Proj2Demo Example Design Description
(Last updated 29-Jan-2021)

Proj2Demo provides a partial framework that must be extended to get the demo running. The design example also includes an assembly language program (the "firmware") that demonstrates the basic functions of Rojobot31. The Proj2Demo framework can be extended to complete Project 2.

User Interface

The Proj2Demo firmware implements a modified version of SimpleBot from Project #1. Instead of simulating the compass with a counter, Proj2Demo controls an instance of the Rojobot31 moving on a simple world map. The compass is still present but the chase indicator has been simplified and the position of the Rojobot is also displayed. As with SimpleBot, the pushbuttons are used to tell the Rojobot what to do, but in this demo the Rojobot responds by moving through its world map.

PUSHBUTTONS

The pushbutton switches control two wheel motors and reset the system as shown:

Pushbutton	Motor Function
BTN_LEFT	Left Forward
BTN_UP	Left Reverse
BTN_RIGHT	Right Forward
BTN_DOWN	Right Reverse
BTN_CENTER	Not Used
BTN_CPU_RESET	System Reset

If *both* of the buttons that control a motor are pushed the actions cancel each other leaving that motor stopped. If all 4 buttons are pushed at the same time they cancel each other leaving both motors stopped.

LEDS

The low order (rightmost) eight LEDs on the Nexys4 DDR board are driven with the current value of the Rojobot *Sensors* register. The contents of the *Sensors* register are described in the *Rojobot31 Functional Spec.* The high order eight LEDs are unused by the demo.

SEVEN SEGMENT DISPLAY

Digits[7:5] (the leftmost) of the display show the current heading (0 degrees, 45 degrees, etc.) of the Bot.

Digit[4] of the display shows the current movement (**F**orward, **b**ackwards, slow **L**eft turn, slow **R**ight turn, fast **l**eft turn, fast **r**ight turn, or **H**alted).

Movement	BotInfo Register Value	Display Character	Character Code (Hex)
Halted (stopped)	00	H (upper case H)	0x17
Forward	04	F (upper case F)	0x0F
Reverse	08	b (lower case B)	0x0B
Slow left turn (1X)	0C	L (upper case L)	0x18
Fast left turn (2X)	0D	l (lower case L)	0x1A
Slow right turn (1X)	0E	R (upper case R)	0x19
Fast right turn (2X)	0F	r (lower case R)	0x1B

Digits[3:2] of the display show the current X coordinate (column) of the Rojobot's location. The output is in Hex.

Digits[1:0] of the display show the current Y coordinate (row) of the Rojobot's location. The output is in Hex.

Decimal point 0 (rightmost) – Blinks in response to the *upd_sysregs* signal. You should observe the decimal point blinking rapidly.

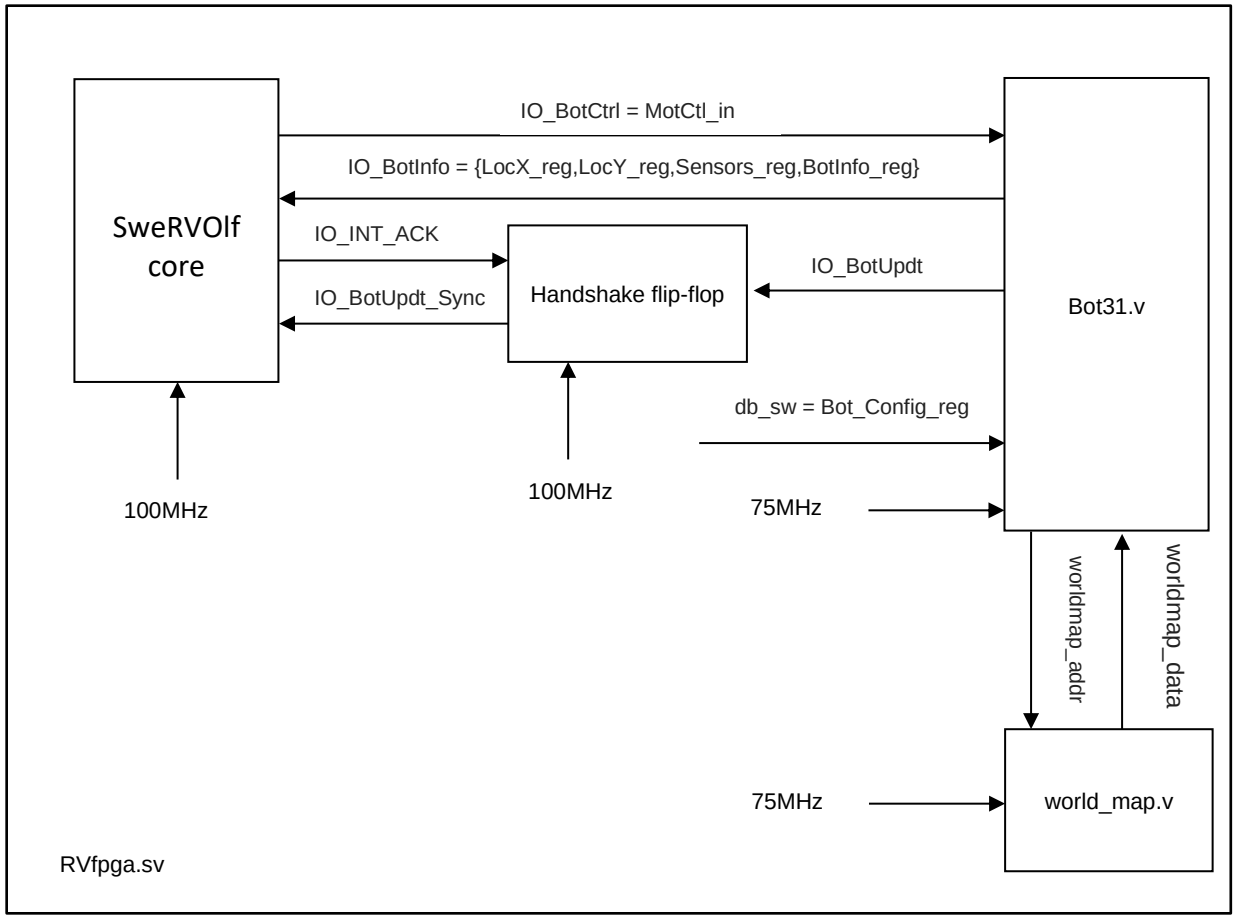
SLIDE SWITCHES

Switches[15:8] are not used in Proj2Demo.

Switches[7:0] can be used to set the Rojobot configuration. These switches are only read by Proj2Demo when the program first starts so you will have to restart the Rojobot between configuration changes.

Hardware

The block diagram below shows the design hierarchy for the demo:



You can start your Project 2 from your system in Project #1. Although the application is considerably different, Proj2Demo also makes use of the buttons, switches, LEDs, and the 7-segment display. You will have to modify the `rvfpga.v` to instantiates two new modules we provide in the Project #2 release. You will also need to modify the `gpio` module and all the associated files in the hierarchy to add input and output registers to control the Rojobot. Alternatively, you could create a new Wishbone bus peripheral to connect the I/O registers in the Rojobot with the application CPU

Rojobot31

Rojobot31 implements the Rojobot emulation. It has been packaged into a Vivado IP block and is added to your design in a manner similar to the way you added the clock wizard/clock generator to your project. The register interface to the Rojobot is described in the *Rojobot31 Functional Specification*. Additional technical details about the implementation can be found in the *Rojobot31 Theory of Operation*. The Project #2 write-up provides instructions for adding the ECE 540 IP repository and the Rojobot31 to your design.

World_map

world_map is a dual port memory storing the "world" that the Rojobot is moving through. There are three maps provided in the release, select *world_map_part1* for the demo. The world maps are generated using the memory generation wizard in the Vivado IP Integrator. You don't need to develop additional maps for Project #2 but there is a Botworld development kit that you can use to generate your own worlds if you'd like.

Handshaking flip-flop

The Proj2Demo program assumes the existence of a Set/Reset flip-flop that can be queried by the program to determine whether the Rojobot emulation has updated the state of the Rojobot. You will need to add this flip-flop to your design in a way that is accessible by the Proj2Demo. Practically speaking this means that this flip-flop needs to be accessible on the Wishbone bus as a register bit that can be read and cleared under program control. The Rojobot emulator indicates that it has updated the state of the Rojobot by toggling its *upd_sysregs* signal. The following code implements the handshaking flip-flop. *IO_BotUpdt* is the *upd_sysregs* signal from the Rojobot. IO_INT_ACK is a signal that is controlled via an I/O port. Since the *IO_BotUpdt* signal last for more than one 75MHz clock period this logic should capture every occurrence in the flip-flop and sustain it until the flip-flop is cleared by the application.

```
always @ (posedge clk50) begin
  if (IO_INT_ACK == 1'b1) begin
     IO_BotUpdt_Sync <= 1'b0;
  end
  else if (IO_BotUpdt == 1'b1) begin
     IO_BotUpdt_Sync <= 1'b1;
  end else begin
     IO_BotUpdt_Sync <= IO_BotUpdt_Sync;
  end
end // always
```

Software

The *firmware* directory in the Project #2 release contains the assembly language program for the demo. The majority of the Project 2 program that you need to write is identical to this application. You will have to add the code to do black line following but the basic functionality and support functions are present in the demo. The next few sections describe some of the functionality in the application; studying those examples, and the remainder of the code in the demo will be a worthwhile exercise in that it will help you code the additional code for Project #2.

NOTE: THE DEMO ASSUMES THE PUSHBUTTONS INPUT HAS THE FOLLOWING BIT ORDERING:

- BIT[7:5] : RESERVED
- BIT[4]: BTN_CENTER
- BIT[3]: BTN_LEFT
- BIT[2]: BTN_UP
- BIT[1]: BTN_RIGH
- BIT[0]: BTN_DOWN

MAIN LOOP

Like most embedded programs, the demo program is based on a main (infinite) loop. The program enters the main loop (label *main_L0*) after initializing the tables and variables used in the application. The main loop starts with a busy-wait loop that tests the state of a handshaking flip-flop to determine whether the Rojobot has updated its state. After the loop exits the busy-wait loop the application clears the handshaking flip-flop by toggling the `IO_INT_ACK signal` and continues by calling functions (*next_loc()*, *nxt_mvmt()*, and *next_hdg()*) that synchronize the program state to the state of the Rojobot. The loop continues (label *main_L2*) by writing the new digits to the 7-segment display, writing the Sensor register to the LEDs, and getting the next step for the Rojobot (*next_step()*) by reading the pushbuttons.

MOVEMENT DIRECTION TO CHARACTER CODE CONVERSION – MVMT2CC()

The *init_mvmttbl ()* and *mvmt2cc ()* functions provide an example for set up and access a lookup table in memory.

The *init_mvmttbl ()* function loads register t3 with the base address of the lookup table. It then loads constants (SP_MVMT0, SP_MVMT1 …) into sequential data locations. *mvmt2cc ()* expects the value to convert in register s9 and does the lookup by simply adding that value to the base address of the lookup able and using base addressing to get corresponding value from the table. The character code is returned in register t3.

CALCULATING THE NEW HEADING – NEXT_HDG()

The *next_hdg()* function is used to translate the RojoBot's orientation into degrees. The demo program uses what amounts to a case statement to do the translation. The *next_hdg()* function gives an example of how to implement a C **switch** (case) statement in assembly Language. The source code is documented, but the short explanation is that each case consists of a compare/jump combination which compares the switch statement value with one of the valid cases and jumps to the next case if the compare fails. The **break** statement for each case is done with a jump to the end of the switch code.

NEXYS4 I/O MODULES API (APPLICATION PROGRAM INTERFACE)

The LEDs, pushbuttons and switches and the seven segment display digits and

decimal points are accessed by using short functions which, to some degree, isolate the hardware specifics from your application. The demo program uses the following functions:

DEB_rdbtns() SS_wrdigx() SS_wrdpts() LED_wrleds()

Examine the source code comments for calling conventions, usage, and functionality and, if necessary, refactor those functions for your hardware.