UNIVERSIDADE FEDERAL DA PARAÍBA - UFPB
CENTRO DE ENERGIAS ALTERNATIVAS E RENOVÁVEIS - CEAR
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Daniel Ribeiro dos Santos - 20170157528
Marcos Vinícius Lisboa Melo - 2016019081
Danrley Santos Felix - 11328463

Processador de ciclo único de 4 bits

João Pessoa
26 de abril de 2021

Daniel Ribeiro dos Santos - 20170157528
Marcos Vinícius Lisboa Melo - 2016019081
Danrley Santos Felix - 11328463

Processador de ciclo único de 4 bits

Relatório sobre a primeira avaliação da disciplina de arquitetura avançada para computação, com o foco na criação de um processador de ciclo único de 4 bits, no período 2020.2

Prof. Dr. José Maurício de Souza Filho

Universidade Federal da Paraíba - UFPB
Centro de Energias Alternativas e Renováveis - CEAR
Curso de Graduação em Engenharia Elétrica

João Pessoa
26 de abril de 2021

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Objetivos

O trabalho é dividido em duas grandes partes: desenvolvimento de um montador que traduz um código em *Assembly* (baixo nível) para linguagem de máquina (bits) e o projeto e concepção do processador em blocos lógicos usando uma placa FPGA, levanto em conta as instruções produzidas pelo montador.

Nas próximas seções serão apresentados os desenvolvimentos detalhados de cada etapa do projeto: *software* e *hardware*, dispostos em 3 capítulos que abordarão respectivamente: apresentação e construção do *set* de instruções ISA, concepção e codificação do *Mini-Assembler* e construção do Processador em blocos lógicos.

Para melhor compreensão serão apresentados vários recursos em formato de tabelas, fluxogramas e imagens prezando sempre pela apresentação didática da construção.

2 Desenvolvimento

Para o projeto de CPUs, é necessário determinar alguns pontos importantes antes de começar:

- Quantas instruções será necessário?
- Quantos registradores nossa CPU terá?
- Nosso processador realizará operações com quantos bits?
- Qual o número máximo de instruções na memória?
- Quantas posições de memória da RAM?

Estas perguntas são respondidas pelos requisitos do projeto, fornecido pelo Prof. Dr. José Maurício de Souza Filho. Estamos projetando um processador que realizará, no mínimo, 17 instruções básicas (fornecidas pelo discente), com 8 registradores de 4 bits. Nossa memória de instruções terá 64 linhas, enquanto a de dados possuirá 16.

Em sequência, será fornecido, em detalhes, o conjunto de instruções (ISA) do processador.

2.1 *Instructions Set Architecture (ISA)*

Antes do projeto do processador "físico"e, antes ainda, do compilador, precisamos saber o que exatamente o que nossa CPU será capaz de realizar. Dito isto, precisamos definir o conjunto de instruções. Como requerido, precisamos de 5 bits de OP-Code (o OP-Code será explicado mais adiante), resultando em, no máximo, $2^5 = 32$ instruções. Parâmetros do tipo IMEDIATO devem ser valores decimais representados em 4 bits, ou seja, no máximo, 15. Dito isto, decidimos implementar as seguintes instruções:

Instrução	**Operandos**	**Descrição**
ADD 00000	REG, REG, REG	Soma o valor de um registrador ao valor de outro registrador. Algoritmo: Operando1 = Operando 2 + Operando 3 Exemplo: ADD R0,R1,R2 -> R0 = R1 + R2
SUB 00001	REG, REG, REG	Subtrai o valor de um registrador ao valor de outro registrador. Algoritmo: Operando1 = Operando 2 - Operando 3 Exemplo: SUB R0,R1,R2 -> R0 = R1 - R2

ADDi 00010	REG, IM	Soma o valor de um imediato ao valor de um registrador. Algoritmo: Operando1 = Operando 2 + IM Exemplo: ADDi R0,R1,IM -> R0 = R1 + IM OBS.: Pode-se usar apenas um registrador. Exemplo: ADDi R0,IM -> R0 = R0 - IM OBS.: Imediatos devem ser escritos em formato decimal
SUBi 00011	REG, IM	Subtrai o valor de um imediato ao valor de um registrador. Algoritmo: Operando1 = Operando 2 - IM Exemplo: SUBi R0,R1,IM -> R0 = R1 - IM OBS.: Pode-se usar apenas um registrador. Exemplo: SUBi R0,IM -> R0 = R0 - IM OBS.: Imediatos devem ser escritos em formato decimal
MUL2 00100	REG,REG	Multiplica o valor de um registrador por 2. Algoritmo: Operando1 = Operando2«1 Exemplo: MUL2 R0,R1 -> R0 = R1«1
DIV2 00101	REG, REG	Divide o valor de um registrador por 2. Algoritmo: Operando1 = Operando2»1 Exemplo: DIV2 R0,R1 -> R0 = R1»1

CLR 00110	REG	Zera o conteúdo de um registrador. Algoritmo: Operando1 = 0 Exemplo: CLR R0 -> R0 = 0
RST 00111	Sem operando	Zera o conteúdo de todos os registradores e reinicia a execução da primeira instrução de memória.
MOV 01000	REG, REG	Copia o conteúdo de um registrador em outro registrador. Algoritmo: Operando1 = Operando2 Exemplo: MOV R0,R1 -> R0 = R1
JMP 01001	ENDEREÇO	Altera a sequência de execução das instruções definindo o endereço da próxima instrução a ser executada. Algoritmo: Operando1 - Operando2 Exemplo: CMP R0,R1 -> R0-R1
CMP 01010	REG,REG	Compara o conteúdo de dois registradores e atualiza os flags de status de acordo com o resultado (sem salvá-lo). Algoritmo: Sempre pula Exemplo: JMP 0x13 -> PC = 0x13 OBS.: O endereço deve sempre ser fornecido em formato hexadecimal e tem limite de 0x40.

JZ 01011	ENDEREÇO	Modifica o valor de PC para apontar para o endereço definido caso a flag de zero esteja em alto. Algoritmo: Pula se a flag de zero estiver em alto. Exemplo: JZ 0x13 -> PC = 0x13 OBS.: O endereço deve sempre ser fornecido em formato HEXADECIMAL e tem limite de 0x13. OBS1.: Antes de se utilizar a instrução deve-se confirmar que a flag de zero foi obtida através de instruções de SUB, DEC ou CMP.
INC 01100	REG, REG	Incrementa o conteúdo de um registrador. Algoritmo: Operando1 = Operand2 + 1 Exemplo: INC R0,R1 -> R0 = R1 + 1
DEC 01101	REG, REG	Decrementa o conteúdo de um registrador. Algoritmo: Operando1 = Operand2 - 1 Exemplo: INC R0,R1 -> R0 = R1 - 1
LOADM 01110	REG, ENDEREÇO	Carrega um dado da memória em um registrador. Algoritmo: Operando1 = $[0xF]$ Exemplo: LOAD R0,0xF -> R0 = $[0xF]$ OBS.: O endereço deve ser sempre fornecido em formato HEXADECIMAL e tem limite de 0xF.

STOREM 01111	REG, ENDEREÇO	Armazena um dado do registrador na memória. Algoritmo: $[0xF]$ = Operando1 Exemplo: STORE R0,0xF -> $[0xF]$ = R0 OBS.: O endereço deve ser sempre fornecido em formato HEXADECIMAL e tem limite de 0xF.
RIO 10000	REG	Lê os 4 bits disponíveis do barramento externo e põe no registrador desejado. Algoritmo: Operando1 = 4'b1111 Exemplo: RIO R0 -> R0 = 4'b1111
MOVi 10001	REG,IM	Copia um valor imediato para um registrador. Algoritmo: Operando1 = IM Exemplo: MOVi R0,IM -> R0 = IM

Tabela 1: ISA do processador desenvolvido

Após a ISA, podemos trabalhar no projeto do processador e no compilador em paralelo, pois ambos devem respeitar a instruções fornecidas.

2.2 Mini-Assembler

De posse da ISA devidamente projetada, conhecendo a sintaxe das instruções *Assembly* e o formato de palavra binária esperado na saída, foi possível desenvolver o tradutor de mnemônicos para linguagem de máquina (binária), denominado de *Mini-Assembler*. O *Mini-Assembler* atua de forma bem semelhante a um montador ou compilador tradicional, um arquivo de entrada é introduzido contendo caracteres ASCII, ou seja texto, e esse terá seu conteúdo convertido para dados binários, ou seja, bits, que serão utilizados posteriormente pelo decodificador. Vale salientar que neste cenário em específico o *Mini-Assembler* projetado recebe seus arquivos de entrada em formato ".txt"e não em formato ".asm", padrão tipíco do *Assembly* Intel. E também, sua saída gerará um arquivo ".txt"e não ".bin"pois esse arquivo precisará ser acessador pelo *Quartus* e seria bem mais complexo a interpretação como arquivo binário direto.

É sabido por todos os estudantes de Arquitetura de Computadores que cada montador ou *Assem-*

bler deterá características especiais quanto a forma como interpreta as instruções e a sintaxe lexical esperado como entrada, sendo assim instruções contidas fora do set da ISA, não serão compreendidas pelo programa. Foram implementados tratamentos para que seja possível ao usuário utilizar de "tabs", "espaços"e outros recursos visuais que não comprometem a montagem do arquivo de saída.

Sendo assim, não se faz necessário a estética das instruções está exatamente como pontuado na ISA, a robustez do *Assembler* construído permite a entrada de espaços excessivos e tabs, já prevendo possíveis deslizes por parte do programador. Caso seja detectado um código escrito fora do padrão esperado o *Assembler* exibirá uma mensagem de erro alertando ao usuário que deve ser feita uma verificação do código no arquivo de entrada. Ademais, caso o arquivo de entrada seja fornecido com caminho incorreto ou não exista o mesmo na pasta citada, também será exibida uma mensagem de erro alertando para correção. Entretanto, caso a montagem ocorra sem erros, o usuário poderá apreciar a mensagem de sucesso e conferir o arquivo resultante sempre intitulado "bin.txt".

O *Assembler* desenvolvido neste projeto foi escrito em linguagem de programação Python v3.7. A Fig. 1 apresenta o fluxograma de funcionamento geral do código, desde o arquivo de entrada até o de saída. Mais detalhes sobre o funcionamento do código e funções podem ser verificados no Apêndice e através do GitHub que contém os códigos supracitados. Em vias gerais, o código funciona com 4 funções principais que são resposáveis pelo *parse* do texto de entrada, formatação, seleção de operação e por fim escrita em saída, a parte de seleção de operação pode ser considerada o ponto chave para o sucesso do *Assembler* e funciona com um encadeamento de condicionais para melhor selecionar a instrução inserida.

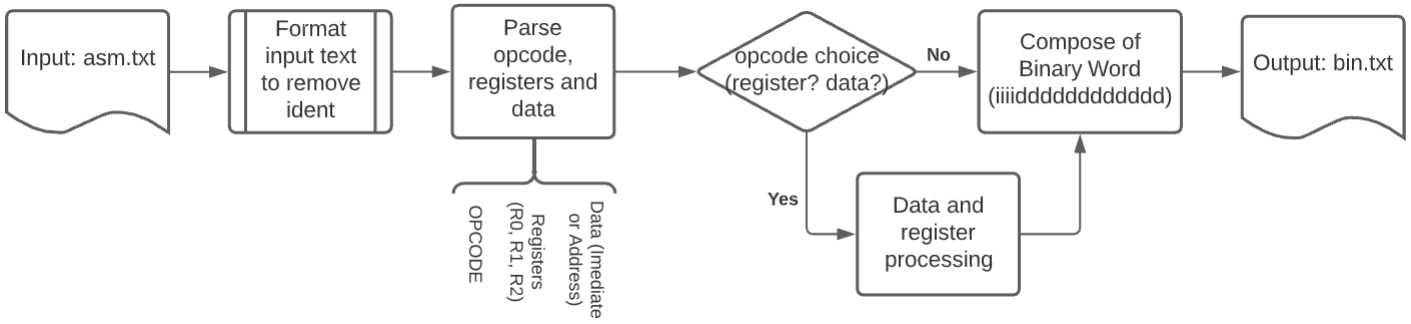

Figura 1: Diagrama de blocos do *Mini-Assembler desenvolvido.*

Cada linha do código de entrada é separada em diferentes *strings*, utilizando como separadores entre dados o carácter de vírgula. A *string* do mnemônico é o que definirá qual o condicional acessado e assim determinará o opcode utilizado e por fim a sequência de composição da palavra de 17 bits. Após feito o *parse* (pode ser traduzido como separação de elementos) entre mnemônico, registradores e dados é realizada a tradução binária dos dados inseridos e por fim a escrita da linha no arquivo de saída "bin.txt", vale salientar que o loop de tradução percorrerá todas as linhas contidas no arquivo de entrada.

Para utilizar o *Mini-Assembler*, é necessário que exista o arquivo de entrada de text contendo as instruções e em formato ".txt"e deverá ser executado o *script* em Python intitulado *assembler.py*, que receberá como argumento o caminho relativo ao arquivo de entrada, caso estejam na mesma pasta, apenas será necessário informar o nome do arquivo (sempre com extensão). Tomando por base a Fig. 1 citada anteriormente, teremos a entrada o arquivo "asm.txt"e o arquivo de saída será o "bin.txt"a ser usado no simulador.

2.3 Hardware

Com as instruções em mãos, pode-se começar o projeto do hardware. Para isto, foi-se utilizado o software "Quartus II 13.0spi Web Edition", podendo então simular o processador em uma FPGA.

Em resumo, o processador possui algumas estruturas bem características, como:

- Banco de Registradores
- Memória de Instrução
- Memória de Dados (RAM)
- Program Counter (PC)
- ULA
- Decodificador

Serão analisados com detalhes cada bloco do hardware.

2.3.1 Banco de Registradores

O banco de registradores possui todos os registradores necessários no nosso processador. Os registradores são essenciais para o funcionamento da CPU, pois é a memória mais próxima e mais frequentemente utilizada para as operações em geral. Estes são estruturas síncronas, ou seja, dependem de um clock externo para seu funcionamento. Na atual implementação, os registradores mudam os valores na borda de subida. A existência dos mesmos dentro do processador acelera o processamento, pois os dados que estão sendo manipulados ficam armazenados próximo dos recursos de processamento (ULA, por exemplo), o que reduz os acessos feitos à memória. Em outras palavras a vantagem de um registrador frente a uma posição de memória é a sua tamanha versatilidade de movimentação de bits.

A Fig 2 abaixo apresenta bloco funcional gerado para um de 4 bits.

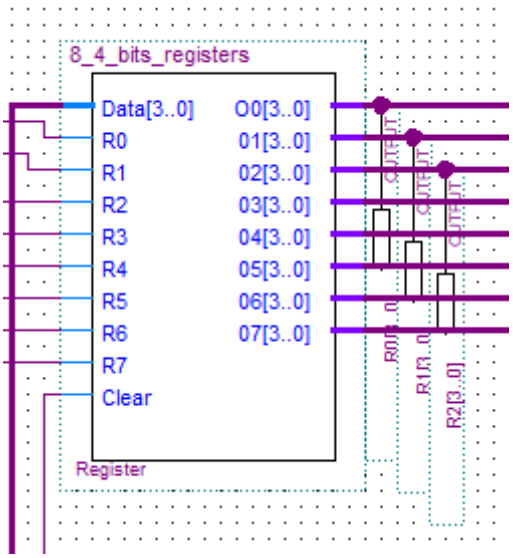

Figura 2: Registrador de 4 Bits

2.3.2 Unidade Lógica e Aritmética (ULA)

Uma Unidade Lógica Aritmética (ULA) pode ser encontrada em diversos processadores de dados. Esta é a parte do processador que realmente efetua cálculos aritméticos. Hoje em dia uma ULA pode realizar as seguintes operações:

- Operações aritméticas com inteiros;
- Operações lógicas bit a bit And, Not, Or, XOR;
- Operações de deslocamento de bits;
- Operações de divisão e multiplicação.

O deslocamento binário pode ser interpretado como uma multiplicação ou divisão por 2.

Os sinais de controle da ULA são gerados pela unidade de controle de acordo com a instrução a ser executada. A unidade de controle também gera sinais adicionais para o controle dos componentes externos a ULA que estão integrados ao caminho de dados, como a leitura/escrita de dados em registradores/memórias externas.

No projeto em questão na parte da ULA, há dois seletores na entrada (S0 e S1), ao qual de acordo com os valores dos mesmos uma operação matemática será realizada. As entradas A e B contém os dados dos operandos e Cin será o "carry in"da operação. O carry para adição de dois números binários é setado como 1 se o resultado dessa operação for maior que o conjunto de bits especificado possa representar (neste caso, 4 bits). Com isso temos as saídas Cout e G que representam respectivamente o bit "vai um"e o resultado da operação. Nas instruções lógicas e aritméticas é preciso armazenar o resultado produzido pela ULA em um registrador.

A Fig 3 abaixo apresenta bloco funcional gerado para ULA.

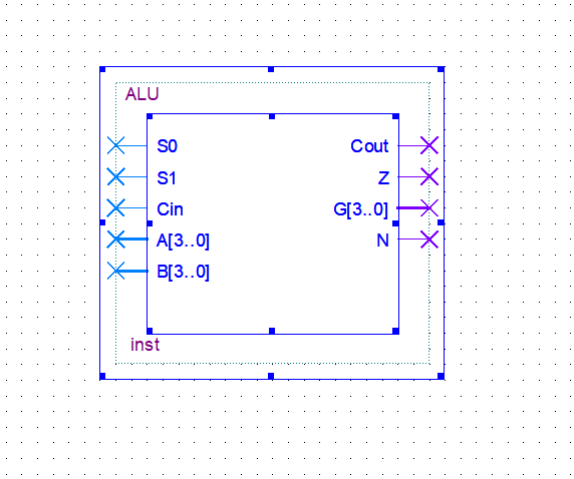

Figura 3: Unidade Aritmética e Lógica

Também é possível observar na Tabela 2 a estrutura de Tabela Verdade, na qual pode-se notar os códigos seletores de entrada e as operações subsequentes realizadas após a seleção.

S1	S0	Cin	Saída
0	0	0	G = B
0	0	1	G = A + 1
0	1	0	G = A + B
0	1	1	G = $A * 2$
1	0	0	G = A/2
1	0	1	G = A - B
1	1	0	G = A - 1
1	1	1	G = 0

Tabela 2: Tabela Verdade: ULA

2.3.3 Program Counting (PC)

Contador de programa (Program counter - PC), também conhecido como registro de endereço de instrução ou ponteiro de instrução, é um tipo de registrador encontrado na unidade central de processamento de um computador.

O contador de programa, como um dos vários registros diferentes embutidos na CPU, executa a tarefa de receber cada uma das instruções na sequência de tarefas. Manter a sequência lógica simplifica a progressão para frente de cada etapa, concluindo a tarefa. Essa progressão lógica é mantida, apontando para os dados que serão usados a seguir, mesmo quando os dados anteriores estiverem em uso, e rapidamente apontando para os próximos dados na sequência, à medida que cada instrução é executada por sua vez. A função de um contador de programa é essencial para a execução bem-sucedida das instruções envolvidas em qualquer tarefa. Ao apontar o caminho para cada instrução na sequência, o contador ajuda a fornecer uma execução lógica das etapas que acabam levando à conclusão da tarefa com rapidez e eficiência

Por se tratar de um componente síncrono, o mesmo está associado a um clock. A cada borda de subida o PC incrementa 1 e passa para a próxima instrução a ser lida. A instrução de salto é totalmente vinculada a este componente, pois ao ser detectada, o valor de PC (entrada - Salto[5:0]) passa a ser o endereço correspondente a instrução recebida.

A Fig 4 abaixo apresenta um bloco funcional gerado para o PC.

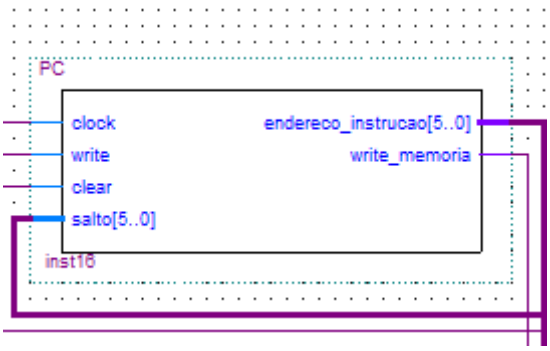

Figura 4: Program Counting

2.3.4 Memória de Instruções

Para manusear os códigos de instrução, a CPU necessitará de um registrador para armazenar os códigos de operação. O código de instrução é armazenado no registro denominado memória de instrução. A CPU sempre irá interpretar o conteúdo do registrador de instrução como sendo um

código de operação. Após o armazenamento dos códigos de instrução no Registro de Instrução, inicia-se o processo de decodificação do código de operação.

A Fig 5 abaixo apresenta um bloco funcional gerado para memória de instruções.

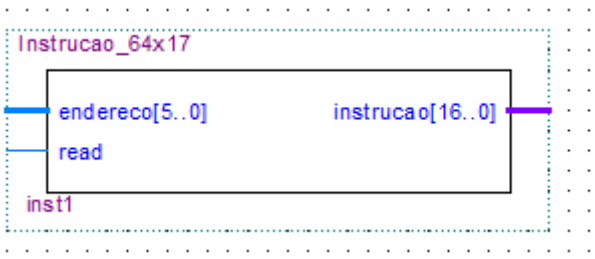

Figura 5: Memória de Instruções

2.3.5 Memória de Dados

A Memória de acesso randômico (do inglês Random Access Memory, frequentemente abreviado para RAM) é um tipo de memória que permite a leitura e a escrita. Isso fornece ao processador fácil acesso as informações essenciais para executar os programas, pois a mesma funciona com maior velocidade.

No caso em questão tem-se uma memória 16x4, ou seja, 16 palavras de 4 bits. Como a mesma funciona de forma síncrona necessita-se de um clock. As entradas "write"e "read", como o nome já sugere, servem para a leitura e escrita dos dados. A entrada "dado-in"faz parte do caminho de dados e que serão lidos na memória.

A Fig 6 abaixo apresenta um bloco funcional gerado para memória de dados (RAM).

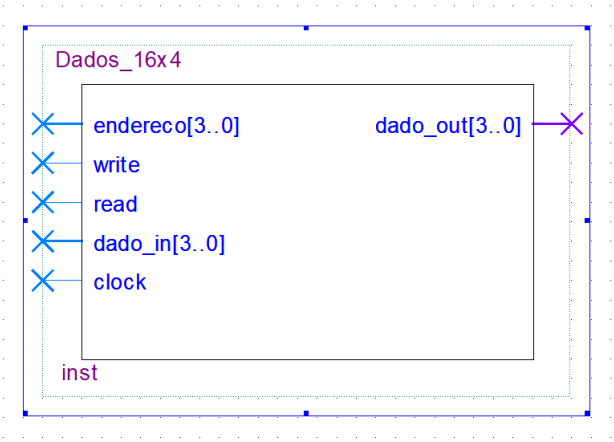

Figura 6: Memória de Dados

2.3.6 Decodificador

O decodificador é o limiar entre software (linguagem de máquina) e hardware (palavra de controle), e pode ser considerado uma das estruturas mais importantes do processador. É esta estrutura que determina exatamente como as operações serão realizadas na CPU.

A estrutura de contagem de instrução (Program Counter - PC) mostra qual instrução será lida da memória. Então, o decodificador recupera essa instrução, que, no nosso caso, terá 17 bits ("iiii-idddddddddddd"), dos quais os 5 primeiros contém a instrução a ser realizada e os outros 12 possuem

os parâmetro das instruções. O decodificador vai modificar seus bits de saída de acordo com as instruções. Tais bits estão interligados em todas as unidades de controle do sistema, como os MUXs e os DEMUXs, os bits de seleção da ULA (ALU0, ALU1 e Cin), o clear do PC e dos registradores, os endereços de saltos e os valores imediatos. Há também os bits de escrita, que permitem escrever nos registradores (Enable_Write) ou na memória de dados (Write_Ram). Na 7, temos o bloco do decodificador do processador.

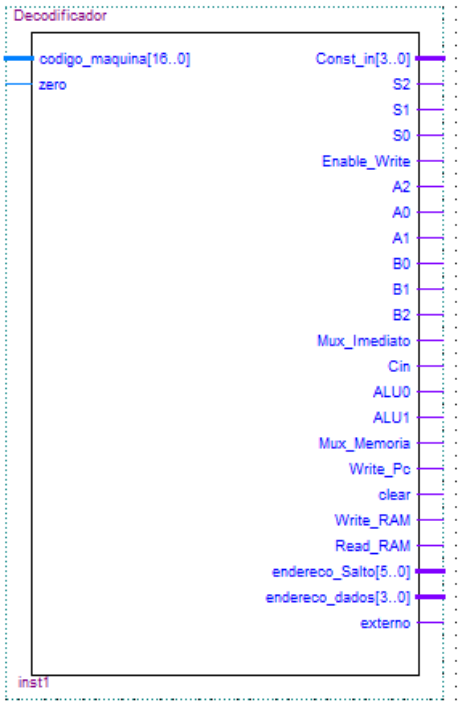

Figura 7: Decodificador

Por motivos de simplicidade, o decodificador foi implementado por meio de código, utilizando a linguagem *System Verilog*.

2.3.7 Processador

Conforme visto anteriormente nele abrange, principalmente, a ULA e os registradores. Neste projeto constituiu-se de 8 registradores de pelo menos 4 bits, uma ULA que realiza todas as operações aritméticas com diversas buscas na memória de acordo com a palavra de controle estabelecida, sendo que as operações do processador serão realizadas em único ciclo de *clock*, portanto RISC. Em síntese, o mesmo irá executar as instruções seguindo estes passos: busca, execução e resultado. A decodificação de uma instrução (vista no tópico anterior) pertence à seção de controle do processador.

O passo de busca neste processador necessita de três componentes: dois elementos de estado, que seriam o PC (armazena a próxima instrução a ser executada) e a memória de instruções, e por fim mais um somador como o terceiro elemento (calcula o endereço da próxima instrução a ser executada). O somador é quem incrementa o valor de PC. Na execução, dependendo do tipo (se é ou não uma operação lógica ou aritmética), é necessário realizar a leitura de certos registradores. No caso da ULA (instruções de add, sub, div etc), dois registradores (A e B) é o essencial para realizar as operações matemáticas. Neste caso o resultado se dá em um terceiro registrador e todos eles fazem parte do nosso banco de registradores vistos em tópicos anteriores (conforme a Fig 8).

A Fig 8 abaixo representa, em diagrama de blocos, o processador do projeto realizado.

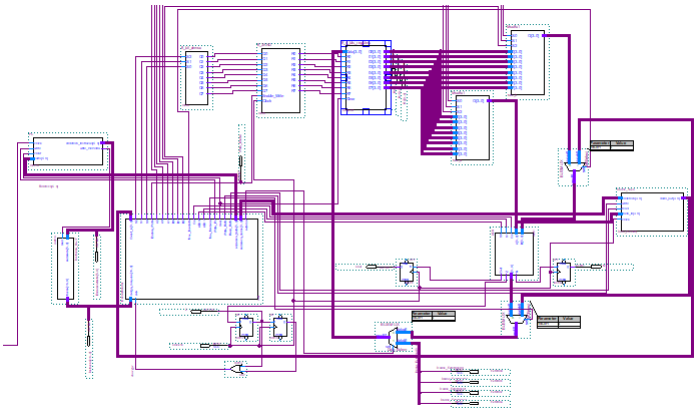

Figura 8: Processador

2.3.8 Simulação

Após a montagem do processador, alguns códigos em assembly foram testados. O ambiente de teste é integrado ao próprio quartus, chamado *University Program VWF*. Analisou-se, então, os valores dos registradores para diversas instruções. Note que temos dois valores de clock nas simulações. Isto se dá pois o clock de atualização dos registradores deve ser minimamente atrasado ao clock para obtenção das instruções (que, naturalmente, ocorre em um circuito, mas na simulação não).

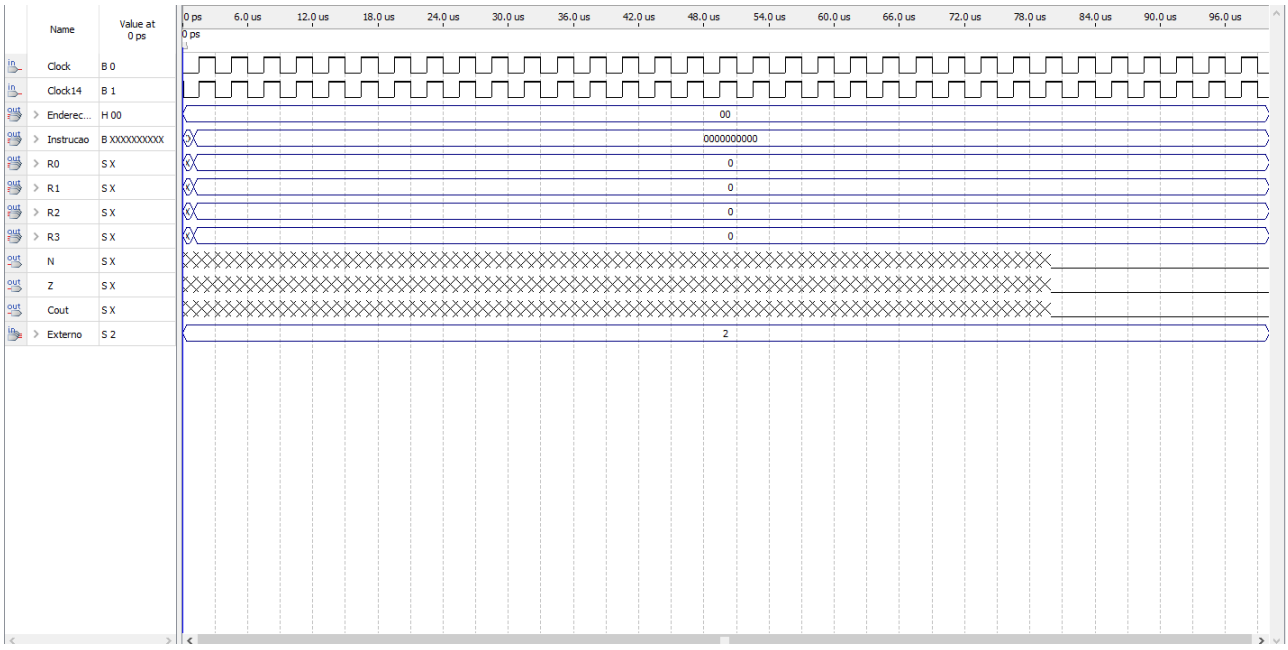

Figura 9: Estrutura da Simulação

Veremos então alguns exemplos de simulação (todos os arquivos estão na pasta do projeto).

- **Exemplo 1**
  Para o primeiro exemplo, temos as seguintes instruções:

```
movi r0,5
addi r1,5
cmp r1,r0
```

jz 0x00
movi r2,1
rst

O resultado está mostrado na figura 10.

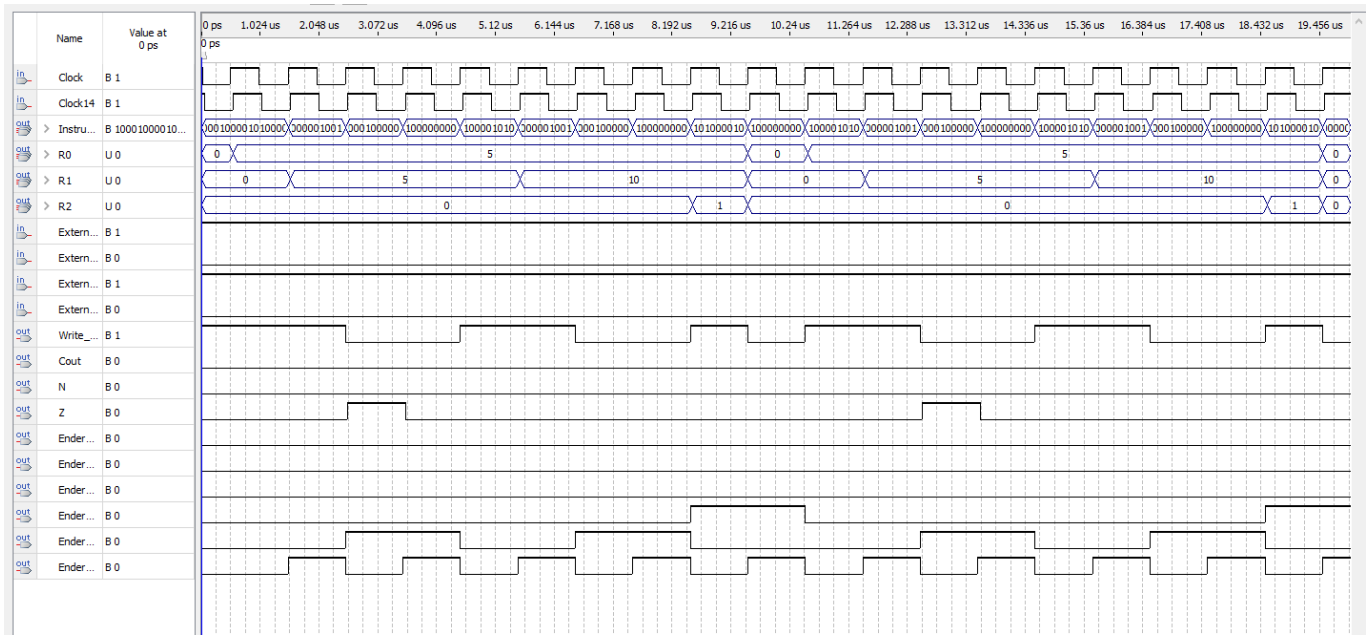

Figura 10: Resultado - Exemplo 1

Observemos a eficiência do processador na execução de todas as instruções.

- **Exemplo 2**
  No segundo exemplo, temos as seguintes instruções:

addi r0,2
storem r0,0x0
loadm r1,0x0
jmp 0x05
addi r1,10
movi r2, 5
rst

Na figura 11 vemos que o resultado foi satisfatório.

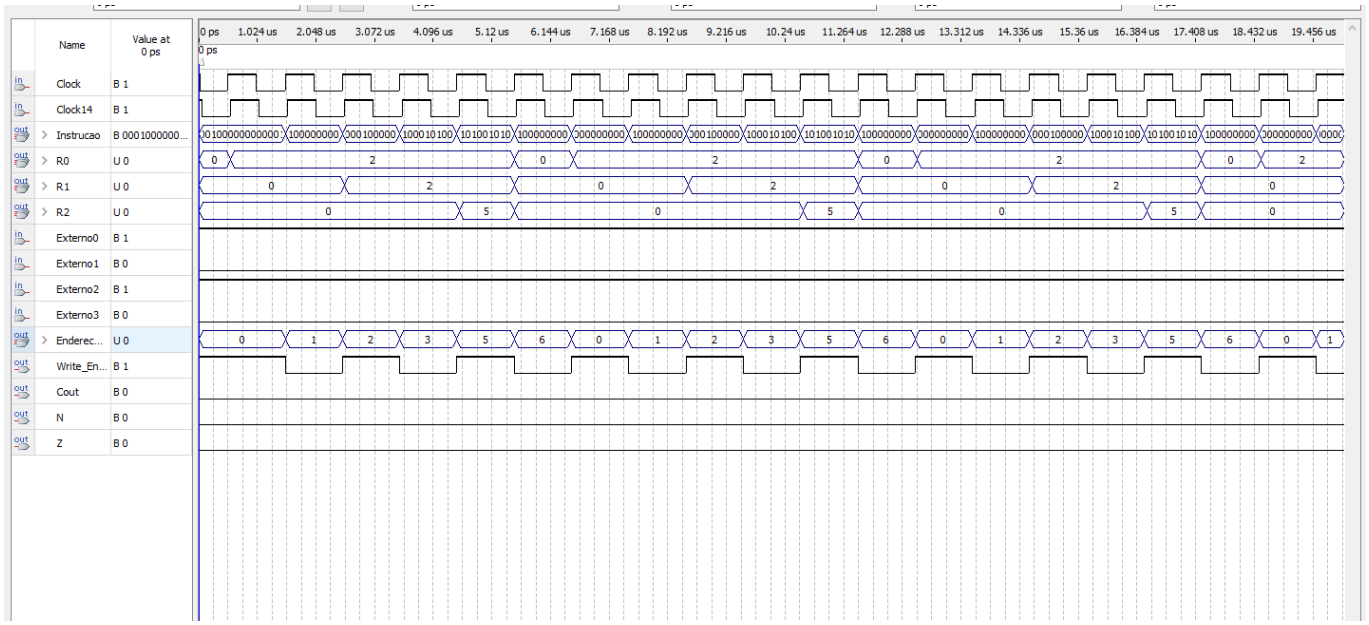

Figura 11: Resultado - Exemplo 2

- **Exemplo 3** Neste exemplo, mostramos a eficiência da função JMP. As instruções estão mostradas abaixo.

movi r2, 2

addi r1,r2,2

addi r2, 3

clr r2

jmp 0x8

storem r1,0x0

loadm r2, 0x0

addi r5,12

dec r2,r1

inc r1,r1

rst

A figura 12 mostra o resultado, ainda satisfatório.

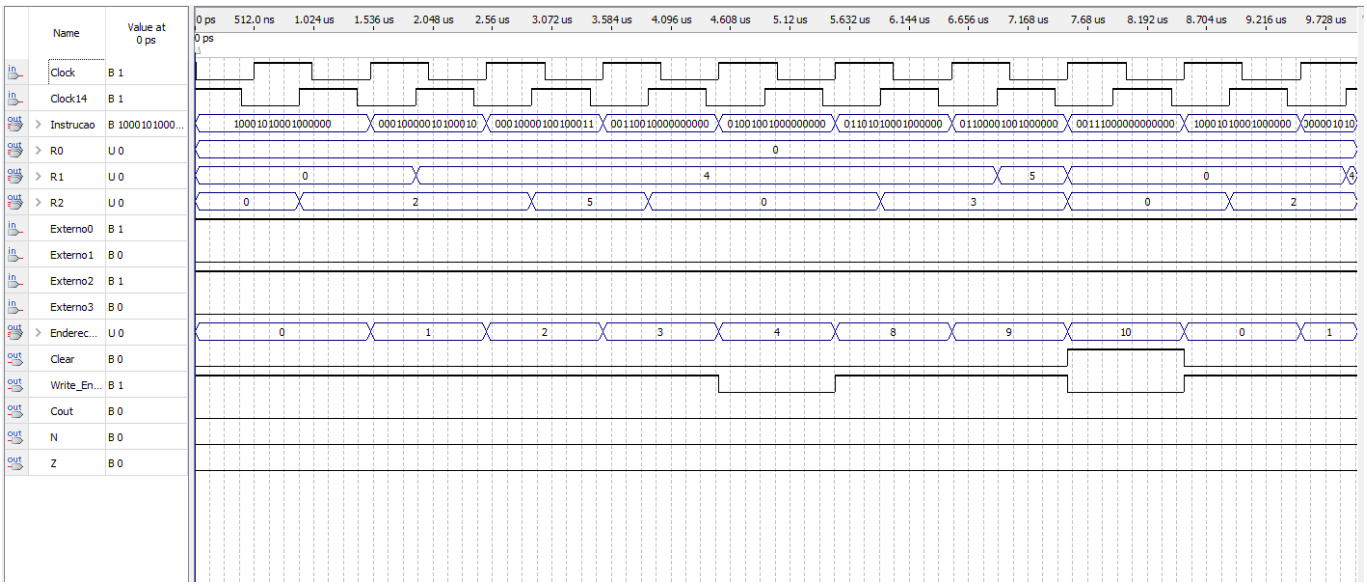

Figura 12: Resultado - Exemplo 3

- **Exemplo 4** No exemplo 4, mostrou-se a diferença entre as instruções CLR e RST. Coloca o valor 0 em um registrador sincronamente, enquanto a função RST zera todos os registradores assincronamente e volta para a primeira posição de memória. As instruções estão mostradas abaixo.

ADDi R0,1 ADDi R1,2

ADDi R2,3

CLR R0

CLR R1

CLR R2

ADDi R0,1

ADDi R1,2

ADDi R2,3

RST

O resultado obtido está mostrado na figura 13, também satisfatório.

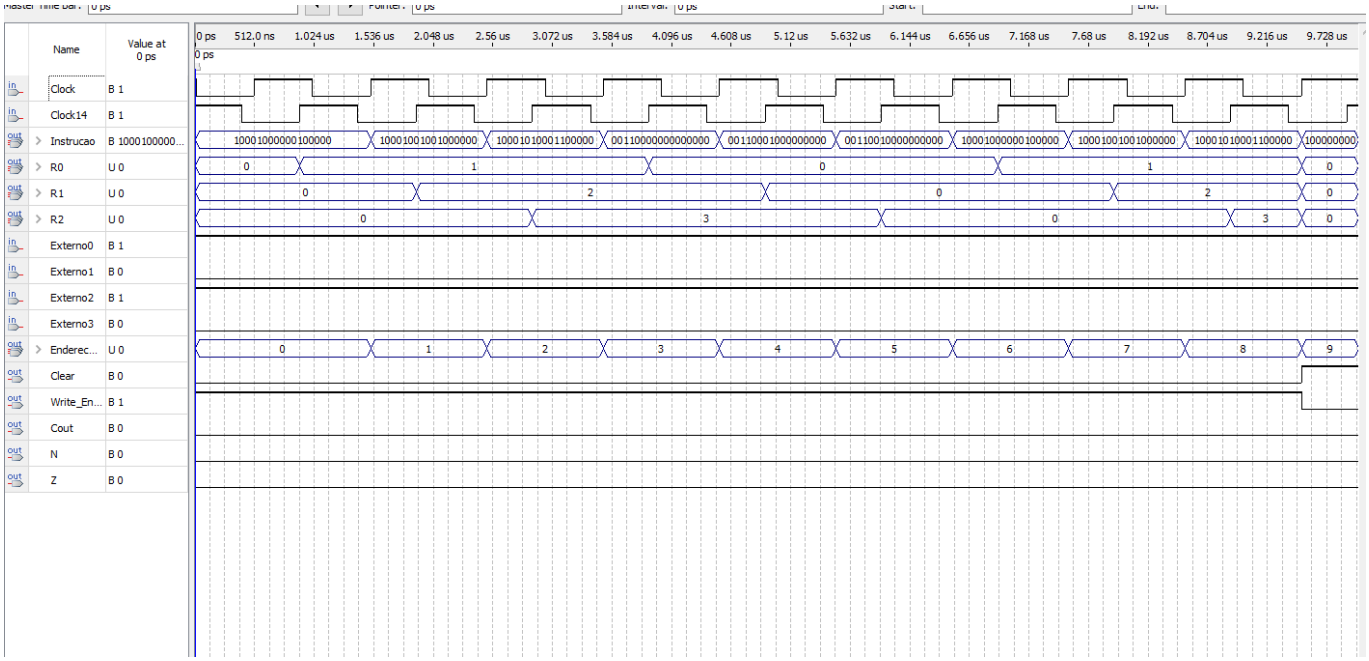

Figura 13: Resultado - Exemplo 4

3 Conclusão

Este trabalho abrangeu o desenvolvimento de um processador de ciclo único de 4 bits *(single-cycle 4 bits processor)* desde a definição de seu conjunto de instruções (ISA), percorrendo a construção de um *Mini-Assembler* para transcrição em linguagem de máquina do Assembly proposto e por fim criando os blocos que compõem o sistema descritos em *System Verylog*. O desafio motriz desse projeto era orquestrar todas as etapas de forma ter uma ingreção completa entre todos os componentes do sistema.

A ISA, o *Assembler* e o *hardware* implementados seguiram todas as especificações de projeto. Com o intuito de validar a construção do projeto, foram utilizados arquivos de testes, de forma que percorressem todo o *set* de instruções, sendo possível verificar todas as respectivas saídas em forma de onda, já que, devido a atual crise sanitária no país, não foi possível ver a verificação física em FPGA dos resultados obtidos durante o período de concepção do processador. Todas as etapas de simulação encontram-se descritas detalhadamente neste compêndio.

Como um todo esse projeto apresentou grande importância para aplicar a fundamentação obtida na primeira unidade, podendo percorrer todas as etapas teóricas apresentadas em aula. Foi possível implementar 100% do requerido para o trabalho e ainda adicionar certas *features* no projeto com o intuito de aumentar sua robustez, os resultados foram verificados, testadis e validados.

Apesar de se tratar de um trabalho inicial de projeto processador a sua importância não pode ser desmerecida, já que seu principal intuito é a validação didática do conhecimento adquirido. Ademais, esse projeto será base para futuras implementações mais avançadas dentro da disciplina, como por exemplo, a implementação de *pipeline*.