DMA-DE1SOC
Project report

Haoran Geng, Donglai Guo

Stephen Edwards

Columbia University

Important Notice: All hardware files are based on those in 2019 Spring Lab3.

Introduction:

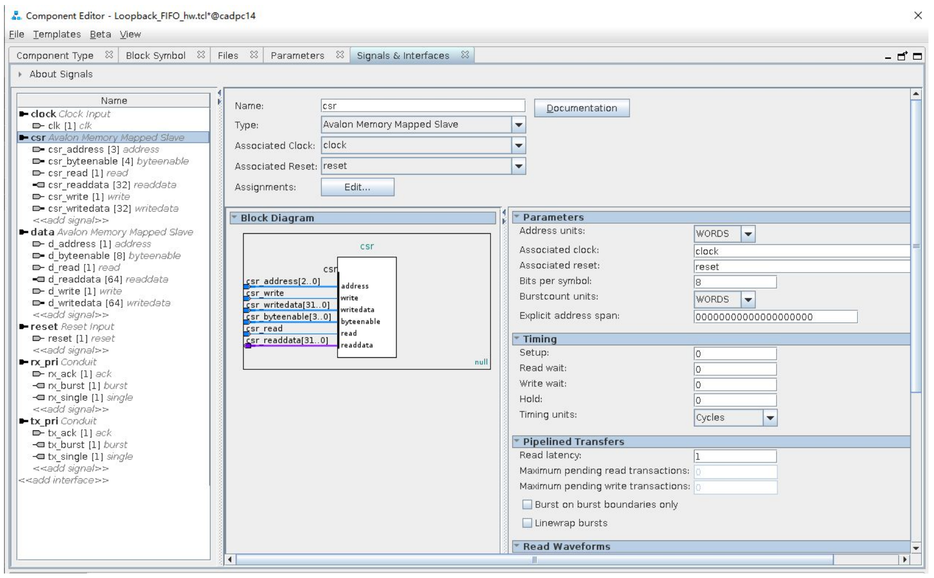
In this project you need to develop a prototypical computational accelerator that 4840 students can use as a starting point for their own project. We supply a base hardware design to extend and working example of a dma peripheral you will have to modify, and a device driver for the existing peripheral that you will have to adapt to work with your own peripheral.

Compile the loopback_FIFO component into a new FPGA Image

In this section you will tell Qsys about a new peripheral component, connect it ultimately to the ARM processors, and synthesize a new FPGA configuration bitstream.

Run qsys-edit soc_system.qsys, which bring up a GUI.

Create a new Loopback_FIFO component and connect it to the base design. Select file-> New Component. In the Component Type tabe set Name to Loopback_FIFO and Display Name to Loopback_FIFO. In the File tab, click Add File… under synthesis Files and select the flow_control_fifo.v file. Click on Analyze Synthesis Files. This should quickly complete successfully; close the pop-up window. Set Top-Level to Loopback_fifo. When Qsys analyzes the synthesis files, it make some good guesses about the meaning of each signal on the peripheral, but it is not perfect. Click on Signal & Interfaces tab. Click on add interface and select conduit. Create all necessary conduit interfaces and instantiate



```
Name
clock Clock Input
    clk [1] clk
csr Avalon Memory Mapped Slave
    csr_address [3] address
    csr_byteenable [4] byteenable
    csr_read [1] read
    csr_readdata [32] readdata
    csr_write [1] write
    csr_writedata [32] writedata
    <<add signal>>
data Avalon Memory Mapped Slave
    d_address [1] address
    d_byteenable [8] byteenable
    d_read [1] read
    d_readdata [64] readdata
    d_write [1] write
    d_writedata [64] writedata
    <<add signal>>
reset Reset Input
    reset [1] reset
    <<add signal>>
rx_pri Conduit
    rx_ack [1] ack
    rx_burst [1] burst
    rx_single [1] single
    <<add signal>>
tx_pri Conduit
    tx_ack [1] ack
    tx_burst [1] burst
    tx_single [1] single
    <<add signal>>
<<add interface>>
```

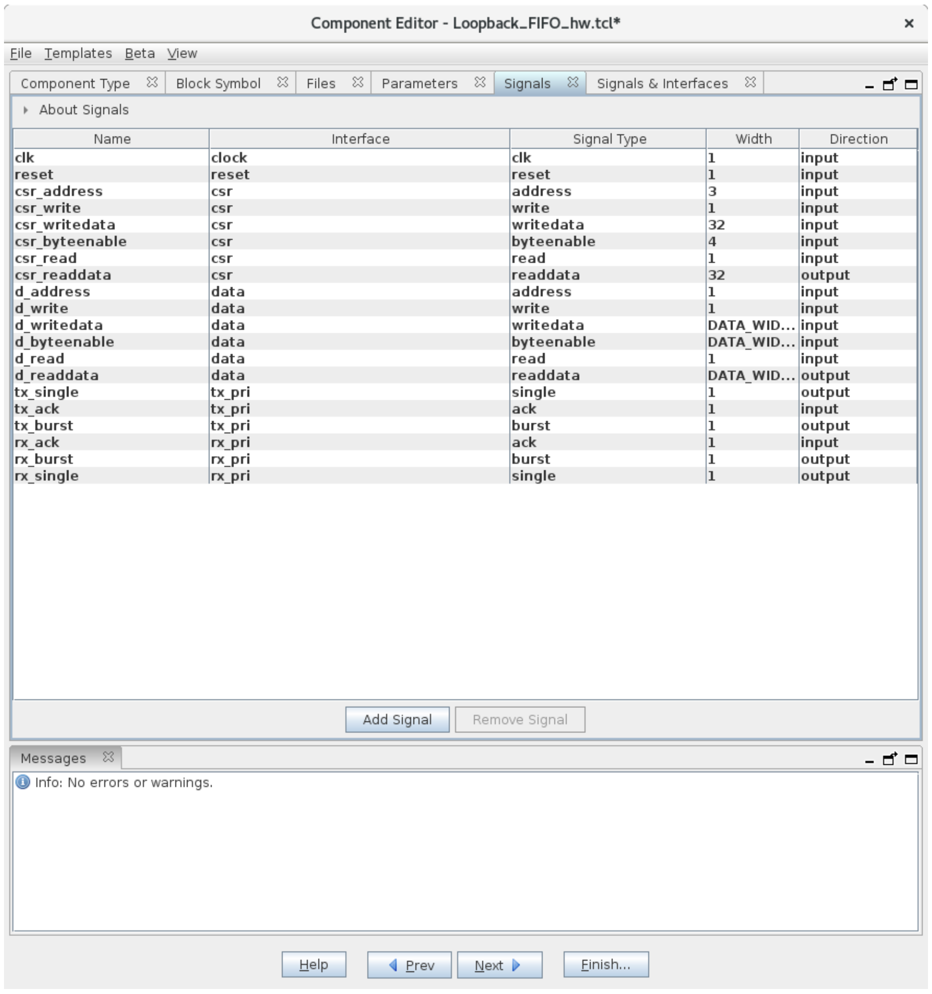
```
Name:                csr
Type:                Avalon Memory Mapped Slave
Associated Clock:    clock
Associated Reset:    reset
Assignments:         Edit...

csr_address[2..0]
csr_write
csr_writedata[31..0]
csr_byteenable[3..0]
csr_read
csr_readdata[31..0]

Parameters
Address units:                        WORDS
Associated clock:                     clock
Associated reset:                     reset
Bits per symbol:                      8
Burstcount units:                     WORDS
Explicit address span:                00000000000000000000
Timing
Setup:                                0
Read wait:                            0
Write wait:                           0
Hold:                                 0
Timing units:                         Cycles
Pipelined Transfers
Read latency:                         1
Maximum pending read transactions:    0
Maximum pending write transactions:   0
[ ] Burst on burst boundaries only
[ ] Linewrap bursts
Read Waveforms
```

all signals like the list on the right.

Select View -> signals from top menu. Verify that your signals match with the screenshot below.

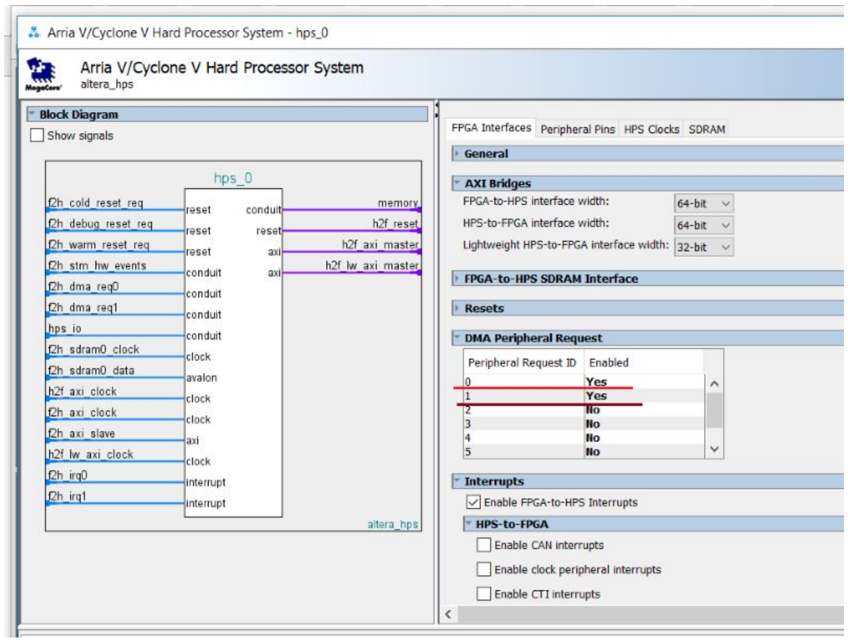


Name	Interface	Signal Type	Width	Direction
clk	clock	clk	1	input
reset	reset	reset	1	input
csr_address	csr	address	3	input
csr_write	csr	write	1	input
csr_writedata	csr	writedata	32	input
csr_byteenable	csr	byteenable	4	input
csr_read	csr	read	1	input
csr_readdata	csr	readdata	32	output
d_address	data	address	1	input
d_write	data	write	1	input
d_writedata	data	writedata	DATA_WID...	input
d_byteenable	data	byteenable	DATA_WID...	input
d_read	data	read	1	input
d_readdata	data	readdata	DATA_WID...	output
tx_single	tx_pri	single	1	output
tx_ack	tx_pri	ack	1	input
tx_burst	tx_pri	burst	1	output
rx_ack	rx_pri	ack	1	input
rx_burst	rx_pri	burst	1	output
rx_single	rx_pri	single	1	output

Info: No errors or warnings.

Once you have eliminated all errors, clock on Finish and connect it to the rest of your design.
In order to make DMA work on hps, we need to change some hps settings. Edit the hps properties in the GUI by right-clicking on hps_0 and edit. Enable DMA channel 0 and 1 by

changing the Enabled setting from No to Yes. Also Enabling FPGA-to-HPS Interrupt by



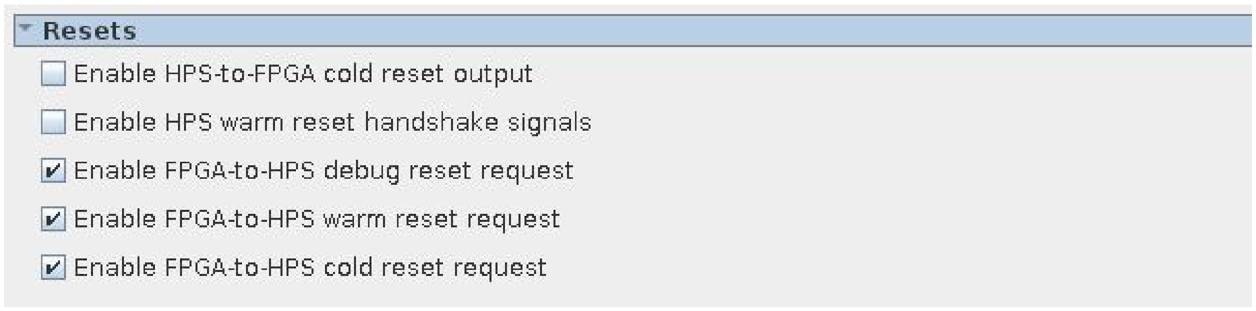
FPGA-to-HPS interface width: 64-bit
HPS-to-FPGA interface width: 64-bit
Lightweight HPS-to-FPGA interface width: 32-bit

Peripheral Request ID	Enabled
0	Yes
1	Yes
2	No
3	No
4	No
5	No

☑ Enable FPGA-to-HPS Interrupts
☐ Enable CAN interrupts
☐ Enable clock peripheral interrupts
☐ Enable CTI interrupts

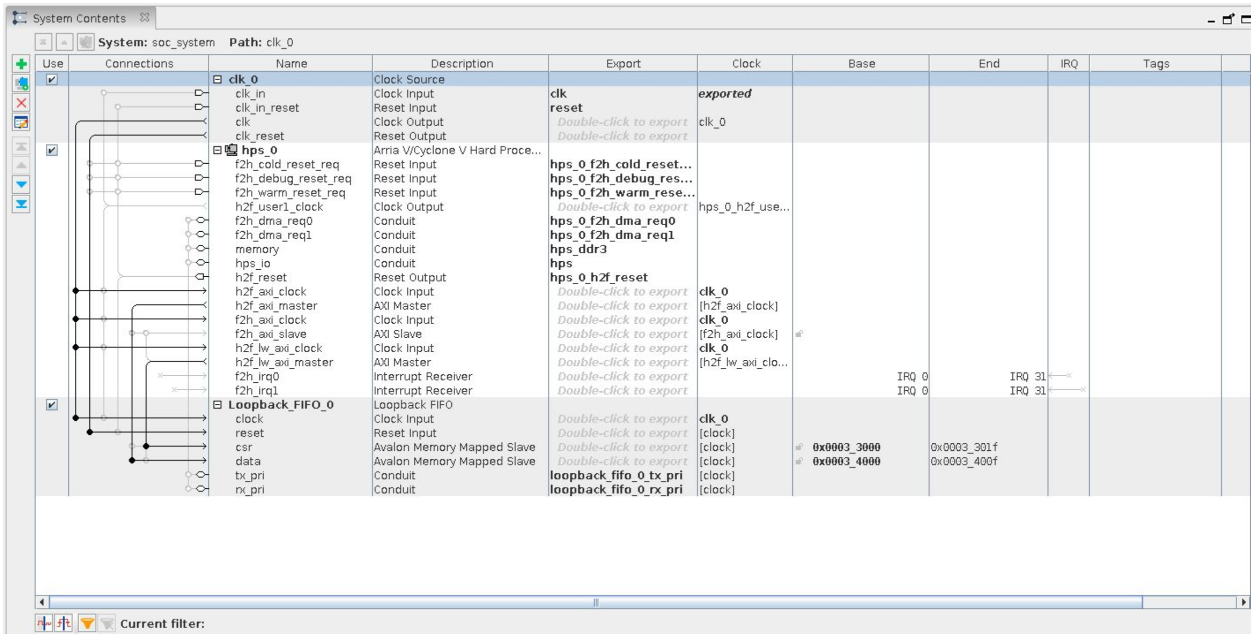
checking the box.

In addition, Enable the debug/warm/cold resets in the hps by checking the boxes.



☐ Enable HPS-to-FPGA cold reset output
☐ Enable HPS warm reset handshake signals
☑ Enable FPGA-to-HPS debug reset request
☑ Enable FPGA-to-HPS warm reset request
☑ Enable FPGA-to-HPS cold reset request

Finally, export signals on top-level design appropriately as shown in the diagram below:



System Contents

System: soc_system **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags
✓		⊟ **clk_0**	Clock Source						
		clk_in	Clock Input	**clk**	***exported***				
		clk_in_reset	Reset Input	**reset**					
		clk	Clock Output	*Double-click to export*	clk_0				
		clk_reset	Reset Output	*Double-click to export*					
✓		⊟ **hps_0**	Arria V/Cyclone V Hard Proce...						
		f2h_cold_reset_req	Reset Input	**hps_0_f2h_cold_reset...**					
		f2h_debug_reset_req	Reset Input	**hps_0_f2h_debug_res...**					
		f2h_warm_reset_req	Reset Input	**hps_0_f2h_warm_rese...**					
		h2f_user1_clock	Clock Output	*Double-click to export*	hps_0_h2f_use...				
		f2h_dma_req0	Conduit	**hps_0_f2h_dma_req0**					
		f2h_dma_req1	Conduit	**hps_0_f2h_dma_req1**					
		memory	Conduit	**hps_ddr3**					
		hps_io	Conduit	**hps**					
		h2f_reset	Reset Output	**hps_0_h2f_reset**					
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**				
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_clock]				
		f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**				
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_clock]				
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**				
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi_clo...				
		f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31		
		f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31		
✓		⊟ **Loopback_FIFO_0**	Loopback FIFO						
		clock	Clock Input	*Double-click to export*	**clk_0**				
		reset	Reset Input	*Double-click to export*	[clock]				
		csr	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0003_3000**	0x0003_301f		
		data	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0003_4000**	0x0003_400f		
		tx_pri	Conduit	**loopback_fifo_0_tx_pri**	[clock]				
		rx_pri	Conduit	**loopback_fifo_0_rx_pri**	[clock]				

Current filter:

In the top level design, export tx_pri and rx_pri as loopback_fifo_0_tx_pri and loopback_fifo_1_rx_pri. Export ward/cold/debug reset by double-click to export. Also, export hps_0_f2h_dma_req0 and hps_0_f2h_dma_req1 and later we will use them to connect with the rx_pri and tx_pri. Click on generate HDL button.

Add the following statement to connect dma_req0 and dma_req1 conduits with rx_pri and tx_pri conduits in soc_system_top.sv (near the end of file):

```
.hps_0_h2f_reset_reset_n              ( hps_fpga_reset_n ),
.hps_0_f2h_cold_reset_req_reset_n      (~hps_cold_reset ),
.hps_0_f2h_debug_reset_req_reset_n     (~hps_debug_reset ),
.hps_0_f2h_warm_reset_req_reset_n      (~hps_warm_reset ),

.hps_0_f2h_dma_req0_dma_req        (rx_burst),
.hps_0_f2h_dma_req0_dma_single     (rx_single),
.hps_0_f2h_dma_req0_dma_ack        (rx_ack),
.hps_0_f2h_dma_req1_dma_req        (tx_burst),
.hps_0_f2h_dma_req1_dma_single     (tx_single),
.hps_0_f2h_dma_req1_dma_ack        (tx_ack),

.loopback_fifo_0_tx_pri_single       (tx_single),
.loopback_fifo_0_tx_pri_burst        (tx_burst),
.loopback_fifo_0_tx_pri_ack          (tx_ack),
.loopback_fifo_0_rx_pri_single       (rx_single),
.loopback_fifo_0_rx_pri_burst        (rx_burst),
.loopback_fifo_0_rx_pri_ack          (rx_ack)
```

Modify the line .reset_reset_n   1'b1 into .reset_reset_n(hps_fpga_reset_n) to make the overall system consistent on one reset signal. Then, insert in the existing ips under lab3-hw/ip into the soc_system_top.sv file by adding the following lines after the declaration of soc_system instance:

```
debounce debounce_inst (
  .clk                              (fpga_clk_50),
  .reset_n                          (hps_fpga_reset_n),
  .data_in                          (KEY),
  .data_out                         (fpga_debounced_buttons)
);
  defparam debounce_inst.WIDTH = 4;
  defparam debounce_inst.POLARITY = "LOW";
  defparam debounce_inst.TIMEOUT = 50000;            // at 50Mhz this is a debounce time of 1ms
  defparam debounce_inst.TIMEOUT_WIDTH = 16;         // ceil(log2(TIMEOUT))

// Source/Probe megawizard instance
hps_reset hps_reset_inst (
  .source_clk (fpga_clk_50),
  .source     (hps_reset_req)
);

altera_edge_detector pulse_cold_reset (
  .clk       (fpga_clk_50),
  .rst_n     (hps_fpga_reset_n),
  .signal_in (hps_reset_req[0]),
  .pulse_out (hps_cold_reset)
);
  defparam pulse_cold_reset.PULSE_EXT = 6;
  defparam pulse_cold_reset.EDGE_TYPE = 1;
  defparam pulse_cold_reset.IGNORE_RST_WHILE_BUSY = 1;

altera_edge_detector pulse_warm_reset (
  .clk       (fpga_clk_50),
  .rst_n     (hps_fpga_reset_n),
  .signal_in (hps_reset_req[1]),
  .pulse_out (hps_warm_reset)
);
  defparam pulse_warm_reset.PULSE_EXT = 2;
  defparam pulse_warm_reset.EDGE_TYPE = 1;
  defparam pulse_warm_reset.IGNORE_RST_WHILE_BUSY = 1;

altera_edge_detector pulse_debug_reset (
  .clk       (fpga_clk_50),
  .rst_n     (hps_fpga_reset_n),
  .signal_in (hps_reset_req[2]),
  .pulse_out (hps_debug_reset)
);
  defparam pulse_debug_reset.PULSE_EXT = 32;
  defparam pulse_debug_reset.EDGE_TYPE = 1;
  defparam pulse_debug_reset.IGNORE_RST_WHILE_BUSY = 1;
```

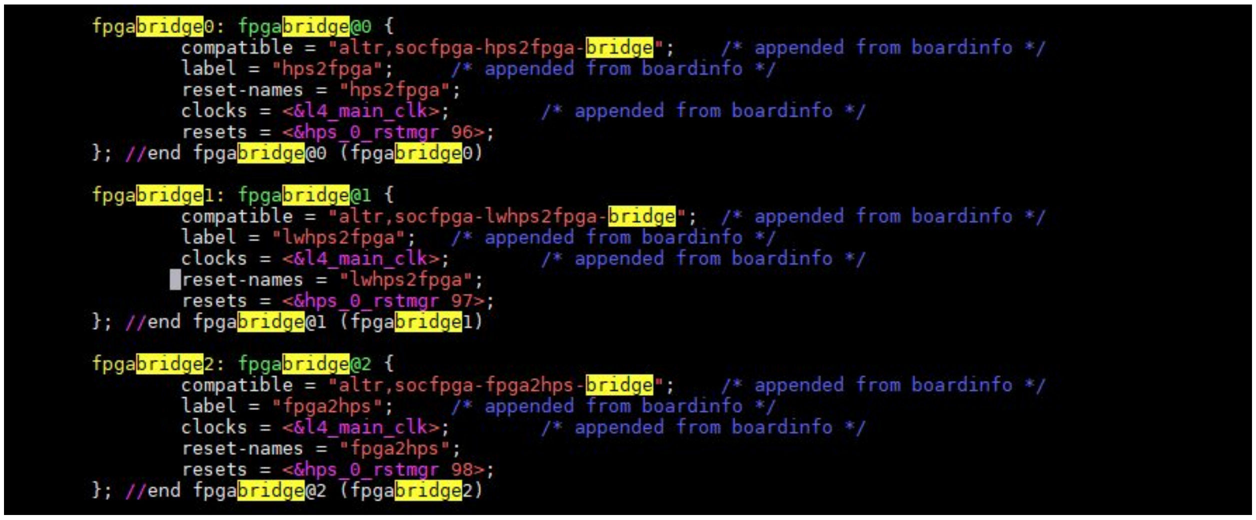
Compile the Hardware Design with Quartus by running make quartus followed by make rbf and make dts. Open up the soc_system.dts file and edit the loopback_fifo part into fpga-dma like the following:

```
fpga_dma: fpga_dma@0x10033000 {
                    compatible = "altr,fpga-dma";
                    reg = <0x00000001 0x00033000 0x00000020>,
                        <0x00000000 0x00034000 0x00000010>;
                    reg-names = "csr", "data";
                    dmas = <&hps_0_dma 0 &hps_0_dma 1>;
                    dma-names = "rx", "tx";
                    clocks = <&clk_0>
        }
```

Notice that the address of fpga_dma component is auto-generated, if your address does not match with the ones shown above, it is fine too. As long as the length of csr registers space is 0x20 and length of data registers space is 0x10 (64-bit data width). In the hps_0_dma section, manually add another interrupt line to the dma like the following:

```
hps_0_dma: dma@0xffe01000 {
         compatible = "arm,pl330-16.1", "arm,pl330",
     "arm,primecell";
     reg = <0xffe01000 0x00001000>;
     interrupt-parent = <&hps_0_arm_gic_0>;
     interrupts = <0 104 4>, <0 105 4>;
     clocks = <&l4_main_clk>;
     ......
}
```

If you do not add the interrupt for the second receive line, the rx dma transaction will always report a timeout error.
In addition, you need to add the resets signals to all three bridges that you are using like the following:



```
        fpgabridge0: fpgabridge@0 {
                compatible = "altr,socfpga-hps2fpga-bridge";    /* appended from boardinfo */
                label = "hps2fpga";     /* appended from boardinfo */
                reset-names = "hps2fpga";
                clocks = <&l4_main_clk>;        /* appended from boardinfo */
                resets = <&hps_0_rstmgr 96>;
        }; //end fpgabridge@0 (fpgabridge0)

        fpgabridge1: fpgabridge@1 {
                compatible = "altr,socfpga-lwhps2fpga-bridge";  /* appended from boardinfo */
                label = "lwhps2fpga";   /* appended from boardinfo */
                clocks = <&l4_main_clk>;        /* appended from boardinfo */
                reset-names = "lwhps2fpga";
                resets = <&hps_0_rstmgr 97>;
        }; //end fpgabridge@1 (fpgabridge1)

        fpgabridge2: fpgabridge@2 {
                compatible = "altr,socfpga-fpga2hps-bridge";    /* appended from boardinfo */
                label = "fpga2hps";     /* appended from boardinfo */
                clocks = <&l4_main_clk>;        /* appended from boardinfo */
                reset-names = "fpga2hps";
                resets = <&hps_0_rstmgr 98>;
        }; //end fpgabridge@2 (fpgabridge2)
```

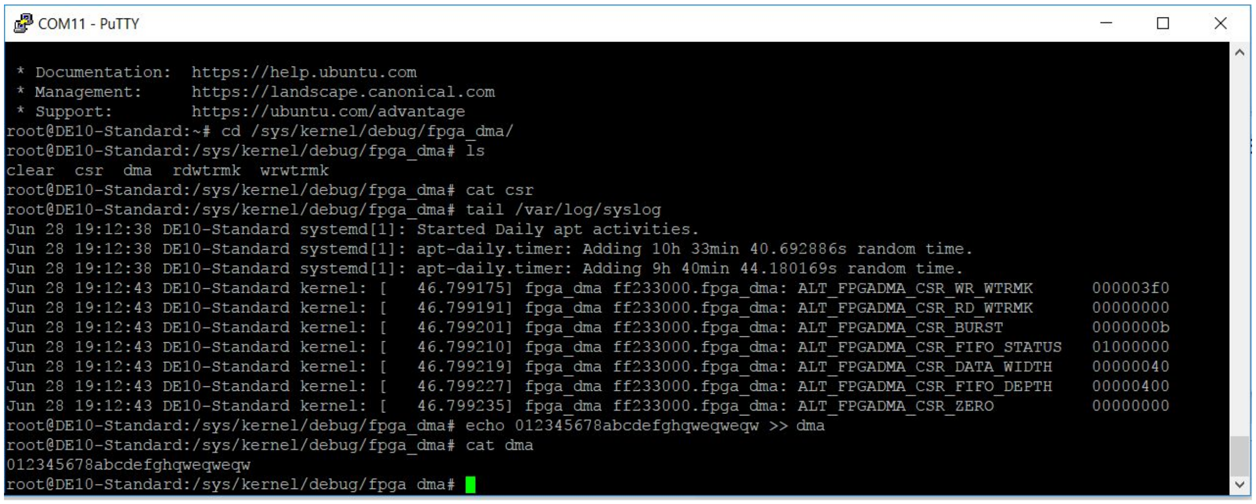
This will ensure that when booting up the boards, all these bridges could be registered as devices in the device tree. Otherwise, the system will complain about the missing resets signals.

Run make dtb and copy over the rbf file and dtb file to the micro sd card by using scp command.

Reboot the fpga system and run insmod fpga-dma.ko.

Before actually performing real DMa transactions, you will need to make sure that the reset manager cleared out the bits for the dma channels that you are using. According to the Altera Cyclone V manual, the dma reset registers are located at 0xFFD05018. Configure the Reset manager using software can be done by writing 0xFC to that memory location. Our way of doing it is by using *busybox devmem 0xFFD05018 w 0xfc (where busybox is a linux tool and you can get it by apt-get install busybox).* After this you could start writing user-level testbenches to verify the correctness of DMA.

If you want to do a quick experiment with DMA in SoC, I recommend starting experiments with the fpga-dma alter driver. It uses DebugFS, this allows you to use the simple “cat”, “echo” commands directly in the terminal console to perform transactions in the DMA channel:



```
 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
root@DE10-Standard:~# cd /sys/kernel/debug/fpga_dma/
root@DE10-Standard:/sys/kernel/debug/fpga_dma# ls
clear  csr  dma  rdwtrmk  wrwtrmk
root@DE10-Standard:/sys/kernel/debug/fpga_dma# cat csr
root@DE10-Standard:/sys/kernel/debug/fpga_dma# tail /var/log/syslog
Jun 28 19:12:38 DE10-Standard systemd[1]: Started Daily apt activities.
Jun 28 19:12:38 DE10-Standard systemd[1]: apt-daily.timer: Adding 10h 33min 40.692886s random time.
Jun 28 19:12:38 DE10-Standard systemd[1]: apt-daily.timer: Adding 9h 40min 44.180169s random time.
Jun 28 19:12:43 DE10-Standard kernel: [   46.799175] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_WR_WTRMK      000003f0
Jun 28 19:12:43 DE10-Standard kernel: [   46.799191] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_RD_WTRMK      00000000
Jun 28 19:12:43 DE10-Standard kernel: [   46.799201] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_BURST         0000000b
Jun 28 19:12:43 DE10-Standard kernel: [   46.799210] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_FIFO_STATUS   01000000
Jun 28 19:12:43 DE10-Standard kernel: [   46.799219] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_DATA_WIDTH    00000040
Jun 28 19:12:43 DE10-Standard kernel: [   46.799227] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_FIFO_DEPTH    00000400
Jun 28 19:12:43 DE10-Standard kernel: [   46.799235] fpga_dma ff233000.fpga_dma: ALT_FPGADMA_CSR_ZERO          00000000
root@DE10-Standard:/sys/kernel/debug/fpga_dma# echo 012345678abcdefghqweqweqw >> dma
root@DE10-Standard:/sys/kernel/debug/fpga_dma# cat dma
012345678abcdefghqweqweqw
root@DE10-Standard:/sys/kernel/debug/fpga_dma# 
```

In our system, if you want to know the contents of all csr registers. You could type in cat csr and then dmesg.
The csr documentation could be found in the flow_control.v file, also listed here:

This component is an Avalon-MM based FIFO with hardware flow control compatible with the DMA-330 core in the hard processor system (HPS). This component has a data port for writing data into the FIFO and reading data out of the FIFO. The CSR port is for setting the read and write watermarks as well as reading various status bits.

The write watermark represents the amount of space (in words) that the component needs to wait for being available in the FIFO before it performs a burst request to the DMA. This watermark must match the burst size programmed into the DMA memory-to-device channel code.

The read watermark represents the amount of space (in words) that the component needs to buffer in the FIFO before it performs a burst request to the DMA. This watermark must match the burst size programmed into the DMA device-to-memory channel code.

The data width configured for this component must match the burst size programmed into the DMA channel code. Failure to do so will cause a FIFO over/under flow.

The Synopsys protocol is used for the hardware flow control which is somewhat different than the protocol used by the DMA-330 core (ARM protocol). The protocol is as follows:

1) Single transfer line is asserted any time the TX FIFO channel is not full or the RX FIFO channel is not empty.

2) Burst transfer line is asserted when the programmed watermark has not been crossed.

3) The single and burst transfer lines must remain asserted until the DMA issues the acknowledge back to the peripheral. Even if the programmed watermark is crossed or the FIFO becomes full/empty, the peripheral must continue issuing the burst and single requests until acknowledged by the DMA.

4) When a transfer is acknowledged the peripheral *must* deassert the burst and single request lines for at least one clock cycle.

The following diagram shows the FIFO fill level and watermarks and describes when each of the request lines will be asserted:

```
<------------------------------ FIFO DEPTH ------------------------------->

|-----------------------------------------------------------------------|
|**************************************                                 |
|-----------------------------------------------------------------------|
0       ^                      ^                      ^   FIFO_DEPTH
        |                      |                      |
    RX watermark           Fill Level            TX Watermark
```

- RX single is high because the FIFO is not empty
- RX burst is high because the fill level is equal to or greater than the read watermark
- TX single is high because the FIFO is not full
- TX burst is high because the fill level is equal to or less than the write watermark

Author: JCJB
Date: 11/14/2013
Revision: 1.0

Revision History:

1.0 - First version

Data Port (Width depends on the DATA_WIDTH parameter)

Offset	Access	Register
0	W	Write Data
1	R	Read Data

CSR Port (32-bit)

Offset	Access	Register
0	R/W	TX Watermark
1	R/W	RX Watermark
2	R	[3..0] --> rx_burst, rx_single, tx_burst, tx_single
3	R	[25..0] --> fifo_full, fifo_empty, fifo_used[23:0]
4	R	Data width
5	R	FIFO depth
6	Wclr	FIFO clear (write 1 to clear the FIFO)
7	R	Unused (zeros)

Working flow:

We first followed the Linux example on the Xilinx wiki. We cannot pass the initial test on the example code. The kernel reported the bug at dmaengine.h line 329, and it also reported dma slave request time out. We tried several more examples and the bug is the same. We find out that all the examples are work on xilinx fpga and they all have corresponding hardware to support their dma driver. We could not find hardware examples for DE1-Soc.

Therefore, we tried to use the lab3 hardware from 4840 as our starting point to build our hardware for the dma driver. We opened the HPS access to the dma and tied to run the same xilinx example on software side. It still showed the same error before.

Finally, we decided to change the software for this project. We used Altera libaray fpga-dma.c as our example code. We removed all the VGA display parts on the hardware, and added a FIFO connect to the HPS to handle the data. After we figured out how to get the dma access on the kernel, we finally made the dma transfer work.