Universidade Federal de Itajubá
Campus Itabira

Relatório Projeto Eletrônica Digital II - ELT013

Implementação de um Processador em VHDL

Aluno: Matheus Venturyne Xavier Ferreira

Professor(a) Orientador(a): Rodrigo A. da Silva Braga

Itabira, 05 de Outubro de 2013.

1. Sumário

1. Introdução

Nesse projeto foi desenvolvido o código VHDL para um processador de 16 bits com sete instruções. Ele segue a arquitetura de Von Neumann onde dados e instruções são armazenados juntos. Em sua arquitetura ele é composto de sete registradores R[7..0], um acumulador A, um bloco de soma e subtração com um registrador para o resultado G, um contador de programa inicializado em zero PC ou R7, um registrador de instruções IR, um multiplexador para o barramento, uma central de controle e um buffer de saída para dados e endereço, ADDR e DOUT respectivamente. Vemos que PC e R7 são equivalentes, isso permite manipular o contador de programas diretamente. Na interface com o mundo exterior, tanto para a simulação quanto para testes físicos, o processador foi conectado a uma memória RAM e uma saída para os LEDs da placa Cyclone IV. A saída para os LEDs são acionadas para determinadas faixas de endereço disjuntos ao endereço da memória.

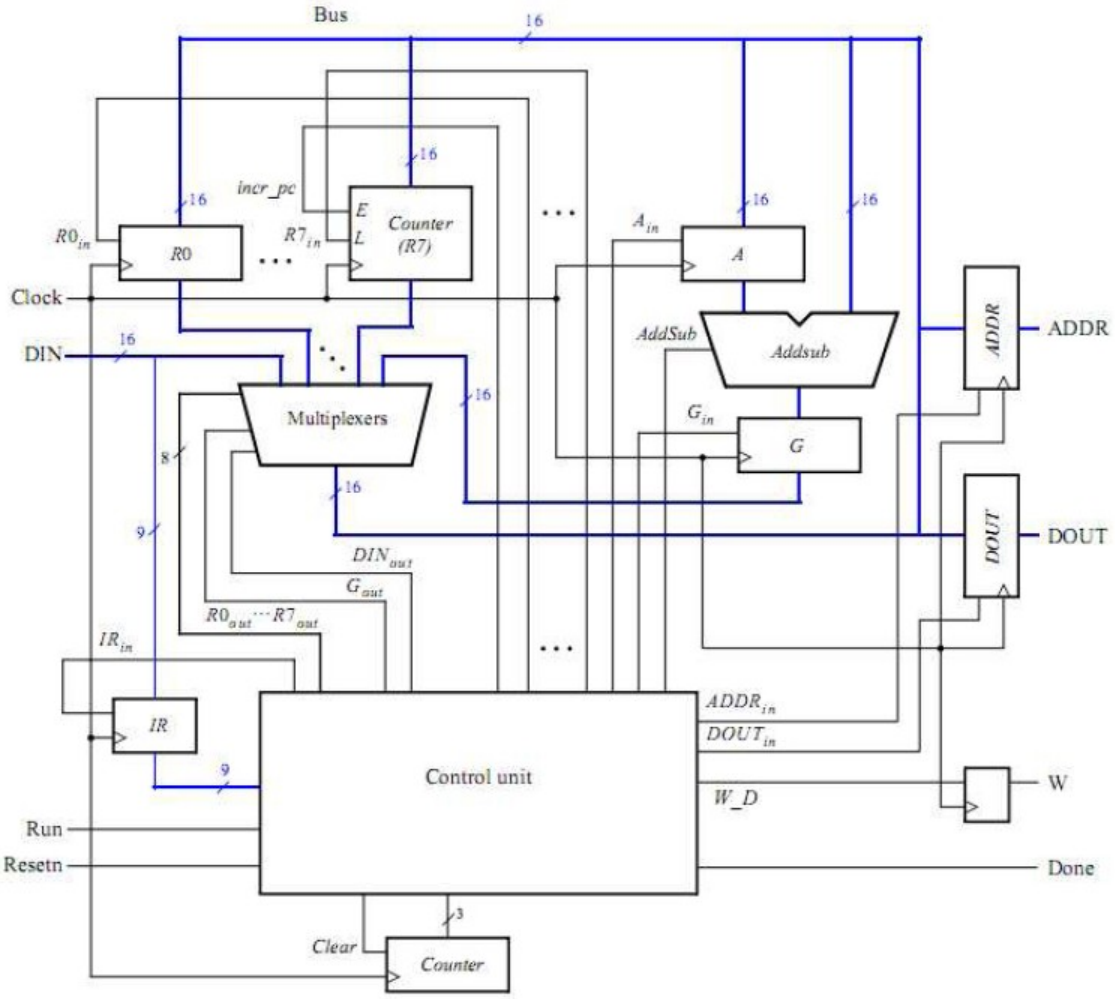

Figura 1: Diagrama esquemático do circuito do processador proposto a ser desenvolvido. O circuito implementado sofreu algumas modificações que acreditamos facilitar o projeto do sistema. As diferenças estão relacionados ao fato da Control Unit possuir clock de entrada (especificamente foi usada a borda de descida) e o Counter na parte inferior da figura está interno à unidade de controle (também controlado por borda de descida).

2. Conjunto de instruções

O conjunto de instruções no processador são indicadas na tabela 1. Cada instrução é composta de 9 dígitos binários, 3 bits para o upcode, 3 bits para o operando 1 e 3 para o operando 2. O processador trabalha com palavras de 16 bits e não faz distinção de dado por instrução, e por definição em palavras da memória que são instruções os bits mais significativos são usados como entrada para o registrador de instrução instrução.

Comando	Upcode	Ciclos	Descrição
Move Rx, Ry	000	1	Rx ← [Ry]
LoadImmediate Rx, #D	001	4	Rx ← #D
Addition Rx, Ry	010	3	Rx ← [Rx] + [Ry]
Subtraction Rx, Ry	011	3	Rx ← [Rx] – [Ry]
Load Rx, [Ry]	100	4	Rx ← [[Ry]]
Storage Rx, [Ry]	101	3	[Ry] ← [Rx]
Movenz Rx, Ry	110	1 ou 2	If Z != 0 : Rx ← [Ry]

Tabela 1: Conjunto de Instruções do processador. Cada instrução possui um upcode que o representa e a quantidade de ciclos gastos para sua execução. Na tabela o operador [ ] representa desreferenciação, por exemplo, na operação Load Rx, [Ry] o registrador Rx recebe o valor de Ry desreferenciado duas vezes. A primeira desreferenciação de Ry, [Ry], retorna o valor armazenado no registrador y. A segunda aplicação da desreferenciação indica que o valor retornado por [Ry] é um endereço, logo [[Ry]] representa o valor armazenado no endereço de valor [Ry].

As instruções fazem uso direto dos registradores 0 até 7 como operando de suas operações. Para isso cada registrador tem um endereço como pode ser visto na tabela abaixo:

Registrador	Endereço
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7 ou PC	111

Tabela 2: Endereço dos registradores. Vemos que o registrador R7 é equivalente ao contador de programa PC.

Na tabela 1 vemos que o ciclo de execução de cada instrução pode varia. A operação Movenz por exemplo, dependendo da flag Z[1] pode chamar a operação Move adicionando 1 ciclo à sua execução. Além disso o processador desenvolvido está sujeito às limitações da velocidade de memória. A memória utiliza três ciclos de clock para apresentar um dado na entrada. Um ciclo

1 A flag Z é uma flag utilizada para indicar se o resultado de uma operação é nula. Se Z armazena 0 indica que o resultado não é nulo. Se o resultado é nulo estará armazenado o valor 1.

para receber o endereço, um ciclo para garantir a busca do dado e um ciclo para apresentar o dado na saída. Devido à perda de três ciclo toda operação que envolve um acesso à memória tem um acréscimo de 3 ciclos de processamento. Como iremos ver isso afeta diretamente na implementação da máquina de estados. Além disso veremos que nossa operação de buscar instrução, da requisição da instrução a partir do PC até sua decodificação, tem um custo de 5 ciclos de clock e portanto é a operação mais custosa no processador desenvolvido.

O processador faz uso tanto da borda de decida quanto de subida de clock. A borda de descida é responsável somente pela mudança de estados e e a borda de subida é utilizada para atualizar os registradores e no controle da memória. Dessa forma a velocidade do barramento é aproveitada ao máximo ao passo que garante que os estados não sofram transições no momento de leitura e escrita dos dados.

3. Máquina de Estados

A máquina de estados do processador é composta de nove estados (Figura 1). Há um estado para cada uma das sete instruções, um estado de buscar instrução – fetch instruction e um estado quando reset ativo ou run não ativo, chamado idle. Cada estado possui uma sub máquina de estados cujos estados internos representam a quantidade de ciclos de clock que já foram executados pela instrução; um exemplo é a máquina de estados para o estado fetch instruction na Figura 2. Usando essa lógica separamos o contador de ciclos para cada instrução tornando mais simples a descrição do hardware. Acreditamos que com essa abordagem tornamos o código mais simples e de fácil modificação para acrescentar novas funcionalidades.

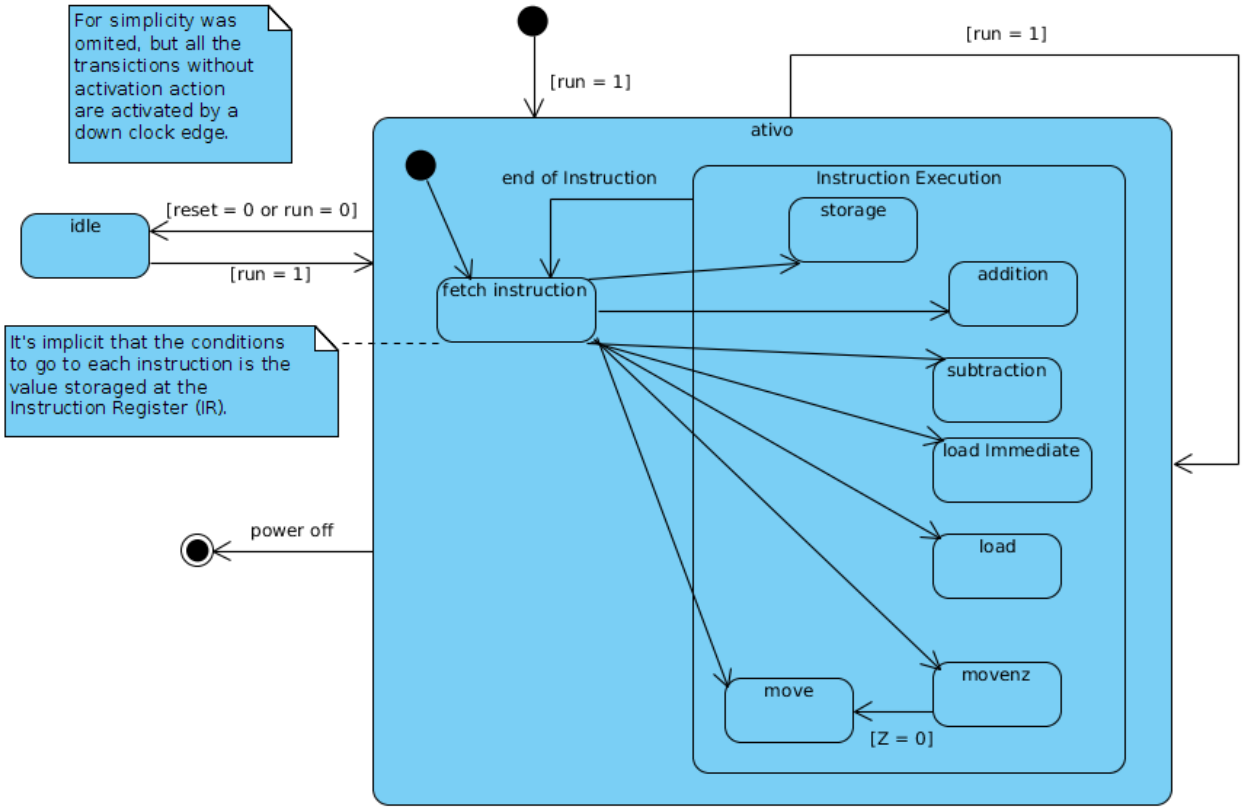

Figura 2: Diagrama UML de Máquina de estado do processador. Fonte: Autoria própria. Criado no Visual Paradigma.

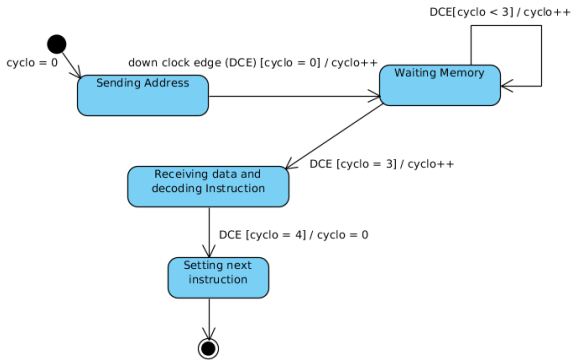

Figura 3: Sub diagrama de máquina de estados para o estado fetch instruction. Todas as transições são ativas por uma ação de borda de descida de clock (DCE). Fonte: Autoria própria.

4. Análise dos resultados

Para o teste do circuito e realização das simulações o processador foi conectado a uma memória RAM como pode ser visto no diagrama abaixo.

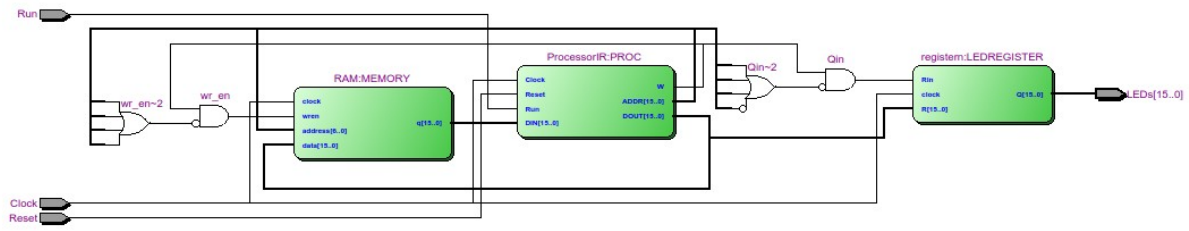

Figura 4: Diagrama da conexão do processador com uma memória RAM de 128 bits e palavras de 16 bits.

Para os testes foi gerado o programa assembly abaixo. Este programa inicializar o registrador 2 com 1. Em seguida inicializa o registrador com o valor binário `#10000000`. Em seguida o conteúdo de R7 (R7 é equivalente ao contador de programa) é copiado para o registrador R5. Nesse passo é importante notar que foi feita uma cópia do estado atual do contador de programa. Em seguida o conteúdo do registrador R4 é subtraído pelo conteúdo do registrador R2 e o resultado é armazenado no registrador R4. Na linha 5 se o resultado da subtração for nulo o conteúdo do contador de programar (R7) recebe o conteúdo do registrador R5. Ou seja, ele é carregado com seu estado anteriormente salvo. Nesse passo então ocorre um loop, o programa só passa para a instrução da linha 6 se a cópia na linha 5 não ocorrer. Ou seja, R4 é subtraído por R2 até que seu resultado seja nulo. As próximas instruções testam as outras funcionalidades do processado. Na linha 6 o contador R0 é carregado com #20 (decimal). Na linha 7, R1 é carregado com o valor na memória referente ao endereço em R0. Na linha 8 o conteúdo de R1 é armazenado na memória no endereço armazenado em R0. Na linha 9 o registrador R3 é carregado com o valor armazenado na memória na posição armazenada em R0.

```
mvi R2, #1
mvi R4, #10000000
mv R5, R7
sub R4, R2
mvnz R7, R5
mvi R0, #20
ld R1, [R0]
st R1, R0
ld R3, [R0]
```

Tabela 3: Código do programa de teste criado.

Este programa foi armazenado na memória RAM utilizando um arquivo .mif (memory initialization file). Ao sintetizar o circuito esse arquivo é usado para dar um pré set na memória. O conteúdo do arquivo pode ser visto na tabela 4.

```
DEPTH = 128;                   -- The size of data in bits
WIDTH = 16;                    -- The size of memory in words
ADDRESS_RADIX = DEC;           -- The radix for address values
DATA_RADIX = HEX;              -- The radix for data values
CONTENT                        -- start of (address : data pairs)
BEGIN

00 : 2800;                -- memory address : data
01 : 0001;
02 : 3000;
03 : 0008;
04 : 1780;
05 : 7100;
06 : DE80;
07 : 2000;
08 : 0014;
09 : 8400;
10 : A400;
11 : 8C00;
20 : 000F;

end;
```

Tabela 4: Arquivo .mif utilizado para inicializar a memória.

Com o circuito sintetizado e a memória carregado foi gerado o teste abaixo. Nesse teste é dado um reset no processador. No instante 80ns o sinal de run é enviado e a execução do processador inicial. Tanto o processador quando a memória são alimentados com um clock de 100MHz.

```
restart

force -deposit /reset 0 1, 1 5ns
force -deposit /run 0 0, 1 80ns
force -deposit /clock 0 0, 1 {5ns} -r 10ns
run 2000ns
```

Tabela 5: Arquivo teste.do utilizado para executar a simulação no modelSim.

O resultado da simulação pode ser visto na figura 5.

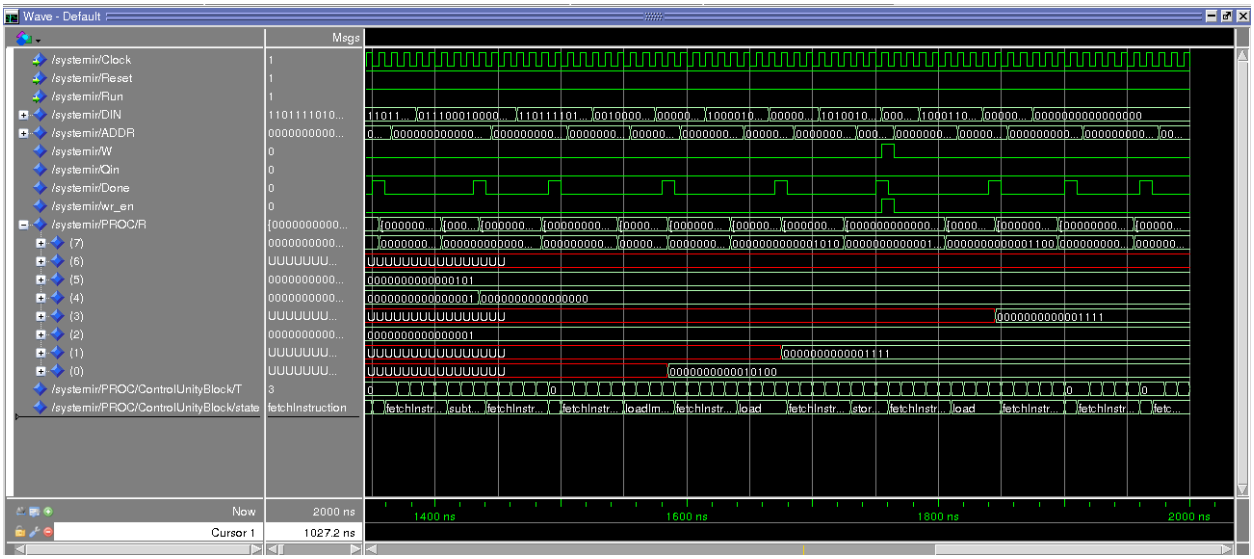

Figura 5: Simulação da execução do programa.

5. Dificuldades Encontradas

As dificuldades encontradas vem principalmente relacionadas à inexperiência para desenvolver um sistema mais complexo na linguagem VHDL além da necessidade de ser aprender técnicas eficientes de teste do circuito para facilitar a detectar as falhas que são inevitáveis durante a programação. Inicialmente teve-se o empasse de pensar numa organização do sistema de forma que a mudança de estado não afetasse a leitura de dados, uma vez que nesse instante o sistema se comporta de forma imprevisível. A solução encontrada foi simples, utilizar a borda de subida do clock para atualizar os registradores e a borda de descida na mudança de estados.

Outro ponto é a necessidade de buscar conhecimento em diferentes fontes de pesquisa. Este ponto no entanto é facilitado pela facilidade de se encontra boas referências na literatura e documentações amplas sobre VHDL.

6. Fontes de consulta

D'AMORE, Roberto. VHDL – Descrição e Síntese de Circuitos Digitais. 2º Edição. Editora LTC, 2012.

LPM Quick Reference Guide. Altera, 1996.