PCS3225 - Sistemas Digitais II
Atividade Formativa 12 - Projeto do Processador PoliLEGv8 em VHDL
Parte 1 - Biblioteca de Componentes Básicos do PoliLEGv8

Antonio Vieira da Silva Neto
(Agradecimentos de Anos Anteriores: Edson Toshimi Midorikawa, Glauber de Bona e Sergio Roberto de Mello Canovas)

Data do Arquivo: 30/10/2025; Prazo da AF12: 13/12/2025

Introdução

A Atividade Formativa 12 (AF12) de PCS3225 baseia-se na implementação e na simulação do processador PoliLEGv8 em VHDL. Ao se utilizar um método estruturado de projeto de Engenharia, aderente a um ciclo de vida em V, parte-se de uma **especificação** sobre as características do processador e de sua **arquitetura** - ambas vistas em aula - para projetá-lo.
A primeira fase do projeto, especificada neste enunciado, tem a finalidade de construir e validar o funcionamento dos componentes elementares que integram o Fluxo de Dados do processador PoliLEGv8. Por meio do diagrama da Figura 1, que ilustra a versão monociclo do processador, é possível identificar a necessidade dos seguintes componentes elementares:

1. Registrador de 'n' bits com carga paralela (para banco de registradores e *Program Counter - PC*;
2. Multiplexador com duas entradas de 'n' bits;
3. ROM (*Read-Only Memory* para o armazenamento do programa;
4. RAM *Reandom Access Memory* para o armazenamento de dados;
5. Somador binário de 'n' bits;
6. Unidade Lógica e Aritmética (ULA) de 1 bit que realiza pelo menos cinco operações: soma, subtração, E lógico (bit a bit), OU lógico (bit a bit) e passagem direta do segundo operando (*pass B*);
7. Extensor de sinal com entrada de 'n' bits e indicação do k-ésimo bit válido ($k \leq n$);
8. Deslocador assíncrono de dois bits para a esquerda.

Na sequência, alguns componentes serão integrados para construir dois módulos maiores do Fluxo de Dados do processador PoliLEGv8: o **banco de registradores** e a **ULA de 64 bits**. Esses blocos maiores **serão especificados na próxima parte do projeto.**
O objetivo desta parte da Atividade Formativa 12 é projetar os seis itens prévios em VHDL. Para tanto, deve-se atender às seguintes determinações:

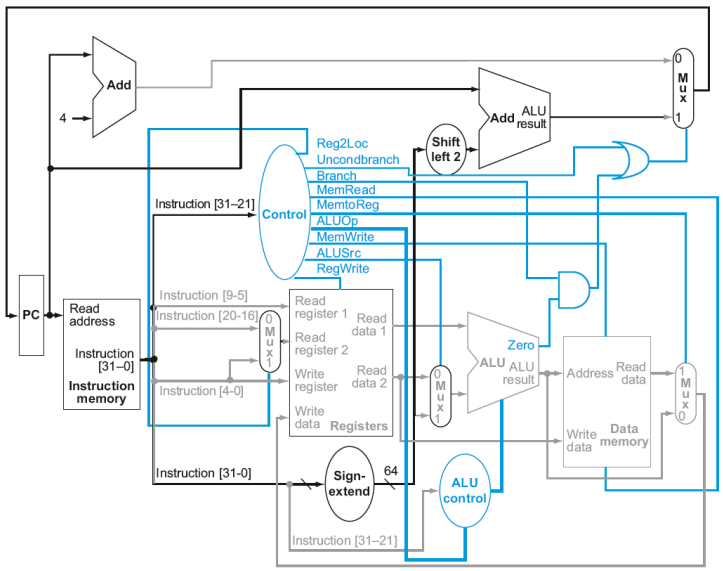

Figura 1: Processador PoliLEGv8 - Versão Monociclo

1. As interfaces de entrada e saída, as características funcionais e eventuais técnicas de projeto indicadas no enunciado devem ser respeitadas;

2. Para cada componente, deve-se especificar um plano de testes que evidencie cobertura adequada de funcionamento e projetar uma bancada de testes que exercite os casos de teste previstos;

3. O projeto e a verificação funcional de cada componente devem ser documentados em um relato cumulativo a ser produzido no projeto. Nesse relato, recomenda-se seguir a estrutura subsequente: criar um capítulo para cada componente e dividi-lo em quatro partes: (i.) descrição funcional do projeto (interfaces e lógica interna), (ii.) descrição do plano de testes, (iii.) descrição da bancada de testes e (iv.) apresentação e análise de conformidade dos resultados obtidos. Cada capítulo não deve ocupar mais do que três páginas do relato.

Componente 1 - Registrador

Implemente e verifique o funcionamento de um componente em VHDL correspondente a um registrador parametrizável que respeite a seguinte entidade:

```
entity reg is
    generic(dataSize: natural :=64);
    port(
        clock   : in  bit;                                  -- entrada de clock
        reset   : in  bit;                                  -- clear assincrono
        enable  : in  bit;                                  -- write enable (carga paralela)
        d       : in  bit_vector(dataSize-1 downto 0); -- entrada
        q       : out bit_vector(dataSize-1 downto 0)  -- saida
    );
end entity reg;
```

O registrador é parametrizável por meio do parâmetro `dataSize`, que determina sua largura de dados em bits. Ele recebe um sinal de *clock* periódico e é sensível à borda de subida desse sinal. O sinal `reset` é assíncrono e força todos os bits do registrador para zero. Quando o sinal `enable` é alto, a escrita é considerada habilitada, e o valor na entrada `d` é gravada quando ocorrer uma borda de subida do *clock*, refletindo-se, então, na saída `q`. Contrariamente, quando o sinal `enable` é baixo, nada acontece e a saída permanece inalterada. A saída `q` é assíncrona e mostra o conteúdo atual do registrador.
Dica: Este registrador serve como referência para a implementação do *Program Counter* (PC) e para os elementos que integram o banco de registradores do processador PoliLEGv8.

Para o PoliLEGv8, `dataSize = 64` porque os registradores permitem o armazenamento de 8 bytes (64 bits).

Tanto a entrada `d` quando a saída `q` são paralelas e contém `dataSize` bits.

Componente 2 - Multiplexador

Implemente e verifique o funcionamento de um componente em VHDL correspondente a um multiplexador de duas entradas de tamanho parametrizável que respeite a seguinte entidade:

```
entity mux_n is
    generic(dataSize: natural := 64);
    port(
        ino     : in  bit_vector(dataSize-1 downto 0); -- entrada de dados 0
        in1     : in  bit_vector(dataSize-1 downto 0); -- entrada de dados 1
        sel     : in  bit;                             -- sinal de selecao
        dOut    : out bit_vector(dataSize-1 downto 0)  -- saida de dados
    );
end entity mux_n;
```

O multiplexador tem largura de dados configurável por meio do parâmetro `dataSize`. Quando o sinal *sel* é igual a zero, a saída *dOut* deve mostrar em sua saída a entrada *ino*; caso contrário, a saída *dOut* deve ser *in1*.

Componente 3 - Memória de Instruções

Implemente e verifique o funcionamento de um componente em VHDL correspondente à ROM de Instruções do PoliLEGv8. Ela deve ser aderente à entidade a seguir.

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

A memória possui um barramento de endereços denominado *addr*. Ao se escolher um endereço atribuindo um valor a esse barramento,

o conteúdo da posição de memória correspondente é disponibilizado no barramento de dados, denominado *data*. Para tornar o projeto mais flexível, os tamanhos dos barramentos de endereços e de dados são configuráveis e respectivamente determinados pelos parâmetros *addressSize* e *dataSize*.
Ademais, o parâmetro configurável *datFileName* deve ser definido na instanciação do componente e serve para apontar para um arquivo externo que possui o conteúdo da memória.

Para o PoliLEGv8, `dataSize = 8` porque cada posição de memória tem um único byte. Para ler instruções de 4 bytes, devem ser realizadas leituras de quatro posições consecutivas da memória.

Componente 4 - Memória de Dados

Implemente e verifique o funcionamento de um componente em VHDL correspondente à RAM de Dados do PoliLEGv8. Ela deve ser aderente à entidade a seguir, cujas interfaces são descritas na sequência:

```
entity memoriaDados is
  generic (
    addressSize : natural := 8;
    dataSize    : natural := 8;
    datFileName : string  := "memDados_conteudo_inicial.dat"
  );
  port (
    clock  : in  bit;
    wr     : in  bit;
    addr   : in  bit_vector(addressSize-1 downto 0);
    data_i : in  bit_vector(dataSize-1 downto 0);
    data_o : out bit_vector(dataSize-1 downto 0)
  );
end entity memoriaDados;
```

- clock: *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 no endereço addr 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.

A memória utiliza um sinal de *clock* para viabilizar a síntese da RAM em um FPGA. A escrita síncrona significa que o dado colocado em data_i deve ser escrito na posição de memória apontada no barramento de endereços `addr` 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.
Para tornar o projeto mais flexível, os tamanhos dos barramentos de endereços e de dados são configuráveis e respectivamente determinados pelos parâmetros *addressSize* e *dataSize*.
Ademais, o parâmetro configurável *datFileName* deve ser definido na instanciação do componente e serve para apontar para um arquivo externo que possui o conteúdo inicial da memória.

Para o PoliLEGv8, `dataSize = 8` porque cada posição de memória tem um único byte. Para manipular dados de 8 bytes, devem ser realizadas leituras ou escritas de oito posições consecutivas da memória.

Componente 5 - Somador Binário

Implemente e verifique o funcionamento de um componente em VHDL correspondente a um somador binário de duas entradas de 'n' bits que respeite a seguinte entidade:

```
entity adder_n is
    generic(dataSize: natural := 64);
    port(
        ino     : in  bit_vector(dataSize-1 downto 0); -- primeira parcela
        in1     : in  bit_vector(dataSize-1 downto 0); -- segunda parcela
        sum     : out bit_vector(dataSize-1 downto 0); -- soma
        cOut    : out bit
    );
end entity adder_n;
```

O somador não tem entrada 'vem-um' (*carry in*), que deve ser assumido como igual a zero. O componente soma dois números, representados por *ino* e *in1*, e produz duas saídas: *sum* com o resultado da soma, e *cOut* com eventual vai-um resultante da soma das duas parcelas. Note que todos os dados têm largura configurável por meio do parâmetro `dataSize` a menos da saída *cOut*, que é um único bit.

Componente 6 - ULA de 1 Bit

Implemente e verifique o funcionamento de um componente em VHDL correspondente a uma ULA de 1 bit que respeite a seguinte entidade, baseada no diagrama da Figura 2.

```
entity ula1bit is
  port (
    a         : in  bit;
    b         : in  bit;
    cin       : in  bit;
    ainvert   : in  bit;
    binvert   : in  bit;
    operation : in  bit_vector(1 downto 0);
    result    : out bit;
    cout      : out bit;
    overflow  : out bit
  );
end entity;
```

```
entity fulladder is
  port (
    a    : in  bit;
    b    : in  bit;
    cin  : in  bit;
    s    : out bit;
    cout : out bit
  );
end entity;
```

Você pode utilizar o somador completo de 1 bit (`fulladder.vhd`),

fornecido no e-Disciplinas, cuja entidade está ao lado da entidade da ULA, dentro do projeto da ULA.

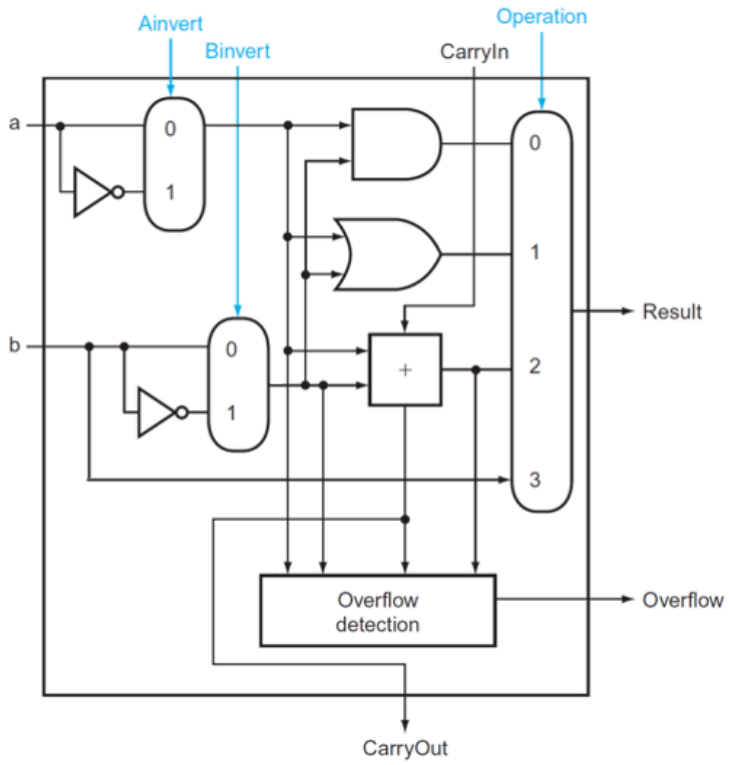

Figura 2: Diagrama da ULA de 1 bit.

A ULA de 1 bit possui as seguintes características: opera sempre sobre as entradas `a` e *b*, ou sobre suas versões negadas caso as entradas `ainvert` ou `binvert` estejam, respectivamente, ativas. As funções da ULA de 1 bit são `AND`, `OR`, `ADD` ou `Pass B`, e devem ser escolhidas por meio de um multiplexador que decide qual resultado será colocado na saída `result` (*F*) na ordem mostrada. Dessa forma, `AND` é a operação da saída quando `operation=00`; `OR` é a operação da saída quando `operation=01`; `ADD` é a operação da saída quando `operation=10`; e `Pass B` é a operação da saída quando `operation=11`.
As operações realizadas nas unidades `AND` e `OR` são bit a bit (*bitwise*). A adição leva em consideração o *carry-in* e pode gerar um *carry-out*. A saída `overflow` é alta quando houver *overflow* no somador, independente da seleção do multiplexador de saída e considerando que esse módulo formará uma ULA de múltiplos bits. Por último, caso o multiplexador selecione a função `Pass B`, a entrada `B` é copiada para a saída.

A saída *overflow* só faz sentido e só será usada no bit mais significativo.

Componente 7 - Extensor de Sinal com Tamanho Configurável

Implemente e verifique o funcionamento de um componente em VHDL correspondente a um extensor de sinal de um dado de entrada contido em uma sequência de 'dataISize' bits para um dado de saída com 'dataOSize' bits que respeite a seguinte entidade:

```
entity sign_extend is
    generic(
         dataISize          : natural := 32;
         dataOSize          : natural := 64;
         dataMaxPosition    : natural := 5 -- sempre fazer log2(dataISize)
         );
    port(
        inData          : in  bit_vector(dataISize-1 downto 0); -- dado de entrada
                          -- com tamanho dataISize
        inDataStart     : in  bit_vector(dataMaxPosition-1 downto 0); -- posicao do bit
                          -- mais significativo do valor util na entrada (bit de sinal)
        inDataEnd       : in  bit_vector(dataMaxPosition-1 downto 0); -- posicao do bit
                          -- menos significativo do valor util na entrada
        outData         : out bit_vector(dataOSize-1 downto 0) -- dado de saida
                          -- com tamanho dataOSize e sinal estendido
    );
end entity sign_extend;
```

O extensor de sinal funciona da seguinte forma: dentro do sinal de entrada (*inData*), pode ser que apenas parte do seu conteúdo seja aproveitado para realizar uma extensão de sinal (exemplo: o imediato das instruções de memória tem apenas 9 bits, mas a instrução completa é passada como entrada para o módulo *sign_extend*). Para delimitar as posições de início e fim da entrada (*inData*) efetivamente úteis, passam-se duas entradas adicionais: *inDataStart*, que deve receber o valor da posição do bit mais significativo (bit de sinal) da parte da entrada *inData* a ter seu sinal estendido, e *inDataEnd*, que deve receber o valor da posição do bit menos significativo da parte da entrada *inData* a ter seu sinal estendido.
De posse dessas informações, a saída *outData* deve corresponder aos bits situados entre as posições *inDataStart* e *inDataEnd* de *inData*, incluindo-as, e replicando o bit de sinal (i.e., bit da posição *inDataStart*) em todas as posições mais significativas até *'dataOSize-1'*.
Exemplo: *inData* = 00010001, *inDataStart* = 100, *inDataEnd* = 001. Nesse caso, apenas os bits 4 até 1 da entrada são considerados na extensão de sinal - i.e., apenas o trecho em negrito da entrada *inData*: 000**1000**1. Supondo que o parâmetro *dataOSize* valha 10, a saída *outData* deve ser 1111111000: '1000' representa o dado extraído de *inData*, e os uns à esquerda são a extensão do seu sinal negativo até completar os 10 bits de *outData*.

Componente 8 - Deslocador Assíncrono de 2 Bits à Esquerda

Implemente e verifique o funcionamento de um componente em VHDL que desloque a entrada `input` duas vezes para a esquerda de forma assíncrona e disponibilize o resultado da operação em `output`. Ele deve respeitar a seguinte entidade:

```
entity two_left_shifts is
    generic(
         dataSize        : natural := 64
     );
    port(
```

```
        input    : in  bit_vector(dataSize-1 downto 0);
        output   : out bit_vector(dataSize-1 downto 0)
    );
end entity two_left_shifts;
```

Os bits menos significativos devem ser sempre preenchidos com zeros.

Dicas de Projeto - Dados em 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 a seguir 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 desse 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))
```

Dicas de Projeto - Iniciação de Memórias com Arquivo de Texto

Para iniciar uma memória com um conteúdo lido de um arquivo de texto, pode-se utilizar uma função em VHDL que inicia a 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_carga.dat" indica o nome do arquivo a ser lido.

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

Uma referência sobre como codificar a função init_mem pode ser encontrada na seção **Arquivo** do endereço `https://balbertini.github.io/vhdl_mem-pt_BR.html`. 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 de dados, em binário, na ordem dos endereços.
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.

Instruções para os Grupos

A Atividade Formativa deve ser realizada no mesmo **grupo de 4 a 6 alunos** das Atividades Formativas anteriores. A verificação funcional de funcionamento dos componentes deve ser realizada com a implementação dos casos de teste, a criação dos *testbenches* e a simulação com GHDL/GTKWave, EDAPlayground ou Quartus/ModelSim.

Não é necessário realizar a entrega das produções intermediárias do projeto e do relato desta parte da Atividade Formativa: ela deve ser guardada e combinada com os desenvolvimentos das próximas partes para uma única entrega.

Reforça-se, porém, todo o material produzido pelo grupo (**relato E arquivos de código**) deve conter **explicitamente** a identificação de **todos** os membros do grupo por meio de **Número USP, Nome Completo e Turma**.

Atenção: É vedado o uso da biblioteca `ieee.std_logic_1164`, ou qualquer outra biblioteca não padronizada, no projeto do PoliLEGv8.