Facultade de Informática
UNIVERSIDADE DA CORUÑA

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

Simulador de RISC-V empregando SystemC

Estudante: Hugo Mato Cancela

Dirección: Roberto Rodríguez Osorio

A Coruña, xuño de 2025.

Aos meus pais, familiares e amigos que me apoiaron sempre sen importar as circunstancias. A todos os profesores e profesoras que tiven ao longo da miña traxectoria académica, que fixeron posible isto.

Agradecementos

Lorem ipsum dolor sit amet, consectetuer 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, consectetuer 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 Advanced RISC Machine (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 deste 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

Índice de Figuras

Índice de Táboas

Capítulo 1

Introdución

Este proxecto busca crear un simulador de RISC-V empregando a libraría 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 REV: - biblio verilog e vhdl - agradecementos - resultados - siglas acrfull / acrshort

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 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 existen varios simuladores, cada un está especializado nun rango de aplicacións, e consideramos que a simulación orientada a unha implementación posterior non está suficientemente cuberta. Así, simuladores como RARS [3] ou Spike [4] son especialmente útiles na docencia ou valoracións da arquitectura. Por outra banda, RISC-V-TLM [5] funciona a un nivel moi superior, simulando a nivel de transferencias, e permite unha simulación de sistemas completos aínda que relaxando moito a precisión temporal. Por último, Chisel [6] está baseado en Scala, e permite o modelado de circuítos dixitais. A relevancia de Chisel é que foi utilizado no desenvolvemento de RISC-V, ademais de que permite xerar automáticamente código Verilog simulable e sintetizable.

As vantaxes de empregar SystemC son o control absoluto sobre todo os aspectos do modelado, a velocidade de simulación, e a facilidade para depurar o código ao estar baseado en C++.

Durante o proceso de deseño, unha parte clave é a verificación do correcto funcionamento [7, 8]. Debido ao elevado custo de fabricación, e ao tempo necesario (ao redor de 3 meses), é inviable crear un chip para cada versión. 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).
- 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 o multiplicador e o divisor comparten circuítos, cunha na que ambos son independentes, e tamén comparar un divisor totalmente segmentado cun 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 foi incremental, dividindo as tarefas en partes independentes que foron implementadas, simuladas e verificadas por orde de complexidade antes de proceder coa seguinte.

O procedemento habitual foi dunha reunión semanal na que se revisaba o feito anteriormente, acompañado de comprobación cos tests correspondentes para esa parte. Despois, decidíase cal era 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 describirase 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.** Aqui 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 [9], foi crecendo pouco a pouco, incluso con axuda de voluntarios fóra do ámbito académico. Os puntos fortes desta arquitectura son a súa aposta por unha 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 a fai unha boa candidata para ser empregada en dispositivos IoT. Se ben xa existen ISAs moito máis populares e amplamente estendidas, como por exemplo a ARMv7 [10], si que hai 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 á 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 máis habituais contan cunha versión validada [11], xa que se empregan na inmensa maioría de deseños. Cada unha traballa sobre unha ou máis bases, engadindo funcionalidades adicionais, creando un deseño modular. En canto ás extensións:

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

Táboa 2.1: Nome e descripcións das extensións

Capítulo 3

Modelado e simulación

Neste apartado explicaránse os fundamentos dun simulador, os motivos para crear un e o funcionamento típico. Ademais, indicaránse 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. O xeito máis económico de acadar isto é a creación dunha versión dixital mediante software. Idealmente, o modelo creado poder ser executado permitindo simular o sistema, e mesmo ter acceso aos detalles internos do funcionamento. A elección axeitada das ferramentas usadas para o modelado e a simulación é fundamental, xa que definen o tempo necesario para codificar o sistema, a posibilidade de cometer erros e maila capacidade para detectalos, o grao de detalle que se pode acadar, as distintas restricións e a velocidade de simulación. É por iso que existen distintas opcións, polo que neste traballo nos decidimos por usar SystemC.

3.2 VHDL e Verilog

Estas linguaxes son o que coñecemos como Hardware Description Language (HDL). Verilog [12] foi o pioneiro, e VHSIC (Very High Speed Integrated Circuit) e HDL (Hardware Description Language) (VHDL) [13] a resposta como estándar internacional. Ambas permiten describir circuítos de forma moi precisa, incluíndo todas as conexións e elementos de memoria. Ademais, existe software de síntese capaz de xerar circuítos de alta calidade a partir de Verilog ou VHDL, imposibles de realizar para un ser humano. Ao ser o estándar na industria, son as máis empregadas para modelar e simular a baixo nivel. Pero esta flexibilida-

de implica tamén desvantaxes en termos de menor produtividade dos enxeñeiros e elevados tempos de simulación. Aínda que houbo intentos de mellorar estas linguaxes, como é o caso de SystemVerilog [14], non tiveron realmente éxito ó estar demasiado vencelladas á linguaxe orixinal.

3.3 SystemC

SystemC [15] é unha meta-linguaxe implementada como unha biblioteca de C++ e é habitualmente empregada para codeseño. O feito de que sexa de alto nivel proporciona unha flexibilidade e sinxeleza á hora de traballar que carecen as alternativas máis próximas ao hardware. Algunhas das vantaxes de SystemC son as seguintes:

- Inicialización automática de datos internos.
- Funcionalidades de linguaxes orientados a obxectos.
- Xestión automática de eventos.
- Difrenciación entre datos públicos e privados.
- Tipos de datos orientados ao modelado de hardware de aplicación específica.
- Soporte específico para sinais de reloxo.
- Sobrecarga de operadores que aumenta a produtividade e a claridade do código.
- Benefíciase dos avances en depuradores para C++.

SystemC soporta varios paradigmas de modelado, dos que os máis utilizados son, en orde crecente de abstracción: Register Transfer Level (RTL), Data Flow e Transaction-Level Modeling (TLM). No noso caso, o modelado en RTL permite acadar o mesmo nivel de detalle que VHDL ou Verilog cunha produtividade e velocidade de simulación moi superiores.

3.4 Spike

A propia organización de RISC-V xa ofrece un simulador [4], pero seguen existindo motivos para crear unha alternativa. Non simula cada ciclo, senón que é un 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 M (multiplicación/división), A (atómicas), F/D (punto flotante simple/dobre precisión), C (instrucións comprimidas) e V (vectorial). Inclúe soporte para debug, simula diferentes niveis de privilexio e ofrece 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 e orientado a unha posterior implementación do deseño. 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.

Capítulo 4

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 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) [16].

Se ben unha alternativa interesante sería TLM, que é o seguinte nivel de deseño electrónico, a abstracción que proporciona neste caso é demasiado alta para os detalles máis relevantes deste proxecto.

Tipicamente, para este nivel tan baixo, o habitual é empregar VHDL ou Verilog. Como se comentou no capítulo 3.2 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 1941, aínda que non se popularizaron ata os anos 70 [17]. A idea é aumentar o rendemento ao permitir que o procesador execute máis dunha instrución por ciclo. Para iso, a versión máis básica é a Reduced Instruction Set Computing (RISC) como se mostra na imaxe 4.1, formada por 5 etapas: 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 á súa sinxeleza. A aproximación realizada foi dividir cada etapa en un módulo do simulador, salvando Decod e Write Back que se uniron por comodidade.

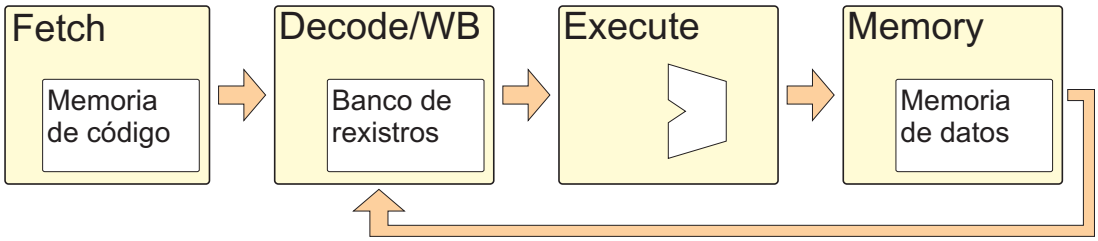

Figura 4.1: Figura onde se motra a estrutura do pipeline de 5 etapas.

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 conectado directamente con Decod, como se pode ver na figura 4.2. 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, podendo elixir a latencia de determinadas instrucións ou cambiar o funcionamento do módulo (ver 4.4).

4.4 Modos de operación

Coa fin de mellorar a calidade da simulación, decidíuse 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 circuítos para ambas operacións. O segundo modo permite que se executen

todas as multiplicacións necesarias, pero só unha división ao mesmo tempo, simulando que existe un circuíto para as divisións e que este só pode realizar unha á vez.

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 durante a etapa de Execución. 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 anterior 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 finalicen. O simulador replica este funcionamento, polo que para detectar estas dependencias emprega sinais de hazard.

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

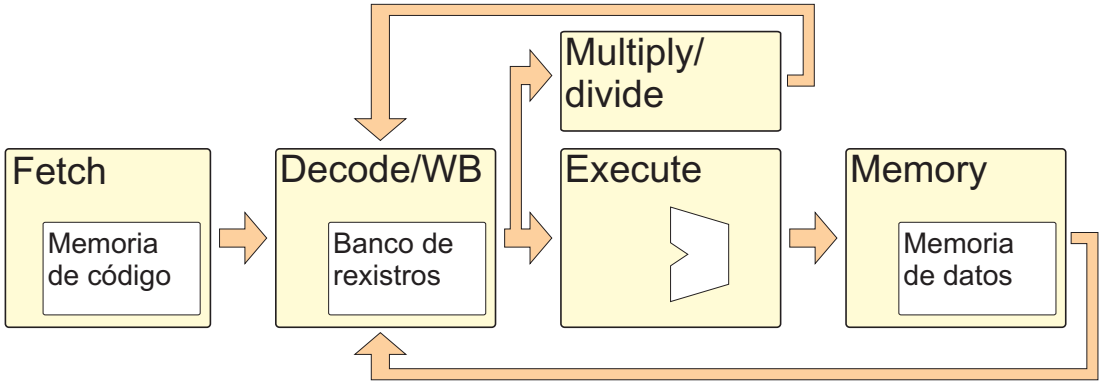

Figura 4.2: Esquema sobre o pipeline do módulo de multiplicación.

Capítulo 5

Implementación

TRAS haber creado o deseño, é necesario realizar a implementación. Neste capítulo, trataranse 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 compoñente, 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 as extensións M, F (parcialmente), Zifencei e Zicsr, ademais das funcionalidades da base RV32I. 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	
Mul	Implementada
Mulh	Implementada

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

Táboa 5.1 – *(vén da páxina anterior)*

Nome da operación	Estado da implementación
Mulhsu	Implementada
Mulhu	Implementada
Div	Implementada
Divu	Implementada
Rem	Implementada
Remu	Implementada
Extensión F — Punto flotante en simple precisión	
Fadd.s	Implementada
Fcvt.s.w	Implementada
Fcvt.s.wu	Implementada
Fcvt.w.s	Implementada
Fcvt.wu.s	Implementada
Flw	Implementada
Fmul.s	Implementada
Fmv.w.x	Implementada
Fmv.x.w	Implementada
Fsub.s	Implementada
Fsw	Implementada
Extensión Zicsr — Acceso a rexistros CSR	
Csrrw	Implementada
Csrrs	Implementada
Csrrc	Implementada

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

Táboa 5.1 – *(vén da páxina anterior)*

Nome da operación	Estado da implementación
Csrrwi	Implementada
Csrrsi	Implementada
Csrrci	Implementada
Extensión Zifencei — Sincronización de instrucións	
Fence.i	Implementada

5.3 Implementacións dos pipelines

Neste proxecto 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 imitar 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 saiu é un No Operation (NOP), búscase a anterior para que sexa executada.

5.4 Funcionalidades do simulador

Antes de comezar este proxecto, xa existía unha 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.

Ao finalizar este traballo, 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 Zicsr e Zifencei. Agora, permite a execución de multiplicacións, divisións e operacións con datos de tipo float, xunto fence e instrucións de tipo CSR.

5.5 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.

5.5.1 Segger Embedded Studio for RISC-V

Segger Embedded Studio for RISC-V é un Integrated Development Environment (IDE) que permite compilar para RISC-V, incluindo obxectivos concretos como RV32, producir arquivos .elf e ver o código ensamblador. Foi principalmente empregado á hora de escribir código en C para tests ou benchmarks. Ademais, o depurador permite ver código ensamblador coas direccións, polo que foi realmente útil á hora de encontrar bugs. 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 cunha configuración avanzada tamén podería ser unha alternativa. 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 e todas as ferramentas necesarias sen apenas configuración.

5.5.2 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 infinidade de alternativas, pero este foi o elixido por ser a elección máis habitual para este tipo de proxectos polo estudante. Ademais, xa fora empregado na asinatura de Codeseño hardware/software xunto a SystemC.

5.5.3 GTK Wave

Para solventar algúns dos problemas máis complexos, 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.

5.5.4 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 a perda do traballo en caso de fallo do equipo de traballo.

5.5.5 SystemC

As bibliotecas de SystemC son gratuítas, e integralas con Visual Studio é relativamente doado. Sen embargo, non se ofrecen precompiladas, e o usuario debe compilar o código fonte para cada unha das configuracións: Debug e Release. A correcta configuración do proxecto de Visual Studio fai que o compilador alterne entre ámbalas dúas versións automáticamente.

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 é 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 extremos ou menos frecuentes. A fonte destes benchmarks é o repositorio de RISC-V [18]. 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
SPMV	Multiplicacións
Median	Suma, comparacións e desplazamentos de datos
Multiply	Suma, resta, comparacións e desplazamentos de datos
Qsort	Suma e comparacións con operando inmediato e desplazamentos de datos
Rsort	Suma e comparacións con operando inmediato e desplazamentos de datos
Vvadd	Suma de vectores

Táboa 6.1: Benchmarks empregados e con que fin.

6.2 Tests propios

Ademais de empregar os benchmarks, foron creados varios exames buscando probar especificamente certas funcionalidades segundo fose necesario. O concepto básico foi imitar algún benchmark de instrución atopado no repositorio oficial [18]. Como se ve no apéndice A.1 , consiste en empregar código ensamblador embebido [19] 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 ferramentas perfectamente integradas. Neste proxecto, tamén foi moi útil GTK Wave (ver 5.5.3) para poder visualizar as trazas dos sinais máis relevantes, para así ver como varían ciclo a ciclo e poder comparar de forma visual e sinxela.

6.4 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 6.2, 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 cando se detectan hazards, estas fan a función de burbullas, non se fai nada salvo pasar ao seguinte ciclo. Sen embargo, 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 a operación mul é imprescindible para isto, mentres que mulhu encárgase da parte superior da multiplicación, innecesaria cando se empregan números pequenos.

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

Táboa 6.2: Rendemento do benchmarks SPMV segundo as latencia de distintas operacións.

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.

Capítulo 7

Uso do simulador

Neste capítulo explícase brevemente como empregar o simulador, explicando cómo 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 escribirase 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).

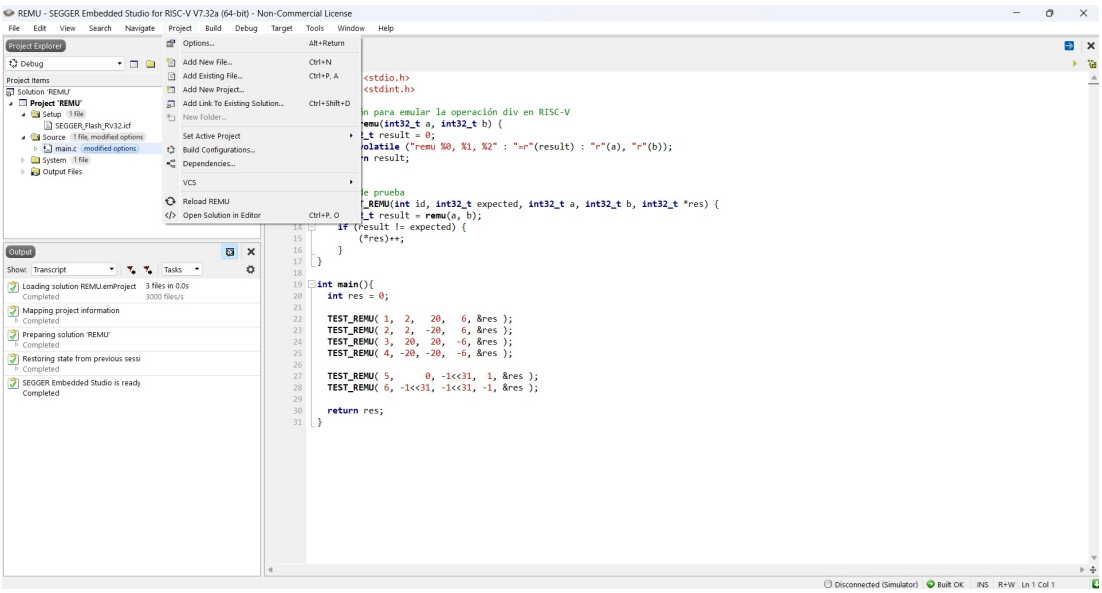

Figura 7.1: Opcións do proxecto.

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, abrirase unha pantalla onde se mostrará que módulos están compilados, o tempo, o número de ciclos e o número de instrucións que levou o test. É 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.

Para o proceso de depuración, como se comentou na sección 5.5.3 e 6.3, GTK Wave foi empregada para mostrar de forma gráfica os cambios de valores dos distintos sinais. Tamén engadía a posibilidade de ver os cambios de valor segundo o ciclo como se mostra na figura 7.7, o que fixo máis sinxelo solventar problemas de comunicación entre módulos ou de envío de alertas por hazards, por exemplo.

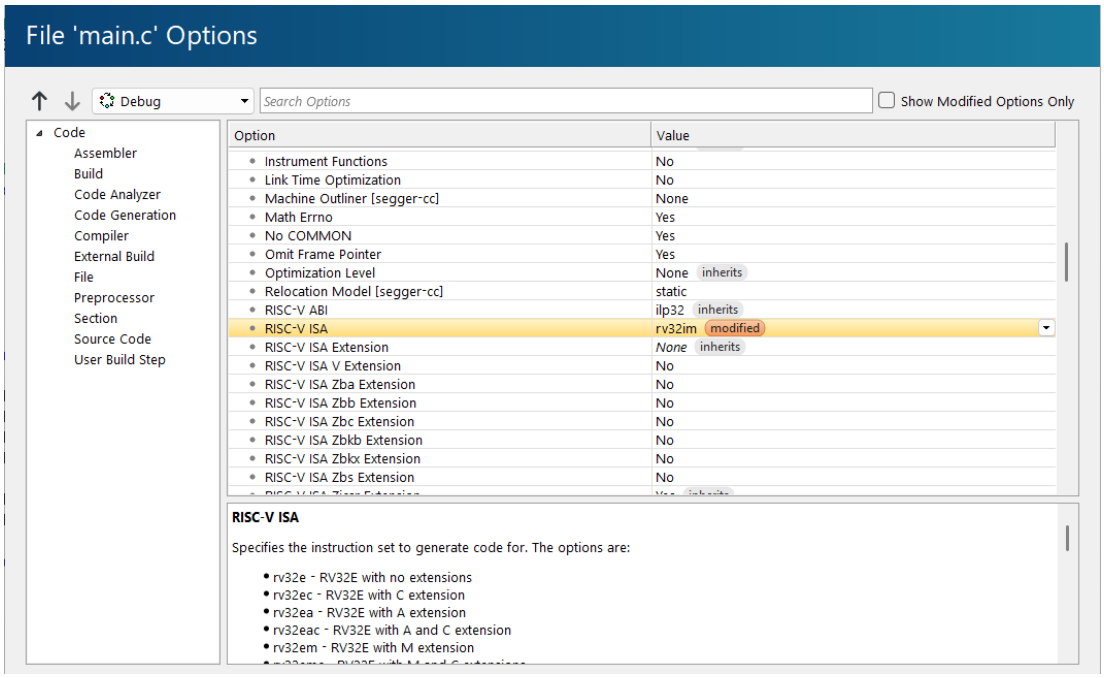

Figura 7.2: Elección da extensión correcta.

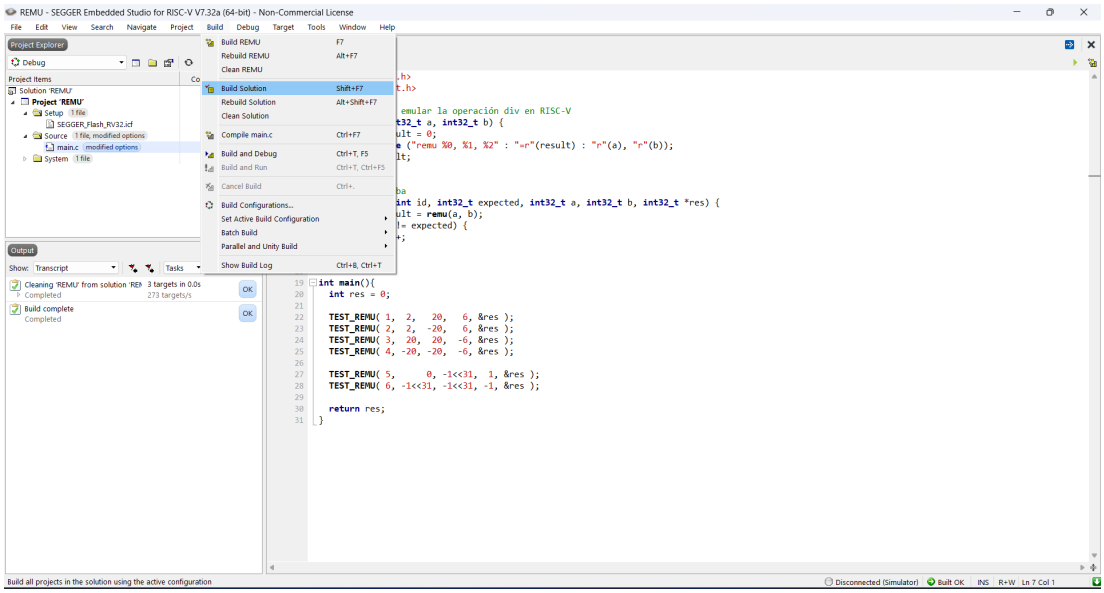

Figura 7.3: Compilación do proxecto.

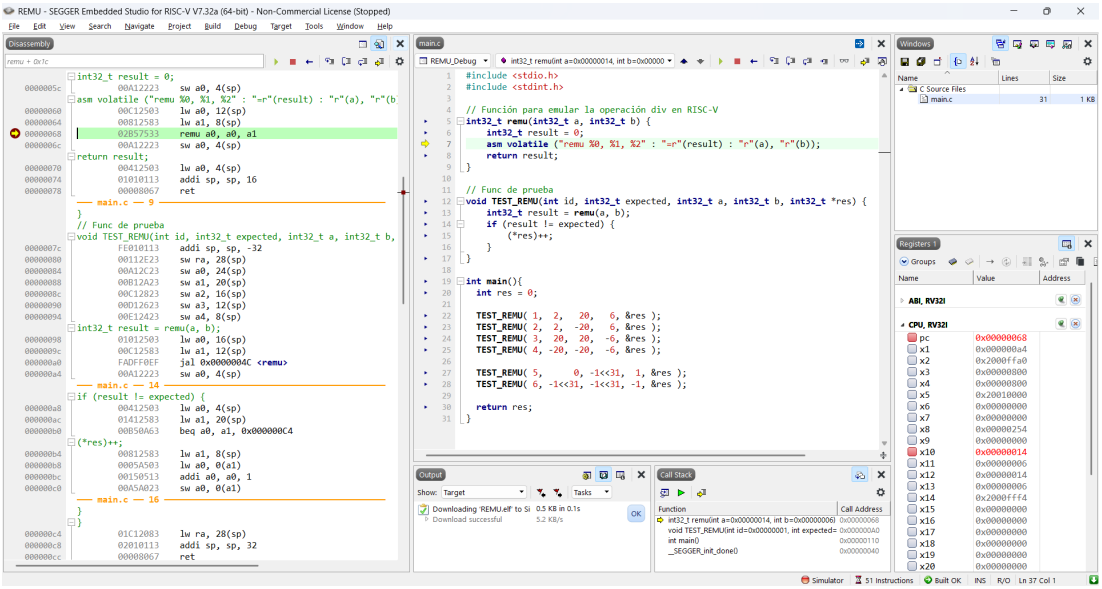

Figura 7.4: Captura coas opcións de depuración de Segger.

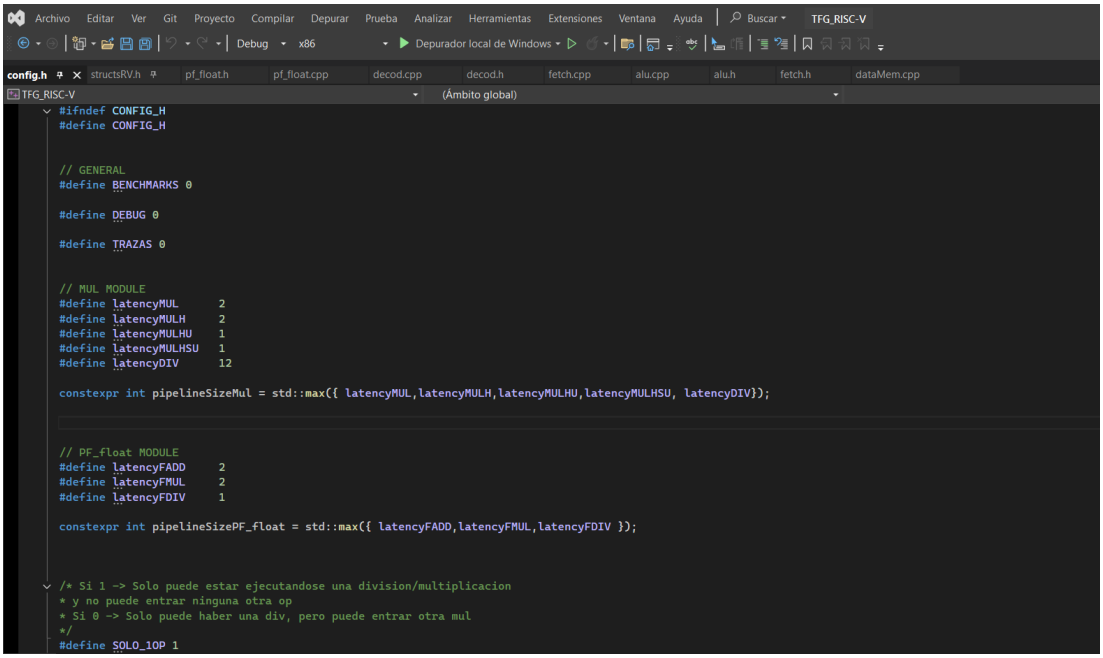

Figura 7.5: Cambio de parámetros en Config.h.

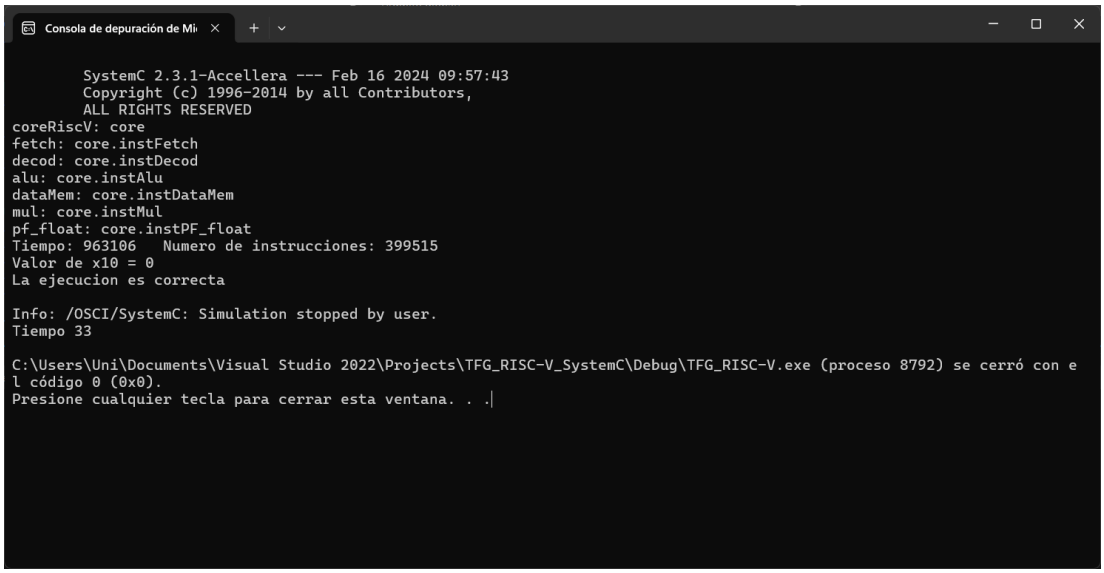

Figura 7.6: Resultados tras executar o benchmark SPMV.

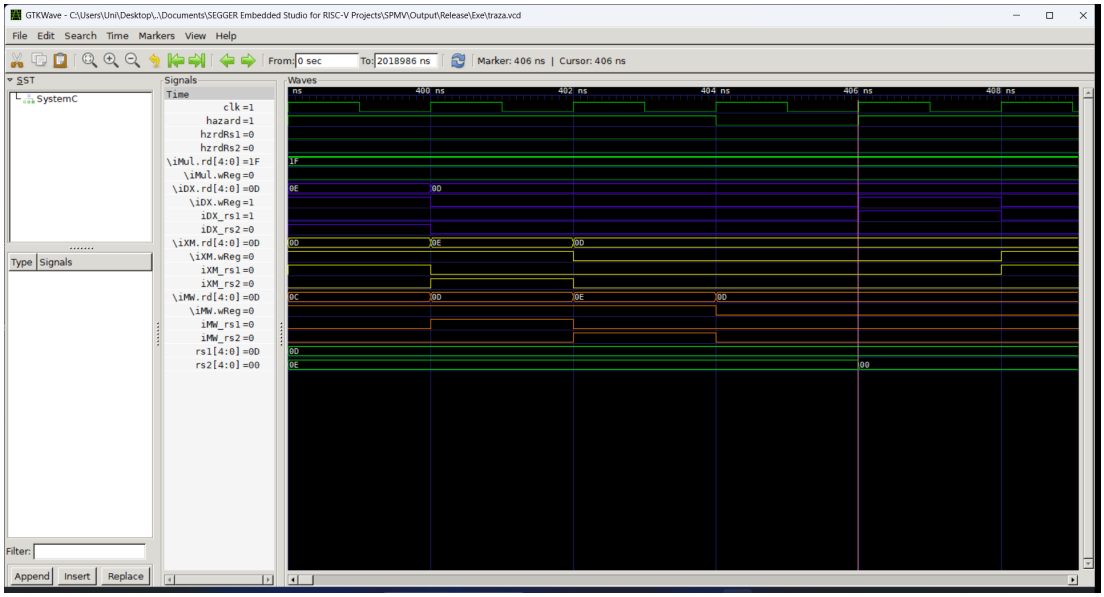

Figura 7.7: Captura de GTK Wave onde mostran varios sinais e o eixo temporal.

Capítulo 8

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 Traballo futuro

Agora mesmo, o simulador ten implementadas todas as instrucións das extensións M (multiplicación/división), Zicsr, Zifencei e unha gran parte da extensión F (punto flotante simple). Poderíanse engadir máis extensións, como a A, D ou L. Da mesma forma, sería interesante simular a memoria e a súa xerarquía. A implementación actual non é moi realista xa que se trata dun sinxelo módulo que sempre escribe ou lee nun só ciclo. Tamén sería interesante incluir 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.

Apéndices

Apéndice A

Material adicional

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

A.1 Exemplo de código de probas

```
#include <stdio.h>
#include <stdint.h>

// Función para emular la operación remu en RISC-V
int32_t remu(int32_t a, int32_t b) {
    int32_t result = 0;
    asm volatile ("remu %0, %1, %2" : "=r"(result) : "r"(a),
    "r"(b));
    return result;
}

// Funcion de prueba
void TEST_REMU(int id, int32_t expected, int32_t a, int32_t b,
    int32_t *res) {
    int32_t result = remu(a, b);
    if (result != expected) {
        (*res)++;
    }
}

int main(){
  int res = 0;

  TEST_REMU( 1,  2,   20,   6, &res );
  TEST_REMU( 2,  2,  -20,   6, &res );
  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  }
```

A.2 Módulo de multiplicación

A continuación móstrase o código do módulo de multiplicación, composto polo ficheiro de cabeceira e o correspondente corpo.

```
1  #ifndef MUL_H
2  #define MUL_H
3
4  #include "systemc.h"
5  #include "structsRV.h"
6  #include "config.h"
7  #include "auxFuncs.h"
8
9
10 SC_MODULE(mul) {
11 public:
12
13   sc_in <bool> clk, rst;
14   sc_in <instruction> I;
15
16   // Hazard detection from Decod
17   sc_in <sc_uint<5>> rs1In, rs2In;
18   sc_out <bool> hzrdRs1Out, hzrdRs2Out;
19
20   sc_out <bool> readyFenceMulOut;
21
22   sc_out <instruction> instOut;
23
24
25   void multiplication();
26
27   void hazardDetection();
28
29   SC_CTOR(mul) {
30     cout << "mul: " << name() << endl;
31
32     // NOP
```

```
33      INST = createNOP();
34
35      SC_METHOD(multiplication);
36      sensitive << clk.pos();
37
38      SC_METHOD(hazardDetection);
39      sensitive << rs1In << rs2In << fire;
40
41      fire.write(true);
42
43    }
44
45  private:
46
47    instruction INST;
48    sc_signal <bool> fire;
49
50    instruction pipeline[pipelineSizeMul];
51
52    bool pipelineFull = false;
53    bool flagDiv = false;
54
55  };
56
57  #define MUL 16
58  #define MULH 17
59  #define MULHSU 18
60  #define MULHU 19
61  #define DIV 20
62  #define DIVU 21
63  #define REM 22
64  #define REMU 23
65
66  #endif
```

```
1  #include "mul.h"
2  #include "alu.h"
3
4
5  // COMPLETELY SEGMENTED
6  void mul::multiplication() {
7
8    sc_int <32> A, B, res;
9    sc_uint <5> opCode;
10   short target;
11   double tiempo;
```

```
12
13  tiempo = sc_time_stamp().to_double() / 1000.0;
14
15  if (rst.read()) {
16
17    // NOP
18    INST = createNOP();
19    instOut.write(INST);
20
21    // empty pipeline
22    for (int i = 0; i < pipelineSizeMul; i++) {
23      pipeline[i] = createNOP();
24    }
25
26
27  } else {
28
29    // Get data
30    INST = I.read();
31
32    A = INST.opA;
33    B = INST.opB;
34    target = INST.rd;
35    opCode = INST.aluOp;
36
37    // Independant pipeline for each instruction
38    int cyclesInPipeline = 0;
39    instruction output = pipeline[0];
40
41    for (int i = 0; i < pipelineSizeMul - 1; i++) {
42
43      cyclesInPipeline = pipelineSizeMul - i;
44
45      if (pipeline[i].wReg &&
    getLatencyOp(pipeline[i].aluOp,pipeline[i].target) <=
    cyclesInPipeline) {
46
47        output = pipeline[i];
48        pipeline[i] = createNOP();
49
50        // Div in output
51        if (pipeline[i].aluOp == DIV || pipeline[i].aluOp == DIVU ||
52          pipeline[i].aluOp == REM || pipeline[i].aluOp == REMU ) {
53          flagDiv = false;
54          pipelineFull = false;
55        }
```

```
56          break;
57        }
58      }
59
60      if (output.aluOp != 0) {
61        pipelineFull = false;
62      }
63
64      instOut.write(output);
65
66      // Loop to shift pipeline content
67      // Pos 0: exit
68      // Pos latencyMUL-1: newElement
69      for (int i = 0; i < pipelineSizeMul - 1; i++) {
70        pipeline[i] = pipeline[i + 1];
71      }
72
73      sc_int<64> tmp = 0;
74
75      // Operate
76      switch (opCode)
77      {
78      case MUL:
79        tmp = ((sc_int<32>)A) * ((sc_int<32>)B);
80        INST.aluOut = INST.dataOut = tmp(31,0);
81        strcpy(INST.desc, "mul");
82        break;
83
84      case MULH:
85        tmp = ((sc_int<32>)A) * ((sc_int<32>)B);
86        INST.aluOut = INST.dataOut = tmp(63,32);
87        strcpy(INST.desc, "mulh");
88        break;
89
90      case MULHU:
91        tmp = ((sc_uint<32>)A) * ((sc_uint<32>)B);
92        INST.aluOut = INST.dataOut = tmp(63, 32);
93        strcpy(INST.desc, "mulhu");
94        break;
95
96      case MULHSU:
97        tmp = ((sc_int<32>)A) * ((sc_uint<32>)B);
98        INST.aluOut = INST.dataOut = tmp(63, 32);
99        strcpy(INST.desc, "mulhsu");
100       break;
101
```

```
102     case DIV:
103       if (B == 0) {
104         cerr << "Divider can't be 0 " << endl;
105       }
106       INST.aluOut = INST.dataOut = ((sc_int<32>)A) /
      ((sc_int<32>)B);
107       strcpy(INST.desc, "div");
108       flagDiv = true;
109       break;
110
111     case DIVU:
112       if (B == 0) {
113         cerr << "Divider can't be 0 " << endl;
114       }
115       INST.aluOut = INST.dataOut =((sc_uint<32>)A) /
      ((sc_uint<32>)B);
116       strcpy(INST.desc, "divu");
117       flagDiv = true;
118       break;
119
120     case REM:
121       if (B == 0) {
122         cerr << "Divider can't be 0 " << endl;
123       }
124       INST.aluOut = INST.dataOut = ((sc_int<32>)A) %
      ((sc_int<32>)B);
125       strcpy(INST.desc, "rem");
126       flagDiv = true;
127       break;
128
129     case REMU:
130       if (B == 0) {
131         cerr << "Divider can't be 0 " << endl;
132       }
133       INST.aluOut = INST.dataOut =  ((sc_uint<32>)A) %
      ((sc_uint<32>)B);
134       strcpy(INST.desc, "remu");
135       flagDiv = true;
136       break;
137
138     default:
139       INST = createNOP();
140       break;
141     }
142
143     // New instruction
```

```
144     pipeline[pipelineSizeMul - 1] = INST;
145
146   }
147   fire.write(!fire.read());
148 }
149
150 void mul::hazardDetection() {
151
152   int rs1 = rs1In.read();
153   int rs2 = rs2In.read();
154
155
156   bool aux1 = false,
157      aux2 = false;
158
159   int cont = 0;
160   bool emptyPipeline = false;
161
162   // Prevents RAW
163   for (int i = 0; i < pipelineSizeMul; i++) {
164
165     if (pipeline[i].wReg) {
166
167       if (rs1 == pipeline[i].rd) {
168         aux1 = true;
169       }
170
171       if (rs2 == pipeline[i].rd) {
172         aux2 = true;
173       }
174     }
175     else {
176       cont++;
177     }
178   }
179
180   if (instOut.read().wReg) {
181
182     if (rs1 == instOut.read().rd) {
183       aux1 = true;
184     }
185
186     if (rs2 == instOut.read().rd) {
187       aux2 = true;
188     }
189   }
```

```
190   else {
191     emptyPipeline = true;
192   }
193
194 #if SOLO_1OP
195
196   if (I.read().wReg) {
197     int opCode = I.read().aluOp;
198
199     if (isMulModuleOp(opCode)) {
200       aux1 = aux2 = true;
201       pipelineFull = true;
202     }
203     else if (!pipelineFull) {
204       aux1 = aux2 = false;
205     }
206   }
207   else {
208     emptyPipeline = emptyPipeline && true;
209   }
210
211
212 #else
213
214   if (I.read().wReg) {
215     int opCode = I.read().aluOp;
216
217     if (flagDiv && isMulModuleOp(opCode)) {
218       if (opCode == DIV || opCode == DIVU || opCode == REM ||
      opCode == REMU) {
219         aux1 = aux2 = true;
220       }
221       else {
222         aux1 = aux2 = false;
223       }
224     }
225   }
226   else {
227     aux1 = aux2 = false;
228     emptyPipeline = emptyPipeline && true;
229   }
230 #endif
231
232   if (cont == pipelineSizeMul && emptyPipeline) {
233     readyFenceMulOut.write(true);
234   }
```

```
235     else {
236       readyFenceMulOut.write(false);
237     }
238
239     hzrdRs1Out.write(aux1);
240     hzrdRs2Out.write(aux2);
241   }
```

Relación de Acrónimos

ALU Arithmetic and Logical Unit. 10, 12

ARM Advanced RISC Machine. 1

CSR Control, Status and Register. 5, 16

HDL Hardware Description Language. 6

IDE Integrated Development Environment. 17, 20

IoT Internet of Things. 1, 4

ISA Instruction Set Architecture. 1, 4

ISS Instruction Set Simulator. 7

NOP No Operation. 16, 20

RAW Read After Write. 12

RISC Reduced Instruction Set Computing. 9

RTL Register Transfer Level. 7, 9

TLM Transaction-Level Modeling. 7, 9

VHDL VHSIC (Very High Speed Integrated Circuit) e HDL (Hardware Description Language). 6, 7, 9, 21

WAR Write After Read. 12

WAW Write After Write. 12

Glosario

arquitectura No contexto da informática, refírese ao deseño e estructura dun conxunto de circuítos e outros compoñentes dos que se compón un sistema. 11

benchmarks Examen que se realiza coa fin de comprobar que un programa funciona sen erros e producindo a saída correcta. 17

bits Unidade mínima de información en informática, é un díxito do sistema binario, polo que pode valer 0 ou 1. 4

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. 17

hazard Risco producido por unha dependencia RAW,WAR ou WAW. Pode chegar a causar erros na execución dun programa informático. 12

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. 17

tests Proba mediante a cal se revisa que o funcionamento dun programa é o esperado. Tipícamente, busca simular casos reais de execución. 17

Bibliografía

[1] P. Valerio, "Reshaping the Landscape of IoT with RISC-V," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://web.archive.org/web/20250622211928/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 23 de xuño de 2025. [En liña]. Dispoñible en: https://web.archive.org/web/20250622212131/https://www.xataka.com/moviles/risc-v-necesitaba-dar-paso-gigante-para-competir-arm-acaba-hacerlo-gracias-a-google

[3] rars, "RARS – RISC-V Assembler and Runtime Simulator," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://github.com/TheThirdOne/rars

[4] RISC-V International, "Spike," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://github.com/riscv-software-src/riscv-isa-sim

[5] Màrius Montón, "RISC-V-TLM," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://github.com/mariusmm/RISC-V-TLM

[6] Chisel, "Chisel," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://www.chisel-lang.org/

[7] ChipVerify, "Introduction to verification," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://www.chipverify.com/tutorials/verification

[8] J. R. Scott, "RISC-V Designs," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://johnrscott.github.io/rvdocs/single-cycle/0.1.0/verification/verification.html

[9] Wikipedia, "RISC-V," consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/RISC-V

[10] 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

[11] RISC-V International, “RISC-V Ratified Extensions,” consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://riscv.org/specifications/ratified/

[12] , “,” consultado o 23 de xuño de 2025.

[13] ——, “,” consultado o 23 de xuño de 2025.

[14] “Ieee standard for systemverilog–unified hardware design, specification, and verification language,” *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[15] , “,” consultado o 23 de xuño de 2025.

[16] Wikipedia, “Register-Transfer Level,” consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://en.wikipedia.org/wiki/Register-transfer_level

[17] Wikipedia, “Segmentación de instrucciones,” consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/Segmentaci%C3%B3n_de_instrucciones

[18] RISC-V International, “RISC-V Benchmarks,” consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://github.com/riscv-software-src/riscv-tests

[19] GCC GNU, “Extended ASM using the GNU Compiler Colection (GCC),” consultado o 23 de xuño de 2025. [En liña]. Dispoñible en: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html