Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Modelagem de uma arquitetura RISC-V com aceleração por Cache

Fábio Trevizolo de Souza
Universidade de Brasília - UnB

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Modelagem de uma arquitetura RISC-V com aceleração por Cache

Fábio Trevizolo de Souza
Universidade de Brasília - UnB

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. Marcus Vinicius Lamar
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenadora do Curso de Engenharia da Computação

Brasília, 10 de julho de 2019

Dedicatória

Dedico este trabalho em primeiro lugar a Deus, soberano, criador de todas as coisas. Com sua infinita sabedoria me orientou e cuidou ao longo dessa jornada. Toda honra e toda glória aquele que me capacitou e proporcionou criatividade para realização deste trabalho, sem Ele nada disso seria possível.

Ao meu pai, minha mãe e aos meus irmãos.

Agradecimentos

Primeiramente a Deus que permitiu que tudo isso acontecesse, ao longo da minha vida, e não somente nestes anos como universitário, mas que em todos os momentos é o maior mestre que alguém pode conhecer.

Aos meus pais e irmãos, pelo amor, incentivo e apoio incondicional.

Aos amigos, colegas e Hermanos, pelo incentivo e apoio incondicional.

Ao Professor Dr. Ricardo Pezzuol Jacobi pela oportunidade e apoio na elaboração deste trabalho.

A todos os professores por me proporcionar o conhecimento não apenas intelectual, mas a manifestação do caráter e afetividade da educação no processo de formação profissional, por tanto que se dedicaram a mim, não somente por terem me ensinado, mas por terem me feito aprender.

A Universidade de Brasília, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro.

E a todos que direta ou indiretamente fizeram parte da minha formação.

Resumo

À medida que a diferença de desempenho entre os processadores e a memória continuam a aumentar, implementações de memórias cache são cada vez mais indispensáveis na tentativa de diminuir essa brecha. Nesse projeto, um modelo em VHDL da ISA RISC-V é implementado junto com uma cache mapeada diretamente. O objetivo é tentar otimizar o tempo de acesso e o tempo ocioso gasto pelo processador aguardando os níveis mais altos de memória, transferir dados. Além disso, avaliar o custo de elementos lógicos na implementação em uma FPGA, bem como o ganho de desempenho quando a implementação do RISC-V trabalha em conjunto com uma memória cache.

Palavras-chave: RISC-V, Cache, Hardware, Arquitetura de Processadores, Conjunto de Instruções

Abstract

As the performance gap between processors and main memory continues to widen, implementations of cache memories are needed to bridge the gap. In this project, a VHDL design of the RISC-V ISA is implemented along with a direct mapped cache looking to optimize the access time and idle time spent waiting for the upper levels of memory, transfer data. Also, to evaluate the cost of logic elements when loading it on an FPGA as well as the performance gain when the RISC-V implementation works with a cache.

Keywords: RISC-V, Cache, Hardware, Processor Architecture, Instruction Set

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CMOS *Complementary Metal-Oxide-Semiconductor* - Semicondutor de Metal-Óxido Complementar.

CPU *Central Processing Unit* - unidade central de processamento.

DARPA *Defense Advanced Research Projects Agency* - Agência de Projetos de Pesquisa Avançada de Defesa.

DUT *Device Under Test* - Dispositivo em teste.

FPGA *Field Programmable Gate Array* - Arranjo de Portas Programáveis em Campo.

HDL *Hardware Description Language* - Linguagem de Descrição de Hardware.

HxD *Freeware Hex Editor and Disk Editor* - Editor Hexadecimal e Editor de Disco Gratuito.

IDE *Integrated Development Environment* - Ambiente de Desenvolvimento Integrado.

IEEE *Institute of Electrical and Electronic Engineers* - Instituto de Engenheiros Eletricistas e Eletrônicos.

ISA *Instruction Set Architecture* - Arquitetura do conjunto de instruções.

RARS *RISC-V Assembler and Runtime Simulator* - Montador e Simulador de tempo de execução do RISC-V.

RISC *Reduced Instruction Set Computer* - Computador com um conjunto reduzido de instruções.

SRAM *Static Random-Access Memory* - Memória de Acesso Aleatório Estático.

TTL *Transistor–transistor Logic* - Lógica transistor-transistor.

UnB Universidade de Brasília.

VHDL *VHSIC Hardware Description Language* - Linguagem de descrição de hardware VHSIC.

VHSIC *Very High Speed Integrated Circuits* - Circuito integrado de velocidade muito alta.

Capítulo 1

Introdução

1.1 Hardware Livre

Software livre é uma questão de liberdade, não de preço; em termos gerais, significa que os usuários estão livres para usar o software, copiar e redistribuir com ou sem alterações. Aplicando o mesmo conceito diretamente ao hardware, hardware livre significa hardware que os usuários são livres para usar, copiar e redistribuir com ou sem alterações. O hardware de código aberto é um hardware cujo *design* é disponibilizado publicamente para que qualquer pessoa possa estudar, modificar, distribuir, fabricar e vender o *design* ou o hardware com base nesse *design* [5].

As pessoas que encontram pela primeira vez a ideia de software livre geralmente pensam que isso significa que se pode obter uma cópia grátis. Muitos programas gratuitos estão disponíveis por preço zero, já que não custa nada fazer o download da sua cópia, mas não é o que significa "livre" aqui. Para hardware, essa confusão tende a ir na outra direção; o hardware custa dinheiro para produzir, portanto, o hardware fabricado comercialmente não será gratuito, mas isso não impede que seu *design* seja livre/gratuito. Por exemplo, as coisas que se faz em uma impressora 3D própria podem ser bem baratas, mas não exatamente grátis, já que as matérias-primas normalmente custam algo. Em termos éticos, a questão da liberdade supera totalmente a questão do preço, já que um dispositivo que nega liberdade aos usuários vale menos que nada.

É possível usar o termo "hardware livre" como um equivalente conciso para "hardware feito de um *design* livre".

1.1.1 Hardware e Software

Hardware e software são fundamentalmente diferentes. Um programa, mesmo em forma executável compilada, é uma coleção de dados que podem ser interpretados como

instruções para um computador. Como qualquer outro trabalho digital, ele pode ser copiado e alterado usando um computador. Uma cópia de um programa não tem forma física ou forma de realização inerente. O hardware é uma estrutura física e sua estrutura física é essencial. Enquanto o *design* do hardware pode ser representado como dados, em alguns casos, mesmo como um programa, o *design* não é o hardware. Um *design* para uma CPU não pode executar um programa.

Além disso, embora seja possível usar um computador para modificar ou copiar o *design* de hardware, um computador não pode converter o desenho na estrutura física descrita. Isso requer equipamento de fabricação. Qual é o limite, em dispositivos digitais, entre hardware e software? Segue das definições: O software é a parte operacional de um dispositivo que pode ser copiado e alterado em um computador; hardware é a parte operacional física que não pode ser [6]. Essa é a interpretação mais correta para fazer tal distinção pois se relaciona com as consequências práticas.

Quanto a *Hardware Description Language* - Linguagem de Descrição de Hardware (HDL), o próprio código pode atuar como software (quando é executado em um emulador ou carregado em um FPGA) ou como um projeto de hardware (quando é realizado em silício imutável ou em uma placa de circuito).

1.2 Evolução Computacional

No mundo globalizado em que vivemos, cada vez mais, as empresas procuram por soluções que agreguem ao seu modelo de negócio, inevitavelmente, a tecnologia veio para atender isso e muitas outras coisas que necessitam de agilidade e exatidão. A cada dia podemos observar a evolução, por exemplo, na área da medicina, onde tem influenciado em vários avanços como mapeamento genético, processamento de imagens, processamento de DNA, raios-X, entre outros. Esses processamentos são bastante complexos e custosos exigindo uma grande quantidade de recursos como: capacidade de processamento, grande capacidade de armazenamento, saída e entrada de dados, processamento de áudio e imagens, segurança de dados, sistemas embutidos (embarcados) e indústrias com segmentos robotizados. Da mesma forma isso proporciona maior qualidade de vida com um diagnóstico mais ágil e seguro para as pessoas.

O avanço das arquiteturas de hardware gerou uma rápida evolução nos sistemas computacionais. Segundo Gordon Moore, um dos fundadores da Intel, a capacidade dos processadores dobra a cada 18 meses enquanto os custos dos chips permanecem constantes [7]. As placas de circuito eletrônicos estão cada vez menores e mais baratas, oferecendo máquinas com capacidade de multiprocessada mais acessíveis e menores, diferentemente dos primeiros computadores e mainframes que, eram máquinas de grande porte que che-

gavam a ocupar salas inteiras [8] e com elevados custos. A crescente necessidade de poder computacional das aplicações gerou a exigência de novos conceitos de tecnologia, novas metodologias, novas implementações e inovações nos modelos de arquitetura existentes para que essa demanda de processamento fosse atendida.

Nos dias de hoje, não há sequer um estabelecimento, pessoa, negócio que não se utilize de sistemas computacionais. Isso influencia a forma que os seres humanos vivem e interagem uns com os outros. A internet veio como uma rápida forma de comunicação entre duas pessoas que se encontram em diferentes partes do mundo, apesar dessa tecnologia vir para agregar valores, ajudar nos processos e aproximar pessoas, muitas vezes podemos ver ou sentir que isso tem deixado pessoas fisicamente próximas mais distantes. Quem nunca parou para olhar para o lado em uma mesa de jantar e ver seu amigos e familiares se conectando mundo afora e deixando de compartilhar experiências ali onde estava? Desde a preocupação das pessoas é visão que as pessoas têm sobre ela? Este é apenas um dos casos a serem pensados quando falados de evolução tecnológica. Tudo tem seus pontos positivos e negativos, cabe a nós seres humanos sabermos dosar a medida de até onde podemos chegar com isso.

Desde o ENIAC [8], primeiro computador digital, muitas coisas aconteceram no cenário das arquiteturas de Hardware, muitas delas relacionadas ao desempenho. É comum olharmos para um sistema computacional e achar que é apenas o processador que determinada se o computador é rápido ou não [1]. No entanto, é necessário levar em consideração tudo o que envolve a CPU, memórias, barramentos, coprocessadores, etc. O desempenho é diretamente relacionado a como cada um dos componentes interagem entre si, por exemplo, caso a memória principal tenha velocidade X e o processador 3X, na execução de um programa que se utiliza da memória o sistema como um todo será obrigado a rodar na velocidade X, ou seja, o processador estará usando apenas um terço do seu potencial pelo fato de ter que esperar pela memória principal.

Historicamente, o processador foi um componente que sofreu grande evolução desde a sua primeira concepção, com transistores cada vez menores os chips de computador diminuíram de tamanho e ficaram ainda mais potentes. Mas este alto desempenho do processador não foi acompanhado pelos outros componentes do computador como memórias e dispositivos de entrada/saída, que ficaram mais lentos em relação ao processador no decorrer do tempo [9]. Isso por um lado foi muito satisfatório para o poder de processamento mas por outro, resultou em uma grande lacuna entre as CPUs e os outros componentes dentro do computador. Como no exemplo anterior, o processador não tinha a oportunidade de usar todo seu potencial pelo fato de ficar esperando outros componentes muito mais lentos. É possível observar na Figura Figura 1.1 a disparidade na evolução entre processadores e memórias seguindo a Lei de Moore, enquanto o processador crescia

em média 55% por ano as memórias evoluíam lentamente, em média 7% por ano. .

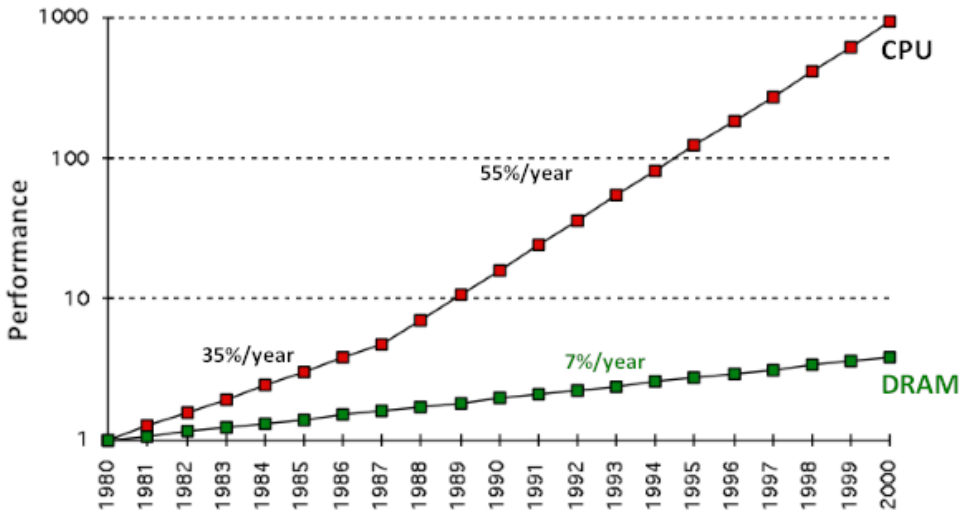

Figura 1.1: Evolução da performance entre 1980 e 2000 (Fonte:[1]).

1.3 VHDL

O *VHSIC Hardware Description Language* - Linguagem de descrição de hardware VHSIC (VHDL) é uma linguagem *Hardware Description Language* - Linguagem de Descrição de Hardware (HDL) que originalmente foi desenvolvida pela *Defense Advanced Research Projects Agency* - Agência de Projetos de Pesquisa Avançada de Defesa (DARPA) nos Estados Unidos com o objetivo inicial de documentação dos projetos desenvolvidos por terceiros e fornecidos às Forças Armadas. Por questões de segurança e sigilo no que tangem uma instituição militar, as empresas terceirizadas não possuíam acesso ao conteúdo total do projeto que tinha contribuições de mais de uma empresa. Como é possível imaginar, por ter várias diferentes empresas contribuindo em um mesmo projeto com diferentes metodologias e implementações de um circuito, gerava um grande custo e consumo de tempo no momento de sua unificação e interação. Dessa forma, surgiu a necessidade de criar uma linguagem de descrição de hardware para padronizar a documentação das tecnologias desenvolvidas pelas empresas parceiras, facilitando o desenvolvimento do projeto como um todo.

Inicialmente, o único objetivo era a padronização da documentação, no entanto, a DARPA já estaria cogitando a possibilidade de gerar circuitos lógicos a partir da descrição em VHDL. Por fim a linguagem se mostrou promissora para alcançar esse objetivo e, logo em seguida, surgiram alguns sintetizadores de VHDL capazes de gerar um *design* de

circuito escrito em VHDL e, consequentemente, simuladores para testar o comportamento que aquele circuito teria quando fabricado.

Isso tudo aconteceu nos meados do ano 1980 e já em 1987 a linguagem VHDL foi posta em domínio público, sendo padronizada pela IEEE (IEEE 1076). Em domínio público, como é de se esperar, a linguagem cresceu rapidamente, tendo uma nova versão lançada em 1993, alterações feitas em 2000 e 2002 e a última versão sendo lançada em 2008 - quase 12 anos da última versão e os sintetizadores comerciais ainda não suportam completamente esta versão [10].

1.4 Motivação

O RISC-V é um *Instruction Set Architecture* - Arquitetura do conjunto de instruções (ISA) modular e de código aberto, o projeto teve início em uma Universidade da Califórnia, em Berkeley, com o objetivo de ser um conjunto de instruções universal.

Fazer parte do processo de inovação no que tange a nova Arquitetura do Conjunto de Instruções gratuita e de código aberto. Sendo que a arquitetura RISC-V atualmente lidera tal inovação através da codificação colaborativa que teve muito apoio recentemente.

O conjunto de instruções RISC-V foi projetado para ser utilizado pela maioria dos sistemas computacionais, sendo adequada para uso em diversas aplicações diferentes, desde sistemas embarcados de baixa potência a supercomputadores de alta performance [11]. Por ser de certa forma bastante flexível possibilitando a utilização de instruções personalizadas e, módulos e extensões opcionais que podem ser acopladas para trabalhar em conjunto. Consequentemente, abrindo um vasto leque de possibilidades tanto para aplicações especificas quanto para trabalhos com o objetivo de melhorar o desempenho de um sistema embarcado.

1.5 Objetivos

Contribuir e acrescentar com o projeto RISC-V dentro da Universidade de Brasília, buscando construir um produto que possa ser usado para ensinar didaticamente como um processador funciona.

Implementar um processador RISC-V com pipeline de 5 estágios e uma memória cache para trabalhar junto com tal processador de forma a otimizar o desempenho do mesmo dentro do pipeline.

Capítulo 2

Conceitos Básicos

2.1 Características

Com tantos conjuntos de instruções disponíveis no mercado, qual o diferencial do RISC-V? Essa pergunta é essencial para entender essa arquitetura emergente.

O objetivo dos autores do RISC-V era de fornecer diversos *designs* de CPU diferentes sob a licença BSD. Tal licença permite que *designs* baseados em RISC-V possam ser implementados tanto de maneira aberta como proprietária, ao contrário de outros *designs* como ARM, que cobram taxas pelo uso de suas patentes além de requererem acordos de não-divulgação para liberação de suas respectivas documentações. Recentemente surgiu a *MIPS Open initiative* visando tambem o código aberto para projetos baseados no MIPS.

O seu caráter aberto também facilita um possível uso educacional, a escrita de compiladores e sistemas operacionais otimizados além da auditoria de segurança da arquitetura.

Desenvolver uma nova arquitetura de CPU requer um esforço conjunto de especialistas de diversas áreas, o que torna a criação de uma arquitetura aberta viável extremamente complicada. O sucesso do RISC-V foi possível graças ao trabalho de diversos especialistas e voluntários, o que de acordo com seus contribuidores o torna um projeto provindo do esforço comunitário.

2.2 Linguagens de Descrição de Hardware

Na Engenharia de Computação, uma das formas de se modelar circuitos eletrônicos são as *Hardware Description Language* - Linguagem de Descrição de Hardware (HDL). Tais linguagens permitem que se modele um sistema lógico complexo e por sua vez, são muito semelhantes com linguagens de programação convencionais como C e C++. Existem duas principais linguagens atualmente para a modelagem de circuitos digitais lógicos, são elas: Verilog e VHDL.

VHDL é a sigla de *VHSIC Hardware Description Language*, onde VHSIC é a sigla de *Very High Speed Integrated Circuits*; ou seja, em uma tradução livre, significa linguagem de descrição de hardwares para circuitos integrados de altíssima velocidade. Foi esta a linguagem utilizada para a implementação do produto resultante desta monografia.

O conceito de linguagem de programação é um tanto quanto abstrato pois, intuitivamente a linguagem de programação é aquela capaz de criar um programa - sequência de comandos que instruem e guiam a execução em hardware para efetuar uma determinada tarefa. Nesta definição, VHDL não é uma linguagem de programação pois, o resultado de um código VHDL não é um programa, mas um circuito eletrônico - na verdade um mapeamento de rotas que definirão o circuito quando gravado em um circuito integrado tal como ASIC ou FPGA.

2.3 FPGA

Field Programmable Gate Array - Arranjo de Portas Programáveis em Campo (FPGA) são dispositivos semicondutores baseados em uma matriz de blocos lógicos configuráveis (CLBs) conectados via interconexões programáveis. Os FPGAs podem ser reprogramados para os requisitos desejados de aplicação ou funcionalidade após a fabricação. Esse recurso distingue os FPGAs dos ASICs (*Application Specific Integrated Circuits*), que são fabricados sob encomenda para tarefas de *design* específicas. Embora FPGAs de programação única (OTP) estejam disponíveis, os tipos dominantes são baseados em SRAM, que podem ser reprogramados à medida que o projeto evolui.

Algumas das vantagens no uso de FPGAs são: Flexibilidade, Aceleração (*Time to Market*), Integração e Custo total de propriedade (TCO).

2.4 ISA

Uma arquitetura de conjunto de instruções (ISA) é um modelo abstrato de um computador. Também é conhecido como arquitetura ou arquitetura de computadores. Uma realização de um ISA é chamada de implementação. Uma ISA permite várias implementações que podem variar em desempenho, tamanho físico, custo monetário, entre outros; pelo fato de que a ISA serve como interface entre software e hardware. O software que foi escrito para uma ISA pode ser executado em diferentes implementações da mesma ISA. Isso permitiu que a compatibilidade binária entre diferentes gerações de computadores fosse facilmente alcançada assim como o desenvolvimento de famílias de computadores. Ambos os desenvolvimentos ajudaram a reduzir o custo dos computadores e aumentar

sua aplicabilidade. Por essas razões, a ISA é uma das abstrações mais importantes da computação atualmente.

Uma ISA define tudo que um programador de linguagem de máquina precisa saber para programar um computador. O que uma ISA define difere entre ISAs; em geral, as ISAs definem os tipos de dados suportados, o estado (como a memória principal e registradores) e sua semântica (como a consistência de memória e modos de endereçamento), o conjunto de instruções (o conjunto de instruções de máquina que compõe a linguagem de máquina de um computador) e o modelo de entrada / saída.

2.5 RISC-V ISA

O RISC-V possui quatro tipos de ISA base que serão comentados a seguir. Com o intuito de fundamentar o trabalho realizado, a ISA especificada para RISC-V se encontra no Anexo I para ser consultado a qualquer momento.

2.5.1 RV32I

O RV32I é a ISA base do RISC-V, ou seja, o conjunto de instruções que lida com inteiros de 32 bits e que contém outras instruções relacionadas a chamada e controle de acesso voltada para sistemas operacionais. Basicamente, a estrutura base contém um banco de registradores de 32 bits e um registrador adicional de 32 bits que armazena o PC (*Program Counter*).

A grande maioria das instruções utiliza a codificação em 32 bits independentemente do tamanho da palavra da arquitetura, seja ela 32, 64 ou 128 bits. As instruções base para qualquer implementação da arquitetura RISC-V podem ser encontradas no Anexo I.

O RV32I foi projetado para ser suficiente na formação de um objeto compilado e suportar ambientes modernos de sistema operacional. A ISA também foi projetada para reduzir o hardware necessário em uma implementação mínima. O RV32I pode conter até 47 instruções exclusivas. O RV32I pode emular praticamente qualquer outra extensão ISA (exceto a extensão A, que requer suporte adicional de hardware para atomicidade) [12].

2.5.2 RV32IE

O RV32E foi desenvolvido especialmente para ser utilizados em sistemas embarcados e micro controladores que possuem pouco recurso computacional.

A maior diferença entre do RV32E em relação ao RV32I é que o número de registradores de uso geral são reduzidos de 32 para 16, além disso o RV32E não implementa temporizadores e contadores.

2.5.3 RV64I

O RV64I é construído em cima do RV32I, dessa forma, as diferenças são bem sutis como no banco de registradores a diferença se dá apenas no tamanho do registrador, ou seja, ao invés de ter tamanho de 32 bits são de 64 bits. É como se essa arquitetura fosse uma adição a RV32I com algumas instruções que manipulam de forma mais eficiente dados de 32 bits (instruções com sufixo 'W') vide Anexo I.

Uma diferença interessante se dá na codificação das instruções onde em um primeiro momento, intuitivamente se o tamanho da palavra muda de 32 bits para 64 bits então as instruções da mesma forma mudariam. A verdade é que elas se mantém, ou seja, mesmo que a memória tenha por padrão o tamanho da palavra em 64 bits, as instruções continuam sendo codificadas em 32 bits.

2.5.4 RV128I

Da mesma forma que a arquitetura RV64I é construída em cima da arquitetura RV32I, a RV128I se constrói sobre a RV64I. Dessa forma, no banco de registradores tem se 32 registradores com tamanho de 128 bits. As instruções continuam com o tamanho de 32 bits ainda.

2.5.5 Extensões

O RISC-V além das ISAs base, possui uma série de ISAs de extensão definidas que são úteis em aplicações específicas. Essas estão brevemente listadas abaixo, maiores detalhes podem ser encontrados na especificação do RISC-V [12].

1. A - Instruções atômicas
2. C - Instruções comprimidas
3. V - Vetor
4. M - Multiplicação
5. Ponto Flutuante

   Precisão simples F

   Precisão dupla D

Precisão quádrupla Q

2.6 Hierarquia de Memória

A partir da demanda crescente por sistemas que lidam com grande quantidade de dados, é inevitável a procura por dispositivos que consigam guardar grande quantidade de informações. Claro, existem vários dispositivos que atendem a esse propósito, no entanto, existe uma Lei que atua nesse segmento: o custo por bit. Uma empresa que preza pelo seu patrimônio sempre há de se preocupar com seus custos, isso é um fator determinante para escolha de um dispositivo que irá atender à necessidade e que esteja dentro do orçamento.

É aqui que introduzimos o conceito de Hierarquia de Memórias. Um usuário comum de computador hoje em dia, preza tanto pela velocidade de acesso a seus arquivos, a quantidade que pode armazenar e, quando aplicável, a qualidade deles (imagens, vídeos, etc.). Como veremos a seguir, nem sempre é possível ter o melhor em todos os sentidos, o custo por bit é um grande impeditivo para a maioria das pessoas.

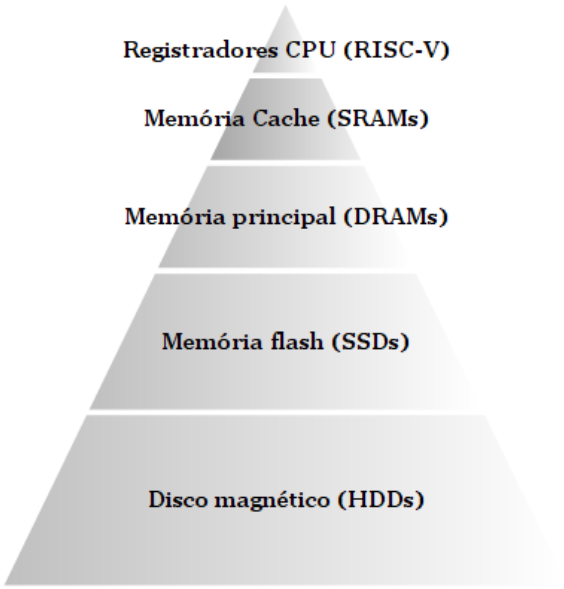

Figura 2.1: Hierarquia das memórias no geral.

A Figura 2.1 mostra de modo geral como é uma hierarquia de memória. Quanto mais alto na pirâmide, maior é o custo por bit, maior a velocidade de acesso, menor capacidade de armazenamento e menor o tamanho físico do componente. Analogamente, quanto mais baixo na pirâmide, menor o custo por bit, mais lenta a velocidade de acesso, maior capacidade de armazenamento e maior o espaço físico que o dispositivo ocupa. O escopo

deste projeto é otimizar a transmissão de dados entre os Registradores do RISC-V e a memória principal. A seguir, veremos resumidamente o quão rápidas são essas memórias e quanto suportam de armazenamento.

Os 3 níveis superiores dessa pirâmide se referem a memórias voláteis, isto é, células de memória que precisam constantemente de alimentação elétrica para guardarem informações. O restante são memórias não voláteis, ideais para casos que precisam armazenar dados mesmo que o dispositivo esteja desligado. Pela característica não volátil, são ideais também para backup de dados.

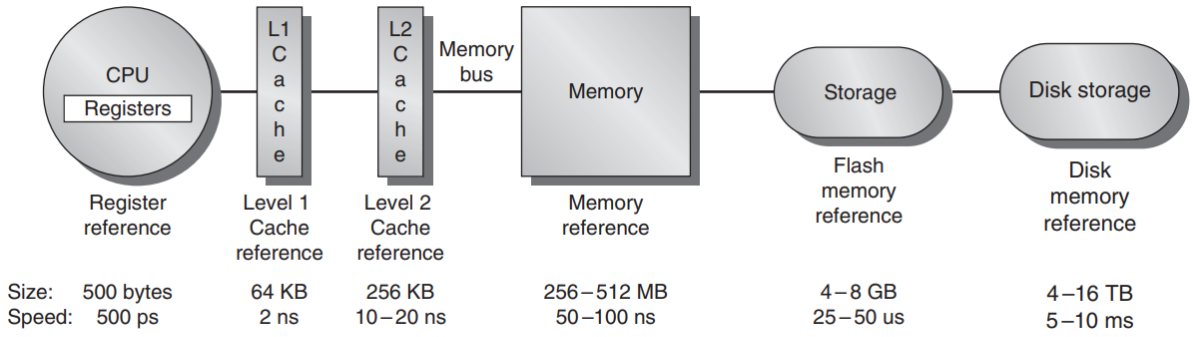

Figura 2.2: Tamanho e velocidade das memórias (Fonte:[2]).

Analisando a Figura 2.2 vemos que, a medida que nos afastamos do processador, a memória no nível abaixo fica mais lenta e maior. É importante se atentar que as unidades de tempo mudam por um fator de 10^9 - de picosegundos para milissegundos - e que as unidades de tamanho mudam por um fator de $10^(12)$ - de bytes para terabytes [2]. Essa é uma diferença enorme quando falamos de uma arquitetura que executa instruções na casa dos picosegundos.

2.7 Memória Cache

Na computação, um cache é um componente de hardware ou software que armazena dados para que solicitações futuras desses dados possam ser atendidas mais rapidamente; os dados armazenados em uma memória cache podem ser o resultado de uma computação anterior ou uma cópia de dados armazenados em outro lugar. Um acerto de cache (*Cache Hit*) ocorre quando os dados solicitados podem ser encontrados na cache, enquanto um erro de cache (*Cache Miss*) ocorre quando não é possível encontrar esse dado. Quando acontecem *Cache Hits* existe um ganho de desempenho, ou seja, é mais rápido do que recalcular um resultado ou ler de um armazenamento secundário de dados mais lento.

Dessa forma, quanto mais solicitações puderem ser atendidas pela cache, mais rápido será o desempenho do sistema como um todo. [2]

Para ter uma boa relação custo-benefício e permitir o uso eficiente de dados, os caches devem ser relativamente pequenos. No entanto, os caches provaram-se em muitas áreas da computação, porque os aplicativos típicos de computador acessam dados com um alto grau de localidade de referência. Esses padrões de acesso fazem com que a forma como a cache está estruturada tome vantagem.

A memória cache se utiliza de chips *Static Random-Access Memory* - Memória de Acesso Aleatório Estático (SRAM) que são um tipo de memória volátil, ou seja, precisam de constante energia para armazenar dados. Para otimizar a velocidade de acesso entre o processador e as outras memórias, a cache atua como uma intermediária na comunicação. A seguir, serão explicados alguns conceitos importantes ao entendimento deste trabalho.

Princípio da localidade

Os computadores em geral e qualquer dispositivo similar fazem constante uso do princípio da localidade, pelo fato de que a execução de um programa é, em sua maioria, sequencial, existe uma grande probabilidade de que a próxima instrução executada seja a instrução subsequente a atual. Vários programas executam funções em loops fazendo com que a mesma instrução seja executada repetidas vezes, portanto, seria extremamente útil que essa instrução ficasse disponível de forma fácil para consultas futuras. Dessa forma, os programas tendem a usar dados e instruções com endereços próximos ou iguais aos que eles usaram recentemente. A Figura 2.3 exemplifica visualmente esse conceito.

- Localidade temporal: itens acessados ou requisitados recentemente, provavelmente serão referenciados novamente em futuro próximo
- Localidade espacial: os itens com endereços próximos a ele tendem a ser referenciados também

2.7.1 Parâmetros de Cache

Nos dias de hoje, para diferentes necessidades de um sistema, existem diferentes tipos de tecnologia de cache, essas caches têm implementações distintas uma das outras. No momento em que uma empresa opta pela utilização de uma memória cache, ela precisa saber quais parâmetros de implementação atendem seu negócio da melhor forma. A Tabela 2.1 resume as escolhas possíveis que devem ser levados em conta em um desenvolvimento de memória cache.

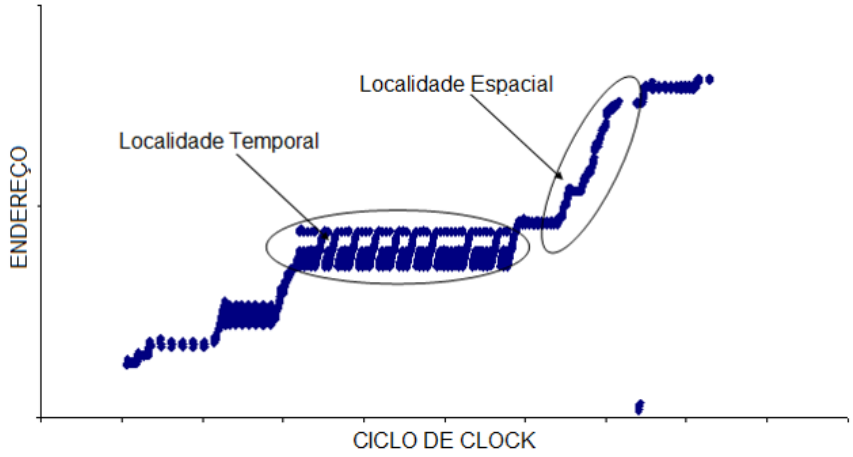

Figura 2.3: Exemplo de localidade temporal e espacial.

Tabela 2.1: Elementos de projeto de uma memória cache

Tamanho da memória cache	
Níveis de memória cache	Um nível (L1)
	Dois níveis (L2)
	Três níveis (L3)
	Quatro níveis (L4)
Tipo de mapeamento	Direto
	Associativo
	Associativo por conjuntos
Algoritmos de substituição	Primeiro a chegar primeiro a sair (FIFO)
	Menos recentemente usado (LRU)
	Menos freqüentemente usado (LFU)
Política de escrita (*Cache Hit*)	*Write-Throgh*
	Write-Back
Política de escrita (*Cache Miss*)	*No Write Allocate*
	Write Allocate

Mapeamento Direto

A função de mapeamento direto é simples e tem baixo custo de implementação [2]. Nessa técnica, cada bloco da memória principal é mapeado em uma única linha da cache. O endereço enviado pelo processador é usado como referência nesta técnica.

A idéia do mapeamento direto é associar blocos da memória a posições fixas na cache usando a equação $i = j \bmod m$ onde i é o número da linha da cache (onde o bloco vai ser armazenado), j é o número do bloco da memória principal e m é a quantidade de linhas da cache. A Figura 2.4 exemplifica como esse processo funciona.

O mapeamento direto é o método mais simples e rápido de projetar. A principal

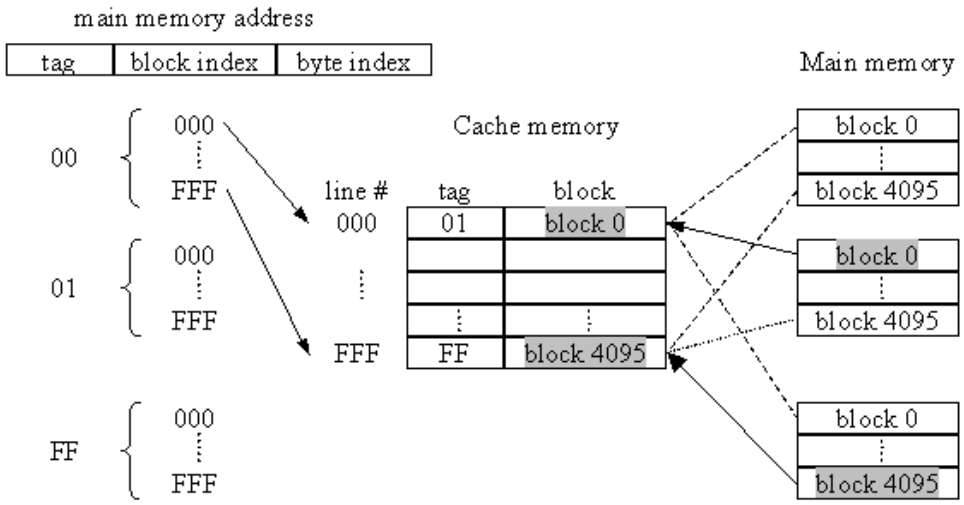

Figura 2.4: Exemplo de mapeamento direto em cache (Fonte:[3]).

desvantagem deste tipo de mapeamento é que diferentes linhas da memória principal devem ser mapeadas na mesma linha da cache. Para seguir o princípio básico da equação descrita anteriormente.

Mapeamento Associativo

A ideia aqui é compensar a desvantagem do mapeamento direto, que mapeia cada bloco da memória principal em uma única linha da cache. Esse tipo de mapeamento é vantajoso pois possibilita um uso mais otimizado do espaço da memória cache, ou seja, é possível mapear os blocos da memória principal em qualquer lugar da cache. Nessa modalidade, o endereçamento é feito usando apenas as tags que, são usadas para fazer a comparação no momento de requisição dos dados.

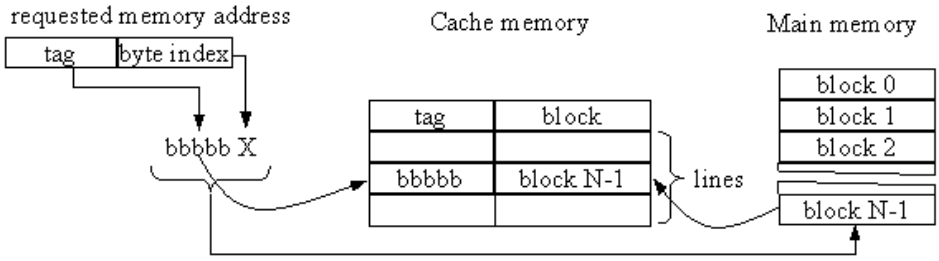

Figura 2.5: Exemplo de mapeamento associativo em cache (Fonte:[3]).

Quando a CPU solicita um novo endereço, o controlador compara simultaneamente todas as tags armazenadas na cache com o rótulo do endereço solicitado. A desvantagem

desse tipo de mapeamento se encontra no tamanho físico do módulo comparador, o circuito necessário para comparar todas as tags em paralelo são muito complexos e custosos. A Figura 2.5 exemplifica como esse processo funciona.

Neste caso, quando é necessário carregar um novo dado para a cache, é preciso se dar o uso de algum dos algoritmos de substituição. As escolhas mais comuns variam entre:

- FIFO: *First in first out*, ou seja, o primeiro dado a entrar será o primeiro dado escolhido para ser substituído quando houver necessidade.
- LFU: *Least frequently used*, ou seja, cada entrada da cache deverá possuir um contador que é incrementado a cada vez que aquela entrada for solicitada. Quando surge a necessidade de alocar um novo espaço na cache, o campo escolhido será aquele que contem a menor frequência de solicitações.
- LRU: *Least recently used*, ou seja, a cada ciclo de clock que se passa, todos os blocos ficam cada vez mais "velhos". É mantido um registro de a quantos ciclos de clock atrás foi a última solicitação daquela entrada. Quando surge a necessidade de alocar um novo espaço, a entrada escolhida será aquela que contém a maior idade, isto é, aquele a mais tempo na cache sem ser solicitado.

Mapeamento Associativo por conjuntos

Este tipo de mapeamento é um hibrido entre mapeamento direto e associativo, o objetivo aqui é atenuar as desvantagens dos outros enquanto alia as suas vantagens. Dessa forma, é possível armazenar em cache, não apenas um dado em qualquer lugar, mas um bloco inteiro deles. A Figura 2.6 exemplifica como esse processo funciona.

A substituição de entradas nesta cache é análoga a do mapeamento totalmente associativo. É necessário usar um algoritmo de substituição, mas, dessa vez, aplicando ao bloco inteiro de dados. A taxa de Hit do mapeamento associativo por conjuntos é significativamente maior que os mapeamentos direto e totalmente associativo [2].

O projeto desenvolvido nesta monografia usa a estrutura de mapeamento direto por ser mais simples e mais rápido de projetar do que os outros tipos de mapeamento (associativo e associativo por conjuntos). A implementação deste produto não visa alcançar o melhor desempenho nem avaliar os mapeamentos e algoritmos existentes, o propósito desta monografia é avaliar o impacto de elementos lógicos, dentro da FPGA, causado por diferentes tamanhos da cache e apresentar alguns parâmetros de avaliação para as mesmas.

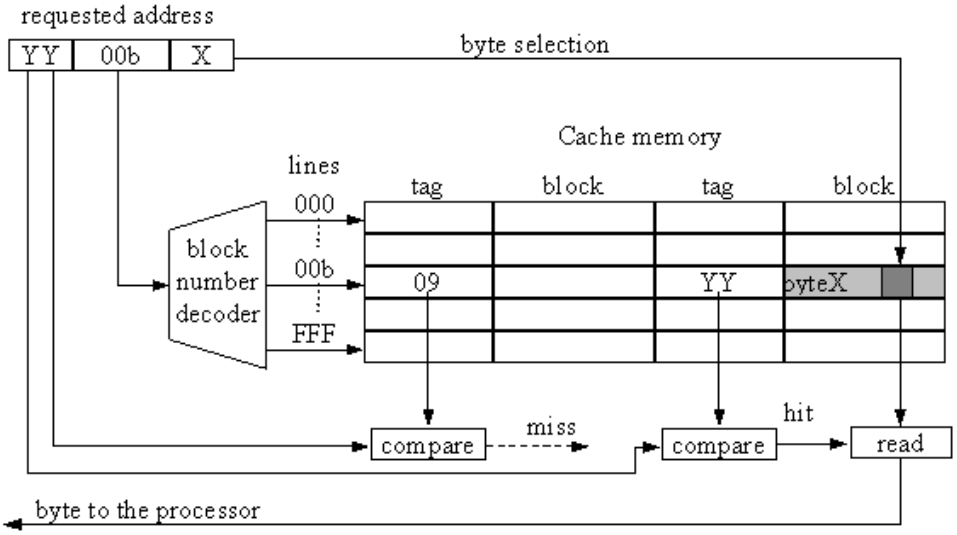

Figura 2.6: Exemplo de mapeamento associativo por conjuntos em cache (Fonte:[3]).

Capítulo 3

Implementação

Este capítulo descreve os detalhes da implementação do RISC-V com Cache. O código fonte pode ser encontrado em github.com/FabioTS/RISC-V_Cache.

3.1 RISC-V

3.1.1 Visão Geral

O processador desenvolvido para este trabalho é uma *Central Processing Unit* - unidade central de processamento (CPU) pipeline da ISA de código aberto RISC-V. O processador RISC-V foi desenvolvido completamente para o modelo definido no contexto deste trabalho, inicialmente com a intenção de incluir todas as extensões implementáveis da ISA - RV64G que representa a implementação do RISC-V com as extensões ("IMAFD"). Porém, como será discutido a seguir, estas não foram levadas a frente e o núcleo permaneceu somente com as instruções já previstas na ISA padrão.

Para esse trabalho foi escolhida uma variação do conjunto de instruções definidas no *RV32I Base Instruction Set* cujas instruções disponíveis podem ser consultadas no Anexo I de referência. Foram omitidas as instruções do tipo FENCE usadas para controle de *threads*, visto que as mesmas não são necessárias nesse trabalho, bem como as do tipo CSR, usadas para ler e escrever em registradores de sistema. Cada módulo dentro do projeto foi individualmente projetado e testado, ou seja, a implementação não contém uso de *designs* de terceiros, a não ser aqueles disponibilizados livremente pela própria Intel (*IP Cores*) para as FPGAs. Os únicos *IP Cores* usados no projeto foram as memórias ROM e RAM (*On Chip Memory*) para lidar com os blocos M9K de memória disponíveis na FPGA.

A seguir, serão apresentados os detalhes da implementação junto com as ferramentas utilizadas.

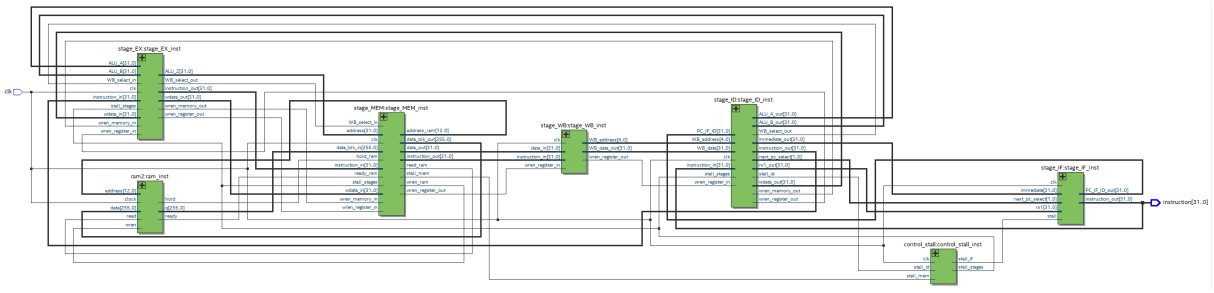

Figura 3.1: Visão geral RTL do projeto RISC-V.

A Figura 3.1 representa a construção geral do projeto e os sinais de comunicação entre os módulos . O processamento se dá através de uma estrutura pipeline de 5 estágios que serão então explicados.

1. *stage_IF*: *Instruction Fetch*
2. *stage_ID*: *Instruction Decode*
3. *stage_EX*: *Execute*
4. *stage_MEM*: *Memory stage*
5. *stage_WB*: *Write back*

3.1.2 Instruction Fetch

Esse é o estágio primordial do pipeline. O PC, *Program Counter*, é um registrador de controle que guarda o endereçamento a partir do qual, as instruções são lidas da memória de instruções e consequentemente passadas ao próximo estágio para decodificação.

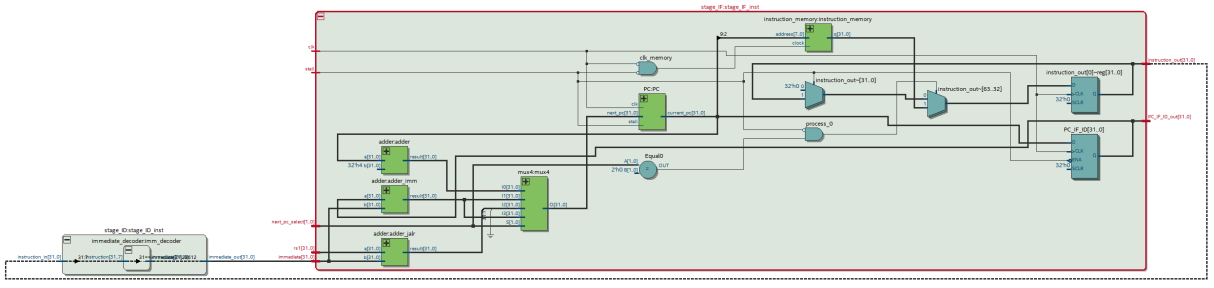

Figura 3.2: Visão geral RTL do módulo stage_IF.

Também é crucial o recebimento dos sinais provenientes do componente *Jump Control* que a partir da central de controle localizada no estágio seguinte, recebe sinais de controle para determinar se a instrução anterior, é um *Jump* ou um *Branch* e, toma as medidas

necessárias para ignorar a instrução apontada naquele momento pelo PC e calcula o endereço da instrução a qual o *Jump* ou *Branch* aponta. Ainda dentro deste processo, altera o PC para seguir com a correta execução do programa.

3.1.3 Instruction Decode

Este é o estágio que possui o maior número de componentes dentro do projeto. Temos 7 módulos que desempenham funções essenciais dentro de todo o processo do pipeline. É aqui que se encontra o banco de registradores e a central de controle que decodifica cada parte de uma instrução. Nesse estágio é feito também a previsão de saltos, ou seja, a partir da instrução decodificada é possível determinar qual será a próxima instrução a ser executada.

Ainda no escopo desse estágio, está o controle que detecta se uma instrução usa o resultado gerada por outra instrução, mas que ainda está sendo calculada nos estágios seguintes. Exemplo: add x3, x2, x1 -> add x4, x3, x1. Neste caso são duas operações de soma onde a segunda usa o resultado da primeira, porém, para o resultado ser gravado em x3, são necessários 3 ciclos. Para que o resultado da segunda operação seja correto e mais rápido, a implementação desse controle adianta o resultado do estágio de execução, possibilitando que essa segunda operação possa ser executada após apenas 1 ciclo, antes mesmo do resultado x3 ser gravado no banco de registradores.

A implementação disso se dá por uma fila de registradores e resultados do estágio de execução, guardando informações de quais e como os registradores serão alterados nos próximos 3 ciclos. Isso dá a possibilidade de comparar os operados de uma instrução com a fila, aumentando a performance de todo o processo do pipeline.

3.1.4 Execute

É no estágio de execução que se encontra a Unidade Lógica e Aritmética (*Arithmetic and Logic Unit - ALU*) que, de certa forma, é onde acontece a maior parte da computação. Toda parte de cálculo é realizada nesse estágio, onde a ALU recebe dois operandos, o shamt (*Shift Amount*) e um sinal derivado do controle que determinada qual será a operação realizada entre estes operandos.

A Unidade Lógica e Aritmética é responsável por operações de adição e subtração, assim como operações booleanas (and, or, xor). Geralmente este estágio também contém um modulo chamado *Bit Shifter* mas, no caso deste produto, as operações de deslocamento e rotação (sll, srl, sra) foram integradas junto a ALU com os devidos sinais de controle.

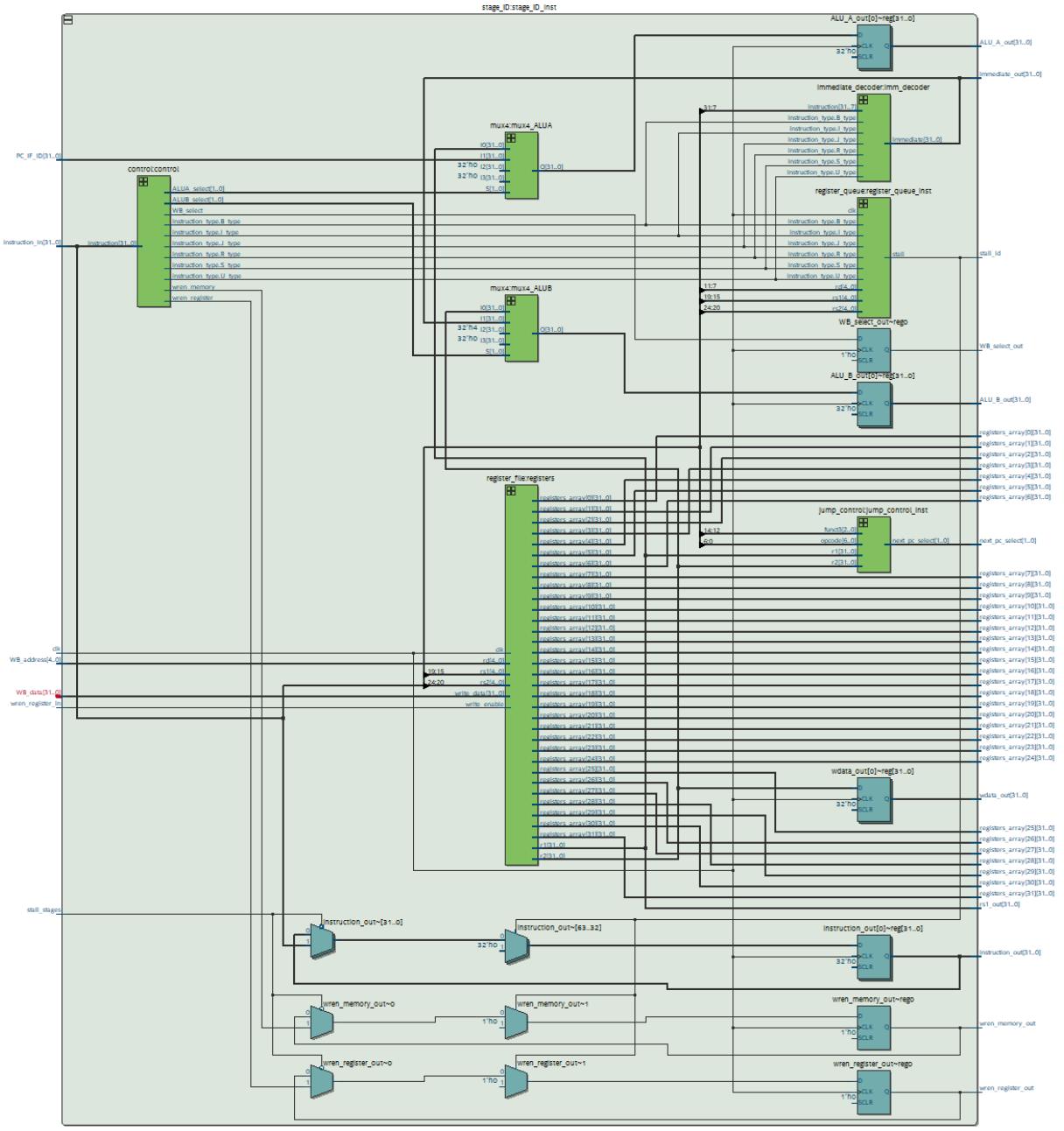

Figura 3.3: Visão geral RTL do módulo stage_ID.

3.1.5 Memory

É neste estágio que a instrução decodificada que solicita acesso de leitura ou escrita na memória a acessa. No caso de o acesso não ser requisitado, a instrução é simplesmente passada ao próximo estágio.

Historicamente, o estágio de memória é aquele que toma o maior tempo dentro do pipeline. Enquanto a velocidade de uma operação básica de adição e a frequência de clock dos processadores melhoraram constantemente nas décadas de 1980 e 1990, a melhoria

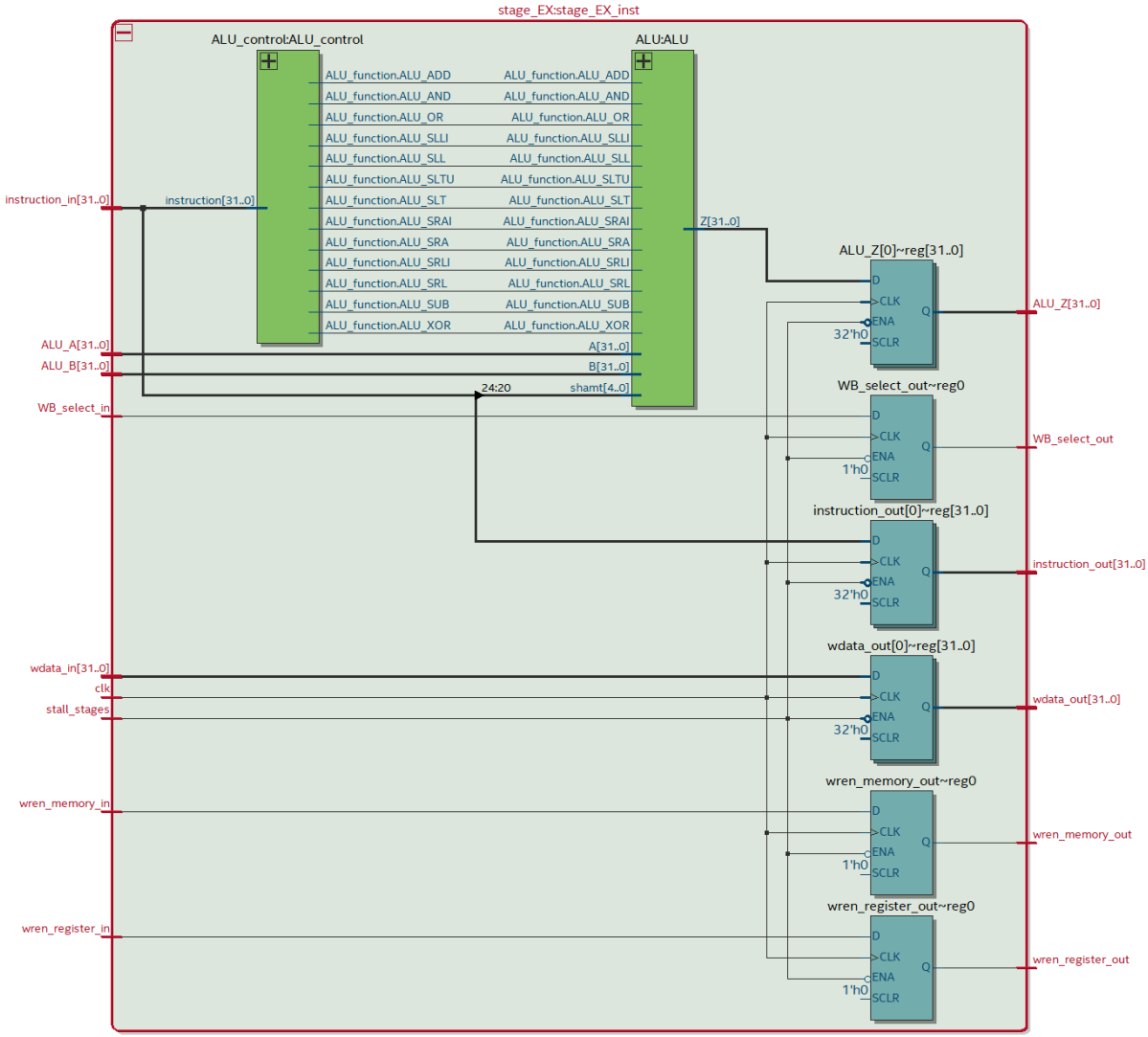

Figura 3.4: Visão geral RTL do módulo stage_EX.

no desempenho do sistema como um todo em sua maioria, foi limitada pela velocidade de comunicação entre o processador e outros componentes no sistema. Em particular, a latência do acesso à memória e a largura de banda da memória não melhoraram proporcionalmente. Isto foi referido como a Memory Wall [13] é aqui onde a maior esforço de otimização com o uso da hierarquia de memória se encontra.

3.1.6 Write back

Esse estágio simplesmente garante que os dados serão escritos nos registradores por apenas uma instrução por vez e, sempre no mesmo ciclo de clock para todas elas.

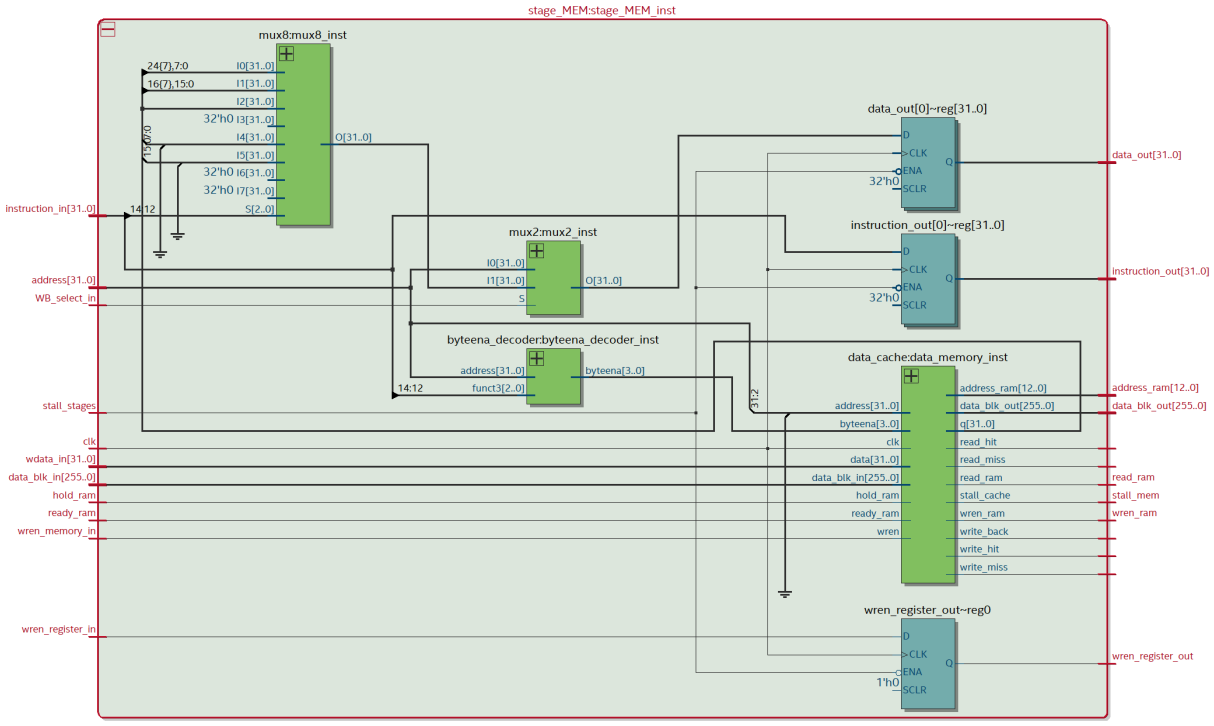

Figura 3.5: Visão geral RTL do módulo stage_MEM.

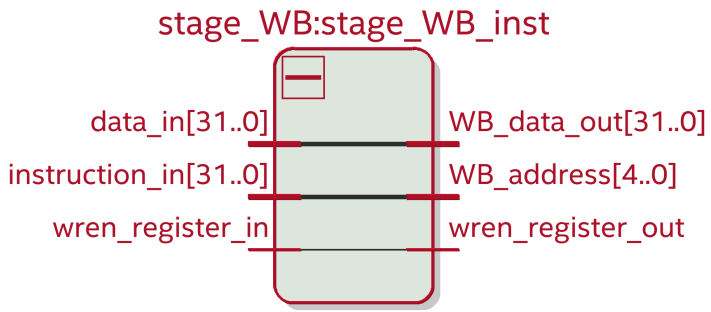

Figura 3.6: Visão geral RTL do módulo stage_WB.

3.2 Cache

A escolha dos parâmetros definidos na implementação da Cache foi baseada na multiplicação de matrizes, um algoritmo que se utiliza de vetores. Dessa forma, a grande maioria dos acessos em memória são sequenciais, ou seja, alocados lado a lado na memória. Isso volta ao conceito de localidade vista anteriormente.

A partir da Figura 3.7, é possível ver os componentes que fazem parte da memória cache implementada. Basicamente, contém um array que guarda as informações e um controle que, por sua vez, contém os sinais de controle e a tabela de tags referente aquela memória.

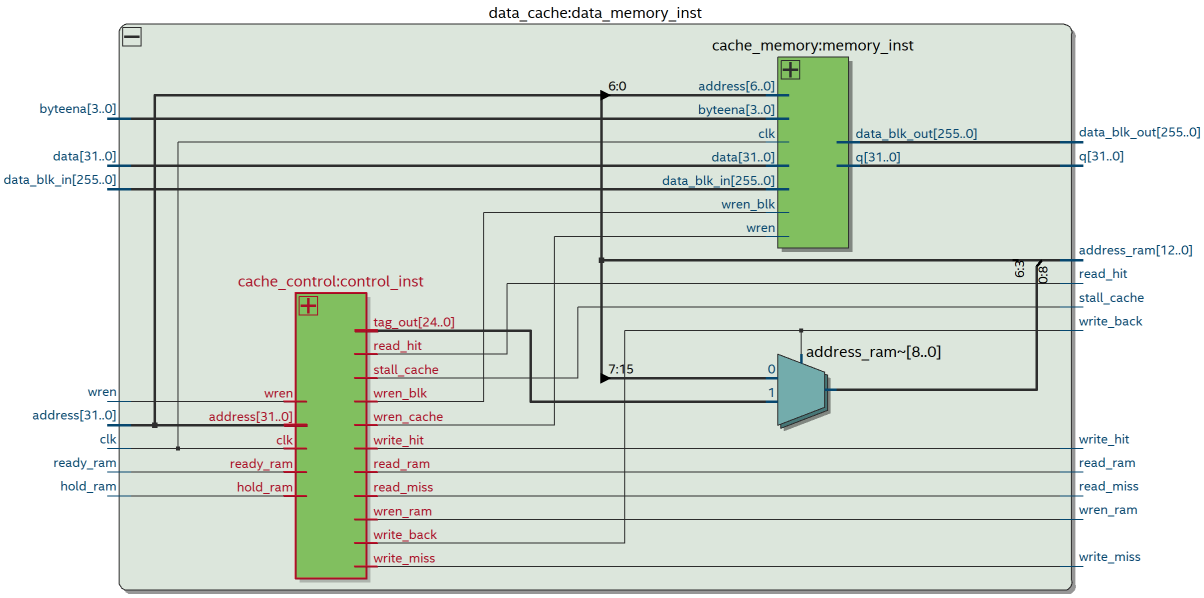

Figura 3.7: Visão geral RTL do módulo data_cache.

O tamanho da memória cache é parametrizado, ou seja, é possível alterar os parâmetros no pacote de constantes "constants.vhd". Isso é bastante útil pois permite que se gere várias versões de forma a comparar o desempenho de acordo com o tamanho da cache. A partir dos conceitos básicos já vistos na secção anterior, serão detalhados a seguir os parâmetros escolhidos na implementação deste produto.

3.2.1 Controlador de cache

O módulo *cache_control* foi desenvolvido usando como base o conceito de máquina de estados. Existem 5 estados possíveis que são definidos a partir das variáveis do sistema, ou seja, analisando o endereço requisitado pela instrução e consultando a tabela de tags para determinar qual ação deve ser tomada para atender a requisição:

- *rh* (Read Hit): Esse estado é ativado quando não é uma operação de escrita e a linha de cache correspondente estiver válida e com a mesma tag do endereço solicitado. Aqui não há interrupção de execução.
- *rm* (Read Miss): Esse estado é ativado quando não é uma operação de escrita e a linha de cache correspondente estiver inválida ou com uma tag que não é a do endereço solicitado. Aqui o sinal Stall do pipeline é ativado e o controle busca a informação na memória principal.
- *wh* (Write Hit): Esse estado é ativado quando é uma operação de escrita e a linha de cache correspondente estiver válida e com a mesma tag do endereço solicitado.

Neste caso, a linha também é marcada como Dirty. Aqui não há interrupção de execução.

- *wm* (Write Miss): Esse estado é ativado quando é uma operação de escrita e a linha de cache correspondente não estiver válida ou com uma tag que não é a do endereço solicitado. Aqui o sinal Stall do pipeline é ativado e o controle busca a informação na memória principal. Neste caso, após a alocação na cache, é seguido de um Write Hit.
- *wb* (Write Back): Esse estado é ativado quando é uma operação de escrita e a linha de cache correspondente estiver com uma tag que não é a do endereço solicitado e suja, isto é, marcada como Dirty. Neste caso, após a escrita do conteúdo daquela linha de volta na memória principal, a linha é desmarcada como Dirty e segue um Write Miss.

3.2.2 Mapeamento direto

Foram utilizados os conceitos de mapeamento direto para esta Cache. A quantidade de entradas na cache foi implementada de forma modular, ou seja, é possível escolher o tamanho do bloco e a quantidade de blocos. O tamanho da cache resultante é calculada multiplicando o tamanho da palavra, o tamanho e quantidade de blocos. No cenário deste produto, o tamanho padrão da cache é de $0,5KiB$, ou seja, palavra de 32 bits, tamanho do bloco de 8 palavras e 16 blocos no total ($32 \times 8 \times 16 = 4096bits = 512bytes$). É possível ver na Tabela 3.1 uma visão geral da implementação.

Tabela 3.1: Estrutura genérica da Memória Cache desenvolvida

	[0,1]	*[0,1]*	*[22..0]*	*[31..0]*	*[31..0]*	*...*	*[31..0]*	*[31..0]*
	Valid	**Dirty**	**Tag**	**Data [Offset]**				
				000	**001**	**BLK_SIZE**	**110**	**111**
0000	0	0	X	X	X	...	X	X
0001	1	0	..01	0A	0B	...	0C	0D
0010	1	0	..00	01	02	...	04	08
0011	0	0	X	X	X	...	X	X
0100	1	1	..10	AA	BB	...	55	EE
...	...	...	...	...	...	...	...	...
N_BLK-1	*N_BLK-1*	*N_BLK-1*	*N_BLK-1*	*N_BLK-1*	*...*	*BLK_SIZE*	*...*	*N_BLK-1*

3.2.3 Cache de dados e de instruções

O primeiro nível de cache em um processador, em sua maioria, consiste de duas memórias separadas, uma que guarda instruções e a outra, dados. Esse produto foi construído pensando na memória de dados e, portanto, apenas a cache referente ao armazenamento de dados foi implementada. Analogamente, a cache de instruções é uma variação da

primeira. Essa por sua vez consiste apenas em operações de leitura de instruções e, consequentemente, não se faz necessário o bit de controle Dirty.

3.2.4 Write Allocate

A politica de escrita para cache de dados foi o Write-Allocate, que, como dito anteriormente, gera primeiramente um Write-Miss e logo em seguida um Write-Hit. Isso se deve ao fato de que o processador sempre ler e escrever apenas na cache.

Essa política de escrita é comumente usada com Write-back explicado na seção 3.2.5. É possível seguir detalhadamente o fluxo resultante dessas duas políticas atuando em conjunto pelo diagrama representado na Figura 3.8.

3.2.5 Write Back

A cada escrita na Cache de dados, o controlador de cache muda o bit de controle Dirty para verdadeiro para aquele bloco. Dessa forma, posteriormente quando aquele bloco precisar ser substituído, ou seja, um outro bloco da memória precisar ser mapeado a este bloco da cache "sujo", é preciso antes escrevê-lo na memória principal. Esta é a definição da política *Write-Back*.

Quando é gerado um *Read-Miss* ou *Write-Miss*, o bit de controle Dirty é sempre verificado e a operação é redirecionada caso ele seja verdadeiro. Neste caso, o sinal *Stall* é ativado e a CPU aguarda até que o controlador de memória cache envie o bloco de memória "sujo" que, por sua vez, aguarda a escrita ser realizada na memória secundária. Após realizado este procedimento, o bloco com o bit "sujo" é limpo e a operação que ativou o *Write-Back* volta a ser executada.

É comum que nesta política de escrita, os dados serem escritos em memória secundária sempre depois de um certo número de ciclos. Isso para que seja mantida a consistência da memória em casos onde outros componentes possam acessá-la. No caso em questão, isso não se faz necessário pois não é um sistema multiprocessado, ou seja, o único componente a acessar a memória é o RISC-V.

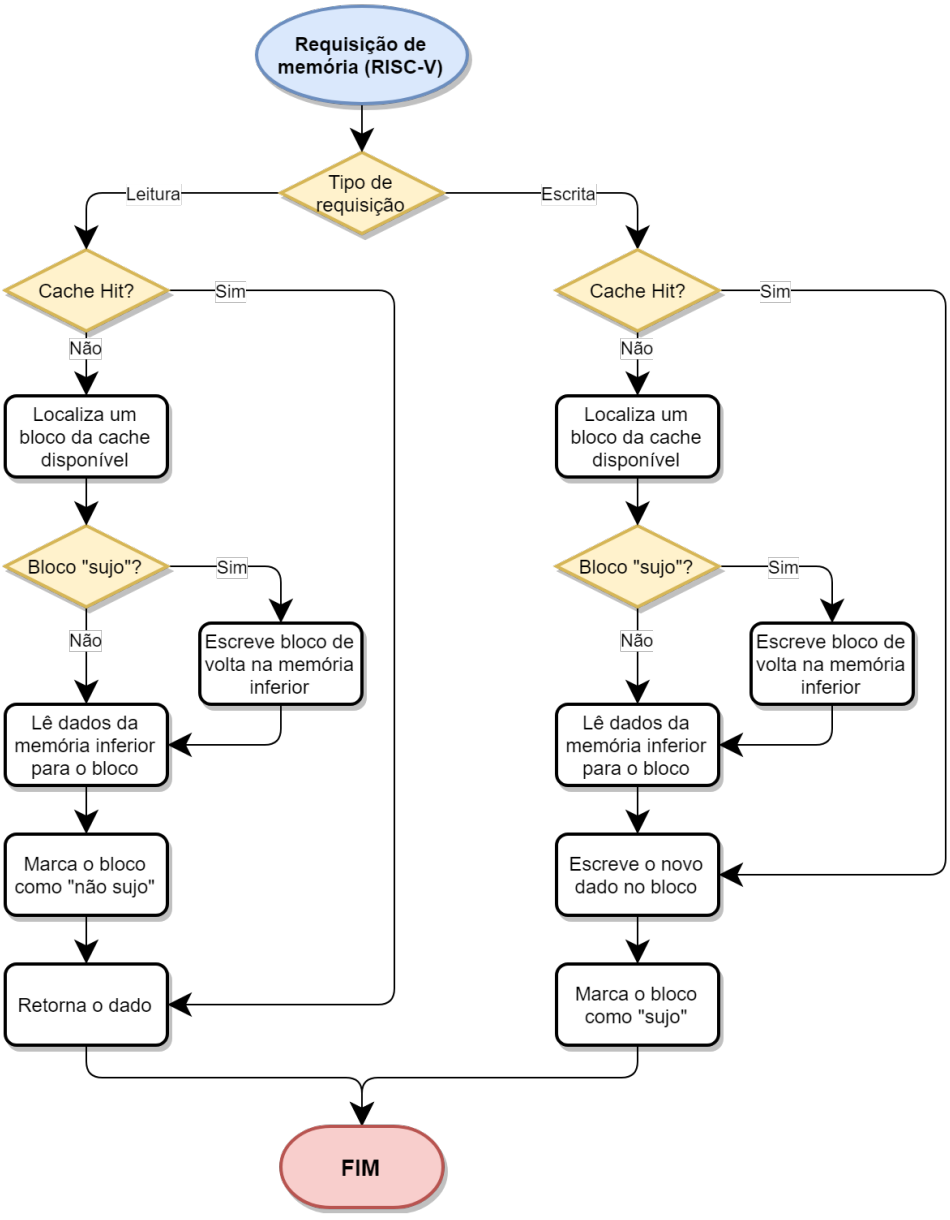

Figura 3.8: Diagrama que representa o fluxo de estado da cache com as políticas Write allocate e Write back atuando em conjunto.

Capítulo 4

Verificação

A partir do ciclo de desenvolvimento de um projeto, é possível entender a dificuldade da verificação de um design. Dados estatísticos mostram que cerca de 70% do desenvolvimento de um projeto em hardware é dedicado à verificação do mesmo [14].

A verificação de um *design* é indispensável para a validação. Existem várias empresas capazes de produzir um circuito que desempenha as mesmas funções. Um exemplo são as fabricantes de circuitos integrados CMOS e TTL que, para uma mesma especificação, produzem seus circuitos integrados. Agora, como decidir qual adquirir, ou melhor, qual desses será que funciona corretamente no meu *design*? Essa é uma pergunta que em grande parte é respondida pelo resultado de testes. Quando uma empresa adquire um equipamento, ela exige que ele cumpra o que está especificado e, para provar isso, inúmeros testes são feitos para sua validação.

Não poderia seria diferente neste produto então, resumidamente, cada componente presente no *design* do RISC-V com Cache foi testado e verificado individualmente. No entanto, neste capítulo será focado apenas os módulos principais do RISC-V e Memória Cache.

Geralmente em projetos VHDL, os testes são feitos através de *testbenches* que, são escritas na mesma linguagem para gerar estímulos a um módulo chamado comumente de *Device Under Test* - Dispositivo em teste (DUT). Uma outra possibilidade, por questões de tempo, é a ferramenta chamada VUnit **??** que, se utiliza de testes unitários automatizados fazendo com que o projetista ganhe tempo na correta elaboração do *design*.

No escopo deste projeto, os dois métodos foram utilizados onde, para verificar os módulos principais (estágios do pipeline) a ferramenta VUnit foi utilizada. Já para a memória cache, foram criadas *testbenches* com todos os tipos de operações tangíveis a cache e inseridos de forma a agilizar o processo, verificações *Assert*.

Como teste de integração das duas partes, foi compilado um algoritmo de multiplicação de matrizes usando a ferramenta de código livre *RISC-V Assembler and Runtime*

Simulator - Montador e Simulador de tempo de execução do RISC-V (RARS) [15]. Para carregar os dados de instruções e de dados no projeto, foi utilizada um software gratuito *Freeware Hex Editor and Disk Editor* - Editor Hexadecimal e Editor de Disco Gratuito (HxD) [16] para auxiliar no mapeamento de acordo com o tamanho dos blocos de memória. O formato de arquivo usado foi o Intel Hex [4].

4.1 Ferramentas Utilizadas

4.1.1 Quartus Prime 18.1

O software Quartus é uma IDE para desenvolvimento em linguagem HDL desenvolvido pela Intel para os dispositivos FPGA. Este é o ambiente padrão de *design* de hardware da Intel que contém diversas ferramentas para auxiliar tanto no desenvolvimento de projeto quanto em otimização e verificação. Este também é responsável por compilar o código VHDL, configurar e carregar na FPGA o circuito sintetizado.

Modelsim Altera

Essa é a principal ferramenta que acompanha o Quartus para execução de bancada de testes, verificação e depuração do *design* de hardware. Seja ele escrito em VHDL ou *Verilog*. Apesar de ter desempenho reduzido na versão utilizada, a visualização em forma de ondas e *Asserts* são bastante úteis para seguir o correto fluxo de execução. É possível identificar na Figura 4.1 um dos testes feitos com o processador mostrando o fluxo de instruções através do pipeline.

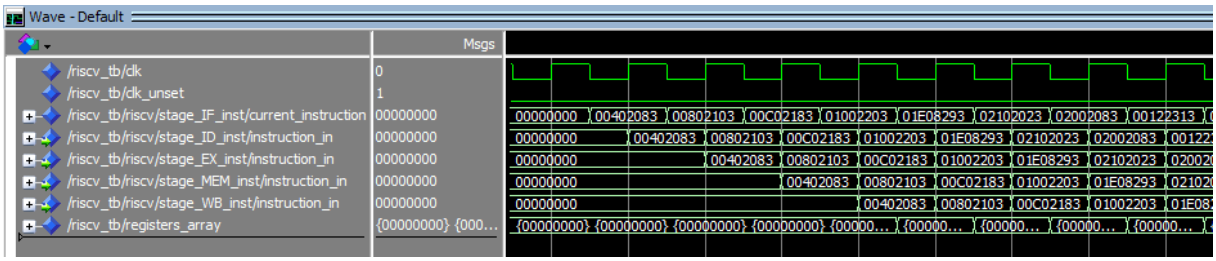

Figura 4.1: Simulação feita no Modelsim Altera para RISC-V.

4.1.2 Sigasi Studio

Esta é também uma IDE, no entanto, olhando pelo lado da codificação apenas, este conta com muito mais recursos quando comparado ao Quartus. O Sigasi Studio foi construído em cima do Eclipse - famoso software para Java - e conta com recursos como: *Auto*

complete e *Type-time linting* que ajudam e agilizam bastante o processo de construção do código. Foi usada a licença educacional disponível para estudantes da área.

4.1.3 RARS

O *RISC-V Assembler and Runtime Simulator* - Montador e Simulador de tempo de execução do RISC-V monta e simula a execução de programas na linguagem de montagem (*assembly*) do RISC-V [15]. Seu principal objetivo é ser um ambiente de desenvolvimento efetivo para pessoas começando com o RISC-V. Essa ferramenta bastante útil, apesar de estar ainda sendo desenvolvida, foi indispensável na elaboração de testes a CPU proposta. É possível escrever, executar e exportar o código de máquina em alguns formadas inclusive Intel Hex. A versão utilizada foi a v1.2.

HxD

Este software gratuito foi a solução encontrada para corrigir uma pequena incompatibilidade entre o RARS e a estrutura de memória dentro da implementação. Para carregar os dados na memória RAM, o Quartus lê um bloco por vez, como a memória ram escolhida possui blocos de 8 palavras, é necessário que o arquivo de carregamento da memória Intel Hex possua n=8 como pode ser visto a seguir. Portanto, essa ferramenta permite importar um arquivo Intel Hex gerado pelo RARS e depois exporta-lo da forma desejada já sendo feitos os cálculos para o campo **CHKSUM**.

Intel Hex

O arquivo de dados Intel Hex, fornece um conjunto de dígitos hexadecimais que representam o código ASCII para bytes de dados que compõem uma parte da imagem da memória [4]. É possível ver a partir da Figura 4.2 a especificação do formato de uma linha do arquivo de memória. Este formato é bastante utilizado nos dias de hoje para diversas aplicações e circuito que necessitam de inicialização de memória diretamente a partir de um arquivo externo.

Data Record (8-, 16-, or 32-bit formats)

RECORD MARK ' : '	RECLEN	LOAD OFFSET	RECTYP '00'	DATA	CHKSUM
1-byte	1-byte	2-bytes	1-byte	n-bytes	1-byte

Figura 4.2: Estrutura de dados para arquivo Intel Hex (Fonte:[4]).

Os campos são explicados a seguir:

- *RECORD MARK*: Este campo sempre contém a constante "03AH", a codificação hexadecimal do caractere de ASCII (':').
- *RECLEN*: O campo contém dois dígitos hexadecimais ASCII que especificam o número de bytes de dados no registro. O valor máximo é "FF". É aqui que o 'n' igual a 8 é apresentado.
- *LOAD OFFSET*: Este campo contém quatro dígitos hexadecimais ASCII representando o deslocamento dos dados.
- *RECTYP*: Este campo contém a constante "03030H", a codificação hexadecimal do caractere ASCII '00', que especifica o tipo de linha para ser um registro de dados.
- *DATA*: Este campo contém os bytes de dados propriamente ditos, representados em pares de dígitos hexadecimais ASCII.
- *CHKSUM*: Este campo serve para identificar se a linha está com dados válidos, contém a soma de verificação dos campos anteriores.

Capítulo 5

Resultados

5.1 Utilização de recursos

O projeto foi desenvolvido usando como base o kit de desenvolvimento com chip Cyclone IV FPGA DE2-115 da Altera. Utilizando o software Quartus, foi feita a síntese do *design* e compilação completa para a FPGA de referência. A partir da Figura 5.1 o número de elementos lógicos da placa usados foi igual a 13.264, cerca de 12% da capacidade prevista para essa FPGA. É valido avaliar que, ainda que nessa placa de prototipação o espaço ocupado seja de apenas 12%, o número de elementos ainda pode ser maior do que alguns componentes no mercado poderiam suportar. A diferença entre uso de recursos na implementação com e sem a memória Cache pode ser observada na Tabela 5.1, vale ressaltar que na implementação com Cache a disparidade no uso de bits de memória é resultado do componente "ram" - um *On Chip Memory* de tamanho $256KiB$.

Tabela 5.1: Utilização de recursos da FPGA

FPGA	**RISC-V**	**RISC-V com Cache**
Nº de elementos lógicos	4,007 (4%)	13,264 (12%)
Nº de registradores	1,337	5,545
Nº de bits de memória	16,384 (0,4%)	2,105,744 (53%)

A utilização dos blocos de memória da FPGA foram cerca de 53% do disponível, isso resulta em aproximadamente $500KiB$ de armazenamento. Esse número é suficiente para programas simples com "pouco"volume de dados. Para casos onde há grande volume de dados, é necessário armazenamento externo e, isso requer comunicação via barramento com os devidos controladores elaborados. Como o escopo do trabalho se resume apenas a aplicações simples, não foram implementadas rotinas de comunicação com dispositivos externos, embora isso seja essencial para aplicações mais robustas.

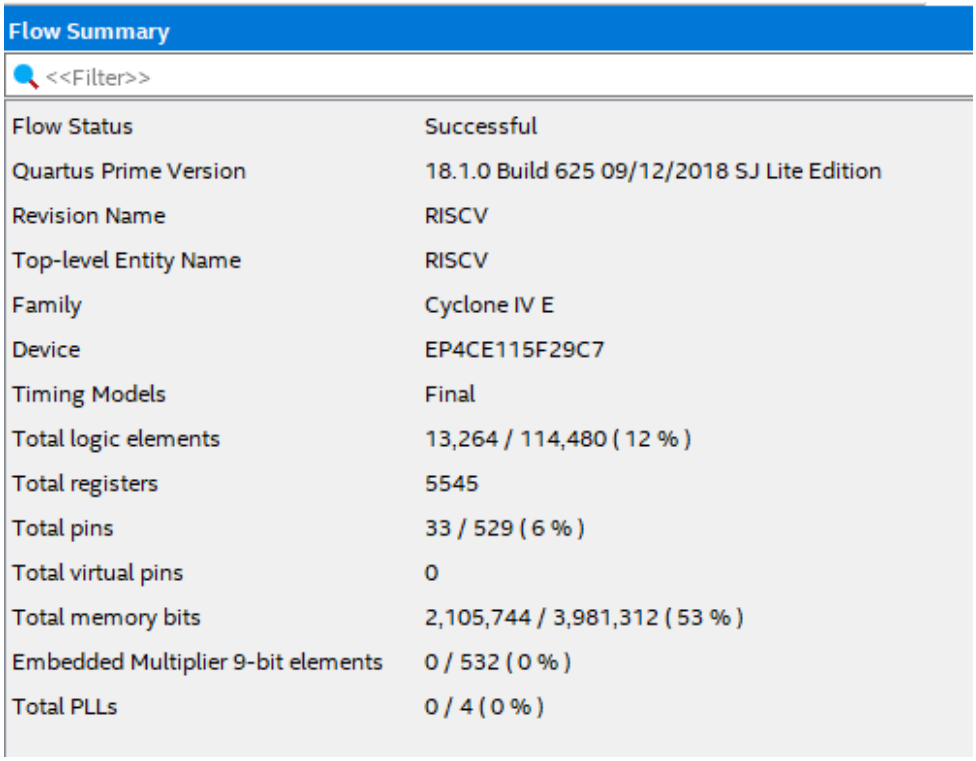

Figura 5.1: Resultado gerado pelo Quartus na compilação do projeto.

5.2 Desempenho

Para avaliar o desempenho da memória cache implementada, foi utilizado um algoritmo de busca binaria em dois casos: com e sem a memória cache. O parâmetro utilizado para essa avaliação foi o número de ciclos de clock gastos para achar o elemento. De acordo com o algoritmo de busca binária, os elementos do vetor devem estar ordenados em ordem crescente. O vetor onde foi feita a busca contém 1000 elementos armazenados em memória. A diferença no tempo de acesso entre processador e memória é que irá dizer qual o ganho de desempenho proporcionado pela cache. O tamanho da memória cache utilizada neste caso foi o padrão do projeto ($N\_BLK = 16, BLK\_SIZE = 8$) com blocos contendo 8 palavras de 4 bytes e 16 linhas de endereçamento.

Foi possível comprovar que o *delay* de barramentos de comunicação entre a CPU e a Memória, fazem com que o processador acumule muito tempo ocioso. Aqui é onde a Hierarquia de memória apresentada na Seção 2.6 entre em ação, como os dados de uma busca passada já estão presentes na cache, buscas posteriores são feitas em apenas um ciclo de clock caso o endereço esteja no mesmo bloco armazenado.

Uma outra forma de verificação bastante pertinente que é possível fazer é a comparação de desempenho entre READ HIT e READ MISS, de acordo com diferentes tamanhos

especificados para a cache, para um mesmo algoritmo descrito. Isso é o que veremos na próxima secção.

Métricas de desempenho de cache

Alguns outros conceitos são importantes e serão introduzidos em seguida, cabe ressaltar que esses parâmetros são os padrões básicos para determinar a performance da cache. Esses números variam bastante de acordo com o tamanho da cache, ou seja, a quantidade de palavras que um bloco contém e o número de linhas que a cache contém. No caso do projeto RISC-V, esses parâmetros podem ser alterados no pacote de constante: N_BLK e BLK_SIZE.

- Taxa de Miss
  - Endereços requisitados da memória principal não encontradas no cache $(Taxa\_de\_Miss/Acessos) = 1 - Taxa\_de\_Hit$.
  - Números típicos são: 3-10% para L1 ou pode ser bem pequeno (por exemplo, <1%) para L2, dependendo do tamanho, etc. Pelo fato do projeto ter apenas a cache de primeiro nível (L1) e ser bem menor do que a das CPUs comerciais, essa taxa pode variar bastante.
- Tempo de Hit
  - Tempo para atender uma requisição de uma linha no cache para o processador, incluindo o tempo de comparação de tags e para determinar se a linha está valida no cache.
  - Números típicos: 4 ciclos de clock para L1, 10 ciclos de clock para L2. Para o este projeto, o tempo de Hit leva apenas 1 ciclo de clock pois, como a proporção é menor, os bits de memória são como registradores e não como a SRAM comercial.
- Penalidade de Miss
  - Tempo adicional necessário para buscar dados na memória principal devido a um Cache Miss.
  - Tipicamente 50-200 ciclos para memória principal. Dentro do projeto, para simular a penalidade de tempo na busca de dados da memória principal, foi introduzido um delay de 7 ciclos de clock típico de um barramento Avalon das FPGAs da Intel [17].

Multiplicação de matrizes

Como a multiplicação de matrizes é uma operação tão essencial para diversas aplicações e algoritmos numéricos, muitas pesquisas são feitas com o intuito de otimizar os algoritmos de multiplicação de matrizes. Aplicações de multiplicação de matrizes em problemas computacionais são vastamente encontrados, incluindo computação científica e reconhecimento de padrões que tem grande espaço na área. Além disso, é um importante algoritmo que se utiliza de muitos acessos a memória e, inevitavelmente, testará a performance de uma memória cache. Para estimar o desempenho da cache em um cenário genérico de multiplicação de matrizes foi usada como base a lógica do código 5.1 que, representa uma das possíveis formas de fazer esse cálculo.

Listing 5.1: Código C++ de referência para multiplicação de matrizes

```
// This function multiplies mat1[][] and mat2[][],
// and stores the result in res[][]

void multiply(int mat1[][N],
              int mat2[][N],
              int res[][N])
{
    int i, j, k;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            res[i][j] = 0;
            for (k = 0; k < N; k++)
                res[i][j] += mat1[i][k] *
                             mat2[k][j];
        }
    }
}
```

A estruturação desses dados, novamente, leva vantagem quando os conceitos de localidade são aplicados. O resultados do desempenho para esse algoritmo executando em cima de duas matrizes quadradas de tamanho $N = 3$, variando o tamanho tanto de linhas quanto do bloco, podem ser observados na Tabela 5.2. Estes foram medidos usando contadores que se comunicam com o processador e recebem os sinais do controlador (maquina de estados) da Cache e então, aplicando a equação apresentada anteriormente. É possível

Tabela 5.2: Desempenho da cache de acordo com tamanho para multiplicação de uma matriz 3x3

N_BLK	**BLK_SIZE**	**MISS RATE**	**HIT RATE**
1	1	100%	0%
2	2	53%	47%
4	2	47%	53%
4	4	45%	55%
8	4	30%	70%
8	8	15%	85%
16	*8*	*15%*	*85%*
16	16	9%	91%

ver que quanto maior o tamanho da cache, menor é a taxa de miss e, consequentemente, menor o tempo que o processador ficará ocioso aguardando os dados da memória principal.

5.3 Dificuldades e Desafios

Uma das dificuldades encontradas se deu no timing da memória, todas as operações estavam sendo feitas na borda de subida do relógio, no entanto, para que um dado escrito em um determinado ciclo conseguisse ser lido no ciclo seguinte foi necessário que as operações de escrita fossem realizadas na borda de descida da memória. Isso acontece pois existe um certo atraso entre escrever no banco de registradores e a saída refletir essa mudança, impedindo que um dado fosse escrito e logo em seguida lido consistentemente. Isso refletiu em vários momentos da implementação e cada vez precisou de ajustes para funcionar corretamente com outros componentes.

Um outro grande problema que consumiu bastante tempo no projeto foi o Stall do processador interagindo com o Stall do controlador da memória cache. Foi necessário criar um módulo separado que atua em sua maior parte, de forma assíncrona. A integração estre os módulos é de certa forma mais complicada que o desenvolvimento e validação individualizados. Grande parte do tempo utilizado foi gasto na depuração entre a CPU e a Cache.

Capítulo 6

Conclusão

Foi feita a implementação do processador RISC-V em pipeline de 5 estágios e memória cache com mapeamento direto. Apesar das dificuldades encontradas ao longo do caminho, o trabalho foi realizado e gerou grande aprendizado na área de formação. É encorajado o uso desse produto para utilização em ambientes educacionais com intuito de introduzir e ensinar como funciona a arquitetura RV32I do RISC-V. Através deste trabalho, foi possível entender mais de perto a aplicação prática dos vários conceitos aprendidos durante o curso na Universidade de Brasília.

A memória cache, como foi possível ver, gerou um grande custo de recursos lógicos mas que certamente, apresentaria grande melhora no desempenho do processador. Muitas vezes se esquece que todos os componentes ao redor de uma CPU também contam para sua performance, sendo, a memória, um dos principais e indispensáveis componentes passiveis de pesquisa e enriquecimento.

6.1 Trabalhos Futuros

Os resultados obtidos foram condizentes para com o objetivo do trabalho. Muito ainda se pode fazer para melhorar e tornar mais robusto o trabalho apresentado aqui, sempre há espaço para novas ideias e funcionalidades:

- RV64I: A implementação do conjunto de instruções em 64 bits.
- Implementação FPGA: Este era um escopo desejado dentro deste trabalho, no entanto, não foi realizado.
- Memória Cache Associativa: A ideia de implementar diferentes versões da memória cache foi chegou a ser considerada, com o intuito de comparar a performance entre as diferentes possibilidades de implementação.

- Implementar comunicação com módulos externos de armazenados, no caso da FPGA, sendo feito pela interface USB e comunicação através do barramento.

Referências

[1] Ferlin, Edson Pedro: *O avanço tecnológico dos processadores e sua utilização pelo software*. Revista da Vinci, Curitiba, páginas 43–60, 2004. https://www.up.edu.br/davinci/pdf03.pdf, acesso em 29/06/2019. ix, 3, 4

[2] John L. Hennessy, David A. Patterson: *Computer Architecture: A Quantitative Approach, 5th Edition*. Morgan Kaufmann; 5 edition; Elsevier, 2012, ISBN 012383872X 978-8178672663. ix, 11, 12, 13, 15

[3] Tudruj, Marek: *Fundamentals of computer architecture: Cache memory organization*. https://edux.pjwstk.edu.pl/mat/264/lec/main84.html, acesso em 14/03/2019. ix, 14, 16

[4] Coorporation, Intel: *Hexadecimal object file format specification*, Jan 1988. https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/ads264_mws228/Final%20Report/Final%20Report/Intel%20HEX%20Standard.pdfC, acesso em 02/07/2019. ix, 28, 29

[5] Association, Open Source Hardware: *Open source hardware definition*. https://www.oshwa.org/definition/, acesso em 17/06/2019. 1

[6] Dr. Paul Mullins, Slippery Rock University: *Introduction to computers: Hardware and software*. http://cs.sru.edu/~mullins/cpsc100book/module02_introduction/module02-03_introduction.html, acesso em 18/06/2019. 2

[7] Coorporation, Intel: *Mais de 50 anos da lei de moore*. https://www.intel.com.br/content/www/br/pt/silicon-innovations/moores-law-technology.html, acesso em 30/06/2019. 2

[8] Engineering e Technology History Wiki (ETHW): *Eniac (electronic numerical integrator and computer)*, 2017. https://ethw.org/ENIAC, acesso em 30/06/2019. 3

[9] John L. Hennessy, David A. Patterson: *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann; 5 edition, 2013, ISBN 9780124077263 978-0124077263. 3

[10] Woss, Anderson Carlos: *Vhdl é linguagem de programação?*, Jun 2017. https://pt.stackoverflow.com/questions/216032/vhdl-%C3%A9-linguagem-de-programa%C3%A7%C3%A3o. 5

[11] David Patterson, Andrew Waterman: *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon; 1 edition, 2017, ISBN 0999249118 978-0999249116. 5

[12] Andrew Waterman, Krste Asanovi: *The risc-v instruction set manual*, 2017. `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. 8, 9

[13] Wm Wulf, Sally A. McKee: *Hitting the memory wall: Implications of the obvious*. Dez 1994. `https://dl.acm.org/citation.cfm?id=216588`. 21

[14] Lam, William K.: *Hardware Design Verification: Simulation and Formal Method-Based Approaches 1st Edition*. Prentice Hall PTR, 2005, ISBN 0137010923 978-0137010929. 27

[15] TheThirdOne, Benjamin Landers: *Risc-v assembler, simulator, and runtime*, Mar 2019. `https://github.com/TheThirdOne/rars`, acesso em 01/04/2019. 28, 29

[16] Hörz, Maël: *Hxd - freeware hex editor and disk editor*, Fev 2019. `https://mh-nexus.de/en/hxd/`, acesso em 07/04/2019. 28

[17] Coorporation, Intel: *Avalon® interface specifications*, Set 2018. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf`, acesso em 02/04/2019. 33

Anexo I

RV32/64G Instruction Set Listings

Chapter 19

RV32/64G Instruction Set Listings

One goal of the RISC-V project is that it be used as a stable software development target. For this purpose, we define a combination of a base ISA (RV32I or RV64I) plus selected standard extensions (IMAFD) as a "general-purpose" ISA, and we use the abbreviation G for the IMAFD combination of instruction-set extensions. This chapter presents opcode maps and instruction-set listings for RV32G and RV64G.

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Table 19.1: RISC-V base opcode map, inst[1:0]=11

Table 19.1 shows a map of the major opcodes for RVG. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. Opcodes marked as *reserved* should be avoided for custom instruction set extensions as they might be used by future standard extensions. Major opcodes marked as *custom-0* and *custom-1* will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked *custom-2/rv128* and *custom-3/rv128* are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions in RV32 and RV64.

We believe RV32G and RV64G provide simple but complete instruction sets for a broad range of general-purpose computing. The optional compressed instruction set described in Chapter 12 can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

As we move beyond IMAFDC into further instruction set extensions, the added instructions tend to be more domain-specific and only provide benefits to a restricted class of applications, e.g., for multimedia or security. Unlike most commercial ISAs, the RISC-V ISA design clearly separates the base ISA and broadly applicable standard extensions from these more specialized additions. Chapter 21 has a more extensive discussion of ways to add extensions to the RISC-V ISA.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

RV32A Standard Extension

00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64A Standard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	LR.D
00011	aq	rl	rs2	rs1	011	rd	0101111	SC.D
00001	aq	rl	rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	aq	rl	rs2	rs1	011	rd	0101111	AMOADD.D
00100	aq	rl	rs2	rs1	011	rd	0101111	AMOXOR.D
01100	aq	rl	rs2	rs1	011	rd	0101111	AMOAND.D
01000	aq	rl	rs2	rs1	011	rd	0101111	AMOOR.D
10000	aq	rl	rs2	rs1	011	rd	0101111	AMOMIN.D
10100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAX.D
11000	aq	rl	rs2	rs1	011	rd	0101111	AMOMINU.D
11100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAXU.D

RV32F Standard Extension

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.W
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.W.X

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64F Standard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	FCVT.L.S
1100000	00011	rs1	rm	rd	1010011	FCVT.LU.S
1101000	00010	rs1	rm	rd	1010011	FCVT.S.L
1101000	00011	rs1	rm	rd	1010011	FCVT.S.LU

RV32D Standard Extension

imm[11:0]		rs1	011	rd	0000111	FLD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01 rs2	rs1	rm	rd	1000011	FMADD.D
rs3	01 rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	01 rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	01 rs2	rs1	rm	rd	1001111	FNMADD.D
0000001	rs2	rs1	rm	rd	1010011	FADD.D
0000101	rs2	rs1	rm	rd	1010011	FSUB.D
0001001	rs2	rs1	rm	rd	1010011	FMUL.D
0001101	rs2	rs1	rm	rd	1010011	FDIV.D
0101101	00000	rs1	rm	rd	1010011	FSQRT.D
0010001	rs2	rs1	000	rd	1010011	FSGNJ.D
0010001	rs2	rs1	001	rd	1010011	FSGNJN.D
0010001	rs2	rs1	010	rd	1010011	FSGNJX.D
0010101	rs2	rs1	000	rd	1010011	FMIN.D
0010101	rs2	rs1	001	rd	1010011	FMAX.D
0100000	00001	rs1	rm	rd	1010011	FCVT.S.D
0100001	00000	rs1	rm	rd	1010011	FCVT.D.S
1010001	rs2	rs1	010	rd	1010011	FEQ.D
1010001	rs2	rs1	001	rd	1010011	FLT.D
1010001	rs2	rs1	000	rd	1010011	FLE.D
1110001	00000	rs1	001	rd	1010011	FCLASS.D
1100001	00000	rs1	rm	rd	1010011	FCVT.W.D
1100001	00001	rs1	rm	rd	1010011	FCVT.WU.D
1101001	00000	rs1	rm	rd	1010011	FCVT.D.W
1101001	00001	rs1	rm	rd	1010011	FCVT.D.WU

RV64D Standard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	FCVT.L.D
1100001	00011	rs1	rm	rd	1010011	FCVT.LU.D
1110001	00000	rs1	000	rd	1010011	FMV.X.D
1101001	00010	rs1	rm	rd	1010011	FCVT.D.L
1101001	00011	rs1	rm	rd	1010011	FCVT.D.LU
1111001	00000	rs1	000	rd	1010011	FMV.D.X

Table 19.2: Instruction listing for RISC-V