Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Computação

Departamento de Engenharia Elétrica

Laboratoire TIMA

Équipe AMfoRS

Self-Adaptative Approximate System-On-a-Chip

SysAx Project

Development of the Hardware Platform

Autor:

Gabriel Villanova N. Magalhães

Campina Grande, Paraíba

Data: 2018

Self-Adaptative Approximate System-On-a-Chip

Development of the Hardware Platform

Autor:

Gabriel Villanova Novaes Magalhães

Orientador:

Prof. D.Sc. Gutemberg Gonçalves dos Santos Júnior

Prof. D.Sc. Gutemberg Gonçalves dos Santos Júnior

Prof. D.Sc. Marcos Ricardo Alcântara Morais

Componentes da banca

Relatório de estágio obrigatório apresentado no curso de Engenheria Elétrica, como requisito parcial para obtenção do título de Bacharel em Engenheria Elétrica na Universidade Federal de Campina Grande (UFCG).

Campina Grande, Paraíba
Data: 2018

"Nós somos o que fazemos repetidamente. A excelência, portanto, não é um ato, mas um hábito." - Aristóteles.

Resumo: *Approximate Computing* é um tema bastante recente e que tem tomado a atenção dos pesquisadores, principalmente pelo aumento de softwares do tipo RMS e aplicações IoT, que processam dados com erro intrínseco, permitindo portanto processamento computacional de forma aproximada. Entretanto, a pesquisa nessa área ainda precisa de muito avanço, um deles, é construir uma plataforma computacional que permita testar ideias de AxC com bastante produtividade, em todo o espectro da Engenharia da Computação, do transistor aos Sistemas Operacionais Multiprocessados. O objetivo principal desse trabalho é estabelecer uma plataforma para que os pesquisadores da área possam testar suas ideias, treinar pessoas e avançar na pesquisa. Para isso, várias plataformas computacionais gratuitas foram avaliadas, sendo a mais promissora, o projeto da University of Berkeley, o *Rocket Chip SoC Generator*. Esse projeto, apesar de bastante completo para o que a pesquisa demanda, não possui uma documentação que permita entender e usar a plataforma e toda a sua capacidade de forma produtiva. Esse trabalho cria uma documentação abordando de forma gradativa quase tudo que essa plataforma oferece. No final, todo o conhecimento adquirido é usado para implementar e testar um Coprocessador na plataforma com técnica de Computação Aproximada.

Palavras-chave: Computação Aproximada, Engenharia da Computação, RISC-V, Rocket Chip, Chisel, FIR.

Abstract: Approximate Computing is a very recent issue that has been attracting the attention of the researchers, mainly due to the increase of RMS software and IoT applications, which process data with intrinsic error, thus allowing computational processing of approximate form. However, the research in this area still needs a lot of progress, one of them, is to build a computational platform that allows to test ideas of AxC with enough productivity, in the entire spectrum of Computer Engineering, from the transistor to Multiprocessed Operating Systems. The main objective of this work is to establish a platform for researchers in the field to test their ideas, train people and advance research. To this end, several free computing platforms have been evaluated, the most promising being the University of Berkeley's the Rocket Chip SoC Generator. This project, although quite complete for what the research demands, does not have documentation that allows to understand and use the platform and all its capacity in a productive way. This work creates documentation by gradually addressing almost everything that this platform offers. In the end, all knowledge acquired is used to implement and test a Coprocessor on the platform with Approximate Computing technique.

Keywords: Approximate Computing, Computer Engineering, RISC-V, Rocket Chip, Chisel, FIR.

"Eu dedico esse trabalho primeiramente a Deus, pois sem Ele nada disso seria possível. Aos meus pais, George e Angela, que sempre acreditaram no meu potencial, possibilitando a realização do sonho de me tornar Engenheiro. A minha namorada Mirtys, por todos os incentivos e cuidados. E a todos os meus professores, em especial, o Prof. Gutemberg Júnior e Prof. Marcos Morais, por me mostrar, de todas as formas, o caminho da excelência."

Lista de Figuras

1 Logo TIMA. 2
2 AMfoRS. 3
3 Qualidade de imagem com aritmética exata, aproximação dos 8 bits LSB e implementação de largura de bit reduzida (fonte: [1]). 6
4 AxC. 9
5 Compilador gerando códigos para executar parte em Processador parte em Unidade de Processamento Neural (Fonte: [3]). 10
6 Exemplo de processador com circuitos precisos e não precisos (Fonte: [3]). . . 11
7 Somador de precisão configurável (Fonte: [3]). 12
8 Logo PULP-platform. 13
9 Arquitetura do PULP. 14
10 Arquitetura simplificada do PULPino. 15
11 Arquitetura simplificada do PULPissimo. 16
12 Logo Nios II. 17
13 IP MicroBlaze. 18
14 Visão geral dos componentes do Rocket Chip. 19
15 Logo lowRISC. 21
16 Arquitetura simplificada lowRISC. 21
17 Logo OpTiMSoC. 22
18 Ecossistema do Rocket Chip. 25
19 Componentes do SoC Rocket Chip [14]. 26
20 Visão geral da arquitetura do Rocket Chip. 27
21 Diretórios das implementações dos componentes de hardware do Rocket Chip. 28
22 Implementação Chisel do Rocket Tile. 30
23 The Rocket Core pipeline. 31
24 Códigos dos componentes do Rocket Core. 32
25 Microarquitetura do Rocket Core [21]. 35
26 Arquitetura de hierarquia de memória do Rocket Chip simplificada. 37
27 Rocket Custom Coprocessor Interface (fonte: [24]). 39
28 RoCC instruction format. 41
29 Módulos CLINT e PLIC conectados ao Rocket Core [23]. 51
30 Mapa de Memória do Módulo CLINT, fonte [23]. 53
31 Mapa de Memória do Módulo PLIC em sistema 64 bits, fonte [23]. 54
32 Códigos do bootloader do Rocket Chip. 56
33 Arquitetura do sistema de depuração do RISC-V. 59
34 Arquivos de implementação do DTM. 60
35 Arquivos de implementação do JTAG. 60

36 Logo Chisel 62
37 Fluxo de compilação de um programa Chisel. 64
38 Registrador p/ Exemplo 1. 75
39 Registrador p/ Exemplo 2. 75
40 Registrador p/ Exemplo 3. 76
41 ULA para exemplo com condicional. 79
42 Circuito meio somador. 81
43 Circuito somador completo usando dois meio somadores. 81
44 Waveform do teste da ULA. 87
45 Infraestrutura de diretórios p/ projetos Chisel. 88
46 RTL Identificador de Sequência. 90
47 FSM para Identificador de Sequência. 90
48 Diretório desenvolvimento projeto identificador de sequências. 90
49 Waveform 1 testbench FSM. 94
50 Waveform 2 testbench FSM. 94
51 Fluxo de compilação cruzada RISC-V. 98
52 Arquitetura do model de simulação Spike. 99
53 Árvore de arquivos para compilação cruzada. 100
54 Saída de compilação cruzada do exemplo Spike. 101
55 Mapa de endereços gerados com implementação do RoccExConfig. 103
56 DTS do exemplo RoccExConfig - A. 104
57 DTS do exemplo RoccExConfig - B 105
58 Fluxo de geração do emulator 106
59 Interface de acesso ao SoC Rocket Chip. 108
60 Arquitetura do Rocket Chip Tethered. 109
61 Verificando mensagem escrita na região de memória MMIO. 112
62 Arquitetura de Filtro FIR Serial. Fonte: [40]. 115
63 Arquitetura de Filtro Serial FIR AxC. 117
64 Interface do componente FIR Serial AxC. 118
65 Arquitetura do circuito comparador do FIR Serial AxC. 118
66 Modelo de programação para o RoCC FIR AxC, sem interrupção. 120
67 Modelo de programação para o RoCC FIR AxC, com interrupção. 121
68 Coprocessor FIR AxC Architecture. 122
69 Máquina de estado implemtando interface RoCC. 123
70 Extensão de ISA RISC-V com Instruções do FIR AxC Desenvolvido. 123
71 Fontes Chisel e Verilog gerarados. 124
72 Coeficientes e Responta ao Impulso de Filtro Passa Baixas. 125
73 RoCC FIR AxC executando filtragem de Onda Quadrada. 126
74 RoCC FIR AxC executando filtragem de ECG. 126

75 Instruções RoCC adicionadas ao binário gerado. 129
76 RoCC FIR AxC embutido no Rocket Chip. 130
77 FIFO recebendo as amostras. 131
78 Banco de Registradores recebendo valores do SW. 132

Lista de Tabelas

1 Técnicas de Computação Aproximada em diferentes camadas [3]. 10
2 Descrição de recursos OpTiMSoC. 22
3 Pacotes de Código do Rocket-Chip Generator [15] 29
4 Parâmetros do processador Rocket Core. 34
5 L1 Cache parameters and caractheristics. 37
6 Sinais do Core Control. 41
7 Register Mode - Sinais de controle. 41
8 Register Mode - Sinais de resposta. 42
9 RoCC sinais de requisição de memória. 42
10 RoCC sinais de resposta da memória. 43
11 AUTL acquire signals. 44
12 AUTL grant signals. 44
13 RoCC FPU Request signals. 45
14 RoCC FPU Response signals. 45
15 RoCC CSR signals. 45
16 RoCC Opcodes. 46
17 Mapa de Memória Default do Rocket Chip. 61
18 Construtores do tipo Bool. 66
19 Operadores que retornam tipo Bool. 67
20 Construtores do tipo Bits. 67
21 Extensão de operadores lógico da Tabela 19. 68
22 Construtores dos tipos UInt e SInt. 69
23 Operadores aritméticos. 69
24 Conversão de tipos. 72
25 sintaxe tipo Vec. 72
26 Construtores de Registradores. 74
27 Argumentos construção do Verilog. 84
28 API Chisel Tester. 86
29 Argumentos construção do teste. 87
30 Comandos SBT p/ compilação Chisel. 89
31 Revisao dos diretórios do Rocket Chip. 95
32 Revisao dos diretórios do Rocket Chip. 96
33 Opções de comandos do C-Emulator. 107
34 Extensão de ISA com instruções de Filtro FIR AxC. 119
35 Banco de Registrador do FIR AxC 119
36 Registrador de Status 120
37 Coeficientes de 8 bits do Filtro Passa Baixas. 125

Lista de Códigos

1 Downgrade in Rocket Chip git repository. 25
2 Part of RocketTile.scala. 30
3 Implem. do processador WithNBigCores no Código subsystem/Configs.scala. . 33
4 Implem. do processador WithNSmallCores no Código subsystem/Configs.scala. 33
5 Parte do Código subsystem/Configs.scala p/ instanciar e config. RoCCs. . . . 46
6 Parte do Código subsystem/Configs.scala para remover mecanismo de coerência de cache 49
7 Parte do Código subsystem/Configs.scala para adic. binário do bootloader. . . 55
8 Parte do Código system/Configs.scala para modificar endereço da SRAM. . . . 57
9 Parte do Código system/Configs.scala para modificar endereço da DRAM. . . 57
10 Mux 4:1 em Verilog. 63
11 Uma possível abstração para Mux. 63
12 Uma possível abstração para Mux 4:1. 63
13 Exemplo de uso para tipo Bool. 66
14 Exemplo de operadores retornando Bool. 67
15 Exemplo de uso para tipo Bits. 67
16 Exemplo de operadores com tipo Bool. 68
17 Exemplo de uso do tipo UInt e SInt. 69
18 Exemplo de operadores aritméticos. 70
19 Exemplo de criação de porta, fio e tipo inferido. 70
20 Exemplo de uso de constantes. 71
21 sintaxe construção de Enum. 71
22 Exemplo de conversão de tipo. 72
23 Exemplo uso de Vec. 72
24 Exemplo uso de Bundles. 73
25 Exemplos de implementação de circuitos elementares. 73
26 Verilog gerado por Chisel ex. 1. 75
27 Exemplo 1 de Reg. em Chisel. 75
28 Exemplo 2 de Reg. em Chisel. 75
29 Verilog gerado por Chisel ex. 2. 75
30 Verilog gerado por Chisel ex. 3. 76
31 Exemplo 3 de Reg. em Chisel. 76
32 Outra forma de de escrever reg. do exemplo 3. 76
33 Tipos de atribuições. 77
34 Estrutura When [36] 77
35 Estruturas Swithc [36] 77
36 Exemplo de circuito usando condicional When. 78

37 Exemplo de circuito usando condicional Switch. 79
38 Classe Module. 80
39 Instanciação de módulos e parametrização. 80
40 Implem. meio somador. 81
41 Implementação do somador completo com 2 meio somadores. 82
42 Exemplo com classe BlackBox. 83
43 Verilog gerado do Exemplo com classe BlackBox. 83
44 Função Main p/ geração de Verilog. 84
45 Função Main p/ geração de Verilog. 85
46 Função Main p/ geração de Simulador. 86
47 Teste básico para circuito da ULA. 87
48 Comandos para instalação do JDK no linux. 88
49 Comandos para instalação do SBT no linux. 88
50 Script build.sbt. 89
51 Implementação da FSM do circuito identificador de sequências. 91
52 Comandos para compilação da FSM identificador de sequências. 92
53 Implementação do teste do circuito identificador de sequências. 92
54 Comandos para compilação do testbench do circuito identificador de sequências. 94
55 Comandos para exemplo Spike. 101
56 Exemplos de implementações do Rocket Chip. 102
57 Exemplos de implementações do Rocket Chip. 103
58 Construção de um emulador para configuração DefaultConfig. 110
59 main.c para exemplo com Emulador. 111
60 Emulando o sistema DefaultConfig. 111
61 Abrindo arquivo VCD gerado pelo emulador. 111
62 Parte do código tile/LazyRoCC.scala para instanciar FIR RoCC AxC. 127
63 Parte do código subsystem/Configs.scala para instanciar FIR RoCC AxC. . . 128
64 Módulo topo instanciando RoCC FIR AxC no Rocket Chip. 128
65 Comando para gerar o emulador do SoC com RoCC implementado. 129
66 entry.S. 137
67 syscalls.c. 140
68 main.c. 145
69 link.ld. 146
70 util.h 147
71 enconding.h. 148
72 Variáveis de ambiente para configuração do Rocket Chip. 164
73 Makefile. 165
74 Testbench para RoCC FIR AxC em SystemVerilog. 166
75 main.c para testar RoCC FIR AxC. 170

Lista de Abreviações

RMS	Recognition, Mining and Synthesis
AxC	Approximate Computing
PNSR	Peak Signal to noise ratio
BER	Bit Error Rate
SoC	System-On-a-Chip
HW	Hardware
SW	Software
SDK	Software Development Kit
ASIC	Application Specific Integrated Circuits
HDL	Hardware Description Language
CPU	Central Processing Unit
ISA	Instruction Set Archicteture
BOOM	Berkeley Out-of-Order
ISS	Instruction Set Simulator

Sumário

Resumo iv

Abstract v

Dedicatória vi

Lista de Figuras ix

Lista de Tabelas x

Lista de Códigos xii

Lista de Abreviações xiii

I Estrutura de Acolhimento e Projeto SysAx 1

1 Estrutura de Acolhimento 2
- 1.1 Laboratório TIMA . . . 2
- 1.2 Equipe AMfoRS . . . 3

2 Projeto SysAx 5
- 2.1 Contexto . . . 5
- 2.2 Objetivos . . . 6
- 2.3 Trabalhos propostos . . . 7

II Estudos Iniciais 8

3 Introdução ao *Approximate Computing* 9
- 3.1 Técnicas HW/SW de *Approximate Computing* . . . 10
- 3.2 Conclusão . . . 12

4 Avaliação de Plataformas de Prototipagem de SoCs 13
- 4.1 PULP-platfom . . . 13
- 4.2 Nios II . . . 17
- 4.3 MicroBlaze . . . 18
- 4.4 Rocket Chip SoC Generator . . . 19
- 4.5 LowRISC . . . 21
- 4.6 OpTiMSoC . . . 22
- 4.7 Critério de escolha de plataforma . . . 23

III A Plataforma Rocket Chip Generator e suas Ferramentas 24

5 Rocket Chip SoC Generator 25

6 Visão Geral dos Componentes de Hardware 26
6.1 Rocket Tile 30
6.1.1 Processador - Rocket Core 31
6.1.2 FPU 36
6.1.3 L1 Cache e MMU 36
6.2 RoCC - Rocket Custom Coprocessor Interface [24] 39
6.2.1 Visão Geral 39
6.2.2 Interface Padrão 40
6.2.3 Interface Estendida 43
6.2.4 Implementação do RoCC no Rocket Chip 46
6.3 Barramentos 48
6.3.1 TileLink Bus [25] 48
6.3.2 AMBA AXI4, AHB-Lite e APB Protocols 50
6.4 Controladores de Interrupção 51
6.4.1 CLINT - Core Local Interrupt 52
6.4.2 PLIC - Platform-Level Interruptor Controller 54
6.5 Memórias 55
6.5.1 ROM 55
6.5.2 SRAM - Memória Principal 56
6.5.3 MMIO - Memory Mapped I/O 57
6.6 Dispositivos de E/S 58
6.7 Debug - DTM (Device Transport Module) e JTAG 59
6.8 Mapa de Memória 61

7 Chisel - Uma nova Linguagem de Descrição de Hardware 62
7.1 Introdução (Overview) 62
7.2 Tipos de Dados (Datatypes) e Operadores 66
7.2.1 Bool 66
7.2.2 Bits 67
7.2.3 UInt e SInt 68
7.2.4 Constantes 70
7.2.5 Enum 71
7.2.6 Conversão de Tipos 72
7.2.7 Tipos compostos: Vec e *Bundle* 72
7.2.8 Conclusão: Lógica Combinacional e Fios (Wires) 73
7.3 Lógica Sequencial: Registradores, Clock e Reset 74

7.4 Semântica . . . 77
7.4.1 Atribuições . . . 77
7.4.2 When e Switch . . . 77
7.5 Construindo Módulos . . . 80
7.5.1 Hierarquia de Módulos e Parametrização . . . 80
7.5.2 Classe *BlackBox*, conectando IPs Verilog . . . 82
7.5.3 Classe geradora de Verilog . . . 84
7.6 Chisel Testbenches . . . 86
7.7 Preparando o ambiente de programação Chisel . . . 88
7.8 FSM, um identificador de sequência binária . . . 90

8 Demais Ferramentas do Rocket Chip **95**
8.1 Ferramentas de Desenvolvimento de SW - riscv-tools/ . . . 96
8.1.1 RISC-V GNU Compiler Toolchain . . . 97
8.1.2 Simulador de ISA RISC-V Spike . . . 98
8.1.3 Construindo um “Hello World!” e executando no Spike . . . 100
8.2 Gerando Verilog de SoC Rocket Chip - vsim/ . . . 102
8.3 Emulador - emulator/ . . . 106
8.3.1 Bootloader e Rocket Chip Tethered . . . 108
8.3.2 Exemplo de uso do Emulador . . . 110

9 Conclusão **113**

IV Desenvolvimento de Hardware AxC no Rocket Chip **114**

10 Filtro FIR AxC **115**
10.1 Filtro FIR Serial . . . 115
10.2 Filtro FIR em paradigma AxC . . . 116

11 Coprocessador FIR AxC **118**
11.1 Interface de Filtro Serial FIR AxC . . . 118
11.2 ISA para Coprocessador FIR AxC . . . 119
11.3 Banco de Registradores . . . 119
11.3.1 Registrador de *Status* . . . 120
11.4 Arquitetura . . . 122
11.4.1 Módulo de Controle . . . 123
11.4.2 Testbench em SystemVerilog e Resultados . . . 124
11.5 Integração do RoCC no Rocket Chip . . . 127
11.5.1 Construção de SW e Emulação do Sistema . . . 129
11.5.2 Problemas Enfrentados e Conclusão . . . 130

12 Conclusão **133**

Referências **134**

Anexo A: Códigos p/ compilação cruzada com RISC-V GNU Toolchain **137**

Anexo B: Testbench RoCC FIR AxC **166**

Anexo C: main.c para teste de RoCC FIR AxC **170**

Parte I

Estrutura de Acolhimento e Projeto SysAx

1 Estrutura de Acolhimento

1.1 Laboratório TIMA

Figura 1: Logo TIMA.

TIMA - **Techniques of Informatics and Microelectronics for integrated systems Architecture**, é um laboratório de pesquisa pública do CNRS (*Centre National de la Recherche Scientifique*), Instituto Grenoble-INP e UGA (*Université Grenoble Alpes*).

Os tópicos de pesquisa do TIMA abrangem a especificação, projeto, verificação, testes, ferramentas CAD e métodos de *design* para sistemas integrados, desde componentes analógicos a digitais, em uma extremidade do espectro, até SoCs multiprocessados junto a sistemas operacionais básicos.

O TIMA está na origem de oito empresas *spin-off*. Entre os mais recentes, o TIEMPO, criada em 2007 para industrializar a tecnologia de *design* de circuitos assíncronos inventada pelo grupo CIS; UROMEMS financiado por Hamid Lamraoui (grupo MNS) para desenvolver seus resultados de PhD em sensores de pressão MEMS, em cooperação com o TIMC e o hospital Parisiense Pitié Salpétrière: a empresa visa resolver problemas de incontinência urinária com um esfíncter artifical mais sofisticado e melhor tolerado.

O TIMA é uma equipe multinacional, com membros e estagiários de todo o mundo. O Laboratório está estruturado em cinco equipes de pesquisa:

- **AMfoRS**: Architectures and Methods for Resilient Systems.
- **CDSI**: Circuits, Devices and System Integration.
- **RIS**: Robust Integrated Systems.
- **RMS**: Reliable Mixed-signal Circuits and Systems.
- **SLS**: System Level Synthesis.

Site oficial `http://tima.univ-grenoble-alpes.fr`.

1.2 Equipe AMfoRS

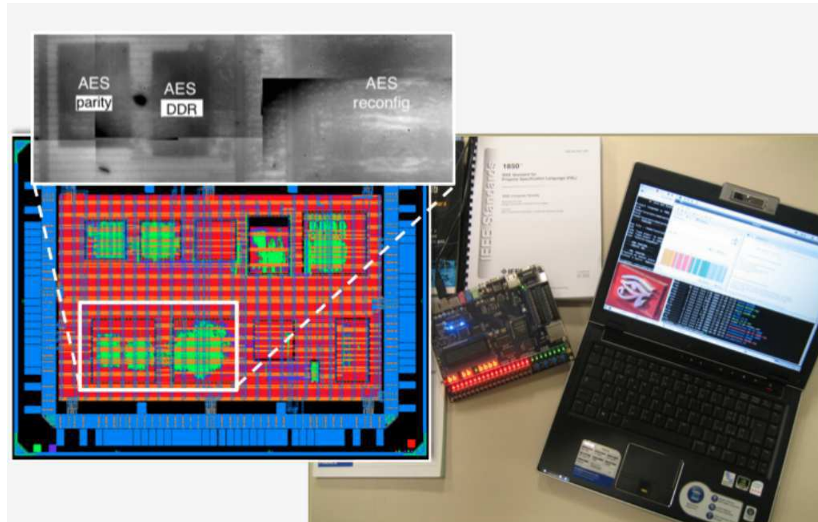

Figura 2: AMfoRS.

A equipe da AMfoRS foi criada em janeiro de 2015. Sua principal experiência é em verificação e resiliência de sistemas. Esta perícia vem das habilidades conjuntas de parte dos membros dos grupos ARIS e VDS anteriores. O AMfoRS visa aumentar a sinergia entre atividades de projeto, verificação e validação de sistemas integrados confiáveis e seguros, com foco em componentes digitais.

A equipe aborda os desafios cruciais relacionados ao que é resumido como "sistemas resilientes", ou seja, a garantia de que os sistemas não se comportarão de maneira inesperada e que proporcionarão um nível substancial de robustez e segurança.

Atingir um nível significativo de segurança requer o *design* de componentes criptográficos otimizados e robustos que fornecem proteção contra ataques de hardware. Além do desenvolvimento de tais blocos, vários métodos e ferramentas para resiliência são necessário, para garantir melhor confiabilidade e segurança.

As soluções "alvo" visam realizar previsão de envelhecimento, protegendo circuitos digitais contra falhas e dando-lhes a possibilidade de adaptar dinamicamente seu comportamento em caso de erros. Soluções analíticas também são tratadas para análise de confiabilidade, ferramentas para a instrumentação automática de sistemas de hardware com monitores e para testar esses sistemas instrumentados, além de ferramentas especializadas para melhorar o projeto de circuitos 3D.

Como os sistemas estão se tornando cada vez mais complexos, há também uma necessidade crucial de métodos eficientes para a análise em nível de sistema. O trabalho em andamento e futuro da equipe inclui *debug* e testes online para System-On-Chips embutidos em seu ambiente funcional.

Principais temas de pesquisa da equipe:

- Multi-level specification and verification of hardware/software on-chip architectures: formal and semi-formal approaches
- System-level modeling, analysis and testing.
- Dependability of integrated systems: fault detection/tolerance, on-line monitoring, self-adapting and self-healing circuits.
- Dependability evaluations: fault injections and alternatives (analytical or formal approaches), prediction of ageing effects on lifetime.
- Security of integrated systems: cryptographic accelerators, counter-measures against hardware attacks.

O estágio em questão foi desenvolvido junto a essa equipe. Mais informações sobre ela pode ser encontrada no site `http://tima.imag.fr/amfors/`. Informações sobre as demais equipes estão disponíveis em `http://tima.imag.fr/`.

2 Projeto SysAx

2.1 Contexto

Durante as últimas décadas, a indústria de semicondutores mostrou muitas melhorias em termos de eficiência energética dos dispositivos produzidos: projetos multiprocessadores, componentes de baixa potência (*ultra-low power*), entre outros. No entanto, como descrito em [2], o consumo de energia de sistemas de computacionais ainda está crescendo rapidamente e em um ritmo alarmante. Além disso, um grande número de aplicativos, geralmente chamados de RMS (do inglês, Recognition, Mining and Synthesis), surgiram e estão ganhando cada vez mais popularidade. Dos sistemas portáteis e Internet of Things (IoT) aos grandes centros de dados, eles representam agora uma parcela significativa e crescente dos recursos computacionais globais e do consumo de energia [3]. Indicando que é preciso melhorar a eficiência energética dos futuros dispositivos de silício. Isso é realmente crítico, já que a tecnologia de semicondutores continua diminuindo para transistores nanométricos: melhores desempenhos, mas ao preço de uma crescente ineficiência energética [9]–[10].

Felizmente, aplicativos como o RMS geralmente apresentam propriedades de resiliência de erro intrínsecas [11]. De fato, esses aplicativos não precisam fornecer um resultado único ou exato. Por exemplo, em aplicativos de vídeo, alguns tipos de erros, como a quantização adaptativa local, podem ser facilmente tolerados, contanto que o erro permaneça abaixo do limiar visual da percepção visual humana. Com base nessas observações, nos últimos anos, uma solução muito promissora conhecida como "Computação Aproximada" (ou AxC - Approximate Computing) [2], [12], está ganhando cada vez mais interesse na comunidade científica, tanto na indústria quanto na academia. O AxC baseia-se na observação intuitiva de que, ao executar a computação exata, é necessária uma quantidade elevada de recursos, permitindo a aproximação seletiva ou a violação ocasional da especificação, o que proporciona ganhos de consumo de energia superiores a uma ordem de magnitude. Ou, para a mesma quantidade de consumo, os desempenhos podem ser melhorados. Várias aplicações de AxC foram pesquisadas, tais como análise de dados, computação científica, multimídia, processamento de sinais, aprendizado de máquina e assim por diante.

Olhando para o estado da arte, as técnicas de AxC propostas podem ser classificadas como: software AxC (por exemplo, iterações de algoritmos reduzidas, gerenciamento de tarefas do SO, entre outros), arquitetura AxC (por exemplo, processadores com blocos aritméticos aproximados, entre outros) ou circuitos AxC (por exemplo, tensão de alimentação reduzida, lógica imprecisa, entre outros).

Muitas técnicas são propostas, mas para serem mais amplamente aceitas, a AxC precisa de mais pesquisas sobre gerenciamento de qualidade. Na verdade, a maioria das abordagens é orientada por componente ou camada para cada uma das métricas dedicadas: PNSR (Peak Signal to noise ratio) para aplicações de processamento de imagem e vídeo, BER (Bit Error Rate) para componentes de hardware aproximados, etc. No entanto, um sistema aproximado requer uma avaliação do impacto de cada componente de cada camada (local) na precisão global da aplicação. É preciso, portanto, definir quais são os Metadados adequados (atributos de qualidade/precisão) e quão coerentes, correlacionados e simplificados são as métricas associadas a serem aplicadas em todas as camadas significativas do sistema.

2.2 Objetivos

O projeto SysAx proposto aqui tem como objetivo final o desenvolvimento de um SoC auto-adaptativo composto por técnicas de hardware e software de Computação Aproximada. A ideia é construir um sistema que modifique seu contexto de um modo preciso para um modo impreciso e vice-versa, executando diferentes aplicativos RMS, tais como reconhecimento de imagens e/ou algoritmos de aprendizado de máquina.

Grosso modo, pretende-se avaliar a relação entre computação com aproximações, qualidade do resultado e desempenho do sistema. Por exemplo, em processamento de imagem, o quanto de erro pode ser aceito sem que aja degradação da imagem em relação ao que o olho humano percebe. Aproximar cálculos desse grupo de aplicações implica na alteração de alguns parâmetros computacionais, como velocidade de processamento, diminuição de *overhead* de software e hardware, menor consumo de energia, entre outros. É esse tipo de avaliação e conclusão que a pesquisa pretende formalizar.

Figura 3: Qualidade de imagem com aritmética exata, aproximação dos 8 bits LSB e implementação de largura de bit reduzida (fonte: [1]).

Um outro objetivo do estágio é educacional. Isto é, pretende-se usar a plataforma de pesquisa também como uma plataforma de ensino, com intuito de auxiliar a compreensão dos estudantes a relação entre hardware e software no desenvolvimento de sistemas integrados.

Portanto, esse projeto de longo prazo possuí dois aspectos principais:

- **Pesquisa**: Desenvolver um SoC para ser usada na exploração de novas ideias de Computação Aproximada, bem como outros campos de pesquisa das equipes do laboratório TIMA, por exemplo para avaliar técnicas de tolerância a falhas (equipe RIS) ou SoCs multiprocessados de sistemas embarcados e prototipagem (equipe SLS).
- **Educação**: Desenvolver um SoC para ser usada como uma ferramenta de prototipagem para criar SoCs personalizados destinados como suporte de educação para o departamento SEI (Systèmes Electroniques Integrés) na escola de engenheria PHELMA.

Devido a existencia de duas linhas de pesquisa (hardware AxC e software AxC) o laboratório contou com dois estágiarios para iniciar a pesquisa, sendo o presente relatório relacionado ao estágio de desenvolvimento de hardware.

2.3 Trabalhos propostos

Motivado por esses desafios, este trabalho surge como uma fase preliminar, para estabelecer, documentar e demonstrar o uso de uma plataforma de prototipação de SoC para dar base a pesquisa e ser usada como ferramenta educacional. Sendo os macro objetivos os seguintes items:

1. Ler e entender o estado-da-arte de técnicas de hardware para projetar circuitos integrados (CI) de Computação Aproximada.
2. Explorar projetos de SoCs de código aberto e escolher o que melhor se adeque aos objetivos do projeto.
3. Documentar a plataforma selecionada explanando: arquitetura, componentes de hardware e ferramentas de desenvolvimento.
4. Experimentar o fluxo de co-projeto de software e hardware usando na plataforma selecionada.
5. Sugerir e desenvolver uma demonstração de uma técnica de Computação Aproximada usando a plataforma.
6. Sintetizar demonstração em FPGA.
7. Estimar consumo de potência para diferentes cenários de aproximação.

Parte II

Estudos Iniciais

3 Introdução ao *Approximate Computing*

Os computadores continuam se tornando cada vez mais rápidos e poderosos, abrindo as portas para processamento de algoritmos e aplicações como aprendizado de máquina (*Machine Learning*), *Deep Learning*, visão computacional, processamento de linguagem natural e outros. Todos esses aplicativos são chamados de aplicativos de reconhecimento, mineração e síntese (RMS). Hoje, essas aplicações são responsáveis por uma parcela significativa de recursos computacionais em todo o espectro da computação, de dispositivos móveis e Internet das Coisas (IoT) até *Data Centers* de grande escala [2].

As aplicações de RMS geralmente apresentam as seguintes características:

- Resiliência ao erro: processam enormes quantidades de dados ruidosos, por exemplo, provenientes de vários sensores que apresentam variações aleatórias.
- Saídas não precisas podem ser aceitáveis: significa que a saída ou alguns dos cálculos intermediários não são necessários para alcançar valores aceitáveis
- Alto consumo de energia devido a cálculos intensos.

À partir da observação dessas características, os pesquisadores e projetistas de sistemas começaram a investigar que ao reduzir a precisão do processamento, pode-se ter ganhos significativos na eficiência energética dos sistemas, dado que essas aplicações toleram erro. Técnicas que visão provar esse conceito são chamadas de *Approximate Computing* ou Computação Aproximada.

O *Approximate Computing* pode ser definida como: "Uma técnica de computação (de forma geral um paradigma) que um resultado possivelmente impreciso em vez de um resultado preciso e pode ser usado para aplicações em que um resultado aproximado é suficiente para o propósito" [3].

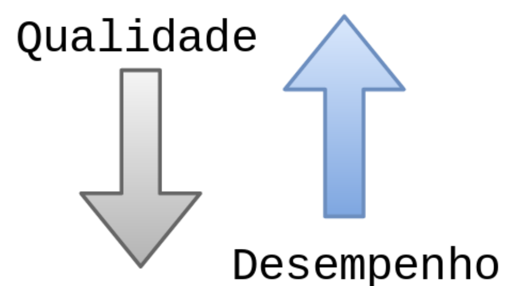

Figura 4: AxC.

A computação aproximada é introduzida apenas em dados não críticos, uma vez que a aproximação de dados críticos (por exemplo, operações de controle) pode levar a conseqüências desastrosas. Exemplos de aplicações são: processamento de vídeo devido às limitações perceptuais dos seres humanos, mecanismos de buscas (Google), entre outros.

Será apresentado, de forma superficial, algumas ideias de software e hardware para computação em paradigma AxC, como uma forma de apresentar o estado-da-arte desse domínio de estudo.

3.1 Técnicas HW/SW de *Approximate Computing*

Na Tabela 1 são apresentadas técnicas de Computação Aproximada nas diferentes camadas dos sistemas, software, arquitetura e circuitos.

Camadas	Técnicas de Computação Aproximada
Software	Loop perforation, Code perforation, Theread fusion, Tunable kernels, Patter reduction.
Arquitetura	Approximate storage, ISA extensions, Approximate accelerators.
Circuito	Imprecise logic, Voltage overscaling, Analog computation, Precision scaling.

Tabela 1: Técnicas de Computação Aproximada em diferentes camadas [3].

No que diz respeito a camada de software, a pesquisa atual busca desenvolver ferramentas de software que possam expressar idéias relacionadas ao AxC. O objetivo dessas ferramentas é estimar como os erros se propagam e quão confiável um sistema pode ser, nas diferentes partes que compõe o desenvolvimento de software: código, compiladores, mecanismos de análise, entre outros.

Algumas linguagens de programação já foram desenvolvidas em pesquisas, como EnerJ [4] e Rely [5], elas fornecem uma abstração para cálculos aproximados, como tipos de dados especiais relacionados a Computação Aproximada, permitindo avaliar diferentes algoritmos que permitam inexatidão em seus resultados. Nos compiladores, os pesquisadores buscam modificar os compiladores adicionando técnicas como *Loop Perforation*, que pode gerar menos iterações de loop em tempo de compilação e, assim, gerar um programa binário que seja executado mais rapidamente. Esses compiladores também podem produzir código de máquina que tenha como alvo hardware específico aproximado, de forma autônoma ou ajudado pelas diretivas do programador. Com a existencia de compiladores abertos, a exemplo do GCC (GNU C Compiler), torna-se viável a pesquisa.

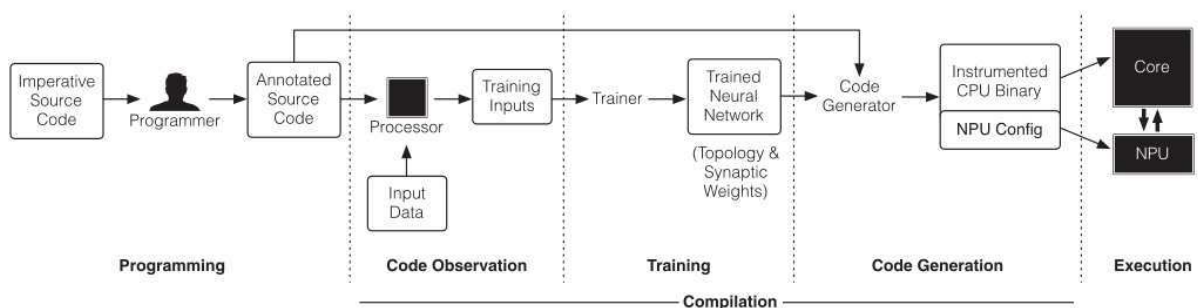

Figura 5: Compilador gerando códigos para executar parte em Processador parte em Unidade de Processamento Neural (Fonte: [3]).

Na camada seguinte, das arquiteturas de processadores, a AxC é investigada em dois grupos distintos. O primeiro grupo tem como objetivo dar suporte à Computação Aproximada executando código tradicional em processadores de uso geral com recurso para executar algumas instruções escolhidas ou segmentos de código no modo aproximado, implicando em uma eficiência de energia (veja Figura 6). O segundo grupo transforma segmentos aproximados do código tradicional em um algoritmo neuralmente inspirado em aceleradores, veja Figura 5. Ambos os grupos deve-se ter um compilador ou programador para identificar ou anotar segmentos de códigos que possam ser cálculados de forma aproximada [3].

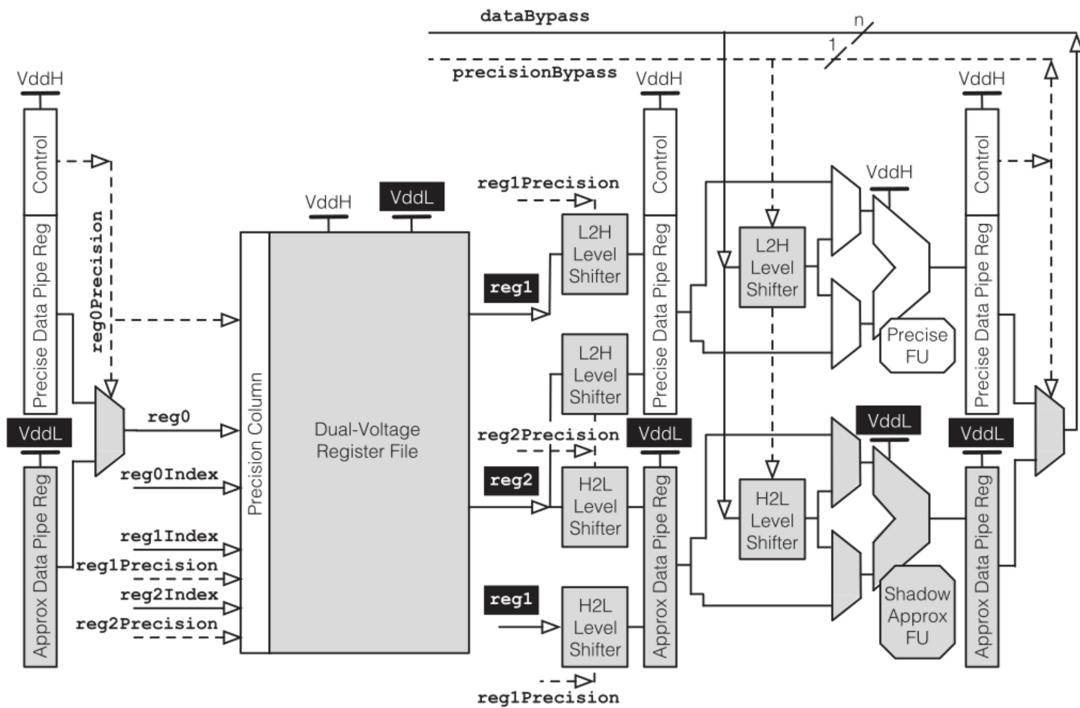

Figura 6: Exemplo de processador com circuitos precisos e não precisos (Fonte: [3]).

Na camada de mais baixo nível, existem várias linhas de pesquisa que estudam arquiteturas de armazenamento e memória aproximadas, assim como circuitos aritméticos aproximados, como: somadores, multiplicadores, divisores, ponto flutuante, etc. O objetivo principal é reduzir a sobrecarga de hardware, e consequentemente acelerar processamento e diminuir consumo de energia elétrica.

Em armazenamento e memória aproximados, em vez de armazenar valores de dados exatos, eles podem ser armazenados de forma aproximada, por exemplo, truncando os bits inferiores em dados de ponto flutuante. Outro método é aceitar uma memória menos confiável. Para isso, DRAMs e eDRAMs, a taxa de atualização pode ser reduzida e, em SRAMs, a tensão de alimentação pode ser também reduzida [6], [7]. Maior largura de banda do canal de memória [8]. Em geral, qualquer mecanismo de detecção e correção de erros deve ser desativado, diminuindo o *overhead* de hardware.

Em circuitos aritméticos as técnicas variam de redução de tensão (o que parece não dar ganhos significativos de consumo de energia [3]) e especulação de valores. Um exemplo é apresentado na Figura 7 em que a precisão do circuito somador é configurável, isto é, vários sub-somadores podem aceitar o transporte previsto ou a propagadação normal do *carry* sendo os resultados parciais são combinados para formar a saída final aproximada [3]. Dado que muitas aplicações do tipo RMS usam operações aritméticas, essa abordagem para promissora.

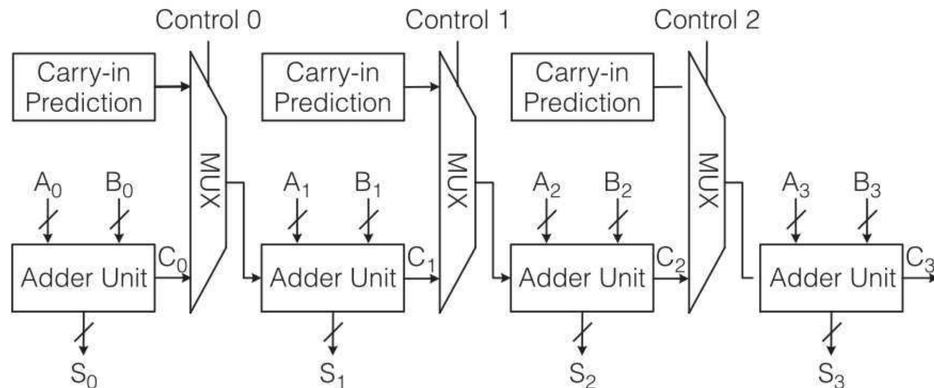

Figura 7: Somador de precisão configurável (Fonte: [3]).

3.2 Conclusão

A pesquisa em Computação Aproximada ainda precisa ser bastante aprofundada. Tendo em vista que as abordagens estão presentes em todo espectro da criação de Sistemas Computacionais (de circuitos aritméticos ao software), é preciso estabelecer uma ferramenta que permita avaliar e desenvolver novas técnicas de AxC em todas as direções. É nesse ponto que esse trabalho tem seu objetivo principal, preparar uma plataforma computacional que permita explorar e desenvolver ideias AxC.

Na próxima seção, é apresentado alguns dos sistemas avaliados durante o estágio e qual plataforma se mostrou mais promissora para os objetivos futuros de pesquisa no Laboratório TIMA.

4 Avaliação de Plataformas de Prototipagem de SoCs

Nessão seção, serão apresentadas as plataformas de prototipagem de *System-On-Chip* que foram avaliadas no estágio. O objetivo dessa atividade foi escolher a plataforma computacional mais adequeada para torna-se instrumento de base na pesquisa, ou seja, que ofereça maiores vantagens para os objetivos que a pesquisa apresenta: desenvolvimento de ideias de Computação Aproximada e plataforma de ensino de Sistemas Embarcados.

4.1 PULP-platfom

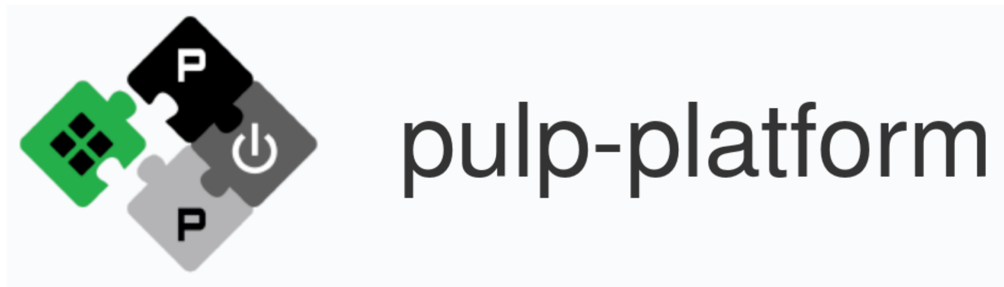

Figura 8: Logo PULP-platform.

O PULP-platform é um projeto que iniciou-se nos grupos: Integrated Systems Laboratory (IIS) da ETH Zürich e Energy-efficient Embedded Systems (EEES) da University of Bologna, com intuito de explorar novas ideias de arquiteturas de computadores para processamento com Ultra-Baixa Potência, termo mais comumente referenciado no mundo acadêmico por *Ultra Low-Power Processing*. O projeto busca desenvolver plataformas computacionais para desenvolvimento escalável [1] de hardware e software visando sempre a eficiência energética.

O *framework* desenvolvido do projeto encontra-se disponível em `https://github.com/pulp-platform` e já conta com algumas plataformas, destacando-se:

- PULP - Parallel Ultra-Low-Power
- PULPino
- PULPissimo
- bigPULP
- ARIANE

[1] Um Sistema Escalável é um sistema que ao ser acrescido de novos compomentes aumenta seu desempenho. Sistema escalável implica desempenho.

As unidades de processamento do projeto, desenvolvidas pelo grupo, são implementações de algumas variações da ISA aberta RISC-V, sendo elas: o processador RI5CY (32 bits) com 4 níveis de *pipeline* e implementação de ISA RV32IMFC, o zero-riscy, um processador com 2 níveis de *pipeline* de 32 bits e ISA RV32IMC e o processador ARIANE com 6 níveis de pipeline e implementação de ISA RV64G (G referindo ao conjunto completo da ISA), capaz de executar sistemas Linux. Cada uma dessas implementações são usadas como unidades de processamentos nos SoCs, isto é, o PULP, PULPino, etc.

O PULP [2] é um cluster [3] que pode ser síntetizado com os processadores zero-riscy e RI5CY. É uma arquitetura avançada de Microcontroladores, com suporte a multiprocessamento e periféricos como I2S, I2C, SPI e UART, veja Figura abaixo.

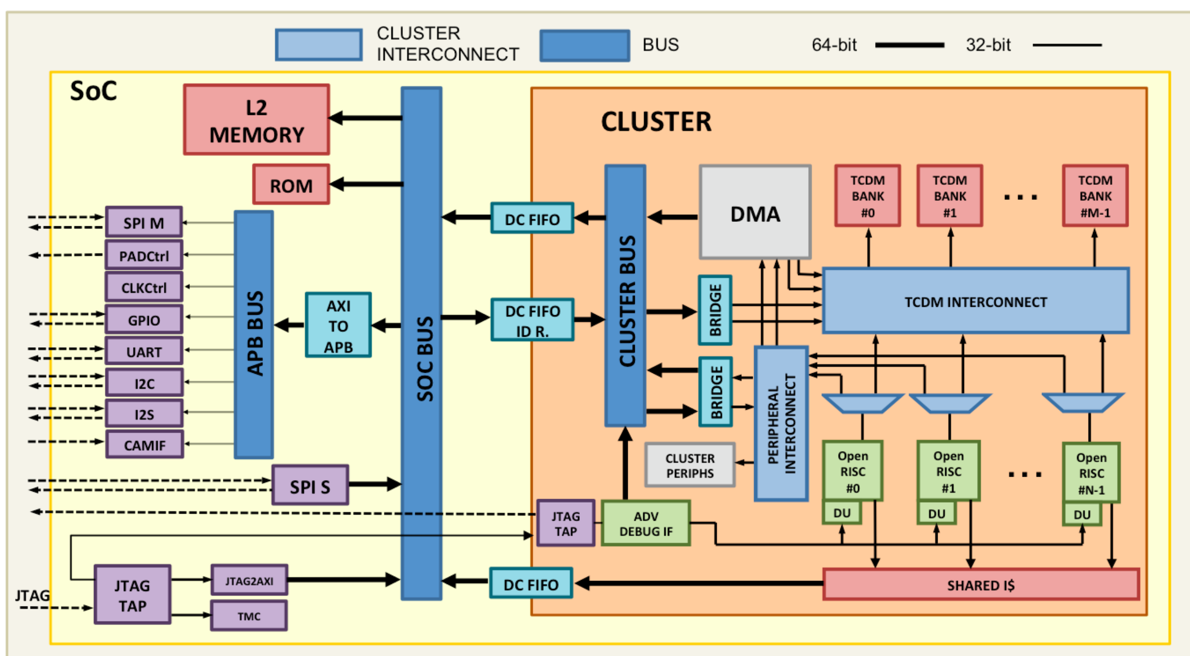

Figura 9: Arquitetura do PULP.

Em resumo o PULP tem as seguintes caraterísticas:

- RI5CY ou zero-riscy como processador principal
- micro-DMA
- *Hardware Processing Engines*, ou máquina de processamento em HW, em tradução livre
- Computação paralela por cluster
- Periféricos
- Unidade de detectação de evento (Event Unit)
- SDK (*Software Development Kit*)

[2] Repositório oficial: `https://github.com/pulp-platform/pulp`

[3] Um cluster consiste em computadores fracamente ou fortemente ligados que trabalham em conjunto, de modo que, em muitos aspectos, podem ser considerados como um único sistema (Fonte: Wikipédia `https://pt.wikipedia.org/wiki/Cluster`).

Já o PULPino [4] é um Microcontrolador de baixo consumo de energia com apenas uma unidade de processamento. Quando seu processador está inativo, ele pode ser colocado em um modo de baixa energia, onde apenas uma unidade de evento simples está ativa, fazendo com que o sistema consuma o mínimo de energia. Uma unidade de evento especializada ativa o processador no caso de um evento/interrupção de forma configurável. O PULPino foi sintetizado como ASIC (*Application Specific Integrated Circuits*) em tecnologia UMC 65nm em janeiro de 2016 e ele possui suporte completo a debug. A arquitetura é apresentada na Figura 10.

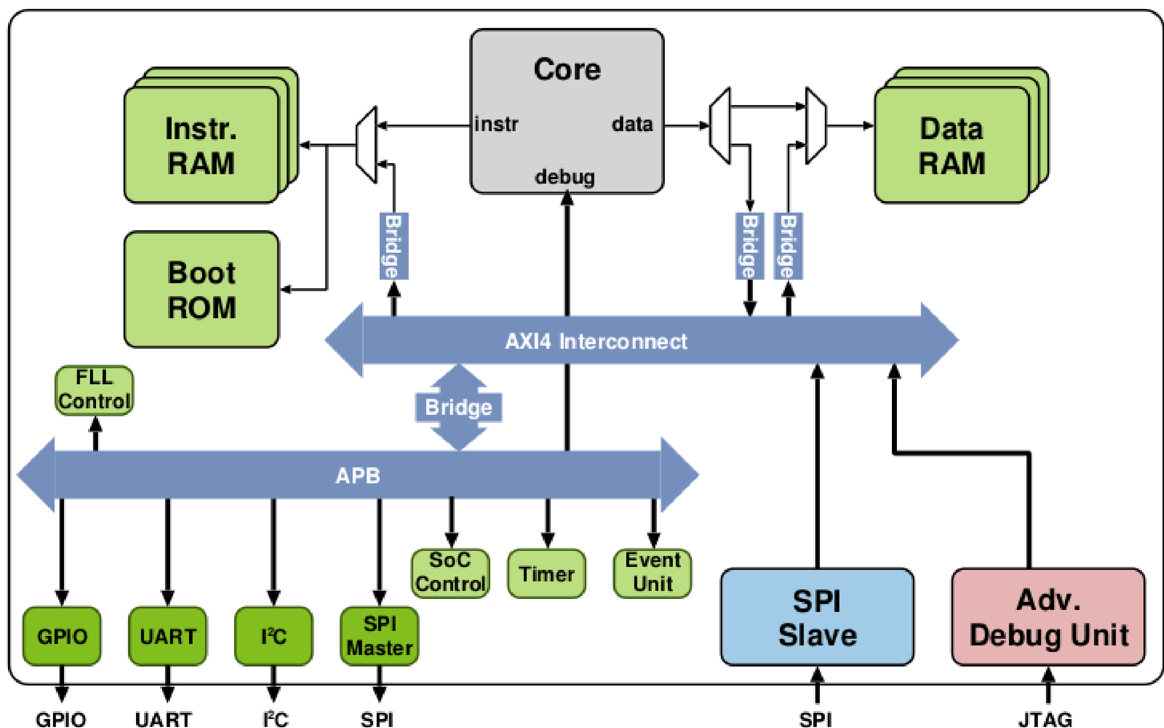

Figura 10: Arquitetura simplificada do PULPino.

Como características principais, destacam-se:

- RI5CY ou zero-riscy como processador principal (único processador)
- Sistema de baixo consumo de energia (*Low-Power*)
- Unidade de detectação de evento (Event Unit) em modo *low-power*
- Periféricos (inclusive JTAG para depuração)
- Suporte a interrupções
- SDK

[4]Repositório oficial: `https://github.com/pulp-platform/pulpino`

O PULPino e PULPissimo [5] são bastantes similares, ambos são projetos de Microcontroladores e possuem um único processador, que pode ser o RI5CY ou zero-riscy. A principal diferença é que o PULPissimo incrementa ao SoC: micro-DMA, HWPE (*Hardware Processing Engines*, ou máquina de processamento, em hardware em tradução livre) e um novo subsistema de memória. Sua arquitetura é apresentada na Figura 11.

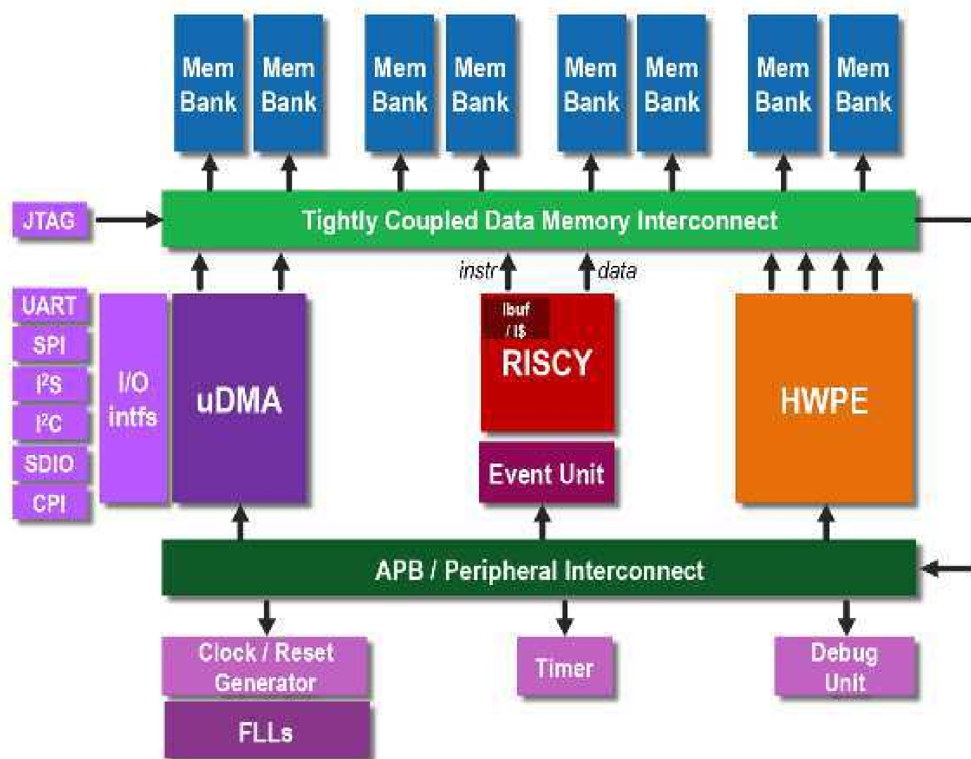

Figura 11: Arquitetura simplificada do PULPissimo.

O SoC com processador ARIANE não fora avaliado, porém pode ser facilmente explorado em seu repositório oficial `https://github.com/pulp-platform/ariane`. Todos esses projetos foram escritos com HDL (*Hardware Description Language*) SystemVerilog e usufrui de algumas ferramentas para geração de código de máquina, permitindo: desenvolvimento, simulação, síntese e avaliação das plataformas.

Interfaces de Entrada e Saída (E/S), permitem adicionar periféricos próprios sem muito esforço (sem alterar drasticamente a plataforma), porém, componentes internos como unidade de ponto flutuante, multiplicadores, micro-DMA, cache, etc, não foram projetados para serem modificados facilmente, tornando a tarefa de adicionar implementações próprias para teste e avaliação (no caso em questão, de Computação Aproximada) de ideias um trabalho não trivial.

[5] Repositório oficial: `https://github.com/pulp-platform/pulpissimo`

4.2 Nios II

Figura 12: Logo Nios II.

Plataformas computacionais para desenvolvimento paralelo de HW e SW podem ser desenvolvidas com *framework* oferecido pelas ferramentas FPGAs, tendo em vista que elas possuem IPs como: processadores modificáveis, periféricos, interfaces, etc. Isso permite prototipação rápida de SoCs, possibilitando testes e avaliação de novas ideias de HW e SW (para a pesquisa em questão, permite avaliar ideias de AxC). Portanto, foi avaliado o Nios II da empresa Intel FPGAs.

O Nios II [6] é uma arquitetura de processador RISC de 32 bits projetada especificamente para a família de FPGAs Intel/Altera. Ele incorpora muitos recursos, tornando-o adequado para uma ampla gama de aplicações de computação. O Nios II permite:

- Definição de instruções personalizadas: como forma de atingir metas de desempenho
- Periféricos padrões (disponibilizados como IPs)
- Periféricos personalizados (definidos pelo usuário)
- Unidade de Gerenciamento de Memória (MMU)
- Execução de Sistemas Operacionais (SO), como o kernel Linux
- SDK

Existem várias versões do Nios II, sendo destacando-se:

- **Nios II/e**: baixo custo para FPGAs pequeno porte, sem necessidade de licença
- **Nios II/s**: manter um equilíbrio entre desempenho e custo, necessita de licença
- **Nios II/f**: projetado para desempenho máximo às custas do aumento de área do processador, necessita de licença

As grandes desvantagens de escolher essa plataforma é a incapacidade de alterar componentes internos ao processador, como, unidades aritméticas (ponto flutuante, multiplicadores, somadores), cache, DMAs, etc, que podem ser alvo de interesse para pesquisa. Além disso, algumas versões que possuem maior desempenho são pagas.

[6]Site oficial: `https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html`

4.3 MicroBlaze

Figura 13: IP MicroBlaze.

A CPU MicroBlaze é uma família de configurações de processador RISC de 32 bits, predefinidas e modificáveis, disponível para se usar na maioria das FPGAs Xilinx. O projetista do sistema pode aproveitar o Xilinx SDK sem custo, baseado no Eclipse, e sem experiência prévia em FPGA, para desenvolver diferentes soluções com MicroBlaze, que é capaz de fornecer CPUs para projetos de Microcontroladores, para processamento em tempo real ou capaz de executar sistemas Linux.

Partindo de uma configuração, é possível uma personalização adicional de uma variedade de opções do processador, além disso um catálogo de periféricos são disponível, como PWMs, UARTs, DMAs, controladores de interrupção, interfaces seriais, entre outros, que podem ser adicionados na construção de SoCs. Também é possível adicionar periféricos próprios para construção do sistema, com intuito de satisfazer as necessidades específicas do projeto.

Em resumo, o MicroBlaze possui:

- Tamanho de cache configurável
- Opção de pipeline de 3, 5 ou 8 estágios
- Periféricos (IPs Xilinx ou IP próprio)
- Unidade de gerenciamento de memória configurável
- Interfaces de barramento configurável
- Instruções para operações de multiplicação, divisão e ponto flutuante podem ser adicionadas/removidas
- Suporta SOs que requerem paginação e proteção com base, como o kernel do Linux
- Suporta SOs com uma proteção simplificada e modelo de memória virtual, por exemplo, FreeRTOS ou Linux sem suporte a MMU

Várias personalizações permitem que um desenvolvedor faça as trocas de design apropriadas para um conjunto específico de hardware host e requisitos de software de aplicativos, mas, de forma semelhante ao Nios II, componentes internos não podem ser modificados.

4.4 Rocket Chip SoC Generator

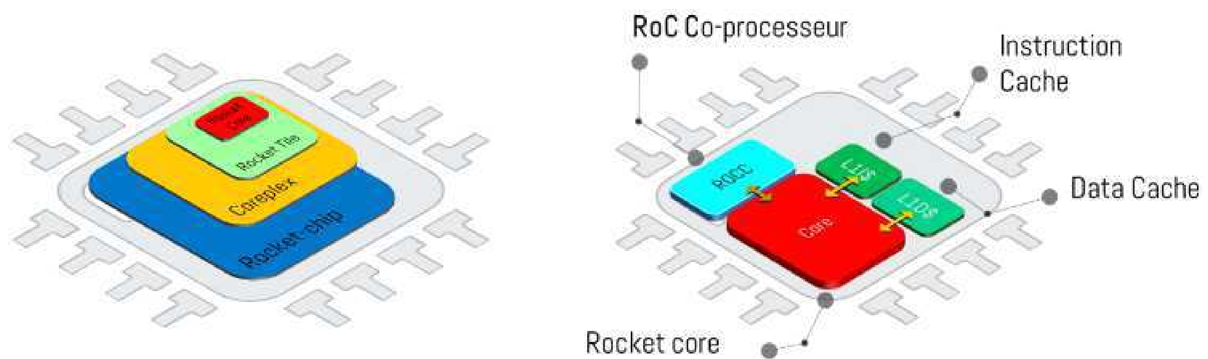

Figura 14: Visão geral dos componentes do Rocket Chip.

O Rocket Chip é um gerador de SoC de código aberto que emite RTL sintetizável. Ele aproveita a Linguagem de Descrição de Hardware Chisel para compor uma biblioteca de geradores sofisticados para processadores, caches e interconexões em um SoC integrado. O Rocket Chip tem as seguintes características:

- Possui implementações de processadores de uso geral: o Rocket Core com 5 estágios de pipeline e Z-Scale com 3 estágios, ambos configuráveis
- Processador Fora de Ordem (BOOM - *Berkeley Out-Of-Order*) também configurável
- Implementa variações de ISA aberta RISC-V permitindo adicionar/remover extensões de ISA RISC-V no sistema
- Possui interface de depuração JTAG (pode ser facilamente removida)
- Permite configurar tamanho de barramento: 32, 64 ou 128 bits
- Permite adicionar, remover ou configurar memória cache
- Permite integrar arquiteturas com multi-processadores heterogêneas para ganhos adicionais de eficiência
- Suporta a integração de aceleradores personalizados na forma de extensões de conjunto de instruções: coprocessadores (RoCC - *Rocket Custom Coprocessor*) ou novos processadores totalmente independentes
- Controladores de Interrupção locais e globais
- Temporizadores e interrupção de software para cada processador
- Pode construir pequenos SoCs (como Microcontroladores) até sistemas computacionais complexos como servidores e clusters
- Fornece várias ferramentas de teste/simulação/validação: simulação de ISA, simulação funcional QEMU, emulação acelerada com precisão de ciclo e ferramentas para exportar RTL para implementação real de hardware FPGA.

A principal desvantagem do Rocket Chip é o fato de não possuir uma versão *Untethered* (não ligada), i.e., os SoCs gerados do projeto são conectados a processador que faz o papel da memória principal e bootloader do sistema. Isso faz com que as aplicações que se executam em FPGA tornem-se bastante lentas, por serem gerenciadas por outro processador, impossibilitando o teste e avaliação de desempenho de soluções em FPGAs. Um outro ponto importante, é a necessidade de aprender uma nova HDL, a linguagem Chisel. E que o projeto não possuem periféricos de E/S como I2C, I2S, UART, PWM, etc, necessitando de implementação caso necessário.

No entanto, mesmo com os inconvenientes, o gerador de SoC ainda parece interessante para pesquisa e fins de educação, já que possue uma boa infraestrutura que permite a construção de diferentes configurações de SoCs (de simples Microcontroladores a sistemas multi-processados), além de permitir novas implementações ou modificações da sua gama de componentes e avaliar usando as ferramentas de simulação e emulação. Sobre o problema da versão *Untethered*, ela pode ser solucionada com alguns esforços de programação.

O Rocket Chip [7] foi fabricado onze vezes e produziu protótipos funcionais de silício para executar programas bare-metal, bem como inicializar o Linux. É uma fonte aberta, bem mantida e livre de encargos para implementação e fabricação.

[7]Repositório oficial: `https://github.com/freechipsproject/rocket-chip`

4.5 LowRISC

Figura 15: Logo lowRISC.

O lowRISC [8] é uma organização sem fins lucrativos que trabalha em estreita colaboração com a University of Cambridge e a comunidade *open-source*. O projeto se baseia principalmente no fluxo de desenvolvimento do Rocket Chip e todas as ferramentas são muito semelhantes, inclusive os processadores usados no lowRISC é o Rocket Core, implementando a extensão RISC-V RV64GC com 5 estágios de pipeline.

O lowRISC suporta personalização, possui um conjunto de periféricos implementados, multi-processamento, entre outros. Apesar de ser derivado do Rocket Chip, os SoCs gerados não são ligados (*Tethered*) a outro processador como no Rocket Chip, permitindo síntese em FPGAs ou ASICs diretamente. O repositório já possui versões de SoC capaz de executar Sistema Operacional Debian e Linux em geral.

O objetivo principal do projeto é construir diferentes plataformas capazes de executar sistemas baseados em Linux com baixo custo de fabricação. Soluções *Ultra-Low Power*, geração de Microcontroladores, entre outros, não são possíveis, o que torna a plataforma pouco flexivel em relação a outras para o ensino de Sistemas Embarcados e como plataforma de pesquisa. Uma visão geral da plataforma é apresentada a seguir.

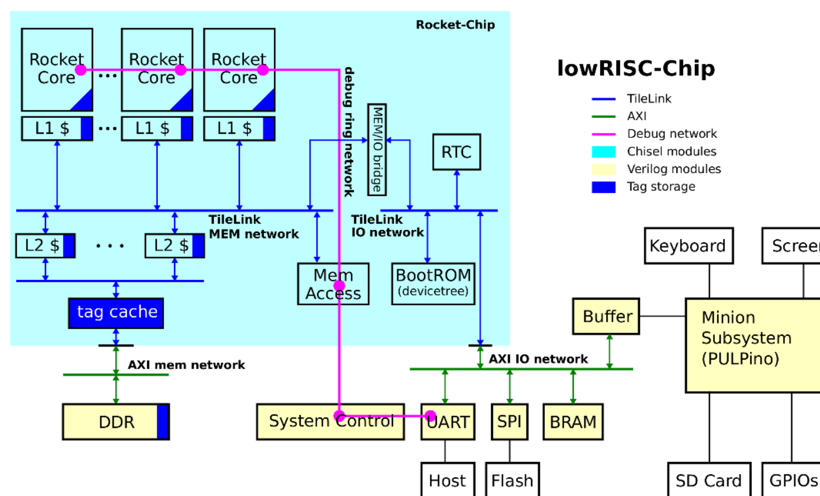

Figura 16: Arquitetura simplificada lowRISC.

[8]Site oficial: `https://www.lowrisc.org/`

4.6 OpTiMSoC

Figura 17: Logo OpTiMSoC.

OpTiMSoC [9] - *Open Tiled Manycore System-on-Chip.* É um *framework* de código aberto escrita principalmente em Verilog que permite construir *System-on-Chips*. Fornece blocos como processadores, memórias, aceleradores de hardware, componentes de E/S, entre outros, todos conectados em arquitetura *Network-on-Chip* [10] (NoC). O sistema resultante pode então ser simulado ou sintetizado em FPGA.

O OpTiMSoC oferece os recursos apresentados na Tabela seguinte:

Recursos	**Descrição**
Toolchain	Consiste em um conjunto de ferramentas usado para compilar programas de software para arquitetura OpenRISC.
Bibliotecas	Fornecem todas as funcionalidades das plataformas OpTiMSoC para os aplicativos. Este inclui drivers de hardware, suporte a tempo de execução, APIs de comunicação e APIs de gerenciamento de tarefas.
Simulação RTL	É o ponto de partida do desenvolvimento. Além disso, a simulação RTL é usada para executar pequenos pedaços de software para teste ou desenvolvimento de drivers e o sistema em tempo de execução.
Verilator	Ferramenta de código aberto que transforma código Verilog em modelos C++ ou SystemC para desenvolvimento de simulações do sistema.
Síntese	Atualmente, a equipe do OpTiMSoC se concentra na síntese de FPGAs. O Xilinx WebPack grátis é suportado, assim como o Synopsys Synplify, mas a licença ainda é necessária em alguns casos.
Interface gráfica	Fornece fácil controle e observabilidade dos sistemas em execução.

Tabela 2: Descrição de recursos OpTiMSoC.

Apesar de ser uma plataforma bastante completa, o objetivo dessa plataforma de pesquisa é o estudo de arquiteturas NoCs.

[9]Site oficial: `https://www.optimsoc.org/`

[10]NoCs, de forma simplificada, são topologias de redes de comunicação entre componentes de SoC, que visão melhorar a eficiência de energia em SoCs.

4.7 Critério de escolha de plataforma

A decisão sobre qual plataforma escolher para ser utilizada, analisou vários critérios, levando em consideração também os objetivos apresentados na seção 2.2. Os critérios estabelecidos foram:

1. Escolher uma plataforma de código aberto *upstream* (disponível para download na Internet) com todos os recursos de hardware e software incluidos, para garantir uma boa acessibilidade e manutenção da plataforma por todos usuários.
2. Geração de arquiteturas de SoCs amplamente personalizáveis, i.e., quantidade de processadores, personalização de memória cache, unidades de aceleração (e.g. coprocessadores) periféricos, etc. O intuito é poder explorar o máximo de componentes de um SoC, de forma produtiva.
3. Possibilitar geração de Microcontroladores de pequeno porte e sistemas capazes de executar sistemas Linux, o objetivo é fornecer uma plataforma mais genérica possível.
4. Possibilitar simulação e síntese das soluções geradas.
5. Possuir uma ISA aberta.

O PULP-platform (PULP, PULPino ou PULPissimo) são microcontroladores projetados para alçancar o baixo consumo de energia. Eles são bem mantidos e documentados, mas sua arquitetura não é personalizável e não são projetados para executar sistemas como Linux.

As plataformas Nios II e MicroBlaze, tem os componentes internos do processador fechados, suas ISAs não são de código aberto e algumas ferramentas de análise não abertas.

O OpTiMSoC atende a maior parte desses critérios, mas é relativamente desatualizado (última versão é 2016) e é baseado na arquitetura OpenRISC, que tem muitas limitações em comparação com o ISA RISC-V (Rocket Chip e lowRISC). Estas limitações são apresentadas em detalhes no manual de especificação RISC-V [13], página 15.

Levando em consideração os argumentos mencionados, ficamos apenas com duas opções lowRISC e Rocket Chip. Pesquisas adicionais e experimentos técnicos mostraram que o lowRISC é baseado em uma versão antiga do Rocket Chip, que não é mais mantida/atualizada. Além disso, ambos não são bem documentados, mas a vantagem do Rocket Chip é que o trabalho é mantido, sendo uma referência nas implementações relacionadas ao RISC-V.

Com base nesses critérios e nessa análise, escolheu-se o Rocket Chip SoC generator, sendo a solução que mais se ajusta enquadra com as necessidades do projeto.

Parte III

A Plataforma Rocket Chip Generator e suas Ferramentas

5 Rocket Chip SoC Generator

Figura 18: Ecossistema do Rocket Chip.

O Rocket Chip é um gerador de System-on-Chip de Código aberto que emite RTL sintetizável. Ele aproveita a linguagem de construção de hardware Chisel para compor uma biblioteca de geradores sofisticados para cores, caches e interconexões em um SoC integrado. O Rocket Chip gera processadores de uso geral que usam a ISA (do inglês, *Instruction Set Archictecture*) RISC-V e fornece tanto um gerador de processador em ordem (Rocket-Core [14]) quanto um fora de ordem (BOOM) . Para projetistas de SoC interessados em usar especialização heterogênea para ganhos adicionais de eficiência, o Rocket Chip suporta a integração de aceleradores personalizados na forma de instrução de extensões, coprocessadores ou cores totalmente independentes.

Em vez de ser uma única instancia de um projeto de SoC, o Rocket Chip é um gerador de SoC, capaz de produzir várias instancias de projeto a partir de uma única fonte de alto nível. A parametrização extensiva torna-o flexível, permitindo a fácil personalização para uma aplicação específica. Ao alterar uma única conFiguração, um usuário pode gerar SoCs que variam em tamanho, desde microcontroladores embutidos até chips de servidores com vários processadores [14].

Em outras palavras, podemos imaginar o Rocket Chip como um grande menu de componentes de um SoC, em que o cliente faz a escolha dos componentes e pode também escolher as características de cada um deles, e no final da respectiva escolha, o SoC é entregue de acordo com as especificações.

O projeto está disponível para download em `https://github.com/freechipsproject/rocket-chip`. No decorrer do relatório, os repositórios contidos neste projeto serão referenciados, fornecendo uma compreensão prática e teórica da plataforma passo a passo. Devido a atualização constante deste repositório foi fixada uma versão, que pode ser conseguida usando o comando:

```
$ cd rocket-chip
$ git checkout 4ba8acb4aa26901899963136704d065a22e36460 -b branch_name
```

Código 1: Downgrade in Rocket Chip git repository.

6 Visão Geral dos Componentes de Hardware

Nesta Seção será apresentada a arquitetura do SoC Rocket Chip, expondo os seus principais componentes, seus barramentos e como tudo se conecta de forma global. Nas próximas subseções, descreveremos esses componentes de uma forma mais detalhada, apresentando também onde encontrar os Códigos de implementação destes componentes e quais são as opções de personalização que cada um oferece.

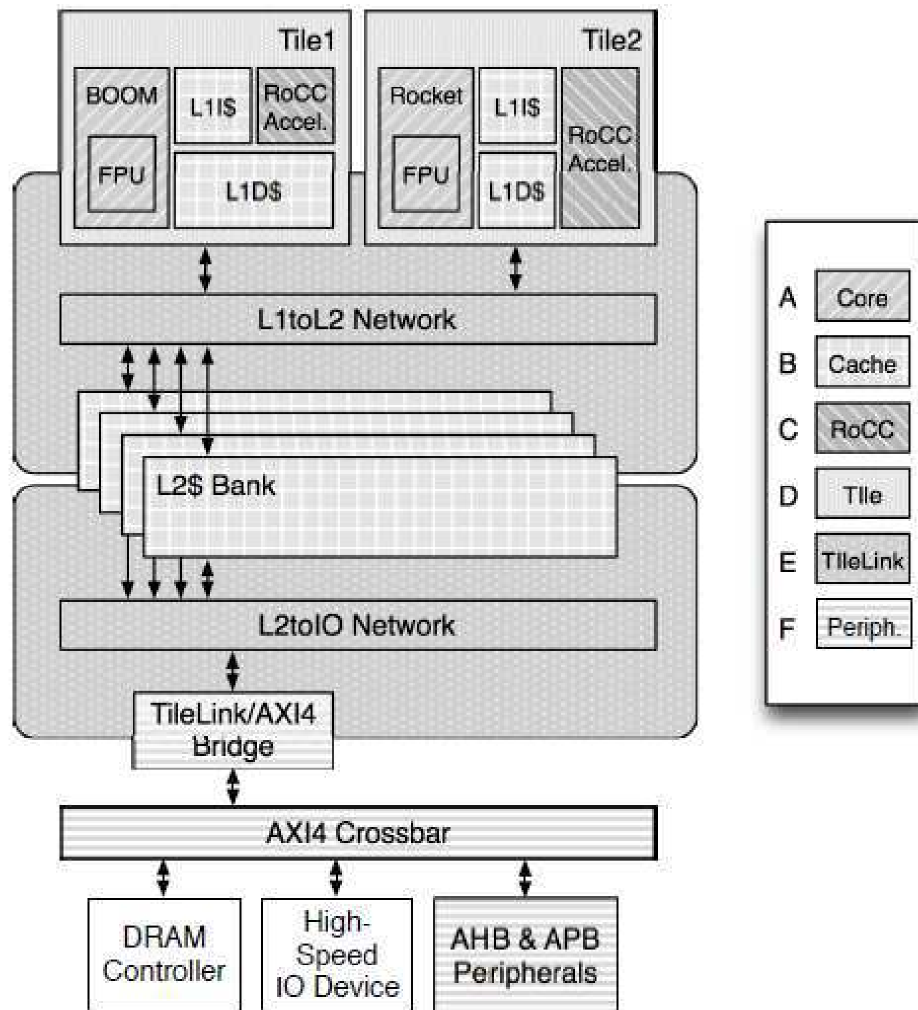

Figura 19: Componentes do SoC Rocket Chip [14].

A arquitetura apresentada na Figura 19 mostra os principais componentes existentes do Rocket Chip e como eles são agrupados e conectados.

O módulo Tile (ou Rocket Tile, letra D na Figura) é um componente que instancia importantes circuitos, como: processador, cache L1, coprocessador (RoCC - Rocket Custom Coprocessor) e FPU. Na Figura acima, o sistema possui dois Tiles, o primeiro (Tile1) possui: processador BOOM, FPU, L1 Cache e RoCC. O segundo Tile (Tile2) é muito semelhante, deferenciado na instancia do processador e nos paramêtros da cache

L1.

Os Tiles são conectados ao barramento TileLink (principal barramento do Rocket Chip), que além de fornecer conectividade entre componentes, implementa o mecanismo de coerência de cache. Esse barramento também possui interface AMBA AXI4 (ou *bridge* TileLink-to-AMBA) e usa isso para conectar outros dispositivos de E/S e memórias, como pode ser observado na Figura.

A Figura 20 é outro exemplo de instanciação do Rocket Chip. Nessa arquitetura mais simplificada o sistema possui apenas um Tile, porém evidencia outros componentes de hardware como o Core Local Interruptor (CLINT), que é o controlador de interrupções locais dos processadores, o Platform Level Interrupt Controller (PLIC), responsável pelo controle das interrupções globais, a memória ROM que contém parte do Código do boot-loader e o dispositivo I/O SimDTM (Debug Transport Module) responsável pela interface E/S de depuração do sistema.

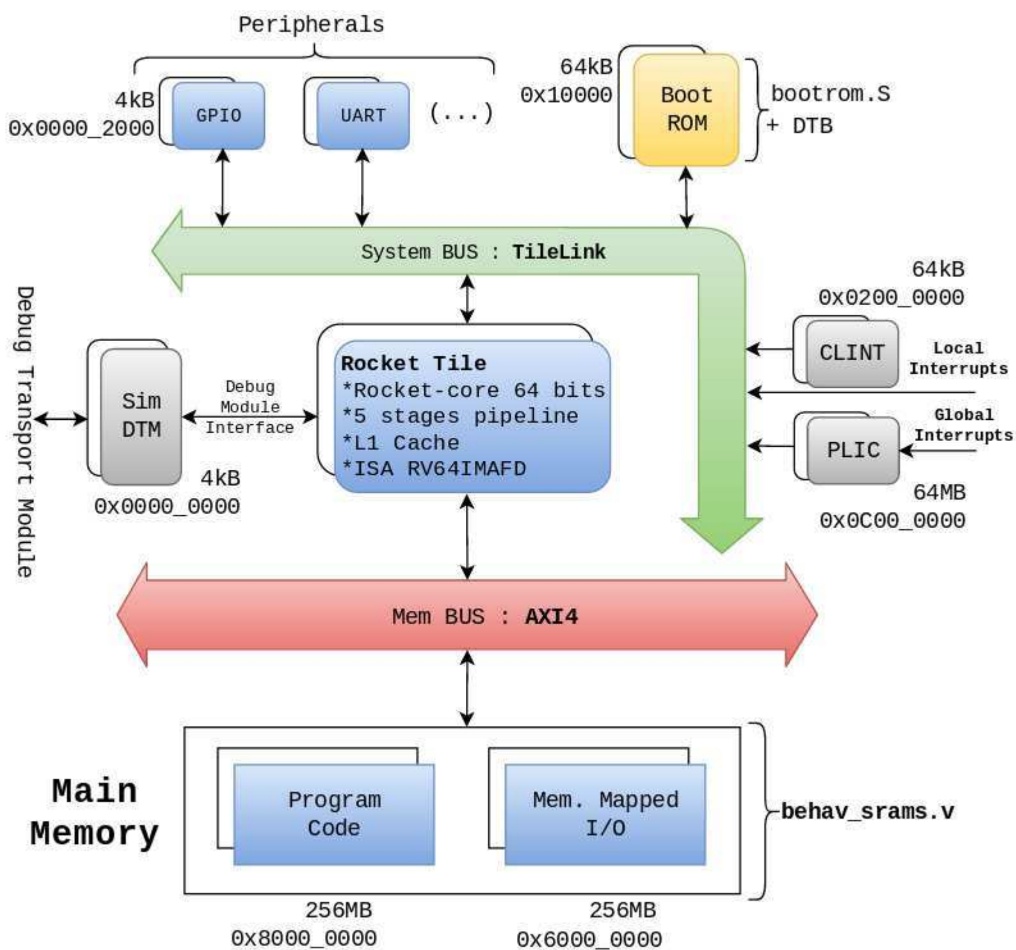

Figura 20: Visão geral da arquitetura do Rocket Chip.

No geral, o Rocket Chip é uma biblioteca de geradores de hardware (processadores, FPU, barramentos, memórias, etc) que podem ser parametrizados e compostos de diversas formas, possibilitando uma larga variedade de projetos de SoC.

No repositório oficial todos os Códigos que implementam esses componentes estão localizados em rocket-chip/src/main/scala,conforme pode ser observado na Figura 21. Cada sub-repositório agrupa partes do gerador de SoC, sendo um responsável pela união de todos os demais (módulo topo), o que possibilita a criação de diferentes soluções de SoC. Uma breve descrição dos componentes é apresentada na Tabela 3.

```
rocket-chip/src/main/scala/
├── amba
│   ├── ahb
│   ├── apb
│   └── axi4
├── config
├── devices
│   ├── debug
│   └── tilelink
├── diplomacy
├── groundtest
├── interrupts
├── jtag
├── regmapper
├── rocket
├── subsystem
├── system
├── tile
├── tilelink
├── unittest
└── util

20 directories
```

Figura 21: Diretórios das implementações dos componentes de hardware do Rocket Chip.

Adiantamos aqui que esse plataforma foi desenvolvida em uma nova linguagem de descrição de hardware (ou HDL - *Hardware Description Language*), o Chisel, ela foi desenvolvida a partir do framework Scala (por isso veremos formato .scala nos arquivos de implementação). Basicamente, essa linguagem foi feita para facilitar o quesito modularização e parametrização nos projetos de hardware e ela também possui capacidade de gerar Código Verilog equivalente do circuito, o que é importante para se aproveitar todas as ferramentas de síntese de ASICs e FPGAs que interpretam Verilog. Enfim, nas próximas subseções mostraremos Códigos de implementação dos componentes com o intuito de apresentar tanto o hardware como o que é fornecido para em termos de personalização, isso nos ajudará a entender como a plataforma foi desenvolvida e se familiarizar com o modo de trabalhar com ela. Na Seção sobre as ferramentas, mais informações sobre essa linguagem serão expostas.

Componentes	Descrição
amba	Este pacote RTL usa diplomacy para gerar implementações de barramento de protocolos AMBA, incluindo AXI4, AHB-lite e APB.
config	Este pacote de utilitários fornece interfaces Scala para conFigurar um gerador através de uma biblioteca de parametrização com escopo dinâmico.
coreplex	Esse pacote RTL gera um coreplex completo reunindo uma variedade de componentes de outros pacotes, incluindo: processadores e Tiles, uma rede de barramento de sistema, agentes de coerência, dispositivos de depuração, manipuladores de interrupção, periféricos externos, cruzadores de clock e conversores da TileLink para protocolos de bus externos (por exemplo, AXI ou AHB).
devices	Este pacote RTL contém implementações para dispositivos periféricos, incluindo o módulo de depuração e vários escravos de TileLink.
diplomacy	Esse pacote de utilitários estende o Chisel permitindo a elaboração de hardware em duas fases, na qual determinados parâmetros são negociados dinamicamente entre os módulos.
groundtest	Esse pacote RTL gera testadores de hardware sintetizáveis que emitem fluxos de acesso aleatório a memória, a fim de testar a hierarquia de memória sem processadores.
jtag	Este pacote RTL fornece definições para gerar interfaces de barramento JTAG.
regmapper	Esse pacote de utilitários gera dispositivos escravos com uma interface padronizada para acessar seus registradores mapeados na memória.
rocket	Este pacote RTL gera o Rocket Core, bem como as caches de instruções e dados L1. Esta biblioteca destina-se a ser usada por um gerador de chip que instancie o processador dentro de um sistema de memória e conecte-o ao mundo externo.
tile	Esse pacote RTL contém componentes que podem ser combinados com processadores para construir blocos, como FPUs e aceleradores.
tilelink	Este pacote RTL usa o diplomacy para gerar implementações de barramento do protocolo TileLink. Ele também contém uma variedade de adaptadores e conversores de protocolo.
system	Esse pacote de utilitários de nível superior (módulo topo) chama o Chisel para elaborar uma conFiguração específica de um coreplex, junto com a garantia de teste apropriada.
unittest	Este pacote de utilitários contém uma estrutura para gerar testadores de hardware sintetizáveis de módulos individuais.
util	Este pacote de utilitário fornece uma variedade de construções Scala e Chisel comuns que são reutilizadas em vários outros pacotes.

Tabela 3: Pacotes de Código do Rocket-Chip Generator [15]

.

6.1 Rocket Tile

O Rocket Tile (ou simplesmente Tile) é um componente que foi projetado para instanciar, personalizar e conectar vários outros sub-componentes, sendo eles: processador, FPU, cache L1 e coprocessador RoCC. Ele também possui uma interface conectada ao processador para receber fontes de interrupções externas.

Os componentes desse módulo encontram-se no repositório tile/, conforme ilustrado na Figura 22.

```
src/main/scala/tile
├── BaseTile.scala
├── Core.scala
├── FPU.scala
├── Interrupts.scala
├── L1Cache.scala
├── LazyRoCC.scala
└── RocketTile.scala
```

Figura 22: Implementação Chisel do Rocket Tile.

O Código abaixo é o módulo topo do componente RocketTile.scala, ou seja, o Código que instancia e conecta os outros módulos. Note ainda que este Código contém uma classe chamada RocketTileParams e que ela possui parâmetros como: core, icache, dcache, rocc, etc. Esses parâmetros chamam outras classes e funções que na verdade são conFigurações dos componentes que compõem o Tile que podem ser modificadas para personalizar o respectivo componente.

Por exemplo, o Código de implementação do processador, o Core.scala, possui uma instancia chamada RocketCoreParams() e nela há opções para personalizar o processador Rocket Core. Nas seções do processador, cache, etc, serão apresentados quais são os parâmetros desses sistemas.

```scala
case class RocketTileParams(
    core: RocketCoreParams = RocketCoreParams(),
    icache: Option[ICacheParams] = Some(ICacheParams()),
    dcache: Option[DCacheParams] = Some(DCacheParams()),
    rocc: Seq[RoCCParams] = Nil,
    btb: Option[BTBParams] = Some(BTBParams()),
    dataScratchpadBytes: Int = 0,
    trace: Boolean = false,
    hcfOnUncorrectable: Boolean = false,
    name: Option[String] = Some("tile"),
    hartId: Int = 0,
    blockerCtrlAddr: Option[BigInt] = None,
    boundaryBuffers: Boolean = false
```

Código 2: Part of RocketTile.scala.

Apesar da possibilidade de customização de um componente no seu próprio Código de implementação, não é dessa forma que se personaliza SoCs nessa plataforma. O Rocket Chip fornece Códigos de nível de hierarquia mais alto que importam consigo quase todos os paramêtros do sistema. O que é mais interessante, pois nesse caso o usuário pode modificar todo o sistema de uma só vez, ao invés de ir até a implementação de cada componente do sistema. Ao decorrer do relatório isso será mostrado.

É importante destacar ainda que para construir sistemas multiprocessados não se instancia vários processadores dentro de um Tile, na verdade se instancia vários Tiles que contém processadores. Foi assim que o Rocket Chip foi projetado, veja o exemplo dual core na Figura 19, tem-se dois Tile.

Na Seção sobre as ferramentas do Rocket Chip serão apresentados exemplos de como construir SoCs com diferentes conFigurações, o que facilitará a compreensão dessa infraestrutura de Códigos apresentada aqui e como manipula-lós. O objetivo desta primeira Seção é apresentar o RocketChip e descrever a organização referente à infraestrutura de Códigos.

6.1.1 Processador - Rocket Core

O processador do Rocket Chip é o Rocket Core. Ele é um processador *in-order* que permite diversos tipos de personalização, como será apresentado adiante. Entretanto, existem outros projetos derivados do Rocket Chip que usam a mesma infraestrutura da plataforma e subsistuem apenas o Rocket Core, eles são o projeto BOOM [16], que implementa um processador fora-de-ordem (Berkeley Out-of-Order Machine) e o Z-Scale [17], que é um projeto que implementa um processador parecido com o Rocket Core, porém menos complexo.

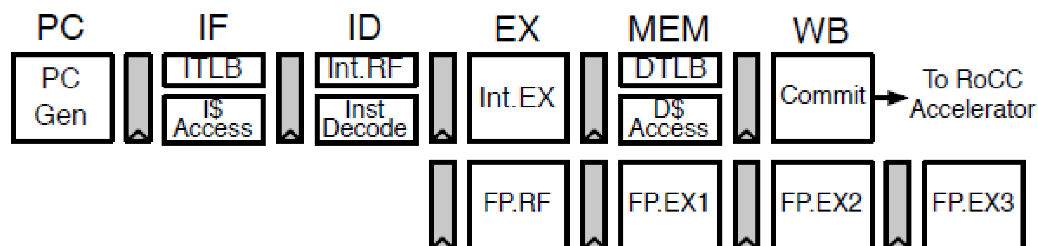

Figura 23: The Rocket Core pipeline.

O Rocket é um processador escalar *in-order* de 5 estágios de pipeline, que implementa as ISAs RISC-V: RV32G e RV64G. Ele possui uma MMU (Memory Management Unit) que suporta memória virtual baseada em página, um cache de dados sem bloqueio e um *front-end* com branch prediction. O branch prediction é configurável e fornecida por

um buffer de destino de branch (BTB, Branch Target Buffer), um Branch History Table (BHT), e uma pilha de endereços de retorno (Return Stack Address, RAS). Para ponto flutuante, o Rocket faz uso das implementações de unidades de ponto flutuante desenvolvidas em Chisel pela Universidade de Berkeley. O Rocket também suporta os níveis de privilégios de usuário (User), supervisor (Supervisor) e máquina (Machine))para mais informações veja manual RISC-V [18], [20]). Vários parâmetros são expostos, incluindo o suporte opcional de algumas extensões ISA (M, A, F, D), o número de estágios de pipeline de ponto flutuante e os tamanhos de cache e TLB. O Rocket também pode ser considerado uma biblioteca de componentes do processador. Vários módulos originalmente projetados para o Rocket são reutilizados por outros designs, incluindo unidades funcionais, caches, TLBs, o Page Table Walker e a implementação de arquitetura privilegiada (ou seja, o arquivo de registro de status e controle) [14]. A sua microarquitetura pode ser visualizada na Figura 25.

A implementação do Rocket encontra-se no diretório rocket/ (ver Figura 24) onde o Código RocketCore.scala é o módulo topo desse processador. Devido a quantidade de elementos possíveis de serem personalizados para implementar diferentes soluções de processador (ISAs diferentes, tamanho do barramento, tamanho de cache, etc) a plataforma Rocket Chip fornece um Código de mais alto nível que contempla toda a modularização possível para esse hardware.

```
rocket-chip/src/main/scala/rocket/
├── ALU.scala
├── AMOALU.scala
├── Breakpoint.scala
├── BTB.scala
├── BusErrorUnit.scala
├── Consts.scala
├── CSR.scala
├── DCache.scala
├── Decode.scala
├── Events.scala
├── Frontend.scala
├── HellaCacheArbiter.scala
├── HellaCache.scala
├── IBuf.scala
├── ICache.scala
├── IDecode.scala
├── Instructions.scala
├── Multiplier.scala
├── NBDcache.scala
├── package.scala
├── PMP.scala
├── PTW.scala
├── RocketCore.scala
├── RVC.scala
├── ScratchpadSlavePort.scala
├── SimpleHellaCacheIF.scala
├── TLBPermissions.scala
└── TLB.scala
```

Figura 24: Códigos dos componentes do Rocket Core.

Esse Código é o Configs.scala que encontra-se no repositório subsystem/. Esse Código fornece três exemplos de diferentes implementações de Rocket Core, criados com as classes: WithNBigCores, WithNSmallCores e With1TinyCore. O que essas classes fazem é sobrescrever as variáveis criadas nos outros módulos para criar novas conFigurações.

Como exemplo, compare os Códigos seguintes que mostram a classe WithNBigCores e WidthNSmallCores implementando, respectivamente, um processador com ISA RV64G, cache L1, FPU e MMU, e outro que não contém FPU (fpu=none) nem MMU (useVM=false), mas que também possui parâmetros de cache modificados.

Deve-se ter em mente que ao se sobrescrever variáveis de um módulo em outro de maior hierarquia, o valor neste é que permanecerá.

```
class WithNBigCores(n: Int) extends Config((site, here, up) => {
  case RocketTilesKey => {
    val big = RocketTileParams(
      core   = RocketCoreParams(mulDiv = Some(MulDivParams(
        mulUnroll = 8,
        mulEarlyOut = true,
        divEarlyOut = true))),
      dcache = Some(DCacheParams(
        rowBits = site(SystemBusKey).beatBits,
        nMSHRs = 0,
        blockBytes = site(CacheBlockBytes))),
      icache = Some(ICacheParams(
        rowBits = site(SystemBusKey).beatBits,
        blockBytes = site(CacheBlockBytes))))
```

Código 3: Implem. do processador WithNBigCores no Código subsystem/Configs.scala.

```
class WithNSmallCores(n: Int) extends Config((site, here, up) => {
  case RocketTilesKey => {
    val small = RocketTileParams(
      core = RocketCoreParams(useVM = false, fpu = None),
      btb = None,
      dcache = Some(DCacheParams(
        rowBits = site(SystemBusKey).beatBits,
        nSets = 64,
        nWays = 1,
        nTLBEntries = 4,
        nMSHRs = 0,
        blockBytes = site(CacheBlockBytes))),
      icache = Some(ICacheParams(
        rowBits = site(SystemBusKey).beatBits,
        nSets = 64,
        nWays = 1,
        nTLBEntries = 4,
        blockBytes = site(CacheBlockBytes))))
```

Código 4: Implem. do processador WithNSmallCores no Código subsystem/Configs.scala.

Em resumo, a Tabela abaixo mostra as principais possibilidades de personalização para o Rocket Core.

Parâmetro	**Variável**	**Opções/Descrição**
Tam. Barramentos	xLen	32 ou 64 bits
FPU	fpu	Adicionar ou Remover
Multiplicador/Divisor	mulDiv	Adicionar ou Remover
Interrupções externas	nExtInts	Quantidade de int. externas suportada
Memória Virtual	useVM	Adicionar ou Remover
Cache	-	Personaliza cache (veremos na Seção cache)
Coprocessador	-	Adic. or Rem. interface (veremos na Seção RoCC)

Tabela 4: Parâmetros do processador Rocket Core.

A cache L1, será detalhada na Seção 6.1.3. Veja que dispositivos como: FPU, Multiplicador e Divisor podem ser adicionados ou removidos do sistema, o que modifica a extensão da ISA (entre as extensões I, M, F, D - ver [18]). Pode-se também modificar o tamanho do barramento 32 ou 64 bits, fazendo com que a ISA torne-se a RV32 ou RV64, respectivamente.

Por fim, o Rocket Core implementa a ISA RISC-V de forma modular, i.e., ele foi projetado visando flexibilidade conforme supramencionado. Neste contexto, para modificar a ISA, o usuário deve realizar atribuições simples às variáveis desejadas, demonstrado o alto poder de personalização de projetos de SoC através do Chisel. Informações adicionais acerca da ISA RISC-V estão disponíveis nos manuais [18], [20].

A forma mais recomendada de se usar o Rocket Chip, é reaproveitando toda essa infraestrutura de Códigos fornecida, utilizando os exemplos como base para um melhor entendimento da plataforma de forma a se projetar o sistema desejado.

O fluxo completo de desenvolvimento, i.e., como criar SoCs distindo e como emular o sistema com binários compilados com o compilador RISC-V, que será apresentado na Seção 8, esclarecendo ainda mais o uso da plataforma e sua alta capacidade de geração de SoCs.

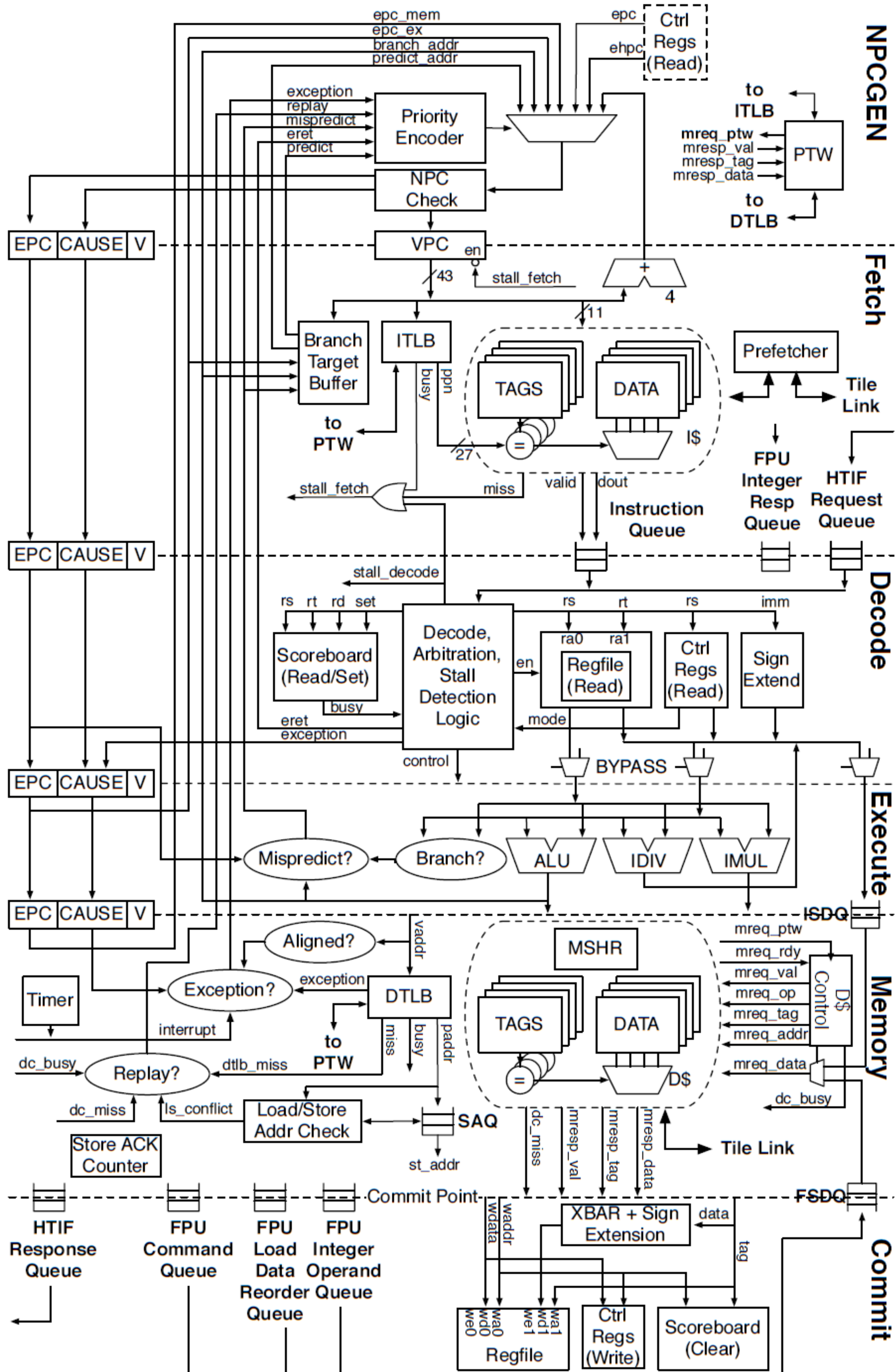

Figura 25: Microarquitetura do Rocket Core [21].

6.1.2 FPU

A FPU fornece suporte de hardware completo para o padrão de ponto flutuante IEEE 754-2008, incluindo tanto a precisão simples (32 bits) quanto a aritmética de precisão dupla (64 bits). A FPU inclui uma unidade de pipeline e uma unidade iterativa de divisão e raiz quadrada, comparadores de magnitude e unidades de conversão de flutuação para inteiros, todos com suporte de hardware completo para subnormais e todos os valores padrão IEEE [22].

A implementação está no diretório rocket-chip/hardfloat/ e tile/ permitindo apenas adicionar ou remover esse módulo no sistema e configurá-o em 32 ou 64 bits (depende da palavra definida para o processador).

Entretanto, caso o usuário queira personalizar esse circuito além do que é fornecido pelo Rocket Chip, pode-se modificar o seu Código de implementação. A desvantagem disso é que esse circuito não possui nenhuma documentação (arquitetura, interface, registradores, etc), , dificultando o processo de modificação.

Observe que ao se adicionar uma FPU no Tile, automaticamente é feita uma extensão da ISA RISC-V. A extensão F - Standard Extension for Single-Precision Floating-Point é adicionada para arquiteturas de 32 bits e a extensão D - Standard Extension for Double-Precision Floating-Point para arquiteturas 64 bits.

6.1.3 L1 Cache e MMU

O Rocket Chip possui componente de Cache L1 de instruções e de dados. Essa memória implementa política de escrita Write-Back e possui duas opções para políticas de substituição: Random Replacement e Pseudo-LRU, além de possibilitar personalização de tamanho de Blocks, Sets e Ways.

Ela possui interface para cada processador dentro do Rocket Tile e interface para o barramento TileLink que implementa a coerência de Caches (no caso de sistemas multicores) e a conecta a níveis mais altos da hierarquia de memória.

O Rocket Chip também possui componente de Virtual Memory. Quando instanciado é adicionado uma Memory Menegement Unit (MMU) no sistema. Esta MMU suporta um espaço de endereço virtual de 39 bits mapeado para um espaço de endereço físico de 50 bits. O Page Table Walker (PTW) de hardware recarrega o cache de conversão de endereços, que pode ser conFigurado com até 128 entradas totalmente associativas [22].

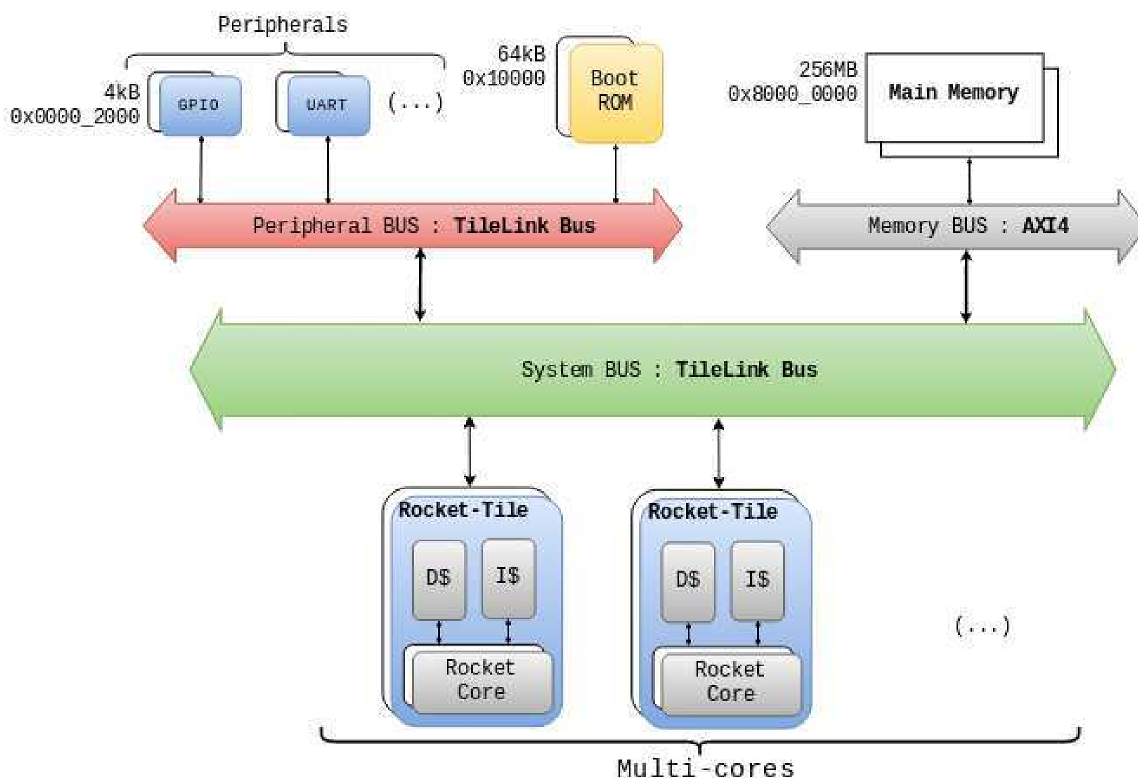

Figura 26: Arquitetura de hierarquia de memória do Rocket Chip simplificada.

No Código 4 é mostrado um exemplo dos parâmetros disponíveis pelo Rocket Chip para Virtual Memory. Existem dois parâmetros: o **useVM** que define se adiciona ou remove a MMU ao sistema, e o **nTLB** que define a quantidade de páginas, sendo o valor default de 32 páginas. A Figura 25 ilustra os TLBs e PTW (Page Table Walker) que representam o mecanismo de VM no Rocket Core.

<table>
<tr><th colspan="2">L1 Data Cache (default parameters)</th><th colspan="2">L1 Instruction Cache (default parameters)</th></tr>
<tr><td>Set</td><td>64</td><td>Set</td><td>64</td></tr>
<tr><td>Blocks</td><td>64 bytes</td><td>Blocks</td><td>64 bytes</td></tr>
<tr><td>Ways</td><td>4</td><td>Ways</td><td>4</td></tr>
<tr><td>Size</td><td>64*64*4=16KB</td><td>Size</td><td>16KB</td></tr>
<tr><th colspan="4">Translation Lookaside Buffer - TLB</th></tr>
<tr><td colspan="2">TLB</td><td colspan="2">32 pages entries (default parameter)</td></tr>
<tr><th colspan="4">Write policies</th></tr>
<tr><td colspan="4">Write-Back</td></tr>
<tr><th colspan="3">Replacement policies options</th><th>Class name implementation</th></tr>
<tr><td colspan="3">Random Replacement (default option)</td><td>RandomReplacement(Int)</td></tr>
<tr><td colspan="3">Pseudo-LRU (Least recently used)</td><td>PseudoLRU(Int)</td></tr>
</table>

Tabela 5: L1 Cache parameters and caractheristics.

O Código 4 também exemplifica como modificar as características da Cache, fornecendo como parâmetros as seguintes variáveis: **rowBits** (equivalente aos Blocks), **nSets** e **nWays**. A Tabela 5 sintetiza as informações de ambos os mecanismos.

O Rocket Chip possui interface para L2 Cache (veja Figura 19), porém a implementação em hardware dela foi excluida da plataforma devido a bugs encontrados. Dessa forma a hierarquia de memória segue o modelo L1 Cache e Memória Principal.

6.2 RoCC - Rocket Custom Coprocessor Interface [24]

A interface RoCC permite integrar coprocessadores ou aceleradores personalizados ao Rocket Core. A implementação desta interface permite criar novas extensões para a ISA RISC-V e é uma ferramenta interessante para explorar paralelismo em software.

Vamos detalhar o RoCC nesta seção, entendendo a interface e onde ela é implementada no repositório do Rocket Chip.

6.2.1 Visão Geral

A interface RoCC possui um conjunto básico de sinais geralmente necessários para aceleradores. Nós nos referimos a eles como a interface RoCC padrão. No entanto, a interface RoCC também fornece algumas extensões configuráveis que podem ser exigidas pelos aceleradores, dependendo da sua funcionalidade. Nós nos referimos a eles como a interface RoCC estendida.

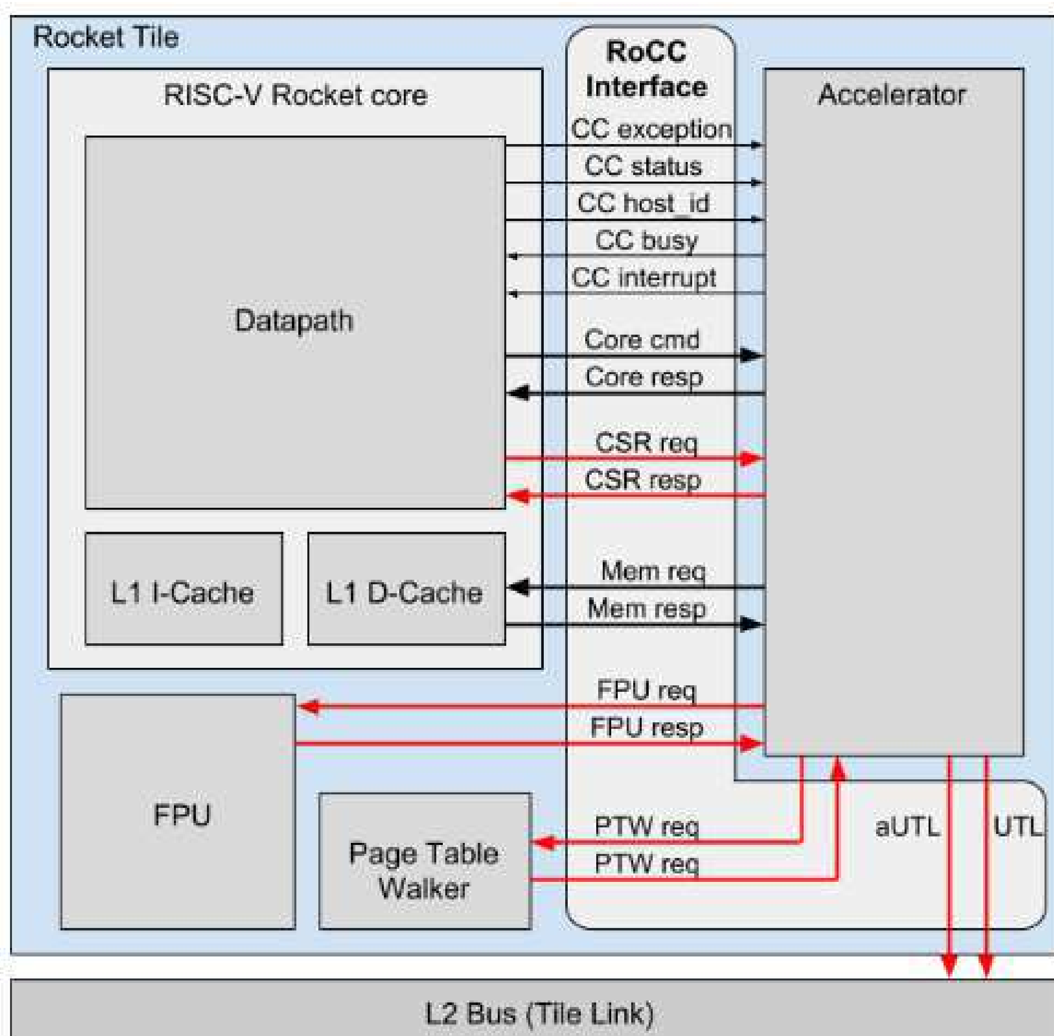

Figura 27: Rocket Custom Coprocessor Interface (fonte: [24]).

A interface padrão do RoCC podem ser classificados nos seguintes grupos de sinais:

- Core control (CC): para coordenação entre um acelerador e um núcleo Rocket Core.
- Register mode (Core): para troca de dados entre um acelerador e um núcleo Rocket.
- Memory mode (Mem): para comunicação entre um acelerador e o cache L1-D.

A interface estendida do RoCC é dividida nos seguintes grupos de sinais:

- Uncached Tile Link (UTL): para comunicação entre uma memória aceleradora & L2.
- Floating Point Unit (FPU): para um acelerador enviar e receber dados de um FPU.
- Control Status Register (CSR): usado por Linux no Core- p/ reconhecer o acelerador.
- Page Table Walker (PTW): para tradução de endereços de um acelerador.

A arquitetura da Figura 27 mostra a interface RoCC conectada nos demais dispositivos. Os prefixos: cc, core, mem, fpu, etc, são usados nos sinais que implementam a interface, além disso tais prefixos são usados para se referir a esses grupos no decorrer dessa seção.

6.2.2 Interface Padrão

A interface RoCC padrão é composta de 3 subgrupos, ou seja, os sinais do modo Controle, Registrador e Memória. Uma lista abrangente dos sinais, com descrições curtas e seus valores padrão, é fornecida do ponto de vista do projetista.

Os principais sinais de controle listados na Tabela 6 garantem a coordenação entre o Rocket Core e o acelerador. Todos os nomes de sinal são prefixados com “cc_to” que denotam o grupo ao qual eles pertencem - Core Control.

Os sinais do Register Mode são compostos pelos subgrupos Comando e Resposta. Todos os nomes dos sinais são prefixados com “core_” para indicar sua origem. Os sinais do Comando são usados pelo processador para enviar instruções ao acelerador e são acionados diretamente pela Instrução RoCC do Rocket Core. A instrução RoCC é apresentada na Figura 28 e indica o tamanho e as posições dos bits para decodificar uma instrução.

A Tabela 7 contém uma descrição do sinais do subgrupo de comando. Uma resposta RoCC ao comando (se esperado) é enviada pelo acelerador usando a interface de resposta. Os sinais de resposta são descritos na Tabela 8 [24].

Direção	Sinal	Valor Default	Descrição
output	cc_busy_o	0	Definido durante solicitações de memória em andamento.
input	cc_status_i	0	Definido quando um processo privilegiado é executado no processador.
output	cc_interrupt_o	0	Sinal de interrupção, causando interrupção em modo de máquina (mcause) com valor 0x800000000000000C (reservado no ISA RISC-V [18]).
input	cc_exception_i	0	Definido pelo processador para acionar o comportamento de exceção.
input	cc_host_id_i	host_id	Usado para distinguir entre pacotes de comando de diferentes hosts. Nota: Bit width = log2(number_of_cores).

Tabela 6: Sinais do Core Control.

Direção	Sinal	Valor Default	Descrição
output	core_cmd_ready_o	0	Sinais de controle.
input	core_cmd_valid_i	0	
input	[6:0] core_cmd_inst_funct_i	funct7	Para diferentes tipos de instruções do acelerador; O valor é a escolha dos designers.
input	[4:0] core_cmd_inst_rs2_i	rs2	ID de registradores de origem [19].
input	[4:0] core_cmd_inst_rs1_i	rs1	
input	core_cmd_inst_xd_i	xd	Definir se o registrador de destino existe.
input	core_cmd_inst_xs1_i	xs1	Define se há registrador de entrada.
input	core_cmd_inst_xs2_i	xs2	
input	[4:0] core_cmd_inst_rd_i	rd	ID regist. de destino.
input	[6:0] core_cmd_inst_opcode_i	0x1, 0x2, 0x3 ou 0x4	O Código de operação da instrução personalizada pode ser usado no caso de vários aceleradores.
input	[63:0] core_cmd_rs1_i	rs1_data	Origem de registradores de dados.
input	[63:0] core_cmd_rs2_i	rs2_data	

Tabela 7: Register Mode - Sinais de controle.

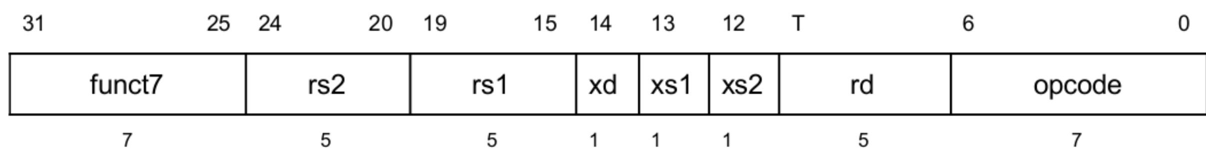

Figura 28: RoCC instruction format.

Direção	Sinal	Valor Default	Descrição
input	core_resp_ready_i	0	Sinais de controle.
output	core_resp_valid_o	0	
output	[4:0]core_resp_rd_o	rd	ID de registrador de destino na resposta.
output	[63:0]core_resp_data_o	rd_data	Dado do registrador de destino na resposta.

Tabela 8: Register Mode - Sinais de resposta.

A interface de memória é composta por subgrupos de requisição e resposta da memória. Todos os nomes dos sinais são prefixados com "mem_" para indicar o grupo ao qual pertencem. As requisições a memória é feita pelo RoCC usando os sinais descritos na Tabela 9.

Direção	Sinal	Valor Default	Descrição
input	mem_req_ready_i	0	Sinais de controle.
output	mem_req_valid_o	0	
output	[39:0]mem_req_addr_o	addr	Endereço de memória p/ leitura/escrita.
output	[9:0]mem_req_tag_o	tag	Identidade exclusiva atribuída a cada solicitação de memória, especialmente para o mesmo local de memória, para oferecer suporte a respostas fora de ordem.
output	[4:0]mem_req_cmd_o	cmd	Opcode requisão de memória [0x0000=load,0x0001=store].
output	[2:0]mem_req_typ_o	typ	Tamanho de resposta; [0x000=8bits, 0x001=16bits, 0x010=32 bits, 0x011=64 bits].
output	mem_req_phys_o	1	*Assert* se os endereços são virtuais e precisam de tradução.
output	[63:0]mem_req_data_o	w_data	Grava dado.

Tabela 9: RoCC sinais de requisição de memória.

As respostas da memória para o RoCC são passadas pela interface descrita na Tabela 10. Há uma resposta para solicitações de leitura/escrita. Normalmente, o RoCC pode usar apenas os campos de tag e dados da resposta. Observe que não há sinal de "pronto" do acelerador para reconhecer a aceitação de respostas de memória, o que significa que é esperado que o acelerador esteja pronto para aceitar dados em cada ciclo.

Direção	Sinal	Valor Default	Descrição
input	mem_resp_valid_i	0	Controle.
input	[39:0]mem_resp_addr_i	addr	Exibe o pedido de leitura/escrita.
input	[9:0]mem_resp_tag_i	tag	Para diferenciar entre respostas a várias solicitações em andamento.
input	[4:0]mem_resp_cmd_i	cmd	Retorna o Código de comando da solicitação.
input	[2:0]mem_resp_typ_i	typ	Indica tamanho de dado na resposta.
input	[63:0]mem_resp_data_i	data	Contém resposta de dado apra requisão de leitura.
input	mem_resp_nack_i	0	(desconhecido).
input	mem_resp_replay_i	0	(desconhecido).
input	mem_resp_has_data_i	0	Defina se o campo de dados for válido em resposta.
input	[63:0]mem_resp_data_word_bypass_i	bypass_data	Ignora escrita para uma resposta de leitura no mesmo ciclo.
input	[63:0]mem_resp_store_data_i	w_data	Retorna dado escrito na solicitação de escrita durante a resposta correspondente.

Tabela 10: RoCC sinais de resposta da memória.

6.2.3 Interface Estendida

Estes sinais são incluídos somente se o RoCC estiver configurado para incluí-los.

O grupo UTL (Uncached TileLink) é composto de 2 subgrupos, a saber: aUTL e sinais UTL (ver a arquitetura na Figura 27).

Os sinais aUTL está ligado na cache de instruções L2. Os sinais aUTL são compostos pelos subgrupos conforme listados na Tabela 11 e 12 respectivamente. Eles são prefixados com “autl_” para denotar o grupo ao qual eles pertencem.

Os sinais UTL são vetores de sinais do mesmo tipo que aUTL. O tamanho do vetor UTL é igual ao número de canais de memória independentes, configuráveis no Rocket Core. No entanto, esses sinais ainda precisam ser adicionados no documento.

Como foi citado na seção L1 Cache e MMU a interface para cache L2 foi excluída do projeto temporariamente devido a falhas encontradas nesse sistema, podendo ser readicionada em novas versões.

A interface FPU pode ser usada pelo acelerador se tiver uma unidade de ponto flutuante conectada a ele. Todos os nomes dos sinais são prefixados com "fpu_" para indicar o grupo ao qual pertencem. A interface é composta de subgrupos de solicitação FP e de resposta FP, conforme listado na Tabela 13 e na Tabela 14 respectivamente.

Os Registradores de Status e Controle (CSR) podem ser usados opcionalmente dentro do acelerador. Se existirem, os sinais de interface na Tabela 15 podem ser usados pelo sistema operacional no núcleo para mapear a memória do acelerador. Seu mecanismo deve ser adicionado. Todos os nomes dos sinais são prefixados com "csr_' para indicar o grupo ao qual pertencem.

Os sinais do Page Table Walker (PTW) são usados para conversão de endereço do acelerador (se o núcleo já não estiver manipulando-o). Esses sinais serão adicionados como parte de um trabalho futuro.

Direção	**Nome do Sinal**
input	autl_acquire_ready_i
output	autl_acquire_valid_o
output	[25:0] autl_acquire_bits_addr_block_o
output	[2:0] autl_acquire_bits_client_xact_id_o
output	[1:0] autl_acquire_bits_addr_beat_o
output	autl_acquire_bits_is_builtin_type_o
output	[2:0] autl_acquire_bits_a_type_o
output	[16:0] autl_acquire_bits_union_o
output	[127:0] autl_acquire_bits_data_o

Tabela 11: AUTL acquire signals.

Direção	**Nome do Sinal**
output	autl_grant_ready_o
input	autl_grant_valid_i
input	[1:0] autl_grant_bits_addr_beat_i
input	[2:0] autl_grant_bits_client_xact_id_i
input	[3:0] autl_grant_bits_manager_xact_i
input	autl_grant_bits_is_builtin_type_i
input	[3:0] autl_grant_bits_g_type_i
input	[127:0] autl_grant_bits_data_i

Tabela 12: AUTL grant signals.

Direção	Nome do Sinal
input	fpu_req_ready_i
output	fpu_req_valid_o
output	[4:0]fpu_req_bits_cmd_o
output	fpu_req_bits_ldst_o
output	fpu_req_bits_wen_o
output	fpu_req_bits_ren1_o
output	fpu_req_bits_ren2_o
output	fpu_req_bits_ren3_o
output	fpu_req_bits_swap12_o
output	fpu_req_bits_swap23_o
output	fpu_req_bits_single_o
output	fpu_req_bits_fromint_o
output	fpu_req_bits_toint_o
output	fpu_req_bits_fastpipe_o
output	fpu_req_bits_fma_o
output	fpu_req_bits_div_o
output	fpu_req_bits_sqrt_o
output	fpu_req_bits_round_o
output	fpu_req_bits_wflags_o
output	[2:0]fpu_req_bits_rm_o
output	[1:0]fpu_req_bits_typ_o
output	[64:0]fpu_req_bits_in1_o
output	[64:0]fpu_req_bits_in2_o
output	[64:0]fpu_req_bits_in3_o

Tabela 13: RoCC FPU Request signals.

Direção	Nome do Sinal
output	fpu_resp_ready_o
input	fpu_resp_valid_i
input	[64:0]fpu_resp_bits_data_i
input	[4:0]fpu_resp_bits_exc_i

Tabela 14: RoCC FPU Response signals.

Direção	Nome do Sinal
input	[11:0]csr_waddr_i
input	[63:0]csr_wdata_i
input	csr_wen_i
output	[63:0]csr_rdata_o

Tabela 15: RoCC CSR signals.

6.2.4 Implementação do RoCC no Rocket Chip

O RoCC é um componente do Rocket Tile, dessa forma ele está implementado no diretório tile/ no Código LazyRoCC.scala. Nesse Código, escreve-se o projeto de acelarador com linguagem Chisel ou Verilog. Existe ainda o Código em subsystem/Configs.scala que serve para instanciar as soluções presentes no Código LazyRoCC.scala. Na seção do FIR será demonstrado esse fluxo para melhor entendimento.

O Rocket Chip suporta até quatro aceleradores por Tile, cada um com os 7 bits de opcodes pré-definidos com os valores mostrados na Tabela 16 e Figura 28. Eles são conFigurados no Código subsystem/Configs.scala, como mostrado no Código 5.

RoCC ID	Código binário
RoCC 0	b0001011
RoCC 1	b0101011
RoCC 2	b1011011
RoCC 3	b1111011

Tabela 16: RoCC Opcodes.

```
class WithRoccExample extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(rocc = Seq(
        RoCCParams(
          opcodes = OpcodeSet.custom3, // define rocc ID
          generator = (p: Parameters) => {
            val accumulator = LazyModule(new // instancia RoCC
            AccumulatorExample()(p))
            accumulator}),
        RoCCParams(
          opcodes = OpcodeSet.custom2, // define rocc ID
          generator = (p: Parameters) => {
            val translator = LazyModule(new // instancia RoCC
            TranslatorExample()(p))
            translator},
            nPTWPorts = 1),
        RoCCParams(
          opcodes = OpcodeSet.custom1, // define rocc ID
          generator = (p: Parameters) => {
            val counter = LazyModule(new // instancia RoCC
            CharacterCountExample()(p))
            counter}),
        ...
        ...
```

Código 5: Parte do Código subsystem/Configs.scala p/ instanciar e config. RoCCs.

O Código de implemetanção do RoCC fornece quatro exemplos de aceleradores e classes auxiliares que o encapsulam para ser integrado oo sistema. Seguindo a filosofia de uso do Rocket Chip, deve-se reaproveitar essa infraestrutura de Códigos, seguindo o padrão exemplificado para construção de novas implementações de coprocessadores, isso é uma recomendação da University of Berkeley para o início do desenvolvimento de SoCs através do uso da plataforma de uso da plataforma.

O Código subsystem/Configs.scala ainda não é o módulo topo do sistema, porém nele pode-se conFigurar e personalizar vários componentes do sistema, como: core, cache (como já foi visto nas subseções precedentes), RoCC, entre outros. Veja no Código 5 que é o exemplo fornecido pelo Rocket Chip, instanciando alguns projetos de aceleradores, a saber: AccumulatorExample, translator, CharacterCountExample. Quando essa classe é adicionada ao módulo topo do sistema (system/Configs.scala) então é adicionado os aceleradores ao sistema.

Na seção 10 é implementado um RoCC, o que facilita a compreensão da teoria explanada nessa seção.

6.3 Barramentos

O Rocket Chip implementa vários protocolos para interfacear os seus componentes, sendo eles: TileLink, AXI4, AHB-Lite e APB. O TileLink Bus é o principal deles, por estabelecer a hierarquia de memória, coerência entre caches e interface para os demais protocolos listados. Esses outros protocolos são usados para conexão da memória principal e dispositivos de E/S.

A Figura 19 ilustra como esses barramentos estão presentes na plataforma. A implementação desses protocolos estão nos dirétorios tilelink/ e amba/. A Tabela 3 cita e descreve brevemente eles.

Enfim, nessa Seção iremos abordar de maneira resumida o que é o protocolo TileLink e as possíveis personalizações que a plataforma fornece para ele. Também será destacado como os outros protocolos estão presentes na plataforma.

6.3.1 TileLink Bus [25]

O TileLink é um barramento de interconexão para chips que fornece vários mestres com coerência em acesso de memória e outros dispositivos escravos. O TileLink é projetado para uso em SoC para conectar multiprocessadores de propósito geral, coprocessadores, aceleradores, mecanismos DMA e dispositivos simples ou complexos, usando uma interconexão escalonável rápida, fornecendo transferências de baixa e alta taxa de transferência [25]. O TileLink:

- é um padrão aberto para barramentos SoC fortemente acoplados e de baixa latência;
- foi projetado para RISC-V, mas suporta outras ISAs;
- fornece um sistema de memória compartilhada endereçado fisicamente;
- pode ser implementado em redes ponto a ponto escaláveis e hierarquicamente compostas;
- fornece acesso coerente para uma mistura arbitrária de mestres de caching/non-caching;
- pode ser reduzido a simples dispositivos escravos ou escalar até escravos de alto rendimento;

Algumas das características importantes do TileLink incluem:

- memória compartilhada com suporte a mecanismo de cache-coerente;
- conclusão fora de ordem para melhorar o rendimento;
- interfaces desacopladas, facilitando a inserção no estágio de registro;
- adaptação de largura de barra sem estado e fragmentação de rajada;
- codificação de sinal com reconhecimento de energia;

Uma rede TileLink pode suportar uma mistura de agentes de comunicação, cada um suportando diferentes subconjuntos do protocolo. A especificação TileLink inclui três níveis de conformidade para agentes conectados [25].

O mais simples é o TileLink Uncached Lightweight (TL-UL), que suporta apenas operações de leitura e gravação de memória simples (Get/Put) de palavras isoladas. O intermediário é o TileLink Uncached Heavyweight (TL-UH), que adiciona várias dicas, operações atômicas e acessos em rajada, mas sem suporte para caches coerentes. Finalmente, o TileLink Cached (TL-C) é o protocolo completo, que suporta o uso de caches coerentes [25]..

Quando um agente do processador TL-C se comunica com um agente de dispositivo TL-UL, o agente do processador deve se abster de usar os recursos mais avançados ou deve haver um adaptador TL-C para TL-UL na rede entre os dois. Os agentes podem suportar outras combinações de recursos, mas apenas os três níveis de conformidade listados são cobertos por esta especificação [25].

A implementação do TileLink na plataforma Rocket Chip está localizada em tilelink/. Esse conjunto de Códigos implementam todos os modos do protocolo TileLink, e pontes para outros protocolos, como AMBA AXI4, AHB-Lite e APB.

A plataforma usa como padrão a implementação do TL-C como o barramento de comunicação entre os Rocket Tiles. Para dispositivos de E/S, por exemplo: barramento de periféricos (veja a Figura 26) o padrão de implementação é TL-Uncached e a memória principal usa a ponte TL-C para AXI4 que se aproveita do mecanismo de coerência de memória.

Como opção de personalização, a plataforma permite criar sistemas onde o barramento principal implementa o TL-UH. O que leva a um circuito mais simples e, portanto, menor porque exclui o mecanismo de coerência do Caches. Isso é feito usando a classe WithIncoherentTiles no módulo topo. Essa classe é implementada no Código Configs.scala localizado em subsystem/. Informações descritivas e de manipulação do módulo topo serão apresentadas na Seção 8.

```
class WithIncoherentTiles extends Config((site, here, up) => {
        case RocketCrossingKey => up(RocketCrossingKey, site)
          map { r => r.copy(master = r.master.copy(cork = Some(true)))}
        case BankedL2Key => up(BankedL2Key, site).copy(coherenceManager
           = { subsystem =>
        val ww = LazyModule(new TLWidthWidget(subsystem.sbus.beatBytes)
                 (subsystem.p))(ww.node, ww.node, () => None)})
```

Código 6: Parte do Código subsystem/Configs.scala para remover mecanismo de coerência de cache .

6.3.2 AMBA AXI4, AHB-Lite e APB Protocols

O Rocket Chip possui conversores do TileLink para os protocolos AMBA AXI4, AHB-Lite e APB, a fim de permitir a interface com periféricos externos de terceiros e controladores de memória, mas também fornece a implementação separada desses protocolos, localizados no diretório amba/.

Esses Códigos podem ser reutilizados para adicionar novos dispositivos AMBA ao sistema, porque o TileLink (o barramento principal) fornece os conversores TileLink para estes outros protocolos. A especificação AMBA [26] pode ser consultada para maiores informações.

É importante salientar que não são fornecidos exemplos pela plataforma para criar e integrar dispositivos com esse barramentos, e no presente trabalho não foi desenvolvido nada com esse recurso. Porém veremos na Seção 6.5 como a plataforma usa esses Códigos e portanto, como ele pode ser explorado caso o usuário deseje usar esse barramento em algum projeto.

6.4 Controladores de Interrupção

O Rocket Core suporta as seguintes interrupções: locais de software e temporização e globais. Essas interrupções são recebidas pelo processador seguindo a especificação do manual do RISC-V [18].

As **Interrupções locais** são sinalizadas diretamente para um hart (hardware thread ou processador, Figura 29) com um valor de interrupção dedicado. Isso permite reduzir a latência de interrupção, já que não é necessária a arbitragem para determinar qual hart atende a uma determinada solicitação, nem acessos de memória adicionais necessários para determinar a causa da interrupção. Interrupções de software e timer são interrupções locais geradas pelo Core Local Interruptor (CLINT).

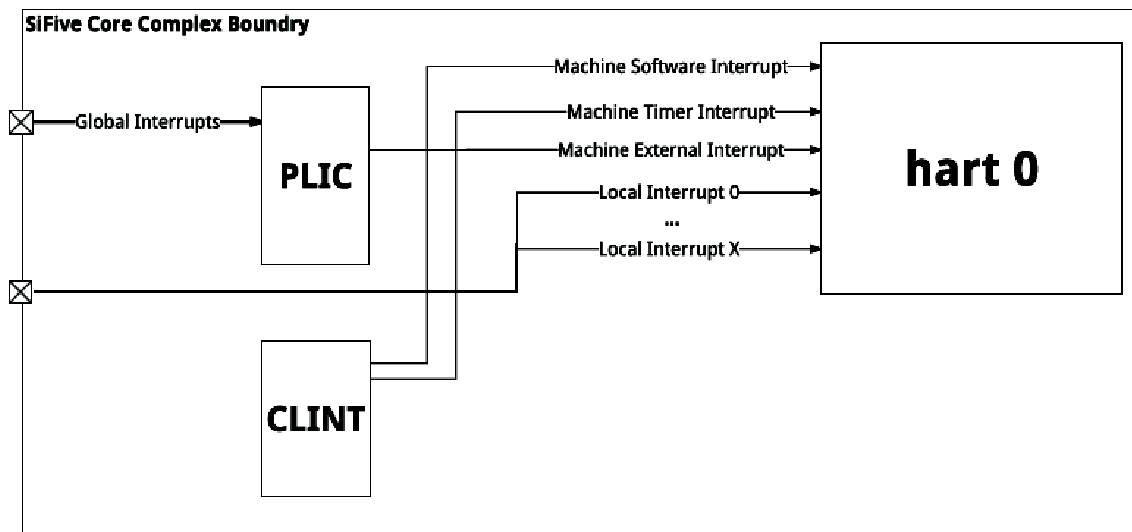

Figura 29: Módulos CLINT e PLIC conectados ao Rocket Core [23].

Interrupções Globais, em contrapartida, são roteadas através do PLIC (Platform-Level Interrupt Controller), que pode direcionar interrupções para qualquer hart no sistema através de interrupção externa. Consequentemente, ele possui uma latência maior no tratamento de interrupções comparado com CLINT. Porém, o desacoplamento das interrupções globais do(s) hart(s) permite uma ampla gama de número de interrupções e esquemas de prioridades e roteamento.

Os projetos U5 Coreplex [22] e E31 Coreplex [23] da empresa SiFive, são projetos derivados do Rocket Chip e que usam esses mecanismos. Toda as informações nos documentos da SiFive como registradores de conFigurações de interrupção e mapa de memória são os mesmo para o Rocket Chip, por isso iremos referencia-los nessa explicação.

Estes módulos são mapeados em memória e conectados ao sistema pelo barramento TileLink, o que permitindo sua configuração via software.

O Rocket Chip, fornece em qualquer implementação, o sistema de interrupções locais conectado aos Rocket Cores, i.e., toda implementação do SoC tem um módulo CLINT mapeado em memória com um temporizador integrado que pode ser acessado via software para gatilhar interrupções locais de software e Timer.

O PLIC também está presente nas implementações do Rocket Chip, porém não possui nenhuma fonte de interrupção conectada a ele. O que o Rocket Chip fornece com a implementação do PLIC são portas para conexão de fontes de interrupções externas. Por exemplo, o usuário pode criar um periférico mapeado em memória contendo sinais projetados para serem fontes de interrupção e esses sinais são conectados ao PLIC utilizando-se as classes do PLIC que fornecem essa conectividade. Ou seja, é necessário programação adicional por parte do usuário para se usar esse mecanismo.

A implemetação destes módulos está localizada no repositório devices/tilelink/ nos Códigos CLINT.scala e Plic.scala. Como dito, o Plic.scala contém classes que podem ser usadas dentro de outros componentes para conectar e criar novas fontes de interrupção ao PLIC. Existe um exemplo do uso dessas classes no Código Example.scala presente no diretório tilelink/, esse exemplo mostra como criar um periférico mapeado em memória e como conectar sinais ao módulo PLIC.

6.4.1 CLINT - Core Local Interrupt

O bloco CLINT contém registradores de status e controle mapeados na memória associados a interrupção de software e Timer. Nessa Seção serão descritas as informações básicas do uso desse módulo que deve ser feita via software. Todas as informações do CLINT está em conformidade com o manual RISC-V [18] e as informações detalhadas sobre o uso desse módulo para desenvolvimento de software está em [22] e [23].

Ele possui três registradores que podem ser acessados via software para conFigurar as interrupções de software e Timer, sendo eles: 1) Machine-mode Software Interrupts (MSIP), 2) MTIME (Machine-mode Interrupt Pending) e 3) MTIMECMP (Machine-mode Timer Compare).

MSIP ou Machine-mode Software Interrupts é o registrador usado para gatilhar interrupções de software, mapeado em memória, conforme ilustrado na Figura 30. As interrupções de software são mais úteis para a comunicação entre processadores em sistemas multiprocessadores (ou multi-harts), já que os harts podem escrever no msip um dos outros permitindo a comunicação entre processadores. Na reinicialização, os registros msip são zerados.

CLINT Register Map				
Address	**Width**	**Attr.**	**Description**	**Notes**
0x0200_0000	4B	RW	msip for hart 0	MSIP Registers
0x0200_0004	4B	RW	msip for hart 1	
0x0200_0008	4B	RW	msip for hart 2	
0x0200_000C	4B	RW	msip for hart 3	
0x0200_0010	4B	RW	msip for hart 4	
0x0200_0014 ... 0x0200_3FFF			*Reserved*	
0x0200_4000	8B	RW	mtimecmp for hart 0	Timer compare register
0x0200_4008	8B	RW	mtimecmp for hart 1	
0x0200_4010	8B	RW	mtimecmp for hart 2	
0x0200_4018	8B	RW	mtimecmp for hart 3	
0x0200_4020	8B	RW	mtimecmp for hart 4	
0x0200_4028 ... 0x0200_BFF7			*Reserved*	
0x0200_BFF8	8B	RO	mtime	Timer register
0x0200_C000 ... 0x0200_FFFF			*Reserved*	

Figura 30: Mapa de Memória do Módulo CLINT, fonte [23].

MTIME ou Machine-mode Timer é um registrador R/W (Read/Write ou Leitura/Escrita) de 64 bits que contém o número de ciclos contados a partir do sinal de alternância do RTC (Real Time Clock). Uma interrupção de timer torna-se pendente sempre que mtime for maior ou igual ao valor no registro mtimecmp (Machine-mode Timer Compare).

Por padrão, todas as interrupções interrompem a máquina em modo máquina. Para as interrupções de software e de temporizador serem delegadas no modo de supervisor deve-se primeiro conFigurar o processador ao modo de supervisor e conFigurar os respecitivos resgistros, que muda somente o prefixo do nome, e.g: ao invés de MSIP é SSIP (Supervisor-mode Software Interrupts). Mais informações sobre os Modos de Máquina, consulte [18] e [20].

É importante destacar que apenas a conFiguração do CLINT não causa interrupções no sistema. Para sensibilizar o processador no atendimento de interrupções faz-se necessário ainda conFigurar os Registros de Controle e Status (CSR, Control Status Registers) da máquina, para habilitar ou mascarar interrupções (mstatus, Machine Status Register e mie, Machine Interrupt Enable Register) e para gerir as interrupções, descobrindo quais foram as fontes da causa de interrupção (mip, Machine Interrupt Pending, mcause, Machine Cause Register), etc. Todas essas informações estão detalhadas na documentação SiFive [23].

6.4.2 PLIC - Platform-Level Interruptor Controller

O bloco PLIC contém registradores de status e controle mapeados na memória associados a interrupções globais. Nessa Seção serão descritas as informações básicas do uso desse módulo que deve ser feita via software. Todas as informações do PLIC está em conformidade com o manual RISC-V [18] e as informações detalhadas sobre o uso desse módulo para desenvolvimento de software está em [22] e [23].

O PLIC suporta até 255 fontes de interrupções externas caso o sistema seja 32 bits ou 511 caso 64 bits. Como dito anteriormente, não existe nenhum dispositivo conectado a ele como fonte de interrupção, sendo papel do usuário criar e conectar essas fontes.

PLIC Register Map			
Address	**Width**	**Attr.**	**Description**
0x0C00_0000			*Reserved*
0x0C00_0004	4B	RW	source 1 priority
0x0C00_0008	4B	RW	source 2 priority
...			
0x0C00_0800	4B	RW	source 511 priority
0x0C00_0804 ... 0x0C00_0FFF			*Reserved*
0x0C00_1000	4B	RO	Start of pending array
...			
0x0C00_103C	4B	RO	Last word of pending array
0x0C00_1040 ... 0x0C00_1FFF			*Reserved*
0x0C00_2000	4B	RW	Start Hart 0 M-Mode interrupt enables
0x0C00_203C	4B	RW	End Hart 0 M-Mode interrupt enables
0x0C00_2040 ... 0x0C1F_FFFF			*Reserved*
0x0C20_0000	4B	RW	Hart 0 M-Mode priority threshold
0x0C20_0004	4B	RW	Hart 0 M-Mode claim/complete
0x0C20_0008 ... 0x0FFF_FFFF			*Reserved*

Figura 31: Mapa de Memória do Módulo PLIC em sistema 64 bits, fonte [23].

Quando fontes de interrupções são conectadas ao PLIC, a lógica para se gerir essas interrupções é similar ao CLINT, ou seja, deve-se conFigurar os registradores de status e controle da máquina (CSR) e os registradores desse bloco. O detalhamento desses registradores encontra-se nas documentações citadas anterirormente. Na Seção das ferramentas, será apresentado um exemplo demonstrando como connectar um dispositivo ao PLIC e as necessidades de software para se usar esse recurso computacional.

6.5 Memórias

Nessa Seção iremos falar sobre as memórias presentes no Rocket Chip, onde estão implementados no repositório oficial e quais as opções de personalização que a plataforma oferece para o usuário.

6.5.1 ROM

A memória ROM do Rocket Chip, também chamada de BootROM possui dois Códigos importantes para a inicialização do sistema: o Código de bootloader e o DTB (Device Tree Blob). Ela está conectada ao sistema atráves do barramento TileLink e está mapeada entre os endereços 0x10000 e 0x20000.

O software de bootloader do sistema está localizado na primeira parte da mémoria, entre 0x10000 ao 0x10040, e é o primeiro Código a ser executado na máquina, i.e., o contador de programa PC aponta para o endereço 0x10000 quando a máquina é inicializada. Esse software basicamente faz com que a máquina entre em um *loop* de espera enquanto o processador não é interrompido.

O que causa a interrupção do processador é o dispositivo E/S DTM - Device Transport Module que falaremos adiante. Ele é comandado por agentes externos e é capaz de gerar interrupções na máquina. Na Seção sobre o Emulador do Rocket Chip será detalhado como funciona a estrutura de bootloader fornecida pelos projetista do sistema.

O Código de bootloader default é compilado com a *toolchain* RISC-V e é fornecido no diretório rocket-chip/bootrom/ (ver Figura 32). A saída de compilação gera um arquivo .img que é adicionado aos Códigos Chisel do sistema:

```
class BaseSubsystemConfig extends Config ((site, here, up) => {
...
case BootROMParams =>
BootROMParams(contentFileName = "./bootrom/bootrom.img")
...
```

Código 7: Parte do Código subsystem/Configs.scala para adic. binário do bootloader.

```
rocket-chip/bootrom/
├── bootrom.img
├── bootrom.S
├── linker.ld
└── Makefile

0 directories, 4 files
```

Figura 32: Códigos do bootloader do Rocket Chip.

Portanto, o usuário tem como opção de personalização a criação de outras rotinas de bootloader. Porém, ao modificar o bootloader a infraestrutura de emulação deve também ser modificada, pois o projeto do emulador foi desenvolvido tomando como base esse Código de bootloader.

O segundo Código presente na memória ROM é o Código do DTB. O DTB é um banco de dados que representa os componentes de hardware no sistema. Ele é derivado das especificações do IBM OpenFirmware e foi escolhido como o mecanismo padrão para passar informações de hardware de baixo nível para o bootloader de inicialização de kernel Linux [28]. O DTB da ROM é escrito em Chisel no Código diplomacy/DeviceTree.scala e é adicionado diretamente sem opções de personalização. Não é fornecido nenhum exemplo na plataforma para portar Sistemas Operacionais (SO) no Rocket Chip, portanto, esse sistema não foi usado nesse trabalho.

6.5.2 SRAM - Memória Principal

A mémoria principal do Rocket Chip simula o comportamento de uma SRAM (Static Random Access Memory) com interface AMBA AXI4. Ela é conectada ao sistema atráves de um conversor AMBA AXI4 para TileLink e ocupa o espaço de memória entre 0x8000_0000 e 0x9000_0000 (256MB).

Essa memória recebe o Código da aplicação (baremetal ou SO). O transporte do binário para essa memória é feito usando a interface DTM ou JTAG. Na conFiguração default da plataforma, veremos posteriormente que o projeto do Emulador contém os drivers para fazer a operação de transporte e ao fim do carregamento modifica o PC para o endereço 0x8000_0000 fazendo com que a aplicação seja executada.

A plataforma aceita a mudança do endereço e tamanho dessa memória, conforme ilustrado no Código system/Configs.scala (parte dele pode ser visto no Código 8). Porém, não aconselha-se a mudança para manter o padrão no qual a plataforma foi desenvolvido.

É interessante destacar também que quando o projetista de SoC inicializar o processo de síntese a interface AMBA AXI4 deve estar disponível para a instalação de uma verdadeira mémoria no sistema. É importante ressaltar que o processo de síntese da plataforma está fora do escopo do presente trabalho.

```
class BaseConfig extends Config(new BaseSubsystemConfig().alter((site,here,
up) => {
...
  case ExtMem => MasterPortParams(
                      base = x"8000_0000",
                      size = x"1000_0000",
...
```

Código 8: Parte do Código system/Configs.scala para modificar endereço da SRAM.

6.5.3 MMIO - Memory Mapped I/O

De forma análoga a SRAM, a MMIO é um componente de memória presente no Rocket Chip que simula o comportamento de uma DRAM com interface AMBA AXI4.

Ela consome o espaço de endereçamento que vai do 0x6000_0000 ao 0x8000_0000 (512MB), que pode ser facilmente modificado no Código system/Configs.scala no parâmetro **size**. Veja abaixo:

```
class BaseConfig extends Config(new BaseSubsystemConfig().alter((site,here,
up) => {
...
  case ExtBus => MasterPortParams(
                      base = x"6000_0000",
                      size = x"1000_0000",
                      //size = x"2000_0000",
...
```

Código 9: Parte do Código system/Configs.scala para modificar endereço da DRAM.

Essa região de memória também pode ser particionada para adicionar periféricos com interface AMBA AXI4 no sistema. Ressalta-se, porém, que a infraestrutura de Códigos necessária não está finalizada. Portanto, necessita-se de programação adicional por parte do usuário.

6.6 Dispositivos de E/S

As implementações default do Rocket Chip não possuem dispositivos de E/S (periféricos) comumente encontrados em SoCs comerciais, como: GPIO, Timer, PWM, UART, etc. Porém, a plataforma fornece um Código exemplo sem implementação de hardware que mostra o passo-a-passo para integrar circuitos mapeados em memória e com interface TileLink-Uncached. Esse Código encontra-se em tilelink/ com o nome Example.scala.

Além desse Código, existe um repositório no GitHub, desenvolvido também pela Universidade de Berkeley, que contém outros exemplos para criar periféricos. Esse repositório demonstra também o fluxo de desenvolvimento de softwares em C para acessar e testar o dispositivo no Emulador. Pode ser acessado a partir desse link `https://github.com/ucb-bar/project-template`.

A plataforma também fornece uma maneira de integrar periféricos desenvolvidos diretamente em Verilog, ao invés de Chisel. Na Seção sobre as ferramentas do Rocket Chip, será apresentado um exemplo demonstrando como integrar um periférico em Verilog no sistema. Veremos que essa capacidade de adicionar Códigos pode ser usada em outras partes do projeto Rocket Chip, pois na verdade é o Chisel que consegue importar e instanciar Códigos Verilog.

6.7 Debug - DTM (Device Transport Module) e JTAG

O sistema de debug (depuração) presente no Rocket Chip é apresentado na Figura 33. Esse sistema está em conformidade com a especificação de debug para sistemas RISC-V.

Como pode ser visto na Figura, o usuário interage com o Debug Host (por exemplo, notebook), que está executando um depurador (por exemplo, gdb). O depurador se comunica com um tradutor de depuração (por exemplo, OpenOCD, que pode incluir um hardware driver) para se comunicar com o Hardware de Transporte de Depuração (por exemplo, o adaptador Olimex USB-JTAG). O hardware de transporte de depuração conecta o host de depuração ao módulo de transporte de depuração (DTM) da plataforma. O DTM fornece acesso ao Módulo de Depuração (DM) usando a Interface do Módulo de Depuração (DMI)

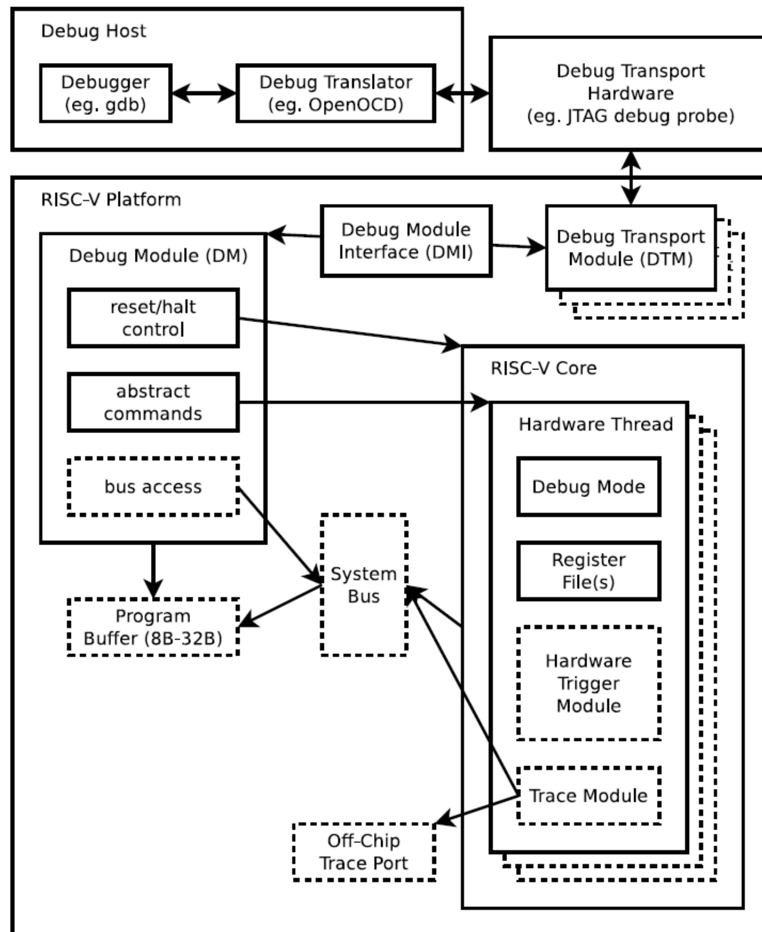

Figura 33: Arquitetura do sistema de depuração do RISC-V.

O DM permite que o depurador interrompa qualquer hart na plataforma. Comandos abstratos fornecem acesso a GPRs (General Purpose Registers). Registradores adicionais são acessíveis através de comandos abstratos ou por programas escritos para o Buffer de Programa opcional.

O programa buffer permite que o depurador execute instruções arbitrárias em um hart. Este mecanismo pode ser usado para acessar a memória. Um bloco de acesso de barramento de sistema opcional permite acessos de memória sem usar uma harware RISC-V para executar o acesso.

Cada hart RISC-V pode implementar um Módulo Trigger. Quando as condições de acionamento são atendidas, os harts param e informam ao módulo de depuração que eles têm parado.

A documentação completa sobre esse mecanismo de debug pode ser obtida na documentação [29].

O Rocket Chip implementa todo esse mecanismo. Os Códigos para o DTM estão localizados em devices/debug e o JTAG no diretório jtag/. O DTM sempre está presente no sistema, porém, o JTAG pode ser adicionado ou removido do SoC. Isso é feito no módulo topo do sistema (system/Configs.scala). Lá a plataforma disponibiliza uma classe chamada WithJtagDTMSystem que ao ser adicionada na criação de um SoC adiciona automaticamente o dispositivo JTAG. Nos exemplos das próximas seções ficará mais claro como isso é feito.

```
rocket-chip/src/main/scala/devices/debug/
├── abstract_commands.scala
├── DebugRomContents.scala
├── Debug.scala
├── DebugTransport.scala
├── dm_registers.scala
└── Periphery.scala
```

Figura 34: Arquivos de implementação do DTM.

```
rocket-chip/src/main/scala/jtag/
├── JtagShifter.scala
├── JtagStateMachine.scala
├── JtagTap.scala
├── JtagUtils.scala
├── package.scala
└── Utils.scala
```

Figura 35: Arquivos de implementação do JTAG.

Além disso, o sistema de emulação fornecido pela plataforma emula o SoC construído em que o acesso a ele é dado somente pela interface do DTM. Dessa forma, o Rocket Chip já fornece os drivers para DTM e JTAG e os usa no processo de bootloader do sistema. Todo esse processo de funcionamento do Emulador e como é feito esse estímulo será discutido em detalhes na Seção do Emulador.

6.8 Mapa de Memória

Start Address	End Address	Component
0x0000_0000	0x0000_1000	Debug - DTM e JTAG
0x0000_2000	0x0000_3000	Dispositivos de E/S
0x0000_3000	0x0000_4000	Error Device
0x0001_0000	0x0002_0000	BootROM
0x0200_0000	0x0201_0000	CLINT - Core Local Interrupt Controller
0x0C00_0000	0x1000_0000	PLIC - Platform-Level Interrupt Controller
0x6000_0000	0x8000_0000	MMIO
0x8000_0000	0x9000_0000	Memória Principal (256MB)

Tabela 17: Mapa de Memória Default do Rocket Chip.

Nessa Seção vamos sintetizar os endereços de todos os componentes abordados até aqui e falar sobre outros espaços de endereços ainda não abordados. Veja na Tabela 17 como os endereços são organizados na conFiguração default (sem modificações) da plataforma Rocket Chip.

Vale destacar que esse mapa de memória é bastante similar às implementações comerciais do Rocket Chip, o SiFive E31 [22] e SiFive E51 [23]. Então, de certa forma as documentações dessa plataforma também são úteis para entender os dispositivos do Rocket Chip já que para ele mesmo não existe uma documentação formal de componentes.

Como já vimos, esse mapa de memória é volátil, i.e., como o usuário pode fazer modificações no sistema, como: adicionar ou remover Dispositivos E/S, adicionar, particionar ou remover a região de MMIO, modificar o tamanho da memória principal, adicionar ou remover dispositivo de JTAG, etc. Portanto, deve-se ter em mente que esse mapeamento muda de acordo com o projeto de SoC e isso gera consequências na construção de softwares para o sistema.

O único dispositivo não abordado nessa região de memória foi o Error Device. Esse dispositivo está implementado no Código Error.scala localizado em device/tilelink/ e é conectado ao sistema atráves do barramento TileLink. Não existe nenhum documentação sobre esse componente, porém analisando o Código ele é componente do barramento TileLink.

7 Chisel - Uma nova Linguagem de Descrição de Hardware

Figura 36: Logo Chisel

7.1 Introdução (Overview)

Para entendermos Chisel, primeiro devemos entender o que motivou a criação de uma nova HDL - *Hardware Description Language*, já que existem linguagens como VHDL e Verilog que são amplamente usadas e portanto demonstram serem suficientes para projetos de sistemas digitais.

Na verdade, as HDLs tradicionais (VHDL e Verilog) foram originalmente desenvolvidas como linguagens de simulação de hardware e só mais tarde foram adotadas como base para a síntese de hardware. Como a semântica dessas linguagens é baseada na simulação, os projetos sintetizáveis devem ser inferidos de um subconjunto da linguagem, o que complica o desenvolvimento de ferramentas e a formação de projetistas. Essas linguagens também não possuem as poderosas facilidades de abstração que são comuns nas modernas linguagens de software, o que leva a uma baixa produtividade do projetista ao dificultar a reutilização de componentes. Construir projetos de hardware eficientes requer extensa exploração do espaço de desenvolvimento de microarquiteturas, mas essas HDLs tradicionais são limitadas na geração de módulos, de tal forma que são inadequadas para produzir e compor módulos de hardware altamente parametrizados necessários para suportar a exploração completa do espaço de desenvolvimento. Extensões recentes, como SystemVerilog, melhoram o sistema de tipos e parametrização, mas ainda não possuem muitos recursos poderosos de linguagem de programação de alto nível [30].

Uma forma de resolver esses problemas é construindo uma nova HDL, mas essa não é a melhor solução. Tendo em vista que os principais softwares para sintetizar hardware em FPGA ou ASIC (*Application Specific Integrated Circuit*) interpretam VHDL e/ou Verilog, ao criar uma nova HDL toda essa infraestrutura teria que ser atualizada para interpretar essa nova linguagem.

Uma abordagem mais aceita é o uso de uma linguagem de programação de alto nível para interpretar macros capazes de descrever hardware com alto nível de abstração, parametrização e modularidade, e ainda capaz de gerar o Código equivalente em alguma HDL tradicional, como Verilog. Para exemplificar, suponha que é preciso criar um Mux 4:1, em Verilog uma possível implementação é mostrado no codigo 10.

```
module Mux4to1(
        input  in0, in1, in2, in3,
        output out,
        input  [1:0] sel);

always@(*)
begin
        case(sel)
                2'b00: out <= in0;
                2'b01: out <= in1;
                2'b10: out <= in2;
                2'b11: out <= in3;
        endcase
end
endmodule
```

Código 10: Mux 4:1 em Verilog.

Imagine agora uma linguagem que interpreta macros e que possuisse uma macro chamada Mux que recebe como parâmetros: entradas e seletor, e retorna o resultado para ser lido em uma variável, como mostrado no Código abaixo.

```
out = Mux(input0,input1,input2,...,...,seletor)
```

Código 11: Uma possível abstração para Mux.

Dessa forma, um Mux 4:1 poderia ser descrito como apresentado no Código 12 e no processo de compilação a linguagem de alto nível interpretaria esse Código (ou macro) e construiria o Código Verilog equivalente, como o apresentado no Código 10 ou em um nível mais baixo, como uma *netlist*. Perceba que com essa metodologia, ao precisar modificar esse Mux com mais ou menos entradas ao invés de modificarmos o Código Verilog (que não é produtivo) basta excluir ou adicionar as entradas nos parâmetros e o interpretador fará o trabalho de construir/modificar o Verilog equivalente. Além disso, a reutilização do Código se torna muito mais produtiva já que basta chamar a macro Mux, em qualquer parte da descrição do hardware.

```
out = Mux(in0,in1,in2,in3,sel)
```

Código 12: Uma possível abstração para Mux 4:1.

O que essa abordagem faz, de forma análoga a macro Mux, é definir e criar um conjunto de outras macros capazes de descrever hardware em alto nível, de forma superior as linguagens VHDL e Verilog. Porém, como o produto da compilação continua sendo alguma dessas linguagens, toda a infraestrutura para síntese de hardware continua sendo aproveitada. Isso torna-se possível graças ao uso de uma linguagem de programção de alto nível (ou *High-Level* DSL - *Domain-Specific Language*), que possui poderosos recursos de programação não presentes nas HDLs tradicionais.

Um exemplo de linguagem de descrição de hardware construída sobre uma DSL de alto nível e ainda capaz de gerar Verilog equivalente é o Chisel, abreviação para *Constructing Hardware in a Scala Embedded Language*, que como podemos deduzir foi desenvolvida sobre a linguagem de programação Scala. Essa linguagem suporta paradigmas de programação: funcional, orientado à objetos e imperativa, além de permitir tipos parametrizados e inferência de tipos. Embutindo Chisel no Scala, ela herda todos esses recursos e permite um alto nível de abstração ao descrever hardware, como será visto no decorrer dessa seção.

Segundo os desenvolvedores do Chisel, o Scala foi escolhido devido aos seguintes motivos: 1) é uma linguagem muito poderosa com recursos importantes para a construção de geradores de circuitos; 2) é especificamente desenvolvida como base para DSL; 3) compila para a JVM (Java Virtual Machine); 4) tem um grande conjunto de ferramentas de desenvolvimento e IDEs; e 5) tem uma comunidade de usuários bastante grande e crescente [30].

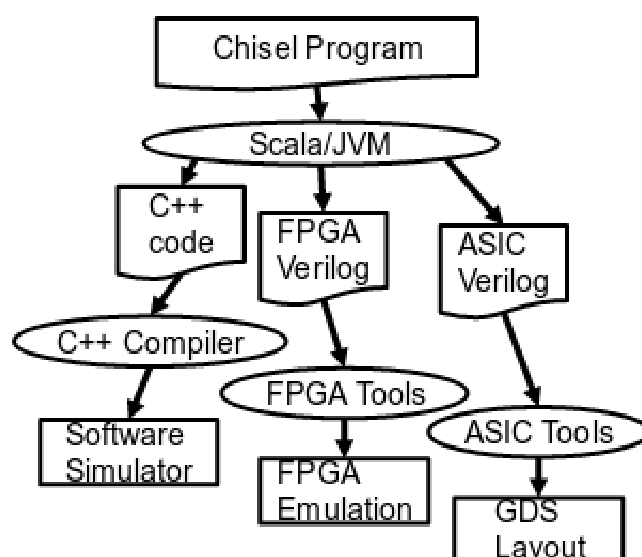

Figura 37: Fluxo de compilação de um programa Chisel.

O Chisel compreende um conjunto de bibliotecas **Scala** que definem novos tipos de dados de hardware e um conjunto de rotinas para converter uma estrutura de dados de hardware em um simulador rápido de C++ ou Verilog de baixo nível para emulação

ou síntese [30], como pode ser visto na Figura 37. Veja que o Chisel é equivalente a um programa Scala e é compilado no Scala JVM. Essa "aplicação" gera como saída um arquivo Verilog, que pode ser usado nas ferramentas de síntese em FPGA e/ou ASIC, e gera também um simulador C++, também escrito em Chisel, e capaz de estimular o circuito desenvolvido, conforme demonstrado na seção Desenvolvimento de Testbenches.

É importante destacar também o pacote Scala FIRRTL (*A Flexible Intermediate Representation for RTL*) embutido no Chisel. O FIRRTL cria a representação do circuito imediatamente após a elaboração Chisel e tem como objetivo reduzir essa representação a um baixo nível de descrição por meio de uma seqüência de transformações. Isto é, durante cada transformação o circuito é reescrito em um circuito equivalente usando construções mais simples e de nível inferior. Eventualmente, o circuito é simplificado para a sua forma mais restrita, assemelhando-se a uma *netlist* estruturada, que permite uma fácil tradução para uma linguagem de saída (por exemplo, Verilog). Para uma descrição detalhada do FIRRTL, consulte [32].

O melhor exemplo da alta capacidade que o Chisel possui em termos de parametrização e modularização no desenvolvimento de circuitos digitais é o projeto Rocket Chip. Esse projeto usa o gerador de Verilog e simulador C++, demonstrando toda a capacidade e a qualidade na geração de Código que essa linguagem oferece.

Contudo, o Chisel ainda está em processo de evolução e desenvolvimento. Por conta de suas constantes atualizações, existem várias documentações e artigos que divergem no que diz respeito à sintaxe e tipos de dados, tornando o aprendizado um tanto improdutivo. Existem duas versões da linguagem: Chisel2 e Chisel3. A versão do Rocket Chip utilizada nesse trabalho (ver Código 2) usa ambas, predominando o Chisel2, que iremos abordar aqui. Vale ressaltar que o Chisel3 tenta manter o máx. de compatibilidade entre elas. Outra observação importante é que o FIRRTL só é usado na última versão Chisel3. Além disso o sistema de teste dele usa o Verilator em conjunto com o gerador de simulação C++, melhorando então o mecanismo de teste, conforme observado em [34].

Nessa seção, será visto de forma básica os principais recursos do Chisel, como: principais Tipos de Dados (Datatypes), construção e hierarquia de módulos, atribuições para desenvolvimento de lógica combinacional e sequencial. Além disso, por meio dos exemplos práticos, poderemos verificar e avaliar o Verilog gerado no processo de compilação e a metodologia de teste oferecida usando o simulador C++.

O objetivo é entender de forma básica como programar com Chisel, que consequentemente, ajudará a ter uma melhor compreensão do projeto Rocket Chip e auxiliará no desenvolvimento dos trabalhos propostos no estágio. Além disso, o material pode ser usado como documentação inicial para desenvolvimento de circuitos digitais com essa nova HDL, na pesquisa e no ensino.

7.2 Tipos de Dados (Datatypes) e Operadores

Tipos de Dados são usados para especificar o tipo de valores mantidos em elementos de estado (registradores) ou fluindo em fios. Em Chisel, os tipos são: Bool, Bits, UInt, SInt e Enum. Além disso, esses tipos podem ser compostos como estruturas de dados (Bundle) e como conjunto de tipos usando Vetores (Vec), que serão apresentados posteriormente.

Uma observação interessante é que os tipos de Chisel são diferentes dos tipos incorporados do Scala. Isso permite ao Chisel checar e responder erros sobre o uso desses elementos em um Código. Os tipos também possuem parâmetros que definem tamanho da palavra binária e sua direção de entrada ou saída (caso o sinal componha a interface do componente a ser projetado), igualmente como as HDLs tradicionais.

Nesta seção falaremos dos Tipos de Dados em Chisel e seus Operadores, explorando por meio de pequenos trechos de Códigos suas construções e uso.

7.2.1 Bool

O Bool é o tipo usado para representar valor booleano (verdeiro ou falso). Além disso, ele pode receber como parâmetro a direção do sinal (entrada ou saída), que se faz necessário na criação de interface de componentes.

A Tabela 18 mostra a sintaxe de construção do tipo Bool e no Código 13 são apresentados exemplos de uso.

sintaxe (Construtor)	Descrição	Retorno
Bool(dir: PortDir)	Cria Bool com direção especificada por dir, sendo INPUT para entrada e OUTPUT para saída. Essa sintaxe é compátivel com o Chisel3, veja [33] e [35].	Bool
Bool(x: Boolean)	Cria Bool literal, true p/ verdadeiro (1) e false p/ falso (0).	Bool
Bool()	Cria Bool vazio, tipo inferido.	Bool

Tabela 18: Construtores do tipo Bool.

```
val a = Bool(INPUT) // cria entrada booleana "a"
val b = Bool()      // cria variavel tipo Bool
  b := a            // b recebe a
val c = Bool(true)  // atribui valor verdade p/ variavel b
```

Código 13: Exemplo de uso para tipo Bool.

Por fim, na Tabela 19 tem-se os principais operadores que retornam o tipo Bool.

Operador	Descrição	Retorno
!x	NOT no bit x.	Bool
x && y	AND dos bits x e y.	Bool
x || y	OR dos bits x e y.	Bool
x === y	Compara se x é igual a y.	Bool
x =/= y	Compara se x é diferente de y.	Bool
x > y	Compara se x é maior que y.	Bool
x >= y	Compara se x é maior ou igual a y.	Bool
x < y	Compara se x é menor que y.	Bool
x <= y	Compara se x é menor ou igual a y	Bool

Tabela 19: Operadores que retornam tipo Bool.

```
val a, b = Bool()       // cria variaveis tipo Bool
val c = (a === b)       // c herda tipo Bool e vale
                        // true sempre que a for igual a b
val d = a && b          // d recebe valor de AND entre a e b e herda tipo Bool
```

Código 14: Exemplo de operadores retornando Bool.

7.2.2 Bits

O tipo Bits é o tipo que corresponde a um **vetor de bits** ou um vetor de tipos Bool que não trasmite nenhum significado aritmético, diferentemente dos tipos UInt e SInt, que veremos na próxima subseção.

A sintaxe para esse tipo é mostrado na Tabela 21, em seguida o Código 15 exemplifica formas de uso.

sintaxe (Construtor)	Descrição	Retorno
Bits(dir: PortDir, width: Int)	Cria Bits com direção especificada por dir, sendo INPUT para entrada e OUTPUT para saída, o parâmetro width é um número inteiro que define o comprimento do vetor.	Bits
Bits(width: Int)	Cria Bits com comprimento do vetor defindo por width.	Bits
Bits()	Cria Bits vazio, tipo inferido.	Bits

Tabela 20: Construtores do tipo Bits.

```
val myArray = Bits(OUTPUT, width = 32) // cria uma saida de 32 bits
val halfWord= Bits(width = 16) // cria um variavel com 16 bits
val inferred= Bits()
    inferred := UInt(123) // e' atribuido constante 123 e tamanho 7 bits
                          // a variavel inferred
```

Código 15: Exemplo de uso para tipo Bits.

A Tabela 21 complementa as operações listadas na Tabela 19. Como pode ser observado, agora essa operações podem fazer cálculos em cadeia de bits, ou seja, em variáveis do tipo Bits. Todavia, esses cálculos são permitidos entre variáveis do tipo Bool, já que uma variável tipo Bits(width=1) é equivalente a uma variável do tipo Bool.

Operador	Descrição	Retorno
˜ x	Bitwise NOT	Bits(w(x) bits)
x & y	Bitwise AND	Bits(w(xy) bits)
x | y	Bitwise OR	Bits(w(xy) bits)
x ˆ y	Bitwise XOR	Bits(w(xy) bits)
x.xorR	XOR todos os bits de x	Bool
x.orR	OR todos os bits de x	Bool
x.andR	AND todos os bits de x	Bool
x » y	Deslocamento lógico p/ direita, y: Int	Bits(w(x)-y bits)
x » y	Deslocamento lógico p/ direita, y: UInt	Bits(w(x) bits)
x « y	Deslocamento lógico p/ esquerda, y: Int	Bits(w(x)+y bits)
x « y	Deslocamento lógico p/ esquerda, y: UInt	Bits(w(x)+max(y) bits)
x(y)	Lê bit da posição y da palavra Bits, y=0 (LSB)	Bool
x(high,low)	Lê campo da posição high até low	Bits(high-low+1 bits)

Tabela 21: Extensão de operadores lógico da Tabela 19.

```
val inp = Bits(INPUT, width = 10) // cria entrada tipo Bits tam. 10
val out0, out1, out2 = Bits(OUTPUT, width = 10) // sinais de saida tipo
val outBit = Bool()                              // Bits tam. 10

out0 := inp.xorR // out0 recebe XOR em todos os bits de inp
out1 := inp >> UInt(2) // out1 recebe deslocam. de inp 2 bits p/ direita
out2 := inp ^ UInt(0xCAFE) // out2 recebe XOR entre inp e 0xCAFE

outBit := inp(9) // outBit recebe MSB da palavra inp
```

Código 16: Exemplo de operadores com tipo Bool.

7.2.3 UInt e SInt

Os tipos UInt e SInt correspondem a **vetores de bits** que podem ser usados para aritméticas de inteiros sem e com sinal, respectivamente. Essa é a única diferença entre esses tipos e o tipo Bits. Portanto, uma variável do tipo UInt(width=1) ou SInt(width=1) é equivalente a uma variável do tipo Bool. Todas as operações listadas anteriormente nas Tabelas 19 e 21 são válidas para esses elementos.

A implementação desse tipo é mostrado na Tabela 22

Sintaxe (Construtor)	Descrição	Retorno
UInt(dir: PortDir, width: Int)	Cria UInt com direção especificada por dir, sendo INPUT para entrada e OUTPUT para saída, o parâmetro width é um número inteiro que define o comprimento do vetor.	T(max(w(x), w(y) bits)
SInt(dir: PortDir, width: Int)	Idem a descrição anterior, porém cria SInt.	T(max(w(x), w(y) bits)
UInt(width: Int)	Cria UInt com comprimento do vetor definido por width.	T(w(x) + w(y) bits)
SInt(width: Int)	Idem a descrição anterior, porém cria SInt.	T(w(x) bits)
UInt()	Cria UInt vazio, tipo inferido.	T(w(x) bits)
SInt()	Idem a descrição anterior, porém cria SInt.	T(w(x) bits)

Tabela 22: Construtores dos tipos UInt e SInt.

```
val a = UInt(width = 32) // cria fios a UInt 32 bits
val b = UInt(width = 16) // cria fios b UInt 16 bits

val s = SInt(width = 8) // cria variavel s SInt 8 bits

val c = UInt()
    c := a+b // c recebe soma de a e b, herdando o tipo UInt
             // e o tamanho do valor max. de a+b
```

Código 17: Exemplo de uso do tipo UInt e SInt.

Como incremento das Tabelas de operações apresentadas anteriormente, apresentamos aqui as operações aritméticas disponíveis em Chisel. Elas também podem ser usadas com o tipo Bits. Porém em operações aritméticas os tipos UInt e SInt são mais apropriados, ficando a critério do programador.

Operador	Descrição	Retorno
x + y	Adição	T(max(w(x), w(y) bits)
x - y	Subtracão	T(max(w(x), w(y) bits)
x * y	Multiplicação	T(w(x) + w(y) bits)
x / y	Divisão	T(w(x) bits)
x % y	Módulo	T(w(x) bits)

Tabela 23: Operadores aritméticos.

```
val inp1, inp2 = UInt(w)   // cria fios  UInt de w bits

val out1 = SInt(width = w) // cria fios out1 SInt de w bits
val out2 = Bits(width = w*2) // cria fios out2 SInt de w bits

out1 := inp1 + inp2 // soma inp1 e inp2 sem sinal e guarda
                    // em variavel com sinal
out2 := inp1 * inp2 // variavel tipo Bits out2 recebe resultado
                    // de multiplicacao, perceba que o tamanho de
                    // out2 e' duas vezes tamanho da entrada
```

Código 18: Exemplo de operadores aritméticos.

7.2.4 Constantes

Como foi visto anteriormente, a construção dos tipos em Chisel permite criar portas de entrada/saída (atribuindo valor a variável dir) e fios, ambos com tamanho definidos pela variável width. É permitido também criações vazias que ao serem atribuidas à alguma outra variável herda seu tamanho. Por exemplo:

```
val in_port = Bits(dir = OUTPUT, width = w) // saida tipo Bits de tam. w
val wire    = UInt(width = 2+w) // fio de tamanho 2+w
val inferred= SInt() // tipo SInt sem tamanho
```

Código 19: Exemplo de criação de porta, fio e tipo inferido.

Em contrapartida, quando uma construção dos tipos UInt e SInt é feita sem definir direção e com algum valor no parâmetro sem o uso da variável width, o Chisel interpreta essa construção como sendo uma constante, positiva ou negativa, respectivamente.

Em resumo, o tipo UInt pode ser usado para representar constantes positivas de no máximo 32 bits, ou seja, números entre 0 e 4294967295. Da mesma forma, constantes podem ser criadas com SInt, que por conta do sinal, é capaz de representar números entre -2147483648 e 2147483647. Constantes de 64 bits também podem ser criadas usando o tipo BigInt, nativo do Scala. Quando uma constante é criada o Chisel automaticamente atualiza a variável width com a quantidade de bits mínima que possa representar o número.

Atribuições de valores com MSB recebendo bit 1 no tipo UInt não são aceitas pelo compilador Chisel, pois isso significa que queremos definir um número negativo em um tipo positivo. Para contornar isso e fazer com que o compilador interprete a constante como um número positivo, deve-se fazer uma extensão do número com MSB 1 o conventendo para o tipo Long usando a letra L no final do número.

Para exemplificar, algumas construções são apresentadas no Código 20.

```
val out0 = Bits(width = 32)
val out1 = Bits(width = 32)
val out2 = UInt(width = 64)
val out3 = SInt(width = 64)
val out4 = UInt(width = 128)
val out5 = Bits(width = 32)

out0 := UInt(4294967295L)
// saida 0xFFFFFFFF, precisa ser Long (L) quando MSB igual a 1

out1 := SInt(-1) // 0xFFFFFFFF
out2 := BigInt("FFFFFFFFFFFFFFFF", 16).U // max valor 64 bits
out3 := BigInt("FFFFFFFFFFFFFFFF", 16).S // -1 em 64 bits

val constant1 = UInt(0xDEADBEFFL) // constant1 tera width igual a 32
out4 := constant1 // recebe valor 0xDEADBEEF concatenado

val constant2 = UInt(width = 20)
out5 := constant2
// out5 recebe valor de constant2 aleatorio concatenado
// com 12 bits em zero nos bits mais significativos
```

Código 20: Exemplo de uso de constantes.

7.2.5 Enum

O tipo Enum (Enumerado) atribui a cada elemento valores na ordem crescente, começando do 0 até o N-ésimo termo. O Chisel usa a macro Enum(type, width) para construí-los. O primeiro parâmetro recebe o tipo, podendo ser Bits ou UInt e o segundo parâmetro width define o tamanho dos elementos. Sua sintaxe é apresentada no Código 21.

```
// Enum de N elementos tipo Bits
val element0 :: element1 :: ... :: elementN :: Nil = Enum(Bits(), N)
// Enum de N elementos tipo UInt
val element0 :: element1 :: ... :: elementN :: Nil = Enum(UInt(), N)
```

Código 21: sintaxe construção de Enum.

Cada elemento recebe seu respectivo valor, que poderá ser usado para compor lógica, atribuições, cálculos, etc, não podendo ser alterados, como constantes. Caso altere o compilador Chisel acusará erro.

7.2.6 Conversão de Tipos

O Chisel permite conversão entre tipos, veja a Tabela 24. As conversões podem ser aplicadas a todos os elementos que possuem um tipo específico.

Operador	Descrição	Retorno
x.asBits	Constrói x como tipo Bits	Bits(w(x) bits)
x.asUInt	Constrói x como tipo UInt	UInt(w(x) bits)
x.asSInt	Constrói x como tipo SInt	SInt(w(x) bits)
x.asBools	Constrói como vetor de Bool	Vec(Bool, w(x))
x.U	Constrói x como UInt, só funciona em BigInt	UInt(w(x) bits)
x.S	Constrói x como SInt, só funciona em BigInt	SInt(w(x) bits)

Tabela 24: Conversão de tipos.

```
val out = SInt(width = 32)

val unsignedValue = UInt(0xDEADBEFFL)
  out := unsignedValue.asSInt // constroi unsignedValue como SInt
```

Código 22: Exemplo de conversão de tipo.

7.2.7 Tipos compostos: Vec e *Bundle*

O Vec é um tipo composto que define um grupo de sinais indexados (de qualquer tipo básico do Chisel) sob um único nome.

sintaxe	Descrição
Vec(T: Data, n: Int)	Onde T recebe o tipo e n a quantidade de elementos.

Tabela 25: sintaxe tipo Vec.

```
val outputWord = UInt(width = 32)
val outputBit  = Bool()

// cria Vec com 16 elementos tipo UInt de 32 bits
val myArrayUInt = Vec(UInt(width = 32), 16)

// atribui valor 0xCAFE ao elemento 7 do vetor
myArrayUInt(7) := UInt(0xCAFE)

outputWord := myArrayUInt(7) // outputWord recebe 0xCAFE
outputBit  := myArrayUInt(7)(0) // Recebe o LSB do elemento 7
```

Código 23: Exemplo uso de Vec.

O Bundle é um tipo composto que define um grupo de sinais nomeados (de qualquer tipo básico Chisel) sob um único nome. Esse tipo pode ser comparado de forma análoga aos Structs da linguagem C, podendo ser usado para modelar estruturas de dados, barramentos e interfaces.

A sintaxe para declarar e instanciar um Bundle é apresentada no Código 24.

```
// declara Bundles
class myFirstBundle(param1: Int, param2: Int, ...) extends Bundle {
  val structSignals = new Bundle {
    val signal0 = Bool()
    val signal1 = Bits(width = param1)
    val signal2 = UInt(width = param2)
    ...
  }
}
// fim declaracao

// instancia
val myStruct = new myFirstBundle(16,32)
   // acesso aos membros
   myStruct.structSignals.signal0 := ...
   myStruct.structSignals.signal1 := ...
   myStruct.structSignals.signal2 := ...
   ...
```

Código 24: Exemplo uso de Bundles.

7.2.8 Conclusão: Lógica Combinacional e Fios (Wires)

Para encerrar essa seção sobre tipos de dados, enfatizamos que todas essas estrututuras vistas dão base para o desenvolvimento elementar de circuitos combinacionais, ou seja, conseguimos atribuir a sinais ("fios") operações com portas lógicas. Na próxima seção veremos os elementos de estado (registradores), para posteriormente aprender as estruturas de mais alto nível When e Switch que permitem descrever lógicas mais complexas.

```
val wireAnd = a & b # operacao AND entre a e b
val wire = wireAnd | (~c & d) # wireAnd OR ((not c) AND d)
```

Código 25: Exemplos de implementação de circuitos elementares.

7.3 Lógica Sequencial: Registradores, Clock e Reset

A criação e o uso de registradores em Chisel é bastante diferente das linguagens VHDL e Verilog. Em Chisel, registradores são instaciados, ou seja, cria-se o objeto registrador com **Clock e Reset implícito**, diferentemente das linguagens tradicionais, que criam blocos, *process* no VHDL ou *always* no Verilog, e devem explicitar Clock, Reset e suas lógicas.

Essa diferença em relação às HDLs tradicionais orientado a eventos tem um grande impacto: 1) você pode atribuir registradores e fios no mesmo escopo; 2) você não precisa dividir seu Código entre blocos de *process* ou *always*, o que torna a implementação bastante flexível.

A instanciação padrão de um registrador em Chisel irá atualizá-lo em borda de subida com o valor que ele recebe (atualização incondicional). Em contrapartida o reset não é definido, fazendo com que o registrador receba valores aletórios em sua inicialização. Para defini-lo, deve-se especificar a variável init no construtor do registrador. Consequentemente, todo sinal ativo do reset, que é síncrono, altera o valor do registrador para o valor de init.

Os registradores também recebem tipos de dados, podendo ser qualquer um daqueles listados na seção precedente, e essa definição é obrigatória. Por fim, existe ainda a variável next no construtor que é uma forma de atribuir um sinal ao registrador na mesma linha de instanciação dele. Porém, atualizações condicionais também são possíveis com as estruturas When e Switch, que veremos nas próximas sessões.

O Clock em Chisel é implicito nos registradores, sendo o mesmo para todo o circuito, mas pode-se criar também múltiplos domínios de Clock. Para mais informações sobre implementações desse tipo consulte [33].

Enfim, a Tabela 26 exibe as formas de instanciar um registrador e em seguida são apresentados formas de uso.

sintaxe (Construtor)	**Descrição**
Reg(data: T)	Cria um registrador com tipo T e sem inicialização.
Reg(data: T, init: UInt)	Cria um registrador com tipo T e com inicialização (reset) definido por init.
Reg(data: T, init: UInt, next: T)	Cria um registrador com tipo T , com inicialização (reset) definido por init e será atualizado a cada ciclo com a variável recebida por next.

Tabela 26: Construtores de Registradores.

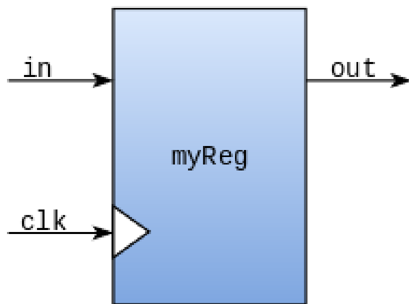

Figura 38: Registrador p/ Exemplo 1.

```
module myReg(input clk,
    input  io_in,
    output io_out
);
  reg  myReg;

  assign io_out = myReg;

  always @(posedge clk) begin
    myReg <= io_in;
  end
endmodule
```

Código 26: Verilog gerado por Chisel ex. 1.

```
val io = IO(new Bundle{
        val in  = Bool(INPUT)
        val out = Bool(OUTPUT)
})
val myReg = Reg(Bool())
myReg  := io.in
io.out := myReg
```

Código 27: Exemplo 1 de Reg. em Chisel.

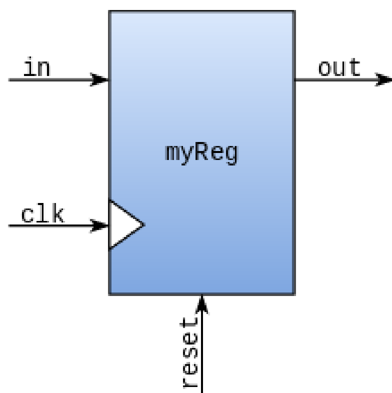

Figura 39: Registrador p/ Exemplo 2.

```
val io = IO(new Bundle{
        val in  = Bool(INPUT)
        val out = Bool(OUTPUT)})
val myReg = Reg(Bool(),
            init=Bool(false))
myReg  := io.in
io.out := myReg
```

Código 28: Exemplo 2 de Reg. em Chisel.

```
module Foo(input clk, input reset,
    input  io_in,
    output io_out
);

  reg  myReg;
  wire T0;

  assign io_out = myReg;
  assign T0 = reset ? 1'h0 : io_in;

  always @(posedge clk) begin
    if(reset) begin
      myReg <= 1'h0;

    end else begin
      myReg <= io_in;
    end
  end

endmodule
```

Código 29: Verilog gerado por Chisel ex. 2.

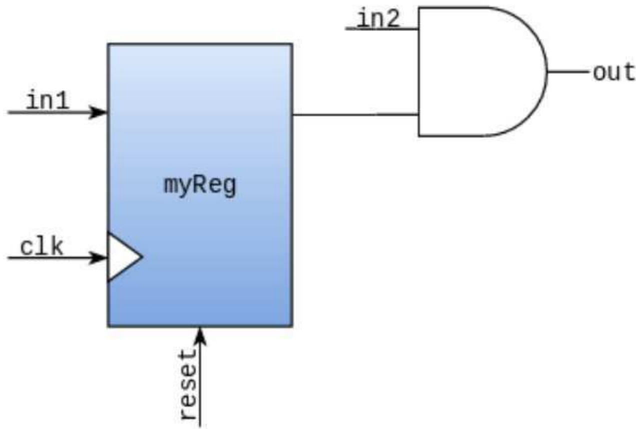

Figura 40: Registrador p/ Exemplo 3.

```
module Foo(input clk,input reset,
    input  io_in1,
    input  io_in2,
    output io_out
);
  wire T0;
  reg  myReg;
  wire T1;

  assign io_out = T0;
  assign T0 = myReg & io_in1;
  assign T1 = reset?1'h0:io_in1;

  always @(posedge clk) begin
    if(reset) begin
      myReg <= 1'h0;
    end else begin
      myReg <= io_in1;
    end
  end
endmodule
```

Código 30: Verilog gerado por Chisel ex. 3.

```
val io = IO(new Bundle{
        val in  = Bool(INPUT)
        val out = Bool(OUTPUT)})
val myReg = Reg(Bool(),init=Bool(false))
myReg  := io.in
io.out  := myReg
```

Código 31: Exemplo 3 de Reg. em Chisel.

Perceba no Exemplo 1 que o reset não aparece, aparecendo nos outros exemplos devido a construção de init. Note também que esse último exemplo, poderia ser re-escrito usando next, de forma equivalente:

```
  val io = IO(new Bundle{
    val in1 = Bool(INPUT)
    val in2 = Bool(INPUT)
    val out = Bool(OUTPUT)
  })
  val myReg = Reg(Bool(), init = Bool(false), next = io.in1)
  io.out := myReg & io.in1
```

Código 32: Outra forma de de escrever reg. do exemplo 3.

7.4 Semântica

Iremos falar nessa seção sobre os tipos de atribuições do Chisel e as estruturas para programação condicionais When e Switch.

7.4.1 Atribuições

Em Chisel o operador de atribuição entre fios é o ":=", como pode ser verificado nos exemplos precedentes, ele é equivalente a atribuição **não-bloqueante** do Verilog, não existindo um operador para atribuições bloqueantes.

Sinais de entrada/saída e fios são criados com sinal "=", como fora visto nos exemplos anteriores. Existe ainda um operador de atribuição "<>", ele é usado para conexão de mesmas interfaces, é bastante usado no projeto Rocket Chip, por facilitar a conexão entre componentes e barramento.

```
val x = UInt() // aloca x como um fio do tipo UInt()
x := y // atribui (conecta) fio y em fio x
x <> y // conecta x e y, podem ser fios ou uma interface
```

Código 33: Tipos de atribuições.

7.4.2 When e Switch

Existem duas estruturas para descrição de lógica condição, são elas: When e Switch. O When executa blocos condicionais com tipo Bool, ele é equivalente ao **if** do Verilog e VHDL. Já o Switch executa blocos condicionais com valores/dados.

```
when(condition1) {
// run if condition1 true
// and skip rest
}.elsewhen(condition2) {
// run if condition2 true
// and skip rest
}.elsewhen(condition3) {
// run if condition3 false
// and skip rest
}.otherwise{
// run if none of the above ran
}
```

Código 34: Estrutura When [36]

```
switch(x) {
  is(value1) {
     // run if x === value1
  } is(value2) {
     // run if x === value2
  }
}
```

Código 35: Estruturas Swithc [36]

Para exemplificar o uso do When, vamos detalhar a implementação de um Timer que conta de forma decrescente até chegar em zero. Analisando o Código 36 podemos ver que quando o sinal enable está em nível baixo (0 ou false) a variável auxiliar auxTimerLoad recebe o valor de entrada timerLoad e coloca saída timeout em nível baixo. Mas, quando o enable é ativo e o auxTimerLoad é diferente de zero o sinal auxTimerLoad é decrementado de 1 em todo ciclo de clock (sim, pois auxTimerLoad é um registrador), se o circuito continuar nessas condições quando o registrador auxTimerLoad zerar o bit timeout é ativo.

```
class Timer(w: Int) extends Module {
  val io = IO(new Bundle {
    val enable = Bool(INPUT)
    val timerLoad = UInt(INPUT, w)
    val timeout = Bool(OUTPUT)
    val timerValue = UInt(OUTPUT, width = w)
  })
  val auxTimerLoad = Reg(UInt(width = w), init = UInt(0))

  when (io.enable && (auxTimerLoad =/= UInt (0))) {
    auxTimerLoad := auxTimerLoad - UInt(1) // decrementa
  } .elsewhen (io.enable && (auxTimerLoad === UInt (0))) {
    io.timeout := Bool(true) // timeout
  } .otherwise {
    io.timeout := Bool(false)
    auxTimerLoad := io.timerLoad
  }
  io.timerValue := auxTimerLoad
}
```

Código 36: Exemplo de circuito usando condicional When.

Para exemplificar o condicional Switch vamos implementar uma ULA com capacidade de fazer 4 operações: somar, subtrair, multiplicar e dividir número inteiros.

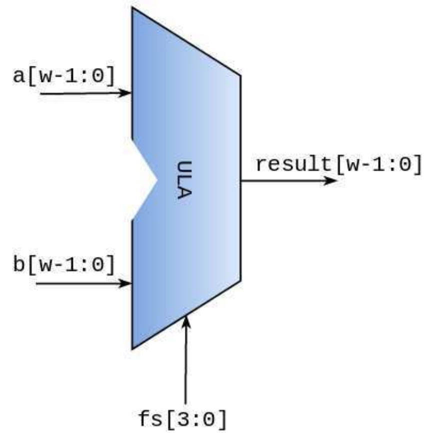

Figura 41: ULA para exemplo com condicional.

```
class ULA(w: Int) extends Module {
  val io = IO(new Bundle{
    val result = UInt(OUTPUT, w)
    val a  = UInt(INPUT, w)
    val b  = UInt(INPUT, w)
    val fn = UInt(INPUT, 2)
  })
  // wires
  val result = UInt(width = 4)
  val a  = io.a
  val b  = io.b
  val fn = io.fn

  // default    obrigat rio
  result := UInt(0)
  switch(fn) {
    is(0.U) { result := a + b }
    is(1.U) { result := a - b }
    is(2.U) { result := a * b }
    is(3.U) { result := a / b }
  }

  // output
  io.result := result
}
```

Código 37: Exemplo de circuito usando condicional Switch.

7.5 Construindo Módulos

No Chisel, os módulos são muito semelhantes aos módulos em Verilog, definindo uma estrutura hierárquica no circuito gerado. Um módulo definido pelo usuário é definido como uma classe que [33]:

- herda a classe Module;
- contém uma interface Bundle armazenada em um campo chamado io;
- conecta os subcircuitos em seu construtor.

Os usuários escrevem seus próprios módulos como uma subclasse da classe Module, que é definido da forma mostrada abaixo. Para esclarecer ainda mais, veja os exemplos anteriores 36 e 37. Na primeira linha é criado uma classe que herda Module, então uma interface Bundle é feita definindo a interface e por fim, os sinais da interface são usados no campo de construtor.

```
abstract class Module {
        val io: Bundle
        var name: String = ""
        def compileV: Unit
        def compileC:
}
```

Código 38: Classe Module.

7.5.1 Hierarquia de Módulos e Parametrização

Como outras linguagens de descrição de hardware, o Chisel permite a instanciação de módulos de forma direta, permitindo modularidade e hierarquia.

Módulos são criados dentro de outros módulos, se estes estiverem disponíveis, também usando a classe abstrata Module, mostrada no Código 39.

```
..
val nomeInstancia0 = Module(new nomeImplementacao0)
val nomeInstancia1 = Module(new nomeImplementacao1(param0))
val nomeInstancia2 = Module(new nomeImplementacao2(param0,param2,...))
..
```

Código 39: Instanciação de módulos e parametrização.

Instanciando um módulo e o armazenando em uma variável podemos acessar os membros da interface por meio delas. Todo circuito que extende a Classe Module, permite parametrização, veja os Exemplos 36 e 37, em que o parâmetro w é usado para configurar o tamanho das palavras binárias de cada circuito.

Como exemplo de hierarquia, abaixo temos a implementação de um circuito somador completo usando dois meio somadores.

```
// cria pacote hafadd que sera
// importado no modulo topo
package halfadd
// importa o conjuto de classes
// do Chisel
import Chisel._

// implementa meio somador
class halfadd extends Module {
val io = IO(new Bundle {
  val a = Input(Bool())
  val b = Input(Bool())
  val s = Output(Bool())
  val c = Output(Bool())
})
  io.s := io.a ^ io.b
  io.c := io.a & io.b
}
```

Código 40: Implem. meio somador.

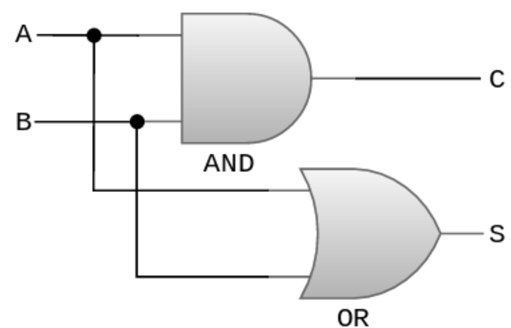

Figura 42: Circuito meio somador.

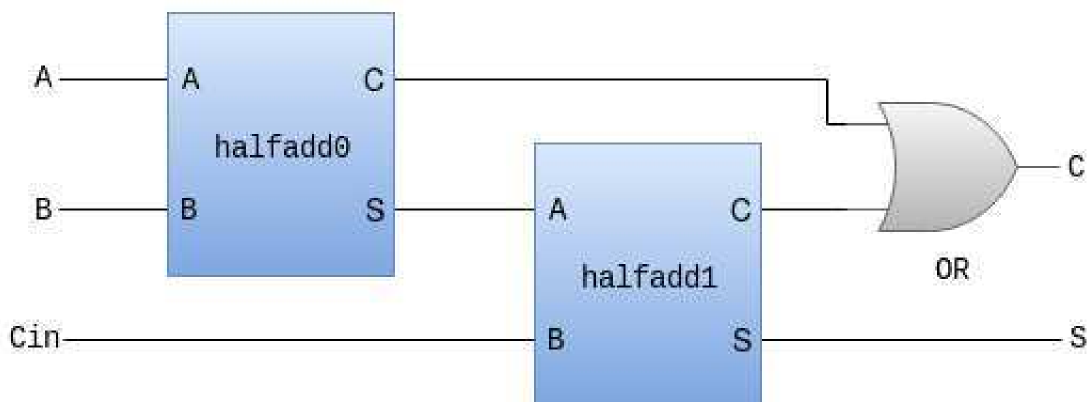

Figura 43: Circuito somador completo usando dois meio somadores.

```scala
package fulladd

import halfadd
import Chisel._

// implementa somador completo
// modulo topo
class fulladd extends Module {
val io = IO(new Bundle {
  val A   = Input(Bool())
  val B   = Input(Bool())
  val Cin = Input(Bool())
  val S   = Output(Bool())
  val C   = Output(Bool())
})
// instancia meio somador halfadd0
val halfadd0 = Module(new halfadd())
  // faz conexoes com entrada IO
  halfadd0.io.a := io.A
  halfadd0.io.b := io.B

// instancia meio somador halfadd1
val halfadd1 = Module(new halfadd())
  // faz conexoes com sinais do halfadd0 e IO
  halfadd1.io.a := halfadd0.io.s
  halfadd1.io.b := io.Cin

// escreve saida do circuito
io.S := halfadd1.io.s
io.C := halfadd0.io.c | halfadd1.io.c
}
```

Código 41: Implementação do somador completo com 2 meio somadores.

7.5.2 Classe *BlackBox*, conectando IPs Verilog

Em Chisel é permitido criar "*wrappers*" para componentes Verilog existentes, usando a classe BlackBox, que são componentes com apenas IO e sem corpo. Em outras palavras, a criação de classes que herdam a classe BlackBox faz o compilador Chisel criar o módulo opaco no Código Verilog gerado pelo compilador.

O Código 42, exemplifica o uso dessa classe, e em seguida é mostrado o Verilog gerado. Perceba que o Chisel adicionou módulos vazios no Código, dessa forma ao sintetizar esse circuito para FPGA ou ferramentas ASIC é necessário adicionar os respectivos Códigos fontes em Verilog.

```
package bb
import Chisel._
// mesmo nome do verilog e mesma interface
class verilogImplementation extends BlackBox {
  val io = IO(new Bundle {
    val in  = UInt(INPUT, 32)
    val out = UInt(OUTPUT,32) })
    debug(Reg(init = Bits(0))) // adiciona o reset e clock implicitos
}
class BlackBoxMod(w: Int) extends Module{
 val io = IO(new Bundle {
      val in0 = new Bundle {
        val in  = UInt(INPUT, w)
        val out = UInt(OUTPUT, w)}
      val in1 = new Bundle {
        val in  = UInt(INPUT, w)
        val out = UInt(OUTPUT, w)}
  })
 val bb0 = Module(new verilogImplementation)
 val bb1 = Module(new verilogImplementation)
 bb0.io <> io.in0
 bb1.io <> io.in1
}
```

Código 42: Exemplo com classe BlackBox.

```
module BlackBoxMod(input clk, input reset,
     input  [31:0] io_in0_in,
     output [31:0] io_in0_out,
     input  [31:0] io_in1_in,
     output [31:0] io_in1_out
);
  wire[31:0] bb0_io_out;
  wire[31:0] bb1_io_out;
  assign io_in1_out = bb1_io_out;
  assign io_in0_out = bb0_io_out;
  verilogImplementation bb0(.clk(clk), .reset(reset),
        .io_in( io_in0_in ),
        .io_out( bb0_io_out )
  );
  verilogImplementation bb1(.clk(clk), .reset(reset),
        .io_in( io_in1_in ),
        .io_out( bb1_io_out )
  );
endmodule
```

Código 43: Verilog gerado do Exemplo com classe BlackBox.

7.5.3 Classe geradora de Verilog

As classes que eram Module ou BlackBox implementam a lógica do circuito, a geração do Código deve ser feita usando a função **main**. Ela deve ser construída no módulo topo e deve ser conFigurada com alguns argumentos especificando o tipo de geração de Código Verilog do circuito.

Sua sintaxe é dada apresentada no Código 44, e a Tabela 27 descreve os argumentos.

```
object moduloTopoMain {
  def main(args: Array[String]): Unit = {
    chiselMain(Array[String]("--backend", "v", "--targetDir", "generated"),
      () => Module(new ModuloTopo(param1, param2, ...)))
  }
}
```

Código 44: Função Main p/ geração de Verilog.

Opções	Descrição
–targetDir	target pathname prefix
–backend	v generate verilog

Tabela 27: Argumentos construção do Verilog.

No módulo topo, se for desejado gerar o Verilog, deve-se definir a função main(). O compilador Chisel interpreta essa função para gerar o Verilog. Veja, por exemplo, a main() implementada no Código 37. Posteriormente, serão exibididos os comandos de compilação.

```
package ula

import Chisel._

class ULA(w: Int) extends Module {
  val io = IO(new Bundle{
    val result = UInt(OUTPUT, w)
    val a  = UInt(INPUT, w)
    val b  = UInt(INPUT, w)
    val fn = UInt(INPUT, 2)
  })
  // wires
  val result = UInt(width = 4)
  val a  = io.a
  val b  = io.b
  val fn = io.fn

  // default   obrigat rio
  result := UInt(0)
  switch(fn) {
    is(0.U) { result := a + b }
    is(1.U) { result := a - b }
    is(2.U) { result := a * b }
    is(3.U) { result := a / b }
  }

  // output
  io.result := result
}
object ULAMain {
  def main(args: Array[String]): Unit = {
    chiselMain(Array[String]("--backend", "v", "--targetDir", "generated"),
      () => Module(new ULA(8)))
  }
}
```

Código 45: Função Main p/ geração de Verilog.

7.6 Chisel Testbenches

O Chisel também pode escrever testbenches para o circuito desenvolvido, porém é bastante limitado podendo fazer somente testes procedurais. Dessa forma, o teste Chisel deve ser usado somente como primeira linha de investigação de bugs, e verificações mais avançadas devem serem feitas usando o Verilog gerado e uma outra linguagem para teste, como SystemVerilog.

Para o desenvolvimento de testes, usa-se a classe Tester. Atenção: nesse relatório a versão do Chisel explanada é a versão Chisel2, a documentação mais nova da linguagem implementa [34] outras classes de testes. Portanto, deve-se ter atenção com as documentações e versão.

Uma classe que herda a classe Tester pode estimular circuitos passados pra ela, usando a seguinte API:

Funções	**Descrição**
poke(Data: T, value: Int)	Atribui ao sinal de entrada T, o valor v.
peek(Data: T, var: T)	Ler o sinal Data e atribui a variável var.
expect(Data: T, var: T)	Compara sinal Data com variável var, retornando mensagem PASS caso valores iguais.
step(cycles: Int)	cycles determina o número de ciclos que devem ser feitos do relógio até seguir para prox. instrução.
printf()	Função printf para imprimir informações extras (uso igual da linguagem C).

Tabela 28: API Chisel Tester.

Vale ressaltar que quando a compilação das classes que herdam Tester é feita ela pode gerar tanto o Verilog equivalente do teste ou como o simulador C++ (conFiguração default). Porém, o simulador C++ é mais aconselhado por ser mais ser rápido.

O gerador do teste é construído chamando a função main, assim como o gerador do RTL, e passando alguns argumentos, sendo eles os apresentados na Tabela 29.

```
object ModuloTopoTester {
  def main(args: Array[String]): Unit = {
    println("Testing the ...")
    chiselMainTest(Array("--genHarness", "--test", "--backend", "c",
      "--compile", "--vcd", "--targetDir", "testgenerated"),
      () => Module(new ModuloTopo(8))) {
        f => new ModuloTopoTester(f)
      }
  }
```

Código 46: Função Main p/ geração de Simulador.

Opções	Descrição
–targetDir	target pathname prefix
–genHarness	generate harness file for C++
–debug	put all wires in C++ class file
–compile	compiles generated C++
–test	runs tests using C++ app
–backend	v generate verilog
–backend	c generate C++ (default)
–vcd enable	vcd dumping

Tabela 29: Argumentos construção do teste.

Como exemplo, veja o teste feito para a ULA do Código 37, é colocado o valor 10 em a, 12 em b e escolhido operação soma (fn = 0). Esses valores são atribuídos aos sinais durante 1 ciclo de relógio (mesmo sendo o circuito combinatório, não importa, é a forma que o Chisel cria tempo) e a simulação é encerrada.

```
package ulatester
import ula._
import Chisel._
class ULATester(dut: ULA) extends Tester(dut) {
  poke(dut.io.a, 10) // atribui 10 ao sinal a
  poke(dut.io.b, 12) // atribui 12 ao sinal b
  poke(dut.io.fn, 0) // soma
  step(1)             // espera 1 ciclo de relogio
}
object ULATester {
  def main(args: Array[String]): Unit = {
    println("Testing the ALU")
    chiselMainTest(Array("--genHarness", "--test", "--backend", "c",
      "--compile", "--vcd", "--targetDir", "testgenerated"),
      () => Module(new ULA(8))) {
        f => new ULATester(f)
}}}
```

Código 47: Teste básico para circuito da ULA.

Como foi escolhida a opção –vcd, foi gerada a *waveform* mostrada na Figura 44.

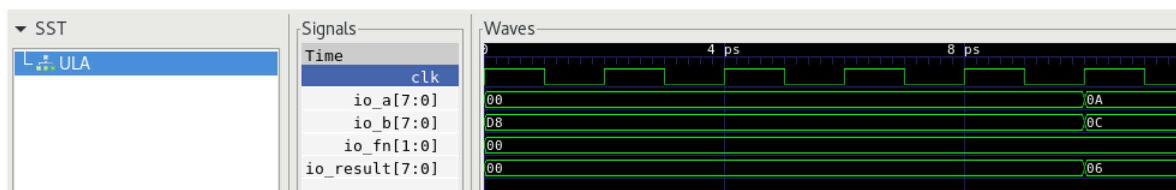

Figura 44: Waveform do teste da ULA.

7.7 Preparando o ambiente de programação Chisel

Nessa seção iremos preparar e entender o ambiente de programação Chisel no terminal Linux. Para a versão do Chisel usada nesse trabalho, versão Chisel2, é necessário instalar os seguintes softwares:

- JDK(Java Development Kit);
- SBT (Scala Build Tools).

Para a versão Chisel3 é acrescentado o Verilator e o FIRRTL. Mais informações podem ser obtidas na referência [39].

Como o Chisel está embutido no Scala, usa-se o mesmo compilador, o SBT. Além disso, esse compilador roda na JVM (Java Virtual Machine), por isso é preciso instalar também o JDK. Para instalar, os seguintes comandos devem ser efetuados:

```
$ sudo apt-get install openjdk-8-jre # p/ sistemas Ubuntu, Debian...
$ sudo dnf install java-1.8.0-openjdk # p/ sistemas CentOS, Fedora...
```

Código 48: Comandos para instalação do JDK no linux.

```
$ # Instalacao Ubuntu, Debian...
$ echo "deb https://dl.bintray.com/sbt/debian /" | '
sudo tee -a /etc/apt/sources.list.d/sbt.list'
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 642AC823
$ sudo apt-get update
$ sudo apt-get install sbt
$ # Instalacao CentOS, Fedora...
$ sudo dnf intall sbt
```

Código 49: Comandos para instalação do SBT no linux.

Com as ferramentas devidamente instaladas, é preciso criar a infraestrutura de diretórios, no modelo apresentado abaixo, para desenvolvimentos de projetos em Chisel (é o mesmo do Scala puro).

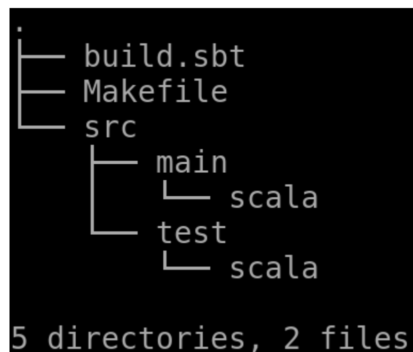

Figura 45: Infraestrutura de diretórios p/ projetos Chisel.

O diretório src/main/scala conterá todas as implementações dos circuitos com extensão .scala, de forma similiar o diretório src/test/scala recebe Códigos .scala, porém com implementações de testbenches. O Makefile é opcional, contém os comandos para compilação, geração de Verilog e geração de teste. O build.sbt é um conFigurador do compilador sbt. Ele será usado para definir a versão Scala usada e algumas bibliotecas extras, que no nosso caso é o Chisel. Veja o Código 50.

```
scalaVersion := "2.11.7"

// Only needed if not organized according to the sbt standard
// scalaSource in Compile := baseDirectory.value / "src"

// versao usada Chisel2
// use 2.2.30 till VCD issue is fixed
libraryDependencies += "edu.berkeley.cs" %% "chisel" % "2.2.38"

// ultima versao, precisa de Verilator!
//"chisel3" % "3.2-SNAPSHOT"

// This is from a locally published version
// libraryDependencies += "edu.berkeley.cs" %% "chisel" % "2.3-SNAPSHOT"
```

Código 50: Script build.sbt.

Outras versões Chisel estão comentadas, porém nesse trabalho, sempre usou-se a versão 2.2.30.

Por fim, a Tabela 30 descreve os comandos que serão usados para compilação, geração de Verilog e simulação.

Comando	Descrição
sbt compile	Compila todos os Códigos em src/-main.
sbt "runMain example.DriverMain"	Executa o método main no exemplo.Driver. É usado tanto para gerar Verilog, veja o Código 44.
sbt "test:runMain example.TestbenchMain"	Executa o método main no exemplo.TestbenchMain. É usado tanto para gerar simulador, veja o Código 46.

Tabela 30: Comandos SBT p/ compilação Chisel.

Essa é a infraestrutura básica para desenvolvimento em Chisel. Mais informações sobre comandos SBT, pode ser encontrada em [34]. Na próxima seção, é apresentado um exemplo de circuito sequencial, encerrando o estudo de Chisel feito no estágio.

7.8 FSM, um identificador de sequência binária

O objetivo dessa seção é mostrar um circuito com lógica sequencial, uma FSM (*Finite State Machine*), já que nas seções anteriores a maioria dos circuitos descrevem logica combinacional.

O circuito implementado foi um identificador de sequência de 4 bits. Esse circuito deve colocar o bit de saída (hit) em nível alto, sempre que a sequência definida pelo sinal de 4 bits seq for identificada e o sinal enable estiver ativo. A FSM da Figura 47, modela tal comportamento.

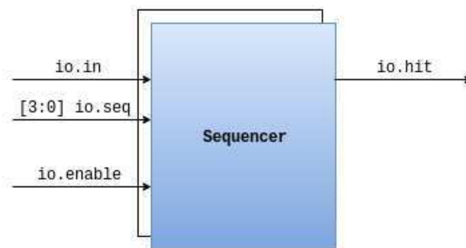

Figura 46: RTL Identificador de Sequência.

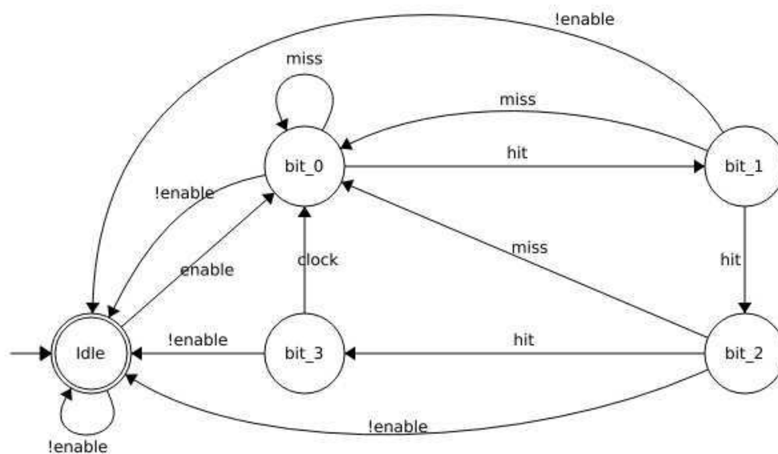

Figura 47: FSM para Identificador de Sequência.

Na infraestrutura de desenvolvimento em Chisel, foi criado um arquivo no diretório src/main com nome Sequencer.scala, implementando a FSM.

```
.
├── build.sbt
├── Makefile
└── src
    ├── main
    │   └── scala
    │       └── Sequencer.scala
    └── test
        └── scala

5 directories, 3 files
```

Figura 48: Diretório desenvolvimento projeto identificador de sequências.

```
package sequencer # cria pacote sequencer

import Chisel._ # importa lib Chisel

class Sequencer extends Module {
  val io = IO(new Bundle {
    // inputs
    val in = Bool(INPUT)
    val seq = UInt(INPUT, width = 4)
    val enable = Bool(INPUT)

    // output
    val hit = Bool(OUTPUT)
  })
  /* FSM States */
  val bit_0 :: bit_1 :: bit_2 :: bit_3 :: Nil = Enum(Bits(), 4)

  /* Wires and registers */
  val State = Reg(init = bit_0)

  /* FSM
   * read LSB to MSB
   */
   when(io.enable) {
    when(State === bit_0) {
      when(io.seq(0) === io.in) {
        State := bit_1
      }
      .otherwise {
        State := bit_0
      }
    }
    .elsewhen(State === bit_1) {
      when(io.seq(1) === io.in) {
        State := bit_2
      }
      .otherwise {
        State := bit_0
      }
    }
    .elsewhen(State === bit_2) {
      when(io.seq(2) === io.in) {
        State := bit_3
      }
      .otherwise {
        State := bit_0
```

```
      }
    }
    .elsewhen(State === bit_3) {
      // hit
      when(io.seq(3) === io.in) {
        io.hit := Bool(true)
        State := bit_0
      }
      .otherwise {
        State := bit_0
      }
    }
  }
  .otherwise {
    State := bit_0
    io.hit := Bool(false)
  }
}
```

Código 51: Implementação da FSM do circuito identificador de sequências.

Esse circuito foi compilado e gerado o Verilog no diretório generated/, com os respectivos comandos:

```
$ sbt compile
$ sbt "runMain sequencer.SequencerMain"
```

Código 52: Comandos para compilação da FSM identificador de sequências.

Por fim, foi escrito um testbench, SequencerTestbench.scala, e colocado no respectivo diretório src/test/scala/. Veja o Código de teste:

```
package sequencer
import Chisel._

class SequencerTestbench(s: Sequencer) extends Tester(s) {
  /* clear inputs during one cycle */
  poke(s.io.in, 0)
  poke(s.io.seq, 0)
  poke(s.io.enable, 0)
  step(1)
  /* basic test 1101 */
  poke(s.io.enable, 1)
  poke(s.io.in, 1)
  poke(s.io.seq, 13)
  step(1)
```

```
  poke(s.io.in, 0)
  step(1)

  poke(s.io.in, 1)
  step(1)

  poke(s.io.in, 1)
  step(1)

  poke(s.io.in, 0)
  poke(s.io.seq, 0)
  poke(s.io.enable, 0)
  step(1)

  /* random test */
  printf("\n\nRandom test!\n\n")
  poke(s.io.enable, 1)
  val rnd_seq = rnd.nextInt(0xF) // generates random between 0x0 to 0xF
  poke(s.io.seq, rnd_seq)

  val rnd_bit = rnd.nextInt(0xF)
  for(i <- 0 until 32) {
   val rnd_bit = rnd.nextInt(0xF)
   poke(s.io.in, rnd_bit)
   peek(s.io.in)
   step(1)
  }
}
object SequencerTestbench {
  def main(args: Array[String]): Unit = {
    chiselMainTest(Array(
      "--genHarness", "--test", "--backend", "c",
      "--compile", "--targetDir", "test", "--vcd"
       ),
      () => Module(new Sequencer)) {
        s => new SequencerTestbench(s)
      }
  }
}
```

Código 53: Implementação do teste do circuito identificador de sequências.

Esse Código foi compilado com o comando apresentado no Código 54 e foi gerando como saída, no diretório test/, o arquivo Sequencer.vcd com *waveforms* dos respectivos estímulos do teste. Estes resultados são apresentados nas Figuras 49 e 50.

```
$ sbt "test:runMain pem_sequencer.SequencerTestbench"
```

Código 54: Comandos para compilação do testbench do circuito identificador de sequências.

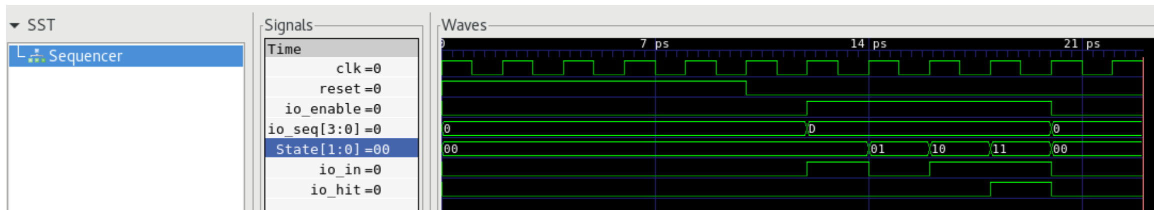

Figura 49: Waveform 1 testbench FSM.

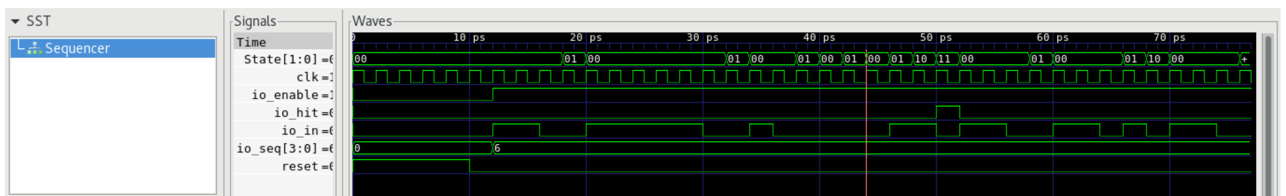

Figura 50: Waveform 2 testbench FSM.

8 Demais Ferramentas do Rocket Chip

O repositório do Rocket Chip contém uma game de diretórios adicionais que dão infraestrutura para desenvolvimento de software básico, geração de hardware, simulação e emulação. Nessa seção, serão apresentados os principais pontos dessas ferramentas, será explanado também sobre alguns aspectos de funcionamento da plataforma, como o Rocket Chip Tethered e o sistema de Bootloader e apresentados exemplos simples de uso das principais ferramentas.

Na Tabela abaixo são apresentados a descrição geral desses diretórios:

Diretório	**Descrição**
riscv-tools	Este repositório contém um conjunto de simuladores RISC-V, compiladores e outras ferramentas.
bootrom	Fontes para o bootloader de primeiro estágio incluído no BootROM (veja seção 6.5.1).
csrc	Contém fontes C para uso em simulação com Verilator.
emulator	Diretório no qual as simulações do Verilator são compiladas e executadas.
project	Diretório do projeto usado pelo SBT para compilação Chisel/S-cala.
regression	Define integração contínua.
scripts	Utilitários para analisar a saída de simulações ou manipular o conteúdo dos arquivos de origem.
vsim	Diretório no qual é gerado model Verilog do Rocket Chip preparado para ser simulado com Synopsys VCS.
vsrc	Fontes Verilog auxiliares contendo interfaces, *harnesses* e VPI (*Verilog Procedural Interface*) necessárias para construção do modelo Verilog final do Rocket Chip.

Tabela 31: Revisao dos diretórios do Rocket Chip.

8.1 Ferramentas de Desenvolvimento de SW - riscv-tools/

O repositório riscv-tools/ contém um conjunto de ferramentes que provê meios para compilação cruzada (*cross-compiling*), simulação e depuração. A Tabela apresentada a seguir descreve as ferramentas embutidas no riscv-tools/.

riscv-tools/	**Descrição**
riscv-fesvr/	Conjunto de SW relacionados ao Servidor Front-End (*Front-End Server* - fesvr). Este componente estabelece comunicação entre o Rocket Chip (HW) e *host* (PC), facilitando comunicação entre eles. Esse esquema é usado no simulador, emulador e FPGA.
riscv-gnu-toolchain/	Infraestrutura para criação de código de máquina. Aqui também contém o código fonte do simulador QEMU suportando duas arquiteturas RISC-V 32 e RISC-V 64 bits. Scripts adicionais também são fornecidos para automatizar o processo de compilação e geração de binários.
riscv-isa-sim/	Contém os códigos-fonte necessários para construir o simulador Spike.
riscv-opcodes/	Enumeração de todos os *opcodes* RISC-V executáveis pelo simulador.
riscv-openocd/	Ferramenta de depuração, o OpenOCD é executado no *host* e se conecta com *target* via algum adaptador de HW (eg JTAG) e possibilitando programação e depuração do mesmo.
riscv-riscv-pk/	O RISC-V Proxy Kernel, pk, é um ambiente leve de execução de aplicativos que pode hospedar binários ELF RISC-V vinculados estaticamente. Ele é projetado para suportar implementações RISC-V com acesso limitado à capacidade de E/S e, portanto, lida com chamadas de sistema relacionadas a E/S, fazendo proxy para um computador host. Este pacote também contém o Berkeley Boot Loader, bbl, que é um ambiente de execução de supervisor para sistemas RISC-V conectados. Ele é projetado para hospedar a porta RISC-V Linux.
riscv-riscv-tests/	Contém vários testes unitários para processadores RISC-V, além de *benchmarks* que podem ser usados para avaliar o desempenho do hardware.

Tabela 32: Revisao dos diretórios do Rocket Chip.

Nas próximas subseções serão abordados em mais detalhes o RISC-V GNU Compiler Toolchain e o simulador Spike.

8.1.1 RISC-V GNU Compiler Toolchain

O RISC-V GNU Compiler Toolchain é uma coleção de ferramentas de programação que permite fazer compilação cruzada [11](ou *cross-compiler*) de fontes C/C++ para produzir código de máquina para ISA RISC-V. Essa ferramenta suporta dois modos de construção de código: 1) genérico ELF/Newlib toolchain e mais sofisticado 2) Linux-ELF/glibc toolchain.

O projeto Rocket Chip fornece alguns fontes que permitem, com auxílio do compilador, construir código de máquina para os SoCs Rocket Chip construídos (HW) ou para serem executados no simulador da ISA RISC-V Spike. Esses códigos [12] são o entry.S, syscalls.c e o link.ld, e são detalhados em seguinda:

- O entry.S é o fonte que define o primeiro código a ser executado pelo sistema após o bootloader. Esse código é responsável por inicializar os registradores de propósito geral e configurar/gerenciar as interrupções do sistema, ao final ele executa instrução JUMP para função _init() definida em syscalls.c.
- O syscalls.c, além da implementação da função _init() (responsável por "chamar" a função main()), implementa outras funções comumente usadas em aplicações C, como por exemplo: putchar, printf, memcpy e outras funções. Essas funções se comunicam com o RISC-V Front-End Server (executado dentro do simulador ou emulador). Com o Spike ou Emulador sendo executado no host, eles lêem as requisições do Front-End Server e executam os serviços demandados pelo *target*, como e.g., a função printf().
- O link.ld, arquivo responsável por "ligar" (juntar ou unir) todos os fontes da aplicação, organizando seção de memória, entre outros.
- hearders.h, o Rocket Chip

A Figura 51 apresenta o fluxo de compilação cruzada de maneira simplificada. Deve-se ter atenção que dependendo da ISA do hardware, *flags* de configuração do compilador devem ser adicionadas, para que aja um **alinhamento entre código de máquina gerado e hardware alvo**. Essas *flags* podem ser encontradas nas documentações do RISC-V Toolchain.

[11]Um compilador cruzado é um compilador que é capaz de produzir código executável para uma plataforma diferente da qual o compilador está sendo executado.
[12]Os códigos citados podem ser encontrados no diretório riscv-tools/riscv-tests/env/.

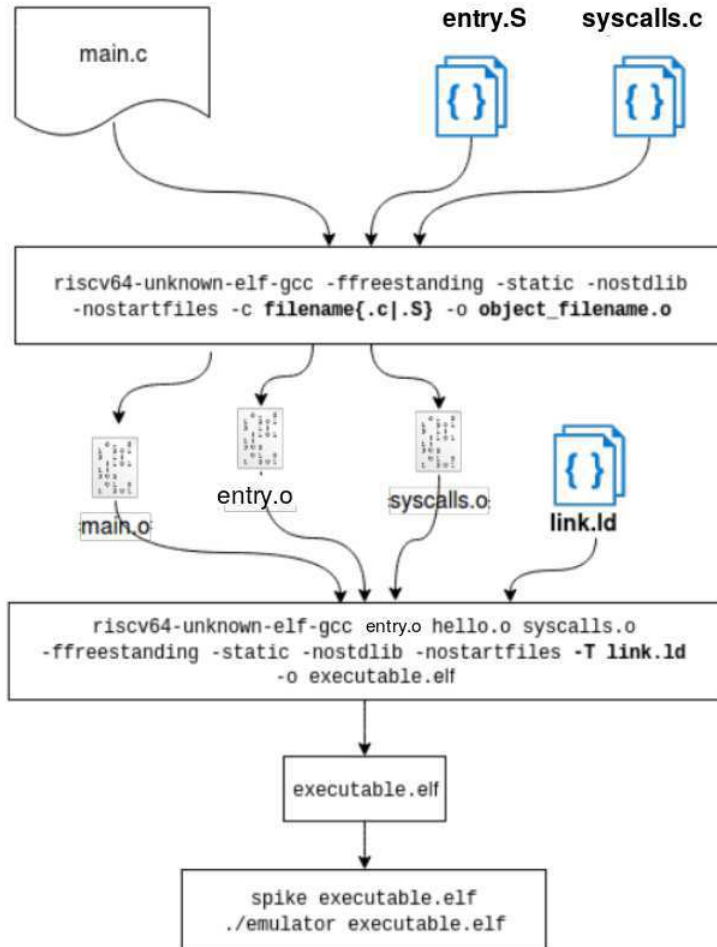

Figura 51: Fluxo de compilação cruzada RISC-V.

Essa ferramenta pode ser instalada, de forma separada do Rocket Chip, seguindo os passos apresentados contidos no repositório oficial do projeto `https://github.com/riscv/riscv-gnu-toolchain`. Na subseção seguinte, será apresentado o simulador da ISA RISC-V, o Spike, e em seguida um simples exemplo de uso.

8.1.2 Simulador de ISA RISC-V Spike

O Spike é um ISS [13] denominado "*Golden Model*" do RISC-V, implementando um modelo funcional de um ou mais processadores RISC-V. Ele serve como uma referência para testar o software RISC-V. Escrito por pessoas que iniciaram a revolução RISC-V. É sempre atualizado para respeitar a evolução da especificação oficial.

[13] Um simulador de conjunto de instruções (ISS - *Instruction Set Simulator*) é um modelo de simulação, geralmente codificado em uma linguagem de programação de alto nível, que imita o comportamento de um mainframe ou microprocessador "lendo"instruções e mantendo variáveis internas que representam os registros do processador. (Fonte: `https://en.wikipedia.org/wiki/Instruction_set_simulator`), traduzido livremente).

Spike modela sistema com único processador ou multi-processados, dependendo do argumento -pN_PROC, em que N_PROC é um número inteiro maior que 0 e representa o número de processadores a serem simulados. Cada núcleo inclui uma MMU para memória virtual e todos os processadores possuem um cache instrução (I1$) e dados (D1$) próprios, nível 1. Então, tanto I1$ como D1$ conectam-se a uma cache L2$ (compartilhada entre os processadores), que por fim conecta-se à memória principal. A Figura 52 mostra uma visão geral das principais classes usadas dentro do código-fonte do Spike.

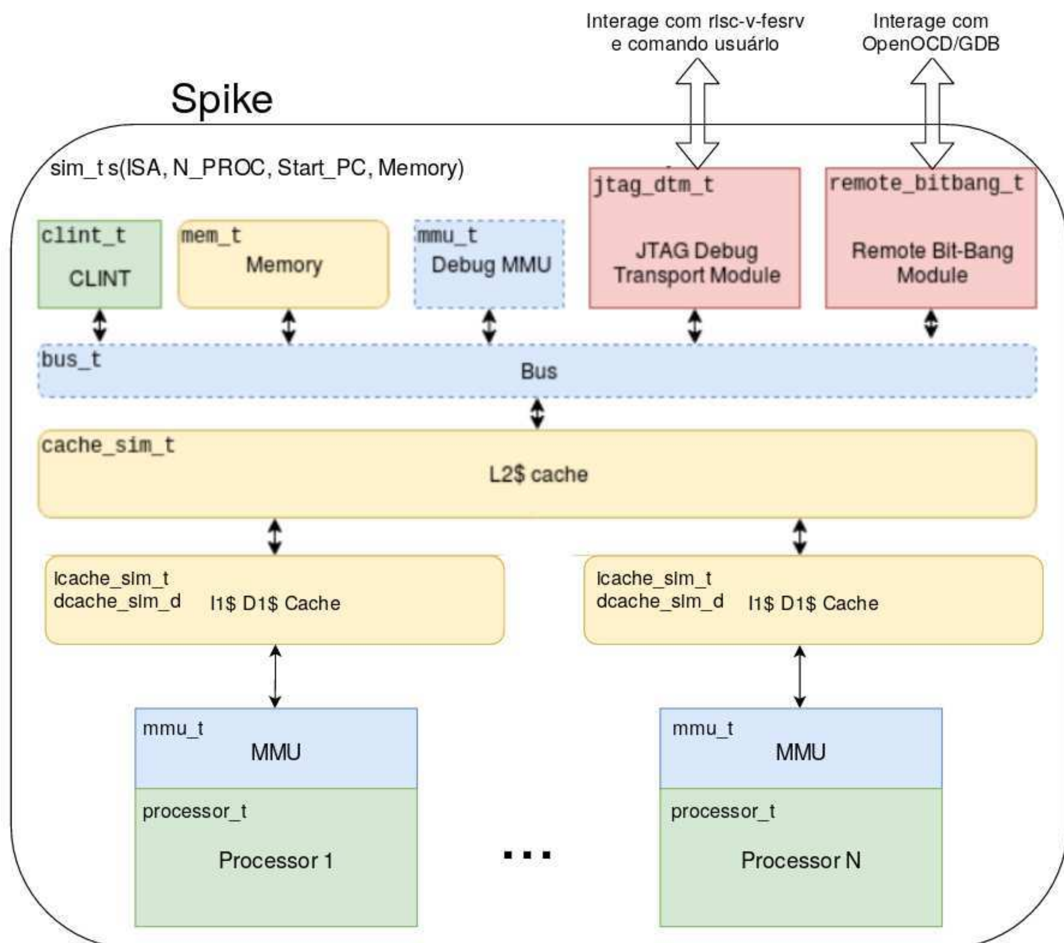

Figura 52: Arquitetura do model de simulação Spike.

A classe global sim_t é a classe principal. Ele contém todos os outros componentes e pode interagir com o usuário por comandos interativos para execução, parametrização, depuração de software.

Spike modela o comportamento de vários componentes de hardware reais. Por exemplo, processadores, MMU, cache, etc. Cada componente é modelado por uma classe de software em C++. O simulador foi projetado por relacionamentos de herança entre os componentes de software.

A configuração do hardware simulado pode ser feito com algumas *flags* passadas no momento de execução do código, por exemplo, –with-isa=rv32ima, para simular hardware com ISA RISC-V IMA de 32 bits, -d para depuração, entre outros. A configuração de hardware simulado *default* (onde não há passagem de comandos adicionais) é a ISA completa de 64 bits, i.e., RV64G com único processador. Mais informações sobre essas configurações podem ser encontradas no repositório oficial desse componente `https://github.com/riscv/riscv-isa-sim/`, além de informações para uso do OpenOCD e GDB com Spike.

Toda essa capacidade do Spike permite que ele execute softwares RISC-V de complexidade baixa (*baremetal*) à aplicativos mais avançados como Sistemas Operacionais de Tempo Real (RTOS) ou de Propósito Geral (GPOS).

Conclui-se que Spike é bastante útil para a simulação de softwares RISC-V, possibilitando a exploração de vários tipos de arquiteturas, inclusive com múltiplos processadores. Também é útil para simular o sistema de depuração do Rocket Chip junto ao OpenOCD e GNU GDB. O repositório riscv-tools/ contém esse componente internamente, com os passos de instalação detalhados. Na próxima seção é apresentado um exemplo básico de uso e os códigos usados.

8.1.3 Construindo um "Hello World!" e executando no Spike

Nessa seção, será feito um demonstração de compilação cruzada de uma aplicação básica, um simples "Hello World!". O objetivo é fornecer o conhecimento base para uso da ferramenta e reprodução dos resultados contidos nesse relatório. É preciso instalar [14] o riscv-tools/ para reprodução desse experimento.

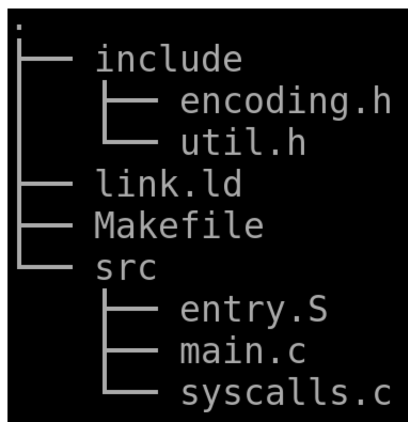

Figura 53: Árvore de arquivos para compilação cruzada.

[14] Download e passos de instalação do RISC-V GNU Compiler Toolchain no repositório oficial: `https://github.com/riscv/riscv-gnu-toolchain`.

Os códigos apresentados na Figura 53 foram explanados anteriormente na seção 8.1.1 e encontram-se em detalhes no Anexo A (página 137). Eles podem ser analisados e estudados para maior compreensão do funcionamento de baixo nível do SoC Rocket Chip.

É importante atentar-se ao alinhamento da construção do software e do hardware simulado, isto é, a ISA simulada no Spike deve executar o software *cross* compilado configurado com a mesma ISA. No exemplo apresentado aqui, a ISA simulada pelo Spike é a RV64G (configuração *default* como já fora dito anteriormente), assim sendo, o software básico construído segue a ISA, como pode ser visto nos códigos em anexo, mais especificamente no Makefile.

Instalada as ferramentes e construído a infraestrutura apresentada, pode-se executar a aplicação com os comandos apresentados no código 55. Com sucesso, a saída será identica ao da Figura 54.

```
$ make all
$ make sim # ou spike main
$ spike -d main # para entrar em modo debug
```

Código 55: Comandos para exemplo Spike.

```
tdlib -nostartfiles -std=gnu99 -O0 -ffast-math -fr
ain.o -c ./src/main.c
riscv64-unknown-elf-gcc -Tlink.ld -I./include  ./s
tic -nostdlib -nostartfiles -lm -lgcc -Tlink.ld -c
riscv64-unknown-elf-size main
   text    data     bss     dec     hex filename
   7118      72     172    7362    1cc2 main
[gvillanova@localhost hello-world]$ make sim
Hello World!
```

Figura 54: Saída de compilação cruzada do exemplo Spike.

8.2 Gerando Verilog de SoC Rocket Chip - vsim/

O vsim/ é um repositório que permite gerar o modelo de HW do Rocket Chip em Verilog, ou seja, ele transforma o SoC escrito em Chisel para linguagem Verilog. Isso permite que ferramentas de simulação e síntese, como Synopsis VCS e ModelSim, possam ser usadas para testar e avaliar o Rocket Chip.

O uso do vsim/ pode ser dividido em três etapas, a primeira é a configuração do HW do Rocket Chip, isto é, o usuário precisa configurar, personalizar e/ou adicionar todos os componentes que deseja para construção do SoC, isso é feito nas classes apresentadas anteriormente na seção 6. Em seguida, é preciso configurar o módulo topo, i.e., o código rocket-chip/src/main/scala/system/Configs.scala. Esse código permite a escolha da quantidade de processadores e a adição/remoção dos demais componentes personalizados na etapa anterior, que são os componentes majoritariamente disponíveis no código rocket-chip/src/main/scala/subsystem/Configs.scala. Por fim, aproveitando-se do Makefile do repositório vsim/ é possível gerar o sistema personalizado em Verilog.

Abaixo é apresentado alguns exemplos do código system/Configs.scala.

```
// default config. com core RV64G
class DefaultConfig extends Config(new WithNBigCores(1) ++ new BaseConfig)

// implementa core RV32G
class DefaultRV32Config extends Config(new WithRV32 ++ new DefaultConfig)

// dual core RV64G sem sistema de coerencia cache
class DualCoreIncoherentTile extends Config(
        new WithIncoherentTiles ++
        new WithNBigCores(2) ++ new BaseConfig)

// quad core RV64G sem FPU
class QuadCoreNFPU extends Config(
        new WithoutFPU ++
        new WithNBigCores(4) ++ new BaseConfig)

// core RV64G com interface JTAG adicionada
class DefaultConfigJTAG extends Config(
        new WithJtagDTMSystem ++
        new WithNBigCores(1) ++ new BaseConfig)

// core RV64G com coprocessador RoCC
class RoccExConfig extends Config(new WithRoccExample ++ new DefaultConfig)
```

Código 56: Exemplos de implementações do Rocket Chip.

O código 57, ajuda no entendimento do padrão de construção do sistema como um todo. Perceba que basicamente, é preciso escolher a quantidade de processadores e adicionar/remover os componentes definidos em subsystem/Configs.scala. Os componentes de subsystem/Configs.scala podem ser alterados ou os sub-componentes a eles associados, descendo o nível o quanto for necessário para resolver a especificação desejada. Um código que vale a pena destacar é o system/TestHarness.scala, ele implementa os sinais de E/S do chip, portanto, pode ser modificado para receber sinais de periféricos e coloca-los em acesso para fora do chip, com os exemplos de códigos é possível fazer esse tipo de personalização.

Com a personalização finalizada, basta executar o comando abaixo no diretório vsim/ indicando qual modelo construir, nesse caso, escolheu-se o RoccExConfig, ou seja um SoC com único processador implementando ISA RV64G, adicionado de um Coprocessador RoCC. O resultado desse comando é o modelo Verilog do SoC.

```
$ make verilog CONFIG=RoccExConfig
```

Código 57: Exemplos de implementações do Rocket Chip.

No processo de construção do Verilog, algumas informações do sistema construído são apresentadas, como DTS (*Device Tree Source*) e o mapa de endereço gerado. Para o exemplo em questão, as saídas foram as apresentadas pelas Figuras 55, 56 e 57.

```
Generated Address Map
               0 -     1000 ARWX  debug-controller@0
            2000 -     3000 ARW   customperipherals@2000
            3000 -     4000 ARWX  error-device@3000
           10000 -    20000  R XC rom@10000
         2000000 -  2010000 ARW   clint@2000000
         c000000 - 10000000 ARW   interrupt-controller@c000000
        60000000 - 70000000  RWX  mmio-port-axi4@60000000
        70000000 - 80000000  RWX  custom-axi4-port@70000000
        80000000 - 90000000  RWXC memory@80000000
```

Figura 55: Mapa de endereços gerados com implementação do RoccExConfig.

Esse *log* de informação é bastante útil para depurar se a especificação está de acordo, assim como auxiliar nas construções dos softwares, tendo em vista que o mapa de memória real é disponibilizado.

```
/dts-v1/;

/ {
	#address-cells = <1>;
	#size-cells = <1>;
	compatible = "freechips,rocketchip-unknown-dev";
	model = "freechips,rocketchip-unknown";
	L15: cpus {
		#address-cells = <1>;
		#size-cells = <0>;
		L5: cpu@0 {
			clock-frequency = <0>;
			compatible = "sifive,rocket0", "riscv";
			d-cache-block-size = <64>;
			d-cache-sets = <64>;
			d-cache-size = <16384>;
			d-tlb-sets = <1>;
			d-tlb-size = <32>;
			device_type = "cpu";
			i-cache-block-size = <64>;
			i-cache-sets = <64>;
			i-cache-size = <16384>;
			i-tlb-sets = <1>;
			i-tlb-size = <32>;
			mmu-type = "riscv,sv39";
			next-level-cache = <&L7>;
			reg = <0>;
			riscv,isa = "rv64imafdc";
			status = "okay";
			timebase-frequency = <1000000>;
			tlb-split;
			L3: interrupt-controller {
				#interrupt-cells = <1>;
				compatible = "riscv,cpu-intc";
				interrupt-controller;
			};
		};
	};
	L7: memory@80000000 {
		device_type = "memory";
		reg = <0x80000000 0x10000000>;
	};
```

Figura 56: DTS do exemplo RoccExConfig - A.

```
L14: soc {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "freechips,rocketchip-unknown-soc", "simple-bus";
        ranges;
        L1: clint@2000000 {
                compatible = "riscv,clint0";
                interrupts-extended = <&L3 3 &L3 7>;
                reg = <0x2000000 0x10000>;
                reg-names = "control";
        };
        L9: custom-axi4-port@70000000 {
                #address-cells = <1>;
                #size-cells = <1>;
                compatible = "simple-bus";
                ranges = <0x70000000 0x70000000 0x10000000>;
        };
        L11: customperipherals@2000 {
                compatible = "ucbbar,customperipherals";
                interrupt-parent = <&L0>;
                interrupts = <3 4 5 6>;
                reg = <0x2000 0x1000>;
                reg-names = "control";
        };
        L2: debug-controller@0 {
                compatible = "sifive,debug-013", "riscv,debug-013";
                interrupts-extended = <&L3 65535>;
                reg = <0x0 0x1000>;
                reg-names = "control";
        };
        L12: error-device@3000 {
                compatible = "sifive,error0";
                reg = <0x3000 0x1000>;
                reg-names = "mem";
        };
        L6: external-interrupts {
                interrupt-parent = <&L0>;
                interrupts = <1 2>;
        };
        L0: interrupt-controller@c000000 {
                #interrupt-cells = <1>;
                compatible = "riscv,plic0";
                interrupt-controller;
                interrupts-extended = <&L3 11 &L3 9>;
                reg = <0xc000000 0x4000000>;
                reg-names = "control";
                riscv,max-priority = <7>;
                riscv,ndev = <6>;
        };
        L8: mmio-port-axi4@60000000 {
                #address-cells = <1>;
                #size-cells = <1>;
                compatible = "simple-bus";
                ranges = <0x60000000 0x60000000 0x10000000>;
        };
        L10: rom@10000 {
                compatible = "sifive,rom0";
                reg = <0x10000 0x10000>;
                reg-names = "mem";
        };
};
```

Figura 57: DTS do exemplo RoccExConfig - B

8.3 Emulador - emulator/

Para testar os binários construídos com o compilador RISC-V, os desenvolvedores podem usar o Spike para verificar sua conformidade com a especificação do projeto. Mas, o Spike não forneça simulações CABA (*cycle-accurate, bit-accurate*), i.e, que leva em consideração todas as restrições enfrentadas no hardware real, como tempo de processamento. Esse problema é solucionado pelo emulador contido no *framework* Rocket Chip.

Antes do uso do emulador, vamos entender primeiro como ele é construído. Pois bem, o Emulator, também chamado de "*Verilated Emulator*" ou C-Emulator é construído a partir de um conjunto de softwares, sendo eles: o modelo Verilog do SoC gerado pelo Chisel, fontes Verilog complementares (disponíveis no diretório vsrc/) e demais códigos C++ disponíveis nos diretórios csrs/ e riscv-tools/riscv-fesrv.

Esse conjunto de fontes são compilados usando o Verilator. O Verilator é uma ferramenta desenvolvida em C++, de código aberto, e é capaz de transformar modelos Verilog de HW em modelos C++ equivalentes. Além disso, permite que o modelo gerado seja estimulado, possibilitando o desenvolvimento de complexos *testbenches*, graças aos recursos da linguagem no qual o Verilator é construído. A saída de compilação do Verilator é um executável C++, que ao ser executado, roda o *testbench* desenvolvido.

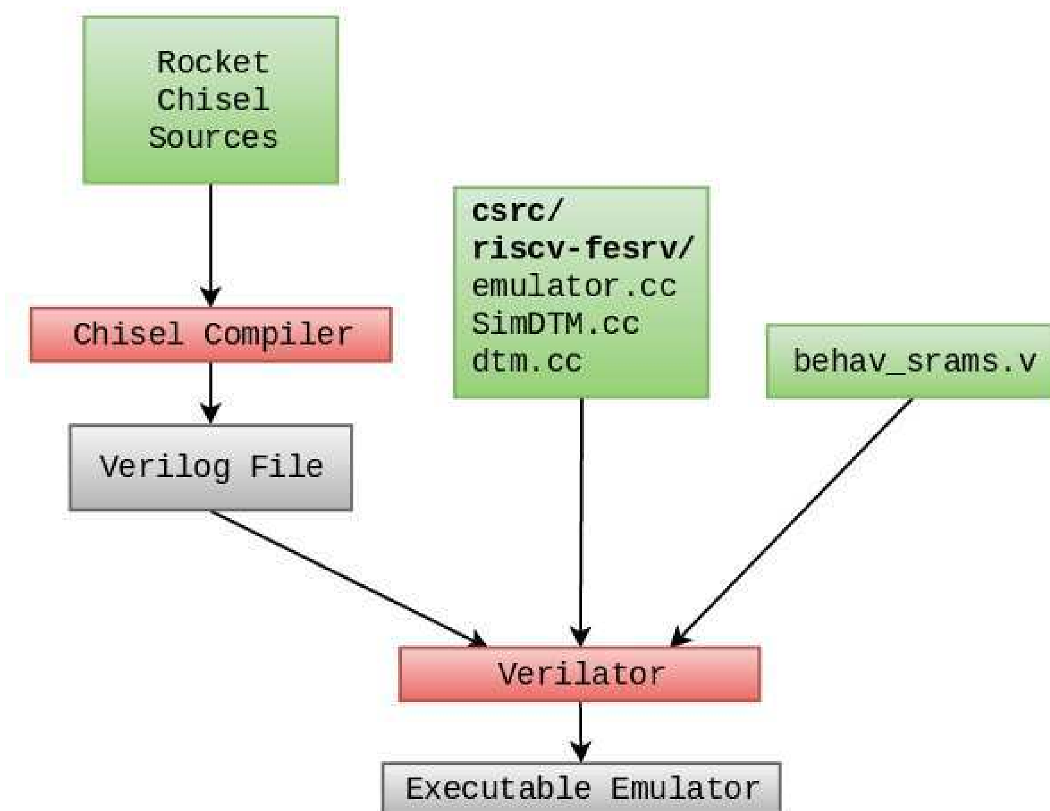

Figura 58: Fluxo de geração do emulator

Os códigos csrc/ e riscv-fesrv/, apresentados na Figura 58, implementam o módulo topo do emulador. Basicamente, os fontes contidos nesses diretórios estabelecem conexão com a interface DTM ou JTAG (isso depende da configuração do HW e do emulador) com o SoC Rocket Chip gerado. Os fontes presentes nos diretórios vsrc/, complementam o HW gerado pelo Chisel com módulos de interface do sistema, como SimDTM.v e SimJTAG.v, esses módulos possuem interface DPI [15] permitindo que os fontes C++ estimulem o hardware e acessem a plataforma. O fonte behav_srams.vé responsável por simular comportamento da RAM (memória principal) e é construído em Chisel.

A compilação desse conjunto de fontes pelo Verilator gera como saída um executável C++, o emulator propriamente dito, como mostra o fluxo da Figura 58. Esse executável aguarda como entrada do usuário algum binário construído com a toolchain RISC-V. Executando o emulador e passando como argumento um binário, o *testbench* (emulador) executa todos os processos seguintes: através das interfaces DTM ou JTAG, transporta o código binário para a memória principal do sistema, depois, passa o controle para o Rocket Chip que executará a aplicação presente na RAM. O detalhe desse esquema de boot será visto na próxima seção.

A Tabela 33 apresenta as várias opções que o emulador oferece para o usuário. Na seção 8.3.2 será demonstrado o uso do mesmo facilitando o seu entendimento.

Opções	**Descrição**
-c, –cycle-count	Imprima a contagem de ciclos.
-h, –help	Imprime o help do emulador e sai.
-m, –max-cycles=CYCLES	Finaliza a emulação até o ciclo CYCLES.
-s, –seed=SEED	Usa número aleatório para SEED.
-V, –verbose	Habilita todos os printfs do Chisel (*cycle-by-cycle info*).
-v, –vcd=FILE	Gera arquivo vcd da emulação.
-x, –dump-start=CYC	Inicia geração de arquivo vcd no ciclo CYCLE.
Emulator Verilog Plusarg	
+tilelink_timeout=INT	Finaliza emulação após INT esperas do ciclo TileLink. Off if 0.
+max-core-cycles=INT	Finaliza emulação após INT rdtime ciclos. Off if 0.

Tabela 33: Opções de comandos do C-Emulator.

A Figura 59, ilustra como todo esse conjunto de códigos apresentados se relacionam entre si. Perceba que a única interface de E/S do sistema é o SimDTM ou JTAG (que é opcional). Essas interfaces são conectadas no emulador através do conjunto de softwares representados por emulator.cc, que provê também, meios para que o utilizador possa usar o SoC, permitindo avaliar execuções e depuração de aplicações em um HW sintetizável. No repositório oficial [16] é exemplificado como se conectar com o JTAG (emulado) do sistema usando o GDB e OpenOCD.

[15]DPI é uma interface que pode ser usada para interface do SystemVerilog com outras linguagens de programação como C, C++, SystemC e outras.

[16]Repositório oficial Rocket Chip: `https://github.com/freechipsproject/rocket-chip`

8.3.1 Bootloader e Rocket Chip Tethered

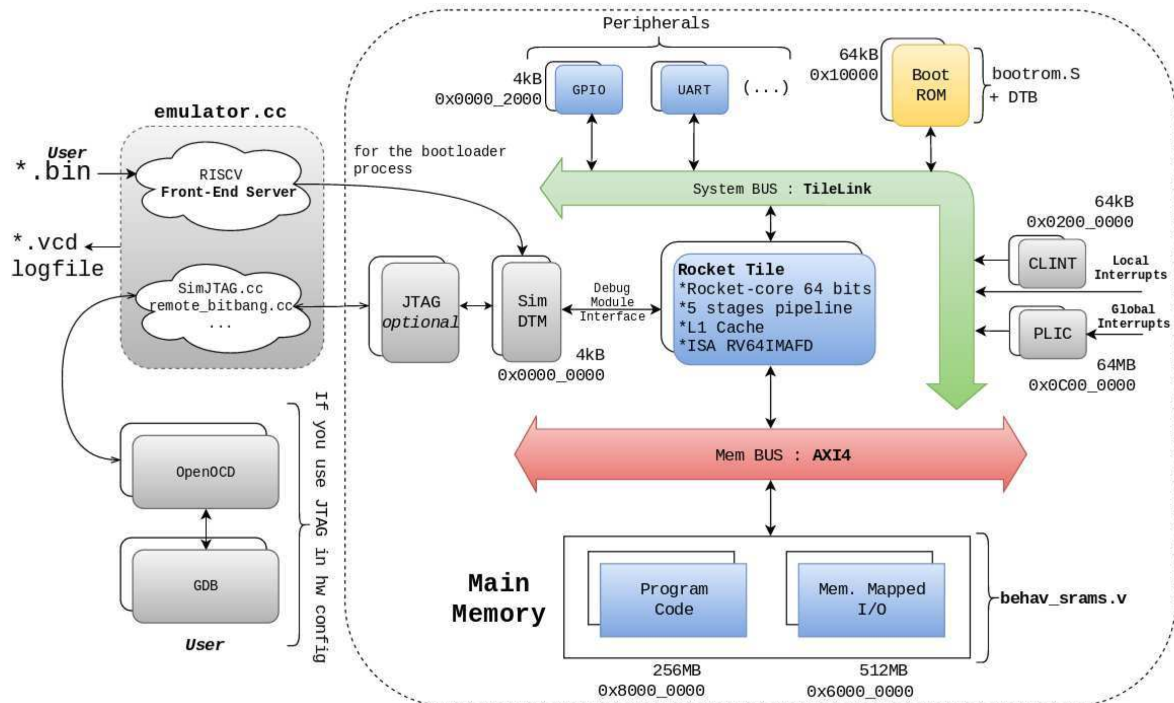

Figura 59: Interface de acesso ao SoC Rocket Chip.

O processo de bootloader [17] e de execuação do Rocket Chip é dado como segue:

1) Rocket-chip sempre "inicia"executando um código fora do bootROM. Esse código é apenas uma linha de código que diz "gire aqui para sempre". Isso acontece com fidelidade.

2) O riscv-fesvr (rodando no emulador ou FPGA) usa a interface DTM para interromper o Rocket Core e injetar pequenos trechos de código a serem executados no Rocket Chip, no modo de depuração, um modo privilegiado mais alto que o modo de máquina. Então, o binário (construído com RISC-V Toolchain) é carregado, palavra por palavra na RAM através do DTM. Uma vez que o programa é totalmente carregado no espaço de memória do Rocket Chip, o DTM força Rocket Chip dar um JUMP para iniciar a execução do programa, no endereço 0x80000000.

3) Quando o Rocket Chip termina de executar o programa, ele grava um código de saída na localização da memória "tohost" (veja entry.S, seção 8.1.1) que reside no espaço de memória do Rocket Chip. O código de saída é armazenado na memória como (exitcode « 1 | 1). O riscv-fesvr periodicamente acessa o local da memória pelo DTM para ler "tohost". Se ele vir a ser 1 no LSB, a execução da emulação é finalizada e o código de saída é informado. Esse esquema pode ser verificado, analisando os códigos do Anexo A.

[17]Dados retirados e modificados de discussão Google HW-Devel: `https://groups.google.com/a/groups.riscv.org/forum/m/#!topic/hw-dev/Pv8jUk0DzKI`.

4) Se o riscv-fesvr enxergar na memória "tohost" um valor diferente de zero, mas um 0 no LSB, ele reconhece isso como um local de memória que está mantendo uma chamada syscall (a inteligência aqui é que os locais da memória estão sempre alinhadas com meia palavra, então podemos fazer o trabalho duplo aqui e compartilhar o mesmo mecanismo de comunicação para sair das simulações e do syscalls). Um printf pode ser manipulado pelo riscv-fesvr lendo a memória do Rocket Chip conforme especificado pelo syscall armazenado no local da memória tohost. Esse protocolo ele Rocket Chip e Front-End Server permite que os serviços oferecidos no syscall.c (ver seção 8.1.1) sejam atendidos. Como é um projeto aberto, pode-se investigar os códigos do riscv-fesrv e demais para entender a fundo o mecanismo.

Esse sistema (com Front-End Server) também é usado quando deseja-se sintetizar o Rocket Chip em FPGA. Por conta disso, é preciso que a FPGA possua um processador capaz de rodar um sistemas Linux (FPGAs-SoC, com um ARM A9, por exemplo), onde o Linux deve construir em seu sistema a aplicação riscv-fesrv. Essa aplicação deve se conectar ao Rocket Chip sintetizado, fazendo todos os passos apresentados anteriormente. Foi assim que os desenvolvedores do Rocket Chip o projetaram [18]. Essa implementação ficou conhecida como Rocket Chip Tethered (ou "ligado"). A Figura 60 ilustra como ficam interligadas os componentes na FPGA.

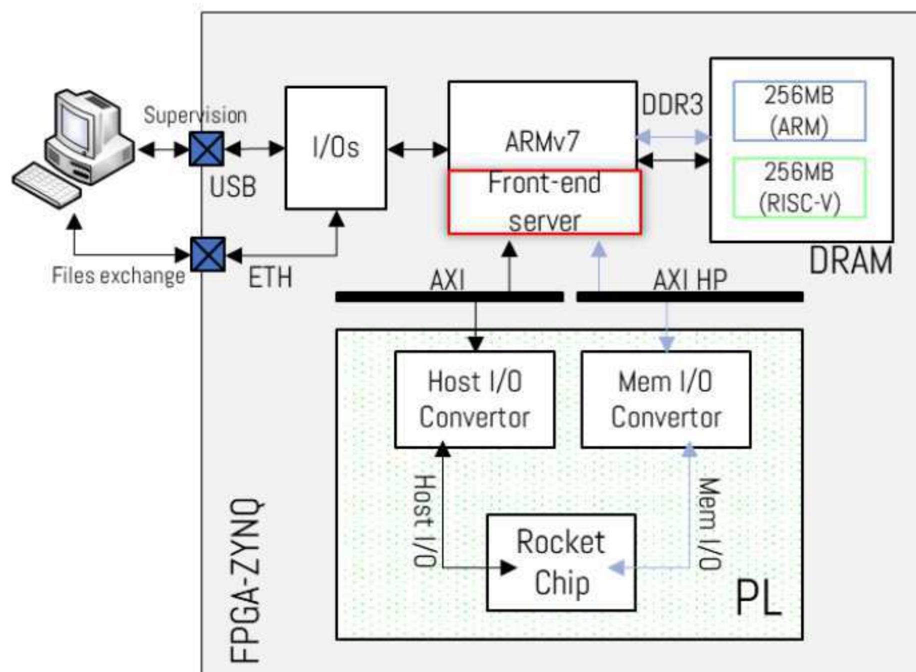

Figura 60: Arquitetura do Rocket Chip Tethered.

[18]Mais detalhes desse esquema pode ser conseguido no repositório `https://github.com/ucb-bar/fpga-zynq`, que fornece a infraestrutura necessária para sintetizar o Rocket Chip em FPGA Zynq da empresa Xilinx.

Essa solução é problemática, pois aumenta a latência do acesso a memória principal, já que o Rocket Chip não tem acesso direto a ela. Na verdade, acessos a memória são capturados pelo processador ARM que através do servidor front-end envia ou guarda dados (instruções LOAD/STORE). Os desenvolvedores do Rocket Chip ainda não disponibilizaram uma versão *Untethered* ("não ligada"), porém o projeto SiFive [19] e LowRISC já possuem implementações *Untethered* derivadas do Rocket Chip e são abertos, servindo, portanto, como referência para conseguir-se gerar HW Rocket Chip "não ligado".

8.3.2 Exemplo de uso do Emulador

Como forma de exemplificar toda essa infraestrutura, será apresentado nessa seção como construir o emulador de um sistema Rocket Chip. Esse conhecimento de base possibilita explorar completamente o HW Rocket Chip, e portanto, permite que se possa reproduzir todos os resultados presentes nesse relatório.

O gerador de emulador encontra-se no diretório rocket-chip/emulator/. Nesse diretório existe um Makefile que automatiza o processo de construção do emulador, discutido na seção anterior. Basicamente, ele aguarda como entrada do usuário a configuração do sistema que ele deseja construir (variável CONFIG), essa configuração, como já fora dito, deve ser feita também pelo usuário (veja o Código 57).

Pode-se construir dois tipos de emuladores: 1) a versão padrão que apenas executa o software e imprime saídas, e 2) a versão "debug" (precisa indicar esse argumento), que além dos recursos padrão (imprimir saídas), pode gerar arquivos VCD (*Value Change Dump*), contendo os *waveforms* do sistema e também tem a capacidade de executar a depuração usando as ferramentas GDB e OpenOCD [20].

No exemplo, será construído o emulador da configuração DefaultConfig do Rocket Chip em modo *debug*. Essa configuração implementa um SoC com único processador com ISA RV64G. Então, dentro do diretório do Rocket Chip, basta executar as linhas de comando abaixo. Ao final o executável **emulator-freechips.rocketchip.system-DefaultConfig-debug** deve ser gerado.

```
$ cd emulator
$ make debug CONFIG=DefaultConfig
```

Código 58: Construção de um emulador para configuração DefaultConfig.

[19]Mais informações sobre o projeto em `https://www.sifive.com/`.

[20]O sistema de depuração do emulador não fora usado, porém no repositório ofical do Rocket Chip existe um tutorial de como usar essas ferramentas como o emulador.

Em seguida, deve-se construir o software que será executado no sistema, lembrando que esse SW deve conter a mesma ISA do HW. A mesma infraestrutura desenvolvida na seção 8.1.3 pode ser usado, inclusive pode-se usar as mesmas configurações do Makefile, pois o HW em questão é o mesmo, alinhando portanto SW e HW. Nesse exemplo, o main.c usado escreve no primeiro endereço da região de memória MMIO a mensagem “Hello!”, como pode ser visto abaixo:

```
#define MMIO_ADDRESS 0x60000000

int main(void) {
        char msg[] = "Hello!";
        unsigned long int msg_sent=0;

        for (int i = 0; i < 6; ++i)
                msg_sent = (unsigned long int)msg[i]<<((6-i)*8)|msg_sent;

        *(volatile unsigned long int*)(MMIO_ADDRESS) = msg_sent;

        return 0;
}
```

Código 59: main.c para exemplo com Emulador.

Compilado esse código, o sistema pode ser emulado usando o executável do emulador gerado e passando como argumentos: alguma das opções mostradas na Tabela 33 e o binário construído anteriormente (main). Para o exemplo, pretende-se verificar a forma de onda, para isso, basta executar o comando abaixo.

```
$ ./emulator-(...)-DefaultConfig-debug -v hello.vcd main
```

Código 60: Emulando o sistema DefaultConfig.

Ao final desse comando (pode demorar alguns segundos), o arquivo hello.vcd estará disponível. Com auxílio do software gtkwave, pode-se verificar as formas de onda do sistema executando o comando abaixo. Esse arquivo contém todos os sinais do sistema, permitindo verificar registradores, região de memória, componentes aritméticos, execução do boot, enfim, todos os módulos do Rocket Chip construído. A Figura 61 apresenta a execução do código 59. É interessante perceber o tempo até a mensagem “Hello!” aparecer, isso acontece por conta do sistema de bootloader discutido na seção anterior.

```
$ gtkwave hello.vcd
```

Código 61: Abrindo arquivo VCD gerado pelo emulador.

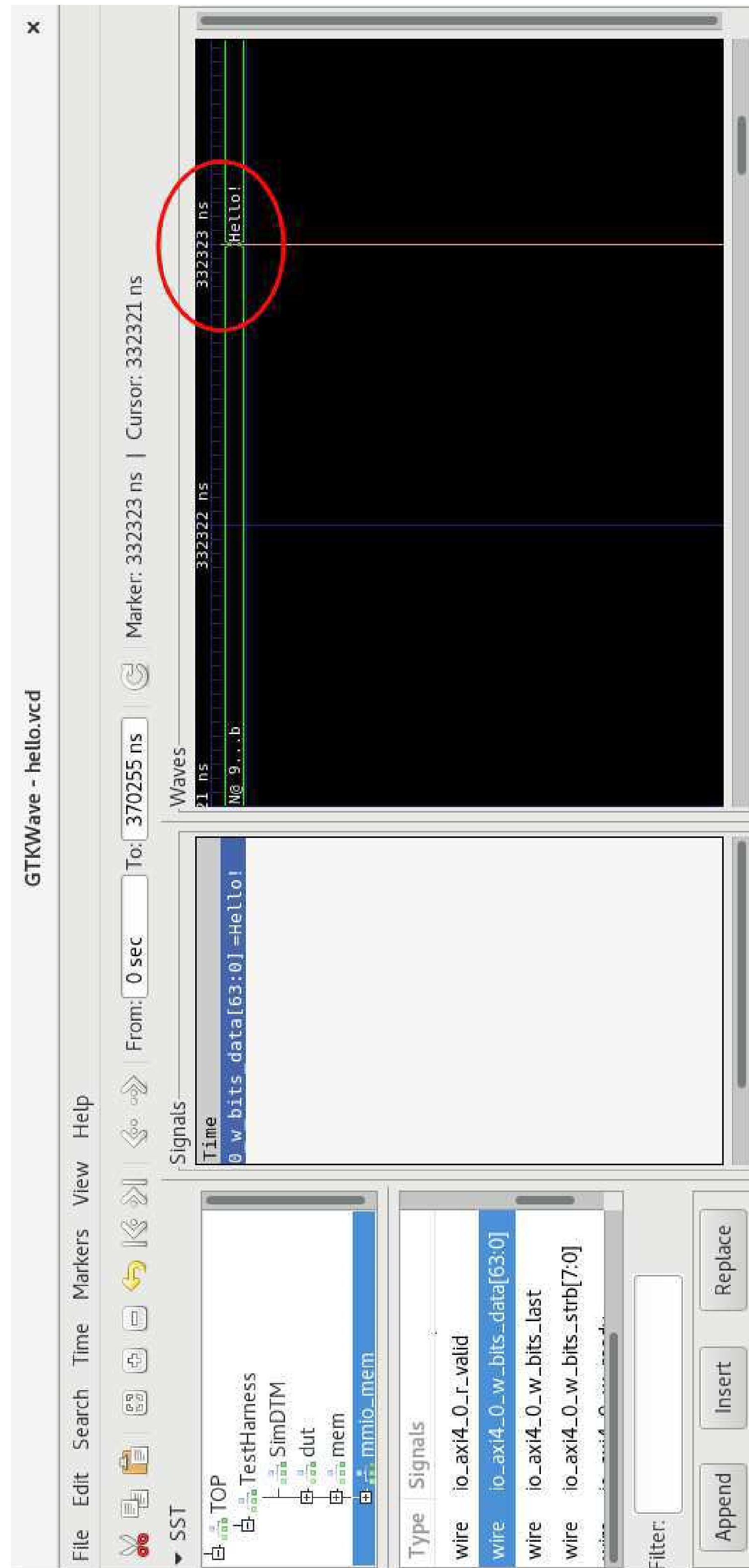

Figura 61: Verificando mensagem escrita na região de memória MMIO.

Uma observação importante: pode-se usar os serviços do syscalls.c com o emulador, como por exemplo a função printf(). Deve-se ter em mente que o emulador está executando o Front-End Server, assim como o Spike, logo, serviços como printf() estão disponíveis para o emulador também.

9 Conclusão

Após vários testes, análise de códigos, uso do simulador, emulador, estudos de artigos públicados, entre outros, sobre a plataforma Rocket Chip, foi possível gerar essa primeira versão de documentação, que engloba quase tudo o que é oferecido pelo *framework*, como pode ser visto nas seções de 5 a 8. Isso serve como base de estudo, para treinar outras pessoas a usar o Rocket Chip, contribuindo para a pesquisa e ensino.

Porém, devido ao trabalho contínuo dos desenvolvedores na plataforma, esse documento deve ser constantemente revisado e melhorado, para tentar sempre manter a coerência das informações. Além disso, outros pontos não abordados mais profundamente, ou até mesmo não abordados podem ser adicionados posteriormente, tornando o documento cada vez mais completo.

Em seguida, será apresentada a última parte do trabalho, o desenvolvimento de um Filtro FIR, com ideias de Computação Aproximada, no coprocessador do Rocket Chip, o RoCC.

Parte IV

Desenvolvimento de Hardware AxC no Rocket Chip

10 Filtro FIR AxC

Essa Seção tem como objetivo apresentar uma arquitetura de Filtro de Resposta Infinita (ou Filtro FIR, do inglês *Filter Impulse Response*). Em seguida, será apresentado, por meio de figuras e equações, as alterações feitas nesse Filtro para processamento aproximado, uma técnica de AxC para o Filtro. Por fim, as implementações serão apresentadas, assim como os resultados obtidos, permitindo avaliar como essa técnica afeta o desempenho, o consumo de energia e a qualidade do resultado do filtro, que são os parâmetros de interesse em Computação Aproximada.

10.1 Filtro FIR Serial

Existem varias formas de projetar Filtros FIR, sendo as mais comuns a arquitetura forma paralela e serial. Uma implementação paralela direta é composta de (N + 1) registradores que formam uma linha de atraso, (N + 1) multiplicadores para calcular os diferentes (x_i x h_j) produtos e N somadores para formar y_k. Essa arquitetura geralmente necessita de mais hardware.

Uma forma de reduzir o hardware do FIR paralelo é desenvolvendo um FIR serial. Esse último usa apenas um multiplicador, um somador com um registrador de armazenamento para os resultados parciais e uma ROM (ou qualquer outra forma de armazenamento) dedicada para fornecer os coeficientes h_i do filtro. Porém, requer uma unidade de controle para gerenciar a sequência de transição dos dados no *datapath*. O preço que se paga por ter menos hardware é tempo de processamento, que pode ser mais elevado em relação ao paralelo. A Figura 62 apresenta a arquitetura do filtro serial estudada.

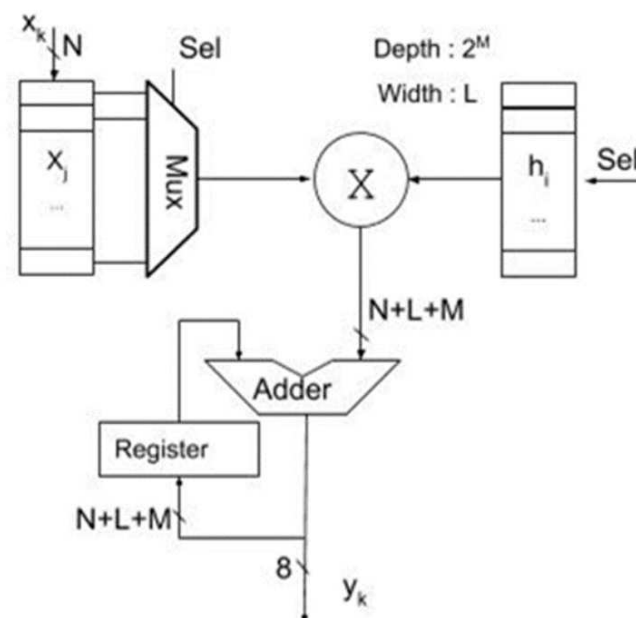

Figura 62: Arquitetura de Filtro FIR Serial. Fonte: [40].

Perceba que essa arquitetura é capaz de implementar a equação 2, equação de um filtro FIR: existem 2 *buffers*, um para propagar o sinal de entrada e outro com os coeficientes, então, de forma simples, o controle do filtro deve multiplicar a respectiva entrada com o respectivo coeficiente (equação 1) e acumular o resultado no registrador até que todas as multiplicações e soma se completem, respeitando a equação abaixo. Quando todo o acúmulo é feito a saída torna-se válida, uma nova amostra é colocada no *buffer* $x[n]$ e o ciclo se repete.

$$y[n] = h[0]x[n] + h[1]x[n1] + h[2]x[n2] + ... + h[N]x[nN] \tag{1}$$

$$y[n] = \Sigma_{n=0}^{N} x[n-i]h[i] \tag{2}$$

Esse filtro pode ser estudado mais projundamente no artigo [40]. Uma abordagem mais formal também pode ser conseguida através das diversas bibliografias sobre filtros. Na Seção seguinte, será apresentado as modificações feitas nesse filtro com técnica de AxC.

10.2 Filtro FIR em paradigma AxC

O método de aproximação usado avalia o quão próximo uma amostra está da outra. Quando a diferença dessas amostras esta abaixo de um limiar (T), que pode ser definido e controlado, algumas multiplicações [21] podem ser não executadas no sistema.

$$|x[n] - x[n-1]| < T \Rightarrow x[n] \approx x[n-1] \tag{3}$$

$$x[n] \approx x[n-1] \Rightarrow y[n] = (h_0 + h_1)x[n-1] + h_2 x[n-2] + ... + h_n x[n-N] \tag{4}$$

$$x[n] \approx x[n-1] \approx ... \approx x[n-k] \Rightarrow y[n] = (h_0 + h_1 + ... + h_k)x[n-k] + ... + h_n x[n-N] \tag{5}$$

Na equação 3, se a diferença entre a amostra atual e anterior for menor que o limiar estabelecido, considera-se que são amostras quivalentes (de mesmo valor), permitindo que o processamento pelo multiplicador possa ser ignorado, como pode ser vista na equação 4. Se a amostra seguinte continuar próxima a anterior, o multiplicador pode ser ignorado mais vezes, como mostra a equação 5.

Essa ideia pode ser implementada pela arquitetura apresentada na Figura 63. A principal mudança é a necessidade de mais registradores e um comparador (componente colorido), esse último compara as amostras e decide se deve executar uma multiplicação

[21] O multiplicador nesse contexto é o elemento que mais consome área de circuito e energia [40].

e acumulação ou apenas acumulação de coeficientes consecutivos.

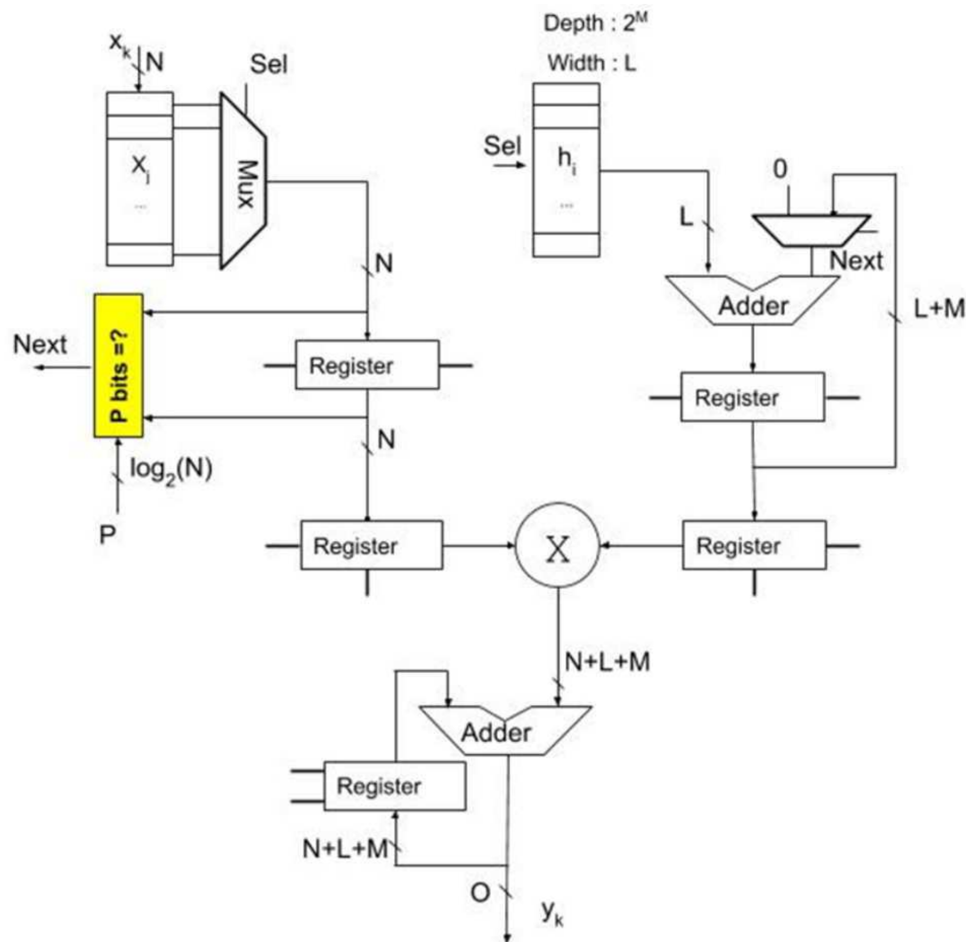

Figura 63: Arquitetura de Filtro Serial FIR AxC.

Esse Filtro foi implementado por Pedro Ochsendorf Portugal, um membro da equipe AMFoRS do Laboratório TIMA, em VHDL. Em seu relatório para o Laboratório ele apresenta os testes e os resultados obtidos, apresentando as melhorias em termos de consumo de energia, área de circuito e qualidade do sinal de saída. O trabalho exposto aqui, entretando, teve como objetivo re-escrever esse filtro em linguagem Chisel, encapsulá-lo com interface RoCC (veja Seção 6.2), e integra-lo no Rocket Chip. Isso permitirá que possa se fazer aplicações mais complexas com filtro, por meio de linguagem C, e o avalie em um contexto de aplicação, como processamento de imagens. Esse desenvolvimento será apresentado em seguida.

11 Coprocessador FIR AxC

Esse Seção apresentará todo o desenvolvimento feito para portar e integrar o filtro desenvolvido na Seção anterior como um coprocessador do Rocket Chip, assim como os resultados obtidos.

11.1 Interface de Filtro Serial FIR AxC

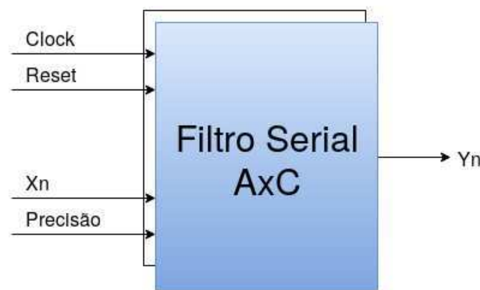

Figura 64: Interface do componente FIR Serial AxC.

A Figura 64 apresenta a interface da implementação do filtro discutido na Seção 10.2. O sinal Xn representa as amostras de um sinal de entrada qualquer, Yn a saída do cálculo e o sinal de $Precisao$ a configuração da quantidade de bits LSB a serem ignorados nos sinais de entrada.

A comparação das amostras consecutivas é feita de acordo com a arquitetura da Figura 65. Como pode ser observado, o sinal de precisão cria uma máscara para as entradas, isso permite escolher a quantidade de bits LSB a serem ignorados nesses sinais. A comparação, então, é feita com os sinais truncados, em que quando as amostras forem iguais, o circuito sinaliza que deve-se ser feita uma aproximação, como explicado anteriormente.

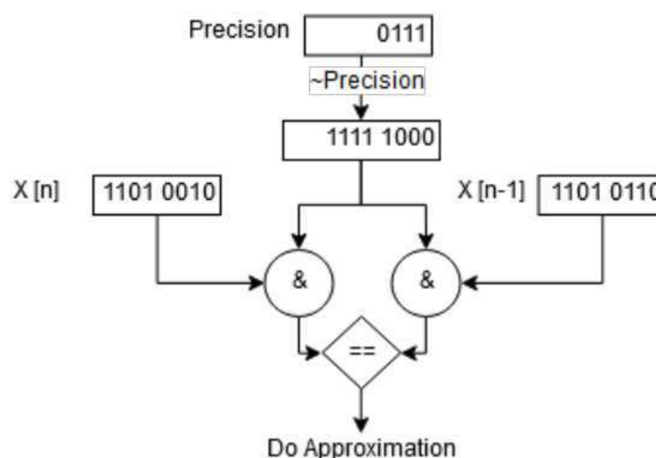

Figura 65: Arquitetura do circuito comparador do FIR Serial AxC.

11.2 ISA para Coprocessador FIR AxC

Com base no circuito apresentado na Seção precedente e com objetivo de encapsulá-lo com interface RoCC (veja Seção 6.2), para que se possa integrá-lo no Rocket Chip, criou-se uma especificação de ISA para provê meios suficientes de utilização do Filtro em linguagem de programação de assembly/C. A ISA é desenvolvida é apresentada na Tabela 34. É interessante notar que por ser um coprocessador, está sendo feita uma expansão da ISA RISC-V.

Instrução	**Pseudônimo**	**Descrição**	**Opcode**
fir.mov rs2, rs1	rs2 ← $rs1$	Move valor rs1 para reg. rs2	0b0000000
fir.fifo.put rs1	FIFO(put,rs1)	Coloca valor de rs1 em uma FIFO	0b0000001
fir.str rs2, rs1	Reg[rs2] → $Mem[rs1]$	Armazena dado Reg[rs2] no end. de rs1	0b0000010

Tabela 34: Extensão de ISA com instruções de Filtro FIR AxC.

A instrução fir.mov foi pensada para mover dados do programa para um banco de registradores do coprocessador, isso permite armazenamento de informações necessárias para o filtro, como coeficientes de filtros (h_n), valor de precisão e configurações do sistema (como inicializar de cálculo, etc). Em seguida, o fir.fifo.put, foi pensado para armazenar amostras dos sinais de entrada, ou seja, o *buffer* x_n, veja Figura 63. Por fim, a leitura dos resultados gerados pelo coprocessador podem ser lidos usando a instrução fir.str, que escreve no endereço de memória, fornecido pelo programador, o valor de algum registrador interno ao coprocessador.

11.3 Banco de Registradores

Projetou-se o banco de registradores apresentado na Tabela 35. Ou seja, considerou um filtro com capacidade de até 32 coeficientes, estabeleceu-se um registrador para configurar precisão do sistema, um registrador de status (que será visto em mais detalhes em seguida) e o registrador de resultado, que guarda valor de y_n. No reset, todos esses bits inicializam com zero.

Endereço de Reg.	Pseudônimo	Descrição	Tipo de Acesso
0x00	Coef[0]	Coeficiente FIR n=0	Leitura/Escrita
0x01	Coef[1]	Coeficiente FIR n=1	Leitura/Escrita
0x02	Coef[2]	Coeficiente FIR n=2	Leitura/Escrita
0x03	Coef[3]	Coeficiente FIR n=3	Leitura/Escrita
...	...	Coeficiente FIR n=...	Leitura/Escrita
0x1F	Coef[31]	Coeficiente FIR n=31	Leitura/Escrita
0x20	Precisão	Configure the precision to calculate	Leitura/Escrita
0x21	Status	Status do Coprocessador	Leitura/Escrita
0x22	Result	Resultado de cálculo FIR AxC	Leitura/Escrita

Tabela 35: Banco de Registrador do FIR AxC

11.3.1 Registrador de *Status*

Esse registrador, como pode ser observado na 36, foi projetado da seguinte forma: se o bit Enable FIR for 1, deve se realizar o cálculo do filtro considerando os valores dos coeficientes do filtro no banco de registradores e os valores do *buffer* x_n, quando o processamento terminar, o bit Done vai para 1 e o Enable FIR retorna a zero. O bit Enable Int. serve para habilitar sinal de interrupção do RoCC (causando excessão de valor 0x800000000000000C, que é um valor reservado na especificação da ISA RISC-V).

Reg. Status	[xLen-1]	[xLen-2:2]	[1]	[0]
Leitura/Escrita	Done	Reserved	Enable Int.	Enable FIR

Tabela 36: Registrador de Status

No reset, todos esses bits inicializam com zero. xLen é a variável definida em Chisel do Rocket Chip que configura tamanho do barramento do sistema, 32 ou 64 bits.

As Figuras seguintes, ilustram dois modelos de programação desse coprocessador. O primeiro sem uso de interrupção e o outro com interrupção, à partir da ISA desenvolvida, esse modelo de programa pode ser implementado.

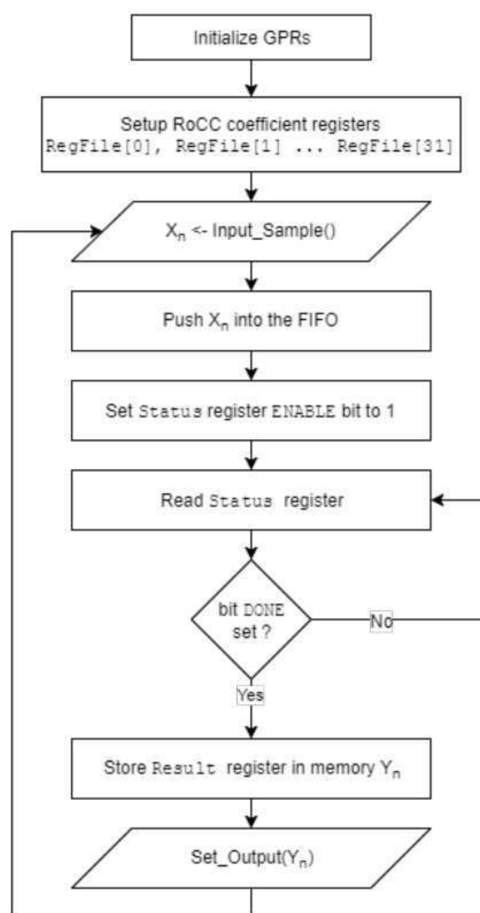

Figura 66: Modelo de programação para o RoCC FIR AxC, sem interrupção.

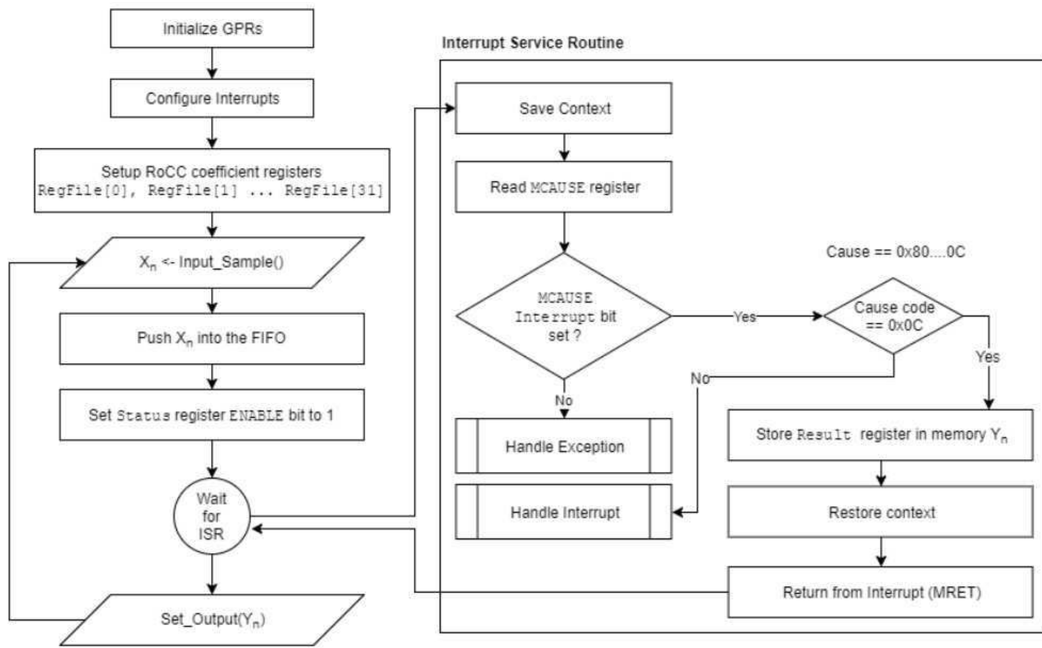

Figura 67: Modelo de programação para o RoCC FIR AxC, com interrupção.

O uso da interrupção, engloba configurações adicionais, como configuração das interrupções globais (no componente CLINT) e alteração no software para tratamente de interrupção, discutidos em seções anteriores. Devido a quantidade de detalhes, será aprentado na Seção de integração do coprocessador no Rocket Chip, apenas o código C que implementa o fluxo da Figura 66.

11.4 Arquitetura

Estabelecida as especificações e conhecendo a interface RoCC (Seção 6.2), foi desenvolvido a arquitetura apresentada na Figura 68, que implementa essa especificação.

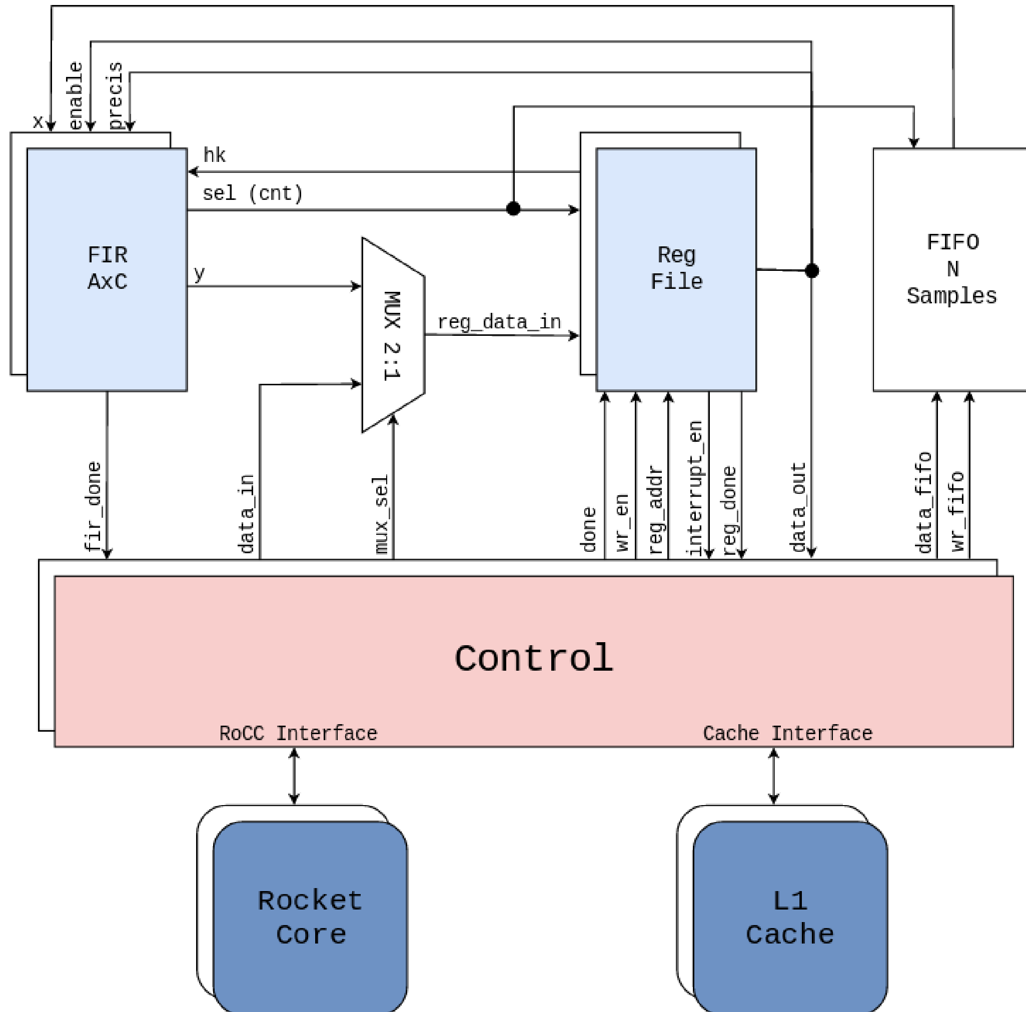

Figura 68: Coprocessor FIR AxC Architecture.

O módulo FIR AxC é o módulo apresentado na Seção 11.1, sua versão em VHDL, foi transcrita em Chisel com algumas alterações para se adequar a arquitetura criada. O módulo FIFO, recebe as amostras do sinal de entrada. O banco de registradores (Reg File), implementa a Tabela 35. O módulo de Controle será discutido na seção seguinte. Todos esses componentes foram desenvolvidos em Chisel, usando todo conhecimento adquirido na Seção 7. À partir daqui esse circuito será referenciado por RoCC FIR AxC.

11.4.1 Módulo de Controle

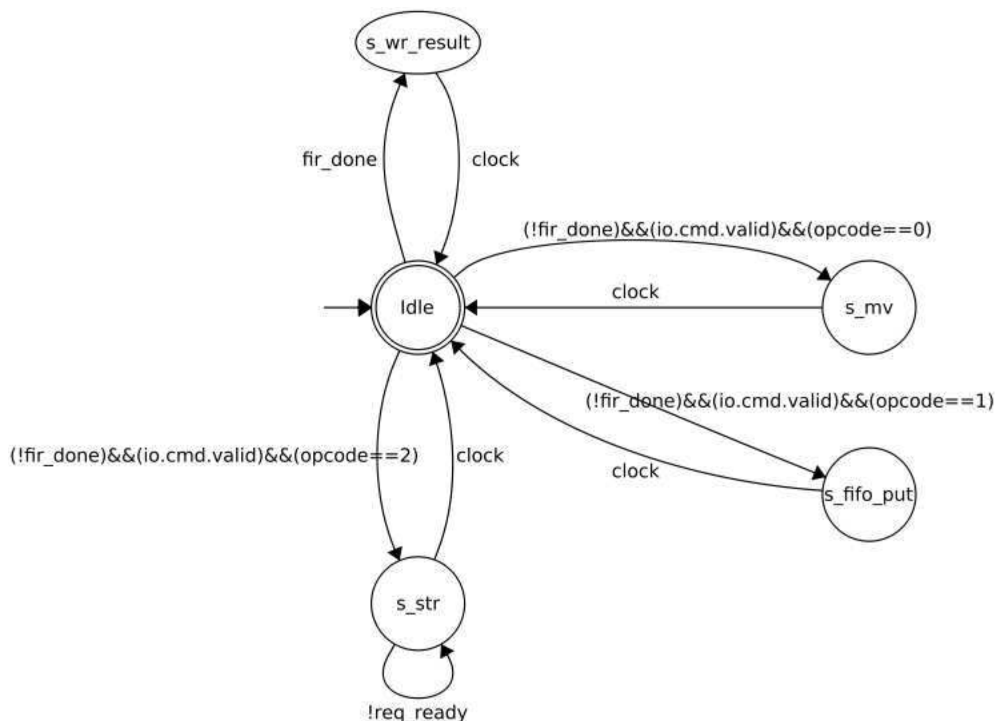

Figura 69: Máquina de estado implemtando interface RoCC.

Esse componente é a interface entre o processador e o sistema criado (coprocessador - RoCC), veja Figura 68. Os sinais de interfaces foram criados baseando-se nos sinais de controle do RoCC, definidos na Tabela 7 da Seção 6.2. Em resumo, quando o processador busca uma instrução do tipo coprocessador, a decodicação da mesma será repassada para o RoCC equivalente. Ao receber a instrução (alguma das apresentadas na Figura abaixo), a Máquina de Estados implementada no controle, Figura 69, executa a operação requisitada, ou seja, recebe a requisição e encaminha os dados para o *datapath*.

31 – 25	24 – 20	19 – 15	14	13	12	T – 6	6 – 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7

0000000	rs2	xd	xd	xs1	xs2	rd	0001011	fir.mov
0000001	rs2	xd	xd	xs1	xs2	rd	0001011	fir.fifo.put
0000010	rs2	xd	xd	xs1	xs2	rd	0001011	fir.store

Figura 70: Extensão de ISA RISC-V com Instruções do FIR AxC Desenvolvido.

Na Seção 6.2, é apresentado que o RoCC pode ter 4 opcodes e que cada opcode suporta 2^7 instruções (campo funct7, da Figura 70). No projeto em questão, fora usada apenas um opcode 0001011 e 3 instruções, como pode ser visto na Figura acima.

11.4.2 Testbench em SystemVerilog e Resultados

Usando as informações adquiridas na Seção 7, todos os componentes foram construídos em HDL Chisel. Porém, o teste do coprocessador não foi feito com essa linguagem, por não oferecer recursos suficientes para escrever bons testes [22]. Devido a isso, o projeto escrito em Chisel foi compilado para gerar os fontes Verilogs equivalentes, sendo eles usados como DUT (*Device Under Test*) para o testbench desenvolvido em SystemVerilog para o RoCC FIR AxC. O fonte do testbench pode ser visualizado no Anexo B e os fontes em Chisel, por pertencerem ao Laboratório TIMA, não serão apresentados nesse relatórios.

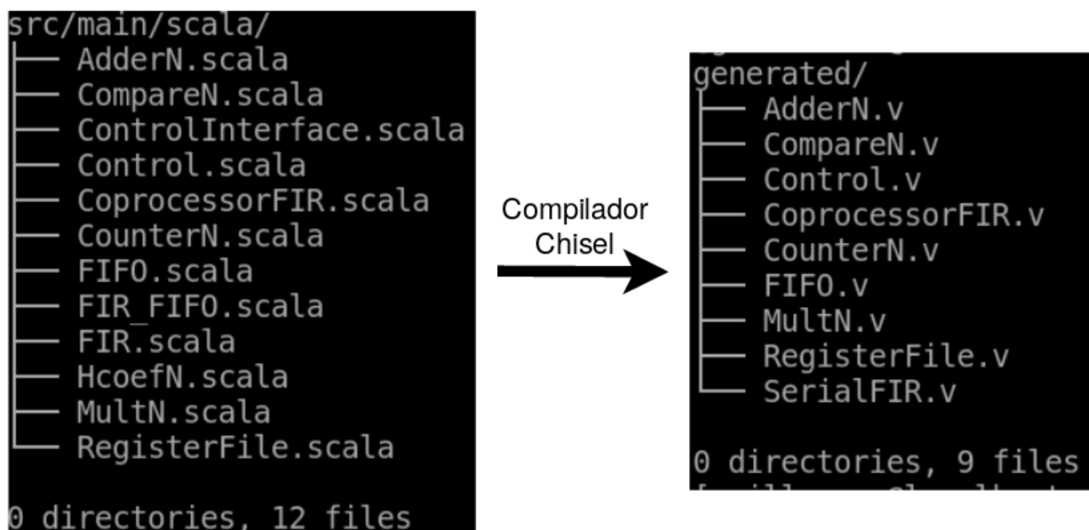

Figura 71: Fontes Chisel e Verilog gerarados.

O teste desenvolvido basicamente reproduz os modelo de programação ilustrados nas Figuras 66 e 67. O teste usou como base um Filtro Passa Baixa com 32 coeficientes e dois sinais (2000 amostras) para simular sinal analógico de entrada no sistema, sendo eles: 1) Onda quadrada e um 2) ECG (Eletrocardiograma). Além disso, o registrador de precisão foi sendo alterado para valiar a degradação do sinal.

A análise quantitativa do filtro isolado (somente o módulo do filtro sem as interfaces adicionais) foi feito por outro membro da equipe AMFoRS. Infelizmente, no contexto do Rocket Chip apenas a análise qualitativa do coprocessador fora feito, isto é, foi comparado as saídas do filtro original desenvolvido em VHDL com as saídas do RoCC FIR AxC.

Os coeficientes e sua resposta ao impulso são exibidos na sequência.

[22]O projeto Chisel apresenta-se bem desenvolvido para descrição de HW, porém, a ferramenta para desenvolvimento de testbenches ainda precisa ser bastante melhorada.

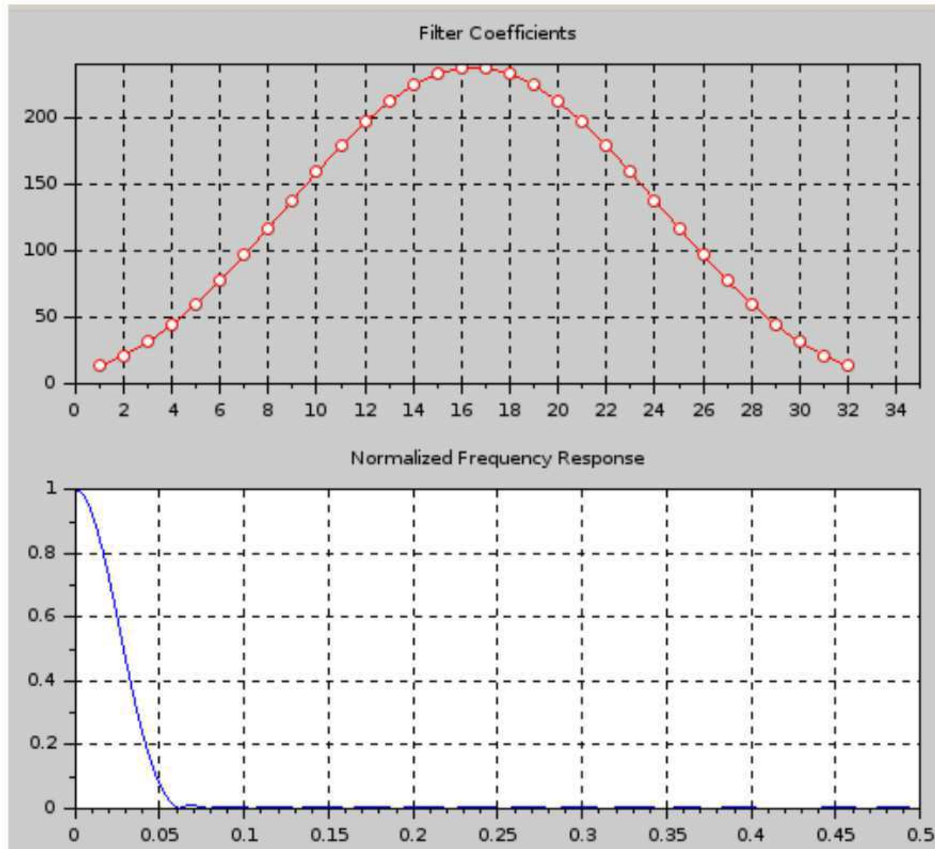

Figura 72: Coeficientes e Responta ao Impulso de Filtro Passa Baixas.

Coeficiente	Valor (em hexadec.)	Coeficiente	Valor (em hexadec.)
h_0	0x0D00	h_{16}	0xEE00
h_1	0x1500	h_{17}	0xE900
h_2	0x1F00	h_{18}	0xE100
h_3	0x2C00	h_{19}	0xD400
h_4	0x3C00	h_{20}	0xC500
h_5	0x4D00	h_{21}	0xB300
h_6	0x6100	h_{22}	0x9F00
h_7	0x7500	h_{23}	0x8A00
h_8	0x8A00	h_{24}	0x7500
h_9	0x9F00	h_{25}	0x6100
h_{10}	0xB300	h_{26}	0x4D00
h_{11}	0xC500	h_{27}	0x3C00
h_{12}	0xD400	h_{28}	0x2C00
h_{13}	0xE100	h_{29}	0x1F00
h_{14}	0xE900	h_{30}	0x1500
h_{15}	0xEE00	h_{31}	0x0D00

Tabela 37: Coeficientes de 8 bits do Filtro Passa Baixas.

A execução do teste foi feita usando a versão gratuita do ModelSim. O coprocessador foi construído em uma arquitetura de 64 bits, dessa forma, os coeficientes foram deslocados para esquerda até completar os 64 bits.

Na Figura 73 é mostrado a filtragem de uma onda quadrada. Para esse sinal, qualquer variação na precisão (registrador 33) degrada bastante o resultado esperado, que é a primeira componente senoidal do sinal de entrada. A mesma resposta foi conseguida no teste do filtro original.

Figura 73: RoCC FIR AxC executando filtragem de Onda Quadrada.

O segundo sinal, um ECG com ruído de alta frequência e de baixa amplitude, mostrou um resultado satisfatório analisando apenas os dois primeiros bits MSB das amostras, ou seja, truncando 62 bits LSB. A mesma resposta foi conseguida no teste do filtro original. Com isso, o coprocessador foi validado.

Figura 74: RoCC FIR AxC executando filtragem de ECG.

Infelizmente, uma análise quantitativa do ganho em consumo de energia e aumento do desempenho do coprocessador desenvolvido não foi possível de ser feita, pois o foco do presente estágio foi a integração desse sistema no Rocket Chip. Na seção seguinte os passos para de integração do RoCC no sistema é discutido.

11.5 Integração do RoCC no Rocket Chip

Após validação do coprocessador desenvolvido, pela análise qualitativa anteriormente discutida, os fontes Chisel desenvolvidos foram adicionados na plataforma Rocket Chip.

Como apresentado na Seção 6.2, o fonte src/main/scala/tile/LazyRoCC.scala permite instanciar coprocessadores no SoC. Então, com os fontes Chisel do projeto dentro do diretório tile/, bastou instanciar o módulo topo do RoCC no código LazyRoCC.scala e conectá-lo a interface do Rocket Core. O código abaixo ilustra esse passo:

```
class MyRoCCTestModule(outer: MyRoCCTest)(implicit p: Parameters)
extends LazyRoCCModule(outer)
  with HasCoreParameters {
  val coprocessorfir = Module(new CoprocessorFIR(64, 5, 35));
  coprocessorfir.io.rocc.cmd.inst.funct := io.cmd.bits.inst.funct
  coprocessorfir.io.rocc.cmd.inst.rs1 := io.cmd.bits.inst.rs1
  coprocessorfir.io.rocc.cmd.inst.rs2 := io.cmd.bits.inst.rs2
  coprocessorfir.io.rocc.cmd.inst.rd := io.cmd.bits.inst.rd
  coprocessorfir.io.rocc.cmd.rs1 := io.cmd.bits.rs1
  coprocessorfir.io.rocc.cmd.rs2 := io.cmd.bits.rs2
  io.cmd.ready := coprocessorfir.io.rocc.cmd.ready
  coprocessorfir.io.rocc.cmd.valid := io.cmd.valid
  io.resp.valid := coprocessorfir.io.rocc.resp.valid
  io.mem.req.bits.addr := coprocessorfir.io.rocc.mem.req.addr
  io.mem.req.bits.tag  := coprocessorfir.io.rocc.mem.req.tag
  io.mem.req.bits.cmd  := coprocessorfir.io.rocc.mem.req.cmd
  io.mem.req.bits.typ  := coprocessorfir.io.rocc.mem.req.typ
  io.mem.req.bits.data := coprocessorfir.io.rocc.mem.req.data
  io.mem.req.bits.phys := coprocessorfir.io.rocc.mem.req.phys
  io.mem.req.valid     := coprocessorfir.io.rocc.mem.req.valid
  coprocessorfir.io.rocc.mem.req.ready := io.mem.req.ready
  coprocessorfir.io.rocc.mem.resp := io.mem.resp.valid
  io.mem.invalidate_lr := coprocessorfir.io.rocc.mem.invalidate_lr
  io.busy := coprocessorfir.io.rocc.busy
  io.interrupt := coprocessorfir.io.rocc.interrupt
}
```

Código 62: Parte do código tile/LazyRoCC.scala para instanciar FIR RoCC AxC.

Após essa etapa, configurou-se o código src/main/subsystem/Configs.scala para tornar esse componente disponível no módulo topo do Rocket Chip. Perceba que o opcode usado foi o custom0, que foi discutido na Seção 6.2, esse opcode tem o mesmo valor usado na especificação da ISA feita anteriormente.

```
class WithRoccExample extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(rocc =
      Seq(
        RoCCParams(
          opcodes = OpcodeSet.custom0,
          generator = (p: Parameters) => {
            val myrocc = LazyModule(new MyRoCCTest()(p))
            myrocc
          })
        ))
    }
})
```

Código 63: Parte do código subsystem/Configs.scala para instanciar FIR RoCC AxC.

Finalmente, tornando o coprocessador disponível no módulo topo, o passo final para configuração do sistema é feito no código src/main/scala/system/Configs.scala fazendo:

```
class RoccExampleConfig extends Config(
        new WithRoccExample ++
        new DefaultConfig)
```

Código 64: Módulo topo instanciando RoCC FIR AxC no Rocket Chip.

Essa configuração RoccExampleConfig constroí um SoC com único processador (ISA RV64G) e coprocessador-RoCC. Feito esses ajustes nos códigos Scala, pode-se então gerar o modelo Verilog do sistema no diretório vsim/ e/ou construir o emulador, no diretório emulator/. Na próxima seção será apresentado como fora feita a emulação do sistema.

11.5.1 Construção de SW e Emulação do Sistema

Antes de gerar o emulador do sistema desenvolvido, foi preciso preparar a construção do SW a ser executado no SoC. A rigor, seria preciso modificar a *toolchain* RISC-V para adicionar o novo conjunto de instruções criado. Porém, uma solução alternativa é construir a instrução em C. . Essa forma de construir código torna-se mais produtiva, dado que não é preciso recompilar a toolchain quando quiser adicionar ou remover novas instruções. O código desenvolvido pode ser analisado no Anexo C.

Esse código implementa o modelo de programa apresentado na Figura 66 e ele pode ser compilado com a mesma infraestrutura desenvolvida na Seção 8.1.1. Com o binário construído, pode-se verificar com o comando "make mem_dump" (Makefile - Anexo A) a memória de instrução da máquina. Com o sucesso ela será igual à apresentada na Figura 75. Sendo os circulos vermelhos as novas instruções. Uma variável do tipo vetor guarda as amostras dos sinais, usou-se o mesmo sinal de ECG para o teste.

```
8a4:   00000393      li     t2,0
8a8:   0273328b      0x273328b
8ac:   fe543023      sd     t0,-32(s0)
8b0:   0c8000ef      jal    ra,80002978
8b4:   00100313      li     t1,1
8b8:   02100393      li     t2,33
8bc:   0073328b      0x73328b
8c0:   fe543023      sd     t0,-32(s0)
8c4:   0b4000ef      jal    ra,80002978
8c8:   ed840793      addi   a5,s0,-296
8cc:   00078313      mv     t1,a5
8d0:   02100393      li     t2,33
8d4:   0473328b      0x473328b
8d8:   fe543023      sd     t0,-32(s0)
8dc:   09c000ef      jal    ra,80002978
8e0:   01c0006f      j      800028fc <ma
8e4:   ed840793      addi   a5,s0,-296
8e8:   00078313      mv     t1,a5
8ec:   02100393      li     t2,33
8f0:   0473328b      0x473328b
8f4:   fe543023      sd     t0,-32(s0)
```

Figura 75: Instruções RoCC adicionadas ao binário gerado.

Com a infraestrutura para gerar os binários pronta, foi construído o emulador e foi gerado a emulação com geração de arquivo .vcd:

```
$ make debug CONFIG=RoccExampleConfig # gera emulado
$ ./emulador-...-RoccExempleConfig-debug -v rocc.vcd main
```

Código 65: Comando para gerar o emulador do SoC com RoCC implementado.

Com auxílio da ferramenta gtkwave, pode-se verificar que o RoCC FIR AxC fora embutido no Rocket Chip 76. Que a FIFO está recebendo os sinais de amostra, que os registradores estão recebendo os valores corretos, etc (veja Figuras 77 e 78). Na próxima seção seguinte abordamos os problemas enfrentados.

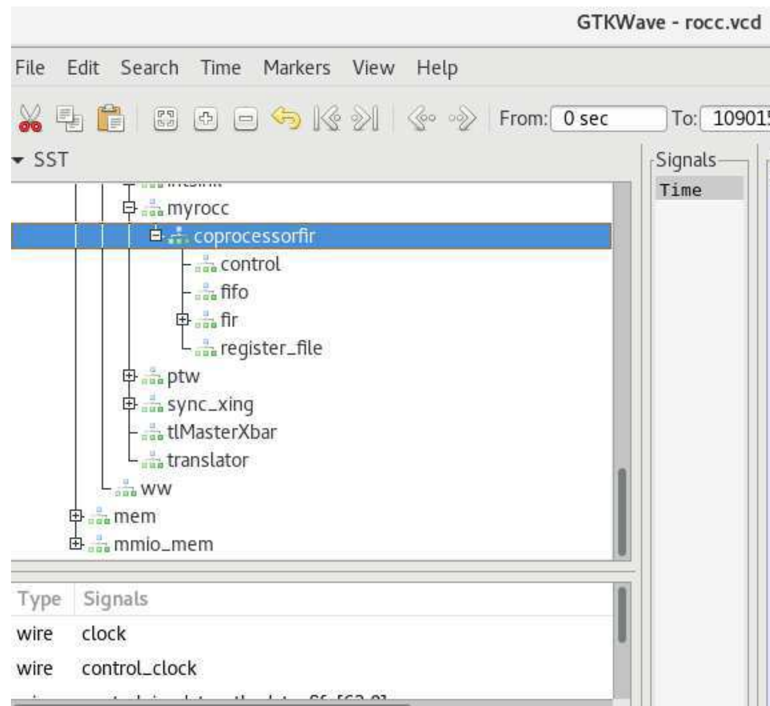

Figura 76: RoCC FIR AxC embutido no Rocket Chip.

11.5.2 Problemas Enfrentados e Conclusão

O integração do RoCC FIR AxC foi feita como pôde ser vista nas seções anteriores, porém, os resultados gerados pelo coprocessador (y_n) não foram satisfatórios, precisando de mais depuração para avaliar quais sinais precisão de correção. Isso aconteceu devido a incompatibilidade das linguagens Chisel usadas no Rocket Chip e da versão usada para desenvolvimento do RoCC FIR AxC.

Além disso, o sistema de emulação é bastante lento, precisando de horas, ou até mesmo um dia inteiro, para processar as 2000 amostras dos sinais, impossibilitando avaliar tempo de processamento, etc.

Não houve tempo para sintetizar a solução em FPGA, isso impossibilitou o levantamente de consumo de energia, testar o circuito em uma aplicação real, com amostras reais, etc.

Apesar desses problemas, um grande legado é deixado, possibilitando que os próximos estágiarios ou Engenheiros do Laboratório TIMA possam entender o *framework* Rocket Chip mais rapidamente e com isso, resolver os problemas enfrentados e avançar nos desafios restantes.

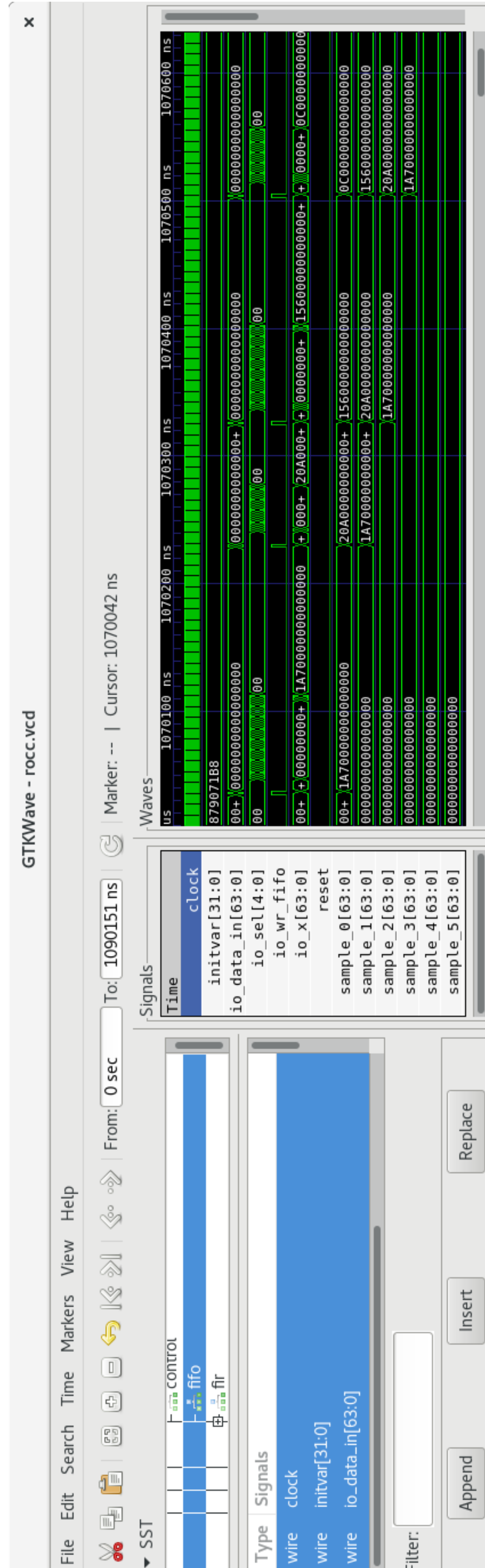

Figura 77: FIFO recebendo as amostras.

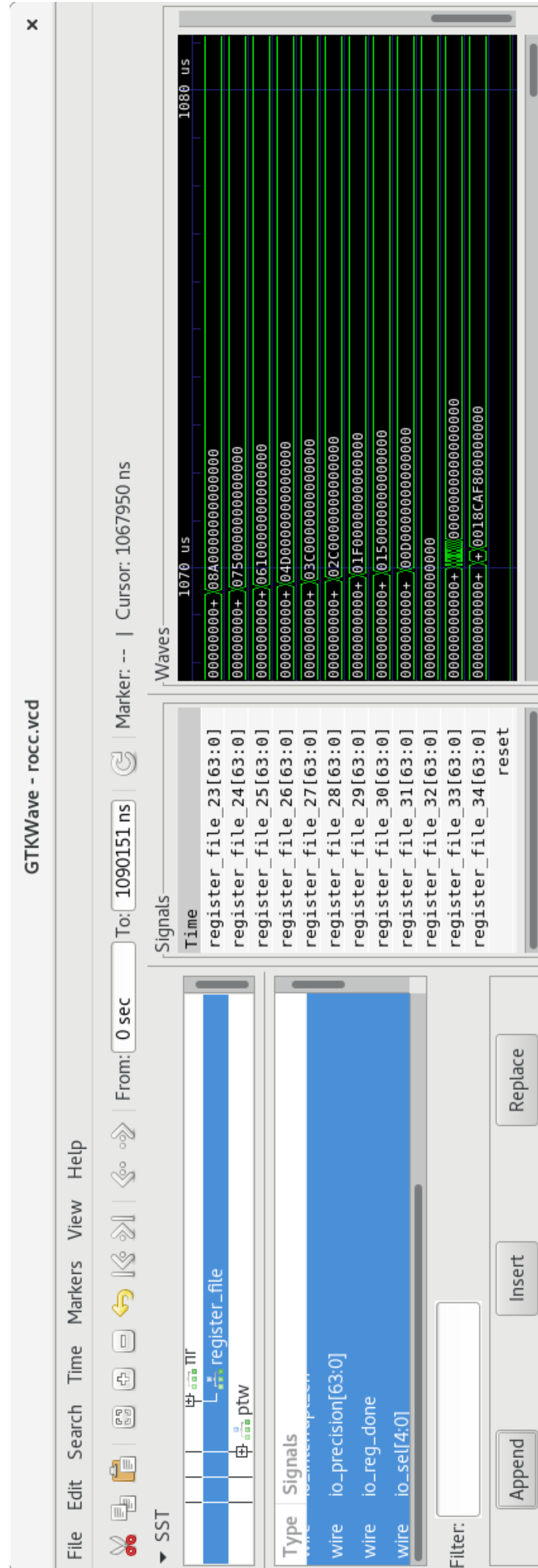

Figura 78: Banco de Registradores recebendo valores do SW.

12 Conclusão

A realização desse trabalho foi, sem dúvidas, um grande desafio e os aprendizados adquiridos foram inúmeros. Com o objetivo de escolher uma plataforma computacional para a pesquisa *Approximate Computing*, muitos projetos que envolvem concepção de SoCs foram avaliados, sendo escolhida a plataforma Rocket Chip. Essa plataforma é destaque nos projetos relacionados ao RISC-V e traz o estado-da-arte no que diz respeito a geração e prototipação rápida de SoCs.

O desafio de documentar um projeto como o Rocket Chip aprimora os conhecimentos e as habilidades exigidas no que está relacionado a *design* de computadores. Um sistema de construção de SoC complexo como esse, que permite a construção de sistemas multi-processados *Linux-Capable* ou até mesmo Microcontroladores para aplicações *Ultra-Low-Power*, com a maioria de seus componentes personalizáveis, como: FPU, MMU, coprocessadores, cache, barramentos e periféricos, exige que o Engenheiro conheça em um nível mais aprofundado arquiteturas computacionais e a relação entre hardware e software.

A necessidade de entender e documentar a *Hardware Description Language* Chisel usada para o projeto do Rocket Chip, foi outro desafio que, apesar das dificuldades, por ser algo totalmente novo e com pouquissíma documentação, foi vencido e isso mostra que toda a base adquirida ao longo do curso de Engenheria Elétrica, nos torna capazes de resolver os problemas que nos são dados.

Após a documentação do Rocket Chip e sua HDL, um estudo de caso fora realizado no tema da pesquisa *Approximate Computing*. Um coprocessador implementando um filtro FIR em paradigma AxC usou muitas das habilidades aprendidas ao longo do estágio, isto é, desenvolvimento de hardware com linguagem Chisel, criação e integração de coprocessadores no Rocket Chip, ajustes no processo de construção de software para teste e validação do projeto.

Isso mostra que o estágio foi bastante proveitoso e o legado deixado é que novos estudiosos, pesquisadores e estudantes usem esse material como forma de avançar nas pesquisas relacionadas a *design* de computadores, em especial a pesquisa *Approximate Computing* realizada no laboratório TIMA.

Referências

[1] Approximate Computing
https://gi.de/informatiklexikon/approximate-computing/.

[2] Y.-K. Chen et al., "Convergence of recognition, mining, synthesis workloads and its implications," Proc. IEEE, vol. 96, no. 5, pp. 790–807, May 2008.

[3] Qiang Xu, Todd Mytkowicz and Nam Sung Kim, "Approximate Computing: a survey", in IEEE Design & Test journal, Vol. 33, Issue 1, PP. 8-22, February 2016).

[4] "EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In Proc. International Conference on Programming Language Design and Implementation (PLDI)", A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. - 2011.

[5] "Verifying quantitative reliability for programs that execute on unreliable hardware.", M. Carbin, S. Misailovic, and M. C. Rinard. - 2013.

[6] S. Z. Gilani, N. S. Kim, and M. Schulte, "Scratchpad memory optimization for digital signal processing applications," in Proc. IEEE/ACM Design Autom. Test Eur., 2011, DOI: 10.1109/DATE.2011.5763158.

[7] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in Proc. Int. Conf. Architect. Support Programm. Lang. Oper. Syst., 2011, pp. 213–224.

[8] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in Proc. Int. Conf. Parallel Architect. Compilat. Tech., 2008, pp. 325–334.

[9] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in Proc. IEEE/ACM Design Autom. Conf., 2004, pp. 7–11.

[10] Jörg Henkel ; Santiago Pagani ; Heba Khdr ; Florian Kriebel ; Semeen Rehman ; Muhammad Shafique"Towards performance and reliability-efficient computing in the dark silicon era", in IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016 pp 1 - 6.

[11] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in Proc. IEEE/ACM Design Autom. Conf., 2013, DOI: 10.1145/2463209.2488873.

[12] S. Mittal, A Survey of Techniques for Approximate Computing, journal of ACM Computing Surveys (CSUR), Volume 48 Issue 4, May 2016.

[13] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2, Andrew Waterman, Krste Asanović, May 7, 2017.

[14] The Rocket-Chip Generator, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[15] The Rocket-Chip Generator - GitHub Repository https://github.com/freechipsproject/rocket-chip.

[16] BOOM project https://github.com/ucb-bar/riscv-boom.

[17] Z-Scale project https://github.com/ucb-bar/zscale.

[18] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2, Andrew Waterman, Krste Asanović, May 7, 2017.

[19] RISC-V, Calling Convention https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdfPDF Doc).

[20] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Privileged Architecture Version 1.10, Andrew Waterman, Krste Asanović, May 7, 2017.

[21] RISC-V, Spike and the Rocket Core Lab. http://www-inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab2-riscv.pdf.

[22] SiFive U5 Project Documentation https://static.dev.sifive.com/SiFive-U5-Coreplex-v1.0.pdf.

[23] SiFive E31 Coreplex Docuementaion https://static.dev.sifive.com/E31-Coreplex.pdf.

[24] RoCC Documentation https://docs.google.com/document/d/1CH2ep4YcL_ojsa3BVHEW-uwcKh1FlFTjH_kg5v8bxVw/edit.

[25] SiFive, TileLink Specification https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdfPDF Doc).

[26] AMBA AXI Reference Guide https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[27] Bootloader process of Rocket-Chip
`https://groups.google.com/a/groups.riscv.org/forum/m/#!topic/hw-dev/Pv8jUk0DzKI`.

[28] DTB - Bootloaders in Embedded Linux Systems link.

[29] RISC-V External Debug Support Version 0.13 link.

[30] UC Berkeley Paper, Chisel: Constructing Hardware in a Scala Embedded Language
`https://chisel.eecs.berkeley.edu/chisel-dac2012.pdf`.

[31] UC Berkeley, Chisel Official WebSite
`https://chisel.eecs.berkeley.edu/`.

[32] UC Berkeley, Specification for the FIRRTL Language
`https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html`.

[33] Chisel Manual, UC Berkeley
`https://chisel.eecs.berkeley.edu/2.2.0/chisel-manual.pdf`.

[34] Chisel Wiki
`https://github.com/freechipsproject/chisel3/wiki`.

[35] API Chisel3
`https://chisel.eecs.berkeley.edu/api/3.0.1/index.html`;

[36] Chisel2 Cheat Sheet, May 22, 2015
`https://chisel.eecs.berkeley.edu/2.2.0/chisel-cheatsheet.pdf`.

[37] Chisel3 Cheat Sheet, February 2018
`https://chisel.eecs.berkeley.edu/doc/chisel-cheatsheet3.pdf`.

[38] SpinalHDL Documentation
`https://spinalhdl.github.io/SpinalDoc/spinal/core/registers/`.

[39] Chisel 3 GitHub
`https://github.com/freechipsproject/chisel3#installation`.

[40] "Seeking Low-Power Synchronous/Asynchronous Systems: A FIR Implementation Case Study", Ali Skaf, Jean Simatic and Laurent Fesquet - 2017.

Anexo A: Códigos p/ compilação cruzada com RISC-V GNU Toolchain

```
# See LICENSE for license details.
#include "encoding.h"
#if __riscv_xlen == 64
# define LREG ld
# define SREG sd
# define REGBYTES 8
#else
# define LREG lw
# define SREG sw
# define REGBYTES 4
#endif
  .section ".text.init"
  .globl _start
_start:
  li  x1, 0
  li  x2, 0
  li  x3, 0
  li  x4, 0
  li  x5, 0
  li  x6, 0
  li  x7, 0
  li  x8, 0
  li  x9, 0
  li  x10,0
  li  x11,0
  li  x12,0
  li  x13,0
  li  x14,0
  li  x15,0
  li  x16,0
  li  x17,0
  li  x18,0
  li  x19,0
  li  x20,0
  li  x21,0
  li  x22,0
  li  x23,0
  li  x24,0
  li  x25,0
  li  x26,0
  li  x27,0
  li  x28,0
  li  x29,0
  li  x30,0
  li  x31,0
  # enable FPU and accelerator if present
  li t0, MSTATUS_FS | MSTATUS_XS
  csrs mstatus, t0
  # make sure XLEN agrees with compilation choice
  li t0, 1
  slli t0, t0, 31
#if __riscv_xlen == 64
  bgez t0, 1f
#else
  bltz t0, 1f
#endif
2:
  li a0, 1
  sw a0, tohost, t0
  j 2b
1:
#ifdef __riscv_flen
  # initialize FPU if we have one
  la t0, 1f
  csrw mtvec, t0

  fssr    x0
  fmv.s.x f0, x0
```

```
  fmv.s.x f1, x0
  fmv.s.x f2, x0
  fmv.s.x f3, x0
  fmv.s.x f4, x0
  fmv.s.x f5, x0
  fmv.s.x f6, x0
  fmv.s.x f7, x0
  fmv.s.x f8, x0
  fmv.s.x f9, x0
  fmv.s.x f10,x0
  fmv.s.x f11,x0
  fmv.s.x f12,x0
  fmv.s.x f13,x0
  fmv.s.x f14,x0
  fmv.s.x f15,x0
  fmv.s.x f16,x0
  fmv.s.x f17,x0
  fmv.s.x f18,x0
  fmv.s.x f19,x0
  fmv.s.x f20,x0
  fmv.s.x f21,x0
  fmv.s.x f22,x0
  fmv.s.x f23,x0
  fmv.s.x f24,x0
  fmv.s.x f25,x0
  fmv.s.x f26,x0
  fmv.s.x f27,x0
  fmv.s.x f28,x0
  fmv.s.x f29,x0
  fmv.s.x f30,x0
  fmv.s.x f31,x0
1:
#endif
  # initialize trap vector
  la t0, trap_entry
  csrw mtvec, t0
  # initialize global pointer
.option push
.option norelax
  la gp, __global_pointer$
.option pop
  la  tp, _end + 63
  and tp, tp, -64

  # get core id
  csrr a0, mhartid
  # for now, assume only 1 core
  li a1, 1
1:bgeu a0, a1, 1b

  # give each core 128KB of stack + TLS
#define STKSHIFT 17
  sll a2, a0, STKSHIFT
  add tp, tp, a2
  add sp, a0, 1
  sll sp, sp, STKSHIFT
  add sp, sp, tp

  j _init

  .align 2
trap_entry:
  addi sp, sp, -272

  SREG x1, 1*REGBYTES(sp)
  SREG x2, 2*REGBYTES(sp)
  SREG x3, 3*REGBYTES(sp)
  SREG x4, 4*REGBYTES(sp)
  SREG x5, 5*REGBYTES(sp)
  SREG x6, 6*REGBYTES(sp)
  SREG x7, 7*REGBYTES(sp)
  SREG x8, 8*REGBYTES(sp)
  SREG x9, 9*REGBYTES(sp)
  SREG x10, 10*REGBYTES(sp)
  SREG x11, 11*REGBYTES(sp)
  SREG x12, 12*REGBYTES(sp)
  SREG x13, 13*REGBYTES(sp)
  SREG x14, 14*REGBYTES(sp)
  SREG x15, 15*REGBYTES(sp)
  SREG x16, 16*REGBYTES(sp)
```

```
  SREG x17, 17*REGBYTES(sp)
  SREG x18, 18*REGBYTES(sp)
  SREG x19, 19*REGBYTES(sp)
  SREG x20, 20*REGBYTES(sp)
  SREG x21, 21*REGBYTES(sp)
  SREG x22, 22*REGBYTES(sp)
  SREG x23, 23*REGBYTES(sp)
  SREG x24, 24*REGBYTES(sp)
  SREG x25, 25*REGBYTES(sp)
  SREG x26, 26*REGBYTES(sp)
  SREG x27, 27*REGBYTES(sp)
  SREG x28, 28*REGBYTES(sp)
  SREG x29, 29*REGBYTES(sp)
  SREG x30, 30*REGBYTES(sp)
  SREG x31, 31*REGBYTES(sp)

  csrr a0, mcause
  csrr a1, mepc
  mv a2, sp
  jal handle_trap
  csrw mepc, a0

  # Remain in M-mode after eret
  li t0, MSTATUS_MPP
  csrs mstatus, t0

  LREG x1, 1*REGBYTES(sp)
  LREG x2, 2*REGBYTES(sp)
  LREG x3, 3*REGBYTES(sp)
  LREG x4, 4*REGBYTES(sp)
  LREG x5, 5*REGBYTES(sp)
  LREG x6, 6*REGBYTES(sp)
  LREG x7, 7*REGBYTES(sp)
  LREG x8, 8*REGBYTES(sp)
  LREG x9, 9*REGBYTES(sp)
  LREG x10, 10*REGBYTES(sp)
  LREG x11, 11*REGBYTES(sp)
  LREG x12, 12*REGBYTES(sp)
  LREG x13, 13*REGBYTES(sp)
  LREG x14, 14*REGBYTES(sp)
  LREG x15, 15*REGBYTES(sp)
  LREG x16, 16*REGBYTES(sp)
  LREG x17, 17*REGBYTES(sp)
  LREG x18, 18*REGBYTES(sp)
  LREG x19, 19*REGBYTES(sp)
  LREG x20, 20*REGBYTES(sp)
  LREG x21, 21*REGBYTES(sp)
  LREG x22, 22*REGBYTES(sp)
  LREG x23, 23*REGBYTES(sp)
  LREG x24, 24*REGBYTES(sp)
  LREG x25, 25*REGBYTES(sp)
  LREG x26, 26*REGBYTES(sp)
  LREG x27, 27*REGBYTES(sp)
  LREG x28, 28*REGBYTES(sp)
  LREG x29, 29*REGBYTES(sp)
  LREG x30, 30*REGBYTES(sp)
  LREG x31, 31*REGBYTES(sp)

  addi sp, sp, 272
  mret
.section ".tdata.begin"
.globl _tdata_begin
_tdata_begin:
.section ".tdata.end"
.globl _tdata_end
_tdata_end:
.section ".tbss.end"
.globl _tbss_end
_tbss_end:
.section ".tohost","aw",@progbits
.align 6
.globl tohost
tohost: .dword 0
.align 6
.globl fromhost
fromhost: .dword 0
```

Código 66: entry.S.

```
// See LICENSE for license details.

#include <stdint.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <limits.h>
#include <sys/signal.h>
#include "util.h"

#define SYS_write 64

#undef strcmp

extern volatile uint64_t tohost;
extern volatile uint64_t fromhost;

static uintptr_t syscall(uintptr_t which, uint64_t arg0, uint64_t arg1, uint64_t arg2)
{
  volatile uint64_t magic_mem[8] __attribute__((aligned(64)));
  magic_mem[0] = which;
  magic_mem[1] = arg0;
  magic_mem[2] = arg1;
  magic_mem[3] = arg2;
  __sync_synchronize();

  tohost = (uintptr_t)magic_mem;
  while (fromhost == 0)
    ;
  fromhost = 0;

  __sync_synchronize();
  return magic_mem[0];
}

#define NUM_COUNTERS 2
static uintptr_t counters[NUM_COUNTERS];
static char* counter_names[NUM_COUNTERS];

void setStats(int enable)
{
  int i = 0;
#define READ_CTR(name) do { \
    while (i >= NUM_COUNTERS) ; \
    uintptr_t csr = read_csr(name); \
    if (!enable) { csr -= counters[i]; counter_names[i] = #name; } \
    counters[i++] = csr; \
  } while (0)

  READ_CTR(mcycle);
  READ_CTR(minstret);

#undef READ_CTR
}

void __attribute__((noreturn)) tohost_exit(uintptr_t code)
{
  tohost = (code << 1) | 1;
  while (1);
}

uintptr_t __attribute__((weak)) handle_trap(uintptr_t cause, uintptr_t epc, uintptr_t regs[32])
{
  tohost_exit(1337);
}

void exit(int code)
{
  tohost_exit(code);
}

void abort()
{
  exit(128 + SIGABRT);
}

void printstr(const char* s)
{
  syscall(SYS_write, 1, (uintptr_t)s, strlen(s));
```

```
}

void __attribute__((weak)) thread_entry(int cid, int nc)
{
  // multi-threaded programs override this function.
  // for the case of single-threaded programs, only let core 0 proceed.
  while (cid != 0);
}

int __attribute__((weak)) main(int argc, char** argv)
{
  // single-threaded programs override this function.
  printstr("Implement main(), foo!\n");
  return -1;
}

static void init_tls()
{
  register void* thread_pointer asm("tp");
  extern char _tls_data;
  extern __thread char _tdata_begin, _tdata_end, _tbss_end;
  size_t tdata_size = &_tdata_end - &_tdata_begin;
  memcpy(thread_pointer, &_tls_data, tdata_size);
  size_t tbss_size = &_tbss_end - &_tdata_end;
  memset(thread_pointer + tdata_size, 0, tbss_size);
}
void _init(int cid, int nc)
{
  init_tls();
  thread_entry(cid, nc);

  // only single-threaded programs should ever get here.
  int ret = main(0, 0);

  char buf[NUM_COUNTERS * 32] __attribute__((aligned(64)));
  char* pbuf = buf;
  for (int i = 0; i < NUM_COUNTERS; i++)
    if (counters[i])
      pbuf += sprintf(pbuf, "%s = %d\n", counter_names[i], counters[i]);
  if (pbuf != buf)
    printstr(buf);

  exit(ret);
}

#undef putchar
int putchar(int ch)
{
  static __thread char buf[64] __attribute__((aligned(64)));
  static __thread int buflen = 0;

  buf[buflen++] = ch;

  if (ch == '\n' || buflen == sizeof(buf))
  {
    syscall(SYS_write, 1, (uintptr_t)buf, buflen);
    buflen = 0;
  }

  return 0;
}

void printhex(uint64_t x)
{
  char str[17];
  int i;
  for (i = 0; i < 16; i++)
  {
    str[15-i] = (x & 0xF) + ((x & 0xF) < 10 ? '0' : 'a'-10);
    x >>= 4;
  }
  str[16] = 0;

  printstr(str);
}

static inline void printnum(void (*putch)(int, void**), void **putdat,
                    unsigned long long num, unsigned base, int width, int padc)
{
  unsigned digs[sizeof(num)*CHAR_BIT];
```

```c
  int pos = 0;

  while (1)
  {
    digs[pos++] = num % base;
    if (num < base)
      break;
    num /= base;
  }

  while (width-- > pos)
    putch(padc, putdat);

  while (pos-- > 0)
    putch(digs[pos] + (digs[pos] >= 10 ? 'a' - 10 : '0'), putdat);
}

static unsigned long long getuint(va_list *ap, int lflag)
{
  if (lflag >= 2)
    return va_arg(*ap, unsigned long long);
  else if (lflag)
    return va_arg(*ap, unsigned long);
  else
    return va_arg(*ap, unsigned int);
}
static long long getint(va_list *ap, int lflag)
{
  if (lflag >= 2)
    return va_arg(*ap, long long);
  else if (lflag)
    return va_arg(*ap, long);
  else
    return va_arg(*ap, int);
}
static void vprintfmt(void (*putch)(int, void**), void **putdat, const char *fmt, va_list ap)
{
  register const char* p;
  const char* last_fmt;
  register int ch, err;
  unsigned long long num;
  int base, lflag, width, precision, altflag;
  char padc;

  while (1) {
    while ((ch = *(unsigned char *) fmt) != '%') {
      if (ch == '\0')
        return;
      fmt++;
      putch(ch, putdat);
    }
    fmt++;

    // Process a %-escape sequence
    last_fmt = fmt;
    padc = ' ';
    width = -1;
    precision = -1;
    lflag = 0;
    altflag = 0;
  reswitch:
    switch (ch = *(unsigned char *) fmt++) {

    // flag to pad on the right
    case '-':
      padc = '-';
      goto reswitch;

    // flag to pad with 0's instead of spaces
    case '0':
      padc = '0';
      goto reswitch;

    // width field
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
```

```
case '7':
case '8':
case '9':
  for (precision = 0; ; ++fmt) {
    precision = precision * 10 + ch - '0';
    ch = *fmt;
    if (ch < '0' || ch > '9')
      break;
  }
  goto process_precision;

case '*':
  precision = va_arg(ap, int);
  goto process_precision;

case '.':
  if (width < 0)
    width = 0;
  goto reswitch;

case '#':
  altflag = 1;
  goto reswitch;

process_precision:
  if (width < 0)
    width = precision, precision = -1;
  goto reswitch;

// long flag (doubled for long long)
case 'l':
  lflag++;
  goto reswitch;

// character
case 'c':
  putch(va_arg(ap, int), putdat);
  break;

// string
case 's':
  if ((p = va_arg(ap, char *)) == NULL)
    p = "(null)";
  if (width > 0 && padc != '-')
    for (width -= strnlen(p, precision); width > 0; width--)
      putch(padc, putdat);
  for (; (ch = *p) != '\0' && (precision < 0 || --precision >= 0); width--) {
    putch(ch, putdat);
    p++;
  }
  for (; width > 0; width--)
    putch(' ', putdat);
  break;

// (signed) decimal
case 'd':
  num = getint(&ap, lflag);
  if ((long long) num < 0) {
    putch('-', putdat);
    num = -(long long) num;
  }
  base = 10;
  goto signed_number;

// unsigned decimal
case 'u':
  base = 10;
  goto unsigned_number;

// (unsigned) octal
case 'o':
  // should do something with padding so it's always 3 octits
  base = 8;
  goto unsigned_number;

// pointer
case 'p':
  static_assert(sizeof(long) == sizeof(void*));
  lflag = 1;
  putch('0', putdat);
```

```
      putch('x', putdat);
      /* fall through to 'x' */

    // (unsigned) hexadecimal
    case 'x':
      base = 16;
    unsigned_number:
      num = getuint(&ap, lflag);
    signed_number:
      printnum(putch, putdat, num, base, width, padc);
      break;

    // escaped '%' character
    case '%':
      putch(ch, putdat);
      break;

    // unrecognized escape sequence - just print it literally
    default:
      putch('%', putdat);
      fmt = last_fmt;
      break;
    }
  }
}
int printf(const char* fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);

  vprintfmt((void*)putchar, 0, fmt, ap);

  va_end(ap);
  return 0; // incorrect return value, but who cares, anyway?
}
int sprintf(char* str, const char* fmt, ...)
{
  va_list ap;
  char* str0 = str;
  va_start(ap, fmt);

  void sprintf_putch(int ch, void** data)
  {
    char** pstr = (char**)data;
    **pstr = ch;
    (*pstr)++;
  }

  vprintfmt(sprintf_putch, (void**)&str, fmt, ap);
  *str = 0;

  va_end(ap);
  return str - str0;
}
void* memcpy(void* dest, const void* src, size_t len)
{
  if ((((uintptr_t)dest | (uintptr_t)src | len) & (sizeof(uintptr_t)-1)) == 0) {
    const uintptr_t* s = src;
    uintptr_t *d = dest;
    while (d < (uintptr_t*)(dest + len))
      *d++ = *s++;
  } else {
    const char* s = src;
    char *d = dest;
    while (d < (char*)(dest + len))
      *d++ = *s++;
  }
  return dest;
}
void* memset(void* dest, int byte, size_t len)
{
  if ((((uintptr_t)dest | len) & (sizeof(uintptr_t)-1)) == 0) {
    uintptr_t word = byte & 0xFF;
    word |= word << 8;
    word |= word << 16;
    word |= word << 16 << 16;

    uintptr_t *d = dest;
    while (d < (uintptr_t*)(dest + len))
      *d++ = word;
```

```c
  } else {
    char *d = dest;
    while (d < (char*)(dest + len))
      *d++ = byte;
  }
  return dest;
}
size_t strlen(const char *s)
{
  const char *p = s;
  while (*p)
    p++;
  return p - s;
}
size_t strnlen(const char *s, size_t n)
{
  const char *p = s;
  while (n-- && *p)
    p++;
  return p - s;
}
int strcmp(const char* s1, const char* s2)
{
  unsigned char c1, c2;

  do {
    c1 = *s1++;
    c2 = *s2++;
  } while (c1 != 0 && c1 == c2);

  return c1 - c2;
}

char* strcpy(char* dest, const char* src)
{
  char* d = dest;
  while ((*d++ = *src++))
    ;
  return dest;
}
long atol(const char* str)
{
  long res = 0;
  int sign = 0;

  while (*str == ' ')
    str++;

  if (*str == '-' || *str == '+') {
    sign = *str == '-';
    str++;
  }

  while (*str) {
    res *= 10;
    res += *str++ - '0';
  }

  return sign ? -res : res;
}
```

Código 67: syscalls.c.

```c
#include <stdio.h>

int main(void)
{
        printf("Hello World!\n");
        return 0;
}
```

Código 68: main.c.

```
OUTPUT_ARCH( "riscv" )
ENTRY(_start)
/* Sections */
SECTIONS
{
  /* text: test code section */
  . = 0x80000000;
  .text.init : { *(.text.init) }

  .tohost ALIGN(0x1000) : { *(.tohost) }

  .text : { *(.text) }

  /* data segment */
  .data ALIGN(0x40) : { *(.data) }

  .sdata : {
    __global_pointer$ = . + 0x800;
*(.srodata.cst16)*(.srodata.cst8)*(.srodata.cst4)*(.srodata.cst2)*(.srodata*)
    *(.sdata .sdata.* .gnu.linkonce.s.*)
  }
  /* bss segment */
  .sbss : {
    *(.sbss .sbss.* .gnu.linkonce.sb.*)
    *(.scommon)
  }
  .bss ALIGN(0x40) : { *(.bss) }
  /* thread-local data segment */
  .tdata :
  {
    _tls_data = .;
    *(.tdata.begin)
    *(.tdata)
    *(.tdata.end)
  }
  .tbss :
  {
    *(.tbss)
    *(.tbss.end)
  }

  /* End of uninitalized data segement */
  _end = .;
}
```

Código 69: link.ld.

```c
// See LICENSE for license details.
#ifndef __UTIL_H
#define __UTIL_H
//--------------------------------------------------------------------------
// Macros

// Set HOST_DEBUG to 1 if you are going to compile this for a host
// machine (ie Athena/Linux) for debug purposes and set HOST_DEBUG
// to 0 if you are compiling with the smips-gcc toolchain.
#ifndef HOST_DEBUG
#define HOST_DEBUG 0
#endif
// Set PREALLOCATE to 1 if you want to preallocate the benchmark
// function before starting stats. If you have instruction/data
// caches and you don't want to count the overhead of misses, then
// you will need to use preallocation.
#ifndef PREALLOCATE
#define PREALLOCATE 0
#endif
// Set SET_STATS to 1 if you want to carve out the piece that actually
// does the computation.
#if HOST_DEBUG
#include <stdio.h>
static void setStats(int enable) {}
#else
extern void setStats(int enable);
#endif
#include <stdint.h>
#define static_assert(cond) switch(0) { case 0: case !!(long)(cond): ; }
static void printArray(const char name[], int n, const int arr[])
{
#if HOST_DEBUG
  int i;
  printf( " %10s :", name );
  for ( i = 0; i < n; i++ )
    printf( " %3d ", arr[i] );
  printf( "\n" );
#endif
}
static void printDoubleArray(const char name[], int n, const double arr[])
{
#if HOST_DEBUG
  int i;
  printf( " %10s :", name );
  for ( i = 0; i < n; i++ )
    printf( " %g ", arr[i] );
  printf( "\n" );
#endif
}
static int verify(int n, const volatile int* test, const int* verify)
{
  int i;
  // Unrolled for faster verification
  for (i = 0; i < n/2*2; i+=2)
  {
    int t0 = test[i], t1 = test[i+1];
    int v0 = verify[i], v1 = verify[i+1];
    if (t0 != v0) return i+1;
    if (t1 != v1) return i+2;
  }
  if (n % 2 != 0 && test[n-1] != verify[n-1])
    return n;
  return 0;
}
static int verifyDouble(int n, const volatile double* test, const double* verify)
{
  int i;
  // Unrolled for faster verification
  for (i = 0; i < n/2*2; i+=2)
  {
    double t0 = test[i], t1 = test[i+1];
    double v0 = verify[i], v1 = verify[i+1];
    int eq1 = t0 == v0, eq2 = t1 == v1;
    if (!(eq1 & eq2)) return i+1+eq1;
  }
  if (n % 2 != 0 && test[n-1] != verify[n-1])
    return n;
  return 0;
}
```

```
static void __attribute__((noinline)) barrier(int ncores)
{
  static volatile int sense;
  static volatile int count;
  static __thread int threadsense;

  __sync_synchronize();

  threadsense = !threadsense;
  if (__sync_fetch_and_add(&count, 1) == ncores-1)
  {
    count = 0;
    sense = threadsense;
  }
  else while(sense != threadsense)
    ;

  __sync_synchronize();
}

static uint64_t lfsr(uint64_t x)
{
  uint64_t bit = (x ^ (x >> 1)) & 1;
  return (x >> 1) | (bit << 62);
}
#ifdef __riscv
#include "encoding.h"
#endif

#define stringify_1(s) #s
#define stringify(s) stringify_1(s)
#define stats(code, iter) do { \
    unsigned long _c = -read_csr(mcycle), _i = -read_csr(minstret); \
    code; \
    _c += read_csr(mcycle), _i += read_csr(minstret); \
    if (cid == 0) \
      printf("\n%s: %ld cycles, %ld.%ld cycles/iter, %ld.%ld CPI\n", \
             stringify(code), _c, _c/iter, 10*_c/iter%10, _c/_i, 10*_c/_i%10); \
  } while(0)
#endif //__UTIL_H
```

Código 70: util.h

```
// See LICENSE for license details.
#ifndef RISCV_CSR_ENCODING_H
#define RISCV_CSR_ENCODING_H
#define MSTATUS_UIE         0x00000001
#define MSTATUS_SIE         0x00000002
#define MSTATUS_HIE         0x00000004
#define MSTATUS_MIE         0x00000008
#define MSTATUS_UPIE        0x00000010
#define MSTATUS_SPIE        0x00000020
#define MSTATUS_HPIE        0x00000040
#define MSTATUS_MPIE        0x00000080
#define MSTATUS_SPP         0x00000100
#define MSTATUS_HPP         0x00000600
#define MSTATUS_MPP         0x00001800
#define MSTATUS_FS          0x00006000
#define MSTATUS_XS          0x00018000
#define MSTATUS_MPRV        0x00020000
#define MSTATUS_PUM         0x00040000
#define MSTATUS_MXR         0x00080000
#define MSTATUS_VM          0x1F000000
#define MSTATUS32_SD        0x80000000
#define MSTATUS64_SD        0x8000000000000000
#define SSTATUS_UIE         0x00000001
#define SSTATUS_SIE         0x00000002
#define SSTATUS_UPIE        0x00000010
#define SSTATUS_SPIE        0x00000020
#define SSTATUS_SPP         0x00000100
#define SSTATUS_FS          0x00006000
#define SSTATUS_XS          0x00018000
#define SSTATUS_PUM         0x00040000
#define SSTATUS32_SD        0x80000000
#define SSTATUS64_SD        0x8000000000000000
#define DCSR_XDEBUGVER      (3U<<30)
#define DCSR_NDRESET        (1<<29)
#define DCSR_FULLRESET      (1<<28)
```

```
#define DCSR_EBREAKM           (1<<15)
#define DCSR_EBREAKH           (1<<14)
#define DCSR_EBREAKS           (1<<13)
#define DCSR_EBREAKU           (1<<12)
#define DCSR_STOPCYCLE         (1<<10)
#define DCSR_STOPTIME          (1<<9)
#define DCSR_CAUSE             (7<<6)
#define DCSR_DEBUGINT          (1<<5)
#define DCSR_HALT              (1<<3)
#define DCSR_STEP              (1<<2)
#define DCSR_PRV               (3<<0)
#define DCSR_CAUSE_NONE        0
#define DCSR_CAUSE_SWBP        1
#define DCSR_CAUSE_HWBP        2
#define DCSR_CAUSE_DEBUGINT    3
#define DCSR_CAUSE_STEP        4
#define DCSR_CAUSE_HALT        5
#define MCONTROL_TYPE(xlen)       (0xfULL<<((xlen)-4))
#define MCONTROL_DMODE(xlen)      (1ULL<<((xlen)-5))
#define MCONTROL_MASKMAX(xlen)    (0x3fULL<<((xlen)-11))
#define MCONTROL_SELECT        (1<<19)
#define MCONTROL_TIMING        (1<<18)
#define MCONTROL_ACTION        (0x3f<<12)
#define MCONTROL_CHAIN         (1<<11)
#define MCONTROL_MATCH         (0xf<<7)
#define MCONTROL_M             (1<<6)
#define MCONTROL_H             (1<<5)
#define MCONTROL_S             (1<<4)
#define MCONTROL_U             (1<<3)
#define MCONTROL_EXECUTE       (1<<2)
#define MCONTROL_STORE         (1<<1)
#define MCONTROL_LOAD          (1<<0)
#define MCONTROL_TYPE_NONE         0
#define MCONTROL_TYPE_MATCH        2
#define MCONTROL_ACTION_DEBUG_EXCEPTION    0
#define MCONTROL_ACTION_DEBUG_MODE         1
#define MCONTROL_ACTION_TRACE_START        2
#define MCONTROL_ACTION_TRACE_STOP         3
#define MCONTROL_ACTION_TRACE_EMIT         4
#define MCONTROL_MATCH_EQUAL       0
#define MCONTROL_MATCH_NAPOT       1
#define MCONTROL_MATCH_GE          2
#define MCONTROL_MATCH_LT          3
#define MCONTROL_MATCH_MASK_LOW    4
#define MCONTROL_MATCH_MASK_HIGH   5
#define MIP_SSIP               (1 << IRQ_S_SOFT)
#define MIP_HSIP               (1 << IRQ_H_SOFT)
#define MIP_MSIP               (1 << IRQ_M_SOFT)
#define MIP_STIP               (1 << IRQ_S_TIMER)
#define MIP_HTIP               (1 << IRQ_H_TIMER)
#define MIP_MTIP               (1 << IRQ_M_TIMER)
#define MIP_SEIP               (1 << IRQ_S_EXT)
#define MIP_HEIP               (1 << IRQ_H_EXT)
#define MIP_MEIP               (1 << IRQ_M_EXT)
#define SIP_SSIP MIP_SSIP
#define SIP_STIP MIP_STIP
#define PRV_U 0
#define PRV_S 1
#define PRV_H 2
#define PRV_M 3
#define VM_MBARE 0
#define VM_MBB   1
#define VM_MBBID 2
#define VM_SV32  8
#define VM_SV39  9
#define VM_SV48  10
#define IRQ_S_SOFT    1
#define IRQ_H_SOFT    2
#define IRQ_M_SOFT    3
#define IRQ_S_TIMER   5
#define IRQ_H_TIMER   6
#define IRQ_M_TIMER   7
#define IRQ_S_EXT     9
#define IRQ_H_EXT     10
#define IRQ_M_EXT     11
#define IRQ_COP       12
#define IRQ_HOST      13
#define DEFAULT_RSTVEC     0x00001000
#define DEFAULT_NMIVEC     0x00001004
#define DEFAULT_MTVEC      0x00001010
```

```
#define CONFIG_STRING_ADDR 0x0000100C
#define EXT_IO_BASE          0x40000000
#define DRAM_BASE            0x80000000
// page table entry (PTE) fields
#define PTE_V       0x001 // Valid
#define PTE_R       0x002 // Read
#define PTE_W       0x004 // Write
#define PTE_X       0x008 // Execute
#define PTE_U       0x010 // User
#define PTE_G       0x020 // Global
#define PTE_A       0x040 // Accessed
#define PTE_D       0x080 // Dirty
#define PTE_SOFT  0x300 // Reserved for Software
#define PTE_PPN_SHIFT 10
#define PTE_TABLE(PTE) (((PTE) & (PTE_V | PTE_R | PTE_W | PTE_X)) == PTE_V)
#ifdef __riscv
#ifdef __riscv64
# define MSTATUS_SD MSTATUS64_SD
# define SSTATUS_SD SSTATUS64_SD
# define RISCV_PGLEVEL_BITS 9
#else
# define MSTATUS_SD MSTATUS32_SD
# define SSTATUS_SD SSTATUS32_SD
# define RISCV_PGLEVEL_BITS 10
#endif
#define RISCV_PGSHIFT 12
#define RISCV_PGSIZE (1 << RISCV_PGSHIFT)
#ifndef __ASSEMBLER__
#ifdef __GNUC__
#define read_csr(reg) ({ unsigned long __tmp; \
  asm volatile ("csrr %0, " #reg : "=r"(__tmp)); \
  __tmp; })
#define write_csr(reg, val) ({ \
  if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
    asm volatile ("csrw " #reg ", %0" :: "i"(val)); \
  else \
    asm volatile ("csrw " #reg ", %0" :: "r"(val)); })
#define swap_csr(reg, val) ({ unsigned long __tmp; \
  if (__builtin_constant_p(val) && (unsigned long)(val) < 32) \
    asm volatile ("csrrw %0, " #reg ", %1" : "=r"(__tmp) : "i"(val)); \
  else \
    asm volatile ("csrrw %0, " #reg ", %1" : "=r"(__tmp) : "r"(val)); \
  __tmp; })
#define set_csr(reg, bit) ({ unsigned long __tmp; \
  if (__builtin_constant_p(bit) && (unsigned long)(bit) < 32) \
    asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "i"(bit)); \
  else \
    asm volatile ("csrrs %0, " #reg ", %1" : "=r"(__tmp) : "r"(bit)); \
  __tmp; })
#define clear_csr(reg, bit) ({ unsigned long __tmp; \
  if (__builtin_constant_p(bit) && (unsigned long)(bit) < 32) \
    asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "i"(bit)); \
  else \
    asm volatile ("csrrc %0, " #reg ", %1" : "=r"(__tmp) : "r"(bit)); \
  __tmp; })
#define rdtime() read_csr(time)
#define rdcycle() read_csr(cycle)
#define rdinstret() read_csr(instret)
#endif
#endif
#endif
#endif
/* Automatically generated by parse-opcodes */
#ifndef RISCV_ENCODING_H
#define RISCV_ENCODING_H
#define MATCH_BEQ 0x63
#define MASK_BEQ  0x707f
#define MATCH_BNE 0x1063
#define MASK_BNE  0x707f
#define MATCH_BLT 0x4063
#define MASK_BLT  0x707f
#define MATCH_BGE 0x5063
#define MASK_BGE  0x707f
#define MATCH_BLTU 0x6063
#define MASK_BLTU  0x707f
#define MATCH_BGEU 0x7063
#define MASK_BGEU  0x707f
#define MATCH_JALR 0x67
#define MASK_JALR  0x707f
#define MATCH_JAL 0x6f
```

```
#define MASK_JAL  0x7f
#define MATCH_LUI 0x37
#define MASK_LUI  0x7f
#define MATCH_AUIPC 0x17
#define MASK_AUIPC  0x7f
#define MATCH_ADDI 0x13
#define MASK_ADDI  0x707f
#define MATCH_SLLI 0x1013
#define MASK_SLLI  0xfc00707f
#define MATCH_SLTI 0x2013
#define MASK_SLTI  0x707f
#define MATCH_SLTIU 0x3013
#define MASK_SLTIU  0x707f
#define MATCH_XORI 0x4013
#define MASK_XORI  0x707f
#define MATCH_SRLI 0x5013
#define MASK_SRLI  0xfc00707f
#define MATCH_SRAI 0x40005013
#define MASK_SRAI  0xfc00707f
#define MATCH_ORI 0x6013
#define MASK_ORI  0x707f
#define MATCH_ANDI 0x7013
#define MASK_ANDI  0x707f
#define MATCH_ADD 0x33
#define MASK_ADD  0xfe00707f
#define MATCH_SUB 0x40000033
#define MASK_SUB  0xfe00707f
#define MATCH_SLL 0x1033
#define MASK_SLL  0xfe00707f
#define MATCH_SLT 0x2033
#define MASK_SLT  0xfe00707f
#define MATCH_SLTU 0x3033
#define MASK_SLTU  0xfe00707f
#define MATCH_XOR 0x4033
#define MASK_XOR  0xfe00707f
#define MATCH_SRL 0x5033
#define MASK_SRL  0xfe00707f
#define MATCH_SRA 0x40005033
#define MASK_SRA  0xfe00707f
#define MATCH_OR 0x6033
#define MASK_OR  0xfe00707f
#define MATCH_AND 0x7033
#define MASK_AND  0xfe00707f
#define MATCH_ADDIW 0x1b
#define MASK_ADDIW  0x707f
#define MATCH_SLLIW 0x101b
#define MASK_SLLIW  0xfe00707f
#define MATCH_SRLIW 0x501b
#define MASK_SRLIW  0xfe00707f
#define MATCH_SRAIW 0x4000501b
#define MASK_SRAIW  0xfe00707f
#define MATCH_ADDW 0x3b
#define MASK_ADDW  0xfe00707f
#define MATCH_SUBW 0x4000003b
#define MASK_SUBW  0xfe00707f
#define MATCH_SLLW 0x103b
#define MASK_SLLW  0xfe00707f
#define MATCH_SRLW 0x503b
#define MASK_SRLW  0xfe00707f
#define MATCH_SRAW 0x4000503b
#define MASK_SRAW  0xfe00707f
#define MATCH_LB 0x3
#define MASK_LB  0x707f
#define MATCH_LH 0x1003
#define MASK_LH  0x707f
#define MATCH_LW 0x2003
#define MASK_LW  0x707f
#define MATCH_LD 0x3003
#define MASK_LD  0x707f
#define MATCH_LBU 0x4003
#define MASK_LBU  0x707f
#define MATCH_LHU 0x5003
#define MASK_LHU  0x707f
#define MATCH_LWU 0x6003
#define MASK_LWU  0x707f
#define MATCH_SB 0x23
#define MASK_SB  0x707f
#define MATCH_SH 0x1023
#define MASK_SH  0x707f
#define MATCH_SW 0x2023
```

```
#define MASK_SW  0x707f
#define MATCH_SD 0x3023
#define MASK_SD  0x707f
#define MATCH_FENCE 0xf
#define MASK_FENCE  0x707f
#define MATCH_FENCE_I 0x100f
#define MASK_FENCE_I  0x707f
#define MATCH_MUL 0x2000033
#define MASK_MUL  0xfe00707f
#define MATCH_MULH 0x2001033
#define MASK_MULH  0xfe00707f
#define MATCH_MULHSU 0x2002033
#define MASK_MULHSU  0xfe00707f
#define MATCH_MULHU 0x2003033
#define MASK_MULHU  0xfe00707f
#define MATCH_DIV 0x2004033
#define MASK_DIV  0xfe00707f
#define MATCH_DIVU 0x2005033
#define MASK_DIVU  0xfe00707f
#define MATCH_REM 0x2006033
#define MASK_REM  0xfe00707f
#define MATCH_REMU 0x2007033
#define MASK_REMU  0xfe00707f
#define MATCH_MULW 0x200003b
#define MASK_MULW  0xfe00707f
#define MATCH_DIVW 0x200403b
#define MASK_DIVW  0xfe00707f
#define MATCH_DIVUW 0x200503b
#define MASK_DIVUW  0xfe00707f
#define MATCH_REMW 0x200603b
#define MASK_REMW  0xfe00707f
#define MATCH_REMUW 0x200703b
#define MASK_REMUW  0xfe00707f
#define MATCH_AMOADD_W 0x202f
#define MASK_AMOADD_W  0xf800707f
#define MATCH_AMOXOR_W 0x2000202f
#define MASK_AMOXOR_W  0xf800707f
#define MATCH_AMOOR_W 0x4000202f
#define MASK_AMOOR_W  0xf800707f
#define MATCH_AMOAND_W 0x6000202f
#define MASK_AMOAND_W  0xf800707f
#define MATCH_AMOMIN_W 0x8000202f
#define MASK_AMOMIN_W  0xf800707f
#define MATCH_AMOMAX_W 0xa000202f
#define MASK_AMOMAX_W  0xf800707f
#define MATCH_AMOMINU_W 0xc000202f
#define MASK_AMOMINU_W  0xf800707f
#define MATCH_AMOMAXU_W 0xe000202f
#define MASK_AMOMAXU_W  0xf800707f
#define MATCH_AMOSWAP_W 0x800202f
#define MASK_AMOSWAP_W  0xf800707f
#define MATCH_LR_W 0x1000202f
#define MASK_LR_W  0xf9f0707f
#define MATCH_SC_W 0x1800202f
#define MASK_SC_W  0xf800707f
#define MATCH_AMOADD_D 0x302f
#define MASK_AMOADD_D  0xf800707f
#define MATCH_AMOXOR_D 0x2000302f
#define MASK_AMOXOR_D  0xf800707f
#define MATCH_AMOOR_D 0x4000302f
#define MASK_AMOOR_D  0xf800707f
#define MATCH_AMOAND_D 0x6000302f
#define MASK_AMOAND_D  0xf800707f
#define MATCH_AMOMIN_D 0x8000302f
#define MASK_AMOMIN_D  0xf800707f
#define MATCH_AMOMAX_D 0xa000302f
#define MASK_AMOMAX_D  0xf800707f
#define MATCH_AMOMINU_D 0xc000302f
#define MASK_AMOMINU_D  0xf800707f
#define MATCH_AMOMAXU_D 0xe000302f
#define MASK_AMOMAXU_D  0xf800707f
#define MATCH_AMOSWAP_D 0x800302f
#define MASK_AMOSWAP_D  0xf800707f
#define MATCH_LR_D 0x1000302f
#define MASK_LR_D  0xf9f0707f
#define MATCH_SC_D 0x1800302f
#define MASK_SC_D  0xf800707f
#define MATCH_ECALL 0x73
#define MASK_ECALL  0xffffffff
#define MATCH_EBREAK 0x100073
```

```
#define MASK_EBREAK  0xffffffff
#define MATCH_URET 0x200073
#define MASK_URET  0xffffffff
#define MATCH_SRET 0x10200073
#define MASK_SRET  0xffffffff
#define MATCH_HRET 0x20200073
#define MASK_HRET  0xffffffff
#define MATCH_MRET 0x30200073
#define MASK_MRET  0xffffffff
#define MATCH_DRET 0x7b200073
#define MASK_DRET  0xffffffff
#define MATCH_SFENCE_VM 0x10400073
#define MASK_SFENCE_VM  0xfff07fff
#define MATCH_WFI 0x10500073
#define MASK_WFI  0xffffffff
#define MATCH_CSRRW 0x1073
#define MASK_CSRRW  0x707f
#define MATCH_CSRRS 0x2073
#define MASK_CSRRS  0x707f
#define MATCH_CSRRC 0x3073
#define MASK_CSRRC  0x707f
#define MATCH_CSRRWI 0x5073
#define MASK_CSRRWI  0x707f
#define MATCH_CSRRSI 0x6073
#define MASK_CSRRSI  0x707f
#define MATCH_CSRRCI 0x7073
#define MASK_CSRRCI  0x707f
#define MATCH_FADD_S 0x53
#define MASK_FADD_S  0xfe00007f
#define MATCH_FSUB_S 0x8000053
#define MASK_FSUB_S  0xfe00007f
#define MATCH_FMUL_S 0x10000053
#define MASK_FMUL_S  0xfe00007f
#define MATCH_FDIV_S 0x18000053
#define MASK_FDIV_S  0xfe00007f
#define MATCH_FSGNJ_S 0x20000053
#define MASK_FSGNJ_S  0xfe00707f
#define MATCH_FSGNJN_S 0x20001053
#define MASK_FSGNJN_S  0xfe00707f
#define MATCH_FSGNJX_S 0x20002053
#define MASK_FSGNJX_S  0xfe00707f
#define MATCH_FMIN_S 0x28000053
#define MASK_FMIN_S  0xfe00707f
#define MATCH_FMAX_S 0x28001053
#define MASK_FMAX_S  0xfe00707f
#define MATCH_FSQRT_S 0x58000053
#define MASK_FSQRT_S  0xfff0007f
#define MATCH_FADD_D 0x2000053
#define MASK_FADD_D  0xfe00007f
#define MATCH_FSUB_D 0xa000053
#define MASK_FSUB_D  0xfe00007f
#define MATCH_FMUL_D 0x12000053
#define MASK_FMUL_D  0xfe00007f
#define MATCH_FDIV_D 0x1a000053
#define MASK_FDIV_D  0xfe00007f
#define MATCH_FSGNJ_D 0x22000053
#define MASK_FSGNJ_D  0xfe00707f
#define MATCH_FSGNJN_D 0x22001053
#define MASK_FSGNJN_D  0xfe00707f
#define MATCH_FSGNJX_D 0x22002053
#define MASK_FSGNJX_D  0xfe00707f
#define MATCH_FMIN_D 0x2a000053
#define MASK_FMIN_D  0xfe00707f
#define MATCH_FMAX_D 0x2a001053
#define MASK_FMAX_D  0xfe00707f
#define MATCH_FCVT_S_D 0x40100053
#define MASK_FCVT_S_D  0xfff0007f
#define MATCH_FCVT_D_S 0x42000053
#define MASK_FCVT_D_S  0xfff0007f
#define MATCH_FSQRT_D 0x5a000053
#define MASK_FSQRT_D  0xfff0007f
#define MATCH_FLE_S 0xa0000053
#define MASK_FLE_S  0xfe00707f
#define MATCH_FLT_S 0xa0001053
#define MASK_FLT_S  0xfe00707f
#define MATCH_FEQ_S 0xa0002053
#define MASK_FEQ_S  0xfe00707f
#define MATCH_FLE_D 0xa2000053
#define MASK_FLE_D  0xfe00707f
#define MATCH_FLT_D 0xa2001053
```

```
#define MASK_FLT_D  0xfe00707f
#define MATCH_FEQ_D 0xa2002053
#define MASK_FEQ_D  0xfe00707f
#define MATCH_FCVT_W_S 0xc0000053
#define MASK_FCVT_W_S  0xfff0007f
#define MATCH_FCVT_WU_S 0xc0100053
#define MASK_FCVT_WU_S  0xfff0007f
#define MATCH_FCVT_L_S 0xc0200053
#define MASK_FCVT_L_S  0xfff0007f
#define MATCH_FCVT_LU_S 0xc0300053
#define MASK_FCVT_LU_S  0xfff0007f
#define MATCH_FMV_X_S 0xe0000053
#define MASK_FMV_X_S  0xfff0707f
#define MATCH_FCLASS_S 0xe0001053
#define MASK_FCLASS_S  0xfff0707f
#define MATCH_FCVT_W_D 0xc2000053
#define MASK_FCVT_W_D  0xfff0007f
#define MATCH_FCVT_WU_D 0xc2100053
#define MASK_FCVT_WU_D  0xfff0007f
#define MATCH_FCVT_L_D 0xc2200053
#define MASK_FCVT_L_D  0xfff0007f
#define MATCH_FCVT_LU_D 0xc2300053
#define MASK_FCVT_LU_D  0xfff0007f
#define MATCH_FMV_X_D 0xe2000053
#define MASK_FMV_X_D  0xfff0707f
#define MATCH_FCLASS_D 0xe2001053
#define MASK_FCLASS_D  0xfff0707f
#define MATCH_FCVT_S_W 0xd0000053
#define MASK_FCVT_S_W  0xfff0007f
#define MATCH_FCVT_S_WU 0xd0100053
#define MASK_FCVT_S_WU  0xfff0007f
#define MATCH_FCVT_S_L 0xd0200053
#define MASK_FCVT_S_L  0xfff0007f
#define MATCH_FCVT_S_LU 0xd0300053
#define MASK_FCVT_S_LU  0xfff0007f
#define MATCH_FMV_S_X 0xf0000053
#define MASK_FMV_S_X  0xfff0707f
#define MATCH_FCVT_D_W 0xd2000053
#define MASK_FCVT_D_W  0xfff0007f
#define MATCH_FCVT_D_WU 0xd2100053
#define MASK_FCVT_D_WU  0xfff0007f
#define MATCH_FCVT_D_L 0xd2200053
#define MASK_FCVT_D_L  0xfff0007f
#define MATCH_FCVT_D_LU 0xd2300053
#define MASK_FCVT_D_LU  0xfff0007f
#define MATCH_FMV_D_X 0xf2000053
#define MASK_FMV_D_X  0xfff0707f
#define MATCH_FLW 0x2007
#define MASK_FLW  0x707f
#define MATCH_FLD 0x3007
#define MASK_FLD  0x707f
#define MATCH_FSW 0x2027
#define MASK_FSW  0x707f
#define MATCH_FSD 0x3027
#define MASK_FSD  0x707f
#define MATCH_FMADD_S 0x43
#define MASK_FMADD_S  0x600007f
#define MATCH_FMSUB_S 0x47
#define MASK_FMSUB_S  0x600007f
#define MATCH_FNMSUB_S 0x4b
#define MASK_FNMSUB_S  0x600007f
#define MATCH_FNMADD_S 0x4f
#define MASK_FNMADD_S  0x600007f
#define MATCH_FMADD_D 0x2000043
#define MASK_FMADD_D  0x600007f
#define MATCH_FMSUB_D 0x2000047
#define MASK_FMSUB_D  0x600007f
#define MATCH_FNMSUB_D 0x200004b
#define MASK_FNMSUB_D  0x600007f
#define MATCH_FNMADD_D 0x200004f
#define MASK_FNMADD_D  0x600007f
#define MATCH_C_NOP 0x1
#define MASK_C_NOP  0xffff
#define MATCH_C_ADDI16SP 0x6101
#define MASK_C_ADDI16SP  0xef83
#define MATCH_C_JR 0x8002
#define MASK_C_JR  0xf07f
#define MATCH_C_JALR 0x9002
#define MASK_C_JALR  0xf07f
#define MATCH_C_EBREAK 0x9002
```

```
#define MASK_C_EBREAK  0xffff
#define MATCH_C_LD 0x6000
#define MASK_C_LD  0xe003
#define MATCH_C_SD 0xe000
#define MASK_C_SD  0xe003
#define MATCH_C_ADDIW 0x2001
#define MASK_C_ADDIW  0xe003
#define MATCH_C_LDSP 0x6002
#define MASK_C_LDSP  0xe003
#define MATCH_C_SDSP 0xe002
#define MASK_C_SDSP  0xe003
#define MATCH_C_ADDI4SPN 0x0
#define MASK_C_ADDI4SPN  0xe003
#define MATCH_C_FLD 0x2000
#define MASK_C_FLD  0xe003
#define MATCH_C_LW 0x4000
#define MASK_C_LW  0xe003
#define MATCH_C_FLW 0x6000
#define MASK_C_FLW  0xe003
#define MATCH_C_FSD 0xa000
#define MASK_C_FSD  0xe003
#define MATCH_C_SW 0xc000
#define MASK_C_SW  0xe003
#define MATCH_C_FSW 0xe000
#define MASK_C_FSW  0xe003
#define MATCH_C_ADDI 0x1
#define MASK_C_ADDI  0xe003
#define MATCH_C_JAL 0x2001
#define MASK_C_JAL  0xe003
#define MATCH_C_LI 0x4001
#define MASK_C_LI  0xe003
#define MATCH_C_LUI 0x6001
#define MASK_C_LUI  0xe003
#define MATCH_C_SRLI 0x8001
#define MASK_C_SRLI  0xec03
#define MATCH_C_SRAI 0x8401
#define MASK_C_SRAI  0xec03
#define MATCH_C_ANDI 0x8801
#define MASK_C_ANDI  0xec03
#define MATCH_C_SUB 0x8c01
#define MASK_C_SUB  0xfc63
#define MATCH_C_XOR 0x8c21
#define MASK_C_XOR  0xfc63
#define MATCH_C_OR 0x8c41
#define MASK_C_OR  0xfc63
#define MATCH_C_AND 0x8c61
#define MASK_C_AND  0xfc63
#define MATCH_C_SUBW 0x9c01
#define MASK_C_SUBW  0xfc63
#define MATCH_C_ADDW 0x9c21
#define MASK_C_ADDW  0xfc63
#define MATCH_C_J 0xa001
#define MASK_C_J  0xe003
#define MATCH_C_BEQZ 0xc001
#define MASK_C_BEQZ  0xe003
#define MATCH_C_BNEZ 0xe001
#define MASK_C_BNEZ  0xe003
#define MATCH_C_SLLI 0x2
#define MASK_C_SLLI  0xe003
#define MATCH_C_FLDSP 0x2002
#define MASK_C_FLDSP  0xe003
#define MATCH_C_LWSP 0x4002
#define MASK_C_LWSP  0xe003
#define MATCH_C_FLWSP 0x6002
#define MASK_C_FLWSP  0xe003
#define MATCH_C_MV 0x8002
#define MASK_C_MV  0xf003
#define MATCH_C_ADD 0x9002
#define MASK_C_ADD  0xf003
#define MATCH_C_FSDSP 0xa002
#define MASK_C_FSDSP  0xe003
#define MATCH_C_SWSP 0xc002
#define MASK_C_SWSP  0xe003
#define MATCH_C_FSWSP 0xe002
#define MASK_C_FSWSP  0xe003
#define MATCH_CUSTOM0 0xb
#define MASK_CUSTOM0  0x707f
#define MATCH_CUSTOM0_RS1 0x200b
#define MASK_CUSTOM0_RS1  0x707f
#define MATCH_CUSTOM0_RS1_RS2 0x300b
```

```
#define MASK_CUSTOM0_RS1_RS2  0x707f
#define MATCH_CUSTOM0_RD 0x400b
#define MASK_CUSTOM0_RD  0x707f
#define MATCH_CUSTOM0_RD_RS1 0x600b
#define MASK_CUSTOM0_RD_RS1  0x707f
#define MATCH_CUSTOM0_RD_RS1_RS2 0x700b
#define MASK_CUSTOM0_RD_RS1_RS2  0x707f
#define MATCH_CUSTOM1 0x2b
#define MASK_CUSTOM1  0x707f
#define MATCH_CUSTOM1_RS1 0x202b
#define MASK_CUSTOM1_RS1  0x707f
#define MATCH_CUSTOM1_RS1_RS2 0x302b
#define MASK_CUSTOM1_RS1_RS2  0x707f
#define MATCH_CUSTOM1_RD 0x402b
#define MASK_CUSTOM1_RD  0x707f
#define MATCH_CUSTOM1_RD_RS1 0x602b
#define MASK_CUSTOM1_RD_RS1  0x707f
#define MATCH_CUSTOM1_RD_RS1_RS2 0x702b
#define MASK_CUSTOM1_RD_RS1_RS2  0x707f
#define MATCH_CUSTOM2 0x5b
#define MASK_CUSTOM2  0x707f
#define MATCH_CUSTOM2_RS1 0x205b
#define MASK_CUSTOM2_RS1  0x707f
#define MATCH_CUSTOM2_RS1_RS2 0x305b
#define MASK_CUSTOM2_RS1_RS2  0x707f
#define MATCH_CUSTOM2_RD 0x405b
#define MASK_CUSTOM2_RD  0x707f
#define MATCH_CUSTOM2_RD_RS1 0x605b
#define MASK_CUSTOM2_RD_RS1  0x707f
#define MATCH_CUSTOM2_RD_RS1_RS2 0x705b
#define MASK_CUSTOM2_RD_RS1_RS2  0x707f
#define MATCH_CUSTOM3 0x7b
#define MASK_CUSTOM3  0x707f
#define MATCH_CUSTOM3_RS1 0x207b
#define MASK_CUSTOM3_RS1  0x707f
#define MATCH_CUSTOM3_RS1_RS2 0x307b
#define MASK_CUSTOM3_RS1_RS2  0x707f
#define MATCH_CUSTOM3_RD 0x407b
#define MASK_CUSTOM3_RD  0x707f
#define MATCH_CUSTOM3_RD_RS1 0x607b
#define MASK_CUSTOM3_RD_RS1  0x707f
#define MATCH_CUSTOM3_RD_RS1_RS2 0x707b
#define MASK_CUSTOM3_RD_RS1_RS2  0x707f
#define CSR_FFLAGS 0x1
#define CSR_FRM 0x2
#define CSR_FCSR 0x3
#define CSR_CYCLE 0xc00
#define CSR_TIME 0xc01
#define CSR_INSTRET 0xc02
#define CSR_HPMCOUNTER3 0xc03
#define CSR_HPMCOUNTER4 0xc04
#define CSR_HPMCOUNTER5 0xc05
#define CSR_HPMCOUNTER6 0xc06
#define CSR_HPMCOUNTER7 0xc07
#define CSR_HPMCOUNTER8 0xc08
#define CSR_HPMCOUNTER9 0xc09
#define CSR_HPMCOUNTER10 0xc0a
#define CSR_HPMCOUNTER11 0xc0b
#define CSR_HPMCOUNTER12 0xc0c
#define CSR_HPMCOUNTER13 0xc0d
#define CSR_HPMCOUNTER14 0xc0e
#define CSR_HPMCOUNTER15 0xc0f
#define CSR_HPMCOUNTER16 0xc10
#define CSR_HPMCOUNTER17 0xc11
#define CSR_HPMCOUNTER18 0xc12
#define CSR_HPMCOUNTER19 0xc13
#define CSR_HPMCOUNTER20 0xc14
#define CSR_HPMCOUNTER21 0xc15
#define CSR_HPMCOUNTER22 0xc16
#define CSR_HPMCOUNTER23 0xc17
#define CSR_HPMCOUNTER24 0xc18
#define CSR_HPMCOUNTER25 0xc19
#define CSR_HPMCOUNTER26 0xc1a
#define CSR_HPMCOUNTER27 0xc1b
#define CSR_HPMCOUNTER28 0xc1c
#define CSR_HPMCOUNTER29 0xc1d
#define CSR_HPMCOUNTER30 0xc1e
#define CSR_HPMCOUNTER31 0xc1f
#define CSR_SSTATUS 0x100
#define CSR_SIE 0x104
```

```
#define CSR_STVEC 0x105
#define CSR_SSCRATCH 0x140
#define CSR_SEPC 0x141
#define CSR_SCAUSE 0x142
#define CSR_SBADADDR 0x143
#define CSR_SIP 0x144
#define CSR_SPTBR 0x180
#define CSR_MSTATUS 0x300
#define CSR_MISA 0x301
#define CSR_MEDELEG 0x302
#define CSR_MIDELEG 0x303
#define CSR_MIE 0x304
#define CSR_MTVEC 0x305
#define CSR_MSCRATCH 0x340
#define CSR_MEPC 0x341
#define CSR_MCAUSE 0x342
#define CSR_MBADADDR 0x343
#define CSR_MIP 0x344
#define CSR_TSELECT 0x7a0
#define CSR_TDATA1 0x7a1
#define CSR_TDATA2 0x7a2
#define CSR_TDATA3 0x7a3
#define CSR_DCSR 0x7b0
#define CSR_DPC 0x7b1
#define CSR_DSCRATCH 0x7b2
#define CSR_MCYCLE 0xb00
#define CSR_MINSTRET 0xb02
#define CSR_MHPMCOUNTER3 0xb03
#define CSR_MHPMCOUNTER4 0xb04
#define CSR_MHPMCOUNTER5 0xb05
#define CSR_MHPMCOUNTER6 0xb06
#define CSR_MHPMCOUNTER7 0xb07
#define CSR_MHPMCOUNTER8 0xb08
#define CSR_MHPMCOUNTER9 0xb09
#define CSR_MHPMCOUNTER10 0xb0a
#define CSR_MHPMCOUNTER11 0xb0b
#define CSR_MHPMCOUNTER12 0xb0c
#define CSR_MHPMCOUNTER13 0xb0d
#define CSR_MHPMCOUNTER14 0xb0e
#define CSR_MHPMCOUNTER15 0xb0f
#define CSR_MHPMCOUNTER16 0xb10
#define CSR_MHPMCOUNTER17 0xb11
#define CSR_MHPMCOUNTER18 0xb12
#define CSR_MHPMCOUNTER19 0xb13
#define CSR_MHPMCOUNTER20 0xb14
#define CSR_MHPMCOUNTER21 0xb15
#define CSR_MHPMCOUNTER22 0xb16
#define CSR_MHPMCOUNTER23 0xb17
#define CSR_MHPMCOUNTER24 0xb18
#define CSR_MHPMCOUNTER25 0xb19
#define CSR_MHPMCOUNTER26 0xb1a
#define CSR_MHPMCOUNTER27 0xb1b
#define CSR_MHPMCOUNTER28 0xb1c
#define CSR_MHPMCOUNTER29 0xb1d
#define CSR_MHPMCOUNTER30 0xb1e
#define CSR_MHPMCOUNTER31 0xb1f
#define CSR_MUCOUNTEREN 0x320
#define CSR_MSCOUNTEREN 0x321
#define CSR_MHPMEVENT3 0x323
#define CSR_MHPMEVENT4 0x324
#define CSR_MHPMEVENT5 0x325
#define CSR_MHPMEVENT6 0x326
#define CSR_MHPMEVENT7 0x327
#define CSR_MHPMEVENT8 0x328
#define CSR_MHPMEVENT9 0x329
#define CSR_MHPMEVENT10 0x32a
#define CSR_MHPMEVENT11 0x32b
#define CSR_MHPMEVENT12 0x32c
#define CSR_MHPMEVENT13 0x32d
#define CSR_MHPMEVENT14 0x32e
#define CSR_MHPMEVENT15 0x32f
#define CSR_MHPMEVENT16 0x330
#define CSR_MHPMEVENT17 0x331
#define CSR_MHPMEVENT18 0x332
#define CSR_MHPMEVENT19 0x333
#define CSR_MHPMEVENT20 0x334
#define CSR_MHPMEVENT21 0x335
#define CSR_MHPMEVENT22 0x336
#define CSR_MHPMEVENT23 0x337
#define CSR_MHPMEVENT24 0x338
```

```
#define CSR_MHPMEVENT25 0x339
#define CSR_MHPMEVENT26 0x33a
#define CSR_MHPMEVENT27 0x33b
#define CSR_MHPMEVENT28 0x33c
#define CSR_MHPMEVENT29 0x33d
#define CSR_MHPMEVENT30 0x33e
#define CSR_MHPMEVENT31 0x33f
#define CSR_MVENDORID 0xf11
#define CSR_MARCHID 0xf12
#define CSR_MIMPID 0xf13
#define CSR_MHARTID 0xf14
#define CSR_CYCLEH 0xc80
#define CSR_TIMEH 0xc81
#define CSR_INSTRETH 0xc82
#define CSR_HPMCOUNTER3H 0xc83
#define CSR_HPMCOUNTER4H 0xc84
#define CSR_HPMCOUNTER5H 0xc85
#define CSR_HPMCOUNTER6H 0xc86
#define CSR_HPMCOUNTER7H 0xc87
#define CSR_HPMCOUNTER8H 0xc88
#define CSR_HPMCOUNTER9H 0xc89
#define CSR_HPMCOUNTER10H 0xc8a
#define CSR_HPMCOUNTER11H 0xc8b
#define CSR_HPMCOUNTER12H 0xc8c
#define CSR_HPMCOUNTER13H 0xc8d
#define CSR_HPMCOUNTER14H 0xc8e
#define CSR_HPMCOUNTER15H 0xc8f
#define CSR_HPMCOUNTER16H 0xc90
#define CSR_HPMCOUNTER17H 0xc91
#define CSR_HPMCOUNTER18H 0xc92
#define CSR_HPMCOUNTER19H 0xc93
#define CSR_HPMCOUNTER20H 0xc94
#define CSR_HPMCOUNTER21H 0xc95
#define CSR_HPMCOUNTER22H 0xc96
#define CSR_HPMCOUNTER23H 0xc97
#define CSR_HPMCOUNTER24H 0xc98
#define CSR_HPMCOUNTER25H 0xc99
#define CSR_HPMCOUNTER26H 0xc9a
#define CSR_HPMCOUNTER27H 0xc9b
#define CSR_HPMCOUNTER28H 0xc9c
#define CSR_HPMCOUNTER29H 0xc9d
#define CSR_HPMCOUNTER30H 0xc9e
#define CSR_HPMCOUNTER31H 0xc9f
#define CSR_MCYCLEH 0xb80
#define CSR_MINSTRETH 0xb82
#define CSR_MHPMCOUNTER3H 0xb83
#define CSR_MHPMCOUNTER4H 0xb84
#define CSR_MHPMCOUNTER5H 0xb85
#define CSR_MHPMCOUNTER6H 0xb86
#define CSR_MHPMCOUNTER7H 0xb87
#define CSR_MHPMCOUNTER8H 0xb88
#define CSR_MHPMCOUNTER9H 0xb89
#define CSR_MHPMCOUNTER10H 0xb8a
#define CSR_MHPMCOUNTER11H 0xb8b
#define CSR_MHPMCOUNTER12H 0xb8c
#define CSR_MHPMCOUNTER13H 0xb8d
#define CSR_MHPMCOUNTER14H 0xb8e
#define CSR_MHPMCOUNTER15H 0xb8f
#define CSR_MHPMCOUNTER16H 0xb90
#define CSR_MHPMCOUNTER17H 0xb91
#define CSR_MHPMCOUNTER18H 0xb92
#define CSR_MHPMCOUNTER19H 0xb93
#define CSR_MHPMCOUNTER20H 0xb94
#define CSR_MHPMCOUNTER21H 0xb95
#define CSR_MHPMCOUNTER22H 0xb96
#define CSR_MHPMCOUNTER23H 0xb97
#define CSR_MHPMCOUNTER24H 0xb98
#define CSR_MHPMCOUNTER25H 0xb99
#define CSR_MHPMCOUNTER26H 0xb9a
#define CSR_MHPMCOUNTER27H 0xb9b
#define CSR_MHPMCOUNTER28H 0xb9c
#define CSR_MHPMCOUNTER29H 0xb9d
#define CSR_MHPMCOUNTER30H 0xb9e
#define CSR_MHPMCOUNTER31H 0xb9f
#define CAUSE_MISALIGNED_FETCH 0x0
#define CAUSE_FAULT_FETCH 0x1
#define CAUSE_ILLEGAL_INSTRUCTION 0x2
#define CAUSE_BREAKPOINT 0x3
#define CAUSE_MISALIGNED_LOAD 0x4
#define CAUSE_FAULT_LOAD 0x5
```

```
#define CAUSE_MISALIGNED_STORE 0x6
#define CAUSE_FAULT_STORE 0x7
#define CAUSE_USER_ECALL 0x8
#define CAUSE_SUPERVISOR_ECALL 0x9
#define CAUSE_HYPERVISOR_ECALL 0xa
#define CAUSE_MACHINE_ECALL 0xb
#endif
#ifdef DECLARE_INSN
DECLARE_INSN(beq, MATCH_BEQ, MASK_BEQ)
DECLARE_INSN(bne, MATCH_BNE, MASK_BNE)
DECLARE_INSN(blt, MATCH_BLT, MASK_BLT)
DECLARE_INSN(bge, MATCH_BGE, MASK_BGE)
DECLARE_INSN(bltu, MATCH_BLTU, MASK_BLTU)
DECLARE_INSN(bgeu, MATCH_BGEU, MASK_BGEU)
DECLARE_INSN(jalr, MATCH_JALR, MASK_JALR)
DECLARE_INSN(jal, MATCH_JAL, MASK_JAL)
DECLARE_INSN(lui, MATCH_LUI, MASK_LUI)
DECLARE_INSN(auipc, MATCH_AUIPC, MASK_AUIPC)
DECLARE_INSN(addi, MATCH_ADDI, MASK_ADDI)
DECLARE_INSN(slli, MATCH_SLLI, MASK_SLLI)
DECLARE_INSN(slti, MATCH_SLTI, MASK_SLTI)
DECLARE_INSN(sltiu, MATCH_SLTIU, MASK_SLTIU)
DECLARE_INSN(xori, MATCH_XORI, MASK_XORI)
DECLARE_INSN(srli, MATCH_SRLI, MASK_SRLI)
DECLARE_INSN(srai, MATCH_SRAI, MASK_SRAI)
DECLARE_INSN(ori, MATCH_ORI, MASK_ORI)
DECLARE_INSN(andi, MATCH_ANDI, MASK_ANDI)
DECLARE_INSN(add, MATCH_ADD, MASK_ADD)
DECLARE_INSN(sub, MATCH_SUB, MASK_SUB)
DECLARE_INSN(sll, MATCH_SLL, MASK_SLL)
DECLARE_INSN(slt, MATCH_SLT, MASK_SLT)
DECLARE_INSN(sltu, MATCH_SLTU, MASK_SLTU)
DECLARE_INSN(xor, MATCH_XOR, MASK_XOR)
DECLARE_INSN(srl, MATCH_SRL, MASK_SRL)
DECLARE_INSN(sra, MATCH_SRA, MASK_SRA)
DECLARE_INSN(or, MATCH_OR, MASK_OR)
DECLARE_INSN(and, MATCH_AND, MASK_AND)
DECLARE_INSN(addiw, MATCH_ADDIW, MASK_ADDIW)
DECLARE_INSN(slliw, MATCH_SLLIW, MASK_SLLIW)
DECLARE_INSN(srliw, MATCH_SRLIW, MASK_SRLIW)
DECLARE_INSN(sraiw, MATCH_SRAIW, MASK_SRAIW)
DECLARE_INSN(addw, MATCH_ADDW, MASK_ADDW)
DECLARE_INSN(subw, MATCH_SUBW, MASK_SUBW)
DECLARE_INSN(sllw, MATCH_SLLW, MASK_SLLW)
DECLARE_INSN(srlw, MATCH_SRLW, MASK_SRLW)
DECLARE_INSN(sraw, MATCH_SRAW, MASK_SRAW)
DECLARE_INSN(lb, MATCH_LB, MASK_LB)
DECLARE_INSN(lh, MATCH_LH, MASK_LH)
DECLARE_INSN(lw, MATCH_LW, MASK_LW)
DECLARE_INSN(ld, MATCH_LD, MASK_LD)
DECLARE_INSN(lbu, MATCH_LBU, MASK_LBU)
DECLARE_INSN(lhu, MATCH_LHU, MASK_LHU)
DECLARE_INSN(lwu, MATCH_LWU, MASK_LWU)
DECLARE_INSN(sb, MATCH_SB, MASK_SB)
DECLARE_INSN(sh, MATCH_SH, MASK_SH)
DECLARE_INSN(sw, MATCH_SW, MASK_SW)
DECLARE_INSN(sd, MATCH_SD, MASK_SD)
DECLARE_INSN(fence, MATCH_FENCE, MASK_FENCE)
DECLARE_INSN(fence_i, MATCH_FENCE_I, MASK_FENCE_I)
DECLARE_INSN(mul, MATCH_MUL, MASK_MUL)
DECLARE_INSN(mulh, MATCH_MULH, MASK_MULH)
DECLARE_INSN(mulhsu, MATCH_MULHSU, MASK_MULHSU)
DECLARE_INSN(mulhu, MATCH_MULHU, MASK_MULHU)
DECLARE_INSN(div, MATCH_DIV, MASK_DIV)
DECLARE_INSN(divu, MATCH_DIVU, MASK_DIVU)
DECLARE_INSN(rem, MATCH_REM, MASK_REM)
DECLARE_INSN(remu, MATCH_REMU, MASK_REMU)
DECLARE_INSN(mulw, MATCH_MULW, MASK_MULW)
DECLARE_INSN(divw, MATCH_DIVW, MASK_DIVW)
DECLARE_INSN(divuw, MATCH_DIVUW, MASK_DIVUW)
DECLARE_INSN(remw, MATCH_REMW, MASK_REMW)
DECLARE_INSN(remuw, MATCH_REMUW, MASK_REMUW)
DECLARE_INSN(amoadd_w, MATCH_AMOADD_W, MASK_AMOADD_W)
DECLARE_INSN(amoxor_w, MATCH_AMOXOR_W, MASK_AMOXOR_W)
DECLARE_INSN(amoor_w, MATCH_AMOOR_W, MASK_AMOOR_W)
DECLARE_INSN(amoand_w, MATCH_AMOAND_W, MASK_AMOAND_W)
DECLARE_INSN(amomin_w, MATCH_AMOMIN_W, MASK_AMOMIN_W)
DECLARE_INSN(amomax_w, MATCH_AMOMAX_W, MASK_AMOMAX_W)
DECLARE_INSN(amominu_w, MATCH_AMOMINU_W, MASK_AMOMINU_W)
DECLARE_INSN(amomaxu_w, MATCH_AMOMAXU_W, MASK_AMOMAXU_W)
```

```
DECLARE_INSN(amoswap_w, MATCH_AMOSWAP_W, MASK_AMOSWAP_W)
DECLARE_INSN(lr_w, MATCH_LR_W, MASK_LR_W)
DECLARE_INSN(sc_w, MATCH_SC_W, MASK_SC_W)
DECLARE_INSN(amoadd_d, MATCH_AMOADD_D, MASK_AMOADD_D)
DECLARE_INSN(amoxor_d, MATCH_AMOXOR_D, MASK_AMOXOR_D)
DECLARE_INSN(amoor_d, MATCH_AMOOR_D, MASK_AMOOR_D)
DECLARE_INSN(amoand_d, MATCH_AMOAND_D, MASK_AMOAND_D)
DECLARE_INSN(amomin_d, MATCH_AMOMIN_D, MASK_AMOMIN_D)
DECLARE_INSN(amomax_d, MATCH_AMOMAX_D, MASK_AMOMAX_D)
DECLARE_INSN(amominu_d, MATCH_AMOMINU_D, MASK_AMOMINU_D)
DECLARE_INSN(amomaxu_d, MATCH_AMOMAXU_D, MASK_AMOMAXU_D)
DECLARE_INSN(amoswap_d, MATCH_AMOSWAP_D, MASK_AMOSWAP_D)
DECLARE_INSN(lr_d, MATCH_LR_D, MASK_LR_D)
DECLARE_INSN(sc_d, MATCH_SC_D, MASK_SC_D)
DECLARE_INSN(ecall, MATCH_ECALL, MASK_ECALL)
DECLARE_INSN(ebreak, MATCH_EBREAK, MASK_EBREAK)
DECLARE_INSN(uret, MATCH_URET, MASK_URET)
DECLARE_INSN(sret, MATCH_SRET, MASK_SRET)
DECLARE_INSN(hret, MATCH_HRET, MASK_HRET)
DECLARE_INSN(mret, MATCH_MRET, MASK_MRET)
DECLARE_INSN(dret, MATCH_DRET, MASK_DRET)
DECLARE_INSN(sfence_vm, MATCH_SFENCE_VM, MASK_SFENCE_VM)
DECLARE_INSN(wfi, MATCH_WFI, MASK_WFI)
DECLARE_INSN(csrrw, MATCH_CSRRW, MASK_CSRRW)
DECLARE_INSN(csrrs, MATCH_CSRRS, MASK_CSRRS)
DECLARE_INSN(csrrc, MATCH_CSRRC, MASK_CSRRC)
DECLARE_INSN(csrrwi, MATCH_CSRRWI, MASK_CSRRWI)
DECLARE_INSN(csrrsi, MATCH_CSRRSI, MASK_CSRRSI)
DECLARE_INSN(csrrci, MATCH_CSRRCI, MASK_CSRRCI)
DECLARE_INSN(fadd_s, MATCH_FADD_S, MASK_FADD_S)
DECLARE_INSN(fsub_s, MATCH_FSUB_S, MASK_FSUB_S)
DECLARE_INSN(fmul_s, MATCH_FMUL_S, MASK_FMUL_S)
DECLARE_INSN(fdiv_s, MATCH_FDIV_S, MASK_FDIV_S)
DECLARE_INSN(fsgnj_s, MATCH_FSGNJ_S, MASK_FSGNJ_S)
DECLARE_INSN(fsgnjn_s, MATCH_FSGNJN_S, MASK_FSGNJN_S)
DECLARE_INSN(fsgnjx_s, MATCH_FSGNJX_S, MASK_FSGNJX_S)
DECLARE_INSN(fmin_s, MATCH_FMIN_S, MASK_FMIN_S)
DECLARE_INSN(fmax_s, MATCH_FMAX_S, MASK_FMAX_S)
DECLARE_INSN(fsqrt_s, MATCH_FSQRT_S, MASK_FSQRT_S)
DECLARE_INSN(fadd_d, MATCH_FADD_D, MASK_FADD_D)
DECLARE_INSN(fsub_d, MATCH_FSUB_D, MASK_FSUB_D)
DECLARE_INSN(fmul_d, MATCH_FMUL_D, MASK_FMUL_D)
DECLARE_INSN(fdiv_d, MATCH_FDIV_D, MASK_FDIV_D)
DECLARE_INSN(fsgnj_d, MATCH_FSGNJ_D, MASK_FSGNJ_D)
DECLARE_INSN(fsgnjn_d, MATCH_FSGNJN_D, MASK_FSGNJN_D)
DECLARE_INSN(fsgnjx_d, MATCH_FSGNJX_D, MASK_FSGNJX_D)
DECLARE_INSN(fmin_d, MATCH_FMIN_D, MASK_FMIN_D)
DECLARE_INSN(fmax_d, MATCH_FMAX_D, MASK_FMAX_D)
DECLARE_INSN(fcvt_s_d, MATCH_FCVT_S_D, MASK_FCVT_S_D)
DECLARE_INSN(fcvt_d_s, MATCH_FCVT_D_S, MASK_FCVT_D_S)
DECLARE_INSN(fsqrt_d, MATCH_FSQRT_D, MASK_FSQRT_D)
DECLARE_INSN(fle_s, MATCH_FLE_S, MASK_FLE_S)
DECLARE_INSN(flt_s, MATCH_FLT_S, MASK_FLT_S)
DECLARE_INSN(feq_s, MATCH_FEQ_S, MASK_FEQ_S)
DECLARE_INSN(fle_d, MATCH_FLE_D, MASK_FLE_D)
DECLARE_INSN(flt_d, MATCH_FLT_D, MASK_FLT_D)
DECLARE_INSN(feq_d, MATCH_FEQ_D, MASK_FEQ_D)
DECLARE_INSN(fcvt_w_s, MATCH_FCVT_W_S, MASK_FCVT_W_S)
DECLARE_INSN(fcvt_wu_s, MATCH_FCVT_WU_S, MASK_FCVT_WU_S)
DECLARE_INSN(fcvt_l_s, MATCH_FCVT_L_S, MASK_FCVT_L_S)
DECLARE_INSN(fcvt_lu_s, MATCH_FCVT_LU_S, MASK_FCVT_LU_S)
DECLARE_INSN(fmv_x_s, MATCH_FMV_X_S, MASK_FMV_X_S)
DECLARE_INSN(fclass_s, MATCH_FCLASS_S, MASK_FCLASS_S)
DECLARE_INSN(fcvt_w_d, MATCH_FCVT_W_D, MASK_FCVT_W_D)
DECLARE_INSN(fcvt_wu_d, MATCH_FCVT_WU_D, MASK_FCVT_WU_D)
DECLARE_INSN(fcvt_l_d, MATCH_FCVT_L_D, MASK_FCVT_L_D)
DECLARE_INSN(fcvt_lu_d, MATCH_FCVT_LU_D, MASK_FCVT_LU_D)
DECLARE_INSN(fmv_x_d, MATCH_FMV_X_D, MASK_FMV_X_D)
DECLARE_INSN(fclass_d, MATCH_FCLASS_D, MASK_FCLASS_D)
DECLARE_INSN(fcvt_s_w, MATCH_FCVT_S_W, MASK_FCVT_S_W)
DECLARE_INSN(fcvt_s_wu, MATCH_FCVT_S_WU, MASK_FCVT_S_WU)
DECLARE_INSN(fcvt_s_l, MATCH_FCVT_S_L, MASK_FCVT_S_L)
DECLARE_INSN(fcvt_s_lu, MATCH_FCVT_S_LU, MASK_FCVT_S_LU)
DECLARE_INSN(fmv_s_x, MATCH_FMV_S_X, MASK_FMV_S_X)
DECLARE_INSN(fcvt_d_w, MATCH_FCVT_D_W, MASK_FCVT_D_W)
DECLARE_INSN(fcvt_d_wu, MATCH_FCVT_D_WU, MASK_FCVT_D_WU)
DECLARE_INSN(fcvt_d_l, MATCH_FCVT_D_L, MASK_FCVT_D_L)
DECLARE_INSN(fcvt_d_lu, MATCH_FCVT_D_LU, MASK_FCVT_D_LU)
DECLARE_INSN(fmv_d_x, MATCH_FMV_D_X, MASK_FMV_D_X)
DECLARE_INSN(flw, MATCH_FLW, MASK_FLW)
```

```
DECLARE_INSN(fld, MATCH_FLD, MASK_FLD)
DECLARE_INSN(fsw, MATCH_FSW, MASK_FSW)
DECLARE_INSN(fsd, MATCH_FSD, MASK_FSD)
DECLARE_INSN(fmadd_s, MATCH_FMADD_S, MASK_FMADD_S)
DECLARE_INSN(fmsub_s, MATCH_FMSUB_S, MASK_FMSUB_S)
DECLARE_INSN(fnmsub_s, MATCH_FNMSUB_S, MASK_FNMSUB_S)
DECLARE_INSN(fnmadd_s, MATCH_FNMADD_S, MASK_FNMADD_S)
DECLARE_INSN(fmadd_d, MATCH_FMADD_D, MASK_FMADD_D)
DECLARE_INSN(fmsub_d, MATCH_FMSUB_D, MASK_FMSUB_D)
DECLARE_INSN(fnmsub_d, MATCH_FNMSUB_D, MASK_FNMSUB_D)
DECLARE_INSN(fnmadd_d, MATCH_FNMADD_D, MASK_FNMADD_D)
DECLARE_INSN(c_nop, MATCH_C_NOP, MASK_C_NOP)
DECLARE_INSN(c_addi16sp, MATCH_C_ADDI16SP, MASK_C_ADDI16SP)
DECLARE_INSN(c_jr, MATCH_C_JR, MASK_C_JR)
DECLARE_INSN(c_jalr, MATCH_C_JALR, MASK_C_JALR)
DECLARE_INSN(c_ebreak, MATCH_C_EBREAK, MASK_C_EBREAK)
DECLARE_INSN(c_ld, MATCH_C_LD, MASK_C_LD)
DECLARE_INSN(c_sd, MATCH_C_SD, MASK_C_SD)
DECLARE_INSN(c_addiw, MATCH_C_ADDIW, MASK_C_ADDIW)
DECLARE_INSN(c_ldsp, MATCH_C_LDSP, MASK_C_LDSP)
DECLARE_INSN(c_sdsp, MATCH_C_SDSP, MASK_C_SDSP)
DECLARE_INSN(c_addi4spn, MATCH_C_ADDI4SPN, MASK_C_ADDI4SPN)
DECLARE_INSN(c_fld, MATCH_C_FLD, MASK_C_FLD)
DECLARE_INSN(c_lw, MATCH_C_LW, MASK_C_LW)
DECLARE_INSN(c_flw, MATCH_C_FLW, MASK_C_FLW)
DECLARE_INSN(c_fsd, MATCH_C_FSD, MASK_C_FSD)
DECLARE_INSN(c_sw, MATCH_C_SW, MASK_C_SW)
DECLARE_INSN(c_fsw, MATCH_C_FSW, MASK_C_FSW)
DECLARE_INSN(c_addi, MATCH_C_ADDI, MASK_C_ADDI)
DECLARE_INSN(c_jal, MATCH_C_JAL, MASK_C_JAL)
DECLARE_INSN(c_li, MATCH_C_LI, MASK_C_LI)
DECLARE_INSN(c_lui, MATCH_C_LUI, MASK_C_LUI)
DECLARE_INSN(c_srli, MATCH_C_SRLI, MASK_C_SRLI)
DECLARE_INSN(c_srai, MATCH_C_SRAI, MASK_C_SRAI)
DECLARE_INSN(c_andi, MATCH_C_ANDI, MASK_C_ANDI)
DECLARE_INSN(c_sub, MATCH_C_SUB, MASK_C_SUB)
DECLARE_INSN(c_xor, MATCH_C_XOR, MASK_C_XOR)
DECLARE_INSN(c_or, MATCH_C_OR, MASK_C_OR)
DECLARE_INSN(c_and, MATCH_C_AND, MASK_C_AND)
DECLARE_INSN(c_subw, MATCH_C_SUBW, MASK_C_SUBW)
DECLARE_INSN(c_addw, MATCH_C_ADDW, MASK_C_ADDW)
DECLARE_INSN(c_j, MATCH_C_J, MASK_C_J)
DECLARE_INSN(c_beqz, MATCH_C_BEQZ, MASK_C_BEQZ)
DECLARE_INSN(c_bnez, MATCH_C_BNEZ, MASK_C_BNEZ)
DECLARE_INSN(c_slli, MATCH_C_SLLI, MASK_C_SLLI)
DECLARE_INSN(c_fldsp, MATCH_C_FLDSP, MASK_C_FLDSP)
DECLARE_INSN(c_lwsp, MATCH_C_LWSP, MASK_C_LWSP)
DECLARE_INSN(c_flwsp, MATCH_C_FLWSP, MASK_C_FLWSP)
DECLARE_INSN(c_mv, MATCH_C_MV, MASK_C_MV)
DECLARE_INSN(c_add, MATCH_C_ADD, MASK_C_ADD)
DECLARE_INSN(c_fsdsp, MATCH_C_FSDSP, MASK_C_FSDSP)
DECLARE_INSN(c_swsp, MATCH_C_SWSP, MASK_C_SWSP)
DECLARE_INSN(c_fswsp, MATCH_C_FSWSP, MASK_C_FSWSP)
DECLARE_INSN(custom0, MATCH_CUSTOM0, MASK_CUSTOM0)
DECLARE_INSN(custom0_rs1, MATCH_CUSTOM0_RS1, MASK_CUSTOM0_RS1)
DECLARE_INSN(custom0_rs1_rs2, MATCH_CUSTOM0_RS1_RS2, MASK_CUSTOM0_RS1_RS2)
DECLARE_INSN(custom0_rd, MATCH_CUSTOM0_RD, MASK_CUSTOM0_RD)
DECLARE_INSN(custom0_rd_rs1, MATCH_CUSTOM0_RD_RS1, MASK_CUSTOM0_RD_RS1)
DECLARE_INSN(custom0_rd_rs1_rs2, MATCH_CUSTOM0_RD_RS1_RS2, MASK_CUSTOM0_RD_RS1_RS2)
DECLARE_INSN(custom1, MATCH_CUSTOM1, MASK_CUSTOM1)
DECLARE_INSN(custom1_rs1, MATCH_CUSTOM1_RS1, MASK_CUSTOM1_RS1)
DECLARE_INSN(custom1_rs1_rs2, MATCH_CUSTOM1_RS1_RS2, MASK_CUSTOM1_RS1_RS2)
DECLARE_INSN(custom1_rd, MATCH_CUSTOM1_RD, MASK_CUSTOM1_RD)
DECLARE_INSN(custom1_rd_rs1, MATCH_CUSTOM1_RD_RS1, MASK_CUSTOM1_RD_RS1)
DECLARE_INSN(custom1_rd_rs1_rs2, MATCH_CUSTOM1_RD_RS1_RS2, MASK_CUSTOM1_RD_RS1_RS2)
DECLARE_INSN(custom2, MATCH_CUSTOM2, MASK_CUSTOM2)
DECLARE_INSN(custom2_rs1, MATCH_CUSTOM2_RS1, MASK_CUSTOM2_RS1)
DECLARE_INSN(custom2_rs1_rs2, MATCH_CUSTOM2_RS1_RS2, MASK_CUSTOM2_RS1_RS2)
DECLARE_INSN(custom2_rd, MATCH_CUSTOM2_RD, MASK_CUSTOM2_RD)
DECLARE_INSN(custom2_rd_rs1, MATCH_CUSTOM2_RD_RS1, MASK_CUSTOM2_RD_RS1)
DECLARE_INSN(custom2_rd_rs1_rs2, MATCH_CUSTOM2_RD_RS1_RS2, MASK_CUSTOM2_RD_RS1_RS2)
DECLARE_INSN(custom3, MATCH_CUSTOM3, MASK_CUSTOM3)
DECLARE_INSN(custom3_rs1, MATCH_CUSTOM3_RS1, MASK_CUSTOM3_RS1)
DECLARE_INSN(custom3_rs1_rs2, MATCH_CUSTOM3_RS1_RS2, MASK_CUSTOM3_RS1_RS2)
DECLARE_INSN(custom3_rd, MATCH_CUSTOM3_RD, MASK_CUSTOM3_RD)
DECLARE_INSN(custom3_rd_rs1, MATCH_CUSTOM3_RD_RS1, MASK_CUSTOM3_RD_RS1)
DECLARE_INSN(custom3_rd_rs1_rs2, MATCH_CUSTOM3_RD_RS1_RS2, MASK_CUSTOM3_RD_RS1_RS2)
#endif
#ifdef DECLARE_CSR
DECLARE_CSR(fflags, CSR_FFLAGS)
```

```
DECLARE_CSR(frm, CSR_FRM)
DECLARE_CSR(fcsr, CSR_FCSR)
DECLARE_CSR(cycle, CSR_CYCLE)
DECLARE_CSR(time, CSR_TIME)
DECLARE_CSR(instret, CSR_INSTRET)
DECLARE_CSR(hpmcounter3, CSR_HPMCOUNTER3)
DECLARE_CSR(hpmcounter4, CSR_HPMCOUNTER4)
DECLARE_CSR(hpmcounter5, CSR_HPMCOUNTER5)
DECLARE_CSR(hpmcounter6, CSR_HPMCOUNTER6)
DECLARE_CSR(hpmcounter7, CSR_HPMCOUNTER7)
DECLARE_CSR(hpmcounter8, CSR_HPMCOUNTER8)
DECLARE_CSR(hpmcounter9, CSR_HPMCOUNTER9)
DECLARE_CSR(hpmcounter10, CSR_HPMCOUNTER10)
DECLARE_CSR(hpmcounter11, CSR_HPMCOUNTER11)
DECLARE_CSR(hpmcounter12, CSR_HPMCOUNTER12)
DECLARE_CSR(hpmcounter13, CSR_HPMCOUNTER13)
DECLARE_CSR(hpmcounter14, CSR_HPMCOUNTER14)
DECLARE_CSR(hpmcounter15, CSR_HPMCOUNTER15)
DECLARE_CSR(hpmcounter16, CSR_HPMCOUNTER16)
DECLARE_CSR(hpmcounter17, CSR_HPMCOUNTER17)
DECLARE_CSR(hpmcounter18, CSR_HPMCOUNTER18)
DECLARE_CSR(hpmcounter19, CSR_HPMCOUNTER19)
DECLARE_CSR(hpmcounter20, CSR_HPMCOUNTER20)
DECLARE_CSR(hpmcounter21, CSR_HPMCOUNTER21)
DECLARE_CSR(hpmcounter22, CSR_HPMCOUNTER22)
DECLARE_CSR(hpmcounter23, CSR_HPMCOUNTER23)
DECLARE_CSR(hpmcounter24, CSR_HPMCOUNTER24)
DECLARE_CSR(hpmcounter25, CSR_HPMCOUNTER25)
DECLARE_CSR(hpmcounter26, CSR_HPMCOUNTER26)
DECLARE_CSR(hpmcounter27, CSR_HPMCOUNTER27)
DECLARE_CSR(hpmcounter28, CSR_HPMCOUNTER28)
DECLARE_CSR(hpmcounter29, CSR_HPMCOUNTER29)
DECLARE_CSR(hpmcounter30, CSR_HPMCOUNTER30)
DECLARE_CSR(hpmcounter31, CSR_HPMCOUNTER31)
DECLARE_CSR(sstatus, CSR_SSTATUS)
DECLARE_CSR(sie, CSR_SIE)
DECLARE_CSR(stvec, CSR_STVEC)
DECLARE_CSR(sscratch, CSR_SSCRATCH)
DECLARE_CSR(sepc, CSR_SEPC)
DECLARE_CSR(scause, CSR_SCAUSE)
DECLARE_CSR(sbadaddr, CSR_SBADADDR)
DECLARE_CSR(sip, CSR_SIP)
DECLARE_CSR(sptbr, CSR_SPTBR)
DECLARE_CSR(mstatus, CSR_MSTATUS)
DECLARE_CSR(misa, CSR_MISA)
DECLARE_CSR(medeleg, CSR_MEDELEG)
DECLARE_CSR(mideleg, CSR_MIDELEG)
DECLARE_CSR(mie, CSR_MIE)
DECLARE_CSR(mtvec, CSR_MTVEC)
DECLARE_CSR(mscratch, CSR_MSCRATCH)
DECLARE_CSR(mepc, CSR_MEPC)
DECLARE_CSR(mcause, CSR_MCAUSE)
DECLARE_CSR(mbadaddr, CSR_MBADADDR)
DECLARE_CSR(mip, CSR_MIP)
DECLARE_CSR(tselect, CSR_TSELECT)
DECLARE_CSR(tdata1, CSR_TDATA1)
DECLARE_CSR(tdata2, CSR_TDATA2)
DECLARE_CSR(tdata3, CSR_TDATA3)
DECLARE_CSR(dcsr, CSR_DCSR)
DECLARE_CSR(dpc, CSR_DPC)
DECLARE_CSR(dscratch, CSR_DSCRATCH)
DECLARE_CSR(mcycle, CSR_MCYCLE)
DECLARE_CSR(minstret, CSR_MINSTRET)
DECLARE_CSR(mhpmcounter3, CSR_MHPMCOUNTER3)
DECLARE_CSR(mhpmcounter4, CSR_MHPMCOUNTER4)
DECLARE_CSR(mhpmcounter5, CSR_MHPMCOUNTER5)
DECLARE_CSR(mhpmcounter6, CSR_MHPMCOUNTER6)
DECLARE_CSR(mhpmcounter7, CSR_MHPMCOUNTER7)
DECLARE_CSR(mhpmcounter8, CSR_MHPMCOUNTER8)
DECLARE_CSR(mhpmcounter9, CSR_MHPMCOUNTER9)
DECLARE_CSR(mhpmcounter10, CSR_MHPMCOUNTER10)
DECLARE_CSR(mhpmcounter11, CSR_MHPMCOUNTER11)
DECLARE_CSR(mhpmcounter12, CSR_MHPMCOUNTER12)
DECLARE_CSR(mhpmcounter13, CSR_MHPMCOUNTER13)
DECLARE_CSR(mhpmcounter14, CSR_MHPMCOUNTER14)
DECLARE_CSR(mhpmcounter15, CSR_MHPMCOUNTER15)
DECLARE_CSR(mhpmcounter16, CSR_MHPMCOUNTER16)
DECLARE_CSR(mhpmcounter17, CSR_MHPMCOUNTER17)
DECLARE_CSR(mhpmcounter18, CSR_MHPMCOUNTER18)
DECLARE_CSR(mhpmcounter19, CSR_MHPMCOUNTER19)
```

```
DECLARE_CSR(mhpmcounter20, CSR_MHPMCOUNTER20)
DECLARE_CSR(mhpmcounter21, CSR_MHPMCOUNTER21)
DECLARE_CSR(mhpmcounter22, CSR_MHPMCOUNTER22)
DECLARE_CSR(mhpmcounter23, CSR_MHPMCOUNTER23)
DECLARE_CSR(mhpmcounter24, CSR_MHPMCOUNTER24)
DECLARE_CSR(mhpmcounter25, CSR_MHPMCOUNTER25)
DECLARE_CSR(mhpmcounter26, CSR_MHPMCOUNTER26)
DECLARE_CSR(mhpmcounter27, CSR_MHPMCOUNTER27)
DECLARE_CSR(mhpmcounter28, CSR_MHPMCOUNTER28)
DECLARE_CSR(mhpmcounter29, CSR_MHPMCOUNTER29)
DECLARE_CSR(mhpmcounter30, CSR_MHPMCOUNTER30)
DECLARE_CSR(mhpmcounter31, CSR_MHPMCOUNTER31)
DECLARE_CSR(mucounteren, CSR_MUCOUNTEREN)
DECLARE_CSR(mscounteren, CSR_MSCOUNTEREN)
DECLARE_CSR(mhpmevent3, CSR_MHPMEVENT3)
DECLARE_CSR(mhpmevent4, CSR_MHPMEVENT4)
DECLARE_CSR(mhpmevent5, CSR_MHPMEVENT5)
DECLARE_CSR(mhpmevent6, CSR_MHPMEVENT6)
DECLARE_CSR(mhpmevent7, CSR_MHPMEVENT7)
DECLARE_CSR(mhpmevent8, CSR_MHPMEVENT8)
DECLARE_CSR(mhpmevent9, CSR_MHPMEVENT9)
DECLARE_CSR(mhpmevent10, CSR_MHPMEVENT10)
DECLARE_CSR(mhpmevent11, CSR_MHPMEVENT11)
DECLARE_CSR(mhpmevent12, CSR_MHPMEVENT12)
DECLARE_CSR(mhpmevent13, CSR_MHPMEVENT13)
DECLARE_CSR(mhpmevent14, CSR_MHPMEVENT14)
DECLARE_CSR(mhpmevent15, CSR_MHPMEVENT15)
DECLARE_CSR(mhpmevent16, CSR_MHPMEVENT16)
DECLARE_CSR(mhpmevent17, CSR_MHPMEVENT17)
DECLARE_CSR(mhpmevent18, CSR_MHPMEVENT18)
DECLARE_CSR(mhpmevent19, CSR_MHPMEVENT19)
DECLARE_CSR(mhpmevent20, CSR_MHPMEVENT20)
DECLARE_CSR(mhpmevent21, CSR_MHPMEVENT21)
DECLARE_CSR(mhpmevent22, CSR_MHPMEVENT22)
DECLARE_CSR(mhpmevent23, CSR_MHPMEVENT23)
DECLARE_CSR(mhpmevent24, CSR_MHPMEVENT24)
DECLARE_CSR(mhpmevent25, CSR_MHPMEVENT25)
DECLARE_CSR(mhpmevent26, CSR_MHPMEVENT26)
DECLARE_CSR(mhpmevent27, CSR_MHPMEVENT27)
DECLARE_CSR(mhpmevent28, CSR_MHPMEVENT28)
DECLARE_CSR(mhpmevent29, CSR_MHPMEVENT29)
DECLARE_CSR(mhpmevent30, CSR_MHPMEVENT30)
DECLARE_CSR(mhpmevent31, CSR_MHPMEVENT31)
DECLARE_CSR(mvendorid, CSR_MVENDORID)
DECLARE_CSR(marchid, CSR_MARCHID)
DECLARE_CSR(mimpid, CSR_MIMPID)
DECLARE_CSR(mhartid, CSR_MHARTID)
DECLARE_CSR(cycleh, CSR_CYCLEH)
DECLARE_CSR(timeh, CSR_TIMEH)
DECLARE_CSR(instreth, CSR_INSTRETH)
DECLARE_CSR(hpmcounter3h, CSR_HPMCOUNTER3H)
DECLARE_CSR(hpmcounter4h, CSR_HPMCOUNTER4H)
DECLARE_CSR(hpmcounter5h, CSR_HPMCOUNTER5H)
DECLARE_CSR(hpmcounter6h, CSR_HPMCOUNTER6H)
DECLARE_CSR(hpmcounter7h, CSR_HPMCOUNTER7H)
DECLARE_CSR(hpmcounter8h, CSR_HPMCOUNTER8H)
DECLARE_CSR(hpmcounter9h, CSR_HPMCOUNTER9H)
DECLARE_CSR(hpmcounter10h, CSR_HPMCOUNTER10H)
DECLARE_CSR(hpmcounter11h, CSR_HPMCOUNTER11H)
DECLARE_CSR(hpmcounter12h, CSR_HPMCOUNTER12H)
DECLARE_CSR(hpmcounter13h, CSR_HPMCOUNTER13H)
DECLARE_CSR(hpmcounter14h, CSR_HPMCOUNTER14H)
DECLARE_CSR(hpmcounter15h, CSR_HPMCOUNTER15H)
DECLARE_CSR(hpmcounter16h, CSR_HPMCOUNTER16H)
DECLARE_CSR(hpmcounter17h, CSR_HPMCOUNTER17H)
DECLARE_CSR(hpmcounter18h, CSR_HPMCOUNTER18H)
DECLARE_CSR(hpmcounter19h, CSR_HPMCOUNTER19H)
DECLARE_CSR(hpmcounter20h, CSR_HPMCOUNTER20H)
DECLARE_CSR(hpmcounter21h, CSR_HPMCOUNTER21H)
DECLARE_CSR(hpmcounter22h, CSR_HPMCOUNTER22H)
DECLARE_CSR(hpmcounter23h, CSR_HPMCOUNTER23H)
DECLARE_CSR(hpmcounter24h, CSR_HPMCOUNTER24H)
DECLARE_CSR(hpmcounter25h, CSR_HPMCOUNTER25H)
DECLARE_CSR(hpmcounter26h, CSR_HPMCOUNTER26H)
DECLARE_CSR(hpmcounter27h, CSR_HPMCOUNTER27H)
DECLARE_CSR(hpmcounter28h, CSR_HPMCOUNTER28H)
DECLARE_CSR(hpmcounter29h, CSR_HPMCOUNTER29H)
DECLARE_CSR(hpmcounter30h, CSR_HPMCOUNTER30H)
DECLARE_CSR(hpmcounter31h, CSR_HPMCOUNTER31H)
DECLARE_CSR(mcycleh, CSR_MCYCLEH)
```

```
DECLARE_CSR(minstreth, CSR_MINSTRETH)
DECLARE_CSR(mhpmcounter3h, CSR_MHPMCOUNTER3H)
DECLARE_CSR(mhpmcounter4h, CSR_MHPMCOUNTER4H)
DECLARE_CSR(mhpmcounter5h, CSR_MHPMCOUNTER5H)
DECLARE_CSR(mhpmcounter6h, CSR_MHPMCOUNTER6H)
DECLARE_CSR(mhpmcounter7h, CSR_MHPMCOUNTER7H)
DECLARE_CSR(mhpmcounter8h, CSR_MHPMCOUNTER8H)
DECLARE_CSR(mhpmcounter9h, CSR_MHPMCOUNTER9H)
DECLARE_CSR(mhpmcounter10h, CSR_MHPMCOUNTER10H)
DECLARE_CSR(mhpmcounter11h, CSR_MHPMCOUNTER11H)
DECLARE_CSR(mhpmcounter12h, CSR_MHPMCOUNTER12H)
DECLARE_CSR(mhpmcounter13h, CSR_MHPMCOUNTER13H)
DECLARE_CSR(mhpmcounter14h, CSR_MHPMCOUNTER14H)
DECLARE_CSR(mhpmcounter15h, CSR_MHPMCOUNTER15H)
DECLARE_CSR(mhpmcounter16h, CSR_MHPMCOUNTER16H)
DECLARE_CSR(mhpmcounter17h, CSR_MHPMCOUNTER17H)
DECLARE_CSR(mhpmcounter18h, CSR_MHPMCOUNTER18H)
DECLARE_CSR(mhpmcounter19h, CSR_MHPMCOUNTER19H)
DECLARE_CSR(mhpmcounter20h, CSR_MHPMCOUNTER20H)
DECLARE_CSR(mhpmcounter21h, CSR_MHPMCOUNTER21H)
DECLARE_CSR(mhpmcounter22h, CSR_MHPMCOUNTER22H)
DECLARE_CSR(mhpmcounter23h, CSR_MHPMCOUNTER23H)
DECLARE_CSR(mhpmcounter24h, CSR_MHPMCOUNTER24H)
DECLARE_CSR(mhpmcounter25h, CSR_MHPMCOUNTER25H)
DECLARE_CSR(mhpmcounter26h, CSR_MHPMCOUNTER26H)
DECLARE_CSR(mhpmcounter27h, CSR_MHPMCOUNTER27H)
DECLARE_CSR(mhpmcounter28h, CSR_MHPMCOUNTER28H)
DECLARE_CSR(mhpmcounter29h, CSR_MHPMCOUNTER29H)
DECLARE_CSR(mhpmcounter30h, CSR_MHPMCOUNTER30H)
DECLARE_CSR(mhpmcounter31h, CSR_MHPMCOUNTER31H)
#endif
#ifdef DECLARE_CAUSE
DECLARE_CAUSE("misaligned fetch", CAUSE_MISALIGNED_FETCH)
DECLARE_CAUSE("fault fetch", CAUSE_FAULT_FETCH)
DECLARE_CAUSE("illegal instruction", CAUSE_ILLEGAL_INSTRUCTION)
DECLARE_CAUSE("breakpoint", CAUSE_BREAKPOINT)
DECLARE_CAUSE("misaligned load", CAUSE_MISALIGNED_LOAD)
DECLARE_CAUSE("fault load", CAUSE_FAULT_LOAD)
DECLARE_CAUSE("misaligned store", CAUSE_MISALIGNED_STORE)
DECLARE_CAUSE("fault store", CAUSE_FAULT_STORE)
DECLARE_CAUSE("user_ecall", CAUSE_USER_ECALL)
DECLARE_CAUSE("supervisor_ecall", CAUSE_SUPERVISOR_ECALL)
DECLARE_CAUSE("hypervisor_ecall", CAUSE_HYPERVISOR_ECALL)
DECLARE_CAUSE("machine_ecall", CAUSE_MACHINE_ECALL)
#endif
```

Código 71: enconding.h.

```
export RISCV=$TOP/riscv
export PK=$RISCV/riscv64-unknown-elf/bin
export PATH=$PATH:$RISCV/bin:$PK
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:rocket-chip/riscv-tools/riscv/lib
```

Código 72: Variáveis de ambiente para configuração do Rocket Chip.

```
#=======================================================================
# UCB VLSI FLOW: Makefile for riscv-bmarks
#-----------------------------------------------------------------------
# Yunsup Lee (yunsup@cs.berkeley.edu)
#-----------------------------------------------------------------------
# Modify by Gabriel Villanova
#--------------------------------------------------------------------
# Set arch.
#--------------------------------------------------------------------
XLEN ?= 64
ISA  ?= rv64imafd
ABI  ?= lp64
APP   = main

#--------------------------------------------------------------------
# Sources and Includes
#--------------------------------------------------------------------
SOURCES = ./src
INCS    += -I./include
LINKER  = link.ld

#--------------------------------------------------------------------
# Build rules
#--------------------------------------------------------------------
RISCV_PREFIX ?= riscv$(XLEN)-unknown-elf-
RISCV_GCC ?= $(RISCV_PREFIX)gcc
RISCV_GCC_OPTS ?= -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -ffreestanding \
         -static -nostdlib -nostartfiles -std=gnu99 -O0 -ffast-math -fno-common $(INCS)
RISCV_LINK ?= $(RISCV_GCC) -T$(LINKER) $(INCS)
RISCV_LINK_OPTS ?= -static -nostdlib -nostartfiles -lm -lgcc -T$(LINKER)
RISCV_OBJDUMP ?= $(RISCV_PREFIX)objdump --disassemble-all --disassemble-zeroes \
         --section=.text --section=.text.startup --section=.data
RISCV_SIM ?= spike --isa=rv$(XLEN)gc
#RISCV_GCC_OPTS ?= -DPREALLOCATE=1 -mcmodel=medany -static -std=gnu99 -O2 \
         -ffast-math -fno-common -fno-builtin-printf

#--------------------------------------------------------------
# make all
all: obj link size

#--------------------------------------------------------------
# size
size:
	$(RISCV_PREFIX)size $(APP)

#--------------------------------------------------------------
# make link
link:
	$(RISCV_LINK) $(SOURCES)/syscalls.o $(SOURCES)/crt.o \
		$(SOURCES)/$(APP).o $(RISCV_LINK_OPTS) -o $(APP)

#--------------------------------------------------------------
# make objects .o
obj:
	$(RISCV_GCC) $(RISCV_GCC_OPTS) $(INCS) -c $(SOURCES)/syscalls.c -o $(SOURCES)/syscalls.o
	$(RISCV_GCC) $(RISCV_GCC_OPTS) $(INCS) -c $(SOURCES)/crt.S -o $(SOURCES)/crt.o
	$(RISCV_GCC) $(RISCV_GCC_OPTS) $(INCS) -o $(SOURCES)/$(APP).o -c $(SOURCES)/$(APP).c

sim:
	@spike $(APP)

mem_dump:
	$(RISCV_PREFIX)objdump -d $(APP)

hex:
	riscv64-unknown-elf-objcopy -I binary -O ihex $(APP) $(APP).hex
	@#riscv64-unknown-elf-objcopy -O ihex $(APP) $(APP).hex


clean:
	rm -rf src/*.o
	rm -rf a.out
	rm -rf $(APP)
```

Código 73: Makefile.

Anexo B: Testbench RoCC FIR AxC

```
/* @author : Gabriel Villanova N. Magalhães
 * @company: Laboratory TIMA
 * @date    : July 2018
 */

module tb_coprocessor();
  /* Defines */
  `define SYNTHESIS

  /* Parameters */
  localparam CLOCK          =5;
  localparam CLOCK_CYCLE =10;
  localparam WIDTH        =64;
  localparam ADDR_COEF    =5;
  localparam NB_COEFS     =32;
  localparam NB_SAMPLES  =2000;
  localparam ADDR_PRECIS =32;
  localparam ADDR_STATUS =33;
  localparam ADDR_RESULT =34;

  /* Wires to interface module */
  logic CLK = 1'b1;
  logic RST = 1'b1;

  /* RoCC Interface */

  // RISC-V instruction coprocessor:
  // FUNCT[7] | RS2 | RS1 | XD XRS1 XRS2 | RD | OPCODE
  logic [6:0]  cmd_inst_funct;
  logic [4:0]  cmd_inst_rs2 = 0;
  logic [4:0]  cmd_inst_rs1 = 0;
  logic [4:0]  cmd_inst_rd;

  // Data values
  logic [WIDTH-1:0] cmd_rs1;
  logic [WIDTH-1:0] cmd_rs2;

  // READY/VALID
  logic cmd_ready;
  logic cmd_valid = 0;

  // memory
  // input
  logic mem_req_ready = 1'b0;

  // outputs
  logic resp_valid;
  logic [39:0] mem_req_addr;
  logic [9:0] mem_req_tag;
  logic [4:0] mem_req_cmd;
  logic [2:0] mem_req_typ;
  logic [WIDTH-1:0] mem_req_data;
  logic mem_req_phys;
  logic mem_req_valid;
  logic mem_invalidate_lr;
  logic busy;
  logic interrupt;

  /* input signal and filtre coefs (1 byte) */
  logic [WIDTH-1:0] inputSignal[NB_SAMPLES];
  logic [WIDTH-1:0] coef[32] =
  {
   'h00D0000000000000,  'h0150000000000000,  'h01F0000000000000,  'h02C0000000000000,
   'h03C0000000000000,  'h04D0000000000000,  'h0610000000000000,  'h0750000000000000,
   'h08A0000000000000,  'h09F0000000000000,  'h0B30000000000000,  'h0C50000000000000,
   'h0D40000000000000,  'h0E10000000000000,  'h0E90000000000000,  'h0EE0000000000000,
   'h0EE0000000000000,  'h0E90000000000000,  'h0E10000000000000,  'h0D40000000000000,
   'h0C50000000000000,  'h0B30000000000000,  'h09F0000000000000,  'h08A0000000000000,
   'h0750000000000000,  'h0610000000000000,  'h04D0000000000000,  'h03C0000000000000,
   'h02C0000000000000,  'h01F0000000000000,  'h0150000000000000,  'h00D0000000000000
   // 'h0D00,  'h1500,  'h1F00,  'h2C00,
   // 'h3C00,  'h4D00,  'h6100,  'h7500,
   // 'h8A00,  'h9F00,  'hB300,  'hC500,
   // 'hD400,  'hE100,  'hE900,  'hEE00,
   // 'hEE00,  'hE900,  'hE100,  'hD400,
   // 'hC500,  'hB300,  'h9F00,  'h8A00,
   // 'h7500,  'h6100,  'h4D00,  'h3C00,
```

```
 // 'h2C00,  'h1F00,  'h1500,  'h0D00
};

/* DUT connection */
CoprocessorFIR dut_coprocessor(
.clk(CLK),
.reset(RST),
.io_rocc_cmd_inst_funct(cmd_inst_funct),
.io_rocc_cmd_inst_rs2(cmd_inst_rs2),
.io_rocc_cmd_inst_rs1(cmd_inst_rs1),
.io_rocc_cmd_inst_rd(cmd_inst_rd),
.io_rocc_cmd_rs1(cmd_rs1),
.io_rocc_cmd_rs2(cmd_rs2),
.io_rocc_cmd_ready(cmd_ready),
.io_rocc_cmd_valid(cmd_valid),
.io_rocc_resp_valid(resp_valid),
.io_rocc_mem_req_addr(mem_req_addr),
.io_rocc_mem_req_tag(mem_req_tag),
.io_rocc_mem_req_cmd(mem_req_cmd),
.io_rocc_mem_req_typ(mem_req_typ),
.io_rocc_mem_req_data(mem_req_data),
.io_rocc_mem_req_phys(mem_req_phys),
.io_rocc_mem_req_valid(mem_req_valid),
.io_rocc_mem_req_ready(mem_req_ready),
.io_rocc_mem_invalidate_lr(mem_invalidate_lr),
.io_rocc_busy(busy),
.io_rocc_interrupt(interrupt)
);

/* Clock generator */
initial begin
  forever #CLOCK CLK = ~CLK;
end

/* Main */

// variables
int i = 0;
logic aux_done = 0;
logic [WIDTH-1:0] aux_mem_read = 0;
int aux_t=0;

// test
initial begin
  $display("\n************************");
  $display("Testbench starts!");
  $display("************************\n\n");

  flush_inputs();
  RST = 1'b1;
  #(CLOCK_CYCLE*2)
  RST = 1'b0;
  #5;

  /* Basic tests */

  // write i in each register
  // for (i = 0; i < 35; i++) begin
  //  if(i == ADDR_STATUS) begin
  //    s_mv(CLK, 'h0BAD0000, i);
  //    $display("i = %0d", i);
  //  end
  //  else begin
  //    s_mv(CLK, i,i);
  //  end
  // end
  // s_mv(CLK, 0, ADDR_PRECIS);

  // // make fifo puts
  // s_fifo_put(CLK, 'hDEAD);
  // s_fifo_put(CLK, 'hBEEF);
  // s_fifo_put(CLK, 'h0BAD);
  // s_fifo_put(CLK, 'hF00D);
  // s_fifo_put(CLK, 'h0);

  // // make a simple s_str instruction of status regs (BAD)
  // s_str(CLK, ADDR_STATUS, 'hCAFE, aux_mem_read);
  // $display("result read mem = %x", aux_mem_read);
  // aux_mem_read = 0;
```

```
  // // enable fir and wait bit done
  // s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt

  // aux_t = $time();
  // $display("\nstart polling:");
  // while(!aux_mem_read[WIDTH-1]) begin
  //   s_str(CLK, ADDR_STATUS, 'hCAFE, aux_mem_read); // "read" registers
  //   $display("Status Register = %x", aux_mem_read);
  // end
  // $display("end polling, time %0t!", $time()-aux_t);
  // aux_mem_read = 0;

  // // interrupt flag test
  // s_mv(CLK, 'h3, ADDR_STATUS); // clear bit done and enable FIR and interrupt
  // aux_t = $time();
  // $display("\nstart polling:");
  // while(!aux_mem_read[WIDTH-1]) begin
  //   s_str(CLK, ADDR_STATUS, 'hCAFE, aux_mem_read); // "read" registers
  //   $display("Status Register = %x", aux_mem_read);
  // end
  // $display("end polling, time %0t!", $time()-aux_t);
  // aux_mem_read = 0;

  // FIR test
  $readmemh("stimulus/ecg_64.txt", inputSignal); // (ref: frequency sample = 1k Hz)
  //$readmemh("stimulus/file_64_step.txt", inputSignal); // (ref: frequency sample = 1k Hz)
  foreach(inputSignal[i]) begin
    //$display("inputSignal[%d] = %x", i, inputSignal[i]);
  end
  $display("\n");

  // load coefs
  for (int i = 0; i < NB_COEFS; i++) begin
    s_mv(CLK, coef[i], i);
  end

  $display("Read coeficients in register file:\n");
  for (int i = 0; i < NB_COEFS; i++) begin
    s_str(CLK, i, 'h0, aux_mem_read); // "read" registers
    $display("coef[%0d]\t= %x", i, aux_mem_read);
  end
  aux_mem_read = 0;

  s_mv(CLK, 'h0, ADDR_STATUS); // clear status
  for (int i = 0; i < NB_SAMPLES; i++) begin
    s_fifo_put(CLK, inputSignal[i]);
    $display("data_in= %x", inputSignal[i]);

    // if(i == 500) begin
    //   s_mv(CLK, 'h0x1FFFFFFF, ADDR_PRECIS);
    //   s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt
    // end
    if(i == 500) begin
      s_mv(CLK, 'h0x2FFFFFFFFFFFFFFF, ADDR_PRECIS);
      s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt
    end else if(i == 1000) begin
      s_mv(CLK, 'h0x3FFFFFFFFFFFFFFF, ADDR_PRECIS);
      s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt
    end else if(i == 1500) begin
      s_mv(CLK, 'h0xFFFFFFFFFFFFFFFF, ADDR_PRECIS);
      s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt
    end else begin
      s_mv(CLK, 'h1, ADDR_STATUS); // enable FIR is active and not use interrupt
    end

    // wait done bit
    while(!aux_mem_read[WIDTH-1]) s_str(CLK, ADDR_STATUS, 'hCAFE, aux_mem_read); // "read" status regs

    s_str(CLK, ADDR_RESULT, 'h0BAD, aux_mem_read);
    $display("data_out= %x", aux_mem_read);

    s_mv(CLK, 'h0, ADDR_STATUS); // clear bit done
    aux_mem_read = 0;
  end

  $display("\n*************************");
  $display("End simulation =D");
  $display("*************************\n\n");
  $finish;
end // initial
```

```
  /* tasks */
  task flush_inputs();
    cmd_inst_funct = 0;
    cmd_inst_rs2 = 0;
    cmd_inst_rs1 = 0;
    cmd_rs1 = 0;
    cmd_rs2 = 0;
    cmd_valid = 0;
    mem_req_ready = 0;
  endtask : flush_inputs

  task automatic cmd_handshake(
    ref logic CLK
  );
    cmd_valid = 1'b1;
    @(posedge CLK);

    while(!cmd_ready) @(posedge CLK);
    cmd_valid = 1'b0;
    @(posedge CLK);
  endtask : cmd_handshake

  task automatic s_mv(
    ref logic CLK,
    input [WIDTH-1:0] data_rs1,
    input [WIDTH-1:0] address_rs2
  );
    cmd_inst_funct = 7'b0000000;
    cmd_rs1 = data_rs1; // data
    cmd_rs2 = address_rs2; // address

    cmd_handshake(CLK);
  endtask : s_mv

  task automatic s_fifo_put(
    ref logic CLK,
    input [WIDTH-1:0] data // rs1
  );
    cmd_inst_funct = 7'b0000001;
    cmd_rs1 = data;

    cmd_handshake(CLK);
  endtask : s_fifo_put

  task automatic s_str(
    ref logic CLK,
    input [WIDTH-1:0] register_address, // rs2
    input [WIDTH-1:0] mem_address, // rs1
    output [WIDTH-1:0] read
  );
    cmd_inst_funct = 7'b0000010;
    cmd_rs2 = register_address;
    cmd_rs1 = mem_address;
    cmd_handshake(CLK);

    // mem_handshake:
    mem_req_ready = 1'b1;
    @(posedge CLK);
    read = mem_req_data;
    mem_req_ready = 1'b0;
    @(posedge CLK);
  endtask : s_str

endmodule // tb_coprocessor
```

Código 74: Testbench para RoCC FIR AxC em SystemVerilog.

Anexo C: main.c para teste de RoCC FIR AxC

```
#include "encoding.h"
#include "in_signal.h"

#define STR1(x) #x
#define STR(x) STR1(x)
#define EXTRACT(a, size, offset) (((~(~0 << size) << offset) & a) >> offset)

#define CUSTOMX_OPCODE(x) CUSTOM_ ## x
#define CUSTOM_0 0b0001011
#define CUSTOM_1 0b0101011
#define CUSTOM_2 0b1011011
#define CUSTOM_3 0b1111011

// xd xs1 xs2 using always
#define CUSTOMX(X, rd, rs1, rs2, funct)          \
  CUSTOMX_OPCODE(X)                 |           \
  (rd                   << (7))         |           \
  (0x3                  << (7+5))       |           \
  (rs1                  << (7+5+3))     |           \
  (rs2                  << (7+5+3+5))   |           \
  (EXTRACT(funct, 7, 0) << (7+5+3+5+5))

// Standard macro that passes rd, rs1, and rs2 via registers
#define ROCC_INSTRUCTION(X, rd, rs1, rs2, funct)                \
  ROCC_INSTRUCTION_R_R_R(X, rd, rs1, rs2, funct, 5, 6,  7)

// rd, rs1, and rs2 are data
// rd_n, rs_1, and rs2_n are the register numbers to use
#define ROCC_INSTRUCTION_R_R_R(X, rd, rs1, rs2, funct, rd_n, rs1_n, rs2_n) {                \
    register unsigned long int rd_  asm ("x" # rd_n);                                  \
    register unsigned long int rs1_ asm ("x" # rs1_n) = (unsigned long int) rs1;       \
    register unsigned long int rs2_ asm ("x" # rs2_n) = (unsigned long int) rs2;       \
    asm volatile (                                                  \
        ".word " STR(CUSTOMX(X, rd_n, rs1_n, rs2_n, funct)) "\n\t"     \
        : "=r" (rd_)                                                \
        : [_rs1] "r" (rs1_), [_rs2] "r" (rs2_)                      \
        : "cc"                                                      \
        );                                                          \
    rd = rd_;                                                       \
  }
#define k_DO_WRITE 0
#define XCUSTOM_ACC 0
#define doWrite(y, rocc_rd, data)                                   \
  ROCC_INSTRUCTION(XCUSTOM_ACC, y, data, rocc_rd, k_DO_WRITE);

#define S_MV    0
#define S_FIFO  1
#define S_STR   2
#define NB_COEFS 32

unsigned long int h[NB_COEFS] =
{
    0x00D0000000000000, 0x0150000000000000, 0x01F0000000000000, 0x02C0000000000000,
    0x03C0000000000000, 0x04D0000000000000, 0x0610000000000000, 0x0750000000000000,
    0x08A0000000000000, 0x09F0000000000000, 0x0B30000000000000, 0x0C50000000000000,
    0x0D40000000000000, 0x0E10000000000000, 0x0E90000000000000, 0x0EE0000000000000,
    0x0EE0000000000000, 0x0E90000000000000, 0x0E10000000000000, 0x0D40000000000000,
    0x0C50000000000000, 0x0B30000000000000, 0x09F0000000000000, 0x08A0000000000000,
    0x0750000000000000, 0x0610000000000000, 0x04D0000000000000, 0x03C0000000000000,
    0x02C0000000000000, 0x01F0000000000000, 0x0150000000000000, 0x00D0000000000000
};
int main() {
//Funct(7bits)|rs2(5bits)|rs1(5bits)|xd(1bit)|xs1(1bit)|xs2(1bit)|rd(5bits)|opcode(7 bits)
// 0000001 01011 01010 000 01010 1111011
// rs2 = a1
// rs1 = a0
  unsigned long int y;
  unsigned long int read_h[NB_COEFS] = {0};
  unsigned long int status_read = 0;
  unsigned long int result = 0;

  write_csr(mie, 0);
  write_csr(sie, 0);
  write_csr(mip, 0);
  write_csr(sip, 0);
  write_csr(mideleg, 0);
  write_csr(medeleg, 0);
```

```c
  // enable machine interrupts
  __asm__("li       t0, 4096\n\t"// mcause = 12 to the RoCC int.
          "csrrs    zero, mie, t0\n\t" // Machine External Interrupt Enable
          "li       t0, 8\n\t"
          "csrrs    zero, mstatus, t0\n\t" // Machine Status enable Machine Int. Enable
          );

  // delay
  volatile int aux=500;
  while(aux != 0) aux--;

  for (int i = 0; i < 4; i++)
  {
    // take a sample
    ROCC_INSTRUCTION(0, y, in_signal[i], 0, S_FIFO)

    // enable rocc fir
    ROCC_INSTRUCTION(0, y, (unsigned long int)0x01, (unsigned long int)33, S_MV)

    // read status register
    ROCC_INSTRUCTION(0, y, &status_read, 33, S_STR)

    // is it done?
    while(!(status_read & 0x8000000000000000)) {
      ROCC_INSTRUCTION(0, y, &status_read, 33, S_STR) // ROCC_INSTRUCTION(0, y, &status_read, 33, S_STR)
      delay();
    }

    // it's done!
    ROCC_INSTRUCTION(0, y, &result, 34, S_STR)
    delay();
    //printf("result = 0x%lx\n", result);

    // acknowlegemnt that's done!
    ROCC_INSTRUCTION(0, y, (unsigned long int)0, (unsigned long int)33, S_MV)
    delay();
  }
  return 0;
}
```

Código 75: main.c para testar RoCC FIR AxC.