

实验五 寄存器组及存储器

2021 年秋季学期

此情可待成追忆，只是当时已惘然。

— 《锦瑟》，李商隐

寄存器组（Register File）与存储器（Memory）是数字系统中的记忆器件，用来存放程序和数据。从程序员的角度来看，CPU 的状态由其寄存器及存储器中的信息唯一确定。其中寄存器包括程序计数器 PC、通用寄存器，存储器指主存。我们可以将计算机看成一个巨大的有限状态自动机，当这些存储部件中的信息确定后，计算机的状态也确定了。在没有外部输入时，计算机后续的运行状态也是唯一确定的。

本实验的目的是了解 FPGA 的触发器及片上存储器的特性，分析存储器的工作时序和结构，并学习如何设计寄存器组和主存。

5.1 寄存器与寄存器组

FPGA 上有大量的触发器资源来实现数据的存储。D 触发器可以用于存储比特信号，给 D 触发器加上置数功能就变成了一位寄存器，如图 5-1 所示。由图中可以看出，如果 load 信号为 1，则输入信号 in 被送入或门中，或门的另一个输入端为 0，此时 $D=in$ ，所以在下一个时钟里 $q=in$ 。当 load 值为 0 时，q 值被反馈到或门中，或门的另一个输入值为 0，此时 $D=q$ ，因此在下一个时钟周期里 q 值保持先前的值不变。

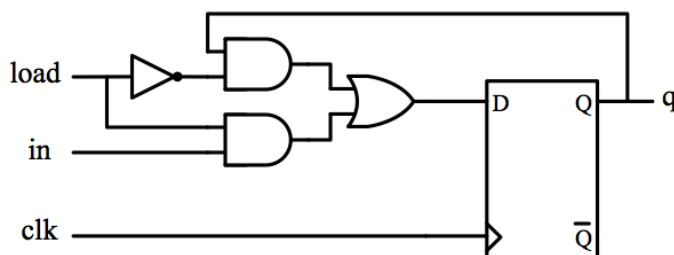


图 5-1: 1 位寄存器

用 Verilog 语言设计寄存器也很简单，如表 5-1 所示。

表 5-1: 1 位寄存器代码

```

1 module register1(load,clk,clr,inp,q);
2     input  load,clr,clk,inp;
3     output reg q;
4
5     always @(posedge clk)
6         if (clr==1)
7             q <= 0;
8         else if (load == 1)
9             q <= inp;
10 endmodule

```

表 5-1 的程序的仿真图如图 5-2 所示。

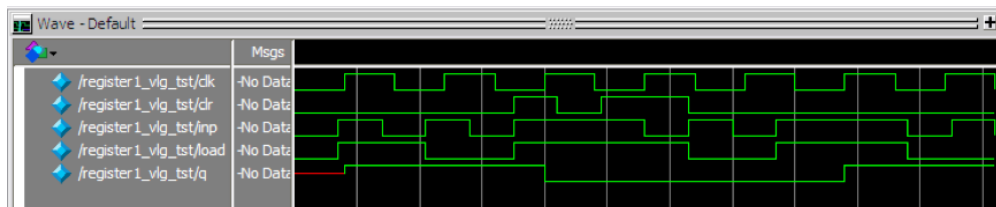


图 5-2: 1 位寄存器仿真结果

本例实现的是一个带有清 0 端和输入端的 1 位寄存器，还有的寄存器带有置位（置 1）端的，图 5-3 是同时带有清 0 端、输入端和置位端的寄存器的逻辑示意图，读者可自行设计此寄存器。

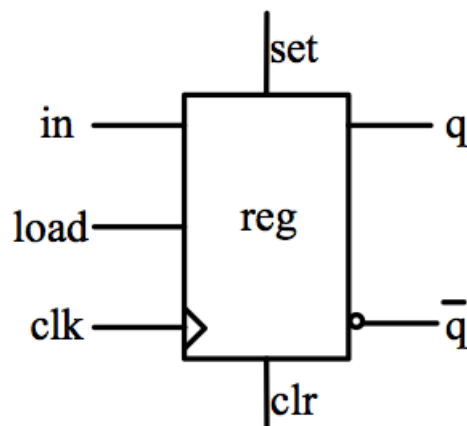


图 5-3: 1 位寄存器框图

将 2 个或者 2 个以上的 1 位寄存器组合在一起，这些寄存器共用一个时钟信号，这就构成了多位寄存器，寄存器常被用在计算机中存储数据，如指令寄

存器、数据寄存器等。表 5-2是利用 Verilog 语言设计寄存器的例子。

表 5-2: 4 位寄存器代码

```
1 module register4(load,clk,clr,d,q);
2     input  load,clr,clk;
3     input  [3:0] d;
4     output reg [3:0] q;
5
6     always @(posedge clk)
7         if (clr==1)
8             q <= 0;
9         else if (load == 1)
10            q <= d;
11 endmodule
```

表 5-2的程序的仿真图如图 5-4所示。

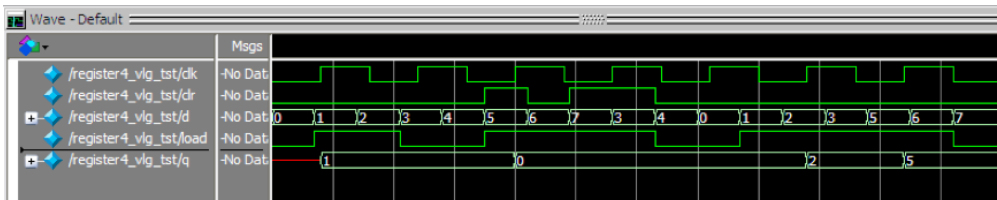


图 5-4: 4 位寄存器仿真结果

5.2 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据，如图 5-5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：

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。

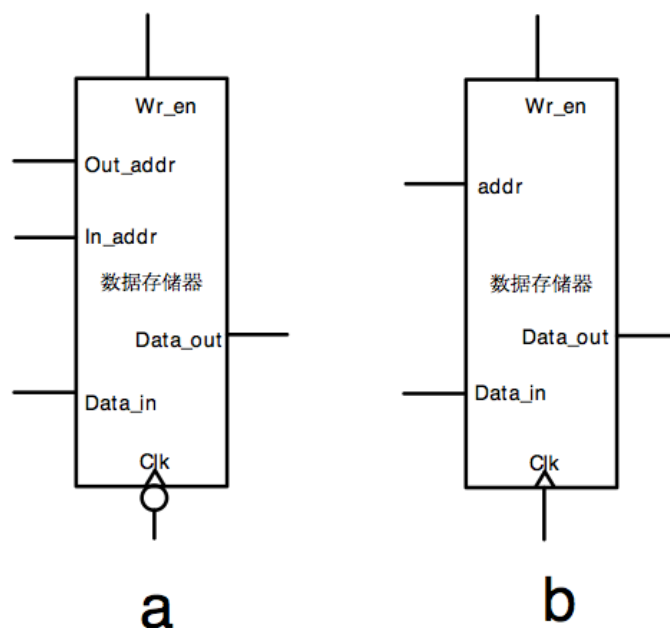


图 5-5: 存储器结构

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。

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。

寄存器与存储器的异同

虽然寄存器和存储器都是用来存储状态信息的，但是它们在用途和实现上较大的区别：

表 5-3: 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

- 寄存器一般要求存取速度快、并行访问要求高，所以通常寄存器的容量较小。在 CPU 中，PC 及通用寄存器会经常被访问，因此存取的时延要求在一个时钟周期内。对于单周期 CPU，每个时钟周期往往要求同时读取 2 个通用寄存器并完成 1 个寄存器的写回。在要求较高的时候，有可能寄存器组输出的结果需要异步输出，即不在时钟沿上读取，输出随着输入地址实时改变。在这样高的要求下，寄存器组的大小不可能太大，否则会消耗非常多的资源。
- 主存一般容量较大，但是读写时间较长，并且读写过程有严格的时序要求。
- 在 Verilog 中，虽然寄存器组和存储器的描述都是二维数组的方式。但是，编译和综合过程中会根据代码访问的要求来选择具体的实现方式。例如，当代码中没有严格在时钟信号沿上进行读写时，系统会认为该存储单元的读写要求较高，直接采用 FPGA 逻辑单元实现。这种实现方式消耗的资源巨大，一般只能支持数 K 量级的存储单元。如果要求大量的此类存储功能，系统可能会花很长时间进行编译综合，甚至无法实现。如果一个存储单元的访问严格按照时序要求，仅在时钟沿上进行每次单个单元的读写时，系统可以用大容量的 M10K 实现存储，一般可以支持到数百 K 字节的容量。因此，在实验中对存储器的读写应特别关注，避免用高级语言的二维数组的思路来看待存储器，否则会造成很多意想不到的后果。

5.3 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 5-6 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 5-6: 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 5-4: 存储器实现代码

```
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-1:0] 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

上述存储器综合时，综合器是否会用 FPGA 的 RAM 模块来实现这个模块？如果将表 5-4 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？

5.3.1 存储器实例分析

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

5.3.2 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。

5.4 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。

5.4.1 通过 IP 生成 RAM

在 Quartus 工作区的右边，就是 IP 目录，如下图所示

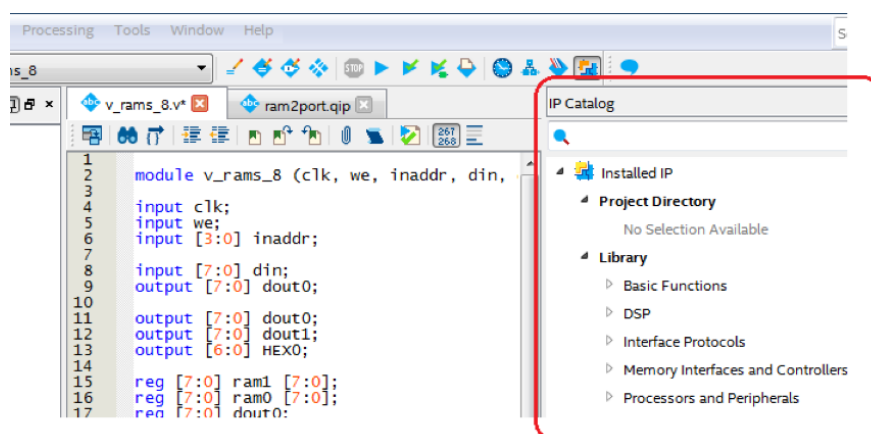


图 5-7: IP 目录

展开 **Library** 可以看见所有用的 IP，继续展开 **Basic Functions→On Chip Memory**，双击 **RAM: 1-PORT**，即单口 RAM。

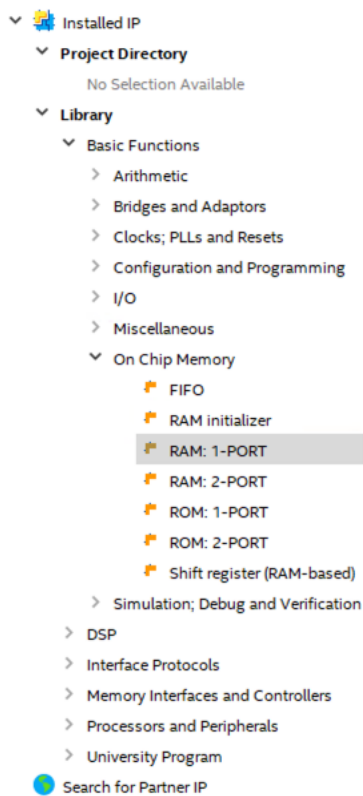


图 5-8: 选择 RAM 类型

弹出对话框，为此 IP 取一个名字，此处取名为“ram1port”，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。

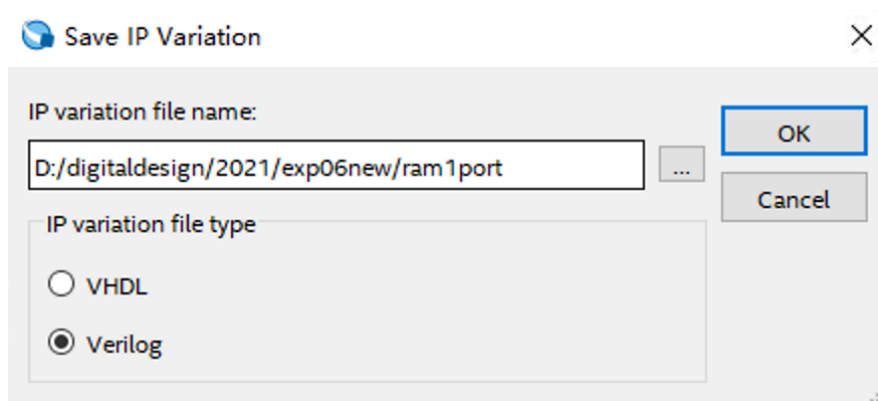


图 5-9: 选择目标文件名

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。同时我们选择了一个时钟统一控制读写。

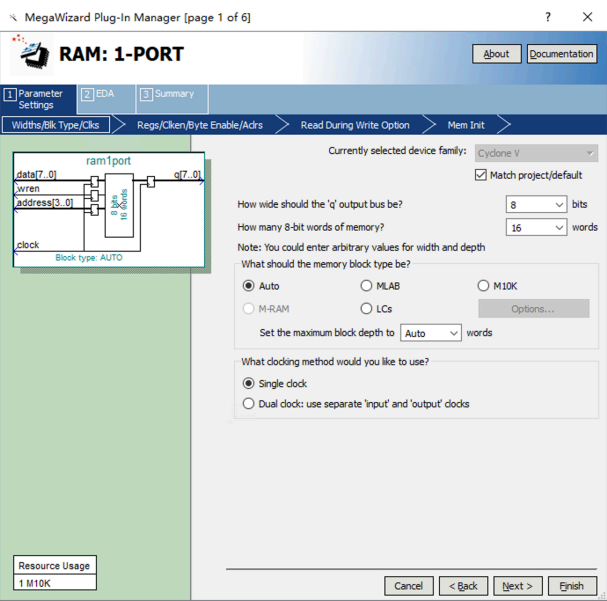


图 5-10: 选择 RAM 规模

对缓冲和使能信号等进行配置。注意我们这里没有对输出进行缓存。可以自行尝试增加输出缓存，实验 RAM 在有缓存时需要多少个时钟周期才能输出。

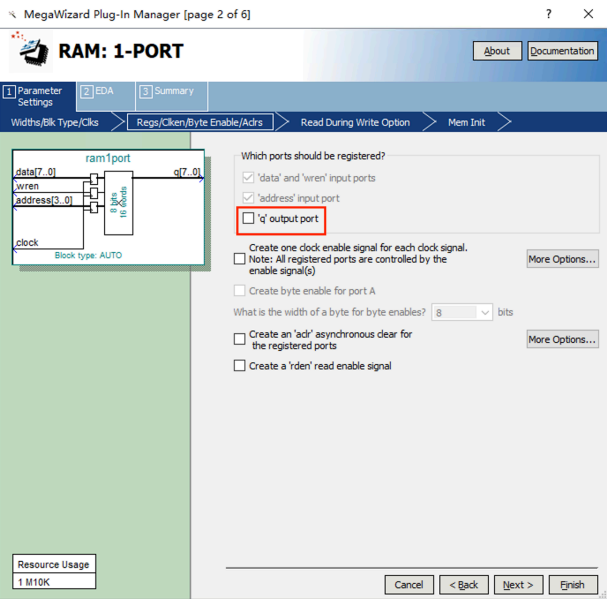


图 5-11: 信号缓存配置

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。

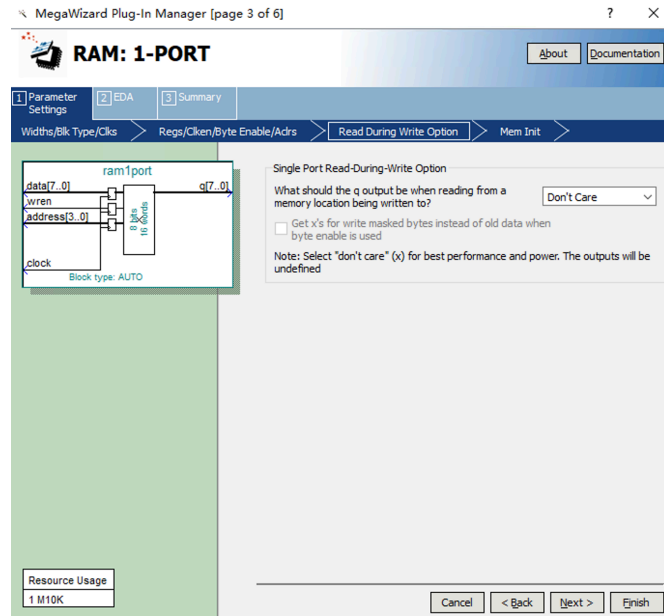


图 5-12: 读写冲突解决

5.4.2 存储器初始化

在建立存储器的时候，选择不初始化，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。在配置进行到图 5-13 时可以选择利用文件初始化内存。在此步骤中还可以配置内存动态更新，选择 Allow In-System Memory Content Editor ...，并给你的内存模块起一个合适的名字，如 RAM1。

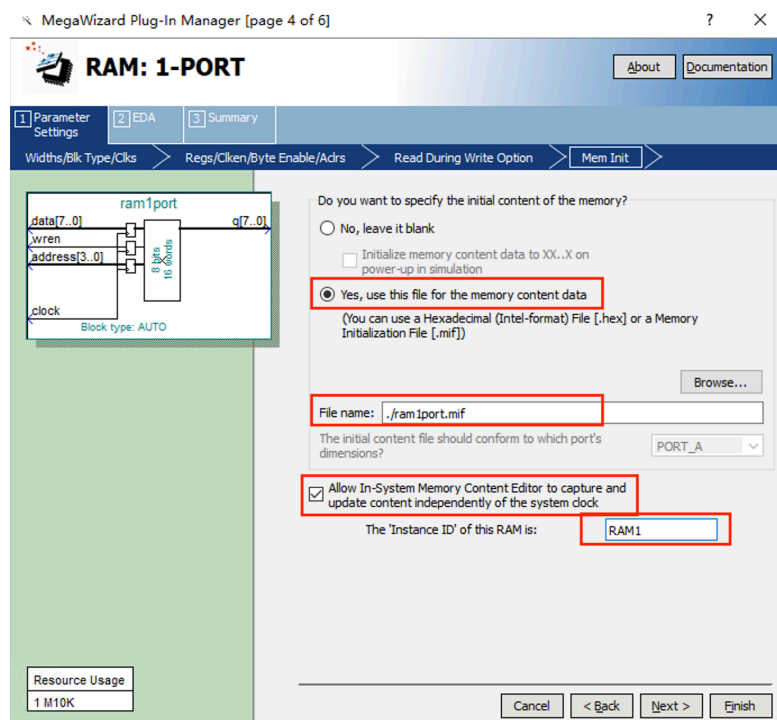


图 5-13: 初始化选择及动态内存更新选择

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：

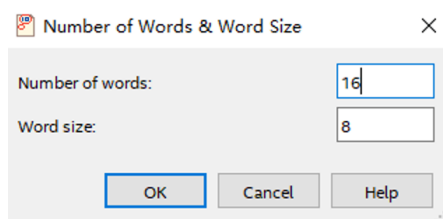


图 5-14: 初始化文件大小选择

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：

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	00	01	02	03	04	05	06	07	-----
8	08	09	10	11	12	13	14	15	-----

图 5-15: 编辑初始化文件

保存。回到 IP 核生成对话框，点击Browse...

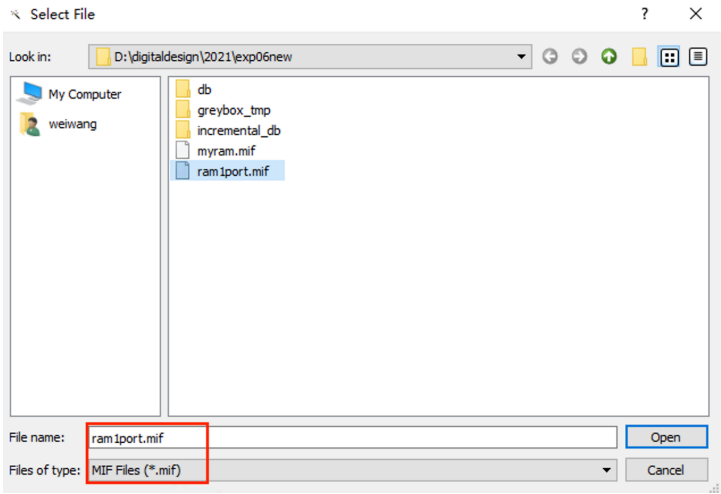


图 5-16: 初始化文件选择

选择刚刚保存的.mif 文件，点击Open，选择存储器初始化文件。点击Next，Next，Finished，完成整个单口 RAM 的配置。

在项目导航栏，Files目录下，展开ram1port.qip，可以看见为此 RAM 生成的ram1port.v 文件，双击打开，可以看见此 ram1port.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 ram1port my_ram(  
2     .address(addr),  
3     .clock(clk),  
4     .data(din),  
5     .wren(we),  
6     .q(dout0));
```

✎ 利用 mif 文件初始化非 IP 核存储器

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];
```

5.4.3 存储器动态更新

Quartus 提供了 In-System Memory Content Editor 来实时观察和更新 RAM 中的内容。这对我们 Debug 是非常有用的。尤其是在 CPU 实验中，如果 CPU 设计没有改变，但是需要对系统中运行的汇编代码进行修改时，可以直接用 In-System Memory Content Editor 来改变，不需要重新编译整个工程。

在使用 In-System Memory Content Editor 之前，请先完整编译整个工程，并确保之前生成 IP 核时勾选了动态内存更新的功能。在完成对开发版编程并运行后，打开 Quartus 的 [Tools→In-System Memory Content Editor]。进入如图 5-17 所示界面。此时，需要首先选择硬件，连接开发板。然后会自动扫描 JTAG，请选择第二个 Device。这时左边应该出现了刚刚配置过的 RAM1 的标识，但是数据尚未更新。

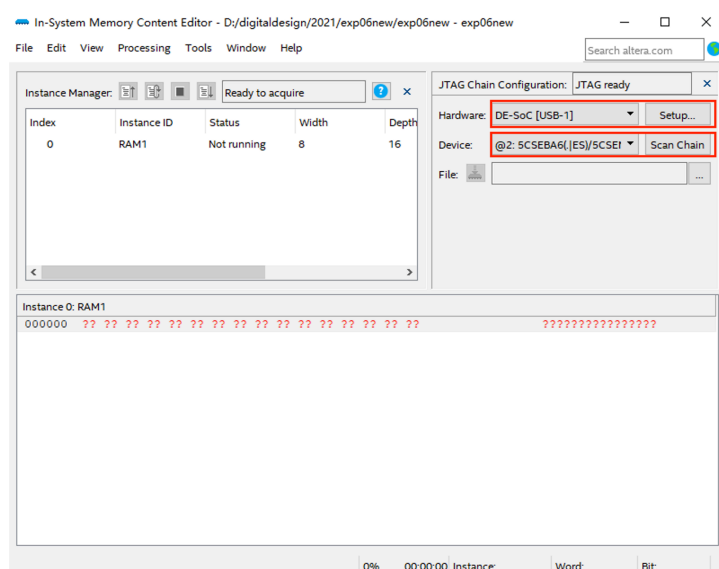


图 5-17: 内存查看器配置

在 RAM01 上右击鼠标键，选择菜单中的 Read Data ..., 可以看到 RAM 中最新的数据。

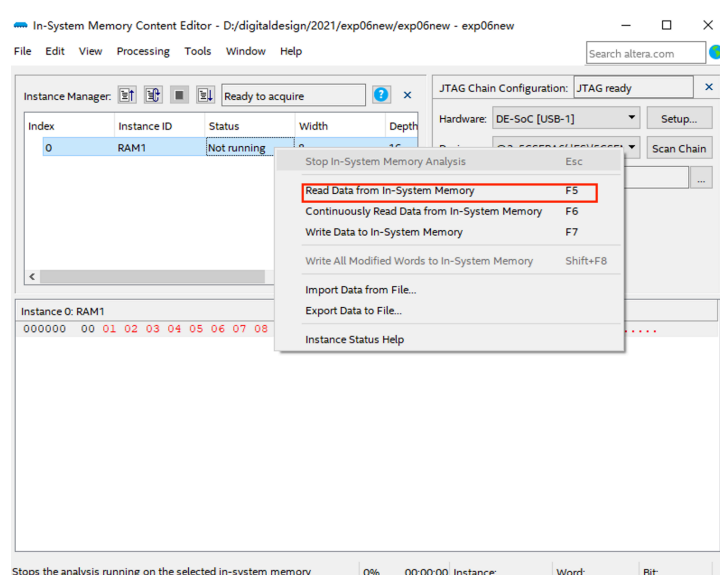


图 5-18: 读取实时内存

如果需要改变 RAM 中的数据，可以直接手动在数据上修改，也可以右键选择 Import Data from file，用新的 mif 文件来更新。在更新后请注意要右键选择 Write data to In-System Memory，让更新生效。

5.5 实验内容

5.5.1 上板验收

请在工程完成如下的寄存器堆和 RAM。寄存器堆和 RAM 的大小均为 16×8 ，即都有 16 个存储单元，每个存储单元都是 8 位的，均可以进行读写。

寄存器堆：读取时不需要时钟控制，即读地址有效后，直接输出数据。写入时通过时钟上升沿进行控制。

此时可用以下方式输出：

```
1 assign out = ram[addr];
```

采用下面的方式进行初始化。

```
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

```

✎ **RAM:** 利用 IP 核设计一个单口存储器，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。如果 IP 核不支持最小 16 单元的单口存储器，可以使用 32 单元或 64 单元的单口存储器替代，地址高位置零来只用 16 个 RAM 单元。

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。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。请将两个存储器读出的结果分别用 2 个七段数码管显示。请合理使用 FPGA 开发板的输入/输出资源，完成此寄存器堆和 RAM 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。

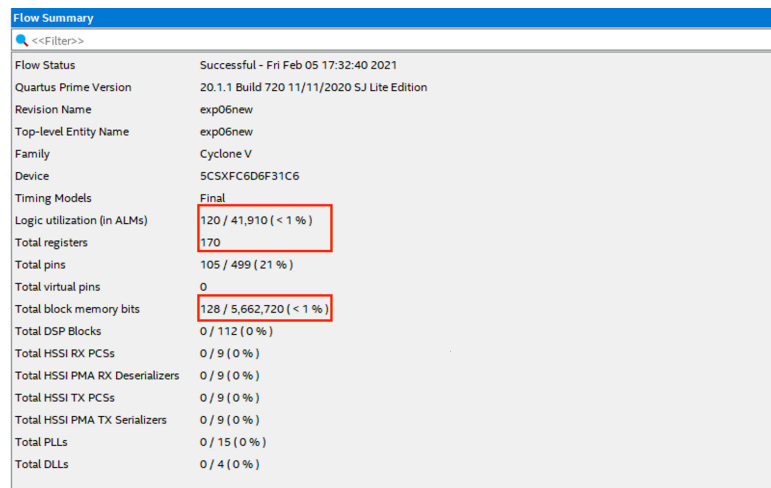
✎ 请使用 In-System Memory Content Editor 来修改 RAM 中的数据，验证你的修改确实更新到开发板上了。

✎ 请使用开发板上的按钮来做为存储器的时钟信号。观察两个不同的实现方式下各需要几个时钟周期来完成读取或写入操作？

✎ 打开 `Tools→Netlist Viewers→Technlogy Map Viewer`，点开实现的树形结构寻找到你生成的两个存储器，观察综合后这两个存储器分别使用了什么方

式来实现，为什么？

注意观察综合后输出的资源消耗情况，图 5-19 中两个红框部分消耗的资源可能是由哪个存储器产生的？如果用寄存器方式，我们用的开发板大约可以支持多大容量的存储？用 Block Memory 呢？



Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri Feb 05 17:32:40 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	exp06new
Top-level Entity Name	exp06new
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	120 / 41,910 (< 1 %)
Total registers	170
Total pins	105 / 499 (21 %)
Total virtual pins	0
Total block memory bits	128 / 5,662,720 (< 1 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

图 5-19: 系统资源消耗

5.5.2 在线测试

必做 寄存器堆实现

选做 前导零个数判断