Project 1: SimpleBot

Theory of Operations

Chuck Faber

Portland State University - ECE 540 - Winter 2020

I. Design Summary

In this project we were tasked with creating a simulation of a simple 2-wheel robot, with a seven segment display which would display a motion animation and its compass heading based on the input of 4 buttons to control the left and right motors. The compass heading would be determined with dead reckoning, and the project firmware would be implemented using the Nexys A7 Development board, and the software was to be written in RISC-V assembly language.

Several modifications needed to be made to the firmware to enable hardware, such as the push buttons and extended capability of the seven segment display (more characters and addition of decimal points). Primary goals in the software were responsiveness of the buttons and cleanness of the displays with few to no bugs in the display of the headings and motion indicator.

II. Firmware Modifications

Implementing new GPIO registers

Implementing Buttons in Constraints File

In order to get the push-buttons connected, I first had to add their mapping in the firmware constraints file. See Box 1.

Box 1: rvfpga.xdc - Mapping button pins to new signals in firmware code

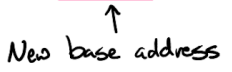
```
## Buttons
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { i_btn_u }];
set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { i_btn_d }];
set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { i_btn_l }];
set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { i_btn_r }];
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { i_btn_c }];
```

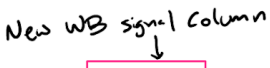
Mux Code / New Address Declaration

A new GPIO module needed to be instantiated in swervolf_core.v. A new base address for this GPIO module also needed to be included in wb_intercon.v with its own signals, as shown in box 2 below.

Box 2: wb_intercon.v - Added new base address and column to Wishbone Mux

```
wb_mux
  #(.num_slaves (8),
    .MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400, 32'h00001800, 32'h00002000}), // Added 0x00001800 base address
    .MATCH_MASK ({32'hfffff000, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hffffffc0, 32'hfffff000}))
wb_mux_io
   (.wb_clk_i   (wb_clk_i),
    .wb_rst_i   (wb_rst_i),
    .wbm_adr_i  (wb_io_adr_i),
    .wbm_dat_i  (wb_io_dat_i),
    .wbm_sel_i  (wb_io_sel_i),
    .wbm_we_i   (wb_io_we_i),
    .wbm_cyc_i  (wb_io_cyc_i),
    .wbm_stb_i  (wb_io_stb_i),
    .wbm_cti_i  (wb_io_cti_i),
    .wbm_bte_i  (wb_io_bte_i),
    .wbm_dat_o  (wb_io_dat_o),
    .wbm_ack_o  (wb_io_ack_o),
    .wbm_err_o  (wb_io_err_o),
    .wbm_rty_o  (wb_io_rty_o),
    .wbs_adr_o  ({wb_rom_adr_o, wb_sys_adr_o, wb_spi_flash_adr_o, wb_spi_accel_adr_o, wb_ptc_adr_o, wb_gpio_adr_o, wb_gpio_a_adr_o, wb_uart_adr_o}),
    .wbs_dat_o  ({wb_rom_dat_o, wb_sys_dat_o, wb_spi_flash_dat_o, wb_spi_accel_dat_o, wb_ptc_dat_o, wb_gpio_dat_o, wb_gpio_a_dat_o, wb_uart_dat_o}),
    .wbs_sel_o  ({wb_rom_sel_o, wb_sys_sel_o, wb_spi_flash_sel_o, wb_spi_accel_sel_o, wb_ptc_sel_o, wb_gpio_sel_o, wb_gpio_a_sel_o, wb_uart_sel_o}),
    .wbs_we_o   ({wb_rom_we_o,  wb_sys_we_o,  wb_spi_flash_we_o,  wb_spi_accel_we_o,  wb_ptc_we_o,  wb_gpio_we_o,  wb_gpio_a_we_o,  wb_uart_we_o }),
    .wbs_cyc_o  ({wb_rom_cyc_o, wb_sys_cyc_o, wb_spi_flash_cyc_o, wb_spi_accel_cyc_o, wb_ptc_cyc_o, wb_gpio_cyc_o, wb_gpio_a_cyc_o, wb_uart_cyc_o}),
    .wbs_stb_o  ({wb_rom_stb_o, wb_sys_stb_o, wb_spi_flash_stb_o, wb_spi_accel_stb_o, wb_ptc_stb_o, wb_gpio_stb_o, wb_gpio_a_stb_o, wb_uart_stb_o}),
    .wbs_cti_o  ({wb_rom_cti_o, wb_sys_cti_o, wb_spi_flash_cti_o, wb_spi_accel_cti_o, wb_ptc_cti_o, wb_gpio_cti_o, wb_gpio_a_cti_o, wb_uart_cti_o}),
    .wbs_bte_o  ({wb_rom_bte_o, wb_sys_bte_o, wb_spi_flash_bte_o, wb_spi_accel_bte_o, wb_ptc_bte_o, wb_gpio_bte_o, wb_gpio_a_bte_o, wb_uart_bte_o}),
    .wbs_dat_i  ({wb_rom_dat_i, wb_sys_dat_i, wb_spi_flash_dat_i, wb_spi_accel_dat_i, wb_ptc_dat_i, wb_gpio_dat_i, wb_gpio_a_dat_i, wb_uart_dat_i}),
    .wbs_ack_i  ({wb_rom_ack_i, wb_sys_ack_i, wb_spi_flash_ack_i, wb_spi_accel_ack_i, wb_ptc_ack_i, wb_gpio_ack_i, wb_gpio_a_ack_i, wb_uart_ack_i}),
    .wbs_err_i  ({wb_rom_err_i, wb_sys_err_i, wb_spi_flash_err_i, wb_spi_accel_err_i, wb_ptc_err_i, wb_gpio_err_i, wb_gpio_a_err_i, wb_uart_err_i}),
    .wbs_rty_i  ({wb_rom_rty_i, wb_sys_rty_i, wb_spi_flash_rty_i, wb_spi_accel_rty_i, wb_ptc_rty_i, wb_gpio_rty_i, wb_gpio_a_rty_i, wb_uart_rty_i}));
```





Passing in the Buttons as an Array

The buttons declared in the constraints file needed to be connected to the swervolf_core module through a newly declared 32 bit io_data port which was done as follows. Note that the unused 27 most significant bits allows future addition of new peripherals to this GPIO module as shown in box 3. Also the order in which the buttons inputs are passed to the io_data port are important to consider as well, as they will determine how to decode the values read from the new GPIO module's register.

Box 3: rvfpga.sv - swervolf_core instantiation -- passing in button press inputs to second gpio port

```
    .io_data        ({i_sw[15:0],gpio_out[15:0]}),
    .io_data_a      ({27'bz,i_btn_u,i_btn_d,i_btn_l,i_btn_r,i_btn_c}),
```

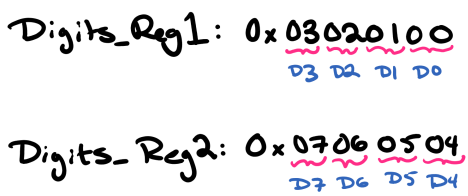
Seven Segment Display

The project requirements ask us to implement new characters to be used for the seven segment display. The current decoder only took 4 bits in, which allowed us to choose between 16 different characters, so to expand the number of possible characters, we had to expand this input to 8 bits.

The goal was to implement all the characters with their associated values from Table 2 of the project description which added 16 additional characters to be able to be displayed.

The current method in which the seven segment decoder was implemented was to use a single 32 bit Digits_Reg to store 8 groups of 4-bit segments, each which was responsible for a single digit in the seven-segment display.

Instead, I wanted each byte to represent a single digit on the SSD, and so I would need a second Digits_Reg register. For example, to write the number 76543210 to the SSD, I would store the following values in Digits_Reg1 and Digits_Reg2:



And each register would correspond to the following digits on the SSD:

Implementing New SSD Digits Register

The Enables_Reg and the Digits_Reg base addresses were implemented in a switch case in swervolf_syscon.v and the looked as follows (note some changes implemented to include a decimal enable functionality into the Enables_Reg):

Box 4: swervolf_syscon.v - Digits_Reg2 and adding 8 bits for decimal point enables in Enables_Reg address

```
14 : begin //0x38-3B 0x80001038 -- LSB is digit enables, 2nd byte is decimal enables.
   if (i_wb_sel[0]) Enables_Reg[7:0]  <= i_wb_dat[7:0];
   if (i_wb_sel[1]) Decimals_Reg[7:0]  <= i_wb_dat[15:8];
end
15 : begin //0x3C-3F 0x8000103C for second digits reg.
     if (i_wb_sel[0]) Digits_Reg2[7:0]   <= i_wb_dat[7:0];
     if (i_wb_sel[1]) Digits_Reg2[15:8]  <= i_wb_dat[15:8];
     if (i_wb_sel[2]) Digits_Reg2[23:16] <= i_wb_dat[23:16];
     if (i_wb_sel[3]) Digits_Reg2[31:24] <= i_wb_dat[31:24];
end
```

Each case in this switch case represented an offset address with each new case being 4 addresses more than the previous, with a byte select of 0-3. To add a new register, I implemented a new case at ‘9’ which represented the address 0x80001024, and I had this represent Digits_Reg1 (digits 0-3) and let the prior case at ‘15’ represent Digits_Reg2 (digits 4-7).

We didn’t end up needing digits 4-7 for this project, but I felt it was useful to have them accessible in case we might need them later.

Box 5: swervolf_syscon.v - Added new case/address (0x80001024) for new Digits_Reg

```
        // Added new register to accommodate more digits.
        9 : begin //0x24-27 Addresses 0x80001024 for first digits reg
            if (i_wb_sel[0]) Digits_Reg1[7:0]   <= i_wb_dat[7:0];
            if (i_wb_sel[1]) Digits_Reg1[15:8]  <= i_wb_dat[15:8];
            if (i_wb_sel[2]) Digits_Reg1[23:16] <= i_wb_dat[23:16];
            if (i_wb_sel[3]) Digits_Reg1[31:24] <= i_wb_dat[31:24];
        end
```

Implementing decimal point enables

As with the push buttons, the first thing needed to implement decimal point functionality was to enable it in the constraints file. In our case it only required uncommenting a line.

Box 6: rvfpga.xdc - Mapping decimal point cathode to DP signal in firmware

```
set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { DP }];
#IO_L19N_T3_A21_VREF_15 Sch=dp
```

Adding the 8 bit enable to the prior digits Enable Register

For decimal point functionality, I needed a place in memory to store which digits had their decimal points enabled, so I added this functionality to the already existing Enable_Reg base address, since only the 8 least significant bits were being used to enable or disable the digits. I used the next byte as a select for whether or not a decimal would be enabled. See code in Box 4.

Adding the SevSegMux

The decimal point requires its own seven segment time multiplexer which outputs to the single bit wide DP signal, similar to how the other cathode segments are implemented. This was implemented as follows below in box 6.

Box 7: swervolf_syscon.v - Instantiation of SevSegMux that reads Decimals_Reg and outputs time muxed DP signal

```
SevSegMux
#(
  .DATA_WIDTH(1),
  .N_IN(8)
)
Select_Decimals
(
  .IN_DATA(decimal),
  .OUT_DATA(DP),
  .SEL(countSelection[(COUNT_MAX-1):(COUNT_MAX-3)])
);
```

III. Software

After the hardware had been modified, the software needed to be implemented. The software was written in the RISC-V assembly language. A full flowchart describing the how the main loop of the program was implemented can be found below.

In summary, a forever loop runs, each time reading the combination of button presses to direct it to the appropriate case in a switch case statement. Counters are incremented on each run through the loop, and conditionals check whether these counters hit an appropriate value to simulate the 5 Hz or 10 Hz updates required for the compass and motion indicators.

This method was used to ensure the program remained fast and responsive to user input, as the button value is checked upon every single loop and there are no time intensive loops where the program could get stuck doing something and not be able to read user input. Using conditional checks to implement the 5Hz and 10Hz updates makes each run through the loop before the next button check faster.

Flow Chart Diagram of Main Loop

See Appendix A for a full flow chart of the software design.

Jump Tree and Button Mapping

As mentioned earlier, the button inputs are mapped to the lowest 5 significant bits of the new GPIO input register as follows: `{i_btn_u,i_btn_d,i_btn_l,i_btn_r,i_btn_c}`. On each loop of the main function, we must read in the register, mask all but the lowest 5 bits, and for this assignment, we logically shift the value to the right to eliminate the center button press so we only have to deal with 4 bits instead of 5.

This gives us a possible value range of 0 to 15, or b0000 to b1111. Using the requirements of the project, I mapped the combinations of button presses to each of the 7 possible motion modes: forward, reverse, left at 1x speed, left at 2x speed, right at 1x speed, right at 2x speed, and stopped. I then extrapolated the binary value that would appear in the GPIO register in each of these combinations and converted them to decimal values.

My assembly switch case compares the value to the defined decimal values in the third table below to choose the appropriate branch to take in the main loop.

Table 1: Mapping of motion modes to button presses according to project requirements.

Motion Modes						
Forward	**Reverse**	**Left 1x**	**Left 2x**	**Right 1x**	**Right 2x**	**Stop**
BTNL+BTNR	BTNU+BTND	BTNR	BTNR+BTNU	BTNL	BTNL+BTND	None
		BTNU		BTND		BTNL+BTNU
		BTNR+BTND+ BTNU		BTNR+BTND+ BTNL		BTNR+BTND
		BTNL+BTNU+ BTNR		BTNL+BTNU+ BTND		BTNU+BTND+ BTNL+BTNR

Table 2: Binary values that would be present in the register for each mode

Binary Values						
Forward	**Reverse**	**Left 1x**	**Left 2x**	**Right 1x**	**Right 2x**	**Stop**
0011	1100	0001	1001	0010	0110	0000
		1000		0100		0101
		1011		0111		1010
		1101		1110		1111

Table 3: Conversion of binary values to decimal values for switch case

Decimal Values						
Forward	**Reverse**	**Left 1x**	**Left 2x**	**Right 1x**	**Right 2x**	**Stop**
3	12	1	9	2	6	0
		8		4		5
		11		7		10
		13		14		15

Below is a snippet of the assembly code used to code the switch case or jump tree:

Box 8: project1_asm.s - Jump tree code from software in RISC-V assembly code

```
loop:
    li  t0, GPIO_A_IN       # Address to read button presses
    lw  t1, 0(t0)           # Read button press values
    andi t1, t1, 0x1F       # Mask just the lowest 5 bits
    srli t1, t1, 1          # Shift all bits to right by 1 (masking center push button)

    # Jump Tree Here:
    li  t2, SEV_SEG_DIG1    # Address to SSD digit display 0-3
    li  t3, 3               # If t1 == 3 jump to forward
    beq t3, t1, forward
    li  t3, 12              # If t1 == 12 jump to reverse
    beq t3, t1, reverse
    ...
    ...
    j   stop                # Else go to stop
```

hex2bcd Function

In addition to the main loop, I implemented a function in the assembly code which would convert a literal value (in hex or decimal) to its display in binary coded decimal. Below you can view the pseudocode of the hex2bcd function. Actual implementation was done in RISC-V assembly.

Box 9: Pseudocode for the hex2bcd function.

```
function hex2bcd (int val):
    if val >= 100:
        for (i = 3; i > 0; i--)
            if val >= i*100:
                write i to DisplayRegister[Byte 2]
                val -= i*100
                break
    else:
        write 0 to DisplayRegister[Byte 2]
    if val >= 10:
        for (i = 9; i > 0; i--)
            if val >= i*10:
                write i to DisplayRegister[Byte 1]
                val -= i*10
                break
    else:
        write 0 to DisplayRegister[Byte 1]
    write val to DisplayRegister[Byte 0]
```

Appendix A - Software Flow Chart