UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RAFAEL DE OLIVEIRA CALÇADA

Design of Steel: a RISC-V Core

Porto Alegre
2020

RAFAEL DE OLIVEIRA CALÇADA

Design of Steel: a RISC-V Core

Monografia apresentada ao Instituto de Informática da Universidade Federal do Rio Grande do Sul como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Ricardo Augusto da Luz Reis

Porto Alegre
2020

CIP - Catalogação na Publicação

Calçada, Rafael de Oliveira
Design of Steel: a RISC-V Core / Rafael de Oliveira Calçada. – 2020.
81 p. : il. : 21cm.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Curso de Engenharia de Computação, Porto Alegre, 2020.

Orientador: Ricardo Augusto da Luz Reis.

1. Projeto de circuitos integrados. 2. Microprocessadores. 3. RISC-V. I. Título.

AGRADECIMENTOS

Minha jornada para entrar na universidade foi peculiar. Não conheço nenhum colega (nem qualquer outro aluno) que tenha prestado o vestibular por seis vezes! Seis! Algum tempo atrás me questionei o porquê de não ter desistido na quarta ou quinta vez, quando tudo indicava que mais esforço talvez não valesse a pena. Os dois anos que antecederam meu ingresso na UFRGS foram de grandes reveses e, justamente por isso, meus anos de maior crescimento espiritual.

Do ingresso no curso até a formatura passaram-se sete anos. Nos quatro anos finais tive que administrar meu tempo entre aprender engenharia e um trabalho com carga horária de quarenta horas semanais. Fazer essa grande travessia só foi possível com a ajuda de muitas pessoas, a quem sou infinitamente grato!

Primeiramente, quero agradecer ao meu orientador, Ricardo, pelo incentivo à realização deste trabalho e pela generosidade.

Agradeço a minha chefe, Solange, pela compreensão e pelas inúmeras vezes em que me apoiou compatibilizando meus horários com aulas, provas e trabalhos. Agradeço pela troca de ensinamentos e pela amizade durante estes onze anos em que trabalhamos juntos.

Agradeço a todos os meus colegas de trabalho pela amizade e pelo ambiente sempre acolhedor. Eu só consegui conciliar trabalho e estudos porque pude contar com o carinho de vocês.

Sou grato aos meus colegas pela amizade e apoio ao longo do curso. Agradeço ao Gubert, ao Oberdan, ao Chico e a muitos outros colegas pelos milhões de cafés e pela conversa fora durante o caminho de volta pra casa. Chico merece um agradecimento em especial por me ajudar opinando sobre vários aspectos deste trabalho.

Agradeço ao meu companheiro, Guilherme, pela companhia, pela amizade, pela confiança, pelo carinho e pelo amor ao longo destes seis anos em que estamos juntos. Sem sua presença ao meu lado os dias seriam chatos e sem cor!

Finalmente, e não menos importante, agradeço à minha família, o meu esteio! Obrigado por me educar, por acreditar em mim, por me incentivar e por me encher de amor e carinho. Mil vezes obrigado! Não existem palavras para expressar minha gratidão e meu amor por vocês. Agradeço em especial à minha avó, Maria Helena, que me ensinou a ser grato e a ter compaixão; ao meu pai, um trabalhador incansável, meu maior exemplo do que é ter garra e disposição para enfrentar desafios; e à minha mãe, minha maior incentivadora, que sempre acreditou no meu potencial, que me ensinou a acreditar em mim mesmo e que me levantou nos tropeços que tive ao longo do caminho. O que vocês me ensinaram me deu a firmeza necessária para nunca desistir dos meus objetivos!

RESUMO

Este trabalho apresenta o projeto do Steel, um microprocessador com 3 estágios de pipeline que implementa os conjuntos de instruções RV32I e Zicsr das especificações do RISC-V. A descrição do hardware (em Verilog) está disponível sob a Licença MIT no repositório online do projeto. A conformidade com as especificações do RISC-V foi certificada pela aplicação dos testes da RISC-V Compliance Suite. A performance foi medida com o benchmark EEMBC® CoreMark, atingindo o escore de 1.36 CoreMarks/MHz. O consumo de recursos em um FPGA Artix-7 foi comparado a outras duas implementações RISC-V similares, Ibex e SCR1. O Steel mostrou-se competitivo, utilizando apenas 1.626 look-up tables e 624 flip-flops. O core é uma implementação open-source documentada e pronta para uso por projetistas de sistemas embarcados.

Palavras-chave: Projeto de circuitos integrados. Microprocessadores. RISC-V. FPGA. Microeletrônica.

ABSTRACT

This work presents the design of Steel, a microprocessor core with 3 pipeline stages that implements the instruction sets RV32I and Zicsr from RISC-V specifications. Its hardware description (in Verilog) is available under the MIT License in the project's online repository. The compliance with RISC-V specifications was certified by applying the RISC-V Compliance Suite tests. The performance was measured using the EEMBC® CoreMark benchmark, reaching a score of 1.36 CoreMarks/MHz. The resource usage in an Artix-7 FPGA was compared to two other similar RISC-V implementations, Ibex and SCR1. Steel proved to be competitive, using only 1.626 lookup tables and 624 flip-flops. The core is a documented open-source implementation and is ready for use by embedded systems designers.

Keywords: Integrated circuits design. Microprocessors. RISC-V. FPGA. Microelectronics.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSR	Control and Status Register
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose Input/Output
IPC	Instructions per Cycle
IPS	Instructions per Second
I/O	Input/Output
IoT	Internet of Things
ISA	Instruction Set Architecture
JSON	JavaScript Object Notation
MMU	Memory Management Unit
PC	Program Counter
RISC	Reduced Instruction Set Computer
RTOS	Real Time Operating System
SoC	System on Chip
UFRGS	Universidade Federal do Rio Grande do Sul
ULA	Unidade Lógica e Aritmética
VLIW	Very Long Instruction Word

SUMÁRIO

1 INTRODUÇÃO

Arquitetura do conjunto de instruções (ou ISA, do inglês Instruction Set Architecture) é um conjunto de especificações sobre as instruções que um computador é capaz de executar, além de características e recursos que devem estar nele presentes, como quantidade de registradores e modos de endereçamento de memória. Em conjunto, as instruções e características especificadas pela ISA determinam a forma de programar o computador, o que Patterson e Hennessy [1] chamam apropriadamente de *interface hardware/software*.

As instruções e os recursos especificados pela ISA podem ser implementados em hardware de diversas formas. Diferentes implementações da mesma arquitetura formam uma família de computadores. O software desenvolvido para determinada ISA pode ser executado por qualquer computador da mesma família, pois a interface hardware/software é a mesma. A performance da execução, por outro lado, é variável e dependente da implementação.

1.1 Arquiteturas CISC e RISC

Existem duas principais abordagens arquiteturais para o projeto e construção de computadores: CISC (Complex Instruction Set Computer) e RISC (Reduced Instruction Set Computer), que diferem entre si pela quantidade e complexidade das instruções. Em linhas gerais, as arquiteturas RISC possuem um conjunto de instruções significativamente menor e mais simples quando comparadas às arquiteturas CISC [2].

Nos primeiros anos que seguiram ao surgimento dos computadores baseados na arquitetura de Von Neumann (décadas de 1940-50) todos os computadores tinham, naturalmente, um conjunto pequeno de instruções, devido às severas limitações do hardware disponível à época. À medida em que o hardware foi ficando menor, mais barato e mais eficiente, a tendência entre os projetistas foi adicionar mais instruções e recursos a cada nova geração de computadores, o que levou ao surgimento das arquiteturas CISC. Como a programação, nesta época, normalmente era feita em linguagem de máquina, a adição de novas instruções e modos de endereçamento de memória facilitava o trabalho de programação, fazendo com que a abordagem CISC predominasse por um longo período de tempo. Intel x86, AMD, VAX, PDP-11 e Motorola 68000 são algumas das arquiteturas CISC mais famosas, sendo todas criadas entre as décadas de 1960-80.

O aumento da quantidade de instruções e a adição de recursos nas máquinas CISC implicou, no entanto, em um aumento significativo da complexidade do hardware projetado, sobretudo da lógica de decodificação e controle. Ainda na década de 80 foram publicados trabalhos (como [3]) sugerindo que computadores RISC, mais simples e com menos instruções, eram mais eficientes em alguns aspectos, como performance, área do chip e tempo de projeto. Também nesta época o desenvolvimento tecnológico levou ao surgimento de linguagens compiladas e de alto nível, e percebeu-se que para a maioria dos programas apenas um subconjunto reduzido das instruções disponíveis nas máquinas CISC era efetivamente utilizado.

Instalou-se um debate sobre as vantagens e desvantagens de cada abordagem [4], e as arquiteturas RISC ficaram progressivamente mais comuns com o tempo. ARM, MIPS, SPARC

(surgidas na década de 80) e PowerPC e DEC Alpha (surgidas na década de 90) são exemplos de arquiteturas RISC. Elas são caracterizadas pela presença de um pequeno conjunto de instruções, de poucos modos de endereçamento, e de bancos de registradores maiores para reduzir o número de acessos à memória.

1.2 Surgimento da arquitetura RISC-V

Nos dias atuais algumas arquiteturas são bastante populares, como Intel x86 (CISC) e ARM (RISC). Construir implementações compatíveis com essas arquiteturas possui muitos benefícios. Existe um amplo conjunto de softwares e documentação pré-existente, que podem ser prontamente portados para novas implementações. Além disso, há uma grande comunidade de desenvolvedores de software para essas ISAs, que dispõem de ambientes de desenvolvimento integrado para programação e outras facilidades.

Contudo, o desenvolvimento de hardware em conformidade com essas arquiteturas apresenta desvantagens importantes. Elas são proprietárias e possuem marcas e patentes associadas. Portanto, novas implementações podem requerer licenças dos proprietários ou acabar em disputas jurídicas. Além disso, elas possuem especificações complexas e difíceis de implementar, e suas implementações mais eficientes são mantidas em segredo pelos seus desenvolvedores. Por fim, elas são populares apenas em nichos de mercado específicos e correm o risco de perder a popularidade com o tempo [5].

Outra desvantagem relacionada a arquiteturas proprietárias reside no fato de serem dirigidas por uma única entidade privada, que pode mudar as especificações a qualquer momento e sem levar em consideração qualquer fator externo. Além disso, a complexidade das especificações destas arquiteturas as tornam problemáticas para uso em universidades como ferramenta ensino e pesquisa de arquitetura de computadores [6].

Com o objetivo de superar estas e outras desvantagens, um grupo de pesquisadores da University of California, Berkeley iniciou em 2010 o desenvolvimento de uma nova arquitetura de conjunto de instruções, o RISC-V, um padrão livre e aberto desde sua origem. A ISA do RISC-V foi desenvolvida para possibilitar a padronização, a colaboração entre empresas e indivíduos no desenvolvimento da arquitetura, e também para ser flexível e extensível. Desde seu lançamento, um número crescente de indivíduos e organizações tem adotado o RISC-V em novas implementações de microprocessadores. Em 2015 foi fundada a RISC-V Foundation, responsável por dirigir o desenvolvimento da arquitetura até 2020. Neste ano, após a transferência de sua sede para a Suíça, foi criada RISC-V International, entidade sucessora da fundação. O quadro de associados da RISC-V International tem hoje mais de 500 membros, entre eles algumas das principais companhias de tecnologia do mundo.

1.3 Objetivo do trabalho

O objetivo deste trabalho foi projetar o core de um microprocessador que implemente o conjunto básico de instruções RV32I [7], a extensão Zicsr [8] e a arquitetura privilegiada

do modo máquina [9] das especificações do RISC-V. A descrição do hardware projetado, em Verilog, foi disponibilizada sob uma licença livre e aberta, a Licença MIT, em um repositório online de projetos (GitHub).

A parte operativa do core foi implementada com a técnica de *pipelining*. O core foi projetado para despachar uma única instrução por ciclo de relógio, com execução na ordem do programa. A implementação é voltada para uso como unidade de processamento em projetos de microcontroladores e sistemas embarcados em FPGAs.

O funcionamento do core foi testado através de simulações (*testbenches*), através da execução de programas-teste e pela aplicação dos testes de compliance disponibilizados pela RISC-V International. Para facilitar o reúso por desenvolvedores, a implementação também disponibiliza documentação completa em língua inglesa.

A implementação foi avaliada quanto à performance através da aplicação de um dos principais benchmarks para sistemas embarcados, o CoreMark. Para atingir esse objetivo, o core foi integrado a um pequeno sistema com um transmissor UART, necessário para colher as informações de performance geradas pela execução do benchmark.

O core também foi comparado a outras duas implementações RISC-V similares. A comparação foi realizada através da síntese dos cores em um FPGA Artix-7 na plataforma Nexys-4 da Digilent. Os relatórios de consumo dos recursos do FPGA foram colhidos, tabulados e, então, analisados.

1.4 Motivação

Muitas universidades tem contribuído com a comunidade de hardware livre desenvolvendo sistemas computacionais baseados nestes princípios. University of California, Berkeley, ETH Zürich e Università di Bologna, por exemplo, são universidades pioneiras que tem portfólios de cores e systems-on-chip RISC-V de qualidade industrial livres e abertos. Muitas implementações de microprocessadores foram desenvolvidas na Universidade Federal do Rio Grande do Sul (UFRGS). No entanto, nenhuma das implementações desenvolvidas possuem as características propostas neste trabalho. Assim, este trabalho foi desenvolvido visando ampliar o portfólio de microprocessadores desenvolvidos pela UFRGS, bem como contribuir para a ampliação do ecossistema RISC-V e para o fortalecimento da comunidade de hardware livre.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo revisa conceitos essenciais para a compreensão do projeto de um core RISC-V. O capítulo inicia com uma revisão do conceito de arquitetura do conjunto de instruções (seção 2.1). Em seguida, a arquitetura RISC-V é apresentada (seção 2.2) e, por fim, são revisados alguns dos principais conceitos relacionados à organização de computadores (seção 2.3).

2.1 Arquitetura do conjunto de instruções

Chama-se *arquitetura do conjunto de instruções*[1] a descrição abstrata do comportamento de um computador relativamente ao seu conjunto de instruções, de forma independente ao modo como o computador é construído. O hardware de um dispositivo de acordo com tal descrição é chamado de *implementação* [10]. A ISA define uma interface para programação, possibilitando que programadores desenvolvam software abstraindo as particularidades das implementações. A arquitetura especifica quais instruções o computador executa, seus códigos de operação e modo de execução, recursos disponíveis (registradores, memórias, interrupções) e o modo de usá-los, além de outras características importantes da máquina (espaço e modos de endereçamento, modos de operação, etc.) [11].

Como mencionado na introdução, existem duas principais abordagens para o projeto da arquitetura de computadores: RISC (Reduced Instruction Set Computer) ou CISC (Complex Instruction Set Computer) [4]. As arquiteturas CISC possuem um conjunto grande de instruções capazes de realizar operações complexas, que envolvem a execução de várias operações menores (ler, modificar e gravar dados da memória com uma única instrução, por exemplo). As arquiteturas RISC, por outro lado, tem como característica a existência de um conjunto reduzido de instruções que executam operações simples. Essas arquiteturas, com algumas exceções, possuem poucos modos de endereçamento, acessam a memória através de instruções especiais (load/store) e suas instruções só podem operar sobre dados previamente carregados em registradores [2].

2.2 A arquitetura RISC-V

O RISC-V é uma ISA aberta, livre, flexível, extensível, dirigida por um grupo amplo de empresas e indivíduos (através da RISC-V International) e baseada na abordagem RISC. Ela é formada por um conjunto pequeno de instruções sobre inteiros, obrigatoriamente presente em qualquer implementação, ao qual podem ser adicionadas extensões, de implementação opcional[2]. As extensões (apresentadas na Tabela 1, p. 15) são voltadas à realização de tarefas específicas e melhoram a performance, possibilitando a especialização das implementações [13].

[1] Neste trabalho utilizam-se as expressões *arquitetura do conjunto de instruções, ISA* ou simplesmente *arquitetura* para se referir ao mesmo conceito.

[2] O conjunto básico de instruções pode emular as extensões que não forem implementadas, com exceção da extensão A [12].

A possibilidade de incluir (ou não) as extensões nas implementações torna o RISC-V adequado para muitas aplicações, de pequenos microcontroladores a grandes supercomputadores. Projetistas também podem especializar implementações para baixa potência, desempenho, segurança, etc. A facilidade de especialização das implementações (ou de adaptação a uma finalidade) é uma característica importante da arquitetura e também uma inovação importante que a distingue de outras arquiteturas RISC.

Tabela 1 – Extensões do RISC-V

Extensão	Status	Descrição
A	Ratificada	Contém instruções que lêem, modificam e escrevem atomicamente na memória.
M	Ratificada	Contém instruções para multiplicação e divisão de números inteiros.
F	Ratificada	Extensão para aritmética de números em ponto-flutuante de precisão simples conforme padrão IEEE 754-2008.
D	Ratificada	Extensão para aritmética de números em ponto-flutuante de precisão dupla conforme padrão IEEE 754-2008.
Q	Ratificada	Extensão para aritmética de números em ponto-flutuante de precisão quádrupla conforme padrão IEEE 754-2008.
C	Ratificada	Extensão para compressão de instruções, visando diminuir o tamanho do código.
Zicsr	Ratificada	Contém instruções para manipulação de registradores de estado e controle (CSR).
Zifencei	Ratificada	Contém instruções para sincronização de escritas e buscas à memória de instruções dentro da mesma hardware thread.
Ztso	Congelada	Altera o modelo de consistência de memória para uma versão mais restrita chamada Total Store Ordering.
Counters	Rascunho	Define os contadores de hardware da arquitetura.
Zam	Rascunho	Expande a extensão A, permitindo operações atômicas desalinhadas.

2.2.1 Especificações

As especificações do RISC-V estão divididas em dois volumes, subdivididos em capítulos dedicados a um módulo específico da arquitetura. O prefácio do volume indica o status de cada módulo contido nele. O status pode ser *ratificado*, indicando que não é esperado que o módulo sofra modificações; *congelado*, indicando que o módulo pode sofrer pequenas modificações antes de ser ratificado; ou *rascunho*, indicando que o módulo está sujeito a alterações.

O primeiro volume trata da arquitetura não-privilegiada, onde estão especificadas instruções normalmente executadas por software aplicativo. Neste volume estão especificados o conjunto básico de instruções (em diferentes versões) e diversas extensões. O volume também

apresenta diretrizes para nomenclatura de implementações e detalha o modelo de consistência de memória do RISC-V.

O segundo volume é dedicado à arquitetura privilegiada, que especifica o funcionamento de todos os recursos necessários para execução de sistemas operacionais e comunicação com dispositivos externos (I/O). A arquitetura privilegiada define o espaço de endereçamento dos registradores de estado e controle (CSRs) e as funcionalidades ativadas/desativadas através de leitura/escrita nesses registradores. Também define instruções executadas exclusivamente por software privilegiado, como sistemas operacionais. São definidos três níveis de privilégio, aos quais correspondem modos de operação: M-mode (modo máquina), S-mode (modo supervisor) e U-mode (modo usuário). Os modos mais privilegiados possuem controle mais amplo dos recursos da máquina.

2.2.2 Características

Assim como a maioria das arquiteturas RISC, o RISC-V é uma arquitetura do tipo load/store, o que significa que somente essa classe de instruções pode ler e escrever dados na memória. Todas as demais instruções operam sobre dados previamente carregados no banco de registradores, que dispõe de 32 registradores nomeados de x0-x31. Além destes, a arquitetura prevê um registrador especial, o contador de programa (PC), que contém o endereço da instrução sendo executada. O registrador x0 possui o valor zero fixo (gravar dados neste registrador não muda o seu valor) [14].

Embora a maioria dos recursos da arquitetura sejam de implementação opcional, as especificações determinam dois módulos de implementação obrigatória: o conjunto básico de instruções e o modo máquina (M-mode). O modo máquina é o modo mais privilegiado da arquitetura e é o modo em que a implementação deve operar ao ligar. As implementações que disponibilizam apenas este modo podem executar software embarcado e sistemas operacionais de tempo real. A implementação dos demais modos de operação (S- e U-mode) é necessária para execução de sistemas operacionais baseados em Unix.

Existem duas versões ratificadas do conjunto básico de instruções do RISC-V: RV32I e RV64I. A versão RV32I tem tamanho de palavra e espaço de endereçamento de 32 bits. Analogamente, a versão RV64I tem tamanho de palavra e espaço de endereçamento de 64 bits, além de 12 instruções extras para operandos de 32 bits. Há também duas versões não ratificadas, RV32E e RV128I. A primeira reduz o banco de registradores para 16 registradores, com o objetivo de criar implementações menores voltadas a sistemas embarcados. A segunda aumenta o tamanho de palavra dos registradores para 128 bits, possibilitando um espaço de endereçamento maior.

O RISC-V conta com três modos simples de endereçamento: base-deslocamento, absoluto e PC-relativo [15]. O modo base-deslocamento é utilizado pelas instruções de load/store e consiste no uso de um endereço-base (armazenado em registrador) somado a um valor imediato (contido na instrução) para formar o endereço da região de memória que será acessada. Os dois demais modos são utilizados por instruções de desvio para calcular o endereço da próxima

instrução a ser buscada da memória. O modo de endereçamento absoluto utiliza o valor contido em um registrador como endereço, e é usado apenas pela instrução **jalr** (jump and link register). Já o modo de endereçamento PC-relativo utiliza o valor do contador de programa somado a um valor imediato, e é usado pelas demais instruções de desvio.

As instruções do RISC-V são subdivididas em campos, apresentados na Tabela 2. Existem ao todo 6 formatos de instrução, apresentados e descritos nas Tabelas 3 e 4 (p. 18). Os tipos B e J são variantes dos tipos S e U, respectivamente, alternando apenas a posição dos bits que formam o valor do imediato. Os campos **opcode**, **rs1, rs2, rd, funct3** e **funct7** ficam em posições fixas para facilitar a decodificação das instruções.

Tabela 2 – Descrição dos campos das instruções do RISC-V

Campo	Nome	Descrição
rs1	Source register 1	Contém o endereço do primeiro operando da operação.
rs2	Source register 2	Contém o endereço do segundo operando da operação.
rd	Destination register	Contém o endereço do registrador de destino, onde o resultado da operação será salvo.
funct3	3-bit function	Um campo de 3-bits que armazena os códigos de operação para instruções lógicas e aritméticas. Usado também para codificar o tamanho da palavra das instruções load/store.
funct7	7-bit function	Usado apenas com instruções lógicas e aritméticas, contém um código de operação adicional para o controle da ALU.
opcode	Operation code	Contém o código da operação da instrução.

2.3 Organização de computadores RISC

O estudo das formas de implementação em hardware de microprocessadores é chamado de *organização de computadores* [16]. Enquanto a *arquitetura* trata de definir uma interface entre o hardware e o software, a *organização* trata do projeto do hardware que oferece essa interface. É costume utilizar o termo *microarquitetura* para se referir à organização interna do hardware que implementa determinada ISA.

Implementações com microarquiteturas distintas variam quanto às unidades funcionais que as compõem, estrutura interna, técnicas empregadas, etc., razão pela qual variam relativamente à performance e eficiência energética, por exemplo. O objetivo do projeto da microarquitetura pode ser maximizar a performance da implementação para algum propósito, reduzir o consumo de energia, torná-la segura, entre outros. A microarquitetura do Steel, por exemplo, tem o objetivo de torná-lo adequado para uso em sistemas embarcados de pequeno porte.

As seções seguintes exploram alternativas para a implementação de microprocessadores RISC a partir da discussão de três aspectos microarquiteturais de grande relevância. Na seção 2.3.1 são discutidas técnicas de implementação da parte operativa, e na seção 2.3.2 são discutidos o despacho de instruções e a respectiva ordem de execução.

Tabela 3 – Descrição dos formatos de instrução do RISC-V

Formato	Nome	Descrição
R	Register/register	O formato R é utilizado pelas instruções que operam sobre dois registradores (rs1 e rs2). O resultado da operação é armazenado no registrador rd.
I	Immediate	O formato I é utilizado pelas instruções que operam sobre um registrador (rs1) e um valor imediato de 12 bits (imm). O resultado da operação é armazenado no registrador rd.
S	Store	O formato S é utilizado pelas instruções do tipo *store*. O endereço onde o dado será gravado é formado pelo valor de rs1 somado ao imediato imm. O dado gravado está em rs2.
B	Branch	O formato B é utilizado pelas instruções do tipo *branch*. Os valores de rs1 e rs2 são comparados. Se a comparação for verdadeira, o contador de programa é incrementado (ou decrementado) do valor contido em imm.
U	Upper immediate	O formato U é utilizado pelas instruções que operam com imediatos de 20 bits (upper immediates), **lui** e **auipc**.
J	Jump	O formato J é utilizado pela instrução **jal**. O valor imm é somado ao valor do contador de programa, permitindo saltos na faixa de $\pm 2^{20}$ bytes.

2.3.1 Implementações monociclo e *pipelined*

Todo processador funciona repetindo um ciclo que compreende a busca de instrução, sua decodificação e execução. A primeira etapa do ciclo (busca) traz da memória para o processador a instrução contida no endereço indicado pelo contador de programa. A etapa seguinte (decodificação) interpreta a instrução, gera os sinais de controle correspondentes e despacha a instrução para execução na unidade funcional apropriada. A última etapa do ciclo (execução) envolve o cálculo do endereço dos operandos, a busca dos operandos, a seleção da operação apropriada na unidade de execução e a gravação do resultado [17].

Construir um microprocessador consiste basicamente em criar as unidades funcionais

Tabela 4 – Formatos de instrução do RISC-V

Formato	Bits					
	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12 | 10:5]	rs2	rs1	funct3	imm[4:1 | 11]	opcode
U	imm[31:12]				rd	opcode
J	imm[20 | 10:1 | 11 | 19:12]				rd	opcode

que executam as tarefas necessárias para cada etapa e interconectá-las [18]. Os dados manipulados pelas instruções percorrem um caminho ao longo das unidades funcionais do microprocessador, chamado de parte operativa. Em uma implementação *monociclo*, os dados percorrem toda a parte operativa dentro de um único ciclo de relógio, ou seja, as etapas de busca, decodificação e execução ocorrem inteiramente dentro do mesmo ciclo.

Uma implementação monociclo possui a lógica de controle da parte operativa bastante simplificada, mas possui uma desvantagem importante, que limita a frequência máxima de operação do microprocessador: a existência de um longo *caminho crítico*, o maior caminho que os dados percorrem ao longo da parte operativa. A frequência de operação deve ser baixa o suficiente para permitir que os dados percorram inteiramente o caminho crítico antes do início do próximo ciclo de relógio.

Uma das soluções para o problema do caminho crítico é a utilização da técnica de *pipelining*, que consiste em dividir a parte operativa de acordo com as etapas de execução das instruções, colocando registradores ao final de cada etapa para armazenar os resultados parciais e transmiti-los às etapas seguintes. Para exemplificar o uso da técnica, a Fig. 1 (p. 20), extraída do livro de Patterson & Hennessy [19], apresenta a parte operativa de um microprocessador RISC-V com 5 estágios de pipeline. O uso dos registradores ao longo da parte operativa encurta o caminho crítico de forma significativa, permitindo que o processador opere em frequências mais altas. A cada ciclo de relógio o microprocessador executa uma etapa diferente das instruções que ocupam o pipeline. A Fig. 2 (p. 20), extraída do mesmo livro, exemplifica a ocupação do pipeline e a temporização da execução das instruções no processador apresentado na Fig. 1.

A utilização da técnica de *pipelining* permite construir processadores que operam em altas frequências, o que aumenta a quantidade de instruções por segundo (IPS) executadas. Contudo, a quantidade de instruções por ciclo (IPC) continua limitada a 1.0, o valor teórico máximo para um processador que despacha apenas uma instrução de cada vez [20].

2.3.2 Despacho e execução de instruções

É possível classificar os processadores em *single-* e *multiple-issue*, de acordo com a quantidade de instruções que são despachadas para execução a cada ciclo de relógio. Processadores *single-issue* despacham uma única instrução a cada ciclo. Processadores *multiple-issue*, por outro lado, podem despachar mais de uma instrução, permitindo às implementações a obtenção de valores maiores de IPC e, consequentemente, de IPS. Duas técnicas alternativas podem ser empregadas para implementação de processadores *multiple-issue*: despacho múltiplo estático e dinâmico [21].

Processadores com despacho múltiplo estático dependem que o compilador "empacote" instruções para serem executadas em paralelo. Instruções que não utilizam as mesmas unidades funcionais ou que possam ser executadas em paralelo são "empacotadas" juntas, possibilitando uso eficiente dos recursos do hardware. É possível pensar em um "pacote" como uma única instrução, razão pela qual a abordagem de despacho múltiplo estático é chamada de VLIW, Very Long Instruction Word. De outro modo, processadores com despacho múltiplo dinâmico

Figura 1 – Parte operativa de um processador RISC-V com pipeline [19]

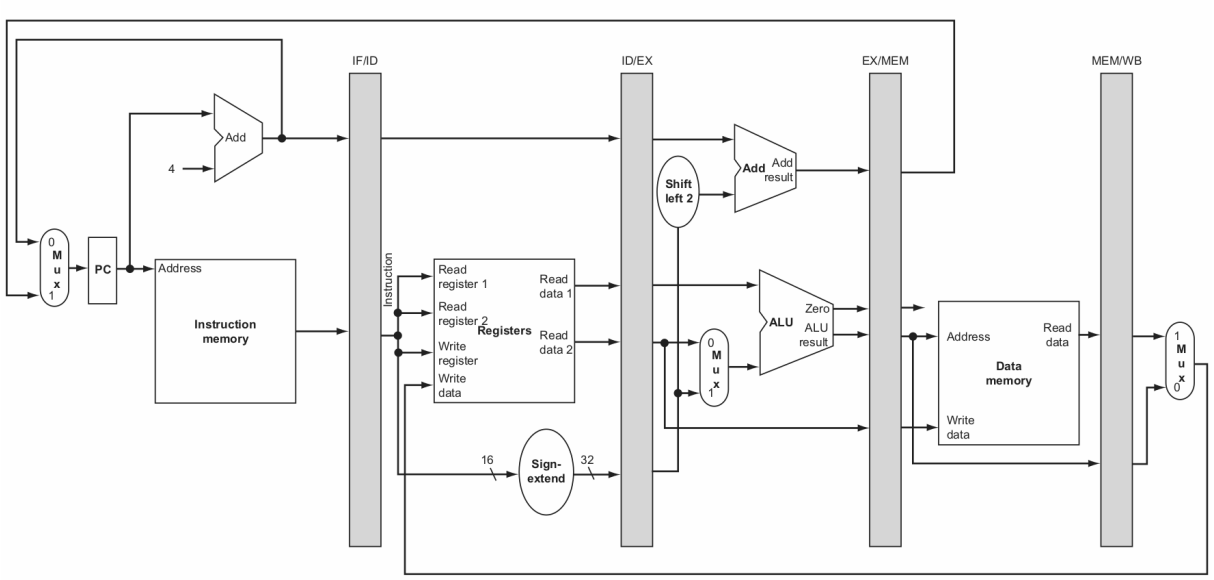
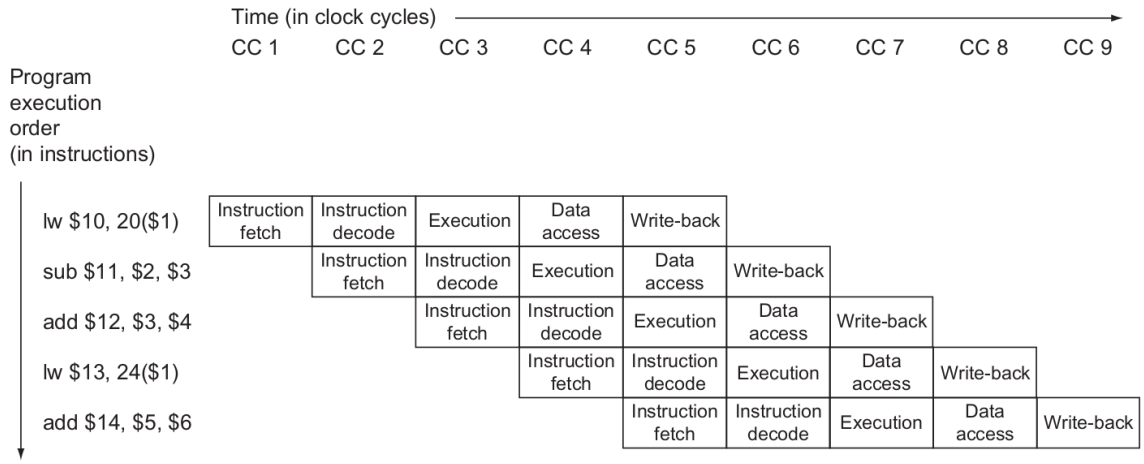


Figura 2 – Execução de instruções no processador da Fig. 1 [19]



utilizam o hardware para identificar as instruções que podem ser executadas em paralelo, em tempo de execução.

Os processadores com despacho múltiplo dinâmico são chamados de *superescalares* e podem executar as instruções em ordem (*in-order execution*) ou fora de ordem (*out-of-order execution*), relativamente à ordem do programa. Num processador superescalar é importante distinguir entre o *despacho* das instruções, que é sempre em ordem; a *execução* das instruções, que pode ser em ordem ou fora de ordem; e entre a entrega dos resultados (*commit*), que também ocorre sempre em ordem [22].

O despacho múltiplo dinâmico com execução em ordem é mais simples de implementar, diferindo da abordagem VLIW apenas pelo momento de identificação das instruções que podem ser executadas em paralelo (tempo de compilação/tempo de execução). A implemen-

tação da execução fora de ordem, por outro lado, requer o emprego de técnicas sofisticadas (e. g. *dynamic scheduling*). Tendo em vista que o Steel é uma implementação *single-issue*, um aprofundamento maior quanto às técnicas de despacho múltiplo está fora do escopo deste trabalho.

3 ESTADO DA ARTE E TRABALHOS RELACIONADOS

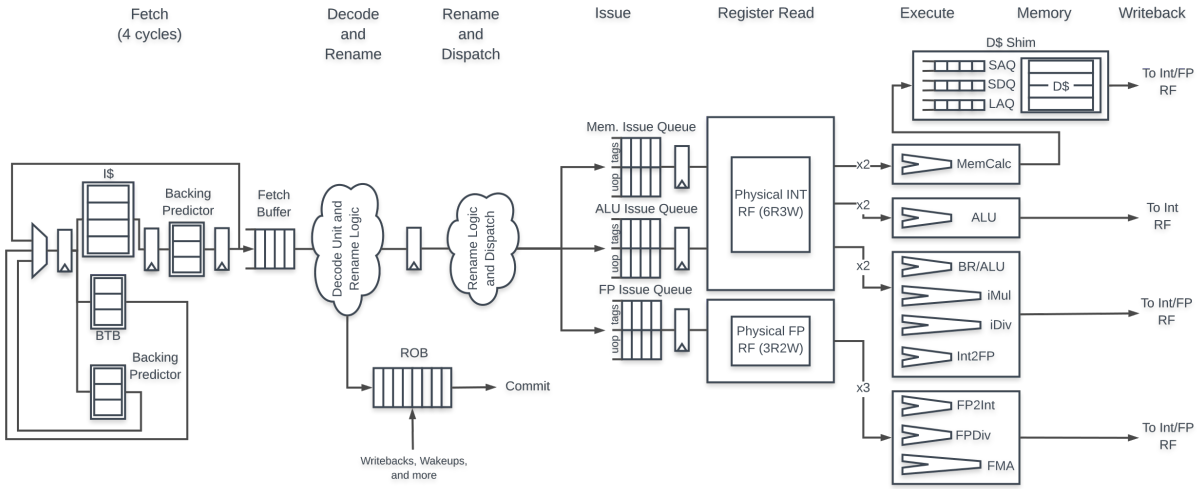
Muitas implementações da ISA do RISC-V estão disponíveis livremente e algumas estão sendo comercializadas. No website da RISC-V International é possível consultar uma lista de cores e *systems-on-chip* atualmente disponíveis [23]. Serão apresentados três projetos/implementações notáveis relacionados ao RISC-V (BOOM, Rocket e PULP). Também serão apresentadas as duas implementações (Ibex e SCR1) com as quais o Steel será comparado neste trabalho.

3.1 Berkeley Out-of-order Machine (BOOM)

BOOM é um processador de código aberto desenvolvido por pesquisadores da University of California, Berkeley, que implementa a ISA RV64G[3] do RISC-V [24]. Como a maior parte dos cores de alta performance, ele é superescalar (capaz de executar mais de uma instrução por ciclo) e *out-of-order* (capaz de executar as instruções enquanto suas dependências são resolvidas, sem ficar restrito à ordem do programa). BOOM cores são produzidos por um gerador parametrizável escrito em Chisel [25] chamado Rocket Chip Generator (apresentado a seguir na seção 3.2), e podem ser utilizados tanto em FPGAs quanto na síntese de ASICs [26].

A versão 2 do BOOM possui um pipeline de 7 estágios e pode executar até quatro instruções paralelamente. Uma visão de alto nível do pipeline, obtida de sua documentação, é apresentada na Fig. 3 [27]. Os cores são capazes de executar sistemas operacionais Linux. Vários parâmetros da implementação são configuráveis, como tamanho das caches, tipos de predição de desvios, etc.

Figura 3 – Visão geral do pipeline de um BOOM Core [27]



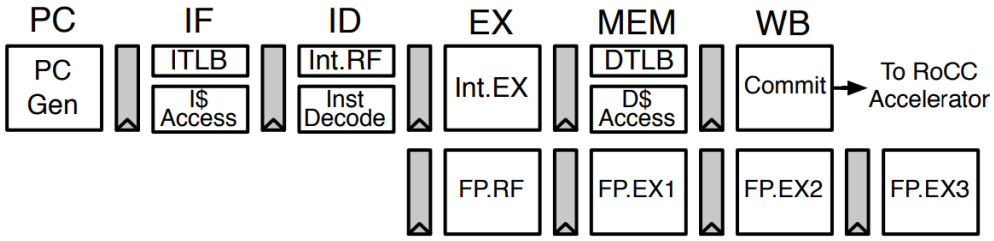
[3]G é uma abreviatura para a implementação conjunta dos módulos I, M, A, F, D, Zicsr e Zifencei do RISC-V, formando um processador de propósito geral.

3.2 Rocket Chip Generator

Rocket Chip Generator é um gerador de SoCs de código aberto desenvolvido na University of California, Berkeley, capaz de produzir SoCs a partir de descrições de alto nível. Diversos parâmetros do SoC gerado podem ser configurados, como o tipo e número de cores e o tamanho e associatividade das caches. Os cores podem ser do tipo Rocket (*in-order*) ou BOOM (*out-of-order*). BOOM Cores são processadores RV64G superescalares (vide seção anterior). Rocket Cores são processadores RV32G/RV64G *single-issue* implementados em um pipeline de 5 estágios (Fig. 4). As extensões M, A, F e D podem ser removidas da implementação opcionalmente. Rocket Cores também possuem MMUs com suporte à memória virtual paginada, caches de dados não bloqueantes e preditores de desvios [28].

O Rocket Chip Generator gera a descrição RTL de um sistema RISC-V completo, composto por cores, caches, unidades de gerenciamento de memória, barramentos e interfaces para comunicação com periféricos. Segundo os autores, o software pode gerar sistemas que variam em tamanho desde pequenos microcontroladores até circuitos integrados com múltiplos cores. Rocket chips já foram manufaturados por pelo menos onze vezes, levando a implementações funcionais e capazes de executar sistemas operacionais Linux.

Figura 4 – Visão geral do pipeline de um Rocket Core



3.3 PULP Platform

PULP (Parallel Ultra Low Power) Platform é um projeto de pesquisa conjunto entre a ETH Zürich e a Università di Bologna dedicado ao desenvolvimento de hardware livre de alta eficiência energética. O projeto disponibiliza um portfólio de cores e SoCs livres, testados em silício e amplamente utilizados pela indústria em projetos IoT e low-power [29].

A plataforma disponibiliza três cores energeticamente eficientes: Ibex, RI5CY e Ariane. Ibex (discutido em detalhe na seção seguinte) é um core RV32IMC *single-issue* com 2 estágios de pipeline. RI5CY é um core RV32IMC *single-issue* com 4 estágios de pipeline e suporte opcional à extensão F. Ariane é a implementação mais sofisticada da plataforma, um core RV64IMAC *single-issue* com 6 estágios de pipeline e suporte aos modos M, S e U da arquitetura privilegiada, capaz de executar sistemas Linux [30].

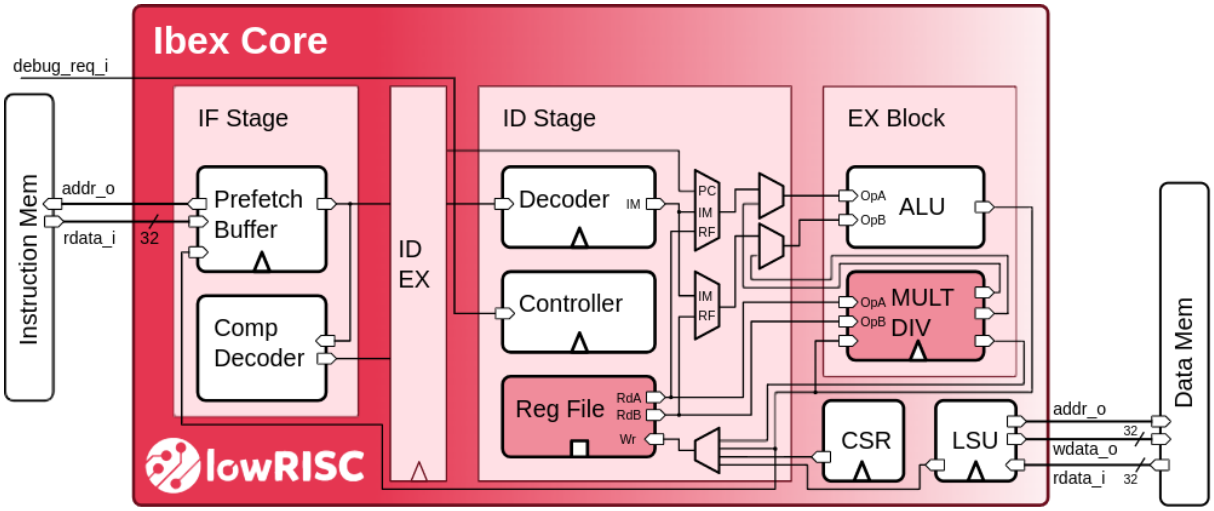
Também são disponibilizados pela plataforma os SoCs PULPino e PULPissimo. PULPino é um microcontrolador *single-core* que pode ser configurado para utilizar um dos cores de 32 bits da plataforma, RI5CY ou Ibex. PULPissimo é um microcontrolador semelhante ao

PULPino energeticamente mais eficiente. Ambos podem ser utilizados em FPGAs e na síntese de ASICs. Os SoCs são voltados a aplicações IoT e possuem interfaces I2S, I2C, CPI, SPI, UART e JTAG para comunicação com dispositivos periféricos.

3.4 Ibex

O Ibex [31] (anteriormente denominado Zero-riscy) é uma implementação RV32IMCZicsr da ISA do RISC-V. Ele possui despacho único de instruções, execução em ordem e suporta os modos M (máquina) e U (usuário) da arquitetura privilegiada. A extensão M, não presente no Steel, pode ser removida opcionalmente. A parte operativa é implementada em 2 estágios de pipeline, com suporte opcional a um terceiro estágio, ainda em fase de testes. Também é possível reduzir a quantidade de registradores, transformando-o em uma implementação RV32E. A Fig. 5 (reproduzida de sua documentação [32]) apresenta uma visão de alto nível de sua microarquitetura.

Figura 5 – Microarquitetura do Ibex Core [32]



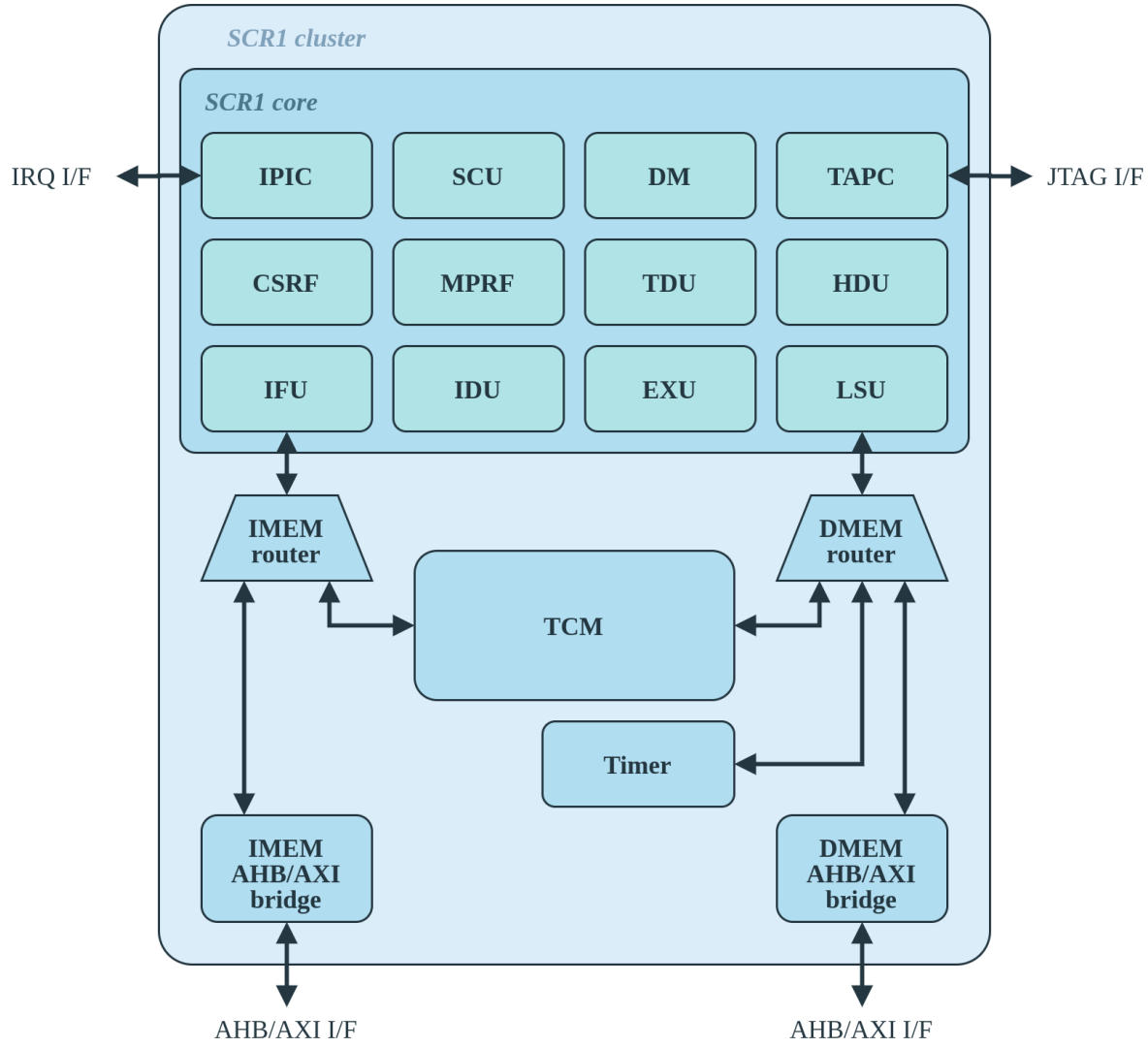
Inicialmente desenvolvido por pesquisadores da ETH Zürich e Università di Bologna, o projeto do Ibex é atualmente mantido pela *lowRISC*, uma companhia sem fins lucrativos voltada ao desenvolvimento colaborativo de hardware livre. O core pode ser utilizado em FPGAs e também na síntese de ASICs, sendo voltado para aplicações *ultra-low-power* e *ultra-low-area*. O Ibex já foi utilizado com sucesso na fabricação de ASICs para projetos *low-power* e IoT.

3.5 SCR1

O SCR1 é um core RV32IZicsr (com suporte opcional para as extensões M e C) produzido pela Syntacore, uma empresa especializada na produção de cores e ferramentas para a arquitetura RISC-V. É o projeto mais simples da companhia e o único de código livre e aberto. Assim como o Ibex, pode ser configurado para ter o banco de registradores reduzido,

transformando-se em uma implementação RV32E. A Fig. 6 (reproduzida de sua documentação [33]) apresenta uma visão de alto nível da microarquitetura do SCR1.

Figura 6 – Microarquitetura do SCR1 [33]



A parte operativa do core pode ser configurada para ter entre 2 e 4 estágios de pipeline. Assim como o Ibex, o SCR1 é voltado para aplicações tanto em FPGAs como em ASICs, tendo sido utilizado com sucesso na fabricação de microcontroladores e SoCs. De acordo com seus desenvolvedores, chips da versão mais simples do core podem ser fabricados utilizando apenas 15 kGates.

4 PROJETO DO STEEL CORE

O Steel é um core com despacho único de instruções, execução em ordem, 3 estágios de pipeline e suporte ao M-mode [9] da arquitetura privilegiada que implementa as ISAs RV32I e Zicsr do RISC-V. Ele é voltado para uso em sistemas embarcados de pequeno porte, sendo capaz de executar software embarcado e sistemas operacionais de tempo real (RTOS). Simplicidade, facilidade de reúso e eficiência foram diretrizes durante todas as etapas de seu projeto.

Este capítulo contém informações detalhadas sobre o projeto do Steel. São apresentadas as ferramentas utilizadas, o fluxo de projeto, a microarquitetura, as interfaces de comunicação do core com outros dispositivos e os diagramas de temporização desses processos. Também são descritos os CSRs implementados, as interrupções e exceções suportadas, os parâmetros que podem ser configurados e as unidades funcionais que compõem a implementação. O capítulo encerra demonstrando um sistema construído com o core para ilustrar sua utilização em outros projetos.

4.1 Ferramentas utilizadas

A microarquitetura do Steel foi projetada com o auxílio do Diagrams.net [34], um *webapp open-source* de uso intuitivo voltado ao desenho de diagramas. Diagramas lógicos da parte operativa, unidades funcionais, interconexões, sinais de controle e interfaces de comunicação foram projetados com o auxílio desta ferramenta.

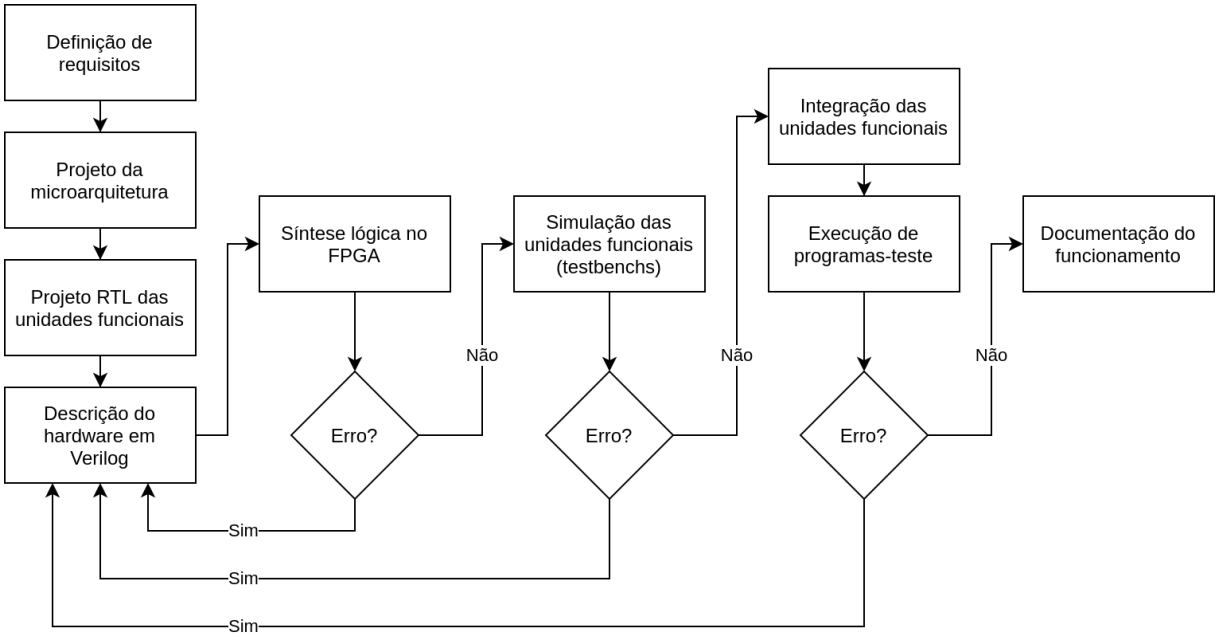
O hardware projetado foi descrito em linguagem Verilog (IEEE Std. 1364-2005 [35]) na última versão (2019.2) do Xilinx Vivado. Com o auxílio do Vivado também foram realizadas simulações de funcionamento de todas as unidades funcionais e testes de integração. A programação do FPGA utilizado (Xilinx Artix-7 XC7A100TCSG324-1) e a extração de relatórios pós-síntese do Steel e de outros cores também foram feitos com o auxílio do Vivado.

Para testar a execução de software (incluindo a suíte de compliance do RISC-V) foram utilizadas diversas ferramentas do RISC-V GNU Toolchain, em especial compiladores e assemblers. Os formatos de onda e os diagramas de temporização da comunicação do core com outros dispositivos foram feitos no WaveDrom, um aplicativo de código aberto que gera waveforms a partir de descrições em JSON (JavaScript Object Notation) [36]. A documentação de todo o projeto foi realizada com o Overleaf, plataforma online para geração de PDFs a partir de documentos LaTeX, e também com o MkDocs, ferramenta para geração automatizada de websites de documentação. Todos os arquivos do projeto foram colocados em um repositório online no GitHub, tornando o projeto acessível ao público em geral.

4.2 Fluxo de projeto

O fluxo de projeto do Steel é apresentado na Fig. 7 (p. 27). O projeto teve início com a definição de quais recursos da arquitetura do RISC-V estariam presentes na implementação. Tendo em vista que o projeto é voltado ao desenvolvimento de sistemas embarcados de pequeno porte, optou-se pela versão RV32I (32 bits) do conjunto básico de instruções. Estes sistemas

Figura 7 – Fluxo de projeto do Steel



executam software inerentemente confiável, dispensando a implementação dos modos S e U da arquitetura privilegiada. No Steel, portanto, optou-se pela implementação apenas do modo M. Por fim, optou-se pela implementação da extensão Zicsr, o que possibilita a desenvolvedores a leitura e gravação de registradores de estado e controle. Isso permite que recursos da arquitetura sejam ativados/desativados via software, tais como interrupções, exceções, contagem de ciclos, de instruções, manipulação de traps, etc.

Após a definição de requisitos, o Diagrams.net foi utilizado para criar um projeto de alto nível do hardware desta implementação. A Fig. 9 (p. 30) apresenta o diagrama da microarquitetura projetada, a ser detalhada na seção seguinte.

A seguir as unidades funcionais previstas pela microarquitetura foram projetadas em detalhes e descritas em linguagem Verilog na última versão do Xilinx Vivado. As unidades funcionais e o módulo principal que integra as unidades, após descritos, foram sintetizados para o FPGA Artix-7 XC7A100TCSG324-1 da Xilinx. A detecção de erros e alertas na etapa de síntese levou à correção da descrição realizada.

Após a ferramenta reportar ausência de erros ou alertas na etapa de síntese, a correta funcionalidade das unidades funcionais foi testada através de simulações. Para testar as unidades foram escritos testes de bancada (*testbenches*) para verificar se o comportamento observado em resposta a estímulos corresponde ao projetado. Os testes foram executados com o auxílio do XSim, simulador integrado do Xilinx Vivado. A detecção de erros nesta etapa também levou à correção da descrição do hardware.

As unidades funcionais foram integradas para compor o core após certificadas quanto à correta funcionalidade. Os testes finais compreenderam a execução da suíte de programas-teste da RISC-V Compliance Suite. A execução dos testes gera uma assinatura, que pode ser

comparada a um padrão de referência disponibilizado pela suíte. A detecção de erros levou à correção da descrição do hardware, até que todas as assinaturas correspondessem às referências.

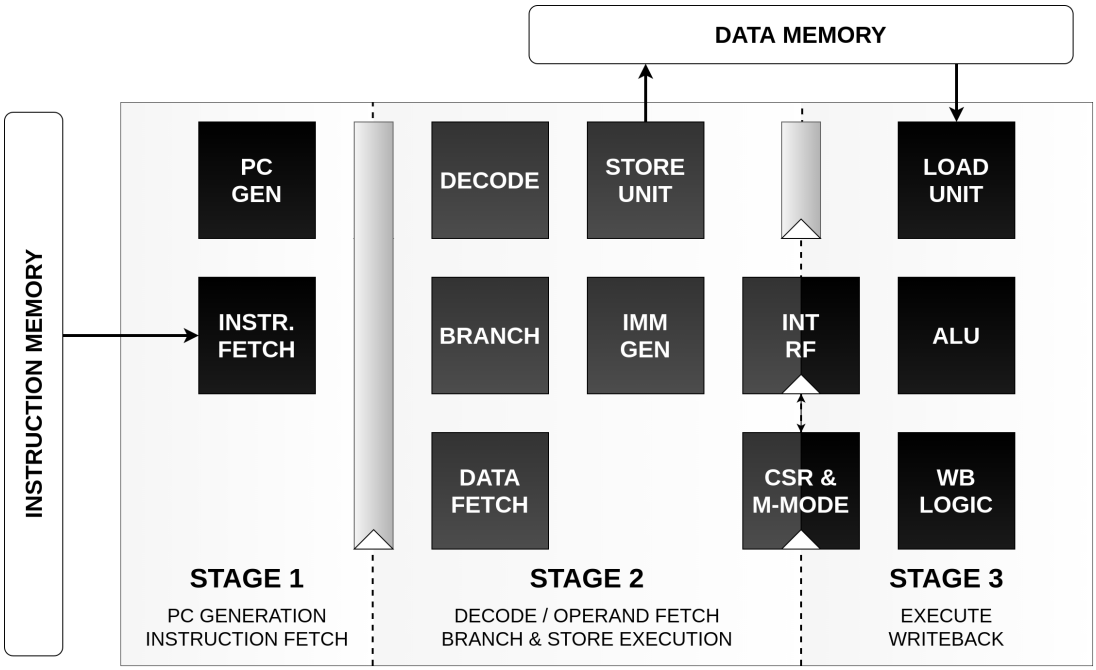
Por fim, o projeto do core foi detalhadamente documentado para possibilitar o reúso por outros desenvolvedores. A documentação foi escrita com o auxílio da plataforma Overleaf, voltada à editoração de documentos LaTeX. Um website da documentação, criado com a ferramenta MkDocs, também foi disponibilizado.

4.3 Microarquitetura

A microarquitetura do Steel foi projetada para que ele seja adequado para uso como unidade de processamento em sistemas embarcados de pequeno porte. Esses sistemas normalmente utilizam processadores *single-core*, possuem pouca capacidade de armazenamento e executam softwares simples dedicados ao controle de poucos periféricos. A frequência de operação das CPUs raramente ultrapassa a faixa de algumas centenas de MHz e a maioria desses sistemas não possui requisitos rígidos de segurança.

A partir desses requisitos foram decididas as principais características da microarquitetura. O pipeline foi projetado com três estágios, possui uma única *thread* e executa as instruções em ordem. Apenas o M-mode da arquitetura privilegiada é suportado, pois espera-se que apenas software confiável seja executado. Essas características tornam o projeto simples e de fácil compreensão, o que facilita o reúso por outros desenvolvedores e possibilita sua expansão futura.

Figura 8 – Pipeline do Steel



A Fig. 8 apresenta uma visão de alto nível da microarquitetura projetada, com ênfase nas tarefas realizadas por cada estágio do pipeline. O primeiro estágio de pipeline é responsável pela geração do valor do contador de programa, que contém o endereço da instrução a ser buscada. O

segundo estágio decodifica a instrução, busca os operandos no Banco de Registradores Inteiros e gera o valor dos imediatos. As instruções do tipo *branch* e *store* são executadas antecipadamente neste estágio. O último estágio executa todas as demais instruções e grava o resultado das operações no banco de registradores. Convém notar que os bancos de registradores (inteiro e CSR) estão posicionados na Fig. 8 em uma posição intermediária entre o segundo e terceiro estágios porque são lidos no segundo estágio (que realiza a busca dos operandos) e escritos no estágio seguinte (que grava o resultado das operações).

A Fig. 9 (p. 30) apresenta um diagrama lógico detalhado da microarquitetura do Steel tal como ela foi descrita em Verilog. O diagrama apresenta as interfaces de comunicação do core com dispositivos externos (detalhadas na seção 4.4), a posição das unidades funcionais dentro do pipeline, as conexões entre as unidades funcionais (detalhadas na seção 4.9) e os sinais de controle da parte operativa.

As tarefas executadas pelo primeiro estágio do pipeline (geração do contador de programa e busca de instrução) são realizadas por um conjunto de multiplexadores controlados por sinais gerados no segundo estágio pelo Decodificador de Instruções e pela Unidade de Controle. Os multiplexadores estão conectados às múltiplas fontes do contador de programa (endereço de *boot*, registradores de estado e controle, somador para cálculo de endereços, etc). O valor do contador de programa gerado é fornecido à memória de instruções através do barramento I_ADDR. A instrução é trazida da memória após um ciclo de relógio através do barramento INSTR.

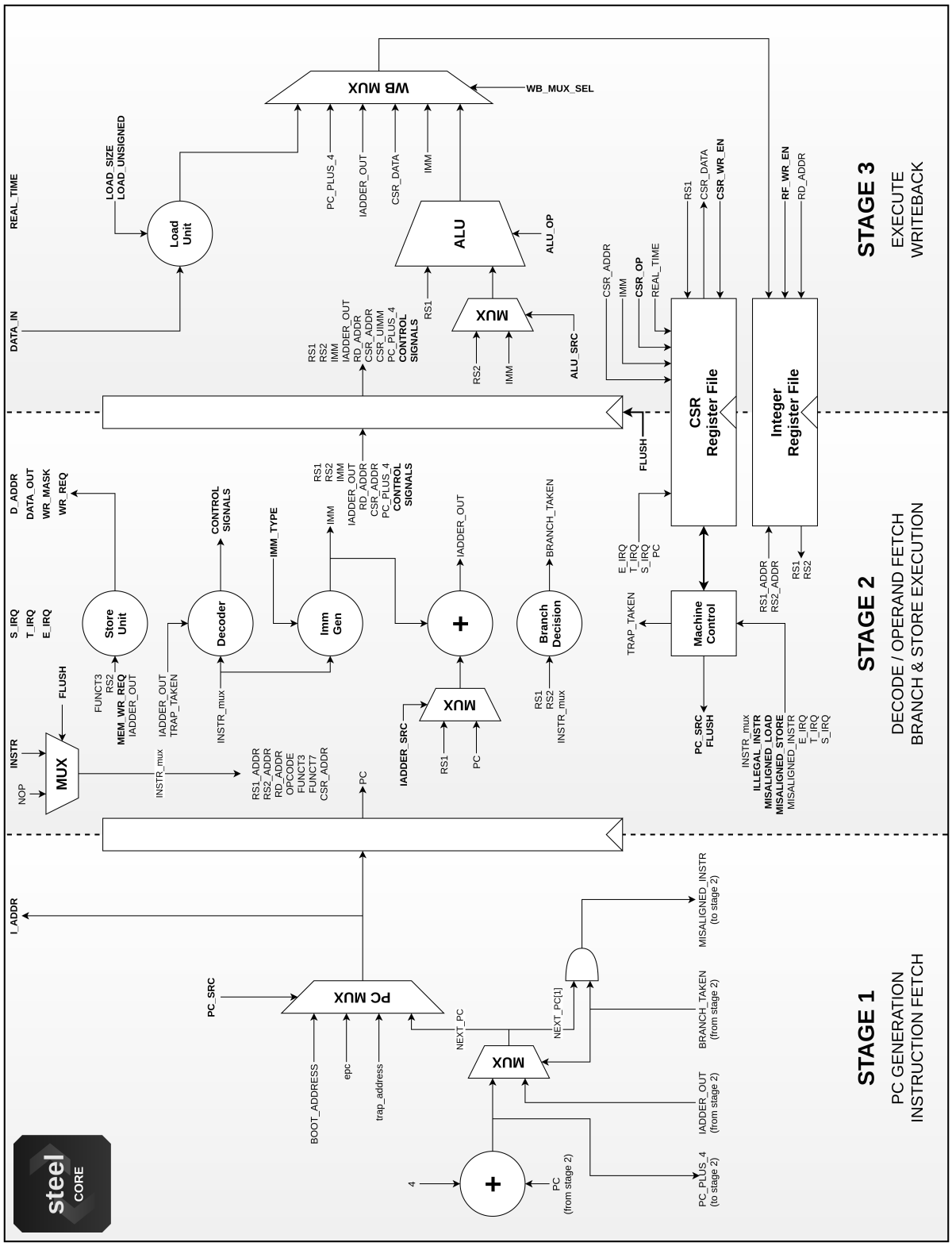
No segundo estágio estão posicionadas as unidades funcionais responsáveis pela decodificação das instruções (Decodificador de Instruções), busca dos operandos (bancos de registradores de inteiros e CSR), geração de imediatos (Gerador de Imediatos) e controle do estado operacional do core (Unidade de Controle). Optou-se por colocar neste estágio, também, as unidades responsáveis pela execução das instruções de desvio (Unidade de Desvios) e do tipo *store* (Unidade de Escrita). A execução antecipada das instruções de desvio no segundo estágio dispensa o uso de preditores de desvio, o que representa uma significativa simplificação ao projeto.

A execução das instruções do tipo *store* no segundo estágio também simplifica o projeto, pois elimina dependências de dados. Uma dependência de dados ocorre quando uma instrução depende do resultado da que a antecedeu. A execução antecipada das instruções *store* no Steel elimina as dependências de dados em relação a eventuais *loads* que sucedam essas instruções, dispensando a construção de circuitos adicionais para detectar e tratar essas dependências.

A Unidade de Controle, posicionada no segundo estágio, está conectada a todas as fontes de interrupções e exceções. Ela é responsável por decidir se as interrupções/exceções serão aceitas (baseado na leitura dos registradores de estado e controle), alterando a geração do contador de programa e limpando o pipeline quando necessário. O banco de registradores de estado e controle possui uma interface especial de comunicação com este módulo.

No terceiro estágio estão posicionadas as unidades funcionais responsáveis pela execução das demais instruções (Unidade de Leitura e ALU). A Unidade de Leitura é responsável por

Figura 9 – Microarquitetura do Steel



estender o sinal do valor lido da memória antes que ele seja gravado no Banco de Registradores Inteiros. A ALU é responsável pela execução das instruções lógicas e aritméticas. Um multiplexador especial, denominado de *Writeback Multiplexer*, está conectado a múltiplas fontes de conteúdo para o banco de registradores e é controlado pelos sinais gerados pelo Decodificador de Instruções no estágio anterior.

4.4 Integração com outros dispositivos

O Steel possui duas interfaces para acesso à memória, uma interface para conexão com um controlador de interrupções e uma interface para conexão com um contador de tempo real. Essas interfaces são apresentadas na Fig. 10 (p. 31). A Tabela 5 (p. 32) contém a descrição de cada sinal que as compõem, sua largura em bits e direção.

Uma das interfaces para acesso à memória é dedicada à busca de instrução, enquanto a outra é voltada à leitura/gravação de dados. As interfaces foram projetadas para conexão a arranjos simples de memória (*memory arrays*) ou Block RAMs *dual-port*, de modo que uma das portas fique ligada à interface para instruções e a outra ligada à interface para dados. Contudo, é possível conectar o core a subsistemas de memória mais complexos, compostos de caches separadas para instruções e dados e múltiplas hierarquias de memória.

A interface de comunicação com um contador de tempo real possui um único sinal de entrada que recebe o valor com o qual o registrador CSR **time** é atualizado. Sistemas que não necessitam de temporizadores podem ligar este sinal a um valor fixo. A interface com o controlador de interrupções possui três sinais de entrada para solicitação de interrupções de três tipos: externas, temporais e de software. Caso alguma dessas interrupções não seja utilizada pela plataforma deve-se conectar o respectivo sinal em nível lógico baixo.

Figura 10 – Interface do Steel

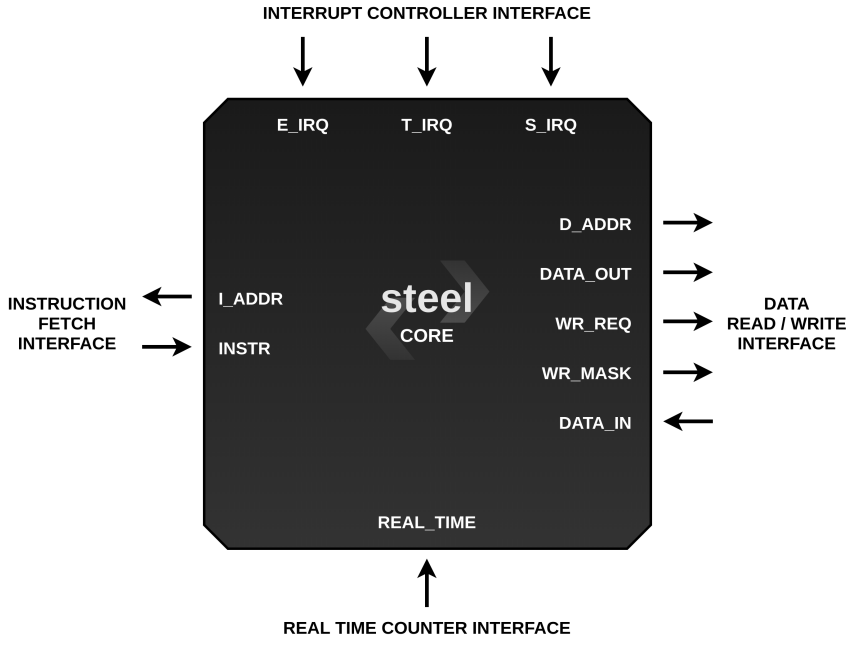


Tabela 5 – Sinais da interface do Steel

Sinal	Largura	Direção	Descrição
REAL_TIME	64 bits	Entrada	Contém o valor atualizado do contador de tempo real.
I_ADDR	32 bits	Saída	Contém o endereço da instrução a ser buscada.
INSTR	32 bits	Entrada	Contém a instrução buscada da memória.
D_ADDR	32 bits	Saída	Contém o endereço de memória do dado a ser lido ou o endereço onde o dado em DATA_OUT será gravado.
DATA_OUT	32 bits	Saída	Contém o dado a ser gravado na memória.
WR_REQ	1 bit	Saída	Sinaliza uma solicitação de escrita quando em nível lógico alto.
WR_MASK	4 bits	Saída	Uma máscara de quatro *write-enable* bits que sinaliza quais bytes de DATA_OUT devem ser escritos na memória. Um bit em nível lógico alto indica que o byte correspondente deve ser gravado.
DATA_IN	32 bits	Entrada	Contém o dado buscado da memória.
E_IRQ	1 bit	Entrada	Sinaliza uma solicitação de interrupção externa quando em nível lógico alto.
T_IRQ	1 bit	Entrada	Sinaliza uma solicitação de interrupção temporal quando em nível lógico alto.
S_IRQ	1 bit	Entrada	Sinaliza uma solicitação de interrupção de software quando em nível lógico alto.

4.5 Diagramas de temporização

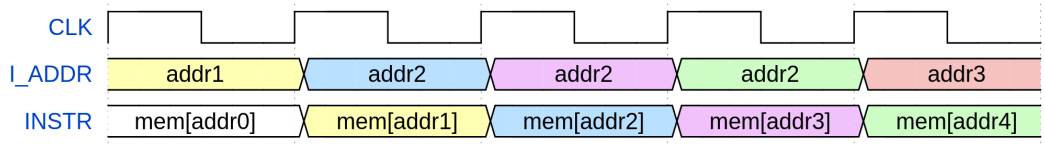
Nesta seção são apresentados os diagramas de temporização dos processos de comunicação do core com dispositivos externos. O core realiza ao todo cinco processos de comunicação distintos: busca de instrução, leitura de dados, gravação de dados, solicitação de interrupção e atualização do contador de tempo real. Os processos de busca de instrução, leitura de dados e gravação de dados ocorrem através das interfaces com a memória, e os processos de solicitação de interrupção e atualização do contador ocorrem através das interfaces com o controlador de interrupções e o contador de tempo real, respectivamente.

4.5.1 Busca de instrução

A Fig. 11 apresenta o diagrama de temporização do processo de busca de instrução. Neste processo, o core coloca no barramento I_ADDR o endereço da instrução que deseja buscar. A memória deve colocar a instrução solicitada no barramento INSTR na próxima borda de subida do relógio. Os dois últimos bits do sinal I_ADDR são sempre zero (o core não realiza

acessos à memória desalinhados).

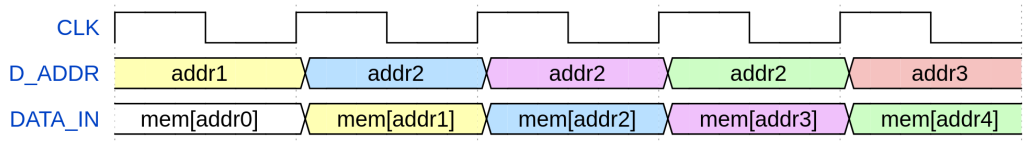
Figura 11 – Diagrama de temporização da busca de instrução



4.5.2 Leitura de dados

A Fig. 12 apresenta o diagrama de temporização do processo de leitura de dados da memória. Neste processo, o core coloca no barramento D_ADDR o endereço do dado que deseja ler. A memória deve colocar o dado solicitado no barramento DATA_IN na próxima borda de subida do relógio. Os dois últimos bits do sinal D_ADDR são sempre zero (o core não realiza acessos à memória desalinhados).

Figura 12 – Diagrama de temporização da leitura de dados



4.5.3 Gravação de dados

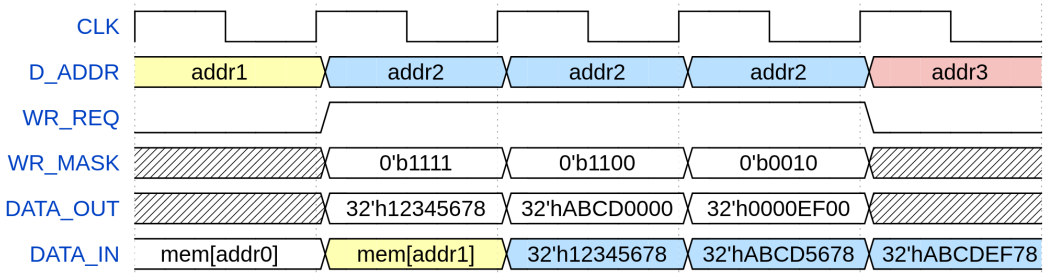
A Fig. 13 apresenta o diagrama de temporização do processo de gravação de dados na memória, no qual são utilizados os sinais D_ADDR, WR_REQ, WR_MASK e DATA_OUT da interface com a memória de dados. O sinal DATA_IN aparece na figura apenas para mostrar o dado lido da memória imediatamente após a gravação.

Para gravar dados na memória, o core coloca o sinal WR_REQ em nível lógico alto, o endereço no barramento D_ADDR e o dado a ser gravado no barramento DATA_OUT, além de ajustar os bits do sinal WR_MASK. O core pode solicitar a gravação de *bytes* (8 bits), *halfwords* (16 bits) ou *words* (32 bits). Assim como no processo de leitura, os dois últimos bits de D_ADDR serão sempre zero, ou seja, os acessos à memória são alinhados em *words*[4]. A gravação de *bytes* e *halfwords* em posições fora deste alinhamento é feita através da configuração apropriada do sinal WR_MASK, que possui quatro *byte-write enable* bits.

A Fig. 13 mostra cinco ciclos de relógio, nos quais o core solicita gravação no segundo, terceiro e quarto ciclos. No segundo ciclo, o core solicita a gravação da *word* 0x12345678 no endereço **addr2**. No terceiro, solicita a gravação da *halfword* 0xABCD na parte superior do mesmo endereço. No quarto, solicita a gravação do *byte* 0xEF no segundo byte menos

[4]Como o Steel foi projetado para uso em FPGAs, espera-se que a memória seja implementada com o uso do Block RAMs. A geração de endereços alinhados em *words* tem o objetivo de facilitar o processo de integração do core com essas memórias.

Figura 13 – Diagrama de temporização da gravação de dados



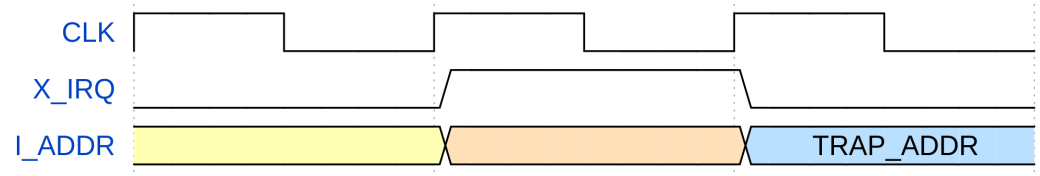
significativo de **addr2**. Observe a mudança do valor armazenado em **addr2** após cada uma dessa operações, apresentado pelo sinal DATA_IN e realçado em azul.

4.5.4 Solicitação de interrupção

A Fig. 14 apresenta o diagrama de temporização do processo de solicitação de interrupção. Como o processo é idêntico para os três tipos de interrupção previstas pela arquitetura do RISC-V (externa, temporal e software), os sinais E_IRQ, T_IRQ e S_IRQ são representados na figura com o nome de X_IRQ. Neste diagrama de temporização é assumido que os bits de *interrupt enable* apropriados estão devidamente setados.

O controlador de interrupções pode solicitar uma interrupção ao core colocando o sinal correspondente (E_IRQ, T_IRQ ou S_IRQ) em nível lógico alto e aguardando a próxima borda de subida do relógio, quando deve colocar o sinal em nível lógico baixo. Se as interrupções estiverem globalmente habilitadas e o tipo de interrupção solicitada também estiver habilitado, o core interromperá a execução da instrução no pipeline e buscará a primeira instrução do tratador de interrupções (indicada na figura por TRAP_ADDR).

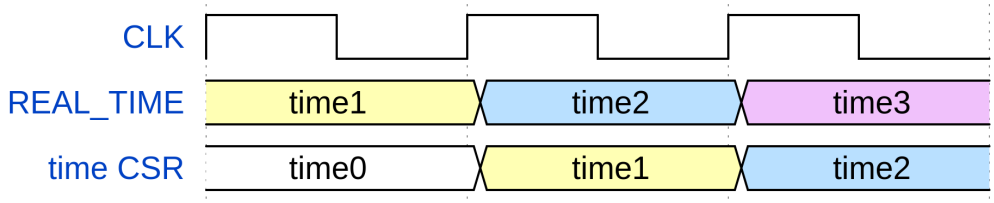
Figura 14 – Diagrama de temporização da solicitação de interrupção



4.5.5 Atualização do contador de tempo real

A Fig. 15 apresenta o diagrama de temporização do processo de atualização do contador de tempo real. A ISA do RISC-V define que cada core deve possuir um registrador CSR chamado **time**, que no Steel é atualizado por um contador de tempo real externo. Esse registrador contém o valor atualizado do tempo transcorrido a partir de um marco temporal arbitrário, medido pelo contador. A atualização do contador é um processo simples: a cada borda de subida do relógio, o registrador CSR **time** é atualizado com o valor lido.

Figura 15 – Diagrama de temporização da atualização do contador de tempo real



4.6 CSRs implementados

A Tabela 6 apresenta a relação completa de registradores de estado e controle implementados no Steel. As funcionalidades ativadas/desativadas através de leitura/gravação nestes registradores estão de acordo com a especificação do M-mode da arquitetura privilegiada do RISC-V. Todas as instruções da extensão Zicsr foram implementadas, possibilitando a atualização destes registradores.

Outros registradores do M-mode que não constam na lista da Tabela 6 (contadores de performance, por exemplo), se lidos, retornarão o valor padrão fixo previsto pelas especificações. Escritas nestes registradores serão ignoradas.

Tabela 6 – CSRs implementados no Steel

CSR	Nome por extenso	Endereço
cycle	Cycle Counter	0xC00
time	System Timer	0xC01
instret	Instructions Retired	0xC02
mstatus	Machine Status	0x300
misa	Machine ISA	0x301
mie	Machine Interrupt Enable	0x304
mtvec	Machine Trap Vector	0x305
mscratch	Machine Scratch	0x340
mepc	Machine Exception Program Counter	0x341
mcause	Machine Cause	0x342
mtval	Machine Trap Value	0x343
mip	Machine Interrupt Pending	0x344
mcycle	Machine Cycle Counter	0xB00
minstret	Machine Instructions Retired	0xB01
mcountinhibit	Machine Counter Inhibit	0x320

4.7 Configuração do core

O Steel possui apenas um parâmetro de configuração, o endereço de *boot*. Ele pode ser definido opcionalmente durante a instanciação do módulo principal do core. Se omitido, o core utiliza como endereço de boot a posição de memória 0x00000000.

4.8 Interrupções e exceções

As interrupções e exceções previstas pela arquitetura RISC-V são resolvidas por um programa armazenado na memória desenvolvido especialmente para tratar interrupções (*trap handler*). O endereço da primeira instrução deste programa é armazenado no registrador CSR **mtvec**, cujo valor pode ser modificado pelo software carregado na memória utilizando as instruções da extensão Zicsr. Quando uma interrupção ou exceção é solicitada, o core pode ou não atendê-la, dependendo da configuração dos registradores **mstatus** e **mie**. Se atendidas, o valor do registrador **mcause** é atualizado para um valor definido nas especificações. A lista de interrupções e exceções suportadas pelo Steel é apresentada na Tabela 7. O Steel procede da seguinte forma ao atender uma interrupção/exceção:

- o endereço da instrução interrompida, ou o endereço da instrução que encontrou a exceção, é salvo no registrador **mepc**;
- o valor do registrador **mtval** é alterado para:
  - o valor do endereço que causou a exceção do tipo *address-misaligned*, ou;
  - zero, nas demais exceções/interrupções;
- o valor do campo MPIE do registrador **mstatus** recebe o valor do campo MIE do mesmo registrador;
- o valor do campo MIE é alterado para zero, desabilitando interrupções;
- o contador de programa é alterado para o endereço da primeira instrução do tratador de interrupções.

A arquitetura RISC-V prevê dois modos de interrupção, direto ou vetorizado. A configuração do modo de interrupção é feita ajustando o campo MODE do registrador **mtvec**. No modo direto, todas as exceções e interrupções aceitas alteram o valor do contador de programa para o valor armazenado no campo BASE do mesmo registrador. No modo vetorizado, as interrupções alteram o contador de programa para BASE + 4 x *exception code* (apresentado na Tabela 7), enquanto exceções são tratadas da mesma forma que no modo direto.

O tratador de interrupções deve encerrar sua execução com a instrução **mret** (trap return from M-mode). Quando esta instrução é executada o core procede da seguinte forma:

- o valor do campo MIE do registrador **mstatus** recebe o valor do campo MPIE do mesmo registrador;
- o valor do campo MPIE é alterado para um;
- o contador de programa é alterado para o endereço salvo no registrador **mepc**.

Tabela 7 – Exceções e interrupções suportadas pelo Steel

	Valor de mcause	
Exceção / Interrupção	**Interrupt bit**	**Exception code**
Machine external interrupt	1	11
Machine software interrupt	1	3
Machine timer interrupt	1	7
Illegal instruction exception	0	2
Instruction address-misaligned exception	0	0
Environment call from M-mode exception	0	11
Environment break exception	0	3
Store address-misaligned exception	0	6
Load address-misaligned exception	0	4

4.9 Unidades funcionais

Esta seção contém descrições detalhadas das unidades funcionais que compõem a implementação do Steel: Decodificador de Instruções, ALU, Gerador de Imediatos, Unidade de Controle, Unidade de Leitura, Unidade de Escrita, Banco de Registradores CSR, Banco de Registradores Inteiros e Unidade de Desvios. Cada unidade funcional foi implementada em um arquivo Verilog distinto. O módulo principal (**steel_top.v**) instancia todas as unidades e as interconecta conforme o diagrama apresentado na Fig. 9 (p. 30).

4.9.1 Decodificador de Instruções

O Decodificador de Instruções (**decoder.v**) decodifica a instrução e gera os sinais que controlam a memória, a ALU, o Gerador de Imediatos, a Unidade de Leitura, a Unidade de Escrita e os dois bancos de registradores (CSR e inteiro). A descrição dos sinais de entrada e saída da unidade são apresentados na Tabela 9 (p. 39).

4.9.2 ALU

A ALU (**alu.v**) aplica dez operações lógicas e aritméticas distintas, em paralelo, sobre dois operandos de 32 bits, entregando o resultado selecionado pelo sinal OPCODE. Os códigos de operação válidos são apresentados na Tabela 8 (p. 38), e os sinais de entrada e saída da ALU na Tabela 10 (p. 40).

Os valores dos códigos de operação foram atribuídos com o objetivo de facilitar a tradução da instrução. O bit mais significativo de OPCODE corresponde ao sexto bit mais significativo do campo **funct7**, enquanto os demais bits correspondem ao campo **funct3**. Assim,

torna-se desnecessário o uso de circuito extra para formação dos códigos de operação durante a etapa de decodificação.

Tabela 8 – Códigos de operação da ALU

Opcode	Operação	Valor binário
ALU_ADD	Adição	4'b0000
ALU_SUB	Subtração	4'b1000
ALU_SLT	Setar se menor que	4'b0010
ALU_SLTU	Setar se menor que sem sinal	4'b0011
ALU_AND	AND lógico bit a bit	4'b0111
ALU_OR	OR lógico bit a bit	4'b0110
ALU_XOR	XOR lógico bit a bit	4'b0100
ALU_SLL	Deslocamento lógico à esquerda	4'b0001
ALU_SRL	Deslocamento lógico à direita	4'b0101
ALU_SRA	Deslocamento aritmético à direita	4'b1101

4.9.3 Banco de Registradores Inteiros

O Banco de Registradores Inteiros (**integer_file.v**) possui 32 registradores de propósito geral e suporta operações de leitura e escrita. A operação de leitura é solicitada pelo segundo estágio do pipeline e pode fornecer o valor de um ou dois registradores. A operação de escrita é solicitada pelo terceiro estágio e coloca no registrador selecionado o valor de saída do multiplexador de *writeback*. Se o terceiro estágio solicita escrita em um registrador sendo lido pelo segundo estágio, o valor sendo escrito é imediatamente encaminhado para leitura. Cada operação é controlada por um grupo próprio de sinais, apresentados nas Tabelas 11 e 12 (p. 40).

4.9.4 Unidade de Desvios

A Unidade de Desvios (**branch_unit.v**) decide se o desvio solicitado por uma instrução do tipo *branch* deve ser tomado ou não. Ela recebe dois operandos do Banco de Registradores Inteiros e decide sobre o desvio baseando-se nos campos **opcode** e **funct3**. As instruções **jal** e **jalr** são tratadas como um tipo especial de *branch* que sempre é tomado. Internamente, a unidade realiza duas comparações, derivando outras quatro a partir delas. A Tabela 13 (p. 41) apresenta os sinais de entrada e saída da unidade.

Tabela 9 – Sinais do Decodificador de Instruções

Nome do sinal	Largura	Direção	Descrição
OPCODE_6_TO_2	5 bits	Entrada	Conectado ao campo **opcode** da instrução.
FUNCT7_5	1 bit	Entrada	Conectado ao campo **funct7** da instrução.
FUNCT3	3 bits	Entrada	Conectado ao campo **funct3** da instrução.
IADDER_OUT_1_TO_0	2 bits	Entrada	Usado para verificar o alinhamento dos endereços de *loads* e *stores*.
TRAP_TAKEN	1 bits	Entrada	Indica quando uma trap será tomada.
ALU_OPCODE	4 bits	Saída	Seleciona a operação a ser realizada pela ALU. Ver Tabela 8.
MEM_WR_REQ	1 bit	Saída	Indica uma solicitação para gravar dados na memória.
LOAD_SIZE	2 bits	Saída	Sinaliza o tipo de instrução de leitura.
LOAD_UNSIGNED	1 bit	Saída	Sinaliza o tipo de instrução de leitura.
ALU_SRC	1 bit	Saída	Controla o multiplexador que seleciona o segundo operando da ALU.
IADDER_SRC	1 bit	Saída	Controla o multiplexador que seleciona o operando do *Immediate Adder*.
CSR_WR_EN	1 bit	Saída	Controla a entrada WR_EN do Banco de Registradores CSR.
RF_WR_EN	1 bit	Saída	Controla a entrada WR_EN do Banco de Registradores Inteiros. Veja Tabela 12.
WB_MUX_SEL	3 bits	Saída	Controla o multiplexador de *writeback*.
IMM_TYPE	3 bits	Saída	Controla o Gerador de Imediatos.
CSR_OP	3 bits	Saída	Seleciona a operação a ser realizada pelo Banco de Registradores CSR.
ILLEGAL_INSTR	1 bit	Saída	Indica que uma instrução não implementada ou inválida foi buscada da memória.
MISALIGNED_LOAD	1 bit	Saída	Indica uma tentativa de leitura da memória fora das regras de alinhamento.
MISALIGNED_STORE	1 bit	Saída	Indica uma tentativa de gravação na memória fora das regras de alinhamento.

4.9.5 Unidade de Leitura

A Unidade de Leitura (**load_unit.v**) recebe o valor de 32 bits lido da memória e a partir dele forma o valor correto a ser gravado no Banco de Registradores Inteiros, de acordo com o tipo da instrução de *load* (indicado no campo **funct3** da instrução). Os sinais de entrada e saída

Tabela 10 – Sinais da ALU

Nome do sinal	Largura	Direção	Descrição
OP_1	32 bits	Entrada	Primeiro operando da operação.
OP_2	32 bits	Entrada	Segundo operando da operação.
OPCODE	4 bits	Entrada	Código da operação. Controlado pelos campos **funct7** e **funct3** da instrução.
RESULT	32 bits	Saída	Resultado da operação selecionada.

Tabela 11 – Sinais do Banco de Registradores Inteiros para a operação de leitura

Nome do sinal	Largura	Direção	Descrição
RS_1_ADDR	5 bits	Entrada	Endereço do registrador fonte 1. O dado será colocado no barramento RS_1 imediatamente.
RS_2_ADDR	5 bits	Entrada	Endereço do registrador fonte 2. O dado será colocado no barramento RS_2 imediatamente.
RS_1	32 bits	Saída	Dado lido da fonte 1.
RS_2	32 bits	Saída	Dado lido da fonte 2.

Tabela 12 – Sinais do Banco de Registradores Inteiros para a operação de escrita

Nome do sinal	Largura	Direção	Descrição
RD_ADDR	5 bits	Entrada	Endereço do registrador de destino.
RD	32 bits	Entrada	Dado a ser escrito no registrador endereçado por RD_ADDR.
WR_EN	1 bit	Entrada	*Write enable*. Quando em nível lógico alto, o dado localizado em RD é escrito no registrador endereçado por RD_ADDR na próxima borda de subida do relógio.

da unidade são apresentados na Tabela 14 (p. 41).

4.9.6 Unidade de Escrita

A Unidade de Escrita (**store_unit.v**) é a unidade responsável por controlar os sinais da interface com a memória de dados. Ela coloca o dado a ser gravado (que pode ser um *byte*, *halfword* ou *word*) na posição correta em DATA_OUT e ajusta o valor de WR_MASK de forma apropriada. A Tabela 15 (p. 42) apresenta os sinais de entrada e saída da unidade.

Tabela 13 – Sinais da Unidade de Desvios

Nome do sinal	Largura	Direção	Descrição
OPCODE_6_TO_2	5 bits	Entrada	Conectado ao campo **opcode** da instrução.
FUNCT3	3 bits	Entrada	Conectado ao campo **funct3** da instrução.
RS1	32 bits	Entrada	Conectado ao primeiro operando selecionado do Banco de Registradores Inteiros.
RS2	32 bits	Entrada	Conectado ao segundo operando selecionado do Banco de Registradores Inteiros.
BRANCH_TAKEN	1 bits	Saída	HIGH se o desvio deve ser tomado, LOW caso contrário.

Tabela 14 – Sinais da Unidade de Leitura

Nome do sinal	Largura	Direção	Descrição
LOAD_SIZE	2 bits	Entrada	Conectado aos dois bits menos significativos do campo **funct3** da instrução.
LOAD_UNSIGNED	1 bit	Entrada	Conectado ao bit mais significativo do campo **funct3** da instrução.
DATA_IN	32 bits	Entrada	Palavra de 32 bits lida do endereço de memória endereçado por D_ADDR.
IADDER_OUT_1_TO_0	2 bits	Entrada	Indica a posição do *byte* ou *halfword* dentro de DATA_IN.
OUTPUT	32 bits	Saída	Valor formado que será escrito no Banco de Registradores Inteiros.

4.9.7 Banco de Registradores CSR

O Banco de Registradores CSR (**csr_file.v**) contém os registradores de estado e controle necessários para a implementação do M-mode (apresentados na Tabela 6, p. 35). As operações *read/write*, *set* e *clear*, necessárias para a implementação das instruções Zicsr, podem ser aplicadas sobre os registradores. A Tabela 16 (p. 43) apresenta os sinais de entrada e saída da unidade, com exceção dos sinais utilizados para comunicação com a Unidade de Controle, que são apresentados na Tabela 17 (p. 44).

4.9.8 Gerador de Imediatos

O Gerador de Imediatos (**imm_generator.v**) reorganiza os bits do imediato contido na instrução e, quando necessário, estende o sinal, formando um valor de 32 bits. A unidade é controlada pelo sinal IMM_TYPE, gerado pelo Decodificador de Instruções. A Tabela 18 (p. 42) apresenta os sinais de entrada e saída da unidade.

Tabela 15 – Sinais da Unidade de Escrita

Nome do sinal	Largura	Direção	Descrição
FUNCT3	3 bits	Entrada	Indica o tamanho do dado a ser gravado (byte, halfword ou word).
IADDER_OUT	32 bits	Entrada	Contém o endereço onde o dado será gravado (desalinhado).
RS2	32 bits	Entrada	Conectado ao Banco de Registradores Inteiros, contém o dado a ser gravado na memória.
MEM_WR_REQ	1 bit	Entrada	Sinal gerado pelo Decodificador de Instruções que indica uma solicitação para escrita na memória.
DATA_OUT	32 bits	Saída	Contém o valor a ser gravado na posição correta.
D_ADDR	32 bits	Saída	Endereço onde o dado será gravado (alinhado).
WR_MASK	4 bits	Saída	Máscara de *byte-write enable* bits, indicando quais bytes de DATA_OUT devem ser gravados.
WR_REQ	1 bit	Saída	Sinaliza à memória uma solicitação de escrita.

Tabela 18 – Sinais do Gerador de Imediatos

Nome do sinal	Largura	Direção	Descrição
INSTR	25 bits	Entrada	Conectado aos bits de imediato da instrução (32 a 7).
IMM_TYPE	2 bits	Entrada	Sinal de controle que indica o tipo de imediato que deve ser gerado.
IMM	32 bits	Saída	Imediato de 32 bits gerado.

4.9.9 Unidade de Controle

A Unidade de Controle (**machine_control.v**) implementa o M-mode da arquitetura privilegiada, controlando a geração do contador de programa e atualizando diversos registradores de estado e controle. A unidade possui uma interface especial de comunicação com o Banco de Registradores CSR, apresentada na Tabela 17 (p. 44). Seus sinais de entrada e saída são apresentados na Tabela 19 (p. 45).

Internamente, a unidade implementa a FSM (*Finite State Machine*, máquina de estados finitos) apresentada na Fig. 16 (p. 46). O core entra sincronamente em seu estado inicial (*Reset State*) quando o sinal RESET é setado, e se mantém neste estado até que o sinal seja posto em nível lógico baixo. Neste estado, o core limpa o pipeline, seta o contador de programa para o endereço de *boot* e desativa a atualização do contador de instruções finalizadas.

Tabela 16 – Sinais do Banco de Registradores CSR

Nome do sinal	Largura	Direção	Descrição
WR_EN	1 bit	Entrada	*Write enable*. Quando setado, atualiza o valor do CSR endereçado por CSR_ADDR na próxima borda de subida do relógio, de acordo com a operação realizada.
CSR_ADDR	12 bits	Entrada	Endereço do CSR a ler/gravar/modificar.
CSR_OP	3 bits	Entrada	Determina a operação a ser realizada, de acordo com o campo **funct3** da instrução Zicsr.
CSR_UIMM	5 bits	Entrada	Conectado aos 5 bits menos significativos do sinal de saída do Gerador de Imediatos.
CSR_DATA_IN	32 bits	Entrada	Para operação de escrita, contém o dado a ser gravado. Para operações *set/clear*, contém uma máscara de bits.
PC	32	Entrada	Atualiza o valor do CSR **mepc**.
E_IRQ	1 bit	Entrada	*External interrupt request.* Atualiza o valor do CSR **mip.MEIP**.
T_IRQ	1 bit	Entrada	*Timer interrupt request.* Atualiza o valor do CSR **mip.MTIP**.
S_IRQ	1 bit	Entrada	*Software interrupt request.* Atualiza o valor do CSR **mip.MSIP**.
REAL_TIME	64 bits	Entrada	Sinal usado para atualizar o valor dos CSRs **time** e **timeh**.
CSR_DATA_OUT	32 bits	Saída	Contém o dado lido do CSR endereçado por CSR_ADDR.
EPC_OUT	32 bits	Saída	Contém o valor atual do registrador **mepc**.
TRAP_ADDRESS	32 bits	Saída	Contém o endereço da primeira instrução do tratador de interrupções.

No estado operacional (*Operating State*), o core habilita o incremento do contador de instruções finalizadas e seta o contador de programa para o valor do sinal NEXT_PC. Neste estado, o core pode fazer uma transição para o estado *Trap Return*, se a instrução **mret** for executada, ou para o estado *Trap Prepare*, se interrupções/exceções forem tomadas. Nesses estados, o core procede da forma detalhada na seção 4.8 (p. 36). Além disso, o pipeline é limpo e a atualização do contador de instruções finalizadas é desabilitado.

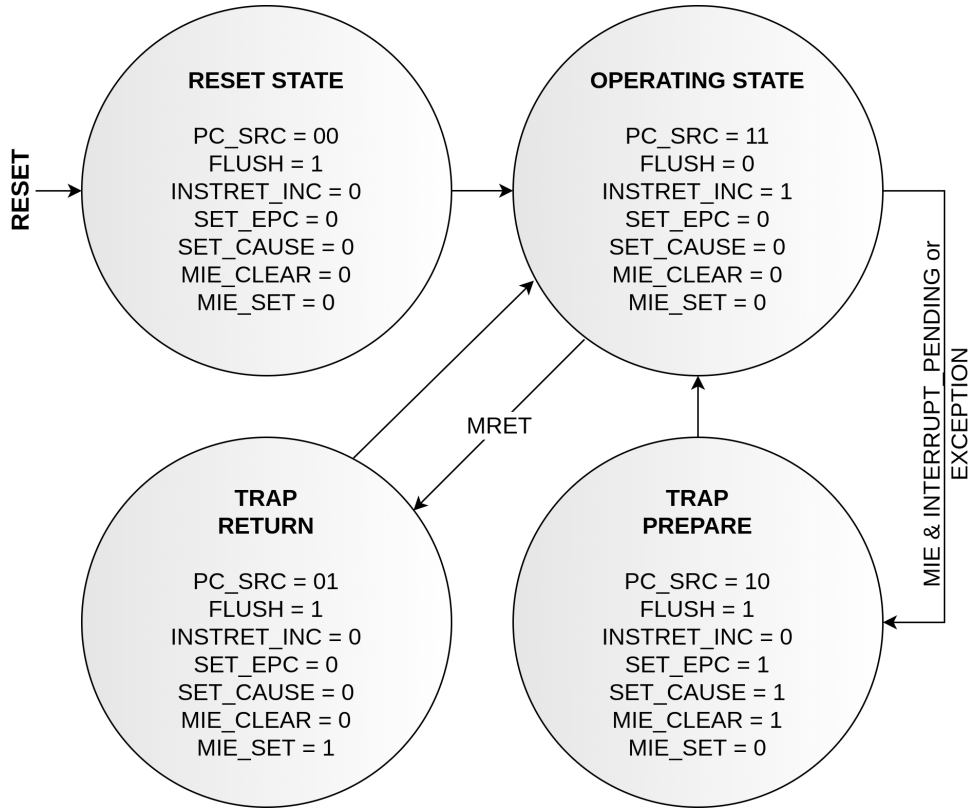
Tabela 17 – Sinais da interface entre os registradores CSR e a Unidade de Controle

Nome do sinal	Largura	Direção	Descrição
I_OR_E	1 bit	Entrada	*Interrupt or exception*. Quando setado, indica uma interrupção, e uma exceção caso contrário. É utilizado para alterar o valor do bit mais significativo do registrador **mcause**.
CAUSE_IN	4 bits	Entrada	Contém o código que indica a causa da interrupção ou exceção. Ver Tabela 7.
SET_CAUSE	1 bit	Entrada	Quando setado, atualiza o valor do registrador **mcause** com os valores de I_OR_E e CAUSE_IN.
SET_EPC	1 bit	Entrada	Quando setado, atualiza o valor do registrador **mepc** com o valor de PC.
INSTRET_INC	1 bit	Entrada	Quando setado, habilita o incremento do contador **minstret**.
MIE_CLEAR	1 bit	Entrada	Quando setado, escreve 1'b0 no registrador **mstatus.MIE** (desabilita interrupções globalmente). O valor antigo de **mstatus.MIE** é salvo no registrador **mstatus.MPIE**.
MIE_SET	1 bit	Entrada	Quando setado, escreve 1'b1 no registrador **mstatus.MPIE**. O valor anterior de **mstatus.MPIE** é salvo em **mstatus.MIE**.
MIE	1 bit	Saída	Valor atual do registrador **mstatus.MIE**.
MEIE_OUT	1 bit	Saída	Valor atual do registrador **mie.MEIE**.
MTIE_OUT	1 bit	Saída	Valor atual do registrador **mie.MTIE**.
MSIE_OUT	1 bit	Saída	Valor atual do registrador **mie.MSIE**.
MEIP_OUT	1 bit	Saída	Valor atual do registrador **mip.MEIP**.
MTIP_OUT	1 bit	Saída	Valor atual do registrador **mip.MTIP**.
MSIP_OUT	1 bit	Saída	Valor atual do registrador **mip.MSIP**.

Tabela 19 – Sinais da Unidade de Controle

Nome do sinal	Largura	Direção	Descrição
ILLEGAL_INSTR	1 bit	Entrada	*Illegal instruction*. Quando setado, indica que uma instrução desconhecida ou não implementada foi buscada da memória.
MISALIGNED_INSTR	1 bit	Entrada	*Misaligned instruction*. Quando setado, indica a tentativa de buscar da memória uma instrução cujo endereço está fora de alinhamento.
MISALIGNED_LOAD	1 bit	Entrada	*Misaligned load*. Quando setado, indica a tentativa de ler da memória um dado cujo endereço está fora de alinhamento.
MISALIGNED_STORE	1 bit	Entrada	*Misaligned store*. Quando setado, indica a tentativa de gravar na memória um dado cujo endereço está fora de alinhamento.
OPCODE_6_TO_2	5 bits	Entrada	Campo **opcode** da instrução sendo executada.
FUNCT3	3 bits	Entrada	Campo **funct3** da instrução sendo executada.
FUNCT7	7 bits	Entrada	Campo **funct7** da instrução sendo executada.
RS1_ADDR	5 bits	Entrada	Campo rs1 da instrução sendo executada.
RS2_ADDR	5 bits	Entrada	Campo rs2 da instrução sendo executada.
RD_ADDR	5 bits	Entrada	Campo rd da instrução sendo executada.
E_IRQ	1 bit	Entrada	*External interrupt request*. Ligado ao controlador de interrupções.
T_IRQ	1 bit	Entrada	*Timer interrupt request*. Ligado ao controlador de interrupções.
S_IRQ	1 bit	Entrada	*Software interrupt request*. Ligado ao controlador de interrupções.
PC_SRC	2 bit	Saída	Controla o multiplexador do contador de programa.
FLUSH	1 bit	Saída	Limpa o pipeline quando setado.
TRAP_TAKEN	1 bit	Saída	Quando setado, indica que uma trap será tomada no próximo ciclo de clock.

Figura 16 – Máquina de estados do M-mode



4.10 Sistema exemplo construído com o Steel

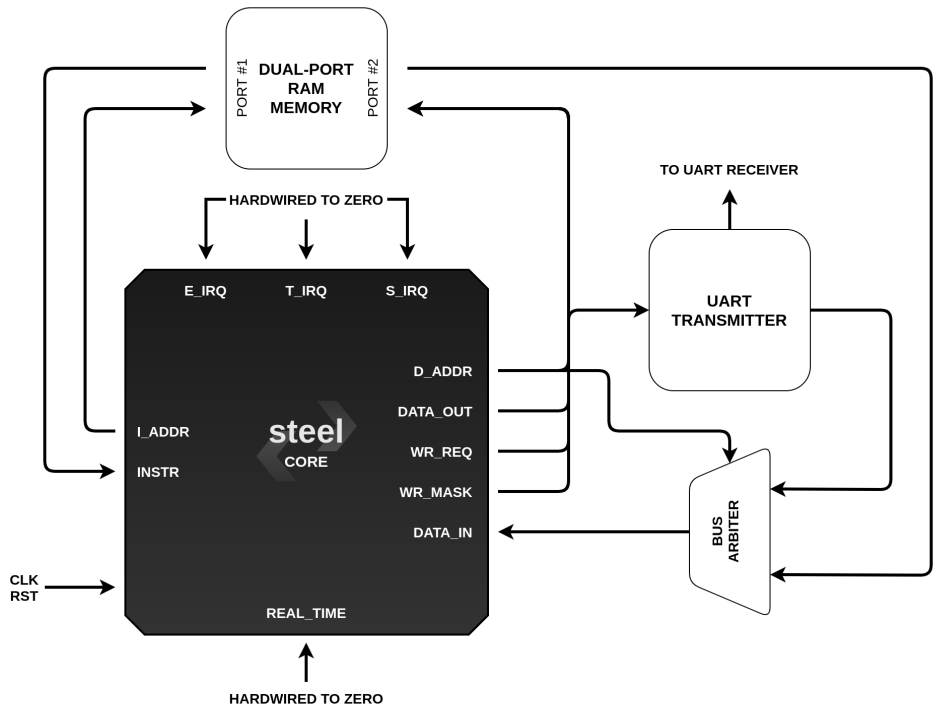
Para demonstrar como o Steel pode ser utilizado em um sistema embarcado de pequeno porte, um sistema-exemplo simples é apresentado na Fig. 17. Ele é formado por uma unidade de memória RAM, um árbitro de barramento, um transmissor UART e, logicamente, pelo Steel. O sinal do contador de tempo real foi ligado a um valor fixo, assim como os sinais de solicitação de interrupção, pois neste sistema esses recursos não são utilizados. A figura mostra uma visão de alto nível da arquitetura deste sistema.

No exemplo, a comunicação do core com o transmissor UART é realizada através da técnica de mapeamento em memória, que consiste em dividir o espaço de endereçamento em faixas e atribuir cada faixa a um componente. Para implementá-la no sistema, um árbitro de barramento multiplexa os sinais da interface de dados de acordo com o endereço acessado pelo core. Para transmitir dados pela UART e verificar se a unidade está pronta o core lê e grava dados em endereços de memória predeterminados.

Outros sistemas, com diferentes interfaces e periféricos, podem ser construídos utilizando uma arquitetura semelhante. Além disso, outras interfaces de comunicação, como I2C, I2S e SPI podem ser adicionadas, possibilitando a comunicação com diferentes tipos de dispositivos externos.

Como qualquer outro sistema RISC-V, é possível desenvolver software para este sistema exemplo com o auxílio do RISC-V GNU Toolchain [37]. O software desenvolvido pode ser carregado na memória do sistema com o auxílio das ferramentas de programação do FPGA (por exemplo, Xilinx Vivado ou Intel Quartus).

Figura 17 – Sistema exemplo construído com o Steel



5 RESULTADOS

Neste capítulo são apresentados os resultados da síntese e implementação do Steel em um FPGA Xilinx Artix-7 XC7A100TCSG324-1. Os dados de utilização dos recursos do FPGA reportados após a síntese são tabulados e comparados com dados da síntese de outros dois cores, Ibex e SCR1, apresentados no capítulo 3. Esses cores foram escolhidos por serem implementações similares ao Steel amplamente testadas e utilizadas em outros projetos. Ao final, também são apresentados os resultados da aplicação dos testes da suíte de compliance do RISC-V e do benchmark CoreMark.

5.1 Comparação entre o Steel, Ibex e SCR1

Para realizar a comparação com o Steel, os arquivos mais recentes da implementação dos cores Ibex e SCR1 foram obtidos de seus respectivos repositórios na internet [38], [39]. Em seguida, foram criados projetos para esses cores no Xilinx Vivado. Os projetos foram configurados para sintetizar a descrição do hardware para o FPGA Artix-7 XC7A100TCSG324-1, um FPGA de pequeno porte que conta com 15.850 logic slices. Cada logic slice é equipado com 4 look-up tables de 6 entradas (6-LUTs) e 8 flip-flops. O FPGA também conta com 240 DSP slices (unidades para processamento de sinais), 4.860 Kb em blocos de memória RAM e 1 conversor digital-analógico. As LUTs de alguns logic slices podem ser configuradas para funcionar como memória RAM distribuída (LUT RAM).

Antes de sintetizados, o Ibex e SCR1 foram configurados para suas versões mais simples, a fim de tornarem-se implementações mais próximas ao Steel. No Ibex, a extensão M foi desabilitada. No SCR1, as extensões M e C foram removidas, além de todos os componentes *uncore* (caches, controladores de interrupção, etc).

Os dados do relatório gerado pelo Vivado após a síntese de cada core foram resumidos e estão apresentados na Tabela 20. É importante ressaltar que a comparação apresentada não implica que uma das implementações é superior às demais, tampouco que a comparação é justa. A arquitetura RISC-V é modular e flexível, com muitas possibilidades de *tradeoff* mesmo quando os recursos da arquitetura implementados sejam os mesmos.

Tabela 20 – Quadro comparativo do uso de recursos do Steel, Ibex e SCR1

Recurso/característica	**Steel**	**Ibex**	**SCR1**
6-LUTs	1.626	2.010 (+24%)	2.359 (+45%)
LUT RAMs	48	48	0
Flip-flops	624	790 (+27%)	1.392 (+123%)
Conjunto de Instruções	RV32I + Zicsr	RV32I + C + Zicsr	RV32I + Zicsr
Modos suportados	M	M e U	M

A implementação do Steel é a que utiliza o menor número de 6-LUTs e flip-flops quando

sintetizada no FPGA utilizado. Além de economizar recursos do FPGA, deixando-os livres para implementação de outras funções do sistema, o Steel é o core que possivelmente ocuparia a menor área em um ASIC.

É preciso observar, no entanto, que a síntese do Ibex realizada incluiu o suporte a instruções comprimidas (extensão C) e ao modo U da arquitetura privilegiada, ambos não presentes no Steel. Como esses recursos exigem unidades de hardware especiais para sua respectiva implementação, isso justifica a maior quantidade de recursos consumidas pelo Ibex. A síntese exclusiva do módulo descompressor de instruções do Ibex (um módulo puramente combinacional) utiliza ao total 146 LUTs. A implementação do modo U está distribuída entre diversas unidades, de modo que não é possível obter métricas de síntese exclusivas para este recurso. Assim, é possível que a quantidade maior de recursos consumidas pelo Ibex se deva exclusivamente à implementação dessas funcionalidades.

A síntese do SCR1, por outro lado, incluiu os mesmos módulos da ISA do RISC-V (RV32I, Zicsr e modo M) presentes no Steel. É possível afirmar, portanto, que o Steel é uma implementação que consome menos recursos e que implementa o mesmo conjunto de instruções que a versão RV32IZicsr do SCR1. O número significativamente maior de flip-flops utilizado pelo SCR1 deve-se em parte à opção pela implementações do banco de registradores com flip-flops, diferentemente do Steel e do Ibex, que utilizam a memória distribuída do FPGA (LUT RAMs) para implementá-lo. Para uma mesma quantidade de bits de memória, a implementação com LUT RAMs utiliza menos recursos em comparação com a implementação por flip-flops.

5.2 Testes de compliance

O Steel foi testado em relação a sua correta funcionalidade e aderência às especificações do RISC-V pela aplicação dos testes de compliance da RISC-V Compliance Suite, uma suíte de testes desenvolvida e distribuída pela RISC-V International. A aplicação dos testes com sucesso permite afirmar que a implementação "interpreta corretamente as especificações e que o dispositivo sob teste pode ser declarado *RISC-V compliant*"[40].

Os testes estão agrupados de acordo com os módulos da arquitetura e podem ser aplicados em grupo ou individualmente. Cada um dos testes da suíte avalia o funcionamento de algum recurso, como uma instrução, um tipo de exceção, etc. Após a execução de cada teste é gerada uma assinatura, que deve ser comparada ao padrão de referência (*golden model*). Atualmente, esse padrão é obtido através da execução da suíte de testes em um simulador da ISA do RISC-V. A aderência às especificações é certificada quando a assinatura e o padrão de referência são idênticos.

Os testes para os módulos da arquitetura presentes no Steel (RV32I e Zicsr) foram compilados seguindo as instruções da página de documentação e transformados para o formato de arquivo de memória (.mem). A seguir, foram carregados em um arranjo de memória conectado ao Steel e executados. As assinaturas foram extraídas da memória para arquivos-texto e então comparados às referências. Todas as assinaturas obtidas correspondem às referências, atestando a conformidade do Steel às especificações do RISC-V. Os resultados dos testes estão resumidos

na Tabela 21.

Tabela 21 – Resultados dos testes de compliance no Steel

Grupo de testes	Total de testes	Testes bem-sucedidos	Resultado
RV32I	48	48	Adere às especificações
Zicsr	6	6	Adere às especificações

5.3 Execução do benchmark CoreMark

A performance do Steel foi avaliada através da aplicação do *EEMBC® CoreMark*, um benchmark popular para avaliação da performance do core de microprocessadores. A sua execução produz um escore composto por um único número, permitindo comparações rápidas entre microprocessadores. O benchmark avalia a estrutura básica do pipeline de um processador, bem como as operações de I/O, de controle e sobre números inteiros [41].

O sistema exemplo da seção 4.10 (p. 47) foi implementado na plataforma Nexys-4 da Digilent especialmente para a execução do CoreMark. A plataforma é equipada com o FPGA XC7A100TCSG324-1, além de uma ponte USB-UART e osciladores, necessários para a implementação do sistema. Os arquivos do benchmark foram configurados para enviar os resultados pelo transmissor UART seguindo as orientações fornecidas pela documentação e, após compilados, foram carregados na memória. O sistema foi conectado a um oscilador de 50 MHz para a execução do benchmark.

Um simulador de terminal PuTTY foi executado em um computador hospedeiro e conectado ao transmissor UART do sistema exemplo para permitir a visualização dos resultados. O sistema completo é apresentado na Fig. 18 (p. 51) e a tela com o resultado da execução do benchmark na Fig. 19 (p. 51).

O escore de 68 iterações por segundo atingido pelo Steel foi normalizado dividindo-o pela frequência do sistema, que é de 50 MHz, totalizando 1.36 CoreMarks / MHz. Essa normalização é realizada pois o escore é diretamente dependente da frequência, de modo que dobrar a frequência dobraria o número de iterações por segundo. A Tabela 22 apresenta uma comparação da performance do Steel com o Ibex e o SCR1 utilizando os resultados divulgados pelos seus respectivos desenvolvedores para o benchmark CoreMark [42], mostrando que o Steel também é comparável a esses cores relativamente à performance de funcionamento.

Tabela 22 – Quadro comparativo da performance do Steel, Ibex e SCR1

Microprocessador	Configuração usada no teste	CoreMarks / MHz
Ibex [38]	RV32EZicsr	0.904
SCR1 [39]	RV32ICZicsr	1.27
Steel	RV32IZicsr	1.36

Figura 18 – Sistema para execução do benchmark CoreMark

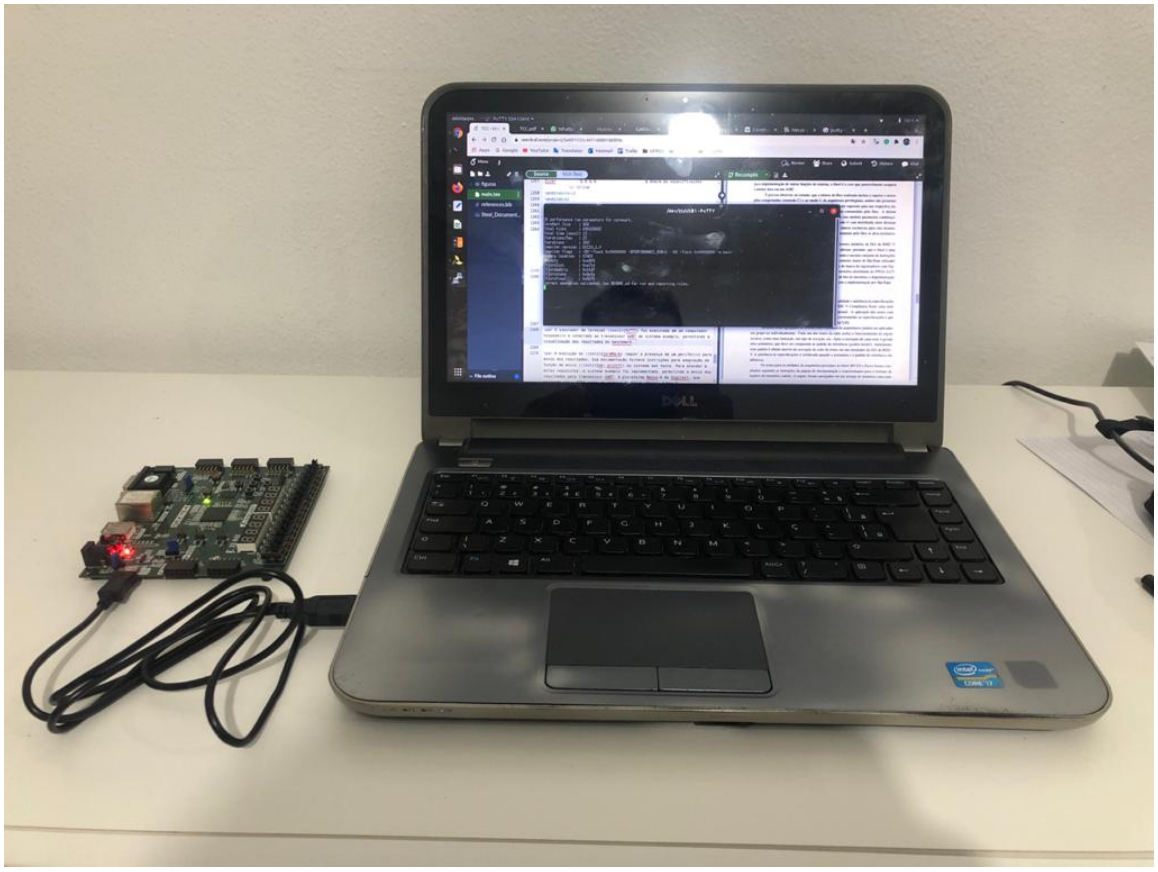


Figura 19 – Resultado da execução do benchmark CoreMark

6 CONCLUSÕES

O Steel foi desenvolvido utilizando exclusivamente ferramentas livres e, em sua maioria, de código aberto. O projeto atingiu os objetivos propostos e foi amplamente testado quanto a seu funcionamento pela aplicação da suíte de compliance disponibilizada pela RISC-V International. Os resultados dos testes indicam que o Steel adere às especificações das ISAs RV32I e Zicsr do RISC-V.

O Steel também foi testado em relação a sua performance pela aplicação do benchmark EEMBC® CoreMark, atingindo o escore de 1.36 CoreMarks / MHz. Este resultado demonstra que o Steel apresenta performance comparável aos cores Ibex e SCR1.

A Tabela 23 apresenta o resultado da comparação entre o Steel e os cores Ibex e SCR1 quanto ao consumo de recursos pós-síntese para o FPGA Artix-7 XC7A100TCSG324-1. Os resultados demonstram que o Steel é o core que utiliza a menor quantidade de recursos do FPGA. É preciso ressalvar, contudo, que o Ibex possui funcionalidades da ISA do RISC-V não presentes no Steel, o que pode justificar o maior consumo de recursos por este core.

Tabela 23 – Resumo dos principais resultados da comparação entre o Steel, Ibex e SCR1

Recurso ou característica	**Steel**	**Ibex**	**SCR1**
Configuração utilizada	RV32IZicsr + M-mode	RV32ICZicsr + M e U-mode	RV32IZicsr + M-mode
6-LUTs	1.626	2.010 (+24%)	2.359 (+45%)
LUT RAMs	48	48	0
Flip-flops	624	790 (+27%)	1.392 (+123%)

O Steel, dadas suas características e os resultados apresentados, mostra-se adequado para uso em sistemas embarcados como unidade de processamento. Ele também é competitivo em relação a implementações RISC-V semelhantes. O público interessado em utilizar o Steel ou aprimorá-lo pode consultar sua documentação, disponível no repositório do projeto.

6.1 Trabalhos futuros

Um dos objetivos do autor com o desenvolvimento do Steel é que ele possa ser reutilizado em diversos projetos. Existem diversas possibilidades para trabalhos futuros relacionados ao Steel, seja para melhorá-lo, seja para expandi-lo. A documentação completa e o código aberto possibilitam que outros indivíduos e empresas interessados na arquitetura RISC-V realizem trabalhos a partir do core.

O autor iniciou recentemente a adaptação da descrição do hardware para uma implementação ASIC. O objetivo é tornar o Steel, no futuro, um core testado em silício que possa ser utilizado em unidades microcontroladoras e *systems-on-chip*.

Existem iniciativas para verificação formal de implementações RISC-V, como o *RISC-V Formal Verification Framework* [43]. Embora ainda em progresso, estes trabalhos são úteis para realização de verificações funcionais e podem ser aplicados ao Steel.

Outra possibilidade de trabalho é a incorporação de extensões da arquitetura RISC-V à implementação. Os módulos M (para multiplicação e divisão de números inteiros) e C (para compressão de instruções) são de especial interesse por serem importantes em sistemas embarcados de pequeno porte.

Também é possível melhorar a implementação do Steel através da modificação/expansão de suas características microarquiteturais. Por exemplo, é possível adicionar ao core preditores de desvios, caches de dados e instruções integradas, controladores de interrupção integrados, etc.

Por fim, novos microcontroladores e sistemas embarcados RISC-V podem ser desenvolvidos a partir do Steel. O core foi especialmente projetado para facilitar o reúso por outros desenvolvedores.

6.2 Considerações finais

Através do desenvolvimento do presente projeto, o autor espera ter contribuído com a comunidade de hardware livre, oferendo uma implementação RISC-V livre e aberta que pode ser utilizada no desenvolvimento de sistemas embarcados e no projeto de cores mais complexos. Igualmente, espera ter contribuído para o desenvolvimento científico e tecnológico nacional, mostrando que a universidade pública e gratuita gera tecnologia e conhecimentos fundamentais para o nosso progresso.

REFERÊNCIAS

[1] D. Patterson e J. Hennessy, «Introduction», em *Computer Organization and Design, the Hardware/Software Interface, RISC-V Edition*. Elsevier, 2017, cap. 1, pp. 3–10.

[2] W. Stallings, «Reduced Instruction Set Computers», em *Computer Organization and Architecture, Designing for Performance*, 9ª ed. Pearson, 2016, cap. 15, pp. 531–574, ISBN: 978-0-13-410161-3.

[3] D. A. Patterson e D. R. Ditzel, «The case for the Reduced Instruction Set Computer», *ACM SIGARCH Computer Architecture News*, vol. 8, n.º 6, pp. 25–33, 1980.

[4] E. Blem, J. Menon, T. Vijayaraghavan e K. Sankaralingam, «ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures», *ACM Transactions on Computer Systems*, vol. 33, n.º 1, mar. de 2015. DOI: 10.1145/2699682.

[5] K. Asanović e D. Patterson, «Instruction sets should be free: the case for RISC-V», University of California, Berkeley, rel. téc., 2014.

[6] A. Waterman e K. Asanović, «Why develop a new ISA? Rationale from Berkeley Group», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 28, pp. 153–160, Document Version 20191213.

[7] ——, «RV32I Base Integer Instruction Set, Version 2.1», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 2, pp. 13–32, Document Version 20191213.

[8] ——, «Zicsr, Control and Status Register (CSR) Instructions, Version 2.0», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 9, pp. 55–59, Document Version 20191213.

[9] ——, «Machine-Level ISA, Version 1.11», em *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V Foundation, 2019, cap. 3, pp. 15–54, Document Version 20190608-Priv-MSU-Ratified.

[10] Wikipedia contributors, *Instruction set architecture — Wikipedia, The Free Encyclopedia*, https://en.wikipedia.org/w/index.php?title=Instruction_set_architecture&oldid=950316007, [Online; accessed 14-April-2020].

[11] W. Stallings, *Computer Organization and Architecture, Designing for Performance*, 9ª ed. Pearson, 2016, ISBN: 978-0-13-410161-3.

[12] A. Waterman e K. Asanović, «RV32I Base Integer Instruction Set», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 2, p. 13, Document Version 20191213.

[13] ——, «RISC-V ISA Overview», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 1, pp. 4–6, Document Version 20191213.

[14] ——, «Programmers' model for base interger ISA», em *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, cap. 1, pp. 13–15, Document Version 20191213.

[15] P. Dabbelt. (2017). All Aboard, Part 4: The RISC-V Code Models, URL: https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models (acedido em 19/04/2020).

[16] W. Stallings, «Organization and Architecture», em *Computer Organization and Architecture, Designing for Performance*, 9ª ed. Pearson, 2016, cap. 1, p. 7, ISBN: 978-0-13-410161-3.

[17] R. F. Weber, «Ciclo de busca, decodificação e execução das instruções», em *Fundamentos de Arquitetura de Computadores*, 4ª ed. Bookman, 2012, cap. 3, p. 44, Série Livros Didáticos do Instituto de Informática da UFRGS, ISBN: 978-85-407-0142-7.

[18] D. Patterson e J. Hennessy, «Building a Datapath», em *Computer Organization and Design, the Hardware/Software Interface, RISC-V Edition*. Elsevier, 2017, cap. 4, pp. 251–259.

[19] ——, *Computer Organization and Design, the Hardware/Software Interface, RISC-V Edition*. Elsevier, 2017.

[20] J. Hennessy e D. Patterson, «Basic Performance Issues in Pipelining», em *Computer Architecture, a Quantitative Approach*, 5ª ed. Elsevier, 2012, pp. 657–658.

[21] ——, «Instruction-Level Parallelism and its Exploitation», em *Computer Architecture, a Quantitative Approach*, 5ª ed. Elsevier, 2012, cap. 3, pp. 176–289.

[22] D. Patterson e J. Hennessy, «Parallelism via Instructions», em *Computer Organization and Design, the Hardware/Software Interface, RISC-V Edition*. Elsevier, 2017, cap. 4, pp. 332–343.

[23] RISC-V Foundation, *RISC-V Cores and SoC Overview — RISC-V Foundation Website*, https://riscv.org/risc-v-cores/, [Online; accessed 8-May-2020].

[24] C. Celio, D. A. Patterson e K. Asanovic, «The Berkeley Out-of-order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor», *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.

[25] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek e K. Asanović, «Chisel: constructing hardware in a scala embedded language», em *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[26] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson e K. Asanović, «BOOMv2: an open-source out-of-order RISC-V core», em *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[27] University of California, Berkeley, *BOOM's Documentation*, https://docs.boom-core.org/en/latest/, [Online; accessed 8-May-2020].

[28] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo e A. Waterman, «The Rocket Chip Generator», EECS Department, University of California, Berkeley, rel. téc. UCB/EECS-2016-17, abr. de 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[29] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse e L. Benini, «PULP: A parallel ultra low power platform for next generation IoT applications», em *2015 IEEE Hot Chips 27 Symposium (HCS)*, IEEE, 2015, pp. 1–39.

[30] F. Zaruba e L. Benini, «The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology», *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, n.º 11, pp. 2629–2640, nov. de 2019, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[31] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand e L. Benini, «Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications», em *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[32] ——, *Ibex User Manual*, https://ibex-core.readthedocs.io/en/latest/index.html, 2020.

[33] Syntacore, *SCR1 User Manual*, versão 1.1.0, dez. de 2019, 16 pp.

[34] D. Benson, *Diagrams.net — Diagram with anyone, anywhere*, https://www.diagrams.net/, 2020.

[35] D. Automation, «Committee, S. IEEE Standard for Verilog Hardware Description Language; IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)», *IEEE: Piscataway, NJ, USA*, 2006.

[36] A. Chapyzhenka e J. Probell, «WaveDrom: rendering beautiful waveforms from plain text», em *Synopsys User Group (SNUG) Silicon Valley 2016 Proceedings*, 2016.

[37] *RISC-V GNU Toolchain*, https://github.com/riscv/riscv-gnu-toolchain/, 2020.

[38] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand e L. Benini, *Ibex RISC-V Core GitHub Repository*, https://github.com/lowRISC/ibex, 2020.

[39] Syntacore, *SCR1 RISC-V Core GitHub Repository*, https://github.com/syntacore/scr1, 2020.

[40] J. Bennett, M. Bennett, S. Davidmann, N. Gala, R. Hajek, L. Moore, M. Nostersky e M. Zachariasova, *RISC-V Compliance Tests Documentation*, https://github.com/riscv/riscv-compliance/tree/master/doc, 2020.

[41] EEMBC Consortium, *CoreMark, an EEMBC benchmark - Frequently Asked Questions*, https://www.eembc.org/coremark/faq.php, 2020.

[42] ——, *CoreMark Scores*, https://www.eembc.org/coremark/scores.php, 2020.

[43] *RISC-V Formal Verification Framework*, https://github.com/SymbioticEDA/riscv-formal, 2020.

[44] A. Waterman e K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019, Document Version 20191213.

[45] J. Hennessy e D. Patterson, *Computer Architecture, a Quantitative Approach*, 5ª ed. Elsevier, 2012.

Documentation

Version 1.0

Table of Contents

1 Overview

1.1 About Steel Core

Steel is a microprocessor core that implements the RV32I and Zicsr instruction sets of the RISC-V specifications. It is designed to be easy to use and targeted for embedded systems projects.

1.2 Licensing

Steel is distributed under the MIT License. The license text is reproduced below. Read it carefully and make sure you understand its terms before using Steel in your projects.

1.3 Specifications

Steel implements the base instruction set RV32I, the Zicsr extension and the M-mode privileged architecture of the RISC-V specifications.

Steel aims to be compliant with the following versions of the RISC-V specifications:

- Base ISA RV32I version 2.1
- Zicsr extension version 2.0
- Machine ISA version 1.11

1.4 Online repository

Steel files and documentation are available at GitHub (`github.com/rafaelcalcada/steel-core`).

1.5 Getting started

To start using Steel, follow these steps:

1. Import all files inside the **rtl** directory into your project;
2. Instantiate the core into a Verilog/SystemVerilog module (an instantiation template is provided below);
3. Connect Steel to a clock source, a reset signal and memory. There is an interface to fetch instructions and another to read/write data, so we recommend a dual-port memory.

There is also interfaces to request for interrupts and to update the time register. The signals of these interfaces must be hardwired to zero if unused.

```
steel_top #(

  // You must provide a 32-bit value. If omitted the boot address is set to 0x00000000
  // -------------------------------------------------------------------------------
  .BOOT_ADDRESS()

  ) core (

  // Clock source and reset
  // -------------------------------------------------------------------------------
  .CLK(),         // System clock (input, required, 1-bit)
  .RESET(),       // System reset (input, required, 1-bit, synchronous, active high)

  // Instruction fetch interface
  // -------------------------------------------------------------------------------
  .I_ADDR(),      // Instruction address (output, 32-bit)
  .INSTR(),       // Instruction data (input, required, 32-bit)

  // Data read/write interface
  // -------------------------------------------------------------------------------
  .D_ADDR(),      // Data address (output, 32-bit)
  .DATA_IN(),     // Data read from memory (input, required, 32-bit)
  .DATA_OUT(),    // Data to write into memory (output, 32-bit)
  .WR_REQ(),      // Write enable (output, 1-bit)
  .WR_MASK(),     // Write byte mask (output, 4-bit)

  // Interrupt request interface (hardwire to zero if unused)
  // -------------------------------------------------------------------------------
  .E_IRQ(),       // External interrupt request (optional, active high, 1-bit)
  .T_IRQ(),       // Timer interrupt request (optional, active high, 1-bit)
  .S_IRQ()        // Software interrupt request (optional, active high, 1-bit)

  // Time register update interface (hardwire to zero if unused)
  // -------------------------------------------------------------------------------
  .REAL_TIME(),   // Value read from a real-time counter (optional, 64-bit)

);
```

1.6 Configuration

Steel has only one configuration parameter, the boot address, which is the address of the first instruction the core will fetch after reset. It is defined when instantiating Steel. If you omit this parameter at instantiation, the boot address will be set to 0x00000000.

1.7 Microarchitecture

Steel has 3 pipeline stages, a single execution thread and issues one instruction per clock cycle. Therefore, all instructions are executed in program order. Its pipeline is plain simple, divided into fetch, decode, and execution stages. The reduced number of pipeline stages eliminates the need for branch predictors and other advanced microarchitectural units, like data hazard detectors and forwarding units. Fig. 1 shows the Steel microarchitecture in register-transfer level (RTL). More implementation details can be found in section 6.

Fig. 2 (next page) shows the tasks performed by each pipeline stage. In the first stage, the core generates the program counter and fetches the instruction from memory. In the second, the instruction is decoded and the control signals for all units are generated. Branches, jumps and stores are executed in advance in this stage, which also generates the immediates and fetches the data from memory for load instructions. The last stage executes all other instructions and writes back the results in the register file.

Figure 1 – Steel Core microarchitecture in detail

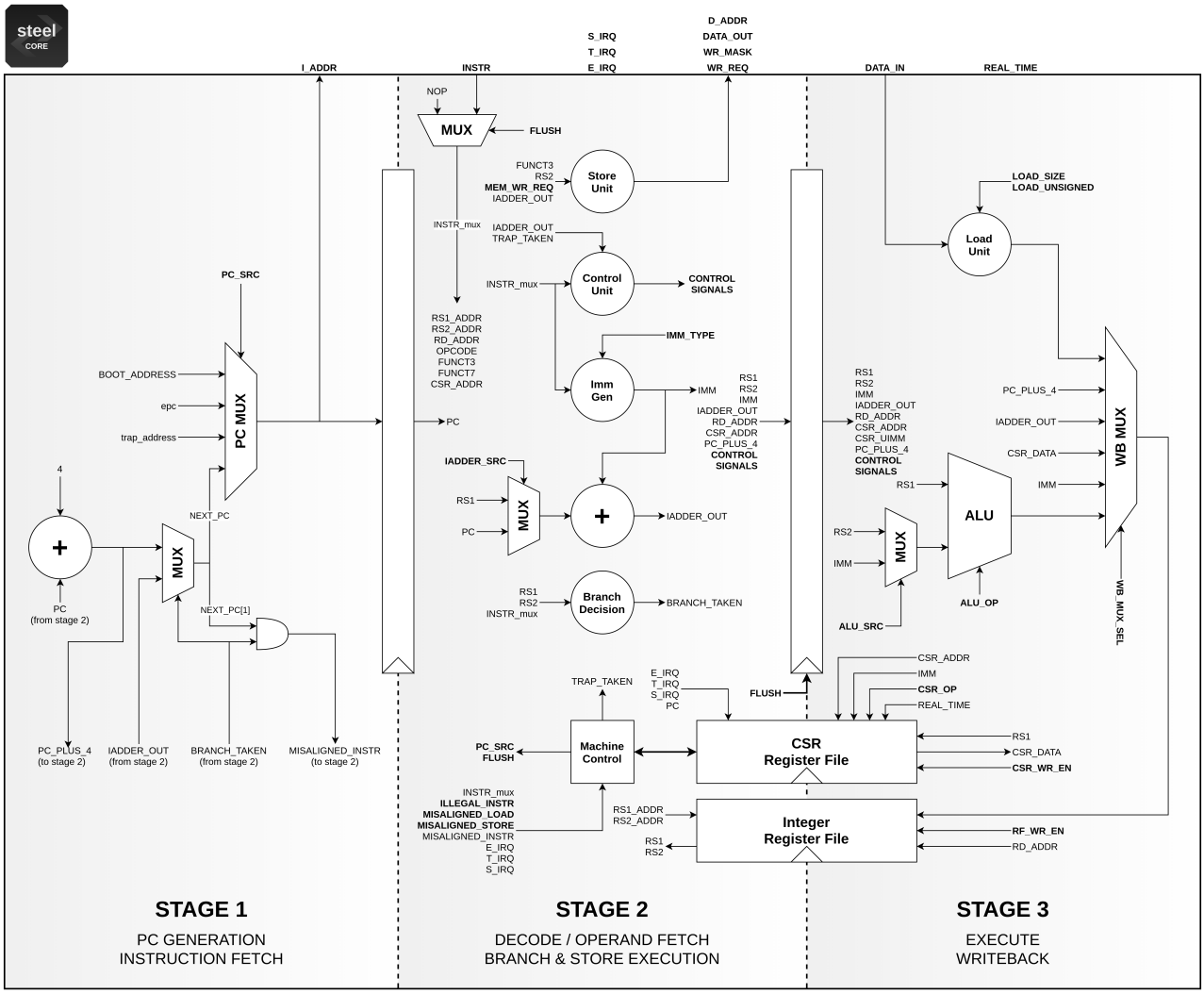
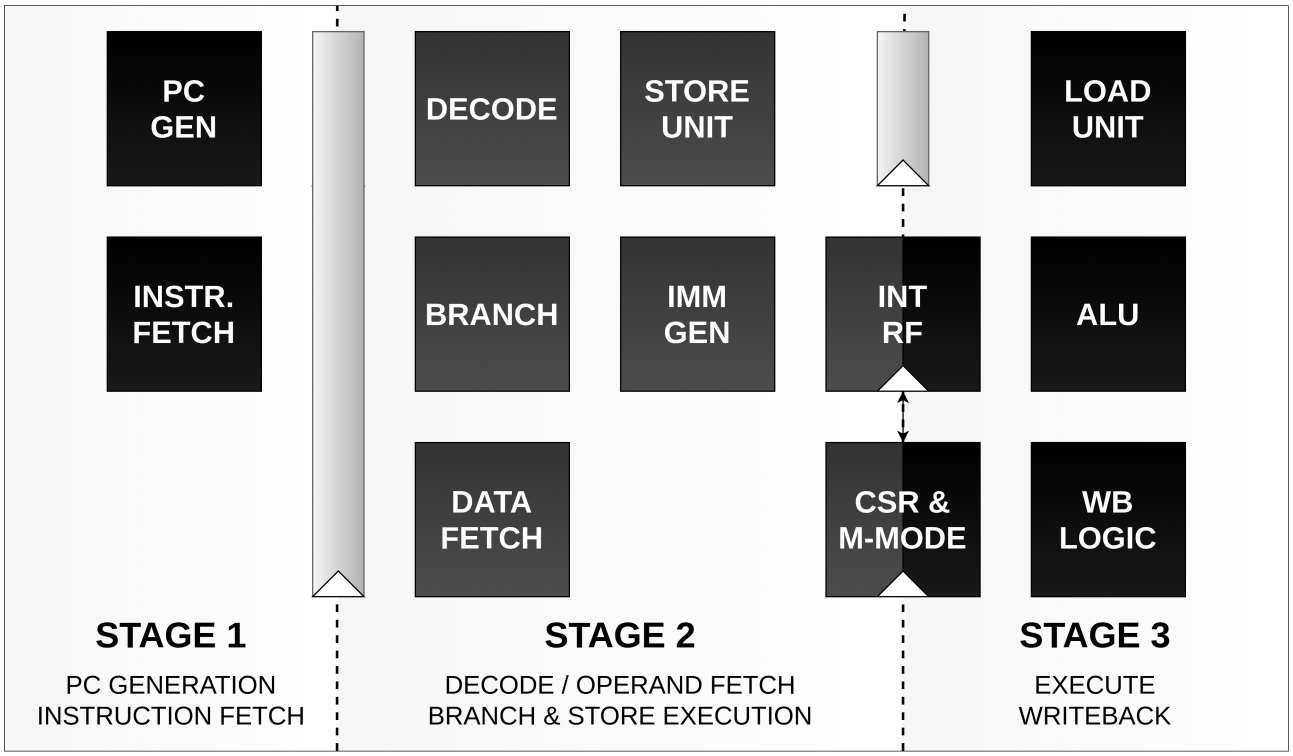


Figure 2 – Steel Core pipeline overview



2 Input and output signals

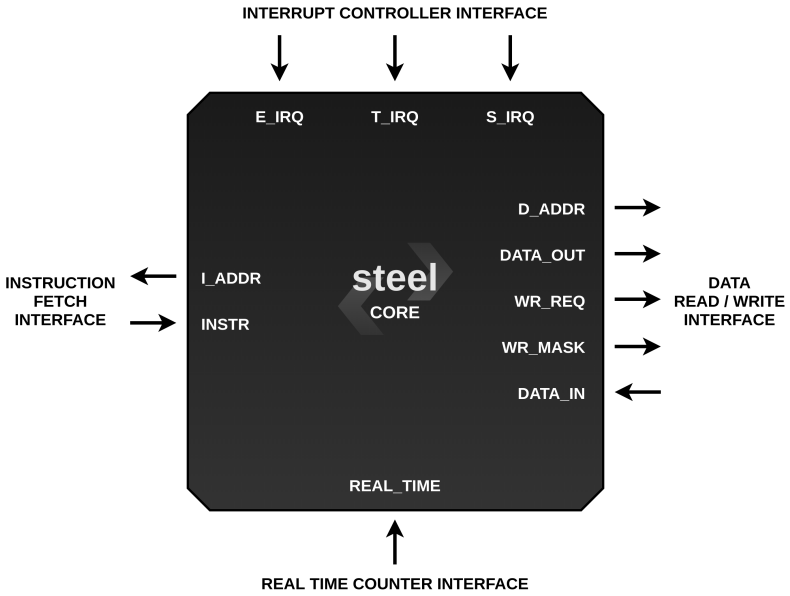
Steel has 4 communication interfaces, shown in the figure below.

The core was designed to be connected to a memory with one clock cycle read/write latency, which means that the memory should take one clock cycle to complete both read and write operations.

The interrupt request interface has signals to request for external, timer and software interrupts, respectively. They can be connected to a single device or to an interrupt controller managing interrupt requests from several devices. If your system does not need interrupts you should hardwire these signals to zero.

The real-time counter interface provides a 64-bit bus to read the value from a real-time counter and update the time register. If your system does not need hardware timers, you should hardwire this signal to zero.

Figure 3 – Steel Core input and output signals



2.1 Instruction fetch interface

The instruction fetch interface has two signals used in the instruction fetch process, shown in Table 1. The process of fetching instructions is explained in section 3.1.

Table 1 – Instruction fetch interface signals

Signal	Width	Direction	Description
INSTR	32 bits	Input	Contains the instruction fetched from memory.
I_ADDR	32 bits	Output	Contains the address of the instruction the core wants to fetch from memory.

2.2 Data read/write interface

The data read/write interface has five signals used in the process of reading/writing data from/to memory. The signals are shown in Table 2. The process of fetching data from memory is explained in section 3.2. The process of writing data is explained in section 3.3.

2.3 Interrupt controller interface

The interrupt controller interface has three signals used to request external, timer and software interrupts, shown in Table 3. The interrupt request process is explained in sections 3.4 and 4.

Table 2 – Data read/write interface signals

Signal	Width	Direction	Description
DATA_IN	32 bits	Input	Contains the data fetched from memory.
D_ADDR	32 bits	Output	In a write operation, contains the address of the memory position where the data will be stored. In a read operation, contains the address of the memory position where the data to be fetched is. The address is always aligned on a four byte boundary (the last two bits are always zero).
DATA_OUT	32 bits	Output	Contains the data to be stored in memory. Used only with write operations.
WR_REQ	1 bit	Output	When high, indicates a request to write data. Used only with write operations.
WR_MASK	4 bits	Output	Contains a mask of four *byte-write enable* bits. A bit high indicates that the corresponding byte must be written. See section 3.3 for details. Used only with write operations.

Table 3 – Interrupt controller interface signals

Signal	Width	Direction	Description
E_IRQ	1 bit	Input	When high indicates an external interrupt request.
T_IRQ	1 bit	Input	When high indicates a timer interrupt request.
S_IRQ	1 bit	Input	When high indicates a software interrupt request.

2.4 Real time counter interface

The real time counter interface has just one signal used to update the `time` CSR, shown in Table 4. The process of updating the `time` register is explained in section 3.5.

Table 4 – Real time counter interface

Signal	Width	Direction	Description
REAL_TIME	64 bits	Input	Contains the current value read from a real time counter.

2.5 CLK and RESET signals

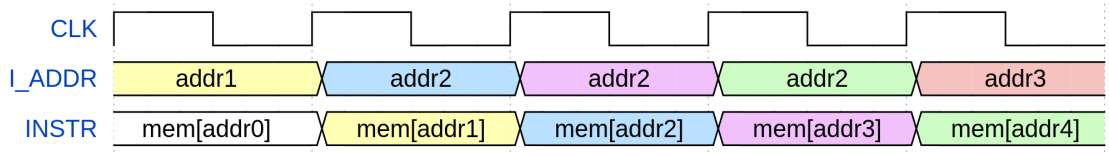
The core has CLK and RESET input signals, which were not shown in figure 3 (above). The CLK signal must be connected to a clock source. The RESET signal is active high and resets the core synchronously.

3 Timing diagrams

3.1 Instruction fetch

To fetch an instruction, the core places the instruction address on the I_ADDR bus. The memory must place the instruction on the INSTR bus at the next clock rising edge. Fig. 4 shows the timing diagram of this process. In the figure, `mem[addrX]` denotes the instruction stored at the memory position `addrX`.

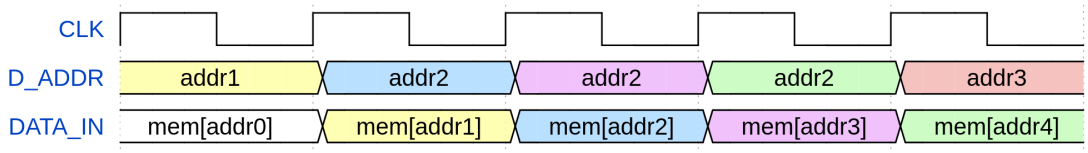
Figure 4 – Instruction fetch timing diagram



3.2 Data fetch

To fetch data from memory, the core puts the data address on the D_ADDR bus. The memory must place the data on the DATA_IN bus at the next clock rising edge. Fig. 5 shows the timing diagram of this process. In the figure, `mem[addrX]` denotes the data stored at the memory position `addrX`.

Figure 5 – Data fetch timing diagram



3.3 Data writing

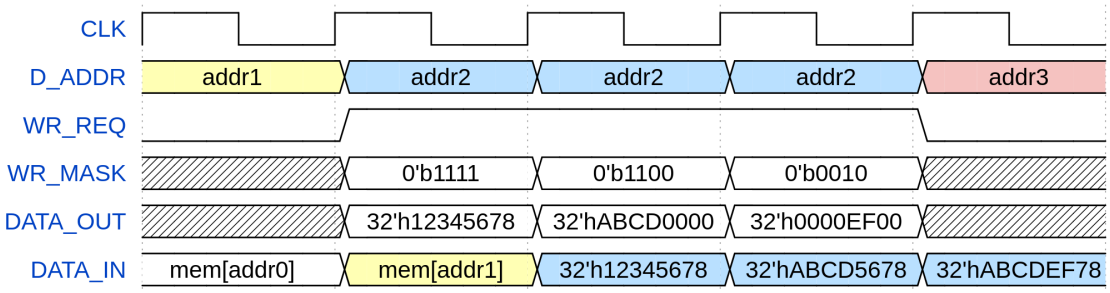
To write data to memory, the core drives D_ADDR, DATA_OUT, WR_REQ and WR_MASK signals as follows:

- D_ADDR receives the address of the memory position where the data must be written;
- DATA_OUT receives the data to be written;
- WR_REQ is set high;
- WR_MASK receives a byte-write enable mask that indicates which bytes of DATA_OUT must be written.

The memory must perform the write operation at the next clock rising edge. The core can request to write bytes, halfwords and words.

Fig. 6 (next page) shows the process of writing data to memory. DATA_IN is not used in the process and appears only to show the memory contents after writing. The figure shows five clock cycles, in which the core requests to write in the second, third and fourth cycles. In the second clock cycle, the core requests to write the word `0x12345678` at the address `addr2`. In the third, requests to write the halfword `0xABCD` at the upper half of `addr2`, and in the fourth requests to write the byte `0xEF` at the second least significant byte of `addr2`. The content of `addr2` after each of these operations appears on the DATA_IN bus and are highlighted in blue.

Figure 6 – Data writing timing diagram

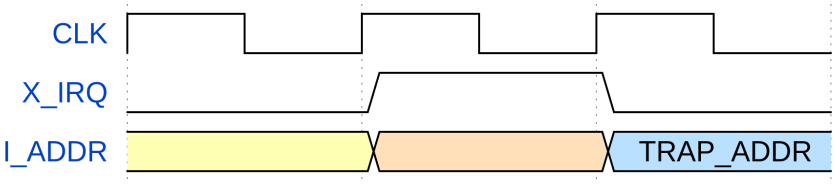


3.4 Interrupt request

An external device (or an interrupt controller managing several devices) can request interrupts by setting high the appropriate IRQ signal, which is E_IRQ for external interrupts, T_IRQ for timer interrupts and S_IRQ for software interrupts. The IRQ signal of the requested interrupt must be set high for one clock cycle and set low for the next.

Fig. 7 shows the timing diagram of the interrupt request process. Since the process is the same for all types of interrupt, X_IRQ is used to denote E_IRQ, T_IRQ or S_IRQ. TRAP_ADDR denotes the address of the trap handler first instruction.

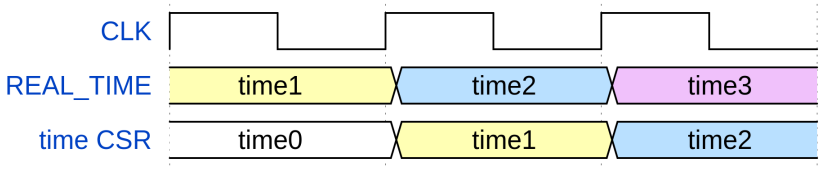
Figure 7 – Interrupt request timing diagram



3.5 Time CSR update

When connected to a real-time counter, the core updates the `time` CSR with the value placed on REAL_TIME at each clock rising edge, as shown in Fig. 8 . In the figure, `timeX` denotes arbitrary time values.

Figure 8 – `time` CSR update timing diagram



4 Exceptions and Interrupts

4.1 Supported exceptions and interrupts

Steel supports the exceptions and interrupts shown in Table 5. They are listed in descending priority order (the highest priority is at the top of the table). If two or more exceptions/interrupts occur at the same time, the one with the highest priority is taken.

Exceptions always cause a trap to be taken. An interrupt will cause a trap only if enabled. Each type of interrupt has an interrupt-enable bit in the `mie` register. Interrupts are globally enable/disabled by setting the MIE bit of `mstatus` register.

Table 5 – Steel supported exceptions and interrupts

	`mcause` **value**	
Exception / Interrupt	**Interrupt**	**Exception code**
Machine external interrupt	1	11
Machine software interrupt	1	3
Machine timer interrupt	1	7
Illegal instruction exception	0	2
Instruction address-misaligned exception	0	0
Environment call from M-mode exception	0	11
Environment break exception	0	3
Store address-misaligned exception	0	6
Load address-misaligned exception	0	4

4.2 Trap handling in Steel

Exceptions and interrupts are handled by a trap handler routine stored in memory. The address of the trap handler first instruction is configured using the `mtvec` register. Steel supports both direct and vectorized interrupt modes.

When a trap is taken, the core proceeds as follows:

- the address of the interrupted instruction (or the instruction that encountered the exception) is saved in the `mepc` register;
- the value of the `mtval` register is set to zero;
- the value of the `mstatus` MIE bit is saved in the MPIE field and then set to zero;
- the program counter is set to the trap handler first instruction.

The `mret` instruction is used to return from traps. When executed, the core proceeds as follows:

- the value of the `mstatus` MPIE bit is saved in the MIE field and then set to one;
- the program counter is set to the value of `mepc` register.

4.3 Nested interrupts capability

The core globally disables interrupts when takes into a trap. The trap handler can re-enable interrupts by setting the `mstatus` MIE bit to one, enabling nested interrupts. To return from nested traps, the trap handler must stack and manage the values of the `mepc` register in memory.

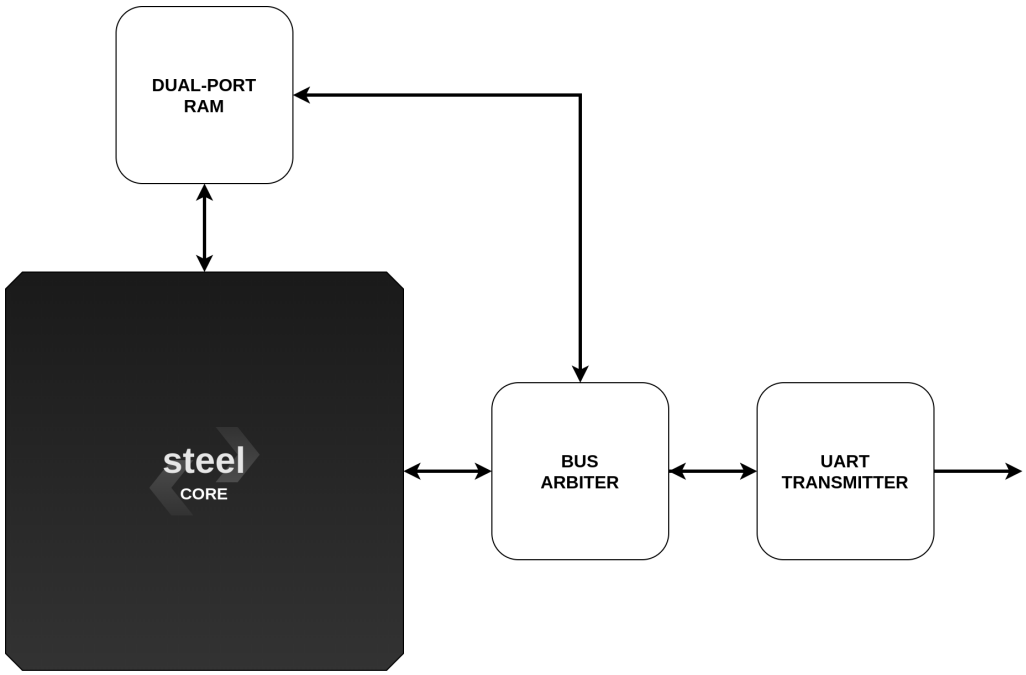
5 Example system built with Steel

The figure below shows an example system built with Steel, composed of an RAM memory array, a memory mapped UART transmitter, a bus arbiter and, of course, the Steel Core. The timer and interrupt request signals are hardwired to zero because neither timers nor interrupts are needed in this system. The implementation files of this system are inside the **soc** directory in the project repository. The **util** directory has an example program (**hello.c**) for this system. The program sends the string ”Hello World, Steel!” through the UART transmitter.

Note that the RAM memory and the UART transmitter share the interface to read/write data. The bus arbiter is used to multiplex this interface signals according to the address the core wants to access. In this example, the address 0x00010000 is used to access the UART transmitter. RAM addresses start at 0x00000000 and end at 0x00001fff. All other addresses are invalid.

The RISC-V GNU Toolchain provides the Newlib cross-compiler, which can be used to compile software for Steel. Instructions to install it can be found in the toolchain repository (available at `https://github.com/riscv/riscv-gnu-toolchain`).

Figure 9 – Steel-based system example



6 Implementation details

This section contains information on implementation details. It is intended for those who want to know more about how Steel works.

6.1 Implemented control and status registers

The control and status registers implemented in Steel are shown in Table 6. Other M-mode registers not shown in the table return the hardwired value defined by the RISC-V specifications when read.

Table 6 – Steel Core implemented CSRs

CSR	Name	Address
cycle	*Cycle Counter*	`0xC00`
time	*System Timer*	`0xC01`
instret	*Instructions Retired*	`0xC02`
mstatus	*Machine Status*	`0x300`
misa	*Machine ISA*	`0x301`
mie	*Machine Interrupt Enable*	`0x304`
mtvec	*Machine Trap Vector*	`0x305`
mscratch	*Machine Scratch*	`0x340`
mepc	*Machine Exception Program Counter*	`0x341`
mcause	*Machine Cause*	`0x342`
mtval	*Machine Trap Value*	`0x343`
mip	*Machine Interrupt Pending*	`0x344`
mcycle	*Machine Cycle Counter*	`0xB00`
minstret	*Machine Instructions Retired*	`0xB01`
mcountinhibit	*Machine Counter Inhibit*	`0x320`

6.2 Modules

6.2.1 Decoder

The Decoder (`decoder.v`) decodes the instruction and generates the signals that control the memory, the Load Unit, the Store Unit, the ALU, the two register files (Integer and CSR), the Immediate Generator and the Writeback Multiplexer. The description of its input and output signals are shown in Table 7.

Table 7 – Decoder input/output signals

Signal name	Width	Direction	Description
OPCODE_6_TO_2	5 bits	Input	Connected to the instruction *opcode* field.
FUNCT7_5	1 bit	Input	Connected to the instruction *funct7* field.
FUNCT3	3 bits	Input	Connected to the instruction *funct3* field.
IADDER_OUT_1_TO_0	2 bits	Input	Used to verify the alignment of loads and stores.
TRAP_TAKEN	1 bit	Input	When high indicates that a trap will be taken in the next clock cycle. Connected to the Machine Control module.
ALU_OPCODE	4 bits	Output	Selects the operation to be performed by the ALU. See Table 9 for possible values.
MEM_WR_REQ	1 bit	Output	When high indicates a request to write to memory.
LOAD_SIZE	2 bits	Output	Indicates the word size of load instruction. See Table 14.
LOAD_UNSIGNED	1 bit	Output	Indicates the type of load instruction (signed or unsigned). See Table 14.
ALU_SRC	1 bit	Output	Selects the ALU 2nd operand.
IADDER_SRC	1 bit	Output	Selects the Immediate Adder 2nd operand.
CSR_WR_EN	1 bit	Output	Controls the WR_EN input of CSR Register File.
RF_WR_EN	1	Output	Controls the WR_EN input of Integer Register File. See Table 11.
WB_MUX_SEL	3	Output	Selects the data to be written in the Integer Register File.
IMM_TYPE	3	Output	Selects the immediate based on the type of the instruction.
CSR_OP	3	Output	Selects the operation to be performed by the CSR Register File (read/write, set or clear).
ILLEGAL_INSTR	1 bit	Output	When high indicates that an invalid or not implemented instruction was fetched from memory.
MISALIGNED_LOAD	1 bit	Output	When high indicates an attempt to read data in disagreement with the memory alignment rules.
MISALIGNED_STORE	1 bit	Output	When high indicates an attempt to write data to memory in disagreement with the memory alignment rules.

6.2.2 ALU

The ALU (`alu.v`) applies ten distinct logical and arithmetic operations in parallel to two 32-bit operands, outputting the result selected by `OPCODE`. ALU input and output signals are shown in Table 8, and opcodes are shown in Table 9.

The opcode values were assigned to facilitate instruction translation. The most significant bit of `OPCODE` matches with the second most significant bit in the instruction *funct7* field. The remaining three bits match with the instruction *funct3* field.

Table 8 – ALU signals

Signal name	Width	Direction	Description
OP_1	32 bits	Input	Operation first operand.
OP_2	32 bits	Input	Operation second operand.
OPCODE	4 bits	Input	Operation code. This signal is driven by *funct7* and *funct3* instruction fields.
RESULT	32 bits	Output	Result of the requested operation.

Table 9 – ALU opcodes

Opcode	Operation	Binary value
ALU_ADD	Addition	4'b0000
ALU_SUB	Subtraction	4'b1000
ALU_SLT	Set on less than	4'b0010
ALU_SLTU	Set on less than unsigned	4'b0011
ALU_AND	Bitwise logical AND	4'b0111
ALU_OR	Bitwise logical OR	4'b0110
ALU_XOR	Bitwise logical XOR	4'b0100
ALU_SLL	Logical left shift	4'b0001
ALU_SRL	Logical right shift	4'b0101
ALU_SRA	Arithmetic right shift	4'b1101

6.2.3 Integer Register File

The Integer Register File (`integer_file.v`) has 32 general-purpose registers and supports read and write operations. Reads are requested in the pipeline stage 2 and provide data from one or two registers. Writes are requested in the pipeline stage 3 and put the data coming from the Writeback Multiplexer into the selected register. If stage 3 requests to write to a register being read by stage 2, the data to be written is immediately forwarded to stage 2. Each operation is driven by a distinct set of signals, shown in tables 10 and 11.

Table 10 – Integer Register File signals for read operation

Signal name	Width	Direction	Description
RS_1_ADDR	5 bits	Input	*Register source 1 address*. The data is placed at RS_1 immediately after an address change.
RS_2_ADDR	5 bits	Input	*Register source 2 address*. The data is placed at RS_2 immediately after an address change.
RS_1	32 bits	Output	Data read (source 1).
RS_2	32 bits	Output	Data read (source 2).

Table 11 – Integer Register File signals for write operation

Signal name	Width	Direction	Description
RD_ADDR	5 bits	Input	*Destination register address*.
RD	32 bits	Input	Data to be written in the destination register.
WR_EN	1 bit	Input	*Write enable*. When high, the data placed on RD is written in the destination register at the next clock rising edge.

6.2.4 Branch Unit

The Branch Unit (`branch_unit.v`) decides if a branch instruction must be taken or not. It receives two operands from the Integer Register File and, based on the value of *opcode* and *funct3* instruction fields, decides the branch. Jump instructions are interpreted as branches that must always be taken. Internally, the unit realizes just two comparisions, deriving other four from them. Table 12 shows the module input and output signals.

Table 12 – Branch Unit signals

Signal name	Width	Direction	Description
OPCODE_6_TO_2	5 bits	Input	Connected to the *opcode* instruction field.
FUNCT3	3 bits	Input	Connected to the *funct3* instruction field.
RS1	32 bits	Input	Connected to the register file 1st operand source.
RS2	32 bits	Input	Connected to the register file 2nd operand source.
BRANCH_TAKEN	1 bit	Output	High if the branch must be taken, low otherwise.

6.2.5 Load Unit

The Load Unit (`load_unit.v`) reads DATA_IN input signal and forms a 32-bit value based on the load instruction type (encoded in the *funct3* field). The formed value (placed on OUTPUT) can then be written in the Integer Register File. The module input and output signals are shown in Table 13. The value of OUTPUT is formed as shown in Table 14.

Table 13 – Load Unit signals

Signal name	Width	Direction	Description
LOAD_SIZE	2 bits	Input	Connected to the two least significant bits of the *funct3* instruction field.
LOAD_UNSIGNED	1 bit	Input	Connected to the most significant bit of the *funct3* instruction field.
DATA_IN	32 bits	Input	32-bit word read from memory.
IADDER_OUT_1_TO_0	2 bits	Input	Indicates the byte/halfword position in DATA_IN. Used only with load byte/halfword instructions.
OUTPUT	32 bits	Output	32-bit value to be written in the Integer Register File.

Table 14 – Load Unit output generation

LOAD_SIZE	**Effect on OUTPUT**
2'b00	The byte in the position indicated by IADDER_OUT_1_TO_0 is placed on the least significant byte of OUTPUT. The upper 24 bits are filled according to the LOAD_UNSIGNED signal.
2'b01	The halfword in the position indicated by IADDER_OUT_1_TO_0 is placed on the least significant halfword of OUTPUT. The upper 16 bits are filled according to the LOAD_UNSIGNED signal.
2'b10	All bits of DATA_IN are placed on OUTPUT.
2'b11	All bits of DATA_IN are placed on OUTPUT.
LOAD_UNSIGNED	**Effect on OUTPUT**
1'b0	The remaining bits of OUTPUT are filled with the sign bit.
1'b1	The remaining bits of OUTPUT are filled with zeros.

6.2.6 Store Unit

The Store Unit (`store_unit.v`) drives the signals that interface with memory. It places the data to be written (which can be a byte, halfword or word) in the right position in DATA_OUT and sets the value of WR_MASK in an appropriate way. Table 15 shows the unit input and output signals.

Table 15 – Store Unit signals

Signal name	Width	Direction	Description
FUNCT3	3 bits	Input	Connected to the *funct3* instruction field. Indicates the data size (byte, halfword or word).
IADDER_OUT	32 bits	Input	Contains the address (possibly unaligned) where the data must be written.
RS2	32 bits	Input	Connected to Integer Register File source 2. Contains the data to be written (possibly in the wrong position).
MEM_WR_REQ	1 bit	Input	Control signal generated by the Control Unit. When high indicates a request to write to memory.
DATA_OUT	32 bits	Output	Contains the data to be written in the right position.
D_ADDR	32 bits	Output	Contains the address (aligned) where the data must be written.
WR_MASK	4 bits	Output	A bitmask that indicates which bytes of DATA_OUT must be written. For more information, see section 3.3.
WR_REQ	1 bit	Output	When high indicates a request to write to memory.

6.2.7 Immediate Generator

The Immediate Generator (`imm_generator.v`) rearranges the immediate bits contained in the instruction and, if necessary, sign-extends it to form a 32-bit value. The unit is controlled by the IMM_TYPE signal, generated by the Control Unit. Table 16 shows the unit input and output signals.

Table 16 – Immediate Generator signals

Signal name	Width	Direction	Description
INSTR	25 bits	Input	Connected to the instruction bits (32 to 7).
IMM_TYPE	2 bits	Input	Control signal generated by the Control Unit that indicated the type of immediate that must be generated.
IMM	32 bits	Output	32-bit generated immediate.

6.2.8 CSR Register File

The CSR Register File (`csr_file.v`) has the control and status registers required for the implementation of M-mode. Read/write, set and clear operations can be applied to the registers. Table 17 shows the unit input and output signals, except those used for communication with the Machine Control, which are shown in Table 18.

Table 17 – CSR Register File signals

Signal name	Width	Direction	Description
WR_EN	1 bit	Input	*Write enable*. When high, updates the CSR addressed by CSR_ADDR at the next clock rising edge according to the operation selected by CSR_OP.
CSR_ADDR	12 bits	Input	Address of the CSR to read/write/modify.
CSR_OP	3 bits	Input	Control signal generated by the Control Unit. Selects the operation to be performed (read/write, set, clear or no operation).
CSR_UIMM	5 bits	Input	*Unsigned immediate*. Connected to the five least significant bits from the Immediate Generator output.
CSR_DATA_IN	32 bits	Input	In write operations, contains the data to be written. In set or clear operations, contains a bit mask.
PC	32 bits	Input	*Program counter* value. Used to update the `mepc` CSR.
E_IRQ	1 bit	Input	*External interrupt request*. Used to update the `MEIP` bit of `mip` CSR.
T_IRQ	1 bit	Input	*Timer interrupt request*. Used to update the `MTIP` bit of `mip` CSR.
S_IRQ	1 bit	Input	*Software interrupt request*. Used to update the `MSIP` bit of `mip` CSR.
REAL_TIME	64 bits	Input	Current value of the real time counter. Used to update the `time` and `timeh` CSRs.
CSR_DATA_OUT	32 bits	Output	Contains the data read from the CSR addressed by CSR_ADDR.
EPC_OUT	32 bits	Output	Current value of the `mepc` CSR.
TRAP_ADDRESS	32 bits	Output	Address of the trap handler first instruction.

Table 18 – CSR Register File and Machine Control interface signals

Signal name	Width	Direction[1]	Description
I_OR_E	1 bit	Input	*Interrupt or exception*. When high indicates an interrupt, otherwise indicates an exception. Used to update the most significant bit of `mcause` register.
CAUSE_IN	4 bits	Input	Contains the exception code. Used to update the `mcause` register. See Table 5.
SET_CAUSE	1 bit	Input	When high updates the `mcause` register with the values of I_OR_E and CAUSE_IN.
SET_EPC	1 bit	Input	When high, updates the `mepc` register with the value of PC.
INSTRET_INC	1 bit	Input	When high enables the instructions retired counting.
MIE_CLEAR	1 bit	Input	When high sets the MIE bit of `mstatus` to zero (which globally disables interrupts). The old value of MIE is saved in the `mstatus` MPIE field.
MIE_SET	1 bit	Input	When high sets the MPIE bit of `mstatus` to one. The old value of MPIE is saved in the `mstatus` MIE field.
MIE	1 bit	Output	Current value of MIE bit of `mstatus` CSR.
MEIE_OUT	1 bit	Output	Current value of MEIE bit of `mie` CSR.
MTIE_OUT	1 bit	Output	Current value of MTIE bit of `mie` CSR.
MSIE_OUT	1 bit	Output	Current value of MSIE bit of `mie` CSR.
MEIP_OUT	1 bit	Output	Current value of MEIP bit of `mip` CSR.
MTIP_OUT	1 bit	Output	Current value of MTIP bit of `mip` CSR.
MSIP_OUT	1 bit	Output	Current value of MSIP bit of `mip` CSR.

[1] Direction regarding to the CSR Register File. An input of CSR Register File is an output of Machine Control and vice-versa.

6.2.9 Machine Control

The Machine Control module (`machine_control.v`) implements the M-mode, controlling the the program counter generation and updating several CSRs. It has a special communication interface with the CSR Register File, already shown in Table 18 (above). Its input and output signals are shown in Table 19.

Internally, the module implements the finite state machine shown in figure 10 (next page).

Table 19 – Machine Control module signals

Signal name	Width	Direction	Description
ILLEGAL_INSTR	1 bit	Input	*Illegal instruction*. When high indicates that an invalid or not implemented instruction was fetched from memory.
MISALIGNED_INSTR	1 bit	Input	*Misaligned instruction*. When high indicates an attempt to fetch an instruction which address is in disagreement with the memory alignment rules.
MISALIGNED_LOAD	1 bit	Input	*Misaligned load*. When high indicates an attempt to read data in disagreement with the memory alignment rules.
MISALIGNED_STORE	1 bit	Input	*Misaligned store*. When high indicates an attempt to write data to memory in disagreement with the memory alignment rules.
OPCODE_6_TO_2	5 bits	Input	Value of the *opcode* instruction field.
FUNCT3	3 bits	Input	Value of the *funct3* instruction field.
FUNCT7	7 bits	Input	Value of the *funct7* instruction field.
RS1_ADDR	5 bits	Input	Value of the *rs1* instruction field.
RS2_ADDR	5 bits	Input	Value of the *rs2* instruction field.
RD_ADDR	5 bits	Input	Value of the *rd* instruction field.
E_IRQ	1 bit	Input	*External interrupt request*.
T_IRQ	1 bit	Input	*Timer interrupt request*.
S_IRQ	1 bit	Input	*Software interrupt request*.
PC_SRC	2 bit	Output	Selects the program counter source.
FLUSH	1 bit	Output	Flushes the pipeline when set.
TRAP_TAKEN	1 bit	Output	When high indicates that a trap will be taken in the next clock cycle.

Figure 10 – M-mode finite state machine