Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

SSC0108 - Prática em Sistemas Digitais

Pedro Calciolari Jardim – 11233668

Joao Vitor Pereira Candido – 13751131

Maicon Chaves Marques - 14593530

1 - Introdução

Neste trabalho foi implementado um processador de 8 bits, desenvolvido para uma Intel FPGA, utilizando o Software Intel® Quartus® Prime, através da linguagem VHDL, linguagem de descrição de hardware VHSIC "Very High Speed Integrated Circuits". O projeto da CPU envolveu os componentes: Unidade de Controle (UC), Unidade Lógica Aritmética (ULA) e a Unidade de Memória.

2 - Desenvolvimento

A seguir será explicado como a estrutura do processador foi desenvolvida na UC e na ULA .

2.1 - Unidade de Controle (UC)

2.1.1 - Variáveis da Estrutura

As seguintes variáveis foram utilizadas para organizar e manipular as informações no processador:

```
signal RA : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal RB : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal RR : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal PC : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal IR : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
signal ESPERA        : STD_LOGIC := '0';
```

Optou-se por utilizar registradores do tipo STD_LOGIC_VECTOR, de 8 bits cada para estruturar as atividades da CPU.

2.1.2 - Estados do Processador

A máquina de estados que gere a CPU foi dividida inicialmente em 3 estados: Busca, Decodificação e Execução. No entanto, por limitações do tamanho da instrução e por ciclos de clock da memória de acesso aleatório (RAM), optou-se por usar 5 estados: Busca1, Busca2, Decodificação, Execução1 e Execução2. Vale ressaltar que nem todas as instruções vão passar por todos os estados, somente aqueles que precisarem usarão todos os estados disponíveis.

```
TYPE STATES is (fetch, fetch2, decode, exec, exec2);
case state is
```

```
            when fetch =>
            …
            STATE := fetch2;
            when fetch2 =>
            …
            STATE := decode;
            when decode =>
            …
            state := exec;
            when exec =>
            …
            state := exec2;
            when exec2 =>
            …
            state := fetch;
    end case;
```

O processador inicia-se no estado de busca, e a cada ciclo de clock vai para o próximo estado ou volta para a busca. A mudança dos estados segue essencialmente a ordem do código acima.

2.1.3 - Instruções do Processador

O processador possui um total de 15 instruções que podem ser observadas nas constantes abaixo listadas:

```
CONSTANT ADD : STD_LOGIC_VECTOR(3 downto 0) := "0000";
CONSTANT SUB  : STD_LOGIC_VECTOR(3 downto 0) := "0001";
CONSTANT AND  : STD_LOGIC_VECTOR(3 downto 0) := "0010";
CONSTANT OR : STD_LOGIC_VECTOR(3 downto 0) := "0011";
CONSTANT NOT  : STD_LOGIC_VECTOR(3 downto 0) := "0100";
CONSTANT CMP : STD_LOGIC_VECTOR(3 downto 0) := "0101";
CONSTANT JMP : STD_LOGIC_VECTOR(3 downto 0) := "0110";
CONSTANT JEQ  : STD_LOGIC_VECTOR(3 downto 0) := "0111";
CONSTANT JGR : STD_LOGIC_VECTOR(3 downto 0) := "1000";
CONSTANT LOAD : STD_LOGIC_VECTOR(3 downto 0) := "1001";
CONSTANT STORE  : STD_LOGIC_VECTOR(3 downto 0) := "1010";
CONSTANT MOV : STD_LOGIC_VECTOR(3 downto 0) := "1011";
CONSTANT IN : STD_LOGIC_VECTOR(3 downto 0) := "1100";
CONSTANT OUT  : STD_LOGIC_VECTOR(3 downto 0) := "1101";
CONSTANT WAIT : STD_LOGIC_VECTOR(3 downto 0) := "1110";
```

Cada instrução possui 8 bits, os primeiros 4 bits de cada instrução identificam qual é o seu tipo e os bits subsequentes irão indicar quais parâmetros serão usados.

2.1.3.1 - Operações Aritméticas

O processador possui 6 operações aritméticas, no código em VHDL, as instruções foram todas unidades em único caso, com exceção da instrução NOT. A instrução NOT foi separada pois necessita de apenas um único operador.

```
--=====================================================================
-- ADD ou SUB ou AND ou OR ou CMP
--=====================================================================
IF(IR(7 DOWNTO 4) = ADD or (IR(7 DOWNTO 4) = "0001") or (IR(7 DOWNTO 4) = "0010")
or (IR(7 DOWNTO 4) = "0011") or (IR(7 DOWNTO 4) = "0101")) THEN

        IF((IR(3 DOWNTO 0) = "0000")) THEN  -- A A
                X <= RA;
                Y <= RA;
                INIT_ULA <= '1';
        ELSIF(IR(3 DOWNTO 0) = "0011") THEN  -- A i
                X <= RA;
                raddress <= PC;
                …
--=====================================================================
```

Operações aritméticas utilizam da ULA, e colocam os operadores em variáveis próprias para que a mesma possa realizar as operações. Após todos os sinais estarem no lugar, um sinal é emitido para que a unidade lógica possa iniciar os cálculos. Os estados seguintes das operações aritméticas são usados para pegar o valor imediato, endereçando a memória e atualizando as entradas novamente, mas possuem comportamento idêntico.

2.1.3.2 - Operações de Desvio

As operações de desvio servem para mudar o curso do programa caso uma variável tenha um valor específico. Nesse caso utilizou-se 3 instruções de desvio.

```
--=====================================================================
-- JMP
--=====================================================================
        IF(IR(7 DOWNTO 4) = JMP) THEN
                raddress <= PC;
                state := exec;
        END IF;
--=====================================================================
-- JEQ
```

```
--=====================================================================
        IF(IR(7 DOWNTO 4) = JEQ) THEN
                IF(FRzero = '1') THEN
                raddress <= PC;
                END IF;
                state := exec;
        END IF;
--=====================================================================
-- JGR
--=====================================================================
        IF(IR(7 DOWNTO 4) = JGR) THEN
                IF(FRzero = '0' and FRmaior ='1') THEN
                        raddress <= PC;
                        END IF;
                        state := exec;
                END IF;
--=====================================================================
```

A lógica implementada nos três desvios foi muito semelhante, o jump incondicional endereça a memória com valor imediato abaixo, e atualiza o registrador PC para que vá para o lugar especificado. Já os desvios condicionais verificam as Flags deixadas pela ULA. Após uma comparação, a unidade aritmética mantém o resultado da comparação entre as variáveis, e então olhando para esse sinal é possível fazer ou não o desvio, baseado no resultado do cálculo anterior comparativo.

mas possuem comportamento idêntico.

2.1.3.3 - Operações de LOAD e STORE

As operações de acesso e alteração da memória foram definidas como LOAD e STORE.

Para a instrução de LOAD, desejamos carregar no primeiro operando, o valor do endereço que está no registrador ou no valor imediato logo abaixo. Inicialmente então endereça a memória com o valor do registrador ou o valor imediato. Assim que a saída for gerada, esse valor é colocado no valor destino, que é o primeiro parâmetro.

```
--=====================================================================
-- LOAD
--=====================================================================
        IF (IR(7 DOWNTO 4) = LOAD) THEN
```

```
                IF (IR(3 DOWNTO 0) = "0000") THEN  -- A A: Carrega para o registrador RA o conteúdo do endereço em RA
                        raddress <= RA;
                …
                        RA <= data_out;
                ELSIF (IR(3 DOWNTO 0) = "0011") THEN  -- A i: Carrega para o registrador RA o conteúdo do endereço imediato
                        raddress <= PC;
                …
                        RA <= data_out;
--=====================================================================
```

Já na instrução de STORE deseja-se fazer o contrário, o objetivo é colocar o valor do Registrador especificado, que caracteriza o primeiro parâmetro da instrução no endereço especificado como registrador ou imediato no segundo parâmetro da instrução. Após obter todos os operadore, basta endereçar a memória com o sinal de escrita ligado e alterar o valor da entrada de dados com o conteúdo do registrador.

```
--=====================================================================
-- STORE
--=====================================================================
        IF (IR(7 DOWNTO 4) = STORE) THEN
                IF (IR(3 DOWNTO 0) = "0000") THEN  -- A A:
                        waddress <= RA;
                        data_in <= RA;
                        write1 <= '1';
                ELSIF (IR(3 DOWNTO 0) = "0011") THEN  -- A i:
                         raddress <= PC;
                …
                        waddress <= data_out;
                        data_in <= RA;
                        PC <= STD_LOGIC_VECTOR(unsigned(PC) + 1);
                        write1 <= '1';
--=====================================================================
```

2.1.3.4 - Operação de MOV

A operação de MOV tem por objetivo transferir o conteúdo de um registrador ou valor imediato para outro registrador. Caso o valor a ser transferido seja um valor imediato, faz-se a transição entre os estados a fim de completar a operação.

```
--=====================================================================
-- MOV
```

```
--=====================================================================
                IF (IR(7 DOWNTO 4) = MOV) THEN
                        IF (IR(3 DOWNTO 0) = "0001") THEN  -- A B: Move o valor de RB para RA
                                RA <= RB;
                         ELSIF (IR(3 DOWNTO 0) = "0011") THEN  -- A i: Move o valor imediato para RA
                                raddress <= PC;
                        …
                                RA <= data_out;
--=====================================================================
```

2.1.3.5 - Operações de IN, OUT e WAIT

O processador possui 3 operações que fazem intermédio com o usuário, e dizem respeito a entrada e saída.

```
--=====================================================================
-- IN
--=====================================================================
        IF (IR(7 DOWNTO 4) = IN1) THEN
                IF (IR(3 DOWNTO 2) = "00") THEN  -- A
                        RA <= entrada;
                ELSIF (IR(3 DOWNTO 2) = "01") THEN  -- B
                        RB <= entrada;
                ELSIF (IR(3 DOWNTO 2) = "10") THEN  -- R
                        RR <= entrada;
                 END IF;
        state := fetch;
        END IF;
--=====================================================================
```

A operação de entrada simplesmente aloca a entrada no registrador indicado.

```
--=====================================================================
-- OUT
--=====================================================================
        IF (IR(7 DOWNTO 4) = OUT1) THEN
                IF (IR(3 DOWNTO 2) = "00") THEN  -- A
                        saidaREAL <= RA;
                 ELSIF (IR(3 DOWNTO 2) = "01") THEN  -- B
                        saidaREAL <= RB;
                ELSIF (IR(3 DOWNTO 2) = "10") THEN  -- R
                        saidaREAL <= RR;
                END IF;
        state := fetch;
        END IF;
```

```
--=====================================================================
```

A operação de saída simplesmente coloca o registrador indicado na saída.

```
--=====================================================================
-- WAIT
--=====================================================================
                        IF (IR(7 DOWNTO 4) = WAIT1) THEN
                                IF(ESPERA = '0') THEN
                                        ESPERA <= '1';
                                        state := decode;
                                ELSE
                                        IF(WAIT2 = '1') THEN
                                                ESPERA <= '0';
                                                state := fetch;
                                        END IF;
                                END IF;
                        END IF;

--=====================================================================
```

Já a operação de WAIT, mantém o processador em espera preso em um loop no estado atual e o mantém assim até que um entrada específica seja detectada. Após isso, o estado volta ao estado de busca e o processador pode continuar a executar outras instruções.

2.2 - Unidade Lógica Aritmética (ULA)

A unidade lógica Aritmética é o componente do processador que processa as instruções que de fato computam informações. Para melhor manipulação dos dados, a ULA possui variáveis de controle, além das variáveis de entrada e saída. Tais variáveis permitem organizar o fluxo de dados e direcionar o resultado das operações.

```
--=====================================================================
                signal X : STD_LOGIC_VECTOR (7 downto 0) := (others => '0'); --Operadores da ULA
                signal Y : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
                signal RESULT   : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
                signal INIT_ULA : STD_LOGIC := '0';
                signal FRzero   : STD_LOGIC := '0';
                signal FRmaior  : STD_LOGIC := '0';
                signal FRmenor  : STD_LOGIC := '0';
                signal FRover   : STD_LOGIC := '0';
--=====================================================================
```

A ULA também possui variáveis internas, que são registradores que mantêm a operação temporariamente e que permitem a tomada de decisões.

```
--=====================================================================
        VARIABLE AUX : STD_LOGIC_VECTOR(7 downto 0);
        VARIABLE RESULT15 : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
        VARIABLE SUPERIOR : STD_LOGIC_VECTOR(15 downto 0) := "0000000001111111";
        VARIABLE INFERIOR : STD_LOGIC_VECTOR(15 downto 0) := "1111111100000001";
        VARIABLE X_SOMA : STD_LOGIC_VECTOR(15 downto 0) := "00000000" & X;
        VARIABLE Y_SOMA : STD_LOGIC_VECTOR(15 downto 0) := "00000000" & Y;
--=====================================================================
```

A variável ‘signal INIT_ULA : STD_LOGIC’ indica se a ULA deve ou não funcionar e realizar uma operação.

2.2.1 - Operação de ADD

A operação de adição utiliza dois operadores e faz a soma com carry in para os bits subsequentes. Além disso, testa a possibilidade de overflow.

```
--=====================================================================
    IF(IR(7 DOWNTO 4) = ADD) THEN
        RESULT15 := STD_LOGIC_VECTOR(signed(X_SOMA) + signed(Y_SOMA));
        -- Trunca RESULT15 para 8 bits e atribui a AUX
        AUX := RESULT15(7 downto 0);

        -- Verificação de overflow usando comparação
    if (signed(RESULT15) > signed(SUPERIOR)) THEN  -- Limite máximo
        FRover <= '1';  -- Overflow: resultado maior que o maior valor positivo
    elsif (signed(RESULT15) < signed(INFERIOR)) THEN  -- Limite mínimo
        FRover <= '1';  -- Overflow: resultado menor que o maior valor negativo
    else
        FRover <= '0';  -- Sem overflow
    end if;
    RESULT <= AUX;
--=====================================================================
```

2.2.2 - Operação de SUB e CMP

Semelhante a operação de soma, a subtração e comparação faz subtração com carry in para os bits subsequentes. Além disso, testa a possibilidade de overflow, e liga as flags que são utilizadas para comparação.

```
--=====================================================================
    IF(IR(7 DOWNTO 4) = SUB OR IR(7 DOWNTO 4) = CMP) THEN
        RESULT15 := STD_LOGIC_VECTOR(signed(X_SOMA) - signed(Y_SOMA));
```

```
		-- Trunca RESULT15 para 8 bits e atribui a AUX
		AUX := RESULT15(7 downto 0);

		-- Verificação de overflow usando comparação
	if (signed(RESULT15) > signed(SUPERIOR)) THEN  -- Limite máximo
		FRover <= '1';  -- Overflow: resultado maior que o maior valor positivo
	elsif (signed(RESULT15) < signed(INFERIOR)) THEN  -- Limite mínimo
		FRover <= '1';  -- Overflow: resultado menor que o maior valor negativo
	else
		FRover <= '0';  -- Sem overflow
	end if;
	if (RESULT15(15) = '1') then
		FRzero <= '0';
		FRmaior <= '0';
		FRmenor <= '1';
		AUX := STD_LOGIC_VECTOR(unsigned(NOT RESULT15(7 downto 0))+1);
	elsif (RESULT15(15 DOWNTO 0) = "0000000000000000") then
		FRzero <= '1';
		FRmaior <= '0';
		FRmenor <= '0';
	AUX := STD_LOGIC_VECTOR(RESULT15(7 downto 0));
		else
		FRzero <= '0';
		FRmaior <= '1';
		FRmenor <= '0';
	AUX := STD_LOGIC_VECTOR(RESULT15(7 downto 0));
		end if;
		IF IR(7 DOWNTO 4) = SUB THEN
			RESULT <= AUX;
		ELSE
			RESULT <= RR;
		END IF;
--=========================================================================
```

2.2.3 - Operação de AND

Realiza a operação de AND bit a bit com os dois operadores passados como parâmetro na ULA.

```
--=========================================================================
		ELSIF(IR(7 DOWNTO 4) = AND1) THEN
			RESULT <= X AND Y;
--=========================================================================
```

2.2.4 - Operação de OR

Realiza a operação de OR bit a bit com os dois operadores passados como parâmetro na ULA.

```
--=====================================================================
                ELSIF(IR(7 DOWNTO 4) = OR1) THEN
                        RESULT <= X OR Y;
--=====================================================================
```

2.2.5 - Operação de NOT

Realiza a operação de NOT bit a bit com o único operador passado como parâmetro na ULA.

```
--=====================================================================
                ELSIF(IR(7 DOWNTO 4) = NOT1) THEN
                        RESULT <= NOT X;
--=====================================================================
```

2.2.6 - Resultado

Toda operação que passou pela ULA deve ser colocada no registrador resultado RR, com exceção das operações de comparação. Assim que há uma borda de subida no Clock, a UC verifica se a ULA realizou uma operação recentemente, então atualiza o registrador resultado, para que possa receber o resultado da ULA.

```
--=====================================================================
                        IF(INIT_ULA = '1') THEN --Controle das saidas da ULA
                                RR <= RESULT;
                                INIT_ULA <= '0';
                        END IF;
--=====================================================================
```

2.3 - Memória (RAM)

A memória RAM foi instanciada de acordo com o módulo existente no Quartus, apresentado na “Aula 7 - Memory Blocks”, onde existe um sinal para ativação da escrita e endereços separados para leitura e escrita. Foram utilizadas variáveis internas da UC para tomadas de decisão e manipulação dos dados na memória. A memória instanciada tem especificações de 2048 bits, sendo 256 endereços com 8 bits cada.

```
--=====================================================================
```

```
signal data_in    : std_logic_vector(7 downto 0) := (others => '0');
signal raddress   : std_logic_vector(7 downto 0) := (others => '0');
signal waddress   : std_logic_vector(7 downto 0) := (others => '0');
signal write1     : std_logic := '0';
signal data_out   : std_logic_vector(7 downto 0) := (others => '0');

component ram256x8
                port(
                        clock           : IN STD_LOGIC  := '1';
                        data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        rdaddress               : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        wraddress               : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        wren            : IN STD_LOGIC  := '0';
                        q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
                );
end component;
--============================================================================
```

Graças a essa implementação, foi possível armazenar os dados da memória em um arquivo de extensão “.mif” e inicializar a mesma com valores prévios. Dessa forma, é possível inicializar a memória já com um programa a ser operado.

3 - Conclusão

Por meio da atividade prática, foi possível desenvolver habilidades de criação de código em VHDL, além do entendimento do funcionamento de um processador, analisando cada um dos componentes e implementando suas instruções.

Com isso, foi possível obter uma maior familiaridade com os conceitos de registradores, tamanho das instruções, fluxo de informações e sinais de controle. Além disso, o projeto proporcionou uma visão prática sobre como módulos, como a memória RAM (implementada como ram256x8), se integram ao processador para manipular dados. A implementação das instruções permitiu compreender como operações aritméticas, lógicas e de controle de fluxo são traduzidas em sinais digitais e executadas de forma sequencial.

O desenvolvimento desse processador também destacou a importância de planejar e organizar sinais e componentes, garantindo a clareza no fluxo de dados e a funcionalidade correta da arquitetura. Finalmente, a atividade reforçou o

aprendizado sobre o uso de sinais de controle como FRzero, FRmaior, e FRover para determinar o comportamento do sistema em diferentes cenários operacionais.

O repositório pode ser acessado por meio do link https://github.com/MaiconChavesMarques/SSC0108-Pratica-em-Sistemas-Digitais-2024/tree/main/Processador.