ID: 92482

Ponto de Checagem 4:
Projeto de Implementação de MIPS

São José dos Campos - Brasil

Julho de 2017

ID: 92482

Ponto de Checagem 4:
Projeto de Implementação de MIPS

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Aluno: Ricardo Manhães Savii

Docente: Prof. Dr. Tiago de Oliveira

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Julho de 2017

Resumo

Este projeto será o relatório do processo de implementação de um sistema computacional simplificado semelhante aos utilizados nos computadores pessoais. Os processos internos necessários de um computador pessoal serão descritos de acordo com a bibliografia e implementadas em uma arquitetura digital capaz de processar instruções provenientes de uma memória principal. As instruções são compostas de números binários e serão interpretados pelo circuito interno do processador. A arquitetura deste projeto inicial do ciclo de laboratórios no curso de Engenharia de Computação do Instituto de Ciência e Tecnologia (ICT) será composta por: *central processing unit* (CPU), memória e interface de comunicação. Este projeto não tem como objetivo alcançar a complexidade dos processos de compilação o que será considerado como extensão deste em futuros laboratórios ao longo do curso. Assim, o objetivo neste curso é planejar e construir a arquitetura base para o futuro desenvolvimento de um sistema computacional completo.

Palavras-chaves: Arquitetura e Organização de Computadores (AOC). *field programmable gate array (FPGA). Microprocessor without interlocked pipeline stages (MIPS).*

Glossário

CISC *Complex Instruction Set Computer*. 12

CPU *central processing unit*. 2, 7, 10–13, 20–23, 25, 27, 29

CWP *current-window pointer*. 14, 15

DMA *Direct Memory Access*. 17, 26

FPGA *field programmable gate array*. 8, 10, 11, 18, 22, 26, 34, 37, 39, 42

HDL *hardware description language*. 10

I/O *Input-Output*. 10, 13

ICT Instituto de Ciência e Tecnologia. 2, 10

IEEE *Institute of Electrical and Electronics Engineers*. 10

ISA *Instruction Set Architecture*. 12

MIPS *Microprocessor without interlocked pipeline stages*. 3, 10, 12, 15, 18, 20, 22, 41, 42

PC *Program Counter*. 7, 8, 21, 23, 25–27, 34–36, 39, 41, 49

RAM *random access memory*. 11, 18

RISC *Reduced Instruction Set Computer*. 12, 15, 20

SMIPS *Simple MIPS*. 15

UC unidade de controle. 7, 11, 20, 30, 31, 33–36

ULA unidade lógica e aritmética. 7, 8, 11, 21, 23–25, 30, 33, 50

Lista de Códigos

Lista de ilustrações

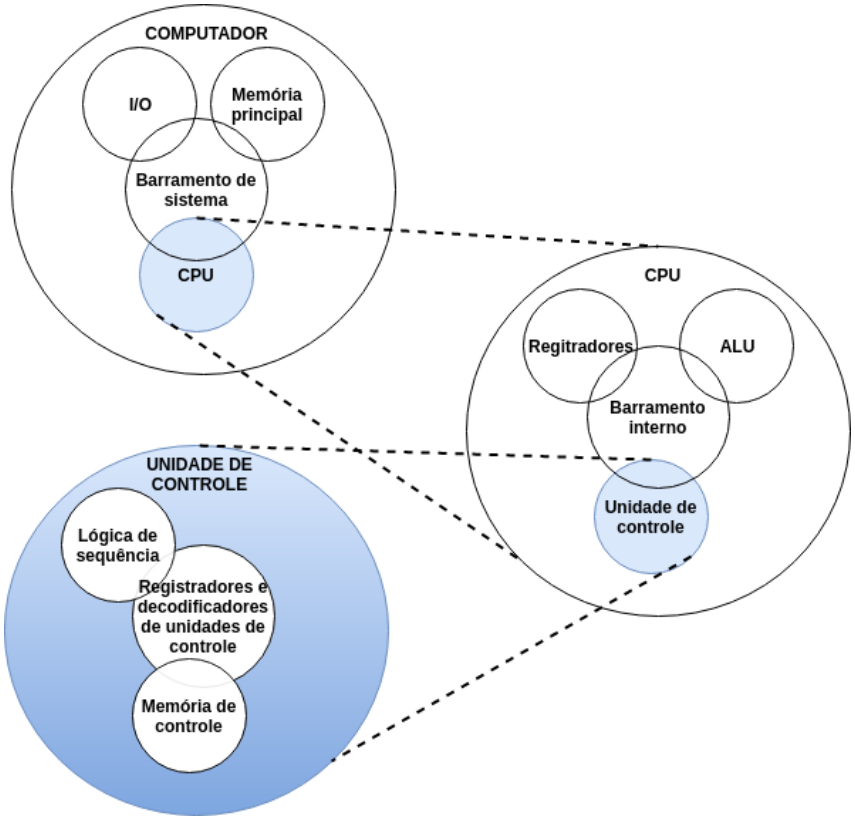
Lista de tabelas

Sumário

1 Introdução

A implementação de sistemas digitais é um campo muito importante na Engenharia de Computação, estes são um primeiro passo para entender modelos mais complexos. [1] Um computador é um sistema complexo digital, com subsistemas inter-relacionados e em hierarquia. A natureza hierárquica é essencial para seu projeto e descrição, esta idéia é apresentada na **Figura 1**.

Figura 1 – A estrutura de nível superior do computador



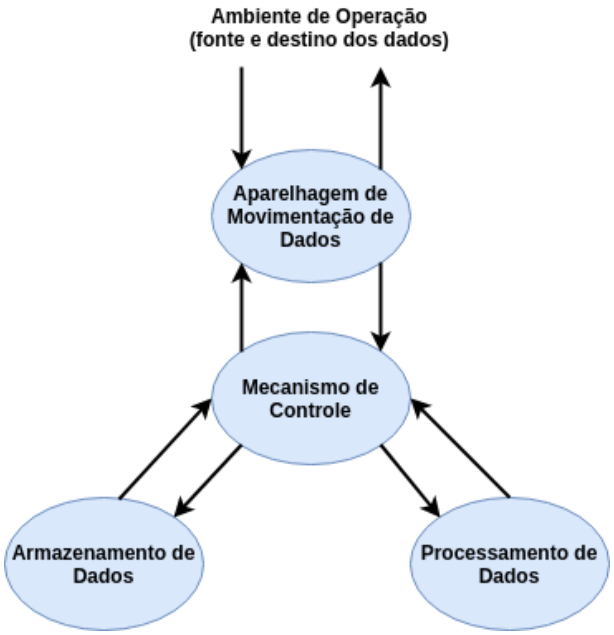
Traduzido de: [1, pag. 13]

A cada nível o projeto preocupa-se com estrutura e função:

- **Estrutura:** a forma como cada componente está inter-relacionado.
- **Função:** a operação de cada componente individual como parte da estrutura.

Na **Figura 2** apresenta as 4 funções que um computador realiza:(i) **processar dados**, (ii) **armazenar dados**, (iii) **movimentar dados** e (iv) **controlar** as três funções. Cada uma destas funções possui suas próprias complexidades que serão estudadas e implementadas no decorrer do semestre.

Figura 2 – Uma visão funcional do Computador



Do ponto de vista de **estrutura** o computador possui quatro principais componentes estruturais: (i) **Unidade central de processamento (CPU):** Controla a operação do computador e opera funções de processamento; este é o **processador**. (ii) **Memória principal:** Armazena dados. (iii) **Entrada e saída - *Input-Output* (I/O):** Movimenta os dados entre o computador e seu ambiente externo. (iv) **Interconexão do sistema:** Algum mecanismo que proporciona comunicação entre CPU, memória principal, e I/O. i.e.: o **barramento de sistema** (*system bus*).

Este projeto terá como produto final a implementação de um processador computacional em *verilog*[1] no Software **Quartus® II 64-Bit Version 13.1.0 Build 162 10/23/2013 Sj Web Edition**[2] produzido pela **Altera** com licença gratuita governada pelo **Intel FPGA Software License Subscription Agreement**[3]. Após definição do circuito no *software*, ele será testado em uma placa **DE2-115** (EP4CE115F29C7) com chip **FPGA** (em português Arranjo de Portas Programável em Campo) **Cyclone IV** produzida pela **Altera** disponibilizado pelo ICT para fins educativos.

Cada um dos componentes será detalhado e implementado para uma arquitetura *Microprocessor without interlocked pipeline stages* (MIPS) ao longo do projeto.

[1] *Verilog*, cuja padronização atual é a *Institute of Electrical and Electronics Engineers* (IEEE) 1364-2005, é uma linguagem de descrição de hardware (*hardware description language* (HDL)) usada para modelar sistemas eletrônicos.

[2] *software* disponível em: <http://dl.altera.com/13.1/?edition=web>

[3] licença disponível em: <http://dl.altera.com/eula/16.1/>

2 Objetivos

2.1 Geral

Conforme a ementa da unidade curricular Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores

> o aluno deverá ter implementado um sistema digital composto por processador, memória e interfaces de comunicação. Descrever a arquitetura de um processador utilizando uma ferramenta de descrição de *hardware*; utilizar lógica programável para implementar um processador; realizar simulações e testes para verificar a funcionalidade do sistema projetado; desenvolver em lógica programável um sistema de memória; desenvolver em lógica programável um sistema de comunicação; elaborar apresentações orais e redações de textos.

Portanto este projeto tem como objetivo documentar e implementar uma CPU, uma memória principal (*random access memory* (RAM)) e uma interface de comunicação implementados em *hardware* por meio de uma FPGA.

2.2 Específico

Para alcançar o objetivo geral, o projeto foi subdividido em etapas que serão desenvolvidas conforme o a **Tabela 1**. Estas são desenvolver: (i) uma ULA, registradores específicos e a CPU; (ii) uma memória principal; (iii) a UC; (iv) uma interface de comunicação com o meio externo em que podem ser utilizados dispositivos como: *mouse*, teclado, monitor, *display* de 7 segmentos ou *LEDs*; (v) integração das partes.

Tabela 1 – Cronograma de atividades

Etapas \meses	Março	Abril	Maio	Junho
ULA, Registradores, CPU	X	X		
Memória Principal		X		
UC		X	X	
Interface de Comunicação			X	
Integração			X	X
Testes e simulações		X	X	X
Escrita de relatório	X	X	X	X

3 Fundamentação Teórica

A *Instruction Set Architecture* (ISA) (em português: arquitetura do conjunto de instruções) é o que irá permitir a realização de tarefas com o processador, e define a estrutura física da CPU. As duas principais ISAs são *Complex Instruction Set Computer* (CISC) e *Reduced Instruction Set Computer* (RISC).

3.1 CISC vs RISC

A principal vantagem da arquitetura CISC é que seu conjunto maior de instruções facilita o trabalho dos programadores de linguagem de máquina. O número de linhas necessárias para operações é menor ao necessário em outras arquiteturas. Porém, há a impossibilidade de personalizar instruções *compostas* para, por exemplo, melhorar a performance. O código de instruções *compostas* CISC podem ser escritos nos RISC igualmente, ou usando um conjunto de instruções simples, de forma personalizada ao problema. Logo, a discussão entre escolher uma arquitetura RISC ou CISC é uma questão de escolher entre **tamanho de código X desempenho**.

A arquitetura CISC exige uma construção de *hardware* mais complexa, para abranger todas as suas instruções. Elas operam diretamente nos bancos de memória e não requer o uso discreto das funções de acesso à memória, *load* e *store*. Resumindo, uma arquitetura CISC possui as seguintes propriedades: (i) a maioria das instruções possuem acesso à memória; (ii) modos de endereçamento são substanciais em número; (iii) formatos de instrução apresentam tamanhos variados; (iv) instruções realizam tanto instruções simples como complexas.[2]

Já o conjunto de instruções RISC enfatiza instruções simples e flexibilidade. Exige um número maior de linhas de código, ou seja, mais uso de memória para a mesma tarefa, porém requer menos espaço de transistores em *hardware*. Uma arquitetura RISC possui as seguintes propriedades: (i) somente funções *load* e *store* possuem acesso à memória, e instruções de manipulação de dados ocorrem entre registradores; (ii) modos de endereçamento são limitados em número; (iii) todas as instruções possuem o mesmo tamanho; (iv) operações realizam somente operações consideradas simples.

3.2 Uso otimizado de registradores em MIPS

A aplicação da arquitetura MIPS espera um acesso rápido a operandos que são referenciados frequentemente. Para isso são indicados os usos de registradores, método mais

rápido de guardar informação, mais rápido que memória principal e cache. Na arquitetura deste projeto será utilizado o número padrão de registradores (32), o que é um valor pequeno de memória. O que necessita de uma estratégia para manter os operandos mais utilizados salvos nos registradores e minimizar as operações entre registradores e outras memórias.

Duas estratégias são possíveis, uma fundamentada em *software* e outra em *hardware*. O método de *software* irá recair no compilador, que tentará alocar registradores para essas variáveis de mais uso. O que requer algoritmos sofisticados e complexos. A estratégia por *hardware* é basicamente incluir o maior número de registradores possível.

3.2.1 Registradores

Para entender o papel dos registradores na CPU, devemos considerar os requisitos da CPU, o que deve fazer:

1. Buscar instruções (*fetch instructions*) - deve ler instruções da memória.
2. Interpretar as instruções (*Interpret*) - Decodifica as instruções para determinar ações requeridas.
3. Busca dados (*fetch data*) - a execução de uma instrução pode requerer a leitura de dados de um módulo I/O.
4. Processa dados (*Process*) - a execução de uma instrução pode requerer realizar operações aritméticas ou lógicas em dados.
5. Registrar (*Writeback*) - os resultados de uma execução podem requerer escrever dados na memória ou em um módulo I/O.

Para realizar estas operações, é claro que a CPU precisa guardar dados temporariamente. Ou seja, a CPU precisa de uma pequena memória interna, que consiste de registradores de alta velocidade. Estes registradores servem dois propósitos:

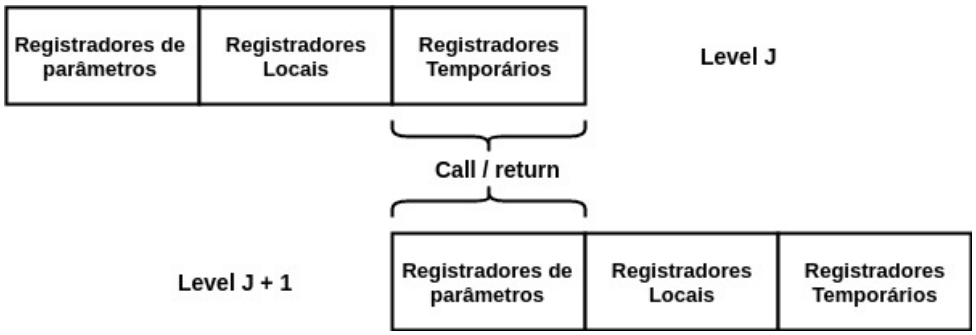
- registradores visíveis ao usuário: permitir ao programador minimizar referências à memória ao otimizar o uso de registradores.
- registradores de controle e status: estes são usados pela unidade de controle para gerir as operações da CPU, e por programas do sistema operativo, privilegiados, para controlar a execução de programas.

Não há clara divisão dos registradores nestas categorias, sendo dependente das opções durante a implementação. [3] Com o maior uso de registradores, a necessidade

de acessar a memória irá diminuir, e uma tarefa do projeto é organiza-los de forma que este objetivo seja alcançado. Sabe-se [4] que procedimentos típicos consomem poucos parâmetros e variáveis locais, e a profundidade de ativação de uma procedimento flutua em um alcance raso. Para explorar estas propriedades, diversos conjuntos pequenos de registradores são usados, cada um definido para um diferente procedimento.

Um procedimento utiliza automaticamente uma "janela" (*window*) diferente de registradores. Janelas para procedimentos adjacentes são sobrepostos para permitir parâmetros serem repassados. O conceito é ilustrado na **Figura 3**. As janelas são divididas em registradores de parâmetros, que guardam parâmetros do procedimento atual e resultados para serem repassados para cima. Registradores locais são usados para variáveis, conforme definido pelo compilador. E registradores temporários são usados para trocar parâmetros e resultados com o próximo nível mais baixo - procedimento chamado pelo procedimento atual. Esta sobreposição permite passar parâmetros sem necessidade de movimentar dados.

Figura 3 – Janelas sobrepostas de registradores

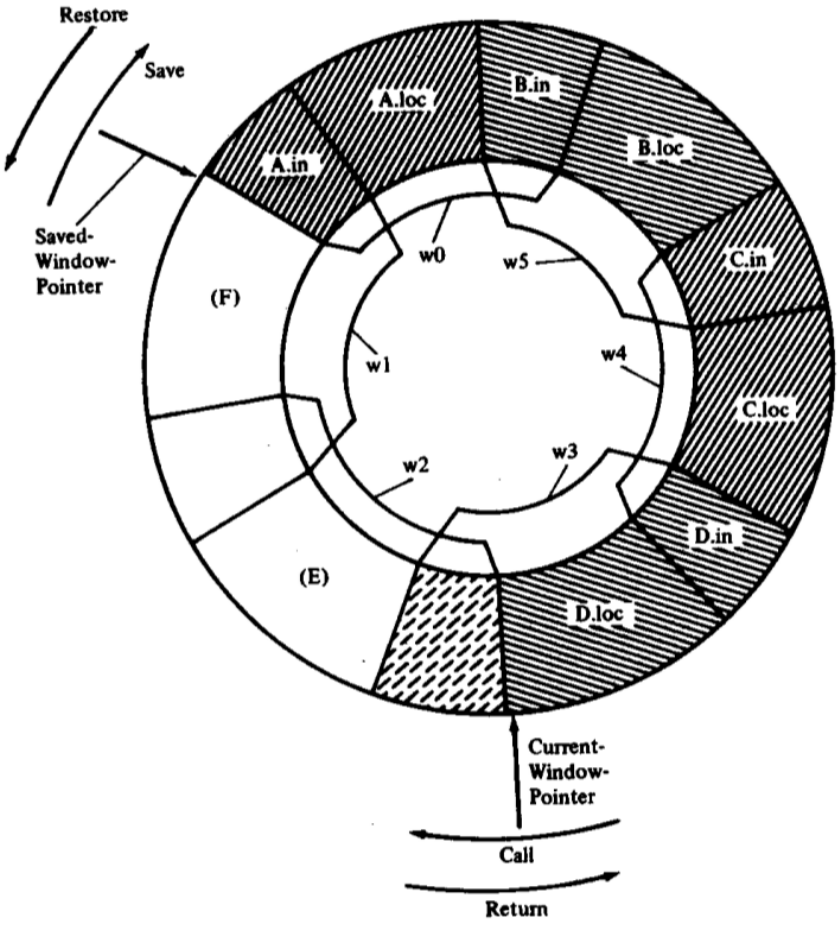


Traduzido de: [4, pag. 42]

Para gerir qualquer padrão de chamadas e retornos, o número de janelas de registradores precisaria ser ilimitada. Porém, as janelas podem ser utilizadas para guardar as últimas chamadas de procedimentos. Chamadas mais antigas podem ser guardadas na memoria e requisitadas somente se necessário, quando a profundidade da chamada aumenta em excesso. Logo, a organização do arquivo de registradores é um *buffer* (amortecedor, em português) circular com janelas sobrepostas. O esquema desta organização é visível na **Figura 4**.

O *buffer* da figura está preenchido até uma profundidade 4 (A chamou B; B chamou C; C chamou D) com o procedimento D ativo. O ponteiro para a janela atual - *current-window pointer* (CWP) - aponta para a janela com procedimento ativo. As referências de registradores por uma instrução da máquina são deslocadas por este ponteiro para determinar o registrado físico correto. O ponteiro para janela salva (*saved-window pointer* identifica a janela mas recente salva na memória. Se o procedimento D chamar o procedimento E, argumentos para E são colocados nos registradores temporários de D (a

Figura 4 – Organização de *buffer* circular com janelas sobrepostas



Fonte: [4]

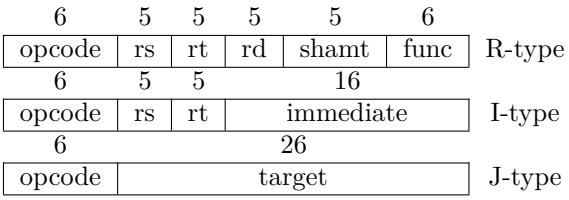
sobreposição entre w3 e w4) e o CWP é avançado em uma janela. Os computadores RISC da *Berkeley* utilizam 8 janelas de 16 registradores cada.

Porém este esquema é ineficiente quando incluímos o trabalho com variáveis globais, variáveis comuns para todos os procedimentos. Para resolver este problema normalmente são definidos registradores para uso único de variáveis globais, outra opção é começar a utilizar as memórias cache. Neste projeto serão utilizados os 32 registradores de uso comum e mais um conjunto, ainda, indefinido de registradores para controle.

3.3 SMIPS ISA

A arquitetura MIPS é fundamentada em RISC, e um subconjunto de instruções é conhecida como SMIPS (significa *Simple* MIPS), suas instruções podem ser agrupadas em três categorias apresentadas na **Tabela 2**:

Tabela 2 – Formatos das instruções



Para a qual temos as seguintes descrições: (i) ***opcode***: são os 6 primeiros bits do código de operação; (ii) ***rs***: o registrador da fonte (***source***) de 5 bits; (iii) ***rt***: registrador de alvo (***target***) também de 5 bits e utilizado para funções do tipo I; (iv) ***rd***: o registrador de destino de 5 bits; (v) ***shamt***: valor de desvio de 5 bits e (vi) ***func***: Campo de função de 6 bits usado para especificar as funções especiais dentro do opcode primário; (vii) ***immediate***: 16 bits de valor imediato, com sinal, usado para operações lógicas, operandos aritméticos e deslocamento de *bytes* para endereços de ***load/store***; (viii) ***target***: índice de 26 bits deslocados para a esquerda dois bits para fornecer os 28 bits *low-order* do endereço de destino do salto (*jump*).

3.3.1 Modos de Endereçamento

As instruções possuem operandos, e para acessa-los há diversas formas. Os endereços em uma instrução típica são relativamente pequenos, e pode ser interessante poder referenciar uma grande área de memória. Para obter este acesso há diversas estratégias de endereçamento, cada qual com seus *trade-offs* entre alcance, flexibilidade, complexidade entre outros [5, 1]. A **Tabela 3** apresenta as principais de forma resumida, que serão detalhadas posteriormente.

Tabela 3 – Modos de Endereçamento

Método	**Algoritmo**	**Principal Vantagem**	**Principal desvantagem**
Imediato	Operando = A	Sem referência à memória	Magnitude do operando limitada
Direto	EA = A	Simples	Espaço limitado de endereços
Indireto	EA = (A)	Amplo espaço de endereço	Múltiplas referências de memória
Registrador	EA = R	Sem referência à memória	Espaço limitado de endereços
Registrador indireto	EA = (R)	Amplo espaço de endereço	Referência extra de memória
Deslocamento	EA = A + (R)	Flexibilidade	Complexidade
Pilha	EA = topo da pilha	Sem referência À memória	Aplicação limitada

Fonte: traduzido de [1, pag. 454]

Para os métodos ilustrados na **Tabela 3** foram utilizadas as seguintes notações: (i) **A** = conteúdo de um campo de endereço em um instrução, (ii) **R** = conteúdo de um campo de endereço na instrução com referência à um registrador, (iii) **EA** = endereço efetivo do local contendo o operando referenciado, (iv) **(X)** = conteúdo do local de memória X ou registrador X. É importante observar que todas as arquiteturas de computadores provém mais de um método de endereçamento.

O método mais simples é o (i) **endereçamento direto**, no qual o operando está presente na instrução. Este método é normalmente usado para definir valores de variáveis iniciais. O (ii) **endereçamento direto** também é bem simples, no qual o campo de endereço contém o endereço efetivo do operando, era comum nos primeiros computadores por não precisar de cálculos especiais de endereços, porém por ter pouco alcance é pouco utilizado atualmente. (iii) **Endereçamento indireto** possui um campo de endereço que referencia o endereço de uma palavra (*word*) na memória, que contém um endereço completo de um operando. A principal vantagem é que um palavra de tamanho N nos permite um alcance de 2^N endereços diferentes, com o custo de duas referências à memória. Já o (iv) **endereçamento por registrador** é semelhante ao endereçamento direto, a diferença é que o campo de endereço se refere a um registrador ao invés de um endereço da memória principal. Da mesma forma como o endereçamento por registrador é análogo ao endereçamento direto, o (v) **endereçamento por registrador indireto** é análogo ao endereçamento indireto, com a mesma diferença ao referenciar para um registrador ao invés da memória. Um método bem poderoso é o (vi) método de **deslocamento** que combina as capacidades do endereçamento direto e registrador indireto. O valor contido em um campo de endereço (valor = A) é usado diretamente, e outro campo referencia um registrador cujo conteúdo será adicionado à A para produzir o endereço efetivo. O método de deslocamento se diferencia em outros três métodos: **endereçamento relativo**, (vii) **endereçamento base-registrador** e (viii) **indexação** explicados mais adiante. Por último temos (ix) **endereçamento de pilha** que é um bloco reservado de locais, com um ponteiro de topo e funciona como a estrutura de dados pilha. Assim as instruções não precisam endereçar um local, o operando será recolhido sempre do topo da pilha.

3.4 Sistema de Entrada e Saída

Este é responsável pela conexão entre o processador e qualquer dispositivo externo conectado ao computador, podem ser teclados, *mouse*, monitor, impressora, etc. Os periféricos (dispositivos externos) podem ser classificados entre periférico de (i) entrada, (ii) saída ou (iii) entrada e saída.[6] Há também componentes intermediários que facilitam a variabilidade destes periféricos e seus controles específicos. Estes componentes podem ser (i) interfaces, (ii) controlador, (iii) *driver*, (iv) portas de entrada e saída e (v) barramentos.

As transferências de dados podem ser realizadas direto ou por intermediação de um controlador *Direct Memory Access* (DMA). Este fica responsável pela transferência de blocos de dados entre o periférico e a memória principal. [6] O DMA possui alguns registradores próprios, por ser um processador simplificado e independente, que recebe ordens do processador principal e avisa-o (interrupções) quando os dados estão prontos.

3.5 Memória Principal

A memória principal de um computador é a sua memória RAM, e permitem que seja feita leitura e escrita de dados. Porém, são memórias voláteis, só existem enquanto houver suprimento de energia elétrica. Estas normalmente são utilizadas como memória suporte e de acesso rápido para operações do computador - mais lentas que registradores e cache, porém mais rápidas que memórias não voláteis. As memórias não voláteis são utilizadas para armazenamento contínuo, não necessitam de suprimento contínuo de energia elétrica [7], ou seja, a memória é mantida após desligar o computador.

3.6 FPGA

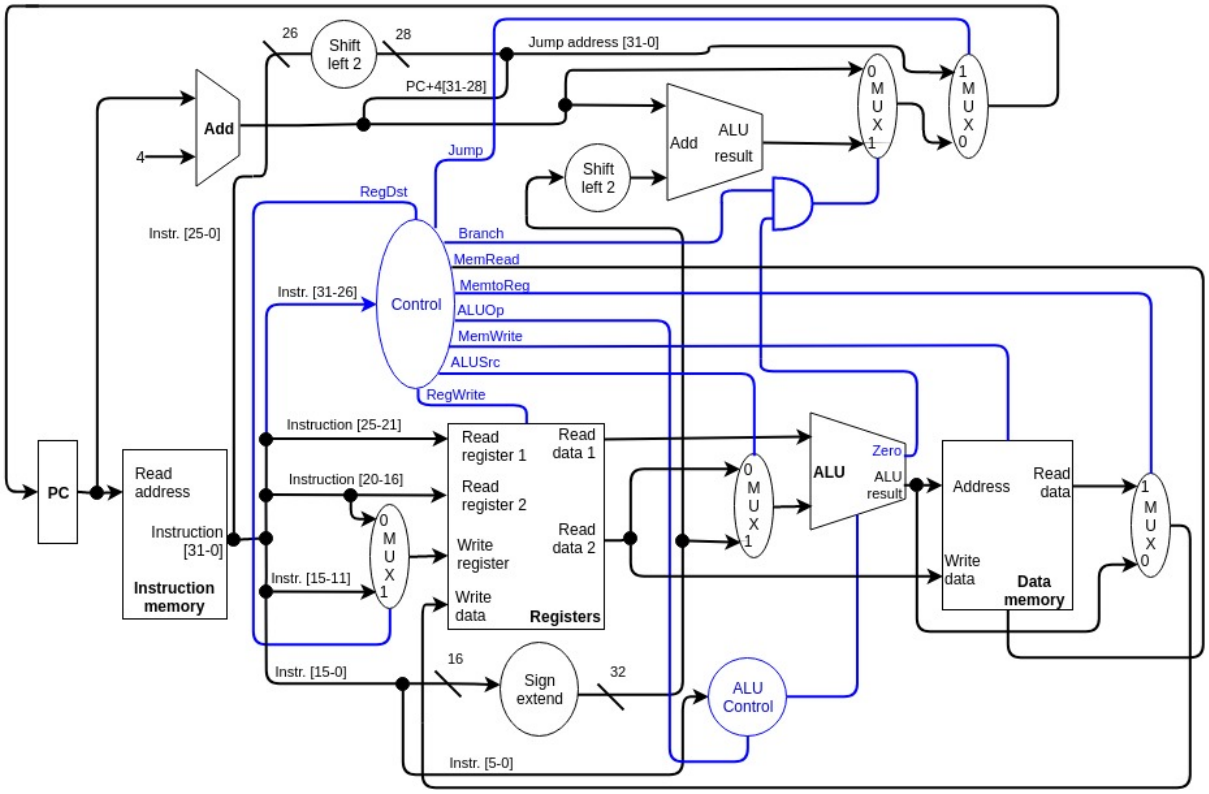
Uma FPGA é um **dispositivo lógico programável** que pode ser reprogramado um número indefinido de vezes após ser fabricada. Internamente FPGAs contem conjuntos de portas de elementos lógicos programáveis remanufaturados chamados de células. Uma única célula pode implementar uma rede de diversas portas lógicas que são alimentadas em *flip-flops*. Estes elementos lógicos são organizados internamente em uma configuração de matriz e são conectados automaticamente entre si utilizando uma rede de interconexão programável. Sua natureza reprogramável as torna muito eficientes para motivos acadêmicos.

As principais fabricantes são *Altera* e *Xilinx*, que possuem "Programas Universitários" com descontos e suporte à pesquisa que utilizem estes dispositivos. Mais especificamente, as placas da série **DE2** estão à frente do desenvolvimento educacional, pela abundância de interfaces para acomodar diversas aplicações. A DE2-115 apresenta 114.480 elementos lógicos, o maior oferecido na série Cyclone IV E, até 3,9-Mbits de RAM, e 266 multiplicadores.

3.7 Definições para implementação do projeto

Neste projeto será projetado um processador MIPS *Single-Cycle* que realizar todo o processo de uma operação em um único siclo. O diagrama é visível na **Figura 5**.

Figura 5 – Diagrama MIPS Single-cycle



4 Desenvolvimento

Nesta seção serão apresentadas as definições para este projeto de forma mais detalhada. O conjunto de instruções com o caminho de memória de cada tipo de instrução (R, I e J), apresentada a implementação dos componentes da CPU e da UC. O projeto seguira um conjunto de instruções RISC em uma arquitetura MIPS, fundamentados na **Seção 3** e na bibliografia escolhida.

4.1 Conjunto de Instruções

Na **Tabela 4** estão listados todas as instruções escolhidas para implementação. Os operandos podem ser **rd**, **rs** e **rt** são registradores, **I** o imediato, *shamt* é o *shift ammount* explicados na **Seção 3.3**. Já **M** é a memória e M[x] é o endereço x da memória, e **PC** é o *Program Counter* (contador de memória).

Tabela 4 – Conjunto escolhido de Instruções MIPS

Função	**Tipo**	**Instrução**	**Comentários**	
Aritméticas	R	*ADD*	$rd \leftarrow rs + rt$	soma de inteiros
	R	*SUB*	$rd \leftarrow rs - rt$	subtração de inteiros
	R	*SRA*	$rd \leftarrow rs >> shamt$	*shift* direita aritmético
	R	*SLA*	$rd \leftarrow rs << shamt$	*shift* esquerda aritmético
	R	*MUL*	$rd \leftarrow rs \times rt$	multiplicação
	R	*DIV*	$rd \leftarrow rs \div rt$	divisão
Lógicas	R	*AND*	$rd \leftarrow rs \wedge rt$	*and* lógico
	R	*MOD*	$rd \leftarrow rs \ \% \ rt$	módulo / resto
	R	*OR*	$rd \leftarrow rs \vee rt$	*or* lógico
	R	*XOR*	$rd \leftarrow rs \oplus rt$	*xor* lógico
	R	*SLL*	$rd \leftarrow rs \ll shamt$	*shift* esquerda lógico
	R	*SRL*	$rd \leftarrow rs \gg shamt$	*shift* direita lógico
Transferência de dados	I	*LD*	$rs \leftarrow M[I]$	*load word*
	I	*ST*	$M[I] \leftarrow rs$	*store word*
	I	*LDI*	$rs \leftarrow I$	*load immediate*
Controle de programa	J	*JMP*	$PC \leftarrow (PC + 4)[31 - 28] \| target$	*jump*
	I	*BEQ*	$if(rs = rt)\ goto\ PC + 4 + 4 * I$	desvio se igual
	I	*BNE*	$if(rs \neq rt)\ goto\ PC + 4 + 4 * I$	desvio se diferente

As operações das instruções foram escolhidas para suprir as necessidades apresentadas no início do curso, em que o processador deverá resolver problemas lógicos e aritméticos como calcular itens da Sequência de Fibonacci, resolver cálculos matemáticos de entrada, etc.

4.1.1 Mapeamento das Instruções e Caminhos no Diagrama

As instruções dos tipos *R-type*, *I-type* e *J-type* estão mapeadas visualmente respectivamente nas **Figuras 14 15 e 16** disponíveis no **Apêndice A**. As instruções da **Tabela 4** foram mapeadas de acordo com as categorias apresentadas na **Tabela 2**. Cada conjunto de bits das instruções tem sua utilidade e um caminho próprio ao longo das operações da CPU, descritos abaixo.

A instrução *ADD* (**Figura 14**), separada em **opcode**, **rs**, **rt**, **rd**, **shamt** e **func**, e terá o seguinte caminho. O **rs** e o **rt** são os endereços dos registradores que contém os operandos (na figura, *Instruction* [25-21] e *Instruction* [20-16] respectivamente), que irão para o *Read register 1* e o *Read register 2*. O **rd** é o registrador que irá salvar o resultado (*Instr.* [15-11]) e irá até o *Write register*, todos estes estão coloridos em verde na figura. Já o **opcode** irá direto para a *ALU (ULA) Control* que irá indicar para a *ULA* qual operação será realizada, neste caso, a soma. De *Read Data 1* e *Read data 2* saem os conteúdos dos registradores apontados por **rs** e **rt** que serão alimentados na ULA principal para realizar a soma. A saída da ULA não irá para a memória, mas passará 'direto' para o último multiplexador e retornara para a memória de registradores, onde será salvo no endereço definido por **rd**.

Já a instrução *ANDI* (**Figura 15**) utiliza o formato de **opcode**, **rs**, **rt** e ***immediate***. Onde a seção da instrução **rs** irá até *Read register 1*, que contém um operando e o registrador, e o **rt** irá até o *Write register* e irá salvar o resultado da operação. O Imediato (*Instr.* [15-0] irá passar pelo *Sign extend*, onde serão adicionados bits mais significativos enquanto preserva o sinal e valor do imediato. A *ULA Control* recebe o **opcode** e indica a operação de *AND* para a ULA principal. O imediato estendido e o *Data Read 1* são dados como entrada na ULA. O resultado não irá para a memória de dados, mas retornará para a memória de registradores e será salvo no local indicado pelo **rt**.

Nas duas instruções acima há um processo que ocorre em paralelo, que é a soma de 1 no PC - por utilizar uma memória em matriz pelo *Verilog*. Que indicará ao programa acessar os próximos 32 *bits* (4 *bytes*) da próxima instrução na memória de instruções - ***Instruction memory***.

A instrução *JMPR* tem como objetivo atualizar o PC conforme o **target** da instrução. A instrução possui o **opcode** e o **target**, que ocupa 26 *bits*. Os 26 *bits* da instrução serão somados aos 6 *bits* mais significativos do endereço atual mais 4, resultando em um endereço de 32 *bits* final. Este endereço de 32 *bits* é salvo no próprio PC, e irá indicar o local das próximas instruções à serem buscadas na memória de instruções.

4.1.2 Modos de Endereçamento

O conjunto de instruções apresentado na **Tabela 4** será implementado para adotar os seguintes modos de endereçamento: (i) imediato, (ii) direto, (iii) por registrador e (iv) relativo. Detalhados na **Seção 3.3.1**

4.2 Implementação da CPU - *Central Process Unit*

Abaixo serão apresentadas as implementações de cada componente, e os resultados obtidos (ondas das simulações) serão apresentadas na **Seção 5.2**

4.2.1 Multiplexadores, FlipFlops e Debouncer

Neste projeto foram utilizados diversos multiplexadores e *flipflops*. Os multiplexadores [8] são circuitos seletores, utilizados para "filtrar" entre diversos valores de entradas uma saída, foi utilizada uma versão disponível online[1]. Já os *flipflops* são circuitos de armazenamento, cada *flipflop* é capaz de armazenar um bit de informação. O *Debouncer* [9] é um circuito que evita múltiplos sinais elétricos ao acionar os botões do FPGA, foi utilizada uma versão disponível online[2] cuja entrada é o sinal de pressionamento dos botões e o *clock* da placa, e sua saída um sinal 'limpo' de ruídos.

Como estes dispositivos foram utilizados em disciplinas anteriores não foram inclusos seus códigos ao longo do texto, mas estão disponíveis na **Seção de Anexos**.

4.2.2 Banco de Registradores

Com discutido na **Seção 3.2.1** o MIPS possui 32 registradores de propósito geral, cada um com 32 *bits*. Este componente implementado conforme o **Código 4.1**, seguindo o diagrama na **Figura 5**, necessita de 5 entradas (*readRegister1*, *readRegister2*, *writeRegister*, *writeData* e *RegWrite*) mais 1 (*clock*); e 2 saídas (*readData1* e *readData2*) de dados.

Código 4.1 – Banco de Registradores

```
module registers (
  readRegister1, readRegister2, writeRegister,
  clock, RegWrite, user_number,
  writeData,
  readData1, readData2,
  toDisplay
);
  input [4:0] readRegister1, readRegister2, writeRegister;
  input clock, RegWrite;
```

[1] Multiplexador disponível em: <http://electrosofts.com/verilog/mux.html>
[2] *Debounceur* disponível em: <https://www.eewiki.net/pages/viewpage.action?pageId=13599139>

```
  input [31:0] writeData;
  input [5:0] user_number;

  output [31:0] readData1, readData2;
  output [31:0] toDisplay;

  reg [31:0] registerFile[31:0];

  always @ (posedge clock) begin
    if(RegWrite == 1) begin
      registerFile[writeRegister] = writeData;
    end
    registerFile[30] = user_number;
  end

  assign readData1 = registerFile[readRegister1];
  assign readData2 = registerFile[readRegister2];
  assign toDisplay = registerFile[31];

endmodule
```

4.2.3 ULA - Unidade Lógica e Aritmética

A ULA é responsável por aplicar as operações definidas pelo **opcode** em duas entradas (*data1* e *data2*) de dados. A operação será definida pelo sinal de entrada de 4 *bits* (*operation*) proveniente da ***ALU Control***. E possui duas saídas, *aluResult* resultado da operação sobre os dados, e *aluZero* utilizado para as operações *BNE* e *BEQ*.

No projeto há 3 ULAs à serem utilizadas, uma mais completa (**Código 4.2**) que irá realizar a maioria das instruções, e outras duas mais simples que sempre farão a função de soma, uma *ULAadd4* (**Código ??**) que sempre somará 4 ao PC e uma *ULAadd* (**Código 4.3**) que somará duas entradas de dados.

Código 4.2 – ULA

```
module ALU (
  data1, data2,
  operation,
  zero,
  aluResult
);
  input [31:0] data1, data2;
  input [5:0] operation;

  output zero;
  output reg [31:0] aluResult;
```

```
  always @ (data1 or data2 or operation) begin
    case (operation)
      6'b000001: aluResult = data1 + data2;     // add
      6'b000010: aluResult = data1 - data2;     // sub
      6'b000011: aluResult = data1 & data2;     // and
      6'b000100: aluResult = data1 | data2;     // or
      6'b000101: aluResult = data1 ^ data2;     // xor
      6'b000110: aluResult = ~data1;          // not
      6'b000111: aluResult = data1 << data2;    // shift left
      6'b001000: aluResult = data1 >> data2;    // shift right
      6'b001001: aluResult = data1 * data2;     // multiplicacao
      6'b001010: aluResult = data1 / data2;     // divisao
      6'b001011: aluResult = data1 % data2;     // modulo
      6'b100000: aluResult = data2;          // passa data2 direto - st -
    ldi
      6'b100001: aluResult = data1 - data2;     // beq
      6'b100010: aluResult = (data1 == data2); // bnq
      default: aluResult = data1;             // passa direto - ld
    endcase
  end

  assign zero = (aluResult == 0);
endmodule
```

Código 4.3 – ULAadd

```
module ALUadd (
 data1, data2,
 aluResult
);
 input wire [31:0] data1, data2;
 output reg [31:0] aluResult;

 always @ (data1 or data2) begin
  assign aluResult= data1 + data2;
 end
endmodule
```

Código 4.4 – ULAadd4

```
module ALUadd4 (
 data1,
 aluResult
);
 input wire [31:0] data1;
 output reg [31:0] aluResult;

 always @ (data1) begin
  assign aluResult = data1 + 1;
 end
endmodule
```

A ULA segue a implementação das operações conforme a **Tabela 5**

Perceba que no **Código 4.2** é recebida a entrada *ALUOp*, a qual define sua saída *zero* (para comandos *beq* e *bne*) e a saída *aluResult* como *data2* para comandos *I-Type*. Este subcomponente é considerado a *ULA Control* e será definido e detalhado na **Seção 4.3**.

Tabela 5 – Tradução de bit para operações da ULA

bits	operação
000000	ld (*data1* passa direto)
000001	sum
000010	sub
000011	and
000100	or
000101	xor
000110	not
000111	shift left
001000	shift right
001001	mult
001010	div
001011	mod

4.2.4 Extensor de bits

Este componente será utilizado nas seções ***immediate*** das instruções do tipo I, e nas seções **shamt** nas instruções R-type para operações de *shift*. Estes dois casos serão filtrados depois pelo multiplexador anterior à ULA principal. Conforme o **Código 4.5** apresenta a entrada identificada por *input16* com 16 *bits* que será estendida (*sign extend*, conforme explicado na **Seção 4.1.1**) para um *output* com 32 *bits* para uma saída *output32*.

Código 4.5 – Extensor de bits

```
module bitExtender (
  input16,
  output32
);
  input [15:0] input16;
  output reg [31:0] output32;

  always @ ( * ) begin
    output32 = input16;
    if(input16[15] == 1'b1) begin
      output32 = {16'b0000000000000000, input16};
    end
  end
endmodule
```

4.2.5 PC - *Program Counter*

De forma bem simples o PC é um registrador que guarda o endereço da instrução atual à ser executada. A implementação de um módulo somente para o PC é importante para abranger todas as suas funções principais. Por padrão ele funciona de forma síncrona ao *clock* e seu valor é modificado à cada iteração. Operações específicas podem realizar alterações diversas em seu valor, como *jump* e *branch*. Estas alterações específicas precisam

de sinais de controle que permitam a alteração correta do PC ou, até, reinicia-lo.

O **Código 4.6** apresenta a implementação utilizada, onde possui as entradas *clock* cujo próprio nome identifica sua função, *address* o novo endereço para ser guardado no registrador *programCounter* e buscada a próxima operação, *interrupt* um sinal capaz de interromper a atualização do registrador *programCounter*, *reset* um sinal de controle para reiniciar a máquina, e a variável *programCounter* que é um registrador de 8 *bits* para armazenar o endereço de memória e também a saída que indicará o endereço da próxima instrução.

Código 4.6 – Program Counter

```
module program_counter(clock, address, interrupt, reset, programCounter,
    progr);

  input wire clock, interrupt, reset;
  input wire [1:0] progr;
  input [31:0] address;
  output reg [31:0] programCounter = 0;

  always @( posedge clock ) begin
    if (reset) begin
      case (progr)
        2'b00: programCounter <= 0;
        2'b01: programCounter <= 25;
        2'b11: programCounter <= 35;
        default: programCounter <= address;
      endcase
    end
    else begin
      programCounter <= address;
    end
  end

endmodule
```

4.2.6 Memória

Na placa FPGA que será utilizada para a implementação deste projeto existe uma memória principal disponível, e é possível acessa-la. Porém foi apresentada durante aula que há muita dificuldade nesta tarefa. Alunos anteriormente utilizaram um processador já feito no *software Quartus* como uma espécie de DMA para esta memória.

Inicialmente foram implementadas as memórias abaixo para a funcionalidade do processador.

4.2.6.1 Memória de Dados e Saída

Esta memória irá permitir salvar dados do sistema. Inicialmente foi testada uma memória de 40 posições de 32 *bits*. E conforme apresentado no **Código 4.7** temos uma variável *ram* registradora de 40 bits que serão utilizados para emular a memória de saída

Código 4.7 – Memória de Saída

```
module mainMemory(clock, data_in, address, MemWrite, MemRead, data_out);
  input clock, MemWrite, MemRead;
  input [31:0] data_in;
  input [9:0] address;
  output reg [31:0] data_out;
  reg [9:0] addressRegister;

  reg [31:0] ram[40:0];
  always @ ( posedge clock ) begin
    if ( MemWrite ) begin
      ram[address] = data_in;
    end
    addressRegister = address;
  end
  always @ (MemRead) begin
    data_out = ram[address];
  end

endmodule
```

4.2.6.2 Memória de Instruções

A memória de instruções é semelhante à memória anterior porém ela é a que será acessada pelo endereço do PC e serão extraídas as instruções para os algoritmos programados. O Código 4.8 apresenta uma memória com 80 posições de 32 bits com a implementação de 3 algoritmos. A corretude dos algoritmos será apresentada posteriormente.

Código 4.8 – Memória de Instruções

```
module Instructions_memory(clock, address, instrucao);
  input clock;
  input [9:0] address;
  output [31:0] instrucao;
  integer clock0 = 0;

  reg [31:0] RAM[80:0];

  always @ (posedge clock) begin
    if (clock0 == 0) begin
    // programa 1: fibonacci
```

```
RAM[1] = 32'b100011_00000_11111_0000000000000001;
// ldi $31, 0
RAM[2] = 32'b101010_00000_11110_0000000000000000;
// st user_number $30
RAM[3] = 32'b100011_00000_11111_0000000000000001;
// ldi display_reg $31 1
RAM[4] = 32'b100010_00000_00000_0000000000000000;
// ld user_number $0
RAM[5] = 32'b100011_00000_00001_0000000000000001;
// ldi $1, 1
RAM[6] = 32'b100011_00000_00100_0000000000000000;
// ldi $4, 0
RAM[7] = 32'b100011_00000_00010_0000000000000001;
// ldi $2, 1
RAM[8] = 32'b100011_00000_00011_0000000000000001;
// ldi $3, 1
RAM[9] = 32'b000000_00000_00001_00000_00000_000010;
// sub $0, $0, $1
RAM[10] = 32'b000100_00000_00100_0000000000111101;
// beq $0 $4, pqp
RAM[11] = 32'b000000_00000_00001_00000_00000_000010;
// sub $0, $0, $1
RAM[12] = 32'b000100_00000_00100_0000000000111101;
// beq $0 $4, pqp
RAM[13] = 32'b000000_00010_00011_11111_00000_000001;
// *loop add $31, $2, $3
RAM[14] = 32'b000000_00000_00001_00000_00000_000010;
// sub $0, $0, $1
RAM[15] = 32'b000100_00000_00100_0000000000111101;
// beq $0 $4, pqp
RAM[16] = 32'b101010_00000_11111_0000000000000000;
// st fibonumber $31
RAM[17] = 32'b100010_00000_00010_0000000000000000;
// ld fibonumber $2
RAM[18] = 32'b000000_00010_00011_11111_00000_000001;
// add $31, $2, $3
RAM[19] = 32'b000000_00000_00001_00000_00000_000010;
// sub $0, $0, $1
RAM[20] = 32'b000100_00000_00100_0000000000111101;
// beq $0 $4, pqp
RAM[21] = 32'b101010_00000_11111_0000000000000000;
// st fibonumber $31
RAM[22] = 32'b100010_00000_00011_0000000000000000;
// ld fibonumber $3
RAM[23] = 32'b010000_00000000000000000000001100;
// jump to 14
```

```
    // programa 2: fatorial
      RAM[25] = 32'b100011_00000_11111_0000000000000010;
      // ldi $0, 0
      RAM[26] = 32'b101010_00000_11110_0000000000000000;
      // st $30 M[0]
      RAM[27] = 32'b100010_00000_11111_0000000000000000;
      // ld display_reg $31 user_number
      RAM[28] = 32'b100010_00000_00000_0000000000000000;
      // ld user_number $0
      RAM[29] = 32'b100011_00000_00001_0000000000000001;
      // ldi $1, 1
      RAM[30] = 32'b100011_00000_00010_0000000000000000;
      // ldi $2, 0
      RAM[31] = 32'b000000_00000_00001_00000_00000_000010;
      // *loop sub $0 = $0 - $1
      RAM[32] = 32'b000100_00000_00010_0000000000111101;
      // beq $0 $2, pqp
      RAM[33] = 32'b000000_11111_00000_11111_00000_001001;
      // $31 = $31 * $0
      RAM[34] = 32'b010000_00000000000000000000011111;
      // jump to 31

    // programa 3: sintetico
      RAM[35] = 32'b100011_00000_11111_0000000000000011;
      // ldi $31, 0
      RAM[36] = 32'b101010_00000_11110_0000000000000000;
      // st user_number $30 M[0]
      RAM[37] = 32'b100010_00000_11111_0000000000000000;
      //ld display_reg $31 user_number
      RAM[38] = 32'b100011_00000_00001_0000000000000010;
      // ldi $1, 2
      RAM[39] = 32'b000000_11111_00001_11111_00000_000111;
      // shift left $31 << 2
      RAM[40] = 32'b000000_11111_00001_11111_00000_001000;
      // shift left $31 >> 2

      clock0 <= 0;
    end
  end
  assign instrucao = RAM[address];

endmodule
```

4.3 Implementação da UC

A UC é um dispositivo crítico do processador, ela será responsável por coordenar sinais de controle entre diversos outros dispositivos direcionando o fluxo dos bits corretos para realizar a operação solicitada pelo programador. A UC foi implementada no formado de Máquina de Estados, nas quais as entradas definem uma única saída de sinais. A entrada são os 6 primeiros *bits* mais significativos da instrução, relativos ao *opcode* - a única subestrutura da nossa instrução presente nos três tipos de instruções **Tabela 2**. E para referência futura e melhor preparo na implementação da UC criou-se a **Tabela 6** abaixo.

Tabela 6 – Verdade para a Unidade de Controle

entrada ou saída	sinal	R-type / I-type	ld / ldi	st	beq / bne	jump
entradas	Op5	0	1	1	0	0
	Op4	0	0	0	0	1
	Op3	0	0	1	0	0
	Op2	0	0	0	1	0
	Op1	0	1	1	0 / 1	0
	Op0	0 / 1	0 / 1	0	0	0
saídas	RegDst	1	0	0	0	0
	ALUSrc	0 / 1	1	1	0	0
	MemtoReg	0	1 / 0	0	0	0
	RegWrite	1	1	0	0	0
	MemRead	0	1 / 0	0	0	0
	MemWrite	0	0	1	0	0
	Branch	0	0	0	1	0
	Jump	0	0	0	0	1
	ALUOp	000	011	011	100 / 101	000

Perceba também no diagrama do processador (**Figura 5**) que a UC possui um dispositivo separado, a **ALU Control**. Este outro módulo, que pode ser considerar como parte da UC, será responsável por um sinal de saída que indicará o valor da saída *zero*, da ULA, utilizada para os comandos *beq* e *bne*, e também definirá a saída *data2* em casos de uma instrução *I-Type*. Os códigos da UC está disponível nesta seção, **Códigos 4.9**. O código da *ALU Control* está acoplado à implementação da ULA disponível na **Seção 4.2.3**.

Código 4.9 – UC

```
module Unidade_de_controle(instrucao, regDst, jump, branch, memRead,
    memtoReg, aluOp, memWrite, aluSrc, regWrite);
  input [5:0] instrucao;

  output reg regDst, jump, branch, memRead, memtoReg, memWrite, aluSrc,
    regWrite;
  output reg [2:0] aluOp;

  always@ (instrucao) begin
```

```
    case(instrucao)
      6'b000000:begin // R-type logic arithmetic
        regDst    = 1'b1;
        aluSrc    = 1'b0;
        memtoReg  = 1'b0;
        regWrite  = 1'b1;
        memRead   = 1'b0;
        memWrite  = 1'b0;
        branch    = 1'b0;
        jump      = 1'b0;
        aluOp     = 3'b000;
      end
      6'b000001:begin // I-type logic arithmetic
        regDst    = 1'b1;
        aluSrc    = 1'b1;
        memtoReg  = 1'b0;
        regWrite  = 1'b1;
        memRead   = 1'b0;
        memWrite  = 1'b0;
        branch    = 1'b0;
        jump      = 1'b0;
        aluOp     = 3'b000;
      end
      6'b100010:begin // load word
        regDst    = 1'b0;
        aluSrc    = 1'b1;
        memtoReg  = 1'b1;
        regWrite  = 1'b1;
        memRead   = 1'b1;
        memWrite  = 1'b0;
        branch    = 1'b0;
        jump      = 1'b0;
        aluOp     = 3'b011;
      end
      6'b100011:begin // load word immediate
        regDst    = 1'b0;
        aluSrc    = 1'b1;
        memtoReg  = 1'b0;
        regWrite  = 1'b1;
        memRead   = 1'b0;
        memWrite  = 1'b0;
        branch    = 1'b0;
        jump      = 1'b0;
        aluOp     = 3'b011;
      end
      6'b101010:begin // store word
        regDst    = 1'b0;
```

```
        aluSrc   = 1'b1;
        memtoReg = 1'b0;
        regWrite = 1'b0;
        memRead  = 1'b0;
        memWrite = 1'b1;
        branch   = 1'b0;
        jump     = 1'b0;
        aluOp    = 3'b011;
      end
      6'b000100:begin // branch if equal
        regDst   = 1'b0;
        aluSrc   = 1'b0;
        memtoReg = 1'b0;
        regWrite = 1'b0;
        memRead  = 1'b0;
        memWrite = 1'b0;
        branch   = 1'b1;
        jump     = 1'b0;
        aluOp    = 3'b100;
      end
      6'b000110:begin // branch if not equal
        regDst   = 1'b0;
        aluSrc   = 1'b0;
        memtoReg = 1'b0;
        regWrite = 1'b0;
        memRead  = 1'b0;
        memWrite = 1'b0;
        branch   = 1'b1;
        jump     = 1'b0;
        aluOp    = 3'b101;
      end
      6'b010000:begin // jump
        regDst   = 1'b0;
        aluSrc   = 1'b0;
        memtoReg = 1'b0;
        regWrite = 1'b0;
        memRead  = 1'b0;
        memWrite = 1'b0;
        branch   = 1'b0;
        jump     = 1'b1;
        aluOp    = 3'b000;
      end
      default:begin
        regDst   = 1'b0;
        aluSrc   = 1'b0;
        memtoReg = 1'b0;
        regWrite = 1'b0;
```

```
        memRead   = 1'b0;
        memWrite = 1'b0;
        branch   = 1'b0;
        jump     = 1'b0;
        aluOp    = 3'b000;
      end
    endcase
  end

endmodule
```

A *ALUControl* recebe o sinal de saída de 3 *bits ALUOp*, e os últimos 6 *bits* da instrução relativos ao ***func*** (**Tabela 2**), e com isso ela deve definir a instrução final para que a ULA realize. O Código 4.10 abaixo irá tratar as instruções *I-Type*, que não são capazes de enviar uma instrução para a ULA nos seus *bits* menos significativos, pois estão sendo usados no imediato.

Código 4.10 – ALU Control

```
module ALUControl (
  instr, ALUOp,
  instr_out
);
  input [5:0] instr;
  input [2:0] ALUOp;

  output reg [5:0] instr_out;

  always @ (ALUOp) begin
    case (ALUOp)
      3'b001: begin
        instr_out = 6'b000000; // passa data1 original
      end
      3'b011: begin
        instr_out = 6'b100000; // passa data2 original - ldi
      end
      3'b100: begin
        instr_out = 6'b100001; // branch equal
      end
      3'b101: begin
        instr_out = 6'b100010; // branch not equal
      end
      default: begin
        instr_out = instr; // r-type
      end
    endcase
  end
```

```
endmodule
```

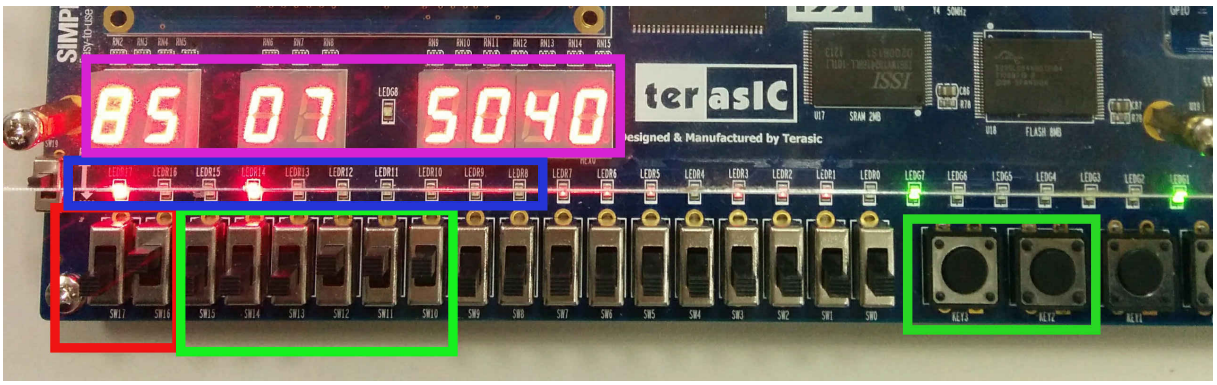
4.4 Interface de Comunicação - Entrada e Saída

Para interface de comunicação foram ligados os mecanismos de entrada da placa FPGA direto com um registrado. É possível ver no Código 3.2.1 que o $registerFile[30]$ recebe a $user\_number$ a cada clock, e o $regiterFile[31]$ está ligado à saída $toDisplay$. Esses dois registradores são "reservados" para comunicação entre usuário e o processador. Há também implementada a opção de *reset* e *program* (visíveis no Código **??**), que definem o PC para um dos três algoritmos implementados na Memória de instruções.

Uma observação é que foi levantado pelo professor a incoerência de definir diretamente valores aos registradores. E para futuros projetos não recomenda-se definir o PC como feito aqui, isto deve ser feito através de implementação de instruções na Memória de Instruções.

Abaixo foi adicionada a Figura 6 para servir como referência de uso da interface definida para este processador. No retângulo vermelho, as duas alavancas mais à esquerda, foram utilizadas como definidoras do programa, havendo 3 possíveis escolhas (00, 01 e 11). As seguintes alavancas à direita, no retângulo verde, são as que definem o número de entrada do usuário, este é um número que será utilizado no algoritmo. No caso, foi inserido o número 7, e foi calculado que $7! = 5.040$, como visto no *display* de 7 segmentos central e mais à direita dentro do retângulo roxo. O *display* de 7 segmentos mais à esquerda mostra o valor do PC. No retângulo azul está indicados os *leds* vermelhos da placa que foram conectados aos sinais da UC, e seguindo da esquerda à direita a ordem da Tabela 6 de cima para baixo. Os dois botões à direita, em um quadrado verde, foram utilizados como *reset* (esquerda) e *clock* (direita).

Figura 6 – Interface para o processador



Uma segunda observação para continuação deste projeto é não utilizar o *clock* de forma manual, um botão, mas utilizar o *clock* automático. E incluir um botão para entrada de informações (*enter*).

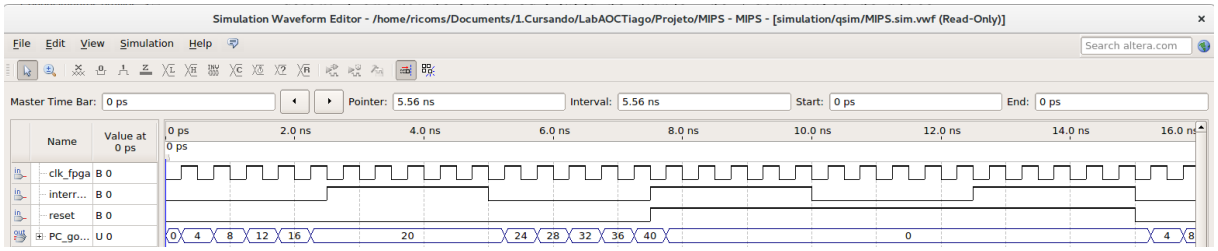
5 Resultados Obtidos e Discussões

5.1 Testes em forma de onda

No início desta seção serão apresentados os formatos de ondas dos módulos apresentados na seção anterior. A simulação dos circuitos, no *software Quartus* possibilita gerar os gráficos que serão utilizados aqui. Eles simulam a variação dos bits das entradas e com isso vemos se os bits das saídas se comportam como esperado para o projeto.

Como foi orientado a implementação da UC será entregue somente para o Ponto de Controle 3. Com isso os testes ficam limitados à cada dispositivo por vez. Logo segue primeiro a **Figura 7**, resultado do **Código C.1**.

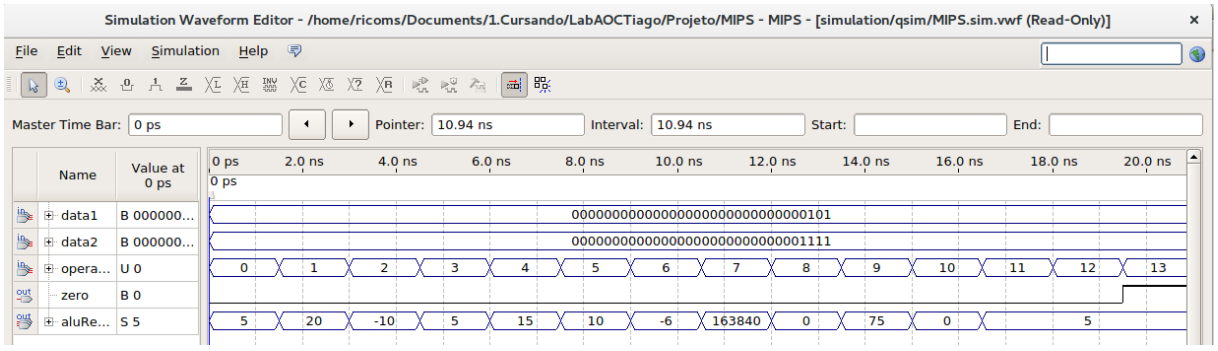
Figura 7 – Ondas dos dispositivos *Program Counter* e *ULAadd4*



Conforme vemos na figura acima as ondas de PC são iterativamente e de forma síncrona ao *clock* somadas de 4, mantida constante quando *interrupt* e reiniciada para 0, quando *reset* estão altos. O comportamento esperado para esta etapa do projeto.

Para testar a o **Código 4.2**, foi utilizado o **Código D.1** e gerada a **Figura 8** abaixo.

Figura 8 – Ondas do dispositivo ULA



Neste teste foram fixadas as entradas *data1* e *data2* no valores, respectivamente, 5 e 15. A entrada *operation* teve seus 4 bits menos significativos variando, de forma à abranger todas as operações previstas no **Código 4.2**. Com isso temos as saídas *zero* e

aluResult variando, de acordo com essas duas entradas e para cada operação. A saída *zero* é fácil ver que seu objetivo foi cumprido, ela irá indicar 1 somente nas operações 12 e 13 (BEQ e BNE), indicando se *data1* e *data2* são, respectivamente, iguais ou diferentes.

Analisando a onda da saída *aluResult*, é possível ver como as operações *store*, *add*, *sub*, *mult*, *div* e *modulo* (respectivamente, operações 0, 1, 2, 9, 10 e 11) tiveram seus valores corretos. Uma observação sobre a operação de divisão, que obteve como resultado 0, na verdade o resultado seria $5/15 = 0,33$, porém as operações possuem resultado de inteiros truncados, explicando o resultado 0. Para as operações restantes foi elaborada a **Tabela 7** para apresentar a transformação do resultado decimal para binário e comprovar se o comportamento está correto.

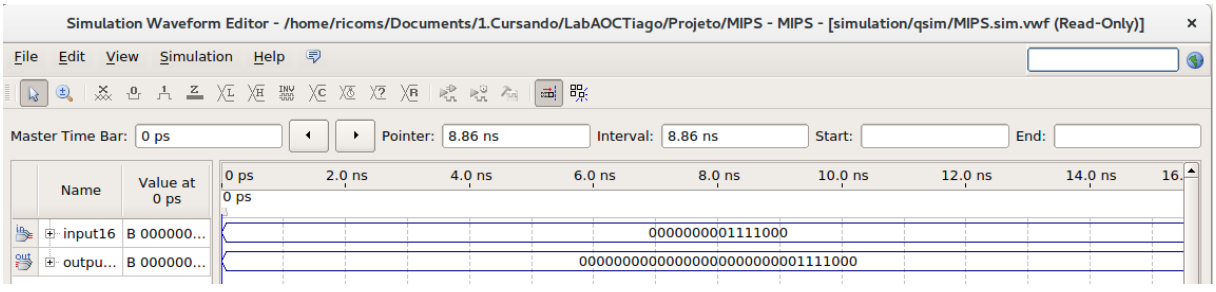
Tabela 7 – Testes de operações lógicas da ALU

	3	4	5	6	7	8
operation	and	or	xor	not	shift left	shift right
aluResult decimal	5	15	10	-6	16380	0
aluResult binário	(1)...0101	(0)...01111	(0)...01010	(1)...11010	(0)...1010...14*(0)	(0)...0

Analisando a **Tabela 7** acima, podemos ver como os resultados concordam com o esperado para cada uma das operações. Considere que (0) e (1) indicam continuação do número contido até completar os 32 *bits* e $14 * (0)$ indica que há 14 zeros (0) do lado daquele número binário.

O dispositivo *bit_extender* foi testado com o **Código E.1** e resultou no comportamento correto visto na **Figura 9** abaixo.

Figura 9 – Ondas do dispositivo *bit_extender*

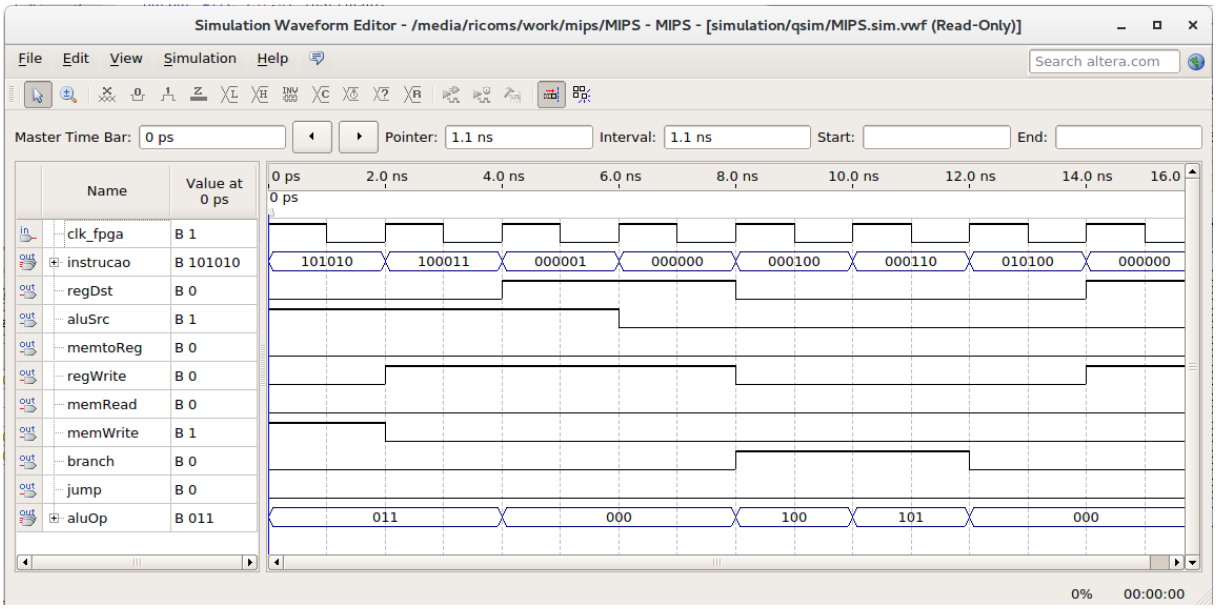


Com isto faltam somente os testes da **memória de dados**, do **banco de registradores** em conjunto com a UC e *ALU Control*. Os sinais de controle da UC foram testados com uma sequência de comandos fictícia, para avaliar o comportamento conforme previsto na **Tabela 6**.

A sequencia de comandos (*st, ldi, addi, mult, beq, bne, jump*) foi inserida na **Memória de Instruções** à qual foi integrada à estrutura mínima necessária para funcionamento do PC e à UC com o **Código F.1**, os módulos estão em um mesmo arquivo, porém a

implementação foram em arquivos separados conforme indicado na observação no código. E foi obtido as ondas vistas na **Figura 10**

Figura 10 – Ondas do dispositivo *UC*



Vemos que as tabelas acima confirma o comportamento previsto para cada um dos dispositivos. O próximo passo é integrar todos os dispositivos e garantir sua sincronicidade. Na próxima seção serão apresentados os resultados da integração do projeto.

5.2 Testes de implementação na placa FPGA

A integração de todos os dispositivos seguiu o diagrama apresentado na Figura 5. O Código 5.1 teve essa função.

Código 5.1 – Integração MIPS

```
module MIPS (
  input wire clk_fpga, Dreset, interrupt, Dclock,
  input wire [5:0] user_number,
  input wire [1:0] progr,
  output wire [6:0] ones, tens, hundreds, thousands, programOnes,
  programTens, programHundreds, programThousands, userOnes, userTens,
  userHundreds, userThousands,
  output wire clock, jump, branch, memRead, memtoReg, memWrite, aluSrc,
   regWrite, regDst, reset,
  output wire [2:0] aluOp
);
  wire zero;
  wire [5:0] ALUC_operacao;
  wire [31:0] PC_backfrom_add1, PC_goto_add1, instrucao, data1, data2,
   memDataOut,
```

```
muxBranch_out, muxJump_out, aluAddResult, mainAluResult, output32,
  toDisplay,
returnToRegisters, muxRegDst_out, ULASrc_out;

DeBounce db1(.clk(clk_fpga), .n_reset(1), .button_in(Dreset), .DB_out(
  reset));
//input clk, n_reset, button_in, // inputs
//output reg  DB_out // output
//divisor_frequencia df(.clk_alta_f(clk_fpga), .clk_baixa_f(clock));

DeBounce db2(.clk(clk_fpga), .n_reset(1), .button_in(Dclock), .DB_out(
  clock));

program_counter pc(.clock(clock), .address(muxJump_out),
            .interrupt(interrupt), .reset(~reset),
            .programCounter(PC_goto_add1), .progr(progr));

ALUadd1 AluAdd1(.data1(PC_goto_add1), .aluResult(PC_backfrom_add1));

ALUadd ALUadd(.data1(PC_backfrom_add1), .data2(output32), // << 2),
     .aluResult(aluAddResult));

MUX32bits muxBranch(.data1(PC_backfrom_add1), .data2(aluAddResult),
              .sign(branch & zero), .mux_out(muxBranch_out));
MUX32bits muxJump(.data1(muxBranch_out), .data2({{PC_backfrom_add1
  [31:26]}, {instrucao[25:0]}}), //<< 2}}),
            .sign(jump), .mux_out(muxJump_out));

Instructions_memory IM(.clock(clock), .address(PC_goto_add1), .instrucao(
  instrucao));

Unidade_de_controle UC(.instrucao(instrucao[31:26]), .regDst(regDst),
              .jump(jump), .branch(branch), .memRead(memRead),
              .memtoReg(memtoReg), .aluOp(aluOp), .memWrite(memWrite),
              .aluSrc(aluSrc), .regWrite(regWrite));

MUX5bits muxRegDst(.data1(instrucao[20:16]), .data2(instrucao[15:11]),
              .sign(regDst), .mux_out(muxRegDst_out));

registers rgs(.readRegister1(instrucao[25:21]), .readRegister2(instrucao
  [20:16]),
    .writeRegister(muxRegDst_out), .clock(clock), .RegWrite(regWrite), .
  user_number(user_number),
   .writeData(returnToRegisters), .readData1(data1), .readData2(data2), .
  toDisplay(toDisplay)
);
```

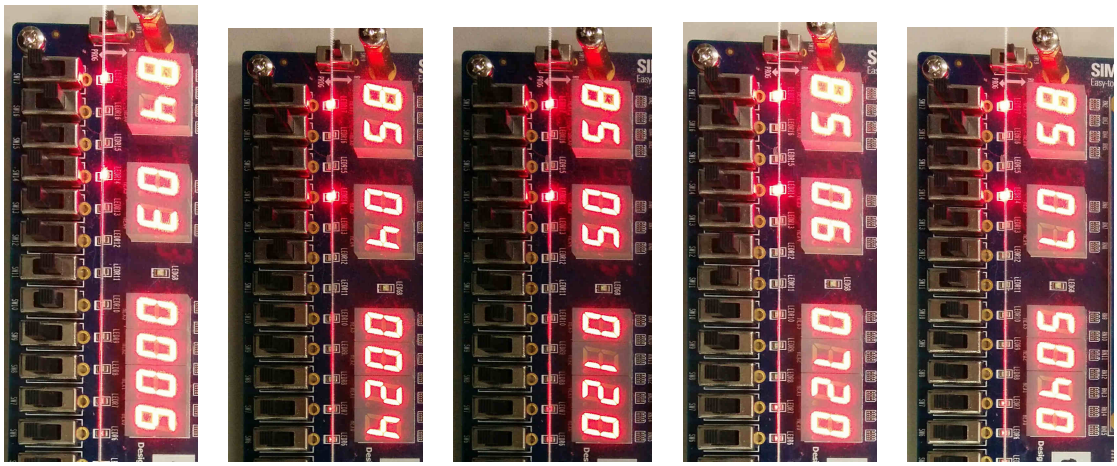
```
bitExtender be(.input16(instrucao[15:0]), .output32(output32));

MUX32bits ULASrc(.data1(data2), .data2(output32),
            .sign(aluSrc), .mux_out(ULASrc_out));

ALUControl ALUC(.instr(instrucao[5:0]), .ALUOp(aluOp), .instr_out(
  ALUC_operacao));

ALU MainALU(.data1(data1), .data2(ULASrc_out), .operation(ALUC_operacao),
    .zero(zero), .aluResult(mainAluResult));

mainMemory MainMem(.clock(clock), .data_in(data2), .address(mainAluResult
  ),
          .MemWrite(memWrite), .MemRead(memRead), .data_out(memDataOut));

MUX32bits finalMux(.data1(mainAluResult), .data2(memDataOut),
                .sign(memtoReg), .mux_out(returnToRegisters));

Output out1(.binary(toDisplay), .ones(ones), .tens(tens),
      .hundreds(hundreds), .thousands(thousands));

Output out2(.binary(PC_goto_add1), .ones(programOnes), .tens(programTens)
  ,
      .hundreds(programHundreds), .thousands(programThousands));

Output out3(.binary(32'b0 + user_number), .ones(userOnes), .tens(userTens
  ),
      .hundreds(userHundreds), .thousands(userThousands));

endmodule
```

E seu funcionamento foi testado com os três algoritmos implementados na memória de instruções. Cujos resultados são apresentados nas Figuras 11, 12 e 13.

Onde nas imagens da Figura 11 podemos ver os três *leds* indicando o valor do PC, que só indicará se já saiu do programa, o valor de entrada pelo usuário, e o valor de fatorial obtido para o número de entrada do usuário. E com isso temos os valores corretos calculados: (a) $3! = 6$, (b) $4! = 24$, (c) $5! = 120$, (d) $6! = 720$ e (e) $7! = 5040$.

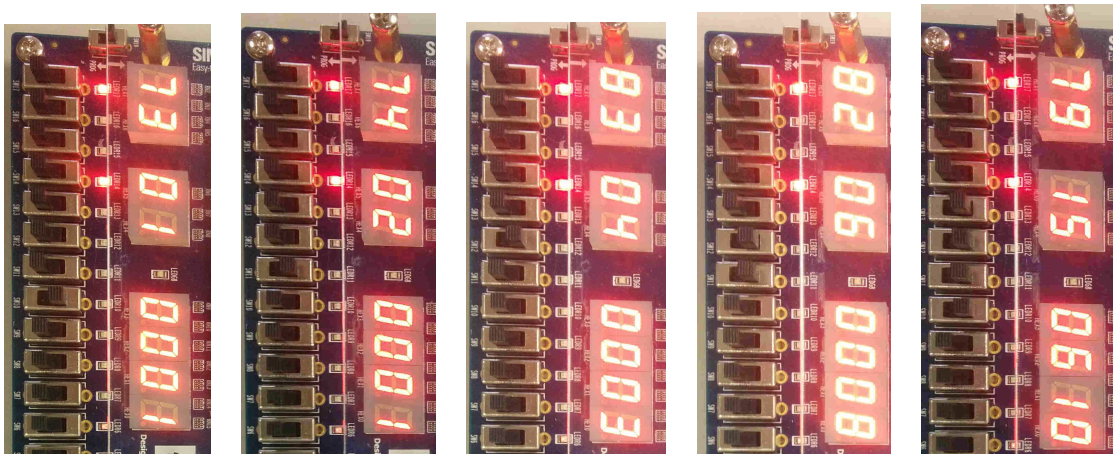
As imagens da Figura 12 calculando o enésimo número de fibonacci, sendo n o número de entrada do usuário. Com isso verificamos que o processador foi capaz de calcular corretamente os valores: (a) $fibo(1) = 1$, (b) $fibo(2) = 1$, (c) $fibo(4) = 3$, (d) $fibo(6) = 8$ e (e) $fibo(15) = 610$.

E o algoritmo sintético que simplesmente implementa as instruções de *shif left* de 2 *bits* e o *shift right* de 2 do resultado anterior. As entradas do usuário testadas foram 3 e 7.



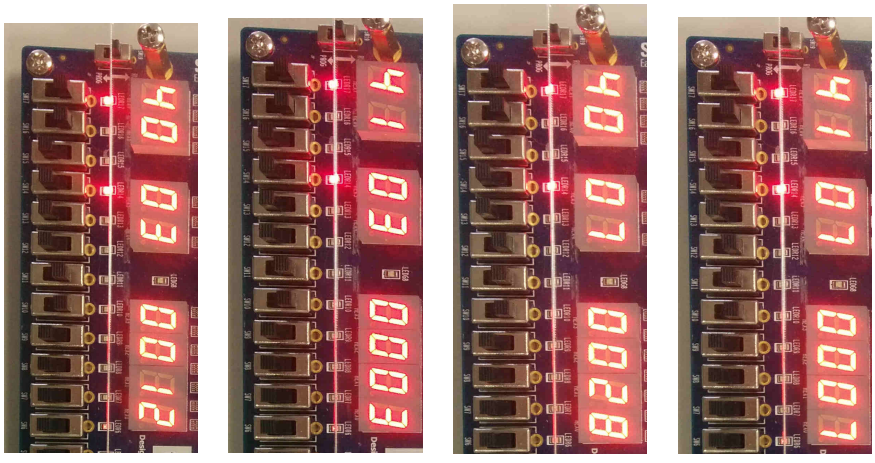
(a) 3! (b) 4! (c) 5! (d) 6! (e) 7!

Figura 11 – Resultados do algoritmo Fatorial



(a) fibo(1) (b) fibo(2) (c) fibo(4) (d) fibo(6) (e) fibo(15)

Figura 12 – Resultados do algoritmo Fibonacci



(a) << (2)3 (b) >> (2)12 (c) << (2)7 (d) >> (2)28

Figura 13 – Resultados do algoritmo sintético

6 Considerações Finais

O sistema computacional implementado com sucesso trata-se de um MIPS monociclo com, de momento, 18 instruções e com capacidade de receber códigos (implementados em binários em sua memória de instruções) e percorrer suas instruções para realizar os mais diversos algoritmos. O sistema computacional é o início para futuros projetos e foram levantadas algumas necessidades de melhorias necessárias: (1) retirar a definição do PC direto no registrador, e realizar através de instruções, (2) implementar com o *clock* automático e incluir um botão de entrada (*enter*), e sugestões ouvidas de alunos veteranos, (3) incluir a instrução *set less than* ou outro comparativo assimétrico.

A maior dificuldade do projeto foi no momento da integração, onde foram encontrados diversos *bugs*, definições anteriores equivocadas e a sincronização de todos os dispositivos. Para a sincronia do sistema foi importante entender que a leitura de informações de memória não devem estar ligadas ao *clock*, mas a escrita de informações sim. Esse foi o principal ponto, que levava à fios de transmissão de bits estarem com valores atrasados - relativos à instrução do clock anterior - o que dificultava bastante o entendimento e correção dos erros.

7 Conclusão

O projeto foi bem sucedido, tendo como produto um processador MIPS funcional, e simulável em uma placa FPGA conforme os objetivos iniciais da disciplina. Os objetivos específicos foram seguidos conforme o cronograma apresentado, somente com extensão do desenvolvimento da interface de comunicação que foi conjuntamente com a integração durante o mês de Junho.

Referências

1 STALLINGS, W. *Computer organization and architecture: designing for performance*. [S.l.]: Pearson Education India, 2000. Citado 2 vezes nas páginas 9 e 16.

2 MANO, M. M. et al. *Logic and computer design fundamentals*. [S.l.]: Prentice Hall, 2008. v. 3. Citado na página 12.

3 PATTERSON, D. A. Reduced instruction set computers. *Communications of the ACM*, ACM, v. 28, n. 1, p. 8–21, 1985. Citado na página 13.

4 STALLINGS, W. Reduced instruction set computer architecture. *Proceedings of the IEEE*, IEEE, v. 76, n. 1, p. 38–55, 1988. Citado 2 vezes nas páginas 14 e 15.

5 TANENBAUM, A. S. *Structured computer organization*. [S.l.]: Prentice Hall PTR, 1984. Citado na página 16.

6 WEBER, R. F. Arquitetura de computadores pessoais multimídia. *Porto Alegre*, 1995. Citado na página 17.

7 MURDOCCA, M. J.; HEURING, V. P. *Introdução à arquitetura de computadores*. [S.l.]: Elsevier, 2001. Citado na página 18.

8 PERLA, H. *Multiplexer implementation using verilog*. 2007. Accessado: 2017-05-06. Disponível em: <http://electrosofts.com/verilog/mux.html>. Citado na página 22.

9 SEMICONDUCTOR, L. *Debounce Logic Circuit*. 2013. Accessado: 2017-05-06. Disponível em: <https://www.eewiki.net/display/LOGIC/Debounce+Logic+Circuit+(with+Verilog+example>. Citado na página 22.

Apêndices

APÊNDICE A – Caminhos das Instruções

Figura 14 – Caminho de Instrução R-Type ADD

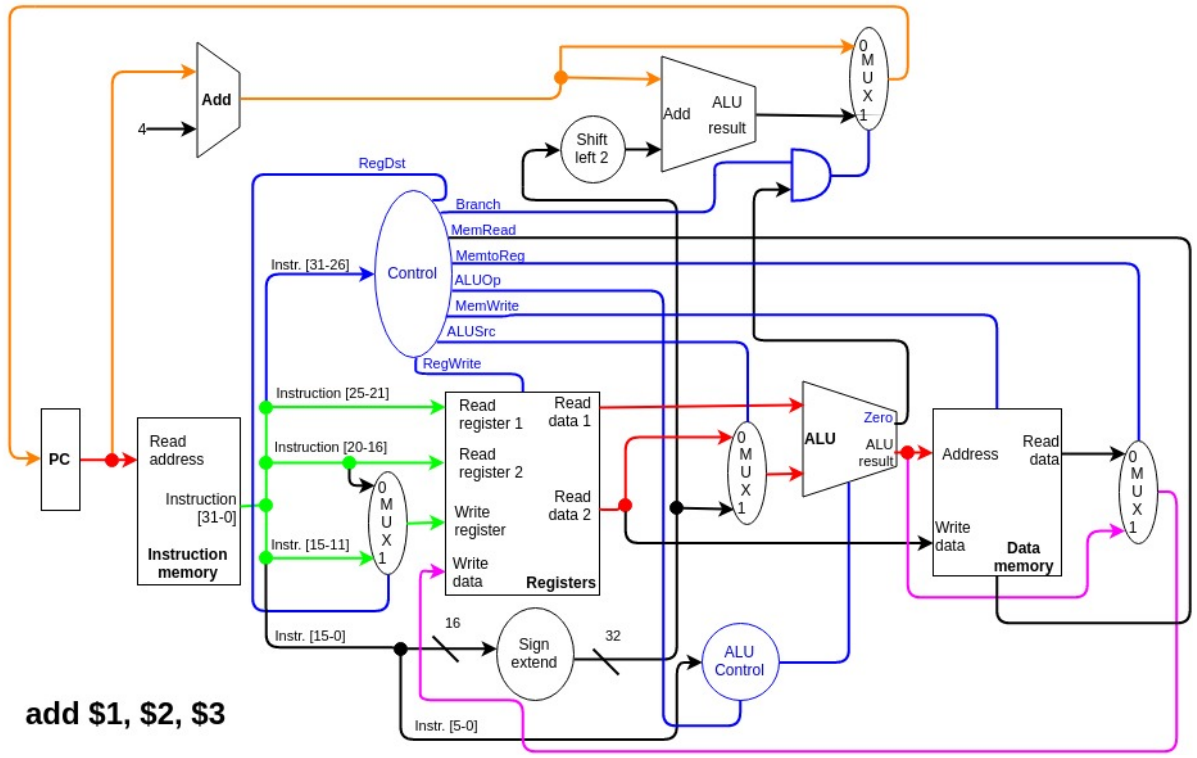


Figura 15 – Caminho de Instrução I-Type ANDI

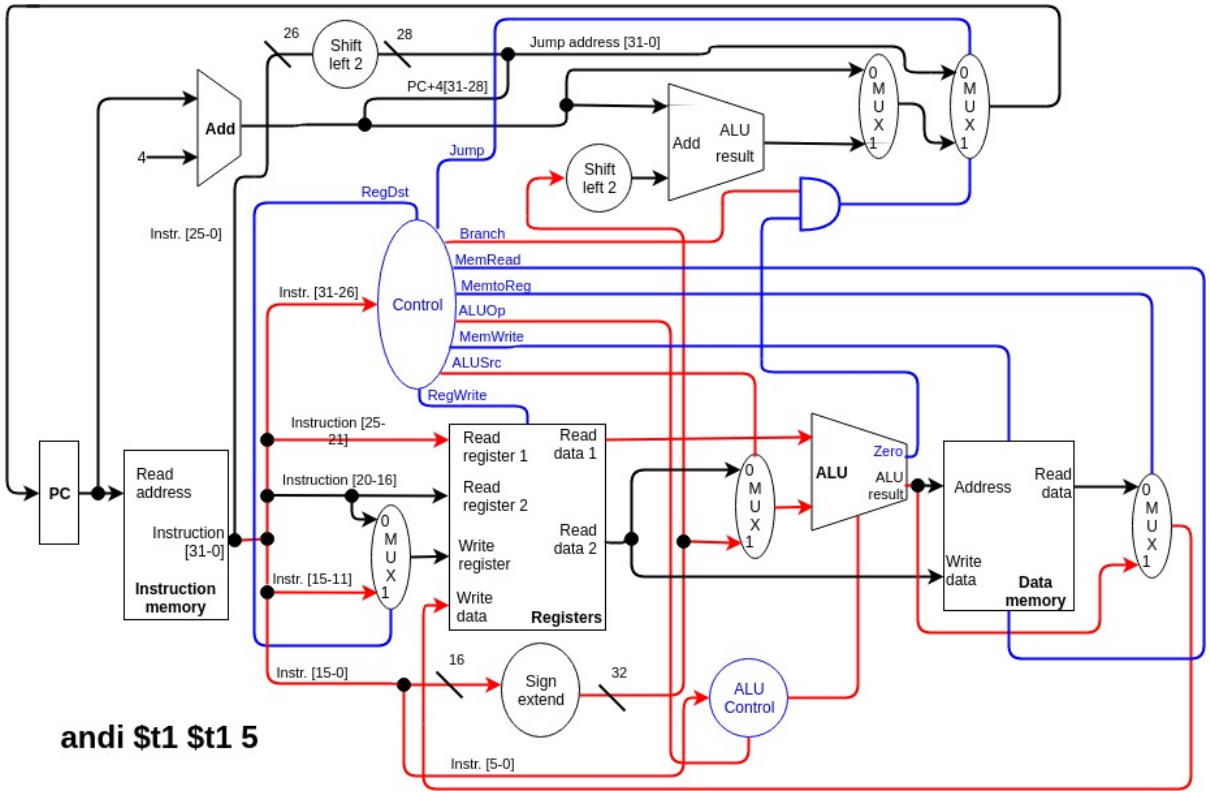
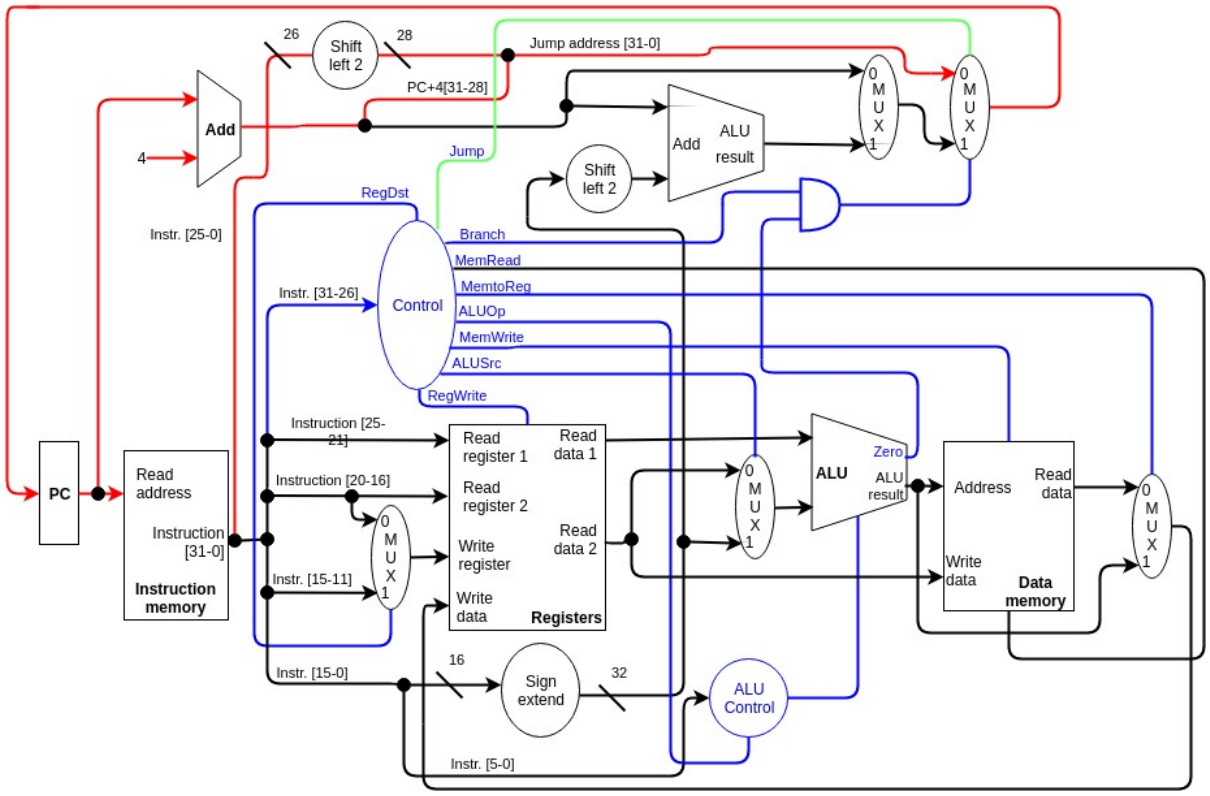


Figura 16 – Caminho de Instrução JUMP



APÊNDICE B – Código divisor de frequência

Código B.1 – Divisor de Frequência

```
module divisor_frequencia(
 clk_alta_f ,
 clk_baixa_f
);
 input clk_alta_f;
 parameter n = 26;
 // 26 eh o numero de registradores para que 50Mhz/(2^26) = 1Hz

 reg [n:0] flip_flop;

 // 50.000.000 em binario = 10111110101111000010000000
 wire [n:0] incremento = flip_flop[n] ? (1'b1):(1- 26'
   b10111110101111000010000000);
 wire [n:0] dN = flip_flop + incremento;

 always @( posedge clk_alta_f )
 begin
  flip_flop = dN;
 end
 output wire clk_baixa_f = ~flip_flop[n];
 // clock de saida ira mudar a cada 50M de mudancas do clock interno
endmodule
```

APÊNDICE C – Código teste de PC e ULAadd4

Código C.1 – teste parcial PC e ULAadd4

```
module MIPS (
  input wire clk_fpga, reset, interrupt,
  output wire [31:0] PC_goto_add4
);
  wire [31:0] PC_backfrom_add4;
  ALUadd4 add4(.data1(PC_goto_add4), .aluResult(PC_backfrom_add4));
  program_counter pc(
        .clock(clk_fpga), .address(PC_backfrom_add4),
            .interrupt(interrupt), .reset(reset),
            .programCounter(PC_goto_add4));
endmodule
```

APÊNDICE D – Código teste de ULA

Código D.1 – teste parcial ALU

```
module MIPS (
  input wire [31:0] operation, data1, data2,
  output wire zero,
  output wire [31:0] aluResult
);
  ALU alu(data1, data2, operation, zero, aluResult);
endmodule
```

APÊNDICE E – Código teste do *bit Extender*

Código E.1 – teste parcial bit_extender

```
module MIPS (
  input wire [15:0] input16,
  output wire [31:0] output32
);

  bitExtender be( .input16(input16), .output32(output32));

endmodule
```

APÊNDICE F – Código teste do *UC*

Código F.1 – teste parcial UC

```
module Instructions_memory(clock, address, instrucao);
  input clock;
  input [9:0] address;

  output wire [31:0] instrucao;
  integer clock0 = 0;
  reg [31:0] RAM[80:0];

  always @ ( posedge clock ) begin
    if (clock0==0) begin
      RAM[0] = 32'b10101000000000000000000000000000;
             // st 32'b101010 00000 00000 0000000000000000
      RAM[1] = 32'b10001100000000010000000000000001;
             // ldi 32'b100010 00000 00001 0000000000000001
      RAM[2] = 32'b00000100001000100000000000000001;
             // addi 32'b000001 00001 00010 0000000000000001
      RAM[3] = 32'b00000000001000100001100000001001;
             // mult 32'b000000 00001 00010 00011 00000 001001
      RAM[4] = 32'b00010000001000100000000000000000;
             // beq 32'b000100 00001 00010 0000000000000000
      RAM[5] = 32'b00011000001000100000000000000000;
             // bne 32'b000110 00001 00010 0000000000000000
      RAM[6] = 32'b01010000000000000000000000000010;
             // jump 32'b010100 00000000000000000000000010
       clock0 <= 0;
    end
  end
  assign instrucao = RAM[address];
endmodule

// o modulo acima foi definido em arquivo separado ao modulo abaixo

module MIPS (
  input wire clk_fpga, reset, interrupt,
  output wire regDst, jump, branch, memRead, memtoReg, memWrite, aluSrc,
    regWrite,
  output wire [1:0] aluOp,
  output wire [31:0] instrucao
);

  wire [31:0] PC_backfrom_add1;
```

```
  wire [31:0] PC_goto_add1;
  ALUadd1(.data1(PC_goto_add1), .aluResult(PC_backfrom_add1));
  program_counter pc(.clock(clk_fpga), .address(PC_backfrom_add1), .
    interrupt(interrupt), .reset(reset), .programCounter(PC_goto_add1));

  Instructions_memory(.clock(clk_fpga), .address(PC_goto_add1), .instrucao(
    instrucao));

  Unidade_de_controle(.instrucao(instrucao[31:26]), .regDst(regDst),
                      .jump(jump), .branch(branch), .memRead(memRead),
                      .memtoReg(memtoReg), .aluOp(aluOp), .memWrite(memWrite),
                      .aluSrc(aluSrc), .regWrite(regWrite));
endmodule
```

Anexos

ANEXO A – Anexo código *Debouncer*

Código A.1 – Debounce

```
////////////////////////// Button Debounceer
// Version History
// Version 1.0 04/11/2013 Tony Storey
// Initial Public Release
// Small Footprint Button Debouncer

`timescale 1 ns / 100 ps
module  DeBounce
  (
  input clk, n_reset, button_in, // inputs
  output reg  DB_out // output
  );
//// ----------------- internal constants ---------------
  parameter N = 11 ; // (2^ (21-1) )/ 38 MHz = 32 ms debounce time
////----------------- internal variables ----------------
  reg  [N-1 : 0]  q_reg; // timing regs
  reg  [N-1 : 0]  q_next;
  reg DFF1, DFF2; // input flip-flops
  wire q_add; // control flags
  wire q_reset;
//// ------------------------------------------------------

////contenious assignment for counter control
   // xor input flip flops to look for level chage to reset counter
  assign q_reset = (DFF1  ^ DFF2);
   // add to counter when q_reg msb is equal to 0
  assign  q_add = ~(q_reg[N-1]);

//// combo counter to manage q_next
  always @ ( q_reset, q_add, q_reg)
    begin
      case( {q_reset , q_add})
        2'b00 :
            q_next <= q_reg;
        2'b01 :
            q_next <= q_reg + 1;
        default :
            q_next <= { N {1'b0} };
      endcase
    end
```

```
//// Flip flop inputs and q_reg update
  always @ ( posedge clk )
    begin
      if(n_reset ==  1'b0)
        begin
          DFF1 <= 1'b0;
          DFF2 <= 1'b0;
          q_reg <= { N {1'b0} };
        end
      else
        begin
          DFF1 <= button_in;
          DFF2 <= DFF1;
          q_reg <= q_next;
        end
    end

//// counter control
  always @ ( posedge clk )
    begin
      if(q_reg[N-1] == 1'b1)
          DB_out <= DFF2;
      else
          DB_out <= DB_out;
    end

endmodule
```

ANEXO B – Anexo código Multiplexador

Código B.1 – Multiplexador

```
// Verilog code for Multiplexer implementation using assign
// File name: mux1.v
// by Harsha Perla for http://electrosofts.com
// harsha@electrosofts.com
// Available at http://electrosofts.com/verilog

module mux1( select, d, q );

input[1:0] select;
input[3:0] d;
output q;

wire q;
wire[1:0] select;
wire[3:0] d;

assign q = d[select];

endmodule
```