Síntese de linguagens de descrição de arquitetura

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Samuel Shoji Fukujima Goto e aprovada pela Banca Examinadora.

Campinas, 25 de agosto de 2010.

Prof. Dr. Rodolfo Jardim de Azevedo
Instituto de Computação - UNICAMP
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Miriam Cristina Alves – CRB8 / 5089

Goto, Samuel Shoji Fukujima
G712s Síntese de linguagens de descrição de arquitetura/Samuel Shoji Fukujima Goto -- Campinas, [S.P. : s.n.], 2010.

Orientador : Rodolfo Jardim de Azevedo
Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Circuitos lógicos – Projetos e construção. 3. Sistemas e computação. 4. Hardware – Linguagens descritivas. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Architecture description languages synthesis

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Logic circuits – Design and implementation. 3. Computer systems. 4. Hardware – Description languages.

Área de concentração: Arquitetura de Computador

Titulação: Mestre em Ciência da Computação

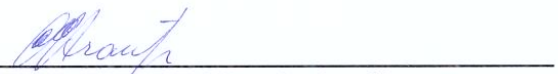
Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC – IMECC)
Prof. Dr. Paulo Cesar Centoducatte (IC – IMECC)
Prof. Dr. Cristiano Coêlho de Araújo (UFPE)

Data da defesa: 23/06/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

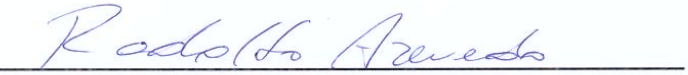
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 23 de junho de 2010, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Cristiano Coêlho de Araújo
Cin / UFPE

Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Síntese de linguagens de descrição de arquitetura

Samuel Shoji Fukujima Goto

Agosto de 2010

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação - UNICAMP (Orientador)
- Prof. Dr. Paulo Cesar Centoducatte
  Instituto de Computação - UNICAMP
- Prof. Dr. Cristiano Coêlho de Araújo
  Centro de Informática - UFPE
- Prof. Dr. Jorge Luiz e Silva
  Departamento de Sistemas de Computação - USP (Suplente)
- Prof. Dr. Sandro Rigo
  Instituto de Computação - UNICAMP (Suplente)

Resumo

Desde sua popularização, processadores dobraram de capacidade e desempenho à cada dois anos. No entanto, paralelamente, essa tendência foi apenas sustentada pelo crescimento da sofistificação das implementações utilizadas. Atualmente, apesar de eficientes, processadores são complexos e difíceis de projetar. Para gerenciar esse problema, foram criadas linguagens chamadas ADLs que simplificam a especificação e simulação em níveis mais abstratos, enquanto HDLs ainda são utilizadas para a descrição RTL. Esse trabalho unifica o fluxo de especificação e simulação de processadores com o fluxo de implementação RTL a partir da mesma linguagem ADL. Para isso, escolhemos uma linguagem de descrição de arquitetura chamada ArchC. Sintetizamos com sucesso parte de processadores descritos em ArchC, como o PIC16F84, o I8051, o MIPS-I, o R3000 e uma JVM. Subconjuntos dos processadores foram prototipados em FPGA, com frequências de operação entre 80MHZ à 120MHZ projetados duas a três vezes mais rapidamente do que os desenvolvidos com HDLs. Mostramos que ArchC é sintetizável, completando o fluxo de projeto da linguagem até o nível RTL. Criamos as ferramentas necessárias e fornecemos modelos RTL dos processadores citados.

Abstract

The design and implementation of processors is a complex task. Architecture Description Languages (ADLs) were created to extend existing HDLs to manage the inherit complexity of modern processors. Along with HDLs, they ease the development and prototyping of new architectures by providing a set of tools and algorithms to optimize and automate some of the tedious parts. However, while much has been done on using ADLs for simulating high level specifications, the academia knows very little about how to reuse them to implement real life processors. This work addresses the issues of synthesizing processors from an ADL model. To accomplish that, we chose an ADL called ArchC and we successfully synthesized pieces of its most stable models, like the PIC16F84, the i8051, the MIPS-I, the R3000 and a JVM. The processors were prototyped in FPGAs, with frequencies of operation as fast as 80MHZ to 120MHZ developed two to tree times faster compared to current approaches. We show that ArchC is in fact synthesizable, completing the design flow down to the RTL level. We provide all the necessary tools that were created to synthesize the models as well as the RTL models themselves.

Agradecimentos

Agradeço profundamente meu orientador, Rodolfo, pela confiança, inspiração, direção, clareza, educação e amizade.

Aos professores Paulo e Cristiano pelas revisões e sugestões. Ao professor Guido pelas inúmeras contribuições na minha formação pessoal e profissional.

Aos responsáveis pela secretaria de graduação e pós-graduação, Daniel, Flavio, Ademilson e Seu Nelson, pela importância e graça no serviço prestado. Ao Instituto de Computação pela formação e aos professores pelo desafio.

Aos meus pais, pelo carinho.

À minha esposa, pela paciência, compreensão e apoio.

Para os meus pais, com carinho.
Para a minha esposa, com amor.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Circuitos lógicos são uma das abstrações mais fascinantes criadas pelo homem. Com apenas algumas primitivas, eles são capazes de dar sentido e propósito à dispositivos físicos. Por baixo de todo aparelho eletrônico utilizado no nosso dia-a-dia existe um circuito lógico que o controla: televisões, computadores, Internet, tocadores de MP3, carros etc.

Para entender porque eles são úteis e como eles funcionam, considere o seguinte problema: dados dois números binários `a` e b, como você criaria um mecanismo físico para calcular o resultado da soma de `a` e `b`?

O primeiro passo para resolver qualquer problema é entendê-lo. O que significa somar dois números binários?

Vamos enumerar todos as situações possíveis. `a` e `b` são variáveis binárias, então elas só podem assumir dois valores (0 ou 1, falso ou verdadeiro respectivamente): se `a` for 0 e `b` for 0, `a` soma de `a` + `b` = 0 + 0 é 0; se `a` for 1 e `b` for 0, `a` soma de `a` + `b` = 1 + 0 é 1; se `a` for 0 e `b` for 1, `a` soma de `a` + `b` = 0 + 1 é 1; se `a` for 1 e `b` for 1, `a` soma de `a` + `b` = 1 + 1 é 10 (ou 2 em decimal);

Dessa observação, podemos reescrever:

$S \leftarrow A \oplus B$

$C \leftarrow A \cdot B$

$soma \leftarrow C * 2^1 + S * 2^0$

Onde $\oplus$ é o operador lógico "ou-exclusivo" (verdade se apenas um dos operandos é verdadeiro e falso caso contrário) e $\cdot$ é o operador lógico "e" (verdade se ambos os operandos são verdadeiros e falso caso contrário). Por exemplo, suponha que `a` seja 0 e `b` seja 1: S é igual `a` 1 (porque apenas `b` é verdadeiro na operação $\oplus$) e C é 0 (porque apenas `b` é verdadeiro na operação $\cdot$). Portanto, o resultado da soma de `a` = 0 e `b` = 1 é igual `a` 1 porque $0 * 2^1 + 1 * 2^0 = 1$.

Felizmente, atualmente existem dispositivos físicos que implementam os operadores

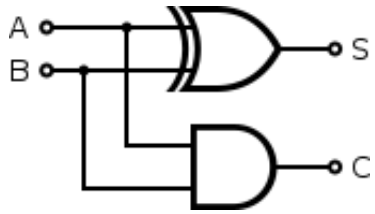


Figura 1.1: Somador de dois bits

lógicos $\oplus$ e $\cdot$ descritos anteriormente, chamados de portas lógicas. A Figura 1.1 ilustra como somadores são construídos na prática com portas lógicas: A e B alimentam a entrada das portas lógicas $\oplus$ e $\cdot$ que geram o resultado S e C descritos anteriormente. Os valores de A e B são dados como entrada nesse circuito e o resultado da computação da soma S e C são obtidos a partir deles. É com um circuito como esse que calculadoras são construídas, assim como a maioria dos computadores.

Note como, nesse processo, temos a definição do problema (como somar dois números binários?), uma solução abstrata representada matematicamente (equações) e uma solução concreta representada como um diagrama (circuitos lógicos). Note também que, nesse processo, é importante saber quais são os dispositivos concretos disponíveis (portas lógicas). O processo entre a representação abstrata e a implementação concreta chama-se síntese lógica que é o tema dessa dissertação.

Agora, por mais fascinante que circuitos lógicos sejam, somar dois bits não é tão interessante assim. Afinal, calculadoras, computadores, televisões, vídeo-games fazem muito mais do que isso.

De fato, circuitos lógicos podem representar muito mais do que soma de bits, e podem se tornar imensamente complexos em função da quantidade de lógica que o circuito implementa. Um dos resultados apresentados nesse trabalho, por exemplo, é um circuito com mais de 2 mil portas lógicas!

Para resolver esse problema, foram criadas representações e abstrações que simplificam a descrição de um circuito. Nesse caso, por exemplo, mostramos duas representações da lógica de somadores, uma abstrata e outra concreta: fórmulas matemáticas e desenhos esquemáticos.

Por um tempo, essas representações foram suficientes e os circuitos eram projetados usando esses diagramas esquemáticos. No entanto, os equipamentos eletrônicos foram se popularizando, os circuitos foram implementando cada vez mais lógica e essas representações esquemáticas se tornaram difíceis de gerenciar.

Essa dissertação trata exatamente desse problema: dada uma descrição abstrata, seja ela qual for (fórmulas, desenhos, documentos, palavras etc), de uma funcionalidade complexa (processadores, vídeo-games etc), quais são as características dessa descrição que a torna representável em um circuito lógico que a implemente e quais transformações são

necessárias para torná-la concreta?

Dado a abrangência de problemas que podem ser resolvidos com circuitos lógicos, esse problema é importante por diversos motivos. O principal deles é o estudo dos processos envolvidos em tornar ideias abstratas em mecanismos concretos. Idealmente, gostaríamos de descrever o problema o mínimo possível e ter como resultado um mecanismo que resolva o problema da melhor forma possível. Gostaríamos de falar "eu gostaria de fazer X" e um circuito lógico que resolvesse X fosse construído.

Resolver esse problema significa permitir o desenvolvimento de novos circuitos lógicos de maneira mais conveniente e eficiente. Definir como a descrição da funcionalidade deve ser representada e como essa descrição se transforma no mecanismo que a implemente significa prover diretivas de como a descrição de 'eu quero fazer X' deve ser feita e prover o algoritmo que a transforma na implementação. Significa também saber responder se 'X não pode ser feito' a partir da mesma pergunta.

Para entender o contexto e propósito desse problema, é importante entender o ecossistema das equipes de desenvolvimento de *hardware*, em particular processadores. O projeto de processadores envolve diversas equipes: as equipes variam desde engenheiros responsáveis pelo desenvolvimento de compiladores, bibliotecas, sistemas operacionais, placas, componentes externos, fabricação até os profissionais responsáveis pelo marketing, requisitos, logística e distribuição dos processadores. Essas equipes trabalham com diferentes especificações entre elas que representam contratos que definem quais são as características do produto final de cada equipe. Isso significa que, com uma especificação do processador, os engenheiros responsáveis pelo compilador conseguem projetar o compilador inteiro antes mesmo do processador ser fabricado ou até mesmo projetado. Analogamente, a equipe de marketing consegue ter uma noção geral de qual o tipo de performance e mercado o processador final ira se encaixar sem depender de atrasos da equipe de engenharia.

Esse trabalho aborda o problema descrito à partir do estudo de uma instância específica: a síntese de processadores à partir de representações dadas em linguagens de descrições de arquitetura (ADLs).

O principal resultado desse trabalho é a descrição dessas características e a implementação desses algoritmos para uma linguagem cuja síntese, até o momento da publicação dessa dissertação, não foi resolvida.

Para validar as ideias propostas nesse trabalho, escolhemos uma linguagem de descrição de arquitetura chamada ArchC, provemos as ferramentas que implementam a síntese dessa linguagem e mostramos como o circuito final pode ser utilizado prototipando os processadores em *field-programmable gate arrays* (FPGAs). Utilizamos o método escolhido com descrições já existentes dos processadores i8051, PIC16F84, MIPS-I, R3000 e a JVM e mostramos o circuito que os implementa com frequências de operação de até 120MHz.

Mostramos que o tempo de desenvolvimento de processadores com esse método é duas a três vezes mais curto que o tempo de desenvolvimento com HDLs.

No segundo capítulo, as principais abordagens para o problema descrito são analisadas. Tanto processadores parametrizáveis quanto outras linguagens de descrição de arquitetura são considerados e comparados.

O terceiro capítulo começa com uma breve introdução à linguagem que foi utilizada, ArchC. O fluxo de desenvolvimento com a linguagem é descrito e o método é proposto como uma extensão. Os principais conceitos representados na linguagem são analisados e um circuito é proposto para cada construção. Os algoritmos utilizados para a síntese dos processadores são descritos.

No quarto capítulo, os conceitos propostos no terceiro capítulo são colocados em prática. A implementação dos algoritmos da síntese é mostrada e o processamento das representações é descrito, desde a linguagem ADL até o diagrama esquemático. A metodologia de verificação proposta também é analisada e os resultados mostrados. A metodologia proposta é comparada com os trabalhos relacionados.

Por fim, no quinto capítulo, algumas considerações finais são feitas. As possíveis consequências e extensões desse trabalho são apresentadas, assim como as restrições.

Capítulo 2

Trabalhos relacionados

Enquanto muito tem sido feito na área de Linguagens de Descrição de Arquitetura (LDA), a maioria dessas linguagens atualmente suporta apenas alguns aspectos da síntese dos processadores descritos com elas [22]. Na grande maioria dos trabalhos publicados na área, o interesse em linguagens dessa categoria tem sido no sentido de ajudar na simulação, teste e exploração de alternativas. As principais linhas de desenvolvimento, tanto na academia quanto na indústria, ainda contam com re-escrever os projetos de *hardware* em linguagens de mais baixo nível, como, por exemplo, Verilog ou VHDL.

Apesar disso, existe uma grande pressão da comunidade e da indústria em resolver o problema da crescente complexidade dos projetos de processadores e como isto impacta a produtividade dos desenvolvedores [21]. Existem duas abordagens para lidar com a complexidade dos processadores: prototipação rápida dos processadores e a automatização do processo. A primeira, diz respeito à quão rápido os projetistas conseguem obter um protótipo das idéias sendo projetadas. Isso envolve ser capaz de explorar e comparar as diferentes alternativas de projeto do processador. A segunda, diz respeito à identificar as partes que podem ser automatizadas e implementa-las (por exemplo, geração de montadores e simuladores). Nesse contexto, as duas principais abordagens para esse problema são os chamados processadores parametrizáveis e a síntese de ADLs.

Processadores parametrizáveis são processadores base que provêem algum tipo de extensibilidade. Síntese de ADLs são os algoritmos e ferramentas utilizados para transformar processadores descritos em uma linguagem de descrição de arquitetura em circuitos lógicos.

Embora este trabalho esteja mais relacionado com a segunda abordagem, serão apresentadas, brevemente, as principais alternativas comerciais e acadêmicas da primeira. Após isso, será descrito o que tem sido publicado a respeito do estado atual das ADLs e explicado mais detalhadamente a metodologia e resultados das linguagens que suportam algum tipo de simulação e síntese no nível RTL dos seus modelos. Por fim, serão descritos

alguns trabalhos e esforços em otimizações que podem ser feitas em ADLs assim como algumas tendências de trabalhos recentes.

2.1 Processadores parametrizáveis

Os processadores parametrizáveis são processadores base que podem ser extendidos a partir de um conjunto de ferramentas e linguagens. O projetista começa com um processador base e o estende (por exemplo, adicionando novas instruções personalizadas, configurando o tamanho das caches, do barramento de memória etc) conforme as necessidades da aplicação que o processador terá.

O processador é inteiramente projetado para ser modificado e o nível de flexibilidade que ele oferece depende de opção para opção. As modificações podem ser classificadas como estáticas (processadores parametrizáveis) ou dinâmicas (processadores reconfiguráveis), dependendo de quão flexível o processador é com relação às modificações em tempo de execução. Existem diversos compromissos entre flexibilidade e desempenho. Exemplos do primeiro caso que são tipicamente citados são o PEAS e o Xtensa da Tensilica, e exemplos do segundo caso são os processadores da Chameleon, da BOPS e da PACT.

Em [18] são listados os principais esforços acadêmicos e industriais, assim como são descritos quais são as principais linhas de desenvolvimento na área. São mostradas, também, tecnologias de projeto que ajudam a identificar as reconfigurações necessárias e como explora-las efetivamente.

Nessa seção, as duas principais alternativas serão descritas: o PEAS e o Xtensa da Tensilica.

2.1.1 PEAS

O PEAS [13] (*practical environment for application specific integrated processor development*) é uma plataforma de desenvolvimento que permite que o projetista use como entrada uma série de programas específicos da sua aplicação em C e um conjunto de dados de entrada, restrições do projeto (como, por exemplo, área e consumo de potência) e gera automaticamente a descrição VHDL de uma CPU otimizada para a sua aplicação, assim como um conjunto de ferramentas (compiladores, montadores etc).

O processador gerado é baseado na arquitetura Harvard com os barramentos de instruções e de dados fixado em 32 bits.

O APA (*Application Program Analyser*) da ferramenta toma como entrada os programas em C e conjuntos de dados e faz analises estáticas e dinâmicas para determinar quais são as partes mais computacionalmente relevantes dos programas. A execução dos programas de teste e o conjunto de dados geram informações e estatísticas de uso de

Tabela 2.1: Exemplos de opções de configuração do Xtensa

Parâmetro	Opções
Tamanho da cache de instruções	1, 2, 4, 8 e 16Kb
Tamanho da cache de dados	1, 2, 4, 8 e 16Kb
Tamanho da RAM	1, 2, 4, 8 e 16Kb
Quantidade de registradores	32, 64 registradores
Largura do barramento externo	32, 64, 128 bits
Quantidade de interrupções	0-32
Níveis de interrupção	0-6
Temporizadores	0-3
Ordem da memória	Big-endian, little-endian

cada parte da arquitetura. Em seguida, uma outra parte da ferramenta, chamada AIG (*Architecture Information Generator*), recebe como entrada a informação de perfil gerado pelo APA, e define um conjunto de instruções especializado e otimizado para a execução do conjunto de teste.

2.1.2 AutoTIE e Xtensa da Tensilica

O Xtensa é um processador estaticamente parametrizável desenvolvido pela Tensilica. Os autores em [12] advogam a escolha pela parametrização estática devido ao fato dos processadores dinamicamente reconfiguráveis serem limitados pela frequência de operação entre os circuitos reconfiguráveis e o processador base. Xtensa facilita a extensão de processadores que anteriormente eram tipicamente desenvolvidos como co-processadores. A Tabela 2.1 mostra alguns dos parâmetros que podem ser configurados no Xtensa, como tamanhos das caches de instruções e dados, interrupções disponíveis, ordem da memória etc.

O Xtensa possui um conjunto base de 80 instruções, baseado em um superconjunto das tradicionais instruções de 32 bits dos RISCs [20]. A implementação típica do *hardware*, mostrada na Figura 2.1, é baseada no tradicional *pipeline* de cinco estágios do RISC: busca da instrução, decodificação, execução, acesso à memória e escrita nos registradores.

O TIE (*Tensilica's instruction extension*) é uma linguagem de alto nível que expressa a semântica e codificação das instruções e é utilizada para estender o processador. As instruções definidas pelo usuário rodam em uma unidade de execução pré definida, mostrada na Figura 2.1. A Figura 2.1 mostra também como algumas partes do processador podem ser usadas opcionalmente, como o suporte à exceções e módulos de depuração. As partes opcionais são utilizadas ou não conforme as restrições do projeto no que diz respeito à desempenho e área disponível.

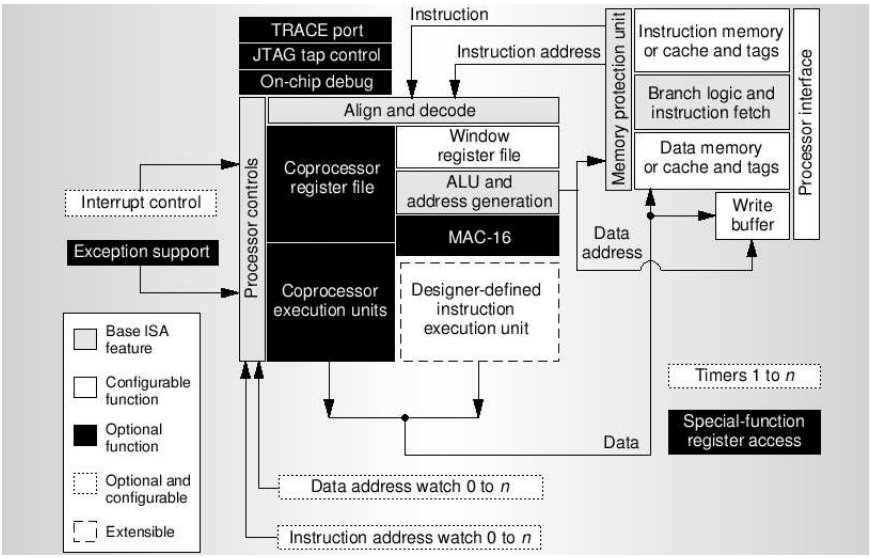


Figura 2.1: Diagrama de blocos do Xtensa

O AutoTIE [11] é um sistema que simplifica a criação de ASICs (*Application Specific Integrated Circuits*) estendendo automaticamente um processador base com um conjunto de instruções otimizado, operações e registradores. O AutoTIE gera automaticamente a integração com compiladores C/C++ para suportar as novas instruções. A ferramenta provê múltiplas alternativas, que podem ser utilizadas para comparar o balanço entre custo de *hardware* e desempenho.

2.2 Síntese de Linguagens de Descrição de Arquitetura

Em [21] o autor oferece uma introdução ao estado atual das ADLs no que se refere à diversos aspectos, como geração de ferramentas, simuladores, verificação etc. Como pode ser visto na Tabela 2.2 apenas algumas linguagens como Mimola, UDL, ISDL, EXPRESSION, LISA e AIDL oferecem algum tipo de suporte à síntese dos seus modelos, enquanto NML, HMDES, RADL não oferecem [21]. Vale notar que MIMOLA e UDL são originalmente HDLs e são, por consequência, naturalmente sintetizáveis. EXPRESSION e LISA, como será discutido mais adiante, são as ADLs que oferecem o suporte mais extenso à síntese dos seus modelos.

Em [22] é detalhada a comparação entre as ADLs - categorias, expressividade, ferramentas etc -, e é descrito o suporte à síntese detalhadamente. A Figura 2.2 mostra a estrutura base adotada pela maioria das ferramentas de síntese das ADLs. De forma simplificada, as ferramentas se baseiam em modelos genéricos RTL que são combinados com um gerador de HDL a partir da especificação ADL. Esses modelos RTLs são tipi-

Tabela 2.2: Comparação das ADLs no que se refere ao suporte à síntese

	MIMOLA	UDL	NML	ISDL	HMDES	EXPRESSION	LISA	RADL	AIDL	ARCHC
Geração de Compiladores	x	x	x	x	x	x	x			o
Geração Simuladores	x	x	x	x	x	x	x	x	x	x
Simulação cycle-accurate	x	o		x	x	x	x	x	o	o
Verificação Formal						o			o	
Síntese	x	x		o		x	x		o	
Geração de testes	x		x			x	x			

x Suportado
o Suportado com restrições

camente baseados no RISC-DLX [20]. As principais HDLs escolhidas para representar a descrição em nível de *hardware* são Verilog e VHDL. Essa abordagem é bastante similar com a adotada por esse trabalho.

Nesta seção os detalhes da metodologia de cada ADL que suporta síntese no nível RTL serão descritos, assim como os resultados e problemas encontrados.

2.2.1 EXPRESSION

EXPRESSION [1, 2] usa uma técnica de abstração funcional para gerar automaticamente descrições sintetizáveis a partir de especificações ADLs. O desenvolvimento, mostrado na Figura 2.3, tipicamente começa com uma especificação do processador no nível mais abstrato (por exemplo, um documento em inglês ou português sobre o nicho de mercado que o processador vai se localizar) até passar por diversos processos manuais e automatizados (como compiladores, geradores de código etc) que levam a descrição para o nível mais concreto (por exemplo, uma descrição física dos transístores envolvidos).

EXPRESSION utiliza descrições do processador estruturais e comportamentais, assim como um mapeamento entre essas descrições. Estruturalmente, EXPRESSION divide a descrição do grafo em nós (unidades, armazenamento, portas e conexões) e arestas (áreas de *pipeline* e arestas de transferência de dados). Arestas de *pipeline* descrevem como os dados são transferidos entre as unidades do *pipeline* e arestas de dados especificam como os dados são transferidos entre componentes.

EXPRESSION traduz a sua ADL para VHDL, utilizando primitivas de abstração funcional, que são depois sintetizadas usando o *Design Compiler* da Synopsys [39].

O código VHDL gerado tem três principais componentes: o decodificador de instruções, o *datapath* e a lógica de controle.

O decodificador de instruções usa a descrição do formato das instruções para decidir como decodificar cada instrução, e o mapeamento das instruções às unidades funcionais para decidir como e para onde despachar cada instrução decodificada.

O *datapath* é gerado em duas partes: primeiramente, cada componente é instanciado e

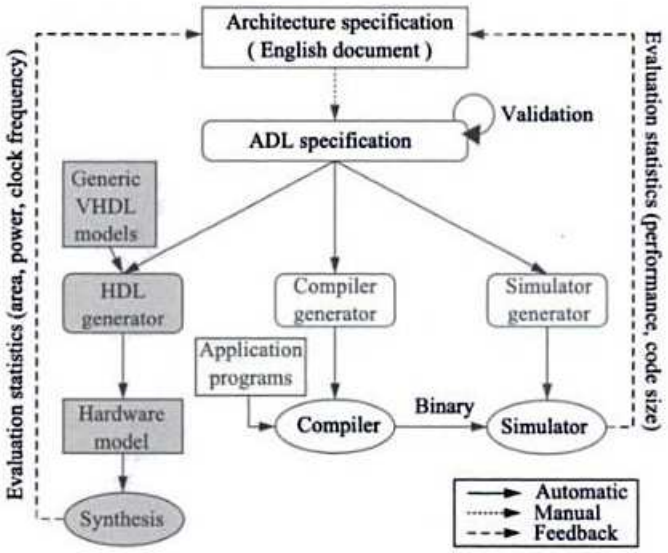


Figure 6.7 ADL-driven implementation generation and exploration

Figura 2.2: Estrutura básica adotada na Síntese de ADLs

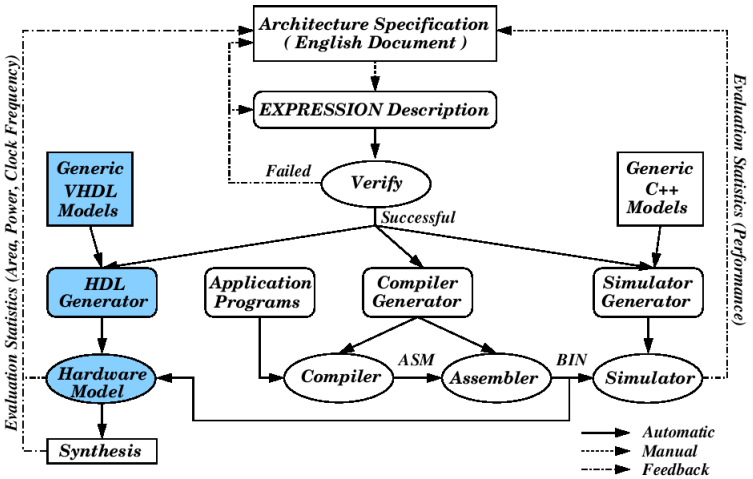


Figura 2.3: Fluxo de projeto de EXPRESSION

Tabela 2.3: Comparação do RISC-DLX gerado pela EXPRESSION e pelo PEAS

Método	Especificação (palavras)	Código HDL (palavras)	Área (gates)	Desempenho (Mhz)
RISC-DLX (ADL)	2063	6612	118K	33
PEAS-DLX (Parametrização)	1196	6259	105K	5.3

Tabela 2.4: Comparação do DLX superescalar gerado por EXPRESSION e descritos manualmente

Método	Área (gates)	Desempenho (Mhz)	Potência (mW)
EXPRESSION-DLX (ADL)	239K	20	108.27
Darmstadt-DLX (HDL)	198K	27.7	72

a partir disso eles são compostos e as conexões entre eles são estabelecidas. Para compor cada unidade, EXPRESSION avalia o que é necessário para implementar a funcionalidade. EXPRESSION gera lógica de controle inter e intra funcionais.

Em [1], o autor reporta sua técnica aplicada para síntese do processador DLX e como seus resultados se comparam com o DLX gerado pelo PEAS como mostra a Tabela 2.3. Versões superelescalares e VLIW do DLX também são comparadas com versões escritas manualmente como mostra a Tabela 2.4. Nos dois casos, o tempo de desenvolvimento da rápida exploração do projeto é considerada a principal vantagem em detrimento do desempenho do projeto.

2.2.2 LISA

Em [5] é reportado pela primeira vez a geração de HDL sintetizável a partir de modelos LISA. Novamente, os autores partem de dois modelos (um escrito em VHDL e outro escrito em LISA) de um processador (no caso, uma unidade de pós-processamento de transmissão de vídeo digital DVB-T) e comparam as características dos modelos finalizados. Os processos e metodologias escolhidos são muito parecidos com os apresentados previamente por EXPRESSION [1, 2] e pelas outras ADLs [22].

O fluxo de desenvolvimento, mostrado na Figura 2.4, permite o suporte à transição entre a fase de exploração e implementação no nível RTL. Unidades funcionais escritas independentemente do processo podem ser adicionadas ao projeto, por exemplo, um módulo

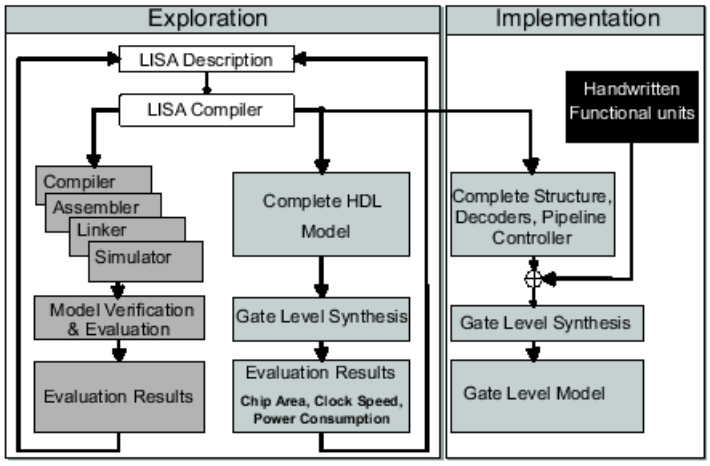


Figura 2.4: Fluxo de desenvolvimento RTL de LISA

escrito independentemente em outra linguagem.

O módulo escolhido nesse experimento para ser implementado foi chamado de ICORE, e é uma implementação de baixo consumo de potência de um ASIC utilizado para a aquisição e processamento de sinais DVB-T. Novamente vemos o padrão de uso da arquitetura RISC-DLX [20] descrito em [22] utilizada na síntese de ADLs como mostra a Figura 2.5: a típica sequência de estágios de busca, decodificação, execução, acesso à memória e escrita nos registradores é especializada para a implementação dos modelos LISA. Fica claro na Figura 2.5 quais são as partes que são automaticamente geradas e as partes que são descritas pelo usuários: a especificação da decodificação das instruções, as unidades de execução personalizadas, número de registradores e o *layout* da memória são descritos pelo usuário, enquanto a estrutura de comunicação entre os *pipelines*, a lógica de controle, a implementação do decodificador e o controle de entrada e saída são gerados automaticamente pela ferramenta.

As principais funções da arquitetura do ICORE incluem cálculos de FFT e interpolação. O modelo foi escolhido porque o grupo de pesquisadores já havia desenvolvido anteriormente um modelo em alto nível em LISA e uma implementação sintetizável em VHDL para o projeto, e eles queriam investigar e comparar o quanto uma versão automaticamente gerada se compararia à escrita em HDL. O modelo inicial escrito em VHDL rodava à 120Mhz como mostra a Tabela 2.5.

O modelo inicial escrito em VHDL e LISA do ICORE foi escrito por um projetista, e tomou aproximadamente três meses para ser desenvolvido. O mesmo projetista demorou aproximadamente um mês para descrever o módulo *cycle-accurate*. A partir do modelo *cycle-accurate*, o projetista demorou aproximadamente dois dias para tornar o modelo RTL sintetizável. Os resultados desse experimento da Tabela 2.5 mostram uma grande

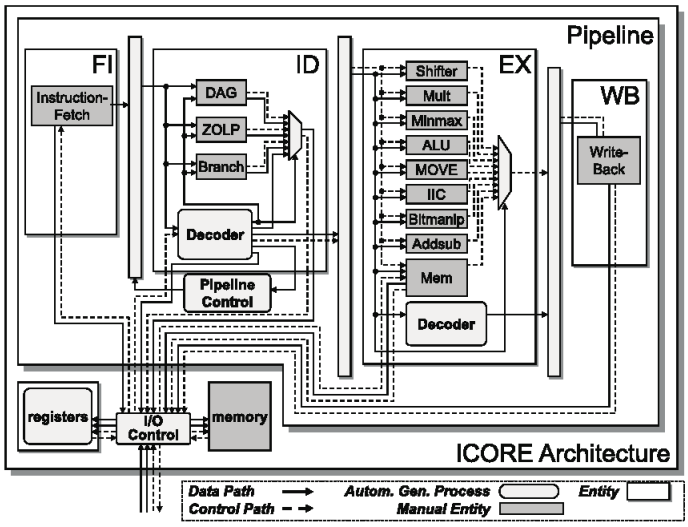


Figura 2.5: Arquitetura do ICORE sintetizado por LISA

Tabela 2.5: Resultados experimentais da síntese do ICORE usando LISA

	Tempo de desenvolvimento (meses)	Frequência (Mhz)	Área (gates)	Consumo de potência (mW)
VHDL	3	120	58473	12,64
LISA	1	125	59009	14,51

vantagem da metodologia proposta em relação ao tempo de desenvolvimento mas também, surpreendentemente, em relação à frequência de operação e área consumida. Ainda que o consumo de potência seja um pouco maior, os resultados em tempo de desenvolvimento, área consumida e frequência de operação apresentados são relevantes e não podem ser ignorados.

LISA [3, 4] têm três diferentes tipos de descrições de *hardware*: explícita, implícita e não formalizada. O *datapath* é sintetizado convertendo-se operandos e recursos de destinos em portas. Unidades funcionais escritas manualmente em SystemC podem também ser utilizadas para gerar os processadores LISA (LISA lida com os problemas de compartilhamento de recursos derivando operações de exclusão mútuas usando o grafo de operações [3]).

Em um trabalho relacionado [4], o autor reporta o desenvolvimento da arquitetura Leon e do ASMD DSP (*Infineon Technology*) em LISA. De forma similar às outras abordagens, um modelo escrito inteiramente em Verilog é comparado a um modelo escrito em LISA no que se refere à tempo de desenvolvimento, desempenho etc. As Tabelas 2.6 e 2.7 mostram os resultados do experimento. Conforme esperado, por um lado, os modelos descritos no nível de ADLs tem um desempenho menor e ocupam mais área, mas, por

Tabela 2.6: Resultados experimentais da síntese do Leon usando LISA

	Desempenho (ns)	Área (gates)
SystemC	3.08	16.7K
LISA	4.42	37K

Tabela 2.7: Comparação do tempo de desenvolvimento do ASMD utilizando LISA

	Tempo de desenvolvimento (meses)	Tamanho da especificação (linhas de código)
VHDL	5	1762
LISA	2	738

outro lado, são desenvolvidos em menos tempo. A metodologia proposta é considerada válida pelos autores porque o balanço entre esses dois fatores (desempenho e tempo de desenvolvimento) é mais equilibrado. Por exemplo, o tempo de desenvolvimento do ASMD cai de 5 para 2 meses se escrito em LISA.

2.2.3 AIDL

Em [10], é reportado o desenvolvimento de processadores em AIDL, cobrindo *pipelines* simples, *data forwarding* e execução fora de ordem. Os resultados foram obtidos implementando um processador com as 23 instruções da Tabela 2.8 que são baseadas no processador PA-RISC.

O método utilizado pelos autores em [10] para avaliar o fluxo de projeto mostrado na Figura 2.6 se baseia na comparação de três processadores descritos em uma versão em VHDL e outra em AIDL. Cada versão foi desenvolvida por projetistas separadamente (ambos com fluência nas linguagens). Os autores compararam o tempo de desenvolvimento de cada projetista (que inclui o projeto, implementação, simulação e depuração da arquitetura em um nível alto) como mostra a Tabela 2.9.

Infelizmente, no momento em que [10] foi publicado, a interface entre tradutor AIDL VHDL e o *hardware* escolhido não tinha sido acabada e os resultados da síntese não foram comparados. Ainda assim, os benefícios da metodologia em termos de tempo de desenvolvimento são claramente exemplificados na Tabela 2.9, com ganhos de produtividade no desenvolvimento das três diferentes arquiteturas escolhidas no experimento.

A arquitetura de *cache* de dados, *cache* de instruções não foram implementadas e os autores supõem que cada instrução pode ser recebida em um ciclo de *clock*.

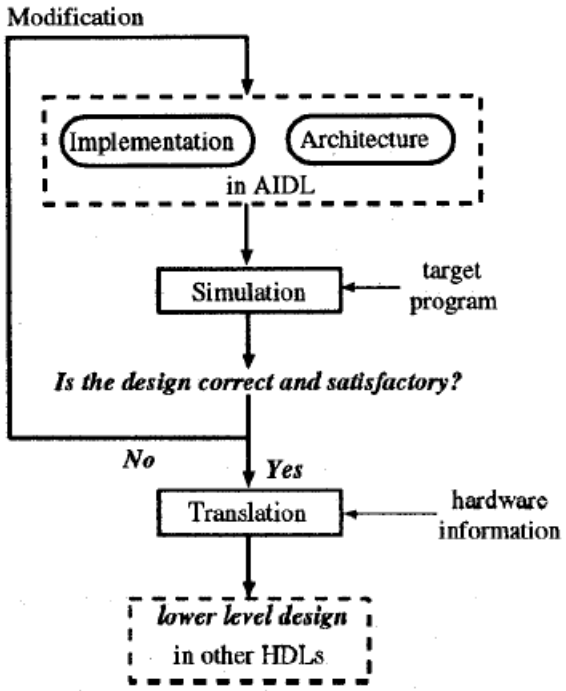


Figura 2.6: O fluxo de projeto de AIDL

Tabela 2.8: Instruções utilizadas no experimento com AIDL

Tipo de dado	Tipo da operação	Instrução
Inteiros	load e store	LDW, LDWX, LDWS, STW, STWS
	aritméticas	ADDIL, SH2ADD, OR, LDIL, LDO
	salto	COMBF, COMBT, COMBIBF, ADDIBT, ADDIBF
	outras	HALT
Ponto flutuante	load e store	FLDDS, FLDDX, FSTDS
	aritméticas	FADD, FCPY, FDIV, FMPY

Tabela 2.9: Tempo de desenvolvimento de processadores em AIDL sintetizáveis

	Pipeline básico (horas)	*Data forward* (horas)	*Out of order completion* (horas)
VHDL	83	24	49
AIDL	18	19	29

Figura 2.7: NML HMEs de memórias, quatro componentes diferentes de ALU, uma área de sequenciamento e contextos condicionais

2.2.4 NML

De forma geral, o método que NML [8] adota define quatro tipos básicos de entidades de modelos de *hardware* (HMEs) que se conectam via sinais: bancos de memória com largura definidas, unidades lógicas e aritméticas, arestas de sequenciamento (um sinal do tipo produtor consumidor) e contextos condicionais (multiplexadores, de-multiplexadores), que são exemplificados na Figura 2.7. A geração automática acontece em três fases: inicialmente, o arquivo NML é lido e uma representação interna da descrição é construída; a seguir, para cada regra op-/mode- é gerado um modelo de célula de *hardware* (HMC) hierarquicamente; por fim, um estágio de otimização é feito, refinando e reduzindo o número de HMEs necessários (por exemplo, agregando HMEs com comportamento similar). O algoritmo constrói uma representação RTL da árvore de sintaxe abstrata (AST) casando padrões de HMEs (por exemplo, construções *if* são implementadas como HMEs de contextos condicionais etc).

Em [8] é descrito um método de síntese de modelos NML, mas nenhum resultado experimental é reportado. O projeto Sim-HS é baseado em NML e gera modelos em Verilog sintetizáveis a partir de modelos Sim-NML como descrito em [23].

[24] reporta os resultados experimentais da síntese comportamental e estrutural de modelos Sim-NML. Para o experimento um subconjunto do processador PowerPC 603 foi sintetizado a partir de uma descrição comportamental de forma semi-automática (algumas partes, como alguns sinais de controle, são adicionados manualmente). A arquitetura alvo escolhida para a implementação do algoritmo especificado é a multiciclo. O autor descreve possíveis trabalhos futuros para explorar arquiteturas mais complexas e aponta que o objetivo do trabalho é mostrar a viabilidade da síntese de processadores especificados em Sim-NML.

2.2.5 ISDL: HGEN

ISDL [7] é uma ADL desenvolvida no departamento LSC/MIT e tem ferramentas que geram automaticamente simuladores de instruções (GENSIN) assim como uma implementação em *hardware* (HGEN) da descrição do processador.

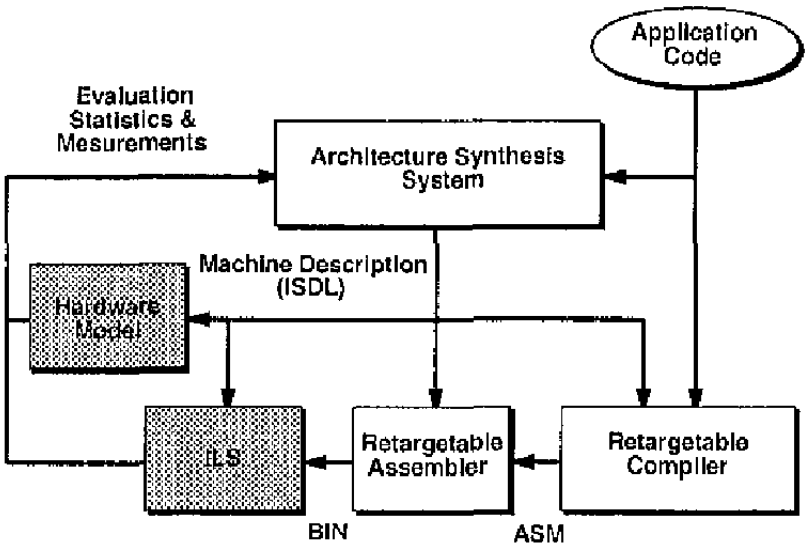


Figura 2.8: Fluxo de projeto do ISDL

ISDL é fortemente baseado em estados e transições de estados. Cada descrição de instrução contêm ações de operações e efeitos colaterais, que são uma série de comandos RTL que descrevem o efeito de cada operação no estado do processador. Cada operação também contém meta informações como custo, temporização e restrições que ajudam o ISDL a gerar um *hardware* eficiente.

ISDL resolve o problema de Compartilhamento de Recursos, que é uma otimização que permite que códigos semelhantes sejam implementados com recursos compartilhados. Para resolver esse problema, HGEN quebra cada operação RTL em recursos que são utilizados. Os recursos utilizados por todas as instruções que são equivalentes e que não são utilizados ao mesmo tempo são agrupados. O HGEN implementa esse algoritmo produzindo uma matriz de recursos n *versus* n, com os seguintes atributos:

A_{ij} é igual a 1 se:

1. o nó i e o nó j não estão sendo utilizados na mesma operação.

2. o nó i e o nó j são funcionalmente equivalentes.

3. o nó i e o nó j são operações no mesmo parâmetro.

e 0 caso contrário.

Cada entrada da matriz diz se o recurso i pode ser compartilhado com o recurso j. ISDL cria o clique máximo dos nós que podem ser compartilhados. O clique máximo é sintetizado como um circuito e a interface entre esses recursos é automaticamente gerada.

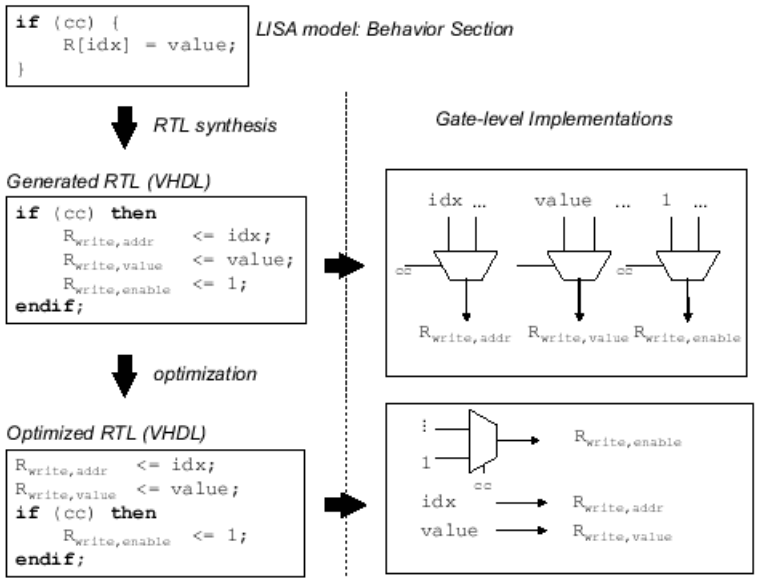


Figura 2.9: Otimizações comportamentais feitas em um modelo LISA

2.3 Técnicas de otimização

Nenhum dos trabalhos iniciais das ADLs [1, 2, 3, 4, 7, 8, 10] reporta explicitamente otimizações feitas na síntese da linguagem: os trabalhos estão basicamente preocupados em tornar a transição possível. Em [17] são descritas otimizações feitas no nível estrutural e no nível comportamental de ADLs.

Otimizações estruturais fazem proveito da exclusividade explícita entre diferentes partes de uma única instrução, assim como a exclusividade entre difererentes instruções do processador. Otimizações comportamentais removem o custo herdado pelo nível de abstração mais alto (semelhante à como um compilador faz o desenrolamento de constantes e remoção de código não utilizado ao compilar uma linguagem de alto nível em código de máquina).

A análise de exclusividade é feita em duas partes: informações explícitas (como a codificação das instruções binárias, códigos condicionais) e implícitas (dependências em tempo de execução).

O grafo de conflitos estruturais é utilizado, por exemplo, para otimizar o tamanho de multiplexadores que acessam os elementos globais, transformando multiplexadores baseados em decodificação em multiplexadores compartilhados [17].

Otimizações comportamentais acontecem simplificando blocos *if/else* com sinais de ativação explícitas como pode ser visto na Figura 2.9.

Os autores mostram que lidar com esses problemas em níveis de abstração mais altos

Tabela 2.10: HDL utilizada para a descrição do modelo RTL.

EXPRESSION	Lisa	AIDL	NML	Mimola	UDL	ArchC
VHDL	SystemC	VHDL	Verilog	Mimola	UDL	SystemC

gera resultados melhores do que em níveis mais baixo.

2.4 Conclusão

De forma geral, a maioria das ADLs tem um suporte relativamente limitado no que diz respeito à síntese. Industrialmente e academicamente, as ADLs são na maioria das vezes utilizadas em níveis altos de especificação e o uso de ADLs para síntese ainda é um problema recente e ativamente sendo estudado. O principal motivo pelo qual ADLs não são utilizadas para a fabricação do processador final se deve ao fato das limitações (como, por exemplo, consumo de área e frequência de operação) de performance impostas pela geração automática. Como descrito na seção anterior, algumas dessas limitações estão sendo tratadas por diferentes autores.

Dito isso, EXPRESSION e LISA são as linguagens que oferecem as melhores coberturas no que diz respeito à síntese. EXPRESSION possui processadores RTL do tipo monociclo, multiciclo, pipeline, superescalares, VLIW e com *pipelines* de múltiplos estágios, projetados de 3 à 7 vezes mais rápidos do que as alternativas manuais (HDL) e com desempenhos razoáveis (20%-30% de perda em frequência de operação e área). LISA, por sua vez, possui processadores RTL do tipo monociclo, multiciclo, *pipeline* e DSPs, projetados 2.5 à 3 vezes mais rápidos e com desempenhos comparáveis à EXPRESSION (15%-40% de perda em frequência e área).

ArchC não possui nenhum suporte à síntese atualmente. O objetivo desse trabalho é projetar e implementar as especificações da linguagem e ferramentas necessárias para tornar a síntese de modelos de processadores descritos em ArchC possível. A Tabela 2.10 mostra as linguagens HDL escolhidas para a síntese de cada ADL. ArchC utiliza SystemC para representar seus modelos RTL, como será apresentado nas seções que se seguem.

Capítulo 3

ArchC RTL

Nesse capítulo serão descritos os principais conceitos utilizados nesse trabalho.

A linguagem ArchC será introduzida assim como as ferramentas que foram construídas em torno da linguagem e o fluxo típico de desenvolvimento.

A seguir será mostrado como a síntese dos processadores se insere nesse fluxo de desenvolvimento e as ferramentas e conceitos envolvidos no processo. As principais ideias e técnicas utilizadas na síntese das principais construções da linguagem (`AC_ARCH` e `AC_ISA`) serão explicadas e a representação RTL das construções da linguagem serão ilustradas.

As implicações das técnicas propostas serão explicadas e os impactos nas construções que podem ser utilizadas no nível RTL serão apresentadas. O subconjunto sintetizável da linguagem ArchC será definido e a relação desse subconjunto com a descrição ArchC tradicional será apresentada.

Por fim, a metodologia utilizada na verificação dessas ideias é descrita. Os resultados obtidos por essa metodologia são mostrados e analisados no próximo capítulo.

3.1 Introdução à ArchC

ArchC [26, 27] é uma linguagem de descrição de arquiteturas desenvolvida na Universidade Estadual de Campinas e mantida pelo Laboratório de Sistemas de Computacão da mesma universidade. ArchC é uma ADL baseada em SystemC, uma linguagem de descrição de *hardware*. ArchC provê a especificação da linguagem [26, 27] e ferramentas de desenvolvimento como simuladores do modelo comportamental [26, 33], co-simuladores [31], montadores [32], interface com sistemas operacionais [29], suporte à diferentes hierarquias de memória [28] e ferramentas de análise de consumo de potência [30]. Além disso, outros trabalhos [35, 36] reportam o uso da linguagem para o desenvolvimento de processadores.

Existem diversos modelos de processadores descritos na linguagem como, por exemplo,

```
1 AC_ISA(mips){
2   ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
3   ac_format Type_I = "%op:6 %rs:5 %rt:5 %imm:16:s";
4   ac_format Type_J = "%op:6 %addr:26";
5
6   ac_instr<Type_R> add, sub, instr_and, instr_or, mult, div;
7   ac_instr<Type_R> mfhi, mflo, slt, jr;
8   ac_instr<Type_R > addu, subu, multu, divu, sltu;
9   ac_instr<Type_I> load, store, beq, bne;
10  ac_instr<Type_J> j, jal;
11
12  ISA_CTOR(mips){
13    load.set_asm("lw %rt, %imm(%rs)");
14    load.set_decoder(op=0x23);
15
16    store.set_asm("sw %rt, %imm(%rs)");
17    store.set_decoder(op=0x2B);
18    ...
19  };
20 };
```

Figura 3.1: Declaração do conjunto de instruções do MIPS em `AC_ISA`.

PowerPC, ARM, MIPS-I, SPARC-V8, Intel 8051 e PIC 16F84. A linguagem também é utilizada para o ensino de Arquiteturas de Computadores e diversos modelos de processadores foram gerados a partir desse trabalho (como, por exemplo, JVMs) [34] .

A descrição estrutural da arquitetura é composta por duas partes: a definição do conjunto de instruções (`AC_ISA`) e a descrição dos elementos estruturais da arquitetura (`AC_ARCH`).

Na primeira, o projetista declara quais são as instruções disponíveis, o formato binário e diretivas de compilação (que auxiliam na geração das ferramentas, como montadores) e decodificação (que definem como as instruções serão decodificadas). Na Figura 3.1, por exemplo, a descrição do *instruction set architecture* do MIPS mostra a declaração dos formatos das instruções (`R`, `I`, `J` nas linhas 2 à 4), a declaração de cada instrução associada à um formato (`add, sub, j, jal, load, store` etc nas linhas 6 à 10), a declaração de como cada instrução é decodificada (`store.set_decoder(op=0x2b)` na linha 14) e a declaração de como os montadores serão gerados (`load.set_asm('lw %rt, %imm(%rs)')` na linha 13).

Na segunda parte, o projetista declara os recursos estruturais da arquitetura como, por exemplo, o tamanho da palavra, a quantidade de memória disponível, a hierarquia de *caches*, os estágios do *pipeline*, o banco de registradores etc. Na Figura 3.2, por exemplo, as informações estruturais do MIPS são declaradas: cinco estágios de *pipeline* (linha 6), palavras de largura de 32 bits (linha 2), registradores entre os estágios do *pipeline* (linhas

```
1 AC_ARCH(mips){
2   ac_wordsize 32;
3   ac_mem MEM:256K;
4   ac_regbank BANK:34;
5
6   ac_pipe pipe = {IF, ID, EX, MEM, WB};
7   ac_stage IF, ID, EX, MEM, WB;
8
9   ac_format Fmt_IF_ID = "%npc:32";
10  ac_format Fmt_ID_EX = "%npc:32 %data1:32 %data2:32 %imm:32:s
11      %rs:5 %rt:5 %rd:6 %regwrite:1 %memread:1 %memwrite:1";
12  ac_format Fmt_EX_MEM = "%alures:32 %wdata:32 %rdest:5 %regwrite:1
13      %memread:1 %memwrite:1";
14  ac_format Fmt_MEM_WB = "%wdata:32 %rdest:5 %regwrite:1";
15
16  ac_reg<Fmt_IF_ID> IF_ID;
17  ac_reg<Fmt_ID_EX> ID_EX;
18  ac_reg<Fmt_EX_MEM> EX_MEM;
19  ac_reg<Fmt_MEM_WB> MEM_WB;
20
21  ARCH_CTOR(mips) {
22    ac_isa("mips_isa.ac");
23  };
24 };
```

Figura 3.2: Declaração dos recursos do *pipeline* do MIPS em `AC_ARCH`.

16 à 19), 34 registradores (linha 4) e 256K de memória (linha 3).

Essas duas partes são combinadas e transformadas em um simulador executável pelas ferramentas da linguagem. O fluxo do gerador do simulador executável, mostrado na Figura 3.3, parte da descrição em `AC_ISA` e `AC_ARCH`, gera um modelo em SystemC intermediário que é, por fim, compilado no modelo executável.

Até esse estágio o simulador da arquitetura contém as informações estruturais e já é capaz de receber como entrada programas binários compilados para a arquitetura sendo desenvolvida. No próximo passo, o projetista descreve o comportamento de cada instrução, que implementa cada comando com os recursos estruturais disponíveis.

A Figura 3.4 mostra a declaração de como o comportamento da instrução *add* se relaciona com os estágios do *pipeline*, os registradores disponíveis e a interface da memória externa. Para cada estágio do *pipeline* declarado, o comportamento da instrução *add* é definido.

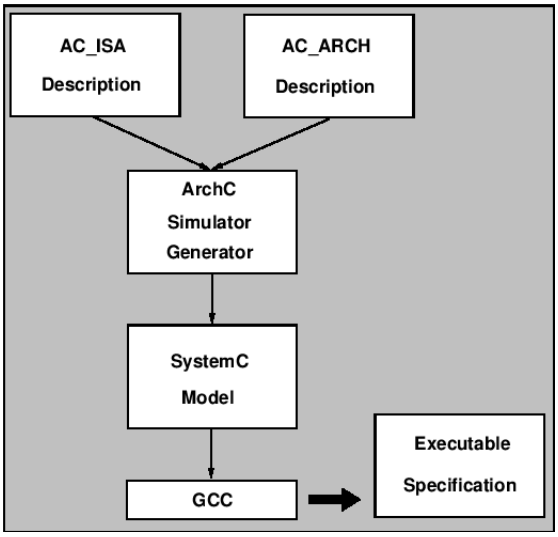


Figura 3.3: Fluxo de geração dos simuladores de ArchC.

```
1  void ac_behavior(add, stage){
2    switch (stage) {
3      case IF:
4        IF_ID.npc = ac_pc + 4;
5        break;
6      case ID:
7        ...
8        break;
9      case EX:
10       EX_MEM.alu_result = ID_EX.rs + ID_EX.rt;
11       break;
12     case MEM:
13       MEM_WB.alu_result = EX_MEM.alu_result;
14       MEM_WB.rd = EX_MEM.rd;
15       break;
16     CASE WB:
17       RB.write(MEM_WB.rd, MEM_WB.alu_result);
18       break;
19     default:
20       break;
21   }
22 };
```

Figura 3.4: Descrição do comportamento da instrução *add* para cada estágio do *pipeline*.

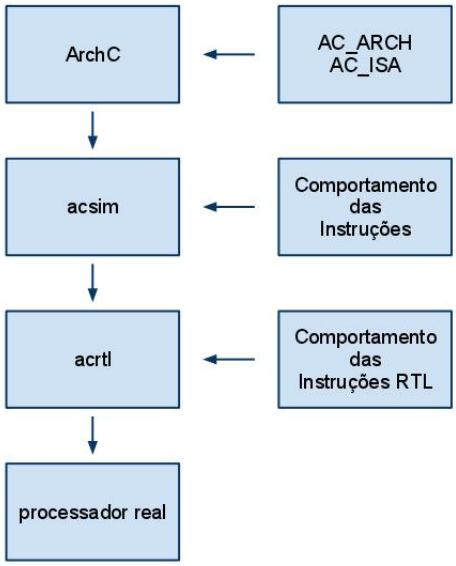


Figura 3.5: Fluxo de desenvolvimento RTL.

3.2 ArchC RTL

O fluxo descrito na seção anterior e exemplificado na Figura 3.3 gera simuladores executáveis da descrição da arquitetura sendo desenvolvida. De forma geral, o problema que esse trabalho se propõem a resolver é definir um método que gere um circuito que seja suficiente para implementar a descrição dessa arquitetura. Vale notar que, como nas abordagens descritas anteriormente [1, 2, 3, 4, 7, 8, 10], o objetivo desse trabalho é tornar a síntese dos processadores possível, gerando algum subconjunto suficiente que implemente a arquitetura e não o subconjunto mais eficiente. Otimizações serão discutidas adiante como trabalhos futuros.

Nesse contexto, ArchC RTL é a definição do subconjunto sintetizável da linguagem ArchC e o conjunto de algoritmos e ferramentas necessários para o suporte à síntese dos processadores nesse nível.

Para tornar isso possível, ArchC RTL adiciona mais uma etapa no fluxo de projeto do ArchC descrito anteriormente na Figura 3.3. Como mostra a Figura 3.5, ArchC RTL toma como entrada um modelo .ac válido e retorna como saída um modelo RTL sintetizável.

De forma análoga à como a ferramenta `acsim` gera um simulador comportamental em SystemC na Figura 3.3, a ferramenta que será descrita nesse trabalho gera um modelo RTL em Verilog que pode ser simulado e sintetizado como mostra a Figura 3.6. Com efeito, por um lado, `acsim` toma como entrada a linguagem ArchC e gera um simulador executável a partir da transformação da entrada em SystemC. Por outro lado, `acrtl` toma

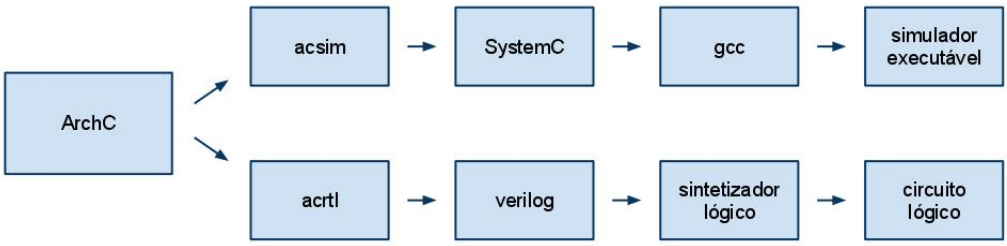


Figura 3.6: Comparação do contexto de trabalho de cada ferramenta.

como entrada a mesma especificação ArchC e gera uma descrição sintetizável, através da síntese da descrição em Verilog que é então sintetizado em um circuito digital.

Essa comparação é importante porque mostra o escopo do trabalho que está sendo descrito. ArchC RTL é uma especificação compatível com o último nível do fluxo da Figura 3.5, mas o inverso não é verdade (a especificação do nível anterior não é necessariamente compatível com ArchC RTL). Nesse nível RTL, não estamos tentando ajudar o projetista a encontrar quais são os melhores formatos de instrução ou quais são as instruções que devem estar disponíveis: assume-se que essas análises já foram feitas em etapas anteriores em níveis mais alto de abstração. Nesse nível, o projetista está interessado em tornar as decisões que foram feitas previamente em um processador real, concreto.

A abordagem desse trabalho para o problema descrito é muito similar à abordagem adotada pelas outras ADLs [1, 2, 3, 4, 7, 8, 10], ilustrada de forma generalizada na Figura 2.2. No Capítulo 5 serão mostradas as principais diferenças entre as abordagens, assim como a comparação dos resultados.

No caso de ArchC, como ilustra a Figura 3.6, uma ferramenta - chamada `acrtl` - cria modelos RTL em SystemC que combinam as definições estruturais de `AC_ARCH` e os pontos de entrada das instruções descritas em `AC_ISA`, a partir da especificação da arquitetura. A seguir, o modelo sintetizável é composto com a descrição do comportamento das instruções feita pelo usuário. Essa descrição é então traduzida para uma linguagem HDL (no caso, Verilog) e sintetizada utilizando um sintetizador de HDL. Por fim, o resultado da síntese é então prototipado e verificado em circuitos reconfiguráveis (como por exemplo *FPGAs*) ou implementado em *ASICs*.

O processo da geração do modelo sintetizável ilustrado na Figura 3.7 pode então ser entendido por essas duas fases ortogonais: a geração e síntese da definição estrutural da arquitetura a partir da `AC_ARCH` (Figura 3.2) e `AC_ISA` (Figura 3.1), e a integração e síntese da definição comportamental das instruções a partir da implementação SystemC/C++ (Figura 3.4). Os detalhes desses dois processos serão explicados nas seções seguintes.

O método apresentado também se baseia na arquitetura proposta por [20] utilizada em [1, 3, 7]. Da mesma forma que [10] escolhe o conjunto representativo de instruções

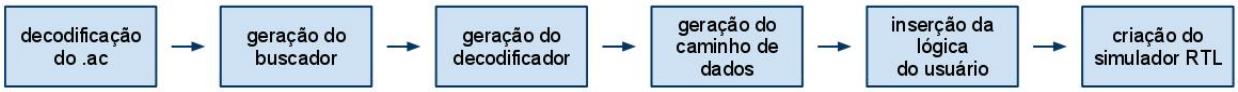


Figura 3.7: Geração dos processadores em RTL a partir dos arquivos .ac

da Tabela 2.8, esse trabalho também escolherá apenas algumas instruções para serem implementadas que serão discutidas no próximo capítulo. Assim como [4], `acrtl` também suporta a instanciação de módulos escritos pelo usuário a partir do suporte à diretiva `ac_helper` definida na linguagem.

As ADLs variam no que diz respeito a qual HDL é utilizada para representar o modelo RTL. Por exemplo, enquanto EXPRESSION e AIDL utilizam VHDL [1, 10], Lisa utiliza SystemC [3] e NML Verilog [8]. Como MIMOLA [6] e UDL são originalmente HDLs, as próprias linguagens são utilizadas para a síntese. A Tabela 2.10 faz a relação entre as ADLs e as HDLs escolhidas para a síntese. ArchC escolheu implementar seus modelos em SystemC por ArchC já ser uma linguagem fortemente baseada em SystemC.

3.3 Síntese dos recursos estruturais

A ferramenta `acrtl` depende de duas informações dadas pelo usuário: informações estruturais e comportamentais. A síntese da estrutura da arquitetura se refere ao problema de gerar automaticamente modelos RTL sintetizáveis a partir da entrada das informações estruturais, como mostra o exemplo da Figura 3.1 e da Figura 3.2. A lógica estrutural é recebida como entrada na estrutura `AC_ARCH` e `AC_ISA` e gera a arquitetura estrutural do processador.

Nessa etapa, como ilustra a Figura 3.7, são gerados o decodificador de instruções, a interface com a memória, o banco de registradores, os estágios do *pipeline*, os multiplexadores para a implementação do comportamento das instruções, a lógica de controle etc.

Os principais macromódulos base dos processadores ArchC especializados para cada arquitetura são:

1. *Fetcher*: responsável pela busca das instruções da memória.
2. Decodificador: decodifica as instruções recebidas pelo *fetcher*.
3. Controlador: lógica de controle entre os macromódulos, multiplexadores etc. Despacha as instruções decodificadas pelo *decoder* para a implementação pertinente.

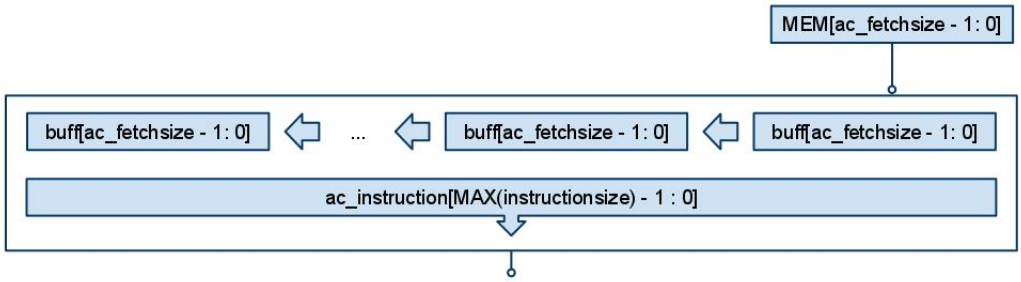


Figura 3.8: A generalização do módulo ac_fetcher responsável pela busca das instruções da memória.

4. Unidade de execução: fornece a multiplexação entre as instruções buscadas e a implementação da execução.

5. Lógica do usuário: comportamento definido pelo usuário que geram os novos valores para cada registrador do sistema.

6. Registradores do Sistema: elemento de armazenamento do sistema como, por exemplo, registradores entre estágios do *pipeline*, registradores do ponteiro de instruções etc.

7. Registradores do usuário: conjunto de elementos de armazenamento definidos pelo usuário.

8. Unidades externas: memória externa, *caches*, pinos de interrupção, portas extras etc.

Cada macromódulo é gerado a partir das informações contidas na descrição do `AC_ARCH` ou do `AC_ISA`.

A implementação de cada macromódulo é escolhida pelo projetista. `acrtl` prove uma biblioteca de estruturas generalizadas básica que oferece uma implementação padrão para cada um dos macromódulos existentes em ArchC. Cada implementação das estruturas generalizadas impacta diretamente como a descrição ArchC é feita: desde as portas de interface de controle da implementação até as características de performance (sincronização, área, e temporização). Isso possibilita, por exemplo, o projetista escrever o seu próprio `ac_fetcher` ou `ac_decoder` se for do seu interesse.

O *Fetcher*, ilustrado na Figura 3.8, é gerado a partir de um fetcher genérico com relação à largura máxima das instruções, a largura do barramento de memória - definidos explicitamente em `AC_ARCH` utilizando a declaração de `ac_fetchsize` - e o tipo de representação das palavras - little ou big endian. As palavras são buscadas da memória e são inseridas em um registrador do tipo 'shift'. Esse processo se repete até a instrução inteira

Tabela 3.1: Tabela verdade utilizada para a decodificação das instruções. DC são estados *don't care*, onde o valor do campo não interfere no resultado.

formato_1_campo_1	formato_1_campo_2	...	formato_n_campo_m	ID
0xCA	DC	DC	DC	INSTR_1
...	...	0xFE	...	...
DC	DC	DC	0xBA	INSTR_n

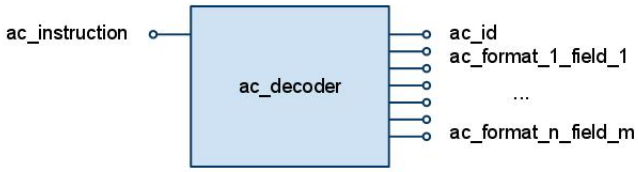


Figura 3.9: O decodificador de instruções ac_decoder generalizado.

ter sido trazida da memória (estaticamente definido para processadores com instruções de tamanho definido ou dinamicamente para processadores com instruções de tamanho variável).

O decodificador, ilustrado na Figura 3.9, é um circuito combinacional criado a partir da síntese de uma tabela verdade implementada com um circuito construído a partir de mapas de Karnaugh. A tabela verdade da decodificação, mostrada de forma generalizada na Tabela 3.1, é gerada a partir das informações contidas em `set_decoder` e `ac_instr` declaradas em `AC_ISA`. Para cada instrução, uma linha é adiciona à tabela. Em cada linha, para cada critério da decodificação, uma coluna é adiciona.

O decodificador e multiplexador, ilustrado na Figura 3.10, do sistema é responsável por fornecer o acesso aos dados necessários para os circuitos definidos pelo usuário e escolher quais desses circuitos são utilizados. Juntos eles controlam como funciona a interface entre a lógica estrutural automaticamente gerada e a lógica comportamental definida pelo usuário.

Tome como exemplo a especificação do comportamento das instruções de adição e subtração definidos pelo usuário na Figura 3.11. Em conjunto com as informações estruturais dos arquivos .ac da Figura 3.1 e da Figura 3.2, a descrição das instruções tem de ser automaticamente integrada com a lógica estrutural. O somador da instrução `add` tem acesso ao banco de registradores e aos parâmetros `rs`, `rt` e `rd` que são decodificados. Analogamente, a instrução `sub` também tem acesso à esses registradores. O resultado da execução, no entanto, depende de qual instrução está sendo executada e é escolhido com multiplexadores à partir da decodificação da instrução mostrada anteriormente.

Todos os elementos externos à caixa pontilhada da Figura 3.10 são gerados automaticamente a partir da instanciação de módulos previamente descritos, assim como os

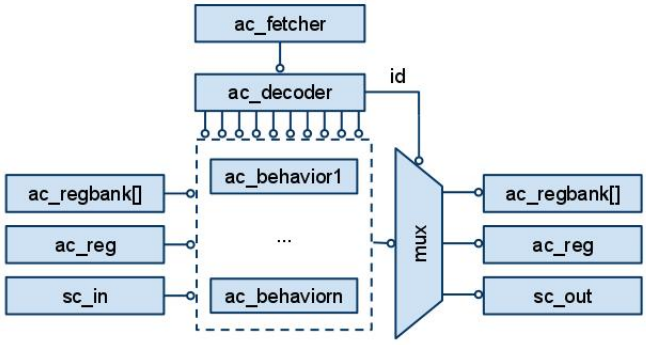


Figura 3.10: O *datapath* generalizado dos processadores e como os componentes ac_fetcher e ac_decoder se relacionam com a lógica do usuário.

```
1 ac_behavior(add) {
2   registers[rd] = registers[rs] + registers[rt];
3 }
4 ac_behavior(sub) {
5   registers[rd] = registers[rs] - registers[rt];
6 }
```

Figura 3.11: Descrição do comportamento das instruções de adição e subtração.

elementos de armazenagem definidos pelo usuário. Vale notar que todos esses módulos fornecidos pela ferramenta são RTL-sintetizáveis.

Os diferentes *datapaths* são criados a partir da arquitetura escolhida pelo projetista do modelo descrito em `AC_ARCH`. As Figuras 3.12, 3.14 e 3.15 ilustram como os macromódulos se relacionam com o circuito gerado pelo comportamento das instruções definido pelo projetista para arquiteturas monociclo, multiciclo e *pipeline* respectivamente.

Nessas figuras, as caixas retangulares são os macromódulos estruturais gerados automaticamente pela ferramenta a partir do conteúdo das descrições em `AC_ISA` e `AC_ARCH` e as nuvens são os circuitos gerados a partir da descrição do comportamento das instruções feita pelo projetista.

A arquitetura RTL é escolhida de forma consistente com a comportamental: todas as restrições e implicações de projeto no simulador comportamental são compatíveis com o modelo RTL. Por exemplo, o *datapath* da Figura 3.12 mostra a arquitetura monociclo, que impõem que todas as instruções executem em um único ciclo. Essa restrição é imposta tanto pelo simulador comportamental quanto pelo modelo RTL. Outra restrição tipicamente citada em modelos *pipeline* é o fato dos sinais terem apenas um único *driver*: cada *flip flop* pode ter apenas um sinal de entrada. Dessa forma, em modelos *pipeline* como o da Figura 3.15, cada elemento de armazenamento pode apenas ser escrito por um

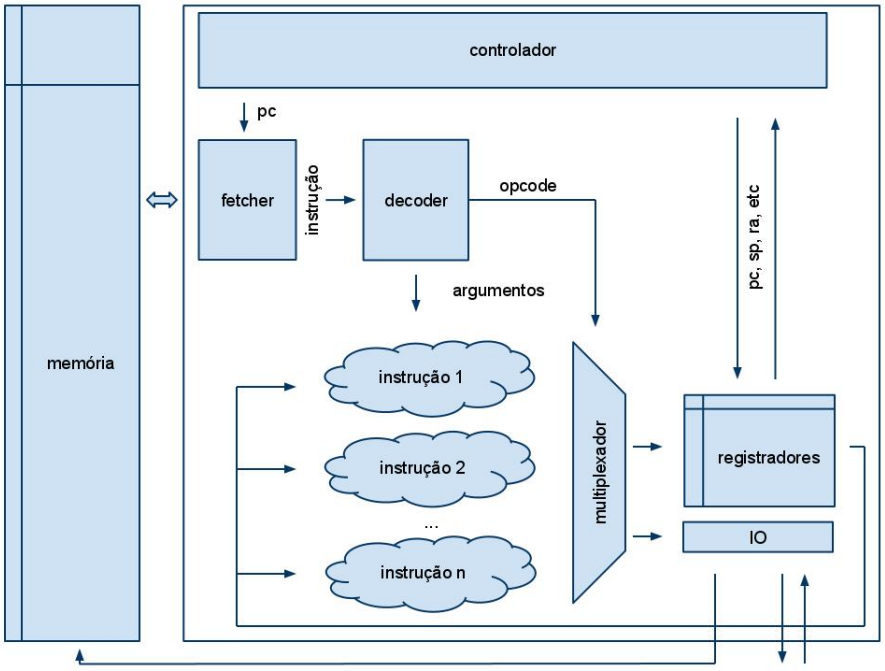


Figura 3.12: *Datapath* da arquitetura monociclo.

único estágio do *pipeline*.

A síntese dos processadores é feita integrando as informações estruturais e as descrições do comportamento das instruções. Considere, por exemplo, como as arquiteturas monociclos são sintetizadas: o circuito de cada instrução é inserido no *datapath* (as nuvens na Figura 3.12), recebe como entrada o estado atual do processador (registradores), os argumentos das instruções e o controlador escolhe (através dos multiplexadores) o resultado de qual circuito é utilizado em função da instrução que está sendo executada. As arquiteturas monociclo são as mais simples de serem geradas, pois o circuito que implementa a descrição do comportamento da instrução pode ser diretamente integrado na arquitetura estrutural. Esse conceito é ilustrado na Figura 3.12.

Considere como o circuito das instruções de adição e subtração mostrados na Figura 3.11 são introduzidos na arquitetura monociclo. O somador e o subtrador da Figura 3.13 representam o circuito que implementa o comportamento dessas instruções. O multiplexador da mesma figura mostra como o controlador (ilustrado na Figura 3.12) do processador decide qual resultado utilizar a partir do `opcode` da instrução sendo executada e para qual registrador guardar o resultado a partir dos operandos da instrução (`rd`). Tanto o multiplexador, o controlador, o decodificador (que fornece o `opcode` e os índices `rs`, `rt` e `rd`) como banco de registradores são gerados automaticamente através das informações estruturais. O somador e o subtrador são gerados pelo código do projetista e são inseridos

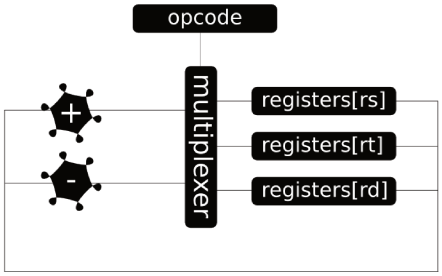


Figura 3.13: Seleção da entrada pelos multiplexadores.

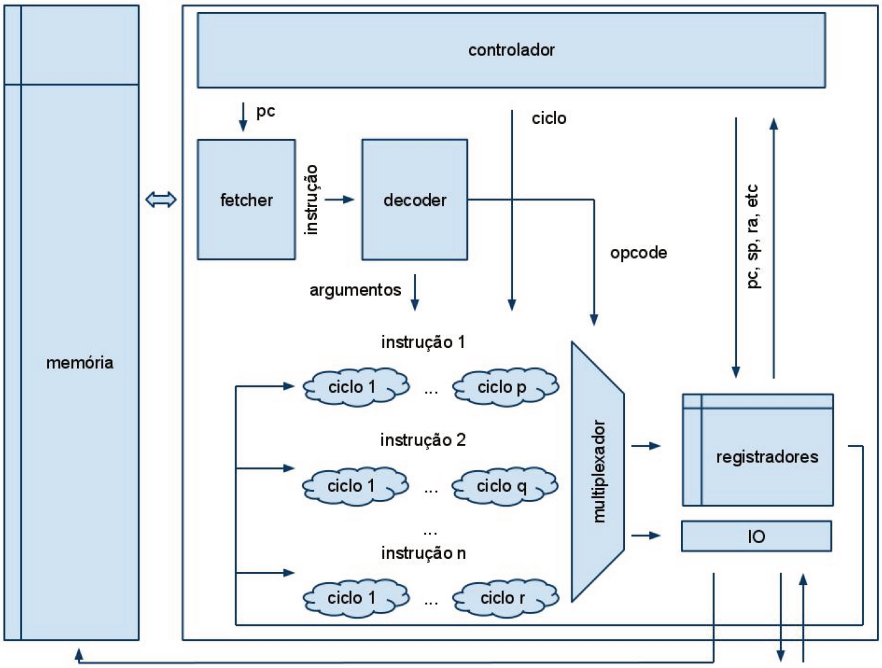


Figura 3.14: *Datapath* da arquitetura multiciclo.

nas nuvens da Figura 3.12.

As arquiteturas multiciclos são um pouco mais complicadas de serem sintetizadas, mas funcionam de forma análoga ao método de síntese das arquiteturas monociclo. A arquitetura estrutural é gerada a partir das informações estruturais e são criados espaços reservados para as instruções serem inseridas na máquina de estados do processador, que são representados pelas nuvens na Figura 3.14. No entanto, dessa vez, cada espaço reservado da máquina é responsável por executar apenas um dos ciclos da implementação da instrução. O circuito que implementa a instrução é então quebrado na lógica dos diferentes ciclos, e apenas o sub-circuito que implementa determinada instrução em determinado ciclo é inserido em cada espaço. Essa implementação, ainda que mais complicada, consegue atingir níveis de frequência de operação mais rápidos.

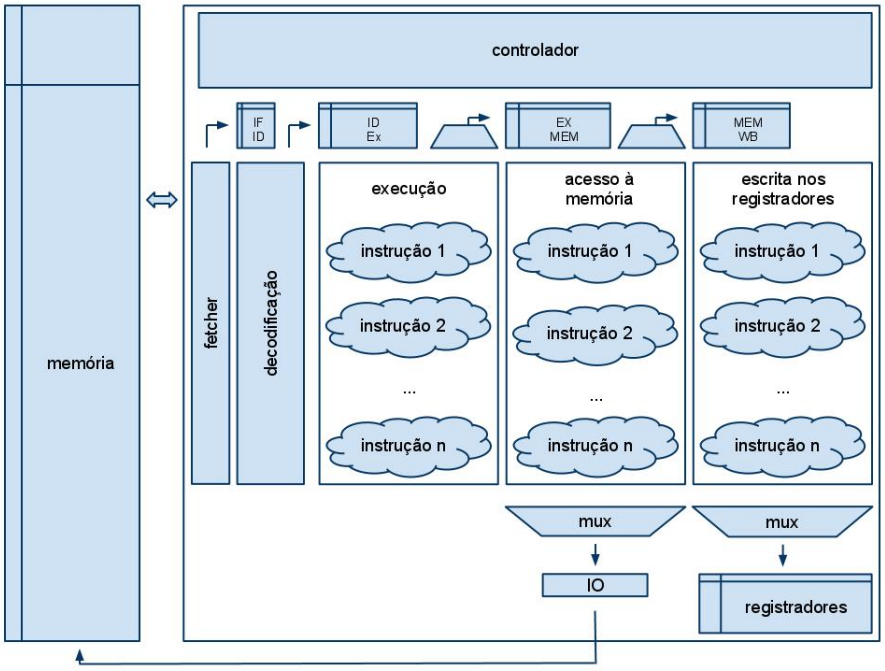


Figura 3.15: *Datapath* da arquitetura *pipeline*.

Por fim, considere como arquiteturas *pipeline* são produzidas. Novamente, a arquitetura estrutural é gerada e as instruções são inseridas no *datapath*. O circuito lógico das instruções é quebrado em estágios e executados independentemente em paralelo. Na Figura 3.15, por exemplo, apesar do comportamento das instruções serem descritos como um só circuito, é necessário quebrá-los em estágios ortogonais e utilizá-los independentemente para adquirir o caráter paralelo da arquitetura *pipeline*.

Nessa seção foi explicado como os recursos estruturais são entendidos no nível RTL. A seção que se segue explica com mais detalhes como os circuitos gerados a partir das descrições das instruções são quebrados e inseridos na arquitetura estrutural.

3.4 Síntese do comportamento das instruções

A segunda categoria de informações entradas pelo usuário é a descrição do comportamento das instruções. É nessa parte que o projetista, a partir dos recursos estruturais definidos, parte para implementar a execução de cada instrução. Durante a síntese, o comportamento das instruções é inserido na estrutura criada na etapa anterior. Vale notar que a arquitetura estrutural construída na primeira fase utiliza apenas código RTL, e implica que o usuário também é forçado a escrever código RTL na descrição do comportamento das instruções. Isso é feito provendo um conjunto de regras e guias de desenvolvimento,

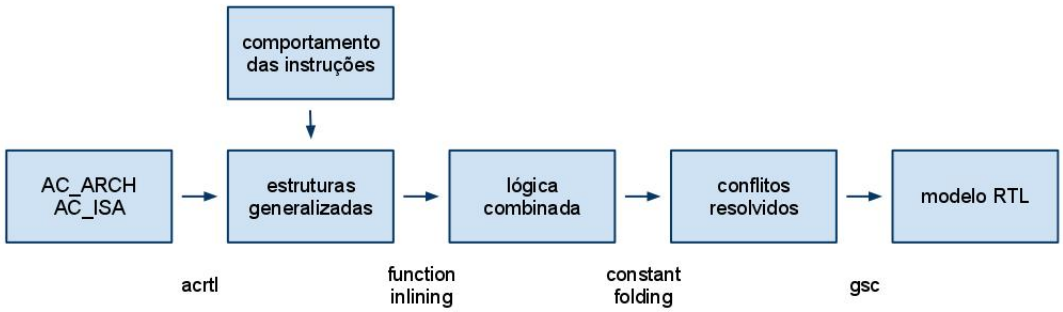


Figura 3.16: Uso da técnica de *inline* de funções e desenrolamento de constantes.

similar ao manual [38], cujos conceitos gerais serão discutidos no capítulo 5. A versão completa do guia de desenvolvimento pode ser encontrada na distribuição do manual da linguagem ArchC [25]. O uso de código comportamental é discutido mais adiante no Capítulo 5 como possíveis trabalhos futuros.

Nessa segunda fase, a ferramenta utiliza três técnicas de otimização usadas em compiladores para combinar a informação comportamental descrita pelo usuário com a descrição estrutural gerada automaticamente: *function inlining*, *constant folding* e *dead code removal* como mostra a Figura 3.16.

Function inlining faz basicamente o trabalho de combinar o código, combinando a entrada do usuário com os recursos estruturais gerados anteriormente, como mostra a Figura 3.17. A seguir, *constant folding* em conjunto com *dead code removal* resolvem conflitos, como garantir que os sinais tenham apenas um único driver quando múltiplos processos são utilizado em modelos *pipeline* como mostra a Figura 3.18.

As Figuras 3.12, 3.14 e 3.15 ilustram como o circuito gerado pela descrição do comportamento das instruções é combinado com os recursos estruturais gerados automaticamente para respectivamente monociclos, multiciclos e *pipeline*s, utilizando as técnicas descritas anteriormente.

A estrutura do monociclo ilustrada na Figura 3.12 exemplifica bem como a técnica de *Function inlining* é utilizada. A unidade estrutural de execução gerada anteriormente contém pontos de entrada para a descrição do comportamento das instruções. A lógica de controle, em conjunto com as informações do decodificador de instruções, identifica qual instrução deve ser executada nesse ciclo e delega a implementação da execução para o circuito gerado por `ac_behavior(add)`. A Figura 3.17 ilustra como a técnica atua na injeção do circuito definido pelo usuário na arquitetura estrutural: as linhas 1-10 da coluna da esquerda da figura ilustram o comportamento definido pelo usuário e as linhas 11-21 ilustram a maquina de estados do controlador de execução; a coluna da direita da figura ilustra como o comportamento descrito pelo usuário é combinado com a estrutura

```
1 void add(int cycle){
2   switch(cycle)
3   case 0 :
4     result = regs[rs] + regs[
          rd];
5     break;
6   case 1 :
7     regs[rd] = result;
8     break;
9   }
10 }
11 void fsm(){
12   if(resetn){
13   } else {
14     switch(opcode) {
15     case ADD :
16       add(cycle);
17       cycle++;
18       break;
19     }
20   }
21 }
```

```
1 void fsm(){
2   if(resetn){
3   } else {
4     switch(opcode){
5     case ADD :
6       switch(cycle): {
7       case 0 :
8         result = regs[rs] +
              regs[rd];
9         break;
10      case 1 :
11        regs[rd] = result;
              break;
12        break;
13      }
14      cycle++;
15      }
16    }
17  }
18 }
```

Figura 3.17: *Inline* de funções: o método `add` da coluna da esquerda é inserido no método `fsm` mostrado na coluna da direita.

```
1 void add(int stage){
2   switch(stage):{
3   case EX :
4     result = regs[rs] + regs[
          rd];
5   case WB :
6     regs[rd] = result;
7   }
8 }
9 void WB_fsm(){
10  if(resetn){
11  } else {
12    switch(opcode) {
13    case ADD :
14      add(WB);
15    }
16  }
17 }
18 void EX_fsm(){
19  if(resetn){
20  } else {
21    switch(opcode) {
22    case ADD :
23      add(EX);
24    }
25  }
26 }
```

```
1 void EX_fsm(){
2   if(resetn){
3   } else {
4     switch(opcode) {
5       case ADD :
6         result = regs[rs] +
              regs[rd];
7     }
8   }
9 }
10 void WB_fsm(){
11   if(resetn){
12   } else {
13     switch(opcode) {
14       case ADD :
15         regs[rd] = result;
16     }
17   }
18 }
```

Figura 3.18: Desenrolamento de constantes: partes da função `add` são inseridas nas funções `EX_fsm` e `WB_fsm` através da remoção do código não utilizado com as constantes `WB` e `EX` fixas.

generalizada.

A construção da máquina multiciclo da Figura 3.14 e da máquina *pipeline* da Figura 3.15 ilustram também como a técnica de *constant folding* e *dead code removal* são utilizadas. Por exemplo, a função *add* é inserida nos dois estágios do *pipeline writeback* e *execute* na Figura 3.18. Note, no entanto, como apenas a linha 4 da função é inserida na *FSM* do estágio de *execute* e apenas a linha 7 da função é inserida na *FSM* do estágio de *writeback*. Isso acontece porque quando a função *add* é chamada, por exemplo, com o valor constante `stage=EX` como parâmetro, a definição da linguagem nos permite inferir que as linhas 2 à 3 e 5 à 10 são irrelevantes na execução da função com o parâmetro fixo. Isso nos permite reescrever a função *add* como equivalente à execução da linha 4, que é inserida e mostrada na linha 6 do código na coluna da direita da Figura 3.18.

Essas duas técnicas são utilizadas em conjunto para inserir a lógica do comportamento

das instruções descritas pelo projetista na arquitetura estrutural.

Capítulo 4

Resultados experimentais

Neste capítulo serão mostrados como os conceitos propostos no capítulo anterior foram implementados e os resultados obtidos.

O capítulo começa com a ideia geral da metodologia adotada no que diz respeito às linguagens utilizadas, ferramentas, plataformas, verificação e testes.

A seguir, será explicado como as linguagens intermediárias do fluxo de síntese são transformadas e processadas, desde a descrição em ArchC até o diagrama esquemático do circuito resultante. As características da linguagem de cada etapa serão descritas, assim como as ferramentas que foram construídas para tornar a síntese possível.

Com o fluxo explicado, as medidas das principais características de cada nível são apresentadas, desde as medidas de eficiência da linguagem ArchC para o nível RTL até o desempenho do circuito final como frequência de operação, consumo de energia e área.

Por fim, a metodologia proposta é comparada com a aplicada nos trabalhos relacionados. Também será visto o que foi feito de diferente no desenvolvimento do ArchC RTL com relação ao desenvolvimento do ArchC comportamental.

4.1 Metodologia

O método escolhido para a síntese de modelos ArchC é muito parecido com o descrito anteriormente pelas outras linguagens. Como em [3], começaremos analisando quais são os processadores existentes implementados na linguagem e quais foram os paradigmas utilizados na implementação de cada modelo. A partir disso, identificamos os elementos arquiteturais e os generalizamos. Para comprovar a cobertura dos modelos existentes, escolhemos um subconjunto das instruções de cada modelo e implementamos o processador generalizado de forma análoga ao processo de escolha em [10]. O objetivo é escolher um subconjunto de instruções para ser sintetizado que represente as principais funcionalidades do processador sem perda da generalidade. Por fim, para cada processador, geramos um

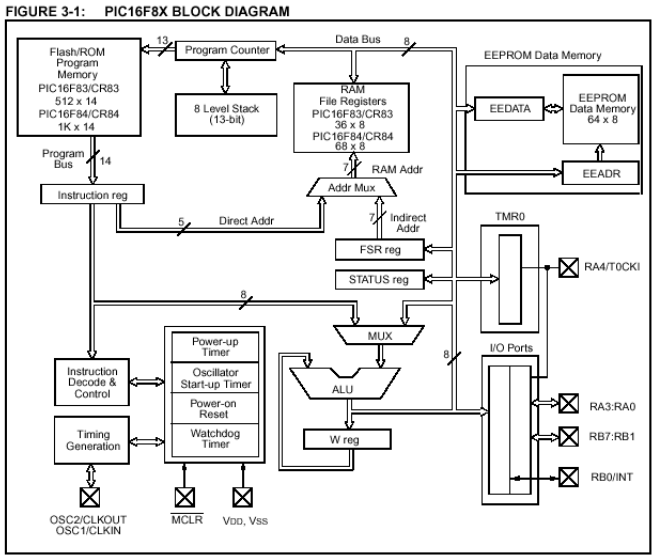


Figura 4.1: Arquitetura do PIC16F84.

conjunto de programas de teste à fim de verificar o funcionamento das implementações. O fluxo dessas etapas e ferramentas utilizadas são descritos e as restrições dessa abordagem apresentadas.

Os processadores que estão atualmente disponíveis em ArchC são o I8051, o PIC16F84, o MIPS-I, o R3000, o SPARC-V8, o PowerPC, o x86, o NIOSII, o AVR5 e o TMS320. Desses, apenas o I8051 e o PIC16F86 estão descritos no nível de ciclo que é o nível mais preciso que a linguagem suporta até esse trabalho.

O PIC16F84 é um processador de 8 bits que utiliza a arquitetura Harvard. O PIC possui 35 instruções e todas as instruções executam em apenas um ciclo, com exceção das instruções que não continuam no próximo endereço (por exemplo `GOTO`, `RETURN` etc). Note como a arquitetura mostrada na Figura 4.1 se assemelha com a arquitetura monociclo da Figura 3.12: as instruções são simples o suficiente para serem decodificadas e executadas no mesmo ciclo de relógio depois de terem sido buscadas da memória. Um subconjunto das instruções da Tabela 4.1 foi escolhido, mostrado na coluna da direita da tabela, para exercitar as capacidades e restrições da arquitetura monociclo.

O I8051 é um microcontrolador baseado em uma arquitetura Harvard cuja arquitetura é ilustrada na Figura 4.2. As instruções, mostradas na Tabela 4.2, tem tamanhos diferentes e são buscadas da memória em múltiplos ciclos. Essa arquitetura se assemelha com a arquitetura multiciclo da Figura 3.14: instruções como `reti`, `movx`, `jb`, `jmp` são implementadas em até três ciclos.

De fato, a descrição em ArchC existente utiliza uma arquitetura multiciclo para implementar o processador e um exemplo da implementação do comportamento da instrução

Tabela 4.1: Instruções sintetizadas do processador PIC1684

Mnemônico	Parâmetros	Descrição	Sintetizada
Instruções em bytes e registradores			
ADDWF	f, d	Soma W e f	**sim**
ANDWF	f, d	AND binário de W e F	não
CLRF	f	Zera f	**sim**
CLRW	-	Zera W	não
COMF	f, d	Complemento de F	não
DECF	f, d	Decrementa F	**sim**
DECFSZ	f, d	Decrementa f, pula se 0	não
INCF	f, d	Incrementa f	não
INCFSZ	f, d	Incrementa f, pula se 0	**sim**
MOVF	f, d	Move f	**sim**
MOVWF	f, d	Move W para f	**sim**
NOP	-	Operação nula	**sim**
RLF	f, d	Rotaciona para a esquerda de f	não
RRF	f, d	Rotaciona para a direita de f	não
SUBWF	f, d	Subtrai W de f	**sim**
SWAPF	f, d	Troca nibbles em f	não
XORWF	f, d	Ou-exclusivo de f	não
Instruções em bits			
BCF	f, b	Zera o bit b de f	não
BFS	f, b	Seta o bit b de f	não
BTFSC	f, b	Testa o bit b de f, pula se zero	**sim**
BTFSS	f, b	Testa o bit b de f, pula se setado	**sim**
Instruções de controle e literais			
ADDLW	k	Adiciona a literal k à W	**sim**
ANDLW	k	AND lógico do literal k à W	não
CALL	k	Chama sub rotina	não
CLRWDT	-	Zera o temporizador	não
GOTO	k	Salto incondicional para k	**sim**
IORLW	k	Ou-inclusivo da literal k com W	não
MOVLW	k	Move a literal k para W	**sim**
RETFI	-	Retorna de uma interrupção	não
RETLW	k	Retorna com para a literal em W	não
RETRN	-	Retorna da sub-rotina	não
SLEEP	-	Entra em mode standby	não
SUBLW	k	Subtrai k de W	não
XORLW	k	Ou-exclusivo de k com W	**sim**

Tabela 4.2: Instruções sintetizadas do processador 8051

Mnemônico	Descrição	Sintetizada
ACALL	Absolute Call	não
ADD, ADDC	Add Accumulator (With Carry)	**sim**
AJMP	Absolute Jump	não
ANL	Bitwise AND	não
CJNE	Compare and Jump if Not Equal	**sim**
CLR	Clear Register	não
CPL	Complement Register	não
DA	Decimal Adjust	não
DEC	Decrement Register	não
DIV	Divide Accumulator by B	não
DJNZ	Decrement Register and Jump if Not Zero	**sim**
INC	Increment Register	**sim**
JB	Jump if Bit Set	não
JBC	Jump if Bit Set and Clear Bit	não
JC	Jump if Carry Set	não
JMP	Jump to Address	não
JNB	Jump if Bit Not Set	não
JNC	Jump if Carry Not Set	não
JNZ	Jump if Accumulator Not Zero	não
JZ	Jump if Accumulator Zero	não
LCALL	Long Call	não
LJMP	Long Jump	**sim**
MOV	Move Memory	**sim**
MOVC	Move Code Memory	não
MOVX	Move Extended Memory	**sim**
MUL	Multiply Accumulator by B	não
NOP	No Operation	**sim**
ORL	Bitwise OR	não
POP	Pop Value From Stack	não
PUSH	Push Value Onto Stack	não
RET	Return From Subroutine	não
RETI	Return From Interrupt	não
RL	Rotate Accumulator Left	não
RLC	Rotate Accumulator Left Through Carry	não
RR	Rotate Accumulator Right	não
RRC	Rotate Accumulator Right Through Carry	não
SETB	Set Bit	não
SJMP	Short Jump	sim
SUBB	Subtract From Accumulator With Borrow	não
SWAP	Swap Accumulator Nibbles	não
XCH	Exchange Bytes	não
XCHD	Exchange Digits	não
XRL	Bitwise Exclusive OR	não

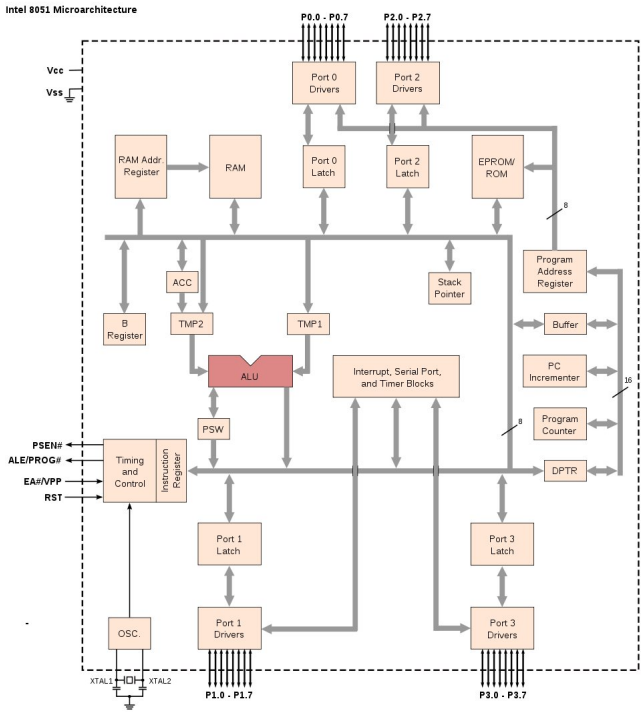


Figura 4.2: Arquitetura do I8051.

`JMP` pode ser visto na Figura 4.3. Um subconjunto da Tabela 4.2 foi escolhido para ser sintetizado (como mostra a coluna da direita da tabela) como um experimento do trabalho apresentado.

O MIPS R3000, ilustrado na Figura 4.4, é um processador de 32 bits projetado na universidade de Stanford [20]. A sua arquitetura é baseada na arquitetura mostrada na Figura 3.15. De forma parecida com o modelo do 8051 e do PIC, um subconjunto das instruções do processador, mostrado na Tabela 4.3, foi escolhido e sintetizado.

O SPARC também é um processador RISC, projetado pela SUN. Por fim, o PowerPC também é um processador RISC, projetado em 1991 pelo consórcio AIM composto pela Apple, IBM e Motorola. Sem perda de generalidade, esses processadores não foram sintetizados: todas essas arquiteturas são, de uma forma ou de outra, variações das arquiteturas mostradas nas Figuras 3.12, 3.14 e 3.15.

Além desses processadores, sintetizamos um subconjunto da JVM (*Java Virtual Machine*) ilustrado na Tabela 4.4. Escolhemos implementar um novo processador além dos já existentes para estudar todos os passos do fluxo de desenvolvimento do ArchC RTL e não apenas a síntese.

[32] define instruções sintéticas, ou pseudo instruções, que são instruções definidas pela linguagem sendo criada mas não implementadas diretamente pela máquina. São

```
1 void ac_behavior(jmp) {
2  switch (cycle) {
3   case 1:
4    dptr.range(7, 0) = IRAM.read(DPTRL);
5    break;
6   case 2:
7    dptr.range(15, 8) = IRAM.read(DPTRH);
8    break;
9   case 3:
10   acc = IRAM.read(ACC);
11   break;
12  case 4:
13   pc = acc + dptr;
14   break;
15 }
16 ac_cycle++;
17 return;
18 }
```

Figura 4.3: Implementação da instrução jmp do 8051.

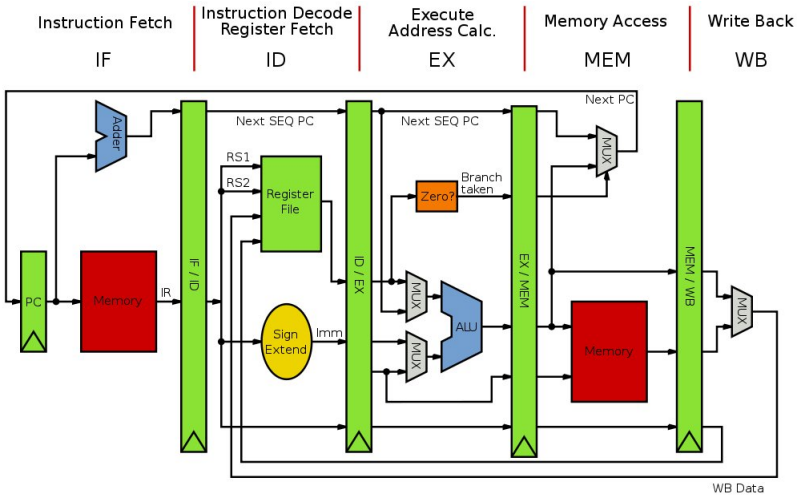


Figura 4.4: Arquitetura do MIPS R3000.

Tabela 4.3: Instruções sintetizadas do MIPS R3000

Instrução	Função	Resultado	Opcode	Funct
`add @1, @2, @3`	Adição	@1 ← @2 + @3	000000	100000
`sub @1, @2, @3`	Subtração	@1 ← @2 - @3	000000	100010
`and @1, @2, @3`	E binário	@1 ← @2 and @3	000000	100100
`or @1, @2, @3`	OU binário	@1 ← @2 or @3	000000	100101
`slt @1, @2, @3`	Set less than	@1 ← 1 set @2 < @3, 0 c.c.	000000	100101
`addi @1, @2, imm`	Adição	@1 ← @2 + imm	000000	100101
`beq @1, @2, imm`	Branch	@PC ← @PC + imm se @1 = @2	000000	100101
`jmp imm`	Jump	@PC ← @PC + imm	000000	100101
`lb @1, imm(@2)`	Load	@1 ← mem[@2 + imm]	000000	100101
`sb @1, imm(@2)`	Store	mem[@2 + imm] ← @1	000000	100101

Tabela 4.4: Instruções sintetizadas da JVM

Instrução	Resultado
`iconst_n`	push(n);
`isub`	`a ← pop(); b ← pop();` `push(b - a);`
`imul`	`a ← pop(); b ← pop();` `push(b * a);`
`if_icmpne`	`a ← pop(); b ← pop();` `if (a != b) pc ← pc + offset;`
`istore_n`	`stack[FP + n] ← pop();`
`iload_n`	`push(stack[FP + n]);`

`invokestatic`	`arg1 ← pop();`
	`push(FP); push(BP); push(ac_pc + get_size());`
	`FP = SP - 4; BP = SP - 3;`
	`push(arg1);`
	`ac_pc = ac_pc + offset;`

`return`	`ac_pc = pop(); BP = pop(); FP = pop();`

`ireturn`	`result ← pop();`
	`ac_pc = pop(); BP = pop(); FP = pop();`
	`push(result);`

`goto`	`ac_pc ← ac_pc + offset;`

```
pseudo_instr('mul %reg, %reg, %imm') {
  'addiu @at, @zero, %2';
  'mult %1, @at';
  'mflo %0';
}
```

Figura 4.5: Instruções sintéticas criadas pelo gerador de Assemblers para ArchC.

```
* fib.asm
* register usage: @3: n @4: f1 @5: f2
* return value written to address 255

fib:
  addi @3, @0, 8
  addi @4, @0, 1
  addi @5, @0, -1
loop:
  beq @3, @0, end
  add @4, @4, @5
  sub @5, @4, @5
  addi @3, @3, -1
  j loop
end:
  sb @4, 255(@0)
```

Figura 4.6: Programa para cálculo do Fibonacci de n utilizado no processador MIPS.

instruções tipicamente complexas, derivadas do encadeamento de instruções existentes, expandidas estaticamente em tempo de montagem como no exemplo mostrado na Figura 4.5. ArchC RTL se encaixa de forma transparente nesse modelo, sintetizando as instruções do processador que compõem as pseudo instruções utilizadas pelos montadores.

Com os processadores implementados, escrevemos uma série de programas de teste para verificar a execução nos nossos processadores. Os programas foram cuidadosamente escolhidos para exercitar todas as instruções piloto implementadas assim como dependência entre elas para o exercício de *data hazards*. A Figura 4.6 ilustra um exemplo de um programa que foi utilizado nos experimentos.

O fluxo de desenvolvimento envolveu diversas ferramentas e, a cada passo, a quantidade de detalhes da descrição cresceu. O fluxo começa, tipicamente, com uma descrição de alto nível em ArchC como mostra o primeiro passo da Figura 4.7. Essa descrição inicial é tipicamente desenvolvida e testada com o simulador comportamental de ArchC, `acsim`. Esse modelo implementa todas as instruções desejadas (como as da Tabela 4.3) e executa de forma correta os programas escolhidos, como o da Figura 4.6.

A partir dela, um novo simulador SystemC é criado em RTL pela ferramenta `acrtl`.

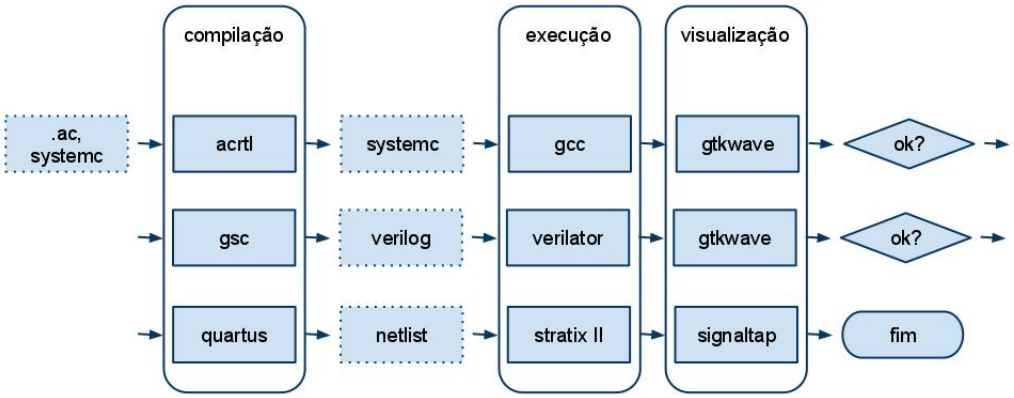


Figura 4.7: Fluxo dos programas e linguagens utilizadas na síntese dos processadores ArchC.

`acrtl` toma como entrada exatamente os mesmos arquivos que `acsim` e produz um simulador compatível com o simulador comportamental. A partir dessa primeira compilação o modelo estrutural RTL é executado (`acrtl` gera SystemC RTL que pode ser compilado em um executável utilizando-se um compilador C++, nesse caso o `gcc`) e verificado (SystemC gera formas de onda da execução do circuito, que podem ser, por exemplo, visualizadas e verificadas manualmente com um visualizador de ondas ou comparadas automaticamente com um modelo comportamental).

Com a arquitetura estrutural RTL, restrições são impostas e o modelo deve ser verificado novamente. Nesse estágio, o projetista tem de se preocupar com, por exemplo, o fato da memória não ser mais comportamental e o fato dos estágios de *pipeline* serem executados em paralelo.

Assim que o modelo é adaptado às restrições impostas pela nova arquitetura estrutural (resolução de múltiplos *drivers*, compartilhamento de recursos etc, como descritor anteriormente), ele é traduzido para uma linguagem que possa ser sintetizada. Nesse caso, utilizamos um tradutor SystemC-Verilog, chamado `gsc`. Da mesma forma, o código Verilog é executado e verificado, utilizando um simulador de Verilog. No nosso experimento, escolhemos utilizar o `Verilator` porque era uma opção de código fonte aberto e conhecida pelo autor.

Por fim, o modelo Verilog RTL é passado por um sintetizador. O papel do sintetizador é compilar a HDL em um circuito lógico conhecido como *netlist*. Escolhemos o *Quartus II* [40] como a ferramenta de síntese porque já era uma ferramenta conhecida pelo autor. O *Netlist* é compatível com uma descrição do esquemático do circuito e é gerado automaticamente pelo sintetizador lógico. O passo final do desenvolvimento envolve executar o circuito em uma plataforma alvo e verificar a execução. Para esse passo, nós utilizamos

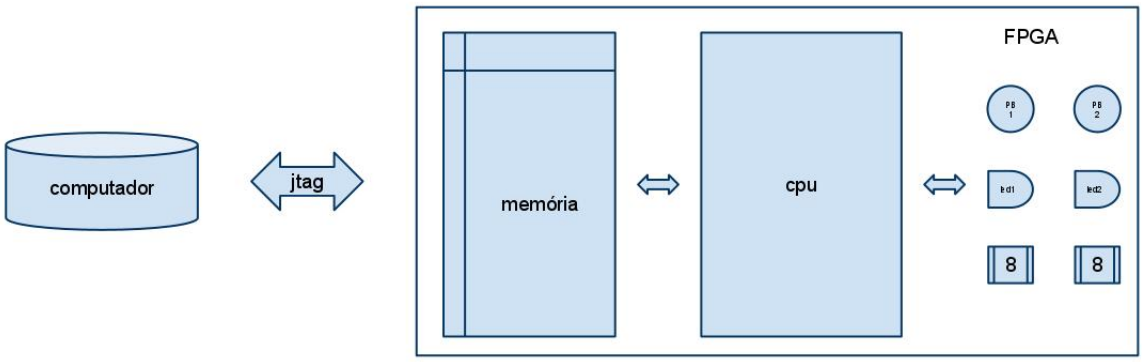


Figura 4.8: Plataforma de desenvolvimento e depuração.

uma *FPGA Stratix II* e um analisador lógico, chamado *SignalTap* [44].

Essa foi sem dúvida a fase mais demorada do processo. É nessa fase que descobrimos o que pode ser sintetizado ou não. É nessa fase que ficam claras as restrições de área, desempenho e consumo de energia. Em todos os modelos desenvolvidos, essa iteração exigiu rever a arquitetura escolhida e voltar para o primeiro passo diversas vezes. O primeiro problema que o projetista encontra é a restrição de área. Isso acontece porque, nos níveis anteriores, o projetista não se preocupava com o compartilhamento de recursos. Nessa fase, o projetista tem de escolher entre otimizar a área ocupada ou aumentar a área disponível que tinha sido prevista anteriormente no projeto. Outro problema tipicamente enfrentado é a necessidade de atingir determinado desempenho. Pela primeira vez no processo inteiro, a frequência de operação é definida e a arquitetura é revista a partir disso (caminhos críticos são eliminados, instruções são quebradas em mais ciclos, estágios dos *pipeline* são revistos etc).

Essa fase do desenvolvimento do *hardware* também é a que consume mais tempo e concentração do projetista porque as ferramentas de desenvolvimento são mais complexas e difíceis de serem utilizadas: como será mostrado no próximo capítulo, os compiladores (sintetizadores lógicos) são consideravelmente mais lentos, as ferramentas de depuração menos eficazes e a infraestrutura de execução menos flexíveis. A plataforma de desenvolvimento dessa fase utilizada nesse trabalho é ilustrada na Figura 4.8. A CPU é o resultado do projeto que estamos verificando. Utilizamos uma memória reconfigurável pelas ferramentas do *Quartus*, chamada *In-System Memory*, e depuramos todo o processador utilizando o analisador lógico da Altera chamado *SignalTap*.

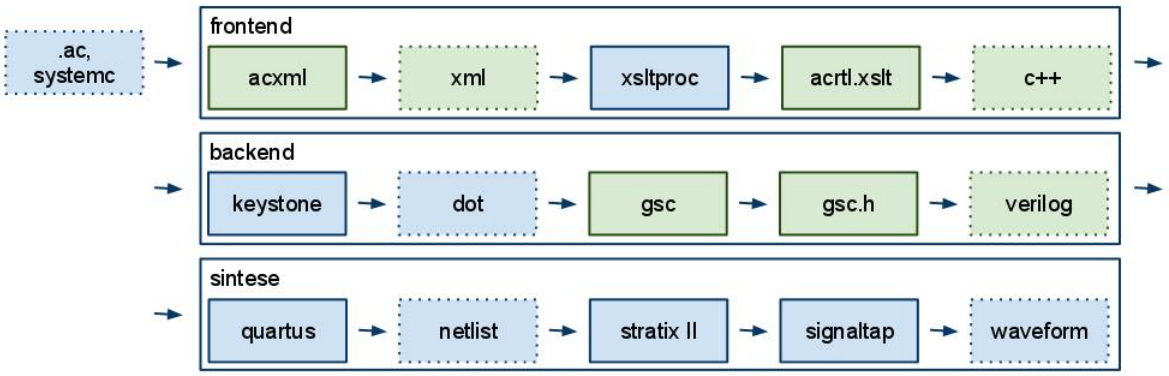


Figura 4.9: Fluxo do processamento da linguagem detalhado nas diversas ferramentas e linguagens utilizadas. O `acrtl` é o conjunto de ferramentas e linguagens que implementa o *frontend*.

4.2 Processamento

Nesta seção descreveremos o processamento das linguagens envolvidas na síntese dos processadores e as ferramentas que foram construídas para tornar isso possível.

A Figura 4.7 mostra o conceito geral do fluxo de síntese dos processadores: os arquivos de entrada estruturais (`AC_ARCH` e `AC_ISA`) e comportamentais (`ac_behavior`) alimentam o início do fluxo; o programa `acrtl` gera SystemC utilizando apenas diretivas RTL; o programa `gsc` traduz as diretivas RTL em Verilog, que é utilizada em um sintetizador lógico como o Quartus que gera o diagrama esquemático; por fim, o circuito é utilizado em uma FPGA.

A Figura 4.9 mostra como que cada um desses passos foi resolvido e as ferramentas que tiveram de ser construídas para torná-los possíveis. Nessa figura as caixas pontilhadas representam as linguagens utilizadas no processo e as caixas sólidas representam as ferramentas que fazem a transformação entre uma linguagem e outra. As caixas em verde dizem respeito às linguagens ou ferramentas que foram desenvolvidas como fruto desse trabalho e as em azul às ferramentas e linguagens já existentes que foram re-utilizadas. A principal ferramenta desenvolvida nesse trabalho, o `acrtl`, é composta pelas ferramentas que implementam o *frontend*.

A primeira linha da Figura 4.9 ilustra como os arquivos ArchC são transformados em SystemC RTL (C++). Logo abaixo, a segunda linha ilustra como esse modelo SystemC RTL é transformado em Verilog RTL. Por fim, a última linha mostra como o modelo Verilog é sintetizado e utilizado como uma máquina física.

4.3 *Frontend*

O *frontend* da ferramenta é responsável por gerar um modelo RTL em SystemC a partir da descrição estrutural e comportamental.

```
1  AC_ARCH(pic){
2    ac_regbank BANK:256;
3    ac_wordsize 16;
4    ac_reg W;
5    ac_reg PSW;
6
7    ARCH_CTOR(pic) {
8      ac_isa("pic_isa.ac");
9      set_endian("big");
10   };
11 };
12
13 AC_ISA(pic){
14   ac_format Format_Byte = "%dummy:2 %opbyte:6 %d:1 %f:7";
15   ac_format Format_Bit = "%dummy:2 %opbit:4 %b:3 %f:7";
16   ac_format Format_Literal = "%dummy:2 %oplit:6 %k:8";
17   ac_format Format_Control = "%dummy:2 %opctrl:3 %kaddress:11";
18
19   ac_instr<Format_Byte>  CLRWDT,CLRW,CLRF,ADDWF,MOVWF,ANDWF,DECF,INCF
       ,MOVF,
20      NOP,IORWF,SUBWF,XORWF,COMF,DECFSZ,INCFSZ,RLF,RRF,SWAPF,RETURN,
          RETFIE,SLEEP;
21   ac_instr<Format_Bit> BCF, BSF, BTFSC, BTFSS;
22   ac_instr<Format_Literal > MOVLW,ANDLW,IORLW,XORLW, ADDLW, SUBLW,
       RETLW;
23   ac_instr<Format_Control> GOTO, CALL;
24
25   ISA_CTOR(pic){
26     ADDWF.set_asm("ADDWF f, d");
27     ADDWF.set_decoder(opbyte=0x07);
28
29     NOP.set_asm("NOP");
30     NOP.set_decoder(opbyte=0x00, d=0, f=0x00);
31   };
32 };
```

Figura 4.10: Entrada estrutural do `acrtl`

Para entender como cada ferramenta do fluxo do *frontend* da Figura 4.9 foi projetada, considere a descrição estrutural `AC_ARCH` e `AC_ISA` de entrada do processo mostrada na Figura 4.10 e comportamental `ac_behavior` na Figura 4.11. A primeira decisão de projeto que tivemos de tomar foi qual abordagem utilizar para gerar código SystemC RTL (C++)

```
1 void ac_behavior(BTFSC) {
2   if ((BANK[f].read() & (0x1 << b)) == 0) {
3     ac_pc = (ac_pc.read() + 1) % 128;
4   }
5 }
6
7 void ac_behavior(MOVLW) {
8   W = k;
9 }
```

Figura 4.11: Entrada comportamental do `acrtl`.

a partir de uma descrição ArchC.

Note como as informações estruturais das diretivas em `AC_ARCH` e `AC_ISA` são inerentemente declarativas e não sequenciais, ou seja, não existe ordem ou comportamento representados nessa estrutura. São nessas diretivas que a estrutura do processador é composta, as instruções declaradas e os formatos das instruções definidos. Essa estrutura contém também como cada instrução é decodificada.

Para representar a estrutura declarativa, escolhemos transformar as informações contidas na linguagem ArchC para um formato estritamente declarativo: XML. Os arquivos .ac são lidos pelas bibliotecas do ArchC e transformadas em XML. O arquivo mostrado na Figura 4.12 mostra um trecho do arquivo XML gerador a partir dos arquivos da Figura 4.10. Note, por exemplo, como as declarações do banco de registradores em `AC_ARCH` são mapeada em tags `<storage type='ac_regbank' />` e como cada instrução é mapeada em tags como `<instruction name='GOTO' format='Format_Control' />`.

A representação em XML fornece um formato universal para as informações contidas nos arquivos .ac, o que facilita a criação de ferramentas. O formato XML provê também uma separação entre o modelo semântico e sintático: futuras mudanças na linguagem ArchC não afetam o fluxo do `acrtl`. A interface da ferramenta `acxml` é muito simples como mostra a Figura 4.13.

A segunda etapa da geração do código SystemC RTL recebe como entrada o arquivo XML gerado na etapa anterior e gera como saída o modelo RTL em SystemC do processador que implementa as estruturas declaradas no arquivo XML. Como mostra o exemplo da Figura 4.15, estamos interessados em gerar um `SC_MODULE` com os elementos estruturais declarados. A Figura 4.14 mostra a representação do diagrama de blocos do `SC_MODULE` gerado para o MIPS (`clk`, `resetn`, e as portas de interface com a memória externa).

Implementamos essa etapa com uma série de transformações XSLT no arquivo XML. O *template* utilizado no `acrtl` mostrado na Figura 4.17 é usado no processador XSLT como mostra a Figura 4.16.

Os *templates* XSLT utilizados no `acrtl` são generalizações das estruturas encontra-

```
1 <?xml version="1.0"?>
2 <archc>
3   <head><helper></helper></head>
4   <body project='pic'>
5     <arch filename='pic.ac' wordsize='16' fetchsize='16'>
6       <storage type='ac_regbank' name='BANK' width='16' size='256'/>
7       <storage type='ac_reg' name='W' width='16'/>
8     </arch>
9     <isa filename='pic_isa.ac'>
10      <format name='Format_Byte' size='16'>
11        <field name='dummy' width='2' signed='0'/>
12        <field name='opbyte' width='6' signed='0'/>
13        <field name='d' width='1' signed='0'/>
14        <field name='f' width='7' signed='0'/>
15      </format>
16 ...
17      <instruction name='ADDWF' format='Format_Byte' mnemonic='ADDWF'
            cycles='1'>
18        <asm>ADDWF f, d</asm>
19        <decode>
20          <field name='opbyte' value='0x7'/>
21        </decode>
22      </instruction>
23 ...
24    </isa>
25  </body>
26 </archc>
```

Figura 4.12: Exemplo de saída do programa acxml a partir da entrada mostrada na Figura 4.10.

```
1 acxml pic.ac -o pic.xml
```

Figura 4.13: Uso da ferramenta acxml: pic.ac é mostrado na Figura 4.10 e o pic.xml gerado é mostrado na Figura 4.12.

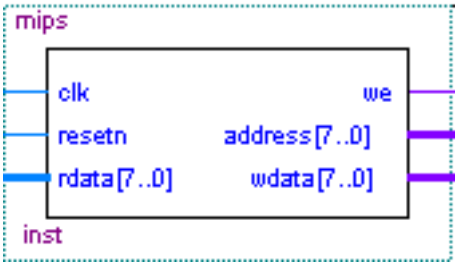


Figura 4.14: Diagrama de blocos do processador MIPS.

das em monociclos, multiciclos e *pipelines*. Esses *templates* generalizam decodificadores, *fetchers*, *pipelines*, controladores, banco de registradores, memórias etc, em função das informações disponíveis na linguagem ArchC.

4.4 *Backend*

O segundo passo no processo de síntese dos processadores é responsável por gerar um modelo Verilog RTL a partir do modelo SystemC RTL como mostra a Figura 4.9.

Uma decisão de projeto que tivemos de tomar foi qual HDL utilizar para sintetizar os processadores. Como foi visto na Tabela 2.10, cada ADL escolhe uma HDL para representar seus modelos RTL.

Escolhemos Verilog como a linguagem de saída do `acrtl` porque SystemC é mais próximo de Verilog do que VHDL ou outras alternativas menos conhecidas. Escolhemos implementar também o nosso próprio tradutor SystemC Verilog porque, até o momento da publicação desse trabalho, não existia nenhuma alternativa disponível que fosse estável e gratuita.

Tentamos ao máximo evitar escrever um tradutor SystemC Verilog. Analisamos alternativas comerciais como o `Design Compiler` da Synopsys [39], o Cynthesizer da Forte [42] e alternativas abertas como o `sc2v` [43]. Descartamos as alternativas comerciais porque periodicamente precisaríamos renovar as licenças de desenvolvimento e, ocasionalmente, a ferramenta poderia ser descontinuada. Descartamos as alternativas abertas porque nenhuma era estável o suficiente para suportar o nosso caso de uso.

O processo de tradução de SystemC RTL para Verilog RTL é mostrado na Figura 4.9. O SC_MODULE (Figura 4.15) composto com a descrição do comportamento das

```
SC_MODULE(pic) {
  sc_in<bool> clk;
  sc_in<bool> resetn;
  sc_in<sc_uint<FETCHSIZE> > rdata;
  sc_out<sc_uint<8> > address;
...
  sc_signal<sc_uint<8> > ac_pc;
  sc_signal<sc_uint<BUFFERSIZE> > ac_instruction;
  sc_signal<sc_uint<32> > ac_cycle;
  sc_signal<state_t> ac_state;
...
  // ac_regbank
  sc_signal<sc_uint<16> > BANK[256];
...
```

Figura 4.15: Trecho do código RTL gerado pelo `acrtl`.

```
xsltproc acrtl.xsl pic.xml > pic.cpp
```

Figura 4.16: Uso da ferramenta xsltproc: pic.xml é mostrado na Figura 4.12 o template na Figura 4.17 e o pic.cpp gerado é mostrado na Figura 4.15

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0">
  <xsl:template match="/">
// File automatically generated from <xsl:value-of select="archc/body
    /arch/@filename" />
SC_MODULE(<xsl:value-of select="archc/body/@project" />) {
  sc_in&lt;bool&gt; clk;
  sc_in&lt;bool&gt; resetn;
...
  // Contents of the achelper directive.
  <xsl:value-of select="archc/head/helper"/>
...
  // For each instruction format, create helper signals.
  <xsl:for-each select="archc/body/isa/format">
    <xsl:value-of select="@name"/>_t <xsl:value-of select="@name"/>;
  </xsl:for-each>
...
  </xsl:template>
</xsl:stylesheet>
```

Figura 4.17: Trecho do código XSLT de geração de c++.

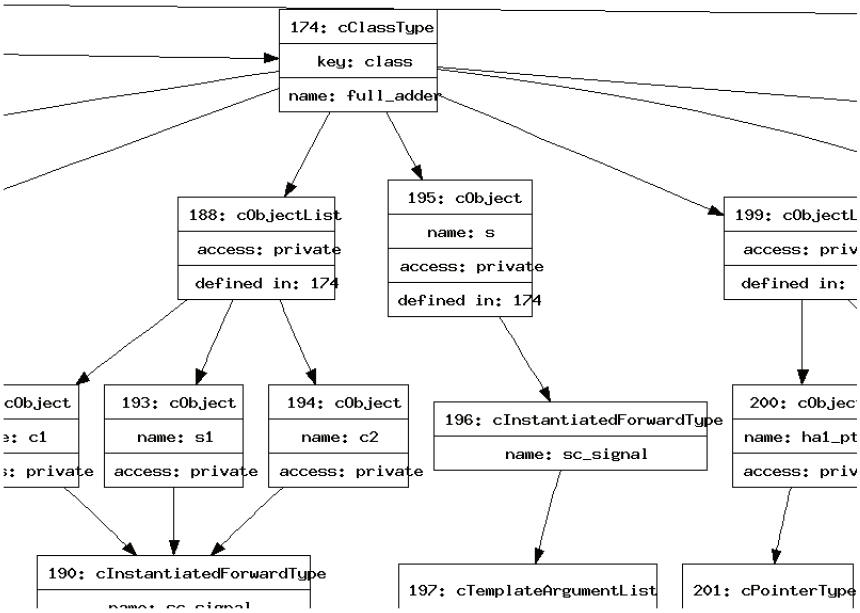


Figura 4.18: Trecho do grafo da AST de um módulo systemc.

instruções `ac_behavior` (Figura 4.11) é primeiro lido por um *frontend* C++ chamado `keystone` [41]. A saída do *frontend* é uma AST (*Abstract Syntax Tree*) como a da Figura 4.18. A saída é armazenada em um arquivo .dot apropriado para armazenar grafos. Desenvolvemos então uma ferramenta chamada `gsc` para traduzir a AST C++ para uma AST Verilog aplicando transformações na árvore. O tradutor, por fim, gera Verilog a partir da AST Verilog como o da Figura 4.19.

A Figura 4.20 mostra com a ferramenta `gsc` é utilizada.

4.5 Síntese lógica

A última etapa do processo mostrada na Figura 4.9 é a síntese lógica dos modelos Verilog em circuitos lógicos. A entrada desse processo é o arquivo Verilog RTL gerado anteriormente e as saídas são circuitos lógicos representados em *netlists*.

Nesse experimento, utilizamos uma FPGA para prototipar os nossos processadores. FPGAs são apropriadas para essa aplicação porque elas podem ser reconfiguradas inúmeras vezes. Foram necessárias diversas iterações dos *templates* XSLT até encontrarmos uma versão generalizada das estruturas.

Vale notar que o resultado gerado pelo trabalho descrito é genérico o suficiente para ser utilizado em outros dispositivos, como ASICs (*Application-specific integrated circuit*).

O Verilog gerado no passo anterior (Figura 4.19) é sintetizado no Quartus II. O sinte-

```
1 // file automaticaly generated by gsc
2
3 module PIC16F84( clk, resetn, rdata, address, wdata, waddress, we );
4   input               clk;
5   input               resetn;
6   input        [15:0] rdata;
7   output        [7:0] address;
8   output       [15:0] wdata;
9   output        [7:0] waddress;
10  output              we;
11
12  reg          [7:0]  address;
13  reg         [15:0]  wdata;
14  reg          [7:0]  waddress;
15  reg                 we;
16
17  parameter fetch = 0;
18  parameter decode = 1;
19  parameter execute = 2;
20 ...
```

Figura 4.19: Trecho do código Verilog gerado pelo `gsc`.

```
1 gsc pic.cpp -o pic.v
```

Figura 4.20: Exemplo de uso da ferramenta gsc: pic.cpp é mostrado na Figura 4.15 e o pic.v gerado é mostrado na Figura 4.19.

Tabela 4.5: Resultados da síntese das instruções da Tabela 4.1 do processador PIC16F84

Lógica combinacional (ALUTs)	Registradores dedicados (Registradores)	Frequência (Mhz)	Consumo de potência (mW)
821	1102	79.21	323.63

Tabela 4.6: Resultados da síntese das instruções da Tabela 4.2 do processador i8051

Lógica combinacional (ALUTs)	Registradores dedicados (Registradores)	Frequência (Mhz)	Consumo de potência (mW)
960	1185	106.47	323.66

tizador lógico da Altera cria o *netlist* a partir da descrição RTL que é então prototipado na FPGA Stratix II.

A Figura 4.21 mostra parte do diagrama esquemático que representa o resultado da síntese da implementação das instruções do processador PIC16F84 (Figura 4.10 e Figura 4.11). Essa figura ilustra a implementação da instrução `XORLW`. Note como os XORs são encadeados com os parâmetros da decodificação da instrução e direcionado para os multiplexadores que escrevem no banco de registradores.

A Figura 4.22 mostra em detalhes o decodificador RTL do ArchC implementando o cálculo do campo f do formato Format_Byte das instruções.

As características do processador gerado são mostradas na Tabela 4.5. Implementamos o PIC16F84 utilizando uma estrutura multiciclo e a síntese gerou um circuito que é capaz de executar à 79Mhz. Cada instrução gasta 3 ciclos para completar (`fetch`, `decode` e `execute`), o que significa que o processador executa 25M instruções por segundo. A área ocupada é majoritariamente dedicada aos registradores declarados: um banco de 256 registradores de tamanho 16 bits foi declarado na estrutura da Figura 4.10. As ALUTs representam os elementos lógicos de controle, como multiplexadores, portas lógicas etc. O processador consome aproximadamente 323.63mW implementado na Stratix II.

A Figura 4.23 mostra parte do diagrama esquemático que representa a implementação do processador i8051. A figura mostra em detalhes como a máquina de estados do cálculo de `state` é implementada: ela depende de entradas como `clk`, `resetn`, `cycle`, `instruction` e `rdata` e gera como saída os *bits* de saída que representam o estado `fetch`, `decode` e `execute`.

As características do processador multiciclo i8051 gerado são mostradas na Tabela 4.6. O processador é um pouco mais complexo e atinge uma frequência de operação de 106.47Mhz. O processador utiliza poucos registradores dedicados (1185) e consome 323mW de potência. O número de elementos lógicos utilizados para implementar a lógica de controle é comparável com o do PIC16F84.

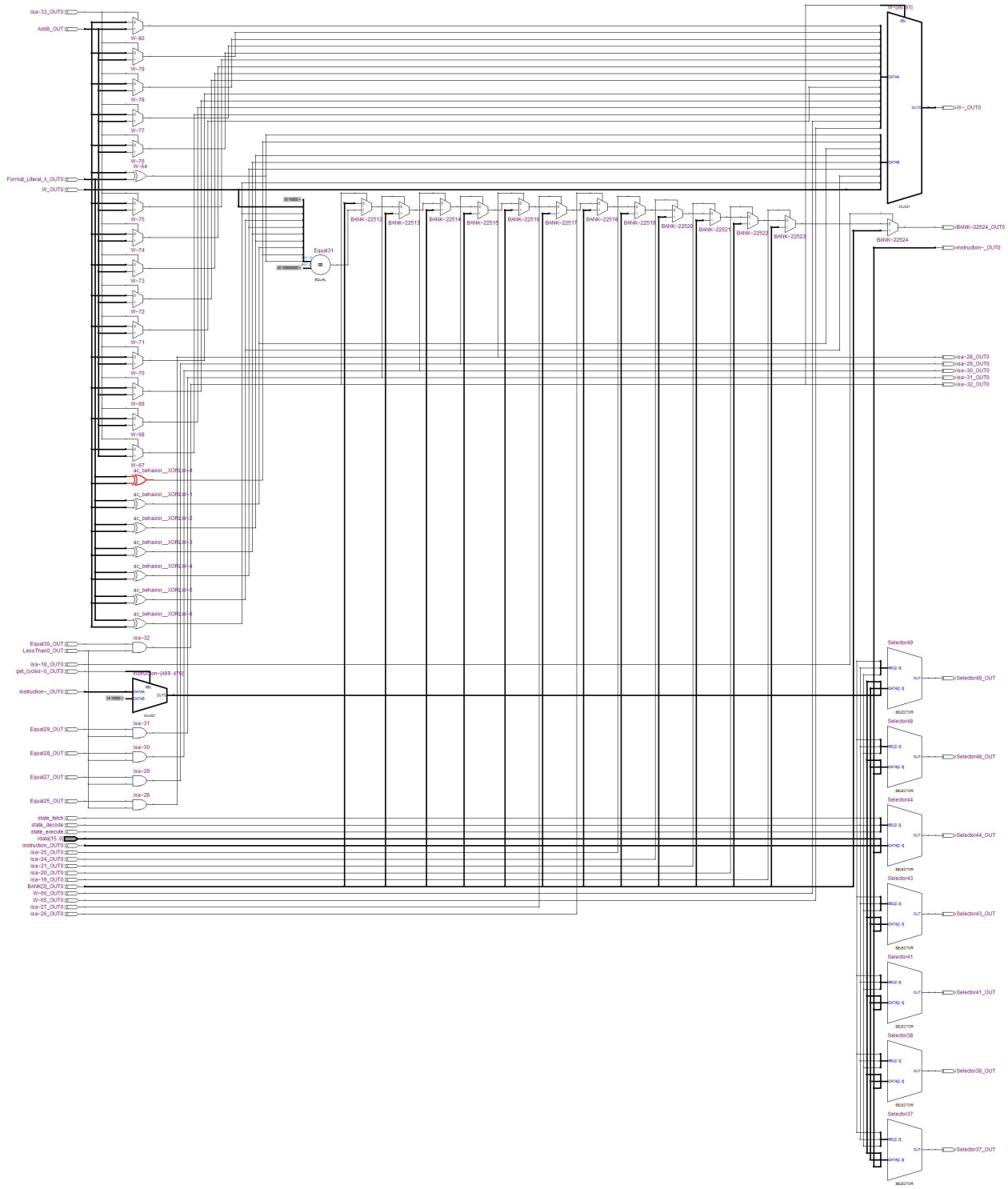


Figura 4.21: Circuito esquemático resultado da síntese do processador PIC.

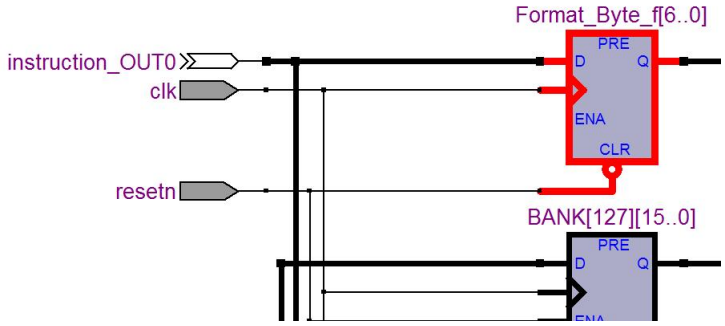


Figura 4.22: Circuito esquemático do cálculo do registrador do campo b do formato Byte do PIC.

Tabela 4.7: Resultados da síntese das instruções da Tabela 4.3 do processador MIPS-I

Lógica combinacional (ALUTs)	Registradores dedicados (Registradores)	Frequência (Mhz)	Consumo de potência (mW)
1445	2396	110.42	323.6

O processador MIPS-I também foi sintetizado utilizando o mesmo processo. Implementamos uma versão multiciclo e *pipeline* do MIPS para fornecer uma base de comparação com outras ADLs. A Tabela 4.7 e 4.8 mostram os resultados da síntese do MIPS.

A Figura 4.24 mostra o desenho esquemático que ilustra o resultado da síntese do MIPS-I. A caixa amarela da figura ilustra a máquina de estados do controlador do processador, que depende do ciclo sendo executado assim como do ponteiro de instruções.

A Figura 4.25 mostra em detalhes como parte do comportamento da instrução de soma é implementada: note como na figura o valor do sinal rformat_rs é utilizado como entrada do multiplexador usado para selecionar qual registrador vai ser utilizado como entrada do somador adiante.

A Figura 4.26 mostra em detalhes como que a decodificação das instruções do MIPS é feita: a partir do registrador que armazena a instrução, são criados registradores que recebem como entrada determinadas regiões da instrução (nesse caso, os primeiros 4 *bits* do registrador da instrução são mapeados para o parâmetro rd das instruções de formato R).

Tabela 4.8: Resultados da síntese das instruções da Tabela 4.3 do processador R3000

Lógica combinacional (ALUTs)	Registradores dedicados (Registradores)	Frequência (Mhz)	Consumo de potência (mW)
570	469	81.97	302.99

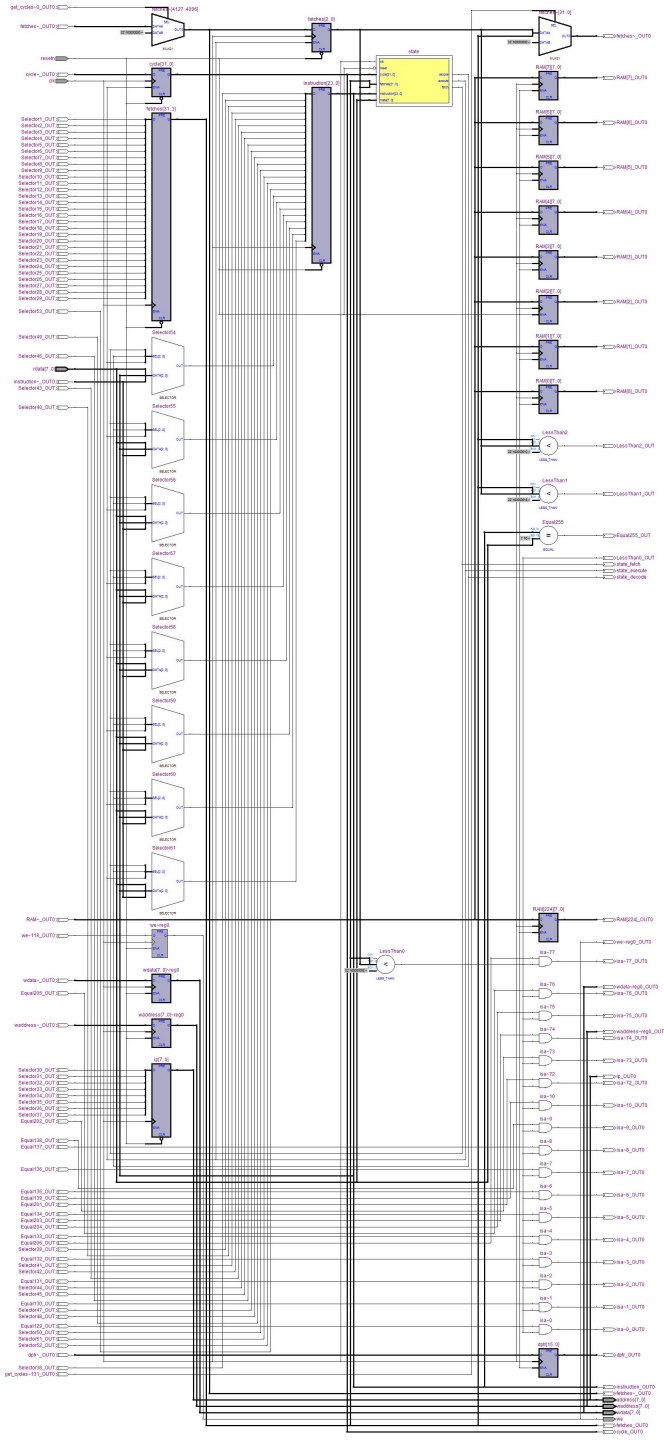


Figura 4.23: Parte do circuito esquemático resultado da síntese processador i8051.

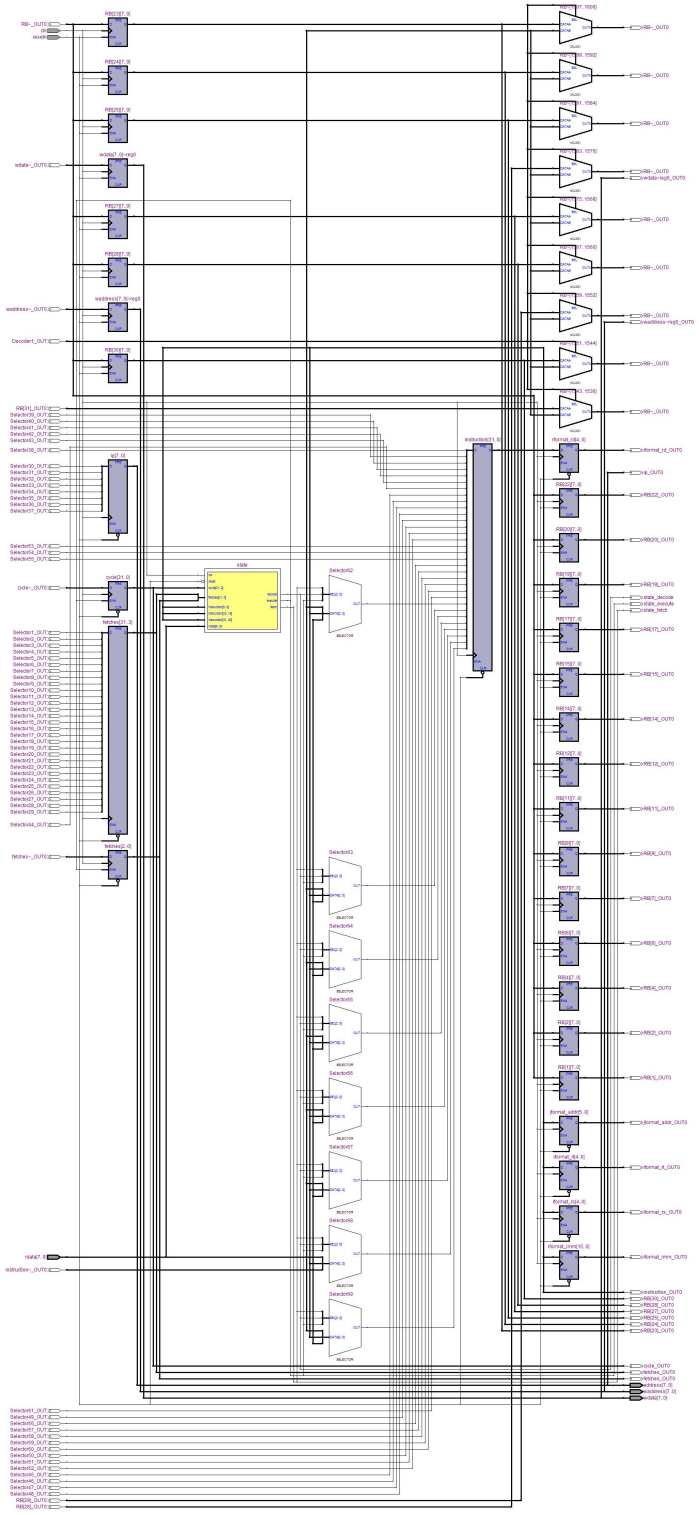


Figura 4.24: Circuito esquemático resultado da síntese processador MIPS-I.

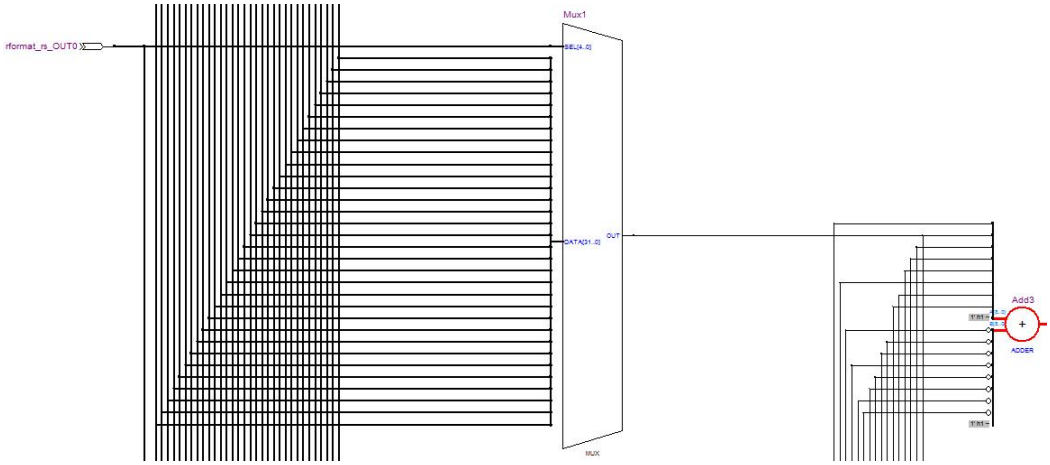


Figura 4.25: Detalhe do circuito que implementa a seleção do registrador rs do banco de registradores utilizado como entrada do multiplexador usado em uma operação de soma.

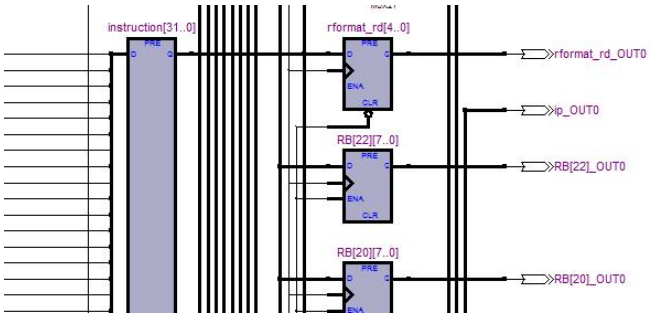


Figura 4.26: Detalhe da implementação da decodificação do parâmetro rd do formato rformat.

Tabela 4.9: Resultados da síntese das instruções da Tabela 4.4 da JVM

Lógica combinacional (ALUTs)	Registradores dedicados (Registradores)	Frequência (Mhz)	Consumo de potência (mW)
672	172	121.67Mhz	323.89

A Tabela 4.9 mostra os resultados da síntese da JVM. Foram necessárias quatro iterações no projeto da JVM até obtermos os resultados mostrados. Inicialmente, implementamos a JVM como um processador monociclo, com uma pilha fixa em 256 posições implementada com registradores. A partir daí, estendemos o modelo com uma memória externa, para retirar a restrição de poucas posições na pilha. Com a memória externa, foi necessário quebrar as instruções em diferentes ciclos.

4.6 Verificação e testes

Os processadores desenvolvidos foram testados em todos os níveis: desde o modelo SystemC comportamental gerado pelo `acsim`, o modelo SystemC RTL gerado pelo `acrtl`, o modelo Verilog gerado pelo `gsc`, o modelo *netlist* gerado pelo Quartus II até a execução do circuito na FPGA.

A cada passo da adição de detalhes na descrição, o resultado do modelo foi comparado com o do passo do nível de abstração anterior. Escolhemos um conjunto de programas de teste para verificar a implementação dos processadores nos diferentes níveis. Utilizamos os modelos comportamentais gerados pelo `acsim` para gerar resultados para a computação dos diferentes programas de entrada e comparamos esses resultados com os resultados obtidos pelos processadores rodando na FPGA gerados pelo `acrtl`.

Utilizamos programas de teste nos processadores para exercitar as instruções implementadas. Escolhemos os programas de teste de forma tal que:

1. Todo o comportamento das instruções implementadas fosse exercitado
2. Todas as características estruturais do processador fossem exercitadas

Por exemplo, a Figura 4.27 mostra o programa para o cálculo de Fibonacci de n utilizado no processador 8051. Esse é um programa interessante, porque faz uso das características estruturais (como o acesso à memória externa) e das diferentes instruções (aritméticas, controle, memória etc).

A Figura 4.28 mostra o programa para o cálculo de Fibonacci usado na arquitetura do PIC16F84. Analogamente, o programa utiliza conceitos importantes como acesso registrador-registrador, operações aritméticas e controle.

```
1 mov r1,#0ah  ;Load r1 with immediate data 0ah
2 mov dptr,#9000h ;load 9000h into dptr register
3 movx a,@dptr  ; move data from external memory location to a
4 inc dptr  ;increment dptr
5 mov r0,a  ;move data from a to r0
6 movx a,@dptr  ; move data from external memory location to a
7 back:mov r2,a  ;move data from a to r2
8 add a,r0  ;add a and r0
9 inc dptr  ;increment dptr
10 movx @dptr,a  ;move data from a to external memory location
11 mov r3,a  ;move data from a to r3
12 mov a,r2  ; move data from r2 to a
13 mov r0,a  ; move data from a to r0
14 mov a,r3  ; move data from r3 to a
15 djnz r1,back  ;decrement r1, if not 0, jump to label back
16 here:sjmp here
17 end
```

Figura 4.27: Programa para cálculo do Fibonacci de n usado no processador i8051.

```
1 CLRF    r0x20
2 MOVLW   0x01
3 MOVWF   r0x21
4 MOVLW   0x03
5 MOVWF   r0x22
6 __LOOP__
7 MOVF    r0x21,W
8 MOVWF   r0x23
9 MOVF    r0x21,W
10 ADDWF   r0x20,W
11 MOVWF   r0x21
12 MOVF    r0x23,W
13 MOVWF   r0x20
14 DECF    r0x22,F
15 MOVF    r0x22,W
16 XORLW   0x01
17 BTFSS   STATUS,2
18 GOTO    __LOOP__
19 MOVF    r0x21,W
20 MOVWF   r0x20
21 MOVWF   STK00
22 MOVLW   0x00
23 RETURN
```

Figura 4.28: Programa para cálculo do Fibonacci de n usado no processador PIC16F84 gerado com o SDCC.

```
1 class factorial extends java/lang/Object {
2   @signature "(I)I"
3   factorial {
4     @max_stack 4
5     @max_locals 1
6     iload_0
7     iconst_1
8     if_icmpne 0 5
9     iconst_1
10    ireturn
11    iload_0
12    iload_0
13    iconst_1
14    isub
15    invokestatic 0 8
16    imul
17    ireturn
18  }
19
20  @signature "([Ljava/lang/String;)V"
21  main {
22    @max_stack 1
23    @max_locals 2
24    iconst_5
25    invokestatic 0 8
26    istore_1
27    return
28  }
29 }
```

Figura 4.29: Programa para o cálculo de fatorial de n para a JVM em *bytecode*.

Para a JVM, utilizamos o programa da Figura 4.29 para exercitar os elementos arquiteturais e o comportamento das instruções. O programa é interessante porque faz bastante uso da arquitetura baseada em pilha da JVM com chamadas de funções recursivas.

A Figura 4.7 mostra as ferramentas utilizadas em cada etapa da verificação. O modelo comportamental gerado pelo `acsim` é verificado com a execução dos programas de teste. O modelo RTL gerado pelo `acrtl` é executado no simulador SystemC e verificado com a execução dos mesmos programas e a visualização do comportamento dos registradores com o `gtkwave` [45]. O modelo RTL gerado pelo `gsc` é também executado em um simulador Verilog e verificado no `gtkwave`. Para o modelo *netlist* gerado pelo Quartus, utilizamos o ModelSim para simular o circuito com temporização. Por fim, verificamos o *netlist* gerado na placa utilizando um analisador lógico, o SignalTap [44].

A Figura 4.31 mostra a forma de onda da execução do programa de cálculo do Fibonacci de n mostrado na Figura 4.28 no processador PIC. O pseudo codigo da Figura

```
1 static int main() {
2   char n = 8;
3   unsigned char n0 = 0;
4   unsigned char n1 = 1;
5   unsigned char naux;
6   do {
7     naux = n1;
8     n1 = n0 + n1;
9     n0 = naux;
10    n--;
11  } while(n != 1);
12  return n1;
13 }
```

Figura 4.30: Programa para cálculo do fibonacci de n usado no processador PIC16F84.

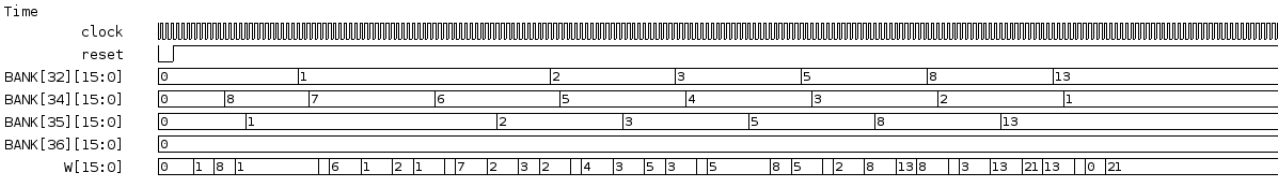


Figura 4.31: Forma de onda da execução do programa da Figura 4.28 no processador PIC obtida com um analisador lógico conectado à FPGA.

4.28 é mostrado na Figura 4.30. Note como o registrador `BANK[34]` é utilizado como o contador, o registrador `BANK[32]` é utilizado como `n0` e o registrador `BANK[35]` é utilizado como `n1`. É interessante notar como à cada iteração do contador (n = 8) o Fibonacci de n é calculado e finaliza a computação com Fibonacci(n = 8) = 21.

Analogamente, a Figura 4.32 mostra a forma de onda da execução do programa da Figura 4.27 no processador i8051. É interessante notar como a sequência de Fibonacci é escrita na memória externa (com os sinais `we` habilitando a escrita na memória, o endereço em `waddress` e o valor em `wdata`). As variáveis são armazenadas na `RAM`, com `RAM[0]` à `RAM[3]` armazenando os registradores `r0` à `r3`. Note também como `dptr` exerce um papel importante no acesso à memória externa à cada escrita do resultado do Fibonacci de n.

As Figuras 4.33 e 4.34 mostram a execução do cálculo do Fibonacci de n obtidas com um analisador lógico conectado à placa da Stratix II. Da mesma forma, é interessante analisar como os registradores se comportam até a escrita na memória externa é habilitada (`we`) com `wdata` contendo o valor Fibonacci no registrador 4. A execução do *pipeline* é ainda mais interessante (Figura 4.34), porque mostra o quão mais eficiente a execução é com os estágios executando em paralelo.

A Figura 4.35 mostra a forma de onda da JVM implementada na FPGA executando o

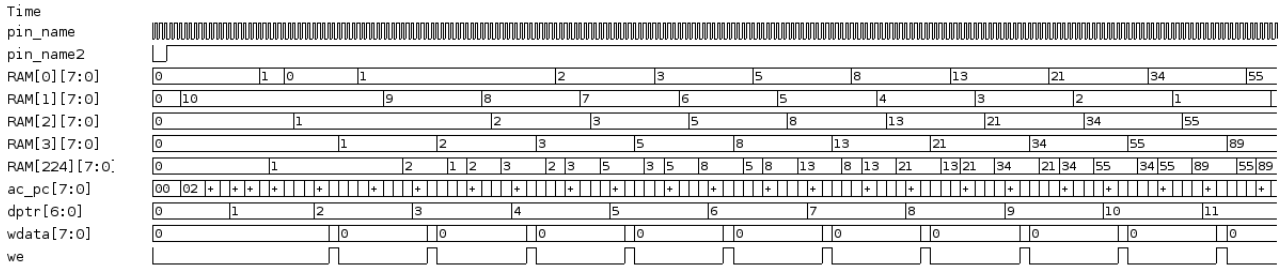


Figura 4.32: Forma de onda da execução do programa da Figura 4.27 no processador i8051 obtida com um analisador lógico conectado à FPGA.

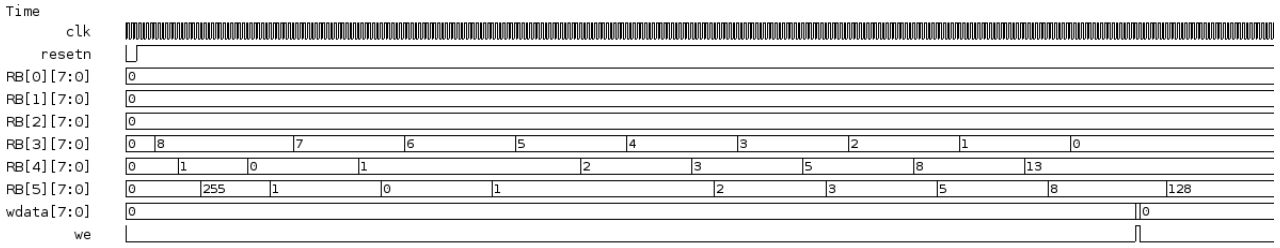


Figura 4.33: Forma de onda da execução do programa da Figura 4.6 no processador MIPS multiciclo obtida com um analisador lógico conectado à FPGA.

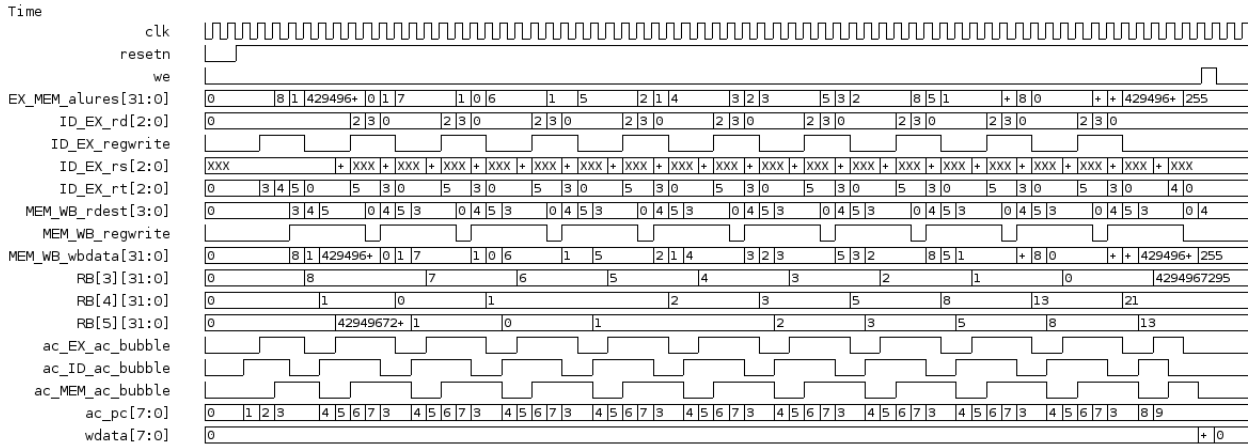


Figura 4.34: Forma de onda da execução do programa da Figura 4.6 no processador MIPS *pipeline*.

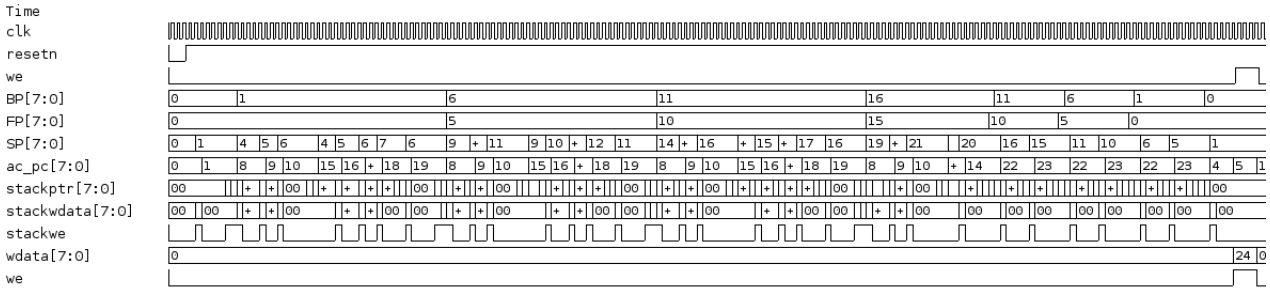


Figura 4.35: Forma de onda da execução do programa da Figura 4.29 no processador JVM obtida com um analisador lógico conectado à FPGA.

programa que calcula fatoriais da Figura 4.29. Essa forma de onda foi obtida usando um analisador lógico conectado à FPGA. A cada momento que `BP` (*base pointer*) e `FP` (*frame pointer*) mudam, um novo quadro é criado com a chamada da instrução `invokestatic` e os argumentos são empilhados com os ponteiros de `SP`, `stackwdata` e `stackwe`. Assim que a computação recursiva acaba os valores são desempilhados e o resultado (fatorial(n = 4) = 24) é escrito na memória externa (sinalizados pela saída `we` e `wdata`).

4.7 Análise comparativa

Nessa seção, iremos comparar os resultados obtidos à alternativas disponíveis.

Primeiramente, compararemos os resultados obtidos com HDLs, que é a abordagem mais aceita atualmente. Mostraremos que ArchC RTL é uma alternativa eficiente para o desenvolvimento de processadores, com ganhos de 2 à 3 vezes em tempo de desenvolvimento e perdas de desempenho na faixa de 20% à 30%.

A partir daí, compararemos os resultados obtidos com outras ADLs. Mostraremos que ArchC RTL consegue sintetizar uma gama comparável de processadores com desempenhos similares.

Por fim, faremos uma breve comparação entre ArchC RTL e ArchC comportamental. Ainda que as ferramentas tenham escopo diferentes, é importante notar quais são as diferenças entre elas e o que é necessário para tornar modelos ArchC comportamentais em RTL e vice versa.

4.7.1 ArchC vs HDLs

Para compararmos a nossa metodologia automatizada a partir de descrições de processadores em ArchC com a abordagem manual, desenvolvemos alguns dos processadores utilizando HDLs.

Tabela 4.10: Resultados da síntese das instruções da Tabela 4.3 do processador MIPS multiciclo e *pipeline* implementados manualmente comparados com os automaticamente. Resultados obtidos para o dispositivo Stratix II EPS290.

Arquitetura HDL/ADL	Elementos lógicos (ALUTs)	Frequência (Mhz)
MIPS-I HDL / MIPS-I ADL	1.7K / 2.3K	78.78 / 110
R3000 HDL / R3000 ADL	1.7K / 0.5K	107.52 / 81

Tabela 4.11: Tempo de desenvolvimento do conjunto de instruções escolhidos.

processador	`acsim` (dias)	`acrtl` (dias)	manual (dias)
PIC16F84	3	6	15
i8051	3	7	18
MIPS-I	1	2	4
R3000	1	5	15
JVM	4	8	não disponível

A Tabela 4.10 mostra os resultados obtidos pela síntese dos processadores descritos manualmente comparado com os resultados obtidos automaticamente.

A Tabela 4.11 mostra a comparação entre o tempo de desenvolvimento tomado pelo desenvolvimento manual e automatizado.

Esses resultados mostram, como será discutido a seguir, ganhos de duas a três vezes em tempo de desenvolvimento e perdas de 20% à 30% de desempenho (no que diz respeito à área e frequência de operação).

4.7.2 ArchC vs ADLs

A Tabela 4.12 mostra a comparação dos resultados obtidos com essa metodologia com a escolhida pelas outras ADLs.

Ainda que não diretamente relacionada com os resultados obtidos, o primeiro ponto de diferença é a HDL utilizada para representar o modelo sintetizável.

É interessante notar como todas as ADLs que suportam algum tipo de síntese escolheram começar seus trabalhos com a síntese de RISCs. Na maioria dos trabalhos publicados, essa escolha é justificada por conta da simplicidade do projeto e da notoriedade da arquitetura na academia. Como as outras ADLs, ArchC seguiu o mesmo caminho.

Nesse sentido, tanto EXPRESSION quanto LISA também possuem alguma versão do DLX implementada e sintetizada. É interessante notar como as propriedades dos modelos DLX são comparáveis em termos de desempenho e área ocupada com esse trabalho.

Tabela 4.12: Comparação de ArchC RTL com outras ADLs.

	EXPRESSION	LISA	AIDL	ArchC
Linguagem RTL	VHDL	SystemC	VHDL	SystemC/Verilog
Modelo experimental	RISC-DLX (20MHZ, 239K) Superescalar-DLX (20MHZ, 239K)	ICORE/DVB-T (125MHz, 59K, 14mW) LEON (226MHz, 37K) ASMD DSP	PA-RISC (não sintetizado)	PIC16F84 (79MHz, 1K, 323mW) I8051 (106MHz, 1.2K, 323mW) MIPS-I (110MHz, 2.3K, 323mW) R3000 (81MHz, 0.5K, 302mW) JVM (121MHz, 0.7K, 323mW)
Paradigmas	monociclo multiciclo *pipeline* superescalar VLIW multi-stage *pipeline*	monociclo multiciclo *pipeline* DSP	*pipeline* *data forwarding* *out of order*	monociclo multiciclo *pipeline* baseado em pilha
Eficiência de desenvolvimento (HDL / ADL)	7X	2.5X - 3.0X	1.2X - 4.6X	2X - 3X
Desempenho (ADL / HDL)	0.72X	0.7X - 0.9X		0.75X - 1.4X
Área (ADL / HDL)	1.2X	1X - 2.2X		1.3X
Otimizações	Compartilhamento de Recursos, *Bit-width*, *Overhead* de comunicação			

EXPRESSION e LISA ainda estão na frente no que diz respeito à quais paradigmas são suportados. EXPRESSION possui modelos sintetizáveis de superescalares, VLIW, *pipelines* de múltiplos estágios etc. AIDL suporta descrições de *data forwarding* e *out of order execution*, mas não possui processadores sintetizados.

Por um lado, a principal vantagem oferecida pelas ADLs é a eficiência de desenvolvimento, medida em quanto tempo um projetista demora para implementar um processador. Os ganhos de eficiência variam de 1.2 à 7 vezes mais rápidos. Nesse questão ArchC também tem uma desempenho comparável com as outras ADLs. Vale notar que essa medida é pouco objetiva, porque depende de fatores que foram desconsiderados nas análises como nível de comforto do projetista com a arquitetura, com a ADL e com a HDL, além das comparações terem sido feitas com amostras pequenas (dois à três processadores foram escritos com HDLs e ADLs e comparados o tempo de desenvolvimento).

Por outro lado, as desvantagens do uso de ADLs são as perdas de desempenho e de aumento na área utilizada. Os processadores descritos manualmente em HDLs são tipicamente 30% à 40% mais eficientes e ocupam 10% à 30% menos área. ArchC não foge à regra, apresentando níveis de desempenho dentro desse espectro. Uma exceção ocorreu com um dos modelos onde houve um ganho de 40% na frequência de operação do circuito.

Essa diferença se deu por um fator interessante: durante o desenvolvimento da ferramenta, foram feitas diversas otimizações nas estruturas RTL genéricas, que se propagaram para todos os modelos. Nesse caso, uma otimização feita no decodificador generalizado no experimento com o i8051 foi herdada e reutilizada pelo MIPS-I, que gerou um circuito 1.4 vezes mais eficiente do que o inicialmente escrito com uma HDL.

Esse é um resultado importante porque mostra que ADLs podem gerar circuitos mais eficientes que HDLs pois provêem otimizações generalizáveis que podem ser ignoradas por programadores de HDLs. Essa linha de raciocínio é muito parecida com a diferença de desempenho de programas em alto nível como C otimizadas por compiladores e programas em Assembly.

Por fim, a última linha da tabela mostra as otimizações disponíveis em cada ADL. EXPRESSION lidera também nesse quesito, com otimizações importantes como compartilhamento de recursos. Essa é uma otimização que se mostrou claramente necessária durante esse trabalho mas que não foi resolvida, como será discutido na próxima seção.

4.7.3 ArchC RTL vs ArchC comportamental

Para tornar o modelo sintetizável, o projetista é responsável por descrever o comportamento das instruções tendo em vista o nível de descrição RTL. Nesse nível, o projetista tem que tipicamente lidar com problemas que anteriormente não eram tratados como, por exemplo:

```
1  void lb(int cycle){
2    switch(cycle):{
3      case 0 :
4      regs[rd] = MEM[regs[rs]
           + imm];
5      case 1 :
6    }
7  }
```

```
1  void ac_behavior( lb ){
2    switch(cycle):{
3      case 0 :
4      re = 1;
5      raddr = regs[rs] + imm;
6      case 1 :
7      regs[rd] = rdata;
8    }
9  }
```

Figura 4.36: Acesso à memória por multiciclos: o código comportamental de acesso à memória na coluna da esquerda é mostrado na sua versão RTL na coluna direita.

1. Restrições de temporização
2. Recursos finitos
3. Interface entre componentes existentes

A Figura 4.36, por exemplo, mostra a diferença entre códigos tipicamente escritos para modelos comportamentais e modelos RTL. Na coluna da esquerda, o acesso à memória é feito de forma comportamental, sem considerar que a memória é na realidade um módulo externo. Na coluna da direita, no entanto, o acesso à memória é feito da maneira apropriada, considerando os pinos de entrada e saída da interface da memória.

Outra diferença comum encontrada entre modelos comportamentais e modelos RTL é o acesso à sinais em diferentes métodos, como mostra a Figura 4.37. Note como no código comportamental o registrador `ac_pc` está sendo atribuído em dois diferentes estágios do *pipeline* que são executados em paralelo. A cada ciclo do relógio, o valor de `ac_pc` não está definido se vai ser o valor do estágio `IF` ou o valor do estágio `WB`. Compare com o código RTL, mostrado na coluna da direita da tabela. No código RTL, o valor de `ac_pc` é explicitamente definido com o auxilio de um multiplexador, ora atribuindo o valor do estágio `IF` ou o valor do estágio `WB` (no caso de um salto acontecer, sinalizado pelo sinal `branch_en`).

Por fim, no nível RTL temos que lidar com o fato dos elementos serem finitos. O código comportamental da Figura 4.38 mostra a implementação de duas instruções, adição e o carregamento da memória. Em ambas, um somador de 32 *bits* é utilizado. No código RTL da coluna da direita fica claro o compartilhamento do somador pelas duas instruções o que diminui a área utilizada pelo circuito. A necessidade de resolver problemas de compartilhamento de recursos varia conforme quantos elementos estão disponíveis (por exemplo, quantos elementos o dispositivo alvo tem disponível) e quão capaz o sintetizador lógico utilizado é de fazer otimizações automáticas.

```
void ac_behavior(instruction){
  switch(stage) {
  case IF :
    ac_pc = ac_pc + 4;
  }
}
void ac_behavior(beq){
  switch(stage) {
  case EX :
    result = regs[rs] == regs[
       rt];
  case WB :
    if(result)
      ac_pc = ac_pc + imm;
  }
}
```

```
void ac_behavior(instruction){
  switch(stage) {
  case IF :
    if(!branch_en){
      ac_pc = ac_pc + 4;
    } else {
      ac_pc = branch_pc;
    }
  }
}
void ac_behavior(beq){
  switch(stage) {
  case EX :
    result = regs[rs] == regs[
       rt];
  case WB :
    branch_en = false;
    if(result){
      branch_pc = ac_pc + imm;
      branch_en = true;
    }
  }
}
```

Figura 4.37: Múltiplos *drivers* ao `ac_pc`: uso de multiplexadores na coluna da direita para selecionar qual o próximo valor do registrador, controlados pelos sinais `branch_en` e `branch_pc` adicionados pelo usuário.

```
void ac_behavior( add ){
  switch(stage):{
    case EX :
    result = regs[rs] + regs[rd
       ];
  }
}
void ac_behavior( lb ){
  switch(stage):{
    case EX :
    address = regs[rs] + imm;
  }
}
```

```
void ac_behavior( add ){
  switch(stage):{
    case EX :
    alu_operation = ADD;
    alu_operand1 = regs[rs];
    alu_operand2 = regs[rd];
  }
}
void ac_behavior( lb ){
  switch(stage):{
    case EX :
    alu_operation = ADD;
    alu_operand1 = regs[rs];
    alu_operand2 = imm;
  }
}
```

Figura 4.38: Duplicação de somadores: o código à esquerda duplica os somadores na operação de `add` e `lb`, enquanto na coluna da direita uma unidade lógica e aritmética é reutilizada pelas duas instruções.

A Tabela 4.13 mostra a comparação entre quais partes da linguagem ArchC definida em [27] são comportamentais e quais são suportadas na síntese.

Algumas das funcionalidades da linguagem são tipicamente comportamentais e não foram incluidas na síntese. TLM é um bom exemplo desse caso: TLM é tipicamente utilizado em um nível alto da especificação e não se aplica à modelos RTL. A função `get_name` retorna uma *string* com o nome da instrução que está sendo executada e também não foi implementada em RTL, porque é tipicamente utilizada para depuração em simuladores.

Outras funcionalidades não foram implementadas por escopo. `ac_instr_counter` é um contador de instruções executadas e é um bom exemplo desse tipo de funcionalidade: não existe nada que impeça a sua implementação, mas ela não é uma funcionalidade necessária para verificarmos a validade da metodologia sendo proposta e pode ser implementada em um trabalho futuro.

A quase totalidade da linguagem ArchC foi implementada em RTL, com poucas exceções, como mostra a Tabela 4.13.

Tabela 4.13: Comparação de *ArchC RTL* com *ArchC comportamental*.

Feature	`acsim`	`acrtl`
Declaração estrutural		
ac_wordsize	sim	sim
ac_fetchsize	sim	sim
ac_format	sim	sim
ac_mem	sim	sim
ac_tlm_port	sim	**não** [1]
ac_tlm_intr_port	sim	**não** [1]
ac_reg_bank	sim	sim
ac_reg	sim	sim
ac_pipe	sim	sim
ARCH_CTOR	sim	sim
ac_isa	sim	sim
set_endian	sim	**não** [2]
ac_helper	sim	sim
Declaração das instruções		
ac_instr	sim	sim
ISA_CTOR	sim	sim
ac_asm_map	sim	sim [4]
set_asm	sim	sim [4]
set_decoder	sim	sim
set_cycles	sim	sim
pseudo_instr	sim	sim [4]
Declaração do comportamento		
get_size	sim	sim
get_cycles	sim	sim
get_name	sim	**não** [1]
ac_stall	sim	sim
ac_flush	sim	**não** [2]
delay	sim	**não** [3]
ac_anull	sim	**não** [3]
ac_pc	sim	sim
ac_cycle	sim	sim
ac_instr_counter	sim	**não** [2]

[1] Construções TLM são inerentemente funcionais.

[2] Essa construção é perfeitamente válida no nível RTL, mas não foi implementada como parte desse trabalho.

[3] ac_anull e delay são tipicamente utilizados em modelos comportamentais.

[4] Não utilizado diretamente pelo `acrtl`, mas por ferramentas relacionadas (`acasm`, `acsim` etc).

Capítulo 5

Conclusões

Nesse trabalho mostramos a viabilidade da síntese de modelos de processadores descritos em ArchC e um conjunto de ferramentas que implementa essa transformação. Mostramos que a metodologia proposta é equivalente às implementações relacionadas e que ArchC tem de fato um subconjunto RTL-sintetizável.

Definimos quais são as características da linguagem que não são sintetizáveis e criamos propostas para quais mudanças devem ser feitas na descrição para torná-las sintetizáveis.

Propusemos um algoritmo para transformar os modelos descritos em ArchC em modelos sintetizáveis Verilog e construímos as ferramentas necessárias para implementar esses conceitos.

Uma série de processadores foram descritos especificamente para validar o algoritmo proposto. Sintetizamos processadores relevantes na academia e na industria, como o PIC16F84, o I8051, o MIPS-I, o R3000 e uma JVM. Programas de teste foram gerados e testados no circuito digital real implementado em FPGAs.

5.1 Contribuições

A principal contribuição desse trabalho é o preenchimento da lacuna entre descrições ArchC e a implementação real de processadores. Os resultados mostrados nesse trabalho reconectam ArchC à natureza física inerente à todos processadores, mantendo todos os benefícios herdados pelas linguagens de alto nível.

A partir desse trabalho, os projetistas que utilizam ArchC para descrever seus processadores podem assumir que a linguagem expressa o circuito desejado desde os níveis mais abstratos até os níveis mais concretos. Os mesmos modelos utilizados durante o desenvolvimento da arquitetura são reutilizados e a transformação entre as etapas são agora conhecidas. Os modelos ArchC possuem agora um *feedback* de características físicas, como frequência de operação, consumo de potência e utilização de área que influenciam

o projeto da arquitetura.

Esse trabalho adiciona ao repertório de modelos ArchC um subconjunto sintetizável dos processadores PIC16F84, I8051, MIPS-I, R3000 e JVM no nível RTL (Verilog). As principais características desses projetos foram implementadas, sintetizadas e prototipadas em FPGA. Mostramos que o nosso trabalho é genérico o suficiente para essas descrições serem estendidas para implementar todas as instruções e funcionalidades desses processadores.

Esse trabalho também contribui com uma série de ferramentas úteis para o desenvolvimento de hardware. Um dos principais frutos práticos desse trabalho é a criação de uma alternativa de código fonte aberta para o problema de tradução da linguagem SystemC para Verilog. Ainda que esse não seja um problema academicamente relevante, na prática ele se mostra útil e desejável, sendo atualmente utilizado por inúmeros grupos de pesquisa e empresariais.

Esse trabalho foi apresentado no décimo segundo encontro da *North American SystemC User's Group (NASCUG)* em San Jose, Califórnia.

5.2 Trabalhos futuros

Existem diversas atividades que gostaríamos de ter feito nesse trabalho que não foram feitas.

A primeira e mais óbvia é a síntese completa dos processadores. Ainda que sintetizar os processadores completamente não adicione muito academicamente, na prática é interessante ter especificações completas. Os processadores sintetizados são suficientes para rodar um número muito pequeno de programas e, na realidade não estão prontos para serem utilizados fora do contexto desse trabalho. Aproximadamente 20% à 60% de todas as instruções dos processadores foram implementadas, mas acreditamos que essas instruções piloto são suficientes para exemplificar como as demais podem ser implementadas.

A segunda linha de desenvolvimento que não foi endereçada se refere à síntese comportamental das descrições. A metodologia proposta define as características suficientes da descrição ArchC para serem consideradas RTL-sintetizáveis, mas não define apenas as necessárias. São conhecidos métodos que inferem circuitos lógicos a partir de descrições comportamentais, chamados de algoritmos de síntese comportamental, que poderiam ser utilizados para relaxar as restrições impostas por esse trabalho no que se refere à capacidade de síntese dos processadores. Relaxar as restrições significa abstrair detalhes da descrição sem perder a capacidade de síntese.

Outra linha de extensão desse trabalho diz respeito ao espectro de paradigmas que podem ser expressos em ArchC e sintetizados. Com informações sobre latência, frequência de operação, consumo de potência e área ocupada, fica cada vez mais importante suportar

paradigmas de arquitetura mais eficientes como execução fora de ordem, superescalares, VLIW, hierarquias de memórias, *pipelines* multi-ciclos (DSPs) etc. Esses paradigmas devem ser suportados na linguagem desde o nível mais abstrato possível até o nível RTL.

Por fim, existem diversas otimizações descritas na literatura que poderiam ter sido utilizadas. A solução para o problema do compartilhamento de recursos, por exemplo, ficou claramente útil nos experimentos mas não foi resolvida. Conhecem-se otimizações que podem ser aplicadas nos diversos eixos que caracterizam um processador como, por exemplo, frequência de operação, consumo de potência e área utilizada que seriam interessante aplicar à síntese de modelos ArchC.

Referências Bibliográficas

[1] P. Mishra, A. Kejariwal, N. Dutt Rapid Exploration of Pipelined Processors Through Automatic Generation of Synthesizable RTL Model, 14th IEEE International Workshop on Rapid Systems Prototyping, San Diego, CA, USA, 2003.

[2] P. Mishra, A. Kejariwal, N. Dutt, Synthesis-driven Exploration of Pipelined Embedded Processors, 17th International Conference on VLSI Design, Bangalore, Índia, 2004.

[3] O. Schliebusch, M. Steinert, G. Braun, A. Nohl, RTL Processor Synthesis for Architecture Exploration and Implementation, Design, Automation and Test in Europe Conference and Exhibition Designers Forum, 2004.

[4] O. Schliebusch, A. Chattopadhyay, E. Witte, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, Optimization Techniques for ADL-Driven RTL Procesor Synthesis, 16th IEEE International Workshop on Rapid System Prototyping,Chania, Crete, Greece, 2005.

[5] O. Schliebusch, A. Hoffman, A. Nohl, G. Braun, H. Meyr, Architecture Implementation Using the Machine Description Language LISA, Bangalore , Índia, 2002.

[6] V. Ventakataraman, C. Wilcox, GEMS: An Automatic Layout Tool for Mimola Schematics, Proceedings of the 23rd ACM/IEEE conference on Design automation, Las Vegas, Nevada, USA, 1986

[7] G. Hadjiyiannis, P. Russo, S. Devadas, Automatic Architecture Evaluation for Hardware/Software Codesign, 36th ACM/IEEE conference on Design automation, New Orleans, LA, USA, 1999

[8] A. Fauth, M. Freericks, A. knoll, Generation of Hardware Machine Models from Instruction Set Descriptions, Workshop on VLSI Signal Processing, Veldhoven , Holanda, 1993

[9] Wei Qin,*Modeling and Description of Embedded Processors for the development of Software Tools*. 206f. Tese (Doutorado em Engenharia Elétrica) - Universidade de Princeton, 2004.

[10] T. Morimoto, K. Saito, H. Nakamura, T. Boku, K. Nakazawa, Advanced Processor Design Using Hardware Description Language AIDL, Design Automation Conference, Anaheim, Califórnia, USA, 1997.

[11] D. Goodwin, D. Petkov, Automatic Generation of Application Specific Processors, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San José, CA, USA, 2003

[12] R. Gonzalez, Xtensa: A Configurable and Extensible Processor, *Micro, IEEE*, v.20, n.2, p.60–70, 2000.

[13] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, M. Imai, PEAS-I: A Hardware/Software co-design for ASIPs, Design Automation Conference, Dallas, Texas, USA, 1993.

[14] P. Grun, A. Halambi, N. Dutt, A. Nicolau, RTGEN - An Algorithm for Automatic Generation of Reservation Tables From Architectural Descriptions, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v.11, n.4, p.731–737, 2003.

[15] P. Preeti Ranjan, Describing Synthesizable RTL in SystemC, Proceedings of the 14th International Symposium on Systems Synthesis, Madrid, Espanha, 2001.

[16] R. M. Tomasulo, An Efficient Algorithm For Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development* v.11, n.1, p.25–33, 1967.

[17] O. Schliebusch, A. Chattopadhyay, E. Witte, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, Optimization Techniques for ADL-driven RTL Processor Synthesis, 16th IEEE International Workshop on Rapid System Prototyping, Montreal, Canadá, 2005.

[18] P. Schaumont, I. Verbauwhede, K. Keutzer, M. Sarrafzadeh, A Quick Safari Through the Reconfiguration Jungle, Annual ACM IEEE Design Automation Conference, Las Vegas, Nevada, USA, 2001.

[19] Y. Boshmaf, T. Bergmann, Synthesis of RTL Descriptions for Architecture Exploration, Seminário: Algorithms for Design-Automation - Mastering Nanoelectronic Systems, http://www.ra.informatik.uni-stuttgart.de/studies/ss07/adams/07_rtl_report.pdf, 2007.

[20] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 4^a ed., Morgan Kaufmann Publishers, 2007.

[21] B. Al-Hashimi, System-on-chip: Next Generation Eletronics, 1^a ed., The Institution of Engineering and Technology, London, United Kingdom, 2006.

[22] H. Tomiyama, A. Halambi, P. Grun, Architecture Description Languages for Systems-on-Chip Design, In Proceedings of the 6th Asia Pacific Conference on Chip Design Languages (Fukuoka, Japan, Oct.). 109-116, 1999.

[23] V. Rajesh, R. Moona, Processor Modelling for Hardware Software Codesign, Proceedings of the 12th International Conference on VLSI Design, Goa, Índia, 1999.

[24] S. Basu, R. Moona, High Level Synthesis from Sim-nML Processor Models, 16th International Conference on VLSI Design, New Delhi, Índia, 2003.

[25] The ArchC Team, The ArchC Architecture Description Language, Reference Manual, http://archc.sourceforge.net/, 2007.

[26] S. Rigo, R. Azevedo, G. Araújo, The ArchC architecture description language, Relatório Técnico, IC-03-015, 2003.

[27] S. Rigo, G. Araújo, M. Bartholomeu, R. Azevedo, ArchC: A SystemC-Based Architecture Description Language, 16th Symposium on Computer Architecture and High Performance Computing (SBAC'04), Foz do Iguaçu, Brasil, 2004.

[28] P. Viana, E. Barros, S. Rigo, R. Azevedo, G. Araújo, Exploring Memory Hierachy with ArchC, 15th Symposium on Computer Architecture and High Performance Computing (SBAC'03), São Paulo, Brasil, 2003.

[29] M. Bartholomeu, S. Rigo, R. Azevedo, G. Araújo, Emulating Operating System Calls in Retargetable ISA Simulators, Relatório Técnico, IC-03-29, 2003.

[30] F. Klein, G. Araújo, R. Azevedo, PowerSC: A SystemC Framework for Power Estimation, 6th NASCUG (co-located with DVCon), San José, USA, 2007.

[31] R. Maciel, B. Albertini, S. Rigo, R. Azevedo, A Methodology and Toolset to Enable SystemC and VHDL Cosimulation, IEEE Computer Society Annual Symposium on VLSI (ISVLSIÂ´07), Porto Alegre, Brasil, 2007.

[32] A. Baldassin, P. Centoducatte, Geração Automática de Montadores para Modelos de Arquiteturas Escritos em ArchC, 9th Brazilian Symposium on Programming Languages (SBLP05), Recife, Brasil, 2005.

[33] M. Bartholomeu, R. Azevedo, S. Rigo, G. Araújo, Optimizations for Compiled Simulation Using Instruction Type Information, 16th Symposium on Computer Architecture and High Performance Computing (SBAC'04), Foz do Iguaçu, Brasil, 2004.

[34] S. Rigo, M. Juliato, R. Azevedo, G. Araújo, P. Centoducatte, Workshop on Computer Architecture Education (WCAE'04), Held in Conjunction with International Symposium on Computer Architecture (ISCA), Munique, Alemanha, 2004.

[35] P. Lira, V. Schwambach, E. Barros, A Strategy for Specifying SystemC Micro-Controllers Models Using The ADL ArchC, IP Based SoC Design Conference & Exhibition, http://www.design-reuse.com/ipbasedsocdesign/slides_2005-ufpe_01.html, France, 2005.

[36] D. Alves, T. Lins, S. Neto, E. Barros, A 8051 uC Functional RTL Description Using SystemC For Platform Based Design, http://www.cin.ufpe.br/ greco/publicacoes/A_8051_uC_FUNCTIONAL_RTL_DESCRIPTION_USING_SYSTEMC_FOR_PLATFORM_BASED_DESIGN_djca,tsl,svfn,ensb.pdf.

[37] Sailer, Philip M.; Kaeli, David R.. The DLX Instruction Set Architecture Handbook, 1^a ed., Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996.

[38] CoCentric, SystemC Compiler: RTL User and Modelling guide, Versin 2001.08, http://www.es.ele.tue.nl/mininoc/doc/scrmg.pdf, 2008.

[39] H. Bhatnagar, Advanced ASIC Chip Synthesis: Using Synopsys Design Compiler, Physical Compiler, and PrimeTime, 2 ed., Kluwer Academic Publishers, 1999.

[40] Introduction to the Quartus II Software, Version 10.0, www.altera.com/literature/manual/intro_to_quartus2.pdf, San José, CA, 2010.

[41] Keystone: A parser and front-end for ISO C++, http://keystone.sourceforge.net, 2004.

[42] FORTE Design Systems, http://www.forteds.com/products/cynthesizer.asp, San José, CA, 2008.

[43] SystemC to Verilog Synthesizable Subset Translator :: Overview, http://opencores.org/project,sc2v, 2010.

[44] Design Debugging Using the SignalTap II Logic Analyzer, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf, 2010

[45] GTKWAVE, http://gtkwave.sourceforge.net/, 2010