Universidade Federal de Pernambuco
Centro de Informática
Departamento de Engenharia da Computação

Bacharelado em Engenharia da Computação

Um modelo RTL configurável de memória cache sintetizável em FPGA

Ary Guedes Lins

Trabalho de Graduação

Recife
14 de julho de 2017

Universidade Federal de Pernambuco
Centro de Informática
Departamento de Engenharia da Computação

Ary Guedes Lins

Um modelo RTL configurável de memória cache sintetizável em FPGA

Trabalho apresentado ao Programa de Bacharelado em Engenharia da Computação do Departamento de Engenharia da Computação da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Edna Natividade da Silva Barros*

Recife
14 de julho de 2017

Agradecimentos

Agradeço ao meu bom Deus que por sua poderosa mão me sustentou até aqui. Aos meus pais pelos valores e virtudes que aprendi com seu exemplo. Ao meu pai, Aryed Lins, com quem aprendi a honestidade e o caráter. À minha mãe, Rosemery Guedes, que me ensinou humildade e a bondade. À minha melhor amiga e companheira Giovanna Cavalcanti que em todo tempo me incentivou e animou, não me deixando nunca desistir. À minha orientadora Edna Barros por seu apoio e suporte, um exemplo de dedicação. À todos amigos e colegas do CIn, em especial a Rafael Nunes, Stefano Laporte, Marcus Felipe, Carolina Maria, Luiz Lima, Lucas Félix, Felipe Nunes, Gabriel Lima e Gabriel Medeiros, companheiros em diversos momentos durante graduação: projetos, almoços, viagens, e tempos de estudo e descontração. À todos os amigos e familiares que trazem-me alegria todos os dias. Agradeço também a Vanessa Ogg, que cedeu não só o seu trabalho de graduação, mas também seu tempo e paciência para me dar suporte nesse projeto.

O sucesso é ir de fracasso em fracasso sem perder entusiasmo.
—WINSTON CHURCHILL

Resumo

O uso de *hardware* personalizado para aplicações específicas tem se tornado uma prática comum, tanto para otimizar o desempenho como para reduzir o consumo de energia de um sistema embarcado. O conjunto de instruções (ISA) de um processador é a parte crítica de um sistema, definindo a interface entre o *hardware* e o *software*, sendo sua especificação um processo custoso em termos de tempo e passível de erros. Neste contexto, surgiu o projeto RISC-V, uma especificação de ISA aberto, livre e flexível para reuso e modificações; deixando a arquitetura e a implementação do *hardware* a cargo do projetista. Até o RISC-V os ISAs existentes eram proprietários, complexos e fechados para reuso e modificação, por isso o RISC-V deu uma alternativa mais simples e ágil ao desenvolvimento de *Systems on a Chip* (SoCs). Outro ponto crítico a se considerar na implementação SoCs além do ISA é a sua hierarquia de memória, onde as memórias caches são dispositivos críticos da hierarquia e que afetam o desempenho, custo e consumo de energia de um SoC. Com o mesmo objetivo do projeto RISC-V – otimizar o tempo de projeto de um SoC – propomos neste trabalho um modelo RTL de memória cache configurável para dar agilidade ao processo de desenvolvimento e flexibilidade para testar várias configurações diferentes de maneira simples e ágil. Este modelo foi validado funcionalmente e em *timing* tanto isoladamente como também num modelo RTL de uma arquitetura de um sistema contendo uma *Celtral Processor Unit* (CPU) RISC-V, com auxílio das ferramenta de EDA QuestaSim e Quartus II. Este modelo de sistema passou por síntese e prototipação para a FPGA Cyclone IV GX da Altera com uma frequência de *clock* de 50MHz, ocupando 11% dos elementos lógicos e 2% do total da memória interna da FPGA.

Palavras-chave: memória configurável ISA SoC cache RTL FPGA

Sumário

1 Introdução **1**

2 Conceitos Básicos **4**

2.1 Memórias *Cache* 4

2.1.1 Capacidade da Cache 6

2.1.2 Tamanho do bloco 6

2.1.3 Associatividade 7

2.1.4 Politicas de Substituição 8

2.2 RISCV 8

2.2.1 ISA RISC-V 8

3 Trabalhos Relacionados **9**

4 Metodologia **11**

4.1 Especificação de requisitos do projeto 11

4.2 Validação funcional do modelo de cache 13

4.3 Validação em *timing* do modelo de cache 14

4.4 Validação funcional e em *timing* do modelo de sistema 14

4.5 Síntese e prototipação em FPGA 15

5 Implementação **16**

5.1 Modelo de Referência 16

5.1.1 *Instruction Gen* 17

5.1.2 *Mem Gen* 17

5.1.3 Módulo Principal 17

5.2 Modelo RTL de Cache 17
5.2.1 Parametrização 17
5.2.2 Interface 19
5.2.3 Unidade de Controle 19
5.3 Modelo RTL de sistema 20

6 Resultados e Discussão 25
6.1 Validação do modelo da Cache 25
6.2 Validação do modelo do Sistema 25
6.3 *Benchmarking* 27
6.4 Síntese e Prototipação em FPGA 30

7 Conclusão e Trabalhos Futuros 31

Lista de Figuras

1.1 Pirâmide representando o conceito de hierarquia de memória em sistemas computacionais. 2

2.1 Organização básica de memória cache com associatividade 2. 5

3.1 Plataforma de validação utilizada em [SB+12] 9

3.2 Estágios de pipeline da implementação do RISC-V desenvolvida em [Ogg16]. 10

4.1 Fluxo de desenvolvimento. 12

4.2 *Setup* do *testbench* para validação do modelo RTL de cache isoladamente. 14

4.3 *Setup* do *testbench* para validação do modelo RTL da arquitetura proposta integrada com a cache. 15

5.1 Modelo de referência. 16

5.2 Trecho do arquivo contendo sequência aleatória de *loads & stores*. 17

5.3 Máquina de estados de controle da memória cache. 20

5.4 Arquitetura de sistema proposta. 21

5.5 Endereçamento do sistema. 21

5.6 Barramento wishbone. 22

5.7 Leitura simples no barramento WISHBONE. 23

5.8 Escrita simples no barramento WISHBONE. 24

6.1 Comparação de arquivos contendo o conteúdo da memória ao fim da execução dos modelos de referência e RTL. O texto em verde são conteúdo do arquivo de saída do modelo RTL, em vermelho são do arquivo de saída do modelo de referência e o texto em branco significa conteúdo idêntico de ambos arquivos. Este é um exemplo de falha na validação do modelo RTL. 26

6.2 Comparação de arquivos contendo o conteúdo da memória ao fim da execução dos modelos de referência e RTL. O texto em verde são conteúdo do arquivo de saída do modelo RTL, em vermelho são do arquivo de saída do modelo de referência e o texto em branco significa conteúdo idêntico de ambos arquivos. Este é um exemplo de sucesso na validação do modelo RTL. 26

6.3 Trecho de código na memória ROM do teste da instrução *sb*. 28

6.4 Trecho de código na memória ROM do teste da instrução *sb* onde estão as instruções finais de falha ou sucesso. 28

6.5 Exemplo da simulação da instrução *sb* falhando. Podemos verificar marcado em vermelho que o PC parou sua execução na instrução que indica falha de execução do teste. Esta instrução está marcada em vermelho na fig. 6.4. 29

6.6 Dados de ocupação da FPGA para diversas configurações de cache. 30

Lista de Tabelas

5.1 Variáveis *parameters* da implementação do modelo RTL da cache configurável. 18

5.2 *Structs* da interface da cache e descrições de seus componentes. 19

6.1 Parâmetros configuráveis para os quais o modelo de cache foi validado. 25

6.2 Teste do ISA RV32MI. 27

6.3 Teste *matrix multiplication* (total de 840 instruções) do *benchmarking* TI embedded Benchmark Suite em caches com diversas configurações. *bytes*. 29

6.4 Dados comuns as configurações da fig. 6.6 da ocupação da FPGA para a síntese do sistema. 30

CAPÍTULO 1

Introdução

O uso de *hardware* personalizado para aplicações embarcadas específicas tem se tornado uma prática comum, tanto para otimizar o desempenho como para reduzir o consumo de potência. Sistemas eletrônicos de aplicação específica incluem, em geral, processadores de propósito geral e memória e estão sendo prototipados como SoCs (System on a Chip). Para produção em larga escala SoCs são prototipados como *Application Specific Integrated Circuits* (ASIC), caso contrário tem se tornado cada vez mais comum o uso de lógica reconfigurável, por exemplo, FPGAs (*Field-programmable gate array*) para produtos finais, e não só como meio para prototipação e teste dos chamados *Systems on a Chip* (SoCs).

Uma parte crítica no projeto de SoCs é a escolha dos processadores em função do conjunto de instruções dos processadores ou *Central Processor Unit* (CPU). O conjunto de instruções (ISA) define a interface entre o *hardware* e o *software*, a linguagem binária entendida pela CPU. Sua especificação é um processo custoso em termos de tempo e passível de uma grande quantidade de erros. Até então, o reuso de outros ISAs já estabelecidos não era um processo simples pois estes são, em geral, proprietários, complexos e não abertos a modificações.

Nesse contexto, com objetivo de reduzir o tempo de projeto de *hardware* personalizado baseados em processores como SoCs surgiu o projeto RISC-V [WL$^+$13], um processador com conjunto de instrução *Instruction Set Architecture* (ISA) aberto e livre desenvolvido por um grupo de pesquisa da Universidade da California – Berkeley. O ISA do processador RISC-V está disponível em versões de 32, 64 e 128 *bits*. O projeto não se limita apenas a especificação do conjunto de instruções, mas também fornece conjunto de ferramentas ou *riscv-tools*, uma *toolchain* completa para a arquitetura RISC-V que inclui *GNU tools*, testes do ISA, binários de sistemas operacionais (SOs), *benchmarks* e outras ferramentas úteis para o projeto de um sistema baseado em processadores. O ISA do RISC-V possui um conjunto base de instruções de operações com inteiros que deve estar obrigatoriamente presente em qualquer implementação, e possui também extensões opcionais para o ISA (e.g. extensão de ponto flutuante). Também é possível adicionar extensões personalizadas para processamento focado em aplicações específicas, tornando o RISC-V uma alternativa menos custosa para um projeto de *hardware* personalizado.

Além do ISA, outro ponto crítico a se considerar no projeto de um sistema é sua hierarquia de memória. Num projeto de um dispositivo de memória existem três características essenciais para se considerar: capacidade, custo e tempo de acesso (ou velocidade). Memórias com maior capacidade tendem a ser mais lentas, porém com menor custo por unidade de armazenamento (*bit*) (e.g. discos rígido e memórias RAM); enquanto que memórias com menor capacidade

tendem a ser mais rápidas e com maior custo por *bit* (e.g. registradores e caches). Dado essas limitações, a hierarquia de memória é utilizada para diminuir o tempo de acesso da *Central Processor Unit* (CPU) às informações necessárias para execução da aplicação (i.e. instruções e dados). Ela pode ser entendida como uma pirâmide onde as memórias com menor capacidade e mais velozes estão na ponta de cima e mais próximas do processador e as memórias maiores e lentas estão nos níveis mais baixos da hierarquia (Fig. 1.1).

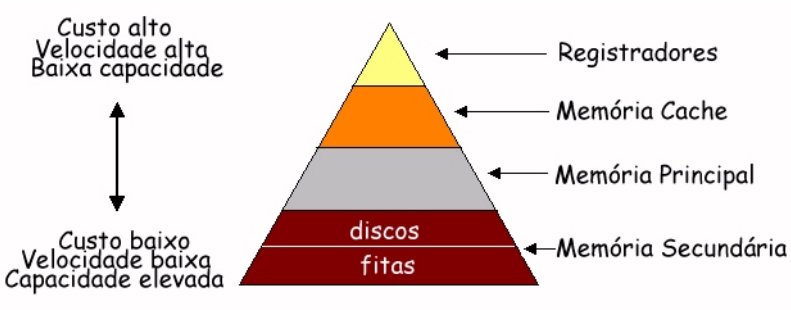

Figura 1.1 Pirâmide representando o conceito de hierarquia de memória em sistemas computacionais.

.

Dentro da hierarquia de memória, as memórias caches são dispositivos críticos pois podem afetar o sistema em questões de desempenho, pois são o nível de memória mais próximo da CPU sendo necessário que trabalhem na mesma frequência de *clock*; em questões de custos, pois são circuitos digitais complexos que diferentemente dos outros dispositivos da hierarquia tem seu gerenciamento realizado por *hardware* e não pelo sistema operacional (SO); e em questões de consumo de energia, caches com mais capacidade e mais complexas, embora em alguns casos possam melhorar o desempenho, tendem a consumir mais energia.

Levando isso em consideração, o projeto de memórias cache é muito peculiar de cada projeto, cada aplicação diferente pode demandar um processador que possua uma cache com características específicas, considerando um *tradeoff* entre desempenho, custo, consumo de energia. Observando esta demanda e com objetivo de otimizar o tempo de projeto de um *hardware* personalizado assim como no projeto RISC-V, este trabalho propõe um modelo RTL configurável de memória cache nível 1, ou seja a mais próxima da CPU, que seja sintetizável em FPGA, e com uma interface simples e genérica, além de ser facilmente configurável através de parâmetros fornecidos antes da síntese e de prototipação do modelo.

Este documento está organizado de acordo com a estrutura a seguir. No próximo capítulo (Cap. 2) discutiremos conceitos básicos para o entendimento da implementação como estrutura e funcionamento das memórias caches, o conjunto de instruções de um processador e sobre o projeto RISC-V. No capítulo 3 iremos apresentar brevemente alguns trabalhos relacionados a este envolvendo tanto o ISA do processador RISC-V, como modelos RTL de memórias cache configuráveis. A metodologia utilizada no projeto será detalhada no capitulo 4. E finalmente, no capítulo 5 apresentaremos a implementação proposta por este trabalho. Então mostraremos e discutiremos os resultados obtidos no capítulo 6. O capítulo 7 sumariza as contribuições deste

trabalho, e apresenta perspectivas para o futuro.

CAPÍTULO 2

Conceitos Básicos

Neste capítulo apresentaremos conceitos básicos importantes para o entendimento deste trabalho, apresentando os conceitos básicos de memória cache e sobre a implementação do processador RISC-V utilizado neste trabalho.

2.1 Memórias *Cache*

As memórias cache surgiram com objetivo de diminuir o gargalo entre a velocidade da *CPU* e a latência da memória principal de um computador. Essas memórias se baseiam em 2 princípios encontrados nas aplicações atuais: princípio da localidade temporal – se determinado endereço foi referenciado num determinado momento, então tem grande probabilidade de ser referenciado novamente num futuro próximo – e localidade espacial – se um determinado endereço foi referenciado num determinado momento, então há grande probabilidade que endereços próximos sejam referenciados também num futuro próximo. É comprovado experimentalmente que esses princípios ocorrem na prática na maioria dos programas [PH12], este fato pode ser explicado por conta de como em geral são estruturados os programas e os tipos de dados (e.g. execução de *loops*, processamento de *arrays* e etc.).

Um exemplo de organização básica de uma memória cache pode ser visualizada na figura 2.1. Quando um dado (instrução ou variável) é armazenado na cache seu endereço é dividido em três partes: a *tag*, o *index* (ou linha) e o *offset*. A *tag* é armazenada numa estrutura de memória auxiliar, e serve para identificar o bloco de dados. O *index* indica a posição na memória cache em que o dado será armazenado. O *offset* serve para selecionar a palavra (32 *bits*) dentro de um bloco de dados (quando for maior ou igual a 1 palavra).

O funcionamento da cache ocorre da seguinte maneira: sempre que o processador tenta acessar dados em um determinado endereço de memória, é verificado se os dados daquele determinado endereço estão presentes na cache através da *tag*. Se o dado estiver presente, então a cache fornece o dado para a *CPU* que continua a execução, se não a cache envia um sinal para bloquear a execução *CPU* e realiza a busca do bloco de dados contendo aquele endereço na memória principal. No caso em que o dado está presente na cache dizemos que ocorreu um cache *hit*, caso contrário dizemos que ocorreu um cache *miss*. O desempenho de uma memória cache pode ser medido através de sua taxa de *hits* – quanto maior melhor – ou através da taxa de *misses* – quanto menor melhor. Essas taxas podem ser definidas pelas seguintes equações:

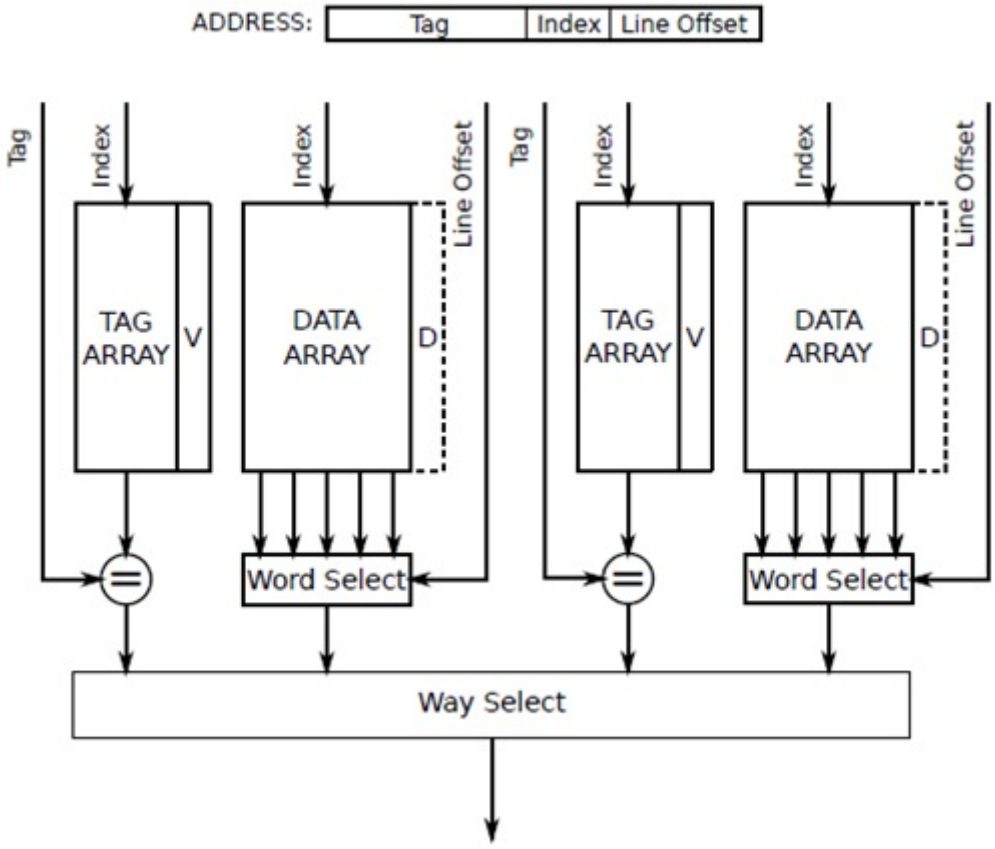

Figura 2.1 Organização básica de memória cache com associatividade 2.

$$hit_rate = \frac{\#hits}{\#access} \tag{2.1}$$

$$miss_rate = \frac{\#miss}{\#access} \tag{2.2}$$

onde *#hits* refere-se a quantidade total de cache *hits*, *#miss* a quantidade total de cache *miss* e *#access* a quantidade de acessos a cache.

Algumas características da memória cache irão definir seu comportamento durante a execução de um programa. Essas características irão influenciar o seu desempenho, custo, consumo de energia e etc. São elas: a capacidade da cache, tamanho do bloco, o grau de associatividade, política de escrita e política de substituição. Discutiremos, a seguir, cada uma dessas características e sua influência sobre o desempenho da memória cache.

2.1.1 Capacidade da Cache

A capacidade da cache refere-se a quantidade de dados total que podem ser armazenados. Pode ser definido em termos de *bytes* (8 bits) ou em termos palavras (32 bits). Caches maiores implicam em menor probabilidade de ocorrer um cache *miss*, porém também um tempo de acesso é maior [PH12], o que pode ser crítico pois as caches precisam trabalhar na mesma frequência de *clock* da CPU. O custo também é um fator que aumenta em caches maiores e deve ser considerado.

2.1.2 Tamanho do bloco

O bloco é um grupo de palavras que serão sempre carregados na cache ou escritos de volta para a memória conjuntamente (unidade de transferência). O seu tamanho é definido em termos de palavras ou *bytes*.

Para exemplificar, dado o seguinte cenário: uma memória principal com 32 palavras endereçadas de 0 a 31, uma cache de tamanho de 16 palavras com mapeamento direto e tamanho do bloco igual a 4 palavras. Utilizamos a seguinte equação para calcular a quantidade de linhas (quantidade de blocos) dessa cache:

$$\#linhas = \frac{tam_cache}{tam_bloco} \tag{2.3}$$

onde *tam_bloco* é o tamanho do bloco, *tam_cache* a capacidade total da cache e *#linhas* a quantidade de linhas ou blocos da cache.

Usando essa equação (2.3) concluímos que esta cache possui 4 linhas. No mapeamento direto, cada uma das linhas é mapeada da memória principal para a cache da seguinte forma:

$$end_linha = end_bloco \bmod \#linhas \tag{2.4}$$

onde *end_bloco* é o endereço do bloco na memória principal e *end_linha* é o *índice* para o qual será mapeado o bloco na cache.

Ou seja, para a cache mencionada, a linha 0 pode armazenar os dados do bloco 0 (palavras 0–3) e do bloco 5 (palavras 16–19), a linha 1 os dados dos blocos 1 (palavras 4–7) e 6 (palavras 20–23), e assim sucessivamente. Então, sempre que a palavra do endereço 2 for solicitada pelo processador, a memória cache irá transferir da memória principal o bloco contendo as palavras de 0-3. Quando aumentamos tamanho do bloco, estamos utilizando o princípio da localidade espacial para tentar reduzir a taxa de cache *miss*, diminuímos então o número de vezes que a cache bloqueia a *CPU* para transferir blocos da memória principal, uma vez que a cache transferirá grandes blocos de uma vez. Porém, quando ocorre um cache *miss* aumentamos o tempo necessário para carregar um bloco da memória, já que o barramento de dados possui tamanho limitado, essa penalidade é chamada de *miss penalty*.

2.1.3 Associatividade

Numa memória cache é necessário mapear os blocos de dados que chegam da memória principal para uma posição. No exemplo anterior consideramos uma cache com mapeamento direto, i.e. cada bloco é mapeado para apenas uma linha da cache. Se aumentarmos o grau de associatividade da memória cache, é possível mapear um endereço de memória para mais de uma posição na cache. Exemplificando: consideremos agora que a cache do exemplo anterior seja uma cache associativa de grau 2. Então, significa que teremos duas estruturas denominadas de *sets*, e em cada set podem ser armazenados dois blocos que mapeiam para mesmo set (possuem o mesmo índice no endereço). Cada um desses *sets* terão o número de linhas calculados pela fórmula a seguir:

$$\#linhas = \frac{tam_cache}{tam_bloco \cdot assoc} \tag{2.5}$$

onde *assoc* significa o grau de associatividade da cache.

Ou seja, teremos 2 *sets* cada um com 2 linhas endereçadas de 0 a 1. O mapeamento dos blocos ainda é feito com a equação 2.4, porém agora a cache irá escolher um dos 2 *sets* para carregar o bloco segundo sua **política de substituição**. A medida em que o grau de associatividade de uma cache aumenta, diminui a taxa de faltas otimizando o uso de seu espaço diminuindo substituições desnecessárias quando ainda há espaço livre na cache. Porém o aumento do grau de associatividade também aumenta a complexidade do *controlador* e quantidade de hardware para decodificação do endereço, aumentando a quantidade de portas lógicas e consequentemente aumentando seu custo.

2.1.4 Politicas de Substituição

Políticas de substituição são técnicas utilizadas pelas memórias cache associativas para escolher que blocos de dados serão substituídos quando não há mais espaço para novos blocos em um conjunto. Alguns exemplos são:

- Substituição Aleatória - escolhe aleatóriamente um bloco para ser substituído.
- *First in First Out* (*FIFO*) - o primeiro bloco que foi alocado é o primeiro que sai, uma fila.
- *Least Recently Used* (*LRU)* - o bloco com referência mais antiga do processador é escolhido para ser substituído.

Dentre estes geralmente o LRU apresenta maior taxa de cache *hit* [KS16], porém o seu custo em termos *hardware* é maior que os outros dois. O *FIFO* é uma boa aproximação do *LRU* e bem menos custoso. Já o de substituição aleatória é o menos custoso, porém em geral apresenta pior taxa de cache *hit*.

2.2 RISCV

O RISC-V é uma arquitetura de conjunto de instruções (ISA) aberto e livre de um processador de propósito geral. Foi desenvolvido por um grupo de pesquisa da Universidade da Califórnia – Berkeley, e está disponível nas versões de 32, 64 e 128 bits. Neste projeto além da especificação do conjunto de instruções é fornecido uma *toolchain* com compiladores e binários de sistemas operacionais para a construção de um sistema completo, integrando *hardware* e *software*.

2.2.1 ISA RISC-V

O ISA RISC-V é definido por um conjunto base de instruções de operações com inteiros que precisam estar presentes em qualquer implementação, e extensões opcionais a este conjunto base [WA17]. Existem dois conjuntos bases principais que são o RV32I e o RV64I, que referem-se ao conjunto base de intruções com inteiros para espaços de endereçamento de 32 e 64 bits respectivamente. A extensão M refere-se às instruções de multiplicação e divisão de inteiros, ou seja RV32MI é o ISA RISC-V com as instruções bases de inteiro, mais as instruções de multiplicação e divisão de inteiros. Este foi o ISA utilizado na implementação desenvolvida em [Ogg16] e que foi utilizada neste trabalho e será descrita com mais detalhes no capítulo 3.

CAPÍTULO 3

Trabalhos Relacionados

Nessa seção apresentaremos um modelo RTL configurável de cache desenvolvido em [SB+12], e também a implementação RTL do processador RISC-V desenvolvida em [Ogg16] utilizada na plataforma de validação.

Em [SB+12] são propostos dois modelos RTLs de caches genéricas em VHDL, sendo uma de dados e outra de instrução. Nestas implementações os seguintes parâmetros são configuráveis: *tamanho do bloco, quantidade de linhas da cache, associatividade, politica de substituição e política de escrita*. As políticas de substituição implementadas são substituição pseudo-aleatória e LRU. Já as de escrita são *write back* e *write through* (podendo ser *write allocate* ou *no write allocate*). Na associatividade existe um limite em que o grau máximo configurável é 4, sendo necessário alterações na implementação das políticas de substituição para comportar associatividade maiores.

Para a validação do modelo RTL de cache desenvolvido no trabalho, foi utilizado um sistema com um *processador* MIPs com *pipeline* de 5 estágios; um controlador DDR de memória RAM; um barramento *AMBA*^{TM} *High-performance Bus* (AHB) que faz a interface entre o a memória o controlador da memória RAM e as caches; um árbitro AHB para sincronizar o acesso das caches a memória principal; e ainda porta UART com interface *Advanced Peripheral Bus* (APB) que se conecta ao barramento AHB através de uma *AHB-to-APB bridge*. Essa configuração pode ser visualizada na Fig. 3.1.

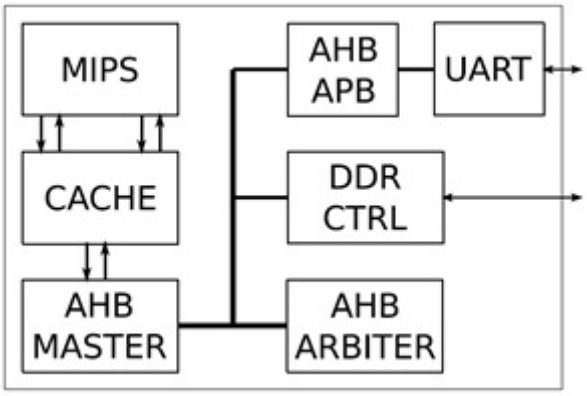

Figura 3.1 Plataforma de validação utilizada em [SB+12]

.

Uma implementação RTL genérica do processador RISC-V desenvolvida em SystemVerilog [Ogg16] foi utilizada no modelo RTL do sistema, no qual as caches desenvolvidas neste trabalho foram integradas. Esta implementação possui um conjunto base de instruções aritméticas com inteiros, endereçamento de 32 bits, pipeline de 3 estágios – *fetch, decode/execute e write back* –, previsão de desvio estática com penalidade de 1 ciclo de *clock*, e controle de conflitos utilizando técnicas de *fowarding*. Este modelo RTL de CPU utiliza as 47 instruções do subconjunto RV32MI do ISA RISC-V que implementa instruções aritméticas de inteiros e a extensão de multiplicação e divisão também de inteiros. Mais detalhes deste trabalho serão apresentados posteriormente.

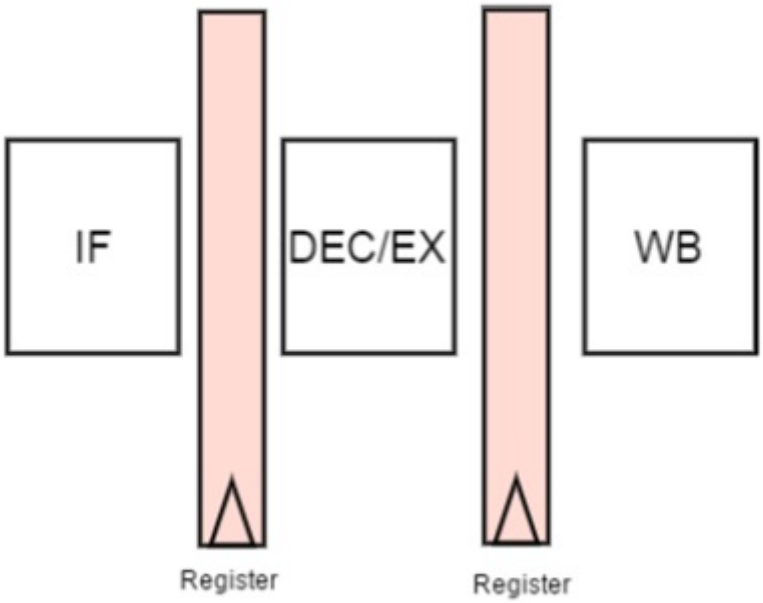

Figura 3.2 Estágios de pipeline da implementação do RISC-V desenvolvida em [Ogg16].

CAPÍTULO 4

Metodologia

Conforme o fluxo da fig. 4.1, definimos primeiramente os requisitos e restrições do projeto, então realizamos a implementação do modelo RTL na HDL SystemVerilog. Após isso foi necessário realizar uma validação funcional RTL de cache para verificar se o modelo se comporta como esperado. Então realizamos uma validação em *timing* para verificar se o modelo de cache funcionava com as restrições impostas pela plataforma – no caso desse projeto a FPGA Cyclone IV GX. Passando por essas etapas realizamos a integração numa arquitetura de sistema proposta, que passou por nova validação funcional e em *timing*. Atingindo sucesso nessas etapas realizamos a síntese e prototipação desse sistema para a plataforma. Neste capítulo iremos dar mais detalhes de cada passo seguido no fluxo de desenvolvimento desse projeto.

4.1 Especificação de requisitos do projeto

Definimos primeiramente os requisitos funcionais do projeto para o modelo de cache proposto, os quais estão descritos a seguir.

- Fornecer ao projetista de sistema uma interface simples e intuitiva para integração com outros módulos.
- Ser configurável através de parâmetros fornecidos antes da síntese e prototipação do sistema. Estes parâmetros são: capacidade da cache, tamanho do bloco e associatividade.
- Ser genérico de forma que seja possível a integração em diversos sistemas.
- Ser sintetizável em FPGA.
- Possuir um modelo de referência para que possa ser validada funcionalmente para quaisquer configurações.
- Funcionar com política de escrita *write back*.
- Funcionar com política de substituição FIFO quando for associativa.

Os requisitos não funcionais para o modelo RTL de cache proposto sào descritos a seguir.

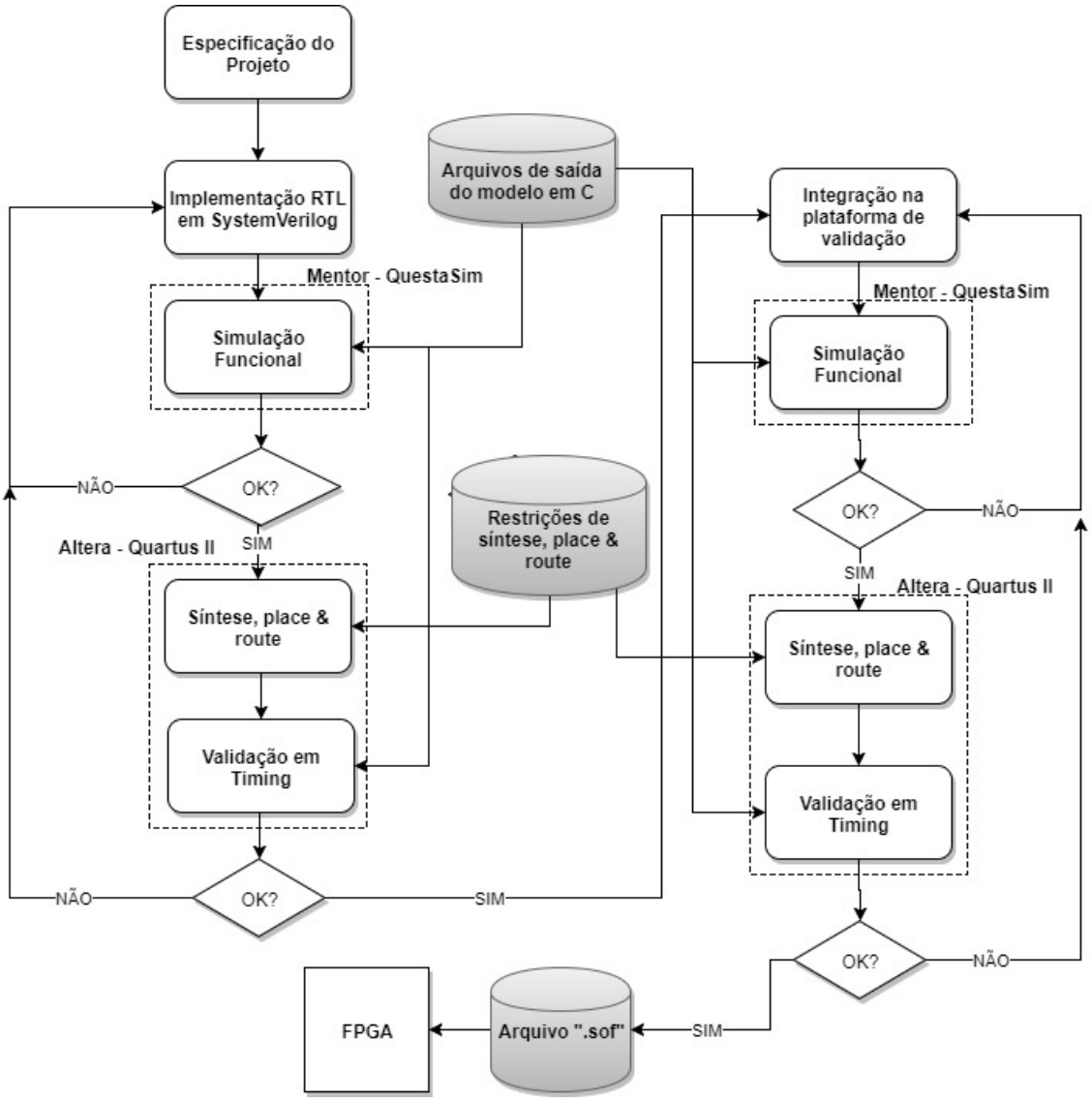

Figura 4.1 Fluxo de desenvolvimento.

- Ser integrado em um sistema contendo uma CPU, memória RAM e barramento para ser validado em um cenário real.
- Ser mapeado para a FPGA Cyclone IV GX da Altera dentro dessa arquitetura de sistema definida.

Para alcançar estes objetivos escolhemos a HDL SystemVerilog, tirando vantagem da sua simplicidade, e de seus recursos de alto nível como *structs*, *parameters* e *generate blocks* para implementar a parametrização do modelo RTL. Para suporte a implementação foram utilizados os *softwares* QuestaSim da Mentor Graphics para simulação, e o software Quartus II da Altera para síntese e prototipação em FPGA. Para validação do modelo, criamos também um modelo de referência na linguagem de programação C, que simula o comportamento funcional à nível de algoritmo. Como definido em [GAG+09] partimos de um modelo com nível de abstração maior para chegar a um modelo em nível de portas lógicas (RTL). Este modelo de referência é de extrema importância, visto que neste trabalho seria impossível testar todas as configurações possíveis de cache, então o projetista pode utilizá-lo como apoio para validar quaisquer configurações que desejar.

4.2 Validação funcional do modelo de cache

Para esta validação foi utilizado o *software* QuestaSim 10.2c. Aqui é considerado apenas o aspecto funcional do modelo RTL de cache desenvolvido, ou seja não são considerados detalhes específicos da plataforma, no caso deste projeto a FPGA Cyclone IV GX. Este tipo de validação foi realizada em duas etapas do projeto como podemos visualizar no fluxo da figura 4.1. Nesta seção descreveremos como a primeira validação foi realizada a qual considerou apenas o modelo RTL de cache isoladamente. O esquema montado para esta validação pode ser visualizado na (Fig. 4.2). O primeiro passo foi gerar uma sequência aleatória de instruções de *load* e *store* e uma inicialização aleatória das posições de memórias. Estes dados foram salvos em dois arquivos separados e fornecidos como entrada para o modelo de referência para o qual também foi fornecida a configuração de cache desejada. O modelo forneceu como saída um arquivo contendo o estado final da memória para esta execução, chamamos este arquivo da memória de referência.

O próximo passo foi fornecer os parâmetros desejados ao modelo RTL de cache e executar o *testbench* que leu a sequência de instruções aleatórias e gerou estímulos para que a cache executasse as operações de leitura e escrita na memória. Após o final da simulação, extraímos um arquivo contendo o conteúdo final da memória de simulação e comparamos com o arquivo da memória de referência. Como houve inconsistências (Fig. 6.1) entre os dois arquivos, verificamos o *waveform* da simulação para detectar possíveis falhas e voltamos a etapa de implementação para realizar ajustes. Quando não haviam mais inconsistências, consideramos que o modelo RTL de cache estava corretamente implementado.

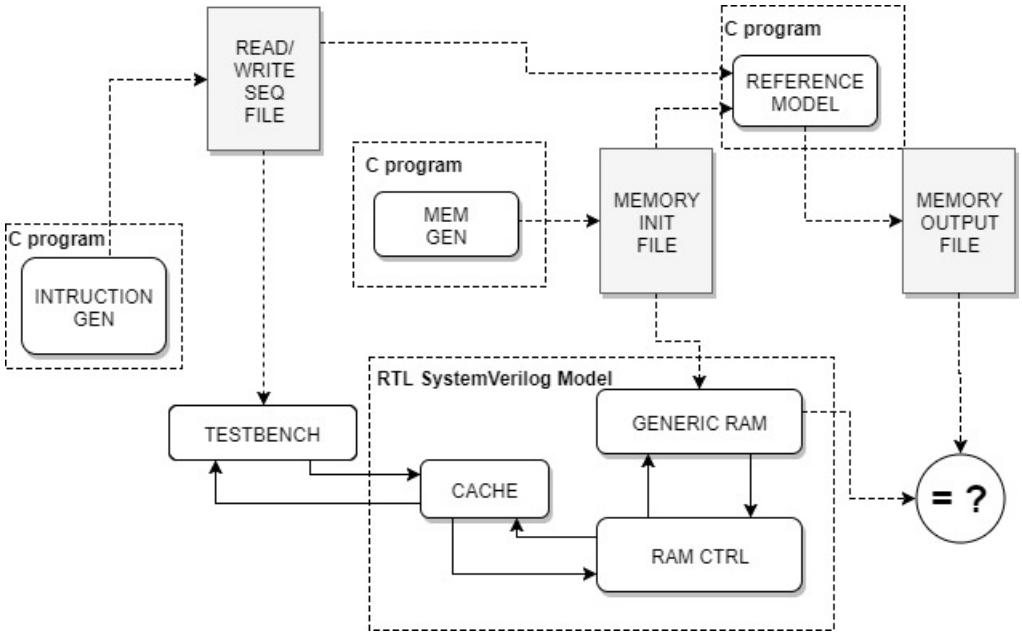

Figura 4.2 *Setup* do *testbench* para validação do modelo RTL de cache isoladamente.

4.3 Validação em *timing* do modelo de cache

Para a simulação com temporização, foi necessário previamente realizar o mapeamento – síntese; e place & route – do modelo RTL para a plataforma destino: a FPGA Cyclone IV GX da Altera. Para esta tarefa utilizamos a ferramenta de EDA Quartus II 13.1 da Altera. Após o mapeamento para plataforma específica, inserimos as restrições temporais, ou seja o atraso real da portas lógicas junto com a frequência de *clock* desejada. A ferramenta Quartus então gerou um arquivo em linguagem Verilog do modelo contendo as restrições temporais da plataforma, que pode ser utilizado no *software QuestaSim* para validação. Empregamos então os mesmos testes e critérios de aceitação utilizados na validação funcional.

4.4 Validação funcional e em *timing* do modelo de sistema

Após a integração na arquitetura destino realizamos uma nova validação funcional, considerando agora os teste do ISA da *toolchain* do RISC-V. Considerando que o RISC-V é uma arquitetura *load & store* e a nossa implementação do RISC-V utilizada [Ogg16] já foi testada para todas as 47 instruções do subconjunto RV32IM do ISA RISC-V, então selecionamos apenas as instruções de *load* e *store* para realizar a validação da arquitetura, são elas: *lw, lh, lb, sw, sh* e *sb*. O esquema desse teste pode ser visualizado na Fig. 4.3.

A simulação em *timing* seguiu os mesmos passos da seção 4.3, considerando agora o teste do ISA.

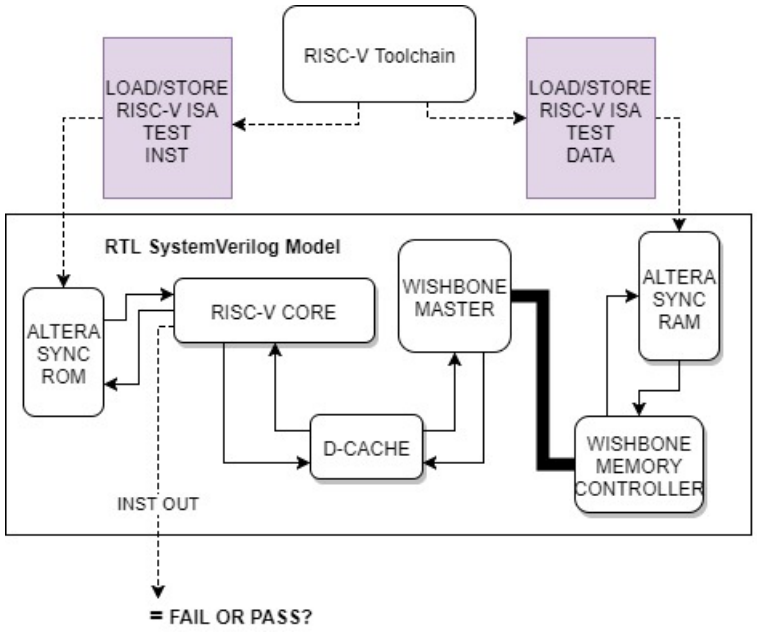

Figura 4.3 *Setup* do *testbench* para validação do modelo RTL da arquitetura proposta integrada com a cache.

4.5 Síntese e prototipação em FPGA

Realizamos a síntese através do *software Quartus II 13.1* da Altera. A plataforma destino foi FPGA Cyclone IV GX presente na placa DE2i-150 da Terasic. Neste processo foram gerados o relatório com os dados de ocupação da FPGA em elementos lógicos, registradores e pinos, também o arquivo '.sof' necessário para realizar a prototipação na FPGA.

Após a prototipação visualizamos o estado atual do processador através dos LEDs da placa configurados para indicar se o processador está estado de execução, falha ou sucesso.

CAPÍTULO 5

Implementação

Neste capítulo descreveremos os detalhes da implementação tanto do modelo de referência em linguagem C, do modelo RTL da cache configurável desenvolvido em SystemVerilog e do modelo em nível de sistema utilizado para validar em cache em FPGA.

5.1 Modelo de Referência

Baseado nos requisitos do projeto foi criado um modelo de referência desenvolvido em linguagem C utilizado para validar o modelo RTL em SystemVerilog. Adicionalmente a este modelo, foram criados dois geradores aleatórios também em linguagem C (Fig. 5.1): o *instruction gen* gera um arquivo com uma sequência aleatória de instruções apenas de leitura e escrita, que são executadas pelo módulo *fake processor*, que gera requisições de leitura e escrita para memória cache; e o *mem gen* que gera um arquivo de inicialização para o módulo *fake ram* – a memória principal do modelo. Após o término da execução é gerado um arquivo '.mif' com o estado final da memória principal, que posteriormente foi utilizado na validação do modelo RTL.

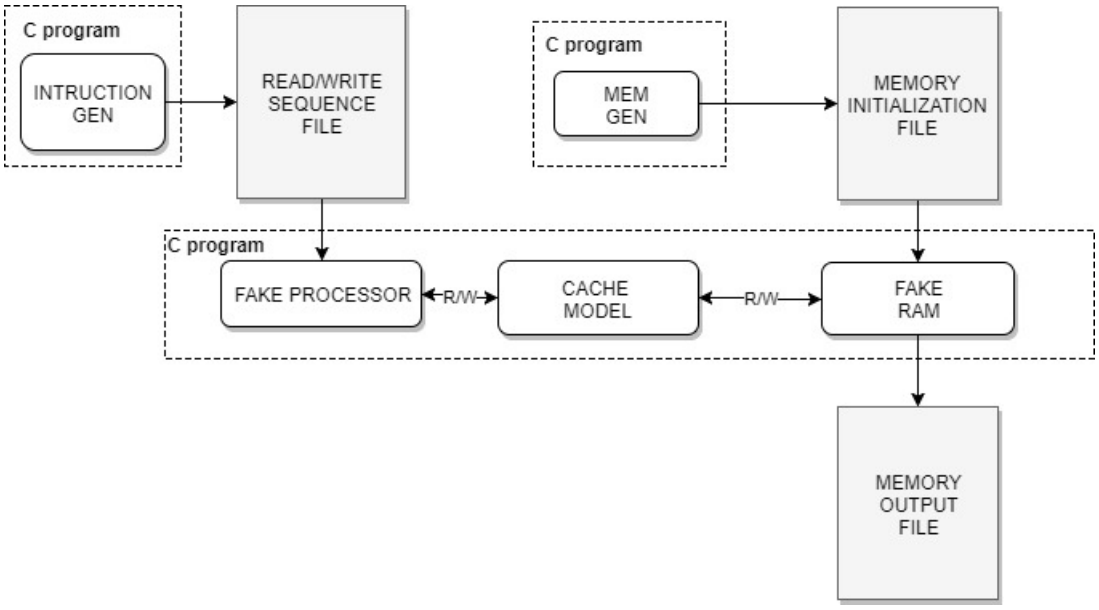

Figura 5.1 Modelo de referência.

5.1.1 *Instruction Gen*

O modulo *Instruction Gen* gera uma sequência de instruções de leitura e escrita da memória (*loads* e *stores*) de forma aleatória. A partir dessa sequência de instruções é gerado um arquivo (Fig. 5.2) que serve como entrada tanto para o modelo de referência como para o *testbench* do modelo da cache.

```
1 LD 0000078c
2 LD 00000314
3 LD 000001c0
4 ST 000000bc 000035e8
5 ST 0000071c 00000683
```

Figura 5.2 Trecho do arquivo contendo sequência aleatória de *loads & stores*.

5.1.2 *Mem Gen*

Gera um vetor de inteiros aleatórios que são usados para construir um arquivo '.mif' utilizado para inicializar as memórias RAM do modelo de referência e do modelo RTL.

5.1.3 Módulo Principal

Dividimos este módulo em três partes (Fig. 4.2), o *fake processor* é a função *main* do programa que lê as instruções do arquivo gerado pelo *instruction gen* e utiliza as funções do *cache model* – uma biblioteca em C chamada *cache.h* contendo uma *struct* para representar a estrutura da cache e funções para simular as operações e funcionamento. Nesta biblioteca utilizamos constantes definidas pelo recurso *define* para parametrizar a cache, sendo necessário sua modificação sempre que se desejar mudar a configuração da cache simulada. A *fake ram* é apenas um vetor onde o índice é o endereço de memória e o conteúdo, os dados daquele endereço.

5.2 Modelo RTL de Cache

5.2.1 Parametrização

Para parametrização do modelo RTL foi criado um *package* de SystemVerilog (cache_parameters.sv) contendo os parâmetros definidos pelo tipo *parameter* e descritos na tabela 5.1. Para a repetição de blocos de *hardware* que são dependentes dos parâmetros utilizamos o recurso *generate*.

Tabela 5.1 Variáveis *parameters* da implementação do modelo RTL da cache configurável.

Parâmetro	**Definição**	**Descrição**
ADDR_WIDTH	Definido pelo projetista.	Quantidade de *bits* dos endereços das posições memória. É dependente do tamanho da memória principal.
WORD_WIDTH	Definido pelo projetista	Quantidade de *bits* de uma *word*.
BLOCK_SIZE	Definido pelo projetista.	Quantidade de *words* em um bloco de dados da cache.
NUMBER_OF_SETS	Definido pelo projetista.	Associatividade da cache. (Se for igual a 1 a cache funcionará como *Direct Mapped*).
CACHE_SIZE	Definido pelo projetista	Tamanho total (contando com todos os *sets*) em *words* da cache.
LINES_PER_SET	Calculado Internamente.	Quantidade de linhas por *set* da cache.
LINE_WIDTH	Calculado Internamente.	Quantidade de *bits* para identificar a linha da cache.
OFFSET_WIDTH	Calculado Internamente.	Quantidade de *bits* para identificar o *offset* do bloco da cache.
TAG_WIDTH	Calculado Internamente.	Quantidade de *bits* para identificar a *tag* da cache.
SET_COUNT_WIDTH	Calculado Internamente.	Quantidade de *bits* para um contador utilizado no algoritmo de substituição FIFO da cache.
TAG_MSB	Calculado Internamente.	*Bit* mais significativo da *tag* no endereço de uma posição de memória.
TAG_LSB	Calculado Internamente.	*Bit* menos significativo da *tag* no endereço de uma posição de memória
LINE_MSB	Calculado Internamente.	*Bit* mais significativo da linha da cache no endereço de uma posição de memória.
LINE_LSB	Calculado Internamente.	*Bit* menos significativo da linha da cache no endereço de uma posição de memória.
OFFSET_MSB	Calculado Internamente.	*Bit* mais significativo do *offset* do bloco no endereço de uma posição de memória.
OFFSET_LSB	Calculado Internamente.	*Bit* menos significativo do *offset* do bloco no endereço de uma posição de memória

5.2.2 Interface

A interface da cache é definida por meio do recurso *struct* de SystemVerilog, que aglutina os diversos sinais dentro de um mesmo bloco. As *structs* criadas, com seus sinais internos e a descrição desses sinais estão na Tabela 5.2. Também foi definida no *package* "cache_parameters.sv".

Tabela 5.2 *Structs* da interface da cache e descrições de seus componentes.

Struct	***Logic Signal***	**Descrição**
processor_request_t	addr	Endereço de dado solicitado pelo processador.
	cs	Sinal para indicar requisição válida.
	rw	Sinal que indica leitura = 0, escrita = 1.
	data	Dado em caso de escrita.
processor_response_t	hold_cpu	Sinal indicador para bloquear CPU.
	data	Dado em caso de leitura.
memory_request_t	addr	Endereço de bloco de dados solicitado pela cache.
	cs	Sinal que indica se a requisição é válida.
	rw	Sinal que indica leitura = 0, escrita = 1.
	data	Bloco de dados em caso de escrita.
memory_response_t	ack	Sinal para indicar que requisição foi respondida.
	data	Bloco de dados em caso de leitura.

5.2.3 Unidade de Controle

Nesta secção descreveremos a máquina de estados projetada e desenvolvida para controle da memória cache. A máquina de estados desenvolvida é uma *mealy machine*, ou seja os sinais de saída dependem não apenas do estado atual, mas também dos sinais de entrada. Os seus estados e transições estão representados na Fig. 5.3. Seus estados são: *flush, idle, rw_op, def_set_replaced, write_back* e *allocate*. Iremos agora descrever a função de cada um deles:

- *flush* – Estado inicial, sempre que a cache sofre *reset*, entra neste estado que varre todos os *valid bits* setando-os para zero. Durante este estado, a cache bloqueia a execução da CPU.
- *idle* – Após a realização do *flush* a cache entra em estado de espera, liberando a CPU. Se houver uma requisição válida de leitura ou escrita a cache passa para o estado *rw_op*.
- *rw_op* – O estado rw_op (read & write operation) checa se a posição de memória requisitada está na memória cache, se não estiver (*cache miss*), vai para o estado *def_set_replaced*, se estiver (*cache hit*), executa a operação e volta para estado *idle*.
- *def_set_replaced* – Neste estado, def_set_replaced(*define set replaced*), a cache define em qual dos *sets* da cache será alocado o dado que virá da memória. Além disso checa se

há um dado válido com *dirty bit* setado na posição da cache que será alocada, se houver vai para o estado *write_back*, se não para o estado *allocate*.

- *write_back* – Neste estado a cache escreve dados que vão ser descartados de volta para a memória principal. Quando a memória confirma que os dados foram recebidos com sucesso, passa para o estado *allocate*.
- *allocate* – O estado *allocate* requisita os dados da memória principal para carregar em cache, espera até que a transferência desses dados seja completada, então volta para o estado *rw_op*.

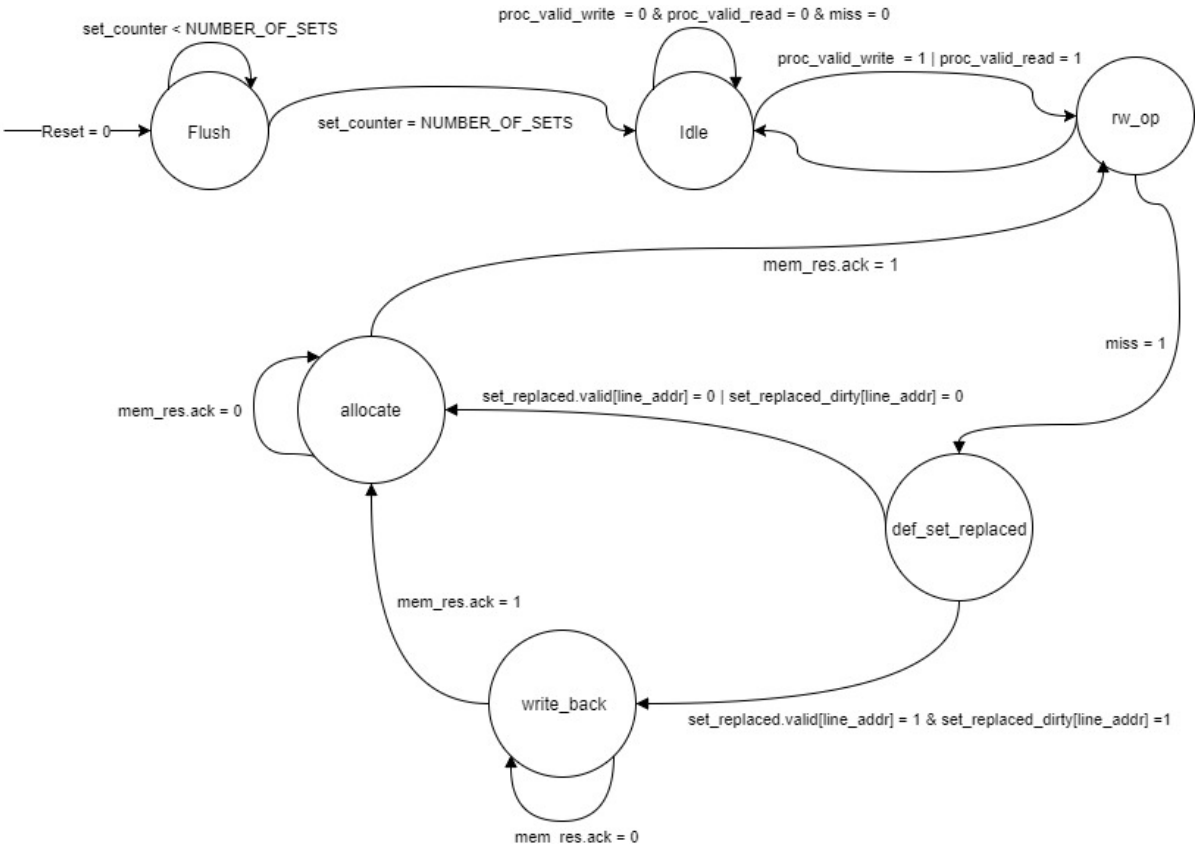

Figura 5.3 Máquina de estados de controle da memória cache.

5.3 Modelo RTL de sistema

Para validar a cache em um sistema real, montamos a arquitetura apresentada na figura 5.4 com os seguintes componentes:

- Um *core* RISC-V.

- Uma memória ROM de instruções com 8KB
- Uma instância da cache implementada neste projeto servindo como cache de dados (testamos quatro configurações diferentes).
- Uma memória RAM de dados com 8KB.
- Um barramento do padrão WISHBONE conectando a cache com a memória RAM.

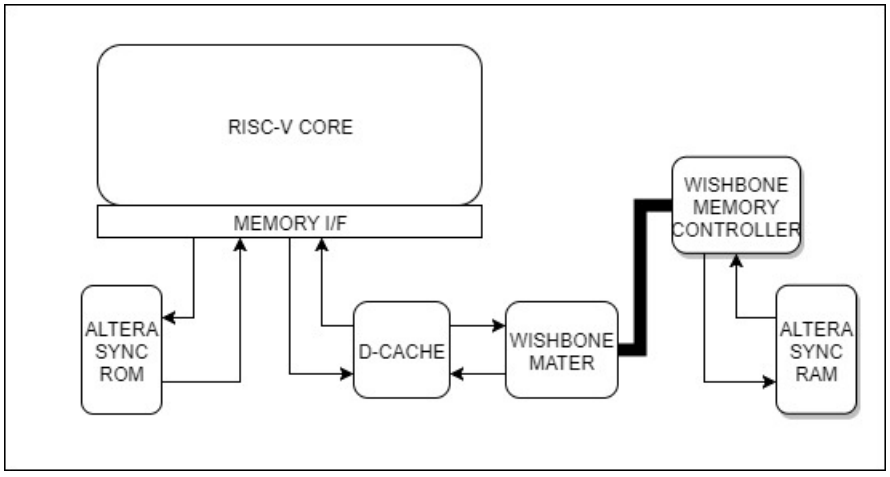

Figura 5.4 Arquitetura de sistema proposta.

Como já detalhamos no capítulo 3, utilizamos o modelo RTL de *core* RISC-V desenvolvido em [Ogg16], um pipeline de 3 estágios, com ISA RV32IM – ou seja, possui endereçamento de 32 bits e executa instruções aritméticas de operações com inteiros mais a extensão de operações de multiplicação e divisão.

As memórias ROM e RAM são IP-CORE da Altera para prototipação de memórias internas em suas FPGAs. São previamente carregadas com um arquivo '.mif', sendo um o binário das instruções e outro um binário com dados gerados pela *toolchain* do projeto RISC-V. Estão endereçadas de acordo com a figura 5.5. Esse mapeamento é feito pelo módulo *memory i/f* da figura 5.4.

RANGE ADDRESS	MEMORY REGION
0x0000 - 0x2000	Processor Reserved
0x2000 - 0x4000	Inst ROM
0x4000 - 0x6000	Data RAM

Figura 5.5 Endereçamento do sistema.

Para comunicação entre os módulos utilizamos o padrão WISHBONE de barramento [Wis02]. Um padrão de barramento aberto e flexível criado para facilitar a integração entre IP cores e

incentivar o seu reuso. Os sinais padrões da interface estão mostrados na figura 5.6. Os modelos RTL dos módulos WISHBONE MASTER e WISHBONE MEMORY CONTROLLER também foram desenvolvidos neste trabalho em SystemVerilog.

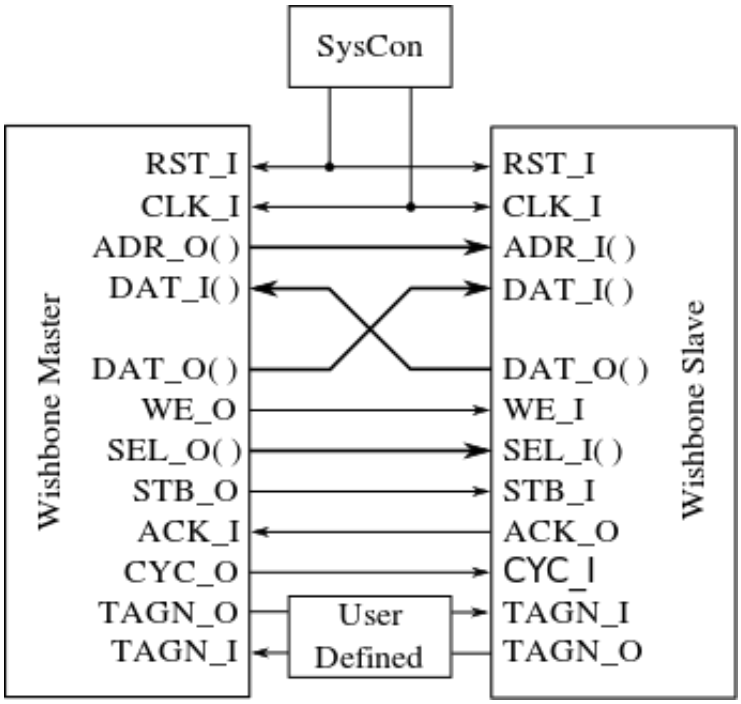

Figura 5.6 Barramento wishbone.

No ciclo básico de leitura (Fig. 5.7), o mestre começa a comunicação setando sinal de nível lógico alto em STB_O e em CYC_O para indicar que o barramento e está ocupado e indicar requisição válida respectivamente. O sinal WE_O é mantido em nível lógico baixo para indicar ciclo de leitura e o endereço desejado é colocado na porta ADR_O. O escravo quando pronto completa a transação enviando nível lógico alto na porta ACK_I e o dado requisitado na porta DAT_I.

No ciclo básico de escrita (Fig. 5.8), o mestre inicia a comunicação da mesma forma, porém o sinal WE_O é setado para nível lógico alto para indicar ciclo de leitura e o endereço desejado é colocado na porta ADR_O junto com o dado a ser escrito na porta DAT_O. O escravo quando pronto completa a transação enviando nível lógico alto na porta ACK_I.

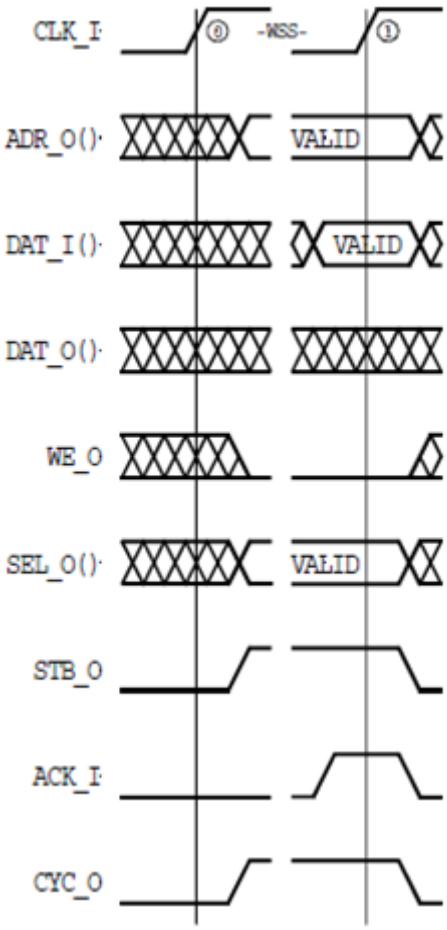

Figura 5.7 Leitura simples no barramento WISHBONE.

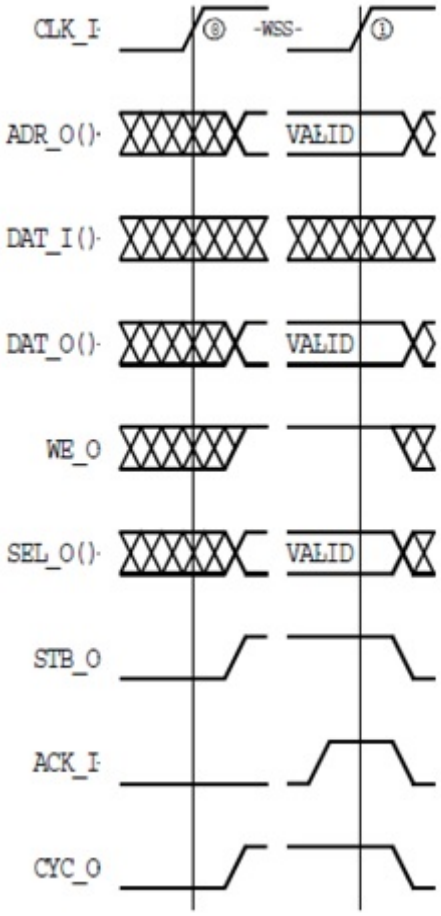

Figura 5.8 Escrita simples no barramento WISHBONE.

CAPÍTULO 6

Resultados e Discussão

Neste capítulo iremos descrever os testes e simulações realizados para validar a tanto a memória cache isoladamente, como a plataforma baseada no processador RISC-V e que inclui a cache projstada (cache de dados). Também incluiremos os resultados da prototipação na FPGA Cyclone IV GX para quatro diferentes configurações de cache.

6.1 Validação do modelo da Cache

Para validação funcional e em *timing* da cache foram utilizados os esquemas das figuras 4.2 já descrito no Capítulo 4. Comparando os arquivos de saída das memórias dos modelos em C e SystemVerilog, validamos as configurações de cache da tabela 6.1 utilizando 10000 instruções de leitura e escrita para cada configuração. Nesta etapa repetimos os testes até que todas essas configurações obitivessem sucesso na validação.

Tabela 6.1 Parâmetros configuráveis para os quais o modelo de cache foi validado.

Capacidade	Associatividade	Tamanho do Bloco
512 Bytes	*1-Way*	1 *word*
	2-Way	4 *words*
1KB	*2-Way*	2 *words*
	4-Way	4 *words*
2KB	*4-Way*	4 *words*
	8-Way	8 *words*

Na Fig. 6.1 temos um exemplo de falha nessa validação, onde podemos visualizar que alguns endereços de memória contém dados diferentes entre os dois arquivos de saída das memórias de referência e simulação. Já na Fig. 6.2 observamos um caso em que houve sucesso, ou seja todos os endereços possuem dados idênticos.

6.2 Validação do modelo do Sistema

O ISA RISC-V é uma arquitetura *load/store*, ou seja as únicas instruções que acessam a memória são as instruções de *load* e *store*. A *toolchain* do RISC-V fornece um código fonte

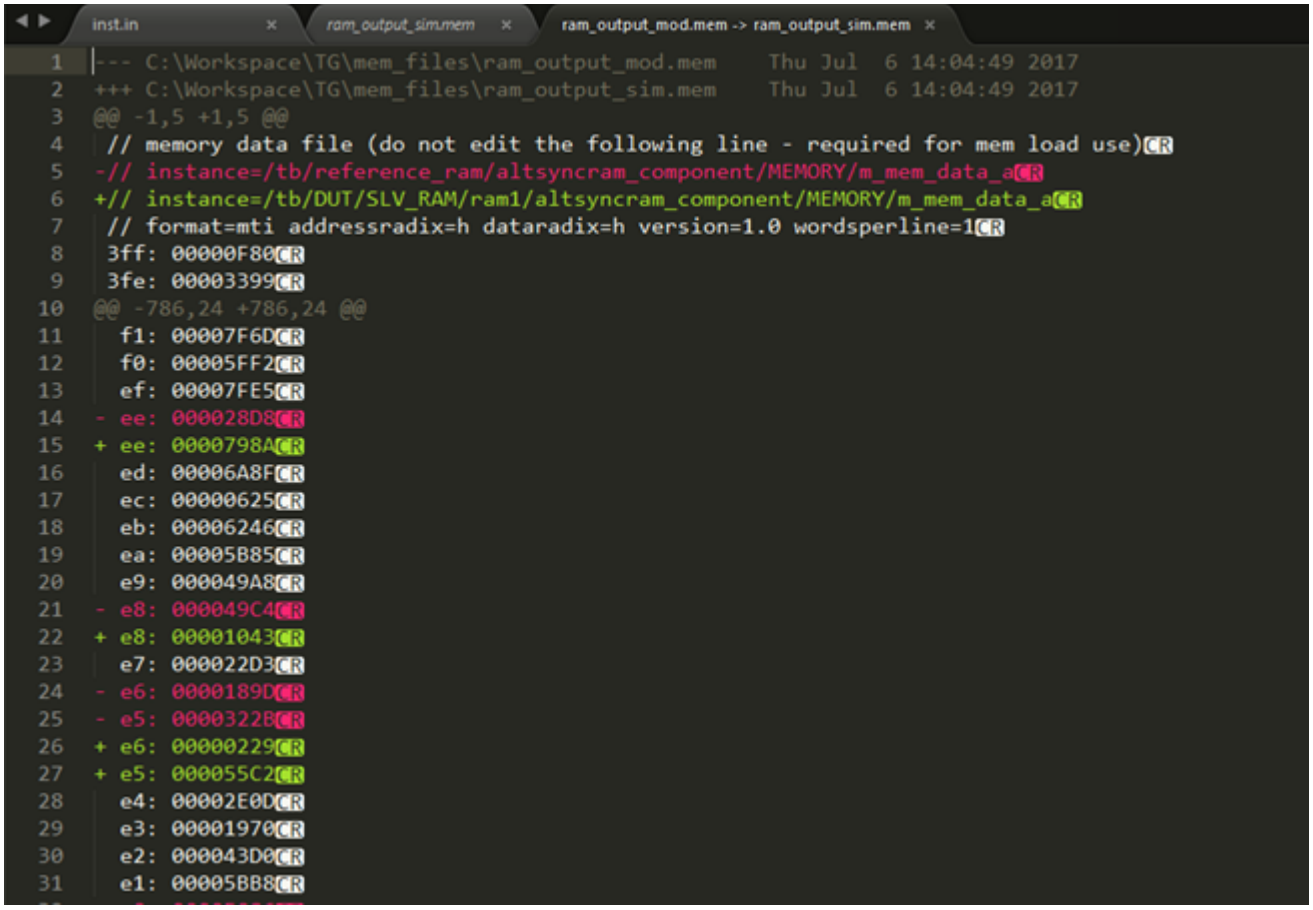

Figura 6.1 Comparação de arquivos contendo o conteúdo da memória ao fim da execução dos modelos de referência e RTL. O texto em verde são conteúdo do arquivo de saída do modelo RTL, em vermelho são do arquivo de saída do modelo de referência e o texto em branco significa conteúdo idêntico de ambos arquivos. Este é um exemplo de falha na validação do modelo RTL.

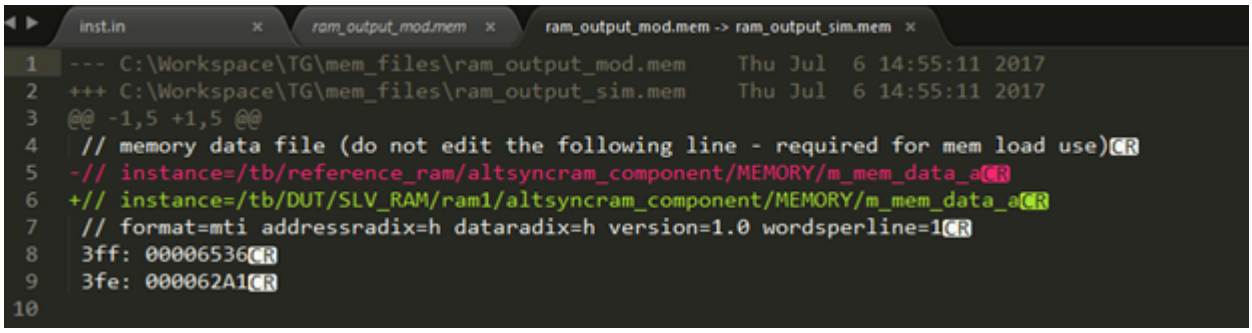

Figura 6.2 Comparação de arquivos contendo o conteúdo da memória ao fim da execução dos modelos de referência e RTL. O texto em verde são conteúdo do arquivo de saída do modelo RTL, em vermelho são do arquivo de saída do modelo de referência e o texto em branco significa conteúdo idêntico de ambos arquivos. Este é um exemplo de sucesso na validação do modelo RTL.

contendo um teste para cada instrução do ISA. A implementação do RISC-V utilizada neste trabalho [Ogg16] já foi validada para todas as 47 instruções do subconjunto RV32MI do ISA RISC-V. Porém, como neste projeto houve modificação no acesso a memória, revalidamos as instruções *lw*, *sw*, *sh*, *lh*, *sb* e *lb* utilizando os testes fornecidos pela *toolchain* do *RISC-V*.

Nessa validação cada instrução passa por uma bateria de testes em sequência em que se houver falha o programa desvia para instrução de falha onde a execução é bloqueada. Caso contrário, o programa continua a bateria de testes até chegar na instrução de sucesso onde a execução é bloqueada. Na fig. 6.3 vemos o exemplo de um trecho do código em *assembly* do teste da instrução *sb*, está marcado em vermelho a instruções de desvio para a instrução de falha caso o teste falhe.

Na fig. 6.4 temos o trecho de código contendo as instruções de falha e sucesso da instrução *sb* marcadas de vermelho e azul respectivamente, e na fig. 6.5 temos a simulação da instrução *sb* exemplificando um caso de falha, com a execução bloqueada na instrução de falha marcada em vermelho.

Na tabela 6.2 temos o resultado do teste executados. Conseguimos 100% de sucesso nos testes das instruções *lw*, *sw* e *lh*. Porém nas outras 3 instruções não obtivemos sucesso em nenhum teste. Verificamos que um erro no mecanismo de *stall* da CPU é a causa deste problema, a cache desenvolvida envia corretamente o sinal para bloquear a CPU mais o PC é incrementado mesmo assim.

Tabela 6.2 Teste do ISA RV32MI.

Instrução	Sucesso	Total de Testes
lw	18	18
sw	22	22
lh	18	18
sh	0	22
lb	0	18
sb	0	22

6.3 *Benchmarking*

Para avaliar o desempenho de algumas configurações de cache no sistema desenvolvido, foi executado em simulação o teste *matrix multiplication* do *benchmarking* TI embedded Benchmark Suite (Tabela 6.3). Para garantir um resultado real, verificou-se que nenhuma das instruções que falharam na validação do ISA eram executadas nesse teste.

```
18 00002014 <test_2>:
19     2014:   00002097            auipc   ra,0x2
20     2018:   fec08093            addi    ra,ra,-20 # 4000 <begin_signature>
21     201c:   faa00113            li  sp,-86
22     2020:   00208023            sb  sp,0(ra)
23     2024:   00008f03            lb  t5,0(ra)
24     2028:   faa00e93            li  t4,-86
25     202c:   00200193            li  gp,2
26     2030:   3ddf1c63            bne t5,t4,2408 <fail>
27
28 00002034 <test_3>:
29     2034:   00002097            auipc   ra,0x2
30     2038:   fcc08093            addi    ra,ra,-52 # 4000 <begin_signature>
31     203c:   00000113            li  sp,0
32     2040:   002080a3            sb  sp,1(ra)
33     2044:   00108f03            lb  t5,1(ra)
34     2048:   00000e93            li  t4,0
35     204c:   00300193            li  gp,3
36     2050:   3bdf1c63            bne t5,t4,2408 <fail>
37
38 00002054 <test_4>:
```

Figura 6.3 Trecho de código na memória ROM do teste da instrução *sb*.

```
315 00002408 <fail>:
316     2408:   0ff0000f            fence
317     240c:   00018063            beqz    gp,240c <fail+0x4>
318     2410:   00119193            slli    gp,gp,0x1
319     2414:   0011e193            ori gp,gp,1
320
321 00002418 <f>:
322     2418:   000000ef            jal ra,2418 <f>
323
324 0000241c <pass>:
325     241c:   0ff0000f            fence
326     2420:   00100193            li  gp,1
327
328 00002424 <p>:
329     2424:   000000ef            jal ra,2424 <p>
330     2428:   c0001073            unimp
331     242c:   0000                    unimp
```

Figura 6.4 Trecho de código na memória ROM do teste da instrução *sb* onde estão as instruções finais de falha ou sucesso.

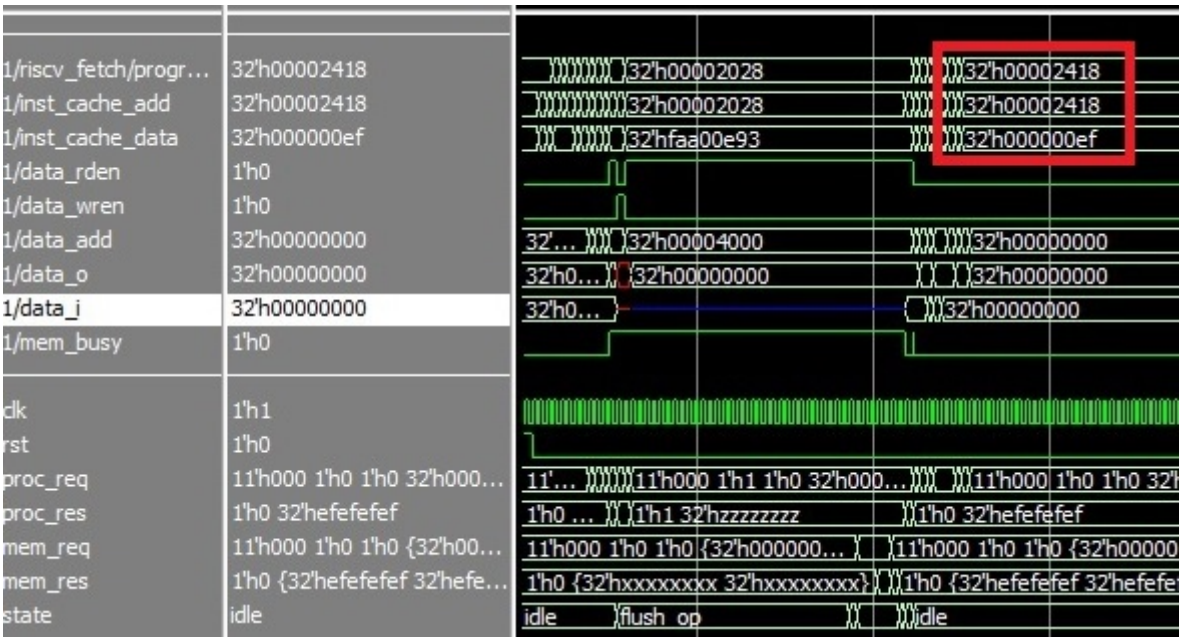

Figura 6.5 Exemplo da simulação da instrução *sb* falhando. Podemos verificar marcado em vermelho que o PC parou sua execução na instrução que indica falha de execução do teste. Esta instrução está marcada em vermelho na fig. 6.4.

Tabela 6.3 Teste *matrix multiplication* (total de 840 instruções) do *benchmarking* TI embedded Benchmark Suite em caches com diversas configurações. *bytes*.

Tamanho	Associatividade	Tamanho do Bloco	Ciclos de *Clock*	CPI	Miss Rate
64 Bytes	*2-Way*	8 Bytes	1706	2,03	25,88%
	4-Way	8 Bytes	1608	1,91	22,35%
	2-Way	16 Bytes	2468	2.94	37,25%
128 Bytes	*2-Way*	8 Bytes	1590	1,89	20%
	4-Way	8 Bytes	1372	1,63	12,16%
	2-Way	8 Bytes	1712	2,04	16,08 %
256 Bytes	*2-Way*	8 Bytes	1304	1,55	9,41%
	4-Way	8 Bytes	1296	1,54	9,41%
	2-Way	16 Bytes	1248	1,49	4,7%

6.4 Síntese e Prototipação em FPGA

Após as validações garantindo o funcionamento do modelo, realizamos a síntese do modelo do sistema para a FPGA Cyclone IV GX utilizando quatro configurações de cache diferentes, os dados de ocupação da FPGA para cada uma das configurações estão na fig. 6.6 e na tabela 6.4. Como podemos observar na fig. 6.6, quando aumentamos o tamanho da cache de 512 Bytes para 1KB houve uma aumento significativo nos recursos utilizados da FPGA, não houve grande diferença quando duplicamos a associatividade, porém houve uma diminuição na utilização desses recursos chegando a se aproximar a cache de 512 Bytes. É interessante observar que essas caches que utilizaram menos recursos tem a mesma quantidade de linhas totais, enquanto as que usaram mais tem o dobro da quantidade de linhas totais que essas e permaneceram praticamente iguais. Isso nos dá um indicativo que a quantidade de linhas é o fator que mais influencia o espaço ocupado pela cache na FPGA. Porém, ainda é necessário maior investigação para resultados mais conclusivos.

Figura 6.6 Dados de ocupação da FPGA para diversas configurações de cache.

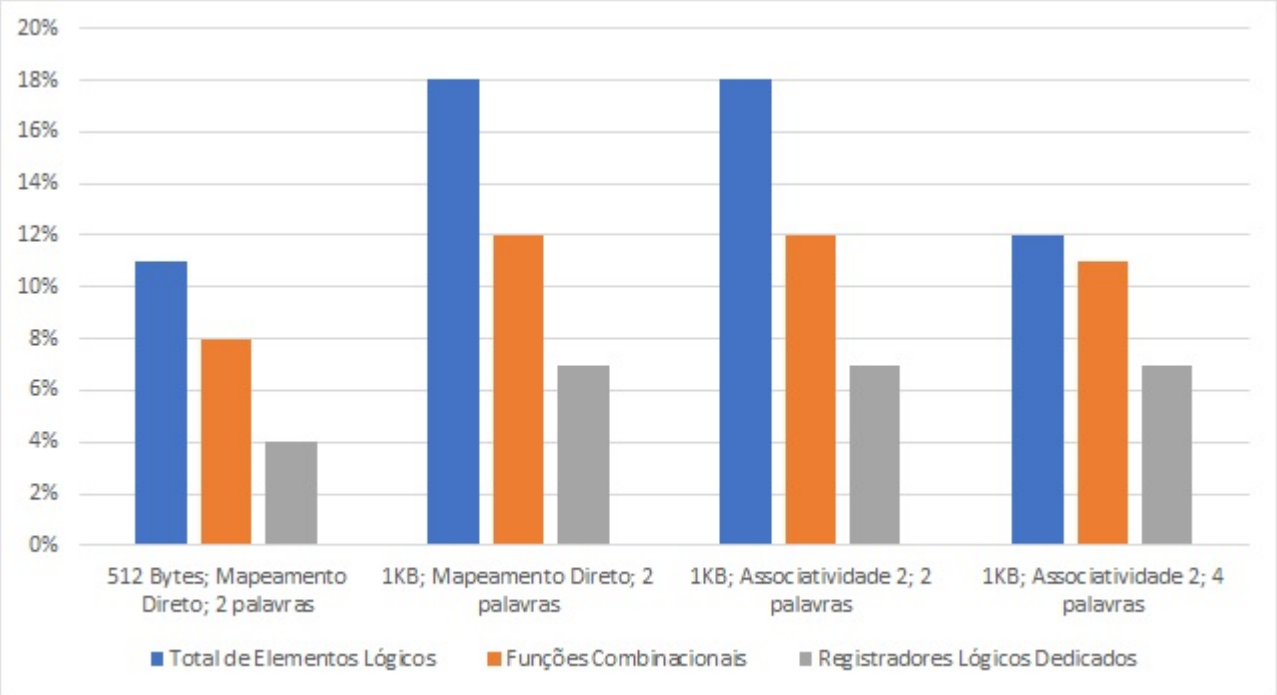

Tabela 6.4 Dados comuns as configurações da fig. 6.6 da ocupação da FPGA para a síntese do sistema.

	Utilizados	Totais	%
Total de Pinos	30	508	6%
Elementos multiplicadores	8	720	1%
Memória ocupada	131,072 Kbits	6,635,520	2%

CAPÍTULO 7

Conclusão e Trabalhos Futuros

Foi desenvolvido neste trabalho um modelo RTL configurável de memória cache genérico em SystemVerilog. Este modelo possui 3 parâmetros configuráveis que são: a capacidade da cache, a associatividade e o tamanho do bloco. O modelo foi validado para os parâmetros definidos na seção 6.1 utilizando 10000 instruções de leitura e escrita. Foi utilizando como referência um modelo em linguagem C também desenvolvido neste projeto que também é parametrizável e pode servir para validar outras configurações de parâmetros. O modelo RTL de cache desenvolvido possui política de escrita *write back*, algoritmo de substituição *FIFO* e penalidade de 1 ciclo para leitura e escrita. Também integramos a cache em um sistema contendo um *core* de arquitetura RISC-V para validação em um sistema real, e utilizamos o conjunto de teste do ISA fornecido pela *toolchain* do projeto RISC-V para validar as instruções de *load* e *store*. O sistema final passou por simulação sendo validado funcionalmente, e em *timing* considerando o atraso real das portas lógicas. Foi realizada a síntese de quatro configurações de cache diferentes para a FPGA Clyclone IV GX da Altera, obtendo para a mais simples dessas configurações 11% dos elementos lógicos totais e 2% da memória ocupada com uma frequência de clock de 50MHz. Como trabalhos futuros, sugere-se a parametrização da cache para outras políticas de substituição como *least recently used* (LRU) e substituição aleatória, e também para a política de escrita *write through*. Também sugere-se montar uma arquitetura para testar a cache desenvolvida em múltiplos níveis, e ainda modificação da cache para ser integrada em sistemas *multi-core*.

Referências Bibliográficas

[Acc04] Accellera Organization, Inc. *SystemVerilog 3.1a Language Reference Manual*, 3.1a edition, 2004.

[GAG+09] D.D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, et al. *Embedded System Design*. Springer, 2009.

[KS16] Swadhesh Kumar and P K Singh. An overview of modern cache memory and performance analysis of replacement policies. *2nd IEEE International Conference on Engineering and Technology*, 2016.

[Ogg16] V. O. Ogg. Modelo rtl do processador risc-v sintetizável em fpga. Programa de graduação em engenharia da computação, Universidade Federal de Pernambuco, Centro de Informática, 2016.

[PH05] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3rd edition, 2005.

[PH12] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[SB+12] Vahid Saljooghi, Alen Bardizbanyan, et al. Configurable rtl model for level-1 caches. Proc. of the Norchip Conference, November 2012.

[WA17] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. RISC-V Foundation, 2.2 edition, 05 2017.

[Wis02] Herveille R. Wishbone. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores Organization, b.3 edition, 09 2002.

[WL+13] A Waterman, Y Lee, et al. The risc-v instruction set. *IEEE HOT CHIPS 25 SYMPOSIUM*, 2013.