MIPSfpga
by Imagination

Lab5 存储映射 I/O

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

Lab5 存储映射 I/O

1. 引言

在这次实验中，你将了解存储映射的输入和输出（I/O）。首先我们会向你展示如何从硬件上扩展 MIPSfpga 系统的 7-段数码管显示接口，这样程序员就能通过这个接口控制 Nexys4 DDR 板卡上 7-段数码管的显示。然后我们会通过一小段汇编代码来仿真测试所添加的硬件。同时我们还提供了 MIPS 汇编和 C 语言的数码管显示实例程序。最后你将写一个自己的 7-段数码管显示 C 程序。

2. MIPSfpga 存储映射 I/O 教程

在这个教程中，你将了解到处理器是如何通过存储器接口来和外围设备（Nexys4 DDR FPGA 板上的拨码开关、LED 灯、7-段数码管显示器）进行交互的。存储映射 I/O 允许处理器像读写内存那样读写外围设备，比如说让 7-段数码管显示器显示数据。每个外围设备被分配一个或者多个内存地址。当处理器访问这些地址的时候访问到的是外围设备而不是内存。MIPSfpga 系统通过 AHB-Lite 总线连接到外部存储器上。

AHB-Lite 总线

AHB-Lite 总线是 MIPSfpga 系统和外部存储器的接口，总线包含了时钟信号、写使能信号、地址信号、读写数据信号（HCLK, HWRITE, HADDR, HRDATA, and HWDATA），如图 1 所示。“H”前缀说明它们是 AHB-Lite 总线的一部分。内存和外设都通过这个接口接收和提供数据。MIPSfpga 核负责把下面这些信号送到 AHB-Lite 总线上。

- **HCLK**：62MHz 系统时钟
- **HWRITE**：写使能（1 为写，0 为读）
- **HADDR**：读写地址
- **HWDATA**：写操作时被写的数据

MIPSfpga 核通过下面这个 AHB-Lite 总线信号获取数据。

- **HRDATA**：从内存或外设中读到的数据

MIPSfpga 系统的 AHB-Lite 总线连接了 3 个模块：两个内存模块（RAM0 和 RAM1）和一个通用 I/O 模块（GPIO）。RAM0 中存的是引导代码，RAM1 中存的是用户代码和数据。GPIO 模块连接到 Nexys4 DDR 板卡上的 LED 灯、拨码开关和按钮。

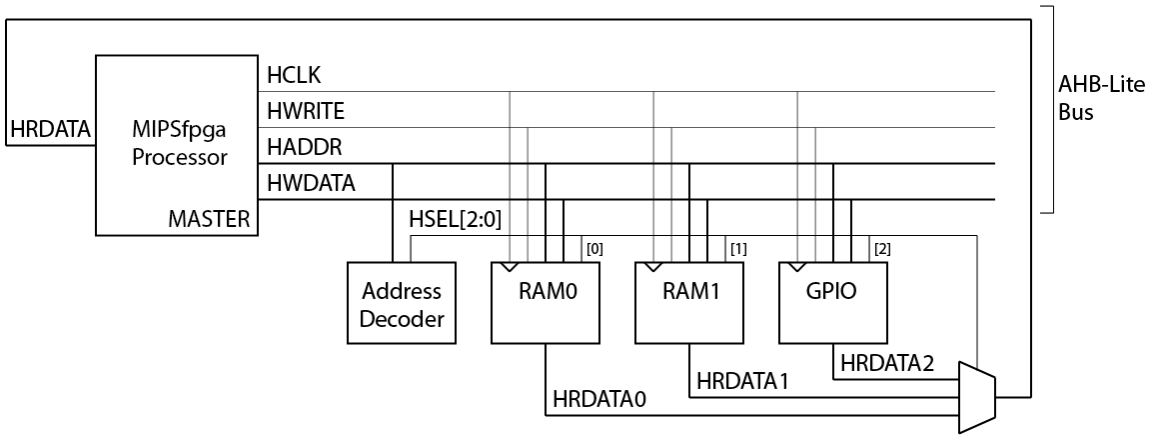

图 1. MIPSfpga 处理器和三个外围设备

除了三个外围设备外，存储映射 I/O 接口还需要一个地址译码器和一个多路选择器。地址译码器根据处理器给出的地址（HADDR[31:0]）进行判断，从而确定处理器对三个外围设备中的哪一个进行访问。地址译码器会产生一个选择信号 HSEL[2:0] 来供外围模块和 3:1 多路选择器使用。

RAM0 中存放的是引导代码（虚拟地址 0xbfc00000-0xbfc1fffc = 物理地址 0x1fc00000-0x1fc1fffc）。RAM1 中存的是用户代码（虚拟地址 0x80000000-0x8003fffc = 物理地址 0x00000000-0x0003fffc）。Nexys4 DDR 板卡上的 LED 灯、拨码开关和按钮映射到虚拟地址 0xbf800000-0xbf80000c，如表 1 所示。处理器的代码使用的是虚拟存储地址而 AHB-Lite 总线上接收的是物理地址。MIPSfpga 核中的内存管理单元（MMU）负责完成这一地址转换工作。

表 1. Nexys4 DDR fpga 板卡内存地址

虚拟地址	物理地址	信号名	Nexys4 DDR
0xbf80 0000	0x1f80 0000	IO_LEDR	LEDs
0xbf80 0008	0x1f80 0008	IO_SW	switches
0xbf80 000c	0x1f80 000c	IO_PB	U, D, L, R, C pushbuttons

下面的汇编指令代码写一个值 5 到 LED 灯上：

```
lui $8, 0xbf80    # $8 = 0xbf800000 (address of LEDs)
addi $9, $0, 5          # $9 = 5
sw$9, 0($8)
```

lui 指令把 16 位数 0xbf80 加载到$8 寄存器的高 16 位，低 16 位清 0。执行到 sw 指令时，HADDR = 0x1f800000，HWRITE = 1，HWDATA = 5。地址译码器发现 0x1f800000 这个地址为 GPIO 外围设备地址，然后就把外设选择信号 HSEL[2]拉高。之后 GPIO 模块就会检测到 HSEL[2]和 HWRITE 信号被拉高。由于 GPIO 能够对多个外设进行写操作，该模块会根据地址判断出 HWDATA 信号的值是传给 LED 灯的，因此一个输出连接到 LED 灯的寄存器就会被赋上 HWDATA 信号的值，这个值会一直保持到后面有指令再次写这个寄存器为止。

同样的，下面两行代码读取拨码开关的值：

```
lui $8, 0xbf80          # $8 = base address of the I/O
lw $9, 8($8)            # $9 = value of the switches
```

执行 lw 这条指令的时候，HADDR = 0x1f800008，HWRITE = 0（为 0 表示读）。地址译码器发现 0x1f800008 这个地址为 GPIO 外围设备地址然后就把 HSEL[2]拉高（保持 HSEL[1]和 HSEL[0] 为低电平）。GPIO 模块发现这个地址是拨码开关的地址然后就把拨码开关的值作为输出送到 HRDATA2 上。然后多路选择器根据 HSEL[2:0]来选择送到 HRDATA 上的数据。因为这里 HSEL[2] 已经被拉高了，多路选择器将选择 HRDATA2 中的数据送到 HRDATA 上。然后处理器读内存数据一样读取 HRDATA 上的数据并存到$9 寄存器中。这样的话，lw 指令结束之后，$9 寄存器中存的就是拨码开关的值。

MIPSfpga AHB-Lite 硬件定义在 mipsfpga_ahb 模块以及其子模块中。你最好在 Vivado 项目中查看这个模块，这样就能使得模块之间的层次关系更加清晰。打开实验 1 创建的项目（在 MIPSfpga_Wuhan_v12\Project1 目录中），在 Project Manager 窗口中展开 mipsfpga_nexys4 和 mipsfpga_sys，然后展开 mipsfpga_ahb，你就能看到 mipsfpga_ahb 模块的层次结构，如图 2 所示。双击其中任何一个就能在右边的编辑框内查看模块的 Verilog 实现。

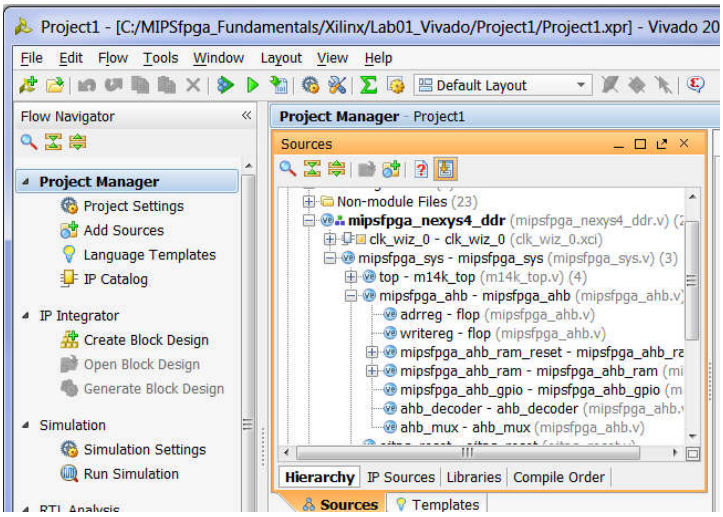

图 2. mipsfpga_ahb 层次结构

比如你可以双击 mipsfpga_ahb 来查看 AHB-Lite 总线模块的接口信号（如图 3 所示）。除了前面图 1 中出现的 AHB-Lite 信号（HCLK, HADDR, HWRITE, HWDATA, 和 HRDATA）外，如果你需要，还可以添加其他 AHB-Lite 信号。mipsfpga_ahb 模块除了 AHB-Lite 信号外还有存储映射 I/O 信号 IO_Switch、IO_PB、IO_LEDR，这几个信号分别接到 Nexys4 DDR 板卡的拨码开关、按钮、和 LED 灯。IO_LEDG 没有连接到 Nexys4 DDR 板卡上。

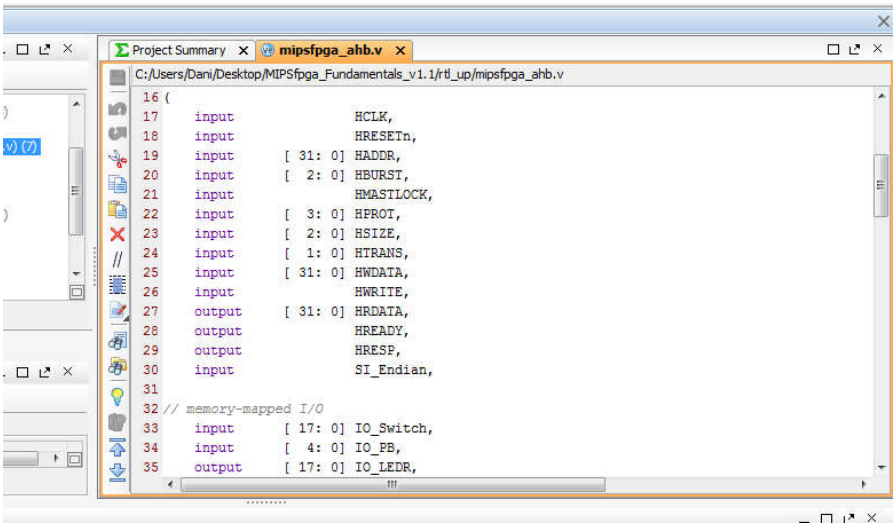

图 3.mipsfpga_ahb 接口信号

在 mipsfpga_ahb 中实例化的模块是 3 个外设、地址译码器和多路选择器，如图 1 所示。相应的 Verilog 模块名称在表 2 中列出。查看 Verilog 代码看如何实现上述功能。

表 2. AHB-Lite 模块

图 1 中的名称	模块名称
RAM0	mipsfpga_ahb_ram_reset
RAM1	mipsfpga_ahb_ram
GPIO	mipsfpga_ahb_gpio
Address Decoder	ahb_decoder
Multiplexer (for HRDATA)	ahb_mux

GPIO 模块（mipsfpga_ahb_gpio）和 Nexys4 DDR 板卡的通用 I/O 相连接。MIPSfpga 系统包含了对 LED 灯、拨码开关和按钮这三个外围设备的访问。在本实验和下一个实验中，我们会从 Nexys4 DDR 板上的 8 个 7-段数码管显示器开始，向你展示如何扩展 MIPSfpga 系统的外围设备。

7 段数码管显示器

如图 4 所示，我们可以使用 7-段数码管显示器来显示数字。这 7 个段分别用 a 到 g 表示。图 5 中列出了数码管显示 0 到 F 时点亮的段。比如说在显示数字 0 的时候，除了中间的 g 段外其他的段都被点亮了。

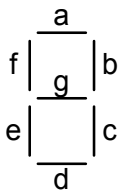

图 4. 7-段数码管

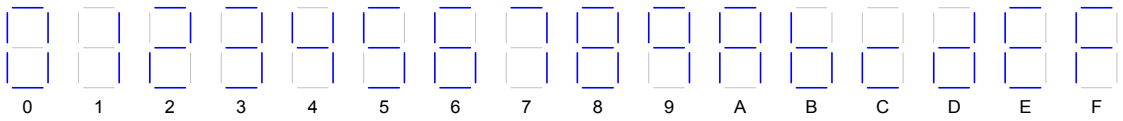

图 5. 7-段数码管显示功能

给定输入数字的范围为 0x0-0xF，我们将向你展示如何扩展 MIPSfpga 系统来在 7-段数码管上显示给定的数字。数码管上的段都是低电平有效的，为 0 时才会亮。

下面（表 3）是 7-段数码管显示译码器的输入输出真值表，译码器接收 4 位输入信号（4 位表示数字 0-15），输出为对应 4 位数字的数码管段信号。比如说输入的是“0”，要想在数码管上显示 0 这个数字就应该点亮除了 g 段之外的所有段，所以真值表中的第一行的

输出为除了 S_g 之外都为 0（别忘了数码管的段都是低电平有效的，为 0 的时候才会亮）。当输入为数字 1 的时候只需要点亮 S_b 和 S_c 这两个段，所以这一行的输出只有 S_b 和 S_c 这两个是 0。

表 3. 7 段数码管显示译码器真值表

十六进制数	输入				输出							
	D_3	D_2	D_1	D_0	S_a	S_b	S_c	S_d	S_e	S_f	S_g	HEX
0	0	0	0	0	0	0	0	0	0	0	1	01
1	0	0	0	1	1	0	0	1	1	1	1	4f
2	0	0	1	0	0	0	1	0	0	1	0	12
3	0	0	1	1	0	0	0	0	1	1	0	06
4	0	1	0	0	1	0	0	1	1	0	0	4c
5	0	1	0	1	0	1	0	0	1	0	0	24
6	0	1	1	0	0	1	0	0	0	0	0	20
7	0	1	1	1	0	0	0	1	1	1	1	0f
8	1	0	0	0	0	0	0	0	0	0	0	00
9	1	0	0	1	0	0	0	1	1	0	0	0c
A	1	0	1	0	0	0	0	1	0	0	0	08
B	1	0	1	1	1	1	0	0	0	0	0	60
C	1	1	0	0	1	1	1	0	0	1	0	72
D	1	1	0	1	1	0	0	0	0	1	0	42
E	1	1	1	0	0	1	1	0	0	0	0	30
F	1	1	1	1	0	1	1	1	0	0	0	38

建立 7-段数码管译码器

写一个 Verilog 模块来描述 7-段数码管显示译码器的硬件实现。这个模块的声明在下列文件中已经提供给你：mipsfpga_ahb_sevensegdec.v

这个模块有一个 4 位输入（data[3:0]）和一个 7 位的输出（segments[6:0]）对应 a~g 的每个段。使用 XSIM 在仿真中测试你的硬件，如果需要的话就请做调试。在下一步，会使用这个模块驱动在 Nexys4 DDR 板上的 7-段数码管显示器。

Nexys4 DDR 板上的 7-段数码管显示器

Nexys4 DDR 板卡上的所有 8 个 7-段数码管显示器的段都是连接到同一组低电平触发的引脚上，他们被称为 CA、CB、CC、…、CG。但是这 8 个数码管有它们各自的使能信号，使能信号同样是低电平触发的。Figure 6 展示了 Nexys4 DDR 板卡上的 8 个 7-段数码管显示器。CA 接到这 8 个数码管中每一个数码管 A 段的负极,CB 接到这 8 个数码管中每一个数码管 B 段的负极，以此类推。每一个数码管都有一个使能信号 AN[7:0]。AN[7:0]通过一个反相器接到对应数码管的每一个段的正极上。比如说，只有到 AN[7]为 0 的时候，数码管 7 的显示才会受到 CA…CG 这几个信号的驱动。

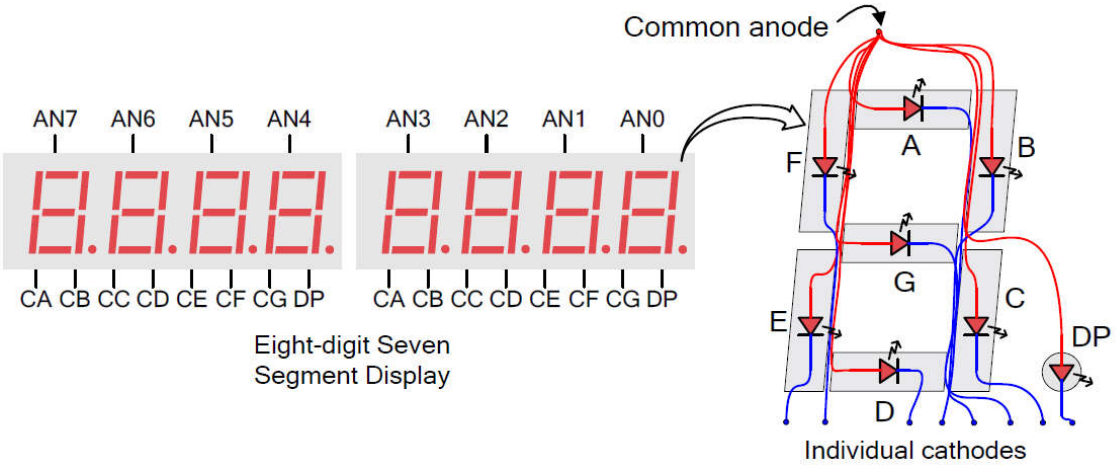

图 6. Nexys4 DDR 板上的 8 个 7-段数码管显示器

(©Nexys4 DDR Reference Manual)

要想让每个数码管显示不同的数字，使能信号（AN[7:0]）和段信号（CA…CG）必须依次地被持续驱动，数码管之间的刷新速度应该足够快这样就看不出来数码管之间在闪烁。举个例子，如果我们想在数码管 0 上显示数字 3 而数码管 1 上显示数字 9，我们可以先把 CA…CG 设置为显示数字 3，然后拉低 AN[0]信号，然后再把 CA…CG 设置为显示数字 9 然后拉高 AN[0]拉低 AN[1]。我们可以把刷新频率设置为 2ms 刷新一次，这样人眼就看不出闪烁了。

建立 HDL 模块来驱动 Nexys4 DDR 7-段数码管显示器

现在你要写一个 Verilog 模块来驱动 Nexys4 DDR 板上的 8 个 7-段数码管显示器。模块的声明在下列文件中提供：mipsfpga_ahb_sevensegtimer.v

这个模块接收要在 8 个数码管的每一个上显示的数字（DISP0[3:0] – DISP7[3:0]）和一个指示 8 个数码管的哪一个应该显示的使能信号（EN[7:0]）。它也接收 50MHz 时钟（clk）和

一个低电平有效的复位信号（resetn）作为输入。输出是是 8 个 7-段数码管显示器的使能信号(DISPENOUT[7:0])和 7 段的值 A-G (DISPOUT[6:0])。在后面的实验中，你将通过顶层模块(mipsfpga_nexys4_ddr)连接这些输出来让它们驱动 8 个显示器的使能信号(AN[7:0])和 7 段的引脚(CA…CG)。

你的模块需要连续地每隔大约 2ms 依次驱动 8 个中的一个数码管的使能信号。你要用你上面写的 7-段数码管显示译码器。注意，如果需要，你可以扩展这个模块的功能来包含小数点的显示。使用 XSIM 在仿真中测试你的硬件，如果需要的话就请做调试。

在 GPIO AHB-Lite 模块中添加 7-段数码管显示功能

现在你已经写了一个能够控制这 8 个 7-段数码管显示器的硬件模块，接下来就是在 MIPSfpga 系统上添加显示器的接口功能。你的目标是允许用户用 SW 指令来写 8 个 7-段数码管显示器。从下面的步骤开始：

1. 为数码管使能信号和显示的 8 位数字分配内存地址
2. 修改 GPIO 模块使其能够识别上面所分配的地址并把这些地址跟存储映射 I/O 寄存器关联起来。
3. 把存储映射 I/O 寄存器接到你刚创建的 mipsfpga_ahb_sevensegtimer 模块中。

为了完成上面这些改动，你需要修改下面这几个模块：

- mipsfpga_ahb_const.vh
- mipsfpga_ahb_gpio.v

下面针对上面步骤的一些指导

1) 分配存储映射 I/O 地址

我们给 7-段数码管显示器分配了 9 个地址，其中一个是使能信号地址，另外 8 个地址分别接收每个数码管的值，如表 4 所示。用户将写这些地址来设置使能信号和数字值。

表 4. Nexys4 DDR FPGA 7 段数码管显示器存储地址

虚拟地址	物理地址	信号名	Nexys4 DDR
0xbf80 0010	0x1f80 0010	SEGEN_N[7:0]	AN[7:0]
0xbf80 0014	0x1f80 0014	SEG0_N[3:0]	Digit 0 value
0xbf80 0018	0x1f80 0018	SEG1_N[3:0]	Digit 1 value
0xbf80 001c	0x1f80 001c	SEG2_N[3:0]	Digit 2 value
0xbf80 0020	0x1f80 0020	SEG3_N[3:0]	Digit 3 value
0xbf80 0024	0x1f80 0024	SEG4_N[3:0]	Digit 4 value

0xbf80 0028	0x1f80 0028	SEG5_N[3:0]	Digit 5 value
0xbf80 002c	0x1f80 002c	SEG6_N[3:0]	Digit 6 value
0xbf80 0030	0x1f80 0030	SEG7_N[3:0]	Digit 7 value

为了定义这些地址，我们首先需要修改的是 mipsfpga_ahb_const.vh 这个 Verilog 头文件。在 Vivado 中打开项目 Project1。在 Project Manager 窗口中找到 Design Sources → Verilog Header，然后双击打开 **mipsfpga_ahb_const.vh**（如图 7 所示）。

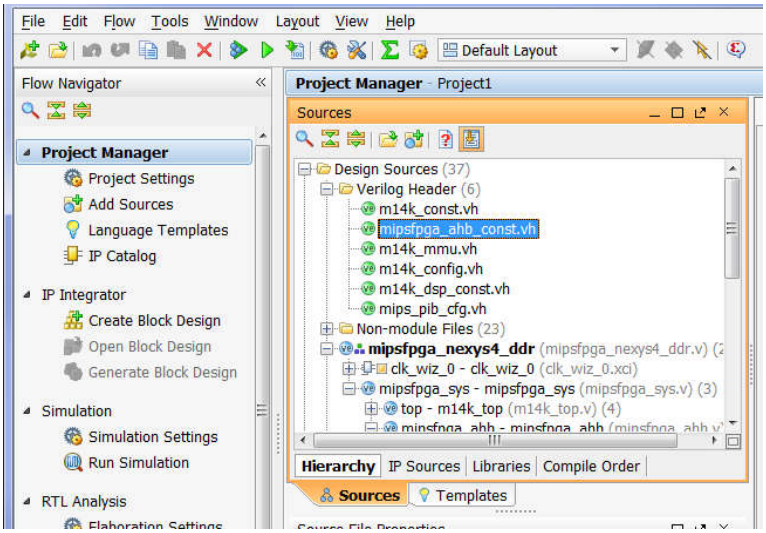

图 7. mipsfpga_ahb_const.vh Verilog 头文件

数码管的地址定义为 H_7SEGEN_ADDR, H_7SEG0_ADDR, … H_7SEG7_ADDR。地址译码器（ahb_decoder 模块）根据地址高位判断应该使能三个 AHB 从设备（复位 RAM，程序 RAM 和 GPIO 模块）中的哪一个。然后， GPIO 模块一旦被选中就会根据地址的低位判断应该对哪一个外设进行写或读。在 mipsfpga_ahb_const.vh 文件下，我们用 H_*_IONUM 来表示存储映射 I/O 地址的 5:2 位。如下所示：

```
`define H_LEDR_IONUM                    (4'h0)
`define H_LEDG_IONUM                    (4'h1)
`define H_SW_IONUM                  (4'h2)
`define H_PB_IONUM                  (4'h3)
```

比如拨码开关映射到物理地址 0x1f800008，因此位 5:2 是 0x2（也就是 H_SW_IONUM 是 4'h2）。

将 7-段数码管显示器变量的 I/O 号命名为：H_7SEGEN_IONUM, H_7SEG0_IONUM,等。比如当 7 段数码管的使能信号的地址为 0x1f800010 时，表示地址的 5:2 位的 H_7SEGEN_IONUM 就为 0x4。

2) 修改 GPIO 模块

现在来修改 GPIO 模块用以用于检测你刚刚定义的 9 个地址，当检测到相应的地址时，就将数据(HWDATA)写到那些寄存器中。在 Vivado 的工程中打开 **mipsfpga_ahb_gpio.v**。在模块声明中，声明和输出了使能和段信号(A-G)。分别明明这些信号为 IO_7SEGEN_N[7:0]和 IO_7SEG_N[6:0]。在高一层的模块中，这些信号将分别驱动 Nexys4 DDR 板上的 7-段数码管的使能信号（AN[7:0]）和段信号（CA…CG）。

你现在必须创建 9 个寄存器来保存使能信号的值和要显示到 8 个数码管显示器中的数字的值。用户将通过存储映射 I/O 来写这些寄存器。命名保存使能信号的寄存器 SEGEN_N[7:0]和保存 8 个数的 8 个 4 位寄存器 SEG0_N[3:0], …,SEG7_N[3:0]。修改 GPIO 模块以便在正确的地址被检测到的时候这些寄存器被写入。

3) 连接寄存器到你刚创建的 mipsfpga_ahb_sevensegtimer 模块

现在在 GPIO 模块中实例化和连接之前在本实验中创建的 mipsfpga_ahb_sevensegtimer 模块。你将连接它的输入到存储器映射 I/O 寄存器（包括时钟和复位信号）。它的输出连接到 7-段数码管显示器信号(IO_7SEGEN_N and IO_7SEG_N)。

连接 7-段数码管显示器信号到 Nexys4 DDR 板

现在你将从 AHB GPIO 模块连接输出信号去驱动 Nexys4 DDR 板上的 7-段数码管显示器。将从 GPIO 模块中来的输出信号(IO_7SEGEN_N and IO_7SEG_N)沿着项目的层次逐步向上传递，直到到达 Nexys4 DDR 板上（也就是 mipsfpga_nexys4 模块的输出）。为了达到这个目的，你将需要修改下面的模块。

```
mipsfpga_ahb.v
mipsfpga_sys.v
mipsfpga_nexys4_ddr.v
```

在最高层模块(mipsfpga_nexys4.v)，命名那些驱动 7-段数码管显示器的输出信号为 CA, CB, CC, CD, CE, CF, CG 和 AN[7:0]。CA…CG 驱动段而 AN[7:0]驱动使能信号。

你还需要修改 Xilinx 设计约束文件(.xdc)。在 Vivado 中，从工程中通过展开 Project Manager 窗口中的 Constraints →constrs_1 并双击 mipsfpga_nexys4_ddr.xdc 打开这个文件

（如图 8 所示）。XDC 文件分配了信号名（AN[7:0]和 CA – CG）到 Artix-7 FPGA 的引脚，使它们通过导线在物理上连接到了 Nexys4 DDR 板上的 7-段数码管显示器的输入。

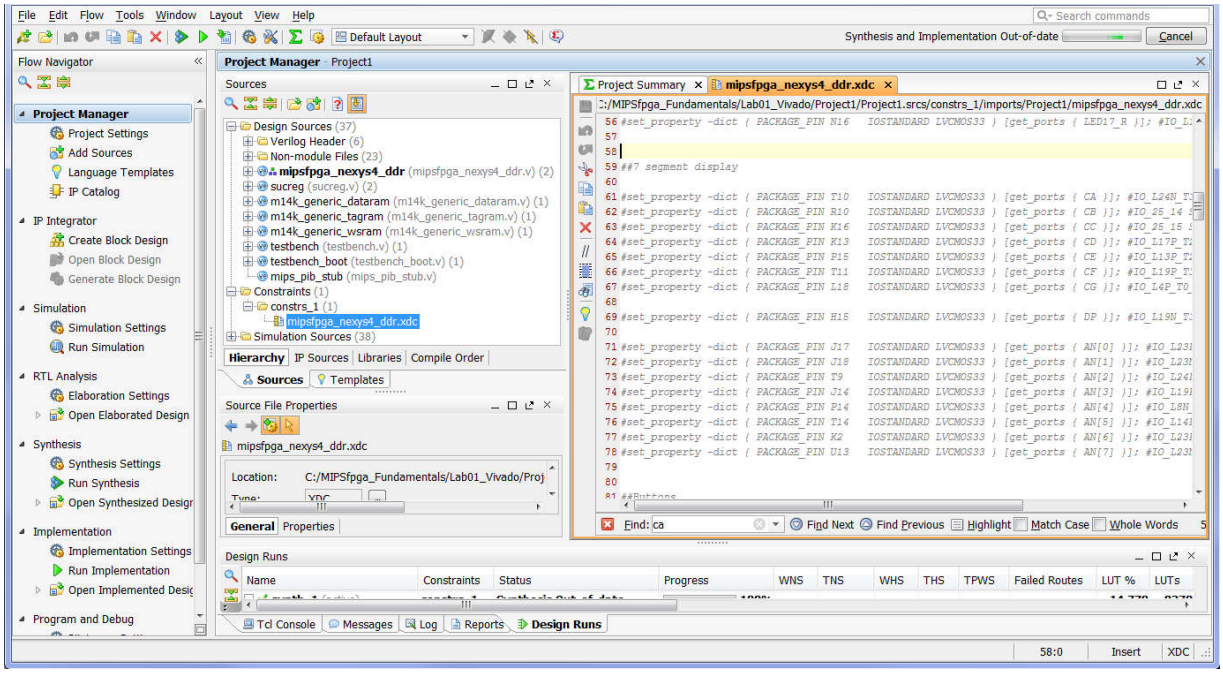

图 8. 打开 Xilinx 设计约束文件

在 Vivado 中打开的.xdc 文件中搜索(ctrl-F) “segment”来寻找 7-段数码管的输出信号列表，如图 8 所示。在这个文件中已经可以看到有关哪些引脚被连接到 7-段数码管显示器的信息，只需要去掉她们的注释。将你所需要那行之前的#删除掉。比如，通过下面的句子，CA 这个信号驱动连接在 Artix-7 FPGA 的引脚 T10 上的 7-段数码管的 A 段连接起来。

```
set_property -dict { PACKAGE_PIN T10    IOSTANDARD LVCMOS33 } [get_ports
{ CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
```

这个引脚被用导线连接到 Nexys 4 DDR 板上的 7-段数码管的 A 段输入。CB 就和 Artix-7 的 R10 引脚连接，以此类推。驱动 7-段数码管显示器阳极的信号(AN[7:0]，使能信号)同样也要和 Artix-7 的引脚接起来。DP 信号这一行被注释掉了(#)，因为我们这个实验没有用到它。

在约束文件的底端为 7-段数码管显示器信号添加下面的输出时钟约束：

```
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports
{AN[*]}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports
{AN[*]}]
```

```
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CA}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CA}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CB}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CB}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CC}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CC}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CD}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CD}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CE}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CE}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CF}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CF}]
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports {CG}]
set_output_delay -clock "clk_virt" -max -add_delay 10.000 [get_ports {CG}]
```

测试 7-段数码管显示功能

现在我们要编写一个简单的写 7-段数码管显示器的汇编程序来测试我们前面添加的 7-段数码管硬件的基本功能。然后我们使用 Xilinx 内建模拟器 XSIM 来仿真简单的程序（不使用 boot 代码）。

写一个简单的 MIPS 汇编程序来使能 7-段数码管显示器并写 8 个数字的每一个的值。修改 main.S 文件。然后用 make 编译程序，如果需要，也可以调试。

为仿真而提取机器码

为了仿真指令在 MIPSfpga 系统上运行，我们需要把 MIPS 汇编指令转换成机器码。你可以用你喜欢的任何方法将汇编程序转成 32 位机器码，比如你可以手动把汇编代码转换成机器码，或者使用其他的模拟器（比如说 QtSpim）来生成机器码，或者从 Codescape 产生的可执行文件中提取出机器码。在这里我们教你两种方法来使用 Codescape 把 MIPS 汇编代码转换成机器码，这两种方法都需要首先使用 Codescape 把 MIPS 汇编代码编译一遍（就如实验 2-4 和《MIPSfpga 入门指南》的 7.2 节中所描述的那样）。第一种方法需要手动从可执行文件中提取初机器码，第二种使用脚本来提取机器码。对于简单的程序，第一种方法比较快。

方法 1：手动从可执行文件中提取机器码

首先我们向你展示如何手动的从 Codescape 编译器生成的可执行文件中提取机器码：

make 编译完之后会生成好几个文件，打开其中的 **FPGA_Ram_dasm.txt** 或者 **FPGA_Ram_modelsim.txt** 这两个可执行文件的反汇编文件。FPGA_Ram_modelsim.txt 里面没有交叉的高级程序语言，读起来会更轻松一点，但是里面包含的信息就相对较少。注意，如果这两个文件不存在，则你还需要用 make 来编译程序。在文件中搜索用户程序的开始标号 main。提取这些机器指令并把它们复制到 **ram_reset_init.txt** 文件中。

我们将用这个文件来初始化 boot RAM，boot RAM 的地址是从物理地址 0x1fc00000 开始的。当我们执行仿真的时候会根据这个文件来把 boot RAM 初始化为文件里面的指令。复位之后，MIPSfpga 将会从第一条指令开始执行。注意上面这种“重定位”的方法只有在代码中没有跳转指令时才管用。

方法 2：使用脚本自动提取机器码

现在我们向你展示如何使用脚本来完成方法 1 中手动完成的工作。打开命令窗口，修改目录到 Codescape_Scripts 下，运行 createMemfiles 批处理脚本。

这个脚本会把引导代码和用户代码的机器码分别放到 boot RAM（ram_reset_init.txt）和 program RAM（ram_program_init.txt）。你可以使用这些文件来仿真整个程序（boot 代码和用户代码）。但在这里我们并不需要对整个程序进行仿真，只需仿真用户代码就足够了，因为我们只是想测试一下 7-段数码管的硬件是否正确，所以这里我们只把用户代码提取出来放到 boot RAM 中就可以了，这样的话系统在复位之后就能直接运行测试程序。

提取过程需要 10 分钟或者更长的时间，这取决于你电脑的运行速度。完成之后会在命令窗口中提示提取完成。然后进入到下面这个文件夹，脚本生成的文件就在 MemoryFiles 目录中。

下面四个文件就是脚本运行后生成的文件：

- ram_reset_init.txt
- ram_program_init.txt
- ram_reset_init.mif
- ram_program_init.mif

.txt 文件包含了 reset（boot）和 program RAM 的存储内容。ram_reset_init.txt 中存的是引导代码，指令从物理地址 0x1fc00000 开始存放（虚拟地址 0xbfc00000 或 0x9fc00000），系统复位之后会把这个地址赋值给 PC。ram_program_init.txt 中存的是从物理地址 0x00000000（虚拟地址 0x80000000）开始的用户程序指令。.mif 文件被称为*存储器初始化文件（memory initialization files）*，它是被用于某些 CAD 工具中的对存储单元内容的描述，这种描述方法和.txt 文件的描述方法有些许不同。

打开 FPGA_Ram_dasm.txt 和 ram_program_init.txt。在 FPGA_Ram_dasm.txt 里面搜索“80000000”。这个地址上的代码是异常处理程序的一部分，如下所示。

```
80000000:     3c1b8000    lui   k1,0x8000
```

```
80000004:     277b1430    addiu k1,k1,5168

   ...
80000220:     3c1b0000    lui   k1,0x0
80000224:     277b0000    addiu k1,k1,0
   ...
```

在描述从虚拟地址 0x80000000（物理地址 0x0）开始的用户程序代码的 ram_program_init.txt 文件中，其顶部列着下列这些指令：

```
@0
3c1b8000
277b1f34
...
```

ram_program_init.txt 文件中列出从物理地址 0x0 开始存放的指令（@0 表明从物理地址 0 开始）。MIPS 使用的是字节编址内存，但 256KB 的 program RAM 中每个地址对应的是 32 位的字数据（也就是它采用的是 2^{16} x 32 位的结构）。所以内存模块直接丢弃 AHB-Lile 总线上传输地址的低 2 位，本质上就是将地址除以 4。因此物理地址 0x0000005c（虚拟地址为 0x8000005c）在 program RAM 中的地址为 0x5c/4 = 0x17。

ram_program_init.txt 文件更下面一点就能看到存放 RAM1，地址为 0x88 上的如下指令。

```
@88
3c1b0000
277b0000
...
```

这就相当于 FPGA_Ram_dasm.txt 中从 0x80000220 开始的指令（0x80000000+0x88*4）。

在 ram_program_init.txt 文件里面搜索“@1d7”。这些都是地址 0x80000000 + (0x1d7*4) = 0x8000075c 上的指令，也就是用户程序代码的开始位置。在 FPGA_Ram_dasm.txt 文件中搜索“8000075c:”来查看整个程序的指令。现在在 FPGA_Ram_dasm.txt 和 ram_program_init.txt 找到你的程序代码的末尾。在下一步，你将从 ram_program_init.txt 中提起程序代码，并将它们放到新的 ram_reset_init.txt 文件中，这个文件就是你将要仿真的文件。

现在你将提取你程序的机器码并将它们重定位到虚拟地址 0xbfc00000，这样你的代码在复位后就会被仿真（而不是仿真原来的引导代码）。为了达到这一步，你要创建一个新的 **ram_reset_init.txt 文件**，让其包含你刚才位于 ram_program_init.txt 中的程序代码。去掉

这个文件中的所有地址指示符（@1d7 等）。没有这些地址指示符的情况下默认从相对于引导代码开始的地址 0 开始存放这些指令（开始于虚拟地址 0xbfc00000）。

仿真的时候你需要用刚刚创建的 ram_reset_init.txt 这个文件来初始化 reset/boot RAM(RAM0)。再次提醒一下，只有代码中没有跳转指令时我们才可以重定位代码。

仿真

接下来我们就要在 MIPSfgpa 系统上仿真运行用户程序来测试前面添加的 7-段数码管显示硬件。在仿真之前，你需要添加你的新的和修改过的 Verilog 文件到你在实验 1 中创建的项目中。

打开实验 1 创建的 Vivado 项目，添加你在本实验的前面创建的 7-段数码显示译码器和 7-段数码显示时钟模块（通过在 Vivado 里的 Project Manager 窗格下选择 Add Sources）。

现在你将准备仿真你的新硬件。在 Project Management 窗口中下拉滚动条找到并展开 Simulation Sources 和 sim_1。确保 testbench 是粗体，表明已经设置成仿真顶层文件。如图 9 所示。

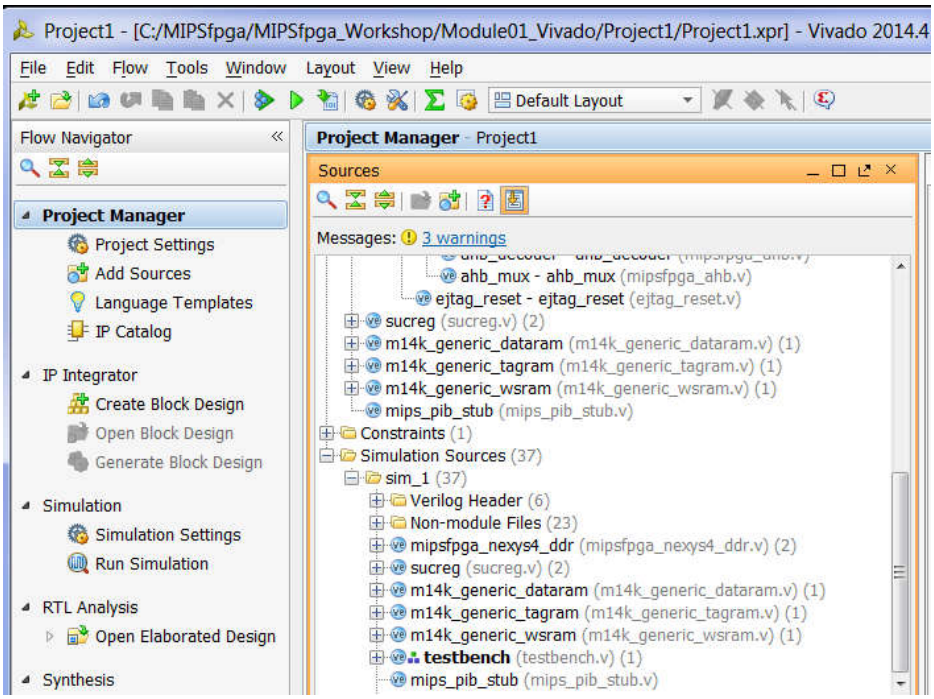

图 9.设置 testbench 为仿真顶层模块

修改 testbench.v 来实例化修改后的 MIPSfpga 系统 （mipsfpga_sys）。

如果在实验 1 的时候你添加过仿真内存初始化文件，现在需要先删除它们。展开 Simulation Sources → sim_1→ Text 并且删除已存在的文件（也就是 ram_reset_init.txt）如图 10 所示。右键点击 ram_reset_init.txt 选择 Remove File from Project 然后点击 OK。

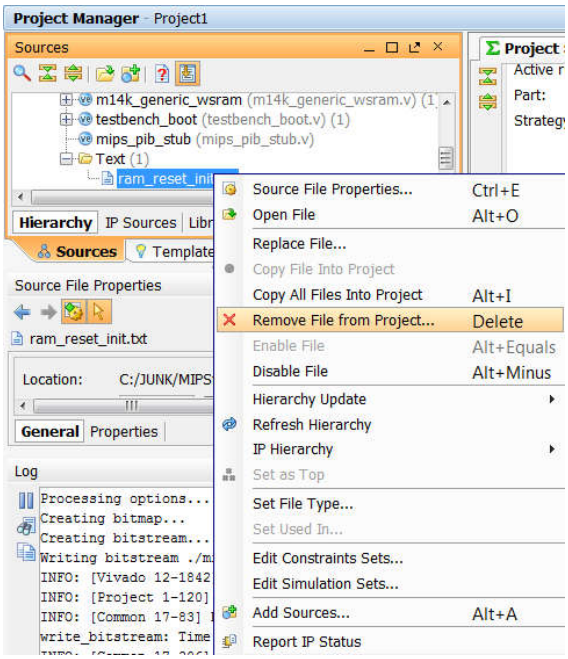

图 10.移除已经存在的内存初始化文件

接下来我们就要把刚生成的 ram_reset_init.txt 添加进项目作为仿真源文件。如果你忘记该怎么做了，去参考一下实验 1。

执行仿真并且检查 MIPSfpga 系统输出的 7-段数码管显示器信号(IO_7SEGEN_N 和 IO_7SEG_N)的正确性。如果它们不正确，调试你的模块。为了调试，你将可能需要查看低层信号。同样，如果你忘记怎么操作，参考实验 1。

仿真时记住显示每一个 7-段数码管显示器数字的时钟要比系统中其他部分的时钟慢很多相对于 20ns 的时钟周期，这里只有大约 2ms）。

在 MIPSfpga 系统上运行示例汇编程序

现在我们已经完成对新增加的支持 7-段数码管显示器的硬件之仿真测试，接下来你将把示例汇编程序运行到 MIPSfpga 系统上。

确信你的 MIPS 汇编程序在硬件上操作是对的，否则，你将获得另一次锻炼你调试技巧的机会。

在 MIPSfpga 系统上运行示例 C 程序

现在，编写一个简单的 C 程序来些 7-段数码管显示器。编译和调试你的程序，然后在硬件上运行和测试你的 C 程序。

3. 写一个使用 7-段数码管显示器的程序

现在你将编写写两个新的 C 语言程序来锻炼使用 7-段数码管显示器的能力：

1. 第一个程序叫做 **SwTo7segHex**，以 **16 进制**的形式将读取到的 16 个拨码开关的值显示到 7-段数码管显示器上。
2. 第二个程序叫做 **SwTo7segDec**，以 **10 进制**的形式将读取到的 16 个拨码开关的值显示到 7-段数码管显示器上。

编写完代码之后就在你修改过的 MIPSfpga 系统上运行、调试并测试你的程序。