Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Departamento de Engenharia Elétrica

RELATÓRIO DE ESTÁGIO INTEGRADO

Desenvolvimento de um gerador de modelos de referência para compiladores de memória

VICTOR DE SOUSA CAVALCANTE

Campina Grande

Maio de 2013

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RELATÓRIO DE ESTÁGIO INTEGRADO

Desenvolvimento de um gerador de modelos de referência para compiladores de memória

Estagiário: Victor de Sousa Cavalcante

Empresa: ARM Ltda.

Período de Estágio: 21 de Fevereiro a 29 de Julho de 2011

Supervisor: Bastien Aghetti

Orientador: Antônio Marcus Nogueira Lima

Campina Grande

Maio de 2013

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

RELATÓRIO DE ESTÁGIO INTEGRADO

Desenvolvimento de um gerador de modelos de referência para compiladores de memória

Relatório de Estágio Integrado apresentado ao Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande em cumprimento às exigências para obtenção do grau de Bacharel em Engenharia Elétrica

ALUNO: _____

Victor de Sousa Cavalcante

ORIENTADOR: _____

Antônio Marcus Nogueira Lima

Campina Grande

Maio de 2013

AGRADECIMENTOS

Primeiramente, eu gostaria de agradecer Christophe Frey, o gerente de projetos do centro no escritório da ARM em Grenoble, e Jean-Luc Pelloie, o representante legal da *ARM France*, pela oportunidade ímpar de realizar o meu estágio integrado dentro de uma das maiores empresas de microeletrônica.

Igualmente, eu sou muito grato ao meu supervisor Bastien Aghetti. Sua disponibilidade, conhecimento técnico, apoio total e paciência fizeram dele um grande conselheiro. Eu, certamente, não poderia ter um melhor supervisor do que ele.

Agradeço a meus familiares que sempre conversavam comigo durante o meu intercâmbio apesar da distância e do fuso-horário. Especialmente à minha irmã Eugênia Cavalcante e aos meus pais, Antonia Cavalcante e José Ivan Cavalcante.

Ainda, gostaria de agradecer aos meus grandes amigos Arthur Monteiro, João Viniccius Gomes, Matheus Telles, Rafael Ângelo Vieira e Túlio Vidal que me acompanharam desde o início do curso compartilhando momentos bons e ruins.

Igualmente, agradeço a minha namorada Tiziana Mombelli que eu tanto amo e que apesar de morar a milhares de quilômetros de distância ela é a melhor companheira que eu poderia ter, me acalmando, ajudando e amando nas horas mais difíceis de maneira incomparável.

Além disso, eu gostaria de agradecer toda a equipe de memória pelos seus conselhos, compartilhamento de conhecimentos e convivência que contribuíram em desenvolver uma ótima atmosfera de trabalho.

Do mesmo modo, eu sou bastante grato a Andrew Sowden, Saravanan Manickanainar Dakshinamoorthy e Viney Gautam por terem seguido o meu trabalho periodicamente apesar da distância e por terem me aconselhado e ajudado em alguns momentos de dificuldade.

Finalmente, mas não menos importante, gostaria de agradecer a todos os colegas de trabalho na ARM em Grenoble pelo excelente acolhimento e suas contribuições, fazendo o escritório um local agradável de trabalhar.

Dedico este trabalho à minha mãe e ao meu pai que me apoiaram durante esta longa caminhada. A Tizi que tem me proporcionado momentos felizes. Aos meus grandes amigos que sem eles eu não chegaria até aqui.

RESUMO

As memórias são um dos circuitos mais densos em número de transistores por unidade de área. A confiabilidade dos circuitos é um ponto muito importante e focado pelos projetistas de circuitos integrados. Com isso, os projetistas planejam mais tipos de simulações a serem executadas durante a fase de projeto do dispositivo. Um desses tipos de simulação utiliza modelos de referência descritos em *Verilog* para analisar a lógica do circuito.

Atualmente, o fluxo de desenvolvimento de compiladores de memória da ARM possui alguns pontos fracos. Dentre os principais, destaca-se a ausência de uma cobertura funcional completa e o tempo de alteração dos complexos modelos de referência em *Verilog* atuais. O objetivo principal deste estágio foi propor um novo modo de descrever os modelos de referência, eliminando as suas fraquezas anteriormente citadas. Outro foco do trabalho foi o de encontrar uma ferramenta alternativa para o ESP-CV, devido a dificuldades internas enfrentadas pela empresa.

Inicialmente, foi realizada a concepção de um modelo simples com o intuito de avaliar a viabilidade desse tipo de modelo. Posteriormente, o novo gerador de modelos de referência de uma arquitetura foi desenvolvido e avaliado, destacando suas vantagens e desvantagens. Assim, os mesmos procedimentos foram repetidos para outra arquitetura. Com isso, uma avaliação mais precisa foi realizada para esta nova metodologia através de um estudo mais detalhado dos resultados.

Por fim, os objetivos principais foram alcançados com um novo fluxo de desenvolvimento proposto. Embora não tenha sido encontrado um substituto ideal para o ESP-CV, foi utilizada a verificação formal com o Conformal para os modelos de circuitos digitais. Os módulos analógicos continuam sendo simulados com o mesmo *software*.

Palavras-Chaves: ARM; Arquitetura de Sistemas Digitais; Compilador de Memória; Memória; Microeletrônica; Modelo de Referência; SRAM; Verificação Formal; Verificação Funcional; *Verilog*.

ABSTRACT

Memories are one of the densest electronic circuits in respect to the number of transistors per unit area. The reliability of new circuits has been a very important point for designers of integrated circuits. Therefore, they plan to deal with a wider sort of simulations to be executed during the phase of design of this device. One of these sorts of simulations uses reference models written in *Verilog* to analyze the circuit logic.

Nowadays, the design flow of ARM memory compilers has some weaknesses. Among those weaknesses, it will be treated here the absence of a complete functional verification and the level of complexity to modify the current *Verilog* reference models. The main goal of this internship is to propose a different way of writing *Verilog* reference models, suppressing those weaknesses. This internship also aims to find an alternative software to the ESP-CV due to internal affairs faced by the company.

Firstly, a concept of a simple model was made to evaluate the viability of this sort of model. A reference model generator for the same architecture was afterward developed and its advantages and disadvantages evaluated. Thereafter, the same proceedings were made to a different architecture. Therefore, a better evaluation for this new methodology was acquired through a more detailed study.

Finally, the purpose of a new design flow has brought the possibility to reach the main targets. Although a substitute for the ESP-CV was not found, the Conformal was used to formally verify the digital circuits. The analog ones continue being verified by the same software.

Keywords: ARM; Architecture of Digital Systems; Formal Verification; Functional Verification; Memory Compiler; Memory; Microelectronics; Reference Model; SRAM; Verilog.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

#

2D – Duas Dimensões

3D – Três Dimensões

A

ARM – *Advanced RISC Machine*

ASIC – *Application Specific Integrated Circuit*

B

BIST – *Built-in Self Test*

BL – Bit-line

C

CI – *Circuito Integrado*

CMOS – *Complementary Metal-Oxide Semiconductor*

CPU – *Central Processing Unit*

D

DH – *Design House*

DSL – *Digital Subscriber Line*

DSP – *Digital Signal Processor*

E

EDA – *Electronic Design Automation*

ESD – *Electrostatic Discharge*

G

GDS – *Graphic Database System*

GPS – *Global Positioning System*

H

HPP – *High Performance Process*

I

IP – *Intellectual Property*

M

MCU – Microcontrolador

MUX – Multiplexador

N

NMOS – *N-type Metal-Oxide-Semiconductor*

P

PDA – *Personal Digital Assistant*

PG – *Power Gating*

PIPD – *Physical IP Division*

PMOS – *P-type Metal-Oxide-Semiconductor*

R

RAM – *Random Access Memory*

RF1 – *Register File Two-Port*

RF2 – *Register File Single-Port*

RISC – *Reduced Instruction Set Computing*

ROM – *Read Only Memory*

S

SPICE – *Simulated Program with Integrated Circuits Emphasis*

SoC – *System-on-Chip*

SOI – *Silicon-on-Insulator*

SOISIC – *Silicon on Insulator Systems and Integrated Circuits*

SRAM – *Static Random Access Memory*

STB – *Set-top Box*

V

VLSI – *Very-Large-Scale Integration*

W

WL – *Word-line*

SUMÁRIO

1. Introdução

A microeletrônica está em desenvolvimento acelerado, tornando-se cada vez mais competitiva nas últimas décadas. Com o intuito de se manterem no mercado, os projetistas de circuitos integrados têm visto a confiabilidade dos circuitos como um dos pontos principais na execução do projeto. Durante o desenvolvimento, esta é a fase em que o custo para se corrigir erros é menor. Depois que o produto já está no mercado, esse custo é bastante elevado e isso causará uma imagem negativa para a empresa. Portanto, uma atenção suplementar tem sido prestada pelos mesmos, a fim de realizar uma melhor verificação dos circuitos antes da sua inserção no mercado. Essa confiabilidade é comprometida principalmente pelo aumento do número de transistores por unidade de área dentro dos circuitos integrados. Esse aumento foi previsto pela lei de Moore desde 1970, a qual descreveu uma tendência de longo prazo no aumento da quantidade de transistores dentro de um circuito integrado. Segundo essa lei, ocorre aproximadamente uma duplicação da quantidade de transistores que são colocados em um circuito integrado a cada dois anos. Essa tendência tem continuado por mais de meio século e espera-se que essa tendência ainda seja mantida nos próximos anos [1]. Esse crescimento em progressão geométrica é ilustrado na Figura 1 desde a década de 70 à década atual.

Em termos de quantidade de transistores, as memórias são circuitos bastante densos comparados aos que existem atualmente. Devido à miniaturização desses circuitos integrados, os projetistas passaram a ter maior cautela também no desenvolvimento desses dispositivos. Com isso, uma quantidade cada vez maior de testes e simulações durante a fase de projeto de memórias é realizada, garantindo uma maior confiabilidade das mesmas. Dentre os vários tipos de simulações realizadas durante a fase de projeto das memórias, existe uma que verifica apenas a lógica do circuito. Essa simulação compara o circuito analógico com um modelo de referência. O circuito analógico é descrito utilizando o *Simulation Program with Integrated Circuit Emphasis* [7] (SPICE) que descreve cada transistor e as ligações entre eles. O modelo de referência é descrito em *Verilog* [8], uma linguagem de descrição de *hardware*. Tal modelo descreve apenas a lógica comportamental do

circuito. Com isso, é feita uma simulação do circuito analógico com o modelo de referência, considerando o circuito como sendo um circuito digital.

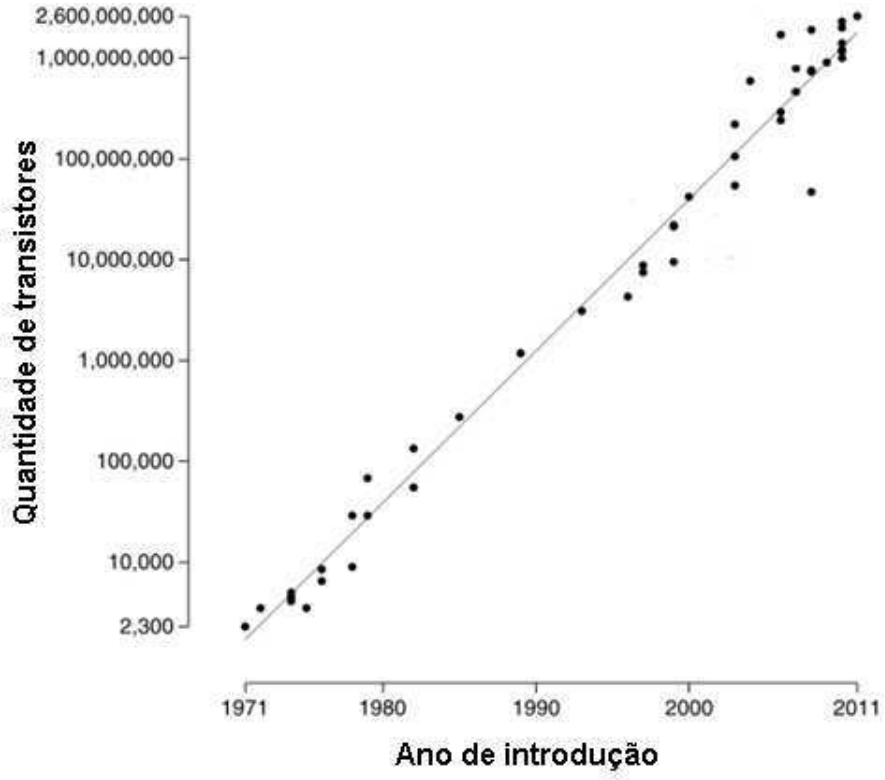

Figura 1: Crescimento do número de transistores em CI nas últimas décadas [1]

Em posse da especificação de certo tipo de memória, uma equipe faz a descrição do modelo de referência em *Verilog* enquanto outra equipe isolada geograficamente da primeira faz o projeto da memória e do compilador de memória. Compiladores de Memória são ferramentas que geram memórias do tipo especificado a partir de parâmetros fornecidos pelo usuário. Esses parâmetros podem, por exemplo, ser a quantidade de posições que define o tamanho da memória, a quantidade de bits de cada palavra guardada, assim como a configuração de funções especiais como pipeline, multiplexadores e redundância.

1.1 Objetivos Gerais

O trabalho realizado durante este estágio integrado tinha como principal objetivo propor uma melhora no gerador automático de modelos de referência em *Verilog*. Esse gerador automático é integrado ao compilador de memória. Desta forma, o trabalho consistiu no estudo da viabilidade de alterar o fluxo de desenvolvimento de compiladores de memórias utilizado atualmente na ARM.

O novo fluxo proposto, a ser apresentado mais a seguir neste texto, deve ser mais rápido no tempo de desenvolvimento dos modelos de referência. Nos casos de possíveis alterações nesses modelos, também será apresentada uma diminuição no tempo de correção. O novo fluxo tem como objetivo ser mais rápido também durante as simulações.

1.2 Objetivos Específicos

Para realizar este estudo, foram desenvolvidos modelos de referência hierárquicos em *Verilog* para um dado tipo de arquitetura de memória. A primeira equipe mencionada anteriormente descreve um modelo comportamental em apenas um arquivo e sem hierarquia. Também foi feita a analise dos efeitos causados por esta mudança para um dado tipo de memoria e das vantagens e desvantagens em integrar esse novo método ao fluxo atual do desenvolvimento de compiladores de memória da empresa.

Primeiramente, é realizada a concepção de um modelo simples com o intuito de avaliar a viabilidade desse novo método. Posteriormente, o gerador de modelos de referência de uma mesma arquitetura é desenvolvido, avaliado e subsequentemente as vantagens e desvantagens desta metodologia.

Este trabalho preliminar é considerado o primeiro passo sólido para desenvolver, validar e promover, por conseguinte essa nova metodologia de descrever modelos de referência em *Verilog*. Caso seja provado que a empresa terá benefícios ao adotar esse novo método, o estudo desenvolvido será adicionado ao fluxo atual de desenvolvimento de compiladores de memórias.

1.3 Cronograma Inicial

Inicialmente, o estágio foi dividido em diversas tarefas, como:

- Aprendizado sobre o básico de memórias (1);
- Entendimento do circuito da memória mais básica (2);
- Modelagem comportamental em *Verilog* da *bit-cell* (3);
- Modelagem comportamental em *Verilog* das *leaf-cells* (4);
- Realizar simulações dos modelos desenvolvidos das *leaf-cells* com os circuitos analógicos das mesmas já existentes em SPICE (5);

- Construção manual do modelo hierárquico para a instância mais simples possível (6);
- Realizar simulação do modelo em *Verilog* hierárquico desenvolvido com o circuito já existente em SPICE (7);
- Escrever o relatório de estágio integrado (8).

O estágio integrado na ARM foi realizado durante 23 semanas dos dias 21 de Fevereiro de 2011 a 29 de Julho de 2011. Apesar de algumas mudanças a serem apresentadas na Sessão 6, o cronograma foi dividido da seguinte maneira com respeito às atividades anteriormente citadas:

Tabela 1: Cronograma inicial do estágio integrado

Semanas/ Tarefas	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1																							
2																							
3																							
4																							
5																							
6																							
7																							
8																							

2. Introdução à ARM

2.1 Visão Geral

A ARM é a empresa líder no fornecimento de *IP cores* para a indústria de semicondutores [4]. *IP cores* são propriedades intelectuais para circuitos integrados. A empresa apenas desenvolve e licencia as propriedades intelectuais ao invés de fabricar e revender circuitos integrados. A ARM é mais conhecida por seus núcleos de microprocessadores e suas arquiteturas que estão presentes em grande parte dos eletrônicos atualmente. Além disso, a empresa também projeta, licencia e vende ferramentas de desenvolvimento de *software* e sistemas embarcados [4].

2.2 História

A empresa foi fundada em Novembro de 1990 como *Advanced RISC Machines Ltd* e foi estruturada como um empreendimento conjunto entre *Acorn Computers*, *Apple Computer* e *VLSI Technology*. Essas empresas tiveram um importante papel no âmbito da microeletrônica, como por exemplo, a *Acorn Computers* que foi a desenvolvedora do primeiro *chip* comercial do mundo que era um processador *Reduced Instruction Set Computer* (RISC) e a *Apple Computers* que desenvolveu a tecnologia RISC e criou um novo padrão de processadores bastante utilizado atualmente: a arquitetura ARM [3].

A ARM criou a primeira arquitetura RISC de baixo custo. Entretanto, outras arquiteturas focadas em maximizar o desempenho foram primeiramente utilizadas por empresas com tecnologia de ponta. No início da década de 1990, com a introdução do seu primeiro processador RISC embarcado, a família de processadores ARM6™, a ARM designou *VLSI Technology* como a sua primeira licença. No passar dos anos, a empresa expandiu significativamente o seu portfólio de *IP cores* e a quantidade de parceiros [3].

Desde 1999, a ARM estabeleceu uma política de expansão da empresa com uma estratégia de adquirir empreendimentos especializados no assunto. Na divisão chamada *Physical IP Division* (PIPD) onde o trabalho citado neste texto foi desenvolvido, a ARM adquiriu a *Artisan Components* em Dezembro de 2004, uma desenvolvedora de IP físico, e SOISIC em 2006 [3], a empresa especializada em IP

físico utilizando uma tecnologia chamada *Silicon-On-Insulator* (SOI). A empresa atualmente domina o mercado de circuitos integrados na telefonia móvel [4] e atua também em vários outros nichos da eletrônica.

Hoje, a ARM atua em diferentes países com escritórios e centros de desenvolvimento, mais conhecidos como *Design Houses*, principalmente no Reino Unido, França, Índia e nos Estados Unidos. A Figura 2 indica todos os locais ao redor do mundo onde existem escritórios da companhia até o atual momento [4]. O escritório de Cambridge, no Reino Unido, representa a matriz da empresa, enquanto que os escritórios de São José, nos Estados Unidos, e Shangai, na China, representam respectivamente os escritórios principais da América do Norte e Ásia. O trabalho descrito neste relatório foi desenvolvido em Grenoble, na França, com a ajuda de funcionários dos escritórios de Bangalore, na Índia, e Austin, nos Estados Unidos.

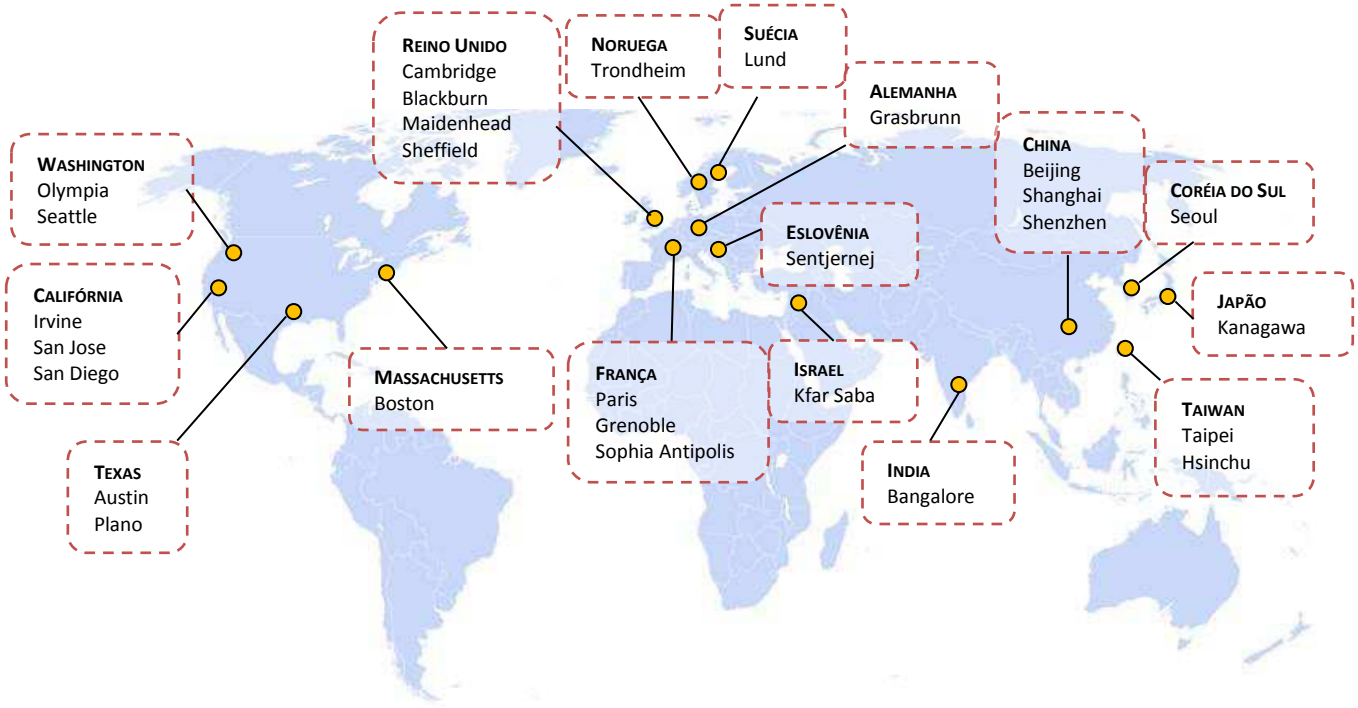

Figura 2: Localização dos escritórios da ARM *(© ARM Ltda.)*

2.3 Produtos e Soluções

A ARM desenvolve os seus produtos para cinco tipos diferentes de domínios de aplicação, são eles: residencial, dispositivos embarcados, empresarial, soluções embarcadas e aplicações emergentes [4].

O mercado para aplicações residenciais reúne todos os equipamentos eletrônicos que podem ser encontrados em residências. Dentre eles, encontram-se principalmente os conversores de televisão digital, também conhecidos como *set-top box* (STB), câmeras fotográficas digitais, televisões e aparelhos de jogos portáteis.

Dispositivos embarcados são aplicações alimentadas por bateria. A ARM fornece uma vasta gama de soluções para *smartphones*, *tablets*, aparelhos de multimídia, assistentes pessoais digitais também conhecidos como *Personal Digital Assistant* (PDA), e dispositivos de navegação GPS.

As soluções empresariais são dispositivos usados por nos diariamente para capturar, armazenar, transferir e alterar dados em discos rígidos, cartões de memória, computadores e impressoras. Esses são também dispositivos que nos ajudam em nossas atividades de trabalho e facilitam as nossas tarefas diárias como os *switches* de rede, *modens* DSL, roteadores e soluções *wireless*.

Soluções embarcadas são dispositivos eletrônicos embarcados ligados à indústria automotiva onde os eletrônicos têm se tornado incrivelmente mais importante nos veículos atuais. Isto concerne na indústria de controle, medição, mensuração, controle de motores, dispositivos para biomedicina e eletrodomésticos de consumo como, por exemplo, máquinas de lavar roupa e brinquedos.

As aplicações emergentes levam produtos inovadores e soluções de alta tecnologia para o mercado. Para ter sucesso neste fim, a ARM tenta promover esses produtos com investimentos significativos na área de pesquisa e desenvolvimento.

2.4 Organização da Empresa

A ARM Ltda. é estruturada em cinco divisões diferentes como pode ser visto na Figura 3 e são nomeadas internamente de: Divisão de IP para Processadores (*Processor IP Division)*, Divisão de IP Físicos (*Physical IP Division)*, Divisão de Projetos de Sistemas (*System Design Division)*, Divisão de Processamento Multimídia (*Media Processing Division)* e Divisão de Serviços (*Services Division)* [2].

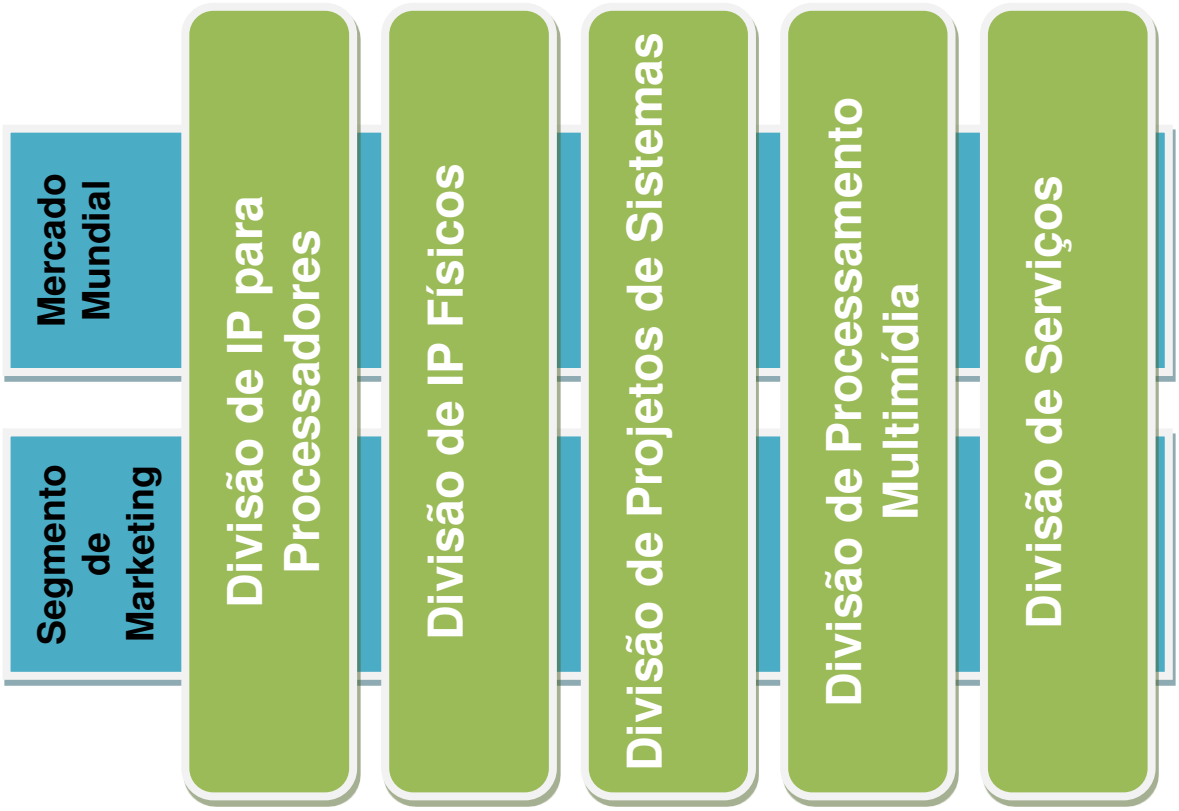

Figura 3: Diagrama da organização da ARM *(© ARM Ltda.)*

- Divisão de IP para Processadores: Inclui todo o desenvolvimento de microprocessadores, fazendo essa divisão a mais importante com 65% da receita da empresa;
- Divisão de IP Físicos: Compõe o desenvolvimento de todos os blocos básicos necessários para o projeto de um *System-On-Chip* (SoC). Essa divisão representa 20% da receita da empresa;
- Divisão de Projetos de Sistemas: Essa divisão é a combinação de duas áreas. A área de Desenvolvimento de Sistemas (*Development Systems*) é responsável por desenvolver sistemas, no qual proporcionam ferramentas e modelos usados para criar e depurar *software* e projetos de SoC. A outra área é chamada de Estruturação de IP (*Fabric IP*) que desenvolve *IP cores* para sistemas como, por exemplo, barramentos e controladores que conectam blocos funcionais (microprocessadores e blocos DSP) dentro de um SoC;
- Divisão de Processamento Multimídia: Essa divisão agrupa a Unidade da Empresa para Mecanismos de Dados (*Data Engines Business Unit*) que desenvolve DSP configuráveis para serem usados em algoritmos complexos para processamento de dados; a Unidade da Empresa para *Software*

Embarcado (*Embedded Software Business Unit*) que desenvolve *software* como, por exemplo, gerenciamento de energia, segurança e aceleração *Java e* a Unidade da Empresa para Gráficos (*Graphic Business Unit*) que desenvolve *IP cores* que criam gráficos de alta definição em 2D e 3D;

- Divisão de Serviços: Presta suporte e manutenção para os produtos da ARM, realizando treinamentos e prestando serviços de consultoria.

O trabalho desenvolvido e citado neste relatório de estágio foi realizado na Divisão de IP Físicos no centro de projetos em Grenoble.

A Figura 4 contém um gráfico de pizza que resume a participação que cada uma das cinco divisões supracitadas tem em seus respectivos mercados representados pelas suas fatias da receita da empresa [1].

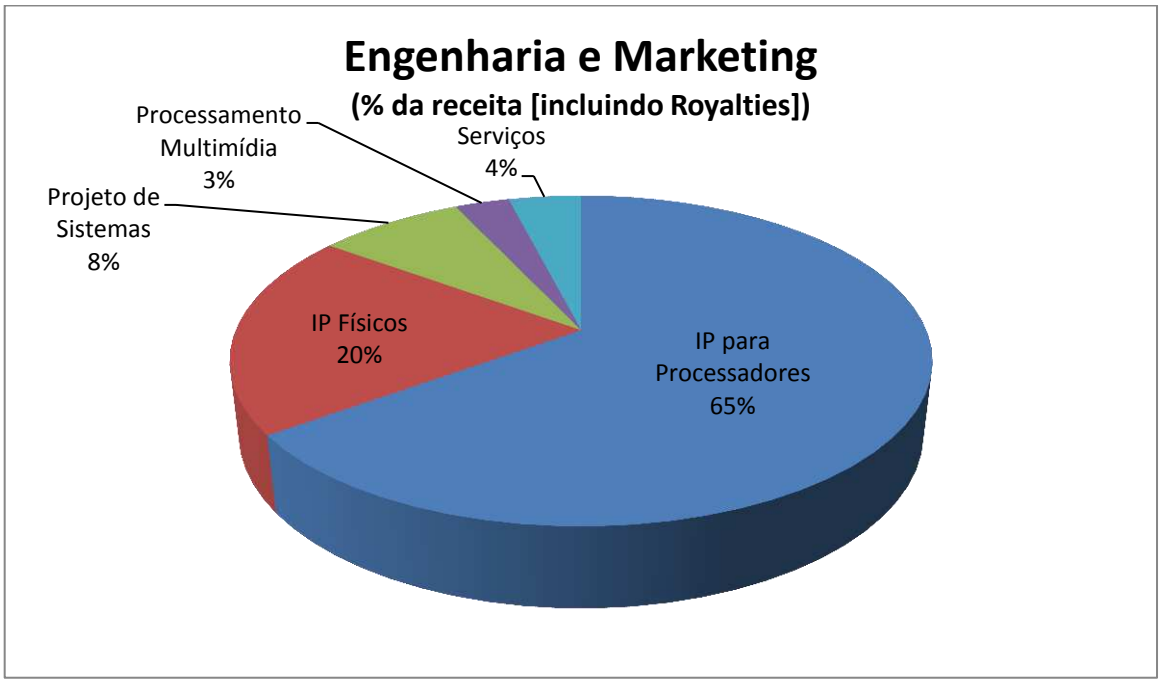

Figura 4: A receita da ARM separada por divisão *(© ARM Ltda. - 2006)*

2.5 A ARM na França

A empresa está implantada na França legalmente sob a entidade *ARM France*. Os escritórios da *ARM France* são:

- Uma subsidiaria da Divisão de IP para Processadores localizada no polo científico de Sophia-Antipolis;
- Um escritório de vendas situado em Paris;

- Um centro de projetos de *IP cores* em Grenoble onde realizei o estudo aqui citado.

2.5.1 Centro de projetos em Grenoble

O centro de projetos de Grenoble foi criado com a aquisição da companhia SOISIC, uma empresa líder em IP físicos baseado na tecnologia *Silicon-on-Insulator* (SOI). O escritório é uma parte da Divisão de IP Físicos dedicada ao projeto de produtos SOI. O portfólio da ARM em SOI é composto de três partes: Células Padrões (*Standard Cells*) [10], Células de Entrada/Saída (*Input/output Cells*) e Memórias. Esse centro de projetos possui uma equipe dedicada a cada uma dessas partes [2].

A equipe de Células Padrões trabalha no desenvolvimento de todas as funções típicas e blocos básicos, os quais são, sobretudo, utilizados em Circuitos Integrados de Aplicação Específica (ASIC) e projetos de SoC.

As Células de Entrada/Saída são usadas para garantir uma melhor conexão entre o núcleo de silício do *chip* e os pinos de entrada e saída. As células principais, que são desenvolvidas em Grenoble, são conversores de tensão comumente chamados de *level shifters*, *buffer* e proteções contra descargas eletrostáticas (ESD).

A última equipe do escritório em Grenoble a ser aqui descrita é a equipe em que eu realizei a minha pesquisa. A equipe de memória é especializada no desenvolvimento de compiladores de memórias embarcadas para diferentes tipos como, por exemplo, ROM e SRAM [1]. Compiladores de memória são *softwares* em uma linguagem de alto nível como *Perl* ou *Java*, que gera um projeto de memória, dentre todas as possibilidades previamente definidas, em função dos parâmetros de entrada dados pelo usuário. Para diferentes tamanhos, dimensões ou combinação de opções suplementares à memória, o compilador gera todas as diferentes visões do modelo solicitado (esquemático, modelo de referência *Verilog*, leiaute, GDS-II...) necessárias para utilizar um *Software* de Projeto de Circuitos Integrados, também conhecido como *Electronic Design Automation* (EDA).

A sessão 3 explica o atual processo de desenvolvimento de compiladores de memória pela equipe de memória da divisão PIPD da ARM em Grenoble. Também será introduzido o problema enfrentado no estágio, assim como a solução proposta.

3. Introdução ao Trabalho Realizado

Na sessão 2 foi feita a descrição da empresa em que realizei o meu trabalho de estágio integrado. Começando por fatos históricos que levaram ao surgimento da ARM, foram mostrados detalhes desde como ela está dividida internamente até como ela está espalhada mundialmente. Na subseção 2.5.1, foi mostrado como o escritório da ARM em Grenoble está dividido e que este estudo foi realizado na equipe de memória.

Nesta sessão, será explanado brevemente o fluxo de projeto utilizado atualmente no desenvolvimento de compiladores de memória. Esse fluxo possui alguns problemas que serão aqui enumerados. Mais adiante, será proposto um novo fluxo de projeto para correção desses problemas.

3.1 Compiladores de Memória

Como foi mencionada anteriormente, a equipe de memória da ARM em Grenoble não projeta memórias separadamente. Quando uma especificação é fornecida para a equipe de projeto, ela desenvolve um compilador de memória. A licença para uso desse compilador é então vendida para a empresa que deseja fazer uso da memória especificada.

Compiladores de memória são *softwares* capazes de gerar as diversas representações de memória em função das especificações dadas pelo usuário. Para diferentes tamanhos de memórias ou dimensões, esse *software* pode fornecer todas as visões específicas do modelo da memória. Alguns exemplos desses tipos de visões são: esquema elétrico descrito em SPICE, modelos de referência em *Verilog* [6], leiaute, GDS-II [11], etc. Essas visões são necessárias no uso de um *Software* de Projeto de Circuitos Integrados durante o projeto de um SoC. Essas ferramentas de *software* também são comumente conhecidas por *Electronic Design Automation* (EDA) [12].

Os compiladores de memória também geram diferentes tipos de testes para cada uma dessas visões de cada memória gerada. Isso serve para que o cliente que está adquirindo o compilador de memória simule e valide a memória.

Consequentemente, não haverá dúvidas de que o circuito que foi gerado está de acordo com o especificado.

A Figura 5 ilustra um diagrama que simplifica o entendimento do *software* gerador de memórias. De forma simplificada, o compilador de memória recebe parâmetros de entrada de seu usuário e gera a memória desejada. Esses parâmetros podem ser o tamanho da palavra em *bits*, a quantidade de posições de memória, as opções adicionais da memória que serão explicadas na sessão 4.5, a quantidade de multiplexadores, dentre outras possibilidades.

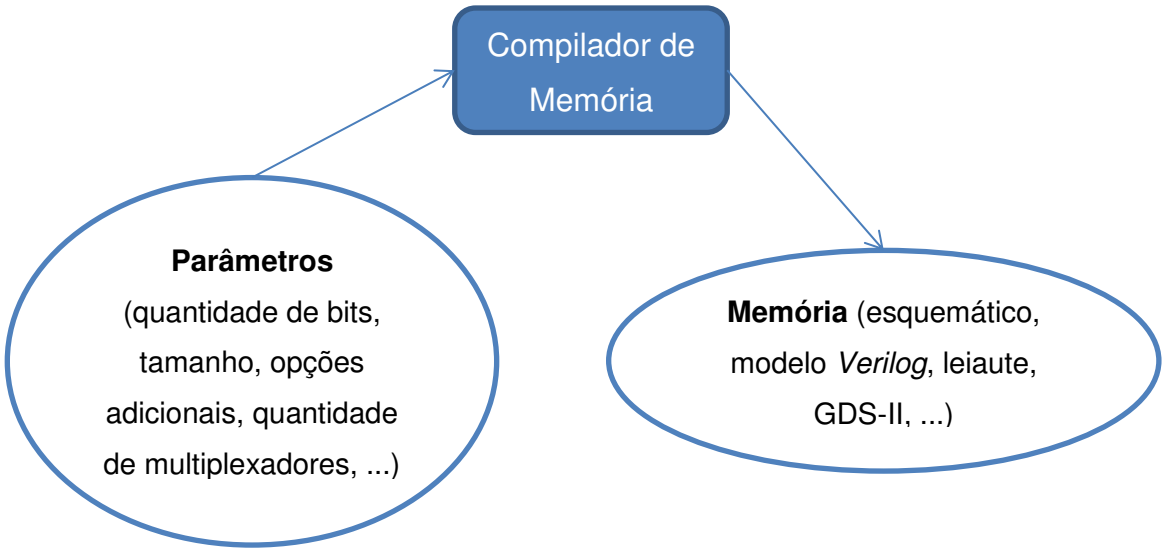

Figura 5: Diagrama simplificado da função dos compiladores de memória

3.2 Fluxo de Projeto

Como foi mencionada anteriormente, a principal tarefa da equipe de memória é projetar compiladores de memória. Para desenvolver tal produto, eles seguem um fluxo de projeto bem definido internamente na ARM. Por questões de confidencialidade, um diagrama simplificado desse fluxo é fornecido na Figura 6 apenas a parte que concerne aos testes da lógica do circuito foi mostrada.

Figura 6: Fluxo atual de projeto para o desenvolvimento de compiladores de memória – Elipses são códigos/arquivos e retângulos são ferramentas/software. A seta tracejada significa que o projetista deve criar o item e a seta cheia significa que o item é gerado.

Inicialmente, a primeira elipse representa o ponto de partida no fluxo. Quando especificação de certo tipo de memória que uma empresa tenha demandado é recebida, a equipe de memória faz o modelo dos esquemáticos de todas as memórias possíveis. Ela também desenvolve os esquemáticos das *leaf-cells* necessárias para gerar todas as memórias especificadas. *Leaf-cells* são divisões de uma memória completa que realizam tarefas definidas e serão mais detalhadas nas sessões seguintes.

Uma ferramenta desenvolvida pela ARM chamada internamente de *Tiler* usa os modelos esquemáticos das memórias e coloca os esquemáticos das *leaf-cells* necessárias para montar a memória desejada pelo usuário. Essa ferramenta gera o circuito completo da memória descrito em SPICE.

Outra equipe da ARM, em Bangalore na Índia, desenvolveu um gerador de modelos de referência em *Verilog*. Esses modelos são utilizados para comparar com o circuito gerado em SPICE. Com isso, a ferramenta ESP-CV é utilizada para verificar a lógica do circuito com o intuito de validar cada memória.

O ESP-CV [13] é uma ferramenta que foi desenvolvida pela empresa Synopsis [9]. Esse *software* faz a simulação de dois circuitos descritos em *Verilog*. Ele não é capaz de simular circuitos descritos em SPICE. Sendo assim, a ARM desenvolveu outra ferramenta capaz de converter modelos em SPICE para modelos em *Verilog* em um nível mais baixo que o modelo de referência. Esse modelo define em *Verilog* cada transistor MOSFET utilizado no circuito original.

Em posse dos dois circuitos em *Verilog* e dos vetores de teste a ferramenta pode simular e comparar os dois circuitos. Os vetores de teste são valores aplicados na entrada das memórias que estimulam igualmente ambos os circuitos. Com isso, cada saída é comparada até que todos os vetores de teste sejam testados. Se todas as saídas forem iguais, nenhum erro é relatado e a simulação termina com sucesso.

Todas as memórias possíveis com todas as combinações especificadas são testadas desta maneira. Ao terminar o teste de todas as combinações especificadas, é entendido que a lógica das memórias geradas por esse compilador de memória está funcionando corretamente.

3.3 Desvantagens

A forma como o modelo de referência em *Verilog* é descrito atualmente apresenta algumas desvantagens. Esse gerador de modelos é desenvolvido pela equipe da ARM em Bangalore.

Um dos principais problemas está ligado à cobertura de verificação funcional do circuito. Essa cobertura nunca é atingida por completo já que não é possível definir todas as possibilidades possíveis de estímulo para a memória nos vetores de teste. Atualmente, a cobertura é definida com os vetores de teste mais importantes e caso a simulação passe por isso, assume-se que a memória está com a lógica correta e que ela está livre de erros. Sabe-se que isso pode não ser sempre verdade, dado que algum erro humano pode acontecer na definição dos vetores de teste.

Normalmente, os modelos de referência devem ser passados para a equipe de memória totalmente livre de erros. Com isso, eles podem ser utilizados como modelos para os novos circuitos em desenvolvimento. Eventualmente, algum erro pode ser encontrado. Caso isso ocorra, é solicitado que a equipe de Bangalore corrija o erro enquanto a equipe de memória já está desenvolvendo o compilador de memória. Possivelmente isso vai causar atrasos no projeto que pode ser pequeno, caso o erro seja corrigido rapidamente, ou grande caso o tempo requisitado pela equipe de Bangalore para corrigir algum possível erro pode ser elevado. Esse problema ocorre por causa da maneira em que o código está descrito. Ele não apresenta nenhuma hierarquia em seu código e está descrito em um único arquivo.

Assim, o modelo passa a ser bastante complexo e de difícil entendimento, o que pode tornar difícil de encontrar erros.

Por motivos internos à empresa, foi solicitado que alguma alternativa ao ESP-CV fosse encontrada.

A sessão seguinte descreve a proposta de estudo feita pelo estágio aqui descrito para modificar o fluxo de projeto da empresa citado na subseção 3.2.

3.4 Concepção dos Modelos

Devido aos problemas citados na subseção 3.3, foi proposto um novo fluxo de projeto para o desenvolvimento de compiladores de memória. A Figura 7 ilustra o fluxo proposto. Para ser colocado em prática, ele precisa ser testado e validado, sendo esta a função deste estágio. Tudo que está marcado em azul nesta figura, destaca o que foi acrescentado ao fluxo de projeto.

Essa nova metodologia de descrição do modelo de referência em *Verilog* na ARM, baseia-se em modelar separadamente cada *leaf-cell* do modelo da memória que serão geradas a partir destas divisões. Essas divisões formam uma hierarquia na descrição do modelo de referência. Sendo assim, o novo modelo de referência foi batizado simplesmente como Modelo de Referência Hierárquico, enquanto o atual é chamado de Modelo de Referência Comportamental.

Essa divisão do modelo de referência possibilita uma simplificação na descrição da memória, o que pode diminuir a busca por eventuais erros. A separação do modelo em diferentes modelos menores para cada *leaf-cell* possibilita a simulação separadamente para cada uma delas, antes mesmo de testar a memória completa. Esta nova descrição possibilita melhorar a verificação funcional como será mostrada mais adiante. É possível até mesmo provar através da verificação formal para algumas dessas *leaf-cells* que o circuito está totalmente livre de erros. Como será mostrada mais adiante, esta solução também ajuda na busca por uma alternativa ao ESP-CV.

A sessão 4 detalhará o funcionamento básico dos elementos principais das memórias. Esta explanação vai ser focada no modelo SRAM, que foi a arquitetura adotada neste trabalho de estágio. Na sessão 5 vai ser mostrada a solução proposta

aos problemas anteriormente citados. Desse modo, será feita uma explicação de como foi desenvolvido e como funciona o Modelo de Referência Hierárquico, os resultados do estudo e as suas implicações no novo fluxo de projeto apresentado na Figura 7.

Figura 7: Novo fluxo de projeto para compiladores de memória proposto pelo o estudo descrito neste relatório de estágio

A parte em azul no fluxo de projeto proposto, ilustrado na Figura 7, foi acrescentada no fluxo atual como pode ser visto ao comparar com o da Figura 6. A validação dessa nova metodologia se baseia em testar todos estes itens. O gerador de modelos (inst_hier_verilog), que é representado pelo retângulo *cdl2ver.pl,* precisa ser testado juntamente das *leaf-cells* projetadas, representadas por outra elipse azul. Os retângulos Conformal, AMS e Modelsim representam possíveis EDA integrantes do novo fluxo.

Vale salientar ainda que apesar da linguagem de descrição dos modelos de referência utilizada na empresa ser o *Verilog*, este fluxo de projeto proposto independe da linguagem utilizada e poderia ser aplicado para qualquer outro tipo de linguagem de descrição de *hardware*, como por exemplo, System Verilog [15].

4. Introdução à Memória

4.1.1 Introdução a Memórias Embarcadas

Devido à forte tendência de miniaturização dos eletrônicos, os novos circuitos vêm utilizando cada vez mais memórias embarcadas. Memórias embarcadas são quaisquer memórias não autônomas que estejam integradas em *chips*.

Dentre as principais vantagens de utilizar memórias embarcadas, encontra-se a redução no número de *chips*, a possibilidade de utilizar memórias com várias portas, a menor exigência no espaço da placa de circuito impresso, a maior velocidade de resposta, a redução de consumo de energia e a maior rentabilidade devido à redução de custo. Dentre as desvantagens desta técnica, pode-se destacar a necessidade usual de *chips* com dimensões maiores e mais complexos o que pode dificultar em certos aspectos tanto o projeto como a fabricação.

As memórias SRAM embarcadas são amplamente utilizadas. De fato, o mercado de memórias SRAM embarcadas é mais extenso que o mercado de SRAM autônomas [1]. Dada a maior velocidade, a maior densidade e o menor consumo de energia, diversas empresas têm surgido para produzir dispositivos eletrônicos com SRAM embarcadas. Devido a este crescimento na demanda, consequentemente há um crescimento na oferta desse produto e com isso um aumento na quantidade de projetos de memórias embarcadas.

Memórias SRAM embarcadas são normalmente empregadas em CPU, DSP e MCU, os quais precisam de memória *cache* dentro do *chip* para uma comunicação e um processamento veloz. Células SRAM são também bastante utilizadas em ASIC ou SoC. Essas podem ser usadas através da programação delas ou como um bloco.

A sessão seguinte descreve o funcionamento das memórias semicondutoras genéricas, passando posteriormente a explanação do funcionamento das memórias SRAM genéricas.

4.2 Organização das Memórias Semicondutoras

Memórias semicondutoras são geralmente compostas de quatro elementos. O primeiro bloco, chamado de Matriz da Memória (*Memory Array*), ou núcleo (C*ore*), é

dedicado ao armazenamento propriamente dito da informação [6]. Ela consiste em uma matriz de células de memória, as quais retêm a informação binária de cada *bit* da memória. Portanto, essas células são chamadas de *bit-cell* [1].

Decodificadores são usados para acessar as células de memória posicionadas na intersecção entre uma linha selecionada, também conhecida por *word-line*, e uma coluna selecionada, conhecida por *bit-line* [2]. O diagrama de blocos das memórias semicondutoras genéricas é ilustrado na Figura 8. Os decodificadores contêm também outros componentes, os quais serão introduzidos posteriormente como, por exemplo, *buffers* e elementos de leitura e escrita.

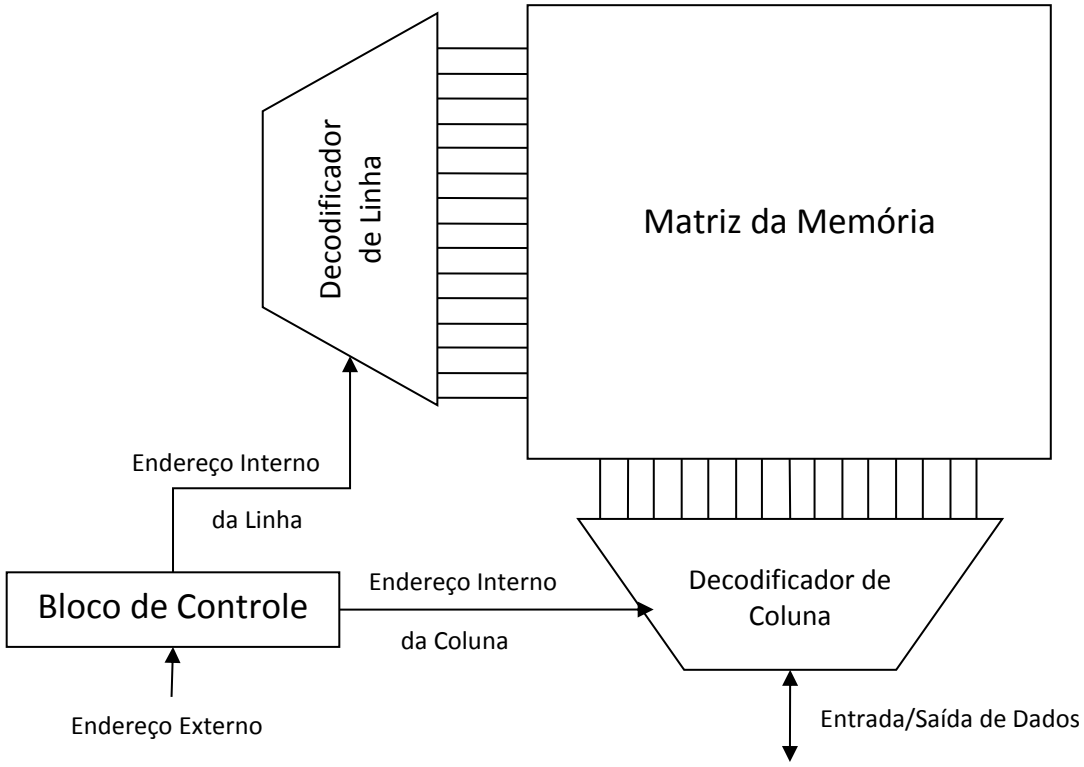

Figura 8: Diagrama de blocos de uma memória simples

Todo esse dispositivo é controlado por um bloco de controle que executa o ciclo de leitura e escrita. Internamente, esse bloco fornece para os decodificadores os endereços necessários para acessar as posições de memória desejadas.

A sessão seguinte é focada na explanação das Memórias Estáticas de Acesso Aleatório. Elas também são conhecidas como *Static Random Access Memory* (SRAM). Esse foi o tipo de memória utilizada durante o estudo realizado neste estágio.

4.3 Descrição das Memórias SRAM

A Memória Estática de Acesso Aleatório é um tipo de memória semicondutora volátil, visto que ela retém a informação somente enquanto o dispositivo está sendo alimentado. O termo *estático* indica que a célula não precisa de uma atualização periódica para manter o *bit* armazenado. *Acesso aleatório* significa que cada posição da memória pode ser lida ou escrita indiferente da posição anteriormente acessada.

4.3.1 Componentes da SRAM

Nesta sessão, todos os principais componentes da SRAM serão introduzidos, desde o ponto de armazenamento até o bloco de controle lógico. A Figura 9 representa o diagrama de blocos que indica as posições de cada componente dentro da memória.

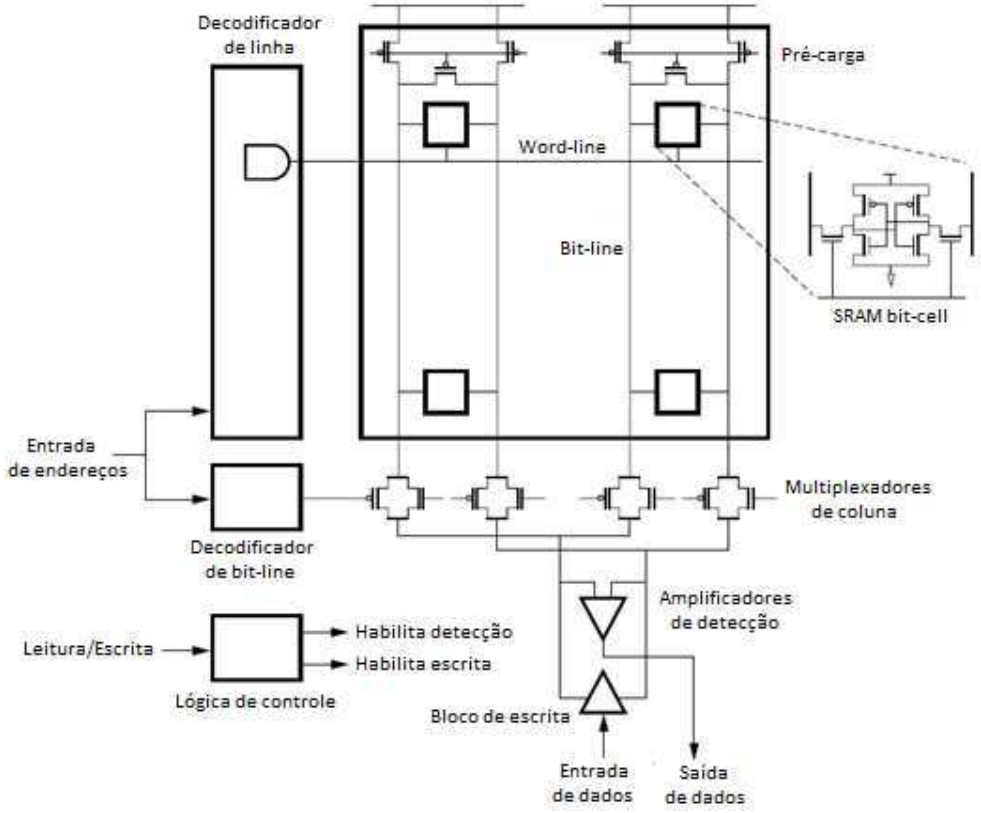

Figura 9: Diagrama da organização geral da SRAM

4.3.1.1 *Bit-cell* da SRAM

O componente principal das memórias SRAM recebe o nome de *bit-cell*. Este termo se refere a uma célula que possui a capacidade de armazenar uma unidade de informação digital ou como é popularmente conhecido, um *bit*.

A bit-cell baseia-se em seis transistores, como ilustrado na Figura 10a. Como pode ser visto na Figura 10b, quatro destes transistores (M_1, M_2, M_3 e M_4) são utilizados para fazer um par de inversores opostamente conectados. Esta estrutura forma o elemento de armazenamento biestável utilizado para memorizar um *bit* [2]. A informação é guardada nestas células básicas durante todo o período em que elas estão sendo alimentadas. Dois transistores NMOS adicionais (M_5 e M_6) habilitam o acesso à memória para leitura ou escrita através das *bit-lines* (BL e $\overline{\text{BL}}$). Eles são controlados pela *word-line* (WL) [1].

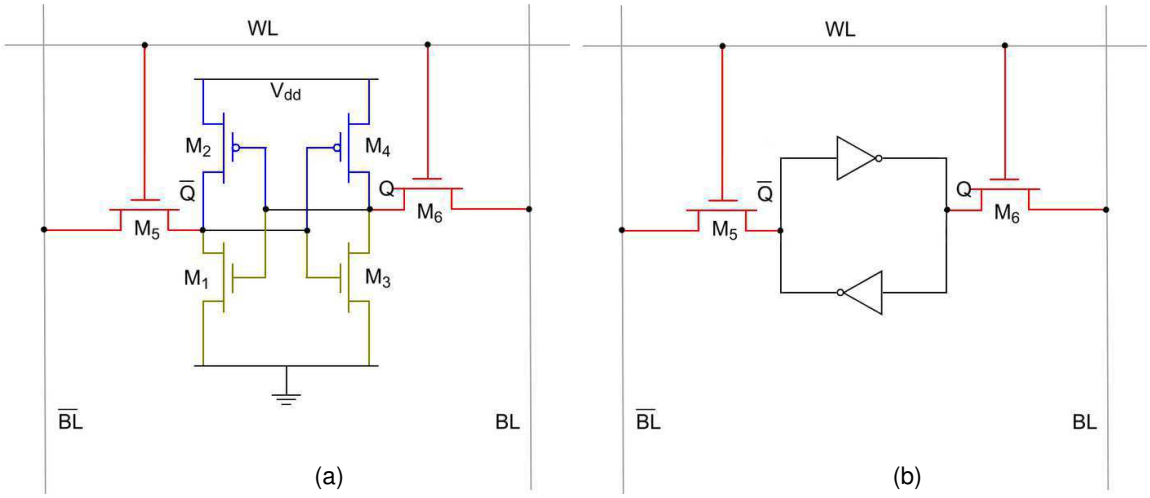

Figura 10: Esquema elétrico completo da *bit-cell* da SRAM representado por transistores (a) e por inversores e transistores (b)

A *word-line* é usada para selecionar a célula. Enquanto isso, as *bit-lines* são usadas para executar as operações de leitura e escrita na célula [2]. A *word-line* e as *bit-lines* podem ser facilmente entendidas quando analisamos a Matriz da Memória. A interseção de uma linha e uma coluna de uma matriz qualquer é dada por um elemento da mesma. De forma análoga, ao fazer a seleção da *bit-line* e da *word-line* é possível encontrar uma posição de memória na Matriz de Memória.

Por construção, a *bit-cell* tem a particularidade de guardar a informação e o seu complemento na mesma célula, respectivamente nos nós Q e $\overline{\text{Q}}$ do circuito representado na Figura 10. É importante ter uma dupla saída para cada um desses nós para fazer a comparação entre as tensões da *bit-line* e de sua complementar através de um amplificador diferencial. Isto permite aumentar a velocidade e reduzir o consumo de energia, o que será explicado na próxima sessão.

4.3.1.2 Amplificador de Detecção

O propósito do Amplificador de Detecção, também conhecido como *Sense Amplifier*, é de amplificar a pequena diferença de tensão entre as *bit-lines* durante a operação de leitura [2]. O resultado disso é uma oscilação em uma saída digital de uma ou duas extremidades. Existem diversos tipos de Amplificadores de Detecção que podem ser classificados em duas categorias: amplificadores modo de corrente e amplificadores do tipo *latch*. Ambos os tipos de amplificadores são ativados através do sinal SAE (*Sense Amplifier Enable*).

A Figura 11a ilustra o primeiro tipo que é, sobretudo, um amplificador diferencial com um par diferencial e um espelho de corrente com carga ativa [5]. Amplificadores do tipo *latch*, como o da Figura 11b, são baseados em um par de inversores opostamente conectados, também conhecido por *latch*.

Quando existe uma diferença de tensão suficiente entre as *bit-lines*, o amplificador apresenta uma tensão na saída. A diferença de tensão é copiada nos nós ligados aos *gates* dos transistores do Amplificador de Detecção. Com isso, os inversores colocam o nó inferior em nível baixo. Para esse último tipo de Amplificador de Detecção, a saída pode ser no nó A ou B. Isto depende se um *buffer* é necessário para acionar uma saída de alta capacitância.

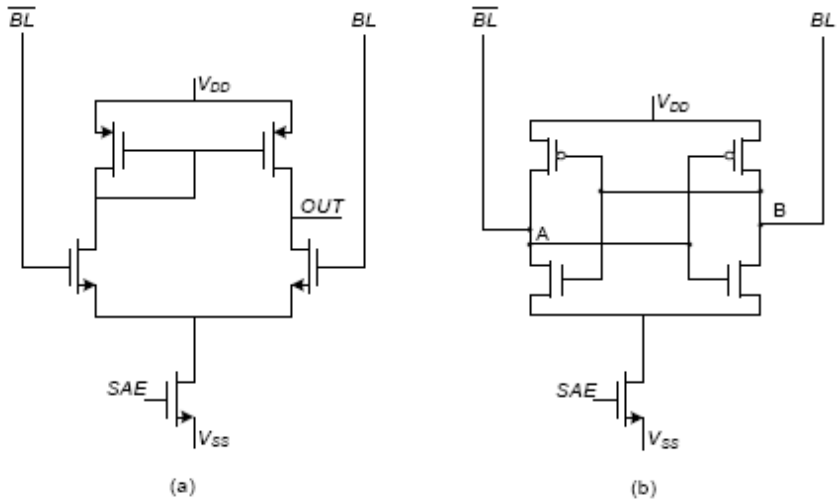

Figura 11: Amplificador linear (a) e do tipo *latch* (b)

4.3.1.3 Circuito de Pre-Carga

O Circuito de Pré-Carga é usado para conduzir as *bit-lines* (BL e $\overline{\mathrm{BL}}$) a uma mesma tensão. Como a operação de leitura baseia-se na comparação entre as

tensões das *bit-lines*, é necessário ter o mesmo estado inicial [2]. Para colocar as *bit-lines* nessa tensão de referência o Circuito de Pré-Carga é acionado.

A Figura 12 ilustra como pode ser o circuito de Pré-Carga. Tal circuito é composto de três transistores PMOS usados como chaves entre a alimentação e as *bit-lines*. O transistor central é um transistor de balanceamento que simplesmente equaliza os dois níveis de tensão nas *bit-lines*.

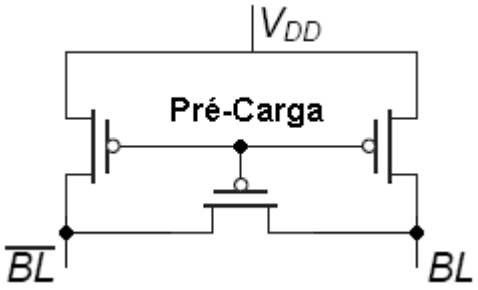

Figura 12: Esquemático básico do Circuito de Pré-Carga

4.3.1.4 Decodificadores de Linha e de Coluna

Decodificadores são utilizados em memórias com a finalidade de selecionar a *bit-line* e a *word-line* de uma posição de memória desejada. A precisão desses circuitos é essencial em memórias de acesso randômico porque elas têm uma influência no tempo de acesso e no consumo de energia.

O Decodificador de Linha faz a conversão entre um endereço *n-bit* e uma saída no formato 2^n. O nível alto corresponde à operação de leitura ativa para a *word-line* em questão [1]. O Decodificador de Coluna para a *bit-line* tem um princípio operacional semelhante e permite conduzir multiplexadores.

Assim como qualquer circuito em eletrônica digital, esse decodificador pode ser construído com portas NAND ou portas NOR [5]. Esse circuito é ilustrado na Figura 13. Para reduzir os atrasos e a ocupação de área, pode-se utilizar um bloco de decodificador com dois estágios.

Juntos, esses dois tipos de decodificadores, podem selecionar uma célula básica na memória onde se deseja escrever ou ler a informação nela contida.

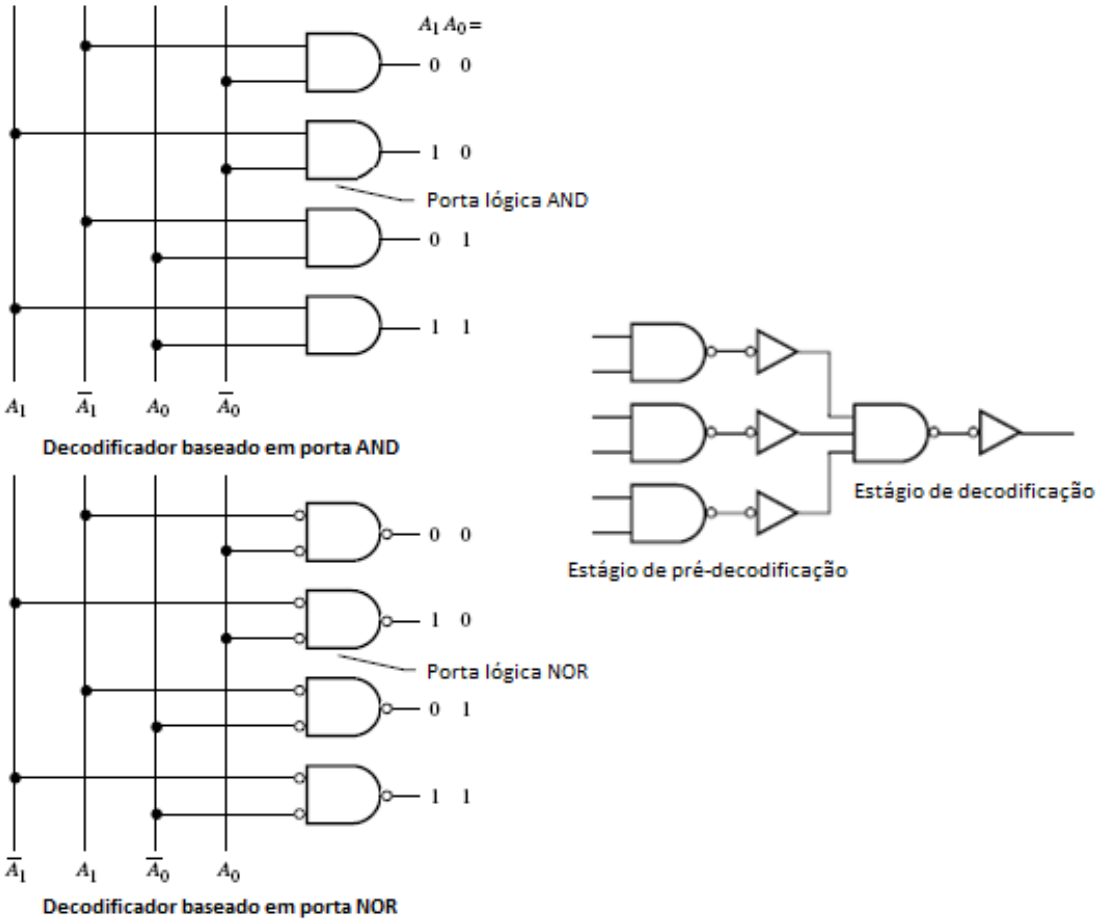

Figura 13: Decodificador NAND usando pré-decodificadores de duas entradas

4.3.1.5 Bloco de Escrita

O Bloco de Escrita permite a modificação do conteúdo de uma *bit-cell*. Para realizar isto, ele impõe um nível de tensão nas *bit-lines*. Geralmente um par de inversores é usado no intuito de controlar cada *bit-line* e sua complementar ao mesmo tempo [2].

A estrutura, cujo esquemático é ilustrado na Figura 14, é complementada por dois transistores NMOS destinados em ativar a função assim que for solicitado. Este artifício de ativação será explicado mais adiante, quando for mencionada a opção de Máscara de Escrita na sessão 4.5.1.

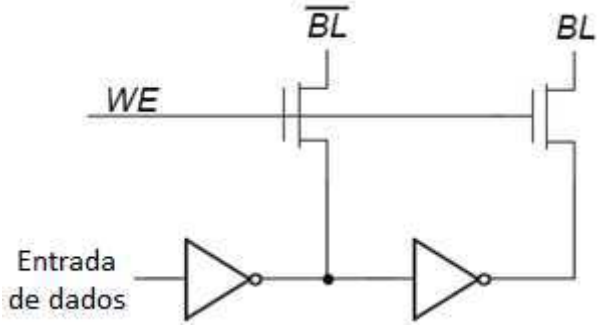

Figura 14: Esquemático do Bloco de Escrita

4.3.1.6 Bloco de Controle Lógico

O controle lógico gera os sinais de comando para o Circuito de Pré-Carga, decodificadores, multiplexadores, Amplificadores de Detecção e todos os elementos que são utilizados na memória [2].

A ativação de todos esses elementos necessita de uma sequência específica. De fato, as variações de tecnologias e processos de fabricação dos circuitos integrados implicam que cada dispositivo não possui as mesmas características elétricas. Esta especificidade evita um tempo de atraso fixo devido às capacitâncias parasitas, por exemplo. Por esta razão é essencial à integração de um gerador de atrasos que corresponda à tecnologia de fabricação e ao circuito utilizado [1].

Existem dois métodos básicos para realizar esse tipo de função: o da cadeia de atrasos e o projeto de uma réplica da *bit-line*. O primeiro consiste em usar uma cadeia de inversores que causam um atraso para especificar o tempo entre o gatilho da *word-line* e a ativação do Amplificador de Detecção. Essa aproximação implica em uma margem de erro porque o atraso ideal não poderia ser precisamente estimado antes da fabricação.

A segunda técnica é a melhor, pois utiliza um processo de estimação. Esse processo baseia-se em um modelo com dimensões idênticas à *bit-line* posto internamente na memória. Essa *bit-line* falsa possui as mesmas características das *bit-lines* originais do circuito e, por serem fabricadas no mesmo processo de fabricação, o modelo será uma cópia fiel e apresentará os mesmos atrasos que às originais. Essa característica nos permite descobrir exatamente como a *bit-line* está descarregando e, portanto o tempo necessário para ter a diferença de tensão que o Amplificador de Detecção precisa.

4.4 Arquiteturas da SRAM

Os Arquivos de Registradores, também conhecido como *Register File,* são uma matriz de *bit-cells* chamada de Matriz da Memória. Circuitos integrados modernos baseados em Arquivos de Registradores são usualmente projetados mediante memória RAM rápidas e estáticas com múltiplas portas. Tais SRAM são distinguidas por terem portas de leitura e escrita dedicadas, enquanto que a SRAM de porta ordinária lê e escreve através da mesma porta.

Esta sessão descreve as duas arquiteturas de memórias SRAM utilizadas em meu estágio integrado: Arquivo de Registradores de Porta Dupla (RF2) e Arquivo de Registradores de Porta Simples (RF1). Essas arquiteturas são também conhecidas internamente na empresa pelas suas siglas RF2 e RF1 respectivamente.

4.4.1 Arquivo de Registradores de Porta Dupla

Esse tipo de arquitetura tem duas portas, sendo cada uma delas dedicada à sua própria tarefa. A primeira porta é exclusiva para leitura, sendo nomeada simplesmente de porta A. A segunda porta é chamada de porta B e é dedicada à operação de escrita na memória.

Cada porta tem separadamente pinos de *clock*, endereçamento, dados, habilitação (*chip enable*) e um ajustamento de margem extra. Uma tabela detalhada com todas essas entradas e saídas dessa arquitetura pode ser encontrada no Apêndice A.

A Figura 15 ilustra um diagrama simplificado da organização do Arquivo de Registradores de Porta Dupla.

O bloco de controle também é dividido em duas partes, A e B, sendo um para leitura e o outro para escrita respectivamente. Eles podem trabalhar separadamente cada um com seu próprio *clock*, entradas e saídas de dados e entradas e saídas de sinais de controle.

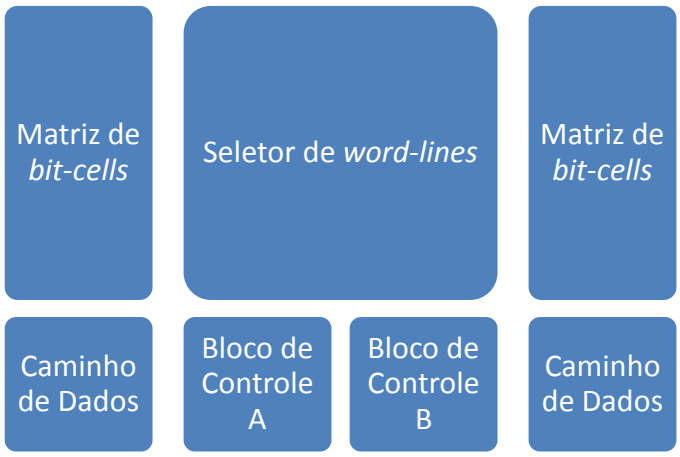

Figura 15: Diagrama simplificado da arquitetura RF2

A sessão 4.5 introduz algumas características adicionais que são opcionais e que podem ser acrescentadas nas memórias. Cada opção tem seus próprios sinais de entrada e saída. Para esse tipo de memória, é possível controlar separadamente as opções através de pinos em cada uma das portas.

A arquitetura usa uma *bit-cell* chamada *tpp251*. O esquemático dessa célula é ilustrado na Figura 16. Esse circuito possui uma diferença quando comparado com a *bit-cell* mostrada anteriormente na Figura 10. Essa diferença se encontra na operação de leitura que é controlada por *word-line* e *bit-lines* diferentes daquelas utilizadas na operação de escrita. Quando a *word-line* de leitura é ativada, o valor do nó Q é passado para BL que é a *bit-line* de leitura. A operação de escrita trabalha de forma similar à *bit-cell* apresentada anteriormente.

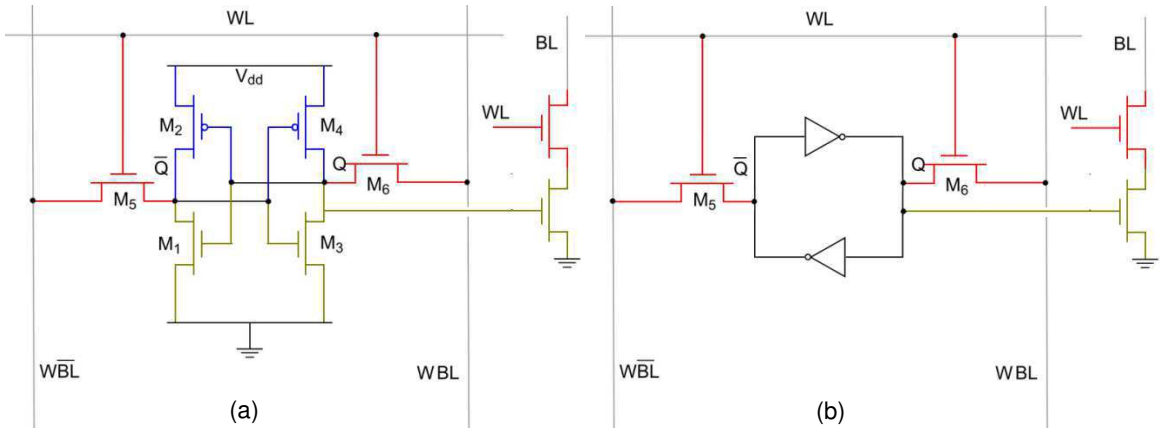

Figura 16: Esquema elétrico da bit-cell SRAM *tpp251* feito por transistores (a) e por inversores e transistores (b)

4.4.2 Arquivo de Registradores de Porta Simples

O Arquivo de Registradores de Porta Simples é o segundo tipo de arquitetura de memória SRAM usada durante este estágio integrado.

Essa arquitetura tem somente uma porta, a qual é utilizada para as operações de leitura e escrita de forma não simultânea. Diferentemente da memória de duas portas, a de porta simples possui apenas um sinal de *clock*, endereçamento, dados, habilitação (*chip enable*) e ajustamento de margem extra que controlam as operações de leitura e escrita.

A respeito das opções que podem ser adicionadas nessa memória, também é integrado apenas um sinal para cada um deles. Uma tabela detalhada com todas as entradas e saídas dessa arquitetura podem ser encontradas no Apêndice B.

A Figura 17 mostra um diagrama simplificado da organização da memória SRAM do tipo Arquivo de Registradores de Porta Simples. O bloco de controle aqui não é dividido em dois como no caso da arquitetura da sessão anterior. Neste caso, o bloco de controle utiliza apenas um *clock*, um exemplar de cada entrada e saída de dados e também um de cada entrada e saída de sinais de controle para as duas operações (leitura e escrita).

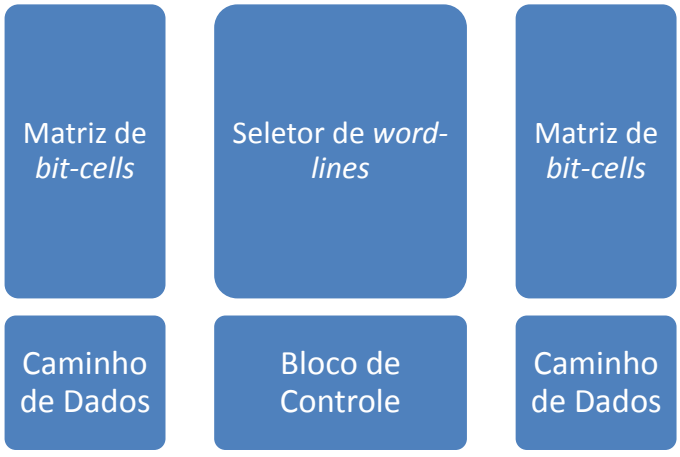

Figura 17: Diagrama simplificado da arquitetura RF1

A *bit-cell* usada nessa arquitetura tem o mesmo esquemático daquela mostrada na Figura 10 e é chamada internamente de *p152*.

4.5 Opções Adicionais das Memórias

Esta sessão vai explicar e descrever o conteúdo das opções padrões previstas nos compiladores de memória da ARM que foram tratados durante o trabalho realizado neste estágio integrado.

4.5.1 Máscara de Escrita

A opção Máscara de Escrita integra uma entrada chamada WEN (*Write Enable*) permitindo o usuário de controlar a operação de escrita na memória. Cada pino da entrada de dados D tem um pino de entrada WEN associado. Esse pino habilita a operação de escrita de cada *bit* separadamente [2]. Quando o comando de escrita é dado ao bloco de controle, todos os *bits* com o WEN ativado serão gravados na memória, enquanto que os restantes serão ignorados.

A Figura 14 ilustra o esquemático que representa a opção Máscara de Escrita. Quando o sinal WE está ativo, a informação em Data In é passada para BL e o seu complemento é copiado para $\overline{BL}$.

4.5.2 BIST Mux

Quando a opção BIST Mux está presente na memória, os caminhos de dados dentro da memória possuem Cadeias de Testes integradas e circuitos de *bypass* para os sinais de controle. O propósito dessa opção adicional é o de testar a propagação das saídas da memória através da Cadeia de Teste dentro da memória.

A função de teste é controlada pelos pinos DFTRAMBYP, TEN e SE [2]. Os pinos utilizados para essa função são aqueles em que as saídas têm terminações Y, assim como o barramento de saída SO usado como *bypass* da memória através da Cadeia de Teste e do barramento de entrada SI.

A Figura 18 ilustra o diagrama de blocos do circuito do BIST associada com suas respectivas entradas e saídas que controlam essa opção.

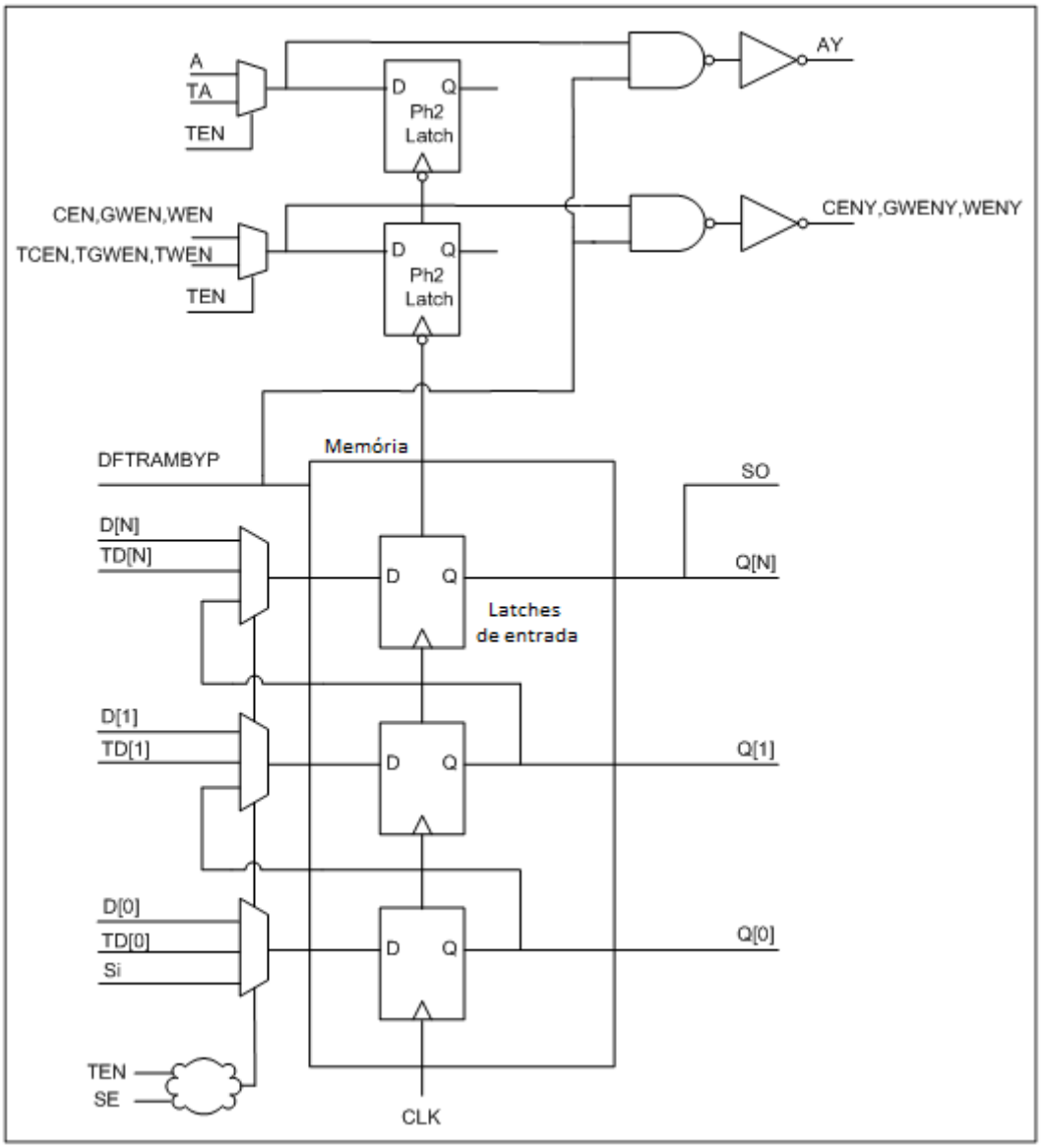

Figura 18: Diagrama de blocos da opção BIST Mux

4.5.3 Pipeline

A opção de Pipeline integra *flip-flops* no caminho de dados formando a Cadeia de Teste, também conhecida como *scan chains*. A opção BIST é requerida quando a opção *Pipeline* é selecionada, de modo a garantir que o circuito de *pipeline* possa ser testado.

A Figura 19 ilustra o diagrama de blocos da opção BIST associada com o *Pipeline*, juntamente das respectivas entradas e saídas que controlam ambas essas opções.

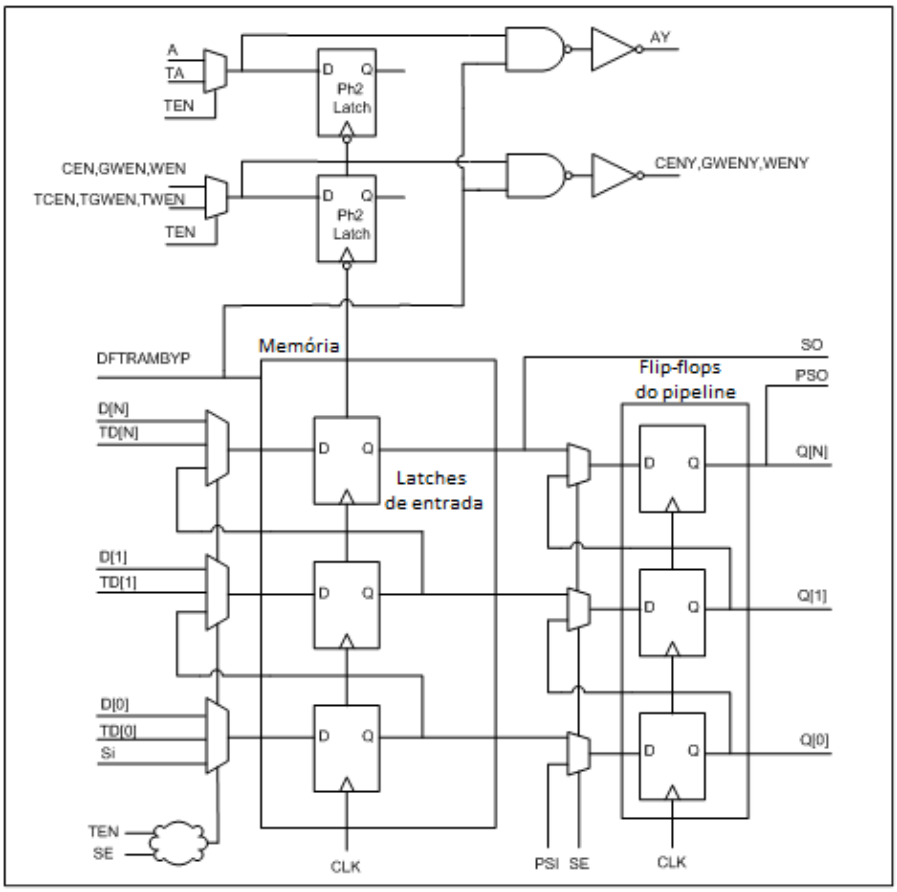

Figura 19: Diagrama de blocos das opções BIST e *pipeline* juntas

Os pinos de entrada e saída de teste para essas cadeias formam os barramentos PSI e PSO. Além disso, os pinos PENSI e PENSO formam a entrada e saída de teste para PEN [2].

A entrada DFTRAMBYP pode forçar o *clock* do registrador do *pipeline* a ser ativado. Caso o pino DFTRAMBYP seja ativo ou o valor no *latch* do ciclo anterior de PEN/TPEN seja nível baixo, os *flip-flops* do *pipeline* serão ativados. O estado de TEN será dependente da seleção feita pelo multiplexador através de PEN/TPEN. Caso contrário, Q mantém seu valor anterior porque o *flip-flop* não possui um sinal de *clock* no momento. PEN é colocado concorrentemente com CEN para determinar se uma operação de leitura particular estará disponível no próximo ciclo, ou se ela será paralisada. Um diagrama de blocos que representa o circuito de controle de *clock* através desse *flip-flop* da opção *Pipeline* é mostrado na Figura 20.

O pino de Modo de Missão é chamado de PEN e o pino de teste é o TPEN. A seleção desse multiplexador é controlada por TEN e SE. Um *flip-flop* ativo na borda positiva do *clock* é controlado pela saída desse multiplexador. A entrada de teste

desse *flip-flop* é PENSI e a saída do mesmo é o sinal PENSO. A função de teste desse *flip-flop* é controlada pelo sinal de controle SE.

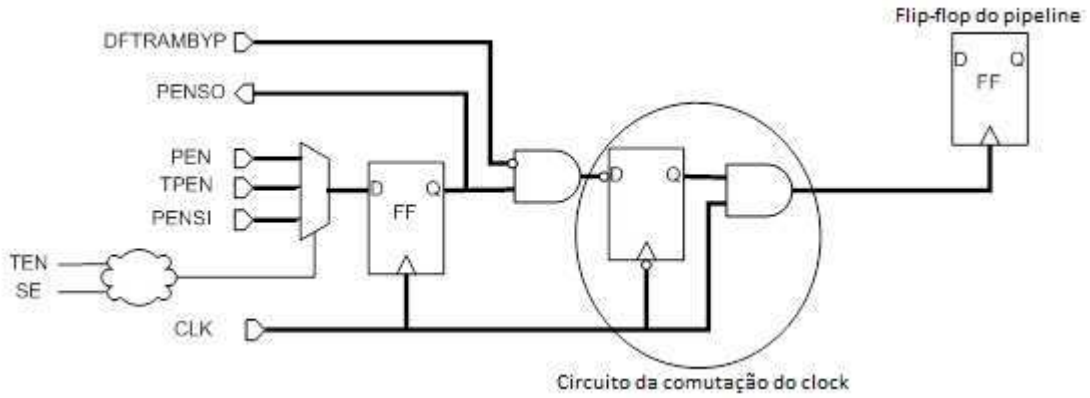

Figura 20: Diagrama de blocos do controle do *pipeline*

4.5.4 Comutação da Alimentação

Quando a opção de Comutação da Alimentação, também conhecida por *Power Gating* (PG), está presente, a memória contém um circuito para habilitar o modo de economia de energia.

A Figura 21 ilustra como as *leaf-cells* da opção Comutação de Alimentação são alocadas na memória quando ela está ativa.

Essa opção é controlada pelos pinos RET1N, RET2N e PGEN [2]. Usando esses pinos juntamente dos pinos de alimentação (VDDPE, VDDCE e VSSE), é possível controlar a economia de energia e alternar os modos de operação. Esses

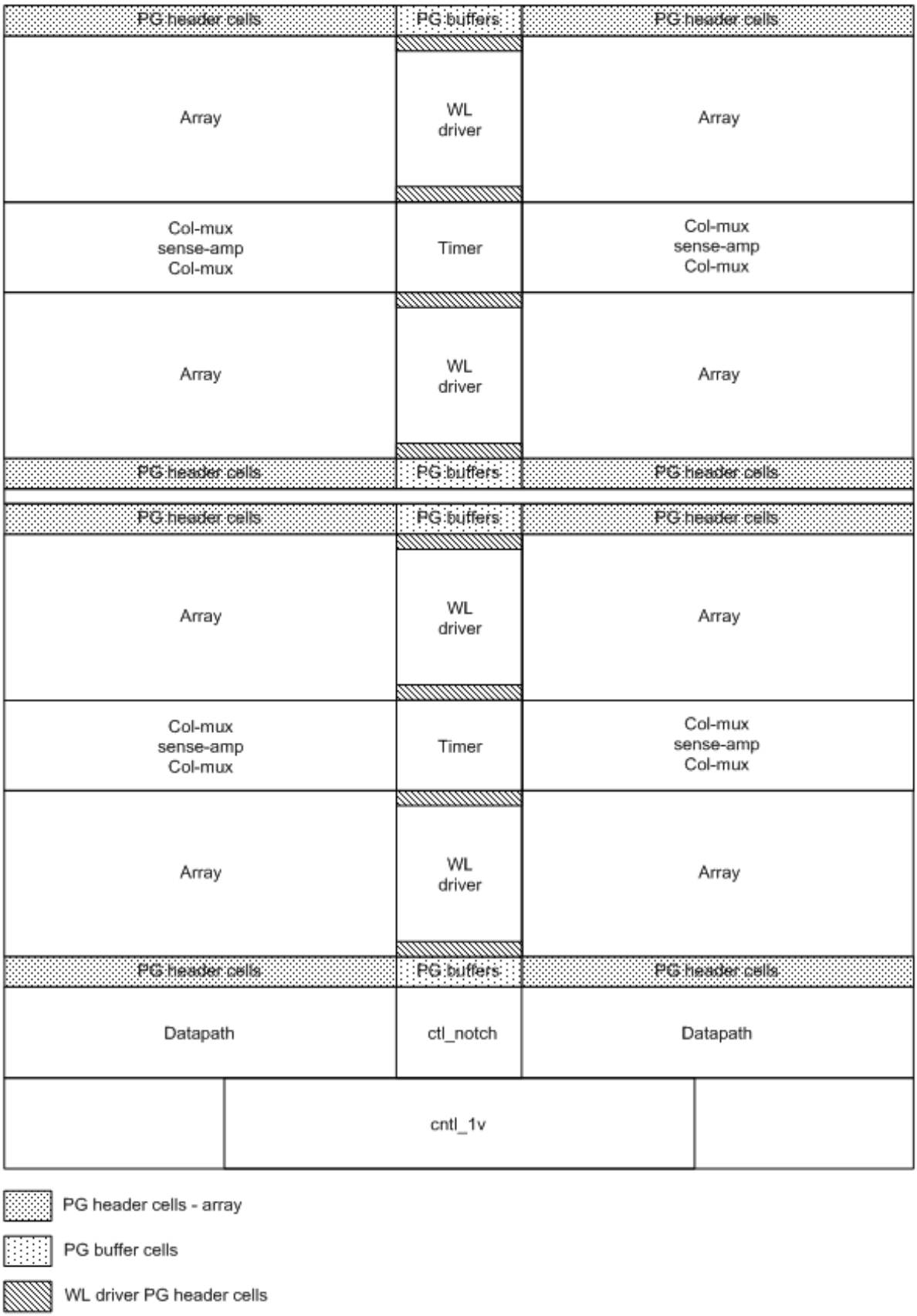

Figura 21: Organização das l*eaf-cells* da Comutação de Alimentação

modos de operação podem ser: Modo Normal, Modo de Pré-Carga Seletiva de Bit-Line, Modo de Retenção 1, Modo de Retenção 2 e Modo Desligado.

A memória por completo está alimentada durante todo o tempo quando ela está operando no Modo Normal e então ela funciona normalmente.

No Modo de Pré-Carga Seletiva de Bit-Line, as *word-lines* são grampeadas em nível baixo e as *bit-lines* flutuam. Este artifício reduz a fuga de corrente, embora as tensões de alimentação não possam ser alteradas.

No Modo de Retenção 1, as *word-lines* são grampeadas em nível baixo e as saídas são grampeadas em VDDPE. As *bit-lines* flutuam para reduzir a fuga de corrente e as tensões de alimentação podem ser alteradas pelo usuário da memória. Assim como, a energia periférica é desligada.

As diferenças do modo de operação anteriormente descrito para o Modo de Retenção 2 são as *bit-lines* que não estão flutuando. Assim como, as tensões de alimentação que devem estar sempre ligadas.

O Modo Desligado, como o próprio nome sugere, desliga toda a memória. Ao fazer isso, toda a informação armazenada é perdida e todas as saídas são grampeadas em VDDPE. As tensões de alimentação VDDPE e VDDCE devem estar sempre ligadas, mesmo quando entram no Modo Desligado.

4.5.5 Redundância

A opção Redundância integra uma coluna adicional entre cada lado da coluna principal da memória e as matrizes direita e esquerda da memória. Essas colunas adicionais serão usadas no caso de ocorrer algum eventual problema em alguma das colunas.

Quando há uma coluna defeituosa na memória, ela pode ser identificada depois de um teste por um elemento externo que fornece o endereço da coluna defeituosa. Após essa identificação ter sido realizada e estando essa opção presente na memória, os blocos responsáveis da Redundância irão deslocar todas as entradas e saídas de dados uma coluna para a direita ou para a esquerda, caso o defeito seja respectivamente no lado esquerdo ou direito da memória.

Essa opção substitui a coluna defeituosa encontrada na matriz da memória somente quando uma coluna é defeituosa em cada lado caso tenha uma coluna de redundância. É possível adicionar mais do que uma coluna extra a essa opção e assim corrigir mais do que uma coluna afetada.

A Figura 22 ilustra como as *leaf-cells* da Redundância são alocadas na memória quando essa opção está presente.

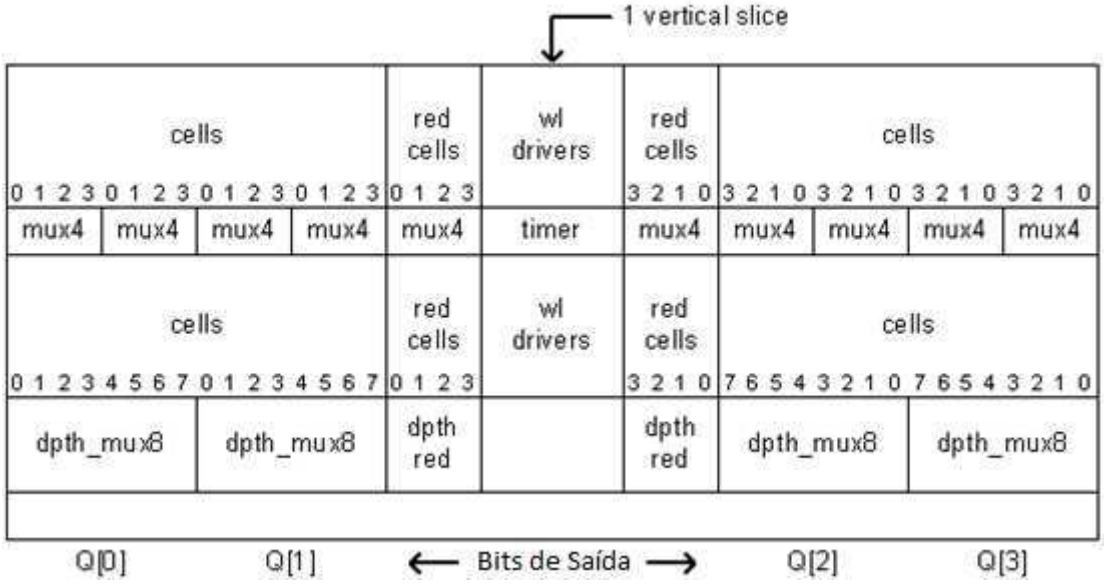

Figura 22: Organização da memória com a Redundância presente

Essa opção também integra um circuito de controle pelos pinos CRE1 e CRE2 que permitem o usuário da memória ativar essa opção. Os barramentos FCA1 e FCA2 são utilizados para configurar os endereços das colunas defeituosas respectivamente do lado esquerdo e direito.

4.5.6 Multiplexadores

Caso estejam presentes na memória, um conjunto de multiplexadores pode ser usado para selecionar a coluna das *bit-cells* acessadas. Isso se aplica aos blocos das *bit-cells* e divide a memória em diferentes blocos. Esses blocos são selecionados pelo multiplexador para serem acessados e o endereço distingui a linha e a coluna dentro desses blocos. Dentre os tipos de arquiteturas utilizadas neste estágio, existiram memórias com multiplexadores de 1, 2, 4 ou 8 entradas.

Em termos de dimensões da memória, normalmente quando se deseja aumentar o tamanho da memória, basta aumentar a quantidade de posições dela.

Fisicamente falando, para a arquitetura em estudo, isso aumenta a memória em sua altura, já que quanto mais posições, mais *bit-cells* serão postas umas sobre as outras e quanto maior for o tamanho da palavra, mais *bit-cells* serão postas uma do lado das outras e maior será seu comprimento. Essa opção possibilita aumentar o armazenamento de dados no comprimento ao invés da altura [2]. Quanto maior a quantidade de multiplexadores, mais capacidade de armazenamento a memória terá, assim como um maior comprimento, sem aumentar o tamanho da palavra.

5. Novo Modelo *Verilog*

Este capítulo é dedicado à explicação do trabalho realizado durante este estágio integrado. A sessão 5.1 descreve brevemente o modelo de referência atual (Modelo de Referência Comportamental) usado nas simulações *Verilog*. A sessão 5.2 examina o desenvolvimento do novo modelo de referência em *Verilog* (Modelo de Referência Hierárquico). A sessão 5.3 descreve a maneira atual de realizar a verificação funcional da lógica dos projetos de memória. A sessão 5.4 explica a introdução da do método de verificação formal na metodologia atual do projeto de memórias. A sessão 5.5 comenta os resultados obtidos das simulações *Verilog* usando o novo modelo de referência. A sessão 5.6 descreve as dificuldades encontradas durante o trabalho de estágio e alguns detalhes que podem ser melhorados e estudados com mais detalhes em trabalhos futuros.

5.1 Modelo de Referência Comportamental

Existem diferentes aspectos do compilador de memória que precisam ser validados para que se possa ter a certeza de que as memórias geradas por ele estão de acordo com a especificação do cliente. Esta sessão focará em um tipo de verificação que foi coberta neste trabalho de estudo: a simulação *Verilog*. Essa simulação permite a comparação do modelo de referência escrito em linguagem de descrição de hardware, neste caso *Verilog*, com o circuito elétrico gerado. Este tipo de verificação é usado para testar se a lógica do circuito projetado está conforme a especificação.

Todo compilador de memória pode gerar diferentes tipos de visões das memórias que ele é capaz de gerar e uma dessas visões é o Modelo de Referência Comportamental em *Verilog*. Com o modelo que descreve todas as funcionalidades da memória, é possível realizar simulações comparando esse modelo com a *netlist* SPICE [7] que também é uma das visões geradas pelo compilador de memória.

A *netlist* SPICE é uma forma de descrever um circuito elétrico utilizando uma linguagem escrita. Esta linguagem representa cada componente eletrônico em um esquemático e a forma como eles estão interconectados. A sessão 5.3 explica como essas simulações são feitas.

Os modelos *Verilog* atuais são gerados sem qualquer divisão estrutural, o que pode ser bastante complexo de entender e modificar quando for necessário. Consequentemente, se o modelo é complexo então o desenvolvimento do gerador também pode se tornar complexo. Isso pode custar muito tempo para ser finalizado. Igualmente, também pode ser mais fácil de cometerem-se erros escrevendo dessa maneira. Quando uma nova arquitetura é desenvolvida ou quando alguma modificação precisa ser feita para corrigir algum possível erro, o gerador de código precisa ser atualizado, o que pode levar muito tempo para entender o código, encontrar o erro e corrigi-lo.

5.2 Modelo de Referência Hierárquico

A solução encontrada para resolver os problemas citados na sessão anterior foi dividir o modelo de referência em diferentes partes, ou módulos, chamados aqui de *leaf-cells*. Desta forma, foi possível facilitar o entendimento e as modificações, aumentando potencialmente a cobertura funcional dos circuitos. Como será mostrada, outra vantagem é o tempo de desenvolvimento do modelo que pode ser reduzido. Esse tipo de modelo foi nomeado de Modelo Hierárquico em *Verilog*.

5.2.1 Leaf-cells

Todo compilador de memória tem um acervo com todas as *netlists* SPICE para uma arquitetura específica. Essas *netlists* podem ser combinadas por um software denominado de *tiller*. Essa ferramenta desenvolvida pela ARM tem o intuito de gerar memórias dependendo dos parâmetros fornecidos pelo usuário.

Antes de tudo, divide-se o modelo *Verilog* da mesma maneira que o esquemático está dividido, em *leaf-cells*. Essa divisão facilita o desenvolvimento porque nos possibilita dividir e testar separadamente cada *leaf-cell* e o novo modelo *Verilog* com a *netlist* SPICE. Consequentemente, toda a memória é testada. Esta prática faz com que a depuração do código seja dividida em partes menores, antes de chegar à depuração da memória completa.

Deste modo, caso seja necessário fazer alguma alteração é muito mais fácil modificar somente alguma *leaf-cell* específica ao invés de todo o modelo de referência. Tais alterações podem ser requisitadas ao encontrar-se um erro ou ao

desenvolver-se uma nova arquitetura. Contudo, no desenvolvimento de uma nova arquitetura ou no aperfeiçoamento de uma arquitetura atual, apenas algumas *leaf-cells* são alteradas. O tempo requerido para modificar apenas uma *leaf-cell* certamente é muito menor do que o tempo necessário para modificar uma memória inteira.

Ao desenvolver um gerador de Modelos Hierárquicos em *Verilog*, é indispensável primeiramente provar o conceito para um simples modelo. Portanto, a memória mais simples foi escolhida e sem qualquer opção adicional da arquitetura Arquivo de Registradores de Porta Dupla. Com isso, iniciou-se a escrita do código em *Verilog* dos modelos das *leaf-cells* referentes a essa memória.

Após a escrita e teste de um modelo simples, desenvolveu-se os modelos em *Verilog* das outras *leaf-cells* necessárias para fazer o gerador de qualquer memória dessa arquitetura. Devido ao bom andamento do trabalho comparado com o cronograma inicial, todo este processo foi repetido posteriormente para a arquitetura Arquivo de Registradores de Porta Simples com a finalidade de obter uma melhor argumentação para a metodologia ao final do período de estágio. Foram desenvolvidos 34 modelos para *leaf-cells* da primeira arquitetura e 43 para a segunda. A sessão 5.3 explica como essas *leaf-cells* foram testadas.

O Apêndice C e o Apêndice D ilustram todas as *leaf-cells* desenvolvidas para ambas as arquiteturas: Arquivo de Registradores de Porta Dupla e Arquivo de Registradores de Porta Simples. Estas tabelas ilustram também um tempo de desenvolvimento estimado associado a cada uma das *leaf-cell*s.

Para a primeira arquitetura, as *leaf-cells* mais complicadas de terem os seus modelos *Verilog* desenvolvidos foram o bloco de controle (*cntla*), o bloco do caminho de dados (*dpth_rw_m2*) e as *leaf-cells* da opção adicional de Comutação de Alimentação (*pg_cntl*, *pg_m2* and *pg_edge*). As *leaf-cells* com maior complexidade no desenvolvimento de seus modelos *Verilog* para a segunda arquitetura foram o bloco de controle (*ck*) e o bloco de caminho de dados (*colm2x2*). A sessão 5.3 explica porque esses modelos foram difíceis de serem escritos.

Um exemplo do esquemático de uma *leaf-cell* é dado no Apêndice E. O respectivo modelo de referência escrito em *Verilog* é dado no Apêndice F.

5.2.2 Células Primitivas

Apesar das memórias serem divididas em *leaf-cells*, uma divisão menor ainda pôde ser feita em blocos chamados de Células Primitivas. Essas Células Primitivas são específicas da tecnologia em questão e podem ser usadas em diferentes tipos de arquiteturas. Sendo assim, é possível colocar em prática a ideia de reutilização de código.

A tecnologia na qual o meu trabalho de estágio foi baseada é chamada de *CMOS28HPP*, o que significa que é usado um processo de tecnologia CMOS em 28nm. HPP significa Processo de Alto Desempenho (*High Performance Process*). Ambas as arquiteturas compartilhavam da mesma tecnologia. Portanto, foi possível reaproveitar algumas Células Primitivas na confecção das *leaf-cells* da segunda arquitetura.

Utilizando esse tipo de divisão, o tempo de desenvolvimento das *leaf-cells* pode ser ainda menor quando comparado com a redução comentada na sessão anterior. O reuso de código também aumenta tão quanto haja mais Células Primitivas iguais ou similares em diferentes arquiteturas e compiladores. Uma vez que todas as Células Primitivas estão feitas e há uma biblioteca com todas elas, é necessário apenas conectá-las para fazer as *leaf-cells*, já que elas compartilham diversas Células Primitivas idênticas. Posteriormente, é possível fazer o mesmo com o modelo completo das memórias.

Antes de escrever as *leaf-cells*, foram desenvolvidas as Células Primitivas necessárias para fazer um gerador para a arquitetura Arquivo de Registradores de Porta Dupla. Para colocar em prática o conceito de reuso das células primitivas entre diferentes arquiteturas de um mesmo processo de tecnologia, foram desenvolvidas as Células Primitivas para fazer um gerador para a arquitetura Arquivo de Registradores de Porta Simples também. Algumas Células Primitivas já tinham sido previamente desenvolvidas para a primeira arquitetura e assim não foi necessário refazê-las. Foram desenvolvidos 26 modelos de Células Primitivas da primeira arquitetura e 58 da segunda. As sessões 5.3 e 5.4 explicam como essas células primitivas foram testadas. O Apêndice G e o Apêndice H mostram tabelas com os resultados do desenvolvimento das Células Primitivas para ambas as arquiteturas.

Estas tabelas também ilustram um tempo estimado de desenvolvimento associado a cada Célula Primitiva.

Para a primeira arquitetura, as células mais complicadas de terem os seus modelos *Verilog* desenvolvidos foram o *mem_gtp_latch* e o *dpth_bit*. As células com maior complexidade no desenvolvimento de seus modelos *Verilog* para a segunda arquitetura foram o *mem_senseamp* e o *mem_cmux1*. Todas essas células citadas como complicadas são blocos analógicos. Portanto, as células mais complicadas de serem descritas com um modelo *Verilog* são sempre as analógicas, já que são circuitos analógicos sendo descritos por modelos digitais. Como essas Células Primitivas estavam instanciadas naquelas *leaf-cells* citadas na subseção anterior como as mais difíceis de serem descritas, isso explica o porquê delas serem também difíceis de descrever.

Um exemplo do esquemático de uma Célula Primitiva é dado no Apêndice I. O respectivo modelo de referência escrito em *Verilog* é dado no Apêndice J. Essa célula foi instanciada na *leaf-cell* ilustrada no Apêndice E para representar um caso no qual se aplica a reutilização de código novamente.

5.2.3 Nível de Instância

Depois de todas as *leaf-cells* de uma arquitetura específica serem descritas, basta conectar elas para criar o modelo de referência da memória proposta, conhecido como o nível de instância. Esse modelo também é conhecido como *instance level*.

Por se tratar de uma manipulação de texto, a maneira mais simples de fazer isso foi escrever um script em *Perl*. Esta linguagem, já utilizada na empresa, possui ótimos manipuladores de arquivos, facilitando esta geração. Esse *script* utiliza a *netlist* SPICE gerada e modifica a sintaxe SPICE de instanciação para a sintaxe de *Verilog*, além de ignorar todos os transistores instanciados dentro da *netlist* e apagá-los. O resultado é um arquivo de instanciação com a mesma hierarquia da *netlist*, mas que irá instanciar os modelos em *Verilog* das *leaf-cells* que foram aqui desenvolvidos. Esse arquivo não contém qualquer lógica dentro dele, apenas o cabeçalho de cada módulo com seu corpo vazio, com exceção da instancia de cada *leaf-cell*.

Esse *script* representa o gerador de Modelos Hierárquicos em *Verilog* que cria automaticamente qualquer modelo da arquitetura específica dependendo dos parâmetros passados pelo usuário.

5.3 Verificação Funcional

O termo *Verificação Funcional* tenta responder à seguinte pergunta: “Este projeto faz o que ele deve fazer?”. Em automação de projetos de eletrônica, isso significa verificar se a lógica do projeto está conforme a especificação. Essa é uma tarefa muito complexa e normalmente leva a maior parte do tempo do projeto. Algumas bibliografias afirmam que esse tempo chega à maioria das vezes a 70% do tempo total do projeto.

A representação da especificação é o modelo já existente e gerado pelos compiladores de memória atuais: o Modelo de Referência Comportamental em *Verilog*. Atualmente, as simulações são feitas apenas no Nível de Instância usando o *ESP-CV* [1] que é uma ferramenta da *Synopsys* utilizada na simulação de códigos *Verilog* e *Verilog switch-level*. Esse último é uma maneira de escrever projetos usando algumas estruturas primitivas do *Verilog* que descrevem um circuito em um nível baixo cada transistor por transistor. Uma ferramenta embarcada é usada para converter a *netlist* SPICE em *Verilog switch-level*. Com isso, é possível simular códigos em V*erilog* com *netlists* em SPICE.

Para realizar a simulação com esse *software*, os vetores das entradas precisam ser descritos. Após cada vetor aplicado simultaneamente em ambos, obtêm-se as saídas correspondentes para compará-las. Se todas as saídas para todos os vetores aplicados forem iguais, isso significa que a simulação foi completada com sucesso. Caso contrário, uma mensagem de erro aparece e uma depuração é realizada para concertar o projeto e simular novamente até que a cobertura funcional chegue a 100%.

Juntamente aos vetores, é necessário escrever um *script* para cada simulação descrevendo as entradas e as saídas a serem analisadas. O *testbench* fornece as entradas para o projeto e para o modelo de referência. Após isso, o *testbench* recebe as saídas de ambos para analisá-las. Esses *scripts* foram escritos

[1] *ESP (R) Version A-2007.12 -- Nov 20, 2007*

para os casos de testar *leaf-cells* ou Células Primitivas separadamente. Na simulação do Nível de Instância, isso já é automaticamente gerado pelo compilador de memória.

A Figura 23 ilustra um diagrama que resume o fluxo do processo de verificação funcional com o ESP-CV na ARM.

Figura 23: Fluxo do processo de verificação funcional com o ESP-CV

5.3.1 Restrições

Existem algumas desvantagens em simular um projeto usando a ferramenta ESP-CV. Comumente, pode ocorrer de não ser possível ou de ser bastante complexo de testar todos os casos de um circuito sequencial lógico. Isso se deve ao fato de que o número de possibilidades é muito grande, o que exigiria bastante tempo de CPU para simulá-lo. Assim como, por causa do número de combinações de entradas serem muito grandes, torna-se impraticável de escrever todos os vetores de teste necessários. Com isso, nos temos que escolher um conjunto de vetores importantes para testar. Contudo, a cobertura estará sempre incompleta já que não foram testados todos os casos possíveis.

Adicionalmente aos vetores, é necessário escrever um *script* para cada *leaf-cell* e Célula Primitiva como foi citado anteriormente. Cada um deles descreve todas as entradas e saídas detalhadamente para auxiliar a passagem de vetores de teste ao modelo de referência e ao projeto da memória. Isso pode tomar um tempo suplementar para verificar as *leaf-cells* e as Células Primitivas. Consequentemente, essa prática pode levar um tempo considerável para finalizar a escrita dos modelos.

No caso de usar o ESP-CV para simular o Nível de Instância, isso não é um problema, visto que esses *scripts* já são gerados pelo compilador de memória. De qualquer maneira, a cobertura ainda continua incompleta.

É importante realçar outras duas desvantagens sobre a escrita dos vetores para as *leaf-cells* e as Células Primitivas. A primeira é que quanto mais se escreve vetores para aumentar a cobertura, o tempo de CPU requisitado para finalizar a simulação também irá aumentar. A segunda desvantagem está ligada ao fato de que escrever vetores é uma tarefa que consome tempo e é uma tarefa complexa. Sendo assim, o projetista precisa correlacionar: o tempo de CPU para simulação e a cobertura da verificação funcional. A situação ideal seria obter uma boa cobertura funcional usando o mínimo de tempo de CPU possível. Essa é a situação almejada neste estágio.

5.4 Verificação Formal

Verificação Formal é o ato de provar o quão correto o projeto está usando métodos formais da matemática para comparar com o modelo de referência. Esse tipo de verificação não aplica nenhum tipo de estímulo nas entradas; apenas compara os circuitos e determina se eles são equivalentes ou não equivalentes. O *software* utilizado neste estágio para fazer esse tipo de simulação é chamado Conformal[2] [14] que é uma ferramenta da *Cadence*.

A ausência de vetores é uma grande vantagem da verificação formal quando comparada com a verificação funcional discutida na sessão 5.3. Outra vantagem substancial é que o tempo de CPU requisitado para verificar o projeto diminui consideravelmente. Pode-se analisar essa diferença quando comparamos o tempo de CPU requerido para simular as Células Primitivas usando o Conformal e das Células Primitivas e *leaf-cells* usando o ESP-CV nas tabelas do Apêndice C, Apêndice D, Apêndice G e Apêndice H. O tempo de simulação com o ESP-CV pode ser aumentado tão quanto a quantidade de vetores seja aumentada, e consequentemente a cobertura. Isso não acontece com o Conformal. A verificação custará aproximadamente o mesmo tempo de CPU, dado que a grande maioria tem uma complexidade de circuito parecida. Entretanto, a vantagem principal é a

[2] *CONFORMAL (R) Version 09.10-s440 (30-Jun-2010) (64 bit executable)*

cobertura que será sempre completa, caso o circuito seja considerado equivalente ao modelo de referência. Caso a simulação termine com sucesso, pode-se ter total certeza de que o projeto é realmente equivalente ao modelo de referência.

Essa metodologia também implica em um número reduzido de *scripts* escritos e utilizados para simular, já que não é necessário descrever todas as entradas e saídas como era feito anteriormente no ESP-CV. Obviamente, não há necessidade também de descrever vetores de teste, o que economiza bastante tempo quando comparado com a verificação funcional. É necessário apenas um *script* bastante simples para realizar a verificação formal e que pode ser reutilizado em todas as células.

Devido a essas vantagens, foi decidido incluir o Conformal no novo fluxo de projeto para compiladores de memória desenvolvido e proposto por este trabalho de estágio.

A Figura 24 ilustra um diagrama que resume o fluxo do processo de verificação funcional com o ESP-CV na ARM.

Figura 24: Fluxo do processo de verificação formal com o Conformal

5.4.1 Restrições

Normalmente, as Células Primitivas analógicas usam as declarações *force* e *release* do *Verilog* para descrever o seu modelo de referência. O Conformal não aceita esse tipo de declaração ao menos que o projeto seja uma caixa preta. Infelizmente, não foi possível desenvolver um modelo em *Verilog* que pudesse ser tratado como uma caixa preta, para poder ser verificado formalmente pelo

Conformal. Por consequência, todas as Células Primitivas analógicas foram testadas usando o ESP-CV. Assim como as *leaf-cells* que possuíam alguma dessas células instanciadas dentro delas. É importante realçar que existem sempre blocos analógicos dentro das memórias e consequentemente o Conformal não é uma alternativa para o ESP-CV no caso do Nível de Instância, apenas para Células Primitivas digitais.

Existem alguns blocos com entradas e suas respectivas entradas invertidas, como por exemplo, o *clock* (*clk*) e o *clock* invertido (*nclk*). Quando isso acontece, é necessário criar um bloco empacotador, também conhecido de *wrapper block*, que instancia o bloco a ser testado. Esse bloco recebe apenas um dos sinais, por exemplo, o *clock* (*clk*), e fornece ambos os sinais (*clk* e *nclk*) para o bloco principal instanciado dentro dele.

A Figura 25 ilustra um exemplo do uso desse bloco em uma Célula Primitiva simples nomeada *mem_p1latch* que representa um *latch*.

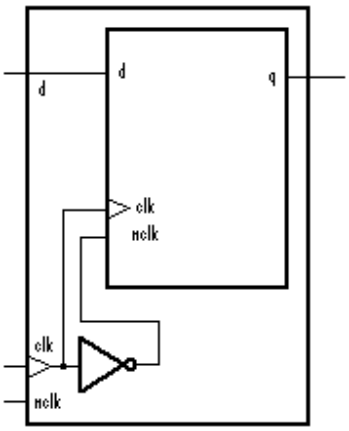

Figura 25: Empacotador da Célula Primitiva que representa um *latch*

5.5 Resultados

Utilizando as estimativas feitas para o tempo de desenvolvimento das *leaf-cells* e Células Primitivas, é possível remarcar que o tempo total de desenvolvimento de todas elas junta foram de aproximadamente 8420 minutos ou 140 horas para ambas as arquiteturas. Esse cálculo é mostrado na Tabela 2. As estimativas do tempo de desenvolvimento de cada Célula Primitiva e *leaf-cell* podem ser vistas no Apêndice C, Apêndice D, Apêndice G e Apêndice H.

Tabela 2: Tempo de desenvolvimento estimado dos modelos

	Tempo de desenvolvimento (RF2)	Tempo de desenvolvimento (RF1)
Células Primitivas (ESP-CV)	815	1690
Células Primitivas (Conformal)	560	1765
Leaf-cells	1760	1830
Sub-total	3135	5285

Também é possível ver que o tempo de desenvolvimento para todas as Células Primitivas foram de aproximadamente 4830 minutos. Esse tempo significa 74,33% do tempo total de desenvolvimento. Isso prova que se for feita uma biblioteca de modelos em *Verilog* para as Células Primitivas, isso diminuiria consideravelmente o tempo de desenvolvimento das *leaf-cells* em longo prazo. A ideia principal para melhorar o tempo de desenvolvimento é criar uma biblioteca de modelos em *Verilog* para as Células Primitivas para cada processo de tecnologia diferente. Uma vez as primitivas estão feitas, é bastante fácil, rápido e seguro de escrever as *leaf-cells*. Em consequência disso, o tempo de desenvolvimento também decresce, já que se pode reutilizar o código delas em novas arquiteturas de uma mesma tecnologia.

Com todos os modelos das Células Primitivas e *leaf-cells* feitos para ambas as arquiteturas, foi possível simular um grande número de modelos de memórias diferentes e analisar as diferenças dos modelos atuais. Todas as memórias testadas podem ser vistas no Apêndice K para a arquitetura Arquivo de Registradores de Porta Dupla e no Apêndice L para a arquitetura Arquivo de Registradores de Porta Simples. Nestes apêndices, pode ser visto que foi escolhida certa variedade de memórias de diferentes tamanhos e combinações diversas das opções suplementares.

As simulações dos modelos foram sempre usando a ferramenta ESP-CV e comparando o Modelo Comportamental em *Verilog* e o Modelo Hierárquico em *Verilog* com a respectiva *netlist* SPICE. Os *scripts* e vetores com os estímulos foram sempre gerados por um compilador de memória juntamente da *netlist* SPICE e do Modelo Comportamental em *Verilog*. Os mesmos foram reutilizados posteriormente para simular com o novo modelo. No Apêndice M e no Apêndice N são mostrados os resultados de todas as simulações de todas as memórias de ambas as arquiteturas.

Apesar das melhorias supracitadas, como o tempo de desenvolvimento e a facilidade de alteração dos modelos, o tempo requerido da CPU para simular foi aumentado. A Tabela 3 indica uma média entre as diferenças de tempo de CPU requisitado mostrados no Apêndice M e no Apêndice N.

Tabela 3: Média das diferenças entre os tempos de CPU das simulações

	RF2	RF1
Média das diferenças	3,33%	16,09%

Apesar dos ganhos citados anteriormente, o tempo de CPU de simulação das memórias aumentou pouco e foram classificados apenas como uma pequena desvantagem no novo fluxo de projeto.

5.6 Dificuldades

Os resultados obtidos no estágio são suficientes para provar o conceito de que a nova metodologia melhora no tempo de desenvolvimento, assim como facilita modificações e diminui a probabilidade de erros no projeto.

Por outro lado, existem ainda alguns pontos que poderiam ser melhorados antes de se colocar a nova metodologia em prática. O ponto principal é o tempo de CPU requisitado que deveria ser pelo menos o mesmo ou até mesmo menor do que a metodologia atual. Algum estudo de como escrever o Modelo Hierárquico em *Verilog* mais rápido pode ser realizado em trabalhos futuros.

O script *Perl* que traduz a sintaxe SPICE para a sintaxe *Verilog* pode ser também outro ponto a ser melhorado. Por exemplo, algumas vezes existem diferentes sinais do mesmo bloco na *netlist* SPICE nomeados de *SO*, *SO[0]* e *SO[1]*. Quando ele é traduzido para a sintaxe em *Verilog*, o segundo e o terceiro sinais

fazem um barramento de dois *bits* com o identificador *SO*. Isso ocorre, embora o primeiro sinal seja diferente e também seja chamado de *SO*. Este conflito deve ser reparado, ou no padrão dos nomes dos sinais desde o início do projeto em todos os circuitos e modelos ou simplesmente no *script*.

Um estudo sobre verificação formal deveria ser realizado para tentar simular também os blocos analógicos utilizando a ferramenta Conformal. Caso seja possível, essa ferramenta poderia ser utilizada como uma alternativa para o ESP-CV. Caso contrário, uma alternativa para o ESP-CV precisa ser encontrada para evitar algumas restrições e problemas internos na empresa.

Finalmente, o mesmo processo poderia ser feito para outras arquiteturas com diferentes tecnologias. Isso possibilitaria verificar melhor a questão da reutilização de código das Células Primitivas, até mesmo em tecnologias diferentes.

6. Conclusão

O objetivo principal deste trabalho era investigar a possibilidade de implantação de uma nova metodologia e as suas vantagens e desvantagens. Para isto, foi proposto fazer o uso de um novo gerador de modelos em Verilog para compiladores de memória.

Depois de entender as especificidades gerais da memória SRAM, o trabalho foi focado em uma das arquiteturas de memória SRAM particular (RF2). Por estar adiantado no cronograma inicialmente proposto, demonstrando o bom progresso no desenvolvimento do estágio, foi possível realizar o mesmo trabalho para a segunda arquitetura (RF1). Isso permitiu adquirir um melhor resultado no final do estudo, com dados mais concisos. Consequentemente, isso possibilitou uma maior ajuda na conclusão sobre essa nova metodologia, determinando se ela é vantajosa para ser integrada no fluxo atual de desenvolvimento de compiladores de memória.

Primeiramente, iniciou-se a escrita dos modelos em *Verilog* para algumas Células Primitivas que poderiam ser utilizados posteriormente para fazer modelos de memórias. Cada uma delas foi simulada com a sua respectiva *netlist* SPICE. Na posse dessas primitivas, fez-se o modelo de referência em *Verilog* para uma memória simples. Com isso, foi possível simular a lógica do esquemático de uma memória inteira.

Em seguida, foi possível escrever todos os blocos primitivos que estavam faltando para fazer todos os tipos de memória para uma arquitetura e assim foi escrito um *script* que colocava todos os módulos juntos para gerar modelos em *Verilog* automaticamente. Finalmente, o novo modelo em *Verilog* para compiladores de memória proposto no meu estágio estava pronto.

Dado que foi possível realizar todo o estudo em um pouco mais da metade do cronograma inicialmente proposto, todo o estudo anterior foi repetido para uma segunda arquitetura. Com isso, possibilitou-se o desenvolvimento de um grande número de simulações utilizando esse novo tipo de modelos em *Verilog* para memórias, provando a importância desse novo conceito. Portanto, o estudo dessa nova metodologia merece ser continuado para que em um futuro próximo ela possa ser integrada no fluxo atual de desenvolvimento de compiladores de memórias.

Pessoalmente, este estágio foi uma excelente experiência para o meu início de carreira. Este estudo aprimorou o meu conhecimento de descrições em *Verilog* e me levou a aprender conceitos sobre memórias em geral e sobre novas arquiteturas de memórias SRAM. Além disso, eu também tive a experiência de trabalhar em uma das mais importantes *Design Houses* do mundo como a ARM Ltda.. Também aprendi bastante a respeito de microeletrônica analógica e digital com os meus companheiros de trabalho dentro da empresa, onde eu tive a oportunidade de interagir e adquirir conhecimentos tanto de aspectos sociais como técnicos.

7. Referências Bibliográficas

[1] Aghetti Bastien (2008), “New Architecture for advanced memory compiler”, internship report belonging to ARM Ltd, Grenoble Design Center PIPD, France;

[2] Just Guillaume (2009), “Power-oriented SRAM design evaluation”, internship report belonging to ARM Ltd, Grenoble Design Center PIPD, France;

[3] ARM Holdings. <http://en.wikipedia.org/wiki/ARM_Holdings>. Acesso em: 2 de Março de 2013;

[4] ARM. Disponível em: <http://www.arm.com/>. Acesso em: 2 de Março de 2013;

[5] David Hodges, Horace Jackson, Resve Saleh, “Analysis and Design of Digital Integrated Circuits”, Third Edition (2003), McGraw-Hill Science/Engineering/Math;

[6] Morris Mano, “Logic and Computer Design Fundamentals”, Third Edition (2003), Prentice Hall;

[7] SPICE 3 User’s Manual. Disponível em: <http://newton.ex.ac.uk/teaching/CDHW/Electronics2/userguide/>. Acesso em: 13 de Maio de 2013;

[8] IEEE Standard Verilog Hardware Description Language, "*IEEE Std 1364-2001*, vol., no., pp.0_1,856, 2001;

[9] Synopsys. Disponível em: <http://www.synopsys.com>. Acesso em: 13 de Maio de 2013;

[10] Bibliotecas de *Standard Cells* da ARM. Disponível em: <http://www.arm.com/products/physical-ip/logic-ip/standard-cell-libraries.php>. Acesso em: 13 de Maio de 2013;

[11] Steven M. Rubin, “Computer Aids for VLSI Design”, 1994, Appendix C: GDS II Format;

[12] Comparison of EDA software. Disponível em: <http://en.wikipedia.org/wiki/Comparison_of_EDA_software>. Aceso em: 13 de Maio de 2013;

[13] Site do ESP-CV. Disponível em: <http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/ESP-CV.aspx>. Acesso em: 13 de Maio de 2013;

[14] Site do Encounter Conformal Equivalence Checker. Disponível em: <http://www.cadence.com/products/ld/equivalence_checker/pages/default.aspx>. Acesso em: 13 de Maio de 2013;

[15] IEEE Standard for System Verilog- Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2005* , vol., no., pp.0_1,648, 2005.

8. Sumário dos Apêndices

Apêndice A. Descrição dos pinos das memórias SRAM da arquitetura Arquivo de Registradores de Porta Dupla

Pinos Básicos

Nome (Porta A)	Nome (Porta B)	Tipo	Descrição
AA[m-1:0]	**AB[m-1:0]**	Entrada	Adress
CENA	**CENB**	Entrada	Chip Enable (active low)
CLKA	**CLKB**	Entrada	Clock
EMAA[1:0]	**EMAB[1:0]**	Entrada	Extra Margin Adjustment
QA[n-1:0]		Saída	Data Outputs
	DB[n-1:0]	Entrada	Data Inputs
	WENB[]	Entrada	Write Enable (active low)

Pinos do BIST

Nome (Porta A)	Nome (Porta B)	Tipo	Descrição
TENA	**TENB**	Entrada	Test mode enable (active low)
TAA[m-1:0]	**TAB[m-1:0]**	Entrada	Test mode address
AYA[m-1:0]	**AYB[m-1:0]**	Saída	Address MUX output
TCENA	**TCENB**	Entrada	Test mode chip enable
CENYA	**CENYB**	Saída	Chip enable MUX output
SEA	**SEB**	Entrada	Scan enable input
SIA[1:0]	**SIB[1:0]**	Entrada	Scan input
SOA[1:0]	**SOB[1:0]**	Saída	Scan output
	TDB[n-1:0]	Entrada	Test mode data inputs
	TWENB[]	Entrada	Test mode write enable (active low)
	WENYB[]	Saída	Write enable MUX output

Pinos do Pipeline

Nome	Tipo	Descrição
TPENA	Entrada	Pipeline test mode enable
PENA	Entrada	Pipeline enable (active low)
PENSIA	Entrada	PEN scan input, for scan testing
PENSOA	Saída	Output of pipeline enable register, for scan testing
PSIA[1:0]	Entrada	Pipeline register scan input
PSOA[1:0]	Saída	Pipeline register scan output

Pinos da Redundância

Nome	Tipo	Descrição
CRE1	Entrada	Column redundancy enable for LSB (active high)
CRE2	Entrada	Column redundancy enable for MSB (active high)
FCAi [(n-1):0]	Entrada	Fuse address

Pinos da Comutação de Alimentação

Nome	Tipo	Descrição
RET1N	Entrada	Retention mode 1 enable (active low)
RET2N	Entrada	Retention mode 2 enable (active low)
PGEN	Entrada	Power mode enable (active high)

Pinos de Alimentação e Terra

Nome	Tipo	Descrição
VDDPE	Entrada	Periphery power supply pin
VDDCE	Entrada	Core array power supply pin
VSSE	Entrada	Core and periphery ground pin

Pinos de Back-Bias

Nome	Tipo	Descrição
BIASNW	Entrada	Periphery nwell pin
BIASCNW	Entrada	Core nwell pin
BIASPW	Entrada	Psubstrate

Pinos de Controle de Teste

Nome	Tipo	Descrição
DFTRAMBYP	Entrada	Test control input

Apêndice B. Descrição dos pinos das memórias SRAM da arquitetura Arquivo de Registradores de Porta Simples

Pinos Básicos

Nome	Tipo	Descrição
A[m-1:0]	Entrada	Addresses
CEN	Entrada	Chip Enable (active low)
CLK	Entrada	Clock
EMA[1:0]	Entrada	Extra Margin Adjustment
Q[n-1:0]	Saída	Data Outputs
D[n-1:0]	Entrada	Data Inputs
WEN[]	Entrada	Write Enable (active low)

Pinos do BIST

Nome	Tipo	Descrição
TEN	Entrada	Test mode enable (active low)
TA[m-1:0]	Entrada	Test mode address
AY[m-1:0]	Saída	Address MUX output
TCEN	Entrada	Test mode chip enable
CENY	Saída	Chip enable MUX output
SE	Entrada	Scan enable input
SI[1:0]	Entrada	Scan input
SO[1:0]	Saída	Scan output
TD[n-1:0]	Entrada	Test mode data inputs
TWEN[]	Entrada	Test mode write enable (active low)
WENY[]	Saída	Write enable MUX output

Pinos do Pipeline

Nome	Tipo	Descrição
TPEN	Entrada	Pipeline test mode enable
PEN	Entrada	Pipeline enable (active low)
PENSI	Entrada	PEN scan input, for scan testing
PENSO	Saída	Output of pipeline enable register, for scan testing
PSI[1:0]	Entrada	Pipeline register scan input
PSO[1:0]	Saída	Pipeline register scan output

Pinos de Redundância

Nome	Tipo	Descrição
CRE1	Entrada	Column redundancy enable for LSB (active high)
CRE2	Entrada	Column redundancy enable for MSB (active high)
FCAi [(n-1):0]	Entrada	Fuse address

Pinos da Comutação de Alimentação

Nome	Tipo	Descrição
RET1N	Entrada	Retention mode 1 enable (active low)

Pinos de Alimentação e Terra

Nome	Tipo	Descrição
VDDPE	Entrada	Periphery power supply pin
VDDCE	Entrada	Core array power supply pin
VSSE	Entrada	Core and periphery ground pin

Pinos de Back-Bias

Nome	Tipo	Descrição
BIASNW	Entrada	Periphery nwell pin
BIASCNW	Entrada	Core nwell pin
BIASPW	Entrada	Psubstrate

Pinos de Controle de Teste

Nome	Tipo	Descrição
DFTRAMBYP	Entrada	Test control input

Apêndice C. *Leaf-cells* modeladas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Dupla

Leaf-cell	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
cc2x2	150	1,26
cmuxsa	60	0,88
cntla	270	3,37
cntla_m1	35	3,3
cntlb	150	3,18
cntlb_m1	35	3,13
dpth_rw_edge	25	0,79
dpth_rw_edge_m1	20	0,73
dpth_rw_m1	40	2,4
dpth_rw_m2	150	2,36
dpth_rw_m4	40	4,4
dpth_rw_m8	40	9,27
pg_cntl	50	2,55
pg_edge	40	0,44
pg_m1	20	2,39
pg_m2	35	2,4
pg_m4	20	2,38
pg_m8	20	2,35
pl_cntl	45	2,23
pl_cntl_m1	20	2,22
pl_m1	35	0,65
pl_m2	40	0,82
pl_m4	20	0,74
pl_m8	20	0,87
red_cntl	35	1,19
red_m1	30	1,02
red_m2	35	0,98
red_m4	20	0,83
red_m8	20	0,86
Sadrv	40	1,02
tpp251	50	
wldrvx2	150	1,39

Apêndice D. *Leaf-cells* modeladas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Simples

Leaf-cell	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
cch4	150	5,42
cchs16x4	30	6,9
ck	30	8,16
ck_bist	30	11,84
ck_bist_pl	30	10,71
ck_bist_red	30	15,84
ck_bist_red_pl	30	16,34
ckm2	35	9,81
ckm2_bist	25	12,17
ckm2_bist_pl	30	13,08
ckm2_bist_pl_wdx8	25	12,14
ckm2_bist_red	35	13,12
ckm2_bist_red_pl	40	13,83
ckm2_bist_wdx8	120	12,14
ckm2_std	30	8,86
ckm2_wdx8	40	10,04
ckm2_wdx8_std	30	8,87
colm2x2	25	4,21
colm2x2_bist	40	7
colm2x2_bist_pl	25	6,2
colm2x2_bist_red	120	7,16
colm2x2_bist_red_centre	25	6,86
colm2x2_bist_red_pl	35	8,93
colm2x2_bist_red_pl_centre	25	8,36
colm2x2_std	25	4,19
colm4	25	4,16
colm4_bist	25	10,02
colm4_bist_pl	25	10,37
colm4_bist_red	25	12,66
colm4_bist_red_centre	25	12,24
colm4_bist_red_pl	25	12,75
colm4_bist_red_pl_centre	25	12,53
colm4_std	25	4,06

Leaf-cell	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
colm8	25	5,75
colm8_bist	25	12,38
colm8_bist_pl	25	10,41
colm8_bist_red	25	15,16
colm8_bist_red_centre	25	14,57
colm8_bist_red_pl	25	15,77
colm8_bist_red_pl_centre	25	15,47
colm8_std	25	5,49
midtop	20	3,46
p152_cell	90	
wdtop	25	3,61
wdx16	30	4,04
wdx2	25	3,38
wdx32	30	8,29
wdx64	30	10,34
wdx8	120	3,36

Apêndice E. Exemplo do esquemático de uma *leaf-cell* modelada *(cntla)*

Apêndice F. Exemplo do modelo de referência de uma *leaf-cell* modelada em *Verilog (cntla)*

```
module ck_base (blprech_l, blprech_r, gtp, iwclk0, iwclk1, la_p2, nla_p2, nsae, nwtdft, nyr0, nyr1,
nyr2, nyr3, rdmux, ret, saprech, yw0, yw1, yw2, yw3, dbl, log0, log0_sa, nact_cc, nra_p2, ra_p2,
vddce, vddp, vddpe, vnwc, vnwp, vpw, vss, vsse, cedft, clk, dftrambyp, ema, emas, emaw, na0, na1,
na2, na3, na4, na5, na6, na7, na8, na9, na10, ngwen, nwr_thru, rdt, ret1n);

  input vddce, vddpe, vsse;
  inout vddp, vnwc, vnwp, vpw, vss;
  input [2:0] ema;
  input [1:0] emaw;
  input cedft, clk, dftrambyp, emas, na0, na1, na2, na3, na4, na5, na6, na7, na8, na9, na10, ngwen,
        nwr_thru, rdt, ret1n;
  inout dbl, log0, log0_sa, nact_cc;
  output [5:0] ra_p2, nra_p2;
  output [1:0] la_p2, nla_p2;
  output blprech_l, blprech_r, gtp, iwclk0, iwclk1, nsae, nwtdft, nyr0, nyr1, nyr2, nyr3, rdmux, ret,
         saprech, yw0, yw1, yw2, yw3;

  wire nclk, bclk, niclk;
  wire ndft_p2, dft_p2, gwen_p2, nret, ngtpemas, ck_emas, nsetsa;
  wire rdt_p2, nblpreclk;
  wire [10:0] a_p2, na_p2;
  wire nrda_p1, iwen0, iwen1;
  wire isaprech;

  //gtp clock generator & EMA delay
  not (nclk, gtp);
  not (niclk, gtp);
  not (bclk, niclk);
  clkgen_hs_bist clkgen_hs_bist_i (ck_emas, gtp, nsetsa, vddpe, vddpe, vnwp, vpw, vss, vsse, cedft,
                                   clk, dbl, ema, emaw, gwen_p2, ndft_p2, ngtpemas, nret);

  //RDT latch and bit line precharge
  nand (nblpreclk, gtp, !(dft_p2 || (gwen_p2 && rdt_p2)));
  ctl_prechbl ctl_prechbl_i (blprech_l, blprech_r, vddpe, vnwp, vpw, vss, vsse, nblpreclk, nret, {nyr3,
                             nyr2, nyr1, nyr0});
  mem_p2latch mem_p2latch_0 (vddpe, vnwp, vss, vpw, nclk, bclk, rdt, rdt_p2, );

  //PGEN logic - VDDCE DOMAIN
  buf (nret, ret1n);
  not (ret, nret);

  //gwen latch
  mem_p2latch_bist mem_p2latch_bist_0 ( , gwen_p2, vddpe, vnwp, vpw, vss, bclk, nclk, ngwen);

  //column address latches
  mem_p2latch_bist mem_p2latch_bist_1 (na_p2[0], a_p2[0], vddpe, vnwp, vpw, vss, bclk, nclk, na0);
  mem_p2latch_bist mem_p2latch_bist_2 (na_p2[1], a_p2[1], vddpe, vnwp, vpw, vss, bclk, nclk, na1);
```

```
ydec2to4 ydec2to4_i ({nyr3, nyr2, nyr1, nyr0}, {yw3, yw2, yw1, yw0}, vddpe, vnwp, vpw, vss, vsse,
                        a_p2[1:0], dft_p2, gtp, gwen_p2, na_p2[1:0]);

//row address latches
nor (nla_p2[1], a_p2[10], dft_p2);
nor (la_p2[1], na_p2[10], dft_p2);
not (nra_p2[5], a_p2[9]);
not (nra_p2[4], a_p2[8]);
not (nra_p2[3], a_p2[7]);
not (nra_p2[2], a_p2[6]);
not (nra_p2[1], a_p2[5]);
not (nra_p2[0], a_p2[4]);
not (ra_p2[5], na_p2[9]);
not (ra_p2[4], na_p2[8]);
not (ra_p2[3], na_p2[7]);
not (ra_p2[2], na_p2[6]);
not (ra_p2[1], na_p2[5]);
not (ra_p2[0], na_p2[4]);
nor (nla_p2[0], a_p2[3], dft_p2);
nor (la_p2[0], na_p2[3], dft_p2);
mem_p2latch_bist mem_p2latch_bist_3 (na_p2[3], a_p2[3], vddpe, vnwp, vpw, vss, bclk, nclk, na3);
mem_p2latch_bist mem_p2latch_bist_4 (na_p2[4], a_p2[4], vddpe, vnwp, vpw, vss, bclk, nclk, na4);
mem_p2latch_bist mem_p2latch_bist_5 (na_p2[5], a_p2[5], vddpe, vnwp, vpw, vss, bclk, nclk, na5);
mem_p2latch_bist mem_p2latch_bist_6 (na_p2[6], a_p2[6], vddpe, vnwp, vpw, vss, bclk, nclk, na6);
mem_p2latch_bist mem_p2latch_bist_7 (na_p2[7], a_p2[7], vddpe, vnwp, vpw, vss, bclk, nclk, na7);
mem_p2latch_bist mem_p2latch_bist_8 (na_p2[8], a_p2[8], vddpe, vnwp, vpw, vss, bclk, nclk, na8);
mem_p2latch_bist mem_p2latch_bist_9 (na_p2[9], a_p2[9], vddpe, vnwp, vpw, vss, bclk, nclk, na9);
mem_p2latch_bist mem_p2latch_bist_10 (na_p2[10], a_p2[10], vddpe, vnwp, vpw, vss, bclk, nclk,
                                        na10);

//Mux 8 logic only
nand (iwen0, ndft_p2, (gwen_p2 || a_p2[2]));
nor (iwen1, gwen_p2, na_p2[2]);
buf (rdmux, nrda_p1);
mem_p2latch_bist mem_p2latch_bist_11 (na_p2[2], a_p2[2], vddpe, vnwp, vpw, vss, bclk, nclk, na2);
mem_mux2p2latch mem_mux2p2latch_i (vddpe, vnwp, vss, vpw, gtp, nclk, gwen_p2, nrda_p1,
                                     a_p2[2], nrda_p1, );

//SA enable & precharge
nand (isaprech, gtp, gwen_p2, ndft_p2);
and (saprech, isaprech, nsetsa);
emas emas_i (ngtpemas, nsae, vddpe, vnwp, vpw, vss, ck_emas, emas, nsetsa);

//DFTRAMBYP latch, write, clocks, write-thru logic
not (ndft_p2, dft_p2);
mem_p2latch mem_p2latch_1 (vddpe, vnwp, vss, vpw, nclk, bclk, dftrambyp, dft_p2, );
ctl_write_m8 ctl_write_m8_i (iwclk0, iwclk1, nwtdft, vddpe, vnwp, vpw, vss, gtp, gwen_p2, iwen0,
                               iwen1, ndft_p2, nwr_thru);
endmodule
```

Apêndice G. Células Primitivas modeladas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Dupla

Simulados com o Conformal

Célula Primitiva	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
addr_lat	35	0,79
ck_dly_2_2	20	0,75
ck_mux	30	2,2
col_dec	25	0,84
mem_dec2to4_pla	30	0,85
mem_dec3to8_pla	30	0,86
mem_mux2_byp_pla_noinv	30	0,84
mem_mux2p2latch	30	0,84
mem_mux2p2latch_byp_pla	30	0,88
mem_mux3dff_v2_pla	35	0,89
mem_mux3p2latch_pla	35	0,96
mem_p1latch_pla	25	0,76
mem_p2latch	25	0,75
mem_pgctrl_pla	35	0,81
mem_pipectrl_pla_pl_cntl	35	0,82
mem_scanlogic_pla	20	0,82
mem_tiehilo	30	0,79
mem_tmux2x1_pla	35	0,75
row_dec	25	0,86

Simulados com o ESP-CV

Célula Primitiva	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
adv_ret	25	0,9
dpth_bit	150	1,74
mem_gtp_latch	270	1,08
Timinga	90	1,64
Timingb	40	1,31
Wrdrv	210	0,64
wrdrv_m1	30	0,62

Apêndice H. Células Primitivas modeladas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Simples

- *Simulados com o Conformal*

Célula Primitiva	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
bist_ck	30	0,8
bist_col	30	0,76
ck_dly_2_2	10	0,75
ck_dly_2_2_a	10	0,73
ck_mux	20	0,75
ck_mux_tg_hs	25	0,81
ck_rd_mux	35	0,85
clkgen_pulse_hs	150	0,88
ctl_prechbl	30	0,83
ctl_write_m8	35	0,83
Emas	30	0,76
ibuf_ck	30	0,76
ibuf_col	25	0,82
m2_bist_ck	40	0,81
m2_mem_pipectrl	30	0,82
m2_pl_ck	20	0,81
m2_pl_col	20	0,78
m2_red_ck	20	0,84
mem_d2wldrv	120	0,75
mem_dec2to4	15	0,75
mem_dec3to8	15	0,76
mem_mux2dff_pipe_v2	25	0,78
mem_mux2p2latch	10	0,83
mem_mux3dff_v2	15	0,85
mem_nor2_enable	120	0,79
mem_p1latch	10	0,77
mem_p2latch	10	0,76
mem_p2latch_bist	15	0,76
mem_pipectrl	35	0,81
mem_salat_sdl_wt	60	0,77
mem_tiehilo	10	0,73

Célula Primitiva	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
pl_ck	25	0,81
pl_col	25	0,78
red_ck	45	0,77
red_col	25	0,79
RF1ADec8	20	0,78
RF1RowDrvx16	20	0,76
Rmux	90	0,9
tiehilo_hs	15	0,74
tri_wr	25	0,78
tri_wr_m2	45	0,77
wdx2_base	150	0,79
Wmux	30	0,92
wrm2x2	120	0,81
wrm4	25	0,82
wrm8	25	1,01
ydec2to4	30	0,96

- *Simulados com o ESP-CV*

Célula Primitiva	Tempo de Desenvolvimento Estimado (min)	Tempo de CPU (s)
cdm2x2	210	2,11
cdm4	30	2,02
cdrwx2	90	0,73
ck_base	90	5,1
clkgen_hs_bist	45	1,37
mem_cmux1	420	0,68
mem_dbl_hs	90	0,87
mem_gtp_bist	30	1,15
mem_senseamp	450	0,66
As	25	0,66
sax2	210	0,72

Apêndice I. Exemplo do esquemático de uma Célula Primitiva modelada *(mem_p2latch_bist)*

Apêndice J. Exemplo do modelo de referência de uma Célula Primitiva modelada em *Verilog (mem_p2latch_bist)*

```
module mem_p2latch_bist (nq, q, vdd, vnw, vpw, vss, clk, nclk, nd);

   input vdd;
   inout vnw, vss, vpw;
   input nclk, clk, nd;
   output q, nq;

   reg q_;

   assign q = ((vdd === 1'b1) && (vss === 1'b0)) ? q_ : 1'bx;

   assign nq = ~q;

   always @(clk or nclk or nd) begin
      if ((clk === 1'b0) && (nclk === 1'b1)) begin
         q_ <= ~nd;
      end
   end

endmodule
```

Apêndice K. Simulações realizadas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Dupla

Instância	Words	*Bits*	Mux	BIST Mux	Pipe line	Red.	Másc. de Escrita	Com. de Alim.
i1024x40m8	1024	40	8	on	off	off	off	off
i1024x4m8	1024	4	8	on	off	off	off	off
i12x14m2pl	12	14	2	on	on	off	off	off
i12x14m2red	12	14	2	on	off	on	off	off
i12x4m1wm	12	4	1	on	off	off	on	off
i12x4m2wm	12	4	2	on	off	off	on	off
i24x5m4	24	5	4	1	off	off	off	off
i24x5m4bmux	24	5	4	on	off	off	off	off
i24x5m4red	24	5	4	on	off	on	off	off
i256x128m2	256	128	2	on	off	off	off	off
i256x14m2red	256	14	2	on	off	on	off	off
i256x24m2redwm	256	24	2	on	off	on	on	off
i256x24m2redwmpg	256	24	2	24	off	on	on	on
i256x34m2redwmpl	256	34	2	34	on	on	on	off
i256x34m2redwmplpg	256	34	2	34	on	on	on	on
i256x44m2redplpg	256	44	2	on	on	on	off	on
i48x5m8	48	5	8	1	off	off	off	off
i512x64m4	512	64	4	on	off	off	off	off
i64x54m2plpg	64	54	2	on	on	off	off	on
i8x4m2b	8	4	2	1	off	off	off	off
i8x4m2brp	8	4	2	on	on	on	off	off
i8x4m2bwp	8	4	2	on	on	off	on	off

Apêndice L. Simulações realizadas das memórias SRAM da arquitetura Arquivo de Registradores de Porta Simples

Instância	Words	Bits	Mux	BIST Mux	Pipe line	Redundância	Máscara de Escrita
rf1hs1024x32m8	1024	32	8	off	off	off	off
rf1hs1024x36m8	1024	36	8	off	off	off	on
rf1hs128x10m8rp	128	10	8	on	on	on	off
rf1hs128x15m8r	128	15	8	on	off	on	off
rf1hs128x28m8p	128	28	8	on	on	off	on
rf1hs128x36m8	128	36	8	off	off	off	off
rf1hs128x4m8b	128	4	8	on	off	off	on
rf1hs16x124m2p	16	124	2	on	on	off	on
rf1hs16x4m2	16	4	2	off	off	off	off
rf1hs16x4m2b	16	4	2	on	off	off	on
rf1hs256x144m2	256	144	2	off	off	off	off
rf1hs256x144m2w1	256	144	2	off	off	off	on
rf1hs256x16m4	256	16	4	off	off	off	off
rf1hs256x32m4w1	256	32	4	off	off	off	on
rf1hs256x36m4	256	36	4	off	off	off	off
rf1hs256x8m4	256	8	4	off	off	off	off
rf1hs32x16m2r	32	16	2	on	off	on	off
rf1hs32x16m2rp	32	16	2	on	on	on	on
rf1hs512x16m4	512	16	4	off	off	off	off
rf1hs512x32m4	512	32	4	off	off	off	off
rf1hs512x72m4	512	72	4	off	off	off	off
rf1hs512x8m8	512	8	8	off	off	off	on
rf1hs64x15m4rp	64	15	4	on	on	on	off
rf1hs64x18m4r	64	18	4	on	off	on	on
rf1hs64x24m4b	64	24	4	on	off	off	off
rf1hs64x33m4	64	33	4	off	off	off	off
rf1hs64x52m4p	64	52	4	on	on	off	off
rf1hs64x72m4	64	72	4	off	off	off	on

Apêndice M. Resultados das simulações das memórias SRAM da arquitetura Arquivo de Registradores de Porta Dupla

Instância	Tempo de CPU (s) *Modelo Verilog Comportamental*	Tempo de CPU (s) *Modelo Verilog Hierárquico*	Diferença (%)	Memória máxima utilizada (MB) *Modelo Verilog Comportamental*	Memória máxima utilizada (MB) *Modelo Verilog Hierárquico*
i1024x40m8	495,22	696,01	40,55	442	347
i1024x4m8	86,01	100	16,27	129	105
i12x14m2pl	65,78	60,58	-7,91	86	65
i12x14m2red	134,49	126,68	-5,81	234	95
i12x4m1wm	30,68	29,48	-3,91	80	47
i12x4m2wm	34,65	32,68	-5,69	78	38
i24x5m4	46,69	43,75	-6,30	87	35
i24x5m4bmux	46,6	44,59	-4,31	86	66
i24x5m4red	57,64	61,32	6,38	111	68
i256x128m2	1202,06	1428,89	18,87	485	323
i256x14m2red	166,67	199,59	19,75	248	152
i256x24m2redwm	457,87	526,79	15,05	418	245
i256x24m2redwmpg	1534,01	822,56	-46,38	1041	241
i256x34m2redwmpl	1278,41	1342,59	5,02	445	262
i256x34m2redwmplpg	1999,61	2444,32	22,24	549	411
i256x44m2redplpg	4320,44	2681,94	-37,92	710	404
i48x5m8	58,16	68,58	17,92	58	78
i512x64m4	645,3	690,52	7,01	427	287
i64x54m2plpg	282,37	357,18	26,49	199	123
i8x4m2b	31,43	29,5	-6,14	79	36
i8x4m2brp	53,33	52,58	-1,41	103	65
i8x4m2bwp	37,94	39,3	3,58	79	34
Média	593,88	539,9741	3,33%	280,6364	160,3182

Apêndice N. Resultados das simulações das memórias SRAM da arquitetura Arquivo de Registradores de Porta Simples

Instância	Tempo de CPU (s) *Modelo Verilog Comportamental*	Tempo de CPU (s) *Modelo Verilog Hierárquico*	Diferença (%)	Memória máxima utilizada (MB) *Modelo Verilog Comportamental*	Memória máxima utilizada (MB) *Modelo Verilog Hierárquico*
rf1hs1024x32m8	1533,22	1880,28	22,64	1509	471
rf1hs1024x36m8	3598,4	1901,72	-47,15	1335	510
rf1hs128x10m8rp	108,29	101,69	-6,09	243	104
rf1hs128x15m8r	139,94	120,44	-13,93	386	148
rf1hs128x28m8p	219,23	258,9	18,10	660	226
rf1hs128x36m8	452,05	571,68	26,46	418	238
rf1hs128x4m8b	36,31	41,82	15,17	145	50
rf1hs16x124m2p	541,05	590,53	9,15	421	231
rf1hs16x4m2	8,45	6,92	-18,11	87	34
rf1hs16x4m2b	16,57	15,08	-8,99	43	34
rf1hs256x144m2	3581,24	4535,45	26,64	1450	469
rf1hs256x144m2w1	3938,49	5032,52	27,78	1337	442
rf1hs256x16m4	154,87	236,28	52,57	378	229
rf1hs256x32m4w1	555,05	866,41	56,10	715	390
rf1hs256x36m4	690,04	877,83	27,21	724	428
rf1hs256x8m4	76,12	122,77	61,28	227	148
rf1hs32x16m2r	44,39	49,95	12,53	120	96
rf1hs32x16m2rp	65,08	66,67	2,44	156	98
rf1hs512x16m4	183,3	254,02	38,58	382	227
rf1hs512x32m4	600,26	805,95	34,27	680	424
rf1hs512x72m4	2368,92	3086,17	30,28	1363	665
rf1hs512x8m8	191,58	221,29	15,51	379	231
rf1hs64x15m4rp	81,88	90,02	9,94	245	136
rf1hs64x18m4r	102,91	88,54	-13,96	373	135
rf1hs64x24m4b	83,55	99,72	19,35	240	139
rf1hs64x33m4	160,53	172,46	7,43	402	145
rf1hs64x52m4p	266,53	378,02	41,83	402	225
rf1hs64x72m4	631,8	653,42	3,42	682	235
Total	729,6446	825,9482	16,0875	553,6429	246,7143