MIPSfpga by Imagination

Lab 5

Memory-Mapped I/O: 7-Segment Displays

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

MIPSfpga Lab 5:

Memory-Mapped I/O: 7-Segment Displays

1. Introduction

This lab teaches the principles of memory-mapped inputs and outputs (I/O) by showing how to build hardware modules to expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board. You will then test your new hardware by simulating a short sequence of MIPS assembly code. At the end of the lab, you will write C programs that display the value of the switches on the 7-segment displays.

2. MIPSfpga Memory-Mapped I/O

A processor uses the memory interface to interact with peripheral devices, such as the switches, LEDs, and 7-segment displays on the Nexys4 DDR FPGA board. *Memory-mapped I/O* enables a processor to write to or read from a peripheral device in the same manner that it reads or writes memory. Each peripheral device is assigned one or more memory addresses. When the processor accesses such a memory address, the peripheral device is accessed instead of memory. The MIPSfpga system uses the AHB-Lite bus to access external memory and peripherals.

AHB-Lite Bus

The AHB-Lite bus has a clock, write enable, address, and read and write data signals (HCLK, HWRITE, HADDR, HRDATA, and HWDATA), as shown in Figure 1. The "H" prefix indicates that they are part of the AHB-Lite bus. Memory and peripherals are connected to this interface to receive and supply data. The MIPSfpga core sends these signals to the AHB-Lite Bus:

- **HCLK**: the 50 MHz system clock
- **HWRITE**: write enable (1 when writing, 0 when reading)
- **HADDR**: the address being read or written
- **HWDATA**: the data being written on a write

The MIPSfpga core receives the following input from the AHB-Lite bus:

- **HRDATA**: the read data produced by memory or the peripherals

The MIPSfpga system has three modules on the AHB-Lite bus: two memories (RAM0 and RAM1) and a general-purpose I/O module (GPIO). RAM0 contains the boot code and RAM1 contains the user code and data. The GPIO unit interfaces with the LEDs, switches, and pushbuttons on the Nexys4 DDR board.

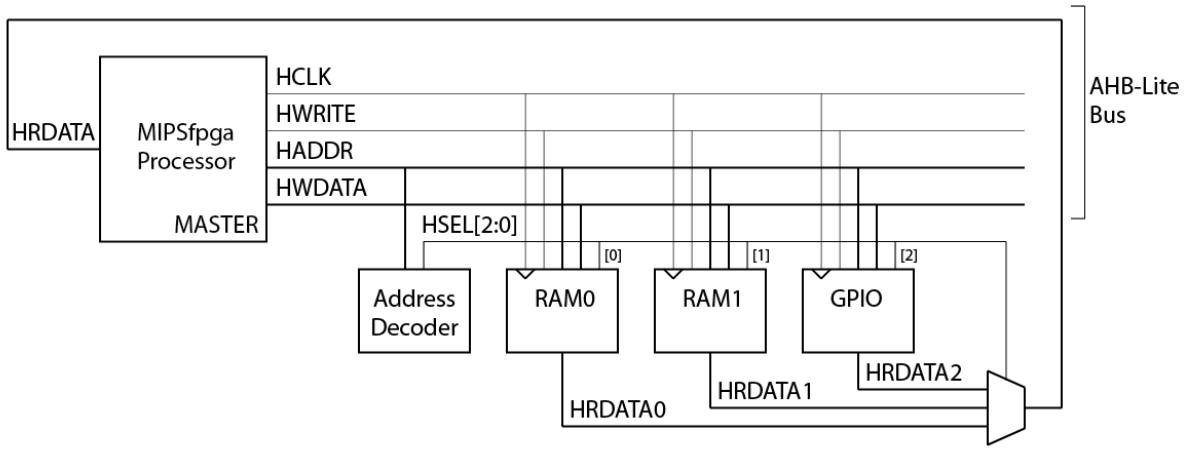

Figure 1. MIPSfpga processor with three peripheral devices

In addition to the three peripherals, the memory-mapped I/O interface requires an Address Decoder and a multiplexer. Depending on the address generated by the processor (HADDR[31:0]), the Address Decoder will enable the processor to access one of the three modules. The Address Decoder generates a select signal HSEL[2:0] that is used by the modules and by the 3:1 multiplexer.

RAM0 is 1 KB and holds the boot code (virtual addresses 0xbfc00000-0xbfc003fc = physical addresses 0x1fc00000-0x1fc003fc). RAM1 is 256 KB and holds the user code (virtual addresses 0x80000000-0x8003fffc = physical addresses 0x00000000-0x0003fffc). The LEDs, switches, and pushbuttons on the Nexys4 DDR board are mapped to virtual memory addresses 0xbf800000-0xbf800008, as shown in Table 1. The processor code uses virtual memory addresses, and the AHB-Lite bus receives physical addresses. The memory management unit (MMU) on the MIPSfpga core performs this address translation.

Table 1. Memory addresses for Nexys4 DDR FPGA board

Virtual address	Physical address	Signal Name	Nexys4 DDR
0xbf80 0000	0x1f80 0000	IO_LED	LEDs
0xbf80 0004	0x1f80 0004	IO_SW	switches
0xbf80 0008	0x1f80 0008	IO_PB	U, D, L, R, C pushbuttons

The following sequence of MIPS assembly instructions writes the value 5 to the LEDs:

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

Recall that load-upper-immediate (`lui`) loads the 16-bit value 0xbf80 into the upper half of $8 and clears the lower half. Upon execution of the store word instruction (`sw`), HADDR = 0x1f800000, HWRITE = 1, and HWDATA = 5. The Address Decoder detects that address

0x1f800000 belongs to the general-purpose I/O (GPIO) peripheral and asserts HSEL[2], the select signal associated with that peripheral. The GPIO module detects that HSEL[2] and HWRITE are asserted. Because the GPIO module could potentially write to multiple peripherals, the module uses the address to determine that the LEDs should be written with the value on the HWDATA bus. Specifically, a register whose output is physically connected to the LEDs is updated with the value on HWDATA. That way, the value persists until the LEDs are written again by a later instruction.

Similarly, the following sequence of code reads the value of the switches:

```
lui $8, 0xbf80      # $8 = base address of the I/O
lw  $9, 4($8)       # $9 = value of the switches
```

Upon execution of the load word instruction (`lw`), HADDR = 0x1f800004 and HWRITE = 0 (indicating a read). The Address Decoder detects that address 0x1f800004 belongs to the GPIO peripheral and asserts HSEL[2] (and keeps the other select signals HSEL[1] and HSEL[0] low). The GPIO module detects the address corresponding to the switches and selects to send the value of the switches to its read data output, HRDATA2. The select signals HSEL[2:0] control the multiplexer. Because HSEL[2] is asserted, the multiplexer sends HRDATA2 through to HRDATA. The MIPSfpga processor then reads the value on HRDATA, as it would with a typical read from memory, and stores that value in $9. Thus, after the `lw` completes, $9 contains the value of the switches.

The hardware for the MIPSfpga AHB-Lite modules is located in the mfp_ahb module and its submodules. It is best to view this module in your Vivado project, so that the hierarchy is clear. Open the Vivado project that you created in Lab 1. In the Project Manager window, expand mfp_nexys4_ddr → mfp_sys → mfp_ahb_withloader → mfp_ahb to view the mfp_ahb hierarchy, as shown in Figure 2. Double-click on any of the modules and the Verilog file will open in the neighboring panel.

The mfp_ahb_withloader interfaces the AHB-Lite bus with the UART as well to allow direct memory access between the UART and the MIPSfpga memories.

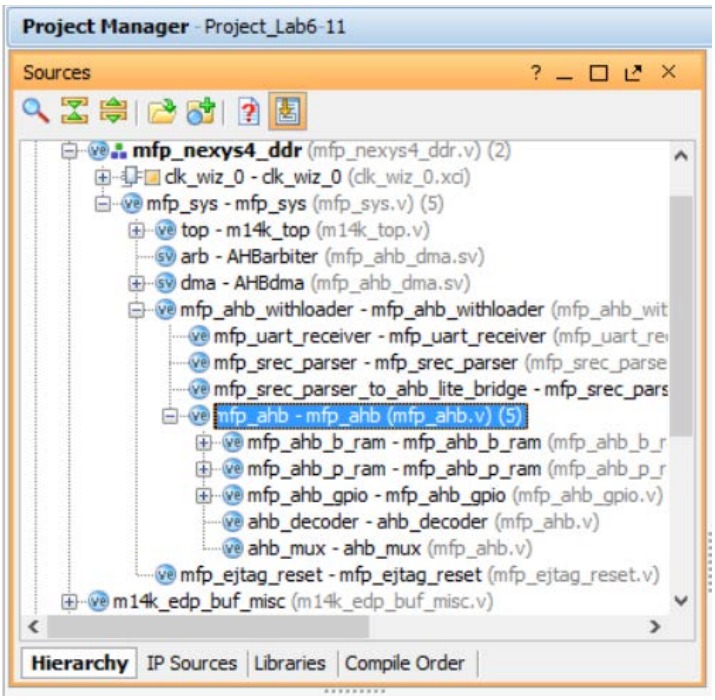

Figure 2. mfp_ahb hierarchy shown in Vivado

Double-click on mfp_ahb to view the FPGA board interface signals (see Figure 3). Notice all of the AHB-Lite signals from Figure 1 (HCLK, HADDR, HWRITE, HWDATA, and HRDATA). Additional AHB-Lite signals are also available and will be discussed in later labs. The module also has the memory-mapped I/O signals IO_Switch, IO_PB, and IO_LED that connect to the switches, pushbuttons, and LEDs on the Nexys4 DDR board.

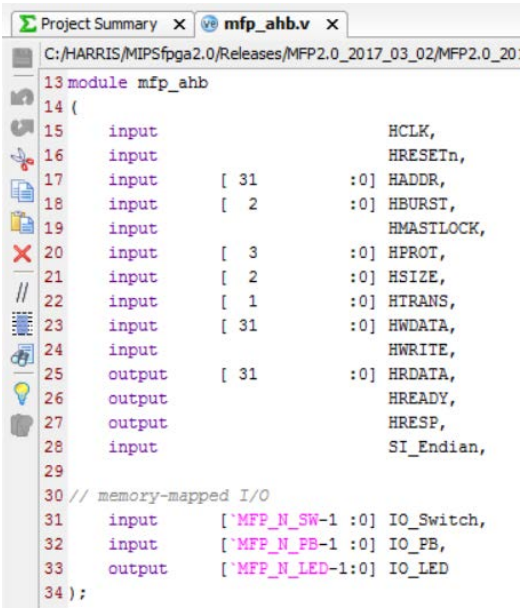

Figure 3. mfp_ahb interface signals

The modules instantiated within mfp_ahb are the three peripherals (boot memory, program memory, and GPIO interface), address decoder, and multiplexer shown in Figure 1. The corresponding Verilog module names are given in Table 2. View the Verilog code to see how the functionality described above is implemented.

Table 2. AHB-Lite Modules

Name from Figure 1	Module Name
RAM0	mfp_ahb_b_ram
RAM1	mfp_ahb_p_ram
GPIO	mfp_ahb_gpio
Address Decoder	ahb_decoder
Multiplexer (for HRDATA)	ahb_mux

The GPIO module (mfp_ahb_gpio) interfaces with the general-purpose I/O on the Nexys4 DDR board. The MIPSfpga system includes access to the LEDs, switches and pushbuttons on the board. In this and the next labs, you will expand the MIPSfpga functionality to extend to other peripherals, starting with the eight 7-segment displays available on the Nexys4 DDR board.

7-Segment Displays

Digits can be represented using 7-segment displays, as shown in Figure 4. Each of the seven segments is labeled a through g. The numbers 0 through F light up the segments shown in Figure 5. For example, the number 0 lights up all but the middle segment, g.

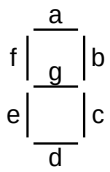

Figure 4. Seven-segment display

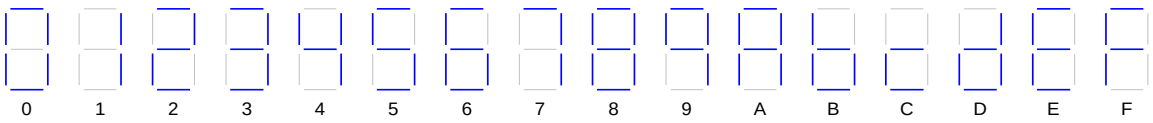

Figure 5. Seven-segment display function

Given an input number ranging from 0x0 – 0xF, we will show how to expand the MIPSfpga system to drive the 7-segment displays to show that number. Each segment of the display is low-asserted, so it turns ON when it is 0.

The truth table below (Table 3) shows the inputs (a 4-bit value from 0-15) and outputs for a *7-segment display decoder* that takes in a 4-bit number and produces the value of the segments corresponding to that number. So, for example, with an input of "0", the 7-segment display decoder turns all but the middle segment (S_g) ON. Thus, the first row for Hexadecimal digit "0" shows all the segments as 0 except S_g. (Remember that the segments are low-asserted, so they are ON when they receive 0.) The digit "1" should only have S_b and S_c ON (so S_b and S_c are 0 in that row), and so forth.

Table 3. Truth table for 7-segment display decoder

Hexadecimal Digit	Inputs				Outputs							
	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

Build 7-Segment Decoder

Write a Verilog module that describes the seven-segment display decoder in hardware. The module declaration is provided for you in:

Lab05_7seg\VerilogFiles\rtl_up\system\mfp_ahb_sevensegdec.v

The module has a 4-bit input, data[3:0], and a 7-bit output, seg[6:0], corresponding to each of the segments a-g. Test your hardware in simulation using XSIM and debug as needed. In the next step, you will use this module to drive the 7-segment displays on the Nexys4 DDR board.

7-Segment Displays on the Nexys4 DDR board

The Nexys4 DDR board has eight 7-segment digits. All eight of the digits on the Nexys4 DDR board are connected to the same low-asserted segment pins, referred to as CA, CB, CC,…,CG. However, each digit has its own enable which is also low-asserted. Figure 6 shows the eight 7-segment displays on the Nexys4 DDR board. CA is connected to the cathode of the A segment for all eight displays, CB to the cathode of the B segment for all displays, and so forth. Each digit has an enable signal, corresponding to the respective bit of the signal AN[7:0]. AN[7:0] is connected via an inverter, to the anode of all segments for the respective digit. For example, if AN[7] is 0, digit 7 will be driven to the values on CA…CG.

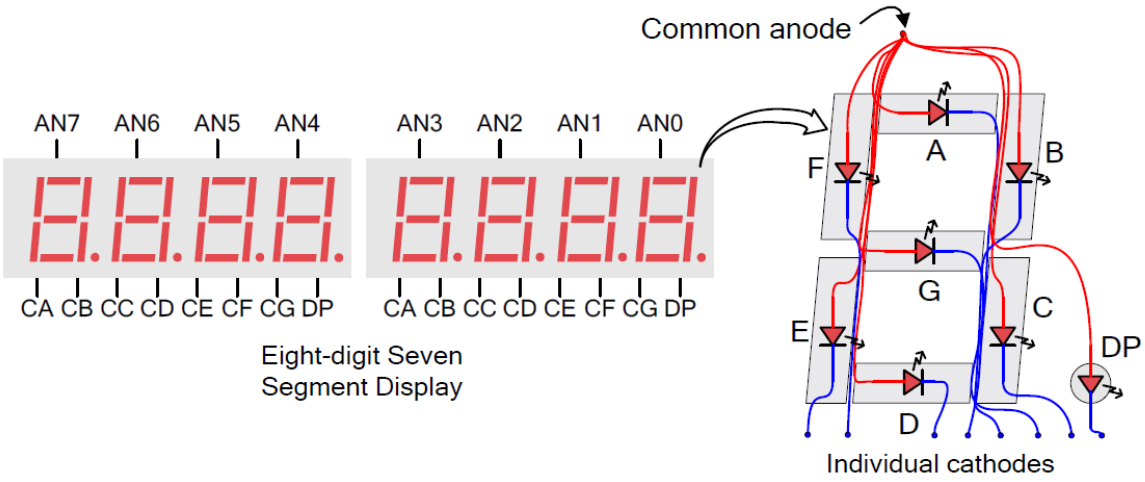

Figure 6. Eight 7-segment displays on the Nexys4 DDR board

(© Nexys4 DDR Reference Manual)

To drive each segment to a different value, the enables (AN[7:0]) and segment values (CA…CG) must be driven sequentially, at a rapid enough speed that our eyes don't detect the flicker. For example, to drive display 0 and 1 to the values 3 and 9, we drive CA…CG to display the value 3, and then drive AN[0] LOW, then we drive CA…CG to display the value 9 and drive AN[1] LOW. If we refresh each digit about every 2 ms, our eyes can't detect any flicker.

Build HDL Module to Drive Nexys4 DDR 7-Segment Displays

Now you will write a Verilog module that drives the eight digits of 7-segment displays on the Nexys4 DDR board. The module declaration is provided in:

Lab05_7seg\VerilogFiles\rtl_up\system\mfp_ahb_sevensegtimer.v

The module receives the number to display on each of the eight digits (DISP0[3:0] – DISP7[3:0]) and a signal indicating which of the eight displays are enabled (EN[7:0]). It also receives the 50 MHz clock (clk) and a low-asserted reset signal (resetn) as inputs. The outputs are the 7-segment display enables (DISPENOUT[7:0]) and the values of the 7 segments, A-G (DISPOUT[6:0]). Later in the lab you will connect these outputs through to the top-level module (mfp_nexys4_ddr) so that they drive the eight display enables (AN[7:0]) and the seven segment pins (CA…CG).

Your module should drive each enabled digit sequentially about every 2 ms. You will need to use your 7-segment display decoder that you wrote in the previous section. Note that you could expand the functionality of this module to include the decimal point (DP) if desired. Test your hardware in simulation using XSIM and debug as needed.

Adding Seven-Segment Display Functionality to the GPIO AHB-Lite Module

Now that you have written the hardware modules that will write the eight 7-segment displays, add functionality to the MIPSfpga system to interface with the displays. Your goal is to enable the user to write to the eight 7-segment displays using SW. Start by doing the following:

1. Assign memory-mapped I/O addresses to the enable signal and each of the eight digits
2. Modify the GPIO module to detect these addresses and store the written data to the associated memory-mapped I/O registers
3. Connect these registers to the mfp_ahb_sevensegtimer module you just created

To make these changes, you will need to modify the following files (found in the MIPSfpga_Labs\rtl_up directory):

- mfp_ahb_const.vh
- mfp_ahb_gpio.v

Note that you may want to save a **backup** of the rtl_up directory or use version control before continuing. Below is some guidance for each of the above steps.

1. Assign memory-mapped I/O addresses

Assign two addresses to the seven-segment displays: one for the enable and one for the value of the digits, as shown in Table 4. The value to write to the digits is mapped to a 4-bit portion of the DIGITS_N[31:0] signal. The value of digit 0 is in DIGITS_N[3:0], digit 1 is in DIGITS_N[7:4], and so on. The user writes to these addresses to set the enables and the digit values.

Table 4. Memory addresses for Nexys4 DDR FPGA 7-segment displays

Virtual address	Physical address	Signal Name	Nexys4 DDR
0xbf80 000c	0x1f80 000c	SEGEN_N[7:0]	AN[7:0]
0xbf80 0010	0x1f80 0010	DIGITS_N[31:0]	Value of digits 7:0

To define these memory-mapped addresses, modify the mfp_ahb_const.vh Verilog header file. In Vivado, open Project1. Browse to **mfp_ahb_const.vh** in the Project Manager window (as shown in Figure 7), under Design Sources → Verilog Header.

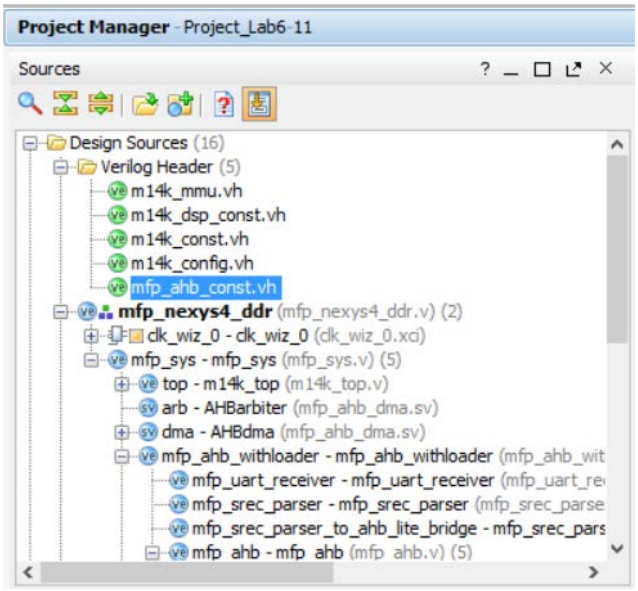

Figure 7. mfp_ahb_const.vh Verilog header file

Define the new memory-mapped I/O addresses for the 7-segment displays as H_7SEGEN_ADDR and H_7SEGDIGITS_ADDR. The Address Decoder (ahb_decoder module) uses the most significant bits of the address to detect which of the three AHB slaves to enable (the reset RAM, program RAM, or GPIO module). Then, once selected, the GPIO module uses the lower bits of the address to determine which of its peripherals should be written or read. Bits 5:2 of the memory-mapped I/O address are saved in another constant: H_*_IONUM, lower in the mfp_ahb_const.vh file, as shown below:

```
`define H_LED_IONUM                (4'h0)
`define H_SW_IONUM                 (4'h1)
`define H_PB_IONUM                 (4'h2)
```

For example, the switches are mapped to physical address 0x1f800004, so bits 5:2 are 0x1 (i.e., H_SW_IONUM is 4'h1).

Name the I/O numbers for the 7-segment display variables: H_7SEGEN_IONUM and H_7SEGDIGITS_IONUM. For example, because the address for the 7-segment digit enables is 0x1f80000c, H_7SEGEN_IONUM is 0x3.

2. Modify the GPIO module

Now modify the GPIO module to detect the nine memory-mapped I/O addresses you just defined and write the data (HWDATA) to those registers when the corresponding address is detected. In Project1 in Vivado, open **mfp_ahb_gpio.v.** In the module declaration, declare and output the enable and segment signals (A-G). Name these signals IO_7SEGEN_N[7:0] and IO_7SEG_N[6:0], respectively. In a higher-level module, these will drive the enables (AN[7:0]) and segment values (CG…CA) on the Nexys4 DDR board.

You must now create two registered signals, one that holds the values of the enables and the other that holds the value of the eight 7-segment display digits. The user will write to these registers using memory-mapped I/O. Name the registered signal that holds the enables SEGEN_N[7:0]. Name the register that holds the eight 4-bit values to display on the eight digits SEGDIGITS_N[31:0]. Modify the GPIO module so that these registers get written when the correct address is detected.

3. Connect these registers to the mfp_ahb_sevensegtimer module you just created

Now, within the GPIO module, instantiate and connect the `mfp_ahb_sevensegtimer` module that you built earlier in this lab. You will connect its inputs to the memory-mapped I/O registers (as well as the clock and reset signals) and its outputs to the 7-segment display signals (IO_7SEGEN_N and IO_7SEG_N).

Connect the 7-Segment Display Signals to the Nexys4 DDR Board

Now you will connect the output signals from the AHB GPIO module through to drive the 7-segment displays on the Nexys4 DDR board. Feed the output signals from the GPIO module (IO_7SEGEN_N and IO_7SEG_N) up through the levels of hierarchy until they reach the Nexys4 DDR board (i.e., outputs of the mfp_nexys4_ddr module). To do so, you will need to modify the following modules:

- mfp_ahb.v
- mfp_ahb_withloader.v
- mfp_sys.v
- mfp_nexys4_ddr.v
- testbench.v

In the highest-level module (mfp_nexys4_ddr.v), name the output signals that will drive the 7-segment display: CA, CB, CC, CD, CE, CF, CG, and AN[7:0]. Recall that CA…CG drive the segments and AN[7:0] drives the enables.

You will also modify the Xilinx Design Constraint (.xdc) file. Open this file from Project1 (in Vivado) by expanding Constraints → constrs_1 in the Project Manager window as shown in Figure 8. Double-click on the mfp_nexys4_ddr.xdc file to open it. The XDC file assigns the signal

names, AN[7:0] and CA – CG, to pins on the Artix-7 FPGA that are physically connected to the 7-segment display inputs on the Nexys4 DDR board using wire traces.

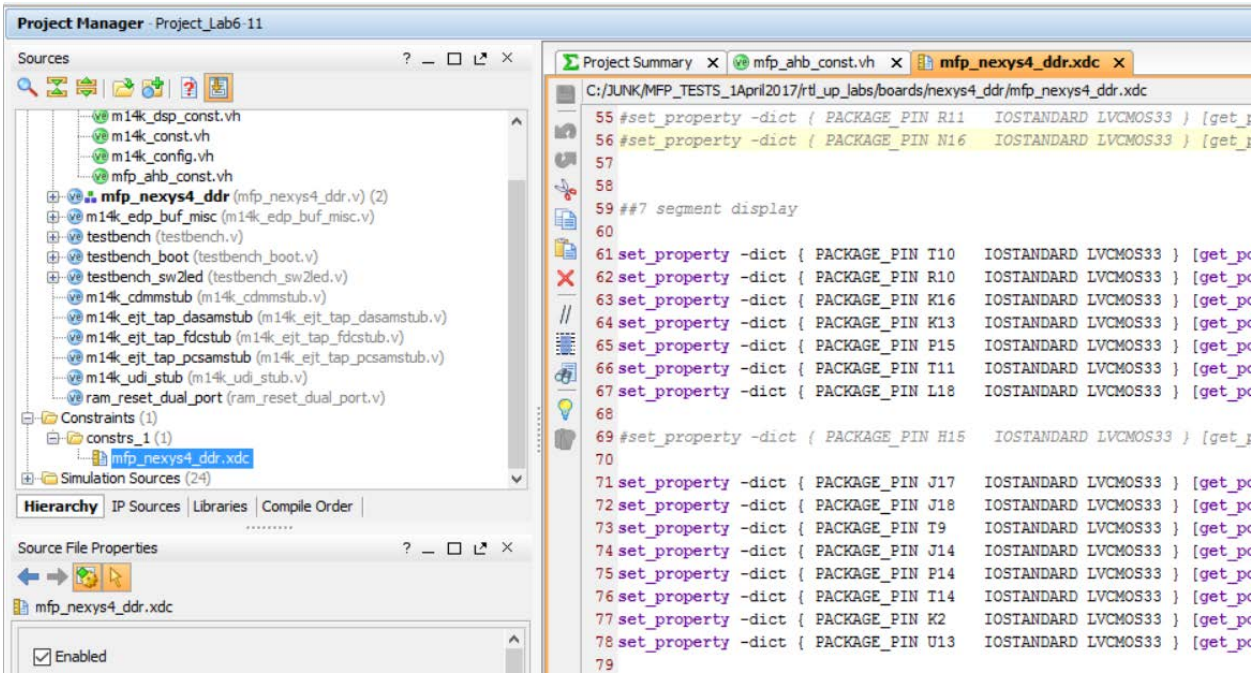

Figure 8. Opening Xilinx Design Constraint file

With the .xdc file open in Vivado, search for (ctrl-F) "segment" to find the listing of 7-segment display outputs, as shown in Figure 8. The information about which pins are connected to the 7-segment displays is already available in the file – it needs only be uncommented. Do so by deleting the # before each line you'd like to use. For example, signal CA that drives the A segment of the 7-segment displays connects to pin T10 on the FPGA by the following line:

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

This pin is connected via a wire trace to the segment A input of the 7-segment displays on the Nexys4 DDR board. CB should output to the R10 Artix-7 pin, and so on. The signals driving the anodes of the 7-segment displays (AN[7:0]) are also assigned Artix-7 package pins. Leave the DP signal is commented out (#) because it is not used.

Near the bottom of the constraints file, add the following output timing constraints for the 7-segment display signals:

```
set_output_delay -clock "clk_virt" -min -add_delay 0.000 [get_ports
{AN[*]}]
set_output_delay -clock "clk_virt" -max -add_delay 20.000 [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 20.000 [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 20.000 [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 20.000 [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 20.000 [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 20.000 [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 20.000 [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 20.000 [get_ports {CG}]
```

Testing Seven-Segment Display Functionality

Now test the basic functionality of the 7-segment display hardware by writing a simple assembly program that writes to the 7-segment displays. You will then test your hardware by simulating this simple program **without the bootcode** using XSIM.

Your simple assembly program should enable the 7-segment displays and then write values to each of the eight digits. Create your program in the following directory:

`MIPSfpga_Labs\Labs\Xilinx\Part2_IO\Lab05_7seg\AssemblyExample`

Copy the entire contents of the ReadSwitches directory from Lab 3 to the AssemblyExample directory as a starting point for your new MIPS assembly program. Modify the main.S file. Then compile the program using make and debug as needed. **Note:** do not use any jump (`j` or `jal`) instructions in your code.

Extract Machine Code for Simulation

You can either simulate the entire program (with bootcode) using testbench_boot or extract just the simple assembly program you wrote and place it at 0xbfc00000 (or 0x9fc00000) to save simulation time. We recommend the latter option and show how to do so.

First, convert the MIPS assembly program you wrote to machine code in order to simulate just this program (not the boot code) on the MIPSfpga core. You can use any method you prefer to convert assembly to 32-bit machine code: by hand (tedious, so not recommended – but doable!), using another simulator (such as QtSpim), extracting the machine code from the executable generated by Codescape, etc. We show you how to use Codescape to convert MIPS assembly to machine code, using these steps:

Step 1. Compile the assembly program
Step 2. Isolate the assembly program code in the generated machine code file
Step 3. Run the CreateMemFiles.exe program

Step 1. Compile the assembly program

Use the Makefile to compile the assembly program, as you have done before. This will generate the **program.txt** file that shows the assembly code next to the machine code. You will use this file to extract the machine code for your assembly program.

Step 2. Isolate the assembly program code in the generated machine code file

Now open the **program.txt** file in the AssemblyExample directory and search for 80000204, this is the beginning of the main function in the assembly program. Your program may look similar to Figure 9.

```
80000204 <main>:
80000204:   3c08bf80    lui   t0,0xbf80
80000208:   24090007    li    t1,7
...
```

Figure 9. Assembly program in program.txt

Delete everything in program.txt except your assembly program and then place the first instruction of your assembly program at 0x9fc00000 and the remaining instructions consecutively after that. Recall that 0xbfc00000 and 0x9fc00000 map to the same physical address (0x1fc00000), but the compiler generates code at 0x9fc00000 so that it can be cacheable. Thus, address 0x9fc00000 is the address the CreateMemfiles.exe script seeks. The program in Figure 9 will now look as shown in Figure 10.

```
9fc00000 <main>:
9fc00000:   3c08bf80    lui   t0,0xbf80
9fc00004:   24090007    li    t1,7
...
```

Figure 10. Modified program.txt file

Now copy the CreateMemfiles.exe file from the MIPSfpga_GSG\Scripts directory to the AssemblyExample directory you just created. Open a command shell and change to your AssemblyExample directory and run the CreateMemfiles.exe program at the command shell prompt by typing:

```
CreateMemfiles.exe
```

This program generates the byte-wide versions of each of the instructions. ram_b0.txt – ram_b3.txt list the instructions' least to most significant byte. You'll notice that ram_p0.txt – ram_p3.txt are empty because no code is placed in program memory space (virtual address 0x80000000+).

You will point the simulation to initialize the reset/boot RAM (RAM0) with the ram_b0.txt – ram_b3.txt files you just created. Note again that you can only relocate code in this manner if there are no jump instructions.

Simulation

Now run the assembly code in simulation on the MIPSfgpa system to test your new 7-segment display hardware. Either create a new project with your new files or add your new and modified Verilog files to the project you created in Lab1.

Now, you are ready to simulate your new hardware. In the Project Manager window, scroll down to and expand Simulation Sources and sim_1. Make sure **testbench** is bold, indicating that it is the top-level simulation module, as shown in Figure 11.

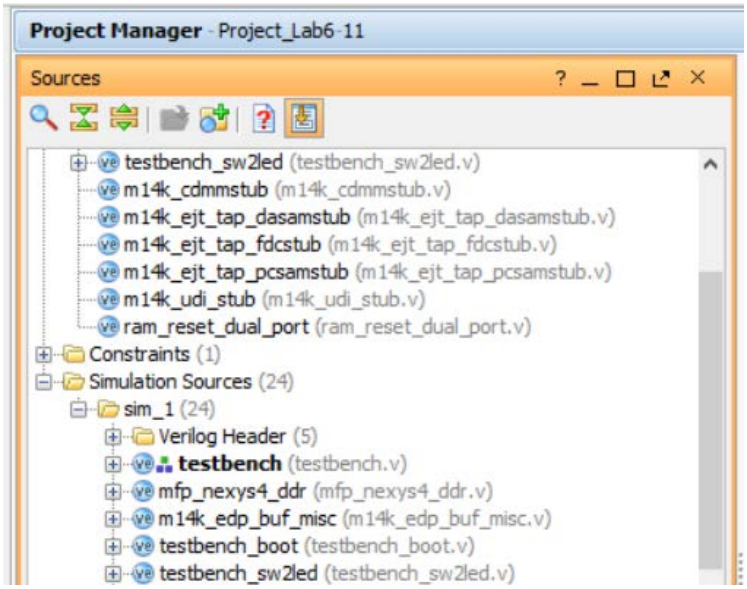

Figure 11. testbench as top-level module for simulation

Modify testbench.v to instantiate your modified MIPSfpga system (mfp_sys).

If you added a memory initialization file for simulation in Lab 1, remove it by expanding Simulation Sources → sim_1 → Text and deleting any existing files (i.e., ram_*.txt) as shown in Figure 12. Shift-click, then right-click on the ram_*.txt files and select **Remove File from Project** and click OK.

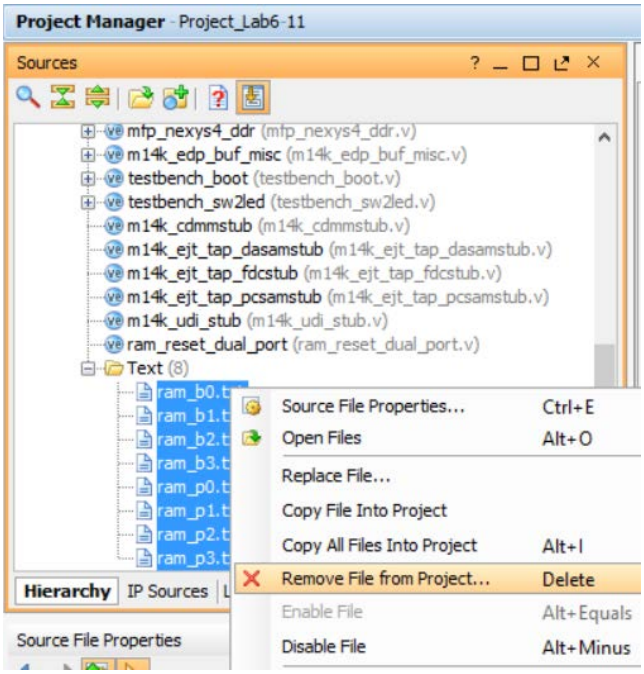

Figure 12. Remove existing memory definition files

Now add the ram_b?.txt files that you just created as simulation sources by choosing Project Manager → Add Sources → Add or create simulation sources → Add Files. Then browse to where you created the ram_*.txt files from the AssemblyExample program. (Be sure to choose Files of type: All Files). Select all of the files and click OK, as shown in Figure 13.

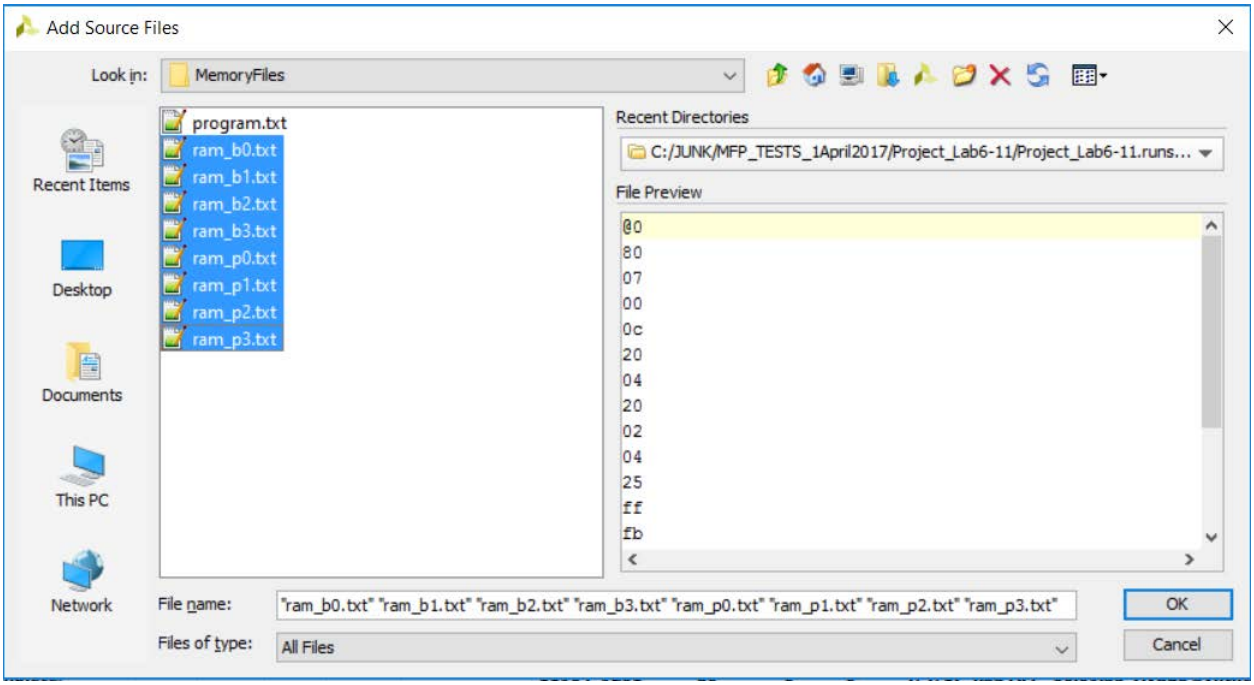

Figure 13. Adding source files

Run the simulation and check that the MIPSfpga system outputs the correct values for the 7-segment display signals (IO_7SEGEN_N and IO_7SEG_N). If it doesn't, debug your modules. To debug, you'll likely want to view signals from lower levels in the hierarchy. Again, refer to Lab 1 instructions if you've forgotten how to do this.

As you're simulating, remember that the timing of displaying each of the 7-segment display digits is much slower than the rest of the system (~2 ms versus 20 ns cycle time). You may want to temporarily decrease this time to test your hardware in simulation. However, if you do so remember to change it back later!

Running the Example Assembly Program on the MIPSfpga System

Now that you have simulated and tested your added hardware to support writing values to the 7-segment displays, you are ready to run your entire compiled MIPS assembly program (with the boot code) on the MIPSfpga system in hardware. Use the loadMIPSfpga.bat script, as you have done before.

Example C Program on the MIPSfpga System

Now write a simple C program that writes to the 7-segment displays. Compile and debug your program and then run and test your C program in hardware.

3. Write a Program Using 7-segment Displays

After you have tested your hardware above, write two new C programs to exercise the 7-segment display capabilities:

1. The first program, called **SwTo7segHex**, should write the **hexadecimal** value of the 16 switches to the 7-segment displays.
2. The second program, called **SwTo7segDec**, should write the **decimal** value of the 16 switches to the 7-segment displays.

When you are done, run, debug, and test your programs on your modified MIPSfpga system.

