Universidade Regional Integrada do Alto Uruguai e Missões

Departamento de Engenharia e Ciência da Computação - DECC
Curso de Informática

AIDA-16: UM MICROPROCESSADOR DESTINADO AO ENSINO UTILIZANDO HARDWARE PROGRAMÁVEL

Santo Ângelo, Junho 2004

AIDA-16: UM MICROPROCESSADOR DESTINADO AO ENSINO UTILIZANDO HARDWARE PROGRAMÁVEL

Rodrigo Bittencourt Motta

Orientador: Msc. Luciano Lores Caimi

Monografia apresentada no curso de Informática da Universidade Regional Integrada do Alto Uruguai e das Missões como requisito parcial para aprovação na disciplina de Trabalho de Conclusão.

Santo Ângelo, Junho 2004

Aos meus pais, Luiz Carlos e Maria Elisabete, pelo amor, torcida e incentivo.

A minha namorada, Mariana Ignaczak, pela compreensão e paciência.

Ao meu orientador, Luciano Caimi, pela ajuda e amizade.

"First they ignore you,
then they ridicule you,
then they fight you,
then you win."

Mohandas Karamchand Gandhi (1869-1948)

Agradecimentos

Enfim, chega o momento de escrever a última e mais difícil parte deste trabalho, pois é complicado citar em um espaço tão pequeno todos aqueles de participaram de uma maneira ou outra desta trajetória acadêmica e possibilitaram esta grande conquista. Primeiramente gostaria agradecer a meus pais, Luiz Carlos e Maria Elisabete, por todo o auxílio financeiro e principalmente pelo incentivo e pela torcida que sempre demonstraram. Agradeço também a minha namorada Mariana Ignaczak, pelos momentos de paciência e compreensão durante este percurso. Um agradecimento especial ao meu orientador, Luciano Caimi, que mais que um orientador foi sempre um amigo e sempre esteve disposto a ajudar na realização deste trabalho. Deixo aqui um agradecimento também para o professor Daniel Mesquita, pelo incentivo ao trabalho com dispositivos programáveis e por mesmo sem ter mais vínculo com a universidade sempre oferecer seu apoio. Não posso deixar de lembrar também do amigo e colega Gabriel Marchesan, pela ajuda com LaTeX e pelos momentos de desespero compartilhados na véspera da entrega deste trabalho. Agradeço aos demais professores do curso pelo conhecimento transmitido durante esta longa caminhada. A todos os colegas e amigos do curso pelos momentos que deixarão saudades, ficam, também, meus agradecimentos. Um agradecimento mais do que especial a Deus, pela minha família, por todos os amigos e pessoas especiais que tenho ao meu lado e, principalmente por me conduzir pelo caminho correto nas horas de maior dificuldade.

Sumário

Resumo

Este trabalho objetiva solucionar algumas das principais dificuldades enfrentadas pelos professores na tarefa de ensinar aos alunos o processo de funcionamento dos dispositivos eletrônicos utilizados nas áreas de Sistemas Digitais e Arquitetura de Computadores em ambiente de sala de aula. São abordadas as principais limitações dos métodos tradicionais empregados no ensino dos processos lógicos e dinâmicos aos quais estes dispositivos são submetidos. Em função disso, é proposto um microprocessador, nomeado AIDA-16, para ser usado como ferramenta de ensino. Este microprocessador é descrito em VHDL e utiliza um dispositivo programável (FPGA) como alvo de prototipação. Neste trabalho é explanado o processo de concepção de cada módulo funcional desta plataforma, relacionando-os com os módulos funcionais das arquiteturas microprocessadas utilizadas atualmente. Os métodos de projeto empregados pela microeletrônica na construção de circuitos digitais com o uso de dispositivos programáveis (chamados *chips* pré-difundidos) através de HDLs também são estudados, com o intuito de esclarecer ao leitor esta nova abordagem de prototipação de *hardware*.

Abstract

This work aims to solve some of main difficulties faced by teachers in their effort to teaching to students the process of operation of electronic devices used in the Digitals Systems and Computer Architecture areas at the classroom's environment. The main limitations of traditional lecturing methods are discussed, regarding their application on teaching dynamic and logical process in which such devices are submitted. Therefore it is proposed a microprocessor, named AIDA-16, to be used as a teaching tool. The microprocessor is described in VHDL and prototyped on a programmable device (FPGA). The text provides a detailed description of each functional module of AIDA-16, and correlates them with the current microprocessor architectures functional modules. Modern digital circuit design methods using programmable devices (called pre-fabricated chips) by means of HDL are described too, providing an explanation about this new hardware prototyping approach.

Lista de Figuras

Lista de Tabelas

Lista de Descrições

Lista de Abreviaturas

ACM *Association for Computing Machinery*

ASIC *Application Specific Integrated Circuit*

CAD *Computer Aided Design*

CI Circuito Integrado

CISC *Complex Instruction Set Computer*

CLB *Configurable Logic Block*

CPI *Clock Cycles Per Instruction*

DMA *Direct Memory Access*

DSP *Digital Signal Processor*

E/S Entrada/Saída

FPD *Field-Programmable Devices*

FPGA *Field-Programmable Gate Array*

FU *Functional Unit*

GPP *General Purpose Processor*

IDE *Integrated Drive Eletronics*

IEEE *Institute of Electrical and Eletronics Engineers*

ILP *Instruction Level Parallelism*

IOB *Input/Output Block*

LIW *Long Instruction Word*

LUT *Look-Up Table*

MPP *Massive Parallel Processors*

PIA *Programmable Interconect Array*

PLD *Programmable Logic Device*

RAM *Random Access Memory*

RISC *Reduced Instruction Set Computer*

RISP *Reconfigurable Instruction Set Processors*

ROM *Read Only Memory*

GPR *General Purpose Register*

SDAC Sistemas Digitais e Arquitetura de Computadores

SRAM *Static Random Access Memory*

UCP Unidade Central de Processamento

ULA Unidade Lógica e Aritmética

USB *Universal Serial Bus*

VHDL *VHSIC Hardware Description Language*

VHSIC *Very High Speed Integrated Circuit*

VLSI *Very Large Scale Integration*

Capítulo 1

Introdução

O ramo da computação é composto por diversas áreas de estudo. A maioria delas envolve raciocínio lógico e dinâmico e conta com aplicações em tempo-real e do mundo-real que são extremamente difíceis de ensinar em ambiente de sala de aula [RUS91]. A escassez de ferramentas que possibilitem que o ensino destas aplicações seja mais dinâmico e conciso deixa os professores em um dilema constante.

Uma destas áreas de estudo é a de Sistemas Digitais e Arquitetura de Computadores (SDAC). O foco de Sistemas Digitais está na forma como dispositivos eletrônicos são projetados a partir de blocos básicos de construção, como portas lógicas e *flip-flops*. Já em Arquitetura de Computadores, o ensino é direcionado ao funcionamento dos componentes que integram os microcomputadores, tais como: microprocessadores, placas-mãe, sistemas de armazenamento de memória e demais periféricos.

O ensino do funcionamento de dispositivos e componentes utilizados na área SDAC torna-se muitas vezes restrito devido à dificuldade de demonstrar como determinadas operações são executadas. Por isso, retratar como tais dispositivos e componentes manipulam os dados recebidos, gerando um resultado, torna-se uma tarefa complicada dada a necessidade de algum tipo de representação ou simulação de determinados comportamentos, aos quais, os circuitos internos destes sistemas estão sujeitos, de modo a fazer com que o aluno consiga vincular seu conhecimento teórico com a experiência prática.

Além disso, as ferramentas de simulação de circuitos digitais disponíveis não têm o mesmo impacto de um sistema real, apesar de demonstrarem o funcionamento de sistemas computacionais com um alto nível de dinamismo, e conseqüentemente, não permitem sua aplicabilidade em tarefas reais.

Dentro deste contexto, a existência de uma plataforma que sirva como base para experimentos práticos, dando suporte a disciplinas que necessitem de um *hardware* de apoio para determinadas aplicações, favorece o processo de aprendizado. Contudo, o uso de plataformas proprietárias exige que as aplicações desenvolvidas para serem usadas sobre elas tenham que adequar-se às suas características (formato e conjunto de instruções, modos de endereçamento, linhas de interrupção, arquitetura do barramento, etc.), impossibilitando dessa forma que se possa “moldar” o *hardware* tornando-o adaptável a necessidade de uma aplicação.

Por outro lado, se métodos tradicionais de prototipação de *hardware* forem utilizados para a construção de uma plataforma própria, qualquer modificação que se deseje realizar no dispositivo acarreta na necessidade de alteração de componentes internos utilizados na sua construção, tornando na maioria das vezes a construção de um novo dispositivo o método mais eficaz. Além disso, no que tange à construção de circuitos digitais complexos como microprocessadores,

a disponibilidade de equipamentos para manipulação, fusão e dopagem do silício bem como para os processos de criação das camadas correspondentes ao leiaute do circuito (posicionamento e conexões dos transistores sobre a pastilha de silício representando as portas lógicas e suas interconexões dentro do circuito), através de fotolitografia[1] com o uso de máscaras e material fotosensível por exemplo, é muito limitada devido a seu custo extremamente elevado [KAH99][AKC00].

Ainda que seja possível realizar a construção do *chip* na pastilha de silício através de empresas internacionais especializadas, existe a necessidade de fornecimento de todo o leiaute inerente ao circuito e, as ferramentas de CAD (*Computer Aided Design*) voltadas para a construção de leiaute não têm uma usabilidade trivial, além de terem um custo relativamente alto.

Como computadores são sistemas digitais, formados ou por uma composição de diversos subsistemas ou por um *chip* VLSI (Very Large Scale Integration), uma excelente maneira de ensinar e aprender as suas capacidades e limitações definitivamente é projetando-os [CAL01]. A junção desta possibilidade com o potencial das Linguagens de Descrição de *Hardware* (HDLs, do ingês *Hardware Description Language*), tais como VHDL (VHSIC[2] *Hardware Description Language*) ou Verilog para o desenvolvimento de circuitos digitais direcionados ao ensino vem crescendo ultimamente. VHDL é uma ferramenta ideal para estudantes aprenderem sobre Arquitetura de Computadores com um alto nível de detalhamento por causa de sua habilidade em modelar sistemas digitais [HUA97]. Utilizando-se dispositivos programáveis, tais como FPGAs (*Field-Programmable Gate Arrays*), PLDs (*Programmable Logic Devices*) ou ASICs (*Application Specific Integrated Circuits*), conjuntamente com uma HDL, pode-se tornar o processo de construção de um circuito digital rápido e eficiente.

O presente trabalho, por propor uma plataforma de *hardware* com características voltadas para o ensino de Arquitetura de Computadores e Sistemas Digitais visando fornecer meios que facilitem aos estudantes realizarem um salto cognitivo dos conceitos aprendidos em sala de aula para a respectiva aplicação destes conceitos na prática e por tencionar sua utilização como uma plataforma própria a ser utilizada na Universidade Regional Integrada (URI - Campus de Santo Ângelo) provendo uma base para experimentos práticos das disciplinas oferecidas pelo curso de Ciência da Computação e, conseqüentemente, ultrapassando as limitações impostas pelos sistemas de simulação de circuitos digitais existentes, agrega ao curso uma repleta gama de aplicações computacionais passíveis de serem realizadas. Além aperfeiçoar e trazer um dinamismo muito maior ao ensino oferecido aos estudantes, este projeto por ter como alvo de implementação um dispositivo programável, permite ainda a possibilidade de redefinição dos aspectos referentes à arquitetura interna, funcionalidade e até mesmo pinagem (interconexões do circuito digital com um sistema externo) da plataforma objetivada, permitindo a adaptabilidade do *hardware* com quaisquer aplicações que necessitem uma estrutura ou comportamento específico do circuito sobre o qual elas irão ser executadas.

Devido a esta maleabilidade possibilitada pela utilização de um dispositivo programável, é possível que diferenças de funcionalidade entre arquiteturas distintas de *hardware* sejam demonstradas através de alterações nos módulos desejados da descrição original da plataforma utilizando uma linguagem de descrição de *hardware* e re-sintetizando-a novamente no dispositivo programável. O uso de um dispositivo programável elimina ainda os problemas impostos pela fabricação física de um circuito de alta complexidade, permitindo a definição de toda a lógica

[1] Processo de reprodução do padrão do circuito de um *chip* em um *waffer* de silício usando luz ultravioleta e estêncil ou máscaras para transferir a imagem fotomecânicamente [INT04].

[2] *Very High Speed Integrated Circuit*

comportamental do circuito a partir de um nível de abstração mais elevado, uma vez que toda a parte física (leiaute do *chip* na pastilha de silício) já está definido no próprio dispositivo programável.

Além disso, a escolha de um dispositivo programável como alvo de implementação favorece o estado da arte da computação reconfigurável que é um ramo novo (relativamente aos demais ramos da computação) da área de Sistemas Digitais e Arquitetura de Computadores (SDAC). Trata-se de sistemas baseados em dispositivos de *hardware* passíveis de modificações após saírem da fábrica. As possibilidades desse ramo de pesquisa são imensas e, como dispositivos deste tipo têm evoluído significativamente nos últimos anos, alcançando elevados níveis de densidade, altos índices de desempenho e menores custos de fabricação, a distância entre eles e os CIs (Circuitos Integrados) tem diminuído significativamente [RIB02]

Para solucionar os problemas expostos anteriormente, o objetivo principal deste trabalho foi o desenvolvimento do projeto de um microprocessador multiciclo[3], chamado AIDA-16 (Arquitetura Instrucional Destinada ao Ensino), utilizando uma linguagem de descrição de *hardware* e sua síntese em um dispositivo programável.

Isso foi realizado primeiramente através de um estudo da organização e funcionamento interno dos microprocessadores que possibilitou definir as características desejáveis bem como aspectos a serem evitados em uma arquitetura com o intuito de favorecer o aprendizado. A partir disso, foi realizada a definição da arquitetura interna, conjunto e formatos de instruções, conjunto de registradores específicos e de propósito geral, módulos funcionais internos com seus canais de comunicação e barramentos de comunicação do AIDA-16.

Terminada a fase de especificação do AIDA-16, os módulos funcionais foram descritos individualmente utilizando a linguagem VHDL e testados através de ferramentas de simulação. Em seguida, os componentes foram interligados através dos canais de comunicação definidos na especificação e simulados novamente em conjunto por meio de testes de execução de instruções para proporcionar a observação do comportamento do circuito mediante diferentes estímulos externos.

Finalmente, o AIDA-16 foi sintetizado em um dispositivo FPGA para permitir a realização de testes reais de funcionalidade do circuito.

1.1 Organização

O presente trabalho está organizado da seguinte forma:

- O Capítulo 2 destaca os diferentes paradigmas arquiteturais usados na construção de microprocessadores, especificando suas principais características e diferenças entre eles. Em seguida, é apresentado o esquema estrutural e funcional de um microprocessador CISC genérico.

- O Capítulo 3 aborda os diferentes métodos de prototipação de circuitos digitais utilizados, com ênfase especial nos dispositivos programáveis. É realizado um estudo mais abrangente em cima dos dispositivos FPGAs, onde são explicados seus recursos de roteamento e funcionamento de componentes internos. São apresentados também a linguagem VHDL, utilizada na descrição de circuitos digitais, e o processo de prototipação física

[3]Estabelecimento de dois ou mais estágios para a execução de uma instrução

de uma descrição. Para finalizar, os equipamentos e *softwares* utilizados no processo de construção do AIDA-16 são abordados.

- O Capitulo 4 descreve o microprocessador projetado. São apresentados os módulos funcionais presentes na arquitetura, bem como sua estrutura e funcionalidade; o conjunto de instruções e registradores definidos; os formatos de representação de instruções e modos de endereçamento utilizados; e sua organização interna conjuntamente com a interface de comunicação entre seus módulos funcionais e ainda alguns testes de simulação realizados. São mostradas também as estatísticas da síntese física do AIDA-16 em FPGA.

- No Capítulo 5 é realizada a conclusão do trabalho, onde também são indicados módulos adicionais que podem ser agregados à plataforma desenvolvida e também sugestões para aplicações futuras.

Capítulo 2

Sistemas Digitais

Um sistema digital é basicamente um circuito elétrico que utiliza uma combinação de dispositivos desenvolvidos para manipular quantidades físicas ou informações que são expressas na forma digital. Normalmente, os sistemas digitais utilizam a representação binária para expressar as informações, onde a unidade atômica é o *bit*, que pode assumir apenas dois valores discretos "0" e "1".

A construção física de um sistema digital é possível com a utilização de componentes que controlam o fluxo de corrente elétrica de forma a implementar as operações lógicas [UYE02]. Entretanto, antes da construção física de um sistema digital, é realizado todo o seu projeto utilizando lógica digital.

Dentre os sistemas digitais de alta complexidade se encontram os microcomputadores, que são constituídos de vários componentes eletrônicos que trabalham conjuntamente e são comandados por um microprocessador. Os microprocessadores são circuitos digitais construídos a partir da interligação de transistores para representar as combinações de portas lógicas necessárias para implementar a lógica do circuito.

As próximas seções apresentam a classificação das arquiteturas utilizadas pelos microprocessadores atuais e descrevem seus principais blocos funcionais, entretanto, o presente capítulo não pretende ser uma discussão completa a respeito dos paradigmas de implementação de uma arquitetura de computadores, mas sim informar as principais características e diferenças entre elas.

2.1 Classificação das Arquiteturas Microprocessadas

O principal objetivo dos projetistas de microprocessadores é aprimorar seu desempenho. Para aumentar o desempenho, pode-se desenvolver um microprocessador com instruções que executem operações complexas diretamente pelo *hardware*, com um ciclo de *clock* por instruções (CPI, do inglês *Clock Cycles Per Instruction*) mais baixo ou com uma freqüência de operação interna mais elevada. Aumentar a performance através de instruções complexas acaba aumentando a complexidade interna dos circuitos do microprocessador e, conseqüentemente, seu processo de construção. A Redução do CPI das instruções exige suporte a técnicas de *pipeline*[4] e/ou a utilização de uma arquitetura superescalar[5]. Já a utilização de uma freqüência de operação in-

[4]Processo de divisão da execução de uma tarefa em diversas etapas

[5]Utilização de múltiplas canalizações de execução

terna mais elevada cada vez se torna mais complicada, já que existe a necessidade de aprimorar o processo de fabricação do *chip* do microprocessador.

Um fator de grande influência no desempenho de um microprocessador é a organização de seus circuitos internos e a forma como eles são projetados para desempenhar as instruções requisitadas. Desta forma, arquiteturas distintas podem realizar uma mesma operação de formas diferentes, devido ao processo pelo qual as instruções de máquina devem passar para serem executadas.

Nas subseções a seguir, as arquiteturas usadas nos microprocessadores comerciais disponíveis atualmente são abordadas com maior profundidade.

2.1.1 Arquitetura CISC

CISC é uma abreviação para *Complex Instruction Set Computer*, a microprogramação é uma característica essencial deste tipo de arquitetura. Cada instrução de máquina é interpretada por um microprograma localizado em uma memória ROM presente no circuito integrado do microprocessador.

A microprogramação permite ao projetista a criação de uma nova instrução combinando uma seqüência de operações contidas no microcódigo, possibilitando inclusive que uma única instrução seja equivalente à execução de grupo inteiro de instruções que anteriormente eram necessárias para a realização de uma determinada tarefa. Na realidade, este grupo de instruções continua a existir dentro do microprograma, mas para referenciar este grupo inteiro basta invocar a nova instrução. Um exemplo disso é a operação de multiplicação que é referenciada com uma única instrução apesar de ser composta por sucessivas instruções de soma e deslocamento. Então, quando uma instrução de multiplicação é recebida, ela deve ser localizada dentro do microprograma a fim de verificar qual é a seqüência de microinstruções que deve ser disparada para a multiplicação ser efetivada.

À medida que a linguagem de máquina (as instruções que o microprocessador reconhece) se torna maior e mais complicada, seu interpretador, o microprograma, se torna maior e mais lento [TAN99]. Além disso, quanto mais instruções existirem no conjunto de instruções de um microprocessador maior é o tempo gasto em decodificação dos códigos de operação destas instruções.

A característica mais marcante de máquinas CISC é o fato de que o *hardware* tende a crescer em complexidade com o aumento do conjunto de microinstruções [UYE02]. Desta forma, apesar de facilitar o trabalho do compilador, que é o programa responsável por traduzir um código de alto nível para a linguagem de máquina, o processo de execução das instruções sofre um retardo, pois estas instruções devem antes ser localizadas dentro do microcódigo e desmembradas em microinstruções, que ainda devem ser decodificadas antes de serem finalmente executadas.

As máquinas CISC apresentam ainda um número pequeno de registradores, sendo que a maioria deles é utilizada para funções específicas durante a execução das instruções. Dessa maneira, existe um grande fluxo de dados entre o microprocessador e a memória para o armazenamento de informações.

A segunda coluna da Tabela 2.1 [TAN99] apresenta as principais características de uma plataforma baseada na arquitetura CISC.

Tabela 2.1: Principais diferenças entre as arquiteturas CISC e RISC

CISC	RISC
Instruções complexas consumindo múltiplos ciclos	Instruções simples consumindo 1 ciclo
Qualquer instrução pode referenciar a memória	Apenas LOADs/STOREs referenciam a memória
Não tem *pipeline*, ou tem pouco	Altamente *pipelined*
Instruções interpretadas pelo microprograma	Instruções executadas pelo *hardware*
Instruções com vários formatos	Instruções com formato fixo
Muitas instruções e modos	Poucas instruções e modos
A complexidade está no microprograma	A complexidade está no compilador
Conjunto único de registradores	Múltiplos conjuntos de registradores

2.1.2 Arquitetura RISC

Uma máquina RISC (*Reduced Instruction Set Computer*) é essencialmente uma arquitetura com um pequeno número de microinstruções, dessa forma, programas do usuário são compilados para seqüências destas microinstruções, e então executadas diretamente pelo *hardware* [TAN99]. Isso elimina a necessidade do microcódigo, presente nas arquiteturas CISC, pois não existe a necessidade de interpretação das instruções recebidas pelo microprocessador, em contrapartida, a demanda pela execução de instruções é maior e mais acessos ao barramento devem ser realizados para busca de novas instruções e operandos na memória principal.

Para contornar este problema o microprocessador incorpora um grande número de registradores para armazenamento de dados temporários, de modo que, as operações aritméticas são realizadas basicamente entre registradores. Os acessos à memória existem apenas para transferir dados de/para registradores. Como a maioria dos programas de usuários utiliza várias vezes as mesmas variáveis ao longo do código, se algumas destas se mantiverem em determinados registradores muitos acessos à memória poderão ser evitados. Outro método para reduzir o número de acessos à memória é o uso de uma maior quantidade de memória *cache* dentro da pastilha do microprocessador para armazenar as próximas instruções que possivelmente serão executadas.

O conjunto de instruções reduzido da arquitetura RISC também proporciona uma redução significante da complexidade da parte de controle, dessa forma, liberando área no *chip* para unidades de execução mais potentes [SRI97]. Além disso, uma outra característica desta arquitetura é o uso de instruções de tamanho fixo e com um número pequeno de parâmetros facilitando o processo de decodificação das instruções e o controle de *pipeline*. Os comandos RISC passam por um número menor de transistores em circuitos mais curtos. Dessa forma cada instrução geralmente necessita apenas um ciclo de *clock* para ser executada. O número de ciclos necessários para completar uma operação inteira depende do número de comandos que constituem esta operação.

Como o microprocessador RISC só executa instruções simples, todo o trabalho conversão de instruções complexas para microinstruções migrou do microcódigo, presente na arquitetura CISC, para o compilador. É claro que isso torna os programas compilados maiores em termos de número de instruções, mas em compensação, faz com que o microprocessador possa executar as instruções recebidas imediatamente, tornando a execução mais rápida.

Investigações sobre as arquiteturas VLSI (*Very Large Scale Integration*) indicaram que uma das maiores limitações no *design* de circuitos digitais é o atraso elétrico penalizado pelas transferências de dados nos limites do *chip* e a ainda limitada quantidade de recursos disponíveis

em um único *chip* [PAT81]. Como as instruções RISC para serem executadas precisam da ativação de um número muito menor de transistores em relação às instruções CISC, este problema é menos relevante para as máquinas RISC.

A terceira coluna da Tabela 2.1 lista as principais características de um microprocessador RISC em comparação com uma máquina CISC.

2.1.3 Arquitetura LIW

Os microprocessadores LIW (*Long Instruction Word*) exploram o paralelismo em nível de instrução (ILP, do inglês *Instruction Level Parallelism*) através da otimização de sua performance com o auxílio da execução paralela de instruções de dados dentro de um único ciclo de *clock* [RAU93]. Para isso, são utilizadas de instruções longas compostas por múltiplas operações, cada uma das quais, equivale a uma operação individual em um microprocessador RISC. Esta simultaneidade na execução das instruções é possível através da utilização de várias unidades funcionais arranjadas em um enorme *pipeline*, de maneira que a arquitetura superescalar do microprocessador seja totalmente aproveitada.

Existem *softwares* que compactam as operações de um programa em instruções LIW. Quanto maior a compactação, maior o desempenho [ROS03]. As instruções que utilizam codificação compactada são de tamanhos variáveis, sendo este tamanho dependente do número de FUs (*Functional Units*) que receberão uma operação [CON96]. Este tipo de codificação tem uma utilização maior de memória e permite uma banda de memória mais efetiva que a codificação descompactada. Por outro lado, isso torna os microprocessadores LIW incompatíveis no *software* com os GPPs (Processadores de Propósito Geral, do inglês *General Purpose Processors*).

As instruções de desvio condicional de microprocessadores tradicionais, geralmente especificam um endereço de destino e uma condição. Se essa condição for satisfeita, seu contador de programa é atualizado com o endereço contido na instrução, de forma que, no decorrer da execução do programa será o próximo endereço de memória acessado. Os conjuntos de instruções LIW, todavia, freqüentemente contém um recurso para especificar o número de condições e endereços alvo, de forma que o valor do contador de programa é selecionado entre os endereços alvo dependendo de uma combinação de condições [MOO95].

As instruções LIW são organizadas em pacotes de tamanho e número de instruções fixos. Além disso, estes pacotes contêm *bits* de controle, que indicam quais de suas instruções podem ser executadas paralelamente. O escalonamento das instruções bem como a organização dos pacotes fica a cargo do compilador, ao contrário das arquiteturas superescalares dos microprocessadores CISC e RISC, onde boa parte da lógica é usada para detectar o paralelismo implícito do programa em execução. Logo, o número de instruções em microprocessadores LIW é reduzido em relação às outras arquiteturas.

Os microprocessadores que utilizam o modelo Intel-64 (IA-64) utilizam esta idéia de encapsulamento de instruções em pacotes. No IA-64, cada pacote possui 128 *bits* e contém três instruções de tamanho fixo (40 *bits*). Os 8 *bits* restantes contêm informações referentes a quais instruções podem ser executadas em paralelo [TAN99].

Existe nesta arquitetura uma grande dependência do microprocessador para com o compilador, assim como na arquitetura RISC, pois a responsabilidade pela organização das instruções encapsuladas nos pacotes fica totalmente a critério do compilador. Desta forma, a concorrência na execução das instruções possibilitada pela arquitetura LIW somente pode ser aproveitada se instruções que não tenham dependências forem agrupadas dentro de um mesmo pacote.

A grande vantagem das arquiteturas LIW é a que elas não precisam de um *hardware* para detecção do paralelismo, já que ele está inerentemente especificado nas próprias instruções [ROS03]. Por outro lado, em trechos de código com pouco paralelismo haverá um desperdício que causará um aumento do número de instruções.

2.2 Microprocessadores

Um microprocessador é composto por um conjunto de circuitos integrados que tem por finalidade manipular e transformar dados através de um conjunto de instruções pré-definido.

A função da UCP (Unidade Central de Processamento), conforme [MON96] consiste basicamente nas seguintes tarefas:

- buscar uma instrução na memória (operação de leitura), uma de cada vez;
- interpretar que operação a instrução está explicitando (pode ser uma soma de dois números, uma multiplicação, uma operação de entrada ou saída de dados, ou ainda uma movimentação de um dado de uma célula para outra);
- buscar os dados onde estiverem armazenados, para trazê-los até a UCP;
- executar efetivamente a operação com o(s) dado(s), guardar os resultados (se houver algum) no local definido na instrução;
- reiniciar o processo apanhando a nova instrução.

Todas estas tarefas se repetem indefinidamente até que seja encontrada uma instrução de parada ou que o sistema seja desligado (ou por um erro ou por intervenção direta do usuário), formando o que se denomina *ciclo de instrução*.

A Figura 2.1 [TAN99], representa um diagrama detalhado de uma UCP genérica que utiliza a arquitetura CISC. O diagrama tem duas partes: a unidade de execução (à esquerda) e a unidade de controle (à direita), que serão discutidas com detalhes nas próximas seções.

2.2.1 Unidade Operacional

A unidade operacional é a parte da UCP que contém a ULA (Unidade Lógica e Aritmética), suas entradas e suas saídas [TAN99]. A ULA é o dispositivo que realiza operações aritméticas como a adição e a subtração, ou operações lógicas como AND e OR [PAT00]. As operações da ULA são, geralmente muito simples. Funções mais complexas, exigidas pelas instruções da máquina, são realizadas pela ativação seqüencial das várias operações básicas disponíveis.

A unidade operacional contém ainda um conjunto de registradores, que na Figura 2.1 estão assinalados com nomes simbólicos tais como PC, SP e MDR. Grande parte destes registradores é utilizada para armazenamento temporário de operandos ou resultados de operações. As saídas destes registradores estão conectadas ao barramento B, que alimenta uma das entradas da ULA. A outra entrada da ULA está conectada ao barramento A, o qual está ligado à saída do registrador H. A saída da ULA possui uma conexão com o circuito *Deslocador*, que se conecta ao barramento C e, que por sua vez está conectado com as entradas dos registradores, podendo desta forma escrever dados em suas células. A linha demarcada com o número "6" informa que

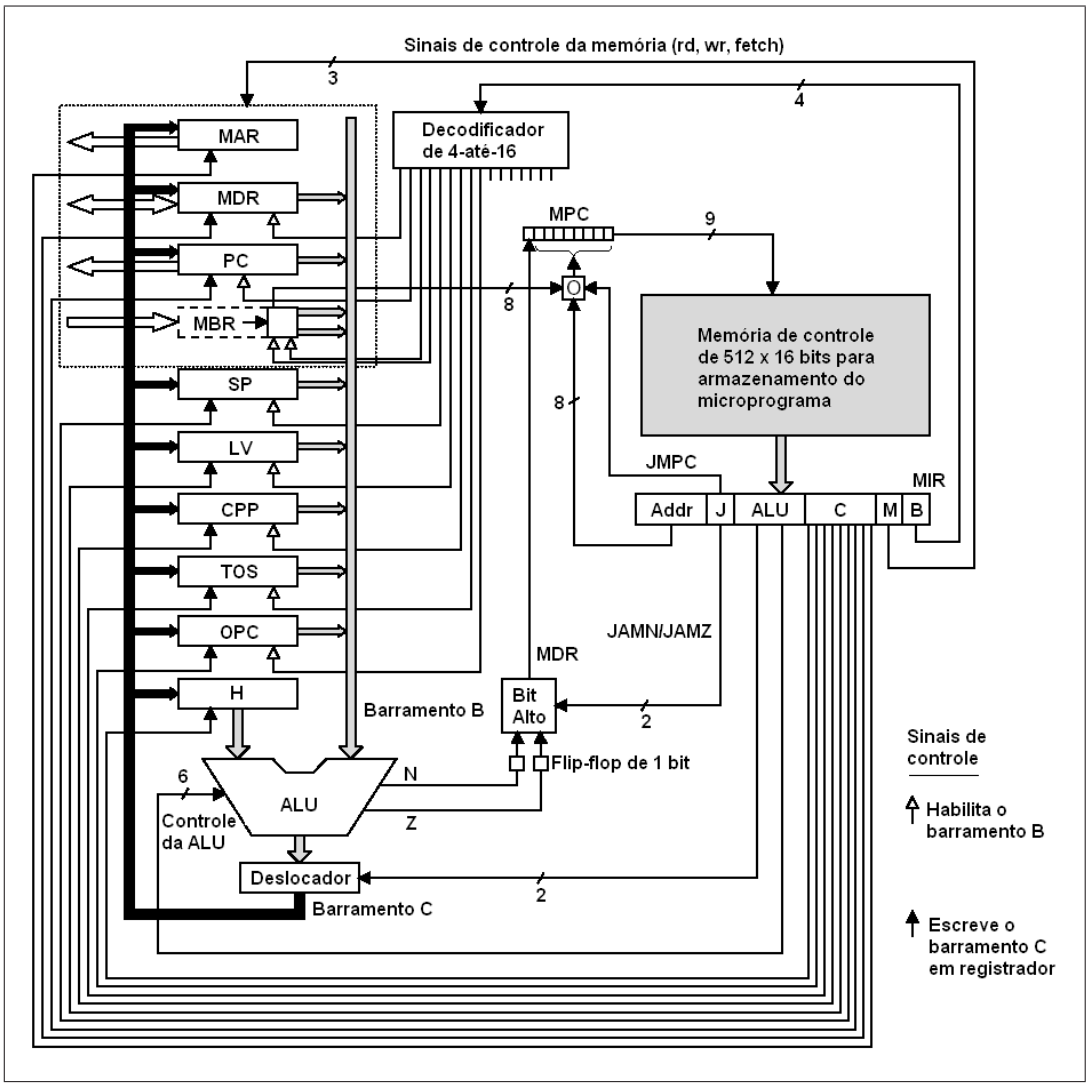

Figura 2.1: Diagrama genérico de uma UCP CISC

existem seis sinais de controle da ULA, que informam o que deve ser feito com o(s) dado(s) que forem recebidos.

Abaixo são listados os principais registradores da unidade operacional presentes nas arquiteturas CISC genéricas conjuntamente com suas respectivas funções:

- *Shifter Register* ou Deslocador: como o próprio nome já diz, é um registrador utilizado para realizar deslocamentos nos resultados gerados pela ULA, tendo grande importância em operações como a multiplicação, por exemplo.
- PC (*Program Counter* ou Contador de Programa): é um registrador utilizado para apontar o endereço de memória da próxima instrução a ser executada.
- SP (*Stack Pointer* ou Ponteiro de Pilha): é um registrador que trabalha dentro do segmento de pilha, usado para determinar o *offset* do topo da pilha.

- Acumulador ou Registrador de Trabalho: é um registrador que tem por função armazenar um operando e/ou um resultado fornecido pela ULA, representado na Figura 2.1 pelo registrador denominado H.

- MAR (*Memory Address Register* ou Registrador de Endereço da Memória): Contém o endereço da posição de memória a ser lida/escrita.

- MDR (*Memory Data Register* ou Registrador de Dados da Memória): Contém o dado a ser lido/escrito na memória.

2.2.2 Unidade de Controle

A unidade de controle, além de possuir a lógica necessária para realizar a movimentação de dados e instruções de/para a UCP, através dos sinais de controle que emite em instantes de tempo programados, controla a ação da ULA [MON96]. Todos estes sinais são gerados por um oscilador e possuem duração fixa e igual e sincronizam todas as operações que são realizadas entre os componentes da UCP. Além disso, também cabe a unidade de controle a interpretação e decodificação das instruções buscadas na memória.

O maior e mais importante bloco da unidade de controle de uma máquina CISC é a memória de controle, onde está armazenado o microprograma, que armazena o conjunto de microinstruções que o microprocessador pode executar [TAN99]. Com base nessa memória, o microprocessador consegue determinar qual é a seqüência de passos que deve ser realizada para executar uma instrução.

Na Figura 2.1 as setas que interligam todos os blocos do microprocessador, indicam os sinais de controle. As setas abertas, indicam sinais de controle que habilitam a saída dos registradores no barramento B; enquanto que as setas fechadas, indicam sinais de controle para escrita do conteúdo do barramento C nos registradores.

Conforme a Figura 2.1, para o controle da unidade operacional são necessários 29 sinais, que podem ser divididos em 5 grupos funcionais, descritos a seguir:

- 9 sinais para controlar a escrita dos dados do barramento C nos registradores;

- 9 sinais para controlar a liberação dos dados dos registradores no barramento B para uma das entradas da ULA;

- 8 sinais para controlar a ULA e a função de deslocamento;

- 2 sinais (não exibidos) para indicar leitura/escrita na memória via MAR/MDR;

- 1 sinal (não exibido) para indicar o acesso à memória através via PC/MBR.

Estes 29 sinais de controle especificam as operações para um ciclo da unidade operacional. Este ciclo consiste em liberar os valores dos registradores para o barramento B, propagá-los através da ULA e do deslocador, liberá-los no barramento C e finalmente escrever o resultado no(s) registrador(es) apropriado(s).

2.2.3 Registradores

Um registrador é um dispositivo de armazenamento capaz de guardar dados binários; é um conjunto de um ou mais *flip-flops*. Um registrador de *n bits* é um registrador construído com *n flip-flops* [HUA97].

A generalização do conceito de máquinas com registradores dedicados permite que todos eles possam ser usados com qualquer finalidade, daí o nome máquinas com *registradores de propósito geral*. Este estilo de projeto de conjunto de conjunto de instruções pode ainda ser dividido entre aqueles que permitem que um operando esteja na memória, como máquinas baseadas no acumulador, chamadas de arquitetura *registrador-memória*, e naqueles que obrigam que os operandos estejam sempre em registradores, chamados de máquinas *load-store* ou máquinas *registrador-registrador*. A Figura 2.2 [PAT00] lista o número de GPRs (Registradores de Propósito Geral, do inglês *General Purpose Registers*) de algumas máquinas mais populares desenvolvidas ao longo do tempo.

Máquina	Número de Registradores de Propósito Geral	Estilo da Arquitetura	Ano
EDSAC	1	acumulador	1949
IBM 701	1	acumulador	1953
CDC 6600	8	load-store	1963
IBM 360	16	registrador-memória	1964
DEC PDP-8	1	acumulador	1965
DEC PDP-11	8	registrador-memória	1970
Intel 8008	1	acumulador	1972
Motorola 68000	16	registrador-memória	1974
DEC VAC	16	registrador-memória; memória-memória	1977
Intel 8086	1	acumulador estendido	1978
Intel 80386	8	registrador-memória	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992

Figura 2.2: Número de GPRs de algumas máquinas desenvolvidas ao longo do tempo

Pode-se observar que as máquinas com arquitetura baseada em registrador possuem um grande número de GPRs quando comparadas às máquinas convencionais.

2.2.4 Acesso à Memória

Em um sistema de computação, a destinação final do conteúdo de qualquer tipo de memória é o microprocessador (UCP). Isto é, o objetivo final de cada uma das memórias (ou do subsistema de memória) é armazenar informações destinadas a serem, em algum momento, utilizadas pelo microprocessador [MON96].

Em um microprocessador CISC convencional, a interface com a memória é realizada utilizando o fluxo de dados dos registradores MAR e MDR (indicado na Figura 2.1 pelas setas transparentes).

A combinação MAR/MDR é utilizada para ler e escrever dados na memória. Então, quando é necessário acessar a memória, o endereço da célula que deve ser lido/escrito é determi-

nado através do valor corrente do registrador MAR. Se for uma operação de leitura, o conteúdo da célula de memória indicada no registrador MAR é copiado para o registrador MDR. Se for uma operação de escrita, o conteúdo do registrador MDR é copiado para a célula de memória indicada no registrador MAR.

No esquema da Figura 2.1 os registradores PC e MBR também possuem uma interface com a memória e, neste exemplo, são usados em combinação para realizar a leitura de instruções de um programa em execução em cadeias de *bytes*.

Como os acessos à memória principal do microcomputador são os eventos que despendem maior tempo durante a execução de uma aplicação, os microprocessadores atuais geralmente possuem uma memória interna, chamada de memória *cache*, que tem por objetivo reduzir esta quantidade de acessos. Dessa forma, como a memória *cache* está dentro da pastilha do microprocessador, ela pode ser acessada na mesma freqüência de *clock* em que ele opera.

2.2.5 Acesso a Dispositivos de E/S

Um microprocessador pode ser destinado não somente para manipular dados, mas também para controlar as operações de algum processo externo. Num tal caso, o sistema pode ter que aceitar novos dados em ocasiões imprevisíveis durante o próprio tempo em que alguma computação ou manipulação estiver em andamento, e o sistema também pode ser solicitado a fornecer sinais de controle da saída em tempos igualmente imprevisíveis [TAU84].

Para que o microprocessador possa se comunicar com os demais periféricos do microcomputador, ele deve ser capaz de endereçar tais dispositivos. Existem dois métodos para endereçar dispositivos: E/S (Entrada/Saída) mapeada na memória e comandos especiais de E/S [PAT00]. No mapeamento do dispositivo em memória, o acesso é realizado através de faixas de endereços de memória previamente definidos, de forma que, qualquer dado enviado para uma destas faixas de endereço é automaticamente ignorado pelo sistema de memória e enviado diretamente para a controladora de dispositivos. Na técnica de utilização de comandos especiais, a solução é agregar ao conjunto de instruções do microprocessador instruções que tratem especificamente de E/S, que devem especificar tanto o dispositivo com o qual se deseja realizar a comunicação quanto o comando que deve ser executado.

Entretanto, quando um periférico precisa se comunicar com o microprocessador, ou para enviar algum dado ou para informar que já está preparado para receber mais comandos, normalmente são utilizadas três técnicas: *polling*, DMA (*Direct Memory Access* ou Acesso Direto à Memória) e *interrupções*. A técnica de *polling* gera muita sobrecarga, pois consiste basicamente em verificações que o microprocessador realiza em determinados instantes de tempo para verificar se algum periférico precisa de sua atenção. Já a técnica de DMA consiste em permitir que os dados possam ser enviados de um dispositivo (tal como um disco rígido, por exemplo) diretamente para a memória sem a intervenção do microprocessador, que fica livre para executar as instruções durante a transferência. Contudo, a técnica de interrupções é mais difundida, e será explicada na Seção 2.2.6.

2.2.6 Interrupções e Exceções

Interrupções e exceções são eventos que acontecem de forma inesperada e interrompem a execução seqüencial das instruções de uma tarefa. As interrupções podem ser geradas ou por *hardware* ou por *software*. As interrupções, de uma forma geral, estão associadas a algum circuito externo ao microprocessador que precisa de “atenção”.

Mas apenas informar ao microprocessador a ocorrência de uma interrupção não é o suficiente, é necessário indicar qual circuito foi o responsável pela sua geração. Para que isto seja possível, cada interrupção está associada a um identificador e, este identificador referencia um determinado circuito, possibilitando que se determine quem causou o pedido de interrupção. Isto é necessário, devido ao fato que, os microprocessadores disponíveis comercialmente têm apenas uma linha de interrupção, necessitando de um outro circuito para auxiliar no gerenciamento destas interrupções. O responsável por este gerenciamento é o controlador de interrupções, que está geralmente presente no *chipset*[6] da placa-mãe.

Os números de interrupção mais baixos estão associados a circuitos de maior importância para o funcionamento do computador, de forma que existe um nível de privilégio que deve ser obedecido pelo controlador de interrupções. Assim, quando houverem pedidos de interrupção simultâneos, o microprocessador irá atender primeiro ao dispositivo de maior prioridade.

2.2.7 Arquiteturas de Endereçamento

A arquitetura de endereçamento de um microprocessador serve para indicar como os dados pertencentes ao seu pacote de instrução devem ser interpretados. As principais arquiteturas de endereçamento utilizadas são as de 4 endereços, 3 endereços, 2 endereços e 1 endereço. A arquitetura de 4 endereços não é mais utilizada, mas consistia basicamente de uma instrução com um campo para o código de instrução, outros dois para o primeiro e segundo operandos fonte, outro para o endereço onde deveria ser armazenado o resultado da operação e um campo para indicar o endereço da próxima instrução. Entretanto, o principal problema da arquitetura de 4 endereços é que a instrução acaba se tornando muito grande e ocupando um grande espaço na memória. Além disso, grande parte das instruções de um programa acabam não ocupando todos os campos do pacote de instrução disponíveis nesta arquitetura, o que acaba gerando desperdício espaço.

Nas arquiteturas de 3 endereços o formato de instrução é o mesmo da arquitetura de 4 endereços, exceto pela remoção do campo que especificava o endereço da próxima instrução. Para suprir a falta desta informação, este tipo de arquitetura possui um registrador específico, denominado Contador de Programa, que tem por função apontar o endereço seguinte da instrução que está sendo executada.

A instrução da arquitetura de 2 endereços possui apenas os campos destinados para o primeiro e segundo operandos fonte. O endereço da próxima instrução é definido pelo Contador de Programa. Além disso, deve-se atribuir a função de receptor do resultado da operação para um dos endereços passados como parâmetro dentro da própria instrução, pois o campo reservado para a especificação do destino da operação não existe. Isso introduz uma séria restrição a esta arquitetura de endereçamento: um dos dois operandos necessariamente será alterado, dessa maneira, devem também ser adicionadas instruções para movimentação de dados, que permitam copiar valores entre as posições de memória.

A arquitetura de 1 endereço trabalha com instruções pequenas, possuindo somente o código da operação que se deseja executar e o endereço do primeiro operando fonte. Além de ter que usar um registrador específico para armazenar o endereço da próxima instrução, esta arquitetura precisa especificar um outro registrador, geralmente o Acumulador, para ser assumido implicitamente como o outro operando e também como destino da operação.

[6]Conjunto de CIs que compõe a placa-mãe de um microcomputador

2.2.8 Modos de Endereçamento

A existência de vários métodos para localizar um dado que está sendo referenciado em uma instrução se prende à necessidade de dotar os sistemas de computação da necessária flexibilidade no modo de atender aos diferentes requisitos de programas [MON96]. A adoção de diferentes formas de se endereçar uma informação permite adequar uma arquitetura a diversas de aplicações com características distintas.

As seções a seguir descrevem alguns dos modos de endereçamento utilizados pelas arquiteturas disponíveis atualmente.

2.2.9 Modo Imediato

No modo imediato, o campo da instrução destinado ao operando contém o próprio dado a ser manipulado, ou seja, não existe a necessidade de acessar a memória para buscar seu conteúdo. Este modo geralmente é utilizado para realizar a definição de constantes ou para atribuir valores iniciais às variáveis de um programa.

A vantagem da utilização deste modo está no fato de que não existe qualquer referência à memória, além da busca da própria instrução. Desta maneira, poupa-se o ciclo de memória ou de *cache* no ciclo da instrução. Por outro lado, o tamanho do campo de endereçamento é fica restrito ao campo de endereço, que, na maioria dos casos é pequeno quando comparado ao tamanho da palavra.

2.2.10 Modo Direto

No modo direto, o campo da instrução reservado para o operando contém o endereço efetivo da posição de memória onde ele se encontra armazenado. Isso permite que o tamanho da palavra utilizada possa ser maior em termos de *bits*, entretanto, gera uma limitação na quantidade de memória que pode ser endereçada dentro da instrução. Além disso, o modo direto é mais lento em relação ao modo imediato, pois requer uma referência à memória para buscar o operando.

2.2.11 Modo Indireto

O modo indireto pode ser comparado analogamente com o uso de *ponteiros* em linguagens de programação, pois o campo destinado ao endereço do operando contém um endereço que por sua vez armazena outro endereço, no qual está localizado o endereço real do dado a ser manipulado.

As vantagens da utilização deste modo de endereçamento estão na possibilidade de implementação de estruturas de organização de dados mais complexas e na eliminação do problema da limitação da quantidade de memória que pode ser acessada. Já a principal desvantagem é a necessidade de uma quantidade maior de acessos à memória para completar o ciclo de execução de uma instrução.

2.2.12 Endereçamento por Registrador

O endereçamento por registrador pode ser direto ou indireto. Quando empregado o modo direto, o campo reservado para o operando se refere ao endereço de um registrador. Quando empregado o modo indireto, este campo se refere ao endereço do registrador que indica o endereço

de memória ou o endereço de outro registrador, onde se encontra o dado. A diferença entre os endereçamentos direto por registrador e indireto por registrador com relação aos modos de endereçamento direto e indireto, descritos anteriormente, está no fato de que ao invés de células de memória, são endereçados registradores.

Existem duas vantagens deste modo em relação aos anteriores. Primeiro, como os registradores geralmente são poucos, bastam poucos *bits* para endereçá-los, o que torna as instruções menores. Segundo, o acesso ao dado é mais rápido, uma vez que os registradores têm um tempo de acesso menor em relação à memória.

2.2.13 Conjunto de Instruções

Uma instrução é uma informação codificada, incluindo um identificador, que informa o tipo de operação a executar e operandos (registradores ou memória) a serem manipulados na operação [MEN96]. A agregação dessas instruções forma o que se chama de conjunto de instruções de um microprocessador. O conjunto de instruções realiza a interface entre os programas escritos pelo usuário e o *hardware* do sistema.

Como visto anteriormente, quanto maior e mais complexo é o conjunto de instruções de um microprocessador, maior é o tempo gasto na decodificação dos códigos de operação das instruções, seu microprograma (para as arquiteturas que o utilizam) e seu espaço físico na pastilha do microprocessador. Por outro lado, uma máquina com um conjunto pequeno de instruções exige a utilização de um compilador mais poderoso e resulta em programas maiores em termos de instruções.

De qualquer forma, o formato e tamanho do conjunto de instruções de um microprocessador estão intimamente ligados ao tipo de arquitetura interna usado na sua construção e no nível de flexibilidade que se deseja transferir para o programador.

Capítulo 3

Métodos de Prototipação

Atualmente, um sistema eletrônico pode ser implementado mediante a definição de todos os elementos do circuito integrado (*chip*), pode-se usar *chips* parcialmente pré-fabricados (os chamados pré-difundidos) ou pode usar componentes programáveis externamente [REI02]. A forma de implementação utilizada deve levar em consideração aspectos como a finalidade do circuito, o tempo e o custo de fabricação, a necessidade de desempenho, entre outros.

Para circuitos produzidos em grande escala, como microprocessadores e microcontroladores, geralmente tem-se a necessidade de realizar o projeto completo do *chip*, especificando a geometria de seus diversos componentes, pois se consegue estabelecer padrões de integração, desempenho e confiabilidade superiores com relação aos outros métodos. Por outro lado, o custo de fabricação do circuito é mais elevado, mas, dependendo da demanda de mercado, plenamente justificável.

Nos *chips* pré-difundidos, onde os transistores já se encontram pré-fabricados, e nos componentes programáveis o tempo de projeto é radicalmente diminuído. Nos componentes programáveis especialmente, onde a parte física do circuito já se encontra inteiramente projetada, resta ao projetista realizar a síntese lógica. Dessa forma, análises de funcionamento, desempenho e integração do circuito podem ser efetuadas no decorrer do processo de implementação. Além disso, como estes métodos propiciam uma rápida prototipação, são adequados no desenvolvimento de projetos comerciais que exigem velocidade de implantação no mercado.

Uma combinação destas metodologias também pode ser utilizada para fins de ganho de desempenho como na arquitetura GARP [HAU97], que consiste em um microprocessador que utiliza um co-processador reconfigurável com seus demais componentes fundidos em um *chip*, ou ser puramente implementado em uma HDL para servir de protótipo de testes antes da implementação física definitiva como na arquitetura Chameleon [RAM96]. Existem ainda os microprocessadores RISP (Reconfigurable Instruction Set Processors) que possuem um conjunto de instruções reconfigurável, podendo adaptar o seu conjunto de instruções para a aplicação que está sendo executada [BAR00][ATH93].

3.1 Hardware Programável

Muitas aplicações emergentes em telecomunicações e multimídia necessitam que suas funcionalidades permaneçam flexíveis mesmo depois do sistema ter sido manufaturado. Tal flexibilidade é fundamental, uma vez que requisitos dos usuários, características dos sistemas, padrões e protocolos podem mudar durante a vida do produto [MES02].

O uso de *hardware* programável surge como uma alternativa para modificações de funcionalidade no *hardware* de sistemas digitais, permitindo que mudanças estruturais possam ser realizadas mesmo após o produto ter saído da fábrica, rompendo desta maneira as limitações impostas pelas atualizações de *software* que ficam restritas somente à parte programável dos sistemas.

Um dispositivo programável pode freqüentemente computar, em um único ciclo, uma computação que um GPP ou DSP (*Digital Signal Processing*) levaria centenas de ciclos [DEH00]. Nestes dispositivos, as operações são implementadas de forma espacial, explorando-se o paralelismo inerente das aplicações alvo [CAP01]. Ou seja, diferentes aplicações podem estar sendo implementadas pelo dispositivo programável em diferentes instantes de tempo.

Contando com o avanço da tecnologia VLSI (*Very Large Scale Integration*) tornou-se possível desenvolver circuitos digitais com baixo custo, elevado número de transistores e alta freqüência de funcionamento. Os dispositivos mais beneficiados com esta tecnologia foram os componentes programáveis do tipo FPGA. Estes dispositivos têm auxiliado na prototipação de circuitos digitais, pois são facilmente reprogramados através da utilização de ferramentas de CAD [SHE95]. Além disso, o uso de dispositivos programáveis possibilita a definição qualquer sistema digital a partir de um nível de abstração mais elevado, dispensando os métodos tradicionais de prototipação de *hardware* (processos para criação dos transistores e definição de suas interconexões sobre a pastilha de silício).

Atualmente, FPGAs contém lógica suficiente para implementar pequenas máquinas de computação personalizadas. Pequenas empresas e universidades especialmente, podem desenvolver *hardware* baseado em FPGAs, como microprocessadores ou microntroladores, para atender a especificações exatas, de uma forma melhor que construir circuitos por meio de métodos tradicionais de prototipação.[MEI95]. Como os dispositivos FPGAs permitem ser reconfigurados, é possível que um circuito digital seja gradativamente construído e adaptado até alcançar um grau de maturidade que permita sua prototipação definitiva em um *chip*.

3.1.1 Dispositivos FPGA

FPGAs são circuitos integrados derivados dos PLDs. Os dispositivos PLDs são programados através de *softwares* especiais disponibilizados por seus fabricantes. A programação é feita através de campos elétricos induzidos no dispositivo.

No início da década de 80 PLDs eram utilizados na implementação de diversos dispositivos lógicos. Atualmente, PLDs integram em um único dispositivo grande quantidade de blocos lógicos capazes de implementar muitas funções lógicas sendo em muitos casos preferidos aos ASICs. Um ASIC é um dispositivo dedicado, ou seja, é destinado a atender uma aplicação específica e após ser manufaturado não pode mais ser alterado.

Contudo, a utilização de ASICs é restrita em relação ao seu custo. Há necessidade de produzir-se um grande volume de determinado ASIC para amortizar o custo de produção. FPGAs oferecem um compromisso entre custo de produção e desempenho que justifica sua utilização para um leque variado de aplicações. Além disso, hoje em dia é possível encontrar no mercado FPGAs com densidade e desempenho muito próximos ao de um dispositivo não programável.

A computação com FPGAs é chamada *computação configurável* por que ela é definida através de *bits* de configuração no dispositivo, que dizem como suas portas e interconexões devem se comportar [DEH00]. Dessa forma, um dispositivo FPGA pode ser programado após

sua fabricação para desempenhar virtualmente qualquer tarefa que se não exceda seus recursos operacionais.

A arquitetura de um FPGA, ilustrada na Figura 3.1, consiste de uma matriz de blocos lógicos que podem ser programavelmente interconectados para realizarem diferentes funções. FPGAs são configurados através de comutadores eletricamente programáveis de forma parecida com os PLDs, contudo, FPGAs podem alcançar níveis de integração muito maiores que os PLDs devido a sua arquitetura de roteamento e implementação lógica serem mais complexas [ROS93]. As duas categorias básicas de FPGAs no mercado hoje em dia são a baseada em SRAM (*Static Random Access Memory*) e a anti-fusível [BRO96].

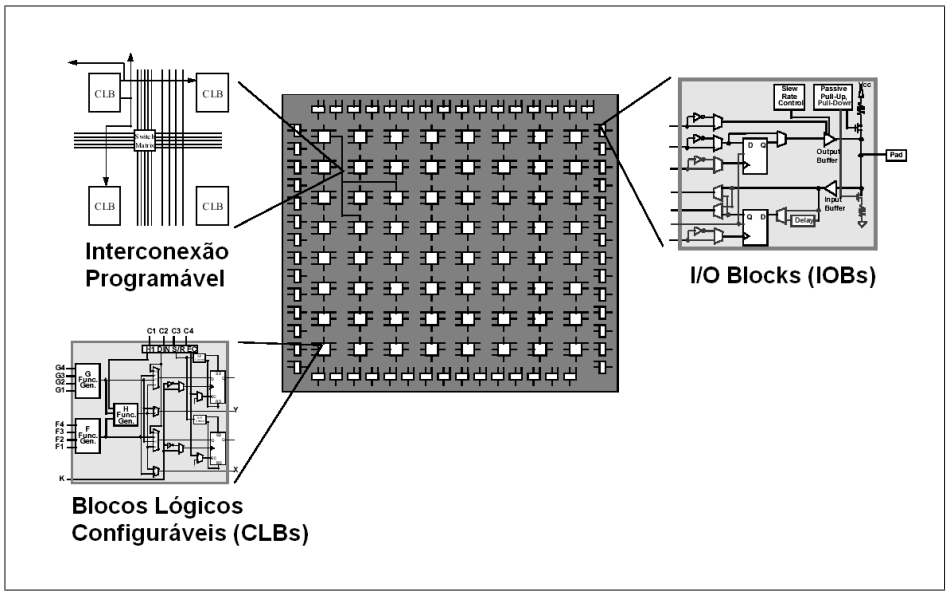

Figura 3.1: Arquitetura de um FPGA baseado em LUTs

Conforme Rose [ROS93], o bloco lógico de um FPGA pode ser simples como um transistor ou complexo como um microprocessador, sendo tipicamente capaz de implementar diversas funções lógicas seqüenciais e combinacionais diferentes. Esta característica se refere à granularidade do dispositivo. Entende-se por grão a menor unidade configurável da qual é composto o FPGA. FPGAs atuais empregam blocos lógicos baseados em tabelas-verdade (*Look-Up Tables*, ou simplesmente LUTs), mas já verifica-se a tendência de utilização de pequenos microprocessadores [SAS01].

Quando o tamanho do grão do bloco lógico aumenta, espera-se que o número de blocos necessários para implementar um projeto diminua. Por outro lado, um bloco lógico mais poderoso (de grão maior) exige mais interconexões configuráveis para implementá-lo, e por conseqüência ocupa mais área. Este situação antagônica sugere a existência de um ponto ótimo para a granularidade de bloco lógico, no qual a área de FPGA dedicada à implementação da lógica é minimizada [TOR01].

A funcionalidade desempenhada pelo FPGA é definida através de um arquivo de configuração chamado de *bitstream*. O *bitstream* é composto por um fluxo de palavras que segue o protocolo de configuração determinado pelo fabricante do dispositivo [XIL03] .

Diversas pesquisas demonstram os ganhos de desempenho através da implementação de sistemas baseados em FPGAs, nas mais diversas áreas de conhecimento. Por exemplo:

- Criptografia com o algoritmo RSA: A máquina PAM (*programmable-active-memory*) construída no INRIA[7] em conjunto com a *Digital Equipment* conseguiu uma taxa de descriptografia do algoritmo RSA maior do que qualquer outra máquina já construída (600 *Kbits* por segundo, com chaves de 512 *bits*).

- Seqüenciamento de DNA. A máquina Splash2 [ARN92], do *Supercomputer Research Center* consegue realizar rotinas de seqüenciamento de DNA mais que duas ordens de grandeza mais rápido que MPPs (*Massive Parallel Processors*) e supercomputadores (Cray-2) e 3 ordens de grandeza mais rápido que estações de trabalho (*Sun Sparcstation*).

- Processamento de sinal: Filtros implementados em FPGAs Xilinx têm desempenho maior que DSPs (*Digital Signal Processors*) e outros processadores em uma ordem de grandeza [KNA03].

3.1.2 Classificação dos FPGAs

A Figura 3.2 mostra os quatro principais tipos arquitetura interna presentes em dispositivos FPGA: matriz simétrica, *row-based*, *sea-of-gates* e PLD hierárquico.

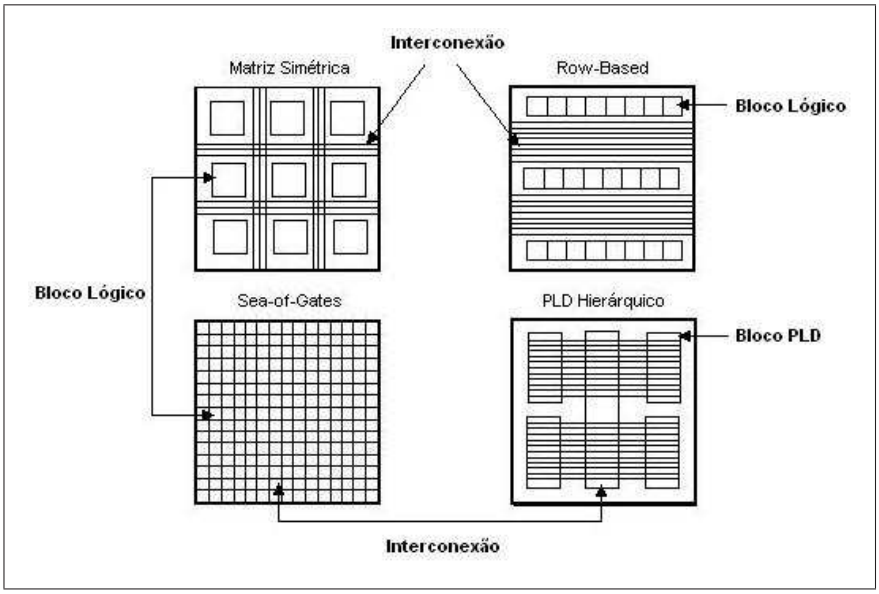

Figura 3.2: Arquiteturas de dispositivos FPGA

Na arquitetura de matriz simétrica, os blocos lógicos são arranjados em uma rede onde existem interconexões horizontais e verticais entre os blocos lógicos, o que possibilita grande flexibilidade de roteamento.

No grupo que utiliza a estrutura *row-based*, os blocos lógicos são organizados em linhas e são interconectados pelos canais de roteamento apenas no sentido horizontal, não havendo interconexões verticais (seguindo a metodologia utilizada em ASICs).

[7] *Institut National de Recherche en Informatique et en Automatique* , de Paris

Os dispositivos com arquitetura *sea-of gates* são completamente formados por blocos lógicos, compostos por transistores ou portas lógicas simples. Apesar de ter uma grande densidade de blocos lógicos, não existe uma região dedicada para o roteamento, fazendo com que vários destes blocos sejam utilizados para este fim e, conseqüentemente, inutilizando-os.

Já os circuitos com a estrutura de PLD hierárquico são constituídos por uma matriz de blocos lógicos interconectados através de um recurso de roteamento denominado PIA (*Programmable Interconect Array*), permitido que os blocos se conectem entre si.

3.2 Linguagens de Descrição de Hardware

Linguagens de Descrição de *Hardware* (HDLs[8]) são linguagens de programação que foram desenvolvidas e otimizadas para projeto e modelagem de circuitos digitais. Para tanto, HDLs possuem características de desenvolvimento que permitem a definição do comportamento de componentes eletrônicos através de comandos com alto nível de abstração da mesma forma que em linguagens de programação tradicionais como C, C++ ou Pascal.

Linguagens como VHDL (o "V" significa VHSIC[9]) e Verilog (as duas HDLs predominantes atualmente) possibilitam uma descrição precisa de todos os aspectos elétricos envolvidos no comportamento de um circuito (tempo de subida e descida de um sinal, atraso de portas lógicas e operações funcionais). Além destas linguagens, existem algumas outras não tão difundidas tais como: SystemC, SpecC, HandelC, Esterel, etc [CAL03].

Segundo Ordonez [ORD03], a importância da utilização de linguagens de descrição de *hardware* manifesta-se em diversos aspectos do projeto:

- Documentação do sistema: a própria descrição do sistema já é uma forma de documentação para os projetistas.
- Simulação em diversos níveis: desde a sua especificação funcional e temporal o circuito pode ser simulado, de forma a validar sua funcionalidade.
- Simplifica a migração tecnológica: o sistema pode ser facilmente re-sintetizado em outras tecnologias, desde que se disponha das ferramentas adequadas.
- Reutilização de recursos: a construção de bibliotecas ou módulos, permite reutilizar parte de projetos já realizados.

A maior vantagem que o uso de uma HDL proporciona é a possibilidade de projetar circuitos eletrônicos da mesma forma que se projeta *software*. Isso evita a obrigatoriedade do desenvolvedor de sistemas digitais em lidar diretamente com equações boolenas e sua posterior conversão em blocos básicos de construção de circuitos digitais como portas lógicas e *flip-flops*. Além disso, o uso de tais blocos de construção se torna impraticável quando existe a necessidade de construção de um sistema complexo como um microprocessador, por exemplo.

[8] do inglês *Hardware Description Languages*

[9] *Very High Speed Integrated Circuit*

3.2.1 VHDL

VHDL é uma linguagem para descrição de sistemas eletrônicos digitais. Ela surgiu com o programa VHSIC do Departamento de Defesa dos Estados Unidos, iniciado em 1980. No andamento deste programa, ela se encaixou totalmente no que era necessário para uma linguagem que se propunha descrever a estrutura e a funcionalidade de circuitos digitais. Dessa forma, a VHSIC *Hardware Description Language* (VHDL) foi desenvolvida e subseqüentemente foi padronizada pelo IEEE (*Institute of Electrical and Electronic Engineers*) dos Estados Unidos [VHD90].

Em 1992 foram propostas várias alterações para a padronização do IEEE. Em 1993 foi lançada a versão revisada, mais flexível e com novos recursos. Esta norma revisada, chamada VHDL-93, é a hoje a mais amplamente utilizada.

Uma das características dos dispositivos de *hardware* é que sua funcionalidade pode ser definida pela parte mais independente do meio com o qual eles operam, de forma que, componentes de várias fontes podem ser interconectados para formar novos componentes de maior complexidade [SHA86]. VHDL reflete esta característica na sua organização geral, uma vez que pode-se descrever componentes isolados, chamados *entidades*, que podem por sua vez ser combinados com descrições de outros componentes para formar descrições mais complexas.

Uma descrição VHDL consiste basicamente de uma interface e uma arquitetura. A entidade define quais são os canais de comunicação do componente com o mundo exterior, ou seja, suas portas de E/S. A arquitetura específica o comportamento e organização interna do componente. A arquitetura de um componente pode ser especificada usando a descrição procedural, por fluxo de dados ou estrutural, ou ainda uma combinação destes três estilos.

A descrição comportamental utiliza uma notação semelhante com as linguagens de programação tradicionais. Em outras palavras, utilizam algoritmos para manipular os dados de entrada do componente e desempenhar sua funcionalidade. A Descrição 1 apresenta o código utilizando a notação comportamental para representar um circuito comparador de 1 *bit*.

Descrição 1 Comparador de 1 *bit* - Descrição comportamental

```
01: entity comparador is
02: port (e1, e2: in bit;
03:         s: out bit);
04: end comparador;
05:
06: architecture comportamental of comparador is
07: begin
08:     process (e1, e2)
09:     begin
10:         if e2 < e1 then
11:             s <= '1';
12:         else
13:             s <= '0';
14:         end if;
15:     end process;
16: end comportamental;
```

Na descrição por fluxo de dados deve-se indicar como os dados fluem das entradas para as saídas do componente através de conexões, chamadas em VHDL de *sinais*. A Descrição 2

mostra o código para criação de um comparador de 1 *bit* utilizando a representação por fluxo de dados.

Descrição 2 Comparador de 1 *bit* - Descrição de fluxo de dados

```
01: architecture fluxo of comparador is
02: begin
03:     s <= '1' when e2 < e1
04:     else '0';
05: end fluxo;
```

Na descrição estrutural devem ser especificados os componentes que devem ser usados e como eles devem ser conectados para atingir os resultados esperados, ou seja, define a estrutura de um sistema. A descrição estrutural é mais fácil de sintetizar que as demais porque se refere a componentes físicos concretos; em compensação sua especificação é mais difícil pois exige um desenvolvedor experiente para criá-la de maneira mais efetiva. A Descrição 3 apresenta a notação estrutural para a construção de um comparador de 1 *bit*.

Descrição 3 Comparador de 1 *bit* - Descrição estrutural

```
01: architecture estrutural of comparador is
02: signal l1: bit;
03: begin
04:     u1: entity xor2 portmap (e1, e2, l1);
05:     u2: entity and2 portmap (e1, l1, s);
06: end estrutural;
```

A linguagem VHDL apesar de ter a estrutura e a interface de codificação extremamente parecidas com as linguagens de programação de *software*, possui algumas diferenças importantes:

- VHDL provê mecanismos para modelar a concorrência e sincronização que ocorrem no nível físico do *hardware*, de forma que, os comandos são executados paralelamente (com exceção de comandos inseridos em um bloco *process*), ao contrário de linguagens de programação seqüenciais;
- O código VHDL não gera binário, mas é processado por um simulador;
- A "compilação" de uma codificação VHDL produz uma descrição de *hardware* ao invés de um arquivo executável.

3.3 Integração Física da Descrição

A implementação física de uma descrição HDL em um dispositivo FPGA consiste geralmente nas seguintes etapas:

1. Descrição lógica do circuito utilizando uma HDL, tal como VHDL ou Verilog, ou através de seu desenho em um editor de esquemáticos.

2. Transformar a descrição HDL ou esquemático em uma *netlist*[10] com o auxílio de um programa de sintetização lógica.

3. Mapeamento lógico das portas e suas interconexões no respectivo modelo de FPGA, que será usado no mapeamento físico, através de ferramentas de roteamento. Os CLBs do FPGA podem posteriormente ser decompostos em LUTs para desempenhar as operações lógicas. Os CLBs e LUTs são combinados com vários recursos de roteamento. A ferramenta então recolhe a *netlist* em grupos que se encaixam dentro das LUTs para depois assinalar as coleções de portas aos específicos CLBs, até que a abertura ou fechamento das chaves nas matrizes de roteamento conecte as portas.

4. Uma vez a completa a fase de implementação, um programa extrai o estado das chaves nas matrizes de roteamento e gera a *bitstream*.

5. Para finalizar, é realizado o *download* da *bitstream* para o *chip* físico do FPGA (usualmente embutido em um sistema maior). As chaves eletrônicas no FPGA abrem ou fecham em resposta aos *bits* binários da *bitstream*. Quando o *download* é completado, o FPGA irá desempenhará as operações especificadas no código HDL ou esquemático. As operações do circuito podem então ser testadas através da indução de tensões nos pinos de E/S do FPGA.

A Figura 3.3 [XES04] ilustra as etapas descritas acima:

3.4 Equipamento Base

Para a síntese física do AIDA-16, foi utilizada a placa de prototipação XSA-100 [XES04a] com o FPGA Spartan II XC2S100 [XIL04] e a placa de expansão XST-2.1 [XES04b] com recursos adicionais de E/S.

A placa XSA-100 pode ser ligada através de uma fonte de alimentação com que forneça uma tensão de 9V DC com um plugue tipo fêmea de centro positivo com 2.1 mm de espessura. A conexão com o microcomputador é realizada através da porta paralela com um cabo de dois conectores DB25, por onde se pode fazer o *download* das descrições implementadas para o FPGA. São oferecidos também recursos de E/S para se conectar um periférico de interface PS/2 (como um mouse ou teclado) com um conector J4 mini-DIN e um dispositivo de interface VGA (como um monitor de vídeo) com um conector J3 de 15 pinos.

Dentre os recursos funcionais da placa XSA-100 estão um oscilador programável de 100 MHz usado para geral o sinal de *clock* para os demais componentes da placa, 128 KBytes de memória *flash* para armazenamento não volátil de dados ou configurações de *bitstreams*, 16 Mbytes de memória SDRAM para armazenamento volátil acessível pelo FPGA, um *display* de 7 segmentos para dar um *feedback* do funcionamento da placa ao usuário , 4 *DIP-Switches* que permitem passar configurações em tempo de execução para a placa e 1 botão para o envio de informações momentâneas para o FPGA. Por meio destes recursos, o usuário pode interagir com a descrição adotada pelo FPGA em um dado momento.

Já a placa de expansão XST-2.1, oferece alguns recursos extras de E/S para a interação com outros periféricos, tais como: interface RS-232 para comunicação serial, interface USB

[10] Descrição das portas lógicas do circuito e a maneira como elas são interconectados

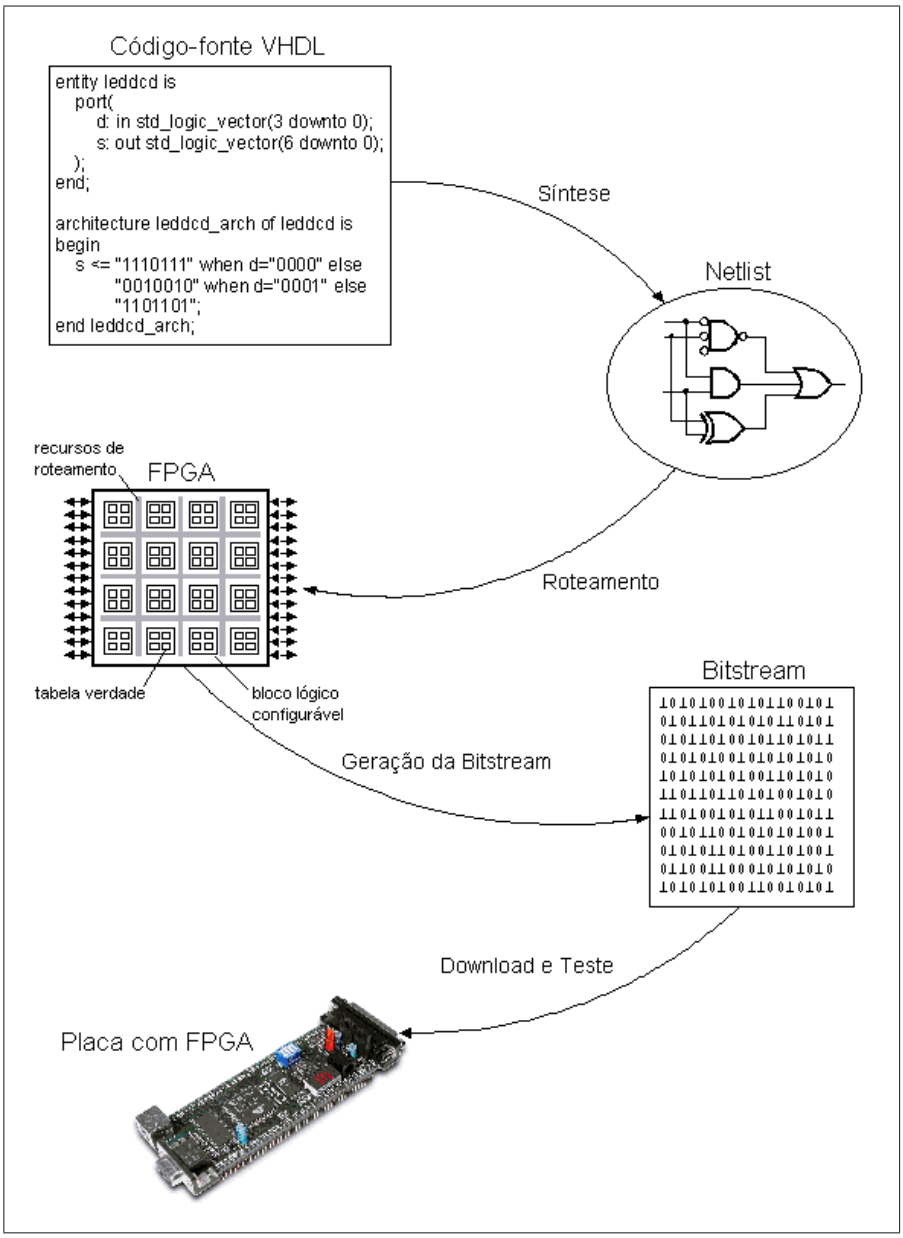

Figura 3.3: Etapas para a integração física no dispositivo programável

1.1, interface IDE para conexão de discos rígidos, *codec stereo* de áudio AK4551 de 20 *bits* que aceita dois canais de entrada, entre outros.

A Figura 3.4 [XES04b] mostra a placa de prototipação XSA-100 encaixada na placa de extensão XST-2.1:

Depois de efetuado o encaixe das placas de prototipação, o FPGA passa a contar com os recursos extras oferecidos pela placa de expansão. A comunicação entre o FPGA e os recursos oferecidos pelas placas de prototipação é realizado através da junção de camadas de metal, por onde são realizadas as conexões.

A Figura 3.5 [XES04a] ilustra as principais conexões e barramentos de comunicação entre os componentes da placa XSA-100:

A Tabela 3.1 [XIL04]lista as principais características dos FPGAs da família Spartan II:

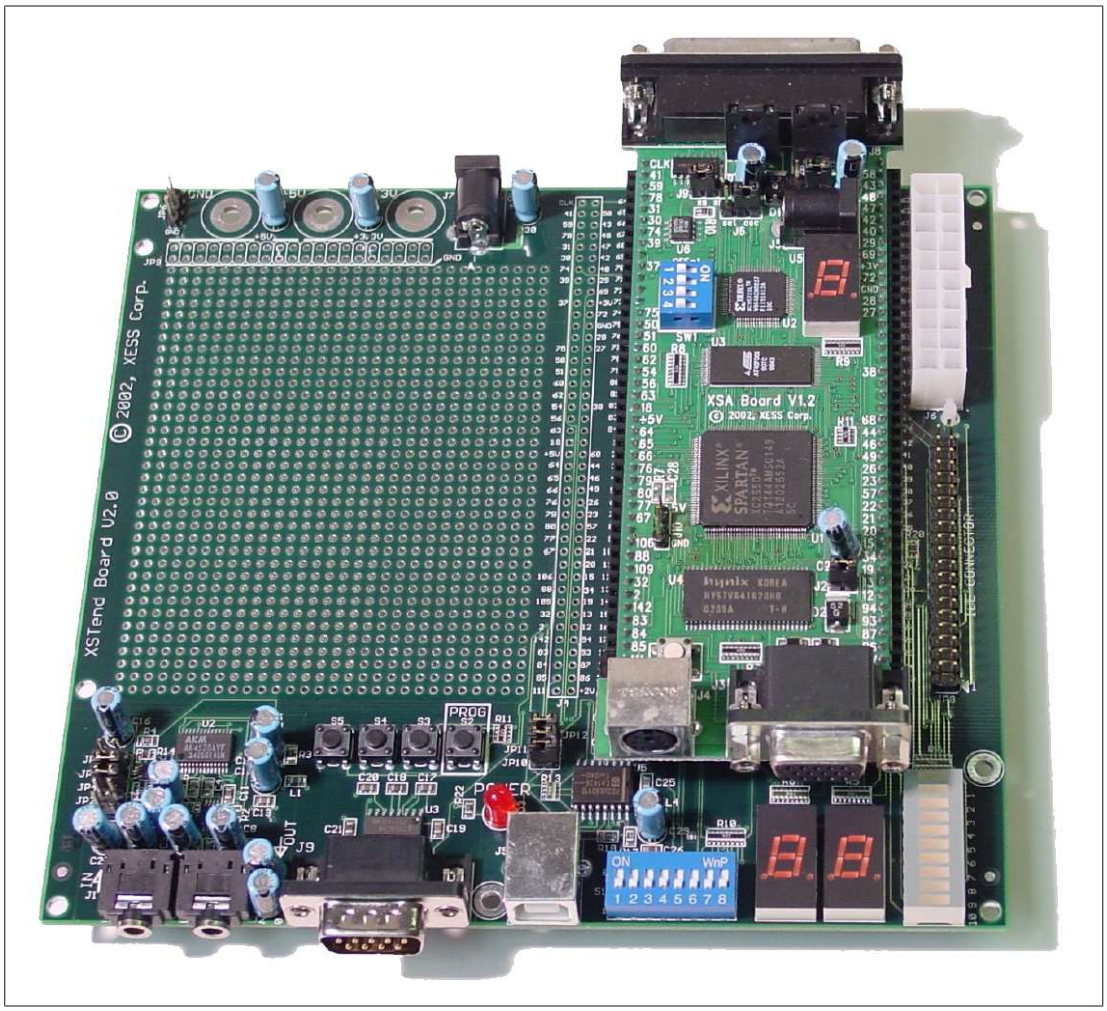

Figura 3.4: Placa de prototipação XSA-100 encaixada na placa de extensão XST-2.1

Tabela 3.1: Principais características dos FPGAs da família Spartan II

Dispositivo	Células lógicas	Portas lógicas	Matriz de CLBs	Número de CLBs	Pinos de E/S	Bits de RAM	Blocos de RAM
XC2S15	432	15000	8 x 12	96	86	6144	16K
XC2S30	972	30000	12 x 18	216	132	13824	24K
XC2S50	1728	50000	16 x 24	384	176	24576	32K
XC2S100	2700	100000	20 x 30	600	196	38400	40K
XC2S150	3888	150000	24 x 36	864	260	55296	48K
XC2S150	5292	200000	28 x 42	1176	284	75264	56K

A primeira coluna da Tabela 3.1 mostra o modelo do dispositivo FPGA. A coluna seguinte indica a quantidade de células lógicas presentes em cada dispositivo. A terceira coluna especifica a quantidade máxima de portas lógicas que cada dispositivo pode implementar de acordo com a sua quantidade de células lógicas. A quarta coluna apresenta o tamanho da matriz (em linhas e colunas) de CLBs dos dispositivos. A coluna seguinte apresenta a quantidade total

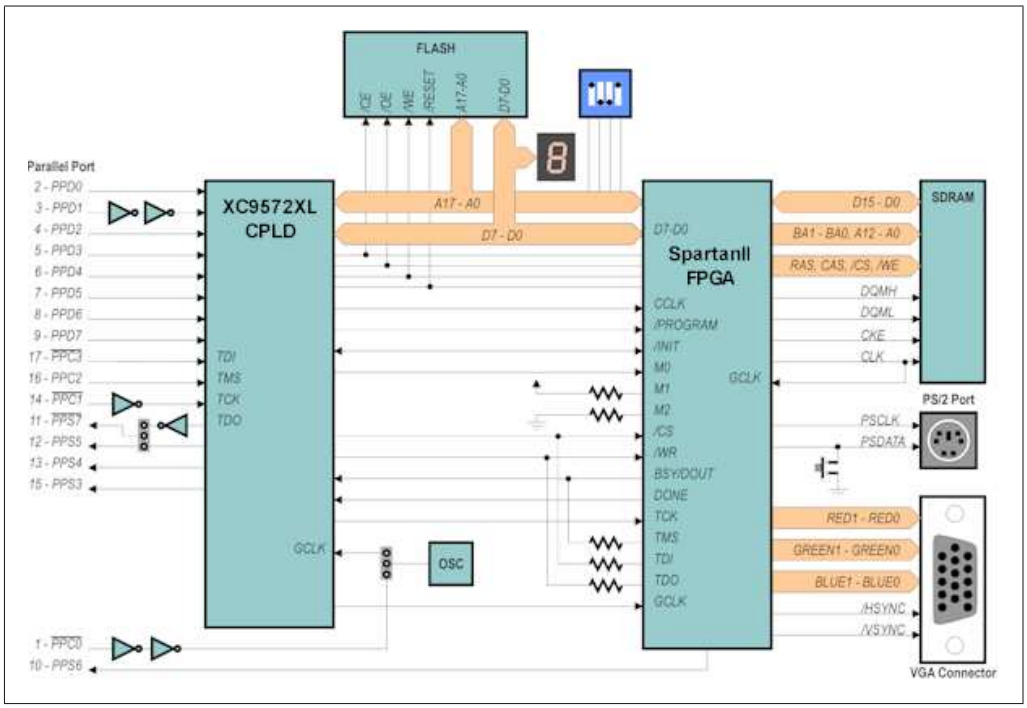

Figura 3.5: Conexões entre os componentes da placa XSA-100

de CLBs, obtida através da multiplicação dos valores da coluna anterior. A sexta coluna mostra a quantidade de pinos de E/S disponíveis ao usuário, por onde é possível realizar a comunicação com o exterior do circuito. A sétima e oitava colunas apresentam respectivamente quantidade de *bits* de RAM distribuídos e o total de *bits* dos blocos de RAM.

3.5 Softwares para Projeto

Para a descrição VHDL do AIDA-16 foi utilizado o *software* Active-HDL [ALD04] que oferece recursos para simulação e geração do *testbench*[11] do circuito criado. A partir desta ferramenta foram descritos, simulados e testados todos os módulos funcionais presentes no AIDA-16.

A Figura 3.6 apresenta o ambiente de simulação do Active-HDL:

Na parte central da Figura 3.6 estão os canais de E/S do circuito descrito. O usuário pode visualizar o formato de onda com base no tempo de simulação para cada *bit* destes canais quando aplicados os valores de entrada do circuito. Na parte inferior se localiza o console que mostra as mensagens de depuração geradas durante a compilação da descrição. Na parte inferior esquerda são apresentados os estados das variáveis e sinais declarados na descrição do circuito que está sendo simulado. Na parte superior esquerda são mostrados os componentes presentes na descrição, por onde o usuário pode fazer a seleção do componente que deseja simular.

Já para as tarefas de transformação da descrição VHDL do circuito em *netlist*, mapeamento lógico dos canais do circuito descrito para os pinos do FPGA, geração da *bitstream* e seu *download* para o FPGA, foi utilizado o *software* WebPACK [XES04c]. Esta ferramenta, entretanto, possibilita a realização todos os passos para a implementação de um circuito, desde a descrição

[11] Método para verificação do comportamento do circuito frente aos sinais de entrada aplicados

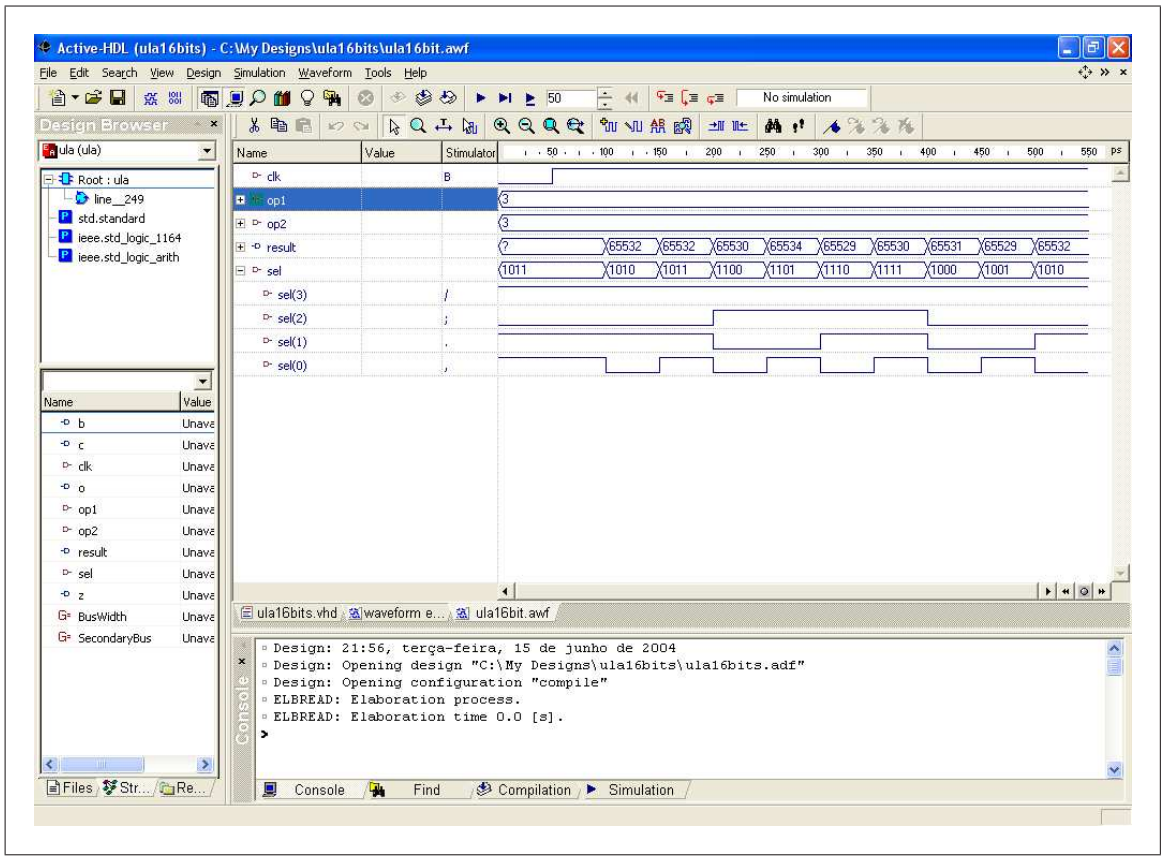

Figura 3.6: Ambiente de simulação do Active-HDL

VHDL até a síntese física, sendo que, o *software* Active-HDL foi preferencialmente utilizado na descrição devido a grande quantidade de recursos oferecidos para simulação e realização de testes pré-síntese.

Capítulo 4

Especificação do AIDA-16

O AIDA-16 utiliza uma arquitetura híbrida. Algumas das características das arquiteturas CISC e das arquiteturas RISC foram empregadas na sua construção. Por exemplo, este microprocessador usa um conjunto de instruções complexas e microinstruções binárias para representá-las, refletindo características CISC. Por outro lado, o AIDA-16 faz uso da arquitetura *load-store* e, conseqüentemente, possui um grande número de registradores, denotando características RISC.

O presente capítulo tem por objetivo apresentar o processo de concepção do AIDA-16. São discutidos os detalhes referentes à sua arquitetura interna, desenvolvimento de seus módulos funcionais e a interface de comunicação entre eles. São apresentados ainda testes de para observar o comportamento do circuito através de simulação da execução de determinadas instruções. Além disso, este capítulo também poderá servir ao leitor como um manual de utilização do AIDA-16, pois são abordados os seus formatos de representação e conjunto de instruções, modos de endereçamento, estágios de execução e todas as suas diretivas de programação. Para finalizar são mostradas também as estatísticas de síntese e design do circuito no dispositivo FPGA.

4.1 Conjunto de Registradores

A seguir são listados os registradores (específicos e de propósito geral) do AIDA-16:

- PC (*Program Counter*): Registrador que armazena o endereço da próxima instrução a ser executada. Possui 16 *bits*.
- IR (*Instruction Register*): armazena a instrução atual com todos os seus campos. Possui 16 *bits*.
- MAR (*Memory Address Register*): é o registrador que armazena o endereço de memória que deverá ser acessado. Possui 16 *bits*.
- MDR (*Memory Data Register*): é o registrador que armazena o valor que deve ser lido/escrito na memória. Possui 16 *bits*.
- *Flags* Z, C e V: Registradores de 1 *bit* (discutidos na Seção A.12).
- Um banco de 16 GPRs, cada um com 16 *bits*, denominados $\$r_0$ a $\$r_13$ (0 a 13) e $\$t_0$ a $\$t_1$ (14 a 15).

A Figura 4.1 mostra o esquema externo do circuito do banco de GPRs do AIDA-16 com todos os seus canais de E/S.

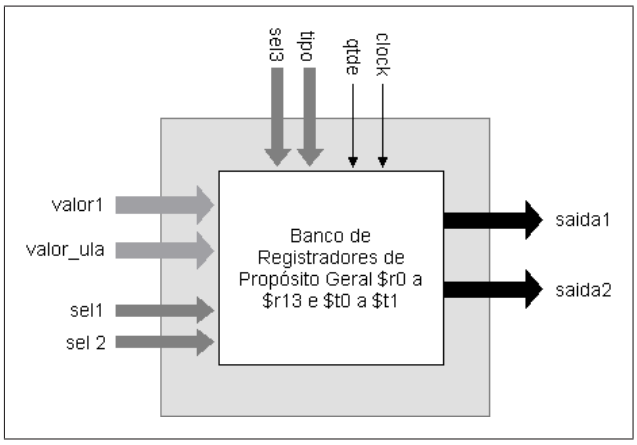

Figura 4.1: Esquema externo do circuito do banco de 16 GPRs

Na parte esquerda do circuito estão os canais de entrada $valor_1$ e $valor\_ula$, que informam respectivamente os valores de 16 *bits* referentes ao canal reservado para carga de constantes em registradores e ao canal reservado para o retorno do resultado de uma operação realizada pela ULA. Os canais de entrada sel_1 e sel_2 selecionam respectivamente quais são os GPRs do AIDA-16 que irão armazenar os valores dos canais $valor_1$ e $valor\_ula$.

Na parte superior da Figura 4.1 se encontra o canal sel_3, que informa qual será o registrador que irá armazenar o resultado da operação, em caso de ser uma instrução deste gênero. Através do canal $tipo$ (a lista completa dos tipos permitidos são listados na Figura 4.1), é possível determinar qual é o tipo de acesso que está sendo realizado e, conseqüentemente, distinguir quais os canais que devem ser ignorados. O sinal $qtde$ serve para determinar se estão sendo informados um ou dois operandos. Finalmente, o sinal $clock$ habilita o funcionamento do banco de registradores do AIDA-16.

A parte direita da Figura 4.1 apresenta os canais $saida_1$ e $saida_2$, que são os canais de saída do banco de registradores. Ambos os canais são de 16 bits e, dependendo do tipo de acesso podem não ser usados simultaneamente.

A Figura 4.1 apresenta quais são os tipos de acesso permitidos ao banco de GPRs:

Código do tipo de acesso	Descrição do acesso	Tipo de instrução	Canais/sinais utilizados	
000	Leitura direta com *link* para a ULA	Lógicas e Aritméticas	sel1, sel3, tipo, qtde, clk e saida1	um operando
			todos menos valor1 e v_ula	dois operandos
001	Leitura indireta com *link* para a ULA	Lógicas e Aritméticas	sel1, sel3, tipo, qtde, clk e saida1	um operando
			todos menos valor1 e v_ula	dois operandos
010	Carregamento de parte baixa de registrador	Transferência de dados	valor1, sel1, tipo, qtde, clk	
011	Carregamento de parte alta de registrador	Transferência de dados	valor1, sel1, tipo, qtde, clk	
100	Movimentação entre registradores	Transferência de dados	sel1, sel2, tipo qtde, clk	
101	Leitura com *link* para a memória	Transferência de dados	todos menos valor1 e v_ula	
110	Retorno da ULA	Lógicas e Aritméticas	v_ula, sel3, tipo, clk	
111	Reservado	Reservado	Reservado	

Figura 4.2: Códigos enviados para o banco de registradores

A primeira coluna indica os códigos dos tipos de acesso que podem ser passados ao banco de GPRs, ao lado está a descrição do tipo de operação correspondente a cada código, na coluna seguinte estão relacionados os tipos de operação que podem habilitar cada um dos tipos de acesso e na última coluna são listados os canais e sinal que são necessários pelos tipos de acesso. Por exemplo, em uma instrução de carregamento de registrador, o código do tipo de acesso é 010_2 e somente precisam ser verificados os canais $valor_1$ e sel_1 para se determinar respectivamente onde e o que escrever, além do canal $tipo$ e clk, que são utilizados por todos os tipos de acesso. Nesta instrução os canais $valor\_ula$, sel_2, $saida_1$ e $saida_2$ e o sinal $qtde$ podem ser ignorados, pois não influenciam no resultado do acesso. As linhas hachuradas representam os códigos que não são utilizados pelo banco de GPRs.

A Figura 4.3 apresenta o esquema interno de componentes e conexões do banco de GPRs do AIDA-16:

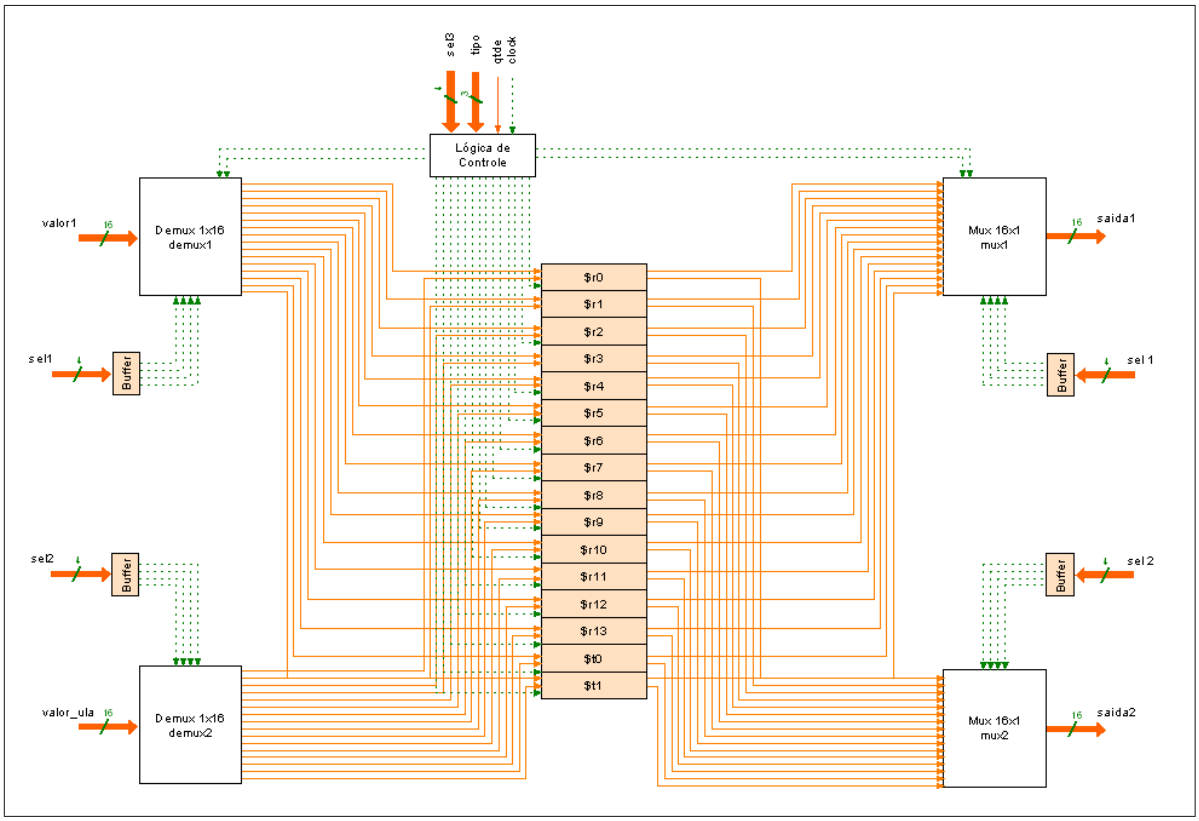

Figura 4.3: Esquematização interna do banco de 16 GPRs

Os 16 *bits* do canal $valor_1$ estão conectados ao demultiplexador $demux_1$, que é controlado pelo canal sel_1. Por meio deste canal de 4 *bits* é possível realizar a comutação dos *bits* do canal $valor_1$, ligados à entrada do demultiplexador, que tem seus 16 *bits* de saída conectados a cada um dos GPRs do banco de registradores. Para se carregar um determinado registrador com um valor qualquer, por exemplo, basta enviar através do canal sel_1 o código de seleção associado ao registrador de destino dos dados para que o demultiplexador $demux_1$ envie o conteúdo do canal $valor_1$ para a saída correta. Já os canais $valor\_ula$ e sel_2 utilizam-se do demultiplexador $demux_2$, para realizar operações de escrita do valor de retorno da ULA nos registradores do banco de GPRs.

Em uma instrução de soma que utiliza o modo de endereçamento direto, por exemplo, existem dois registradores que devem ser lidos para buscar os operandos e enviá-los para a ULA. Dessa maneira, os dois operandos podem ser buscados simultaneamente, cada um utilizando uma via de comunicação dentro do circuito do banco de GPRs.

O componente *Lógica de Controle* é responsável por interpretar os o conteúdo dos dados que lhe são entregues e acionar o funcionamento dos demais blocos do conjunto de GPRs do AIDA-16. Entre estes dados, está o canal sel_3, que é utilizado em instruções aritméticas e lógicas para determinar qual registrador armazenará o resultado da operação executada pela ULA. O canal $tipo$ indica qual é o tipo de acesso que se deseja realizar (ver referência completa na Figura 4.1). O sinal $qtde$ indica se a via de comunicação por onde o segundo operando é informado deve ser levada em consideração. Em outras palavras, determina se a instrução corrente possui um ou dois operandos. Finalmente, o sinal $clock$ é o responsável por acionar o funcionamento do banco de GPRs.

Na saída do circuito estão dois multiplexadores mux_1 e mux_2 que, dependendo do tipo de acesso, recebem respectivamente os valores $valor_1$ e $valor\_ula$ dos registradores selecionados por sel_1 e sel_2. As entradas destes multiplexadores estão conectadas a cada uma das saídas do conjunto de GPRs, de forma que eles agem para comutar os *bits* de suas entradas em apenas uma saída com base no valor indicado por sel_1 e sel_2. Por exemplo, em uma instrução de incremento no valor do registrador $\$r_0$, deve-se informar o valor correspondente a este registrador no canal sel_1 e o tipo de acesso no sinal $tipo$ Então, o demultiplexador $demux_1$ saberá qual registrador deve ser acessado. Em seguida, como se trata de uma operação aritmética, o conteúdo do registrador $\$r_0$ é copiado para a saída do multiplexador mux_1 através do canal sel_1 que indica qual o canal dos registradores deve ser comutado. Para finalizar, o valor é enviado do banco de GPRs para a ULA que, em seguida retorna o resultado da operação para ser escrito no registrador apropriado.

4.2 Modos de Endereçamento

Como o AIDA-16 foi implementado com base na arquitetura *load-store*, é importante fornecer ao programador a possibilidade de manipular de maneiras diferentes as informações que estão armazenadas em registradores. Para isso, foram implementados três modos de endereçamento, que são utilizados pelas instruções lógicas e aritméticas e podem ser selecionados a partir do campo *me* do formato-R de representação de instruções (os formatos de representação de instruções serão abordados em detalhes na Seção 4.4). A Figura 4.4 mostra os códigos binários que representam respectivamente cada um dos modos de endereçamentos suportados pelo AIDA-16:

Código Binário (campo *rc*)	Modo de Endedreçamento
00	Modo Direto
01	Modo Indireto
10	Modo Imediato
11	Reservado

Figura 4.4: Modos de endereçamento suportados

O campo *Código* da Figura 4.4 referencia o campo *me* do formato-R. Como foram reservados 2 *bits* na instrução para permitir a seleção do modo de endereçamento, mas, existem apenas 3 modos oferecidos, as instruções com a combinação binária "11" no campo *me* são imediatamente descartadas pelo decodificador de instruções do AIDA-16.

As subseções a seguir descrevem os modos de endereçamento suportados pelo AIDA-16. A expressão abaixo será utilizada para demonstrar a sua programação no AIDA-16 utilizando cada um dos modos de endereçamento disponíveis.

$$a = a + b$$

Para os exemplos que serão descritos nas seções a seguir, será presumido que as variáveis *a* e *b* denotam os registradores $\$t_0$ e $\$t_1$ respectivamente, exceto parao modo imediato, onde não existe endereçamento.

4.2.1 Modo Imediato

Neste modo de endereçamento, o valor do campo operando contém próprio dado que se deseja operar, ou seja, as instruções que utilizam este formato são executadas mais rapidamente por não precisarem realizar o ciclo de busca do(s) operando(s) nos registradores do AIDA-16. Como os modos de endereçamento são utilizados apenas pelas instruções descritas pelo formato-R, somente é permitida a indicação de constantes imediatas de no máximo 4 *bits*. O problema deste modo de endereçamento é que existe a necessidade de se pré-determinar um registrador específico do AIDA-16 para armazenar o resultado das operações que o utilizem, já que nos campos destinados aos operandos não constam endereços. Dessa maneira, o resultado das operações imediatas do AIDA-16 é direcionada para um dos registradores temporários $\$t_0$ ou $\$t_1$, que podem ser selecionados a partir do campo do *rd* da instrução, sendo que, o *bit* não assinalado referencia $\$t_0$ e o *bit* assinalado referencia $\$t_1$.

Considerando a expressão da Seção 4.2, e supondo-se que os valores atribuídos as variáveis *a* e *c* fossem respectivamente 6_{10} e 15_{10}, a instrução correspondente utilizando o modo imediato para executá-la seria:

```
add 0, 10, 0110, 1111
```

Assim, quando o decodificador de instruções do AIDA-16 recebe esta instrução, ao fazer a leitura do terceiro campo, já determina que se trata de uma instrução que utiliza o modo imediato e, conseqüentemente, que os valores dos operandos correspondem a constantes imediatas, que podem ser enviadas diretamente para a ULA. O resultado da operação é copiado no registrador $\$t_0$, pois o segundo campo da instrução não está assinalado.

A grande vantagem da utilização deste modo de endereçamento é a agilidade na execução das operações já que não é necessário realizar o acesso a registradores para buscar os operandos. A desvantagem está relacionada à limitação da capacidade de representação das constantes imediatas já que deve-se respeitar o tamanho dos campos especificados pelo formato de representação de instrução que está em uso.

4.2.2 Modo Direto

Nesse modo de endereçamento, o valor do campo operando da instrução indica um dos registradores internos do AIDA-16, o qual contém o valor do operando. Como os tamanhos

dos campos reservados para os operandos no formato-R são de 4 *bits*, é possível que sejam endereçados 16 registradores utilizando o modo direto.

A instrução do AIDA-16 necessária para executar a expressão descrita na Seção 4.2 utilizando o modo de endereçamento direto é a seguinte:

```
add 0, 00, 1110, 1111
```

O terceiro campo da instrução informa ao decodificador de instruções do AIDA-16 que se está usando os campos restantes da instrução para informar endereços de registradores. Neste caso, a instrução indica que o registrador $\$t_0$ armazena o valor do primeiro operando e o $\$t_1$ armazena o valor do segundo operando, sendo que, o resultado da operação de adição deve ser colocado em $\$t_0$.

A desvantagem do modo de endereçamento direto está na limitação da capacidade de representar endereços. No caso do AIDA-16, se o programador desejar fazer uso do modo de endereçamento direto, ficará preso à capacidade de endereçar 16 registradores (mesmo que o microprocessador ofereça um número maior), pois os campos reservados aos operandos no formato-R apresentam 4 *bits* cada e, dependendo do tamanho do programa e do volume de manipulação de dados do código, pode haver a necessidade de se manter um número maior de valores dentro de registradores. Por outro lado, a principal deficiência do modo imediato é sanada, pois o valor que os operandos podem assumir é de até 65335_{10}, já que os registradores do AIDA-16 são todos de 16 *bits*.

4.2.3 Modo Indireto

No modo indireto, o valor do campo operando contém o endereço de um registrador cujo conteúdo é o endereço do registrador com o valor do dado a ser operado, de forma que, existe um duplo endereçamento e, o endereço intermediário é chamado de *ponteiro*.

Supondo-se que os registradores $\$r_0$ e $\$r_1$ contém os endereços dos registradores $\$t_0$ e $\$t_1$ respectivamente, abaixo é mostrada a instrução que deve ser entregue ao AIDA-16 para executar a expressão da Seção 4.2.

```
add 0, 01, 0000, 0001
```

O modo indireto é indicado no terceiro campo da instrução, então, os campos de endereçamento do formato-R estão armazenando respectivamente os endereços dos registradores onde os se encontram os dados a serem operados.

O endereçamento indireto resolve o problema principal do endereçamento direto, pois já que existe um duplo endereçamento, os endereços dos registradores que armazenam os valores dos operandos envolvidos na operação passam de 16_{10} para 65335_{10} (pelo fato de que os registradores possuem 16 *bits*). Em contrapartida, quando utilizado este modo de endereçamento, existe a necessidade de um ciclo a mais de execução em comparação com o modo direto, pois a busca do(s) operando(s) gasta dois ciclos de execução.

4.3 Conjunto de Instruções

O AIDA-16 possui um total de 30 instruções. O campo *op* (de 5 *bits*), presente em todos os formatos de representação de instrução, permite que sejam implementadas até 32 instruções,

mas os modos de endereçamento possibilitam que variações de uma mesma instrução possam ser utilizadas e, se levados em consideração, estabelecem a existência de 62 instruções diferentes.

O conjunto de instruções suportado pelo AIDA-16 pode ser dividido em 5 categorias:

- 7 instruções aritméticas: proporcionam capacidades computacionais para o processamento de dados numéricos.
- 8 instruções lógicas: operam nos *bits* de um operando.
- 4 instruções de transferência de dados: permitem carregar informações da memória principal em registradores internos, movimentar valores entre registradores e escrever diretamente nas células da memória principal.
- 6 instruções de desvio condicional: são usadas para fazer com que o programa tome cursos diferentes a partir da satisfação ou não de uma determinada condição.
- 1 instrução de desvio incondicional: serve para alterar o fluxo de controle seqüencial de um programa, possuindo um campo que armazena o endereço (lógico) alvo da ramificação.

A Tabela 4.1 apresenta o conjunto completo de instruções do microprocessador projetado:

A primeira coluna da Tabela 4.1 subdivide as instruções nas 5 categorias já mencionadas. A segunda coluna lista o mnemônico de cada umas das instruções do AIDA-16. Na terceira coluna é dado um exemplo de operação usando cada uma das instruções. Na quarta coluna são listados os *flags* de sinalização que são afetados pelas respectivas instruções. A quinta coluna apresenta uma breve descrição de cada instrução.

As instruções aritméticas e lógicas utilizam o formato-R da Figura 4.5. Nas instruções de transferência de dados, as instruções `lli` e `lui` utilizam o formato-I apresentado na Figura 4.6 e as demais instruções desta categoria utilizam o formato-*Load/Store* da Figura 4.10. As instruções de desvio condicional e incondicional compartilham o formato-*Branch* da Figura 4.11.

A Tabela 4.2 mostra algumas informações adicionais sobre o conjunto de instruções do AIDA-16:

A segunda coluna da Tabela 4.2 apresenta o código de operação de cada uma das instruções do AIDA-16. Estes códigos foram organizados de maneira que através da vizualização de seus dois primeiros *bits* é possível identificar a categoria de instrução que está sendo executada. A quarta coluna apresenta o mnemônico referente a cada uma das instruções. Na quinta coluna as instruções são relacionadas ao formato de representação de instrução que podem utilizar. A sexta coluna mostra o número de ciclos de execução necessários para executar cada uma das instruções.

4.4 Representação de Instruções

O formato de representação e a maneira como as informações serão manipuladas são fatores cruciais no projeto de qualquer sistema eletrônico que realize processamento de dados. A representação de instruções de um microprocessador pode ser realizada utilizando diversos

Tabela 4.1: Conjunto de instruções da arquitetura projetada

Categoria	**Instr.**	**Exemplo**	**Significado**	**Flag**[1]	**Descrição**
NA	*nop*	*nop*	NA	NA	nenhuma operação
Aritmética	*add*	$add\ 0,\ 00,\ \$r_0, \$r_1$	$\$r_0 \leftarrow \$r_0 + \$r_1$	c, o, z	adição
	sub	$sub\ 0,\ 00,\ \$r_0, \$r_1$	$\$r_0 \leftarrow \$r_0 - \$r_1$	c, o, z	subtração
	inc	$inc\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow \$r_0 + 1$	c, o, z	incremento
	dec	$dec\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow \$r_0 - 1$	c, o, z	decremento
	mul	$mul\ 0,\ 00,\ \$r_0, \$r_1$	$\$r_0 \leftarrow \$r0 \times \$r1$	c, o, z	multiplicação
	div	$div\ 0,\ 00,\ \$r_0, \$r_1$	$\$r0 \leftarrow \$r_0 \div \$r_1$	c, o, z	divisão
	mod	$mod\ 0,\ 00,\ \$r_0, \$r_1$	$\$r_0 \leftarrow \$r_0\ mod\ 1$	c, o, z	resto da divisão
Lógica	*shr*	$shr\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow srh(\$r_0)$	c, z	desloc. para a direita
	shl	$shl\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow shl(\$r_0)$	c, z	desloc. para a esquerda
	ror	$ror\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow ror(\$r_0)$	z	rotação para a direita
	rol	$rol\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow rol(\$r_0)$	z	rotação para a esquerda
	or	$or\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow (\$r_0) or (\$r_1)$	z	or lógico *bit*-a-*bit*
	and	$and\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow (\$r_0) and (\$r_1)$	z	and lógico *bit*-a-*bit*
	xor	$xor\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow (\$r_0) xor (\$r_1)$	z	xor lógico *bit*-a-*bit*
	not	$not\ 0,\ 00,\ \$r_0$	$\$r_0 \leftarrow not(\$r_0)$	z	complemento
Transferência de dados	*li*	$li\ 0,\ 50$	$\$t_0 \leftarrow 50$	NA	carga baixa de reg.
	lui	$lui\ 0,\ 500$	$\$t_0 \leftarrow 500$	NA	carga alta de reg.
	lw	$lw\ \$r_0, \$r_1$	$\$r_0 \leftarrow memory[\$r_1]$	NA	busca *word* na memória
	sw	$sw\ \$r_0, \$r_1$	$memory[\$r_0] \leftarrow \$r_1$	NA	carrega *word* na memória
	lb	$lb\ \$r_0, \$r_1$	$\$r_0 \leftarrow memory[\$r_1]$	NA	busca *byte* na memória
	sb	$sb\ \$r_0, \$r_1$	$memory[\$r_0] \leftarrow \$r_1$	NA	carrega *byte* na memória
	mov	$mov\ \$r_0, \$r_1$	$\$r_1 \leftarrow \$r_0$	NA	copia entre registradores
Desvio Condicional	*jv*	$jv\ 100$	$if\ v = 1\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se *overflow*
	jnv	$jnv\ 100$	$if\ v = 0\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se não *overflow*
	jz	$jz\ 100$	$if\ z = 0\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se zero
	jnz	$jnz\ 100$	$if\ n = 1\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se não zero
	jc	$jc\ 100$	$if\ c = 1\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se *carry*
	jnc	$jnc\ 100$	$if\ c = 0\ then\ \$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia se não *carry*
Desvio Incondicional	*jmp*	$jmp\ 100$	$\$pc \leftarrow 100 \times 2 + \$pc$	NA	desvia para endereço

[1] Indica os *flags* de sinalização que são afetados pela instrução

formatos. Cada formato de instrução tem suas características próprias, com suas vantagens e desvantagens, podendo ser eficaz em certas aplicações e desaconselhável em outras [MON96]. Em se tratando de um microprocessador ou um microcontrolador, o tamanho da instrução afeta diretamente a quantidade de memória externa ao circuito que poderá ser acessada. As instruções geralmente contêm o código da operação que se deseja efetuar e os endereços dos operandos envolvidos. Quanto maior a informação contida na instrução, menor será a quantidade de instruções necessárias em um programa e o conjunto de instruções presente no microprocessador. Em contrapartida, além de cada instrução ocupar um espaço muito grande na memória, é fato que grande parte das instruções codificadas em um programa é composta de atribuições de

Tabela 4.2: Informações adicionais sobre o conjunto de instruções

Categoria	**Código**	**Instrução**	**Significado**	**Formato**	**Ciclos de Execução**
NA	00000	nop	*no operation*	formato-R[1]	2
Aritmética	00001	add	*addition*	formato-R	$5, 5, 4$[5]
	00010	sub	*subtraction*	formato-R	$5, 5, 4$
	00011	inc	*increment*	formato-R	$5, 5, 4$
	00100	dec	*decrement*	formato-R	$5, 5, 4$
	00101	mul	*multiplication*	formato-R	$5, 5, 4$
	00110	div	*division*	formato-R	$5, 5, 4$
	00111	mod	*module of division*	formato-R	$5, 5, 4$
Lógica	01000	shr	*shift right*	formato-R	$5, 5, 4$
	01001	shl	*shift left*	formato-R	$5, 5, 4$
	01010	ror	*rotate right*	formato-R	$5, 5, 4$
	01011	rol	*rotate left*	formato-R	$5, 5, 4$
	01100	or	*logical or*	formato-R	$5, 5, 4$
	01101	and	*logical and*	formato-R	$5, 5, 4$
	01110	xor	*logical xor*	formato-R	$5, 5, 4$
	01111	not	*logical not*	formato-R	$5, 5, 4$
Transferência	10000	li	*load immediate*	formato-I[2]	3
de dados	10001	lui	*load upper immediate*	formato-I	3
	10100	lw	*load word*	formato-*Load/Store*[3]	3
	10010	sw	*store word*	formato-*Load/Store*	3
	10011	lb	*load byte*	formato-*Load/Store*	3
	10100	sb	*store byte*	formato-*Load/Store*	3
	10101	mov	*move*	formato-*Load/Store*	3
Desvio	11000	jv	*jump overflow*	formato-*Branch*[4]	3
Condicional	11001	jnv	*jump on not overflow*	formato-*Branch*	3
	11010	jz	*jump zero*	formato-*Branch*	3
	11011	jnz	*jump on not zero*	formato-*Branch*	3
	11100	jc	*jump carry*	formato-*Branch*	3
	11101	jnc	*jump on not carry*	formato-*Branch*	3
Desvio	11110	jmp	*jump*	formato-*Branch*	3
Incondicional					

[1] Formato de representação para instruções aritméticas e lógicas da Figura 4.5

[2] Formato de representação para carga de constantes em registradores da Figura 4.7

[3] Formato de representação para instruções de acesso à memória da Figura 4.10

[4] Formato de representação especial para instruções de desvios da Figura 4.11

[5] Para os modos direto, indireto e imediato respectivamente

valores a variáveis e desvios condicionais ou incondicionais, ou seja, instruções compostas de apenas um operando.

Apesar disso, o padrão adotado para as instruções aritméticas e lógicas do AIDA-16 utiliza a arquitetura de dois endereços para permitir que o programador tenha maior flexibilidade

na manipulação das informações e também para retratar de maneira mais fiel a arquitetura de endereçamento utilizada em microprocessadores comercialmente disponíveis.

Como a arquitetura do AIDA-16 é do tipo *load-store*, onde operações são realizadas somente entre registradores ou constantes imediatas, os endereços indicados na instrução correspondem a endereços de GPRs presentes em seu conjunto de registradores. Para tornar o processo de programação do AIDA-16 mais flexível, ao invés de pré-determinar um registrador específico para armazenar o resultado da execução de uma operação lógica ou aritmética, foi criado um campo dentro do pacote da instrução que permite selecionar um dos dois registradores que estão sendo endereçados na instrução corrente para este propósito. Por exemplo, considerando-se as instruções a seguir:

```
add 0, $r0, $r1
add 1, $r0, $r1
```

A primeira instrução informa que o microprocessador deve somar os conteúdos dos registradores $\$r_0$ e $\$r_1$ e armazenar o resultado no registrador $\$r_0$. A segunda instrução, também é uma operação de soma entre os mesmos registradores, mas o *bit* "1" ao lado da operação indica que o resultado da soma deve ser copiado para o registrador $\$r_1$.

A Figura 4.5 apresenta o *formato-R*, usado para lembrar o uso de registradores pela instrução, utilizado na representação de de todas as instruções lógicas e aritméticas do AIDA-16.

Figura 4.5: Formato-R para representação de instruções aritméticas e lógicas

A seguir, é descrito o significado de cada um dos campos do pacote de instruções do formato-R:

- op *(opcode)*: é o código da operação, serve para informar qual a instrução que o microprocessador deve executar.
- am *(addressing mode)*: seleciona o modo de endereçamento a ser utilizado (direto, indireto e imediato). Os modos de endereçamento servem para diferenciar a localização do(s) operando(s) da instrução e foram discutidos detalhadamente na Seção 4.2 do presente trabalho.
- rd *(destination register)*: define qual dos registradores presentes na instrução armazenará o resultado da operação executada.
- rs *(first register source)*: indica o registrador contendo o primeiro operando-fonte.
- rt *(second register source)*: indica o registrador contendo o segundo operando-fonte.

Os dados e endereços do AIDA-16 são de 16 *bits*. Logo pode-se dizer que a *palavra* deste microprocessador é 16 *bits* e que a quantidade máxima de células de memória que podem ser endereçadas é 2^{16} ou 65536_{10} células. Como o tamanho de cada célula de memória é de 8 *bits*, em cada acesso à memória são trazidas duas células consecutivas.

A Figura 4.6 (a) é uma representação decimal de uma instrução no formato-R. O primeiro campo indica ao microprocessador que esta é uma operação de adição. O segundo campo indica que se está usando o modo de endereçamento direto. O terceiro campo, informa qual dos registradores presentes na instrução armazenará o resultado da operação (neste caso, o registrador que contém o primeiro operando-fonte). O quarto campo informa em qual registrador está presente o primeiro operando-fonte ($5 = \$r_5$). O quinto e último campo informa o registrador que armazena o segundo operando-fonte ($12 = \$r_{12}$). A Figura 4.6 (b) tem o mesmo significado, porém, na representação binária. Este formato é a linguagem entendida internamente pelo microprocessador, ou seja, seu formato de instrução.

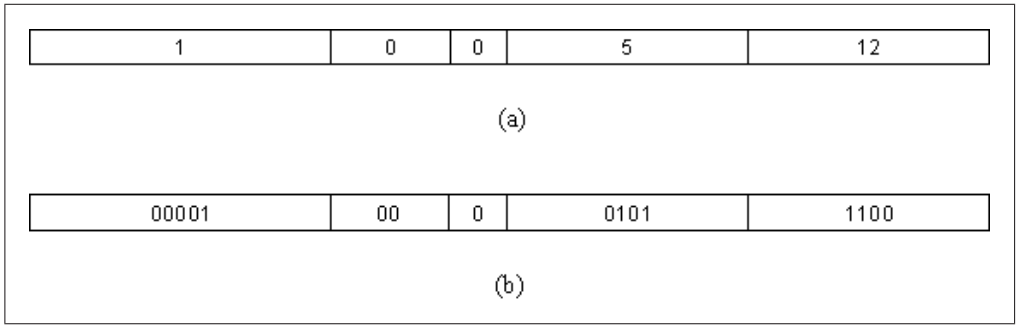

Figura 4.6: Representações decimal e binária de uma instrução

É muito comum que programas usem constantes em diversas operações que ocorrem com muita freqüência, como, por exemplo, o incremento de um índice para fazê-lo apontar para o próximo elemento de um *array*, a contagem de iterações de um *loop*, ou o ajuste do ponteiro de pilha em chamadas a procedimentos aninhados [PAT00]. Por isso, o grande problema do formato de instrução definido acima, está na necessidade de lidar com campos maiores dos que os que foram especificados na instrução. Por exemplo, para se atribuir uma constante a um registrador, é necessário que o valor da constante não seja maior que 15_{10} que é o máximo que se pode representar com os quatro *bits* dos campos *rs* ou *rt*, que são os únicos que ficam disponíveis em uma instrução deste tipo.

Para manter a regularidade no tamanho da instrução, optou-se pela criação de um segundo formato de instrução, que pode ser visualizado na Figura 4.7, chamado *formato-I*, pois referencia constantes imediatas, para representar de instruções de carregamento de constantes em registradores e possibilitar que se trabalhe com constantes com mais de 4 *bits*.

Como em operações de carregamento de registradores só é necessária a especificação de um operando, pode-se juntar os campos *rs* e *rt* da Figura 4.5 para formar um campo de 8 *bits* (campo *v* da Figura 4.7), que reserva espaço para constantes com valor máximo de 255_{10}. No formato-I, o campo *op* continua a especificar a instrução que está sendo referenciada. Os dois *bits* do segundo campo (área hachurada) são ignorados pelo decodificador de instruções. O campo *rd* seleciona qual registrador (de dois registradores pré-determinados, chamados registradores temporários $\$t_0$ e $\$t_1$) se deseja escrever. Os registradores $\$t_0$ e $\$t_1$ fazem parte do conjunto de 16 GPRs do AIDA-16 e foram discutidos com detalhes na Seção 4.1.

Considerando-se a instrução a seguir, onde os valores correspondem respectivamente aos campos *op*, *rd* e *v* do formato de instrução da Figura 4.7:

```
lli 0, 180
```

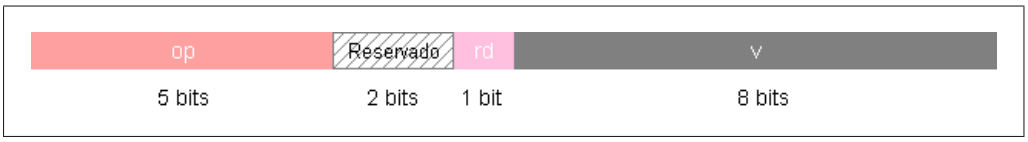

Figura 4.7: Formato-I para tratamento de constantes imediatas

O registrador $\$t_0$ é carregado com o valor 180_{10}. A seleção do registrador $\$t_0$ como destino do carregamento é realizada através do valor "0" no campo *rd*. Se o formato-R da Figura 4.5 fosse utilizado esta operação não poderia ser efetuada, pois o valor 180_{10} ultrapassa a capacidade de qualquer um dos campos de 4 *bits* que estariam disponíveis. Já com o formato de instrução da Figura 4.7, o procedimento pode ser realizado sem problemas.

A questão que surge, é como identificar qual o formato de instrução que o programador deseja utilizar. A resposta está no campo *op* da instrução, uma vez os formatos instrução estão associados aos códigos de operação definidos no AIDA-16. Então, no momento em que a instrução é decodificada, o microprocessador já consegue determinar qual o formato que está sendo utilizado. Por exemplo, ao receber a instrução do exemplo anterior, o microprocessador ao identificar que se trata de uma instrução de carregamento de registrador, já associa o conteúdo da instrução ao formato-R.

Para se realizar a soma de 100_{10} com 150_{10} por exemplo, deve-se utilizar as seguintes linhas de código:

```
lli 0, 100
lli 1, 150
add 0, 0, $t0, $t1
```

Primeiramente, carrega-se o valor 100_{10} no registrador $\$t_0$ (o registrador $\$t_0$ é selecionado através do *bit* "0" no campo *rd*). Em seguida, o valor 150_{10} é armazenado em $\$t_1$ (o registrador $\$t_1$ é escolhido setando-se o campo *rd*). Finalmente, executa-se uma instrução de adição entre os registradores $\$t_0$ e $\$t_1$, armazenando-se a resposta da operação em $\$t_0$. É importante ressaltar que as duas primeiras instruções estão utilizando o formato-I da Figura 4.7 por se tratarem de operações de carregamento constantes em registradores, enquanto que a última instrução se refere ao formato-R da 4.5 por se tratar de uma instrução aritmética. A associação completa das instruções com os formatos de representação instrução definidos foi abordada na Seção 4.3.

A Figura 4.8 mostra a notação em linguagem de máquina das instruções descritas anteriormente:

op	am	rd	rs	rt
10000	xx	0	01100100	
10000	xx	1	10010110	
00001	00	0	1110	1111

Figura 4.8: Descrição das operações em linguagem de máquina

No primeiro campo da primeira instrução está armazenado o código 00110_2 que corresponde à instrução `lli`, ou seja, está indicando que se carregar um registrador. O próximo

campo seleciona o registrador temporário $\$t_0$ como destino do carregamento. No terceiro campo está localizado o valor 100_{10} (01100100_2), que será carregado (neste caso em $\$t_0$). A instrução seguinte é idêntica a primeira, com exceção de que o registrador a ser carregado agora é $\$t_1$ (campo *rd* da instrução), e do valor, que agora é 150_{10} (10010110_2).

A última instrução possui no campo *op* o valor 01110_2 que corresponde a operação de adição. O campo *am* indica que se está utilizando o endereçamento direto. O terceiro campo informa ao microprocessador que o resultado da operação deve ser armazenado no registrador onde se encontra o primeiro operando. No penúltimo campo está o endereço do registrador que armazena o primeiro operando, 1110_2 (referenciando o registrador $\$t_0$). O último campo guarda a localização do registrador que armazena o segundo operado 1111_2 (referenciando o registrador $\$t_1$).

Aqui cabe colocar que os registradores $\$t_0$ e $\$t_1$ além de serem pré-determinados para algumas tarefas, podem também ser acessados em instruções de qualquer formato, como quaisquer outros GPRs do AIDA-16. Estes registradores utilizam uma nomenclatura diferenciada para facilitar a compreensão do leitor, mas poderiam ser chamados de $\$r_{14}$ e $\$r_{15}$ respectivamente.

De qualquer forma, mesmo com este novo formato de representação, o tratamento de constantes ainda é limitado ao valor 255_{10}. Dependendo da necessidade do programador, isso pode tornar impraticável, ou pelo menos trabalhosa, a execução de um determinado programa. Por exemplo, se um programador desejar carregar um registrador com o valor 1000_{10} para acessar um o índice de um vetor, deverá carregá-lo primeiramente com o valor 250_{10} e em seguida executar uma seqüência de três instruções de adição com o conteúdo atual do registrador e o valor 250_{10}, ou então executar uma instrução para multiplicar o conteúdo do registrador por quatro.

Como o tamanho dos registradores internos é de 16 *bits*, fica-se preso à capacidade de representação de constantes com no máximo 16 *bits*. O problema é que se torna impossível incorporar mais *bits* à instrução sem alterar a sua regularidade.

A solução encontrada para a alocação de constantes de 16 *bits* nos registradores é permitir que se possa carregar os 8 *bits* mais significativos de um registrador com a parte alta de uma constante, zerando seus 8 *bits* menos significativos. E em seguida realizar uma adição com outro registrador cujos 8 *bits* menos significativos correspondam a parte baixa da mesma constante.

Ambas as situações utilizam o formato-I da Figura 4.7 e, para diferenciar que se deseja carregar os *bits* mais significativos de um registrador, zerando seus *bits* menos significativos, pode-se criar uma nova instrução, denominada `lui` (*load upper immediate*). Tome-se como exemplo deste processo as instruções a seguir, onde se deseja carregar o valor 50000_{10} (1100001101010000_2) no registrador $\$t_0$:

```
lui 0, 195
lli 1, 80
add 0, 0, $t0, $t1
```

Primeiramente carrega-se o valor 195_{10}, parte alta do valor 50000_{10}, nos *bits* mais significativos do registrador temporário $\$t_0$ (selecionado pelo 2º campo da primeira instrução), zerando ao mesmo tempo seus *bits* menos significativos, através da instrução *lui*. Em seguida, carrega-se o valor 80_{10}, parte baixa do valor 50000_{10}, nos *bits* menos significativos do registrador temporário $\$t_1$ (selecionado pelo 2º campo da segunda instrução) por meio da instrução *lli (load lower immediate)*. Agora, realizando uma adição entre os registradores temporários $\$t_0$ e $\$t_1$, usando formato de representação de instrução da Figura 4.5, consegue-se armazenar o valor

50000_{10} no registrador $\$t_0$, que foi selecionado para armazenar o resultado da operação através do 2º campo da terceira instrução.

A Figura 4.9 ilustra o mesmo processo, utilizando notação em linguagem de máquina, onde é possível observar todos os campos do pacote da instrução com seus valores codificados. Como pode-se perceber, a diferença entre a primeira e a segunda instrução reside apenas no código da operação, por onde o microprocessador consegue distinguir se o programador deseja carregar a parte alta ou a parte baixa de um registrador. Já na ultima instrução, por se tratar de uma operação aritmética, foi usado o formato-R da Figura 4.5.

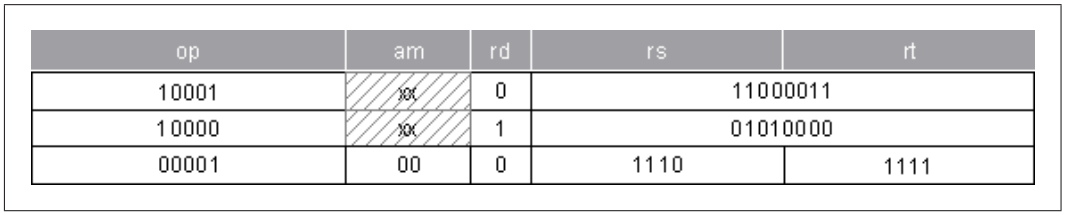

Figura 4.9: Descrição das operações em linguagem de máquina

Agora, o valor máximo que pode ser carregado em registradores passou de apenas 255_{10} ($2^8 - 1$) para 65535_{10} ($2^{16} - 1$), o que expande a aplicabilidade do microprocessador projetado. Apesar da utilização de diferentes formatos de instrução, a proposta inicial de manter o tamanho original da instrução nos 16 *bits* foi preservada.

Contudo, ainda é necessária a definição de um formato de representação para as instruções de acesso à memória. Neste tipo de instrução é necessário informar o endereço de dois registradores: um que armazena o endereço de memória a ser escrito e outro que armazena o dado propriamente dito. Este formato, foi denominado formato-*Load/Store*, e pode ser visualizado na Figura 4.10.

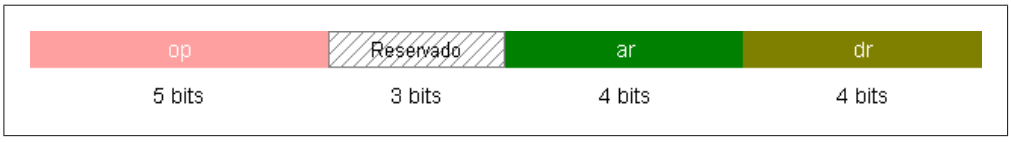

Figura 4.10: Formato-*Load/Store* para instruções de acesso à memória

Os primeiros 5 *bits* especificam o código da operação. Os 3 *bits* seguintes (área hachurada) são ignorados pelo microprocessador. Os próximos 4 *bits* (campo *ar*) especificam qual o endereço do registrador onde se encontra o endereço da célula de memória que deve ser escrita. Os 4 *bits* restantes (campo *dr*) indicam o endereço do registrador onde se encontra o dado a ser escrito no endereço do registrador especificado pelo campo *ar*. Pegando-se como exemplo a instrução a seguir:

```
sw 0, 1
```

A operação `sw` indica que se trata de uma instrução de escrita na memória, portanto, já se sabe que o formato no qual os dados da instrução estão encapsulados é o formato-*Load/Store* da Figura 4.10. O campo seguinte indica que o registrador $\$r_0$ (registrador de código 0000_2) armazena o endereço da célula de memória que deverá ser escrita. O último campo indica que

o registrador $\$r_1$ (registrador de código 0001_2) armazena o dado que será escrito na célula de memória informada por $\$r_0$.

A operação de leitura de dados da memória utiliza também o formato-*Load/Store*. Considerando-se que se deseja carregar o conteúdo de uma célula de memória no registrador $\$r_8$ e, supondo-se que o endereço da célula de memória está armazenada no registrador $\$r_5$, o código na linguagem do AIDA-16 para realizar esta operação seria o seguinte:

```
lw 5, 8
```

A operação `lw` informa que a operação corrente é uma leitura de 16 *bits* na memória, ou seja, utiliza o formato-*Load/Store*. O campo seguinte indica o registrador, neste caso $\$r_5$, no qual está armazenado a célula de memória que se deseja carregar. O campo seguinte informa o registrador de destino dos dados, neste caso o registrador $\$r_8$.

Além disso, o AIDA-16 dispõe de operações para a escrita/leitura de 8 *bits* na memória, por meio das instruções `lb` e `sb`. Estas instruções usam também o formato-*Load/Store*.

Agora, resta a criação de mais um formato para representar as instruções de desvio condicional e incondicional, chamado *formato-Branch*. A Figura 4.11 apresenta este formato:

Figura 4.11: Formato-*Branch* para instruções de desvio

O primeiro campo permanece inalterado em relação aos outros formatos de representação descritos anteriormente e, armazena o código da operação. O outro campo tem 11 *bits* e serve para indicar o número de instruções (ou duplas de células de memória) que devem ser "saltadas" pela operação de desvio. É interessante destacar que este número, por referenciar instruções e não células individuais de memória, deve ser multiplicado por 2 (processo realizado por meio de um circuito deslocador do AIDA-16), pois cada instrução mapeada na memória ocupa duas células consecutivas.

É necessário indicar também o sentido do salto. Dessa forma, é necessário que o valor indicado no campo *d* do formato-*Branch* da Figura 4.11 adote representação numérica em complemento de dois para que seja possível especificar valores com sinal. Dessa maneira, analizando-se o *bit* mais significativo do campo *d* é possível determinar se o valor que se encontra em seus 11 *bits* é positivo ou negativo, e conseqüentemente, o número máximo de células de memória que podem ser abrangidas por um salto é de 2048_{10}, isto é, de -1024 a $+1024$ posições. Deve-se notar que as instruções de desvio condicional são realizadas a partir da verificação de *bits* usados como *flags* nas operações aritméticas realizadas pela ULA.

Na realidade, como o contéudo do campo *d* é o alvo do desvio, baseado em instruções, o valor real em células de memória é obtido através de um deslocamento em uma unidade para a direita se o número for positivo ou para esquerda se for negativo, para representar respectivamente uma multiplicação ou uma divisão por dois. Em seguida, este valor é enviado para um somador dedicado, que realiza sua soma com o conteúdo corrente do registrador $\$pc$. Como o valor do campo *d* da instrução pode conter tanto um valor positivo quanto um valor negativo, a célula de memória alvo do desvio pode estar tanto em um endereço anterior ao da célula atual quanto em um posterior ao da célula atual.

Considerando a seguinte instrução e supondo que a endereço da célula de memória atual é $0000\mathrm{F}_{16}$:

```
jmp 20
```

A instrução `jmp` indica um desvio incondicional e o valor 20_{10} informa ao microprocessador que a próxima célula de memória que deve ser acessada se encontra 40 posições a frente da endereçada atualmente, pois o valor 20_{10} se refere ao número de instruções. Para tanto, é realizado um deslocamento para a direita do valor 20 e, em seguida, é realizada uma operação de adição com o conteúdo do registrador $\$pc$ e o valor devolvido pelo deslocador, fazendo que no próximo ciclo de instrução a célula de memória que será acessada seja a de endereço 00037_{16}.

Tomando-se como exemplo o seguinte trecho de código em uma linguagem de alto nível:

```
i=10
do {
    x=x+y;
    i--;
} while (i>0)
```

Supondo que os valores iniciais das variáveis *i*, *x*, *y* estejam carregados nos registradores $\$t_0$, $\$r_0$ e $\$r_1$ respectivamente. A codificação em linguagem de máquina do AIDA-16 para realizar a seqüência de comandos definidos acima seria:

```
0000F: add 00, 0, $r0, $r1
00010: dec 00, 0, $t0
00011: jnz -2
```

A primeira instrução adiciona os valores das variáveis *x* (registrador $\$r_0$) e *y* (registrador $\$r_1$) e armazena o resultado desta operação em $\$r_0$. Em seguida, a variável *i* (registrador $\$t_0$) é decrementada em 1 unidade. A última instrução testa se o resultado da última operação foi diferente de zero, em caso positivo, o programa é desviado de volta para a célula de endereço $0000\mathrm{F}_{16}$.

Existe ainda a possibilidade de realizar operações específicas, como limpar o conteúdo de um registrador, através da combinação de instruções com formatos diferentes de representação. Por exemplo, considerando-se que se deseja limpar o conteúdo do registrador $r8, poderiam ser usadas as seguintes instruções do AIDA-16:

```
lui 0, 0
mov 14, 8
```

Primeiramente é utilizada a instrução `lui`, que utiliza o formato de representação da Figura 4.7, para carregar os *bits* da parte alta do registrador $\$t_0$ com valor 0_2. Como a instrução `lui` limpa automaticamente a parte baixa do registrador que esta sendo referenciado, o valor de $\$t_0$ é zerado. Em seguida, é necessário copiar o conteúdo de $\$t_0$ para o registrador que deve ser limpo, no caso $\$r_8$. A segunda instrução, que utiliza o formato de representação da Figura 4.10, pode ser usada para resolver este problema.

O exemplo abaixo apresenta Uma maneira alternativa de se efetuar esta mesma tarefa:

```
xor 7, 7
```

A instrução `xor`, compara *bit*-a-*bit* o conteúdo do registrador $\$r_8$ com ele mesmo. Como esta operação lógica retorna resultado zero quando comparados dois valores iguais, o valor de $\$r_8$ ao término de sua execução será zero independentemente de seu conteúdo.

A limpeza do conteúdo de um registrador pode ser útil para vários casos, por exemplo, para garantir que em um carregamento da parte baixa de um registrador a parte alta não guarde "lixo".

Com a utilização dos 4 formatos de instrução descritos anteriormente é possível utilizar todas as instruções do AIDA-16, mantendo o tamanho da instrução inalterado.

4.5 Organização Interna

A Figura 4.12 apresenta um diagrama da arquitetura interna do AIDA-16. As setas espessas representam os barramentos por onde trafegam os dados de/para o banco de GPRs e para a ULA. As setas sólidas representam sinais de 1 *bit* por onde são enviados os dados para o decodificador de instruções e para a unidade de controle. Já as setas pontilhadas representam os sinais de controle enviados pela unidade de controle para comandar todos os eventos realizados pelos blocos funcionais do AIDA-16 e sincronizar todas as operações que são realizadas.

O processo de execução de uma instrução do ponto de vista interno da arquitetura, pode ser descrito como segue. Primeiramente, a unidade de controle envia um sinal que solicita ao barramento externo que busque uma instrução, que se encontra na célula de memória com endereço indicado pelo registrador $\$mar$ (copiado do registrador $\$pc$ através do barramento) e na posição consecutiva[12], armazenando-os no registrador $\$mdr$. Em seguida, o registrador $\$pc$ é incrementado em duas unidades para apontar o endereço que deve ser buscado na próxima instrução de acesso a memória. Assim que o registrador $\$mdr$ é carregado com os 16 *bits* de dados vindos da memória, seu conteúdo é copiado para o registrador $\$ir$. Então, o decodificador de instruções é acionado para verificar se a instrução contida no registrador $\$ir$ faz parte do conjunto de instruções do AIDA-16; em caso positivo, a unidade de controle assume o comando e aciona os módulos funcionais necessários para executar a operação que foi solicitada na instrução.

Por exemplo, para se subtrair 2500_{10} de 1200_{10}, as instruções que devem ser executadas são as seguintes:

```
lui 0, 10110000
lli 1, 00000100
add 0, $t0, $t1
mov 0, $t0, $r0
lui 0, 00000000
lui 1, 00000000
lui 0, 11000100
lli 1, 00001001
add 0, $t0, $t1
sub 0, $r0, $t0
```

Já os eventos internos que acontecem, são um pouco mais complexos. Antes de executar a primeira instrução, o microprocessador deve saber em qual endereço de memória ela está

[12] A cada acesso à memória são buscados os conteúdos de duas células de 8 *bits* por vez

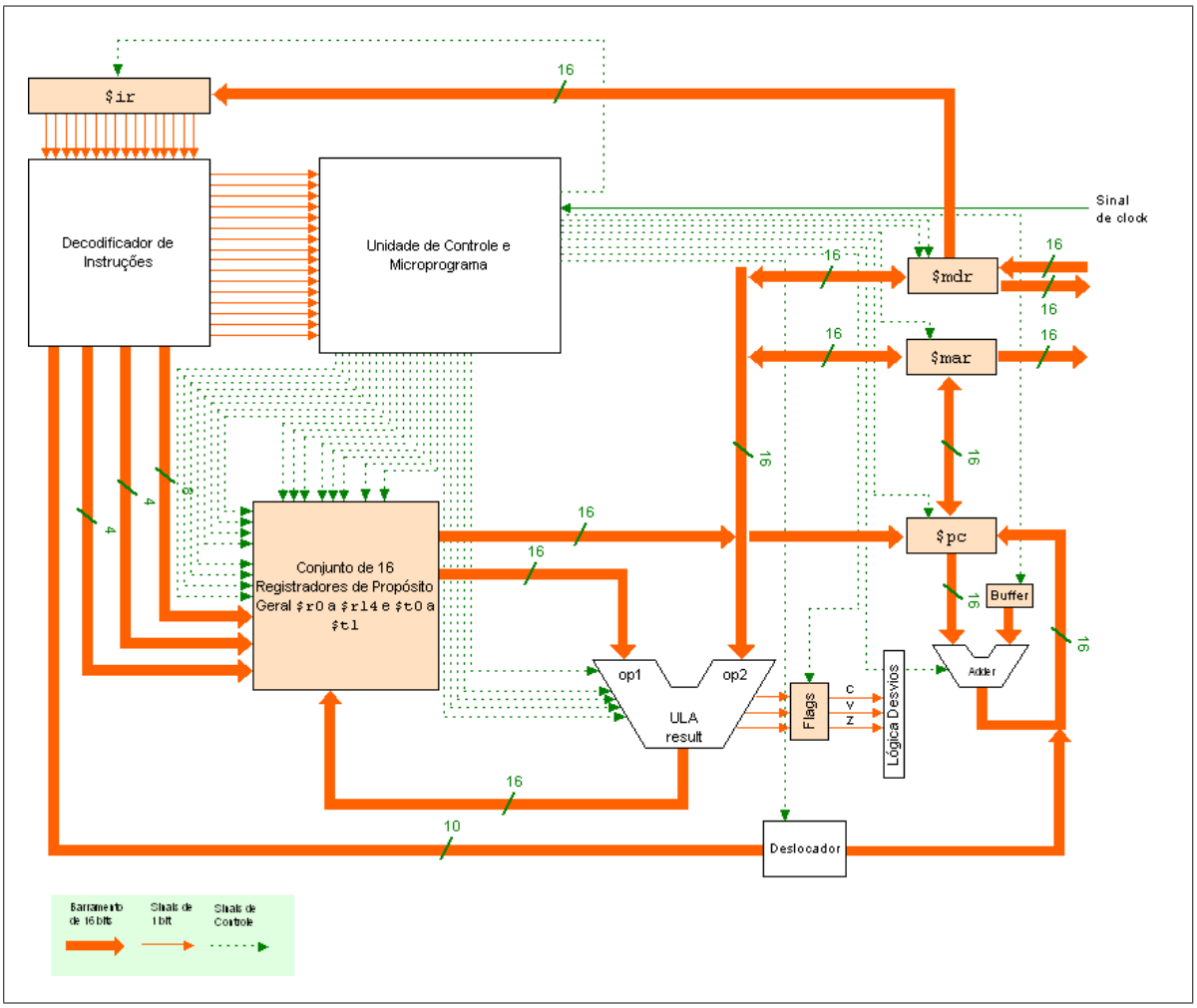

Figura 4.12: Diagrama da arquitetura interna do AIDA-16

posicionada. Para isso, o conteúdo do registrador $\$pc$ é copiado para o registrador $\$mar$ e em seguida é incrementado em duas unidades. Então, a unidade de controle envia um sinal para que os conteúdos da célula de memória indicada no registrador $\$mar$ e da próxima sejam copiados para o registrador $\$mdr$. A seguir, a instrução deve ser decodificada. Para isso, o conteúdo do registrador $\$mdr$ é copiado para o registrador $\$ir$, onde a instrução será manipulada. A instrução neste caso, é o carregamento do valor 176_{10} na parte alta do registrador $\$t_0$. Então, o campo *v* de 8 *bits* do formato-I da Figura 4.7 é copiado para o registrador $\$t_0$.

Agora deve-se buscar a segunda instrução na memória. Como os passos são sempre os mesmos até o pacote da instrução chegar até o decodificador de instruções, os comentários das instruções seguintes serão realizados a partir deste evento. A instrução seguinte é o carregamento do valor 4_{10} na parte baixa do registrador $\$t_1$. A unidade de controle sinaliza ao registrador $\$ir$ para enviar os 8 *bits* do campo *v* pelo barramento até os 8 *bits* menos significativos do registrador $\$t_1$. Com o registrador $\$pc$ incrementado novamente em duas unidades, pode-se buscar a próxima instrução.

A instrução seguinte é uma adição. Os campos *rs* e *rt* indicam os registradores nos quais estão presentes os dados a serem somados. A unidade de controle envia um sinal indicando que estes registradores devem liberar seus conteúdos no barramento que chega até a ULA. O código

da operação de soma 0001_2 é colocado no canal da ULA que seleciona a operação desejada, e o sinal de *clock* é enviado para que a operação seja executada e o resultado da operação é colocado no registrador $\$t_0$.

A Figura 4.13 apresenta as formas de onda de alguns módulos internos do AIDA-16 durante a execução de uma operação de carga da constante 1792_{10} no registrador $\$t_1$.

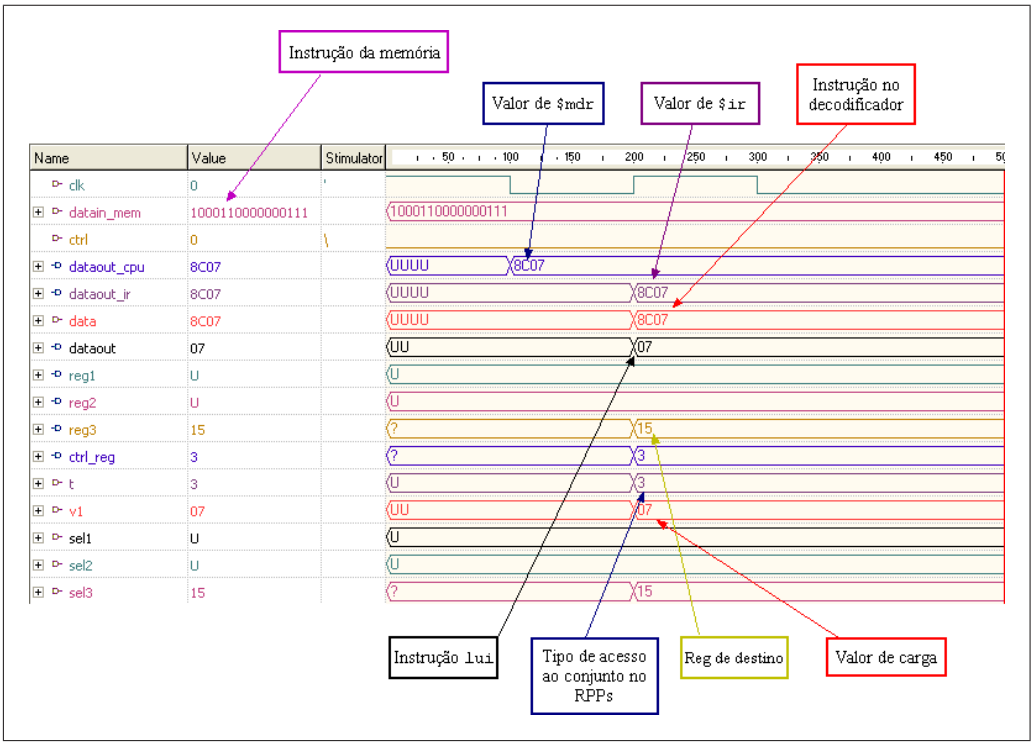

Figura 4.13: Formas de onda decorrentes da execução de uma instrução de carga

O sinal de $datain\_mem$ indica a instrução recebida no canal de entrada da memória. No instante seguinte, é possível observar que este sinal é transferido para o registrador $\$mdr$, indicado em $dataout\_cpu$. Em seguida, o canal $dataout\_ir$ recebe este mesmo sinal, vindo do registrador $\$mdr$. O canal $data$ é a entrada para o decodificador de instruções e a unidade de controle do AIDA-16. Após a decodificação do sinal recebido pelo decodificador de instruções e determinação de que se trata de um carregamento de constante na parte alta de um registrador (5 primeiros *bits* do sinal), a unidade de controle gera os sinais indicados em $crtl\_reg$, que informa ao conjunto de GPRs o tipo de acesso a ser realizado, o sinal reg_3, que indica o registrador de destino dos dados, ou da operação (registrador identificado pelo valor 15, ou seja, $\$t_1$)e o sinal $data\_out$, que indica neste caso a constante a ser carregada (valor 7_{10}).

A Figura 4.14 mostra o estado do conjunto de GPRs após a execução da instrução descrita acima.

O conjunto indicado por set_1 indica os registradores presentes no AIDA-16. Logo abaixo, é possível verificar que o conteúdo da parte alta do registrador $\$t_1$ ($set_1(15)$) é igual ao valor 7_{10} e que a sua parte baixa foi limpa.

A Figura 4.15 apresenta a representação em formas de onda do comportamento do circuito do AIDA-16 em uma operação de adição:

É interessante ressaltar que os registradores contendo os operandos envolvidos na operação, $\$t_0$ e $\$t_1$, foram previamente carregados com os valores 15_{10} e 15_{22} respectivamente.

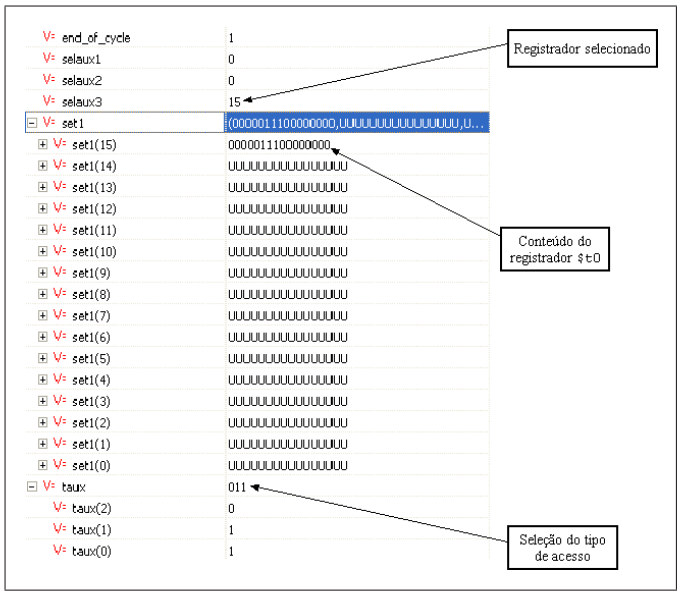

Figura 4.14: Estado do conjunto de GPRs após a execução de uma instrução de carga

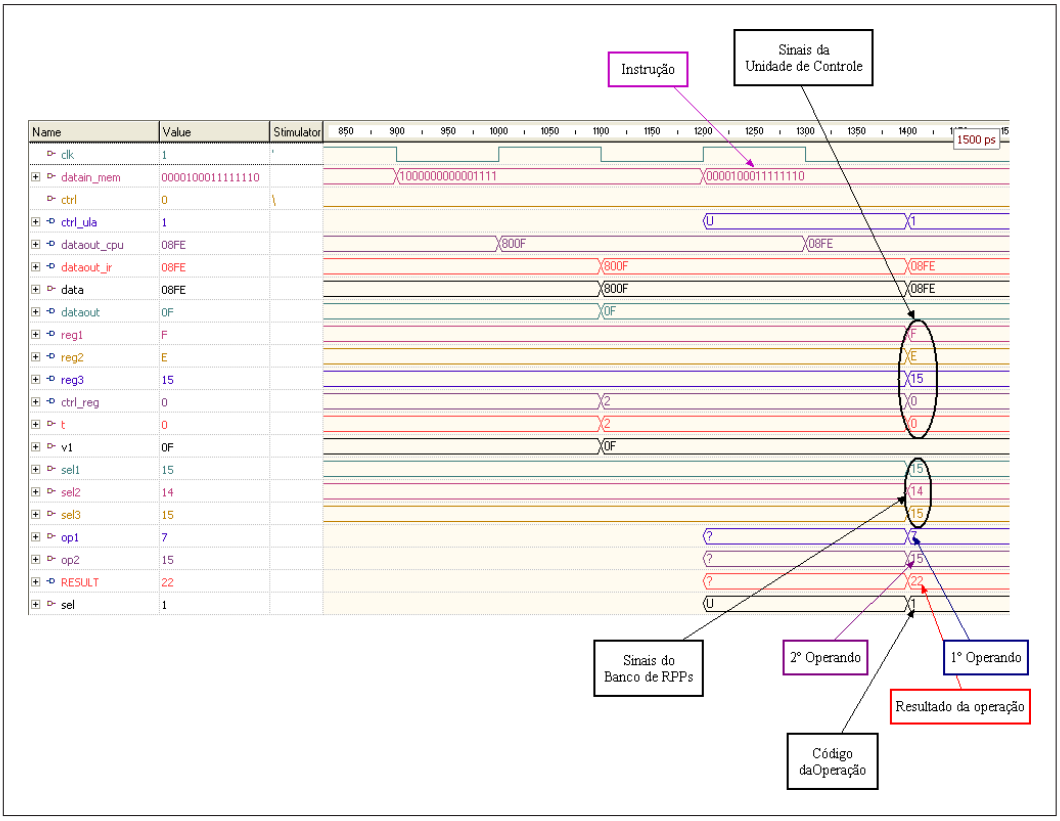

Figura 4.15: Formas de onda decorrentes da execução de uma instrução aritmética

O resultado da operação será em seguida enviado de volta para o banco de GPRs no registrador indicado na instrução.

4.6 Interface com a Memória

O canal de comunicação com a memória externa se dá através dos registradores $\$mar$ e $\$mdr$. Como o microprocessador somente opera dados presentes em registradores, todo e qualquer dado deve ser carregado em um registrador interno para que possa ser manipulado.

O registrador $\$mar$ é unidirecional, trabalha no sentido microprocessador/memória, indicando qual o endereço de memória deverá ser acessado. Por meio deste registrador consegue-se estabelecer o número de células de memória que podem ser referenciadas. Como este registrador é de 16 *bits*, pode-se acessar 65536_{10} células de memória.

Já registrador $\$mdr$ é bidirecional, e é utilizado para armazenar o dado que se deseja escrever na memória ou o dado que se deseja trazer da memória. As células de memória possuem 8 *bits* cada uma, de maneira que, devem ser copiadas em pares para o registrador $\$mdr$ que possui 16 *bits*. No caso de uma escrita em memória, o conteúdo do registrador $\$mdr$ é desmembrado em duas partes, destinadas a um par de células consecutivas da memória.

Por exemplo, para escrever um valor na célula de memória identificada pelo endereço $2EE0_{16}$, é preciso colocar o valor binário correspondente a este endereço no registrador $\$mar$ e em seguida enviar um sinal de controle para que o registrador $\$mdr$ copie os 8 *bits* mais significativos na célula identificada pelo endereço especificado pelo registrador $\$mar$ e os seus 8 *bits* menos significativos na célula de memória seguinte.

4.7 Execução de Instruções

O ciclo de execução de instruções do AIDA-16, pode ser resumido conforme o esquema da Figura 4.16.

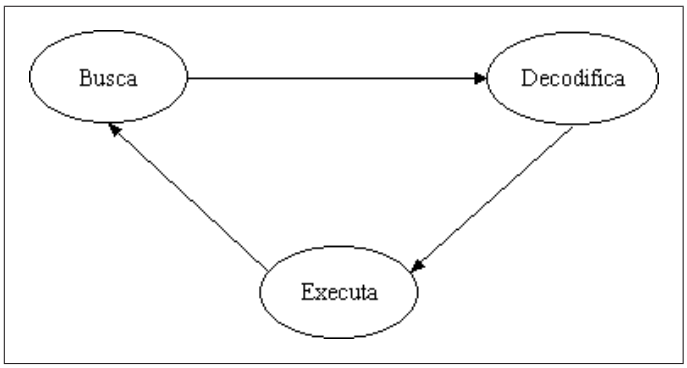

Figura 4.16: Ciclo de execução resumido do AIDA-16

Na primeira etapa, deve-se buscar a instrução a ser executada na memória principal. Na segunda etapa, a instrução deve ser decodificada para determinar se ela faz parte do conjunto de instruções presente no microprograma do AIDA-16. Em seguida, a instrução é executada através do acionamento dos blocos funcionais envolvidos na operação em questão.

Este ciclo de execução de instruções pode ser detalhado através dos seguintes passos:

1. Buscar a próxima instrução na posição de memória indicada pelo registrador $\$pc$, por meio dos registradores de acesso à memória $\$mar$ e $\$mdr$, carregando-a em seguida no registrador $\$ir$.

2. Incrementar o valor do registrador $\$pc$ em duas unidades para que no próximo ciclo, se não houver uma instrução de desvio, a célula de memória acessada seja a que armazena a instrução consecutiva do programa que está sendo executado.

3. Decodificar a instrução recebida para determinar o tipo de operação que se deseja executar, para que a unidade de controle consiga identificar quais são os blocos funcionais que devem ser acionados através dos sinais de controle.

4. Buscar os dados envolvidos na operação (se existirem) no banco de GPRs através do acesso ao(s) registrador(es) correto(s).

5. Executar a instrução através do envio dos dados aos blocos funcionais responsáveis pela sua manipulação.

6. Armazenar o resultado da operação no registrador apropriado ou em uma determinada posição de memória.

7. Voltar ao primeiro passo para iniciar a execução da próxima instrução.

Claro que estes passos não são válidos para todas as instruções. Em uma instrução de carregamento de registrador, por exemplo, no item 4 o dado é escrito no registrador apropriado e a execução da instrução pode ser encerrada pois não existem mais tarefas a cumprir. Entretanto, a maioria das instruções segue o processo de execução descrito acima.

4.8 Decodificador de Instruções

O decodificador de instruções é responsável pela decodificação do código de operação, fornecido pela instrução de máquina, para os sinais de controle necessários à operação dos blocos funcionais internos do microprocessador.

No processo de decodificação de instruções o bloco decodificador recebe do registrador $\$ir$ um pacote de dados 16 *bits*, que deve estar de acordo com o um dos 4 formatos de representação de instrução definidos na Seção 4.4. Como independente do formato utilizado, todas as instruções possuem em seus 5 primeiros *bits* o código referente à instrução que deve ser executada, o primeiro passo então, é determinar se este código realmente existe dentro do microcódigo do AIDA-16. Em caso positivo, o decodificador de instruções já consegue realizar a associação do código de operação corrente com o formato de representação no qual a instrução está encapsulada e, conseqüentemente, informar à unidade de controle quais serão os sinais de controle que devem ser acionados para que a instrução seja executada.

4.9 Unidade de Controle

A Unidade de Controle é o módulo que comanda o funcionamento e sincronização entre os módulos do AIDA-16, tendo como lógica base microinstruções binárias. Estas microinstruções têm como função acionar cada módulo através de um vetor de *bits* de controle, fazendo com que todos os módulos se comuniquem em um determinado tempo a fim de executar a instrução selecionada.

Tomando-se, por exemplo, o seguinte trecho de código em linguagem de alto nível:

```
a=50;
b=25;
a=a+b;
```

A codificação de instruções no AIDA-16 para executar os códigos acima seria:

```
lli  0, 50
lli  1, 25
add 0, 00, $t0, $t1
```

Do ponto de vista da unidade de controle, os sinais de controle necessários para realizar a execução das instruções acima listadas estão descritos como segue. Para primeira instrução, enviar os sinais de controle para o banco de GPRs que indicam o desejo de realizar o carregamento do valor 50_{10} no registrador de destino dos dados, no caso o registrador $\$t_0$. Em seguida o banco de GPRs envia um sinal de retorno à unidade de controle informando que o ciclo de execução encerrou. Desta forma, é necessário que a próxima instrução seja buscada na memória. Isso acontece através do envio de sinais de controle para que os registradores $\$tmar$ e $\$mdr$ efetuem o acesso à memória trazendo as células de memória que correspondem a seqüência de execução do programa. Para a segunda instrução, os sinais que a unidade de controle deve enviar são basicamente os mesmos da primeira, com exceção de que o valor de carregamento agora é 25_{10} e o registrador de destino é $\$t_0$. Na terceira instrução, a unidade de controle deve acionar o banco de GPRs, informando que se deseja realizar um acesso direto aos conteúdos dos registradores $\$t_0$ e $\$t_1$ simultaneamente e que estes valores devem ser direcionados à entrada da ULA. Em seguida, devem ser enviados os sinais de controle para a ULA que selecionam a operação de adição. No final da execução da operação, deve ser enviado um sinal de volta para o banco de GPRs onde é informado que se deseja escrever o resultado da operação no registrador $\$t_0$.

4.10 Unidade Lógica e Aritmética

A ULA do AIDA-16 foi projetada para realizar operações lógicas e aritméticas em operandos de 16 *bits* e suportar a execução direta de instruções complexas presentes em microprocessadores CISC atuais, tais como a multiplicação e a divisão. Em microprocessadores RISC, as operações complexas são executadas por meio do encadeamento de várias instruções simples, porém, o código de máquina gerado para obter o resultado de uma operação deste tipo torna-se bem mais extenso. Portanto, para deixar a codificação em linguagem de máquina mais enxuta, facilitando o trabalho do compilador, e agregar algumas características dos microprocessadores CISC à plataforma desenvolvida, foi optado pela implementação em *hardware* destas operações.

A Figura 4.17 apresenta o modelo estrutural da ULA.

A ULA é habilitada pela unidade de controle através do *Sinal de Clock*. Quando é detectado um nível lógico alto neste pino, a ULA lê o barramento de saída do banco de GPRs, onde se encontram o primeiro e o segundo operandos, e efetua a operação indicada no canal de *Seleção da Operação*. O resultado da operação é encaminhado para o canal *Resultado* e os *Flags de Sinalização* são enviados para os registradores de estado (utilizados pelas instruções de desvio condicional).

A Descrição 4 apresenta o código VHDL que representa a interface do circuito da ULA:

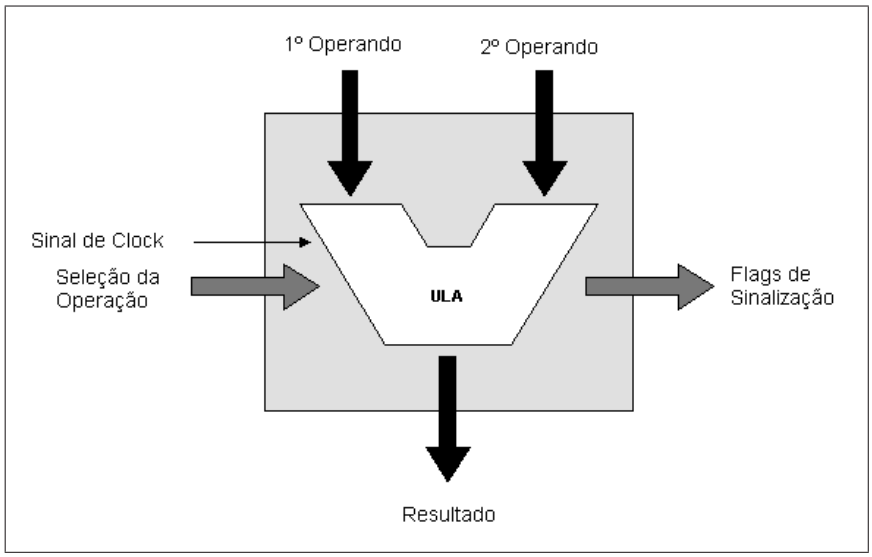

Figura 4.17: Modelo estrutural da ULA

Entre as linhas `02` a `05` é realizada a declaração da constante *BusWidth*, que define o tamanho do barramento de dados da ULA, e da constante *SecondaryBus* que define um barramento interno utilizado pela operação de divisão. Entre as linhas `07` e `10` são definidos os canais de entrada da ULA. Os canais de saída estão indicados entre as linhas `12` e `15`.

A seguir são listados cada um dos canais e sinais de E/S da ULA do AIDA-16 com as suas respectivas descrições:

Canais de Entrada:

- *op1*: Primeiro operando (16 *bits*)
- *op2*: Segundo operando (16 *bits*)
- *sel*: Canal de seleção da operação desejada (4 *bits*)
- *clk*: Sinal para habilitar/desabilitar o funcionamento da ULA (1 *bit*)

Canais de Saída:

- *result*: Resultado da operação selecionada (16 *bits*)
- *c*: Sinal que indica a ocorrência de *carry* na operação selecionada (1 *bit*)
- *z*: Sinal que indica se o resultado da operação é igual a zero (1 *bit*)
- *o*: Sinal que indica a ocorrência de *overflow* na operação selecionada (1 *bit*)

4.10.1 Códigos de Operação

A ULA utiliza 4 *bits* para representação das operações que podem ser efetuadas, de maneira que podem ser expressas 16 operações diferentes ($2^4 = 16$). A Tabela 4.3 lista todas as operações da ULA associadas aos seus respectivos códigos de operação:

Descrição 4 Unidade Lógica e Aritmética - Descrição da entidade

```
01: entity ula is
02:       generic (
03:            BusWidth: Integer := 15;
04:            SecondaryBus: Integer := 31;
05:       );
06:       port (
07:          op1: in std_logic_vector (BusWidth downto 0);
08:          op2: in std_logic_vector (BusWidth downto 0);
09:          sel: in std_logic_vector (3 downto 0);
10:          clk: in std_logic_vector;
11:
12:          result: out std_logic_vector (BusWidth downto 0);
13:          c: out std_logic;
14:          z: out std_logic;
15:          o: out std_logic;
16:       );
17: end ula;
```

Tabela 4.3: Códigos de operação da ULA

Categoria	Código	Instrução	Descrição
NA	0000	nop	nenhuma operação
Aritméticas	0001	add	adição entre operandos
	0010	sub	subtração entre operandos
	0011	inc	incrementa o primeiro operando
	0100	dec	decrementa o primeiro operando
	0101	mul	multiplicação entre operandos[1]
	0110	div	divisão entre operandos
	0111	mod	resto da divisão entre operandos
Lógicas	1000	shr	desloca primeiro operando 1 *bit* para a direita
	1001	shl	desloca primeiro operando 1 *bit* para a esquerda
	1010	ror	rotaciona primeiro operando 1 *bit* para a direita
	1011	rol	rotaciona primeiro operando 1 *bit* para a esquerda
	1100	and	and lógico entre operandos
	1101	or	or lógico entre operandos
	1110	xor	xor lógico entre operandos
	1111	not	complementa primeiro operando

[1] Somente para operandos de até 8 *bits*

4.11 Estatísticas da Síntese e de Design

A seguir serão apresentados os resultados[13] da síntese e design do AIDA-16 no FPGA Spartan II XC2S100 da Xilinx.

A Figura 4.18 (a) apresenta o resultado do processo de conversão do código VHDL do AIDA-16 em uma *netlist* sem otimização do circuito. Já a Figura 4.18 (b) mostra o mesmo re-

[13] Os resultados foram obtidos a partir do *software* WebPACK do fabricante Xilinx.

sultado após a otimização do circuito. Os resultados refletem a quantidade de componentes lógicos (latches, multiplexadores, somadores/subtratores, comparadores e portas XOR) que foram necessários para representar o circuito do AIDA-16.

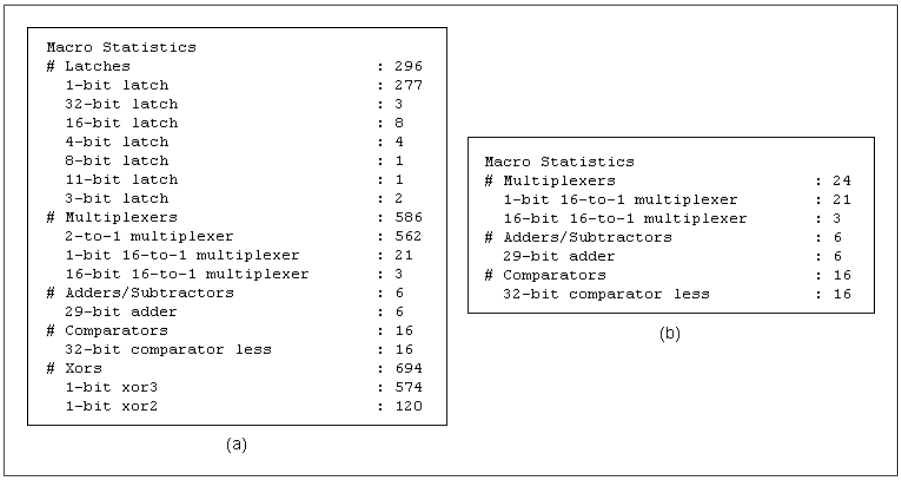

Figura 4.18: Estatísticas da Síntese - Netlist

Na Figura 4.19 (a) são apresentados os recursos do FPGA que foram utilizados para implementação da lógica do AIDA-16. É possível observar a quantidade de recursos dos CLBs, blocos de roteamento e ainda os canais de E/S consumidos. Já na Figura 4.19 (b) são mostrados os valores percentuais de ocupação e de roteamento no FPGA e ainda as estatísticas temporais.

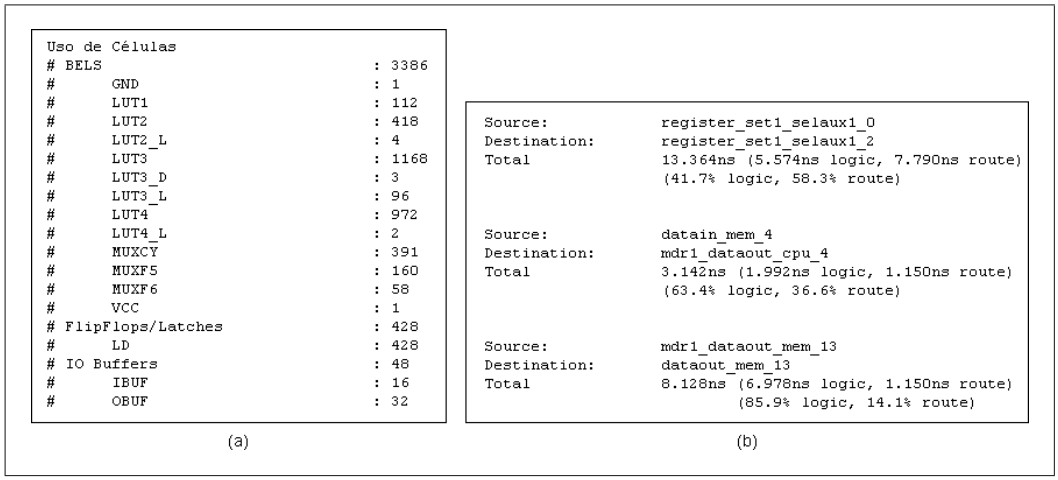

Figura 4.19: Estatísticas da Síntese - Design

Capítulo 5

Conclusões

O presente trabalho propôs o projeto e prototipação em *hardware* de um microprocessador para ser usado como uma plataforma base para o ensino de disciplinas como Arquitetura de Computadores e Sistemas Digitais e proporcionar uma base experimental para o desenvolvimento de projetos de outras áreas de ensino no ramo da computação. O microprocessador, chamado de AIDA-16, foi descrito em VHDL e sintetizado em um dispositivo FPGA.

Foram apresentadas as diferentes classificações das arquiteturas microprocessadas que estão em uso atualmente, suas características e diferenças do ponto de vista arquitetural. O esquema funcional de um microprocessador genérico foi discutido, com o intuito de esclarecer os aspectos comuns a todos os tipos de arquiteturas microprocessadas. Foram abordados os diferentes tipos de tecnologias programáveis, com enfoque especial nos dispositivos FPGAs, que tiveram sua estrutura e funcionamento estudados. Finalmente, foi descrita a especificação do AIDA-16 e o processo de concepção de seus módulos internos juntamente com os equipamentos e *softwares* utilizados.

A primeira e principal contribuição do presente trabalho se refere à criação de um método alternativo para o ensino de disciplinas da área SDAC, através do desenvolvimento de uma arquitetura para propiciar aos estudantes um estudo aprofundado na maneira como os circuitos digitais funcionam. Como conseqüência, é possível a segunda contribuição, que provê para as demais disciplinas do curso de Ciência da Computação uma base para o desenvolvimento de experimentos e aplicações. A terceira contribuição se refere à consolidação do uso de dispositivos programáveis e HDLs no âmbito do curso, encorajando o desenvolvimento de outros projetos nesta área.

Foi constatado que dos modos de endereçamento criados para o AIDA-16, o único que mostrou grande aplicabilidade foi o modo direto. Os modos indireto e imediato, por se mostrarem pouco utilizados nesta plataforma, poderiam sofrer algumas modificações ou até mesmo serem removidos da arquitetura. Uma alternativa para aumentar a usabilidade destes modos sem efetuar sua remoção, seria reorganizar o formato de instrução do AIDA-16, de maneira a proporcionar que os campos destinados aos operadores no modo imediato fossem maiores em termos de *bits*, já para o modo indireto, um aumento no número de GPRs estenderia seria o suficiente para estender seu uso e, permitir que o uso de registradores como ponteiros pudesse ser mais explorado.

Outra conclusão a que se chega é que as HDLs aliadas os dispositivos programáveis podem ser de grande valia para tornar o processo de aprendizado de circuitos digitais mais rápido e eficiente, podendo até mesmo servir para a implementação de protótipos para serem testados

de maneira a se obter melhores resultados no desenvolvimento definitivo de qualquer sistema digital.

5.1 Trabalhos Futuros

A seguir serão descritas algumas sugestões para implementações de módulos adicionais para o AIDA-16.

- No formato de instrução para as instruções de acesso à memória seria possível utilizar um dos *bits* reservados para indicar se a operação será realizada em modo *big-endian*[14] ou *little-endian*[15]. Isso permitiria fazer com facilidade operações de inversão de *string* na memória.

- Criação de uma unidade de ponto flutuante (co-processador matemático). A utilização co-processadores matemáticos já é adotada a bastante tempo nos microprocessadores comerciais, e serve para otimizar e execução das operações de ponto flutuante.

- Descrição de um bloco de memória *cache* para aumentar o desempenho na execução de aplicações através da redução de acessos à memória externa. A memória *cache* armazenaria algumas células subseqüentes à que está sendo executada em um dado momento, fazendo com que as próximas instruções a serem executadas não precisassem ser buscadas na memória (se não houver alguma instrução de desvio que altere o andamento seqüencial da execução do programa).

- Adição de *pipeline* de instruções para reduzir a ociosidade dos módulos funcionais que não estão participando do estágio de execução de uma determinada instrução. Dessa maneira, várias instruções podem ser executadas paralelamente no microprocessador, cada uma em um estágio diferente de execução (desde que não haja uma relação de dependência entre estas instruções).

- Aumento do número de GPRs, para aumentar a flexibilidade na manipulação e armazenamento de informações. Tais registradores não poderiam ser endereçados utilizando os formatos de instrução que foram definidos, mas seriam acessados através do modo de endereçamento indireto, usando um dos 16 GPRs endereçáveis como ponteiro.

- Suporte a execução especulativa de instruções para lidar com o problema gerado pelas instruções de desvio condicional. Em um desvio condicional com duas ramificações por exemplo, o microprocessador se adianta e carrega na memória *cache* uma das ramificações alvo do desvio. Dessa forma, a chance de acerto fica em 50%, e em caso de erro, apenas se deixa de ganhar desempenho.

Um trabalho interessante seria o desenvolvimento de um compilador e um montador para a arquitetura AIDA-16 para disponibilizar o suporte a execução de programas escritos pelo programador por meio de uma linguagem de alto nível. Consequentemente poderia-se escrever um sistema operacional específico para o AIDA-16 para oferecer suporte a execução de aplicações

[14]Modo de organização onde os *bytes* são numerados da esquerda para a direita

[15]Modo de organização onde os *bytes* são numerados da direita para a esquerda

desenvolvidas para o mesmo. Poderia-se ter então uma arquitetura computacional completa a ser usada especificamente para o ensino, fornecendo uma base para os conceitos abordados por muitas disciplinas do curso de Ciência da Computação.

O desenvolvimento de uma aplicação didática por meio de uma linguagem de alto nível, para possibilitar a observação dos estados correntes dos canais de comunicação e dos registradores do AIDA-16 durante a execução das instruções, seria também um trabalho fundamental, e poderia tornar o processo de aprendizado ainda mais intuitivo.

Outra aplicação para a plataforma AIDA-16 seria a sua implantação em um sistema embarcado, realizando o processamento cooperativo de instruções com um ou mais GPPs.

Referências Bibliográficas

[ALD04] ALDEC, Inc. *Active-HDL* http://www.aldec.com/ActiveHDL/. Acessada em 14 de junho de 2004.

[ARN92] ARNOLD, Jeffrey M. *The Splash 2*. In: Symposium on Parallel Algorithms and Architectures, p.316-324, 1992.

[AKC00] AKÇAL, Elif; UZSOY, Reha; HISCOCK, David G.; MOSER, Anne L.; TEYNER, Timothy J. *Alternative Loading and Dispatching Policies for Furnace Operations in Semiconductor Manufacturing: A Comparison by Simulation.* Proceedings of the 32nd conference on Winter simulation, Orlando, Florida, 2000, pp. 1428-1435.

[ATH93] ATHANAS , Peter M.; SILVERMAN, Harvey F. *Processor Reconfiguration Through Instruction-Set Metamorphosis.* IEEE Computer Society, Vol. 26, No. 3, 1993, pp. 11-18.

[BAR00] BARAT, Francisco; LAUWEREINS Rudy. *Reconfigurable Instruction Set Processors: A Survey* . 11^{th} IEEE International Workshop on Rapid System Prototyping (RSP 2000), Paris, FRANCE, 2000.

[BRO96] BROWN, Stephen; ROSE Jonathan. *FPGA and CPLD Architectures: A Tutorial.* IEEE Design & Test of Computers, 1996, pp. 42-57.

[CAL01] CALAZANS, Ney Laert Vilar; MORAES, Fernando Gehm. *Integrating the Teaching of Computer Organization and Architecture with Digital Hardware Design Early in Undergraduate Courses*. IEEE Transactions on Education, v. 44, n. 2, p. 109-119, 2001.

[CAL03] CALAZANS, Ney; MORENO, Edson; HESSEL, Fabiano; ROSA, Vitor; MORAES, Fernando; CARARA, Everton. *From VHDL Register Transfer Level to SystemC Transaction Level Modeling: A Comparative Case Study*. 16^{th} Symposium on Integrated Circuits and Systems Design (SBCCI'03), São Paulo, Brazil, 2003.

[CAP01] CAPPELATTI, Ewerton A. *Implementação do Barramento PCI para Interação Hardware/Software em Dispositivos Reconfiguráveis.* Dissertação de mestrado apresentada ao DCC-PUC, 2001.

[CON96] CONTE, T. M.; BANERJIA, S.; LARIN, S.Y.; MENEZES, K.N.; SATHAYE, S.W. *Instruction Fetch Mechanisms for VLIW Architectures With Compressed*

Encodings. 29^{th} Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29), Paris, FRANCE, 1996.

[DEH00] DEHON, A. *The Density Advantage of Configurable Computing.* IEEE Computer, pp. 41-49, Abril 2000.

[GAJ97] GAJSKI, Daniel D. *Principles of Digital Design.* New Jersey: Prentice Hall, 1997.

[HAU97] HAUSER, John R. Hauser; WAWRYZYNEK, John. *Garp: A MIPS Processor With a Reconfigurable Coprocessor.* In: 5^{th} IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97). Napa Valley, Califórnia, 1997, p. 12.

[HUA97] HUANG, Tsai Chi; et al. *The Teaching of VHDL in Computer Architecture.* Proceedings 1997 IEEE International Conference on Microelectronic SystemsEducation, Arlington, Virginia, USA (1997), pp 133-134.

[INT04] INTEL Corporation. *Intel Education: Learning About Technology: How Chips Are Made*. Disponível em http://www.intel.com/education/makingchips/shock.htm. Acessada em 01 de Junho de 2004.

[KAH99] KAHNG, A. B.; PATI, Y. C. *Subwavelength Optical Lithography: Challenges and Impact on Physical Design.* Proceedings of the 1999 international symposium on Physical design, Monterey, California, United States, 1999, pp. 112-119.

[KNA03] KNAPP, S. *Using Programmable Logic to Accelerate DSP Functions* Xilinx Corporation. San Jose, Califórnia, EUA. 1998. Disponível em http://www.xilinx.com/appnotes/dspintro.pdf. Acessada em 23 de setembro de 2003.

[MEI95] MEIER, Russell D. *Rapid Prototyping of a RISC Architecture for Implementation in FPGAs.* In: IEEE Symposium on FPGA's for Custom Computing Machines (FCCM '95). Napa Valley, California, 1995, p. 0190.

[MEN96] MENDONÇA, Alexandre; ZELENOVSKY, Ricardo. *PC e Periféricos - Um Guia Completo de Programação.* Rio de Janeiro: Editora Ciência Moderna Ltda, 1996.

[MES02] MESQUITA, Daniel G. *Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs.* Dissertação de mestrado apresentada ao DCC-PUC, 2002.

[MON96] MONTEIRO, Mário A. *Introdução à Organização de Computadores.* Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora S.A. Terceira Edição, 1996.

[MOO95] MOON, Soo-Mook; CARSON, Scott D. *Generalized Multiway Branch Unit for VLIW Microprocessors.* IEEE Transactions on Parallel and Distributed Systems, Vol. 6, nº 8, 1995, pp. 850-862.

[ORD03] ORDONEZ, Edward David Moreno & PEREIRA, Fábio Dacêncio & PENTEADO, Cesar Giacomini & PERICINI, Rodrigo de Almeida. *Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs).* São Paulo: Bless Gráfica e Editora Ltda, 2003.

[PAT00] PATTERSON, David A. & HENNESSY, John L. *Organização e projeto de computadores: A Interface Hardware/Software.* Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora S.A. Segunda Edição, 2000.

[PAT81] PATTERSON, David A; SEQUIN Carlo H. *RISC I: A Reduced Instruction Set VLSI Computer.* Proceedings of the 8^{th} annual symposium on Computer Architecture, Minneapolis, Minnesota, United States, 1981. pp. 443-457.

[RAM96] RAMANADIN, B.; POGODALLA. *CO: The Chameleon 64-bit Microprocessor ASIC Prototype.* In: 7^{th} IEEE International Workshop on Rapid System Prototyping (RSP '96). Thessaloniki, GREECE, 1996, P.140.

[RAU93] RAU, B.; FISHER J. *Instruction-Level Parallel Processing: History, Overview, and Perspective.* J. of Supercomputing, Special Issue on Instruction-Level Parallelism, vol. 7, no. l/2, pp. 9-50, 1993.

[RIB02] RIBEIRO, Alexandre A. de L. *Reconfigurabilidade Dinâmica e Remota de FPGAs.* Dissertação de mestrado apresentada ao ICMC-USP, 2002

[REI02] REIS, Ricardo Augusto da Luz. *Concepção de Circuitos Integrados.* Porto Alegre: Editora Sagra Luzzatto. Segunda Edição, 2002.

[ROS03] ROSE, César A. F. de; NAVAUX, Philippe O. A. *Arquiteturas Paralelas.* Porto Alegre: Editora Sagra Luzzatto. Primeira Edição, 2003.

[ROS93] ROSE, Jonathan; GAMMAL, Abbas El & SANGIOVANNI-VINCENTELLI, Alberto.*Architecture of Field-Programmable Gate Arrays.* Proceedings of the IEEE, v.81, n.7, pp.1013-104, July 1993.

[RUS91] RUSSEL, David W.; Haden Kirtley B. *A Configurable, Virtual Microprocessor System for Instructional Use in Real-Time, Real-World Studies.* IEEE Micro, 1991, pp. 26-29.

[SAS01] SASSATELLI, Gilles; CAMBON, Gaston; GALY, Jerome; TORRES, Lionel. *The Systolic Ring: A Dynamically Reconfigurable Architecture for Embedded Systems.* In: PROCEEDINGS OF THE FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS WORKSHOP, 2001, Belfast, Irlanda do Norte.

[SHA86] SHAHDAD, Moe. *An Overview of VHDL Language and Technology.* Proceedings of the 23rd ACM/IEEE conference on Design automation, 1986.

[SHE95] N. Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 1995, 538p.

[SIL93] SILVEIRA, Reinaldo. *Unidades Lógicas e Aritméticas de Alto Desempenho: Uma Experiência Utilizando Técnicas Alternativas.* Dissertação de mestrado apresentada ao DEE-UPS, 1993.

[SRI97] SRIVASTAVA, S.; et al. *Evolution of Architectural Concepts and Design Methods of Microprocessors.* In: Eleventh International Conference on VLSI Design VLSI for Signal Processing. Índia, 1997. p. 312.

[TAN99] TANENBAUM, Andrew S. *Structured Computer Organization*. Rio de Janeiro: Editora Prentice-Hall do Brasil Ltda. Quarta Edição, 1999.

[TAU84] TAUB, Herbert. *Circuitos Digitais e Microprocessadores*. Rio de Janeiro: Editora McGraw-Hill do Brasil, Ltda, 1984.

[TOR01] TOROK, Delfim Luiz. *Projeto Visando a Prototipação do Protocolo de Acesso ao Meio em Redes Ethernet*. Dissertação de mestrado apresentada ao DCC-PUC, 2001.

[UYE02] UYEMURA, John P. *Sistemas Digitais - Uma Abordagem Integrada*. São Paulo: Editora Thomson, 2002.

[VHD90] ASHENDEN, Peter J. *The VHDL Cookbook*. Dept. Computer Science University of Adelaide South Australia, 1990.

[XES04] XESS Corporation. *What Are CPLDs and FPGAs?* Disponível em http://www.xess.com/fpgatut.htm. Acessada em 30 de abril de 2004.

[XES04a] XESS Corporation. *XSA Board V1.1, V1.2 User Manual*. Disponível em http://www.xess.com/manuals/xsa-manual-v1_2.pdf. Acessada em 14 de junho de 2004.

[XES04b] XESS Corporation. *XStend Board V2.1 Manual*. Disponível em http://www.xess.com/manuals/xst-manual-v2_1_0.pdf. Acessada em 14 de junho de 2004.

[XES04c] XESS Corporation. *Introduction to WebPACK 6.1 (XSA Board version)*. Disponível em http://www.xess.com/appnotes/webpack-6_1-xsa.pdf. Acessada em 14 de junho de 2004.

[XIL03] XILINX Inc. *Virtex series configuration architecture user guide*. Disponível em http://www.xilinx.com/xapp/xapp151.pdf. Acessada em 4 de Setembro de 2003.

[XIL04] XILINX Inc. *Spartan-II 2.5V FPGA Family: Introduction and Ordering Information*. Disponível em http://direct.xilinx.com/bvdocs/publications/ds001_1.pdf. Acessada em 14 de junho de 2004.

[WEB00] WEBER, Raul Fernando. *Fundamentos de Arquitetura de Computadores*. Porto Alegre: Editora Sagra Luzzatto. Primeira Edição, 2000.

Apêndices

Apêndice A

Descrição das Operações da ULA

A seguir serão listadas as operações da ULA do AIDA-16 com uma detalhada descrição sobre o processo de implementação utilizado.

A.1 Adição

Das operações aritméticas mais importantes, deve-se destacar a operação de soma, pois além de ter um índice de utilização alto, participa indiretamente da execução de outras operações, como subtração, multiplicação, comparação, etc. [SIL93].

Existem inúmeros tipos de somadores propostos. A seguir são apresentados alguns dos métodos clássicos de implementação de somadores com suas funções de complexidade de tempo correspondentes, onde *n* representa o número de *bits* do somador.

- Somador *Ripple-carry*: $O\ (n)$
- Somador *Carry-skip* (*Manchester/bypass*): $O\ (n\ 1/a)$ onde a corresponde ao número de *skip layers*.
- Somador *Carry-predict* (*Lookahead*): $O\ (log_n)$.
- Somador *Carry-select* (*Conditional Sum*): $O\ (log_n)$.

Quase todos os somadores são baseados em blocos ou elementos funcionais chamados de somadores completos. Estes somadores são circuitos compostos de três entradas (A, B e Cin) e duas saídas (S e $Cout$), relacionados segundo a Tabela A.1:

As colunas tituladas A, B e Cin da Tabela A.1 representam respectivamente um *bits* do primeiro operando, um *bits* do primeiro operando e o canal de entrada para o "vai-um". Já as colunas tituladas por S e $Cout$ representam respectivamente o resultado da adição com seu e o "vai-um" da operação.

A Figura A.1 representa o esquema de portas lógicas necessário para implementação de um circuito somador completo.

Qualquer somador binário pode ser implementado como uma conexão serial de somadores completos, arranjados de forma que o *carry* de saída de cada somador completo alimente o *carry* de entrada do próximo somador completo mais de valor posicional mais significativo [GAJ97]. Por exemplo, um somador de *n bits* consiste em *n* somadores completos conectados de acordo com a Figura A.2. No geral, qualquer somador de *n bits* pode ser construído desta maneira.

Tabela A.1: Tabela-verdade de um somador completo

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

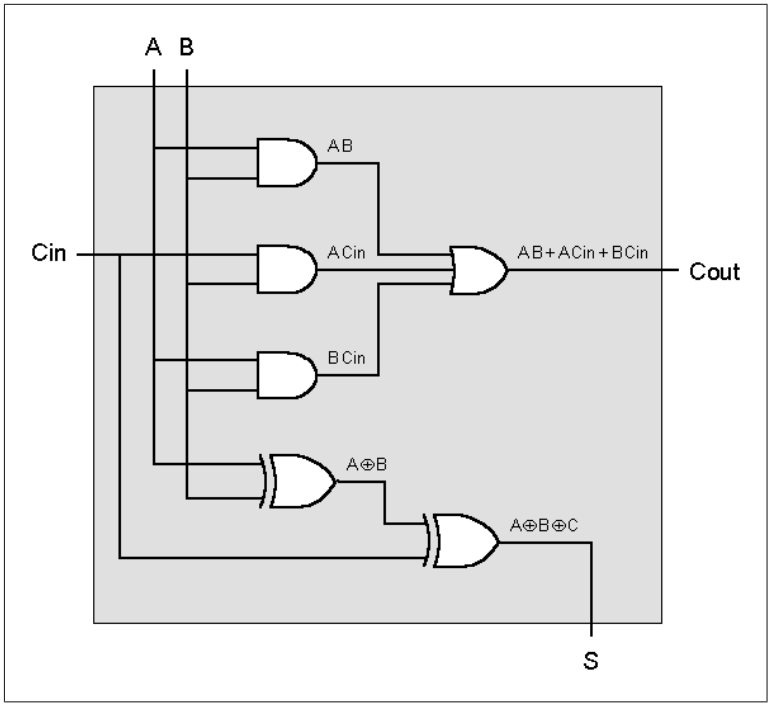

Figura A.1: Diagrama de portas lógicas de um somador completo

Como se pode ver, o atraso de operação mais longo em tal somador seria da entrada Cin_0, ou dos *bits* menos significativos A_0 e B_0, para o *carry* de saída ($Cout_n$). Em outras palavras, qualquer mudança em Cin, A_0 ou B_0 se propaga através de todos somadores completos.

A implementação do somador contido na ULA segue a metodologia *Ripple-carry*, que consiste em conectar *n* somadores completos (Figura A.2), onde *n* corresponde ao número de *bits* dos operandos.

Cada somador completo fica responsável por efetuar a soma entre dois *bits* de mesmo valor posicional da cadeia de *bits* dos operandos. O somador mais a esquerda realiza a operação sobre os *bits* menos significativos dos operandos (A_0 e B_0), sendo que o sinal de *carry* conecta os somadores entre si e propaga o "vai-um" do resultado da soma anterior para o somador de ordem imediatamente superior. Desta forma, nenhum elemento de ordem *i* pode ser calculado sem que elementos de ordem inferior tenham sido calculados.

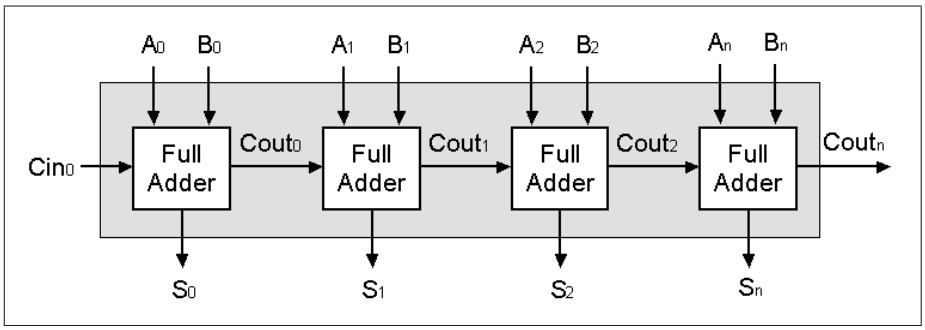

Figura A.2: Esquema de um somador *ripple-carry* de *n bits*

A.2 Subtração

A subtração nada mais é do que uma soma em complemento, ou seja, $A - B = A + (-B)$. Uma regra prática para se achar o complemento de um número binário é inverter o valor dos dígitos, ou seja, trocar todos os dígitos "1" por "0", e todos os dígitos "0" por "1", e, ao final, somar 1_{10} ao número obtido anteriormente.

A Figura A.3 mostra o circuito utilizado para a construção de um subtrator de *n bits*.

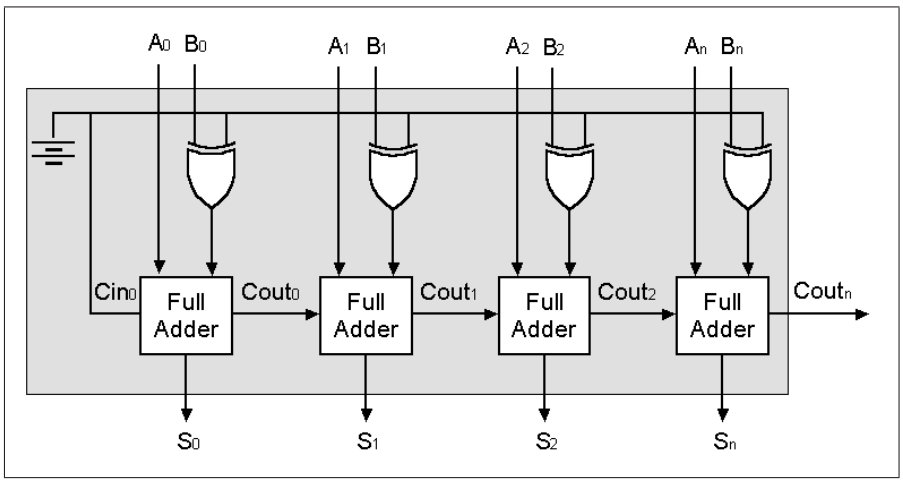

Figura A.3: Esquema de um somador-subtrator de *n bits*

Do ponto de vista do circuito físico, um subtrator de *n bits* nada mais é que um somador *n* de *bits*, onde cada uma dos *bits* do segundo operando esta conectado a um dos pinos de uma porta *xor* de duas entradas e o outro pino, juntamente com o sinal de *carry* do somador do *bit* menos significativo estão recebendo uma tensão que represente um sinal lógico alto.

A.3 Multiplicação

Em qualquer sistema de numeração, a multiplicação é uma forma resumida da soma repetitiva de uma parcela consigo mesma (o multiplicando) um determinado número de vezes (dado pelo multiplicando) [WEB00]. Entretanto, essa soma repetitiva torna o processo da

multiplicação extremamente lento para números grandes, de forma que, foi desenvolvido um método otimizado para a multiplicação decimal.

A idéia base usada na multiplicação de números binários é mesma usada para números decimais com a utilização deste método. Deve-se multiplicar cada dígito do multiplicador pelo multiplicando para formar um produto parcial, e então, realiza-se a soma dos produtos parciais para se obter o produto final. Isto está ilustrado na parte esquerda da Figura A.4. Geralmente, se produzem todos os produtos parciais e então se efetua a soma de todos eles juntos; todavia, isto não é necessário e, no lugar disso, na medida em que cada produto parcial é produzido, ele pode ser somado com o anterior. Uma vez que os somadores utilizados em computadores só conseguem somar dois números ao mesmo tempo, a soma dos produtos parciais deve ser efetuada da maneira anteriormente citada. A multiplicação binária de números positivos pode ser realizada usando esta mesma idéia, porém, de forma mais simples, uma vez que cada produto parcial ou é igual à zero ou é igual ao multiplicando propriamente alinhado com seu respectivo multiplicando. Isto está ilustrado na parte direita da Figura A.4 para um multiplicando e um multiplicador de 4 *bits*.

```
   341  -> Multiplicando          0110  -> Multiplicando
 x 273  -> Multiplicador        x 1101  -> Multiplicador
 01023  -> Produto Parcial 1   0000110  -> Produto Parcial 1
 16870  -> Produto Parcial 2   0000000  -> Produto Parcial 2
+68200  -> Produto Parcial 3   0011000  -> Produto Parcial 3
 86093  -> Produto Final       0110000  -> Produto Parcial 4
                              1001110  -> Produto Final
```

Figura A.4: Exemplo das multiplicações decimal e binária

Como pode ser visto, o número de dígitos dos produtos (parciais e final) é consideravelmente maior que o número de dígitos do multiplicando ou do multiplicador. De forma que, o comprimento da multiplicação de um multiplicando de *n bits* por um multiplicador de *m bits* tem $n + m$ *bits* [PAT00].

A formação dos produtos parciais na multiplicação binária é uma operação trivial, uma vez que os valores possíveis dos dígitos do multiplicador são "0" e "1". Em outras palavras, cada produto parcial ou é igual ao multiplicando ou é uma string de zeros [GAJ97]. Uma vez que uma grande porção dos dígitos do multiplicador é igual a zero (geralmente metade), é ineficiente realizar a multiplicação sobre todos seus dígitos.

A Figura A.5 apresenta o fluxograma que demonstra o processo da multiplicação binária para operandos de 4 *bits*:

No primeiro passo, o produto é deslocado 1 *bit* para a esquerda e a variável denominada *carry*, recebe o *bit* menos significativo do multiplicador. Em seguida, a variável *carry* é testada. Se estiver setada, o passo 2, que consiste realizar uma adição entre o produto e o multiplicando, deve ser realizado e em seguida o passo 3. Senão, o passo 3, que consiste em deslocar o multiplicador 1 *bit* para a esquerda, deve ser executado sem passar pelo passo 2. Para finalizar, é realizado um teste para verificar se o processo está na quarta iteração; em caso positivo se encerra o processo, caso negativo volta-se para o passo 1.

A Descrição 5 apresenta o módulo principal da multiplicação binária do AIDA-16:

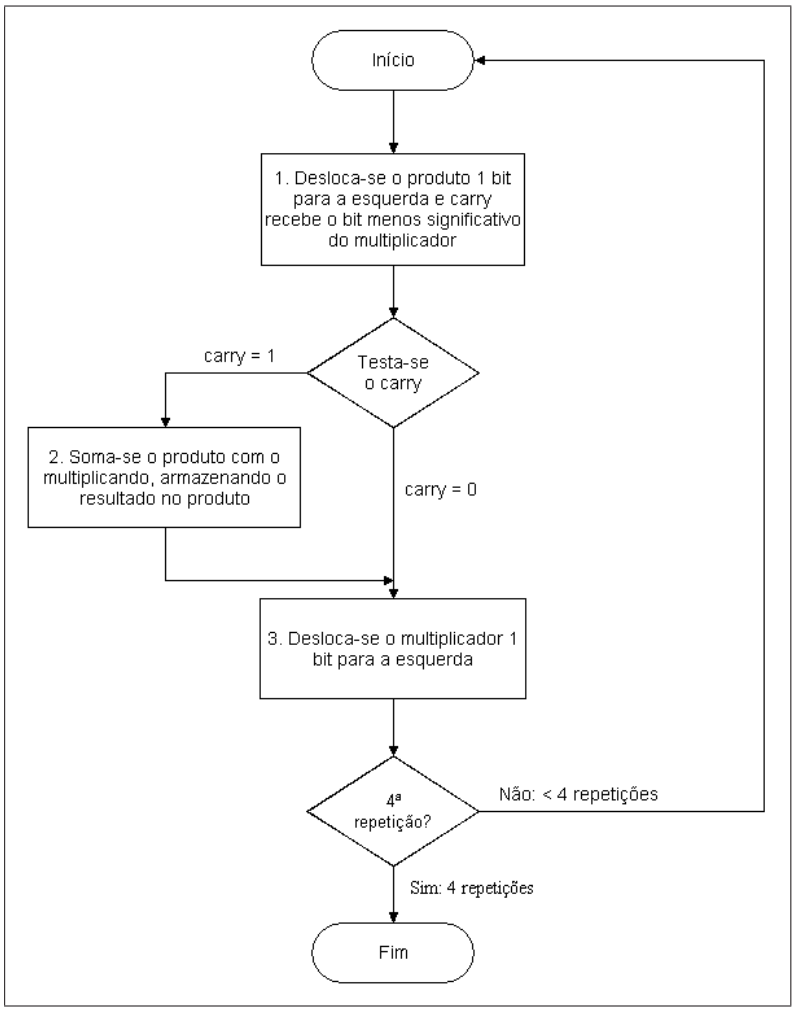

Figura A.5: Fluxograma de uma multiplicação binária de 4 *bits*

Na linha `06` o primeiro valor passado como parâmetro para a função é copiado para a variável *multiplicando*. Na linha `07` o segundo parâmetro é copiado para a variável *multiplicador*. Na linha `08` a variável *produto*, que receberá o resultado da multiplicação, é zerada. Na linha `09` é disparado um *loop* que executará 7 vezes o trecho de código da linha `10` até a linha `15`. A linha `10` envia o variável produto para a função *shiftl*, que desloca o parâmetro recebido 1 *bit* para a esquerda. Em seguida, na linha `11` a variável *carry* recebe o *bit* mais significativo da variável *multiplicador*. A linha `12` testa a variável *carry*. Se a variável *carry* estiver setada, a linha `13`, que envia o as variáveis *produto* e *multiplicando* para a função *add16bits* (responsável por realizar a adição dos seus parâmetros). Caso a variável *carry* não esteja setada, a linha `15` é executada e, envia a variável *multiplicador* para ser deslocada pela função *shiftl*.

A.4 Divisão e Resto da Divisão

A divisão binária usa o método de deslocamento e subtração. Neste método, primeiro subtrai-se o maior múltiplo possível do divisor para determinar o primeiro dígito do quociente.

Descrição 5 Multiplicação Binária - Módulo principal

```
01: function "*" (x,y : std_logic_vector) return std_logic_vector is
02: variable multiplicando, multiplicador, produto: std_logic_vector (BusWidth downto 0);
03: variable carry: std_logic;
04: begin
05:     multiplicando:=x;
06:     multiplicador:=y;
07:     produto:="0000000000000000";
08:     for i in 7 downto 0 loop;
09:         produto:=shiftl(produto);
10:         carry:=multiplicador(7);
11:         if carry='1' then
12:             produto:=add16bits(produto,multiplicando);
13:         end if;
14:         multiplicador:=shiftl(multiplicador);
15:     end loop;
16:     return produto;
17: end "*";
```

Então, efetua-se este processo novamente com o dividendo diminuído e deslocamos o divisor [GAJ97].

Um divisor de *n bits* requer $n + 1$ ciclos de computação (laços de programa) para ser completado. Além disso, um divisor de *n bits* e um dividendo de *m bits* precisarão no máximo *m bits* para expressar o quociente e *n bits* para expressar o resto.

O exemplo a seguir ilustra o procedimento detalhado de uma divisão binária de 4 *bits*. Dividir 0111_2 por 0010_2 (quer dizer, dividir 7_{10} por 2_{10}).

1. Usa-se um par de registradores de 8 *bits*, escreve-se o dividendo na parte baixa do primeiro registrador, o divisor na parte alta do segundo registrador, e coloca-se o quociente em um terceiro registrador também de 8 *bits*.

2. Desloca-se o divisor um *bit* para a direita e o dividendo um *bit* para a esquerda.

3. Compara-se o dividendo e o divisor.

4. Se o dividendo é menor que o divisor, coloca-se "0" no *bit* menos significativo do quociente.

5. Senão o *bit* menos significativo do quociente recebe "1" e subtrai-se o divisor do dividendo.

6. Repete-se os passos 2 e 3 mais quatro vezes.

O fluxograma da Figura A.6 ilustra os passos listados acima:

O primeiro passo consiste em se deslocar o divisor 1 *bit* para a direita e o quociente 1 *bit* para a esquerda. Em seguida o dividendo e o divisor são comparados. Se o dividendo for maior ou igual ao divisor, é executado o passo $2a$, que consiste em setar o *bit* menos significativo do quociente e subtrair o dividendo do divisor. Caso o dividendo for menor que o divisor, é executado o passo $2b$, que consiste em zerar o *bit* menos significativo do quociente. Em seguida basta verificar se o processo já se encontra na quinta iteração; em caso positivo o processo é encerrado, em caso negativo, caso negativo volta-se para o passo 1.

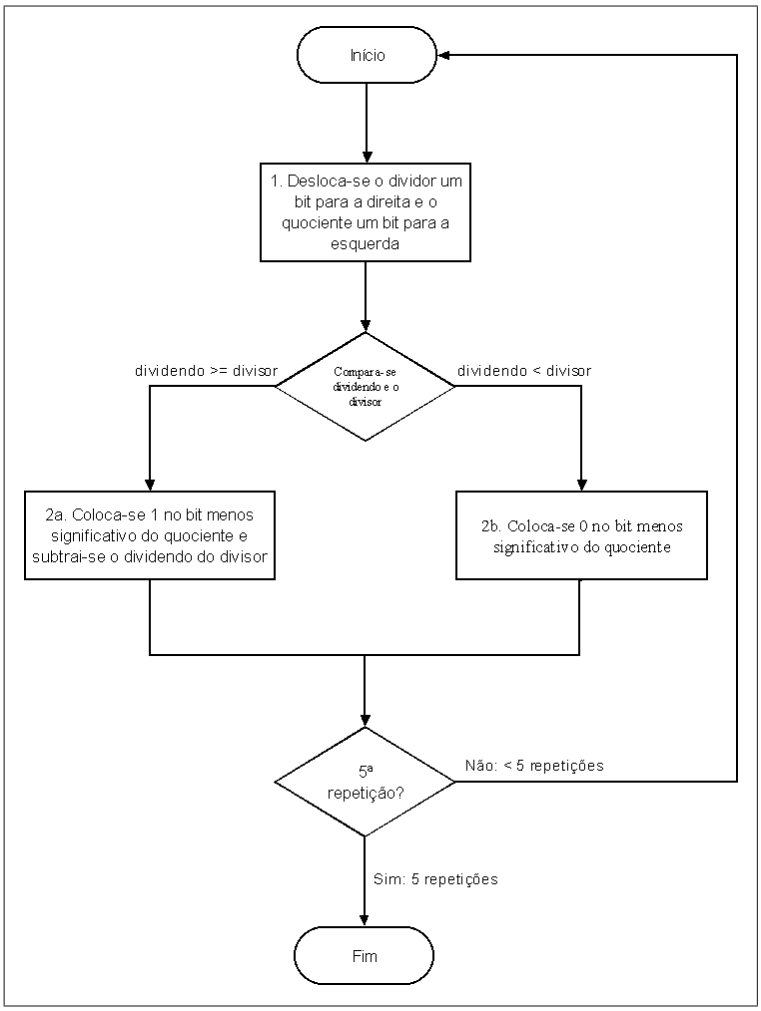

Figura A.6: Fluxograma de uma divisão binária de 4 *bits*

A Descrição 6 apresenta o módulo principal da divisão binária do AIDA-16:

Na linha `06` é disparado um *loop* que percorre todos os *bits* das variáveis *dividendo* e *divisor* colocando o primeiro parâmetro nos *bits* menos significativos da variável dividendo e o segundo parâmetro nos *bits* mais significativos da variável divisor. Em seguida, na linha `13` é disparado outro *loop* que executa 16 vezes (o número de *bits* dos operandos) o conteúdo linhas `14` a `21`. Na linha `14` o conteúdo da variável *divisor* é enviado para a função *shiftr*, que desloca o parâmetro de entrada 1 *bit* para a direita. Na linha `15` o conteúdo da variável *quociente* é enviado para a função *shiftl*, desloca o parâmetro de entrada 1 *bit* para a esquerda. Na linha `16` os conteúdos das variáveis *dividendo* e *divisor* são enviados para a função *comp*, que compara os seus parâmetros de entrada retornando "1" se o primeiro parâmetro for maior ou igual ao segundo e "0" caso contrário. Se o retorno da função textit comp for diferente de zero, a linha `17`, que zera o *bit* menos significativo da variável *quociente*, é executado; senão, as linhas `19`, que seta o *bit* menos significativo da variável *quociente* e `20`, que envia os conteúdos das variáveis *dividendo* e *divisor* para a função *sub16bits* (que subtrai o primeiro parâmetro do segundo) são executadas.

Tendo a operação de divisão implementada, descrita anteriormente, obter o seu resto torna-se muito simples, uma vez que a variável *dividendo* da Descrição 6 ao final da execução da

Descrição 6 Divisão Binária - Módulo principal

```
01: function div(x,y: std_logic_vector) return std_logic_vector is
02: variable dividendo, divisor: std_logic_vector (15 downto 0);
03: variable quociente: std_logic_vector (BusWidth downto 0);
04: variable resto: std_logic_vector (BusWidth downto 0);
05: begin
06:     for i in BusWidth downto 0 loop
07:         dividendo(i):=x(i);
08:         dividendo(i):=x(i);
09:         dividendo(i+8):='0';
10:         divisor(i+8):=y(i);
11:         divisor(i):='0';
12:     end loop;
13:     for i in 0 to BusWidth loop
14:         divisor:=shiftr(divisor);
15:         quociente:=shiftl(quociente);
16:         if comp(dividendo,divisor) '1' then
17:             quociente(0):='0';
18:         else
19:             quociente(0):='1';
20:             dividendo:=sub16bits(dividendo,divisor);
21:         end if;
22:     end loop;
23:         return quociente;
24: end div;
```

operação de divisão, contém o valor parcial que não pôde ser dividido o pelo divisor, ou seja, o próprio resto da divisão. Dessa maneira, basta inserir na Descrição 6 um parâmetro que permita selecionar o tipo de retorno que o usuário deseja para que a operação de resto da divisão possa ser utilizada.

A.5 Incremento e Decremento

As operações de incremento e decremento para um operando de *n bits* podem ser representadas respectivamente pelos circuitos da Figura A.2 e da Figura A.3, sendo que, em ambos os circuitos, o pino *Cin* do somador mais a esquerda deve estar conectado a uma tensão que represente o valor "1" e os pinos destinados ao segundo operando conectados a uma tensão que represente o valor "0". Como resultado desta modificação no circuito somador, qualquer operando enviado para o circuito de incremento será incrementado em 1 unidade e o operando enviado para o circuito de decremento será decrementado em 1 unidade.

A.6 Deslocamento para Direita e Esquerda

As operações de deslocamento são realizadas no primeiro operando de entrada da ULA e consistem basicamente em movimentar a cadeia de *bits* do operando uma casa para a direita ou para a esquerda. No deslocamento para a esquerda, o primeiro *bit* da cadeia de *bits* do operando e movido uma casa adiante, sendo que isto ocorre em todos os *bits* da cadeia e o último operando é descartado. Para a posição que fica vaga no primeiro bit do operando é inserido o valor "0". No

deslocamento à esquerda o procedimento é o inverso, os *bits* são deslocados no sentido inverso e o primeiro *bit* e descartado. O valor "0" é colocado no último *bit* da cadeia.

A.7 Rotacionamento para Direita e Esquerda

As operações de rotacionamento são quase idênticas às operações de deslocamento. Na rotação para a direita, a diferença está no fato de que o último *bit* da cadeia, que era desprezado no deslocamento, é colocado no lugar liberado pelo primeiro *bit* da cadeia. Na rotação para a esquerda, acontece a mesma situação, desta forma, o primeiro *bit* da cadeia, que era desprezado no deslocamento, é colocado na posição liberada pelo último *bit* da cadeia.

A.8 AND Lógico

A Figura A.7 (a) representa o diagrama lógico de uma porta AND. A Figura A.7 (b) é a forma de representação da equação de uma porta AND. A Figura A.7(c) é a tabela-verdade de uma porta AND, representando seu comportamento dadas as diferentes combinações de entrada possíveis. A porta AND combina dois ou mais sinais de entrada de forma equivalente a um circuito em série, para produzir um único sinal de saída. Em uma porta AND, se um dos sinais de entrada for zero a saída será também zero; só se têm uma saída positiva em caso de todos os sinais de entrada serem positivos.

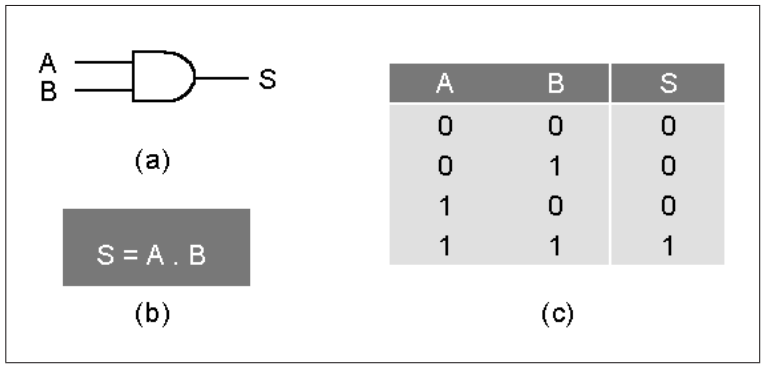

Figura A.7: Esquemas de representação de uma porta AND

A.9 OR Lógico

A Figura A.8 (a) representa o diagrama lógico de uma porta OR. A Figura A.8 (b) é a forma de representação da equação de uma porta OR. A Figura A.8 (c) é a tabela-verdade de uma porta OR, representando seu comportamento dadas as diferentes combinações de entrada possíveis. A porta OR combina dois sinais de entrada de forma equivalente a um circuito em paralelo, para produzir um único sinal de saída. Pode-se dizer que. O funcionamento de uma porta OR é o inverso ao de uma porta AND, já que sua saída somente será nula quando todos os sinais de entrada também forem.

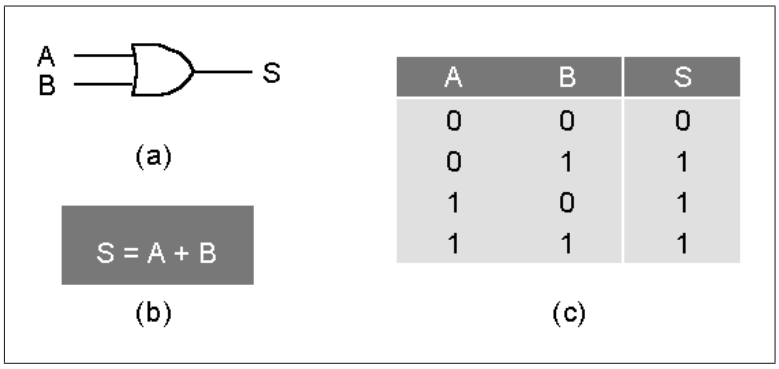

Figura A.8: Esquemas de representação de uma porta OR

A.10 XOR Lógico

A Figura A.9 (a) representa o diagrama lógico de uma porta XOR. A Figura A.9 (b) é a forma de representação da equação de uma porta XOR. A Figura A.9 (c) é a tabela-verdade de uma porta XOR, representando seu comportamento, dadas as diferentes combinações de entrada possíveis. A porta XOR é constantemente utilizada para realizar a comparação de *bits*, uma vez ela acusa uma saída positiva sempre que os *bits* de entrada forem diferentes.

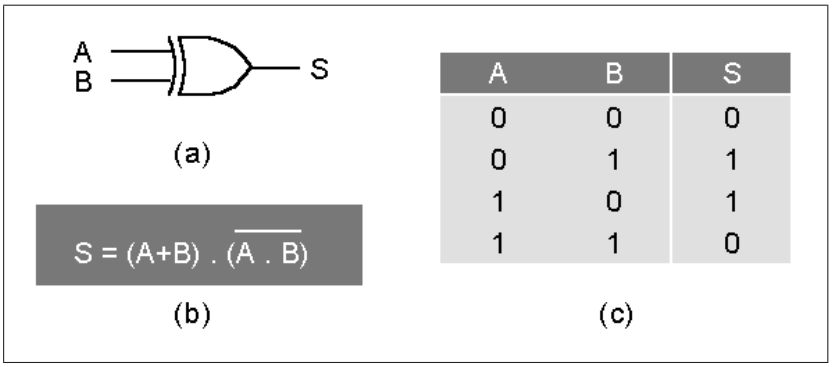

Figura A.9: Esquemas de representação de uma porta XOR

A.11 NOT (Complemento)

A Figura A.10 (a) representa o diagrama lógico de uma porta NOT. A Figura A.10 (b) é a forma de representação da equação de uma porta XOR. A Figura A.10 (c) é a tabela-verdade de uma porta XOR, representando seu comportamento dadas as diferentes combinações de entrada possíveis. A porta XOR é chamada também de inversor, pois a sua saída será sempre equivalente ao inverso de sua entrada.

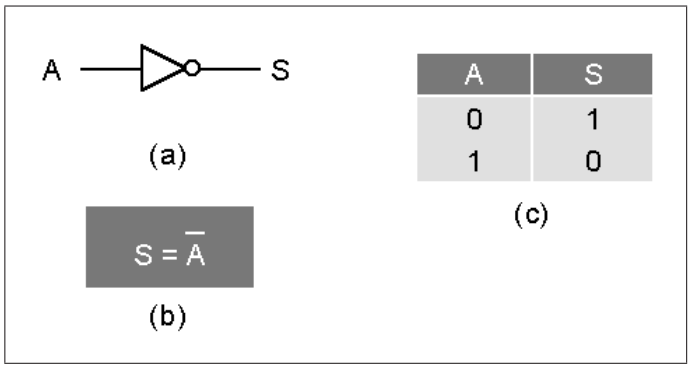

Figura A.10: Esquemas de representação de um inversor

A.12 Flags de Sinalização

Os *flags* de sinalização são representados por registradores de 1 *bit*, de forma que, em um dado momento podem conter apenas o valor "0" ou o valor "1". Eles são utilizados como indicadores para as instruções de desvio condicional, tendo por base o resultado das operações realizadas pela ULA. Existem três qualificadores de estado (*flags*) na arquitetura desenvolvida. Estes qualificadores são: *flag* de *overflow* ou de transbordo, *flag* zero e *flag* de carry.

A.12.1 Overflow

O *flag* de *overflow* indica que houve um estouro na capacidade de representação numérica do resultado de uma operação, ou seja, não se pode expressar o valor da operação com a quantidade de *bits* posicionados nos canais de saída do circuito. Por exemplo, supondo que uma instrução de adição entre o valor 255_{10} e o valor 1_{10} seja disparada em um circuito onde se tem apenas 8 *bits* destinados ao resultado da operação, haverá um estouro de representação, pois o valor máximo que se pode expressar com 8 *bits* é 255_{10}. O *flag* de *overflow* é utilizado pelas instruções `jv` e `jnv`.

A.12.2 Zero

O *flag* de zero, como o próprio nome já sugere, é utilizado para indicar que o resultado de uma operação foi igual a zero. Este flag é utilizado pelas instruções `jz` e `jnz`, utilizadas respectivamente para realizar um desvio quando o *flag* de zero estiver setado e não estiver setado. Um exemplo do uso deste *flag*, é quando existe a necessidade de realizar um desvio se a última operação realizada pela ULA resultou em zero.

A.12.3 Carry

O *flag* de *carry* é usado para indicar o estouro da capacidade de representação de uma casa numérica, o conhecido "vai-um". O *flag* de *carry* também pode indicar a ocorrência de um *overflow*, se o *bit* da casa numérica que sofreu o estouro for o mais significativo. O flag de *carry* é usado pelas instruções `jc` e `jnc`.

Apêndice B

Descrição VHDL do AIDA-16

```
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 17 de junho de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Registrador de Instrucao ($ir)

--------------------------------------------------------------------------------
-- Bibliotecas
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Pinagem do circuito
--------------------------------------------------------------------------------
entity ir is
    port (
        -- canais de entrada
        data_ir: in STD_LOGIC_VECTOR (15 downto 0);    -- canal de entrada de dados
        clk: in STD_LOGIC;                             -- sinal de clock

        -- canais de saida
        dataout_ir: out STD_LOGIC_VECTOR (15 downto 0) -- canal de saida de dados
    );
end ir;

--------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------
architecture ir of ir is
begin
    process (data_ir)   -- processo sensivel ao canal data_ir
    begin
        if clk='1' then          -- quando o sinal de clock estiver habilitado
            dataout_ir<=data_ir; -- canal de saida recebe o canal de entrada
        end    if;
    end process;
end ir;


-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 18 de junho de 2004
```

```
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Registrador de Enderecos da Memoria  ($mar)

--------------------------------------------------------------------------------
-- Bibliotecas
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Pinagem do circuito
--------------------------------------------------------------------------------
entity mar is
    port (
        -- canais de entrada
        data_regset: in STD_LOGIC_VECTOR (15 downto 0);  -- canal de entrada de dados 1
        data_pc: in STD_LOGIC_VECTOR (15 downto 0);      -- canal de entrada de dados 2
        clk: in STD_LOGIC;                               -- sinal de clock
        ctrl: in STD_LOGIC;                              -- sinal de controle

        -- canais de saida
        dataout: out STD_LOGIC_VECTOR (15 downto 0)      -- canal de saida de dados
    );
end mar;

--------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------
architecture mar of mar is
begin
    process (data_regset, data_pc)      -- proc. sensivel aos canais data_regset e data_pc
    begin
        if clk='1' then                 -- quando o sinal de clock estiver habilitado
            if ctrl='0' then            -- quando o sinal de controle for igual a '0'
                dataout<=data_regset;   -- canal de saida recebe o canal de entrada 1
            else                        -- quando o sinal de controle for igual a '1'
                dataout<=data_pc;       -- canal de saida recebe o canal de entrada 2
            end if;
        end if;
    end process;
end mar;

-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 17 de junho de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Registrador de Dados da Memoria  ($mdr)

--------------------------------------------------------------------------------
-- Bibliotecas
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Pinagem do circuito
--------------------------------------------------------------------------------
entity mdr is
    port (
        -- canais de entrada
        datain_mem: in STD_LOGIC_VECTOR (15 downto 0);   -- canal de entrada de dados 1
        datain_cpu: in STD_LOGIC_VECTOR (15 downto 0);   -- canal de entrada de dados 2
```

```
        clk: in STD_LOGIC;                                  -- sinal de clock
        ctrl: in STD_LOGIC;                                 -- sinal de controle

        -- canais de saida
        clk_ir: out STD_LOGIC;                              -- sinal de clock p/ $ir
        dataout_mem: out STD_LOGIC_VECTOR (15 downto 0); -- canal de saida de dados 1
        dataout_cpu: out STD_LOGIC_VECTOR (15 downto 0)  -- canal de saida de dados 2
    );
end mdr;

-------------------------------------------------------------------------------
-- Funcionalidade do circuito
-------------------------------------------------------------------------------
architecture mdr of mdr is
begin
    process (datain_mem) -- processo sensivel ao canal datain_mem
    begin
        if clk='1' then                  -- quando o sinal de clock estiver habilitado
            if ctrl='0' then             -- quando o sinal de controle for igual a '0'
                dataout_cpu<=datain_mem;  -- canal de saida 2 recebe canal de entrada 1
                clk_ir<='1';             -- habilita o sinal de clock p/ $ir
            else                         -- quando o sinal de controle for igual a '1'
                dataout_mem<=datain_cpu;  -- canal de saida 1 recebe canal de entrada 2
                clk_ir<='0';             -- desabilita o sinal de clock p/ $ir
            end if;
        end if;
    end process;
end mdr;


-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 18 de junho de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Contador de Programa  ($pc)

-------------------------------------------------------------------------------
-- Bibliotecas
-------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

-------------------------------------------------------------------------------
-- Pinagem do circuito
-------------------------------------------------------------------------------
entity pc is
    port (
        -- canais de entrada
        data: in STD_LOGIC_VECTOR (15 downto 0);     -- canal de entrada de dados
        clk: in STD_LOGIC;                           -- sinal de controle

        -- canais de saida
        dataout: out STD_LOGIC_VECTOR (15 downto 0)  -- canal de saida de dados
    );
end pc;

-------------------------------------------------------------------------------
-- Funcionalidade do circuito
-------------------------------------------------------------------------------
architecture pc of pc is
begin
    process (data)         -- processo sensivel ao canal data
    begin
        if (clk='1') then  -- quando o sinal de clock estiver ativo
            dataout<=data; -- canal de saida recebe o canal de entrada
```

```
        end if;
    end process;
end pc;



-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 10 de maio de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Somador dedicado de 16 bits de $pc

--------------------------------------------------------------------------------
-- Bibliotecas
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Pinagem do circuito
--------------------------------------------------------------------------------
entity pc_adder is
    generic (
        BusWidth: Integer := 15  -- tamanho do barramento de dados
    );

    port (
        op1: in STD_LOGIC_VECTOR (BusWidth downto 0);    -- primeiro operando
        op2: in STD_LOGIC_VECTOR (10 downto 0);          -- segundo operando
        clk: in STD_LOGIC;                               -- sinal de clock
        ctrl: in STD_LOGIC;                              -- sinal de controle
        result: out STD_LOGIC_VECTOR (BusWidth downto 0) -- resultado da operacao
    );
end pc_adder;


--------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------
architecture pc_adder of pc_adder is
begin
  process (op1)
      variable op3: STD_LOGIC_VECTOR (BusWidth downto 0);
      variable op4: STD_LOGIC_VECTOR (BusWidth downto 0);
      variable temp: STD_LOGIC_VECTOR (BusWidth downto 0);
      variable carry: STD_LOGIC_VECTOR (BusWidth+1 downto 0);

  begin
      if ctrl='0' then  -- quando o sinal de controle for igual a '0'
            op3:="0000000000000010";         -- define variavel op3 com o valor '2'
            carry(0):='0';                   -- zera o bit '0' da variavel carry
                for i in 0 to BusWidth loop  -- adiciona 1o op. com a variavel op3
                      temp(i):=op1(i) xor op3(i) xor carry(i);
                  carry(i+1):=(op1(i) and op3(i)) or (op1(i) and carry(i))
                              or (op3(i) and carry(i));
             end loop;

          result<=temp; -- copia variavel temp para o canal de saida do circuito
        else            -- quando o sinal de controle for igual a '1'
          op4(10 downto 0):=op2;             -- move 2o op. para variavel op4
          op4(15 downto 11):="00000";        -- completa os bits restantes com '0'
            carry(0):='0';                   -- zera o bit '0' da variavel carry
                for i in 0 to BusWidth loop  -- adiciona 1o op. com a variavel op4
                      temp(i):=op1(i) xor op4(i) xor carry(i);
                  carry(i+1):=(op1(i) and op4(i)) or (op1(i) and carry(i))
```

```
                              or (op4(i) and carry(i));
            end loop;
          result<=temp; -- copia variavel temp para o canal de saida do circuito
      end if;
  end process;
end pc_adder;


-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 17 de junho de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Deslocador para intrucoes de desvio

--------------------------------------------------------------------------------
-- Bibliotecas
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Pinagem do circuito
--------------------------------------------------------------------------------
entity shifter is
    port (
        -- canais de entrada
        data: in STD_LOGIC_VECTOR (10 downto 0);     -- canal de entrada de dados
        clk: in STD_LOGIC;                           -- sinal de clock

        -- canais de saida
        dataout: out STD_LOGIC_VECTOR (10 downto 0); -- canal de saida de dados
        ctrl_adder: out STD_LOGIC                    -- canal de controle do modulo adder
    );
end shifter;

--------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------
architecture shifter of shifter is
begin
    process (data)
        variable temp: STD_LOGIC_VECTOR (10 downto 0);
    begin
          for i in 0 to 10-1 loop    -- percorre todos os bits do canal de entrada
              temp(i):=data(i+1);    -- desloca o conteudo da entrada 1 bit p/ a esq.
        end loop;
        temp(10):='0';               -- insere '0' no ultimo bit do resultado
                ctrl_adder<='1';
    end process;
end shifter;


-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 17 de junho de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Unidade de Controle

--------------------------------------------------------------------------------
```

```
-- Bibliotecas
-------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

-------------------------------------------------------------------------------------------
-- Pinagem do circuito
-------------------------------------------------------------------------------------------
entity control is
    generic (
        OpBus: Integer:=3;
        RegBus: Integer:=3
    );

    port (
        -- canais de entrada
        data: in STD_LOGIC_VECTOR (15 downto 0);    -- canal de entrada para a instrucao
        clk: in STD_LOGIC;                          -- sinal de clock

        -- canais de saida
        reg1: out STD_LOGIC_VECTOR (RegBus downto 0);     -- seletor do 1o reg. fonte
        reg2: out STD_LOGIC_VECTOR (RegBus downto 0);     -- seletor do 2o reg. fonte
        reg3: out STD_LOGIC_VECTOR (RegBus downto 0);     -- seletor do reg. destino
        dataout: out STD_LOGIC_VECTOR (7 downto 0);       -- canal de saida de dados 1
        dataout2: out STD_LOGIC_VECTOR (10 downto 0);     -- canal de saida de dados 2
        ctrl_ula: out STD_LOGIC_VECTOR (RegBus downto 0); -- canal de ctrl da ula
        ctrl_reg: out STD_LOGIC_VECTOR (2 downto 0);      -- canal de ctrl banco GPRs
        ctrl_reg2: out STD_LOGIC;                         -- canal de ctrl banco GPRs 2
        ctrl_branch: out STD_LOGIC_VECTOR (2 downto 0);   -- canal de ctrl modulo branch
        clk_ula: out STD_LOGIC;                           -- sinal de clock da ula
        clk_reg: out STD_LOGIC                            -- sinal de clock banco de GPRs
    );
end control;

-------------------------------------------------------------------------------------------
-- Funcionalidade do circuito
-------------------------------------------------------------------------------------------
architecture control of control is

  -- definicao dos modos de enderecamento
  constant direct:    STD_LOGIC_VECTOR (1 downto 0) := "00";
  constant indirect:  STD_LOGIC_VECTOR (1 downto 0) := "01";
  constant immediate: STD_LOGIC_VECTOR (1 downto 0) := "10";

  -- definicao dos tipos de acesso ao banco de GPRs
  constant acc_direct:     STD_LOGIC_VECTOR (2 downto 0) := "000";
  constant acc_indirect:   STD_LOGIC_VECTOR (2 downto 0) := "001";
  constant acc_load:       STD_LOGIC_VECTOR (2 downto 0) := "010";
  constant acc_load_upper: STD_LOGIC_VECTOR (2 downto 0) := "011";
  constant acc_mov:        STD_LOGIC_VECTOR (2 downto 0) := "100";
  constant acc_mem:        STD_LOGIC_VECTOR (2 downto 0) := "101";

  constant i_nop: STD_LOGIC_VECTOR (OpBus downto 0) := "00000";

  -- definicao das instrucoes aritmeticas
  constant i_add: STD_LOGIC_VECTOR (OpBus downto 0) := "00001";
  constant i_sub: STD_LOGIC_VECTOR (OpBus downto 0) := "00010";
  constant i_inc: STD_LOGIC_VECTOR (OpBus downto 0) := "00011";
  constant i_dec: STD_LOGIC_VECTOR (OpBus downto 0) := "00100";
  constant i_mul: STD_LOGIC_VECTOR (OpBus downto 0) := "00101";
  constant i_div: STD_LOGIC_VECTOR (OpBus downto 0) := "00110";
  constant i_mod: STD_LOGIC_VECTOR (OpBus downto 0) := "00111";

  -- definicao das instrucoes logicas
  constant i_shr: STD_LOGIC_VECTOR (OpBus downto 0) := "01000";
  constant i_shl: STD_LOGIC_VECTOR (OpBus downto 0) := "01001";
  constant i_ror: STD_LOGIC_VECTOR (OpBus downto 0) := "01010";
  constant i_rol: STD_LOGIC_VECTOR (OpBus downto 0) := "01011";
  constant i_or:  STD_LOGIC_VECTOR (OpBus downto 0) := "01100";
  constant i_and: STD_LOGIC_VECTOR (OpBus downto 0) := "01101";
  constant i_xor: STD_LOGIC_VECTOR (OpBus downto 0) := "01110";
```

```
  constant i_not: STD_LOGIC_VECTOR (OpBus downto 0) := "01111";

  -- definicao das instrucoes de acesso a memoria
  constant i_lli: STD_LOGIC_VECTOR (OpBus downto 0) := "10000";
  constant i_lui: STD_LOGIC_VECTOR (OpBus downto 0) := "10001";
  constant i_lw:  STD_LOGIC_VECTOR (OpBus downto 0) := "10010";
  constant i_sw:  STD_LOGIC_VECTOR (OpBus downto 0) := "10011";
  constant i_lb:  STD_LOGIC_VECTOR (OpBus downto 0) := "10100";
  constant i_sb:  STD_LOGIC_VECTOR (OpBus downto 0) := "10101";
  constant i_mov: STD_LOGIC_VECTOR (OpBus downto 0) := "10110";

  -- definicao das instrucoes de desvio condicional
  constant i_jv:  STD_LOGIC_VECTOR (OpBus downto 0) := "11000";
  constant i_jnv: STD_LOGIC_VECTOR (OpBus downto 0) := "11001";
  constant i_jz:  STD_LOGIC_VECTOR (OpBus downto 0) := "11010";
  constant i_jnz: STD_LOGIC_VECTOR (OpBus downto 0) := "11011";
  constant i_jc:  STD_LOGIC_VECTOR (OpBus downto 0) := "11100";
  constant i_jnc: STD_LOGIC_VECTOR (OpBus downto 0) := "11101";

  -- instrucoes de desvio incondicional
  constant i_jmp: STD_LOGIC_VECTOR (OpBus downto 0) := "11110";

  -- registradores temporarios
  constant t00: STD_LOGIC_VECTOR (RegBus downto 0) := "1110";
  constant t01: STD_LOGIC_VECTOR (RegBus downto 0) := "1111";

begin
    process (data)
        variable opcode: STD_LOGIC_VECTOR (4 downto 0); -- armazena codigo da operacao
        variable mode: STD_LOGIC_VECTOR (1 downto 0);   -- armazena modo de enderecamento
        variable destiny: STD_LOGIC;                    -- aemazena registrador de destino

    begin
        opcode:=data(15 downto 11); -- define o codigo da operacao
        mode:=data(10 downto 9);    -- define o modo de enderecamento

        case opcode is

            -- formato-R de representacao de instrucoes
            -- instrucoes aritmeticas e logicas
            when i_or | i_add | i_sub | i_inc | i_dec | i_mul | i_div | i_mod
                 | i_shl | i_shr | i_rol | i_ror |i_and | i_xor | i_not =>

                if opcode=i_add then
                    ctrl_ula<="0001";          -- indica operacao de adicao
                    ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
                end if;
                if opcode=i_sub then
                    ctrl_ula<="0010";          -- indica operacao de subtracao
                    ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
                end if;
                if opcode=i_inc then
                    ctrl_ula<="0011";          -- indica operacao de incremento
                    ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
                end if;
                if opcode=i_dec then
                    ctrl_ula<="0100";          -- indica operacao de decremento
                    ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
                end if;
                if opcode=i_mul then
                    ctrl_ula<="0101";          -- indica operacao de multiplicacao
                    ctrl_reg2<='1';            -- indica que serão acessados 2 reg.
                end if;
                if opcode=i_div then
                    ctrl_ula<="0110";          -- indica operacao de divisao
                    ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
                end if;
                if opcode=i_mod then
                    ctrl_ula<="0111";          -- indica operacao de resto da divisao
                    ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
                end if;
```

```
    if opcode=i_shl then
        ctrl_ula<="1000";          -- indica operacao de deslc. p/ esq.
        ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
    end if;
    if opcode=i_shr then
        ctrl_ula<="1001";          -- indica operacao de desloc. p/ dir.
        ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
    end if;
    if opcode=i_rol then
        ctrl_ula<="1010";          -- indica operacao de rotacao p/ esq.
        ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
    end if;
    if opcode=i_ror then
        ctrl_ula<="1011";          -- indica operacao de rotacao p/ dir.
        ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
    end if;
    if opcode=i_or then
        ctrl_ula<="1100";          -- indica operacao de or logico
        ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
    end if;
    if opcode=i_and then
        ctrl_ula<="1101";          -- indica operacao de and logico
        ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
    end if;
    if opcode=i_xor then
        ctrl_ula<="1110";          -- indica operacao de xor logico
        ctrl_reg2<='1';            -- indica que serao acessados 2 reg.
    end if;
    if opcode=i_not then
        ctrl_ula<="1111";          -- indica operacao de not logico
        ctrl_reg2<='0';            -- indica que sera acessado 1 reg.
    end if;

    destiny:=data(8);        -- define o registrador de destino da operacao
    clk_ula<='1';            -- libera o funcionamento da ula
    clk_reg<='1';            -- libera o funcionamento do banco de GPRs
    reg1<=data(7 downto 4);  -- atribui endereco do registrador com 1o operando
    reg2<=data(3 downto 0);  -- atribui endereco do registrador com 2o operando

    case mode is
        when direct | indirect =>       -- quando modo direto ou indireto
            if mode=direct then         -- quando direto
                ctrl_reg<=acc_direct;   -- indica acesso direto
            else                        -- quando indireto
                ctrl_reg<=acc_indirect; -- indica acesso indireto
            end if;
            if destiny='0' then         -- quando bit de destino vale '0'
                reg3<=data(7 downto 4); -- reg. c/ 1o op. recebe o resultado
            else                        -- quando bit de destino vale '1'
                reg3<=data(3 downto 0); -- reg. c/ 1o op. recebe o resultado
            end if;

        when immediate =>        -- quando modo imediato
            if destiny='0' then  -- quando bit de destino vale '0'
                reg3<=t00;       -- $t0 recebe o resultado
            else                 -- quando bit de destino vale '1'
                reg3<=t01;       -- $t1 recebe o resultado
            end if;
        when others =>
    end case;

-- formato-I de representacao de instrucoes
-- instrucoes de carregamento de constantes em registradores
when i_li | i_lui =>
    destiny:=data(10);         -- define o reg. de destino da operacao
    clk_reg<='1';              -- libera o funcionamento do banco de GPRs
    dataout<=data(7 downto 0); -- define a constante a ser carregada
    if destiny='0' then        -- quando bit que indica o destino vale '0'
        reg3<=t00;             -- carrega $t0 com a constante
    else                       -- quando bit de destino vale '1'
        reg3<=t01;             -- carrega $t1 com a constante
```

```
                end if;
                if opcode=i_li then           -- quando instrucao load immediate
                    ctrl_reg<=acc_load;       -- indica carregamento da parte baixa
                else                          -- quando instrucao load upper immediate
                    ctrl_reg<=acc_load_upper; -- indica carregamento da parte alta
                end if;

            -- formato-Load/Store
            -- instrucoes de acesso a memoria
            when  i_lw | i_sw | i_lb | i_sb | i_mov =>
                clk_reg<='1';           -- libera o funcionamento do banco de GPRs
                reg1<=data(7 downto 4); -- define reg. onde se encontra o endereco
                reg2<=data(3 downto 0); -- define reg. onde se encontra o dado

                if opcode=i_mov then    -- quando instrucao move
                    ctrl_reg<=acc_mov;  -- indica uma copia entre registradores
                else                    -- quando demais
                    ctrl_reg<=acc_mem;  -- indica uma instrucao de acesso a memoria
                end if;

            -- formato-Branch
            -- instrucoes de desvio condicional e incondicioal
            when i_jmp | i_jc | i_jnc | i_jv | i_jnv | i_jz | i_jnz | i_jeq | i_jen =>

                dataout2<=data(10 downto 0); -- numero de instrucoes a serem "saltadas"

            when others =>

        end case;
    end process;
end control;


-- ************************************************************************************
-- ************************************************************************************
-- ************************************************************************************
-- ************************************************************************************

-- Ultima alteracao: 10 de maio de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Banco de GPRs de 16 bits

------------------------------------------------------------------------------------
-- Bibliotecas
------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

------------------------------------------------------------------------------------
-- Pinagem do circuito
------------------------------------------------------------------------------------
entity register_set is
    generic (
        BusWidth: Integer := 15  -- Tamanho do barramento de dados
    );

    port (
        -- canais de entrada
        v1: in STD_LOGIC_VECTOR (7 downto 0);           -- constante para carregamento
        v_ula: in STD_LOGIC_VECTOR (BusWidth downto 0); -- resultado da operacao da ula
        sel1: in STD_LOGIC_VECTOR (3 downto 0);         -- 1o seletor de registrador
        sel2: in STD_LOGIC_VECTOR (3 downto 0);         -- 2o seletor de registrador
        sel3: in STD_LOGIC_VECTOR (3 downto 0);         -- seletor de reg. de destino
        t: in STD_LOGIC_VECTOR (2 downto 0);            -- seletor de tipo da operacao
        q:  in STD_LOGIC;                               -- seletor do numero de op.
        clk: in STD_LOGIC;                              -- sinal de clock

        -- canais de saida
```

```
        o1: out STD_LOGIC_VECTOR (BusWidth downto 0);    -- 1o sinal de saida
        o2: out STD_LOGIC_VECTOR (BusWidth downto 0);    -- 2o sinal de saida
        o3: out STD_LOGIC_VECTOR (BusWidth downto 0);    -- 1o sinal de saida
        o4:    out STD_LOGIC_VECTOR (BusWidth downto 0)  -- 2o sinal de saida
    );
end register_set;

architecture register_set of register_set is


-- ********************************************************************************
--    Codificacao de rotinas
-- ********************************************************************************
--------------------------------------------------------------------------------------
-- Nome: bit_vec2int
-- Parametro(s) de entrada: A: STD_LOGIC_VECTOR
-- Parametro de saida: INTEGER
-- Chamada: main
-- Descricao: converte um valor binario em um valor decimal
--------------------------------------------------------------------------------------
function bit_vec2int(A : STD_LOGIC_VECTOR) return integer is
variable RESULT: integer:=0;
variable TMP: integer:=1;
begin
  if A'length = 0
    then return RESULT;
  end if;
  for i in A'reverse_range loop
    if a(i)='1'
      then RESULT:=RESULT+TMP;
    end if;
    TMP:=TMP*2;
  end loop;
  return RESULT;
end;

--------------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------------

begin
  process (t,v_ula) -- processo sensivel ao canal t

  -- defincao de um tipo, denominado conjunto de registradores, com 16 itens de 16 bits
  type reg_set is array (15 downto 0) of STD_LOGIC_VECTOR (15 downto 0);

  variable set1: reg_set;                       -- instancia uma variavel do tipo reg_set
  variable taux: STD_LOGIC_VECTOR (2 downto 0); -- variavel que guarda o tipo de acesso
  variable end_of_cycle: INTEGER;               -- var. que indica o fim de uma execucao
  variable selaux1,                             -- var. que define reg. com o 1o op.
           selaux2,                             -- var. que define reg. com o 2o op.
           selaux3: INTEGER;                    -- var. que define reg. de destino

  begin

  if end_of_cycle=1 then
    taux:="111";      -- indica que se trata de uma escrita de dados da ula
  else
    taux:=t;          -- copia o tipo do acesso para a variavel taux
    end_of_cycle:=0;  -- incializa variavel end_of_cycle
    selaux1:=bit_vec2int(sel1);  -- conv. p/ decimal valor do 1o reg. e copia em selaux1
    selaux2:=bit_vec2int(sel2);  -- conv. p/ decimal valor do 2o reg. e copia em selaux2
    selaux3:=bit_vec2int(sel3);  -- conv. p/ decimal valor do reg. dest. e copia em selaux3
  end if;

      case taux is
        when "000" =>                           -- quando for acesso direto
            if q='0' then                       -- quando for um operando
                o1 <= set1(selaux1)(15 downto 0); -- copia valor do 1o reg. para a 1a saida
                end_of_cycle:=1;                -- indica que resta um ciclo
            else                                -- quando forem dois operandos
```

```
                o1 <= set1(selaux1)(15 downto 0); -- copia valor do 1o reg. para a 1a saida
                o2 <= set1(selaux2)(15 downto 0); -- copia valor do 1o reg. para a 2a saida
                end_of_cycle:=1;                  -- indica que resta um ciclo
            end if;

        when "001" =>                             -- quando for acesso indireto
            if q='0' then                         -- quando for um operando
                selaux1:=bit_vec2int(set1(selaux1)(3 downto 0)); -- 1o reg. real
                o1 <= set1(selaux1)(15 downto 0); -- copia valor do 1o reg. p/ 1a saida
                end_of_cycle:=1;                  -- indica que resta um ciclo
            else                                  -- quando forem dois operandos
                selaux1:=bit_vec2int(set1(selaux1)(3 downto 0)); -- 1o reg. real
                selaux2:=bit_vec2int(set1(selaux2)(3 downto 0)); -- 2o reg. real
                o1 <= set1(selaux1)(15 downto 0); -- copia valor do 1o reg. p/ 1a saida
                o2 <= set1(selaux2)(15 downto 0); -- copia valor do 1o reg. p/ 2a saida
                end_of_cycle:=1;                  -- indica que resta um ciclo
            end if;

        when "010" =>                       -- quando carga de em parte baixa de reg.
            set1(selaux3)(7 downto 0):=v1;  -- copia valor para parte baixa do reg.
            end_of_cycle:=2;                -- indica que a instrucao encerrou

        when "011" =>                       -- quando carga de em parte alta de reg.
            set1(selaux3)(15 downto 8):=v1; -- copia valor para parte alta do reg.
            set1(selaux3)(7 downto 0):="00000000"; -- limpa parte baixa do reg.
            end_of_cycle:=2;                    -- indica que a instrucao encerrou

        when "100" =>                     -- quando movimentacao entre reg.
            set1(selaux1):=set1(selaux2); -- copia valor do 1o reg. para o 2o reg.
            end_of_cycle:=2;              -- indica que a instrucao encerrou

        when "101" =>                         -- quando escrita na memoria
            o3 <= set1(selaux1)(15 downto 0); -- copia valor do 1o reg. para $mar
            o4 <= set1(selaux2)(15 downto 0); -- copia valor do 1o reg. para $mdr

        when "111" =>                         -- quando for retorno da ula
            set1(selaux3):=v_ula;             -- copia resultado da op. p/ reg.
            end_of_cycle:=2;                  -- indica que a instrucao encerrou

        when others =>
    end case;


  end process;
end register_set;


-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************
-- ********************************************************************************

-- Ultima alteracao: 9 de maio de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Unidade Logica e Aritmetica de 16 bits

-- Diagrama do circuito:

--        -------------
--       |             |
--   op1 -      U      - result
--       |      L      |
--   op2 -      A      - c
--       |             |
--   sel -     16      - z
--       |             |
--   clk -      B      - o
--       |      I      |
```

```
--       |      T      - b
--       |      S      |
--       |             |
--        -------------


--   Codigos de operacao da ULA:

--   ****************************************************************
--   | Codigo   |  Operacao                                           |
--   ****************************************************************
--   | 0000     |  NOP (no operation)                                 |
--   | 0001     |  Soma 1o operando com o 2o operando                 |
--   | 0010     |  Subtrai    2o operando do 1o operando              |
--   | 0011     |  Incrementa o 1o operando                           |
--   | 0100     |  Decrementa o 1o operando                           |
--   | 0101     |  Multiplica 1o operando com o 2o operando           |
--   | 0110     |  Divide 1o operando pelo 2o operando                |
--   | 0111     |  Resto da divisao do 1o operando pelo 2o operando   |
--   | 1000     |  And bit a bit entre os operandos                   |
--   | 1001     |  Or bit a bit entre os operandos                    |
--   | 1010     |  Xor    bit a bit entre os operandos                |
--   | 1011     |  Complementa o 1o operando                          |
--   | 1100     |  Desloca o 1o operando para a esquerda              |
--   | 1101     |  Desloca o 1o operando para a direta                |
--   | 1110     |  Rotaciona o 1o operando para a esquerda            |
--   | 1111     |  Rotaciona o 1o operando para a direta              |
--   ****************************************************************


-------------------------------------------------------------------------------------------
-- Bibliotecas
-------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-------------------------------------------------------------------------------------------
-- Pinagem do circuito
-------------------------------------------------------------------------------------------
entity ula is
    generic (
        BusWidth: Integer := 15;        -- Tamanho do barramento de dados
        SecondaryBus: Integer := 31     -- Tamanho do barramento secundario de dados
    );

    port (
        -- Canais de entrada
        op1: in STD_LOGIC_VECTOR (BusWidth downto 0);     -- Primeiro operando
        op2: in STD_LOGIC_VECTOR (BusWidth downto 0);     -- Segundo operando
        sel: in STD_LOGIC_VECTOR (3 downto 0);            -- Seletor da operacao
        clk: in STD_LOGIC;                                -- Flag de habilitacao

        -- Canais de saida
        result: out STD_LOGIC_VECTOR (BusWidth downto 0); -- Resultado da operacao
        c: out STD_LOGIC;                                 -- Flag indicador de carry
        z: out STD_LOGIC;                                 -- Flag indicador de zero
        o: out STD_LOGIC                                  -- Flag indicador de overflow
    );
end ula;

architecture ula of ula is

-- ********************************************************************************
--    Codificacao de rotinas
-- ********************************************************************************
-------------------------------------------------------------------------------------------
-- Nome: comp
-- Parametro(s) de entrada: x: STD_LOGIC_VECTOR, y: STD_LOGIC_VECTOR
-- Parametro de saida: STD_LOGIC
-- Chamada: div
```

```
-- Descricao: faz uma comparacao entre dois valores, retornando '1' se o valor de x
--            for maior que o valor de y e '0' em caso contrario.
-------------------------------------------------------------------------------------
function comp (x,y: STD_LOGIC_VECTOR) return STD_LOGIC is
variable z:STD_LOGIC;
begin
    if (x<y) then
        z:='1';
    else
        z:='0';
    end if;
    return z;
end comp;

-- *************************************************************************************
--    Codificacao de rotinas
-- *************************************************************************************
-------------------------------------------------------------------------------------
-- Nome: sub32bits
-- Parametro(s) de entrada: x,y: STD_LOGIC_VECTOR
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: div
-- Descricao: retorna o resultado da subtracao de dois valores de 32 bits
-------------------------------------------------------------------------------------
function sub32bits (x,y:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
variable s:STD_LOGIC_VECTOR(SecondaryBus downto 0);
variable comp_y:STD_LOGIC_VECTOR(SecondaryBus downto 0);
variable carry:STD_LOGIC_VECTOR(SecondaryBus+1 downto 0);
begin
    comp_y:=not y;
    carry(0):='1';
    for i in 0 to SecondaryBus loop
          s(i):=x(i) xor comp_y(i) xor carry(i);
          carry(i+1):=(x(i) and comp_y(i)) or (x(i) and carry(i)) or
                      (comp_y(i) and carry(i));
    end loop;
    return s;
end;

-- *************************************************************************************
--    Codificacao de rotinas
-- *************************************************************************************
-------------------------------------------------------------------------------------
-- Nome: add16bits
-- Parametro(s) de entrada: x,y: STD_LOGIC_VECTOR
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: mux
-- Descricao: retorna o resultado da adicao de dois valores de 16 bits
-------------------------------------------------------------------------------------
function add16bits (x,y:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
variable s:STD_LOGIC_VECTOR(BusWidth downto 0);
variable comp_y:STD_LOGIC_VECTOR(BusWidth downto 0);
variable carry:STD_LOGIC_VECTOR(BusWidth+1 downto 0);
begin
    carry(0):='0';
    for i in 0 to BusWidth loop
          s(i):=x(i) xor y(i) xor carry(i);
          carry(i+1):=(x(i) and y(i)) or (x(i) and carry(i)) or (y(i) and carry(i));
    end loop;
    return s;
end;

-- *************************************************************************************
--    Codificacao de rotinas
-- *************************************************************************************
-------------------------------------------------------------------------------------
-- Nome: shiftr
-- Parametro(s) de entrada: x: STD_LOGIC_VECTOR
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: div
-- Descricao: retorna o valor de entrada deslocado 1 bit para a direita
```

```
-------------------------------------------------------------------------------
function shiftr (x:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
variable y:STD_LOGIC_VECTOR(SecondaryBus downto 0);
begin
    for i in 0 to SecondaryBus-1 loop
        y(i):=x(i+1);
    end loop;
    y(SecondaryBus):='0';
    return y;
end;

-- *****************************************************************************
--    Codificacao de rotinas
-- *****************************************************************************
-------------------------------------------------------------------------------
-- Nome: shiftl
-- Parametro(s) de entrada: x: STD_LOGIC_VECTOR
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: div, mux
-- Descricao: retorna o valor de entrada deslocado 1 bit para a esquerda
-------------------------------------------------------------------------------
function shiftl (x:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
variable y:STD_LOGIC_VECTOR(BusWidth downto 0);
begin
    for i in 0 to BusWidth-1 loop
        y(i+1):=x(i);
    end loop;
    y(0):='0';
    return y;
end;

-- *****************************************************************************
--    Codificacao de rotinas
-- *****************************************************************************
-------------------------------------------------------------------------------
-- Nome: div
-- Parametro(s) de entrada: x,y: STD_LOGIC_VECTOR; z:STD_LOGIC
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: main
-- Descricao: realiza a divisao de dois valores de 16 bits, retornando o resultado in-
--            teiro se o parametro z for igual a '0' ou o resto da divisao se z for i-
--              gual a '1'
-------------------------------------------------------------------------------
function div(x,y: STD_LOGIC_VECTOR; z:STD_LOGIC) return STD_LOGIC_VECTOR is
variable dividendo, divisor:STD_LOGIC_VECTOR(SecondaryBus downto 0);
variable quociente: STD_LOGIC_VECTOR(BusWidth downto 0);
variable resto: STD_LOGIC_VECTOR(BusWidth downto 0);
begin
    for i in BusWidth downto 0 loop
        dividendo(i):=x(i);             -- copia o valor de x para 16 LSBs do dividendo
        dividendo(i+(BusWidth+1)):='0'; -- coloca '0' nos 16 MSBs do dividendo
        divisor(i+(BusWidth+1)):=y(i);  -- copia o valor de y nos MSBs do divisor
        divisor(i):='0';                -- coloca '0' nos 16 LSBs do divisor
    end loop;

    for i in 0 to BusWidth loop
    for i in 0 to BusWidth loop
            divisor:=shiftr(divisor);           -- desloca divisor 1 bit p/ dir.
            quociente:=shiftl(quociente);       -- desloca quociente 1 bit p/ esq.
            if comp(dividendo,divisor)='1' then -- se for dividendo < divisor
                quociente(0):='0';              -- zera o LSB do divisor
            else                                -- se dividendo >= ao divisor
                quociente(0):='1';              -- coloca '1' no LSB do divisor
                dividendo:=sub32bits(dividendo,divisor); -- subtrai divisor do dividendo
            end if;
    end loop;

    for i in BusWidth downto 0 loop
        resto(i):=dividendo(i); -- copia o que restou no dividendo para a variavel resto
    end loop;
```

```
    if z='0' then               -- quando z for igual a '0'
        return quociente;       -- retorna o resultado inteiro da divisao
    else                        -- quando z for igual a '1'
        return resto;           -- retorna o resto da divisao
    end if;

end div;

-- ********************************************************************************
--    Codificacao de rotinas
-- ********************************************************************************
--------------------------------------------------------------------------------
-- Nome: *
-- Parametro(s) de entrada: x,y: STD_LOGIC_VECTOR; z:STD_LOGIC
-- Parametro de saida: STD_LOGIC_VECTOR
-- Chamada: main
-- Descricao: multiplica dois valores de 8 bits
--------------------------------------------------------------------------------
function "*" (x,y : STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
variable multiplicando, multiplicador, produto: STD_LOGIC_VECTOR (BusWidth downto 0);
variable carry: STD_LOGIC;
begin
    multiplicando:=x; -- variavel multiplicando recebe o 1o operando
    multiplicador:=y; -- variavel multiplicador recebe o 2o operando
    produto:="0000000000000000";

    for i in 7 downto 0 loop
        produto:=shiftl(produto); -- desloca variavel produto 1 bit para a esquerda
        carry:=multiplicador(7);  -- variavel carry recebe MSB do multiplicador
        if carry='1' then         -- quando carry for igual a '1'
            produto:=add16bits(produto,multiplicando); -- soma produto com multiplicando
        end if;
        multiplicador:=shiftl(multiplicador);          -- desloca mutiplicador p/ esq.
    end loop;
    return produto; -- retorna produto
end "*";


--------------------------------------------------------------------------------
-- Funcionalidade do circuito
--------------------------------------------------------------------------------
begin
  process (op1)
      variable carry: STD_LOGIC_VECTOR (BusWidth+1 downto 0);  -- var. de carry
      variable temp: STD_LOGIC_VECTOR (BusWidth downto 0);     -- var. para resultado
      variable comp_op2: STD_LOGIC_VECTOR (BusWidth downto 0); -- var. c/ comp. do 2o op.

  begin
      if (clk='1') then
          case sel is
              when "0000" =>                -- NOP (no operation)

              when "0001" =>                -- Soma
                    carry(0):='0';
                    for i in 0 to BusWidth loop
                        temp(i):=op1(i) xor op2(i) xor carry(i);
                        carry(i+1):=(op1(i) and op2(i)) or (op1(i) and carry(i))
                        or (op2(i) and carry(i));
                  end loop;

              when "0010" =>                -- Subtracao
                    comp_op2:=not(op2);
                    carry(0):='1';
                    for i in 0 to BusWidth loop
                          temp(i):=op1(i) xor comp_op2(i) xor carry(i);
                          carry(i+1):=(op1(i) and comp_op2(i)) or (op1(i) and carry(i))
                                      or (comp_op2(i) and carry(i));
                    end loop;

              when "0011" =>                -- Incremento
                    carry(0):='1';
```

```
                for i in 0 to BusWidth loop
                    temp(i):=op1(i) xor carry(i);
                    carry(i+1):=op1(i) and carry(i);
                end loop;

            when "0100" =>                -- Decremento
                carry(0):='0';
                for i in 0 to BusWidth loop
                    temp(i):=op1(i) xor '1' xor carry(i);
                    carry(i+1):=op1(i) or carry(i);
                end loop;

            when "0101" =>                -- Multiplicacao
                temp:=op1 * op2;

            when "0110" =>                -- Divisao
                temp:=div(op1,op2,'0');

            when "0111" =>                -- Resto da divisao
                temp:=div(op1,op2,'1');

            when "1000" =>                -- And
                temp:=op1 and op2;

            when "1001" =>                -- Or
                temp:=op1 or op2;

            when "1010" =>                -- Xor
                temp:=op1 xor op2;

            when "1011" =>                -- Not
                temp:=not op1;

            when "1100" =>                -- Deslocamento para a esquerda
                for i in 0 to BusWidth-1 loop
                    temp(i):=op1(i+1);
                end loop;
                temp(BusWidth):='0';

            when "1101" =>                -- Deslocamento para a direita
                for i in 0 to BusWidth-1 loop
                    temp(i+1):=op1(i);
                end loop;
                temp(0):='0';

            when "1110" =>                -- Rotacao para a esquerda
                for i in 0 to BusWidth-1 loop
                    temp(i):=op1(i+1);
                end loop;
                temp(BusWidth):=op1(0);

            when "1111" =>                -- Rotacao para a direita
                for i in 0 to BusWidth-1 loop
                    temp(i+1):=op1(i);
                end loop;
                temp(0):=op1(BusWidth);

            when others =>

        end case;
    end if;

    if temp="0000000000000000" then       -- seta flag de zero
        z<='1';
    else
        z<='0';
    end if;

    if carry(8)=carry(7) then             -- seta flag de overflow
        o<='0';
    else
```

```
          o<='1';
      end if;

      c<=carry(8);                          -- seta flag de carry

      result<=temp;                         -- atualiza o resultado

  end process;
end ula;



-- ************************************************************************************
-- ************************************************************************************
-- ************************************************************************************
-- ************************************************************************************

-- Ultima alteracao: 9 de maio de 2004
-- Autor: Rodrigo Bittencourt Motta
-- Curso: Ciencia da Computacao
-- Disciplina: Trabalho de Conclusao de Curso
-- Orientador: Luciano Lores Caimi
-- Descricao: Unidade Central de Processamento

------------------------------------------------------------------------------------
-- Bibliotecas
------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

------------------------------------------------------------------------------------
-- Pinagem do circuito
------------------------------------------------------------------------------------
entity cpu is
    port (
        -- Canais de entrada
        clk: in STD_LOGIC;                              -- sinal de clock
        datain_mem: in STD_LOGIC_VECTOR (15 downto 0);  -- canal de entrada da memoria

        -- Canais de saida
        dataout_mem: out STD_LOGIC_VECTOR (15 downto 0); -- canal de saida p/ memoria
        addr_mem: out STD_LOGIC_VECTOR (15 downto 0)     -- canal de saida c/ end. de mem.
    );
end cpu;

------------------------------------------------------------------------------------
-- Funcionalidade do circuito
------------------------------------------------------------------------------------
architecture cpu of cpu is

-- declaracao do registrador de instrucao
component ir
    port (
        data_ir: in STD_LOGIC_VECTOR (15 downto 0);
        clk: in STD_LOGIC;
        dataout_ir: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;

-- declaracao do registrador de enderecos da memoria
component mar
    port (
        data_regset: in STD_LOGIC_VECTOR (15 downto 0);
        data_pc: in STD_LOGIC_VECTOR (15 downto 0);
        clk: in STD_LOGIC;
        ctrl: in STD_LOGIC;
        dataout: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;
```

```
-- declaracao do registrador de dados da memoria
component mdr
    port (
        datain_mem: in STD_LOGIC_VECTOR (15 downto 0);
        datain_cpu: in STD_LOGIC_VECTOR (15 downto 0);
        clk: in STD_LOGIC;
        ctrl: in STD_LOGIC;
        clk_ir: out STD_LOGIC;
        dataout_mem: out STD_LOGIC_VECTOR (15 downto 0);
        dataout_cpu: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;

-- declaracao do contador de programa
component pc
    port (
        data: in STD_LOGIC_VECTOR (15 downto 0);
        clk: in STD_LOGIC;
        dataout: out STD_LOGIC_VECTOR (15 downto 0)
    );
end component;

-- declaracao do somador dedicado para o contador de programa
component pc_adder
    generic (
        BusWidth: Integer := 15  -- tamanho do barramento de dados
    );

    port (
        op1: in STD_LOGIC_VECTOR (BusWidth downto 0);
        op2: in STD_LOGIC_VECTOR (10 downto 0);
        clk: in STD_LOGIC;
        ctrl: in STD_LOGIC;
        result: out STD_LOGIC_VECTOR (BusWidth downto 0)
    );
end component;

-- declaracao deslocador pra instrucoes de desvio
component shifter
    port (
        data: in STD_LOGIC_VECTOR (10 downto 0);
        clk: in STD_LOGIC;
        ctrl_adder: out STD_LOGIC;
        dataout: out STD_LOGIC_VECTOR (10 downto 0)
    );
end component;

-- declaracao do conjunto de registradores de proposito geral
component register_set
    generic (
        BusWidth: Integer := 15
    );

    port (
        v1: in STD_LOGIC_VECTOR (7 downto 0);
        v_ula: in STD_LOGIC_VECTOR (BusWidth downto 0);
        sel1: in STD_LOGIC_VECTOR (3 downto 0);
        sel2: in STD_LOGIC_VECTOR (3 downto 0);
        sel3: in STD_LOGIC_VECTOR (3 downto 0);
        t: in STD_LOGIC_VECTOR (2 downto 0);
        q:  in STD_LOGIC;
        clk: in STD_LOGIC;
        o1: out STD_LOGIC_VECTOR (BusWidth downto 0);
        o2: out STD_LOGIC_VECTOR (BusWidth downto 0);
        o3: out STD_LOGIC_VECTOR (BusWidth downto 0);
        o4:    out STD_LOGIC_VECTOR (BusWidth downto 0)
    );
end component;

-- declaracao da unidade logica e aritmetica
```

```
component ula
    generic (
        BusWidth: Integer := 15;
        SecondaryBus: Integer := 31
    );

    port (
        op1: in STD_LOGIC_VECTOR (BusWidth downto 0);
        op2: in STD_LOGIC_VECTOR (BusWidth downto 0);
        sel: in STD_LOGIC_VECTOR (3 downto 0);
        clk: in STD_LOGIC;
        result: out STD_LOGIC_VECTOR (BusWidth downto 0);
        c: out STD_LOGIC;
        z: out STD_LOGIC;
        o: out STD_LOGIC
    );
end component;

-- declaracao da unidade de controle
component control
    generic (
        OpBus: Integer:=4;
        RegBus: Integer:=3
    );

    port (
        data: in STD_LOGIC_VECTOR (15 downto 0);
        clk: in STD_LOGIC;
        reg1: out STD_LOGIC_VECTOR (RegBus downto 0);
        reg2: out STD_LOGIC_VECTOR (RegBus downto 0);
        reg3: out STD_LOGIC_VECTOR (RegBus downto 0);
        dataout: out STD_LOGIC_VECTOR (7 downto 0);
        dataout2: out STD_LOGIC_VECTOR (10 downto 0);
        ctrl_ula: out STD_LOGIC_VECTOR (RegBus downto 0);
        ctrl_reg: out STD_LOGIC_VECTOR (2 downto 0);
        ctrl_reg2: out STD_LOGIC;
        ctrl_branch: out STD_LOGIC_VECTOR (2 downto 0);
        clk_ula: out STD_LOGIC;
        clk_reg: out STD_LOGIC
    );
end component;

-- lista de sinais
signal  ir_in,       -- canal de entrada para $ir
        mdr_in,      -- canal de entrada para $mdr
        mar_in,      -- canal que conecta $pc com $mar
        mar_in2,     -- canal que conecta o conjunto de GPRs com $mar
        pc_in,       -- canal de entrada de $pc
        adder_in,    -- canal de entrada para o somador de $pc
        control_in,  -- canal de entrada para a unidade de controle
        ula_in1,     -- canal de entrada para o primeiro operando da ula
        ula_in2,     -- canal de entrada para o segundo operando da ula
        result_ula: STD_LOGIC_VECTOR (15 downto 0); -- canal de saida p/ result. da ula

signal  reg1_addr,   -- canal de entrada de ctrl para 1o op. no banco de GRPs
        reg2_addr,   -- canal de entrada de ctrl para 1o op. no banco de GRPs
        reg3_addr,   -- canal de entrada de ctrl para reg. de dest. no banco de GRPs
        ctrl_ula : STD_LOGIC_VECTOR (3 downto 0); -- canal de ctrl para op. da ula

signal regset_in1: STD_LOGIC_VECTOR (7 downto 0);

signal  regset_in2, -- canal de entrada do banco de GRPs
        adder_in2,  -- canal de entrada para o somador de $pc
        shifter_in: STD_LOGIC_VECTOR (10 downto 0); -- canal de entrada p/ deslocador

signal  ctrl_branch,
        ctrl_regset: STD_LOGIC_VECTOR (2 downto 0);

signal  clk_ir,          -- sinal de clock de $ir
        clk_mar,         -- sinal de clock de $mar
        clk_adder,       -- sinal de clock do somador de $pc
```

```
        clk_pc,          -- sinal de clock de $pc
        clk_shifter,     -- sinal de clock do deslocador
        clk_ula,         -- sinal de clock da ula
        clk_regset,      -- sinal de clock do conjunto de GPRs
        clk_control,     -- sinal de clock da unidade de controle
        ctrl_mdr,        -- sinal de controle de $mdr
        ctrl_mar,        -- sinal de controle de $mar
        ctrl_adder,      -- sinal de controle do somador de $pc
        ctrl_regset2,    -- sinal de controle do conjunto de GPRs
        overflow,        -- flag de overflow
        carry,           -- flag de carry
        zero: STD_LOGIC; -- flag de zero

begin
   -- instanciacao de $mdr
   mdr1: mdr
   port map (
    datain_mem => datain_mem,
    datain_cpu => mdr_in,
    clk => clk,
    ctrl => ctrl_mdr,
    clk_ir => clk_ir,
    dataout_mem => dataout_mem,
    dataout_cpu => ir_in
   );

   -- instanciacao de $mdr
   mar1: mar
   port map (
    data_regset => mar_in,
    data_pc => mar_in2,
    clk => clk_mar,
    ctrl => ctrl_mar,
    dataout => addr_mem
   );

   -- instanciacao de $ir
   ir1: ir
   port map (
    data_ir => ir_in,
    clk => clk_ir,
    dataout_ir => control_in
   );

   -- instanciacao de $pc
   pc1: pc
   port map (
    data => pc_in,
    clk => clk_pc,
    dataout => adder_in
   );

   -- instanciacao do somador dedicado de $pc
   pc_adder1: pc_adder
   port map (
    op1 => adder_in,
    op2 => adder_in2,
    clk => clk_adder,
    ctrl => ctrl_adder,
    result => mar_in
   );

   -- instanciacao do deslocador para instrucoes de desvio
   shifter1: shifter
   port map (
    data => shifter_in,
    clk => clk_shifter,
    ctrl_adder => ctrl_adder,
    dataout => adder_in2
   );
```

```
   -- instanciacao da unidade de controle
   control1 : control
   port map (
    data => control_in,
    clk => clk_control,
    reg1 => reg1_addr,
    reg2 => reg2_addr,
    reg3 => reg3_addr,
    dataout => regset_in1,
    dataout2 => shifter_in,
    ctrl_ula => ctrl_ula,
    ctrl_reg => ctrl_regset,
    ctrl_reg2 => ctrl_regset2,
    ctrl_branch => ctrl_branch,
    clk_ula => clk_ula,
    clk_reg => clk_regset
   );

   -- instanciacao do banco de GPRs
   register_set1 : register_set
   port map (
    v1 => regset_in1,
    v_ula => result_ula,
    sel1 => reg1_addr,
    sel2 => reg2_addr,
    sel3 => reg3_addr,
    t => ctrl_regset,
    q => ctrl_regset2,
    clk => clk_regset,
    o1 => ula_in1,
    o2 => ula_in2,
    o3 => mar_in2,
    o4 => mdr_in
   );

   -- instanciacao da unidade logica e aritmetica
   ula1 : ula
  port map (
   op1 => ula_in1,
   op2 => ula_in2,
   sel => ctrl_ula,
   clk => clk_ula,
   result => result_ula,
   c => carry,
   z => zero,
   o => overflow
  );

end cpu;
```

AIDA-16: UM MICROPROCESSADOR DESTINADO AO ENSINO UTILIZANDO HARDWARE PROGRAMÁVEL

por

Rodrigo Bittencourt Motta

Trabalho de Conclusão apresentado aos Professores(as):

_____
Prof. Alexandro Magno dos Santos Adário

_____
Prof. Carlos Augusto Moreira dos Santos

_____
Prof. Luciano Lores Caimi

Vista e permitida a impressão.

Santo Ângelo, 03 de julho de 2004.

_____
Prof. Luciano Lores Caimi
Orientador