ANEM - Um microcontrolador didático de 16 bits baseado em MIPS

Abstract. This paper concerns the development of a Harvard 16 bits microcontroller based in the MIPS microprocessors. The microcontroller was developed entirely in VHDL and implemented in a Cyclone II FPGA donated by Altera. It has a reduced instruction set of constant size. This last characteristic facilitaded the inclusion of a pipeline with f ve stages. Peripherals are disposed in a modular way that permits easy addition or deletion of new elements using a simple mapping of data memory.

Resumo. Este artigo trata do desenvolvimento de um microcontrolador de 16 bits de arquitetura Harvard e baseado nos processadores da família MIPS. O microcontrolador foi desenvolvido inteiramente em VHDL e implementado numa FPGA Cyclone II cedida pela Altera. O processador possui um conjunto reduzido de 21 instruções de tamanho f xo. Esta característica facilitou o desenvolvimento de um pipeline de cinco estágios. Os periféricos estão dispostos de forma modular permitindo fácil adição ou remoção destes elementos através de simples mapeamente dos registradores na memória de dados.

1. Introdução

Com as mudanças pelas quais vem passando a eletrônica digital nas últimas decadas, o VHDL vem ganhando cada vez mais força. O uso de ferramentas de projeto atuais, CPLDs e FPGAs, permite aos engenheiros progredirem do conceito ao silício funcional com muita rapidez. Além disso, os microcontroladores passaram a dominar muitas aplicações, substituindo os componentes padrão SSI e MSI. [8] Atualmente, praticamente todos os elementos digitais de um projeto, incluindo o próprio microcontrolador, podem ser concentrados numa única FPGA, permitindo obter consideráveis melhorias em relação a desempenho, conf abilidade, energia e tamanho. A capacidade dos FPGAs cresceu ao ponto de ser possível que um sistema multiprocessador completo possa ser sintetizado em um único dispositivo.

Esses fatores vêm estimulando a popularização dos microprocessadores conhecidos como *soft-core processor*, que é um processador completamente descrito em software, usualmente usando uma linguagem de descrição de hardware, com o VHDL entre as mais populares. Um *soft-core* é extremamente f exível, já que seus parâmetros podem ser facilmente trocados através de uma simples reprogramação do dispositivo [7]. Além disso, estes processadores embarcados em FPGAs podem servir como uma importante ferramenta didática para disciplinas de lógica programável em Engenharia Eletrônica ou da Computação, representando uma ponte entre teoria e prática. [2, 4, 6]

Atualmente existem vários *soft-cores* comerciais, destacando-se o Nios II® da Altera® [1] e o Micro Blaze® da Xilinx® [10]. Entretanto, esses *soft-cores* não permitem uma visão detalhada de sua implementação interna. Além disso, esses núcleos são de propriedade intelectual dos fabricantes, devendo ser sintetizados em suas respectivas plataformas. Do ponto de vista didático também não são interessantes, justamente devido às restrições de acesso ao código fonte. Este trabalho descreve detalhadamente o desenvolvimento de um *soft-core* simples, baseado na arquitetura dos processadores MIPS [5] e com características de microcontrolador. Batizado de Anem - acrônimo recursivo que significa *Anem não é MIPS*, o projeto pode servir ao propósito de ser uma ferramenta didática para cursos de projeto lógico de dispositivos programáveis ou sobre microprocessadores.

2. Arquitetura focada no uso didático

O ANEM possui arquitetura Harvard, com memória de instruções e memória de dados separadas, simplificando seu uso como microcontrolador e diferenciando-o um pouco do MIPS. Possuindo também, um conjunto RISC de instruções, com 21 instruções, sendo o tamanho das palavras (tanto instruções como dados) de 16 bits. Mesmo número de registradores internos, nomeados de $0 a $15, tendo o $0 sempre o valor fixo de zero (*hardcoded*), podendo ser utilizado para testes em *loops* ou para a limpeza do conteúdo de um outro registrador.

O microcontrolador foi implementado sem pilha em hardware, quando chamada uma sub-rotina o valor do *program counter* (PC) é salvo no registrador $15. Cabe ao programador salvar este valor de retorno em um novo registrador ou, mais indicado, guardar os valores numa pilha em software na memória de dados. Fica incubido também o programador de verificar quais registradores precisam ter seus valores salvos quando chamada uma sub-rotina e quais irão retornar valores quando necessário.

As simplificações realizadas na arquitetura permitem uma fácil compreensão do ANEM por estudantes de cursos introdutórios na área. Aliado ao fato de que o código é aberto (todo o projeto está disponível em *http://code.google.com/p/anem-tg*), o processador desenvolvido pode ser utilizado como ferramenta didática de formas diversas: desenvolvimento de novos dispositivos periféricos, aperfeiçoamento do código existente, desenvolvimento de co-processadores, monitoramento dos sinais internos durante a execução de programas ou até mesmo servindo como base para o desenvolvimento de outros processadores nele baseados.

3. O Conjunto de instruções

Assim como o MIPS em que foi baseado, o ANEM possui um conjunto de instruções que englobam instruções aritméticas, de salto e tomada de decisão, de deslocamento e de transferência de dados. Entretanto, o ANEM possui um conjunto ainda mais reduzido de instruções (apenas 21 instruções precisam ser aprendidas). A seguir, serão detalhados os pontos mais relevantes do projeto do conjunto de instruções.

3.1. Instruções Aritméticas

O MIPS tem como uma característica básica o fato de todas as instruções aritméticas realizarem apenas uma operação e sempre necessitarem de três registradores. Mantendo a característica do MIPS de ter as instruções sempre com o mesmo tamanho, facilitando

o controle, foi preferível reduzir o número de registradores indicados por uma instrução para dois, assim como encontrado na família 8086.

Por exemplo, na instrução de soma de duas variáveis do ANEM, o conteúdo do registrador $a, é somado com o conteúdo do registrador $b e a soma é então guardada no registrador $a. Uma breve descrição das instruções aritméticas pode ser vista na tabela 1, onde a e b são números de 0 a 15.

Tabela 1. Instruções Aritméticas

Instrução	Assembler	Descrição
ADD	ADD $a , $b	Soma o conteudo dos registradores e guarda o valor em $a
SUB	SUB $a , $b	Subtrai o conteudo dos registradores e guarda o valor em $a
AND	AND $a , $b	Faz o E lógico bit a bit entre os dois registradores e guarda o valor em $a
OR	OR $a , $b	Faz o OU lógico bit a bit entre os dois registradores e guarda o valor em $a
XOR	XOR $a , $b	Faz o OU exclusivo lógico bit a bit entre os dois registradores e guarda o valor em $a
NOR	NOR $a , $b	Faz o Não-OU bit a bit entre os dois registradores e guarda o valor em $a
SLT	SLT $a , $b	Se $a for menor que $b, $a recebe o valor 1, e 0 caso contrário

Como dito, as instruções no ANEM tem um tamanho fixo de 16 bits, isso implica que para os diferentes tipos de instrução seja necessário uma divisão adequada dos bits e suas funções. As instruções aritméticas são chamadas do tipo R (trabalha com registradores). A divisão dos bits para a instrução tipo R pode ser vista na figura 1.

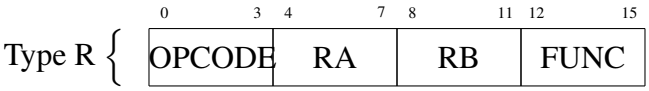

Figura 1. Formato tipo R

Nas instruções tipo R temos quatro campos com 4 bits cada. O campo OPCODE indica a operação básica da instrução, no caso indica ao controle que é uma operação aritmética que utilizará a ULA (Unidade Lógica Aritmética). A variante específica da instrução é definida no campo FUNC. Nos campos RA e RB é indicado o endereço dos registradores que serão utilizados na instrução. Na tabela 2 é possível ver os OPCODEs e FUNCs das funções aritméticas.

3.2. Instruções de Deslocamento

O Anem16 tem diferentes instruções de deslocamento de bits em um registrador, que ajudam o programador a simplificar a lógica do programa dependendo do deslocamento necessário. Na tabela 3 está uma breve descrição das suas funções.

As instruções de deslocamento tem o formato S (*Shift*) como mostrado na figura 2. Este formato também possui os campos OPCODE e FUNC para indicar que instrução está sendo passada. O registrador RA indica o endereço do registrador sobre o qual vai ser

Tabela 2. Mapeamento das Instruções Aritméticas

Instrução	Tipo	OPCODE	FUNC
ADD	R	0000	0010
SUB	R	0000	0110
AND	R	0000	0000
OR	R	0000	0001
XOR	R	0000	1111
NOR	R	0000	1100
SLT	R	0000	0111

Tabela 3. Instruções de Deslocamento

Instrução	Assembler	Descrição
SHL	SHL $a , SHAMT	Deslocamento lógico do registrador $a para esquerda o número de posições indicadas em SHAMT
SHR	SHR $a , SHAMT	Deslocamento lógico do registrador $a para direita o número de posições indicadas em SHAMT
SAR	SAR $a , SHAMT	Deslocamento aritmético do registrador $a para direita o número de posições indicadas em SHAMT
ROL	ROL $a , SHAMT	Rotaciona o registrador $a para esquerda o número de posições indicadas em SHAMT
ROR	ROR $a , SHAMT	Rotaciona o registrador $a para direita o número de posições indicadas em SHAMT

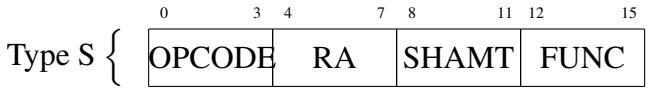

Figura 2. Formato tipo S

efetuado o deslocamento. A diferença para as funções do formato R é o campo SHAMT (*SHift AMounT*) indicando o número de deslocamentos a serem efetuados no registrador.

Na tabela 4 estão os OPCODEs e FUNCs para as instruções de deslocamento.

Tabela 4. Mapeamento das Instruções de Deslocamento

Instrução	Tipo	OPCODE	FUNC
SHL	S	0001	0010
SHR	S	0001	0001
SAR	S	0001	0000
ROL	S	0001	1000
ROR	S	0001	0100

3.3. Instruções de Transferência de Dados

O acesso à memória de dados é feito pelas funções SW, *Store Word*, e LW, *Load Word*. Na instrução SW é indicado primeiramente o endereço do registrador em que está a palavra

a ser gravada na memória, seguido do registrador que indica o endereço da memória de dados em que será gravado o dado e um valor constante complemento a dois (*offset* a ser somado ao endereço), como mostrado abaixo.

- SW $a, OFFSET($b)
- LW $a, OFFSET($b)

A função LW salva em um registrador passado como parâmetro, $a, a palavra guardada no endereço passado por um segundo registrador, $b. Essas instruções utilizam o formado W (*Word*) onde temos os campos OPCODE, indicando uma instrução de salto, um registrador para manipular o dado, RA, e RB indicando um endereço da memória de dados. Os últimos quatro bits são utilizados para gerar um OFFSET, em complemento a dois, no endereço contido no registrador $b. Esse formato é mostrado na figura 3.

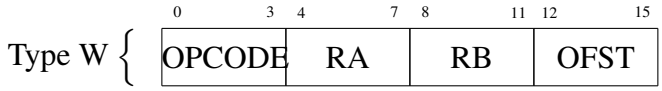

Figura 3. Formato tipo W

Para ser possível a inicialização de um registrador com um valor são utilizadas duas funções, LIU e LIL. Como um registrador guarda uma palavra de 16 bits, mesmo tamanho de uma instrução, foi preferível dividir o carregamento do imediato em duas funções. A função LIL, *Load Immediate Low*, escreve no registrador $a os 8 bits menos significativos, e a função LIU , *Load Immediate Upper*, escreve os 8 bits mais significativos.

- LIL $a,IMEDIATO
- LIU $a,IMEDIATO

As funções para o carregamento de um imediato tem o formato L, mostrado na figura 4. O formato L possui três campos, o OPCODE, indicando o tipo de função, o endereço do registrador em que será carregado o imediato, RA, e um número de 8 bits no campo BYTE. Na tabela 6 é mostrado os OPCODES das funções de transferência de dados.

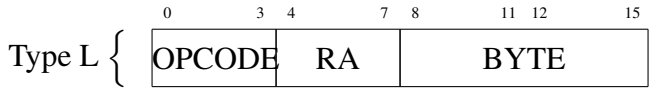

Figura 4. Formato tipo L

Tabela 5. Mapeamento das Instruções de Transferência de Dados

Instrução	Tipo	OPCODE
LW	W	0100
SW	W	0101
LIU	L	1100
LIL	L	1101

3.4. Instruções de Salto e Desvio Condicional

A instrução de salto, J (*Jump*) possui o endereçamento mais simples possível junto com a função de chamada de sub-rotina JAL e a função HAB. Elas utilizam o formato de instrução J (ver figura 5), que consiste em 4 bits para o campo de operação e o restante dos bits para o campo de endereço.

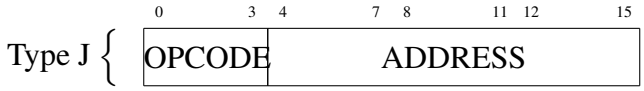

Figura 5. Formato tipo J

Na instrução de salto J temos apenas 12 bits para o endereço e é feita a concatenação dos quatro bits mais significativos do *Program Counter* com os 12 bits indicados pela instrução. O assembly da instrução pode ser visto a seguir.

- J LABEL

A diferença entre JAL e J é que a JAL guarda o endereço da instrução antes do salto, possibilitando então, o retorno a este endereço após a sub-rotina ser atendida. O endereço é salvo automaticamente no registrador $15 e é necessário salvá-lo em uma pilha em *software*. pois se chamada uma instrução JAL antes do retorno de uma sub-rotina o valor armazenado no registrador $15 será perdido.

- JAL LABEL

A função HAB, embora seja do do formato J, não é uma função de salto, ela apenas habilita as interrupções. Sempre que uma interrupção é chamada, ou uma sub-rotina é chamada, as interrupções do Anem são desabilitadas. Cabe ao programador habilitá-las através desta instrução quando achar conveniente. Como mostrado abaixo a instrução não possui nenhum operando ou endereço.

- HAB

Ao contrário das instruções de salto, as instruções de desvio condicional necessitam especificar dois operandos além do endereço de desvio. Por isso é preciso um formato diferente para essas instruções sendo o formato utilizado o W. A função BEQ (*Branch on equal*) é a instrução de salto condicional implementada no ANEM. Como o indicado no seu nome, o salto é efetuado quando os dois registradores passados para instrução, $a e $b, tem seus valores iguais. O salto é realizado pelos 4 bits do campo *Offset*, interpretado como complemento a dois, podendo, então, efetuar saltos de 7 instruções para frente ou 8 instruções para atrás.

- BEQ $a,$b, IMEDIATO

Para ser possível o retorno de uma sub-rotina foi criada a função JR. Ela consiste em um salto para o endereço guardado em um registrador passado como parâmetro para função como visto abaixo.

- JR $a

Como é necessário o endereço do registrador é utilizado o formato W para a instrução JR, sendo apenas o registrador RA passado como parâmetro, os demais campos não são verificados pelo controle. Os OPCODEs das instruções de salto de desvio podem ser vistos na tabela 6.

Tabela 6. Mapeamento das Instruções de Salto e Desvio Condicional

Instrução	Tipo	Opcode
J	J	1000
JAL	J	1001
HAB	J	1111
JR	W	0111
BEQ	W	0110

4. *Pipeline*

Para promover uma melhora de desempenho no ANEM, a execução das instruções foi dividida em cinco etapas - *Instruction Fetch* (IF), *Instruction Decode* (ID), *Execute* (EX), *Memory* (MEM) e *Write Back* (WB) - num esquema similar ao descrito para a arquitetura MIPS em [5]. Desta forma, é possível executar cinco instruções simultâneamente, cada instrução podendo levar até cinco ciclos do relógio (*clock*) para concluir, permitindo que o processador alcance a taxa de uma instrução completada a cada ciclo do relógio (em condições ideais - dependendo da organização do código).

O grande ganho de desempenho em relação a um esquema de execução monocíclico - que executa invariavelmente uma instrução a cada ciclo de *clock* - reside no fato de que a quebra do *datapath* (o caminho que os dados precisam seguir) em cinco etapas possibilita um grande aumento na frequência máxima suportada pelo processador. Ora, se o caminho agora está dividido em cinco partes e os dados precisam apenas percorrer uma das partes a cada ciclo do relógio, o relógio poder operar a frequências muito superiores ao caso em que os dados precisam percorrer todas as cinco partes a cada período. Sendo assim, grandes ganhos de desempenho foram obtidos graças ao esquema mostrado na figura 6 e detalhado a seguir.

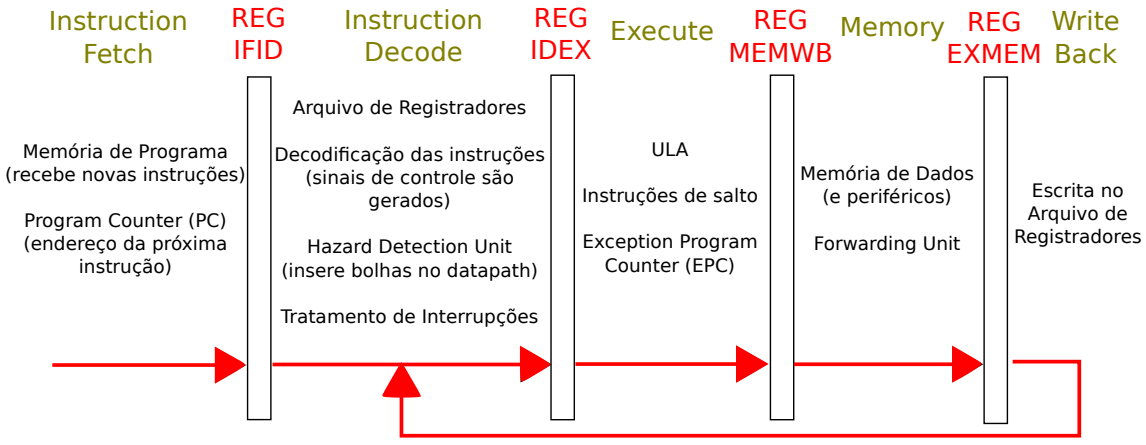

Figura 6. Pipeline no ANEM

No primeiro estágio do *datapath*, Instruction Fetch, uma nova instrução é lida da memória ROM no endereço indicado por PC (o registrador *Program Counter*). Ao final desta etapa, a nova instrução é salva no registrador IF/ID e o PC é incrementado e também salvo neste registrador intermediário.

Em ID, a instrução em IF/ID é decodificada: são gerados todos os sinais de controle que serão necessários nas etapas seguintes e o arquivo de registradores é lido, entre outras ações. Ao final do ciclo, as informações geradas são salvas no registrador ID/EX.

Em EX, são realizadas operações na ULA, decodificação de endereços e operações de salto (que é realizada sobre PC ao final deste ciclo). Os dados gerados nesta etapa continuam o caminho através do registrador EX/MEM.

Em MEM são realizadas operações de escrita e leitura de dados na memória RAM. E, finalmente, em WB, resultados são gravados de volta no arquivo de registradores. Como o arquivo de registradores também é usado na etapa ID, optou-se por realizar a operação de escrita na borda de descida do clock, enquanto as operações de leitura são feitas na borda de subida.

O conjunto de instruções do ANEM herdou do conjunto de instruções do MIPS uma série de características amigáveis à execução em pipeline. Primeiramente, o fato de todas as instruções possuírem a mesma largura facilita bastante a implementação do estágio IF. Além disso, existem apenas alguns poucos formatos de instruções, com os campos dos registradores sempre na mesma posição. Isto permite que, em ID, o arquivo de registradores seja lido ao mesmo tempo em que o tipo de instrução é determinado. Outro ponto importante é que operandos de memória só aparecem nas instruções de *loads* e *stores*. Isto permite que o próprio estado EX seja usado para calcular endereços de memória e apenas depois, em MEM, o acesso seja efetivamente realizado. Ainda em relação à memória, os dados precisam estar alinhados, sempre em palavras de 16 bits, permitindo que o acesso sempre possa ser feito de uma só vez.

Apesar de efetivamente boa e amplamente utilizada, a solução do pipeline introduz uma série de problemas de conflito, *hazards*, entre as instruções que executam simultaneamente nos diversos estágios. Entretanto, graças ao conjunto de instruções bem projetado para esse fim, esses problemas foram contornados eficientemente em *hardware* em troca de um certo aumento de complexidade e uma pequena perda de desempenho em relação ao quadro ideal de uma instrução por ciclo de clock.

4.1. *Hazards*

Podem ocorrer situações no pipeline em que a próxima instrução não pode ser executada imediatamente após a anterior. Essas situações caracterizam os *hazards*, que podem ser classificados em três diferentes tipos: estruturais, de dados e de controle. Os *hazards* estruturais ocorrem quando o próprio hardware não comporta a execução concomitante de duas instruções quaisquer. Por exemplo se um processador possui memória compartilhada entre dados e instruções, em um momento em que o estágio MEM (referência à memória) e o estágio IF (busca uma instrução) ficam sobrepostos no pipeline, ocorre um *hazard* estrutural. Este primeiro tipo não ocorre no ANEM, graças principalmente a sua arquitetura de memória Harvard. Os *hazards* de dados ocorrem quando um passo precisa aguardar que outro termine para que possa ter acesso a dados atualizados no arquivo de registradores. O terceiro tipo de *hazard*, chamado de controle, surge da necessidade de tomar uma decisão de desvio baseado nos resultados de instruções que ainda estão executando.

Quanto aos *harzards* de dados no ANEM, uma instrução pode modificar um determinado registrador durante o estágio EX, mas a modificação só será salva no último

estágio WB. Ou seja, neste caso, ocorre um atraso de até 3 ciclos de clock para que o registrador seja efetivamente alterado no arquivo de registradores. Neste meio tempo, outras instruções estarão executando em outros estágios e poderão requerer este valor já atualizado para que possam utilizá-lo.

Uma primeira estratégia para minimizar o problema é através de um pequeno detalhe no projeto do arquivo de registradores: a escrita e leitura de registradores são realizadas em momentos distintos, sendo a escrita realizada na primeira metade do ciclo do relógio e a leitura realizada na segunda metade. Isto já elimina o problema no caso em que uma instrução precisa do registrador atualizado exatamente no momento em que a instrução que o modificou está executando em WB, já que no momento da leitura o registrador já estará corretamente modificado. Entretanto, mesmo com essa atenuação do problema, ele continua sendo de grande relevância.

A sequência de instruções mostrada na figura 7 será útil para ilustrar o problema. Parece óbvio que a quinta instrução não deveria executar nunca (lembre-se que $0 é sempre nulo no ANEM). Entretanto, se nada for feito para evitar os *data hazards* a instrução poderá sim ser executada. O problema é justamente que todas as instruções estão usando o registrador $2. Veja na figura 7 que quando a segunda instrução chega ao estágio ID (acesso ao arquivo de registradores para leitura) a primeira instrução ainda está no estágio EX. Para que o valor acessado pela segunda instrução estivesse correto, a primeira instrução precisaria estar no estágio WB, 2 ciclos de relógio depois. O mesmo problema ocorre entre as outras instruções do exemplo.

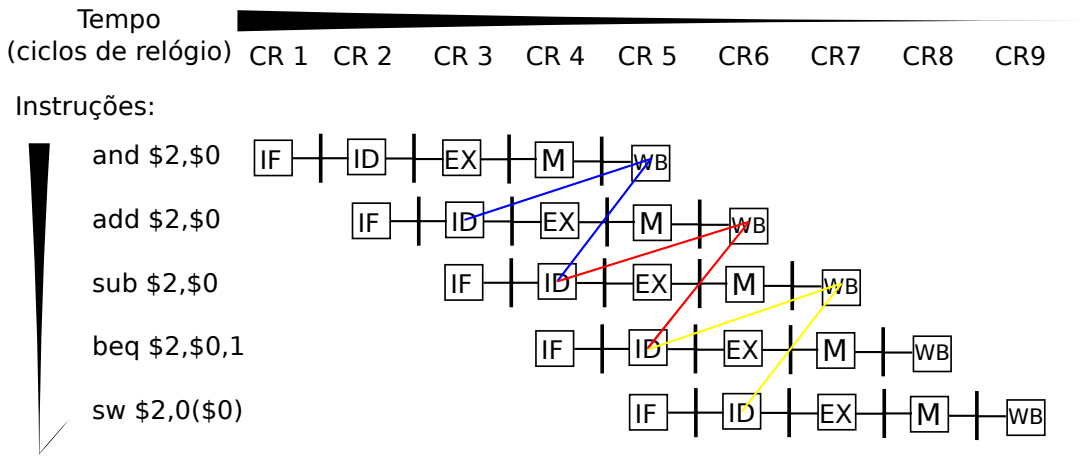

Figura 7. Dependência entre cinco instruções executando no pipeline ao mesmo tempo

Este problema poderia ser evitado se o programador (ou compilador) usasse instruções *nop* (*no operation*) entre as instruções conflitantes para que o estágio WB de uma instrução coincida com o estágio ID da seguinte. Esta estratégia, no entanto, aumenta bastante o tempo de execução do programa e ainda exige que o programador (ou compilador) esteja preparado para a possibilidade de conflito. Para corrigir o segundo

problema, basta que o próprio processador detecte os conflitos e insira bolhas[1] (equivalentes às instruções *nop*) automaticamente entre uma instrução e outra. Isto não resolveria, entretanto, o mais importante: o tempo de execução continua sendo muito penalizado.

É fácil notar através da figura 7 que o valor requerido do registrador $2 já está disponível no pipeline nos momentos em que é necessário, apenas não tendo sido gravado de volta no arquivo de registradores. Uma solução mais eficiente, então, seria criar atalhos ao longo do *datapath* que permitam que as instruções sempre tenham acesso aos dados mais atualizados possível, mesmo que não tenham ainda sido gravados de volta. Esta técnica é conhecida como *forwarding* ou *bypassing* e é amplamente utilizada no ANEM, através de uma unidade lógica descrita em *forwarding_unit.vhd* presente no estágio EX do pipeline.

Em certas situações, porém, a técnica não consegue evitar o *hazard*. Considere o caso de uma instrução tenta ler um registrador imediatamente após uma instrução que lê um dado da memória e escreve neste mesmo registrador. O dado ainda estaria sendo lido da memória enquanto a instrução seguinte já estivesse no estágio EX. Neste caso, o *pipeline* precisa ser atrasado em uma instrução para que em seguida a técnica de *forwarding* consiga resolver o problema. Desta forma, o processador precisa detectar situações deste tipo e inserir bolhas na linha de execução.

Para resolver o terceiro tipo do problema, os *control hazards*, o ANEM sempre insere bolhas na linha de execução logo após as instruções de salto (seja condicional ou não). A execução é atrasada até que o endereço da próxima instrução a ser resgatada possa ser determinado corretamente.

4.2. *Forwarding* no ANEM

Para ilustrar os casos em que é necessária a realização de *Forwarding* será necessário definir uma notação que nomeie os dados e campos de instrução ao longo do *datapath*. Escolhemos chamar os registradores de *pipeline* usando as siglas dos dois estágio que dividem. O registrador entre os estágios *Instruction Fetch* e *Instruction Decode*, por exemplo, será chamado *IFID*. Além disso, o dado referido presente num determinado registrador será apresentado logo após o nome deste, usando um ponto. Desta forma, se quisermos nos referir ao registrador *ra* presente no estágio Execute, usamos *IDEX.ra*.

Além dos diversos caso que provocam a ação de *Forwarding* existem também diversos tipos diferentes desta ação. O caso que envolve mais possibilidades e ocorre mais frequentemente está no estágio *Execute*, onde os dados *IDEX.ra* e *IDEX.rb* (ou apenas um de seus bytes) podem precisar ser trocados por versões mais recentes presentes nos estágios *Memory* e *Write Back*. A partir disso, pode-se contar 11 possibilidades de origem mais recente para os dados *IDEX.ra* e mais 11 possibilidades para os dados em *IDEX.rb* (para mais detalhes, consultar o arquivo *forwarding_unit.vhd* e também *anem16.vhd*).

O segundo caso de *Forwarding*, mais simples e menos frequente, ocorre no estágio *Memory*, onde os dados mais recentes podem estar apenas no estágio *Write Back* e também apenas se o problema já não tiver sido resolvido quando os dados passavam pelo estágio *Execute*. O único caso que provoca isso é quando precisa-se gravar na memória

[1] Chamadas bolhas no sentido de serem um vazio no fluxo de instruções.

um dado que foi lido da memória num ciclo imediatamente anterior, pois os dados provenientes da memória ainda não estavam disponíveis em *Execute*.

5. Memória e periféricos

A arquitetura Harvard de memória usada no ANEM impede que ocorram *hazards* estruturais na execução em pipeline do ANEM. Esta simplificação de projeto foi o principal motivo que levou a adoção desse tipo de memória, ao contrário do que é usado no MIPS, onde dados e programa compartilham a mesma memória física.

Na memória, as palavras possuem 16 bits e precisam estar alinhadas. Isto permite que o processador consiga endereçar até 128 KB para programa e mais 128 KB de dados (512 KB usando bancos de memória, como explicador a seguir). Entretanto, nem toda a faixa de memória de dados está disponível para uso geral, uma faixa de endereço que começa em 0xFFD0 e se estende até 0xFFFF corresponde a registradores de periféricos, fisicamente localizados fora da memória RAM.

5.1. Memória de Dados

Existe uma entidade no ANEM chamada *RamPerifController*, descrita no arquivo *controladorRAM.vhd* que é responsável por atender qualquer requisição de acesso a memória (de dados) vinda do processador. Esta unidade decodifica o endereço recebido para determinar se a requisição deve ser encaminhada à memória RAM de dados ou à entidade *perifericos* (descrita em *perifericos.vhd*) que centraliza todos os periféricos e a unidade de interrupções do processador.

Quando um endereço encontra-se na faixa abaixo de 0xFFD0, os sinais são encaminhados a um chip externo de memória SRAM. Neste chip (IS61LV25616 da *ISSI*®), a memória está organizada em palavras de 16 bits endereçadas por 18 bits. Como o ANEM apenas consegue endereçar usando no máximo 16 linhas de endereço, os bits mais significativos do endereço são acessados através de um registrador especial mapeado na memória de periféricos (técnica conhecida como bancos de memória). Os registradores de periféricos podem ser acessados a partir de qualquer um dos 4 bancos de memória.

O IS61LV25616 usa apenas um único barramento tanto para entrada como para saída de dados, sendo necessário então o uso de elementos com lógica tri-state (incorporando a alta impedância como o terceiro estado). O VHDL permite o uso de alta impedância a partir do pacote *ieee.std_logic_1164*. Caso o endereço encontre-se na faixa de periféricos, a entidade *perifericos* torna-se responsável por decodificá-lo e encaminhar os sinais necessários ao periférico referenciado.

5.2. Memória de Programa

Como o *chip* de memória *flash* disponível (um S29AL032D da *Spansion*®) possui um barramento de dados de apenas 8 bits e permite frequências de operação de apenas cerca de $14\ \mathrm{MHz}$ (muito abaixo da frequência de operação do processador), optou-se por utilizar os bits de memória RAM disponíveis na própria FPGA para sintetizar a memória de programas, permitindo uma frequência igual à do microcontrolador e possibilitando o uso de um barramento de 16 bits. Como desvantagem, temos que os dados precisam ser regravados a cada vez que é retirada a alimentação da FPGA. Entretanto, esta abordagem serve bem ao propósito didático ou de testes.

5.3. Programador

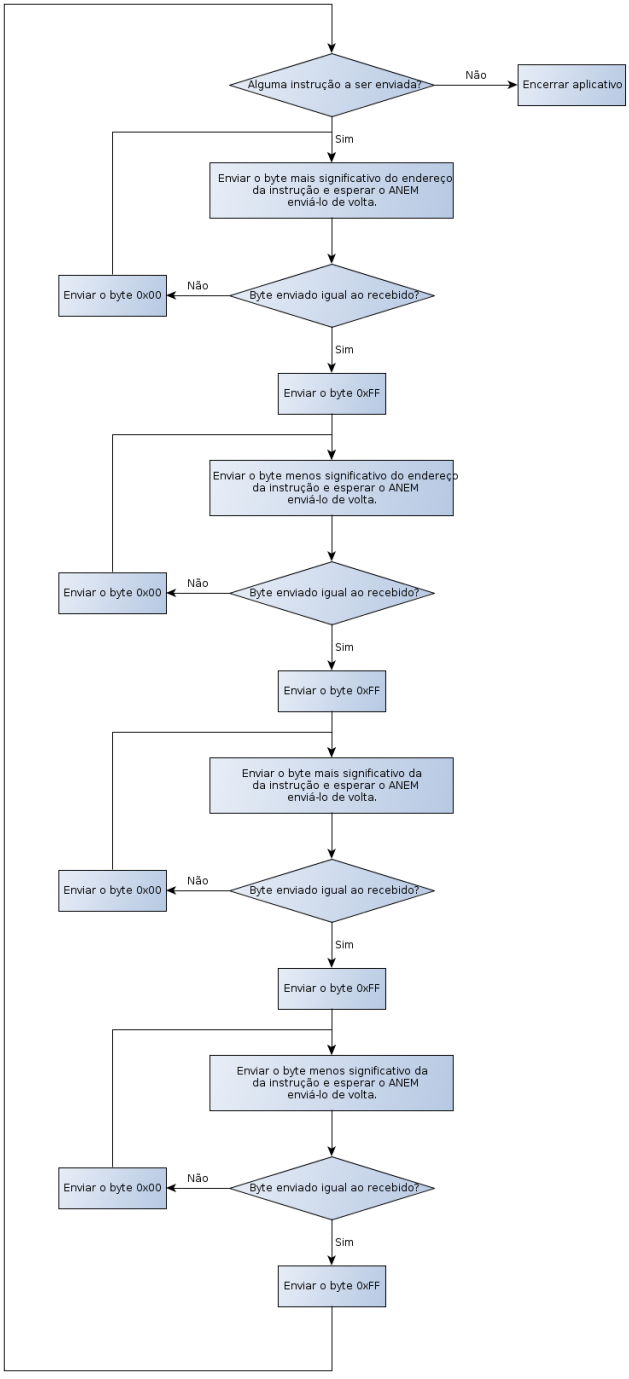

Figura 8. Algoritmo de programação do ANEM

O ANEM possibilita que o conteúdo da memória de programas seja alterado através da porta serial. Para isso, o programador deve estar habilitado (pino *Prog* deve ser mantido em nível lógico alto), e algum software precisa enviar o código já em formato

binário seguindo o algoritmo esquematizado na figura 8. Foi desenvolvido um programa em *Processing* [3] para executar a função de programar o ANEM. Este programa possui executáveis para Linux, Mac e Windows.

6. Unidade Lógica Aritmética

A ULA (Unidade Lógica Aritmética) é responsável por realizar diretamente as seguintes instruções do ANEM:

- ADD $a, $b
- SUB $a, $b
- AND $a, $b
- OR $a, $b
- XOR $a, $b
- NOR $a, $b
- SLT $a, $b
- SHL $a, valor_imediato
- SHR $a, valor_imediato
- SAR $a, valor_imediato
- ROR $a, valor_imediato
- ROL $a, valor_imediato

Onde as primeiras sete operações são do tipo R e as últimas cinco são do tipo S. Além disso, a ULA também deve informar sempre que o resultado de uma subtração for zero, pois este sinal é necessário à operação de BEQ. Além disso, a unidade pode também ser usada secundariamente em outros casos, como no cálculo do deslocamento provocado pelo *Shamt* (*Shift Amount*) sobre os endereços em algumas instruções.A figura 9 mostra simplificadamente como está inserida a ULA no ANEM.

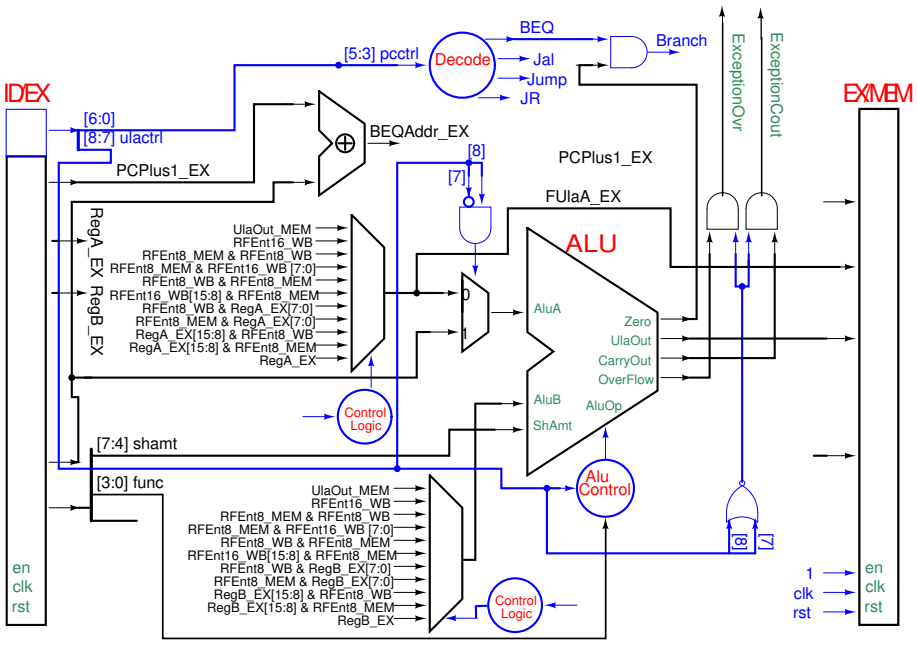

Figura 9. Esquema simplificado mostrando como a ULA está inserida no estágio EX do ANEM

A entidade da ULA descrita em *mips_alu.vhd* possui 4 entradas e 4 saídas. Como entradas possui *ula_a*, *ula_b* (os operandos), *ula_op* (sinal de controle) e *ula_shamt* (quantidade de deslocamentos). Como saída possui os sinais *ula_zero* (indica se o valor de uma soma ou subtração resultou em zero), *CarryOut*, *Overflow* e *ula_out* (resultado). Na tabela 7 são determinados os códigos para controle da operação operação da ula (entrada *ula_op*).

Tabela 7. Códigos para operação da ULA.

alu_op	operação
00000	AND $a, $b
00001	OR $a, $b
00010	ADD $a, $b
00110	SUB $a, $b
00111	SLT $a, $b
01100	NOR $a, $b
10001	SHR $a, valor_imediato
10010	SHL $a, valor_imediato
10000	SAR $a, valor_imediato
10100	ROR $a, valor_imediato
11000	ROL $a, valor_imediato
11111	XOR $a, $b

Para implementar uma ULA deste tipo, existem várias possibilidades de arquiteturas. A implementação realizada para o projeto ANEM visa principalmente o desempenho, usando um esquema de previsão de *carry* (*carry lookahead*). Pois na arquitetura com *pipeline*, quando menor for o tempo máximo que um sinal leva para percorrer um determinado estágio, maior poderá ser a frequência do *clock*. Tratamos da soma pois é a mais demorada das instruções, junto com a subtração que é feita pelo mesmo circuito.

6.1. Sistema de Previsão de Carry

Um dos pontos chaves para se obter uma boa velocidade de soma é conseguir prever o *carry* de um bit de alta ordem de forma mais rápida do que simplesmente esperando que ocorram todas as somas nos bits de ordem mais baixa. A idéia é fazer com que o sinal de *carry* passe por menos portas lógicas em série do que num sistema de *carry* normal (ou *ripple carry*, onde o *carry in* de uma soma é o *carry out* da soma anterior). Em contrapartida, um sistema de previsão de *carry* exigirá muito mais portas lógicas trabalhando em paralelo e ainda um maior nível de complexidade e menor nível de generalidade para descrever o sistema.

O sistema de previsão de *carry* adotado na ULA do ANEM é composto por dois níveis de abstração: o somador completo é composto por quatro somadores menores de 4 bits cada. Cada um dos somadores menores possui um sistema de previsão de *carry*. O sistema de previsão de *carry* do somador completo, por sua vez, considera os somadores menores como blocos fundamentais. Esta estratégia de utilizar mais de um nível de abstração permite reduzir bastante a quantidade de *hardware* utilizada em troca de uma certa perda de desempenho.

7. Interrupções e exceções no ANEM

Quando ocorre uma interrupção, a unidade de interrupções desabilita a ocorrência de outras interrupções e envia um comando para a unidade de controle do pipeline. Neste comando, estão as seguintes informações: se a interrupção foi gerada por uma exceção e o endereço de pulo para tratar esta interrupção. Ao receber estes sinais, a unidade de controle (presente no estágio ID do pipeline) pausa a execução do programa (da mesma forma que é feita para evitar os *control hazards*) e insere uma instrução JAL no *datapath*. O endereço de pulo é aquele fornecido pela unidade de interrupções e o endereço de retorno é fornecido por PC ou EPC (nas exceções), como descrito a seguir. As interrupções apenas podem ser reabilitadas por software, através da instrução *HAB*.

7.1. Exceções no ANEM

Exceções podem ser geradas pela ULA ao executar uma instrução de subtração (*overflow* e *borrow*) e adição (*overflow* e *carry out*). Como a ULA está localizada no estágio EX do *pipeline*, enquanto o PC (*Program Counter*) está localizado no estágio IF, tornou-se necessário a criação de um registrador especial para guardar o endereço da instrução que gerou a última exceção. Este registrador está localizado em EX e chama-se EPC (*Exception Program Counter*). Desta forma, sempre que ocorreu uma interrupção gerada por uma exceção aritmética, o endereço a ser salvo no registrador *$15* está disponível em EPC (ao contrário do que ocorre nos outros tipos de interrupções, quando o endereço a ser salvo está em PC.

Graças ao correto gerenciamento das exceções aritméticas através de interrupções no ANEM, é possível realizar operações sobre operandos de qualquer comprimento. Por exemplo, pode-se somar dois números de 32 bits fazendo duas somas separadas e, caso ocorra uma exceção de *carry out*, adicionando 1 no resultado mais significativo. Isto pode ser realizado sucessivamente para obter somas de comprimentos arbitrários.

7.2. A unidade de interrupções

A unidade interrupção recebe os sinais provenientes da ULA e dos periféricos indicando quando uma interrupção ou exceção ocorrer. Cabe a unidade de interrupção verificar se a interrupção ocorrida está habilitada e verificar sua prioridade. As interrupções que serão habilitadas são indicadas pelo programador através de um registrador acessado como uma posição da memória de dados. A ordem das interrupções no registrador de configuração é também a ordem de prioridade das interrupções, sendo então fixa.

Apenas nove tipos diferentes de interrupções tem tratamento previsto pelo ANEM, como o registrador de configuração das interrupções possui 16 bits, apenas os 9 primeiros bits são utilizados. Sendo possível uma fácil adição de outras interrupções a medida que sejam adicionados periféricos ou tratamento de outras exceções. O tratamento com maior prioridade são os das exceções de *Overflow* e *Carry out*, bits 0 e 1 respectivamente do registrador de configuração. Nos bits 2 ao 4 é indicando as interrupções do periférico UART, do 5 ao 7 os dos *Timers* e o bit 8 do MAC, como pode ser visto na tabela 8.

8. Interface para um coprocessador

O controle do pipeline do ANEM permite que instruções desconhecidas ao processador (que possuem opcodes ainda não utilizados) sejam encaminhadas para alguma entidade

Tabela 8. Registrador de Configuração das Interrupções

Bit	Interrupção
0	ocorreu *overflow*
1	ocorreu *carry out*
2	RX (UART)
3	TX (UART)
4	WRIF (UART)
5	ocorreu o estouro do TIMER 1
6	ocorreu o estouro do TIMER 2
7	ocorreu o estouro do TIMER 3
8	resultado pronto no MAC
9 ao 15	ainda sem função

externa. Além da instrução, também é disponibilizado à entidade externa a informação presente num registrador indicado pelo campo *ra* da instrução. Caso a entidade externa indique que não pode receber a instrução no momento, o pipeline fica travado até que a instrução possa ser despachada. Desta forma, deve-se manter o sinal de controle responsável por isso num nível lógico baixo caso não seja usado um coprocessador externo.

9. Periféricos

Ao processador descrito nas seções anteriores, foram adicionados alguns periféricos para que o mesmo pudesse então assumir um perfil de microcontrolador de uso geral. Como exemplo, foram desenvolvidos os seguintes periféricos: porta de comunicação serial assíncrona (UART), multiplicador e acumulador (MAC) e timers.

9.1. UART

A UART do Anem usa pilhas do tipo FIFO (First In, First Out) como buffers de transmissão e recepção. Esta solução usa a grande disponibilidade de bits de memória da FPGA (já que a memória de dados foi implementada num chip externo) e reduz o gargalo de desempenho provocado pela diferença de velocidade entre a UART (taxa de transmissão na faixa de *kilo* bits por segundo) e o processador (600 Mbps a 50 MHz). Desta forma o programador pode escrever muitos dados de uma só vez no buffer de transmissão sem precisar esperar que a transmissão termine para prosseguir com a execução de outras tarefas. Na recepção, a vantagem é semelhante: o usuário pode optar por esperar que uma grande quantidade de dados seja recebida e processar a informação apenas quando o buffer atingir um certo limite de espaço livre. Os pilhas foram implementadas usando uma mega função fornecida pela Altera para que desta forma a memória fosse implementada utilizandos os bits de RAM e não registradores das macro-células. Na compilação atual, foram usadas pilhas de 256 palavras de 9 bits. Os 8 bits menos significativos correspondem ao byte a ser enviado, o nono bit corresponde ao bit de paridade recebido ou o nono bit a ser enviado no caso da transmissão de palavras de 9 bits estar habilitada.

No Anem, a UART é vista e controlada através de 4 registradores de 16 bits mapeados na memória de periféricos nos endereços de FFDA até FFDD (valores em hexa). Palavras a serem enviadas devem ser escritas no registrador RegTXData (endereço FFDA),

apenas os 8 ou 9 bits menos significativos são considerados. Escrever neste registrador provoca uma operação de *push* na pilha de transmissão. Da mesma forma, as palavras recebidas podem ser lidas no registrador RegRXData (endereço FFDB), provocando uma operação de *pop* na pilha de recepção.

A taxa de transmissão da porta pode ser configurada de forma flexível através do registrador RegUARTBaud (endereço FFDD). O valor em binário escrito neste registrador indica por quanto a taxa de clock do processador será dividida para fornecer o *baudrate* da UART. Por exemplo, para se obter uma taxa de 9600 bits por segundo a partir de um clock de 50 MHz, deve-se escrever o valor 5208 neste registrador pois $5208 = \frac{50 \cdot 10^6}{9600}$.

Bits de configuração e flags estão disponíveis no registrador RegUARTCfg (endereço FFDC). Uma descrição resumida de todos os bits pode ser vista na tabela 9.

Tabela 9. Registrador de Configuração da UART

Bit	Nome	Significado
0	WRIF	*flag* de palavra recebida
1	RBE	*buffer* de recepção vazio
2	RBF	*buffer* de recepção cheio
3	TBE	*buffer* de transmissão vazio
4	TBF	*buffer* de transmissão cheio
5	TXIF	*flag* de erro de transmissão
6	RXIF	*flag* de erro de recepção
7	FERR	ocorreu erro de *framming*
8	OERR	ocorreu erro de *over run*
9	PBEN	habilitar transmissão de bit de paridade
10	CREN	habilitar recepção contínua de palavras
11	SREN	habilitar recepção de uma única palavra
12	TXEN	habilitar transmissão das palavras do *buffer*
13	RX9	habilitar recepção de um nono bit de dados
14	TX9	habilitar transmissão de um nono bit de dados
15	SPEN	habilitar a porta serial

9.2. MAC (Multiplicador Acumulador

O MAC implementado permite a escolha da multiplicação levando em consideração o sinal dos operandos, ou a multiplicação sem sinal. A configuração do MAC é feita através dos cinco registradores visíveis ao usuário através da Memória de Dados. Tendo 3 registradores para variáveis, **RegMacC**, **RegMacBu** e **RegMacBs**. O primeiro a registrador a ser gravado deve ser o **RecMacC**, pois quando gravado o valor um valor em **RegMacBu** ou **RegMacBs** a multiplicação é iniciada. Quando gravado um valor em **RegMacBu** a multiplicação ocorrerá sem sinal e quando gravado em **RegMacBs** com sinal.

O resultado da multiplicação é adicionado ao valor anterior dos Registradores RegMacA0 e RegMacA1, sendo a parte menos significativa e mais significativa do resultado anterior respectivamente. Esses registradores podem ser acessados tanto para lida e escrita, dessa forma, caso não seja mais interessante o valores anteriores dos registradores

de saída do MAC, é necessário apagar o conteúdo deles. Quando um resultado fica pronto é levantada uma *flag* gerando uma interrupção no microcontrolador caso esteja habilitada.

9.3. Timer

Foi implementado dois temporizadores de 16 bits, *Timer1* e *Timer2*, podendo ainda os dois trabalharem como um único de 32 bits. O controle é feito pelo usuário pelo registrador de configuração, os registradores de parada e os registradores de saída dos *Timers*. Ao ser escrito no registrador de configuração os temporizadores ativados irão começar a contagem e poderão ser paralisados por uma nova escrita no registrador de configuração que indique isso, ou quando terminarem a contagem que foram programados.

Os *timers* utilizam o *clock* que pode ser o mesmo da CPU ou uma divisão desse, através do *prescale* passado como parâmetro pelo registrador de configuração. O *prescale* é um número de p, de 4 bits, que indica por um fator N que divide o *clock* interno, sendo $N = 4^p$. Ainda é possível configurar se o *timer* irá contar até o estouro do seu contador de 16 bits ou até o número indicado em seu registrador de parada, os bits do registrador de configuração podem ser vistos na tabela 10.

Tabela 10. Registrador de Configuração do Timer

Bit	Nome	Significado
0	HAB32	Habilitar *Timer* único de 32 bits.
1	HAB1	Habilitar o *Timer1*
2	PART1	Habilitar registrador de parado do *Timer1*
3	-	Reservado para uso futuro
4-7	PRET1	*Prescale* para o *Timer1*
8	HAB2	Habilitar o *Timer2*
9	PART2	Habilitar registrador de parado do *Timer2*
10	-	Reservado para uso futuro
11-14	PRET2	*Prescale* para o *Timer2*
15	-	Reservado para uso futuro

10. Validando o ANEM

Para testar o correto funcionamento do Anem, foram escritos diversos programas em assembly. O exemplo mais relevante consiste no jogo Genius, baseado no clássico brinquedo lançado nos anos 80 como o primeiro jogo eletrônico vendido no Brasil [9]. Usando a placa de desenvolvimento DE2 da Altera®, o jogo foi executado pelo Anem usando os leds e botões.

Uma sequência aleatória é gerada pelo microcontrolador usando a duração em que o jogador aperta o botão. Isto foi conseguido usando o periférico *timer* (os 2 bits menos significativos capturados no momento em que o jogador solta o botão produzem um resultado aleatório). Os valores sorteados são gravados na memória e uma sequência é então gerada. Cabe ao jogador repetir a sequência usando os botões. A cada nova jogada um novo valor é sorteado, aumentando a sequência a ser repetida. Usando o periférico UART, uma comunicação RS232 foi utilizada para enviar comandos a um computador.

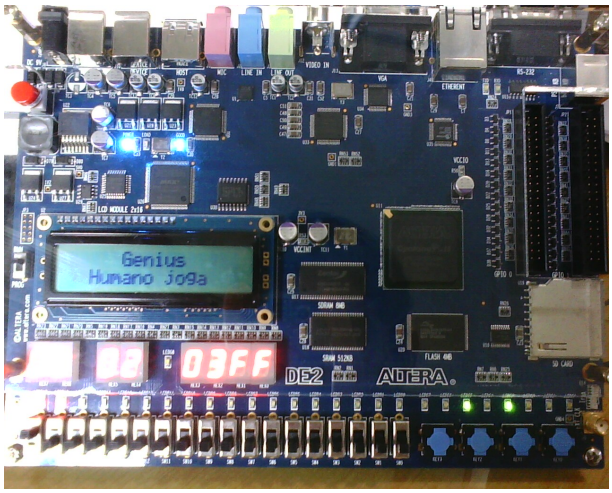

Figura 10. Genius executando na placa de desenvolvimento DE2®

Uma interface gráfica escrita em *Processing* pôde então ser utilizada a partir do PC. Pode-se ver a aplicação sendo executada na figura 10.

Com este exemplo simples, foi possível utilizar interrupções e periféricos: *timer* para gerar atrasos e para geração da sequência e a UART para comunicação com um computador. Para permitir a chamada de subrotinas aninhadas, foi utilizada uma pilha em software, usando a memória RAM e um registrador para apontar o topo da pilha. O código é composto de 699 instruções, em 723 linha de código, usando praticamente todas as instruções disponíveis e assim cumprindo o papel de validar o projeto do microcontrolador.

Outro exemplo utilizado para validar o Anem16 foi um programa para calcular o fatorial de um valor de 8 bits recebido através da porta serial. Foram usadas as exceções aritméticas para obter somas de números de 32 bits. E essas somas foram utilizadas iterativamente para obter multiplicações. Desta forma, pôde-se calcular corretamente o fatorial de valores menores ou iguais à 12.

11. Conclusões

Este trabalho apresenta o projeto de uma arquitetura de microcontrolador voltada principalmente para fins didáticos. Foi utilizada uma descrição em VHDL para que o dispositivo final fosse implementado numa FPGA Cyclone® II presente numa placa de desenvolvimento DE2® doada pela Altera®. A arquitetura surgiu a partir de um estudo da arquitetura de processadores MIPS e por isso compartilha com esta alguma características básicas.

Ao longo do projeto, optou-se por uma série de simplificações tendo em vista o escopo didático do projeto, uma destas simplificações ocorreu, por exemplo, na escolha por uma arquitetura *Harvard* de memória. Por outro lado, a adoção de algumas técnicas mais complexas, como a execução de instruções em *pipeline*, permitiram a exposição de estratégias de projeto atuais mais avançadas, como o uso da solução de *data forwarding*. Uma das grandes vantagens de uma descrição em VHDL é a possibilidade de uma fácil alteração do sistema. O sistema implementado permite, por exemplo, uma fácil

adição e remoção de dispositivos periféricos. Os três exemplos desenvolvidos (porta de comunicação serial, MAC e *timer*) puderam demonstrar esta facilidade do Anem.

Como melhoria futura, pode-se desenvolver um compilador de linguagem C para o conjunto de instruções do Anem. Também é interessante o desenvolvimento de um simulador para a arquitetura do ANEM. Essas duas melhorias poderiam ser integradas ao programador na forma de uma IDE de programação para o ANEM.

O conjunto reduzido de instruções (RISC) composto por 21 instruções simples e a arquitetura modularizada e descrita em VHDL em formato hierarquizado permite uma fácil compreensão de todo o projeto, tornando a arquitetura ANEM de microcontroladores uma interessante ferramenta no estudo de sistemas digitais. A abertura do código fonte permite que todo o projeto possa ser lido, compreendido, replicado e melhorado por profissionais e estudantes em busca de uma ponte entre a aprendizagem teórica e a implementação prática de um computador digital completo.

Referências

[1] Altera Coorporation. Nios ii embedded processor. Disponível on-line: http://www.altera.com/products/ip/processors/nios2/ni2-index.html. Visitado em Fevereiro de 2011.

[2] Edson Pedro Ferlin and Valfredo Pilla Júnior. Microprocessors: From theory to practice, a didactic experience. *Frontiers in Education. FIE 2004. 34th Annual*, 2004.

[3] Ben Fry and Casey Reas. Processing language. Disponível on-line: http://processing.org/about/. Visitado em Fevereiro de 2011.

[4] Koji Nakano and Yasuaki Ito. Processor, assembler, and compiler design education using an fpga. *14th IEEE International Conference on Parallel and Distributed Systems*, 2008.

[5] David A. Patterson and John L. Hennessy. *Computer Organization And Design*. Morgan Kaufmann Elsevier, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 4th edition, 2009.

[6] Murray Pearson, Dean Armstrong, and Tony McGregor. Design of a processor to support the teaching of computer systems. *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications*, 2002.

[7] Franjo Plavec. Soft-core processor design. Master's thesis, University of Toronto, 2004.

[8] Ronald J. Tocci, Neal S. Widmer, and Gregory L. Moss. *Sistemas Digitais*. Pearson Prentice Hall, São Paulo - SP, 10 edition, 2007.

[9] Varios. Genius. Disponível on-line: http://pt.wikipedia.org/wiki/Genius. Visitado em Setembro de 2011.

[10] Xilinx. Microblaze soft processor. Disponível on-line: http://www.xilinx.com/tools/microblaze.htm. Visitado em Fevereiro de 2011.