

## 实验七 存储器

2021 年春季学期

此情可待成追忆，只是当时已惘然。

— 《锦瑟》，李商隐

存储器（Memory）是电子设备中的记忆器件，用来存放程序和数据。电子设备中全部信息，包括输入的原始数据、程序、中间运行结果和最终运行结果都保存在存储器中。

本实验的目的是了解 FPGA 的片上存储器的特性，分析存储器的工作时序和结构，并学习如何设计存储器。

### 7.1 存储器结构

存储器是一组存储单元，用于在计算机中存储二进制的数，如图 7-1 所示。存储器的端口包括输入端、输出端和控制端口。输入端口包括：读/写地址端口、数据输入端口等；输出端口一般指的是数据输出端口；控制端口包括时钟端和读/写控制端口。存储器的工作过程如下：

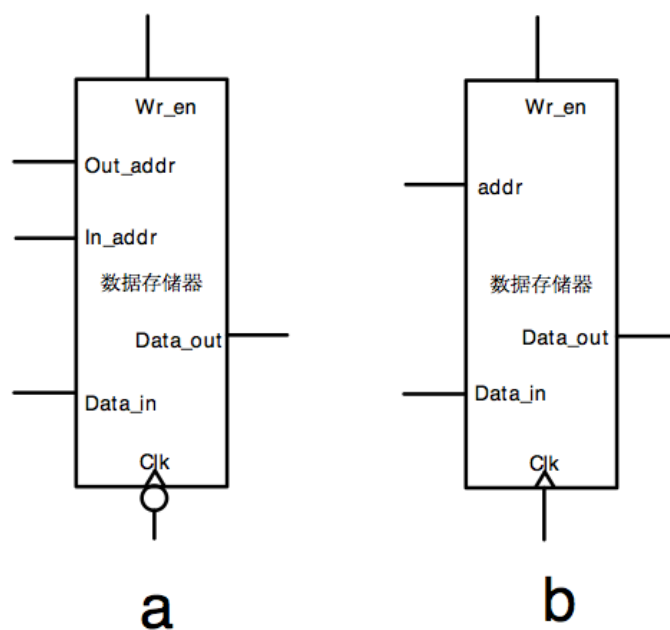



图 7-1: 存储器结构

**写数据：**在时钟（clk）有效沿（上升或下降沿），如果写使能（Wr\_en，也可以没有使能端）有效，则读取输入总线（Data\_in）上的数据，将其存储到输入地址线（In\_addr）所指的存储单元中。

**读数据：**存储器的输出可以受时钟和使能端的控制，也可以不受时钟和使能端的控制。如果输出受时钟的控制，则在时钟有效沿，将输出地址所指示的单元中的数据，输出到输出总线上（Data\_out）；如果不受时钟的控制，则只要输出地址有效，就立即将此地址所指的单元中的数据送到输出总线上。

 对于存储器，其读写时序非常重要，也是实践中容易出错的地方。读取数据时在哪个时间点数据有效，写入数据过多久可以读取这些都要在设计时反复检查和验证。

FPGA 存储器的工作模式有很多，如：真双口 RAM、简单双口 RAM、单口 RAM、ROM 或者 FIFO 缓存等。常见的模式请参照下表。

表 7-1: 存储器的工作模式

存储器模式	说明
单口存储器	某一时刻，只读或者只写
简单双口存储器模式	简单双口模式支持同时读写（一读一写）
混合宽度的简单双口存储器模式	读写使用不同的数据宽度的简单双口模式
真双口存储器模式	真双口模式支持任何组合的双口操作：两个读口、两个写口和两个不同时钟频率下的一读口一写口
混合宽度的真双口存储器模式	读写使用不同的数据宽度的真双口模式
ROM	工作于 ROM 模式，ROM 中的内容已经初始化
FIFO 缓冲器	可以实现单时钟或双时钟的 FIFO

在 Verilog HDL 中，可以用多维数组定义存储器。例如，假设需要一个 32 字节的 8 位存储器块，即此存储器共有 32 个存储单元，每个存储单元可以存储一个 8 位的二进制数。这样的存储器可以定义为 32×8 的数组，在 Verilog 语言中可以作如下变量声明：

```
1 Reg [7:0] memory_array [31:0];
```

存储单元为 memory\_array [0]~memory\_array [31]，每个存储单元都是 8 位的存储空间。在读取时，可以用 memory\_array [13][3:0] 直接读取第 13 号单元的低 4 位。

## 7.2 存储器的实现

Cyclone V 系列 FPGA 内部含有两种嵌入式存储器块：

10Kb 的 M10K 存储器块——这是专用存储器资源块。M10K 存储器块是理想的大存储器阵列，并提供大量独立端口。

64 位存储器逻辑阵列（MLABs）——是一种嵌入式存储器阵列是由双用途逻辑阵列块配置而来的。MLAB 是理想的宽而浅的存储阵列。MLAB 是经过优化的可以用于实现数字信号处理（DSP）应用中的移位寄存器、宽浅 FIFO 缓存和滤波延迟线。每个 MLAB 都由 10 个自适应逻辑块（ALM）组成。在 Cyclone V 系列器件中，你可以将这些 ALM 可配置成 10 个 32×2 模块，从而每个 MLAB 可以实现一个 32×20 简单双端口 SRAM 模块。

Cyclone V 系列 FPGA 嵌入式存储器资源如图 7-2 所示，我们可以对应比较一下 DE10-standard 开发平台上配置的 Cyclone V SX C6 的存储器资源。

Variant	Member Code	M10K		MLAB		Total RAM Bit (Kb)
		Block	RAM Bit (Kb)	Block	RAM Bit (Kb)	
Cyclone V GX	C3	135	1,350	291	182	1,532
	C4	250	2,500	678	424	2,924
	C5	446	4,460	678	424	4,884
	C7	686	6,860	1338	836	7,696
	C9	1,220	12,200	2748	1,717	13,917
Cyclone V GT	D5	446	4,460	679	424	4,884
	D7	686	6,860	1338	836	7,696
	D9	1,220	12,200	2748	1,717	13,917
Cyclone V SE	A2	140	1,400	221	138	1,538
	A4	270	2,700	370	231	2,460
	A5	397	3,970	768	480	4,450
	A6	557	5,570	994	621	5,761
Cyclone V SX	C2	140	1,400	221	138	1,538
	C4	270	2,700	370	231	2,460
	C5	397	3,970	768	480	4,450
	C6	557	5,570	994	621	5,761
Cyclone V ST	D5	397	3,970	768	480	4,450
	D6	557	5,570	994	621	5,761

图 7-2: Cyclone V 系列的存储器资源

Quartus 会根据用户存储器设计的速度与大小，来自动选择硬件实现时使用的存储器模块的数量与配置。例如，为提供设计性能，Quartus 可能将可以由 1 块 RAM 实现的存储器设计扩展为由多块 RAM 来实现。

表 7-2: RAM 代码

```

1 module ram #(
2     parameter RAM_WIDTH = 32,
3     parameter RAM_ADDR_WIDTH = 10
4 ) (
5     input clk,
6     input we,
7     input [RAM_WIDTH-1:0] din,
8     input [RAM_ADDR_WIDTH-1:0] inaddr,
9     input [RAM_ADDR_WIDTH-1:0] outaddr,
10    output [RAM_WIDTH-1:0] dout
11 );
12
13    reg [RAM_WIDTH:0] ram [(2*RAM_ADDR_WIDTH)-1:0];
14
15    always @(posedge clk)
16        if (we)
17            ram[inaddr] <= din;
18
19    assign dout = ram[outaddr];
20
21 endmodule

```

### 思考题

如果将表 7-2 中存储器实现部分改为

```

1 always @(posedge clk)
2     if (we)
3         ram[inaddr] <= din;
4     else
5         dout <= ram[outaddr];

```

该存储器的行为是否会发生变化?

## 7.2.1 存储器实例分析

表 7-3 是一个存储器实例，实例中为此存储器设置了三个输出端口，请分析存储器结构和工作过程，查看此存储器的 RTL 图，检查存储器的输入输出和存储体的结构，并分析其三个输出端的结构的不同。为此实例设计一个测试代码，研究此三个端口输出数据时在时序上的差别，结合 RTL 图，给出其工作时序的解释。

其中 initial 语句块完成了在启动时对 RAM 的初始化。

表 7-3: 存储器实例代码

```

1 module v_rams_8 (clk, we, inaddr, outaddr, din, dout0, dout1, dout2);
2   input clk;
3   input we;
4   input [2:0] inaddr;
5   input [2:0] outaddr;
6   input [7:0] din;
7   output [7:0] dout0, dout1, dout2;
8
9   reg [7:0] ram [7:0];
10  reg [7:0] dout0, dout1;
11
12  initial
13  begin
14    ram[7] = 8'hf0; ram[6] = 8'h23; ram[5] = 8'h20; ram[4] = 8'h50;
15    ram[3] = 8'h03; ram[2] = 8'h21; ram[1] = 8'h82; ram[0] = 8'h0D;
16  end
17
18  always @(posedge clk)
19  begin
20    if (we)
21      ram[inaddr] <= din;
22    else
23      dout0 <= ram[outaddr];
24  end
25  always @(negedge clk)
26  begin
27    if (!we)
28      dout1 <= ram[outaddr];
29  end
30  assign dout2 = ram[outaddr];
31 endmodule

```

适当选择输入输出端口宽度，将此实例进行引脚约束，利用开关或按钮作为时钟端，在开发板上再次验证其不同输入/输出方式的工作时序。

### 7.2.2 存储器初始化

当需要初始化的 RAM 数据量较大的时候，可以使用文件来在系统启动时直接装入 RAM 数据。Verilog 提供了以下语句来将文件中的数据导入到 RAM 中：

```

1 initial
2 begin
3     $readmemh("D:/digital_logic/mem1.txt", ram, 0, 7);
4 end

```

以上内容可以替代前例中的 RAM 初始化部分，将 mem1.txt 中的数据导入到 ram 变量的第 0 单元至第 7 单元。请注意，这里 mem1.txt 可以存在任何不包含中文字符的目录下，但是在初始化语句中一定要给出此文件的绝对路径，否则仿真时将看不到初始化数据。

mem1.txt 的内容和格式如下：

```
1 @0 0d
2 @1 82
3 @2 21
4 @3 03
5 @4 20
6 @5 ff
7 @6 50
8 @7 04
```

其中 @ 符号后为 ram 地址，随后是 16 进制的 ram 数据。在 verilog 中，\$readmemh 方法读取 16 进制数据，\$readmemb 方法读取 2 进制数据。

初始化存储器时可以选择存储器的部分单元进行初始化，其他单元不初始化。如，假设存储器 ram 有 8 个存储单元，下面的初始化表示只对存储器的 0~5 号单元进行初始化，这也是可以的。

```
1 initial
2 begin
3     $readmemh("D:/digital_logic/mem1.txt", ram, 0, 5);
4 end
```

假设存储器 ram 有 8 个存储单元，下面的初始化试图对存储器的 0~8 号单元，共 9 个单元进行初始化，这是不可以的。

```
1 initial
2 begin
3     $readmemh("D:/digital_logic/mem1.txt", ram, 0, 8);
4 end
```

对存储器进行初始化还有其他方式，我们会在以后的实验中继续介绍。

## 7.3 使用 IP 核生成存储器

Quartus 提供了很多非常实用的 IP 核，利用这些 IP 核可以很方便的实现复杂的设计。下面我们以设计一个存储器为例来介绍如何使用 Quartus IP 核。

### 7.3.1 通过 IP 生成 RAM

在 Quartus 工作区的右边，就是 IP 目录，如下图所示

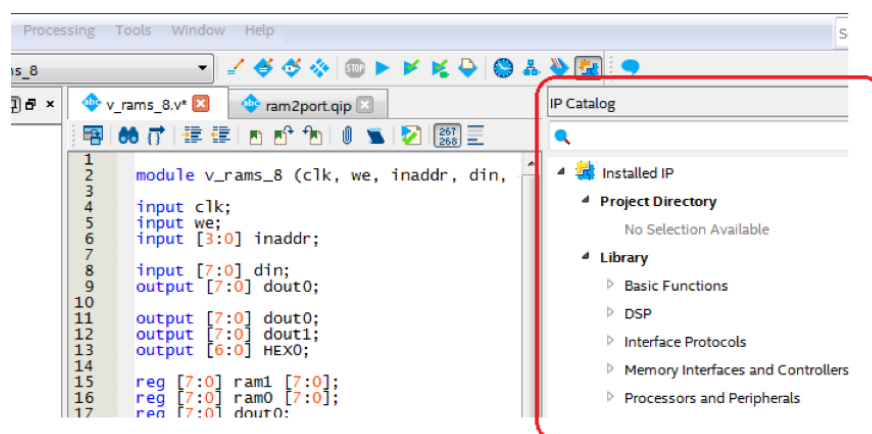


图 7-3: IP 目录

展开 **Library** 可以看见所有用的 IP，继续展开 **Basic Functions→On Chip Memory**，双击 **RAM: 2-PORT**，即双口 RAM。

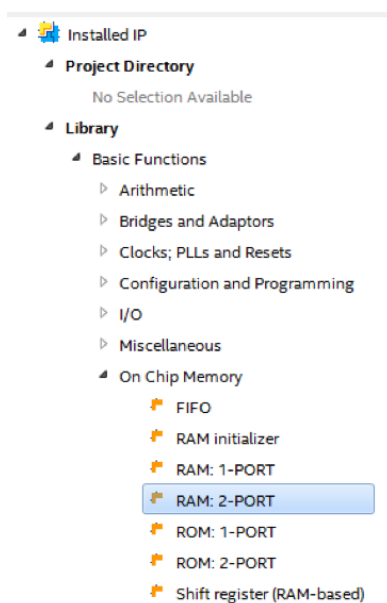


图 7-4: 选择 RAM 类型

弹出对话框，为此 IP 取一个名字，此处取名为“ram2port”，默认保存在当前工程目录下，IP 核对应的硬件描述语言文件选择 Verilog。

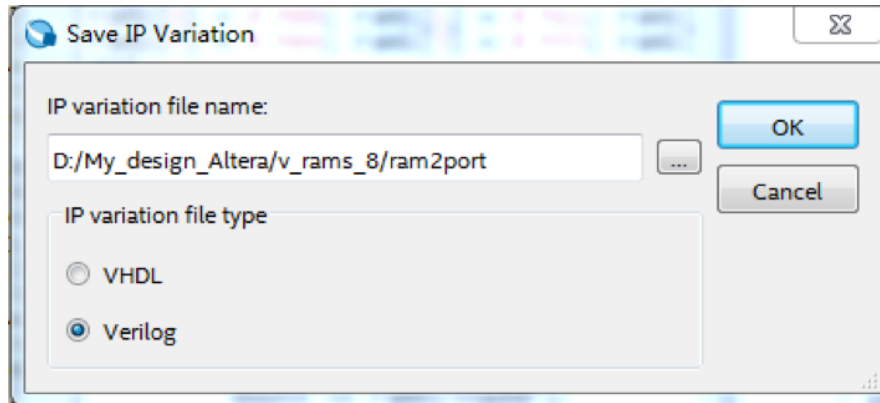


图 7-5: 选择目标文件名

选择 RAM 的端口为一个读口和一个写口，以“Words”为单位计大小。

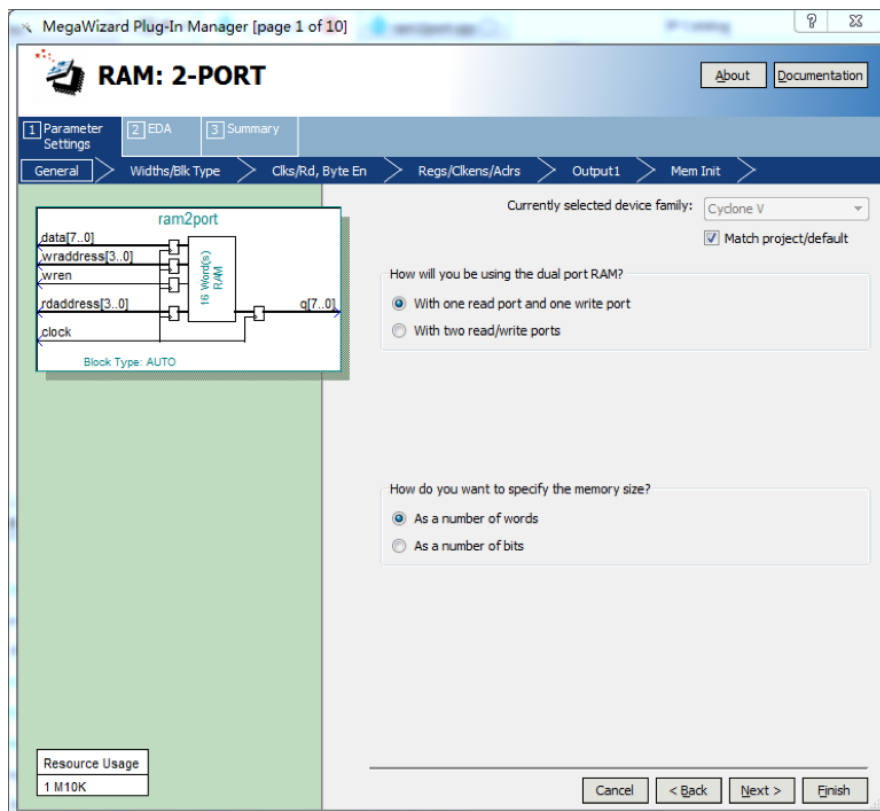


图 7-6: 选择 RAM 格式



选择存储器的大小：这里我们选择的是一个 16×8 bit 的存储器，由编译器自动选择实现存储器的方式是 M10K 还是 MLAB。

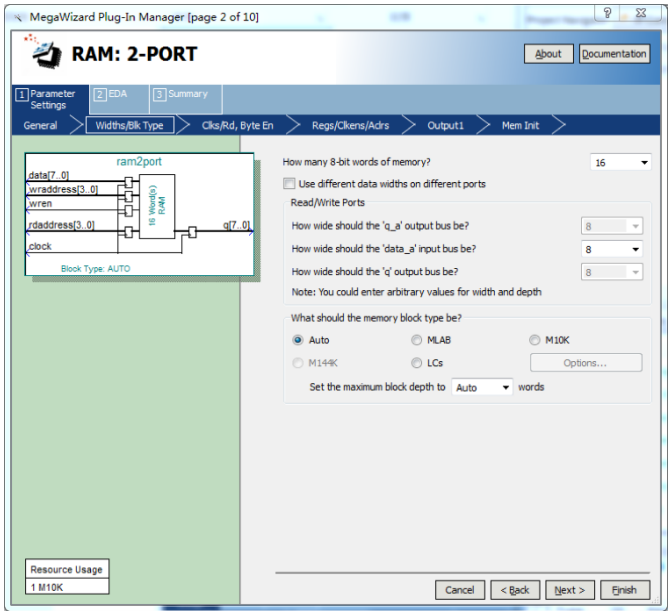


图 7-7: 选择 RAM 规模

对时钟和使能信号等进行配置

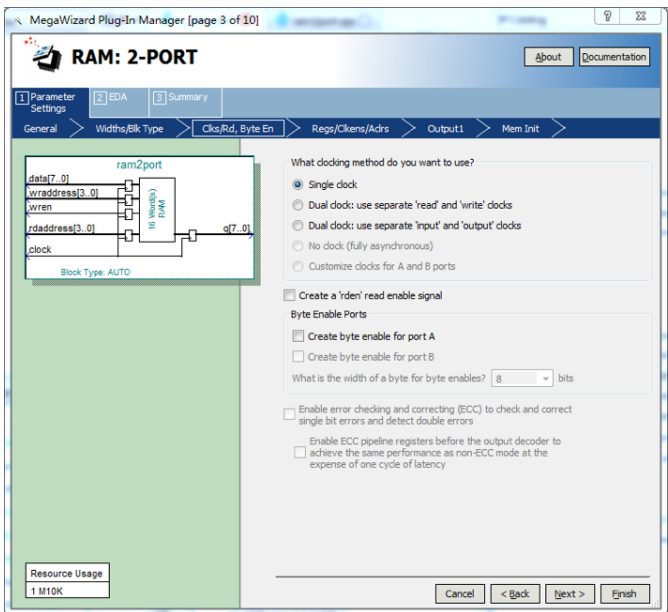


图 7-8: 时钟信号配置

选择是否需要输出缓存寄存器。

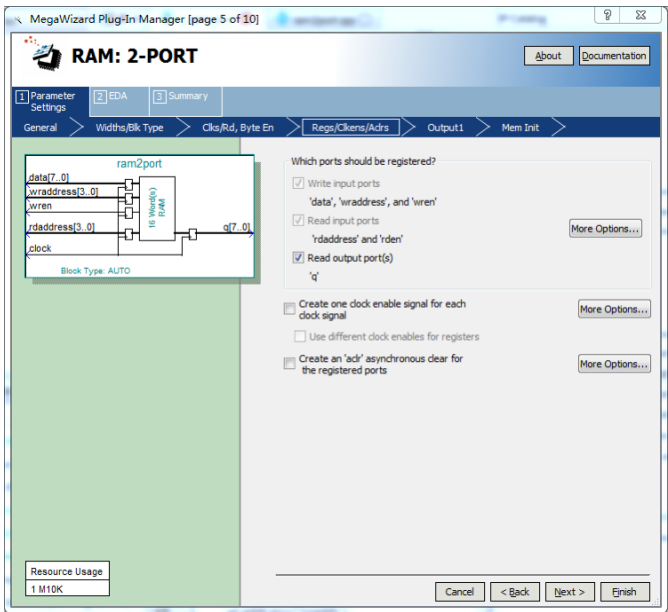


图 7-9: 输出缓存选择

对于单时钟 RAM，选择如何解决“写时读”的数据冲突。如篇首《锦瑟》所言，本周期写入的数据，不一定能够在本周期读出。

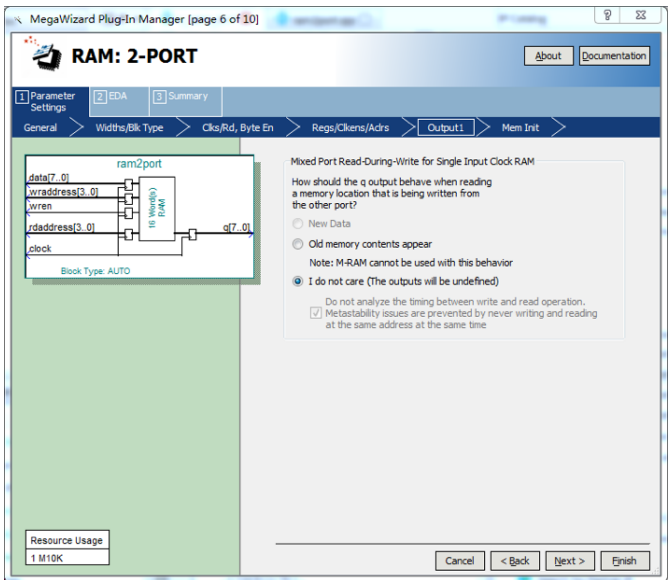


图 7-10: 读写冲突解决

### 7.3.2 存储器初始化

在建立存储器的时候，可以不初始化，也可以利用一个十六进制文件.hex 或者一个存储器初始化文件.mif 进行初始化。

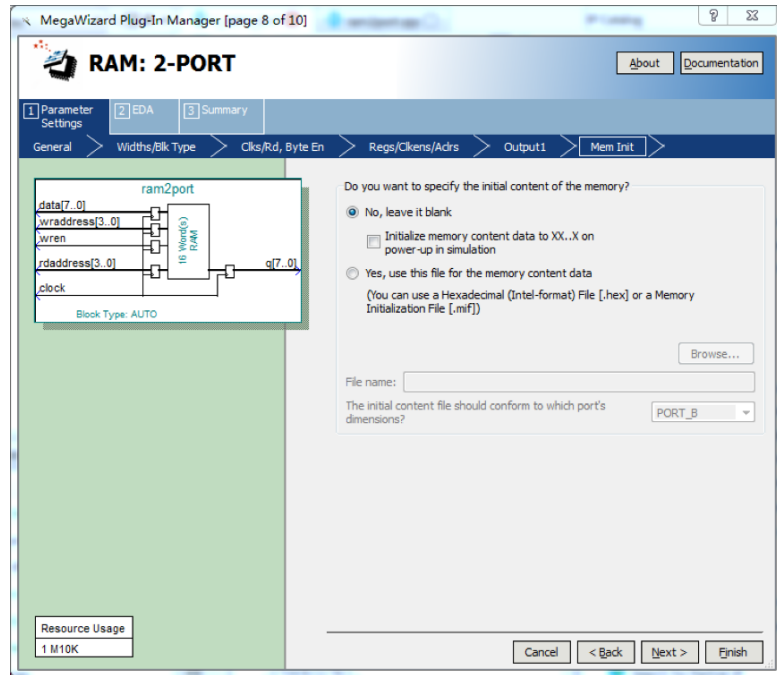


图 7-11: 初始化选择

下面介绍一下.mif 文件的生成。

回到 Quartus 工作区，点击 **File→New** 在 **Memory Files** 目录下选择：“Memory Initialization File”，点击 **OK**。根据存储器大小选择进行设置：

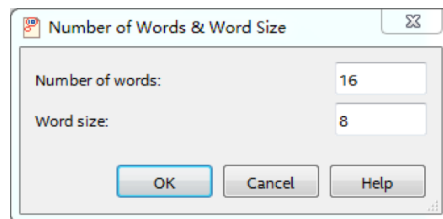


图 7-12: 初始化文件大小选择

点击 **OK**。

编译器自动跳出.mif 文件初值设置界面，对其进行初值设置：

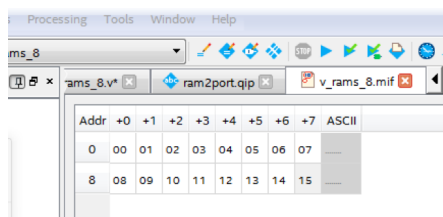


图 7-13: 编辑初始化文件

保存。回到 IP 核生成对话框，点击 **Browse...**。

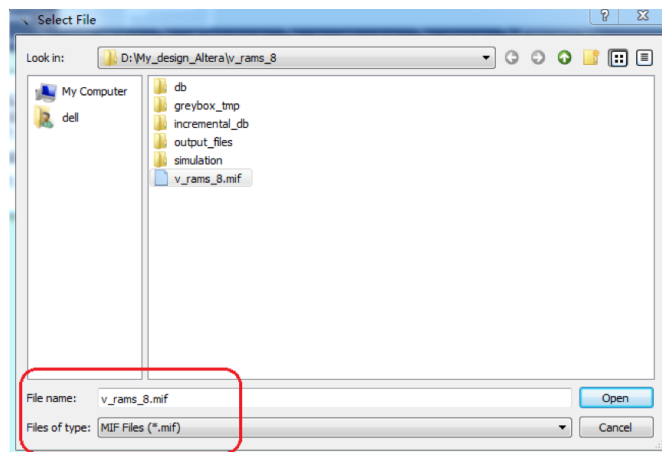


图 7-14: 初始化文件选择

选择刚刚保存的.mif 文件，点击 **Open**，对存储器进行初始化。点击 **Next**，**Next**，**Finished**。

在项目导航栏，**Files** 目录下，展开 **ram2port.qip**，可以看见为此 RAM 生成的 **ram2port.v** 文件，双击打开，可以看见此 **ram2port.v** 的接口参数，在存储器设计的顶层实体中，对此 RAM 进行实例化，即可在设计中使用该 RAM：

```

1 ram2port my_ram(
2     .clock(clk),
3     .data(din),
4     .rdaddress(inaddr),
5     .wraddress(inaddr),
6     .wren(we),
7     .q(dout0));

```

### ✎ 利用 mif 文件初始化存储器

编程中也可以使用 mif 文件来初始化存储器，如下语句即使用 data.mif 来初始化 myrom。这时要求该 mif 文件与.v 文件在一个目录下。

```
1 (* ram_init_file = "data.mif" *) reg [7:0] myrom[255:0];
```

## 7.4 实验内容

请在一个工程中完成如下两个存储器。两个存储器的大小均为  $16 \times 8$ ，即每个存储器共有 16 个存储单元，每个存储单元都是 8 位的，均可以进行读写。

✎ **RAM1**：采用下面的方式进行初始化，输出端有输出缓存，输出地址有效后，等时钟信号的上升沿到来时才输出数据。

```
1 initial
2 begin
3     $readmemh("D:/digital_logic/mem1.txt", ram, 0, 15);
4 end
```

初始化数值为

```
1 @0 00
2 @1 01
3 @2 02
4 @3 03
5 @4 04
6 @5 05
7 @6 06
8 @7 07
9 @8 08
10 @9 09
11 @a 0a
12 @b 0b
13 @c 0c
14 @d 0d
15 @e 0e
16 @f 0f
```

✎ **RAM2**: 利用 IP 核设计一个双口存储器，利用 .mif 文件进行初始化，十六个单元的初始化值分别为：0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff。

此两个物理上完全不同的存储器共用时钟、读写地址和写使能信号，当写使能有效时，在时钟信号的有效沿写入数据；当写使能信号无效时，在时钟信号的有效沿输出数据。适当选择时钟信号和写使能信号，以能够分别对此两个存储器进行读写。

请合理使用 **FPGA** 开发板的输入/输出资源，完成此两个存储器的设计。由于开发板上输入数量不够，写入时可以只写入 2 位数据。