Departamento de Ciência da Computação

GCC260 – Laboratório de Circuitos Digitais

Arquitetura Registrador-Acumulador + Memória ROM

1. Objetivos

- Aprender a usar componentes parametrizáveis da biblioteca de componentes do Quartus Prime;
- Adicionar memória ROM no projeto da arquitetura registrador-acumulador;
- Entender e verificar como um programa em binário pode ser executado em um processador.

2. Passo a passo

O passo a passo para implementar e testar este projeto consiste nas seguintes etapas:

- Crie uma nova pasta na Área de Trabalho, chamada PraticaAcumuladorROM;
- Baixe os arquivos dos módulos que serão necessários:
  - SEG7_LUT.v (https://github.com/brabreus/GCC260-UFLA/blob/main/PraticaAcumuladorROM/SEG7_LUT.v);
  - ALU.v (https://github.com/brabreus/GCC260-UFLA/blob/main/PraticaAcumuladorROM/ALU.v);
  - Register.v (https://github.com/brabreus/GCC260-UFLA/blob/main/PraticaAcumuladorROM/Register.v);
  - Counter.v (https://github.com/brabreus/GCC260-UFLA/blob/main/PraticaAcumuladorROM/Counter.v);
  - Salve na pasta criada anteriormente.
- Crie um projeto no Quartus Prime, mude o diretório de criação do projeto para a pasta criada anteriormente, adicione os arquivos baixados ao projeto, selecione o *Board* como *Family*: MAX10 e o kit DE10-LITE. Lembre-se de deixar marcada a opção "*Create top-level design file*";
- Entenda os códigos baixados;
- Crie a memória ROM (seguir instruções na Seção 3);
- Leia a Seção 4 e complete o código *DE10_LITE_Golden_Top.v* para atender às características pedidas para o circuito;

- Compile o projeto (*Processing->Start Compilation*);
- Use a ferramenta em *Tools->Netlist Viewers->RTL Viewer* para visualizar o desenho gerado pelo seu código verilog e verificar se o projeto está codificado corretamente;
- Conecte a FPGA ao computador;
- Programe a FPGA em *Tools->Programmer*;
- Verifique o funcionamento do circuito.

Apresente o projeto funcionando para o Professor e envie para o Campus Virtual.

3. Criando a memória ROM usando a biblioteca do Quartus Prime

O primeiro passo é criar um arquivo que será usado para inicializar o conteúdo da memória. Esse arquivo pode ser tanto extensão .mif quanto .hex. Nesta prática, usaremos o .mif (*Memory Initialization File*). Vá no menu principal do Quartus Prime *File->New...*, escolha *Memory Initialization File* e clique em OK (Figura 1).

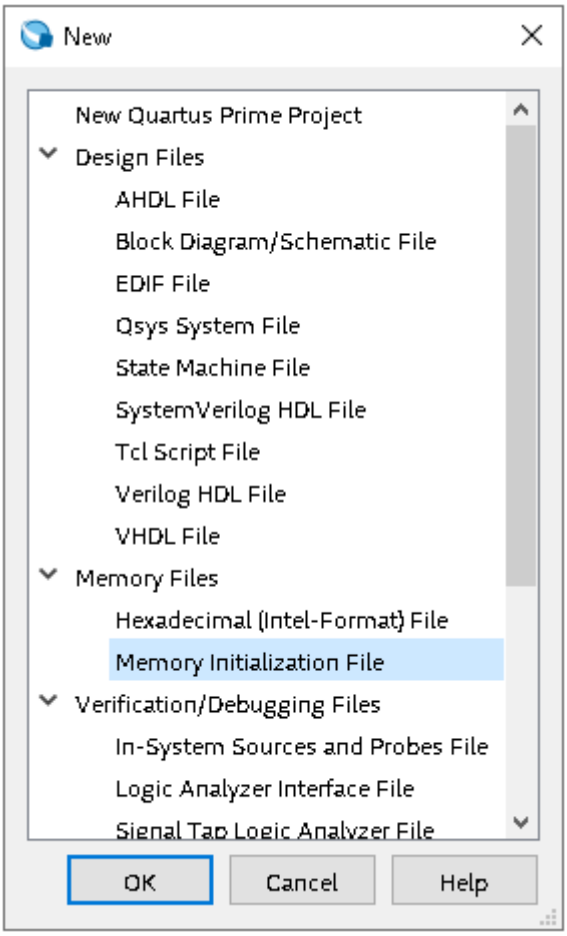

Figura 1: Tela para selecionar o tipo de arquivo a ser criado.

Na tela seguinte, como nossa memória terá 32 palavras, cada uma com 12 bits, defina *Number of words* como 32 e *word size* como 12 (Figura 2).

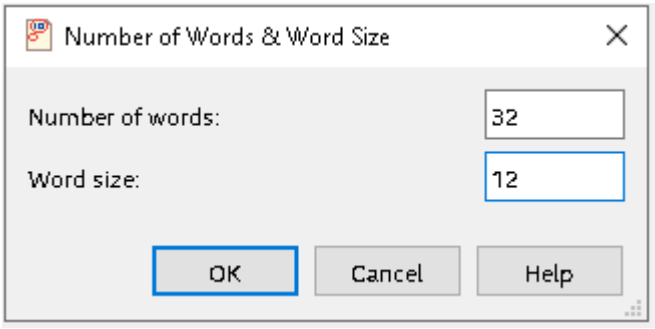

Figura 2: Tela para definir as dimensões do arquivo de inicialização de memória.

O arquivo será criado e aberto no Quartus no formato de uma tabela (Figura 3).

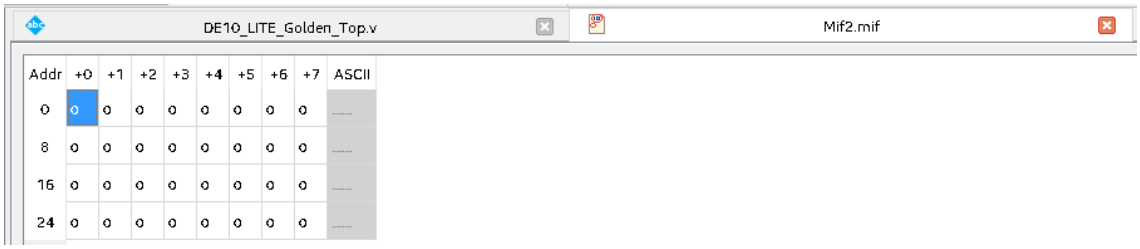

Figura 3: Exibição em forma de tabela do arquivo de inicialização de memória.

Clique com o botão direito do mouse em cima da primeira linha e defina *Memory Radix* como *Binary* para visualizar melhor o conteúdo de cada palavra da memória (Figura 4).

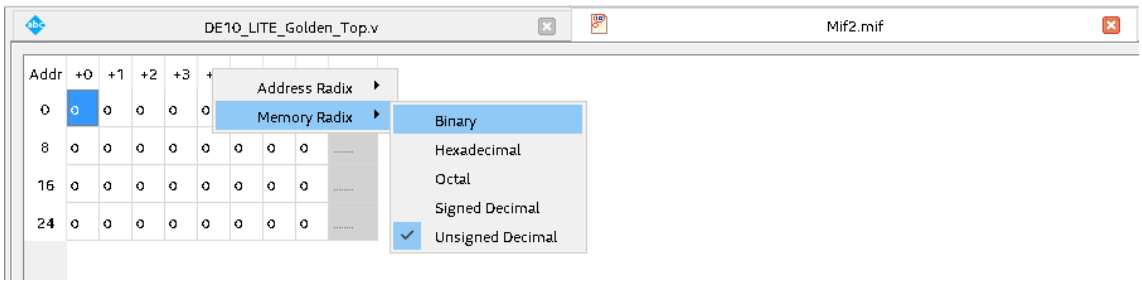

Figura 4: Selecionando o formato de exibição das informações da memória.

Vamos agora preencher o conteúdo da memória para que o sistema execute o programa (sequência de instruções) que desejamos. **Na prática de hoje, calcularemos a soma de dez números e por fim dividiremos o resultado por 10. Ou seja, vamos calcular a média aritmética de 10 números.** Para que isso aconteça, a memória deve ser preenchida por você com os valores apresentados na Figura 5.

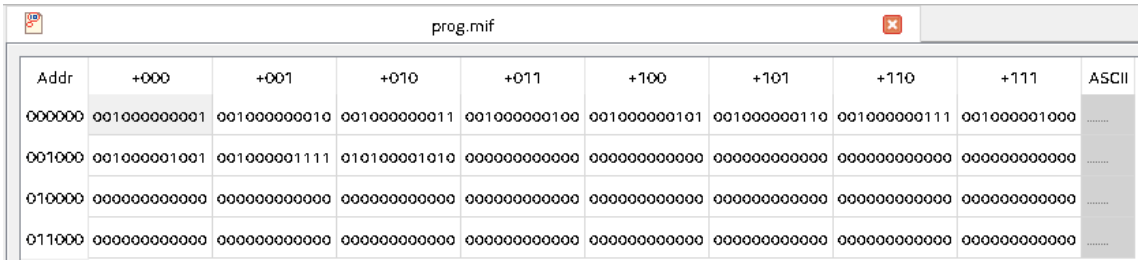

Figura 5: Arquivo de inicialização de memória preenchido com os dados binários para calcular a média de dez números.

Em cada palavra de 12 bits da memória, os bits de 11 a 8 representam o código da operação (soma = 0010 e divisão = 0101) e os bits de 7 a 0 representam o valor de B que será usado na operação. **Salve o arquivo de inicialização da memória como *prog.mif*.**

Esse arquivo que acabamos de criar e preencher não é a memória em si, mas somente o arquivo que armazena qual será o seu conteúdo após gravar o circuito na FPGA. Por isso, o passo seguinte é criar o componente de hardware que de fato implementa a memória. Utilizaremos a biblioteca de componentes do Quartus Prime (*IP Catalog*). Por padrão, o *IP Catalog* é exibido do lado direito da janela do Quartus Prime, caso ela não esteja visível, acesse o menu *View->Utility Windows->IP Catalog* (Figura 6).

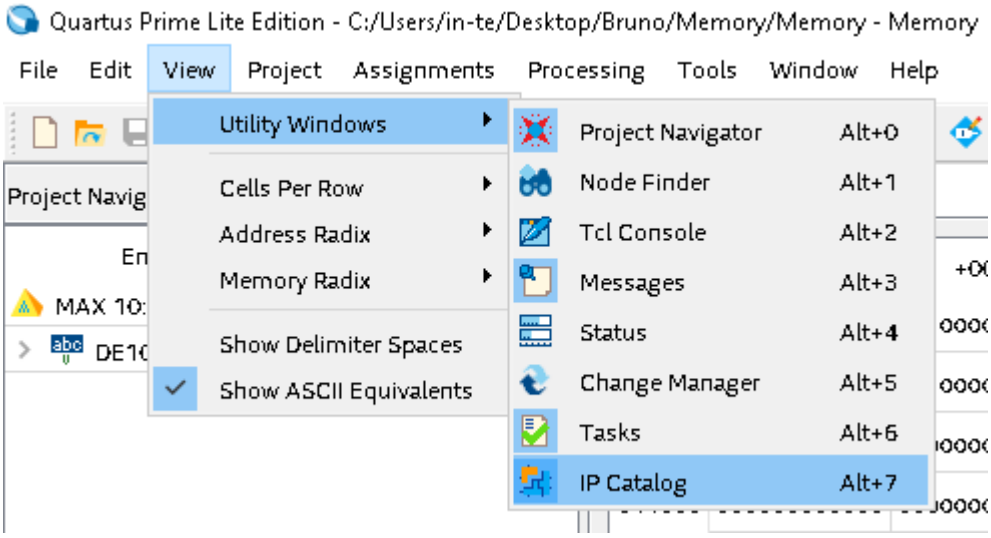

Figura 6: Como exibir o IP Catalog.

Navegue pelo IP Catalog (perceba que há vários componentes prontos que podem ser usados), vá em *Library->Basic Functions->On-chip Memory* e dê um duplo clique em *ROM: 1-PORT* (Figura 7).

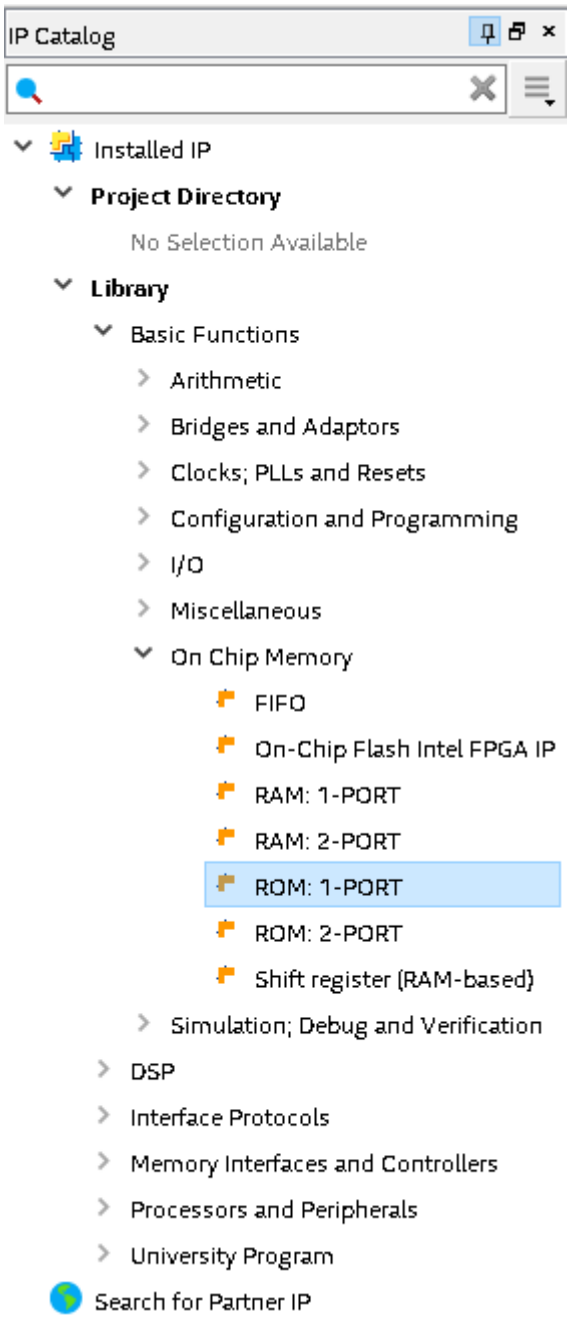

Figura 7: Como selecionar o tipo correto de memória para o projeto.

Na sequência, você irá escolher o nome para o componente de memória e qual linguagem será usada para cria-lo. Digite o nome (pode ser *rom*) e selecione Verilog. Clique em OK (Figura 8).

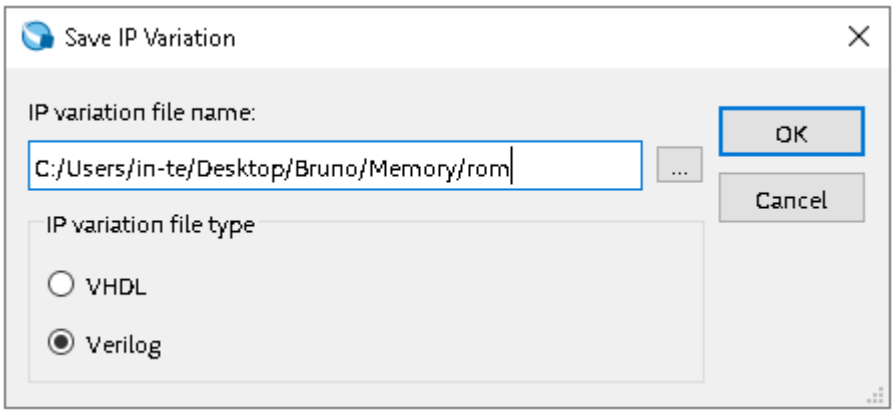

Figura 8: Definindo nome e linguagem para o componente de memória.

Então, será aberto o assistente para criação do componente. Nesse assistente, haverá uma sequência de 5 etapas onde iremos configurar os parâmetros desejados para a nossa memória. Configure o componente de acordo com as Figuras 9, 10, 11, 12 e 13.

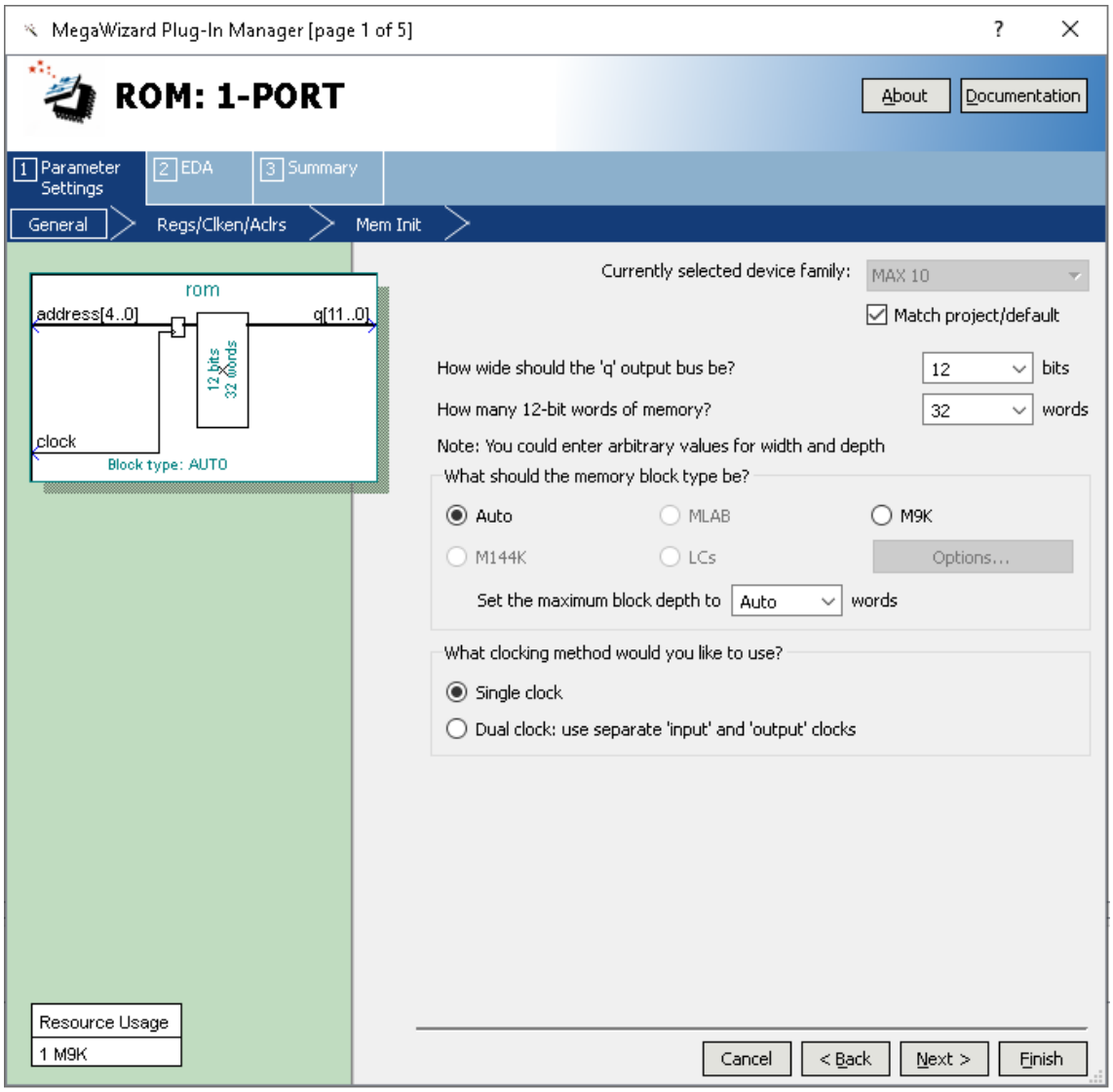

Figura 9: Etapa 1 de configuração do componente de memória ROM.

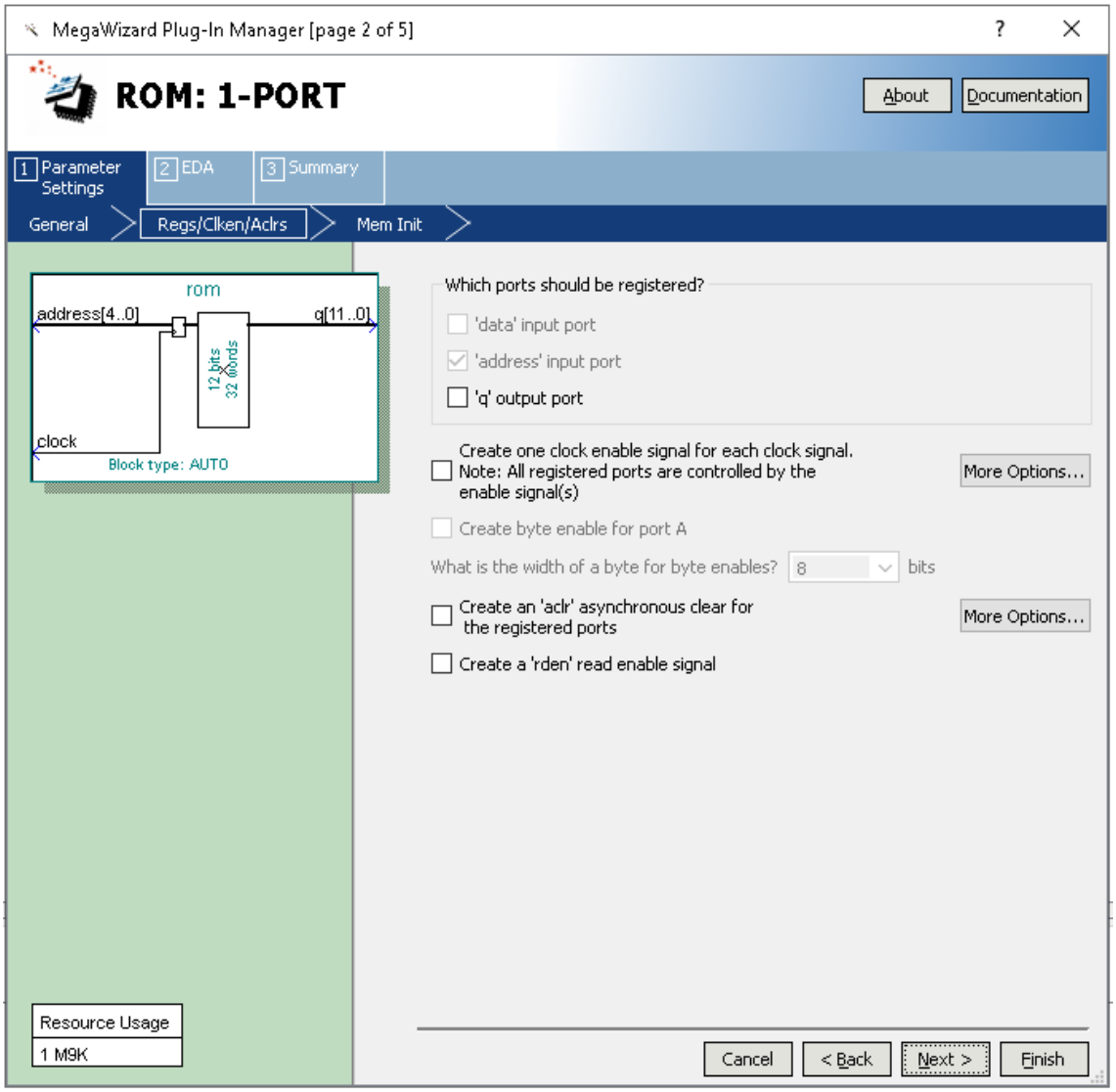

Figura 10: Etapa 2 de configuração do componente de memória ROM.

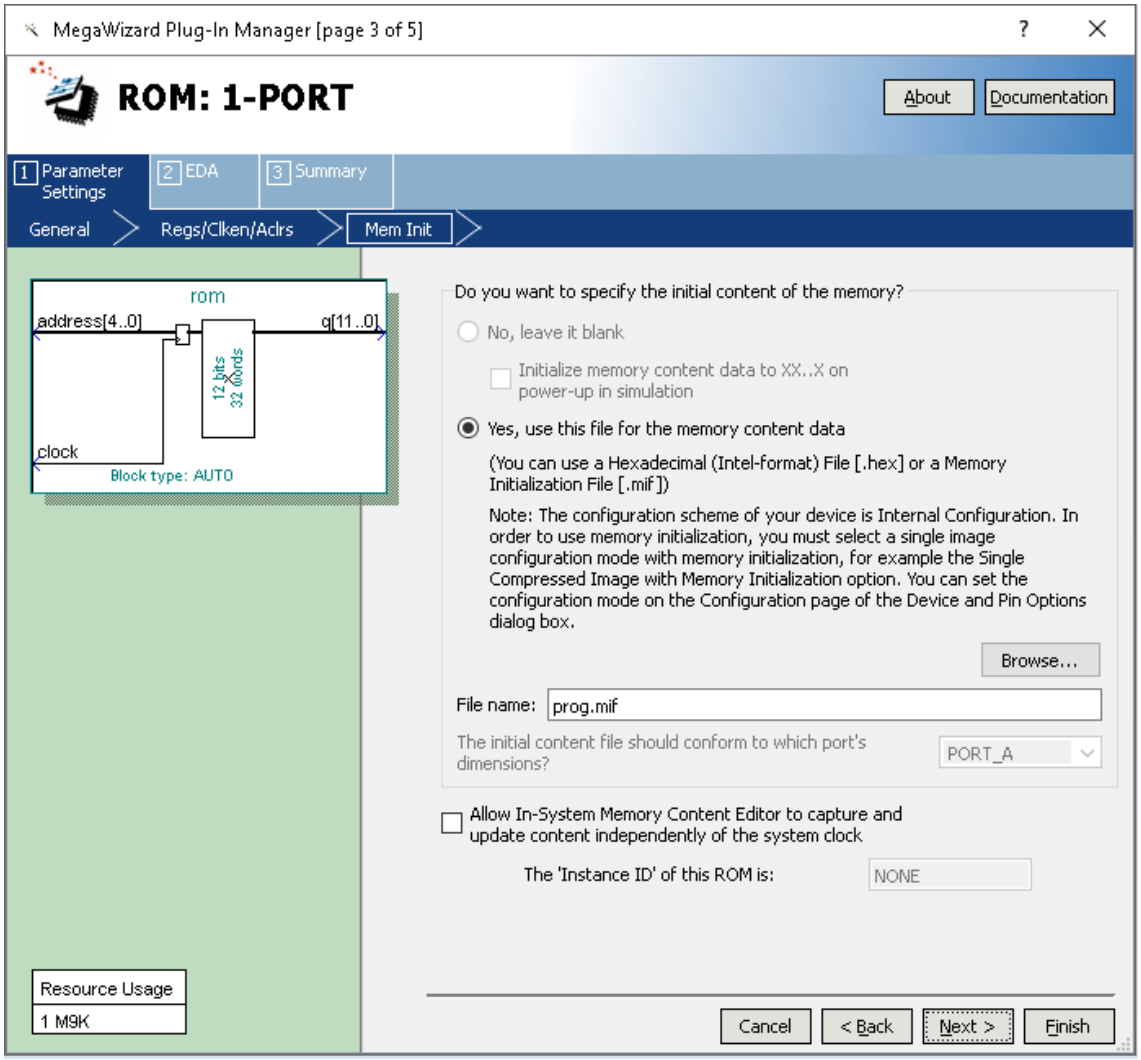

Figura 11: Etapa 3 de configuração do componente de memória ROM.

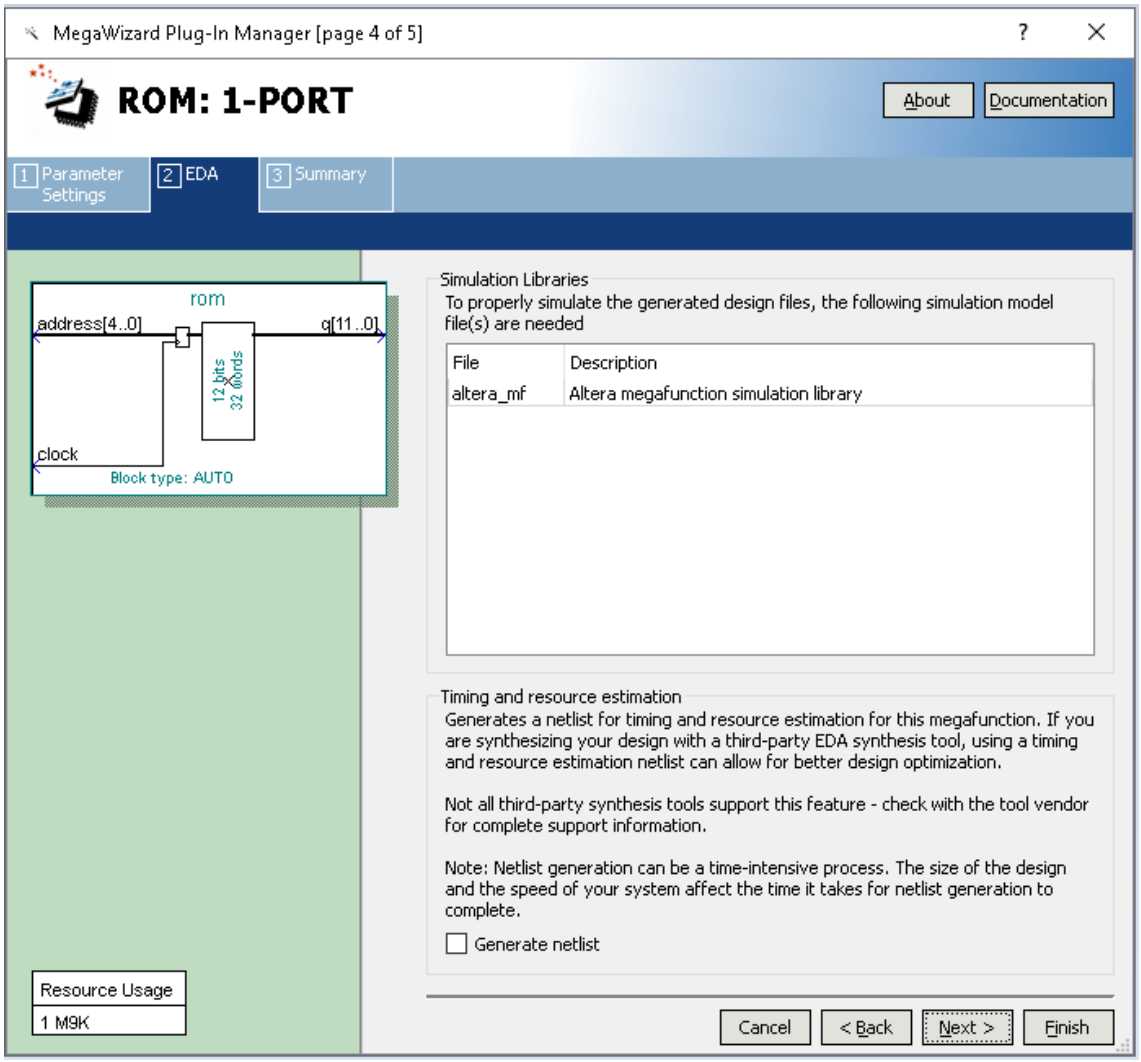

Figura 12: Etapa 4 de configuração do componente de memória ROM.

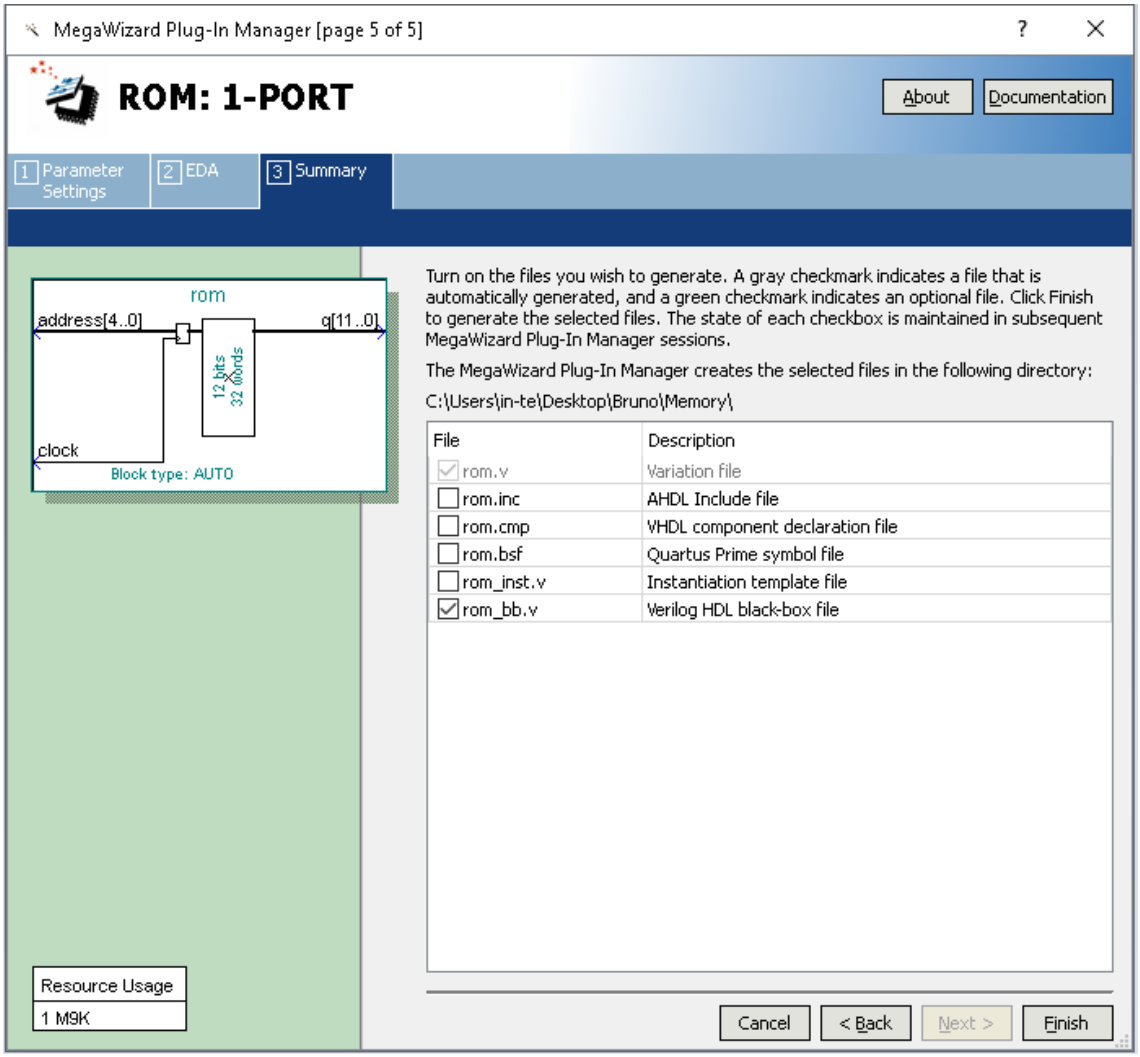

Figura 13: Etapa 5 de configuração do componente de memória ROM.

Após clicar em *Finish*, a memória já estará criada e poderá ser usada no projeto. Porém, como usaremos um arquivo de inicialização de memória, devemos configurar o projeto para compilar corretamente considerando esse arquivo. Acesse o menu principal do Quartus *Assignments->Device*. Clique no botão *Device and Pin Options* (Figura 14).

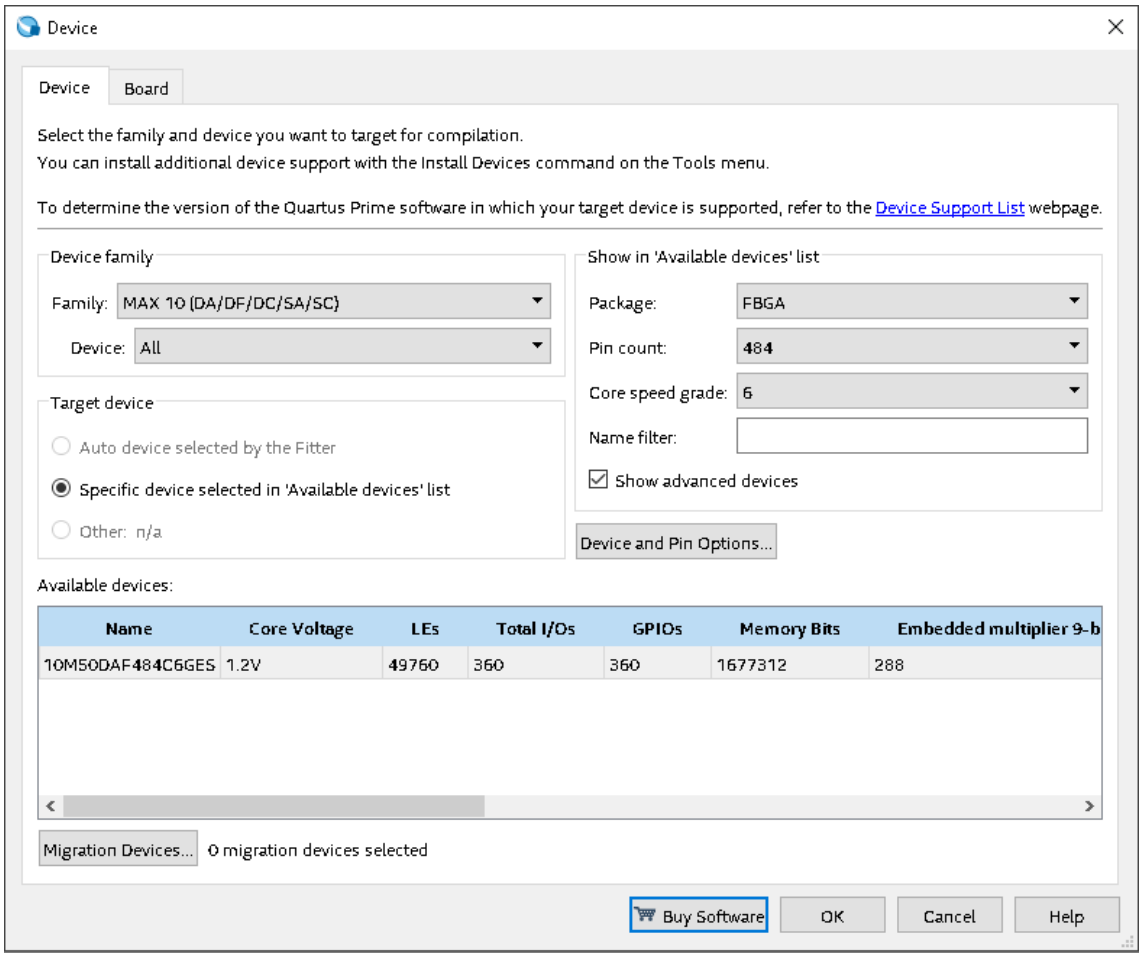

Figura 14: Configurações do projeto relacionadas ao dispositivo.

Selecione a categoria *Configuration* e, em *Configuration mode*, selecione a opção **Single Uncompressed Image with Memory Initialization (512 Kbits UFM)** (Figura 15). Clique em OK.

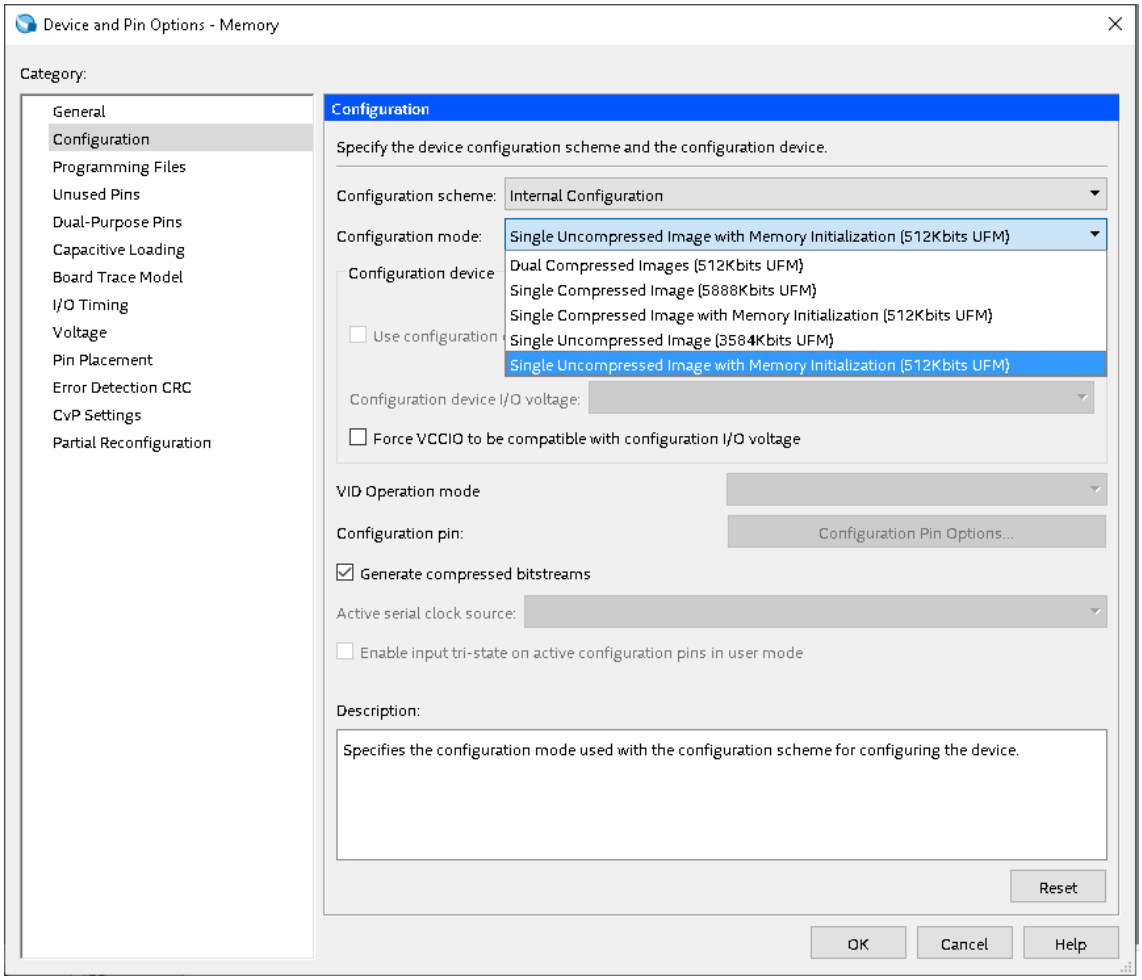

Figura 15: Configuration mode para inicialização de memória.

Agora, temos todos os arquivos necessários para implementar nosso processador de 8 bits que será capaz de executar um programa que calcula a média de 10 números inteiros.

4. Arquitetura Registrador-Acumulador + ROM

O circuito, que será implementado na DE10-LITE, está representado na Figura 1.

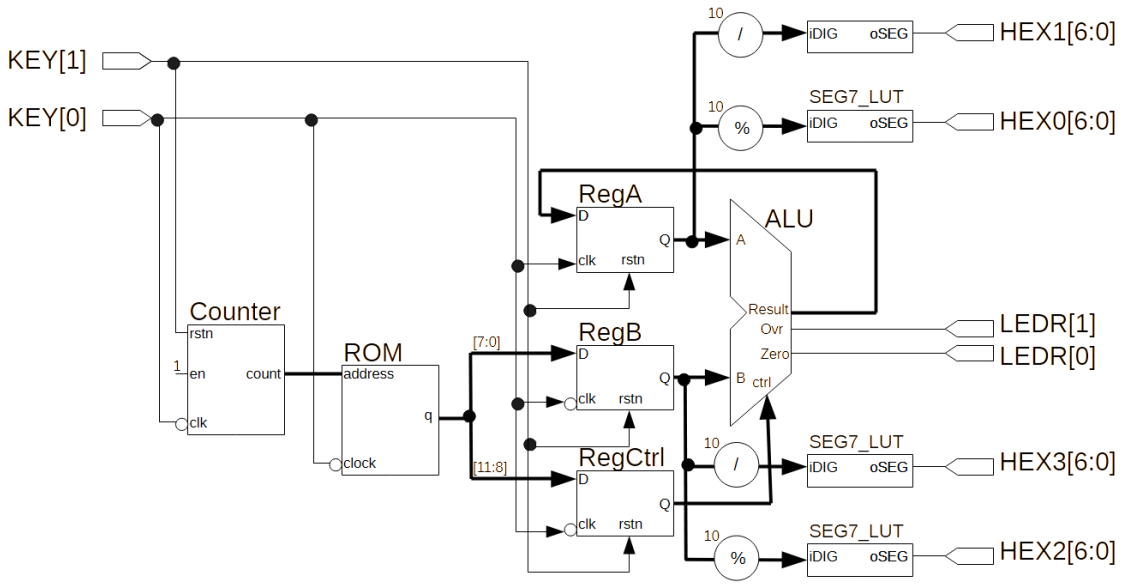

Figura 16: Processador de 8 bits com dois registradores A e B, onde um deles é acumulador. As conexões em negrito representam conexões de 8 bits. A memória ROM 32x12 é utilizada para armazenar os valores de B e as operações que serão realizadas na ALU e o contador permite que os endereços da memória sejam acessados sequencialmente, de modo que o sistema execute o programa armazenado na ROM.

5. Códigos usados

5.1. Descrição de um registrador usando o Flip-Flop tipo D:

```
// Register.v
module Register(
	input wire clk, rstn,
	input wire [7:0] D,
	output reg [7:0] Q
);

	always@(posedge clk, negedge rstn)
	begin
		if(rstn == 0)
			Q <= 8'b00000000;
		else
			Q <= D;
	end

endmodule
```

5.2. Descrição de um contador com módulo MOD e quantidade de bits igual a NUM_BITS:

```
// Counter.v
module Counter
#(
	parameter MOD = 32,
	parameter NUM_BITS = 5
(
	input wire clk, rstn,
	output reg [NUM_BITS-1:0] count
);

	always@(posedge clk, negedge rstn)
	begin
		if(rstn == 0)
			count <= {NUM_BITS{1'b0}};
		else
			count <= (count < MOD)? count + 1'b1 : count;
	end

endmodule
```

5.3. Descrição da ALU:

```
// ALU.v
module ALU(
	input wire [3:0] A, B,
	input wire [3:0] ctrl,
	output reg [3:0] Result,
	output reg Ovr,
	output wire Zero
);
	always@*
	begin
		case(ctrl)
			4'b0000: {Ovr, Result} = A & B;
			4'b0001: {Ovr, Result} = A | B;
			4'b0010: {Ovr, Result} = A + B;
			4'b0011: {Ovr, Result} = A - B;
			4'b0100: {Ovr, Result} = A * B;
			4'b0101: {Ovr, Result} = A / B;
			4'b0111: {Ovr, Result} = (A < B)? 1: 0;
			4'b1100: {Ovr, Result} = ~(A | B);
			default: {Ovr, Result} = 0;
		endcase
	end
	assign Zero = (Result == 0);
endmodule
```

5.4. Descrição do SEG7_LUT:

```
module SEG7_LUT      (
       input  wire [3:0]    iDIG,
       output  reg [6:0]    oSEG
);

always @(iDIG)
begin
       case(iDIG)
              4'h0: oSEG = 7'b1000000;
              4'h1: oSEG = 7'b1111001;    // ---t----
              4'h2: oSEG = 7'b0100100;    // |      |
              4'h3: oSEG = 7'b0110000;    // lt    rt
              4'h4: oSEG = 7'b0011001;    // |      |
              4'h5: oSEG = 7'b0010010;    // ---m----
              4'h6: oSEG = 7'b0000010;    // |      |
              4'h7: oSEG = 7'b1111000;    // lb    rb
              4'h8: oSEG = 7'b0000000;    // |      |
              4'h9: oSEG = 7'b0011000;    // ---b----
              4'ha: oSEG = 7'b0001000;
              4'hb: oSEG = 7'b0000011;
              4'hc: oSEG = 7'b1000110;
              4'hd: oSEG = 7'b0100001;
              4'he: oSEG = 7'b0000110;
              4'hf: oSEG = 7'b0001110;
       endcase
end

endmodule
```