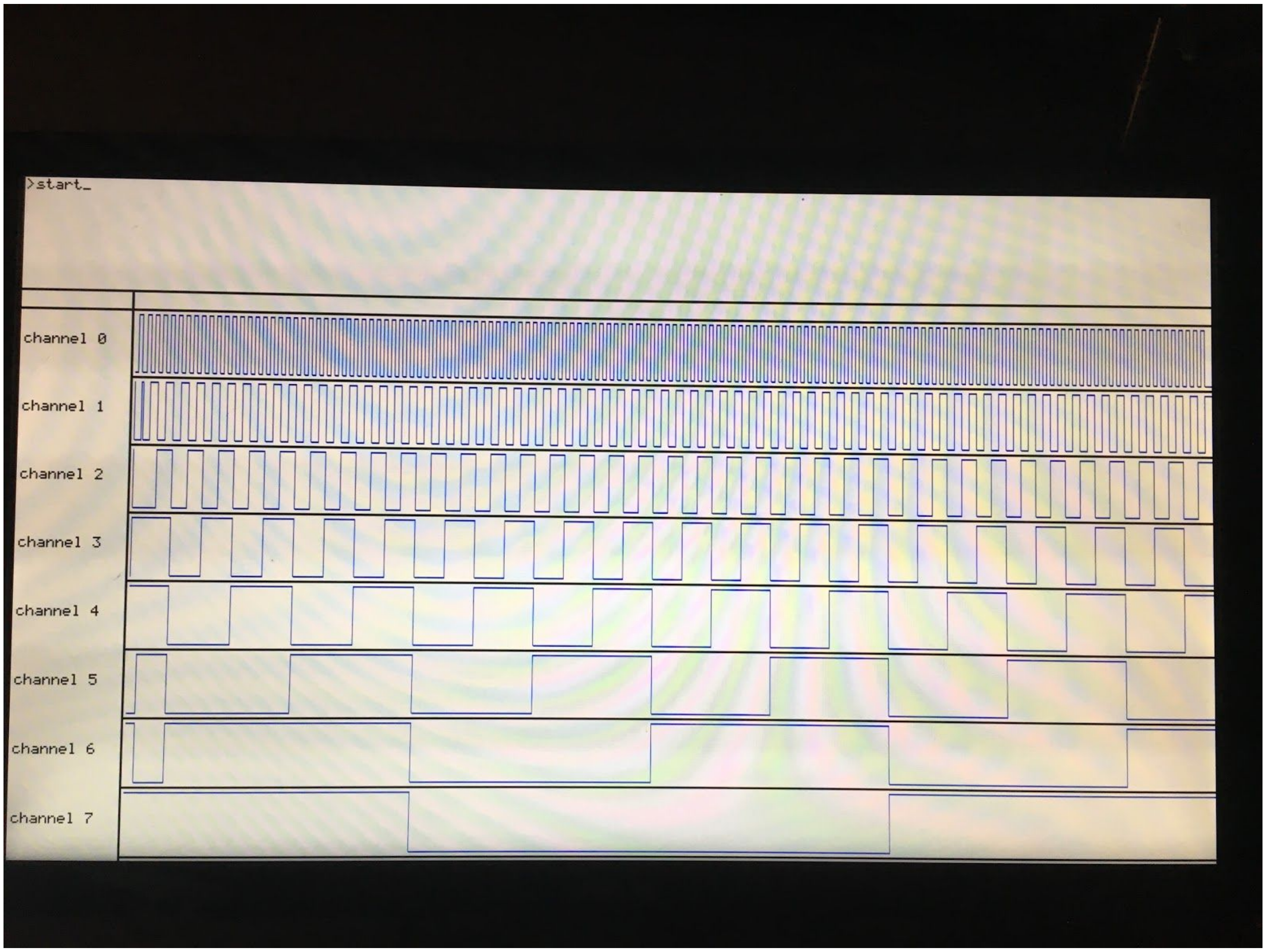
How to Build a Logic Analyzer



By: Cole Havener & Joseph Prachar

In this tutorial we will be going over the steps needed to create a logic analyzer using the zybo z7-10 development board created by Digilant. A logic analyzer reads and displays multiple digital signals, and is often used to debug and test complex digital circuits. Using the zybo z7-10s chip which has both a cortex-A9 processor and xilinx Artix FPGA on one chip enabling us to create custom IP blocks to handle things the processor cannot.

The completed logic analyzer has 8 ports to read from and is able to take user input through a terminal for simple commands such as stop, start and set frequency. This project is a good introduction to RTOS programing as well as developing both hardware and software in parallel.

Our project is on github here: https://github.com/JosephPrachar/logic_analyzer

Necessary components:

- Zybo z7-10 board.
- HDMI cable
- HDMI Monitor display
- Vivado 2017.2
- Xilinx SDK 2017.2
- FreeRTOS

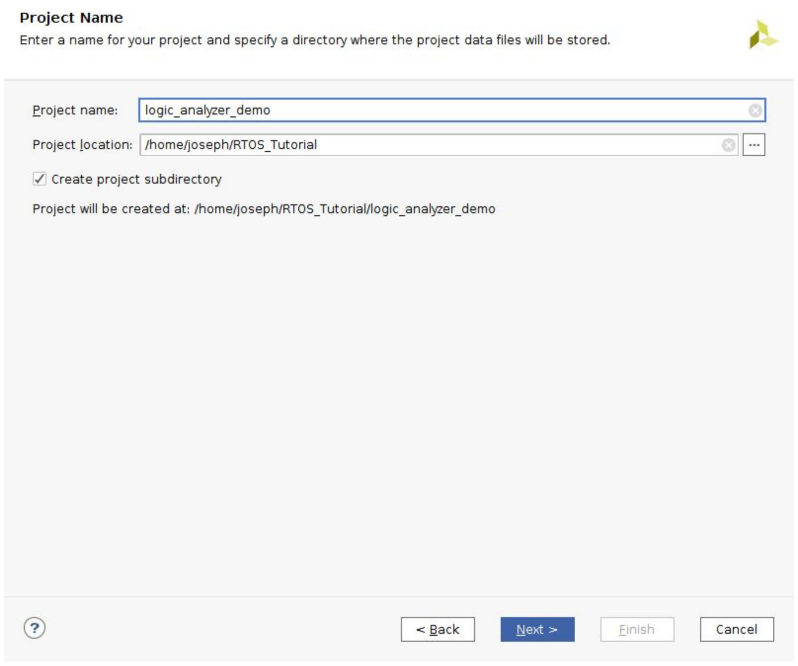
Vivado Tips/Pitfalls

1. If you start to feel crazy and think that no matter what changes you make to the vivado side of you project nothing seems to change; you might not actually crazy. Double check your debug configuration settings <insert step here> to ensure that the sdk is actually programming the zybo board with the most recent bitstream file. (vivado puts its most recent bit file in <project root>/<project name>.runs/impl_1/<name>.bit)
2. If a block design run fails, it never hurts to re-generate the HDL wrapper. You can do this even when you have made changes to the wrapper file by selecting not to overwrite the current file when asked. While this may seem useless, it seems that generating the HDL wrapper also generates some support files that may have disappeared (in our case, due to the imperfect use of source control with the project).
3. If Vivado fails to launch giving you “Microsoft Visual C++Runtime Library” error, and you are using windows 10, fear not, as you are not alone. To fix this go to c/xilinx/Vivado/2017.2/bin/unwrapped/win64.o. Once there rename vivado.exe to vivado.exe.backup, just to be safe. Then copy and paste vivado-vg.exe and rename the copy to vivado.exe. Vivado should now startup normally.
4. If the SDK fails to launch from vivado remember: it's just another program. You can start it manually with an empty workspace then open your project manually once it starts successfully.

5. If you are getting the “xparameters.h: No such file or directory” or other header files missing it likely means that your project cannot see the BSP that you have totally generated and updated from your latest hardware export (right???). The first solution to this problem is to delete any BSP’s that you have generated, you may even have to go into the file explorer to do this, and generate a new one. This however is quite annoying. A less annoying solution is to only regenerate BSP 0 when updating hardware, then go back to approach 1 if unsuccessful. This *usually* works. If neither of these workflow changes works we must dive into the build settings. Go to Project->properties. In this window go to C/C++ Build->Settings. This is where all of the build settings are; the Xilinx SDK (just a thinly veiled version of Eclipse) may seem to build your software with magic, but even it has to call gcc just like us mere mortals. All the flags that it is using are in this menu, so the last ditch effort would be to manually include the BSP sources with this menu. You'll need to compile with the additional include path of <bsp #>/ps7_cortexta9_0/include and link with the actual library at <bsp #>/ps7_cortexa9_0/lib.

Step-by-Step Tutorial

1. Get FreeRTOS running
    a. Vivado
        1. Open vivado 2017.2
        2. Select file->New Project, click next
        3. Enter project name and storage location, click next

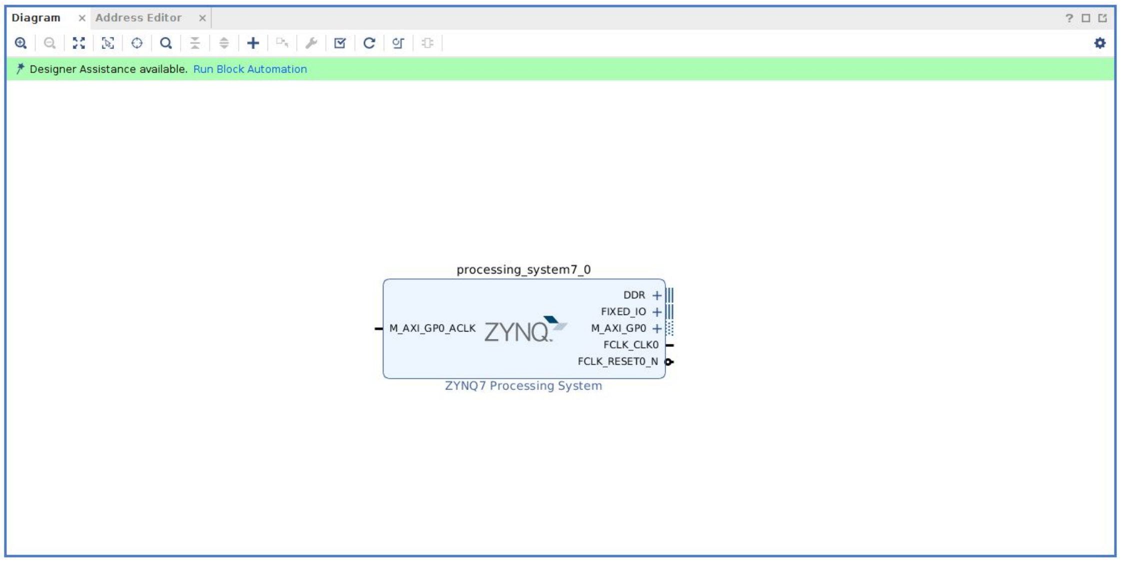


Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name: logic_analyzer_demo
Project location: /home/joseph/RTOS_Tutorial
☑ Create project subdirectory
Project will be created at: /home/joseph/RTOS_Tutorial/logic_analyzer_demo

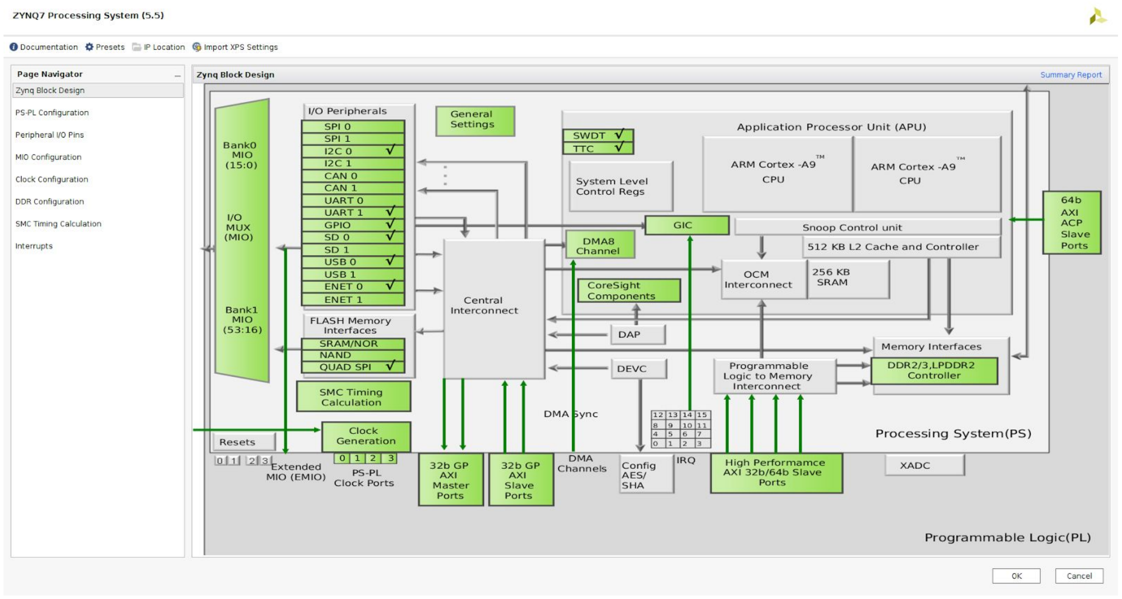
        4. Select RTL Project and check do not specify sources at this time, click next
        5. Open boards tab at top and select Zybo Z7-10
            1. How to add boards to vivado: https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1

6. We now have an empty vivado project!
7. Click create block design from left sidebar
8. Give it a name and leave everything else to defaults (click ok)
9. Right click and select add IP (should be a '+' symbol)
10. Start typing into the textbox to filter all the different ip blocks. We are looking for ZYNQ7 Processing System. Select and add it.

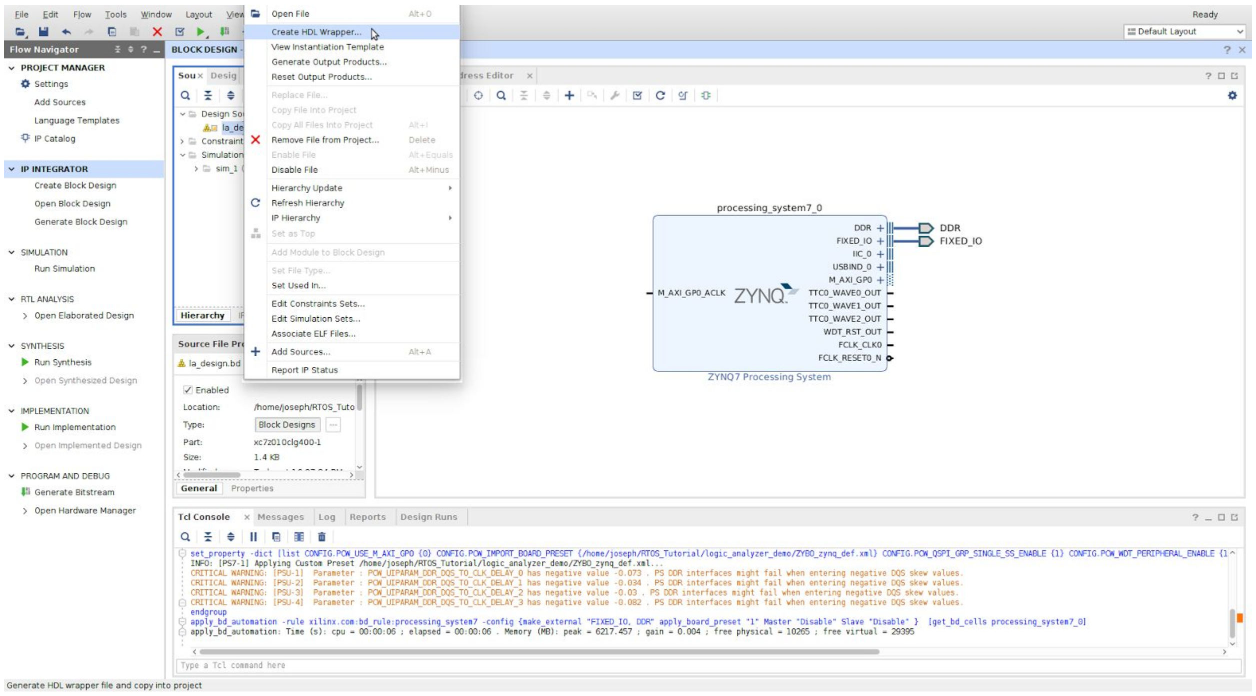


11. Now we have a way to design with the Dual-core ARM A9 Hard processor on the zynq chip!
12. Double click on the IP block you just created (Pro tip: this is the way to open any ip block customization menu that you add in the future)
13. Download this file https://raw.githubusercontent.com/ucb-bar/fpga-zynq/master/zybo/src/xml/ZYBO_zynq_def.xml
14. Select Import XPS Settings in the toolbar at the top of the window and navigate to/select the file from the previous step. (click ok)
15. Now we will remove the AXI Interface. In the PS-PL Configuration tab use the filter to search for M AXI GP0 Interface and deselect it.
16. Like the previous step, in the MIO Configuration tab SELECT the checkboxes GPIO MIO, ENET Reset, Timer 0, and Watchdog (Some of these may already be enabled)

17. Now your zynq block should look like this. (click ok)



18. Click on the run block animation that should be in the green bar at the top of the block design, accept the defaults and select ok.
19. Finally, in the sources pane, right click on the block design and select Create HDL Wrapper



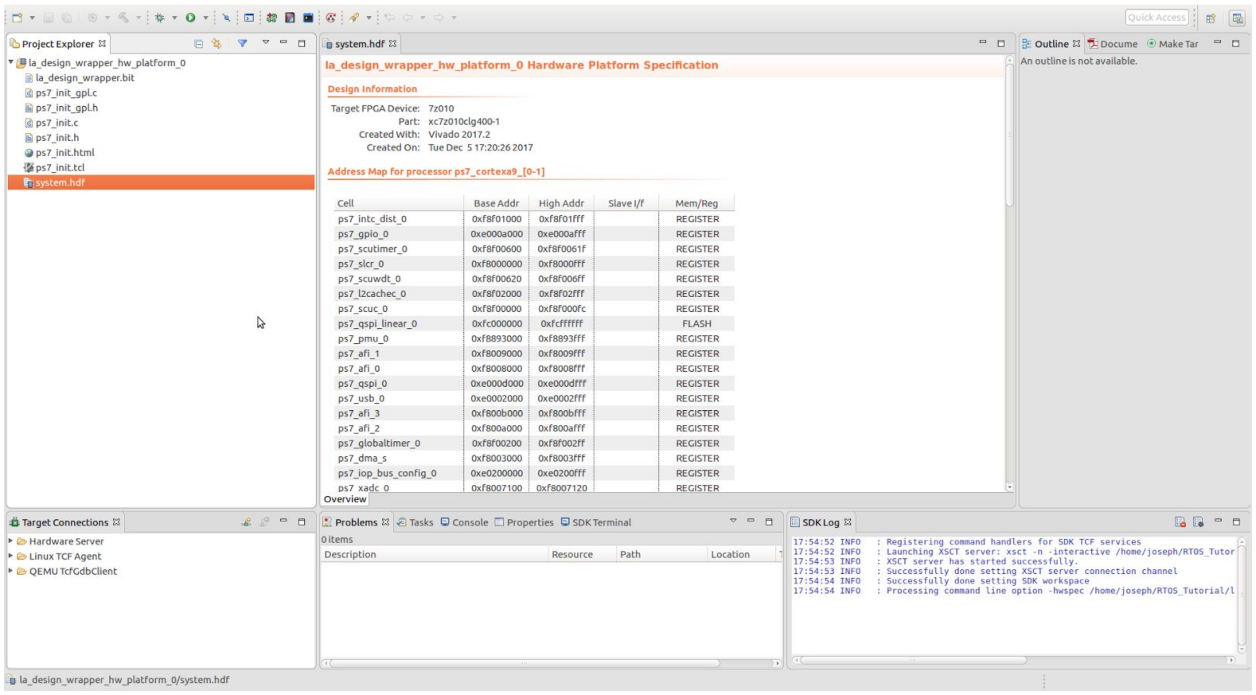
```
set_property -dict [list CONFIG.PCW_USE_M_AXI_GP0 {0} CONFIG.PCW_IMPORT_BOARD_PRESET {/home/joseph/RTOS_Tutorial/logic_analyzer_demo/ZYBO_zynq_def.xml} CONFIG.PCW_QSPI_GRP_SINGLE_SS_ENABLE {1} CONFIG.PCW_WDT_PERIPHERAL_ENABLE {1
INFO: [PS7-1] Applying Custom Preset /home/joseph/RTOS_Tutorial/logic_analyzer_demo/ZYBO_zynq_def.xml...
CRITICAL WARNING: [PSU-1]  Parameter : PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_0 has negative value -0.073 . PS DDR interfaces might fail when entering negative DQS skew values.
CRITICAL WARNING: [PSU-2]  Parameter : PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_1 has negative value -0.034 . PS DDR interfaces might fail when entering negative DQS skew values.
CRITICAL WARNING: [PSU-3]  Parameter : PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_2 has negative value -0.03 . PS DDR interfaces might fail when entering negative DQS skew values.
CRITICAL WARNING: [PSU-4]  Parameter : PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_3 has negative value -0.082 . PS DDR interfaces might fail when entering negative DQS skew values.
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config {make_external "FIXED_IO, DDR" apply_board_preset "1" Master "Disable" Slave "Disable" }  [get_bd_cells processing_system7_0]
apply_bd_automation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 6217.457 ; gain = 0.004 ; free physical = 10265 ; free virtual = 29395
```

20. Select the “Copy generated wrapper to allow user edits” radio button and hit ok
21. Click generate bitstream and wait…
22. Once that is complete, go to File->Export->Export Hardware
23. Check the include bitstream and click ok

24. Now click FIle->Launch SDK and ok

b. SDK

1. The Xilinx SDK should have launched and you should have a window that looks like this:



la_design_wrapper_hw_platform_0 Hardware Platform Specification

Design Information

Target FPGA Device: 7z010
Part: xc7z010clg400-1
Created With: Vivado 2017.2
Created On: Tue Dec 5 17:20:26 2017

Address Map for processor ps7_cortexa9_[0-1]

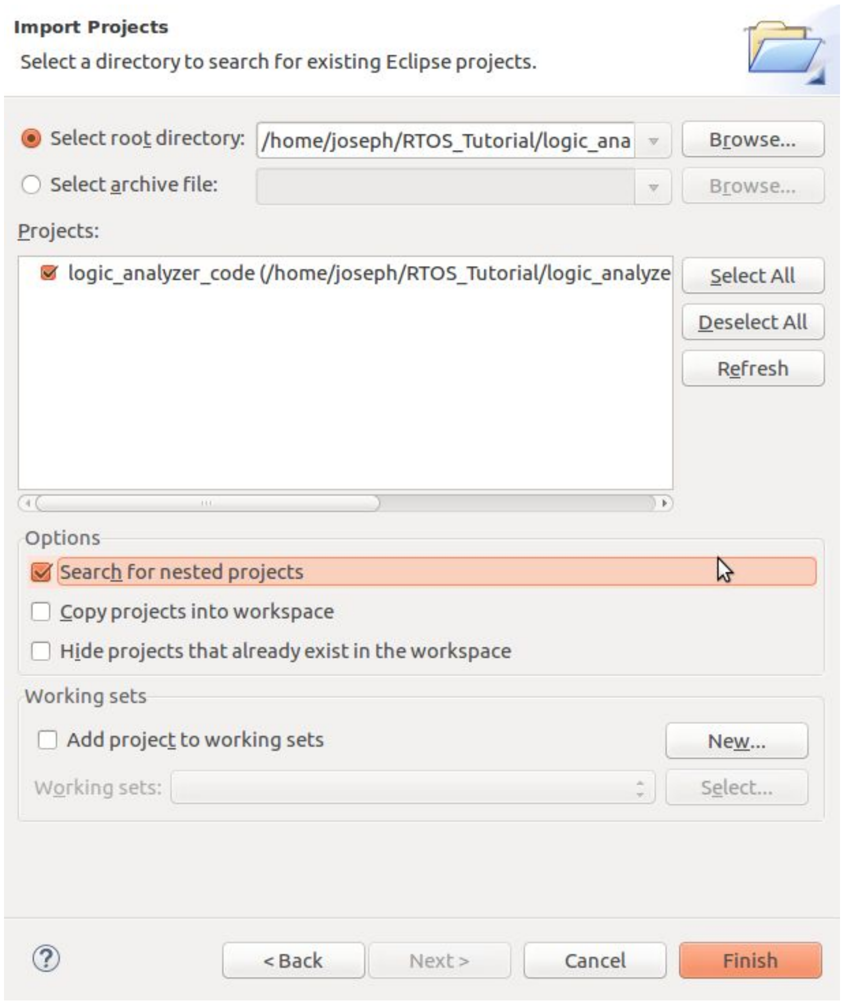
Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
ps7_intc_dist_0	0xf8f01000	0xf8f01fff		REGISTER
ps7_gpio_0	0xe000a000	0xe000afff		REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f0061f		REGISTER
ps7_slcr_0	0xf8000000	0xf8000fff		REGISTER
ps7_scuwdt_0	0xf8f00620	0xf8f006ff		REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_scuc_0	0xf8f00000	0xf8f000fc		REGISTER
ps7_qspi_linear_0	0xfc000000	0xfcffffff		FLASH
ps7_pmu_0	0xf8893000	0xf8893fff		REGISTER
ps7_afi_1	0xf8009000	0xf8009fff		REGISTER
ps7_afi_0	0xf8008000	0xf8008fff		REGISTER
ps7_qspi_0	0xe000d000	0xe000dfff		REGISTER
ps7_usb_0	0xe0002000	0xe0002fff		REGISTER
ps7_afi_3	0xf800b000	0xf800bfff		REGISTER
ps7_afi_2	0xf800a000	0xf800afff		REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff		REGISTER
ps7_dma_s	0xf8003000	0xf8003fff		REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff		REGISTER
ps7_xadc_0	0xf8007100	0xf8007120		REGISTER

Overview

```
17:54:52 INFO   : Registering command handlers for SDK TCF services
17:54:52 INFO   : Launching XSCT server: xsct -n -interactive /home/joseph/RTOS_Tutor
17:54:53 INFO   : XSCT server has started successfully.
17:54:53 INFO   : Successfully done setting XSCT server connection channel
17:54:54 INFO   : Successfully done setting SDK workspace
17:54:54 INFO   : Processing command line option -hwspec /home/joseph/RTOS_Tutorial/l
```

2. Download the basic_freertos.zip project and extract to the <project root>/<project name>.sdk/ directory
3. Go to File->Import and select Existing Projects into Workspace option

4. Use the Select root Directory text box to find the project you just extracted and select finish.



Import Projects

Select a directory to search for existing Eclipse projects.

[● Select root directory:] /home/joseph/RTOS_Tutorial/logic_ana

Select archive file:

Projects:

[✓] logic_analyzer_code (/home/joseph/RTOS_Tutorial/logic_analyze

Options

[✓] Search for nested projects

[ ] Copy projects into workspace

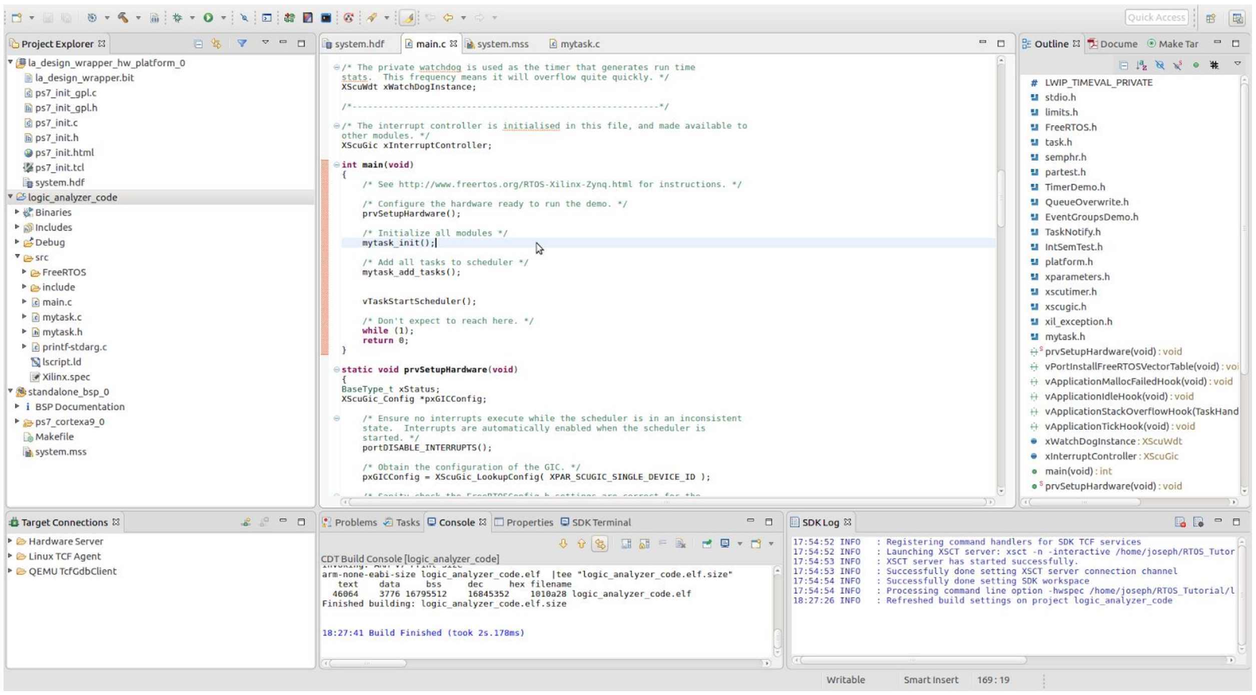
[ ] Hide projects that already exist in the workspace

Working sets

[ ] Add project to working sets

5. The project should now be visible in the project Explorer
6. Oh no, the project has some compilation errors! Let's fix those. (Pro tip: Remember this error! If you ever get something like the “xparameters.h: No such file or directory” it likely means that your project cannot see the BSP that we are about to generate <view vivado tips/pitfalls section #5>)
7. To Generate this BSP go to File->New->Board Support Package, accept the defaults and click finish, then Ok when prompted with the Board Support Package Settings window

8. The project should build now! The last step now is to tell the SDK how/what to upload to your Zybo



```c
/* The private watchdog is used as the timer that generates run time
stats.  This frequency means it will overflow quite quickly. */
XScuWdt xWatchDogInstance;

/*-----------------------------------------------------------*/

/* The interrupt controller is initialised in this file, and made available to
other modules. */
XScuGic xInterruptController;

int main(void)
{
    /* See http://www.freertos.org/RTOS-Xilinx-Zynq.html for instructions. */

    /* Configure the hardware ready to run the demo. */
    prvSetupHardware();

    /* Initialize all modules */
    mytask_init();

    /* Add all tasks to scheduler */
    mytask_add_tasks();


    vTaskStartScheduler();

    /* Don't expect to reach here. */
    while (1);
    return 0;
}

static void prvSetupHardware(void)
{
BaseType_t xStatus;
XScuGic_Config *pxGICConfig;

    /* Ensure no interrupts execute while the scheduler is in an inconsistent
    state.  Interrupts are automatically enabled when the scheduler is
    started. */
    portDISABLE_INTERRUPTS();

    /* Obtain the configuration of the GIC. */
    pxGICConfig = XScuGic_LookupConfig( XPAR_SCUGIC_SINGLE_DEVICE_ID );
```

CDT Build Console [logic_analyzer_code]

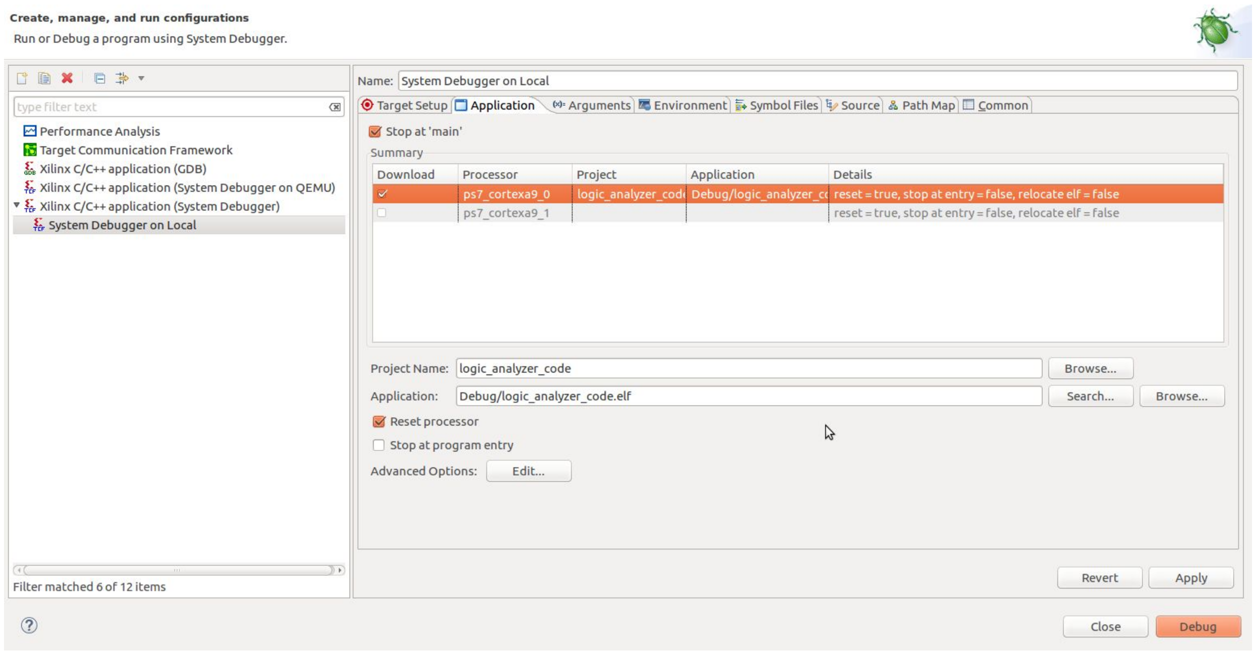
```
arm-none-eabi-size logic_analyzer_code.elf  |tee "logic_analyzer_code.elf.size"
   text    data     bss     dec     hex filename
  46064    3776 16795512    16845352    1010a28 logic_analyzer_code.elf
Finished building: logic_analyzer_code.elf.size


18:27:41 Build Finished (took 2s.178ms)
```

```
17:54:52 INFO   : Registering command handlers for SDK TCF services
17:54:52 INFO   : Launching XSCT server: xsct -n -interactive /home/joseph/RTOS_Tutor
17:54:53 INFO   : XSCT server has started successfully.
17:54:53 INFO   : Successfully done setting XSCT server connection channel
17:54:54 INFO   : Successfully done setting SDK workspace
17:54:54 INFO   : Processing command line option -hwspec /home/joseph/RTOS_Tutorial/l
18:27:26 INFO   : Refreshed build settings on project logic_analyzer_code
```

9. Go to Run->Debug Configurations…
10. Right click on the option in the list Xilinx C/C++ application (System Debugger) and select new
11. Change the Debug Type to Standalone Application
12. Select both the Reset entire system and Run ps7_post_config options. (Note the bitstream file textbox and vivado tip/pitfall #1)

13. Now go to the Application tab at the top. Select the download checkbox for the ps7_cortexa9_0 row.



Create, manage, and run configurations
Run or Debug a program using System Debugger.

type filter text
Performance Analysis
Target Communication Framework
Xilinx C/C++ application (GDB)
Xilinx C/C++ application (System Debugger on QEMU)
Xilinx C/C++ application (System Debugger)
System Debugger on Local
Filter matched 6 of 12 items

Name: System Debugger on Local

Target Setup | Application | Arguments | Environment | Symbol Files | Source | Path Map | Common

☑ Stop at 'main'

Summary

Download	Processor	Project	Application	Details
✓	ps7_cortexa9_0	logic_analyzer_code	Debug/logic_analyzer_co	reset = true, stop at entry = false, relocate elf = false
☐	ps7_cortexa9_1			reset = true, stop at entry = false, relocate elf = false

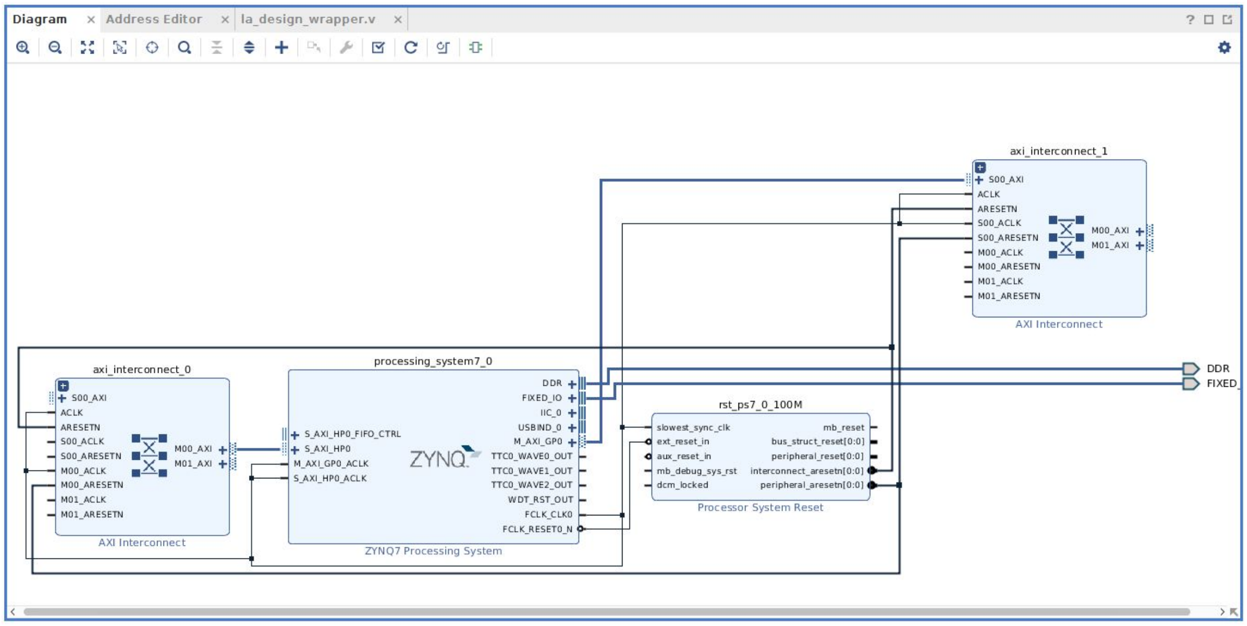
Project Name: logic_analyzer_code
Application: Debug/logic_analyzer_code.elf
☑ Reset processor
☐ Stop at program entry
Advanced Options: Edit...

14. At this point the project name and/or the Application text boxes might have been filled for you. If not, give your project a name and point the application field to your cross compiled executable (this can be found in <project root>/<project name>.sdk/basic_freertos/Debug/<project_name>.elf).
15. Press the Debug button at the bottom of the window and the bitstream and code should be uploaded to your board.
16. Your program should stop at program entry. Set a few break points to explore how it works!

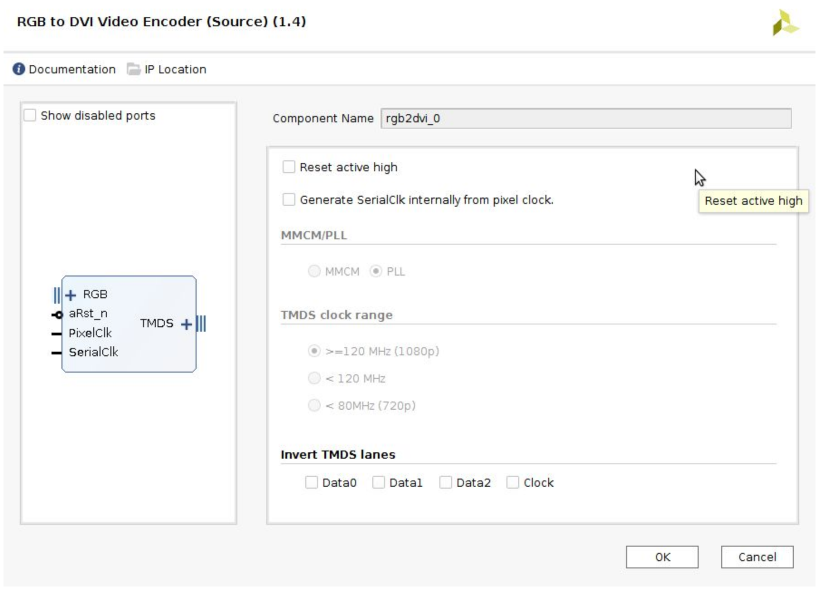
2. Create Video Pipeline
   a. Intro: The video pipeline is one of the most important parts of this logic analyzer project. Since the new Zybo board has an HDMI output port, we elected to use that. This allowed us to get much better resolution than with the previous board's VGA output. While possible to display full HD with the HDMI port that seemed to be overkill, so we chose a display size of 1600x900. A less complicated approach to this pipeline would have been to create a framebuffer in bram then have the video driver directly access that. While this is simpler, a framebuffer is a poor usage of FPGA resources and would be unattainable at a 1600x900 resolution with 3 bytes per pixel (4.3 MB). This however, is a trivially small amount of DRAM; so much so that it was trivial to add double buffering. All of these points directed our development towards the Video DMA IP block and friends.

b. Add IP repo
   1. Intro: The demo that this hardware structure is based off of is this demo project: https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-z7-hdmi-demo/start. Please download this project, it contains the IP repo that we must add for our project to function.
   2. Open vivado, click on IP Catalog in sidebar
   3. Right click on Vivado Repository
   4. Select Add Repository…
   5. Select <Demo Project root>/Zybo-Z7-10-HDMI/repo, click Select

c. Vivado
   1. Open vivado back up and go into the block design
   2. Add a High performance AXI connection to our ZYNQ block. Double click on the ZYNQ7 Processing System block and select S AXI HP0 Interface and M AXI GP0 Interface in the PS-PL Configuration tab.
   3. Now add a new IP block called AXI Interconnect. Drag a connection from the M00_AXI connector on the interconnect to the newly created S_AXI_HP0 on the ZYNQ block. Customize this block so that it only has one master interface and one slave. "Run Connection Automation" will be very useful for the rest of this project. Click it.
   4. Select all boxes available and run
   5. Create a new AXI Interconnect and drag a connection from M_AXI_GP0 on the ZYNQ to S00_AXI on the new Interconnect. Run Connection Automation again, with all defaults.



6. Create new IP: RGB to DVI Video Encoder (Source).


RGB to DVI Video Encoder (Source) (1.4)

Component Name: rgb2dvi_0

[ ] Reset active high

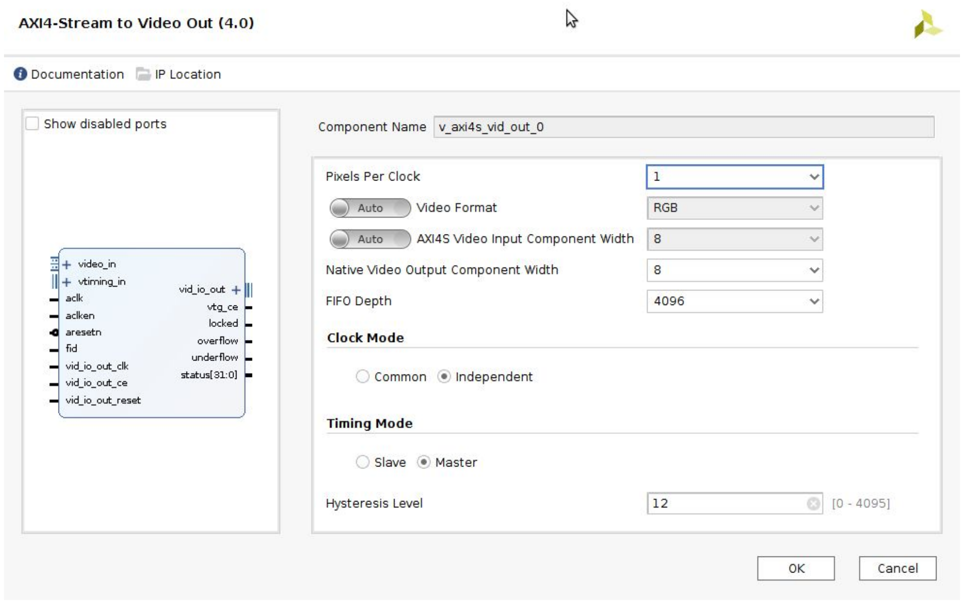
[ ] Generate SerialClk internally from pixel clock.

MMCM/PLL: MMCM [✓ PLL]

TMDS clock range: [✓ >=120 MHz (1080p)] < 120 MHz < 80MHz (720p)

Invert TMDS lanes: [ ] Data0 [ ] Data1 [ ] Data2 [ ] Clock

7. Create new IP: AXI4-Stream to Video Out.


AXI4-Stream to Video Out (4.0)

Component Name: v_axi4s_vid_out_0

Pixels Per Clock: 1

Auto Video Format: RGB

Auto AXI4S Video Input Component Width: 8

Native Video Output Component Width: 8

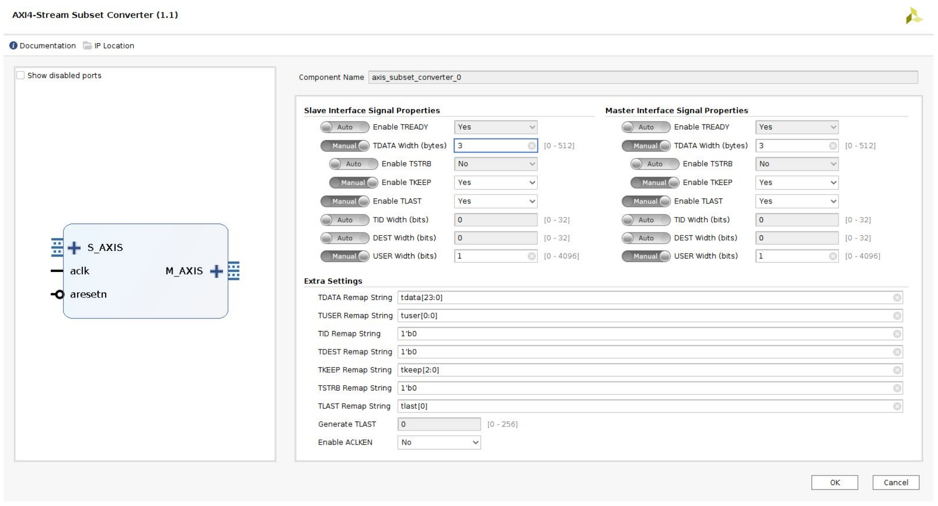
FIFO Depth: 4096

Clock Mode: Common [✓ Independent]

Timing Mode: Slave [✓ Master]

Hysteresis Level: 12 [0 - 4095]

8. Create new IP: AXI4-Stream Subset Converter.



AXI4-Stream Subset Converter (1.1)

Documentation | IP Location

Show disabled ports

Component Name: axis_subset_converter_0

Slave Interface Signal Properties

Auto	Enable TREADY	Yes	
Manual	TDATA Width (bytes)	3	[0 - 512]
Auto	Enable TSTRB	No	
Manual	Enable TKEEP	Yes	
Manual	Enable TLAST	Yes	
Auto	TID Width (bits)	0	[0 - 32]
Auto	DEST Width (bits)	0	[0 - 32]
Manual	USER Width (bits)	1	[0 - 4096]

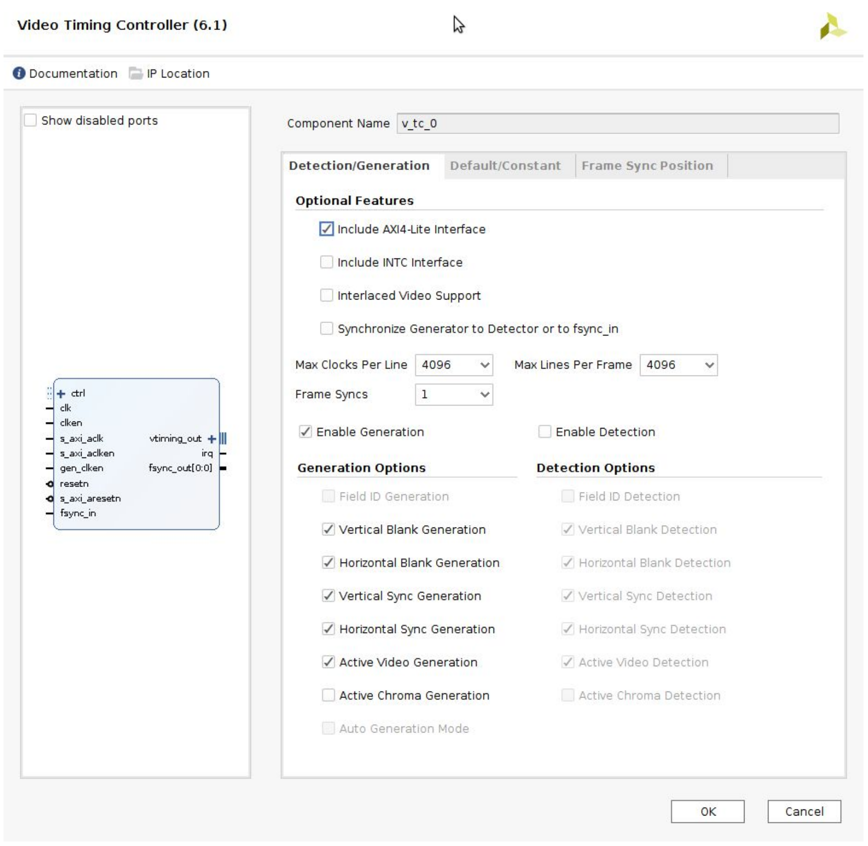
Master Interface Signal Properties

Auto	Enable TREADY	Yes	
Manual	TDATA Width (bytes)	3	[0 - 512]
Auto	Enable TSTRB	No	
Manual	Enable TKEEP	Yes	
Manual	Enable TLAST	Yes	
Auto	TID Width (bits)	0	[0 - 32]
Auto	DEST Width (bits)	0	[0 - 32]
Manual	USER Width (bits)	1	[0 - 4096]

Extra Settings

TDATA Remap String	tdata[23:0]	
TUSER Remap String	tuser[0:0]	
TID Remap String	1'b0	
TDEST Remap String	1'b0	
TKEEP Remap String	tkeep[2:0]	
TSTRB Remap String	1'b0	
TLAST Remap String	tlast[0]	
Generate TLAST	0	[0 - 256]
Enable ACLKEN	No	

9. Create new IP: Video Timing Controller.



Video Timing Controller (6.1)

Documentation | IP Location

Show disabled ports

Component Name: v_tc_0

Detection/Generation | Default/Constant | Frame Sync Position

Optional Features

- [✓] Include AXI4-Lite Interface
- [ ] Include INTC Interface
- [ ] Interlaced Video Support
- [ ] Synchronize Generator to Detector or to fsync_in

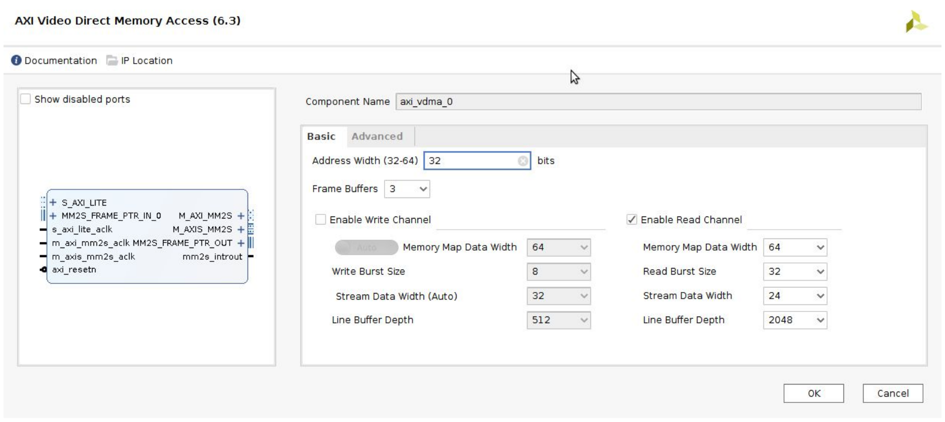
Max Clocks Per Line: 4096 | Max Lines Per Frame: 4096

Frame Syncs: 1

[✓] Enable Generation | [ ] Enable Detection

Generation Options	Detection Options
[ ] Field ID Generation	[ ] Field ID Detection
[✓] Vertical Blank Generation	[✓] Vertical Blank Detection
[✓] Horizontal Blank Generation	[✓] Horizontal Blank Detection
[✓] Vertical Sync Generation	[✓] Vertical Sync Detection
[✓] Horizontal Sync Generation	[✓] Horizontal Sync Detection
[✓] Active Video Generation	[✓] Active Video Detection
[ ] Active Chroma Generation	[ ] Active Chroma Detection
[ ] Auto Generation Mode	

10. Create new IP: Dynamic Clock Generator. (keep defaults)
11. Create new IP: Video Direct Memory Access (keep defaults in advanced tab)



Component Name: axi_vdma_0

Address Width (32-64): 32 bits

Frame Buffers: 3

☐ Enable Write Channel

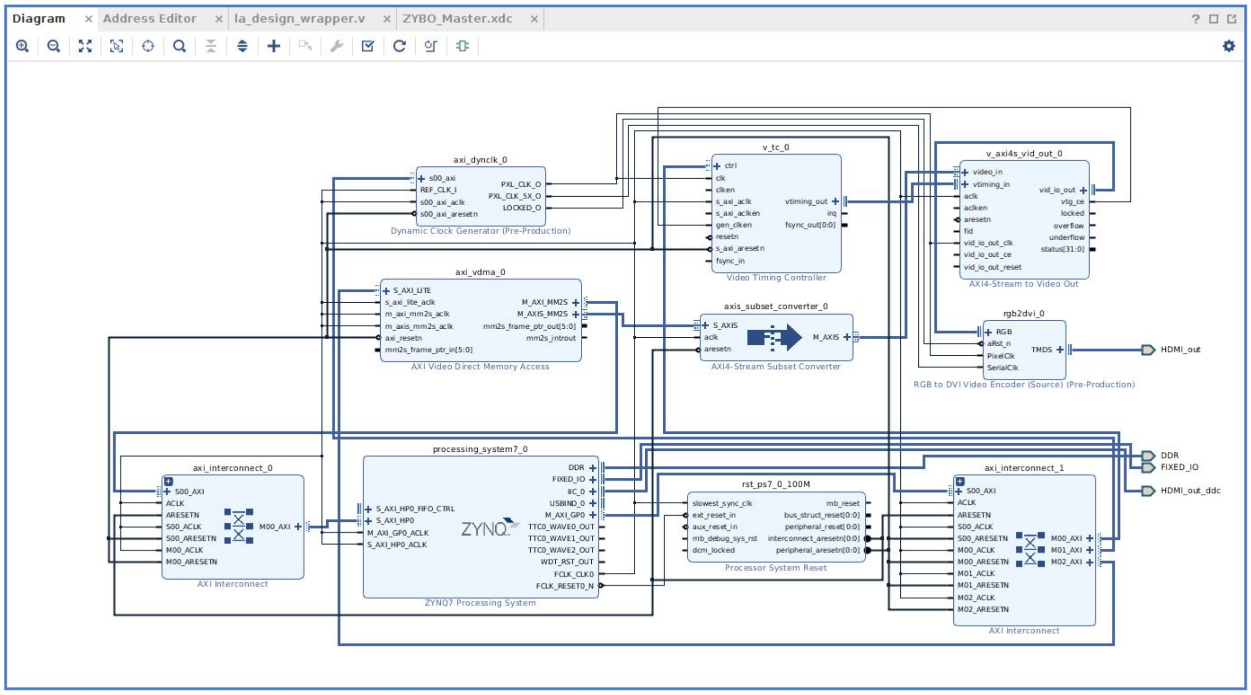
Memory Map Data Width	64
Write Burst Size	8
Stream Data Width (Auto)	32
Line Buffer Depth	512

✓ Enable Read Channel

Memory Map Data Width	64
Read Burst Size	32
Stream Data Width	24
Line Buffer Depth	2048

12. Run Connection Automation. Select everything **except** for rgb2dvi_0 and run
13. Make following connections manually:
    1. Axi_dynclk_0.LOCKED_O -> rgb2dvi_0.aRst_n
    2. v_tc_0.clk -> rgb2dvi_0.PixelClk -> V_axi4s_vid_out_0.vid_io_out_ckl -> axi_dynclk_0.PXL_CLK_O
    3. Axi_dynclk_0.PXL_CLK_5X_O -> rgb2dvi_0.Serial_Clk
    4. V_axi4s_vid_out_0.vtg_ce -> v_tc_0.gen_clken
    5. V_tc_0.vtiming_out -> v_azi4s_vid_out_0.vtiming_in
    6. axi_vdma_0.M_AXIS_MM2S -> axis_subset_converter.S_AXIS
    7. axis_subset_converter.M_AXIS -> v_axi4s_vid_out_0
    8. V_axi4s_vid_out_0.vid_io_out -> rgb2dvi_0.RGB
14. Right click on rgb2dvi.TMDS and select make external. Rename this HDMI_out.

15. Right click on the ZYNQ block’s IIC_0 connection and make that external as well. Rename this HDMI_out_ddc.

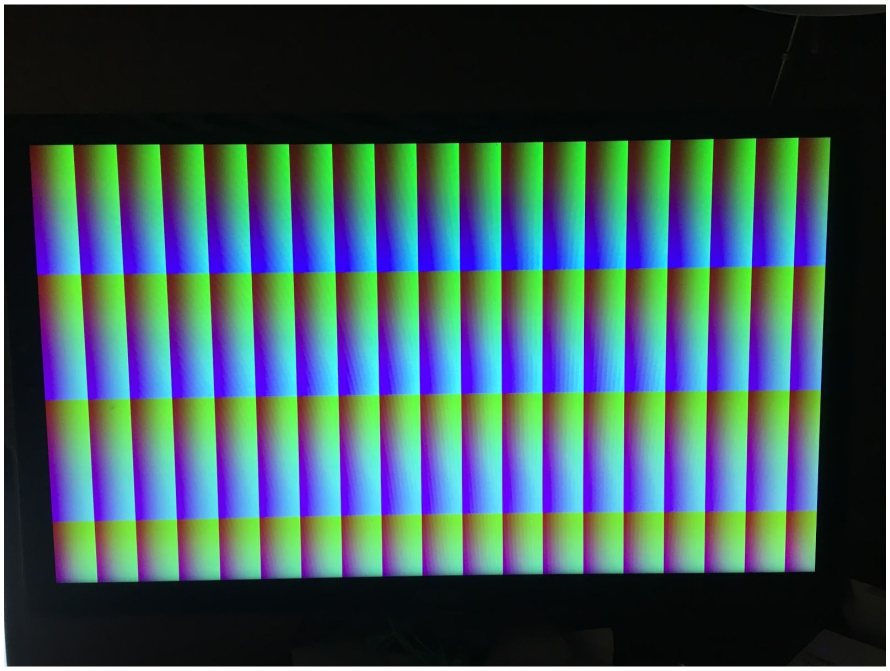


16. Download constraints file ZYBO_Master.xdc
17. Go to sources, right click on Constraints, click on add sources.
18. Select add or create constraints. Click next.
19. Select add files and navigate to the downloaded file. Press finish.
20. Re-generate HDL wrapper.
21. Generate bitstream.
22. Export hardware (include bitstream).
23. Switch to sdk. (Note: if your sdk was already open and you overwrote your last hardware export, the SDK will prompt you with a Warning! Hardware Specification File Change window. Hit yes and it will update itself with the new hardware)

d. SDK

1. Regenerate the BSP by deleting the last one and going to File->new->board support package (just as before).
2. Download the video_source_files.zip and extract the contents into the sdk project src directory.
3. Now only one thing before getting the video output setup!
4. In the main function we need to call our module’s init function and its add_tasks function. (don't forget to include display.h in the main.c file)
5. Once that is complete upload the program to the board, plug in an HDMI cable connected to a monitor, and check out the zybo generated graphics.

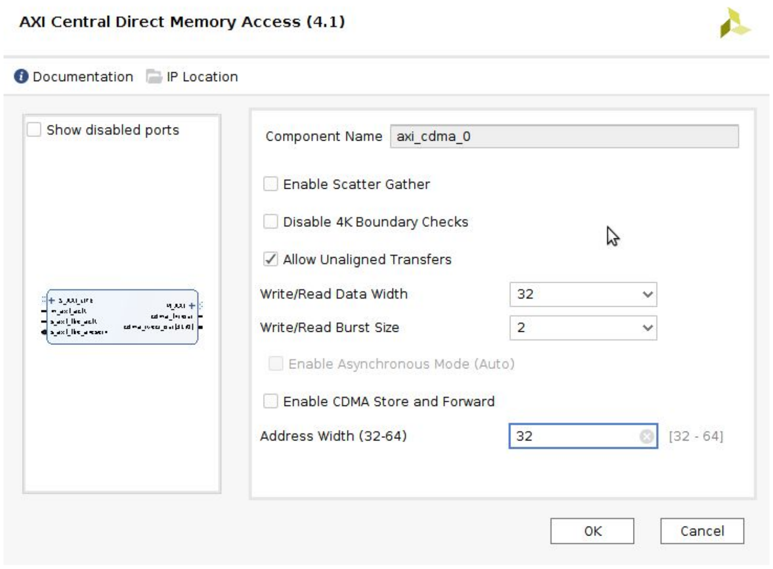
6. Looks great!



7. Now that we have a real task working, we can remove the "mytask" module as it was just a placeholder.

3. Create Data Capture Pipeline
   a. Intro: In this step we will be setting up the blocks used to capture and store the data from the ports, and how to get that data to our main memory. For this project we used a bram buffer to store the input from the ports. Every time a frame update is requested the buffer is transferred via DMA over to the DRAM and is processed for that frame's video output.
   b. Vivado
      1. Back to vivado we go.
      2. Open the block design.

3. Create new IP: AXI Central Direct Memory Access



AXI Central Direct Memory Access (4.1)

Component Name: axi_cdma_0

- [ ] Enable Scatter Gather
- [ ] Disable 4K Boundary Checks
- [x] Allow Unaligned Transfers

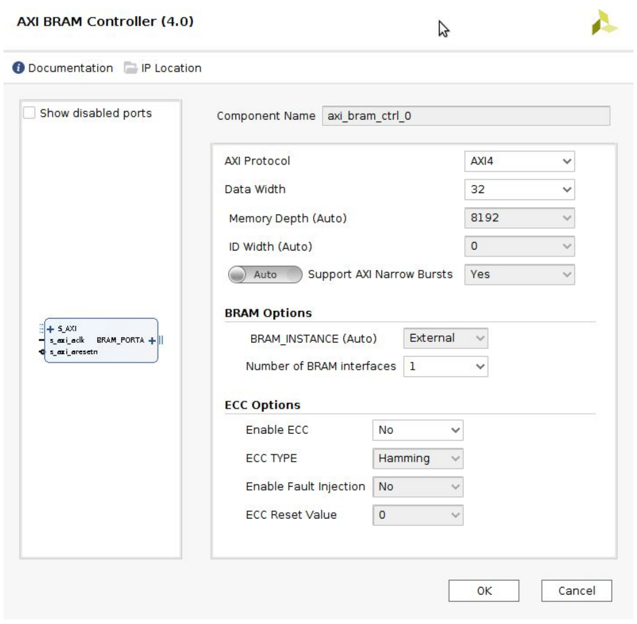
Write/Read Data Width: 32

Write/Read Burst Size: 2

- [ ] Enable Asynchronous Mode (Auto)
- [ ] Enable CDMA Store and Forward

Address Width (32-64): 32 [32 - 64]

4. Create new IP: AXI Bram Controller (don't worry if Memory Depth is not correct, it will be corrected when connected to the BRAM)



AXI BRAM Controller (4.0)

Component Name: axi_bram_ctrl_0

AXI Protocol: AXI4

Data Width: 32

Memory Depth (Auto): 8192

ID Width (Auto): 0

Auto | Support AXI Narrow Bursts: Yes

BRAM Options

BRAM_INSTANCE (Auto): External

Number of BRAM interfaces: 1

ECC Options

Enable ECC: No

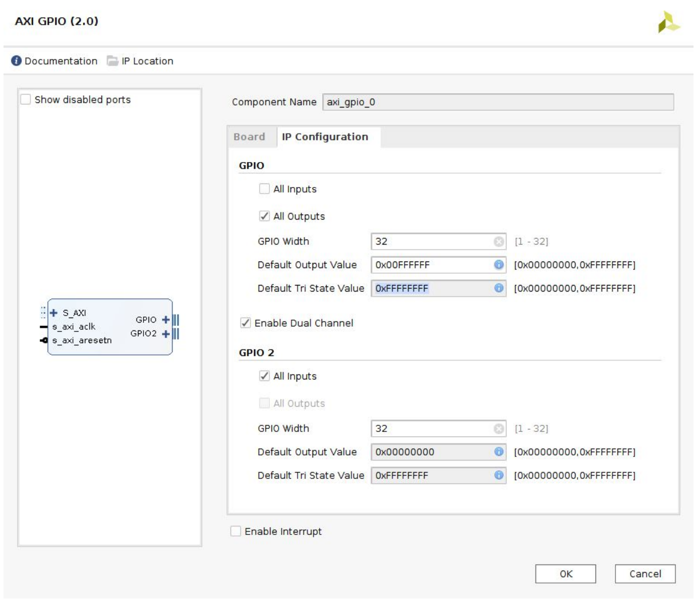
ECC TYPE: Hamming

Enable Fault Injection: No

ECC Reset Value: 0

5. Create new IP: Block Memory Generator
    1. Mode = stand alone
    2. Memory type = true dual port ram
    3. Port A write depth = 2048
6. Customize the HP AXI interconnect to have 2 slaves and 2 masters

7. Make these connections
    1. blk_mem_gen_0.PortA -> axi_bram_ctrl_0.PortA
    2. axi_bram_ctrl_0.S_AXI -> axi_interconnect_0.M01_AXI
    3. axi_cdma_0.M_AXI -> axi_interconnect_0.S01_AXI
8. Right click on blk_mem_gen_0.PortB and make that connection external
9. Run Connection Automation and run all available.
10. Create new IP: AXI GPIO



Component Name: axi_gpio_0

GPIO
- [ ] All Inputs
- [✓] All Outputs

GPIO Width	32	[1 - 32]
Default Output Value	0x00FFFFFF	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

- [✓] Enable Dual Channel

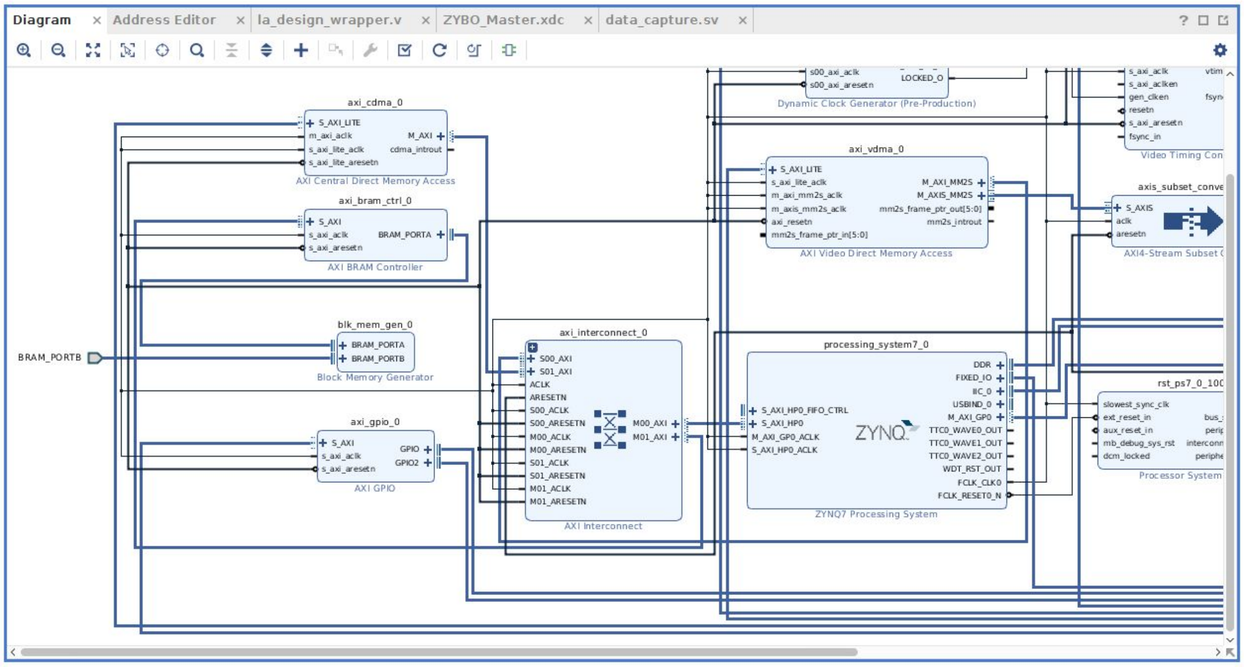
GPIO 2
- [✓] All Inputs
- [ ] All Outputs

GPIO Width	32	[1 - 32]
Default Output Value	0x00000000	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

- [ ] Enable Interrupt

11. Right click on the GPIO connector and make it external with name data_sw_reg
12. Right click on the GPIO2 connector and make it external with name data_hw_reg
13. Right click on the FCLK_CLK0 connector of the PS and make it external as well

14. Run Connection Automation again and run all available



15. Open the address editor tab.
16. Under axi_cdma_0 right click on processing_system7_0 and choose assign address. Do the same for axi_bram_ctrl_0.
17. Under axi_vdma_0, exclude the axi_bram_ctrl_0.
18. Re-generate HDL wrapper.
19. Download data_capture.sv
20. Right click on design sources and select add sources.
21. Select Add or create design sources. click next.
22. Navigate to source file and click finish.
23. Open up your block design wrapper file and remove all of the BRAM_PORTB/data_hw/sw_reg/FCLK_CLK0 inputs and outputs to the module. (Pro tip: from now on, vivado will be more than happy to overwrite your wrapper file if you tell it to; now that you have made changes to it, that isn't a great idea <vivado tips/pitfalls #2)

24. Copy these lines into your wrapper file under the wire declarations:

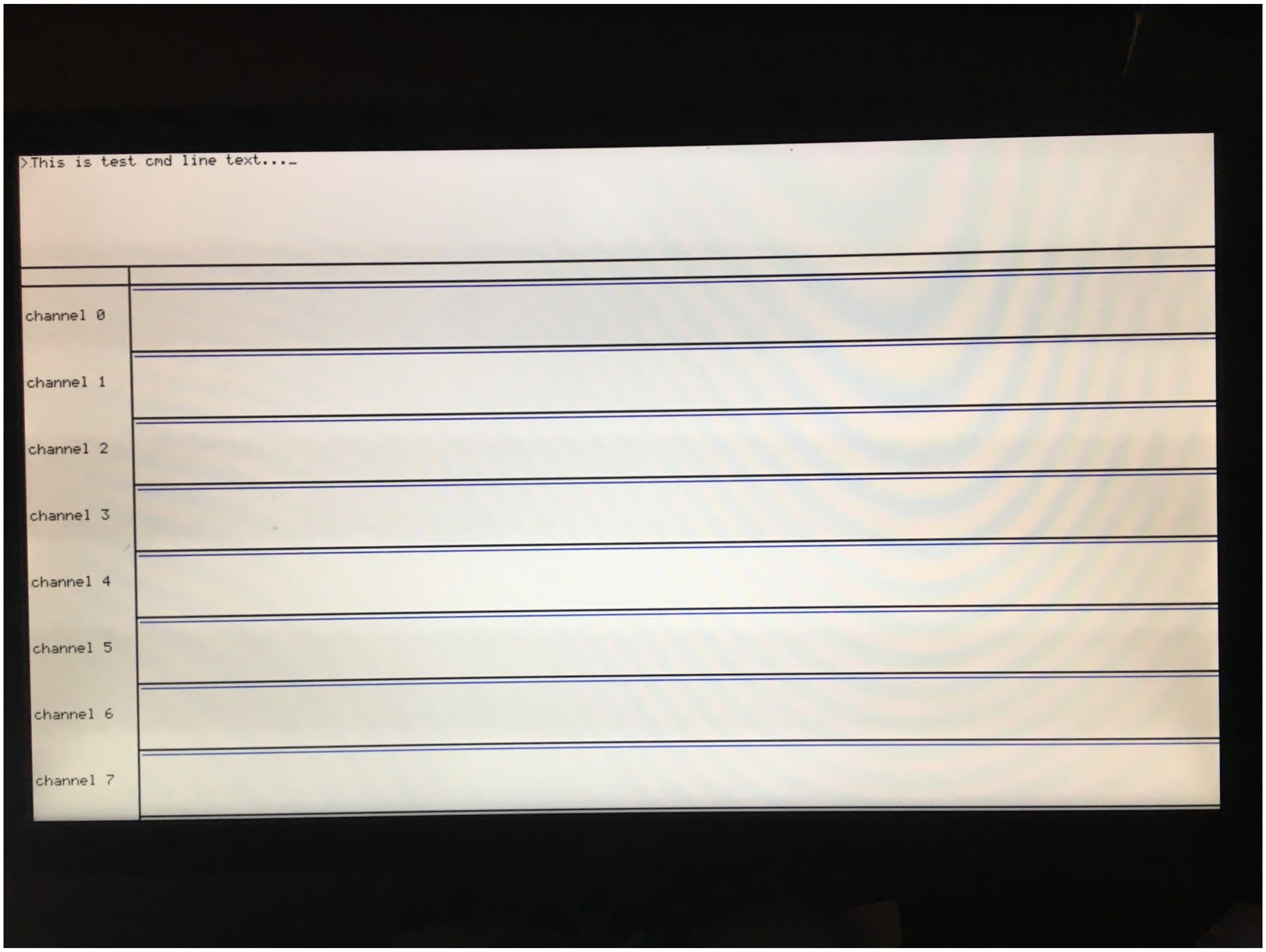
```
data_capture data_capture_i
   (.clock(FCLK_CLK0),
    .reg_in(data_sw_reg_tri_o),
    .reg_out(data_hw_reg_tri_i),
    .ports(ports),
    .addr(BRAM_PORTB_addr),
    .dout(BRAM_PORTB_din),
    .en(BRAM_PORTB_en),
    .we(BRAM_PORTB_we));
```

25. In the actual design block instantiation remove the BRAM_PORTB_dout line and connect FCLK_CLK0 to the BRAM_PORTB_clk.
26. Now add an 8 bit input bus to the wrapper module and call it "ports"
27. Finally uncomment the JC Pmod set_property lines in the ZYBO_Master.xdc file and replace "jc" with "ports".
28. Generate bitstream.
29. Export Hardware.
30. Open sdk.

c. SDK
   1. Regenerate the BSP by deleting the last one and going to File->new->board support package (just as before)
   2. Download these additional source files and extract the contents into the sdk project src directory.
   3. Add in the init and add_tasks for the scope module and data capture is ready to go!
   4. Upload it to the board and set a breakpoint to see the data that is coming off of the pmod connector.

4. Create keyboard Module
   a. For the user interface, a command line was chosen as it can serve as a familiar tool that can also simplify the graphical display of a program. This version uses a keyboard connected through the uart connection to a host computer due to easy and simple implementation. A reasonable effort was put into exploring using a conventional usb keyboard, but do to lack of Xilinx driver support for USB Host mode, this was unfeasible for a quarter long project. A PMOD PS/2 connector option was also explored and seems to be the best option for next features to add to this project.
   b. No vivado changes necessary!
   c. Download and extract the keyboard_source_files.zip into the src directory to enable a uart keyboard.
   d. Upload to the board and set breakpoints to see that the keys are being pressed (Pro tip: use a tool like putty to open a serial connection over the usb cable to the Zybo board)

5. Putting it all together
    a. Developing the graphical display is next. This is a complicated task so depending on your ability/time commitment you can either try to develop your own library or you can use these updated video source files.



    b. Finally, we are going to use semaphores to synchronize our scope task and our display task since display inherently depends on the scope task. A clever way to do this is by creating two counting semaphores. The scope task's semaphore starts at one while the display's starts at zero. On entrance to each of their tasks, each attempts to grab their semaphore. Scope will succeed while display will block. Then upon completing one iteration of a task each will increment the other's semaphore. This creates a strict alternation synchronization. This is best seen in the github repository.