Relatório do experimento 6: um processador de propósito geral

Henrique Koji Miyamoto (RA 169614) e Pedro Luís Azevedo Costa (RA175857)

1 INTRODUÇÃO

O projeto final da disciplina de Laboratório de Circuitos Lógicos permite ao grupo escolher o projeto que deseja executar. Nós propusemos realizar a montagem de um processador de propósito geral de 4 bits. O conjunto de instruções ou a linguagem de programação escolhida para ser nele usada foi *brainfuck* (WIKIPEDIA, 2015a 2015b). Essas escolhas foram motivadas pelas possibilidades de aprendizado que os desafios impostos por esse projeto nos proporcionariam.

A relevância social desse projeto, isto é, sua importância para o público-alvo está diretamente relacionada à larga variedade de aplicações de um processador no cotidiano. Desde controladores de projetos de automação simples, passando por eletrodomésticos até computadores – tanto pessoais como servidores de grandes empresas – e *smartphones*, todos têm processadores em seu núcleo. Conforme veremos na seção 2, o processador é o elemento mais importante de um computador.

Nesse texto, apresentamos os resultados (nem sempre de sucesso) do nosso trabalho[1]: na seção 2, introduzimos as noções por trás da ideia de um processador; na seção 3, discorremos sobre o funcionamento da linguagem *brainfuck*; a seção 4 traz uma apresentação da arquitetura implementada; na seção 5, expomos nossos resultados; na seção 6, fazemos sua discussão e análise, apontando as origens de falhas e problemas identificados; por fim, deixamos nossas considerações finais sobre a experiência do projeto na seção 7.

O projeto foi desenvolvido e simulado no *Quartus II versão 13.0 Web Edition* e testado na placa FPGA (*Field-¬Programmable Gate Array*) da Altera, modelo *EP2C20F484C7*, da família *Cyclone II*.

2 O QUE É UM PROCESSADOR

Para iniciar a discussão do que é um processador, apresentaremos a descrição de Turing (1950) dos elementos de um computador digital, em seu clássico artigo que discute a possibilidade de máquinas inteligentes. De acordo com ele, um computador digital deve ter três partes, a saber: uma

[1] Até o presente momento, o projeto não pôde ser satisfatoriamente finalizado. Nem todas as dificuldades encontradas no caminho puderam ser resolvidas até agora, de modo que o que apresentamos torna-se, em verdade, uma tentativa de montagem de um processador de propósito geral e registramos nesse relatório nossos erros e acertos, o que pôde ou não ser implementado e as razões para tal. No início, o relatório é mais teórico e expõe como é o funcionamento de um processador de propósito geral e como deveria ser o funcionamento do nosso. A seguir, nossos sucessos e fracassos são narrados e explicados.

unidade de armazenamento (memória), uma unidade executiva (que realiza as operações) e uma unidade de controle.

Arroz et al. (2003) definem um computador como "um sistema digital programável através de uma sequência de instruções guardadas em memória". De fato, um processador é o principal componente de um computador. Ele pode ser definido como um circuito digital que recebe entradas, faz operações e registra (armazena) os resultados. Um processador pode ser de propósito único ou de propósito geral. Um de propósito único realiza uma única tarefa de processamento e tem sua funcionalidade implementada diretamente no *hardware* (como no experimento 5). Já em um processador de propósito geral (também chamado processador programável), o *hardware* é construído de forma tal que possa realizar diversas tarefas, segundo as instruções armazenadas em uma memória (VAHID, 2008).

Podemos retomar o artigo de Turing e perceber que a ideia por trás do seu computador digital é a de um processador de propósito geral, pois ele descreve que uma de suas propriedades mais importantes é que a capacidade "imitar" qualquer máquina de estados finitos. A descrição feita por ele é compatível com as descrições atuais de elementos de processadores, como a de Vahid (2008).

Vamos apresentar as operações básicas de um processador baseado na descrição de Arroz et al. e de nosso próprio processador:

- O endereço da próxima instrução a ser executada é obtido. A instrução é lida e decodificada;
- Os dados necessários para executar a instrução são carregados no banco de registradores;
- A operação relacionada à instrução é executada;
- Os seus resultados são enviados para a memória ou para outra saída de dados.

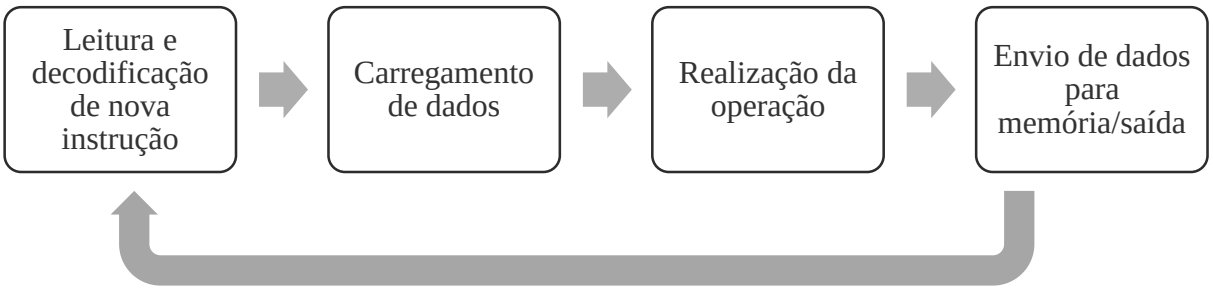


Figura 1. Diagrama simplificado da sequência de operações no processador

3 A LINGUAGEM *BRAINFUCK* E PROGRAMAÇÃO

Brainfuck é uma linguagem de programação esotérica, isto é, cujo objetivo é experimentar novas ideias, testar os limites dos programadores ou ser uma brincadeira, em vez de ter uso prático (ESOLANG, 2015). Foi criada em 1993 por Urban Müller e se caracteriza pelo extremo minimalismo. Com seus oito comandos, é uma linguagem Turing-completa, ou seja, ela pode realizar todas as funções que uma máquina de Turing realizaria (WIKIPEDIA, 2015a e 2015b). A

questão da completude Turing é analisada com mais detalhes no texto *BF is Turing-complete* (I WRITE, THEREFORE I AM, 2015).

A linguagem *brainfuck* conta com um ponteiro para as posições de memória de dados, iniciado na primeira posição, que pode ser avançado ou recuado, de forma a acessar todas as posições. Uma vez acessada uma posição de memória, pode-se incrementar ou decrementar seu valor, imprimi-lo na saída ou substituí-lo pelo valor da entrada. A linguagem conta ainda com um par de operações que permite realizar instruções em *loop*. A tabela a seguir lista as oito operações de *brainfuck*:

Tabela 1. Lista de instruções de *brainfuck* (adaptado de WIKIPEDIA, 2015b)

Símbolo	Operação
+	Incrementa em um o valor da célula de memória selecionada.
−	Decrementa em um o valor da célula de memória selecionada.
>	Incrementa o ponteiro (acessa a célula de memória seguinte).
<	Decrementa o ponteiro (acessa a célula de memória anterior).
.	Imprime na saída de dados o valor da célula de memória selecionada.
,	Salva na célula de memória selecionada o valor configurado na entrada de dados.
[	Estrutura de controle que repete os comandos, enquanto a célula selecionada for diferente de zero.
]	Fim da estrutura [.

Em nosso projeto, não foi possível implementar todos os comandos, de forma a tornar a linguagem Turing-completa. No entanto, pensamos a arquitetura de forma a facilitar uma implementação futura das operações faltantes. A tentativa atual de implementação foca nas quatro primeiras operações da lista, consideradas as mais relevantes para o projeto.

A codificação que adotamos para as instruções está expressa na tabela abaixo:

Tabela 2. Nossa codificação das instruções de *brainfuck*

Número (decimal)	Número (binário)	Símbolo da operação
0	000	+
1	001	−
2	010	<
3	011	>
4	100	.
5	101	,
6	110	[
7	111	]

Para programar nosso processador com essa linguagem, criamos um arquivo hexadecimal com 16 palavras de 3 bits (de 0_{10} a 7_{10}), que codificam as instruções, conforme a tabela acima. Esse

tamanho é o tamanho da memória RAM de instruções, onde fica salvo o arquivo, que é lido ao longo e executado ao longo do funcionamento do processador.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	0	1	2	3	4	5	6	7	.....
08	0	1	2	3	4	5	6	7	.....

```
+ - > < . , [ ] + - > < . , [ ]
```

Figura 2. Exemplo de código de programação para nosso processador em arquivo hexadecimal do Quartus (à esquerda) e o equivalente usando os símbolos apresentados na Tabela 1 (à direita). Esse código não implementa algoritmo algum e foi usado apenas para testar a saída de sinais do processador.

4 ARQUITETURA DE PROJETO E IMPLEMENTAÇÃO

A arquitetura do processador pode assumir diferentes formas. Ela depende, em primeiro lugar, do conjunto de operações que o processador deve realizar (no nosso caso, *brainfuck*). Existem, ainda, duas classificações de arquitetura clássica no que se refere ao projeto de processadores: em arquiteturas *von Neumann*, a mesma memória guarda tanto as instruções como os dados; já nas arquiteturas *Harvard*, utiliza-se uma memória para os dados e outra para as instruções (ARROZ et al., 2003). Nosso projeto utiliza a arquitetura Harvard.

O processador possui então duas memórias RAM, uma que armazena as instruções a serem executadas e uma que armazena valores (dados). Possui também dois registradores (Tavares, 2015) para armazenar o valor atual e o endereço atual de memória. A cada ciclo, as operações devem ser realizadas de acordo com as instruções na memória e os possíveis novos valores devem ser carregados nos registradores e salvos na memória de dados.

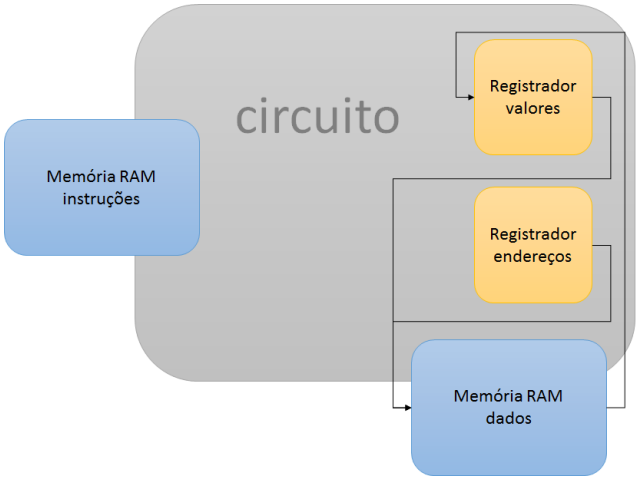


Figura 3. Digrama simplificado para evidenciar as relações entre os registradores e a memória RAM de dados

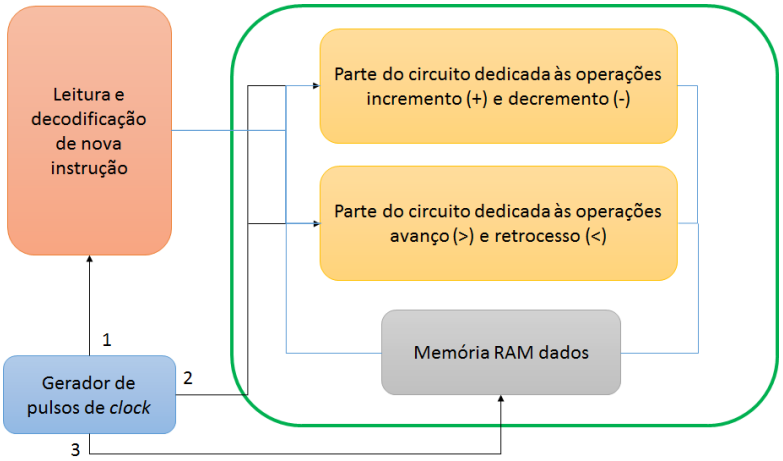
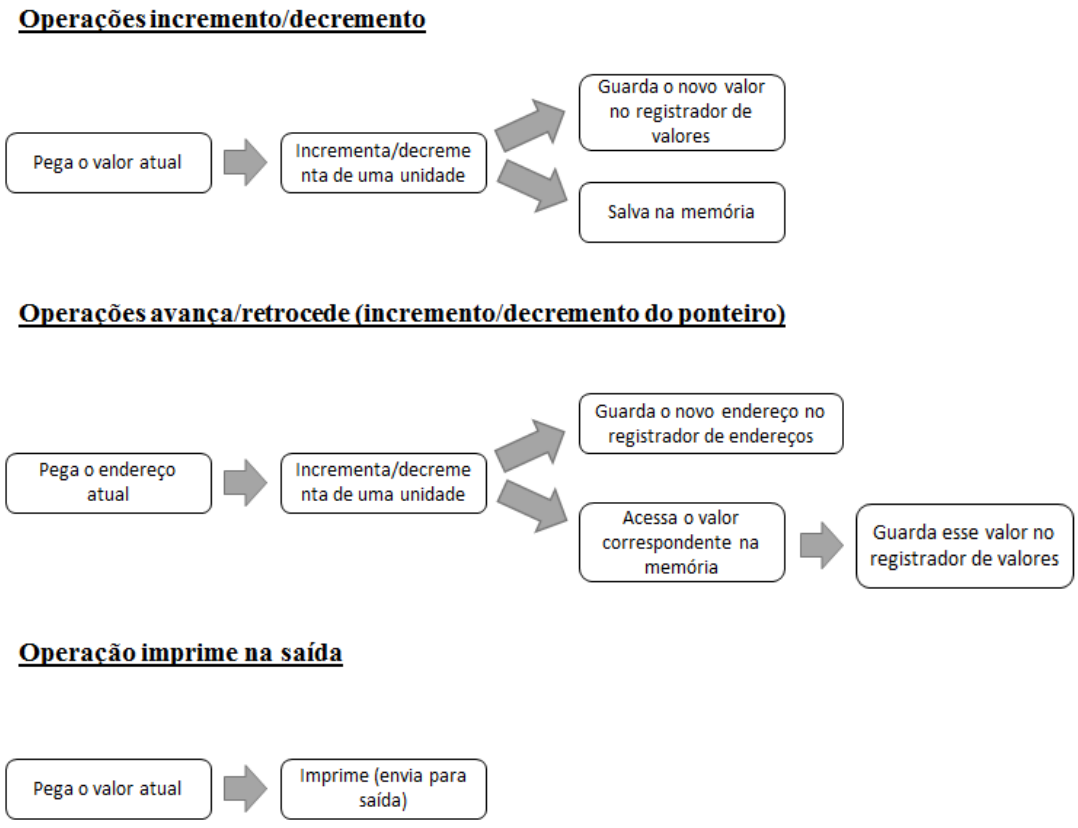


Figura 4. Diagrama simplificado dos módulos e partes destacadas do circiuto. Os registradores estão contidos nas partes em amarelo da figura.

4.1 IMPLEMENTAÇÃO DE OPERAÇÕES EM *BRAINFUCK*

Antes de prosseguir com a descrição da arquitetura adotada, parece apropriado olhar com mais detalhes cada uma das instruções disponíveis em *brainfuck*, e quais os passos para serem executadas, em nível de circuitos lógicos, no intuito de melhor entender sua implementação. Apesar de, nessa fase do projeto, estarmos tentando implementar apenas as primeiras quatro funções (Tabela 1), apresentamos, na figura a seguir os passos de execução das seis primeiras delas.



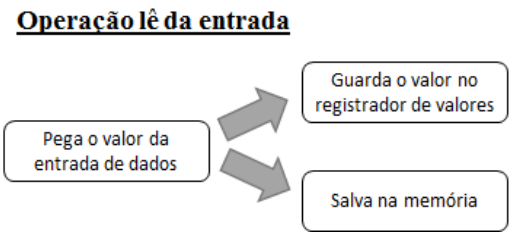


Figura 5. Esquema de etapas para execução das seis primeiras instruções de *brainfuck*

4.2 MÓDULOS DO CIRCUITO

No atual estágio de desenvolvimento, nosso processador conta com um módulo de leitura e decodificação de nova instrução, que é responsável por obter da memória a nova instrução; um módulo somador, que pode somar $+1_{10}$ ou -1_{10} tanto no valor quanto no endereço; um módulo gerador de pulsos de *clock*, que gera quatro pulsos distintos a cada período; e um módulo de controle de escrita na memória. Essas quatro partes serão melhor descritas a seguir.

4.2.1 Módulo gerador de pulsos de *clock*

Trata-se de um módulo que gera, a partir do sinal de *clock* para todo o circuito, quatro sinais de *clock* distintos. Isso é feito para melhor controlar a ordem das operações envolvidas no funcionamento do processador. O módulo é composto por um contador 7469 e um decodificador 74139 (FAIRHILD SEMICONDUCTOR, 2015). Ele é, na verdade, uma máquina de estados, da mesma forma que utilizado no experimento 5.

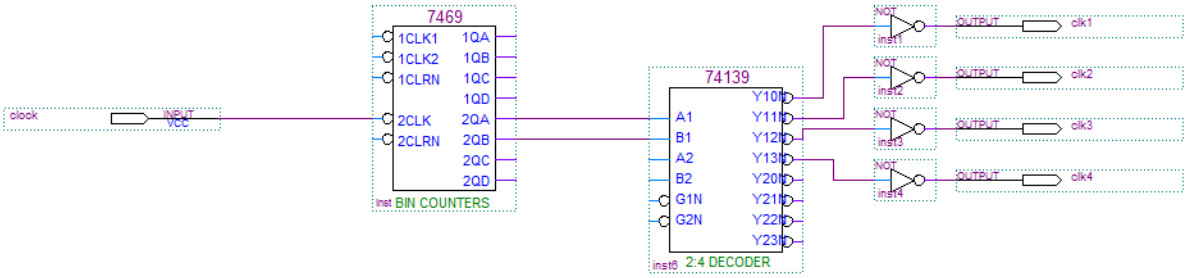


Figura 6. Circuito do módulo gerador de pulsos de *clock*

4.2.2 Módulo de leitura e decodificação de nova instrução

Esse módulo é constituído por um contador 7493 (NATIONAL SEMICONDUTOR, 2015b), uma memória RAM de 16 palavras de 3 bits e um decodificador projetado por nós. A memória RAM é criada no próprio Quartus II com *Megawizard Plug-In Manager*. A função desse módulo é obter a próxima instrução do programa e decodificá-la. O contador recebe o pulso de *clock* e, a cada ciclo, envia para a memória RAM um novo endereço de memória (de 0000_2 a 1111_2). A saída da RAM fornece a instrução guardada nesse endereço de forma codificada. O decodificador criado para esse fim específico decodifica a instrução e ativa somente a saída correspondente a ela, desativando as demais.

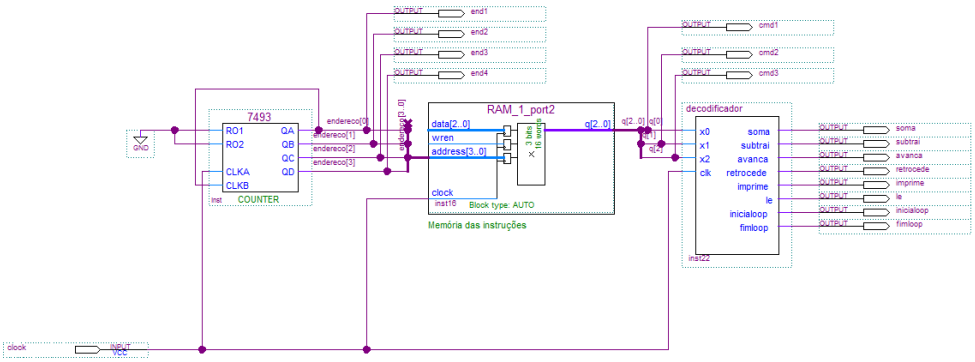


Figura 7. Circuito do módulo de leitura e decodificação de nova instrução

Nosso decodificador é construído a partir de um decodificador 74138 (MOTOROLA, 2015a), conforme imagem a seguir.

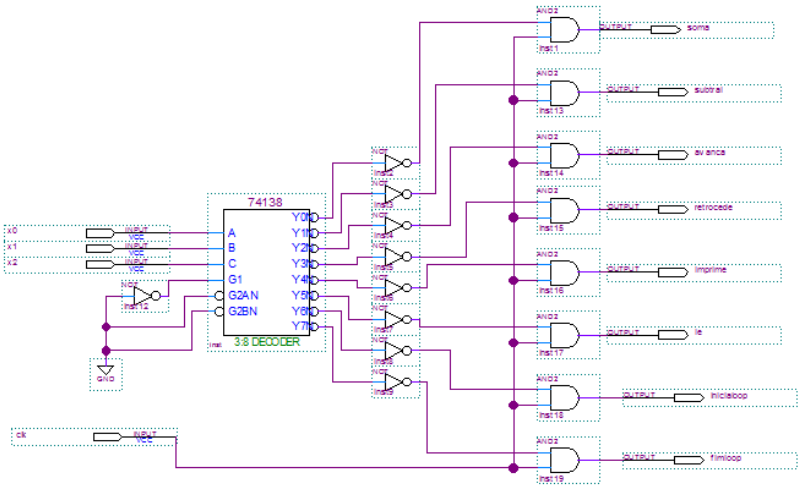


Figura 8. Circuito do nosso bloco decodificador

Tabela 3. Tabela-verdade do funcionamento do nosso registrador

x2	x1	x0	soma	subtrai	avanca	retrocede	imprime	le	inicialoop	fimloop
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

4.2.3 Módulo somador

Esse módulo é responsável por somar ou subtrair uma unidade (incrementar/decrementar) tanto no valor da posição de memória quanto no ponteiro que a acessa no momento (endereço). Recebe um barramento de 4 bits (`input[3..0]`), dois sinais de controle (`menos` e `mais`) e tem um barramento de saída (`output[3..0]`). Se o sinal `mais` for ativado, o resultado da saída é o valor da entrada incrementado de uma unidade; se o sinal `menos` o for, é o valor decrementado de uma uniuade. O esquema da unidade é mostrado a seguir.

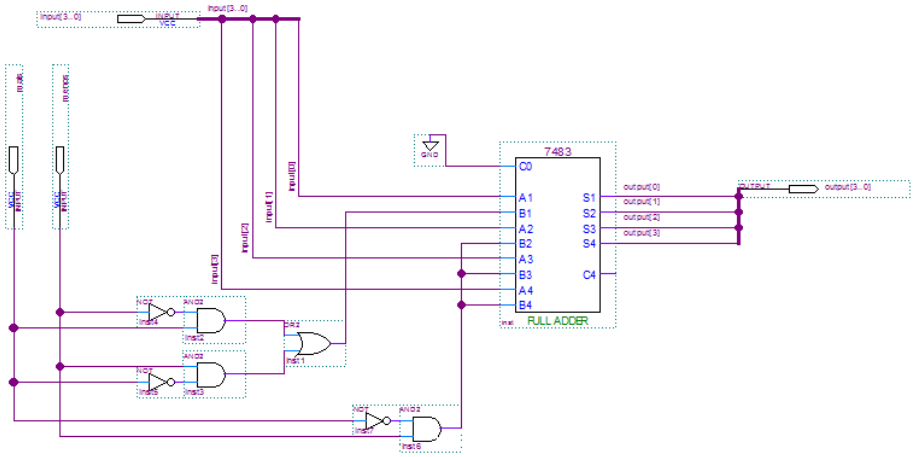


Figura 9. Circuito da unidade somadora

O módulo é montado a partir de um circuito somador total 7483 (MOTOROLA, 2015b) e de uma lógica combinacional que segue a tabela-verdade a seguir. O objetivo dessa lógica é que, quando o sinal que indica que um incremento deve ser feito (`mais`) for ativo, o número $1_{10}=0001_2$ seja configurado em uma das entradas dodecremento for feito (sinal `menos` ativo), o número $-1_{10}=1111_2$ será configurado em uma das entrada do somador; caso contrário, o número $0_{10}=0000_2$ será configurado.

Tabela 4 Tabela-veradade da lógica combinacional dentro da unidade somadora

x_1 (mais)	x_0 (menos)	z_3 (A4)	z_2 (A3)	z_1 (A2)	z_0 (A1)
0	0	0	0	0	0
0	1	1	1	1	1
1	0	0	0	0	1
1	1	0	0	0	0

A simplificação dessa tabela foi feita por soma de produtos, usando mapas de Karnaugh, através da ferramenta *online* disponível em Logic Circuit Simplification (2015). Os resultados são mostrados a seguir.

$$z_0 = \overline{x_1}x_0 + x_1\overline{x_0}$$

$$z_3 = z_2 = z_1 = \overline{x_1}x_0$$

4.2.4 Módulo de controle de escrita na memória

Esse é um módulo simples, formado por um flip-flop SR e uma lógica combinacional cujo objetivo é armazenar, a cada ciclo completo de *clock* se a instrução utilizada foi incremento ou decremento do valor. Isso porque essas são as duas operações (dentro do conjunto das quatro operações que ora são trabalhadas) que devem realizar escrita na memória. A importância de guardar essa informação reside no fato de que é apenas no terceiro *clock* do módulo gerador de pulsos de *clock* que a escrita na memória é realizada, mas a informação da instrução é fornecida no primeiro pulso, de forma que deve ser armazenada. A saída desse módulo é ligada à entrada `wren` (*write enable*) da memória RAM de dados. A seguir, o circuito do módulo, tabela-verdade do circuito combinacional e sua simplificação, realizada através de mapas de Karnaugh, com soma de produtos, pela ferramenta Logic Circuit Simplification (2015).

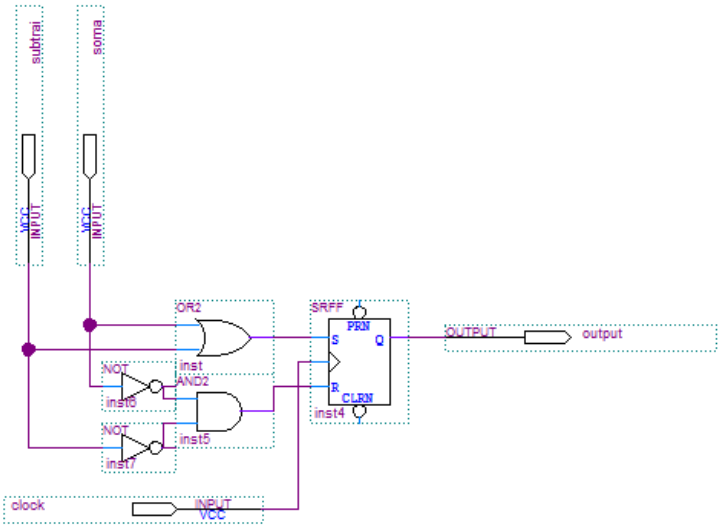


Figura 10. Circuito do módulo de controle de escrita na memória.

Tabela 5 Tabela-veradade da lógica combinacional dentro da unidade somadora

x_1 (soma)	x_0 (subtrai)	S	R	output
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	x	x	x

$$S = x_1 + x_0$$

$$R = \overline{x_1} \cdot \overline{x_0}$$

4.3 CIRCUITO COMPLETO

Após os textos das duas subseções anteriores, podemos introduzir e explicar o funcionamento do circuito como um todo.

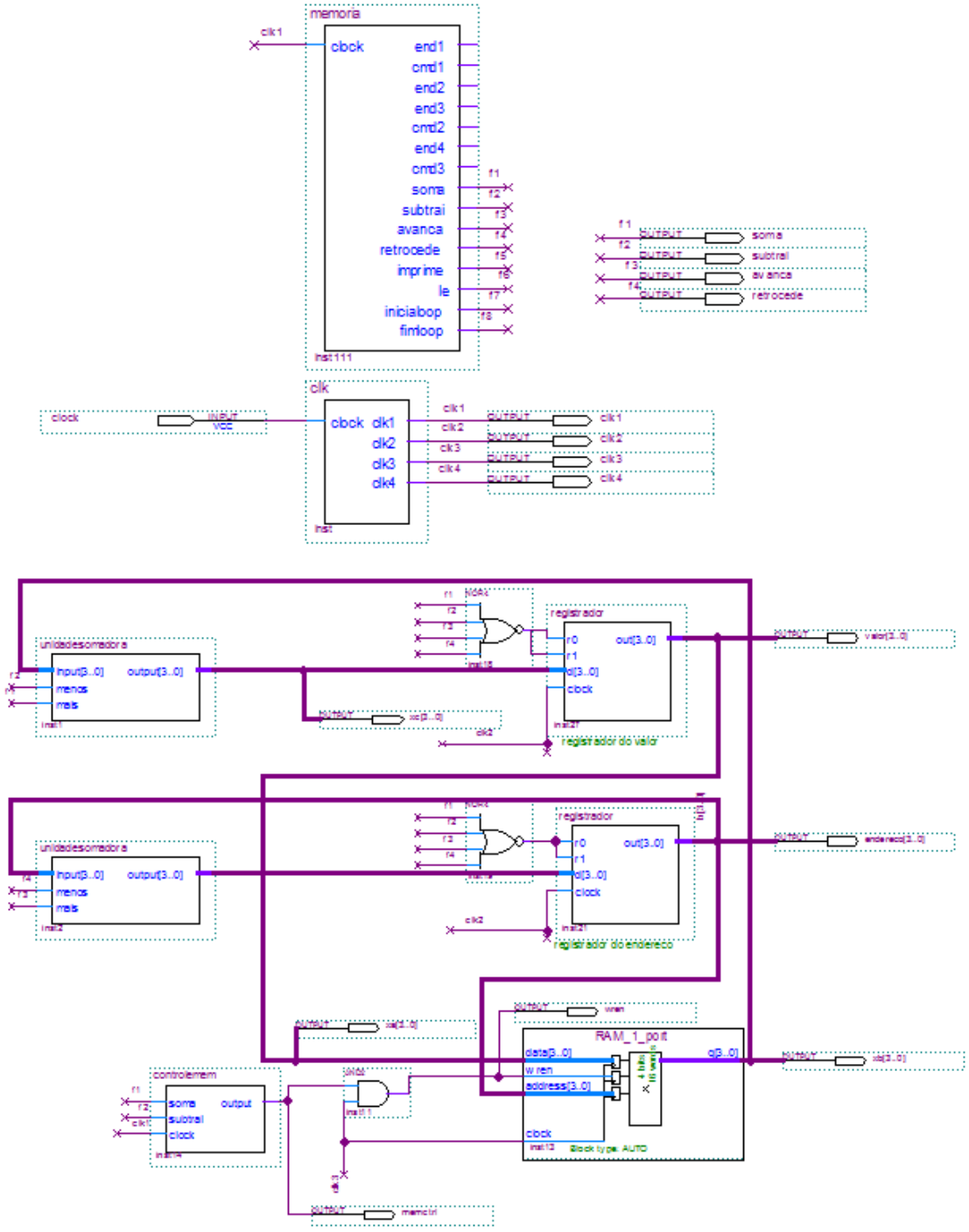


Figura 11. Circuito completo

No funcionamento do nosso circuito, inicialmente, o módulo gerador de pulsos de *clock* envia um pulso para o módulo de leitura e decodificação de instrução. Este irá ativar apenas a saída

correspondente à instrução a ser realizada. Se a instrução for incremento/decremento, o primeiro módulo somador será ativo, gerando em sua saída um novo valor, aumentado ou diminuído de uma unidade. Esse valor é carregado no registrador de valores no segundo pulso de *clock*. Daí, segue para a entrada da memória RAM de dados, onde deve ser salvo quando o terceiro pulso de *clock* for ativado. A saída da memória retorna para o módulo somador.

Caso a operação seja avança/retrocede (incrementa/decrementa no ponteiro), no segundo pulso, o novo endereço é carregado no registrador de endereços, ligado à memória, que deveria gerar o valor armazenado nesse endereço e levá-lo para o registrador de valores, passando pelo módulo somador (que somará 0_{10}, ou seja, nada se alterará no valor).

Esses passos deveriam se repetir em *loop*, durante cada ciclo de execução das 16 instruções gravadas na memória de comandos.

5 RESULTADOS

Apesar da lógica descrita na seção anterior, os resultados nem sempre condizem com o esperado. Conseguimos decodificar corretamente todas as instruções da memória de instruções e conseguimos realizar as operações incremento operações incremento/decremento e avança/retrocede o ponteiro, separadamente (esses serão os resultados corretos mostrados a seguir). No entanto, não conseguimos recuperar um valor já teoricamente salvo na memória. Isto é, ao retornar a uma posição de memória já acessada, em vez de recuperarmos o último valor que deixamos ali, a posição aparece zerada – esse resultado, embora incorreto, também é mostrado nessa seção.

5.1 DECODIFICAÇÃO DAS INSTRUÇÕES NA MEMÓRIA

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	0	1	2	3	4	5	6	7	.....
08	0	1	2	3	4	5	6	7	.....

Figura 12. Código usado para esse teste

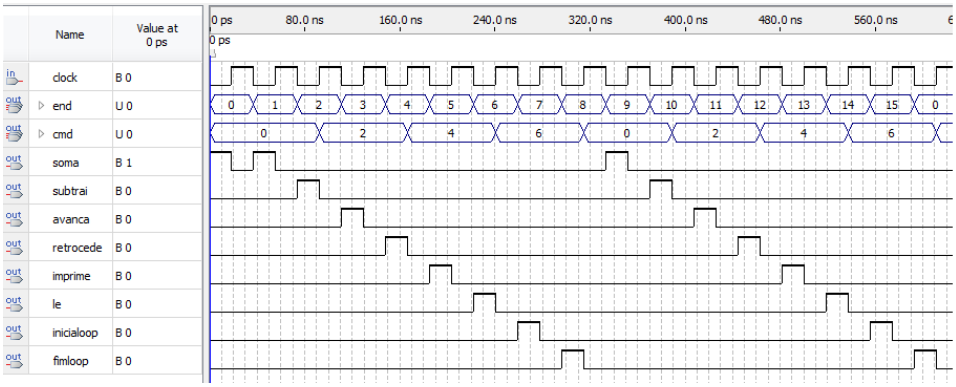


Figura 13. Resultado da simulação funcional

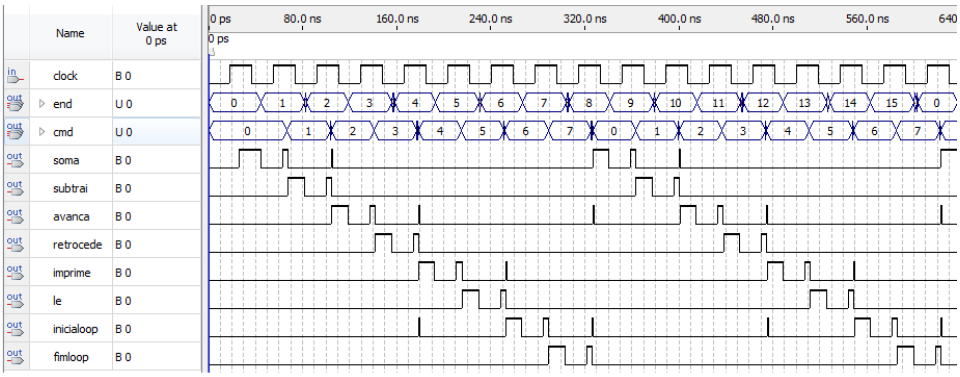


Figura 14. Resultado da simulação temporal

5.2 OPERAÇÕES INCREMENTO E DECREMENTO DO VALOR

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	00	00	00	00	00	00	00	00	.....
08	01	01	01	01	01	01	01	01	.....

Figura 15. Código usado para esse teste

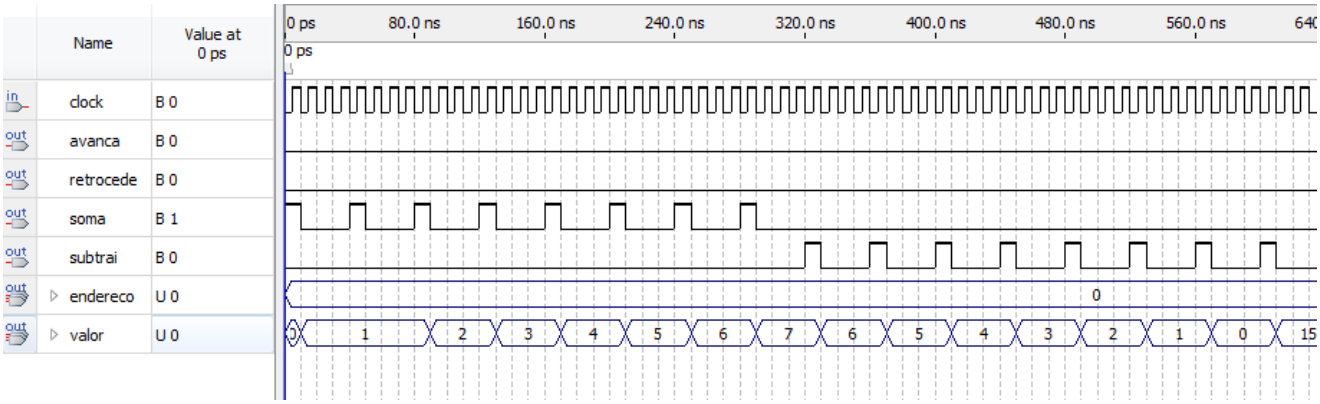


Figura 16. Resultado da simulação funcional

5.3 OPERAÇÕES AVANÇO E RETROCESSO (INCREMENTO E DECREMENTO DO PONTEIRO)

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	02	02	02	02	02	02	02	02	.....
08	03	03	03	03	03	03	03	03	.....

Figura 17. Código usado para esse teste

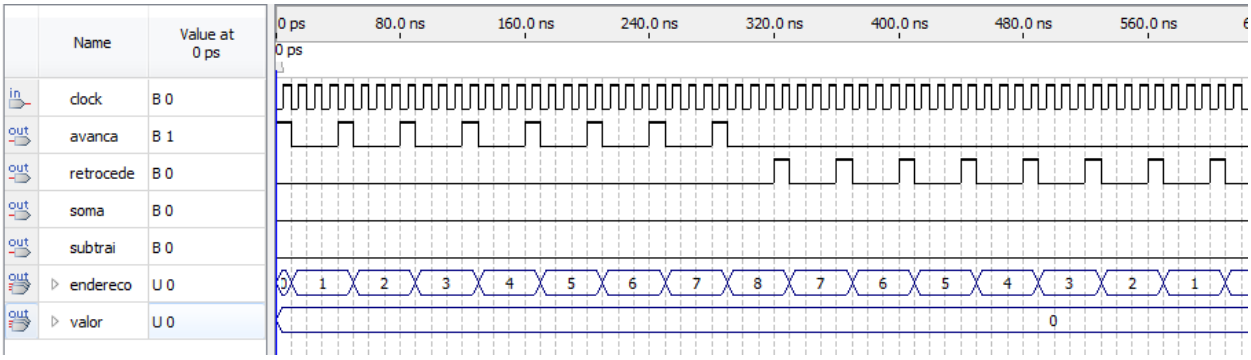


Figura 18. Resultado da simulação funcional

5.4 TENTATIVA DE RECUPERAR UM VALOR DA MEMÓRIA

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	00	00	00	00	00	02	01	01	.....
08	01	03	00	00	00	02	00	02	.....

Figura 19. Código usado para esse teste

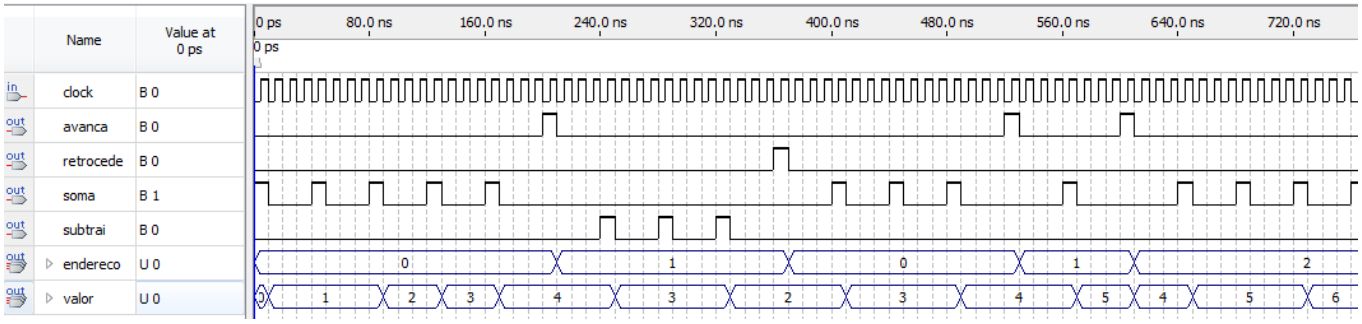


Figura 20. Resultado da simulação funcional

6 DISCUSSÃO E ANÁLISE

Nosso projeto sofreu muitas mudanças desde o começo de seu planejamento. A ideia que tínhamos no início sobre a implementação de um processador estava equivocada e foi, com o passar do tempo, amadurecendo para chegar à noção que temos hoje.

Tendo uma ideia melhor da implementação, incialmente, a ideia era implementar um conjunto de instruções Turing-completo, o que seria possível com as oito operações de *brainfuck*, mas não tivemos tempo hábil para tal. Primeiro deixamos de lado as últimas duas operações (*loop*) e depois as operações de impressão na saída e leitura da entrada.

Em uma versão anterior do esquemático no Quartus essas duas últimas funções chegaram a ser usadas; no entanto, esse diagrama de blocos não funcionou. Isso ocorreu porque muitos erros se sobrepuseram, de modo que tornou-se difícil identificar e saná-los um por um. Um motivo para

tantos erros estava no modo como realizamos o projeto: em vez de testarmos pequenas partes à medida que íamos montando o circuito, deixamos para fazê-lo apenas no final, o que gerou um acúmulo de problemas. Então, recomeçamos o diagrama no Quartus, montando o circuito por partes e realizando pequenos testes à medida que montávamos. Essa operação foi repetida cerca de duas vezes, quando chegávamos a um erro muito grande e difícil de resolver.

Em versões anteriores, os principais problemas estavam relacionados ao *clock* e ao controle de sinais. Valores não desejados apareciam entre os valores de dados e de endereços corretos, mesmo na simulação funcional. Quando ainda existia, o uso da operação "imprime" ocasionava algum problema não muito bem estudado que alterava os sinais de *clock* em relação ao que é esperado. Essas considerações motivaram a introdução do módulo gerador de pulsos de *clock* na versão atual.

Nessa versão, conseguimos fazer funcionar as funções com que estávamos trabalhando (as quatro primeiras). No entanto, nosso maior problema se tornou a recuperação de valores de memória já lidos, o que ainda não conseguimos fazer, como citado na seção 5. Ao retornar para uma posição de memória já acessada, em vez de recuperarmos o último valor que deixamos ali, a posição aparecia zerada. Após esforços para tentar resolver esse problema, a situação foi alterada e chegamos a pensar que estávamos na direção certa, pois, ao avançar e retroceder nas posições de memória, os valores não eram zerados. No entanto, percebemos que era como se estivéssemos acessando sempre a mesma posição de memória e alterando sempre o valor nela contido, conforme último resultado da seção 5.

Outro problema que ainda encontramos é que não conseguimos testar sequer os incrementos/decrementos nos valores e endereços na placa FPGA. O resultado do teste na placa coincide com os resultados da simulação temporal no Quartus: por motivos ainda desconhecidos, nas simulações temporais, os valores de dados e da memória permanecem em 0000_2 constantemente.

7 CONSIDERAÇÕES FINAIS

Apesar dos esforços empreendidos, o projeto não pôde, até o presente momento, ser satisfatoriamente concluído. Uma das funções mais importantes e interessantes de um processador, que é o uso da memória, ainda não foi totalmente implementado. No entanto, isso não exlcui o aprendizado que tivemos no decorrer do projeto: trata-se de um projeto extenso e de certa complexidade, o que fez com que aprendêssemos muito sobre o funcionamento, em nível de circiutos lógicos de um procesador, mesmo sem ter feito o nosso funcionar. Podemos dizer que pelo menos ficamos satisfeitos quanto ao nível de aprendizado que a constução de um processador nos propiciaria, conforme motivação apreesntada na seção 1. Mesmo com o fim da disciplina, o desafio do projeto e a possibilidade de ainda mais conhecimento fazem com que tenhamos vontade de ainda concluí-lo.

8 REFERÊNCIAS

ARROZ, Guilherme; MONTEIRO, José; OLIVEIRA, Arlindo. *Introdução aos Sistemas Digitais e Microprocessadores*. 23 maio 2003. Disponível em: < http://web.ist.utl.pt/pedro.m.s.oliveira/Introducao_aos_Sistemas_Digitais_e_Microprocessadores .pdf>. Acesso em: 01 dez. 2015.

EA-773 – Instruções para o projeto. Disponível em: <http://www.dca.fee.unicamp.br/~tavares/courses/2015s2/ea773-6.pdf>. Acesso em: 01 dez. 2015.

ESOLANG. *Esoteric programming language*. Disponível em: < https://esolangs.org/wiki/Esoteric_programming_language>. Acesso em: 01 dez 2015.

FAIRCHILD SEMICONDUTOR. *DM74LS138, DM74LS139 Decoders/Demultiplexers*. Disponível em: < http://ecee.colorado.edu/~mcclurel/dm74ls138.pdf>. Acesso em: 02 dez. 2015.

I WRITE, THEREFORE I AM. BF is Turing-complete. Disponível em: < http://www.iwriteiam.nl/Ha_bf_Turing.html>. Acesso em: 01 dez. 2015.

LOGIC CIRCUIT SIMPLIFICATION. Disponível em: <http://www.32x8.com/>. Acesso em: 02 dez. 2015.

MOTOROLA. *1-of-8 decoder/demultiplexer*. Disponível em: <http://ecee.colorado.edu/~mcclurel/sn74ls138rev5.pdf>. Acesso em: 02 dez. 2015a.

_____. *4-bit binary full adder with fast carry*. Disponível em: <http://pdf.datasheetcatalog.com/datasheet/motorola/SN74LS83D.pdf>. Acesso em: 02 dez. 2015b.

NATIONAL SEMICONDUTOR. *54153/DM54153/DM74153 Dual 4-Line to 1-Line Data Selectors/Multiplexers*. Disponível em: <>. Acesso em: 02 dez. 2015a.

_____. *DM5490/DM7490A, DM7493A Decade and Binary Counters*. Disponível em: <http://www.ece.usu.edu/ece_store/spec/7493-14p.pdf>. Acesso em: 02 dez. 2015b.

TAVARES, Tiago. *Um projeto de registrador: passo a passo*. Disponível em: <http://www.dca.fee.unicamp.br/~tavares/courses/2015s2/relatorio-exemplo.pdf>. Acesso em: 01 dez. 2015.

TURING, Alan. Computing Machinery and Intelligence. *Mind*, v. 59, n. 236, p. 433-460, out. 1950.

VAHID, Franklin. *Sistemas Digitais: projeto, otimização e HDLs*. Capítulo 8: Processadores Programáveis. Bookman: Porto Alegre, 2008.

WIKIPEDIA. *Brainfuck* (página em inglês). Disponível em: <https://en.wikipedia.org/wiki/Brainfuck>. Acesso em: 01 dez. 2015a.

_____. *brainfuck* (página em português). Disponível em: < https://pt.wikipedia.org/wiki/Brainfuck>. Acesso em: 01 dez. 2015b.