Desenvolvimento do Jogo Tetris para execução em processadores MIPS

Adarley L. G. Filho
Nur Corezzi
Paulo da C. Passos
Departamento de Ciência da Computação

Guilherme S. Lemos
Matheus A. Cerqueira
Rafael Lima
Departamento de Engenharia Elétrica

Abstract

O presente artigo descreve a implementação do jogo *multiplayer* Tetris em linguagem de programação assembly. Que tem como requisito ser compatível com os processadores MIPS uniciclo, multiciclo e pipeline desenvolvidos na disciplina de Organização e Arquitetura de computadores da Universidade de Brasília. Os processadores MIPS e o jogo são simulados por meio do hardware educativo FPGA Altera DE2-70 e os comandos do jogo são recebidos por meio do componente transceptor de infravermelho(IrDA) desta plana.

Keywords: DE2-70, assembly, tetris, MIPS, uniciclo, multiciclo, pipeline, organização de computadores, jogo, IrDA, NEC.

Author's Contact:

cunha.passos@gmail.com
guilhermelemos96@gmail.com
raffaellimma@gmail.com
adarleyunb@gmail.com
matheus.abrantesc@gmail.com
nurcorezzi@hotmail.com

1 Introdução

A evolução da tecnologia e a crescente demanda pelo uso de computadores, smartphones, tablets, entre outros, tem impulsionado os profissionais que trabalham na área de computação a buscarem soluções com maior desempenho, menor custo, maior portabilidade e um tamanho físico reduzido para componentes dessa área. Por isso, é fundamental para os desejam trabalhar nessa área conhecer os fundamentos da computação e como funciona a interação entre hardware e software, principalmente no que diz respeito ao processador.

Nesse sentido, o presente trabalho busca entender o processo de desenvolvimento de um jogo simples (Tetris), escrito em linguagem assembly MIPS*(Microprocessor without Pipeline Interlocked Stages)*, para ser jogado por no máximo quatro jogadores e ser executado em cada um dos três processadores MIPS (uniciclo, multiciclo e pipeline) desenvolvidos na matéria de Organização e Arquitetura de Computadores da Universidade de Brasília. Esses processadores foram desenvolvidos ao longo de vários semestres na universidade, foram escritos em linguagem Verilog por meio do software Quartus II e são simulados por meio da placa de desenvolvimento FPGA Altera DE2-70.

Além disso, desenvolveu-se ainda uma interface integrada a cada um dos processadores para receber e transmitir sinais infravermelho por meio do componente de hardware IrDA da Placa FPGA Altera DE2-70. Pois, neste trabalho, um dos requisitos do jogo era que os comandos dos jogadores sejam passados aos processadores por meio de sinais infravermelho emitidos por controles remotos que utilizam o protocolo de transmissão NEC.

2 Tetris

Tetris é um jogo eletrônico que foi desenvolvido por Alexey Pajitnov, Dmitry Pavlovsky e Vadim Gerasimov, e inspirado no quebra cabeças Pentaminó [Wikipédia 2016]. É um jogo mundialmente famoso, simples e de fácil jogabilidade, por isso foi escolhido para demonstrar o funcionamento do dispositivo de transmissão e recepção de infravermelho (IrDA) da placa DE2-70.

O objetivo do jogo é encaixar peças de diversos formatos, conhecidos como "tetraminós", que descem do topo de uma tela. Ao se encaixarem de forma que não reste mais espaços vazios em um determinada linha, tal linha desaparece e pontos são somados para o jogador. Esse jogador perderá o jogo caso as linhas incompletas se empilhem até o topo da tela do jogo.

3 Ferramentas Utilizadas

Nesse projeto as principais ferramentas utilizadas para o desenvolvimento e teste do jogo Tetris foram: O simulador assembly MIPS MARS; A placa de desenvolvimento FPGA ALtera DE2-70; E os Processadores MPIS uniciclo, multiciclo e pipeline desenvolvidos em linguagem verilog por meio do sotware Quartus II

3.1 MARS

O MARS *(MIPS Assembly and Runtime Simulator)* é um ambiente de desenvolvimento interativo (IDE) para programação em linguagem assembly MIPS, e tem foco educacional [Vollmar and Sanderson 2006]. Foi inicialmente desenvolvido pelo professor Ph.D. Kenneth R. Vollmar (Universidade Estadual de Missouri), e posteriormente pelo professor Dr. Peter Sanderson (Faculdade Otterbein de Westerville) [Schemberger et al. 2010]. É uma alternativa ao software SPIM, foi desenvolvido em JAVA usando técnicas de interação homem-computador via pacotes Swing e AWT, e apresenta uma interface com usuário que inclui:

- Execução passo-a-passo de instruções, que permite debugar o código;
- Apresentação dos 32 registradores e seu conteúdo;
- Apresentação de registradores de ponto flutuante, co-processador1 e co-processador2;
- Seleção e apresentação de valores em decimal e Hexadecimal;
- Editor assembly integrado;
- Instruções e pseudos instruções MIPS;
- Visualização do conteúdo da memória de dados, de instruções e da pilha;

O MARS possui ainda uma gama de ferramentas integradas com finalidades didáticas e que possibilitam a verificação do funcionamento de um programa assembly MIPS, tais como: *Instruction Counter* que conta o número de instruções executadas por tipo (R, I, J); *Screen Magnifer* que captura a tela de execução; o *Floating Point Representation* que apresenta a representação de ponto flutuante em 32-bit IEEE 754; o *Data Cache Simulation Tool* que apresenta o desempenho dos dados em cache em forma de ilustração e simulação; dentre outras. Uma dessas ferramentas que foi utilizada no trabalho foi o *Bitmap Display* que serviu para apresentar a visualização gráfica do jogo.

3.2 Placa de Desenvolvimento FPGA Altera DE2-70

A placa de desenvolvimento DE2-70 da Altera é uma placa equipada com quase 70.000 elementos lógicos de Altera Cyclone(r) II 2C70 [Altera 2016]. A placa tem muitas características que permitem ao usuário implementar uma ampla gama de circuitos projetados, desde simples circuitos a vários projetos multimídia. Possui interfaces de multimídia, armazenamento e rede. No experimento usou-se como dispositivo de entrada de comandos para o jogo, o dispositivo receptor de infravermelho (IrDA) que recebe sinas de

controle remotos convencionais. Abaixo, segue o suporte de hardware oferecido pela DE2-70:

- SSRAM de 2MB;
- 9 LEDs verdes;
- 1 conector SMA;
- 18 interruptores;
- Transceptor IrDA;
- 18 LEDS vermelhos;
- Duas SDRAM de 32MB;
- 4 botões de Pressão;
- Memória Flash de 8MB;
- Soquete para cartão SD;
- 10/100 Ethernet Controller;
- 2 TV Decoder (NTSC/PAL/SECAM);
- CD-quality audio CODE de 24bit;
- Conector para mouse/teclado PS/2;
- Transceptor RS-232 e conector de 9 pinos;
- Dispositivo Altera Cyclone® II 2C70 FPGA device;
- VGA DAC (10-bit high-speed triple DACs) com conector VGA;
- Um oscilador de 50MHz e de 28,63 MHz para a fonte do *clock*
- Controle USB Host/Slave com conectores USB tipo A e tipo B;
- USB Blaster (on board) para programação e controle API do usuário, com suporte para os modos de programação JTAG e Active Serial (AS);

Além desses recursos de hardware, a placa DE2-70 possui suporte de software para interfaces de I/O padrão e uma facilidade de painel de controle para acessar vários componentes. Além disso, o software Quartus II é fornecido e serve para uma série de demonstrações que ilustram as capacidades avançadas da placa [Altera 2006].

3.2.1 IrDA

A DE2-70 oferece a possibilidade de se realizar comunicação sem fio que pode ser viabilizada por meio do transceptor infravermelho (IrDA) de baixa potência Agilent HSDL-3201. A taxa mais alta de transmissão/recepção de sinal suportada pelo transceptor é de 115.2 Kbits/s [Altera 2006]. Na figura 1 é apresentado o esquema deste transceptor.

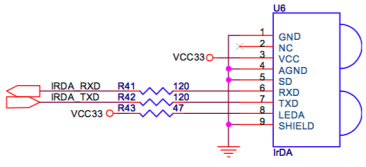

Figure 1: *Esquema IrDA*

Para regular a transmissão e recepção de sinais infravermelhos por meio do IrDA optou-se por utilizar o protocolo de transmissão NEC, por causa de sua simplicidade e pelo fato de ser um padrão comumente utilizado em controles remotos. Este padrão utiliza 32 bits de dados com algumas convenções para inicio e encerramento de leitura. A figura 2 temos o padrao exemplificado:

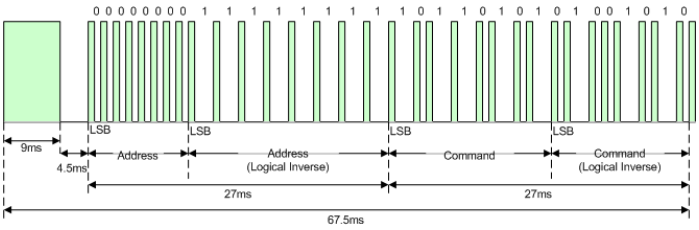

Figure 2: *Pacote completo do protocolo NEC [VASILIY 2007]*

Pela figura 2 pode-se observar que o pacote enviado em conformidade com o protocolo NEC inicia-se com um pulso largo, o qual chamaremos de estado de GUIDANCE. Esse pulso tem duração de 9ms e está em *low* e é seguido de um outro sinal em *high* de 4.5ms que informa o inicio do sinal a ser captado, os dados, e é precedido pelo estado IDLE, o qual por convenção estará sempre em estado lógico alto [Lanznaster 2014]. Vale ressaltar que os sinais utilizados para o desenvolvimento do IrDA são contrários aos exemplificados, uma vez que o receptor IrDA da placa DE2-70 estará sempre em *high* quando em IDLE.

Logo após o estado IDLE segue-se o estado GUIDANCE e a transmisão dos dados que chamou-se de estado de DATAREAD, o qual sempre iniciará dos *bits* menos significativos. Os dados são divididos em *byte* de endereço e de comando, seguidos por seus *bytes* invertidos(figura 2). O *byte* de endereço identifica o dispositivo a ser controlado [Lanznaster 2014].

Os dados em questão são sempre enviados em estado HIGH, seja a informação um bit 0 ou um bit 1. Cada *bit* é representado por um pulso seguido de uma pausa, sendo que o pulso lógico relativo a 0 tem um pulso de de 562,5 μs seguido de uma pausa de 562,5 μs, totalizando 1,125 ms, enquanto o 1 lógico é representado por um pulso de 562,5 μs seguido uma pausa de 1,6875 ms, totalizando 2,25 ms [Lanznaster 2014]. É importante notar que o estado HIGH do IrDA não é simplesmente acionar o diodo emissor de luz durante o tempo especificado. O estado HIGH é representado por uma onda quadrada na qual pulsos são gerados para o diodo emissor de luz à uma frequência de 38kHz.

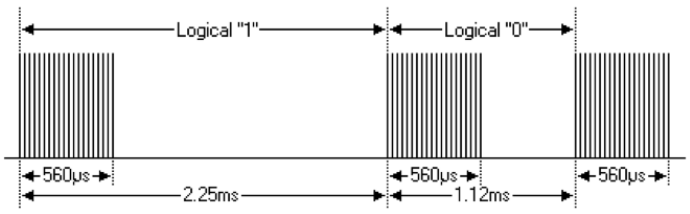

Figure 3: *Bit 1 e bit 0 do protocolo NEC [VASILIY 2007]*

Para finalizar da leitura ou do envio haverá um sinal em HIGH no final da cadeia de bits de duração aproximada de 562,5 μs. Este sinal informa a finalização do recebimento e permite que exista um intervalo para que a próxima cadeia seja lida corretamente.

3.3 MIPS

Segundo [Schemberger et al. 2010] o MIPS*(Microprocessor without Pipeline Stages)* tem sua origem nos anos 80 por meio de um trabalho de John Hennessy, na Universidade de Stanford, que tinha o objetivo de explorar o padrão RISC *(Reduced Instruction Set Computing)*. Essa abordagem da arquitetura RISC possui um número limitado de formatos de instrução que possuem o mesmo tamanho. Possui ainda 32 registradores de propósito geral que comportam palavras de 32 bits [Sweetman 2010].

Existem diversos modelos MIPS, mas as implementações utilizadas neste estudo são referentes às desenvolvidas na matéria Organização e Arquitetura de Computadores da Universidade de Brasília que tem por base as implementações MIPS R2000/R3000, descritas por [Patterson and Hennessy 2014].

3.3.1 Instruções MIPS

A arquitetura MIPS possui diversas instruções, todas contendo 32 bits, que podem ser classificadas em quatro grupos:

- Instruções do tipo R: Instruções que não necessitam de um valor imediato, deslocamentos de endereço de memória, ou endereço de memória para especificcar um operando. Essas instruções possuem *opcode* 000000, e possuem o seguinte formato:

opcode	rs	rt	rd	shamt	function
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  Em que:

  - Opcode: 000000
  - rs: Registrador com o primeiro operando
  - rt: Registrador com o segundo operando
  - rd: registrador de destino
  - Shamt: campo para deslocamentos
  - function: código da função.

- Instruções do tipo I: Instruções que possuem um imediato. Possuem o seguinte formato:

opcode	rs	rt	Imediato
6 bits	5 bits	5 bits	16 bits

- Instruções do tipo J: Instruções de salto que possuem um endereço da memória para onde se deseja saltar. E possuem o seguinte formato:

opcode	Endereço
6 bits	26 bits

- Instruções do coprocessador: Os processadores MIPS possuem dois co-porcessadores padrão, o CP0 e o CP1. CP0 trata as exeecessões e interrupções. E o CP1 é o responsável pelas operações em ponto flutuante. Suas instruções são semelhantes as instruções já citadas, porém apresentam algumas diferenças as quais não serão descritas no presente trabalho.

3.4 Registradores

O MIPS R2000 possui os seguintes registradores:

Número do Registrador	Nome	Uso
0	zero	Sempre retorna 0
1	at	Reservado para o assembly
2 - 3	v0, v1	Receber o valor retornado por subrotinas
4 - 7	a0 – a3	Parâmetros para subrotinas
8 - 15	t0 – t7	Para ser usado pelas subrotinas
24, 25	t8, t9	Para ser usado pelas subrotinas
16 - 23	s0 – s7	Registro de variáveis das subrotinas
26, 27	k0, k1	Reservado para ser usado por interrupções do SO
28	gp	(global pointer) Apontador de área global
29	sp	(stack pointer)
30	fp	(frame pointer)
31	ra	Registrador de endereço de retorno

Table 1: *Relação de registradores*

Há ainda os registradores HI e LO que são utilizados em operações de multiplicação e divisão, onde o resultado é armazenado de forma distribuída entre esses dois registradores. Os 32 bits mais significativos do resultado da operação é colocado em HI e os outros 32 bits menos significativos são colocados em LO [Schemberger et al. 2010].

3.4.1 Endereçamento e Acesso a Memória

Segundo [Schemberger et al. 2010] os sistemas baseados em processadore MIPS dividem a memória em três partes, conforme ilustra a Figura 4.

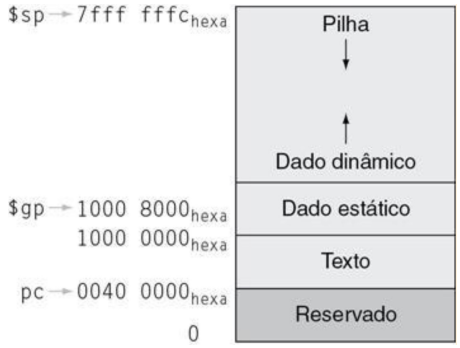

Figure 4: *Estrutura da memória [Patterson and Hennessy 2014]*

Como apresentado na figura 4 o segmento onde são armazenadas as instruções se inicia no endereço 00400000_h . Em seguida, no endereço 10000000_h, se incia o segmento de dados, que é subdividido em duas seções: A primeira que contém todos os dados que tem tamanho conhecido em tempo de compilação (dados estáticos); e a segunda é a que contém os dados dinâmicos, que são alocados em tempo de execução (essa seção pode ser expandida conforme administração do sistema operacional). A terceira parte da memória é o segmento da pilha que fica no topo da memória a partir do endereço $7FFFFFFF_h$ e cresce em direção aos endereços menores da memória.

E ainda, essa memória deve ser byte-endereçada, o carregamento/armazenamento deve ser alinhado, e os saltos na memória não podem exceder 256 MiB, tendo em vista a limitação relacionada ao tamanho das instruções e ao alinhamento da memória.

3.4.2 Estrutura dos Processadores MIPS

Os três processadores (uniciclo, multiciclo e pipeline) implementados na matéria foram desenvolvidos por meio da linguagem Verilog e com o apoio do sotware Quartus II para a FPGA Altera DE2-70. Os dados referentes ao mapeamento da memória e as chamadas do sistema, foram disponibilizados por meio dos arquivos *MIF (Memory inicialization file)* gerados por meio do MARS baseado em um código assembly. Os arquivos *.mif* são gerados pela ferramenta *MIF Exporter* do MARS.

Sendo assim, utilizou-se o arquivo SYSTEMv53.s que contém a rotina de tratamento de exceções/interrupções e syscalls e o arquivo BootLoader.s que contém a rotina do bootloader.

Para o Tetris geraram-se os arquivos seguintes arquivos:

- user_code.mif: Memória de Programa do Usuário;
- user_data.mif: Memória de Dados do Usuário;
- system_code.mif: Memória de Programa do Sistema;
- system_data.mif: Memória de Dados do Sistema

Dessa forma produziu-se um mapeamento de memória como apresentado nas tabelas abaixo:

Endereço	Tamanho	Uso
0x00400000 ... 0x00401FFF	8 kibytes	.text UserCodeBlock
0x8000 0000 ... 0x80001FFF	8 kibytes	.ktext SysCodeBlock

Table 2: *Memória de Instruções*

Endereço	Tamanho	Uso
0x0000 0000 ... 0x0000 007F	128 kibytes	.data Boot loader
0x1001 0000 ... 0x1001 1FFF	8 kibytes	UserDataBlock
0x1001 2000 ... 0x1021 1FFF	2 Mibytes	SRAM
0x9000 0000 ... 0x1021 1FFF	2 Mibytes	.kdata SysDataBlock

Table 3: *Memória de Dados*

Para localizar os dados utilizados pelo transceptor IrDA, definiram-se os endereços na interface do processador para que dados recebidos (RX) ou enviados (TX) estejam nos seguintes endereços na memória de dados:

Endereço	Tamanho	Uso
0xFFFF0500	4 bytes	IrDA_CTRL
0xFFFF0504	4 bytes	IrDA_RX
0xFFFF0508	4 bytes	IrDA_TX

Table 4: *I/O do IrDA mapeado em memória*

E os valores definidos na inicialização do Processador foram:

- Endereço de Boot: PC=0x00000000
- Endereço da Base da Pilha: $sp = 0x10211FFC
- Endereço do Contador de Programa: PC = 0x00400000

3.4.3 MIPS Uniciclo

O processador MIPS uniciclo usado segue a descrição de [Patterson and Hennessy 2014], porém além das nove instruções existentes naquela implementação, foram implementadas outras instruções existentes no processador MIPS R2000, tais como mult, multu, jal, bne, div, entre outras.

Esse processador possui a memória de instruções separada da memória de dados e todas as instruções demoram um ciclo de *clock* para serem processadas, isto é, executa somente uma instrução a cada ciclo de *clock*. Dessa forma, a frequência máxima de *clock* é o inverso do tempo máximo que se leva para executar a instrução mais lenta do conjunto de instruções.

As principais unidades funcionais deste processador são: Unidade de Controle do Processador, Coprocessador 0, Coprocessador 1, Unidade Lógica Aritmética (ULA), Banco de Registradores da ULA, Unidade de Ponto Flutuante (FPULA), Banco de *Flags* da FPULA, Memória de dados e Memória de Instruções (figura 5).

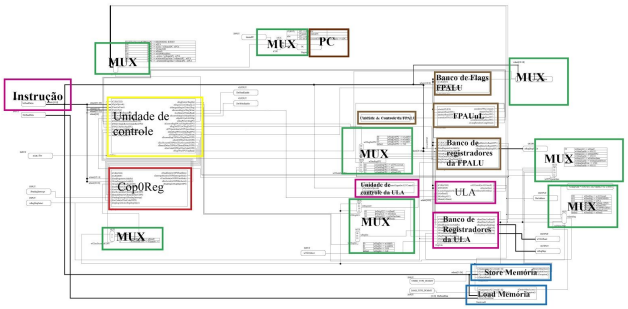

Figure 5: *Caminho de dados do Processador MIPS uniciclo*

3.4.4 MIPS Multiciclo

O processador MIPS multiciclo usado no experimento divide a execução de uma instrução em etapas, sendo que cada etapa é executada em um período do *clock*. Nesse tipo de implementação cada unidade funcional pode ser utilizada mais de uma vez por instrução, desde que em ciclos diferentes de *clock* [Neto 2016]. Assim, esta possibilidade de compartilhamento de unidades funcionais, em uma mesma instrução, pode ajudar a reduzir a quantidade de hardware necessário à implementação.

A execução das instruções pode ter um número diferente de etapas, dependendo do seu tipo/classe funcional. O importante é que ao final de um ciclo de *clock*, todos os dados que precisarem ser usados em ciclos subsequentes estejam armazenados em um elemento de estado. Portanto, os dados a serem usados em outras etapas devem ser armazenados em elementos de estado visíveis ao programador, ou seja, no banco de registradores, no PC ou na memória. No caso do processador MIPS multiciclo os dados são armazenados em registradores auxiliares, que ficam localizados após as unidades funcionais e que mantêm o dado por um ciclo de *clock* (com exceção do IR). Essa forma de implementação permite o uso de uma única ULA e uma única unidade de memória.

- Etapa 1: Busca da instrução.

  É comum a todas as instruções e nela são executadas as seguintes ações: realiza-se a leitura da instrução da memória e sua gravação no registrador IR, pois LeMem e EscreveIR estão ativados no bloco de controle. Incrementa-se o valor de PC, passando-se seu valor e o valor 4 para a ALU, por meio dos sinais de controle OrigAALU, OrigBALU e opALU. E por fim armazena-se o valor incrementado em PC, por meio dos sinais de controle origPC e EscrevePC.

- Etapa 2: Decodificação da Instrução e busca dos operandos.

  É comum a todas as instruções e nela são executadas as seguintes ações: acesso ao banco de registradores para ler os registradores indicados em *rs e rt*, armazenamento destes nos registradores intermediários A e B, calculo do endereço de desvio colocando o PC na ALU juntamente com o imediato estendido de sinal, por meio dos sinais de controle OrigAALU, OrigBALU e OpALU. E por fim, coloca-se o resultado da ALU no registrador SaidaALU.

- Etapa 3: Execução, cálculo do endereço de memória ou conclusão de desvio. Comum a quase todas as instruções.

- Etapa 4: Acesso à memória ou conclusão de instrução do Tipo-R. É realizada conforme a instrução.

- Etapa 5: Conclusão da leitura/escrita da memória.

No MIPS multiciclo utilizado a execução de qualquer instrução requer no mínimo 3 e, no máximo 5 ciclos de *clock*.

Segundo [Neto 2016] tipos diferentes de instruções necessitam de uma quantidade diferente de ciclos para serem executadas, cada etapa requer um ciclo de *clock*. Portanto, cada etapa é uma parte do caminho de dados. Isto é, uma etapa gera dados que serão utilizados

em um etapa seguinte e por isso necessita-se de registradores intermediários (A, B, IR e MDR), para guardar esses dados. Necessita-se ainda de algo que controle as etapas a serem executadas, por isso implementa-se um controlador por meio de uma máquina de estados.

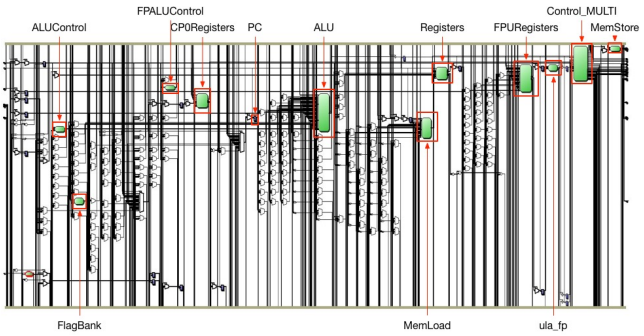

Figure 6: *Caminho de dados do Processador MIPS multiciclo*

3.4.5 MIPS Pipeline

A técnica de pipeline é utilizada para obter um maior ganho de desempenho. Nela o processamento da instrução normalmente é dividida em cinco estágios, cada um com um tempo fixo, que geralmente corresponde a um ciclo de *clock* do processador. Nela mais de uma instrução é processada por vez [Patterson and Hennessy 2014]. A implementação Pipeline tenta agregar os benefícios das implementações Multiciclo, trazendo benefícios como separar uma instrução em etapas menores, e Uniciclo, já que idealmente o número de ciclos de *clock* necessitados para executar uma instrução é 1.

As etapas podem ser resumidas conforme apresentadas na figura 7 e conforme descrito abaixo:

1. **IF**: Busca de instrução na memória;
2. **RD**: Leitura dos registradores e decodificação da instrução;
3. **EX**: Execução da operação ou calculo do endereço de memória;
4. **MEM**: Leitura ou escrita na memória de dados;
5. **WB**: Escrita do resultado em um registrador.

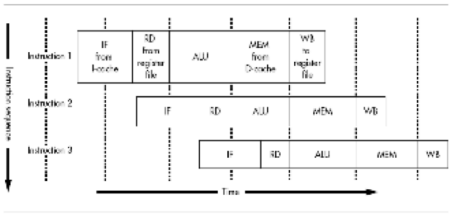

Figure 7: *Os cinco estágios do pipeline do MIPS [Stallings 2010]*

Entretanto, esta técnica possui algumas limitações. Ao se executar mais de uma instrução simultaneamente, caso um operando necessário para uma instrução não tenha sido calculado ainda por outra instrução anterior, gerará uma falha que será propagada e repetida ao longo do funcionamento do processador [Neto 2016]. Portanto, o pipeline para ser viável deve ser executado para um número grande de instruções e deve-se tratar os riscos de dados (*hazards de dados*) e os riscos de controle (*hazards de controle*).

Para contornar parte desses problemas, o processador MIPS pipeline usado no experimento possui uma unidade de controle de *hazard* e uma unidade de *forwarding* conforme descrito por [Patterson and Hennessy 2014] e apresentado na figura 8.

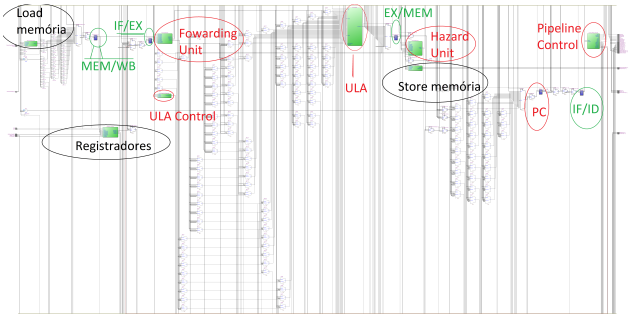

Figure 8: *Caminho de dados do Processador MIPS pipeline*

4 Requisitos

Para o presente estudos foi proposta a implementação de uma versão do jogo Tetris com os seguintes requisitos:

1. O jogo deve ser funcional: Apresentação, Movimentação, Aleatoriedade de blocos, Pontuação;
2. O jogo dever ser compatível com os processadores:
   - MIPS uniciclo
   - MIPS multiciclo
   - MIPS pipeline
3. O jogo deve ter efeitos sonoros e música;
4. Os comandos do jogo devem ser recebidos por meio do IrDA da FPGA Altera DE2-70; e
5. O Jogo deve possibilitar a seleção do número de jogadores no início do jogo (1 a 4);

5 Metodologia

Tendo em vista que de forma pretérita já haviam sido estabelecidos os requisitos e critérios para este trabalho, optou-se por uma abordagem de desenvolvimento que seguiu o seguinte ciclo: desenvolvimento voltado ao atendimento dos requisitos; testes, observação e comparação com o resultado desejado; correção e refatoração.

5.0.1 Receptor

A interface do receptor foi baseada na implementação desenvolvida pela Terasic, utilizada em outra FPGA, a DE2-115. Como o protocolo NEC também foi utilizando nessa implementação, sua adaptação para a DE2-70 foi relativamente simples. Após a adaptação direta da interface com a DE2-70, foi feita a implementação do módulo no processador MIPS. Para isso, o módulo utiliza o mesmo barramento utilizado por outras interfaces, além de ser necessária a definição dos endereços de memória que são utilizados pelo IrDA, o qual já foi descrito na seção 3.2.1. Já a implementação no processador foi feita de acordo com o seguinte código em linguagem Verilog:

```
    module IrDA_Interface(
    input           iCLK_50,
    input           iCLK,
    input           Reset,
    output          oIRDA_TXD,      //      IrDA Transmitter
    input           iIRDA_RXD,      //      IrDA Receiver
    //  Barramento de IO
    input           wReadEnable, wWriteEnable,
    input   [3:0]   wByteEnable,
    input   [31:0]  wAddress, wWriteData,
    output  [31:0]  wReadData
);

wire data_ready;
reg [31:0] hex_data;

IR_RECEIVE u1(
.iCLK(iCLK_50),
.iRST_n(1'b1),
.iIRDA(iIRDA_RXD),
.oDATA_READY(data_ready),
.oDATA(hex_data)
);

always @(*)
```

```
        if(wReadEnable)
                        begin
                                if(wAddress == IrDA_READ_ADDRESS)
                                   wReadData <= hex_data;
                                        else if(wAddress == IrDA_CONTROL_ADDRESS)
                                                wReadData[0] <= data_ready;
                                else
                                                wReadData = 32'hzzzzzzzz;
                        end
        else
                                wReadData = 32'hzzzzzzzz;

endmodule
```

A interface desenvolvida depende então da implementação feita pela Terasic para a placa DE2-115. Nota-se que é necessário um *clock* de 50 MHz, já que os tempos são baseados nessa frequência; necessita-se também de um forma de *reset*, mas como no nosso processador o *reset* já está implementado, este foi apenas desabilitado, definindo como constante; o pino da DE2-70 que se conecta com o pino RX do IrDA, como ilustrado na figura 1; um registrador para armazenar os dados enquanto esses não são repassados para o barramento e um fio para indicar se os dados já estão prontos para serem lidos. Caso algum problema ocorra ou um sinal infravermelho não seja detectado, a interface não altera nada no processador.

Para realizar acesso as informações disponibilizadas pelo IrDA basta apenas via software acessar o endereço reservado para leitura, que automaticamente faz com que o módulo jogue os dados no barramento para ser feita a leitura.

5.1 Abordagem Para o Desenvolvimento

Para que fosse possível o desenvolvimento do programa, o problema foi reduzido a sua forma mais simples para que mantivesse funcionalidade durante seu crescimento. Inicialmente ao invés de utilizar diretamente peças inteiras de tetris, os testes foram realizados apenas com um pixel. O objetivo inicial era apenas possibilitar a sua movimentação através de inputs de um teclado, com *feedback* de movimento na tela do MARS. Em seguida, foi desenvolvida uma forma de se adicionar gravidade ao objeto que juntamente com os comandos do teclado possibilitam direcionamento enquanto a peça segue no sentido para baixo convencional do jogo.

Tendo um objeto afetado pela gravidade agora é necessário adicionar condições de parada que podem seguir, em nosso contexto, de duas formas. A primeira é identificar colisões pelas informações que estão ali presentes em nossa tela como por exemplo as cores decodificadas nos bytes e a segunda reservar uma região específica de memória (diferente da MMIO de nossa VGA) para atualizar uma matriz de colisões. Com o objetivo de permitir maior liberdade a parte gráfica do jogo foi optado pelo desenvolvimento com a matriz de colisões que de forma simplificada é uma cópia de nossa memória da VGA ou seja uma região de memória com 76800 bytes que mapeiam os objetos adicionados na tela.

Para fazer a atualização desta matriz foi convencionado que ao atualizar o pixel na tela da VGA, o endereço do mesmo seria subtraído do endereço base da VGA e esta diferença seria adicionada no endereço base da matriz de colisão, para obter o endereço mapeado. Com este endereço obtido foi possível armazenar um byte contendo a informação de preenchimento da matriz ou de esvaziamento da matriz.

Logo, com um pixel capaz de cair e fazer o preenchimento de uma matriz de colisão, foi desenvolvida uma função que munida de um endereço de um byte qualquer da VGA fosse capaz de verificar se ocorreu uma colisão ou não. Em seguida, para que este pixel se mantivesse em limites desejados, bastou apenas fazer um preenchimento prévio do nosso campo navegável para que assim o pixel estivesse contido dentro dos limites.

Para o próximo passo, seria necessário que após a ocorrência da colisão, um novo pixel seja gerado para ser empilhado e gerar formas. Para que isso fosse possível foi criado um pivô pelo qual o *player* sempre irá comandar. Este pivô é salvo em memória e pode ser modificado sempre que necessário. Portanto para poder obter novos *pixels* controláveis (um a um) é necessário apenas pegar o retorno das colisões, e assim que uma delas for identificada, modificar o pivô para um novo pixel no topo da VGA.

Agora tendo toda a funcionalidade básica desenvolvida foi possível passar a estruturas mais complexas. Para tanto, o *pixel* foi transformado em quadrados, os quais possuem um tamanho variável estipulado por constantes inicialmente mensuradas. Estes quadrados continuam possuindo um pivô, o qual agora será passado para uma função que será encarregada de desenhá-lo na tela da VGA a partir de suas dimensões.

Diferentemente dos *pixels* os quadrados em um jogo convencional de tetris não se movem de forma continua, portanto para obter o efeito de quebra é necessário modificar a forma como seu deslocamento é realizado. Foi utilizada a própria dimensão do quadrado para o cálculo de seu próximo deslocamento, porém é necessário observar que sua posição inicial irá determinar seu alcance, portanto é importante que seja calculada devidamente a posição em que geramos os quadrados para que não terminem pulando colisões ou sobrepondo peças.

Com estes quadrados agora é possível compor peças mais complexas, para que isso seja possível é necessário modelar essa peça como um objeto para que possamos manipulá-lo como um só sem a preocupação de movimentar cada quadrado individualmente, ou de desenhá-los um por um. Sendo assim foram criadas matrizes 4x4 em memória que possuem em cada uma de suas words (16 words) um bit de validação que informa se ali existe um quadrado ou não.

Para a movimentação destas peças é apenas necessário que estipulemos um pivô para esta matriz, da mesma forma que fizemos da transformação dos *pixels* para os quadrados. O pivô das peças foi convencionado como qualquer outro jogo de tetris sendo localizado na segunda linha e terceira coluna. Logo para efetuar sua movimentação, acrescentamos gravidade neste pivô que será capaz de levar todas as unidades que compõem a peça apenas com sua movimentação.

Para que seja desenhada uma peça foi criada uma função que recebe o endereço inicial dessa matriz e a percorre procurando bits válidos para que assim calcule o endereço de fato do quadrado na VGA e passe este pivô encontrado para a função que desenha quadrados compondo assim nossa peça.

Da mesma forma são efetuadas as checagens de colisão, criando-se uma função que retorna o resultado, passamos o endereço inicial da matriz e checamos se cada um dos quadrados válidos (preenchidos) reconhece uma colisão, caso qualquer um deles reconheça que houve uma colisão então retornamos um resultado positivo para colisão da peça uma vez que agora os quadrados juntos representam um objeto só e não queremos fragmentos se movimentando individualmente.

Podemos então produzir qualquer tipo de formato apenas preenchendo uma matriz 4x4, o que inclui todas as rotações possíveis das peças utilizadas no tetris, sendo assim por praticidade vale a pena desenvolver uma máquina de estados para efetuar as rotações desejadas. Desta forma foram criados quatro estados para cada peça que representam suas rotações. Sempre que é requisitada uma rotação por um input mandamos para a função de rotação o endereço da matriz 4x4 com os bits de validação. Nesta função iremos pegar da memória uma word que armazena o estado atual da peça e o formato atual da peça e de acordo com o input desejado, passar para o próximo estado da peça que representa um rotação no sentido horário, atualizando a matriz 4x4 para este novo estado.

Desta forma possuímos uma mecânica básica de jogo munido de peças, rotações, identificação de colisão, geração de novos objetos e inputs para controle. Logo para estar jogável é necessário que seja feito o reconhecimento da formação de linhas no jogo, ou seja quando peças juntas requisitam a eliminação da respectiva linha e a descida do tabuleiro acima. De maneira simples foi desenvolvida uma função que verifica nas quatro possíveis linhas modificadas pela peça se houve a formação de uma linha, efetuando uma pesquisa a partir do pivô da peça mapeada para a matriz de colisão. Caso seja encontrada, iremos descer toda a matriz de colisão e as imagens da VGA contidas no campo do *player* uma unidade quadrada para baixa, efetuando o efeito desejado. Sempre que houver a formação de uma linha também iremos retornar um valor de pontuação para que possa ser validado depois em uma situação de

multiplayer.

5.2 Desenvolvimento Para *Multiplayer*

Para o *multiplayer* é necessário primeiramente imaginar o *player* como uma entidade, portanto este possui características próprias e informações que são necessárias serem guardadas e avaliadas constantemente. Portanto, foram reservadas regiões de memória correspondentes as informações de cada *player*, entre elas, peça atual, tipo da peça atual, estado da peça atual, matriz 4x4 da peça atual e número de pontos.

Para que cada *player* possa jogar seu jogo individualmente é necessário que cada tabuleiro possua um temporizador diferente e que seja efetuado um *update* constantemente para que nenhum jogador tenha que esperar ações de outros jogadores. Para que isso fosse possível foram desenvolidas funções *updates* que atualizam o estado de cada tabuleiro a partir de contadores individuais. Para melhor exemplificar, a cada vez que uma peça é criada é necessário um tempo para que ocorra sua movimentação, este tempo é armazenado em memória como um contador e cada vez que entramos na função *update* do *player* este contador é decrescido avaliado e armazenado novamente em memória. Assim que a contagem termina é possível atualizar os inputs do *player* e adicionar movimentação para a peça.

A partir desta abordagem é possivel replicar o procedimento para quantos *player*s forem necessários, apenas sendo necessário reservar memória para cada um deles e designar uma função *update* para seus dados. Para melhor compreensão desta função iremos mostrar seu funcionamento uma vez que esta é o núcleo de nosso jogo.

5.3 Fluxo principal

Logo abaixo teremos a declaração de nossa função principal, como podemos observar ela espera um parâmetro, o que pode ser um pouco inesperado, uma vez que o *player* possui diversas outras variáveis. O que ocorre é que a variável TIME.COUNT é mapeada como a primeira variável de um *player*, logo as seguintes podem ser obtidas a partir dela:

```
update.p:# (a0 = PLAYER.TIMER.COUNT)
```

Em seguida faremos uma verificação para saber se a contagem já terminou e se já é possível atualizar uma nova peça avaliando a flag NEWPIECE.FLAG que é definida assim que uma peça é criada e abaixada no momento que o contador termina sua contagem:

```
addi    $t0, $s4, 8                 # PLAYER1.NEWPIECE.FLAG
lw  $t0, 0($t0)
beq $t0, $zero, update.p.draw
```

Caso possamos criar uma nova peça iremos atualizar o pivô dessa peça e sortear um estado para ela, populando devidamente a matriz 4x4 do *player*. Em seguida, vamos desenhar a peça da seguinte forma:

```
add     $a0, $zero, $t0             # Endereco do pivo
add     $a1, $zero, COLOR.RED       # Cor para desenhar
addi    $a2, $s4, 44                # PLAYER1.PIECE.FORM
jal     draw.piece
```

Aqui PLAYER1.PIECE.FORM representa a matriz 4x4 sendo passada para a função que desenha uma peça, juntamente com o pivô da matriz já endereçado e uma cor para desenhar a peça. Em seguida, é necessário a obtenção dos inputs para que possamos calcular devidamente a próxima posição do *player* juntamente com sua gravidade:

```
add     $a0, $s4, 176               #PLAYER.KEY.ROTATE
jal     get.input
```

Acima chamaremos a função de input mapeada da mesma forma que a função *update* só que agora para os inputs do *player*. A função get.input irá identificar se um input específico do *player* foi acionado e caso tenha sido acionado irá armazenar zero no endereço de MMIO seja de keyboard ou de infravermelho para que este input não seja contabilizado novamente. Com o retorno desta função calculamos um futuro pivô que será avaliado, para que uma posição inválida não seja alcançada. Desta forma iremos utilizar uma matriz 4x4 temporária para calcular rotações caso tenham sido efetuadas e verificar se são possiveis. Caso sejam, iremos armazenar a matriz temporária na matriz de fato do *player*. Em seguida iremos salvar o novo pivô calculado e iniciar a contagem do contador para que possamos prosseguir o jogo. A contagem é iniciada em 0 e acrescentamos até chegar a um valor estipulado, acrescendo em 1 sempre que entramos na função:

```
beq     $t2, $t3, update.p.lp1.end
addi    $t2, $t2, 1
sw      $t2, 0($t0)             #armazena contagem
j       update.p.end            #terminamos update
```

Portanto, pelos próximos *update*s apenas efetuaremos contagem até que a mesma termine. Assim que ocorrer seu término, iremos apagar a peça para desenhá-la no futuro pivô, ou não iremos mais mexer a peça pois esta colidiu com algo e iremos apenas atualizar a matriz de colisão. Caso ocorra a atualização da matriz de colisão iremos fazer a checagem de linhas que irá nos retornar uma pontuação que será acrescida na pontuação total do *player*, para ser avaliada futuramente quem ganhou e quem perdeu o jogo.

5.4 Primeira versão

A primeira versão do jogo foi desenvolvida para comportar apenas um jogador, e após implementar as funcionalidades e verificar o seu correto funcionamento, implementou-se a possibilidade do jogo comportar até quatro jogadores.

Em princípio a implementação foi desenvolvida para receber os comandos do jogo por meio do teclado e sua jogabilidade foi testada com o auxilio da ferramenta *Bitmap Display* do MARS (figuras 9 e 10).

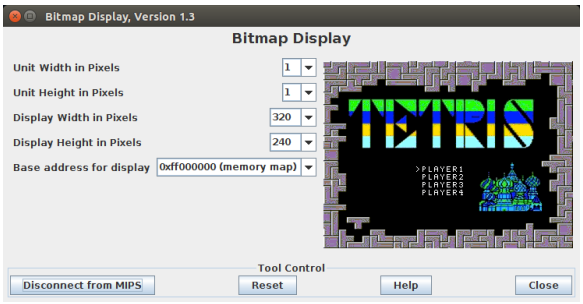

Figure 9: *Tela Inicial do Jogo Executando no Mars*

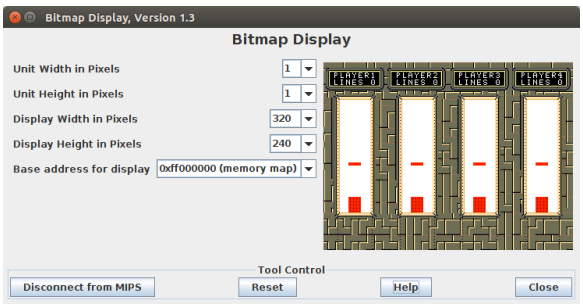

Figure 10: *Tela Principal do jogo com 4 Jogadores*

Após obter um funcionamento inicial que atendesse basicamente aos requisitos de software, passou-se a testar o código desenvolvido no processador uniciclo, ainda utilizando a entrada do teclado e seguiu-se refatorando o código e retestando até que se obtivesse os resultados esperados. Logo após, modificou-se o código para que

recebesse os comandos do jogador por meio da interface do IrDA. O mesmo processo foi seguido para os outros dois processadores (multiciclo e pipeline).

6 Resultados Obtidos

6.1 Testes na placa

Para a realização do jogo na placa, bastou gerar os arquivos .mif(arquivos utilizados para inicialização de memória na placa) via ferramenta disponível no mars. A principio foram realizados testes no processador pipeline que não apresentaram resultados positivos, uma vez que diversas rotinas de interrupção desenvolvidas por terceiros ainda não haviam sido adaptadas para o processador.

Logo devido a grande extensão destas rotinas os testes foram levados para o processador uniciclo o qual já tínhamos conhecimento da funcionalidade destas interrupções.

O processador utilizado em questão se apresenta bem estruturado, uma vez que diversos teste realizados via programas produzidos em assembly retornaram resultados positivos. Poucas modificações foram necessárias entre elas a troca das instruções MUL de três operandos para as instruções MULT de dois operandos, uma vez que a primeira não se encontra implementada, e a modificação do IrDA para que fosse possível modificar os registradores contidos no módulo, pois ao realizar uma leitura de um comando válido é necessário limpar o registrador para que o mesmo dado não seja lido mais de uma vez.

6.2 Requisitos atendidos

O jogo apresentou todas as funcionalidades aqui apresentadas o que inclui geração de peças, movimentação, inputs via IrDA, identificação de colisões e *multiplayer*. As funções *update* executaram tanto na placa quanto no mars de maneira sequencial e respeitando as contagens estipuladas. Porém diferentemente dos computadores utilizados para rodar o assembly, o processador mips da placa possui um *clock* fixo, portanto dependendo da quantidade de *players* é necessário modificar os contadores individuais, uma vez que o programa tenderá a rodar mais lentamente devido a maior quantidade de *update*s.

6.3 Requisitos não atendidos

Um dos problemas encontrados foi a geração de blocos aleatórios na placa. As rotinas existentes no processador são realizadas fazendo-se um embaralhamento de entradas do relógio do processador, o que cria números periódicos que possibilita facilmente a identificação de um padrão de queda de blocos.

O desenvolvimento de músicas apesar de simples não foi o foco do projeto, portanto devido a complexidade do programa esta parte foi deixada para um momento posterior. Entretanto, é importante ressaltar que sua implementação seria idêntica ao *update* realizado para cada *player*. Nela teria-se uma rotina capaz de tocar uma determinada nota musical sempre que fosse chamada por meio de um contador que faria o espaçamento entre elas, de forma que fossem tocadas no intervalo certo ao chamar a rotina novamente.

Como foi comentado logo no inicio a implementação no processador pipeline foi de maior dificuldade, pois foram utilizadas rotinas já prontas que não haviam sido adaptadas. Alguns testes foram realizados com a obtenção de resultados razoáveis, como por exemplo a impressão de strings na tela da VGA que após adaptação se apresentou funcional, mesmo que o programa como um todo não tenha seguido a execução esperada.

Dificuldades:

6.4 Simulação do software no MARS

Durante todo o processo de desenvolvimento do jogo cada funcionalidade foi testada e avaliada no MARS. Primeiramente foram ajustados os temporizadores de cada entidade de forma a adequar ao *clock* simulado no MARS, em virtude que na Placa o jogo executa com uma velocidade maior que em simulação. Para emular as entradas do IrDA foi utilizado o módulo Keyboard and Display MMIO Simulator, permitindo o controle de cada jogador por meio de teclas do teclado.

Uma vez tendo o jogo minimamente funcional com os 4 jogadores implementados e o menu de entrada para seleção do número de jogadores, foram avaliados o layout de cada uma telas tomando por base o jogo original implementado para o videogame Super Nintendo. Desta forma foram geradas as imagens de fundo em um arquivo .asm incorporado ao segmento de dados do código do jogo. Para tal as imagens originais no formato BMP 24 bits foram convertidas para o formato MIF usando o programa *bmp2mif* fornecido e em seguida convertidas em assembly mips usando um script desenvolvido em python. Ao final deste processo, tivemos como impactante a necessidade de uma quantidade de memória maior de dados que a disponível na placa neste segmento de apenas 8kb, para tal adotou-se a estratégia de carregar a imagem inicial diretamente no ato de inicialização da memória de vídeo do processador e a tela de jogadores foi deixada sem imagem de fundo para o uso direto na placa.

6.5 Simulação usando o hardware

7 Conclusâo

O presente trabalho verificou o desenvolvimento do clássico jogo Tetris em linguagem assembly MIPS para execução em processadores MIPS uniciclo e multiciclo, que por sua vez são desenvolvidos em linguagem Verilog e são simulados com o auxílio da placa de desenvolvimento FPGA Altera DE2-70.

A metodologia adotada viabilizou o desenvolvimento do jogo atendendo a grande parte dos requisitos pré-estabelecidos. Não sendo possível apenas a implementação de sons e música para o jogo, e a execução no processador MIPS pipeline. Porém, a equipe deste estudo julga que é possível atender a estes requisitos. Que não foram atendidos pelo fato de demandarem mais tempo de pesquisa e desenvolvimento não disponíveis para a entrega deste trabalho.

Os demais requisitos foram atendidos, alcançando-se as funcionalidades e a jogabilidade desejada, inclusive com a transmissão e recepção dos comandos do jogo por meio de dispositivos infravermelho (Transceptor IrDA e controle remoto).

Portando, pode-se concluir que o estudo foi bem sucedido e que a metodologia adotada é eficaz. Que possibilita a divisão do trabalho entre a equipe, bem como permite a revisão de erros e o aperfeiçoamento dos artefatos desenvolvidos.

Para trabalhos futuros, pretende-se possibilitar que o jogo emita sons e música, que possa ser executado no processador MIPS pipiline e que possa ser jogado com o auxilio de mais de um placa FPGA Altera DE2-70.

Acknowledgements

Ao Prof. Dr. Marcus Vinicius Lamar, pelo apoio, pela forma como conduz a disciplina de OAC e por nos ter levado ao limite de estudo e do esforço, demostrando que a vontade de realizar um feito deve estar acima dos desejos momentâneos da vida.

References

ALTERA, 2006. Gde2 development and education board user manual.

ALTERA, 2016. Altera de2-70 - banco de ensaios em fpga cyclone ii ep2c70f896. [Online; accessed 6-novembro-2016].

LANZNASTER, L. C. 2014. Temporizador digital para controle de reuniões.

NETO, A. M. 2016. Aceleração da implementação do algoritmo de criptografia aes-128 em um processador mips. *CEP 70910*, 900.

PATTERSON, D. A., AND HENNESSY, J. L. 2014. *Organização e projeto de computadores: interface hardware/software*.

SCHEMBERGER, E. E., ARAUJO, K., AND ANDRADE, S. C. 2010. Eks-mips: Um simulador para processador mips. *ENCONTRO NACIONAL DE DIFUSÃO TECNO LÓGICA (ENDITEC)*.

STALLINGS, W. 2010. *Computer organization and architecture : designing for performance*. Pearson, Boston, Colombus, Indianapolis. Bibliogr. p.768-780. Index. Glossaire.

SWEETMAN, D. 2010. *See MIPS run*. Morgan Kaufmann.

VASILIY, R., 2007. Implementation of ir nec protocol. [Online; accessed 10-dezembro-2016].

VOLLMAR, K., AND SANDERSON, P. 2006. Mars: an education-oriented mips assembly language simulator. In *ACM SIGCSE Bulletin*, vol. 38, ACM, 239–243.

WIKIPÉDIA, 2016. Tetris — wikipédia, a enciclopédia livre. [Online; accessed 6-novembro-2016].