

Final Report

Side Scroller Game Demo

Sidescroller Team

Ryan Fallis

Richard Romano

Keerthan Nayak

<https://github.com/Keerthan1994/Nexys-A7-Sidescroller-Game>

Report Submission: 12/10/2020

ECE 540 – SoC Design with FPGAs
Department of Electrical and Computer Engineering
Portland State University

Table of Contents

Current Status	1
Questions for Instructor	1
Team Planning	2
Meeting Notes	3
Final Report	9
1.0 Overview	11
1.1 Project Description	11
1.2 Sidescroller Specifications	11
2.0 Sidescroller Hardware	12
2.1 Rojobot Controller Theory of Operation	12
2.2 Rojobot - Map Muxing & Icon Improvements	12
2.2.1 Map Design	12
2.2.2 Map Muxing	13
2.2.3 Icon Improvements	13
2.3 Rojobot IP Block Modifications	15
2.3.1 Teleport & LocX/LocY Override	15
2.3.2 Jump & Fall:	16
2.4 BotSim31 (Picoblaze Core) Modifications	17
3.0 Sidescroller Software	19
3.1 Sidescroller Software Theory of Operation	19
4.0 Results & Challenges	20
4.1 The Good (Successes)	20
4.2 The Bad (Failures)	20
4.3 The Ugly (Challenges)	21
5.0 Team Contributions	21
5.1 Ryan Fallis	21
5.2 Richard Romano	21
5.3 Keerthan Nayak	21
6.0 Conclusion	22
7.0 BotSim 3.1 - IP Editing Guide	22

1.0 Overview

Crash Bandicoot XS is an iconic side scrolling platformer game that was featured on Game Boy. We attempted to replicate it on a pretty simplistic level using the Nexys A7.

Submission Note: Within our github we made a SS_Final_Project_Files folder this contains all of the important files we modified for the project.



Figure 0 - Crash Bandicoot XS (2002)

1.1 Project Description

The main goal of this project was to dig into the rojobot firmware and alter how it handled movement and orientation. Our second main goal was to create game levels using multiple maps stitched together. Our third goal was to remap the buttons to the accelerometer controls. This required that we further developed our understanding of the rojobot including: Generating and using ROMs, the vivado IP packager, the picoblaze soft core, the SPI port, and interfacing between all of the different major components.

1.2 Sidescroller Specifications

1. Re-colored map and title blocks
2. New icons to match Crash Bandicoot theme.
3. Multiple levels (Ideally 5)
4. Levels stitched together for smooth gameplay.
5. Controls rebuilt for Left, Right, and Jump
6. Accelerometer Controls

2.2.2 Map Muxing

Previously, the Rojobot had several maps that could be accessed by using the FPGA switches. In order to automate maps switching with our custom maps made to create a Crash Bandicoot like feel we needed to create an always block and mux. The always block utilizes conditional statements to set the control bit (current_map) for the mux depending on specific X coordinates in the LocX register. This logic is located in the sidescroller mux file, ss_mux.v. The always block / mux functions as a state machine based on three parameters, current map, previous state, and next state. Current map contains which map the bot is located on, previous state contains which map the bot was previously on, and next state contains which map the bot should be on when moving in the East direction. Figure 2 gives a representation of the necessary conditions to achieve switching into different maps.

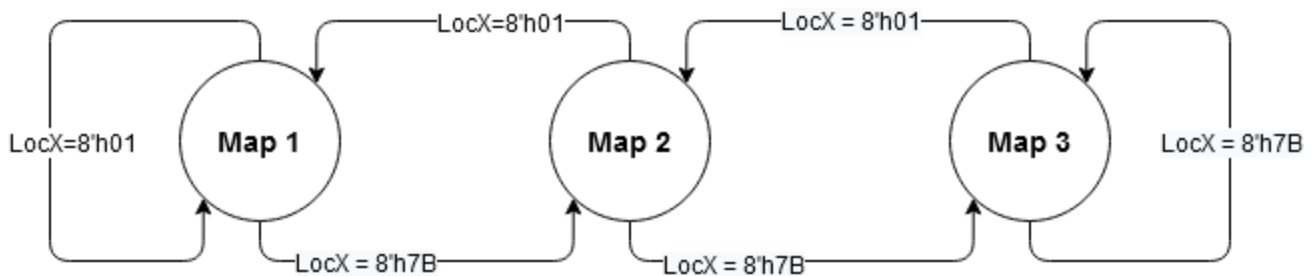


Figure 2 - Map Mux State Machine

The map switching corresponds to which current_map value is set. This current_map value updates the map that is displayed via VGA. Contrary to the naming convention that is used in the above state machine, the maps are numbered 0, 2, 3. While it would have been more intuitive to number them 0-2 or 1-3, this was how it was written in SystemVerilog, was tested, worked, and should be updated in a cleaner version of the code. Figure 3 below indicates how the current_map sets the worldmap_data and world_pixels that are associated with the three maps.

```
// mux to select map based on the SW (Will need to be updated once we have more maps.)
assign {worldmap_data, world_pixel} =
    current_map < 1 ? {worldmap_data_part_1, world_pixel_part_1} :
    (current_map < 3 ? {worldmap_data_lr, world_pixel_lr} :
    (current_map < 4 ? {worldmap_data_loop, world_pixel_loop} : {worldmap_data_loop, world_pixel_loop}));
```

Figure 3 - Map Mux SystemVerilog

2.2.3 Icon Improvements

The starting point for modifying the Rojobot icon is to understand how the example project implemented a Pikachu icon and doing a bit of reverse engineering to create our own version. A few key pieces of information include observing the pikachu.mem file as well as the robot_icon_v2.v file. In the

pikachu.mem file it can be seen that the icon has a 12 bit data width and a depth of 27,744. This doesn't mean all that much until looking into the robot_icon_v2.v parameters as shown in the image below.

```
module robot_icon_v2 #(
    parameter SCALING_FACTOR = 6,
    parameter MARGIN = 128,
    parameter ANIMATION_COUNTDOWN = 8_000_000,
    parameter SPRITE_COLS = 34,
    parameter SPRITE_ROWS = 34,

    localparam MARGIN_ROW = SPRITE_ROWS / 2 - 9,
    localparam MEM_ROWS = SPRITE_ROWS*8,
    localparam MEM_COLS = SPRITE_COLS*3,
    localparam SPRITE_SIZE = SPRITE_COLS * SPRITE_ROWS,
    localparam FRAME_ROW_SIZE = MEM_COLS * SPRITE_ROWS,
    localparam ROBOT_CENTER_TO_BOUND_X = (SPRITE_COLS - SCALING_FACTOR) / 2,
    localparam ROBOT_CENTER_TO_BOUND_Y = (SPRITE_ROWS - SCALING_FACTOR) / 2
);
```

Figure 4 - Robot_icon_v2.v

The highlighted parameters of importance are SPRITE_COLS, SPRITE_ROWS, MEM_COLS, and FRAME_ROW_SIZE. The sprite rows and columns indicate the icon is a 34x34 image. There is however, a frame row size which is later shown as being called on for each orientation N, NE, E, SE, S, SW, W, NW. The frame row size is listed as mem_cols * sprite_rows which is $102 * 34 = 3468$. This means that each orientation has a frame that has a depth of 3468 and 12 bit width. Since we know there is an animation component and that $102/34 = 3$ we determined that each frame had 3 images that were cycled through based on the orientation so long as the bot is in motion. So how does the .mem file depth of 27,744 previously mentioned come into play? We know there are eight orientations used and each one has a depth of 3468, so a full icon file with animations in each direction would have a depth of $3468 * 8 = 27,744$.

The task from this point is to create a Crash Bandicoot version of the pikachu.mem file. I was able to find a great sprite sheet (Figure 5) at https://www.sprisers-resource.com/game_boy_advance/crashband/ to use for the icon as well as map modifications that are discussed later.

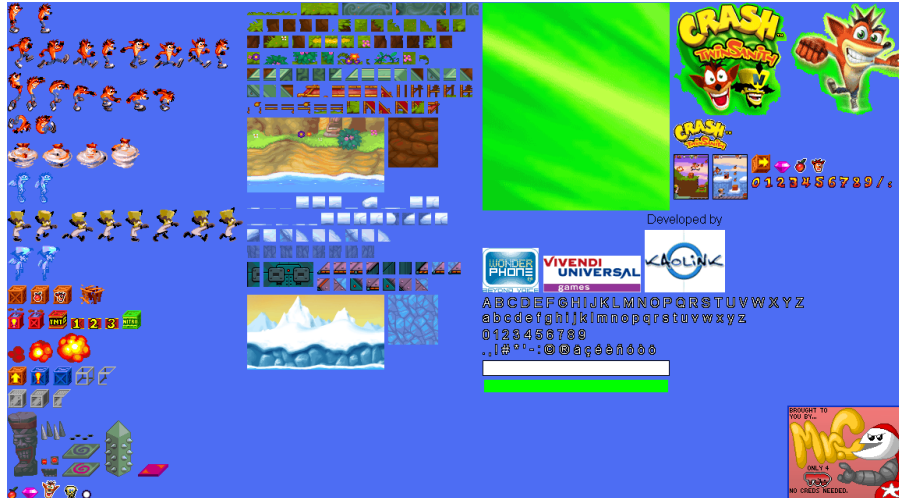


Figure 5 - Crash Bandicoot Sprite Sheet

From here I could easily cut and paste the Bandicoots into MS Paint in a resized 3 Bandicoot wide 102x34 PNG image file for each orientation, or at least the ones that mattered most to our project (E, W, N, S). The next step was to convert PNG to a .mem file for each orientation using Matlab. Initially I had tried adapting a BMP to COE matlab script to output the correct conversion but this ended up not giving the correct data format which made the icon look like a random square block of color after implementation in Vivado. After consulting the tank group about how they handled the icon creation I was able to use a for loop in a new Matlab script to make sure the conversion stepped through each row and column of the PNG outputting a 12 bit wide row of data represented in hex. This script was run for each individual orientation 102x34 PNG image file. The last step was to stitch together each of the converted .mem files into one .mem file containing each orientation. The pre existing SystemVerilog code for the icon was setup in such a way that the frame sequence was done as S, N, W, E, SW, SE, NW, NE so the final .mem file needed to have the individual sprite orientation files stitched together in this sequence from top to bottom. Once implemented we were able to see the new Crash Bandicoot icon on the VGA display with animation functionality.

2.3 Rojobot IP Block Modifications

The majority of our functional changes to the project happened here in the bot31_if.v interface module. This is where we override the LocX and LocY values to accommodate for different side scroller functionality.

2.3.1 Teleport & LocX/LocY Override

The first changes we made to the BotSim core were to make it read LocX and LocY using the "DataOut" 8-bit port into the picoblaze. This makes it work in the same fashion as the MotCtl input. Doing this allows us to override the X/Y coordinates of the rojobot, which opened up a lot of functionality for us.

The same could be done with the other ports but keeping them from interfering with normal rojobot operation was a bit more complex.

Teleport is what we call the rojobot moving from the left-end of the map LocX == h7B back to its starting location of h00. This is done concurrently with the map muxing for a smooth transition from map to map. It works in both directions and at any LocY to allow for jumping/falling through to the next map.

```
// Teleport for map muxing:
else if (LocX_int == 8'h7D) begin // Sidescroller teleport to beginning.
    LocX_int_set <= 8'h01;
end
else if (LocX_int == 8'h00) begin // Sidescroller teleport to ending.
    LocX_int_set <= 8'h7B;
end

// Movement Control:
else if (MotCtl[4] == 0 && MotCtl[0] == 0 && MapVal == 1) begin // Disable "down" movement if on line.
    MotCtl_set <= 8'h00;
end
else if (MotCtl[4] == 1 && MotCtl[0] == 1 && MapVal != 1) begin // Disable "up" movement if not on line.
    MotCtl_set <= 8'h00;
end
```

Figure 6 - Teleport/Movement Restrictions

Other required modifications for this area were the movement controls. We disabled jump/crouch movements when not on the line to prevent the robot from being able to fly off screen. Finally there were changes to the syncing blocks, which was similar to the rojobot controller syncing method. We added an additional set of syncing registers so that overrides would not get ignored if the rojobot was mid-process and outputting different coordinates.

2.3.2 Jump & Fall:

In order to have the bot jump a conditional statement checks if the BotInfo_int register is 8'h40, which indicates the bot is moving forward in the Northern direction, as well as check if a falling flag, a register in bot31_if.v, is clear. If these conditions are met, the bot will shift upwards by four pixels, maintain its position in the X direction, set the falling flag, and start decrementing the falling timer. The falling timer counts down from 800_000_000 to zero. Once the falling timer reaches zero and the falling count is less than five the bot is shifted down by one pixel, a new timer is started, and the falling count register is incremented by one. This is done in order to provide a somewhat natural visual of the bot falling back down from the jump to the origin of its jump action.


```

//checks if falling count has reached 5 falling cycles
else if (falling_count == 5) begin
    falling_flag <= 0;
    falling_timer <= timer_val;
    falling_count <= 0;
    LocY_int_set <= LocY_int + 1; //shifts bot in -Y direciton by one - should set it back to origin point of jump
    LocX_int_set <= LocX_int; //maintains X coordinates
end
//while the falling flag is set, the falling timer decrements by 1
else if (falling_flag == 1'b1) begin
    falling_timer <= falling_timer - 1;
end
//checks if bot is oriented north and if falling flag is 0
else if (BotInfo_int == 8'h40 && falling_flag == 1'b0) begin
    LocY_int_set <= LocY_int - 4; //jumps up 4 pixels
    LocX_int_set <= LocX_int; //mantains position in X coordinates
    falling_timer <= timer_val - 1; //starts falling timer
    falling_flag <= 1; //sets falling flag
end
//checks if timer has counted down and if the total falling count is less than 5
else if (falling_timer == 0 && falling_count < 5) begin
    LocY_int_set <= LocY_int + 1; //shifts bot in -Y directions by 1
    LocX_int_set <= LocX_int; //maintains position in X direction
    falling_timer <= timer_val - 1; //restarts falling timer
    falling_count <= falling_count + 1; //adds a count to total falling count reg
end
//check if not on a black line
else if (MapVal != 2'b01) begin
    LocY_int_set <= LocY_int + 1; //shifts bot in -Y direction by one
    LocX_int_set <= LocX_int; //maintains X coordinates
end

```

Figure 7 - Jump & Fall SystemVerilog

The final portion of the code shown above checks to see if the bot isn't on a black line with MapVal != 2'b01. If this condition is met the bot will shift down by one pixel. This downshift will continue to occur until the user moves the bot enough in the east or west direction to find a black line or otherwise fall to its death.

2.4 BotSim31 (Picoblaze Core) Modifications

Some of the more interesting changes we made were to the BotSim31 picoblaze firmware itself. This code was written by Professor Kravitz and is the heart of the Rojobot. At the start of the project we were strongly advised to work around it, so of course we had to dig in and see what we could modify to make things easier at the higher levels. There are instructions in [section 7.0](#) for how we went about rebuilding the core after modifying it.

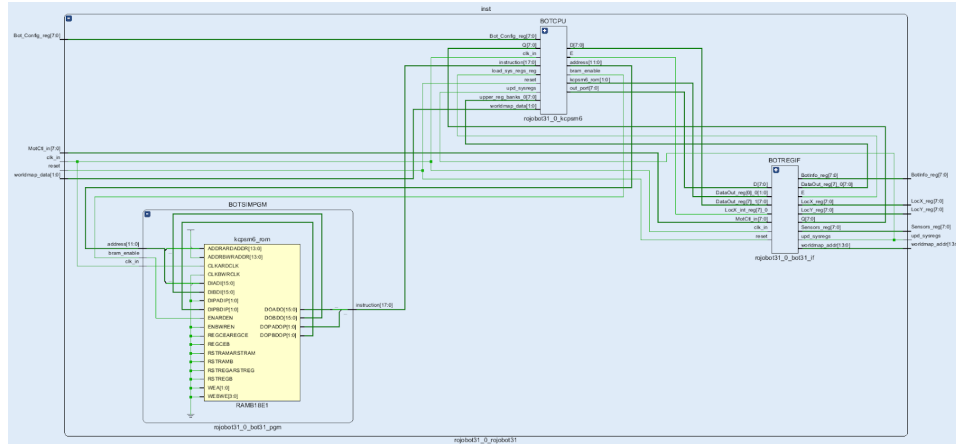


Figure 8 - BotSim31 Core location with Rojobot IP

The first major change we made was the starting LocX to zero instead of 40. This made map muxing and teleporting the rojobot back to the beginning of a map easier. We also made LocX and LocY dependent on their associated port similar to MotCtrl. Originally these variables were stored internally and we had no way to access and override them at the rojobot controller level.

```

708 ; =====
709 ; == Added for Sidescroller
710 ; =====
711
712 gnxy_SLT:    COMPARE s4,    MV_SLT    ; else if (Mvmnt == SLT) { Go Forward }
713             JUMP    Z,      gnxy_11    ;
714             COMPARE s4,    MV_FLT    ; else if (Mvmnt == FLT) { Go Forward }
715             JUMP    Z,      gnxy_11    ;
716             COMPARE s4,    MV_SRT    ; else if (Mvmnt == SRT) { Go Forward }
717             JUMP    Z,      gnxy_11    ;
718             COMPARE s4,    MV_SLT    ; else if (Mvmnt == SLT) { Go Forward }
719             JUMP    NZ,     gnxy_00    ; if none of them jump to reverse
720
721 gnxy_11:     COMPARE s0,    0F        ; if (xcoord == 0x0F)
722             JUMP    NZ,     gnxy_08    ; -- Z != says xcoord != 0F
723             SUB    s6,      01        ; newx = LocX - 1
724             JUMP    gnxy_09          ;
725 gnxy_08:     ADD    s6,      s0        ; newx = LocX + incr (either 0 or 1)
726 gnxy_09:     COMPARE s1,    0F        ; if (ycoord == 0x0F)
727             JUMP    NZ,     gnxy_10    ; newy = LocY - 1
728             SUB    s7,      01        ;
729             JUMP    gnxy_chk0B        ;
730 gnxy_10:     ADD    s7,      s1        ; newy = LocY + incr (either 0 or 1)
731             JUMP    gnxy_chk0B        ; } // end Mvmnt == Fwd
732
733 ; =====
734
735 gnxy_00:     COMPARE s4,    MV_REV    ; else if (Mvmnt == REV) {

```

Figure 9 - Bot31_pgm.psm - get_newxy changes

The next major obstacle was to adjust the controls so that the rojobot acted more like a side scroller character. This was accomplished by modifying two sections. The above is the get_newxy function which handles whether the motion control inputs lead to the robot moving forward/reverse/rotating/stop. We modified all rotating signals to also move forward.

This was combined with modifying the calc_orient function. Instead of the orientation rotating with the robot the robot automatically sets to N/S/W/E and then executes a movement. The result is that if you

press the left button, it re-oriens to the west and moves forward one. Right button does the opposite. Forward oriens north and reverse also oriens north but moves south. This gives us the control map we want using push buttons, which will also be mapped with the accelerometer in assembly.

3.0 Sidescroller Software

Most of our design specifications we decided to implement in hardware, not only because we thought it might be easier but also because the class is about FPGAs and we really wanted to dig deeper into the system verilog and picoblaze core. The major part that we did work on 100% in assembly was the use of the accelerometer.

3.1 Sidescroller Software Theory of Operation

The accelerometer of the Nexys A7 provides an output of 12 bit resolution . Measurement ranges of ± 2 g, ± 4 g, and ± 8 g are available with a resolution of 1 mg/LSB on the ± 2 g range. The FPGA can talk with the ADXL362 via SPI interface. While the ADXL362 is in Measurement Mode, it continuously measures and stores acceleration data in the X-data, Y-data, and Z-data registers.

The accelerometer SPI interface works by pulling a CS line low which enables reading of the SPI port. We then send it 3 sets of control messages to tell it what data we want to receive. We do this three times, one for each axis, and concatenate the final results into the 12-bit xyz result.

Once we have our SPI reading it's time to map it to the robot motion controls. We originally tried to map to orientation + buttons but that didn't work out well with the rest of the rojobot modifications. Instead we map the SPI reading to our modified controls (Left, Right, Jump, Down.)

The major issue we have with the SPI readings for controls is that its sensitivity is difficult to adjust and maintain.

We have used the X and Y axis to control the bot. The accelr function returns the accelerometer values in the a1 register. The bits [3:0] are the X axis values and the bits [7:4] contain the values for Y axis. The MSB is used to determine the orientation of the Bot. If the X axis is negative then the bot orientation is set to North and South if positive. Similarly for the Y axis , the bot orientation is set West if the Y axis is positive and East if X axis is negative.

```

Xaxis: blt t3,t5,OnlyYaxis           # if X axis value is less than the threshold

XoriP: and t1,t2,0x01                # get X sign
      beq t1,t6,XoriN
      add s1,zero,zero                # initialize s1 to zero
      addi s1,s1,0x06                 # set S
      j onset

XoriN: add s1,zero,zero                # initialize s1 to zero
      addi s1,s1,0x02                 # set N
      j onset

OnlyYaxis: blt t4,t5, Nochange

YoriP: and t1,t2,0x02
      srl t1,t1,1
      beq t1,t6,YoriN
      add s1,zero,zero                # initialize s1 to zero
      addi s1,s1,0x04                 # set W
      j onset

YoriN: add s1,zero,zero                # initialize s1 to zero
      addi s1,s1,0x00                 # set E
      j onset

Nochange: add a0,zero,zero
          jr ra

onset: beq s1,t0,movforward
       add a0,zero,zero

```

The threshold is set to 3 . If the value of the X axis is greater than the threshold then the bot moves forward in the X direction. If the value of the Y axis is greater than the threshold then the bot moves forward in the Y direction. The accelerometer values are mapped to the pushbutton values in the acc2mot function .

4.0 Results & Challenges

While there were many successes during project development there were also goals that were unable to be met as well as challenges that seemed insurmountable at first but were able to be overcome. This is the good, the bad, and the ugly of the sidescroller project.

4.1 The Good (Successes)

What worked well for us

- Teleport - It took awhile to figure how to rebuild the BotSim core but once we did this part of the program was surprisingly easy to implement. It also led to quite a few other ideas such as falling/jumping that were also implemented in the same file.
- Muxing - able to successfully switch between maps when moving in the forward and reverse direction
- Icon - Crash Bandicoot icon looks good. Would have liked to increase the icon width for each orientation in order to improve the gait so it had a more natural feel. Three images is not enough to achieve such an animation.
- Remapping Movement - After messing with MotCtl lookup tables didn't work we switched to the forced orientation and overrode the forward/reverse/rotate-left/rotate-right controls instead. This ended up working quite well after trying a few different mapping schemes.

4.2 The Bad (Failures)

What paths completely failed for us.

- Bigger map size - unable to modify the map mux and always block logic to incorporate forward and reverse travel between more than 3 maps however there are 5 maps total that were designed.
- Modifying MotCtl inside of the BotSim turned mostly into a disaster due to how pretty much the entire robot was built on the variable speed left/right motor setup.
- Modifying BotSim lookup tables was the second attempt when MotCtl changes didn't work. All of the lookup tables build off of each other so it was quickly turning into a lot of non-working spaghetti code.
- Making obstructions impassible (the robot always finds a way) - would have been more realistic to implement code that made it so the bot was unable to walk through MapVal = 2 (the brown bricks).
- True side scrolling - After the first week in the project we determined we would have more success having the game function more like super mario where the bot moves as opposed to the

map moving like a traditional side scroller. If we spent more time determining how to increment the map values across the screen we might have been able to get the side scroller function working but we decided to cut our losses early.

- Wumpa Fruit - wanted to include the classic Crash Bandicoot fruit as a power up option that would increase the bot's speed for a short duration but wasn't able to meet this stretch goal in time.

4.3 The Ugly (Challenges)

What did we suffer through but manage to get working.

- Falling/Jump - A few things that could've made jump more realistic:
 1. Fixing the bug that was causing the jump action to not return to origin
 2. Creating a governor that prevents double jumping / infinite jumps
- The accelerometer was very sensitive and hence it was very hard to control the bot using the accelerometer . The values were not stable and it returned values even when there was no movement. The Y axis values flipped between 0 and F causing unnecessary movement in the Y direction.

5.0 Team Contributions

The following subsections include descriptions of each team member's contributions to the project.

5.1 Ryan Fallis

- Map design - BotWorld31_dev_kit & generating hex values for 0,1,2 map pixels.
- Map muxing - setting up always block for forward and reverse map switching.
- Icon design - utilized matlab and ms paint to create icon animations.
- Title design - utilized matlab and ms paint to create custom title.
- Jumping/Falling - modified bot31_if.v file to incorporate jumping and falling.

5.2 Richard Romano

- Map muxing (module) - Built the module itself and the basic muxing by switch.
- BotSim31 Modifications - Focused on rebuilding the BotSim core.
 - Teleport - Spent a lot of time figuring out how to override LocX/LocY
 - Control Remapping - Further BotSim31 modifications to force orientation/control.
- On-Black-Line movement - Attempts at forcing the robot to stay on line.
- Assembly troubleshooting - Helped get some of the accelerometer code running.

5.3 Keerthan Nayak

- Accelerometer
- Mapping the accelerometer values to movement of the rojobot.
- Assembly troubleshooting

6.0 Conclusion

We started the Side Scroller project with quite a few different ideas and possible solutions for implementing them. Many of them turned out to be harder than we had hoped for. The project is still quite far from what anyone would consider a “playable game.” Despite that, we did have a bit of fun figuring out how to implement different specifications like the map muxing and new control scheme. We also spent a considerable amount of effort towards learning about the picoblaze soft core and how the inner workings of the rojobot functioned. Probably the most important part has been learning about the FPGA workflow and how to dive into each part of the chip to modify and troubleshoot functions. Overall we all feel that we learned quite a bit this term, especially from the start when none of us had any experience with FPGAs and limited experience with assembly language.

7.0 BotSim 3.1 - IP Editing Guide

The last few pages are a guide to editing the rojobot IP block and its picoblaze firmware. Editing the IP Block was a key workflow that the team went through many times to get everything to work. So this guide was slowly made to help us remember all the important steps as we kept digging further into the IP.

BOTSIM 3.1 - IP Editing Guide

Created By: Richard Romano

08 December 2020

1.0 Overview

The Rojobot 3.1 IP Block is created using the PicoBlaze soft core KCPSM6 provided by Xilinx. The IP block is broken up into two parts. The top level parts can be edited in the Vivado IP Packager. The rojobot firmware has to be edited externally and recompiled before it can be loaded into the IP Packager.

The top level IP interface:

Rojobot31_0.v - IP Interface File

Bot31_top.v - Top level Rojobot module

Bot31_if.v - 8-Bit interface module between KCPSM6 and external ports

The Picoblaze Core:

Bot31_pgm.psm - The unassembled Rojobot firmware

KCPSM6.v - The picoblaze soft core

Rom form.v - The provided ROM template used for assembly

Bot31_pgm.v - The assembled Rojobot firmware

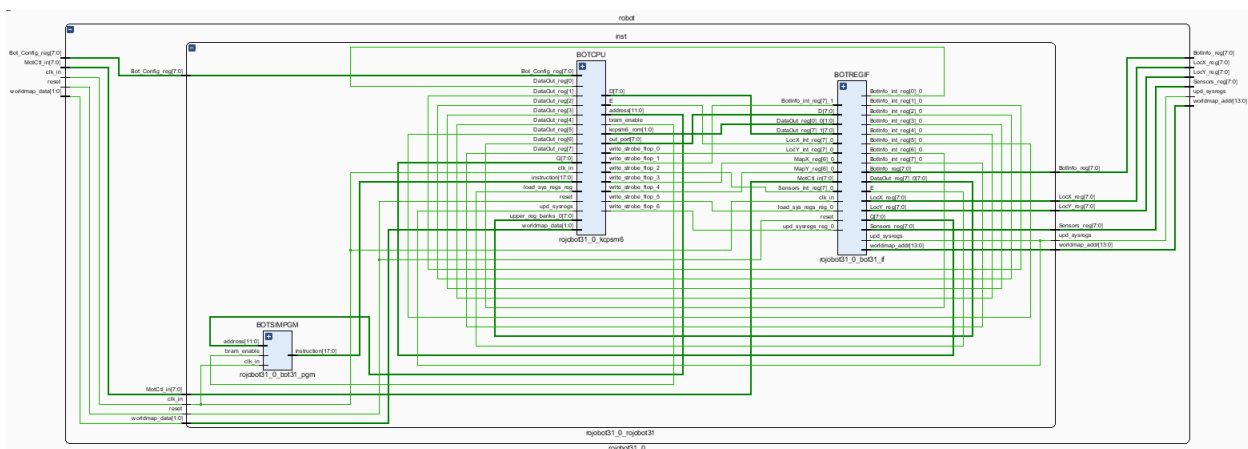
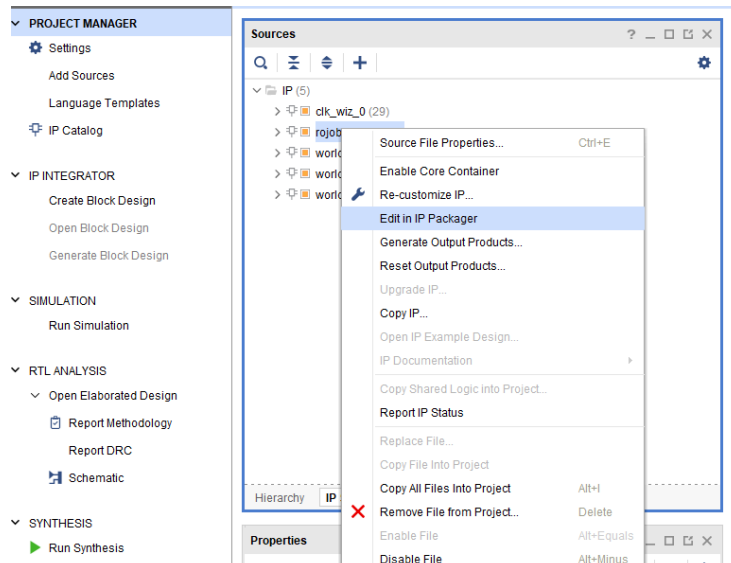


Figure 0 - Rojobot RTL Schematic

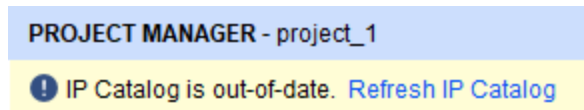
2.0 Vivado IP Packager

The first step is to open the IP source in the vivado IP Packager. This will create a new project and unlock the design files for editing. The three top level files can be edited at this point. Some useful changes include adding additional ports using bot31_if.v, intercepting rojobot outputs, and adjusting the synchronization between the rojobot and the external project.



Once you have finished editing the IP sources you can follow the below steps to rebuild the IP block:

1. Save all of the edited files.
2. Close the IP Packager project.
3. Refresh the IP Catalog



4. At the bottom there should now be an IP Status report
5. Click the “upgrade selected” button at the bottom of the report.

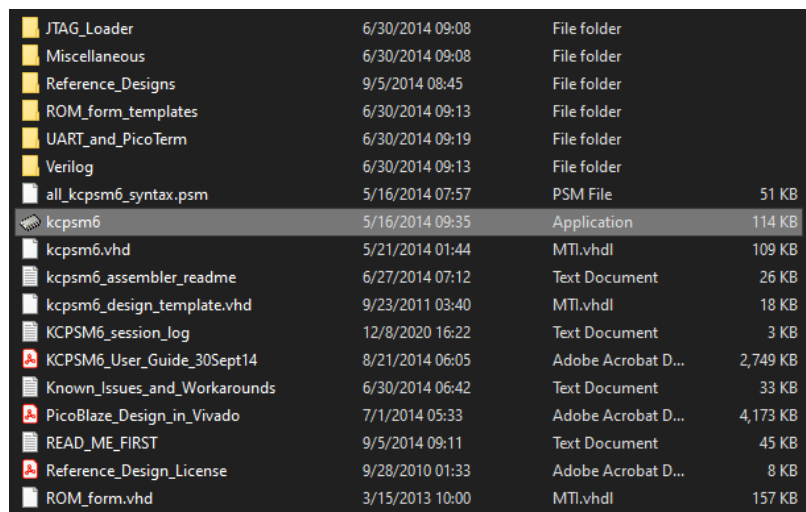
IP Status									
Source File	IP Status	Recommendation	Change Log	IP Name	Current Version	Recommended Version	License	Current Part	
rojobot31_0	IP definition rojobot31_v1_0 (1.0) changed on disk. Upgrade IP	No changes required	More info	rojobot31_v1_0	1.0 (Rev. 2)	1.0 (Rev. 2)	Included	xc7a100tcs324-1	
clk_wiz_0	Up-to-date	No changes required	More info	Clocking Wizard	6.0 (Rev. 4)	6.0 (Rev. 4)	Included	xc7a100tcs324-1	
world_map	Up-to-date	No changes required	More info	Block Memory Generator	8.4 (Rev. 4)	8.4 (Rev. 4)	Included	xc7a100tcs324-1	
world_map_loop	Up-to-date	No changes required	More info	Block Memory Generator	8.4 (Rev. 4)	8.4 (Rev. 4)	Included	xc7a100tcs324-1	
world_map_if	Up-to-date	No changes required	More info	Block Memory Generator	8.4 (Rev. 4)	8.4 (Rev. 4)	Included	xc7a100tcs324-1	
									Upgrade Selected

6. Click the rerun link and make sure all of your IP shows up-to-date in the report.
7. You can now run synthesis. It should re-synthesize the IP source first.

3.0 Assembling the Rojobot Firmware

The rojobot firmware uses the KCPSM6 assembler. Information about it and the design files can be found on Xilinx's website. You should get a zip file containing the KCPSM6 application and various guides and tools to help you with the process.

<https://www.xilinx.com/products/intellectual-property/picoblaze.html>



JTAG_Loader	6/30/2014 09:08	File folder	
Miscellaneous	6/30/2014 09:08	File folder	
Reference_Designs	9/5/2014 08:45	File folder	
ROM_form_templates	6/30/2014 09:13	File folder	
UART_and_PicoTerm	6/30/2014 09:19	File folder	
Verilog	6/30/2014 09:13	File folder	
all_kcpsm6_syntax.psm	5/16/2014 07:57	PSM File	51 KB
kcpsm6	5/16/2014 09:35	Application	114 KB
kcpsm6.vhd	5/21/2014 01:44	MTL.vhdl	109 KB
kcpsm6_assembler_readme	6/27/2014 07:12	Text Document	26 KB
kcpsm6_design_template.vhd	9/23/2011 03:40	MTL.vhdl	18 KB
KCPSM6_session_log	12/8/2020 16:22	Text Document	3 KB
KCPSM6_User_Guide_30Sept14	8/21/2014 06:05	Adobe Acrobat D...	2,749 KB
Known_Issues_and_Workarounds	6/30/2014 06:42	Text Document	33 KB
PicoBlaze_Design_in_Vivado	7/1/2014 05:33	Adobe Acrobat D...	4,173 KB
READ_ME_FIRST	9/5/2014 09:11	Text Document	45 KB
Reference_Design_License	9/28/2010 01:33	Adobe Acrobat D...	8 KB
ROM_form.vhd	3/15/2013 10:00	MTL.vhdl	157 KB

Figure 2 - KCPSM6 Design Files

You will also need to download the BotSim31 zip file containing the Bot31_pgm.psm firmware and ROM_form.v files. You can edit the psm file using your preferred text editor.

Once you have finished editing the firmware you can follow the below steps to reassemble the file:

1. Make sure that both the psm and ROM form file are located in the same folder.
2. Start kcpsm6 and enter the file name, for example:
RojobotPSM\bot31_pgm.psm
3. We don't recommend changing the psm filename unless you also want to deal with filename changes in vivado.
4. The assembler does some basic assembly code checks but should not be heavily relied upon.
5. The assembler will output three files, you will only need the bot31_pgm.v file.
6. Follow the first paragraph in section 2 for opening the IP packager.
7. Locate the bot31_pgm.v file in vivado. You can look at its properties to see its file location.

8. Replace the bot31_pgm.v with your newly assembled version either by using file explorer or if you prefer the “replace file” option in vivado. Note that the replace file points to the new source location.

```
KCPSM6 Assembler v2.70
Ken Chapman - Xilinx Ltd - 16th May 2014

Enter name of PSM file: Rojobot PSM\bot31_pgm.psm

Reading top level PSM file...
C:\Users\rromano\Documents\GitHub\Nexys-A7-Sidescroller-Game\PGM Test\KCPSM6_Release9_30Sept14\Rojobot PSM\bot31_pgm.p

A total of 1028 lines of PSM code have been read

Checking line labels
Checking CONSTANT directives
Checking STRING directives
Checking TABLE directives
Checking instructions

Writing formatted PSM file...
C:\Users\rromano\Documents\GitHub\Nexys-A7-Sidescroller-Game\PGM Test\KCPSM6_Release9_30Sept14\Rojobot PSM\bot31_pgm.f

Expanding text strings
Expanding tables
Resolving addresses and Assembling Instructions
Last occupied address: 316 hex
Nominal program memory size: 1K (1024)    address(9:0)
Occupied memory locations: 540
Assembly completed successfully

Writing LOG file...
C:\Users\rromano\Documents\GitHub\Nexys-A7-Sidescroller-Game\PGM Test\KCPSM6_Release9_30Sept14\Rojobot PSM\bot31_pgm.l

Writing HEX file...
C:\Users\rromano\Documents\GitHub\Nexys-A7-Sidescroller-Game\PGM Test\KCPSM6_Release9_30Sept14\Rojobot PSM\bot31_pgm.h

Writing Verilog file...
C:\Users\rromano\Documents\GitHub\Nexys-A7-Sidescroller-Game\PGM Test\KCPSM6_Release9_30Sept14\Rojobot PSM\bot31_pgm.v

KCPSM6 Options.....
R - Repeat assembly with 'Rojobot PSM\bot31_pgm.psm'
N - Assemble new file.
Q - Quit
-
```

Figure 3 - Example RCPSM6 Assembler Output

9. Follow steps 1-7 of the IP packager section to rebuild the IP block.
10. Congratulations, you now have customized the Rojobot firmware.