UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Modelagem do processador DSP TMS320C4x para uma plataforma de SoCs

Vinícius dos Santos Livramento

Trabalho de conclusão de curso apresentada como parte dos requisito para obtenção do grau Bacharel em Ciências da Computação

Florianópolis – SC

2008/1

Vinícius dos Santos Livramento

Modelagem do processador DSP TMS320C4x para uma plataforma de SoCs

Trabalho de conclusão de curso apresentada como parte dos requisito para obtenção do grau Bacharel em Ciências da Computação

Orientador: Prof. Dr. Luiz Cláudio Villar dos Santos

Banca Examinadora

_____

Prof. Dr. Olinto José Varela Furtado

_____

Prof. Dr. José Luís Almada Güntzel

Agradecimentos

Primeiramente gostaria de agradecer a Deus pelas condições favoráveis à realização deste trabalho. Agradeço também aos meus pais, Mário e Cristina, e à minha irmã Natália por todo amor e carinho por eles dado.

Ao professor Luiz Cláudio pela sua orientação e aos membros da banca pelas sugestões.

A todos os colegas de laboratório, em especial a Alexandre Mendonça, Paulo Kuss, Sandro Carvalho e Emílio Wuergs que de alguma forma contribuiram no trabalho.

Ao colega Bruno Albertini pelo suporte técnico dado ao trabalho.

Aos meus amigos Fábio dos Santos e José Raupp pela ajuda e incentivo.

Resumo

O grande avanço das tecnologias CMOS nanométricas deu origem a sistemas computacionais inteiros em um único chip, conhecido como System On Chip (SoC). As diversas dificuldas enfrentadas pelos projetistas no projeto de SoCs, tornaram indispensáveis o uso de ferramentas que auxiliem no processos de desenvolvimento e teste destes sistemas.

O projeto em nível de sistema eletrônico(ESL) surge como uma alternativa ao projeto em nível RT, já que trabalha em um nível maior de abstração, aumentando assim a compreensão do sistema como um todo. O projeto em nível de sistema permite a um projetista uma maior exploração do espaço de projeto para CPUs alternativas, assim como antecipar o desenvolvimento do software dependente de hardware, para isto a disponibilidade de modelos de CPU são mandatórios.

Esta monografia apresenta a modelagem funcional (atemporal) do processador
DSP TMS 320C4x em uma linguagem de descrição de arquitetura (ADL), para ser utilizado como alternativa de CPU na plataforma ARP (ArchC Reference Platform).

Abstract

The great advancement of nano CMOS technology has led to entire computer systems on a single chip, known as System on Chip (SoC). The various dificuldas faced by designers in the design of SoCs, become necessary the use of tools that help in the process of development and testing of these systems.

The electronic system-level (ESL) design is emerging as an alternative to register transfer level, already working on a higher level of abstraction, thereby increasing the understanding of the system as a whole. The system level design allows a designer greater design space exploration for alternatives CPUs, and anticipate the development of hardware dependent software, for this, the availability of CPU models are required.

This paper presents the functional modeling (timeless) processor DSP TMS 320C4x in an architecture description language (ADL), to be used as an alternative CPU in ARP platform (ArchC Reference Platform).

Sumário

Lista de Tabelas

Lista de Figuras

1 Introdução

1.1 Contextualização

O interesse pela área de sistemas embarcados vem crescendo de uma forma nunca imaginada. Isso se fez possível devido ao expressivo avanço das tecnologias CMOS nanométricas, possibilitando assim sua grande popularização em diversos dispositivos como cameras digitais, PDAs, celulares, dentre outros. Essa crescente demanda por aplicações embarcadas deu origem a sistemas computacionais inteiros em um único chip, conhecidos como *Systems-on-Chip* (SoCs) (GHENASSIA, 2005, p. 2). Em razão das inúmeras dificuldades apresentadas nas diversas etapas de desenvolvimento desses sistemas, fez-se necessária a adoção de várias ferramentas para auxílio nesses projetos.

Inicialmente, linguagens de descrição de hardware (VHDL, Verilog) facilitaram o trabalho dos projetistas para o projeto em nível RT. Atualmente, alternativas estão sendo apresentadas para que se possa contornar as limitações de se trabalhar apenas em nível RT, já que a complexidade destes sistemas aumenta, e, em contrapartida, o *time-to-market* [1] torna-se cada vez menor.

Fatores cruciais na concepção de um sistema levaram os projetistas a adotarem diversas técnicas para aumentar a reusabilidade dos diversos componentes presentes em um SoC. Blocos reusáveis de hardware, chamados de IPs (*Intellectual Property*) se tornaram cada vez mais comuns na indústria de sistemas embarcados. Essa prática fez surgir o projeto orientado a plataforma (SANGIOVANNI-VINCENTELLI et al., 2004), onde vários IPs seriam agrupados para formar uma plataforma de referência, composta por processador, memória, barramento, etc. Novos projetos poderiam fazer o reuso desses componentes, trazendo benefícios como a redução do *time-to-market*, do esforço dos projetistas e conseqüentemente o preço do produto em desenvolvimento.

Com o advento da plataforma, surgiram as linguagens que trabalham em nível de sistema

[1] *time-to-market* corresponde ao tempo entre a concepção e a comercialização do produto.

(ESL), as linguagens de descrição de sistema, como SystemC (SYSTEMC, 2008). SystemC permite tanto modelagem em nível de sistema, quanto em nível de registradores (RTL). Este novo estilo de modelagem facilita a exploração do espaço de projeto (*design-space exploration*), e traz o ciclo de desenvolvimento de software embarcado para as etapas iniciais do projeto.

Embora o suporte oferecido por SystemC seja bastante grande, não é possível extrair uma descrição genérica de um processador, assim como toda a informação necessária para gerar automaticamente o conjunto de ferramentas de software, que permita ao projetista avaliar um novo conjunto de instruções. Para este fim, utiliza-se uma outra categoria de linguagens, as linguagens de descrição de arquitetura (ADL). Com uma ADL é possível se avaliar um novo conjunto de instruções de uma determinada CPU, e gerar automaticamente todo o conjunto de ferramentas necessárias para avaliação da mesma. O presente trabalho apresenta a modelagem do processador C4x implementado na ADL ArchC (ARCHC, 2008), que foi criada com o intuito de superar as restrições impostas por SystemC. ArchC permite gerar automaticamente um simulador do conjunto de instruções (ISS) de um determinado processador, assim como a geração de um kit de utilitários binários necessários para verificação e exploração de uma nova arquitetura (RIGO, 2004).

1.2 Projeto em nível de sistema eletrônico (ESL)

O grande número de funcionalidades embutidas em um único produto, como por exemplo, os aparelhos celulares, está levando a indústria de sistemas embarcados a adotar novas metodologias de desenvolvimento para substituir as tradicionais, que apontam um grande índice de falhas para atingir os requisitos de QoS[2] e ao mesmo tempo lidar com fatores determinantes como consumo de energia, *time-to-market* e custo de produção. Ao analizar a pesquisa realizada pela companhia *Embedded Market Forecasters* (KRASNER, 2003), que entrevistou diversos projetistas de sistemas embarcados, é possível verificar os diversos problemas dessas metodologias.

- Ao menos 30% do desempenho previsto para o projeto é perdido em 70% dos casos no produto final.
- Ao menos 50% das funcionalidades previstas para o projeto são perdidas em 30% dos casos no produto final.

[2]Do inglês, Quality of Service, pode ser definido como o conjunto de características necessárias para que um sistema possa atingir uma determinada funcionalidade.

- Por volta de 54% dos projetos ultrapassam seus prazos de conclusão em pelo menos 4 meses.
- Dos 40.000 projetos iniciados, aproximadamente 6.000, ou seja, 13% foram cancelados.

Os diversos problemas citados anteriormente são consequência de diversos fatores:

- Falta de interação entre os desenvolvedores de hardware e desenvolvedores de software.
- Normalmente trabalham em nível RT.
- Visibilidade limitada do sistema por completo.
- Praticamente não há exploração do espaço de projeto, restringindo assim a exploração de soluções alternativas de harware e software, como por exemplo, a troca de funcionalidades entre as partes.

Este grande índice de falhas das metodologias tradicionais de desenvolvimento de sistemas embarcados foi o principal motivo para a adoção de uma nova abordagem, conhecida como *Electronic System Level* (ESL). Segundo Bailey, Martin e Piziali (2007), a modelagem em nível de sistema eletrônico tem como principal característica o projeto em um nível maior de abstração, cujo objetivo é aumentar a compreensão do sistema por completo, resultando em uma implementação mais eficiente do sistema.

Um fluxo de projeto ESL de acordo com Bailey, Martin e Piziali (2007) pode ser descrito na forma *top-down*: que inicia com a especificação do produto e termina com sua implementação em software e hardware. É importante observar que o fluxo *top-down* é apenas um conceito, sendo raramente seguido de forma estrita no mundo real. Assim, conceitos como *middle-out*, na qual o projeto faz o reuso de componentes pré-existentes, *bottom-up*, para o caso de um novo componente ser diretamente implementado em nível RT sem a utilização de uma abordagem em nível de sistema, podem ser combinados e utilizados.

O fluxo de projeto ESL proposto por Bailey, Martin e Piziali (2007), é dividido em seis etapas:

1. A etapa de *especificação e modelagem* tem por objetivo a especificação, normalmente escrita em linguagem natural, das funcionalidades e restrições do sistema.
2. Para a etapa de *análise pré-particionamento*, são definidos quais algoritmos serão utilizados para implementação do sistema em face de aspectos como tempo, espaço, potência, complexidade e *time-to-market*.

3. A fase de *particionamento* tem como principal objetivo a divisão em software e hardware dos algoritmos definidos na etapa anterior.

4. Para a etapa de *análise pós-particionamento*, são analisados os modelos definidos na fase de análise pré-particionamento e, em seguida, são refinados para refletir o particionamento definido. Esta fase do projeto permite ao projetista avaliar alguns fatores como: desempenho, consumo de potência, etc. Possibilitando assim, uma exploração do espaço de projeto (*design space exploration*), cujo objetivo é verificar a melhor configuração do hardware/software do sistema. Aqui podemos observar a utilidade de modelos de CPUs alternativas disponíveis ao projetista, assim é possível a analise das diversas CPUs sem a necessidade de adquirí-las.

5. Durante a *verificação pós-particionamento*, avalia-se o comportamento dos componentes de hardware e software, para então verificar se o comportamento especificado nas fases iniciais do projeto foi preservado. Essa avaliação é feita comparando-se o modelo de referência do componente especificado com o modelo resultante (já em um nível menor de abstração). Durante esta fase, um plano de verificação é elaborado para então ser utilizado na implementação de um ambiente de verificação.

6. Na *implementação do hardware* são criados os modelos de hardware, normalmente em nível RT, para futuramente serem sintetizados. Esta fase do projeto permite a tomada de decisões que afetam diretamente o desempenho do sistema, como por exemplo, o compartilhamento de recursos.

7. A *implementação do software* em um fluxo ESL, é feita de forma paralela ao desenvolvimento do hardware *hardware-software codesign*, ao contrário do estilo tradicional, onde o desenvolvimento do software era iniciado após a implementação hardware.

   A abordagem ESL permite aos desenvolvedores de software antecipar o desenvolvimento do software dependente de hardware (*hardware-dependent software*), como *drivers* e *firmwares*, a partir de um modelo funcional do hardware.

   Por fim, as partes de hardware e software são submetidas ao ambiente de verificação definido na etapa de *verificação pós-particionamento*.

1.3 SystemC e modelagem em nível de transação (TLM)

Com o objetivo de contornar a grande complexidade e pressão do *time-to-market* para o projeto de SoCs, elevou-se a abstração para o nível de sistema. SystemC (GROTKER et al.,

2002) (SYSTEMC, 2008) surge nesse contexto com o objetivo de elevar o nível de abstração para o projeto e verificação de hardware. SystemC é uma extensão das linguagens C/C++ e inclui novas funcionalidades que suprem a carência das linguagens tradicionais de programação, inadequadas para a modelagem de componentes de hardware. São elas:

- Introdução da noção de tempo;
- Suporte à concorrência;
- Inclusão de alguns tipos de dados necessários para a descrição de componentes de hardware;

SystemC, que está diponível em domínio público (SYSTEMC, 2008), permite tanto modelagem em nível funcional quanto em nível RT. O estilo TLM foi recentemente padronizado pela Open SystemC Iniciative (SYSTEMC, 2008) e, tem se mostrado bastante promissor para o co-projeto de hardware e software.

A modelagem em nível de transação (TLM) (GHENASSIA, 2005) é um estilo de descrição onde detalhes da comunicação entre os componentes são separados da implementação e abstraídos em forma de transações. Esses detalhes, se necessário, podem ser trabalhados em outras etapas do projeto.

Um modelo TLM serve como referência para os desenvolvedores de software e para os desenvolvedores de hardware de um sistema. A partir de uma interface de comunicação bem definida entre os componentes, é possível antecipar o ciclo de desenvolvimento de software embarcado, impactando diretamento no *time-to-market* do produto em desenvolvimento.

1.4 Justificativa

A ausência de modelos de processadores de sinais digitais disponíveis em repositório público foi a principal motivação para concepção deste trabalho. Mesmo um dos mais ricos repositórios públicos de modelos de processadores, o repositório ArchC (ARCHC, 2008), não dispões de um modelo de DSP certificado.

1.5 Objetivos

1.5.1 Objetivos gerais

O objetivo deste trabalho foi prover um modelo puramente funcional (atemporal) do processador de sinais digitais TMS 320C4x. Como conseqüência desta modelagem, será possível gerar um modelo executável (simulador do conjunto de instruções) deste processador.

1.5.2 Objetivos específicos

- O modelo do processador C4x foi implementado na ADL ArchC;
- Foram priorizadas as instruções inteiras do processador C4x;
- Por fim o modelo foi validade através da execução de programas extraídos de três conjuntos de benchmarks conhecidos (Acstone, Mibench e MediaBench).

1.6 Processador de sinais digitais (DSP)

Nesta seção será apresentada uma breve introdução aos processadores de sinais digitais (DSPs), sua diferença em relação a outros tipos de processadores e uma visão geral de sua arquitetura. Por último, serão apresentados alguns comentários sobre seu nicho de mercado.

1.6.1 Visão geral

Ao contrário dos processadores de uso geral, que normalmente são utilizados para tarefas simples como o processamento de textos, os DSPs são otimizados para processamento digital de sinais, ou seja, aplicações com um grande número de operações matemáticas complexas. Em razão da sua grande utilização que se estende desde equipamentos de uso cotidiano como telefones celulares e dispositivos multimídia, até intrumentos de uso científico, os DSPs têm conquistado uma grande parcela do mercado de embarcados.

1.6.2 A Arquitetura de um DSP

Um dos maiores gargalos na execução de um algoritmo de processamento digital de sinais é a transferência de dados de/para a memória. Isso inclui os dados necessários para o processamento, assim como as instruções a serem buscadas. Como exemplo podemos tomar

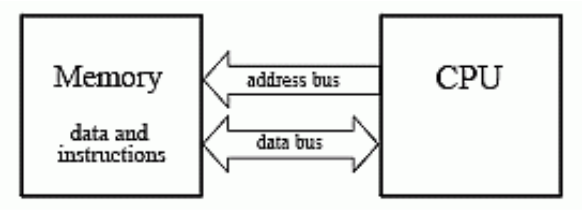

Figura 1.1: Arquitetura de Von Neumann (memória única) (GUIDE, 2008)

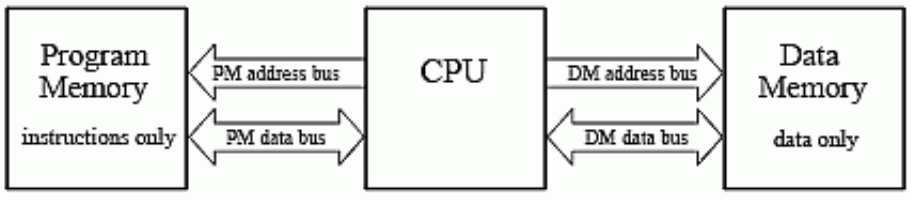

Figura 1.2: Arquitetura de Harvard (duas memórias) (GUIDE, 2008)

a multiplicação de dois números que residem em memória. Para um arquitetura tradicional, como a de **Von Neumann**, mostrada na Figura 1.1, que possui uma única memória e um único barramento para a transferência de dados de/para a CPU, a multiplicação levaria no mínimo três ciclos de relógio, entre busca de dados e instruções. Essa arquitetura satifaz a diversas aplicações, entretanto não é eficiente o bastante para atender as necessidades de processamento de uma aplicação DSP.

Para suprir as necessidades da arquitetura de **Von Neumann**, foi criada uma nova arquitetura que opera com memórias separadas, uma para dados e outra para instruções, com barramentos diferentes para cada uma delas, sendo conhecida como arquitetura de **Harvard** que é apresentada na Figura 1.2. Em conseqüência disso, é possível a busca de instruções e dados num mesmo ciclo de relógio, uma vez que os barramentos são independentes, apresentando assim um ganho significativo em relação à arquitetura de barramento único. Por esta razão, ela se faz presente em muitos DSPs.

A Figura 1.3 apresenta um nível mais alto de sofisticação conhecida como **Super Harvard Architecture** (**SHARC**). O principal objetivo desta arquitetura é diminuir a latência, através de várias otimizações, apresentando uma cache de instruções e um controlador de E/S como as duas principais. Em virtude da maior utilização do barramento de dados em relação ao de instruções, parte dos dados podem ser realocados na memória de programa (Esta realocação é chamada de *secondary data*, como mostrado na Figura 1.3). Levando-se em consideração que os algoritmos de DSP gastam a maior parte de seu tempo de execução em loops, esta se

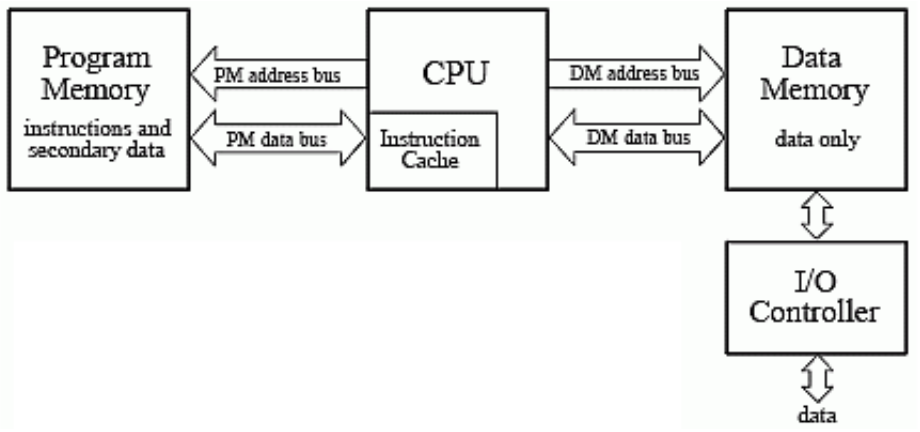

Figura 1.3: Super Harvard Architecture (duas memórias, cache de instruções, controlador de I/O)) (GUIDE, 2008)

torna uma técnica bastante vantajosa, uma vez que as instruções se repetem. Assim, ambos os barramentos ficam livres para a transferência de dados, já que as instruções necessárias para a execução do loop se encontram em cache.

A Figura 1.4 apresenta uma visão mais detalhada da arquitetura SHARC, mostrando a conexão entre o controlador de E/S ligado à memória de dados, por onde os sinais entram e saem do sistema. Esta arquitetura também apresenta portas de comunicação seriais e paralelas com grande velocidade de conexão, podendo atingir altas taxas de transmissão quando as seis portas paralelas operam em conjunto. Vale ressaltar também a grande importância do controldor DMA, que permite a transferência de dados diretamente para a memória sem ocupar CPU. Podemos observar também dois blocos etiquetados como **Data Address generator** (DAG), um para cada uma das memórias, os quais especificam o endereço dos dados a serem lidos/escritos em cada uma delas. A parte responsável pelo processamento matemático é dividida em três partes: Um multiplicador, uma unidade lógica aritmética (ALU) e um **barrel shifter**, sendo que os dois primeiros podem ser usados em paralelo. A ALU executa operações de adição, subtração, valor absoluto, operações lógicas(AND, OR, XOR, NOT), conversões entre ponto fixo e flutuante, etc. As operações binárias elementares são de responsabilidade do **barrel shifter**. Outra interessante vantagem desta arquitetura é o uso dos **shadow registers** que são usados para rápida troca de contexto, podendo as interrupções serem tratadas rapidamente. Estas e outras diversas características tornam um DSP muito mais eficiente para tarefas freqüentes em um algoritmo de processamento digital de sinais (GUIDE, 2008).

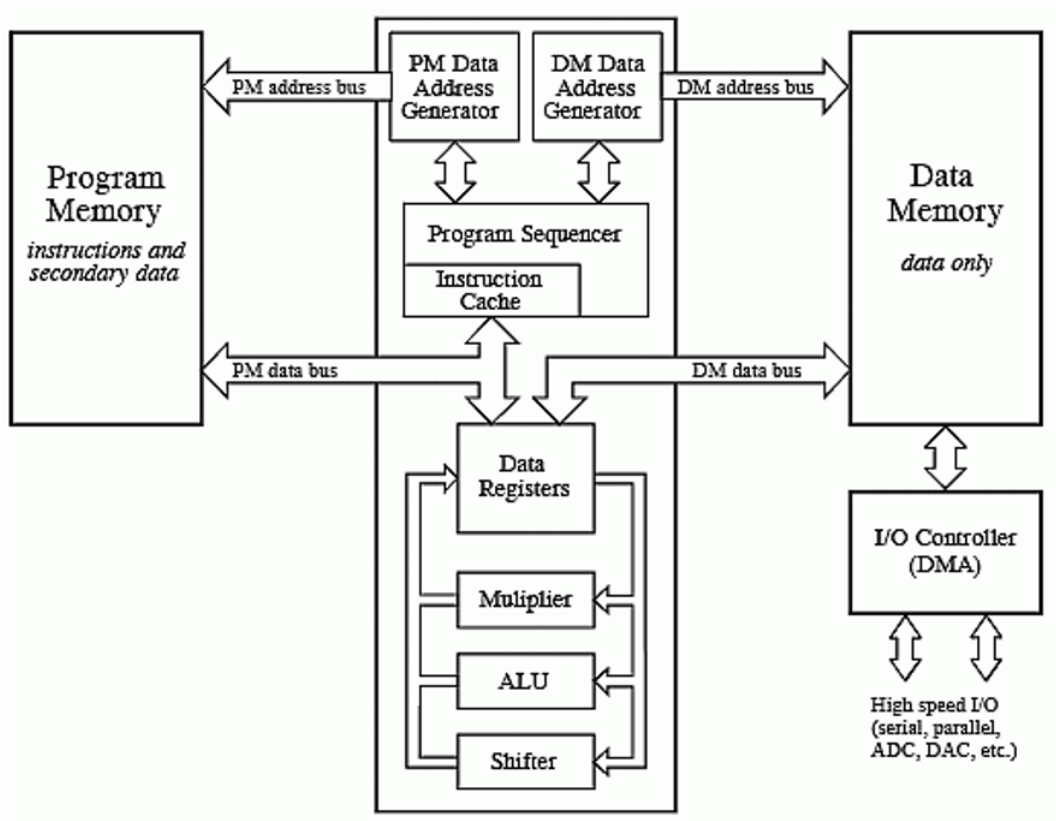

Figura 1.4: Típica arquitetura DSP. Este diagrama representa simplificadamente um SHARC DSP (GUIDE, 2008)

1.6.3 O mercado dos DSPs

O mercado para DSPs de propósitos gerais tinha expectativa de crescimento de 8% para 2007, chegando ao patamar dos nove bilhões de dólares nos EUA, de acordo com a **Forward Concepts**(CONCEPTS, 2008). Guiado pelo crescente e incessante mercado das aplicações multimídia e telecomunicações, a sua previsão de crescimento para 2008 é de 15%. Entretanto, novos estudos apontam crescimento quase duas vezes maior para o promissor mercado de DSPs embarcados, em comparação com o de propósitos gerais. De acordo com Will Strauss[3], autor do estudo, o mercado de DPSs de propósitos gerais é mais antigo e bem conhecido, sendo ele dominado por empresas bastante conhecidas como a **Analog Devices**, **Freescale**, **Agere/LSI** e a **Texas Instruments**. Em contrapartida, o mercado de DSPs embarcados surge como oportunidade para empresas emergentes, apresentando barreiras muito menores. Neste atuam empresas como a **Qualcomm**, **Broadcom**, **Marvell&Infineon**, que oferecem seus DSPs em forma de SoCs. Estudos concluem que as aplicações de comunicação dominam o mercado de propósitos gerais, sendo o celular o seu carro chefe. Já os DSPs embarcados estão mais intimamente ligados à aplicações multimídia. Podemos incluir também neste mercado as aplicações Wi-Fi, WiMax, Bluetooth, etc. Em conseqüência disso as projeções de crescimento para o mundo de DSPs tornam-se cada vez maiores (GEEK, 2008).

1.7 Organização da monografia

O Capítulo 2 aborda alguns trabalhos correlatos e apresenta uma breve descrição da ADL ArchC. O Capítulo 3 apresenta uma visão geral do processador DSP TMS320C4x, dando ênfase apenas aos detalhes relevantes para a concepção do modelo funcional. O Capítulo 4 apresenta as partes julgadas mais importantes da modelagem funcional do C4x e, por fim, lista as limitações das ferramentas usadas no decorrer do projeto. O Capítulo 5 traz detalhes dos resultados experimentais e o Capítulo 6 apresenta a conclusão da monografia e algumas possibilidades de trabalhos futuros.

[3]Presidente e principal analista da Forward Concepts, sendo considerado a principal autoridade sobre as tendências do mercado de DSPs

2 *Trabalhos correlatos*

Este capítulo apresenta uma breve introdução às Linguagens de Descrição de Arquitetura, assim como uma abordagem geral sobre algumas das principais ADLs encontradas na literatura.

2.1 Linguagens de descrição de arquitetura (ADLs)

Devido à crescente complexidade dos sistemas, as linguagens de descrição de arquiteturas (ADL) foram criadas com o objetivo de facilitar a vida dos projetistas na parte de projeto e simulação de processadores. Uma ADL permite o ajuste dos diversos recursos em estudo, como por exemplo, processador, memórias cache e principal, etc. Sendo assim, é possível um projetista avaliar diversas arquiteturas diferentes através da comparação de tempo de processamento, tamanho de código e outros diversos parâmetros cruciais para concepção de um sistema. Modelos de processadores descritos em uma linguagem de descrição de arquitetura permitem também a avaliação de um conjunto de instruções e a geração automática de ferramentas de software como simuladores, montadores, ligadores e compiladores.

2.1.1 LISA

LISA foi criada por Zivojnovic, Pees e Meyr (1996) e, inicialmente permitia a geração automática de simuladores, utilizando a técnica de simulação compilada, e com precisão de bit e de ciclos. Inicialmente, a principal contribuição de LISA foi a sua descrição de pipeline e modelo de seqüenciamento. Entretanto, essa primeira versão de LISA não era apropriada para a geração de código e montadores. Atualmente, LISA apresenta diversas extensões e é citada como a principal ADL do mercado.

2.1.2 Expression

A linguagem Expression (A.HALAMBI et al., 1999) foi criada objetivando a exploração de arquiteturas para SoCs e a geração automática de um conjunto de ferramentas com simulador e compilador. Expression segue a abordagem de misturar informações estruturais e comportamentais e permite a especificação de sistemas de memórias. Para aumentar o desempenho dos simuladores gerados a partir de modelos em Expression, foi introduzida uma técnica derivada da simulação compilada, sendo chamada de *Instruction Set Compiled Simulation*.

2.1.3 ISDL

Hadjiyiannis, Russo e Devadas (1999) criaram a linguagem ISDL (*Instruction Set Description Language for Retargetability*) juntamente com um gerador automático de montadores. ISDL foi projetada para o redirecionamente automático de compiladores, sendo especialmente voltada para a descrição de arquiteturas VLIW [1]. O conjunto de ferramentas de ISDL possui também um gerador de código, um gerador de simuladores em nível de instruções e um gerador de modelos em hardware escritos em Verilog.

2.1.4 A ADL ArchC

Criada com o objetivo de contornar as restrições de abstração impostas pela linguagem SystemC, uma nova linguagem de descrição de arquitetura foi Desenvolvida pelo Laboratório de Sistemas Computacionais (LSC) do Instituto de Computação (IC) da Universidade de Campinas (UNICAMP), denominada ArchC(ARCHC, 2008). Segundo Rigo (2004), ArchC é a primeira linguagem de descrição de arquiteturas capaz de gerar simuladores em alto nível de abstração descritos em SystemC, permitindo também tanto descrições em nível de informações estruturais quanto em nível de conjunto de instruções. ArchC, que é distribuida sob a licença GPL, possui em sua versão 2.0 um gerador automático do kit de utilitários binários (montador (BALDASSIN, 2005), ligador (CASAROTTO; SANTOS, 2006), desmontador e depurador (SCHULTZ et al., 2007)).

2.1.5 O gerador de simuladores de ArchC (acsim)

A ferramenta responsável pela geração de simuladores em ArchC é chamada de *ArchC Simulator Generator (acsim)*. O processo de utilização e geração de um simulador a partir

[1] VLIW refere-se a uma arquitetura de CPU projetada para se beneficiar do paralelismo em nível de instruções

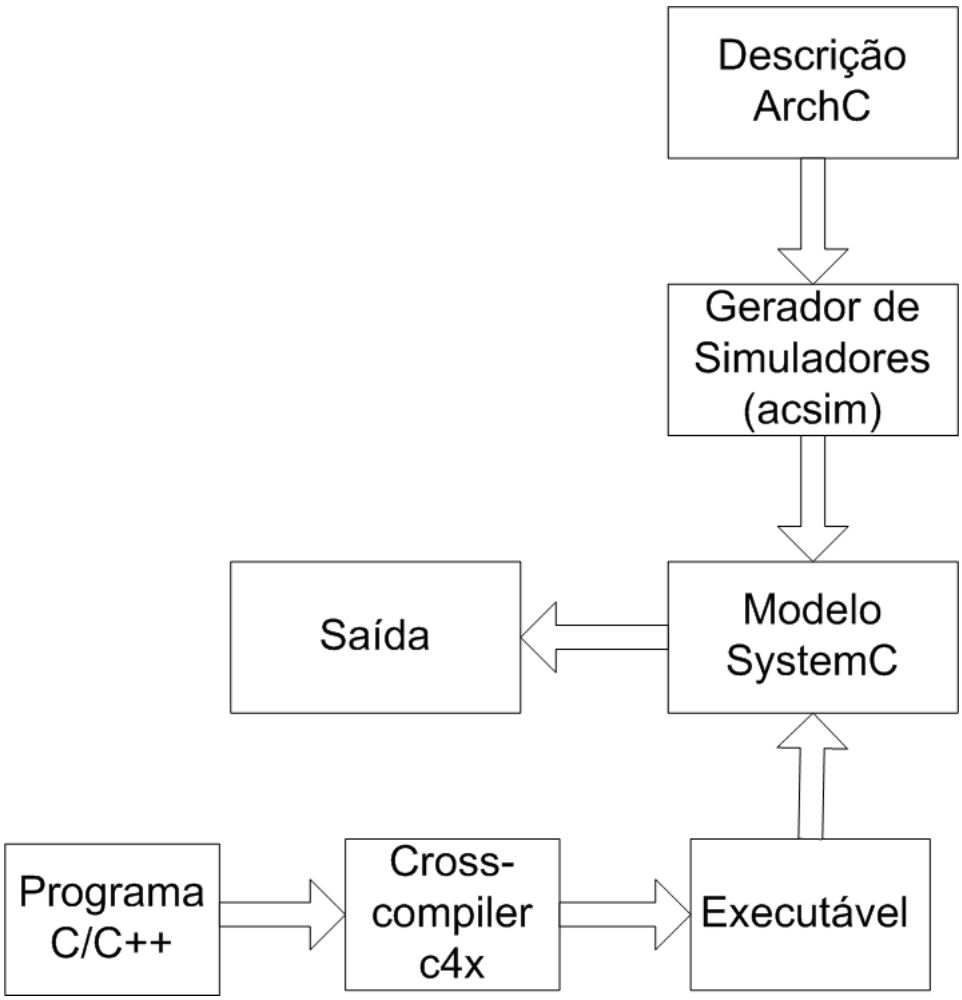

Figura 2.1: Fluxograma de Geração e Utilização do Simulador Acsim

de uma descrição em ArchC é ilustrado pela Figura 2.1. O gerador de simuladores (acsim) recebe como entrada uma descrição de processador feita em ArchC, apresentando como saída o simulador do conjunto de instruções escrito em SystemC. Por outro lado, para rodar um programa no simulador a partir de um programa escrito em C/C++, por exemplo, utiliza-se um compilador cruzado (*cross-compiler)*[2] para se obter um arquivo executável. Por fim, o arquivo objeto é executado no simulador, para então gerar um arquivo de saída contendo as informações do programa rodado.

2.1.6 A descrição ArchC

Uma descrição de arquitetura em ArchC é composta de duas partes: AC_ISA e AC_ARCH, conforme a Figura 2.2. Na parte AC_ISA é feita uma descrição do conjunto de instruções, onde o projetista faz o detalhamento do formato, nome e tamanho de instruções, assim como todas

[2]Cross-compiler é um compilador capaz de gerar um executável para uma plataforma que não seja a que o compilador está sendo executado.

Figura 2.2: Estrutura de descrição ArchC

as informações necessárias para a decodificação de cada instrução. Já a descrição AC_ARCH traz informações sobre vários detalhes da arquitetura como, por exemplo, dispositivos de armazenamento e estrutura do pipeline. O nível de detalhamento necessário para uma descrição AC_ARCH depende do nível de abstração desejado para o modelo. Uma descrição funcional (atemporal) é mais simples que uma descrição com precisão de ciclos, já que para este nível de abstração, diversos detalhes da arquitetura são descartados.

Como exemplo de algumas palavras reservadas em uma descrição AC_ARCH temos:

- **ac_wordsize**: informa o tamanho da palavra da arquitetura.

  ex: ac_wordsize 32;

- **ac_cache**, **ac_icache**, **ac_dcache**: declara um dispositivo do tipo ac_cache para armazenamento.

  ex: ac_cache CACHE:256K;

- **ac_mem**: declara um objeto do tipo ac_mem(memórias).

  ex: ac_mem MEM:5M;

- **ac_regbank**: declara um banco de registradores.

  ex: ac_regbank RB:32;

- **ac_reg**: declara um registrador.

  ex: ac_reg PC;

- **ac_pipe**: cria um pipeline, no qual devem ser informados um nome e uma lista de estágios para o pipeline que está sendo criado.

  ex: ac_pipe pipe = {IF, ID, EX, MEM, WB};

- **AC_CHTOR**: declaração de um construtor AC_ARCH.

  ex: ARCH_CTOR(tmsc4x) { ... };

- **ac_isa**: informa o nome do arquivo que contém a descrição AC_ISA usada para compor o modelo.

  ex: ac_isa("tmsc4x.ac");

- **set_endian**: define o endian da arquitetura, recebendo valores “big” ou “little”.

  ex: set_endian("little");

- **bindsTO**: estabelece uma conexão entre os dispositivos numa hierarquia de memória.

  ex: L1.bindsTo(L2);

Palavras reservadas AC_ISA:

- **ac_format**: declara um formato e seus campos.

  ex: ac_format Type_J = "%op:6 %addr:26";

- **ac_instr**: declara uma instrução com o formato fornecido.

  ex: ac_instr<Type_J> jal;

- **ISA_CTOR**: declaração de um construtor AC_ISA.

  ex: ISA_CTOR(tmsc4x);

- **ac_asm_map**: mapeamento de símbolos usados em código assembly para seus valores reais.

  ex: ac_asm_map reg { "$SP"= 20 };

- **set_asm**: define a sintaxe assembly de uma determinada instrução.

  ex: add.set_asm("add"%rd, %rs, %rt);

- **set_decoder**: Especificação de uma seqüência de decodificação de uma instrução.

  ex: add.set_decoder(op=0x00, func=0x20);

2.1.7 Modelos ArchC disponíveis

Alguns modelos de processadores já estão disponíveis e podem ser encontrados em (ARCHC, 2008).

- **PowerPC(PPC)**: Corresponde uma arquitetura RISC largamente usada em sistemas embarcados. O modelo PPC disponibilizado implementa o PowerPC 32 bits.
- **MIPS-I**: É um processador RISC amplamente utilizado para ensino por possuir uma arquitetura que se mostra fácil de ser compreendida. O modelo disponibilizado implementa a MIPS-I ISA, incluindo *delay slots* e simulação ABI [3].
- **SPARC-V8**: Uma arquitetura RISC muito conhecida por possuir características específicas e uma ISA mais complicada se comparada com a do MIPS.
- **Intel 8051**: Possui uma arquitetura CISC com uma grande e complicada ISA se comparada com uma arquitetura RISC. É um dos microcontroladores mais usados para controle em embarcados.
- **PIC 16F84**: O Microchip PIC 16F84 possui uma arquitetura RISC que é muito conhecida e utilizada.

Observando o repositório público de ArchC, verifica-se a ausência de um processador de sinais digitais descrito nessa ADL. Com o objetivo de suprir essa carência, o DSP TMS320C4x foi escolhido para modelagem.

2.2 Contribuição deste trabalho no contexto de ADLs

A principal contribuição deste trabalho é a concepção de um modelo funcional do processador de sinais digitais TMS320C4x descrito na ADL ArchC, haja vista a carência de modelos de DSPs disponíveis em outras ADLs.

[3] Do inglês, Application Binary Interface, uma ABI define um conjunto de instruções para que um arquivo objeto se comunique com o hardware

3 O processador DSP TMS 320C4x

Este capítulo apresentará algumas características e informações do processador C4x retiradas de (INSTRUMENTS, 1999) e (INSTRUMENTS, 1998).

3.1 Visão geral

O TMS320C4x é um processador de sinais digitais de 32 bits otimizado para processamento paralelo, desenvolvido pela Texas Instruments, que é utilizado tanto no mundo de DSPs de propósitos gerais quanto no de embarcados. As duas principais características da família c4x são as otimizações para trabalho com ponto flutuante e com processamento paralelo, pois combina uma CPU de alta performance e um controlador DMA com até seis portas de comunicação para atender às necessidades de multiprocessamento e de entrada e saída.

3.2 CPU

A CPU do C4x possui uma arquitetura baseada em registradores formada por diversos componentes que podem ser observados na Figura 3.1. Vale ressaltar aqui, sua grande semelhança com a arquitetura **SHARC** mostrada na Figura 1.3.

A seguir são descritos alguns componentes do C4x.

- **Floating-Point/Integer Multiplier**: executa multiplicações com números inteiros de 32 bits e números de ponto flutuante de 40 bits, levando um ciclo de relógio, independente da instrução. Caso a multiplicação seja de ponto flutuante, as entradas e o resultado são também números de ponto flutuante de 40 bits. Na operação com inteiros, a entrada é de 32 bits. Já a saída pode ser formada pelos 32 bits mais/menos significativos do produto de 64 bits.

- **Arithmetic Logic Unit**: executa operações lógicas e aritméticas em números de 32 e

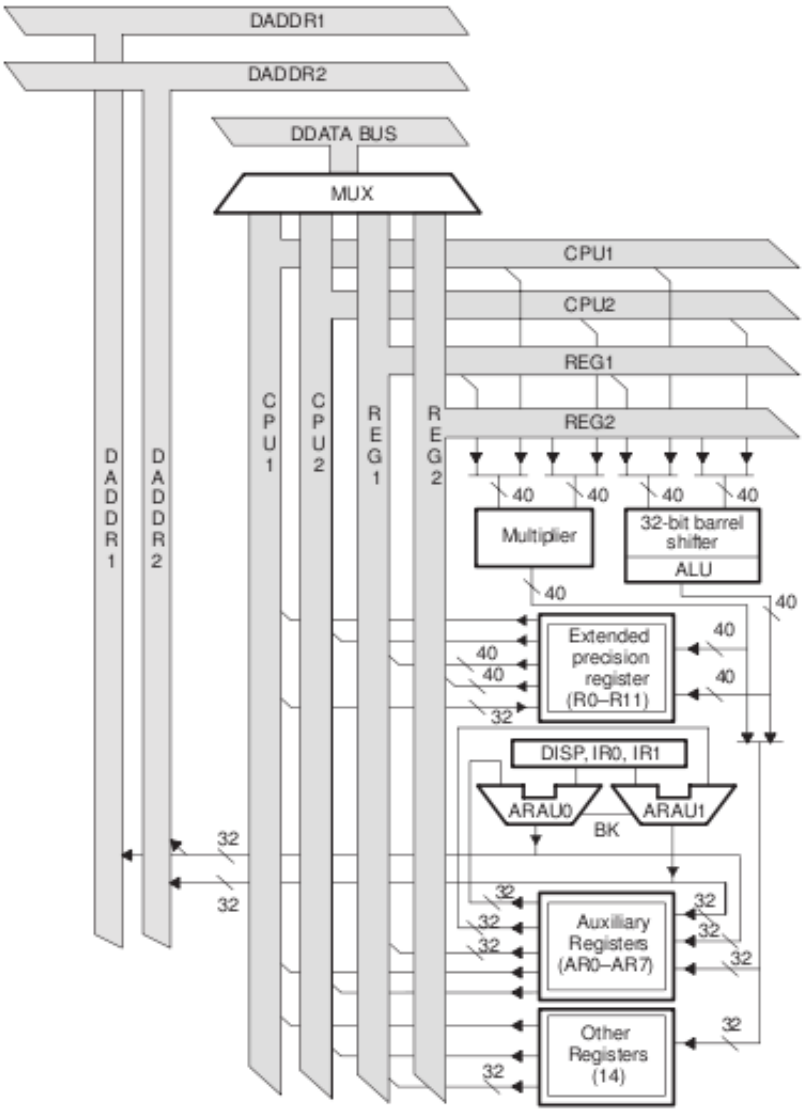

Figura 3.1: TMS320C4x CPU (INSTRUMENTS, 1999)

Registrador	Nome
R0 .. R11	Extended-precision registers
AR0 .. AR7	Auxiliary registers
DP	Data-page pointer
IR0 .. IR1	Index registers
BK	Block-size register
SP	System stack pointer
ST	Status register
DIE	DMA coprocessor interrupt enable
IIE	Internal-interrupt enable register
IIF	IIOF flag register
RS	Repeat stack address
RE	Repeat end address
RC	Repeat counter

Tabela 3.1: CPU primary register file

40 bits, levando também um único ciclo de relógio. Já o **barrel shifter** é usado para deslocamentos de 32 bits para direita ou para esquerda.

- **Internal Buses**: os 4 barramentos internos, CPU1, CPU2, REG1 e REG2, trazem dois operandos da memória e dois do banco de registradores, possibilitando o uso da ALU e do multiplicador em paralelo.

- **Auxiliary Register Arithmetic Units (ARAUs)**: as duas ARAUs podem gerar dois endereços em um único ciclo de relógio e operam em paralelo com a ALU e o multiplicador.

3.2.1 Registradores da CPU

A CPU possui 34 registradores, que são divididos em 2 grupos: **primary register file** que contém 32 registradores, e **expansion register file** que possui 2 registradires. Os primeiros 32 podem ser usados como operandos pelo multiplicador e pela ALU (Unidade lógica aritmética). Esses registradores suportam operações de ponto fixo e flutuante, endereçamento, interrupções, manipulação da pilha, etc. Já o segundo grupo é formado pelo *interrupt-vector table pointer* (IVTP) e o *trap-vector table pointer*(TVTP. Vale observar que o contador de programa(PC) não faz parte dos registradores da CPU. A Tabela 3.1 lista os registradores pertencentes a **primary register file** e seus respectivos nomes.

- **Extended-Precision Registers(R0-R11)**: armazenam e suportam operações de ponto fixo de 32 bits, assim como números de ponto flutuante de 40 bits. Para números de ponto

flutuante, são usados os bits 39-32 no armazenamento do expoente, o bit 31 é usado para sinal e os bits 31-0 armazenam a mantissa. Já para números inteiros com ou sem sinal são utilizados apenas os bits 31-0, permanecendo inutilizados os bits 39-32.

- **Auxiliary Registers(AR0-AR7)**: estes registradores de 32 bits podem ser acessados pela CPU e modificados por qualquer uma das duas ARAU. O principal objetivo dos ARs é a geração de endereços de 32 bits. Ainda assim, podem ser utilizados como registradores de propósito geral e como *loop counters* em endereçamentos indiretos.

- **Data-Page Pointer (DP)**: é um registrador de 32 bits cujos 16 bits menos significativos são usados no modo de endereçamento direto, apontando para a página do dado que está sendo buscado. O DP pode ser carregado através da pseudo-instrução LDP ou pela instrução LDI.

- **Index Registers (IR0, IR1)**: os dois IRs são usados pelas ARAUs para indexar o endereço. O IR0 para endereçamento de bit reverso (*bit-reversed*).

- **Block-Size Register (BK)**: este registrador de 32 bits é usado pela ARAU nos endereçamentos circulares, para especificar o tamanho do bloco de dados.

- **System Stack Pointer (SP)**: é um registrador de 32 bits que guarda o endereço do topo da pilha.

- **Status Register (ST)**: o registrador de status contém informações a respeito do estado da CPU.

- **DMA Coprocessor Interrupt Enable Register (DIE)**: este registrador é dividido em 6 sub-campos os quais determinam as interrupções que são utilizadas para controle de sincronização de cada um dos seis canais do DMA.

- **CPU Internal Interrupt Enable Register (IIE)**: é utilizado para habilitar ou desabilitar algumas interrupções internas da CPU.

- **IIOF Flag Register (IIF)**: controla os pinos de interrupção externa IIOF(3 – 0).

- **Repeat Stack Address (RS)**: se a CPU estiver operando em modo de repetição, este registrador armazena o endereço inicial do bloco de memória a ser repetido.

- **Repeat End Address (RE)**: se a CPU estiver operando em modo de repetição, este registrador armazena o endereço final do bloco de memória a ser repetido.

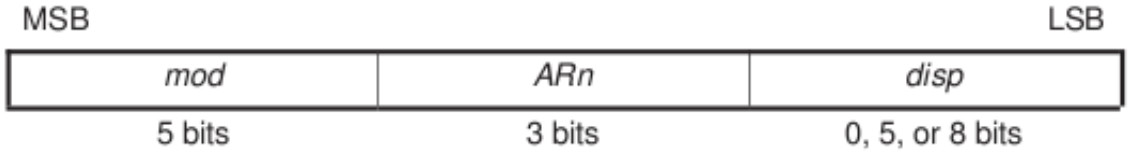

Figura 3.2: *Indirect addressing mode*

- **Repeat-Count (RC)**: caso um bloco de repetição seja criado, este registrador armazena o número de vezes que este bloco de código tem de ser repetido.

3.3 Memória

A arquitetura C4x possui dois blocos de RAM internos de 4K bytes cada um, e uma ROM interna que contém o bootloader, sendo possível dois acessos por bloco em um único ciclo de relógio. Também possui uma cache de instruções, que opera com o algoritmo *last recently used* (LRU) e tem capacidade de 128 palavras, acelerando assim a busca de instruções. Pode-se também fazer o uso das memórias externas sem comprometer o desempenho. Outra vantagem da cache é que ela não ocupa os barramentos externos com a busca de instruções, Sendo assim o barramento fica livre para outras necessidades do sistema.

3.4 Tipos de endereçamento

- **Register Addressing**: os operandos são registradores da CPU e o resultado também é salvo em um desses registradores. Vale lembrar que algumas intruções têm restrições de registradores, como por exemplo, as instruções de ponto flutuante que se restringem aos **extended-precision registers**.

- **Direct Addressing**: neste tipo de endereçamento, o endereço é obtido através da concatenação dos 16 bits menos significativos do **data page pointer**, com os 16 menos significativos da instrução.

- **Indirect Addressing**: o endereço é formado através dos 16 bits menos significativos da instrução, que é dividido em 3 campos, como mostrado na Figura 3.2. O primeiro especifica um entre os 26 modos diferentes através do conteúdo do campo **mod**, o segundo faz papel de um dos operandos e seu conteúdo se encontra em um dos 8 **auxiliary registers** e o terceiro campo especifica um deslocamento opcional.

		G	Destination	Source Operands			
31 29	28 23	22 21	20 16	15 11	10 8	7 5	4 0
0 0 0	operation	0 0	dst	0 0 0 0 0	0 0 0	0 0 0	0 0 0 0 0
0 0 0	operation	0 1	dst	direct			
0 0 0	operation	1 0	dst	mod*n*	AR*n*	disp	
0 0 0	operation	1 1	dst	Immediate			

Figura 3.3: Instruções de propósito geral

G	Mode
00	register (all CPU registers unless specified otherwise)
01	direct
10	indirect
11	immediate

Figura 3.4: Possíveis valores do campo G referente a Figura 3.3

- **Immediate Addressing**: o endereço é obtido a partir dos 8 ou 16 bits menos significativos da instrução.
- **PC-Relative Addressing**: É feita a adição do conteúdo dos 16 ou 24 bits menos significativos da instrução ao PC.

3.5 Formatos das instruções

- **Instruções de propósito geral**: se encaixam neste formato as instruções de propósitos gerais como **ADDI**, **LSH**, **ADDF**, as quais se caracterizam pela seguinte sintaxe: **operation src -> dst**.

  Como se pode observar na Figura 3.3, os bits 31-29 são 0, o que caracteriza este tipo de instrução, já os bits 28-23 definem a instrução que será executada. Os bits 22-21 especificam como os bits 15-0 serão interpretados para o operando fonte, de acordo com o tipo de endereçamento conforme visto em 3.4. Por último, os bits 20-16 definem o operando destino.

- **Instruções de três operandos**: As 19 instruções de três operandos possuem 8 formatos possíveis que são divididos em 2 tipos, como mostrado nas Figuras 3.5 e 3.6. Para o tipo T1 não há restrição a nenhuma instrução. Os bits 31-28 caracterizam o tipo. Sendo 0010

		T	Destination	src1		src2	
31 28	27 23	22 21	20 16	15 13	12 11 10 8	7 5	4 3 2 0
0 0 1 0	operation	0 0	*dst*	0 0 0	*src1*	0 0 0	*src2*
0 0 1 0	operation	0 1	*dst*	mod*n*	AR*n*	0 0 0	*src2*
0 0 1 0	operation	1 0	*dst*	0 0 0	*src1*	mod*n*	AR*n*
0 0 1 0	operation	1 1	*dst*	mod*n*	AR*n*	mod*m*	AR*m*

Figura 3.5: Tipo 1: Presente nas famílias C3x e C4x

		T	Destination	src1		src2	
31 28	27 23	22 21	20 16	15 13	12 11 10 8	7 5 4 3 2 0	
0 0 1 1	operation	0 0	*dst*	0 0 0	R*n*	Immediate	
0 0 1 1	operation	0 1	*dst*	0 0 0	R*n*	*disp*	AR*n*
0 0 1 1	operation	1 0	*dst*	*disp*	AR*n*	immediate	
0 0 1 1	operation	1 1	*dst*	*disp*	AR*n*	*disp*	AR*m*

Figura 3.6: Tipo 2: Presentes apenas na família C4x

para o tipo T1 e, 0011 para o tipo T2. Já os bits 22-21 definem como são interpretados os operandos fonte presentes nos bits 15-0. Para maiores informações a respeito deste formato, consultar a Seção 6.7.2 da referência (INSTRUMENTS, 1999).

- **Instruções paralelas**: instruções paralelas são caracterizadas por duas barras paralelas |"e possuem o formato conforme a Figura 3.7. Os bits 31 e 30 possuem o valor 10, o que indica uma instrução paralela. Já os bits 25 e 24 definem a forma que serão interpretados os bits 21-0 que são os operandos fontes. Por último, os destinos são indicados por d1 e d2.

- **Instruções de desvio condicional**: para este formato podemos tomar como exemplo instruções como Bcond, BcondD, DBcond, CALLcond e algumas outras que podem ser encontradas na tabela 3.2. Os bits 31-27 caracterizam as instruções de desvio condicional, sendo o bit 26 setado para 0 caso o formato seja DBcond, ou para 1 indicando Bcond, de acordo com a Figura 3.8. O bit 25 determina se o tipo de endereçamento é ou não relativo ao PC, informando assim a forma que serão interpretados os últimos 16 bits da instrução. O valor encontrado no bit 21 indica *standard* ou *delayed branch*. Por último, o campo *cond* especifica a condição a ser verificada para determinar a ação que será tomada.

31 30	29 26	25 24	23	22	21 19	18 16	15 11	10 8	7 3	2 0
1 0	operation	P	d1	d2	*src1*	*src2*	mod*n*	AR*n*	mod*m*	AR*m*

Figura 3.7: Codificação para multiplicação paralela com ADD/SUBB

DB*cond* (D):

3 26	25	24 22	21	20 16	15 5	4 0
0 1 1 0 1 1	B	AR*n*	D	*cond*	0 0 0 0 0 0 0 0 0 0 0	*src* reg
0 1 1 0 1 1	B	AR*n*	D	*cond*	Immediate (PC relative)	

Bcond (D):

31 26	25	24 22	21	20 16	15 5	4 0
0 1 1 0 1 0	B	0 0 0	D	*cond*	0 0 0 0 0 0 0 0 0 0 0	*src* reg
0 1 1 0 1 0	B	0 0 0	D	*cond*	Immediate (PC relative)	

Figura 3.8: Codificação das instruções de desvio condicional

3.6 Conjunto de instruções

O conjunto de instruções do C4x é formado por 145 instruções, voltadas especialmente para processamento digital de sinais e outras aplições numéricas. Todas tem o tamanho de uma palavra e, a maioria delas é executada em um único ciclo de relógio. As instruções são divididas em 6 grupos bem definidos, conforme a tabela 3.2.

3.7 Outras características do processador

Uma vez que este trabalho se restringe apenas a descrição funcional (atemporal) do C4x, não serão apresentados neste capítulo detalhes do processador que fogem do escopo desta monografia. Para obtenção de informações adicionais consulte a referência (INSTRUMENTS, 1999).

Categoria	Instruções
Load-and-Store Instructions	LBb, LBUb, LDA, LDE, LDEP, LDF, LDFcond,LDHI, LDI, LDIcond, LDM, LDPE, LDPK, LHw, LHUw, LWLct, LWRct, POP, POPF, PUSH, PUSHF, STF, STI, STIK
Two-Operand Instructions	ABSF, ABSI, ADDC, ADDF, ADDI, AND, ANDN, ASH, CMPF, CMPI, FIX, FLOAT, FRIEEE, LSH, MBct, MHct, MPYF, MPYI, MPYSHI, MPYUHI, NEGB, NEGF, NEGI, NORM, NOT, OR, RCPF, RND, ROL, ROLC, ROR, RORC, RSQRF, SUBB, SUBC, SUBF, SUBI, SUBRB, SUBRF, SUBRI, TOIEEE, TSTB, XOR
Three-Operand Instructions	ADDC3, ADDF3, ADDI3, AND3, ANDN3, ASH3, CMPF3, CMPI3, LSH3, MPYF3, MPYI3, MPYSHI3, MPYUHI3, OR3, SUBB3, SUBF3, SUBI3, TSTB3, XOR3
Program Control Instructions	Bcond, BcondAF, BcondAT, BcondD, BR, BRD, CALL, CALLcond, DBcond, IACK, IDLE, LAJ, LAJcond, LATcond, NOP, RETIcond, RETIcondD, RETScond, RPTB, RPTBD, RPTS, SWI, TRAPcond
Interlocked Operations Instructions	LDFI, LDII, SIGI, STFI, STII
Parallel Instructions	ABSF‖STF, ABSI‖STI, ADDF3‖STF, ADDI3‖STI, AND3‖STI, ASH3‖STI, FIX‖STI, FLOAT‖STF, FRIEEE‖STF, LDF‖STF, LDI‖STI, LSH3‖STI, MPYF3‖STF, MPYI3‖STI, NEGF‖STF, NEGI‖STI, NOT‖STI,OR3‖STI, STF‖STF, STI‖STI, SUBF3‖STF, TOIEEE‖STF, SUBI3‖STI, XOR3‖STI, LDF‖LDF, LDI‖LDI, MPYF3‖ADDF3, MPYF3‖SUBF3, MPYI3‖ADDI3, MPYI3‖SUBI3

Tabela 3.2: Conjunto de instruções do C4x.

4 *Modelagem funcional*

Este capítulo apresenta as partes julgadas mais importantes da modelagem funcional do C4x. Também são relatadas as limitações encontradas nas ferramentas utilizadas no decorrer do projeto. As informações necessárias para a modelagem do processador C4x foram obtidas em (INSTRUMENTS, 1999) (INSTRUMENTS, 1998).

4.1 Instruções capturadas do modelo

O presente modelo prioriza as instruções inteiras do processador C4x, assim 87 de um total de 145 instruções foram modeladas. O gráfico ilustrando as instruções do processador divididas em classes é apresentado na Figura 4.1.

4.2 Modelagem do processador DSP TMS 320C4x

O modelo funcional do processador C4x é composto de três arquivos. Primeiramente, será apresentado o arquivo tmsc4x.ac, que contém detalhes da arquitetura, como pode ser observado na Figura 4.2. Iniciamos com a definição do tamanho da palavra da arquitetura **ac_wordsize**, logo em seguida a quantidade de memória disponível para este modelo **ac_mem**. Continuamos com declaração de dois bancos de registradores **ac_regbank**. Os **extended-precision register** e **auxiliary registers** são declarados na seqüência. Mais adiante podemos observar a descrição de mais alguns registadores **ac_reg**, a definição do arquivo que possui a declaração das instruções **ac_isa** e, por último, é definido o endianess da arquitetura através da palavra reservada **set_endian**.

No próximo arquivo, tmsc4x_isa.ac, mostrado na Figura 4.3, estão definidas as formas de codificação das instruções. Começamos com a descrição dos formatos das instruções, conforme explicado na seção 3.5, através da palavra reservada **ac_format**, para a associação de um formato, utilizamos **ac_instr**. Em seguida declaramos um **ac_asm_map**, que faz o mapeamento

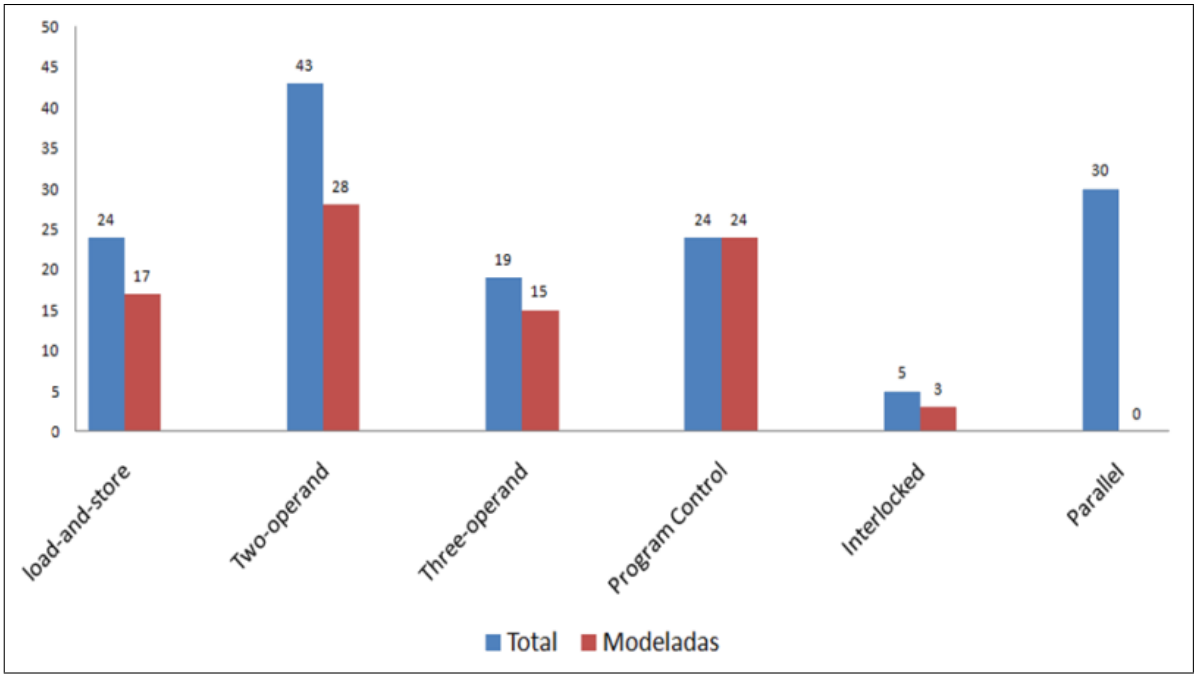

Figura 4.1: Instruções capturadas no modelo.

entre os símbolos da arquitetura e seus respectivos valores. Por fim, o construtor, onde é feita a correspondência entre os mnemônicos das instruções e seus respectivos valores para decodificação.

Já o terceiro e último arquivo, tmsc4x_isa.cpp, contém o comportamento das instruções anteriormente declaradas. É importante observar que para este arquivo, recursos oferecidos por C++/SystemC podem ser usados. O comportamento das intruções descritas em ArchC segue um fluxo de execução bem definido, como pode ser visto na Figura 4.4. Primeiramente, é executado o comportamento *instruction*, como na Figura 4.5, onde é feito o incremento do *program counter* (PC), que é comum para todas as intruções do processador. Depois é executado o comportamento referente ao formato da instrução e, por fim, o comportamento específico da instrução.

Na Figura 4.6, observamos o comportamento do tipo G, que define o tipo de endereçamento a ser calculado. Para o tipo *register addressing* (00), o conteúdo do registrador é colocado na variável *srcField*. Seguimos com o tipo *direct addressing* (01), onde é feita a concatenação dos 16 bits menos significativos do registrador *DP* com os 16 menos significativos da instrução, assim o operando fonte é obtido através do endereço calculado. Para o tipo *indirect addressing* (10), o operando é buscado no endereço de memória calculado pelo função *calc_IndAdress*. Por último, o tipo *immediate addressing* (11), onde *srcField* recebe os 16 bits menos significativos da instrução.

```
AC_ARCH(tmsc4x){

        ac_wordsize 32;

        ac_mem   MEM:8M;

        ac_regbank R:12;

        ac_regbank AR:8;

        ac_reg DP;

        ac_reg IR0;

        ac_reg RC
        ...

        ac_reg IVTP;

        ac_reg TVTP;

    ARCH_CTOR(tmsc4x) {

    ac_isa("tmsc4x_isa.ac");
    set_endian("little");

  };
};
```

Figura 4.2: Descrição dos detalhes da arquitetura do C4x.

A Figura 4.7 traz a descrição da instrução LDI, que faz a carga de um registrador através do conteúdo de memória apontado pela variável *srcField* mostrada acima. Para finalizar a execução da instrução, é feita uma atualização do registrador de status da CPU, através da função *set_CondFlags*.

ArchC permite também emular chamadas ao sistema operacional através do arquivo tmsc4x_syscall.cpp, que deve implementar as especializações das funções da classe ac_syscall. Outra vantagem do simulador gerado por ArchC, é a possibilidade de depuração do modelo através da implementação das funções definidas em tmsc4x_gdb_funcs.h.

As funcionalidades descritas acima não foram implementadas para o modelo do C4x.

```
ac_format G = "[%src:16 | %disp:8 %arn:3 %modn:5] dst:5 g:2 opG:6 fG:3";

ac_format PCrel = "srcPC:24 del:1 opPCrel:7";
...

ac_instr<G> ABSI, AND;

ac_instr<Cond> Bcond, BcondD, RETS;
....

ac_asm_map reg {
        "$"[0..31] = [0..31];
        "R"[0..7]  = [0..7];
        "R"[8..11] = [28..31];
        "AR"[0..7] = [8..15];
        "$DP" = 16;
        ...

        "$RC" = 27;
}

ISA_CTOR(tmsc4x)
{
...
};
```

Figura 4.3: Descrição do formato e conjunto de instruções do C4x.

4.3 Limitações das ferramentas

Nesta seção serão relatadas as limitações das ferramentas utilizadas no decorrer da modelagem do processador C4x.

4.3.1 Limitações da ADL ArchC

1. ArchC não permite a utilização de caracteres não alfanuméricos em construções **ac_asm_map**, conforme a Figura 4.8.

   - **Consequência**: dificulta a construção de instruções com mnemônicos do tipo **ADDI *+AR1(IR1), R0**, já que símbolos como "*+"poderiam ser mapeados em um **ac_asm_map** diretamente para o seu valor de codificação/decodificação.
   - **Solução**: sobrecargas da sintaxe assembly da instrução foram utilizadas para contornar essa limitação.

2. A construção **ac_asm_map** não possibilita a utilização de um mesmo símbolo em mapas diferentes;

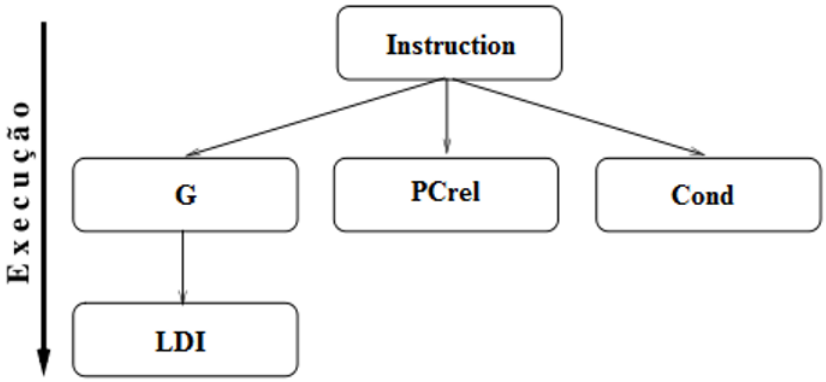

Figura 4.4: Fluxo de execução das instruções.

```
//!Generic instruction behavior method.
void ac_behavior( instruction )
{
.       ac_pc+=4;
};
```

Figura 4.5: Descrição do comportamento genérico das instruções do C4x.

- **Consequência**: impossibilidade de um mesmo símbolo ser mapeado em diferentes construções **ac_asm_map** para valores distintos.
- **Solução**: um modificador foi utilizado junto ao formatador especificado na sintaxe assembly da instrução para fazer uma alteração no valor correspondente ao símbolo especificado na instrução.

3. A construção ac_format não permite que dois formatos de instruções diferentes tenham o mesmo nome para um determinado campo, salvo se tiverem o mesmo tamanho e estiverem na mesma posição. A Figura 4.9 apresenta um exemplo de construção não permitida, já que o campo **src** apresenta tamanhos diferentes para cada um dos formatos apresentados.

   - **Consequência**: se dois formatos diferentes utilizarem um mesmo nome para um determinado campo, estes deverão ter o mesmo tamanho. Caso contrário, os nomes destes campos devem ser diferentes.
   - **Solução**: foram utilizados nomes distintos para campos que possuem mesmo tamanho e posição em diferentes instruções.

4. A linguagem apresenta restrições quanto ao mnemônico das instruções, não sendo possível representar instruções como na Figura 4.10.

```
void ac_behavior( G )
{
.       unsigned int address;
.       unsigned short int gField = g;
.
.       switch(gField) {
.       .       case 0x0:
.       .       {
.       .       .       srcField = reg_read(src);
.       .       }
.       .       break;
.       .       case 0x1:
.       .       {
.       .       .       address = DP.read() << 0x10;
.       .       .       address |= src;
.       .       .       srcField = MEM.read(address);
.       .       }
.       .       break;
.       .       case 0x2:
.       .       {
.       .       .       address = calc_IndAddress(modn, arn, (disp << 2));
.       .       .       srcField = MEM.read(address);
.       .       }
.       .       break;
.       .       case 0x3:
.       .       {
.       .       .       srcField = ((int)src << 0x10) >> 0x10;
.       .       }
.       .       break;
.       }
}
```

Figura 4.6: Descrição do comportamento para o formato G.

- **Consequência**: mnemônicos não permitidos na linguagem devem ser alterados para contornar essa limitação.
- **Solução**: mnemônicos como o da Figura 4.10, foram alterados para **ABSI_STI**.

4.3.2 Limitações das ferramentas de compilação e validação

Por se tratar de um processador proprietário, a fabricante (*Texas Instruments*), não diponibiliza gratuitamente uma ferramenta de compilação para o C4x. O alto custo de aquisição destas ferramentas fez necessária a utilização de um compilador GNU, já que este é disponibilizado em repositório público (FOUNDATION, 2008).

1. Apesar de alguma documentação poder ser obtida em (HAYES, 2001), várias dificuldades foram encontradas na geração do *cross-compiler* para arquitetura C4x, principalmente por esta documentação ser pobre e muito antiga.

```
//!Instruction LDI behavior method.
void ac_behavior( LDI )
{
.       reg_write(dst,srcField);
.
.       //negative result
.       if(srcField & 0x80000000)
.       .       set_CondFlags(2,2,0,1,0,0,2);
.       else if(srcField == 0)
.       .       set_CondFlags(2,2,0,0,1,0,2);
.       else
.       .       set_CondFlags(2,2,0,0,0,0,2);
}
```

Figura 4.7: Descrição do comportamento da instrução LDI.

```
ac_asm_map mod{.
.       "*+" = 00000;
.       "*-" = 00000;
.       "*++" = 00000;
.       "*--" = 00000;
}
```

Figura 4.8: Exemplo de construção ac_asm_map não permitida.

```
ac_format G = "%fG:3 %opG:6 %g:2 %dst:5 %src:16";

ac_format PCrel = "%opPCrel:7 %del:1 %src:24";
```

Figura 4.9: Exemplo de construção não permitida para ac_format.

```
    ABSI   src2, dst1
||  STI    src3, dst2
```

Figura 4.10: Exemplo de mnemônico não permitido.

- **Consequência**: diversos problemas relacionados às versões dos programas envolvidos na geração do *cross-compiler* se tornaram constantes no decorrer do projeto.
- **Solução**: os passos para a geração do *cross-compiler* estão descritos com detalhes no Apêndice A.

2. Limitação da ferramenta gerada (acsim), de possuir único formato binário de entrada *Executable and Linkable Format* (ELF).

3. Limitação do *cross-compiler*, de possuir único formato binário saída; *Common Object*

File Format (COFF).

- **Consequência**: impossibilidade do simulador de fazer a leitura dos arquivos-objeto gerados pelo *cross-compiler*.
- **Solução**: para contornar este problema, foi utilizado um conversor COFF->ELF obtido em (ARM, 2008).

4. Não há porte de um depurador para o C4x disponível na árvore GDB (FOUNDATION, 2007).

   - **Consequência**: impossibilidade de depuração dos programas com o uso de uma ferramenta adequada.
   - **Solução**: foi feita uma depuração manual dos resultado, imprimindo na tela os passos executados por cada instrução.

5 *Resultados experimentais*

5.1 Configuração experimental

Os experimentos foram realizados em um computador com processador AMD Turion(tm) 64 X2, com freqüência de 1,8 GHz, 512 KB de cache L2, com 512 MB de memória principal. Foi utilizado o sistema operacional Ubuntu GNU/Linux, com kernel versão 2.6.22. A versão da ADL ArchC foi a 2.0 e os programas dos benchmarks foram compilados utilizando-se o *cross-compiler* C4x-GCC 3.0 fornecido pela GNU (FOUNDATION, 2008), a partir da versão 2.95.

5.1.1 O conjunto de programas Acstone

O conjunto de programas Acstone foi desenvolvido com pela equipe ArchC 2.1.4 com a finalidade de apontar falhas básicas na modelagem, uma vez que se tratam de programas simples. O Acstone é composto de 75 programas dos quais 67 trabalham somente com instruções de inteiras e 8 trabalham com intruções inteiras e de ponto flutuante (061.div.c ... 068.div.c).

5.2 Validação

A validação do modelo implementado foi feita a partir da execução do conjunto de programas pertencentes ao Acstone, possibilitando assim a correção de erros básicos na modelagem. A verificação do comportamento das instruções foi feita manualmente, imprimindo na tela os passos executados por cada uma, já que não há porte de um depurador Gnu para o C4x, conforme visto na seção 4.3.2.

As Tabelas 5.1 e 5.2 e a Figura 5.1, apresentam o número de instruções, divididas em classes, executadas em cada um dos programas do Acstone. Vale observar que os programas que trabalham com ponto flutuante não foram validados. Essa decisão partiu da limitação do *cross-compiler*, o qual não emula instruções de ponto flutuante, sendo assim esses não foram testados.

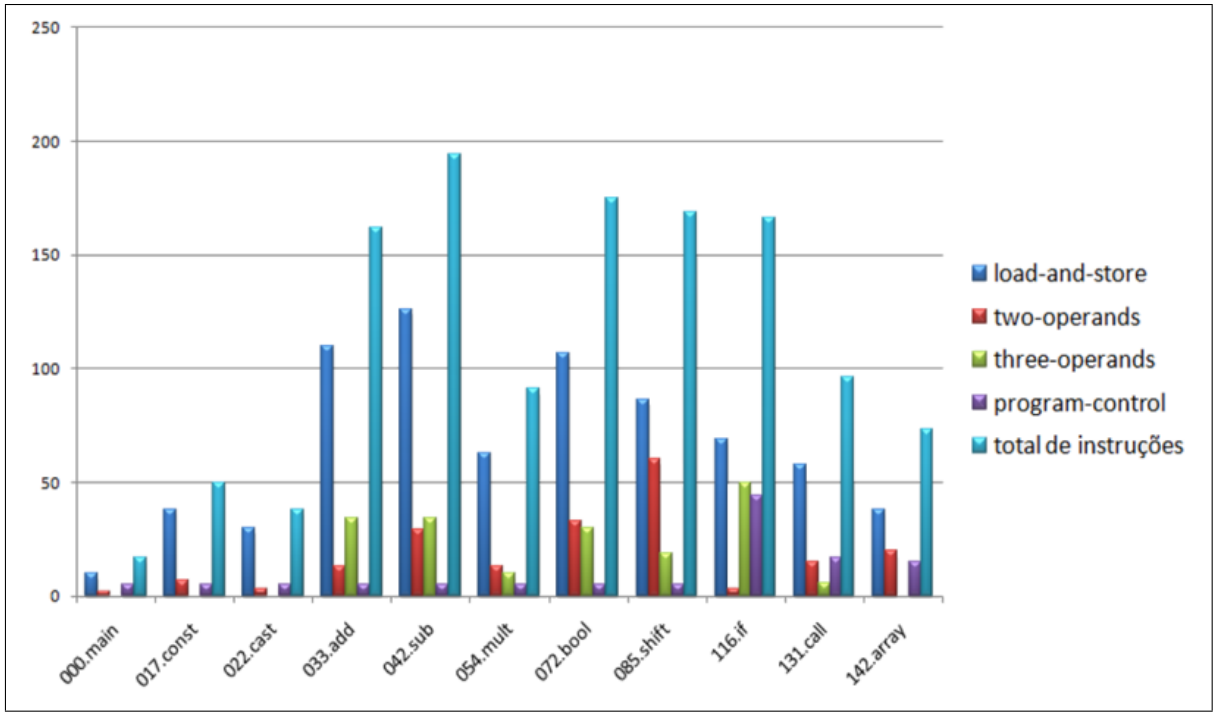

Figura 5.1: Cobertura obtida com o ACSTONE (distribuição em classes (Mix)).

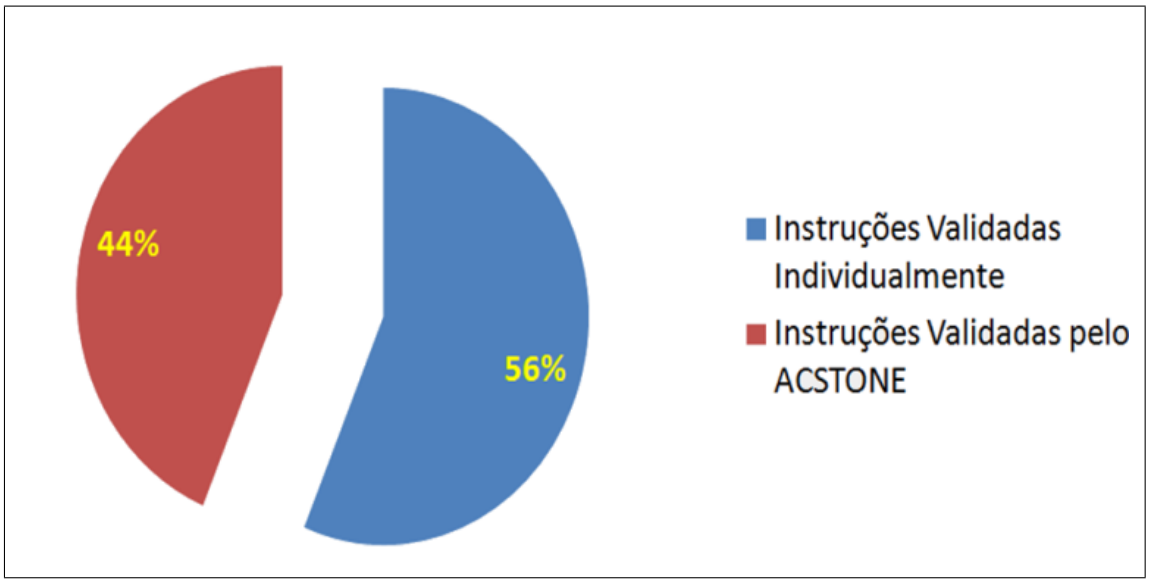

Figura 5.2: Cobertura obtida com o ACSTONE (global).

O conjunto de programas Acstone cobre 49 das 87 instruções do modelo como pode ser visto na Figura 5.2. As demais 39 instruções foram validadas individualmente, através de inspeção do resultado, para que posteriormente possam ser validadas nos conjuntos mibench e mediabench.

Programa	Load-and-Store	Two-Operands	Three-Operands	Program-Control	Total de Instruções
000.main.c	10	2	0	5	17
011.const.c	24	3	0	5	32
012.const.c	24	3	0	5	32
013.const.c	24	7	0	5	36
014.const.c	24	7	0	5	36
015.const.c	22	6	0	5	33
016.const.c	22	6	0	5	33
017.const.c	38	7	0	5	50
018.const.c	38	7	0	5	50
021.cast.c	30	3	0	5	38
022.cast.c	30	3	0	5	38
023.cast.c	39	3	4	5	51
024.cast.c	30	5	0	5	40
025.cast.c	39	5	4	5	53
026.cast.c	38	4	4	5	53
027.cast.c	32	5	0	5	52
031.add.c	122	13	24	5	164
032.add.c	126	29	34	5	194
033.add.c	110	13	34	5	162
034.add.c	262	55	28	5	350
041.sub.c	122	13	24	5	164
042.sub.c	126	29	34	5	194
043.sub.c	110	13	34	5	162
044.sub.c	262	55	28	5	350
051.mul.c	44	3	7	5	59
052.mul.c	44	3	7	5	59
053.mul.c	63	13	10	5	91
054.mul.c	63	13	10	5	91
055.mul.c	230	88	78	31	428
056.mul.c	230	88	78	31	428
057.mul.c	112	36	27	13	188
058.mul.c	114	36	25	13	188
071.bool.c	105	28	30	5	168
072.bool.c	107	33	30	5	175
073.bool.c	104	32	30	5	171
074.bool.c	202	57	60	5	324
075.bool.c	64	29	0	5	98

Tabela 5.1: Número de instruções em cada programa Acstone (distribuição em classes (Mix)) - TMSC4X Funcional (parte 1)

5.3 Dificuldades na validação

Os programas do Acstone que possuem intruções de ponto flutuante (061.div.c ... 068.div.c), não foram validados, uma vez que estas não foram implementadas.

Programa	Load-and-Store	Two-Operands	Three-Operands	Program-Control	Total de Instruções
081.shift.c	110	15	12	5	142
082.shift.c	110	27	12	5	154
083.shift.c	106	23	12	5	146
085.shift.c	86	60	19	5	169
111.if.c	73	3	50	38	164
112.if.c	73	3	50	42	168
113.if.c	73	7	50	38	168
114.if.c	73	7	50	42	172
115.if.c	69	3	50	40	162
116.if.c	69	3	50	44	166
119.if.c	24	3	9	14	50
123.loop.c	379	247	177	175	978
124.loop.c	479	295	288	246	1308
131.call.c	58	15	6	17	96
132.call.c	141	55	6	21	223
142.array.c	38	20	0	15	73
143.array.c	300	72	195	206	773
144.array.c	302	86	195	206	789
145.array.c	302	110	195	206	813

Tabela 5.2: Número de instruções em cada programa Acstone (distribuição em classes (Mix)) - TMSC4X Funcional (parte 2)

Foram encontrados problemas nos programas 084.shift.c, 117.if.c, 118.if.c, 141.array.c e 146.array.c, os quais acessam uma determinada área de memória desconhecida. Ja os programas 121.loop.c, 122.loop.c, 125.loop.c, 126.loop.c, 133.call.c e 134.call.c, não terminam sua execução. Estes problemas estão sendo avaliados junto a um interlocutor da Unicamp para que possam ser resolvidos.

6 *Conclusão*

6.1 Conclusão

O estilo de projeto em nível de sistema eletrônico se torna cada vez mais necessário para lidar com a crescente complexidade dos SoCs. Um modelo de processador disponível ao projetista possibilita a exploração do espaço de projeto de SoCs sem a necessidade de adquirí-lo, podendo assim avaliar os diversos fatores necessários para escolher o que melhor se adequa a aplicação em desenvolvimento. Um modelo de processador permite também, aos desenvolvedores de software embarcado antecipar o desenvolvimento do software dependente de hardware, como firmwares e drivers, impactando diretamente no time-to-market do produto em desenvolvimento.

Este trabalho tem como produto o primeiro modelo de DSP a ser disponibilizado em repositório público ArchC. É importante observar que em razão das diversas limitações das ferramentas usadas no decorrer do projeto, conforme visto na seção 4.3.2, a validação experimental do modelo ainda esta em andamento.

6.2 Trabalho em andamento

As instruções modeladas no presente trabalho estão sendo validadas através da execução de programas retirados dos benchmarks Mibench e Mediabench para o modelo atingir o nível de certificação 0.7.0 e então ser disponibilizado em repositório público ArchC (ARCHC, 2008).

6.3 Trabalhos futuros

São sugeridas algumas alterações na ADL ArchC em face das limitações descritas na seção 4.3.2.

- Possibilitar a inclusão de caracteres não alfanuméricos na construção **ac_asm_map**.

- Permitir a criação de símbolos iguais em duas construções **ac_asm_map** diferentes.
- Alterar a construção **ac_format**, para possibilitar que dois formatos diferentes possuam um campo de mesmo nome, tamanho e posição.
- Incluir uma maior flexibilidade para os mnemônicos das intruções, como por exemplo o ilustrado na Figura 4.10.

Outra possibilidade de trabalho futuro seria modelagem com precisão de ciclos do processador C4x.

Referências Bibliográficas

A.HALAMBI et al. Expression: A language for architecture exploration through compiler/simulator retargetability. In: *Proc. European Conference on Design, Automation and Test(DATE)*. [S.l.: s.n.], 1999.

ARCHC. *The ArchC Architecture Description Language*. [S.l.]: http://www.archc.org, Jun 2008.

ARM. *ARM - the architecture for the digital world*. 2008. Disponível em: <http://www.arm.com/support/downloads/info/2320.html>.

BAILEY, B.; MARTIN, G.; PIZIALI, A. *ESL Design and Verification*. [S.l.]: Elsevier, 2007.

BALDASSIN, A. *Geração automática de montadores em ArchC*. Dissertação (Mestrado) — Instituto de Computação, UNICAMP, Campinas, Março 2005.

CASAROTTO, D. C.; SANTOS, L. C. V. Automatic link editor generation for embedded cpu cores. In: *4th Internation IEEE-NEWCAS Conference, 2006, Gatineau, Canada. Proceedings of the 4th International IEEE-NEWCAS Conference. Piscataway, NJ, USA : IEEE Press, 2006. p. 121-124.* [S.l.: s.n.], 2006.

CONCEPTS, F. *Forward Concepts Electronics Market Research*. 2008. Disponível em: <http://www.fwdconcepts.com/>.

FOUNDATION, F. S. *GNU Binutils*. 2006. Disponível em: <http://www.gnu.org/software/binutils/binutils.html>.

FOUNDATION, F. S. *GDB: The GNU Project Debugger*. [S.l.]: http://www.gnu.org/software/gdb/gdb.html, Jun 2007.

FOUNDATION, F. S. *GCC, the GNU Compiler Collection*. 2008. Disponível em: <http://gcc.gnu.org/>.

GEEK, E. *DSP Market to Grow by 8% in 2007*. 2008. Disponível em: <http://edageek.com/2007/04/24/dsp-2007/>.

GHENASSIA, F. *Transaction-Level Modeling with SystemC: TLM concepts and applications for embedded systems.* Springer 2005.

GROTKER, T. et al. *System Design with SystemC*. [S.l.]: Kluwer Academic Publishers, 2002.

GUIDE, D. *The Scientist and Engineer's Guide to Digital Signal Processing*. 2008. Disponível em: <http://www.dspguide.com/>.

HADJIYIANNIS, G.; RUSSO, P.; DEVADAS, S. A methodology for accurate performance evaluation in architecture exploration. In: *Design Automation Conference (DAC), pages 927-932*. [S.l.: s.n.], 1999.

HAYES, M. *TMS320C[34]x DSP GNU Tools*. [S.l.]: http://www.elec.canterbury.ac.nz/c4x/, Jun 2001.

INSTRUMENTS, T. *TMS320C3x/C4x Assembly Language Tools User's Guide*. 1998. Disponível em: <http://focus.ti.com/lit/ug/spru035d/spru035d.pdf>.

INSTRUMENTS, T. *TMS320C4x User's Guide*. 1999. Disponível em: <http://focus.ti.com/lit/ug/spru063c/spru063c.pdf>.

KRASNER, J. *Embedded Software Development Issues and Challenges: Failure Is NOT An Option - It Comes Bundled With The Software*. july 2003.

RIGO, S. *ArchC: Uma linguagem de descrição de arquiteturas*. Tese (Doutorado) — Instituto de Computação, UNICAMP, Campinas, Junho 2004.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and Challenges for Platform-Based Design. In: *Proceedings of the 41st Annual Conference on Design Automation (DAC-04)*. [S.l.]: ACM Press, 2004. p. 409–414.

SCHULTZ, M. R. O. et al. A model-driven automatically-retargetable debug tool for embedded systems. In: *7th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2007), Samos, Greece. Proceedings of the 7th SAMOS Workshop, LNCS 4599, pp. 13-23.* [S.l.: s.n.], 2007.

SOURCES.HADHAT.COM. *Newlib*. [S.l.]: http://sourceware.org/newlib/, Jun 2008.

SYSTEMC. *The Open SystemC Initiative*. [S.l.]: http://www.systemc.org, Jun 2008.

ZIVOJNOVIC, V.; PEES, S.; MEYR, H. Lisa - machine description language and generic machine model for hw/sw co-design. In: *Proceedings of the IEEE workshop on VLSI Signal Processing, San Francisco*. [S.l.: s.n.], 1996.

APÊNDICE A – Geração do Cross-Compiler para o C4x e seu Porte para ArchC

A geração do *cross-compiler* está descrita a seguir:

1.A maquina utilizada deverá ter instalado o GCC-3.3 (FOUNDATION, 2008).

2.Obter o código fonte do GNU Binutils-2.14 (FOUNDATION, 2006), do GCC-3.0 (FOUNDATION, 2008) e da biblioteca Newlib-1.12 (SOURCES.HADHAT.COM, 2008).

3.Crie uma cadeia de pastas no diretório "/"da seguinte forma: "/l/archc/compilers/c4x".

4.Crie um diretório "build-binutils"para fazer a compilação do código do binutils e, a partir dele configure o binutils da seguinte maneira: "../binutils-2.14/configure –target=c4x –prefix=/l/archc/compilers/c4x". Em seguida digite o comando make install.

5.Após compilado o GNU-Binutils, deverá ser feito um link simbólico desses utilitários criados para a pasta "/usr/bin".

6.Crie um link simbólico da pasta "newlib", que se encontra junto ao código fonte da newlib-1.12, para a pasta onde se encontra o código fonte do gcc-3.0.

7.Crie um diretório "build-gcc"para fazera compilação do código do gcc e, a partir dele configure o Gcc da seguinte maneira: "../gcc-3.0/configure –target=c4x –prefix=/l/archc/compilers/c4x –with-newlib -enable-language=c,c++". Em seguida digite o comando make install.

8.Crie um link simbólico do compilador gerado pelo GCC para a pasta "/usr/bin".

9.Alterar a linha 44 do arquivo "configure.host", que se encontra na pasta "newlib"junto ao código fonte da newlib-1.12, para "newlib_cflags= "${newlib_cflags} -DCOMPACT_CTYPE.

```
*link:
-L/l/archc/compilers/ac_specs/c4x -Tac_link.ld

*startfile:


*endfile:


*lib:
-lc -lac_sysc
```

Figura A.1: Arquivo ac_specs

10.Crie um diretório "build-newlib"para fazer a compilação do código da newlib e, a partir dele configure a newlib da seguinte maneira: "../newlib-1.12/configure –target=c4x –prefix=/l/archc/compilers/c4x". Em seguida digite o comando make install.

11.Crie uma cadeia na pastas no diretório "/l/archc/compilers/"da seguinte forma: "/l/archc/compilers/ac_specs/c4x".

12.No diretório "/l/archc/compilers/ac_specs/c4x"crie um arquivo "ac_specs", conforme a Figura A.1.

13.Crie um link simbólico com o nome archc no diretório GCC specs da seguinte forma: "ln -s /l/archc/compilers/ac_specs/c4x/ac_specs /l/archc/compilers/c4x/lib/gcc-lib/c4x/3.3.1/archc".

14.No diretório "/l/archc/compilers/ac_specs/c4x"crie um arquivo "ac_link.ld"baseado no arquivo "/l/archc/compilers/c4x/c4x/lib/ldscripts/tic4xcoff.x"e substitua a seção *SECTIONS* pela ilustrada nas Figuras A.2 e A.3.

15.Crie um arquivo "ac_start.s"no diretório "/l/archc/compilers/ac_specs/c4x", conforme a Figura A.4.

16.Compile o arquivo "ac_start.s"da seguinte forma: "c4x-gcc -c ac_start.s".

17.Por fim, recompile a biblioteca *ArchC System Calls Library*, obtida em (ARCHC, 2008), da seguinte forma: "make TARGET="c4x-gcc e em seguida copie o arquivo "libac_sysc.a"para o diretório "/l/archc/compilers/ac_specs/c4x".

```
SECTIONS
{
  /* Program code.  */
  .text : {
  /l/archc/compilers/ac_specs/c4x/ac_start.o(.text)
     . = 0x100;
      __text =  .;
     *(.init)
    *(.text)
     ___CTOR_LIST__ = .;
     LONG(___CTOR_END__ - ___CTOR_LIST__ - 2)
     *(.ctors)
     LONG(0);
     ___CTOR_END__  = .;
     ___DTOR_LIST__ = .;
     LONG(___DTOR_END__ - ___DTOR_LIST__ - 2)
     *(.dtors)
     LONG(0)
     ___DTOR_END__  = .;
     *(.fini)
      __etext =  .;
}  > EXT0
  /* Reset, interrupt, and trap vectors.  */
  .vectors  0 : {
    *(.vectors)
}  > EXT0
  /* Constants.  */
  .const : {
    *(.const)
}  > EXT0
  /* Global initialised variables.  */
  .data :
```

Figura A.2: Arquivo ac_link.ld (parte 1)

18.A compilação de um código C, por exemplo, é feita da seguinte maneira:

"c4x-gcc –specs=/l/archc/compilers/ac_specs/c4x/ac_specs arquivo.c"

```
{
      __data  =  .;
    *(.data)
      __edata  = .;
}  > EXT0
  /* Global uninitialised variables.  */
  .bss : {
      __bss  =  .;
    *(.bss)
    *(COMMON)
      __end  =  .;
}  > EXT0
  /* Heap.  */
  .heap :
{
     __heap  =  .;
     . += __HEAP_SIZE;
}  > EXT0
  /* Stack (grows upward).  */
  .stack :
{
     __stack  =  .;
    *(.stack)
     .  =  . + __STACK_SIZE;
}  > EXT0
  .stab 0 (NOLOAD) :
{
    [ .stab ]
}
  .stabstr 0 (NOLOAD) :
{
    [ .stabstr ]
}
}
```

Figura A.3: Arquivo ac_link.ld (parte 2)

```
        .global _start
_start:
        push    ar3
        lda     sp,ar3
        addi    2,sp
        call    _main
        lda     r0,ar2
        call    _exit
        .ref    _main
        .ref    _exit
        .end
```

Figura A.4: Arquivo ac_start.s