Sistemas Digitais II 20/09/2021

PCS3225 - Sistemas Digitais II
Projeto 2 - Memórias em VHDL

Versão 1 Prof. Sergio R. M. Canovas

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 [1]. 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 pela unidade de controle e ULA (Unidade Lógica Aritmética), como pode ser observado na Figura 1.

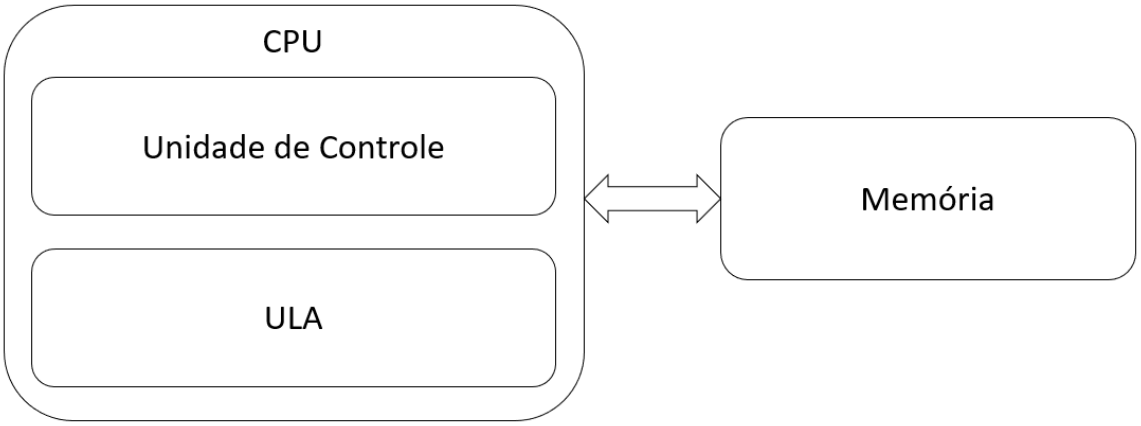

Figura 1: Arquitetura de von Neumann

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 através do barramento que a conecta. Mas os programas produzem e manipulam dados de trabalho, ou

simplesmente dados, como por exemplo o resultado de uma soma, que precisa ser armazenado para posterior uso. Esses dados são armazenados nessa mesma memória, através de 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 sabe 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 registradores na CPU, interrupções, e outras.

O processador PoliLEG, que implementaremos e simularemos em VHDL ao longo deste oferecimento da disciplina, 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 todas as instruções que suporta. Para conseguir este feito e manter sua implementação simples (detalhes serão vistos mais adiante no curso), é necessário usar uma arquitetura diferente da de von Neumann, que é conhecida como arquitetura Harvard.

Na arquitetura Harvard, o armazenamento de instruções e dados são fisicamente separados [2], 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 e a memória de dados é de leitura e escrita. A Figura 2 ilustra esse cenário.

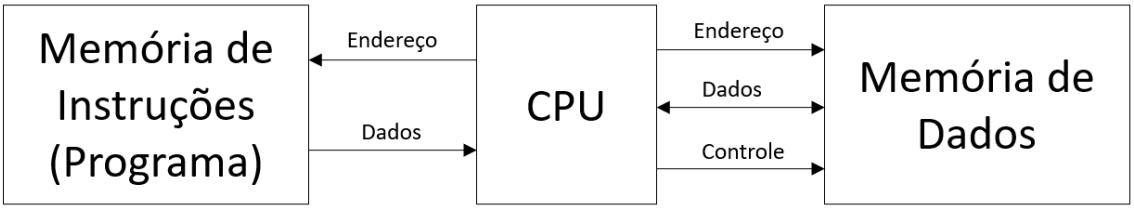

Figura 2: Arquitetura Harvard

No PoliLEG, seguiremos essa arquitetura e trabalharemos com duas memórias separadas: a memória de instruções (para a qual usaremos uma ROM) e a memória de dados (para a qual usaremos uma RAM).

Neste trabalho, o foco será a implementação de uma entidade de memória ROM e outra de memória RAM em VHDL, juntamente com suas arquiteturas que implementarão seus comportamentos. No caso da memória ROM, serão pedidas algumas variações de implementação.

Em trabalhos futuros, essas memórias serão instanciadas e utilizadas no contexto do PoliLEG.

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 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, dizemos 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. Abaixo, 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 abaixo, 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, onde bv é um bit_vector:

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

Atividades

IMPORTANTE: Abaixo, são fornecidos exemplos de código. Caso decida aproveitá-los em seu projeto, **não** copie e cole 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 esta razão, redigite o código necessário no arquivo VHDL.

P2A1 (1 ponto) 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 rom_simples;
```

Vimos em aula que existem tipos de ROM que, apesar do nome, também permitem a escrita de dados, embora seja uma operação mais complicada e menos frequente que a leitura. Porém, esta é uma ROM convencional que não permite escrita, e por isso ela só possui uma entrada e uma saída:

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

Perceba, pelo código da entidade acima, que esta 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 desta ROM deve ser inicializado diretamente no código VHDL da **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

Endereço (dec)	Conteúdo (bin)
23	01011101
24	11100101
25	01111001
26	01010000
27	01000011
28	01010011
29	10110000
30	11011110
31	00110001

P2A2 (3 pontos) A inicialização da ROM diretamente pelo código-fonte em descrições VHDL não é prática. Uma alternativa seria descrever a memória de modo que os dados para sua inicialização fossem carregados de um arquivo separado. Este arquivo poderia então ser gerado por outra ferramenta, facilitando a composição de projetos através da separação entre descrição da memória e dados de conteúdo. Um exemplo de aplicação seria a utilização de um compilador, que converte um programa em linguagem de alto nível para um arquivo com instruções de máquina em binário. Este conteúdo, gerado pelo compilador em um arquivo próprio, poderia ser usado para inicializar uma memória descrita em VHDL (como um cartucho de *videogame*, que consiste em uma memória ROM contendo o programa correspondente ao jogo, já em código de máquina).

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 rom_arquivo;
```

A diferença é que nessa atividade o conteúdo deve ser carregado na inicialização a partir de um arquivo. Isso pode ser feito em VHDL por meio de uma função de inicialização de memória. Na prática, a inicialização do **signal** do vetor da memória passa a fazer uma chamada a uma função que faz a leitura de um arquivo, em vez de inicializar os valores no próprio código VHDL. No exemplo abaixo, a inicialização de mem é feita por meio de uma chamada à função init_mem. O parâmetro "conteudo_rom_ativ_02_carga.dat" indica o nome do arquivo a ser lido.

```
signal mem: mem_tipo := init_mem("conteudo_rom_ativ_02_carga.dat");
```

Uma referência sobre como codificar a função init_mem pode ser encontrada em `http://myfpgablog.blogspot.com/2011/12/memory-initialization-methods.html`. Veja a seção **VHDL with external data files**. 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 na ordem dos endereços, em binário. Por exemplo, o conteúdo mostrado a seguir é o conteúdo de um arquivo DAT usado para uma inicialização com o mesmo conteúdo da memória solicitada

na atividade P2A1:

```
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
```

Nesta atividade, **obrigatoriamente** o nome do arquivo DAT que sua implementação deve utilizar é `conteudo_rom_ativ_02_carga.dat`. Isso é necessário para que o juiz eletrônico funcione corretamente. Esse arquivo DAT **não** deve ser submetido ao juiz. Submeta apenas seu arquivo VHDL.

Dica: No curso, já vimos como realizar leituras em arquivos, utilizando VHDL, para especificar casos de teste. A leitura de arquivos para inicializar memórias é similar. Você deve usar a biblioteca std.textio.

Dica 2: O código fornecido pelo *link* acima trabalha com tipos std_logic, enquanto aqui trabalhamos com tipos bit. Portanto, faça as adaptações necessárias.

P2A3 (3 pontos) Em diferentes projetos de *hardware*, ou às vezes no mesmo projeto, pode-

mos precisar de memórias com diferentes tamanhos de palavra e números de bits de endereço. Se tivéssemos que criar uma nova descrição VHDL a cada nova especificação de memória, copiando da anterior e alterando os números de bits referidos, teríamos muito trabalho, além de ser suscetível a erros. Para isso, podemos 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://vhdlwhiz.com/constants-generic-map/`, que apresenta a criação e uso de um multiplexador descrito com base nessa palavra-chave. Após entender o uso de **generic**, recrie a memória ROM do item P2A2, desta vez respeitando a seguinte entidade:

```
entity rom_arquivo_generica is
  generic (
    addressSize : natural := 5;
    wordSize    : natural := 8;
    datFileName : string  := "conteudo_rom_ativ_02_carga.dat"
  );
  port (
    addr : in  bit_vector(addressSize-1 downto 0);
    data : out bit_vector(wordSize-1 downto 0)
  );
end rom_arquivo_generica;
```

Embora os valores padrões dos parâmetros genéricos sejam iguais aos da atividade anterior, o *test-bench* do juiz eletrônico 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 *test-bench* preveja casos de testes com instanciação de memórias de parâmetros variados, permitindo testar cenários diversos (lembre-se do conceito de cobertura de *test-bench*).

É claro que o conteúdo de cada arquivo DAT utilizado deve estar condizente com os parâmetros da memória instanciada. Por exemplo, se instanciarmos uma memória de 6 bits de endereço, seu arquivo DAT deve passar a ter 64 linhas de dados, não mais 32. Idem para o tamanho da palavra: se instanciarmos uma memória com palavra de 32 bits, cada linha do arquivo deve apresentar uma sequência de 32 bits, e não mais 8. O *test-bench* do juiz eletrônico utilizará arquivos apropriados internamente em cada teste. Considere isso também no seu próprio *test-bench*.

Novamente, não submeta o arquivo DAT. Submeta apenas o arquivo VHDL ao juiz eletrônico.

P2A4 (3 pontos) Implemente um componente em VHDL correspondente a uma memória RAM com escrita síncrona que respeite a seguinte entidade:

```
entity ram is
  generic (
    addressSize : natural := 5;
    wordSize    : natural := 8
  );
```

```
  port (
    ck, 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 ram;
```

A escrita síncrona significa que o dado colocado em data_i deve ser escrito na memória na ocorrência de uma borda de subida do *clock* quando o sinal de escrita estiver ativo. Considere que wr é ativo alto. A leitura deve ocorrer de forma assíncrona, isto é, sem depender de uma borda de subida do *clock*. Basta alterar o valor da entrada addr e o sinal data_o será 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 memória 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 na atividade anterior. A lista completa dos sinais é a seguinte:

- ck: *Clock*;
- wr: Sinal de escrita. Quando estiver em ALTO e ocorrer uma borda de subida em ck, 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.

Instruções para Entrega

Você deve acessar o link específico para cada tarefa (P2A1, P2A2, P2A3 e P2A4) no E-Disciplinas, já logado com seu usuário e senha, que levará à página apropriada do juiz eletrônico. O prazo para a submissão das soluções no Juiz é 06 de outubro de 2021, quarta-feira, às 23:59. O juiz aceitará até 5 submissões para cada atividade deste projeto. Sua submissão será corrigida imediatamente e sua nota será apresentada. A maior nota dentre as submissões será considerada. Neste trabalho, os problemas valem no máximo 10 pontos no juiz, porém a nota final deste trabalho será calculada com as ponderações indicadas em cada atividade neste enunciado, totalizando 10 para o trabalho todo. Como boa prática de engenharia, faça seus *test-benches* e utilize o *EDA Playground* (selecionando o GHDL como simulador) para validar suas soluções antes de postá-las no juiz.

Referências

[1] I. Zhirkov. *Programação em Baixo Nível*. Novatec, 2018.

[2] Harvard vs von neumann architectures. https://developer.arm.com/documentation/ka002816/latest.