ANDRÉIA APARECIDA BARBIERO

AMBIENTE DE SUPORTE AO PROJETO DE SISTEMAS EMBARCADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2006

ANDRÉIA APARECIDA BARBIERO

AMBIENTE DE SUPORTE AO PROJETO DE SISTEMAS EMBARCADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2006

ANDRÉIA APARECIDA BARBIERO

AMBIENTE DE SUPORTE AO PROJETO DE SISTEMAS EMBARCADOS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Roberto André Hexsel
Departamento de Informática, UFPR

Prof. Dr. Vóldi Costa Zambenedetti
Departamento de Engenharia Elétrica, PUCPR

Prof. Dr. Sandro Rigo
Instituto de Computação, Unicamp

Prof. Dr. Bruno Müller Jr
Departamento de Informática, UFPR

Curitiba, 10 de Julho de 2006

AGRADECIMENTOS

Agradeço em primeiro lugar ao meu companheiro Zeno Stivanin pela paciência e apoio constante. Ao professor Roberto Hexsel pela disponibilidade e pelas “injeções” de ânimo no decorrer do projeto. Ao bolsista Antônio Massaro Neto pela ajuda na codificação do simulador Atmega8515. A equipe de desenvolvedores do ArchC pelo rápido retorno no esclarecimento de dúvidas. E ao LACTEC que serviu de inspiração para este trabalho e permitiu a utilização de seus recursos para a realização de testes.

SUMÁRIO

LISTA DE FIGURAS v

LISTA DE TABELAS vi

RESUMO vii

ABSTRACT viii

1 INTRODUÇÃO 1

2 REVISÃO BIBLIOGRÁFICA 5

2.1 Definições Preliminares 5

2.2 Metodologia de Desenvolvimento de Sistemas Embarcados 7

2.2.1 Especificação e Modelagem 9

2.2.2 Validação 12

2.2.3 Síntese 14

2.3 Blocos de Propriedade Intelectual e Reuso de Software 15

2.4 SystemC 17

2.5 Simuladores de Conjuntos de Instruções 19

2.6 Trabalhos Relacionados 25

3 DESCRIÇÃO DOS MICROPROCESSADORES 27

3.1 Motorola DSP56F827 27

3.2 Atmel Atmega8515 31

3.3 Rabbit R2000 34

4 DESENVOLVIMENTO DO AMBIENTE DE SIMULAÇÃO 38

4.1 Implementação dos Simuladores 38

4.1.1 Descrição da Arquitetura da CPU 40

4.1.2 Descrição da Sintaxe das Instruções 42

4.1.3 Descrição da Semântica das Instruções 44

4.2 Implementação das Métricas de Memória 46

5 AVALIAÇÃO DE DESEMPENHO 48

5.1 Descrição do Ambiente de Testes 48

5.2 Comparação entre os Processadores 50

5.3 Precisão do Modelo do DSP56F827 54

5.4 Estudo de Caso - O Conversor Serial-IP 55

6 CONCLUSÃO E TRABALHOS FUTUROS 58

A CÓDIGO FONTE DO SIMULADOR DSP56F827 61

B CÓDIGO FONTE DO SIMULADOR ATMEGA8515 67

C CÓDIGO FONTE DO SIMULADOR R2000 70

BIBLIOGRAFIA 81

LISTA DE FIGURAS

2.1 Fluxo Genérico de Projeto de Sistemas Embarcados. 8

2.2 Estrutura da Linguagem SystemC. 18

2.3 Estrutura de um Contador em SystemC. 19

2.4 Implementação do Processo de Contagem. 19

2.5 Implementação da Interface de Leitura. 19

2.6 Fluxo de um Simulador Interpretado. 20

2.7 Fluxo de um Simulador Compilado. 20

3.1 Segmentação das Instruções no DSP56F827. 31

3.2 Interface do CodeWarrior. 32

3.3 Segmentação das Instruções no Atmega8515. 33

3.4 Interface do AVR Studio 4. 34

3.5 Arquitetura de Memória do R2000. 36

3.6 Interface do WinIDE 1.58.03. 37

4.1 Fluxo de Implementação de um Simulador Utilizando ArchC. 39

4.2 Descrição da Arquitetura Motorola DSP56F827 em ArchC. 40

4.3 Descrição da Arquitetura Atmega8515. 42

4.4 Descrição do Conjunto de Instruções do DSP56F827 em ArchC (parte). . . 43

4.5 Descrição do Comportamento de uma Instrução (semântica) do DSP56F827. 44

4.6 Incremento do Contador de Programa no R2000. 45

4.7 Código que Computa o Tamanho da Memória de Programa. 46

4.8 Trecho do Código que Computa o Tamanho da Memória de Dados. 47

5.1 Exemplo de um Arquivo de Teste em Formato Hexadecimal. 50

5.2 Utilização da Memória de Dados. 52

5.3 Utilização da Memória de Programa. 52

5.4 Número de Ciclos Executados. 53

5.5 Número de Instruções Executadas. 54

5.6 Estrutura do Projeto Serial-IP. 56

LISTA DE TABELAS

3.1 Principais Características dos Microprocessadores. 27

5.1 Programas de Teste. 48

5.2 Utilização de Memória. 51

5.3 Número de Ciclos e Instruções Executadas 51

5.4 Comparação entre Modelo e Processador DSP56F827. 55

5.5 Resultado Obtido com um Trecho da Pilha TCP/IP/PPP. 56

RESUMO

O crescimento na produção de hardware e software para sistemas embarcados exige que o tempo de projeto seja cada vez mais curto. Assim, são necessárias ferramentas que auxiliem os projetistas na escolha do hardware mais adequado para uma determinada aplicação e que possibilitem o início do desenvolvimento do código antes da conclusão do hardware. Este trabalho descreve um ambiente de simulação que permite escolher o processador mais adequado ainda na fase inicial de um projeto e iniciar a codificação do firmware imediatamente.

O ambiente de simulação consiste de simuladores que permitem estimar, através de simulações precisas, métricas de desempenho como tempo de execução e memória utilizada em código, pilha e conjunto de dados. As métricas de memória são uma contribuição inédita deste trabalho à ferramenta ArchC.

O ambiente suporta os processadores Motorola DSP56F827, Rabbit R2000 e Atmel Atmega8515 e pode ser facilmente expandido com outros processadores das famílias dos já citados. O três microprocessadores foram escolhidos por serem bastante populares do domínio de sistemas embarcados.

Os simuladores foram desenvolvidos utilizando a Linguagem de Descrição de Arquiteturas *ArchC* que é baseada na linguagem de modelagem de hardware *SystemC*.

A construção dos simuladores foi relativamente trabalhosa, principalmente tratando-se de arquiteturas complexas, como é o caso do microprocessador DSP56F827. Ao todo foram necessárias mais de 26.000 linhas de código para descrever as instruções dos três processadores.

Os testes realizados permitiram a comparação dos processadores em relação à métricas de desempenho tais como tempo de execução e memória utilizada. Em comparações efetuadas entre o simulador e um módulo de hardware do processador DSP56F827 foi constatado que o simulador possui precisão melhor que 85%.

ABSTRACT

The dynamics of the embedded computing market force the design cycles to be rather short. Thus, efficient tools are needed to assist designers in choosing the right components early in the design cycle and make possible the beginning of the software development before the conclusion of the hardware. This work describes a simulation environment that allows to choose the most adequate processor still in the initial phase of a project and to initiate the codification of firmware immediately. The tools provide performance metrics like execution time and memory usage – code, stack and data set sizes. The memory measurements are a great contribution of this work to ArchC tools.

The environment supports the processors Motorola DSP56F827, Rabbit R2000 and Atmel Atmega8515. The three microprocessors had been chosen for being popular in the domain of embedded systems.

The simulators were developed using the ArchC Architecture Description Language that is based on the hardware modeling language *SystemC*.

The construction of the simulators was relatively laborious, especially for the more complex architecture which is the case of microprocessor DSP56F827. In all, over 26.000 lines of code were necessary to model the three processors.

The tests performed had allowed the comparison of the processors regarding to performance metrics such as, the execution time and memory usage. Comparisons between a hardware module of processor DSP56F827 and the its model had shown that the simulator accuracy is better that 85%.

CAPÍTULO 1

INTRODUÇÃO

Um relatório publicado pela *Intel* em 2000 mostrava que apenas 2% de todos os processadores fabricados no mundo eram destinados a aplicações *desktop*. Os 98% restantes eram divididos em aplicações de robótica (6%), sistemas automotivos (12%), e 80% usados em sistemas embarcados [55]. As aplicações mais comuns deste último mercado são telefones celulares, brinquedos, eletrodomésticos e outros equipamentos eletrônicos. Estes produtos são projetados sob restrições de recursos tais como energia consumida, utilização de memória, tamanho do circuito integrado, e principalmente de custo. No custo deve ser incluído, além dos componentes de hardware, o desenvolvimento do hardware e de software aplicativo.

Existe uma fatia de mercado na qual desempenho e a confiabilidade são críticos enquanto que baixo consumo de energia é desejável mas não obrigatório. Nestes casos, o foco do projeto é o desenvolvimento eficiente do software e a escolha do processador com a melhor relação custo/desempenho. Os produtos desenvolvidos para este tipo de mercado são destinados, geralmente, para automação de processos em setores de geração e distribuição de energia elétrica, telecomunicações e indústria em geral.

Não é raro que os projetos neste segmento possuam o seguinte perfil: (i) tratam-se de aplicações "novas" com a especificação do sistema ainda incompleta; (ii) existem ao menos duas opções de microprocessadores; e (iii) alguns componentes de software já encontram-se disponíveis, tais como uma implementação da pilha TCP/IP ou um algoritmo de criptografia. Neste caso, questões como *"qual processador atenderá aos requisitos de desempenho e/ou consumo?"* e *"quanta memória será necessária?"* devem ser respondidas no início do processo de desenvolvimento.

Quando apenas a experiência dos projetistas é utilizada para responder àquelas perguntas alguns problemas podem ocorrer. Dentre eles salientam-se o sub- ou o super-

dimensionamento de recursos, perda de tempo e o conseqüente aumento no custo de desenvolvimento. Há super-dimensionamento quando são utilizados recursos além do necessário para determinada aplicação, como a utilização de um processador poderoso e caro para executar uma aplicação simples. No sub-dimensionamento ocorre o contrário, e a capacidade do hardware não é suficiente para satisfazer aos requisitos de desempenho, como em uma aplicação que demanda mais memória do que o disponível no microcontrolador escolhido. Neste caso pode ocorrer a inutilização de placas de circuito impresso e seu re-projeto para a inclusão de mais memória. Em ambos os exemplos, é evidente o aumento no custo e no tempo de desenvolvimento.

Este trabalho descreve um ambiente de simulação que pode auxiliar projetistas na escolha de um processador que atenda aos requisitos de sua aplicação. Os simuladores do ambiente contém modelos detalhados de processadores de 8 e 16 bits, de três fabricantes distintos. Os simuladores permitem prever com razoável precisão, no início do ciclo de desenvolvimento, (1) se determinado processador tem desempenho suficiente para a aplicação, (2) qual será o tamanho do código compilado, e (3) qual a capacidade mínima da memória de dados. Com estimativas precisas e referidas à aplicação é mais provável que a escolha do processador ou microcontrolador seja a acertada. A maioria dos fabricantes não fornece ferramentas adequadas para este tipo de comparação, ao contrário, as ferramentas são dirigidas à sua linha de produtos, na forma de software proprietário e com custo elevado, como é o caso de [19], por exemplo.

Os simuladores foram implementados com a *linguagem de descrição de arquitetura* ArchC [50, 6], que por sua vez é baseada em SystemC [31]. ArchC permite modelar um processador de forma mais abstrata do que as linguagens de descrição de hardware convencionais como Verilog e VHDL. ArchC permite descrever a arquitetura desejada de forma eficiente e rápida, e a partir da descrição gera automaticamente o simulador apropriado, que é capaz de estimar o tempo de execução e o número de acessos à memória.

Correntemente, estão implementados modelos de três microcontroladores amplamente utilizados no mercado, que são: o Motorola DSP56F827 [47], Rabbit R2000 [48], e Atmel Atmega8515 [7]. Estes foram escolhidos por conta de nosso envolvimento em projetos

em que estes seriam possíveis candidatos [39, 60]. Além dos três citados acima, os simuladores podem ser facilmente expandidos para aceitar a simulação de outros processadores das três famílias, já que na maioria das vezes o conjunto de instruções não sofre grandes alterações, apenas se adiciona ou remove algumas funções. Por exemplo, o simulador do microprocessador DSP56F827 pode facilmente simular o microprocessador DSP56F826.

O simulador do microprocessador Rabbit R2000 foi testado parcialmente, devido a problemas com o ambiente de geração do código executável que é utilizado como entrada para o simulador.

Para agregar mais informações úteis ao conjunto de métricas disponibilizado pela linguagem ArchC foi desenvolvido, neste trabalho, uma medida para o tamanho, ou utilização, da memória de dados e de programa. A métrica da utilização da memória de dados inclui a medição dinâmica dos tamanhos máximos de *heap* e pilha.

O objetivo deste trabalho é disponibilizar um recurso eficiente e eficaz para a seleção de processadores e especificação de memória, e que permita otimizar o processo de desenvolvimento de sistemas embarcados. Além disso, a disponibilidade de um simulador na fase inicial de um projeto possibilita que o desenvolvimento de software seja iniciado antes que haja hardware disponível, permitindo o levantamento antecipado das características dos algoritmos e de detalhes obscuros da especificação. A utilização de modelos para simular as aplicações antes do desenvolvimento é recomendada principalmente quando se trata de aplicações complexas [58].

Além dos argumentos já apresentados, parte da motivação para este trabalho é a carência, no mercado brasileiro de sistemas embarcados, de ferramentas e recursos que auxiliem no desenvolvimento das aplicações. O mercado de sistemas embarcados é bastante abrangente e oferece diversas possibilidades, portanto é necessário um esforço mais dirigido por parte da comunidade de Computação no sentido de aperfeiçoar e criar novas técnicas e ferramentas para impulsionar o crescimento desta área.

Um exemplo da crescente importância desta área no Brasil é a criação do projeto *CI Brasil* pelo Ministério de Ciência e Tecnologia, que visa a criação de cinco centros de projeto para criação e fabricação de circuitos integrados eletrônicos comerciais. O

CEITEC [18] é um destes centros e está em fase de construção da sede em Porto Alegre.

O texto está organizado como segue. Alguns conceitos sobre desenvolvimento de sistemas embarcados e trabalhos relacionados são discutidos no Capítulo 2. No Capítulo 3 são descritos os microprocessadores utilizados no trabalho. O desenvolvimento do projeto e a avaliação de desempenho são vistos, respectivamente, nos Capítulos 4 e 5. A conclusão e as propostas para a expansão do ambiente de desenvolvimento, estão no Capítulo 6. Os Apêndices A, B e C mostram os modelos de algumas instruções dos microprocessadores, codificados na linguagem ArchC.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Neste capítulo é apresentada a base teórica para o desenvolvimento do trabalho e, está dividido da seguinte forma: A Seção 2.1 apresenta alguns conceitos básicos de sistemas embarcados; na Seção 2.2 é descrita uma metodologia genérica de desenvolvimento de sistemas embarcados; a Seção 2.3 discute sobre o reuso de software e os blocos de propriedade intelectual; a Seção 2.4 apresenta a linguagem de modelagem de hardware, *SystemC* e nas Seções 2.5 e 2.6 são mostradas as técnicas de simulação de sistemas embarcados e os trabalhos relacionados, respectivamente.

2.1 Definições Preliminares

Devido a evolução dos sistemas embarcados e do crescimento da utilização destes no cotidiano, vários grupos tem direcionado seus esforços nesta área. Os principais objetos de estudo estão relacionados ao aumento de desempenho, economia de recursos críticos (energia consumida e utilização de memória, por exemplo), segurança, confiabilidade, reuso de software e metodologias de projeto.

O projeto de um sistema embarcado é bastante complexo, pois requer conhecimentos sobre questões geralmente pouco exploradas ou ignoradas em aplicações *desktop*, como por exemplo a restrição ao consumo de energia e a pouca disponibilidade de memória.

A complexidade do projeto também está relacionada ao espaço arquitetural, que torna a exploração das opções para a construção de um sistema ainda mais difícil. A arquitetura de hardware de um sistema embarcado pode conter um ou mais processadores, memórias de vários tipos, interfaces para periféricos e blocos dedicados. Os componentes são interligados por uma estrutura de comunicação que pode variar de um barramento a uma rede complexa (*Network-on-Chip* ou *NoC*). No caso de sistemas contendo componentes programáveis, o software de aplicação pode ser composto por múltiplos processos,

distribuídos entre diferentes processadores e comunicando-se através de mecanismos variados [17]. Em muitos casos é necessário a implementação de um sistema operacional de tempo real (*RTOS*) para auxiliar no gerenciamento e sincronização dos processos [12].

Os processadores podem ser de arquiteturas diversas: *RISC, VLIW, DSP* e *Processadores Integrados para Aplicações Específicas (ASIP).* O uso de *pipeline* e a exploração de paralelismo em geral já são aplicados em processadores de sistemas embarcados [33].

O projeto de hierarquia de memória também segue os padrões clássicos de computação, cache e memória principal [33], apesar de alguns estudos questionarem a usabilidade da memória cache para sistemas embarcados, já que o seu consumo de energia é bastante elevado [17].

Em muitas aplicações, é necessária a integração do software e do hardware em uma única pastilha, em um sistema chamado de *System-on-Chip* ou *SoC.* Neste caso, quando as restrições de recursos tais como potência, memória e tempo, tornam-se críticas, pode ser indispensável o desenvolvimento de um *Circuito Integrado para Aplicação Específica* (*ASIC*) ou um *Field Programmable Gate Array* (*FPGA*). Pesquisas recentes, indicam a necessidade de algumas aplicações em utilizar *Multiprocessor-on-Chips* (*MPSoCs*), o que aumenta consideravelmente a complexidade de um sistema [37].

Um *FPGA* é um circuito programável composto por um conjunto de células lógicas ou blocos lógicos alocados em forma de uma matriz. Em geral, a funcionalidade destes blocos e o roteamento são configuráveis por software [1]. Tanto um *ASIP* quanto um *ASIC* se baseiam em um conjunto de módulos de hardware construído especialmente para uma determinada aplicação. Os *ASIPs* são processadores que possuem a arquitetura e o conjunto de instruções personalizados para um dado sistema. Um exemplo de *ASIP* é o processador *R6*, desenvolvido para fins didáticos e utilizado na PUC-RS [46]. Um *ASIP* deve ser implementado sobre uma plataforma de hardware também específica, que pode ser um *ASIC* ou um *FPGA*.

Os *ASICs* são bastante utilizados em produtos eletrônicos construídos em larga escala, como brinquedos e eletrodomésticos. Normalmente são necessários meses para construir um *ASIC*, tornando este inviável como ferramenta de avaliação e prototipação de novos

produtos. Neste caso, a utilização de um *FPGA* passa a ser a opção mais interessante.

Geralmente um projeto de sistema embarcado necessita ser desenvolvido em um curto espaço de tempo para atender as necessidades do mercado. Para auxiliar nesta tarefa, é necessário o estudo de metodologias eficientes que atendam aos requisitos de tempo de projeto e que adicionem confiabilidade à aplicação. A próxima seção descreve uma metodologia formal para o desenvolvimento destes sistemas.

2.2 Metodologia de Desenvolvimento de Sistemas Embarcados

Devido a complexidade dos sistemas embarcados e a variedade de soluções possíveis no domínio da aplicação, é necessária a divisão do projeto em etapas de modo a facilitar o desenvolvimento e organizar o fluxo de tarefas. A Figura 2.1 mostra uma metodologia genérica de projeto de sistemas embarcados [17] e como o projeto pode ser dividido.

O projeto de um sistema embarcado é iniciado, normalmente, por uma especificação da funcionalidade desejada, descrita através de uma linguagem ou formalismo adequado. A seguir, explora-se o espaço de projeto arquitetural, com o objetivo de encontrar uma arquitetura que implemente as funções contidas na especificação inicial e que atenda aos requisitos de projeto, em termos de custo, desempenho, potência, área, etc. O resultado final desta etapa é uma macro-arquitetura (ou arquitetura abstrata), contendo um ou mais processadores de determinados tipos (DSP, microcontroladores) e outros componentes necessários (memórias, interfaces, blocos dedicados de hardware), todos interconectados através de uma infra-estrutura de comunicação (um ou mais barramentos ou uma *NoC*).

Entre a especificação funcional e a macro-arquitetura, estabelece-se um mapeamento através do qual cada função do sistema é atribuída a um processador ou a um bloco dedicado de hardware. Este mapeamento é também chamado de particionamento de funções entre hardware e software ou ainda, entre blocos dedicados e funções implementadas por um processador de instruções.

A exploração do espaço de projeto deve ser capaz de responder a três questões básicas: *(1) Quantos e quais processadores e blocos dedicados de hardware são necessários? (2) Qual é o mapeamento ideal entre funções e componentes de hardware? (3) Qual é a estrutura*

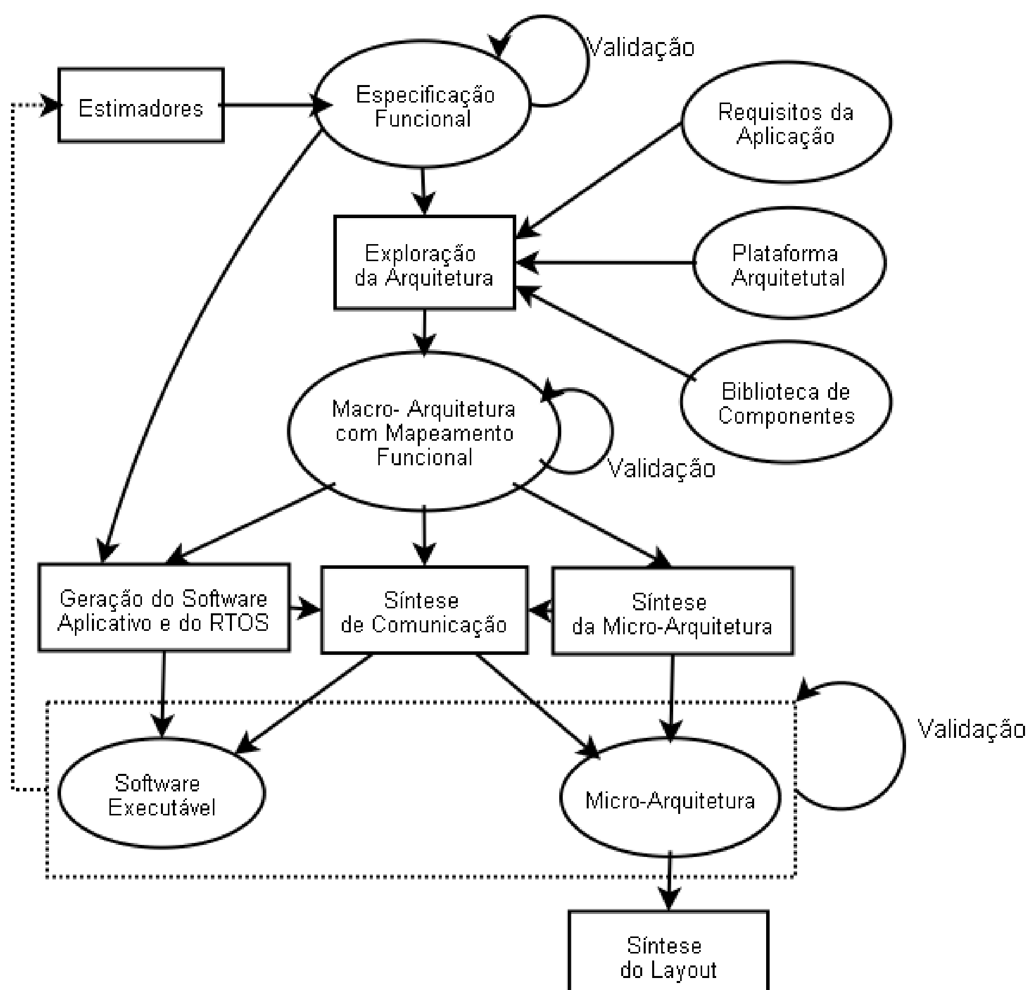

Figura 2.1: Fluxo Genérico de Projeto de Sistemas Embarcados.

de comunicação ideal para conectar os componentes entre si, tendo em vista as trocas de informação que devem ser realizadas entre as funções mapeadas para os componentes? Para que a exploração seja efetuada rapidamente, é fundamental a existência de estimadores que, a partir da especificação funcional do sistema, sejam capazes de informar com um grau de precisão adequado os valores de métricas importantes de projeto como desempenho, potência, área e consumo de memória.

Uma vez definida a macro-arquitetura, é necessária a geração do software para a mesma, a partir da especificação funcional do sistema. Idealmente, seria desejável uma síntese automática do software incluindo tanto o software aplicativo como o *RTOS*.

Os componentes de hardware e software selecionados para a macro-arquitetura podem ter interfaces heterogêneas, implementando diferentes protocolos de comunicação. Neste caso, é necessária a síntese da comunicação entre os componentes. Esta síntese deve gerar

adaptadores (*wrappers*) que fazem a conversão entre os diferentes protocolos.

Uma vez definidos os componentes de hardware da macro-arquitetura, incluindo a infraestrutura de comunicação e os eventuais adaptadores, o hardware pode ser sintetizado. Numa primeira etapa, a macro-arquitetura pode ser expandida para uma micro-arquitetura, contendo o detalhamento de todos os componentes e suas interconexões pino-a-pino e considerando o funcionamento do circuito com precisão no nível de ciclo de relógio. Numa segunda etapa, podem ser usadas ferramentas convencionais de síntese de hardware, que a partir da micro-arquitetura produzem o *layout* final do circuito. Para tanto, é necessário que a micro-arquitetura esteja descrita numa linguagem apropriada para estas ferramentas, como por exemplo *VHDL* [9] ou *Verilog* [40]. A existência prévia de *layouts* para os componentes de hardware selecionados facilita bastante esta síntese, que se limita então ao posicionamento e roteamento de células.

Em todas as etapas da metodologia de projeto, é necessária a validação das descrições funcionais e arquiteturais geradas. Normalmente, esta validação se dá por simulação, sendo portanto necessária a existência de simuladores adequados para o tratamento das linguagens utilizadas no projeto.

As próximas seções descrevem as etapas formais do projeto de sistemas embarcados segundo Edwards e Vincentelli [27], são elas: Especificação e Modelagem, Validação e Síntese. Na Especificação e Modelagem são mapeados os requisitos e as restrições da aplicação e o produto desta etapa é o modelo do sistema. A Validação é utilizada para verificar se o sistema está sendo desenvolvido corretamente. Já na Síntese, o sistema é efetivamente desenvolvido.

2.2.1 Especificação e Modelagem

O processo de desenvolvimento de um projeto pode ser encarado como uma seqüência de passos que transforma uma especificação descrita informalmente em uma especificação detalhada, para então se transformar em um produto. Um projetista pode executar um ou mais passos neste processo. Neste contexto, o processo de entrada é a *especificação* e a saída é a *implementação*. Por exemplo, um programador de sistemas pode ver um

conjunto de rotinas em C como uma implementação do seu projeto na qual diversos outros passos foram executados antes do produto estar concluído. Durante este processo a validação é necessária, principalmente para garantir desempenho e funcionalidade.

Infelizmente os vários passos que compõem o processo de desenvolvimento são, na maioria das vezes, descritos de maneira informal e não possuem uma relação precisa entre os componentes. Neste caso, um *modelo formal* de projeto deve ser gerado e executado. Um modelo formal de projeto consiste dos seguintes componentes:

- Uma especificação funcional que exibe as relações, implícitas e explícitas, entre os sinais de entrada, de saída e de controle;
- Um conjunto de propriedades que especificam as relações entre as entradas, saídas e estados e são comparadas com a especificação funcional;
- Um conjunto de métricas de desempenho; e
- Um conjunto de restrições que estão relacionadas com as métricas.

A especificação funcional caracteriza a operação do sistema, enquanto as restrições de desempenho podem ser utilizadas para limitar o custo. As propriedades são listadas separadamente porque elas são mais simples, mais abstratas e também mais incompletas quando comparadas com a especificação funcional.

Uma propriedade é uma asserção sobre o comportamento do sistema e não uma descrição do mesmo. Por exemplo, quando projeta-se um protocolo de rede é desejável que a implementação nunca entre em impasse (*deadlock*). Esta propriedade pode ser chamada de *liveness*.

Um projeto é representado geralmente por um conjunto de componentes que podem ser vistos como blocos isolados interagindo entre si. Um modelo de computação define o comportamento e a interação entre estes blocos. Existem vários modelos de computação que podem ser utilizados para descrever sistemas embarcados [27]:

- Eventos Discretos: utilizado para modelagem de tempo;

- Máquinas de Estados Finitas (FSM): utilizadas para modelar comportamento sequêncial, mas inviável para modelagem de concorrência e memória;

- Síncrono/Reativo: ordena eventos simultâneos, utilizado em simuladores baseados em ciclo (*cycle based*);

- Rede de Fluxo de Dados: o sistema é representado por um grafo direcionado, no qual os nodos representam o processamento, e as arestas a ordem dos eventos.

Para auxiliar na construção de um determinado modelo várias linguagens podem ser utilizadas, cada qual com suas vantagens e desvantagens. A linguagem *C*, por exemplo, não é uma linguagem ideal para especificação, principalmente se o grau de abstração desejado e de generalidade da aplicação são elevados, mas possui a vantagem de permitir a geração de software para um grande número de processadores utilizados no contexto de sistemas embarcados.

A adoção da orientação a objetos, como em *C++*, se por um lado é vantajosa do ponto de vista de especificações de alto nível, na qual reuso e especialização de componentes são características muito interessantes, por outro lado causa problemas para a síntese de hardware e tamanho do software, além do potencial para comprometer o desempenho do sistema no que diz respeito a tempo de execução.

Na tentativa de aumentar as possibilidades de modelagem e abstração, *Java* também tem sido utilizada como ferramenta de descrição e simulação de sistemas [34, 35]. Contudo, os problemas relacionados à síntese de hardware e desempenho são equivalentes aos provocados pelo C++.

As linguagens C, C++ e Java apresentam uma evidente desvantagem, tendo em vista sua inadequação semântica para a descrição de aspectos de hardware. Para este tipo de descrição podem ser utilizadas linguagens específicas, como *VHDL* e *Verilog*. A grande vantagem destas é a possibilidade de se utilizar os modelos como entrada para simulação e síntese automática de circuitos. A linguagem *VHDL* possui construções mais voltadas à simulação, de tal modo que algumas de suas construções não são sintetizáveis, o que força ferramentas de síntese a aceitarem apenas um determinado sub-conjunto da linguagem

e/ou estilo de descrição. As linguagens *VHDL* e *Verilog* têm uma semântica destinada apenas para a descrição de hardware, não sendo apropriadas para descrições de software nem para especificações funcionais de alto nível.

Outras linguagens como *Matlab* [42], *SpecC* [29] e *SDL* [28] podem ser usadas. Matlab possui à facilidade de modelagem de fenômenos físicos e a possibilidade de descrições mistas com sinais analógicos e digitais. A linguagem SpecC é uma linguagem baseada em C e permite que a funcionalidade de um sistema seja capturada através de uma rede hierárquica de comportamentos (*behaviors*) conectados por canais hierárquicos (*channels*). A desvantagem desta linguagem é que não possui suporte à ferramentas de síntese. A SDL foi criada originalmemte para a área de telecomunicações e possui a vantagem de permitir a especificação de sistemas concorrentes, porém é fraca para descrição de algoritmos porque não possui primitivas de controle de fluxo tais como *if, for, while*.

Com o objetivo de manter o alto nível de abstração e possibilitar a descrição do hardware, foi desenvolvida a linguagem *SystemC* [31]. O SystemC combina as vantagens do C++, e sua adequação ao processo de geração de software, com uma semântica adicional (na forma de uma biblioteca de classes) apropriada para a descrição de hardware. Devido a importância do SystemC no escopo deste trabalho, suas características serão discutidas com mais detalhes na Seção 2.4.

Apesar do avanço das linguagens de programação e especificação atuais, ainda não é possível modelar simultâneamente hardware e software e este é um dos desafios da área de pesquisa relacionada ao *codesign* [4, 59].

2.2.2 Validação

O passo seguinte à construção do modelo é a sua validação. A validação do sistema evolui conforme este é desenvolvido e vários passos são requeridos até se atingir níveis satisfatórios de funcionalidade e desempenho.

A principal ferramenta utilizada nesta etapa é a *simulação*. Idealmente, um simulador de sistema embarcado deveria ser capaz de cobrir aspectos de hardware e software permitindo a *co-simulação*. Porém, como visto anteriormente, nenhuma linguagem de pro-

gramação consegue atender simultâneamente todos os domínios de uma aplicação. Neste caso é necessária a validação de descrições multi-linguagem em determinados passos do projeto.

Algumas ferramentas de co-simulação atuais, como *CoCentric System Studio*, da Synopsys [53], e *Seamless CVE*, da Mentor [45], permitem a integração de SystemC, C, simuladores de software no nível de instruções de máquina de um processador e linguagens de descrição de hardware diversas, como VHDL e Verilog.

Na prática, utiliza-se um simulador de programas de propósito geral, que simula a CPU escolhida executando o software de aplicação, escrito em C por exemplo, sobre um determinado modelo, baseado em *VHDL*, *Verilog*, SystemC, entre outros. Estes simuladores podem ser classificados entre os modelos abaixo [27]:

- *Nível de Porta*: Simulação lógica no nível de hardware e portas lógicas. É indicado para projetos simples onde o custo da validação pode ser pequeno.

- *Conjunto de Instruções da Arquitetura (ISA)*: Simula o comportamento das instruções do processador (geralmente escrito em C), com informações da interface hardware/software;

- *Barramento Funcional*: São modelos de hardware utilizados somente para testar a interface do processador. Não executam nenhum software, e ao invés disso os simuladores são programados para que a interface pareça estar executando algum programa;

- *Baseado em Tradução*: Estes simuladores convertem o código que será executado no processador em um código que pode ser executado nativamente pelo simulador. Além da tradução do código propriamente dita, uma grande dificuldade destes simuladores é manter as informações de tempo do código original.

Atualmente o modelo de Conjunto de Instruções é o mais utilizado, já que permite observar o comportamento da aplicação no nível de instruções e possibilita que a implementação do software seja iniciada antes da conclusão do hardware. Outra vantagem é

que após o término do projeto o mesmo simulador pode ser reutilizado para simular outras aplicações destinadas ao mesmo processador. A Seção 2.5 mostra o que são e como implementar estes simuladores.

Uma outra alternativa para a validação é a verificação formal, que consiste na validação matemática do comportamento do sistema [27]. A verificação formal pode ser aplicada nas fases de (a) *Especificação*, para garantir que o sistema será capaz de atender aos requisistos de funcionamento, e durante (b) *Implementação* para identificar possíveis inconsistências. Por exemplo, em [23] é proposto um método para modelagem e validação formal baseado nas *Redes de Petri* que promete capturar características importantes do sistema e representá-las em diferentes níveis de granularidade.

A eficiência da verificação formal depende da complexidade do modelo utilizado e da aplicação em si. Apesar de ser uma área de estudo em crescimento, ainda não atingiu um grau de maturidade que permita a sua aplicação em todas as etapas do projeto e para diferentes domínios de aplicação [17].

2.2.3 Síntese

No processo de síntese são gerados refinamentos para transformar uma especificação mais abstrata em uma menos abstrata. Esta etapa é geralmente dividida em: (i) *mapeamento da arquitetura*, (ii) *particionamento* e (iii) *síntese do software e do hardware*.

Durante o mapeamento da arquitetura, o projetista define quais componentes e ferramentas serão utilizados e decide se estas atendem às restrições do sistema. Uma arquitetura é geralmente composta de:

- Componentes de Hardware: microprocessadores, memórias, dispositivos de entrada/saída, *ASICs* e *FPGAs*;
- Componentes de Software: sistema operacional, *drivers* de dispositivos e bibliotecas de funções;
- Canais de Interconexão: canais abstratos, barramentos e memórias compartilhadas.

A escolha destes componentes depende da especificação funcional, dos requisitos e restrições impostos ao projeto. Para auxiliar nesta tarefa é recomendado o uso de estimadores ou simuladores.

O particionamento define quem vai fazer o quê dentre o software e o hardware, geralmente com base nos resultados de um simulador/estimador. Por exemplo, o projetista pode decidir implementar determinada função em hardware para atender requisitos de desempenho.

Existem vários estudos sobre particionamento automático [27]. Uma abordagem comum é a utilização de um estimador/simulador para identificar blocos com desempenho crítico. Após a identificação, estes blocos são movidos para implementação em hardware. Este processo é repetido até que uma solução razoável seja encontrada.

Em [43, 44] é descrito um sistema no qual a seleção do hardware e/ou do software é feita automaticamente de acordo com o consumo de energia, desempenho e utilização de memória. Por exemplo, será selecionado para determinada aplicação o algoritmo de ordenação que possuir a melhor relação entre desempenho × consumo de energia × utilização de memória.

A fase final é a síntese do hardware e do software, quando estes são implementados. As entradas para esta fase são a especificação funcional e o conjunto de recursos mapeado na arquitetura.

Para diminuir o tempo do projeto, alguns estudos propõem métodos de síntese automática de hardware e software. A ferramenta TEReCS [16], por exemplo, objetiva a síntese automática de um *RTOS* mínimo e dedicado, com ênfase nas estruturas de comunicação. O *RTOS* é construído a partir de uma biblioteca pré-definida de serviços denominada DREAMS.

2.3 Blocos de Propriedade Intelectual e Reuso de Software

Devido à expansão do mercado de sistemas embarcados é pouco provável que uma empresa desenvolva apenas um projeto desta natureza ou que deseje implementar módulos já desenvolvidos e validados por outras equipes. Por conta disto, muitos esforços estão

sendo dedicados ao reuso de componentes de software e hardware, também chamados de *Intellectual Property Blocs* ou blocos IP [17].

Atualmente é possível encontrar uma grande quantidade de fornecedores de componentes IP de hardware e software [25]. No domínio do software, os componentes que geralmente são reusáveis incluem pilhas de protocolos de comunicação (*TCP/IP*, *Wireless*), sistemas operacionais, *drivers* de dispositivos, algoritmos de compactação e criptografia. No âmbito do hardware pode-se encontrar interfaces de comunicação (*Bluetooth*, 802.11), codificadores/decodificadores de áudio e vídeo e barramentos, dentre outros. A caracterização precisa dos componentes em termos das métricas de projeto deve ser definida pelo projetista do sistema.

Numa situação ideal, componentes selecionados para uma dada macro-arquitetura teriam interfaces compatíveis e poderiam ser conectados diretamente uns aos outros, ou então através de uma estrutura de comunicação também consistente. Isto vale tanto para componentes de hardware, por exemplo conectados através de um barramento padronizado, como para componentes de software, comunicando-se através de funções padronizadas disponíveis numa API. Numa situação mais genérica, no entanto, o projetista pode desejar reusar componentes já disponíveis que foram desenvolvidos em projetos anteriores, dentro de contextos diferentes, e que podem portanto apresentar interfaces inconsistentes.

Segundo [17], os componentes IP de hardware podem ser oferecidos de duas maneiras distintas. Componentes *hard*, já estão sintetizados para uma dada tecnologia-alvo e é seu *layout* que é comercializado, já caracterizado quanto aos resultados obtidos em termos de área, potência e desempenho. Sua restrição, no entanto, é a quase impossibilidade de adaptação para outra arquitetura e/ou tecnologia. Componentes *soft*, por outro lado, são ofertados através de descrições de mais alto nível (tipicamente nível *Register-Transfer (RT)*), podendo ter suas interfaces e comportamentos adaptados, se necessário, e sintetizados para diferentes tecnologias.

Componentes IP de software também podem ser divididos em componentes *hard*, já compilados para um determinado processador e oferecidos através de seu código executável, e *soft*, descritos numa linguagem de alto nível.

A adaptação de componentes IP a um novo projeto pode ocorrer de duas maneiras. No caso de componentes *soft*, descrições de alto nível estão disponíveis e podem ser modificadas de tal modo que as interfaces de componentes a serem conectados passem a ser consistentes entre si. A adaptação pode também afetar a própria funcionalidade do componente, caso isto seja necessário no contexto do novo projeto. No caso de componentes *hard*, no entanto, interfaces heterogêneas não podem ser adaptadas através de modificações nas descrições dos componentes. Neste caso, é imprescindível o desenvolvimento de adaptadores de comunicação.

Devido a importância dos componentes IP no desenvolvimento de sistemas embarcados atuais, foi criado o projeto *Brazil-IP* [15], que é um consórcio constituído por oito universidades brasileiras, UNICAMP, USP, UFPE, UFCG, UFRGS, UFMG, UnB e PUC-RS, com o objetivo de formar mão-de-obra qualificada em projetos de hardware no Brasil.

2.4 SystemC

Devido a crescente necessidade de ferramentas que auxiliem no projeto de sistemas embarcados, muitos esforços vem sendo direcionados à criação e aperfeiçoamento de linguagens de especificação e modelagem de sistemas. Atualmente observa-se uma competição acirrada entre as linguagens, cada qual com suas vantagens e desvantagens [41]. Apesar da concorrência, a linguagem *SystemC* vem sendo adotada por diversos fabricantes como linguagem padrão de modelagem e simulação. Uma comparação recente entre modelos implementados em SystemC e *Register Transfer Level* (RTL) *Hardware Description Language* (HDL) mostrou que a simulação utilizando SystemC pode ser até 10.000 vezes mais rápida [51].

SystemC foi desenvolvido em 1999 através de uma cooperação entre a *Synopsys*, a *CoWare/IMEC* e a *Universidade da Califórnia - Irvine*. Hoje em dia várias empresas e instituições compõem a *Open SystemC Initiative* (OSCI) e disponibilizam ferramentas e código fonte na página do projeto [54].

SystemC é uma biblioteca de classes que estende a linguagem C++ com o objetivo de modelar sistemas. A biblioteca possibilita a descrição de comportamento concorrente,

uma noção de tempo em operações seqüenciais, tipos de dados para descrever hardware, estrutura hierárquica e suporte à simulação. A Figura 2.2 mostra a arquitetura do SystemC.

Figura 2.2: Estrutura da Linguagem SystemC.

O núcleo da linguagem é baseado em um simulador dirigido a evento que trabalha com eventos e processos. Também são incluídos no núcleo da linguagem módulos e portas para representar a estrutura bem como, interfaces e canais para descrever a comunicação.

A biblioteca também oferece um conjunto de tipos de dados necessários para a modelagem de hardware e certos tipos para programação de software. Por fim, a biblioteca oferece um conjunto de primitivas como sinais e filas FIFO.

Em SystemC, um sistema pode ser modelado como uma coleção de módulos que contém processos, portas e canais. Um processo define o comportamento de um módulo específico e disponibiliza métodos para expressar concorrência. Um canal implementa uma ou mais interfaces, onde uma interface é uma coleção de métodos ou funções. Um processo acessa a interface de um canal através de uma porta [21]. A Figura 2.3 mostra a estrutura de um contador ou *timer* [22].

O contador possui uma interface que permite configurar o(s) registrador(es) para leitura ou escrita. Quando o contador expira, uma interrupção é enviada para a porta da interrupção. Deve existir também uma conexão com o relógio do sistema (*clock*) através

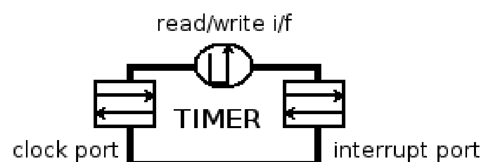

Figura 2.3: Estrutura de um Contador em SystemC.

de outra porta.

O processo de contagem consiste em decrementar o contador toda vez que uma borda do relógio é detectada (subida ou descida). Se o contador chegar a zero é gerada uma interrupção. As Figuras 2.4 e 2.5 mostram um exemplo de implementação, em SystemC, do processo de contagem e da interface de leitura.

```
void timer::tick()
{
   ...
   if (--this->count == 0)
      this->interruptPort = 1;
   ...
}
```

Figura 2.4: Implementação do Processo de Contagem.

```
void timer::read(int register, unsigned char *value)
{
   *value = this->regs[register];
}
```

Figura 2.5: Implementação da Interface de Leitura.

2.5 Simuladores de Conjuntos de Instruções

Um simulador de conjunto de instruções ou *Instruction Set Simulator* (ISS) é uma ferramenta que imita o comportamento da execução de uma aplicação em uma dada arquitetura. Os ISSs são usados para validar o projeto da arquitetura, o projeto de um compilador, bem como para avaliar as decisões do espaço arquitetural durante a exploração do domínio.

Os ISSs podem ser *Compilados* ou *Interpretados* [49]. Os simuladores interpretados são flexíveis, pois permitem a simulação de qualquer arquitetura e aplicação, porém são lentos. Nesta técnica, uma instrução é buscada, decodificada e executada em tempo de execução, como mostra a Figura 2.6. A decodificação da instrução é a etapa que mais consome tempo na simulação. Em [50, 8] utiliza-se uma cache para armazenamento das instruções decodificadas para minimizar o problema.

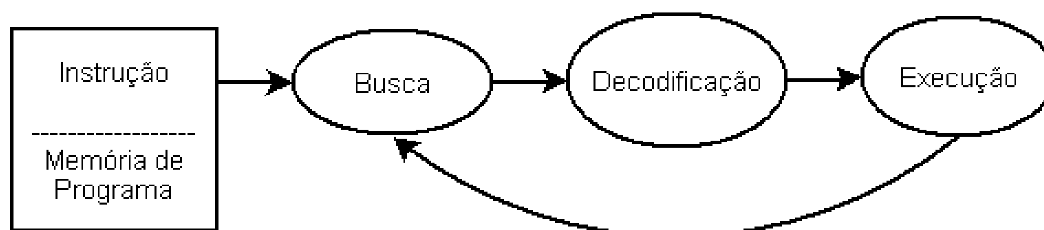

Figura 2.6: Fluxo de um Simulador Interpretado.

Para resolver o problema da velocidade de simulação, o simulador compilado decodifica as instruções em tempo de compilação, conforme é mostrado na Figura 2.7. No entanto, os simuladores compilados dependem que o código do programa seja conhecido antes da geração do simulador. A aplicação passa por um processo de compilação e o código intermediário é gerado com a decodificação das instruções já efetuada [14]. Esta técnica aumenta substancialmente a velocidade de simulação mas perde em flexibilidade, já que cada simulação de uma nova aplicação requer uma nova compilação do simulador. Além disso, o simulador não permite a simulação de códigos que se auto-modificam ou que carregam bibliotecas em tempo de execução. As linguagens *ArchC* [50] e *ABSCISS* [5] permitem a implementação deste tipo de simulador, como será visto adiante.

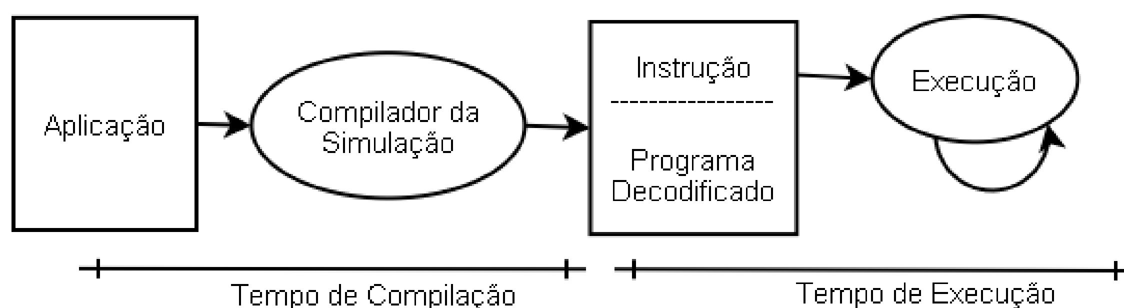

Figura 2.7: Fluxo de um Simulador Compilado.

Um simulador pode ser funcional ou com precisão de ciclo (*cycle-accurate*). Com a

simulação funcional pode-se verificar e testar o comportamento das instruções e a funcionalidade básica do sistema modelado. A descrição das instruções para os simuladores com precisão de ciclo deve conter informações detalhadas sobre o tempo de execução de cada instrução. Os simuladores podem modelar implementações multi-ciclo ou segmentado (*pipeline*). Num simulador multi-ciclo declara-se o número de ciclos necessário para executar cada tipo de instrução, enquanto que em um simulador segmentado a descrição deve conter um modelo detalhado dos segmentos do processador bem como de circuitos de adiantamento ou de bloqueios [33].

Devido a necessidade de aumentar a velocidade de desenvolvimento de um sistema embarcado e permitir a utilização do simulador nas fases iniciais do projeto, é necessário que os simuladores não demandem muito tempo para construção.

Para auxiliar nesta tarefa foram introduzidas as Linguagens de Definição de Arquitetura ou *Architectute Description Languages (ADLs).* As ADLs podem ser classificadas considerando dois aspectos, (1) *conteúdo* e (2) *objetivo*. A classificação orientada ao conteúdo é baseada na natureza da informação que a ADL pode descrever enquanto que a classificação orientada ao objetivo é baseada no propósito da ADL. Por exemplo, a ADL pode ser orientada à simulação, orientada à síntese, orientada à teste, orientada à compilação, orientada à validação e orientada ao Sistema Operacional.

Com base na natureza da informação, pode-se ainda subclassificar as ADLs como estruturais, comportamentais, parciais e mistas. As ADLs estruturais descrevem a estrutura em termos da arquitetura de componentes e sua conectividade. Já as ADLs comportamentais descrevem o comportamento do conjunto de instruções da arquitetura do processador. As ADLs mistas agregam mais que um aspecto apresentados anteriormente. Já as parciais são linguagens que descrevem informações específicas sobre a arquitetura para uma determinada tarefa, como por exemplo uma ADL destinada a síntese da interface não requer a descrição do comportamento do processador. Segue abaixo a descrição de algumas ADLs:

Lisa. *Language for Instruction Set Architecture - LISA* [61] foi desenvolvida na Aachen University of Technology (Alemanha) com o objetivo de produzir simuladores de qualidade

de produção. Um aspecto importante da linguagem LISA é sua habilidade de descrever explicitamente o caminho de controle. A modelagem explícita do caminho de dados e de controle é necessária para a simulação precisa dos ciclos. Quando foi introduzida, a principal contribuição de LISA foi a capacidade de descrição de *pipeline* e de modelos seqüenciais. Algumas desvantagens de LISA são a complexidade da implementação para descrever formatos de instruções complexos e a falta de ferramentas de domínio público para manipulação da linguagem.

EXPRESSION. EXPRESSION [32] é uma Linguagem de Descrição de Arquitetura de código livre focada na exploração do espaço arquitetural para SoCs e na geração automática de compiladores e simuladores. O ponto forte da linguagem é o suporte à especificação de hierarquia de memória. A ADL EXPRESSION, fornece ferramentas para geração do EXPRESS, um compilador otimizado que permite paralelismo em nível de instruções, e do SIMPRESS, um simulador funcional da linguagem.

Sleipnir. Sleipnir é uma ADL que tem o objetivo de facilitar a descrição das arquiteturas e de prover informações sobre desempenho [36]. Os simuladores construídos através da Sleipnir são portáveis e podem ser executados em diferentes plataformas. Para melhorar o tempo de simulação, a Sleipnir utiliza os conceitos de simulação compilada e gera código intermediário que contém as instruções decodificadas. Uma desvantagem do Sleipnir é a dificuldade em descrever arquiteturas com conjunto de instruções complexos, como o de processadores superscalares.

ABSCISS. A ABSCISS [5] é um gerador de simuladores compilados de conjunto de instruções que efetua a simulação de códigos escritos em *Assembly*. Os simuladores gerados podem ser de precisão de ciclo (*cycle-accurate*) ou funcionais. Ao contrário da maioria dos geradores de simuladores, o ABSCISS aceita como entrada um arquivo escrito em Assembly, o que dispensa a descrição dos *opcodes* e o uso de montadores e ligadores. Esta característica aumenta consideravelmente o tempo de descrição da arquitetura. Um inconveniente da abordagem no nível de montador é a baixa precisão na simulação de

caches. Isto ocorre devido ao ABSCISS atribuir por conta própria os endereços de dados e de instruções, possivelmente de forma diferente que um ligador original.

GAPH. A equipe do *Grupo de Apoio ao Projeto de Hardware* (GAPH) [46] apresenta uma ferramenta que permite a definição e a reconfiguração de uma arquitetura implementada em FPGAs. O conjunto de instruções, as operações da ULA, e o conjunto de registradores podem ser configurados. Esta ferramenta é útil como um recurso didático pois permite a exploração do espaço de projeto, mas não é suficientemente poderosa para permitir a descrição de um processador comercial relativamente complexo, pois não há suporte à segmentação.

CACO-PS. O *Cycle-Accurate Configurable Power Simulator* (CACO-PS) [13] permite a descrição de arquiteturas através de uma linguagem própria que relaciona sinais de entrada aos sinais de saída, e onde uma biblioteca em C contém descrições do comportamento dos componentes da arquitetura. O CACO-PS permite otimizar uma aplicação com relação ao consumo de energia seja pelo re-projeto de algum componente "quente", ou pela escolha de um algoritmo mais adequado. Os arquivos de entrada (memória de programa e de dados) devem ser gerados com a ferramenta *Sashimi* a partir de programas escritos em Java [35].

ArchC. *ArchC* é uma especialização do SystemC com o objetivo de geração automática de simuladores [50]. ArchC utiliza as bibliotecas do SystemC para modelar o hardware a ser simulado, e inclui ferramentas capazes de automatizar o processo de construção do simulador. Para gerar um simulador de uma CPU com ArchC é necessário descrever as características básicas da organização da CPU e descrever a sintaxe e a semântica do conjunto de instruções. Na versão 1.6 do ArchC, os arquivos com a descrição da organização (arquitetura), e do conjunto de instruções são submetidos ao pré-processador do ArchC, que gera automaticamente a estrutura do simulador. O arquivo com a semântica do conjunto de instruções é gerado e deve ser preenchido com a descrição do comportamento das instruções.

ArchC pode suportar três tipos de simuladores: (a) funcional, (b) multi-ciclo e (c) segmentado *(pipeline)*, descritos anteriormente.

Os simuladores gerados com ArchC permitem medir diretamente grandezas como (i) o número que ciclos executados, (ii) tempo de execução, (iii) número de instruções executadas, (iv) número de chamadas de E/S, e (v) número de acessos a cada instrução e à memória [8].

Uma medida fundamental para os projetistas de sistemas embarcados, e que não é diretamente produzida pelo ArchC, é a quantidade de memória necessária para executar determinada aplicação. Um mecanismo que supre esta falta foi implementado neste trabalho e é descrito na Seção 4.2.

ArchC permite a construção de simuladores Interpretados e Compilados, o que lhe confere uma grande vantagem em relação as demais linguagem, que geralmente focam em um dos modelos. Além disso, a ArchC permite uma simulação detalhada da hierarquia de memória [57].

Outra característica importante do ArchC é a capacidade dos simuladores gerados emularem chamadas ao sistema operacional. Com isso, aplicações que efetuam operações de entrada/saída, como leitura e escrita em arquivos, também podem ser simuladas. Para utilizar esta funcionalidade, o projetista deve implementar interfaces de acesso às funções do sistema operacional que consistem basicamente em mapear os parâmetros e o valor de retorno das funções.

Na versão 1.5, ArchC foi extendida para permitir a geração automática de montadores para arquiteturas com instruções de tamanho fixo [10]. Uma versão beta do ArchC 2.0 foi disponibilizada recentemente e inclui diversas melhorias, entre elas, (i) geração de montadores/desmontadores para conjuntos de instruções variáveis, (ii) geração de ligadores, (iii) melhoria na coleção de estatísticas e (iv) suporte ao protocolo (*Transaction Level Modeling*) utilizado para comunicação.

No Capítulo 4 são detalhados os passos necessários para a implementação de um simulador utilizando a linguagem ArchC.

As ADLs evoluiram consideravelmente nos últimos anos e o grupo acima representa um pequeno conjunto das diversas linguagens existentes. Outros modelos podem ser encontrados em [56].

Após o estudo das linguagens de descrição de arquiteturas foi concluído que a melhor opção para o trabalho proposto é a linguagem ArchC. ArchC mostrou-se a mais completa, abrangendo o espaço da simulação de sistemas embarcados. Outro motivo é a atividade de desenvolvimento, que propicia atualização das ferramentas e inclusão de novas funcionalidades. A facilidade de manipulação, flexibilidade e boa documentação também foram itens decisivos na escolha.

Uma das propostas deste trabalho é estender o conjunto de estatísticas da linguagem ArchC permitindo a medida do tamanho do conjunto de dados e de programa, determinando dinamicamente, e com precisão, o tamanho da pilha e do *heap*.

2.6 Trabalhos Relacionados

O trabalho aqui descrito consiste na implementação de simuladores para três arquiteturas diferentes com o objetivo de disponilizar métricas de desempenho que auxiliem na etapa da exploração arquitetural. A seguir são descritos os ambientes de desenvolvimento disponíveis para estes processadores

Os microprocessadores Motorola DSP56F827 e Rabbit R2000 não possuem ferramentas de domínio público disponíveis para simulação. A plataforma *CodeWarrior*, da Metrowerks [19] é a ferramenta mais utilizada por desenvolvedores dos microprocessadores da família 56800, a qual pertence o DSP56F827. Esta plataforma permite a implementação do software, execução passo-a-passo e simulação. É possível verificar o número de ciclos executados e a utilização de memória. A desvantagem desta ferramenta é o custo, em torno de $2000, que inviabiliza sua aquisição apenas para fins de teste da arquitetura.

O microcontrolador R2000 da Rabbit não possui ferramenta de simulação no mercado. As plataformas de desenvolvimento, *WinIDE* da Softools [52] e *Dynamic C* [48] da Rabbit, são aplicativos proprietários e só executam quando conectadas diretamente a um módulo de hardware.

O microprocessador Atmega8515, possui a ferramenta AVR Studio, da Atmel [7] que é gratuita, porém não disponibiliza métricas de desempenho. As aplicações desenvolvidas para o Atmega8515 também podem ser compiladas utilizando o *Gnu Compiler Collection (GCC)* [30]. Em [38] e [3] também são apresentados um simulador e um ambiente de desenvolvimento *Assembly*, respectivamente, para o processador Atmega8515. Ambos não disponibilizam métricas de desempenho.

Este trabalho contribuirá para que os desenvolvedores destes dispositivos utilizem o simulador para iniciar o desenvolvimento do software antes da conclusão do hardware, e com os futuros projetistas, que poderão utilizar esta ferramenta para decidir qual processador melhor atende às suas necessidades.

CAPÍTULO 3

DESCRIÇÃO DOS MICROPROCESSADORES

Neste capítulo são descritas as principais características dos microprocessadores utilizados neste trabalho e de seus ambientes nativos de desenvolvimento.

Os três microprocessadores possuem diferenças significativas em termos de recursos e de custo, como mostra a Tabela 3.1. O DSP56F827 da Motorola (DSP) é um processador de 16 bits e instruções de tamanho variável, enquanto que o Atmel Atmega8515 (AT) é um processador de 8 bits de dados, com instruções de 16 bits. O Rabbit R2000 (RB) é um processador de 8 bits com instruções de tamanho variável. As diferenças entre os microprocessadores contribuem para a verificação da qualidade do ambiente, pois através dos testes será possível efetuar a medição e a comparação direta da capacidade de processamento de cada um deles. Comparações como estas permitirão aos projetistas decidir com base em dados concretos, e quando possível, a decisão será baseada na execução do código da aplicação para a qual os processadores são candidatos. Nas Seções que se seguem são descritos os processadores.

Processador	**DSP**	**AT**	**RB**
Freqüência [MHz]	80 *	16	30
Memória Flash [KBytes]	128	8	64**
RAM Interna [Bytes]	8192	512	$64 \cdot 10^3$ **
RAM Externa [KBytes]	128	64	1024
Custo [US$]	11,11	4,84	12,75

* A freqüência do relógio interno é o dobro da externa.
** A memória de dados e código é compartilhada.

Tabela 3.1: Principais Características dos Microprocessadores.

3.1 Motorola DSP56F827

Atualmente, diversos sistemas embarcados necessitam processar sinais digitalmente. Um exemplo clássico é o telefone celular, no qual quase todo o processamento das informações

de entrada/saída da antena é feito no domínio digital para permitir maior repetitibilidade e facilidade de projeto em relação ao processamento analógico. Arquiteturas para processamento digital de sinais tornaram-se muito populares na última década, e impulsionadas pelo mercado de modems e outros equipamentos de comunicação, chegam ao mercado sob forma de processadores especializados para execução de algoritmos específicos de processamento de sinais, com alto desempenho, baixo custo e baixo consumo.

Os algoritmos mais utilizados em DSPs contém, geralmente, diversas operações aritméticas como multiplicações e somas consecutivas. Filtros digitais, a transformada rápida de *Fourier* (FFT) e a transformada discreta do cosseno (DCT) são exemplos de algoritmos que são executados em DSPs. Devido ao número de multiplicações e somas envolvidos nestes algoritmos, ao invés de se utilizar duas instruções (multiplicação seguida de acumulação) e vários registradores, a estratégia utilizada pelos DSPs é o uso da instrução *MAC* (multiplica e acumula) que é uma operação poderosa disponível nestes processadores.

O microprocessador DSP56F827 é membro da família 56800 de Processadores Digitais de Sinais ou *Digital Signal Processors* da Motorola, da qual são membros o DSP56F826, o DSP56F803, dentre outros.

Características Gerais da Arquitetura.

- Segmentado, até 40 MIPS a 80 MHz de freqüência. No mínimo dois ciclos por instrução.
- Dissipação de potência de 150 mW a 198 mW.
- 128 Kbytes de memória Flash de Programa.
- 2 Kbytes memória RAM de programa.
- 8 Kbytes de memória Flash de Dados.
- 8 Kbytes de memória RAM de Dados.
- Possibilidade de expansão de 128 Kbytes de Memória RAM para Dados e Programa.

- Dois acumuladores de 36 bits (A e B)
- Três registradores de índice de 16 bits (X0, Y0 e Y1).
- Quatro registradores de 16 bits (A0, A1, B0 e B1).
- Dois registradores de extensão de sinal de 4 bits (A2 e B2).
- Quatro temporizadores de propósito geral e um temporizador *time-of-day* (base de tempo em segundos).
- Três portas seriais.
- 60 linhas de entrada/saída.
- Instruções de tamanho variável, de 16 a 64 bits.
- Conversores Analógico-Digital e Digital-Analógico.
- Interface *Join Test Action Group* (JTAG) e *On-Chip Emulation* (OnCE) para programação e depuração de programas.

Os acumuladores A e B podem ser usados em porções separadas (A0, A1, B0, B1, A2 e B2) e permitem que estes processadores sejam usados também para aplicações comuns.

Arquitetura de Memória. A arquitetura de memória da família DSP56800 é baseada na arquitetura *Harvard* [33], com a memória e os barramentos de código e dados separados. Endereços separados e barramentos exclusivos para cada espaço de endereçamento permitem acessos simultâneos à memória de dados e de programa. Existe um segundo barramento na memória de dados que é utilizado para mapeamento de dispositivos. Neste caso, podem ser executados duas leituras simultâneas à memória de dados, totalizando três acessos paralelos à memória.

É possível a inclusão de memória externa ao núcleo do processador. São permitidos 128 Kbytes de expansão para a memória de dados e 128 Kbytes para a memória de programa. A memória do DSP56F827 é endereçada em palavras de 16 bits.

Saturação e Limitação de Dados Os algoritmos para DSP muitas vezes necessitam calcular valores maiores que a precisão do processador. Quando o resultado de uma operação ultrapassa o tamanho de um registrador o procedimento padrão é sinalizar um *overflow*. Um exemplo de *overflow* ocorre na operação *0x7fff + 0x5 = 0x8004*. Neste caso o resultado extrapola o limite do registrador e muda de um número grande positivo (correto) para um número grande negativo (incorreto).

A ocorrência de *overflow* cria problemas em sistemas com processamento de sinais em tempo real. Para contornar este problema o microprocessador DSP56F827 utiliza a técnica de *saturação*, que converte o valor excedido para o máximo valor de mesmo sinal que é suportado pelo processador.

A saturação é especialmente importante quando os dados são processados por um filtro e a saída vai para um conversor digital/analógico (DAC), por exemplo. Neste caso, a saída é limitada ao invés de gerar um *overflow*. Sem a saturação, a saída pode ser incorretamente alterada de um valor grande positivo para um valor grande negativo e pode causar problemas na saída de um DAC e conseqüêntemente na aplicação embarcada.

A saturação na arquitetura DSP56800 é opcional e pode ser aplicada a dois limitadores, o limitador de dados que satura quando o dado é movido para um dos acumuladores, e o limitador de saída da multiplicação que satura a saída das unidades de multiplicação.

Segmentação. A execução das instruções é segmentada para permitir que a maioria das instruções execute em dois ciclos de relógio. Porém certas instruções requerem mais ciclos para executar, são elas: (i) instruções com tamanho maior que dois bytes, (ii) instruções que utilizam modo de endereçamento que necessitam mais de um ciclo para executar, (iii) instruções que acessam a memória de programa; e (iv) instruções de desvio. No caso de desvio, é necessário um ciclo para drenar o *pipeline* caso o desvio seja tomado.

A segmentação do DSP56F827 consiste em três estágios: (1) Busca, (2) Decodificação e (3) Execução. Enquanto uma instrução é executada, a próxima instrução é decodificada e a seguinte é buscada. Se uma instrução possui tamanho de quatro bytes, a palavra

adicional será buscada antes da próxima instrução. A Figura 3.1 mostra o fluxo do pipeline [47].

Operation	Instruction Cycle									
	1	2	3	4	5	6	7	•	•	•
Fetch	F1	F2	F3	F3e	F4	F5	F6	•	•	•
Decode		D1	D2	D3	D3e	D4	D5	•	•	•
Execute			E1	E2	E3	E3e	E4	•	•	•

Figura 3.1: Segmentação das Instruções no DSP56F827.

Os estágios F1, D1 e E1 referem-se a busca, decodificação e execução, respectivamente, da primeira instrução. A terceira instrução tem tamanho de duas palavras, logo são necessários dois ciclos de busca (F3 e F3e), dois ciclos de decodificação (D3 e D3e) e mais dois ciclos para executar a instrução (E3 e E3e).

Ferramentas de Desenvolvimento. A ferramenta de desenvolvimento mais utilizada pelos programadores da família DSP56800 é o ambiente *CodeWarrior 7* da Metrowerks [19]. A Figura 3.2 mostra a interface padrão da ferramenta.

O CodeWarrior permite a programação de aplicações em C ou C++, bem como a simulação e depuração. Além disso, o ambiente fornece algumas métricas de desempenho, como número de ciclos e instruções executadas. Esta ferramenta é proprietária.

3.2 Atmel Atmega8515

O microprocessador Atmega8515 faz parte da família de processadores *AVR 8-bits RISC* da Atmel [7]. Os componentes desta família caracterizam-se pelo conjunto simplificado de instruções e baixo consumo de energia.

Características Gerais da Arquitetura.

- Freqüência do processador de 0 a 16 MHz.
- Dissipação de potência de 12 mW a 60 mW.

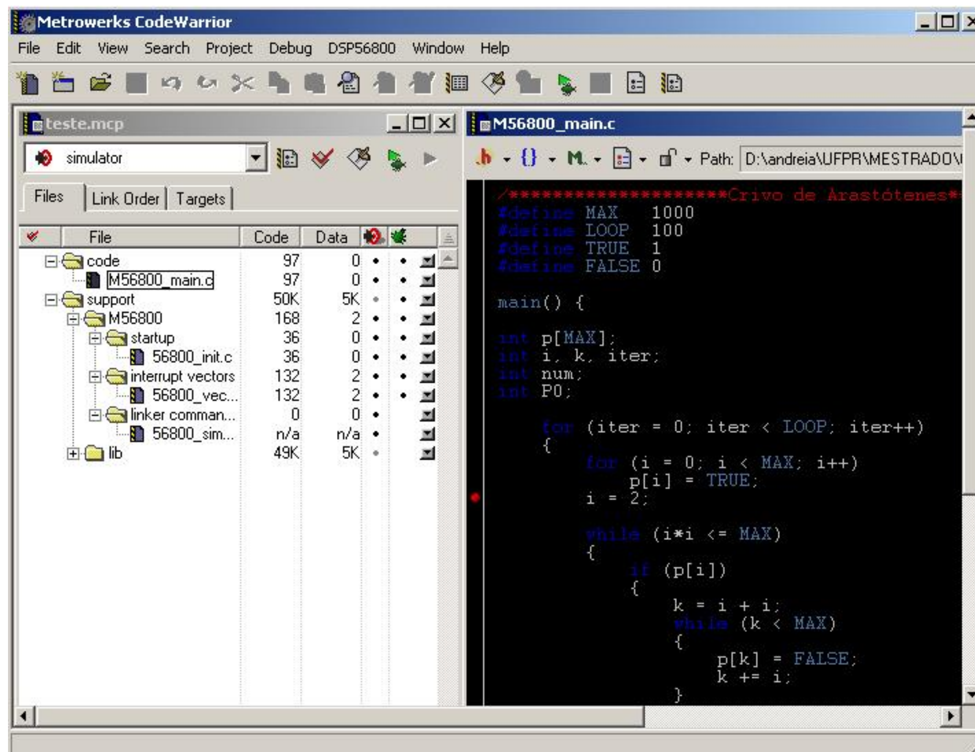

Figura 3.2: Interface do CodeWarrior.

- 32 registradores de propósito geral.
- 8 Kbytes de memória Flash, utilizada para armazenamento de programas.
- 512 bytes de EEPROM.
- 512 bytes de SRAM.
- 32 linhas de entrada/saída de propósito geral.
- Temporizadores/contadores flexíveis, com modos de comparação.
- Porta serial para programação do processador.
- Porta serial síncrona ou *Serial Peripheral Interface* (SPI).
- Contador de Programa (PC) de 12 bits.
- Todas as instruções são palavras de 16 ou de 32 bits (duas palavras).

Arquitetura de Memória. Com o objetivo de aumentar o desempenho e o paralelismo, a família AVR também utiliza a arquitetura de *Harvard*. As instruções *JMP* e *CALL* podem endereçar diretamente todo o espaço de endereçamento. A memória Flash de programa é dividida em dois segmentos, o *Boot Program* e o *Application Program*. Ambos os segmentos possuem bits de proteção à leitura e à escrita. O Atmega8515 possui 8 Kbytes de memória Flash para armazenamento do programa. Como todas as instruções tem tamanho de 16 ou 32 bits, a Flash é organizada como 4K $\times$ 16.

A arquitetura AVR possui dois principais espaços de memória, a memória de dados e a memória de programa. Adicionalmente, o Atmega8515 possui uma memória EEPROM para armazenamento de dados.

Também pode ser incluída, opcionalmente, uma memória SRAM externa de 64 Kbytes. Esta área inicia no endereço seguinte ao da SRAM interna. Como o banco de registradores, os registradores de entrada e saída e a SRAM interna ocupam os primeiros 608 bytes da memória, são disponíveis apenas 64928 bytes da memória externa.

Segmentação. A família AVR é implementada com dois segmentos. Enquanto uma instrução está sendo executada, a busca antecipada da próxima instrução na memória de programas é realizada. Isto permite que instruções sejam executadas em um ciclo de relógio. Dessa forma obtém-se 1 MIPS por MHz. A Figura 3.3 mostra um diagrama de tempo do *pipeline* do microprocessador Atmega8515 [7].

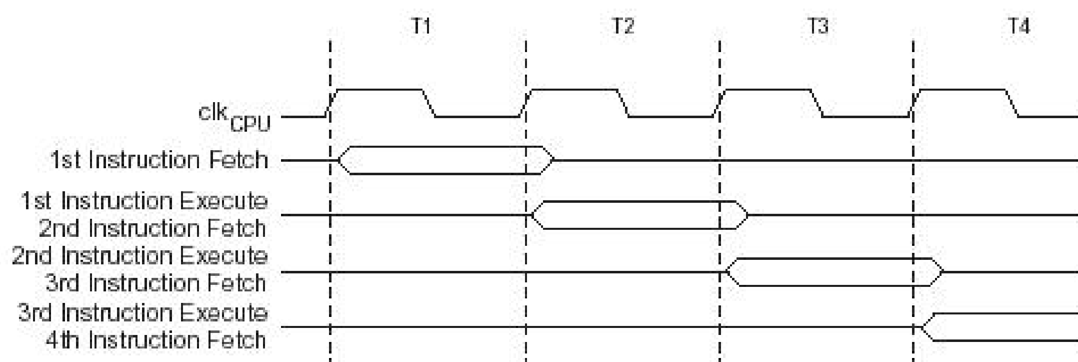

Figura 3.3: Segmentação das Instruções no Atmega8515.

Ferramentas de Desenvolvimento. A Atmel disponibiliza gratuitamente uma ferramenta de desenvolvimento para seus processadores, o ambiente AVR Studio 4 [7]. A ferramenta executa somente em ambiente Windows e é composta por um ambiente de programação, que suporta a linguagem C, além de permitir a simulação e depuração dos componentes do microcontrolador (Memória, E/S, Registradores). O AVR Studio não disponibiliza métricas de desempenho. A Figura 3.4 mostra a interface do AVR Studio. As aplicações para o Atmega8515 também podem ser compiladas com o *GCC* [30].

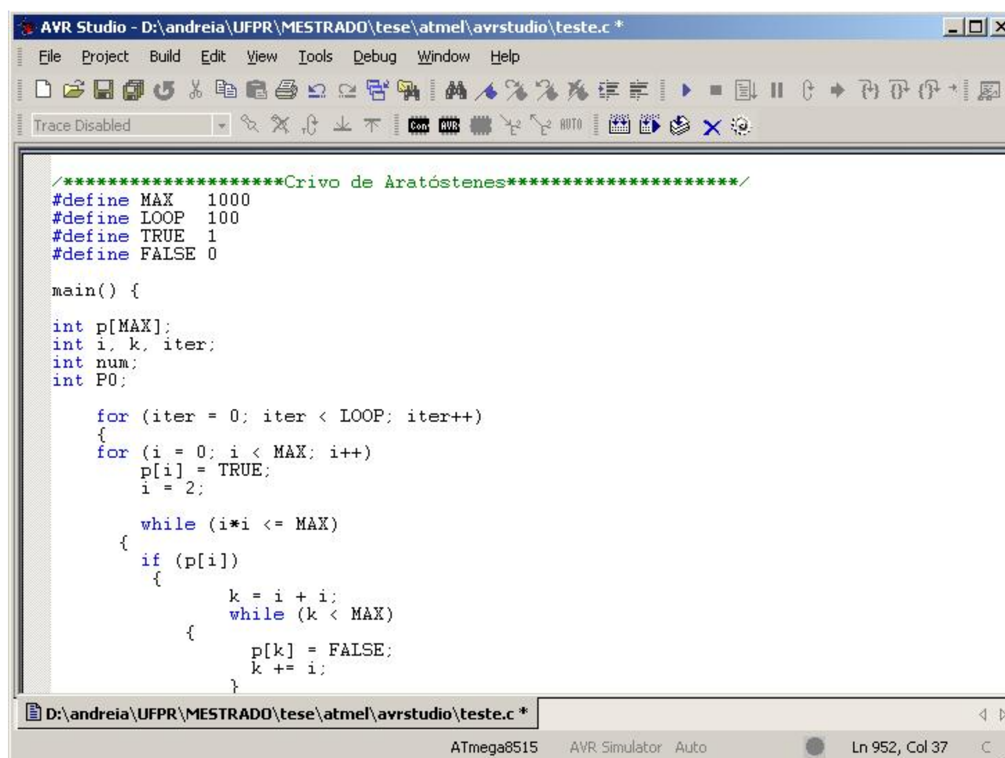

Figura 3.4: Interface do AVR Studio 4.

3.3 Rabbit R2000

O microprocessador R2000 da Rabbit é um processador de propósito geral de 8 bits bastante utilizado no mercado. Este processador é derivado da arquitetura Z80 e o conjunto de instruções é facilmente portável para o seu sucessor R3000.

Características Gerais da Arquitetura.

- Dissipação de potência 22 mW.

- Freqüência do núcleo de 1,8 a 29,5 MHz.
- 64 Kbytes de memória compartilhada entre dados e programa.
- Possibilidade de expansão da memória para 1 Mbyte.
- Quatro níveis de interrupção.
- 40 linhas paralelas de entrada/saída.
- Dois registradores de índice de 16 bits (IX e IY).
- Oito registradores de 8 bits que podem ser concatenados em quatro de 16 bits (A, F, H, L, D, E, B e C).
- Oito registradores alternativos de 8 bits que também podem ser concatenados em quatro de 16 bits (A', F', H', L', D', E', B' e C').
- Quatro portas seriais, duas síncronas e duas assíncronas.
- Porta *slave*, que permite que o R2000 seja usado como um periférico.
- Seis temporizadores/contadores.

Arquitetura de Memória. Os microprocessadores da Rabbit não utilizam a arquitetura *Harvard* de memória como os processadores da Motorola e Atmel, descritos acima. Ao invés disso eles compartilham o espaço de memória entre código e dados. A Figura 3.5 mostra como a memória é segmentada [48].

O registrador de tamanho de segmento (*SEGSIZE*) determina os limites da memória. O segmento de extensão de código ocupa a faixa de endereços entre 0x0E000 e 0x0FFFF. O início do segmento de pilha é especificado pelos 4 bits mais significativos do registrador e possui tamanho de 4 Kbytes. Os bits menos significativos do SEGSIZE são o limite da memória de dados. O restante do espaço é utilizado para código.

O mapeamento de memória permite que 16 bits de endereço sejam traduzidos em 20 bits. Para isso são necessários três registradores de segmento para mapear 16 bits em 1

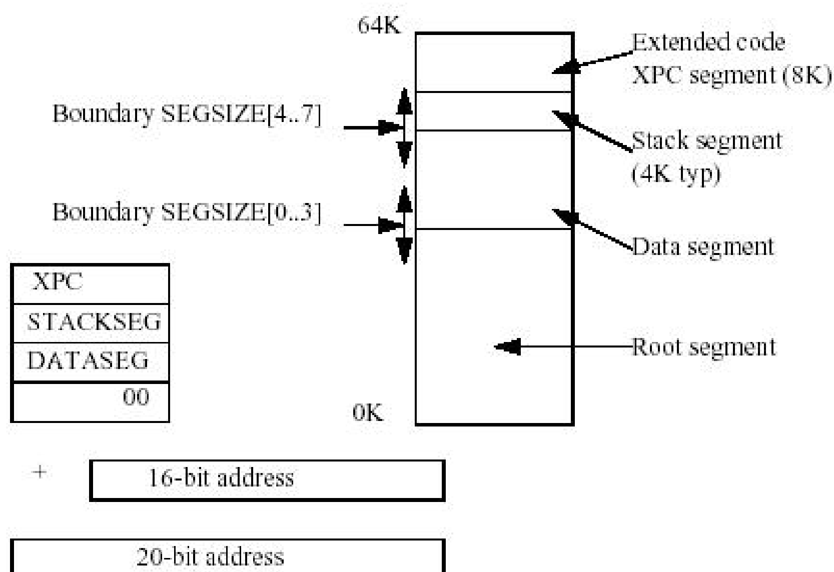

Figura 3.5: Arquitetura de Memória do R2000.

MB de memória. O espaço de 16 bits é separado em quatro segmentos, cada segmento possui um registrador de 8 bits que é adicionado ao endereço de 16 bits, formando assim um endereço de 20 bits.

Busca de Instruções. O Rabbit R2000 não utiliza segmentação na execução das instruções. A instrução é lida a partir do segmento de código e são inseridos *wait states* de memória para controlar o término da execução de uma instrução e o início da próxima.

Ferramentas de Desenvolvimento. Os processadores da Rabbit possuem duas opções para o desenvolvimento de aplicações, a ferramenta *Dynamic C* [7], distribuída pela Rabbit, e a ferramenta *Softools WinIDE* [52]. Ambas permitem a programação em C e oferecem ferramentas de depuração, porém não são capazes de simular o processador. O desenvolvimento do aplicativo deve iniciar considerando a existência de um módulo de hardware já disponível para teste. A Figura 3.6 mostra a interface padrão do *WinIDE 1.58.03* que foi utilizado para geração de código durante este trabalho. Esta versão, que não é mais atual, foi considerada a mais completa e mais fácil de se utilizar.

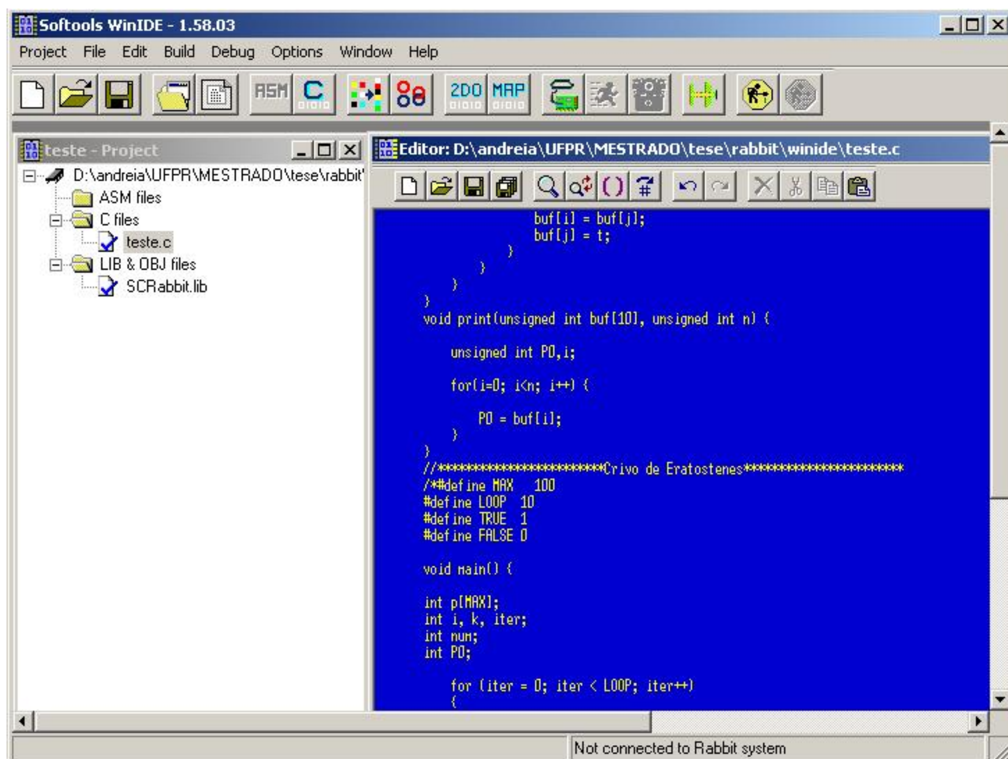

Figura 3.6: Interface do WinIDE 1.58.03.

CAPÍTULO 4

DESENVOLVIMENTO DO AMBIENTE DE SIMULAÇÃO

De acordo com a Seção 2.5, a linguagem mais indicada para auxiliar no desenvolvimento dos simuladores para os microprocessadores DSP56F827, Atmega8515 e R2000 é a linguagem ArchC. Com a linguagem ArchC é possível medir diretamente grandezas como (i) o número que ciclos executados, (ii) tempo de execução, (iii) número de instruções executadas, (iv) número de chamadas de E/S, e (v) número de acessos a cada instrução e à memória [8].

Para complementar o conjunto de métricas do ArchC foi acrescentada, neste trabalho, uma métrica que fornece a quantidade de memória necessária para executar determinada aplicação.

Neste capítulo são descritos o método de implementação dos simuladores e a técnica de obtenção das métricas relacionadas à utilização de memória pelos aplicativos, utilizando a linguagem ArchC. Os modelos descritos aqui foram desenvolvidos com a versão 1.6 do ArchC.

O Apêndices A, B e C contém trechos dos arquivos com a definição da arquitetura, sintaxe e semântica dos três microprocessadores.

4.1 Implementação dos Simuladores

Os modelos foram escritos para a simulação funcional e Multi-Ciclo. A versão funcional foi utilizada para testar o comportamento básico do simulador e a funcionalidade das instruções. Já a versão Multi-ciclo foi implementada com o intuito de produzir métricas de desempenho com uma precisão adequada às fases iniciais do desenvolvimento. Testes preliminares, comparando os resultados de simulação Multi-Ciclo com medidas em módulos de hardware do DSP56F827, indicam que as estimativas de tempo ficam a $\pm 15\%$ dos valores medidos com o hardware.

O desenvolvimento de versões Multi-Ciclo dos três modelos, para o Motorola DSP56F827, Rabbit R2000 e Atmel Atmega8515 foi relativamente trabalhoso, principalmente para o processador da Motorola que possui inúmeras funcionalidades e modos de execução. Ao todo foram necessárias mais de 26.000 linhas de código para descrever os três conjuntos de instruções.

Para gerar um simulador de uma CPU com ArchC é necessário, (i) descrever as características básicas da organização da CPU e (ii) descrever a sintaxe do conjunto de instruções. Estes arquivos são submetidos ao pré-processador do ArchC, que gera automaticamente a estrutura do simulador utilizando SystemC. Um terceiro arquivo é gerado no qual deve ser descrito o comportamento das instruções, definindo-se a semântica das instruções. Após a modelagem das instruções, os arquivos são compilados e produzem o executável do simulador. A Figura 4.1 mostra esta seqüência.

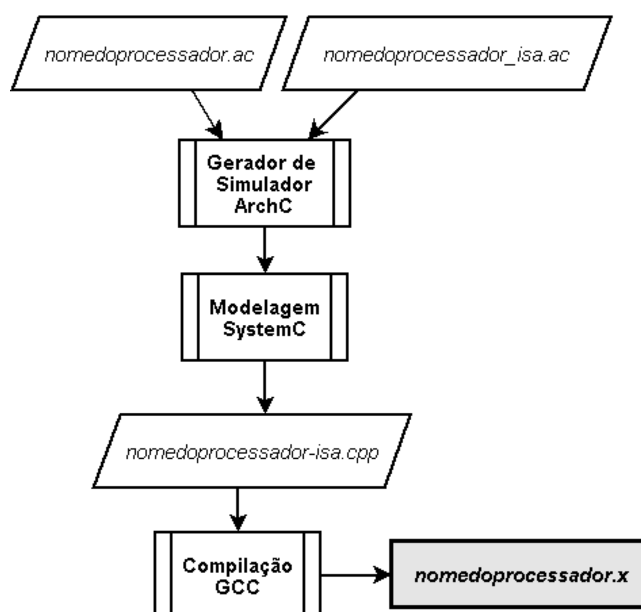

Figura 4.1: Fluxo de Implementação de um Simulador Utilizando ArchC.

O programador é responsável pelo desenvolvimento dos arquivos:

- *nomedoprocessador.ac*, que contém a descrição da arquitetura do processador incluindo, (i) tamanho da memória, (ii) tamanho da palavra da arquitetura e (iii) descrição dos registradores;

- *nomedoprocessador_isa.ac*, onde é descrito o formato das instruções ou *sintaxe*; e
- *nomedoprocessador-isa.cpp*, onde é descrito o comportamento das instruções ou sua *semântica*.

O ArchC é responsável por gerar a estrutura do simulador e efetuar a modelagem do sistema utilizando SystemC. Após a compilação de todos os arquivos utilizando o GCC, é gerado o simulador (nomedoprocessador.x). Para simular uma aplicação utilizando o simulador basta executar: *./nomedoprocessador.x "aplicação"*. As tarefas que devem ser efetuadas pelo programador são detalhadas adiante.

4.1.1 Descrição da Arquitetura da CPU

O primeiro passo para a construção de um simulador utilizando ArchC é a descrição da arquitetura. A Figura 4.2 mostra o formato do arquivo que contém a arquitetura do processador DSP56F827. A nomenclatura do arquivo segue o padrão *nomeprocessador.ac*.

```
AC_ARCH(dsp56F827){
   //Tamanho da palavra em bits
   ac_wordsize 16;
   //Tamanho da memória
   ac_mem DM:128k;
   //Descrição dos Registradores
   ac_reg SR;     //Registrador de Estado
   ac_reg OMR;    //Registrador de Modo de Operação
   ac_reg LC;     //Contador de Loop
   ac_reg LA;     //Endereçador de Loop
   ac_reg HWS0;   //Apontador da Pilha de Hardware
   ac_reg HWS1;   //Apontador da Pilha de Hardware
   ac_reg M01;    //Modificador de Cálculo de Endereço
   ac_reg N;      //Registrador de Deslocamento
   ac_reg SP;     //Apontador da Pilha de Software
   ac_reg X0;     //Registrador de Entrada da Unidade Aritmética

   ARCH_CTOR(dsp56F827) {
      ac_isa("dsp56F827_isa.ac");
      set_endian( "big" );
   };
};
```

Figura 4.2: Descrição da Arquitetura Motorola DSP56F827 em ArchC.

A descrição da arquitetura inicia com a declaração do módulo *AC_ARCH*. Este módulo deve conter, obrigatoriamente, o tamanho da palavra da arquitetura (*ac_wordsize*), no

caso do DSP56F827 é 16 bits, e a declaração do tamanho da memória que é utilizada para programas e dados.

Apesar da arquitetura do processador DSP56F827 seguir o padrão *Harvard*, com a memória de dados e programa separados, no ArchC esta modelagem não é possível devido a falta de métodos para manipular constantes em memória. Por exemplo, em programas que contém constantes inicializadas estaticamente, o ArchC armazena estes dados na área de código ao invés de armazenar na área de dados, inviabilizando o acesso dos mesmos posteriormente. Para corrigir este problema e compartilhar a memória entre dados e programa foi adicionado no arquivo *nomedoprocessador-isa.cpp* um valor de deslocamento ao cálculo de índice para os acessos à memória de dados. Este valor de deslocamento pode ser alterado pelo programador no arquivo de semântica conforme a necessidade da aplicação. Este problema foi observado ao simular arquivos no formato hexadecimal ao invés de simular arquivos binários (vide Seção 5.1). Isto ocorre porque no formato binário, o compilador organiza a memória separando dados e programa e o método de leitura do arquivo no ArchC obedece esta organização.

Além do tamanho da palavra e do tamanho da memória, também devem ser definidos os registradores da arquitetura. Todos os registradores declarados como *ac_reg* possuem o mesmo tamanho de palavra da arquitetura (definido na palavra-chave *ac_wordsize*). Os registradores podem ser criados diretamente no arquivo da semântica. Neste caso não serão contabilizados os acessos à estes recursos pelo ArchC. Os recursos definidos neste arquivo possuem métodos de leitura e escrita. Por exemplo, o registrador X0 pode ser acessado através das funções *X0.read()* e *X0.write("valor")*.

Quando um registrador é subdividido em campos, como é o caso do *Registrador de Estado* pode-se utilizar a palavra-chave *ac_format* para descrever o formato do recurso, como mostra a Figura 4.3 que contém a descrição da arquitetura do Atmega8515. O formato do registrador SREG é definido por STAT_F. Neste caso, o registrador só pode ser acessado através de seus campos e não como um registro único. Por este motivo, alguns registradores do DSP56F827 não puderam ser declarados neste arquivo, como o registrador Y de 32 bits que pode ser utilizado como registrador único ou subdividido em

dois registradores de 16 bits, *Y0* e *Y1*. Os acumuladores *A* e *B* de 36 bits também foram declarados diretamente no arquivo de semântica.

```
AC_ARCH(atmega8515){
   /* Descricao de Memória */
   ac_mem DM:64k;
   /* Tamanho da palavra */
   ac_wordsize 16;

   /* Descricao dos registradores */
   ac_format STAT_F = "%I:1 %T:1 %H:1 %S:1 %V:1 %N:1 %Z:1 %C:1";

   ac_regbank RB:32; //Banco de registradores
   ac_reg <STAT_F>SREG; //Registrador de Estado
   ac_reg SP; //Apontador de Pilha

   ARCH_CTOR(atmega8515){
      ac_isa("atmega8515_isa.ac");
      set_endian("big");
   };
};
```

Figura 4.3: Descrição da Arquitetura Atmega8515.

Após a descrição dos recursos da arquitetura é declarado o construtor *AC_CTOR*, que deve conter o nome do arquivo com a sintaxe das instruções e o formato de endereçamento do processador, que nos três processadores é *big endian*.

Além das características acima, o arquivo de descrição da arquitetura pode conter a estrutura dos segmentos e da hierarquia de memória, quando for o caso. Para este trabalho estas configurações não foram utilizadas.

4.1.2 Descrição da Sintaxe das Instruções

A Figura 4.4 mostra um trecho da descrição da sintaxe das instruções do processador DSP56F827. O nome do arquivo deve ser *nomedoprocessador_isa.ac*.

No segmento *AC_ISA* são declarados os formatos e os campos dos *opcodes*. Utilizando a mesma palavra-chave *ac_format* define-se o formato de uma ou mais instruções. Por exemplo, o tipo *Type_1W* é formado por um único campo chamado *op4* de tamanho de 16 bits. O *Type_2W* descreve instruções de 32 bits com *opcode* mais um número imediato

```
AC_ISA(dsp56F827){
   // Ex: add A,B
   ac_format Type_1W  = "%op4:16";
   // Ex: inc X:0000
   ac_format Type_2W = "%op4:16 %imm16:16";
   // Ex: bfclr #0001,X0
   Type_2W_Bit1 = "%op2:8 %mod:3 reg:5 %imm16:16";

   ac_instr<Type_1W> add_b_a, add_a_b, tfr_b_a, tfr_a_b;
   ac_instr<Type_2W> dec_imm16, inc_imm16;
   ac_instr<Type_2W_Bit1> bfclr_imm_d5,
                            bfset_imm_d5, bfchg_imm_d5;

   ISA_CTOR(dsp56F827){
    add_b_a.set_asm("add B,A");            //A = B + A
    add_b_a.set_decoder(op4=0x6400);
    add_b_a.set_cycles(2);

    tfr_b_a.set_asm("tfr B,A");           //A = B
    tfr_b_a.set_decoder(op4=0x6c00);
    tfr_b_a.set_cycles(2);

    bfclr_imm_d5.set_asm("bfclr %imm16,%d5");
    bfclr_imm_d5.set_decoder(op2=0x81,mod=0x6);
    bfclr_imm_d5.set_cycles(4);
    ...
   };
  };
```

Figura 4.4: Descrição do Conjunto de Instruções do DSP56F827 em ArchC (parte).

de 16 bits.

Em seguida, são identificadas as instruções que seguem o formato especificado, como as instruções *add_b_a e tfr_b_a* que possuem o formato descrito em *Type_1W*.

No construtor *ISA_CTOR* são especificadas as características de cada instrução. A palavra-chave *set_asm* especifica o formato da instrução em *Assembly*. A linha seguinte, *set_decoder*, inicializa a seqüência de decodificação preenchendo os campos fixos do *opcode* declarados no formato. No caso da instrução *add_b_a*, o campo *op4*, descrito no formato Type_1W, é preenchido com o valor 0x6400.

A última configuração da instrução refere-se ao número de ciclos necessários para a sua execução. A palavra-chave *set_cycles* deve receber o valor especificado para cada instrução, disponível na documentação do microprocessador. A descrição do DSP56F827 na Figura 4.4 é para um simulador multi-ciclo: a linha `add_b_a.set_cycles(2)` define que a instrução `add A,B` dura 2 ciclos. Evidentemente, a precisão das medidas de tempo

depende do nível de detalhe com que o modelo é elaborado.

4.1.3 Descrição da Semântica das Instruções

Uma vez descritas a organização e o conjunto de instruções, os arquivos são submetidos ao pré-processamento do ArchC, que cria todos os arquivos necessários para a criação de um simulador interpretado, além de montar a estrutura do arquivo que deve conter a implementação das instruções. O comando para executar esta tarefa é: *acsim nomedo-processador.ac -s*. A opção "-s" habilita a geração de estatísticas.

Todas as instruções declaradas no arquivo de sintaxe devem ter seu comportamento descrito pelo programador no arquivo de semântica (*nomedoprocessador-isa.cpp*). A Figura 4.5 mostra a implementação da função *OR* no processador DSP56F827.

```
//!Instruction or_a1_x0 behavior method.
void ac_behavior( or_a1_x0 )
{
  int res = 0;
  switch (cycle)
  {
    case 1:
      printf("@@@ OR a1 x0 @@@\n");
      //Efetua a operação lógica
      res = A1 | X0.read();
      //Atualiza o SR
      SR_ccr_v = 0;
      SR_ccr_z = (res == 0);
      SR_ccr_n = (res >> 15);
      X0.write(res & 0x0000ffff);
      break;
    case 2:
      break;
  }
  ac_cycle++;
}
```

Figura 4.5: Descrição do Comportamento de uma Instrução (semântica) do DSP56F827.

Na implementação multi-ciclo emprega-se uma cláusula *switch-case* para manter o simulador executando a instrução até o término dos ciclos especificados no arquivo de sintaxe. A variável de controle *ac_cycle* contabiliza os ciclos executados. O restante da

implementação da instrução resume-se ao comportamento propriamente dito e à atualização do registrador de estado.

Além da implementação das instruções, é responsabilidade do programador incrementar o *contador de programa* (PC). Na Figura 4.6 é mostrado a função que incrementa o PC do simulador para o Rabbit R2000. A variável *ac_pc* é interna ao ArchC, enquanto a variável *pc* é local ao programa *nomedoprocessador-isa.cpp* e armazena temporariamente o valor do PC. O contador de programa só é efetivamente incrementado após a execução da instrução no número de ciclos pré-determinado.

```
//!Generic instruction behavior method.
void ac_behavior( instruction )
{
  if( ac_cycle == 1 )
    pc += ac_instr_size/8;
  if( ac_cycle == get_cycles())
  {
    ac_pc = (unsigned int)pc;
  }
}
```

Figura 4.6: Incremento do Contador de Programa no R2000.

ArchC indexa a memória de programa em bytes, logo é necessário dividir o tamanho da instrução (*ac_instr_size*) por oito para saber quantas vezes o PC será incrementado. Por este mesmo motivo os modelos dos microprocessadores DSP56F827 e Atmega8515 tiveram de ser adaptados para permitir a execução dos programas de teste que são gerados pelas ferramentas *CodeWarrior* e *AVR Studio* respectivamente, e indexam a memória em palavras de 16 bits. Para isso, toda atribuição de endereço na implementação das instruções é multiplicada por dois. Por exemplo, se é necessario a execução de um salto para o endereço 0x10, considerando a indexação em 16 bits, na verdade será executado um salto para o endereço 0x20 em bytes.

4.2 Implementação das Métricas de Memória

Com o objetivo de complementar o conjunto de métricas disponibilizado pela linguagem ArchC e oferecer uma ferramenta mais completa para os projetistas, foram implementadas neste trabalho duas métricas relacionadas à quantidade de memória utilizada pela aplicação. As métricas retornam a quantidade de memória, em bytes, do código (estático) e dos dados (estático e dinâmico). Esta foi uma contribuição inédita do trabalho ao conjunto de funcionalidades da ArchC.

Para incluir a métrica de utilização de memória foi necessário a alteração do código fonte do ArchC, nos arquivos *ac_stats.H*, *ac_stats.cpp* e *ac_storage.cpp*.

Como a linguagem já produz um arquivo com estatísticas de número de ciclos/instruções executados, número de acessos à memória e às instruções, as métricas de utilização de memória são geradas a partir de duas novas variáveis, *im_length*, que armazena a utilização da memória de programa, e *dm_length* que verifica a utilização da memória de dados.

A variável *im_length* é incrementada no método de leitura do arquivo executável/hexadecimal da aplicação, no método *load* da classe *ac_storage*, e essencialmente conta o número de bytes do programa, como mostra o trecho de código da Figura 4.7.

```
#ifdef AC_STATS
  //Tamanho do programa
  ac_sim_stats.im_length = program_size - 8; //Retira o incremento anterior
#endif
```

Figura 4.7: Código que Computa o Tamanho da Memória de Programa.

A variável *dm_length* conta os bytes de memória de dados estática e dinâmica do programa. A quantidade de dados utilizados pelo programa é medida nos métodos de escrita na memória, *write* e *write_byte* ambos pertencentes a classe *ac_storage*.

Para evitar que escritas sucessivas no mesmo endereço de memória sejam contabilizados, um vetor mantém um histórico de todos os acessos a memória. Quando uma escrita na memória é efetuada, é verificado se este endereço já foi acessado. Se sim, a variável

dm_length não é incrementada; senão, a variável é incrementada e o acesso é registrado no vetor de histórico. A Figura 4.8 mostra um trecho de código que executa a verificação na gravação de dados de 16 bits.

```
if (mem_control[address] == 0)
{
   ac_sim_stats.dm_length += 2;
   mem_control[address] = 1;
   mem_control[address+1] = 1;
}
```

Figura 4.8: Trecho do Código que Computa o Tamanho da Memória de Dados.

Os valores medidos para estas métricas são armazenados no arquivo *nomedoprocessador.stats*, que além destas, apresenta outras métricas de desempenho geradas pelo ArchC, como tempo de execução, número de ciclos executados, dentre outras. Este arquivo só é gerado se a opção "-s" for executada durante a geração do simulador.

O método utilizado para a medição da memória de dados é coerente para o caso de sistemas embarcados, mas pode ser impraticável para sistemas de uso geral que manipulam grandes conjuntos de dados, já que é necessário que o tamanho do vetor seja igual ao tamanho da memória de dados disponível para a aplicação. Já o método utilizado para a medição da memória de programa poderia ser utilizado, sem custos adicionais, em sistemas de uso geral.

CAPÍTULO 5

AVALIAÇÃO DE DESEMPENHO

Neste capítulo são apresentados os resultados obtidos com uso dos simuladores. As simulações descritas adiante foram úteis para a aferição da qualidade dos modelos bem como para a depuração dos mesmos. As seções que se seguem descrevem o ambiente de testes, os resultados obtidos, a precisão do modelo DSP56F827 e um estudo de caso, respectivamente.

5.1 Descrição do Ambiente de Testes

Os modelos foram validados através da execução dos programas de teste disponibilizados pelo *Dalton Project* [24], programas clássicos de ordenação e um gerador de números primos, todos descritos sucintamente na Tabela 5.1. Os programas do *Dalton Project* foram utilizados pela equipe de desenvolvedores do ArchC para testar o modelo do microprocessador Intel 8051.

Programa	**Descrição**
negcnt.c	Conta 1000 números negativos
int2bin.c	Traduz um número de 16 bits (hexa) para binário
gcd.c	Compara, soma e subtrai 2 números
cast.c	Transforma `long` em `char`
fib.c	Gera 100 elementos da Série de Fibonacci
bubble.c	Ordena 1000 números pelo Bubble Sort
insertion.c	Ordena 1000 números pelo Insertion Sort
quick.c	Ordena 1000 números pelo Quick Sort
selection.c	Ordena 1000 números pelo Selection Sort
shell.c	Ordena 1000 números pelo Shell Sort
crivo.c	Gera números primos < 1000 pelo Crivo de Eratóstenes

Tabela 5.1: Programas de Teste.

ArchC disponibiliza funções capazes de efetuar chamadas ao sistema operacional (*syscalls*), que permitem a simulação de programas que contém operações de entrada/saída, como leitura e escrita em arquivos. Para habilitar esta funcionalidade é necessário a

configuração do compilador GCC para gerar código específico para o ArchC (*compilação cruzada*) [2]. Como somente o processador Atmega8515 pode ser compilado pelo GCC e, dessa forma habilitar as funções de chamada ao sistema, decidiu-se pela utilização de arquivos que não contivessem operações de entrada/saída. Caso contrário, não seria possível a comparação com os modelos para os quais não existe uma versão do GCC.

Os testes foram executados num PC com sistema operacional Linux *Slackware 10.1*, com ArchC versão 1.6 e GCC versão 3.3.6 (ocorrem erros de compilação com as versões 3.4.∗ que foram reportados à equipe de desenvolvimento). Os arquivos de testes foram gerados pelos compiladores *Metrowerks Code Warrior 7.2* [19], *AVR Studio 4.0* [7] e *Softools WinIde 1.58.03* [52] para os processadores da Motorola, da Atmel e da Rabbit, respectivamente.

Os simuladores podem executar aplicações usando dois formatos, hexadecimal e binário. A Figura 5.1 mostra o formato padrão do arquivo hexadecimal aceito pelo ArchC. O arquivo é delimitado pelas seções *.text* e *.data*, que contém o programa e os dados, respectivamente. O formato binário segue o padrão *executable and linking format* (ELF) , mas deve seguir algumas regras do ArchC como, (i) iniciar o programa, obrigatoriamente, no endereço zero e (ii) reservar os endereços 0x40 a 0xFF para utilização das *syscalls*.

As saídas das ferramentas de desenvolvimento foram modificadas para aceitar o padrão hexadecimal do ArchC. As mudanças consistem em multiplicar os endereços por dois, já que o ArchC acessa a memória em bytes e o DSP56F827 e o Atmega8515 acessam a memória em palavras de 16 bits. Além disso, todos os opcodes devem ser separados de acordo com a palavra do processador. Por exemplo, o opcode da instrução *lea* é 32 bits (0xDE40FC19) mas só é executado pelo simulador se for separado em duas palavras de dois bytes (0xDE40 e 0xFC19).

O formato *elf* não foi utilizado pois inviabiliza a alteração dos endereços de um para dois bytes. Tanto a ferramenta CodeWarrior como o AVR Studio produzem a saída da compilação no formato *elf*.

Para produzir os arquivos de teste para o microprocessador Rabbit R2000 foi ne-

```
.text:
00000000 DE4B 03E8; lea (SP+0x03e8)
00000004 880F; move SP,R0
00000006 E040; nop
00000008 DE40 FC19; lea (R0-0x03e7)
0000000c 87D1 2000; movei #1000,R1
00000010 87C1 03E8; movei #1000,Y0
00000014 F001; move X:(R1)+,X0
00000016 D000; move X0,X:(R0)+
00000018 6413; decw Y0
0000001a A67C; bgt *-3
0000001c 8A0F; move SP,R2
0000001e E040; nop
00000020 DE42 FC19; lea (R2-0x03e7)
00000034 87C1 03E8; movei #1000,Y0
0000003c E040; nop
0000003e DE4B FC18; lea (SP-0x03e8)
00000042 E044 stop
.data:
0xf000 0x4c72 0x5e9e 0x4fde 0x5249 0x0c0f 0x4276 0x45a6 0x23c7 0x37b1 0x79cb
0xf014 0x08cd 0x7800 0x4812 0x212d 0x7d4b 0x24bf 0x7de8 0x4cb5 0x33ad 0x7f39
0xf028 0x4406 0x55f5 0x7e8f 0x49d5 0x1731 0x1ca0 0x63da 0x32bd 0x0fc3 0x25fa
0xf03c 0x65be 0x2f29 0x6902 0x46fe 0x5f6e 0x3f66 0x223d 0x4551 0x719f 0x67a8
```

Figura 5.1: Exemplo de um Arquivo de Teste em Formato Hexadecimal.

cessário, além do ambiente Softools WinIDE, um módulo de hardware para conectar-se ao ambiente de desenvolvimento. O módulo utilizado foi o *Rabbit 2000 TCP/IP Develop Board*. Devido a incompatibilidade de versões entre o ambiente de desenvolvimento e o módulo de hardware disponível, não foi possível a geração de todos os testes descritos acima. Quando são executados programas que utilizam mais memória de dados, o ambiente perde a conexão com o módulo de hardware, e conseqüentemente não finaliza a compilação.

5.2 Comparação entre os Processadores

As Tabelas 5.2 e 5.3 mostram os valores obtidos nas simulações com os modelos do DSP56F827 (DSP), Atmega8515 (AT) e R2000 (RB). A Tabela 5.2 mostra a utilização da memória de dados (Dados) e o tamanho estático do código (Código). A Tabela 5.3 indica o número de ciclos dispendidos (# Ciclos) e o número de instruções executadas (# Instruções).

A Figura 5.2 mostra a utilização dinâmica de memória de dados dos processadores

Programa	Dados [bytes]			Código [bytes]		
	DSP	AT	RB	DSP	AT	RB
negcnt	4	5	6	18	134	40
int2bin	10	8	7	68	190	63
gcd	12	10	14	56	206	78
cast	20	19	20	172	246	124
fib	212	215	215	116	414	194
bubble	4010	4023	–	160	668	–
insertion	4010	4017	–	160	498	–
quick	4278	4387	–	338	804	–
selection	4014	4019	–	166	540	–
shell	4014	4019	–	200	590	–
crivo	2018	2010	–	1880	612	–

Tabela 5.2: Utilização de Memória.

Programa	# Ciclos			# Instruções		
	DSP	AT	RB	DSP	AT	RB
negcnt	18022	24064	81088	5006	14060	15016
int2bin	1014	1055	2614	359	1055	581
gcd	188200	199784	521909	55608	119041	89838
cast	344	287	554	100	199	86
fib	7828	13121	31407	2593	8230	6866
bubble	$27{,}2 \cdot 10^6$	$54{,}8 \cdot 10^6$	–	$9{,}6 \cdot 10^6$	$35{,}2 \cdot 10^6$	–
insertion	$14{,}5 \cdot 10^6$	$21{,}3 \cdot 10^6$	–	$5{,}1 \cdot 10^6$	$13{,}6 \cdot 10^6$	–
quick	875233	$1 \cdot 10^6$	–	242197	625292	–
selection	$18{,}1 \cdot 10^6$	$30 \cdot 10^6$	–	$6 \cdot 10^6$	$19{,}2 \cdot 10^6$	–
shell	$5{,}97 \cdot 10^6$	$9{,}46 \cdot 10^6$	–	$2 \cdot 10^6$	$6{,}1 \cdot 10^6$	–
crivo	$12{,}6 \cdot 10^6$	$20{,}3 \cdot 10^6$	–	$3{,}4 \cdot 10^6$	$15 \cdot 10^6$	–

Tabela 5.3: Número de Ciclos e Instruções Executadas

quando executam os programas de teste. No gráfico, o eixo das ordenadas está em escala logarítmica. Nota-se que a utilização é praticamente a mesma para os três processadores, já que todos os programas utilizam números inteiros. Caso fossem utilizados vetores do tipo `char`, o DSP56F827 apresentaria desvantagem pois o compilador deste processador considera que o tamanho do `char` é 16 bits, o que duplicaria sua utilização de memória.

A Figura 5.3 compara tamanho do código (estático) gerado para os processadores. No gráfico, o eixo das ordenadas está em escala logarítmica. Esta medida é relacionada à eficiência do conjunto de instruções e do compilador, que é capaz de gerar código compacto a partir de um programa escrito em C. Neste caso, os pares processador-compilador da Motorola e da Rabbit são mais eficientes: a relação de tamanho de código entre os compi-

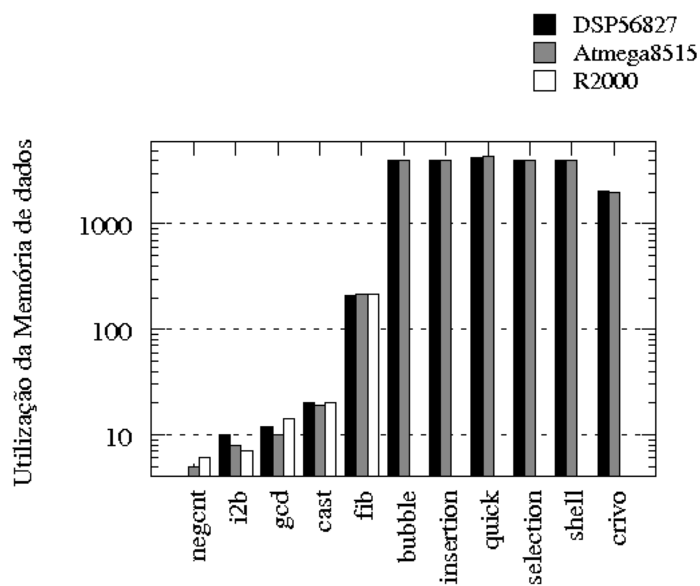

Figura 5.2: Utilização da Memória de Dados.

ladores/processadores Atmega e DSP57827 varia de 1,4 a 7,4 a favor do DSP57827 —*com estes programas simples de teste.*

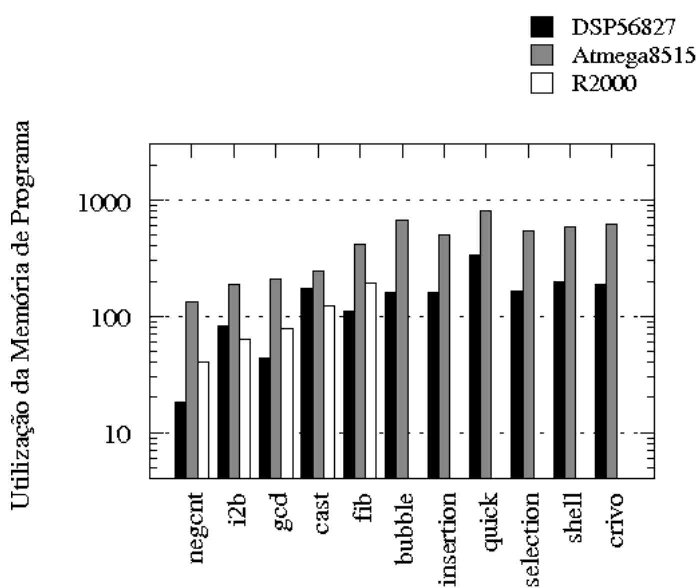

Figura 5.3: Utilização da Memória de Programa.

As Figuras 5.4 e 5.5 mostram os resultados para número de ciclos e número de instruções executadas. Comparando-se o número de ciclos *com estes programas de teste*, o processador da Motorola executa 1,2 vezes mais ciclos que o Atmega somente com o pro-

grama *cast*. Nos demais, o Atmega executa 1,23 a 1,62 vezes mais ciclos que o Motorola. Já o R2000 é consideravelmente mais lento, executa cerca de quatro vezes mais ciclos que o DSP56F827. Isso ocorre porque, ao contrário do DSP56F827 e do Atmega8515, o R2000 não é segmentado, e cada instrução consome de 2 a 15 ciclos para ser executada. Note que o eixo vertical dos gráficos é logarítmico. Quanto ao número de instruções executadas, o Atmega executa de 1,24 a 1,50 mais instruções que o Motorola. A eficiência do código gerado para o Rabbit depende do programa, sendo que o tamanho fica entre aqueles dos outros dois processadores, exceto para *negcnt*, no qual o R2000 possui o maior executável dentre os três.

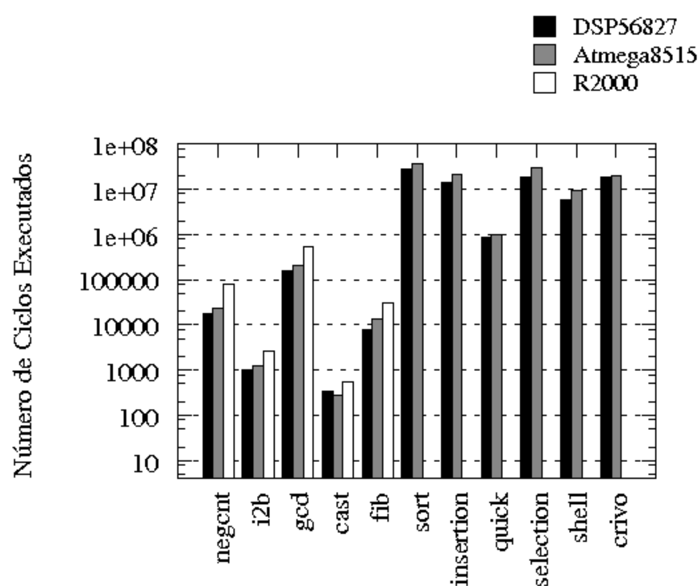

Figura 5.4: Número de Ciclos Executados.

Esta análise preliminar dos resultados indica que para os programas com mais dados ("maiores"), o processador da Motorola possui um desempenho substancialmente melhor, mas para os programas "menores" fica evidente o desperdício de recursos ao utilizar este processador. Contudo, deve-se considerar que a medida de número de ciclos executados *não reflete o tempo de execução* destes programas porque as faixas de freqüência dos três processadores são *muito diferentes*: a freqüência máxima do Atmega é 16 MHz, enquanto que a freqüência máxima do Motorola é 80 MHz. A decisão por um ou por

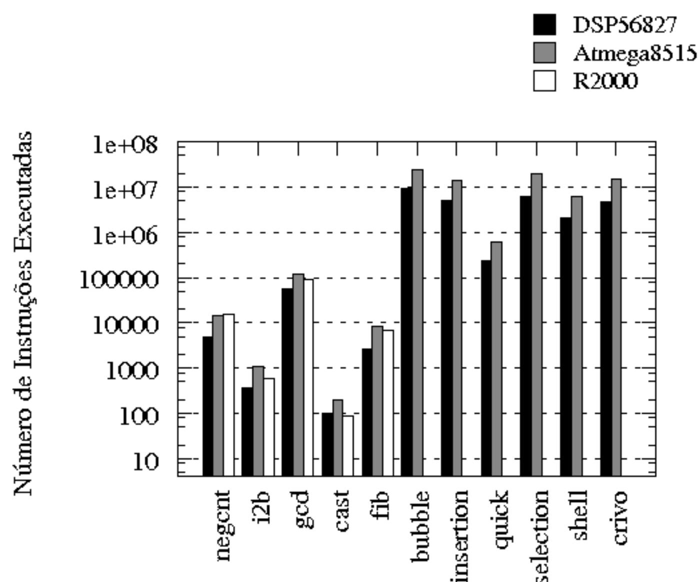

Figura 5.5: Número de Instruções Executadas.

outro processador deve, necessariamente, considerar a natureza e os requisitos do projeto —principalmente o custo e o tempo de resposta dos aplicativos.

Outro ponto que deve ser considerado refere-se à memória interna do processador, principalmente quanto a esta permitir a execução da aplicação sem a necessidade de inclusão de memória externa. Os custos da adição de memória externa a um processador mais simples/barato podem ultrapassar o custo de um processador mais complexo e com mais memória interna.

5.3 Precisão do Modelo do DSP56F827

Com o objetivo de avaliar a precisão dos simuladores foram executadas comparações entre o simulador e um módulo de hardware do processador Motorola DSP56F827. A Tabela 5.4 mostra os resultados para o tempo de execução, medido em ciclos. A coluna marcada "S/H" mostra a razão entre os valores produzidos pelo simulador (SIM) e as medidas no processador (HW).

O processador executa até 15% mais ciclos que o estimado pelo simulador. Isto ocorre porque o simulador ignora alguns dos ciclos adicionais relacionados com desvios. Para corrigir esta deficiência, é necessário escrever um modelo mais detalhado do processador

Programa	# Ciclos		
	SIM	HW	S/H
negcnt	18022	20036	0,90
int2bin	2144	2566	0,84
gcd	162542	171110	0,95
cast	344	422	0,82
fib	74122	86022	0,86
bubble	$27{,}2\cdot10^6$	$32{,}8\cdot10^6$	0,83
insertion	$14{,}5\cdot10^6$	$17{,}6\cdot10^6$	0,82
quick	875233	897174	0,97
selection	$18{,}1\cdot10^6$	$22{,}1\cdot10^6$	0,81
shell	$5{,}9\cdot10^6$	$6{,}7\cdot10^6$	0,88
crivo	$12{,}6\cdot10^6$	$14{,}6\cdot10^6$	0,86

Tabela 5.4: Comparação entre Modelo e Processador DSP56F827.

que simule exatamente o comportamento dos desvios. A versão segmentada dos modelos está sendo desenvolvida por outros membros do grupo de pesquisa e a expectativa é que a precisão das medidas seja melhor do que 95%.

5.4 Estudo de Caso - O Conversor Serial-IP

Umas das motivações para o desenvolvimento deste trabalho foi a participação no projeto do *Conversor Serial-IP* (CSI) [60], em execução no Lactec [39]. O CSI consiste de uma plataforma de hardware, que utiliza o processador Motorola DSP56F827, e um módulo de comunicação celular com a tecnologia *General Packet Radio Services* (GPRS). Este dispositivo é conectado a uma Unidade Terminal Remota (UTR) e funciona como uma ponte entre a UTR e o Centro de Operação da Distribuição de Energia (COD). A UTR e o COD comunicam-se através do protocolo serial DNP3 [26] e utilizam o CSI para enviar as mensagens pela Internet sobre o protocolo TCP/IP [20]. A Figura 5.6 mostra o fluxo de informações entre a Remota e a Central.

Devido ao crescimento da utilização de protocolos de comunicação via Internet em sistemas embarcados e ao gargalo de desempenho que estes protocolos provocam, foi efetuado um teste com a pilha TCP/IP/PPP, utilizada no projeto CSI, com o objetivo de comparar os simuladores dos processadores DSP57F827 (DSP) e Atmega8515 (AT) com o hardware do projeto CSI (baseado no DSP56F827).

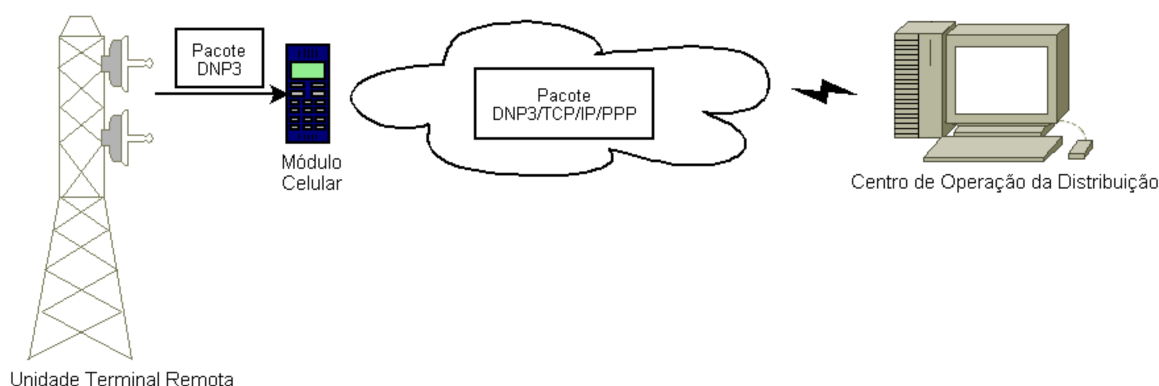

Figura 5.6: Estrutura do Projeto Serial-IP.

O teste consiste em empacotar e desempacotar um conjunto de dez inteiros utilizando os protocolos TCP/IP e PPP[1], sem considerar as funções de conexão e retransmissão de dados. A Tabela 5.5 mostra os resultados obtidos.

Métricas	HW	DSP	AT
# Instruções	7132	7239	16160
# Ciclos	33382	27830	24077
Dados [bytes]	–	666	634
Código [bytes]	4282	4404	7660

Tabela 5.5: Resultado Obtido com um Trecho da Pilha TCP/IP/PPP.

Os resultados seguem o padrão dos testes anteriores, o simulador atinge 85% de precisão na métrica de número de ciclos em relação ao hardware. O tamanho do programa em bytes e o número de instruções executadas pelo Atmega8515 é 1,7 e 2,24 vezes, respectivamente, maior que o DSP56F827. Isto se deve a eficiência do conjunto de instruções do processador da Motorola. A utilização da memória de dados praticamente não se altera entre os dois processadores. O número de ciclos executados no Atmega8515 é menor que no DSP56F827, devido a esta aplicação utilizar as instruções mais simples do Atmega8515, que consomem apenas um ciclo de relógio.

A medida do consumo de memória de dados não é explícita na ferramenta *CodeWarrior*, que gerencia o módulo de hardware do CSI. Ao invés disso o *CodeWarrior* mostra o número de acessos à memória RAM que é aproximadamente 13440 acessos. A ferramenta ArchC disponibiliza esta informação no arquivo de estatísticas, e para a pilha

[1] O protocolo *Point-to-Point Protocol* (PPP) é utilizado para efetuar a conexão GPRS para transmissão de dados via através da rede de celular.

TCP/IP/PPP utilizando o simulador DSP56F827 fica em torno de 15028 acessos, cerca de 89% a mais que o medido no módulo de hardware.

De acordo com os testes, a escolha do processador DSP56F827 para o projeto CSI foi a mais acertada, pois o mesmo possui o melhor desempenho em termos de tempo de execução e utilização de memória. O Atmega8515 só poderia ser utilizado para esta aplicação com a inserção de memória externa, já que apenas um trecho da pilha TCP/IP utilizou 95% da memória Flash interna.

Além disso, o microprocessador da Motorola possui um conversor Analógico-Digital que possibilita, entre outras coisas, a utilização do processador como medidor de qualidade de energia elétrica.

CAPÍTULO 6

CONCLUSÃO E TRABALHOS FUTUROS

O mercado de desenvolvimento de aplicações embarcadas sofre de alguma carência de ferramentas que auxiliem na redução de custos através de melhorias no aproveitamento dos recursos. Além disso, é indiscutível a utilidade de ferramentas que permitam que a codificação dos aplicativos possa ser iniciada antes da conclusão do hardware, reduzindo-se assim o tempo de realização do projeto.

Ao utilizar o ambiente de avaliação de desempenho de processadores descrito aqui, o projetista é capaz de escolher com segurança o microprocessador ou microcontrolador que melhor se adapta a sua aplicação, nos quesitos de custo, capacidade de processamento e quantidade de memória utilizada. Além disso, os simuladores podem ser utilizados para antecipar a codificação dos aplicativos antes que haja hardware disponível. A versão atual do simulador do Motorola DSP56F827 possui precisão melhor que 85%, nas medidas de tempo de execução, e 89% a 97% nas medidas de utilização de memória quando comparado com uma implementação em hardware. Isto permite um alto grau de confiabilidade na escolha do processador.

O custo de desenvolvimento de um simulador utilizando a ferramenta ArchC depende, exclusivamente, da complexidade da arquitetura e do conjunto de instruções do processador. O desenvolvimento do simulador para o DSP56F827, por exemplo, foi bastante custoso e consumiu cerca de cinco meses para ser concluído. Já os modelos Atmega8515 e R2000 levaram cerca de dois meses cada um para serem implementados. Pode-se concluir que a construção de novos modelos para a avaliação de aplicações deve ser considerada apenas se o modelo não for muito complexo, caso contrário, grande parte do tempo do projeto será gasto na construção dos simuladores. Contudo, estes podem ser re-usados em outros projetos.

Os testes realizados permitiram a comparação dos microprocessadores em relação ao

tempo de execução e memória utilizada. O processador DSP56F827 apresentou a melhor relação custo/benefício, devido principalmente à eficiência do seu conjunto de instruções que possibilita uma melhor utilização da memória de programa. A eficiência do conjunto de instruções aliada ao fato de que cada instrução necessita em média dois ciclos de relógio para executar faz com que o DSP56F827 seja a melhor opção para aplicações de "médio" e "grande" porte.

A avaliação dos resultados também permite concluir que no momento da escolha do microprocessador para um projeto não basta apenas o levantamento das características básicas do processador, como quantidade de memória e freqüência do relógio, é necessária uma avaliação mais cuidadosa da arquitetura, como verificação de segmentação e do conjunto de instruções. Por exemplo, o processador Atmega8515 disponibiliza uma quantidade razoável de memória e é mais barato, porém os programas gerados ocupam bastante memória de programa, o que pode requerer inclusão de memória externa, aumentando o custo e diminuindo o desempenho do sistema final.

A linguagem de descrição de arquiteturas ArchC mostrou-se uma ferramenta robusta e de fácil utilização. Durante a implementação alguns problemas foram encontrados e contornados. Seguem como sugestão de melhorias da ferramenta: (i) possibilidade de configuração de arquitetura *Harvard* de memória, (ii) possibilidade de criação de registradores com tamanho maior que a palavra da arquitetura e (iii) acesso direto a um registrador formatado em campos.

Dentre as atividades previstas para o futuro próximo estão a conclusão dos testes com o microprocessador Rabbit R2000 com a correção do problema entre o ambiente de desenvolvimento e o módulo de hardware, a modelagem da versão segmentada dos três processadores, e a coleta e avaliação de um conjunto maior de programas de teste. Com estas tarefas concluídas, será possível efetuar comparações precisas do desempenho dos microprocessadores modelados.

As versões dos modelos com segmentação já estão em fase de desenvolvimento por outros membros da equipe de pesquisa e a expectativa é de que os modelos segmentados apresentem precisão da ordem de ±95% com relação a valores medidos diretamente nos

processadores.

Este trabalho possibilitou a constatação do imenso espaço arquitetural que envolve o projeto de sistemas embarcados e de quanto trabalho ainda é necessário para auxiliar os desenvolvedores. Muitas ferramentas são necessárias, como simuladores, compiladores, montadores, dentre outros, principalmente disponibilizados como software livre. Também é importante a concientização por parte da comunidade desenvolvedora para a utilização de metodologias mais eficientes de especificação e modelagem de sistemas tendo como objetivo a redução do tempo de desenvolvimento e a redução do desperdício de recursos.

As versões multi-ciclos dos simuladores serão enviadas para avaliação da equipe de desenvolvimento ArchC para possível disponibilização dos modelos na página do projeto.

Recentemente foi submetido um artigo, descrevendo este trabalho, entitulado "Simulação de Sistemas Embarcados utilizando ArchC" ao *VII Workshop em Sistemas Computacionais de Alto Desempenho* (WSCAD 2006) [11], o qual foi aceito.

APÊNDICE A

CÓDIGO FONTE DO SIMULADOR DSP56F827

Arquivo de descrição da arquitetura, *dsp56827.ac*:

```
AC_ARCH(dsp56F827){
   //Tamanho da palavra em bits
   ac_wordsize 16;
   //Tamanho da memória
   ac_mem DM:128k; //Considerando a memória externa
   //Descrição dos Registradores
   ac_reg SR;     //Registrador de Estado
   ac_reg OMR;    //Registrador de Modo de Operação
   ac_reg LC;     //Contador de Loop
   ac_reg LA;     //Endereçador de Loop
   ac_reg HWS0;   //Apontador da Pilha de Hardware
   ac_reg HWS1;   //Apontador da Pilha de Hardware
   ac_reg M01;    //Modificador de Cálculo de Endereço
   ac_reg N;      //Registrador de Deslocamento
   ac_reg SP;     //Apontador da Pilha de Software
   ac_reg X0;     //Registrador de Entrada da Unidade Aritmética

   ARCH_CTOR(dsp56F827) {
      ac_isa("dsp56F827_isa.ac");
      set_endian( "big" );
   };
};
```

Trecho do arquivo de descrição da sintaxe das instruções, dsp56827_isa.ac (limitado a quatro instruções):

```
AC_ISA(dsp56827){
  ac_format Type_1W  = "%op4:16";
  ac_format Type_Jsr = "%op2:8 %dif:2 %a0:1 dc:3 %a1_2:2 %addr16:16";
  ac_format Type_2W_Bit2  = "%op2:8 %dif:2 %pp6:6 %imm16:16";
  ac_format Type_Dalu3op2 = "%op2:8 %f0:1 %qqq:3 %mod2:2 %f1f2:2";

  ac_instr<Type_1W> add_b_a, rts;
  ac_instr<Type_Jsr> jsr;
  ac_instr<Type_2W_Bit2> move_r2_pp6;
  ac_instr<Type_Dalu3op2> impy_dalu3op2;

  ISA_CTOR(dsp56827){
    add_b_a.set_asm("add B,A");
    add_b_a.set_decoder(op4=0x6400);
    add_b_a.set_cycles(2);

    jsr.set_asm("jsr %addr16");
    jsr.set_decoder(op2=0xe9,dif=0x3,dc=0x2);
    jsr.set_cycles(8);

    move_r2_pp6.set_asm("move %imm16,X:(R2+<<%pp6)");
    move_r2_pp6.set_decoder(op2=0xa6,dif=0x2);
    move_r2_pp6.set_cycles(6);

    impy_dalu3op2.set_asm("impy %qqq,%f0");
    impy_dalu3op2.set_decoder(op2=0x66,mod2=0x3);
    impy_dalu3op2.set_cycles(2);

    rts.set_asm("rts");
    rts.set_decoder(op4=0xedd8);
    rts.set_cycles(10);
```

```
  };
};
```

Trecho do arquivo de descrição da semântica das instruções, dsp56827-isa.cpp (limitado a quatro instruções):

```
#include  "dsp56827-isa.H"
#include  "ac_isa_init.cpp"
//------------------global var declarations------------------------------
int pc = 0; //Program Counter
//---------------------------------------------------------------------
//!Behavior executed before simulation begins.
void ac_behavior( begin )
{
    printf("@@@ begin behavior @@@\n");
    SP = 0x1000;
};
//!Behavior executed after simulation ends.
void ac_behavior( end )
{
    printf("@@@ end behavior @@@\n");
};
//----------------------Registradores------------------------------
#define OFFSET 0xB000 //Inicio da memória de dados

//Registradores de endereços
int R[4]; //R0, R1, R2, R3

//Acumuladores
long A01; //32 bits
int A0; //16 bits
int A1; //16 bits
int A2; //4 bits LSB

long B01; //32 bits
int B0; //16 bits
int B1; //16 bits
int B2; //4 bits LSB

//Registrador Y
long Y; //32 bits
int Y0; //16 bits
int Y1; //16 bits

//Registrador de status SR
//Mode Register (MR) - Os 5 bits entre o LF e o I1 são reservados
bool SR_mr_lf = 0;
bool SR_mr_i1 = 0;
bool SR_mr_i0 = 1;
//Condition Code register (CCR)
bool SR_ccr_sz = 0; //Size
bool SR_ccr_l = 0; //Limiting
bool SR_ccr_e = 0; //Extension
bool SR_ccr_u = 0; //Unnormalized
bool SR_ccr_n = 0; //Negative
bool SR_ccr_z = 0; //Zero
bool SR_ccr_v = 0; //Overflow
bool SR_ccr_c = 0; //Carry

//Registrador OMR
bool OMR_nl = 0; //Nested Looping
bool OMR_cc = 1; //Conditional Codes - Testes
bool OMR_sd = 0; //Stop Delay
bool OMR_r = 0;  //Rouding
bool OMR_sa = 0; //Saturation
bool OMR_ex = 0; //External X memory
bool OMR_mb = 1; //Operating modes
bool OMR_ma = 1; //Operating modes
//---------------------------------------------------------------------
//!Generic instruction behavior method.
void ac_behavior( instruction )
```

```
{
    if (ac_cycle == 1)
    {
        pc += ac_instr_size/8; //A memória de programa é acessada em bytes
    }
    if (ac_cycle == get_cycles())
    {
        ac_pc = (int)pc;   //Só incrementa o PC após o consumo de todos os ciclos
    }
};
//! Instruction Format behavior methods.
void ac_behavior( Type_1W ){}
void ac_behavior( Type_2W_Bit2 ){}
void ac_behavior( Type_Jsr ){}
void ac_behavior( Type_Dalu3op2 )
{
    //Desabilita a Saturação
    OMR_sa = 0;
}
//---------------Funções Auxiliares------------------------------------
//Verifica e efetua a saturação dos acumuladores durante as instruções MOVE
//Data Limiter
void checkSatDataLim(unsigned short int acum, unsigned short int index, char mem)
{
    int F2 = 0; //EXT
    int F1 = 0; //F1
    int ext_in_use = 0;
    if (acum == 4) //A
    {
        F2 = A2;
        F1 = A1;
    }
    else if(acum == 5) //B
    {
        F2 = B2;
        F1 = B1;
    }
     //Verifica se é necessária a saturação
    ext_in_use = !( (((F2 & 0x000f) == 0) && ((F1 & 0x8000) == 0)) ||
                  (((F2 & 0x000f) == 0x000f) && ((F1 & 0x8000) == 0x8000)));
    if (ext_in_use)
    {
       if((F2 & 0x0008) == 0)
        {
            F1 = 0x7FFF;
            SR_ccr_l = 1;
        }
        else if ((F2 & 0x0008) == 0x0008)
        {
            F1 = 0x8000;
            SR_ccr_l = 1;
        }
    }
   //Calcula o bit SZ
    SR_ccr_sz |= ((F1 >> 29) ^ (F1 >> 28));
    //Efetua o move de 16 bits
    if (mem == 'P') // Memória de Programa
    {
        DM.write(index,F1); //grava F1
    }
    else if (mem == 'D') // Memória de Dados
    {
        DMwrite(index,F1); //grava F1
    }
}
//Verifica e efetua a saturação dos acumuladores durante as instruções MOVE
//Data Limiter
char checkSatMacLim(int f2, int f1)
{
    int ext3,ext0,msp15;
    ext3 = ((f2 & 0x0008) >> 3);
    ext0 = (f2 & 0x0001);
    msp15 = ((f1 & 0x8000) >> 15);
```

```
    if (((ext3 == 0) && (ext0 == 0) && (msp15 == 1)) ||
            ((ext3 == 0) && (ext0 == 1) && (msp15 == 0)) ||
            ((ext3 == 0) && (ext0 == 1) && (msp15 == 1)))
    {
        SR_ccr_v = 1;
        SR_ccr_l = 1;
        SR_ccr_u = 0;
        return ('P'); //Maximo valor positivo
    }
    else if (((ext3 == 1) && (ext0 == 0) && (msp15 == 1)) ||
            ((ext3 == 1) && (ext0 == 0) && (msp15 == 1)) ||
            ((ext3 == 1) && (ext0 == 1) && (msp15 == 0)))
    {
        SR_ccr_v = 1;
        SR_ccr_l = 1;
        SR_ccr_u = 0;
        return ('N'); //Maximo valor negativo
    }
    else
        return (0);
}
//Atualiza os registrador de Status em funções genéricas com dois operandos de 36 bits.
void updateSR_36_36_gen(int s_ext,int d_ext, long src, long dest, long res1, int res2)
{
    SR_ccr_c = ((s_ext & d_ext & 0x0008) |
                (~res2 & (s_ext | d_ext) & 0x0008) );
    SR_ccr_e = !( (((res2 & 0x000f) == 0) && ((res1 & 0x80000000) == 0)) ||
                 (((res2 & 0x000f) == 0x000f) && ((res1 & 0x80000000) == 0x80000000)));
    if (OMR_cc == 0)
    {
        SR_ccr_v = (( s_ext &  d_ext & ~res2 & 0x0008) |
                (~s_ext & ~d_ext &  res2 & 0x0008) );
        SR_ccr_z = ((res2 == 0) && (res1 == 0));
        SR_ccr_n = (res2 >> 3);
    }
    else if (OMR_cc == 1)
    {
        SR_ccr_v = (( src & dest & ~res1 & 0x80000000) |
                (~src & ~dest &  res1 & 0x80000000) );
        SR_ccr_z = (res1 == 0);
        SR_ccr_n = (res1 >> 31);
    }
    SR_ccr_u = !(((res1 & 0x80000000) >> 31) ^ ((res1 & 0x40000000) >>30));
    SR_ccr_l |= SR_ccr_v;
}
//Escreve na memória adicionando o OFFSET para acessar a área de dados
void DMwrite(int index,int data)
{
     //Compatibiliza o endereço
     index = (index * 2) & 0xffff;
     index += OFFSET;
     index &= 0xffff;
     DM.write(index,data);
}
//Lê a memória adicionando o OFFSET para acessar a área de dados
int DMread(int index)
{
     int data = 0;
     //Compatibiliza o endereço
     index = (index * 2) & 0xffff;
     index += OFFSET;
     index &= 0xffff;
     data = DM.read(index);
     return data;
}
//------------------------------INSTRUÇÕES--------------------------------
//!Instruction add_b_a behavior method.
void ac_behavior( add_b_a )
{
    int carry = 0;
    long res1 = 0;
    int res2  = 0;
    switch (cycle)
```

```
    {
        case 1:
            printf("@@@ ADD B A @@@\n");
            //Soma as porções de 31 bits
            res1 = B01 + A01;
            //Verifica se houve Carry na operação de 32 bits
            carry = ((B01 & A01 & 0x80000000) |
                    (~res1 & (B01 | A01) & 0x80000000) );
            carry = (carry >> 31) & 0x00000001;
            res2 = B2 + A2 + carry;
            updateSR_36_36_gen(B2,A2,B01,A01,res1,res2);
             //Efetua a saturação, se necessário
            if (OMR_sa)
            {
                if (checkSatMacLim(res2,((res1 & 0xffff0000) >> 16)) == 'P')
                {
                    res1 = 0x7fffffff;
                    res2 = 0;
                }
                else if (checkSatMacLim(res2,((res1 & 0xffff0000) >> 16)) == 'N')
                {
                    res1 = 0x80000000;
                    res2 = 0xf;
                }
            }
            A01 = res1;
            A0 = A01 & 0x0000ffff;
            A1 = (A01 & 0xffff0000) >> 16;
            A2 = (res2 & 0x000f);
            break;
        case 2:
            break;
    }
    ac_cycle++;
}
//!Instruction jsr behavior method.
void ac_behavior( jsr )
{
    switch (cycle)
    {
        case 1:
            printf("@@@ JSR imm16 @@@\n");
            SP.write(SP.read() + 1); //(SP)+
      DMwrite(SP.read(),pc);
            SP.write(SP.read() + 1); //(SP)+
            mountSR(); //Concatena o Registrador de Status
            DMwrite(SP.read(),SR.read());
            pc = addr16 * 2;
            pc &= 0xffff;
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        case 6:
            break;
        case 7:
            break;
        case 8:
            break;
    }
    ac_cycle++;
}
//!Instruction move_r2_pp6 behavior method.
void ac_behavior( move_r2_pp6 )
{
    int index = 0;
    switch (cycle)
    {
```

```
        case 1:
            printf("@@@ MOVE imm16,x:(r2+<<pp6) @@@\n");
             //Extende positivamente o pp6
            pp6 = (0x0000 | pp6);
            index = (R[2] + (int)pp6);
            DMwrite(index,imm16);
            break;
        case 2:
            break;
        case 3:
            break;
        case 4:
            break;
        case 5:
            break;
        case 6:
            break;
    }
    ac_cycle++;
}
//!Instruction impy_dalu3op2 behavior method.
void ac_behavior( impy_dalu3op2 )
{
    int fff = 0;

    switch (cycle)
    {
        case 1:
            printf("@@@ IMPY qqq,fff @@@\n");
            fff = (f0 << 2) | f1f2;
           dalu3op2Mac(fff,qqq,0x2); //Efetua a operação de multiplicação
            break;
        case 2:
            break;
    }
    ac_cycle++;
}
//!Instruction rts behavior method.
void ac_behavior( rts )
{
    int SR;
    switch (cycle)
    {
        case 1:
            printf("@@@ RTS @@@\n");
            SR = DMread(SP.read()); //Lê o SR e depois descarta
            break;
        case 2:
            break;
        case 3:
            SP.write(SP.read() - 1); //Subtrai o SP
            break;
        case 4:
            break;
        case 5:
            pc = DMread(SP.read());
            pc &= 0xffff;
            break;
        case 6:
            break;
        case 7:
            SP.write(SP.read() - 1);//Subtrai o SP
            break;
        case 8:
            break;
        case 9:
            break;
        case 10:
            break;
    }
    ac_cycle++;
}
```

APÊNDICE B

CÓDIGO FONTE DO SIMULADOR ATMEGA8515

Arquivo de descrição da arquitetura, *atmega8515.ac*:

```
AC_ARCH(atmega8515){
   /* Descrição de memória */
   ac_mem DM:64k; //Considerando a memória externa
   /* Numero de bits que o atmega trabalha */
   ac_wordsize 16;
   /* Descricao dos registradores */
   ac_format STAT_F = "%I:1 %T:1 %H:1 %S:1 %V:1 %N:1 %Z:1 %C:1";

   ac_regbank RB:32;  //Banco de registradores de uso geral
   ac_reg <STAT_F>SREG;   //Registrador de Status
   ac_reg              SP;     //Apontador de Pilha

   ARCH_CTOR(atmega8515){
      ac_isa("atmega8515_isa.ac");
      set_endian("big");
   };
};
```

Trecho do arquivo de descrição da sintaxe das instruções, atmega8515_isa.ac (limitado a quatro instruções):

```
AC_ISA(atmega8515){

   ac_format Type_F1B = "%op8:8 %Rd4:4 %Rr4:4";
   ac_format Type_F4 = "%op:16";
   ac_format Type_F11 = "%op_e1:2 %q1:1 %op_e2:1 %q2:2 %op_e3:1 %Rd:5 %op_e4:1 %q3:3";
   ac_format Type_F12A = "%op16:16 %rk:16";

   ac_instr<Type_F1B>movw;
   ac_instr<Type_F4>ret;
   ac_instr<Type_F11>lddY;
   ac_instr<Type_F12A>call;

ISA_CTOR(atmega8515){

      movw.set_asm("movw %Rd,%Rr");
      movw.set_decoder(op8=0x01);   //Quando não tem set_cycles, o número de ciclos é 1

      call.set_asm("call");
      call.set_decoder(op16=0x940e);
      call.set_cycles(4);

      lddY.set_asm("ldd %Rd,Y+%q");
      lddY.set_decoder(op_e1=0x2, op_e2=0, op_e3=0, op_e4=1);
      lddY.set_cycles(2);

      ret.set_asm("ret");
      ret.set_decoder(op=0x9508);
      ret.set_cycles(4);
   };
};
```

Trecho do arquivo de descrição da semântica das instruções, atmega8515-isa.cpp (limitado a quatro instruções):

```
#include  "atmega8515-isa.H"
#include  "ac_isa_init.cpp"

//-----Variáveis Globais------
```

```
unsigned short int SKIP = 0;
unsigned short int NEXT = 0;
int pc = 0;

//!Behavior executed before simulation begins.
void ac_behavior( begin )
{
      printf("@@@ begin behavior @@@\n");
      SREG.I = 0;      SREG.T = 0;
      SREG.H = 0;      SREG.S = 0;
      SREG.V = 0;      SREG.N = 0;
      SREG.Z = 0;      SREG.C = 0;
      SP = 0;
};

//!Behavior executed after simulation ends.
void ac_behavior( end )
{
      printf("@@@ end behavior @@@\n");
};

//!Generic instruction behavior method.
void ac_behavior( instruction )
{
      if( ac_cycle == 1 )
      {
          pc += ac_instr_size/8; //Busca a próxima instrução
      }
      if( ac_cycle == get_cycles() )
      {
          ac_pc = (int)pc; //Prepara para a execução da próxima instrução
          NEXT = 0;
      }
};

//! Instruction Format behavior methods.
void ac_behavior( Type_F1B ){}
void ac_behavior( Type_F4 ){}
void ac_behavior( Type_F11 ){}
void ac_behavior( Type_F12A ){}

//----------------------FUNÇÕES AUXILIARES-------------------------------------
#define OFFSET 0xD000 //Inicio da memória de dados
//Lê um byte da memória adicionando o OFFSET para acessar a area de dados
int DMread_byte(int index)
{
    int data = 0;
    index += OFFSET;
    index &= 0xffff;
    data = DM.read_byte(index);
    return data;
}
//Escreve um byte na memória adicionando o OFFSET para acessar a area de dados
void DMwrite_byte(int index,int data)
{
   //Compatibiliza o endereço
   index += OFFSET;
   index &= 0xffff;
   DM.write_byte(index,data);
}
//------------------------INSTRUÇÕES-----------------------------------------
//!Instruction movw behavior method.
void ac_behavior( movw )
{
      char AUX1, AUX2; //para evitar colizao
      unsigned short int Rd, Rr;

      Rd = Rd4*2;
      Rr = Rr4*2;
      printf("@@@@@@@@%s\n",get_name());
      AUX1 = RB.read(Rr);
      AUX2 = RB.read(Rr + 1);
      RB.write(Rd,AUX1);
```

```
      RB.write(Rd + 1,AUX2);
      //Passa para a próxima instrução
      NEXT = 1;
}
//!Instruction lddY behavior method.
void ac_behavior( lddY )
{
      int q = 0;
      int R = 0;
      int Y = (((RB.read(29) & 0xff) << 8)|(RB.read(28) & 0xff)) & 0xffff;

      //Obtém o valor do deslocamento
      q = (q1 << 5)|(q2 << 3)|q3;
      switch (cycle)
      {
           case 1:
                printf("@@@@@@@@%s\n", get_name());
                R = Y + q;
                RB.write(Rd, DMread_byte(R) & 0xff);
                break;
           case 2:
                break;
      }
      //Passa para a próxima instrução
      NEXT = 1;
      ac_cycle++;
}
//!Instruction call behavior method.
void ac_behavior( call )
{
      switch (cycle)
      {
           case 1:
               printf("@@@@@@@@%s\n",get_name());
               break;
           case 2:
               DMwrite(SP.read(), pc);
               break;
           case 3:
               pc = rk * 2;
               pc &= 0xffff;
               ac_pc = pc;
               break;
           case 4:
               SP -= 2;
               break;
      }
      //Passa para a próxima instrução
      NEXT = 1;
      ac_cycle++;
}
//!Instruction ret behavior method.
void ac_behavior( ret )
{
      switch (cycle)
      {
           case 1:
               printf("@@@@@@@@%s\n",get_name());
               SP += 2;
               break;
           case 2:
               break;
           case 3:
               pc = DMread(SP.read());
               pc &= 0xffff;
               break;
           case 4:
               break;
      }
      //Passa para a próxima instrução
      NEXT = 1;
      ac_cycle++;
}
```

APÊNDICE C

CÓDIGO FONTE DO SIMULADOR R2000

Arquivo de descrição da arquitetura, R2000.ac:

```
AC_ARCH(r2000){

  /* Descrição da memória */
  ac_mem  DM:64k;
  /* Numero de bits que o rabbit trabalha */
  ac_wordsize 8;
  /* Descrição dos formatos */
  ac_format Fmt_AF = "%A:8 %s:1 %z:1 %dc3:3 %lv:1 %dc1:1 %c:1";
  ac_format Fmt_2B = "%reg:16";

  /* Descrição dos registradores */
  //Acumuladores de 16 bits
  ac_reg <Fmt_AF> AF;
  ac_reg <Fmt_2B> HL;
  ac_reg <Fmt_2B> DE;
  ac_reg <Fmt_2B> BC;
  //Acumuladores Alternativos
  ac_reg <Fmt_AF> AF_a;
  ac_reg <Fmt_2B> HL_a;
  ac_reg <Fmt_2B> DE_a;
  ac_reg <Fmt_2B> BC_a;
  //Contador de Programa
  ac_reg <Fmt_2B> PC;
  //Apontador de Pilha
  ac_reg <Fmt_2B> SP;
  //Registradores de indice
  ac_reg <Fmt_2B> IX;
  ac_reg <Fmt_2B> IY;
  //Registradores que controlam as interrupções
  ac_reg IP;
  ac_reg IIR;
  ac_reg EIR;
  //Extensão do PC
  ac_reg XPC;

  ARCH_CTOR(r2000){
    ac_isa("r2000_isa.ac");
    set_endian("big");
  };
};
```

Trecho do arquivo de descrição da sintaxe, R2000_isa.ac (limitado a quatro instruções):

```
AC_ISA(r2000){

  ac_format Type_1B = "%op8:8";
  ac_format Type_2B = "%op16:16";
  ac_format Type_3B_n_m  = "%op8:8 %n:8 %m:8";
  ac_format Type_1B_r_f_v_rg = "%op2:2 %r_f_v:3 %g:3";

  ac_instr<Type_1B> adc_a_hl;
  ac_instr<Type_2B> and_ix_de;
  ac_instr<Type_3B_n_m> call_mn;
  ac_instr<Type_1B_r_f_v_rg> ret_f;

  ISA_CTOR(r2000){

    adc_a_hl.set_asm("adc A,(HL)");
    adc_a_hl.set_decoder(op8=0x8e);
    adc_a_hl.set_cycles(5);
```

```
    and_ix_de.set_asm("and IX,DE");
    and_ix_de.set_decoder(op16=0xdddc);
    and_ix_de.set_cycles(4);

    call_mn.set_asm("call %mn");
    call_mn.set_decoder(op8=0xcd);
    call_mn.set_cycles(12);

    ret_f.set_asm("ret %f");
    ret_f.set_decoder(op2=0x3,g=0x0);
    ret_f.set_cycles(8);
  };
};
```

Trecho do arquivo de descrição da semântica, R2000-isa.cpp (limitado a quatro instruções):

```
#include  "r2000-isa.H"
#include  "ac_isa_init.cpp"

//!Behavior executed before simulation begins.
void ac_behavior( begin )
{
      printf("@@@ begin behavior @@@\n");
      //@@@@@@--------INICIALIZACAO DOS REGISTRADORES--------------------
      AF.A.write(0);
      HL.reg.write(0);
      DE.reg.write(0);
      BC.reg.write(0);
      AF_a.A.write(0);
      HL_a.reg.write(0);
      DE_a.reg.write(0);
      BC_a.reg.write(0);
      PC.reg.write(0);
      SP.reg.write(0xf000);
      IX.reg.write(0);
      IY.reg.write(0);
};
//!Behavior executed after simulation ends.
void ac_behavior( end )
{
      printf("@@@ end behavior @@@\n");
};
//@@@@@@--------VARIÁVEIS GLOBAIS------------------------------------
int pc = 0;
bool altd = 0;
int F = 0;
int F_a = 0;
int res = 0;
int src = 0;
int dest = 0;
int aux = 0;
int mindex = 0;
bool carry = 0;

//!Generic instruction behavior method.
void ac_behavior( instruction )
{
      if( ac_cycle == 1 )
           pc += ac_instr_size/8;
      if( ac_cycle == get_cycles())
      {
           ac_pc = (unsigned int)pc;
      }
};

//! Instruction Format behavior methods.
void ac_behavior( Type_1B ){}
void ac_behavior( Type_2B ){}
void ac_behavior( Type_3B_n_m ){}
void ac_behavior( Type_1B_r_f_v_rg ){}
```

```
//@@@@@@--------FUNÇÕES AUXILIARES-----------------------------------
#define OFFSET 0x0000
//Lê um byte da memria adicionando o OFFSET para acessar a area de dados
int DMread_byte(int index)
{
    int data = 0;
    index += OFFSET;
    index &= 0xffff;
    data = DM.read_byte(index);
    return data;
}
//Escreve um byte na memria adicionando o OFFSET para acessar a area de dados
void DMwrite_byte(int index,int data)
{
   //Compatibiliza o endere¿
   index += OFFSET;
   index &= 0xffff;
   DM.write_byte(index,data);
}
//@@@Atualiza o Registrador de Status para as operações de 8 bits-----
void update_F_8(char source, char dst, char result, char l_v, int sub)
{
      bool s,lv,z,c;
      s=lv=z=c=0;

      // Em subtrações o cálculo é diferente
      if (sub)
      {
          c = ((~dst & source & 0x80) |
               (result & (~dst | source) & 0x80) );
      }
      else
      {
          c = ((source & dst & 0x80) |
               (~result & (source | dst) & 0x80) );
      }
      // Verifica se é uma operação lógica ou aritmética
      if (l_v == 'l')
      {
          lv = (result & 0xf0) != 0;
      }
      else if (l_v == 'v')
      {
           if (sub)
           {
               lv = (( dst & ~source & ~result & 0x80) |
                     (~dst & source &  result & 0x80) );
           }
           else
           {
               lv = (( source &  dst & ~result & 0x80) |
                     (~source & ~dst & result & 0x80) );
           }
      }
      z = result == 0;
      s = (result & 0x80) >> 7;
      if (altd)
      {
          AF_a.c.write(c);
          AF_a.lv.write(lv);
          AF_a.z.write(z);
          AF_a.s.write(s);
      }
      else
      {
          AF.c.write(c);
          AF.lv.write(lv);
          AF.z.write(z);
          AF.s.write(s);
      }
}
//@@@@@@--------IMPLEMENTAÇÃO DAS INTRUÇÕES---------------------------
```

```
//!Instruction adc_a_hl behavior method.
void ac_behavior( adc_a_hl )
{
     switch (cycle)
     {
          case 1:
               printf("@@@ ADC A,(HL) @@@\n");
               src = DMread_byte(HL.reg.read());
               break;
          case 2:
               dest = AF.A.read();
               break;
          case 3:
               res = dest + src + AF.c.read();
               res = (res & 0xff);
               break;
          case 4:
               update_F_8(src,dest,res,'v',0); //Atualiza o Status
               if (altd) //Registradores alternativos
                   AF_a.A.write(res);
               else
                   AF.A.write(res);
               mount_F();
               break;
          case 5:
               break;
      }
      ac_cycle++;
}
//!Instruction and_ix_de behavior method.
void ac_behavior( and_ix_de )
{
      switch (cycle)
      {
           case 1:
                printf("@@@ AND IX,DE @@@\n");
                src = DE.reg.read();
                break;
           case 2:
                dest = IX.reg.read();
                break;
           case 3:
                res = dest & src ;
                break;
           case 4:
                update_F_16(src,dest,res,'l',0); //Atualiza o Status
                if (altd)
                    AF_a.c.write(0);
                else
                    AF.c.write(0);
                IX.reg.write(res);
                mount_F();
                break;
      }
      ac_cycle++;
}
//!Instruction call_mn behavior method.
void ac_behavior( call_mn )
{
      switch (cycle)
      {
           case 1:
                printf("@@@ CALL mn @@@\n");
                break;
           case 2:
                DMwrite_byte((SP.reg.read()-1),(ac_pc.read() >> 8));
                break;
           case 3:
                DMwrite_byte((SP.reg.read()-2),(ac_pc.read() & 0x00ff));
                break;
           case 4:
                break;
           case 5:
```

```
                pc = (m<<8)|n;
                SP.reg.write(SP.reg.read() - 2);
                break;
           case 6:
                break;
           case 7:
                break;
           case 8:
                break;
           case 9:
                break;
           case 10:
                break;
           case 11:
                break;
           case 12:
                break;
      }
      ac_cycle++;
}
 //!Instruction ret_f behavior method.
void ac_behavior( ret_f )
{
     int ok = 0;
     switch(cycle)
     {
          case 1:
               printf("@@@ %s ----- \n", get_name() );
               switch(r_f_v)
               {
                    case 0x0:
                         if (AF.z.read() == 0)
                         ok = 1;
                         break;
                    case 0x1:
                         if (AF.z.read() != 0)
                            ok = 1;
                         break;
                    case 0x2:
                         if (AF.c.read() == 0)
                             ok = 1;
                         break;
                    case 0x3:
                         if (AF.c.read() != 0)
                            ok = 1;
                         break;
                    case 0x4:
                         if (AF.lv.read() == 0)
                            ok = 1;
                         break;
                    case 0x5:
                         if (AF.lv.read() != 0)
                            ok = 1;
                         break;
                    case 0x6:
                         if (AF.s.read() == 0)
                            ok = 1;
                         break;
                    case 0x7:
                         if (AF.s.read() != 0)
                            ok = 1;
                         break;
               }
               if (ok)
               {
                    pc = (pc & 0xff00)|DMread_byte(SP.reg.read());
                    pc = (DMread_byte(SP.reg.read()+1)<<8)|(pc & 0x00ff);
                    pc += 3; //Executará a instrução após o CALL
                    SP.reg.write(SP.reg.read()+2);
               }
               break;
          case 2:
               break;
```

```
            case 3:
                break;
            case 4:
                break;
            case 5:
                break;
            case 6:
                break;
            case 7:
                break;
            case 8:
                break;
        }
        ac_cycle++;
}
```

BIBLIOGRAFIA

[1] FPGA Research at the University of Toronto, mar 2005. `http://www.eecg.toronto.edu/EECG/RESEARCH/FPGA.html`.

[2] Retargeting GCC to ArchC Models, fev 2005. `http://archc.sourceforge.net/files/compilers/retargeting-gcc-for-archc-models.pdf`.

[3] A. A. Barbiero. Ambiente de Desenvolvimento para Microcontrolador AVR8515. Trabalho de Graduação, UFPR, 2003.

[4] A. M. Amory. Definição e Desenvolvimento de Metodologia de Codesign. Trabalho Individual I, PUC-RS, 2001.

[5] R. Amicel and F. Bodin. Mastering Startup Costs in Assembler-Based Compiled Instruction-Set Simulation. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT.02)*, 2002.

[6] The ArchC Arquitecture Description Language, fev 2005. `http://www.archc.org`.

[7] ATMEL AVR Microcontrollers, mar 2005. `http://www.atmel.com`.

[8] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araújo, C. Araújo, and E. Barros. The ArchC Architecture Description Language and Tools. *International Journal of Parallel Programming*, 33(5), 2005.

[9] L. Baker. *VHDL Programming with Advanced Topics*, volume 1. Wiley, 1993.

[10] A. Baldassin, P. Centoducatte, and S. Rigo. Extending the ArchC Language for Automatic Generation of Assemblers. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC05)*, 2005.

[11] A. A Barbiero and R. A. Hexsel. Simulação de Sistemas Embarcados utilizando ArchC. In *VII Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2006)*, 2006.

[12] M. Barr. *Programming Embedded Systems in C and C++*, volume 1. O'Reilly, 1999.

[13] A. C. Beck, J. C. B. Mattos, F. R. Wagner, and L. Carro. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In *16th Symposium on Integrated Circuits and Systems Design*, pages 349–354, São Paulo, Brazil, 2003. IEEE CompSoc Press.

[14] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1625–1640, 2004.

[15] BRAZIL IP Network, mar 2006. `http://www.brazilip.org.br/`.

[16] C. Böke. Combining Two Customization Approaches: Extending the Customization Tool TEReCS for Software Synthesis of Real-Time Execution Platforms. In *AES 2000 - Workshop on Architectures of Embedded Systems, Karlsruhe, Alemanha*, 2000.

[17] L. Carro and F. R. Wagner. Sistemas Computacionais Embarcados. In *JAI'03 - XXII Jornadas de Atualização em Informática*, Campinas, Brasil, August 2003.

[18] Ceitec. CEITEC - Centro de Excelência em Tecnologia Eletrônica Avançada, mai 2006. `http://www.ceitec.org.br`.

[19] Metrowerks Code Warrior IDE, mar 2005. `http://www.metrowerks.com/MW/Products/CodeWarrior+Technology.htm`.

[20] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, volume 1. Prentice Hall, 4th edition, 2000.

[21] J. Connel and B. Johnson. Early Hardware/Software Integration Using SystemC 2.0. Technical report, ARM and Synopsys, 2002.

[22] J. B. Connell. Early Hardware/Software Integration Using SystemC 2.0. In *Embedded Systems Conference*, San Francisco, USA, 2002.

[23] L. A. Cortés, P. Eles, and Z. Peng. Modeling and Formal Verification of Embedded Systems based on a Petri Net Representation. *Journal of Systems Architecture*, 49(12-15):571–598, 2003.

[24] CS.UCR.edu. Dalton Project, fev 2006. `http://www.cs.ucr.edu/~dalton/i8051/i8051syn`.

[25] Design and Reuse. Design and Reuse, mai 2006. `http://www.us.design-reuse.com`.

[26] Distributed Network Protocol, fev 2006. `http://www.dnp.org`.

[27] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[28] J. Ellsberger; et al. *SDL: Formal Object-Oriented Language for Communicating Systems*, volume 1. Prentice Hall, 1a edition, 1997.

[29] D. D. Gajski. SpecC: Specification Language and Methodology. In *Kluwer Academic. Boston. USA*, 2000.

[30] GCC - The Gnu Compiler Collection, mar 2005. `http://gcc.gnu.org`.

[31] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*, volume 1. Webman, 2002.

[32] A. Halambi, P. Grun, V. Ganesh, A. Khare, N.Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. In *European Conference on Design, Automation and Test*, 1999.

[33] J. L. Hennessy and D. A. Patterson. *Arquitetura de Computadores: Uma abordagem Quantitativa*, volume 1. Ed Campus, 3a edition, 2003.

[34] S. A. Ito, L. Carro, and R. P. Jacobi. Designing a Java Microcontroller to Specific Application. In *XII Brazilian Symp Integrated Circuit Design (SBCCI 99)*, pages 12–15. IEEE CS Press, 1999.

[35] S. A. Ito, L. Carro, and R. P. Jacobi. Making Java Work for Microcontroller Applications. *Computer Magazine*, 18(5):100–110, 2001.

[36] T. E. Jeremiassen. Sleipnir - An Instruction-Level Simulator Generator. In *International Conference on Computer Design*, 2000.

[37] A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*, volume 1. Elsevier, 1a edition, 2005.

[38] L. H. Watter. L@V@ - Um Simulador para o Microcontrolador AVR8515. Trabalho de Graduação, UFPR, 2002.

[39] LACTEC Instituto de Tecnologia para o Desenvolvimento, mar 2005. `http://www.lactec.org.br`.

[40] M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals*, volume 1. Prentice Hall, 2nd edition, 2000.

[41] G. Martin. SystemC and the Future of Design Languages: Opportunities for Users and Research. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI 03)*, 2003.

[42] Mathworks. Mathworks, mai 2006. `http://www.mathworks.com`.

[43] J. C. B. Mattos, A. C. Beck, L. Carro, and F. R. Wagner. Design Space Exploration with Automatic Selection of SW and HW for Embedded Applications. In *SAMOS IV - 4th International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2004.

[44] J. C. B. Mattos, L. Brisolara, R. Hentschke, L. Carro, and F. R. Wagner. Design Space Exploration with Automatic Generation of Ip-Based Embedded Software. In *DIPES'2004 - IFIP Working Conference on Distributed and Parallel Embedded Systems, Toulouse, France*, August 2004.

[45] Mentor. Seamless CVE, mai 2006. `http://www.mentor.com/seamless`.

[46] F. G. Moraes, N. L. Calazans, C. A. M. Marcon, and A. V. Mello. Um Ambiente de Compilação e Simulação para Processadores Embarcados Parametrizáveis. In *VII IBERCHIP WorkShop*, Montevidéo, Uruguai, March 2001.

[47] Motorola DSP Family, mar 2005. `http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012795`.

[48] RABBIT Semiconductor, mar 2005. `http://www.rabbitsemiconductor.com/products/dc/index.shtml`.

[49] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *DAC 2003, Anaheim, Califomia, USA.*, 2003.

[50] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-Based Architecture Description Language. In *16th Symposium on Computer Architecture and High Performance Computing*, Foz do Iguacu, Brazil, Outubro 2004.

[51] T. Rissa, A. Donlin, and W. Luk. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In *Design, Automation and Test in Europe Conference and Exhibition (DATE 05)*, 2005.

[52] Rabbit WinIDE, mar 2006. `http://www.softools.com/`.

[53] Synopsys. CoCentric System Studio, mai 2006. `http://www.synopsys.com/products/cocentric_studio/cocentric_studio.html`.

[54] SystemC.org. SystemC Community, mar 2005. `http://www.systemc.org`.

[55] D. Tennenhouse. Proactive computing. *Communications ACM*, 43(5):43–50, 2000.

[56] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture Description Languages for System–On–Chip Design. In *APCHDL, Fukuoka, Japan*, 1999.

[57] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araújo. Modeling and Simulating Memory Hierarchies in a Platform-based Design Methodology. *Design, Automation and Test in Europe (DATE 04)*, February 2004.

[58] W. Wolf. Applications and Architectures. *Computer*, 37(11):114–116, 2004.

[59] W. H. Wolf. Hardware-Software Co-Design of Embedded Systems. *IEEE*, 82(7):967–989, 1994.

[60] V. Zambenedetti, R. P. Siqueira, F. R. Coutinho, A. A. Barbiero, J. Pereira, and R. A. Hexsel. Uso de Comunicação Celular Digital utilizando a Tecnologia 2.5G para Sistemas de Automação de Energia Elétrica. In *VI SIMPASE, São Paulo, Brasil*, Julho 2005.

[61] V. Zivojnovic, S. Pees, and H. Meyr. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. In *IEEE Workshop on VLSI Signal Processing, San Francisco*, 1996.

ANDRÉIA APARECIDA BARBIERO

AMBIENTE DE SUPORTE AO PROJETO DE SISTEMAS EMBARCADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2006