Artem Kulakevich
Professors Brian Cruikshank & Ataur Patwary
TA Deepen Parmar
ECE-540 System on Chip
12 Oct 2020

HW_LAB_2: SimpleBot

Theory of Operations:

First things first, I prefer to do as much of the code as possible in assembly because it means less synthesis or implementation when I screw up. So with that in consideration, I decided to implement my decade counter in assembly instead of Verilog. There are several main tasks required here:

1. **Implement push buttons.** This requires creating a second GPIO module because according to the gpio_defines.v file, we can only have 32 ports to GPIO. I can modify the current GPIO to remove functionality from the switches or LEDs, but that felt like "cheating" so I decided to implement a second GPIO.

To complete this task I did not touch gpio_top, but instead focused on how it is instanced in swervolf_core.v. I identified which inputs and outputs are necessary for the gpio_module instance, and compared those in/outs with timer_ptc. This was I was able to see which outputs were different and which outputs were the same. For the outputs that were difference I created the necessary wire instances in wb_intercon.v. So for "wb_gpio_dat_i" I created another copy called "wb_gpio2_dat_i". These copies had to be added to wb_mux_io instances so that our second gpio module could receive data:

```
#(.num_slaves (8), //modified 7 -> 8
.MATCH_ADDR ({32'h00000000, 32'h00001000, 32'h00001040, 32'h00001100, 32'h00001200, 32'h00001400,
32'h00001440, 32'h00002000}), // Added 00001440 for gpio2
.MATCH_MASK ({32'hfffff000, 32'hfffffffc0, 32'hfffffffc0, 32'hfffffffc0, 32'hfffffffc0, 32'hfffffffc0,
32'hfffffffc0, 32'hfffff000})) // Added 32'hfffffffc0, for gpio2

 wb_mux_io
   (.wb_clk_i  (wb_clk_i),
       .wb_rst_i  (wb_rst_i),
....
.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_gpio2_adr_o, wb_uart_adr_o}),
....
```

I ended up adding gpio 2 after gpio1, with the address 0x00001440. This also required doubling up the number of gpio bidrec dinstances, so I created additional "en_gpio2", "i_gpio2" etc. lines. And I expanded the io_data inout to be 64 bits. I also had to hookup the io_data pins to new output pins in RVfpga.sv, so I expanded the "i_sw" bus to size 21, and added the necessary pins P17, M17, M18, P18, and N17 to the rvfpga.xdc file.

2. **Add debounce to switches and pushbuttons.** This can be done in swervolf_core.v at some point between the i_sw pin and the input of the wb_mux. Since everything gets muxed in afterward, placing debounce is not an option since other modules will use the same bus.

In my opinion this was one of the most difficult tasks here, it was hard to find the right location to add the debounce without creating errors during synthesis. Placing the debounce between the "i_sw" the bidirec resulted in errors because multiple nets were driving some register input. Placing the debounce after the bidirec between the bidirec and gpio_module instance worked well. This resulted in no errors, and the functionality stayed put. I also modified the debounce file to deal with an 100Mhz clock instead of 50Mhz, since that's what we are supposedly running.

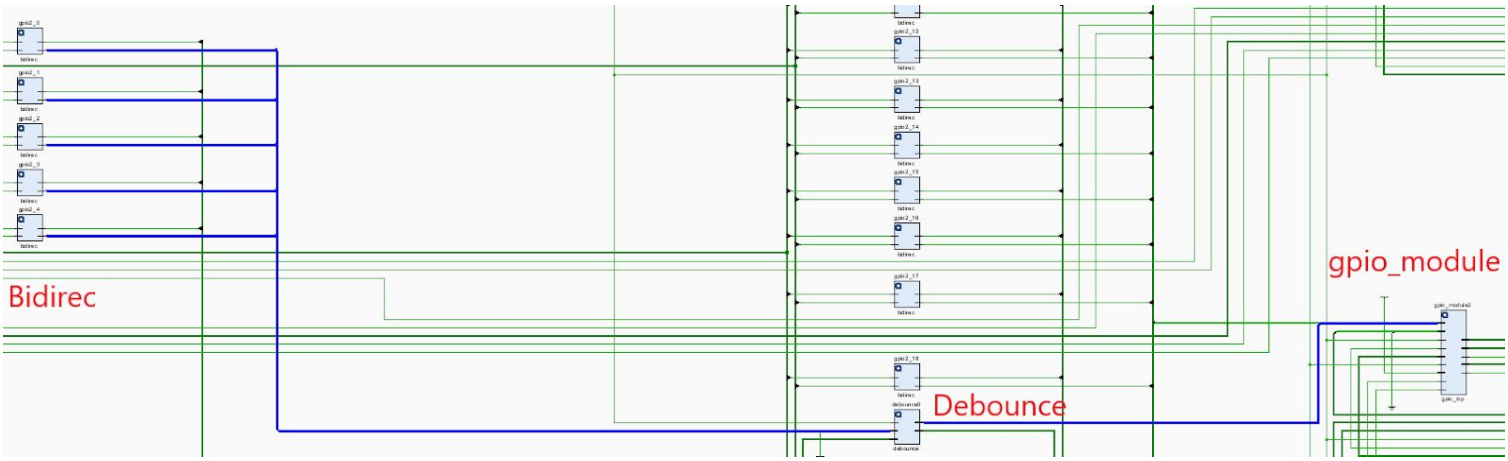


Figure x:

3. **Modify the Seven Segment Controller.** This is done entirely in swervolf_syscon.v. The address 0x80001038 is kind of wasted on the one enable_reg byte input. I decided to convert the 0x80001038 address input to address the four most significant bytes of the seven-segment display. While the address 0x8000103c will address the 4 least significant bytes. I also kept the original "Enable" functionality that 0x80001038 would provide by using the most significant bit of each 2 bytes to enable the output.

This was done by modifying the state machine that distributes the data depending on the address provided to swervolf_syscon.v. Here is an example of this modification:

Original:

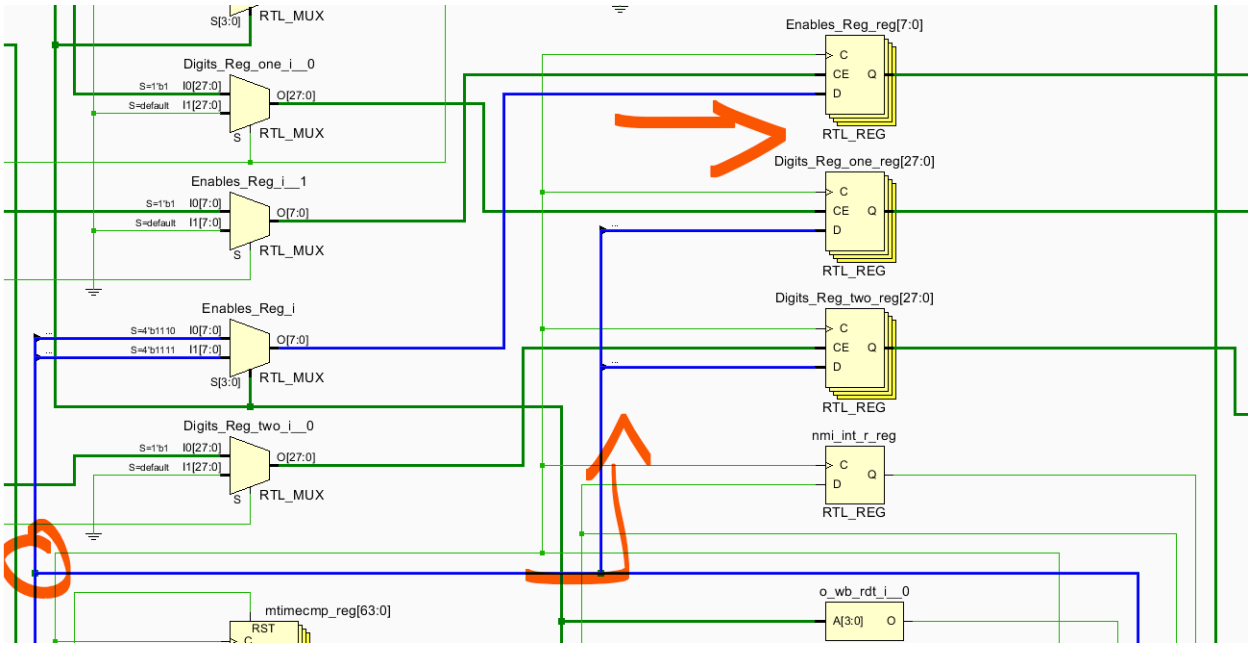
`14 : begin`
 	
`  if (i_wb_sel[0]) Enables_Reg[7:0]  <= i_wb_dat[7:0];`

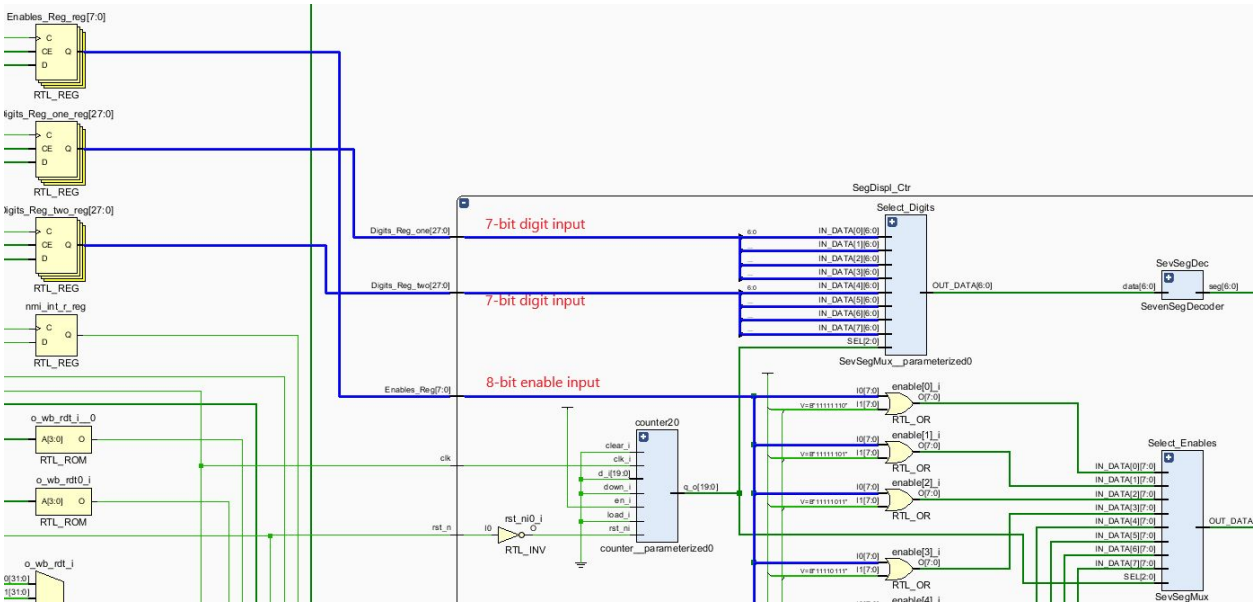
Modified:

```
14 : begin

   if (i_wb_sel[0]) begin Digits_Reg_two[6:0]    <= i_wb_dat[6:0];
                          Enables_Reg[4]         <= i_wb_dat[7];
end
```

So now i_wb_dat input that goes into swervolf_syscon is distributed between “Digits_Reg_two” and “Enables_Reg”, where the MSB is given to the enable register and the rest of the bits describe that wanted display digits.





As a result I have now have to write data differently to the addresses 0x8000103c and 0x80001038. I have to take into consideration that the MSB of every 2 bytes will disable/enable the output. For example, writing the following bitstream (0x04801d0a)

0	000	0100	**1**	000	0000	**0**	001	1010	**0**	000	1010

to 0x80001038 and 0x8000103c, would give me the following out on the 7segment display:

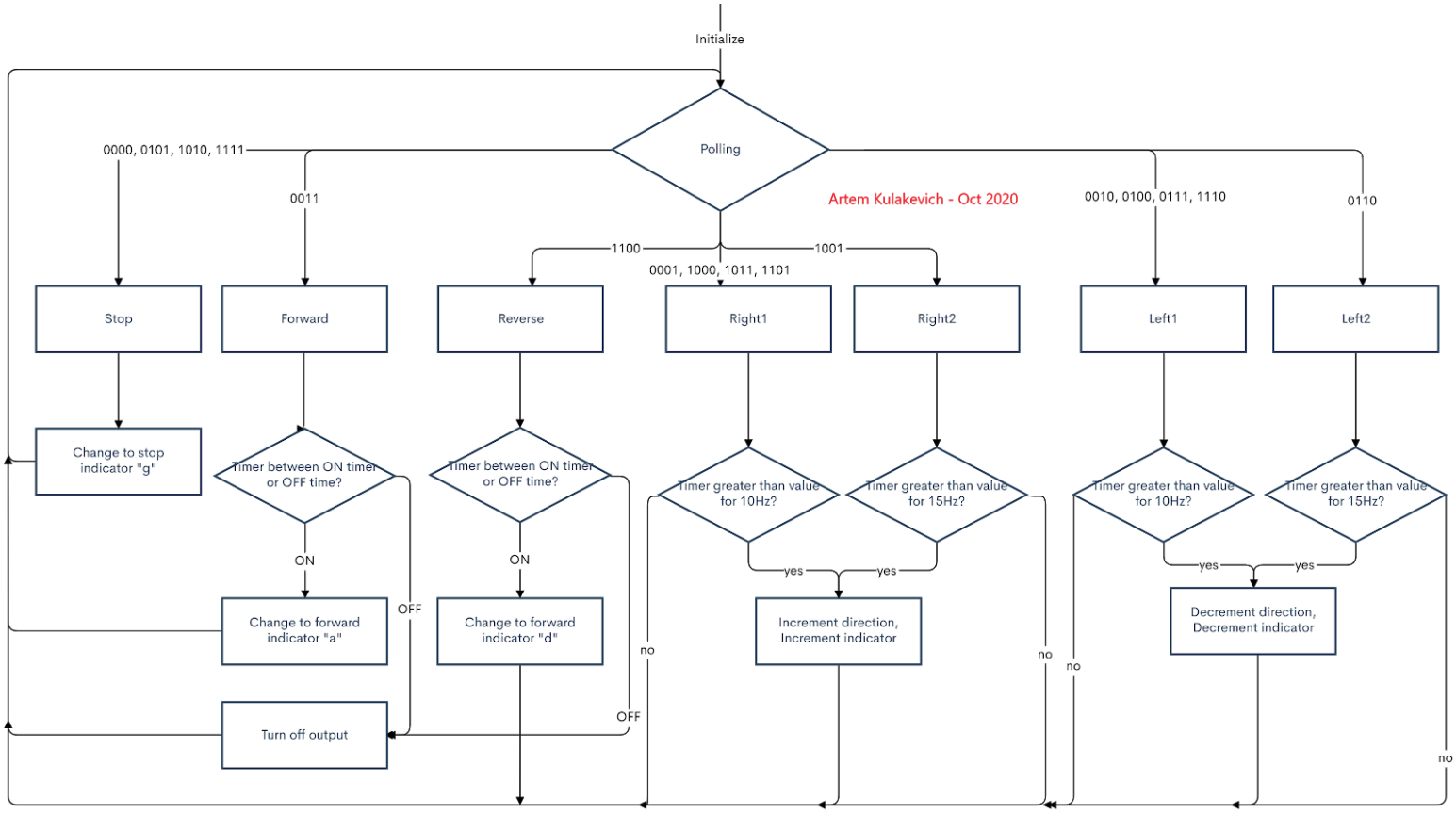
4	8	_	A	4	8	_	A

The MS Byte is 4, the second is disabled, the third is a “d”, and the fourth is “1”. The second is disabled because bit #23 is a 1 which disables our output.

With this method I have 7 bits for each digit now. This means I can create 128 different patterns for each digit which should be more than enough for all of these projects. These changes also required some other modifications to the constants and wires in syscon.v. Parameters like DATA_WIDTH had to be modified depending on the input.

4. **Software development in assembly.** The remaining logic I decided to implement in assembly, I prefer assembly over SV, so as much as possible I work there. I wrote up the

pseudo code for my program in the appendix. The following flow chart describes the whole program, but does need a bit more detail.



The program has 3 main storage registers, S1 holds a counter value, S3 holds the compass degrees value, and S4 holds the indicator value.

Polling reads the input from the buttons, and then has a long “case statement” where it goes to one of the 7 states dependings on the input from the buttons:

```
    li s2, 0x0000
    beq a1, s2, stop # 0000
    li s2, 0x0001
    beq a1, s2, right1 # 0001
    li s2, 0x0002
    beq a1, s2, left1  # 0010
....
```

So depending on the value of the register a1 (which stores the button output) we will go to different states.

Programs Stop, Forwards, and Reverse are pretty basic and simply change the value in S4. Stop will just set S4 to “g”, and Forward/Reverse will set “a”/”d” respectively depending on the timer value. This was I can have a slow pulse on the 7segment output.

Programs Right1, Right2, Left1, and Left2 are all very similar, they either decrement or increment the value at S4 and S3 to change direction of the indicator and adjust the compass degree value. They do this only if the timer register S1 has reached a certain value to allow for change frequency close to specified time of 10Hz or 15Hz.

Right1 and Right2 send to the same program if an increment is necessary, that funciton is called “display_inc”. This function combined with “num_inc” right after do the majority of the addition logic necessary to deal with the 7segment display. Since we want to see numbers 0-9, instead of 0x0 - 0xf, I have logic implemented that checks the first bit for overflow. For example:

```
andi t0, s3, 0xF      # Mask all bytes but LSB
li t1, 0xa            # Load value into t1 for comparison
bne t1, t0, comp_out  # Compare masked s3 value to 0xa
addi s3, s3, -0xa     # if, s3 = 0xa, then add -0xa
addi s3, s3, 0x0100   # increment the next byte
```

Here we have an example of this in play. If the least significant byte overflows to 0xa, we check for that and remove the overflow. Then we push the carry to the next byte. Since s3 is written to my 7segment display, the next byte has to be the third byte. Since each digit is receives 2 bytes, so everything has to be spaced out. For example if I wanted to write the value 359 to the output of the 7segment display. I would write, 0x030509. That way 09 would go the to the least significant digit, 05 to the 2nd least significant, and 03 to the third. I also have an overflow that checks for the value 0x360. Since we’re limited to an 0 - 359, at the value 360 we return back to 0 and never display 360.

This same process is done for Left1 and Left2, but with decrementing. So whenever we subtract enough to reach 0xf, then we subtract 0x6 in addition to get us down to the value 0x9. And we check for 0xf in the most significant bit to decide when we want to jump back up to 359.

After deciding a value for s3 and s4, we go to the polling function where we combine s4 and s3 and write that value to our 7segment output. For example if we had 0x19 (code for “g”) in s4, and 0x030509 in s3. Then we would perform the following bit of code:

```
        sll t0, s4, 24    # shift left into MSB
        or t0, s3, t0     # store s4 in s3
```

First we shift s4 to the left, so s4 = 0x19 becomes 0x19000000, and we or that with 0x030509, resulting in 0x19030509, and store that into our 7segment register:

Conclusion:

In conclusion, all requirements for this project were met. A few assumptions were made, including that 1Hz, 10Hz, and 15Hz were just guideline values, without using a timer, there is no easy way to get these frequencies to be exact. Since the length of the program greatly affects these frequencies, and parts of the code ran later in the case statement would increment slower than part that were ran earlier in the case statement. We have not been introduced to timers, and this was not a requirement so I assumed that these were ballpark figures. One of the greatest difficulties I had with this project was figuring out how to test my verilog code. I wish I explored the simulation more, because waiting to synthesize, implement, uploading the bitstream, and test is unbelievably inefficient. This could become a massive time sink in future projects, so it will be necessary to find alternative ways to test the verilog code before synthesis. So far the best method has been to inspect the schematic, but that only helps me find errors maybe 50% of the time. Additionally in the future I would like to introduce timers and interrupts, but lack the time to look at that now. Since this implementation would in no way fly in a professional requirement, but for this class the requirements are met.

Appendix:

Pseudo Code:
TA, TB, TC, TD = Counter values to give close to the necessary frequency of updates.

Initialize:
1. Set counters, and outputs to 0
2. Enable 7Seg display

Poll:
1. Read Pushbutton Output Register
2. Write data to 7Seg Output
    a. Combine compass with indicator
3. Match Pushbutton Output values to State
    a. States: **stop, right1, right2, left1, left2, forward, reverse**

Stop:
1. Write “G” output to indicator
2. Return to poll

Forward:
1. Turn off “A” indicator bits if below timer value TA
2. Turn on “A” indicator bits if between timer range TA and TB
3. Reset timer if above TB

Reverse:
1. Turn off “D” indicator bits if below timer value TA
2. Turn on “D” indicator bits if between timer range TA and TB
3. Reset timer if above TB

Right1:
1. If timer is below timer value TC, then return to Poll
2. If timer is above timer value TC, then reset timer, jump to Display_inc

Right2:
1. If timer is below timer value TD, then return to Poll
2. If timer is above timer value TD, then reset timer, jump to Display_inc

Left1:
1. If timer is below timer value TC, then return to Poll
2. If timer is above timer value TC, then reset timer, jump to Display_dec

Left2:
1. If timer is below timer value TD, then return to Poll
2. If timer is above timer value TD, then reset timer, jump to Display_dec

Display_inc:

1. Increment the compass values
2. Increment the indicator values
3. Check for indicator overflow, reset indicator if overflow reached
4. Check for compass values going above 0-9, reset back to 0
5. Increment second byte if overflowed, check if values go above 9
6. Increment third byte if overflowed, check if values go above 3
7. Reset counter if value above 3
8. Return to polling

Display_dec:

1. Decrement the compass values
2. Decrement the indicator values
3. Check for indicator overflow, reset indicator if overflow reached
4. Check for compass values going below 0-9, reset back to 9
5. Decrement second byte if overflowed, check if values go below 0
6. Decrement third byte if overflowed, check if values go below 0
7. Jump indicator value to 359 if below 0
8. Return to polling