UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SUPORTE ESPECIALIZADO DE HARDWARE PARA GERAÇÃO AUTOMÁTICA DE LOOP PIPELINING EM FPGAS

GUILHERME STÉFANO SILVA DE SOUZA

ORIENTADOR: MÁRCIO MERINO FERNANDES

São Carlos – SP

Outubro/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SUPORTE ESPECIALIZADO DE HARDWARE PARA GERAÇÃO AUTOMÁTICA DE LOOP PIPELINING EM FPGAS

GUILHERME STÉFANO SILVA DE SOUZA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Arquitetura de Computadores.
Orientador: Márcio Merino Fernandes

São Carlos – SP

Outubro/2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

S729se

Souza, Guilherme Stefano Silva de.
Suporte especializado de hardware para geração automática de *loop pipelining* em FPGAS / Guilherme Stefano Silva de Souza. -- São Carlos : UFSCar, 2015.
238 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2014.

1. Arquitetura de computador. 2. Processamento paralelo (Computadores). 3. *Loop pipelining*. 4. Escalonamento. I. Título.

CDD: 004.35 (20ª)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de dissertação de Mestre em Ciência da Computação do candidato Guilherme Stéfano Silva de Souza, realizada em 19/11/2014:

Prof. Dr. Márcio Mérino Fernandes
UFSCar

Prof. Dr. Edson Borin
UNICAMP

Prof. Dr. Emerson Carlos Pedrino
UFSCar

Aos meus pais, irmão e amigos por estarem sempre ao meu lado.

AGRADECIMENTOS

Ao meu orientador, Prof. Dr. Márcio Merino Fernandes, pela sua paciência e por compartilhar de sua experiência, tempo e disposição contribuindo para a conclusão deste trabalho. Ao Prof. Dr. Emerson Carlos Pedrino, por ter me auxiliado em algumas etapas do projeto. Aos professores do departamento de Computação da Universidade Federal de São Carlos, por passarem grande parte do conhecimento necessário para a minha formação acadêmica e profissional.

Aos meus pais e irmão, por sempre estarem ao meu lado. Aos meus amigos de longa data, por nunca me deixarem desanimar durante o desenvolvimento deste projeto. Aos amigos que adquiri nos corredores da UFSCAR, pelo companheirismo nos dias sofridos do desenvolvimento deste trabalho.

A felicidade às vezes é uma bênção, mas geralmente é uma conquista.

Paulo Coelho

RESUMO

O desempenho na execução de programas, que é cada vez mais uma prioridade, pode ter uma melhora significativa por meio do uso de paralelismo em nível de instrução (ILP). Uma técnica que utiliza o ILP e propicia ganhos de desempenho significativos é o *loop pipelining*, sendo usado não apenas por compiladores para microprocessadores, mas também por ferramentas de Síntese de Alto Nível (HLS), visando arquiteturas heterogêneas e aceleradores de *hardware*. Neste trabalho é apresentado o projeto e implementação de estruturas de *hardware* especializadas, objetivando-se em solucionar o problema de sobreposição de valores que ocorre no *loop pipelining*, facilitar tarefas de compilação em ferramentas HLS e diminuir a repetição de código. Além disso, ganhos potenciais de desempenho e área de silício total podem ser alcançados como resultado do uso das estruturas propostas. Serão apresentados: um arquivo de registradores baseado em filas e um módulo de controle para a execução de instruções predicadas.

Palavras-chave: *Loop Pipelining*, *Software Pipelining*, Escalonamento de Módulo, QRF, Arquivos de Registradores, *Queued Register File*, Filas, Instruções Predicadas

ABSTRACT

Loop pipelining is a technique that may offer significant performance improvements, being employed not only in conventional compilation targeting microprocessors, but also by High Level Synthesis (HLS) tools, targeting heterogeneous architectures and hardware accelerators. This work presents a specialized hardware support aiming at facilitate compilation tasks for HLS tools, along with potential advantages in execution performance and total silicon area employed. Two specialized hardware modules are presented: a queue register file and an instruction predication control module.

Keywords: Loop Pipelining, Software Pipelining, Modulo Scheduling, QRF, Register Files, Queued Register File, Queue, Predicated Instructions

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGENS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Neste capítulo é apresentada a proposta do projeto de pesquisa. Para melhor compreensão, este está subdividido em Introdução, Objetivos Gerais, Objetivos Específicos, Justificativas, Metodologia e Organização do Trabalho.

1.1 Introdução

A execução de algoritmos que demandam alta carga de processamento tem sido buscada cada vez mais; para isso, é necessária a melhoria do desempenho na execução de programas. Uma forma para melhorar o desempenho é utilizar o paralelismo em nível de instrução (ILP), onde pode ser utilizado um compilador para paralelizar o código. O objetivo de várias técnicas que exploram o ILP é paralelizar os *loops*, que muitas vezes são responsáveis por grande parte do tempo na execução de um programa. O *software pipelining* ou *loop pipelining* é uma das técnicas para paralelizar *loops* por sobreposição de operações (ou execução simultânea) de várias iterações consecutivas. A ideia por trás do *loop pipelining* é que, o corpo de um *loop* pode ser reestruturado de modo que uma iteração possa começar antes de iterações anteriores concluírem a execução (JONES; ALLAN, 1990).

A técnica de *loop pipelining* foi inicialmente desenvolvida para arquiteturas do tipo VLIW (*Very Long Instruction Word*), que portanto exploram o ILP, mas fazem isso em tempo de compilação e dependem do compilador para explorar o ILP efetivamente. Um problema que ocorre na técnica de *loop pipelining* é a alocação de registradores. Os requisitos de um arquivo de registradores podem ser determinados pelo tempo de vida das variáveis no *loop*. O tempo de vida de uma variável pode ser medido pelo momento em que o valor dessa variável é colocado no registrador e a última vez que esse valor é utilizado, o que pode ser visto como a produção e o consumo dos valores no registrador. Se um valor for sobrescrito antes de ser consumido,

quando for consumido esse valor será incorreto, causando um erro na execução do programa. Isso ocorre se o valor produzido no registrador for consumido somente vários ciclos após ter sido produzido. Existem várias formas de se contornar esse problema: uma delas é implementar o arquivo de registradores no formato de uma fila. Essa técnica é conhecida como QRF (*Queued Register File*). Nessa fila os dados podem ser escritos e lidos uma vez por ciclo. Ao gravar algum valor no registrador, o mesmo será acrescentado na cauda da fila. Desse modo, nenhum valor será sobrescrito, sendo armazenado até que seja retirado da fila.

Para realizar a paralelização, o *loop pipelining* gera replicação de código. O controle de instruções predicadas é uma técnica que pode ser utilizada junto ao *loop pipelining* com o objetivo de deixar o escalonamento mais limpo e fácil de ser gerado pelo compilador, evitando essa replicação de código. Uma forma de aplicar o controle de instruções predicadas ao *loop pipelining* é utilizar registradores de deslocamento.

O *loop pipelining* é utilizado frequentemente em processadores VLIW, pois estes permitem um grande número de unidades funcionais em paralelo. Porém, processadores VLIW são normalmente implementados como ASICs (*Application Specific Integrated Circuit*), ou seja, não oferecem flexibilidade para aumentar o paralelismo conforme for necessário. Uma alternativa é a implementação do *loop pipelining* em FPGA (*Field Programmable Gate Array*), que oferece possibilidade de alto grau de paralelismo e flexibilidade.

Os circuitos de FPGAs oferecem essa possibilidade de explorar o paralelismo de diferentes formas, pois são dispositivos reconfiguráveis e podem assumir o papel de inúmeros circuitos diferentes, dependendo da forma como são programados. Esses dispositivos podem ser programados utilizando linguagens de descrição de *hardware*, como VHDL, Verilog, entre outras.

Este projeto de pesquisa completa o trabalho de Rettore (2012) utilizando as técnicas citadas. O trabalho de Rettore (2012) propõe aproximar a programação de *hardware* customizado em uma linguagem mais comum através do compilador Cetus Modificado, que é uma alteração do compilador Cetus proposto por Lee, Johnson e Eigenmann (2004). O compilador Cetus Modificado recebe um código em C que é transformado em um código paralelizado na forma de *loop pipelining*. Na saída do compilador Cetusl, o *loop pipelining* é estruturado na forma de máquinas de estados. O código que é utilizado para descrever essas máquinas de estados, não é um código em uma linguagem de descrição de *hardware* e não pode ser executado em FPGA.

1.2 Objetivos gerais

O objetivo geral deste trabalho é a construção de estruturas de *hardware* especializadas para dar suporte à execução de *loop pipelining* em FPGAs. A proposta deste projeto de pesquisa é projetar e implementar arquivos de registradores para auxiliar o *loop pipelining* no armazenamento dos valores das interações e, implementar também um módulo para controle de instruções predicadas. O arquivo de registradores foi nomeado de QRF e o módulo de controle foi nomeado de CP. O QRF e o CP foram implementados em um circuito de FPGA, utilizando-se a linguagem Verilog. Para isso foi utilizado o trabalho criado por Rettore (2012) onde a compilação com a técnica *loop pipelining* já foi implementada no Compilador Cetus Modificado. Portanto, este projeto de pesquisa parte do pressuposto de que o *loop pipelining* já está devidamente implementado e funcionando, e propõe duas estruturas de arquivos de registradores que consigam dar suporte a compilação que utiliza paralelismo em nível de instrução (ILP) para acelerar o código.

Este trabalho, então, será uma continuação do trabalho de Rettore (2012). Mas, nada impede que este projeto de pesquisa seja utilizado em qualquer técnica que pretenda explorar o paralelismo em nível de instrução (ILP) utilizando *loop pipelining*. No diagrama da Figura 1.1 é mostrada a relação entre o trabalho proposto e o trabalho de Rettore (2012).

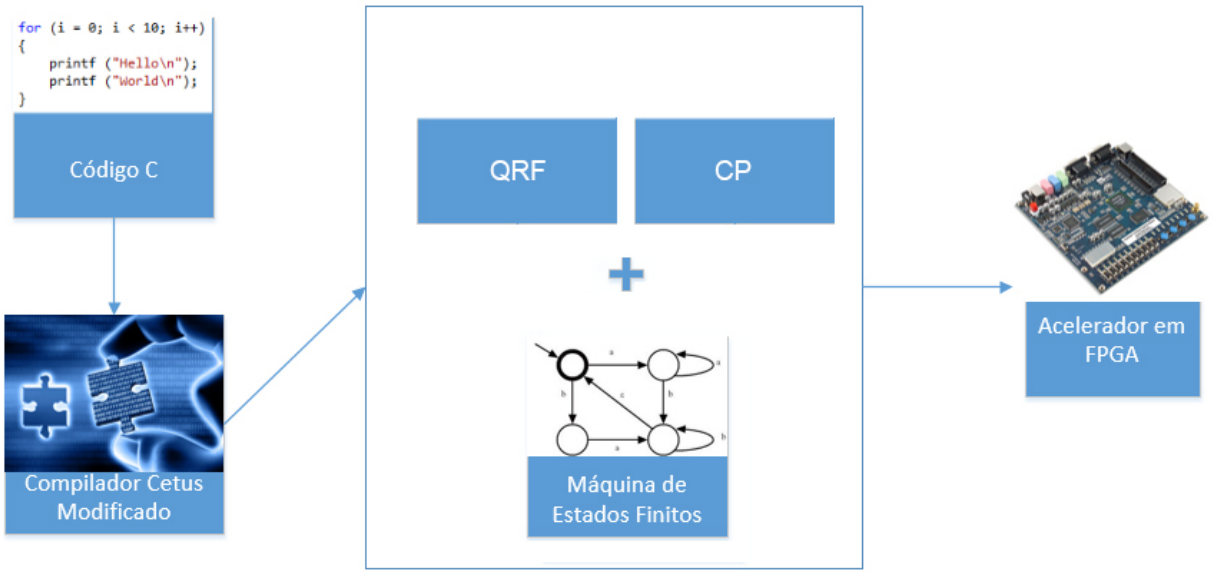

Figura 1.1: O retângulo representa a técnica que foi desenvolvida com o compilador Cetus Modificado.

1.3 Objetivos Específicos

De maneira sucinta, este trabalho, baseia-se nos seguintes objetivos específicos:

- Projetar arquivos de registradores utilizando filas, denominado QRF para solucionar o problema de sobreposição de valores que ocorre no *loop pipelining* a partir do trabalho de Fernandes (1998).
- Projetar um módulo para controle de instruções predicadas denominado CP, para aplicar o controle de instruções predicadas ao *loop pipelining*.
- Implementar o QRF e o CP em FPGA utilizando a linguagem de descrição de *hardware* Verilog.
- Projetar e Implementar uma arquitetura alvo que irá utilizar o QRF e o CP.
- Efetuar análises e testes para verificar se o QRF e CP suprem as exigências do *loop pipelining*.
- Explorar diversas configurações de implementação da arquitetura alvo, definindo diretrizes para futuros projetos envolvendo as técnicas abordadas por este trabalho.

1.4 Justificativa

Para justificar o desenvolvimento desta proposta de trabalho, foram analisados os seguintes aspectos:

- Cada vez mais cresce a demanda por sistemas de computação de alto desempenho, à medida que, as aplicações que necessitam de alto processamento aumentam e não consigam a eficácia desejada devido às restrições da tecnologia atual. Isto incentiva as pesquisas na área de aceleração de processamento.
- O que torna a proposta deste projeto de pesquisa importante, é que o mesmo não irá complementar somente o trabalho anterior, mas também qualquer técnica que pretenda explorar o paralelismo em nível de instrução (ILP) utilizando *loop pipelining*.
- Muitas pesquisas são realizadas nesse meio, mas poucas se preocupam com o problema da sobreposição de valores que ocorre na técnica do *loop pipelining*.

- Algumas técnicas propostas para solucionar o problema têm desvantagens, tais como o aumento do tamanho de código e a perda de desempenho.

- Tratando o problema de sobreposição de valores, novos estudos podem ser realizados buscando melhorar somente o *loop pipelining* sem se preocupar com as restrições de *hardware*.

- O uso de instruções predicadas pode reduzir o tamanho das estruturas de *hardware* em FPGA, pois com essa técnica não há necessidade de replicação de código, o que pode auxiliar a tarefa de compilação.

- Utilizar *loop pipelining* em FPGAs possui vantagens como, flexibilidade, alto grau de paralelismo e baixo custo. Os circuitos de FPGAs e as técnicas de compilação para os mesmos não eram bem desenvolvidos quando as técnicas de *loop pipelining* apareceram (década de 90).

1.5 Metodologia

Para a execução deste trabalho foi preciso um aprofundamento em diversos assuntos. Para o conhecimento em VLIW foi utilizado o livro de Fisher, Faraboschi e Young (2005), e para o *loop pipelining*, basicamente foram utilizados os trabalhos de Allan et al. (1995) e Rau (1994). O conhecimento em computação reconfigurável e em FPGAs foi aperfeiçoado através dos trabalhos de Hamblen, Hall e Furman (2008) e Hauck e Dehon (2008). Para compreender melhor o que é uma ferramenta HLS foi utilizado o trabalho de Coussy et al. (2009) e, para o compilador Cetus Modificado foi utilizado o trabalho de Rettore (2012). Para conhecer técnicas que solucionem o problema do *loop pipelining* foram utilizados os trabalhos de Lam (1988), Rau et al. (1992) e Fernandes (1998). E por fim, para entender os conceitos sobre instruções predicadas foi utilizado o livro de Hennessy e Patterson (2011). Para a implementação dos arquivos de registradores utilizando filas e o controle de instruções predicadas, foram utilizados o Quartus II, a Placa DE-2 da Altera, e a linguagem de descrição de *hardware* Verilog. Para aprofundar o conhecimento na linguagem de descrição de *hardware* Verilog e em sistemas digitais foram utilizados os livros de Tocci, Widmer e Moss (2007), Brown e Vranesic (2003), Chu (2008) e Coffman (2000). E para verificar a efetividade do projeto foram feitas análises e testes para aferir: funcionalidade, desempenho e recursos de *hardware*.

1.6 Organização do Trabalho

O restante desta proposta de trabalho encontra-se organizada da seguinte forma: No Capítulo 2 serão mostrados os principais temas que fundamentam a base teórica desta pesquisa. No Capítulo 3 será mostrado como foi feito o desenvolvimento do trabalho. No Capítulo 4 serão mostrados os resultados experimentais. No Capítulo 5 serão discutidas as contribuições e será feita a conclusão do trabalho. Os códigos de implementação em Verilog e outros dados relacionados à implementação deste trabalho estão disponíveis em forma de Apêndice na parte final deste documento. Busca-se assim possibilitar uma maior clareza sobre como foram obtidos os resultados experimentais apresentados.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é mostrada a base teórica, visando melhorar o entendimento sobre o trabalho em si. Para isso, na Seção 2.1 será mostrado o que é uma arquitetura VLIW, onde será dada uma ideia básica de seu funcionamento, suas vantagens, por que o compilador é importante para essa arquitetura, e sua relação com *loop pipelining*. Na Seção 2.2 será mostrado o que é *loop pipelining*, seu funcionamento, vantagens e dificuldades. Na Seção 2.3, serão mostrados três trabalhos diferentes para resolver o problema da técnica *loop pipelining*. Na Seção 2.4 será apresentado o conceito de instruções predicadas e, como esse conceito pode ser utilizado junto ao *loop pipelining*. Na Seção 2.5 será explicado o conceito de computação reconfigurável, quando surgiu, quais os principais dispositivos, e quais são as suas vantagens em relação a outros tipos de tecnologia. Na Subseção 2.5.1 serão mostrados os conceitos de FPGA, o seu funcionamento e sua arquitetura. Na Seção 2.6 será feita uma breve introdução à linguagem Verilog e sobre projeto de *hardware* digital. Na Seção 2.7 será realizada uma introdução sobre HLS, suas justificativas, e os passos de uma ferramenta HLS. Na Seção 2.8 será mostrado o compilador Cetus, suas funcionalidades, seu funcionamento e sua versão Cetus Modificado. E por último, na Seção 2.9, serão mostrados os trabalhos relacionados.

2.1 Arquitetura VLIW

A produtividade de um processador pode ser aumentada de vários modos e, um deles é fazer com que ele faça mais coisas ao mesmo tempo, e para isso, pode ser utilizado algum tipo de paralelismo em nível de instrução. Emitir múltiplas instruções por ciclo de *clock* é um modo de conseguir o paralelismo. Existem duas formas de emitir múltiplas instruções: através de processadores superescalares e processadores VLIW (*Very Long Instruction word* - palavra de instrução muito longa) (TANENBAUM, 2007). A arquitetura VLIW será explicada nesta seção,

pois é a classe de arquiteturas para a qual o *loop pipelining* foi originalmente criado.

Na arquitetura VLIW o escalonamento do paralelismo em nível de instrução(ILP) é totalmente visível em nível de máquina e compilação, não deixando o *hardware* realizar o escalonamento, pois o mesmo será feito pelo compilador. Já na arquitetura superescalar as operações e escalonamento são feitos pelo *hardware* de um modo que não são visíveis pelo compilador (FISHER; FARABOSCHI; YOUNG, 2005).

Quando foram criadas, as máquinas VLIW, tinham palavras longas que continham instruções que usavam suas várias unidades funcionais podendo executar várias operações simultaneamente, como é mostrado no *pipeline* da Figura 2.1(a). Nessa arquitetura podem ser realizadas duas operações com inteiros, uma operação com ponto flutuante, um *load* e um *store*. As instruções eram ditas longas, porque continham um *opcode* e um par de operandos por unidade funcional. No exemplo da ilustração, a instrução tem cinco pares de operandos e cinco *opcodes*, o que resulta em uma instrução bem comprida. Mas, esse projeto mais antigo obrigava o uso de todas as unidades funcionais, porém, nem todas as instruções podiam utilizar todas as unidades funcionais, o que obrigava o uso de NO-OPs como está ilustrado na Figura 2.1(b). As máquinas VLIW mais modernas, utilizam um *bit* de final de grupo para marcar um grupo de instruções que formam um conjunto, como é mostrado na Figura 2.1(c) (TANENBAUM, 2007).

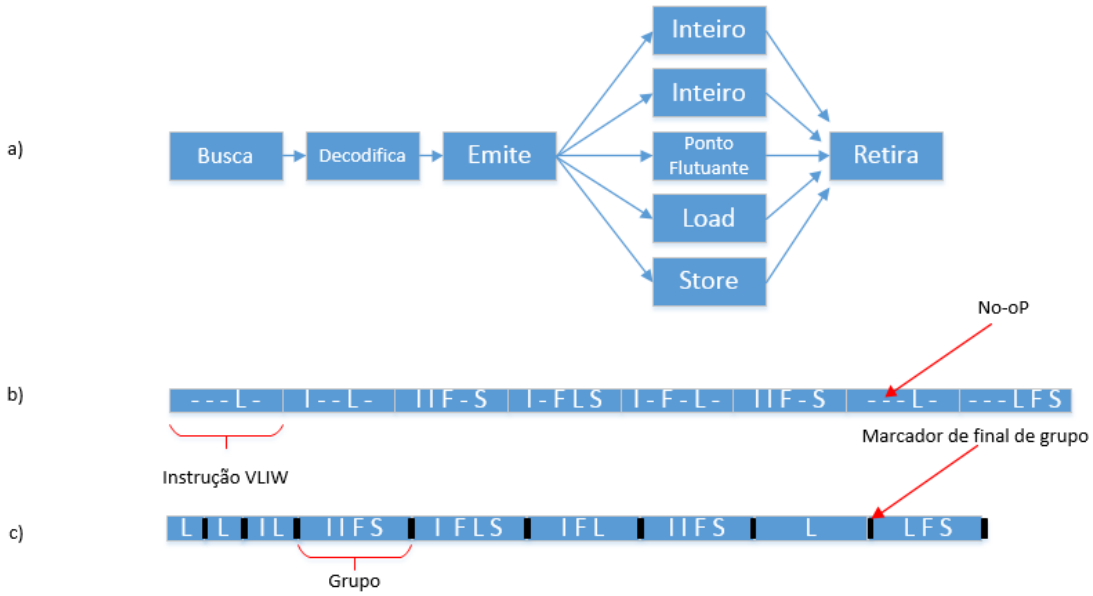

Figura 2.1: (a) *Pipeline*. (b) Sequência de instruções VLIW. (c) Corrente de instruções com pacotes marcados.

Adaptado de Tanenbaum (2007).

O processador irá buscar o grupo e executá-lo de uma só vez; o compilador, então, ficará

com o trabalho de escalonar as operações nos grupos corretamente. Assim, a arquitetura passa a responsabilidade de determinar quais operações devem ser executadas em paralelo no *hardware* para o compilador como é mostrado na Figura 2.2. Essa alteração deixa o *hardware* mais rápido e simplificado (TANENBAUM, 2007).

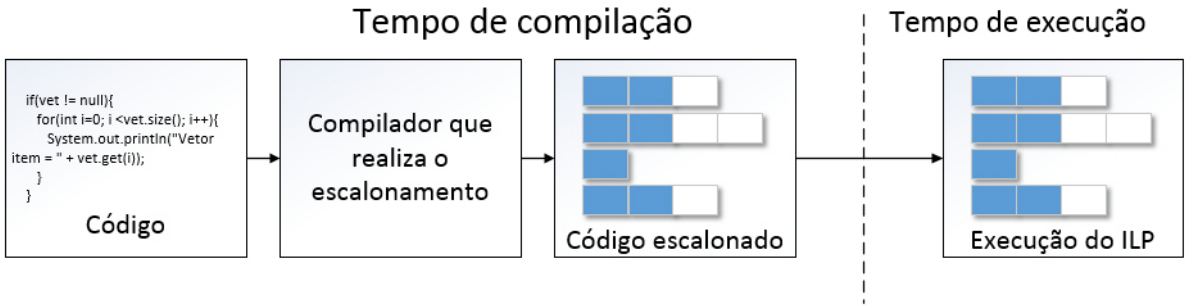

Figura 2.2: Sequência da execução de um código em uma arquitetura VLIW.
Adaptado de Fisher, Faraboschi e Young (2005).

A arquitetura VLIW proporciona as vantagens de precisar de menos *hardware*, pouca energia, um *design* menor em custo e tempo, sendo mais facilmente mutável (FISHER; FARABOSCHI; YOUNG, 2005).

A implementação do VLIW não tem *hardware* que detecta e previne *hazards*. Algumas implementações da arquitetura VLIW podem optar por esconder alguns detalhes que são mais bem tratados por *hardware*; nem tudo é exposto para o compilador, como alguns sinais de controle ou controles de interrupção (FISHER; FARABOSCHI; YOUNG, 2005).

O *datapath* de uma arquitetura VLIW pode ser definido como uma coleção de unidades de execução. A Figura 2.3 mostra conexões básicas de um VLIW com 8 unidades funcionais, onde cada ULA é conectada a um conjunto de registradores. Um compilador decompõe a computação de um programa em operações básicas e, então mapeia essas operações para as unidades correspondentes computacionais. O caminho de dados é organizado em linhas (uma por cada unidade), que aproximadamente correspondem à emissão das instruções VLIW. Cada unidade utiliza uma linha e um conjunto de portas nos arquivos de registradores. Neste exemplo, a memória possui quatro portas. Muitos detalhes do *datapath* real são omitidos (como os caminhos de controle). Como pode ser visto, o compilador possui uma grande participação na arquitetura VLIW e, inclusive, não é recomendado utilizá-la sem um bom compilador (FISHER; FARABOSCHI; YOUNG, 2005). A arquitetura VLIW é a mais apropriada para implementar a técnica do *loop pipelining*, e por isso a maioria das técnicas de *loop pipelinig* foram desenvolvidas para VLIW.

Arquiteturas VLIW convencionais, executando *loop* em *loop pipelining* dependem de um

arquivo de registradores muito grande para alcançar o desempenho esperado. Assim, arquiteturas alternativas foram propostas, como a de Fernandes (1998). A execução do *loop pipelining* em FPGAs reduz em muito essa necessidade, pois um número grande de valores intermediários são armazenados em registradores temporários, e não em um arquivo de registradores centralizado.

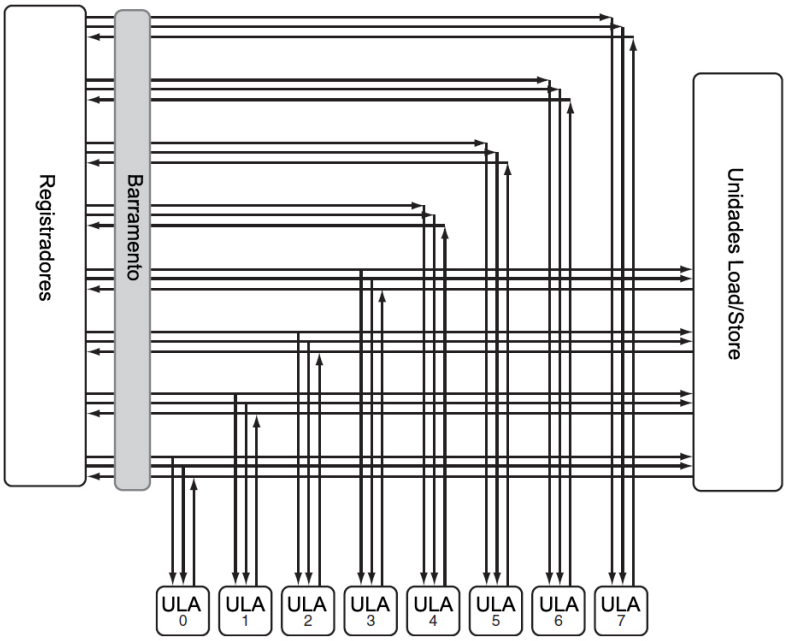

Figura 2.3: VLIW *datapath*.
Adaptado de Fisher, Faraboschi e Young (2005).

2.2 *Loop pipelining*

Cada vez mais se busca melhorar o desempenho na execução dos programas, e uma forma de se conseguir isso é através do ILP, onde pode ser utilizado um compilador para paralelizar o código. Como em muitas situações, os *loops* consomem a maior parte do tempo total de execução, grande parte dos esforços de otimização são concentrados em diminuir o tempo de execução de cada interação. O *software pipelining* ou *loop pipelining* é uma técnica que melhora o tempo de execução de um *loop*, criando uma taxa de execução mais rápida, onde as iterações são executadas de forma sobreposta aumentando o paralelismo (ALLAN et al., 1995).

A ideia por trás do *software pipelining*, é que o corpo de um *loop* seja reordenado de modo que uma iteração do *loop* possa começar antes das iterações anteriores serem concluídas (JONES; ALLAN, 1990).

Um exemplo do funcionamento do *software pipelining* é mostrado na Figura 2.4, onde um

bloco de operações é paralelizado supondo que cada operação é executada em um ciclo de *clock*.

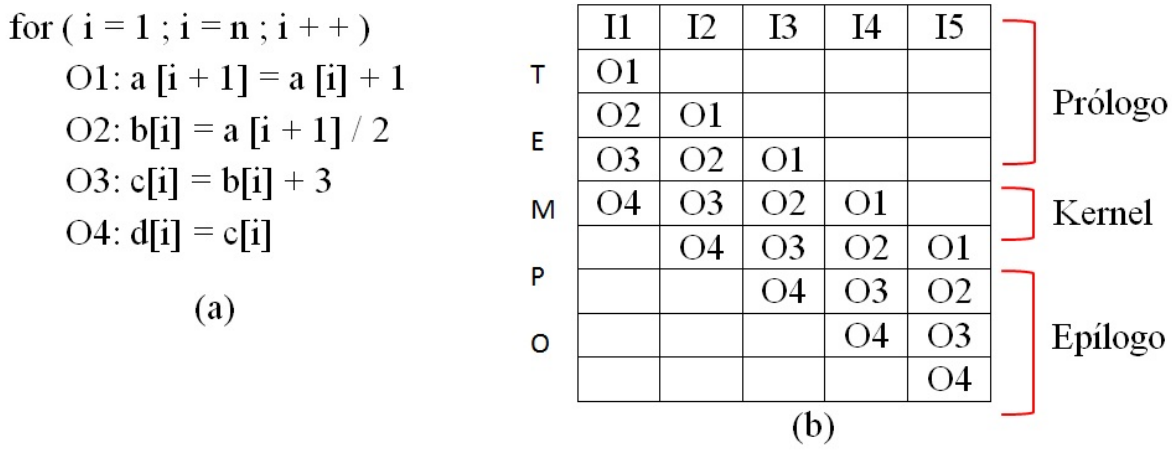

Figura 2.4: (a) Bloco de operações. (b) Escalonamento da execução de iterações supondo que o valor de n seja 5.

Fonte:Allan et al. (1995).

Na Figura 2.4 o bloco de operações (a) é reordenado, de modo que as operações executem simultaneamente (b). A determinação da ordem de disparo de instruções, ciclo a ciclo é chamada de escalonamento. O escalonamento pode ser visto então, como uma matriz de um conjunto de operações, onde as colunas representam as iterações e as linhas o tempo (Figura 2.4(b)). A paralelização, então, será feita no *kernel* que é o novo corpo do *loop*. No *kernel*, as operações serão executadas em paralelo, ocupando as unidades de processamento disponíveis. As operações do *loop* que iniciam o processo de paralelismo, antes da formação do *kernel*, são denominadas prólogo e o epílogo são as que sucedem o *kernel*.

A vantagem do *software pipelining* é que serão executadas várias instruções de várias iterações diferentes simultaneamente, ao invés de uma a uma de modo sequencial, acelerando a execução do programa. Sem o *software pipelining* a tabela da Figura 2.4(b) seria apenas uma coluna com todas as instruções, uma após a outra.

Para decidir quais operações podem ser executadas em conjunto, é importante saber qual a ordem que as operações devem seguir, pois podem ocorrer conflitos. Dependências entre as operações podem restringir o que pode ser feito em paralelo. O grafo de dependência de dados, ou DDG, é utilizado para ilustrar o relacionamento entre as operações (ALLAN et al., 1995). O gráfico de dependência de dados pode ser representado por $DDG(N,A)$, onde:

N: representa os nós (operações).

A: representa as arestas (dependências).

No grafo DDG existem dois tipos de arestas, as denominadas *loop-independent* que mos-

tram dependências dentro de uma mesma iteração, definindo qual ordem as operações devem seguir nessa iteração. Também existem as arestas *loop-carried* que mostram a dependência que as operações de diferentes iterações possuem entre si.

Cada aresta do grafo terá um par de valores (dif, min), o valor dif está relacionado com o *loop-carried* e indica o número de espaços entre iterações diferentes, e o valor *min* indica a espera de uma instrução para a mesma iteração. Um exemplo simples de um grafo DDG é mostrado na Figura 2.5.

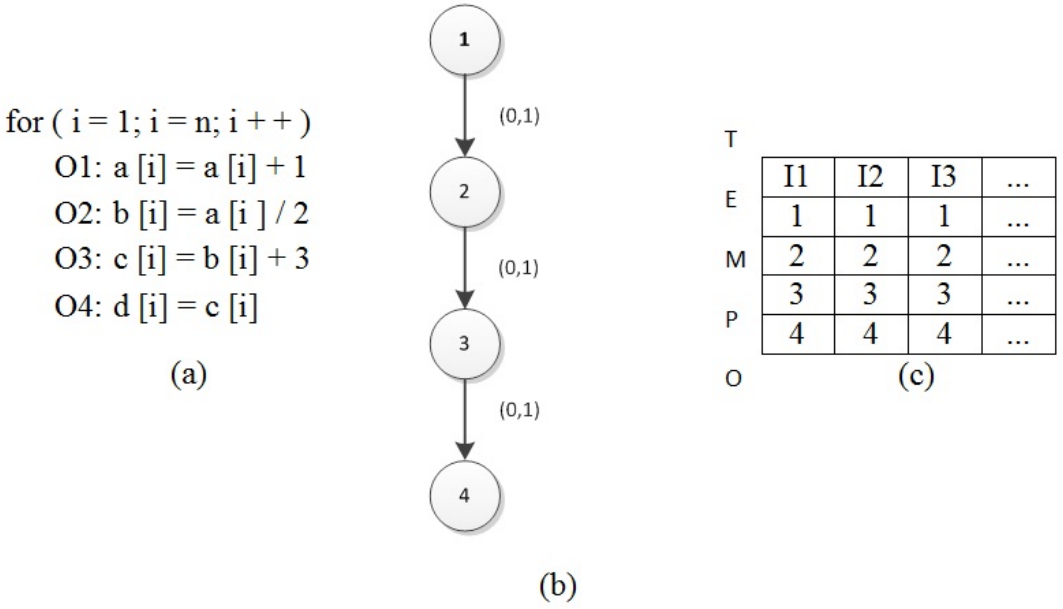

Figura 2.5: (a) Bloco de operações. (b) DDG. (c) Escalonamento.
Fonte: Allan et al. (1995).

No exemplo da Figura 2.5 não existem dependências entre operações de diferentes iterações, assim o grafo não possui nenhuma aresta mostrando dependências *loop-carried*, somente *loop-independent*, e por isso não existe nenhuma aresta com valor diferente de 0 no campo dif. Esse tipo de *loop* é muito fácil de paralelizar, pois todas as instruções de diferentes iterações podem ser executadas simultaneamente como é mostrado o escalonamento na Figura 2.5(c), onde todas as operações 1 podem ser executadas juntas, depois as operações 2 e assim por diante, isso dependendo, claro, do número de unidades funcionais disponíveis(ALLAN et al., 1995).

Nem todos os *loops* são fáceis de paralelizar como na Figura 2.5, outro exemplo mais complexo é mostrado na Figura 2.6.

No caso, a operação $O1$ da primeira iteração deve vir antes da operação $O1$ da próxima iteração, devido à instrução $a[i]$, que é calculada na iteração anterior, como é mostrado na Figura 2.6(c); na segunda iteração, onde é calculado $a[3]$, o valor $a[2]$ utilizado em $a[3]$ foi calculado

anteriormente na primeira iteração, e por isso deve ser preservado.

O bloco da Figura 2.6 possui uma dependência *loop-carried* na operação $O1$ com diferença de 1 (em número de iterações) entre a operação de origem e a operação alvo, que é nada mais que o primeiro valor de dif do par de associações (dif, min). Já para as dependências *loop-independent* são associados valores de atraso nas arestas. Esse valor de atraso é o *min* do par de associações (dif, min). O valor indica que um número especificado de ciclos deve ocorrer entre as operações incidentes. Esse atraso é usado para especificar que certas operações são

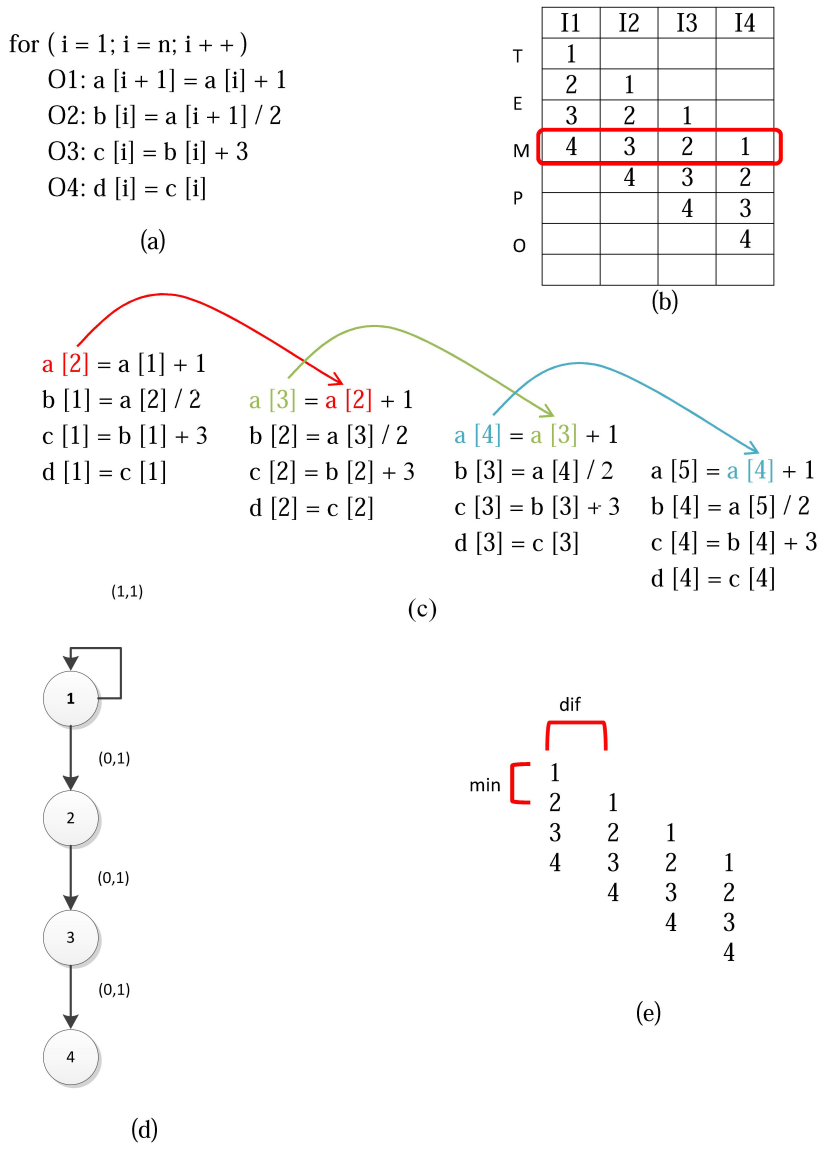

Figura 2.6: (a) Bloco de operações. (b) Escalonamento. (c) Escalonamento mostrando as dependências entre as iterações. (d) grafo DDG. (e) Relação entre o par associações (dif, min) e o código escalonado.

Fonte: Allan et al. (1995).

multiciclo, como uma multiplicação de pontos flutuantes. Assim, para uma aresta é denotado um tempo *min* que é o tempo em que a segunda operação deve esperar para ser executada depois da primeira operação (ALLAN et al., 1995). Este processo é resumido pela Figura 2.6(e).

Um exemplo com um valor diferente de dif é mostrado na Figura 2.7. Aqui ocorre exatamente a mesma situação em todo o bloco de execução, porém, na operação $O1$ o valor que cria a dependência em iterações diferentes, que é o 1 da instrução $a[i+1]$, é substituído por 2, ficando $a[i+2]$, fazendo com que o valor de dif seja agora 2.

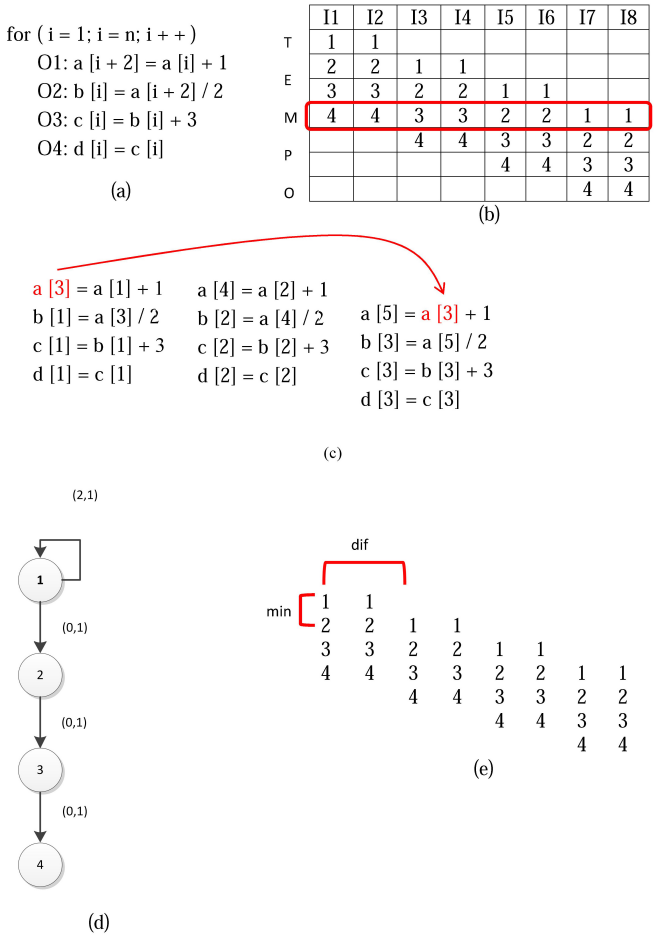

Figura 2.7: (a) Bloco de operações. (b) Escalonamento. (c) Escalonamento mostrando as dependências entre as iterações. (d) grafo DDG. (e) Relação entre o par associações (dif, min) e o código escalonado.

Fonte: Allan et al. (1995).

2.2.1 *Modulo Scheduling*

Para realizar o escalonamento das instruções existem vários algoritmos, dentre eles um muito conhecido, é o *modulo scheduling* ou escalonamento de módulo (RAU, 1994). Para implementar esse algoritmo, é necessária a obtenção de informações das dependências de dados, as distâncias entre instruções e o intervalo de iniciação.

O intervalo de iniciação (II) é o tempo para iniciar uma nova iteração do *loop*, com o propósito de respeitar as dependências entre iterações. Portanto, pode ser definido como um intervalo constante utilizado para iniciar sucessivas iterações. Quanto maior o intervalo de iniciação, maior será o tempo gasto para execução de todas as iterações do *loop*. Pode-se perceber que o intervalo de iniciação é um elemento chave para o paralelismo do *software pipelining*. Resumindo, o intervalo de iniciação determina o tamanho do *kernel*, que é a quantidade de ciclos que o *kernel* possui. Na Figura 2.8 o *kernel* possui o tamanho de 4 ciclos, que é determinado pelo II.

Figura 2.8: *Kernel* com 4 ciclos.
Fonte: Allan et al. (1995). fig21

No algoritmo *modulo scheduling*, inicialmente deve-se prever um valor para o tamanho do *kernel* e realizar adequações para melhorá-lo, gerando um novo *kernel*. O objetivo desse algoritmo é gerar um *kernel* no menor número de ciclos possível, assim, o *modulo scheduling*, procura um *kernel* escalonável com base nas restrições de recursos e de dependências, e busca melhorá-lo reduzindo o intervalo de iniciação.

Como prever o II é um problema muito complexo, é preciso então dar um chute inicial para o valor de II, e incrementar esse valor até chegar o II definitivo. Para esse valor inicial é dado o nome de mínimo intervalo de inicialização (MII), que é o menor valor possível de II.

Um modo de calcular o mínimo intervalo de iniciação é através de uma análise das dependências do grafo DDG. Um valor utilizado nessa análise é o total de necessidades de recursos das operações no *loop* em relação à quantidade de recursos, esse valor é denominado ResMII (*Resource Minimum Initiation Interval*). O valor ResMII representa as restrições causadas pelos recursos. Por exemplo, existem seis operações de adição diferentes e apenas duas

unidades de adição, então o valor de ResMII pode ser obtido pelo cálculo: $ResMII = 6 \div 2 = 3$. Outro valor utilizado é a análise de dependência de dados denominado RecMII (*Recurrence Minimum Initiation Interval*), que é determinado pelas dependências *loop-carried*. O valor de MII é obtido por:

$$MII = Max(ResMII, RecMII)$$

O algoritmo de *modulo scheduling* possui algumas variações e uma das principais é o *modulo scheduling* iterativo proposto por Rau (1994). A entrada do algoritmo é um grafo DDG de um *loop*, e a saída do algoritmo será um *kernel* com $II \geq MII$. O algoritmo cria um novo corpo para o *loop* em forma de um *kernel*, para que as instruções possam ser executadas em paralelo. O algoritmo, basicamente inicia II com o valor de MII e vai incrementando II conforme for realizando o escalonamento.

O primeiro passo do algoritmo é inicializar II com o valor de MII, onde II será incrementado durante a execução do algoritmo até um escalonamento válido ser encontrado, ou o tamanho de II aumentar muito e o escalonamento ficar inviável. O próximo passo é retornar a operação com maior prioridade através da lista de operações não escalonadas. O processo de retornar as operações não escalonadas é representado pela função ProximoNodo() da Listagem 2.1. Após adquirir a operação, a mesma é escalonada, ou seja, é encontrado um ciclo para a operação no *kernel* através de uma análise de recursos e dependência de dados.

O escalonamento é representado pela função Escalonar() da Listagem 2.1. Se o escalonamento falha, então a operação é removida do escalonamento e volta para a lista de operações não escalonadas, o II é incrementado e a função de escalonamento é chamada novamente. A reinserção da operação para a lista de operações não escalonadas é representada pela função Reinserir(). O escalonamento termina quando a lista de operações não escalonadas está vazia, ou o número de tentativas de escalonamento alcança um valor maior que o número máximo de tentativas (RAU, 1994). A Listagem 2.1 mostra o algoritmo de uma forma simplificada.

Listagem 2.1: *Modulo scheduling*.

```
1 bool ModuloScheduling() {
2         int tentativas = 0;
3         int II = MII();
4
5         do {
6                 Nodo n = ProximoNodo();
7                 if (!Escalonar(n)){
8                         Reinserir(n);
```

```
9                       Tentativas++;
10                      II++;
11                  }
12          } while (!ListaVazia() || MaximoTentativas())
13
14          return ListaVazia();
15  }
```

Sobreposição de Valores

Um dos maiores problemas do *software pipelining* é a sobreposição de valores. Os requisitos de um registrador são determinados pelo ciclo de vida das variáveis. O valor de uma variável deve existir ou continuar vivo até ser utilizado pela última vez. O ciclo de vida de uma variável que possui algum tipo de dependência, é medido pelo tempo entre o momento em que o valor da variável é colocado na operação escalonada e a última vez em que a operação utiliza aquele valor. O escalonamento é problemático quando um valor tem seu ciclo de vida sobreposto. Esse problema ocorre quando o comprimento do ciclo de vida é maior que o II. Quando um valor tem seu ciclo de vida sobreposto, a operação em uma iteração sucessiva com execução em paralelo com a iteração corrente irá substituir o valor antes da última utilização do valor na iteração corrente, resultando em um valor errado (STOTZER; LEISS, 2009). Isso ocorrerá caso se use alocação de registradores convencionais. Uma forma de resolver isso sem perder a eficiência, é construir um arquivo de registradores que permita o armazenamento de valores sucessivos, de acordo com as necessidades do escalonamento da estratégia de *software pipelining*. Este é o problema que se pretende tratar neste trabalho. Na Figura 2.9 é ilustrado o escalonamento, a relação entre registradores e o escalonamento, e o principal problema que ocorre com o *software pipelining*. Ao lado da Figura 2.9(a) é mostrada a quantidade de unidades funcionais para *add*, mul e *load*/*store*, e também a latência para as dependências do tipo *loop-independent*. Uma vez que o grafo da Figura 2.9(a) não possui dependências do tipo *loop-carried*; o que determina o valor de MII é o valor de ResMII. O que limita o valor de ResMII é a multiplicação, pois existe apenas uma unidade para quatro operações de multiplicação. Assim, $ResII = 4 \div 1 = 4$ e $RecII = 0$, então $MII = Max(ResII, RecII) = Max(4,1) = 4$. Assim, o tamanho do *kernel* será de 4 ciclos. Para ser feito o escalonamento, a prioridade de cada operação é calculada em relação à profundidade de cada nó, os quais serão escalonados na seguinte ordem: (N1, N2, N5, N8, N9, N3, N10, N6, N4, N11, N12, N7) (ZALAMEA et al., 2004).

A Figura 2.9(b1) mostra o escalonamento para uma iteração, as cores destacam os estágios para montar o *kernel* e podem ser vistas também como diferentes iterações. Os números repre-

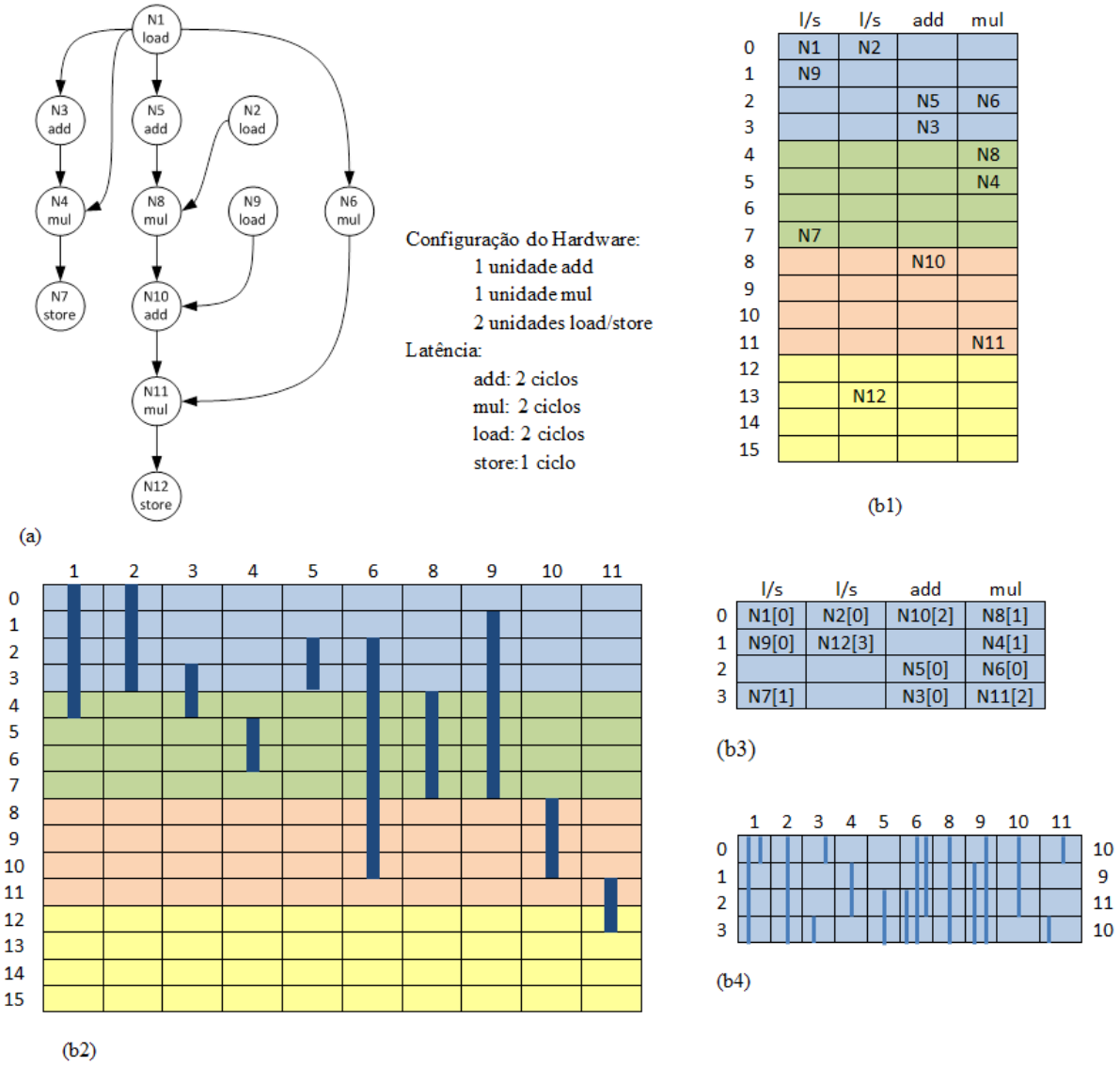

Figura 2.9: (a) Grafo de dependência de dados. (b) Escalonamento e ciclo de vida das variáveis. Fonte: Zalamea et al. (2004).

sentam o tempo em ciclos em toda a Figura 2.9. Apesar da Figura 2.9(b1) mostrar apenas o escalonamento para uma iteração a mudança de cor significa o início de uma nova iteração. O escalonamento de outras iterações é apenas omitido, mas essas iterações existem e podem ser vistas na Figura 2.10, que é uma representação completa do escalonamento da 2.9(b1). Então espaços do *kernel* que foram preenchidos com uma operação em uma cor, não podem ser preenchidos em outra cor, pois os espaços já foram ocupados. A Figura 2.9(b3) mostra o *kernel* montado, os números mostram a qual estágio da montagem do *kernel* cada operação pertence. A Figura 2.9(b2) mostra o ciclo de vida de cada variável (o tempo em ciclos em que o valor foi produzido e depois consumido) em relação ao tamanho do *kernel*. Os números em cima da Figura 2.9(b2) são os nós do grafo da Figura 2.9(a). Na Figura 2.9(b2) as cores também mostram o início de uma nova iteração, ou uma execução do *kernel*, e estão relacionadas com as cores

na Figura 2.9(b1). Um exemplo seria o valor da operação N9, que é produzido na cor azul no ciclo 1 e consumido por N10 na cor laranja no ciclo 9, que é representado pela Figura 2.9(b2). A Figura 2.9(b4) mostra os requisitos que o registrador precisa ter para este escalonamento, e pode ser vista como a Figura 2.9(b2) resumida com a mesma quantidade de ciclos que o *kernel*. A figura mostra os valores vivos simultaneamente, que são necessários em uma iteração. Os valores vivos são mostrados pelos números à direita da Figura 2.9(b4), por exemplo, no ciclo 2 de cada execução do *kernel* devem existir 11 valores vivos simultaneamente.

O número de registradores necessários pode ser aproximado pelo número máximo de valores simultaneamente vivos, em qualquer ciclo, que é chamado de *MaxLive* na Figura 2.9(b4) o valor de *MaxLive* é 11. No exemplo da Figura 2.9 pode ocorrer uma sobreposição de valores, se não existir um tipo de registrador que suporte o acúmulo de valores. O valor de N6 pode ser utilizado como exemplo para ilustrar a sobreposição de valores, onde seu ciclo de vida dura três execuções do *kernel*, como pode ser visto na Figura 2.9 (b2) e na Figura 2.9 (b4). A Figura 2.10 é semelhante à Figura 2.9, porém mostra a montagem do *kernel* e o problema de sobreposição com mais detalhes. Na Figura 2.10 foram adicionadas as operações de várias iterações durante o processo de paralelização do grafo da Figura 2.9(a). Os números na frente das operações, da figura indicam em qual iteração cada operação está em determinado momento. O valor de N6 é produzido três vezes antes de ser consumido pela primeira vez por N11. Assim, os valores de N6(1), N6(2) e N6(3) permanecem vivos simultaneamente, e se esses valores não forem armazenados corretamente, N11(1) pode consumir o valor de N6(3) ao invés de N6(1), e os valores N6(1) e N6(2) podem ser perdidos.

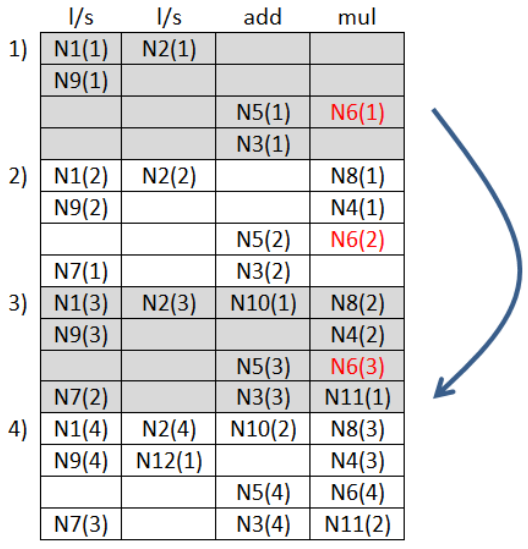

Figura 2.10: Problema de sobreposição de valores no *software pipelining*.

Geração de prólogo e epilógo

Na técnica de *loop pipelining*, para que o *kernel* seja executado são necessários os estágios de prólogo e epílogo, estes que, não deixam de ser uma repetição das instruções executadas no próprio *kernel*. O prólogo e o epílogo na verdade são versões incompletas do *kernel*. No início do estágio de prólogo são executadas poucas instruções simultaneamente, e à medida que são executadas as iterações o número de instruções cresce gradativamente. Quando todas as instruções são executas simultaneamente, a execução do programa chega ao estágio de *kernel*. Após a execução do estágio de *kernel*, o número de instruções simultâneas começa a cair e a execução do programa chega ao estágio de epílogo. A geração do prólogo e epílogo é mais trabalhosa para o compilador do que a geração do próprio *kernel* em si, pois exige que o compilador gere várias sequencias de instruções ao contrário do *kernel* que é apenas uma sequencia repetida várias vezes. Esse pensamento também se aplica ao transformar o *loop pipelining* em *hardware*. Seria muito mais interessante se a saída do compilador fosse apenas as instruções que compõe o *kernel*. Um exemplo é mostrado na Figura 2.11, onde quatro instruções são escalonas em relação ao tempo.

Na figura, os campos em vermelho representam as instruções que devem ser executadas e os campos em marrom as instruções que não devem ser executadas. Analisando a figura pode ser observado que nos estágios de prólogo e epílogo as instruções existem, mas apenas não estão sendo executadas naquele momento, dando uma ideia de que as instruções estão sendo ativadas ou desativadas. Uma forma de colocar essa ideia em prática é utilizar o controle de instruções predicadas, que é uma técnica que pode ser utilizada junto ao *loop pipelining* com o objetivo de deixar o escalonamento mais limpo e fácil de ser gerado pelo compilador, evitando essa replicação de código.

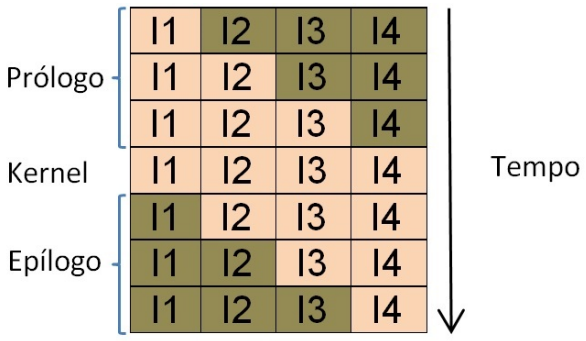

Figura 2.11: Exemplo que ilustra a ativação de instruções durante o escalonamento.

2.3 Arquivos de Registradores para Suporte ao *Loop Pipelining* / VLIW

O *modulo scheduling* proposto por (RAU, 1994) implementa a técnica do *software pipeline*, porém existe o problema de sobreposição de valores, como foi mostrado na seção 2.2, exigindo que se utilize esquemas alternativos para a alocação de registradores e construção de arquivos de registradores. Vários estudos foram feitos para acabar com esse problema de sobreposição de valores, e alguns abordam o problema utilizando soluções em *software* e outros em *hardware*.

Um dos estudos que busca solucionar esse problema é o módulo de expansão variável ou MVE (*Modulo Variable Expansion*) proposto por Lam (1988). A ideia do MVE pode ser ilustrada pela Figura 2.12, onde um valor é escrito em um registrador e pode ser utilizado ciclos mais tarde. O MVE identifica as variáveis que serão redefinidas no início de cada iteração do *loop*, e expande a variável em uma variável de dimensão mais elevada, de modo que cada iteração possa se referir a um local diferente. Consequentemente, a utilização da variável em diferentes iterações é independente. Para isso, serão utilizados vários registradores para cada variável do *loop* que sofrer sobreposição de valores. Para as variáveis que não sofrerem sobreposição será utilizado o mesmo registrador.

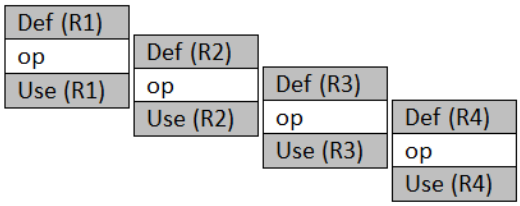

Figura 2.12: MVE (*Modulo Variable Expansion*).
Adaptado de Lam (1988).

O MVE consiste basicamente em descobrir quantos registradores são necessários por variável. No algoritmo primeiro, são identificadas as variáveis que serão redefinidas no início de cada iteração. Em seguida, supõe-se que cada iteração do *loop* tem uma localização dedicada no registrador para cada variável e são removidas todas as restrições de precedência entre as iterações, ou seja, o escalonamento é feito sem preocupações com a sobreposição de valores. O escalonamento resultante é, então, usado para determinar o número real dos registradores que devem ser atribuídos a cada variável. O tempo de vida das variáveis será utilizado no cálculo do número de registradores. O tempo de vida de uma variável em um registrador é definido pela duração entre a primeira atribuição da variável e a última utilização. Se o tempo de vida de uma variável é l, e uma iteração é iniciada cada S ciclos, pelo menos $l \div S$ número de valores devem

ser mantidos vivos ao mesmo tempo (LAM, 1988).

A vantagem dessa técnica é que não necessita de *hardware* especializado, mas para que uma variável referencie diferentes registradores em diferentes iterações, a mesma será replicada várias vezes aumentando o tamanho do código. Outra desvantagem é que a técnica precisa de muitos registradores para executar esse processo.

Outra técnica para suprir esse problema, envolve o uso de arquivos de registradores de rotação ou RFF (*Rotating Register File*) que foi proposta por Rau et al. (1992), e já foi utilizada pelo processador *Cydra* 5 e *Intel Itanium* (STOTZER; LEISS, 2009). O RRF é uma solução para o problema de sobreposição nos registradores que é implementado em *hardware* diferente do MVE que é implementado em *software*.

Essa solução consiste basicamente de um *hardware* gerenciador de registradores, onde o registrador é renomeado dinamicamente para cada instância da variável no *loop*, o que resolve o problema de aumento de código do MVE. É adicionado um nível a mais na especificação do registrador para receber a contagem da iteração do *loop* para fazer essa renomeação. O RRF cria um endereço indireto para o registrador. O endereçamento dos registradores possui uma base ou RRB (*Rotating Register Base*), e na instrução é especificado o endereço base junto com um deslocamento derivado da iteração do *loop*. Assim, a base e o deslocamento são combinados para exibir o verdadeiro registrador. A operação renomeia um registrador de $R[i]$ para $R[(i+1)\%n]$ onde i é o índice original do registrador e n é o número de registradores no arquivo de registradores (HUCK et al., 2000). A Figura 2.13 ilustra o RRF.

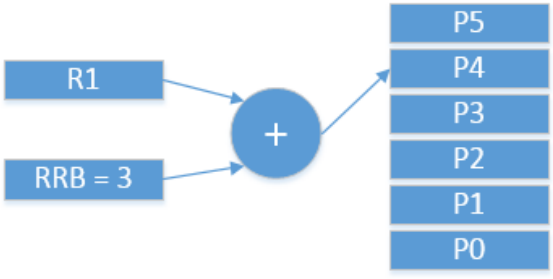

Figura 2.13: Exemplo básico de um RRF onde a base (RRB) é adicionada ao número do registrador atual endereçando o local físico.

Isso torna possível acessar o registrador que foi definido n iterações atrás. Porém, para realizar a renomeação dos registradores o RRF precisa de um grande número de registradores para ser eficiente.

Outro método que também utiliza *hardware* para solucionar o problema de sobreposição do *software pipelining* foi proposto por Fernandes (1998). Esse método utiliza arquivos de

registradores implementados em filas denominados QRF (*Queued Register File*), onde dados podem ser escritos e lidos uma vez por ciclo. O QRF irá ter o mesmo funcionamento de uma fila, quando um valor é escrito no registrador, o mesmo é inserido na cauda da fila e cada leitura remove o valor lido da fila. Assim, o valor poderá ser lido apenas uma vez, e a leitura é feita na cabeça.

Esse método é mais simples do que os outros métodos propostos. Porém, a simplicidade do *hardware* impõe restrições na alocação de registradores, mas o QRF se mostra mais eficiente que outros métodos, em termos de economia na área do *chip* e tempo de acesso. A estrutura de um QRF consiste de uma matriz de armazenamento cercada por circuitos de apoio para selecionar a gravação e leitura da posição atual como é mostrado na Figura 2.14 (FERNANDES, 1998).

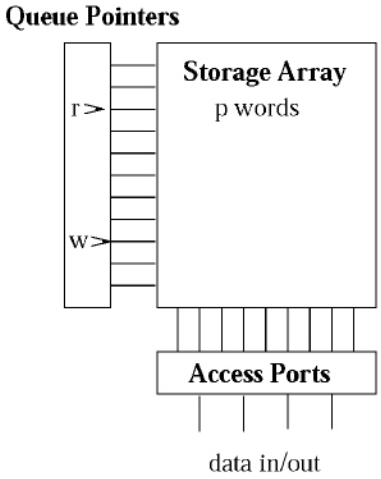

Figura 2.14: Diagrama de bloco de um QRF.
Fonte: Fernandes (1998).

Em um QRF o valor é alocado a uma fila específica, em vez de um registrador específico, sendo bem semelhante a um RRF. Outra vantagem do QRF, e também do RRF, é que a alocação de registradores pode ser realizada sem aumentar o código como no MVE.

De forma bem simples, a solução para o problema de sobreposição de valores da técnica do *software pipelining* utilizando QRF é ilustrada pela Figura 2.15, que aproveita o problema que foi mostrado na Figura 2.10.

Na Figura 2.15(a) é mostrado o DDG que é escalonado na Figura 2.15(b). Para suprir o problema de sobreposição é utilizado o QRF, onde seu funcionamento é mostrado pela Figura 2.15(c) utilizando como exemplo a variável N6. Assim, os valores de N6(1), N6(2), N6(3) são salvos na fila, e quando N6(1) precisa ser utilizado por N11(1) o valor é lido e retirado da fila.

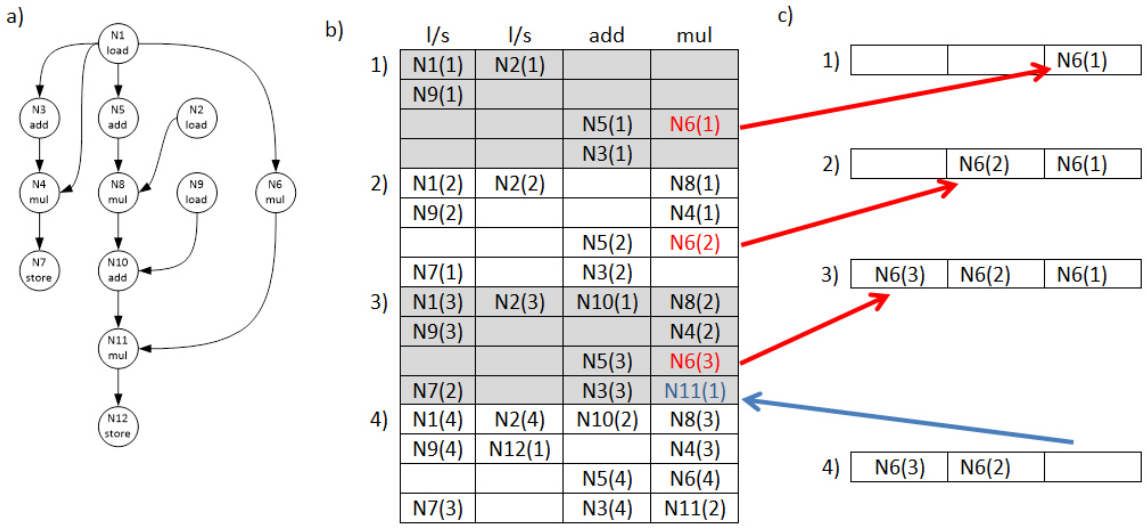

Figura 2.15: QRF em funcionamento.

2.4 Controle de instruções predicadas

As técnicas de *loop pipelining* podem ser facilmente utilizadas para realizar o paralelismo quando o comportamento do código é previsível em tempo de compilação. Já quando existem desvios condicionais no código e seu comportamento não pode ser previsto em tempo de compilação, as técnicas de compilação sozinhas podem não conseguir explorar o ILP de forma eficiente. Nesses casos, as dependências de controle podem afetar severamente o limite de paralelismo que pode ser explorado. Uma forma de solucionar esses problemas é utilizar a técnica de instruções predicadas ou instruções condicionais. Seu conceito é simples: Se a condição é verdadeira, a instrução é executada, mas se a condição é falsa, a execução continua como se a instrução fosse um *no-op*. Nesse caso, a instrução não escreve seu resultado no registrador de destino, ou no caso de uma operação que utiliza a memória não faria leitura e nem escrita. Algumas arquiteturas são totalmente predicadas, o que significa que grandes blocos de operações podem ser executados ou não dependendo da condição (HENNESSY; PATTERSON, 2011).

O conceito de instruções predicadas pode ser aplicado à técnica do *loop pipelining* de uma forma diferente. Da mesma forma em que as instruções predicadas podem controlar a execução ou não de uma instrução devido a um *if-else*, também podem controlar a execução de instruções do *kernel*, prólogo e epílogo. A ideia de utilizar as instruções predicadas junto ao *loop pipelining* é manter apenas as instruções do *kernel* eliminando o prólogo e epílogo, deixando o escalonamento mais limpo e fácil de ser executado em uma arquitetura ou em *hardware*. A Figura 2.16 mostra um exemplo de como as instruções predicadas podem ser utilizadas para controlar cada estado do *loop pipelining*, onde são executadas seis iterações do código. Nesse

exemplo para implementar as instruções predicadas foram utilizados registradores de deslocamento. O escalonamento do *loop pipelining* tem cinco estágios (Figura 2.16(a)), e cada estágio pode ter uma ou várias instruções. O exemplo é apenas ilustrativo e, portanto, o desenho mostra o escalonamento em um nível mais alto deixando de lado as instruções em si. Para ficar mais visível é bom relembrar o exemplo da Figura 2.9(b1) da Seção 2.2 que é semelhante à Figura 2.16(a), porém na Figura 2.9(b1) as instruções são exibidas dentro de cada estágio, diferente da Figura 2.16(a) onde as instruções foram omitidas. Os registradores $R1$ ao $R5$, representados pela Figura 2.16(b), fazem parte do registrador de deslocamento e cada registrador irá ativar um estágio do *loop pipelining*. Todas as instruções de um estágio i do escalonamento são ativadas pelo registrador Ri.

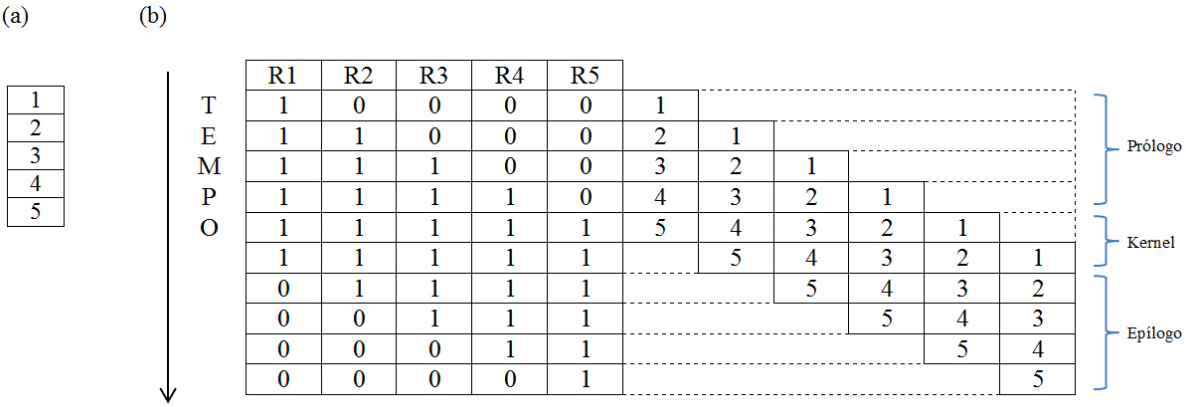

Figura 2.16: Exemplo de controle de predicados. (a) Escalonamento do *loop pipelining* com 5 estágios. (b) Instruções predicadas utilizadas para realizar o escalonamento.

Na Figura 2.16(b) é fácil visualizar como o *loop pipelining* pode ser implementado apenas com registradores de deslocamento. Os valores 0 ou 1 são armazenados dentro do registrador de deslocamento, e são responsáveis pela ativação dos estágios do escalonamento na medida em que o *loop* é executado. Quando o valor do registrador é 1 o estágio do escalonamento correspondente é ativado, e quando o valor é 0 o mesmo é desativado. Inicialmente, o registrador de deslocamento possui o valor 0 em todos os registradores. Antes de iniciar a execução do escalonamento o valor 1 é inserido no primeiro registrador R1. No primeiro ciclo da execução do*loop* o estágio 1 do escalonamento é executado, pois R1 possui o valor 1. Ainda no primeiro ciclo o valor 1 é deslocado para a esquerda, fazendo com que os registradores R1 e R2 fiquem com o valor 1. No segundo estágio, como os registradores R1 e R2 possuem o valor 1, os estágios 1 e 2 são executados e o valor 1 é deslocado novamente. Esse processo ocorre até que o prólogo todo seja executado. Quando a execução chega no *kernel* todos os registradores possuem o valor 1, o que significa que todas as instruções serão executadas. Na execução do *kernel* o registrador de deslocamento pode continuar deslocando os valores, como todos

os valores dos registradores são 1 a execução do *kernel* não será afetada pelo deslocamento. A execução do *kernel* e prólogo ocorrem em 6 ciclos, e no fim do sexto ciclo o valor 0 é inserido no início do registrador de deslocamento em $R1$, iniciando a execução do epílogo. Nos próximos ciclos, é o valor 0 que será deslocado para que os estágios do escalonamento comecem a ser desabilitados. Assim, no ciclo 7 o estágio 1 não será mais executado, pois R1 possui o valor 0, e no ciclo 8 os estágios 1 e 2 não serão executados, pois os registradores $R1$ e $R2$ irão possuir o valor 0. Isso irá continuar até que todo o epílogo seja executado, deixando o valor 0 em todos os registradores e terminando a execução do *loop*. Para que o registrador de deslocamento funcione conforme o especificado, deve existir um contador que mostra quando o valor 1 ou 0 deve ser inserido em $R1$. O valor 1 será inserido antes do inicio da execução do prólogo, e o valor 0 será inserido na ultima execução do *kernel*.

A vantagem de se fazer o escalonamento utilizando as instruções predicadas é que, o compilador deixa de gerar o código necessário para implementar o prólogo e o epílogo, o que facilita a tarefa do compilador, gerando menos código na forma de instruções ou estruturas de *hardware* replicadas, o que pode causar uma diminuição no uso de elementos lógicos e economia de energia quando implementado em FPGAs.

2.5 Computação Reconfigurável

Na computação, comumente são utilizadas duas maneiras diferentes de executar algoritmos: *hardware* e *software*. O *hardware* se sobressai em alguns aspectos, assim como o *software* se sobressai em outros (HAUCK; DEHON, 2008). Os computadores convencionais podem ser utilizados para várias aplicações diferentes através de diferentes tipos de *softwares*. A programação nos computadores convencionais é simples devido à existência de várias ferramentas e linguagens. A vantagem dessa abordagem é a flexibilidade, além do baixo custo. Realizar uma alteração no algoritmo é muito simples, especialmente se o código foi desenvolvido utilizando orientação a objetos. Porém, para aplicações específicas, os computadores convencionais não conseguem garantir o melhor desempenho (SKLIAROVA; FERRARI, 2003).

Se a execução de um programa excede as capacidades dos computadores de uso geral, são necessárias abordagens diferentes, como a computação paralela. Contudo, para o uso de computação paralela, a aplicação deve possuir uma estrutura compatível e muitas vezes não oferece o desempenho esperado(SKLIAROVA; FERRARI, 2003).

Uma abordagem para se obter maior desempenho consiste em projetar e construir circuitos integrados especificamente para executar uma determinada aplicação. Um exemplo é o cir-

cuito integrado de aplicação específica (ASIC - *Application Specific Integrated Circuit*) ou o circuito totalmente customizável (VLSI - Very Large Scale Integration) (HAMBLEN; HALL; FURMAN, 2008). Esses circuitos são projetados especificamente para solucionar um dado problema, portanto, eles são muito rápidos e eficientes ao executar o objetivo exato para o qual foram concebidos. Apesar de rápidos, exigem muito custo e tempo para desenvolvimento de um projeto. O desenvolvimento de um projeto com a tecnologia VLSI pode exigir vários anos de esforço. Tal esforço de desenvolvimento é muito caro, sendo utilizado apenas para grandes escalas. Essa abordagem pode gerar dispositivos de mais alto desempenho, como microprocessadores e *chips* de memória RAM usados em computadores. Os ASICs não são tão eficientes como os VLSIs, mas também possuem um alto custo em sua fabricação. Vários meses são necessários para a produção desses circuitos, sendo viáveis somente se fabricados em grandes quantidades. Qualquer erro de projeto no *chip* levará a atrasos na fabricação e custos adicionais (HAMBLEN; HALL; FURMAN, 2008). Essas soluções são completamente inflexíveis e sua funcionalidade não pode ser modificada após sua fabricação.

Resumindo, os ASICs são mais eficientes, mas estão configurados permanentemente para apenas uma aplicação através de um projeto de alto culto. O *software* proporciona a flexibilidade para alterar os aplicativos e realizar um grande número de tarefas diferentes, porém quando o quesito é desempenho, tamanho do *chip* e uso de energia os ASICs são superiores (HAUCK; DEHON, 2008).

A computação reconfigurável pretende preencher o espaço entre *hardware* e *software* alcançando maior desempenho que os computadores de uso geral, enquanto mantém maior flexibilidade que os ASICs (ver Figura 2.17) (COMPTON; HAUCK, 2002). A principal característica dos dispositivos reconfiguráveis (*reconfigware*) é a possibilidade de seu *hardware* ser modificado em seu ciclo de vida, através de dispositivos lógicos reprogramáveis. Arquiteturas reconfiguráveis são uma alternativa interessante para a exploração de paralelismo em nível de instruções, pois podem se adaptar com eficiência ao grau de paralelismo disponível na aplicação. Por esse motivo, é um dos focos deste trabalho.

O conceito de arquitetura adaptável foi introduzido na década de 60, mas somente com a comercialização dos primeiros dispositivos de lógica programável na década de 80, pela Xilinx (XILINX, 2014) e pela Altera (ALTERA, 2014a), é que o conceito começou a ganhar atenção. O mercado de computação reconfigurável ganhou força somente no final dos anos 90.

Existem vários tipos de dispositivos reconfiguráveis como o PAL (*Programmable Array Logic*), PLA (*programmable logic array*), MPGA (*Mask Programmable Gate Array*), GAL (*Generic Array Logic*), CPLD (*Complex Programmable Logic Device*) e FPGA (*Field Pro-*

Figura 2.17: Vantagens da computação reconfigurável.
Adaptado de Skliarova e Ferrari (2003).

grammable Gate Array). O FPGA é o dispositivo que possui maior flexibilidade podendo assim suportar sistemas mais complexos. A Figura 2.18 mostra uma relação entre FPGAs, PLDs (*Programmable Logic Devices*) que são dispositivos lógicos programáveis simples, ASICs e VLSIs. (HAMBLEN; HALL; FURMAN, 2008).

Os avanços recentes na computação reconfigurável, são na maior parte derivados das tecnologias desenvolvidas para FPGAs em meados dos anos 1980. Os FPGAs foram originalmente criados para servir como um dispositivo híbrido entre MPGAs e PALs. Os *chips* FPGAs são programáveis eletricamente como os PALs, o que significa que o *hardware* pode ser personalizado quase que instantaneamente e também podem implementar *hardwares* bastante complexos, iguais aos dispositivos MPGAs (COMPTON; HAUCK, 2002).

2.5.1 FPGA

Os circuitos de FPGAs são dispositivos que podem ser programados eletricamente para se tornarem quase qualquer tipo de sistema ou circuito digital. Eles oferecem uma série de vantagens atraentes sobre os ASICs ou VLSIs, pois são mais baratos e exigem menos tempo no

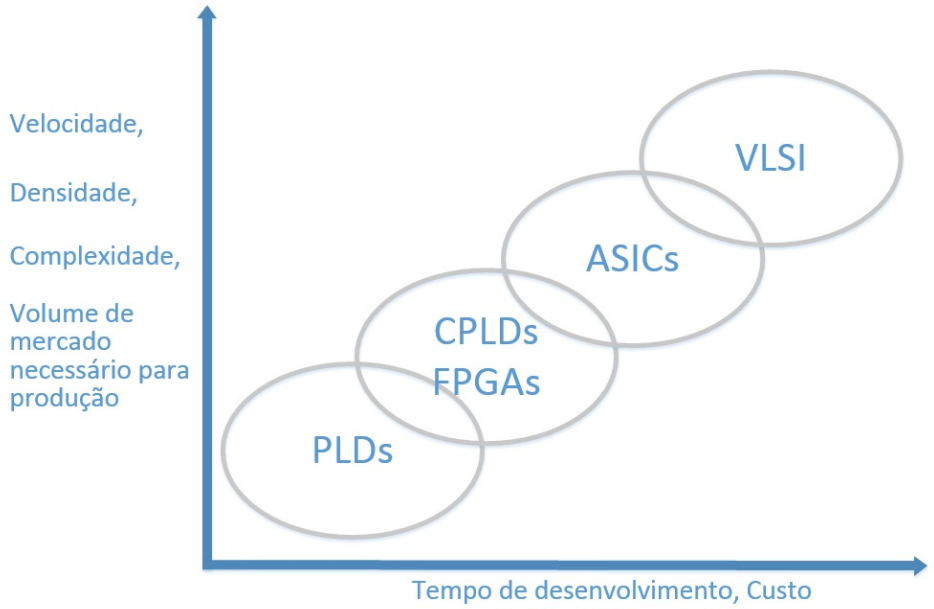

Figura 2.18: Relação entre os tipos de tecnologia digital.
Adaptado de Hamblen, Hall e Furman (2008).

projeto e criação de um circuito. Porém, a natureza flexível de um FPGA tem um custo significativo na velocidade, consumo de energia, e área. O FPGA comparado com o ASIC requer mais área, tem um desempenho em velocidade mais lento e consome mais energia (KUON; TESSIER; ROSE, 2008).

Um fato interessante é que um *chip* de FPGA, dependendo da aplicação, é capaz de superar um CPU ou GPU (*graphics processing units*) mesmo com uma frequência de operação menor. Em vários tipos de aplicações, especialmente as baseadas em ponto flutuante, o desempenho da GPU é ligeiramente melhor ou, muito próximo ao desempenho de um FPGA. Quando se trata de eficiência energética, no entanto, o FPGA é superior a CPUs e GPUs em diferentes tipos de aplicações (BACON; RABBAH; SHUKLA, 2013).

Uma característica chave de um dispositivo FPGA é a tecnologia programável utilizada para configurá-lo. Muitas tecnologias têm sido consideradas para uso em FPGAs, incluindo SRAM, EEPROM/EPROM e antifusíveis. O antifusível (antifuse) é um dispositivo programável eletricamente de dois terminais que assume o valor de alta para baixa resistência quando uma voltagem é aplicada através de seus terminais (GREENE; HAMDY; BEAL, 1993). Essa voltagem é que programa o dispositivo, e geralmente é aplicada através de um pulso de 16 Volts. O dispositivo quando programado é configurado como "queimado", funcionando como um circuito fechado. Quando não programado, o dispositivo funciona como um circuito aberto, desse modo, nenhuma corrente pode fluir entre seus terminais. A desvantagem desse tipo de tecnologia é que o mesmo pode ser programado apenas uma vez. É possível programar os FPGAs baseados em

EEPROM/EPROM eletricamente várias vezes; para apagar os dados de uma EPROM é utilizada luz UV, e de uma EEPROM é utilizada eletricidade. Uma vantagem, é que os dados são mantidos mesmo quando a energia é desligada, evitando-se assim a necessidade de reprogramar o *chip* toda vez que for ligado. No entanto, as altas tensões requeridas para programar o dispositivo dificulta a sua programação. Os FPGAs baseados em SRAM são voláteis e precisam ser reprogramados toda vez que forem iniciados, mas possuem diversas vantagens em relação a outros tipos, pois são facilmente reprogramáveis. Assim, sua configuração pode ser alterada no caso de uma correção de um erro, sendo assim um meio ideal para fazer protótipos (HAUCK, 1998). Portanto, neste trabalho foi utilizado um dispositivo de FPGA baseado em SRAM e seu funcionamento será explicado com mais detalhes.

Existem três tipos de componentes principais que compõe os FPGAs baseados em SRAM:

- CLB (*Configuration Logical Blocks*): são blocos lógicos configuráveis que são responsáveis por implementar as funções lógicas;
- IOB (*Input/Output Block*): Fazem a interface entre os blocos lógicos, funcionando como *buffers* de entrada e saída do FPGA;
- *Switch Matrix* (Matriz de interconexão programável): Representam a conexão entre os blocos lógicos ou entre os blocos lógicos e o IOB usando trilhas com conexões programáveis.

Os CLBs podem ser configurados de diferentes maneiras, uma delas é utilizando LUTs(*Look up Tables*) e *flip-flops*. A LUT é um componente que consegue implementar uma tabela verdade. Qualquer função booleana pode ser representada por uma tabela verdade, o que torna a LUT um dos componentes mais importantes de um FPGA. Uma LUT pode ser composta por um multiplexador com *N* entradas e uma saída e uma memória de *N bits*, que são responsáveis por enumerar a tabela verdade. Portanto, utilizando LUTs, o FPGA pode implementar vários componentes diferentes da lógica digital. Resumindo, uma LUT pode implementar qualquer função booleana de *N* entradas apenas programando sua memória com os valores da tabela verdade da função que se quer implementar (HAUCK; DEHON, 2008).

Com apenas a LUT o FPGA não é capaz de armazenar um estado, não podendo implementar nenhuma forma de lógica sequencial. Para completar o CLB, é adicionado junto a LUT um *flip-flop* D capaz de armazenar um bit (Figura 2.19). A saída do multiplexidador escolhe o resultado da função gerada pela LUT ou pelo bit armazenado no *flip-flop* D.

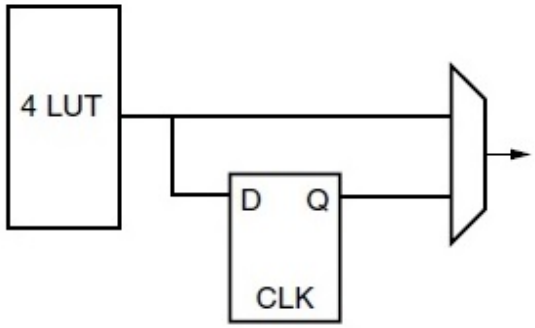

Figura 2.19: Esquema de um CLB composto por um *flip-flop* D e uma LUT.
Fonte: Hauck e Dehon (2008).

Um *chip* de FPGA é composto de vários CLBs, onde a matriz de interconexão tem o papel de conectar os vários CLBs de um FPGA uns com os outros, ou conectar a entrada e saída através da interface IOB com os CLBs. As chaves de interconexão permitem vários tipos de conexões diferentes. A Figura 2.20 mostra um esquema de um FPGA com CLBs, chaves de interconexão e IOBs.

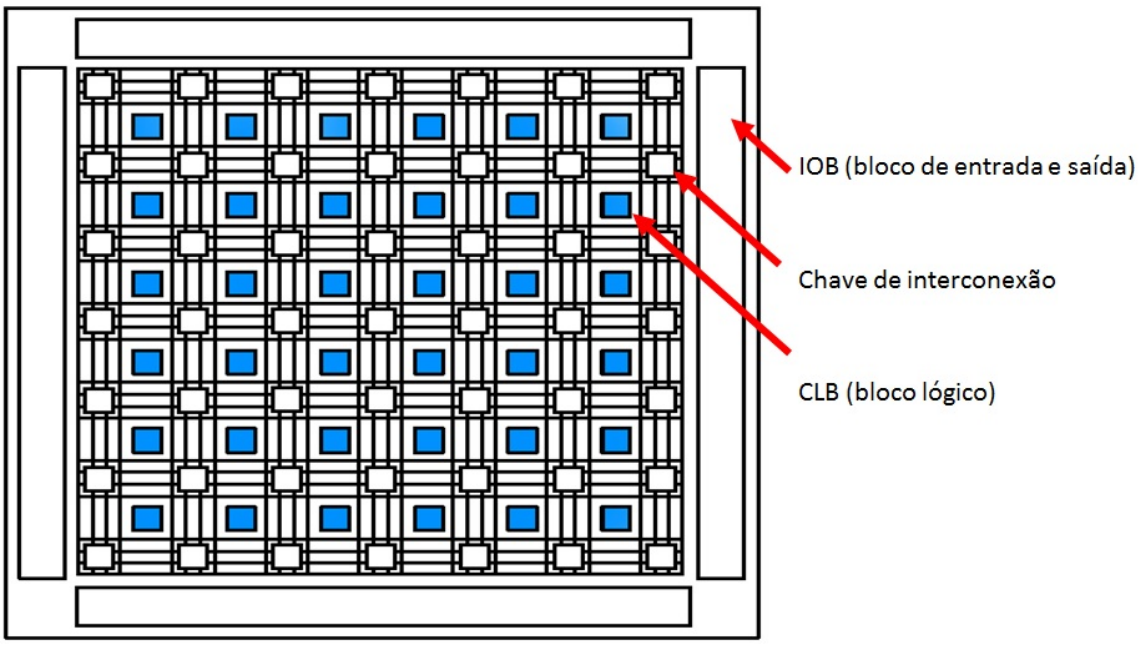

Figura 2.20: Estrutura básica de um FPGA.
Adaptado de Hauck (1998).

A configuração (programação) de um FPGA é, normalmente, feita utilizando-se uma linguagem de descrição de *hardware*, como VHDL ou Verilog. A descrição de um *hardware* possui uma sequência de desenvolvimento, onde sistema é descrito através de uma linguagem de descrição de *hardware* (VHDL ou Verilog). Essa linguagem, então, é transformada em um

esquema de elementos lógicos por um sintetizador; os elementos lógicos são transformados em um sistema que se adapta à estrutura lógica existente no FPGA, definindo a configuração dos blocos lógicos e suas conexões, levando-se em conta as propriedades físicas do dispositivo, e o passo final é gerar o arquivo com as informações para configurar o FPGA, chamado de *bitstream*.

Apesar das vantagens oferecidas pelo FPGA, existe uma grande dificuldade dos programadores de *software* em aprender a programar utilizando FPGA (BACON; RABBAH; SHUKLA, 2013).

2.6 Projeto de *Hardware* Digital

Qualquer projeto conta com uma sequência básica de tarefas que devem ser executadas. A Figura 2.21 mostra esta sequência. Apesar de ser uma sequência bem simples, a mesma pode ser utilizada para o desenvolvimento de um sistema em *hardware* (BROWN; VRANESIC, 2003). Este projeto de pesquisa utilizou uma forma mais elaborada do que neste exemplo, mas seguindo a mesma essência.

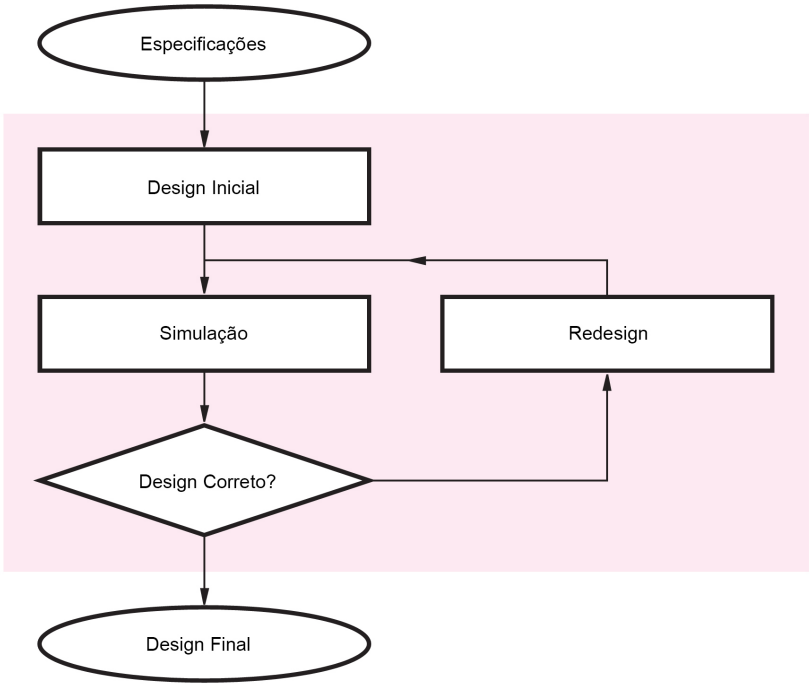

Figura 2.21: Sequência básica de um projeto de *hardware*.
Adaptado de Brown e Vranesic (2003).

Assumindo que se têm as especificações que definem o comportamento esperado e as características do circuito, o primeiro passo é o design inicial, no qual é criada uma versão inicial

ou protótipo do circuito. Este passo muitas vezes requer um grande esforço manual porque a maioria dos projetos têm alguns objetivos específicos que só podem ser alcançados através do conhecimento, habilidade e intuição do projetista. O próximo passo é a simulação do circuito em questão. Existem várias ferramentas para ajudar nesta etapa. No Capítulo 3, serão mostradas as ferramentas utilizadas para a simulação deste projeto de pesquisa. Aplicando condições de entrada, o projetista tenta verificar se o circuito projetado terá o desempenho exigido pelas especificações. Se na simulação forem encontrados erros, então o projeto deve ser alterado para que esses erros sejam resolvidos. Após as alterações serem efetuadas, o projeto é novamente simulado para determinar se os erros foram realmente corrigidos. Este ciclo é repetido várias vezes até que a simulação indique que o projeto não possui mais nenhum erro. O designer deve dar maior prioridade em corrigir os erros durante a simulação, pois erros são normalmente mais difíceis de corrigir se eles são descobertos no final do processo de design. Mesmo assim, alguns são difíceis de detectar durante a simulação, caso em que eles têm de ser tratados em fases posteriores do ciclo de desenvolvimento(BROWN; VRANESIC, 2003).

2.6.1 Linguagem de descrição de *Hardware* Verilog

A linguagem de descrição de *hardware* Verilog é uma notação formal para uso em todas as fases da criação de sistemas eletrônicos, servindo para o desenvolvimento, verificação, síntese e teste de projetos de *hardware* (IEEE, 2006). A linguagem foi desenvolvida em meados dos anos 1980 e, posteriormente, transferida para o IEEE (*Institute of Electrical and Electronics Engineers*). A linguagem é formalmente definida pela norma IEEE 1364. O padrão foi ratificado em 1995 e referido como padrão IEEE 1364-1995, revisto em 2001 e referido como padrão IEEE 1364-2001, e por fim em 2005 foi revisto mais uma vez e referido como padrão IEEE 1364-2005 (IEEE, 2006; CHU, 2008). No Apêndice A é feita uma introdução á linguagem Verilog e máquinas de estados.

2.7 HLS

Aumentar os níveis de abstração para descrever aplicações em *hardware* é necessário devido ao constante aumento na capacidade dos *chips* de silício e ao aumento da complexidade das aplicações . No domínio do *software*, existia apenas a sequência binária, depois surgiu a linguagem *assembly*, e finalmente surgiram as linguagens de alto nível, ou HLLs (*High Level Languages*) para aumentar a capacidade de produção de *softwares*. O mesmo ocorreu no domínio do *hardware*, quando surgiram as linguagens de descrição de *hardware* (HDLs), tais

como Verilog e VHDL que permitiram a adoção de uma variedade de ferramentas. Porém, era preciso um modo de diminuir o tempo de criação e verificação de *hardware*, foi quando surgiu o HLS (*High Level Synthesis*) que é uma forma de descrever circuitos de *hardware* (reconfigurável e ASIC) em uma linguagem de mais alto nível do que VHDL ou Verilog. O HLS, então, é capaz de transformar uma linguagem de alto nível (por exemplo, C, Java, C++) automaticamente ou semiautomaticamente em um circuito de *hardware*. A linguagem de alto nível pode ser convertida em um RTL (*Register Transfer Level*) (COUSSY et al., 2009). Em projetos de circuitos digitais o RTL é uma forma de modelar um circuito digital. O RTL pode conter unidades lógicas funcionais, unidades de memória, interconexões como barramentos e multiplexidadores. As linguagens de descrição de *hardware*, como Verilog e VHDL, podem utilizar o RTL para criar representações de alto nível de um circuito (VAHID, 2009). A relação entre RTL e HLS é mostrada na Figura 2.22.

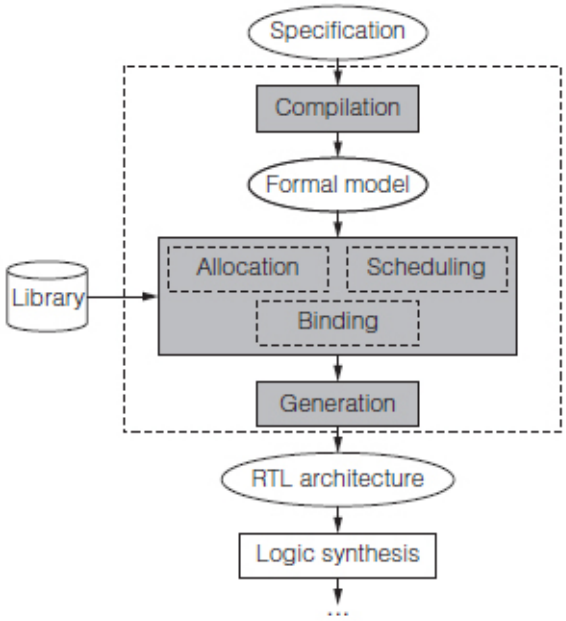

Figura 2.22: Passos de uma ferramenta HLS.
Fonte: Coussy et al. (2009).

A Figura 2.22 mostra os típicos passos de uma ferramenta HLS. A primeira etapa consiste na compilação da especificação funcional, descrita em uma linguagem de alto nível. Nesta etapa, são efetuadas várias otimizações no código, como a eliminação do código morto e falsas dependências de dados. Após isso, a compilação transforma o código de entrada em uma representação formal. Essa representação formal mostra os dados e as dependências, e pode ser representada por um grafo DDG. A atribuição define o tipo e o número de recursos de *hardware* necessários (ex: unidades funcionais, armazenamento, ou componentes de ligação). Todas as operações necessárias no modelo de especificação devem ser escalonadas em ciclos

de *clock*. Na etapa de ligação variáveis e operações são conectadas às unidades funcionais e de armazenamento criadas nos passos anteriores. O último passo é a geração do RTL.

2.8 Compilador Cetus Modificado

O projeto inicial do Cetus foi desenvolvido por Dave et al. (2009) e Lee, Johnson e Eigenmann (2004), onde a proposta original foi criar um compilador em java com o objetivo de transformar um código fonte na linguagem C em um novo código fonte C paralelizado de forma automática, através de análises e transformações que visam a paralelização e otimização automática do código fonte. O código C gerado pelo compilador Cetus será executado pela mesma plataforma de entrada, uma CPU ou GPU. O resultado do processamento do código pelo Cetus é um código C com *pragmas* OpenMP. Esses *pragmas* são comandos colocados nas regiões do código onde o mesmo deve ser paralelizado, auxiliando o compilador com informações adicionais durante a compilação. O OpenMP (*Open Multi Processing*) é uma API para a programação paralela que utiliza *pragmas* para definir regiões paralelizáveis no código. As regiões com *pragmas* serão executadas paralelamente em várias *threads* diferentes.

O compilador Cetus realiza uma sequência de passos para transformação do código fonte dependendo de sua configuração. Os passos que serão mostrados são os que foram utilizados no trabalho de Rettore (2012). O primeiro passo é analisar e converter as anotações externas presentes no código C, em anotações internas ao Cetus. Em seguida, serão feitos alguns passos que simplificam e analisam o código, como identificar e substituir variáveis de indução, simplificar operações matemáticas e encontrar variáveis privadas. As variáveis de indução são variáveis que induzem dependência de dados como, por exemplo, na expressão $vi = vi + expr$, vi pode ser substituído do lado direito da expressão para evitar uma dependência. A manipulação de equações matemáticas, simplifica as expressões integrando símbolos e substituindo expressões complicadas por expressões mais simples. Por exemplo, a expressão $1 + 3 \times a - 2 \times a + 4$ ficaria $5 + a$. Após os passos de simplificação do código, será então executado o passo responsável por realizar os testes de dependência de dados. Esses testes vão identificar variáveis que possuem algum tipo de dependência. Os passos seguintes irão gerar as anotações em OpenMP. Primeiro é realizada uma análise para detectar variáveis no *loop* que podem receber a operação de redução. Nessa operação cada *thread* tem uma cópia local da variável, e uma variável global compartilhada resume (reduz) os valores das cópias locais, e uma anotação de redução é adicionada antes do *loop*. O código então, é varrido levando em consideração as análises de dependência de dados para gerar as anotações em *loops* que serão executados em paralelo. Os *pragmas* do OpenMP então, são inseridos no código fonte. Para exemplificar o funcionamento

do compilador Cetus, é apresentado na Listagem 2.2 um código em C contendo um *loop* que será executado de forma sequencial e na Listagem 2.3 é apresentado o mesmo código após as análises e transformações do compilador Cetus.

Listagem 2.2: *Loop* em C que será executado de forma sequencial. Fonte:Dave et al. (2009).

```
1  int foo(void)
2  {
3          int i;
4          double t,s,a[100];
5          for(i=0;i<50;++i)
6          {
7                  t=a[i];
8                  a[i+50]=t+(a[i]+a[i+50])/2.0;
9                  s=s+2*a[i];
10         }
11         return 0;
12 }
```

Listagem 2.3: *Loop* em C que será executado de forma paralela utilizando OpenMP. Fonte:Dave et al. (2009).

```
1  int foo(void)
2  {
3          int i;
4          double t,s,a[100];
5          #pragma cetus private(i,t)
6          #pragma cetus reduction(+:s)
7          #pragma cetus parallel
8          #pragma omp parallel for reduction(+:s)
9          private(i,t)
10         for (i=0;i<50;++ i)
11         {
12                 t=a[i];
13                 a[(i+50)]=(t+((a[i]+a[(i+50)])/2.0));
14                 s=(s+(2*a[i]));
15         }
16         return 0;
17  }
```

A proposta de Rettore (2012) é criar uma ferramenta próxima a um HLS e, para isso, utilizou o Cetus que foi modificado para gerar o *loop pipelining* a partir do código, na forma de máquinas de estados finito. Estas serão usadas como entrada em ferramentas de EDA(*Electronic*

Design Automation), de modo a produzir circuitos de *hardware* para a execução em *loop pipelining*. O Cetus foi utilizado por Rettore (2012), principalmente pelos recursos avançados para análise de dependência de dados que ajudam a gerar grafos de dependência (DDG), o que auxilia bastante na implementação do *loop pipelining*.

As principais alterações no compilador Cetus foram basicamente: adição de bibliotecas de manipulação e exibição de grafos para criação do DDG, seleção de métodos do Cetus para preencher o DDG com dados do *loop*, alteração do DDG extraído do Cetus para apresentar todas as dependências, escalonamento das instruções utilizando o *loop pipelining* e criação de uma saída para o compilador Cetus modificado.

O primeiro passo para a modificação do Cetus foi adicionar uma biblioteca de manipulação de grafos ao projeto, para que as informações das análises e transformações do Cetus fossem extraídas e utilizadas pelo projeto. O objetivo de adicionar uma biblioteca de manipulação de grafos é poder criar um DDG, que será utilizado no *loop pipelining*. Para a manipulação de grafos foi utilizada a biblioteca JgraphT (NAVEH et al., 2008). A partir dessa biblioteca foram criadas e adicionadas novas classes de manipulação de grafos ao Cetus com o objetivo de receberem informações de um *loop*. Os dados do *loop*, que irão inserir informações nessas classes, serão extraídos das análises realizadas pelo Cetus. Basicamente, esses dados são as instruções do corpo do *loop*. Mas ainda é preciso visualizar o DDG, pois foram criadas somente as classes internas que irão conter dados do *loop* e que não permitem a sua visualização. Para visualização do DDG, seus dados são exportados para a linguagem Dot, que é uma linguagem utilizada especificamente para descrever grafos. Com os dados do DDG no padrão Dot, é utilizada então a ferramenta Graphviz (ELLSON et al., 2002) para exibir essas informações (RETTORE, 2012). A Figura 2.23 mostra um DDG exibido pela ferramenta Graphviz e a Listagem 2.4 mostra o trecho de código que é exibido pelo DDG. No grafo o trecho do código é impresso dentro de cada nó e as arestas podem ser sólidas nas cores preto e vermelho ou tracejadas nas cores marrom, verde e azul. As arestas na cor preto representam dependências verdadeiras entre os vértices e, as arestas na cor vermelho representam arestas não validadas ou incoerentes e que devem ser removidas. As arestas tracejadas na cor marrom representam as antidependências, na cor verde as dependências de saída e na cor azul as dependências de entrada entre os vértices.

Listagem 2.4: Exemplo de um trecho de código fonte para geração do grafo em formato Dot. Fonte:Rettore (2012).

```
1 ...
2 c[i]= a[i];
3 3 a[i]= 20 * i;
```

```
4 4 b[i]= a[i] * k[i];
5 5 k[i]= c[i] + b[i];
6 ...
```

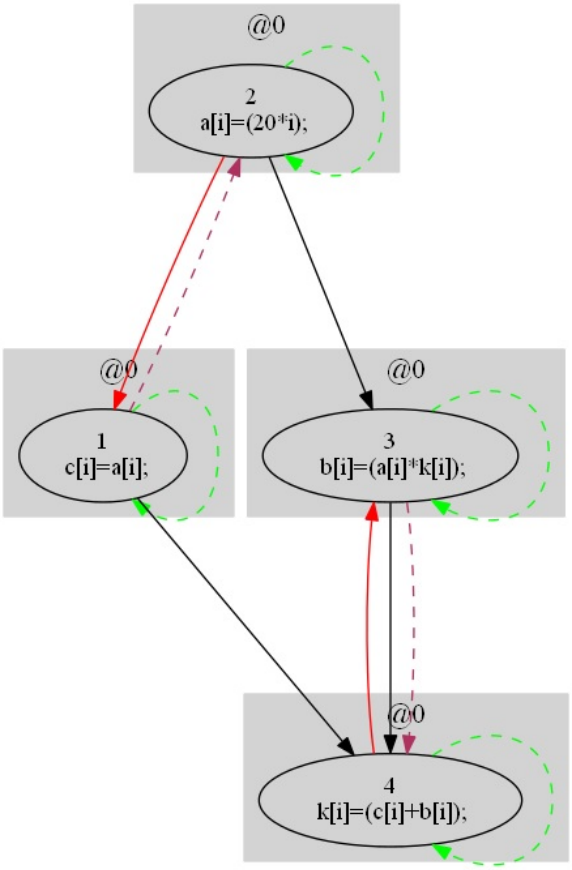

Figura 2.23: Exemplo de grafo exportado para Dot e plotado no Graphviz com todas as dependências existentes.

Fonte: Rettore (2012).

O segundo passo foi conseguir acesso aos métodos de análise do compilador Cetus. Foram então identificadas e extraídas as informações necessárias que serão úteis na criação do *loop pipdelining*. Os métodos selecionados para recuperar essas informações foram: o método responsável por identificar quais partes do *loop* serão executadas em paralelo e o método responsável por fazer os testes e análises de dependências. Esses métodos foram escolhidos, pois são os que disponibilizam mais informações sobre as dependências de dados. Isso tudo foi feito com o objetivo de extrair dados necessários para montar um DDG. Então, as classes criadas a partir da biblioteca de manipulação de grafos irão receber os dados dessas análises do Cetus (RETTORE, 2012).

O terceiro passo foi realizar transformações no DDG extraído do compilador considerando todo o fluxo de controle do programa, devido às necessidades das arquiteturas de *hardware*

customizado, pois o DDG antigo não mostrava todos os tipos de dependências. O grafo DDG foi fundido ao grafo CFG (*Control Flow Graph*), o que possibilitou a exploração de todo o fluxo de controle do programa (código fonte) e dependências entre as instruções. Todas as informações mostradas nos grafos de dependência são referentes apenas ao corpo do *loop* (RETTORE, 2012).

Com o DDG completo as instruções, então, são escalonadas através da técnica *loop pipelining*. Após o escalonamento o grafo DDG recebe então a diretiva @ seguida de um valor numérico que é responsável por dizer a ordem de execução das instruções, respeitando as dependências e possibilitando a otimização na execução do programa. Uma vez gerada, a máquina de estados, teoricamente, estaria pronta para ser sintetizada em *hardware*.

O último passo implementado no compilador Cetus é gerar uma saída que será executada em um FPGA. A saída do compilador Cetus modificado será uma máquina de estados finitos, e também um código LALP proposto por MENOTTI (2010), porém este não é considerado pelo presente trabalho.

A Figura 2.24 mostra o fluxo completo do compilador Cetus modificado. Primeiro é realizada a criação do DDG através dos métodos do Cetus que realizam análises no código e das classes criadas para a manipulação de grafos. Porém, o DDG que foi criado não estava completo, pois as análises do Cetus consideram apenas as dependências com vetores, desconsiderando as variáveis escalares. Então foi feita uma fusão do CFG criado a partir de "anotações" com o DDG. As anotações são um conjunto de procedimentos criados para extrair o Grafo de Fluxo de Controle (CFG). Agora o DDG está completo com todos os nós e suas dependências e pode ser escalonado, e a partir do escalonamento é criada a FSM. Todas essas modificações foram nomeadas de Passo FPGA no compilador Cetus.

Ao executar uma FSM que implementa o *loop pipelining* em um FPGA, é possível que ocorra uma sobreposição de valores, conforme mostrado na Seção 2.2. Para resolver esse problema, é necessário um esquema especializado para alocação e armazenamento de valores (registradores), o que constitui o foco principal deste trabalho. Na próxima seção são apresentadas algumas técnicas para alocação de registradores em *loop pipelining*, as quais servirão de base para o desenvolvimento deste trabalho.

2.8.1 Saídas do Compilador Cetus Modificado

O compilador Cetus Modificado foi configurado para gerar duas máquinas de estados como saída a partir do código em C definido como entrada. Foram dados os nomes *FSM-Target*

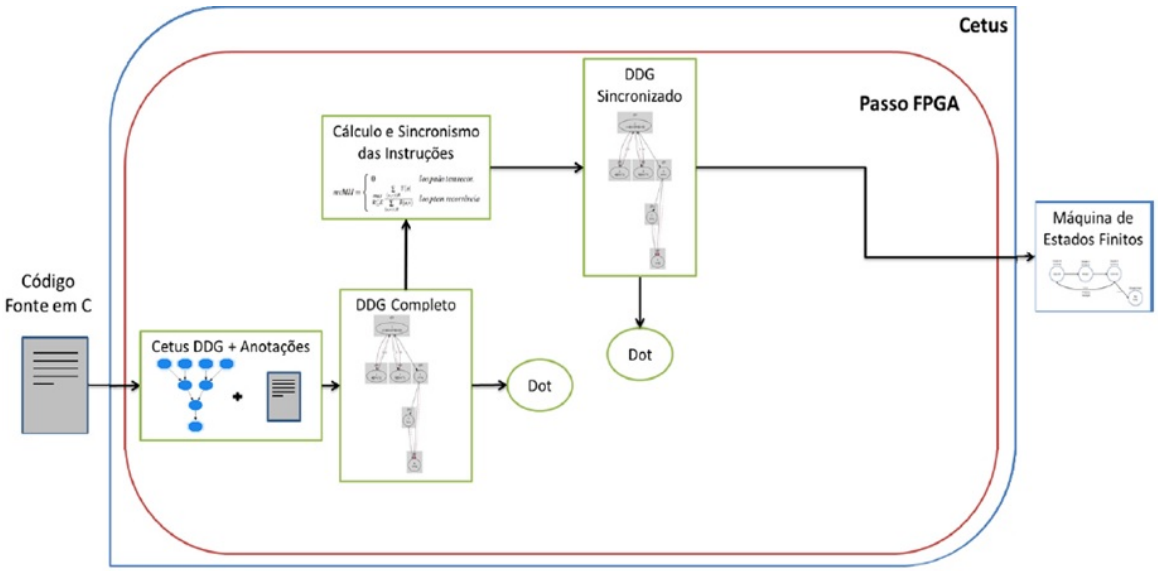

Figura 2.24: Visão geral do fluxo de transformações realizadas no código fonte do Cetus para gerar uma FSM.

Fonte: Rettore (2012).

Machine 1 e *FSM-Target Machine 2* para essas máquinas. Embora elas estejam em pseudo código e, portanto, ainda não sejam código Verilog, as máquinas podem ser utilizadas para criação do código em Verilog manualmente. A Listagem 2.5 mostra um código em C que será paralelizado pelo compilador Cetus Modificado e a Figura 2.25 mostra o escalonado do código utilizando a técnica *loop pipelining*.

Listagem 2.5: Exemplo de um trecho de código fonte para geração de uma FSM pelo compilador Cetus modificado.

```
1  int main()
2  {
3          int i;
4          int s=0;
5          int n = 10;
6          int b, c, k;
7          int a = 0;
8
9          for (i=0; i<n; i++)
10         {
11
12                 c = a + 10;
13                 a = c * 2;
14                 b = c * 5;
15                 s = s + c;
```

```
16                      k = s;
17                      s = s + k;
18              }
19      }
```

	Ciclos	i	i+1	i+2	i+3	...
	0	1				
	1	2,3,4	1			
	2	5	2,3,4	1		
kernel {	3	6	5	2,3,4	1	
	4		6	5	2,3,4	1
repetição do *kernel* para outras interações {	5			6	5	2,3,4
	6				6	5
	7					6
	:			:		

Figura 2.25: Escalonamento do trecho de código da Listagem 2.5.
Fonte: Rettore (2012).

A partir do escalonamento da Figura 2.25, o compilador Cetus Modificado gerou duas máquinas de estados que podem ser vistas na Listagem 2.6 (*FSM-Target Machine 1*) e Listagem 2.7 (*FSM-Target Machine 2*). A primeira, é uma máquina de estados voltada para a execução em microprocessadores e a segunda é uma máquina de estados voltada para a execução em *hardware*. Exemplos de como implementar essas listagens em Verilog são mostrados no Apêndice B.

Apesar das duas arquiteturas funcionarem, dependendo do tipo do código de entrada, é mais aconselhado o uso de um ou de outro tipo de arquitetura. Nesse caso, a *FSM-Target Machine 2* consome mais ciclos de *clock* para ser executada, pois o *kernel* foi dividido em três estados, enquanto na *FSM-Target Machine 1* o *kernel* ocupa apenas um estado. Estas são *targets* do Cetus Modificado, as arquiteturas alvo propostas neste projeto não implementam especificamente nenhuma das duas *targets*. Assim, estas *targets* e as arquiteturas alvo propostas são estruturas distintas.

Listagem 2.6: *FSM-Target Machine 1*.

```
1  ----------Prologue----------
2  Instructions per state
3  State 0:
4  | Interaction || Instruction
5  |      0      || 1 - c=(a+10);
6
7  State 1:
8  | Interaction || Instruction
```

```
|       0       || 2 - a=(c*2);
|       0       || 3 - b=(c*5);
|       0       || 4 - s=(s+c);
|       1       || 1 - c=(a+10);

State 2:
| Interaction || Instruction
|       0       || 5 - k=s;
|       1       || 2 - a=(c*2);
|       1       || 3 - b=(c*5);
|       1       || 4 - s=(s+c);
|       2       || 1 - c=(a+10);


---------Kernel---------
Instructions per state
State 3:
| Interaction || Instruction
|    (n + 0)    || 6 - s=(s+k);
|    (n + 1)    || 5 - k=s;
|    (n + 2)    || 2 - a=(c*2);
|    (n + 2)    || 3 - b=(c*5);
|    (n + 2)    || 4 - s=(s+c);
|    (n + 3)    || 1 - c=(a+10);

Number of Repetitions: 7

---------Epilogue---------
Instructions per state
State 4:
| Interaction || Instruction
|       7       || 6 - s=(s+k);
|       8       || 5 - k=s;
|       9       || 2 - a=(c*2);
|       9       || 3 - b=(c*5);
|       9       || 4 - s=(s+c);

State 5:
| Interaction || Instruction
|       8       || 6 - s=(s+k);
|       9       || 5 - k=s;

State 6:
```

```
| Interaction || Instruction
|       9     || 6 - s=(s+k);
```

Listagem 2.7: ***FSM-Target Machine 2*.**

```
----------Prologue----------
Instructions per state
State 0:
| Interaction || Instruction
|       0     || 1 - c=(a+10);
State 1:
| Interaction || Instruction
|       0     || 2 - a=(c*2);
|       0     || 3 - b=(c*5);
|       0     || 4 - s=(s+c);
State 2:
| Interaction || Instruction
|       1     || 1 - c=(a+10);
State 3:
| Interaction || Instruction
|       0     || 5 - k=s;
|       1     || 2 - a=(c*2);
|       1     || 3 - b=(c*5);
|       1     || 4 - s=(s+c);
State 4:
| Interaction || Instruction
|       2     || 1 - c=(a+10);
----------Kernel----------
Instructions per state
State 5:
| Interaction || Instruction
|    (n + 0)  || 6 - s=(s+k);
State 6:
| Interaction || Instruction
|    (n + 1)  || 5 - k=s;
|    (n + 2)  || 2 - a=(c*2);
|    (n + 2)  || 3 - b=(c*5);
|    (n + 2)  || 4 - s=(s+c);
State 7:
| Interaction || Instruction
|    (n + 3)  || 1 - c=(a+10);
Number of Repetitions: 7
----------Epilogue----------
Instructions per state
```

```
State 8:
| Interaction || Instruction
|       7     || 6 - s=(s+k);
State 9:
| Interaction || Instruction
|       8     || 5 - k=s;
|       9     || 2 - a=(c*2);
|       9     || 3 - b=(c*5);
|       9     || 4 - s=(s+c);
State 10:
| Interaction || Instruction
|       8     || 6 - s=(s+k);
State 11:
| Interaction || Instruction
|       9     || 5 - k=s;
State 12:
| Interaction || Instruction
|       9     || 6 - s=(s+k);
```

2.9 Trabalhos Relacionados

Neste capítulo, são apresentadas algumas pesquisas que foram realizadas na área. Foram escolhidas as pesquisas que mostram o uso do *module scheduling*, *loop pipelining*, paralelização de *loops*, HLS ou da arquitetura VLIW. Muitas dessas pesquisas têm o objetivo de criar aceleradores de *hardware* e executar *loops* em arquiteturas reconfiguráveis. As pesquisas escolhidas possuem uma semelhança com este trabalho na forma como o *loop pipelining* e outras técnicas são modificadas ou otimizadas, seja para a execução em arquiteturas reconfiguráveis, ou para resolver algum problema específico. O objetivo de mostrar essas pesquisas é analisar a forma como outros trabalhos adaptam técnicas de paralelização para arquiteturas reconfiguráveis e, no caso de alguns trabalhos, como foi abordado o problema de sobreposição de valores e alocação de registradores para o *loop pipelining*.

2.9.1 *Loop pipelining* em dispositivos FPGA

Primeiro será analisado um grupo de trabalhos que possuem o objetivo de executar o *loop pipelining* em dispositivos FPGA. O trabalho de Sejong et al. (2008) foi o primeiro analisado; nele, é utilizado o conceito de tradução binária para gerar um acelerador em FPGA. O conceito de tradução binária é, basicamente, a emulação de um conjunto de instruções de uma arquite-

tura alvo por outra arquitetura através da tradução desse conjunto de instruções. O motivo para realizar a tradução binária de um conjunto de instruções pode ser descontinuação de um *hardware*, por ser antigo e não ter mais suporte. No trabalho, é criada uma técnica que sintetiza o *loop pipelining* com as dependências de memória e o executa em FPGA. O conceito de tradução binária é mostrado, pois a arquitetura alvo é um processador de uso geral com um acelerador dedicado, os quais compartilham a mesma memória, e as instruções executadas nessa arquitetura são compiladas para a forma binária. Na tradução binária, analisar as dependências de memória, muitas vezes, é difícil; então, nesse trabalho, foi proposto um *loop pipelining* especulativo com testes de dependência em tempo de execução, a fim de alcançar o *loop pipelining* e superar a limitação de análise de dependência na tradução binária. Essa técnica gera o *netlist* em tempo de compilação. Em seguida, esse *netlist* é modificado de acordo com o resultado de uma segunda análise em tempo de execução. Na técnica, um código é compilado para a forma binária através de um tradutor binário. Na forma binária, é difícil saber muitas das informações sobre as dependências do *loop*; assim, a primeira análise das dependências é bem simples. Em seguida, é realizada uma segunda análise das dependências, que remove acessos redundantes à memória com reuso de dados, essa segunda análise é chamada de análise otimista. Após as análises, é, então, gerado um *netlist* que utiliza a análise otimista como base. Um inspetor de dependências realiza testes junto, ou antes, da execução do *loop*. Se for descoberta alguma dependência que impeça a execução do *loop*, então é realizado um *rollback* e será utilizado um *netlist* gerado através da análise conservativa. Porém, essa técnica é eficiente somente quando o *overhead* do inspetor de dependência pode ser recuperado com a execução do código otimizado. Para o *loop* ser executado em *hardware*, o tradutor binário cria, primeiramente, um arquivo em RTL, do qual é gerado um fluxo de dados. Utilizando um *framework*, o fluxo de dados é convertido em Verilog. Através da ferramenta Xilinx ISE 8.2, o Verilog é, então, mapeado para o *chip* Xilinx FPGA. Para validar o trabalho, foram feitos 9 testes, nos quais houve um *speedup* máximo de 5.81 ao comparar a execução do *loop* em *hardware* em relação à em *software*, quando se utiliza o método especulativo. Quando são comparados os métodos conservativo e especulativo, o *speedup* máximo é de 2.53 . A frequência máxima do *datapath* foi de 204Mhz ao executar o *loop* em FPGA. O método também diminui 66% dos acessos à memória.

O trabalho de Turkington et al. (2008) também utiliza o *loop pipelining* para a execução de *loops* em FPGA. O objetivo dessa pesquisa é criar uma técnica que utiliza o *loop pipelining* para paralelizar um *loop* com vários níveis, pois o *loop pipelining* é, muitas vezes, é restrito ao nível mais interno de um *loop* aninhado. Além de gerar um *loop pipelining* que paralelize *loops* aninhados, os autores também se preocuparam em não diminuir muito a frequência de *clock*, em encontrar o menor escalonamento possível e em executar o *loop* paralelizado em FPGA. O *loop*

pipelining criado a partir dessa nova técnica será executado em FPGA utilizando o dispositivo Stratix II da Altera e a linguagem de descrição de *hardware* VHDL. Foram feitos nove testes e os seus resultados mostraram que a solução mais rápida ocorre quando é aplicado o *loop pipelining* de um a três níveis acima do *loop* mais interno. A técnica proposta teve um *speedup* de 3.2 em relação à solução que paraleliza apenas o *loop* mais interno.

Bem parecido com o trabalho anterior é o trabalho de Lei et al. (2009), cuja principal diferença está na alteração da técnica *loop pipelining*, que é focada na otimização do uso de memórias. Nesse trabalho, é apresentada uma variação do algoritmo *modulo scheduling* para que seja possível explorar o *loop pipelining* em FPGAs. Segundo o trabalho, a maior dificuldade em criar um *loop pipelining* para FPGA está no controle dos acessos à memória e, também, em gerar operações paralelizadas em um CDFG(*Control and Data Flow Graph*). O método proposto é realizar o escalonamento aproveitando o tempo de espera necessário para que as memórias façam suas operações e, também, tratar as dependências circulares. Com isso, espera-se um MII o mais eficiente possível. Para isso, foi criado um *framework* que recebe, como entrada, o CDFG e realiza o escalonamento do *loop* em dois passos. No primeiro passo, é feito o escalonamento levando em conta as dependências circulares. No segundo passo, é implementada a ideia de utilizar o tempo de espera da memória. Para realizar os testes, foi utilizado o compilador FREEDOM, que transforma código binário e *assembly* em VHDL ou Verilog. Para calcular o desempenho do *loop pipelining*, foram utilizados *benchmarks* focados na arquitetura TI C64x. O dispositivo utilizado foi a placa Xilinx Virtex II Pro. O *loop pipelining* foi implementado em um CDFG e depois traduzido para VHDL. A simulação do circuito foi feita utilizando ModelSim e a síntese foi feita utilizando Synplicity's Synplify Pro v9.1. Ao total, foram testados cinco algoritmos, sendo feitas três variações de cada um. As variações são: sem paralelização, com *modulo scheduling* tradicional e com *modulo scheduling* modificado. Em média os testes com o *modulo scheduling* modificado tiveram uma melhoria no desempenho de 15% em relação ao *modulo scheduling* tradicional, devido à redução do II. Além do aumento no desempenho, também houve um aumento, em média, de 6% na área utilizada, ao comparar essas duas abordagens.

Outro trabalho com mesmo tema é o trabalho de Qi et al. (2013). Nele, foi desenvolvida uma técnica de escalonamento de *loops* para plataformas reconfiguráveis utilizando o *loop pipelining*. Para realizar o escalonamento, o *loop* é abstraído em um grafo WDFG (*weighted data flow graph*). Nesse grafo, os nós representam as tarefas realizadas em um *loop* e o seu tempo de execução, e as arestas representam a dependência entre essas tarefas. Cada aresta recebe um rótulo, que significa o *delay* para executar o próximo nó. Esse rótulo representa, por exemplo, o tempo de transferência de um registrador para outro, que é chamado de *overhead*

de comunicação. Para a paralelização do *loop*, foi criado um *framework* que utiliza a técnica de *loop pipelining* com uma otimização, na qual o primeiro passo é criar o grafo WDFG a partir das instruções do *loop*. Após a criação do WDFG, ele é analisado com a finalidade de encontrar a tarefa com o maior tempo de execução e que possa diminuir a eficiência do escalonamento. A maior tarefa, então, pode ser duplicada e executada paralelamente com as outras instruções, ou dividida em duas tarefas menores que possuem dependência entre si. Os testes com a técnica foram feitos para *software* e *hardware* utilizando 7 algoritmos diferentes, sendo 6 testados em *software* e 1 testado em *hardware*. Para os testes em *software*, foi utilizado um processador Intel Core Dual CPU com 2.93GHz e, para os testes em FPGA, foi utilizado o *chip* Xilinx Virtex-5 com a placa XC5VLX110T. Para calcular a melhora que a técnica trouxe, também foram feitos testes com os algoritmos utilizando o *loop pipelining* sem nenhuma otimização. Tanto em *software* quanto em *hardware*, a técnica trouxe um aumento no *speedup* e na eficiência dos algoritmos paralelizados: o *speedup* teve uma média de aumento de 2 vezes e a eficiência teve uma média de aumento de 1.5 vezes.

No trabalho de Ben-Asher, Meisler e Rotem (2010), é feita uma otimização no algoritmo *modulo scheduling* para a execução de *loops* de forma mais eficiente em FPGAs. Essa otimização foi feita para reduzir o tempo de execução, que, muitas vezes, tem um significativo aumento devido às restrições do código. A redução do número de portas nas memórias utilizadas na síntese é uma forma de resolver esse problema, pois uma memória que fornece acessos simultâneos precisa ter várias portas, e isso implica em maiores atrasos devido à latência da memória. Uma forma de reduzir o número de portas é reduzindo o número de acessos a memórias em paralelo em cada linha do *Kernel*. Portanto, a proposta do trabalho é diminuir o número de acessos simultâneos à memória em uma linha do *Kernel* sem aumentar o MII. A técnica se baseia em inserções de operações falsas no *Kernel* e, ao inserir essas operações, realiza um deslocamento de operações. Com isso, o número máximo de acessos em paralelo à memória pode ser diminuído. Nos testes, os módulos foram implementados em Verilog e sintetizados no *chip* Virtex-5 utilizando o Xilinx-ISE, e os *loops* originais foram executados em um processador ARM7. Foram testadas a execução de 9 algoritmos em FPGA e também no processador ARM7. Ao comparar os resultados, a técnica em FPGA, em relação ao ARM7, mostrou um *speedup* médio de 75, sendo o mínimo de 15 e o máximo de 115.

2.9.2 Outros tipos de paralelização de *loops* e outros tipos arquiteturas reconfiguráveis

Todos os trabalhos mostrados até então possuem objetivos parecidos, em que o *loop pipelining* é utilizado para gerar *loops* paralelizados para dispositivos FPGAs. Agora, será mostrado um grupo de pesquisas com objetivos parecidos com os trabalhos mostrados anteriormente, mas que podem utilizar outras técnicas de paralelização de *loops* ou outros tipos de arquiteturas reconfiguráveis.

O primeiro trabalho desse grupo é o trabalho de Banu et al. (2013). A proposta desse trabalho é aumentar o desempenho do algoritmo AES (*Advanced Encryption Standard*) através de técnicas de paralelização em *software* e *hardware*. O AES é um algoritmo de encriptação, normalmente utilizado em dispositivos *wireless*, aplicações embarcadas, HDTV (*High Definition TV*) e videoconferência. Uma forma de conseguir um maior desempenho para o algoritmo é aumentando o número de unidades funcionais para aumentar o nível de paralelismo. Tanto a paralelização em *software* quanto a paralelização em *hardware* do algoritmo AES oferecem as suas vantagens. A vantagem do uso de *software* está na facilidade da implementação, na facilidade de fazer *upgrades*, na portabilidade e na flexibilidade. A implementação em *hardware* oferece uma maior segurança física, o que dificulta a leitura e a escrita de dados caso ocorram ataques. No trabalho, foram feitas as duas abordagens; a paralelização em *software* foi feita utilizando OpenMP e a paralelização em *hardware* foi implementada em FPGA. Para a paralelização em FPGA, foram utilizados a EDA da Xilinx ISE version 6.3i, Modelsim 5.8 e Modelsim 6.0 SE Plus para simulação, em que a linguagem de descrição de *hardware* escolhida foi Verilog. Nos testes, o tempo de execução da implementação em OpenMP foi, em média, de 76 segundos, e em FPGA, 4×10^{-09} segundos, mostrando que a implementação em *hardware* teve um desempenho melhor.

O trabalho de Weinhardt, Krieger e Kinder (2013) é o próximo a ser analisado. A proposta desse trabalho é criar um *framework* para criar aplicações em um *hardware* reconfigurável, com o objetivo de gerar aceleradores de *hardware*. A ideia é que as aplicações sejam facilmente re-escaláveis e portáveis para tipos diferentes de dispositivos FPGAs. Então, as aplicações são desenvolvidas uma vez e se adaptam automaticamente para diferentes tipos de dispositivos FPGAs, e com tamanhos variados. O dispositivo que contém o *chip* FPGA irá se comunicar com o computador através do barramento PCI-Express. O computador também irá executar as aplicações caso não exista nenhuma placa conectada ao barramento PCI-Express. Essa flexibilidade é conseguida através de uma API, que também é responsável pela comunicação entre os processos do PC com o *chip* FPGA. O paralelismo explorado no trabalho é em nível de

tarefa; para isso, uma aplicação é particionada em várias tarefas através do uso da API de comunicação, na qual o código é descrito em VHDL. Para deixar o *framework* flexível para vários tipos diferentes de dispositivos, foi criado um conjunto de módulos em VHDL que devem ser instanciados em todos os *designs* que forem criados em *hardware*. Esses módulos foram chamados de invólucro PCI-Express, e nele, dependendo do tipo de FPGA utilizada, alguns valores são modificados. O invólucro PCI-Express irá utilizar uma frequência de *clock* diferente para cada tipo de FPGA e de arquitetura. Em cada ciclo de *clock*, um pacote de dados pode ser gerado ou consumido. Como o computador possui uma frequência de *clock* diferente do invólucro PCI-Express, filas foram criadas para comunicação entre os dois domínios de frequência. A paralelização no *hardware* é feita dividindo a aplicação em PEs (*Processing Elements*) que processam os dados em paralelo. O número de PEs é definido por um parâmetro *R*, sendo assim, *R* determina o grau de paralelismo. O valor de *R* é ajustado automaticamente através de várias informações do dispositivo utilizado, e que são dadas pelo usuário. Para os testes, foi utilizado um computador com processador Intel Core i5 e sistema operacional Linux. O computador foi equipado com dois dispositivos FPGAs diferentes, a placa Xilinx SP605 x1 lane GEN1 (com o chip FPGA XC6SLX45T Spartan-6) e a placa ML605 x8 lane GEN1 (com o chip FPGA XC6VLX240T Virtex-6). Foram testados quatro algoritmos diferentes. A maior taxa de paralelização foi com $R = 16$, com um *speedup* de 9.2 em relação ao mesmo teste sem nenhuma paralelização.

Todos os trabalhos mostrados anteriormente são focados em FPGA. Agora, será mostrado um trabalho de Gnanaolivu, Norvell e Venkatesan (2010), que utiliza CGRA como arquitetura reconfigurável. A principal proposta desse trabalho é a criação de um algoritmo para escalonar *loops* para CGRAs (*Coarse-Grained Reconfigurable Architectures*) nomeado MCHPSO (*Modulo-Constrained Hybrid Particle Swarm*). Os CGRAs são uma alternativa aos FPGAs e, devido às suas estruturas, são capazes de explorar o paralelismo em *loops*. Os CGRAs são compostos por PEs (*Processing Elements*) que suportam certa quantidade de operações, uma rede de interconexões programável, uma memória de configuração e um controlador. Para a criação do MCHPSO, foram utilizados, como base, o algoritmo PSO (*Particle Swarm Optimization*) e o *Modulo Scheduling*. O PSO foi escolhido na elaboração do algoritmo, pois, com baixo tempo de execução e rápida convergência, soluciona vários problemas combinatórios e de espaço multidimensional. O MCHPSO recebe, como entrada, um programa em C, que é convertido em um grafo DFG (Data Flow Graph). A arquitetura escolhida para o trabalho foi a arquitetura ADRES (*Architecture for Dynamically Reconfigurable Embedded Systems*), que combina pares de processadores VLIW com CGRAs e que será representada por um grafo TAG (*Target Architecture Graph*). O MCHPSO, então, irá combinar o TAG com o DFG e criar um

RRG (*Routing Resource Graph*). O RRG mostra qual recurso da arquitetura é utilizado por qual operação do DDG em relação ao tempo. O algoritmo proposto foi escrito em Java e executado em um processador intel Core 2 Duo com 4GB de RAM e um *clock* de 2GHz. Foram feitos nove testes, os quais foram comparados com testes em outros trabalhos, e que mostraram uma diminuição do tempo de execução.

O trabalho de Jhin-Bin, Kuen-Cheng e Shann (2012) também utiliza CGRAs e sua proposta é uma variação do algoritmo *modulo scheduling* para reduzir a necessidade de recursos de *hardware* em CGRAs. Nesse trabalho, foi utilizada uma técnica de instruções predicadas parecida com a mostrada na Seção 2.4. Os CGRAs consistem em um *array* de uma grande quantidade de unidades funcionais interconectadas por uma rede em forma de malha. Registradores são distribuídos entre os CGRAS para guardar valores temporariamente. As unidades funcionais podem executar operações como adição, subtração e multiplicação. Em contraste com os FPGAs, os CGRAs possuem um pequeno tempo de reconfiguração, baixo *delay* e baixo consumo de energia. Embora os CGRAs tenham várias vantagens, o mapeamento de uma aplicação para um *chip* CGRA é uma tarefa difícil, em que o tempo de execução de uma operação e sua localização precisam ser considerados. Vários parâmetros e decisões podem influenciar a eficiência da execução final do escalonamento. Uma forma de resolver alguns problemas é reduzindo os *overheads* de execuções condicionais; para isso, podem ser utilizadas operações predicadas. A dependência de controle pode ser convertida em dependências de dados através de uma técnica de conversão da cláusula "se". O comportamento de uma operação predicada depende de um operador booleano adicional. Se o operador booleano tem valor verdadeiro, a operação associada com o predicado é executada; se não, a operação é descartada. Nas arquiteturas CGRA, as operações predicadas são mapeadas em recursos individuais, independentemente se o valor de seus predicados é verdadeiro ou falso. Porém, se duas operações predicadas pertencem a caminhos diferentes no grafo CDG (*control dependence graph*) construído antes da conversão da cláusula "se", elas não serão executadas junto da sequencia de execução do código. Essas operações são chamadas de operações disjuntas. Para que essas operações não consumam muitos recursos, elas devem ser atribuídas a um mesmo PE. Assim, é proposta uma variação do algoritmo *module scheduling* para mapear aplicações para CGRAs, em que as operações disjuntas devem compartilhar recursos. Para os testes, foram utilizados 10 algoritmos escalonados e testados em uma CGRA de 3x3. Comparando com trabalhos relacionados, o tamanho do II (intervalo de iniciação) diminuiu, em média, 4.76%.

Os próximos três trabalhos que serão mostrados possuem a arquitetura VLIW e FPGA como foco principal do trabalho. No trabalho de Kapre e DeHon (2011), foi proposto o VLIW-SCORE, que combina o *framework* de programação paralela SCORE (*Stream Computation Or-*

ganized for Reconfigurable Execution) com uma arquitetura hibrida que utiliza VLIW e FPGA. A ideia do VLIW-SOCRE é poder ser utilizado em aplicações *hardware-software* como aceleradores FPGA. Uma aplicação ou abordagem de *design hardware-software* pode ser exemplificada por um acelerador FPGA cuja sua lógica computacional é separada em dois componentes: um componente grande, que é implementado no *chip* FPGA, e um componente pequeno de controle, que pode ser implementado em *software* ou em processadores *soft-core*. Exemplos de processadores *soft-core* são o Xilinx Microblaze ou o Altera NIOS, que são processadores implementados dentro do *chip* FPGA. A vantagem dos processadores *soft-core* é que, com eles, as aplicações *hardware-software* reduzem o uso ou ficam independentes de processadores convencionais. O problema das aplicações *hardware-software* é que elas podem conter gargalos e podem também ter a escalabilidade limitada. O VLIW-SCORE propõe solucionar esses problemas. Um programa em SCORE é composto de um grafo de operadores e memórias interconectados. Os operadores são representados por máquinas de estados. O *framework* SCORE pode ser configurado para gerar diferentes formas de paralelismo, como paralelismo em nível de instrução ou paralelismo em nível de *threads*. Os operadores SCORE serão mapeados para a arquitetura híbrida de duas formas: através do mapeamento estático, no qual o escalonador gera uma configuração em VLIW para a execução que contém o *datapath*, memórias, e o fluxo de controle; e através do mapeamento dinâmico, em que os operadores serão mapeados para o FPGA. A lógica do mapeamento estático é ativada quando a lógica do mapeamento dinâmico determina que as portas de transmissão possuem dados válidos e espaço para escrever nas saídas. A proposta desse trabalho foi validada através do caso de estudos SPICE (*Simulation Program with Integrated Circuit Emphasis*). Foram feitos vários testes com três tipos de arquitetura diferentes: CPU e FPGA, microblaz, e VLIW hibrido. Nos testes, o VLIW híbrido obteve o melhor *speedup* de, no mínimo 2.6, e no máximo, 11.1. O dispositivo Virtex-6 LX760 foi utilizado para os testes.

O trabalho de Brost et al. (2011) também é focado em paralelização utilizando a arquitetura VLIW, que é implementada em FPGA com a intenção de ser utilizada no processamento de imagens digitais em tempo real. Os algoritmos que serão paralelizados primeiro são implementados em C e, depois, são convertidos em uma linguagem de programação semelhante ao *assembly*, que se chama *Lcode*. Essa linguagem é otimizada para explorar o ILP e seu conjunto de instruções. A conversão de C para *Lcode* é feita pelo compilador OpenIMPACT. O *Lcode*, então, é analisado e reorganizado para instruções VLIW. O processador VLIW é gerado em VHDL, a partir de análises feitas no *Lcode*. Os testes experimentais da técnica foram feitos em um computador com um processador Intel Pentium4 com Linux. As simulações foram feitas utilizando o simulador Xilinx ISE configurado para a arquitetura Virtex 6-xc6vlx75T. Foram

utilizados 3 algoritmos para os testes, que foram testados utilizando o processador VLIW implementado em FPGA e, também, um processador DSP TMS320 para comparação. O parâmetro para comparação entre os processadores foi o número de imagens processadas por segundo, em que o processador VLIW teve uma maior taxa de processamento de imagens por segundo que o processador DSP.

O trabalho de Purnaprajna e Ienne (2012) também é focado em VLIW e FPGA; nele, são propostas modificações na arquitetura FPGA que sejam viáveis à criação de um processador soft-VLIW. Essa ideia surgiu porque se notou que, quando um processador VLIW é mapeado para um *chip* FPGA, o mesmo não realiza um bom uso dos recursos que o *chip* oferece. Essa ineficiência ocorre porque o processador VLIW é custoso em termos de área, devido a sua grande quantidade de registradores. A principal barreira para a implementação de um processador VLIW em um *chip* FPGA foram os CLBs e a quantidade de portas aceitas por memórias RAM em FPGAs. O processador VLIW, então, deve se adaptar à limitação de memória imposta pelo *chip* FPGA. Para solucionar esse problema, foi proposta uma alteração na arquitetura dos *chips* FPGAs para fazer os processadores VLIW eficientes no quesito área e consumo de energia. A solução foi introduzir uma memória RAM com múltiplas portas, que pode ser customizada para corresponder com o tamanho das portas do processador VLIW. Nos testes, a área ocupada no FPGA modificado foi 102 vezes menor e o consumo de energia foi reduzido em 41%.

2.9.3 Sobreposição de valores e alocação de registradores

Todos os trabalhos encontrados que possuem algum objetivo envolvendo arquiteturas reconfiguráveis já foram mostrados. Agora, serão mostrados os trabalhos de Rong, Douillet e Gao (2008), Tyson, Smelyanskiy e Davidson (2001) e Stotzer e Leiss (2009), cujos objetivos estão relacionados a adaptações e análises do *loop pipelining* e das técnicas citadas na Seção 2.3. O trabalho de Rong, Douillet e Gao (2008) investiga a alocação de registradores quando é utilizado o *loop pipelining* aplicado a *loops* de *n* dimensões. Quando um *loop* de uma dimensão é escalonado pelo *loop pipelining*, os ciclos de vida das variáveis desse *loop*, em sucessivas iterações, formam um padrão que se repete. Uma das formas de alocar esses registradores é criar um vetor para representar esses ciclos de vida e mapear esse vetor para um registrador de rotação (RAU et al., 1992). Nesse vetor de ciclos de vida, o eixo horizontal representa o tempo e o eixo vertical representa os registradores fisicamente. Porém, o escalonamento do *loop pipelining* para *loops* multidimensionais é mais complexo, o que exige outras abordagens para a alocação de registradores. No trabalho, o problema é abordado como um vetor multidimensional de ciclos de vida que deve ser visualizado como se estivesse sobreposto em um cilindro. A

solução criada foi implementada no compilador ORC para a arquitetura Itanium. Os testes que foram feitos foram comparados com testes de outros estudos e mostraram que, com a técnica, houve uma redução no uso de registradores.

No trabalho de Tyson, Smelyanskiy e Davidson (2001), é examinada a eficiência do mecanismo de *hardware* chamado *Register Queues* (RQs). Com os RQs, o compilador pode alocar registradores para armazenar valores que estão vivos no *loop*. Os RQs combinam a maioria dos aspectos dos registradores de rotação (RAU et al., 1992) e a técnica de registradores de conexão (KIYOHARA et al., 1993) para gerar um escalonamento eficiente do *loop pipelining*. Através do uso dos QRs, foi possível diminuir a expansão do código e acabar com o fator de limitação do espaço nos registradores, que ocorrem com as técnicas MVE e RRF. Ao contrario de outras técnicas, nas quais os registradores baseados em filas são propostos para a arquitetura VLIW como o QRF, nesse trabalho, tais registradores baseados em filas são propostos para a arquitetura superescalar. Para que os RQs sejam compatíveis com qualquer arquitetura, foi necessário adicionar uma nova instrução, $rq-connect$, que é o suficiente para adicionar RQs a qualquer conjunto de instrução de uma arquitetura. A estrutura dos RQs é composta de três partes: um conjunto de registradores em formato de fila, um conjunto de registradores simples e uma tabela que mapeia os dados para os registradores em formato de fila (usando a instrução $rq-connect$) ou para os registradores simples (usando a lógica comum de nomeação de registradores) . Para demonstrar a capacidade do RQ, ele foi comparado com as técnicas RR e MVE. Os *loops* estudados foram obtidos do Perfect Club Suite, do SPEC e o Livermore Kernels. Foram selecionados 983 *loops* para o estudo, os quais foram compilados pelo Compilador Cydra 5 Fortran77. Com os testes foi notado que ocorreu uma redução na requisição de registradores e no tamanho do código.

No trabalho de Stotzer e Leiss (2009) é proposta uma técnica composta de *software* e *hardware* para tratar o problema de sobreposição de valores que ocorre nos *loops* escalonados pelo *modulo scheduling*, que é mostrado na Seção 2.2.1. No artigo, a sobreposição de valores é dividida em dois casos. No primeiro caso, o algoritmo *modulo scheduling*, ao tentar achar um escalonamento com o menor valor possível de II, coloca as operações de tal forma que os ciclos de vida de algumas variáveis são esticadas além do valor de II. No segundo caso, alguns tipos de dependências de dados causam restrições que forçam os ciclos de vida de algumas variáveis serem maiores do que o tempo do II. Utilizando técnicas de análise de dependências, é possível prevenir a sobreposição de valores. O algoritmo *modulo scheduling* não adiciona as antidependências *loop-carried* ao grafo DDG, o que permite que o ocorra o primeiro caso citado. Através de análises de dependências, as antidependências *loop-carried* foram adicionadas ao grafo DDG, prevenindo que o primeiro caso ocorra; porém, essa solução também aumenta o ta-

manho do RecMII. Para que o segundo caso de sobreposição de valores não ocorra, os ciclos de vida sobrepostos são divididos antes do escalonamento. Além de mudanças no escalonamento, também foi adicionado uma modificação no *hardware* para tratar o problema de sobreposição. Foi adicionada uma latência de atribuição aos registradores, que é um tempo em que um registrador é reservado para a escrita de operações que possuem um tempo de execução maior que 1. Os testes foram feitos com o processador C674 VLIW e 1800 *loops*, mostrando que ocorreu uma melhora na performance dos *loops* que utilizam a técnica em relação aos que não a utilizam. Também foi notado que os fatores que causam a sobreposição de valores foram praticamente removidos do escalonamento final, o que reduz a necessidade de *hardwares* extras. Porém, a técnica necessita de mais testes para que se entenda exatamente o seu impacto no *hardware* e em outros aspectos.

Capítulo 3

MÓDULOS DE HARDWARE ESPECIALIZADOS PARA LOOP PIPELINING

Conforme descrito no Capítulo 1, este trabalho apresenta o projeto e a implementação de estruturas de *hardware* especializadas, objetivando facilitar tarefas de compilação em ferramentas HLS. Propõe-se um arquivo de registradores especializado para uso em *loop pipelining*, bem como um módulo para a implementação de instruções predicadas. A implementação do arquivo de registradores baseado em filas foi fundamentada no trabalho de Fernandes (1998); a arquitetura alvo teve como referência a saída do compilador Cetus Modificado, desenvolvido por Rettore (2012), e a implementação das instruções predicadas foram baseadas no livro de Hennessy e Patterson (2011). As técnicas propostas e o seu desenvolvimento são apresentados neste capítulo, assim como os principais recursos utilizados para que essas técnicas fossem desenvolvidas.

3.1 Visão geral

Antes de iniciar as especificações da implementação do projeto, será mostrada uma visão geral do trabalho. Todos os testes que serão mostrados neste capítulo, foram feitos utilizando as ferramentas de desenvolvimento ModelSim e Quartus II, e alguns testes também foram executados na placa DE-2 da Altera. Neste projeto de pesquisa são estudadas duas arquiteturas diferentes para dar suporte à técnica *loop pipelining*, nas quais são utilizados dois tipos diferentes de arquivos de registradores. Na Figura 3.1 é mostrada a primeira arquitetura ilustrada por um diagrama de blocos. A primeira arquitetura proposta utiliza máquinas de estados e o QRF.

A máquina de estados que implementa o *loop pipelining* foi desenvolvida baseada nas saídas do compilador Cetus, que são mostradas na Seção 2.8.1. O processo de criação da

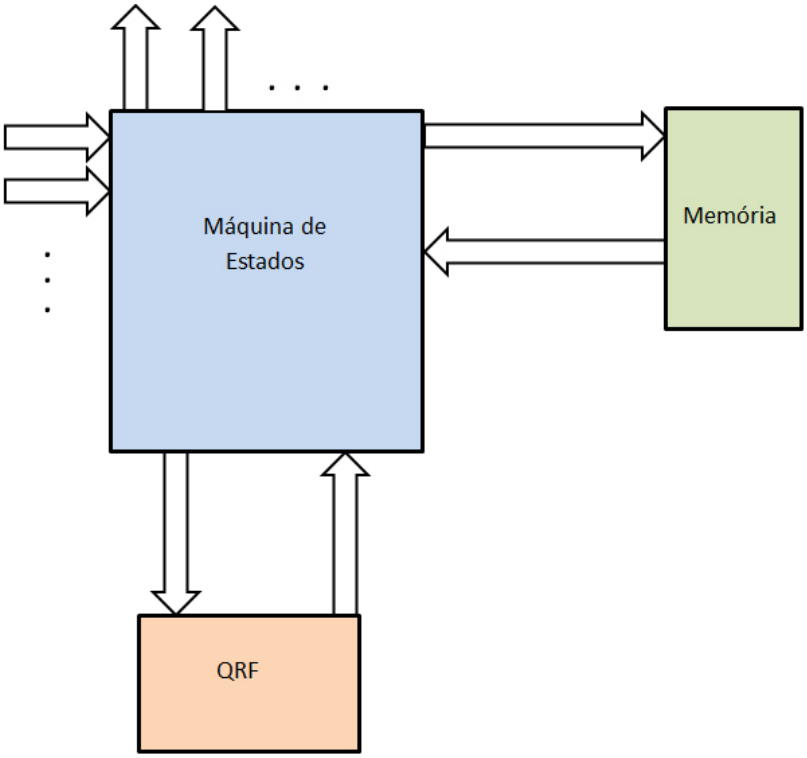

Figura 3.1: Diagrama de blocos da arquitetura proposta com o QRF.

máquina de estados é mostrado na Seção 3.6. A máquina de estados pode ter várias entradas e várias saídas, o que é mostrado pelas setas que entram e saem do bloco azul na Figura 3.1. Para dar suporte ao *loop pipelining* implementado pela máquina de estados, foi desenvolvido um arquivo de registradores baseado em filas ou QRF. O objetivo do QRF é solucionar o problema de sobreposição de valores, ilustrado na Figura 2.15 da Seção 2.3. O desenvolvimento do QRF é mostrado na Seção 3.4. Na Figura 3.1, na qual o QRF é ilustrado pelo bloco laranja, também é mostrada uma memória que é utilizada pela máquina de estados, a qual é importante para armazenar os valores calculados por ela. A escolha de qual memória e como utilizá-la é mostrada na Subseção 3.6.1.

Também foi desenvolvido o CP (controle de instruções predicadas), que é ilustrado na Figura 3.2 pelo bloco vermelho. A segunda arquitetura desenvolvida utiliza, além do QRF, também o CP. O exemplo foi mostrado em outra figura, pois são duas arquiteturas diferentes, apesar de as duas utilizarem praticamente os mesmos componentes. O CP ajuda a implementar a ideia de instruções predicadas mostrada na Seção 2.4. Com o controle de instruções predicadas, o número de estados da máquina de estados vai diminuir drasticamente, já que serão criados apenas os estados de *kernel* do *loop pipelining*. A arquitetura que utiliza somente o QRF (Figura 3.1) será chamada de Arquitetura Alvo 1 e a arquitetura que utiliza o QRF e CP(Figura 3.2) será chamada de Arquitetura Alvo 2.

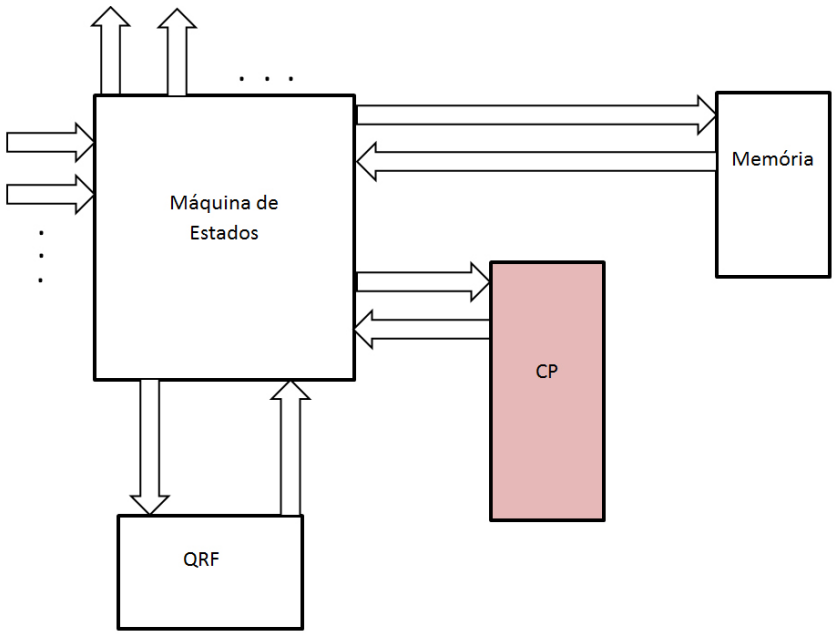

Figura 3.2: Diagrama de blocos da arquitetura proposta com o QRF e o CP.

3.2 Metodologia de projeto de *hardware* digital utilizado

A estrutura de *design*, utilizada neste projeto de pesquisa, foi baseada na estrutura do livro de Brown e Vranesic (2003), que é mostrada na Seção 2.6. A Figura 3.3 mostra essa estrutura, na qual o primeiro passo foi chegar às especificações da arquitetura analisando o problema e as funções que a arquitetura deveria desempenhar. Também, foram definidas as ferramentas e a linguagem de descrição de *hardware* que seriam utilizadas. A linguagem escolhida foi Verilog; a Seção 3.3 mostra mais detalhes da escolha da linguagem. O segundo passo foi analisar quais estruturas deveriam ser separadas em diferentes módulos em Verilog. Então, a arquitetura foi dividida nos seguintes módulos: máquinas de estados, QRF, CP e memória. Em cada um desses módulos foi feito um processo de simulação funcional, teste e correção. Quando ocorriam erros após a simulação funcional, o fluxo voltava para o projeto do módulo; os erros eram corrigidos e a simulação funcional era realizada novamente. Esse processo foi repetido até que os erros fossem eliminados. Após a eliminação dos erros, foram feitas as conexões entre os blocos. O processo de simulação funcional, teste e correção continuaram, porém, desta vez com todo o sistema interligado. Quando os erros eram entre a conexão dos blocos, o fluxo voltava para a seta *B*; quando os erros eram em um módulo específico, o fluxo voltava para a seta *A*, somente para o módulo com problema. Com a simulação funcional de todo o sistema sem erros, foram feitos testes na Placa DE-2. O processo de teste e correção também foram feitos nessa etapa, porém, direto na placa, em vez de realizar somente a simulação funcional. Com os testes na placa sem erros em relação à execução dos mesmos testes em *software*, considerou-se

o desenvolvimento de *hardware* validado.

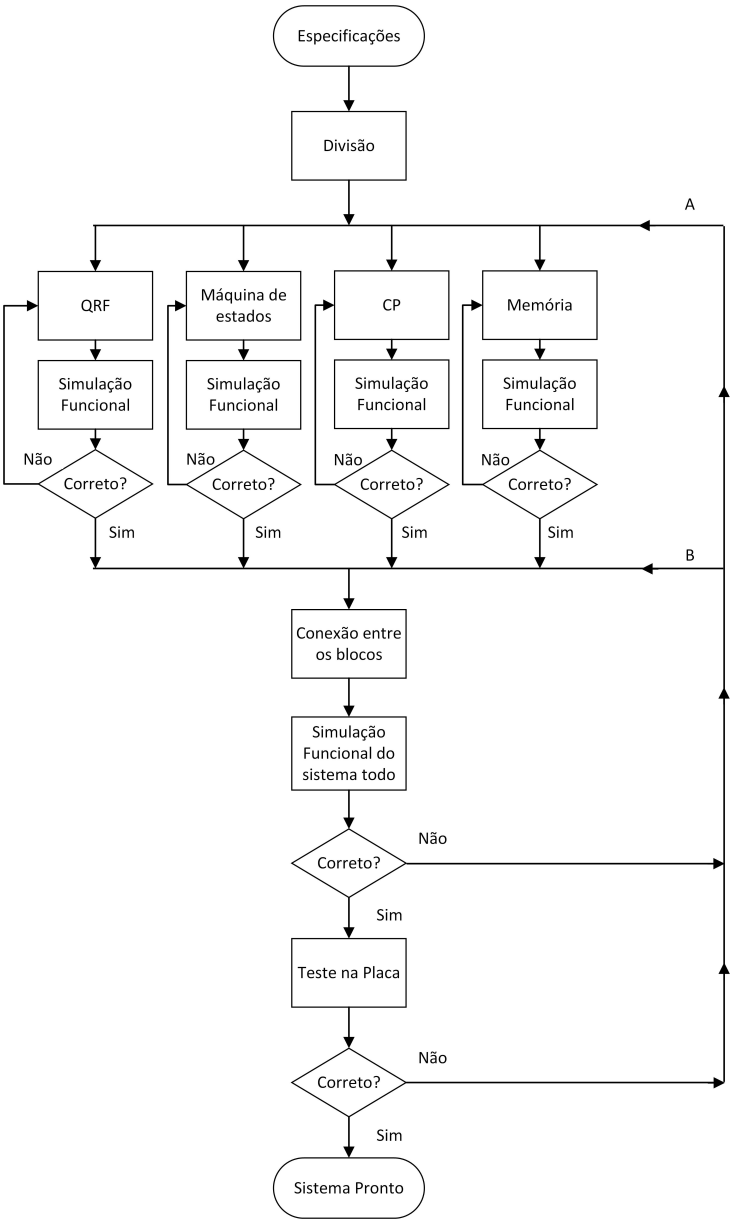

Figura 3.3: Estrutura de projeto utilizada.

3.3 Definição da Ferramenta para a descrição de *hardware*

Como foi dito na Seção anterior, a definição das ferramentas e da linguagem de descrição de *hardware* constituem uma etapa importante no desenvolvimento da arquitetura alvo. A linguagem de descrição de *hardware* escolhida para o desenvolvimento deste trabalho foi Verilog. Inicialmente, foram feitos testes utilizando a ferramenta Active-HDL da Aldec (ALDEC, 2014) para entender o conceito de máquinas de estados e Verilog através da prática e dos diagramas que a ferramenta oferece (mais detalhes no Apêndice C). Apesar de o Active-HDL ser uma

ótima ferramenta, foi decidido utilizar as ferramentas Quartus II e ModelSim-Altera: a primeira para a síntese, a simulação real e o mapeamento dos circuitos criados para a placa DE-2 da Altera e a segunda para simulação funcional e, também, síntese. Essa decisão foi tomada em virtude da necessidade de mapear a arquitetura para a placa através do Quartus II. Além disso, seria mais interessante utilizá-lo também para a programação e síntese, uma vez que essa ferramenta já proporciona as especificações da placa DE-2. Também foi decidido criar as máquinas de estados e todos os outros componentes direto em Verilog, em vez de, depender da ferramenta para gerar o código em Verilog. Todos os circuitos, simulações, formatos de onda e testes que serão mostrados foram feitos utilizando as ferramentas Quartus II e Modelsim.

3.4 QRF

Para implementar o arquivo de registradores QRF, primeiramente, foi necessário entender o conceito de arquivos de registradores e filas. O conceito de arquivos de registradores foi aprofundado através da leitura dos livros de Brown e Vranesic (2003) e Tocci, Widmer e Moss (2007). Para a implementação do QRF e melhor entendimento do conceito de Filas foram utilizados os livros de Goodrich e Tamassia (2007) e Chu (2008).

3.4.1 Filas

Uma fila é uma estrutura de dados que armazena uma coleção de valores que, para serem inseridos e removidos, seguem o princípio de que "o primeiro que entra é o primeiro que sai" ou FIFO(*First In, First Out*). Isso significa que os elementos podem ser inseridos a qualquer momento, mas somente o elemento que está há mais tempo na fila pode ser retirado. Uma fila possui uma coleção de valores que são mantidos em uma sequência, na qual o acesso aos valores e sua remoção são restritos ao primeiro elemento chamado de início ou cabeça da fila e a inserção de elementos é restrita ao fim da sequencia, que é chamada de fim ou calda da fila. Essa restrição garante a regra de que o primeiro que entra é o primeiro que sai (GOODRICH; TAMASSIA, 2007).

Uma fila deve possuir as seguintes funcionalidades fundamentais:

- *enqueue*: insere um valor no fim da fila.
- *dequeue*: retira e retorna o valor do fim da fila.

Algumas funcionalidades auxiliares:

- *size*: mostra o número de valores contidos na fila.
- *isEmpty*: bit que indica se fila está vazia.
- *front*: retorna, mas não remove, o valor do início da fila.

Uma forma de implementar uma fila é utilizando arranjos circulares de tamanho fixo. Um arranjo Q de tamanho fixo N é utilizado para armazenar os elementos que vão de $Q[0]$ a $Q[N-1]$. Duas variáveis f e r são definidas para fazer o controle do início e fim da fila, em que:

- f é um índice de um elemento do arranjo Q que guarda o primeiro elemento da fila (que na próxima operação dequeue será removido) , a não ser que a fila esteja vazia.
- r é um índice para a próxima posição livre em Q. Quando a fila está vazia, $f = r$.

No início $f = r = 0$, o que indica que a fila está vazia. Ao remover um elemento, f será incrementado para indicar a próxima célula. Da mesma maneira, ao inserir um elemento, o mesmo é armazenado em $Q[r]$ e incrementa-se r para indicar a próxima célula livre em Q. Para que o arranjo fique circular, cada vez que se incrementa f ou r, o cálculo desse incremento é feito como $(f+1) mod N$ ou $(r+1) mod N$, em que "*mod*"é o operador módulo. Se forem enfileirados N objetos em Q sem que nenhum seja retirado da fila, o resultado seria $f = r$, que é a mesma condição de quando a fila está vazia, ocasionando um erro. A solução para esse erro consiste em exigir que Q nunca contenha mais que $N-1$ elementos. Para calcular o tamanho da fila, pode ser utilizada a expressão $(N-f+r) mod N$, que fornece o tamanho correto tanto para $f <= r$ ou $r < f$ (GOODRICH; TAMASSIA, 2007). A Figura 3.4 mostra um exemplo de uma fila circular utilizando arranjos.

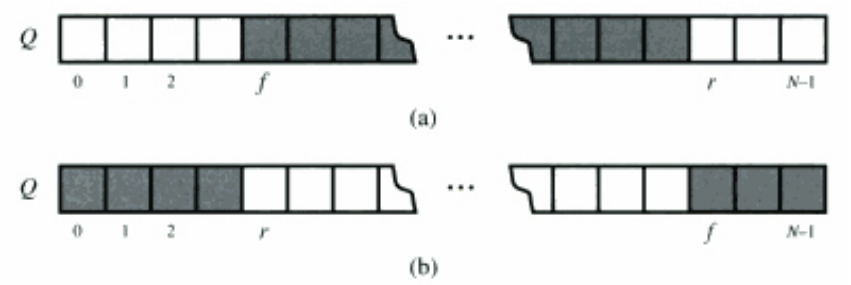

Figura 3.4: Fila utilizando arranjos circulares: (a) Caso em que $f <= r$. (b) Caso em que $r < f$. As posições armazenando elementos estão na cor cinza.

Fonte: Goodrich e Tamassia (2007).

3.4.2 Possíveis Implementações em *Hardware*

Uma fila em *hardware* pode ser implementada como um bloco em RAM com um endereço de escrita e um endereço de leitura independentes entre si. Para cada escrita na fila, o endereço de escrita é incrementado e para cada leitura na fila, o endereço de leitura é incrementado. O maior desafio ao fazer o design de uma fila em *hardware* é prevenir a sobrescrita e perda de dados durante a inserção de um novo valor ou remoção de um valor antigo (COFFMAN, 2000).

As implementações do QRF, que serão mostradas, foram realizadas a partir de filas circulares de tamanho fixo. O tamanho das filas pode ser fixo, pois o QRF, ao ser mapeado para a placa, não possui alterações em seu tamanho. A Figura 3.5 mostra uma visão em bloco do QRF que serve para as suas duas versões criadas.

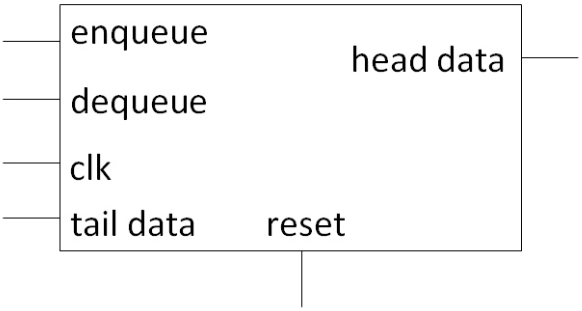

Figura 3.5: Visão em bloco do QRF.

A primeira versão em Verilog do QRF foi feita baseada nas especificações da Subseção anterior. A Listagem 3.1 mostra o código em Verilog dessa versão.

Listagem 3.1: Primeira versão do QRF.

```
1
2  module QRFVersao1(head_data, tail_data, clk, dequeue, enqueue);
3  //parametros que definem o tamanho da fila
4  parameter DATA_WIDTH = 32;
5  parameter REGFILE_WIDTH = 4;
6
7  //Declaração dos pinos
8  input wire enqueue,dequeue,clk;
9  input wire [DATA_WIDTH-1:0] tail_data;
10 output wire [DATA_WIDTH-1:0] head_data;
11
12 reg [REGFILE_WIDTH-1:0] head_address = 0;
13 reg [REGFILE_WIDTH-1:0] tail_address = 0;
14 reg [REGFILE_WIDTH-1:0] ra;
15
```

```
16 //Declaração da Memória
17 reg  [DATA_WIDTH - 1:0] Mem[0:REGFILE_WIDTH - 1];
18
19 //Mostra o tamanho do QRF
20 wire [REGFILE_WIDTH-1:0] size = (REGFILE_WIDTH - head_address +
       tail_address) % REGFILE_WIDTH ;
21
22 //Mostra se o QRF está vazio
23 wire null  = (tail_address == head_address);
24
25 //Lógica do QRF
26 always@(posedge  clk) begin
27     /*insere somente se a fila não estiver cheia*/
28     if((enqueue == 1) && (size!= REGFILE_WIDTH - 1 )) begin
29         /*realiza a inserção*/
30         Mem[tail_address] = tail_data;
31         /*incrementa o fim da fila*/
32         tail_address = (tail_address + 1) % REGFILE_WIDTH;
33         /*remove somente se a fila não estiver nula */
34     end else if((dequeue == 1) && ( null != 1 )) begin
35         /*incrementa o inicio da fila */
36         head_address = (head_address + 1) % REGFILE_WIDTH;
37     end
38     ra = head_address;
39 end
40
41 //Leitura do QRF
42 assign head_data = Mem[ra];
43
44 endmodule
```

Na Listagem 3.1, o componente que armazena os dados do QRF é uma memória que é declarada na linha 17. Os parâmetros *DATA_WIDTH* e *REGFILE_WIDTH* representam o tamanho dos dados armazenados e o tamanho do QRF, respectivamente. Esses parâmetros são declarados no início do código nas linhas 4 e 5 com os valores 32 e 4 como padrão. Uma vez que, o código é transferido para a FPGA, se mantêm fixos. O QRF possui como sinal de entrada os bits *enqueue*, *dequeue* e *clk*, e o valor para ser inserido na cauda da fila *tail_data*, que pode conter de 0 até *DATA_WIDTH-1 bits*. A saída será o sinal *head_data*, que também pode conter de 0 até *DATA_WIDTH-1 bits*. Os sinais *head_adress* e *tail_adress* mostram o início e o fim do QRF, e o sinal *ra* é apenas um sinal auxiliar para a leitura da memória. Na linha 20 é criado o sinal *size*, que mostra quantos valores existem no QRF, e na linha 22 é criado o sinal *null*, mostrando

se o QRF está vazio. Dentro do bloco *always* são feitas duas comparações, deixando ou não que os dados sejam inseridos no QRF, dependendo do valor dos sinais *enqueue*, *dequeue*, *size* e *null*. Na linha 42 é feita a leitura do QRF. Apesar de funcionar, esta primeira versão não trata a possibilidade dos *bits enqueue* e *dequeue* terem o valor 1 simultaneamente, e também faz o uso do operador %, que pode ser evitado.

A segunda versão do QRF foi baseada na fila do livro de Chu (2008). No Apêndice D é mostrada a fila em Verilog que foi utilizada como base, e também outra implementação do QRF. Essa versão é mostrada na Listagem 3.2.

Listagem 3.2: Segunda versão do QRF.

```
1  module QRFVersao2(head_data, tail_data, clk,reset, dequeue, enqueue);
2  parameter DATA_WIDTH = 8;
3  parameter REGFILE_WIDTH = 11;
4
5  //declarções dos pinos
6  input wire enqueue,dequeue,clk,reset;
7  input wire [DATA_WIDTH-1:0] tail_data;
8  output wire [DATA_WIDTH-1:0] head_data;
9  integer tamanho = 2**REGFILE_WIDTH;
10 reg [REGFILE_WIDTH-1:0] head_address;
11 reg [REGFILE_WIDTH-1:0] tail_address;
12 reg [REGFILE_WIDTH-1:0] plus = 1'b1;
13 reg [REGFILE_WIDTH-1:0] ra1;
14 reg  [DATA_WIDTH - 1:0] Mem[0:(2**REGFILE_WIDTH)- 1];
15 reg full  ;
16 reg empty  ;
17 integer i;
18
19 // lógica
20 always@(posedge  clk ) begin
21     if (reset)
22       begin
23         head_address =  {REGFILE_WIDTH{1'b0}};
24         tail_address =  {REGFILE_WIDTH{1'b0}};
25         ra1 = {REGFILE_WIDTH{1'b0}};
26         full = 1'b0;
27         empty = 1'b1;
28         end
29     else
30     begin
31     case ({enqueue,dequeue})
```

```
32          2'b01:
33            if(~empty)
34                begin
35                  head_address = head_address + plus;
36                  full = 1'b0;
37                if ( tail_address == head_address)
38                  empty = 1'b1;
39            end
40
41          2'b10:
42             if(~full)
43                begin
44                  Mem[tail_address] = tail_data;
45                  tail_address = tail_address + plus;
46                  empty = 1'b0;
47                if ( tail_address == head_address)
48                  full = 1'b1;
49            end
50
51          2'b11:
52          begin
53            if(~empty)
54              begin
55                head_address = head_address + plus;
56              end
57            Mem[tail_address] = tail_data;
58            tail_address = tail_address + plus;
59            empty = 1'b0;
60           end
61
62        endcase
63
64       ra1 = head_address;
65       end
66    end
67
68  // leitura da memória
69   assign head_data = Mem[ra1] ;
70
71  endmodule
```

Na segunda versão do QRF foram mantidas muitas características da primeira, como as

entradas, a saída, a memória, a forma de endereçamento da memória e a leitura. O que mudou foi o controle da inserção e remoção dos valores na memória feito utilizando-se a cláusula *case*, parecido com máquinas de estados. Na linha 31, os sinais *enqueue* e *dequeue* são concatenados e utilizados no *case*. Com os valores concatenados, as operações do QRF são divididas da seguinte forma: 01 é feita a remoção do início do QRF, 10 a inserção no fim do QRF, e 11 a inserção e remoção simultaneamente.

A lógica para o controle de inserção é a seguinte:

- Primeiramente, é verificado se o QRF está cheio.
- Se estiver cheio, a operação é finalizada.
- Se o QRF não estiver cheio, o valor é inserido e o endereço da cabeça é incrementado.
- O *bit* que sinaliza que o QRF está vazio recebe o valor 0.
- Após a inserção, é verificado se o QRF ficou cheio.
- Se estiver cheio, o bit que sinaliza que o QRF está cheio recebe o valor 1.

A lógica para o controle da remoção é análoga com a lógica da inserção:

- Primeiramente, é verificado se o QRF está vazio.
- Se estiver vazio, a operação é finalizada.
- Se o QRF não estiver vazio, é incrementado o endereço da cauda.
- O *bit* que sinaliza que o QRF está cheio recebe o valor 0.
- Após a remoção, é verificado se o QRF ficou vazio.
- Se estiver vazio, o *bit* que sinaliza que o QRF está vazio recebe o valor 1.

O operador % foi removido, pois, em *hardware*, após o contador chegar ao valor máximo, ele volta ao valor inicial devido ao *overflow*. Também foi inserida a possibilidade de adição e remoção simultânea na fila. A lógica para a inserção e remoção simultâneas é feita da seguinte forma:

- Primeiramente, é verificado se o QRF está vazio.

- Caso o QRF esteja vazio, não é necessário fazer remoção, somente inserção.
- Após a inserção, o *bit* que sinaliza que o QRF está vazio recebe o valor 0.

O QRF foi projetado para sempre mostrar o valor atual da cabeça da fila, ao contrário de outras implementações, que mostram o valor anterior, já removido. A remoção consiste apenas em atualizar o valor do endereço da cabeça do QRF. A leitura do valor da cabeça do QRF sempre é feita caso o endereço da cabeça mude. Na inserção, valores são inseridos na memória e depois o endereço da cauda é incrementado. Na inserção e remoção simultâneas, se o QRF possuir algum valor ou estiver cheio, os dois endereços, da cabeça e da cauda, são incrementados e o valor é inserido. Se o QRF estiver vazio durante a inserção e remoção simultâneas, não é feita remoção, apenas inserção, pois não é possível remover algum valor de uma fila vazia.

Além da segunda versão, foi criado outro QRF utilizando operadores de atribuição de bloqueio e não bloqueio simultaneamente. Essa terceira versão funciona da mesma forma que a segunda versão do QRF. O código em Verilog dessa versão é mostrado no Apêndice D. Também foram realizados testes com as filas já prontas do pacote da Altera, que também podem ser utilizadas como QRF. Ainda no Apêndice D é mostrado um exemplo de um teste feito com a fila da Altera. A segunda versão do QRF é que foi escolhida para ser utilizada na arquitetura alvo, por ser mais completa que a primeira versão. Não foram escolhidas as filas da Altera, pois é necessário que o código do QRF seja visível para que, em trabalhos futuros, mais funcionalidades possam ser inseridas ao QRF.

3.4.3 Testes com o QRF

Foram feitos vários testes para validar a estrutura do QRF. A Listagem 3.3 mostra o exemplo de um teste, no qual uma máquina de estados é utilizada para fazer operações simples de inserção e remoção no QRF.

Listagem 3.3: Máquina de estados para testar o QRF.

```
1
2 module TesteFilaUtilizada(clk);
3
4
5   parameter Estado1 = 0;
6   parameter Estado2 = 1;
7   parameter Estado3 = 2;
8   parameter Estado4 = 3;
9   parameter Estado5 = 4;
```

```
10    parameter Estado6 = 5;
11    parameter Estado7 = 6;
12    parameter Estado8 = 7;
13    parameter Estado9 = 8;
14    parameter Estado10 = 9;
15    parameter Estado11 = 10;
16    parameter Estado12 = 11;
17    parameter Estado13 = 12;
18    parameter Estado14 = 13;
19    parameter Estado15 = 14;
20    parameter Estado16 = 15;
21    parameter Estado17 = 16;
22
23
24  input clk;
25
26  reg  enqueue, dequeue;
27
28  wire [15:0] head_data;
29  reg [15:0] tail_data;
30
31    wire clk,enable, reset;
32    reg[7:0] estado;
33
34    wire  enqueue_out = enqueue;
35    wire  dequeue_out = dequeue;
36    wire [15:0] head_data_out = head_data;
37    wire [15:0] tail_data_out = tail_data;
38
39
40                  FilaPosedgeUtilizada RegFifo(
41                  .clk(clk),
42                  .reset(reset),
43                  .enqueue(enqueue_out),
44                  .dequeue(dequeue_out),
45                  .head_data(head_data),
46                        .tail_data(tail_data)
47                         );
48          defparam
49                  RegFifo.DATA_WIDTH = 16,
50             RegFifo.REGFILE_WIDTH = 2;
51
52          always @(posedge clk) begin
```

```
53          if (reset) begin
54            enqueue = 0 ;
55            dequeue = 0 ;
56            estado = Estado1;
57          end else begin
58            case (estado)
59
60              Estado1:
61              begin
62               enqueue = 1;
63               tail_data = 1;
64               estado = Estado2;
65              end
66
67              Estado2:
68              begin
69               enqueue = 0;
70               dequeue = 0;
71               estado = Estado3;
72              end
73
74               Estado3:
75              begin
76                enqueue = 1;
77                tail_data = 2;
78                estado = Estado4;
79              end
80
81              Estado4:
82              begin
83               enqueue = 0;
84               dequeue = 0;
85               estado = Estado5;
86              end
87
88              Estado5:
89              begin
90                 enqueue = 1;
91               tail_data = 3;
92               estado = Estado6;
93              end
94
95              Estado6:
```

```
96              begin
97               enqueue = 0;
98               dequeue = 0;
99               estado = Estado7;
100             end
101
102             Estado7:
103             begin
104                enqueue = 1;
105               tail_data = 4;
106               estado = Estado8;
107             end
108
109             Estado8:
110             begin
111              enqueue = 0;
112              dequeue = 0;
113              estado = Estado9;
114             end
115
116             Estado9:
117             begin
118              dequeue = 1;
119              tail_data = 1;
120              estado = Estado10;
121             end
122
123               Estado10:
124             begin
125              enqueue = 0;
126              dequeue = 0;
127              estado = Estado11;
128             end
129
130             Estado11:
131             begin
132                dequeue = 1;
133                tail_data = 6;
134               estado = Estado12;
135             end
136
137               Estado12:
138             begin
```

```
139                 dequeue = 0;
140                enqueue = 0;
141                estado = Estado13;
142               end
143
144               Estado13:
145               begin
146                 dequeue = 1;
147                 estado = Estado14;
148               end
149
150                 Estado14:
151               begin
152                enqueue = 0;
153                dequeue = 0;
154                estado = Estado15;
155               end
156
157               Estado15:
158               begin
159                 dequeue = 1;
160                tail_data = 8;
161                 estado = Estado16;
162               end
163
164                 Estado16:
165               begin
166                enqueue = 0;
167                dequeue = 0;
168                estado = Estado17;
169               end
170
171
172             endcase
173           end
174
175       end
176
177       endmodule
```

No teste, são inseridos os valores 1, 2, 3 e 4 em um QRF de tamanho 4, e após a inserção, os valores são removidos. Na Figura 3.6 são mostradas as operações que são executadas na

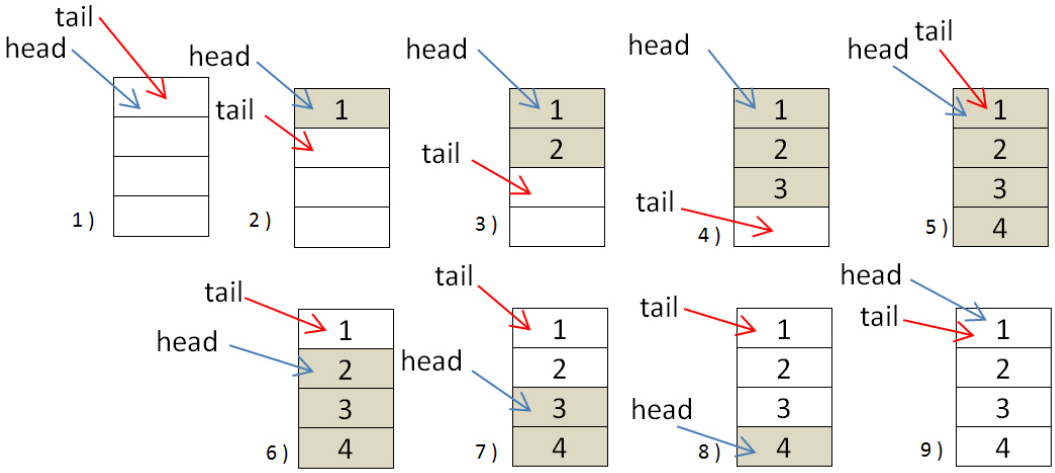

Figura 3.6: Operações executadas na Listagem3.3.

Listagem 3.3. O QRF inicia vazio como é mostrado na Figura 3.6(1), já na Figura 3.6(5) o QRF fica cheio e volta a ficar vazio na Figura 3.6(9). Os valores do QRF não são apagados, apenas sobrescritos. Na Listagem 3.3 foi deixado um estado inoperante entre cada operação, pois são exigidos dois ciclos de *clock* para cada operação do QRF. Para que ocorra a inserção, o sinal *enqueue* é ativado e um valor é inserido no registrador *tail_data*. É necessário aguardar um ciclo, para a operação de inserção terminar sua execução. O processo de inserção ocupa os estados de 1 a 8. Para que ocorra a operação de remoção, é necessário desativar o sinal *enqueue* e ativar o sinal *dequeue*, e o valor do próximo elemento irá aparecer em *head_data* após um ciclo. O processo de remoção ocupa os estados de 9 a 15. Com o sinal *enqueue* desativado, podem ser inseridos valores no registrador *tail_data* e não serão realizadas operações de inserção, como é mostrado nos estados 5, 7, 9 e 10. A Figura 3.7 mostra a simulação funcional da estrutura de teste da Listagem 3.3.

3.5 Módulo de Controle de Instruções Predicadas (CP)

O número de estados necessários para descrever o *loop pipelinig* em *hardware*, pode ser reduzido se for utilizada a técnica de controle de instruções predicadas, que é mostrada na Seção 2.4. Para utilizar a técnica de instruções predicadas junto ao *loop pipelinig*, foi necessário implementar um módulo para o controle das instruções predicadas. Foi dado o nome de Controle de Predicados (CP) para esse módulo. Registradores de deslocamento podem ser utilizados como base para a implementação do CP.

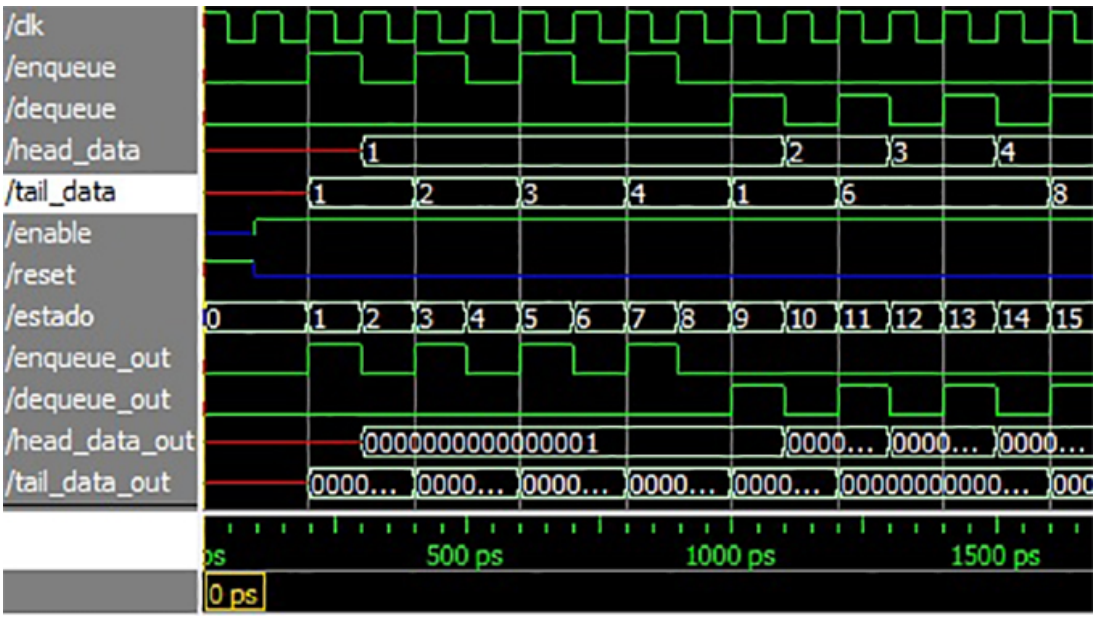

Figura 3.7: Formas de onda da Listagem 3.3.

3.5.1 Registrador de Deslocamento

Um registrador que fornece a capacidade de deslocar o seu conteúdo é chamado de registrador de deslocamento. A Figura 3.8 mostra um exemplo de um registrador de deslocamento de 4 bits que é utilizado para deslocar seu conteúdo 1-*bit* para a direita. Os bits são carregados dentro do registrador de deslocamento de forma serial, através da entrada *In*. O conteúdo de cada *flip-flop* é transferido para o próximo *flip-flop* a cada borda positiva do *clock*. A Tabela 3.1 mostra o que ocorre quando a entrada *In* assume os valores 1,0,1,1,1,0,0 e 0 em oito ciclos de *clock* consecutivos, assumindo que o valor inicial de cada *flip-flop* é 0 (BROWN; VRANESIC, 2003).

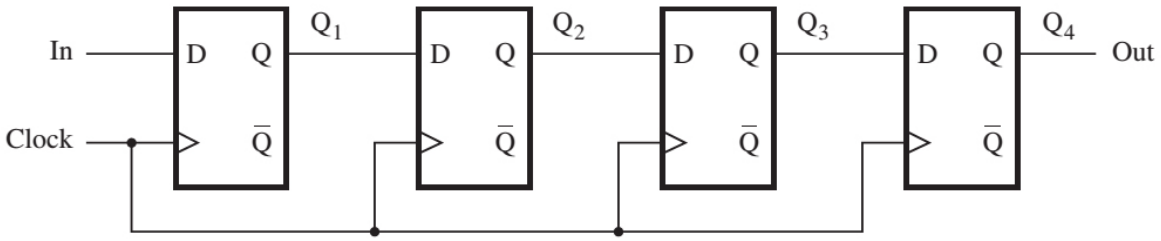

Figura 3.8: Circuito de um registrador de deslocamento.
Fonte: Brown e Vranesic (2003).

3.5.2 Implementação do CP

O CP tem os sinais de entrada: *si*, *enable* e *clock*, e tem somente uma saída, que é o sinal de n-*bits dout*. Na Figura 3.9 é mostrado o bloco CP. O sinal *enable* foi adicionado, pois é

	In	Q1	Q2	Q3	Q4=out
t0	1	0	0	0	0
t1	0	1	0	0	0
t2	1	0	1	0	0
t3	1	1	0	1	0
t4	1	1	1	0	1
t5	0	1	1	1	0
t6	0	0	1	1	1
t7	0	0	0	1	1

Tabela 3.1: Funcionamento do circuito da Figura 3.8 em relação ao tempo *t*.
Fonte : Brown e Vranesic (2003)

necessário que o CP deixe de deslocar seus valores durante alguns ciclos de *clock*. O sinal *si* funciona como o sinal *In* da Figura 3.8.

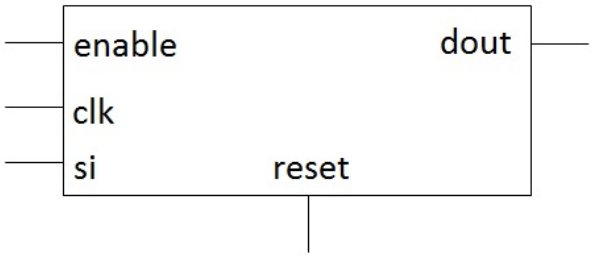

Figura 3.9: Bloco ilustrando o CP.

A Listagem 3.4 mostra uma possível implementação do CP em Verilog e a Figura 3.10 mostra a simulação funcional do CP em formas de onda. Na simulação, inicialmente, o CP possui os valores 0000. Um ciclo depois que os sinais *enable* e *si* são ativados, o valor 1 é inserido no primeiro *flip-flop* e o CP assume o valor 0001. Como o *si* e o *enable* continuam ativos, o deslocamento continua até que o valor do CP seja 1111.

Listagem 3.4: CP em Verilog.

```
1  module RegFileShift3 (clock, reset, enable, si, dout);
2  parameter SHIFT_WIDTH = 4;
3  input             clock;
4  input              reset;
5  input             enable;
6  input             si;
7  output [SHIFT_WIDTH-1:0]     dout;
8  reg    [SHIFT_WIDTH-1:0]     dout;
9
10         always @(negedge clock)
11                 if(reset)
12                 begin
```

```
13                      dout <=0;
14                    end
15                    else
16                      if (enable)
17                      dout <= {dout[SHIFT_WIDTH-2:0], si};
18
19  endmodule
```

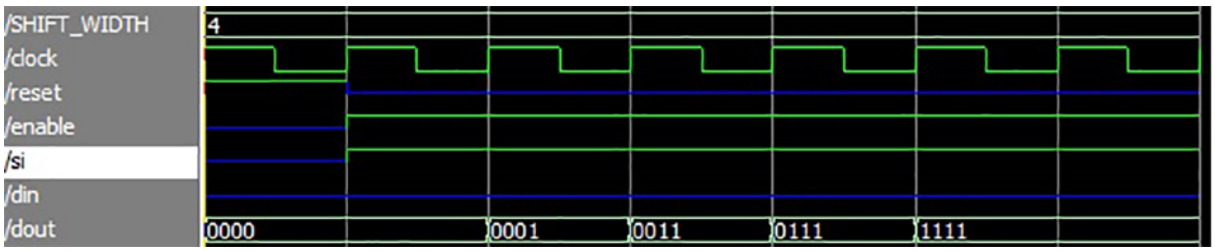

Figura 3.10: Simulação funcional do CP.

3.6 Arquitetura Alvo

A estrutura adotada para criar as máquinas de estados que irão controlar as operações paralelizadas geradas pelo *loop pipelining* será chamada de arquitetura alvo. O objetivo da arquitetura alvo é facilitar a transformação de *software* em *hardware*. O desenvolvimento da arquitetura alvo foi um processo de busca da melhor forma de unir *hardware* e *software*, semelhante às técnicas de *Codesign* de *Hardware/Software*. O *Codesign* de *Hardware/Software* investiga o projeto simultâneo de componentes de sistemas eletrônicos complexos que utilizam *hardware* e *software*. Seu intuito é otimizar a sinergia entre *hardware* e *software* (TEICH, 2012). Como foi mostrado na Seção 3.1, foram criados dois tipos de abordagens que foram chamadas de arquiteturas alvo: a Arquitetura Alvo 1, que não utiliza o CP, e a Arquitetura Alvo 2 que utiliza o CP. Essas arquiteturas foram baseadas nas máquinas de estados citadas na Seção A.5 e nas duas saídas do compilador Cetus Modificado (*FSM-Target Machine 1* e *FSM-Target Machine 2*). A estrutura adotada para criar as máquinas de estado não são voltadas para nenhuma das duas saídas do compilador Cetus Modificado em especifico. Isso se deu graças às memórias, ao QRF e ao CP adicionados às máquinas de estados, o que criou restrições que devem ser respeitadas e que foram consideradas na criação da arquitetura alvo. Um fator importante para a criação da arquitetura foi decidir qual tipo de memória utilizar, o que será explicado na próxima subseção.

3.6.1 Tipos de Memórias

O estudo de memórias RAM em Verilog foi necessário para a definição da arquitetura alvo. Existem várias formas de criar uma memória em Verilog. Uma das formas é criar a memória à mão, como é mostrado na Listagem 3.5. Outra forma é utilizar a memória *altsyncram*, já pronta, da Altera. Existem duas formas de inserir as memórias da Altera ao projeto. Uma delas é através do MegaWizard Plug-In Manager do Quartus II. A outra forma, é inserir a memória direto ao código através da chamada de módulos e configurando os parâmetros, como é mostrado na Listagem 3.6. As memórias mostradas nas duas listagens possuem apenas uma porta para leitura e escrita de dados. Foram consideradas as duas abordagens, mas a memória *altsyncram* da Altera é que foi escolhida. A memória *altsyncram* foi a melhor escolha por possuir vários parâmetros podendo ser modificada de uma porta para duas portas com facilidade, ou podendo ser configurada para inicializar seus dados com o conteúdo de um arquivo *.mif(Memory Initialization File)*. No Apêndice F são mostrados dois exemplos de arquivos *mif*. Outro fator para a escolha da memória *altsyncram* foi ela ter sido produzida pela própria Altera, o que a torna mais eficiente que a memória criada na Listagem 3.6. Uma desvantagem, é que a memória *altsyncram* da Altera não possui a função de múltiplas portas caso seja necessário, limitando assim a exploração do paralelismos em nível de instruções.

Listagem 3.5: Memória RAM de uma porta em Verilog.

```
1  module MemoriaRAM (clk, address, data, we);
2
3  parameter DATA = 8 ;
4  parameter ADDR = 8 ;
5  parameter DEPTH = 1 << ADDR;
6
7  //---------------- portas ---------------------------
8  input clk ;// Clock
9  input [ADDR-1:0] address      ;// EndereÃ§o de entrada
10 input             we          ;// Write Enable/Read Enable
11 inout [DATA-1:0] data        ;// Data bi-direcional
12
13 //----------------Variaveis Internas -----------------
14 reg [DATA-1:0] data_out ;
15 reg [DATA-1:0] mem [0:DEPTH-1];
16
17
18 // Controle de leitura tri-state
19 // quando we = 0 a saÃda recebe mem[address]
20 assign data = !we ? mem[address] : {DATA{1'bz}};
```

```
21
22 //escrita
23 always @ (posedge clk)
24 begin
25     if (we) begin
26         mem[address] = data;
27     end
28 end
29
30
31 endmodule
```

Listagem 3.6: chamada do módulo de Memória RAM *altsyncram* da Altera.

```
1          input clock;
2          output [15:0]register_A, memory_data_register_out;
3          reg [15:0]register_A;
4          reg [7:0]memory_address_register;
5          reg memory_write;
6          wire [15:0]memory_data_register;
7          wire [15:0]memory_data_register_out = memory_data_register;
8          wire [15:0]memory_address_register_out = memory_address_register;
9          wire memory_write_out = memory_write;
10
11         altsyncram altsyncram_component (
12                 .wren_a(memory_write_out),
13                 .clock0(clock),
14                 .address_a(memory_address_register_out),
15                 .data_a(register_A),
16                 .q_a(memory_data_register)
17                 );
18
19         defparam
20                 altsyncram_component.operation_mode="SINGLE_PORT",
21                 altsyncram_component.width_a = 16,
22                 altsyncram_component.widthad_a = 8,
23                 altsyncram_component.outdata_reg_a="UNREGISTERED",
24                 altsyncram_component.lpm_type = "altsyncram",
25                 altsyncram_component.init_file ="program.mif",
26                 altsyncram_component.intended_device_family="Cyclone";
```

Na Listagem 3.5 o tamanho da memória é determinado pelos parâmetros *DATA* e *ADDR*. Na memória *altsyncram* o tamanho é determinado pelos parâmetros *width_a* e *widthad_a*, e

o parâmetro *operation_mode* determina se a memória terá uma ou duas portas. Não foram mostrados todos os sinais de entrada e saída da memória *altsyncram* na Listagem 3.6, mas somente os principais sinais, que são:

- *wren_a*: quando igual a 1 habilita a escrita.
- *clock0*: *clock* da memória.
- *address_a*: controla o endereço da memória tanto para a escrita quanto para a leitura.
- *data_a*: registrador utilizado para a escrita.
- *q_a*: registrador utilizado para a leitura.

A memória *altsyncram* possui duas portas (porta A e porta B) com suas respectivas saídas e entradas de dados, as quais podem ser utilizadas para leitura ou escrita dependendo do modo de configuração de memória escolhido. Em uma memória de apenas uma porta a leitura e escrita compartilham o mesmo endereço da porta A. A Figura 3.11 mostra o diagrama de uma memória com uma porta (ALTERA, 2014b).

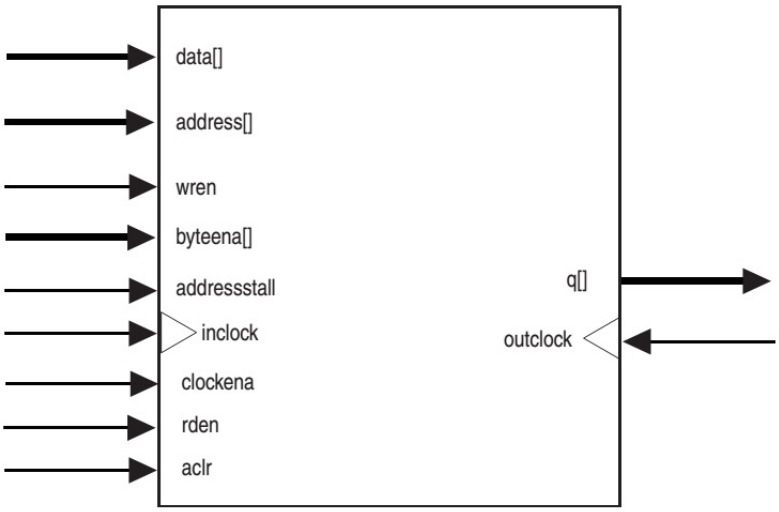

Figura 3.11: Memória RAM de uma porta.
Fonte:Altera (2014b).

A memória de duas portas simples é outro modo de utilizar a memória *altsyncram*. Neste caso, um endereço de memória é dedicado para leitura e um para escrita. A operação de escrita utiliza o endereço de escrita da porta A enquanto a operação de leitura utiliza o endereço de leitura e a saída da porta B. A Figura 3.12 mostra o diagrama de uma memória com duas portas simples (ALTERA, 2014b).

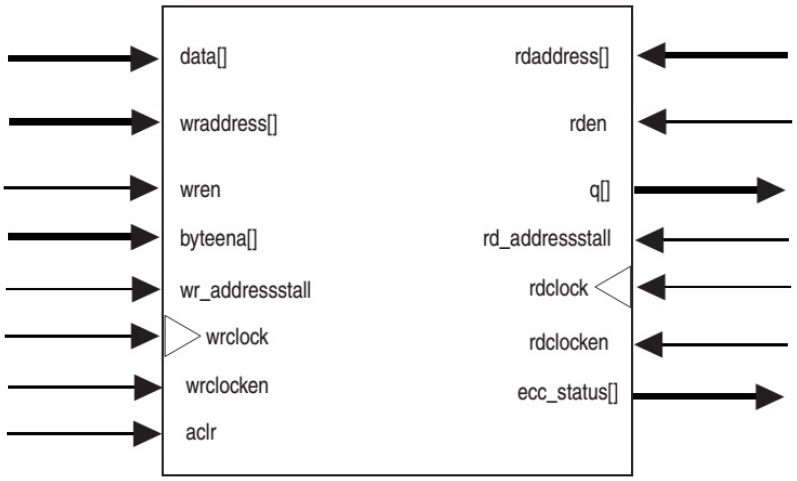

Figura 3.12: Memória RAM de duas portas simples.
Fonte:Altera (2014b).

Um outro modo, ainda, de utilizar a memória *altsyncram*, é o modo de duas portas verdadeiras ou simplesmente duas portas. Nesse modo, dois endereços estão disponíveis tanto para a leitura quanto para a escrita. A memória de duas portas permite escrever e ler utilizando os endereços das portas A ou B. A Figura 3.13 mostra o diagrama de uma memória com duas portas (ALTERA, 2014b).

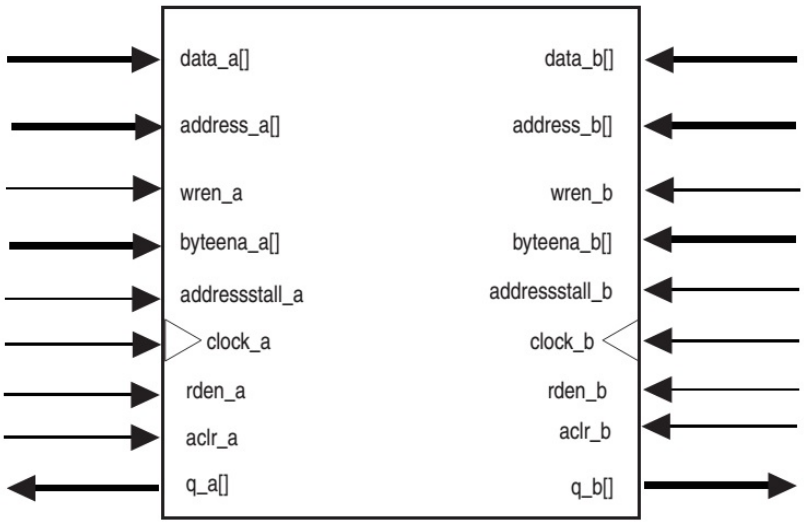

Figura 3.13: Memória RAM de duas portas.
Fonte:Altera (2014b).

3.6.2 Testes com Memórias

Para descobrir qual memória utilizar, inicialmente, foram feitos testes com a memória *altsyncram* utilizando as três configurações diferentes da memória sem nenhum *loop*. Foi verificado que, dependendo da configuração, as memórias podem levar dois ciclos de *clock* para fazer

leitura ou escrita, ou três ciclos, se forem configuradas como memórias registradas. Também foi constatado que, em aplicações simples, a memória com uma porta funciona perfeitamente. Contudo se a complexidade da aplicação cresce e são necessários acessos simultâneos à memória, a melhor escolha são as memórias de duas portas. O próximo passo foi realizar testes com as memórias sendo utilizadas em um *loop* sem *loop pipelining*, que foi implementado em Verilog utilizando máquinas de estados. Foram feitos testes com o *loop* utilizando os três tipos de memória. Nesses testes com *loop* sem *loop pipelining* a memória com duas portas não oferece nenhuma vantagem sobre a memória com uma porta. Foram necessários 17 ciclos para uma execução do *loop*, como é mostrado na Figura 3.14. Os códigos utilizados nos testes estão no Apêndice E.

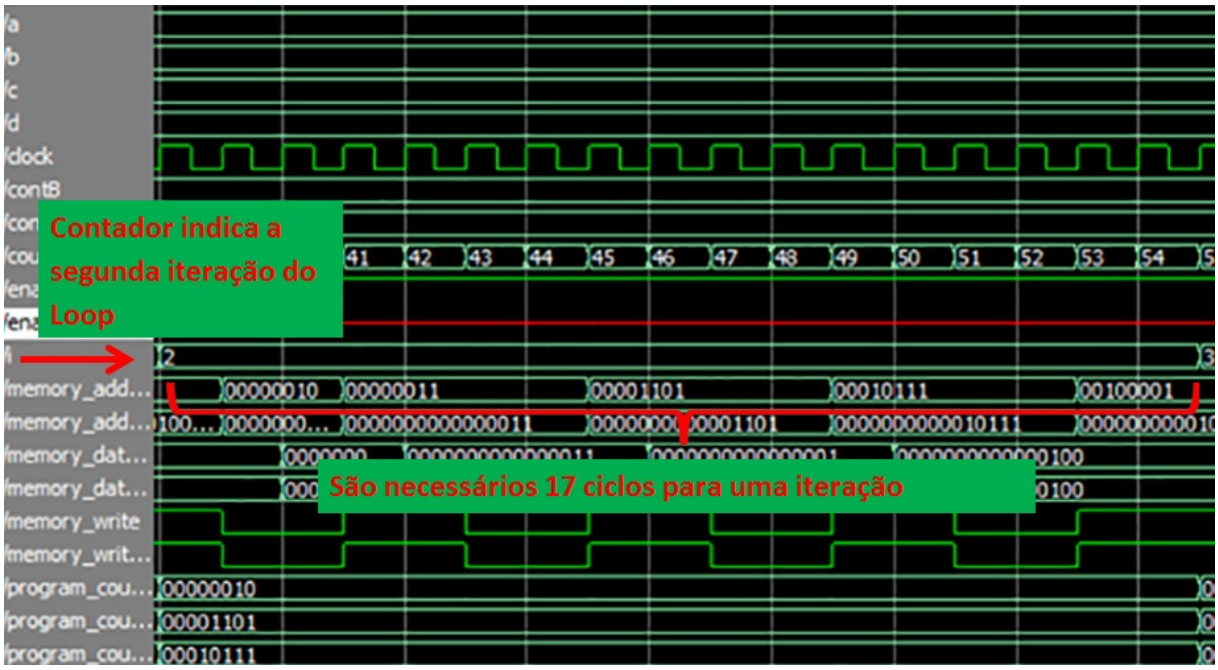

Figura 3.14: Quantidade de ciclos de uma iteração do *loop* sem *loop pipelining*.

O próximo passo foi utilizar as memórias no *loop pipeline*, em que a memória com duas portas tanto de entrada quanto de saída (Figura 3.13) foi utilizada na paralelização do código. Foram realizados dois tipos de teste: um em que os vetores paralelizados são armazenados na mesma memória, sendo que cada vetor possui seu endereço base e seu contador de programa, e outro em que cada vetor possui sua própria memória. No primeiro teste com *loop pipeline* com uma memória, foram necessários muitos estados para a execução do *kernel*, e foram executadas, no máximo, duas instruções por ciclo, por estar sendo utilizada apenas uma memória. Como pode ser visto na Figura 3.15, com uma memória, o *loop pipeline* precisa de 8 ciclos para uma execução do *kernel*.

Para aumentar o grau de paralelização do *loop pipelining*, foi feito o segundo teste, em

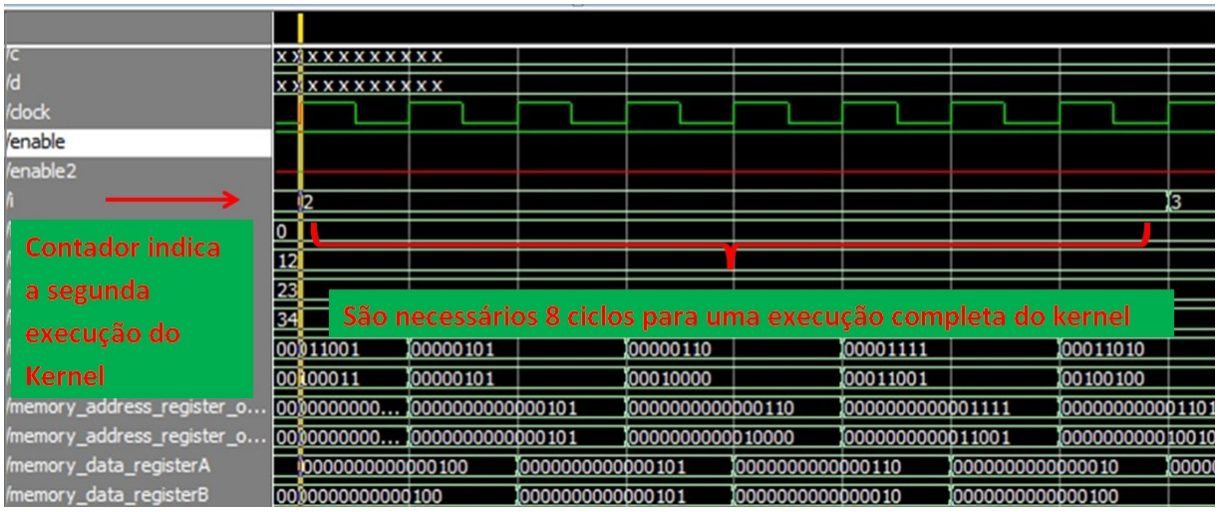

Figura 3.15: Quantidade de ciclos de uma execução do *kernel* com uma memória.

que foram utilizadas várias memórias, todas com duas portas de entrada e saída; porém, foi verificado que nem todas as memórias utilizaram as duas portas. Uma otimização que poderia ser realizada seria a utilização da memória de duas portas quando necessário e, quando as duas portas não forem utilizadas, colocar a memória de uma porta. Na Figura 3.16 é mostrado que, com a adição de múltiplas memórias, o *kernel* foi executado em quatro ciclos, quatro a menos que no teste anterior, comprovando que a utilização de múltiplas memórias torna a paralelização mais eficiente. Após os testes, foi definido que, na arquitetura alvo serão utilizadas várias memórias de duas portas tanto de leitura quanto de escrita. A memória escolhida é mostrada na Figura 3.13.

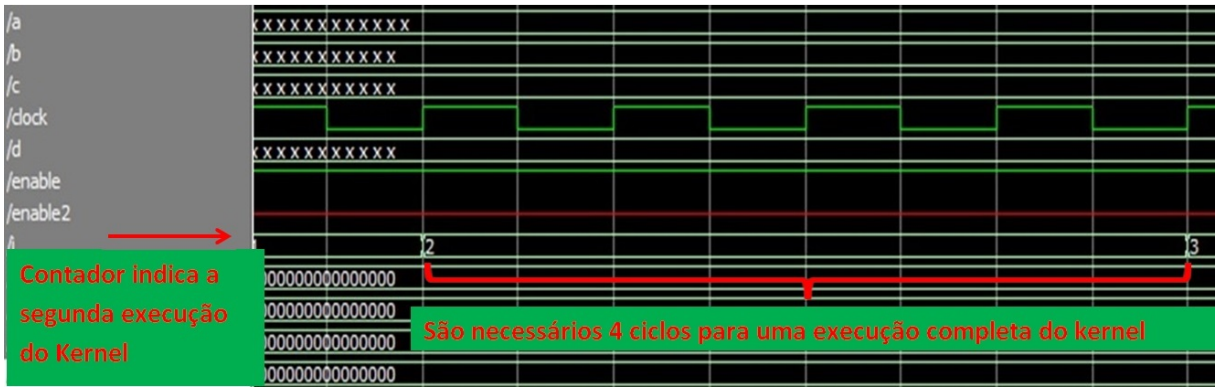

Figura 3.16: Quantidade de ciclos de uma execução do *kernel* com múltiplas memórias.

3.6.3 Máquinas de estados propostas

Pode se dizer que a arquitetura alvo é o modo como os componentes QRF, CP, máquina de estados são interligados entre si. A comunicação entre as máquinas de estados e os outros

componentes das arquiteturas propostas é mostrada pelo diagrama de blocos da Figura 3.17, que representa as duas arquiteturas propostas. A Arquitetura Alvo 1 não utiliza o CP que está em vermelho na figura; já a Arquitetura Alvo 2 faz uso do CP. As setas na cor verde representam entradas e saídas que as máquinas de estados podem possuir dependendo do código paralelizado, como valores utilizados nos cálculos e resultados da execução do código. A máquina de estados também pode possuir duas entradas, *n* e *i*, sendo *n* a quantidade de iterações executadas no *kernel* e *i* o valor inicial do contador. Como as memórias utilizadas são de duas portas, as conexões entre a máquina de estados e a memória devem ser feitas nas duas portas, A e B, da memória, o que é representado pelo A e B entre parênteses junto ao nome do sinal. As setas grossas na cor cinza representam vários *bits* e as setas finas, apenas 1 *bit*.

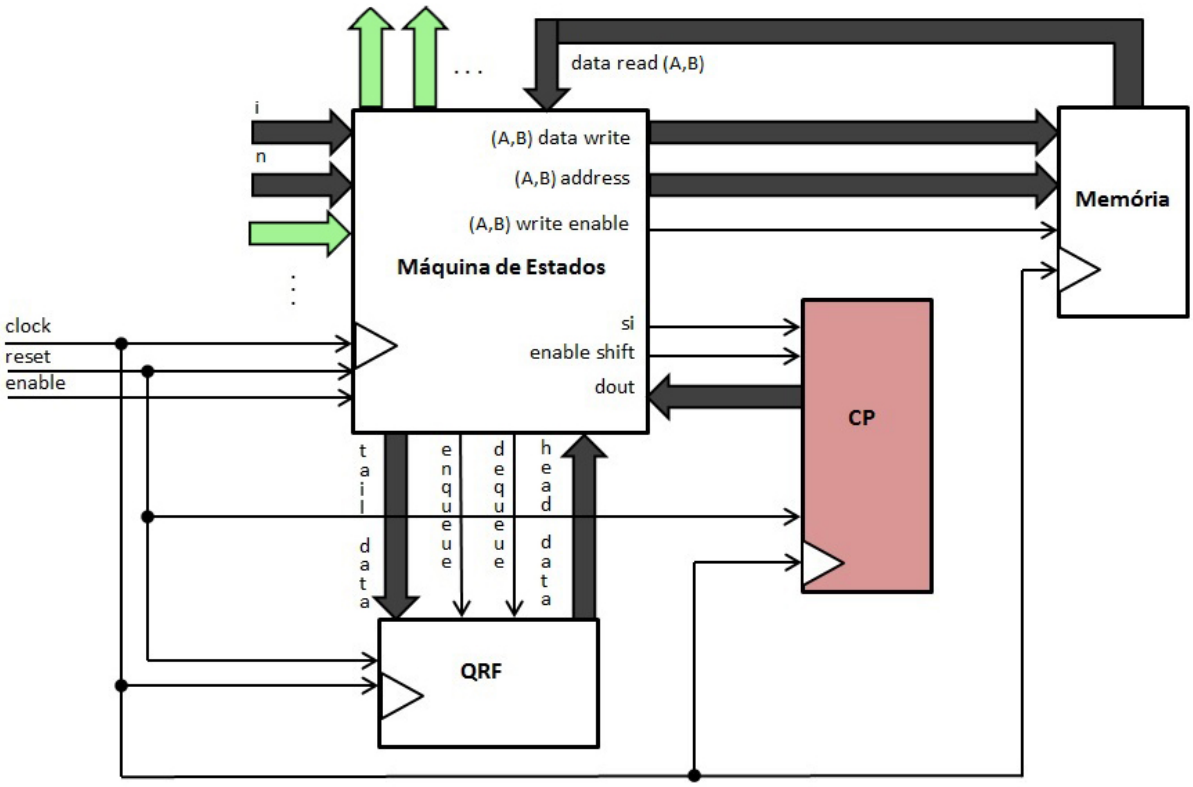

Figura 3.17: Diagrama de blocos da arquitetura proposta.

A Arquitetura Alvo 1 pode possuir uma ou várias memórias e QRFs. A Arquitetura Alvo 2 também possui várias memórias e QRFs, mas apenas um CP, que é o suficiente para fazer o controle das instruções predicadas. As duas arquiteturas também podem possuir vários multiplicadores , somadores ,subtratores e divisores. Na Figura 3.18 é ilustrada essa ideia de que a máquina de estados pode estar conectada a vários componentes simultaneamente. O número de QRFs é determinado pelo número de variáveis com o problema de sobreposição de valores, o número de memórias é determinado pelo número de vetores que existem no código e o número de operadores é determinado pelo número de operações que ocorrem no código.

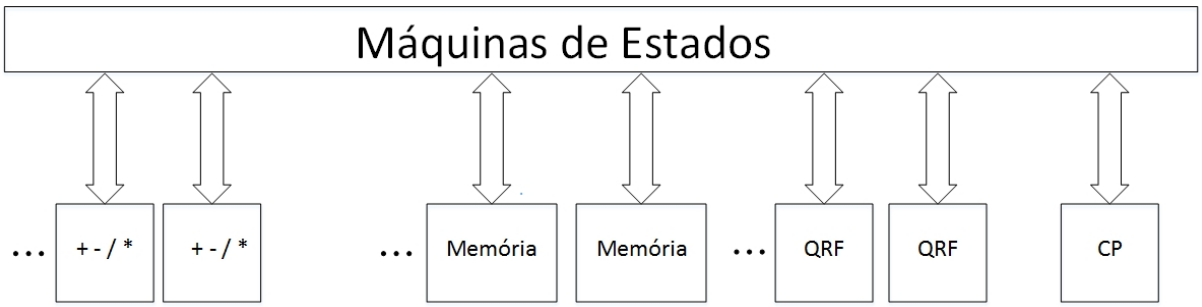

Figura 3.18: Diagrama de blocos que mostra a arquitetura alvo conectada a vários componentes.

3.6.4 Mecanismo de *clock* adotado

Após escolher o tipo de memória, foi decidido o tipo de *clock* que seria utilizado nas máquinas de estados e nos outros componentes. Foram testadas três estratégias para o uso do *clock*: bordas de *clock* alternadas, bordas de *clock* positivas e dois *clocks* simultâneos.

A estratégia de utilizar bordas de *clock* alternadas foi a primeira a ser testada. Ela consiste em utilizar tanto a borda de subida quanto a borda de descida do *clock*. Nesse teste, o QRF e o CP utilizam a borda de descida, e a máquina de estados e as memórias utilizam a borda de subida. A Figura 3.19 ilustra a estratégia de bordas de *clock* alternadas. Ao comparar o *loop pipelining* utilizando essa estratégia com um *loop* sem paralelismo, foi constatado que essa estratégia diminuiu consideravelmente o número de ciclos de *clock* para a execução do *loop pipelining*, porém, também diminui consideravelmente a frequência máxima do *clock*. Esses resultados e comparações serão mostrados no Capítulo 4.

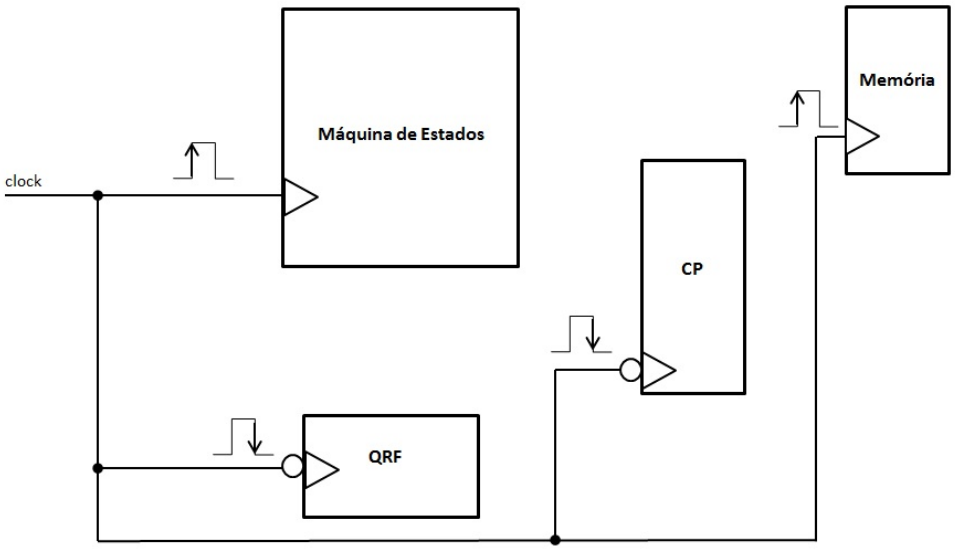

Figura 3.19: Arquitetura alvo utilizando bordas de *clock* alternadas.

A estratégia de utilizar bordas de *clock* positivas foi a segunda estratégia testada. Nela

todos os componentes utilizam a borda positiva do *clock*. A Figura 3.20 ilustra essa estratégia. Comparando o *loop pipelining* utilizando essa estratégia com um *loop* sem paralelismo, foi notado que o número de ciclos diminui , mas não tanto quanto a estratégia de bordas de *clock* alternadas. Por outro lado, a frequência, apesar de diminuir um pouco, não sofreu uma queda tão grande, se comparada à estratégia de bordas de *clock* alternadas.

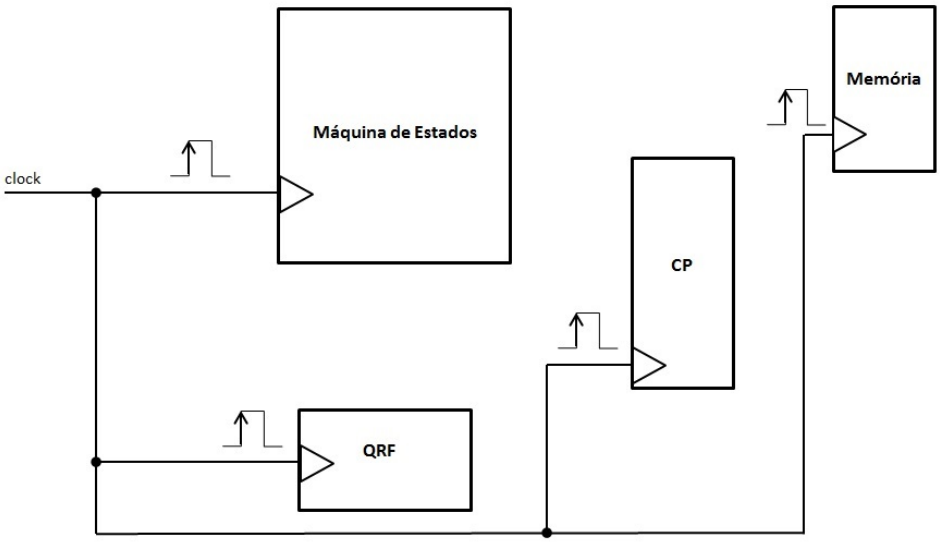

Figura 3.20: Arquitetura alvo utilizando bordas de *clock* positivas.

A estratégia de utilizar dois *clocks* foi a última a ser testada. Nessa estratégia, a frequência do *clock* principal é dividida pela metade e são gerados dois sinais de *clock*. O *clock* com a frequência menor é utilizado na máquina de estados e o *clock* com a frequência maior é utilizado nos outros componentes. Essa técnica se mostrou melhor que as outras duas, mas não foi testada o suficiente por falta de tempo. A técnica está sendo ilustrada na Figura 3.21.

Ao implementar essas estratégias em Verilog, a diferença entre cada uma delas está, principalmente, na forma em que são criadas as máquinas de estados. Na estratégia em que se utilizam bordas de *clock* positivas, é necessário adicionar um estado de *delay* à máquina de estados quando são utilizados o QRF, o CP e a memória. Isso ocorre porque são necessários dois ciclos para que cada um desses componentes complete uma operação. Ao utilizar a técnica de bordas de *clock* alternadas, é necessário adicionar um estado de *delay* apenas quando é utilizada a memória, pois esta também utiliza a borda de subida do *clock*. Na técnica em que são utilizado dois *clocks*, não são necessários estados de *delay*, mas é necessário dividir o *clock* em duas frequências diferentes e adicionar um controle extra do *clock* dentro do CP e outro dentro do QRF. O uso de bordas de *clock* positivas foi escolhido como a técnica principal, pois mostrou um melhor desempenho.

Após testar estratégias de *clock*, foram testados dois estilos de codificação de máquinas de

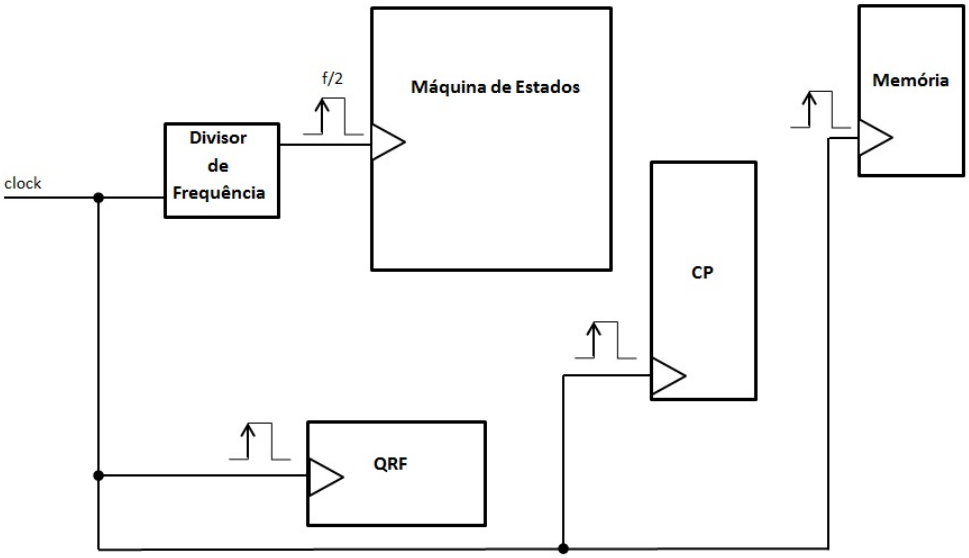

Figura 3.21: Arquitetura alvo utilizando dois *clocks*.

estados. As máquinas de estados foram desenvolvidas seguindo a ideia de máquinas de *Mealy*. O uso de dois blocos *always* para montar a máquina de estados, semelhante à Listagem A.12 do Apêndice A, foi o primeiro estilo testado. Nesse estilo, um *always* é responsável pelo circuito sequencial e outro pelo circuito combinacional. O outro estilo utilizado é semelhante ao da Listagem A.13 do Apêndice A, no qual é utilizado apenas um bloco *always*. Os dois estilos funcionaram, mas o segundo, que utiliza apenas um *always*, foi escolhido por ser mais simples de ser codificado. Ressaltando que os dois estilos geram o mesmo *hardware* e são apenas estilos de codificação. No Apêndice F estão alguns dos códigos utilizados nesses testes.

A escolha de como seriam codificadas as instruções paralelizadas pelo Cetus modificado em Verilog também foi um fator importante para a criação das máquinas de estados. As instruções de *Load* e *Store* tornaram-se acessos à memória *altsyncram* da Altera. Para as operações de soma, subtração, multiplicação e divisão, foram utilizados os operadores $+$,$-$,$*$ e $/$ em Verilog. Uma outra alternativa para fazer essas operações seria instanciar módulos de operações aritimética já prontos da Altera para realizar essas operações, da mesma forma como é feito com a memória *altsyncram*. Os operadores $+$ e $-$ em Verilog funcionaram normalmente. Já ao utilizar o operador $*$ em Verilog, o Quartus II automaticamente mapeia a multiplicação para circuitos dedicados do *chip* FPGA. Utilizar circuitos dedicados para a multiplicação não é um problema, entretanto, como existem apenas 70 destes no dispositivo utilizado, caso sejam necessários mais multiplicadores que do os 70, é aconselhado configurar e utilizar o módulo *lpm_mult* da Altera. O operador de divisão $/$, também funciona, mas utiliza muitos recursos do FPGA.

3.6.5 Testes iniciais com a arquitetura alvo adotada

Para compreender as duas arquiteturas criadas e para visualizar como as máquinas de estados são utilizadas junto aos componentes criados, será mostrado um exemplo prático. O exemplo é ilustrado na Figura 3.22, na qual um bloco de operações é paralelizado supondo que cada operação é executada em um único ciclo de *clock*. Na Figura 3.22, o bloco de operações (Figura 3.22(a)) representa o algoritmo Dotprod modificado, o qual é reordenado de modo que instruções pertencentes a iterações distintas executem simultaneamente (Figura 3.22(b)). A Figura 3.22(b) também mostra o problema de sobreposição de valores que ocorre no *loop pipelining*. O problema ocorre quando a instrução A2 precisa utilizar o valor de E calculado em A1 na primeira iteração (seta 1), mas o valor em E é da terceira iteração (seta 2) devido à sobreposição de valores. A Figura 3.23 mostra como o QRF resolve o problema de sobreposição ilustrado na Figura 3.22, em que o valor calculado em A1 é armazenado no QRF e depois consumido.

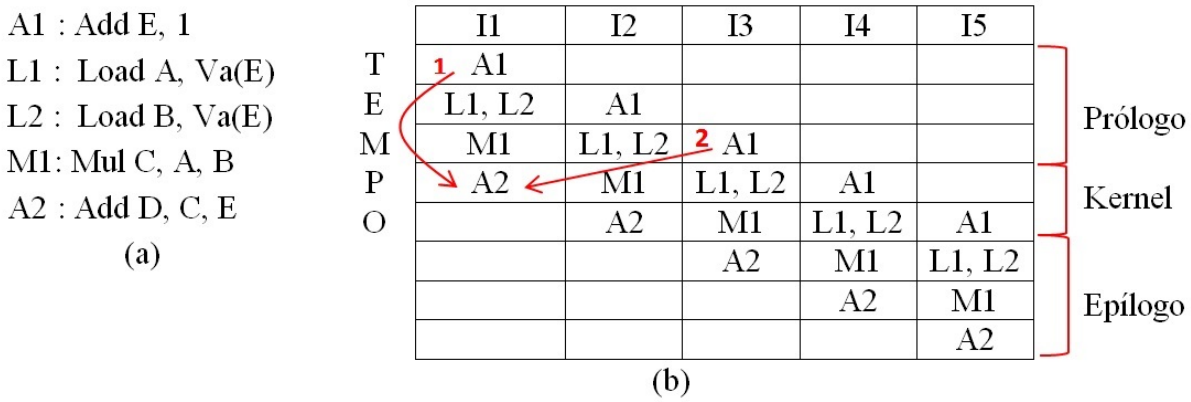

Figura 3.22: (a) Bloco de operações. (b) Escalonamento da execução de iterações, onde I representa as iterações.

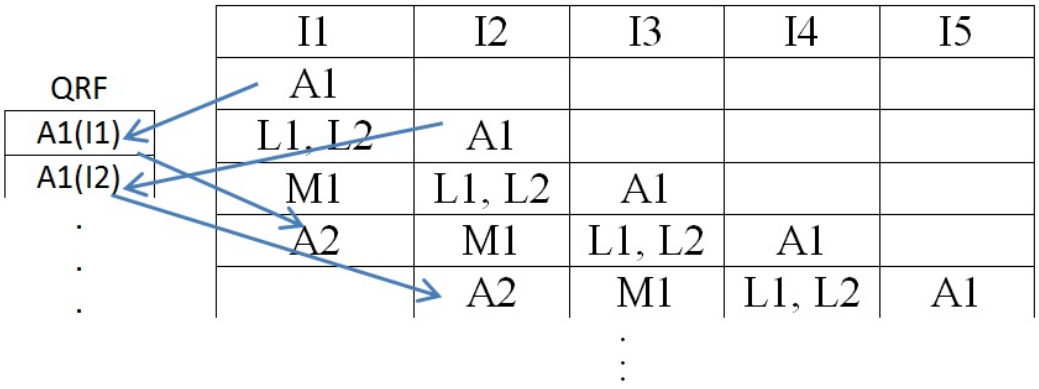

Figura 3.23: QRF Armazenando valores não consumidos do escalonamento da Figura 3.22.

Na Figura 3.24 é mostrada a máquina de estados que executa o *loop pipelining* da Figura 3.22.

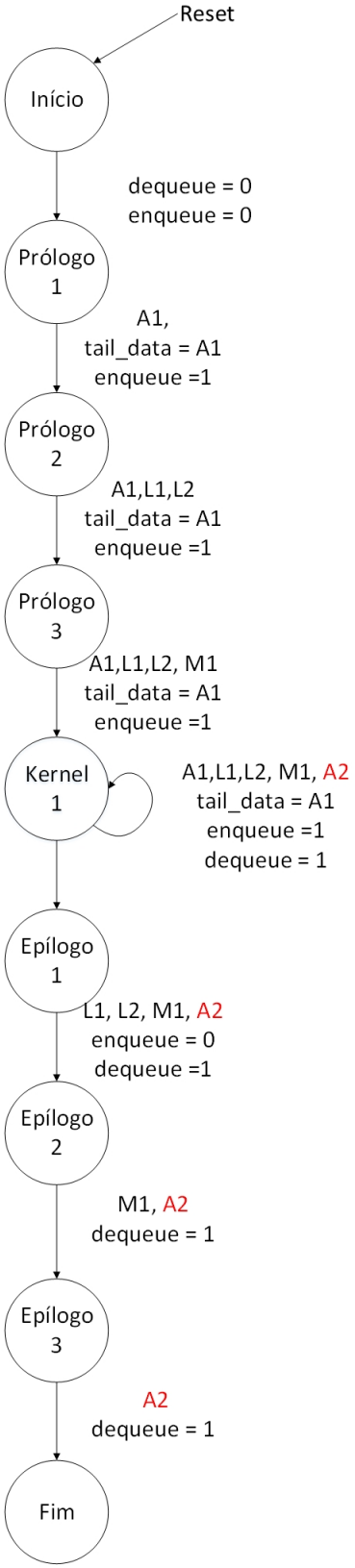

Figura 3.24: Máquina de estados utilizando o QRF (Arquitetura Alvo 1).

Essa máquina de estados representa a arquitetura alvo que utiliza apenas o QRF. Não foram inseridos os estados de *delay*,caso contrário, a máquina ficaria muito grande. No estado de Início, os sinais *enqueue* e *dequeue* recebem o valor 0. Assim que a instrução A1 é executada no estado prólogo 1, seu valor é armazenado no QRF; para isso, o registrador *tail_data* recebe o valor de A1 e o sinal *enqueue* recebe o valor 1. Quando a instrução A2 for executada no estado de *kernel*, ela utilizará no cálculo o valor do registrador A1, anteriormente armazenado no QRF. Após a instrução A2 ser executada, é necessário retirar o valor consumido do QRF; para isso, o sinal *dequeue* recebe o valor 1. O processo de inserção e de remoção no QRF ocorrem simultaneamente no estado de *kernel*. Existe um contador que controla o número de vezes que esse estado deve ser executado; quando o contador chega ao valor determinado, a execução do *kernel* termina, e começam a ser executados os estados de epílogo. No primeiro estado de epílogo, o sinal *enqueue* recebe o valor 0, para que pare de inserir dados no QRF. O código utilizado nesse exemplo é mostrado no Apêndice F.

Ao utilizar o CP na máquina de estados da Figura 3.24, são mantidos apenas os estados: Início, *kernel* e Fim. Todos os outros estados são removidos. Nesse exemplo, o *loop pipelining* possui 4 estágios: estágio 1, composto pela instrução A1; estágio 2, composto pelas instruções L1 e L2. estágio 3, composto pela instrução M1 e estágio 4, composto pela instrução A2. O CP determina a execução ou não das instruções. Na Figura 3.25 é mostrado o CP com 4 *flip-flops* que é utilizado. Cada *flip-flop* é responsável por ativar cada estágio do *loop pipelining*. A saída do CP é mostrada pela Tabela 3.2.

As operações que ativam a inserção e remoção do QRF passam a fazer parte dos estágios. Assim, a inserção no QRF faz parte do estágio 1 e é ativada se o valor do primeiro *flip-flop* do CP for 1(*CP[0]=1*), e a remoção no QRF faz parte do estágio 4 e é ativada se o valor do terceiro

	In	A1	L1,L2	M1	A2
t0	1	0	0	0	0
t1	1	1	0	0	0
t2	1	1	1	0	0
t3	1	1	1	1	0
t4	1	1	1	1	1
t5	1	1	1	1	1
t6	0	1	1	1	1
t7	0	0	1	1	1
t8	0	0	0	1	1
t9	0	0	0	0	1
t10	0	0	0	0	0

Tabela 3.2: Funcionamento do CP da Figura 3.25 em relação ao tempo *t*.

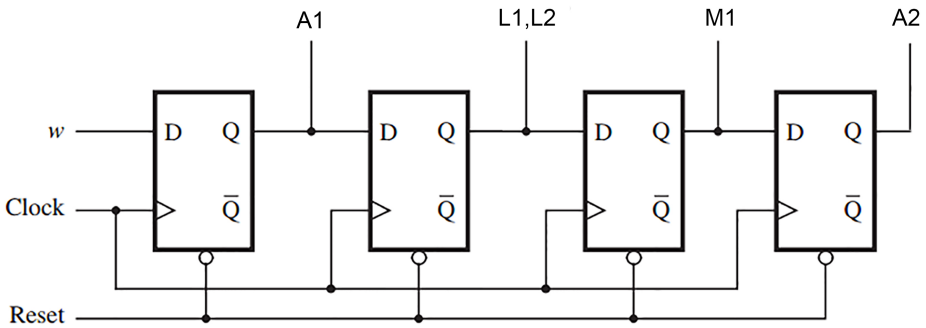

Figura 3.25: CP responsável por ativar as instruções do escalonamento mostrado na Figura 3.22.

flip-flop do CP for 1 (*CP[3]=1*). A máquina de estados que utiliza o CP e o QRF é mostrada na Figura 3.26. Para controlar a entrada do CP, é necessário um contador que o desative quando o *loop pipelining* iniciar a execução do epílogo. Ainda que apenas o estado de *kernel* tenha sido mantido, as etapas de execução do *loop pipelining* continuam as mesmas, com prólogo, *kernel* e epílogo. A entrada do CP é iniciada com o valor 1 e, quando o *loop pipelining* chega na etapa de epílogo, a entrada do CP passa a ser 0. O controle do CP é feito na máquina de estados.

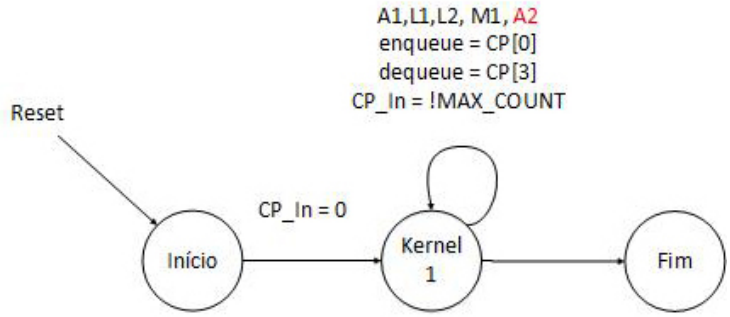

Figura 3.26: Máquina de estados utilizando QRF e o CP (Arquitetura Alvo 2).

Para a implementação dessas máquinas de estados em Verilog, foram utilizadas duas memórias de duas portas *altsyncram* da Altera, nomeadas de memória A e memória B. Na Figura 3.27 é mostrado o resultado da execução da máquina de estados utilizando apenas o QRF, onde a execução vai de 0 até 2047. Foram inseridos dados nas memórias A e B através de um arquivo mif. A memória A é carregada com os valores de 1 até 2048 sequencialmente e, a memória B é carregada com os valores de 2 até 2049. Os valores iniciais das outras variáveis são 0. A figura evidencia a comparação entre a execução da máquina de estados no ModelSim e a execução do código em C. São mostrados dois valores: o valor da soma (A2) e o valor *i*. Em C, o valor *i* determina o número de iterações. Na máquina de estados, o contador *i* controla o número de vezes que o estado de *kernel* deve ser executado. Quando *i* chega ao seu valor máximo, a máquina de estados sai do estado de *kernel*. Ao comparar os dois valores, o valor da somatória é o mesmo, mas o contador *i* possui valores divergentes. Em C, o valor de *i* vai até 2047 e na

máquina de estados utilizando o QRF o valor de *i* vai até 2043. A divergência entre valores ocorre em razão do contador *i* da máquina de estados começar a registrar valores quando entra no estado de *kernel*. Nos três estados de prólogo, o valor de *i* permanece 0 e nos três estados de epílogo, o valor de *i* se mantém o mesmo. Assumindo que o número de estágios de prólogo e epílogo são sempre iguais, o valor máximo de *i* para máquinas de estados que utilizam apenas o QRF é dado por:

i = Quantidade de iterações - Número de Estágios de prólogo ou epílogo

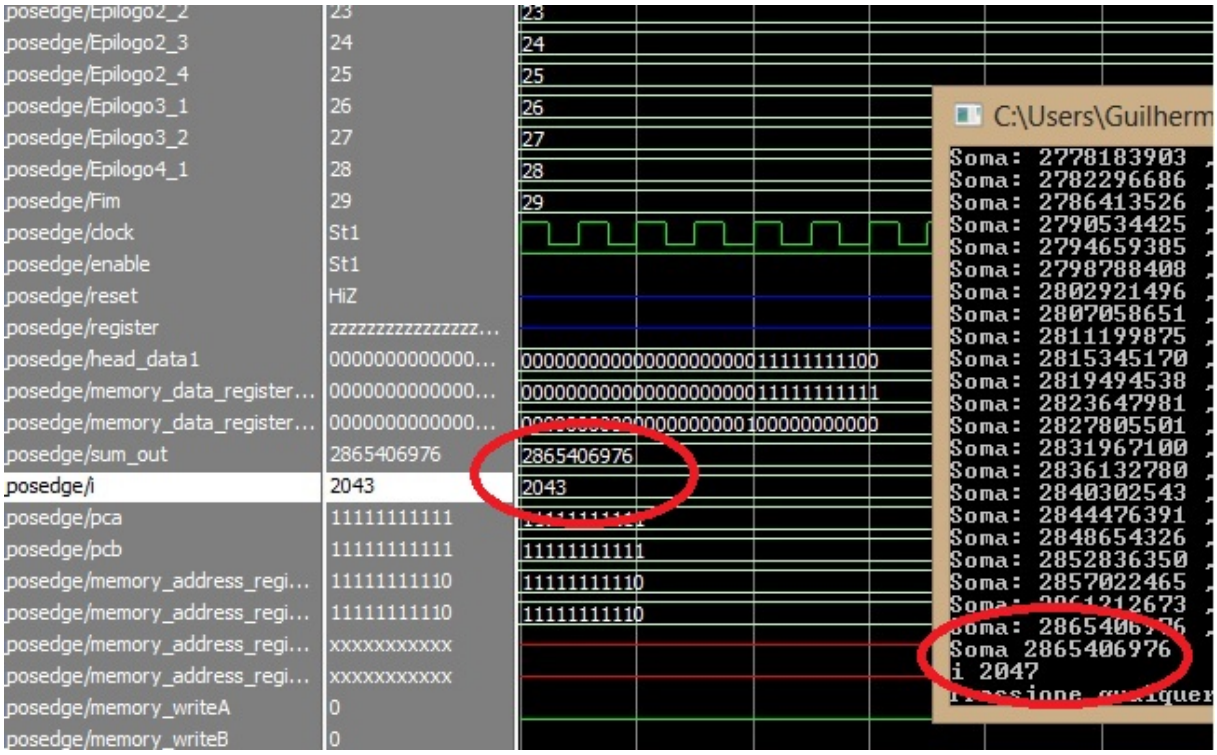

Figura 3.27: Execução da máquina de estados que utiliza QRF (Arquitetura Alvo 1) sendo comparada com C.

Na Figura 3.28 é mostrado o resultado da execução da máquina de estados utilizando o QRF e o CP, também de 0 até 2047. Nessa comparação, ocorre a mesma situação, em que o valor da somatória é o mesmo, mas o contador *i* possui valores divergentes. Em C, o valor de *i* vai até 2047 e na máquina de estados, utilizando o QRF e o CP, o valor de *i* vai até 2050. Isso ocorre porque todo o controle é feito apenas nos estados de *kernel* e o contador *i* deve ser incrementado nos estágios de prólogo e epílogo também. O valor máximo de *i* para máquinas de estados que utilizam o QRF e o CP é dado por:

i = Quantidade de iterações + Número de Estágios de prólogo ou epílogo

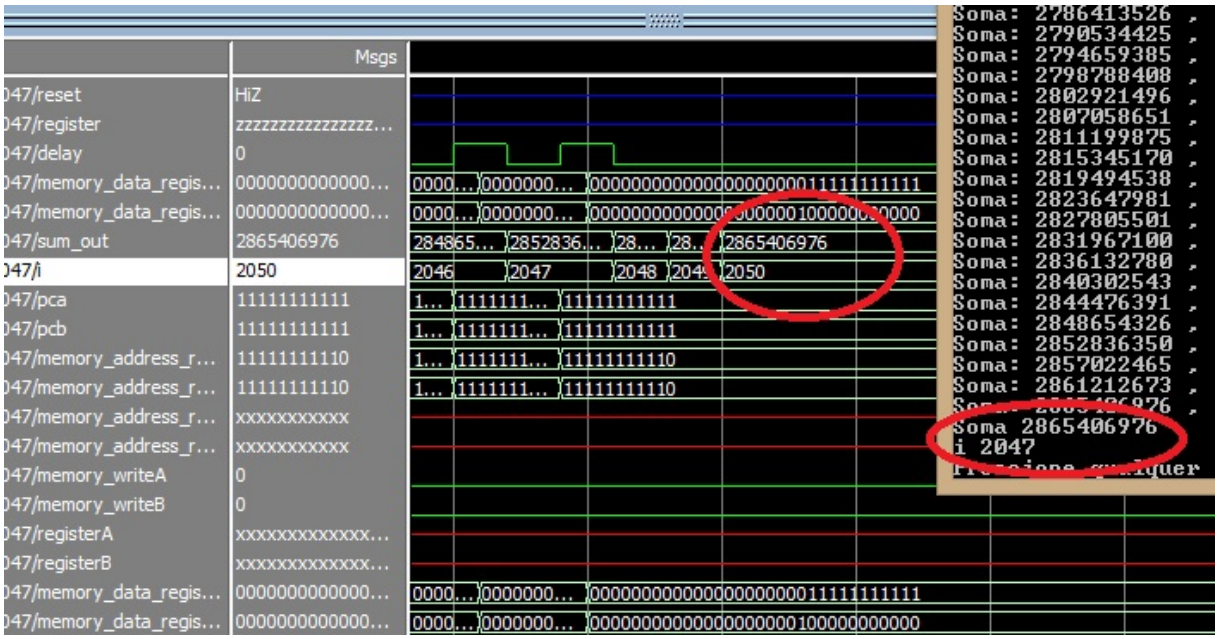

Figura 3.28: Execução da máquina de estados que utiliza QRF e CP (Arquitetura Alvo 2) sendo comparada com C.

Capítulo 4

RESULTADOS EXPERIMENTAIS

Este Capítulo está dividido em duas seções: a Seção 4.1 mostra os resultados com a Arquitetura Alvo 1 e a Seção 4.2 mostra os resultados com a Arquitetura Alvo 2. Todos os testes mostrados neste capítulo, foram realizados utilizando o ModelSim Altera Starter Edition 10.1d e o Quartus II 64-Bit Versão 13.0.0 Build 156 Web Edition e, depois foram mapeados para a placa DE-2 da Altera. Para os testes, foram escolhidos o algoritmo Dotprod modificado, Fibonacci e um *loop* simples. O Dotprod modificado é o mesmo do código da Figura 3.22 da Subseção 3.6.5. O *loop* simples é quase o mesmo utilizado nos testes com memória da Subseção 3.6.2, mudando algumas somas por multiplicações. Para comparar esses algoritmos, as máquinas de estados paralelizadas que implementam o *loop pipelinig* foram comparadas com máquinas de estados que executam os algoritmos de forma sequencial, sem o uso do o *loop pipelinig*. Nestes testes são utilizadas as memórias de duas portas *altsyncram* da Altera da Figura 3.13. Os testes foram divididos em vários grupos para facilitar a visualização e comparação entre si. São mostrados ao total 20 testes.

O objetivo destes testes é avaliar a arquitetura alvo proposta, em particular os módulo especializados QRF e CP quanto a:

- Funcionalidade.
- Desempenho.
- Uso de recursos de *hardware*.

4.1 Testes com a Arquitetura Alvo 1

A Arquitetura Alvo 1 utiliza a técnica do *loop pipelining* e o arquivo de registradores QRF. O objetivo do QRF é solucionar o problema de sobreposição de valores sem que a arquitetura alvo perca muita eficiência.

4.1.1 Primeiro grupo de testes

No primeiro grupo de testes, foi utilizado o algoritmo Dotprod modificado com a Arquitetura Alvo 1. Para validar a arquitetura, foram feitos testes de escalabilidade, aumentando o número de iterações a cada teste. Junto à arquitetura, foi utilizada a técnica de bordas de *clock* alternadas, que foi mostrada na Subseção 3.6.4 na Figura 3.19. Na Tabela 4.1 são mostrados os resultados.

Teste	*Loop Pipe.*	QRF	Iterações	Cam. Crítico	*bits*	Freq. (MHz)	Elem. Lógicos	Pot.(mW)	Ciclos	Tempo(μs)
t1	não	não	0 - 10	mul.	16	132,42	138	117.92	73	0,55
t2	não	não	0 - 255	mul.	16	133,33	186	125.50	1788	13,4
t3	não	não	0 - 2047	mul.	32	85,98	242	125.49	14332	166
t4	sim	sim	0 - 10	mul.	16	79.67	426	116.05	37	0,45
t5	sim	sim	0 - 255	soma	16	70.74	567	122.13	772	10,9
t6	sim	sim	0 - 2047	QRF	32	48.29	836	125.49	6148	127

Tabela 4.1: Primeiro grupo de testes com o algoritmo Dotprod modificado.

O cálculo utilizado para descobrir o tempo em segundos é dado por:

$$Tempo = Ciclos * 1/(Frequência * 1000000)$$

Na Tabela 4.1, os testes foram divididos em dois subgrupos : testes em que o QRF e o *loop pipelining* não são utilizados (testes 1 a 3) e testes em que esses componentes são utilizados (testes 4 a 6). As colunas representam respectivamente: o teste, se este utiliza *loop pipelining*, se utiliza o QRF, a quantidade de iterações, o caminho crítico, a quantidade de *bits* da arquitetura, frequência em *megahertz*, a quantidade de elementos lógicos, a potência em *miliwatt*, o número de ciclos, e o tempo em microssegundos. As tabelas mostradas em outros testes irão seguir esse padrão de organização das colunas. Os testes de 1 a 3 foram feitos apenas para ajudar a comparação dos *loops* paralelizados com *loops* não paralelizados, e serão utilizados na comparação dos próximos grupos de testes também. Os testes de 1 a 3 não possuem QRF ou CP e utilizam bordas de *clock* positivas na máquina de estados e na memória. Ao analisar a tabela, fica claro que à medida em que o número de iterações aumenta, a frequência nos testes de 4 a 6 diminui. A frequência mostrada na tabela é a frequência máxima mostrada pelo Quartus II.

Nos testes de 1 a 3, a frequência diminui apenas ao aumentar o número de iterações para 0 a 2047 (teste 3). Isso ocorre porque, ao aumentar o número de iterações, é necessário aumentar a quantidade de bits utilizados nos cálculos. Por exemplo, a multiplicação que era de 16 *bits* nos testes 1 e 4 passou a ser de 32 *bits* nos testes 3 e 6. O tamanho dos dados armazenados na memória também aumentou de 16 para 32 *bits*. Ao aumentar a quantidades de *bits* utilizados nos cálculos, é necessário mais tempo para que os cálculos sejam realizados e para que os valores sejam transferidos para a memória e QRF, o que diminui a frequência. Comparando a frequência dos testes de 4 a 6, em relação à frequência dos testes 1 a 3, pode ser observado que ocorreu uma diminuição da frequência. Esse fato ocorre por estar sendo utilizada a técnica de bordas de *clock* alternadas. Nessa técnica, todas as operações devem ser feitas em apenas meio ciclo de *clock*, e a frequência máxima teve de ser diminuída para não ocorrer nenhum erro. Já em uma comparação do número de ciclos de *clock* dos testes de 4 a 6, em relação ao número de ciclos de *clock* dos testes de 1 a 3, ficou claro que o número de ciclos caiu mais da metade. Ocorreu um aumento no número de elementos lógicos ao comparar os dois subgrupos de testes. A diminuição no tempo de execução ao comparar os dois subgrupos foi pequeno. Ao analisar esse grupo de testes, pode ser constatado que utilizar a técnica de bordas de *clock* alternadas junto à Arquitetura Alvo 1 não proporciona uma diminuição significativa no tempo de execução. Quanto ao consumo de energia, ocorreu um aumento à medida em que o número de iterações aumentou. Na coluna onde é mostrado o caminho crítico, é analisada a parte do circuito com o maior *delay*. Na maioria dos testes, as operações aritméticas tiveram o maior *delay*, apenas em t6 o QRF se mostrou ser o caminho crítico, onde a arquitetura é de 32 *bits* e o número de dados a serem transferidos é maior.

4.1.2 Segundo grupo de testes

O segundo grupo de testes foi feito da mesma forma que o primeiro. A única mudança foi o tipo de estratégia de *clock*. Nesse grupo de testes, foi utilizada a estratégia de bordas de *clock* positivas junto à Arquitetura Alvo 1. Na Tabela 4.2 são mostrados os testes. Os testes deste grupo são comparados com os testes do grupo anterior da Tabela 4.1.

Teste	Cam. Crítico	*bits*	Iterações	Freq. (MHz)	Elem. Lógicos	Pot.(mW)	Ciclos	Tempo(μs)
t7	mul.	16	0 - 10	160,15	438	118.02	51	0,32
t8	mul.	16	0 - 255	143,53	572	121.86	1030	7,18
t9	QRF	32	0 - 2047	95,45	943	125.69	8198	85,89

Tabela 4.2: Segundo grupo de testes com o algoritmo Dotprod modificado utilizando bordas de *clock* positivas.

Nesse segundo grupo de testes, são utilizados o *loop pipelining* e o QRF, e portanto foram

retiradas as colunas que correspondiam ao uso ou não dos mesmos. O restante das colunas representa a mesma divisão de dados feita no grupo de testes anterior. Ao comparar os testes que utilizam *loop pipelining* e QRF (testes 7 a 9) com os testes que não utilizam essas técnicas (testes 1 a 3 da Tabela 4.1), pode ser constatado que não houve queda na frequência e o número de ciclos diminuiu, mesmo que não tanto quanto nos testes de 4 a 6 da Tabela 4.1. Ocorreu um aumento no número de elementos lógicos dos testes de 7 a 9 da Tabela 4.2 em comparação com os testes 4 a 6 da Tabela 4.1, isso porque foram necessários mais estados de *delay* na máquina de estados ao utilizar a estratégia de *clock* de bordas positivas. O tempo de execução dos testes de 7 a 9 diminuíram significativamente se comparados ao tempo de execução dos testes de 1 a 3. Nesse grupo de testes, o consumo de energia aumentou a medida em que o número de iterações aumentou. Assim como no grupo anterior, o QRF se mostrou parte do caminho crítico ao aumentar a quantidade de iterações e o número de *bits* da arquitetura.

Na Figura 4.1 é mostrado um gráfico comparando o tempo de execução entre os testes 3, 6 e 9, que são os que executam de 0 a 2047 iterações. Os testes 6 e 9 tiveram um melhor tempo de execução, se comparados ao teste 3. Entre os dois testes, o teste 9 se sobressaiu ao teste 6, o que mostra que a estratégia de *clock* de bordas positivas é mais eficiente. Comparando o teste 9 com o teste 3, ocorreu uma diminuição de quase metade do tempo de execução. Os testes com menos iterações seguiram o mesmo padrão de diminuição do tempo de execução.

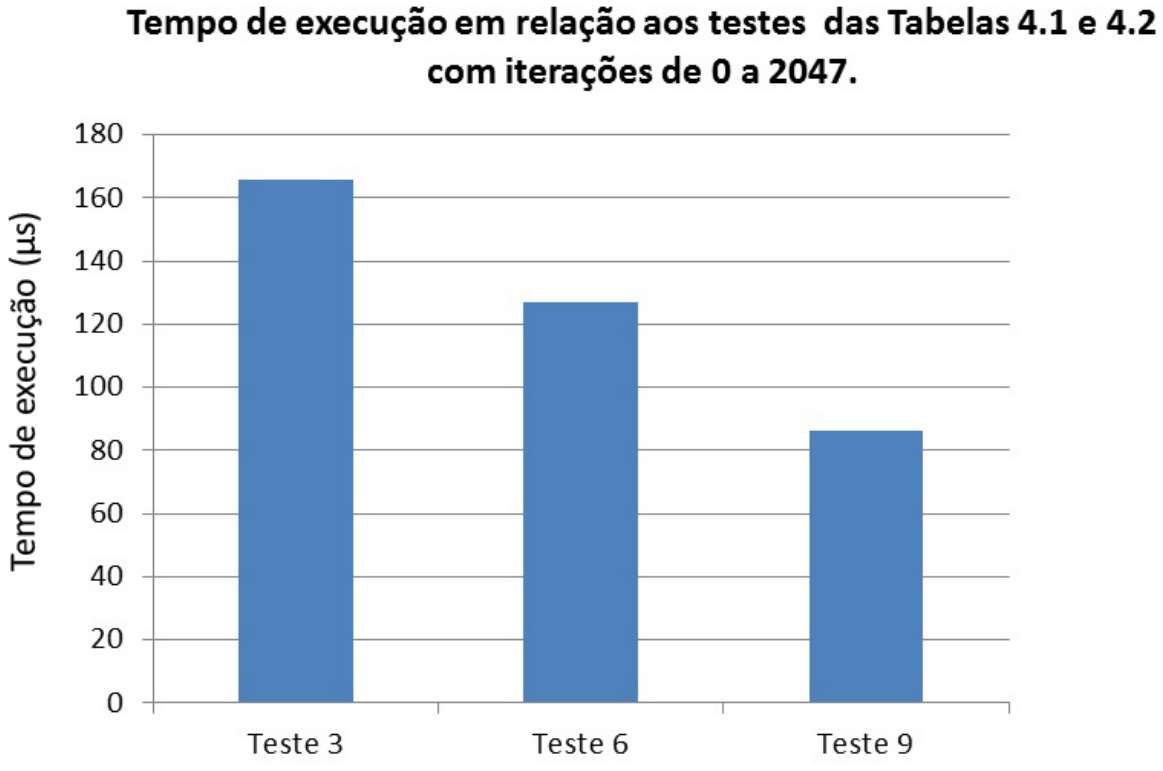

Figura 4.1: Gráfico comparativo entre os testes das Tabelas 4.1 e 4.2.

4.1.3 Terceiro grupo de testes

Foi realizado mais um teste com o algoritmo DotProd Modificado e a Arquitetura Alvo 1. Nesse teste, foi utilizada a estratégia de dois *clocks*, que é mostrada na Figura 3.21 da Subseção 3.6.4. O teste foi comparado com a implementação sequencial do DotProd Modificado também utilizando dois *clocks*: um com uma frequência menor para a máquina de estados e um com uma frequência maior para a memória. O teste, também é comparado com os resultados dos grupos de teste anteriores. O número de iterações nesses testes vão de 0 a 2047 e não foram feitos testes com números menores de iterações. O teste é mostrado na Tabela 4.3 .

Teste	t10	t11
Loop Pipelining	não	sim
QRF	não	sim
Potência (mW)	125.53	125.59
Iterações	0 - 2047	0 - 2047
Caminho Crítico	multiplicação	QRF
Frequência (MHz)	343,17	319,69
Elementos Lógicos	235	944
Ciclos	20473	8202
Tempo(μs)	59,66	25,66

Tabela 4.3: Terceiro grupo de testes com o algoritmo Dotprod modificado utilizando a estratégia de dois *clocks*.

No teste 11 da da Tabela 4.3, ocorreu uma diminuição de mais da metade do tempo de execução em relação ao teste 10. A frequência e a quantidade de ciclos mostrados na tabela, são do *clock* com a frequência maior. O teste 11 em relação ao teste 10 executou o algoritmo em um tempo 57% menor, enquanto o teste 9 em relação ao teste 3 executou o algoritmo em um tempo 48% menor. Apesar dos resultados serem muito promissores, essa arquitetura não foi testada o suficiente devido a falta de tempo. Nas outras arquiteturas, além dos testes mostrados, foram feitos testes com variações dos valores de entrada, nos tipos de memória, nos tipos de pinagem da placa, com filas da altera e outros tipos de QRF e CP, e variações na estrutura da máquina de estados, tudo para buscar possíveis problemas. O mesmo não foi feito com essa arquitetura de dois *clocks*. No teste 11 o QRF foi responsável pelo maior *delay* no circuito, porém conseguiu diminuir o tempo de execução e não ocorreram grandes mudanças no consumo de energia.

4.1.4 Quarto grupo de testes

O quarto grupo de testes foi feito com o algoritmo Fibonacci. Nesse grupo, foi testada a Arquitetura Alvo 1, que utiliza somente o QRF. Os testes foram feitos da mesma forma que nos grupos anteriores, em que primeiro foram testadas poucas iterações e depois um número maior de iterações. Porém, serão mostrados apenas os testes que foram feitos com um grande número de iterações. Os testes com menor quantidade de iterações, possuem resultados bastante semelhantes apenas em uma escala menor. Serão mostrados três testes: um teste sem nenhum tipo de paralelização, um teste com paralelização utilizando bordas de *clock* alternadas e um teste com paralelização utilizando bordas de *clock* positivas. Todos os testes desse grupo utilizam uma arquitetura de 32 *bits*. Na Tabela 4.4 é mostrado o quarto grupo de testes. Para o algoritmo Fibonacci não foram feitos testes com a arquitetura com dois *clocks*.

Teste	Tipo clock	*Loop Pipe.*	QRF	Cam. Crítico	Iterações	Freq. (MHz)	Elem. Lógico	Pot.(mW)	Ciclos	Tempo(μs)
t12	bordas pos.	não	não	soma	0 - 44	136,61	161	121.78	311	2,28
t13	bordas alt.	sim	sim	memória	0 - 44	86,73	498	121.95	228	2,63
t14	bordas pos.	sim	sim	QRF	0 - 44	174,49	501	121.50	271	1,55

Tabela 4.4: Testes com o algoritmo Fibonacci e QRF.

Na Tabela 4.4, o teste 14 teve um melhor desempenho que o teste 12; já o teste 13 teve um desempenho pior , como é mostrado na Figura 4.2. No teste 14 o QRF foi responsável pelo maior *delay* no circuito, porém o mesmo teste teve a maior frequência de *clock*. Isso mostra que apesar de ser parte do caminho caminho crítico o QRF pode aumentar a frequência. O consumo de energia não teve grande variação nos testes.

4.2 Testes com a Arquitetura Alvo 2

O objetivo do CP é diminuir a quantidade de elementos lógicos, sem causar um aumento muito grande no tempo total de execução do algoritmo. Os testes desta Seção utilizam a Arquitetura Alvo 2 e, o foco principal será na diminuição dos elementos lógicos.

4.2.1 Quinto grupo de testes

O quinto grupo de testes foi feito utilizando-se a Arquitetura Alvo 2, que utiliza o CP e o QRF. Neste grupo, serão mostrados apenas os testes com o número máximo de iterações e utilizando a estratégia de bordas positivas de *clock*. Todos os testes mostrados utilizam QRF e *loop pipelining*. Na Tabela 4.5 é mostrado o quinto grupo de testes, no qual foram utilizados os

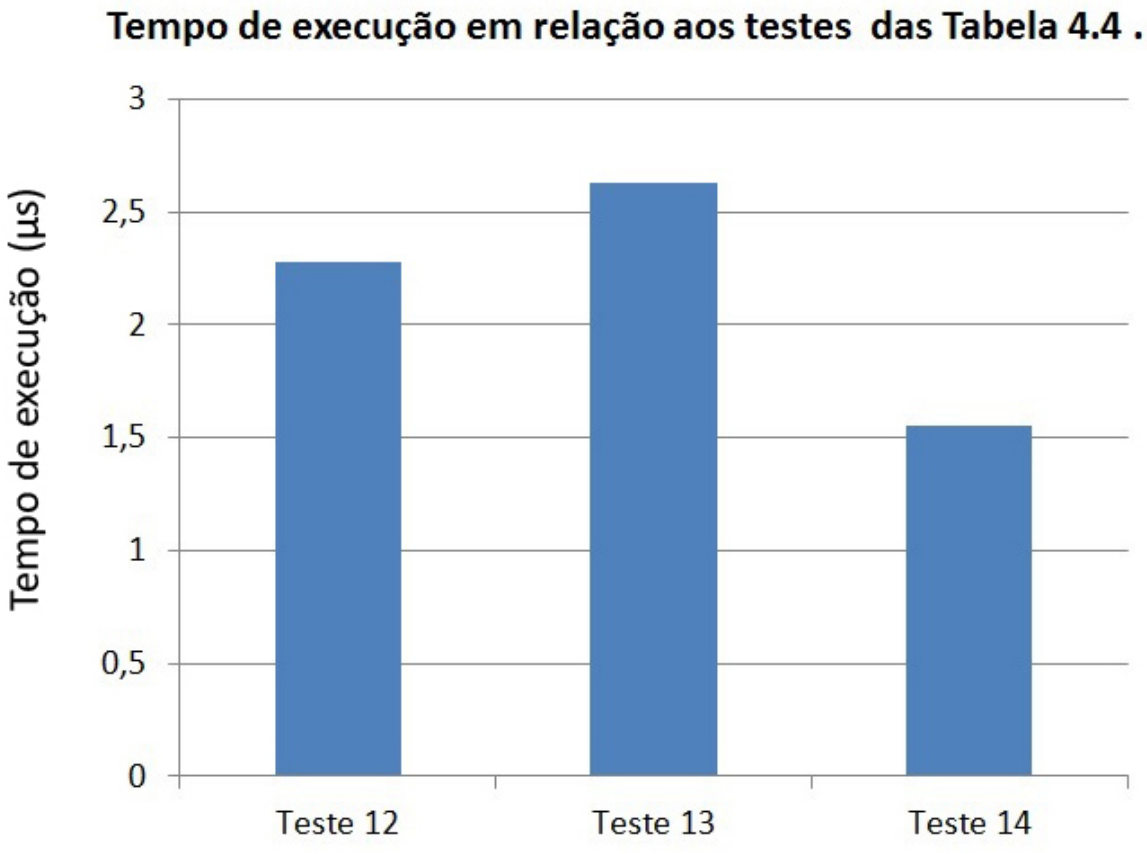

Figura 4.2: Gráfico comparativo dos testes da Tabela4.4.

algoritmos Dotprod modificado e Fibonacci. Os testes da Tabela 4.5 serão comparados com os testes das tabelas anteriores. A arquitetura utilizada nesses testes é de 32 *bits*.

Teste	Algoritmo	CP	Cam. Crítico	Iterações	Frequência (MHz)	Elem. Lógicos	Pot.(mW)	Ciclos	Tempo(μs)
t15	Fibonacci	sim	QRF	0 - 44	170,30	498	122.08	271	1,59
t16	DotProd	sim	QRF	0 - 2047	100,81	800	125.51	8197	81,31

Tabela 4.5: Testes com os algoritmos anteriores utilizando CP.

Os testes 15 e 16, da Tabela 4.5 são equivalentes aos testes 14 e 9 das Tabelas 4.4 e 4.2 respectivamente, mas diferente dos testes anteriores utilizam o CP. Comparando os testes que utilizam CP com os testes que não o utilizam, pode ser dito que com o CP o número de elementos lógicos diminuiu. Apesar do número de elementos lógicos no teste 15 ter diminuído, a diferença foi muito pequena, enquanto no teste 16 a queda de elementos lógicos foi maior. Essa diferença de elementos lógicos entre os testes pode ser mais bem visualizada no gráfico da Figura 4.3. No teste 15, o tempo de execução aumentou pouco (2.5%) e, no teste 16, o tempo teve uma pequena queda. O QRF foi responsável pelo maior *delay* no circuito dos dois testes.

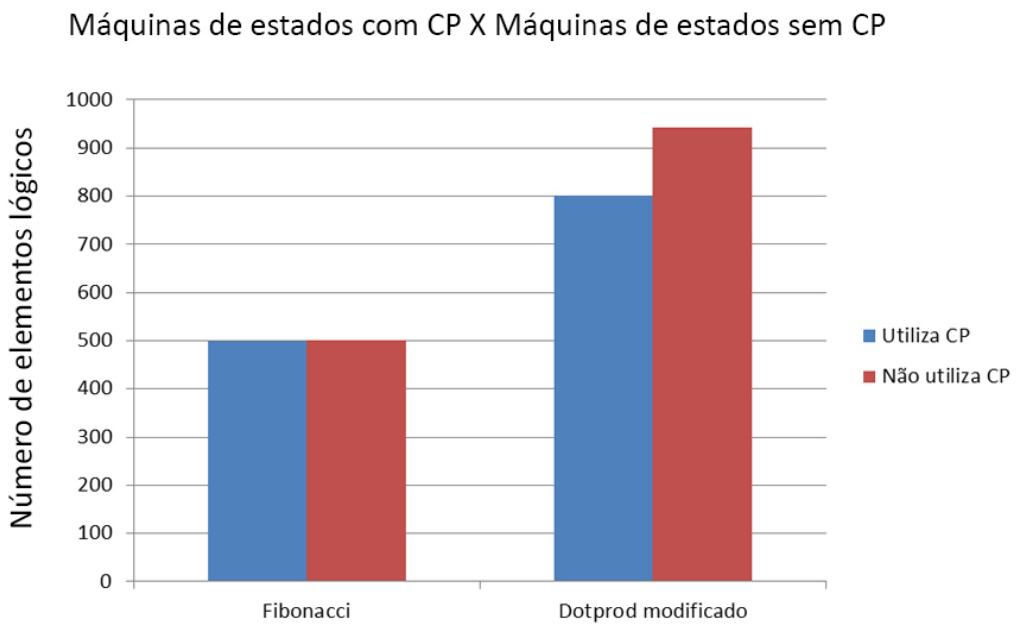

Figura 4.3: Gráfico com a quantidade de elementos lógicos na máquina de estados quando se usa o CP.

4.2.2 Sexto grupo de testes

Foi realizado um sexto grupo de testes em que foi testado um *loop* que não necessita do QRF, pois não possui sobreposição de valores ao ser paralelizado com a técnica do *loop pipelining*. Por não precisar utilizar o QRF, esse *loop* foi chamado de *loop* simples. Serão mostrados apenas os testes com o maior número de iterações e que utilizam a estratégia de bordas de *clock* positivas, pois os resultados com bordas alternadas e dois *clocks* não foram feitos para esse algoritmo. Nesse grupo, foram realizados três testes: um sem *loop pipelining*, um com *loop pipelining* e um com *loop pipelining* e CP. Os testes são mostrados na Tabela 4.6.

Teste	*Loop Pipe.*	CP	Cam. Crítico	Iterações	Frequência (MHz)	Elem. Lógicos	Pot.(mW)	Ciclos	Tempo(μs)
t17	não	não	Memória	0 - 2047	166,14	248	125.59	32755	197,15
t18	sim	não	Memória	0 - 2047	173,34	248	125.58	8203	47,32
t19	sim	sim	Memória	0 - 2047	167,39	250	125.49	8203	49,01

Tabela 4.6: Teste com o CP e o *loop* simples sem QRF.

Ao observar a Tabela 4.6, fica visível que os testes que utilizam o *loop pipelining* (testes 18 e 19) tiveram uma grande diminuição no tempo de execução. Com relação à quantidade de elementos lógicos, a utilização do CP não se mostrou eficiente, pois houve um aumento na quantidade de elementos lógicos. Isso ocorreu, pois como o *loop* é pequeno, o *overhead* do CP é maior que a economia de se eliminar o prólogo e epílogo. Isso deve ser diferente no caso de *loops* maiores. Nesse grupo de testes não ocorreu uma grande mudança no consumo de energia,

e a memória foi responsável pelo maior *delay* nos testes.

4.2.3 Sétimo grupo de testes

Foi realizado um último teste com o algoritmo Dotprod modificado utilizando a Arquitetura Alvo 2 e a técnica de utilizar dois *clocks*. O teste é mostrado na Tabela 4.7.

	t11	t20
CP	não	sim
Potência (mW)	125.59	125.64
Caminho Crítico	QRF	QRF
Iterações	0 - 2047	0 - 2047
Frequência (MHz)	319,69	342,00
Elementos Lógicos	944	804
Ciclos	8202	8198
Tempo(μs)	25,66	23,97

Tabela 4.7: Teste com o uso do CP junto à estratégia de dois *clocks*.

Na Tabela 4.7, o teste 20 é comparado com o teste 11, que também utiliza a técnica de dois *clocks*. O teste 11 é o mesmo da Tabela 4.3 e foi adicionado apenas para facilitar a comparação. Os dois testes utilizam *loop pipelining* e QRF. No teste 20, que utiliza o CP, além de ocorrer uma diminuição no número de elementos lógicos, devido à eliminação do prólogo e epílogo, também ocorreu uma diminuição no tempo de execução. Na Figura 4.4 são mostrados todos os testes feitos no trabalho com o algoritmo Dotprod modificado.

Ao analisar a Figura 4.4, pode-se notar que os testes que utilizam a técnica de duplo *clock* possuem o menor tempo de execução, a técnica de bordas positivas também foi eficiente na redução do tempo de execução, já técnica de bordas alternadas possui o maior tempo de execução. O CP se mostrou eficaz ao analisar os números de elementos lógicos de t16 e t20 em relação a t11 e t9.

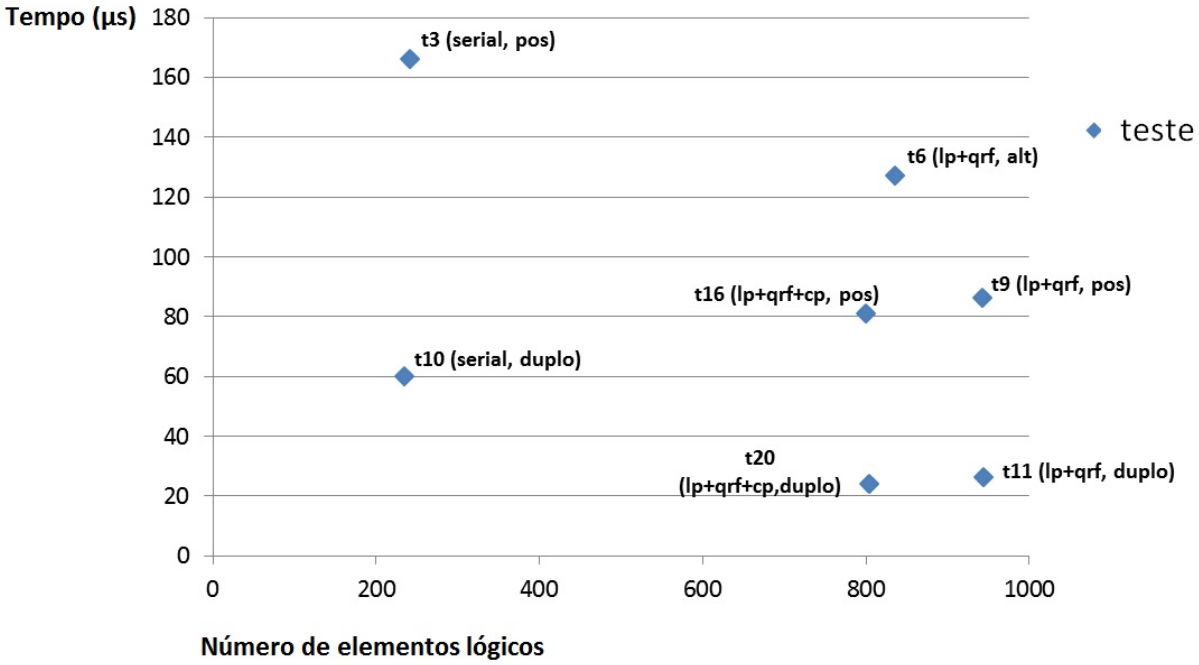

Figura 4.4: Gráfico onde é mostrada a relação entre tempo e elementos lógicos nos testes com o Dotprod modificado.

Capítulo 5

CONCLUSÕES

O trabalho apresentado propôs um arquivo de registradores baseados em filas denominado QRF e propôs também um arquivo de registradores predicados denominado CP. As propostas foram cumpridas e com elas foram obtidas as principais contribuições desejadas, que são:

- A solução para o problema de sobreposição de valores que ocorre na técnica *loop pipelining* através do uso do QRF.
- A transformação da arquitetura alvo em uma arquitetura mais amigável ao compilador e também mais otimizada através do uso do CP.
- A exploração de uma série de opções de projeto de *hardware* que levam a uma definição mais precisa da arquitetura alvo a ser considerada pelo compilador.

Com base nos testes realizados ficou evidente que o *loop pipelining*, mesmo com a adição do QRF, mostrou-se ser vantajoso, pois o número de ciclos caiu consideravelmente em relação aos algoritmos que não foram implementados utilizando essa técnica. Além do número de ciclos, o QRF também se mostrou eficiente em relação ao tempo de execução, pois a maioria dos testes que utilizaram o *loop pipelining* e QRF tiveram um menor tempo de execução. Apesar de em alguns testes com o QRF ocorrer uma queda na frequência máxima, e em alguns casos o mesmo ser responsável pelo maior *delay* no circuito, a queda no número de ciclos compensou essas desvantagens. A maior desvantagem nos testes com *loop pipelining* e QRF foi o aumento no número de elementos lógicos. O responsável por esse aumento não é o QRF diretamente, e sim a própria técnica *loop pipelining* em si. Pensando em diminuir esse aumento no número de elementos lógicos foi utilizado o CP.

A utilização do CP diminuiu a quantidade de estados necessários para realizar o *loop pipelining*. Com relação à quantidade de elementos lógicos, essa diminuição se mostrou vantajosa

em alguns testes, mas em outros não ocorreu uma grande diminuição ou, ocorreu um pequeno aumento no número de elementos lógicos. O CP conseguiu êxito nos testes mais complexos em que o QRF também foi utilizado, já nos testes mais simples o CP causou um pequeno aumento no número elementos lógicos. Uma possível explicação para isso é, que em testes mais simples a diminuição de elementos lógicos que o uso do CP trás não supere a adição de elementos lógicos necessários para que o próprio CP seja utilizado. Apesar de não diminuir o número de elementos lógicos em alguns testes, o uso do CP facilita a tarefa de compilação e geração automática de código otimizado, o que se constitui em um recurso de interesse para a construção de aceleradores em FPGAs.

Os testes não mostraram grandes variações no consumo de energia, mas claro que, ao aumentar a quantidade de elementos lógicos ocorreu um pequeno aumento no consumo de energia.

Este trabalho, além de mostrar o desenvolvimento desses dois arquivos de registradores, também mostrou diferentes formas de implementação da arquitetura alvo, o que em alguns testes mostrou fazer diferença nos resultados. Também foram mostradas maneiras diferentes de se desenvolver a máquina de estados e como utilizar a memória. Grande parte do esforço utilizado no trabalho foi dedicado ao ajuste entre os componentes criados e as máquinas de estados. Além dos testes mostrados, foram realizados testes na placa DE-2 e para isso foi desenvolvido um método de comunicação entre as máquinas de estados e os pinos da placa.

Serão realizados novos testes utilizando *loops* do *"The San Diego Vision Benchmark Suite"* (VENKATA et al., 2009), e assim avaliar com maior profundidade os possíveis benefícios da arquitetura desenvolvida neste trabalho.

Para trabalhos futuros, poderia ser adicionado um novo módulo ao compilador Cetus Modificado para gerar as Arquiteturas Alvo 1 e 2 de forma automática, além disso, também gerar o código em Verilog sem paralelização para que a tarefa de analisar as arquiteturas fosse mais fácil. Com vários testes e comparações, o compilador Cetus Modificado poderia ser capaz de mostrar automaticamente qual arquitetura é mais vantajosa para um determinado código.

O objetivo do uso dos componentes mostrados neste trabalho, não é somente desempenho e economia de elementos lógicos, mas também facilitar a geração de *hardware* semi-automaticamente, a partir de algoritmos descritos em linguagem de alto nível.

REFERÊNCIAS BIBLIOGRÁFICAS

ALDEC. 2014. URL:https://www.aldec.com. Último acesso em: 02/09/2014.

ALLAN, V. H. et al. Software pipelining. *ACM Computing Surveys (CSUR)*, ACM, v. 27, n. 3, p. 367–432, 1995.

ALTERA. 2014. URL:http://www.altera.com. Último acesso em: 02/09/2014.

ALTERA. *Internal Memory (RAM and ROM) User Guide*. 2014. URL:https://www.altera.com. Último acesso em: 16/09/2014.

BACON, D.; RABBAH, R.; SHUKLA, S. Fpga programming for the masses. *Queue*, ACM, v. 11, n. 2, p. 40, 2013.

BANU, J. et al. Loop parallelization and pipelining implementation of aes algorithm using openmp and fpga. In: *Emerging Trends in Computing, Communication and Nanotechnology (ICE-CCN), 2013 International Conference*. Tirunelveli, India: [s.n.], 2013. p. 481–485.

BEN-ASHER, Y.; MEISLER, D.; ROTEM, N. Reducing memory constraints in modulo scheduling synthesis for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, ACM, New York, USA, v. 3, n. 3, p. 15:1–15:19, set. 2010. ISSN 1936-7406. Disponível em: <http://doi.acm.org/10.1145/1839480.1839485>.

BROST, V. et al. Flexible vliw processor based on fpga for real-time image processing. In: *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference*. Tampere, Finland: [s.n.], 2011. p. 1–8.

BROWN, S.; VRANESIC, Z. *Fundamentals of Digital Logic with Verilog Design*. [S.l.]: McGraw-Hill, 2003. ISBN 0072823151.

CHU, P. P. *Fpga Prototyping by Verilog Examples Xilinx Spartan 3 Version*. [S.l.]: John Wiley & Sons, 2008. ISBN 9780470185322.

COFFMAN, K. *Real World FPGA Design with Verilog*. [S.l.]: Prentice Hall, 2000. ISBN 0130998516.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, ACM, v. 34, n. 2, p. 171–210, 2002.

COUSSY, P. et al. An introduction to high-level synthesis. *Design Test of Computers, IEEE*, v. 26, n. 4, p. 8–17, 2009. ISSN 0740-7475.

DAVE, C. et al. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, IEEE, v. 42, n. 12, p. 36–42, 2009.

ELLSON, J. et al. Graphviz open source graph drawing tools. In: SPRINGER. *Graph Drawing*. [S.l.], 2002. p. 483–484.

FERNANDES, M. *A Clustered VLIW Architecture Based on Queue Register Files*. Tese (Doutorado) — University of Edinburgh, 1998.

FISHER, J. A.; FARABOSCHI, P.; YOUNG, C. *Embedded computing: a VLIW approach to architecture, compilers and tools*. [S.l.]: Elsevier Inc., 2005. ISBN 1558607668.

GNANAOLIVU, R.; NORVELL, T.; VENKATESAN, R. Mapping loops onto coarse-grained reconfigurable architectures using particle swarm optimization. In: *Soft Computing and Pattern Recognition (SoCPaR), 2010 International Conference*. Paris, France: [s.n.], 2010. p. 145–151.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados e Algoritmos em Java*. [S.l.]: Bookman, 2007. ISBN 9788560031504.

GREENE, J.; HAMDY, E.; BEAL, S. Antifuse field programmable gate arrays. *Proceedings of the IEEE*, IEEE, v. 81, n. 7, p. 1042–1056, 1993.

HAMBLEN, J. O.; HALL, T. S.; FURMAN, M. D. *Rapid prototyping of digital systems: SOPC edition*. [S.l.]: Springer, 2008. ISBN 9780387726700.

HAUCK, S. The roles of fpgas in reprogrammable systems. *Proceedings of the IEEE*, v. 86, n. 4, p. 615–638, 1998. ISSN 0018-9219.

HAUCK, S.; DEHON, A. *Reconfigurable computing: the theory and practice of FPGA-based computation*. [S.l.]: Morgan Kaufmann, 2008. ISBN 9780123705228.

HENNESSY, J.; PATTERSON, D. *Computer Architecture: A Quantitative Approach*. [S.l.]: Elsevier, 2011. ISBN 9780123838728.

HUCK, J. et al. Introducing the ia-64 architecture. *Micro, IEEE*, v. 20, n. 5, p. 12–23, 2000. ISSN 0272-1732.

IEEE. Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, p. 1–560, 2006.

JHIN-BIN, J.; KUEN-CHENG, C.; SHANN, J.-J. A predicate-aware modulo scheduling for improving resource efficiency of coarse grained reconfigurable architectures. In: *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium*. Karlsruhe, Germany: [s.n.], 2012. p. 311–314.

JONES, R.; ALLAN, V. Software pipelining: a comparison and improvement. In: *Microprogramming and Microarchitecture. Micro 23. Proceedings of the 23rd Annual Workshop and Symposium., Workshop on*. [S.l.: s.n.], 1990. p. 46–56.

KAPRE, N.; DEHON, A. Vliw-score: Beyond c for sequential control of spice fpga acceleration. In: *Field-Programmable Technology (FPT), 2011 International Conference*. New Delhi, India: [s.n.], 2011. p. 1–9.

KIYOHARA, K. et al. Register connection: A new approach to adding registers into instruction set architecture. *Proc. 20th Ann. Int'l Symp. Computer Architecture*, p. 247–256, May 1993.

KUON, I.; TESSIER, R.; ROSE, J. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, Now Publishers Inc., v. 2, n. 2, 2008.

LAM, M. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 23, n. 7, p. 318–328, jun. 1988. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/960116.54022>.

LEE, S.-I.; JOHNSON, T.; EIGENMANN, R. Cetus–an extensible compiler infrastructure for source-to-source transformation. *Languages and Compilers for Parallel Computing*, Springer, p. 539–553, 2004.

LEI, G. et al. A software pipelining algorithm in high-level synthesis for fpga architectures. In: *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*. San Jose, USA: [s.n.], 2009. p. 297–302.

MENOTTI, R. *LALP:uma linguagem para exploração do paralelismo de loops em computação reconfigurável*. Tese (Doutorado) — Universidade de São Paulo, 2010.

NAVEH, B. et al. *Jgrapht*. 2008. URL:http://www.jgrapht.org. Último acesso em: 20/06/2013.

PURNAPRAJNA, M.; IENNE, P. Making wide-issue vliw processors viable on fpgas. *ACM Trans. Archit. Code Optim.*, ACM, New York, USA, v. 8, n. 4, p. 33:1–33:16, jan. 2012. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2086696.2086712>.

QI, G. et al. Automatic loop-based pipeline optimization on reconfigurable platform. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference*. Melbourne, Australia: [s.n.], 2013. p. 919–926.

RAU, B. R. Iterative modulo scheduling: An algorithm for software pipelining loops. In: ACM. *Proceedings of the 27th annual international symposium on Microarchitecture*. [S.l.], 1994. p. 63–74.

RAU, B. R. et al. Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. In: *Proceedings of the ACM SIGPLAN*. [S.l.: s.n.], 1992. v. 92, p. 283–299.

RETTORE, P. *infraestrutura de compilação para a implementação de aceleradores em fpga*. Dissertação (Mestrado) — Universidade Federal de São Carlos – UFSCAR, 2012.

RONG, H.; DOUILLET, A.; GAO, G. R. Register allocation for software pipelined multidimensional loops. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 30, n. 4, p. 23:1–23:68, ago. 2008. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/1377492.1377498>.

SEJONG, O. et al. Speculative loop-pipelining in binary translation for hardware acceleration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, v. 27, n. 3, p. 409–422, March 2008. ISSN 0278-0070.

SKLIAROVA, I.; FERRARI, A. B. Introdução à computação reconfigurável. *Revista do DETUA*, v. 2, n. 6, 2003.

STOTZER, E. J.; LEISS, E. L. Modulo scheduling without overlapped lifetimes. In: ACM. *ACM Sigplan Notices*. [S.l.], 2009. v. 44, n. 7, p. 1–10.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. [S.l.]: Pearson Education - Br, 2007.

TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, v. 100, n. Special Centennial Issue, p. 1411–1430, May 2012. ISSN 0018-9219.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas Digitais Princípios e Aplicações*. [S.l.]: Pearson, 2007. ISBN 9788576050957.

TURKINGTON, K. et al. Outer loop pipelining for application specific datapaths in fpgas. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, v. 16, n. 10, p. 1268–1280, Oct 2008. ISSN 1063-8210.

TYSON, G.; SMELYANSKIY, M.; DAVIDSON, E. Evaluating the use of register queues in software pipelined loops. *Computers, IEEE Transactions*, v. 50, n. 8, p. 769–783, Aug 2001. ISSN 0018-9340.

VAHID, F. *Digital Design with RTL Design, Verilog and VHDL*. [S.l.]: John Wiley and Sons, 2009. ISBN 9780470531082.

VENKATA, S. K. et al. Sd-vbs: The san diego vision benchmark suite. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium*. [S.l.: s.n.], 2009. p. 55–64.

WEINHARDT, M.; KRIEGER, A.; KINDER, T. A framework for pc applications with portable and scalable fpga accelerators. In: *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference*. Cancun, Mexico: [s.n.], 2013. p. 1–6.

XILINX. 2014. URL:http://www.xilinx.com. Último acesso em: 02/09/2014.

ZALAMEA, J. et al. Software and hardware techniques to optimize register file utilization in vliw architectures. *International Journal of Parallel Programming*, Springer, v. 32, n. 6, p. 447–474, 2004.

GLOSSÁRIO

ADRES – *Architecture for Dynamically Reconfigurable Embedded Systems*

AES – *Advanced Encryption Standard*

ALU – *Arithmetic Logic Unit*

API – *Application Programming Interface*

ASIC – *Application Specific Integrated Circuit*

CDFG – *Control and Data Flow Graph*

CDG – *Control Dependence Graph*

CFG – *Control Flow Graph*

CGRAs – *Coarse-Grained Reconfigurable Architectures*

CLB – *Configuration Logical Blocks*

CPLD – *Complex Programmable Logic Device*

CPU – *Central Processing Unit*

CP – *Controle Predicado*

DDG – *Data Dependence Graph*

DFG – *Data Flow Graph*

EDA – *Electronic Design Automation*

EEPROM – *Electrically Erasable Programmable Read Only Memory*

EPROM – *Erasable Programmable Read Only Memory*

FIFO – *First In, First Out*

FPGA – *Field Programmable Gate Array*

FSM – *Finite State Machine*

GAL – *Generic Array Logic*

GPU – *Graphics Processing Unit*

HDL – *Hardware Description Language*

HDTV – *High Definition TV*

HLL – *High Level Languages*

HLS – *High Level Synthesis*

IEEE – *Institute of Electrical and Electronics Engineers*

II – *Initiation Interval*

ILP – *Instruction Level Parallelism*

IOB – *Input/Output Block*

LALP – *Language for Aggressive Loop Pipelining*

LSB – *Least Significant Bit*

LUT – *Look up Table*

MCHPSO – *Modulo-Constrained Hybrid Particle Swarm*

MII – *Minimum Initiation Interval*

MPGA – *Mask Programmable Gate Array*

MSB – *Most Significant Bit*

MVE – *Modulo Variable Expansion*

OpenMP – *Open Multi Processing*

PAL – *Programmable Array Logic*

PEs – *Processing Elements*

PLA – *Programmable Logic Array*

PLD – *Programmable Logic Device*

PSO – *Particle Swarm Optimization*

QRF – *Queued Register File*

RQs – *Register Queues*

RRB – *Rotating Register Base*

RRF – *Rotating Register File*

RRG – *Routing Resource Graph*

RTL – *Register Transfer Level*

RecMII – *Recurrence Minimum Initiation Interval*

ResMII – *Resource Minimum Initiation Interval*

SCORE – *Stream Computation Organized for Reconfigurable Execution*

SPICE – *Simulation Program with Integrated Circuit Emphasis*

SRAM – *Static Random Access Memory*

TAG – *Target Architecture Graph*

VHDL – *VHSIC Hardware Description Language*

VHSIC – *Very High Speed Integrated Circuits*

VLIW – *Very Long Instruction Word*

VLSI – *Very Large Scale Integration*

WDFG – *Weighted Data Flow Graph*

mif – *Memory Initialization File*

Apendice A

LINGUAGEM DE DESCRIÇÃO DE *Hardware* VERILOG

Neste Apêndice, é feita uma introdução à linguagem de descrição de *hardware* Verilog e máquinas de estados.

A.1 Tipos de dados e Operadores

Quatro valores básicos são usados na maioria dos tipos de dados em Verilog:

- 0 : para nível lógico baixo, ou condição falsa.
- 1 : para nível lógico alto, ou condição verdadeira.
- z : para estado de alta impedância.
- x: para valor desconhecido.

O valor de z corresponde à saída de um *buffer tri-state*. O valor x é normalmente usado em modelagem e simulação, o que representa um valor que não é 0, 1, ou z, como uma entrada não inicializada ou conflito de saída (CHU, 2008). Os tipos de dados podem ser divididos em *net* e variável. Os tipos de dados no grupo *net*, representam as conexões físicas entre os componentes em *hardware*. Esses tipos de dados são utilizados em circuitos combinacionais e na conexão de sinais entre diferentes módulos em *hardware*. O tipo de dados *wire*, é o tipo mais comum neste grupo, que representa um fio de conexão. O tipo *wire* pode representar uma sinal de 1-*bit* ou uma coleção de sinais agrupadas em um barramento de dados, isso pode ser representado utilizando-se um vetor, exemplos são mostrados na Listagem A.1.

Listagem A.1: Formas de declarar o tipo de dados *wire*. Adaptado de Chu (2008).

```
1 wire p0, p1;                  // dois sinais de 1-bit cada
2 wire [7:0] data1, data2;      // dois sinais 8-bits
3 wire [31:0] addr;             // sinal de 32-bits
4 wire [0:7] revers_data;       // indice ascendente
```

A faixa de índice pode ser descendente (como em [7:0]) ou ascendente (como em [0:7]). No índice descendente, a posição mais à esquerda do vetor corresponde ao *bit* mais significativo ou MSB(*Most Significant Bit*), e no índice ascendente a posição mais a esquerda corresponde ao *bit* menos significativo ou MSB (*Most Significant Bit*) LSB (*Least Significant Bit*) (CHU, 2008).

Os tipos de dados no grupo de variáveis representam um valor que deve ser armazenado. Existem cinco tipos de dados nesse grupo: *reg*, *integer*, *real*, *time*, e *realtime*. O tipo de dados geralmente mais usados é o *reg*. Ao utilizar uma variável em Verilog a mesma ao ser sintetizada pode ou não ser convertida em componentes de armazenamento físico. Os três últimos tipos de dados só podem ser utilizados em simulações. O tipo de variável *integer* funciona como uma variável *reg* com tamanho de 32 *bits* (CHU, 2008).

Os valores em Verilog podem ser representados em vários formatos, sua forma geral é representada pela Figura A.1.

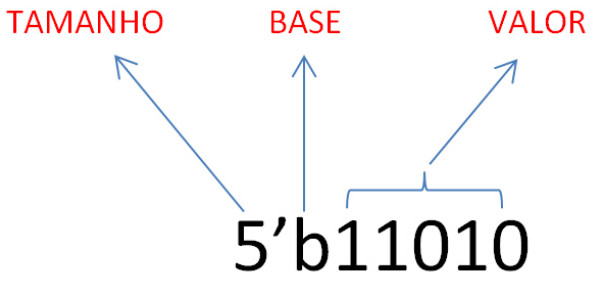

Figura A.1: Representação de um valor em Verilog.
Adaptado de Brown e Vranesic (2003).

A seta que indica a palavra "base" especifica a base do número, que pode ser:

- b ou B: binário.
- o ou O: octal.
- h ou H: hexadecimal.
- d ou D: decimal.

A seta que indica a palavra "valor" especifica o número na base correspondente e a seta que indica a palavra "tamanho" especifica o tamanho do número em *bits*.

O Verilog possui diversos operadores. Para a descrição de portas lógicas *bit* a *bit*, são utilizados os seguintes operadores: $\sim$ (NOT), & (AND), | (OR) e$\hat{}$(XOR). Para operações de deslocamento de *bits* , podem ser utilizados os operadores $\gg$ (deslocamento para a direita) e $\ll$ (deslocamento para a esquerda). Também existem operadores aritméticos no Verilog, que são: $+$,$-$,$*$,$/$,$\%$ e $**$. Eles representam adição, subtração, multiplicação, divisão, módulo e exponenciação. Os operadores $+$ e $-$ são convertidos em somadores e subtratores em *hardware* durante a síntese. O operador $*$ pode ser transformado em um multiplicador, dependendo do *software* de síntese utilizado. Os operadores $/$,$\%$ e $**$ geralmente não podem ser sintetizados automaticamente (CHU, 2008). No Quartus II foi testado o operador $*$, que foi sintetizado normalmente, sendo convertido para um circuito dedicado da placa DE-2, seu uso será melhor discutido no Capítulo 3. Os operadores $/$, $\%$ e $**$ também foram testados no Quartus II, sendo sintetizados normalmente.

A.2 Corpo do programa

Um código em Verilog é composto de três partes: a declaração das portas de I/O, declaração dos sinais, e o corpo do módulo. Cada programa criado em Verilog é denominado módulo, vários módulos podem ser criados e utilizados por outros módulos, assim como em um computador onde a CPU utiliza a memória e várias outras partes do computador. Os módulos são conectados entre si pelas portas de entrada e saída. A sintaxe para criação de portas de I/O é mostrada pela Figura A.2 .

A seta que indica a palavra "modo" especifica se o sinal será do tipo *input*, *output* ou *inout*. Esses tipos especificam se o sinal será de entrada, saída, ou bidirecional, respectivamente.

Em um programa na linguagem C as instruções são executadas sequencialmente, já o corpo de um programa de um módulo em Verilog, pode ser pensado como uma coleção de partes de um circuito. Essas partes são operadas em paralelo e executadas simultaneamente. Existem várias maneiras de descrever essas partes :

- *Continuous assignment* (atribuição contínua);
- Bloco *always*;
- Instanciação de Módulo

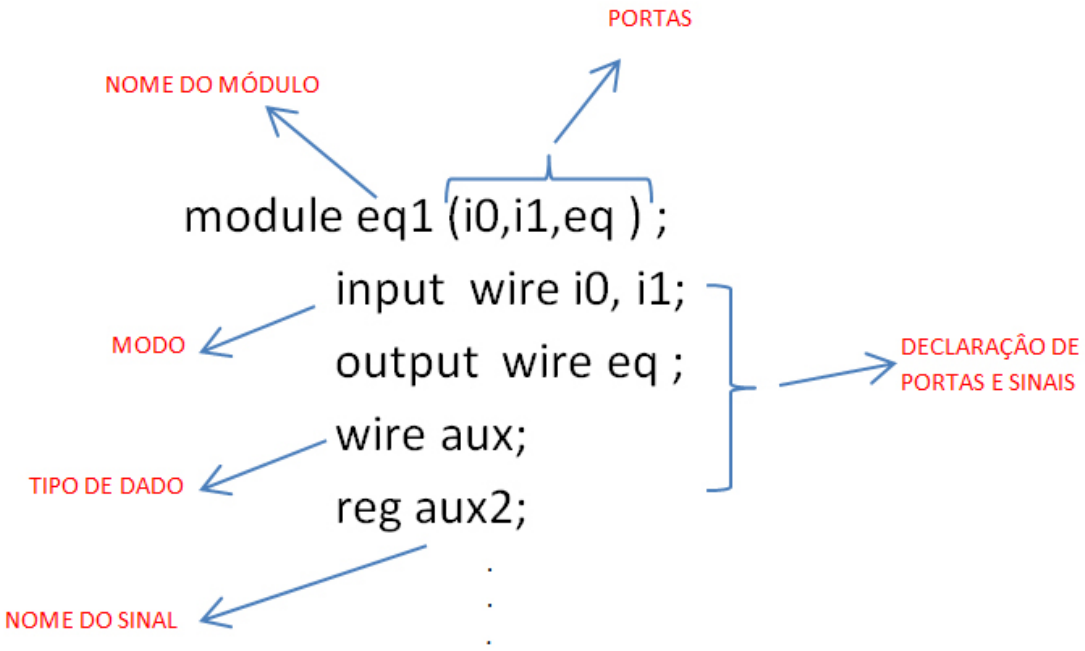

Figura A.2: Declaração de um módulo em Verilog.
Adaptado de Brown e Vranesic (2003).

A atribuição contínua é útil para descrever simples circuitos combinacionais. A Figura A.3 mostra a sintaxe de como descrever esses circuitos.

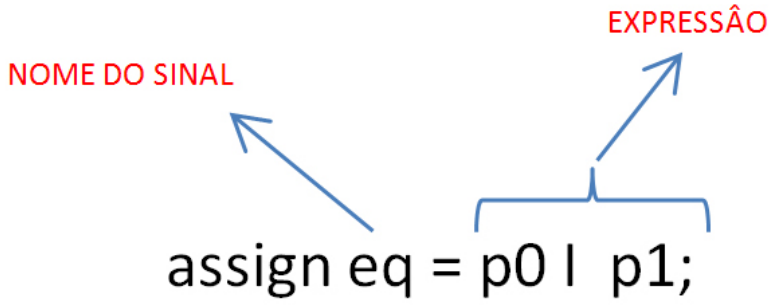

Figura A.3: Sintaxe da atribuição contínua.
Adaptado de Brown e Vranesic (2003).

O circuito da Figura A.3 irá realizar a operação OR. Quando $p0$ ou $p1$ mudam seus valores, o circuito é ativado e a expressão é validada. O novo valor é atribuído para *eq* após a propagação do sinal. A Listagem A.2 mostra um exemplo de um circuito simples de um comparador de 1-*bit* utilizando atribuição contínua. A expressão lógica para o circuito é :

$$eq = i0 \cdot i1 + i0' \cdot i1'$$

Listagem A.2: Comparador de 1-bit em verilog. Adaptado de Chu (2008).

```
1  // declaração do módulo
2  module eq1 ( i0, i1, eq );
3
4  //declaração das portas
5  input wire i0, i1;
6  output wire eq;
7
8  // declaração dos sinais
9  wire p0, p1;
10
11 //corpo
12 assign eq = p0 | p1;
13 assign p0 = ~i0 & ~i1;
14 assign p1 = i0 & i1;
15
16 endmodule
```

A Figura A.4 é uma representação gráfica do circuito da Listagem A.2, e a Figura A.5 é uma representação do circuito utilizando portas lógicas.

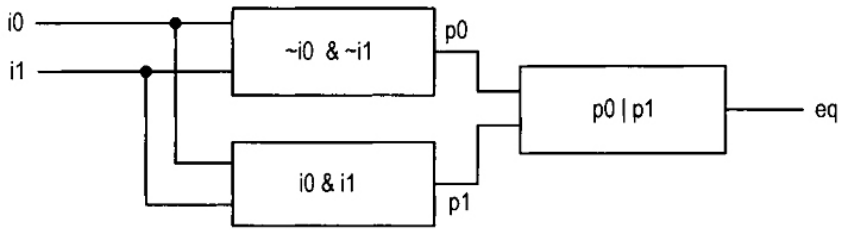

Figura A.4: Comparador de 1-*bit* em alto nível.
Fonte: Brown e Vranesic (2003).

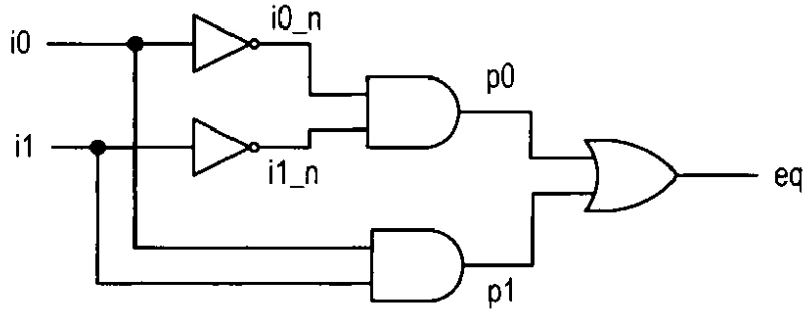

Figura A.5: Comparador de 1-*bit* em baixo nível.
Fonte: Brown e Vranesic (2003).

Na Listagem A.2 a ordem das declarações das expressões não interfere no circuito final .

Em Verilog não é necessário que o tipo do sinal seja declarado. Se o tipo for omitido, então o sinal assume o tipo *wire*. Portanto, na Listagem A.2 se os tipo dos sinais não forem declarados como *wire* não ocorreria nenhum problema.

Um sistema digital frequentemente é composto de várias partes menores. O Verilog proporciona a capacidade de construir um sistema complexo a partir de vários componentes simples, através do uso de módulos. Uma alternativa para se criar um comparador de 2-*bits* é através do uso de componentes em módulos, para isso será instanciado o comparador de 1-*bit* que foi mostrado na Listagem A.2. A Listagem A.3 mostra um exemplo de como criar um comparador de 2-*bits*, e a Figura A.6 mostra uma representação gráfica da Listagem A.3.

Listagem A.3: Comparador de 2-*bits* em Verilog. Adaptado de Chu (2008).

```
1  module eq2 (a,b,aeqb);
2
3  input  wire[1:0] a, b;
4  output wire aeqb;
5  wire e0,  e1;
6
7  //  instanciando dois comparadores de 1-bit
8  eq1 eq_bit0_unit  (.i0(a[0]) , .i1 (b[0]) , .eq(e0)) ;
9  eq1 eq_bit1_unit  (.eq(e1), .i0(a[1]),  .i1(b[1])) ;
10
11 //a e b são iguais se seus bits individuais são iguais
12 assign aeqb = e0  &  e1;
13 endmodule
```

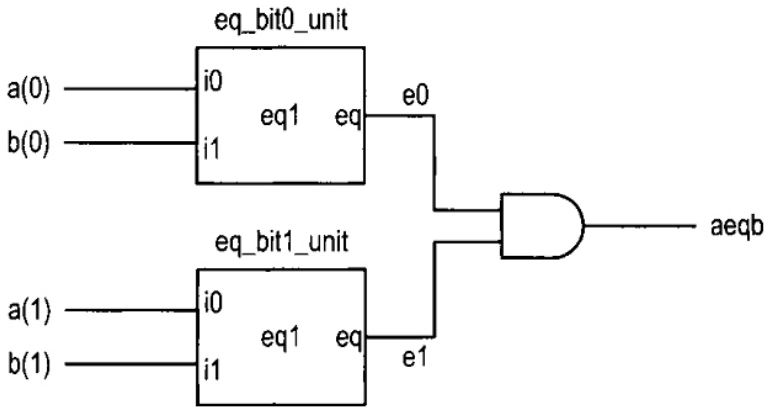

Figura A.6: Construção de um comparador de 2-*bits* utilizando dois comparadores de 1-*bit*.
Fonte: Brown e Vranesic (2003).

Na Listagem A.3 para se instanciar um módulo, primeiro é necessário mostrar qual módulo está sendo instanciado indicando seu nome, no caso *eq1*. Em seguida são dados os nomes

das instâncias, no caso *eq_bit0_unit* e *eq_bit1_unit*. Por fim, são especificadas as conexões entre as portas de entrada e saída entre o módulo instanciado *eq1* e os sinais utilizados no módulo corrente *eq2*. Essa forma de mapeamento de portas é conhecida como conexão por nome. Utilizando essa forma de conexão a ordem em que as portas são escritas no código não importa (CHU, 2008).

A.3 Bloco *Always*

Um bloco *always* pode ser considerado como uma caixa preta, cujo comportamento é descrito pelas instruções procedurais internas. Essas instruções procedurais incluem uma rica variedade de construtores, mas muitos deles não têm uma representação equivalente em *hardware*. A Listagem A.4 mostra a sintaxe de um bloco *always*.

Listagem A.4: Sintaxe do bloco *always*. Adaptado de Chu (2008).

```
1 always @([lista de sensibilidade])
2 begin [nome (opcional)]
3   [declaração de variáveis (opcional) ];
4   [instrução procedural];
5   [instrução procedural] ;
6       .
7       .
8       .
9 end
```

A lista de sensibilidade da Listagem A.4 é uma lista de sinais e eventos que ativam o bloco *always*. Para circuitos combinacionais, todos os sinais de entrada devem ser incluídos nessa lista. O corpo do bloco *always* é composto de várias instruções procedurais. O *begin* e *end* podem ser omitidos se existir apenas uma instrução procedural. A lista de sensibilidade é o único controle temporal sintetizável em um bloco *always*. Um bloco *always* pode ser considerado como uma parte complexa do circuito. Ele pode ser suspenso ou ativado. Quando algum sinal da lista de sensibilidade muda seu valor ou algum evento ocorre, o bloco *always* é ativado e executa as instruções procedurais internas. Uma vez que não existe nenhuma outra estrutura de controle temporal, a sua execução continuará até o fim. Assim, um bloco *always* se parece com um "*loop* eterno" e o inicio de cada *loop* é controlado pela lista de sensibilidade (CHU, 2008).

Existem dois tipos de atribuições que podem ser utilizadas no bloco *always*: atribuições de bloqueio (*blocking assignment*) e atribuições de não bloqueio (*nonblocking assignment*)(CHU, 2008). Quando são utilizadas as atribuições de bloqueio, o compilador Verilog executa as

instruções no bloco *always* na ordem em que foram escritas. Se uma variável está recebendo um valor calculado por uma atribuição de bloqueio, então o seu novo valor é que será utilizado no cálculo de todas as instruções subsequentes que utilizarem essa variável (BROWN; VRANESIC, 2003). Esse tipo de atribuição se comporta como a atribuição de uma variável na linguagem C (CHU, 2008). As atribuições de não bloqueio utilizam o valor que as variáveis possuíam quando a execução do bloco *always* foi iniciada. Assim, uma variável possui o mesmo valor para todas as instruções no bloco. O resultado de cada atribuição não será visto até o fim da execução do bloco *always* (BROWN; VRANESIC, 2003). Resumindo, a ordem em que as instruções foram escritas é significante quando se utiliza atribuições de bloqueio, e quando se utiliza atribuições de não bloqueio não existe sensibilidade na ordem em que as instruções foram escritas (COFFMAN, 2000). A Listagem A.5 mostra um exemplo do uso de atribuições de bloqueio e o circuito correspondente ao código é mostrado pela Figura A.7. A Listagem A.6 mostra o uso de atribuições de não bloqueio e o circuito correspondente ao código é mostrado pela Figura A.8.

Listagem A.5: Exemplo do uso de atribuições de bloqueio. Fonte: Brown e Vranesic (2003).

```
1  module exemplo_blocking (x1, x2, x3, Clock, f, g);
2  input x1, x2, x3, Clock;
3  output f, g;
4  reg f, g;
5  always @(posedge Clock)
6  begin
7      f = x1 & x2;
8      g = f | x3;
9  end
10 endmodule
```

Listagem A.6: Exemplo do uso de atribuições de não bloqueio. Fonte: Brown e Vranesic (2003).

```
1  module exemplo_nonblocking (x1, x2, x3, Clock, f, g);
2  input x1, x2, x3, Clock;
3  output f, g;
4  reg f, g;
5  always @(posedge Clock)
6  begin
7      f <= x1 & x2;
8      g <= f | x3;
9  end
10 endmodule
```

Como pode ser visto na Listagem A.5, para realizar atribuições de bloqueio basta utilizar o operador = para atribuir valores às variáveis. Para realizar atribuições de não bloqueio basta utilizar o operador <=. Ao comparar a Figura A.7 à Figura A.8, fica claro a diferença entre os dois tipos de atribuição. A figura A.7 mostra como as atribuições de bloqueio são sensíveis a ordem de declaração das instruções, pois o valor de f é calculado e logo em seguida é utilizado no cálculo de g dando a ideia de um programa sequencial, igual à linguagem C. Já a Figura A.8 mostra como as atribuições de não bloqueio não são sensíveis à ordem de declaração das instruções, pois f e g são independentes, dando ideia de um programa que executa as instruções em paralelo, igual programação paralela ou *threads*.Quando é utilizada a subida de borda do *clock* na lista de sensibilidade, o sintetizador entende que as variáveis devem ser sensíveis ao *clock* e serão vistas como *flip-flops* do tipo D. Podem ser criados vários *always* dentro de um mesmo módulo em Verilog.

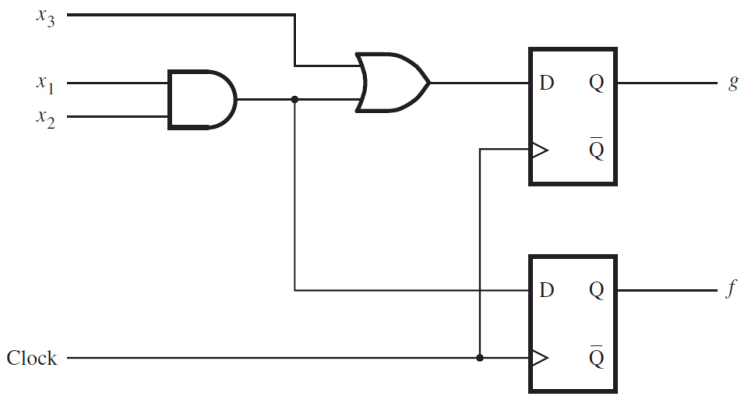

Figura A.7: Circuito correspondente ao código da Listagem A.5.
Fonte: Brown e Vranesic (2003).

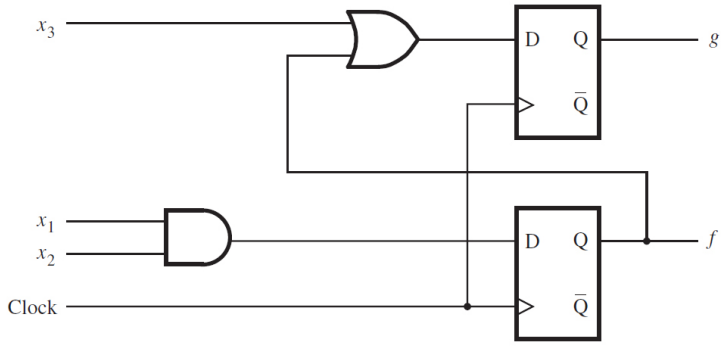

Figura A.8: Circuito correspondente ao código da Listagem A.6.
Fonte: Brown e Vranesic (2003).

A Listagem A.7 mostra como o comparador de 1-*bit* da Listagem A.2 pode ser reescrito utilizando o bloco *always* e atribuições de bloqueio. Diferente das Listagens A.5 e A.6, a lista

de sensibilidade não é sensível ao *clock* e sim às mudanças nos sinais de entrada. Dentro de um bloco *always*, a atribuição de valores pode ser feita apenas aos dados do tipo variável, como *reg* ou *integer*. Visto que valores são atribuídos a *eq*, $p0$, e $p1$, eles devem ser declarados como tipos de dados *reg*. A lista de sensibilidade possui os sinais $i0$ e $i1$, que são separados por vírgula. As três atribuições de bloqueio são executadas sequencialmente. A ordem das instruções é importante, pois $p0$ e $p1$ precisam de receber seus novos valores antes de *eq* (CHU, 2008).

Listagem A.7: Implementação de um comparador de 1-bit utilizando atribuições de bloqueio. Fonte: Chu (2008).

```
1  module eql_always(i0, i1 , eq );
2
3  input wire i0, i1 ;
4  output reg eq ; // eq declared as reg
5  reg p0, p1; // p0 e p1 são declarados como reg
6
7  always @(i0, i1) // i0 e i1 devem estar na lista de sensibilidade
8  begin
9          // a ordem das instruções é importante
10         p0 = ~i0 & ~i1;
11         p1 = i0 & i1;
12         eq = p0 | p1 ;
13 end
14 endmodule
```

A.4 *If* e *Case*

A sintaxe de um *if* em Verilog é mostrada pala Listagem A.8.

Listagem A.8: Sintaxe da cláusula *if*. Fonte: Chu (2008).

```
1  if [expressão_booleana]
2          begin
3          [instrução procedural] ;
4          [instrução procedural] ;
5          . . .
6
7          end
8
9  else
10
11         begin
```

```
12          [instrução procedural] ;
13          [instrução procedural] ;
14          . . .
15
16          end
```

Na Listagem A.8 a primeira expressão booleana é verificada. Se for verdadeira, as instruções dentro do *if* serão executadas. Senão, serão executadas as instruções dentro do *else*. O *else* é opcional e pode ser omitido. O *begin* e *end* podem ser omitidos se existir apenas uma instrução procedural. Um exemplo do uso do *if* é mostrado na Listagem A.9 , onde é criado um multiplexador de 4 para 1. As cláusulas *if-else* setam o valor de f com um dos valores das entradas $w0, ..., w3$ dependendo do valor de S.

Listagem A.9: Código de um multiplexidor de 4 para 1 utilizando *if-else*. Fonte: Brown e Vranesic (2003).

```
1  module mux4to1 (w0, w1, w2, w3, S, f);
2  input w0, w1, w2, w3;
3  input [1:0] S;
4  output f;
5  reg f;
6
7  always@(w0 or w1 or w2 or w3 or S)
8          if(S == 2'b00)
9                  f = w0;
10         else if(S == 2'b01)
11                 f = w1;
12         else if(S == 2'b10)
13                 f = w2;
14         else if(S == 2'b11)
15                 f = w3;
16 endmodule
```

A sintaxe de um *case* em Verilog é mostrada pala Listagem A.10.

Listagem A.10: Sintaxe da cláusula *case*. Fonte: Chu (2008).

```
1  case [expressão_case]
2          [item] :
3                  begin
4                  [instrução procedural] ;
5                  [instrução procedural] ;
6                  . . .
7                  end
```

```
8           [item] :
9                   begin
10                  [instrução procedural] ;
11                  [instrução procedural] ;
12                  . . .
13                  end
14          [item] :
15                  begin
16                  [instrução procedural] ;
17                  [instrução procedural] ;
18                  end
19                  . . .
20          default :
21                  begin
22                  [instrução procedural] ;
23                  [instrução procedural] ;
24                  end
25  endcase
```

O *case* é uma cláusula de decisão que compara uma expressão com um número de itens. A execução pula para o trecho em que o item possui o mesmo valor da expressão. Se muitos itens possuem o mesmo valor da expressão, a execução pula para o primeiro item a ter o mesmo valor da expressão. O item *default* é opcional. O *begin* e *end* podem ser omitidos se existir apenas uma instrução procedural. A ListagemA.11 mostra um exemplo de um multiplexador 4 para 1 utilizando *case*.

Listagem A.11: Código de um multiplexidor de 4 para 1 utilizando *case*. Fonte: Brown e Vranesic (2003).

```
1   module mux4to1Case (W, S, f);
2   input [0:3] W;
3   input [1:0] S;
4   output f;
5   reg f;
6   always@(W or S)
7           case(S)
8                   0: f = W[0];
9                   1: f = W[1];
10                  2: f = W[2];
11                  3: f = W[3];
12          endcase
13  endmodule
```

A.5 Máquinas de Estados Finitos

A maioria dos circuitos mostrados anteriormente eram circuitos combinacionais, agora será mostrada uma classe de circuitos em que a saída depende do estado anterior do circuito, assim como os valores atuais das entradas. Eles são chamados circuitos sequenciais. Em muitos casos um sinal de *clock* é utilizado para controlar as operações do circuito sequencial, esse tipo de circuito que utiliza *clock* é chamado circuito sequencial síncrono. Os circuitos que não utilizam *clock* são chamados de circuitos sequencias assíncronos. Os circuitos sequenciais utilizam circuitos combinacionais e um ou mais *flip-flops*. A estrutura básica de um circuito sequencial é mostrada na Figura A.9. O circuito possui a entrada W, e produz a saída Z. Os valores das saídas dos *flip-flops* são chamados de estados (Q) do circuito. Sobre o controle do *clock*, as saídas dos *flip-flops* mudam seus estados que são determinados pelos circuitos combinacionais que alimentam as entradas dos *flip-flops*. Assim, o circuito muda de um estado para outro. Para assegurar que somente uma transição de um estado para outro ocorra durante o ciclo de *clock*, os *flip-flops* precisam ser sensíveis à borda de subida do *clock*. Eles podem ser ativados pela transição positiva (transição de 0 para 1) ou negativa (transição de 1 para 0). Os sinais de entrada dos *flip-flops* são derivados de duas origens: os valores das entradas, W, e o estado presente das saídas dos *flip-flops*, Q. Assim, a mudança do estado depende do estado presente e dos valores das entradas. No entanto, as saídas sempre dependem do estado atual, mas não necessariamente precisam depender diretamente das entradas. Portanto, a conexão mostrada em vermelho na Figura A.9, pode ou não existir. Para distinguir entre essas duas possibilidades, é comum dizer que circuitos sequencias em que as saídas dependem somente do estado atual do circuito são do tipo *Moore*, enquanto aqueles circuitos em que as saídas dependem do estado atual e dos valores de entrada são do tipo *Mealy* (BROWN; VRANESIC, 2003).

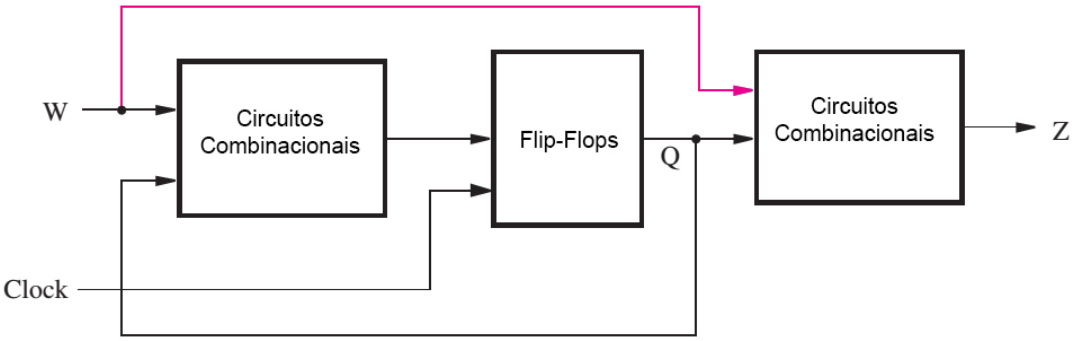

Figura A.9: Forma geral de circuito sequencial.
Adaptado de Brown e Vranesic (2003).

Os circuitos sequenciais também podem ser chamados de máquinas de estados finitos.

Esse nome é utilizado pelo fato de que, esses circuitos podem ser representados utilizando um número finito de estados. Um exemplo de uma máquina de estados é mostrada pela Figura A.10 através de um diagrama de estados. Nesse exemplo é mostrada uma máquina de estados do tipo *Moore* que possui os estados A, B e C cada um representado por um nó no diagrama de estados. O nó A representa o estado inicial, e também o estado em que o circuito vai permanecer após a entrada $w = 0$ ser aplicada. Nesse estado a saída z é 0, o que é indicado por $A/z = 0$ dentro do nó. O circuito irá permanecer no estado A enquanto $w = 0$, o que é representado pelo arco que começa e termina no próprio nó A. Quando $w = 1$ o estado irá mudar de A para o estado B. Essa transição é indicada pelo arco que sai do nó A e vai para o nó B. No estado B a saída permanece com o valor 0 que é representado por $B/z = 0$ dentro do nó . Quando o circuito está no estado B, ele irá mudar para o estado C se w for igual a 1 no próximo ciclo de *clock*. No estado C a saída z terá o valor 1. Se w permanecer com o valor 1 durante os próximos ciclos de *clock*, o circuito irá permanecer no estado C mantendo $z = 1$. No entanto, se o valor de w mudar para 0 o circuito estando no estado C ou B, irá mudar para o estado A no próximo ciclo de *clock*(BROWN; VRANESIC, 2003).

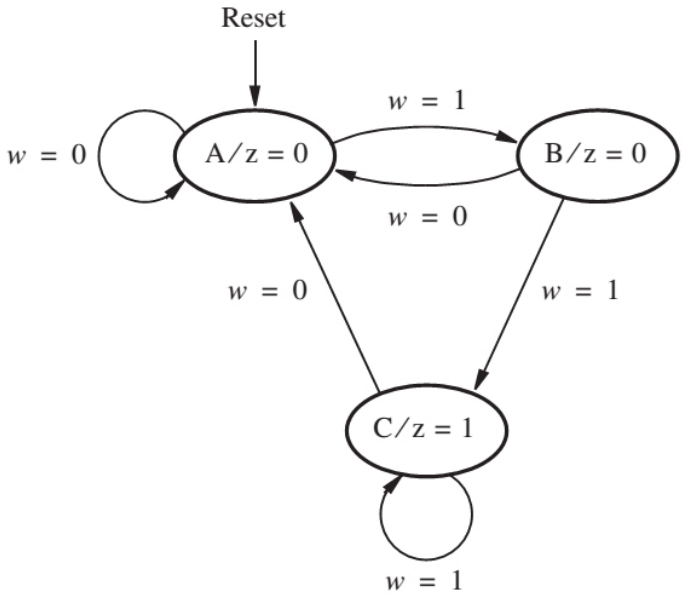

Figura A.10: Diagrama de estados de uma máquina de *Moore*.
Fonte: Brown e Vranesic (2003).

Ciclos de clock:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figura A.11: Saída da máquina de estados da Figura A.10 em relação a t
Adaptado de Brown e Vranesic (2003).

Uma forma de implementar essa máquina de estados em Verilog é mostrada na Listagem

A.12. Nessa listagem são especificados dois blocos *always* separados. O primeiro bloco descreve o circuito combinacional. O estado do circuito é controlado pelas variáveis *y* que é o estado atual, e *Y* que é o próximo estado. O valor do vetor do próximo estado *Y* muda quando os valores dos sinais *y* e *w* mudam. O *case* irá determinar o valor de *Y* em relação aos valores de *y* e *w*. O segundo *always* insere os *flip-flops* ao circuito. Sua lista de sensibilidade é ativada pela mudança de valor do *reset* e do *clock*. O *reset* irá setar *A* como estado inicial. Outra forma de descrever essa máquina de estados é mostrada pela Listagem A.13 onde apenas um bloco *always* é utilizado (BROWN; VRANESIC, 2003). A saída do circuito é mostrada pela Figura A.11.

Listagem A.12: Código em Verilog para a máquina de estados da Figura A.10. Fonte: Brown e Vranesic (2003).

```
1  module MooreTipo1 (Clock, Resetn, w, z);
2  input Clock, Resetn, w;
3  output z;
4  reg [2:1] y, Y;
5
6  parameter[2:1] A = 2'b00, B = 2'b01, C = 2'b10;
7
8  // Circuito combinacional para o próximo estado
9
10 always@(w or y)
11         case(y)
12                 A:      if(w) Y = B;
13                         else   Y = A;
14                 B:      if(w) Y = C;
15                         else   Y = A;
16                 C:      if(w) Y = C;
17                         else   Y = A;
18                 default: Y = 2'bxx;
19 endcase
20
21 // Circuito sequencial
22
23 always@(negedge Resetn , posedge Clock)
24   if (Resetn == 0) y<=A;
25   else y<=Y;
26
27 // Saída
28 assign z=(y==C);
```

Listagem A.13: Outro modo de implementar a máquina de estados da Figura A.10. Fonte: Brown e Vranesic (2003).

```
1  module MooreTipo2 (Clock, Resetn, w, z);
2  input Clock, Resetn, w;
3  output z;
4  reg [2:1] y;
5  parameter[2:1] A = 2'b00, B = 2'b01, C = 2'b10;
6
7
8  //Bloco Sequencial
9  always@(negedge Resetn, posedge Clock)
10         if (Resetn == 0) y<=A;
11         else
12                 case(y)
13                         A: if(w) y<=B;
14                                 else   y<=A;
15                         B: if(w) y<=C;
16                                 else   y<=A;
17                         C: if(w) y<=C;
18                                 else   y<=A;
19                         default: y<=2'bxx;
20 endcase
21
22
23
24 // Saída
25 assign z=(y==C);
26 endmodule
```

A máquina de *Moore* da Figura A.10 pode ser alterada para funcionar como uma máquina de *Mealy*. Essa outra forma de descrever a máquina é mostrada pelo exemplo na Figura A.12. Nesse exemplo, a máquina irá possuir apenas dois estados. A máquina inicia no estado A e irá continuar no estado se $w = 0$ e a saída será 0. Isso é indicado pelo arco $w = 0/z = 0$. Quando w assume o valor 1 a saída será 0 até a máquina mudar para o estado B no próximo ciclo de *clock*. Essa transição é mostrada pelo arco de $w = 1/z = 0$. No estado B a saída será 1 se $w = 1$, e a máquina de estados irá continuar no estado B, o que é indicado pelo arco $w = 1/z = 1$. No entanto se $w = 0$ no estado B, então a saída será 0 e ocorrerá uma transição para o estado A no próximo ciclo de *clock*. Um ponto importante é que no ciclo de *clock* atual a saída corresponde ao valor descrito em cima do arco de transição do estado presente. A Listagem A.14 mostra o circuito para essa máquina de estados em Verilog, e a Figura A.13 a saída do circuito.

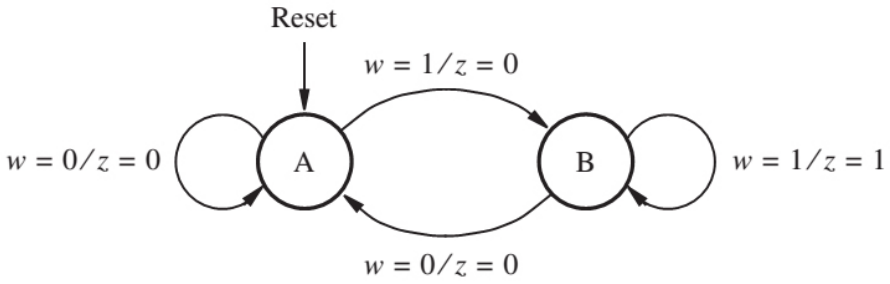

Figura A.12: Diagrama de estados de uma máquina de *Mealy*.
Adaptado de Brown e Vranesic (2003).

Ciclos de clock:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

Figura A.13: Saída da máquina de estados da Figura A.12 em relação a t.
Adaptado de Brown e Vranesic (2003).

Listagem A.14: Código em Verilog para a máquina de estados da Figura A.12. Fonte: Brown e Vranesic (2003).

```
1  module MaquinaMealy (Clock, Resetn, w, z);
2  input Clock, Resetn, w;
3  output z;
4  reg y, Y, z;
5  parameter A=0,B=1;
6
7  //Circuito combinacional
8  always@(w or y)
9          case(y)
10                 A: if(w)
11                                 begin
12                                 z=0;
13                                 Y=B;
14                                 end
15                         else
16                                 begin
17                                 z=0;
18                                 Y=A;
19                                 end
20                 B: if(w)
21                                 begin
22                                 z=1;
23                                 Y=B;
24                                 end
```

```
25                          else
26                                  begin
27                                  z=0;
28                                  Y=A;
29                                  end
30          endcase
31
32  // Circuito Sequencial
33  always@(negedge Resetn, posedge Clock)
34          if (Resetn == 0) y<=A;
35          else y<=Y;
36
37  endmodule
```

Comparando as Figuras A.11 e A.13 das saídas dos circuitos, pode ser visto que a mudança de 0 para 1 da saída z na Figura A.11 ocorre em $t5$ e na Figura A.12 ocorre em $t4$. Isso acontece devido à mudança do valor de saída z na máquina *Mealy* ocorrer no mesmo estado em que a entrada w muda seu valor para 1. Já na máquina de *Moore* a mudança irá ocorrer somente no próximo estado, ou seja, no próximo ciclo de *clock*.

Apendice B

SAÍDAS DO CETUS MODIFICADO EM VERILOG

As Listagens B.1 e B.2 mostram as possíveis implementações em Verilog da Listagem 2.6. As Listagens B.3 e B.4 mostram as possíveis implementações em Verilog da Listagem 2.7. A Figura B.1 mostra o resultado do código em C da Listagem 2.5, e as Figuras B.2 e B.3 mostram os resultados das Listagens B.1 e B.2.

Listagem B.1: Implementação da *FSM-Target Machine 1* da Listagem 2.6 em Verilog.

```
1
2  module FSM_Target1_Tipo1(CLK, enable, reset);
3          input           CLK;
4          input           enable;
5          input           reset;
6
7
8          wire         CLK;
9          wire         enable;
10         integer      i, a, b, c, k, s, n;
11         wire         reset;
12
13         // DefiniÃ§Ã£o dos estados
14         `define Inicio       4'b0000
15         `define Prologo1     4'b0001
16         `define Prologo2     4'b0010
17         `define Prologo3     4'b0011
18         `define Kernel       4'b0100
19         `define Incremento_I 4'b0101
20         `define Epilogo1     4'b0111
21         `define Epilogo2     4'b1000
22         `define Epilogo3     4'b1001
```

```
23          `define Fim           4'b1010
24
25          reg [3:0] Estado;
26
27          //
               ---------------------------------------------------------------------------

28          // Maquina de estados: LoopPipeliningSimples
29          //
               ---------------------------------------------------------------------------

30          //-----------------------------------------
31          always @ (posedge CLK or posedge reset)
32                  begin : LoopPipelining
33                          if (reset)      begin
34                           i = 0;
35                                  s = 0;
36                                  n = 8;
37                                  a = 0;
38                                  Estado = `Inicio;
39                                  end
40                          else
41                                  if(enable) begin
42                                    case (Estado)
43
44                                    `Inicio:
45                                    begin
46                                          Estado = `Prologo1;
47                                    end
48
49                                    `Prologo1:
50                                    begin
51                                          c = a+10;
52                                          Estado = `Prologo2;
53                                    end
54
55                                    `Prologo2:
56                                    begin
57                                          a = (c*2);
58                                          b = (c*5);
59                                          s = (s+c);
60                                          c = (a+10);
61                                          Estado = `Prologo3;
```

```
62                              end
63
64                              `Prologo3:
65                              begin
66                                    k = s;
67                                    a = (c*2);
68                                    b = (c*5);
69                                    s = (s+c);
70                                    c = (a+10);
71                                    Estado = `Kernel;
72                              end
73
74                              `Kernel:
75                              begin
76                                    s = (s+k);
77                                    k = s;
78                                    a = (c*2);
79                                    b = (c*5);
80                                    s = (s+c);
81                                    c = (a+10);
82                                    if (i>=n)
83                                      Estado = `Epilogo1;
84                                    else if (i<n)
85                                      Estado = `Incremento_I;
86                              end
87
88                              `Incremento_I:
89                              begin
90                                    i = i+1;
91                                    Estado = `Kernel;
92                              end
93
94                              `Epilogo1:
95                              begin
96                                    s = (s+k);
97                                    k = s;
98                                    a = (c*2);
99                                    b = (c*5);
100                                   s = (s+c);
101                                   Estado = `Epilogo2;
102                             end
103
104                             `Epilogo2:
```

```
105                                     begin
106                                           s = (s+k);
107                                           k = s;
108                                           Estado = `Epilogo3;
109                                     end
110
111                                     `Epilogo3:
112                                     begin
113                                           s = (s+k);
114                                           Estado = `Fim;
115                                     end
116
117                                     endcase
118                                   end
119                 end
120
121
122
123 endmodule
```

Listagem B.2: Outra implementação da *FSM-Target Machine 1* da Listagem 2.6 em Verilog.

```
1  module FSM_Target1_Tipo2(CLK, enable, i, a, b, c, k,s, reset);
2      input       CLK;
3      input       enable;
4      input       reset;
5      output      i, a, b, c, k, s;
6
7       wire        CLK;
8       wire        enable;
9       integer     i, a, b, c, k, s, n;
10      wire        reset;
11
12      // DefiniÃ§Ã£o dos estados
13     `define Inicio       4'b0000
14     `define Prologo1     4'b0001
15     `define Prologo2     4'b0010
16     `define Prologo3     4'b0011
17     `define Kernel       4'b0100
18     `define Incremento_I 4'b0101
19     `define Epilogo1     4'b0111
20     `define Epilogo2     4'b1000
21     `define Epilogo3     4'b1001
22     `define Fim          4'b1010
```

```
23
24      reg [3:0] Estado;
25
26      always @ (posedge CLK or posedge reset)
27      begin
28        if (reset)
29        begin
30         i = 0;
31         s = 0;
32         n = 7;
33         a = 0;
34         Estado = `Inicio;
35        end
36        else
37         if(enable)
38              begin
39
40         case (Estado)
41
42         `Inicio:
43         begin
44           Estado = `Prologo1;
45         end
46
47         `Prologo1:
48              begin
49           Estado = `Prologo2;
50         end
51
52         `Prologo2:
53         begin
54            Estado = `Prologo3;
55              end
56
57         `Prologo3:
58         begin
59            Estado =`Kernel;
60         end
61
62         `Incremento_I:
63         begin
64          Estado = `Kernel;
65         end
```

```
66
67          `Kernel:
68          begin
69          if (i>=n)
70            Estado = `Epilogo1;
71          else if (i<n)
72            Estado = `Incremento_I;
73          end
74
75          `Epilogo1:
76          begin
77            Estado = `Epilogo2;
78          end
79
80          `Epilogo2:
81          begin
82            Estado = `Epilogo3;
83          end
84
85          `Epilogo3:
86          begin
87            Estado = `Fim;
88          end
89
90        endcase
91      end
92    end
93
94      always @(Estado)
95      begin
96        case (Estado)
97
98        `Prologo1: c = a+10;
99
100       `Prologo2:
101       begin
102         a = (c*2);
103         b = (c*5);
104         s = (s+c);
105         c = (a+10);
106       end
107
108     `Prologo3:
```

```
109       begin
110         k = s;
111         a = (c*2);
112         b = (c*5);
113         s = (s+c);
114         c = (a+10);
115       end
116
117           `Kernel:
118        begin
119         s = (s+k);
120         k = s;
121         a = (c*2);
122         b = (c*5);
123         s = (s+c);
124         c = (a+10);
125        end
126
127        `Incremento_I:
128        begin
129         i = i+1;
130        end
131
132             `Epilogo1:
133        begin
134         s = (s+k);
135         k = s;
136         a = (c*2);
137         b = (c*5);
138         s = (s+c);
139        end
140
141        `Epilogo2:
142        begin
143         s = (s+k);
144               k = s;
145        end
146
147        `Epilogo3:
148        begin
149         s = (s+k);
150        end
151
```

```
152      endcase
153    end
154
155 endmodule
```

Listagem B.3: Implementação da *FSM-Target Machine 2* da Listagem 2.7 em Verilog.

```
1  module FSM_Target2_Tipo1(CLK, enable, i, a, b, c, k,s, reset);
2     input          CLK;
3     input          enable;
4     input          reset;
5     output         i, a, b, c, k, s;
6
7     wire          CLK;
8     wire          enable;
9     integer       i, a, b, c, k, s, n;
10    wire          reset;
11
12    // Definiï¿½ï¿½o dos estados
13    `define Inicio        4'b0000
14    `define Prologo1      4'b0001
15    `define Prologo2      4'b0010
16    `define Prologo3      4'b0011
17    `define Prologo4      4'b0100
18    `define Prologo5      4'b0101
19    `define Kernel1       4'b0110
20    `define Kernel2       4'b0111
21    `define Kernel3       4'b1000
22    `define Incremento_I  4'b1001
23    `define Epilogo1      4'b1010
24    `define Epilogo2      4'b1011
25    `define Epilogo3      4'b1100
26    `define Epilogo4      4'b1101
27    `define Epilogo5      4'b1110
28    `define Fim           4'b1111
29
30    reg [3:0] Estado;
31
32    //-------------------------------------------------------------------------
33    // Maquina de estados: LoopPipeliningSimples
34    //-------------------------------------------------------------------------
35    //---------------------------------------
36    always @ (posedge CLK or posedge reset)
37       begin : LoopPipelining
```

```
38          if (reset)   begin
39              i <= 0;
40              s <= 0;
41              n <= 7;
42              a <= 0;
43              Estado <= `Inicio;
44          end
45          else
46              if(enable) begin
47
48              case (Estado)
49                  `Inicio:
50                      begin
51                          Estado <= `Prologo1;
52                      end
53                  `Prologo1:
54                      begin
55                          c <= a+10;
56                          Estado <= `Prologo2;
57                      end
58                  `Prologo2:
59                      begin
60                          a <= (c*2);
61                          b <= (c*5);
62                          s <= (s+c);
63                          Estado <= `Prologo3;
64                      end
65                  `Prologo3:
66                      begin
67                        c <= (a+10);
68                          Estado <= `Prologo4;
69                      end
70                  `Prologo4:
71                      begin
72                        k <= s;
73                          a <= (c*2);
74                          b <= (c*5);
75                          s <= (s+c);
76                          Estado <= `Prologo5;
77                      end
78                      `Prologo5:
79                      begin
80                        c <= (a+10);
```

```
81                  Estado <= `Kernel1;
82              end
83          `Kernel1:
84              begin
85                  s <= (s+k);
86                 Estado <= `Kernel2;
87              end
88          `Kernel2:
89              begin
90                k <= s;
91                 a <= (c*2);
92                 b <= (c*5);
93                 s <= (s+c);
94                 Estado <= `Kernel3;
95              end
96          `Kernel3:
97              begin
98                c <= (a+10);
99                 if (i>=n)
100                    Estado <= `Epilogo1;
101                else if (i<n)
102                    Estado <= `Incremento_I;
103             end
104         `Incremento_I:
105             begin
106                 i <= i+1;
107                 Estado <= `Kernel1;
108             end
109         `Epilogo1:
110             begin
111                 s <= (s+k);
112                 Estado <= `Epilogo2;
113             end
114         `Epilogo2:
115             begin
116                 k <= s;
117                 a <= (c*2);
118                 b <= (c*5);
119                 s <= (s+c);
120                 Estado <= `Epilogo3;
121             end
122         `Epilogo3:
123             begin
```

```
124                         s <= (s+k);
125                         Estado <= `Epilogo4;
126                      end
127                   `Epilogo4:
128                      begin
129                         k <= s;
130                         Estado <= `Epilogo5;
131                      end
132                   `Epilogo5:
133                      begin
134                         s <= (s+k);
135                         Estado <= `Fim;
136                      end
137               endcase
138            end
139         end
140
141
142
143 endmodule
```

Listagem B.4: Outra implementação da *FSM-Target Machine 2* da Listagem 2.7 em Verilog.

```
1
2  module FSM_Target2_Tipo2(CLK, enable, i, a, b, c, k,s, reset);
3     input          CLK;
4     input          enable;
5     input          reset;
6     output         i, a, b, c, k, s;
7
8     wire          CLK;
9     wire          enable;
10    integer       i, a, b, c, k, s, n;
11    wire          reset;
12
13    // DefiniÃ§Ã£o dos estados
14    `define Inicio          4'b0000
15    `define Prologo1        4'b0001
16    `define Prologo2        4'b0010
17    `define Prologo3        4'b0011
18    `define Prologo4        4'b0100
19    `define Prologo5        4'b0101
20    `define Kernel1         4'b0110
21    `define Kernel2         4'b0111
```

```
22    `define Kernel3         4'b1000
23    `define Incremento_I    4'b1001
24    `define Epilogo1        4'b1010
25    `define Epilogo2        4'b1011
26    `define Epilogo3        4'b1100
27    `define Epilogo4        4'b1101
28    `define Epilogo5        4'b1110
29    `define Fim             4'b1111
30
31    reg [3:0] Estado;
32
33    //-------------------------------------------------------------------------
34    // Maquina de estados: LoopPipeliningSimples
35    //-------------------------------------------------------------------------
36    //---------------------------------------
37    always @ (posedge CLK or posedge reset)
38       begin : LoopPipelining
39          if (reset)   begin
40                i <= 0;
41                s <= 0;
42                n <= 7;
43                a <= 0;
44                Estado <= `Inicio;
45             end
46          else
47             if(enable) begin
48                   case (Estado)
49                      `Inicio:
50                         begin
51                            Estado <= `Prologo1;
52                         end
53                      `Prologo1:
54                         begin
55                            Estado <= `Prologo2;
56                         end
57                      `Prologo2:
58                         begin
59                            Estado <= `Prologo3;
60                         end
61                      `Prologo3:
62                         begin
63                            Estado <= `Prologo4;
64                         end
```

```
65                  `Prologo4:
66                     begin
67                        Estado <= `Prologo5;
68                     end
69                     `Prologo5:
70                     begin
71                        Estado <= `Kernel1;
72                     end
73                  `Kernel1:
74                     begin
75                       Estado <= `Kernel2;
76                     end
77                  `Kernel2:
78                     begin
79                        Estado <= `Kernel3;
80                     end
81                  `Kernel3:
82                     begin
83                        if (i>=n)
84                           Estado <= `Epilogo1;
85                        else if (i<n)
86                           Estado <= `Incremento_I;
87                     end
88                  `Incremento_I:
89                     begin
90                        Estado <= `Kernel1;
91                     end
92                  `Epilogo1:
93                     begin
94                        Estado <= `Epilogo2;
95                     end
96                  `Epilogo2:
97                     begin
98                        Estado <= `Epilogo3;
99                     end
100                 `Epilogo3:
101                    begin
102                       Estado <= `Epilogo4;
103                    end
104                 `Epilogo4:
105                    begin
106                       Estado <= `Epilogo5;
107                    end
```

```
108                     `Epilogo5:
109                         begin
110                             Estado <= `Fim;
111                         end
112                 endcase
113             end
114     end
115
116
117 always @(Estado)
118     begin
119       case (Estado)
120       `Prologo1: c <= a+10;
121
122         `Prologo2:
123         begin
124             a <= (c*2);
125             b <= (c*5);
126             s <= (s+c);
127         end
128
129       `Prologo3:
130         begin
131           c <= (a+10);
132         end
133
134         `Prologo4:
135           begin
136             k <= s;
137             a <= (c*2);
138             b <= (c*5);
139             s <= (s+c);
140           end
141
142         `Prologo5:
143           begin
144            c <= (a+10);
145           end
146
147         `Kernel1:
148          begin
149             s <= (s+k);
150          end
```

```
151
152             `Kernel2:
153              begin
154                  k <= s;
155                  a <= (c*2);
156                  b <= (c*5);
157                  s <= (s+c);
158               end
159
160             `Kernel3:
161              begin
162                 c <= (a+10);
163              end
164
165              `Incremento_I:
166               begin
167                 i <= i+1;
168               end
169
170                `Epilogo1:
171                begin
172                    s <= (s+k);
173                 end
174
175                `Epilogo2:
176                 begin
177                   k <= s;
178                    a <= (c*2);
179                    b <= (c*5);
180                    s <= (s+c);
181                  end
182
183                `Epilogo3:
184                   begin
185                       s = (s+k);
186                   end
187
188                `Epilogo4:
189                   begin
190                       k <= s;
191                   end
192
193                `Epilogo5:
```

```
194                     begin
195                         s <= (s+k);
196                     end
197
198             endcase
199         end
200
201
202
203 endmodule
```

Figura B.1: Resultado do Código em C da Listagem 2.5.

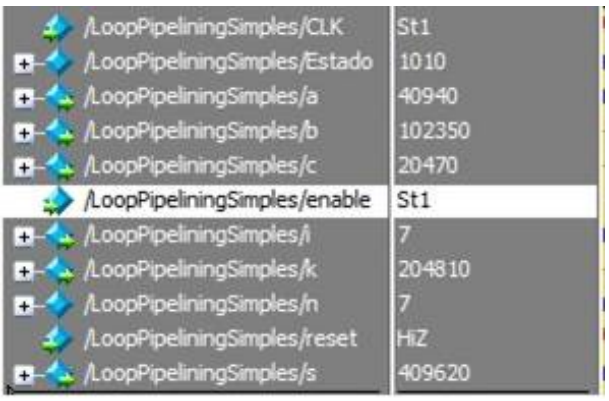

Figura B.2: Resultado do Código em Verilog da Listagem B.1.

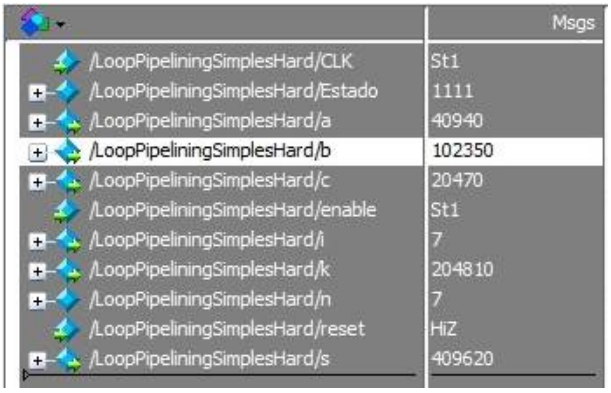

Figura B.3: Resultado do Código em Verilog da Listagem B.3.

Apendice C

DESENVOLVIMENTO INICIAL EM ACTIVE-HDL

Para implementar as máquinas de estados, primeiramente foram realizados testes com a ferramenta Active-HDL da Aldec (ALDEC, 2014) com máquinas de estados simples. O Active-HDL foi utilizado também para a síntese e simulação das mesmas. Essa ferramenta foi testada pela facilidade de implementar as máquinas de estados, em que estas podem ser descritas na forma de diagramas e a própria ferramenta gera o Verilog de forma automática. Na Listagem C.1 é mostrado um *loop* simples em C, que foi descrito em Verilog, utilizando máquinas de estados através da ferramenta Active-HDL.

Listagem C.1: *Loop* simples utilizado para teste da ferramenta Active-HDL.

```
1 ...
2
3 for (i=0; i<=n; i++)
4         {
5                 x=y+1;
6                 y=b+x;
7         }
8 ...
```

A Figura C.1 mostra a implementação do *loop* da Listagem C.1 através da interface gráfica do Active-HDL. As entradas da máquina de estados são: *enable*, *reset* e *CLK*. As saídas são: *y*, *b*, *x* e *i*. Foram criados cinco estados:

- Inicio: onde os valores são resetados.
- Fim: que determina o fim da execução da máquina de estados.
- Incremento_I : onde *i* é incrementado.

•Calculo_X: onde o valor de *x* é calculado.

•Calculo_Y: onde o valor de *y* é calculado.

Dentro de cada estado, são inseridos os comandos que serão executados e que são representados pelos retângulos brancos. A condição de transição de um estado para outro estado é mostrada pelos textos em vermelho. A cada ciclo de *clock*, a máquina de estados transita de um estado para outro. Quando o valor de *i* é maior que 10, a máquina termina sua execução e permanece no estado *Fim* até que a máquina seja resetada. A Listagem C.2 mostra o código em Verilog gerado automaticamente pela ferramenta a partir do diagrama da Figura C.1.

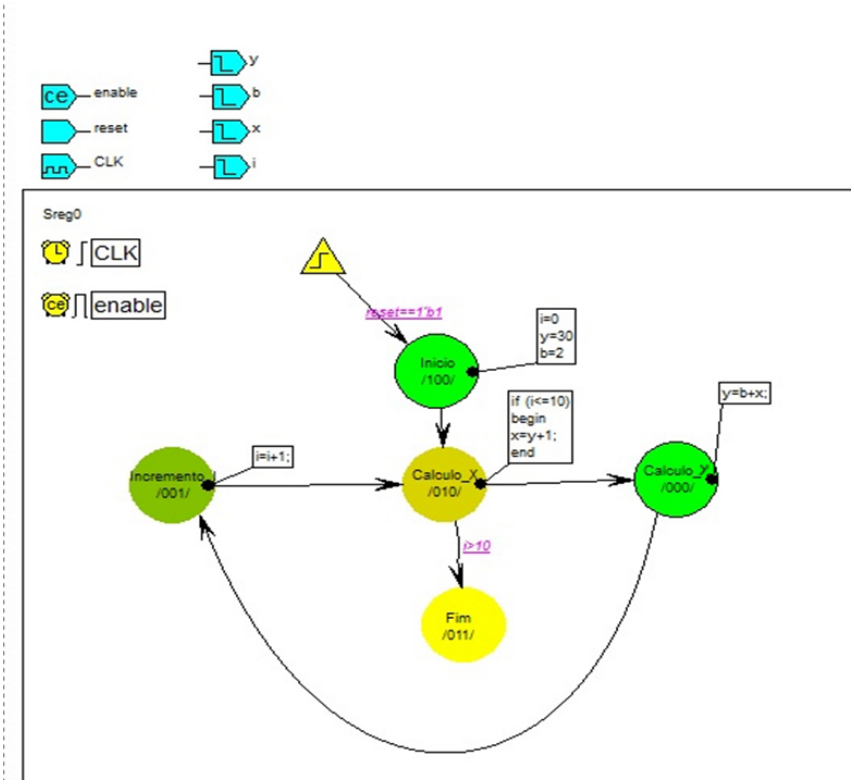

Figura C.1: Máquina de estados descrita através da ferramenta Active-HDL.

Listagem C.2: Código em Verilog gerado automaticamente pela ferramenta Active-HDL a partir do diagrama da Figura C.1.

```
1
2 module LoopSimples (CLK, b, enable, i,
3  reset, x, y);
4 input          CLK;
5 input          enable;
6 input          reset;
7 output         b;
8 output         i;
```

```
9  output          x;
10 output          y;
11
12 wire          CLK;
13 integer       b, next_b;
14 wire          enable;
15 integer       i, next_i;
16 wire          reset;
17 integer       x, next_x;
18 integer       y, next_y;
19 `define Calculo_Y 3'b000
20 `define Incremento_I 3'b010
21 `define Calculo_X 3'b011
22 `define Fim 3'b001
23 `define Inicio 3'b100
24
25 reg [2:0] CurrState_Sreg0;
26 reg [2:0] NextState_Sreg0;
27
28
29 always @ (b or i or x or y or CurrState_Sreg0)
30 begin : Sreg0_NextState
31         NextState_Sreg0 <= CurrState_Sreg0;
32         next_y = y;
33         next_i = i;
34         next_x = x;
35         next_b = b;
36         case (CurrState_Sreg0)
37
38 `Calculo_Y:
39         begin
40         next_y = b+x;
41           NextState_Sreg0 <= `Incremento_I;
42         end
43
44 `Incremento_I:
45         begin
46           next_i = i+1;
47           NextState_Sreg0 <= `Calculo_X;
48         end
49
50 `Calculo_X:
51         begin
```

```
52          if (i<=10)
53          begin
54            next_x = y+1;
55          end
56          if (i>10)
57            NextState_Sreg0 <= `Fim;
58          else
59            NextState_Sreg0 <= `Calculo_Y;
60          end
61
62  `Inicio:
63          begin
64          next_i = 0;
65          next_y = 30;
66          next_b = 2;
67          NextState_Sreg0 <= `Calculo_X;
68          end
69          endcase
70  end
71
72
73  always @ (posedge CLK)
74  begin : Sreg0_CurrentState
75          if (reset==1'b1)
76            CurrState_Sreg0 <= `Inicio;
77          else
78            if (enable)
79          CurrState_Sreg0 <= NextState_Sreg0;
80  end
81
82  always @ (posedge CLK)
83  begin : Sreg0_RegOutput
84          if (reset==1'b1)
85          begin
86                  i <= 0;
87                  y <= 30;
88                  b <= 2;
89          end
90          else if (enable)
91          begin
92                  i <= next_i;
93                  y <= next_y;
94                  x <= next_x;
```

```
95                      b <= next_b;
96              end
97      end
98
99      endmodule
```

O estilo de codificação da Listagem C.2 é parecido com o exemplo mostrado na Listagem A.12 da Subseção A.5. A Ferramenta decidiu dar os seguintes valores para os estados: 0 para o $Calculo_Y$, 2 para o $Incremento_I$, 3 para o $Calculo_X$, 1 para o Fim e 4 para o $Inicio$. Esses valores foram dados seguindo a ordem de criação dos estados. A ferramenta cria duas variáveis para cada uma das saídas descritas no diagrama, e cria também duas variáveis para armazenar os valores dos estados. A ideia é ter um valor atual e um valor a ser utilizado na próxima mudança do *clock*, que será o próximo valor da saída. A ferramenta criou três blocos *always*, em que o primeiro é um bloco de circuito combinacional sensível a mudanças nos valores das saídas e que realiza a transição dos estados e das saídas. Nesse bloco, os valores atualizados são para o próximo pulso de *clock*. O segundo bloco *always* é um bloco sequencial que atribui os novos valores dos estados que foram calculados no primeiro *always*. A mudança do antigo valor do estado para o novo valor ocorre na borda de subida do *clock*. O terceiro bloco *always* é parecido com o segundo, mas atualiza apenas os valores das saídas.

A Figura C.2 mostra a simulação funcional em formato de ondas da máquina de estados utilizando a ferramenta Active-HDL.

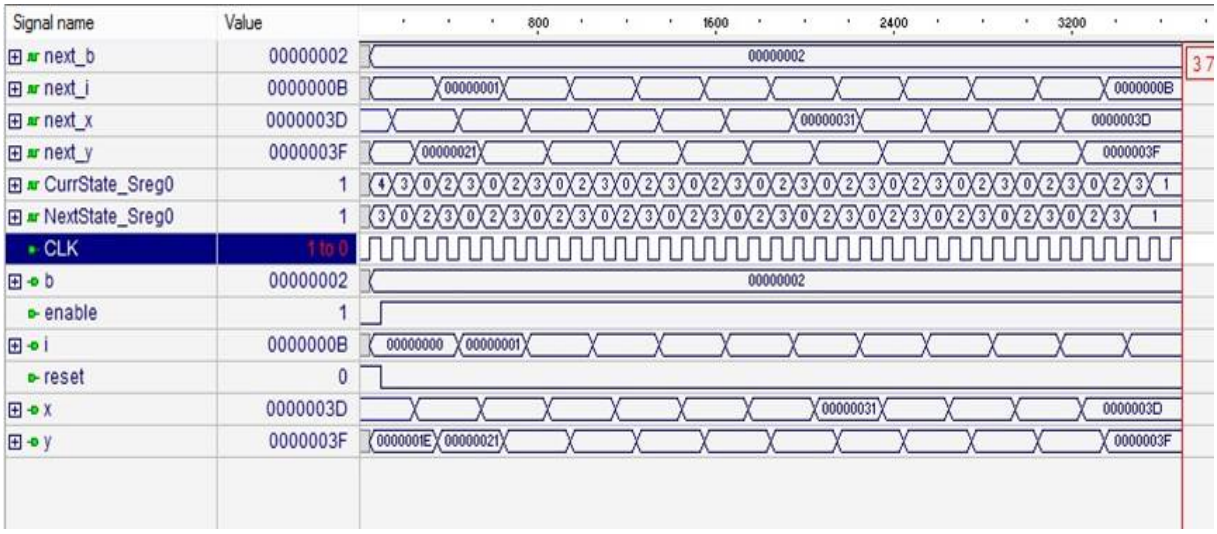

Figura C.2: Simulação em formato de ondas da Figura C.1 através da ferramenta Active-HDL.

A ferramenta Active-HDL é bem completa e fácil de ser utilizada, e também disponibiliza várias funcionalidades interessantes, como a opção de visualizar as mudanças desses estados no próprio diagrama criado através da mudança de cores dos estados. Na Figura C.1, o estado

atual é representado pelo círculo em que a cor é amarelo claro, que no caso, é o estado *Fim*. O estado anterior $Calculo_X$ possui um amarelo mais escuro, e o estado $Incremento_I$ de dois ciclos anteriores, é representado pelo amarelo esverdeado. Apesar de ser uma ótima ferramenta o Active-HDL não foi escolhido para ser utilizada no projeto, as ferramentas escolhidas foram o Quartus II e o ModelSim-Altera.

Por meio das ferramentas escolhidas, o código da Listagem C.2 foi refeito utilizando um estilo de criação de máquinas de estados parecido com o mostrado na Subseção A.5, na Listagem A.13. A Listagem C.3 mostra o novo código. Apesar de não ter sido a ferramenta escolhida, o Active-HDL foi importante para ter um melhor entendimento sobre máquinas de estados e sobre como descrevê-las de forma mais adequada.

Listagem C.3: Código da Listagem C.2 remodelado.

```
1  module LoopSimplesS (CLK, b, enable, i, reset, x, y);
2  input             CLK;
3  input             enable;
4  input             reset;
5  output            b;
6  output            i;
7  output            x;
8  output            y;
9
10 wire          CLK;
11 integer       b;
12 wire          enable;
13 integer       i;
14 wire          reset;
15 integer       x;
16 integer       y;
17
18 // Definição dos estados
19 `define Calculo_X 3'b011
20 `define Calculo_Y 3'b000
21 `define Fim 3'b001
22 `define Incremento_I 3'b010
23 `define Inicio 3'b100
24
25 reg [2:0] Estado;
26   always @ (posedge CLK or posedge reset)
27       begin
28          if (reset)
29          begin
```

```
30              i <= 0;
31              y <= 30;
32              x <= 0;
33              b <= 2;
34              Estado <= `Inicio;
35          end
36          else
37              if( enable) begin
38                  case (Estado)
39                      `Inicio:
40                      begin
41                          i <= 0;
42                          y <= 30;
43                          b <= 2;
44                          x <= 0;
45                          Estado <= `Calculo_X;
46                      end
47
48                      `Calculo_X:
49                      begin
50                          if (i>10)
51                              Estado <=`Fim;
52                          else
53                          begin
54                              x <= y+1;
55                              Estado <= `Calculo_Y;
56                          end
57                       end
58
59                     `Calculo_Y:
60                     begin
61                         y <= b+x;
62                         Estado <= `Incremento_I;
63                     end
64
65                     `Incremento_I:
66                     begin
67                         i <= i+1;
68                         Estado <= `Calculo_X;
69                     end
70
71
72              endcase
```

```
73              end
74          end
75
76  endmodule
```

Apendice D

OUTRAS IMPLEMENTAÇÕES DE FILAS

Na Listagem D.1 é mostrada a implementação de uma fila do livro (CHU, 2008) e na Listagem D.2 a terceira versão do QRF. Na Listagem D.3 é mostrado um exemplo de como foram feitos os testes com a fila da Altera.

Listagem D.1: Fila utilizada para criar o QRF. Fonte:Chu (2008).

```
1  module FilaLivro #(parameter B=8,W=4)// parametros declarados junto ao nome
       do módulo
2  /*Declaração dos pinos de entrada e saída*/
3  (
4  input wire clk, reset,
5  /*bits de entrada de leitura e escrita na fila*/
6  input wire rd, wr,
7
8  /*sinal de entrada que recebe os dados da cabeça da fila*/
9  input wire [B-1:0] w_data,
10
11 /*bits de saída que sinalizam se a fila está cheia ou vazia*/
12 output wire empty, full,
13
14 /*sinal saída que recebe os dados da cauda da fila*/
15 output wire [B-1:0] r_data
16 );
17
18 /*memória utilizada pela fila*/
19 reg [B-1:0] array_reg [(2**W)-1:0] ;
20
21 /*endereços da cauda da fila:  atual, próximo e anterior */
22 reg [W-1:0] w_ptr_reg, w_ptr_next , w_ptr_succ;
```

```
23
24 /*endereços da cabeça da fila:  atual, próximo e anterior */
25 reg [W-1:0] r_ptr_reg , r_ptr_next , r_ptr_succ ;
26
27 /*registradores auxiliares para a controle da fila vazia e da fila cheia
       */
28 reg full_reg, empty_reg, full_next, empty_next;
29
30 /*bit de controle para escrita na memória*/
31 wire wr_en;
32
33 /*always que controla a escrita na memória*/
34 always @(posedge clk)
35 if (wr_en)
36    array_reg[w_ptr_reg] <= w_data;
37
38  /*saída da memória*/
39 assign r_data = array_reg[r_ptr_reg] ;
40
41 /*escreve se o bit de escrita estiver ativo e a memória não estiver cheia*/
42 assign wr_en = wr & ~full_reg;
43
44 /*lógica sequencial, onde os valores dos registradores são atualizados*/
45 always @ ( posedge clk , posedge reset)
46   if (reset)
47     begin
48       w_ptr_reg <= 0;
49       r_ptr_reg <= 0;
50       full_reg <= 1'b0;
51       empty_reg <= 1'b1;
52     end
53   else
54     begin
55       w_ptr_reg <= w_ptr_next ;
56       r_ptr_reg <= r_ptr_next;
57       full_reg <= full_next;
58       empty_reg <= empty_next ;
59     end
60
61 /*lógica combinacional que é ativa caso qualquer bit de entrada mude de
       valor */
62  always @*
63     begin
```

```
64        w_ptr_succ = w_ptr_reg + 1;
65        r_ptr_succ = r_ptr_reg + 1;
66        w_ptr_next = w_ptr_reg;
67        r_ptr_next = r_ptr_reg;
68        full_next = full_reg;
69        empty_next = empty_reg;
70      case ({wr, rd})
71
72        2'b01:
73          if (~empty_reg)
74        begin
75          r_ptr_next = r_ptr_succ ;
76          full_next = 1'b0;
77        if(r_ptr_succ == w_ptr_reg)
78           empty_next = 1'b1;
79        end
80        2'b10:
81          if(~full_reg)
82            begin
83              w_ptr_next = w_ptr_succ ;
84              empty_next = 1'b0;
85          if(w_ptr_succ == r_ptr_reg)
86             full_next = 1'b1;
87          end
88        2'b11:
89          begin
90            w_ptr_next = w_ptr_succ;
91            r_ptr_next = r_ptr_succ ;
92          end
93  endcase
94  end
95
96
97  assign full = full_reg;
98  assign empty = empty_reg;
99
100 endmodule
```

Listagem D.2: Terceira versão do QRF.

```
1  module QRFVersao3(head_data, tail_data, clk,reset, dequeue, enqueue);
2  parameter DATA_WIDTH = 8;
3  parameter REGFILE_WIDTH = 11;
4
```

```
5  //Declaração dos Pinos
6  input wire enqueue ;
7  input wire dequeue ;
8  input wire clk,reset;
9  input wire [DATA_WIDTH-1:0] tail_data;
10 output wire [DATA_WIDTH-1:0] head_data;
11 integer tamanho = 2**REGFILE_WIDTH;
12 reg [REGFILE_WIDTH-1:0] head_address = {REGFILE_WIDTH{1'b0}};
13 reg [REGFILE_WIDTH-1:0] tail_address = {REGFILE_WIDTH{1'b0}};
14 reg [REGFILE_WIDTH-1:0] tail_address_succ = {REGFILE_WIDTH{1'b0}};
15 reg [REGFILE_WIDTH-1:0] plus = 1'b1;
16 reg [REGFILE_WIDTH-1:0] ra1;
17
18 //Declaração da memória
19 reg  [DATA_WIDTH - 1:0] Mem[0:(2**REGFILE_WIDTH)- 1];
20 reg full = 0 ;
21 reg empty = 1 ;
22 integer i;
23 wire wr_en;
24
25 assign wr_en = enqueue & (~full | dequeue);
26
27 //Lógica
28 always@(posedge  clk or posedge reset) begin
29     if (reset)
30       begin
31         head_address =  {REGFILE_WIDTH{1'b0}};
32         tail_address <=  {REGFILE_WIDTH{1'b0}};
33         tail_address_succ = {REGFILE_WIDTH{1'b0}};
34         ra1 = {REGFILE_WIDTH{1'b0}};
35                 end
36     else
37     case ({enqueue,dequeue})
38         2'b01:
39           if(~empty)
40               begin
41                  head_address = head_address + plus;
42                 full <= 1'b0;
43               if ( tail_address == head_address)
44                 empty <= 1'b1;
45           end
46
47         2'b10:
```

```
48              if(~full)
49                 begin
50                   tail_address <= tail_address + plus;
51                   tail_address_succ = tail_address + plus;
52                   empty <= 1'b0;
53                 if ( tail_address_succ == head_address)
54                   full <= 1'b1;
55              end
56
57           2'b11:
58           begin
59              if(~empty)
60                begin
61                  head_address = head_address + plus;
62                end
63              tail_address <= tail_address + plus;
64              tail_address_succ = tail_address + plus;
65              empty <= 1'b0;
66         end
67
68       endcase
69
70       ra1 = head_address;
71   end
72
73   always @(negedge clk)
74   if (wr_en)
75      Mem[tail_address] <= tail_data;
76
77    assign head_data = Mem[ra1] ;
78   endmodule
```

Listagem D.3: Teste com Fila da Altera.

```
1
2
3
4   module TesteFilaAltera(clk,enable,reset, readData1_out,readData1_out);
5
6
7     parameter Estado1 = 0;
8     parameter Estado2 = 1;
9     parameter Estado3 = 2;
10    parameter Estado4 = 3;
```

```
11    parameter Estado5 = 4;
12    parameter Estado6 = 5;
13    parameter Estado7 = 6;
14    parameter Estado8 = 7;
15    parameter Estado9 = 8;
16    parameter Estado10 = 9;
17    parameter Estado11 = 10;
18    parameter Estado12 = 11;
19     parameter Estado13 = 12;
20    parameter Estado14 = 13;
21    parameter Estado15 = 14;
22    parameter Inicio = 15;
23    input clk, enable, reset;
24    output readData1_out,readData1_out;
25
26
27
28    wire clk,enable, reset;
29    reg[5:0] estado;
30    reg writeEn ;
31    reg [1:0] readAddress1, writeAddress;
32    reg [15:0] writeData;
33    wire [15:0] readData1;
34
35    wire [1:0] readAddress1_out = readAddress1;
36    wire [1:0] writeAddress_out = writeAddress;
37    wire  writeEn_out = writeEn;
38    wire [15:0] readData1_out = readData1;
39
40          reg     [7:0]  data;
41          reg        rdreq;
42          reg        wrreq;
43
44          wire [7:0] sub_wire0;
45          wire  sub_wire1;
46          wire  sub_wire2;
47          wire [7:0] sub_wire3;
48          wire [7:0] usedw = sub_wire0[7:0];
49          wire  empty = sub_wire1;
50          wire  full = sub_wire2;
51          wire [7:0] q = sub_wire3[7:0];
52
53          scfifo  scfifo_component (
```

```
54                                  .clock (clk),
55                                  .data (data),
56                                  .rdreq (rdreq),
57                                  .wrreq (wrreq),
58                                  .usedw (sub_wire0),
59                                  .empty (sub_wire1),
60                                  .full (sub_wire2),
61                                  .q (sub_wire3),
62                                  .aclr (),
63                                  .almost_empty (),
64                                  .almost_full (),
65                                  .sclr ());
66          defparam
67                  scfifo_component.add_ram_output_register = "OFF",
68                  scfifo_component.intended_device_family = "Cyclone II",
69                  scfifo_component.lpm_numwords = 256,
70                  scfifo_component.lpm_showahead = "ON",
71                  scfifo_component.lpm_type = "scfifo",
72                  scfifo_component.lpm_width = 8,
73                  scfifo_component.lpm_widthu = 8,
74                  scfifo_component.overflow_checking = "ON",
75                  scfifo_component.underflow_checking = "ON",
76                  scfifo_component.use_eab = "ON";
77
78          always @(posedge clk or posedge reset) begin
79            if (reset) begin
80
81              estado = Inicio;
82            end else begin
83              case (estado)
84                Inicio:
85                begin
86                  wrreq = 1 ;
87                  rdreq = 0 ;
88                  data = 10 ;
89                  estado = Estado1;
90                end
91
92                Estado1:
93                begin
94                 rdreq = 0 ;
95                 wrreq = 0 ;
96                 estado = Estado4;
```

```
97                end
98
99
100               Estado4:
101               begin
102                wrreq = 1 ;
103                data = q + 10;
104                readAddress1 = 1;
105                estado = Estado5;
106               end
107
108               Estado5:
109               begin
110                 rdreq = 1;
111                wrreq  = 0;
112                 estado = Estado6;
113               end
114
115               Estado6:
116               begin
117                 rdreq = 0 ;
118                 rdreq = 0 ;
119                estado = Estado8;
120               end
121
122
123               Estado8:
124               begin
125               rdreq = 0 ;
126                wrreq = 1 ;
127                data = q + 10;
128                estado = Estado9;
129               end
130
131                Estado9:
132               begin
133                 rdreq = 1 ;
134                 wrreq  = 0;
135                 estado = Estado10;
136               end
137
138
139               Estado10:
```

```
140                 begin
141                   rdreq = 0 ;
142                    rdreq = 0 ;
143                   estado = Estado12;
144                 end
145
146
147                  Estado12:
148                 begin
149                  wrreq = 1 ;
150                  data = q + 10;
151                   estado = Estado13;
152                 end
153
154
155                    Estado13:
156                 begin
157                   rdreq = 1 ;
158                    wrreq = 0 ;
159                   estado = Estado14;
160                 end
161
162
163
164               endcase
165             end
166
167         end
168
169         endmodule
```

Apendice E

CÓDIGOS UTILIZADOS NOS TESTES COM MEMÓRIA

Listagem E.1: *Loop* em C que será paralelizado.

```
1  for (i=0; i<=n; i++)
2  {
3          a[i+1] = a[i] + 1;
4          b[i] = a[i+1] + 2;
5          c[i] = b[i] + 3;
6          d[i] = c[i];
7
8  }
```

Listagem E.2: *Loop* sem *loop pipelining* em Verilog com uma memória de uma porta.

```
1  module LoopMemoria1porta(clock, enable, program_counterA, program_counterB,
       program_counterC, program_counterD, register_A, memory_data_register_out,
       reset);
2    parameter Inicio = 1;
3    parameter Calculo_A = 2;
4    parameter Calculo_A2 = 3;
5    parameter Calculo_A3 = 4;
6    parameter Calculo_A4 = 5;
7    parameter Calculo_B = 6;
8    parameter Calculo_B2 = 7;
9    parameter Calculo_B3 = 8;
10   parameter Calculo_B4 = 9;
11   parameter Calculo_C = 10;
12   parameter Calculo_C2 = 11;
13   parameter Calculo_C3 = 12;
```

```
14    parameter Calculo_C4 = 13;
15    parameter Calculo_D = 14;
16    parameter Calculo_D2 = 15;
17    parameter Calculo_D3 = 16;
18    parameter Calculo_D4 = 17;
19    parameter Incremento_I = 18;
20    parameter Fim = 19;
21
22
23    input clock,enable,reset;
24    output [7:0]program_counterA,program_counterB,program_counterC,
          program_counterD;
25    output [15:0]register_A,memory_data_register_out;
26
27    wire reset;
28    reg [15:0]register_A;
29    reg [4:0]Estado,Estado2;
30    integer    i,contB,contC;
31    integer    a[10:0];
32    integer    b[9:0];
33    integer    c[9:0];
34    integer    d[9:0];
35    reg [7:0]memory_address_register;
36    reg memory_write;
37    reg [7:0]program_counterA,program_counterB,program_counterC,
          program_counterD;
38    wire [15:0]memory_data_register;
39    wire [15:0]memory_data_register_out = memory_data_register;
40
41    wire [15:0]memory_address_register_out = memory_address_register;
42
43    wire memory_write_out = memory_write;
44
45    reg enable2;
46    integer count;
47    altsyncram altsyncram_component (
48      .wren_a(memory_write_out),
49      .clock0(clock),
50      .address_a(memory_address_register_out),
51      .data_a(register_A),
52      .q_a(memory_data_register)
53      );
54
```

```
55    defparam
56      altsyncram_component.operation_mode="SINGLE_PORT",
57      altsyncram_component.width_a = 16,
58      altsyncram_component.widthad_a = 8,
59      altsyncram_component.outdata_reg_a="UNREGISTERED",
60      altsyncram_component.lpm_type = "altsyncram",
61      altsyncram_component.intended_device_family="Cyclone";
62
63
64
65
66
67
68
69    always @(posedge clock)
70      begin
71        if (reset) begin
72            contB=2;
73            contC=3;
74            Estado = Inicio;
75            memory_write = 1'b0;
76            program_counterA = 8'b00000000;
77            program_counterB = program_counterA + 11;
78            program_counterC = program_counterB + 10;
79            program_counterD = program_counterC + 10;
80            count = 0;
81            i = 0;
82          end
83        else
84          if(enable)
85            begin
86              case(Estado)
87
88                Inicio:
89                  begin
90                    count = count + 1;
91                    Estado = Calculo_A;
92                  end
93
94                Calculo_A:
95                  begin
96                    count = count + 1;
97                    if (i>=10)  begin
```

```
98                  Estado = Fim;
99                  enable2 = 1'b1;
100               end
101             else
102               begin
103                 memory_write = 1'b0;
104                 memory_address_register = program_counterA;
105                 Estado = Calculo_A2;
106               end
107           end
108
109         Calculo_A2:
110           begin
111             count = count + 1;
112             Estado = Calculo_A3;
113           end
114
115         Calculo_A3:
116           begin
117             count = count + 1;
118             memory_write = 1'b1;
119             memory_address_register = program_counterA + 1;
120             register_A = memory_data_register + 1;
121             Estado = Calculo_A4;
122           end
123
124         Calculo_A4:
125           begin
126             count = count + 1;
127             Estado = Calculo_B;
128           end
129
130         Calculo_B:
131           begin
132             count = count + 1;
133             memory_write = 1'b0;
134             memory_address_register = program_counterA + 1;
135             Estado = Calculo_B2;
136           end
137
138         Calculo_B2:
139           begin
140             count = count + 1;
```

```
141                 Estado = Calculo_B3;
142               end
143
144
145             Calculo_B3:
146               begin
147                 count = count + 1;
148                 memory_write = 1'b1;
149                 register_A = memory_data_register + 2;
150                 memory_address_register = program_counterB;
151                 Estado = Calculo_B4;
152               end
153
154             Calculo_B4:
155               begin
156                 count = count + 1;
157                 Estado = Calculo_C;
158               end
159
160             Calculo_C:
161               begin
162                 count = count + 1;
163                 memory_write = 1'b0;
164                 memory_address_register = program_counterB;
165                 Estado = Calculo_C2;
166
167               end
168
169             Calculo_C2:
170               begin
171                 count = count + 1;
172                 Estado = Calculo_C3;
173               end
174
175             Calculo_C3:
176               begin
177                 count = count + 1;
178                 memory_write = 1'b1;
179                 memory_address_register = program_counterC;
180                 register_A = memory_data_register + 3;
181                 Estado = Calculo_C4;
182               end
183
```

```
184
185             Calculo_C4:
186               begin
187                 count = count + 1;
188                 Estado = Calculo_D;
189               end
190
191
192             Calculo_D:
193               begin
194                 count = count + 1;
195                 memory_write = 1'b0;
196                 memory_address_register = program_counterC;
197                 Estado = Calculo_D2;
198               end
199
200             Calculo_D2:
201               begin
202                 count = count + 1;
203                 Estado = Calculo_D3;
204               end
205
206             Calculo_D3:
207               begin
208                 count = count + 1;
209                 memory_write = 1'b1;
210                 memory_address_register = program_counterD;
211                 register_A = memory_data_register;
212                 Estado = Calculo_D4;
213               end
214
215
216             Calculo_D4:
217               begin
218                 count = count + 1;
219                 Estado = Incremento_I;
220               end
221
222             Incremento_I:
223               begin
224                 count = count + 1;
225                 i = i+1;
226                 program_counterA = program_counterA + 1;
```

```
227                     program_counterB = program_counterB + 1;
228                     program_counterC = program_counterC + 1;
229                     program_counterD = program_counterD + 1;
230                     Estado = Calculo_A;
231                 end
232             endcase
233         end
234     end
235
236
237 endmodule
```

Listagem E.3: *Loop pipelining* em Verilog com uma memória de duas portas.

```
1  module LoopPipeMemo2portTrue(clock,enable,program_counterA,register_A,
       register_B,memory_data_register_outA,memory_data_register_outB,reset,
       program_counterB);
2
3    parameter Inicio        = 1;
4    parameter Prologo1_1    = 2;
5    parameter Prologo1_2    = 3;
6    parameter Prologo1_3    = 4;
7    parameter Prologo1_4    = 5;
8    parameter Prologo2_1    = 6;
9    parameter Prologo2_2    = 7;
10   parameter Prologo2_3    = 8;
11   parameter Prologo2_4    = 9;
12   parameter Prologo3_1   = 10;
13   parameter Prologo3_2   = 11;
14   parameter Prologo3_3   = 12;
15   parameter Prologo3_4   = 13;
16   parameter Prologo3_5   = 14;
17   parameter Prologo3_6   = 15;
18   parameter Prologo3_7   = 16;
19   parameter Prologo3_8   = 17;
20   parameter Kernel_1     = 18;
21   parameter Kernel_2     = 19;
22   parameter Kernel_3     = 20;
23   parameter Kernel_4     = 21;
24   parameter Kernel_5     = 22;
25   parameter Kernel_6     = 23;
26   parameter Kernel_7     = 24;
27   parameter Kernel_8     = 25;
28   parameter Epilogo1_1    = 26;
```

```
29    parameter Epilogo1_2   = 27;
30    parameter Epilogo1_3   = 28;
31    parameter Epilogo1_4   = 29;
32    parameter Epilogo1_5   = 30;
33    parameter Epilogo1_6   = 31;
34    parameter Epilogo1_7   = 32;
35    parameter Epilogo1_8   = 33;
36    parameter Epilogo2_1  = 34;
37    parameter Epilogo2_2  = 35;
38    parameter Epilogo2_3  = 36;
39    parameter Epilogo2_4    = 37;
40    parameter Epilogo3_1    = 38;
41    parameter Epilogo3_2    = 39;
42    parameter Epilogo3_3    = 40;
43    parameter Epilogo3_4    = 41;
44    parameter Fim          = 42;
45
46    input clock,enable,reset;
47    output [7:0]program_counterA,program_counterB;
48    output [15:0]register_A,register_B,memory_data_register_outA,
          memory_data_register_outB;
49    reg enable2;
50    reg [15:0]register_A,register_B;
51
52    reg [7:0]memory_address_registerA,memory_address_registerB;
53    reg memory_writeA,memory_writeB;
54    reg [7:0]program_counterA,program_counterB;
55
56    wire [15:0]memory_data_registerA,memory_data_registerB;
57    wire [15:0]memory_data_register_outA = memory_data_registerA;
58    wire [15:0]memory_data_register_outB = memory_data_registerB;
59    wire [15:0]memory_address_register_outA = memory_address_registerA;
60    wire [15:0]memory_address_register_outB = memory_address_registerB;
61    wire memory_write_outA = memory_writeA;
62    wire memory_write_outB = memory_writeB;
63
64    altsyncram  altsyncram_component (
65      .clock0 (clock),
66      .wren_a (memory_write_outA),
67      .address_a (memory_address_register_outA),
68      .data_a (register_A),
69      .wren_b (memory_write_outB),
70      .address_b (memory_address_register_outB),
```

```
71      .data_b (register_B),
72      .q_a (memory_data_registerA),
73      .q_b (memory_data_registerB),
74      .aclr0 (1'b0),
75      .aclr1 (1'b0),
76      .addressstall_a (1'b0),
77      .addressstall_b (1'b0),
78      .byteena_a (1'b1),
79      .byteena_b (1'b1),
80      .clock1 (1'b1),
81      .clocken0 (1'b1),
82      .clocken1 (1'b1),
83      .clocken2 (1'b1),
84      .clocken3 (1'b1),
85      .eccstatus (),
86      .rden_a (1'b1),
87      .rden_b (1'b1));
88    defparam
89      altsyncram_component.address_reg_b = "CLOCK0",
90      altsyncram_component.clock_enable_input_a = "BYPASS",
91      altsyncram_component.clock_enable_input_b = "BYPASS",
92      altsyncram_component.clock_enable_output_a = "BYPASS",
93      altsyncram_component.clock_enable_output_b = "BYPASS",
94      altsyncram_component.indata_reg_b = "CLOCK0",
95      altsyncram_component.intended_device_family = "Cyclone II",
96      altsyncram_component.lpm_type = "altsyncram",
97      altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
98      altsyncram_component.outdata_aclr_a = "NONE",
99      altsyncram_component.outdata_aclr_b = "NONE",
100     altsyncram_component.outdata_reg_a = "UNREGISTERED",
101     altsyncram_component.outdata_reg_b = "UNREGISTERED",
102     altsyncram_component.power_up_uninitialized = "FALSE",
103     altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
104     altsyncram_component.widthad_a = 8,
105     altsyncram_component.widthad_b = 8,
106     altsyncram_component.width_a = 16,
107     altsyncram_component.width_b = 16,
108     altsyncram_component.width_byteena_a = 1,
109     altsyncram_component.width_byteena_b = 1,
110     altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0";
111
112
113   integer       inia,inib,inic,inid,pca,pcb,pcc,pcd,i,n;
```

```
114   integer      a[11:0];
115   integer      b[10:0];
116   integer      c[10:0];
117   integer      d[10:0];
118
119
120
121
122   reg [6:0] Estado;
123   reg [6:0] Estado2;
124
125   always @ (posedge clock or posedge reset)
126     begin : LoopPipelining
127       if (reset)  begin
128           memory_writeA = 1'b0;
129           memory_writeB = 1'b0;
130           inia = 0;
131           inib = inia + 12;
132           inic = inib + 11;
133           inid = inic + 11;
134           pca = 0;
135           pcb = 0;
136           pcc = 0;
137           pcd = 0;
138           i = 0;
139           n = 7;
140
141           Estado = Inicio;
142         end
143       else
144         if(enable) begin
145             case (Estado)
146               Inicio:
147                 begin
148                   Estado = Prologo1_1;
149                 end
150
151               Prologo1_1:
152                 begin
153                   memory_writeA = 1'b0;
154                   memory_address_registerA = inia + pca;
155                   Estado = Prologo1_2;
156                 end
```

```
157
158             Prologo1_2:
159               begin
160                 Estado = Prologo1_3;
161               end
162
163             Prologo1_3:
164               begin
165                 memory_writeA = 1'b1;
166                 memory_address_registerA = inia + pca + 1;
167                 register_A = memory_data_registerA + 1;
168                 pca = pca + 1;
169                 Estado = Prologo1_4;
170               end
171
172             Prologo1_4:
173               begin
174                 Estado = Prologo2_1;
175               end
176
177
178             Prologo2_1:
179               begin
180                 memory_writeA = 1'b0;
181                 memory_writeB = 1'b0;
182                 memory_address_registerA = inia + pca;
183                 memory_address_registerB = inia + pcb + 1;
184                 Estado = Prologo2_2;
185               end
186
187             Prologo2_2:
188               begin
189                 Estado = Prologo2_3;
190               end
191
192
193             Prologo2_3:
194               begin
195                 memory_writeB = 1'b1;
196                 memory_writeA = 1'b1;
197                 memory_address_registerA = inia + pca + 1;
198                 memory_address_registerB = inib + pcb;
199                 register_A = memory_data_registerA + 1;
```

```
200                 register_B = memory_data_registerB + 2;
201                 pca= pca + 1;
202                 pcb= pcb + 1;
203                 Estado = Prologo2_4;
204               end
205
206             Prologo2_4:
207               begin
208                 Estado = Prologo3_1;
209               end
210
211
212             Prologo3_1:
213               begin
214                 memory_writeA = 1'b0;
215                 memory_writeB = 1'b0;
216                 memory_address_registerA = inia + pca;
217                 memory_address_registerB = inia + pcb + 1;
218                 Estado = Prologo3_2;
219               end
220
221             Prologo3_2:
222               begin
223                 Estado = Prologo3_3;
224               end
225
226
227             Prologo3_3:
228               begin
229                 memory_writeA = 1'b1;
230                 memory_writeB = 1'b1;
231                 memory_address_registerA = inia + pca + 1;
232                 memory_address_registerB = inib + pcb;
233                 register_A = memory_data_registerA + 1;
234                 register_B = memory_data_registerB + 2;
235                 pca= pca + 1;
236                 pcb= pcb + 1;
237                 Estado = Prologo3_4;
238               end
239
240             Prologo3_4:
241               begin
242                 Estado = Prologo3_5;
```

```
243                 end
244
245
246               Prologo3_5:
247                 begin
248                   memory_writeA = 1'b0;
249                   memory_writeB = 1'b0;
250                   memory_address_registerA = inib + pcc;
251                   Estado = Prologo3_6;
252                 end
253
254               Prologo3_6:
255                 begin
256                   Estado = Prologo3_7;
257                 end
258
259
260               Prologo3_7:
261                 begin
262                   memory_writeA = 1'b1;
263                   memory_address_registerA = inic + pcc;
264                   register_A = memory_data_registerA + 3;
265                   pcc = pcc + 1;
266                   Estado = Prologo3_8;
267                 end
268
269               Prologo3_8:
270                 begin
271                   Estado = Kernel_1;
272                 end
273
274
275               Kernel_1:
276                 begin
277                   if (i>=n)
278                     Estado = Epilogo1_1;
279                   else
280                     if (i<n)
281                       begin
282                         memory_writeA = 1'b0;
283                         memory_writeB = 1'b0;
284                         memory_address_registerA = inia + pca;
285                         memory_address_registerB = inia + pcb + 1;
```

```
286                     Estado = Kernel_2;
287                   end
288               end
289
290             Kernel_2:
291               begin
292                 Estado = Kernel_3;
293               end
294
295
296             Kernel_3:
297               begin
298                 memory_writeA = 1'b1;
299                 memory_writeB = 1'b1;
300                 memory_address_registerA = inia + pca + 1;
301                 memory_address_registerB = inib + pcb;
302                 register_A = memory_data_registerA + 1;
303                 register_B = memory_data_registerB + 2;
304                 pca= pca + 1;
305                 pcb= pcb + 1;
306                 Estado = Kernel_4;
307               end
308
309             Kernel_4:
310               begin
311                 Estado = Kernel_5;
312               end
313
314
315             Kernel_5:
316               begin
317                 memory_writeA = 1'b0;
318                 memory_writeB = 1'b0;
319                 memory_address_registerA = inib + pcc;
320                 memory_address_registerB = inic + pcd;
321                 Estado = Kernel_6;
322               end
323
324             Kernel_6:
325               begin
326                 Estado = Kernel_7;
327               end
328
```

```
329
330             Kernel_7:
331               begin
332                 memory_writeA = 1'b1;
333                 memory_writeB = 1'b1;
334                 memory_address_registerA = inic + pcc;
335                 memory_address_registerB = inid + pcd;
336                 register_A = memory_data_registerA + 3;
337                 register_B = memory_data_registerB;
338                 pcc = pcc + 1;
339                 pcd = pcd + 1;
340                 Estado = Kernel_8;
341               end
342
343             Kernel_8:
344               begin
345                 Estado = Kernel_1;
346                 i = i + 1;
347               end
348
349             Epilogo1_1:
350               begin
351                 memory_writeA = 1'b0;
352                 memory_writeB = 1'b0;
353                 memory_address_registerA = inia + pcb + 1;
354                 memory_address_registerB = inib + pcc;
355                 Estado = Epilogo1_2;
356               end
357
358             Epilogo1_2:
359               begin
360                 Estado = Epilogo1_3;
361               end
362
363
364             Epilogo1_3:
365               begin
366                 memory_writeA = 1'b1;
367                 memory_writeB = 1'b1;
368                 memory_address_registerA = inib + pcb;
369                 memory_address_registerB = inic + pcc;
370                 register_A = memory_data_registerA + 2;
371                 register_B = memory_data_registerB + 3;
```

```
372                 pcb= pcb + 1;
373                 pcc= pcc + 1;
374                 Estado = Epilogo1_4;
375               end
376
377             Epilogo1_4:
378               begin
379                 Estado = Epilogo1_5;
380               end
381
382
383             Epilogo1_5:
384               begin
385                 memory_writeA = 1'b0;
386                 memory_writeB = 1'b0;
387                 memory_address_registerA = inic + pcd;
388                 Estado = Epilogo1_6;
389               end
390
391             Epilogo1_6:
392               begin
393                 Estado = Epilogo1_7;
394               end
395
396
397             Epilogo1_7:
398               begin
399                 memory_writeA = 1'b1;
400                 memory_address_registerA = inid + pcd;
401                 register_A = memory_data_registerA;
402                 pcd = pcd + 1;
403                 Estado = Epilogo1_8;
404               end
405
406             Epilogo1_8:
407               begin
408                 Estado = Epilogo2_1;
409               end
410
411
412             Epilogo2_1:
413               begin
414                 memory_writeA = 1'b0;
```

```
415                 memory_writeB = 1'b0;
416                 memory_address_registerA = inib + pcc;
417                 memory_address_registerB = inic + pcd;
418                 Estado = Epilogo2_2;
419               end
420
421             Epilogo2_2:
422               begin
423                 Estado = Epilogo2_3;
424               end
425
426
427             Epilogo2_3:
428               begin
429                 memory_writeB = 1'b1;
430                 memory_writeA = 1'b1;
431                 memory_address_registerA = inic + pcc;
432                 memory_address_registerB = inid + pcd;
433                 register_A = memory_data_registerA + 3;
434                 register_B = memory_data_registerB;
435                 pcc= pcc + 1;
436                 pcd= pcd + 1;
437                 Estado = Epilogo2_4;
438               end
439
440             Epilogo2_4:
441               begin
442                 Estado = Epilogo3_1;
443               end
444
445
446
447             Epilogo3_1:
448               begin
449                 memory_writeA = 1'b0;
450                 memory_address_registerA = inic + pcd;
451                 Estado = Epilogo3_2;
452               end
453
454             Epilogo3_2:
455               begin
456                 Estado = Epilogo3_3;
457               end
```

```
458
459
460                 Epilogo3_3:
461                   begin
462                     memory_writeA = 1'b1;
463                     memory_address_registerA = inid + pcd;
464                     register_A = memory_data_registerA;
465                     pcd = pcd + 1;
466                     Estado = Epilogo3_4;
467                   end
468
469
470                 Epilogo3_4:
471                   begin
472                     enable2 = 1'b1;
473                     Estado = Fim;
474                   end
475               endcase
476             end
477       end
478 endmodule
```

Listagem E.4: ***Loop pipelining*** **em Verilog com quatro memórias de duas portas.**

```
1  module LoopPipe4Memo2Port(clock,enable,register_A1,register_A2,register_B1,
       register_B2, register_C1,register_C2,register_D1,register_D2,
2    memory_data_register_outA1,memory_data_register_outA2,
         memory_data_register_outB1,memory_data_register_outB2,
         memory_data_register_outC1,memory_data_register_outC2,
3    memory_data_register_outD1,memory_data_register_outD2,reset,pca,pcb,pcc,
         pcd);
4
5    parameter Inicio        = 1;
6    parameter Prologo1_1    = 2;
7    parameter Prologo1_2    = 3;
8    parameter Prologo1_3    = 4;
9    parameter Prologo1_4    = 5;
10   parameter Prologo2_1    = 6;
11   parameter Prologo2_2    = 7;
12   parameter Prologo2_3    = 8;
13   parameter Prologo2_4    = 9;
14   parameter Prologo3_1     = 10;
15   parameter Prologo3_2     = 11;
16   parameter Prologo3_3     = 12;
```

```
17    parameter Prologo3_4      = 13;
18    parameter Kernel_1         = 14;
19    parameter Kernel_2         = 15;
20    parameter Kernel_3         = 16;
21    parameter Kernel_4         = 17;
22    parameter Epilogo1_1      = 18;
23    parameter Epilogo1_2      = 19;
24    parameter Epilogo1_3      = 20;
25    parameter Epilogo1_4      = 21;
26    parameter Epilogo2_1      = 22;
27    parameter Epilogo2_2      = 23;
28    parameter Epilogo2_3      = 24;
29    parameter Epilogo2_4     = 25;
30    parameter Epilogo3_1     = 26;
31    parameter Epilogo3_2     = 27;
32    parameter Epilogo3_3     = 28;
33    parameter Epilogo3_4     = 29;
34    parameter Fim               =30;
35
36    input clock,enable,reset;
37    output [7:0]pca,pcb,pcc,pcd;
38    output [15:0]register_A1,register_A2,memory_data_register_outA1,
          memory_data_register_outA2;
39    reg enable2;
40    reg [15:0]pca,pcb,pcc,pcd,inia,inib,inic,inid;
41
42    reg [7:0]memory_address_registerA1,memory_address_registerA2;
43    reg memory_writeA1,memory_writeA2;
44    reg [15:0]register_A1,register_A2;
45    wire [15:0]memory_data_registerA1,memory_data_registerA2;
46    wire [15:0]memory_data_register_outA1 = memory_data_registerA1;
47    wire [15:0]memory_data_register_outA2 = memory_data_registerA2;
48    wire [15:0]memory_address_register_outA1 = memory_address_registerA1;
49    wire [15:0]memory_address_register_outA2 = memory_address_registerA2;
50    wire memory_write_outA1 = memory_writeA1;
51    wire memory_write_outA2 = memory_writeA2;
52
53
54
55    output [15:0]register_B1,register_B2,memory_data_register_outB1,
          memory_data_register_outB2;
56
57    reg [7:0]memory_address_registerB1,memory_address_registerB2;
```

```
58    reg memory_writeB1, memory_writeB2;
59    reg [15:0] register_B1, register_B2;
60    wire [15:0] memory_data_registerB1, memory_data_registerB2;
61    wire [15:0] memory_data_register_outB1 = memory_data_registerB1;
62    wire [15:0] memory_data_register_outB2 = memory_data_registerB2;
63    wire [15:0] memory_address_register_outB1 = memory_address_registerB1;
64    wire [15:0] memory_address_register_outB2 = memory_address_registerB2;
65    wire memory_write_outB1 = memory_writeB1;
66    wire memory_write_outB2 = memory_writeB2;
67
68
69
70    output [15:0] register_C1, register_C2, memory_data_register_outC1,
          memory_data_register_outC2;
71
72    reg [7:0] memory_address_registerC1, memory_address_registerC2;
73    reg memory_writeC1, memory_writeC2;
74
75    reg [15:0] register_C1, register_C2;
76    wire [15:0] memory_data_registerC1, memory_data_registerC2;
77    wire [15:0] memory_data_register_outC1 = memory_data_registerC1;
78    wire [15:0] memory_data_register_outC2 = memory_data_registerC2;
79    wire [15:0] memory_address_register_outC1 = memory_address_registerC1;
80    wire [15:0] memory_address_register_outC2 = memory_address_registerC2;
81    wire memory_write_outC1 = memory_writeC1;
82    wire memory_write_outC2 = memory_writeC2;
83
84
85    output [15:0] register_D1, register_D2, memory_data_register_outD1,
          memory_data_register_outD2;
86
87    reg [7:0] memory_address_registerD1, memory_address_registerD2;
88    reg memory_writeD1, memory_writeD2;
89
90    reg [15:0] register_D1, register_D2;
91    wire [15:0] memory_data_registerD1, memory_data_registerD2;
92    wire [15:0] memory_data_register_outD1 = memory_data_registerD1;
93    wire [15:0] memory_data_register_outD2 = memory_data_registerD2;
94    wire [15:0] memory_address_register_outD1 = memory_address_registerD1;
95    wire [15:0] memory_address_register_outD2 = memory_address_registerD2;
96    wire memory_write_outD1 = memory_writeD1;
97    wire memory_write_outD2 = memory_writeD2;
98
```

```
99   altsyncram  altsyncram_component (
100    .clock0 (clock),
101    .wren_a (memory_write_outA1),
102    .address_a (memory_address_register_outA1),
103    .data_a (register_A1),
104    .wren_b (memory_write_outA2),
105    .address_b (memory_address_register_outA2),
106    .data_b (register_A2),
107    .q_a (memory_data_registerA1),
108    .q_b (memory_data_registerA2),
109    .aclr0 (1'b0),
110    .aclr1 (1'b0),
111    .addressstall_a (1'b0),
112    .addressstall_b (1'b0),
113    .byteena_a (1'b1),
114    .byteena_b (1'b1),
115    .clock1 (1'b1),
116    .clocken0 (1'b1),
117    .clocken1 (1'b1),
118    .clocken2 (1'b1),
119    .clocken3 (1'b1),
120    .eccstatus (),
121    .rden_a (1'b1),
122    .rden_b (1'b1));
123  defparam
124    altsyncram_component.address_reg_b = "CLOCK0",
125    altsyncram_component.clock_enable_input_a = "BYPASS",
126    altsyncram_component.clock_enable_input_b = "BYPASS",
127    altsyncram_component.clock_enable_output_a = "BYPASS",
128    altsyncram_component.clock_enable_output_b = "BYPASS",
129    altsyncram_component.indata_reg_b = "CLOCK0",
130    altsyncram_component.intended_device_family = "Cyclone II",
131    altsyncram_component.lpm_type = "altsyncram",
132    altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
133    altsyncram_component.outdata_aclr_a = "NONE",
134    altsyncram_component.outdata_aclr_b = "NONE",
135    altsyncram_component.outdata_reg_a = "UNREGISTERED",
136    altsyncram_component.outdata_reg_b = "UNREGISTERED",
137    altsyncram_component.power_up_uninitialized = "FALSE",
138    altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
139    altsyncram_component.widthad_a = 8,
140    altsyncram_component.widthad_b = 8,
141    altsyncram_component.width_a = 16,
```

```
142     altsyncram_component.width_b = 16,
143     altsyncram_component.width_byteena_a = 1,
144     altsyncram_component.width_byteena_b = 1,
145     altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0";
146
147
148
149
150
151
152   altsyncram  altsyncram_component2 (
153     .clock0 (clock),
154     .wren_a (memory_write_outB1),
155     .address_a (memory_address_register_outB1),
156     .data_a (register_B1),
157     .wren_b (memory_write_outB2),
158     .address_b (memory_address_register_outB2),
159     .data_b (register_B2),
160     .q_a (memory_data_registerB1),
161     .q_b (memory_data_registerB2),
162     .aclr0 (1'b0),
163     .aclr1 (1'b0),
164     .addressstall_a (1'b0),
165     .addressstall_b (1'b0),
166     .byteena_a (1'b1),
167     .byteena_b (1'b1),
168     .clock1 (1'b1),
169     .clocken0 (1'b1),
170     .clocken1 (1'b1),
171     .clocken2 (1'b1),
172     .clocken3 (1'b1),
173     .eccstatus (),
174     .rden_a (1'b1),
175     .rden_b (1'b1));
176   defparam
177     altsyncram_component2.address_reg_b = "CLOCK0",
178     altsyncram_component2.clock_enable_input_a = "BYPASS",
179     altsyncram_component2.clock_enable_input_b = "BYPASS",
180     altsyncram_component2.clock_enable_output_a = "BYPASS",
181     altsyncram_component2.clock_enable_output_b = "BYPASS",
182     altsyncram_component2.indata_reg_b = "CLOCK0",
183     altsyncram_component2.intended_device_family = "Cyclone II",
184     altsyncram_component2.lpm_type = "altsyncram",
```

```
185     altsyncram_component2.operation_mode = "BIDIR_DUAL_PORT",
186     altsyncram_component2.outdata_aclr_a = "NONE",
187     altsyncram_component2.outdata_aclr_b = "NONE",
188     altsyncram_component2.outdata_reg_a = "UNREGISTERED",
189     altsyncram_component2.outdata_reg_b = "UNREGISTERED",
190     altsyncram_component2.power_up_uninitialized = "FALSE",
191     altsyncram_component2.read_during_write_mode_mixed_ports = "OLD_DATA",
192     altsyncram_component2.widthad_a = 8,
193     altsyncram_component2.widthad_b = 8,
194     altsyncram_component2.width_a = 16,
195     altsyncram_component2.width_b = 16,
196     altsyncram_component2.width_byteena_a = 1,
197     altsyncram_component2.width_byteena_b = 1,
198     altsyncram_component2.wrcontrol_wraddress_reg_b = "CLOCK0";
199
200
201   altsyncram  altsyncram_component3 (
202     .clock0 (clock),
203     .wren_a (memory_write_outC1),
204     .address_a (memory_address_register_outC1),
205     .data_a (register_C1),
206     .wren_b (memory_write_outC2),
207     .address_b (memory_address_register_outC2),
208     .data_b (register_C2),
209     .q_a (memory_data_registerC1),
210     .q_b (memory_data_registerC2),
211     .aclr0 (1'b0),
212     .aclr1 (1'b0),
213     .addressstall_a (1'b0),
214     .addressstall_b (1'b0),
215     .byteena_a (1'b1),
216     .byteena_b (1'b1),
217     .clock1 (1'b1),
218     .clocken0 (1'b1),
219     .clocken1 (1'b1),
220     .clocken2 (1'b1),
221     .clocken3 (1'b1),
222     .eccstatus (),
223     .rden_a (1'b1),
224     .rden_b (1'b1));
225   defparam
226     altsyncram_component3.address_reg_b = "CLOCK0",
227     altsyncram_component3.clock_enable_input_a = "BYPASS",
```

```
228     altsyncram_component3.clock_enable_input_b = "BYPASS",
229     altsyncram_component3.clock_enable_output_a = "BYPASS",
230     altsyncram_component3.clock_enable_output_b = "BYPASS",
231     altsyncram_component3.indata_reg_b = "CLOCK0",
232     altsyncram_component3.intended_device_family = "Cyclone II",
233     altsyncram_component3.lpm_type = "altsyncram",
234     altsyncram_component3.operation_mode = "BIDIR_DUAL_PORT",
235     altsyncram_component3.outdata_aclr_a = "NONE",
236     altsyncram_component3.outdata_aclr_b = "NONE",
237     altsyncram_component3.outdata_reg_a = "UNREGISTERED",
238     altsyncram_component3.outdata_reg_b = "UNREGISTERED",
239     altsyncram_component3.power_up_uninitialized = "FALSE",
240     altsyncram_component3.read_during_write_mode_mixed_ports = "OLD_DATA",
241     altsyncram_component3.widthad_a = 8,
242     altsyncram_component3.widthad_b = 8,
243     altsyncram_component3.width_a = 16,
244     altsyncram_component3.width_b = 16,
245     altsyncram_component3.width_byteena_a = 1,
246     altsyncram_component3.width_byteena_b = 1,
247     altsyncram_component3.wrcontrol_wraddress_reg_b = "CLOCK0";
248
249
250
251   altsyncram  altsyncram_component4 (
252     .clock0 (clock),
253     .wren_a (memory_write_outD1),
254     .address_a (memory_address_register_outD1),
255     .data_a (register_D1),
256     .wren_b (memory_write_outD2),
257     .address_b (memory_address_register_outD2),
258     .data_b (register_D2),
259     .q_a (memory_data_registerD1),
260     .q_b (memory_data_registerD2),
261     .aclr0 (1'b0),
262     .aclr1 (1'b0),
263     .addressstall_a (1'b0),
264     .addressstall_b (1'b0),
265     .byteena_a (1'b1),
266     .byteena_b (1'b1),
267     .clock1 (1'b1),
268     .clocken0 (1'b1),
269     .clocken1 (1'b1),
270     .clocken2 (1'b1),
```

```
271       .clocken3 (1'b1),
272       .eccstatus (),
273       .rden_a (1'b1),
274       .rden_b (1'b1));
275     defparam
276       altsyncram_component4.address_reg_b = "CLOCK0",
277       altsyncram_component4.clock_enable_input_a = "BYPASS",
278       altsyncram_component4.clock_enable_input_b = "BYPASS",
279       altsyncram_component4.clock_enable_output_a = "BYPASS",
280       altsyncram_component4.clock_enable_output_b = "BYPASS",
281       altsyncram_component4.indata_reg_b = "CLOCK0",
282       altsyncram_component4.intended_device_family = "Cyclone II",
283       altsyncram_component4.lpm_type = "altsyncram",
284       altsyncram_component4.operation_mode = "BIDIR_DUAL_PORT",
285       altsyncram_component4.outdata_aclr_a = "NONE",
286       altsyncram_component4.outdata_aclr_b = "NONE",
287       altsyncram_component4.outdata_reg_a = "UNREGISTERED",
288       altsyncram_component4.outdata_reg_b = "UNREGISTERED",
289       altsyncram_component4.power_up_uninitialized = "FALSE",
290       altsyncram_component4.read_during_write_mode_mixed_ports = "OLD_DATA",
291       altsyncram_component4.widthad_a = 8,
292       altsyncram_component4.widthad_b = 8,
293       altsyncram_component4.width_a = 16,
294       altsyncram_component4.width_b = 16,
295       altsyncram_component4.width_byteena_a = 1,
296       altsyncram_component4.width_byteena_b = 1,
297       altsyncram_component4.wrcontrol_wraddress_reg_b = "CLOCK0";
298
299
300     integer     i,n;
301     integer     a[11:0];
302     integer     b[10:0];
303     integer     c[10:0];
304     integer     d[10:0];
305
306
307
308     reg [6:0] Estado;
309     reg [6:0] Estado2;
310
311     always @ (posedge clock or posedge reset)
312       begin : LoopPipelining
313         if (reset)  begin
```

```
314             memory_writeA1 = 1'b0;
315             memory_writeA2 = 1'b0;
316             memory_writeB1 = 1'b0;
317             memory_writeB2 = 1'b0;
318             memory_writeC1 = 1'b0;
319             memory_writeC2 = 1'b0;
320             memory_writeD1 = 1'b0;
321             memory_writeD2 = 1'b0;
322             inia = 8'b00000000;
323             inib = 8'b00000000;
324             inic = 8'b00000000;
325             inid = 8'b00000000;
326             pca = 8'b00000000;
327             pcb = 8'b00000000;
328             pcc = 8'b00000000;
329             pcd = 8'b00000000;
330             i = 0;
331             n = 7;
332
333             Estado = Inicio;
334           end
335         else
336           if(enable) begin
337               case (Estado)
338                 Inicio:
339                   begin
340                     Estado = Prologo1_1;
341                   end
342
343                 Prologo1_1:
344                   begin
345                     memory_writeA1 = 1'b0;
346                     memory_address_registerA1 = inia + pca;
347                     Estado = Prologo1_2;
348                   end
349
350                 Prologo1_2:
351                   begin
352                     Estado = Prologo1_3;
353                   end
354
355                 Prologo1_3:
356                   begin
```

```
357                 memory_writeA1 = 1'b1;
358                 memory_address_registerA1 = inia + pca + 1;
359                 register_A1 = memory_data_registerA1 + 1;
360                 pca = pca + 1;
361                 Estado = Prologo1_4;
362               end
363
364             Prologo1_4:
365               begin
366                 Estado = Prologo2_1;
367               end
368
369             Prologo2_1:
370               begin
371                 memory_writeA1 = 1'b0;
372                 memory_writeA2 = 1'b0;
373                 memory_address_registerA1 = inia + pca;
374                 memory_address_registerA2 = inia + pcb + 1;
375                 Estado = Prologo2_2;
376               end
377
378             Prologo2_2:
379               begin
380                 Estado = Prologo2_3;
381               end
382
383
384             Prologo2_3:
385               begin
386                 memory_writeA1 = 1'b1;
387                 memory_writeB1 = 1'b1;
388                 memory_address_registerA1 = inia + pca + 1;
389                 memory_address_registerB1 = inib + pcb;
390                 register_A1 = memory_data_registerA1 + 1;
391                 register_B1 = memory_data_registerA2 + 2;
392                 pca= pca + 1;
393                 pcb= pcb + 1;
394                 Estado = Prologo2_4;
395               end
396
397             Prologo2_4:
398               begin
399                 Estado = Prologo3_1;
```

```
400                 end
401
402
403               Prologo3_1:
404                 begin
405                   memory_writeA1 = 1'b0;
406                   memory_writeA2 = 1'b0;
407                   memory_writeB1 = 1'b0;
408                   memory_address_registerA1 = inia + pca;
409                   memory_address_registerA2 = inia + pcb + 1;
410                   memory_address_registerB1 = inib + pcc;
411                   Estado = Prologo3_2;
412                 end
413
414               Prologo3_2:
415                 begin
416                   Estado = Prologo3_3;
417                 end
418
419
420               Prologo3_3:
421                 begin
422                   memory_writeA1 = 1'b1;
423                   memory_writeB1 = 1'b1;
424                   memory_writeC1 = 1'b1;
425                   memory_address_registerA1 = inia + pca + 1;
426                   memory_address_registerB1 = inib + pcb;
427                   memory_address_registerC1 = inic + pcc;
428                   register_A1 = memory_data_registerA1 + 1;
429                   register_B1 = memory_data_registerA2 + 2;
430                   register_C1 = memory_data_registerB1 + 3;
431                   pca= pca + 1;
432                   pcb= pcb + 1;
433                   pcc = pcc + 1;
434                   Estado = Prologo3_4;
435                 end
436
437
438               Prologo3_4:
439                 begin
440                   Estado = Kernel_1;
441                 end
442
```

```
443                 Kernel_1:
444                   begin
445                     if (i>=n)
446                       Estado = Epilogo1_1;
447                     else
448                       if (i<n)
449                         begin
450                           memory_writeA1 = 1'b0;
451                           memory_writeA2 = 1'b0;
452                           memory_writeB1 = 1'b0;
453                           memory_writeC1 = 1'b0;
454                           memory_address_registerA1 = inia + pca;
455                           memory_address_registerA2 = inia + pcb + 1;
456                           memory_address_registerB1 = inib + pcc;
457                           memory_address_registerC1 = inic + pcd;
458                           Estado = Kernel_2;
459                         end
460                   end
461
462                 Kernel_2:
463                   begin
464                     Estado = Kernel_3;
465                   end
466
467
468                 Kernel_3:
469                   begin
470                     memory_writeA1 = 1'b1;
471                     memory_writeB1 = 1'b1;
472                     memory_writeC1 = 1'b1;
473                     memory_writeD1 = 1'b1;
474                     memory_address_registerA1 = inia + pca + 1;
475                     memory_address_registerB1 = inib + pcb;
476                     memory_address_registerC1 = inic + pcc;
477                     memory_address_registerD1= inid + pcd;
478                     register_A1 = memory_data_registerA1 + 1;
479                     register_B1 = memory_data_registerA2 + 2;
480                     register_C1 = memory_data_registerB1 + 3;
481                     register_D1 = memory_data_registerC1;
482                     pcc = pcc + 1;
483                     pcd = pcd + 1;
484                     pca= pca + 1;
485                     pcb= pcb + 1;
```

```
486                 Estado = Kernel_4;
487               end
488
489
490             Kernel_4:
491               begin
492                 Estado = Kernel_1;
493                 i = i + 1;
494               end
495
496             Epilogo1_1:
497               begin
498                 memory_writeA2 = 1'b0;
499                 memory_writeB1 = 1'b0;
500                 memory_writeC1 = 1'b0;
501                 memory_address_registerA2 = inia + pcb + 1;
502                 memory_address_registerB1 = inib + pcc;
503                 memory_address_registerC1 = inic + pcd;
504                 Estado = Epilogo1_2;
505               end
506
507             Epilogo1_2:
508               begin
509                 Estado = Epilogo1_3;
510               end
511
512
513             Epilogo1_3:
514               begin
515                 memory_writeB1 = 1'b1;
516                 memory_writeC1 = 1'b1;
517                 memory_writeD1 = 1'b1;
518                 memory_address_registerB1 = inib + pcb;
519                 memory_address_registerC1 = inic + pcc;
520                 memory_address_registerD1 = inid + pcd;
521                 register_B1 = memory_data_registerA2 + 2;
522                 register_C1 = memory_data_registerB1 + 3;
523                 register_D1 = memory_data_registerC1;
524                 pcb= pcb + 1;
525                 pcc= pcc + 1;
526                 pcd = pcd + 1;
527                 Estado = Epilogo1_4;
528               end
```

```
529
530             Epilogo1_4:
531               begin
532                 Estado = Epilogo2_1;
533               end
534
535
536             Epilogo2_1:
537               begin
538                 memory_writeB1 = 1'b0;
539                 memory_writeC1 = 1'b0;
540                 memory_address_registerB1 = inib + pcc;
541                 memory_address_registerC1 = inic + pcd;
542                 Estado = Epilogo2_2;
543               end
544
545             Epilogo2_2:
546               begin
547                 Estado = Epilogo2_3;
548               end
549
550
551             Epilogo2_3:
552               begin
553                 memory_writeC1 = 1'b1;
554                 memory_writeD1 = 1'b1;
555                 memory_address_registerC1 = inic + pcc;
556                 memory_address_registerD1 = inid + pcd;
557                 register_C1 = memory_data_registerB1 + 3;
558                 register_D1 = memory_data_registerC1;
559                 pcc= pcc + 1;
560                 pcd= pcd + 1;
561                 Estado = Epilogo2_4;
562               end
563
564             Epilogo2_4:
565               begin
566                 Estado = Epilogo3_1;
567               end
568
569
570             Epilogo3_1:
571               begin
```

```
572                     memory_writeC1 = 1'b0;
573                     memory_address_registerC1 = inic + pcd;
574                     Estado = Epilogo3_2;
575                   end
576
577                 Epilogo3_2:
578                   begin
579                     Estado = Epilogo3_3;
580                   end
581
582
583                 Epilogo3_3:
584                   begin
585                     memory_writeD1 = 1'b1;
586                     memory_address_registerD1 = inid + pcd;
587                     register_D1 = memory_data_registerC1;
588                     pcd = pcd + 1;
589                     Estado = Epilogo3_4;
590                   end
591
592
593                 Epilogo3_4:
594                   begin
595                     enable2 = 1'b1;
596                     Estado = Fim;
597                   end
598               endcase
599             end
600       end
601   endmodule
```

Apendice F

EXEMPLOS DE CÓDIGOS UTILIZADOS NO ALGORITMO DOTPROD MODIFICADO

Neste apêndice serão mostrados alguns códigos em Verilog utilizados para implementar o algoritmo Dotprod modificado. Nas Listagens F.1 e F.2 são mostrados os arquivos *.mif* que especificam o conteúdo inicial para cada endereço das duas memórias utilizadas. Nos exemplos mostrados o número de iterações vai apenas de 0 a 10. As Listagens F.3 e F.4 representam duas formas de implementar o algoritmo Dotprod modificado utilizando a arquitetura alvo tipo 1. Na Listagen F.3 é mostrada uma maneira de implementar o algoritmo Dotprod modificado utilizando a arquitetura alvo tipo 2.

Listagem F.1: Arquivo .mif utilizado na primeira memória dos exemplos abaixo.

```
1
2  WIDTH=16;
3  DEPTH=256;
4
5  ADDRESS_RADIX=UNS;
6  DATA_RADIX=UNS;
7
8  CONTENT BEGIN
9          0    :    1;
10         1    :    2;
11         2    :    3;
12         3    :    4;
13         4    :    5;
14         5    :    6;
15         6    :    7;
16         7    :    8;
```

```
17          8     :     9;
18          9     :     10;
19          10    :     11;
20          11    :     12;
21          12    :     13;
22          [13..255]   :     0;
23  END;
```

Listagem F.2: Arquivo .mif utilizado na segunda memória dos exemplos abaixo.

```
1
2   WIDTH=16;
3   DEPTH=256;
4
5   ADDRESS_RADIX=UNS;
6   DATA_RADIX=UNS;
7
8   CONTENT BEGIN
9           0     :     2;
10          1     :     3;
11          2     :     4;
12          3     :     5;
13          4     :     6;
14          5     :     7;
15          6     :     8;
16          7     :     9;
17          8     :     10;
18          9     :     11;
19          10    :     12;
20          [11..255]   :     0;
21  END;
```

Listagem F.3: Exemplo de implementação do Dotprod modificado utilizando a arquitetura alvo tipo 1.

```
1
2   module LoopPipeliningDotProd3_Arq1_Exemplo1(clock,
3                                               enable,
4                                               register,
5                                               memory_data_register_outA,
6                                               memory_data_register_outB,
7                                               sum_out,
8                                               reset);
9
```

```
10
11      localparam Inicio         = 1;
12      localparam Prologo1_1     = 2;
13      localparam Prologo1_2     = 3;
14      localparam Prologo2_1     = 4;
15      localparam Prologo2_2     = 5;
16      localparam Prologo3_1     = 6;
17      localparam Prologo3_2     = 7;
18      localparam Prologo3_3     = 8;
19      localparam Prologo3_4     = 9;
20      localparam Prologo4_1     = 10;
21      localparam Prologo4_2     = 11;
22      localparam Prologo4_3     = 12;
23      localparam Prologo4_4     = 13;
24      localparam Kernel_1       = 14;
25      localparam Kernel_2       = 15;
26      localparam Kernel_3       = 16;
27      localparam Kernel_4       = 17;
28      localparam Epilogo1_1     = 18;
29      localparam Epilogo1_2     = 19;
30      localparam Epilogo1_3     = 20;
31      localparam Epilogo1_4     = 21;
32      localparam Epilogo2_1     = 22;
33      localparam Epilogo2_2     = 23;
34      localparam Epilogo2_3     = 24;
35      localparam Epilogo2_4     = 25;
36      localparam Epilogo3_1     = 26;
37      localparam Epilogo3_2     = 27;
38      localparam Epilogo4_1     = 28;
39      localparam Fim            = 29;
40
41
42    input clock;
43    input enable;
44    input reset;
45    output [32:0]register;
46
47
48
49
50    output [15:0]memory_data_register_outA;
51    output [15:0]memory_data_register_outB;
52   output [15:0]sum_out;
```

```
53      reg [7:0]pca,pcb;
54      reg [7:0]memory_address_registerA,memory_address_registerB;
55     reg memory_writeA,memory_writeB;
56      reg [15:0]registerA,registerB;
57      wire [15:0]memory_data_registerA,memory_data_registerB;
58      reg [15:0]sum = 0 ;
59      reg [15:0]j ;
60      reg [15:0] mul ;
61     wire [15:0] sum_out = sum;
62      wire [7:0]memory_data_register_outA = memory_data_registerA;
63      wire [7:0]memory_data_register_outB = memory_data_registerB;
64
65      wire [7:0]memory_address_register_outA = memory_address_registerA;
66      wire [7:0]memory_address_register_outB = memory_address_registerB;
67
68      wire memory_write_outA = memory_writeA;
69      wire memory_write_outB = memory_writeB;
70
71      altsyncram    altsyncram_component (
72                .clock0 (clock),
73                .wren_a (memory_write_outA),
74                .address_a (memory_address_register_outA),
75                .data_a (registerA),
76                .wren_b (1'b0),
77                .address_b (),
78                .data_b (),
79                .q_a (memory_data_registerA),
80                .q_b (),
81                .aclr0 (1'b0),
82                .aclr1 (1'b0),
83                .addressstall_a (1'b0),
84                .addressstall_b (1'b0),
85                .byteena_a (1'b1),
86                .byteena_b (1'b1),
87                .clock1 (1'b1),
88                .clocken0 (1'b1),
89                .clocken1 (1'b1),
90                .clocken2 (1'b1),
91                .clocken3 (1'b1),
92                .eccstatus (),
93                .rden_a (1'b1),
94                .rden_b (1'b1));
95      defparam
```

```
96        altsyncram_component.address_reg_b = "CLOCK0",
97        altsyncram_component.clock_enable_input_a = "BYPASS",
98        altsyncram_component.clock_enable_input_b = "BYPASS",
99        altsyncram_component.clock_enable_output_a = "BYPASS",
100       altsyncram_component.clock_enable_output_b = "BYPASS",
101       altsyncram_component.indata_reg_b = "CLOCK0",
102       altsyncram_component.intended_device_family = "Cyclone II",
103       altsyncram_component.lpm_type = "altsyncram",
104       altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
105       altsyncram_component.outdata_aclr_a = "NONE",
106       altsyncram_component.outdata_aclr_b = "NONE",
107       altsyncram_component.outdata_reg_a = "UNREGISTERED",
108       altsyncram_component.outdata_reg_b = "UNREGISTERED",
109       altsyncram_component.power_up_uninitialized = "FALSE",
110       altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
111       altsyncram_component.widthad_a = 8,
112       altsyncram_component.widthad_b = 8,
113       altsyncram_component.width_a = 16,
114       altsyncram_component.width_b = 16,
115       altsyncram_component.width_byteena_a = 1,
116       altsyncram_component.width_byteena_b = 1,
117       altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0",
118       altsyncram_component.init_file ="DotProd1.mif";
119
120
121
122
123   altsyncram   altsyncram_component2 (
124             .clock0 (clock),
125             .wren_a (memory_write_outB),
126             .address_a (memory_address_register_outB),
127             .data_a (),
128             .wren_b (1'b0),
129             .address_b (),
130             .data_b (),
131             .q_a (memory_data_registerB),
132             .q_b (),
133             .aclr0 (1'b0),
134             .aclr1 (1'b0),
135             .addressstall_a (1'b0),
136             .addressstall_b (1'b0),
137             .byteena_a (1'b1),
138             .byteena_b (1'b1),
```

```
139             .clock1 (1'b1),
140             .clocken0 (1'b1),
141             .clocken1 (1'b1),
142             .clocken2 (1'b1),
143             .clocken3 (1'b1),
144             .eccstatus (),
145             .rden_a (1'b1),
146             .rden_b (1'b1));
147     defparam
148         altsyncram_component2.address_reg_b = "CLOCK0",
149         altsyncram_component2.clock_enable_input_a = "BYPASS",
150         altsyncram_component2.clock_enable_input_b = "BYPASS",
151         altsyncram_component2.clock_enable_output_a = "BYPASS",
152         altsyncram_component2.clock_enable_output_b = "BYPASS",
153         altsyncram_component2.indata_reg_b = "CLOCK0",
154         altsyncram_component2.intended_device_family = "Cyclone II",
155         altsyncram_component2.lpm_type = "altsyncram",
156         altsyncram_component2.operation_mode = "BIDIR_DUAL_PORT",
157         altsyncram_component2.outdata_aclr_a = "NONE",
158         altsyncram_component2.outdata_aclr_b = "NONE",
159         altsyncram_component2.outdata_reg_a = "UNREGISTERED",
160         altsyncram_component2.outdata_reg_b = "UNREGISTERED",
161         altsyncram_component2.power_up_uninitialized = "FALSE",
162         altsyncram_component2.read_during_write_mode_mixed_ports = "OLD_DATA"
                ,
163         altsyncram_component2.widthad_a = 8,
164         altsyncram_component2.widthad_b = 8,
165         altsyncram_component2.width_a = 16,
166         altsyncram_component2.width_b = 16,
167         altsyncram_component2.width_byteena_a = 1,
168         altsyncram_component2.width_byteena_b = 1,
169         altsyncram_component2.wrcontrol_wraddress_reg_b = "CLOCK0",
170         altsyncram_component2.init_file ="DotProd2.mif";
171
172
173
174     reg  enqueue4, dequeue4;
175     wire [15:0] head_data4;
176    reg [15:0] tail_data4;
177     wire  enqueue_out4 = enqueue4;
178    wire  dequeue_out4 = dequeue4;
179
180      RegFileFifo5_Posedge fila4(
```

```
181             .reset(reset),
182            .clk(clock),
183            .enqueue(enqueue_out4),
184            .dequeue(dequeue_out4),
185            .head_data(head_data4),
186             .tail_data(tail_data4)
187              );
188     defparam
189      fila4.DATA_WIDTH = 16,
190      fila4.REGFILE_WIDTH = 2;
191
192      integer      i,n;      //contadores
193
194
195
196     reg [6:0] Estado;
197
198
199
200     always @ (posedge clock or posedge reset)
201        begin
202           if (reset)   begin
203                memory_writeA = 1'b0;
204             memory_writeB = 1'b0;
205                 pca = 8'b00000000;
206                 pcb = 8'b00000000;
207                 i = 0;
208                 n = 7;
209                 j = {16{1'b0}};
210                 enqueue4 = 0;
211                 dequeue4 = 0;
212                 sum = 0 ;
213              Estado = Inicio;
214              end
215           else
216           if(enable) begin
217                    case (Estado)
218                       Inicio:
219                          begin
220
221                             Estado = Prologo1_1;
222                          end
223
```

```
224                 Prologo1_1:
225                 begin
226
227                   j = j + 1;
228                   enqueue4 = 1 ;
229                   tail_data4 = j;
230                   Estado = Prologo1_2;
231                 end
232
233                 Prologo1_2:
234                 begin
235
236                   enqueue4 = 0;
237                   dequeue4 = 0;
238                   Estado = Prologo3_1;
239                 end
240
241               Prologo3_1:
242                 begin
243
244                   enqueue4 = 1 ;
245                   j = j + 1;
246                   tail_data4 = j;
247                   memory_address_registerA =  pca;
248                   memory_address_registerB =  pcb;
249                   pca = pca + 1;
250                   pcb = pcb + 1;
251                   Estado = Prologo3_2;
252                 end
253
254               Prologo3_2:
255                 begin
256                   enqueue4 = 0;
257                   dequeue4 = 0;
258
259                    Estado = Prologo4_1;
260                 end
261
262
263
264               Prologo4_1:
265                 begin
266
```

```
267                 j = j + 1;
268                 enqueue4 = 1 ;
269                 tail_data4 = j;
270                 memory_address_registerA =  pca;
271                 memory_address_registerB =  pcb;
272                 mul = memory_data_registerA*memory_data_registerB
                        ;
273                 pca = pca + 1;
274                 pcb = pcb + 1;
275                 Estado = Prologo4_2;
276               end
277
278               Prologo4_2:
279               begin
280                 enqueue4 = 0;
281                 dequeue4 = 0;
282                 Estado = Kernel_1;
283               end
284
285
286             Kernel_1:
287               begin
288                 enqueue4 = 0;
289                 dequeue4 = 0;
290
291                  if (i>=n)
292                  begin
293                     Estado = Epilogo1_1;
294                  end
295                  else
296                  if (i<n)
297                    begin
298                    enqueue4 = 1;
299                    dequeue4 = 1;
300                    j = j + 1;
301                    tail_data4 = j;
302                    memory_address_registerA =  pca;
303                    memory_address_registerB =  pcb;
304                    sum = (sum + mul) + head_data4;
305                    mul = memory_data_registerA*
                           memory_data_registerB;
306                    pca = pca + 1;
307                    pcb = pcb + 1;
```

```
308                     Estado = Kernel_2;
309                   end
310                 end
311
312                 Kernel_2:
313                 begin
314                   enqueue4 = 0;
315                   dequeue4 = 0;
316                   i = i + 1;
317                   Estado = Kernel_1;
318                 end
319
320
321
322
323
324               Epilogo1_1:
325                begin
326                     dequeue4 = 1;
327                     memory_address_registerA =  pca;
328                     memory_address_registerB =  pcb;
329                     sum = (sum + mul) + head_data4;
330                     mul = memory_data_registerA*
                            memory_data_registerB;
331                     pca = pca + 1;
332                     pcb = pcb + 1;
333                     Estado = Epilogo1_2;
334                 end
335
336                Epilogo1_2:
337                 begin
338                  enqueue4 = 0;
339                  dequeue4 = 0;
340                  Estado = Epilogo2_1;
341                 end
342
343               Epilogo2_1:
344                 begin
345                   dequeue4= 1;
346                   sum = (sum + mul) + head_data4;
347                   mul = memory_data_registerA*memory_data_registerB
                        ;
348                   Estado = Epilogo2_2;
```

```
349                             end
350
351
352                         Epilogo2_2:
353                             begin
354                              enqueue4 = 0;
355                              dequeue4 = 0;
356                              Estado = Epilogo3_1;
357                             end
358
359
360                         Epilogo3_1:
361                             begin
362                               dequeue4 = 1;
363                               sum = (sum + mul)+ head_data4;
364                               Estado = Fim;
365                             end
366
367
368
369                     endcase
370                 end
371         end
372
373 endmodule
```

Listagem F.4: Outro exemplo de implementação do Dotprod modificado utilizando a arquitetura alvo tipo 1.

```
1
2
3  module LoopPipeliningDotProd3_Arq1_Exemplo2(clock,
4                                  enable,
5                                  register,
6                                  memory_data_register_outA,
7                                  memory_data_register_outB,
8                                  sum_out,
9                                  reset);
10
11
12       localparam Inicio        = 1;
13      localparam Prologo1_1      = 2;
14      localparam Prologo1_2      = 3;
15      localparam Prologo2_1      = 4;
```

```
16      localparam Prologo2_2      = 5;
17      localparam Prologo3_1    = 6;
18      localparam Prologo3_2    = 7;
19      localparam Prologo3_3    = 8;
20      localparam Prologo3_4    = 9;
21      localparam Prologo4_1    = 10;
22      localparam Prologo4_2    = 11;
23      localparam Prologo4_3    = 12;
24      localparam Prologo4_4    = 13;
25      localparam Kernel_1          = 14;
26      localparam Kernel_2          = 15;
27      localparam Kernel_3          = 16;
28      localparam Kernel_4          = 17;
29      localparam Epilogo1_1      = 18;
30      localparam Epilogo1_2      = 19;
31      localparam Epilogo1_3      = 20;
32      localparam Epilogo1_4      = 21;
33      localparam Epilogo2_1      = 22;
34      localparam Epilogo2_2    = 23;
35      localparam Epilogo2_3      = 24;
36      localparam Epilogo2_4      = 25;
37      localparam Epilogo3_1      = 26;
38      localparam Epilogo3_2      = 27;
39      localparam Epilogo4_1        = 28;
40      localparam Fim               = 29;
41
42
43    input clock;
44    input enable;
45    input reset;
46    output [32:0]register;
47
48
49
50
51    output [15:0]memory_data_register_outA;
52    output [15:0]memory_data_register_outB;
53   output [15:0]sum_out;
54    reg [7:0]pca,pcb;
55    reg [7:0]memory_address_registerA,memory_address_registerB;
56   reg memory_writeA,memory_writeB;
57    reg [15:0]registerA,registerB;
58    wire [15:0]memory_data_registerA,memory_data_registerB;
```

```
59      reg [15:0]sum = 0 ;
60      reg [15:0]j ;
61      reg [15:0] mul ;
62     wire [15:0] sum_out = sum;
63      wire [7:0]memory_data_register_outA = memory_data_registerA;
64      wire [7:0]memory_data_register_outB = memory_data_registerB;
65
66      wire [7:0]memory_address_register_outA = memory_address_registerA;
67      wire [7:0]memory_address_register_outB = memory_address_registerB;
68
69      wire memory_write_outA = memory_writeA;
70      wire memory_write_outB = memory_writeB;
71
72      altsyncram    altsyncram_component (
73                .clock0 (clock),
74                .wren_a (memory_write_outA),
75                .address_a (memory_address_register_outA),
76                .data_a (registerA),
77                .wren_b (1'b0),
78                .address_b (),
79                .data_b (),
80                .q_a (memory_data_registerA),
81                .q_b (),
82                .aclr0 (1'b0),
83                .aclr1 (1'b0),
84                .addressstall_a (1'b0),
85                .addressstall_b (1'b0),
86                .byteena_a (1'b1),
87                .byteena_b (1'b1),
88                .clock1 (1'b1),
89                .clocken0 (1'b1),
90                .clocken1 (1'b1),
91                .clocken2 (1'b1),
92                .clocken3 (1'b1),
93                .eccstatus (),
94                .rden_a (1'b1),
95                .rden_b (1'b1));
96      defparam
97          altsyncram_component.address_reg_b = "CLOCK0",
98          altsyncram_component.clock_enable_input_a = "BYPASS",
99          altsyncram_component.clock_enable_input_b = "BYPASS",
100         altsyncram_component.clock_enable_output_a = "BYPASS",
101         altsyncram_component.clock_enable_output_b = "BYPASS",
```

```
102         altsyncram_component.indata_reg_b = "CLOCK0",
103         altsyncram_component.intended_device_family = "Cyclone II",
104         altsyncram_component.lpm_type = "altsyncram",
105         altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
106         altsyncram_component.outdata_aclr_a = "NONE",
107         altsyncram_component.outdata_aclr_b = "NONE",
108         altsyncram_component.outdata_reg_a = "UNREGISTERED",
109         altsyncram_component.outdata_reg_b = "UNREGISTERED",
110         altsyncram_component.power_up_uninitialized = "FALSE",
111         altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
112         altsyncram_component.widthad_a = 8,
113         altsyncram_component.widthad_b = 8,
114         altsyncram_component.width_a = 16,
115         altsyncram_component.width_b = 16,
116         altsyncram_component.width_byteena_a = 1,
117         altsyncram_component.width_byteena_b = 1,
118         altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0",
119         altsyncram_component.init_file ="DotProd1.mif";
120
121
122
123
124     altsyncram   altsyncram_component2 (
125                 .clock0 (clock),
126                 .wren_a (memory_write_outB),
127                 .address_a (memory_address_register_outB),
128                 .data_a (),
129                 .wren_b (1'b0),
130                 .address_b (),
131                 .data_b (),
132                 .q_a (memory_data_registerB),
133                 .q_b (),
134                 .aclr0 (1'b0),
135                 .aclr1 (1'b0),
136                 .addressstall_a (1'b0),
137                 .addressstall_b (1'b0),
138                 .byteena_a (1'b1),
139                 .byteena_b (1'b1),
140                 .clock1 (1'b1),
141                 .clocken0 (1'b1),
142                 .clocken1 (1'b1),
143                 .clocken2 (1'b1),
144                 .clocken3 (1'b1),
```

```
145             .eccstatus (),
146             .rden_a (1'b1),
147             .rden_b (1'b1));
148     defparam
149         altsyncram_component2.address_reg_b = "CLOCK0",
150         altsyncram_component2.clock_enable_input_a = "BYPASS",
151         altsyncram_component2.clock_enable_input_b = "BYPASS",
152         altsyncram_component2.clock_enable_output_a = "BYPASS",
153         altsyncram_component2.clock_enable_output_b = "BYPASS",
154         altsyncram_component2.indata_reg_b = "CLOCK0",
155         altsyncram_component2.intended_device_family = "Cyclone II",
156         altsyncram_component2.lpm_type = "altsyncram",
157         altsyncram_component2.operation_mode = "BIDIR_DUAL_PORT",
158         altsyncram_component2.outdata_aclr_a = "NONE",
159         altsyncram_component2.outdata_aclr_b = "NONE",
160         altsyncram_component2.outdata_reg_a = "UNREGISTERED",
161         altsyncram_component2.outdata_reg_b = "UNREGISTERED",
162         altsyncram_component2.power_up_uninitialized = "FALSE",
163         altsyncram_component2.read_during_write_mode_mixed_ports = "OLD_DATA"
                ,
164         altsyncram_component2.widthad_a = 8,
165         altsyncram_component2.widthad_b = 8,
166         altsyncram_component2.width_a = 16,
167         altsyncram_component2.width_b = 16,
168         altsyncram_component2.width_byteena_a = 1,
169         altsyncram_component2.width_byteena_b = 1,
170         altsyncram_component2.wrcontrol_wraddress_reg_b = "CLOCK0",
171         altsyncram_component2.init_file ="DotProd2.mif";
172
173
174
175     reg  enqueue4, dequeue4;
176     wire [15:0] head_data4;
177    reg [15:0] tail_data4;
178     wire  enqueue_out4 = enqueue4;
179    wire  dequeue_out4 = dequeue4;
180
181      RegFileFifo5_Posedge fila4(
182             .reset(reset),
183            .clk(clock),
184            .enqueue(enqueue_out4),
185            .dequeue(dequeue_out4),
186            .head_data(head_data4),
```

```
187             .tail_data(tail_data4)
188              );
189     defparam
190      fila4.DATA_WIDTH = 16,
191      fila4.REGFILE_WIDTH = 2;
192
193      integer      i,n;      //contadores
194
195
196
197     reg [6:0] Estado;
198
199
200
201     always @(posedge clock or posedge reset)
202        begin
203           if (reset)   begin
204                 pca <= 8'b00000000;
205                 pcb <= 8'b00000000;
206                 i <= 0;
207                 n <= 7;
208                 j <= {16{1'b0}};
209                 sum <= 0 ;
210              Estado <= Inicio;
211              end
212           else
213           if(enable) begin
214                    case (Estado)
215                       Inicio:
216                          begin
217                            Estado <= Prologo1_1;
218                          end
219
220                          Prologo1_1:
221                          begin
222                            j <= j + 1;
223                            Estado <= Prologo1_2;
224                          end
225
226                          Prologo1_2:
227                          begin
228                            Estado <= Prologo2_1;
229                          end
```

```
230
231                 Prologo2_1:
232                     begin
233                       j <= j + 1;
234                       Estado <= Prologo2_2;
235                     end
236
237                 Prologo2_2:
238                     begin
239                       pca <= pca + 1;
240                       pcb <= pcb + 1;
241                       Estado <= Prologo3_1;
242                     end
243
244
245
246                 Prologo3_1:
247                     begin
248                       j <= j + 1;
249                       mul <= memory_data_registerA*
                              memory_data_registerB;
250                       Estado <= Prologo3_2;
251                     end
252
253                     Prologo3_2:
254                     begin
255                         pca <= pca + 1;
256                         pcb <= pcb + 1;
257                         Estado <= Kernel_1;
258                 end
259
260
261                 Kernel_1:
262                     begin
263                         if (i>=n)
264                         begin
265                             Estado <= Epilogo1_1;
266                         end
267                         else
268                         if (i<n)
269                           begin
270                           j <= j + 1;
271                           sum <= (sum + mul) + head_data4;
```

```
272                     mul <= memory_data_registerA*
                            memory_data_registerB;
273                     Estado <= Kernel_2;
274                      end
275                end
276
277                Kernel_2:
278                begin
279                  pca <= pca + 1;
280                  pcb <= pcb + 1;
281                  i<= i + 1;
282                  Estado <= Kernel_1;
283                end
284
285              Epilogo1_1:
286               begin
287                    sum <= (sum + mul) + head_data4;
288                    mul <= memory_data_registerA*
                           memory_data_registerB;
289                    Estado <= Epilogo1_2;
290                end
291
292               Epilogo1_2:
293                begin
294                 pca <= pca + 1;
295                 pcb <= pcb + 1;
296                 Estado <= Epilogo2_1;
297                end
298
299              Epilogo2_1:
300                begin
301                  sum <= (sum + mul) + head_data4;
302                  mul <= memory_data_registerA*
                         memory_data_registerB;
303                  Estado <= Epilogo2_2;
304                end
305
306
307              Epilogo2_2:
308                begin
309                 pca <= pca + 1;
310                 pcb <= pcb + 1;
311                 Estado <= Epilogo3_1;
```

```
312                         end
313
314
315                       Epilogo3_1:
316                         begin
317                     sum <= (sum + mul)+ head_data4;
318                     Estado <= Fim;
319                         end
320
321                 endcase
322               end
323       end
324
325
326
327         always @ (Estado)
328          begin
329            case (Estado)
330               Inicio:
331                  begin
332                   memory_writeA = 1'b0;
333                     memory_writeB = 1'b0;
334                     enqueue4 = 0;
335                     dequeue4 = 0;
336                  end
337              Prologo1_2:
338                begin
339                 enqueue4 = 1 ;
340                      tail_data4 = j;
341                end
342
343              Prologo2_1:
344                begin
345                  enqueue4 = 0;
346                       dequeue4 = 0;
347                end
348
349              Prologo2_2:
350                begin
351                 enqueue4 = 1 ;
352                      tail_data4 = j;
353                      memory_address_registerA =  pca;
354                      memory_address_registerB =  pcb;
```

```
355                 end
356
357               Prologo3_1:
358                 begin
359                  enqueue4 = 0;
360                  dequeue4 = 0;
361                 end
362
363               Prologo3_2:
364                 begin
365                   enqueue4 = 1 ;
366                   tail_data4 = j;
367                   memory_address_registerA =  pca;
368                   memory_address_registerB =  pcb;
369                 end
370
371                Kernel_1:
372                 begin
373                  enqueue4 = 0;
374                  dequeue4 = 0;
375                 end
376
377                Kernel_2:
378                 begin
379                   enqueue4 = 1;
380                   dequeue4 = 1;
381                   tail_data4 = j;
382                   memory_address_registerA =  pca;
383                   memory_address_registerB =  pcb;
384                 end
385
386                 Epilogo1_1:
387                 begin
388                  enqueue4 = 0;
389                  dequeue4 = 0;
390                 end
391
392                 Epilogo1_2:
393                 begin
394                      dequeue4 = 1;
395                      memory_address_registerA =  pca;
396                      memory_address_registerB =  pcb;
397                 end
```

```
398
399                 Epilogo2_1:
400                 begin
401                     enqueue4 = 0;
402                     dequeue4 = 0;
403                 end
404
405
406                 Epilogo2_2:
407                 begin
408                     dequeue4 = 1;
409                     memory_address_registerA =  pca;
410                     memory_address_registerB =  pcb;
411                 end
412
413                 Epilogo3_1:
414                 begin
415                     enqueue4 = 0;
416                     dequeue4 = 0;
417                 end
418
419                 Fim:
420                 begin
421                     dequeue4 = 1;
422                 end
423                endcase
424           end
425
426 endmodule
```

Listagem F.5: Exemplo de implementação do Dotprod modificado utilizando a arquitetura alvo tipo 2.

```
1
2
3
4  module LoopPipeliningDotProd_Desloc(clock,
5                                      enable,
6                                      register,
7                                      memory_data_register_outA,
8                                      memory_data_register_outB,
9                                      sum_out,
10                                     reset);
11
```

```
12
13      localparam Inicio        = 1;
14      localparam Predicado     = 2;
15     localparam Delay          = 3;
16     localparam Fim            = 4;
17
18    input clock;
19    input enable;
20    input reset;
21    output [32:0]register;
22
23
24
25
26
27    output [15:0]memory_data_register_outA;
28    output [15:0]memory_data_register_outB;
29   output [15:0]sum_out;
30    reg [7:0]pca,pcb;
31    reg [7:0]memory_address_registerA,memory_address_registerB,
          memory_address_registerA2,memory_address_registerB2;
32   reg memory_writeA,memory_writeB;
33    reg [15:0]registerA,registerB;
34    wire [15:0]memory_data_registerA,memory_data_registerB;
35    reg [15:0]sum ;
36    reg [15:0] mul ;
37   wire [15:0] sum_out = sum;
38    wire [15:0]memory_data_register_outA = memory_data_registerA;
39    wire [15:0]memory_data_register_outB = memory_data_registerB;
40    reg [15:0]j ;
41    wire [7:0]memory_address_register_outA = memory_address_registerA;
42    wire [7:0]memory_address_register_outB = memory_address_registerB;
43
44    wire memory_write_outA = memory_writeA;
45    wire memory_write_outB = memory_writeB;
46
47    altsyncram    altsyncram_component (
48              .clock0 (clock),
49              .wren_a (memory_write_outA),
50              .address_a (memory_address_register_outA),
51              .data_a (registerA),
52              .wren_b (1'b0),
53              .address_b (memory_address_registerA2),
```

```
54              .data_b (),
55              .q_a (memory_data_registerA),
56              .q_b (),
57              .aclr0 (1'b0),
58              .aclr1 (1'b0),
59              .addressstall_a (1'b0),
60              .addressstall_b (1'b0),
61              .byteena_a (1'b1),
62              .byteena_b (1'b1),
63              .clock1 (1'b1),
64              .clocken0 (1'b1),
65              .clocken1 (1'b1),
66              .clocken2 (1'b1),
67              .clocken3 (1'b1),
68              .eccstatus (),
69              .rden_a (1'b1),
70              .rden_b (1'b1));
71      defparam
72          altsyncram_component.address_reg_b = "CLOCK0",
73          altsyncram_component.clock_enable_input_a = "BYPASS",
74          altsyncram_component.clock_enable_input_b = "BYPASS",
75          altsyncram_component.clock_enable_output_a = "BYPASS",
76          altsyncram_component.clock_enable_output_b = "BYPASS",
77          altsyncram_component.indata_reg_b = "CLOCK0",
78          altsyncram_component.intended_device_family = "Cyclone II",
79          altsyncram_component.lpm_type = "altsyncram",
80          altsyncram_component.operation_mode = "BIDIR_DUAL_PORT",
81          altsyncram_component.outdata_aclr_a = "NONE",
82          altsyncram_component.outdata_aclr_b = "NONE",
83          altsyncram_component.outdata_reg_a = "UNREGISTERED",
84          altsyncram_component.outdata_reg_b = "UNREGISTERED",
85          altsyncram_component.power_up_uninitialized = "FALSE",
86          altsyncram_component.read_during_write_mode_mixed_ports = "OLD_DATA",
87          altsyncram_component.widthad_a = 8,
88          altsyncram_component.widthad_b = 8,
89          altsyncram_component.width_a = 16,
90          altsyncram_component.width_b = 16,
91          altsyncram_component.width_byteena_a = 1,
92          altsyncram_component.width_byteena_b = 1,
93          altsyncram_component.wrcontrol_wraddress_reg_b = "CLOCK0",
94          altsyncram_component.init_file ="DotProd1.mif";
95
96
```

```
97
98
99      altsyncram    altsyncram_component2 (
100             .clock0 (clock),
101             .wren_a (memory_write_outB),
102             .address_a (memory_address_register_outB),
103             .data_a (registerB),
104             .wren_b (1'b0),
105             .address_b (memory_address_registerB2),
106             .data_b (),
107             .q_a (memory_data_registerB),
108             .q_b (),
109             .aclr0 (1'b0),
110             .aclr1 (1'b0),
111             .addressstall_a (1'b0),
112             .addressstall_b (1'b0),
113             .byteena_a (1'b1),
114             .byteena_b (1'b1),
115             .clock1 (1'b1),
116             .clocken0 (1'b1),
117             .clocken1 (1'b1),
118             .clocken2 (1'b1),
119             .clocken3 (1'b1),
120             .eccstatus (),
121             .rden_a (1'b1),
122             .rden_b (1'b1));
123     defparam
124        altsyncram_component2.address_reg_b = "CLOCK0",
125        altsyncram_component2.clock_enable_input_a = "BYPASS",
126        altsyncram_component2.clock_enable_input_b = "BYPASS",
127        altsyncram_component2.clock_enable_output_a = "BYPASS",
128        altsyncram_component2.clock_enable_output_b = "BYPASS",
129        altsyncram_component2.indata_reg_b = "CLOCK0",
130        altsyncram_component2.intended_device_family = "Cyclone II",
131        altsyncram_component2.lpm_type = "altsyncram",
132        altsyncram_component2.operation_mode = "BIDIR_DUAL_PORT",
133        altsyncram_component2.outdata_aclr_a = "NONE",
134        altsyncram_component2.outdata_aclr_b = "NONE",
135        altsyncram_component2.outdata_reg_a = "UNREGISTERED",
136        altsyncram_component2.outdata_reg_b = "UNREGISTERED",
137        altsyncram_component2.power_up_uninitialized = "FALSE",
138        altsyncram_component2.read_during_write_mode_mixed_ports = "OLD_DATA"
               ,
```

```
139         altsyncram_component2.widthad_a = 8,
140         altsyncram_component2.widthad_b = 8,
141         altsyncram_component2.width_a = 16,
142         altsyncram_component2.width_b = 16,
143         altsyncram_component2.width_byteena_a = 1,
144         altsyncram_component2.width_byteena_b = 1,
145         altsyncram_component2.wrcontrol_wraddress_reg_b = "CLOCK0",
146         altsyncram_component2.init_file ="DotProd2.mif";
147
148
149
150     reg delay;
151
152     reg  enqueue4, dequeue4;
153     wire [15:0] head_data4;
154   reg [15:0] tail_data4;
155     wire  enqueue_out4 = enqueue4;
156   wire  dequeue_out4 = dequeue4;
157
158       RegFileFifo5_Posedge fila4(
159             .reset(reset),
160             .clk(clock),
161             .enqueue(enqueue_out4),
162             .dequeue(dequeue_out4),
163             .head_data(head_data4),
164              .tail_data(tail_data4)
165               );
166     defparam
167       fila4.DATA_WIDTH = 16,
168       fila4.REGFILE_WIDTH = 2;
169
170
171
172       wire     [3:0]     dout;
173       wire               si;
174       wire          enableShift_out;
175       reg           enableShift;
176       assign        enableShift_out = enableShift;
177
178          RegFileShift3_Posedge regshift(
179           .si(si),
180           .clock(clock),
181           .reset(reset),
```

```
182         .enable(enableShift_out),
183         .dout(dout));
184         defparam
185         regshift.SHIFT_WIDTH = 4;
186
187      integer      i,n;      //contadores
188
189     assign        si = (i>=10)?0:1;
190
191     reg [6:0] Estado;
192
193
194
195     always @ (posedge clock or posedge reset)
196        begin
197           if (reset)   begin
198               memory_writeA = 1'b0;
199               memory_writeB = 1'b0;
200                 pca = 8'b00000000;
201                 pcb = 8'b00000000;
202                 i = 0;
203                 n = 13;
204                 j = {16{1'b0}};
205                 enqueue4 = 0;
206                 dequeue4 = 0;
207                 sum = 0;
208                 delay = 0;
209                 enableShift = 1;
210                 Estado = Inicio;
211              end
212           else
213              if(enable) begin
214                    case (Estado)
215                       Inicio:
216                          begin
217                            enableShift = 0;
218                             Estado = Predicado;
219                          end
220
221                          Predicado :
222                          begin
223                           enableShift = 1;
224                             if (i>=n)
```

```
225                     Estado = Fim;
226                 else
227                 if (i<n)
228                   begin
229
230                     if(dout[0])
231                       begin
232                       j = j + 1;
233                       enqueue4 = 1 ;
234                       tail_data4 = j;
235                       end
236                     else
237                       enqueue4 = 0;
238
239                     if (dout[1])
240                         begin
241                           memory_address_registerA =  pca;
242                           memory_address_registerB =  pcb;
243                           pca = pca + 1;
244                           pcb = pcb + 1;
245                           delay=1;
246                         end
247
248
249
250                      if(dout[3])
251                      begin
252                        dequeue4 = 1;
253                       sum = (sum + mul) + head_data4;
254                      end
255                      else
256                        dequeue4 = 0;
257
258
259
260                     if(dout[2])
261                        mul = memory_data_registerA*
                               memory_data_registerB;
262                     Estado = Delay;
263                     end
264
265
266               end
```

```
267
268                             Delay:
269                             begin
270                                 i = i + 1;
271                                 dequeue4= 0;
272                                 enqueue4 = 0;
273                                 enableShift = 0 ;
274                                 Estado = Predicado;
275                             end
276
277
278
279
280
281                     endcase
282                 end
283         end
284
285 endmodule
```