José Carlos Meireles Monteiro Metrôlho

MiADL: Linguagem para Geração Automática de Simuladores Redireccionáveis

Tese submetida na Universidade do Minho para obtenção do grau de Doutor em Electrónica Industrial, área de conhecimento em Informática Industrial.

Universidade do Minho

DECLARAÇÃO

Nome:

José Carlos Meireles Monteiro Metrôlho

Endereço electrónico: jmetrolho@dei.uminho.pt Telefone: 919228266/ 272339300

Número de bilhete de identidade: 8448972

Titulo da Tese:

MiADL: LINGUAGEM PARA GERAÇÃO AUTOMÁTICA DE SIMULADORES REDIRECCIONÁVEIS.

Orientador(es):

Professor Carlos Alberto Caridade Monteiro e Couto e Professor Carlos Alberto Batista da Silva

\- Ano de Conclusão: 2007

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Electrónica Industrial – Área de Conhecimento em Informática Industrial

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO, APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, 12 de Julho de 2007

Assinatura:

AGRADECIMENTOS

O autor manifesta o seu mais sincero agradecimento a todas as instituições e pessoas que, com a sua valiosa colaboração, tornaram possível a realização deste trabalho.

Ao Doutor Carlos Couto e ao Doutor Carlos A. Silva, agradeço a orientação científica, o incentivo, as sugestões, discussões e condições conducentes à progressão dos trabalhos, a confiança e o apoio constantes ao longo de todo o trabalho. Agradeço-lhes as sugestões feitas durante a escrita e revisão final da tese.

Ao programa PRODEP III medida 5.3 por me ter proporcionado a possibilidade de estar com dispensa de serviço docente durante 3 anos. Esta bolsa permitiu que estivesse em Guimarães a desenvolver os trabalhos apresentados nesta tese.

Aos colegas do departamento de Electrónica Industrial, da Universidade do Minho, o meu sincero agradecimento pela camaradagem, bom ambiente e trocas de impressões no decorrer do meu trabalho. Especialmente ao Doutor Adriano Tavares e ao Mestre Sérgio Lopes.

Aos colegas do Departamento de Engenharia Informática do Instituto Politécnico de Castelo Branco, no qual desempenho funções, o meu sincero agradecimento pelo bom ambiente e trocas de impressões no decorrer do meu trabalho.

Aos meus pais e esposa agradeço o apoio que me têm dado sempre, especialmente nos momentos mais difíceis.

A todos aqueles que, de forma directa ou indirecta, contribuíram para a realização deste trabalho, o meu obrigado.

RESUMO

MIADL: LINGUAGEM PARA GERAÇÃO AUTOMÁTICA DE SIMULADORES REDIRECCIONÁVEIS

Os sistemas embutidos, além de fazerem cada vez mais parte da vida do cidadão comum, são dispositivos cada vez mais sofisticados e complexos. As equipas de desenvolvimento de sistemas embutidos têm por isso de lidar com complexidade crescente para atingir performances e requisitos cada vez mais exigentes (ex. hierarquias de memória complexas). Estes dispositivos exigem ferramentas de software – tais como: simuladores, depuradores, assembladores ou compiladores – que têm que acompanhar a evolução dos mesmos. Torna-se por isso fundamental desenvolver de forma rápida o conjunto de ferramentas para determinado sistema embutido para garantir espaço num mercado bastante competitivo, ou seja, conseguir um curto *time-to-market*. O desenvolvimento de aplicações que permitam a rápida e eficaz geração deste tipo de ferramentas assume relevo no âmbito do desenvolvimento dos sistemas embutidos.

Das ferramentas de software, o simulador é uma das essenciais para o desenvolvimento de novas arquitecturas computacionais. Entre as vantagens que apresenta, destaca-se a flexibilidade e baixo custo, uma vez que permite simular hardware em estágios iniciais do processo de desenvolvimento, sem necessidade de existência física do mesmo. Os primeiros simuladores eram desenvolvidos manualmente. Entretanto têm emergido linguagens de descrição de arquitecturas (ADLs) que facilitam a geração dessas ferramentas de uma forma automática, rápida, redireccionável e menos propensa a erros. Uma das vantagens destas linguagens consiste em permitirem gerar várias ferramentas a partir de uma única descrição, o que garante desde logo compatibilidade e coerência entre elas. Estas linguagens além de aplicação prática para fins industriais, podem também ser usadas para fins educacionais nomeadamente para o ensino de arquitecturas de microprocessadores.

Este trabalho de Doutoramento tem por objectivo contribuir para simplificar o processo de construção de simuladores, usados no projecto de sistemas embutidos. Pretende-se mostrar que a melhor forma de alcançar este objectivo consiste numa abordagem usando uma linguagem estruturada e que explora os comportamentos/sintaxes comuns ao conjunto de instruções da arquitectura alvo. Para suportar esta abordagem propõem-se uma linguagem que introduz uma forma diferente de descrever arquitecturas do conjunto de instruções (ISAs). A linguagem nomeada de *Minho Architecture Description Language* (MiADL), possui uma estrutura que explora o que é comum às instruções (comportamento e *assembly*) e permite a existência de blocos que são descritos uma vez e podem ser usados várias vezes. Desta forma a validação e identificação de incoerências fica facilitada e conduz a descrições claras, robustas e fáceis de depurar. As principais características desta linguagem são a existência de *scopes*, a inferência de argumentos, a estrutura em secções que permitem reutilização em diferentes partes da linguagem, a forma como lida com variabilidades e regularidades presentes em ISAs e a relação com a informação presente geralmente em manuais dos processadores. As modelações usando MiADL de vários ISAs complexos, de arquitecturas conhecidas, resulta em descrições bastante compactas, estruturadas e fáceis de explorar, quando comparadas com as conseguidas por outras ADLs.

Em termos de simulação, com base em modelações MiADL, na tese é apresentada a infra-estrutura de geração automática de simuladores redireccionáveis de ISAs usando a técnica de simulação compilada estática, recorrendo a *generic programming*. Esta possibilidade de recorrer àquela técnica de programação deve-se às características da linguagem, que com outras ADLs não é tão fácil de conseguir. Além de tornar o código mais compacto e estruturado, permite que o compilador nativo explore optimizações para conseguir bom desempenho do simulador.

ABSTRACT

MiADL: LANGUAGE FOR AUTOMATIC GENERATION OF RETARGETABLE SIMULATORS

Embedded system devices, besides having a growing importance in the common citizen's life, are more and more sophisticated and complex devices. The embedded systems development teams have to deal with the rising complexity to provide higher performance and more demanding requisites (ex. complex memory organizations). These devices require software tools – such as: simulator, debugger, linker, assembler or compiler - that must be proper to those devices. It is therefore essential to develop in a quick way the software toolkit, for the target embedded system, to assure place in a competitive market, in other words, to shorten time-to-market. The development of applications to efficiently generate the software toolkit is of great importance in the embedded systems development scenario.

From the set of software tools, the simulator is an essential tool for the development of new computational architectures. It brings flexibility and reduces the cost, since it allows the hardware simulation in early stages of the development process, without the physical existence of such hardware. The first simulators were hand-coded developed. Since then, new Architecture Description Languages (ADLs) have emerged, that have made easier the generation of these tools in an automatic, quicker, retargetable and error resilient way. One advantage of these languages is that they allow generating several tools from a single description. These languages, besides the practical applications for industrial purposes, can also be used for educational purposes namely for teaching microprocessors architectures.

The purpose of this PhD work is to contribute for the simplification of generating simulators to be used in the design of embedded systems. The aim is to demonstrate that the best way is achieved through the use of a structured language which explores the common behavioural/syntaxes of the target instruction set architecture. To support this approach a new language is proposed which introduces a new way to describe Instructions Set Architectures (ISAs). This language, named **Mi**nho **A**rchitecture **D**escription **L**anguage (MiADL), possesses a structure that explores what is common with the instructions (behaviour and assembly) and is block organized. The language

blocks can be described once and used several times. In this way the validation and detection of incoherencies is facilitated, leading to clear, robust and easy to debug descriptions. The main features of this language are: scopes existence; arguments inference; structure organized in sections which allows its reuse in different parts of the description; the way in which it deals with variability and regularities present in complex ISAs and similarity with the information usually available in processors manuals. The descriptions of several complex ISAs of known architectures using MiADL, result in rather compact, structured and easy to explore descriptions, when compared with the ones obtained by other ADLs.

In this thesis is also discussed the retargetable framework designed for automatic generation of compiled simulators, from MiADL descriptions, using generic programming to execute instruction behaviors. The use of function templates is possible due to MiADL features, with other ADLs this isn’t so easy to accomplish. Besides making the code more compact and structured, allows to explore optimizations from the native compiler to achieve improvements in the simulator´s performace.

Índice Geral

Lista de Abreviaturas e Siglas

ADL	*Architecture Description Language*
ANTLR	*ANother Tool for Language Recognition*
ASIPs	*Application-specific Instruction-set Processors*
BNF	*Backus-Naur Form*
CCL	*Calling Convention Language*
CISC	*Complex Instruction Set Computer*
CSDL	*Computer Systems Description Languages*
DSE	*Design Space Exploration*
DSP	*Digital Signal Processors*
EBNF	*Extended Backus-Naur Form*
ELF	*Executable and Linkable Format*
EPIC	*Explicitly Parallel Instruction Computing*
FSCS	*Fast Static Compiled Simulation*
GPP	*General Purpose Processors*
HDL	*Hardware Description Language*
HMDES	*High-level Machine Description*
ISA	*Instruction-set Architecture*
IS-CS	*Instruction-Set Compiled Simulation*
ISDL	*Instruction Set Description Language*
ISS	*Instruction Set Simulator*
JIT-CCS	*Just-in-time Cache Compiled Simulation*
LISA	*Language for Instruction Set Architecture*
MADL	*Mescal Architecture Description Language*
MiADL	*Minho Architecture Description Language*
MIMOLA	*Machine Independent Microprogramming Language*
OSM	*Operation State Machine*
PDA	*Personal Digital Assistant*
PISA	*Portable Instruction Set Architecture*
PLUNGE	*Pipeline Unifying Notation Graphs and Expressions*

RADL	*Retargetable Architecture Description Language*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SIMD	*Single Instruction Multiple Data*
SLED	*Specification Language for Encoding and Decoding*
SoC	*System-on-Chip*
TIC	Tecnologias de Informação e Comunicação
UDL/I	*Unified Design Language for Integrated Circuits*
UFISS	*Ultra-Fast Instruction Set Simulator*
VLIW	Very *Long Instruction Word*
λ-RTL	*Lambda-Register Transfer Lists*

Índice de Figuras

Índice de Tabelas

CAPÍTULO 1 Introdução

1.1 ÂMBITO

As últimas décadas têm proporcionado avanços significativos nas áreas das tecnologias da informação e comunicação (TIC). Estes avanços têm contribuído para que as pessoas, actualmente, possam comunicar entre si de uma forma mais fácil. Estas facilidades de comunicação têm tornado o mundo mais “*plano*” no sentido em que tudo se torna mais próximo. Thomas L. Friedmann no livro “*The World is Flat – A Brief History of the Twenty-First Century*” além de apresentar os marcos que estimularam esta revolução, no capítulo intitulado “*The Steroids*” disserta sobre a importância nos dias de hoje do digital, móvel, pessoal e virtual [1].

Esta realidade é possível devido à cada vez mais facilitada troca de informação na forma digital. Esta por sua vez, faz-se cada vez de forma mais célere permitindo que os interlocutores actuem independentemente da distância que os separa. Ou seja, apesar de estarem geograficamente afastados, a comunicação faz-se como se os mesmos estivessem próximos.

Estas evoluções foram possíveis devido aos avanços que aconteceram na indústria de produção de dispositivos electrónicos programáveis. Estes progressos fizeram com que hoje em dia a electrónica de consumo esteja ao alcance do cidadão comum. É também facilmente constatável que, actualmente, qualquer pessoa, sem se aperceber, lida directamente com dezenas de dispositivos automáticos por dia, exemplo disso são telefones móveis ou PDAs[1]. Estes dispositivos fazem parte do seu dia-a-dia de tal forma, que as mesmas deixaram de notar a sua presença, ou seja tornaram-se ubíquos. A utilização de sistemas embutidos[2] não se esgota no suporte às TIC e está presente num vasto e diversificado universo de aplicações. Saúde, automação, domótica, segurança e aplicações militares são apenas alguns exemplos de áreas onde os sistemas embutidos são usados em inúmeras aplicações, que fazem com que o ambiente onde vivemos seja cada vez mais inteligente [2].

Os sistemas embutidos evoluíram de simples microcontroladores para *Digital Signal Processors* (DSPs), processadores programáveis, processadores reconfiguráveis e *System-on-chip* (SoCs). Estes são caracterizados por serem produzidos massivamente, por fazerem parte de sistemas maiores (i.e. embutidos) e por serem projectados para realizar uma função específica, por vezes com restrições temporais. Assim, o hardware destes dispositivos é geralmente projectado para realizar determinada tarefa e para ser económico. Por seu lado o software não é executável noutros sistemas embutidos sem que sejam promovidas alterações significativas. À medida que as aplicações se tornam cada vez mais complexas, também aumenta a complexidade dos sistemas embutidos. Dependendo das aplicações alvo, os sistemas embutidos podem ter hardware específico, interfaces, controladores ou interfaces dedicadas às funcionalidades pretendidas. Existe uma variabilidade disponível de dispositivos no mercado que abrange arquitecturas de 8 bits até 64 bits, disponibilizados por vários fabricantes. Uma definição mais extensa sobre o que são sistemas embutidos pode ser consultada em [3].

De acordo com [4], os microprocessadores representam apenas 2% do total de semicondutores fabricados. No entanto, esta pequena parcela representa um terço do capital envolvido no negócio de semicondutores. Os computadores pessoais (PCs[3]) representam apenas 2% do número de processadores, ou seja 0,04% do volume de semicondutores vendidos. Conclui-se destes dados que sistemas embutidos além da sua

---

[1] *Personal Digital Assistants*

[2] *Embedded Systems*

[3] *Personal Computers*

diversidade, que depende dos fins a que se destinam, e impacto económico, são cada vez mais sofisticados e consequentemente mais complexos para responder aos requisitos das aplicações a que se destinam.

Um inquérito, feito recentemente por uma reputada revista da área de sistemas embutidos, revelou quais os principais factores tidos em conta (por engenheiros, programadores, investigadores e órgãos de decisão) na escolha de microprocessadores para as aplicações que desenvolvem [4]. Este questionário revela que o principal factor na escolha, são as ferramentas de software e não o desempenho do processador. Esta constatação não é alheia ao facto de o *time-to-market* ser bastante importante para vencer a agressividade do mercado. Estes factores justificam os esforços de projectar ferramentas capazes de diminuir o tempo de desenvolvimento e por outro lado serem redireccionáveis[4], ou seja, permitirem a exploração de diferentes tipos de arquitecturas (RISC[5], CISC[6], VLIW[7], entre outras).

A importância das ferramentas de software reflecte-se não só para quem desenvolve as aplicações que usam os sistemas embutidos, mas também para quem os projecta. Durante as fases de projecto, os projectistas necessitam de produzir um conjunto de ferramentas que os ajude a explorar as melhores soluções.

Durante esta fase de exploração, *Design Space Exploration* (DSE), são estudadas formas de conseguir optimizações de desempenho, consumo de energia, organização da memória ou da qualidade do compilador. Para isso é necessária a geração de várias ferramentas, optimizadas para o processador alvo, tais como: assemblador[8], desassemblador[9], depurador[10], compilador ou simulador.

Estes são os "embriões" das ferramentas que, posteriormente, são disponibilizadas aos projectistas de aplicações envolvendo o processador desenvolvido. O conjunto destas ferramentas designa-se vulgarmente na literatura anglo-saxónica por *software*

---

[4] *Retargetable*

[5] *Reduced Instruction Set Computer*

[6] *Complex Instruction Set Computer*

[7] *Very Long Instruction Word*

[8] *Assembler*

[9] *Disassembler*

[10] *Debugger*

toolkit. Algumas das ferramentas que compõem este conjunto são: simulador, depurador, assemblador, compilador e ligador[11].

Para garantir competitividade são necessárias metodologias que permitam gerar o *software toolkit* com qualidade e num curto espaço de tempo para o sistema embutido em estudo.

A forma tradicional de gerar as ferramentas deixou de ser a mais adequada para esta dinâmica de competição, devido ao período de tempo de desenvolvimento que requer. Assim, a geração a partir de descrições mais abstractas tem vindo a ser adoptada como metodologia para obter ferramentas dedicadas a determinado processador alvo em curtos períodos de desenvolvimento, para melhorar o requisito do *time-to-market*. É neste âmbito que se insere o trabalho desta tese.

1.2 DESENVOLVIMENTO DE FERRAMENTAS DE SOFTWARE PARA SISTEMAS EMBUTIDOS

No caso dos *General Purpose Processors* (GPP) a arquitectura do conjunto de instruções é comum para vários processadores no sentido de manter compatibilidade de *software* por eles executado. No caso dos sistemas embutidos, o desempenho, organização da hierarquia de memória, optimização do software e poupança de energia, são factores primordiais relativamente à garantia de compatibilidade. Este factor faz com que haja grande variabilidade de acordo com as aplicações, ou arquitecturas, destes dispositivos quando comparados com os GPP.

Esta variabilidade exige que as ferramentas de desenvolvimento possam ser adaptadas, ou geradas, para as diferentes variantes com o objectivo de conseguir a solução mais optimizada possível. O que faz com que a reutilização de ferramentas não seja a melhor via para conseguir a solução mais optimizada possível, ou seja, não permite um espaço de exploração amplo. Para que tal seja possível a solução passa por produzir ferramentas dedicadas e optimizadas para o alvo em estudo/projecto. No entanto, esta produção de ferramentas deve cumprir outro dos requisitos principais para os projectistas de sistemas embutidos, ou seja, efectuar-se num curto período de desenvolvimento para garantir *time-to-market* competitivo.

---

[11] *Linker*

À medida que a complexidade dos sistemas embutidos aumenta, a produção de ferramentas à mão deixa de ser viável. Até porque esta metodologia dificulta a possibilidade de alterar a arquitectura do processador durante o processo de desenvolvimento. Assim, são necessárias metodologias que permitam a geração automática de código para vários processadores. Esta característica é vulgarmente designada por redireccionabilidade[12].

Durante a última década vários trabalhos realizados, por vários grupos de investigação, têm proposto infra-estruturas que permitam o desenvolvimento rápido do conjunto de ferramentas de software[13] para determinada família ou famílias de processadores. Estes ambientes facultam ao projectista de sistemas embutidos um DSE adequado à exploração e avaliação de diferentes optimizações durante a fase de projecto. No próximo capítulo desta tese serão apresentados vários trabalhos relacionados com a geração de ferramentas para desenvolvimento de sistemas embutidos. Alguns dos trabalhos permitem flexibilidade através da geração automática do conjunto de ferramentas via descrição do processador usando uma *Architecture Description Language* (ADL), Figura 1.1. O conjunto de ferramentas geralmente é composto por simulador, compilador, assemblador, ligador e outras ferramentas, conforme as características da ADL em causa. Não existe actualmente uma ADL adoptada como padrão, porque nenhuma delas permite descrever todas as arquitecturas do conjunto de instruções e/ou gerar todas as ferramentas. Algumas permitem apenas gerar um conjunto restrito de ferramentas, outras permitem descrever uma família restrita de processadores. Conseguir desenvolver uma ADL que ao mesmo tempo contemple as características de um espectro alargado de arquitecturas de conjuntos de instruções (ISA[14]) e ao mesmo tempo seja capaz de gerar um conjunto de ferramentas completo é uma tarefa complexa devido à vasta gama de variabilidade inerente ao desafio em causa.

---

[12] *Retargetability*

[13] *Software toolkit*

[14] *Instruction-set Architecture*

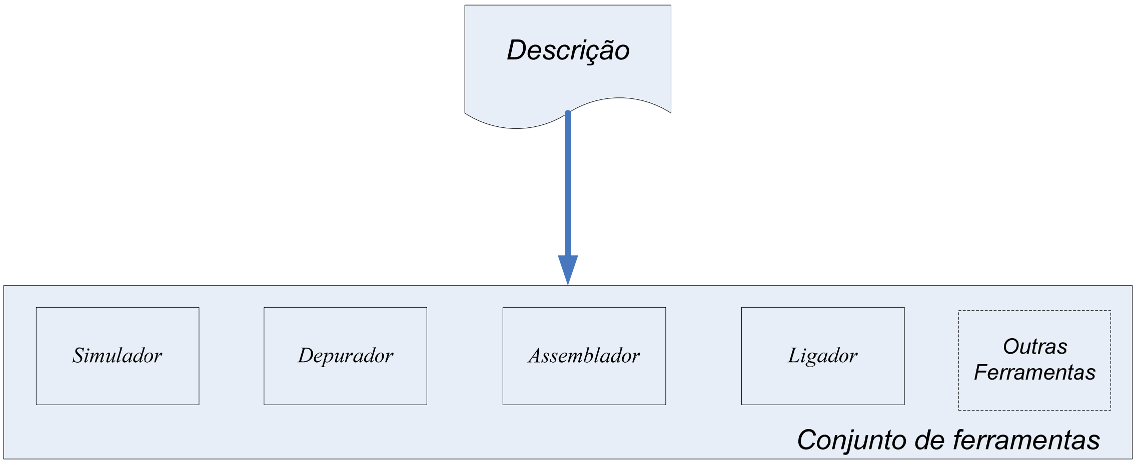


Figura 1.1: Geração Automática de Ferramentas de Software.

1.3 CONTRIBUIÇÕES

Este trabalho contribui com uma ADL que oferece vantagens na descrição relativamente a outras existentes e à qual foi atribuída a designação de ***M**inho* ***A**rchitecture* ***D**escription* ***L**anguage* (MiADL). Para isso foi feito inicialmente um estudo comparativo de vários ISAs e também das ADLs existentes até então. Durante o estudo foram identificadas vantagens e desvantagens daquelas abordagens. Posteriormente foi efectuada a modelação de processadores com ISAs complexos e de processadores usados intensivamente na indústria. Estas modelações permitiram extrair características que foram embebidas na linguagem para que a mesma permita modelar a máxima gama possível de ISAs. O objectivo é a geração automática de simuladores para ISAs. Os simuladores são do tipo *Instruction Set Simulator* (ISS), que são intensivamente utilizados nas fases iniciais do projecto, por a velocidade ser mais relevante do que o detalhe. A geração de compiladores não constitui objectivo do trabalho nesta fase.

As descrições feitas com a linguagem MiADL revelam-se mais compactas e as secções da mesma permitem uma visão global da descrição do ISA do processador que é mais amigável do que as de outras linguagens. Para isso contribui um conjunto de construções[15] que foram projectadas para optimizar as descrições e que permitem que o compilador da linguagem faça um conjunto de verificações que garantem a fiabilidade

---

[15] *Constructs*

das mesmas. Sendo o alvo do trabalho a geração de simuladores ISS, as descrições reportam ao ponto de vista do programador, i.e. comportamento das instruções e respectivos formatos.

Das características da linguagem destaca-se também a hierarquia de *scopes* que é uma das contribuições científicas do trabalho para a área de ADLs, bem como a inferência de argumentos de funções. Estas duas características permitem por um lado detectar erros e por outro simplificar as descrições.

As características da linguagem permitem também recorrer de forma simples a *generic programming*, neste caso usando *function templates*, para executar o comportamento das instruções. Esta é também uma das contribuições desta linguagem em relação às demais.

Além das vantagens já mencionadas da linguagem, esta possui construções próprias e uma estrutura que permite conseguir representações muito compactas do *assembly* das instruções, comparativamente a outros trabalhos, sendo também uma das contribuições do trabalho.

Quanto ao tipo de simulador gerado, actualmente estes são do tipo compilado, estando no entanto em aberto a geração de simuladores do tipo interpretado, tendo por base as mesmas descrições.

Os resultados obtidos com simuladores MiADL demostram que o desempenho dos mesmos é superior a outras abordagens. Assim, além de uma descrição compacta usando MiADL, também é conseguida a geração de simuladores rápidos o que contribui para um curto *time-to-market*.

1.4 ORGANIZAÇÃO DA TESE

Esta tese está organizada em seis capítulos, cujo conteúdo dos restantes cinco está organizado da seguinte forma:

- No ***Capítulo 2***, são apresentadas diferentes abordagens relacionadas com a simulação de arquitecturas do conjunto de instruções. São apresentados exemplos de simuladores que recorrem a cada uma delas. São também apresentados os requisitos desejados para os simuladores tais como velocidade, redireccionabilidade e detalhe. Na segunda parte do capítulo é feita uma apresentação de várias ADLs, as quais estão agrupadas de acordo o *focus* da descrição (comportamento, estrutura ou ambos). Por fim, são tecidas conclusões gerais relativamente aos tópicos abordados neste capítulo.

- No ***Capítulo 3***, é apresentada a nova linguagem, MiADL. Ao longo do capítulo são apresentadas as secções que compõem a linguagem bem como as suas construções. São também descritas as verificações que são feitas, pelo compilador da linguagem, para garantia de consistência das descrições e a hierarquia de *scopes* a que a mesma obriga. São usados excertos de processadores conhecidos que foram modelados em MiADL, para exemplificar as características da linguagem.

- No ***Capítulo 4***, é apresentada a infra-estrutura de geração de simuladores a partir de descrições em MiADL. Durante a explicação da infra-estrutura, são descritos os blocos principais da mesma e a forma como interagem entre si.

- No ***Capítulo 5***, são apresentados exemplos de casos particulares do conjunto de instruções de processadores modelados em MiADL, e como a linguagem os permite descrever. É comparada a eficiência das descrições com outros trabalhos. Neste capítulo são também apresentados resultados de desempenho e é feita uma comparação dos mesmos com outros trabalhos.

- No ***Capítulo 6***, é feito um resumo e são apresentadas as contribuições do trabalho bem como apresentadas algumas das linhas de continuidade do trabalho para o futuro.

Após os seis capítulos, a tese possui uma **lista de referências bibliográficas**, mencionadas ao longo do documento, que fundamentam os assuntos tratados ao longo do texto da tese.

No final a tese termina com um **glossário** de termos e um conjunto de cinco apêndices/anexos, que servem de adenda a informação contida nos capítulos, cujo conteúdo é o seguinte:

Anexo 1 – Gramática da linguagem MiADL: Este anexo apresenta a gramática da linguagem MiADL que é apresentada e discutida no terceiro capítulo da tese. Para a representação da gramática é usada a notação *Backus-Naur Form* (BNF).

Anexo 2 – Tabela de operadores da linguagem MiADL.

Anexo 3 – Modelação do processador SPARC: Este apêndice da tese apresenta o modelo do processador SPARC, versão 8, conforme descrito usando a sintaxe da linguagem MiADL que é apresentada na tese.

Anexo 4 – Modelação do processador ARM: Este apêndice da tese apresenta o modelo do processador ARM, conforme descrito usando a sintaxe da linguagem MiADL que é apresentada na tese.

Anexo 5 – Exemplos de modelações usados para clarificar texto escrito ao longo dos capítulos.

1.5 NOTAÇÃO UTILIZADA

O formato usado para referenciar documentos electrónicos baseia-se no standard ISO690-2 [5] que é dedicado àquele género de referências bibliográficas.

Como nota de rodapé é mencionada a versão inglesa de alguns termos técnicos, comuns na área deste trabalho. Assim, a primeira vez que surge necessidade de usar o termo técnico, no texto da tese, é colocada a designação adoptada em português e, em nota de rodapé, o correspondente sinónimo em inglês (como no exemplo da nota 16 presente neste parágrafo). Esta notação assume importância numa área onde os termos técnicos em inglês abundam e por vezes torna-se difícil relacionar traduções directas entre os dois idiomas.

Ao longo do texto os termos relacionados com estrangeirismos, são distinguidos do resto do texto pelo uso de itálico.

Para diferenciar alguns nomes, ou construções da linguagem, será usado o itálico em conjugação com negrito para realçar tais identificadores do demais texto onde venham a ser usados.

Para a numeração das figuras adoptou-se o formato segundo o qual as mesmas são numeradas por: número_de_capítulo.número_da_figura. Desta forma ao consultar a lista de figuras que é apresentada após o índice geral, é possível pela numeração de determinada figura deduzir o capítulo onde a mesma se encontra.

A tese contém um glossário que se destina a apresentar o significado de diversos termos usados na tese, mas cuja definição não consta nos capítulos da mesma. Consideramos também que alguns termos são do conhecimento geral (fazendo parte do dicionário da língua portuguesa) e que não faria sentido voltar a apresentar a sua definição, tais como por exemplo bit, byte, entre outros.

O formato do documento segue as normas recomendadas pelo Despacho RT-32/2005, Normas para a Formatação das Teses de Mestrado e de Doutoramento da Universidade do Minho, e o Manual de Identidade Gráfica para as Capas de Tese da Universidade do Minho.

1.6 EXTENSÃO WEB

Documentação adicional será disponibilizada, numa página da Internet, no sentido de divulgar o trabalho e possibilitar a obtenção de retorno da comunidade relacionada com a área do trabalho. Entre a documentação a disponibilizar prevê-se um manual da linguagem, modelos de processadores, publicações, relatórios técnicos, extensões à linguagem que venham a ser introduzidas e ferramentas desenvolvidas. O domínio reservado para o efeito tem o endereço http://www.miadl.org. Pretende-se que através da divulgação do trabalho perante a comunidade científica seja possível conseguir melhorá-lo e expandi-lo com novas ferramentas.

CAPÍTULO

2

Estado da Arte

2.1 INTRODUÇÃO

Este capítulo da tese está dividido em duas partes distintas, mas afins. Na primeira parte do capítulo, é feita uma introdução sobre **simulação** de arquitecturas de processadores. São descritas técnicas de simulação e também simuladores conhecidos expondo as características e diferenças entre os mesmos.

Na segunda parte do capítulo, na sequência da parte anterior, são apresentadas diversas **linguagens de descrição de arquitecturas de processadores**, bem como as características relacionadas com as infra-estruturas com elas relacionadas. São abordados os trabalhos considerados mais relevantes tendo em atenção a área do trabalho desta tese.

2.2 SIMULAÇÃO DE ARQUITECTURA DO CONJUNTO DE INSTRUÇÕES

A simulação, no âmbito do projecto de sistemas embutidos, consiste numa ferramenta que corre numa máquina hospedeira e que imita o comportamento correspondente à execução de uma aplicação numa máquina alvo (i.e. que está a ser simulada). Estas ferramentas são importantíssimas durante a fase de exploração, na qual são usadas para avaliar, ajudar a tomar decisões relativas ao projecto de novas arquitecturas, e também para validar o projecto de compiladores. Com base neste retorno de informação, é possível efectuar melhoria do projecto e assim atingir mais rapidamente a solução desejada, mesmo antes de ter disponível o protótipo de hardware, o que propícia benefícios económicos e maior flexibilidade. Isto porque, para além de possibilitarem executar código para hardware que não existe, permitem acesso a estados internos que podem ser inacessíveis em hardware real, e podem permitir executar testes exaustivos que são impossíveis em hardware. Dependendo da sua finalidade, estas ferramentas podem ser utilizadas para verificar a funcionalidade (ex. durante o desenvolvimento dum compilador) ou comportamento temporal do sistema (hardware e software) em desenvolvimento, e fazer análise quantitativa de alguns parâmetros tais como, por exemplo, o consumo de energia. Os simuladores também são empregues para depurar aplicações de software. O tipo de simulação está relacionado com o detalhe necessário para determinada fase do projecto. O detalhe envolvido, afecta a velocidade de simulação de forma negativa.

Tratando-se de sistemas embutidos que, como foi abordado no capítulo anterior, requerem que, na fase de projecto, sejam exploradas diversas alternativas em curtos ciclos de desenvolvimento, o simulador permite simplificar o processo se também possuir a característica de poder ser rapidamente adaptado para vários tipos de arquitecturas diferentes. Ou seja, ser redireccionável. Se no passado o desempenho era o parâmetro mais importante na distinção entre simuladores, a redireccionabilidade é actualmente também considerado um factor bastante importante, nomeadamente na área do desenvolvimento de sistemas embutidos onde é grande a diversidade de arquitecturas.

A capacidade de ser redireccionável ganha especial relevo quando cada vez mais existe em termos de fabrico a necessidade de fornecer, para o mercado, uma gama de produtos/serviços segundo o conceito de linhas de produção de software[16] [6]. Este conceito está relacionado com estratégias de oferecer aos destinatários vários produtos relacionados entre si. Neste caso, refere-se à existência de várias ferramentas de software (assembladores, desassembladores, compiladores, ligadores, simuladores, editores, entre outras) produtos para determinada arquitectura, ou família de

---

[16] *Software Product Lines*

arquitecturas de processadores. Ao ter a capacidade de ser redireccionável a estrutura de desenvolvimento de ferramentas permite ser mais célere, curto *time-to-market*, na geração de diversas linhas de produtos e assim tornar a oferta mais competitiva num mercado agressivo como é o dos sistemas embutidos.

Em suma, no que diz respeito aos simuladores, a **velocidade de simulação**, o **detalhe** com que são capazes de reproduzir os alvos que pretendem simular (i.e., abstracção), e a capacidade com que são capazes de testar novas alternativas de arquitecturas (i.e., serem **redireccionáveis**) são três das características mais importantes para os projectistas de sistemas embutidos. Nas secções seguintes serão abordadas com detalhe estas três vertentes, apresentando classificações, trabalhos, e técnicas que com elas estão relacionada(o)s.

2.2.1 DETALHE DE SIMULAÇÃO

Um dos factores importantes no desenvolvimento de arquitecturas é a capacidade de definir vários níveis de abstracção. No entanto, é necessário que o conjunto/ambiente de ferramentas de desenvolvimento possa ser gerado independentemente do nível de abstracção. Especialmente na fase de exploração, isto assume elevada importância para tornar os ciclos de desenvolvimento curtos, mantendo os modelos e ferramentas consistentes. De acordo com [7], a abstracção pode ser entendida em dois domínios distintos mas relacionados: arquitectural e temporal.

Abstracção arquitectural: Esta abstracção está relacionada com a granularidade segundo a qual a estrutura e comportamento são descritos. Em fases iniciais do projecto, o desenvolvimento da arquitectura começa com uma especificação funcional do algoritmo que deve ser mapeado na arquitectura do processador alvo, sem ter em conta o detalhe temporal. Trata-se de um nível de puro fluxo de dados na arquitectura alvo.

Quando aquele modelo de fluxo de dados está verificado e cumpre os requisitos, é refinado para o nível do conjunto de instruções. Para este propósito a granularidade de hardware e comportamento são enriquecidos com a adição de memória de programa, memória de dados e sequenciador de instruções. Este modelo pode ser depois usado para verificar a arquitectura do conjunto de instruções através da análise de aplicações que são executadas na arquitectura alvo. É a este nível que os projectistas de compiladores validam os compiladores e aplicações. Após esta fase, mais detalhe pode ser adicionado através da inclusão de pormenores de hardware tais como *pipelines*, memória *cache*, interrupções, etc. A este nível de representação da microarquitectura, o modelo possui detalhe suficiente sobre a estrutura e sobre o comportamento para permitir simular ou até mesmo sintetizar estruturas de hardware.

Abstracção temporal: A precisão temporal necessita também de ser modificada ao longo das fases de projecto. Em fases iniciais, pode estar confinada a *statements* (i.e., instruções) do compilador que permitem estimar o tempo necessário para executar partes da aplicação. Nesta fase é possível, por exemplo, verificar se restrições de tempo real podem ser conseguidas. O detalhe temporal, para efeitos de simulação, também pode ser refinado gradualmente com inclusão de detalhe de tempo de instrução, detalhe de ciclos de relógio (*cycle-accurate*) ou mesmo detalhe ao nível de porções de ciclo (*phase-accurate*).

Quando o nível de detalhe aumenta - menos abstracção - a velocidade de simulação diminui. Assim, tipicamente ao nível de abstracção funcional (simulador de conjunto de instruções - ISS) a velocidade conseguida é bastante superior ao que sucede em simuladores com precisão de ciclos (*cycle-accurate*) [8] [9]. Durante o DSE, em estágios iniciais do projecto exigem-se níveis baixos de abstracção e em fases posteriores quando testes mais exaustivos são necessários as ferramentas devem permitir informação mais detalhada. Este detalhe, no caso da simulação, pode atingir o nível do ciclo de relógio (*cycle-accurate*), fases de ciclo de relógio (*phase-acurate*) ou até mesmo ao nível da porta lógica[17] (*gate-level*), no qual são tratados detalhes relacionados com propagação de sinais e temporização das portas lógicas e ligações entre elas. A Figura 2.1 retrata o fluxo de projecto em estágios iniciais de desenvolvimento, para o caso da elaboração de simuladores.

---

[17] *Gate*

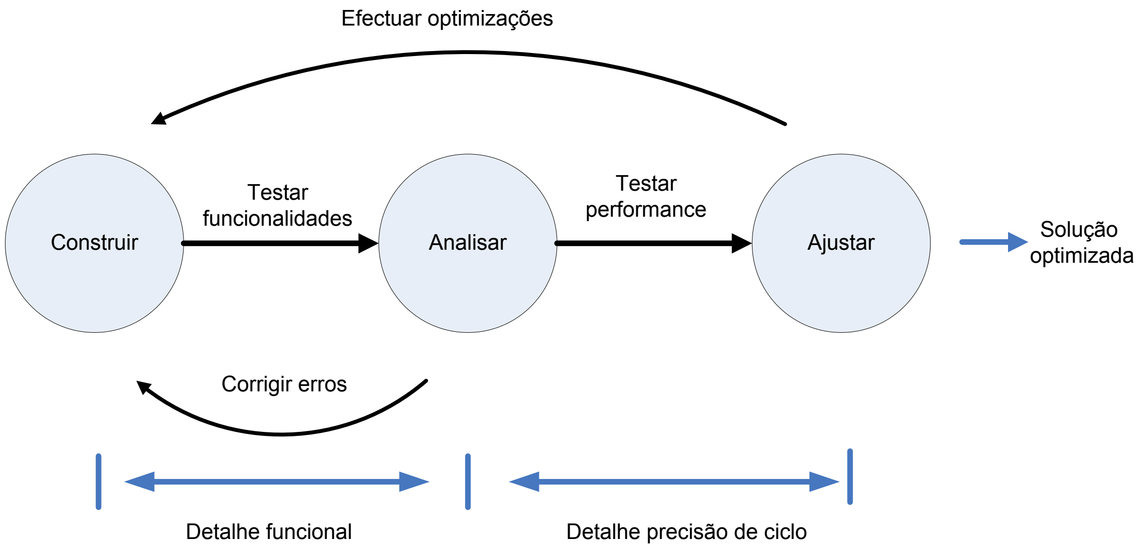


Figura 2.1: Detalhe de simulação ao longo do ciclo de desenvolvimento

2.2.2 TÉCNICAS DE SIMULAÇÃO

Nesta secção apresenta-se uma classificação dos simuladores de acordo com a forma como é implementada a simulação em termos de modelo de execução. As três técnicas conhecidas são a simulação interpretada, a compilada e a híbrida, conforme se apresenta nas secções seguintes.

2.2.2.1 INTERPRETADA

Este tipo de simulação é o mais tradicional. O estado do processador alvo (o que está a ser simulado) é guardado na memória do processador hospedeiro (onde se realiza a simulação). Os passos durante a fase de execução da simulação seguem uma sequência idêntica à do hardware [7], ou seja:

- carregar uma instrução (*Fetch* - leitura duma palavra de instrução a partir da memória que possui a aplicação a simular), a partir da memória do hospedeiro;
- descodificação da informação obtida no passo anterior para determinar que operação deve ser executada;
- execução (actualização do estado do processador), através da chamada da rotina correspondente ao comportamento da operação decifrada no passo anterior.

Após a sequência destas três etapas, o ciclo repete-se para cada instrução guardada na memória de instruções, até acontecer a condição de paragem do programa (ex. número de instruções, suceder um erro, acontecer uma excepção, entre outras condições). Na Figura 2.2, representam-se os estágios principais de um simulador deste tipo.

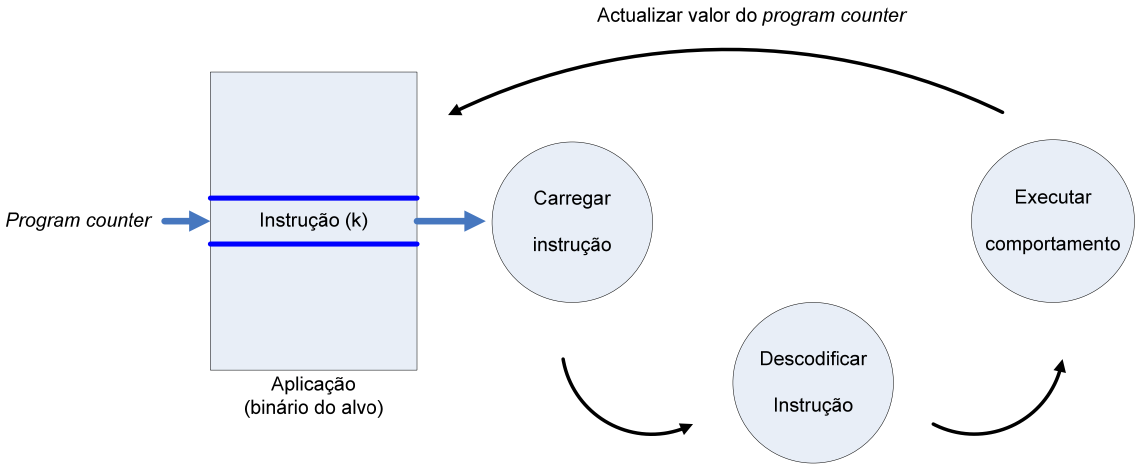


Figura 2.2: Simulação Interpretada

Os simuladores interpretados permitem flexibilidade mas, por outro lado, apresentam como limitação o facto de serem lentos, quando comparados com outros tipos de simulação [10]. A flexibilidade advém da possibilidade de a aplicação a executar no alvo poder sofrer alterações durante o tempo de execução. O défice de desempenho deve-se ao facto de nestes simuladores ter de ser gasto tempo nas múltiplas descodificações para a mesma instrução, desaproveitando a existência de elevada localidade de código de algumas aplicações [7]. No entanto, estes simuladores são os mais usados no desenvolvimento de sistemas embutidos. O seu uso é aplicado na validação, ou para avaliar decisões, durante o projecto de novas arquitecturas, durante o DSE.

Em suma, estes simuladores são máquinas virtuais implementadas em software, que interpretam código objecto (binário do alvo) e que executam acções correspondentes no hospedeiro, para cada instrução descodificada, como ilustrado na Figura 2.2.

Tendo em conta a necessidade de ciclos rápidos de desenvolvimento, devido à pressão do *time-to-market*, foram propostas técnicas alternativas para melhorar a velocidade relativamente a esta técnica. Uma das técnicas é a simulação compilada, que será explicada na secção seguinte.

Exemplos de simuladores que seguem este modelo, são o SimpleScalar [11] e MicroLib [12]. Estes, e outros, simuladores são abordados na secção 2.2.2.4.

2.2.2.2 COMPILADA

Usando esta técnica de simulação, conseguem-se velocidades superiores à técnica anterior através da criação de um simulador optimizado para a aplicação que será executada. Ou seja, é adicionado um estágio prévio à execução da simulação. Neste estágio prévio, antes da compilação, a etapa de descodificação é realizada. É assumido que a aplicação não sofre modificações durante a fase de execução.

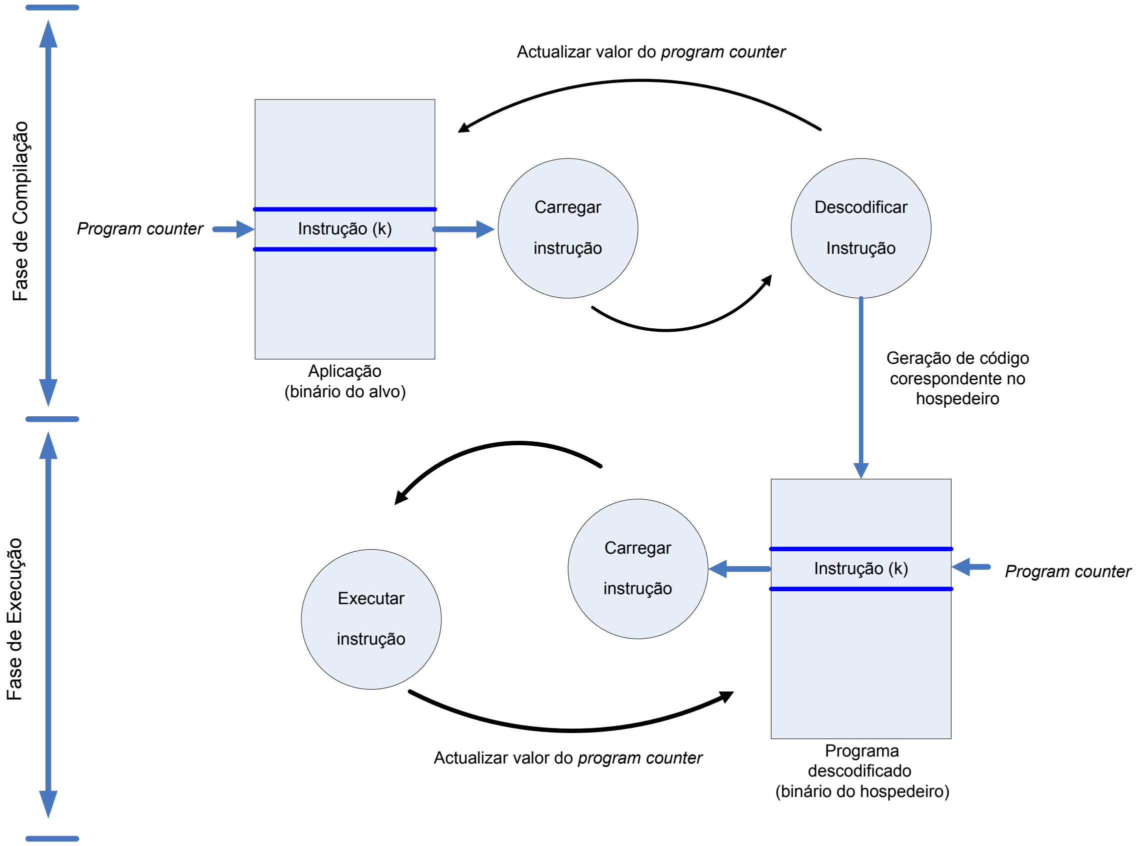


Figura 2.3: Simulação Compilada – Estática

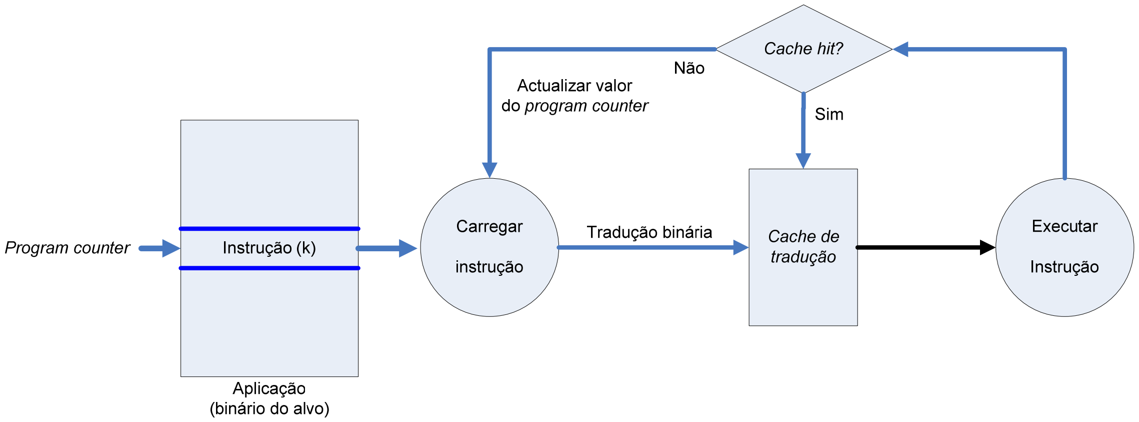


Figura 2.4: Simulação Compilada – Dinâmica

Cada instrução do programa alvo é convertida em instruções do hospedeiro, que por sua vez modificam o estado do processador simulado. Conforme o instante em que é feita esta conversão, assim os simuladores são geralmente classificados como:

- **Compilação estática**: Caracteriza-se por a conversão ocorrer para a totalidade das instruções durante o tempo de compilação, optimizando a sequência carregar-descodificar-executar. A Figura 2.3 pretende ilustrar os estágios principais de um simulador deste tipo. Quanto à conversão para código binário hospedeiro, esta pode ser feita através duma linguagem intermédia (ex. C) ou directamente (exigindo neste caso um compilador especifico para o hospedeiro). O recurso à representação numa linguagem intermédia, pode tornar a simulação mais lenta mas por outro lado também a torna mais portável [13, 14].
- **Compilação dinâmica**: Caracteriza-se por a conversão ocorrer na altura de carregar a instrução, optimizando tempos gastos em repetições de execução de instruções iguais, através de recurso a uma cache. A Figura 2.4 representa os estágios principais de um simulador deste tipo. A velocidade alcançada, usando esta técnica, é inferior à conseguida com a compilação estática.

Estes simuladores são mais rápidos do que os que usam a técnica de simulação interpretada [15], mas por outro lado são menos flexíveis porque assumem que o código do programa a simular não sofre alterações durante tempo de execução.

Exemplos de simuladores que seguem o modelo de simulação compilada estática, são [16] e [13]. Quanto ao modelo de simulação compilada dinâmica, servem de exemplo os simuladores Shade [17] e Embra [18]. Estes simuladores são abordados na secção 2.2.2.4.

2.2.2.3 HÍBRIDA

Para melhorar a velocidade dos simuladores interpretados mas sem comprometer a flexibilidade, foram propostas técnicas que combinam as vantagens dos simuladores interpretados (i.e. flexibilidade) e compilados (i.e. velocidade). Em vez de ser criado código nativo para simular a instrução, estas técnicas armazenam numa *cache* a informação descodificada relativa à instrução. O custo da descodificação é reduzido no caso de ciclos de instruções. Por outro lado, a infra-estrutura continua a ser do tipo interpretada, mantendo flexibilidade [19].

Duas técnicas que foram propostas são:

- *Just-in-time Cache Compiled Simulation* (JIT-CCS) [20]: Nesta técnica as instruções são compiladas durante o tempo de execução, antes de serem executadas. Posteriormente, a informação é armazenada numa cache de simulação para reutilização rápida, no caso de repetição daquele endereço de programa. O simulador verifica se o código do programa de um endereço utilizado anteriormente sofreu alterações e inicia a recompilação, caso tal se verifique.
- *Instruction-Set Compiled Simulation* (IS-CS) [21]: Nesta técnica o programa é compilado e posteriormente executado de forma interpretada. Em tempo de execução, sempre que o simulador detecta que o código do programa em determinado endereço, já executado previamente, sofreu alterações, reinicia novo processo de descodificação para aquele endereço. O novo valor descodificado é armazenado na cache que guarda valores descodificados. São anunciados ganhos significativos em relação à técnica JIT-CCS.

Ambas as técnicas foram usadas no âmbito de infra-estruturas de simuladores gerados a partir de descrições feitas em linguagens projectadas para o efeito. A técnica JIT-CCS é aplicada no âmbito dos trabalhos apresentados na secção 2.4.3.1. Quanto à simulação IS-CS, é aplicada no âmbito dos trabalhos apresentados na secção 2.4.3.3. Em ambos os caso a flexibilidade introduzida é, no entanto, diminuída em termos de desempenho quando comparada com técnicas abordadas na subsecção 2.2.2.2 [22].

2.2.2.4 SIMULADORES

Existe uma gama vasta de simuladores que contemplam diferentes objectivos de simulação. Assim, existem por exemplo simuladores vocacionados para a simulação de sistemas completos (ex. Simics [23]), vocacionados para simulação de multi-processadores (ex. RSIM [24]), etc. Apresentam-se de seguida simuladores bastante referenciados e que usam técnicas de simulação anteriormente apresentadas. Estes simuladores, tal como no caso deste trabalho, apresentam em comum a característica de serem aplicados à simulação de processadores uni-*core*.

SIMPLESCALAR

SimpleScalar [25] é um conjunto de ferramentas de software, criado por Todd Austin, University of Wisconsin in Madison, que compõem uma infra-estrutura para modelação e simulação de arquitecturas. Foi divulgado pela primeira vez em 1995 e a ultima versão 3.0, foi disponibilizada em 1998. É possível com o SimpleScalar, emular uma variedade predefinida de arquitecturas de conjuntos de instruções (Alpha, PISA[18], ARM e x86) [26].

SimpleScalar comporta simuladores e ferramentas (compilador *gcc* [27], bibliotecas, assemblador e ligador, entre outras) que suportam os simuladores. Os simuladores de SimpleScalar permitem diferentes níveis de detalhe e velocidades. Assim, esta gama de simuladores contempla desde simuladores funcionais rápidos, sem modelar detalhe temporal (*sim-fast, sim-safe*), até um simulador que contempla precisão de ciclos e possibilidade de simular processador com execução fora de ordem com uma *pipeline* de cinco estágios (*sim-outorder*) [25].

A ferramenta SimpleScalar usa a técnica de simulação interpretada o que a torna lenta quando comparada com outras propostas. No entanto, apesar de ser uma das ferramentas de simulação mais usadas na área de ensino e investigação de arquitectura de computadores, a sua capacidade de ser redireccionável, através dum mecanismo de

---

[18] *Portable Instruction Set Architecture*

definição do conjunto de instruções no qual o comportamento das mesmas é especificado na forma de macros em *C* [28], está limitada às directivas da arquitectura MIPS [29].

SHADE

Este simulador foi o primeiro a introduzir a técnica de simulação compilada dinâmica [30].

Shade realiza compilação cruzada[19] de porções dos executáveis fonte e executa-as directamente no processador hospedeiro, guardando conversões para reutilização com o objectivo de amortizar o custo de compilação [31]. Este simulador apresenta como limitações, devido ao recurso à geração directa de código binário do hospedeiro, a complexidade e limitação na portabilidade e redireccionabilidade. Relativamente a arquitecturas simuladas, publicadas, elas restringem-se a SPARC (versão 8 e versão 9) e MIPS I correndo em hospedeiro SPARC. Actualmente o simulador é propriedade da *Sun Microsystems, Inc.* [32].

EMBRA

O projecto Embra [18] inspirou-se no Shade e apresenta um simulador de elevado desempenho para processadores MIPS R3000/4000 que é conseguido recorrendo à técnica de simulação compilada dinâmica. Esta contribuição revela como desvantagem a ausência de redireccionabilidade. O simulador foi usado para teste e desenvolvimento de sistemas operativos e para estudos de arquitectura de computadores.

UFISS

Jianwen Zhu e Daniel D. Gajski [16] apresentaram uma técnica designada por *Ultra-Fast Instruction Set Simulator* (UFISS), que veio permitir melhorar a velocidade de simulação através do uso optimizado dos recursos do processador hospedeiro. A técnica proposta por aqueles autores consiste na definição de uma interface de código de baixo nível própria para a simulação da arquitectura do conjunto de instruções, em alternativa à forma tradicional de recorrer a uma linguagem de alto nível, ex. C, como interface na geração de código. Apesar do maior desempenho, em relação a outras

---

[19] *Cross-compile*

técnicas de simulação compilada estática, esta técnica exige que o processador hospedeiro seja possuidor de um vasto conjunto de registos. Por outro lado, é ignorada a *endianess* entre alvo e hospedeiro.

Conforme foi referido no início desta secção, existe uma vasta gama de simuladores disponíveis e o propósito foi o de escolher aqueles que são os considerados mais representativos, de acordo com os objectivos do trabalho. Uma lista de outros simuladores pode ser consultada em [33].

Apesar dos esforços para acelerar a execução dos simuladores, os trabalhos referidos nesta secção apresentam limitação em termos de flexibilidade de redireccionamento.

Desenvolver simuladores manualmente é um processo moroso e restrito a determinada família de processadores. Actualmente esta forma de geração não é tolerável, como vimos na introdução, por ser fundamental explorar novas arquitecturas em curtos períodos de tempo. Assim, torna-se necessário automatizar e tornar redireccionável o processo de geração de simuladores, mantendo como objectivos também a velocidade de simulação e a capacidade de suportar diferentes níveis de abstracção. Nas secções seguintes abordam-se trabalhos que foram propostos para conseguir atingir estes objectivos.

2.3 NECESSIDADE DE AUTOMATIZAR GERAÇÃO DE FERRAMENTAS

A pressão do *time-to-market,* aliada à cada vez mais fugaz longevidade dos produtos na área dos sistemas embutidos, justifica a necessidade de automatizar o mais possível o desenvolvimento de processadores. Assim, para colmatar esta necessidade, a estratégia que tem vindo a ser proposta é a de modelar o processador usando ADLs. A especificação expressa nestas linguagens é usada posteriormente como fonte para a geração automática, ou semi-automática, de várias ferramentas de software (simuladores, compiladores, assembladores, ligadores, entre outras) que permitem avaliar e explorar possíveis arquitecturas até ser conseguida uma solução adequada à aplicação em projecto. Durante esta fase de exploração o objectivo é conseguir projectar a melhor arquitectura possível para os objectivos a que o processador terá que responder. Alguns dos requisitos a ter em conta durante a fase de projecto são o custo, área de silício, consumo de energia ou performance, entre outros. Conforme foi referido no capítulo anterior desta tese a qualidade das ferramentas de software também é de fulcral importância. Assim, as características intrínsecas das linguagens fazem com que as mesmas possam ser classificadas de acordo com o fim a que se destinam. De seguida apresenta-se o estado da arte em termos de linguagens de descrição de arquitecturas de processadores.

2.4 LINGUAGENS PARA DESCRIÇÃO DE ARQUITECTURAS DE PROCESSADORES

As linguagens de descrição de arquitecturas de processadores têm sido alvo de análise por diferentes autores e classificadas de acordo com diferentes pontos de vista. Halambi em [34] classifica em três categorias as linguagens usadas para validação e exploração de arquitecturas de SoCs: *behavioral-centric* (i.e. focadas no comportamento), *structure-centric* (i.e. focadas em detalhes da estrutura) e *mixed* (i.e. focadas na estrutura e comportamento). Por sua vez, Tomiyama em [35] classifica as linguagens em cinco categorias, de acordo com o seu objectivo principal: vocacionadas para síntese, geração de compiladores, geração de simuladores, validação ou vocacionadas para outros fins. Mais recentemente Mishra em [36] apresenta uma taxonomia deste tipo de linguagens e classifica-as quanto a informação que as mesmas possuem (ponto de vista idêntico ao de Halambi) e quanto às ferramentas que estas são capazes de permitir gerar a partir delas (ponto de vista idêntico ao de Tomiyama). Outra classificação interessante é apresentada por Wei Qin em [37], onde analisa modelos de computação usados nos *frameworks* dos quais as linguagens fazem parte, e o mesmo autor em [38] apresenta o ponto de vista em termos de *redireccionabilidade*, para o caso de compilação, de várias ADLs. A classificação inicialmente proposta por Halambi será a seguida neste texto. Em cada uma das linguagens abordadas, será indicado para que tipo de geração de ferramentas tem sido mais utilizada, uma vez que o *focus* deste trabalho está relacionado com a geração de simuladores.

2.4.1 FOCADAS NA ESTRUTURA

Esta classe de linguagens contempla as que em termos de abstracção se posicionam ao nível de descrições *Register Transfer Level* (RTL). Este nível de abstracção esconde detalhes ao nível da *gate* mas expressa informação ao nível da transferência de dados entre registos. Assim, estas providenciam um ponto de vista em termos de blocos que compõem o sistema, ou *net-lists*.

Este tipo de linguagens permite a descrição precisa da microarquitectura. No entanto, torna-se difícil extrair o *instruction-set* para geração de código. Ou seja, são vocacionadas primordialmente para síntese de hardware e não para a geração de ferramentas de software redireccionáveis. No caso concreto da simulação, esta baixa abstracção faz com que os simuladores sejam lentos devido ao nível de detalhe envolvido. Assim, para se conseguirem velocidades superiores é necessária maior abstracção.

Abordam-se de seguida, de forma resumida, linguagens geralmente classificadas como focadas na estrutura: MIMOLA, UDL/I e algumas linguagens de descrição de hardware.

2.4.1.1 MIMOLA

Esta linguagem, desenvolvida na Universidade de Dortmund – Alemanha, foi inicialmente proposta para projecto de microarquitecturas [39, 40]. Um elogio feito a esta linguagem advém do facto de uma descrição poder ser usada para síntese, simulação, geração de teste e compilação [36]. Esta característica é conseguida graças a um conjunto de ferramentas do qual faz parte: um sintetizador de hardware (MSSH), um compilador (MSSQ), compilador de programas de auto-teste (MSST), um simulador funcional (MSSB) e um simulador *event-driven* ao nível RTL (MSSU) [41]. Em *Machine Independent Microprogramming Language* (MIMOLA) as descrições são compostas por três componentes: o algoritmo a ser compilado, o modelo do processador alvo, e regras adicionais de *linkagem* e transformação. A modelação dum processador alvo é efectuada na forma de uma *net-list* de módulos, os quais são descritos ao nível RTL. A descrição do algoritmo, aplicação, é feita usando uma sintaxe similar à da linguagem PASCAL [42]. A informação de *linkagem* é usada pelo compilador para localizar módulos importantes tais como o *program counter* ou memória de instruções. Não existe informação explícita acerca do conjunto de instruções ou *assembly* numa descrição usando MIMOLA [35]. A informação do conjunto de instruções, que é necessária para geração de código, tem de ser extraída a partir da descrição estrutural através de duas estruturas de dados internas do MSSQ (*Connection Operation Graph*

(COG) e *instruction trees* (*I-trees*)) [41]. MIMOLA não suporta geração automática de simuladores, mas descrições em MIMOLA podem ser simuladas usando os simuladores MSSB ou MSSU.

Em desfavor do uso da linguagem para exploração de arquitecturas está a baixa abstracção que a torna ineficaz, porque em estágios iniciais de projecto, nos quais várias alternativas devem poder ser avaliadas, a implementação da microarquitectura não é relevante. Modificações em descrições com este nível de pormenor são trabalhosas e morosas, o que as torna inviáveis devido à necessidade de *time-to-market* céleres.

Quanto à simulação, devido ao nível de abstracção envolvido, estas descrições não são adequadas para a geração de simuladores com desempenhos comparáveis aos conseguidos com outras abordagens [7]. No entanto, em [43] é apresentado o sistema JACOB que gera simuladores interpretados e compilados a partir de informação extraída de descrições MIMOLA.

2.4.1.2 UDL/I

A *Unified Design Language for Integrated Circuits* (UDL/I), foi proposta na Universidade de Kyushu, Japão [44]. Esta linguagem foi desenvolvida para geração de compiladores no âmbito da infra-estrutura COACH para desenvolvimento de *Application-specific Instruction-set Processors* (ASIPs). Esta linguagem é usada para descrever processadores ao nível RTL, com detalhe de ciclo. A informação sobre o conjunto de instruções do processador alvo é extraída automaticamente a partir da descrição da linguagem, e usada para geração de simulador ISS e compilador [45]. Esta não permite, no entanto, extrair informação do conjunto de instruções para arquitecturas VLIW ou superscalar. A infra-estrutura COACH assume processadores RISC e não permite explicitar *pipelines* de processadores. As descrições efectuadas em UDL/I são sintetizáveis [46]. Também no caso da infra-estrutura COACH, como no caso de MIMOLA, um dos principais pontos positivos é a possibilidade de geração, a partir duma descrição, de ferramentas para síntese, simulação e compilação. Para este fim é exigida a marcação de elementos que identificam importantes estados da máquina tais como o *program counter* e bancos de registos.

A linguagem UDL/I apesar dos aspectos positivos que possui, apresenta a desvantagem característica desta classe de linguagens, ou seja excesso de detalhe, o que a torna também ineficiente para exploração de várias arquitecturas durante as fases iniciais de projecto de sistemas embutidos.

2.4.1.3 LINGUAGENS DE DESCRIÇÃO DE HARDWARE

Além das duas linguagens anteriormente abordadas, existem outras que se enquadram nesta classe: VHDL [47], Verilog [48], M [49] e mais recentemente o SystemC [50]. Em [51, 52] é feita uma análise comparativa entre algumas destas linguagens.

Estas linguagens de descrição de hardware[20] (HDL) são usadas para modelar e simular processadores, com o propósito de modelar hardware. O facto de possuírem demasiado detalhe sobre implementação de hardware torna-as desvantajosas para simulação ao nível do *instruction-set*. Isto advém da inadequação daquele nível de detalhe para avaliar performance e verificação do software, acresce a limitação de que a descrição de estruturas de hardware atenua a velocidade de simulação [53]. Uma desvantagem destas linguagens, para geração de ferramentas de software tais como assemblador e desassemblador, é o facto de não ser possível a extracção do *assembly*.

2.4.1.4 CARACTERÍSTICAS E LIMITAÇÕES

Nestas linguagens o excesso de detalhe, baixa abstracção, torna-as verbosas e trabalhosas. Efectuar a exploração de novas soluções torna-se um processo demorado. Por outro lado, algumas características necessárias para geração de algumas ferramentas de software não são possíveis, como por exemplo o *assembly*. Outra limitação advém do facto de a descrição detalhada da estrutura do hardware limitar a velocidade de simulação [53].

Devido a estas limitações, que são transversais às linguagens consideradas como focadas na estrutura do processador, emergiram outras formas de descrição, conforme será abordado em secções seguintes deste capítulo, que são mais adequadas para geração de ferramentas de software necessárias para o projecto de sistemas embutidos.

---

[20] *Hardware Description Languages*

2.4.2 FOCADAS NO COMPORTAMENTO

O que caracteriza estas linguagens é a capacidade de especificarem explicitamente a semântica das instruções e abstrair-se de detalhes de hardware. Geralmente existe uma correspondência próxima da informação presente no manual do *instruction-set* com a descrição expressa nestas linguagens. Por outras palavras, estas linguagens permitem uma visão do ponto de vista do programador através da descrição do conjunto de instruções.

A informação captada pelas linguagens desta classe, permite a geração de ferramentas tais como simuladores e compiladores, entre outras. Seguidamente abordam-se duas linguagens classificadas como focadas no comportamento: nML e ISDL.

2.4.2.1 NML

A linguagem nML foi proposta por Freericks [54] e usada num gerador de código CBC [55] e pelo simulador ISS chamado SIGH/SIM [56]. Mais tarde também é usada no gerador de código CHESS e no simulador ISS CHECKERS que são produtos da empresa *Target Compiler Technologies* [57]. Devido ao facto de em ambos os sistemas os simuladores seguirem o modelo interpretado, ambos oferecem simuladores lentos.

O nível de abstracção desta linguagem é o conjunto de instruções do processador alvo, as quais alteram o estado deste durante a execução do programa. A descrição é feita de forma hierárquica, recorrendo a uma gramática de atributos [58], para conseguir descrições mais sucintas, explorando propriedades comuns a várias instruções. As raízes da hierarquia são instruções, e os elementos intermédios são fragmentos de instruções. Tanto às instruções como aos fragmentos são-lhes atribuídas acções, que descrevem os respectivos comportamentos. Isto permite o reaproveitamento de descrições que possam ser usadas na composição de outras regras. Os fragmentos podem ser agrupados através de regras AND e OR, formando uma árvore. Desta forma cada caminho da árvore corresponde a uma instrução.

Em nML cada elemento da hierarquia possui vários atributos. Estes contemplam o código binário (i.e. *image*), sintaxe do *assembly* e a semântica (i.e. *action*) de cada instrução. Os atributos dos elementos não terminais podem ser determinados através dos valores dos atributos dos respectivos “filhos”.

Apesar de ser classificada como focada no comportamento a linguagem nML também comporta informação relativa à estrutura do processador. Toda a informação sobre a estrutura, vista pelo conjunto de instruções deve ser descrita. Por exemplo, unidades de armazenamento devem ser declaradas uma vez que são visíveis para o conjunto de instruções. Assim, nML suporta por exemplo a declaração de RAM e registos. Uma descrição nML expressa as entidades que representam a estrutura do processador (visível para o conjunto de instruções) e as operações que descrevem o comportamento do processador. Esta linguagem suporta explicitamente modos de endereçamento.

A linguagem nML modela os conflitos entre operações, através da descrição do conjunto completo de combinações válidas entre elas, o que pode tornar as descrições longas e pouco intuitivas.

Em [59], foram testadas as funcionalidades da linguagem nML para a geração redireccionável de várias ferramentas (assembladores, desassembladores e simuladores ISS (interpretados e compilados)). Uma das principais limitações apontadas à linguagem nML, pelos autores daqueles testes, são o facto de aquela possuir um *program counter* implícito, o que se torna inconveniente no caso de instruções com tamanho variável. Outra limitação identificada advém da falta de flexibilidade na descrição comportamental das instruções, devido à impossibilidade de declarar variáveis locais.

2.4.2.2 ISDL

A linguagem *Instruction Set Description Language* (ISDL) foi desenvolvida no *Massachutes Institute of Tecnology*, EUA [60].

Esta linguagem foi projectada para apoio ao *hardware-software codesign* de sistemas embutidos e o seu alvo principal são arquitecturas ASIPs VLIW e DSPs. A estrutura de ISDL é similar à de nML e baseia-se numa gramática de atributos para descrever o conjunto de instruções. Também em ISDL os recursos de armazenamento são a única informação estrutural, mas o *program counter* é explícito.

A descrição do conjunto de instruções de um processador em ISDL inclui detalhes sobre o comportamento de cada instrução, o formato do *assembly*, o custo (número de ciclos de execução, tamanho ou conflito de recursos) e efeitos colaterais (ex.

actualização de *flags*). No entanto, esta linguagem contém uma secção para declaração de restrições, ou seja, combinações não permitidas de instruções, com o objectivo de possibilitar uma forma intuitiva de precaver essas situações. Em nML como só são descritas instruções válidas, para precaver aquelas restrições é requerida a adição de regras/acções adicionais, o que torna em nML as descrições mais longas e menos intuitivas.

O conjunto de ferramentas associado a esta linguagem contempla um gerador de código redireccionável conhecido por AVIV [61], um gerador de simuladores ISS interpretados (designados por XSIM) designado por GENSIM e um gerador de código Verilog para síntese de hardware chamado de HGEN [62].

Uma das limitações apontadas à linguagem ISDL é a sua inaptidão para descrever instruções multi-ciclo com tamanho variável [63, 64]. Outra desvantagem é a de que a estrutura simples em árvore das instruções proíbe a descrição de conjuntos de instruções com múltiplos formatos de codificação[21] [38].

2.4.2.3 OUTROS TRABALHOS

Além das linguagens anteriormente abordadas, outros trabalhos que se enquadram nesta classe de linguagens são:

Valen-C, Universidade de Kyushu: Extensão da linguagem C para geração de ferramentas de software [65].

***Computer Systems Description Languages* (CSDL), Universidade da Virginia:** Conjunto de linguagens (*Specification Language for Encoding and Decoding* (SLED), *Lambda-Register Transfer Lists* (λ-RTL), *Calling Convention Language* (CCL) e *Pipeline Unifying Notation Graphs and Expressions* (PLUNGE)) desenvolvidas para a infra-estrutura Zehpyr [66-68]. Cada uma das linguagens descreve uma determinada propriedade da máquina alvo, no entanto todas têm em comum o ponto de vista das instruções e estado [66]. SLED é usada para descrever o binário e o *assembly* das instruções seguindo um modelo hierárquico, à semelhança de ISDL, e faz parte do *New Jersey Machine-Code Toolkit* cujo objectivo é gerar código para ser usado por aplicações que processem código máquina (assembladores, desassembladores,

---

[21] *Encoding formats*

ligadores, compiladores, depuradores, etc.) [69]. Por outro lado λ-RTL é usada para descrever a semântica das instruções na forma de transferência entre registos. A linguagem CCL é usada para descrever as convenções de chamada de funções e a linguagem PLUNGE é usada para descrever estrutura de *pipelines*. No entanto, existe pouca informação sobre esta última bem como sobre a utilização destas linguagens para geração de simuladores.

Sim-nML: Extensão da linguagem nML melhorando descrição de hierarquia de memória e predição de saltos [70]. Esta linguagem é usada para geração de diversas ferramentas tais como simulador funcional, assemblador, depurador, compilador, etc.

Facile: Linguagem para descrição da microarquitectura e conjunto de instruções de processadores [71], proposta para tornar redireccionável o simulador FastSim [72], embora comprometa o desempenho deste simulador. A sintaxe desta linguagem derivou parcialmente da linguagem de descrição do *New Jersey Machine-Code Toolkit* [73] estendendo-a com a inclusão da possibilidade de poder descrever o comportamento/semântica das operações necessário para a geração de simuladores. Também nesta linguagem são usadas listas OR e listas AND para conseguir maior compactação.

A descrição do código binário das instruções é feita considerando sequências de instruções com mesmo tamanho, designados por *tokens*. No caso de arquitecturas com instruções de tamanho variável serão tantos os *tokens* quantos os diferentes tamanhos previstos. Na definição de cada *token* são declarados os campos que o compõem e as respectivas posições no seio do *token*, podendo haver sobreposições. A declaração dos padrões, *pattern*, associados a determinada mnemónica é feita com recurso à declaração de condições que os campos do respectivo *token* devem verificar para identificar aquela instrução, ou seja, mnemónica. Outra construção, ***sem***, é usada para definir o comportamento das instruções e associá-lo ao nome de *patterns***.** É possível, no entanto, verificar que há algumas limitações em explorar o comportamento comum de diferentes operações que partilham formatos idênticos. A linguagem assumiu um determinado tipo de informação disponível em manuais de processadores, na forma de tabelas como no Anexo E do manual do SPARC V9 [74], que não estando disponível dificulta a adaptação a este modelo de descrição.

2.4.2.4 CARACTERÍSTICAS E LIMITAÇÕES

Esta classe de linguagens apresenta como característica comum o facto de as descrições do conjunto de instruções se fazerem de forma hierárquica e se basearem numa gramática de atributos [58]. Esta abordagem torna simples a descrição porque explora as similitudes entre operações. No entanto é restritivo porque não contempla informação quanto a temporizações e relação com a *pipeline*. Esta restrição faz com que tal abordagem inviabilize a geração de simuladores *cycle-accurate* sem que haja algumas considerações quanto a controlo do comportamento [36]. Estas características levaram ao aparecimento de uma terceira geração de linguagens para ultrapassar as referidas limitações.

2.4.3 FOCADAS NO COMPORTAMENTO E ESTRUTURA

Nesta categoria de ADLs incluem-se as linguagens que permitem descrever detalhes das arquitecturas relacionados com o comportamento e estrutura das mesmas. Ao ter esta abrangência de informação estas linguagens permitem a geração de um leque mais amplo de ferramentas do que as anteriores. De seguida abordam-se de forma resumida várias linguagens geralmente classificadas nesta categoria: LISA, RADL, EXPRESSION, ArchC e MADL.

2.4.3.1 LISA

Language for Instruction Set Architecture (LISA), foi uma linguagem desenvolvida na Universidade de Tecnologia de Aachen, Alemanha [75]. Esta linguagem teve inicialmente como alvo o desenvolvimento de simuladores compilados acurados (bit e *cycle-accurate*) tendo por modelo de sequência de operações, para contemplar *hazards* de dados e controlo na *pipeline*, gráficos de Gantt estendidos que designaram por *L-charts* [76]. Pees, S., em [77] publicou uma evolução da linguagem com sintaxe capaz de permitir descrever: *pipelines* e seus mecanismos; contemplar *Single Instruction Multiple Data* (SIMD), VLIW, e arquitecturas super-escalares como alvos de modelação; suporte de técnicas de simulação compilada, orientação forte para a linguagem de programação C; suporte para *alias* e complexidade de codificação de instruções. Aquela versão da linguagem permite contemplar informação em cinco modelos diferentes (memória, recursos, comportamento, conjunto de instruções e tempo) e as descrições são compostas por declarações de recursos e de operações. A declaração dos recursos captura o estado do hardware do sistema (registos, memória, *pipeline*) e é usada para modelar a disponibilidade limitada de recursos que podem ser acedidos pelas operações em determinado instante. Por outro lado, as operações são os elementos básicos em LISA. Estas representam o ponto de vista do projectista, relativamente ao comportamento, estrutura, e conjunto de instruções do sistema. Para isso as descrições das operações são compostas por secções que aglutinam informações relativas aos diferentes pontos de vista. As secções em LISA são seis: *CODING* (binário da instrução), *SYNTAX* (sintaxe do *assembly*), *SEMANTICS* (semântica da instrução), *BEHAVIOR* (comportamento associado as acções que alteram o estado do sistema devido aquela instrução), *ACTIVATION* (marca o disparo de outras instruções relativamente à instrução corrente [78]) e *DECLARE* (permite declarações locais de identificadores ou alternativas (similar às regras *OR* da linguagem nML)).

Um sétimo ponto de vista de microarquitectura foi adicionado, com inclusão duma nova construção (*ENTITY*) para afectar operações a unidades funcionais na declaração de recursos, posteriormente à linguagem para permitir gerar código HDL para síntese de hardware [79]. A linguagem foi ainda estendida em [80] para modelar memórias não ideais, ou seja, ter em conta a latências em *caches* ou na interligação entre memórias.

A linguagem LISA tem vindo a ser utilizada em sistemas de geração de outras ferramentas de software, além de simuladores, assemblador, ligador e depurador [15], coexistindo com outras ferramentas de acordo com o objectivo dos respectivos trabalhos. Em [81] esta linguagem é usada com a infra-estrutura CoSy [82] para geração de compilador de C. Em [83] é integrada com SystemC [50] e em [84] com VHDL [47] para obtenção de modelos *Register Transfer Level* (RTL) sintetizáveis.

Ao conjugar informações relativas à estrutura e comportamento, duma arquitectura alvo, a linguagem tem sido fonte para geração de várias ferramentas (simuladores, compiladores e síntese) e de forma redireccionável (i.e. para vários processadores alvo).

A linguagem LISA, desenvolvida pelo *Institute for Integrated Signal Processing Systems* da Universidade de Tecnologia de Aachen, foi posteriormente licenciada pela empresa CoWare, Inc. [85] que comercializa o conjunto de ferramentas sob a designação de LISATek.

2.4.3.2 RADL

Retargetable Architecture Description Language (RADL) [86] é uma linguagem similar à descrita anteriormente e que derivou duma versão inicial daquela [75]. Esta linguagem foi desenvolvida na empresa Rockwell, Inc. como extensão da versão inicial da linguagem LISA focando a descrição do comportamento detalhado da *pipeline* para permitir a geração de simuladores acurados (*cycle-accurate* e *phase-accurate*). A linguagem permite descrever explicitamente detalhes da *pipeline* tais como *delay slots* e *hazards* [87]. Não existem, no entanto, publicações que revelem a eficiência de simuladores baseados nesta linguagem.

2.4.3.3 EXPRESSION

A linguagem EXPRESSION foi desenvolvida na Universidade da Califórnia, Irvine [88] para geração de simuladores acurados e compiladores. Esta linguagem é usada no compilador EXPRESS [89] e no simulador interpretado SIMPRESS [90]. A linguagem é composta por seis secções, três dedicadas à especificação de comportamento e outras três relacionadas com informação estrutural. Em relação às linguagens anteriores, a linguagem EXPRESSION, quando foi proposta, veio permitir a especificação de hierarquias de memória e a geração automática de tabelas de restrições [91] (i.e. *reservation tables*) necessárias para geração de compiladores.

No que diz respeito às secções relativas ao comportamento, verifica-se que tal como no caso da linguagem ISDL também em EXPRESSION há diferenciação entre instrução e operação. Nesta última, na versão original, não há descrição do *assembly* ou binário das operações. Assim, as três secções para descrição do comportamento inicialmente propostas são [92]:

- **Especificação das operações**: Conjunto de grupos de operações, em que cada grupo possui várias operações com algo em comum. Cada operação é descrita em termos do respectivo *opcode* (mnemónica), tipo de operação, comportamento e lista de operandos.
- **Descrição das instruções:** Esta secção representa o paralelismo que houver na arquitectura alvo. Uma instrução representa as operações que podem executar em paralelo. Cada instrução tem uma lista de *slots* que por sua vez

correspondem a unidades funcionais. Esta secção é primordial para arquitecturas VLIW.

- **Mapeamento das operações:** Esta secção contempla optimizações para o compilador, estão relacionadas com transformações para geração de código e estão relacionadas com a arquitectura.

Relativamente às três secções que representam a informação referente à estrutura, o seu significado é o seguinte:

- **Especificação dos componentes**: Corresponde à descrição ao nível RTL de cada bloco do sistema. Os blocos podem ser: Unidades da *pipeline*, unidades funcionais, portos, elementos de armazenamento, ligações e barramentos.
- **Descrição dos caminhos de dados e** *pipeline*: Esta secção descreve a *net-list* do processador. Assim a parte relativa a *pipeline* permite especificar as unidades correspondentes aos estágios das pipelines. Os caminhos de dados permitem descrever as transferências válidas. Esta informação é de primordial importância para extracção das tabelas de restrições (i.e. *reservation tables*) necessárias para o escalonador usado na geração do compilador.
- **Subsistema de memória**: Esta secção permite descrever os tipos e atributos dos vários elementos de armazenamento (bancos de registos, registos, memórias, etc.).

A sintaxe com que são descritas as secções anteriormente enumeradas, é similar à da linguagem Lisp [93].

Uma vez que o modelo de comportamento em EXPRESSION não segue uma especificação hierárquica de operações (i.e. AND-OR), torna-se verboso e consequentemente moroso especificar o conjunto de instruções. Ao consultar especificações feitas em EXPRESSION (ex. [94]) é fácil notar repetições de descrições, entre várias operações, porque não são exploradas as características comuns entre elas.

Tal como no caso do LISA, também a linguagem EXPRESSION tem sido estendida. Assim, relativamente à componente comportamental em [91] foi incluída uma subsecção *FORMAT* para que pudesse ser descrita informação sobre o formato do binário de cada operação (i.e. posicionamento dos campos, valor binário e *assembly*). Em relação à parte estrutural da linguagem, houve evoluções significativas relativamente à descrição do subsistema de memória [95], com inclusão de vasta gama de primitivas, e na especificação de detalhes relativos à *pipeline*, nomeadamente no que diz respeito à inclusão de formas de especificação de *hazards* [96].

Outra linha de investigação tem sido a validação formal e automática das descrições da *pipeline*, permitindo detecção nomeadamente de caminhos inválidos [97] e também a geração de modelos RTL a partir desta linguagem para arquitecturas com *pipeline* [98].

Mais recentemente *Reshady*, *M.*, [99] propôs um modelo para representar conjuntos complexos de instruções, componente comportamental. No modelo o projectista é forçado a usar máscaras o que por si é propício a erros e pouco intuitivo. Além disso, obriga a que as operações sejam agrupadas de acordo com os formatos de instrução.

2.4.3.4 ARCHC

A linguagem ArchC surgiu em 2001 a partir da popularização da linguagem de descrição de sistemas SystemC [50, 100], na qual a sintaxe se baseia. O SystemC é uma biblioteca de classes *C++* para facilitar a modelação de hardware. A linguagem, suas ferramentas e modelos estão acessíveis em [101].

Esta linguagem permite gerar automaticamente, várias ferramentas necessárias para o desenvolvimento de software de sistemas embutidos, entre elas: simuladores, assembladores, ligadores e ferramentas de depuração. Aos simuladores gerados por ArchC foi adicionada a capacidade de emular um sistema operativo.

Alguns dos modelos de processadores disponíveis em ArchC são: MIPS, IBM PowerPC [87], SPARC [102], e Intel 8051 [103].

A versão 2.0, segundo os autores, prevê uma fácil integração dos simuladores gerados por ArchC com outros módulos descritos em SystemC, compondo modelos de sistemas complexos. Outra funcionalidade importante será a capacidade de simular mais do que um processador, possibilitando a criação de modelos de sistemas *multi-core* [22].

Apesar da flexibilidade que a infra-estrutura herda pelo facto da microarquitectura ser implementada usando o SystemC, as descrições em ArchC são bastante verbosas e não exploram comportamentos comuns das operações. Havendo um método de comportamento para cada instrução [104]. Por outro lado, as descrições dos comportamentos das operações são efectuadas em *C*++, correspondendo à maior parte da descrição. Assim, o processo descrito em [104] prevê que após descrição em ArchC seja gerado o "esqueleto" das funções que posteriormente deverá ser preenchido com código *C*++ pelo projectista. Este processo de geração tem como desvantagem o facto de que ao alterar a descrição em ArchC ser exigido que também se altere o comportamento das operações em *C*++. Ou seja, recorrer a técnicas para não perder modelação anterior, uma vez que sendo a de maior volume de descrição pode fazer demorar consideravelmente o processo de desenvolvimento.

2.4.3.5 MADL

A linguagem *Mescal Architecture Description Language* (MADL) foi proposta por Wei Qin, Universidade de Princeton, EUA [37]. Esta linguagem serve de entrada para gerar o simulador SimIt-ARM [105], o qual é usado para demonstrar o modelo *Operation State Machine* (OSM) [106] para simulação do processador ARM. Este modelo serve de base à geração de simuladores de conjunto de instruções, com diferentes níveis de abstracção (i.e. emulador (ISS) e *cycle-accurate*), e compiladores *high-level language* (HLL) para processadores com arquitecturas escalares, super-escalares e VLIW.

Em MADL o projectista deve descrever para cada operação, um diagrama de estados designado por OSM. Cada estado dum OSM deve conter a descrição do comportamento nesse estado. Cada OSM executa em paralelo e antes de prosseguir para o estado seguinte, cada OSM tenta alocar recursos. Os recursos são descritos na forma de *tokens*, e as regras da sua alocação são descritas e geridas através de entidades designadas por gestores de *tokens* (*token managers*). A Figura 2.5 representa o modelo das OSM e é réplica duma figura publicada em [106].

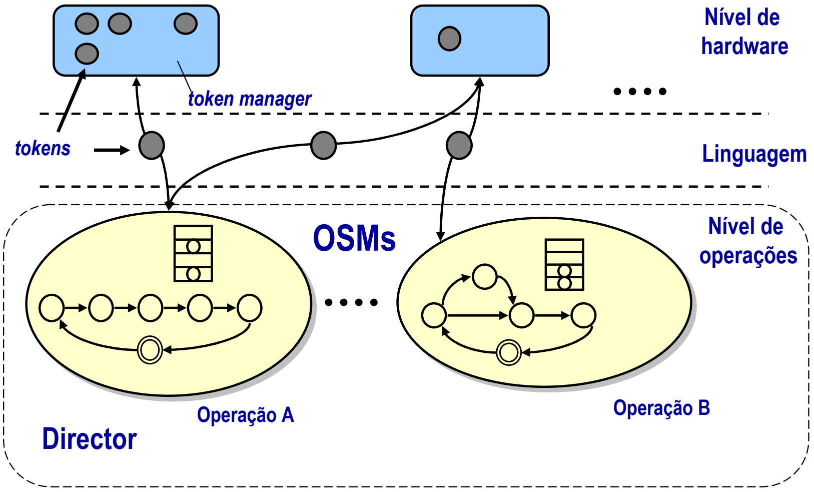


Figura 2.5: Modelo OSM

Este modelo é interessante porque separa a concorrência da execução de operações da concorrência da microarquitectura. A linguagem MADL permite modelar arquitecturas escalares, super-escalares e VLIW.

Também aqui (à semelhança de ISDL, LISA ou nML), é usada a forma AND-OR para modelar as operações em MADL de forma hierárquica.

O nível de hardware das OSMs não é descrito em MADL, nomeadamente o modelo de execução das unidades de hardware, incluído os gestores de *tokens*, mas sim implementado em *C++*. A estrutura das OSMs e as operações são modeladas em cinco secções da linguagem como a seguir se resume: 1 – secção para declaração de variáveis ou funções globais; 2 – declaração de gestores de *tokens*; 3 – definição de OSMs; 4- definição de funções; 5- definição de operações. Quanto a esta última, ela por sua vez pode subdividir-se nas seguintes subsecções: *VAR* (permite declarar variáveis locais acessíveis pela *syntax*); *CODING* (i.e. formato do código binário); *SYNTAX* (i.e. *assembly*); *EVAL* (para permitir inicialização de variáveis) e *TRANS* (i.e. acções em determinado estado da correspondente OSM). O mapeamento entre operações e OSM é declarado usando a construção *USING*.

O modelo de OSMs foi proposto para a geração de simuladores *cycle-accurate*, mas quando se trata de gerar simuladores ISS, revela algumas limitações. Nomeadamente um dos factores limitativos, apontados pelo autor, está relacionado com o facto de a leitura e escrita nos registo e memória ser feita via protocolo de acesso aos *tokens*, o que faz com que este nível extra de indirecção lhe retire performance, quando comparado com simuladores implementados de forma não automática [37].

Ao consultar descrições em MADL [105] facilmente se detectam várias repetições pelo facto de as operações serem agrupadas de acordo com as OSMs, comportamento e modo de endereçamento. Assim, nota-se que para descrever as mesmas operações mas com modo de endereçamento diferente, existe quase uma duplicação da descrição. As diferenças estão apenas na forma como são determinados alguns dos operandos.

2.4.3.6 OUTROS TRABALHOS

Além das linguagens anteriormente abordadas, outros trabalhos que se enquadram nesta classe de linguagens são: HMDES e FlexWare.

***High-level Machine Description* (HMDES), Universidade de Ilinois, EUA**: Esta linguagem [107] serve de entrada para a infra-estrutura de compilação designada por TRIMARAN [108], que suporta geração automática de compiladores para arquitecturas *Explicitly Parallel Instruction Computing* (EPIC). As descrições em HMDES são posteriormente convertidas para outra linguagem, MDES, que adequa a informação para um formato próprio para o compilador [109]. Esta última, por sua vez, permite redireccionabilidade do simulador *cycle-accurate* para famílias do processador HPL-PD [110].

A estrutura e comportamento são descritos em secções (formatos, utilização de recursos, latência, operações, registos, etc.) de forma hierárquica. A linguagem permite descrição explícita de restrições de operações para construção de tabelas de restrições [107]. Uma das críticas feitas ao sistema TRIMARAN é a restrita gama de arquitecturas às quais pode ser aplicado [111].

FlexWare, SGS-Thomson Microelectronics, França: FlexWare [112] é um sistema composto por um gerador de código, CodeSyn [113], e por um simulador, Insulin [114]. A descrição para o CodeSyn inclui três componentes: representação ao nível comportamental do conjunto de instruções, os recursos disponíveis (e respectiva classificação) e a relação de interligação entre eles (representando a estrutura dos caminhos de dados). A descrição do conjunto de instruções consiste numa lista de macro-instruções genéricas para executar cada instrução do processador alvo. O Insulin usa um modelo diferente, baseando-se num modelo parcialmente reconfigurável em VHDL de um processador genérico. O projectista descreve o código *assembly* alvo que depois é convertido, por um *cross assembler*, no *assembly* genérico que é suportado pelo modelo VHDL. Uma desvantagem do sistema FlexWare é o facto de não permitir que a mesma descrição possa ser usada para gerar todas as ferramentas. Outras desvantagens que lhe são atribuídas estão relacionadas com a gama limitada de arquitecturas de processadores e com o desempenho do simulador [14].

2.5 CONCLUSÃO

Apesar de terem sido propostas diversas linguagens, nenhuma delas foi, até ao momento, adoptada como padrão. ADLs são um tópico de investigação relativamente recente e com vários problemas por resolver.

As ADLs centradas na estrutura (ex. MIMOLA e UDL/I) por terem descrições com baixa abstracção tornam difícil a extracção de informação relativa ao conjunto de instruções e não são adequadas para exploração ao nível de sistema e exploração de arquitecturas, devido ao excesso de detalhe que contêm. Estas são adequadas para síntese de hardware, mas esta etapa acontece já em fases finais do projecto.

As ADLs centradas no comportamento (ex. nML e ISDL) impõem sintaxes que se baseiam em gramáticas de atributos. Este não é um estilo de sintaxe amigável para quem desenvolve hardware, e é restritivo relativamente a descrições de comportamentos. Por exemplo, o facto de não permitir a declaração de variáveis locais e o facto de não possuírem um *program counter* explicito, torna difícil a tarefa de modelar processadores com instruções de tamanho variável.

As linguagens híbridas - comportamento e estrutura – tais como EXPRESSION, LISA e ArchC tentam explorar as vantagens das antecessoras. No entanto também estas apresentam algumas características possíveis de melhoria. EXPRESSION apresenta uma sintaxe semelhante a LISP, tornando a legibilidade das descrições de certa forma difícil para projectistas de hardware que não estejam habituados àquela linguagem. Em LISA descrevem-se os comportamentos centrados nas operações, tendo de se obedecer a uma estrutura em árvore, *coding tree*. Os comportamentos podem ser descritos em *C++*, o que dificulta a tarefa de validação das descrições logo na fase de descrição.

Outra linguagem mais recente, ArchC, tem como característica o facto de a respectiva sintaxe seguir um estilo semelhante ao de SystemC. Esta linguagem no entanto impõe que sejam descritas todas as instruções, não agrupando as mesmas, e por isso conduz a descrições longas o que a torna tediosa para processadores com conjuntos de instruções complexos, ex. ARM. Também aqui o projectista é obrigado a descrever o comportamento de cada uma das instruções em *C*++. Analisando as respectivas descrições é possível detectar várias repetições de certos blocos da linguagem.

No Capítulo 3 é apresentada a estrutura e as construções da linguagem que constitui a principal contribuição deste trabalho uma vez que permite representações compactas e intuitivas de conjuntos de instruções de processadores reais. A estrutura da linguagem por sua vez permite a utilização de conceitos de programação não usados nas linguagens anteriormente mencionadas, nomeadamente *generic programming*, permitindo que o código gerado para determinadas ferramentas de software seja menor e por sua vez conduz a melhorias de desempenho, nomeadamente tempo de compilação.

CAPÍTULO 3 Linguagem MiADL

3.1 INTRODUÇÃO

Neste capítulo é apresentada a linguagem proposta nesta tese. Após expor o propósito da linguagem, as restantes secções deste capítulo apresentam a estrutura e as características da linguagem, nomeadamente o significado das suas construções.

Ao longo deste capítulo é apresentada uma perspectiva global da linguagem MiADL, mostrando que esta foi projectada tendo por base fontes de variabilidade e de regularidade presentes nos ISAs de processadores actuais, que entretanto foram estudados e modelados na nova linguagem. Além do comportamento das instruções, a linguagem possui também construções próprias para representação compacta do *assembly* relativo às mesmas.

3.2 PROPÓSITO

A complexidade crescente das arquitecturas exige elevado desempenho por parte dos simuladores e, por outro lado, a variedade de arquitecturas existentes faz com que a redireccionabilidade se torne também um factor de elevada importância para o sucesso daquelas ferramentas. A redireccionabilidade requer modelos genéricos enquanto que o desempenho requer optimizações próprias da arquitectura alvo.

Assim, para conseguir uma infra-estrutura de simulação que seja redireccionável e que permita ao mesmo tempo conseguir bom desempenho, em termos de velocidade, é necessário ter em conta vários factores dos quais se destacam: a forma de descrição das instruções, o algoritmo de descodificação – que usa a forma de descrição para descodificar o binário da aplicação a simular – e a forma como são executadas as instruções descodificadas. Neste capítulo, o foco será dado à forma de descrição, i.e., linguagem, e no capítulo seguinte serão tratados os outros dois factores. Todos estes factores são importantes e ignorar qualquer um deles limita, ou coloca em causa, a qualidade dos simuladores gerados.

Para conseguir boas descrições, uma linguagem de descrição de arquitecturas, ADL, deve permitir representar facilmente, de forma compacta e não ambígua, a informação para uma vasta gama de diferentes arquitecturas, i.e. ser redireccionável. Em termos gerais, a linguagem deve ser capaz de capturar as complexidades dos modos de endereçamento, o código binário das instruções, e o comportamento/semântica das instruções da arquitectura. Propor uma linguagem, que consiga descrever uma vasta gama de arquitecturas, é uma tarefa complexa porque existem muitas diferenças de formatos e especificidades próprias de cada arquitectura. Para se ter um simulador redireccionavel a linguagem deve ser genérica, ou seja abstracta, para suportar uma vasta gama de arquitecturas, mas nesses casos o desempenho é geralmente modesto. Por outro lado, se a linguagem permitir detalhes específicos de determinada família de processadores, perde redireccionabilidade, apesar de permitir atingir desempenhos superiores. As linguagens descritas no capítulo anterior, na sua generalidade, obrigam a descrições longas para conseguir gerar simuladores rápidos.

Assim, tendo por base o anteriormente referido, projectou-se uma linguagem genérica de descrição de processadores que permite conseguir redireccionabilidade de simuladores funcionais para vários processadores tais como RISC e CISC. Os processadores modelados até ao momento correspondem a arquitecturas reais de conjunto de instruções conhecidas, sendo elas: SPARC [115], ARM [116] e MCS8051 [103]. A selecção destes casos de estudo esteve relacionada com a complexidade dos ISAs, o facto de pertencerem a classes diferentes de arquitecturas (RISC *vs* CISC) e por serem processadores populares em sistemas embutidos. Além da redireccionabilidade, a linguagem apresentada nesta tese possui também como característica o facto de permitir descrições compactas e de fácil construção a partir de informação existente na maioria dos manuais de processadores. Estas características tornam mais célere o processo de descrição da arquitectura do conjunto de instruções e facilitam a compreensão da mesma. Ou seja, ajudam a permitir encurtar o tempo de desenvolvimento, conforme desejado.

Além da compactação, redireccionabilidade e forma natural de descrição, a linguagem aqui proposta possui construções para a tornar robusta, evitando redundâncias e permitindo a detecção de erros de descrição que por vezes são difíceis de encontrar, tais como, por exemplo, a repetição de códigos binários de instruções.

As descrições são usadas neste trabalho para gerar simuladores funcionais, seguindo o modelo de simulação compilada estática, de acordo com a infra-estrutura descrita no capítulo seguinte da tese. A geração do simulador (*back end*) é separada do processamento da descrição (*front end*), pelo que se optou por descrever neste capítulo a linguagem e no próximo a infra-estrutura subjacente que a partir dela gera o simulador (*front end* e *back end*). Esta independência faz com que seja possível implementar diferentes geradores a partir de descrições feitas na linguagem, ou seja, gerar diferentes ferramentas (ex. assemblador, desassemblador, entre outras.), a partir da informação que consta da descrição. Para o êxito da infra-estrutura deve ser garantido que a descrição não contenha erros, e dai a importância duma análise semântica robusta e uma estrutura da linguagem que a torne capaz de detectar incoerências ou redundâncias.

No caso da geração de simuladores, a informação presente na descrição relativa à codificação e comportamento das instruções da arquitectura do conjunto de instruções é utilizada para a geração do descodificador de instruções e do mapeamento entre instruções e respectivo comportamento/semântica. A linguagem possui também construções próprias para a descrição do *assembly* das instruções, para geração de outras ferramentas além de simuladores, tais como: assembladores e desassembladores.

Uma linguagem estruturada deve permitir reutilização de descrições. Para facilitar a reutilização, as descrições geralmente usam a noção de símbolo ou identificador para representar uma entidade. Assim, em vez de repetir a descrição da entidade, o respectivo símbolo é usado para a referenciar. Tal como, no caso do uso de variáveis em linguagens de programação. As linguagens existentes fazem corresponder um símbolo a apenas uma entidade. Além disso, estes símbolos representam operandos de instruções. Devido a isso, são limitados por número e por tipo de entidades que podem representar. Por outro lado, na linguagem aqui apresentada um símbolo pode representar mais do que uma entidade. Além disso, pode também representar diferentes tipos de entidades tais como operandos ou operações. Isto permite representações mais compactas, e facilita a exploração de todos os valores possíveis de símbolos para gerar simuladores optimizados.

Em suma, ao longo deste capítulo é apresentada a linguagem MiADL cujo propósito é permitir:

- descrever ISAs tendo a capacidade de lidar com diversas fontes de variabilidade que os mesmos apresentam (i.e., modos de endereçamento, formatos binários das instruções, comportamentos das operações, classes de operações, etc.);
- suporte para geração de ferramentas de *software* para o projecto de sistemas embutidos, nomeadamente simuladores funcionais;
- suporte para geração de assembladores e desassembladores através da descrição de informação sobre o código binário e a sintaxe do *assembly*, contemplada por construções próprias da linguagem;
- redireccionabilidade de descrição de arquitecturas de conjuntos de instruções;
- representações compactas;
- representações fiáveis, i.e. com elevado suporte de validação estática da linguagem;
- fácil descrição a partir de informação que consta geralmente nos manuais de utilizador.

Nas próximas secções apresenta-se a estrutura e secções da linguagem MiADL bem como a discussão sobre o significado de cada uma delas. O objectivo é apresentar as considerações tidas em conta durante o projecto de cada uma das características da linguagem (i.e., secções, construções, entre outros). São também apresentados exemplos com base em três dos processadores modelados, para demonstrar a flexibilidade da linguagem e a consequente capacidade de redireccionabilidade.

3.3 LINGUAGEM MiADL

A linguagem nomeada de ***Mi**nho **A**rchitecture **D**escription **L**anguage* (MiADL) é uma linguagem projectada para permitir a geração de ferramentas usadas na exploração de arquitecturas de processadores. Esta permite a construção de modelos de processadores fáceis de elaborar e manter. As descrições MiADL são compostas por informação suficiente para geração automática de ferramentas de desenvolvimento de software tais como: simulador, assemblador e desassemblador. Desta forma, é possível gerar um conjunto consistente de ferramentas a partir de uma única descrição do processador. A linguagem MiADL representa o modelo de programação do ponto de vista do projectista de software e um modelo de recursos abstracto, composto pelos recursos vistos pelo software.

A linguagem MiADL permite especificar várias arquitecturas de conjunto de instruções. Através de descrições simples, a linguagem explora as características comuns entre operações, resultando em representações compactas e coerentes. A sintaxe da linguagem possibilita a fácil adaptação de informação que tipicamente consta nos manuais de arquitectura de processadores, e possibilita a detecção precoce de incoerências de especificação.

A semântica da linguagem MiADL foi projectada especialmente para modelação de processadores. No sentido de modelar as arquitecturas do conjunto de instruções, vários trabalhos têm estruturado as linguagens de modelação explorando diferentes formas de regularidade. Esta regularidade reflecte possíveis graus de liberdade no projecto. Assim, a linguagem deve ter construções/sintaxe que imponham ou expressem naturalmente a regularidade desejada, mantendo no entanto possível a descrição de casos especiais.

Estudando ISAs actuais, nota-se que as operações partilham determinados comportamentos comuns, mas mesmo assim, a sua variabilidade dificulta uma representação compacta. Explorando os comportamentos comuns é possível compactar a representação desses ISAs.

Para o projecto da linguagem MiADL, foram identificadas quatro fontes de variabilidade que funcionaram como premissas para o projecto da linguagem. As fontes de variabilidade identificadas são as seguintes [117]:

- Existem instruções que apesar de terem comportamento idêntico, variam em tamanho. Por exemplo, no caso do processador MCS8051 [103] existem 4 instruções diferentes de adição, cujos tamanhos podem ser de um ou dois *bytes*, conforme expresso na tabela seguinte:

Tabela 3.1: Exemplo de instruções do MCS8051

Tamanho	***Assembly***	**Codificação da instrução**		
		1.º byte		**2.º byte**
2 bytes	ADD A,#data	0010	0100	Dado imediato (8 bits)
1 byte	ADD A,@Ri	0010	011i	
2 bytes	ADD A,direct	0010	0101	Endereço directo (8 bits)
1 byte	ADD A,Rn	0010	1rrr	

Ou seja, para uma mesma operação de determinado ISA é possível que existam vários tamanhos possíveis.

- Existem instruções que apesar de terem o mesmo comportamento, variam em termos de modos de endereçamento. Por exemplo, no caso do processador ARM [116] existem três instruções de adição cujo tamanho é igual, 32 bits, e cujo comportamento é idêntico excepto na forma como o segundo operando da operação é determinado. Os formatos possíveis para a operação de adição para aquele processador estão representados na Figura 3.1.

Bits →	31..28	27..25	24..21	20	19..16	15..12	11..8	7	6..5	4	3..0
dpis →	cond	000	*opcode=0100*	s	rn	rd	*shift amount*		*shift*	0	rm
dprs →	cond	000	*opcode=0100*	s	rn	rd	rs	0	*shift*	1	rm
dpi →	cond	001	*opcode=0100*	s	rn	rd	*rotate*	*immediate*			

Segundo operando

Figura 3.1: Variabilidade relativa a formatos de instruções do processador ARM

Analisando a figura é possível observar que existe variabilidade entre formatos e que esta é caracterizada pelos bits que se encontram entre as posições 25 e 27 e pelos 12 bits menos significativos da instrução.

A variabilidade em termos dos formatos possíveis para uma determinada operação, neste caso a de adição, reflectem-se também na sintaxe do *assembly* que para o caso do processador e operação usados neste exemplo é [116]:

- ADD{<cond>}{<s>} <rd>, <rn>, <rm>, ShiftOp #<shift amount> , para o formato dpis[22].
- ADD{<cond>}{<s>} <rd>, <rn>, <rm>, ShiftOp <rs> , para o frmato dprs[23].
- ADD{<cond>}{<s>} <rd>, <rn>, #<immediate>, para o formato dpi[24].

Apesar de nos três casos se tratar da operação de adição, os formatos da instrução são quem caracteriza o modo de endereçamento e representam variabilidade na forma como são determinados os respectivos operandos.

- Existem operações que podem pertencer a uma mesma classe, apesar de terem comportamentos diferentes. Geralmente, o comportamento das operações duma classe é implementado na mesma unidade funcional, i.e., de hardware. Exemplos disso são as operações de adição e subtracção que são classificadas no grupo das aritméticas.

  Como exemplo deste tipo de variabilidade, no caso do processador ARM existem seis classes de instruções: instruções de salto; instruções de processamento de dados; instruções de *load* e *store*; instruções do coprocessador; instruções para geração de excepções e instruções relativas ao registo de *status*. Dentro destas classes ainda existem sub-agrupamentos de instruções. Por exemplo a classe de instruções de processamento de dados subdivide-se segundo [116] nos seguintes quatro tipos de grupos de instruções: instruções aritméticas; instruções de comparação; instruções de multiplicação e instruções de contagem de zeros.

  Para cada um dos agrupamentos existem semelhanças entre as instruções nomeadamente no que diz respeito ao comportamento e sintaxe do

---

[22] *Data processing immediate shift*

[23] *Data processing register shift*

[24] *Data processing immediate*

assembly. Entre agrupamentos, ou classes, de instruções diferentes existe variabilidade.

- Todas as operações possuem comportamento próprio que as distingue das demais na arquitectura do conjunto de instruções. Por exemplo, operações de adição ou subtracção apesar de poderem fazer parte do grupo das aritméticas, e poderem implementar os mesmos tipos de modos de endereçamento, distinguem-se entre si pelo tipo de operação aritmética que implementam.

Além das fontes de variabilidade anteriormente mencionadas, existe também uma certa regularidade nas arquitecturas dos conjuntos de instruções que são exploradas pela MiADL. Uma das regularidades está relacionada com a posição de determinados componentes dos formatos das instruções, sempre nas mesmas posições e com o mesmo tamanho (i.e., número de bits) em diferentes formatos. Outra regularidade está relacionada com o facto de para um determinado grupo de operações, grande parte do comportamento ser igual, excepto no que está relacionado com o modo de endereçamento ou código da operação. Estas regularidades também são exploradas em MiADL.

A gramática da linguagem MiADL é anexada a esta tese, Anexo 1. Para representar a gramática usou-se a notação *Backus Naur Form* (BNF).

De seguida é apresentada a estrutura, secções e as construções de MiADL

3.4 ESTRUTURA DE MiADL

A linguagem MiADL permite lidar com as fontes de variabilidade, referidas anteriormente, através da sua estrutura composta por várias secções cuja, relação entre elas se apresenta na Figura 3.2.

As secções da linguagem são: uma secção para declaração de elementos externos à descrição e uma secção para descrição da arquitectura do conjunto de instruções do processador alvo. Esta última subdivide-se nas seguintes secções:

- uma secção para definição de elementos globais (i.e., que são visíveis por todas as secções seguintes);
- uma secção dedicada à declaração de recursos;
- uma secção reservada para a declaração dos formatos das instruções;

- várias secções próprias para a declaração de grupos de operações, nas quais cada grupo representa uma classe de operações que tenham entre si similitudes em termos de comportamento e *assembly*.

```
extern {
    Declaração de funções externas
}
isa isa_name <endianess, palavra> {
    global {
            Definição de funções globais
    }
    resources {
            Declaração de recursos
            Inicialização de recursos
    }
    iformats <tamanho> {
            Declaração de campos com posição variável
            Declaração de formatos de instruções {corpo}
            Declaração de grupos de campos
            Definições cujo scope seja a secção iformats
    }
    group nome_do_grupo {formatos possíveis} <selector> {
            Semântica de cada operação do grupo
            Comportamento comum a todas as operações do grupo
            Definições cujo scope seja a secção group
    }
            …
}
```

Figura 3.2: Estrutura da linguagem MiADL

De seguida são descritas as secções, bem como as construções da linguagem, e a forma como são conseguidas representações compactas e efectuadas diversas verificações.

3.5 EXTERN

Nesta secção da linguagem MiADL, é possível declarar funções que são visíveis a partir das demais secções da linguagem. Esta zona serve como interface com código externo, em *C/C++*, através da possível declaração de funções externas. A secção é caracterizada pela palavra-chave ***extern*** que precede um bloco no qual podem ser declaradas as funções externas.

A sintaxe usada para declarar as funções será apresentada numa das secções seguintes deste capítulo.

3.6 ISA

Esta secção envolve outras que serão abordadas em seguida. A secção inicia-se com a palavra-chave ***isa***. Ainda antes do corpo desta secção, deve ser indicado o nome que identificará o ISA do processador alvo e também dois parâmetros opcionais que são a *endianess* e o tamanho – em número de bits – da palavra do processador alvo, Figura 3.2. Quanto à *endianess*, o projectista poderá indicar uma de duas alternativas possíveis: ***big*** para indicar que a ordem é *big-endian* ou ***little*** para indicar que é *little-endian*. Se não forem indicados aqueles parâmetros opcionais, é assumido por defeito que o tamanho da palavra será igual ao do menor formato de instrução do processador alvo e que o processador em termos de *endianess* será *big-endian*. Seguidamente descrevem-se as características das diversas subsecções que compõem esta secção de MiADL.

3.6.1 GLOBAL

Nesta secção da linguagem MiADL, é permitido definir funções ou variáveis que são visíveis a partir das demais secções da linguagem, excepto na secção ***extern***. A secção inicia com a palavra-chave ***global***, Figura 3.2.

Esta secção da linguagem permite que a partir de qualquer outra secção da descrição MiADL – excepto na secção *extern* – sejam invocadas funções nela definidas as quais desta forma necessitam de ser definidas uma única vez.

3.6.2 RECURSOS

A secção ***resources*** da linguagem MiADL lista as definições de todas as entidades que representam o estado do processador. Estas entidades estão relacionadas com armazenamento de dados tais como memórias, registos ou *flags*. A declaração dos recursos segue um estilo idêntico ao de declaração de variáveis e os mesmos têm um *scope* que é visível pelas restantes secções da linguagem. Isto está relacionado com a salvaguarda de propriedades de recursos de hardware, os quais por natureza são globais.

3.6.2.1 DECLARAÇÃO DE RECURSOS

A secção ***resources*** contempla a declaração dos seguintes tipos de recursos, que correspondem a variáveis da linguagem de programação:

Registos: Para definir um registo, ou lista de registos, de determinado tipo, deve ser usado a construção ***register***. Na declaração, além do nome que deve ser único, os registos são também caracterizados pelo tipo de dados que armazenam. A sintaxe é apresentada na Figura 3.3.

register lista_de_registos **:** tipo **;**

Figura 3.3: Sintaxe para declaração de registo

Memórias: Para definir uma memória, ou lista de memórias, com determinada capacidade deve ser usado a construção ***memory.*** A capacidade de armazenamento da memória pode ser expressa em número de *bytes*, *kilobytes* (K), *megabytes* (M), ou *gigabytes* (G). A sintaxe é apresentada na Figura 3.4.

memory lista_de_memórias **;**

Figura 3.4: Sintaxe para declaração de uma lista de memórias

Cada memória, pertencente à lista de memórias separadas por vírgula, é caracterizada por um nome e tamanho de acordo com a seguinte sintaxe:

nome **[**tamanho**]**

Figura 3.5: Sintaxe para declaração de uma memória

Banco de registos: Para declarar um banco de registos, composto por registos com determinado tipo de dados, deve ser usado a construção ***regfile***. Na declaração é indicado o nome, que deve ser único, e o número de registos de determinado tipo, que consta também da definição. A sintaxe para uso desta construção é apresentada na figura seguinte:

```
regfile nome [número_de_registos] : tipo ;
```

Figura 3.6: Sintaxe para declaração de um banco de registos

Formato de registo: A linguagem MiADL permite definir a composição interna de registos. Isto tem particular importância quando se trata de registos que armazenam o estado do processador. Assim, é possível declarar explicitamente o formato desses registos e posteriormente aceder ao registo como um todo, ou a partes do seu conteúdo, ou seja, a determinadas *flags* que o compõem. A Figura 3.8 apresenta o exemplo da utilização desta construção, ***reglayout*,** na descrição de recursos usando MiADL para o caso do processador ARM [116].

A sintaxe desta construção de MiADL é a seguinte:

```
reglayout nome <tamanho> ( lista_de_campos );
```

Figura 3.7: Sintaxe para definição de um formato de registo

Assim, ao declarar um formato, devem ser indicados os campos que o compõem, sendo para cada campo indicadas as posições que ocupam relativamente ao primeiro bit do formato em que está a ser definido. São verificadas ocorrências de repetições ou de sobreposições de campos para evitar erros de descrição de formatos. É também verificado se o somatório dos tamanhos dos campos que compõem o formato perfaz o tamanho previsto para o formato de registo em causa, o qual é indicado logo após o nome do formato.

Posteriormente, estes formatos de registos podem ser usados como tipos para a declaração de registos cujo formato seja aquele, ou seja, que tenham em comum aquela ordem e composição de *flags*. No caso do processador ARM, os registos *Saved Program Status Register* (SPSR) e *Current Program Status Register* (CPSR) possuem ambos o mesmo formato e por isso, basta definir uma vez aquele formato e declarar aqueles dois registos como sendo daquele padrão.

Para cada formato de registos é possível definir funções que manipulem os campos que o compõem. Estas funções podem ser invocadas a partir de qualquer secção da linguagem. Assim, o projectista poderá abstrair-se de alguns pormenores tais como a localização (bits que ocupam) dos campos no seio do registo daquele formato. Isto permite lidar apenas com determinados campos ou *flags* do registo que se pretende actualizar ou consultar, o que reduz erros de descrição e ao mesmo tempo torna mais fácil e célere a construção do modelo do processador.

Program counter: Este atributo serve para indicar o recurso que possui o endereço corrente do programa. A sintaxe para a declaração deste recurso consiste em, após a palavra-chave ***prgcounter***, indicar o recurso que terá aquele papel na arquitectura alvo. Na Figura 3.8 mostra-se o uso desta construção para o caso da descrição dos recursos do processador ARM.

Funções de apoio: Dentro da secção *resources* da linguagem MiADL, podem ser declaradas funções que manipulem os recursos previamente mencionados. Estas funções, à semelhança dos recursos que manipulam, também são visíveis por todas as outras secções da linguagem. Assim, em MiADL é possível definir uma vez estas funções e usá-las sempre que necessário. De qualquer modo, poderá sempre aceder directamente aqueles recursos uma vez que como vimos na subsecção anterior, estes são globais.

```
resources {
    ...
    memory Memory[4M];  // memória com 4 Megabytes de capacidade
    regfile rFile[16]: int<32>;
    reglayout PSW<32> (NF[31] ZF[30] CF[29] VF[28] QF[27]
                      ?[26:8] IF[7] FF[6] TF[5] MOD[4:0] )
    register SPSR, CPSR: PSW;  // registos cujo formato é PSW
    prgcounter rFile[15];
    …
}
```

Figura 3.8: Excerto da declaração de recursos em MiADL para modelo de processador ARM

3.6.2.2 INICIALIZAÇÃO DE RECURSOS

A secção *resources* pode conter também uma subsecção para inicialização de recursos. Esta secção é opcional e permite caracterizar o estado inicial do processador, permitindo por exemplo atribuir valores iniciais a determinadas *flags* ou registos. A sintaxe desta subsecção é representada na Figura 3.9.

init {
    Inicializações de recursos
}

Figura 3.9: Secção para inicialização de recursos

3.6.3 FORMATOS DAS INSTRUÇÕES

Esta secção representa a codificação das instruções de determinada arquitectura do conjunto de instruções. É nesta secção que são exploradas duas das fontes de variabilidades referidas no início deste capítulo. Trata-se do tamanho das instruções e dos modos de endereçamento, uma vez que é aqui que são descritos todos os formatos e respectivos tamanhos. A regularidade que possa existir entre formatos de instruções, quanto à posição que alguns elementos que os compõem ocupam entre eles, também é descrita nesta secção. Assim, para expressar e testar as variabilidades e regularidades relativas a formatos de instruções, MiADL prevê uma secção *iformats* que é composta pelas seguintes 3 subsecções: declaração dos campos que possam estar em posições diferentes entre formatos, definição de formatos e definição de grupos de campos. A estrutura da secção *iformats* é representada na Figura 3.10.

iformats (<tamanho_dos_formatos>)$_{opt}$ {
    Declaração de campos com posição variável
    Declaração de formatos de instruções ({corpo})$_{opt}$
    Declaração de grupos de campos
    Declarações cujo *scope* seja a secção *iformats*
}

Figura 3.10: Secção *iformats*

No caso de arquitecturas com tamanho de instrução constante, ex. ARM [116] ou SPARC [115], esse tamanho é indicado uma única vez no início da secção após a palavra reservada ***iformats***. Para arquitecturas com instruções de tamanho variável, por exemplo MCS8051 [103], a descrição conterá junto de cada formato a indicação do respectivo tamanho. Desta forma torna-se simples descrever a variabilidade relativa a tamanhos de formatos de instruções para determinado ISA.

Nas próximas secções são descritas as subsecções que compõem a secção *iformats*.

3.6.3.1 CAMPOS DE POSIÇÃO VARIÁVEL

Um dos tipos de elementos que compõem os formatos é o campo. Cada campo é caracterizado por um nome, e pelos bits que ocupa no seio do formato a que pertence.

Verifica-se que nos ISAs existe uma regularidade quanto à posição que os campos ocupam em diferentes formatos. Assim, geralmente, a posição é a mesma, havendo no entanto casos em que tal não acontece. Por exemplo, na Figura 3.11, para o caso do processador ARM [116] verifica-se que o campo correspondente ao índice do registo destino, *rd*, se posiciona sempre entre as posições 12 e 15 independentemente do formato em que esteja envolvido. No entanto, noutras arquitecturas de conjunto de instruções, como a do MCS8051 [103], acontece que o mesmo campo pode estar em diferentes posições. No caso do MCS8051, o campo *opcode* corresponde às posições dos bits mais significativos, verificando-se que as mesmas variam entre formatos, porque se trata de uma arquitectura com instruções de tamanho variável.

Bits →	31..28	27..25	24..21	20	19..16	15..12	11..8	7	6..5	4	3..0
dpis →	cond	000	*opcode*	s	rn	rd	*shift amount*		*shift*	0	rm
dprs →	cond	000	*opcode*	s	rn	rd	rs	0	*shift*	1	rm
dpi →	cond	001	*opcode*	s	rn	rd	*rotate*	*immediate*			

Figura 3.11: Exemplos de formatos de instruções do processador ARM

Em MiADL é assumido por defeito que entre formatos diferentes, campos com o mesmo nome devem ocupar as mesmas posições de bits. No entanto para arquitecturas em que tal deva ser "relaxado", existe uma construção opcional da linguagem com o qual pode ser declarada a lista de campos cujo posicionamento entre formatos de instruções possa variar. A sintaxe para declarar os campos é a seguinte[25]:

relocfields lista_de_campos **;**

Figura 3.12: Declaração de campos com posição variável

Assim, os campos presentes naquela lista não terão as suas posições verificadas por vontade de quem descreve. Este relaxar da obrigatoriedade de campos com o mesmo nome terem de estar em posições iguais, pode facilitar a compactação. Isto é possível porque permite que aqueles campos, apesar de estarem em posições diferentes, mas tendo o mesmo nome e tamanho, possam corresponder a *opcodes* de operações cujo comportamento e *assembly* sejam semelhantes. Na secção 3.6.4 esta característica ficará mais clara ao ser explicado o significado do *selector* em grupos de operações.

3.6.3.2 FORMATOS DE INSTRUÇÕES

A definição dos formatos especifica as possíveis codificações das instruções para determinada arquitectura do conjunto de instruções. Em MiADL a construção ***layout*** permite descrever os formatos de instruções de acordo com a seguinte sintaxe[26]:

layout nome (**:** alias)$_{opt}$ (<tamanho>)$_{opt}$ **(** elementos_do_formato **)** (**{**corpo**}**)$_{opt}$

Figura 3.13: Sintaxe para definição de um formato de instrução

Durante a especificação dos formatos três tipos de elementos podem ser utilizados: campo, constante (em binário) e indefinido[27]. Um campo representa informação útil ao comportamento da instrução durante a execução. As constantes são usadas no processo de descodificação das instruções, pois caracterizam os formatos. Os bits indefinidos são representados pelo carácter '?' e reservam espaço no seio do formato, sendo úteis na verificação da composição do mesmo.

---

[25] A origem da palavra-chave *relocfields*, são as palavras *relocatable fields*.

[26] O *subscript* opt significa que é opcional

[27] *Dont'care*

Cada um destes três elementos deve ser seguido duma gama de bits que indicam a respectiva posição no seio do formato a que pertencem. A Figura 3.14 mostra a descrição em MiADL correspondente aos formatos representados na Figura 3.11, para o caso do processador ARM.

```
iformats <32>{
            ...
            layout dataProcessImmedShift:dpis ( cond[31:28] 0b000[27:25] op[24:21] s[20]
                        rn[19:16] rd[15:12] sa[11:7] shift[6:5] 0b0[4] rm[3:0] ) {…}
             layout dataProcessRgShift:dprs ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16]
                        rd[15:12] rs[11:8] 0b0[7] shift[6:5] 0b1[4] rm[3:0] ) {…}
            layout dataProcessImmed:dpi ( cond[31:28] 0b001[27:25] op[24:21] s[20] rn[19:16]
                        rd[15:12] rotate[11:8] immediate[7:0] ) {…}

            ...
}                                                                                    ARM
```

Figura 3.14: Descrição de formatos de instruções em MiADL

O compilador de MiADL, verifica se o somatório das gamas dos elementos que compõem determinado formato é igual ao respectivo tamanho. Este tamanho é indicado em posições diferentes de acordo com o facto de a arquitectura ser de tamanho variável ou tamanho único, conforme foi referido anteriormente, secção 3.6.3.

É também verificado se cada elemento pertence a uma única gama e se não há sobreposição entre localizações dos elementos. Por exemplo, se num formato tivéssemos dois campos A e B com as seguintes gamas (... *A[31:28] B[28:25]* ...), isto seria assinalado como erro pois o vigésimo oitavo bit daquele formato está a ser alocado a dois campos em simultâneo. Com este tipo de verificações garante-se consistência da descrição e rapidez da mesma. Qualquer descrição feita ao nível de bits é propícia a erros e desta forma a linguagem MiADL ajuda a eliminá-los. Além disso, em MiADL, a descrição é semelhante à informação presente nos manuais das arquitecturas, o que também diminui a propensão para aparecimento de erros de descrição. Se for usada para descrever uma arquitectura do conjunto de instruções de raiz, esta característica ajuda a permitir elaborar a respectiva documentação, pois a facilidade de adequação é biunívoca (manual ⇔ descrição MiADL).

Sempre que um campo está presente em mais do que um formato, por defeito o compilador de MiADL verifica se esse campo está localizado nas mesmas posições relativas daqueles formatos. Isto é usado para explorar uma regularidade encontrada em ISAs correntes conforme descrito na secção anterior, podendo no entanto ser flexibilizado também de acordo com a construção apresentada naquela secção.

A definição de cada formato pode conter também um bloco opcional, onde podem ser definidas funções que acedam directamente aos campos definidos daquele formato. Ou seja, o corpo daquelas funções pode manipular os parâmetros de entrada e campos do formato a que pertencem. Estas funções são muito úteis para, por exemplo, calcular argumentos, de acordo com determinado modo de endereçamento, e podem posteriormente ser chamadas na definição do comportamento das operações que estiverem relacionadas com aquele formato. Esta capacidade faz lembrar o encapsulamento de dados que existem em linguagens orientadas a objectos.

Os campos usados no seio das funções definidas neste bloco opcional de cada *layout*, são posteriormente usados pelo processador da linguagem MiADL afim de inferir, adicionando-os, a lista completa de argumentos das referidas funções. Desta forma, o projectista pode usar os campos do *layout* abstraindo-se do número ou da posição que os mesmos ocupam no seio do formato, sem no entanto deixar de ser garantida fiabilidade da descrição. O compilador de MiADL verifica se todas as variáveis no seio das funções são variáveis locais, parâmetros presentes na definição ou campos válidos do *layout*. Ao gerar o código relativo àquela definição de função, a lista de argumentos resultante será composta pelos parâmetros colocados pelo projectista e pelos campos válidos do *layout* nela usados, precedidos dos respectivos tipos de dados. Esta é uma das características únicas da linguagem MiADL, quando comparada com as demais ADLs referidas no capítulo anterior. Esta inferência de argumentos permite por um lado facilitar a descrição e por outro evitar erros.

Para cada formato é ainda possível declarar um *alias* para o respectivo nome. Na Figura 3.14, após aquela definição o segundo formato pode ser referenciado como *dataProcessRgShift* ou apenas como *dprs*. Esta possibilidade permite tornar as descrições mais simples porque permite simplificar nomes que podem ser longos e que são por vezes usados em várias partes da descrição. Simultaneamente, permite usar em cada formato o nome que o mesmo possui no manual do processador.

3.6.3.3 GRUPOS DE CAMPOS

A linguagem possui uma construção, cuja palavra-chave é ***gpfield,*** que permite uma forma de tratar aglomerados de campos que estejam dispersos pelo formato. Na Figura 3.15 é apresentada a sintaxe desta construção da linguagem MiADL.

gpfield nome_do_grupo_de_campos {campo_#n,…, campo_#1} **in** {layout_x**,** layout_y**,**…}

Figura 3.15: Sintaxe da declaração de grupos de campos

A definição identifica que campos constituem o aglomerado, bem como, que designação passará a ter e para que formatos é aplicado. O compilador de MiADL verifica para cada grupo de campos, se todos os campos do grupo existem nos formatos constantes da lista que acompanha a respectiva definição. Ao *gpfield* é atribuído automaticamente um tamanho, em número de bits, que é igual ao somatório dos tamanhos dos campos que o compõem. Este tamanho é posteriormente usado para efeitos de verificação, sempre que aquele *gpfield* for usado.

Outras linguagens [118] obrigam a usar formas complexas para conseguir representar o mesmo. Usando a construção *gpfield*, é permitida maior liberdade para descrever as operações, porque os símbolos não estão directamente relacionados com posições seguidas de bits dos formatos; e um símbolo pode corresponder a posições distribuídas pelo formato. A partir da definição, o projectista pode abstrair-se e usar aquele aglomerado de campos de determinado formato, como se de um simples campo se trate, nomeadamente para servir de *opcode* ou operando. Esta construção da linguagem ajuda a tornar as descrições claras, compactas e intuitivas. A definição de grupos de campos é uma das características singulares de MiADL relativamente a outras ADLs.

Por exemplo, onde se pode constatar a utilidade do uso desta construção, é no caso das instruções *Absolute Call* (ACALL) e *Absolute Jump* (AJMP) do MCS8051 [103]. Nestas operações, os bits das posições 0 até 7 são concatenados com os bits das posições 13 até 15 para formar o endereço de página, composto por aqueles 11 bits. Usando a construção *gpfield*, a descrição torna-se simples e intuitiva, conforme se pode verificar no excerto da descrição apresentado na Figura 3.16.

```
iformats {
        …
        layout acalljmp <16> (msb[15,13] J[12] 0b0001[11,8] lsb[7,0]){
                // layout body
        }
        …
        gpfield  pageaddr {msb, lsb} in {acalljmp}
        …
}                                                          MCS8051
```

Figura 3.16: Exemplo da declaração de grupos de campos no MCS8051

Durante a declaração dum *gpfield* é importante a ordem dos campos, uma vez que esta será tida em conta para interpretação dos valores associados aquele *gpfield*. Por exemplo se fosse feita a atribuição pageaddr= 0b1100001010, isso equivale a ter: msb= 0b11 e lsb= 0b00001010. Se na declaração aqueles campos estivessem em ordem inversa, então significaria o mesmo que: msb= 0b10 e lsb= 0b11000010. Isto proporciona a flexibilidade de concatenar facilmente os campos por qualquer ordem, podendo ser útil para fazer *swap* de bits, e depois usar aquele símbolo como se fosse um só campo.

Outro exemplo acontece no caso das operações *load/store* do processador ARM [116]. Os formatos associados às instruções de *load/store* tem vários bits unitários (b (1 para *byte*, 0 para *word*), w (1 para registo base é actualizado, 0 para registo base não é actualizado), l (1 para *load*, 0 para *store*)) cuja combinação dos respectivos valores permite distinguir comportamentos e sintaxe de *assembly* de várias operações. Assim, as combinações destes bits servem de *opcode* para as distinguir, Tabela 3.2. Em MiADL, basta criar um grupo de campos e depois usá-lo como se de um campo uniforme se tratasse, Figura 3.17. Assim, no caso do ARM temos o seguinte significado dos campos *b*, *w* e *l* para os formatos *load/store immediate offset* e *load/store register offset* [116], que está resumido na Figura 3.17. Este grupo de campos, pode posteriormente ser usado como *opcode* daquelas oito operações de *load* e *store*, facilitando a descrição e tornando-a mais simples e natural. Ao gerar o descodificador, a posição dos campos nas máscaras é determinada de forma transparente para quem descreve, o que diminui a possibilidade de erros e acelera o processo de descrição.

Tabela 3.2: Exemplo do significado de um grupo de campos para o processador ARM

Field	Field	Field	Group_Field	Operação
b	w	l	bwl	*Opcode*
0	0	0	000	STR
0	0	1	001	LDR
0	1	0	010	STRT
0	1	1	011	LDRT
1	0	0	100	STRB
1	0	1	101	LDRB
1	1	0	110	STRBT
1	1	1	111	LDRBT

```
iformats <32>{
        …
        layout loadStoreRegOffset:lsr ( cond[31:28] 0b011[27:25] p[24] u[23] b[22] w[21]
                l[20] rn[19:16] rd[15:12] sa[11:7] shifti[6:5] 0b0[4] rm[3:0] ) {...}
        ...
        layout loadStoreImed:lsi ( cond[31:28] 0b010[27:25]  p[24] u[23] b[22] w[21]
                l[20] rn[19:16] rd[15:12] immediate2[11:0]) {...}
        ...
        gpfield bwl {b, w, l} in {loadStoreImed, loadStoreRegOffset}
        ...
}                                                                          ARM
```

Figura 3.17: Exemplo da declaração de grupos de campos

3.6.4 GRUPOS DE OPERAÇÕES

Nesta secção da linguagem MiADL, são exploradas outras duas fontes de variabilidade: variabilidade entre classes de operações e variabilidade entre operações duma mesma classe. Esta secção, representa as características que são comuns a um conjunto de operações. As semelhanças entre as operações permite que seja possível compactar a descrição, uma vez que o que é comum é descrito apenas uma vez. Uma descrição pode ter várias destas secções e a composição de cada uma delas segue a estrutura apresentada na Figura 3.18.

group nome_do_grupo **(**lista_de_formatos**)** <selector> **{**
Comportamento e *assembly* de cada operação do grupo
Comportamento que é comum a todas as operações do grupo
Assembly que é comum a todas as operações do grupo
Definições cujo *scope* seja a secção *group*
}

Figura 3.18: Estrutura da secção para descrição dum grupo de operações

Em termos de sintaxe, cada grupo de operações tem um nome, uma lista de formatos e um campo selector. Na Figura 3.19, é possível observar um excerto da descrição das operações aritméticas do processador RISC, ARM [116], que será usado como exemplo, para ajudar a descrever esta secção de MiADL.

A lista de formatos dum grupo representa os modos de endereçamento que podem ser, ou que são, usados pelas operações que compõem o grupo. O campo selector, é o campo que é comum a todos os formatos admitidos pelo grupo de operações, e cujo valor binário identifica as operações do grupo. Ou seja, o selector é um campo cujo valor permite distinguir as operações do grupo, sendo este valor crucial durante o processo de descodificação das mesmas. O valor do selector é usado para geração do descodificador[28] optimizado, como veremos no capítulo seguinte. Por outras palavras, a combinação de um formato com o valor de determinado selector representa uma operação que é única no universo da arquitectura do conjunto de instruções. Isto faz com que o número máximo de operações diferentes, representadas em determinado grupo, seja igual ao produto do número de formatos do grupo pelo número de operações. Por exemplo, na Figura 3.19, para todas as operações daquele grupo (*add*, *sub*,..) existem, para cada uma delas, três modos de endereçamento diferentes (*dpis*, *dprs* e *dpi*). Para *add* existem três variantes de acordo com o modo de endereçamento, sendo que, apesar de o comportamento ser muito semelhante entre aquelas três variantes, a forma como é determinado um dos operandos é consideravelmente diferente.

O compilador da linguagem verifica se o selector do grupo existe e está definido em todos os formatos admitidos pelo grupo de operações. Este selector pode ser um grupo de campos, caso tenha sido definido para todos aqueles formatos do grupo. Esta funcionalidade é particularmente útil para combinar vários campos não contíguos dos formatos, e usar essas combinações para diferenciar operações afim em termos de comportamento. Um destes casos foi previamente referido, em 3.6.3.3., para o caso de operações *load/store* do ARM.

Cada grupo de operações de MiADL é subdividido em quatro tipos de subsecções, Figura 3.18:

---

[28] *Decoder*

```
…
isa ARM < little, 32> { // Inicio da descrição da arquitectura do conjunto de instruções.

global { … }
resources {…}

iformats <32> {
        ….
        layout dataProcessRgShift:dprs ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16] rd[15:12] rs[11:8] 0b0[7]
        shiftr[6:5] 0b1[4] rm[3:0] ) {
                …
                eswitch<dprs.shiftr> arg2 { 0b00: dprs.lsl_r, 0b01: dprs.lsr_r, 0b10: dprs.asr_r, 0b11: dprs.ror_r }
                map<dprs.shiftr> asmdprs {0b00: "lsl", 0b01: "lsr, 0b10: "asr", 0b11: "ror" }
        }
        …
        eswitch<??cond> condAction { 0b0000: eq, 0b0001: ne, 0b0010: cshs, …,  0b1111: nv }
        map<??cond> cond_sym{ 0b0000: "eq",  0b0001: "ne",  0b0010: "cs/hs",  0b1111: "nv" }
        map<layout> ss { dpis:["R%d %s",dpis.rm, @asmdpis], dprs:["R%d,%s R%d", dprs.rm, @asmdprs, dprs.rs], .. }
        eswitch<layout> arg { dpis: $arg1, dprs: $arg2, dpi: arg3 }
        ….
}

…
group aritmetics (dpis, dprs, dpi) <op> {
        …
        operation sub<0b0010> { //  Comportamento da operação 'sub'.
                behavior (n, Vrn) { Vrn - n }
                asm {"sub"}
        }

        operation add<0b0100> { //  Comportamento da operação 'add'.
                behavior (n, Vrn) { Vrn + n }
                asm {"add"}
        }
        ……

        behavior { //  Comportamento comum às operações do grupo de operações
                var aux: int<64>;
                if ($condAction()) {
                        aux= $opAction(rFile.read(??rn), $arg());
                        rFile.write(??rd, readRange(aux,31,0));
                        if (??s){
                                if (??rd == 15){
                                   CPSR = SPSR;
                                }
                                else {
                                   $updateStatus(rFile.read(??rn), $arg(), aux);
                                }
                        }
                }
        }

        asm(string str) {
                "%s%s%c? R%d, R%d, %s", str, @cond_sym, @sflag, ??rd, ??rn, @ss
        }
        …
        eswitch<??op> opAction{0b0101: adc, 0b0100: add, 0b0010: sub, 0b0011: rsb, 0b0110: sbc, 0b0111: rsc}
        eswitch<??op> updateStatus{ 0b0101: update1, 0b0100: update1, 0b0010: update2, … }
…
}
...

} // Fim da descrição da arquitectura do conjunto de instruções.
                                                                                                                    ARM
```

Figura 3.19: Exemplo de descrição MiADL do ARM

- Várias subsecções para descrever as diversas operações do grupo. Uma por operação do grupo. Em cada uma destas secções descreve-se o comportamento e o *assembly* próprio da operação em causa.
- Uma secção onde é descrito o comportamento comum a todas as operações do grupo. Com pontos em que a variabilidade é assinalada através duma construção própria, i. e., *eswitch*. Os pontos de variabilidade são devidos à operação e ao modo de endereçamento em causa (i.e., formato da instrução).
- Secção onde é possível descrever o *assembly* comum a todas as operações. Também aqui os pontos de variabilidade são devidos à operação e ao modo de endereçamento em causa. Para expressar esta variabilidade é usada uma construção própria, i.e., *map*.
- Definição de funções e outras entidades (ex. definição de *eswitchs* ou *maps*) para apoio à descrição do grupo de operações e cujo *scope* é a própria secção do grupo de operações.

De seguida descrevem-se estas subsecções.

3.6.4.1 OPERAÇÕES

Cada operação do grupo pode ser definida como exemplificado na Figura 3.20. Após a palavra reservada ***operation*** deve ser atribuído um nome que identifica a operação. Posteriormente, deve ser definido o valor do selector para aquela operação, ou seja o *opcode*, que será usado para identificar aquela operação durante o processo de descodificação. Dentro desta subsecção, são definidas diferentes características da operação à qual diz respeito. Assim, a palavra ***behavior*** introduz comportamento específico daquela operação, enquanto que a palavra ***asm*** introduz a sintaxe do *assembly* específica da mesma operação.

```
operation add <0b0100> {
   behavior (n, Vrn) { Vrn + n }
    asm{“add”}
}
                                    ARM
```

Figura 3.20: Exemplo da declaração de uma operação em MiADL

Ao analisar as diferentes operações do grupo, facilmente se verifica a variabilidade entre elas, pois aquela subsecção, *operation*, traduz o que é único em cada uma delas relativamente às demais. O compilador de MiADL verifica se dentro de um grupo existe repetição de mnemónicas ou do valor do selector do grupo entre operações distintas. Caso exista, é assinalado o respectivo erro para evitar redundâncias e ambiguidades da descrição.

3.6.4.2 COMPORTAMENTO COMUM

O comportamento comum às operações do grupo é introduzido pela palavra ***behavior***, Figura 3.19.

Nesta subsecção é descrito o comportamento que caracteriza o grupo de operações, ou seja, o que é comum a todas as operações que o compõem. O comportamento traduz-se num conjunto de instruções, i.e., *statements*, cuja ordem reflecte o fluxo de acções que o simulador seguirá para expressar o referido comportamento. Onde houver variabilidade, devida a comportamento próprio das operações ou devida a especificidades de formatos do grupo, esta é descrita recorrendo a uma construção que permite fazer corresponder determinada informação (valor binário dum campo ou formato de instrução) a um comportamento próprio a que aquela informação deve fazer corresponder. A semântica do *eswitch*, como veremos posteriormente neste capítulo com mais detalhe, associa um campo, grupo de campos ou um formato de instrução a uma função ou à chamada de outro *eswitch*.

Quanto ao que é comum a todas as operações do grupo, também aqui através de análise dos *statements* desta secção da linguagem MiADL, é possível identificar campos que são comuns a todos os formatos. Assim, esses campos são facilmente identificados, uma vez que, ao serem usados devem possuir o prefixo '*??*'. Esta marca faz com que o compilador da linguagem MiADL verifique a respectiva existência em todos os *layouts* admitidos pelo grupo de operações. Além dos campos, podem ser definidas nesta secção variáveis locais, que ao serem usadas não devem ter o prefixo anteriormente referido.

3.6.4.3 *ASSEMBLY* COMUM

Os pressupostos aplicados e discutidos na secção anterior também se aplicam para o caso da descrição do *assembly* das operações dum grupo de operações. Assim, também neste caso as fontes de variabilidade são as mesmas pelo que, a necessidade de fazer corresponder àquela informação o *assembly* próprio de determinada operação, é conseguida usando uma construção própria da linguagem, *map*. A semântica desta construção, faz corresponder ao valor duma chave (campo, grupo de campos ou um formato de instrução) uma entidade associada àquele valor da chave que pode ser uma *string* terminal (por exemplo uma mnemónica, valor dum campo comum a todos os formatos do grupo de operações, um carácter fixo, uma *string*, entre outros.), ou uma *string* não terminal que é resultado da chamada a outro *map*. A sintaxe desta construção é explicada mais adiante neste capítulo.

No que diz respeito à sintaxe da subsecção dedicada ao *assembly* comum a todas as operações do grupo, ela é a seguinte:

```
asm( string str){
            “padrão”, lista_de_substrings
}
```

Figura 3.21: Declaração do *assembly* comum

Após a palavra-chave *asm*, o argumento *str* serve para identificar a mnemónica das operações do grupo no seio da *string* padrão que será composta.

Na secção é definida uma *string* padrão, colocada entre aspas, cujo propósito é definir a ordem e formato com que serão concatenados os elementos da *lista_de_substrings*. Assim, a *string* padrão serve para indicar a ordem com que deve ser formado o *assembly* das operações e também, com que formato de texto deverá cada *substring* ser representada nesse padrão, isto faz relembrar a forma de se compor *strings* usando a conhecida função *printf* ao programar em linguagem C. Também aqui existem disponíveis vários formatadores (i.e., caracteres especiais de controlo de formatação de saída de dados) para compor a *string* resultante, sendo eles os seguintes em MiADL: %s para *string*, %c para caracteres e %i para valor numérico inteiro com sinal e %u para valor numérico inteiro sem sinal.

As *sub-strings*, podem ser: mnemónica da operação (parâmetro de entrada desta secção), campos comuns a todos os formatos do grupo, chamada a funções que retornem *string* ou o resultado da chamada de um *map*, que será explicado numa secção seguinte. Na Figura 3.19 é possível ver um exemplo do respectivo uso, e no qual se pode ter uma visão clara e ampla do *assembly* daquele grupo de operações. Também aqui a linguagem MiADL permite conseguir descrições compactas, similares à informação existente nos manuais e fáceis de verificar.

3.6.4.4 ELEMENTOS DE SUPORTE

No seio do grupo de operações, e exteriormente às secções dedicadas à definição de operações ou de comportamento comum, é possível definir funções cuja finalidade seja o apoio ao comportamento ou *assembly* comum. O *scope* destas funções será apenas o grupo de operações e não são vistas exteriormente. Além de funções de apoio podem também ser definidos *eswitchs* ou *maps*. São exemplos disso, os *eswitches* denominados por *opAction* e *updateStatus*, ambos representados na Figura 3.19.

3.7 CONSTRUÇÕES ESPECIAIS

Conforme foi referido anteriormente, as operações são agrupadas de acordo com características que lhes são comuns. Nos grupos de operações existem secções para descrever o que é específico de cada operação e secções para descrever o que é comum a todas as operações do grupo. Nas secções dedicadas a descrever o que é comum, devem existir "pontos" de variabilidade. Ou seja, uma forma de permitir assinalar os pontos onde no comportamento/*assembly* comum, de acordo com o valor de algum campo ou de acordo com determinado formato, existe variabilidade. Ao substituir esses pontos de variabilidade pela versão relativa ao valor do campo, ou ao nome do formato, obtém-se o comportamento/*assembly* próprio de determinada operação.

De seguida apresentam-se duas construções da linguagem MiADL cujo propósito é permitir lidar de forma compacta, clara e natural, com as fontes de variabilidade presentes nas arquitecturas de conjuntos de instruções. Assim, estes permitem fazer corresponder a campos ou a formatos, características que lhe são próprias de acordo com valores que esses mesmos campos ou formatos possam ter e no âmbito em que estejam a ser chamados. De seguida apresentam-se as construções e respectivas sintaxes.

3.7.1 ESWITCH

Conforme foi referido anteriormente, o maior desafio para poder conseguir representações compactas é lidar com a variabilidade a vários níveis.

Para lidar com a variabilidade em termos de comportamento, a linguagem MiADL possui uma construção para o efeito. A sintaxe desta construção está representada na Figura 3.22.

eswitch <*selector*> *nome_do_eswitch* { *chave_1: item_1,..., chave_n: item_n* }

Figura 3.22: Sintaxe da definição de um *eswitch*

O selector pode ser um campo, um grupo de campos ou a palavra reservada *layout*.

As chaves devem ser todas diferentes, entre si, e compatíveis com a natureza do selector. Ou seja, se o selector for *layout* então as chaves devem ser nomes de diferentes *layouts*, previamente definidos na secção *iformats* de MiADL. Se o selector for um campo, então as chaves devem ser valores válidos, que aquele campo pode ter no contexto em que *eswitch* está a ser definido. O valor possível para ser relacionado com determinada chave pode consistir numa constante, na referência a uma função ou na referência à chamada de outro *eswitch*. A chamada de um *eswitch* é assinalada através do respectivo nome precedido do caracter dólar, i.e., '*$*'. No caso de ser chamado no interior do comportamento comum dum grupo, então se o selector for *layout*, os *layouts* do grupo têm de existir todos como chaves daquele *eswitch*. Se o selector for um campo ou um grupo de campos, i.e., *gpfield*, então aqueles devem existir nos *layouts* todos do grupo de operações. Se tal não se verificar, são assinalados os respectivos erros.

Na descrição que tem servido de exemplo, Figura 3.19, as fontes de variabilidade do grupo *aritmetics* são resolvidas da seguinte forma:

- A variabilidade resultante dos modos de endereçamento possíveis para as operações daquele grupo de operações, e de outros grupos (lógicas e *loads*, [116]), é expressa através do *eswitch* chamado '*arg*', definido na secção *iformats*. Este *eswitch* apresenta como selector a palavra reservada *layout*, ou seja, as chaves são formatos e para cada formato em causa corresponde a chamada duma função ou de outro *eswitch*, conforme a variabilidade intrínseca de cada um daqueles formatos. Por exemplo, para o formato ‘*dprs*’, o *eswitch* ‘*arg*’ representa a chamada de outro *eswitch* chamado ‘*arg2*’. Este último por sua vez corresponde, em tempo de execução, à chamada de uma de quatro funções (*lsl_r*, *lsr_r*, *asr_r ou ror_r*) que é seleccionada de acordo com o valor do campo ‘*shiftr*’, do formato ‘*dprs*’, conforme indicado na definição do *eswitch* ‘*arg2*’. A hierarquia de *eswitchs* será tão mais profunda quanto maior for a variabilidade presente.

- A variabilidade, no caso do ARM, devida ao facto de todas as operações serem condicionais também está expressa na Figura 3.19 através dum *eswitch* chamado ‘*condAction’* e cujo selector é o campo ‘*cond’*, que está presente em todos os formatos que fazem parte da arquitectura do conjunto de instruções daquele processador. Assim, para cada valor daquele campo, i.e. chave, corresponde uma função que testa determinadas *flags* e retorna um booleano. Como o campo ‘*cond*’ tem 4 bits, há 16 funções possíveis, estando representadas na Figura 3.19 apenas algumas por questões de simplificação.
- Num grupo de operações uma outra fonte de variabilidade consiste no comportamento específico das operações que fazem parte desse grupo. Este comportamento diverso, é embutido também no comportamento comum através do recurso a *eswitch*(es), cujo selector do grupo será o campo, ou grupo de campos, que serve de selector do grupo de operações. No caso do exemplo da Figura 3.19, o *eswitch* ‘*opAction*’ é um desses casos. Por exemplo, se o valor da chave, i.e. campo ‘*op*’, for ‘*0b0100*’ então a função que implementa o comportamento da operação *add* será chamada e tornada *inline*.

Refira-se ainda, que em MiADL está também contemplada a possibilidade de introduzir como chave um caracter que assinala a situação de outros elementos. O caracter é ‘_’. Desta forma, e em situações de “alguns *vs* restantes”, é possível de forma fácil caracterizar esses cenários. Neste caso, a validação estática persiste, sendo garantido que não há chaves repetidas.

3.7.2 *MAP*

A variabilidade de comportamentos em determinado grupo de operações é idêntica à variabilidade de sintaxe de *assembly*. Para permitir associar correctamente sintaxe *assembly*, em função do valor de determinado campo ou formato (i.e., *layout),* em MiADL, existe uma construção com semântica similar à do *eswitch* que é dedicado à descrição do *assembly* e que apresenta como diferença o facto de só retornar dados do tipo *string*. A sintaxe desta construção está representada na Figura 3.23.

map <*selector*> *nome_do_map* { *chave_1: item_1,…, chave_n: item_n* }

Figura 3.23: Sintaxe da definição de um *map*

Num *map*, à semelhança do que acontece com o *eswitch*, o selector pode ser um campo, *gpfield*, ou a palavra-chave *layout*. Quanto aos itens correspondentes a determinada chave (i.e., valor do selector), eles podem ser: uma constante, um caracter (assinalado entre plicas), uma *string* (assinalada entre aspas), uma expressão que constrói uma *string* com base nos valores de campos de instrução, uma chamada duma função que retorne *string* ou a chamada de outro *map*. A chamada de um *map* é assinalada através do respectivo nome precedido do símbolo '@'.

As descrições de expressões para construção de *strings*, com base no valor de campos da instrução, obedecem a uma sintaxe própria como a que é apresentada na Figura 3.24.

["padrão"**,** lista_de_substrings **]**

Figura 3.24: Sintaxe para expressões de *assembly*

Os significados do "*padrão*" e da *lista_de_substrings* são semelhantes aos que já foram relatados na secção 3.6.4.3. O *map* com nome *ss* definido no exemplo da Figura 3.19 serve para mostrar o uso deste tipo de expressão.

No caso de ser chamado no âmbito do *assembly* comum dum grupo, e se o selector for *layout*, os *layouts* do grupo têm de existir todos como chaves na definição daquele *map*. Se o selector for um campo, ou um grupo de campos (i.e. *gpfield*), então aqueles devem existir nos *layouts* todos do grupo de operações. Caso tal não aconteça, também aqui serão assinalados os respectivos erros. Ou seja, as regras de verificação são idênticas às que são realizadas para verificação da consistência da descrição dos *eswitches*.

Esta construção da linguagem MiADL permite lidar com descrições de arranjos complexos de nomenclaturas de registos, como a do Anexo A.6 de [87], de forma natural. Assim, em MiADL, aquele conjunto de registos poderia ser descrito usando um *map*, cujas chaves seriam os índices dos registos e os itens seriam as respectivas mnemónicas, tal como exemplificado na Figura 3.25.

```
    map regSintaxe <32> { 0b00000: “$zero”, 0b00001: “$at”, 0b00010: “$v0”, ..., 0b00101:
“$a1”, ... , 0b01000: “$t0”, ... , 0b11111: “$ra” }
```

Figura 3.25: Sintaxe para arranjos de registos complexos em *assembly*

Note-se aqui a diferença na definição deste *map* relativamente ao apresentado na Figura 3.23. Esta sintaxe deve ser usada em MiADL para a definição de *maps* que possam ser partilhados por vários campos/*layouts*. Assim, a chamada deste tipo de *map* difere da anterior e deve possuir a indicação do campo/*layout* a que corresponde. A sintaxe para a chamada deste tipo de *map* é:

@ nome_do_*map* <selector>

Outras linguagens adicionaram construções especiais para o efeito [119], no entanto, aqui mantém-se a mesma lógica em termos da semântica dos mesmos.

3.8 NOMES E *SCOPES* EM MiADL

Nome é uma mnemónica usada para representar algo. Em linguagens, estes são identificadores, ou *tokens* alfanuméricos, que permitem identificar variáveis, constantes, operações, tipos e outros elementos da linguagem usando identificadores simbólicos, evitando o recurso a conceitos de mais baixo nível, tais como endereços [120]. Os nomes são essenciais em termos de abstracção. Estes permitem associar nomes a fragmentos de código, que podem ser interpretados posteriormente em termos da função ou propósito que realizam, e não em termos de como aquela função é implementada. Escondendo detalhes irrelevantes, a abstracção reduz complexidade, permitindo que o programador em determinado instante se foque apenas nas fracções do programa que interessam. Funções são exemplo de abstracções de controlo que permitem que o programador esconda código complexo por detrás de uma interface simples.

A relação mútua, *binding*, entre nomes e o que os mesmos representam é bastante importante e está relacionada com *scopes*. A região do texto de um programa na qual a relação entre o nome e o que o mesmo representa está activa, tem a designação de *scope* [120].

Muitas linguagens usam *scopes*, sendo a linguagem *C* um exemplo conhecido. Em *C* um novo *scope* é introduzido de cada vez que se entra na definição de uma subrotina, sendo criados *bindings* relativos a objectos locais e desactivando *bindings* de objectos globais, que ficam escondidos por objectos locais que tenham o mesmo nome. Esta manipulação de *bindings* pode parecer que se faz em tempo de execução, mas na verdade pode processar-se durante a compilação. É o que se passa relativamente à linguagem *C*, na qual é possível saber que nomes estão relacionados com determinados objectos, em determinadas secções do programa, através de análise textual do programa. Neste trabalho a sequência de *scopes* que podem ser examinados, para encontrar o *binding* corrente para determinado nome, está relacionada com a estrutura de MiADL e o significado das respectivas secções, conforme descrito anteriormente. Assim, em MiADL existe uma estrutura de *scopes* que permite, por um lado, tornar as descrições mais coerentes, e por outro, permite que sejam validadas diversas condições o que a torna mais robusta quando comparada com outras ADLs. Esta propriedade de MiADL, é uma das singularidades da linguagem no âmbito do trabalho em que a mesma se enquadra, ou seja ADLs, e tal se deve à existência de blocos bem definidos na respectiva estrutura.

A Figura 3.26 apresenta um esquema em que se representam, por blocos, os *scopes* em MiADL. Como veremos, esta linguagem possui uma hierarquia que além de *scopes* que envolvem outros *scopes*, também há *scopes* que "herdam" de outros. Com ajuda da Figura 3.26 será explicada, de forma idêntica à que é usada em [121], esta relação de *scopes*, que foi projectada tendo em conta as variabilidades existentes nas arquitecturas de conjunto de instruções, expostas em 3.1.

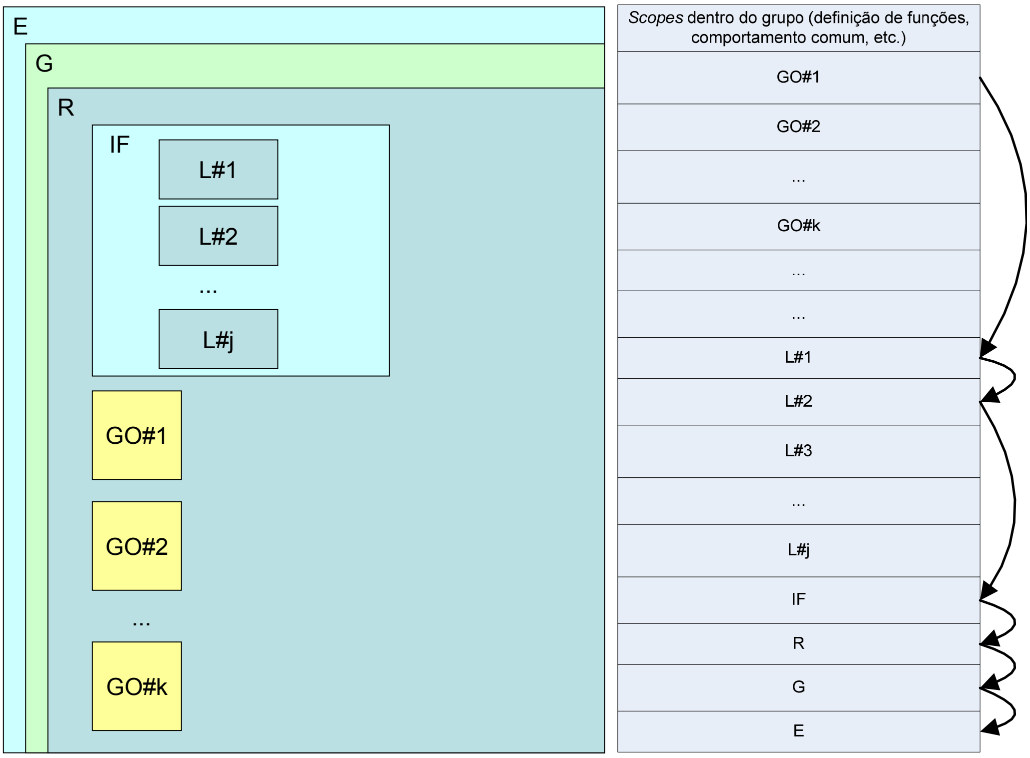


Figura 3.26: *Scopes* de MiADL

Em MiADL cada bloco explicado anteriormente possui um *scope* próprio e pode "ver" variáveis, funções ou *eswitchs* de outros *scopes* circundantes, de acordo com as regras que serão apresentadas a seguir.

Assim, a zona de declarações externas, a secção de recursos, a secção de *iformats*, cada uma das declarações de *layout* (porque têm corpo), definição de funções (porque têm *body*) e os grupos de operações, são exemplos de blocos em MiADL. Note-se que alguns blocos se encontram definidos dentro de outros, ex. *layouts* estão dentro do *iformats*. Em MiADL não se podem definir funções dentro de outras funções, o que de resto se passa com outras linguagens conhecidas, tais como *C/C++*.

Em MiADL existe uma zona para declaração de código externo. Na Figura 3.26 essa região é identificada pela letra E. As declarações feitas nesta secção de MiADL são visíveis por todas as demais, ou seja, são globais.

Existe também uma secção da linguagem para definições, representada por **G** na Figura 3.26, que poderão ser visíveis pelas demais secções, excepto **E**. As definições feitas em **G** podem chamar funções declaradas em **E**.

A zona de descrição de recursos, assinalada com um **R** na Figura 3.26, por sua vez, também é visível a todas as demais dentro do ISA. Uma vez que esta secção retrata o hardware do processador alvo, ela, como vimos anteriormente, representa o estado do sistema e por isso, os nomes das variáveis declaradas/definidas nesta secção não podem ser usados noutras declarações – excepto no interior de funções - precisamente para mantê-las sempre acessíveis a partir de qualquer secção de MiADL.

A secção *iformats,* por sua vez, é um bloco de MiADL que comporta outros blocos que lhe são internos, ou seja, os *layouts*/formatos. As declarações neste bloco, **IF** na Figura 3.26, correspondem aos *layouts*, bem como a definição de funções, *eswitchs* ou *gpfields*. Os sub-blocos *layout*, um por cada formato, têm por defeito como variáveis daqueles *scopes* os campos do formato. Se dentro dos *layouts* houver definição de funções, então estes novos blocos internos aos *layouts* corresponderão a novos *scopes*. Nestes scopes, as variáveis serão os parâmetros de entrada daquelas funções, as variáveis que forem declaradas no seu corpo e os campos do *layout* que estiverem visíveis no interior delas. Desde que não seja declarada uma variável com o mesmo nome que o de um campo daquele formato, uma vez que tal declaração "esconderia" o campo.

No caso dos grupos de operações a relação de *scopes* apresenta uma característica diferente, inclusive de linguagens de programação conhecidas, e que foi projectada para ajudar a compactar a descrição e lidar com variabilidades em termos de formatos/modos de endereçamento que determinadas operações podem ter. Assim, no âmbito dum grupo de operações os blocos existentes correspondem ao comportamento comum, ao *assembly* comum, ao comportamento de cada operação, ao *assembly* de cada operação e funções que sejam definidas no interior do grupo de operações. As entidades que são visíveis por todo o grupo serão todas as de **E**, **G**, **R**, **IF** e dos *layouts* que fazem parte do grupo, bem como as definidas no âmbito do mesmo. Ou seja, funções, *eswitchs* ou *maps* definidos no grupo de operações. Em cada bloco acrescem as que forem declaradas localmente.

Por exemplo se o grupo de operações GO#1, Figura 3.26, tiver como *layouts* do grupo L#1 e L#2, então a hierarquia de *scopes* será a seguinte: *scope* do bloco em análise (ex. comportamento comum), GO#1, (L#1 e L#2), IF, R e finalmente G e E, lado direito da Figura 3.26. Seria esta a sequência de procura da definição de determinado identificador na tabela de símbolos. Se na secção relativa ao comportamento comum do grupo de operações, um identificador fosse assinalado como devendo ser um campo existente em todos os formatos do grupo e ao percorrer a sequência anteriormente referida fosse detectado que aquele existia em apenas um dos *layouts,* tal situação resultaria no assinalar de um erro. Uma vez que o que é particular de cada *layout*, por exemplo do layout #x, não deve ser usado no grupo de operações mas sim apenas aquando da definição daquele, ou seja no bloco L#x da Figura 3.26, com 1<x<j no caso do exemplo. O que é comum a todos os *layouts* do grupo pode ser usado aquando da definição das características comuns do grupo. Com esta estrutura da linguagem, e respectivos *scopes*, consegue-se validar muitos erros de descrição e garantir uma descrição consistente de acordo com as premissas e objectivos de MiADL.

Para realçar esta relação de *scopes*, recorramos novamente ao exemplo da Figura 3.19, mas agora com informação de linha e que repetimos por questões de facilitar sua consulta na Figura 3.27. Assim, por exemplo, o identificador *rd* usado nas linhas 41 e 43, quando encontrado, será pesquisado se se trata de uma das variáveis locais do bloco onde está, comportamento comum do grupo de operações com nome *aritmetics*, e como não se trata de uma variável local a pesquisa prosseguirá. No passo seguinte, será procurada a existência de *rd* em todos os *layouts* do grupo, ou seja em *dpis*, *dprs* e *dpi*. Se existir nos três, como é o caso, então será assumido que se trata de um campo dos formatos do grupo de operações. Caso não estivesse em todos os *layouts*, a busca continuaria pelas secções *iformats*, *resources*, global e finalmente *extern*.

Quanto à atribuição de nomes, refira-se que em MiADL não é permitida a atribuição de nomes iguais a *layouts,* ou grupos de operações, diferentes. Outra característica, consiste no facto de também não ser permitido o uso dum mesmo nome para identificar *eswitchs*, *maps* ou funções diferentes ao longo do caminho de *scopes* com sequência: *extern*, *global*, *resources*, *iformats*, *layouts* ou extern, global, *resources*, *iformats*, grupo de operações. Estas regras na atribuição de nomes tem por objectivo evitar incoerências e serve para tornar mais claras as descrições e os espaços de nomes (i.e., *namespaces*). A declaração duma variável local oculta, i.e., esconde, variáveis com o mesmo nome definidas/declaradas em *scopes* envolventes.

```
…
 isa ARM < little, 32> { // Inicio da descrição da arquitectura do conjunto de instruções.

global { … }
resources {…}

iformats <32> {
        ….
                layout dataProcessRgShift:dprs ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16] rd[15:12]
                                                    rs[11:8] 0b0[7] shiftr[6:5] 0b1[4] rm[3:0] ) {
                        …
                eswitch<dprs.shiftr> arg2 { 0b00: dprs.lsl_r, 0b01: dprs.lsr_r, 0b10: dprs.asr_r, 0b11: dprs.ror_r }
                map<dprs.shiftr> asmdprs {0b00: "lsl", 0b01: "lsr, 0b10: "asr", 0b11: "ror" }
                }
        …
        eswitch<??cond> condAction { 0b0000: eq, 0b0001: ne, 0b0010: cshs, …,   0b1111: nv }
        map<??cond> cond_sym{ 0b0000: "eq",  0b0001: "ne",  0b0010: "cs/hs",  0b1111: "nv" }
        map<layout> ss { dpis:["R%d %s",dpis.rm, @asmdpis], dprs:["R%d,%s R%d", dprs.rm, @asmdprs, dprs.rs], .. }
        eswitch<layout> arg { dpis: $arg1, dprs: $arg2, dpi: arg3 }
        ….
}

…
group aritmetics (dpis, dprs, dpi) <op> {
        …
        operation sub<0b0010> { //  Comportamento da operação ‘sub’.
                behavior (n, Vrn) { Vrn - n }
                asm{“sub”}
        }

        operation add<0b0100> { //  Comportamento da operação ‘add’
                behavior (n, Vrn) { Vrn + n }
                asm{“add”}
        }
        ……

        behavior { //  Comportamento comum às operações do grupo de operações
                var aux: int<64>;
                if ($condAction()) {
                        aux= $opAction(rFile.read(??rn), $arg());
                        rFile.write(??rd, readRange(aux,31,0));
                        if (??s){
                                if (??rd == 15){
                                  CPSR = SPSR;
                                }
                                else {
                                  $updateStatus(rFile.read(??rn), $arg(), aux);
                                }
                        }
                }
        }

        asm(string str)  {
                "%s%s%c? R%d, R%d, %s", str, @cond_sym, @sflag, ??rd, ??rn, @ss
        }
        …
        eswitch<??op> opAction{0b0101: adc, 0b0100: add, 0b0010: sub, 0b0011: rsb, 0b0110: sbc, 0b0111: rsc}
        eswitch<??op> updateStatus{ 0b0101: update1, 0b0100: update1, 0b0010: update2, … }
        …
}
…
} // Fim da descrição da arquitectura do conjunto de instruções.
```

Figura 3.27: Excerto da descrição em MiADL de ARM

A estrutura da linguagem, nomeadamente a hierarquia de *scopes* e as regras de atribuição de nomes permitem a implementação de validação estática robusta – através dum conjunto vasto de validações - que por sua vez tornam possível conseguir descrições de arquitecturas de conjuntos de instruções de processadores alvo, robustas, compactas, claras e sem redundâncias. Além disso permite simplificar o processo de descrição porque a definição de um bloco num determinado *scope* envolvente pode ser usada em *scopes* internos. Assim, é possível uma abordagem modular e o uso de módulos pré-validados de forma coerente e sujeita a validação estática. Esta propriedade é tanto mais importante quanto é reconhecido que um desafio importante na construção de blocos de software consiste na divisão de esforços entre programadores de tal forma que seja possível a progressão do trabalho em múltiplas frentes simultaneamente [120]. Esta modularização de esforços depende fortemente da noção de "*ocultação de informação*", que tornam entidades ou algoritmos invisíveis, sempre que possível, às porções do programa que não necessitam deles. Uma modularização adequada permite reduzir a "carga cognitiva" no programador através da minimização da informação necessária para entender determinada porção do programa. Numa estrutura bem projectada a interface entre módulos deve ser tão simples quanto possível, para que qualquer alteração desejada seja contemplada dentro de determinado módulo. Esta característica é essencial, uma vez que a manutenção - correcção de *bugs* ou melhorias – consome grande proporção de tempo durante processos de desenvolvimento.

Além da redução da carga cognitiva, a ocultação de informação apresenta outras vantagens. Primeiro, reduz o risco de conflitos de nomes – com menos nomes visíveis, existe menos probabilidade de que a introdução de um novo nome entre em conflito com os já existentes. Segundo, protege a integridade de abstracção de dados porque qualquer tentativa de aceder a entidades fora dos módulos (ex. função) a que as mesmas pertençam, provocará erros assinalando a não definição dos mesmos. Terceiro, apoia a localizar erros, uma vez que é possível identificar o *scope* onde determinado erro ocorre.

Estas vantagens não estão presentes nas ADLs apresentadas no segundo capítulo desta tese.

3.9 PREFIXOS DE NOMES

Para que o projectista, quem descreve em MiADL, possa ter comando sobre quais os passos que devem ser seguidos para validar determinado campo no seio de uma construção ou de uma secção, a linguagem possui dois prefixos cujo significados promovem determinadas validações. Assim, sempre que o projectista pretenda que seja verificado se um campo pertence a todos os formatos (i.e. *layouts*) que possam existir no *scope* onde o mesmo está a ser usado, deve preceder esse campo pelo prefixo '**??**'. Isto é, qualquer campo que tenha como prefixo aquele par de pontos de interrogação faz com que seja verificado se em todos os formatos, possíveis para o *scope* onde está a ser usado, o mesmo existe e está definido como campo. Por exemplo, na linha 42 da Figura 3.27, existe um campo **??s**. Neste caso é verificado se o campo **s** existe em todos os formatos possíveis para o grupo de operações, que neste caso são *dpis*, *dprs* e *dpi*. O mesmo pode acontecer também na secção *iformats* ou no caso de selectores de *eswitchs*.

Outro prefixo de nomes, pode ser usado no seio da definição de um formato. Assim, ao definir um *eswitch* relativo a um determinado formato – no seio do corpo do formato – para cada item do *eswitch* a entidade correspondente à chave (i.e., valor de selector do *eswitch*) deve ter como prefixo o nome do formato. Este prefixo será usado pelo compilador da linguagem MiADL para verificar se aquela entidade existe definida no âmbito do *scope* do formato. O mesmo acontece com o uso de campos no seio de funções que sejam definidas no corpo dos formatos. Ou seja, também nesse caso o uso dos campos deve ter como prefixo o nome do formato seguindo a sintaxe nome_do*formato.campo*.

Os prefixos são usados para fins de validação estática e não têm influência no código resultante do processo de geração.

3.10 DECLARAÇÃO DE VARIÁVEIS E TIPOS DE DADOS

Todas as variáveis usadas em MiADL devem ser declaradas antes de ser usadas. Assim, a sintaxe para declarar variáveis é a seguinte:

var lista_de_variáveis **:** tipo **;**

Figura 3.28: Sintaxe relativa à declaração de variáveis sendo o nome das variáveis, presentes na lista de variáveis, separados por vírgulas. Esta declaração de variáveis pode ser efectuada no corpo da definição de funções ou no comportamento comum de um grupo de operações.

Uma vez que os tipos nativos de dados inteiros da linguagem *C* ou *C++* são fortemente dependentes e limitados ao tamanho da palavra da máquina hospedeira, i.e., onde é executado o simulador, em MiADL foram considerados tipos de dados inteiros genéricos. Assim, a linguagem MiADL suporta o uso dos seguintes tipos:

- *void*;
- *boolean*;
- *string*;
- int<*n*>, sendo *n* o número de bits;
- uint<*n*>, sendo *n* o número de bits.

Estes tipos permitem a geração de simuladores funcionais *bit-true*, i.e., nos quais os valores produzidos durante simulação são iguais bit a bit aos que são produzidos por hardware.

Está prevista a implementação de conversão implícita de tipos aquando da análise do tipo de instruções do programa, i.e., *statements*. Assim, algumas conversões que poderão vir a ser permitidas são:

- int<n> para int<m> ou uint<m>, sempre que n <= m;
- uint<n> para int<m> ou uint<m>, sempre que n <= m;
- int<n> ou uint<n> para *string*.

A inclusão dos tipos *double* e *float* está igualmente prevista como trabalho futuro.

3.11 OPERADORES

Fazem parte da linguagem MiADL vários operadores que podem ser usados para descrever operações que podem ser unárias, binárias ou ternária. Estes operadores contemplam operações aritméticas, lógicas, de atribuição, relacionais e *bitwise* (i.e., cujas operações são efectuadas bit a bit).

Tanto os operadores relacionais como os lógicos têm precedência inferior à dos operadores aritméticos ou *bitwise*. Sendo os operadores de atribuição os de menor precedência. As operações são avaliadas da esquerda para a direita quando na presença de operadores de igual precedência. Manteve-se a lógica presente em linguagens de programação tais como, *JAVA* ou *C*, para que o uso seja intuitivo. No Anexo 2 é apresentada uma tabela, com a relação completa dos operadores de MiADL, na qual os mesmos são classificados quanto ao tipo (número de operandos), precedência, e ordem de análise. Estes operadores possuem características idênticas às existentes nas linguagens *C/C++*, por serem estas as representações intermédias geradas a partir de MiADL.

O uso de parênteses, '(' e ')', permite compor a ordem de preferência, uma vez que o que estiver entre parênteses terá maior precedência.

Uma vez que MiADL permite chamar funções externas, acresce que quando definindo funções em *C++* o projectista terá naquele ambiente a flexibilidade de recorrer a operadores daquela linguagem de programação, operadores estes que não estejam contemplados em MiADL, para descrever casos especiais.

3.12 COMENTÁRIOS E ANOTAÇÕES

Em MiADL, apesar das descrições serem compactas e claras, é possível também usar dois tipos de comentários que podem ser colocados em qualquer parte de MiADL. É possível comentar uma linha completa colocando "//" no início da linha. Caso seja desejado comentar um bloco de código, então o bloco deve ser delimitado por "/*" antes do inicio do bloco e "*/" após o final do bloco. Esta sintaxe é idêntica à que é usada em linguagens de programação conhecidas, *JAVA* ou *C*, tornando-se intuitiva a sua utilização.

Uma vez que, as descrições duma ADL podem ser partilhadas entre projectistas de diferentes áreas (por exemplo programadores, analistas de sistemas, engenheiro de hardware, etc.), esta funcionalidade da linguagem permite anotar as descrições e assim evitar percas de ritmo de trabalho, ou seja, ajudar a reduzir tempos gastos durante o processo de desenvolvimento.

3.13 CONTROLO DE FLUXO

Em qualquer linguagem as instruções (i.e., *statements*) de controlo de fluxo, para expressar os passos pelos quais a execução deve ser conduzida, são importantes para conseguir descrever o comportamento das instruções do processador alvo e o consequente efeito durante a execução do simulador.

Actualmente MiADL permite o uso das instruções ***if-then-else*** e ***for***, cujas sintaxes são apresentadas na Figura 3.29.

```
if( condição ) {
        statements
}
elif( condição ) {
        statements
}
...
else(condição) {
        statements
}
```

Nota: Os ramos *elif* e *else* são opcionais.

```
for ( início ; condição de paragem ; expressão[1] ) {
    statements
}
```

1 – Campo de preenchimento opcional

Figura 3.29: Sintaxe de estruturas de decisão de MiADL

3.14 FUNÇÕES

Em MiADL é possível definir funções e também declarar funções externas (na secção ***extern***). O corpo das funções definidas em MiADL faz parte da descrição enquanto que o corpo de funções externas é definido em ficheiros de código *C/C*++ externos.

A definição de uma função em MiADL contém um nome, um tipo de retorno, uma lista de argumentos e um corpo, Figura 3.30. No corpo podem existir declaração de variáveis internas e outros *statements*. Os *statements* podem aceder aos argumentos da função, a variáveis locais da função ou a outras variáveis/funções que sejam visíveis, de acordo com as regras de *scopes* de MiADL. O resultado das funções definidas em MiADL pode ser retornado usando a construção *return*, ou através de argumentos alteráveis, i.e. por referência. As funções devem ser chamadas, respeitando as regras de *scopes* da linguagem. Chamadas ilegais são assinaladas como erro.

No caso de funções externas, a declaração contém após a palavra-chave *extern*, o nome da função, tipo de retorno e os argumentos, Figura 3.31. Estas são declaradas na zona *extern* e por consequência visíveis de qualquer parte da descrição em MiADL. Também neste caso podem haver argumentos alteráveis, i.e. passagem de argumentos por referência.

Para declarar um argumento como sendo alterável, ou seja, que pode ser alterado pelo corpo da função (interna ou externa), esse argumento deve ser assinalado através da inclusão do sufixo '&'. Os argumentos que não possuam o referido sufixo são considerados apenas para leitura por parte do corpo da respectiva função.

tipo nome_da_função **(** lista_de_argumentos)**{**
    Corpo da função, composto por um ou mais *statements*
}

Figura 3.30: Sintaxe para definição de função em MiADL

extern tipo nome_da_função **(** lista_de_argumentos **);**

Figura 3.31: Sintaxe para declaração de função externa em MiADL

Os tipos de dados usados como retorno das funções e nas listas de argumentos, obedecem ao exposto na secção 3.10.

3.15 PALAVRAS-CHAVE DA LINGUAGEM

A linguagem MiADL possui um número reduzido de construções o que permite que o projectista possa facilmente adaptar-se à mesma. A Tabela 3.3, apresenta as palavras-chave, da linguagem MiADL.

A linguagem MiADL é do tipo *case-sensitive*.

Tabela 3.3: Palavras-chave da linguagem MiADL

iformats	*layout*	*if*	*for*	*return*
elif	*else*	*resources*	*reglayout*	*behavior*
regfile	*memory*	*extern*	*global*	*asm*
eswitch	*map*	*group*	*??*	*gpfield*
int	*uint*	*bool*	*string*	*in*
relocfields	*init*	*big*	*little*	*programcounter*

3.16 CONCLUSÃO

Pretendeu-se com este capítulo dar uma perspectiva global da linguagem MiADL, mostrando que esta foi projectada tendo por base fontes de variabilidade e de regularidade presentes nos ISAs de processadores actuais, que entretanto foram estudados e posteriormente modelados na nova linguagem.

Para expor as contribuições da linguagem, descreveu-se a estrutura e as construções que permitem lidar com as premissas focadas anteriormente. Ao longo do capítulo foram mencionadas também várias verificações que são feitas pelo compilador da linguagem, e que são possíveis devido à sintaxe da mesma. Para melhor apresentar a linguagem, foram usados, a título de exemplo, trechos de modelos de processadores entretanto modelados em MiADL.

Neste capítulo foi também focada a inferência de argumentos e a hierarquia de *scopes* e como estas duas características simplificam e tornam verificáveis as descrições. A densidade das descrições conseguidas, foi também realçada, nomeadamente no que se refere às características da linguagem que tornam possível conseguir descrições concisas, claras e sem redundâncias. A semelhança com a informação geralmente disponível nos manuais dos processadores é um dos factores que permite conseguir descrições céleres.

Em MiADL as instruções são agrupadas de acordo com o comportamento comum, que por consequência permite ter grupos que reúnem operações que partilham as mesmas unidades funcionais no que ao hardware diz respeito.

A forma como a carga cognitiva é abordada em MiADL permite que seja seguido o princípio de “descrever uma vez um bloco, usar várias vezes”, o que permite que ao verificar partes de código claramente definidas na sua estrutura, evitar repetição de erros de descrição ou redundâncias. Isto além de tornar as descrições compactas, torna-as mais legíveis o que faz com que durante a fase de exploração das arquitecturas o tempo de desenvolvimento seja menor. Por outro lado, a linguagem possui um conjunto de operadores e construções que a tornam estruturada e permitem que seja verificada como um todo.

Foi apresentado também um modelo para descrição compacta do *assembly* das arquitecturas de instruções alvo. Para conseguir a compactação das descrições foram apresentadas várias construções que tal como para o caso do comportamento das instruções, também neste caso foram projectados tendo por base as fontes de variabilidade detectadas nos ISAs. Assim, para lidar de forma compacta com as fontes de variabilidade foram apresentadas duas construções que permitem fazer corresponder informação relativa à codificação da instrução nos correspondentes comportamentos ou *assembly* respectivamente. A linguagem apresentada permite representar informação suficiente para a geração de simuladores funcionais e desassembladores.

Num dos próximos capítulos da tese, capítulo 5, é feita a comparação de MiADL com outras linguagens relativamente a descrições de várias arquitecturas do conjunto de instruções de processadores conhecidos. No mesmo capítulo, são também apresentados resultados de desempenho de simuladores gerados a partir destas descrições.

CAPÍTULO

4

Geração de Simuladores Redireccionáveis

4.1 INTRODUÇÃO

Neste capítulo descrevem-se os principais blocos da infra-estrutura de geração de simuladores a partir de descrições feitas em MiADL, e os passos seguidos para implementar aqueles blocos. Ao longo do capítulo serão usados alguns exemplos tendo por base modelos de processadores descritos em MiADL. A ênfase será dada à geração de simuladores compilados.

4.2 GERAÇÃO DE SIMULADORES REDIRECCIONÁVEIS

Um simulador redireccionável pode ser visto como sendo composto por dois componentes: por um lado, um modelo genérico que representa as propriedades que são comuns a uma família de processadores e por outro lado, a descrição do processador alvo que se pretende simular. Tendo como ponto de partida os referidos modelos o gerador de simuladores redireccionáveis deve gerar simuladores específicos para os processadores alvo.

Conforme foi referido na introdução desta tese, um dos objectivos, além da proposta do modelo de descrição compacta, é a geração de simuladores redireccionáveis baseados na técnica de simulação compilada. Nas secções seguintes apresentam-se os blocos principais usados nos processos de implementação do compilador da linguagem MiADL e de geração de simuladores.

4.3 SIMULAÇÃO COMPILADA

No Capítulo 2, foram expostas as características da simulação compilada e apresentadas as diferenças da referida técnica para com outras técnicas de simulação. Uma das características referidas destes simuladores é o facto de serem específicos para a aplicação e para o processador alvo. Sendo assim, para a geração de simuladores redireccionáveis deste tipo, o modelo do processador alvo e o código binário da aplicação a simular são entradas para o gerador de simuladores compilados. Quanto ao tipo de simulação compilada, nesta tese optou-se pela técnica de simulação compilada estática recorrendo à conversão para código binário hospedeiro, através de uma linguagem intermédia, o *C++*. Em relação às técnicas que fazem conversão directa para código do hospedeiro a simulação neste caso, pode tornar-se mais lenta, mas por outro lado, permite portabilidade e é mais simples o processo de geração.

A representação na forma intermédia, *C++*, do simulador – gerada a partir de informação relativa à aplicação e ao processador alvo – é posteriormente compilada, usando um compilador de *C++*, para código objecto do hospedeiro que for usado para executar o simulador gerado. Na Figura 4.1, apresenta-se a sequência de passos seguidos durante o processo de geração de simuladores usado nesta tese.

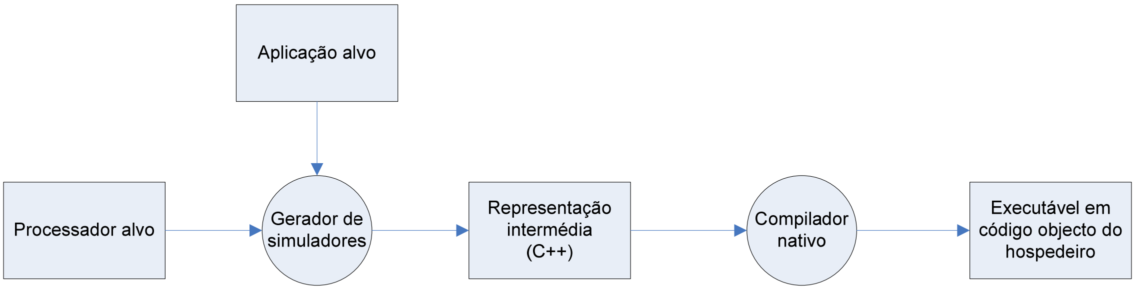


Figura 4.1: Geração de simulador compilado

Nas seguintes secções, apresentam-se a infra-estrutura que permite gerar a representação do processador alvo, o carregamento das aplicações alvo, bem como componentes usados para geração da representação intermédia do simulador. Quanto ao compilador nativo foi usado o *g++* [122], versão 4.1.2, e os ensaios foram feitos no ambiente do sistema operativo Linux, *Ubuntu* versão 6.10, *Edgy* [123].

4.4 INFRA-ESTRUTURA DE GERAÇÃO

A Figura 4.2 apresenta um diagrama de blocos mais detalhado da infra-estrutura implementada para geração de simuladores, e outras ferramentas, tendo como origem modelos de processadores descritos em MiADL. Nas próximas secções deste capítulo será explicado cada um destes blocos. Na Figura 4.2, os blocos a tracejado correspondem a trabalho futuro, ou que se encontra em desenvolvimento, para permitir geração de outras ferramentas além de simuladores, com base em modelos descritos em MiADL.

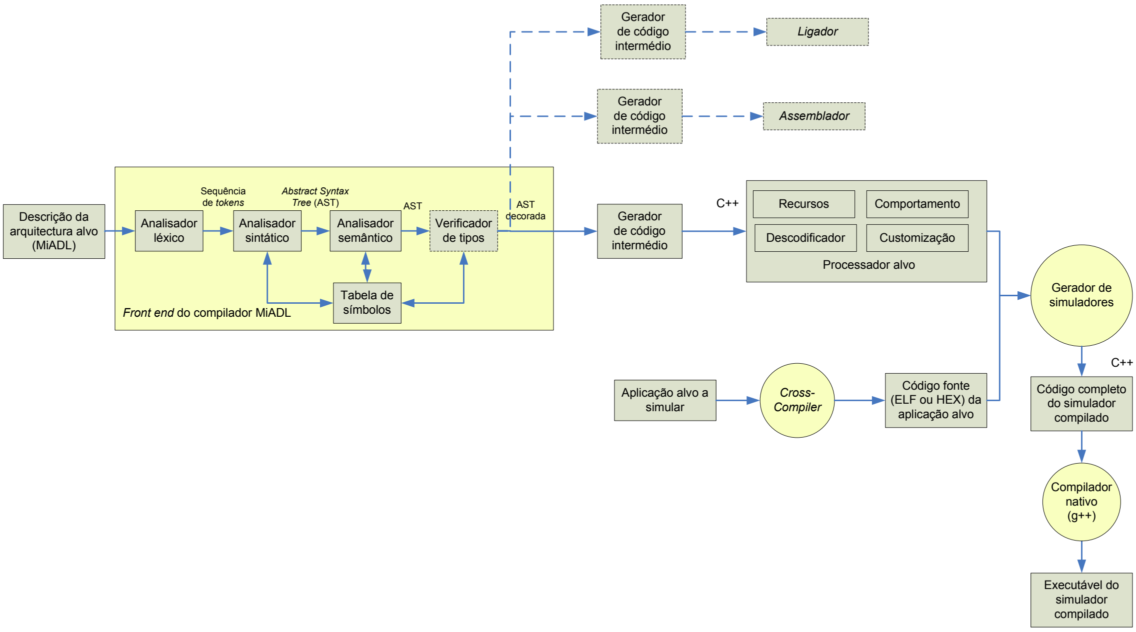


Figura 4.2: Diagrama de blocos da infra-estrutura de geração de ferramentas de software

4.4.1 GERAÇÃO DO MODELO DE UM PROCESSADOR ALVO

Conforme representado na Figura 4.1 para gerar o simulador compilado é necessário ter a representação do modelo do processador alvo que se pretende simular, além da aplicação. Esta representação do processador alvo é extraída a partir da descrição feita usando MiADL. A conversão da descrição em MiADL para um modelo específico do processador é feita pelo *front end* do compilador MiADL. Este *front end* é composto por um analisador léxico, um analisador sintáctico e vários *treewalkers* [124] que transformam a descrição MiADL em estruturas de dados que servem de fonte para os geradores de representações intermédias, Figura 4.3. Durante as fases pertencentes ao *front end* do compilador MiADL as descrições são alvo de verificações para garantir consistência dos modelos de processador gerados. Estes testes, conforme foi referido no Capítulo 3, têm a ver com verificação de tipos, garantia de consistência da codificação (i.e. *coding*) das instruções, eliminação de redundâncias, verificação de declarações de símbolos, etc. Todas estas etapas se processam sobre a gramática da linguagem cuja representação EBNF [125] consta do Anexo 1. A garantia de consistência da representação é fundamental para optimizar a geração e evitar ambiguidades nos modelos que são produzidos pelos geradores de código, um especializado para cada ferramenta (simulador, desassemblador, assemblador, ligador), Figura 4.4.

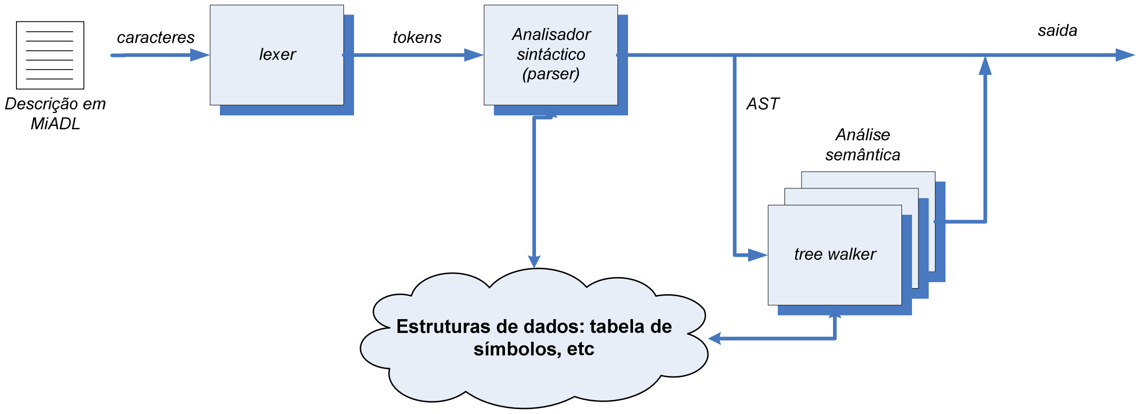


Figura 4.3: Fluxo de processamento no *front end* de MiADL

O *front end* do compilador MiADL foi implementado usando o gerador de analisadores sintácticos, designado por *ANother Tool for Language Recognition* (ANTLR) [124]. O ANTLR é um *compiler-compiler* que gera código fonte para o analisador sintáctico[29] a partir da descrição da gramática da linguagem que se pretende implementar, neste caso MiADL. Esta ferramenta automatiza a geração de

---

[29] *Parser*

reconhecedores de linguagens, sendo uma aplicação que gera outras aplicações. A partir de uma descrição formal duma linguagem, semelhante a EBNF, ANTLR produz um programa que verifica se um texto de entrada está conforme as regras sintácticas daquela linguagem. A ferramenta ANTLR segue uma análise do tipo LL(k)[30], sendo ***k*** o número máximo de *tokens* da sequência de entrada, proveniente do analisador léxico, que o analisador sintáctico (i.e. *parser*) pode analisar para determinar que regra deve ser usada em seguida [121, 126]. Do *front end*, além do analisador léxico[31] e do *parser*, fazem parte vários *tree walkers* [124] que em cadeia permitem implementar a análise semântica da linguagem. Estes passos de análise semântica, *tree walkers,* recorrem à *Abstract Syntax Tree* (AST)[32] gerada pelo analisador sintáctico e à informação da tabela de símbolos, para verificar a consistência do programa de entrada em relação à definição da linguagem. Durante a análise semântica, estes *tree walkers* percorrem a AST e armazenam nela, informações úteis para subsequente utilização por parte dos geradores de código que se encontram a jusante no processo de geração. Uma das informações guardadas na AST, no caso de MiADL, tem a ver com a relação entre determinado símbolo e a zona onde foi feita a respectiva definição/declaração. Para efectuar a validação dos símbolos, é tida em conta a hierarquia de *scopes* durante as sequências de pesquisa e é usada a tabela de símbolos. Esta informação é recolhida – anotada na AST – de forma incremental, durante as várias fases do *front end*, e depois usada pelos geradores para produzir o código alvo.

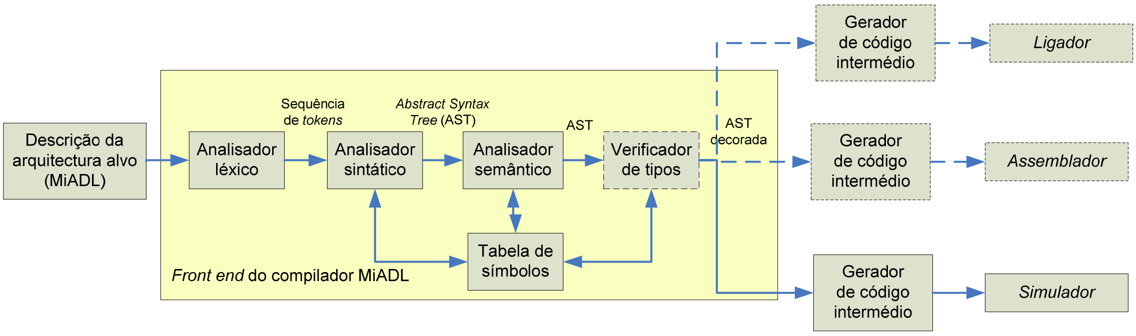


Figura 4.4: Diagrama de blocos do *front end* do compilador MiADL

---

[30] Em *LL*, o primeiro *L* significa que a análise é feita da esquerda para a direita, o segundo *L* significa que é produzida a derivação mais à esquerda.

[31] *Lexer*

[32] A AST é uma estrutura de dados intermédia com representação hierárquica da estrutura sintáctica do programa fonte.

Os geradores de código produzem estruturas na linguagem intermédia - *C++* - que correspondem ao comportamento das instruções do processador alvo, à informação para geração de descodificador do processador alvo, recursos do processador alvo e instanciação de comportamentos a partir do binário das instruções do processador alvo. Estas últimas são usadas, no caso de simuladores compilados, para em tempo de compilação produzir a customização das chamadas dos comportamentos de acordo com a aplicação alvo a simular.

De seguida apresenta-se o modo como em MiADL é gerada a representação intermédia que caracteriza o processador alvo a simular, ou seja, o código relativo ao descodificador, comportamento das instruções, funções de customização e estado do processador, Figura 4.5.

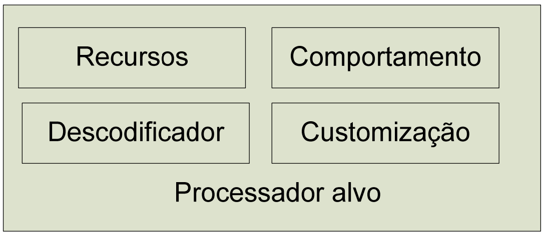


Figura 4.5: Blocos da representação intermédia que caracterizam o processador alvo

4.4.1.1 GERAÇÃO REDIRECCIONÁVEL DE RECURSOS

O modelo de recursos é completamente gerado a partir da descrição efectuada na secção ***resources*** de MiADL. Cada uma das construções, admitidas por aquela secção da linguagem, origina uma estrutura de dados própria e acessível pelo demais código gerado.

Todos os recursos são convertidos em variáveis que representam o estado do processador alvo. Os registos declarados com a palavra–chave ***register*** originam variáveis de tipo equivalente ao que está presente na definição dos mesmos.

No caso das memórias estas são descritas como simples *arrays* que usam o modo de endereçamento directo. O objectivo consiste, nesta fase, em permitir ao projectista implementar qualquer estrutura de memória e modo de endereçamento. Uma vez que uma vasta gama de memórias pode ser categorizada num grupo restrito de tipos de memória, faz sentido no futuro permitir catalogar as memórias durante a descrição. O mesmo se aplica aos modos de endereçamento. Isto será possível, com a adição de estruturas próprias a cada tipo de memória e tipo de endereçamento, o que permitirá maior facilidade na tarefa de explorar hierarquias de memória adequadas ao processador ou ao projecto do sistema.

No caso dos bancos de registos[33], estes são descritos recorrendo a *arrays* - um por cada banco de registos - com tamanho e tipo correspondentes ao indicado na declaração feita em MiADL.

4.4.1.2 GERAÇÃO REDIRECCIONÁVEL DE DESCODIFICADORES

O descodificador binário é um componente que é comum a diversas ferramentas de desenvolvimento de software tais como, simuladores de conjuntos de instruções, assembladores, desassembladores ou depuradores. O desempenho deste componente, conforme foi referido na secção 3.2, tem impacto no desempenho daquelas ferramentas.

A função deste bloco é a de traduzir a informação binária do programa alvo, contida na memória de programa, em instruções que a mesma representa. A informação relativa à instrução está relacionada com os campos e respectivos valores.

Ao ser gerado um simulador para um processador alvo, descrito em MiADL, é imprescindível a geração do respectivo descodificador. Este descodificador é gerado tendo por base a informação relativa à codificação binária dos formatos das instruções e das operações da arquitectura modelada pelo projectista. A informação binária que caracteriza determinado formato de instrução (*layout* de MiADL) corresponde aos campos que são constantes binárias do mesmo. Quanto à informação binária que caracteriza determinada operação, esta corresponde ao valor do selector que lhe é atribuído para a destituir no seio do grupo de operações do qual faz parte.

Assim, uma instrução é identificada através da comunhão entre a máscara que identifica o formato, i.e. modo de endereçamento, e o valor do *opcode* da operação a que corresponde, i.e. valor numérico do selector que é definido no grupo de operações ao qual diz respeito. Esta comunhão dos dois valores permite identificar de forma única esta instrução. Para garantir consistência desta informação, as descrições em MiADL são alvo, durante a análise semântica, de verificações que garantem a inexistência de

---

[33] *Register file*

incoerências ou redundâncias. Isto é conseguido através da proibição da descrição de formatos iguais e operações com o mesmo binário, ou seja, com a mesma máscara. Tais situações são assinaladas como erro.

O gerador de descodificadores de MiADL gera informação para construção de um descodificador baseado em árvore binária de acordo com o algoritmo proposto em [127]. A informação gerada consiste numa assinatura que é única para cada par operação-formato bem como, de um padrão que identifica as posições dos bits que devem ser analisadas para aquele par. Quanto ao número de máscaras geradas, este será igual à soma das instruções de todos os grupos. Em cada grupo, o número de instruções que o mesmo representa é igual ao produto do número de formatos admitidos pelo grupo e pelo número de operações do grupo.

Exemplo:

No caso do processador ARM existem 6 operações (ADC, ADD, RSB, RSC, SBC e SUB) que apresentam comportamentos idênticos, e todas elas podem ter os mesmos formatos, i.e. modos de endereçamento (*dpis*, *dprs* ou *dpi*). Por estes motivos, estas 6 operações podem formar um grupo de operações. A este grupo chamou-se *aritmetics*. Os formatos admitidos pelo grupo serão os três mencionados anteriormente, cujos *layouts* estão representados na Figura 4.6, e cada operação corresponderá a um valor do selector (neste caso é o campo *opcode*) que a caracteriza no seio do grupo, ou seja: ADC (*opcode*= 0101), ADD (*opcode*= 0100), RSB (*opcode*= 0011), RSC (*opcode*= 0111), SBC (*opcode*= 0110) e SUB (*opcode*= 0010) respectivamente.

Bits →	31..28	27..25	24..21	20	19..16	15..12	11..8	7	6..5	4	3..0
dpis →	cond	000	*opcode*	S	Rn	Rd	*shift amount*		*shift*	0	Rm
dprs →	cond	000	*opcode*	S	Rn	Rd	Rs	0	*shift*	1	Rm
dpi →	cond	001	*opcode*	S	Rn	Rd	*rotate*	*immediate*			

Figura 4.6: Exemplos de formatos de instruções do processador ARM

A combinação entre as máscaras dos *layouts* e os valores possíveis do selector para determinado grupo de operações, que para o grupo *aritmetics* resulta num conjunto de 18 máscaras (6 operações * 3 formatos), caracteriza de forma única cada uma daquelas instruções (ADC_DPIS, ADC_DPRS, ADC_DPI, ADD_DPIS, ADD_DPRS, ADD_DPI, RSB_DPIS,...). A tabela resultante que serve de entrada para o algoritmo proposto em [127] tem o seguinte formato:

ARITMETICS_ADC_DPIS	0x0FE00010	0x00A00000
ARITMETICS_ADC_DPRS	0x0FE00090	0x00A00010
ARITMETICS_ADC_DPI	0x0FE00000	0x02A00000

A primeira coluna identifica a nomenclatura da instrução, a segunda coluna o padrão que identifica os bits relevantes para a descodificação e a terceira coluna a assinatura daquela instrução (i.e., o valor concreto que os bits relevantes devem possuir para que se trate daquela operação).

Os bits relevantes para a descodificação são todos os que forem constantes no formato e os que no formato correspondem à posição do selector que identifica a operação. Estes bits são identificados colocando-os com valor ‘1’ e os restantes com valor ‘0’. Por exemplo, para a instrução composta pela operação ADC e pelo formato *dprs*, o padrão resultante teria o seguinte valor binário, Figura 4.7:

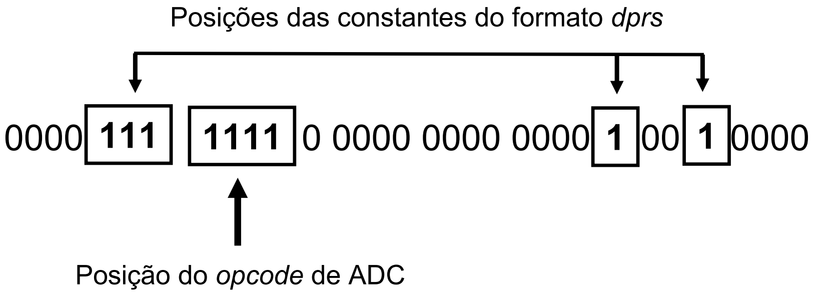


Figura 4.7: Bits relevantes durante a descodificação de uma instrução

A assinatura da instrução terá todos os valores a zero excepto nas posições dos bits relevantes, nas quais terá o valor binário igual ao das constantes do formato e do valor do *opcode*/selector da operação. Por exemplo no caso de ARITMETICS_ADC_DPRS tem-se em binário, Figura 4.8:

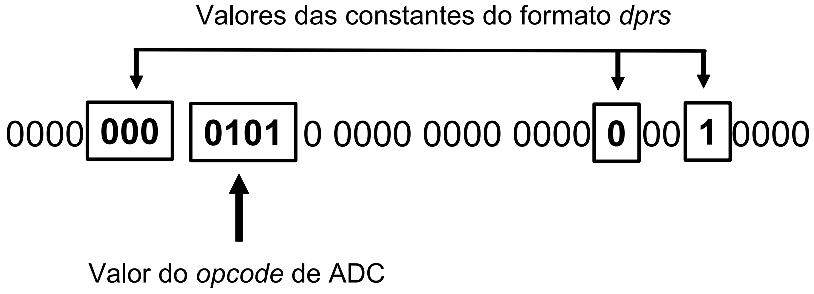


Figura 4.8: Composição da assinatura relativa a uma instrução

Em casos de arquitecturas com instruções de tamanho variável, as máscaras são compostas da mesma forma mas completando com zeros à direita até perfazer um tamanho igual ao tamanho do maior formato admitido pela arquitectura, conforme exposto em [127].

Desta forma, a partir da informação depurada de descrições MiADL é possível gerar a codificação binária das instruções sem ambiguidades (ex. mesmo código binário para operações/comportamentos diferentes) ou redundâncias (repetições da mesma operação para códigos binários diferentes).

Uma característica importante, para o desempenho do descodificador de instruções, está relacionada com o número de variantes de terminais que se geram. No algoritmo [127] todas as variantes com mesmo nome são optimizadas para que seja composta uma máscara que as identifique da forma mais rápida possível. Assim, ao gerar uma variante para cada operação da arquitectura têm-se tabelas grandes que, por sua vez, conduzem a descodificadores mais lentos. Por outro lado, ao gerarem-se variantes para cada par grupo-formato, o número de variantes será consideravelmente menor e por consequência conseguem-se descodificadores mais rápidos. Neste trabalho foi seguida esta segunda abordagem devido a características intrínsecas à forma de representação da linguagem proposta. Estas características ficarão mais claras quando for abordada a forma de especialização/customização. Seguindo modelos de representação tais como o usado em [104] esta vantagem não pode ser explorada porque não é explorado o agrupamento de operações como em MiADL.

A forma como em MiADL se representam as codificações binárias das instruções permite conseguir a geração simples de descodificadores. Neste projecto seguiu-se a implementação segundo o algoritmo [127] por este ser eficaz e permitir o seu uso em simuladores do tipo compilado ou interpretado (trabalho futuro).

4.4.1.3 GERAÇÃO REDIRECCIONÁVEL DO COMPORTAMENTO

O modelo subjacente à linguagem de descrição condiciona a forma como será representada a informação que a mesma abrange, em termos da geração do código que será executado em tempo de execução. Esta influência trás como consequência efeitos na performance e quantidade de código gerado.

Em linguagens nas quais as descrições apresentam um nível de granularidade que vai à operação, e uma função por cada modo de endereçamento de cada operação, o código fonte é um conjunto significativo de funções, pelo menos uma por cada operação e modo de endereçamento. Nestes casos, aquelas funções devem ser todas definidas pelo projectista. Porém, isto trás como inconvenientes o tempo de modelação necessário, é propício a erros, além de em termos de descodificador de instruções se traduzir também em árvores de pesquisa nas quais existe um terminal para cada uma das possibilidades modeladas.

No caso de MiADL, uma vez que as instruções são agrupadas de acordo com o comportamento que lhes é comum, é permitido conseguir descrições mais compactas, conforme foi exposto no Capítulo 3, mas por outro lado, aproveitar a representação para gerar código e *decoders* optimizados. Assim, em vez de funções dedicadas a cada variante de operação, são gerados blocos de código que traduzem o comportamento, ou sintaxe, de grupos de operações, podendo ser customizados de acordo com a instrução corrente a ser executada. Ou seja, é possível recorrer ao conceito de *generic programming* [128], que no caso de MiADL é implementado usando *function templates*. Cada um destes blocos encerra um comportamento que tem pontos de variabilidade que são devidos, por exemplo, ao comportamento da operação, modo de endereçamento, ou outros. Estes pontos de variabilidade em MiADL são facilmente identificados uma vez que correspondem a chamadas de *eswitches*. Mas usando o conceito de *function templates* o código base relativo ao comportamento de cada par grupo-formato é definido uma vez em MiADL e o gerador de simuladores compilados produz especializações que em tempo de execução correspondem a instanciações daquele mesmo comportamento, de acordo com as variabilidades contempladas no acto de descrição do processador. Estas variabilidades correspondem aos parâmetros da *function template* que por sua vez correspondem aos *eswitchs*/*maps* entretanto modelados em MiADL para cada grupo. Quanto aos argumentos da *function template* eles correspondem aos valores dos campos, e grupos de campos se existirem, do formato da instrução. Assim, em vez de um grande número de funções, uma por cada variante possível, em MiADL é gerada uma *function template* para cada par grupo de operações (i.e., *group*) – formato (i.e., *layout*). Se, por um lado, é menor a quantidade de código gerada, por outro também torna o descodificador de instruções mais eficiente uma vez que, o processo de descodificação corresponde a descobrir o par grupo-formato ao qual a instrução corrente corresponde, e instanciar a respectiva *function template*, através da substituição dos parâmetros e argumentos da mesma pelos valores/símbolos correspondentes à codificação binária da instrução corrente.

No caso do SPARC [102], por exemplo ArchC [101] publica que são gerados 119 corpos de funções cujo comportamento deve ser preenchido pelo projectista. No caso de MiADL, são geradas 39 *function templates* especializadas – a partir da descrição de 23 grupos de operações - que posteriormente são instanciadas. O processo de instanciação corresponde a determinar os parâmetros e argumentos para cada instrução do programa alvo a simular. O valor dos argumentos de entrada das funções, tem de ser determinado a partir do binário da instrução, através da informação relativa à posição dos campos nesse formato. Por outro lado, o tempo gasto a customizar a *function template* é atenuado pela eficiência do descodificador de instruções a descobrir a instrução corrente. Existe uma diminuição de performance por ser exigida a customização das *templates*, mas tal é suplantado pelos ganhos durante a descodificação e em tempo de execução, devido a optimizações feitas pelo compilador. Uma vez que, no âmbito deste trabalho o *focus* se centra na geração de simulador compilado, esta determinação dos parâmetros é feita durante tempo de compilação pelo que, o efeito nefasto não se faz sentir em tempo de execução.

Com o objectivo de apresentar um exemplo do código gerado em MiADL, apresenta-se na Figura 4.11 as *function templates* resultantes do processo de geração correspondente ao grupo de operações que comporta as operações aritméticas do processador ARM, conforme modelado em Figura 4.9 e Figura 4.10. Uma vez que, aquele grupo traduz três modos de endereçamento e seis operações (ADD, SUB,...), estas três *function templates* corresponderiam em ArchC à modelação de 18 funções. Sabendo que o ARM apresenta como particularidade a execução condicional de todas as suas operações isto exigiria a modelação e customização de 288 funções em ArchC ou a definição de uma função no comportamento genérico que é executado para qualquer instrução. Em MiADL, por exemplo, as 16 possibilidades correspondentes ao campo *condition* correspondem apenas à substituição de um dos parâmetros da *function template* por uma das 16 condições possíveis, sendo cada uma delas um tipo. Em MiADL, isto resulta no uso de um *eswitch* para modelar esta variabilidade, tendo como selector o campo *cond* (4 bits mais significativos dos formatos (i.e. *layouts*), Figura 4.10), tendo 16 possibilidades como itens relativos às respectivas chaves (valores binários possíveis), ver *eswitch condAction* apresentado na Figura 4.10.

Não existe até à data nenhum modelo ArchC para o processador ARM disponível em [101].

```
// Grupo composto pelas operações aritméticas: ADC, ADD, RSB, RSC, SBC e SUB.
group aritmetics {dpis, dprs, dpi} <op> {// Três modos de endereçamento: dpis, dprs e dpi.
        …
        operation sub<0b0010> {
                behavior(n, Vrn) { Vrn - n }

                ...

        }

        operation add<0b0100> {
                behavior(n, Vrn) { Vrn + n }

                …

        }
        ……
        behavior { // Comportamento comum do grupo de operações
             var aux: int<64>;
             if ($condAction()) {
                 aux= $opAction(rFile.read(??rn), $arg());
                 rFile.write(??rd, aux & 0xFFFFFFFF);
                    if (??s) {
                      if ( ??rd == 15){
                          CPSR=SPSR;}
                      else{
                         $updateStatus( rFile.read(??rn), $arg(), aux );}
                    }
             }
        }
        …
        eswitch<??op> opAction {0b0101: adc, 0b0100: add, 0b0010: sub,  0b0011: rsb, ..}
```

Figura 4.9: Grupo de operações aritméticas do processador ARM em MiADL

```
...
iformats <32> {

…

layout dataProcessImmedShift:dpis ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16] rd[15:12] sa[11:7]
        shifti[6:5] 0b0[4] rm[3:0] )  {

        int<32> lsl_i() { // função que retorna segundo operando quando shifti== 0b00
                ...
        }
        int<32> lsr_i() {  // função que retorna segundo operando quando shifti== 0b01
                ...
        }
        …
        eswitch<dpis.shifti> arg1 { 0b00:dpis.lsl_i, 0b01:dpis.lsr_i, … }
}

…

layout dataProcessRgShift:dprs ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16] rd[15:12] rs[11:8]

        0b0[7] shiftr[6:5] 0b1[4] rm[3:0] ) {

        int<32> lsl_r() { // função que retorna segundo operando quando shiftr== 0b00
        ...
        }
        int<32> lsr_r() { // função que retorna segundo operando quando shiftr== 0b01
        ...
        }
        ...
        eswitch <dprs.shiftr> arg2 { 0b00: dprs.lsl_r, 0b01: dprs.lsr_r, … }
}

...

layout dataProcessImmed:dpi ( cond[31:28] 0b001[27:25] op[24:21] s[20] rn[19:16] rd[15:12] rotate[11:8]
        immediate[7:0] ) {

        int<32> arg3() {     // função que retorna segundo operando deste modo de endereçamento
                ...
        }
}
...
eswitch<layout> arg {dpis: $arg1, dprs: $arg2, dpi: arg3 }

eswitch<??cond> condAction{ 0b0000: eq, 0b0001: ne, 0b0010: cshs, 0b0011: cclo, 0b0100: mi, 0b0101: pl,
0b0110: vs,  0b0111: vc,  0b1000: hi, 0b1001: ls, 0b1010: ge, 0b1011: lt, 0b1100: gt, 0b1101: le, 0b1110: al,
0b1111: nv }
 …
} // Fim da secção iformats
...
```

Figura 4.10: Excerto da secção *iformats* da descrição do processador ARM em MiADL

```
template <class condAction, class opAction, class updateStatus, class arg1>
inline void aritmetics_dpis(int cond, int op, int s, int rn, int rd, int sa, int shifti, int
rm){
   pc= npc;
   npc= npc+4;
   long long aux;
   if(condAction::func()) {
      aux=opAction::func( rFile[rn], arg1::func(rm, sa));
      rFile[rd]= aux & 0xFFFFFFFF;
         if(s){
          if(rd == 15){
            CPSR= SPSR;}
          else{
            updateStatus::func( rFile[rn], arg1::func(rm, sa), aux);}
        }
    }
};
```

```
template <class condAction, class opAction, class updateStatus, class arg2>
inline void aritmetics_dprs(int cond, int op, int s, int rn, int rd, int rs, int shftr, int rm){
   pc= npc;
   npc= npc+4;
   long long aux;
   if(condAction::func()) {
      aux= opAction::func(rFile[rn], arg2::func(rm, rs));
       rFile[rd]= aux & 0xFFFFFFFF;
       if(s){
         if(rd == 15){
           CPSR= SPSR;}
          else {
           updateStatus::func( rFile[rn], arg2::func(rm, rs), aux);}
       }
   }
};
```

```
template <class condAction, class opAction, class updateStatus>
inline void aritmetics_dpi(int cond, int op, int s,  int rn, int rd, int rotate, int immediate){
   pc= npc;
   npc= npc+4;
   long long aux;
   if(condAction::func()) {
      aux= opAction::func( rFile[rn], arg3::func(rotate, immediate));
      rFile[rd]= aux & 0xFFFFFFFF;
      if(s){
        if(rd == 15){
           CPSR= SPSR;}
         else {
         updateStatus::func( rFile[rn], arg3::func(rotate, immediate), aux);}
      }
   }
};
```

Figura 4.11: *Function templates* resultantes da geração de código a partir de MiADL para o grupo das aritméticas do processador ARM

As diferenças fundamentais entre as três *function templates* geradas automaticamente, a partir da descrição realizada em MiADL, e para aquele grupo de operações, relacionam-se com a forma como é determinado o segundo operando, inicialmente modelado usando um *eswitch* com nome ***arg*** (a dupla chamada deste *eswitch* no comportamento comum do grupo das operações aritméticas do exemplo da Figura 4.9, representa dois dos cinco pontos de variabilidade do comportamento daquele grupo de operações, estando marcados a negrito na figura). Para cada caso do *eswitch arg*, cuja variabilidade depende do formato conforme assinalado com o selector "*layout*", durante o processo de especialização das *function templates*, correspondente àquele grupo de operações, este é substituído por funções/*eswitches* de acordo com o formato/*layout* em causa. Além daquela substituição, note-se também na diferença entre argumentos na chamada das funções existentes no seio das *function templates*, uma vez que conforme foi referido no capítulo anterior, secção 3.6.3.2, as funções definidas no corpo de um *layout* podem manipular campos, que são próprios daquele formato e que podem ser diferentes entre formatos diferentes. Assim, na Figura 4.11 pode verificar-se que, por exemplo, para o formato *dpis*, a chamada de *arg1* tem como parâmetros os campos ***rm*** e ***sa***, enquanto que para *arg2* são usados como parâmetros de entrada ***rm*** e ***rs*** e para *arg3* usados ***rotate*** e ***immediate***. Esta **inferência de argumentos** é feita de forma transparente, para quem descreve, e garante as especificidades dos formatos. Ou seja, para cada grupo é gerada uma *function template* por cada formato do grupo, representando as especificidades do mesmo. Esta é uma das contribuições, uma vez que permite que o gerador do simulador extraía a informação correcta e permite ao projectista, quem descreve, focar-se apenas em cada um dos formatos. O compilador da linguagem, com base num conjunto de validações, assegura a correcta extracção dos campos usados pelas diferentes funções, de acordo com os modos de endereçamento.

Ainda em relação à geração de código, relativo ao comportamento das instruções, cada operação do grupo resulta numa *classe* que possui um método que expressa o comportamento da mesma. Uma vez que, a operação traduz o comportamento específico de cada operação, o qual é possível representar geralmente através de expressões simples, este método da classe gerada é classificado de *inline*. Além disso, como aquele comportamento recorre apenas a argumentos de entrada, a recursos (visíveis) ou a funções globais ou externas, a este é também dada a classificação de *static*.

Uma vez que se trata de funções simples, elas ao serem catalogadas de *inline* não serão efectivamente chamadas mas sim expandidas nos pontos em que forem invocadas, tornando o código mais eficiente. Este ganho de eficiência advém do facto de serem eliminados as tarefas próprias das chamadas e retorno de funções, tais como armazenamento e retirada de argumentos da pilha[34] ou armazenamento de registos. Quando uma função é tornada *inline,* todos aqueles passos relativos a chamadas de uma função deixam de ocorrer. Isto possibilita execução mais célere do código.

Assim, por exemplo no caso da operação *add* do grupo *aritmetics* o código correspondente gerado é, Figura 4.12:

```
class add {
        public:
        inline static uint32_t func( uint32_t n, uint32_t  Vrn ){
                return Vrn+n;
        }
}
```

Figura 4.12: Código relativo a comportamento específico de operação

Uma vez que ***add*** faz parte das opções dum *eswitch*, *opAction* na Figura 4.9, este será um dos tipos possíveis de ser usado como parâmetro de entrada de qualquer umas das *function templates* do grupo *aritmetics*, que tem sido usado como exemplo. Sempre que a instrução de entrada, para qualquer um daqueles formatos, tiver o valor 0b0100 para o campo ***op***, nas *function templates* é colocado o tipo *add* na posição ocupada por *opAction* na lista de parâmetros correspondente à instanciação das mesmas.

Todas as funções definidas em MiADL, independentemente da secção onde sejam definidas, que sejam chamadas na definição de *eswitches*, originam classes no código gerado enquanto que as outras originam funções. Isto está relacionado com o facto de os *eswitches* serem as construções, que representam variabilidade no comportamento e, por isso, cada um deles representa uma variabilidade de tipo - parâmetro da *function template* – enquanto que as funções não envolvidas em *eswitches* correspondem a comportamentos comuns e por isso podem ser chamadas directamente.

Ainda relativamente à composição da lista de tipos que compõem os parâmetros da *function template,* relativa a determinado par formato-grupo, refira-se que sempre que para determinado formato houver variabilidade expressa durante a fase de descrição por *eswitches*, essa mesma variabilidade corresponderá à presença dos correspondentes

---

[34] *Stack*

parâmetros na *function template*. Assim, por exemplo, para o caso que foi usado nesta secção, enquanto que para os formatos *dpis* e *dprs* os *eswitches arg1* e *arg2*, respectivamente, terão representatividade nas listas de parâmetros das *function templates* que lhes correspondem, no caso do formato *dpi*, tal não acontecerá porque neste formato não há variabilidade na determinação do segundo argumento. Ou seja, *arg3* é uma classe conhecida na fase de geração e não dependerá do binário da instrução quando se tratar de uma instrução com formato *dpi*.

Em termos de conclusão, pode afirmar-se que ao analisar uma determinada *function template,* a lista de parâmetros daquela traduz a variabilidade comportamental, para determinado par grupo de operações – formato de instrução. Ao mesmo tempo, a lista de argumentos que a mesma tiver representa a variabilidade de dados correspondente à instrução relativa ao valor do *program counter* corrente.

4.4.1.4 GERAÇÃO REDIRECCIONÁVEL DE ESPECIALIZAÇÃO

Existe uma *function template* e respectiva função de especialização, por cada par *layout* - grupo de operações. Esta função de especialização é usada para que seja feita a customização da *function template,* a partir da codificação duma instrução, a qual será instanciada quando o *program counter,* durante o tempo de execução, atingir aquela posição do programa alvo.

Esta especialização, para cada par grupo-formato, corresponde a determinar os parâmetros da *function template*, ou seja, os tipos que estão relacionados com a variabilidade do grupo, e os argumentos da *function template* que, por sua vez, estão relacionados com os valores dos campos do formato a que a mesma corresponde.

Como exemplo apresenta-se na Figura 4.13 um excerto da função de especialização, gerada a partir de informação de MiADL, para customizar a *function template* correspondente ao exemplo apresentado na Figura 4.11, para o caso do par *aritmetics-dpis*.

```
string resolve_aritmetics_dpis(int instr){
        ostringstream ost;
        int cond= (instr >> 28) & sizemsk[4];
        int op= (instr >> 21) & sizemsk[4];
        int s= (instr >> 20) & sizemsk[1];
        ...
        int rm= instr  & sizemsk[4]
        …
        int es1= resolve_condAction(cond);
        int es2= resolve_opAction(op);
        int es3= resolve_updateStatus(op);
        int es4= resolve_arg1(shifti);
        ost << "aritmetics_dpis" << "<" << condAction_itens[es1] <<  "," << dpis_arg1_itens[es4]
              <<  "," << opAction_itens[es2] <<  "," << updateStatus_itens[es3] << "> " << "("
              << cond << ", " << op << ", " << s << ", " <<    …    <<", " << rm << ", " << ")";
        return ost.str();
}
```

Figura 4.13: Exemplo de função de customização

Para cada *eswitch* é criado um *array* ordenado de *strings*, cujos elementos são as *strings* que constam na respectiva definição relativamente a cada uma das chaves.

O *array sizemsk* devolve as máscaras correspondentes aos diferentes tamanhos que são permitidos para os campos, e que constam das declarações feitas na secção *iformats* de MiADL.

No caso de alguns dos *layouts* terem a eles associado um *gpfield*, então esse será adicionado à lista de argumentos, após os campos, e ser-lhe-à associado um valor, logo após determinar o valor de todos os outros campos e antes de determinar o valor dos parâmetros, uma vez que podem haver *eswitchs* que tenham como selector esse *gpfield*.

No caso do simulador compilado, esta customização é processada durante tempo de compilação. Assim, para cada valor do *program counter* do programa alvo, resulta uma instanciação da *function template* correspondente à codificação de instrução contida naquela posição de memória. A *function template* customizada representa o comportamento e dados da instrução. Esta será executada, sempre que em tempo de execução aquela posição do *program counter* for invocada pelo programa alvo.

Em termos de *function template* geradas, existe um somatório de $n_i \times lay_i$, sendo n_i um grupo i e lay_i o número de *layouts* que n_i admite. Além disso, teremos uma classe por cada operação, uma por cada função que seja chamada em *eswitch* e funções de apoio. Ambas, classes e funções, são reaproveitadas em vários grupos conforme o ISA do processador alvo.

4.4.2 GERADOR DE SIMULADORES COMPILADOS

Após a geração da representação intermédia, correspondente ao processador alvo, o passo seguinte, como foi representado na Figura 4.1, consiste em gerar o simulador compilado tendo também como entrada a aplicação alvo a ser simulada.

Com base nestas representações é gerado o simulador compilado que, posteriormente, poderá ser compilado – por um compilador nativo – e de seguida executado numa máquina hospedeira. Durante este processo de geração do simulador compilado dois módulos são importantes: o carregador de aplicações e o gerador da rotina principal do simulador, que são descritos nas secções seguintes, Figura 4.14.

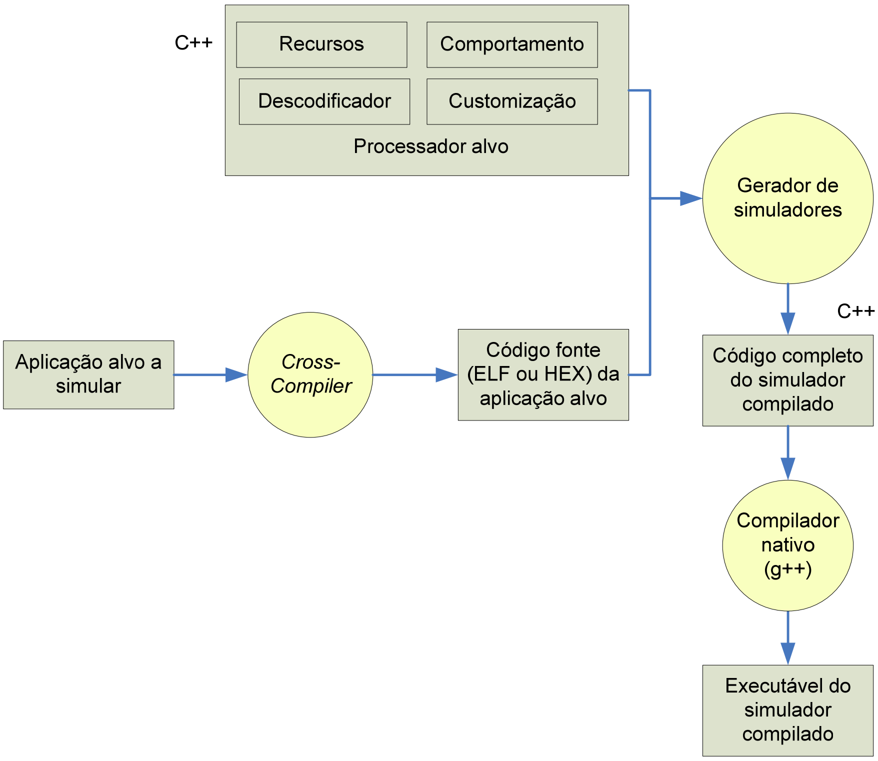


Figura 4.14: Geração de simulador compilado

4.4.2.1 CARREGADOR DE APLICAÇÕES ALVO

Durante a geração do simulador compilado, a aplicação alvo deve ser carregada para memória usando este componente do sistema. Uma vez que, o compilador para a arquitectura alvo pode estar ou não disponível, os simuladores de MiADL permitem carregar ficheiros no formato *Executable and Linkable Format* (ELF).

4.4.2.2 GERAÇÃO DA ROTINA PRINCIPAL DO SIMULADOR

Gerar a rotina principal do simulador depende do processador e da aplicação alvo a simular. Em MiADL esta rotina é construída recorrendo ao descodificador e às funções de customização, cujos processos de geração foram descritos anteriormente.

Assim, para cada posição do *program counter* do programa alvo, é customizada a instanciação de uma *function template*, correspondente à instrução por ele apontada, que representará o comportamento e os dados correspondentes àquela instrução na rotina principal. Este processo repete-se até terminar o programa e o *program counter* é incrementado para a posição seguinte através da adição do tamanho do formato corrente, que pode não ser constante, no caso de arquitecturas com instruções de tamanho variável. O processo de customização está representado em forma de pseudo-código na Figura 4.15.

```
para PC ← Início, PC <= tamanho_do_programa, PC ← PC + INCR
        grupo-formato= Descodificar (instr);
        Ftemplate customizada= Customizar_grupo-formato(instr);
        Adicionar ao main a Ftemplate customizada;
        INCR ← tamanho do formato encontrado;
fim para
```

Figura 4.15: Pseudo-código relativo à geração de simulador compilado

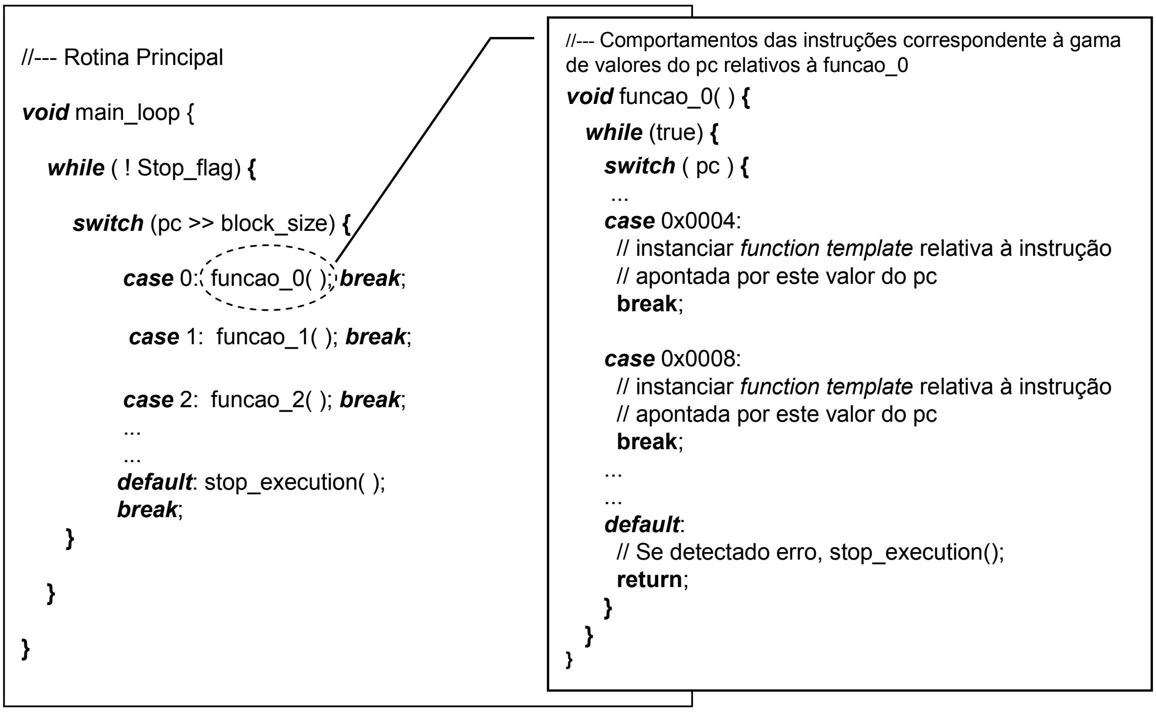
O programa alvo é totalmente descodificado, instrução a instrução, e é criada uma tabela auxiliar onde são guardadas as instanciações das *function templates*, i.e. comportamento das instruções, do simulador relativas às diferentes posições do *program counter*. A ordem de execução das instruções é mantida igual à do programa original, mas aquelas funções devem poder ser acedidas de forma aleatória, conforme o fluxo de programa, uma vez que em tempo de execução todas são potenciais alvos de saltos. Para permitir esta flexibilidade, o programa principal do simulador gerado é composto por uma cadeia de *switchs* e *cases* [128]. Cada *function template* é indexada através do valor do *program counter* em que foi descodificada.

Uma vez que o programa alvo, a simular, pode ser extenso e como cada instrução é descodificada independentemente se vai ser ou não executada, pode haver necessidade de existirem estruturas *switch* com muitas opções, o que por sua vez tem consequência nefasta no tempo de compilação e na quantidade de memória requerida. Neste trabalho foi usado o método proposto em [10], e usado em [22] com separação de código em funções, o qual permite reduzir o tempo de compilação através da subdivisão em vários blocos de menor tamanho. Assim, o programa é dividido em blocos, cujo tamanho é escolhido pelo projectista através de um parâmetro de entrada. Cada um destes blocos contém uma sequência de instruções num único *switch*, por ordem crescente e seguida do respectivo valor do *program counter*. Ou seja, existe um primeiro *switch* que selecciona o bloco da instrução corrente a executar, e para cada bloco um outro *switch* que conduz à *function template*, i.e. comportamento, que deve ser executada naquela iteração do simulador, Figura 4.16.

A diferença para [22] consiste, devido às características de MiADL, em ser possível substituir a chamada de três funções pela instanciação de uma única *function template* que representa toda a informação relativa ao comportamento, para determinado valor do *program counter*. A menor quantidade de código que é necessário gerar, porque é explorado o que é comum a determinado conjunto de instruções, reflecte-se no menor tempo de compilação. As *function templates* são alvo de optimizações por parte do compilador, logo permitem conseguir código mais optimizado e por consequência ganhos de desempenho da simulação.

O simulador assim gerado, corresponde a uma estrutura que em tempo de execução fluí de acordo com as alterações no valor do *program counter*, as quais estão relacionadas com a natureza das instruções (*branch*, *jump*, entre outras) e com o valor dos campos dos formatos (argumentos das *function templates*). Em cada uma das posições do *program counter* o comportamento do processador simulado sofre alteração no seu estado (ex. valor dos registos e memória), de acordo com o que foi modelado em

MiADL no corpo dos grupos de operações, e que consta da respectiva *function template*.



```
//--- Rotina Principal

void main_loop {

   while ( ! Stop_flag) {

      switch (pc >> block_size) {

            case 0: funcao_0( ); break;

             case 1:  funcao_1( ); break;

            case 2:  funcao_2( ); break;
            ...
            ...
            default: stop_execution( );
            break;
      }

   }

}
```

```
//--- Comportamentos das instruções correspondente à gama
de valores do pc relativos à funcao_0
void funcao_0( ) {
   while (true) {
     switch ( pc ) {
       ...
       case 0x0004:
         // instanciar function template relativa à instrução
         // apontada por este valor do pc
         break;

       case 0x0008:
         // instanciar function template relativa à instrução
         // apontada por este valor do pc
         break;
       ...
       ...
       default:
         // Se detectado erro, stop_execution();
         return;
       }
    }
}
```

Figura 4.16: Fluxo de execução do simulador compilado

4.4.3 GERAÇÃO DE DESASSEMBLADORES

Conforme referido no capítulo anterior, a sintaxe das instruções é representada em MiADL de forma compacta, em secções, *asm*, dedicadas a esse fim. Assim, para cada instrução é possível fazer corresponder a informação relativa à codificação binária da mesma, com o formato da representação em *assembly*. Aquele formato será a concatenação de um conjunto de *strings* e variáveis. As *strings* estão relacionadas com as mnemónicas das operações, pontuações e espaços em branco no *assembly*. As variáveis traduzem os valores dos campos do formato ao qual a instrução obedece.

Assim, a partir de uma palavra de instrução, após descodificação da instrução, é possível preencher o padrão de *assembly* a que a mesma diz respeito, com base no valor dos campos que são extraídos a partir daquela informação binária. A representação em *assembly* é posteriormente emitida para cada instrução presente no código de programa.

O desassemblador é usado para imprimir análises, *traces,* de execução dos simuladores para ajudar a depurar os programas a simular.

Tudo indica que o processo de assemblar é também possível de ser gerado a partir da informação descrita em MiADL. Não foi implementado neste trabalho uma vez que o *focus* se centrou na geração de simuladores. No entanto, será verificada futuramente a sua funcionalidade, sendo possivelmente necessário estender a linguagem com o objectivo de contemplar variabilidades que existem no *assembly* de determinados processadores e mesmo entre fabricantes.

Para os processadores modelados, as construções actuais permitiram descrições rápidas e compactas, tendo por base informação presente nos respectivos manuais.

4.5 CONSIDERAÇÕES RELATIVAS AO AMBIENTE DE GERAÇÃO

Durante a execução/implementação deste ambiente de software, construído para a geração redireccionável de ferramentas de software, foram tidas em conta várias considerações, durante a selecção do tipo de ferramentas/ambientes a usar, para garantir fácil adaptabilidade, expansibilidade do projecto e a possibilidade de partilha com a comunidade científica relacionada com a área do mesmo.

Nesta secção da tese o intuito não é evidenciar ferramentas mas sim o paradigma que as mesmas possuem e que foi factor de uso das mesmas em detrimento de outras.

4.5.1 SEPARAÇÃO MODELO-VISTA-CONTROLADOR

Para a construção da infra-estrutura de compilação da linguagem (analisador léxico, analisador sintáctico e cadeia de analisadores semânticos), foi usada uma ferramenta designada por *ANother Tool for Language Recognition* (ANTLR) [126] que se integra com facilidade com outra, *StringTemplate* [129] do mesmo autor, permitindo esta última, uma separação entre o modelo de dados (que neste caso se encontram numa AST decorada), o controlador (*tree walker*) e a vista (código resultante da geração) [130], Figura 4.17. Uma vez que num projecto como este, cujo objectivo futuro é a geração de mais ferramentas (assembladores, depuradores, ligadores, entre outros) torna-se de grande utilidade a existência de possibilidade de redireccionabilidade, também relativamente ao tipo de ferramentas que será possível gerar. Assim, temos uma redireccionabilidade em termos das arquitecturas – ISAs – para as quais é possível gerar ferramentas e por outro lado, intrinsecamente à própria ferramenta de software, uma redireccionabilidade relativa ao que é desejável em termos dos tipos de ferramentas que é possível conseguir. Assim, o facto de se ter adoptado como motor de padrões de saída uma ferramenta [129, 131] que apresenta a separação entre modelo de dados e a vista/saída dos mesmos, permite que a evolução do projecto seja facilitada. Isto deve-se ao facto de ser possível usar o mesmo modelo de dados e focar o trabalho apenas no controlador(es) e nos padrões de saída (i.e., vistas) de acordo com o tipo de ferramentas desejadas. Além da facilidade de expansão, existe também o facto de permitir evoluções em simultâneo e por isso viabiliza mais rápidos desenvolvimentos. Quanto ao facto de ser software livre, facilita o alargamento a uma comunidade mais ampla, o que poderá permitir retirar vantagens para o projecto MiADL, a partir da massa crítica que daí possa advir futuramente.

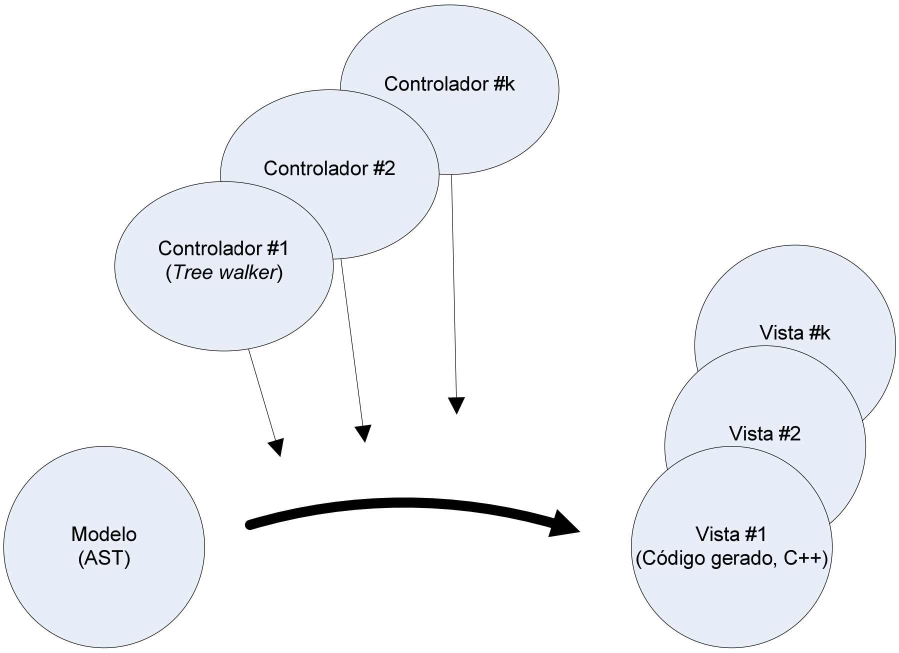


Figura 4.17: Modelo-Vistas-Controladores

4.5.2 AMBIENTE INTEGRADO

Na última década o recurso a ambientes integrados de desenvolvimento[35] (IDEs) tem vindo a substituir o desenvolvimento de aplicações desde a raiz [132]. A área de sistemas embutidos não tem sido alheia a esta evolução. Esta tendência tem vindo a ser estimulada pela crescente complexidade e tamanho do código das aplicações, necessidade de tornar o desenvolvimento mais rápido, automatizando e acelerando os

---

[35] *Integrated Development Environment*

processos de desenvolvimento para conseguir *time-to-market* competitivo, questões relacionadas com a manutenção das aplicações, compatibilidade e expansibilidade e também porque se tornam cada vez mais completos e acessíveis IDEs tais como por exemplo o Eclipse [133].

Esta estratégia de desenvolvimento permite usar recursos do IDE e assim acelerar o desenvolvimento de aplicações alvo. Neste trabalho adoptou-se um IDE [133] disponível em código aberto e multi-plataforma e considerado adequado para o desenvolvimento de ferramentas para sistemas embutidos [134]. Em [135] são apresentadas características que são possíveis herdar do IDE, com impacto no desenvolvimento da infra-estrutura MiADL, bem como proposta uma arquitectura para conseguir construir um ambiente de desenvolvimento baseado na linguagem MiADL, do qual simuladores e outras ferramentas podem fazer parte. O *front end* de MiADL foi implementado usando o IDE [133]. Quanto ao *back end*, nesta fase do trabalho usou-se o *GNU Compiler Collection* (GCC) [27] para compilar o código gerado. Está no entanto em estudo a integração de ambos os blocos da infra-estrutura (*front end* e *back end*) num ambiente mais homogéneo.

4.6 CONCLUSÃO

Na sequência do que foi referido na secção 3.2 desta tese, além do modelo segundo o qual as instruções são definidas, o algoritmo de descodificação e o método de execução das instruções descodificadas, são três factores bastante importantes para o êxito de uma infra-estrutura redireccionável de simulação. Neste capítulo foram apresentados o algoritmo de descodificação e os blocos que compõem o método de execução usado neste trabalho. Foi também descrita a forma simples como é possível construir descodificadores eficazes a partir das descrições em MiADl, explicadas no capítulo anterior. Quanto ao método de execução foram descritos os blocos principais correspondentes ao processo de geração redireccionável de simuladores, que recorrem à técnica de simulação compilada. Ao longo da descrição dos blocos, foram explicadas as formas de conseguir obter o modelo dos recursos, comportamento, descodificação e instanciação de *function templates* a partir de informação representada nas descrições MiADL. Quanto ao método de execução, foi revelado como a linguagem MiADL permite recorrer a *generic programming* para tirar proveito da exploração dos comportamentos comuns. Outra característica relevante, durante o processo de geração, consiste na inferência de argumentos para funções internas aos blocos de comportamento comum e a forma como tal é contemplado na lógica de construção dos simuladores.

Na parte final do capítulo foram abordadas algumas questões relacionadas com a selecção das ferramentas de software usadas para implementar a infra-estrutura de geração, principalmente no que se refere aos conceitos que lhes estão subjacentes. Destes realça-se em primeiro lugar o facto de a infra-estrutura ser facilmente portável, por se usar o *C++* como linguagem de representação intermédia. Realça-se também o facto de o código ser desenvolvido sob software disponível ao abrigo de software livre. E por último, aborda-se também o facto de se terem escolhido ferramentas de software que permitem conseguir redireccionabilidade em termos de diferentes ferramentas de desenvolvimento que, futuramente, possam ser geradas, para além de simuladores compilados.

CAPÍTULO 5 Modelos e Resultados

5.1 INTRODUÇÃO

Neste capítulo, descrevem-se as propriedades da linguagem que beneficiam a modelação de algumas características da arquitectura do conjunto de instruções de diversos processadores. Serão apresentadas essas propriedades tendo por base os exemplos de processadores modelados ao longo do trabalho de Doutoramento. No capítulo é feita uma análise comparativa das descrições feitas em MiADL com as efectuadas usando outras ADLs. São também apresentados os resultados de desempenho de simuladores gerados a partir de MiADL.

5.2 MODELOS

Para avaliar a aplicabilidade e eficácia da linguagem proposta nesta tese, modelaram-se três processadores diferentes. Dois deles, apesar de diferentes entre si, são contemporâneos, ARM [116] e SPARC [102]. O terceiro, é o conhecido MCS8051 [103] o qual é um dos processadores embutidos mais usados na indústria e possui a característica de ter instruções com diferentes tamanhos o que o torna diferente dos dois primeiros.

Nas secções seguintes deste capítulo, apresentam-se os processadores modelados em MiADL e a comparação com modelos conhecidos de outras ADLs. Nem todos os modelos descritos em MiADL existem também noutra ADL, pelo que são comparados apenas os publicados. A Tabela 5.1 resume a relação ADLs *vs* modelos de processadores publicados, que serão utilizados para comparar MiADL com outras linguagens.

Tabela 5.1: ADLs *vs* Modelos

	Processadores Modelados		
ADLs	MCS8051	SPARC	ARM
MiADL	√	√	√
ArchC	√	√	--
Modelo usado c/ IS-CS[36]	--	√	√
LISA	--	--	√
Facile	--	√	--
MADL	--	--	√

5.2.1 SPARC

A arquitectura SPARC é uma arquitectura RISC do tipo *load*/*store* [136] bastante conhecida e muito usada. Todas as operações aritméticas e lógicas usam operandos localizados em registos. As instruções de *load* e *store* destinam-se a ler e armazenar o conteúdo de registos em memória. Em termos de estrutura possui 32 registos disponíveis ao programador. É permitido endereçar memória até um total de 2^{30}, instruções ou inteiros [102].

A descrição em MiADL deste processador possui um total de 149 (117 + 32 de *trap)* instruções diferentes e 10 formatos (i.e. *layouts*). Em termos de tempo de elaboração do modelo do SPARC em MiADL, tendo em conta conhecimento prévio do ISA daquele processador por parte do projectista que o modelou, foi de aproximadamente 16 horas. Para conseguir este tempo de descrição, muito contribui a organização da linguagem e a forma como a mesma permite reutilização de comportamentos e *assembly* que é comum a várias operações do ISA. Em suma, para esta rapidez contribuiu o seguinte:

---

[36] Modelo de descrição usado no trabalho que utiliza a técnica *Instruction-Set Compiled Simulation*

- A forma de descrever as instruções em MiADL ser similar à representação que as mesmas têm no manual do processador.
- As descrições em MiADL serem compactas devido à reutilização intensiva de blocos.
- Uma vez que, só o que não é comum é que deve ser descrito de novo, torna o avanço da descrição mais célere, porque são reaproveitados blocos já previamente definidos.
- Uma vez que, todas as operações de um grupo partilham um comportamento comum, torna-se fácil depurar as descrições desse conjunto de operações.

De seguida, comparam-se as características de MiADL, para o caso da descrição do SPARC, relativamente a outras ADLs. Para o efeito usam-se descrições publicadas do mesmo processador em outras ADLs. Assim, ArchC, FACILE e o modelo usado com IS-CS, são trabalhos nos quais este processador foi usado como caso de estudo, Tabela 5.1.

MiADL *vs* ArchC

Comparando a descrição do processador SPARC usando MiADL, Anexo 3, com a descrição usando ArchC disponível em [101] as diferenças são consideráveis em termos de extensão das respectivas descrições e em termos de estrutura das mesmas.

A Tabela 5.2 apresenta alguns indicadores que permitem comparar a descrição das duas linguagens.

Tabela 5.2: Modelos de SPARC – MiADL *vs* ArchC

ADL	N.º de Instruções descritas	N.º de formatos	Blocos de comportamento		N.º total de linhas da descrição
			N.º de funções	*N.º de function templates geradas*	
ArchC	117 + (2 *trap*)	6	119	---	1939
MiADL	117 + (32 *trap*)	10	---	41	569

Conforme referido na Tabela 5.2, ArchC obriga à definição, por parte do projectista, de 119 funções (uma para cada instrução). Apesar de haver semelhanças entre muitas delas, é necessário definir cada uma isoladamente não tirando partido das similitudes. Pelo contrário, em MiADL é descrito apenas uma vez o que é comum. As

41 *function templates* referidas na tabela são geradas de forma automática - a partir da descrição - e correspondem, do ponto de vista de descrição, a 24 grupos de operações em que vários deles reutilizam blocos relativos a modos de endereçamento. Este poder de compactação da descrição confirma-se pelo número de linhas envolvidas que, conforme se pode constatar na Tabela 5.2, é consideravelmente inferior ao número de linhas de ArchC que foi publicado em [104].

Em relação a estas duas linguagens, há também a referir que em ArchC a partir da descrição é gerado o "esqueleto" das funções relativas ao comportamento das instruções. Ou seja, em cada ajuste feito na descrição é gerado um novo esqueleto de todas as funções que depois deve ser preenchido com código *C++*. Por outro lado, em MiADL a partir da descrição é gerado o código relativo ao comportamento das instruções em *C++* e não apenas o esqueleto. Isto faz com que, uma alteração em MiADL se reflicta em código simulável logo após geração, sem recurso a posteriores manipulações de ficheiros para reaproveitar código anterior. O processo de manipulação de ficheiros de forma não automática pode provocar redundâncias ou incoerências no código resultante.

No que se refere à descrição do *assembly* das instruções também há diferença entre as duas linguagens. Esta deve-se à obrigatoriedade que a linguagem ArchC impõe de definir o *assembly* para todas as variantes possíveis de instruções não explorando também o que é comum entre instruções. Em MiADL, recorrendo às construções próprias para lidar com a variabilidade, conseguem-se representações mais compactas e intuitivas. Também aqui, existe grande semelhança entre a representação em MiADL e a que é disponibilizada nos manuais dos processadores.

A Figura 5.1 apresenta, por questões de espaço, o comportamento de apenas quatro operações de soma retiradas do modelo do processador SPARC V8 descrito usando a linguagem ArchC, disponível em [101]. Na Figura 5.2, apresenta-se a descrição em MiADL das operações de soma e subtracção também do processador SPARC V8, incluindo o comportamento e a respectiva representação do *assembly*.

Conforme se pode constatar a partir da comparação entre a Figura 5.1 e a Figura 5.2, a linguagem MiADL conduz a descrições mais compactas devido à sua estrutura. Ao comparar as figuras verifica-se que em ArchC existe código repetido, enquanto que na Figura 5.2 tal não acontece. Por outro lado, observando a descrição em MiADL é possível obter uma visão ampla e global das operações de determinado grupo, e do ISA em geral. Em ArchC tal visão não é tão clara, devido à numerosa quantidade de funções correspondente às variantes das operações, que são obrigatoriamente descritas de forma individual.

```
…
///!Instruction addcc_reg behavior method.
void ac_behavior( addcc_reg )
{
    int dest = readReg(rs1) + readReg(rs2);
    PSR_icc_n = dest >> 31;
    PSR_icc_z = dest == 0;
    PSR_icc_v = (( readReg(rs1) &  readReg(rs2) & ~dest & 0x80000000) | (~readReg(rs1) &
                    ~readReg(rs2) &  dest & 0x80000000) );
    PSR_icc_c = ((readReg(rs1) & readReg(rs2) & 0x80000000) | (~dest & (readReg(rs1) |
                    readReg(rs2)) & 0x80000000) );
    writeReg(rd, dest);
    update_pc(0,0,0,0,0);
};
///!Instruction addcc_imm behavior method.
void ac_behavior( addcc_imm )
{
    int dest = readReg(rs1) + simm13;
    PSR_icc_n = dest >> 31;
    PSR_icc_z = dest == 0;
    PSR_icc_v = (( readReg(rs1) &  simm13 & ~dest & 0x80000000) | (~readReg(rs1) &
                    ~simm13 &  dest & 0x80000000) );
    PSR_icc_c = ((readReg(rs1) & simm13 & 0x80000000) | (~dest & (readReg(rs1) |
                     simm13) & 0x80000000) );
    writeReg(rd, dest);
    update_pc(0,0,0,0,0);
};
…
///!Instruction add_reg behavior method.
void ac_behavior( add_reg )
{
    writeReg(rd, readReg(rs1) + readReg(rs2));
    update_pc(0,0,0,0,0);
};
//!Instruction add_imm behavior method.
void ac_behavior( add_imm )
{
    writeReg(rd, readReg(rs1) + simm13);
    update_pc(0,0,0,0,0);
};
…
```

Figura 5.1: Instruções de Soma de SPARC em ArchC

```
…

//  Instruções de Soma e Subtracção de SPARC em MiADL

group addsub (F3_1, F3_2) <opcode> {

        operation add <0b10000000> {
                behavior(a, b) {a+b }
                asm {"add"}
        }

        operation sub <0b10000100> {
                behavior(a, b) {a-b }
                asm {"sub"}
        }

        operation addx <0b10001000> {
                behavior(a, b) { a+b+psr.C }
                asm {"addx"}
        }

        operation subx <0b10001100> {
                behavior(a, b) { a-b-psr.C }
                asm {"subx"}
        }

        operation addcc <0b10010000> {
                behavior(a, b) { a+b }
                asm {"addcc"}
        }

        operation addxcc <0b10011000> {
                behavior(a, b) {a+b+psr.C }
                asm {"addxcc"}
        }

        operation subcc <0b10010100> {
                behavior(a, b) { a-b }
                asm {"subcc"}
        }

        operation subxcc <0b10011100> {
                behavior(a, b) {a-b-psr.C }
                asm {"subxcc"}
        }

        behavior {
                var tmp: int<32>;

                tmp= $opAction( rFile.read(??rs1), $secondOp() );
                rFile.write( ??rd, tmp );
                $UPDateAction( tmp, rFile.read(??rs1), $secondOp() );
        }

        eswitch <??opcode> opAction { 0b10000000:add, 0b10000100:sub, 0b10001000:addx,
                0b10001100:subx, 0b10010000:addcc, 0b10011000:addxcc, 0b10010100:subcc,
                0b10011100:subxcc}

        eswitch <??opcode> UPDateAction{0b10000000:no_update_at1, 0b10010000:update_1,
                0b10001000:no_update_at1, 0b10011000:update_1, 0b10000100:no_update_at1,
                0b10010100:update_2, 0b10001100:no_update_at1, 0b10011100:update_2}

        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}
…
```

Figura 5.2: Instruções de Soma e Subtracção de SPARC descritas em MiADL

MiADL *vs* modelo de descrição usado com a técnica IS-CS

Em [137] foi usado o processador SPARC V7 como caso de estudo. Em relação a este trabalho é dito que o mesmo foi modelado usando 400 linhas. No entanto, a informação é diferente da realizada com MiADL, uma vez que, por exemplo, não é apresentada nenhuma forma de modelar o *assembly* daquele processador. A Figura 5.3 ilustra um excerto da modelação das instruções aritméticas inteiras e é uma réplica duma figura publicada recentemente em [137]. Nota-se que no exemplo da Figura 5.3 não estão modelados os casos de operações com actualização de *flags*. Isto, por seu lado, exige a repetição de código de modelação para representar essas variantes de algumas das operações (ex. *addcc* e *subcc* [138]).

```
//Functions
int extract-slice(int hiBit, int loBit) {…}
//
SPARCInst = $(IntegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) | … $;
IntegerOp = < (opcode, OpTypes), (dest, DestType), (src1, Src1Type),
                     (src2, Src2Type) | { dest = opcode(src1, src2); } >;
OpTypes = {
    (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
    (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
    (Or, xxxx-xxxx 0010-xxxx xxxx-xxxx xxxx-xxxx),
    (And, xxxx-xxxx 0001-xxxx xxxx-xxxx xxxx-xxxx),
    (Xor, xxxx-xxxx 0011-xxxx xxxx-xxxx xxxx-xxxx), …
};
DestType = [IntegerRegClass, extract-slice(29, 25)];
Src1Type = [IntegerRegClass, extract-slice(18, 14)];
Src2Type = {
      ([IntegerRegClass, extract-slice(4,0)], xxxx-xxxx xxxx-xxxx xx0x-xxxx
xxxx-xxxx),
     (#extract-slice(12,0)#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Figura 5.3: Instruções Aritméticas Inteiras de SPARC com modelo usado em IS-CS

Não existe um modelo completo acessível para que se possa comparar estes dois trabalhos, no entanto, a informação disponível em [137] permite concluir que MiADL oferece algumas vantagens para quem descreve, tais como:

- em MiADL quem descreve fá-lo de forma semelhante à informação existente em manuais;
- em MiADL os *grpfields* – grupos de campos – permitem manipular de forma transparente grupos de bits, que podem estar espalhados por

determinada instrução, sem ter de descrever máscaras, uma vez que, em MiADL a abstracção é maior com o uso de campos;

- em MiADL existe mais um grau de liberdade de descrição do que naquele modelo, porque as instruções não têm que ser obrigatoriamente agrupadas por possuírem um formato igual, mas sim por terem comportamentos semelhantes;
- em MiADL são feitas várias verificações para assegurar uma descrição válida;
- MiADL é uma linguagem enquanto que [137] é um modelo de descrição;
- MiADL apresenta uma forma de representação compacta do *assembly*;
- em MiADL o projectista foca-se, de forma natural, nas variabilidades de determinado grupo, sem ter que se preocupar com o que são entradas dessa classe/grupo de instruções. Cabe ao gerador traduzir posteriormente os pontos de variabilidade em parâmetros das *function templates*. Ou seja, o projectista abstrai-se de detalhes de programação e lida apenas com variabilidades do ISA em estudo.

Quanto à extensão da descrição em MiADL, foi possível obter uma descrição do SPARC V8, com 569 linhas, sendo 114 delas relacionadas com a representação do *assembly*. Isto permite referir que MiADL alia às vantagens referidas anteriormente, o facto de permitir descrições equivalentes, em termos de compactação, à apresentada pelo modelo de descrição apresentado em [137].

MiADL *vs* FACILE

A linguagem FACILE [71] foi usada para a geração de um simulador que utilizava a técnica de simulação designada por *fast forwarding*. Esta técnica é similar à técnica de simulação compilada e recorre a uma *cache* para guardar comportamentos, que se podem re-executar directamente no caso de se repetirem.

Nesta linguagem são usadas listas OR e AND para conseguir compactação. A descrição do código binário das instruções é feita considerando sequências de instruções com mesmo tamanho, designados por *tokens*. No caso de arquitecturas com instruções de tamanho variável serão tantos os *tokens* quantos os diferentes tamanhos previstos. Na definição de cada *token* são declarados os campos que o compõem e as respectivas posições no seio do *token*, podendo haver sobreposições. A declaração dos padrões, *pattern*, associados a determinada mnemónica é feita com recurso à declaração de condições, que os campos do respectivo *token* devem verificar para identificar aquela instrução, ou seja, mnemónica. Outra construção, ***sem***, é usada para definir o comportamento das instruções e associá-lo ao nome de *patterns*. No exemplo apresentado, Apêndice B de [139], é possível verificar que há algumas limitações em explorar o comportamento comum de diferentes operações que partilham formatos idênticos. Por exemplo, as operações *addccc* e *subccc,* apresentadas na Figura 5.4, têm descrições longas e praticamente iguais, excepto o sinal da operação aritmética a que correspondem (adição ou subtracção, respectivamente).

```
#define C64 (CCR?bit(0)?ext(64))
#define SRC2 get_src2(i, rs2, simm13)

sem [ add sub and or xor ]
  { Rx(rd, op(Rx(rs1),SRC2)); }
  where op in [ + - & | ^ ];

sem [ addcc subcc andcc orcc xorcc ]
  { Rx(rd, op(Rx(rs1),SRC2)?cc(CCR)); }
  where op in [ + - & | ^ ];

sem addc { Rx(rd, Rx(rs1) + SRC2 + C64); };
sem subc { Rx(rd, Rx(rs1) - SRC2 - C64); };

sem addccc {
        val ccr = 0x00?cvt(cc);
        val x1 = Rx(rs1); val x2 = SRC2;
        val xx = ((x1 + x2)?cc(ccr) + C64)?cc(CCR);
        CCR = ((CCR?bits(8) & 0b11011101) | (ccr?bits(8) & 0x11) |
          (((x1?bit(31)==x2?bit(31))&&(xx?bit(31)!=x1?bit(31)))?ext(8)<<1) |
          (((x1?bit(63)==x2?bit(63))&&(xx?bit(63)!=x1?bit(63)))?ext(8)<<5)) ? cvt(cc);
        Rx(rd, xx);
};

sem subccc {
        val ccr = 0x00?cvt(cc);
        val x1 = Rx(rs1); val x2 = SRC2;
        val xx = ((x1 - x2)?cc(ccr) - C64)?cc(CCR);
        CCR = ((CCR?bits(8) & 0b11011101) | (ccr?bits(8) & 0x11) |
          (((x1?bit(31)!=x2?bit(31))&&(xx?bit(31)!=x1?bit(31)))?ext(8)<<1) |
          (((x1?bit(63)!=x2?bit(63))&&(xx?bit(63)!=x1?bit(63)))?ext(8)<<5)) ? cvt(cc);
        Rx(rd, xx);
};
```

Figura 5.4: Instruções de Soma e Subtracção de SPARC descritas em FACILE

Por outro lado, o facto de ser feita a descrição do binário das instruções, dos registos e do comportamento das instruções, em ficheiros separados, torna difícil ter uma visão geral da descrição, sendo necessário algum esforço para conseguir verificar o mapeamento entre ambos, e respectiva consistência. Não são relatadas formas de verificação da descrição, o que pode tornar demorado o processo de descrição de um modelo completo e correcto. Ao analisar descrições em FACILE, do processador SPARC V9, surge a dificuldade em se ter uma visão clara dos formatos de instruções admitidos pela arquitectura do conjunto de instruções, e sobre quais operações os podem usar. Nesta linguagem é também requerido que variantes de um tipo de operação sejam nomeadas com designações diferentes, uma para cada variante, no sentido de gerar o *decoder* que atribuirá àquela versão da operação, i.e. mnemónica, determinado comportamento. Assim, a linguagem limita o reaproveitamento de várias acções que são comuns entre variantes do mesmo tipo de operação, ou de operações diferentes. Por exemplo, a operação de soma e a operação de subtracção têm variantes comuns, tais como a actualização de *flags* e contemplar, ou não, *carry*, etc. Não é explorado este comportamento comum porque, ao descrever o *encoding* das operações todas elas têm de ter designações diferentes para a geração correcta do *decoder*. Cada uma destas designações é depois mapeada num comportamento com o mesmo nome. Se tivermos como alvo, a descrição de processadores com instruções que tenham maior variabilidade, ex. ARM [116], as descrições em FACILE podem tornar-se extensas e complexas de analisar. Além disso, a forma como a informação está disponível no manual daquele processador, exigiria algum esforço para adaptação à forma de descrição em FACILE. Ao consultar a descrição do SPARC, disponível em [139], verifica-se que no caso da operação de soma são descritos quatro blocos de comportamento diferentes, i.e. *sem*, para representar as quatro variantes da operação de soma do processador (relativos à tabela da pág. 137 de [74]). Em MiADL para descrever as mesmas quatro operações é necessário apenas um grupo de operações admitindo dois formatos, conforme exemplificado na Figura 5.2 e no Anexo 3 desta tese. Além de facilitar a descrição e ser mais adequado à informação presente em manuais (não necessitando que esteja representada sob a forma de tabelas), MiADL efectua várias verificações das descrições e torna-as mais legíveis, compactas e estruturadas.

A linguagem FACILE não possui construções para representação do *assembly* das instruções.

Em [139] é referido que é possível descrever em 689 linhas a codificação das instruções, conjuntos de registos e comportamento das instruções para o caso do SPARC-v9. Uma vez que com aproximadamente o mesmo número de linhas, em MiADL se descreve o comportamento e *assembly* das operações, também para este caso MiADL permite conseguir descrições mais curtas.

5.2.2 MCS8051

O microcontrolador MCS8051 [103] da Intel, é um processador muito usado em aplicações de controlo para sistemas embutidos, e representa uma arquitectura que comporta instruções de tamanho variável (8, 16 e 32 bits). É uma arquitectura CISC, com um conjunto de instruções mais complexo do que o das máquinas RISC. A descrição do ISA deste processador serviu para ampliar a linguagem MiADL e assim melhorar o respectivo poder de descrição. No modelo elaborado, foram descritos em MiADL 21 formatos de instruções diferentes e 72 instruções (correspondentes a 41 operações diferentes) organizadas em 21 grupos.

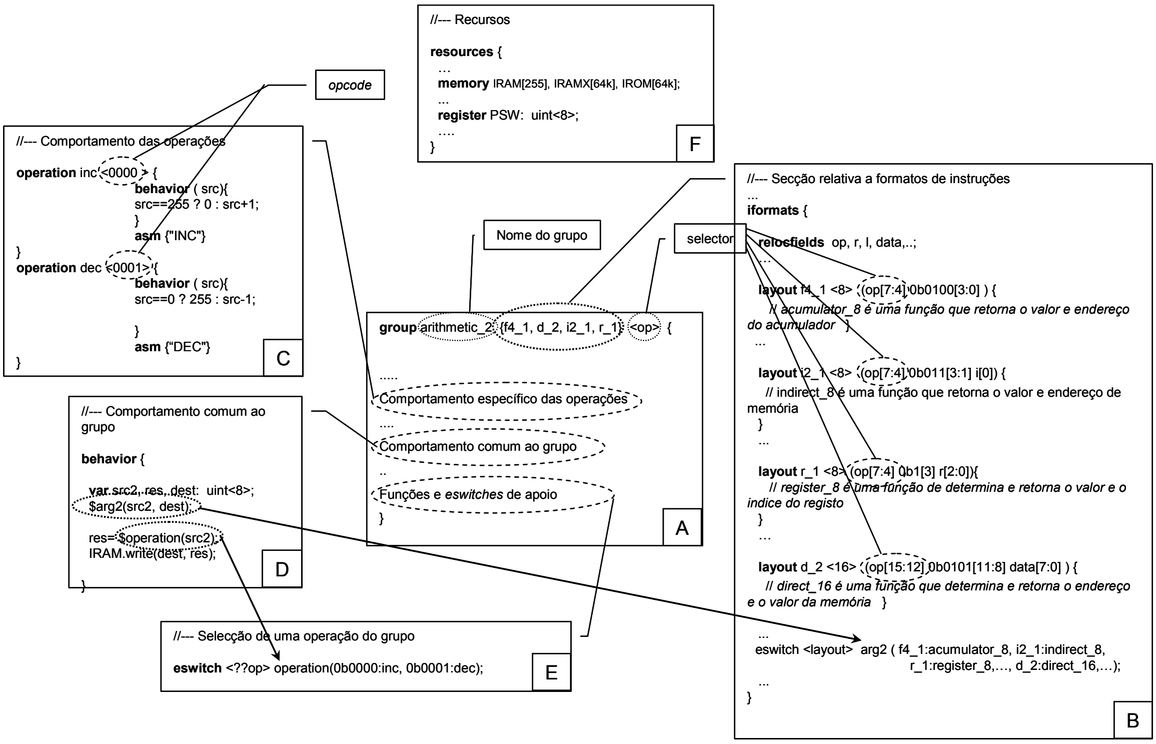
Conforme referido na Tabela 5.1, existe o conhecimento de descrições deste processador apenas em ArchC, pelo que apresentaremos as diferenças entre as duas abordagens. Assim, para modelar as mesmas instruções em ArchC é necessário preencher 72 funções. Mais uma vez, como acontece com o modelo do SPARC, não são aproveitadas as semelhanças de comportamento e *assembly* entre operações, para as agrupar e assim conseguir descrições compactas.

No caso deste processador, em MiADL é usada a definição de campos de posição variável entre formatos e a definição de formatos com tamanhos diferentes. Ou seja, as características da linguagem que permitem lidar com a variabilidade devida a tamanhos de formatos diferentes e com a regularidade de posicionamento de campos entre formatos diferentes. Quanto a esta última, é usada a definição de campos de posição variável para conseguir compactação e eficácia da descrição em MiADL. Isto é possível usando a construção*, relocfields,* que permite declarar os campos de posição variável.

Ao analisar os comportamentos das instruções deste processador verificam-se muitas semelhanças e desde logo as variabilidades contempladas por MiADL. Por exemplo, para descrever as oito operações de incremento e decremento, que não afectam *flags*, representadas na Tabela 5.3, em MiADL as mesmas operações são descritas num grupo que aceita quatro formatos. Em ArchC são necessárias oito funções, uma por cada uma das diferentes instruções. Na Figura 5.5, é possível ver um diagrama de blocos que simboliza a descrição compacta daquelas oito instruções em MiADL.

Tabela 5.3: Instruções de incremento e decremento do MCS8051 que não afectam *flags*

Formato	*Tipo*	*Assembly*	*1.º byte*	*2.º byte*	*3.º byte*	*Comportamento da instrução*	*flags*
d_2 - 2	Arithmetic	DEC direct	0001 0101	direct address		(direct) <- (direct) - 1	----
d_2 - 2	Arithmetic	INC direct	0000 0101	direct address		(direct) <- (direct) + 1	----
f4_1 - 1	Arithmetic	DEC A	0001 0100			(ACC) <- (ACC) - 1	----
f4_1 - 1	Arithmetic	INC A	0000 0100			(ACC) <- (ACC) + 1	----
I2_1 - 1	Arithmetic	DEC @Ri	0001 011i			((Ri)) <- ((Ri)) - 1	----
I2_1 - 1	Arithmetic	INC @Ri	0000 011i			((Ri)) <- ((Ri)) + 1	----
r_1 - 1	Arithmetic	DEC Rn	0001 1rrr			(Rn) <- (Rn) - 1	----
r_1 - 1	Arithmetic	INC Rn	0000 1rrr			(Rn) <- (Rn) + 1	----



```
//--- Recursos

resources {
  ...
  memory IRAM[255], IRAMX[64k], IROM[64k];
  ...
  register PSW:  uint<8>;
  ....
}
```

```
//--- Comportamento das operações

operation inc <0000 > {
                behavior ( src){
                src==255 ? 0 : src+1;
                }
                asm {"INC"}
}
operation dec <0001>{
                behavior ( src){
                src==0 ? 255 : src-1;

                }
                asm {"DEC"}
}
```

```
group arithmetic_2 {f4_1, d_2, i2_1, r_1} <op>  {

.....
Comportamento específico das operações
....
Comportamento comum ao grupo
..
Funções e eswitches de apoio
}
```

```
//--- Comportamento comum ao
grupo

behavior {

  var src2, res, dest:  uint<8>;
  $arg2(src2, dest);

  res=$operation(src2);
  IRAM.write(dest, res);

}
```

```
//--- Selecção de uma operação do grupo

eswitch <??op> operation(0b0000:inc, 0b0001:dec);
```

```
//--- Secção relativa a formatos de instruções
...
iformats {

  relocfields  op, r, l, data,..;
  ...

  layout f4_1 <8>  (op[7:4] 0b0100[3:0] ) {
     // acumulator_8 é uma função que retorna o valor e endereço
do acumulador   }
  ...

  layout i2_1 <8>  (op[7:4] 0b011[3:1] i[0]) {
    // indirect_8 é uma função que retorna o valor e endereço de
memória
  }
  ...

  layout r_1 <8> (op[7:4] 0b1[3] r[2:0]){
     // register_8 é uma função de determina e retorna o valor e o
indice do registo
  }
  …

  layout d_2 <16> (op[15:12] 0b0101[11:8] data[7:0] ) {
    // direct_16 é uma função que determina e retorna o endereço
e o valor da memória   }

  ...
  eswitch <layout>  arg2 ( f4_1:acumulator_8, i2_1:indirect_8,
                              r_1:register_8,…, d_2:direct_16,…);
  ...
}
```

Figura 5.5: Representação em MiADL das instruções da Tabela 5.3

A estrutura de MiADL, conforme se pode observar na Figura 5.5 para os casos da Tabela 5.3, ao agrupar as operações em grupo e a forma como esse grupo se relaciona com a demais informação que consta na descrição, permite de forma simples ter uma visão ampla da arquitectura do conjunto de instruções descrita pelo projectista.

Conforme foi referido na secção 3.6.3.3, também a construção *grpfield* de MiADL, permite lidar de forma simples com alguns aspectos singulares de algumas instruções deste microprocessador, tais como, as de *acall* e *ajmp* [103].

5.2.3 ARM

O processador ARM [116], é um processador RISC com uma arquitectura do conjunto de instruções complexa [136] e com grande sucesso no mercado de microprocessadores embutidos [140].

Este processador apresenta como característica o facto de ter formatos de instruções com 32 bits. Todas as instruções possuem execução condicional para maximizar a cadência de execução. Trata-se de uma arquitectura *load/store* na qual, as operações de processamento de dados actuam apenas com dados de registos e não directamente nos conteúdos de memória. As vantagens que oferece, relativamente a arquitecturas RISC básicas, permitem a estes processadores um equilíbrio de elevado desempenho, baixo consumo de energia, reduzida área de silício e reduzido tamanho de código [116].

Também o facto deste processador oferecer um conjunto complexo de formatos de instruções, permitiu aferir as características de MiADL. O Anexo 4 contém a descrição do ARM, usando MiADL.

A descrição do ISA deste processador em MiADL possui 20 formatos diferente de instruções, tendo sido descritas 87 instruções (correspondentes a 42 operações) as quais são agrupadas em 15 grupos. A descrição inclui o comportamento e *assembly* das instruções.

Ao gerar código, são produzidas de forma automática 27 *function templates* que representam o comportamento daquelas 87 instruções.

Este processador apresenta como característica o facto de todas as operações serem condicionais. Em MiADL, as 16 condições que todas as operações têm de testar foram descritas em termos de comportamento com recurso a um *eswitch*. O *eswitch*, tendo como selector o campo que identifica a condição da instrução corrente (correspondente aos 4 bits mais significativos da instrução) permite seleccionar uma função de teste de 16 possíveis. Ou seja, em MiADL aquela característica do processador ARM é facilmente descrita usando construções da linguagem.

Outros trabalhos, nos quais também foi usada esta arquitectura como caso de estudo, conforme referido na Tabela 5.1, são as ADLs LISA e MADL e o modelo usado com a técnica de simulação IS-CS. De seguida comparam-se aqueles trabalhos com MiADL.

MiADL *vs* MADL

MADL [105] é uma linguagem de descrição de processadores, usada para geração de simuladores com precisão de ciclo e simuladores de instruções. Ao consultar as descrições do processador StrongARM [141] [105] em MADL constata-se, por um lado, que é uma linguagem que foi projectada para o modelo das OSM [106] e que portanto, obriga à descrição de informação relativa aos vários estados do processador. Neste modelo de descrição, uma instrução é vista como uma sequência/máquina de estados. O Anexo 5 contém o excerto da descrição, em MADL, relativo a algumas das operações de processamento de dados, para três modos diferentes de endereçamento. Para comparar com MiADL, interessa analisar as características relacionadas com a geração do emulador, simulador do conjunto de instruções. Analisando deste prisma, facilmente se detectam várias repetições, pelo facto de as operações serem agrupadas de acordo com as OSMs, com o comportamento e com o modo de endereçamento. Assim, nota-se que para descrever as operações com modo de endereçamento diferente, existe quase uma duplicação da descrição. As diferenças verificam-se geralmente apenas na forma como é determinado algum dos operandos.

Devido à diferença entre os objectivos dos trabalhos torna-se difícil compará-los. Segundo o autor de MADL, foram elaborados emuladores de forma não automática. No entanto, as repetições relatadas anteriormente não acontecem ao descrever com a linguagem MiADL, a qual tem como propósito a geração automática de simuladores funcionais.

MiADL *vs* LISA

Que se tenha conhecimento, não existe publicado um modelo completo deste processador usando a linguagem LISA. No entanto, segundo publicado recentemente em [137], para modelar a operação de soma do processador ARM em LISA seriam necessárias pelo menos 55 linhas. Em MiADL essa operação é descrita com 3 linhas, num grupo de operações aritméticas cujo comportamento comum é descrito em 10 linhas e que admite três formatos de instrução. Isto revela, mais uma vez, que MiADL conduz a representações mais compactas e estruturadas. No Capítulo 3, foi usado esse exemplo (operação *add* bem como os formatos *dpis*, *dprs* e *dpi*) para ilustrar a sintaxe da linguagem.

Em LISA existe o recurso a estruturas AND-OR para compor uma árvore de codificações, que representa o conjunto de instruções. Esta estratégia é seguida para conseguir representações compactas, mas obriga a esforço por parte do projectista para "fragmentar" os formatos de instruções, no sentido de combinar o que é comum e formar uma estrutura em árvore que represente todas as instruções. Não existe uma forma clara de ver os formatos da arquitectura, e obriga a analisar várias partes da descrição para verificar a composição de determinada instrução. Geralmente a informação presente nos manuais não é representada em árvore.

Outra característica de LISA é o facto de permitir descrição do comportamento em *C/C++*. A verificação homogénea da descrição, LISA e *C/C++*, não é efectuada como um todo. Em MiADL a descrição é vista como um todo e as construções permitem que seja feito um vasto número de verificações, conforme foi referido ao longo de várias secções do Capítulo 3.

MiADL *vs* modelo de descrição usado com a técnica IS-CS

O que foi referido em relação às diferenças do modelo apresentado em [137] face às características de MiADL para o caso do SPARC, aplica-se de igual forma ao caso da arquitectura do processador ARM.

5.3 RESULTADOS DE DESEMPENHO

Conforme foi referido anteriormente nesta tese, Secção 3.2, além da forma de descrição e da forma como é implementado o descodificador de instruções, a forma como são implementados os comportamentos é também um factor importante porque afecta o desempenho dos simuladores. Nesta secção apresentaremos os resultados de desempenho que traduzem a velocidade de simulação usando a forma de implementação descrita no capítulo 4 para os comportamentos de operações, ou seja, usando *function templates*.

Para a obtenção dos resultados foram usados os *benchmarks* dos conjuntos MediaBench [142] e MiBench [143], por estes representarem vários domínios de aplicação e também por existirem valores publicados de desempenho conseguido por outros trabalhos usando estes *benchmarks*.

Como hospedeiro para realizar as medições foi usado um sistema composto por processador Intel(R) Core(TM)2 CPU T5600@1.83GHz, com 2046 MB de memória RAM. Quanto ao sistema operativo usado foi o Linux sendo usada a distribuição Ubuntu (Feisty Fawn) 7.04. O compilador da GNU *g*++ 3.3.6 foi usado para compilar os simuladores MiADL.

Para medida de desempenhos foi usado o modelo do SPARC e gerados os simuladores correspondentes às aplicações dos *benchmarks* utilizados nos testes. No mesmo ambiente (usando a mesma máquina, sistema operativo e compilador) foram também gerados os simuladores compilados para os modelos do mesmo processador alvo usando a *Fast Static Compiled Simulation* (FSCS) [22] que gera simuladores descritos usando a ADL ArchC, com o objectivo de analisar a diferença de desempenho entre os dois trabalhos, perante o mesmo cenário/ambiente de testes. Esta escolha deveu-se ao facto de FSCS estar disponível para uso pela comunidade e também por ser o que oferecia melhor desempenho relativamente a outros trabalhos [22]. As figuras 5.6, 5.7 e 5.8 mostram os valores obtidos. A Figura 5.6 é relativa ao *benchmark* Mediabench. A Figura 5.7 refere-se ao *benchmark* Mibench para aplicações mais simples, *small,* enquanto que a Figura 5.8 para o caso de aplicações que exigem maior esforço computacional, ou seja, *large*.

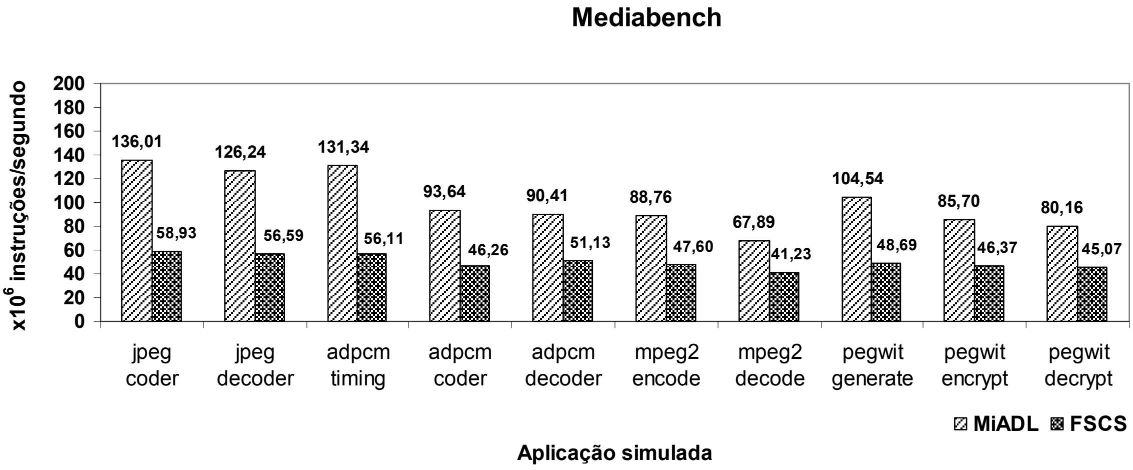


Figura 5.6: Desempenho de MiADL e de FSCS para o *benchmark* Mediabench

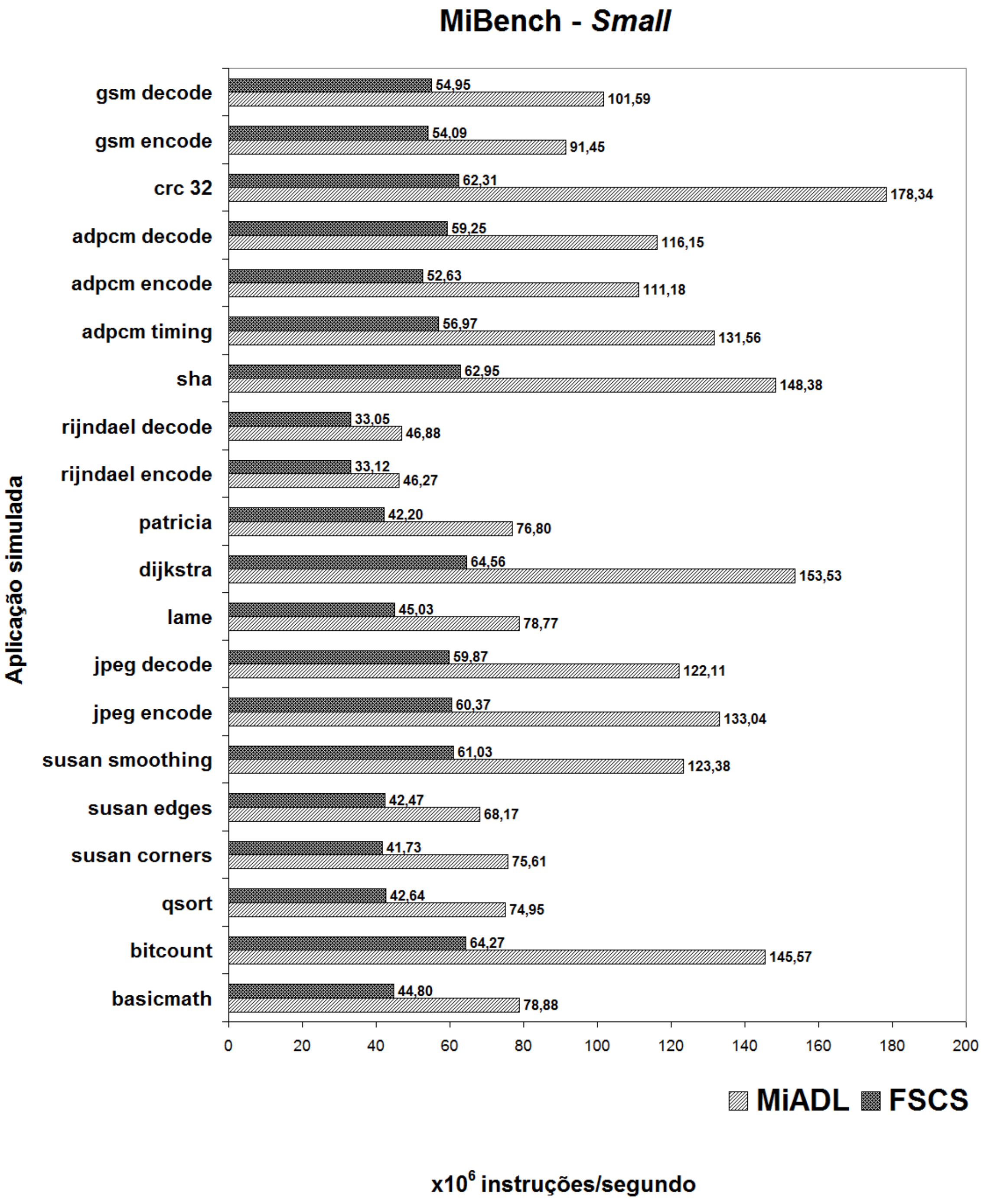


Figura 5.7: Desempenho de MiADL e de FSCS para o *benchmark* MiBench – *small*

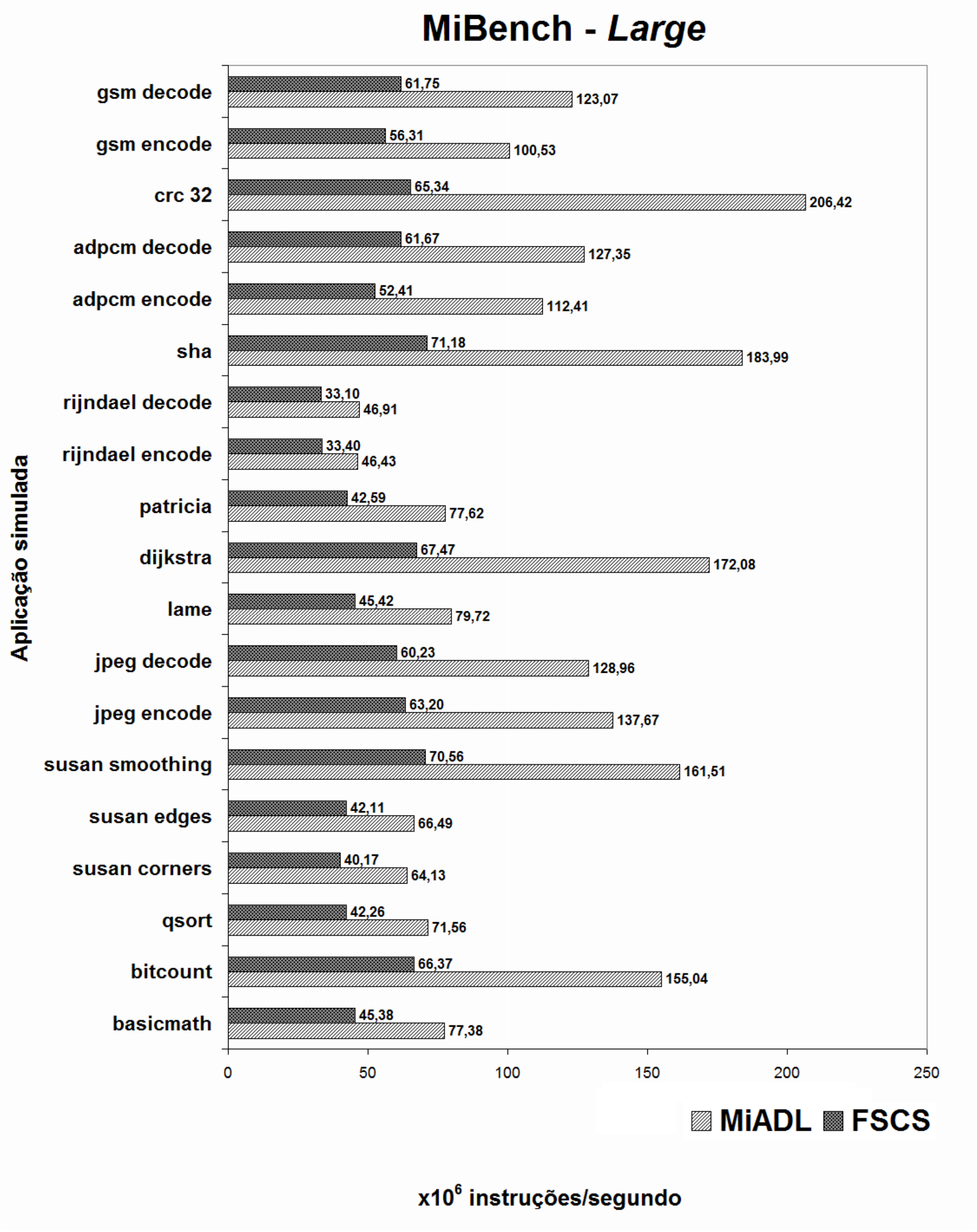


Figura 5.8: Desempenho de MiADL e de FSCS para o *benchmark* MiBench - *large*

Em todos os casos, a validação relativa à correcta execução dos programas por parte dos simuladores, foi feita realizando uma comparação entre os ficheiros gerados pelos simuladores e os ficheiros de referência existentes nos *benchmarks* disponíveis em [101]. O comando *diff* do Linux foi usado para efectuar a comparação de ficheiros.

Para atenuar a influência de cargas do CPU ou entradas/saídas nos resultados dos *benchmarks*, todas as simulações foram executadas 10 vezes. Dos 10 valores medidos foi feita a média e esse valor é o que consta das figuras.

Relativamente aos parâmetros de optimização do compilador, no caso de FSCS usamos o *switch* "*-inline*"[37] e acrescentou-se o parâmetro "*-finline-limit=1200*"[38] para activar e potenciar as optimizações relativas ao *inline* das funções relativas ao comportamento das operações. De forma equivalente, no caso do simulador MiADL usamos os parâmetros "*-O3 -finline-limit=1200*" para potenciar de igual forma o *inline* de funções[39]. Neste caso, o primeiro *switch* faz incluir optimizações *standard* enquanto que o segundo parâmetro faz com que pequenas funções possam ser *inlined*.

Tanto em MiADL como em FSCS, a medição dos tempos e a contagem do número de instruções foram efectuadas da mesma forma.

Ao analisar os resultados expressos nas figuras, conclui-se que a abordagem seguida neste trabalho apresenta ganhos significativos de desempenho porque a forma como os comportamentos das operações são implementadas, usando *function templates,* tira maior proveito das optimizações do compilador. Esta forma de implementação usando *generic programming*, como vimos no capítulo 4, deve-se à forma como a linguagem MiADL conduz a descrições que aglomeram o que é comum e identificam pontos de variabilidade.

Nesta secção, conforme referido anteriormente, apresentamos valores de MiADL *vs* FSCS porque existe disponível o código de FSCS. No entanto, o desempenho de MiADL oferece também ganhos relativos a outros trabalhos. Assim, comparando também com a proposta IS-CS do projecto EXPRESSION, que consegue atingir 14 milhões de instruções por segundo (MIPS) [137] e com a proposta JIT-CCS [20] do projecto LISA que atinge 30 MIPS [144], verifica-se que as mesmas revelam um défice

---

[37] Uma das opções de compilação dos simuladores, descrita no manual do ArchC v1.5

[38] Porque após efectuar testes com adição deste parâmetro, foram obtidos ganhos de desempenho

[39] Fundamental para conseguir bom desempenho usando a técnica das *function templates*

de desempenho perante FSCS [22], após ter em consideração as características dos respectivos sistemas *host [145]*. Os desempenhos conseguidos com o trabalho descrito nesta tese apresentam ganho significativo relativo àquelas contribuições, uma vez que os resultados apresentados nas figuras 5.6, 5.7 e 5.8, mostram que MiADL oferece sempre melhor desempenho do que FSCS, e foram obtidos usando o mesmo sistema *host* em ambos os casos.

Refira-se também que nos simuladores MiADL é mantida a frequência com que o fluxo de execução do programa é avaliado, relativamente ao que acontece com a versão base de FSCS. Ou seja, sem optimizações introduzidas no que diz respeito à avaliação do valor do *programm counter*, uma vez que, com optimizações existem apenas '*breaks'* no caso de instruções que alterem o valor daquele recurso. Por isso, comparou-se os valores de MiADL com valores medidos usando a versão base daqueles simuladores. No entanto, da análise dos resultados conseguidos verifica-se que o desempenho da versão base de MiADL se aproxima dos valores conseguidos usando optimizações em FSCS.

Em termos de ganho de performance entre MiADL e a versão base de FSCS foi atingido o mínimo de 39% (*rijndael encode* da Fig. 5.8) e o máximo de 215% (*crc 32* da Fig. 5.8), nas condições relativas aos gráficos expressos nas figuras 5.6, 5.7 e 5.8.

5.4 CONCLUSÃO

Neste capítulo, foram abordadas descrições feitas em MiADL de três processadores, que foram escolhidos como casos de estudo para comparar as características da linguagem com outras linguagens, ou modelos, onde os mesmos processadores também tenham sido usados. Os três processadores foram escolhidos por terem características diferentes entre si, e assim permitir demonstrar a redirecionabilidade de MiADL.

Ao longo do capítulo, foram apresentados dados relativos ao volume de blocos descritos nas diversas linguagens, para evidenciar a quantidade de esforço necessário ao projectista para construir uma descrição usando as diferentes abordagens.

Foram também reveladas características de MiADL que a tornam uma contribuição para facilitar a descrição de processadores e geração de simuladores funcionais, ou desassembladores. Esta facilidade leva a tempos curtos para elaboração de descrições de ISAs, por parte do projectista.

Foi referida a forma como a estrutura de MiADL conduz à obtenção de descrições compactas dos diferentes ISAs, e como essa estrutura da linguagem permite conseguir descrições mais claras e fáceis de depurar e entender. Para isso, foram apresentados excertos de descrições feitas em MiADL e noutras linguagens.

Na sequência do que foi apresentado no Capítulo 3, as construções de MiADL além de conduzirem a descrições claras, também permitem a verificação das mesmas. Este é um factor importante para a obtenção de descrições completas correctas, em períodos de tempo relativamente curtos.

Neste capítulo foram também apresentados os resultados relativos à performance de simuladores gerados a partir de MiADL e comparado esse desempenho com outros trabalhos.

Como trabalho em curso, estão a ser ultimados e testados simuladores gerados com base nas descrições de outros processadores alvo bem como o estudo de extensões da linguagem para obtenção de melhorias no desempenho dos simuladores.

CAPÍTULO 6

Conclusões

6.1 CONCLUSÃO

Actualmente, as exigências relativas ao *time-to-market* no projecto de sistemas embutidos justificam a investigação de métodos e ferramentas que permitam conseguir de forma célere gerar um conjunto de ferramentas (i.e., *toolkit*) para determinado processador, ou processadores (i.e. ser redireccionável). A não existência de um *standard* para a obtenção deste objectivo está relacionada com a dificuldade de definir modelos de descrição que permitam descrever todas as variantes de arquitecturas conhecidas e também a inexistência de trabalhos que permitam a geração eficiente de um *toolkit* completo.

O trabalho apresentado nesta tese visa a proposta de uma linguagem de descrição de arquitecturas de conjuntos de instruções que permita descrições fáceis, claras, compactas e validadas por uma análise semântica. A linguagem proposta apresenta características até à data não exploradas por outras ADLs, tais como o conceito de *scopes* ou a inferência de argumentos de funções.

As construções de MiADL foram projectadas para tornar a descrição de ISAs compactas mas ao mesmo tempo para permitir a validação estática das descrições, contribuindo assim para a obtenção de modelos consistentes da arquitectura alvo. Para a contribuição foram tidas em conta diversas fontes de variabilidade e de regularidade presentes em ISAs comuns que serviram de casos de estudo. Além daquelas premissas, foram também propostas construções que permitem lidar de forma eficaz e intuitiva com casos particulares, mas existentes em ISAs conhecidos.

São conseguidas descrições compactas, usando a linguagem proposta nesta tese, explorando o comportamento comum de grupos de operações. Assim, ao contrário de outras abordagens, aqui é descrito uma vez o que é comum e o que evidencia variabilidade é descrito recorrendo a construções próprias (tanto ao nível de comportamento como ao nível do *assembly* da arquitectura alvo).

Neste trabalho foi considerado como objectivo a descrição de processadores para obtenção de simuladores do tipo funcional. No entanto a linguagem é aberta para que possa ser estendida para descrição de modelos mais acurados, nomeadamente a evolução para a obtenção de simuladores do tipo *cycle-accurate*.

A forma como a informação é descrita em MiADL permite a exploração de paradigmas de programação até aqui pouco usados, nomeadamente no que diz respeito a *generic programming*. Assim, torna-se possível conseguir código gerado em menor quantidade e por outro lado optimizado para obtenção de ferramentas com boa performance.

Com MiADL é possível descrever a semântica e codificação binária bem como o *assembly* das instruções, pelo que descrições em MiADL de determinado processador alvo, podem ser usadas para simular as instruções alvo.

O autor elaborou modelos de três processadores conhecidos, SPARC [102], ARM [116] e MCS8051 [103].

Foram apresentados resultados de performance de simuladores gerados a partir de MiADL, os quais apresentam ganhos de desempenho consideráveis relativamente a outros trabalhos.

6.2 TRABALHO FUTURO

De seguida apresentam-se algumas possibilidades de extensão e evolução do trabalho proposto nesta tese. Algumas das linhas de evolução correspondem a trabalhos já iniciados pelo grupo de sistemas embutidos da Universidade do Minho e seus colaboradores, outras são possibilidades ainda em hipótese.

6.2.1 NOVOS MODELOS DE ARQUITECTURAS

Serão descritos novos modelos usando MiADL, nomeadamente de classes diferentes das já modeladas pelo autor desta tese. Assim, está previsto prosseguir com a modelação de arquitecturas VLIW, nomeadamente TMS326C62XX [146], e do microcontrolador 80296SA [147] por estas terem ISAs diferentes dos que já foram modelados. Ao modelar estas novas arquitecturas, serão certamente ampliadas as capacidades de modelação de MiADL. Por exemplo algumas construções serão adicionadas no caso de VLIW para modelar os *slots* de instruções. Está prevista também evolução futura para descrição de modelos *cycle-accurate*, e nesse caso certamente serão adicionadas construções para fazer a correspondência entre comportamentos e respectivas unidades funcionais. Será necessário permitir descrever a relação entre segmentos do comportamento de determinada instrução e os estágios da *pipeline* em que os mesmos devem ser executados.

6.2.2 OPTIMIZAÇÕES E TÉCNICAS DE SIMULAÇÃO

Neste trabalho foi explorada a técnica de simulação por compilação estática, no entanto será interessante testar outras técnicas, técnica de simulação interpretada e mista, com base nas mesmas descrições do ISA. Além de novas técnicas estão também em estudo optimizações no sentido de permitir conseguir aumentos de desempenho da técnica aplicada neste trabalho.

6.2.3 SIMULAÇÃO ACURADA

A simulação funcional é a primeira abordagem no desenvolvimento de arquitecturas. A fase seguinte exige maior precisão de simulação do modelo, através de simulação *cycle-accurate*. É espectável que o aumento de detalhe venha a acarretar decréscimo de velocidade de simulação. Este enriquecimento do trabalho trará alguma complexidade uma vez que o gerador de simuladores acurados terá que nomeadamente conseguir informações precisas do *pipeline* para prever *stalls* ou *hazards*. A ADL deverá ser estendida para descrever informação necessária para permitir extrair os estados do *pipeline* durante execução.

6.2.4 ESTIMAÇÃO DE ENERGIA

Uma possibilidade de evolução do trabalho consiste na extensão da linguagem para permitir a estimação do consumo de energia por parte do processador alvo. Já foi mostrado que é possível estimar o consumo de energia tendo por base o ISA do processador através da análise da comutação nas unidades de hardware [148]. Uma vez que o sucesso da aplicação de muitos sistemas embutidos depende da autonomia que os mesmos possam ter, em termos do consumo de energia, esta estimação é de bastante interesse para o projecto do hardware e para a geração do software de tais dispositivos.

6.2.5 SIMULAÇÃO AO NÍVEL DE SISTEMA

Para possibilitar que os modelos de MiADL evoluam para apoio a simuladores ao nível do sistema, poderá ser considerado o uso de *SystemC* [50] para descrição do hardware dos processadores alvo. Se, por um lado, é expectável que a velocidade de simulação diminua uma vez que *SystemC* usa um *kernel discret-event*, por outro lado a utilidade dos modelos será acrescida com a possibilidade de os mesmos poderem fazer parte de outros ambientes baseados em *SystemC*, o qual possui uma comunidade vasta de utilizadores.

6.2.6 APLICAÇÃO AO ENSINO DE ARQUITECTURAS

As características de MiADL podem torná-la amigável para alunos de cursos de informática, uma vez que tendo esses alunos conhecimentos de linguagens de programação, tornar-se-á intuitivo descrever ISAs usando esta linguagem. A visão que a mesma permite ter da arquitectura do conjunto de instruções poderá facilitar a compreensão de vários conceitos tais como: variabilidade de modos de endereçamento, variabilidade de grupos de operações, recursos de um processador, etc. Esta linguagem poderá ser usada para que os alunos descrevam e simulem processadores modernos e assim apreendam de forma interactiva e simples as características dos mesmos. A semelhança das descrições em MiADL relativamente à informação que geralmente está presente nos manuais será factor de empatia entre formandos e ferramenta.

6.2.7 MAIS FERRAMENTAS

A linguagem MiADL pretende ser ponto de partida para a geração de várias ferramentas, software *toolkit*, para apoio ao DSE. Assim, uma das direcções do trabalho incidirá na proposta de novas ferramentas. A ampliação da gama de ferramentas de software é um factor fundamental para potenciar o uso de MiADL como plataforma de exploração de novas arquitecturas. Além de ferramentas de simulação, será importante disponibilizar assembladores, desassembladores, compiladores e editores redireccionáveis. Este universo de ferramentas permitirá por um lado explorar várias alternativas da estrutura e conjunto de instruções para um determinado processador alvo, mas também avaliar o desempenho do software que será usado por essa arquitectura. Nesta fase, está em marcha a geração automática de assembladores, com base em descrições feitas em MiADL. A abordagem a seguir pretende usar a informação captada por MiADL e gerar ficheiros dependentes da arquitectura alvo e compatíveis com a infra-estrutura disponível no pacote *GNU Binutils* [149], seguindo um processo de geração idêntico ao proposto por [119].

Prevê-se a evolução futura e o enriquecimento da infra-estrutura[40] com novas ferramentas. Estas ferramentas serão redireccionáveis à custa da linguagem MiADL uma vez que esta permitirá configurar pontos de variabilidade, que uma vez definidos permitirão gerar ferramentas adequadas. Em [135] é apresentado uma panorâmica geral desta expectativa de evoluções da infra-estrutura.

6.2.8 FORMALISMO

As ADLs permitem a especificação de arquitecturas programáveis, i.e. ISAs. Por outro lado linguagens formais possibilitam especificações mais rigorosas. Assim, uma possível linha de investigação a explorar futuramente será a comunhão das vantagens de ambas as abordagens, tendo por base a ADL apresentada nesta tese.

---

[40] *Framework*

Um dos objectivos será dotar o trabalho com a possibilidade de validação da especificação da arquitectura. Para isso será necessário que a linguagem de especificação tenha semântica formal.

6.2.9 MAIS BLOCOS: HIERARQUIA DE MEMÓRIA, MULTI-*CORES*

Nesta tese considerou-se a descrição do núcleo de um processador. Como trabalho futuro poderá ser estendida a linguagem MiADL para a inclusão de construções que permitam a descrição do subsistema de memória (nomeadamente da hierarquia de memória) e coprocessadores.

Será interessante, tendo como ponto de partida este trabalho, estudar evoluções do mesmo para gerar modelos de arquitecturas programáveis contendo múltiplos *cores* de processadores.

6.2.10 DOCUMENTAÇÃO

Futuramente será produzida e disponibilizada informação no *website* reservado para o efeito. Da informação farão parte endereços de outros *websites* onde estejam disponíveis artigos já publicados ou que venham a ser publicados. Serão também colocados exemplos de descrições realizadas em MiADL, e outro tipo de informações geralmente presentes em *websites* deste género. O endereço do site entretanto dedicado a este propósito é http://www.miadl.org.

Referências Bibliográficas

[1] T. L. Friedman, *The World is Flat - A Brief History of the Twenty-First Century*, Edição Actualizada e Ampliada. Nova York: Farrar, Straus and Giroux, 2006.

[2] Embedded Systems in the Netherlands.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.hollandtrade.com/vko/Sectoren/ShowBouwsteen_CIC_embedded.asp?bstnum=686>.

[3] S. Wong, S. Vassiliadis, and S. Cotofana, "Embedded Processors: Characteristics and Trends", Delft University of Technology, Delft, Holanda, Technical Report: CE-TR-2004-03 2004.

[4] J. Turley, *Survey says: software tools more important than chips*, publicado em *Embedded Systems Design* (Disponível em: http://www.embedded.com), ref. 160700620, 2005. [citado em Junho de 2006].

[5] ISO, "ISO 690 - 2, Information and documentation - Bibliographic references. Part 2: electronic documents or parts thereof." Suiça: International Organization for Standardization, 1997.

[6] Introduction to Software Product Lines. C. W. Krueger and B. Software.[citado em Abril de 2007]. Disponível na World Wide Web em:<http://www.softwareproductlines.com/introduction/introduction.html>.

[7] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Boston: Kluwer Academic Publishers, 2002.

[8] W. Qin and B. Hu, "A Technique to Exploit Memory Locality for Fast Instruction Set Simulation", publicado nos *proceedings* da(o) *6th International Conference On ASIC*, Shanghai, China, 2005. pp. 915- 918.

[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*: Morgan Kaufmann, 2004.

[10] C. Mills, S. C. Ahalt, and J. Fowler, "Compiled Instruction Set Simulation", *Software, Practice and Experience*, Vol. 21, N.º 8, pp. 877-889, 1991.

[11] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling ", *IEEE Computer*, Vol. 35, N.º 2, pp. 59-67, 2002.

[12] D. G. Perez, G. Mouchard, and O. Temam, "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms", publicado nos *proceedings* da(o) *37th annual IEEE/ACM International Symposium on Microarchitecture* Portland, EUA, 2004. pp. 43- 54.

[13] V. Zivojnovic, S. Tjiang, and H. Meyr, "Compiled Simulation of Programmable DSP Architectures", publicado nos *proceedings* da(o) *IEEE Workshop on VLSI Signal Processing*, Sakai, Japão, 1995. pp. 187-196.

[14] S. Pees, "Modeling Embedded Processors and Generating Fast Simulators Using the Machine Desciption Language LISA", PhD Thesis, Integrated Signal Processing Systems, RWTH, Aachen, Novembro, 2002.

[15] A. Hoffmann, A. Nohl, S. Pees, G. Braun, and H. Meyr, "Generating Production Quality Software Development Tools Using a Machine Description Language", publicado nos *proceedings* da(o) *Design Automation and Test in Europe*, Munique, Alemanha, 2001. pp. 674-679.

[16] J. Zhu and D. D. Gajski, "A Retargetable, Ultra-Fast Instruction Set Simulator", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe Conference and Exhibition*, Munique, Alemanha, 1999. pp. 298-302.

[17] B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", in *Fast Simulation of Computer Architectures*, Vol. 1, T. M. Conte and C. E. Gimarc, Eds. Boston/London/Dordrecht: Kluwer Academic Publishers, 1995. pp. 5-46.

[18] E. Witchel and M. Rosenblum, "Embra: Fast and Flexible Machine Simulation", publicado nos *proceedings* da(o) *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, EUA, 1996. pp. 68-79.

[19] W. S. Mong and J. Zhu, "DynamoSim: A Trace-Based Dynamically Compiled Instruction Set Simulator", publicado nos *proceedings* da(o) *IEEE/ACM International Conference Computer Aided Design*, San Jose, EUA, 2004. pp. 131-136.

[20] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, and H. Meyr, "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation", publicado nos *proceedings* da(o) *39th Design Automation Conference*, New Orleans, EUA, 2002. pp. 22- 27.

[21] M. Reshadi, P. Mishra, and N. Dutt, "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation", publicado nos *proceedings* da(o) *40th Design Automation Conference*, Anaheim, EUA, 2003. pp. 758-763.

[22] M. Bartholomeu, "Simulação Compilada para Arquitecturas Descritas em ArchC", PhD Thesis, Instituto de Computação, Universidade Estadual de Campinas, Brasil, Setembro, 2005.

[23] P. S. Magnusson, M. Christensson, J. Askilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform", in *Computer*, Vol. 35, 2002. pp. 50-58.

[24] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors", in *Computer*, Vol. 35, 2002. pp. 40-49.

[25] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", *ACM SIGARCH Computer Architecture News* Vol. 25, N.º 3, pp. 13 - 25, 1997.

[26] SimpleScalar to Serve and Project. SimpleScalar LLC.[citado em Janeiro de 2007]. Disponível na World Wide Web em:<http://www.simplescalar.com/>.

[27] GCC, the GNU Compiler Collection. Free Software Foundation, Inc.[citado em Novembro de 2006]. Disponível na World Wide Web em:<http://gcc.gnu.org/>.

[28] T. Nakada, T. Tsumura, and H. Nakashima, "Design and Implementation of a Workload Specific Simulator", publicado nos *proceedings* da(o) *39th Annual Simulation Symposium.* , Huntsville, 2006. pp. 12-23.

[29] S. Pees, A. Hoffmann, and H. Meyr, "Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language", publicado nos *proceedings* da(o) *Conference on Design, Automation & Test in Europe*, Paris, França, 2000. pp. 669 - 673.

[30] B. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", publicado nos *proceedings* da(o) *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, EUA, 1994. pp. 128 -137.

[31] M. Burtscher and I. Ganusov, "Automatic Synthesis of High-Speed Processor Simulators", publicado nos *proceedings* da(o) *37th International Symposium on Microarchitecture*, Portland, Oregon, EUA, 2004. pp. 55-66.

[32] Using SHADE to Trace Program Execution. Sun Developer Network (SDN). D. Gove.[citado em Novembro de 2006]. Disponível na World Wide Web em:<http://developers.sun.com/sunstudio/articles/shade.html>.

[33] Simulators. Computer Architecture Page. L. Yen, M. Xu, M. Martin, D. Burger, and M. Hill.[citado em Novembro de 2006]. Disponível na World Wide Web em:<http://www.cs.wisc.edu/~arch/www/tools.html>.

[34] A. Halambi, P. Grun, H. Tomiyama, N. Dutt, and A. Nicolau, "Automatic Software Toolkit Generation for Embedded Systems-on-Chip", publicado nos *proceedings* da(o) *International Conference on VLSI and CAD*, Seul, Coreia, 1999. pp. 107-116.

[35] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau, "Architecture Description Languages for Systems-on-Chip Design", publicado nos *proceedings* da(o) *Asia Pacific Conference on Chip Design Language*, Fukuoka, Japão, 1999. pp. 109-116.

[36] P. Mishra and N. Dutt, "Architecture Description Languages for Programmable Embedded Systems", *IEE Proceedings on Computers and Digital Techniques (CDT)*, Vol. 152, n.º 3, N.º Special issue on Embedded Microelectronic Systems: Status and Trends, pp. 285-297, 2005.

[37] W. Qin, "Modeling and Description of Embedded Processors for the Development of Software Tools", PhD Thesis, Electrical Engineering, Princeton University, EUA, Novembro, 2004.

[38] W. Qin and S. Malik, "Architecture Description Languages for Retargetable Compilation", in *The Compiler Design Handbook: Optimizations & Machine Code Generation*: CRC Press, 2002. pp. 535-564.

[39] P. Marwedel, "The Mimola Design System: Tools for Design of Digital Processors", publicado nos *proceedings* da(o) *21st Design Automation Conference*, 1984. pp. 587- 593.

[40] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer, "The MIMOLA Language - Version 4.1", Computer Science Dpt., University of Dortmund, Dortmund, Alemanha, Technical Report, Setembro 1994.

[41] R. Leupers and P. Marwedel, "Retargetable Code Generation based on Structural Processor Descriptions", in *Design Automation for Embedded Systems*, Vol. 3, n.1. Boston: Kluwer Academic Publishers, 1998. pp. 1-36.

[42] D. Cooper, *Standard Pascal: User Reference Manual*, 1.ª Edição: W. W. Norton & Company, 1983.

[43] R. Leupers, J. Elste, and B. Landwehr, "Generation of Interpretive and Compiled Instruction Set Simulators", publicado nos *proceedings* da(o) *Asia-South Pacific Design Automation Conference*, Wanchai, Hong Kong, 1999. pp. 339-342.

[44] H. Akaboshi, "A Study on Design Support for Computer Architecture Design", PhD Thesis, Dep. Information Systems, Kyushu, Japão, 1996.

[45] H. Akaboshi and H. Yasuura, "Behaviour Extraction of MPU from HDL Description", publicado nos *proceedings* da(o) *Second Asia Pacific Conference on Hardware Description Languages*, Toyohashi, Japão, 1994. pp.

[46] UDL/I Simulation Synthesis Environment. Japan Electronic Industry Development Association.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://pjro.metsa.astem.or.jp/udli>.

[47] P. J. Ashenden, *The Designer's Guide to VHDL* 2.ª Edição: Morgan Kaufmann, 2001.

[48] S. Palnitkar, *Verilog HDL*, 2.ª Edição: Prentice Hall, 2003.

[49] "M Language Users Guide & Reference", Mentor Graphics Version 4.0 1989.

[50] SystemC(TM). Open SystemC Initiative(OSCI).[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.systemc.org/>.

[51] S. Maginot, "Evaluation Criteria of HDLs: VHDL compared to Verilog, UDL/I and M", publicado nos *proceedings* da(o) *Design Automation Conference*, Hamburgo, Alemanha, 1992. pp. 746-751.

[52] D. J. Smith, "VHDL & Verilog Compared & Contrasted - Plus Modeled Example Written in VHDL, Verilog and C", publicado nos *proceedings* da(o) *33rd Annual Conference on Design Automation*, Las Vegas, EUA, 1996. pp. 771-776.

[53] J. A. Rowson, "Hardware/Software Co-Simulation", publicado nos *proceedings* da(o) *31st Annual Conference on Design Automation*, San Diego, EUA, 1994. pp. 439 - 440.

[54] M. Freericks, "The nML Machine Description Formalism", TU Berlin Computer Science Tecnical Report - Version 1.5, Julho 1993.

[55] A. Fauth and A. Knoll, "Automated Generation of DSP Program Development Tools", publicado nos *proceedings* da(o) *International Conference on Acoustics, Speech and Signal Processing* 1993. pp. 457-460.

[56] F. Lohr, A. Fauth, and M. Freericks, "Sigh/sim: An Environment for Retargetable Instruction-Set Simulation", Dept. Computer Science, Tech. Univ. Berlin, Berlin, Alemanha, Technical Report 1993/43 1993.

[57] Chess/Checkers Environment Target Compiler Technologies.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.retarget.com/>.

[58] J. Paakki, "Attribute Grammar Paradigms - A high-level methodology in language implementation", *ACM Computing Surveys (CSUR)*, Vol. 27, N.º 2, pp. 196 - 255, 1995.

[59] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, "Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign", publicado nos *proceedings* da(o) *Design Automation Conference*, Anaheim, EUA, 1997. pp. 303-306.

[60] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability", publicado nos *proceedings* da(o) *34th Design Automation Conference*, Anaheim, EUA, 1997. pp. 299-302.

[61] S. Hanono and S. Devadas, "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator ", publicado nos *proceedings* da(o) *35th Design Automation Conference*, San Francisco, USA, 1998. pp. 510-515.

[62] G. Hadjiyiannis, P. Russo, and S. Devadas, "A Methodology for Accurate Performance Evaluation in Architecture Exploration", publicado nos *proceedings* da(o) *36th Design Automation Conference*, New Orleans, EUA, 1999. pp. 927-932.

[63] G. Hadjiyiannis and S. Devadas, "Techniques for Accurate Performance Evaluation in Architecture Exploration", in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 11, 2003. pp. 601 - 615.

[64] G. Hadjiyiannis, "An Architecture Synthesis System for Embedded Processors", PhD Thesis, Electrical Engineering and Computer Science, Massachussets Institute of Technology, 2000.

[65] A. Inoue, H. Tomiyama, F. N. Eko, H. Kanbara, and H. Yasuura, "A Programming Language for Processor Based Embedded Systems ", publicado

nos *proceedings* da(o) *Asia Pacific Conference on Hardware Description Languages*, 1998. pp. 89-94.

[66] The Zephyr Compiler Infrastructure. A. Appel, J. Davidson, and N. Ramsey.[citado em Outubro de 2006]. Disponível na World Wide Web em:<www.cs.virginia.edu/zephyr/>.

[67] D. Guilan, T. Jinlan, Z. Suqin, J. Weidu, and D. Jun, "Retargetable Cross Compilation Techniques: Comparison and Analysis of GCC and Zephyr", *ACM SIGPLAN Notices*, Vol. 37, N.º 6, pp. 38 - 44, 2002.

[68] Computer Systems Description Languages (CSDL). Zephyr Compiler Infrastructure.[citado em Janeiro de 2007]. Disponível na World Wide Web em:<http://www.cs.virginia.edu/zephyr/csdl/>.

[69] N. Ramsey and M. F. Fernandez, "Specifying Representations of Machine Instructions", *ACM Transactions on Programming Languages and Systems*, Vol. 19, N.º 3, pp. 492-524, 1997.

[70] V. Rajesh and R. Moona, "Processor Modeling for Hardware Software Codesign", publicado nos *proceedings* da(o) *International Conference on VLSI Design*, Goa, India, 1999. pp. 132-137.

[71] E. C. Schnarr, M. D. Hill, and J. R. Larus, "Facile: A Language and Compiler for High-Performance Processor Simulators", publicado nos *proceedings* da(o) *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, EUA, 2001. pp. 321 - 331.

[72] E. Schnarr and J. Larus, "Fast Out-Of-Order Processor Simulation Using Memoization", publicado nos *proceedings* da(o) *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, EUA, 1998. pp. 283 - 294.

[73] N. Ramsey and M. F. Fernandez, "The New Jersey Machine-Code Toolkit", publicado nos *proceedings* da(o) *USENIX Technical Conference*, New Orleans, EUA, 1995. pp. 289-302.

[74] D. L. Weaver and T. Germond, *The SPARC Architecture Manual - Version 9*. Santa Clara, California: SPARC International, Inc., 2000.

[75] V. Zivojnovic, S. Pees, C. Schlager, and H. Meyr, "LISA-Machine Description Language and Generic Machine Model for Hw/Sw Co-Design", publicado nos *proceedings* da(o) *IEEE Workshop on VLSI Signal Processing*, San Francisco, EUA, 1996. pp. 127-136.

[76] S. Pees, V. Zivojnovic, A. Hoffman, and H. Mayer, "Retargetable Timed Instruction Set Simulation of Pipelined Processor Architectures", publicado nos *proceedings* da(o) *International Conference on Signal Processing Application and Technology* Toronto, Canada, 1998. pp. 595-599.

[77] S. Pees, A. Hoffman, V. Zivojnovic, and H. Meyr, "LISA-Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures",

publicado nos *proceedings* da(o) *36th Design Automation Conference* New Orleans, 1999. pp. 933-938.

[78] S. Pees, A. Hoffmann, and H. Meyer, "Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, N.º 4, pp. 815 - 834, 2000.

[79] A. Hoffman, O. Scliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, "A Methodology for the Design of Application Specific Instruction set Processors(ASIP) using the machine description language LISA", publicado nos *proceedings* da(o) *International Conference on Computer Aided Design*, San Jose, EUA, 2001. pp. 625 - 630.

[80] G. Braun, A. Wieferink, O. Schliebusch, R. Leupers, H. Meyr, and A. Nohl, "Processor/Memory Co-Exploration on Multiple Abstraction Levels", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe Conference and Exhibition*, Munique, Alemanha, 2003. pp. 966- 971.

[81] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe Conference and Exhibition*, Paris, França, 2004. pp. 1276-1281.

[82] CoSy Technology. ACE Associated Computer Experts.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.ace.nl/compiler/cosy-technology.html>.

[83] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, M. Steinert, G. Braun, and A. Nohl, "RTL Processor Synthesis for Architecture Exploration and Implementation", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe*, Paris, França, 2004. pp. 156-160.

[84] J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun, "C Compiler Retargeting Based on Instruction Semantics Models", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe Conference and Exhibition*, Munique, Alemanha, 2005. pp. 1150-1155.

[85] LISATek. CoWare® Inc.[citado em Janeiro de 2007]. Disponível na World Wide Web em:<http://www.coware.com/>.

[86] C. Siska, "A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools", publicado nos *proceedings* da(o) *International Symposium on System Synthesis*, Hsinchu, Taiwan, China, 1998. pp. 31-36.

[87] D. A. Patterson and J. L. Hennessy, *Computer Organizations & Design - The Hardware/Software Interface*, 2.ª Edição. San Francisco, EUA: Morgan Kaufmann Publishers, Inc., 1997.

[88] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration Through

Compiler/Simulator Retargetability", publicado nos *proceedings* da(o) *DATE*, Munique, Alemanha, 1999. pp. 485-490.

[89] A. Halambi, N. Dutt, and A. Nicolau, "Customizing Software Toolkits for Embedded Systems-On-Chip", publicado nos *proceedings* da(o) *IFIP International Workshop on Distributed and Parallel Embedded Systems*, Eringerfeld, Alemanha, 2000. pp. 87-98

[90] A. Khare, "SIMPRESS: A Simulator Generation Environment for System-on-Chip Exploration", MsC Thesis Thesis, Information and Computer Science, University of California, Irvine, EUA, 1999.

[91] P. Grun, A. Halambi, N. Dutt, and A. Nicolau, "RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions", publicado nos *proceedings* da(o) *12th International Symposium on System Synthesis*, San Jose, CA, 1999. pp. 44-50.

[92] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau, "EXPRESSION: An ADL for System Level Design Exploration", University of California, Irvine Technical Report #98-29, Setembro 1998.

[93] G. Steele, *Common LISP: The Language* 2.ª Edição: Digital Press, 1984.

[94] P. Mishra, F. Rosseau, N. Dutt, and A. Nicolau, "Coprocessor Codesign for Programmable Architectures", Dept. of Information and Computer Science - University od California, Irvine, EUA, Technical Report #01-13, April 2001.

[95] P. Mishra, M. Mamidipaka, and N. Dutt, "Processor-Memory Co-Exploration using an Architecture Description Language", *ACM Transactions on Embedded Computing Systems*, Vol. 3, N.º 1, pp. 140-162, 2004.

[96] P. Mishra, N. Dutt, and A. Nicolau, "Specification of Hazards, Stalls, Interrupts, and Exceptions in EXPRESSION", University of California, Irvine, EUA, Technical Report #01-05, Janeiro 2001.

[97] P. Mishra and N. Dutt, "Modeling and Validation of Pipeline Specifications", *ACM Transactions on Embedded Computing Systems*, Vol. 3, N.º 1, pp. 114-139, 2004.

[98] P. Mishra, A. Kejariwal, and N. Dutt, "Synthesis-Driven Exploration of Pipelined Embedded Processors", publicado nos *proceedings* da(o) *17th International Conference on VLSI Design*, Mumbai, India, 2004. pp. 921-926.

[99] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt, "An Efficient Retargetable Framework for Instruction-Set Simulation", publicado nos *proceedings* da(o) *International Symposium on Hardware/Software Codesign and System Synthesis*, California, EUA, 2003. pp. 13-18.

[100] P. R. Panda, "SystemC: A Modeling Platform Supporting Multiple Design Abstractions", publicado nos *proceedings* da(o) *14th International Symposium on Systems Synthesis*, Montreal, Canada, 2001. pp. 75-80.

[101] ArchC Project. Computer Systems Laboratory of the Institute of Computing at the University of Campinas [citado em Junho de 2007]. Disponível na World Wide Web em:<http://www.archc.org/>.

[102] R. P. Paul, "SPARC Architecture Assembly Language Programming, and C", 2.ª Edição ed: Prentice Hall, 2000.

[103] "MCS(R) 51 Microcontroller Family User´s Manual", Intel Corporation 272383-002 1994.

[104] S. Rigo, "ArchC: Uma Linguagem de Descrição de Arquitecturas", PhD Thesis, Instítuto de Computação, Universidade Estadual de Campinas, Brasil, 2004.

[105] SimIt-ARM. W. Qin.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://simit-arm.sourceforge.net/>.

[106] W. Qin and S. Malik, "Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation", publicado nos *proceedings* da(o) *Design, Automation and Test in Europe* Munique, Alemanha, 2003. pp. 556-561.

[107] J. C. Gyllenhaal, W.-m. W. Hwu, and B. R. Rau, "HMDES Version 2.0 Specification", University of Illinois, Urbana-Champaign, Technical Report IMPACT-96-3 1996.

[108] TRIMARAN. An Infrastructure for Research in Instruction-Level Parallelism [citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.trimaran.org/>.

[109] L. N. Chakrapani, J. Gyllenhaal, W.-m. W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran : An Infrastructure for Research in Instruction-level Parallelism", publicado nos *proceedings* da(o) *Languages and Compilers for High Performance Computing*, West Lafayette, Indiana, EUA, 2004. pp. 32-41.

[110] T. M. U. Manual.[citado em Outubro de 2006]. Disponível na World Wide Web em:<http://www.trimaran.org/>.

[111] B. Middha, V. Raj, A. Gangwar, A. Kumar, M. Balakrishnan, and P. Ienne, "A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units", publicado nos *proceedings* da(o) *15th International Symposium on System Synthesis*, Kyoto, Japão, 2002. pp. 2-7.

[112] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "FlexWare: A Flexible Firmware Development Environment for Embedded Systems", in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston, EUA: Kluwer Academic Publishers, 1995. pp. 67-84.

[113] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "CodeSyn: A Retargetable Code Synthesis System", publicado nos *proceedings* da(o) *7th International Symposium on High-level Synthesis*, Niagra-on-the-Lake, Canada, 1994. pp. 94.

[114] S. Sutarwala, P. G. Paulin, and Y. Kumar, "Insulin: An Instruction Set Simulation Environment", publicado nos *proceedings* da(o) *Conference on Hardware Description Languages*, Ottawa, Canada, 1993. pp. 355-362.

[115] "The SPARC Architecture Manual, Version 8", SPARC International, Inc. SAV080SI9308 1992.

[116] D. Seal, *ARM Architecture Reference Manual* Vol. 1, 2.ª Edição: Addison-Wesley Professional, 2000.

[117] J. C. Metrôlho, C. A. Silva, C. Couto, and A. Tavares, "MiADL: Architecture Description Language for Design Space Exploration", publicado nos *proceedings* da(o) *3rd International IFAC Workshop on Discrete-Event System Design*, Zielona Góra, Polónia, 2006. pp. 239-244.

[118] A. Hoffman, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, "Modeling and Simulation Issues of Programmable Architectures", publicado nos *proceedings* da(o) *5th Workshop on Software & Compilers for Embedded Systems* St. Goar, Alemanha, 2001. pp.

[119] A. Baldassin, P. C. Centoducatte, and S. Rigo, "Extending the ArchC Language for Automatic Generation of Assemblers", publicado nos *proceedings* da(o) *17th International Symposium on Computer Architecture on High Performance Computing*, Rio de Janeiro, Brasil, 2005. pp. 60 - 68.

[120] M. L. Scott, *Programming Language Pragmatics*: Morgan Kaufmann Publishers, 2000.

[121] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2.ª Edição: Pearson, Addison Wesley, 2007.

[122] The GNU Compiler Collection. Free Software Foundation, Inc.[citado em Maio de 2007]. Disponível na World Wide Web em:<http://gcc.gnu.org/>.

[123] Ubuntu. Canonical Ltd.[citado em Maio de 2007]. Disponível na World Wide Web em:<http://www.ubuntu.com/>.

[124] ANTLR, ANother Tool for Language Recognition. Terence Parr T. Parr.[citado em Maio de 2007]. Disponível na World Wide Web em:<http://www.antlr.org/>.

[125] ISO, "ISO/IEC 14977 1996(E)": International Organization for Standardization(ISO) e International Electrotechnical Commission(IEC), 1996.

[126] T. Parr, "The Definitive ANTLR Reference - Building Domain-Specific Languages", 1.ª Edição ed: Pragmatic Bookshelf, 2007. pp. 384.

[127] W. Qin and S. Malik, "Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits", publicado nos *proceedings* da(o) *Design Automation Conference*, Anaheim, EUA, 2003. pp. 764-769.

[128] B. Stroustrup, "The C++ Programming Language", Special Edition ed. Boston: Addison-Wesley, 2006.

[129] StringTemplate. Terence Parr. T. Parr.[citado em Maio de 2006]. Disponível na World Wide Web em:<http://www.stringtemplate.org/>.

[130] T. J. Parr, "Enforcing Strict Model-View Separation in Template Engines", publicado nos *proceedings* da(o) *13th International Conference on World Wide Web*, Nova York, EUA, 2004. pp. 224-233.

[131] T. Parr, *A Functional Language for Generating Sctructured Text*, publicado em *StringTemplate Website* (Disponível em: http://www.stringtemplate.org/), ref. draft paper, 2007. [citado em Maio de 2007].

[132] W. Wong, *IDEs of Change*, publicado em *Electronic Design* (Disponível em: http://www.elecdesign.com/), ref. ID #12381, 2006. [citado em Maio de 2007].

[133] Eclipse - An Open Development Platform. The Eclipse Foundation.[citado em Maio de 2007]. Disponível na World Wide Web em:<http://www.eclipse.org/>.

[134] M. Turner and R. Day, *How Eclipse fits with Embedded Development*, publicado em *Embedded Cumputing Design* (Disponível em: www.embedded-computing.com/), ref. Abril06, 2006. [citado em Maio de 2007].

[135] J. C. Metrôlho, C. A. Silva, C. Couto, and A. Tavares, "Retargetable Frameworks for Embedded Systems Exploration", publicado nos *proceedings* da(o) *International Conference on Industrial Technology*, Mumbai, India, 2006. pp. 2223-2227.

[136] J. L. Hennessy and D.A.Petterson, *Computer Achitecture: A Quantitative Approach*, Vol. 1, 3.ª Edição. San Francisco: Morgan Kauffman Publishers, Inc., 2003.

[137] M. Reshadi, P. Mishra, and N. Dutt, "A Retargetable Framework for Instruction-Set Architecture Simulation", *ACM Transactions on Embedded Computing Systems*, Vol. 5, N.º 2, pp. 431-452, 2006.

[138] Sparc Version 7 Instruction Set. Atmel Wireless & Microcontrollers.[citado em Janeiro de 2007]. Disponível na World Wide Web em:<http://www.atmel.com/dyn/resources/prod_documents/doc3b8b88df7a415.pdf>.

[139] E. C. Schnarr, "Applying Programming language Implementation Techniques to Processor Simulation", PhD Thesis, University of Wisconsin, Madison, 2000.

[140] E. Rotenberg and A. Anantaraman, "Architecture of Embedded Processors", in *Multiprocessor Systems-on-Chips*, *The Morgan Kaufmann Series in Systems on Silicon*, A. A. Jerrays and W. Wolf, Eds. EUA: Elsevier Inc., 2005. pp. 81-112.

[141] Intel StrongARM SA-1110 Microprocessor Developer's Manual. Intel Corporation.[citado em Março de 2007]. Disponível na World Wide Web em:<http://www.intel.com/>.

[142] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems", publicado nos *proceedings* da(o) *30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, EUA, 1997. pp. 330-335.

[143] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", publicado nos *proceedings* da(o) *4th IEEE Annual Workshop on Workload Caracterization*, 2001. pp. 3-14.

[144] The LisaTek Solution. Coware, Inc.[citado em Junho de 2007]. Disponível na World Wide Web em:<http://www.te.rl.ac.uk/europractice/vendors /coware_lisatek.pdf/>.

[145] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC Architecture Description Language and Tools", *International Journal of Parallel Programming*, Vol. 33, N.º 5, pp. 453-484, 2005.

[146] "TMS320C62xx CPU and Instruction Set Reference Guide", Texas Instruments SPRU189B, 1997.

[147] "80296SA Microcontroller User's Manual", Vol. 272803, I. Corporation, Ed., 1996.

[148] H. Mehta, R. M. Owens, and M. J. Irwin, "Instruction Level Power Profiling", publicado nos *proceedings* da(o) *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Atlanta, Georgia, EUA, 1996. pp. 3326-3329.

[149] R. H. Pesch and J. M. Osier, "The Gnu Binary Utilities, Version 2.9.1": Iuniverse Inc., 2000.

[150] D. P. Friedman, M. Wang, and C. T. Haynes, "Essentials of Programming Languages", 2.ª Edição ed: Massachusetts Institute of Technology, 2000.

Glossário de Termos

Abstract Syntax Tree (**AST**): As *Abstract* ***S****yntax* ***T****ree* são usadas como representações internas de uma sequência de entrada de *tokens*, construídas normalmente durante a fase de *parsing*. As ASTs, por serem árvores bidimensionais, podem codificar a estrutura (conforme determinado pelo *parser*) de entrada bem como os símbolos de entrada. Por outra palavras, uma AST consiste numa estrutura de dados que representa algo que foi analisado sintacticamente, muitas vezes é usada como representação interna de um programa, por compiladores ou interpretadores, durante a fase em que aquele é optimizado e a partir do qual a geração de código é realizada.

Analisador Léxico **(**também conhecido por, *scanner*)**:** Estágio inicial dum processador de linguagem (por exemplo um compilador), a componente que realiza a análise léxica da linguagem.

Application-Specific Integrated Circuit (**ASIC**): Hardware dedicado, produzido em grandes quantidades.

Architecture Description language (**ADL**): Esta designação pode ter dois significados diferentes. Assim, em software significa uma linguagem (gráfica, textual, ou ambas) para descrever um sistema de software em termos dos seus elementos arquitecturais e a relação entre eles. Em hardware estas linguagens capturam a estrutura (componentes de hardware e respectiva conectividade) e o comportamento (*instruction set*) das arquitecturas de processadores. No âmbito desta tese, deve entender-se este ultimo significado.

Assembler: Programa de software que traduz uma representação simbólica de um programa, *assembly,* em código máquina que um processador alvo pode executar.

Assembly: Linguagem simbólica de baixo nível próxima de código máquina.

Backus-Naur Form (**BNF**): Estilo formal para especificar a sintaxe de linguagens de programação.

Banco de Registos: Conjunto de registos dum CPU que são alvos independentes para o código ser executado e por vezes complementados com registos que armazenam constantes do tipo 0/1, registos para renomear resultados intermédios, e em alguns casos uma *stack* de registos para armazenar argumentos de funções e endereços de retorno de rotinas.

Binutils: Composto por assembler, *linker* e utilitários binários da GNU. Os programas deste conjunto de aplicações são usados para assemblar, ligar e manipular ficheiros binários e objecto. Podem ser usados em conjunto com um compilador e várias livrarias para construir programas para Linux.

Bitwise: Ao nível do bit. Por exemplo operadores *bitwise* são aqueles que manipulam bits.

Campo: Componente de um formato de instrução. Ou seja, a mínima parte de um formato cujos bits possuem significado quando analisados em conjunto, i.e. bits do campo.

Central Processing Unit (**CPU**): Componente do computador responsável por efectuar o *fetch* das instruções, descodificação daquelas, e por efectuar as operações relativas às operações encontradas sob os dados correctos. Do CPU fazem parte a unidade aritmética e lógica, os registos e a unidade de controlo.

Compilador: Compilador é um programa que traduz uma linguagem de programação de alto nível para uma linguagem (código binário) entendida pela máquina.

Complex Instruction Set Computer (**CISC**): ISA de microprocessadores que usam conjuntos vasto de instruções para manipular dados. Os processadores CISC podem levar vários ciclos de relógio para realizar uma operação. As instruções possuem tamanho variável. Exemplos de processadores CISC são: família Motorola 68000 e Intel x86.

Construção: Elemento, com sintaxe própria, usado para construir uma instrução ou um bloco da linguagem.

Corpo: Para um determinado bloco/construção da linguagem, consiste numa sequência de instruções da linguagem, i.e., *sentences*. Geralmente este corpo em MiADL é delimitado por parênteses (caso das funções e do comportamento comum de cada grupo de operações).

Cycle-accurate: Simulador com precisão ao nível dos ciclos de relógio do alvo.

***Debugger* / Depurador**: Um programa usado para testar e encontrar erros (*bugs*) noutros programas. O *debugger* permite ao programador parar um programa em determinadas posições e examinar o conteúdo de determinadas posições de memória ou registos.

***Decoder* / Descodificador**: Em hardware refere-se a um circuito combinatório que usa os valores de entrada para seleccionar uma saída específica. Em software, é uma rotina que a partir do valor binário da instrução, determina qual a operação que deve ser realizada. Existem várias entradas e várias saídas, havendo apenas uma saída de cada vez, que depende do valor das entradas.

***Disassembler*:** Programa que efectua o processo de geração contrário ao de um *assembler*.

Don't-care: Bit cujo valor binário é indiferente que seja 0 ou 1.

***Executable and Linking Format* (ELF)**: Formato de ficheiros desenvolvido originalmente pela *Unix Systems Laboratories*. É padrão de formato de ficheiros objecto no sistema operativo Linux.

***Explicitly Parallel Instruction Computing* (EPIC)**: Método segundo o qual as instruções são arranjadas de forma a serem ordenadas para serem explicitamente paralelas. Ou seja, várias instruções podem ser executadas de uma só vez, assumindo que o hardware o permite. A arquitectura do IA-64, incluindo processadores Itanium, é desenhada para o maior proveito das instruções arranjadas desta maneira.

Function template: Função parameterizada por tipos, valores, ou *templates*. A função gerada a partir da *function template* pode geralmente ser deduzida a partir dos argumentos presentes na chamada da função.

g++: Compilador de linguagem *C++* da GNU.

GNU C Compiler **(gcc)**: Compilador de linguagem C da GNU.

General Purpose Processor (**GPP**) – Computador genérico cujo alvo é executar uma vasta gama de aplicações.

Hazard: Factor que contribui para a paragem da *pipeline*. Inclui dependência de dados, conflitos de recursos ou atrasos no *fetch* a partir da memória.

Hexadecimal (Intel-Format) File **(.hex)**: Ficheiro de texto no formato ASCII, com extensão "*.hex*".

Instruction-Set Architecture (**ISA**): Conjunto de instruções que um CPU é projectado para executar. Um ISA representa o repertório de instruções que os projectistas consideram adequado para determinado CPU. CPUs de diferentes fabricantes podem possuir o mesmo ISA. Por exemplo os processadores AMD implementam o ISA do Intel IA-32 ISA num processador com diferente estrutura. Ou seja, a arquitectura que define o nível de aplicação e que inclui: instruções ao nível do utilizador, modelos de endereçamento, segmentação e registos visíveis pelo utilizador.

Linker **/ Ligador*:*** Também chamado link editor e ligador. Um *linker* é um programa que combina módulos de código objecto para formar um programa executável. Diversas linguagens de programação permitem escrever diferentes blocos de código, chamados módulos, separadamente. Isto facilita a tarefa de programação porque permite dividir um programa em blocos menores e mais fáceis de verificar. A tarefa do *linker* é a de ligar os vários módulos num só. Além de combinar os diversos módulos, o *linker* também substitui endereços simbólicos por endereços reais. Por isso mesmo, num programa de bloco único o *linker* pode também ser necessário.

Mnemónica: Representação de um *opcode*.

Nome: Designação atribuída a uma entidade (ex. variável, função, tipo, etc.) do programa/descrição.

Opcode: Valor binário que é reconhecível por um processador como sendo uma instrução do respectivo ISA.

Parser: Um algoritmo, ou programa, para determinar a estrutura sintáctica duma frase ou cadeia de símbolos em determinada linguagem. Um *parser* tem por entrada uma sequência de *tokens* produzidos por um analisador léxico e gera como saída uma AST.

Phase-accurate: Nível de abstracção, no qual o detalhe inclui a modelação do relógio da máquina e do estado preservado de um ciclo de relógio para o seguinte.

Program Counter (**PC**): Um registo do CPU que possui o endereço da instrução a ser executada.

Reduced Instruction Set Computer (**RISC**): Trata-se duma arquitectura de computador que privilegia um simples e pequeno conjunto de instruções para correr aplicações com *chips* de baixa complexidade, aumentando a velocidade de processamento. É mais rápido do que os equivalentes mais complexos tais como circuitos integrados CISC. Alguns dos microprocessadores RISC conhecidos são: ARM, MIPS e PowerPC. As instruções efectuam apenas uma operação cada, possuem geralmente todas o mesmo tamanho e as operações aritméticas são efectuadas usando registos.

Register Transfer Level (**RTL**): No tipo de descrição HDL, um circuito é modelado através da especificação do fluxo de dados entre um conjunto de registos, os quais são elementos dum projecto cuja transição entre estados se baseia em eventos (transição de nível alto para baixo, ou vice versa) que ocorrem num sinal de relógio. O nível de abstracção RTL é superior ao do nível de *gate* e é inferior ao do nível comportamental. Tipicamente RTL é escrito usando linguagens tais como Verilog ou VHDL.

***Retargetable*/Redireccionável**: Capacidade de um conjunto de ferramentas de software permitir compilar e/ou simular código para diferentes processadores alvo.

***Scope*:** Zona do programa/descrição na qual os nomes têm significado.

Selector: Campo que funciona como chave para determinar uma opção, de entre um conjunto de hipóteses válidas para diferentes valores daquele campo.

***Sentence* (instrução da linguagem)**: Sequência ordenada de *tokens*.

***Single Instruction Multiple Data* (SIMD)**: Arquitectura na qual uma instrução do computador realiza a mesma acção (carregar, calcular, ou armazenar) simultaneamente em duas ou mais partes dos dados.

Sistema Embutido (i.e., ***Embedded System***): Combinação de hardware e software, e por vezes envolvendo partes mecânicas ou outras, projectada para realizar uma função especializada. Em alguns casos, os sistemas embutidos/embebidos fazem parte de sistemas ou produtos maiores, como é o caso do sistema de anti-bloqueio de travagem de um carro. Contrastam com computadores de uso geral, i.e., GPP. Por exemplo, um forno de microondas possui um sistema embutido que recebe comandos de um painel, controla o display, liga e desliga o elemento de aquecimento que cozinha a comida.

Software toolkit, ou toolchain: Conjunto de ferramentas de software usadas para desenvolver, ou usar, um produto (sistema embutido, processador, etc.).

Statement: Instrução da linguagem. Exemplo, instrução *if*.

String: Combinação de uma ou mais letras e números.

Template: Uma classe ou função parametrizada por tipos, valores, ou *templates*.

Time-to-market: Tempo que demora a desenvolver um produto desde uma ideia inicial até ao início de vendas no mercado. Definições precisas dos pontos de início e fim deste período de tempo variam de empresa para empresa e de projecto para projecto.

Token: Antes que o processamento possa ser realizado sobre o texto de entrada, este tem de ser segmentado em unidades linguistas tais como palavras, pontuação, números, ou alfanuméricos. Estas unidades são designadas por *tokens*.

Unidade Funcional: Unidade de um CPU que é responsável pela execução de uma função predefinida, tal como por exemplo: transferir dados para a cache primária, ou executar uma adição em vírgula flutuante.

Very Long Instruction Word (**VLIW**): Arquitecturas que usam palavras de instrução longas para manter ocupadas várias unidades funcionais em paralelo. O escalonamento de instruções é realizado estaticamente pelo compilador e, como tal, exige que este permita a geração de código de elevada qualidade.

ANEXO

1

Gramática de MiADL

A1.1 NOTAÇÃO

Este apêndice apresenta a gramática da linguagem MiADL. Para apresentar a gramática usa-se a conhecida notação designada por *Backus-Naur Form* (BNF)[125], que foi originalmente proposta para especificar a estrutura da sintaxe de linguagens de programação. De forma semelhante à seguida em [150], também aqui serão adicionadas algumas extensões aquela notação para tornar mais simples e compreensível a representação da linguagem MiADL. Assim, usou-se também a notação do operador ´+´ de Kleene (expresso pela notação {..}+) e do operador ´*´ de Kleene (expresso pela notação {..}*) para expressar sequências de zero ou mais ou de um ou mais elementos respectivamente. Em resumo, além do BNF, a notação usada é a seguinte:

• <não terminal>	Símbolos *não terminais* são colocados entre os caracteres < e >.
• ::=	O símbolo '::=' é usado para separar o lado esquerdo do lado direito de uma *regra* da linguagem.
• [..]	Para representar uma *opção*, ou seja, assinalar que os símbolos no interior podem ter zero ou uma ocorrência.
• {..}+	Operador ´+´ de Kleene, para expressar que o que está no interior dos parênteses, é uma sequência que pode ocorrer *uma ou mais vezes*.
• {..}*	Operador ´*´ de Kleene, para expressar que o que está no interior dos parênteses é uma sequência que pode ocorrer *zero ou mais vezes*.
• "caracteres"	Quando se trata de um símbolo terminal composto por uma cadeia de caracteres, esta será apresentada entre aspas.

- ‘ x ‘ Quando se trata de um caracter terminal, este será apresentado entre plicas.

A1.2 GRAMÁTICA DA LINGUAGEM MIADL

<**MiADL_root**> ::=
<description> EOF

<description>::=
[<extern_section>] <isa_section>

<**extern_section**>::=
‘{‘ {<function_prot>}* ‘}’

<isa_section>::=
“isa" <ident> {<isa_properties>}* <isa_block>

<isa_block>::=
‘{‘ [<global_section>] <resources_section> <iformats_section> <groups_section> ‘}’

<**isa_properties**>::=
‘<’ <endianess> ‘,’ <size> ‘>’

<**endianess**>::=
“big” | “little”

// --- SECÇÃO ***GLOBAL***

<**global_section**>::=
‘{‘ {<function_def>|<variable_def>}* [<init_subsection>] ‘}’

// --- SECÇÃO ***RESOURCES***

<resources_section>::=
"resources" ‘{‘ <resources_block> ‘}’

<resources_block>::=
{<memories_decl> | <regfile_decl> | <regs_decls> | <cfunction_def> | <prgcntr>}* [<init_subsection>]

<regfile_decl>::=
"regfile" <ident> ‘[‘ <size> ‘]’ ‘:’ <typename> ‘;’

<regs_decls>::=
<layout_def>
| <regs_decl>

<regs_decl>::=
"register" <idList> ':' <reg_type> ';'

<reg_type>::=
<typename>
| <ident>

<memories_decl>::=
"memory" <memory_decl> { ',' <memory_decl>}* ';'

<memory_decl>::=
<ident> '[' <size> [<memory_factor>] ']'

<memory_factor>::=
'G'
| 'M'
| 'k'

<**init_subsection**>::=
"init" '{' { <assignment>}* '}'

<**prgcntr**>::=
" prgcounter " <ident> {'[' INT_LITERAL ']'}* ';'

// --- SECÇÃO ***IFORMATS***

<iformats_section>::=
"iformats" [<size>] <iformats_block>

<iformats_block>::=
'{' [<reloc_fields_decl>] [<layouts_def>] {<cfunction_def> | <groupfield_decl>}* '}'

<layouts_def>::=
{layout_def}*

<groupfield_decl>::=
"gpfield" <ident> '{' <idListTagged> '}' "in" <groupfieldtemplatelist>

<groupfieldtemplatelist>::=
'{' <idList> '}'

// --- SECÇÃO ***GROUP***

<groups_section>::=
{ <group_section> }+

<group_section>::=
"group" <ident> ‘(‘ <idListTagged> ‘)’ ‘<’ [<ident>] ‘>’ <group_block>

<group_block>::=
‘{‘ {<operation> | <cfunction_def> | <group_behavior> | <group_asm>}* ‘}’

<operation>::=
"operation" <ident> ‘<’ [<const_literal>] ‘>’ ‘{‘ [<behavior>] | [<op_asm>] ‘}’

<behavior>::=
"behavior" [<ident>] ‘(‘ [<idTypedListTagged>] ‘)’ ‘{‘ [complete_expression] ‘}’

<**op_asm**>::=
"asm" ‘{‘ STRING_LITERAL ‘}’

<group_behavior>::=
"behavior" <compound_statement>

<**group_asm**>::=
"asm" ‘(‘ <group_asm_arguments> ‘)’ ‘{‘ STRING_LITERAL ‘,’ [<string_elements>] ‘}’

<**group_asm_arguments**>::=
<asm_type> <ident>

< **string_elements** >::=
<expression> **{ ‘,’** <expression**>}***

<reloc_fields_decl>::=
"relocfields" <id_list> ‘;’

<id_list>::=
<ident> { ‘,’ <ident> }*

<idListTagged>::=
<id_list>

<**idTypedListTagged**>::=
[**<**typename**>]** <ident> { ‘,’ [**<**typename**>]** <ident> }*

<layout_def>::=
("layout" | "reglayout") <ident> [‘:’ <ident>] [‘<’ <size> ‘>’] ‘(‘ <ranges_def> ‘)’ [<layout_block>]

<layout_block>::=
‘{‘ <cfunctions_def> ‘}’

<ranges_def>::=
<range_collapse>

<range_collapse>::=
{range_def}+

<range_def>::=
<range_id> ‘[‘ <range> ‘]’

<range_id>::=
<ident> [‘:’ ‘s’]
| ‘?’
| HEX_LITERAL
| BIN_LITERAL

<range>::=
INT_LITERAL ‘:’ INT_LITERAL
| INT_LITERAL

<cfunctions_def>::=
{<function_def> | <eswitch_def> | <map_def> }+

<cfunction_def>::=
<function_def>
| <eswitch_def>
| <map_def>
| <map_def2>

<function_prot>::=
<typename> <ident> <arg_list> ‘;’

<function_def>::=
<typename> <ident> <arg_list> <function_block>

<arg_list>::=
‘(‘ [<param_def>] { ‘,’ <param_def>}* ‘)’

<param_def>::=
<typename> [‘&’] <ident>

<function_block>::=
<compound_statement>

<eswitch_def>::=
"eswitch" ‘<’ <eswitch_sel> ‘>’ <ident> <eswitch_block>

<eswitch_sel>::=
"layout"
| “??” <ident>
| <primary>

<eswitch_block>::=
<eswitchpair_list>

<eswitchpair_list>::=
‘{‘ <eswitch_pair> { ‘,’ <eswitch_pair>}* ‘}’

<eswitch_pair>::=
<eswitch_key> ‘:’ <primary>

<eswitch_key>::=
<ident>
| <const_literal>
| ‘_’

<**map_def**>::=
"map" ‘<’ <map_selector> ‘>’ <ident> ‘{‘ <map_body> ’}’

<**map_selector**>::=
"layout"
| “??” <ident>
| <primary>

<**map_body**>::=
<map_pair> { ‘,’ <map_pair>}*

<**map_pair**>::=
<map_key> ‘:’ <primary>

<**map_key**>::=
<ident>
| <const_literal>
| ‘_’

<**map_def2**>::=
"map" <ident> ‘<’ INT_LITERAL ‘>’ ‘{‘ <map_body2> ’}’

<**map_body2**>::=
<map_pair2> { ‘,’ <map_pair2>}*

<**map_pair2**>::=
<map_key2> ‘:’ <map_itens2>

<**map_key2**>::=
<const_literal>

<**map_itens2**>::=
STRING_LITERAL {‘|’ STRING_LITERAL}*

<const_literal>::=
HEX_LITERAL
| BIN_LITERAL

<ident>::=
IDENT

<size>::=
INT_LITERAL

<typename>::=
"int" ‘<’ <size> ‘>’
| "int8"
| "int16"
| "int32"
| "int64"
| "uint" ‘<’ <size> ‘>’
| "uint8"
| "uint16"
| "uint32"
| "uint64"
| "void"
| "boolean"

<**asm_type**>::=
"char"
| "string"

///// --- ***STATEMENTS***

<compound_statement>::=
‘{‘ <statements> ‘}’

<statements>::=
{ statement }*

<statement>::=
<primary> DEC ‘;’
| <primary> INC ‘;’
| <assignment>
| <complete_expression> ‘;’
| <variable_def>
| <compound_statement>
| "return" <complete_expression> ‘;’
| <ifStmt>
| <forStmt>
| ‘;’

<ifStmt>::=
: "if" ‘(‘ <complete_expressionWL> ‘)’ <compound_statement> [elifs_stmt] [else_stmt]

<elifs_stmt>::=
: {elif_stmt }+

<elif_stmt>::=
: "elif" ‘(‘ <complete_expressionWL> ‘)’ <compound_statement>

<else_stmt>::=
: "else" <compound_statement>

<variable_def>::=
"var" <id_list> ‘:’ <typename> ‘;’

<assignment>::=
<lvalue> ‘=’ <complete_expression> ‘;’

<**forStmt**>::=

"for" ‘(‘ <loop_conditions> ‘)’ <compound_statement>

<**loop_conditions**>::=

[<loop_init>] ‘;’ [<loop_cond>] ‘;’ [<loop_advance>]

<**loop_init**>::=

<lvalue> ‘=’ <complete_expression>

<**loop_cond**>::=

<expression>

<**loop_advance**>::=

<expression>

// --- **EXPRESSÕES**

<lvalue>::=

<postfix_expression>

<complete_expression>::=

<expression>

<expression>::=

<cond_expression>

<cond_expression>::=

<logicalOrExpression> [‘?’ <expression> ‘:’ <cond_expression>]

<logicalOrExpression> ::=

<logicalAndExpression> { “||” <logicalAndExpression>}*

<logicalAndExpression>::=

<inclusiveOrExpression> { “&&” <inclusiveOrExpression>}*

<inclusiveOrExpression>::=

<exclusiveOrExpression> { ‘|’ <exclusiveOrExpression>}*

<exclusiveOrExpression>::=

<andExpression> { ‘^’ <andExpression>}*

<andExpression>::=

<equalityExpression> { ‘&’ <equalityExpression>}*

<equalityExpression>::=
<**relationalExpression**> { (“!=” | “==”) <relationalExpression> }*

<relationalExpression>::=
<shiftExpression> { (‘<’ | ‘>’ | “<=” | “>=”) <shiftExpression> }*

<shiftExpression>::=
<additiveExpression> { (“<<” | “>>”) <additiveExpression> }*

<additiveExpression>::=
<multiplicativeExpression> {(‘+’ | ‘-‘) <multiplicativeExpression>}*

<multiplicativeExpression>::=
<unaryExpression> { (“::” | ‘*’ | ‘/’ | '%') <unaryExpression>}*

<unaryExpression>::=
‘-‘ <unaryExpression>
| ‘+’ <unaryExpression>
| '~' <unaryExpression>
| ‘!’ <unaryExpression>
| <postfixExpression>
| <ecall>
| ‘(‘ typename ‘) <unaryExpression>

<postfixExpression>::=
“??” <ident>
| <primary> { ‘.’ <ident> [‘(‘ [<expressions_list>] ‘)’] }* ["++" | “--”]

<ecall>::=
: ‘$’ <ident> ‘(‘ [<expressionsList>] ‘)’
;

<expressions_list>::=
<expression_list>

<expression_list>::=
<complete_expression> { ‘,’ <complete_expression> }*

<primary>::=
<ident_primary>
| "true"
| "false"

| ‘(‘ <expression> ‘)’
| INT_LITERAL
| BIN_LITERAL
| HEX_LITERAL
| ‘$’ <ident>
| ‘@’ <ident>
| ‘@’ <ident> ‘<’ < postfixExpression > ’>’
| CHAR_LITERAL
| STRING_LITERAL
| ‘[‘ STRING_LITERAL ‘,’ < string_elements >’]’

<ident_primary>::=
<ident> { ‘.’ <ident> }* [‘(‘ [<expressions_list>] ‘)’]

<complete_expressionWL>::=
<complete_expression>

DIGIT ::=
'0'..'9'

INT_LITERAL ::=
{DIGIT}+

BIN_DIGIT::=
'0'..'1'

BIN_LITERAL::=
“0b" {BIN_DIGIT}+

CHAR_LITERAL::=
'\'' . '\''

HEX_DIGIT::=
'0'..'9' | 'A'..'F' | 'a'..'f'

HEX_LITERAL::=
"0x" {HEX_DIGIT}+

STRING_LITERAL::=
'"' { '"' '"'! | ~('"'|'\n'|'\r') }* '"'

IDENT::=
('a'..'z'|'A'..'Z'|'_') {'a'..'z'|'A'..'Z'|'0'..'9'|'_'}*

ANEXO 2 Operadores de MiADL

Operador	Categoria	Operação	Tipo	Ordem de análise	Precedência
=	Atribuição	Atribuição	Binário	←	15 (baixa)
:?		Condicional	Ternário	←	14
||	Lógico	Disjunção	Binário	→	13
&&	Lógico	Conjunção	Binário	→	12
|	*Bitwise*	Ou	Binário	→	11
^	*Bitwise*	Ou exclusivo	Binário	→	10
&	*Bitwise*	E	Binário	→	9
!=	Relacional	Diferente de	Binário	→	8
= =	Relacional	Igual a	Binário	→	8
<	Relacional	Menor que	Binário	→	7
>	Relacional	Maior que	Binário	→	7
<=	Relacional	Menor ou igual que	Binário	→	7
>=	Relacional	Maior ou igual que	Binário	→	7
<<	Movimentação de bits	Deslocamento de bits à esquerda	Binário	→	6
>>	Movimentação de bits	Deslocamento de bits à direita	Binário	→	6
+	Aritmético	Adição	Binário	→	5
-	Aritmético	Subtracção	Binário	→	5
*	Aritmético	Multiplicação	Binário	→	4
/	Aritmético	Divisão	Binário	→	4
%	Aritmético	Resto de divisão	Binário	→	4
+	Aritmético	Manutenção de sinal	Unário	←	3
-	Aritmético	Inversão de sinal	Unário	←	3
~	*Bitwise*	Complemento	Unário	←	3
!	Lógico	Negação	Unário	←	3

$		Chamada dum *eswitch*	Unário	←	2
@		Chamada de um *map*	Unário	←	2
( )		Parêntesis		→	1 (elevada)

Legenda:

→ Da esquerda para a direita

← Da direita para a esquerda

ANEXO 3

Descrição do SPARC

Neste apêndice é apresentada seguidamente a descrição do processador SPARC V8 usando a linguagem MiADL:

```
extern {
}

isa SPARCV8 <big, 32> {

        global {
                var WIM: uint<8>;
                var CWP: uint<8>;
                init{
                        WIM= 0x00;
                        CWP= 0xF0;
                }
        }

        resources {


                reglayout PSR<32>(impl[31:28] ver[27:24] N[23] Z[22] V[21] C[20] UNUSED[19:5]
                                CWP[4:0])
                register Y: uint<32>;
                register psr: PSR;
                memory   DM[5242880];
                regfile  rFile[32]: uint<32>;
                regfile  RB[256]: uint<32>;
                init{
                        psr.N=0;
                        psr.Z=0;
                        psr.V=0;
                        psr.C=0;
                }
```

```
void no_update_at1(int<32> x, int<32> y, int<32> z){ }
void no_update_at2(int<32> x){ }
void no_update_at3(uint<64> x){ }
void no_update_at4(int<64> x){ }
void no_update_at5(uint<32> x, boolean y){ }
void no_update_at6(int<32> x, boolean y){ }

void update_1(int<32> dest, int<32> Vop1, int<32> Vop2 ){
        psr.N = dest >> 31;
        psr.Z = dest == 0;
        psr.V = (( Vop1 & Vop2 & ~dest & 0x80000000) | (~Vop1 & ~Vop2 &  dest &
                0x80000000) );
        psr.C = ((Vop1 & Vop2 & 0x80000000) |      (~dest & (Vop1 | Vop2) &
                0x80000000) );
}
void update_2(int<32> dest, int<32> Vop1, int<32> Vop2 ){
        psr.N = dest >> 31;
        psr.Z = dest == 0;
        psr.V = (( Vop1 & ~Vop2 & ~dest & 0x80000000) | (~Vop1 & Vop2 &  dest &
                0x80000000) );
        psr.C = ((~Vop1 & Vop2 & 0x80000000) | (dest & (~Vop1 | Vop2) &
                0x80000000) );
}
void update_3(int<32> dest){
        psr.N = dest >> 31;
        psr.Z = dest == 0;
        psr.V = 0;
        psr.C = 0;
}
void update_4(uint<64> temp){
        psr.N = (uint<32>) temp >> 31;
        psr.Z = (uint<32>) temp == 0;
        psr.V = 0;
        psr.C = 0;
}
void update_5(int<64> temp){
        psr.N = (uint<32>) temp >> 31;
        psr.Z = (uint<32>) temp == 0;
        psr.V = 0;
        psr.C = 0;
}
void update_6(uint<32> result, boolean temp_v){
        psr.N = result >> 31;
```

```
                psr.Z = result == 0;
                psr.V = temp_v;
                psr.C = 0;
        }
        void update_7(int<32> result, boolean temp_v){
                psr.N = result >> 31;
                psr.Z = result == 0;
                psr.V = temp_v;
                psr.C = 0;
        }
        void trap_reg_window_overflow(){
                var x, y:int<32>;
                var i: uint<8>;

                WIM = (WIM-0x10);
                x = (WIM+14) & 0xFF;
                y = (WIM+16) & 0xFF;
                for ( i=0; i<16; i++) {
                        DM.write(RB.read(x)+(i<<2), RB.read(y+i));
                }
        }
        void trap_reg_window_underflow(){
                var x, y: int<32>;
                var i: uint<8>;

                x = (WIM+14) & 0xFF;
                y = (WIM+16) & 0xFF;
                for ( i=0; i<16; i++) {
                RB.write(y+i, DM.read(RB.read(x)+(i<<2)));
                }
                WIM = (WIM+0x10);
        }
} // fim de resources

iformats <32> {  //--- instruction formats

layout F1  (0b01[31:30] disp30[29:0])
layout F2_1  (0b00[31:30] rd[29:25] 0b000[24:22] imm22[21:0])
layout F2_2_nop  (0b00000001000000000000000000000000[31:0])
layout F2_2_sethi  (0b00[31:30] rd[29:25] 0b100[24:22] imm22[21:0])
layout F2_3  (0b00[31:30] an[29] cond[28:25] 0b010[24:22] disp22:s[21:0])
layout F3_1  (op[31:30]  rd[29:25] op3[24:19] rs1[18:14] 0b0[13] asi[12:5] rs2[4:0]){
        int<32> getSecondOp_f31(){
```

```
            return rFile.read(F3_1.rs2);
        }
    }
    layout F3_2  (op[31:30]  rd[29:25] op3[24:19] rs1[18:14] 0b1[13] simm13:s[12:0]){
        int<32> getSecondOp_f32(){
            return F3_2.simm13;
        }
    }
    layout F3_3  (0b10[31:30]  rd[29:25] op3[24:19] rs1[18:14] 0b00000000000000[13:0])
    layout F3_4  (0b10[31:30] ?[29] cond[28:25] 0b111010[24:19] rs1[18:14] 0b0[13] ?[12:5] rs2[4:0])
    layout F3_5  (0b10[31:30] ?[29] cond[28:25] 0b111010[24:19] rs1[18:14] 0b1[13] ?[12:7]
        imm7[6:0])

    gpfield opcode {op, op3} in {F3_1, F3_2}

    eswitch <layout> secondOp{F3_1:getSecondOp_f31, F3_2:getSecondOp_f32}

    map regs <32> {  0b00000:"%r0"|"g0", 0b00001:"%r1"|"%g1",  0b00010:"%r2"|"%g2",
                     0b00011:"%r3"|"%g3",     0b00100:"%r4"|"%g4",     0b00101:"%r5"|"%g5",
                     0b00110:"%r6"|"g6",      0b00111:"%r7"|"g7",      0b01000:"%r8"|"%o0",
                     0b01001:"%r9"|"%o1",     0b01010:"%r10"|"%o2",    0b01011:"%r11"|"o3",
                     0b01100:"%r12"|"%o4",    0b01101:"%r13"|"%o5", 0b01110:"%r14"|"o6"|"%sp",
                     0b01111:"%r15"|"o7",     0b10000:"%r16"|"%l0",    0b10001:"%r17"|"%l1",
                     0b10010:"%r18"|"%l2",    0b10011:"%r19"|"%l3",    0b10100:"%r20"|"%l4",
                     0b10101:"%r21"|"%l5",    0b10110:"%r22"|"%l6",    0b10111:"%r23"|"%l7",
                     0b11100:"%r24"|"%i0",    0b11001:"%r25"|"%i1",    0b11010:"%r26"|"%i2",
                     0b11011:"%r27"|"%i3",    0b10000:"%r28"|"%i4",    0b11101:"%r29"|"%i5",
                     0b11110:"%r30"|"%i6"|"fp",0b11111:"%r31"|"%i7"  }

    map <layout> asmregimm  {F3_1:["%s", @regs<F3_1.rs2>] , F3_2:["%d", F3_2.simm13]};
    map <layout> asmregimmtrap  {F3_4:["%s", @regs<F3_4.rs2>] , F3_5:["%d", F3_5.imm7]};
}

//===------------ Call Instruction
group callOp (F1) <> {
    operation CALL<> {
        behavior call() { }
        asm {"call"}
    }
    behavior {
        var offset: int<32>;
        offset= pc + (??disp30 << 2);
        rFile.write(15, pc);
```

```
                pc = npc;
                npc = offset;
        }
        asm(string str) {"%s %s", str, ??disp30}
}
//===------------ Unimplemented Instructions
group unimplemented (F2_1) <> {
        operation UNIMP <> {
                behavior unimp() {}
                asm {"unimp"}
        }
        behavior {
                out("Instrucao ilegal");
        }
        asm(string str) {"%s %d", str, ??imm22}

}

//===------------ Nop Instructions
group nopOp (F2_2_nop) <> {
        operation NOP <> {
                behavior nop() { }
                asm {"nop"}
        }
        behavior {
                // Nao possui comportamento
        }
        asm(string str) {"%s", str}
}

//===------------ Nop Instructions
group sethiOp (F2_2_sethi) <> {
        operation SETHI <> {
                behavior sethi() { }
                asm {"sethi"}
        }
        behavior {
                rFile.write(??rd, ??imm22 << 10);
        }
        asm(string str) {"%s %d, %s", str, ??imm22, @regs<??rd> }
}

//===------------ Branch Instructions not dependent of icc
```

```
group Branch_BA_BN (F2_3) <cond> {
        operation BN <0b0000> {
                behavior bn(offset, x, y) {  x+4 }
                asm {"bn"}
        }
        operation BA <0b1000> {
                behavior ba(offset, x, y) { y+(offset<<2) }
                asm {"ba"}
        }
        behavior {
                var curPc: uint<32>;
                curPc= pc;
                if(??an){
                        npc= $opAction(??disp22, npc, curPc);
                        pc= npc;
                        npc= npc+4;
                }
                else{
                        pc= npc;
                        npc= $opAction(??disp22, npc, curPc);
                }
        }
        eswitch <??cond> opAction{0b0000:bn, 0b1000:ba}
        map <??an> annull {0b0:"", 0b1:",a"}
        asm(string str) {"%s %s %s", str, @annull, ??disp22 }
}

//===------------ Branch Instructions dependent of icc
group Branch_on_icc (F2_3) <cond> {
        operation BE <0b0001> {
                behavior be()       { psr.Z }
                asm {"be"}
        }
        operation BLE <0b0010> {
                behavior ble()      { psr.Z ||(psr.N ^ psr.V) }
                asm {"ble"}
        }
        operation BL <0b0011> {
                behavior bl()       { psr.N ^ psr.V }
                asm {"bl"}
        }
        operation BLEU <0b0100> {
                behavior bleu()     { psr.C || psr.Z }
```

```
        asm {"bleu"}
}
operation BCS <0b0101> {
        behavior bcs()    { psr.C }
        asm {"bcs"}
}
operation BNEG <0b0110> {
        behavior bneg() { psr.N }
        asm {"bneg"}
}
operation BVS <0b0111> {
        behavior bvs()    { psr.V }
        asm {"bvs"}
}
operation BNE <0b1001> {
        behavior bne()    { !psr.Z }
        asm {"bne"}
}
operation BG <0b1010> {
        behavior bg()     { !(psr.Z ||(psr.N ^ psr.V )) }
        asm {"bg"}
}
operation BGE <0b1011> {
        behavior bge()    { !(psr.N ^ psr.V) }
        asm {"bge"}
}
operation BGU <0b1100> {
        behavior bgu()    { !(psr.C || psr.Z) }
        asm {"bgu"}
}
operation BCC <0b1101> {
        behavior bcc()    { !psr.C }
        asm {"bcc"}
}
operation BPOS <0b1110> {
        behavior bpos() { !psr.N }
        asm {"bpos"}
}
operation BVC <0b1111> {
        behavior bvc()    { !psr.V }
        asm {"bvc"}
}
behavior {
```

```
                if( !$opAction() & ??an){
                        npc= npc+4;
                        pc= npc;
                        npc= npc+4;
                }
                else{
                        var offset: int<32>;
                        offset= pc + (??disp22 <<2);
                        pc= npc;
                        if($opAction()){
                                npc= offset;
                        }
                        else{
                                npc= npc+4;
                        }
                }
        }

        eswitch <??cond> opAction{0b0001:be, 0b0010:ble, 0b0011:bl,0b0100:bleu, 0b0101:bcs, 0b0110:bneg, 0b0111:bvs, 0b1001:bne, 0b1010:bg, 0b1011:bge, 0b1100:bgu, 0b1101:bcc, 0b1110:bpos, 0b1111:bvc}

        map <??an> annull {0b0:"", 0b1:",a"}
        asm(string str) {"%s %s %s", str, @annull, ??disp22 }
    }

    //===------------ ADD/SUB Instructions
    group addsub (F3_1, F3_2) <opcode> {
        operation ADD <0b10000000> {
                behavior add(a, b) {a+b }
                asm {"add"}
        }
        operation SUB <0b10000100> {
                behavior sub(a, b) {a-b }
                asm {"sub"}
        }
        operation ADDX <0b10001000> {
                behavior addx(a, b) { a+b+psr.C }
                asm {"addx"}
        }
        operation SUBX <0b10001100> {
                behavior subx(a, b) { a-b-psr.C }
                asm {"subx"}
```

```
    }
    operation ADDCC <0b10010000> {
        behavior addcc(a, b) { a+b }
        asm {"addcc"}
    }
    operation ADDXCC <0b10011000> {
        behavior addxcc(a, b) {a+b+psr.C }
        asm {"addxcc"}
    }
    operation SUBCC <0b10010100> {
        behavior subcc(a, b) { a-b }
        asm {"subcc"}
    }
    operation SUBXCC <0b10011100> {
        behavior subxcc(a, b) {a-b-psr.C }
        asm {"subxcc"}
    }
    behavior {
        var tmp:int<32>;

        tmp= $opAction(rFile.read(??rs1), $secondOp());
        rFile.write(??rd, tmp);
        $UPDateAction(tmp, rFile.read(??rs1), $secondOp());
    }
        eswitch <??opcode> opAction {0b10000000:add, 0b10000100:sub, 0b10001000:addx,
    0b10001100:subx, 0b10010000:addcc, 0b10011000:addxcc, 0b10010100:subcc,
    0b10011100:subxcc}

    eswitch <??opcode> UPDateAction{0b10000000:no_update_at1, 0b10010000:update_1,
          0b10001000:no_update_at1,0b10011000:update_1, 0b10000100:no_update_at1,
          0b10010100:update_2, 0b10001100:no_update_at1, 0b10011100:update_2}

    asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Logical Instructions
group logical (F3_1, F3_2) <opcode> {
    operation ANDOP <0b10000001> {
        behavior andop(a, b) { a&b }
        asm {"and"}
    }
    operation ANDCC <0b10010001> {
        behavior andcc(a, b) { a&b }
```

```
        asm {"andcc"}
}
operation ANDN <0b10000101> {
        behavior andn(a, b) { a&(~b) }
        asm {"andn"}
}
operation ANDNCC <0b10010101> {
        behavior andncc(a, b) { a&(~b) }
        asm {"andncc"}
}
operation OROP <0b10000010> {
        behavior orop(a, b) { a|b }
        asm {"or"}
}
operation ORCC <0b10010010> {
        behavior orcc(a, b) { a|b }
        asm {"orcc"}
}
operation ORN <0b10000110> {
        behavior orn(a, b) { a|(~b) }
        asm {"orn"}
}
operation ORNCC <0b10010110> {
        behavior orncc(a, b) { a|(~b) }
        asm {"orncc"}
}
operation XOROP <0b10000011> {
        behavior xorop(a, b) { a^b }
        asm {"xor"}
}
operation XORCC <0b10010011> {
        behavior xorcc(a, b) { a^b }
        asm {"xorcc"}
}
operation XNOR <0b10000111> {
        behavior xnor(a, b) { ~(a^b) }
        asm {"xnor"}
}
operation XNORCC <0b10010111> {
        behavior xnorcc(a, b) { ~(a^b) }
        asm {"xnorcc"}
}
behavior {
```

```
                var tmp: int<32>;

                tmp= $opAction(rFile.read(??rs1), $secondOp());
                rFile.write(??rd,  tmp);
                $UPDateFlgLog(tmp);
        }

        eswitch <??opcode>  opAction{0b10000001:andop, 0b10010001:andcc, 0b10000101:andn,
                0b10010101:andncc, 0b10000010:orop,  0b10010010:orcc,  0b10000110:orn,
                0b10010110:orncc,0b10000011:xorop, 0b10010011:xorcc,  0b10000111:xnor,
                0b10010111:xnorcc}

        eswitch <??opcode> UPDateFlgLog{0b10000001:no_update_at2, 0b10010001:update_3,
                0b10000101:no_update_at2, 0b10010101:update_3, 0b10000010:no_update_at2,
                0b10010010:update_3, 0b10000110:no_update_at2, 0b10010110: update_3,
                0b10000011:no_update_at2, 0b10010011:update_3, 0b10000111:no_update_at2,
                0b10010111:update_3}

        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Multiplication Instructions, Unsigned
group umultiply (F3_1, F3_2) <opcode> {
        operation UMUL <0b10001010> {
                behavior umul(a, b) { (uint<64>) (uint<32>) a * (uint<64>) (uint<32>)b }
                asm {"umul"}
        }
        operation UMULCC <0b10011010> {
                behavior umulcc(a, b) { (uint<64>) (uint<32>) a * (uint<64>) (uint<32>)b }
                asm {"umulcc"}
        }
        behavior {
                var temp:uint<64>;

                temp= $opAction(rFile.read(??rs1), $secondOp());
                rFile.write(??rd, (uint<32>) temp);
                Y= (uint<32>) (temp >> 32);
                $UPDateFlgMul(temp);
        }
        eswitch <??opcode>  opAction{0b10001010:umul, 0b10011010:umulcc}
        eswitch <??opcode>  UPDateFlgMul{0b10001010:no_update_at3, 0b10011010:update_4}
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}
```

```
//===------------ Multiplication Instructions, Signed
group smultiply (F3_1, F3_2) <opcode> {
        operation SMUL <0b10001011> {
                behavior smul(a, b) { (int<64>) a * (int<64>) b }
                asm {"smul"}
        }
        operation SMULCC <0b10011011> {
                behavior smulcc(a, b) { (int<64>) a * (int<64>) b }
                asm {"smulcc"}
        }
        behavior {
                var temp:int<64>;

                temp= $opAction(rFile.read(??rs1), $secondOp());
                rFile.write(??rd, (int<32>) temp);
                Y= (int<32>) (temp >> 32);
                $UPDateFlgMul(temp);
        }
        eswitch <??opcode>  opAction{0b10001011:smul, 0b10011011:smulcc}
        eswitch <??opcode>  UPDateFlgMul{0b10001011:no_update_at4, 0b10011011:update_5}
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Multiply Step Instructions
group multiplyStepInstr (F3_1, F3_2) <opcode> {
        operation MULSCC <0b10100100> {
                behavior mulscc() { }
                asm {"mulscc"}
        }
        behavior {
                var rs1_bit0, operand1, operand2, dest: int<32>;

                rs1_bit0 = rFile.read(??rs1) & 1;
                operand1 = ((psr.N ^ psr.V) << 31) | (rFile.read(??rs1) >> 1);
                operand2 = ( ((Y & 1) == 0) ? 0 : $secondOp() );
                dest = operand1 + operand2;
                rFile.write(??rd, dest);
                $UPDateAction(dest, operand1, operand2);
                Y= (rs1_bit0 << 31) | (Y >> 1);
        }
```

```
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
        eswitch <??opcode>  UPDateAction{0b10100100:update_1}
}
//===------------ Divide Instructions, Unsigned
group uDivide (F3_1, F3_2) <opcode> {
        operation UDIV <0b10001110>{
                behavior udiv() {  }
                asm {"udiv"}
        }
        operation UDIVCC <0b10011110>{
                behavior udivcc() {  }
                asm {"udivcc"}
        }
        behavior {
                var  temp: uint<64>;
                var  temp_v: boolean;
                var  result: uint<32>;

                temp = (uint<64>) Y << 32;
                temp = temp | (uint<32>) rFile.read(??rs1);
                temp = temp / (uint<32>) $secondOp();
                result = temp & 0xFFFFFFFF;
                temp_v = ((temp >> 32) == 0) ? 0 : 1;
                if (temp_v){ result = 0xFFFFFFFF;}
                $UPDateFlguDiv(result, temp_v);
                rFile.write(??rd, result);
        }
        eswitch   <??opcode>      UPDateFlguDiv{0b10001110:no_update_at5,   0b10011110:update_6}
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Divide Instructions, Signed
group sDivide (F3_1, F3_2) <opcode> {
        operation SDIV <0b10001111>{
                behavior sdiv() {  }
                asm {"sdiv"}
        }
        operation SDIVCC <0b10011111>{
                behavior sdivcc() {  }
                asm {"sdivcc"}
        }
        behavior {
                var  temp: int<64>;
```

```
            var  temp_v: boolean;
            var  result: int<32>;

            temp = (uint<64>) Y << 32;
            temp = temp | (uint<32>) rFile.read(??rs1);
            temp = temp / (int<32>) $secondOp();
            result = temp & 0xFFFFFFFF;
            temp_v = (((temp >> 31) == 0) |((temp >> 31) == -1)) ? 0 : 1;
            if (temp_v) {
            if (temp > 0) {result = 0x7FFFFFFF; }
            else {result = 0x80000000;}
            }
            $UPDateFlgsDiv(result, temp_v);
            rFile.write(??rd, result);
    }
    eswitch    <??opcode>    UPDateFlgsDiv{0b10001111:no_update_at6,    0b10011111:update_7}

    asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Load Integer Instructions
group LoadIntegerOp (F3_1, F3_2) <opcode> {
    operation LDSB <0b11001001>{
            behavior ldsb(a, b) { (int<32>)(int<8>)DM.readByte(a + b) }
            asm {"ldsb"}
    }
    operation LDSH <0b11001010>{
            behavior ldsh(a, b) { (int<32>)(int<16>) DM.readHalfWord(a + b) }

            asm {"ldsh"}
    }
    operation LDUB <0b11000001>{
            behavior ldub(a, b) { DM.readByte(a + b) }
            asm {"ldub"}
    }
    operation LDUH <0b11000010>{
            behavior lduh(a, b) { DM.readHalfWord(a + b)}
            asm {"lduh"}
    }
    operation LD <0b11000000>{
            behavior ld(a, b) {  DM.read(a + b)}
            asm {"ld"}
    }
```

```
        operation LDD <0b11000011>{
                behavior ldd(a, b) { DM.read(a + b) }
                asm {"ldd"}
        }

        behavior {
                var temp: int<32>;
                if(??op3 == 0x3){
                        temp= DM.read(rFile.read(??rs1)+ $secondOp() + 4);
                }
                rFile.write(??rd, $opAction(rFile.read(??rs1), $secondOp ()));
                if(??op3 == 0x3){
                        rFile.write (??rd+1, temp);             }
        }

        eswitch <??opcode>  opAction{0b11001001:ldsb, 0b11001010:ldsh, 0b11000001:ldub,
                        0b11000010:lduh,0b11000000:ld, 0b11000011:ldd}

        asm(string str) {"%s [%s + %s],%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Store Integer Instructions
group StoreIntegOp (F3_1, F3_2) <opcode> {
        operation STB <0b11000101>{
                behavior stb(a, b, c) { DM.writeByte(a + b, (int<8>)c ) }
                asm {"stb"}
        }
        operation STH <0b11000110>{
                behavior sth(a, b, c) { DM.writeHalfWord(a + b, (int<16>)c ) }
                asm {"sth"}
        }
        operation ST <0b11000100>{
                behavior st(a, b, c)  { DM.write(a+b, c) }
                asm {"st"}
        }
        operation STD <0b11000111>{
                behavior stdop(a, b, c) { DM.write(a+b, c) }
                asm {"std"}
        }
        behavior {
                $opAction(rFile.read(??rs1), $secondOp(), rFile.read(??rd));
                if(??op3 == 0x7){
                         DM.write(rFile.read(??rs1) + $secondOp() + 4, rFile.read(??rd+1));
```

```
                }
        }
        eswitch <??opcode> opAction{0b11000101:stb, 0b11000110:sth, 0b11000100:st,
0b11000111:stdop}
        asm(string str) {"%s %s, [%s + %s]", str, @regs<??rs1>, @regs<??rd>, @asmregimm}
    }

    //===------------ Load-Store special
    group LoadStoreSpacialOp (F3_1, F3_2) <opcode> {
        operation LDSTUB <0b11001101>{
                behavior ldstub(a, b) {}
                asm {"ldstub"}
        }
        behavior {
                rFile.write(??rd, DM.readByte(rFile.read(??rs1) + $secondOp()));
                DM.writeByte(rFile.read(??rs1) + $secondOp(), 0xFF);
        }
        asm(string str) {"%s [%s + %s],%s ", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
    }

    //===------------ Load-Store special
    group SwapOp (F3_1, F3_2) <opcode> {
        operation SWAP <0b11001111>{
                behavior swap(a, b) { }
                asm {"swap"}
        }
        behavior {
                var  temp: int<32>;

                temp= DM.read(rFile.read(??rs1) + $secondOp());
                DM.write(rFile.read(??rs1) + $secondOp(), rFile.read(??rd));
                rFile.write(??rd, temp);
        }

        asm(string str) {"%s [%s + %s],%s ", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
    }

    //===------------  Write Y
    group WriteY (F3_1, F3_2) <opcode> {
        operation WRY <0b10110000>{
                behavior wry() {  }
                asm {"wr"}
        }
```

```
        behavior {
                Y= rFile.read(??rs1) ^ $secondOp();
        }
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, "%y"}
}

//===------------ Read Y

group ReadY (F3_3) <op3> {
        operation RDY <0b101000>{
                behavior rdy() {}
                asm {"rd"}
        }
        behavior {
                rFile.write(??rd, Y);
        }
        asm(string str) {"%s %s, %s ", str,"%y", @regs<??rd>}
}

//===------------ Save Instructions
group saveOp (F3_1, F3_2) <opcode> {
        operation SAVE <0b10111100> {
                behavior save() { }
                asm {"save"}
        }
        behavior {
                var tmp:int<32>;
                var i: uint<8>;

                tmp = rFile.read(??rs1) + $secondOp();
                for (i=16; i<32; i++) {
                RB.write((CWP + i) & 0xFF, rFile.read(i));
                }
                for (i=0; i<8; i++) {
                        rFile.write(i+24, rFile.read(i+8));
                }
                CWP = (CWP-0x10);
                if (CWP == WIM) {trap_reg_window_overflow();}
                for (i=8; i<24; i++) {
                        rFile.write(i, RB.read((CWP + i) & 0xFF));
                }
                rFile.write(??rd, tmp);
        }
```

```
    asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Restore Instructions
group restoreOp (F3_1, F3_2) <opcode> {
    operation RESTORE <0b10111101> {
        behavior restore() { }
        asm {"restore"}
    }
    behavior {
        var tmp:int<32>;
        var i: uint<32>;

        tmp = rFile.read(??rs1) + $secondOp();
        for (i=8; i<24; i++) {
            RB.write((CWP + i) & 0xFF, rFile.read(i));
        }
        for (i=0; i<8; i++) {
        rFile.write(i+8, rFile.read(i+24));
        }
        CWP = (CWP+0x10);
        if (CWP == WIM) {trap_reg_window_underflow();}
        for (i=16; i<32; i++) {
        rFile.write(i, RB.read((CWP + i) & 0xFF));
        }
        rFile.write(??rd, tmp);
    }
    asm(string str) {"%s %s,%s,%s", str,  @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Shift Instructions
group shifts (F3_1, F3_2) <opcode> {
    operation SLL <0b10100101> {
        behavior sll(a, b) { a << b}
        asm {"sll"}
    }
    operation SRL <0b10100110> {
        behavior srl(a, b) { ((uint<32>)a )>> ((uint<32>) b)}
        asm {"srl"}
    }
    operation SRA <0b10100111> {
        behavior sra(a, b) { ((int<32>)a )>> ((int<32>) b)}
        asm {"sra"}
```

```
        }
        behavior {
                rFile.write(??rd, $opAction(rFile.read(??rs1), $secondOp()));
        }
        eswitch <??opcode>  opAction{0b10100101:sll, 0b10100110:srl, 0b10100111:sra}
        asm(string str) {"%s %s,%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Jump with Link Instruction
group jmplOP (F3_1, F3_2) <opcode> {
        operation JMPL <0b10111000> {
                behavior jmpl() { }
                asm {"jmpl"}
        }
        behavior {
                rFile.write(??rd, pc);
                pc = npc;
                npc = rFile.read(??rs1) + $secondOp();
        }
        asm(string str) {"%s %s+%s,%s", str, @regs<??rs1>, @asmregimm, @regs<??rd>}
}

//===------------ Trap Instructions
group traps (F3_4, F3_5) <cond> {
        operation TA <0b1000> {
                behavior ta() {1 }
                asm {"ta"}
        }
        operation TN <0b0000> {
                behavior tn() {0 }
                asm {"tn"}
        }
        operation TNE <0b1001> {
                behavior tne() { ~psr.Z}
                asm {"tne"}
        }
        operation TE <0b0001> {
                behavior te() {psr.Z }
                asm {"te"}
        }
        operation TG <0b1010> {
                behavior tg() {~(psr.Z | (psr.N ^ psr.V)) }
                asm {"tg"}
```

```
}
operation TLE <0b0010> {
        behavior tle() {psr.Z | (psr.N ^ psr.V ) }
        asm {"tle"}
}
operation TGE <0b1011> {
        behavior tge() {~(psr.N ^ psr.V) }
        asm {"tge"}
}
operation TL <0b0011> {
        behavior tl() {psr.N ^ psr.V }
        asm {"tl"}
}
operation TGU <0b1100> {
        behavior tgu() {~(psr.C | psr.Z) }
        asm {"tgu"}
}
operation TLEU <0b0100> {
        behavior tleu() {psr.C | psr.Z }
        asm {"tleu"}
}
operation TCC <0b1101> {
        behavior tcc() {~psr.C }
        asm {"tcc"}
}
operation TCS <0b0101> {
        behavior tcs() {psr.C }
        asm {"tcs"}
}
operation TPOS <0b1110> {
        behavior tpos() {~psr.N }
        asm {"tpos"}
}
operation TNEG <0b0110> {
        behavior tneg() {psr.N }
        asm {"tneg"}
}
operation TVC <0b1111> {
        behavior tvc() {~psr.V }
        asm {"tvc"}
}
operation TVS <0b0111> {
        behavior tvs() {psr.V }
```

```
            asm {"tvs"}
        }
        behavior {
            if($opAction()){
                out(“Trap instruction”);
            }
        }
        eswitch <??cond>  opAction{0b0000:tn, 0b0001:te, 0b0010:tle, 0b0011:tl,  0b0100:tleu,
                0b0101:tcs, 0b0110:tneg, 0b0111:tvs, 0b1000:ta, 0b1001:tne, 0b1010:tg,
                0b1011:tge, 0b1100:tgu, 0b1101:tcc, 0b1110:tpos, 0b1111:tvc}

        asm(string str) {"%s %s", str,  @asmregimmtrap}
    }
    }
```

ANEXO
4
Descrição do ARM

Neste apêndice é apresentada seguidamente a descrição do processador ARM usando a linguagem MiADL:

```
//                    --- Extern  ----
extern{
        int<64> format_rlstr( int32 mask);
        int<32> one_count(int<15> v);
        int<32> lsright(int32 x, int32 y); // logic shift right
        int<32> lsleft(int32 x, int32 y); // logic shift left
        int<32> roright(int32 x, int32 y); // rotate right
        int<32> asright(int32 x, int32 y); // aritmetic shift right
        int<64> concat(int32 x, int32 y);
        int<1> InAPrivilegedMode();
        void assert(boolean x);
}

//                    --- ISA definition ----
isa ARM < little, 32>{

       global{
              uint<32> rotate_right(uint<32> x, uint<32> y) {
                     return (x >> y ) | (x << (32 - y));
              }
              int<1> carryFrom(int<32> src1, int<32> src2, int<64> res){
                     return readBit(src1,31) ^ readBit(src2, 31) ^ readBit(res, 32);
              }
              int<1> overflowFrom(int<64> res) {
                     return readBit(res, 32) ^ readBit(res, 31);
              }
              var shifter_carry_out: int<1>;
```

```
}
resources {
          reglayout PSW<32> (NF[31] ZF[30] CF[29] VF[28] QF[27] ?[26:8] IF[7]
                              FF[6] TF[5] MOD[4:0])
          register CPSR, SPSR: PSW;
          regfile rFile[16]: int<32>;
          memory Memory[4M];

          void update1(int<32> src1, int<32> src2, int<64> resultofoperation){
                    CPSR.NF= readBit(resultofoperation, 31);
                    CPSR.ZF= (resultofoperation == 0) ? 1 : 0;
                    CPSR.CF= carryFrom(src1, src2, resultofoperation);
                    CPSR.VF= overflowFrom(resultofoperation);
          }
          void update2(int<32> src1, int<32> src2, int<64> resultofoperation){
                    CPSR.NF= readBit(resultofoperation, 31);
                    CPSR.ZF= ((resultofoperation == 0) ? 1 : 0);
                    CPSR.CF= ((carryFrom(src1, src2, resultofoperation) == 0) ? 1 : 0);

                    CPSR.VF= overflowFrom(resultofoperation);
          }
          void update3(int<32> src1, int<32> src2, int<64> resultofoperation){
                    CPSR.NF= readBit(resultofoperation, 31);
                    CPSR.ZF= ((resultofoperation == 0) ? 1 : 0);
                    CPSR.CF= shifter_carry_out;
          }
          void update_after_mult(int<32> resultofoperation) {
                    CPSR.NF= readBit(resultofoperation, 31);
                    CPSR.ZF= ((resultofoperation == 0) ? 1 : 0);
          }
          void update_after_lmult(int<64> resultofoperation) {
                    CPSR.NF= readBit(resultofoperation, 63);
                    CPSR.ZF= ((resultofoperation == 0) ? 1 : 0);
          }
          void update4(int<32> src1, int<32> src2, int<64> resultofoperation){
                    CPSR.NF= readBit(resultofoperation, 31);
                    CPSR.ZF= ((resultofoperation == 0) ? 1 : 0);
                    CPSR.CF= shifter_carry_out;
          }

          int32 eq(){ return CPSR.ZF;}
          int32 ne(){ return ~CPSR.ZF;}
          int32 cshs(){return CPSR.CF;}
```

```
            int32 cclo(){return ~CPSR.CF;}
            int32 mi(){return CPSR.NF;}
            int32 pl(){return ~CPSR.NF;}
            int32 vs(){return CPSR.VF;}
            int32 vc(){return ~CPSR.VF;}
            int32 hi(){return (CPSR.CF & (~CPSR.ZF));}
            int32 ls(){return ~(CPSR.CF & (~CPSR.ZF));}
            int32 ge(){return (CPSR.NF == CPSR.VF);}
            int32 lt(){return (CPSR.NF ^ CPSR.VF);}
            int32 gt(){return ~(CPSR.ZF | (CPSR.NF ^ CPSR.VF));}
            int32 le(){return (CPSR.ZF | (CPSR.NF ^ CPSR.VF));}
            int32 al(){return 1;}
            int32 nv(){return 0;}
            prgcounter rFile[15];
  }

//          --- iformats definition ----
  iformats <32> {
    layout dataProcessImmedShift:dpis ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16]
            rd[15:12] sa[11:7] shifti[6:5] 0b0[4] rm[3:0] )   {
            int<32> lsl_i() {
                    var  vRm, shifter_operand: int<32>;

                    vRm=rFile.read(dpis.rm);
                    shifter_carry_out = ( dpis.sa==0 ?  CPSR.CF : readBit(vRm, 32-dpis.sa) );
                    shifter_operand = (dpis.sa==0? vRm:  lsleft(vRm, dpis.sa) );
                    return shifter_operand;
            }
            int<32> lsr_i() {
                    var  vRm, shifter_operand: int<32>;

                    vRm=rFile.read(dpis.rm);
                    shifter_carry_out = (dpis.sa==0 ? readBit(vRm, 31) : readBit(vRm, dpis.sa-1) );
                    shifter_operand = (dpis.sa==0 ? 0: lsright(vRm, dpis.sa) );
                    return shifter_operand;
            }
            int<32> asr_i() {
                    var vRm, shifter_operand: int<32>;

                    vRm=rFile.read(dpis.rm);
                    shifter_carry_out = (dpis.sa==0 ? readBit(vRm, 31) : readBit(vRm, dpis.sa-1) );
                    shifter_operand = ( dpis.sa==0 ?  (readBit(vRm, 31) == 0 ? 0 : 0xFFFFFFFF) :
                                    asright(vRm, dpis.sa) );
```

```
        return shifter_operand;
    }
    int<32> ror_i() {
        var vRm, shifter_operand: int<32>;

        vRm=rFile.read(dpis.rm);
        shifter_carry_out = (dpis.sa==0 ? readBit(vRm, 0) : readBit(vRm, dpis.sa-1) );
        shifter_operand = (dpis.sa==0 ? (lsleft(CPSR.CF, 31) | lsright( vRm, 1)) :
                roright(vRm, dpis.sa) );
        return shifter_operand;
    }
eswitch<dpis.shifti>  arg1  {  0b00:dpis.lsl_i,  0b01:dpis.lsr_i,  0b10:dpis.asr_i,  0b11:dpis.ror_i  }

map<dpis.shifti> asmdpis { 0b00: ["%s", sa == 0b00000 ? "" : [",lsl #%d", sa] ], 0b01: ["%s", sa ==
        0b00000 ? ",lsr #0" : [",lsr #%d", sa] ], 0b10: ["%s", sa == 0b00000 ?  ",asr #0" :
    [",asr #%d", sa] ], 0b11: ["%s", sa == 0b00000 ? ",rrx" : [",ror #%d", sa] ]}
}

layout dataProcessRgShift:dprs ( cond[31:28] 0b000[27:25] op[24:21] s[20] rn[19:16]
        rd[15:12] rs[11:8] 0b0[7] shiftr[6:5] 0b1[4] rm[3:0] ) {
    int<32> lsl_r() {
        var shift, vRm, vRs, shifter_operand: int<32>;

        vRm=rFile.read(dprs.rm);
        vRs= rFile.read(dprs.rs);
        shift= readRange(vRs,7,0);
        shifter_operand = (shift ==0 ?vRm : ( shift<32 ? lsleft(vRm, shift) : 0));
        shifter_carry_out = (shift==0? CPSR.CF : (shift<32 ? readBit(vRm, 32-shift) :
                (shift==32 ? readBit(vRm,0) : 0 )));
        return shifter_operand;
    }
    int<32> lsr_r() {
        var shift, vRm, vRs, shifter_operand: int<32>;

        vRm=rFile.read(dprs.rm);
        vRs= rFile.read(dprs.rs);
        shift= readRange(vRs, 7,0);
        shifter_operand = (shift ==0 ? vRm : (shift<32 ? lsright(vRm, shift) : 0));
        shifter_carry_out = (shift==0? CPSR.CF : (shift<32 ? readBit(vRm, shift-1) : (
                shift==32 ? readBit(vRm, 31) : 0 )));
        return shifter_operand;
    }
    int<32> asr_r() {
```

```
        var shift, vRm, vRs, shifter_operand: int<32>;

        vRm=rFile.read(dprs.rm);
        vRs= rFile.read(dprs.rs);
        shift= readRange(vRs,7,0);
        shifter_operand = (shift ==0 ? vRm : ( shift<32 ? (int<32>)asright(vRm, shift) :
            (readBit(vRm,31) == 1 ? 0xFFFFFFFF: 0 )));
        shifter_carry_out = (shift==0 ? CPSR.CF : (shift<32 ? readBit(vRm,shift-1) :
            readBit(vRm,31)));
        return shifter_operand;
    }
    int<32> ror_r() {
        var shift, vRm, vRs, shifter_operand: uint<32>;

        vRm=rFile.read(dprs.rm);
        vRs= rFile.read(dprs.rs);
        shift= readRange(vRs,7,0);
        shifter_operand = (shift ==0 ? vRm : (  (readRange(shift,4,0)) ? roright(vRm,
            readRange(vRs,4,0) ) : vRm ) );
        shifter_carry_out = (shift==0 ? CPSR.CF : ((readRange(shift,4,0))==0 ?
            readBit(vRm,31) : readBit( vRm,readRange(shift,4,0)-1)));
        return shifter_operand;
    }
eswitch<dprs.shiftr> arg2 { 0b00: dprs.lsl_r, 0b01: dprs.lsr_r, 0b10: dprs.asr_r, 0b11: dprs.ror_r }

map<dprs.shiftr> asmdprs {0b00: "lsl", 0b01: "lsr, 0b10: "asr", 0b11: "ror" }
}

layout dataProcessImmed:dpi ( cond[31:28] 0b001[27:25] op[24:21] s[20]  rn[19:16]
        rd[15:12] rotate[11:8] immediate[7:0] ) {
    int<32> arg3() {
        var rot, shifter_operand: int<32>;

        rot=dpi.rotate*2;
        shifter_operand = roright(dpi.immediate , rot);
        if(rot==0){
            shifter_carry_out= CPSR.CF;
        } else {
            shifter_carry_out= readBit(shifter_operand,31);
        }
        return shifter_operand;
    }
}
```

```
layout miscellaneous ( cond[31:28] 0b00010[27:23] ?[22:21] 0b0[20] ?[19:5] 0b0[4] ?[3:0] )

layout loadStoreImed:lsi ( cond[31:28] 0b010[27:25]  p[24] u[23] b[22] w[21] l[20] rn[19:16]
                rd[15:12] immediate2[11:0]) {
        int<32> arg4() {
                var index:int<32>;

                index= (int<32>)lsi.immediate2;
                return index;
        }
}

layout loadStoreRegOffset:lsr ( cond[31:28] 0b011[27:25] p[24] u[23] b[22] w[21] l[20]
                rn[19:16] rd[15:12] sa[11:7] shifti[6:5] 0b0[4] rm[3:0] ) {
        int<32> lsl_idx() {
                var vRm, index: uint<32>;

                vRm=rFile.read(lsr.rm);
                index= lsleft(vRm, lsr.sa);
                return index;
        }
        int<32> lsr_idx() {
                var vRm, index: uint<32>;

                vRm=rFile.read(lsr.rm);
                index= (lsr.sa==0 ? 0 : lsright(vRm, lsr.sa));
                return index;
        }
        int<32> asr_idx() {
                var vRm, index: uint<32>;

                vRm=rFile.read(lsr.rm);
                index= (lsr.sa==0 ? (readBit(vRm,31) ==1 ? 0xFFFFFFFF : 0 ) : asright(vRm,
                        lsr.sa));
                return index;
                }
        int<32> ror_idx() {
                var vRm, index: uint<32>;

                vRm=rFile.read(lsr.rm);
                index= (lsr.sa==0 ? ( lsleft(CPSR.CF, 31) | lsright(vRm, 1) ) : roright(vRm,
                        lsr.sa));
```

```
            return index;
        }
    eswitch<lsr.shifti> arg5 { 0b00: lsr.lsl_idx, 0b01: lsr.lsr_idx, 0b10: lsr.asr_idx, 0b11: lsr.ror_idx }

    map <lsr.shifti> asmlsr { 0b00: ["%s", sa == 0b00000 ? "" : [",lsl #%d", sa] ], 0b01: ["%s", sa == 0b00000 ? ",lsr #0" : [",lsr #%d", sa] ], 0b10: ["%s", sa == 0b00000 ? ",asr #0" : [",asr #%d", sa] ], 0b11: ["%s", sa == 0b00000 ? ",rrx" : [",ror #%d", sa] ]}
    }

    layout loadStoreMultiple:lsm ( cond[31:28] 0b100[27:25] p[24] u[23] s2[22] w[21] l[20]
            rn[19:16] registerList[15:0] ) {
        void da( int<32>& start, int<32>& end) {
            var vRn: uint<32>;

            vRn=rFile.read(lsm.rn);
            start= vRn - one_count(lsm.registerList)*4 + 4;
            end= vRn;
        }
        void ia(int<32>& start, int<32>& end) {
            var vRn: uint<32>;

            vRn=rFile.read(lsm.rn);
            start= vRn;
            end= vRn + one_count(lsm.registerList)*4 - 4;
        }
        void db(int<32>& start, int<32>& end) {
            var vRn: uint<32>;

            vRn=rFile.read(lsm.rn);
            start= vRn - one_count(lsm.registerList)*4;
            end= vRn - 4;
        }
        void ib(int<32>& start, int<32>& end) {
            var vRn: uint<32>;

            vRn=rFile.read(lsm.rn);
            start= vRn + 4;
            end= vRn + one_count(lsm.registerList)*4;
        }
    eswitch<lsm.pu> argm { 0b00: lsm.da, 0b01: lsm.ia, 0b10: lsm.db, 0b11: lsm.ib}
    }

    layout moveImediateToStatusImmed:mitsi ( cond[31:28] 0b00110[27:23] r[22] opmsr[21:20]
```

```
                fbit[19]  sbit[18]  xbit[17]  cbit[16]  0b1111[15:12]  rotate[11:8]  immediate[7:0]  ) {

            uint<32> msroperand_imed(){
                return  roright(mitsi.immediate, mitsi.rotate*2);
            }
        }
        layout moveImediateToStatusReg:mitsr ( cond[31:28] 0b00010[27:23] r[22] opmsr[21:20]
                fbit[19]  sbit[18]  xbit[17]  cbit[16]  0b1111[15:12]  0b00000000[11:4]  rm[3:0]  ) {

            uint<32> msroperand_reg(){
                return  rFile.read(mitsr.rm);
            }
        }

        layout moveStatusToRegister:mstr ( cond[31:28] 0b00010[27:23] r[22] opmrs[21:20]
                0b1111[19:16] rd[15:12] 0b000000000000[11:0] )
        layout miscellaneous2 ( cond[31:28] 0b00010[27:23] ?[22:21] 0b0[20] ?[19:8] 0b0[7] ?[6:5]
                0b1[4] ?[3:0] )
        layout multiply_extraLdSt ( cond[31:28] 0b000[27:25] ?[24:8] 0b1[7] ?[6:5] 0b1[4] ?[3:0])
        layout normalMultiply ( cond[31:28] 0b000000[27:22] a[21] s[20] rd2[19:16] rn2[15:12]
                rs[11:8] 0b1001[7:4] rm[3:0])
        layout longMultiply ( cond[31:28] 0b00001[27:23] u2[22] a[21] s[20] rdhi[19:16] rdlo[15:12]
                rs[11:8] 0b1001[7:4] rm[3:0])
        layout branch  ( cond[31:28] 0b101 [27:25] l2[24] target[23:0] )
        layout undefined1 ( cond[31:28] 0b00110[27:23] ?[22] 0b00[21:20] ?[19:5] 0b1[4] ?[3:0])
        layout undefined2 ( cond[31:28] 0b011[27:25] ?[24:5] 0b1[4] ?[3:0] )
        layout miscellaneousswap ( cond[31:28] 0b00010[27:23] swb[22] 0b00[21:20] rn[19:16]
                rd[15:12] 0b00001001[11:4] rm[3:0])
        layout ldstspecial_imm:lsei (cond[31:28] 0b000[27:25] p[24] u[23] 0b1[22] w[21] l[20]
                rn[19:16]  rd[15:12]  immedH[11:8]  0b1[7]  s3[6]  h[5]  0b1[4]  immedL[3:0]  ) {

            int<32> idx_ldstspi() {
                return lsei.offset_8;
            }
        }
        layout ldstspecial_reg:lser (cond[31:28] 0b000[27:25] p[24] u[23] 0b0[22] w[21] l[20]
                rn[19:16] rd[15:12] 0b00001[11:7] s3[6] h[5] 0b1[4] rm[3:0] ) {
            int<32> idx_ldstspr() {
                return rFile.read(lser.rm);
            }
        }
    // --- GPFIELD definition ----
        gpfield ua          {u2, a}    in {longMultiply}
```

```
    gpfield bwl         {b, w, l}   in {loadStoreImed, loadStoreRegOffset}
    gpfield swl         {s2, w, l}  in {loadStoreMultiple}
    gpfield pu          {p, u}  in {loadStoreMultiple}
    gpfield offset_8    {immedH, immedL} in  {ldstspecial_imm}
    gpfield lsh         {l, s3, h} in  {ldstspecial_imm, ldstspecial_reg}
    gpfield opcode2   {op, rn} in {dataProcessRgShift, dataProcessImmed, dataProcessImmedShift}
    gpfield opcode3   {op, rd} in {dataProcessRgShift, dataProcessImmed, dataProcessImmedShift}

// --- ESWITCH definitions ----
    eswitch<??cond> condAction{ 0b0000: eq, 0b0001: ne, 0b0010: cshs, 0b0011: cclo, 0b0100: mi,
                                0b0101: pl, 0b0110: vs,   0b0111: vc, 0b1000: hi, 0b1001: ls, 0b1010:
                                ge,   0b1011: lt, 0b1100: gt, 0b1101: le, 0b1110: al,   0b1111: nv }
    eswitch<layout> arg { dpis: $arg1, dprs: $arg2, dpi: arg3 }
    eswitch <layout> getoffset {lsei:idx_ldstspi, lser:idx_ldstspr}
    eswitch<layout> indexval { lsi: arg4, lsr: $arg5 }
    eswitch<layout> ldmstm_arg {lsm:$argm}
    eswitch<layout> getMSROperand{mitsi:msroperand_imed, mitsr:msroperand_reg}

    map<??cond> cond_sym{ 0b0000: "eq",0b0001: "ne", 0b0010: "cs/hs", 0b0011: "cc/lo", 0b0100:
                                "mi", 0b0101: "pl", 0b0110: "vs", 0b0111: "vc", 0b1000: "hi", 0b1001:
                                "ls", 0b1010: "ge", 0b1011: "lt", 0b1100: "gt", 0b1101: "le", 0b1110: "",
                                0b1111: "nv" }
    map<layout> ss { dpis:["R%d %s", dpis.rm, @asmdpis], dprs:["R%d,%s R%d", dprs.rm,
                                @asmdprs, dprs.rs], dpi:["#%d, %d", dpi.imediate, 2*(dpi.rotate)] }
    map <layout> ss2 { lsi: ["R%d, #%c%d", lsi.rn, @uasm, lsi.immediate2], lsr:["[R%d,%cR%d%s]",
                                lsr.rn, @uasm, lsr.rm, @asmlsr] }
    map <layout> addrmode{lser:["R%d", lser.rm] , lsei:["#%d", lsei.offset_8] }
    map <layout> msrasmaddr {mitsr:["R%d", mitsr.rm] , mitsi:["#%d", mitsi.immediate] }
}

// Groups of Operations
// GROUP: aritmetics --------------------
group aritmetics (dpis, dprs, dpi) <op> {
    operation ADC<0b0101> {
            asm {"adc"}
            behavior adc(vrn, n) {vrn + n + CPSR.CF }
    }
    operation ADD<0b0100> {
            asm {"add"}
            behavior add(vrn, n) { vrn + n }
    }
    operation RSB<0b0011> {
            asm {"rsb"}
```

```
                behavior rsb(vrn, n) { n - vrn}
        }
        operation RSC<0b0111> {
                asm {"rsc"}
                behavior rsc(vrn, n) { n - vrn - 1 + CPSR.CF }
        }
        operation SBC<0b0110> {
                asm {"sbc"}
                behavior sbc(vrn, n) {vrn - n - 1 + CPSR.CF }
        }
        operation SUB<0b0010> {
                asm {"sub"}
                behavior sub(vrn, n) { vrn - n }
        }
        behavior {
                var aux: int<64>;
                if ($condAction() ) {
                        aux= $opAction(rFile.read(??rn), $arg());
                        rFile.write(??rd, readRange(aux,31,0));
                        if (??s){
                                if (??rd == 15){
                                        CPSR = SPSR;
                                }
                                else {
                                        $updateStatus(rFile.read(??rn), $arg(), aux);
                                }
                        }
                }
        }

        eswitch<??op> opAction{0b0101: adc, 0b0100: add, 0b0010: sub, 0b0011: rsb, 0b0110: sbc,
                0b0111: rsc}
        eswitch<??op> updateStatus{ 0b0101: update1, 0b0100: update1, 0b0010: update2, 0b0011:
                update2, 0b0110: update2, 0b0111: update2}

        map<??s> sflag{0b0:"", 0b1:"s"}
        asm(str: string) {"%s%s%c? R%d, R%d, %s", str, @cond_sym, @sflag, ??rd, ??rn, @ss }
}

//GROUP: Logical --------------------
group logical (dpis, dprs, dpi) <op> {
        operation ANDOP<0b0000> {
                asm {"and"}
```

```
                behavior andop(n, vrn) {vrn & n }
        }
        operation BIC<0b1110> {
                asm {"bic"}
                behavior bic(n, vrn) {vrn & (~n) }
        }
        operation EOR<0b0001> {
                asm {"eor"}
                behavior eor(n, vrn) { vrn ^ n }
        }
        operation ORR<0b1100> {
                asm {"orr"}
                behavior orr(n, vrn) { vrn | n }
        }
        behavior {
                var result: int<64>;
                if ($condAction()) {
                        result= $opAction($arg(), rFile.read(??rn));
                        rFile.write(??rd, readRange(result,31,0));
                        if (??s){
                                if(??rd == 15){
                                        CPSR= SPSR;
                                }
                                else{
                                        update4(0, 0, result);
                                }
                        }
                }
        }
        eswitch<??op> opAction{0b0000: andop, 0b1110: bic, 0b0001: eor, 0b1100: orr}

        map<??s> sflag{0b0:"", 0b1:"s"}
        asm(str: string){"%s%s%c? R%d, R%d, %s", str, @cond_sym, @sflag, ??rd, ??rn, @ss}
}

// GROUP: Comparison --------------------
group comparison (dpis, dprs, dpi) <opcode3> {
        operation TST<0b10000000> {
                asm {"tst"}
                behavior tst(n, vrn) { vrn & n}
        }
        operation TEQ<0b10010000> {
                asm {"teq"}
```

```
                behavior teq(n, vrn) { vrn ^ n}
        }
        operation CMP<0b10100000> {
                asm {"cmp"}
                behavior cmp(n, vrn) { vrn - n}
        }
        operation CMN<0b10110000> {
                asm {"cmn"}
                behavior cmn(n, vrn) { vrn + n}
        }
        behavior {
                var alu_out: int<32>;
                var result: int<64>;
                        if ($condAction()) {
                        result= $opAction($arg(), rFile.read(??rn));
                        alu_out= readRange(result,31,0);
                        $updateAction(rFile.read(??rn), $arg(), result);
                }
        }
        eswitch<??op> opAction{ 0b1000: tst, 0b1001: teq, 0b1010: cmp, 0b1011: cmn }
        eswitch<??op> updateAction{ 0b1000: update3, 0b1001: update3, 0b1010: update2,
                0b1011: update1 }

        asm(str: string ){"%s%s R%d, %s", str, @cond_sym, ??rn, @ss}
}

//GROUP: Branch ---------------
group gr_branch (branch) <l2> {
        operation B<0b0> {
                asm {"b"}
                behavior b(n) { }
        }
        operation BL<0b1> {
                asm {"bl"}
                behavior bl(n) { rFile.write(14, n+4)  }
        }
        behavior {
                if ($condAction()) {
                  $opAction(pc);
                  pc= pc + ((int32)??target << 2);
                  npc= pc+4;
                }
        }
```

```
        eswitch<??l2> opAction{ 0b0: b, 0b1: bl }

        asm(str: string){"%s%s %d", str, @cond_sym, ??target}
}

//GROUP: Register Movement --------------------
group register_movement (dpis, dprs, dpi) <opcode2> {
        operation MOV<0b11010000> {
                asm {"mov"}
                behavior mov(n) { n }
        }
        operation MVN<0b11110000> {
                asm {"mvn"}
                behavior mvn(n) { ~n }
        }
        behavior {
                if ($condAction()) {
                        rFile.write(??rd, $opAction($arg()));
                        if (??s){
                                if(??rd == 15){
                                        CPSR= SPSR;
                                }
                                else{
                                        update4(0, 0, $opAction($arg()));
                                }
                        }
                }
        }
        eswitch<??op> opAction{ 0b1101: mov, 0b1111: mvn }

        map<??s> sflag{0b0:"", 0b1:"s"}
        asm(str: string){"%s%s%c? R%d, %s", str, @cond_sym, @sflag, ??rd, @ss}
}

//GROUP: Multiply ------------------
group multiply (normalMultiply) <a> {
        operation MUL<0b0> {
                asm {"mul"}
                behavior mul(rm, rs, rn) { (int<32>) (rm * rs ) }
        }
        operation MLA<0b1> {
                asm {"mla"}
                behavior mla(rm, rs, rn) { (int<32>) (rm * rs + rn )}
```

```
    }
    behavior {
        var result: int<32>;

        if ($condAction()) {
            result=$opAction(rFile.read(??rm), rFile.read(??rs), rFile.read(??rn2));
            rFile.write(??rd2, result);
            if (??s){
                update_after_mult(result);
            }
        }
    }

    eswitch<??a> opAction{ 0b0: mul, 0b1: mla }

    map<??s> sflag{0b0:"", 0b1:"s"}
    map<??a> ss1 { 0b0: ["R%d", ??rs], 0b1: ["R%d, R%d", ??rs, ??rn2] }
    asm(str: string){"%s%s%c? R%d, R%d, %s", str, @cond_sym, @sflag, ??rd2, ??rm, @ss1}
    }

//GROUP: Long Multiply -------------------
group lgmultiply (longMultiply) <ua> {
    operation UMULL<0b00> {
        asm {"umull"}
        behavior umull(vrm, vrs, vrdhi, vrdlo) {(uint<64>) (vrm * (uint<64>)vrs )}
    }
    operation UMLAL<0b01> {
        asm {"umlal"}
        behavior umlal(vrm, vrs, vrdhi, vrdlo) {(uint<64>)(vrm * (uint<64>)vrs
            + concat(vrdhi,vrdlo))}
    }
    operation SMULL<0b10> {
        asm {"smull"}
        behavior smull(vrm, vrs, vrdhi, vrdlo) {(int<64>) (vrm * (int<64>)vrs) }
    }
    operation SMLAL<0b11> {
        asm {"smlal"}
        behavior smlal(vrm, vrs, vrdhi, vrdlo) {(int<64>) ( vrm * (int<64>)vrs
            + concat (vrdhi,vrdlo) ) }
    }
    behavior {
        var result: int<64>;
```

```
                if ($condAction()) {
                        result= $opAction(rFile.read(??rm), rFile.read(??rs), rFile.read(??rdhi),
                                rFile.read(??rdlo));
                        rFile.write(??rdhi, readRange(result, 63, 32) );
                        rFile.write(??rdlo, readRange(result, 31, 0) );
                        if (??s) {
                                update_after_lmult(result);
                        }
                }
        }
        eswitch<??ua> opAction { 0b00:umull, 0b01:umlal, 0b10:smull, 0b11:smlal}

        map<??s> sflag{0b0:"", 0b1:"s"}
        asm(str: string){"%s%s%c? R%d, R%d, R%d, R%d", str, @cond_sym, @sflag, ??rdlo, ??rdhi,
                ??rm, ??rs }
}

//GROUP: Load/Store --------------
group ldstore (lsi, lsr) <bwl> {
        operation LDR<0b001> {
                asm {"ldr"}
                behavior ldr(n, m) { f_ldr(n, m) }
        }
        operation LDRB<0b101> {
                asm {"ldrb"}
                behavior ldrb(n, m) {  f_ldrb(n, m)  }
        }
        operation STR<0b000> {
                asm {"str"}
                behavior str(n, m) { f_str(n, m)  }
        }
        operation STRB<0b100> {
                asm {"strb"}
                behavior strb(n, m) {f_strb(n, m)  }
        }
        operation LDRBT<0b111> {
                asm {"ldrbt"}
                behavior ldrbt(n, m) { f_ldrbt(n, m) }
        }
        operation LDRT<0b011> {
                asm {"ldrt"}
                behavior ldrt(n, m) { rFile.write(n,  f_ldrt( n, m ) ) }
        }
```

```
operation STRBT<0b110> {
        asm {"strbt"}
        behavior strbt(n, m) {f_strbt(n, m)  }
}
operation STRT<0b010> {
        asm {"strt"}
        behavior strt(n, m) {f_strt(n, m) }
}
behavior {
        var address, index: int<32>;

        if ($condAction()) {
                index= $indexval();
                if (??p==0) {
                        if(??u){
                                address= rFile.read(??rn) + index;
                        }
                        else{
                                address= rFile.read(??rn) - index;
                        }
                        rFile.write(??rn, address);
                        $opAction(??rd, address);
                }
                else {
                        if(??u){
                                address= rFile.read(??rn) + index;
                        }
                        else{
                                address= rFile.read(??rn) - index;
                        }
                        if ( ??w==1 ){
                                rFile.write(??rn, address);
                        }
                }
        }
}
eswitch<??bwl> opAction { 0b000:str, 0b001:ldr, 0b010:strt, 0b011:ldrt, 0b100:strb, 0b101:ldrb,
        0b110:strbt, 0b111:ldrbt}

void f_ldr(uint<4> reg, uint<32> address) {
        var v:int<32>;

        v= f_ldrt(reg, address);
```

```
        if (reg == 15) {
                pc = (v & 0xFFFFFFFE);
                npc= pc+4;
                CPSR.TF= readBit(v,0);
        }
        else {
                rFile.write(reg, v);
        }
}
void f_ldrb(uint<4> reg, uint<32> address) {
        var aux, aux1: uint<32>;

        aux= Memory.read(address);
        aux1=readRange(aux,7, 0);
        rFile.write(reg, aux1);
}
void f_str(uint<4> reg, uint<32> address) {
        var aux: uint<32>;

        aux=rFile.read(reg);
        Memory.write(address, aux);
}
void f_strb(uint<4> reg, uint<32> address) {
        var aux, aux1: uint<32>;

        aux= rFile.read(reg);
        aux1= readRange(aux,7, 0);
        Memory.write(address, aux1);
}
void f_ldrbt(uint<4> reg, uint<32> address) {
        var aux, aux1: uint<32>;

         aux= Memory.read(address);
         aux1= readRange(aux,7, 0);
         rFile.write(reg,  aux1);
}
int<32> f_ldrt(uint<4> reg, uint<32> address) {
        var v: int<32>;

        if ( readBit(address,1) == 0 & readBit(address,0) == 0){
                v= Memory.read(address);
        }
        elif ( readBit(address,1) == 0 & readBit(address,0) == 1){
```

```
            v= Memory.read(address);
            v= roright(v, 8);
        }
        elif ( readBit(address,1) == 1 & readBit(address,0) == 0){
            v= Memory.read(address);
            v= roright(v, 16);
        }
        else{
            v= Memory.read(address);
            v= roright(v, 24);
        }
        return v;
    }
    void f_strbt(uint<4> reg, uint<32> address) {
        var aux, aux1: uint<32>;

         aux= rFile.read(reg);
         aux1= readRange(aux,7, 0);
         Memory.write(address, aux1);
    }
    void f_strt(uint<4> reg, uint<32> address) {
        var aux: uint<32>;

         aux= rFile.read(reg);
         Memory.write(address, aux);
    }
    map <??u> uasm {0b0:'-', 0b1:''}
    asm(str: string) {"%s%s% R%d, [%s]", str, @cond_sym, ??rd, @ss2}
}

//GROUP: Load/Store Multiple ---------------------
group ldst_multiple (loadStoreMultiple) <l> {
    operation LDM<0b1> {
        asm {"ldm"}
        behavior ldm(i, rl) { f_ldm(i, rl) }
    }
    operation STM<0b0> {
        asm {"stm"}
        behavior stm(i, rl) { f_stm(i, rl) }
    }
    behavior {
        var address, end_address: int<32>;
```

```
        if ($condAction()) {
                $ldmstm_arg(address, end_address);
                if (??w == 1) {
                        rFile.write(??rn, rFile.read(??rn) + one_count(??registerList)*4);
                }
                address= $lAction(address, ??registerList);
                assert((address - 4) == end_address);
        }
}
eswitch<??l> lAction {0b0: stm, 0b1: ldm}

//----- f_stm
int<32> f_stm(int<32> address, int<32> rList) {
        var  mask, n, vR : int<32>;

        mask= 1;
        for( n= 0; n <= 15; ++n){
                if (rList & mask) {
                        vR= rFile.read(n);
                        Memory.write(address, vR);
                        address = address+4;
                        mask << 1;
                }
        }
        return address;
}
//----- f_ldm
int<32> f_ldm(int<32> address, int<32> rList) {
        var  mask, value, aux, n: int<32>;

        mask=1;
        for( n = 0; n < 15 ;++n ){
                if (rList & mask) {
                        aux= Memory.read(address);
                        rFile.write(n, aux);
                        address = address + 4;
                        mask << 1;
                }
        }
        if (readBit(rList,15)) {
                value= Memory.read(address);
                pc= (value & 0xFFFFFFFE);
                npc= pc+4;
```

```
                CPSR.TF= readBit(value,0);
                address = address+4;
            }
            return address;
        }

        map <??w> wasm{0b0: '',  0b1: "! "}
        map <??u> uasm{0b0: 'd', 0b1: 'i' }
        map <??p> pasm{0b0: 'a', 0b1: 'b' }
        asm(str: string) { "%s%s?%c%c R%d%c?%s", str, @cond_sym, @uasm, @pasm, ??rn, @wasm,
            format_rlstr(??registerList) }
}

//GROUP: SWAP ---------------
group gr_swap (miscellaneousswap) <swb> {
        operation SWP<0b0> {
            asm {"SWP"}
            behavior swp() {}
        }
        behavior {
            var address: uint<32>;

            if ($condAction()) {
              address= rFile.read(??rn);
              rFile.write(??rd,rotate_right(Memory.readWord(address), (address&0x3)<<3));
              Memory.writeWord(address, rFile.read(??rm));
            }
        }
        asm(str: string){"%s%s R%d, R%d, [R%d]", str, @cond_sym,  ??rd, ??rm, ??rn}
}
//GROUP: SWPB ---------------
group gr_swapb (miscellaneousswap) <swb> {
        operation SWPB<0b1> {
            asm {"SWP"}
            behavior swpb() {}
        }
        behavior {
            var address: uint<32>;

            if ($condAction()) {
                address= rFile.read(??rn);
                rFile.write(??rd, Memory.readByte(address));
                Memory.writeByte(address, (uint<8>)rFile.read(??rm));
```

```
                }
        }
        asm(str: string){"%s%sB R%d, R%d, [R%d]", str, @cond_sym, ??rd, ??rm, ??rn}
}

//GROUP: LDRH, LDRSH, LDRSB ---------------
group gr_speciallds (ldstspecial_imm, ldstspecial_reg) <lsh> {
        operation LDRH<0b101> {
                asm {"LDR"}
                behavior ldrh() {}
        }
        operation LDRSH<0b111> {
                asm {"LDR"}
                behavior ldrsh() {}
        }
        operation LDRSB<0b110> {
                asm {"LDR"}
                behavior ldrsb() {}
        }
        behavior {
                var address, offset: uint<32>;
                var data: uint<16>;

                offset= $getoffset();
                if ($condAction()) {
                        if(??p){
                                if(??u) {
                                        address= rFile.read(??rn)+offset;
                                }
                                else{
                                        address= rFile.read(??rn)-offset;
                                }
                        }
                        else {
                                address= rFile.read(??rn);
                        }
                        if(!??h){
                                rFile.write(??rd, (int<8>)Memory.readByte(address));
                        }
                        else{
                                data= Memory.readHalfWord(address);
                                if(address & 1){
                                        rFile.write(??rd, 0xCCCCCCCC);
```

```
                }
                else{
                    if(??s3){
                        rFile.write(??rd, (int<16>)data);
                    }
                    else{
                        rFile.write(??rd, data);
                    }
                }
            }
            if(!??p){
                if(??u){
                    rFile.write(??rn, rFile.read(??rn)+offset);
                }
                else{
                    rFile.write(??rn, rFile.read(??rn)-offset);
                }
            }
            else{
                if(??w){
                    rFile.write(??rn, address);
                }
            }
        }
    }
    eswitch <??lsh> opaction{0b101:ldrh, 0b111:ldrsh, 0b110:ldrsb}
    map <??lsh> opasm{0b101:"H", 0b111:"SH", 0b110:"SB"}
    map <??u> uasm{0b0: "+", 0b1:"-"}
    map <??w> wasm{0b0: "", 0b1:"!"}
    asm(str: string){"%s%s%s R%d, [ R%d, %s %s]", str, @cond_sym, @opasm, ??rd, ??rn,
        @uasm, @addrmode, @wasm}
}
//GROUP: STRH ---------------
group gr_specialstrs (ldstspecial_imm, ldstspecial_reg) <lsh> {
    operation STRH<0b001> {
        asm {"STR"}
        behavior strh() {}
    }
    behavior {
        var address, offset: uint<32>;
        var data: uint<32>;

        offset= $getoffset();
```

```
                if ($condAction()) {
                        if(??p){
                                if(??u){
                                        address= rFile.read(??rn) + offset;
                                }
                                else{
                                        address= rFile.read(??rn) - offset;
                                }
                        }
                        else{
                                address= rFile.read(??rn);
                        }
                        if(address & 1){
                                data= 0xCCCCCCCC;
                        }
                        else{
                                data= rFile.read(??rd);
                        }
                        Memory.writeHalfWord(address, data);
                        if(!??p){
                                if(??u){
                                        rFile.write(??rn, rFile.read(??rn) + offset);
                                }
                                else{
                                        rFile.write(??rn, rFile.read(??rn) - offset);
                                }
                        }
                        else{
                                if(??w){
                                        rFile.write(??rn, address);
                                }
                        }
                }
        }
        map <??u> uasm{0b0: "+", 0b1:"-"}
        map <??w> wasm{0b0: "", 0b1:"!"}
        asm(str: string){"%s%sH R%d, [ R%d, %s %s]", str, @cond_sym, ??rd, ??rn, @uasm,
                @addrmode, @wasm}
        }
    //GROUP: MSR  ---------------
    group gr_move2statusregister (mitsi,mitsr) <opmsr> {
        operation MSR<0b10> {
                asm {"MSR"}
```

```
            behavior msr() {}
        }
        behavior {
            var operand: uint<32>;

            if ($condAction()) {
                operand= $getMSROperand();
                if(??r){
                    SPSR=  resolveStatus(??fbit, ??sbit, ??xbit, ??cbit, operand, SPSR);
                }
                else{
                    CPSR= resolveStatus(??fbit, ??sbit, ??xbit, ??cbit, operand, CPSR);
                }
            }
        }

        //----- resolveStatus
        uint<32> resolveStatus(int<1> f, int<1> s, int<1> x, int<1> c, uint<32> operand, uint<32> out) {
            if( f & InAPrivilegedMode() ){ out= (out & 0xFFFFFF00) | (operand & 0x000000FF); }
            if( s & InAPrivilegedMode() ){ out= (out & 0xFFFF00FF) | (operand & 0x0000FF00); }
            if( x & InAPrivilegedMode() ){ out= (out & 0xFF00FFFF) | (operand & 0x00FF0000); }
            if( c & InAPrivilegedMode() ){ out= (out & 0x00FFFFFF) | (operand & 0xFF000000); }
            return out;
        }
        map <??fbit> fflag{0b0:"", 0b1:"f"}
        map <??sbit> sflag{0b0:"", 0b1:"s"}
        map <??xbit> xflag{0b0:"", 0b1:"x"}
        map <??cbit> cflag{0b0:"", 0b1:"c"}
        map <??r> msrasmdest{0b0: "CPSR_", 0b1: "SPSR_"}
        asm(str: string){"%s%s %s%s, %s", str, @cond_sym, @msrasmdest, @fflag, @sflag, @xflag,
            @cflag, @msrasmaddr}
    }
    //GROUP: MRS ---------------
    group gr_movestatus2register (mstr) <opmrs> {
        operation MRS<0b00> {
            asm {"MRS"}
            behavior mrs() {}
        }
        behavior {
            if ($condAction()) {
                if(??r) { rFile.write(??rd, SPSR); }
                else      { rFile.write(??rd, CPSR); }
            }
```

```
        }
        map <??r> mrsasmdest{0b0: "SPSR", 0b1: "CPSR"}
        asm(str: string){"%s%s R%d, %s", str, @cond_sym, ??rd, @mrsasmdest}
    }
}
```

ANEXO 5

Exemplos de outras descrições de processadores

Este apêndice da tese apresenta um excerto da descrição em MADL de operações do processador ARM, conforme descrito no ficheiro *dpi.mad* disponível em [105]. Assim, seguidamente apresenta-se a descrição das operações de processamento de dados (*and, eor, sub, rsb, add, adc, sbc, rsc, orr, bic,ands, eors, subs, rsbs, adds, adcs, sbcs, rscs, orrs, bics*) descritas no referido ficheiro.

```
# binop with immediate operand, spending one cycle in EX
OPERATION dpi_imm_binop

    VAR rotate:uint<4>;              # the rotate field
              imm:uint<8>;           # the imm field

    opcd:{and, eor, sub, rsb, add, adc, sbc, rsc, orr, bic,ands, eors, subs, rsbs, adds, adcs, sbcs, rscs, orrs, bics};

    SYNTAX   opcd reg_names[rd]^"," reg_names[rn]^", #"^v_oprnd2.dec;
    CODING   cond 001 opcd rn rd rotate imm;

    EVAL
      rotate_right(v_oprnd2, imm, rotate);
      + opcd;

    TRANS
      e_id_ex:   {v_rn = *mRF[rn], dst_buffer = mRF[rd], ex_buffer = mEX[], !id_buffer, v_iflag = *mCPSR[]}

      eval_pred(pred, cond, v_iflag);
      c_oprnd2 = (rotate==0)?v_iflag[1]:v_oprnd2[31];

      e_ex_bf:   {pred>0, bf_buffer = mBF[], !ex_buffer,*dst_buffer = v_rd, !dst_buffer, *mReset[]=(rd,1)};
      e_bf_wb:   {wb_buffer = mWB[], !bf_buffer};
      e_wb_in:   {!wb_buffer};

      e_ex_bf_null:   {pred==0, bf_buffer = mBF[], !!ex_buffer, !!dst_buffer};
      e_bf_wb_null:   {wb_buffer = mWB[], !!bf_buffer};
      e_wb_in_null:   {!!wb_buffer};

      e_ex_in:  {*mReset[], !!ex_buffer, !!dst_buffer};

# binop with immediate shift reg operand, spending one cycle in EX
OPERATION dpi_ishift_binop

    VAR # operand 2 of dpi operations
      oprnd2:{zero_shift, imm_shift};
```

```
      # the operation itself
      opcd:{and, eor, sub, rsb, add, adc, sbc, rsc, orr, bic,
                ands, eors, subs, rsbs, adds, adcs, sbcs, rscs, orrs, bics};

    SYNTAX   opcd reg_names[rd]^"," reg_names[rn]^"," oprnd2;
    CODING   cond 000 opcd rn rd oprnd2;

    EVAL
      + oprnd2;
      + opcd;

    TRANS
      e_id_ex:   {v_rn = *mRF[rn], dst_buffer = mRF[rd], v_rm = *mRF[rm], ex_buffer = mEX[],
                          !id_buffer, v_iflag = *mCPSR[]}

      eval_pred(pred, cond, v_iflag);

      e_ex_bf:   {pred>0, bf_buffer = mBF[], !ex_buffer, *dst_buffer = v_rd, !dst_buffer, *mReset[]=(rd,1)};
      e_bf_wb:   {wb_buffer = mWB[], !bf_buffer};
      e_wb_in:   {!wb_buffer};

      e_ex_bf_null:   {pred==0, bf_buffer = mBF[], !!ex_buffer, !!dst_buffer};
      e_bf_wb_null:   {wb_buffer = mWB[], !!bf_buffer};
      e_wb_in_null:   {!!wb_buffer};

      e_ex_in:   {*mReset[], !!ex_buffer, !!dst_buffer};

# binop with reg shift reg operand, spending two cycles in EX
OPERATION dpi_rshift_binop

    VAR # operand 2 of dpi operations
      oprnd2:{r_lsl, r_lsr, r_asr, r_ror};

      # the operation itself
      opcd:{and, eor, sub, rsb, add, adc, sbc, rsc, orr, bic,
                ands, eors, subs, rsbs, adds, adcs, sbcs, rscs, orrs, bics};

    SYNTAX   opcd reg_names[rd]^"," reg_names[rn]^"," reg_names[rm]^"," oprnd2 reg_names[rs];
    CODING   cond 000 opcd rn rd rs 0 oprnd2 1 rm;

    EVAL
      + oprnd2;
      + opcd;

    TRANS
      e_id_ex_pre:  {v_rn = *mRF[rn], dst_buffer = mRF[rd], v_rm = *mRF[rm], v_rs=*mRF[rs],
                          ex_buffer = mEX[], v_iflag = *mCPSR[]}

      eval_pred(pred, cond, v_iflag);

      e_ex_pre_ex:  {!id_buffer};

      e_ex_bf:   {pred>0, bf_buffer = mBF[], !ex_buffer, *dst_buffer = v_rd, !dst_buffer, *mReset[]=(rd,1)};
      e_bf_wb:   {wb_buffer = mWB[], !bf_buffer};
      e_wb_in:   {!wb_buffer};

      e_ex_bf_null:  {pred==0, bf_buffer = mBF[], !!ex_buffer, !!dst_buffer};
      e_bf_wb_null:  {wb_buffer = mWB[], !!bf_buffer};
      e_wb_in_null:  {!!wb_buffer};

      e_ex_pre_in:          {*mReset[], !!ex_buffer, !!id_buffer, !!dst_buffer};
      e_ex_in:              {*mReset[], !!ex_buffer, !!dst_buffer};
```