ECE 540

PORTLAND STATE UNIVERSITY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SOC Design with FPGAs

ROJOBOT REPORT

Atharva Mahindarkar
atharva2@pdx.edu

Prasanna Kulkarni
prasanna@pdx.edu

Date: February 23, 2019

Contents

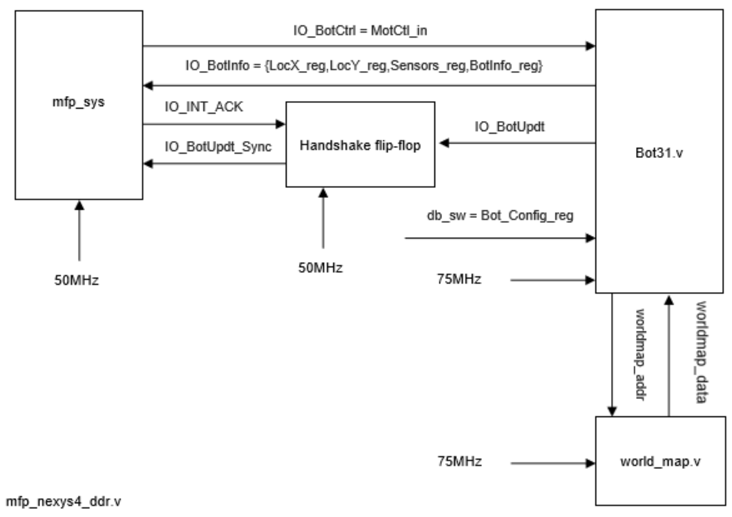
1 Hardware

1.1 RojoBot Driver

The RojoBot Driver is a peripheral that is added to the AHB-lite bus system that is implemented in the MIPSfpga system. There are two parts the read routine and the write routine. The switch case statement that drives the routine uses the value of the address to select the RojoBot peripheral. We drive the required port of the RojoBot using a switch case. A multiplexer in the AHB-lite interface dictates which data is being sent and the RojoBot peripheral is added to this multiplexer.

1.2 Worldmap

The Worldmap is generated using a block RAM. A COE file has been added to the block RAM and a .ngc file has been added to have the pre-loaded data of the world map. The world map is a 128 x 128 memory which has a path inscribed into it. The map also has obstacles. The idea of the RojoBot is to follow the path and stop at the obstacles. The world map needs to be scaled to the appropriate value to be diplayed on the screen. The scale module handles that. The DTG module that is provided with the release actually controls the sending of the values to the screen. The connection of this module with other modules is as shown in the diagram below.



1.3 Scale

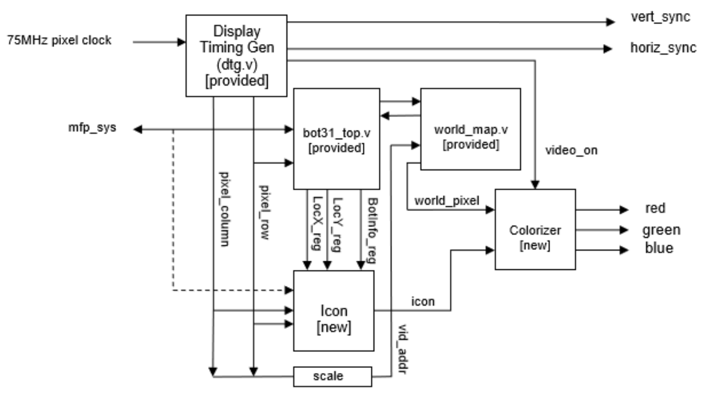
The scale module is where we bring up the dimensions of the map to a size which can be displayed upon a screen. What we essentially do here is divide the pixel_row and pixel_column values from the display timing generator that has been included in the release. Now a 1024 x 768 display will need to be scaled by 8 x 6 in order to display a 128 x 128 map. This is exactly what the scale module does.

1.4 Handshake Flip-Flop

The Handshake Flip-Flop is a set reset flip flop that can be queried by the program in order to check whether the RojoBot has already updated the state values. The code for this was essentially provided as an IP in the project relase and has not been modified.

1.5 Icon

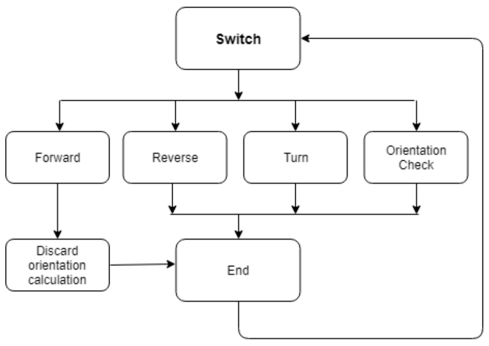
The Icon module is the module which diplays an icon for the RojoBot at the position of the RojoBot on the worldmap. The way this module works is by superimposing the icon, which is given using a COE file to a block RAM, upon the worldmap. Whenever the value of the pixel row and pixel column generated by the Display Timing Generator is in the range of the RojoBot's location the icon module sends the values for the icon to the colorizer which takes care of the superimposotion. The way all these modules are interconnected in the hierarchy is shown in the diagram below. There are eight block RAMs used in the module to store eight different icons that can represent the eight orientations.



1.6 Colorizer

The colorizer is the module that makes brings together all the hardware. What this module does, is to essentially send the appropriate VGA values to the output if we are writing the map. And if the icon is to be written it sends the VGA values for the icon to the output. The way we have implemented this is by using a switch case nested statement. The first case expression is for the icon which is used to ”draw” the icon. The Transparent pixel of the icon has the case statement for the world pixel. The world pixel case statement send the VGA values based on the values encountered in the block RAM. The icon does the same but with the icon.

2 Software



We have simulated 4 different states in form of switch, of our algorithm for black line follower RojoBot. Each state has its unique state code which gets updated after each execution of state. At the execution of state a decision is taken to jump to particular state or to be in the same state. We are using right turn only algorithm, which means we have to discard the opposite orientation with which the Rojo Bot arrives at each turn. Switch case codes for each state:

- forward - 0x00
- reverse - 0x01
- turn - 0x02
- orientation check - 0x03

2.1 Forward

In this forward state, first we update the state code, then check whether the bot is over black line or white space. If it is on black line, Motor control is given an input to go forward or if bot is over white space, then this changes its state code to 0x01 and jumps to reverse state. In this state we calculate a discard orientation for each forward movement of the bot. The discard orientation is a one on the opposite direction of the current orientation at turn. This means, when we arrive at turn and if our orientation is East (90), then we store West (270) as our discard orientation.

```
forward:
li      $s0, 0x00              # forward state code is 0x01
lb      $3, (SENSOR_REG)       # take sensor data
AND     $3, $3,MSKBLKL         # mask sensor data

beq     $3, 0x07, reverse_state_update      # if not on black line jump to
reverse
nop

.
.

reverse_state_update:
li      $s0, 0x01              # reverse state code is 0x01
j       end                    # jump to end
```

2.2 Reverse

In this state, first state code is updated and then sensor data is read to check whether the bot is over black line or white space, if it is on white space then bot moves reverse until it arrives on black line. After arriving on black line it calls for turn function, for this it just changes state code to 0x02. Before performing a jump to turn state, again its orientation is saved in register $t1 to check change in orientation in turn state.

```
reverse:
.
.
lb      $3, (SENSOR_REG)       # take sensor data
AND     $3, $3,MSKBLKL         # mask sensor data

.
.

beq     $0, 0x00, turn_state_update   # if back on black line jump to turn
nop
.
.
turn_state_update:
li      $s0, 0x02               # turn state code is 0x02
j       end                    # jump to end
```

2.3 Turn

This is a 45 degrees slow right turn state. When jumped after updating state code, again new orientation is retrieved from the bot and saved in register \$t2. Then orientations in registers \$t1 & \$t2 are compared. If both the values are same then, turn has not been performed yet. If this happens then motor control is given data to turn by 45 degrees and the state code is not changed as turn is not yet complete. If the orientations saved in registers are not equal it means it has performed turn of 45 degrees and now it jumps to orientation check.

```
turn:
bne     $t2, $t1, orientation_check_state_update # check change in orientation
nop
.
.
li      $2,0x30  # condition to move left motor forward only, i.e, slow right turn movement of the
bot
.
.
beq     $3, 0x07,reverse_state_update_for_turn # check if not on black line
nop
.
.
orientation_check_state_update:
# jump to orientation check state
li      $s0, 0x03
j       end                             # jump to end
.
.
reverse_state_update_for_turn:
li      $s0, 0x01
j       end
```

2.4 Orientation Check

Here first new orientation is compared with the discard orientation saved in $s7 in forward state. If the bot is on discard orientation, then state code is change and it jumps to turn state. If it is not on discard orientation and if bot is on white space it must jump to reverse and if bot is on correct orientation and black line then only bot is given permission to move forward

```
orientation_check:
.
.
beq     $t2, $s7, on_reject_orientation       # check whether bot is having same orientation as dis-
card orientation
nop
.
.
beq     $3, 0x07, reverse_state_update_forcheck      # if not on black line jump to reverse
nop
.
.
.
reverse_state_update_forcheck:
li      $s0, 0x01                        # reverse state code is 0x01
j       end                          # jump to end
.
.

on_reject_orientation:
li      $s0, 0x02
j       end
```

2.5 Discard orientation

This subroutine is defined to calculate discard orientation. Discard orientation is an orientation opposite (180 degrees out of phase) with the current orientation. Here first the current orientation is retrieved from bot registers and then a switch case is simulated. In switch case we have defined 8 different possible orientations which gives us discard orientation for each of those orientation.

```
calculate_discard_orientation:
li      $s7, 0x00
lb      $s7, (BOTINF_REG)      # $25[2:0] = BOTINF_REG[2:0] = orientation
and     $s7, $s7, MSKORIENT    # holds old orientation $t0 hold the value of the stable orientation

beq     $s7, 0x00, North_to_South # if current orientation is 0 degrees,    Discard orientation is
180 degrees
nop
.

.
North_to_South:
li      $s7, 0x04              # Discard orientation is assigned value 180 degrees
j       end                    # jump to end
```

3 References

- ECE-540 RojoBot project release
- The Math Coursework Template on Overleaf by Qiao Han.Liscenced under Creative Commons CC BY 4.0
- The MIPS Green Sheet by UC Berkeley at https://inst.eecs.berkeley.edu/ cs61c/resources/MIPS_Green_Sheet.pdf
- The MIPS Code Reference from the University of Idaho at http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html