GUSTAVO HENRIQUE NIHEI

MODELAGEM DE MEMÓRIAS SRAM EM NÍVEL RT VISANDO À ANÁLISE DE CONSUMO ENERGÉTICO EM SISTEMAS EMBARCADOS

Florianópolis

2008

GUSTAVO HENRIQUE NIHEI

MODELAGEM DE MEMÓRIAS SRAM EM NÍVEL RT VISANDO À ANÁLISE DE CONSUMO ENERGÉTICO EM SISTEMAS EMBARCADOS

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção de grau de Bacharel em Ciências da Computação

Orientador:
Prof. Dr. José Luís Almada Güntzel

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis

2008

Monografia de graduação sob o título *"Modelagem de Memórias SRAM em Nível RT Visando à Análise de Consumo Energético em Sistemas Embarcados"*, defendida por Gustavo Henrique Nihei e aprovada em 04 de novembro de 2008, em Florianópolis, Santa Catarina, pela banca examinadora constituída pelos professores:

Prof. Dr. José Luís Almada Güntzel
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Luiz Cláudio Villar dos Santos
Universidade Federal de Santa Catarina
Membro da Banca

Prof. Dr. Luís Fernando Friedrich
Universidade Federal de Santa Catarina
Membro da Banca

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ARP	ArchC Reference Platform
CMOS	Complementary Metal-Oxide Semiconductor
DRAM	Dynamic Random Access Memory
FSMD	Finite State Machine with Data
GPS	Global Positioning System
HDL	Hardware Description Language
IP	Intellectual Property
NRE	Non-recurring Engineering
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register-Transfer Level
RWM	Read-Write Memory
SET	Single-Event Transient
SEU	Single-Event Upset
SoC	System-on-Chip (Sistema integrado)
SRAM	Static Random Access Memory
TLM	Transaction-Level Modeling
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
kB	kilo bytes

SUMÁRIO

1 *INTRODUÇÃO*

1.1 CONTEXTO TECNOLÓGICO

Uma significativa parcela da área em silício dos sistemas digitais contemporâneos é dedicada ao armazenamento de dados e instruções de programa (RABAEY; CHANDRAKASAN; NIKOLIC, 2003). Segundo Marinissen et al. (2005), mais de 80% da área de um sistema integrado é ocupada por memória e estima-se que em 2014 esta área corresponderá a cerca de 94%, conforme ilustra a figura 1. A inclusão de blocos de memória no *chip* aumenta a velocidade de tráfego de dados no sistema. Além disso, diminui a quantidade de acessos à memória externa, reduzindo o consumo energético total.

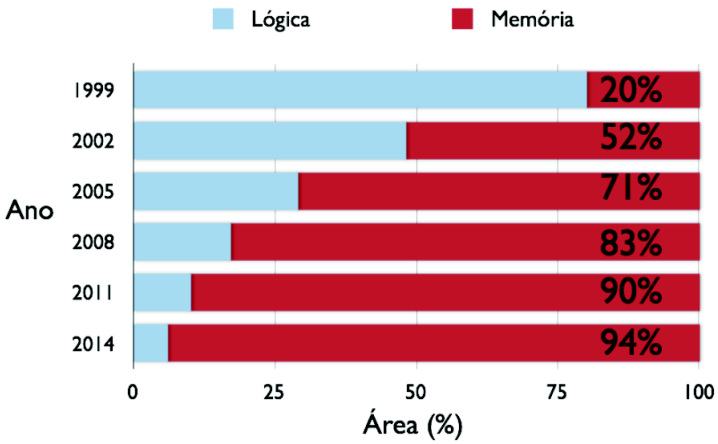

Figura 1: Representação da ocupação de área em chip

Conforme havia previsto o co-fundador da Intel Corporation, Gordon Moore, o número de transistores em um chip dobra a cada 2 anos, aproximadamente. Tal aumento da capacidade de integração se manteve possível devido à contínua redução das dimensões dos transistores, proporcionada pela evolução dos processos CMOS. Porém, isso trouxe novos desafios em decorrência de características do processo de fabricação dos circuitos integrados. Por exemplo, o chamado consumo estático de potência (potência dissipada quando o circuito não está chaveando) era considerado pouco significativo quando comparado ao consumo dinâmico de

potência. Contudo, com os novos processos de fabricação (menores transistores e menores tensões de limiar), esta componente do consumo de potência tem aumentado.

Ao contrário do desempenho e da área, que há muito tempo são capturados nos fluxos de projeto, há ainda muita carência de técnicas e ferramentas para estimativa de potência com bom compromisso entre precisão e velocidade, fazendo com que essa ainda seja uma área de intensiva pesquisa, especialmente nos níveis mais altos de abstração. O aumento da complexidade dos circuitos, a necessidade de explorar o espaço de projeto e a diminuição do *time-to-market* requerem representações mais abstratas do sistema e reuso de componentes (LEãO, 2008).

Além disso, os circuitos integrados fabricados em tecnologias do estado-da-arte estão se tornando cada vez mais vulneráveis a falhas transientes. Dentre estas falhas encontram-se os *single-event upsets* (SEUs) e os *single-event transients* (SETs), ambos causados pela colisão de partículas carregadas em regiões sensíveis dos circuitos. Um SEU corresponde a uma inversão do valor lógico armazenado em uma célula de memória, enquanto um SET é um pulso transiente de tensão que ocorre na lógica combinacional, podendo, através desta, se propagar e, eventualmente, ser capturado por algum elemento de memória (BAUMANN, 2005).

1.2 ESCOPO DO TRABALHO

Este trabalho tem por objetivo propor e analisar um modelo de memória SRAM (*Static Random Access Memory*) que permita capturar detalhes de comportamento para níveis mais abstratos. Para tanto, o projeto é desenvolvido em diferentes níveis de abstração. Inicialmente, apresenta-se um modelo de memória descrito em nível funcional, cuja interface foi desenvolvida com base na metodologia de projeto TLM (*Transaction Level Modeling*), onde as características funcionais dos módulos são separadas da parte de comunicação com o sistema onde a memória está integrada, facilitando seu reuso. Em seguida, inicia-se uma descrição mais aprofundada em nível de transferências entre registradores (RTL – *Register Transfer Level*). Nesse nível menos abstrato o projetista tem liberdade para especificar detalhadamente a arquitetura da memória, podendo modelar cada uma de suas unidades funcionais.

Fazendo uso do conceito de projeto baseado em plataforma, o modelo é incluído em uma plataforma de referência já validada, possibilitando uma maior agilidade na depuração de eventuais erros. O modelo é então submetido a análises de comportamento, considerando o seu desempenho operando em sistema virtual completo, composto por unidades de processamento e um sistema de interconexão.

Devido à modularidade do modelo desenvolvido, esse pode ser utilizado como base para

novas análises e implementações. A modelagem em nível RT é motivada pela possibilidade de agregação de métodos para a estimativa de consumo de energia, que é um dos maiores desafios para o projeto de memórias. Com isso, espera-se que o desenvolvimento deste trabalho em nível de sistema possibilite estimar métricas para determinados casos de uso, como a quantidade de energia consumida por memória durante a execução de uma determinada aplicação.

1.3 ORGANIZAÇÃO DA MONOGRAFIA

O trabalho encontra-se organizado da seguinte forma: o capítulo 2 revisa os principais conceitos abordados no decorrer do trabalho, incluindo uma explicação sobre alguns paradigmas de projeto, além de uma descrição da estrutura interna de uma memória SRAM; o capítulo 3 faz uma breve revisão sobre trabalhos correlatos; o capítulo 4 descreve o processo de modelagem da memória; o capítulo 5 explica a metodologia de validação do modelo de memória, além de uma descrição sobre a infra-estrutura de modelagem da *ArchC Reference Platform* (ARP); já o capítulo 6 expõe os resultados das análises realizadas através de simulações, comparando o comportamento do modelo em seus diferentes níveis de abstração; o capítulo 7 relata as conclusões obtidas e expõe algumas possibilidades de trabalhos futuros.

2 CONCEITOS FUNDAMENTAIS

Este capítulo inicialmente mostra um panorama dos principais paradigmas contemporâneos de projeto e, em seguida, expõe uma breve revisão teórica sobre as características de SRAMs.

2.1 PARADIGMAS DE PROJETO CONSOLIDADOS

Estilos de modelagem de baixa abstração, como nos níveis de transferências entre registradores (RT – *Register Transfer*) e comportamental, vêm sendo gradativamente substituídos pelo projeto em nível de sistema (GRöTKER et al., 2002). Apesar disso, a modelagem nesses níveis de abstração ainda é necessária em projetos de sistemas.

Um grande sistema é geralmente composto por diversos componentes, os quais possuem diferentes requisitos de desempenho, área, consumo de energia, testabilidade, facilidade e flexibilidade de implementação, e outras medidas de custo. Logo, é preciso uma variedade de níveis de abstração para gerenciar esses requisitos. Por exemplo, a interface de um celular não requer o mesmo desempenho que seu processador de sinais. Dados os conflitos entre qualidade dos resultados e nível de abstração do projeto, tais requisitos de desempenho apenas são atingidos com a síntese comportamental ou RTL (GRöTKER et al., 2002).

A grande maioria dos projetos de circuitos digitais é realizada com a descrição do dispositivo desejado em uma linguagem de descrição de *hardware* (HDL – *Hardware Description Language*). HDLs possibilitam a descrição de interconexões estruturais e do comportamento de componentes. Com a utilização de ferramentas de síntese a partir de código HDL, modelos comportamentais podem ser transformados em modelos estruturais, tanto em nível arquitetural quanto em nível lógico. Atualmente, VHDL e Verilog são as linguagens que dominam a indústria de bens eletrônicos.

2.2 PARADIGMAS CONTEMPORÂNEOS DE PROJETO

2.2.1 PROJETO BASEADO EM PLATAFORMA

Segundo Sangiovanni-Vincentelli et al. (2004), o Projeto Baseado em Plataforma é um paradigma de projeto que emergiu da necessidade da indústria eletrônica para lidar com dificuldades relacionadas a três grandes fatores:

- **Desagregação da indústria eletrônica.** Antigamente, as companhias tinham o controle sobre todo o processo de desenvolvimento. Hoje, devido à complexidade dos projetos eletrônicos e ao grande número de tecnologias envolvidas, foram forçadas a concentrarem-se em suas competências principais. Assim, para que pudessem seguir no mercado com produtos de qualidade, a terceirização de partes do processo de produção tornou-se cada vez mais freqüente. Há, portanto, a necessidade de um mecanismo formal para a integração de toda a cadeia de projeto.

- **Redução do *time-to-market*.** Devido ao aumento exponencial da complexidade, a necessidade em reduzir o *time-to-market* dos produtos eletrônicos tem forçado os projetistas de sistemas a adotar técnicas que favorecem o reuso de componentes de hardware em todos os níveis de abstração. Isso também contribui para a flexibilidade do projeto, permitindo ajustes contínuos e mudanças de última hora.

- **Alto custo de engenharia não-recorrente.** O significativo aumento dos custos de engenharia não-recorrente (NRE – *Non-recurring engineering*), devido à fabricação das máscaras no nível de implementação de circuitos integrados e outros fatores, torna crucial que o projeto esteja funcional logo na primeira tentativa de fabricação do *chip*.

Segundo Sangiovanni-Vincentelli et al. (2004), uma plataforma é uma "biblioteca de componentes unida às suas regras de composição". A biblioteca contém não somente blocos computacionais, mas também componentes de comunicação que permitem a interconexão dos componentes funcionais. Assim, um projetista pode escolher um conjunto de componentes da biblioteca ou ajustar parâmetros de componentes reconfiguráveis e derivar, desse modo, sua instância da plataforma.

2.2.2 PROJETO EM NÍVEL DE SISTEMA ELETRÔNICO

As metodologias tradicionais de projeto de sistemas não conseguem atender às necessidades para a integração de tantas funcionalidades em um sistema. Tomemos como exemplo um

celular 3G que é também um dispositivo GPS (*Global Positioning System*), uma câmera digital, reprodutor de vídeo e músicas, terminal *web* e agenda eletrônica, e ainda possui conectividade *Wi-Fi* ou *Bluetooth*. A complexidade de projeto – tanto de hardware quanto de software – é o principal motivo da adoção das metodologias de desenvolvimento em nível de sistema eletrônico (ESL – Electronic System Level design).

Segundo Bailey, Martin e Piziali (2007), uma definição para ESL é a "utilização de abstrações apropriadas para melhorar a compreensão a respeito de um sistema, e para aumentar a probabilidade de sucesso em implementações de funcionalidades de uma maneira custo-benéfica, atendendo aos requisitos necessários".

A migração para o projeto e verificação ESL é um avanço fundamental na metodologia de projetos. Isso proporciona um grande aumento de produtividade e qualidade do projeto, além da redução de risco no desenvolvimento de produtos.

MODELAGEM EM NÍVEL DE TRANSFERÊNCIAS ENTRE REGISTRADORES

Segundo Grötker et al. (2002), o modelo de computação em nível de transferências entre registradores (*RT – Register Transfer*) é um termo informal para a modelagem de *hardware* sincronizado por sinais de relógio.

No estilo RT, toda comunicação entre processos é realizada através de troca de sinais. Os processos podem representar lógicas seqüenciais (sensíveis às transições de relógio) ou lógicas combinacionais (sensíveis às suas entradas). Modelos RTL são *pin-accurate*, ou seja, suas portas correspondem diretamente a fios numa implementação real do modelo. Uma descrição em nível RT é um modelo com precisão de ciclos.

Segundo Vahid (2006), existem alguns passos a serem seguidos para projetar um módulo em RTL:

1. **Capturar uma máquina de estados em um alto nível de abstração (FSMD – *Finite State Machine with Data*).** O comportamento do sistema é descrito como uma máquina de estados. A máquina consiste de estados e transições e é considerada de alto nível pois tanto as condições das transições quanto as ações dos estados são mais do que apenas operações Booleanas nas entradas e saídas.

2. **Criar um bloco operativo.** Criar um bloco operativo (*datapath*) para transportar as operações sobre dados da máquina de estados de alto nível.

3. **Conectar o bloco operativo a um bloco de controle (controlador).** Nessa etapa, deve-

se conectar ao bloco de controle entradas e saídas booleanas externas.

4. **Derivar a máquina de estados finita do bloco de controle.** Por último, deve-se converter a máquina de estados de alto nível para uma máquina de estados finita do bloco de controle. Para isso, é necessário trocar as operações sobre dados por leitura e escrita de sinais de controle.

MODELAGEM BASEADA EM TRANSAÇÕES

Segundo Ghenassia (2005), *Transaction Level Modeling* (TLM) é uma abordagem de modelagem de alto nível onde os detalhes de comunicação entre os módulos são separados dos detalhes de computação dentro de um sistema.

Em uma descrição TLM, os componentes são associados a processos concorrentes que calculam e representam seu comportamento. Esses componentes comunicam-se entre si na forma de transações através de um canal abstrato. Interfaces TLM são implementadas dentro de canais para encapsular os protocolos de comunicação. Para estabelecer uma comunicação, um processo precisa acessar essas interfaces através das portas dos componentes.

TLM define uma transação como uma transferência de dados ou uma sincronização entre dois componentes em um instante determinado pela especificação do *hardware*/*software*. Tais dados podem ser representados em qualquer estrutura de bit, como por exemplo, transferências de meias palavras (*half-words*) entre dois registradores.

Durante o ciclo de projeto de um sistema integrado (SoC), sua descrição TLM atua como referência única para três atividades estratégicas entre diferentes equipes de desenvolvimento:

- desenvolvimento de software;
  - antecipação do desenvolvimento do *software* dependente de *hardware*
  - elaboração de protótipos virtuais
- análise arquitetural;
- verificação funcional.

A modelagem em nível de transações traz diversos benefícios aos desenvolvedores de SoCs, aumentando a produtividade e, conseqüentemente, respeitando os prazos impostos pelo *time-to-market*.

SYSTEMC

SystemC é uma linguagem que estende o padrão C++ através da utilização de bibliotecas de classe (SYSTEMC, 2008). É uma linguagem de projeto e verificação unificada que expressa, na forma de classes C++, atributos arquiteturais, além de outros atributos em nível de sistema. SystemC possibilita o projeto e a verificação em nível de sistema, independentemente da implementação de *hardware* e *software*, assim como a co-verificação com o projeto em nível de transferência entre registradores (RTL). Esse alto nível de abstração permite um ciclo de projeto mais rápido e mais produtivo do que é possível em RTL. Além disso, a verificação de atributos de nível de sistema é extremamente mais veloz que em RTL.

2.3 CARACTERÍSTICAS DAS MEMÓRIAS SRAM

Memórias de acesso randômico são aquelas que possuem um tempo de acesso independente da localização física dos dados. Diferem das memórias de acesso serial, cuja latência é associada à leitura ou à escrita de um dado em particular. Podem ser divididas em dois sub-grupos: as que permitem somente operações de leitura (*Read Only Memory - ROM*) e as que permitem tanto a leitura quanto a escrita de dados (*Read-Write Memory - RWM*). Por razões históricas, o nome RAM tem sido usado como sinônimo de RWM, provavelmente devido à pronúncia complicada do último (RABAEY; CHANDRAKASAN; NIKOLIC, 2003).

Dependendo do estilo de implementação de suas células, as memórias podem ser classificadas em estáticas ou dinâmicas (GAJSKI, 1997). No caso de uma RAM estática (*Static RAM* – SRAM), suas células são construídas com até 6 transistores, utilizando um par de inversores realimentados. Seu conteúdo é mantido enquanto o mesmo não for sobrescrito ou até que haja um corte de energia. Já no caso das RAMs dinâmicas (*Dynamic RAM* – DRAM), cada célula de memória é implementada utilizando somente um transistor. A desvantagem da DRAM é que seu conteúdo é perdido durante uma operação de leitura, devendo assim ser reescrito logo em seguida, o que resulta em perda de desempenho. Apesar do alto custo, as SRAMs são mais apropriadas (quando utilizadas em quantidades moderadas) para uso em sistemas que necessitam de acesso rápido, como por exemplo em memórias cache de processadores.

2.3.1 ARQUITETURA E ORGANIZAÇÃO DE MEMÓRIAS

Ao implementar uma memória de N palavras, cada uma com largura de M bits, a abordagem mais intuitiva é empilhar as palavras de memória subseqüentes linearmente, como mostra a

figura 2 (RABAEY; CHANDRAKASAN; NIKOLIC, 2003).

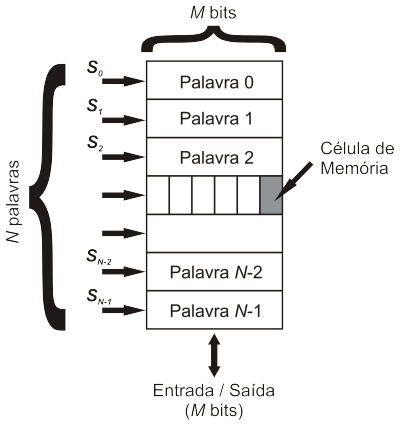

Figura 2: Diagrama pictórico de uma memória RAM NxM

Suponha que se deseja implementar uma memória composta por aproximadamente 1 milhão ($N = 2^{20} = 1.048.576$) de palavras de 8 bits ($M = 8$). Utilizando a estratégia da figura 2, logo se percebe que serão necessários mais de 1 milhão de sinais de seleção – um para cara palavra. Portanto, para evitar essa enorme quantidade de fios e, conseqüentemente, problemas de encapsulamento do chip, utiliza-se um decodificador. A palavra de memória é selecionada por um endereço binário (A_0 a A_{K-1}), conforme mostra a figura 3.

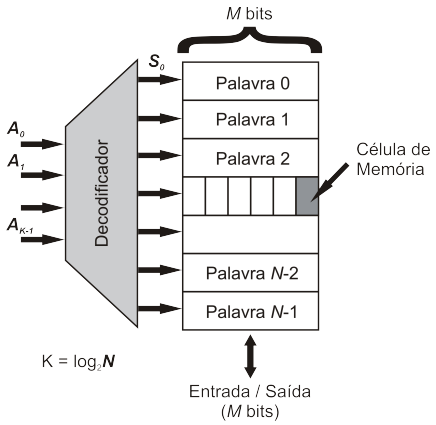

Figura 3: Introdução de decodificador no endereçamento de palavras

O decodificador traduz esse endereço em $N = 2^K$ linhas de seleção, onde somente uma encontra-se ativa por vez. Essa abordagem reduz o número de linhas de endereço de 1 milhão para 20 ($\log_2 2^{20}$), o que praticamente elimina os problemas de encapsulamento.

A figura 4 mostra a estrutura interna de uma RAM. O decodificador recebe o endereço e ativa cada uma das células referentes ao endereço requisitado. O sinal *en* pode desativar o decodificador e prevenir que alguma palavra seja ativada. O sinal de controle de leitura/escrita *rw* também é conectado a cada uma das células para controlar se elas serão escritas com o valor

de *wdata*, ou se seus valores serão lidos e escritos em *rdata*.

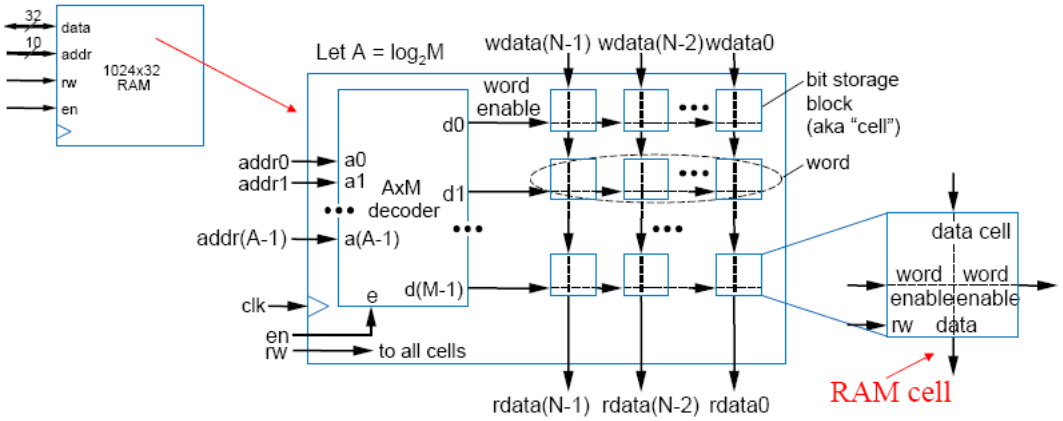

Figura 4: Estrutura interna de uma RAM (VAHID, 2006)

Enquanto essa solução resolve o problema dos sinais de seleção, não contribui para a questão da relação de aspecto da memória. Considere que uma célula de memória ocupe uma área de proporções quadradas. Analisando as dimensões do arranjo de armazenamento do exemplo escolhido, conclui-se que a altura é aproxidamente 128.000 vezes maior que seu comprimento ($2^{20}/2^3$). Isso implica em um elevado tempo de acesso, pois os fios verticais que conectam as células à interface, denominados de linhas de bit (*bit lines*), tornam-se extremamente longos.

Como solução, os arranjos de memória são organizados de modo que as dimensões horizontal e vertical sejam da mesma ordem de magnitude. Diversas palavras são armazenadas em uma mesma linha e são selecionadas simultaneamente. Para rotear a palavra correta aos terminais de entrada/saída é inserido um decodificador de coluna (figura 5). O endereço é então particionado em endereço de coluna (A_0 a A_{K-1}) e endereço de linha (A_K a A_{L-1}).

2.3.2 OPERAÇÃO DE UMA CÉLULA SRAM

A parte principal da estrutura de uma SRAM é o seu conjunto de células. Uma célula individual de 6 transistores está ilustrada na figura 6. Cada célula pode armazenar informação referente a 1 bit, ficando retida no par cruzado de inversores indicado por $M_{p1} - M_{n1}$ e $M_{p2} - M_{n2}$.

Durante uma operação de escrita em uma célula de SRAM, a linha de bit BL (*Bit Line*) é carregada com o valor a ser armazenado e a linha de bit negado $\overline{BL}$ com o valor inverso. Quando a linha de seleção de palavra WL (*Word Line*) for ativada, V_1 e V_2 receberão os valores de BL e $\overline{BL}$, respectivamente. Então, assim que WL for desativada, o dado ficará retido no par cruzado

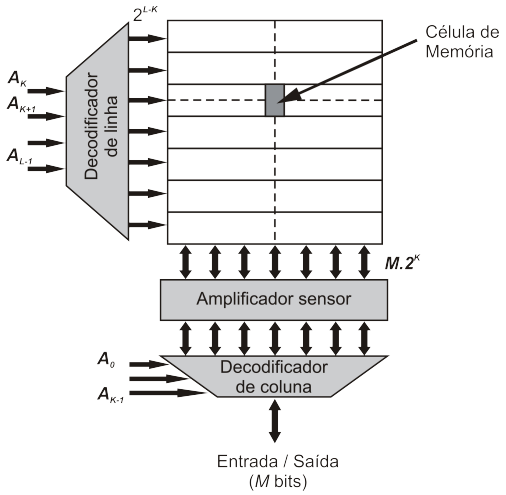

Figura 5: Seleção de uma palavra de memória

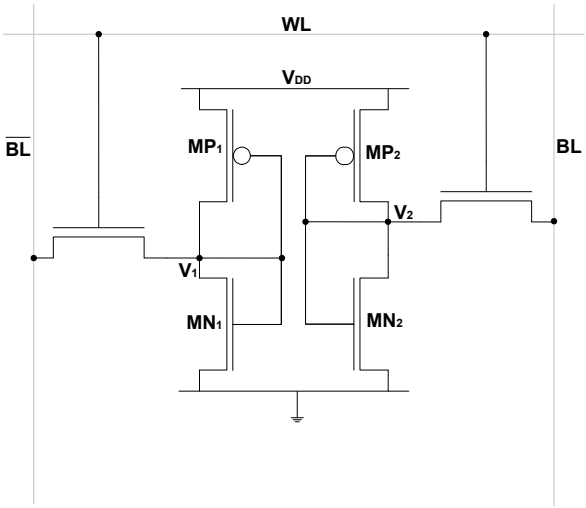

Figura 6: Célula de uma SRAM de 6 transistores

de inversores.

Para uma operação de leitura, as linhas BL e $\overline{BL}$ são pré-carregadas com um valor de tensão intermediário ($V_{DD}/2$, onde V_{DD} é a tensão que corresponde ao nível lógico 1) e, logo em seguida, WL é ativada. Dessa forma, BL e $\overline{BL}$ sofrerão uma leve variação, devido aos valores armazenados em V_1 e V_2 serem contrários entre si. Essa pequena variação, porém, será percebida pelo amplificador sensor (*sense amplifier*), o qual a amplificará e retornará o valor lógico correspondente.

2.4 PROJETO DE MEMÓRIAS EMBARCADAS

No início da computação digital, o foco de pesquisadores era a otimização do tamanho das memórias. O custo era alto, e espaço de memória era um recurso escasso. De acordo com

Benini, Macii e Poncino (2003), o rápido crescimento da tecnologia de semicondutores e o conseqüente aumento no nível de integração mudaram completamente esse cenário. Hoje em dia, o custo por bit de memória é extremamente baixo (tanto para memórias *off-chip* quanto embarcadas), e tamanho raramente é um problema principal. Por outro lado, desempenho e consumo de energia são os principais desafios para o projeto de sistemas. Desta forma, existe atualmente uma demanda por modelos de memória capazes de capturar o comportamento de desempenho e de consumo de energia que possam ser utilizados nos níveis de projeto mais abstratos.

3 TRABALHOS CORRELATOS

As principais motivações para se embutir memória em *chip* são a busca por maior desempenho e a redução no consumo energético. Segundo Marinissen et al. (2005), em 2010, cerca de 90% da área de silício em *chip* será coberta por memória com diferentes funcionalidades. Um exemplo da importância das memórias em sistemas embarcados encontra-se em processadores comerciais de propósito geral. Cerca de 70% da energia consumida por um processador está no fornecimento de dados e instruções, conforme ilustra a figura 7 (DALLY et al., 2008). Contudo, existem alguns desafios e até mesmo algumas desvantagens em relação ao paradigma *multi-chip*. O projeto de memórias torna-se mais complexo, exigindo a introdução de novas metodologias de teste e análise.

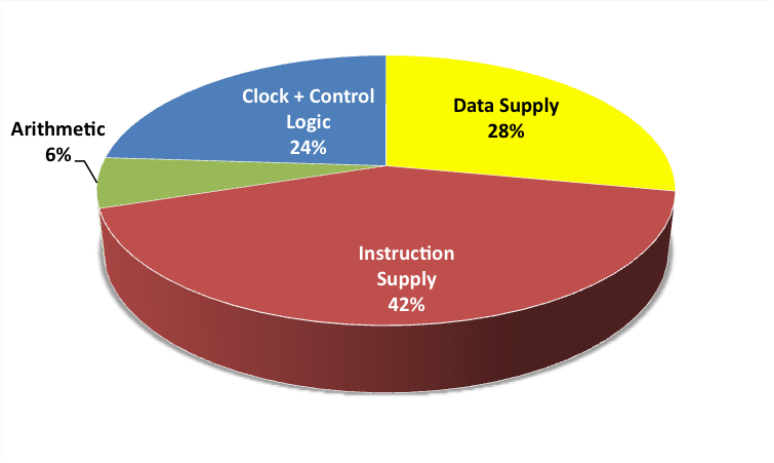

Figura 7: Eficiência energética de um processador embarcado

Wilton e Jouppi (1996) desenvolveram um modelo analítico para o cálculo de diversas métricas em memórias, denominado CACTI. Em sua versão inicial, o CACTI modelava tempos de acesso e de ciclos em caches. Após diversas atualizações, novas funções foram implementadas. Atualmente, também oferece suporte à modelagem de memórias DRAM, além de poder estimar a área e consumo de energia em caches e memória principal. Comparando com um modelo HSpice[1], esta modelagem possui uma margem de erro de apenas 6%. Por se tratar de

[1]Ferramenta para simulação de circuitos desenvolvido por Synopsys, Inc.

um modelo matemático, o modelo não captura o impacto do padrão de acesso de uma determinada aplicação. Porém, o mesmo poderia ser utilizado para caracterizar eletricamente o modelo implementado neste trabalho.

Um dos mais conhecidos e citados simuladores de cache é o Dinero IV (EDLER; HILL, 2007). A principal limitação desse simulador é o fato de que ele não suporta noção de tempo. Isso torna difícil modelar precisamente memórias com diferentes tempos de acesso durante a simulação. A única informação fornecida pelo Dinero é o número de acessos à cache, separado por acertos e faltas. Esforços adicionais seriam necessários para se determinar o comportamento temporizado de toda a hierarquia de memória. Além disso, o Dinero não é um simulador funcional, o que significa que o conteúdo das caches não podem ser considerados. Apesar disso não ser uma desvantagem, examinar o conteúdo da cache pode ajudar a entender o seu comportamento. Dinero suporta somente simulação de caches. Outras arquiteturas, como memórias de rascunho ou scratchpad memories (SPM), não são explicitamente suportadas.

Em Klein et al. (2007), os autores desenvolveram uma extensão para SystemC que instrumenta a linguagem para a caracterização, modelagem e estimativa de energia. Apesar de ser considerada uma das mais promissoras linguagens para modelagem funcional de sistemas, em sua versão original, SystemC não oferece suporte à modelagem de consumo de energia. A PowerSC fornece ao usuário uma maneira simples e transparente de obter informações sobre a atividade de transição, a partir de descrições SystemC RTL. Infelizmente, a biblioteca PowerSC ainda não está instrumentada para a estimativa de consumo de energia em blocos de memória.

Pensando nisso, este trabalho propõe a modelagem de uma SRAM visando a realização de estimativas de consumo de energia com a PowerSC. O trabalho é desenvolvido utilizando uma abordagem *meet-in-the-middle* (MARWEDEL, 2006). O modelo é caracterizado em um fluxo de projeto *top-down*. Primeiramente, se cria uma especificação funcional em um alto nível de abstração com uma interface de comunicação baseada em transações. Em seguida, inicia-se uma especificação mais detalhada do modelo em nível de transferência entre registradores. Simultaneamente à modelagem, é estruturada uma plataforma virtual completa composta por processadores e um sistema de interconexão. Desse modo, analisa-se o comportamento do modelo operando em um sistema real, o que possibilita estimar métricas para determinados casos de uso, como a quantidade de energia consumida por memória durante a execução de uma determinada aplicação. Não é de conhecimento do autor a existência de trabalho similar publicado até então.

4 MODELAGEM DO BLOCO DE MEMÓRIA

Este capítulo descreve o processo de modelagem da memória. O desenvolvimento teve início com a criação de um modelo funcional da memória, visando analisar apenas seu comportamento. Com isso, foi possível obter alguns resultados de referência. Para se atingir um maior grau de detalhamento, o modelo passou a ser caracterizado em nível de transferências entre registradores. Sua estrutura interna foi descrita em diversos módulos, conforme representado no diagrama de blocos da Figura 8.

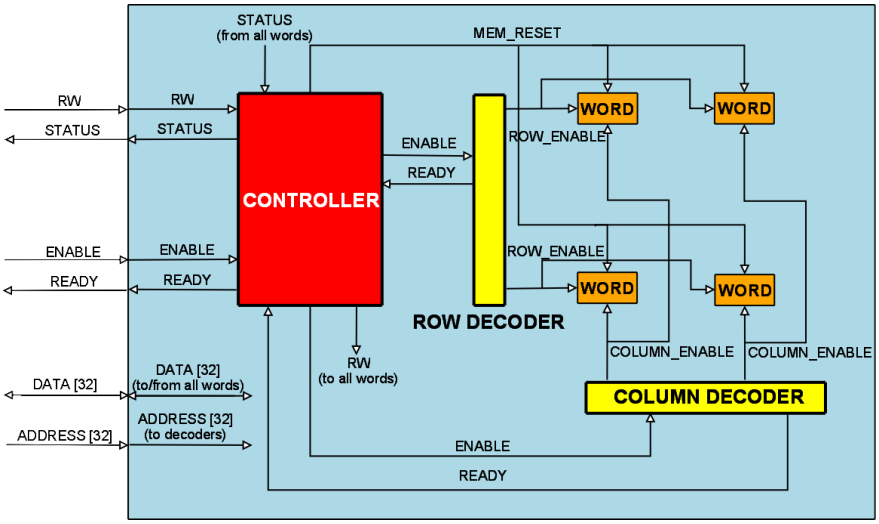

Figura 8: Diagrama de blocos RTL do modelo

A memória está modelada como um conjunto de palavras de 32 bits cada. Estas palavras estão organizadas em uma estrutura M por N, cujo aspecto pode ser facilmente reconfigurado. Devido à característica de indexação por palavra, pode-se afirmar que a memória possui uma estrutura tridimensional.

O modelo desenvolvido é sincronizado pelo sinal de relógio. Possui uma porta bidirecional de dados, ou seja, por uma mesma porta são realizadas tanto escrita como leitura de dados. As linhas de dados e de endereço possuem largura de 32 bits. Além disso, também existem 4 portas para sinais de controle:

- ENABLE: Quando ativado, habilita o modelo de memória.
- READY: Notifica o término das operações.
- RW: Define o modo de operação da memória.
  - 0: Leitura
  - 1: Escrita
- status: Reporta se a operação foi bem-sucedida ou não.
  - 0: Falha
  - 1: Sucesso

4.1 CRIAÇÃO DO MODELO FUNCIONAL

No nível funcional, a unidade de armazenamento de dados foi modelada utilizando *arrays*, uma estrutura de dados composta por um conjunto de elementos acessados por indexação. Os elementos são do tipo *char*, com tamanho equivalente a 1 byte.

Nesse alto nível de abstração, a memória possui uma relação de aspecto linear. A grosso modo, todas as células estão organizadas lado-a-lado em uma única linha. Essa decisão de projeto viabiliza a análise focada apenas no comportamento funcional do modelo.

Uma operação de acesso à memória é resolvida de modo trivial. Para uma operação de leitura, o endereço recebido aponta diretamente para a posição do primeiro *byte* da palavra a ser acessada. Então, a partir do local apontado, os próximos quatro bytes são escritos na porta de dados, finalizando a leitura.

4.2 DESCRIÇÃO RTL DO MODELO

Concluídas as análises sobre o modelo funcional, passou-se ao desenvolvimento de uma modelagem da memória com maior grau de detalhamento, modelando-se individualmente seus principais componentes, tais como os dois decodificadores e a unidade de controle. Além disso, a representação do armazenamento de dados foi modelada de maneira mais fidedigna.

4.2.1 BLOCO DE CONTROLE

A unidade de controle da SRAM atua como uma máquina de estados sincronizada por um sinal de relógio. É composta de 4 estados, conforme mostra a figura 9. Sua operação tem início no estado IDLE. Nesse estado, seus sinais de entrada são constantemente verificados. Assim que o sinal de habilitação ficar ativo, a máquina avança para o estado de decodificação, onde os decodificadores de linha e coluna serão habilitados. A máquina então permanece nesse estado até que os decodificadores terminem suas operações. No próximo estado, BEGIN_OP, o controlador desabilita os decodificadores, e aguarda o término da operação de escrita ou leitura. Por fim, no último estado o bloco de memória é reinicializado, e o controlador ativa o sinal READY, confirmando o término das operações.

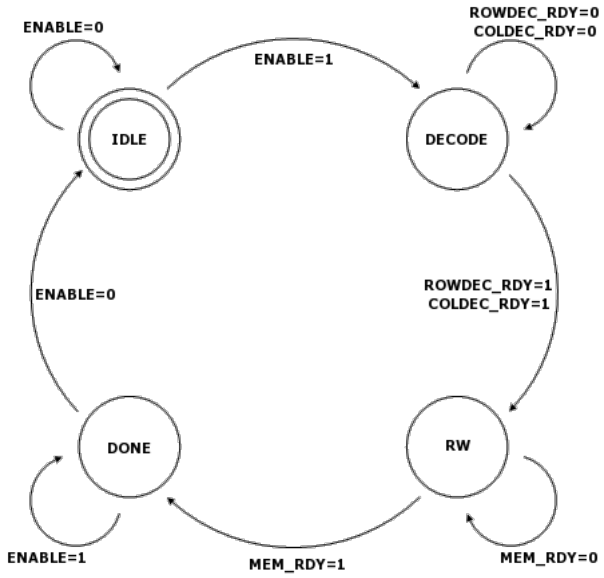

Figura 9: Máquina de estados do bloco de controle

4.2.2 DECODIFICADORES

Existem dois decodificadores, sendo um para a habilitação de uma linha e outro para a habilitação de uma coluna. Os decodificadores realizam a indexação das palavras através de operações lógicas e aritméticas sobre o endereço. Considerando que M seja a quantidade de palavras por linha, as seguintes operações são realizadas para indexação:

- Linha: Endereço $>>$ M + 2.
- Coluna: (Endereço $>>$ 2) & M.

Em ambos os casos é feito um deslocamento à esquerda de 2 bits, pois o endereçamento feito pelo processador é orientado a bytes. Como a indexação interna da memória é orientada a palavras de 32 bits, é necessário dividir o endereço por 4.

O processo de decodificação leva 2 ciclos para ser completado. A máquina de estados dos decodificadores é mostrada na figura 10. Assim que eles são habilitados, a decodificação já é realizada no estado seguinte. No próximo ciclo, os módulos aguardam sua desativação no estado DONE. Também foi definido um estado adicional para a reinicialização dos módulos.

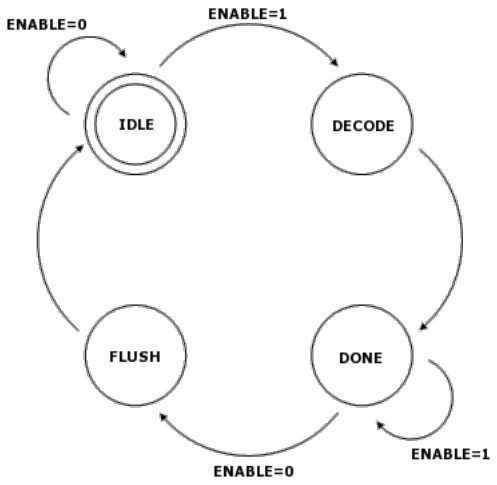

Figura 10: Máquina de estados dos decodificadores

4.2.3 ARMAZENAMENTO DE DADOS

Cada um dos módulos palavra representa um bloco de 32 bits e possui portas de dados, endereço e controle. Internamente, os dados são armazenados em 4 vetores de 8 bits cada, ou seja, 4 *bytes* de memória.

As palavras de memória foram modeladas como dispositivos assíncronos. Teoricamente, uma palavra apenas iniciará sua operação quando receber a ativação simultânea das linhas de seleção de linha e coluna. Porém, a implementação foi realizada de uma maneira um pouco diferente. Cada módulo executa uma função assim que o mesmo percebe a ativação da saída do decodificador de linha. Portanto, todas as palavras de uma linha da memória são ativadas por ciclo. A função chamada, porém, logo de início verifica se a saída do decodificador de coluna também está ativada. Desse modo, apenas o módulo selecionado realiza a operação por completo.

Também foi considerada uma implementação síncrona de uma palavra. Apesar de apresentar comportamento semelhante, o consumo de processamento se tornaria significativamente maior. Como dispositivos síncronos são sensíveis à transição de relógio, todas as palavras seriam ativadas a cada ciclo, e cada uma delas realizaria uma constante verificação de suas entradas. Considerando que uma memória de 32kB necessita que 8192 módulos de palavra sejam instanciados, o processador da máquina hospedeira ficaria sobrecarregado, resultando em um alto tempo de simulação.

5 METODOLOGIA E INFRA-ESTRUTURA

Aqui será descrita a infra-estrutura utilizada para realizar a validação do modelo desenvolvido. Além disso, serão mostradas as adaptações realizadas no processo de inclusão do bloco de memória.

5.1 METODOLOGIA DE PROJETO E VALIDAÇÃO

Depois que se obtém a descrição de um componente de *hardware*, o passo seguinte – independentemente do nível da descrição – é verificar se as funcionalidades e requisitos estão de acordo com o esperado. Gerar manualmente um conjunto de estímulos e resultados - para verificar a corretude do modelo - é extremamente trabalhoso e pouco produtivo, além de bastante propício a erros.

Para lidar com esse problema, será utilizado o conceito de *golden model*. Um *golden model* é um modelo que, garantidamente, possui as funcionalidades e cumpre os requisitos previamente especificados.

Portanto, no projeto de um componente de *hardware*, é útil partir de uma descrição algorítmica não temporizada – que modela funcionalidade. Essa descrição servirá, posteriormente, como *golden model* para a verificação funcional de modelos descritos em outros níveis de abstração, como TLM e RTL.

5.2 UTILIZAÇÃO DE UMA PLATAFORMA VIRTUAL

Segundo Ghenassia (2005), o primeiro passo para um fluxo de projeto e verificação em nível de sistema é o uso de modelos cuja comunicação seja baseada em transações (TLM) – uma vez que essa abordagem propicia a reusabilidade.

Para facilitar o reuso, deve ser fornecida uma infra-estrutura responsável por organizar o código-fonte e gerenciar diferentes tipos de componentes. É recomendável que cada projeto

seja organizado por uma hierarquia definida por diretórios, de modo que componentes de tipos diferentes sejam alocados em diretórios diferentes.

Neste trabalho foi usada como base para a plataforma a infra-estrutura fornecida pela *ArchC Reference Platform* (ARP)[1].

5.2.1 ARCHC REFERENCE PLATFORM

A ARP é um *framework* para auxiliar a construção de modelos de plataformas que usam simuladores ArchC 2.0[2] (AZEVEDO et al., 2005). Para tanto, a ARP provê uma estrutura básica para a rápida conexão dos componentes mais comuns em modelos heterogêneos de plataforma.

Um bloco de propriedade de intelectual (*IP – Intelectual Property*) descrito em SystemC pode ser conectado à ARP possibilitando, então, a análise de sua interação com outros componentes (processadores, memórias, outros IPs).

5.3 MODELAGEM DAS TRANSAÇÕES

Para estabelecer a comunicação do modelo com o barramento da plataforma, foi necessário implementar alguns módulos intermediadores, conforme ilustra a figura 11.

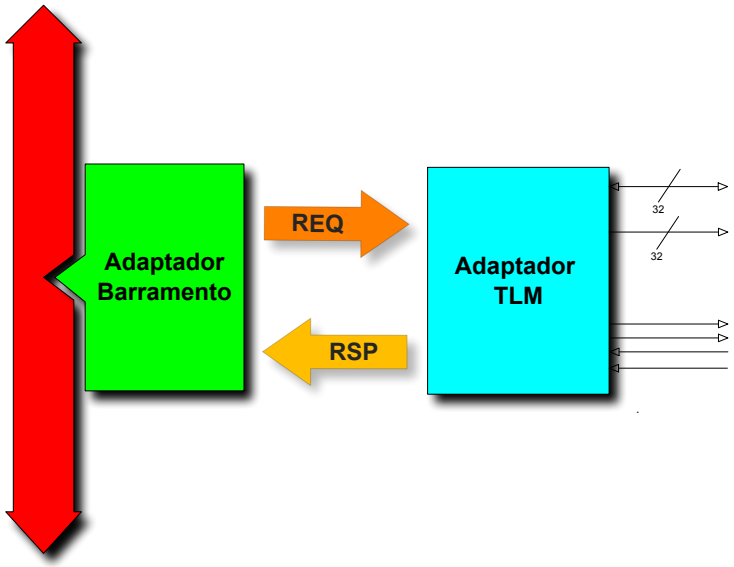

Figura 11: Diagrama de blocos da plataforma virtual

O primeiro módulo, denominado “adaptador TLM”, foi implementado como um adaptador

[1]Mais informações em `http://www.archc.org` na seção *Platforms*.
[2]Mais informações em: `http://www.archc.org/`

do barramento. O barramento realiza operações de leitura e escrita chamando as funções accept_read() (figura 12) e accept_write() (figura 13), respectivamente, implementadas pelo adaptador. Por meio dessas funções, "o adaptador TLM" se comunica com o outro módulo, "adaptador RTL". Esse, então, captura os dados das requisições, os envia ao modelo através de sinais, e aguarda por uma resposta. Assim que a resposta for obtida, o "adaptador RTL" gera um pacote de resposta e o envia devolta ao adaptador do barramento.

```
status accept_read(device_id_t dev, addr_t addr, data_t& d, int p)
{
.       if ((addr < 0) || (addr > memory_size) || (dev != _device_id))
.       {
.       .       return noc_specific::ERROR;
.       }

.       req.op = 0;
.       req.address = addr;

.       rsp = target_port->transport(req);

.       if (rsp.status != 1)
.       {
.       .       return noc_specific::ERROR;
.       }

.       d = rsp.data;

.       return noc_specific::OK;
}
```

Figura 12: Função de leitura do adaptador do barramento

```
status accept_write(device_id_t dev, addr_t addr, data_t d, int p)
{
.       if ((addr < 0) || (addr > memory_size) || (dev != _device_id))
.       {
.       .       return noc_specific::ERROR;
.       }

.       req.op = 1;
.       req.data = d;
.       req.address = addr;

.       rsp = target_port->transport(req);

.       if (rsp.status != 1)
.       {
.       .       return noc_specific::ERROR;
.       }

.       return noc_specific::OK;
}
```

Figura 13: Função de escrita do adaptador do barramento

A função accept_read() gera um pacote de requisição com o endereço a ser acessado na memória e com o campo de operação setado em 0. Em seguida, envia a requisição ao segundo adaptador e aguarda pelo pacote de resposta com os dados da memória. Assim que a resposta for recebida, caso a operação tenha finalizado com sucesso, os dados são escritos devolta no barramento. Caso contrário, a mesma é invalidada.

A função accept_write() procede de maneira semelhante. A diferença está apenas no conteúdo dos pacotes. A requisição possui o campo de operação setado em 1, os dados a serem a escritos na memória e o endereço de escrita. Já o pacote de resposta vem apenas com o valor que estado da operação.

5.4 DESCRIÇÃO DO ESTUDO DE CASO

Para a validação do modelo, foi desenvolvida uma plataforma capaz de executar diversos programas de *benchmark*, ilustrada na figura 14.

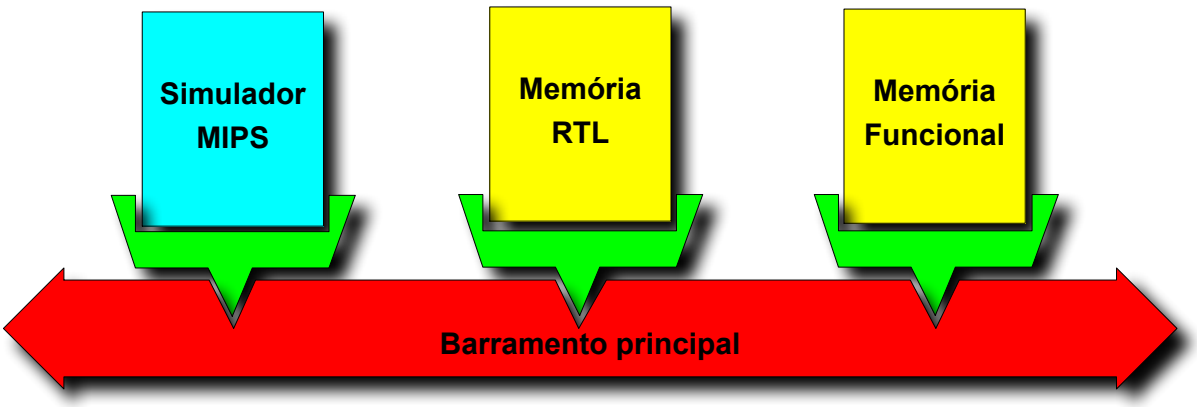

Figura 14: Diagrama de blocos da plataforma virtual

A plataforma é composta por um processador MIPS (cujo simulador foi obtido a partir de um modelo descrito em ArchC) e pelos modelos de memória desenvolvidos. Além disso, existe um barramento interconectando todos os módulos. Os programas de *benchmark*, após compilados para o processador alvo, são carregados na memória da plataforma. Como a descrição do processador não suporta a interface TLM do barramento, foi necessária a utilização de um adaptador que faz a conversão das transações.

Para validar a plataforma com modelo de memória funcional, esse foi conectado ao barramento abrangendo a faixa de endereços com início em 0x000000 e término em 0x9FFFFF.

A validação do modelo RTL foi realizada de maneira híbrida. Foram conectados dois blocos de memória: um modelo RTL com tamanho de 32 kB, com relação de aspecto quadrada (palavras organizadas em 512 linhas e 16 colunas, cada palavra possuindo tamanho de 32 bits), ocupando a faixa de 0x000000 a 0x007FFF; e um modelo funcional abrangendo o restante do espaço de endereçamento até 0x9FFFFF. Nessa configuração da plataforma, pode-se inferir que o processador e o modelo RTL estão conectados ao barramento da forma de um SoC, e a

memória funcional representa uma memória RAM em um nível mais baixo da hierarquia.

Foram escolhidos 4 programas do pacote de benchmarks MiBench:

- cjpeg, com o arquivo input_small.ppm como estímulo,
- sha, com o arquivo input_small.asc como estímulo,
- susan, com o arquivo input_small.pgm como estímulo,
- bitcount, com o parâmetro 75000 como estímulo.

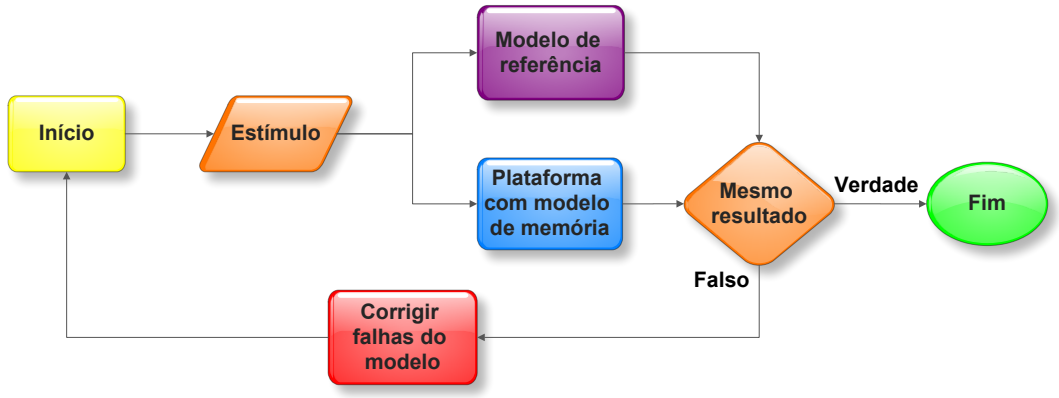

Figura 15: Metodologia de validação utilizada

A figura 15 mostra o fluxo de execução durante o processo de validação. Estímulos são utilizados como entradas para o *software* executando numa plataforma de referência previamente validada. Em seguida, o mesmo arquivo é dado como entrada para a plataforma com o modelo de memória. Então, as saídas das duas execuções são comparadas. Se ambas apresentarem o mesmo resultado, a plataforma em teste está validada. Caso contrário, são aplicadas as devidas correções na plataforma para a próxima validação.

6 RESULTADOS EXPERIMENTAIS

6.1 AMBIENTE DE SIMULAÇÃO

O computador utilizado para simular a operação do modelo de memória possui a seguinte configuração:

- Processador: Intel(R) Core(TM)2 Duo E6550 operando a uma freqüência de 2.33GHz, cache L2 de 4 MB e Front Side Bus de 1333MHz.
- Memória principal: 2 GB de RAM
- Sistema Operacional: Ubuntu Linux 32 bits, kernel 2.6.24

As ferramentas utilizadas durante o processo e suas respectivas versões estão listadas abaixo:

- ***ArchC*** versão 2.0
- ***SystemC*** versão 2.1.v1
- ***SystemC TLM*** versão 2.0 (2008-06-09)
- ***Compilador GCC*** versão 4.2.3
- ***Compilador cruzado GCC para MIPS*** versão 3.3.1

6.2 ANÁLISE COMPARATIVA

As análises das simulações foram realizadas em duas etapas. Primeiramente foi realizada a validação da plataforma com o modelo de memória funcional. Os tempos de cada simulação estão organizados na tabela 1. Em seguida, a memória RTL foi incluída na plataforma para mais uma série de simulações. Foi estipulado um tamanho de 32 KB para o modelo RTL, organizados em uma malha com relação de aspecto quadrada (512x512 bits). Além dos tempos

de simulação (tabela 2), também foi coletado o consumo de RAM da máquina para diversas configurações do modelo (tabela 3).

Tabela 1: Tempo de simulação para a validação do modelo funcional

	Tempo em segundos		
Benchmark	Usuário	Sistema	Real
SHA	13,87	0,03	14,54
susan	4,19	0,04	4,44
cjpeg	14,25	0,08	14,98
Bitcount	44,78	0,12	45,04

Tabela 2: Tempo de simulação para a validação do modelo misto

	Tempo em segundos		
Benchmark	Usuário	Sistema	Real
SHA	81,48	0,22	81,41
susan	72,81	0,22	72,74
cjpeg	77,28	0,28	77,26
Bitcount	124,24	0,24	124,19

É notável a grande diferença nos tempos de simulação. A plataforma com modelo de memória RTL tem execuções mais demoradas, porém isso é um custo a se arcar em conseqüência de uma especificação em nível mais baixo. Conforme anteriormente explicado, a implementação em RTL possui diversas unidades modeladas como máquinas seqüenciais. Ainda que as mesmas estejam desabilitadas, continuam a consumir tempo de processamento, pois precisam verificar constantemente suas entradas. Outro ponto a se considerar é o fato de SystemC não utilizar *threads* nativas do sistema operacional, não aproveitando as arquiteturas de vários núcleos de processadores modernos.

Tabela 3: Consumo de memória da máquina hospedeira em simulação mista

Tamanho simulado kbytes	Memória da máquina mbytes
32	56
64	107
128	212
256	424
512	846
1024	1694

Na tabela 3, pode-se ver a quantidade de RAM utilizada pelas simulações, separadas pelo tamanho de memória simulada. A coleta foi realizada utilizando o programa htop[1]. Observando

[1]Mais informações em `http://htop.sourceforge.net/`

a tabela acima, pode-se inferir que o consumo de memória cresce na mesma proporção do tamanho de memória a ser simulada. Isso ocorre devido ao grande número de objetos instanciados no modelo. Cada palavra de dados modelada está representada por uma classe. No modelo de 32 kB, por exemplo, existem 8192 módulos de palavra de memória.

7 *CONCLUSÕES*

Considerando-se um fluxo de projeto convencional de sistemas eletrônicos, a estimativa de consumo de energia em nível de transistores é a mais precisa. Entretanto, devido à pressão exercida pelo mercado de produtos de eletrônica de consumo, os projetos de sistemas embarcados iniciam em um níveis mais altos de abstração, focando não em componentes isolados, mas sim em sistemas completos, com *hardware* e *software*. Por outro lado, considerando o uso de blocos de hardware pré-existentes (blocos IP), o nível RT tende a ser o nível de menor abstração dentro do fluxo de projeto e, portanto, aquele que pode proporcionar os resultados mais precisos em termos de validação de desempenho e de consumo de energia.

A metodologia de desenvolvimento adotada neste trabalho não só contribuiu para seu rápido desenvolvimento, como também possibilita a sua fácil extensão. Abstraindo o protocolo de comunicação entre os componentes por meio da utilização da metodologia TLM, o desenvolvedor economiza esforços para integrar o modelo na plataforma virtual, antecipando a etapa de testes. Além disso, facilita a reutilização do modelo em outros sistemas, contribuindo para o projeto de sistemas embarcados.

A partir dos resultados obtidos para a validação do método aqui proposto, identificou-se uma limitação no tamanho do bloco de memória que se pode modelar no nível RT. Essa limitação está diretamente relacionado à grande quantidade de memória requerida para realizar a simulação do sistema inteiro em SystemC. Outro fator que influencia essa limitação é o tempo necessário para simular a operação do sistema, pois SystemC não aproveita o paralelismo oferecido pelas arquiteturas dos processadores modernos.

7.1 TRABALHOS FUTUROS

O desenvolvimento deste trabalho proporciona alguns desdobramentos para futuros trabalhos. O alto grau de detalhamento do modelo viabiliza sua extensão para otimizações ou novas metodologias de análise.

A descrição dos componentes internos do modelo foi feita focada na simulação do comportamento de uma memória SRAM real. Através da implementação de um controle de tempos de acesso, é possível otimizar as operações do modelo para se obter um melhor desempenho.

Além disso, através do cálculo de transições em uma simulação, é possível realizar uma estimativa de consumo de energia de determinada execução. Com a integração da biblioteca PowerSC (KLEIN et al., 2007), a atividade de transição pode ser capturada para estimar a potência consumida pelo modelo. Atualmente, as ferramentas comerciais existentes no mercado não são capazes simular comportamento e consumo de energia simultaneamente.

Devido à crescente preocupação em relação às falhas transientes em dispositivos de memória, existe também a possibilidade de implementação de técnicas de injeção de falhas no modelo. Assim, faz-se uma modelagem de funcionamento para permitir a análise da suscetibilidade do modelo a falhas transientes.

REFERÊNCIAS

AZEVEDO, R. et al. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, Kluwer Academic Publishers, v. 33, n. 5, p. 453–484, October 2005. ISSN 0885-7458.

BAILEY, B.; MARTIN, G.; PIZIALI, A. *ESL Design and Verification*. 1st. ed. [S.l.]: Morgan Kaufmann Publishers, 2007.

BAUMANN, R. C. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY*, 2005.

BENINI, L.; MACII, A.; PONCINO, M. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *Trans. on Embedded Computing Sys.*, ACM, New York, NY, USA, v. 2, n. 1, p. 5–32, 2003. ISSN 1539-9087.

DALLY, W. J. et al. Efficient Embedded Computing. *Computer*, IEEE Computer Society, Los Alamitos, CA, USA, v. 41, n. 7, p. 27–32, 2008. ISSN 0018-9162.

EDLER, J.; HILL, M. D. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. 19 dez. 2007 2007. Http://www.cs.wisc.edu/ markhill/DineroIV.

GAJSKI, D. D. *Principles of Digital Design*. 1st. ed. [S.l.]: Prentice-Hall, 1997. 284-292 p.

GHENASSIA, F. *Transaction-Level Modeling With SystemC: TLM Concepts and Applications for Embedded Systems*. 1st. ed. [S.l.]: Springer, 2005.

GRöTKER, T. et al. *System Design with SystemC*. 1st. ed. [S.l.]: Kluwer Academic Publishers, 2002.

KLEIN, F. et al. An Efficient Framework for High-Level Power Exploration. *Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on*, p. 1046–1049, Aug. 2007. ISSN 1548-3746.

LEãO, R. de O. *Análise Experimental de Técnicas de Estimativa de Potência baseadas em Macromodelagem em Nível RT*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, may 2008.

MARINISSEN, E. J. et al. Challenges in Embedded Memory Design and Test. In: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005. p. 722–727. ISBN 0-7695-2288-2.

MARWEDEL, P. *Embedded System Design*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 1402076908.

RABAEY, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. *Digital Integrated Circuits: A Design Perspective*. 2nd. ed. [S.l.]: Prentice-Hall, 2003. 623-719 p.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and Challenges for Platform-Based Design. In: *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM, 2004. p. 409–414. ISBN 1-58113-828-8.

SYSTEMC. *Open SystemC Initiative*. Oct 2008. Disponível em: <http://www.systemc.org>.

VAHID, F. *Digital Design*. 1st. ed. [S.l.]: John Wiley and Sons, Inc., 2006.

WILTON, S.; JOUPPI, N. CACTI: An Enhanced Cache Access and Cycle Time Model. *Solid-State Circuits, IEEE Journal of*, v. 31, n. 5, p. 677–688, May 1996. ISSN 0018-9200.

APÊNDICE A – MODELAGEM TLM

A.1 MODELO DE MEMÓRIA FUNCIONAL

```
/*********************************************************/
/* TLM MEM_TLM Model                                     */
/* Author: Gustavo Henrique Nihei                        */
/* System Design Automation Lab (LAPS)                   */
/* INF-UFSC                                              */
/* http://www.laps.inf.ufsc.br                           */
/*********************************************************/

#ifndef MEM_TLM_H
#define MEM_TLM_H

#include ''systemc.h''
#include ''tlm.h''
#include ''protocol.h''

namespace memory_blk
{

class MEM_TLM:
            public sc_module,
            public tlm::tlm_transport_if< protocol::REQ, protocol::RSP >
{
public:
    MEM_TLM(sc_module_name name, unsigned int size) :
            sc_module(name),
            _size(size)
    {
        MEM = new char [_size];
        for (unsigned int i = 0; i < _size; ++i)
            MEM[i] = 0;
    }

    ~MEM_TLM();

    protocol::RSP transport(const protocol::REQ &req);
    int read(unsigned int address);
    int write(unsigned int address);
```

```
private:
    char * MEM;

    typedef union
    {
        int data;
        unsigned char byte[sizeof(int)];
    } U;

    unsigned int _size;
    U aux;
};

inline MEM_TLM::~MEM_TLM()
{
    if (MEM) delete [] MEM;
    MEM = (char *)0;
}

protocol::RSP MEM_TLM::transport(const protocol::REQ &req)
{
    protocol::RSP rsp;

    // op == 1 => WRITE
    // op == 0 => READ
    if (req.op)
    {
        aux.data = req.data;
        rsp.status = write(req.address);
    }
    else
    {
        rsp.status = read(req.address);
        rsp.data = aux.data;
    }

    return rsp;
}

inline int MEM_TLM::read(unsigned int address)
{
    aux.byte[0] = MEM[address];
    aux.byte[1] = MEM[address+1];
    aux.byte[2] = MEM[address+2];
    aux.byte[3] = MEM[address+3];

    return 1;
}

inline int MEM_TLM::write(unsigned int address)
{
```

```
    MEM[address] = aux.byte[0];
    MEM[address+1] = aux.byte[1];
    MEM[address+2] = aux.byte[2];
    MEM[address+3] = aux.byte[3];

    return 1;
}
};

#endif
```

A.2 MODELO DE MEMÓRIA RTL

A.2.1 SRAM_MODEL.H

```
/*****************************************************/
/* SRAM RTL Memory Model                             */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/

#ifndef SRAM_RTL_MODEL_H
#define SRAM_RTL_MODEL_H

#include ''systemc.h''
#include ''mem_word.h''
#include ''row_decoder.h''
#include ''column_decoder.h''
#include ''controller.h''

namespace memory_blk
{
SC_MODULE(SRAM_MODEL)
{
    SC_HAS_PROCESS(SRAM_MODEL);

    unsigned int _rows, _columns;

    /*************************************
        Ports Declaration
    *************************************/
    sc_in_clk clock;
    sc_in<bool> ENABLE;
    sc_out<bool> READY;
    sc_in<bool> rw;
    sc_inout<sc_bv<32> > data;
```

```
sc_in<unsigned int> address;
sc_out<bool> status;

/************************************
    Signal Declaration
************************************/

// TLM Wrapper to Controller signal lines
sc_signal<bool> WRAPPER_ENABLE_MEM;
sc_signal<bool> WRAPPER_READY_MEM;
sc_signal<bool> WRAPPER_RW_MEM;
sc_signal<int> WRAPPER_DATA_MEM;
sc_signal<unsigned int> WRAPPER_ADDRESS_MEM;
sc_signal<bool> WRAPPER_STATUS_MEM;

// Controller signal lines
sc_signal<bool> row_decoder_enable_line;
sc_signal<bool> column_decoder_enable_line;
sc_signal<bool> row_decoder_ready_line;
sc_signal<bool> column_decoder_ready_line;
sc_signal<bool> mem_rw_line;
sc_signal<bool> mem_ready_line;
sc_signal<bool> mem_status_line;
sc_signal<bool> mem_reset_line;

// Decoders to memory block signal lines
sc_signal<bool> row_enable_line[512];
sc_signal<bool> column_enable_line[16];

/************************************
    Modules Declaration
************************************/
CONTROLLER controller;
ROW_DECODER row_decoder;
COLUMN_DECODER column_decoder;

MEM_WORD *word[8192];

SRAM_MODEL(sc_module_name name, unsigned int rows, unsigned int columns) :
        sc_module(name),
        _rows(rows),
        _columns(columns),
        controller(``controller''),
        row_decoder(``row_decoder''),
        column_decoder(``column_decoder'')
{
    char nome[10];

    for (int i = 0; i < _rows; ++i)
    {
        for (int j = 0; j < _columns; ++j)
```

```
    {
        sprintf(nome, ``%c_%d_%d'', ``word'', i, j);
        word[i*_columns+j] = new MEM_WORD(nome);
    }
}

/************************************
    Port Connection
************************************/

// Controller ports
controller.ENABLE(ENABLE);
controller.READY(READY);
controller.clock(clock);
controller.rw(rw);
controller.status(status);
controller.ROW_DECODER_ENABLE(row_decoder_enable_line);
controller.COLUMN_DECODER_ENABLE(column_decoder_enable_line);
controller.ROW_DECODER_READY(row_decoder_ready_line);
controller.COLUMN_DECODER_READY(column_decoder_ready_line);
controller.mem_rw(mem_rw_line);
controller.MEM_READY(mem_ready_line);
controller.MEM_RESET(mem_reset_line);
controller.mem_status(mem_status_line);

// Row Decoder Ports
row_decoder.clock(clock);
row_decoder.ENABLE(row_decoder_enable_line);
row_decoder.READY(row_decoder_ready_line);
row_decoder.address(address);
for (int i = 0; i < _rows; ++i)
    row_decoder.row_enable[i](row_enable_line[i]);

// Column Decoder Ports
column_decoder.clock(clock);
column_decoder.ENABLE(column_decoder_enable_line);
column_decoder.READY(column_decoder_ready_line);
column_decoder.address(address);
for (int i = 0; i < _columns; ++i)
    column_decoder.column_enable[i](column_enable_line[i]);

for (int i = 0; i < _rows; ++i)
{
    for (int j = 0; j < _columns; ++j)
    {
        word[i*_columns+j]->ROW_ENABLE(row_enable_line[i]);
        word[i*_columns+j]->COLUMN_ENABLE(column_enable_line[j]);
        word[i*_columns+j]->READY(mem_ready_line);
        word[i*_columns+j]->RESET(mem_reset_line);
        word[i*_columns+j]->rw(mem_rw_line);
        word[i*_columns+j]->data(data);
```

```
                word[i*_columns+j]->status(mem_status_line);
            }
        }
    }
};
};

#endif
```

A.2.2 CONTROLLER.H

```
/*****************************************************/
/* RTL Memory Controller                             */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/


#ifndef CONTROLLER_RTL_H
#define CONTROLLER_RTL_H

#include ''systemc.h''

namespace memory_blk
{
SC_MODULE(CONTROLLER)
{
    // External communication
    sc_in_clk clock;
    sc_in<bool> ENABLE;
    sc_out<bool> READY;
    sc_in<bool> rw;
    sc_out<bool> status;

    // Decoder Ports
    sc_out<bool> ROW_DECODER_ENABLE;
    sc_out<bool> COLUMN_DECODER_ENABLE;
    sc_in<bool> ROW_DECODER_READY;
    sc_in<bool> COLUMN_DECODER_READY;

    // Memory Block Ports
    sc_out<bool> mem_rw;
    sc_in<bool> MEM_READY;
    sc_out<bool> MEM_RESET;
    sc_in<bool> mem_status;

    // Signal Declarations
```

```
sc_signal<int> STATE;

SC_CTOR(CONTROLLER)
{
    SC_METHOD(Run);
    sensitive << clock.pos();
}

void Run()
{
    switch (STATE)
    {
    case (IDLE):
    {
        READY = false;

        if (ENABLE)
            STATE = DECODE;
        else
            STATE = IDLE;

        mem_rw = rw;

        break;
    }

    case (DECODE):
    {
        READY = false;
        ROW_DECODER_ENABLE = true;
        COLUMN_DECODER_ENABLE = true;

        if (ROW_DECODER_READY & COLUMN_DECODER_READY)
        {
            STATE = BEGIN_OP;
        }
        else
            STATE = DECODE;

        break;
    }

    case (BEGIN_OP):
    {
        READY = false;
        ROW_DECODER_ENABLE = false;
        COLUMN_DECODER_ENABLE = false;

        if (MEM_READY)
        {
            READY = true;
```

```
                STATE = DONE;
            }
            else
                STATE = BEGIN_OP;

            status = mem_status;

            break;
        }

        case (DONE):
        {
            if (ENABLE)
                STATE = DONE;
            else
            {
                STATE = IDLE;
                READY = false;
                MEM_RESET = (MEM_RESET?false:true);

            }

            break;
        }
        }
    }

    enum
    {
        IDLE, DECODE, BEGIN_OP, DONE
    };
};
};

#endif
```

A.2.3 COLUMN_DECODER.H

```
/*********************************************************/
/* RTL Memory Column Decoder                             */
/* Author: Gustavo Henrique Nihei                        */
/* System Design Automation Lab (LAPS)                   */
/* INF-UFSC                                              */
/* http://www.laps.inf.ufsc.br                           */
/*********************************************************/

#ifndef COLUMN_DECODER_RTL_H
#define COLUMN_DECODER_RTL_H
```

```
#include ‘‘systemc.h’’

namespace memory_blk
{
SC_MODULE(COLUMN_DECODER)
{
    sc_in_clk clock;

    sc_in<bool> ENABLE;
    sc_out<bool> READY;

    sc_in<unsigned int> address;
    sc_out<bool> column_enable[16];

    sc_signal<int> STATE;
    sc_signal<unsigned int> aux_addr;
    int aux_out;

    SC_CTOR(COLUMN_DECODER)
    {
        SC_METHOD(Run);
        sensitive << clock.pos();
    }

    void Run()
    {
        switch (STATE)
        {
        case (IDLE):
        {
            READY = false;
            if (ENABLE)
                STATE = DECODE;
            else
                STATE = IDLE;

            aux_addr = address;

            break;
        }

        case (DECODE):
        {
            READY = false;
            STATE = DONE;

            aux_out = (aux_addr >> 2) & 0xF;
            column_enable[aux_out] = true;

            break;
        }
```

```
        case (DONE):
        {
            READY = true;

            if (ENABLE)
                STATE = DONE;
            else
                STATE = FLUSH;

            break;
        }

        case (FLUSH):
        {
            READY = false;

            STATE = IDLE;

            column_enable[aux_out] = false;
        }
        }
    }

    enum
    {
        IDLE, DECODE, DONE, FLUSH
    };
};
};

#endif
```

A.2.4 ROW_DECODER.H

```
/*****************************************************/
/* RTL Memory Row Decoder                            */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/

#ifndef ROW_DECODER_RTL_H
#define ROW_DECODER_RTL_H

#include ''systemc.h''

namespace memory_blk
```

```
{
SC_MODULE(ROW_DECODER)
{
    sc_in_clk clock;

    sc_in<bool> ENABLE;
    sc_out<bool> READY;

    sc_in<unsigned int> address;
    sc_out<bool> row_enable[512];

    sc_signal<int> STATE;
    sc_signal<unsigned int> aux_addr;
    int aux_out;

    SC_CTOR(ROW_DECODER)
    {
        SC_METHOD(Run);
        sensitive << clock.pos();
    }

    void Run()
    {
        switch (STATE)
        {
        case (IDLE):
        {
            READY = false;
            if (ENABLE)
                STATE = DECODE;
            else
                STATE = IDLE;

            aux_addr = address;

            break;
        }

        case (DECODE):
        {
            READY = false;
            STATE = DONE;

            aux_out = aux_addr >> 6;
            row_enable[aux_out] = true;

            break;
        }

        case (DONE):
        {
```

```
            READY = true;
            if (ENABLE)
                STATE = DONE;
            else
                STATE = FLUSH;

            break;
        }

        case (FLUSH):
        {
            READY = false;

            STATE = IDLE;

            row_enable[aux_out] = false;
        }
        }
    }

    enum
    {
        IDLE, DECODE, DONE, FLUSH
    };
};
};

#endif
```

A.2.5 MEM_WORD.H

```
/*****************************************************/
/* RTL Word Controller Model                         */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/

#ifndef MEM_WORD_H
#define MEM_WORD_H

#define WORD_SIZE 4

#include ''systemc.h''

namespace memory_blk
{
SC_MODULE(MEM_WORD)
```

```
{
    sc_bv<8> MEM_0;
    sc_bv<8> MEM_1;
    sc_bv<8> MEM_2;
    sc_bv<8> MEM_3;

    sc_in<bool> ROW_ENABLE;
    sc_in<bool> COLUMN_ENABLE;
    sc_out<bool> READY;
    sc_in<bool> RESET;
    sc_in<bool> rw;
    sc_inout<sc_bv<32> > data;
    sc_out<bool> status;

    bool aux_rw;
    sc_bv<32> aux_data;

    SC_CTOR(MEM_WORD) :
            MEM_0(0),
            MEM_1(0),
            MEM_2(0),
            MEM_3(0)
    {
        SC_METHOD(Run);
        sensitive << ROW_ENABLE.pos();

        SC_METHOD(Reset);
        sensitive << RESET;
    }

    void Run()
    {
        if (COLUMN_ENABLE)
        {
            aux_rw = rw.read();
            aux_data = data.read();

            if (aux_rw)
            {
                MEM_3 = aux_data.range(31,24);
                MEM_2 = aux_data.range(23,16);
                MEM_1 = aux_data.range(15,8);
                MEM_0 = aux_data.range(7,0);
            }
            else
            {
                aux_data.range(31,24) = MEM_3;
                aux_data.range(23,16) = MEM_2;
                aux_data.range(15,8) = MEM_1;
                aux_data.range(7,0) = MEM_0;
            }
```

```
            READY = true;

            data = aux_data;
            status = 1;
        }
    }

    void Reset()
    {
        READY = false;
        data = 0;
        status = 0;
    }

    enum
    {
        IDLE, INIT_OP, READ, WRITE, DONE
    };
};
};

#endif
```

A.3 PROTOCOLO DE COMUNICAÇÃO

```
/*********************************************************/
/* Simple Bus Communication Protocol                     */
/* Author: Gustavo Henrique Nihei                        */
/* System Design Automation Lab (LAPS)                   */
/* INF-UFSC                                                */
/* http://www.laps.inf.ufsc.br                         */
/*********************************************************/

#ifndef PROTOCOL_H
#define PROTOCOL_H

namespace protocol
{
class REQ
{
public:
    bool op;
    sc_int<32> data;
    unsigned int address;
};

class RSP
```

```
{
public:
    sc_int<32> data;
    bool status;
};

ostream& operator << (ostream& os, const REQ & r)
{
    return os<<''REQ={op={''<<r.op<<''}, data={''<<r.data<< ''}, address={''<<r.address<<
''}}'';
}

ostream& operator << (ostream& os, const RSP & r)
{
    return os<<''RSP={status={''<<r.status<<''}, data={''<<r.data<< ''}}'';
}
}

#endif
```

A.4 ADAPTADORES

A.4.1 NOC_TLM_WRAPPER.H

```
/*****************************************************/
/* NoC to TLM Wrapper                                */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/

#ifndef NOC_TLM_WRAPPER_H
#define NOC_TLM_WRAPPER_H

#include ''systemc.h''
#include ''tlm.h''
#include ''protocol.h''

#include ''noc_specific_adapter_if.h''
#include ''noc_specific_types.h''

using namespace noc_specific;

namespace memory_blk
{
template<typename device_id_t, typename addr_t, typename data_t, int memory_size >
class NOC_TLM_Wrapper:
```

```
            public sc_module,
            public Noc_adapter_if<device_id_t, addr_t, data_t>
{
    SC_HAS_PROCESS(NOC_TLM_Wrapper);

public:
    typedef protocol::REQ REQ;
    typedef protocol::RSP RSP;

    sc_port<tlm::tlm_transport_if< REQ, RSP > > target_port;
    sc_export<Noc_adapter_if<device_id_t, addr_t, data_t> > initiator_port;

    NOC_TLM_Wrapper(sc_module_name name, const device_id_t id) :
            sc_module(name),
            _device_id(id)
    {
        initiator_port(*this);
    }

public:
    device_id_t device_id() const
    {
        return _device_id;
    }

    status accept_write(device_id_t dev, addr_t addr, data_t d, int p)
    {
        if ((addr < 0) || (addr > memory_size) || (dev != _device_id))
        {
            return noc_specific::ERROR;
        }

        req.op = 1;
        req.data = d;
        req.address = addr;

        rsp = target_port->transport(req);

        if (rsp.status != 1)
        {
            return noc_specific::ERROR;
        }

        return noc_specific::OK;
    }

    status accept_read(device_id_t dev, addr_t addr, data_t& d, int p)
    {
        if ((addr < 0) || (addr > memory_size) || (dev != _device_id))
        {
            return noc_specific::ERROR;
```

```
        }

        req.op = 0;
        req.address = addr;

        rsp = target_port->transport(req);

        if (rsp.status != 1)
        {
            return noc_specific::ERROR;
        }

        d = rsp.data;

        return noc_specific::OK;
    }

private:
    const device_id_t _device_id;
    REQ req;
    RSP rsp;
};
};
#endif
```

A.4.2 TLM_RTL_WRAPPER.H

```
/*****************************************************/
/* TLM to RTL Wrapper                                */
/* Author: Gustavo Henrique Nihei                    */
/* System Design Automation Lab (LAPS)               */
/* INF-UFSC                                          */
/* http://www.laps.inf.ufsc.br                       */
/*****************************************************/

#ifndef TLM_RTL_WRAPPER_H
#define TLM_RTL_WRAPPER_H

#include ''systemc.h''
#include ''tlm.h''
#include ''protocol.h''

namespace memory_blk
{
class TLM_RTL_Wrapper:
            public sc_module,
            public tlm::tlm_transport_if< protocol::REQ, protocol::RSP >
{
public:
```

```
typedef protocol::REQ REQ;
typedef protocol::RSP RSP;

// System clock
sc_in_clk clock;

// RTL signals
sc_out<bool> ENABLE;
sc_in<bool> READY;

sc_out<bool> op;
sc_inout<sc_bv<32> > data;
sc_out<unsigned int> address;
sc_in<bool> status;

// Auxiliary
RSP _rsp;
REQ _req;

sc_fifo<REQ> fifo_req;
sc_fifo<RSP> fifo_rsp;

SC_CTOR(TLM_RTL_Wrapper)
{
    SC_THREAD(Run);
    sensitive << clock.pos();

    sc_fifo<REQ> fifo_req (1);
    sc_fifo<RSP> fifo_rsp (1);
}

void Run()
{
    while (true)
    {
        _req = fifo_req.read();

        op = _req.op;
        address = _req.address;
        data = _req.data;

        ENABLE = true;

        do
        {
            wait();
            _rsp.data = data;
            _rsp.status = status;
        }
        while (!READY);
```

```
            ENABLE = false;
            wait();

            fifo_rsp.write(_rsp);
        }
    }

    RSP transport(const REQ& req)
    {
        RSP rsp;
        fifo_req.write(req);
        rsp = fifo_rsp.read();
        return rsp;
    }
};
};

#endif
```

A.5 DESCRIÇÃO DA PLATAFORMA VIRTUAL

```
/**
 * @file          main.cpp
 * @author        Bruno de Carvalho Albertini
 *
 * @author        The ArchC Team
 *                http://www.archc.org/
 *
 *                Computer Systems Laboratory (LSC)
 *                IC-UNICAMP
 *                http://www.lsc.ic.unicamp.br/
 *
 * @version       0.1
 * @date          Wed, 16 Mar 2006 08:07:46 -0200
 *
 * @brief         Main file for dual mips example.
 *
 * @attention Copyright (C) 2002-2006 --- The ArchC Team
 *
 *    This library is free software; you can redistribute it and/or
 *    modify it under the terms of the GNU Lesser General Public
 *    License as published by the Free Software Foundation; either
 *    version 2.1 of the License, or (at your option) any later version.
 *
 *    This library is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.    See the GNU
 *    Lesser General Public License for more details.
```

```
 *
 *
 */


const char *project_name=''mips1'';
const char *project_file=''mips1.ac'';
const char *archc_version=''2.0beta3'';
const char *archc_options=''-abi -dy '';

#include  <systemc.h>
#include  ''mips1.H''

#include ''w_noc.h''

#include ''noc_specific.h''

#include ''memory/rtl/tlm_rtl_wrapper.h''
#include ''memory/rtl/noc_tlm_wrapper.h''
#include ''memory/rtl/sram_model.h''

#include ''memory/tlm/mem_tlm.h''

int sc_main(int ac, char *av[])
{
    // Bus clock
    sc_clock bus_clock;

    // Processor instances
    mips1 mips1_proc1(''mips1'');

    noc_specific::Noc<int, uint32_t, uint32_t> noc2(''bus1'', 2);
    memory_blk::NOC_TLM_Wrapper<int, uint32_t, uint32_t, 0x9F7FFF>  noc_tlm_wrapper2(''noc_tlm_wrap
3);
    memory_blk::MEM_TLM mem_tlm(''mem_tlm'', 0x9F7FFF);


// 32 kilobyte byte-addressed memory block (512x16)
    memory_blk::NOC_TLM_Wrapper<int, uint32_t, uint32_t, 0x007FFF>  noc_tlm_wrapper(''noc_tlm_wrapp
1);
    memory_blk::TLM_RTL_Wrapper tlm_rtl_wrapper(''tlm_rtl_wrapper'');
    memory_blk::SRAM_MODEL sram_model(''sram_model'', 512, 16);

    noc_tlm_wrapper.target_port(tlm_rtl_wrapper);
    noc_tlm_wrapper2.target_port(mem_tlm);

    // Wrapper that map memory with offset
    w_noc wrap1(''wrapper1'',1, 0x000000, false, 3);

    // Bindings
    noc2.adapter_port(noc_tlm_wrapper2);
```

```
noc2.adapter_port(noc_tlm_wrapper);

/************************************
    Signal Declaration
************************************/

// TLM Wrapper to Controller signal lines
sc_signal<bool> WRAPPER_ENABLE_MEM;
sc_signal<bool> WRAPPER_READY_MEM;
sc_signal<bool> WRAPPER_RW_MEM;
sc_signal<sc_bv<32> > WRAPPER_DATA_MEM;
sc_signal<unsigned int> WRAPPER_ADDRESS_MEM;
sc_signal<bool> WRAPPER_STATUS_MEM;

/************************************
    Port Connection
************************************/

// TLM Wrapper ports
tlm_rtl_wrapper.ENABLE(WRAPPER_ENABLE_MEM);
tlm_rtl_wrapper.READY(WRAPPER_READY_MEM);
tlm_rtl_wrapper.clock(bus_clock);
tlm_rtl_wrapper.op(WRAPPER_RW_MEM);
tlm_rtl_wrapper.data(WRAPPER_DATA_MEM);
tlm_rtl_wrapper.address(WRAPPER_ADDRESS_MEM);
tlm_rtl_wrapper.status(WRAPPER_STATUS_MEM);

// SRAM Model ports
sram_model.ENABLE(WRAPPER_ENABLE_MEM);
sram_model.READY(WRAPPER_READY_MEM);
sram_model.clock(bus_clock);
sram_model.rw(WRAPPER_RW_MEM);
sram_model.data(WRAPPER_DATA_MEM);
sram_model.address(WRAPPER_ADDRESS_MEM);
sram_model.status(WRAPPER_STATUS_MEM);

wrap1.bus_port(noc2);

mips1_proc1.DM_port(wrap1.target_export);

// Load elf before start
mips1_proc1.delayed_load(``bitcount.x'');

mips1_proc1.set_instr_batch_size(1);

// Prepare processors
mips1_proc1.init();

cerr << endl;
```

```
    sc_start();

    mips1_proc1.PrintStat();
    cerr << endl;

    return mips1_proc1.ac_exit_status;
}
```