Learning Objectives:

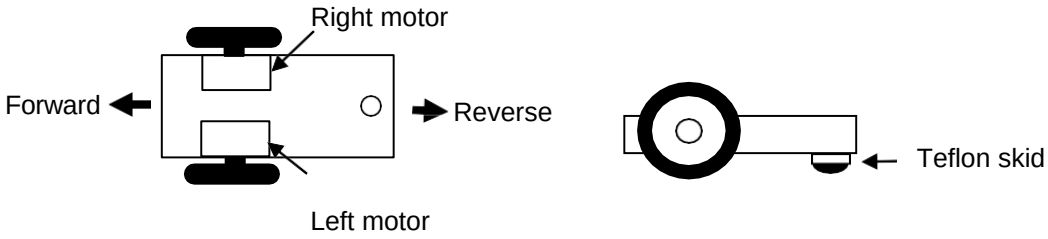
Project 1

- Practice coding Verilog for synthesis
- Build on the MIPSfpga Getting Started projects
- Practice assembly language coding MIPSfpga Processor
- Introduce the Digilent Nexys A7 development board
- Gain experience debugging an SoC embedded system
- (optional) Learn how to use GitHub and GitHub classroom

Project 1 deliverables are due in <u>both</u> GitHub (make a final push of your deliverables) and in D2L.

Project: SimpleBot

This project models a very simple robot using the Nexys A7 board. The "virtual" robot is a platform with two wheels, each driven by an independent motor. A third free rotating wheel or Teflon skid serves to stabilize the platform.



For this project, you will not use a physical robot; instead, we will model the robot's operation using the Digilent Nexys A7 development board. In this first project we will use the pushbuttons, switches and seven-segment display to control your virtual robot's wheel motors and show information about the robot's motion.

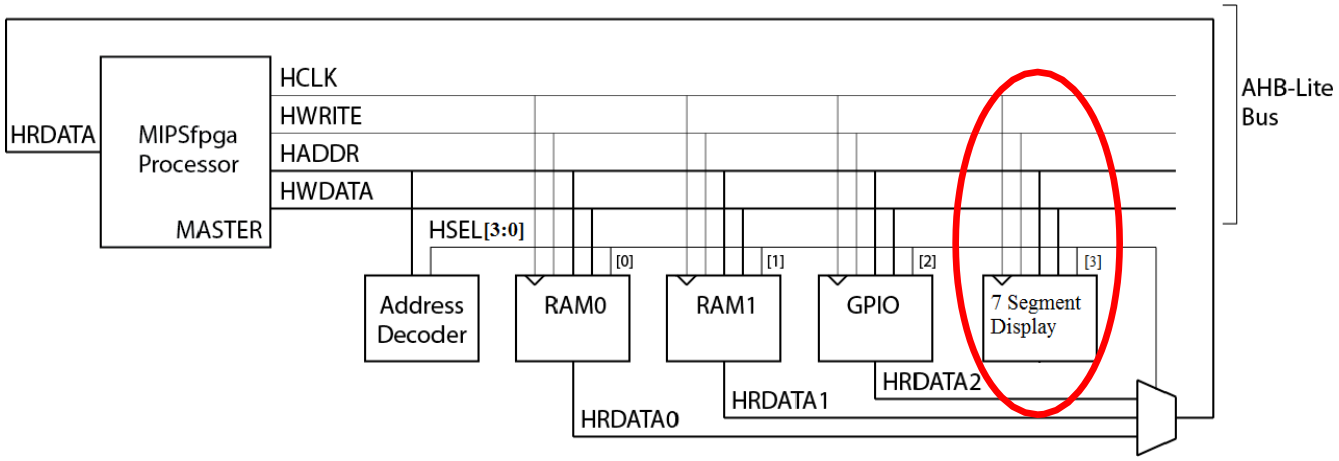
You will add the Verilog code to implement a seven-segment display peripheral by connecting it to the AHB-Lite bus so that the display digits can be written by a program running on the MIPSfpga Core. We have included Verilog modules to decode and drive the seven-segment display digits and timing logic to provide a delay of about 20ms (to avoid flicker) to get you started. You will need to understand the organization of the MIPSfpga core and system modules to know which files to add and which of the existing files in `rtl_up` you will have to modify (if you don't know what the `rtl_up` folder is you need to review the *Getting Started with Hardware* guide). You have also been given a working reference assembly language program to test your hardware and to use as the basis of your SimpleBot application. Use this assembly language program to verify the proper operation of your synthesized design and your

Nexys A7 board. The reference program we provide reads the values on the slide switches and shows them on the LED's and 7-Segment display.

Your task for this project is divided into the Hardware and Software development:

Hardware Development:

Design an AHB-lite peripheral for the MIPSfpga to drive the seven-segment display on the Nexys A7 board.



The MIPSfgpa system in the Getting Started project implemented three address ranges (RAM0, RAM1 and GPIO); adding a memory-mapped I/O interface for the seven-segment display requires an additional address range. Since the seven-segment display peripheral is "write only" (digit codes from the CPU to the peripheral) the interface is straightforward. Your new peripheral takes HCLK (AHB-Lite bus clock), HWRITE (AHB-Lite write enable), HADDR (AHB-Lite address bus), and HWDATA (AHB-Lite write data bus) as inputs. If the seven-segment display peripheral was a read/write peripheral you would to add an additional input to the HRDATA multiplexer (read data from peripheral to CPU) and expand the multiplexer, but it is not. Note that all of the AHB-Lite buses are unidirectional – this is to be expected in a system bus for a component like an FPGA which does not support tristate (Hi-Z) signaling internally. The Address Decoder module in `rtl_up` will need to be updated to generate a HSEL signal which is asserted when an address in the address range recognized by your seven-segment display peripheral is on the HADDR bus.

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, pushbuttons, and your seven-segment display registers are mapped to virtual memory addresses 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. You can reassign the I/O range for your peripheral if you'd like.

Table 1. Memory addresses for Nexys A7 FPGA board

Virtual address	Physical address	Signal Name	Nexys A7
0xBF70_0000 to 0xBF70_000C	0x1F70_0000 to 0x1F70_000C	IO_7SEGEN_N (digit enables), IO_SEG_N (segment values)	Seven-segment display
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

As you can see from this table, the first peripheral in the memory map is the 7-segment display peripheral that you will create. You can use the virtual addresses we suggested or pick a range of your own. Refer to `mfp_ahb.v` and `mfp_ahb_gpio.v` to learn how to design and instantiate your new peripheral.

Software Development:

Design and implement two motion indicators for the SimpleBot: The first indicator (Digit4) indicates whether the SimpleBot is stopped, moving forward, moving in reverse or doing a right turn or left turn. The second is a directional indicator, (Digit3-Digit0), which indicates the compass heading of the SimpleBot (0º = North, 90º = East. 180º = South, and 270º = West and every point in between). The compass range is 0 º to 359 º (e.g. there is no compass heading 360 º, the compass wraps around).

Seven-segment Display:

The seven-segment display on the Nexys4 is a multiplexed display with 8 digits (See Section 9.1 of the Nexys A7 Reference Manual for information on how a multiplexed display works). An application can write to the digits of display by writing to 32-bit locations in the MIPS address space. This is an example of memory mapped I/O. Table 1 places the start of 4 registers that provide the interface between the seven-segment display controller and the CPU at 0xBF70_0000 but you can place it elsewhere in MIPS address space anywhere you'd like; be you need to change the appropriate constants if you do. The 4 registers are as follows:

- Digit enables (Physical address 0x1F70_0000): This is an 8-bit register containing enable bits for each digit. The enable signal is active low, so you need to set the Digit Enable bits to 0 for all of the digits you want to enable. For example, writing 0x0000_00E0 to the Digit Enable register enables Digits 4...0.

- Digit values for Digits 3...0 (Physical address 0x1F70_0008): Each digit can be written with a 5-bit code (see Table 2). The 32-bits in the register are mapped as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
x	x	x	Digit 3					x	x	x	Digit 2					x	x	x	Digit 1					x	x	x	Digit 0				

- Digit values for Digits 7...4 (Physical address 0x1F70_0004): Each digit can be written with a 5-bit code (see Table 2). The 32 bits in the register are mapped as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
x	x	x	Digit 7					x	x	x	Digit 6					x	x	x	Digit 5					x	x	x	Digit 4				

- Decimal points (Physical address 0x1F70_000C): This is an 8-bit register containing the values for the decimal points for each digit. Like the enable signal, setting the Decimal Point for a digit is active low (0) for all of the decimal points you want to light. For example, writing 0x000_000FE to this register will turn on only the LSB decimal point.

Table 2 provides the list of character codes that are recognized by the seven-segment decode module we provide.

Table 2: Display Character Codes

Code (binary)	Displays (ASCII code)
0 - 9	Characters 0 to 9
10 – 15	Upper case characters A to F
16 – 22	Single Segments a to g
23*	Upper case character H
24*	Upper case character L
25*	Upper case character R
26*	Lower case character L (l)
27*	Lower case character R (r)
28 – 31	Space (blank)

* Special characters used in the next project

Each display has 4 digits. The displays need to be refreshed, however, the driver `mfp_ahb_sevensegtimer.v`, does this for you.

SimpleBot Functional Specification

The two independently controlled wheels on the SimpleBot enable the SimpleBot to move forward, backward, turn left or turn right, as shown in Table 3.

Table 3: Robot Motions

Left Motor	Right Motor	Robot motion mode
Stop	Stop	Stop
Forward	Stop	Turn Right 1X Speed
Stop	Reverse	Turn Right 1X Speed
Forward	Reverse	Turn Right 2X Speed
Stop	Forward	Turn Left 1X Speed
Reverse	Stop	Turn Left 1X Speed
Reverse	Forward	Turn Left 2X Speed
Forward	Forward	Forward
Reverse	Reverse	Reverse

Note that the robot can turn at two speeds in either direction depending on whether both or only one wheel is moving during the turn.

PUSHBUTTON MOTOR CONTROL

The four pushbuttons on the Nexys A7 control the two-wheel motors as specified in Table 4:

Table 4: Pushbutton Motor Control

Pushbutton	Motor Function
BTN_LEFT	Left Motor Forward
BTN_UP	Left Motor Reverse
BTN_RIGHT	Right Motor Forward
BTN_DOWN	Right Motor Reverse

A motor is stopped if neither of the two buttons that control the motor are pressed. If *both* buttons are pressed that motors are stopped. Both motors are stopped if all four buttons are pushed at the same time.

You will implement the functionality of the two motion indicators in a MIPS Assembly Language program that you write and debug. You will use four of the eight digits in the 7-segment display for these indicators.

The digits of the seven-segment display are to be configured as follows:

Leftmost Digit							Rightmost Digit
Digit 7	Digit 6	Digit 5	Digit 4	Digit 3	Digit 2	Digit 1	Digit 0
BLANK	BLANK	BLANK	BLANK	Motion Indicator	Compass		

The decimal point to the right of Digit 3 should be lit to separate the Motion Indicator from the Compass. All other decimal points are unassigned. You may, for example, want to use one of the decimal points to blink to show that your program is running (this is up to you).

These functionality of the two indicators is described below:

MOTION INDICATOR

This indictor uses a single seven-segment digit to provide an animated display of the robot's current motion mode, as defined in Table 5.

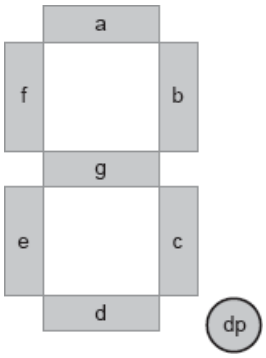


Figure 1: Seven-Segment Digit

Table 5: Motion Indicator

Robot motion mode	**Display Action**
Stop	Segment **g** on **steady**
Turn Right 1X Speed	Chase clockwise 5Hz
Turn Right 2X Speed	Chase clockwise 10Hz
Turn Left 1X Speed	Chase counterclockwise 5Hz
Turn Left 2X Speed	Chase counterclockwise 10Hz
Forward	Segment **a** blinks at about **1Hz**
Reverse	Segment **d** blinks at about **1Hz**

"Chase" means that one lit segment moves around the perimeter of the display digit. Only one segment is lit at a time. The lit segment advances one position at the rate of 5Hz or 10Hz depending on the turning speed. For a clockwise chase, the segments are lit in the following sequence: a, b, c, d, e, f, a, … For a counterclockwise chase, the sequence is reversed.

COMPASS

This indictor uses the three rightmost digits on the seven-segment display indicate the robot's current heading from 0 to 359_{10} degrees. The compass heading is computed by dead reckoning, that is, by accumulating the robot's motions over time. When the robot is stopped or moving in a straight line (either forward or reverse), the compass heading does not change. The compass heading only changes when the robot is turning. The rate of compass change will, of course, depend on the turning rate. The compass can be implemented with an up/down counter. Table 5 defines the actions of the compass counter as a function of the motion mode.

Table 5: Compass Specification

Robot motion mode	Compass Action
Stop	Hold
Forward	Hold
Reverse	Hold
Turn Right 1X Speed	Increment at 5Hz
Turn Right 2X Speed	Increment at 10Hz
Turn Left 1X Speed	Decrement at 5Hz
Turn Left 2X Speed	Decrement at 10Hz

SIMPLEBOT PROJECT DESIGN NOTES

Both the Motion Indicator and Compass modules use the SimpleBot's motion mode as inputs. The motion mode can be decoded from the pushbutton inputs by reading the value of the buttons at physical address 0x1F80_0008. The motion indicator (MI) can be implemented with a finite state machine (FSM). The MI output can use special codes that cause each of the individual segments of a digit to be displayed. Refer to Table 2 for the codes needed to drive the individual segments.

The compass can be implemented with a 0-359 BCD (decimal, not hexadecimal) up/down counter. There is, of course, additional control logic not described in detail here; the implementation is left for you to complete. Figure 2 provides the architecture and a simplified flowchart for your application.

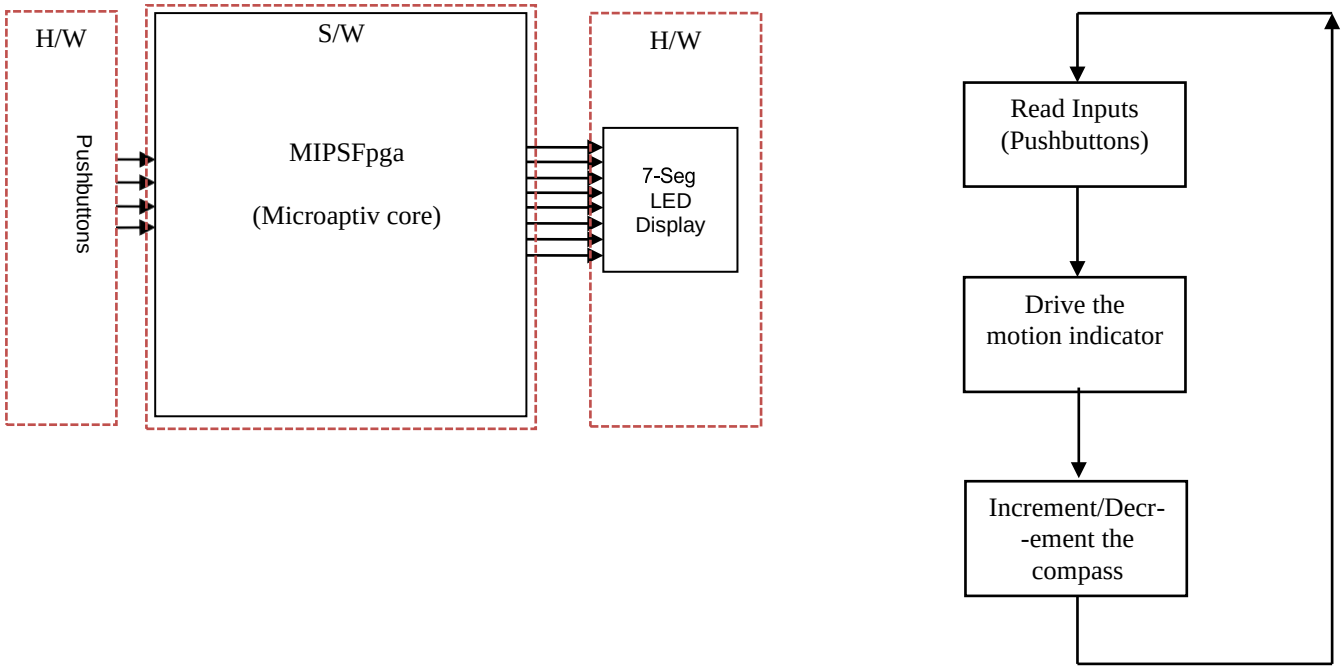


Figure 2: SimpleBot with Compass and Motion Indicator Block Diagram and software flowchart

Project Tasks

Complete the Getting Started Project

Complete the tutorials in the Getting Started Project if you haven't done so already. The hardware and software guides are the most important since they give you an introduction and get you started with the tools to be used for design and debugging in the course and for verifying your setup. This is particularly important if you have your own hardware and software installation. Spend time familiarizing yourself with the Verilog code that interfaces the AHB-Lite bus on the MIPSfpga core with its memory and peripheral devices.

Create a copy of your Vivado project and add the hardware release files to the project

Create a copy of your Getting Started hardware project in Vivado and add the `mfp_ahb_sevensegdec.v` and `mfp_ahb_sevensegtimer.v` files into the project. Replace the existing constraint file with the new file in the `project1_release\constraints` folder. The new constraints file "uncomments" the pin assignments for the 7-segment display on the Nexys A7. They were commented in Getting Started since they were not being used.

Add debounce.v and edit the file mfp_nexys4_ddr.v

Add the `debounce.v` file into the project to debounce the switches and the pushbuttons. Connect the debounced buttons and switches to the GPIO's. Carefully pass all the pushbuttons and switches into the module and drive the inputs to `mfp_sys`. These edits have to be done in `mfp_nexys4_ddr.v`

Design and synthesize the system including your 7-seg display peripheral

Decide on the address for the 7-segment peripheral and enter the required address as constants in the header file `mfp_ahb_const.vh`. Add a select signal for your peripheral and pass that select signal and the AHB-Lite signals into your newly created peripheral. Instantiate the seven-segment timer modules that were provided into your peripheral and drive the signals that drive the display out and up through the hierarchy. If the signals are not driven out onto the `mfp_nexys4_ddr.v`, the display will not be driven. Use the existing I/O devices (the GPIO modules for the LEDs, switches, and buttons) as a model for these modifications.

Double check your design and make sure you have brought the signals that drive the 7-segment display to ports on your top level module (normally `mfp_nexys4_ddr.v`) and that the port names match those in the constraints file and that there is a one-to-one correspondence between any of the uncommented signals in the constraint file and the ports in your top level. Your design will not work if this is done incorrectly. Synthesize the design and review the warnings carefully (there will be many warnings generated from some of the modules in `rtl_up`). If there are errors and synthesis fails, resolve the errors and re-synthesize. After successfully synthesizing your design, implement the design and generate the bitstream in Vivado. NOTE: IF THESE INSTRUCTIONS ARE CONFUSING, REVIEW THE GETTING STARTED WITH HARDWARE GUIDE.

Download the reference software

- Create a new project in C4E, and import the source code files in the `software` folder.
- Modify `main.s` to change the 7 Segment addresses to the addresses that you have assigned the 7-segment peripheral registers to in `mfp_ahb_const.vh` file.

- Build and debug your project on the board. When running the code, toggle the switches and notice the numbers and characters display on the 7-segment module.

Design the motion indicator and compass

Design the software implementation for the motion indicator and compass in MIPS assembly language. Make use of flowcharts and state transition diagrams as appropriate – these are good things to include in your Theory of Operation! Compile and debug the code to eliminate the bugs in your design.

Download your design to the board and test it

Verify that the indicators operate as specified in the functional specification. Verify that all buttons perform as specified.

Write a Theory of Operations document for your implementation

It is important that you document your design so that we can correctly grade your project. Your *Theory of Operations* should include a short summary of the design and then technical documentation describing your implementation. Make use of the flowcharts and state transition diagrams you created when you did your design to explain your code. Select snippets (short pieces of code, typically less than 20 lines) that you feel deserve further explanation and include them, along with a concise description of how they work. The goal for your design report is to provide insight into how your design works.

While you are at it, make sure your code is clearly documented. Include headers for all of your source code with your name, project, and a short description of what the module or assembly language file does. **Comment your code liberally.** This is especially true with assembly language code which is difficult to follow even if it's well-written. You should have one useful comment for each line (or every few lines) is appropriate. You will be docked points if you don't do this!

To be submitted:

- When your design is working correctly, organize the deliverable files along with your *Theory of Operations* and e-submit them in D2L. The files should be delivered in a single .zip file with the last names of the project team in alphabetic order: <last name1>_<last name2>_proj1.zip. (example: *jones_smith_proj1.zip*)[1]

  Your deliverables for this project should include:
  - A written Theory of Operation (5 - 8 pages) explaining the operation of your new indicators. Be sure to note any successful additions to the design spec that you have implemented in your design.

[1] It is always amazing to me how many people do not follow these directions. Not following directions will be a big problem when you are industry, so it is a good habit to get into now!

- Source code for all of the Verilog modules in your peripheral design
- Whole Vivado project files (we need to be able to recreate your Vivado project from it); more below.
- Source code for all of the MIPS assembly language files you wrote for the project.
- A video record to demonstrate all the functionalities; more below

- Record a video demo to show all the functions listed in table 5. You can submit the video to YouTube as private video and share the link at the beginning of the report.

- Additionally zip the whole Vivado project file contain the xpr file. Please test between partners to make sure your partner can recreate your project on their computer by only clicking the xpr file – it will happen if you have all the files included in the zipped file.

NOTE: PLEASE RETURN THE JUMPER SETTINGS ON ANY OF THE NEXYS A7 BOARDS IN THE LABS TO THEIR DEFAULT SETTING (JTAG DOWNLOAD, SO THAT THE SELF-TEST STARTS UP WHEN POWER IS APPLIED)

Related Documents

[1] *Nexys A7 Reference Manual*, Digilent Inc.
(https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual)

[2] *A Guide to Debouncing*, Jack Grassle, June 2008 (good reference)