</i>	Suma e comparacións con operando inmediato e desplazamentos de datos
<i>Vvadd</i>	Suma de vectores

Táboa 6.1: Benchmarks empregados e con que fin

6.2 Tests propios

Ademais de empregar os benchmarks, créaronse varios exames buscando probar especificamente certas funcionalidades segundo fose necesario. O concepto básico foi imitar algún benchmark de instrución atopado no repositorio oficial [10]. Como se ve no apéndice A.1, consiste en empregar código ensamblador embebido [11] para integrar a instrución no código en C. Ademais, compróbase o resultado da operación gardando o que devolve e comparando co resultado esperado. Na súa maioría son bastante sinxelos; sen embargo, tendo en conta determinados casos que poderían ser problemáticos, serven para determinar se unha instrución está ben implementada.

6.3 Depuración

Chámase depuración ao proceso de revisión exhaustiva do software en busca de erros. Calquera programa durante o proceso de desenvolvemento sofre varias revisións, tipicamente empregando o IDE. Este permite deterse en determinada instrución, imprimir o valor dunha variable antes e despois dun cambio, etc. Para este punto, tanto Segger como Visual Studio foron moi útiles, xa que proporcionan incluían ferramentas perfectamente integradas.

Capítulo 7

Uso do simulador

NESTE capítulo explícase brevemente como empregar o simulador, explicando como xerar os arquivos .elf e empregar o simulador, así como interpretar os resultados.

O primeiro paso é empregar Segger Embedded Studio for RISC-V, aquí escribírase o código C para o programa que executará o simulador. Antes de compilar, tendo seleccionado no panel esquerdo Project [Nome do proxecto], débese modificar en Project -> Compiler como se mostra na 7.1, é necesario elixir a extensión correcta. Por exemplo, no caso de que se realicen multiplicacións, débese cambiar de RV32I (por defecto) a RV32IM (ver 7.2).

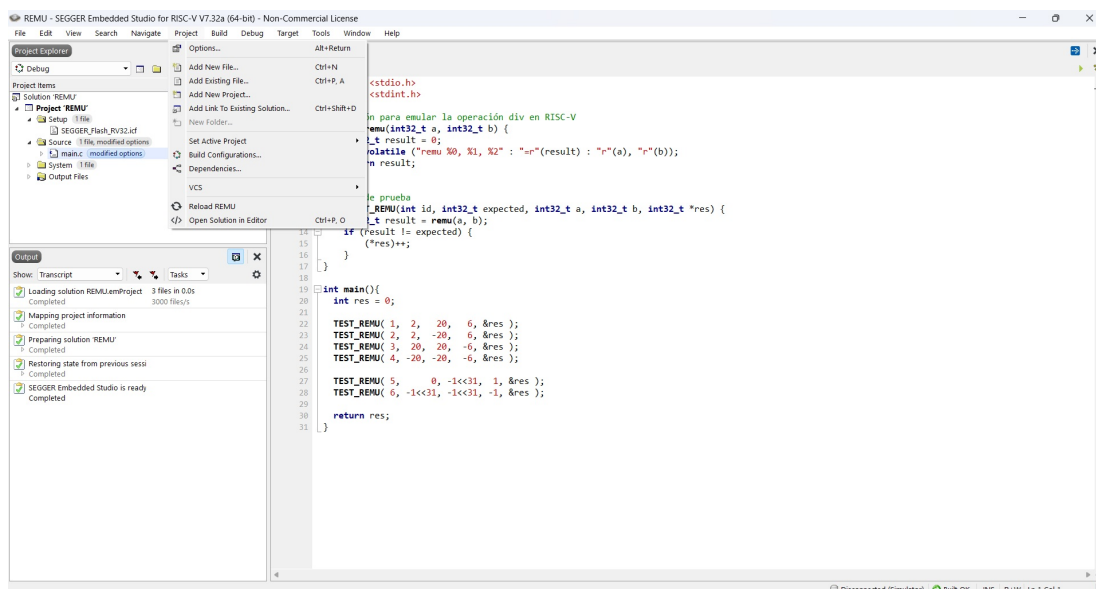


Figura 7.1: Opcións do proxecto

Unha vez feito isto, Build -> Build Solution, como mostra a captura 7.3. Agora na carpeta Output Files, están varios arquivos, entre eles o executable con extensión .elf. É posible executar e depurar o código dentro do propio Segger, podendo así comprobar se o test está

correcto de forma rápida (ver 7.4).

Para o axuste de parámetros debemos ir á Visual Studio 2022, no ficheiro config.h, aparecen definidas constantes para a latencia de instrucións, co nomes que seguen o formato LatencyNomeInstrución, por exemplo LatencyMul, como se ve na captura 7.5. Finalmente, abrírase unha pantalla onde se mostrará que módulos están compilados, o tempo, o número de ciclos e o número de instrucións que levou o test. Importante revisar se o resultado é correcto, para isto tal e como se mostra na 7.6 imprímese o valor do rexistro x10, onde se almacena un 0 se o resultado é o esperado ou 1 se hai algún erro.

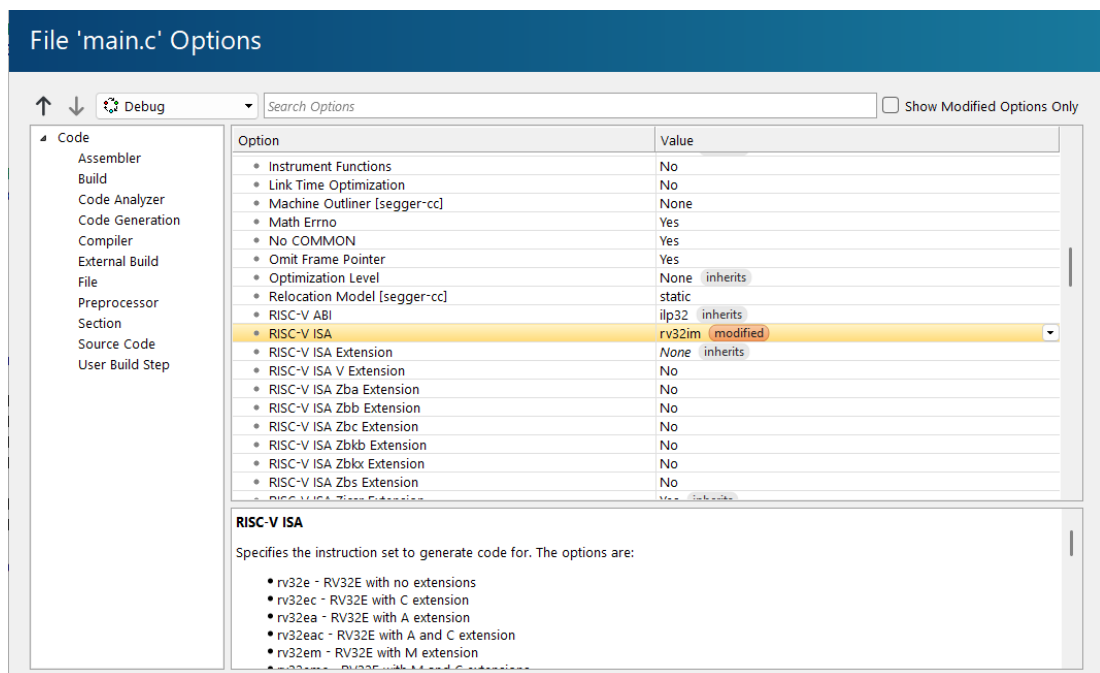


Figura 7.2: Elección da extensión correcta

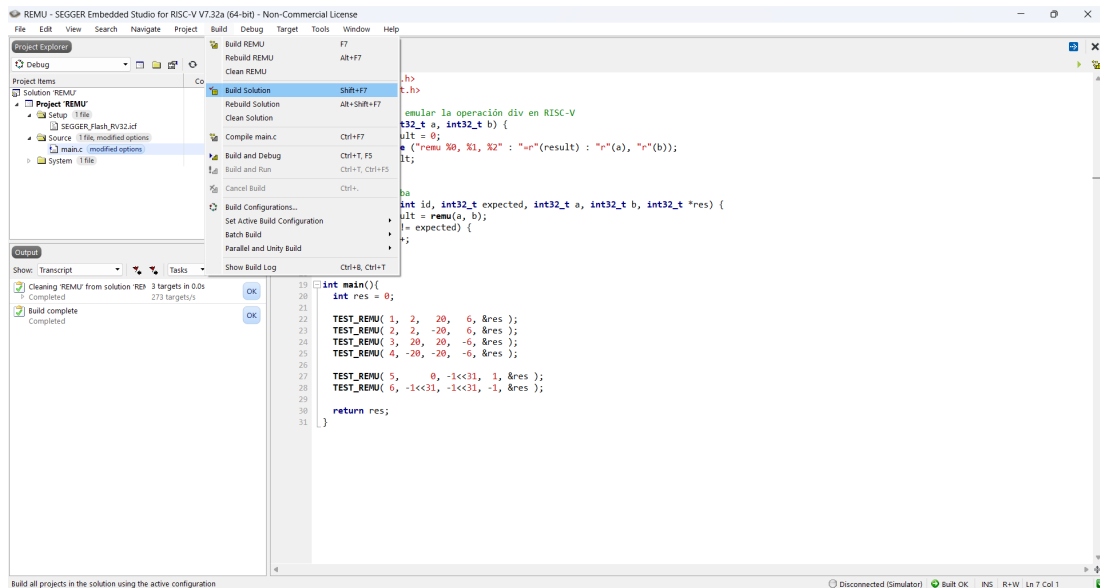


Figura 7.3: Compilación do proxecto

CAPÍTULO 7. USO DO SIMULADOR

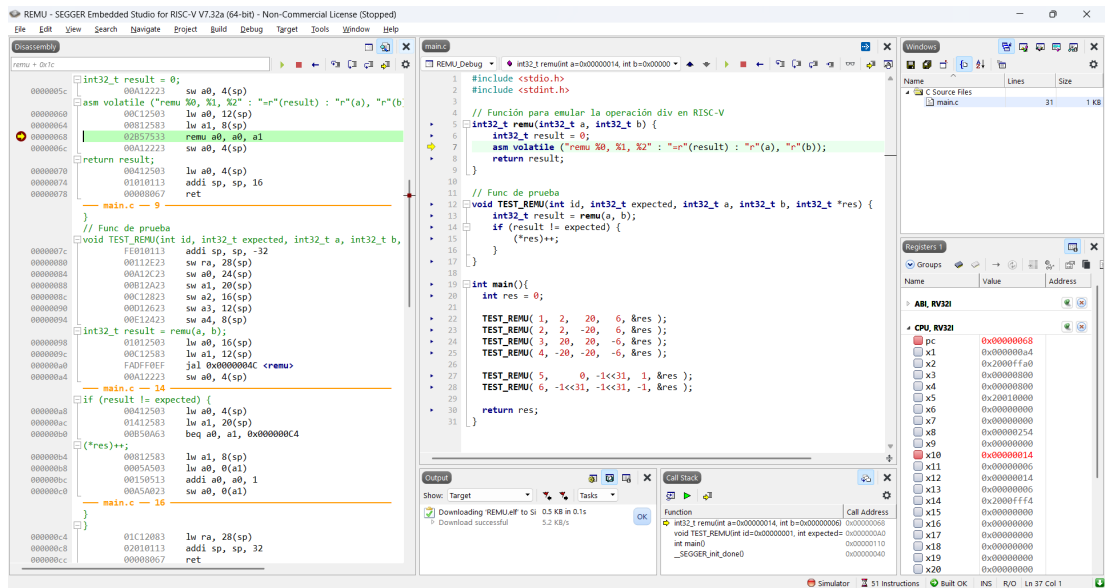


Figura 7.4: Captura coas opcións de depuración de Segger

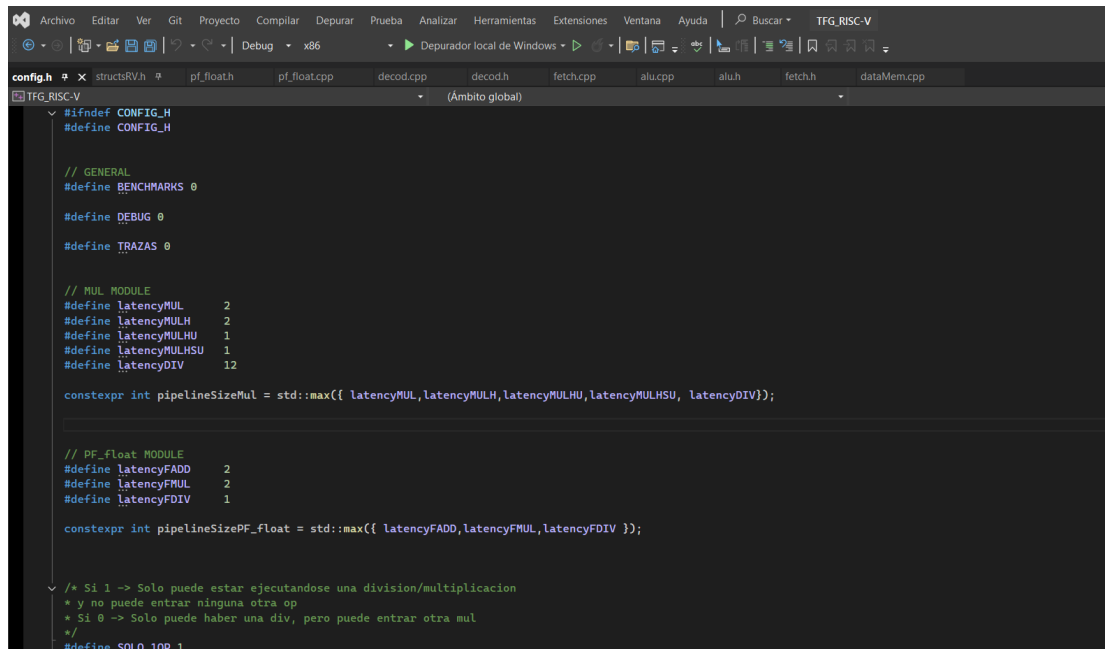
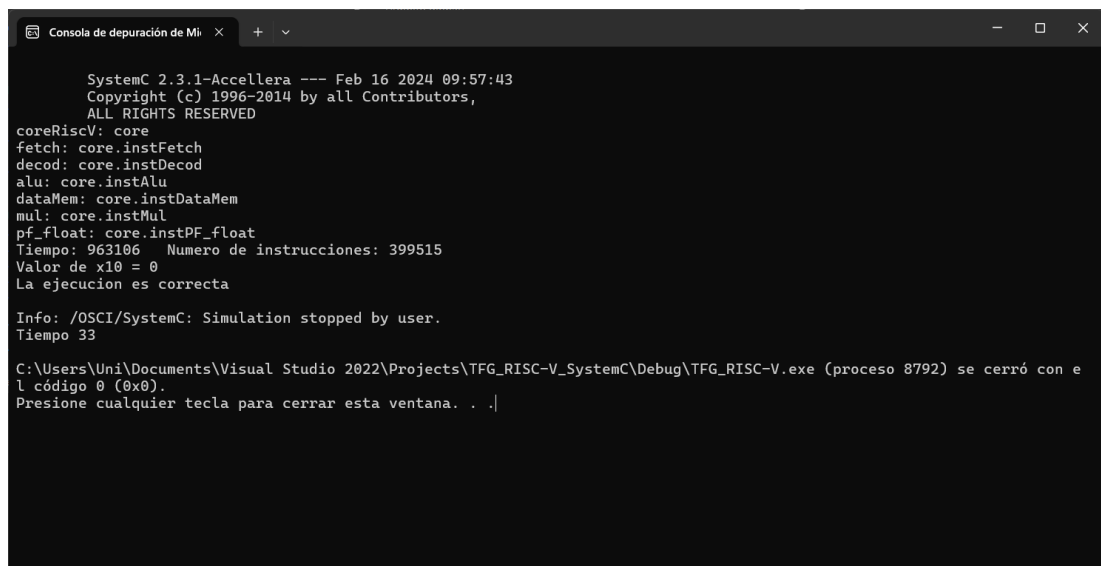


Figura 7.5: Cambio de parámetros en Config.h



```
Consola de depuración de Mi x + v
SystemC 2.3.1-Accellera --- Feb 16 2024 09:57:43
Copyright (c) 1996-2014 by all Contributors,
ALL RIGHTS RESERVED
coreRiscV: core
fetch: core.instFetch
decod: core.instDecod
alu: core.instAlu
dataMem: core.instDataMem
mul: core.instMul
pf_float: core.instPF_float
Tiempo: 963106 Numero de instrucciones: 399515
Valor de x10 = 0
La ejecucion es correcta
Info: /OSCI/SystemC: Simulation stopped by user.
Tiempo 33
C:\Users\Uni\Documents\Visual Studio 2022\Projects\TFG_RISC-V_SystemC\Debug\TFG_RISC-V.exe (proceso 8792) se cerró con e
l código 0 (0x0).
Presione cualquier tecla para cerrar esta ventana. . .|
```

Figura 7.6: Resultados tras ejecutar o benchmark SPMV

Conclusións

DERRADEIRO capítulo da memoria, onde se presentará a situación final do traballo, as leccións aprendidas, a relación coas competencias da titulación en xeral e a mención en particular, posibles liñas futuras,...

8.1 Resultados

Tras finalizar o proxecto, pódese garantir que engadir novas extensións coas súas correspondentes novas instrucións non compromete o traballo anterior. O funcionamento do resto de módulos segue sendo correcto e o rendemento non se viu deteriorado en ningún momento.

Por outra parte, a posibilidade de modificar as latencias dalgunhas instrucións grazas á parametrización engadida, mostra como cambia o rendemento no conxunto dun programa. Por exemplo, á hora de executar o benchmark SPMV que realiza multiplicación de enteiros, obtemos resultados moi interesantes segundo as latencias.

Como vemos na táboa 8.1, o número de instrucións non varía, o que é lóxico xa que só se modifica a súa latencia. Un cambio na cantidade de operacións realizadas implicaría engadir novas instrucións dependendo da latencia de determinadas instrucións. Se ben se emiten instrucións NOP durante cando se detectan hazards, estas fan a función de burbullas, non se fai nada salvo pasar ao seguinte ciclo.

Por outra parte, o tempo, isto é, o número de ciclos necesarios para executar o programa aumenta notablemente. Comparando o aumento de ciclos para a mesma latencia en mul e mulhu, dedúcese que se executan máis instrucións mul. O cal é razoable, xa que revisando o binario, vese que é correcto. Ademais, o test realiza multiplicacións de enteiros, e mul é imprescindible para isto, mentres que mulh encárgase da parte superior da multiplicación, innecesaria cando se empregan números pequenos.

Finalmente, destacar que a velocidade de simulación deste proxecto en comparación coa que se podería obter se VHDL ou Verilog fose empregado é moi superior. Por exemplo, o

benchmark SPMV execútase en apenas 30 segundos.

8.2 Traballo futuro

Agora mesmo, o simulador inclúe todas as instrucións implementadas ata o que é coñecido como extensión G. Poderíanse engadir máis extensións, como a D. Da mesma forma, sería interesante simular un de memoria e a súa xerarquía. A implementación actual non é moi realista. Trátase dun sinxelo módulo que sempre escribe ou lee nun só ciclo. Tamén sería interesante incluír soporte para 64-bits, coa posibilidade de seguir parametrizando o simulador, polo que, por exemplo, o tamaño dos rexistros ou o funcionamento do módulo de decodificación veríanse afectados segundo a base empregada.

Modificacións realizadas	Número de instrucións	Tempo
<i>SPMV base</i>	399515	955924
<i>Latencia de mul = 5</i>	399515	984652
<i>Latencia de mul = 10</i>	399515	1020562
<i>Latencia de mulhu = 5</i>	399515	965500
<i>Latencia de mulhu = 10</i>	399515	940504
<i>Latencia de mul = 5 e mulhu = 5</i>	399515	984652

Táboa 8.1: Rendemento do benchmarks SPMV segundo as latencia de distintas operacións.

Apéndices

Material adicional

ESTE capítulo ten formato de apéndice, inclúe material adicional que non ten cabida no corpo principal do documento, como código de tests.

A.1 Exemplo de código de probas

```

1 #include <stdio.h>
2 #include <stdint.h>
3
4 // Función para emular la operación remu en RISC-V
5 int32_t remu(int32_t a, int32_t b) {
6     int32_t result = 0;
7     asm volatile ("remu %0, %1, %2" : "=r"(result) : "r"(a),
8         "r"(b));
9     return result;
10 }
11
12 // Funcion de prueba
13 void TEST_REMU(int id, int32_t expected, int32_t a, int32_t b,
14     int32_t *res) {
15     int32_t result = remu(a, b);
16     if (result != expected) {
17         (*res)++;
18     }
19 }
20
21 int main(){
22     int res = 0;
23
24     TEST_REMU( 1,  2,  20,  6, &res );
25     TEST_REMU( 2,  2, -20,  6, &res );
26     TEST_REMU( 3, 20,  20, -6, &res );

```

```
25 TEST_REMU( 4, -20, -20, -6, &res );  
26  
27 TEST_REMU( 5,      0, -1<<31, 1, &res );  
28 TEST_REMU( 6, -1<<31, -1<<31, -1, &res );  
29  
30 return res;  
31 }
```

Relación de Acrónimos

ALU Arithmetic and Logical Unit. [9](#), [10](#)

ARM Advanced RISC Machine. [1](#)

CSR Control, Status and Register. [5](#)

HDL Hardware Description Language. [6](#)

IoT Internet of Things. [1](#), [4](#)

ISA Instruction Set Architecture. [1](#), [4](#)

ISS Instruction Set Simulator. [7](#)

NOP No Operation. [15](#)

OOP Object Oriented Programming. [7](#)

RAW Read After Write. [10](#)

RTL Register Transfer Level. [7](#), [8](#), [17](#)

TLM Transaction-Level Modeling. [7](#), [8](#)

VHDL VHSIC (Very High Speed Integrated Circuit) e HDL (Hardware Description Language). [17](#)

WAR Write After Read. [10](#)

WAW Write After Write. [10](#)

Glosario

arquitectura No contexto da informática, refírese ao deseño e estrutura dun conxunto de circuítos e outros compoñentes dos que se compón un sistema. . [10](#)

benchmark Examen que se realiza coa fin de comprobar que un programa funciona sen erros e producindo a saída correcta.. [15](#)

chips Conxunto de circuítos integrados dentro dunha pequena peza de material semiconductor, co que se realizan varias funcións en ordenadores e outros dispositivos electrónicos.. [1](#)

hardware Partes físicas dun sistema informático, formado por todos os compoñentes electrónicos, circuítos e periféricos.. [16](#)

hazard Risco producido por unha dependencia RAW,WAR ou WAW. Pode chegar a causar erros na execución dun programa informático.. [1](#), [10](#)

meta-linguaxe Engadir funcionalidades á unha linguaxe mediante librarías e conxuntos de macros.. [7](#)

software Conxunto de compoñentes lóxicos que permiten realizar determinadas funcións nun equipo tecnolóxico.. [6](#), [16](#)

test Proba mediante a cal se revisa que o funcionamento dun programa é o esperado. Tipicamente, busca simular casos reais de execución.. [15](#)

Bibliografía

- [1] P. Valerio, “Reshaping the Landscape of IoT with RISC-V,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://www.eetimes.com/reshaping-the-landscape-iot-with-risc-v/>
- [2] J. Pastor, “RISC-V necesitaba dar un paso de gigante para competir con ARM. Acaba de hacerlo gracias a Google,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://www.xataka.com/moviles/risc-v-necesitaba-dar-paso-gigante-para-competir-arm-acaba-hacerlo-gracias-a-google>
- [3] RISC-V International, “Spike,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://github.com/riscv-software-src/riscv-isa-sim>
- [4] ChipVerify, “Introduction to verification,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://www.chipverify.com/tutorials/verification>
- [5] J. R. Scott, “RISC-V Designs,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://johnrscott.github.io/rvdocs/single-cycle/0.1.0/verification/verification.html>
- [6] Wikipedia, “RISC-V,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://es.wikipedia.org/wiki/RISC-V>
- [7] A. Waterman, “Design of the RISC-V Instruction Set Architecture,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2016. [En liña]. Dispoñible en: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>
- [8] RISC-V International, “RISC-V Ratified Extensions,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://riscv.org/specifications/ratified/>
- [9] Wikipedia, “Register-Transfer Level,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: https://en.wikipedia.org/wiki/Register-transfer_level

- [10] RISC-V International, “RISC-V Benchmarks,” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://github.com/riscv-software-src/riscv-tests>
- [11] GCC GNU, “Extended ASM using the GNU Compiler Collection (GCC),” consultado o 22 de xuño de 2025. [En liña]. Dispoñible en: <https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>