PCS3225 - Sistemas Digitais II
Atividade Formativa 11 - Projeto do Processador PoliLEGv8 em VHDL
Parte 1 - Memórias do PoliLEGv8

Sergio Roberto de Mello Canovas; revisado por Glauber de Bona, Edson Toshimi Midorikawa (2023) e Antonio Vieira da Silva Neto (2024)

Data do Arquivo: 08/11/2024; Prazo da AF11: 23/11/2024

Introdução

Antes mesmo de os primeiros computadores serem construídos, eles eram idealizados teoricamente por matemáticos e cientistas. Uma **arquitetura de computador** descreve a funcionalidade, a organização e a implementação de sistemas de computadores. A **arquitetura de von Neumann** consiste em uma unidade central de processamento, ou CPU (*Central Processing Unit*), conectada a uma única memória. A CPU, por sua vez, consiste no conjunto formado principalmente pela Unidade de controle e ULA (Unidade Lógica Aritmética), como pode ser observado na Figura 1.

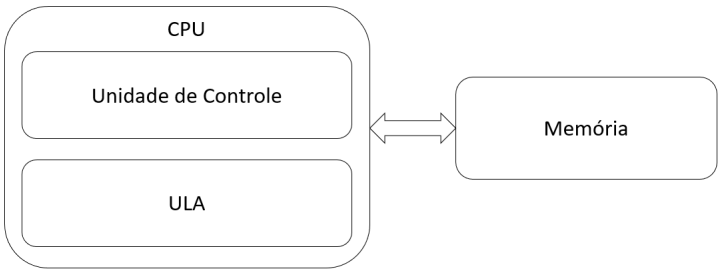

Figura 1: Arquitetura de von Neumann

Usualmente, os termos CPU e processador são usados intercambiavelmente. Mas, para ser rigoroso, uma CPU é um processador de propósito geral, e existem tipos específicos de processadores com outras denominações, tais como a GPU (*Graphics Processing Unit*), que se trata de um processador especializado na execução de instruções relacionadas a cálculos gráficos.

Na arquitetura de von Neumann, a memória armazena tanto instruções de máquina quanto dados de trabalho. Uma instrução de máquina é uma sequência de bits que é interpretada pelo processador, provocando a execução de alguma operação. A CPU executa um **programa** (sequência de instruções) previamente carregado na memória e, para isso, deve obter cada instrução, uma por vez, por meio de uma operação de leitura da memória que ocorre por meio do barramento que a conecta. Os programas também produzem e manipulam dados de trabalho, ou simplesmente dados - a exemplo do resultado de uma soma -, que precisa ser armazenado para posterior uso. Esses dados são armazenados nessa mesma memória mediante operações de escrita em outros endereços que não se misturam com

os endereços usados para o armazenamento das instruções. Entretanto, para um observador externo que hipoteticamente possa consultar o conteúdo de cada posição da memória e que não saiba em quais endereços o programa foi carregado, não é possível distinguir instruções de dados, uma vez que ambos consistem em sequências de 0s e 1s que não possuem identificação autocontida.

Os computadores que conhecemos hoje em dia são baseados na arquitetura de von Neumann com diversas extensões, tais como a inclusão de memórias cache na CPU, *pipelining*, interrupções e outros elementos.

O processador PoliLEGv8, a ser implementado e simulado em VHDL na Atividade Formativa 11 de PCS3225, é um processador monociclo. Isso significa que ele é projetado para executar uma única instrução em um único ciclo de *clock*, seguindo essa regra para toda e qualquer instrução de seu conjunto de instruções. Para atingir esse objetivo com uma implementação simples, é necessário usar uma arquitetura diferente da de von Neumann, que é conhecida como arquitetura de Harvard.

Na arquitetura de Harvard, as memórias de instruções e de dados são fisicamente separadas, possuindo também sinais de acesso independentes. Isso significa que a CPU consegue acessar a memória de instruções e a memória de dados simultaneamente. Tipicamente, a memória de instruções é somente leitura (ROM - *Read-Only Memory*) e a memória de dados é de leitura e escrita (RAM - *Random Access Memory*). A Figura 2 ilustra esse cenário.

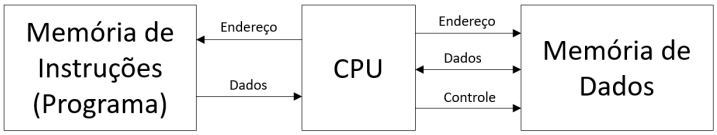

Figura 2: Arquitetura Harvard

No PoliLEGv8, adota-se a arquitetura de Harvard com duas memórias separadas: a memória de instruções (a ser implementada com uma ROM) e a memória de dados (a ser implementada com uma RAM).

Nesta parte da Atividade Formativa 11, o foco será a implementação das descrições de hardware das memórias de instruções e de dados do PoliLEGv8. Para tal, devem ser desenvolvidas duas entidades: uma entidade de memória ROM e outra de memória RAM em VHDL, juntamente com as arquiteturas que implementarão seus respectivos comportamentos. No caso da memória ROM, existem algumas variações de implementação (múltiplas arquiteturas).

Memórias em VHDL

Em VHDL, pode-se criar um tipo para armazenamento de dados correspondente a um vetor cujo tipo do elemento modela cada palavra a ser armazenada. Veja o exemplo abaixo com a definição de mem_tipo:

```
type mem_tipo is array(0 to 255) of bit_vector(7 downto 0);
```

O tipo declarado como mem_tipo corresponde a um vetor de 256 posições indexadas de 0 a 255, e o tipo de cada elemento é um vetor de bits (bit_vector(7 **downto** 0)), estabelecendo que cada palavra tem tamanho de 8 bits. Com relação a uma memória implementada com base nesse tipo, diz-se que ela tem profundidade de 256 palavras e largura de 8 bits. Uma instância deste tipo pode, então, ser declarada como um **signal**:

```
signal mem: mem_tipo;
```

Esse **signal**, que corresponde a um vetor, pode ser inicializado em sua própria declaração. A seguir, apresenta-se um exemplo para o caso de um vetor de 4 posições, supondo que mem_tipo tivesse sido declarado com esse tamanho:

```
signal mem: mem_tipo := ("01010101", "10101010", "00001111", "11110000");
```

Para acessar uma posição específica do vetor, basta indexar o **signal** com um inteiro entre parênteses, o qual corresponde à posição do vetor a ser acessada. No exemplo subsequente, a terceira posição de mem é atribuída ao **signal** chamado data, o qual também é um bit_vector de tamanho 8 (assim como cada elemento do vetor mem).

```
data <= mem(2);
```

Dica: Em VHDL, para converter um bit_vector para um inteiro, inclua a biblioteca ieee.numeric_bit e utilize a seguinte expressão, na qual bv é um bit_vector:

```
to_integer(unsigned(bv))
```

Atividades

IMPORTANTE 1: A seguir, são fornecidos exemplos de código. Caso se decida aproveitá-los em seu projeto, **não** copie e cole diretamente deste arquivo. Foi observado que o código copiado deste PDF carrega caracteres de controle ocultos, relativos ao formato de arquivo PDF, que geram erros de compilação quando copiados para um arquivo VHDL. Por essa razão, redigite o código necessário no arquivo VHDL.

IMPORTANTE 2: O objetivo final desta parte da Atividade Formativa 11 é a realização do exercício AF11-P1E5, que determina a construção, em VHDL, das memórias de instruções e dados do processador PoliLEGv8 e das bancadas de testes (*testbenches*) que comprovam seus respectivos funcionamentos. Os exercícios AF11-P1E1 até AF11-P1E4 servem como um guia de projeto iterativo rumo aos objetivos do exercício AF11-P1E5. Dessa forma, embora a realização dos exercícios AF11-P1E1 até AF11-P1E4 não seja obrigatória, é recomendável seguir o roteiro proposto para facilitar o processo de desenvolvimento da AF11-P1E5.

Atividade Formativa 11 Parte 1, Exercício 1

AF11-P1E1 Implemente um componente em VHDL correspondente a uma memória ROM que respeite a seguinte entidade:

```
entity rom_simples is
  port (
    addr : in  bit_vector(4 downto 0);
    data : out bit_vector(7 downto 0)
  );
end entity rom_simples;
```

Esta é uma ROM convencional que não permite escrita. Portanto, ela só possui uma entrada e uma saída:

- addr: Endereço;
- data: Conteúdo armazenado correspondente ao endereço addr.

Perceba, pelo código da entidade, que tal ROM possui 5 bits de endereço (suportando 32 posições de armazenamento, endereçadas de 0 a 31) e 8 bits de tamanho de palavra. O conteúdo da ROM deve ser inicializado diretamente no código VHDL da porção **architecture** com os seguintes dados:

Endereço (dec)	Conteúdo (bin)
0	00000000
1	00000011
2	11000000
3	00001100
4	00110000
5	01010101
6	10101010
7	11111111
8	11100000
9	11100111
10	00000111
11	00011000
12	11000011
13	00111100
14	11110000
15	00001111
16	11101101
17	10001010
18	00100100
19	01010101
20	01001100
21	01000100
22	01110011
23	01011101
24	11100101
25	01111001
26	01010000
27	01000011
28	01010011
29	10110000
30	11011110
31	00110001

Não deixe de elaborar seu próprio *testbench* para testar e validar sua ROM.

Tal como orientado no "Material Complementar"do "Projeto do Processador PoliLEGv8"disponível no e-Disciplinas, recomenda-se consultar o material do Prof. Bruno Albertini, do PCS/Poli-USP (disponível em `https://balbertini.github.io/vhdl_mem-pt_BR.html`), para mais informações sobre descrição de memórias em VHDL e os respectivos *testbenches*.

Atividade Formativa 11 Parte 1, Exercício 2

AF11-P1E2 A iniciação da ROM diretamente pelo código-fonte em descrições VHDL não é prática. Uma alternativa a essa abordagem consistem em descrever a memória de modo que **os dados para sua iniciação sejam carregados a partir um arquivo separado**. Esse arquivo poderia, então, ser gerado por outra ferramenta, facilitando a

composição de projetos por meio da separação entre a descrição da memória e os dados de seu conteúdo. Um exemplo de aplicação seria a utilização de um compilador, que converte um programa em linguagem de alto nível em um arquivo com instruções de máquina em binário. O conteúdo gerado pelo compilador em um arquivo próprio poderia ser usado para inicializar uma memória descrita em VHDL - como, por exemplo, um cartucho de *videogame*, que consiste em uma memória ROM contendo o programa correspondente ao jogo, já em código de máquina.

Dado o contexto prévio, implemente um componente em VHDL correspondente a uma memória ROM que respeite a **mesma entidade** do item anterior, mas trocando seu nome para `rom_arquivo`, ou seja:

```
entity rom_arquivo is
  port (
    addr : in  bit_vector(4 downto 0);
    data : out bit_vector(7 downto 0)
  );
end entity rom_arquivo;
```

A diferença é que, nessa entidade, o conteúdo deve ser carregado na iniciação a partir de um arquivo. Isso pode ser feito em VHDL por meio de uma função de iniciação de memória. Na prática, a iniciação do **signal** do vetor da memória passa a fazer uma chamada a uma função que faz a leitura de um arquivo, em vez de iniciar os valores no próprio código VHDL. No exemplo subsequente, a iniciação de mem é feita por meio de uma chamada à função init_mem. O parâmetro "conteudo_rom_af11_p1e2_carga.dat" indica o nome do arquivo a ser lido.

```
signal mem: mem_tipo := init_mem("conteudo_rom_af11_p1e2_carga.dat");
```

Uma referência sobre como codificar a função init_mem pode ser encontrada em `https://balbertini.github.io/vhdl_mem-pt_BR.html`. Veja a seção **Arquivo**. O exemplo fornecido é capaz de ler arquivos DAT que, em essência, são arquivos-texto comuns em que cada linha contém o conteúdo de uma palavra, em binário, na ordem dos endereços. Por exemplo, o conteúdo mostrado no Exemplo 1 (ver próxima página) é de um arquivo DAT usado para uma iniciação com o mesmo conteúdo da memória solicitada no exercício AF11-P1E1.

Neste exercicio, **obrigatoriamente** o nome do arquivo DAT que sua implementação deve utilizar é `conteudo_rom_af11_p1e2_carga.dat`

Dica: Em PCS3225, já se estudou como realizar leituras de arquivos utilizando VHDL para especificar casos de teste. A leitura de arquivos para iniciar memórias é similar. Você deve usar a biblioteca std.textio.

Exemplo 1: conteúdo para o arquivo DAT

```
00000000
00000011
11000000
00001100
00110000
01010101
10101010
11111111
11100000
11100111
00000111
00011000
11000011
00111100
11110000
00001111
11101101
10001010
00100100
01010101
01001100
01000100
01110011
01011101
11100101
01111001
01010000
01000011
01010011
10110000
11011110
00110001
```

Atividade Formativa 11 Parte 1, Exercício 3

AF11-P1E3 Em diferentes projetos de *hardware*, ou até mesmo no mesmo projeto, pode-se precisar de memórias com diferentes tamanhos de palavras e números de bits de endereço. Se fosse necessário criar uma nova descrição VHDL a cada nova especificação de memória, copiando da anterior e alterando os números de bits referidos, tanto o esforço gasto seria maior quanto o resultado seria mais suscetível a erros. Para evitar essa situação e possuir uma única implementação configurável de uma memória, pode-se explorar o recurso **generic** do VHDL, que pode ser usado não somente para memórias mas para qualquer componente.

Pesquise sobre o uso do recurso **generic** em VHDL. Uma referência pode ser encontrada em `https://balbertini.github.io/vhdl_generic-pt_BR.html`, que apresenta a criação e uso de um registrador de deslocamento descrito com base nessa palavra-chave.

Em resumo, o recurso **generic** em VHDL permite parametrizar dados de uma entidade. Na especificação da entidade, é possível definir valores padrão para cada parâmetro. Quando a entidade é instanciada em um componente, é possível atribuir um valor para cada parâmetro ou usar o valor padrão - caso em que o parâmetro pode ser omitido na instanciação do componente.

Após entender o uso de **generic**, recrie a memória ROM do item AF11-P1E2, desta vez respeitando a seguinte entidade:

```
entity rom_arquivo_generica is
  generic (
```

```
    addressSize : natural := 5;
    wordSize    : natural := 8;
    datFileName : string  := "conteudo_rom_af11_p1e2_carga.dat"
  );
  port (
    addr : in  bit_vector(addressSize-1 downto 0);
    data : out bit_vector(wordSize-1 downto 0)
  );
end entity rom_arquivo_generica;
```

Embora os valores padrão dos parâmetros genéricos sejam iguais aos da atividade anterior, o PoliLEGv8 poderá instanciar sua memória ROM usando diferentes números de bits de endereço, tamanhos de palavra, e até mesmo distintos nomes do arquivo DAT. Por isso, recomenda-se que seu *testbench* preveja casos de testes com instanciação de memórias de parâmetros variados, permitindo testar cenários diversos (lembre-se do conceito de cobertura de *testbench*).

É evidente que o conteúdo de cada arquivo DAT utilizado deve estar condizente com os parâmetros da memória instanciada. Por exemplo, se for instanciada uma memória de 6 bits de endereço, seu arquivo DAT deve passar a ter 64 linhas de dados, e não mais 32. Idem para o tamanho da palavra: se for instanciada uma memória com palavra de 32 bits, cada linha do arquivo deve apresentar uma sequência de 32 bits, e não mais 8.

Atividade Formativa 11 Projeto 1, Exercício 4

AF11-P1E4 Implemente um componente em VHDL correspondente a uma memória RAM com escrita síncrona que respeite a seguinte entidade:

```
entity ram_generica is
  generic (
    addressSize : natural := 5;
    wordSize    : natural := 8;
    datFileName : string  := "conteudo_ram_af11_p1e4_carga.dat"
  );
  port (
    clk    : in  bit;
    wr     : in  bit;
    addr   : in  bit_vector(addressSize-1 downto 0);
    data_i : in  bit_vector(wordSize-1 downto 0);
    data_o : out bit_vector(wordSize-1 downto 0)
  );
end entity ram_generica;
```

A escrita síncrona significa que o dado colocado em data_i deve ser escrito na memória na ocorrência de uma borda de subida do sinal de *clock* quando o sinal de escrita estiver ativo. Como o PoliLEGv8 trabalha com lógica positiva, considere que wr é ativo em nível alto. A leitura da RAM, por sua vez, deve ocorrer de forma **assíncrona**, isto é, sem depender de uma borda de subida do *clock*. Para tanto, basta alterar o valor da entrada addr para o sinal data_o ser atualizado com o conteúdo armazenado na posição addr. Observe que o número de bits do barramento de endereço e o tamanho da palavra devem ser implementados por meio de generics, ou seja, sua memó-

ria poderá ser instanciada em um projeto com qualquer tamanho de barramento de endereço (não necessariamente 5 bits) e qualquer tamanho de palavra de dados (não necessariamente 8 bits), assim como no exercício anterior. A lista completa dos sinais da RAM é a seguinte:

- clk: *Clock*;
- wr: Sinal de escrita. Quando estiver em ALTO (1) e ocorrer uma borda de subida em clk, o conteúdo de data_i deve ser escrito na posição da memória determinada por addr;
- addr: Endereço;
- data_i: Conteúdo de entrada para escrever na memória com o uso do sinal wr;
- data_o: Conteúdo lido da memória. Deve corresponder sempre ao valor que está na palavra indexada por addr.

O nome do arquivo DAT com o conteúdo inicial da memória que sua implementação deve utilizar é `conteudo_ram_af11_p1e4_carga.dat`.

P1A5 Implemente os componentes em VHDL correspondentes à Memória de Instruções e à Memória de Dados do PoliLEGv8. A Memória de Instruções deve ter 256 endereços com palavras de 32 bits, ao passo que a Memória de Dados deve ter 256 endereços com palavras de 64 bits. Cada componente deve respeitar as entidades a seguir.

Projeto 1, Atividade 5

```
entity memoriaInstrucoes is
  generic (
    datFileName : string  := "conteudo_memInstr_af11_p1e5_carga.dat"
  );
  port (
    addr : in  bit_vector(7 downto 0);
    data : out bit_vector(31 downto 0)
  );
end entity memoriaInstrucoes;

entity memoriaDados is
  generic (
    datFileName : string  := "conteudo_memDados_af11_p1e5_carga.dat"
  );
  port (
    clk    : in  bit;
    wr     : in  bit;
    addr   : in  bit_vector(7 downto 0);
    data_i : in  bit_vector(63 downto 0);
    data_o : out bit_vector(63 downto 0)
  );
end entity memoriaDados;
```

Projete os respectivos *testbenches* de cada memória, execute simulações para verificar o funcionamento das memórias e **documente a análise dos resultados das simulações**.

Nota: A conexão de ambas as memórias do PoliLEGv8 aos demais componentes do projeto será detalhada nas atividades posteriores.

Instruções para os Grupos

Como boa prática de projeto de Engenharia, faça seus *testbenches* e utilize o GHDL/GtkWave ou *EDA Playground* (selecionando o GHDL como simulador) para validar suas soluções.

Use as técnicas apresentadas no módulo de **Verificação de Projetos** para acrescentar formas de verificação de funcionamento para validar os projetos das diversas atividades do projeto (p.ex. uso dos comandos `assert` e `report`).