UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE TECNOLOGIA E CIÊNCIA

Adriano Oliveira Pires

PROJETO E IMPLEMENTAÇÃO EM VHDL PARA FPGAS DE ARQUITETURA COM CONJUNTO SUPER-REDUZIDO DE INSTRUÇÕES: PROCESSADOR PIRES

Florianópolis

2013

Adriano Oliveira Pires

PROJETO E IMPLEMENTAÇÃO EM VHDL PARA FPGAS DE ARQUITETURA COM CONJUNTO SUPER-REDUZIDO DE INSTRUÇÕES: PROCESSADOR PIRES

Trabalho de conclusão apresentado à disciplina EEL4100 38 - Sistemas Digitais e Dispositivos Lógicos Reconfiguráveis, para fim de avaliação parcial.
Orientador: Prof. Eduardo Augusto Bezerra
Coorientador: Prof. Djones Vinicius Lettnin

Florianópolis

2013

1 ESPECIFICAÇÕES

O trabalho prevê o projeto e implementação, através de código de descrição de hardware (VHDL), de um microprocessador com arquitetura multiciclo e um número reduzido de instruções, de acordo com a tabela da figura 1

Tipo	Código de máquina	Instrução	Operação	Descrição
I	00000001 end	LDA end	A <= memória[end]	Registrador A recebe o conteúdo da posição de memória *end*
I	00000010 end	STA end	memória[end] <= A	Posição de memória *end* recebe o conteúdo do registrador A
I	00000011 end	BLT end	PC <= (A < B ? end : PC + 2)	A execução do programa é desviada para o endereço *end*, se A for menor do que B
I	00000100 end	BEQ end	PC <= (A == B ? end : PC + 2)	A execução do programa é desviada para o endereço *end*, se A for igual a B
J	00000101 end	JMP end	PC <= end	A execução do programa é desviada incondicionalmente para o endereço *end*
R	00010000 end	ADD	A <= A + B	Registrador A recebe a soma de A com B (complemento de 2)
R	00100000 end	MUL	AB <= A * B	Multiplicação de A com B, sendo que o registrador A recebe a parte baixa do resultado, e o registrador B a parte alta
R	00110000 end	AND	A <= A AND B	Registrador A recebe o resultado da operação lógica AND de A com B
R	01000000 end	OR	A <= A OR B	Registrador A recebe o resultado da operação lógica OR de A com B
R	01010000 end	NOT	A <= NOT A	Registrador A recebe o conteúdo de A negado
R	01100000 const	LI const	A <= constante	LSB do registrador A recebe o valor *constante* contido na instrução, e MSB do A recebe 0 (zero)
R	01110000 end	SWP	A <= B, B <= A	O conteúdo do registrador A é copiado para o registrador B, e vice-versa (*swap A, B*)
	11111111 end	HALT	Halt	Suspende a execução do processador

Figura 1: Tabela com a especificação das instruções que devem rodar no microprocessador

O processador projetado deve executar um programa exemplo descrito em linguagem Assembly.
O processador implementado foi apelidado de *Pires*.

2 ARQUITETURA PROJETADA

Para fins desta implementação, os conceitos utiliazados foram definidos a partir do diagrama da figura 2.

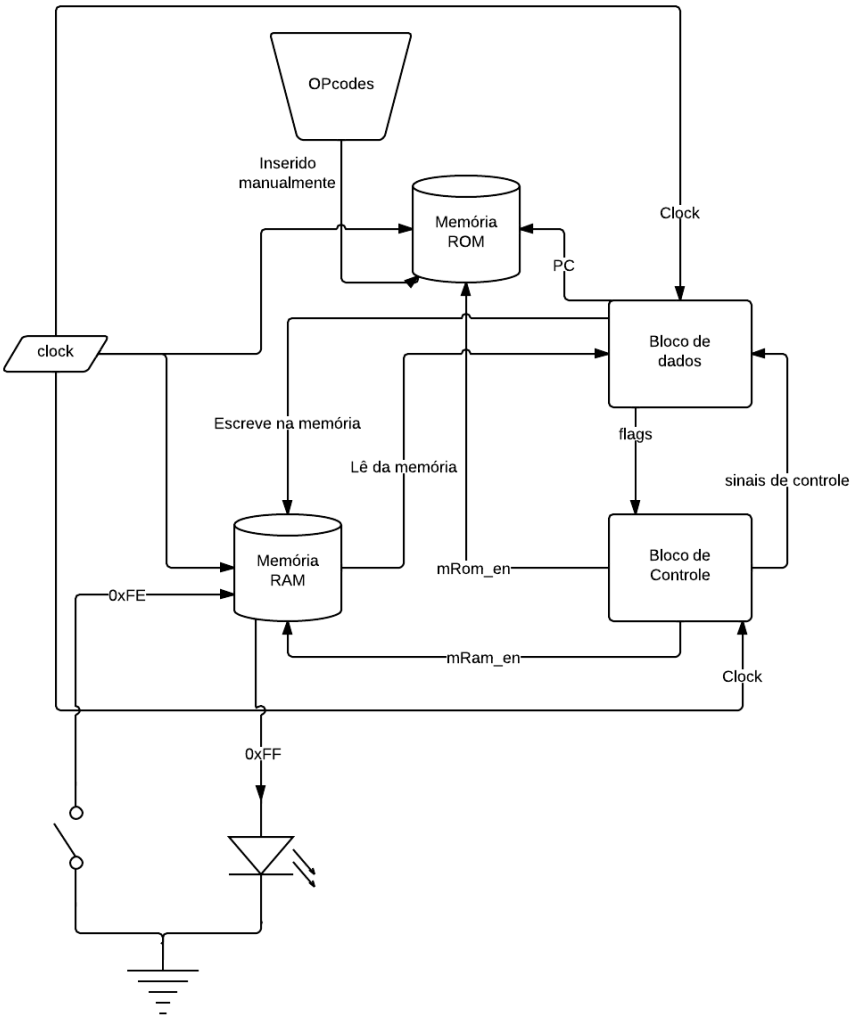

Figura 2: Diagrama geral do processador Pires

Para a arquitetura projetada um clock global sincronizará todo o hardware, composto dos seguintes elementos:

- Uma memória ROM, que irá guardar OPcodes (Já convertidos de assembly para binário) que são inseridos manualmente na ROM através de um método descrito na sessão 3.1.3;
- Uma memória RAM:
  - Com endereçamento para até 256 palavras de 8 bits;
  - Capaz de ler e escrever de registradores especiais do Bloco de dados;
  - Com controle de gravação realizado pelo Bloco de Controle;
  - Com mapeamento de memória possibilitando conectar diretamente *Inputs* e *Outputs* do kit de desenvolvimento;
- Um Bloco de Dados que armazena os dados temporários do processador (através dos registradores IR, PC, A e B) e que é responsável pelo processamento lógico aritmético;

- Um Bloco de Controle que, dependendo da instrução, dos Flags da ULA e de uma máquina de estados, fornece os sinais que coordenam o funcionamento dos demais hardwares já descritos;

2.1 BLOCO DE DADOS

A arquitetura projetada para o bloco de dados foi parcialmente feita baseada no microprocessador Cleópatra, apresentado em aula. Na figura 3 segue o diagrama projetado.

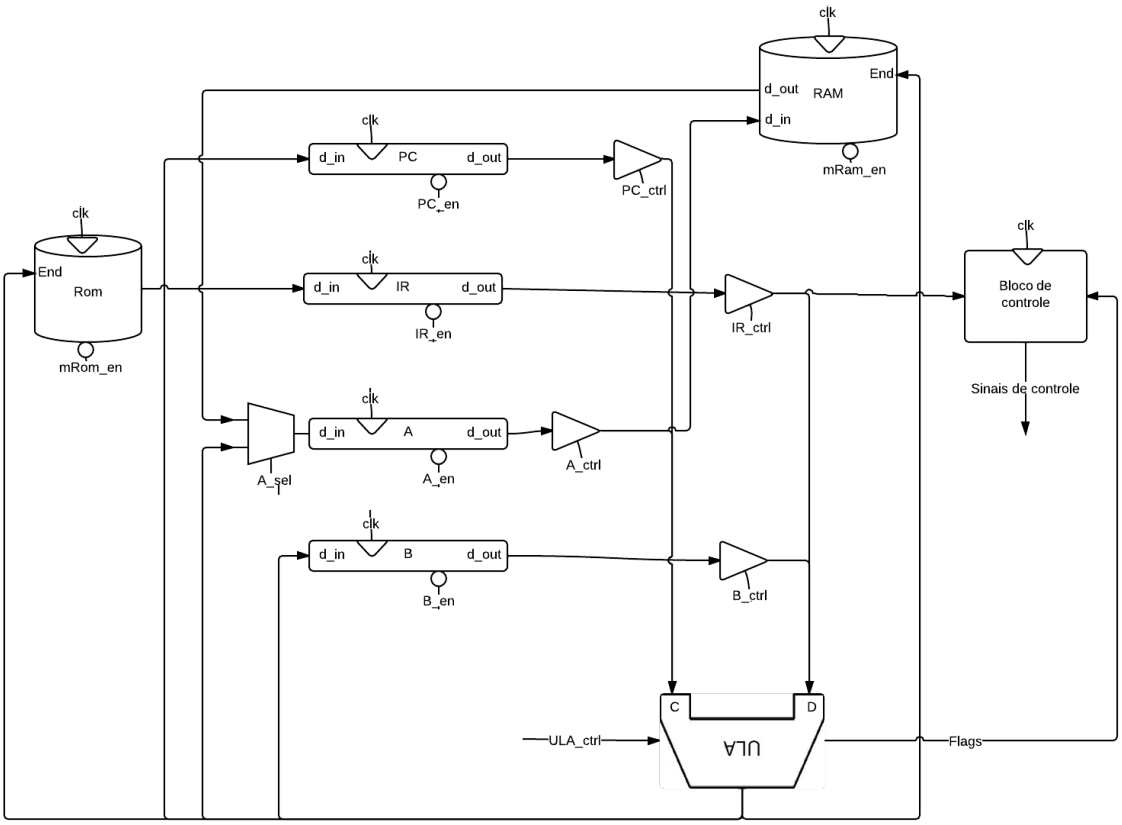

Figura 3: Arquitetura originalmente projetada para o Bloco de Dados.

O Bloco de Dados foi projetado em separado da estrutura de controle, já pensando na facilitação da montagem e modularização dos *TestBenchs*. Sua arquitetura prevê:

- Uma **ULA - Unidade Lógica Aritmética** com as operações que serão descritas na sessão 3.1.1, saída em Latch, um registrador interno de 16 bits e sinalização de flags para o Bloco de Controle;
- 4 Registradores:
  1. **Program Counter - PC:** Que armazena o endereço da ROM que contém o trecho do programa que deve ser executado;
  2. **Instruction Register - IR:** Que armazena a instrução, ou o operando desta, a ser executada pelo processador. O IR também é conectado diretamente com o Bloco de controle;
  3. **B:** Registrador de propósito geral, que pode armazenar dados vindos da ULA;
  4. **A:** Registrador de propósito geral, podendo armazenar dados da RAM, da ULA ou diretamente do registrador B;
- 3 Barramentos de dados:
  1. **Bus C:** Pode receber dados dos Registradores **A** ou **PC**;
  2. **Bus D:** Pode receber dados dos Registradores **B** ou **IR**;

3. **Saída da ULA:** Disponibiliza dados para os endereços da RAM e da ROM, além dos registradores **PC**, **B** e **A**;

- Uma memória ROM, que disponibiliza dados diretamente para o registrador **IR**, dependendo do endereço definido pela saída da ULA;
- Uma memória RAM, que troca dados diretamente com o registrador **A**;

2.2 BLOCO DE CONTROLE

O Bloco de Controle foi projetado baseando-se numa implementação de uma máquina de estado com desvios condicionais dependendo dos dados fornecidos pelo Bloco de Dados (Instrução e Flags). Em cada estado, dependendo da instrução que está sendo executada pelo processador, o Bloco de Controle gera os sinais para os demais dispositivos de Hardware executarem corretamente o que foi programado. A máquina de Estados é representada na figura 4.

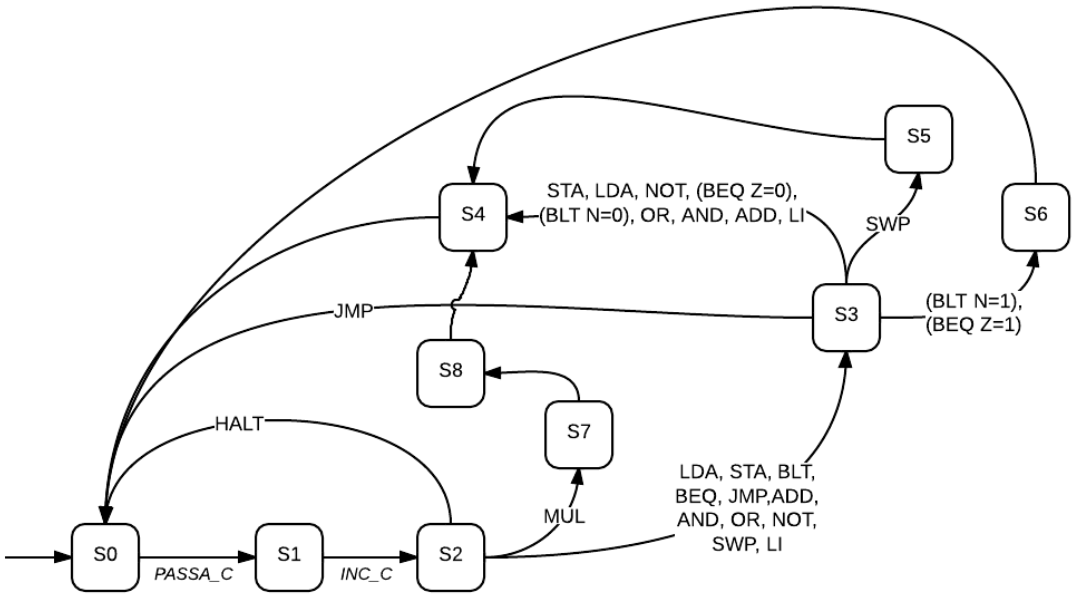

Figura 4: Máquina de estados implementada pelo Bloco de Controle

Os estados podem ter a seguinte compreensão lógica:

- **S0:** É o estado inicial da máquina e será sempre responsável por atualizar o valor de **PC** e atualizá-lo no endereço da memória ROM;
- **S1:** Responsável por transferir a instrução gravada na ROM para **IR** e incrementar o valor de **PC** e atualizá-lo no endereço da memória ROM;
- **S2:** Neste estado o Bloco de Controle já sabe que instrução está tratando e a ULA já executa a primeira operação específica de cada instrução, a partir daqui transições condicionais já são executadas;
- **S3:** Neste estado a ULA executa a segunda operação específica da instrução;
- **S4:** Neste estado **PC** é incrementado e o processador fica pronto para voltar ao estado inicial;
- **S5:** É um estado específico para implementar a função *SWP*, habilita a ULA para seu funcionamento normal;
- **S6:** Estado que trata as funções *BLT* e *BEQ* caso os flags sejam acionados;
- **S7:** Específico para executar a primeira parte da instrução *MUL*;
- **S8:** Específico para executar a segunda parte da instrução *MUL*;

2.3 VALORES DOS REGISTRADORES PARA CADA OPCODE

Para as análises em questão será considerado, para cada função, que todos os registradores estão zerados e o bloco controlador está em seu estado inicial.

Todas as instruções terão em comum o mesmo comportamento dos registradores de acordo com a tabela 1 abaixo.

Tabela 1: Tabela de valores de registradores comum a todos as instruções

CLK	PC	IR	A	B	Sx
1	0b00000000	0b00000000	0bXXXXXXXX	0bYYYYYYYY	S0
2	0b00000000	instrução	0bXXXXXXXX	0bYYYYYYYY	S1

Os demais comportamentos dos registradores para cada estado de todas as instruções estão descritos nas tabelas abaixo

2.4 INSTRUÇÕES TIPO I

2.4.1 LDA end

CLK	PC	IR	A	B	Sx
3	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	end	valor	0bYYYYYYYY	S3
5	0b00000001	end	valor	0bYYYYYYYY	S4
6	0b00000002	end	valor	0bYYYYYYYY	S0

2.4.2 STA end

CLK	PC	IR	A	B	Sx
3	0b00000001	end	valor	0bYYYYYYYY	S2
4	0b00000001	end	valor	0bYYYYYYYY	S3
5	0b00000001	end	valor	0bYYYYYYYY	S4
6	0b00000002	end	valor	0bYYYYYYYY	S0

STA grava na RAM no estado S3

2.4.3 BLT end

CLK	PC	IR	A	B	Sx
3	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S3

Se A < B, a ULA gera o *flag* ***N*** e o comportamento será:

CLK	PC	IR	A	B	Sx
5	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S6
6	end	end	0bXXXXXXXX	0bYYYYYYYY	S0

Se A > B, a ULA não gera o *flag* ***N*** e o comportamento será:

CLK	PC	IR	A	B	Sx
5	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S6
6	0b00000002	end	0bXXXXXXXX	0bYYYYYYYY	S0

2.4.4 BEQ end

CLK	PC	IR	A	B	Sx
3	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S3

Se A = B, a ULA gera o *flag* ***Z*** e o comportamento será:

CLK	PC	IR	A	B	Sx
5	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S6
6	end	end	0bXXXXXXXX	0bYYYYYYYY	S0

Se A ! = B, a ULA não gera o *flag* ***Z*** e o comportamento será:

CLK	PC	IR	A	B	Sx
5	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S6
6	0b00000002	end	0bXXXXXXXX	0bYYYYYYYY	S0

2.5 INSTRUÇÕES TIPO J

2.5.1 JMP end

CLK	PC	IR	A	B	Sx
3	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	end	0bXXXXXXXX	0bYYYYYYYY	S3
5	end	end	valor	0bYYYYYYYY	S0

2.6 INSTRUÇÕES TIPO R

2.6.1 ADD

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	0bZZZZZZZZ	A+B	0bYYYYYYYY	S3
5	0b00000001	0bZZZZZZZZ	A+B	0bYYYYYYYY	S4
6	0b00000002	0bZZZZZZZZ	A+B	0bYYYYYYYY	S0

2.6.2 MUL

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	0bZZZZZZZZ	LSB(A*B)	0bYYYYYYYY	S7
5	0b00000001	0bZZZZZZZZ	LSB(A*B)	HSB(A*B)	S8
6	0b00000002	0bZZZZZZZZ	LSB(A*B)	HSB(A*B)	S0

2.6.3 AND

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	0bZZZZZZZZ	A AND B	0bYYYYYYYY	S3
5	0b00000001	0bZZZZZZZZ	A AND B	0bYYYYYYYY	S4
6	0b00000002	0bZZZZZZZZ	A AND B	0bYYYYYYYY	S0

2.6.4 OR

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	0bZZZZZZZZ	A OR B	0bYYYYYYYY	S3
5	0b00000001	0bZZZZZZZZ	A OR B	0bYYYYYYYY	S4
6	0b00000002	0bZZZZZZZZ	A OR B	0bYYYYYYYY	S0

2.6.5 NOT

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	0bZZZZZZZZ	NOT A	0bYYYYYYYY	S3
5	0b00000001	0bZZZZZZZZ	NOT A	0bYYYYYYYY	S4
6	0b00000002	0bZZZZZZZZ	NOT A	0bYYYYYYYY	S0

2.6.6 SWP

CLK	PC	IR	A	B	Sx	Observação
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2	-
4	0b00000001	0bZZZZZZZZ	B	0bYYYYYYYY	S3	$ULA\_en = 0$
5	0b00000001	0bZZZZZZZZ	B	A	S5	$ULA\_en = 1$
6	0b00000001	0bZZZZZZZZ	B	A	S4	-
7	0b00000002	0bZZZZZZZZ	B	A	S0	-

No estado S3, o processador joga o valor de A para a ULA e depois a desabilita (comportamento de latch). Como A só carrega o seu valor na borda de descida do clock, é possível no mesmo ciclo Armazenar B em A e deixar o valor antigo de A armazenado (através de Latches) na ULA.

2.6.7 LI const

CLK	PC	IR	A	B	Sx
3	0b00000001	const	0bXXXXXXXX	0bYYYYYYYY	S2
4	0b00000001	const	const	0bYYYYYYYY	S3
5	0b00000001	const	const	0bYYYYYYYY	S4
6	0b00000002	const	const	0bYYYYYYYY	S0

2.7 INSTRUÇÕES TIPO I

2.7.1 HALT

CLK	PC	IR	A	B	Sx
3	0b00000001	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S2
6	0b00000000	0bZZZZZZZZ	0bXXXXXXXX	0bYYYYYYYY	S0

3 IMPLEMENTAÇÃO

3.1 BLOCO DE DADOS

A arquitetura implementada efetivamente em VHDL pode ser observada pelo RTL da figura 5.

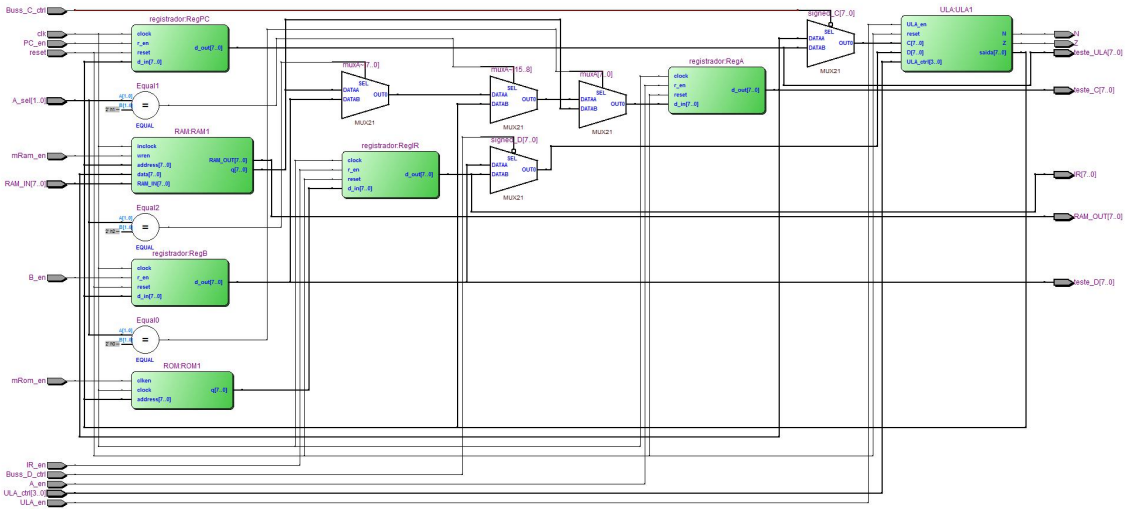

Figura 5: Arquitetura implementada em VHDL para o Bloco de Dados.

Os principais pontos que diferem do projeto inicial (figura 3) do implementado são:

- **A** só grava o dado na borda de descida do clock (*not_clk*). Esta implementação foi feita para facilitar o tratamento de funções como *SWP* evitando a necessidade de passos adicionais;

- **A** recebe sinais advindos, além da ULA e da RAM, do registrador **B**. Esta alteração no multiplexador de entrada de **A** foi realizada para implementar mais facilmente a função *SWP*;

- Os barramentos de dados C e D passam a ser controlados por multiplexadores em vez dos *tristates* de saída dos registradores. Esta medida foi tomada para reduzir a quantidade de sinais de controle.

- Foi colocado um sinal de enable de saída na ULA (*ULA_en*) posibilitando o comportamento de latch da mesma;

- Foram acrescentados sinais de reset em todos os componentes a fim de garantir uma correta inicialização do processador;

- Foram implementados sinais extras a fim de facilitar o *debug* e a análise da simulação;

Para cada uma das subsseções abaixo foi criado um novo projeto de hardware em VHDL e foi montado um *testbench* baseado em estímulos gerados por arquivos *.vhd*.

A estrutura final do Bloco de Dados foi montada usando a estrutura de *COMPONENTS* e *PORT MAP*, excetuando-se alguns multiplexadores que foram implementados utilizando lógica *IF-THEN-ELSE*.

3.1.1 ULA

As operações implementadas pela ULA são descritas na tabela 2.

Tabela 2: Funções implementadas na ULA

Função	Código	Instruções que utilizam	Descrição
PASSA_C	0b0000	Todas	Passa o conteúdo de C para saída da ULA
PASSA_D	0b0001	Todas	Passa o conteúdo de D para saída da ULA
INC_C	0b0110	Todas	Incrementa 1 no conteúdo de C
C AND D	0b0010	AND	Executa operação AND entre C e D
C OR D	0b0011	OR	Executa operação OR entre C e D
NOT C	0b0100	NOT	Executa operação inversão de bits com C
$C + D$	0b0101	ADD	Soma C com D (Complemento de 2)
$C - D$	0b1001	BEQ, BLT	Subtrai D de C (Complemento de 2)
$C * D$	0b1010	MUL	Armazena $C * D$ num registrador interno
$LSB(C * D)$	0b0111	MUL	Passa os 8 primeiros bits de $C * D$
$HSB(C * D)$	0b1000	MUL	Passa os 8 últimos bits de $C * D$
DEC_C	0b1011	HALT	Decrementa 1 no conteúdo de C

Os flags Z(indicando zero) e N(indicando números negativos) na ULA são tratados em processos assíncronos e estão ativos sempre que a saída da ULA estiver ativa.

As operações aritiméticas são realizadas baseadas na biblioteca *IEEE.numeric_std* com sinais do tipo *SIGNED*.

3.1.2 RAM

A memória RAM foi desenvolvida com o auxílio da ferramenta *Mega Wizard* do *Quartus 12.3*, sendo composta por: 256 palavras de 8 bits (equivale a 8 bits de endereçamento); uma saída de dados; uma entrada de endereçamento; uma entrada de dados; e uma entrada de enable de gravação.

O mapeamento da memória foi feito utilizando lógica *IF-THEN-ELSE*, diretamente no arquivo gerado pelo *Mega Wizard*.

3.1.3 ROM

A memória ROM foi desenvolvida com o auxílio da ferramenta *Mega Wizard* do *Quartus 12.3*, sendo composta por: 256 palavras de 8 bits (equivale a 8 bits de endereçamento); uma saída de dados; uma entrada de endereçamento; e uma entrada de habilitação de saída de dados.

Sempre um Ciclo de Clock antes de **IR** ler da ROM, é necessário habilitar a saída de dados da mesma.

O programa é gravado na ROM através de um arquivo de inicialização com extensão *.mif* como exemplificado posteriormente na seção 3.3.

3.1.4 Registradores

Todos os registradores desenvolvidos são estruturas simples seguindo a mesma filosofia dos registradores desenvolvidos pelo treinamento *Altera* reproduzido em aula.

3.2 O PROCESSADOR PIRES

A arquitetura final do processador Pires, implementada em VHDL, pode ser observada pelo RTL da figura 6.

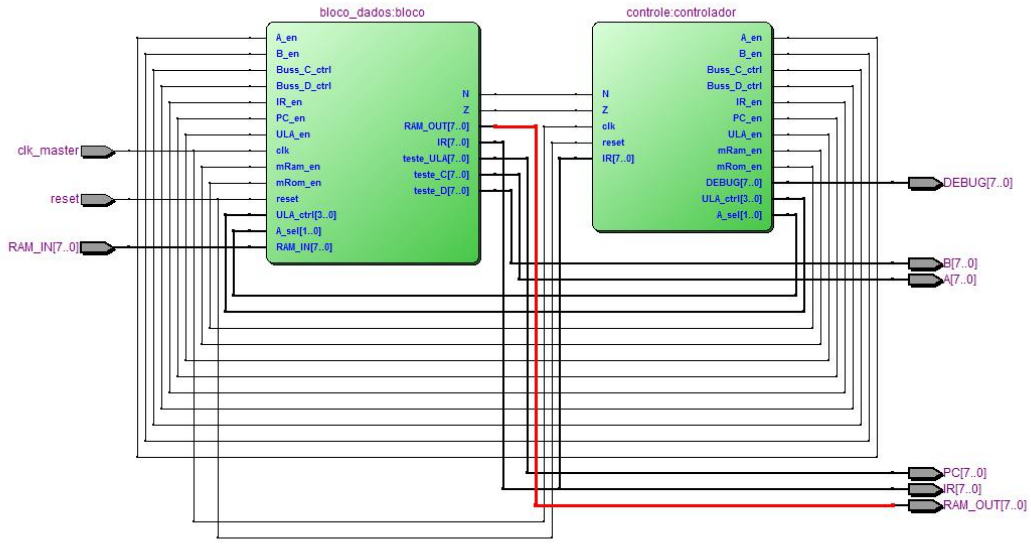

Figura 6: Processador Pires implementado em VHDL.

Esta implementação já está pronta para ser embarcada no kit de desenvolvimento baseado no FPGA *Ciclone II* da *Altera*, Sendo os sinais:

- *clk_master:* Entrada de clock do processador;
- *RAM_OUT:* Pinos de saída ligados ao mapeamento da RAM;
- *RAM_IN:* Pinos de entrada ligados ao mapeamento da RAM;
- *reset:* Entrada de reset do processador;
- *DEBUG:* Sinal útil na simulação, utilizado para identificar em qual estado estão as operações do processador;
- *A:* Sinal para simulação, mostrando o conteúdo do registrador A;
- *B:* Sinal para simulação, mostrando o conteúdo do registrador B;
- *PC:* Sinal para simulação, mostrando o conteúdo do registrador PC;
- *IR:* Sinal para simulação, mostrando o conteúdo do registrador IR;

A figura 6 também mostra as conexões feitas entre as entidades Bloco_dados e Controle, implementadas pela entidade *Top-Level* do processador.

3.3 SIMULAÇÃO DE UM PROGRAMA EM ASSEMBLY

O programa a ser simulado aparece descrito no código Assembly abaixo:

```
#programa assembly  pra memoria ROM_mem.mif #

Inicio:
        LI 00; # carrega A com 0;
        STA 0x10; # grava no endereco 16
        LI 01; # -- Carrega 5 em A
        SWP; #-- B <=> A
        LI 01;
        MUL; #B*A => 1*1 = 01;
        SWP;
        LI 01;
        ADD; #A+B = 1 + 1 = 02;

Loop:
        SWP;
```

```
        LI -1;
        ADD;
        SWP;
        LDA 0x10; #le do endereco 16
        BLT Loop; #desvia pro loop o valor na memoria seja menor que B
Fim:
        HALT;
```

Apenas para fins de teste das funções implementadas, o algoritmo carrega o valor 0 na memória RAM (endereço 16), executa operações matemáticas com os operadores A e B e decrementa o resultado final destas operações até que este valor seja inferior ao valor gravado na posição de memória 16 da RAM, quando isto acontece, o processamento entra em suspensão (*HALT*).

O arquivo de inicialização *ROM_mem.mif* que contém o programa descrito pode ser visto abaixo:

```
%-- Arquivo de descricao dos dados da memoria ROM
DEPTH = 256; % Profundidade (quantidade de palavras)
WIDTH = 8; % numero de bits de cada palavra%
ADDRESS_RADIX = HEX; % Escolhe qual vai ser o radix para a escolha do endereco %
DATA_RADIX = BIN; % Escolhe qual vai ser o radix para a escolha do dado %
% Caso nao especifique nada Radix = HEX %
%-- Specify values for addresses, which can be single address or range
CONTENT
        BEGIN
        0       :       01100000; % LI
        1       :       00000000; % 00
        2       :       00000010; %     STA
        3       :       00010000; % 0x10 => 16 em decimal
        4       :       01100000; % LI
        5       :       00000001; % 01
        6       :       01110000; % SWP
        7       :       11111111; %
        8       :       01100000; % LI
        9       :       00000001; % 01
        0A      :       00100000; % MUL
        0B      :       00000101; %
        0C      :       01110000; % SWP
        0D      :       00000101; %
        0E      :       01100000; % LI
        0F      :       00000001; % 01
        10      :       00010000; % ADD
        11      :       00010000;
        12      :       01110000; % Loop: SWP
        13      :       01110000; %
        14      :       01100000; % LI
        15      :       11111111; %-1 em complemento de 2
        16      :       00010000; %ADD
        17      :       00010000;
        18      :       01110000; % SWP
        19      :       01110000; %
        1A      :       00000001; % LDA
        1B      :       00010000; % 0x10 => 16
        1C      :       00000011; % BLT
        1D      :       00010010; % endereco 0x12 => 18
        1E      :       11111111; % HALT
        1F      :       11111111; %
        [20..FF] : 00000000; %zera o restante da ROM
END ; % eh preciso terminar o arquivo com este end %
```

O resultado simulado do programa pode ser observado nas figuras 7, 8 e 9, onde é possível conferir que o valor dos registradores segue o esperado pelo programa, bem como o comportamento de PC e o fluxo do programa sendo alterado pela instrução *BLT*. É possível também acompanhar o estado em que se encontra o processador em cada instrução.

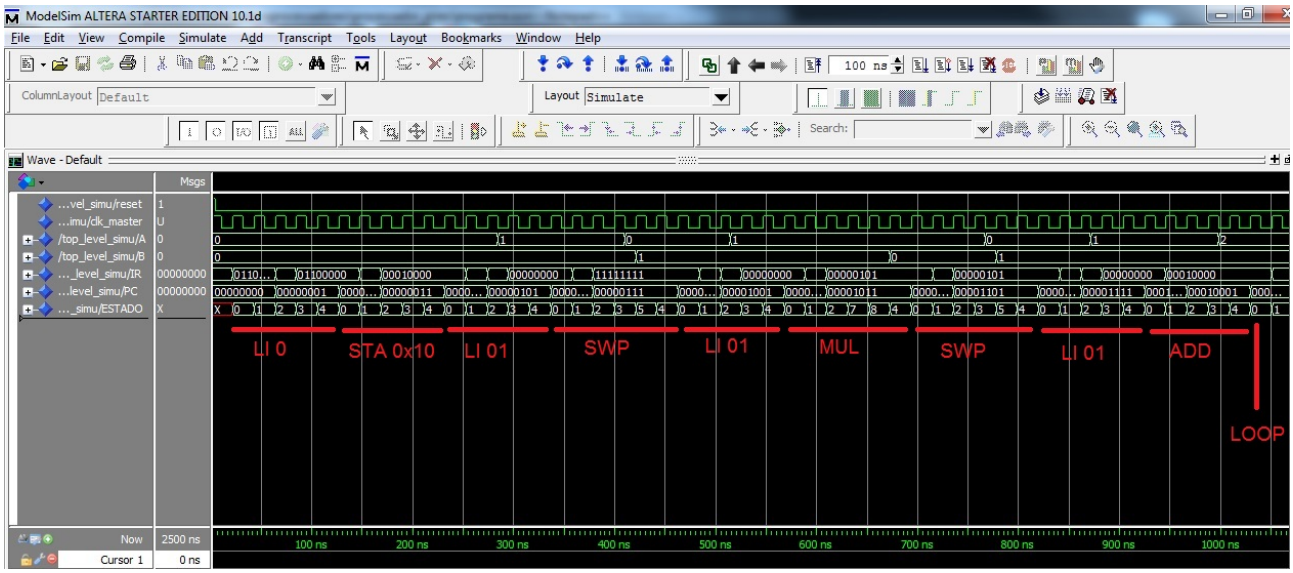

Figura 7: Primeira parte da simulação do programa

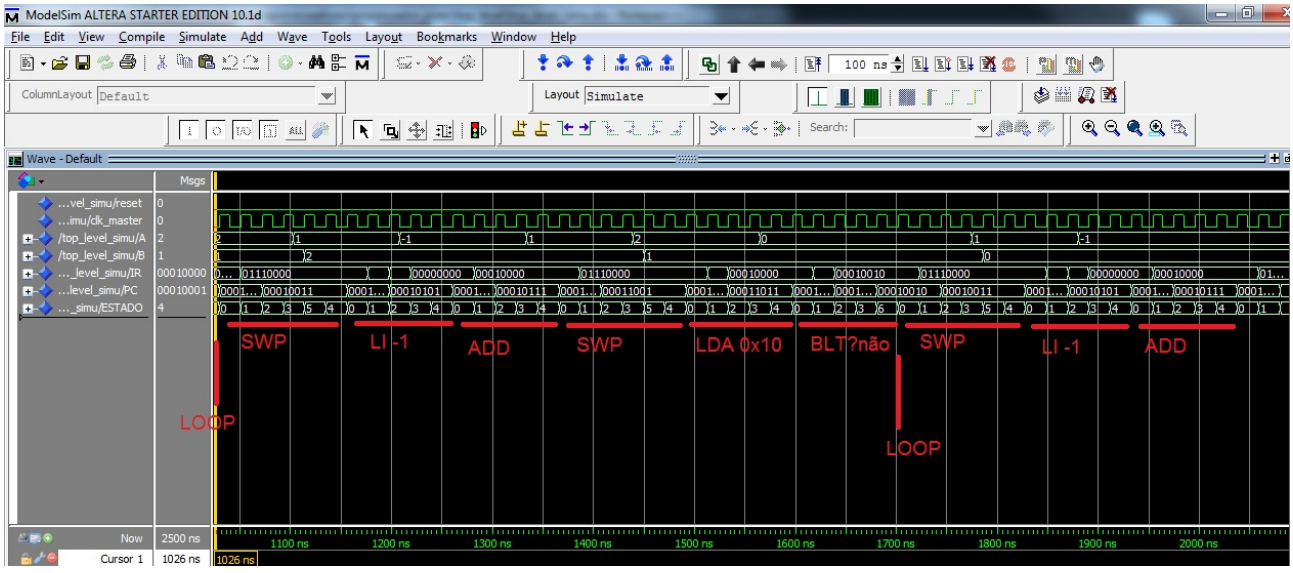

Figura 8: Segunda parte da simulação do programa

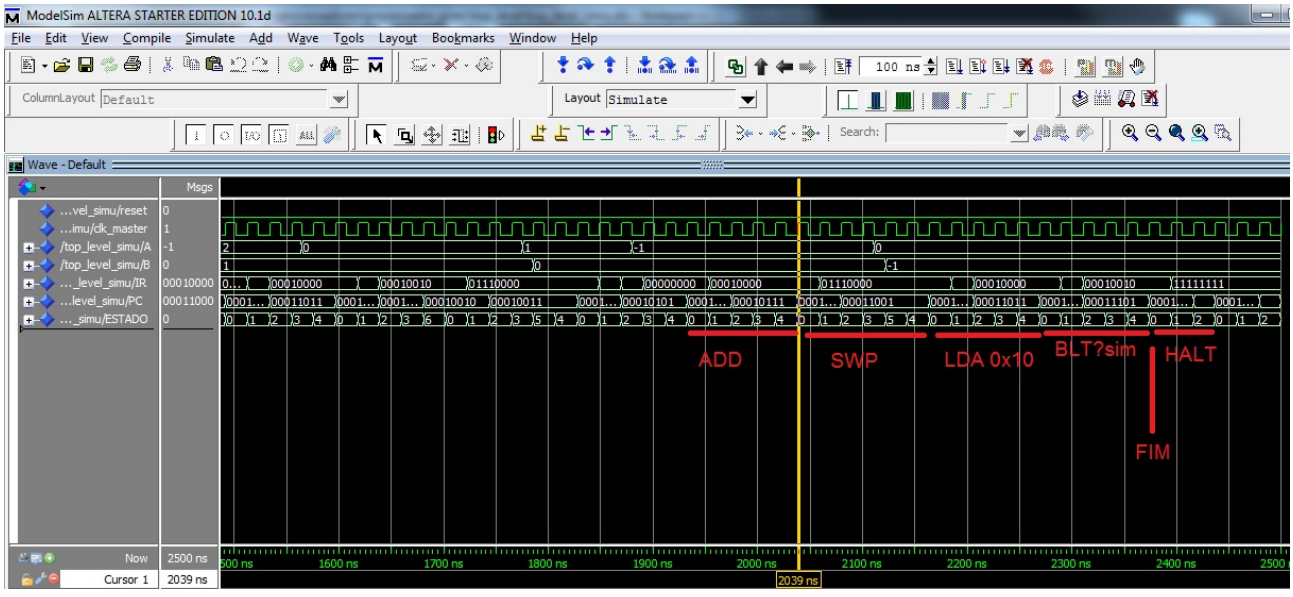

Figura 9: Terceira parte da simulação do programa