Spring 2022

EE 382N-4: Advanced Micro-Controller Systems

Lab Assignment #2

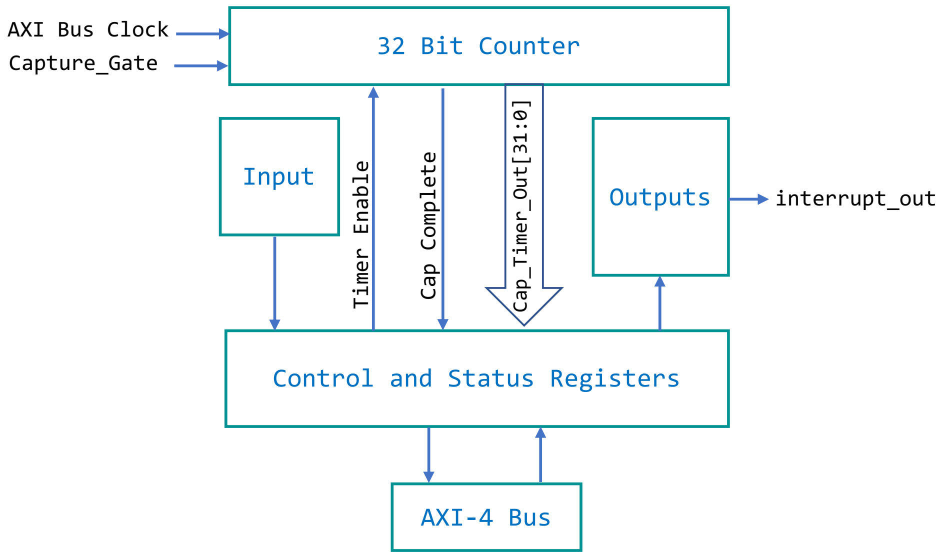
DUE MAR 4TH, 2022

Lab Goals:

This lab focuses on the instrumentation of AXI transfers and Interrupt Service Routines (ISR). The first part will measure the time it takes to DMA data to/from the OCM and from/to the BRAM. The second part measures the time it takes to respond to an interrupt from the PL.

Initial Setup:

Generate a new AXI peripheral: **`CAPTURE_TIMER`**. The block diagram is shown below. The timer-capture counter that is gated by the CDMA Interrupt signal. The counter is started when the transfer starts and then stops when the CDMA unit interrupts the processor. The counter is read in the CDMA ISR and sent to the main routine for processing.



NOTE: The **`interrupt_out`** pin needs to be configured as an interrupt pin during editing in the IP Packager so that the DTB assigns an interrupt number to the pin. The interrupt is asserted when the “Capture Complete” is enabled. The counter and the interrupt are disabled when the ISR exits.

The **`CAPTURE_TIMER`** block will be implemented in VERILOG. It will have 4 registers that will be used for this Lab.

Reading register port `slv_reg0[31:0]` will return the following

```
    slv_reg0[0]     = capture_gate;           // CDMA interrupt out signal
                                              // to the GIC. Used to halt counter.
    slv_reg0[1]     = 1'b0;
    slv_reg0[2]     = 1'b0;
    slv_reg0[3]     = 1'b0;
    slv_reg0[4]     = capture_complete;       // Flag to indicate that the
                                              // capture is complete
    slv_reg0[5]     = 1'b0;
    slv_reg0[6]     = 1'b0;
    slv_reg0[7]     = timer_enable;           // Timer enable signal
    slv_reg0[31:8]  = {16'hBEAD,8'h0};        // Debug
```

Writing to register port `slv_reg1[0]` will be used to assert or negate the "`interrupt_out`" port pin.

```
   assign interupt_out = slv_reg1[0];          // Active high asserts an
                                              // interrupt to the GIC.
```

Writing to register port `slv_reg1[1]` is used to enable counting.

```
   assign timer_enable  = slv_reg1[1];        // Active high enables
                                              // capture timer to count.
```

Reading register `slv_reg1[31:0]` will return the following:

```
    slv_reg1[0]    = interrupt_out;
    slv_reg1[1]    = timer_enable;
    slv_reg1[31:2] = {16'hFEED,14'h0};        // Debug
```

Reading register port `slv_reg2[31:0]` reads the capture timer counter value.

```
    slv_reg2[31:0] = Cap_Timer_Out[31:0];
```

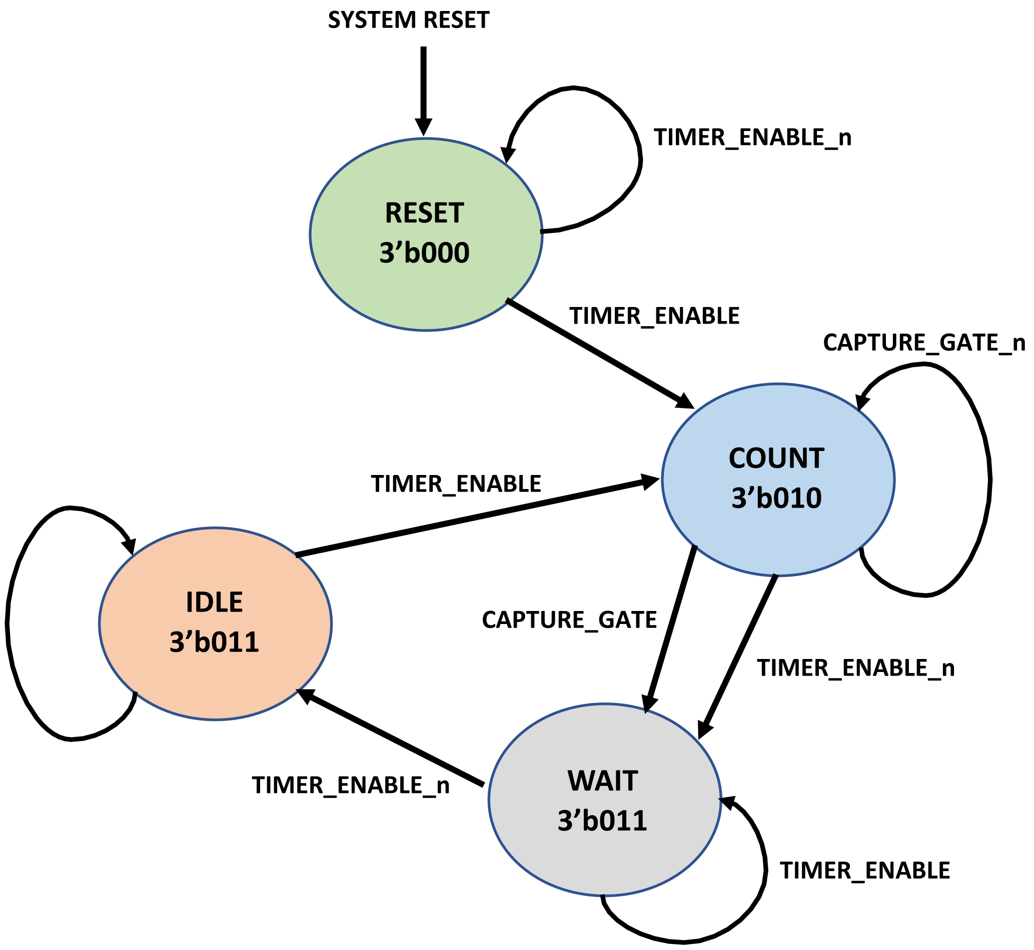
Reading register port `slv_reg3[31:0]` reads the following:

```
    slv_reg3[2:0]    = state[2:0];            // This is the current state of
                                              // the timer counter state machine.
    slv_reg3[3]      = 1'b0;
    slv_reg3[31:4]   = 28'h5555_CAB;          // Debug
```

The `state[2:0]` data is for debug. A typical state assignment for a timer counter would be:

```
    parameter     RESET  = 3'b000;
    parameter     LOAD   = 3'b001;            // Used to load an offset values
    parameter     COUNT  = 3'b010;
    parameter     WAIT   = 3'b011;
    parameter     IDLE   = 3'b100;
```

The state diagram for this assignment would look like:

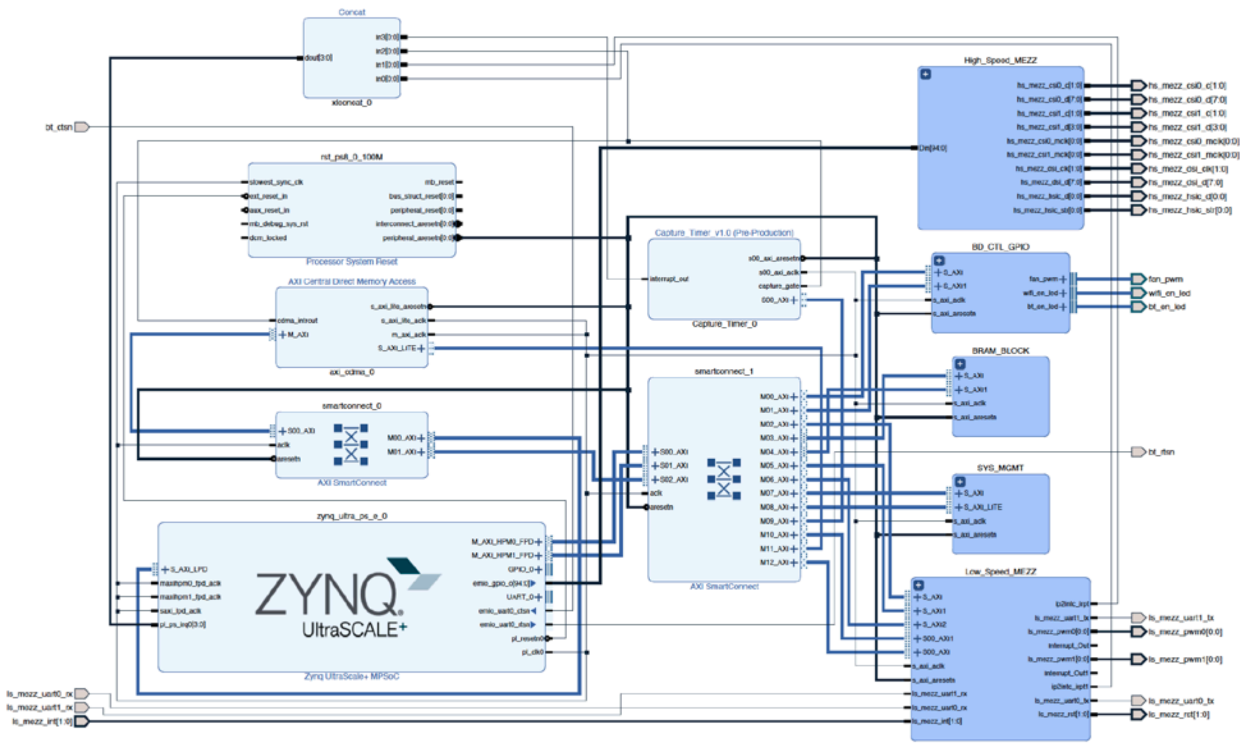


Note: the LOAD function is not implemented in this state machine. This would be used add an offset to the timer to account for the latency of the CDMA starting the transfer.

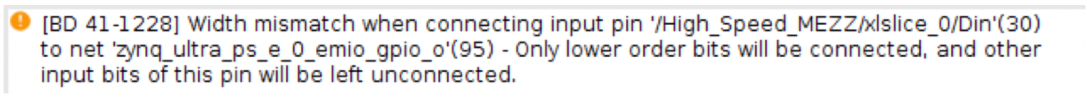
Schematic design procedure:

Referring to the block diagram below:

- Use the Vivado schematic from Lab 1 and insert the ***CAPTURE_TIMER*** as shown in the block diagram below.
- Modify the ***CONCAT*** module to add an additional interrupt input pin. Connect the **interrupt_out** signal from ***CAPTURE_TIMER*** module to the ***CONCAT*** module

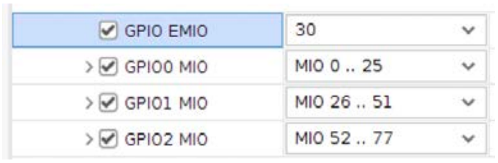


The next step is to generate a bit file that can be dynamically downloaded into the FPGA. Refer to this guide on how to generate the FPGA bit file: [BIT File Generation Flow]
You may see the following warning messages when you generate the bit file:

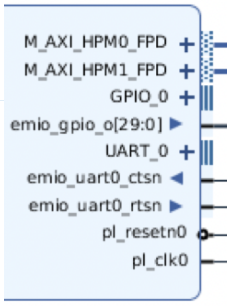


[BD 41-1228] Width mismatch when connecting input pin '/High_Speed_MEZZ/xlslice_0/Din'(30) to net 'zynq_ultra_ps_e_0_emio_gpio_o'(95) - Only lower order bits will be connected, and other input bits of this pin will be left unconnected.

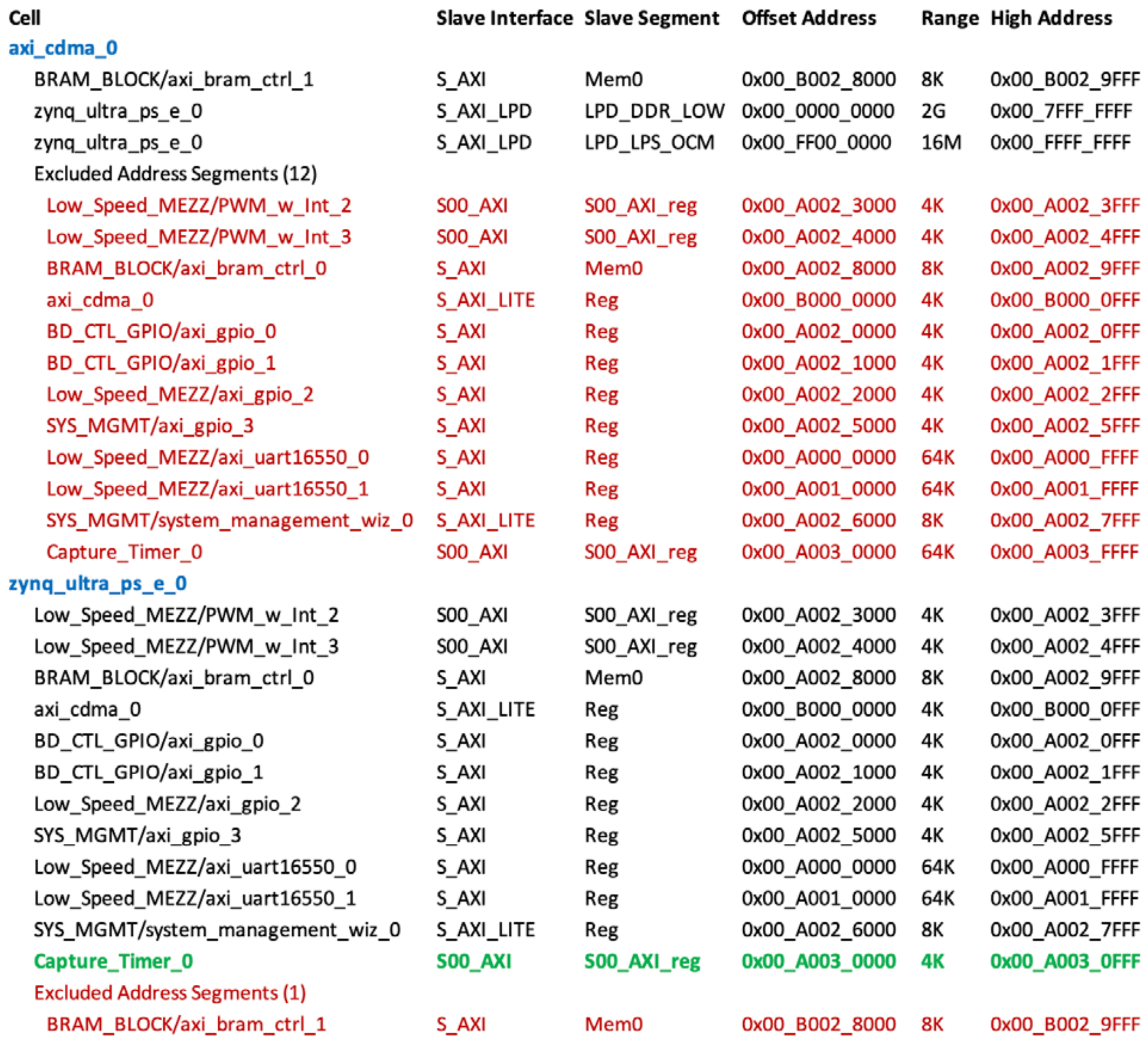
This is normal for the EMIO port on the ZynqMP. This can be fixed by setting the number of port pins to [29:0]:



☑ GPIO EMIO	30
› ☑ GPIO0 MIO	MIO 0 .. 25
› ☑ GPIO1 MIO	MIO 26 .. 51
› ☑ GPIO2 MIO	MIO 52 .. 77



The updated Address Map must look like this:



Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
axi_cdma_0					
BRAM_BLOCK/axi_bram_ctrl_1	S_AXI	Mem0	0x00_B002_8000	8K	0x00_B002_9FFF
zynq_ultra_ps_e_0	S_AXI_LPD	LPD_DDR_LOW	0x00_0000_0000	2G	0x00_7FFF_FFFF
zynq_ultra_ps_e_0	S_AXI_LPD	LPD_LPS_OCM	0x00_FF00_0000	16M	0x00_FFFF_FFFF
Excluded Address Segments (12)					
Low_Speed_MEZZ/PWM_w_Int_2	S00_AXI	S00_AXI_reg	0x00_A002_3000	4K	0x00_A002_3FFF
Low_Speed_MEZZ/PWM_w_Int_3	S00_AXI	S00_AXI_reg	0x00_A002_4000	4K	0x00_A002_4FFF
BRAM_BLOCK/axi_bram_ctrl_0	S_AXI	Mem0	0x00_A002_8000	8K	0x00_A002_9FFF
axi_cdma_0	S_AXI_LITE	Reg	0x00_B000_0000	4K	0x00_B000_0FFF
BD_CTL_GPIO/axi_gpio_0	S_AXI	Reg	0x00_A002_0000	4K	0x00_A002_0FFF
BD_CTL_GPIO/axi_gpio_1	S_AXI	Reg	0x00_A002_1000	4K	0x00_A002_1FFF
Low_Speed_MEZZ/axi_gpio_2	S_AXI	Reg	0x00_A002_2000	4K	0x00_A002_2FFF
SYS_MGMT/axi_gpio_3	S_AXI	Reg	0x00_A002_5000	4K	0x00_A002_5FFF
Low_Speed_MEZZ/axi_uart16550_0	S_AXI	Reg	0x00_A000_0000	64K	0x00_A000_FFFF
Low_Speed_MEZZ/axi_uart16550_1	S_AXI	Reg	0x00_A001_0000	64K	0x00_A001_FFFF
SYS_MGMT/system_management_wiz_0	S_AXI_LITE	Reg	0x00_A002_6000	8K	0x00_A002_7FFF
Capture_Timer_0	S00_AXI	S00_AXI_reg	0x00_A003_0000	64K	0x00_A003_FFFF
zynq_ultra_ps_e_0					
Low_Speed_MEZZ/PWM_w_Int_2	S00_AXI	S00_AXI_reg	0x00_A002_3000	4K	0x00_A002_3FFF
Low_Speed_MEZZ/PWM_w_Int_3	S00_AXI	S00_AXI_reg	0x00_A002_4000	4K	0x00_A002_4FFF
BRAM_BLOCK/axi_bram_ctrl_0	S_AXI	Mem0	0x00_A002_8000	8K	0x00_A002_9FFF
axi_cdma_0	S_AXI_LITE	Reg	0x00_B000_0000	4K	0x00_B000_0FFF
BD_CTL_GPIO/axi_gpio_0	S_AXI	Reg	0x00_A002_0000	4K	0x00_A002_0FFF
BD_CTL_GPIO/axi_gpio_1	S_AXI	Reg	0x00_A002_1000	4K	0x00_A002_1FFF
Low_Speed_MEZZ/axi_gpio_2	S_AXI	Reg	0x00_A002_2000	4K	0x00_A002_2FFF
SYS_MGMT/axi_gpio_3	S_AXI	Reg	0x00_A002_5000	4K	0x00_A002_5FFF
Low_Speed_MEZZ/axi_uart16550_0	S_AXI	Reg	0x00_A000_0000	64K	0x00_A000_FFFF
Low_Speed_MEZZ/axi_uart16550_1	S_AXI	Reg	0x00_A001_0000	64K	0x00_A001_FFFF
SYS_MGMT/system_management_wiz_0	S_AXI_LITE	Reg	0x00_A002_6000	8K	0x00_A002_7FFF
Capture_Timer_0	**S00_AXI**	**S00_AXI_reg**	**0x00_A003_0000**	**4K**	**0x00_A003_0FFF**
Excluded Address Segments (1)					
BRAM_BLOCK/axi_bram_ctrl_1	S_AXI	Mem0	0x00_B002_8000	8K	0x00_B002_9FFF

Note that the `Capture Timer` can only be accessed by the PS. Make sure to exclude it from the CDMA unit.

The system.dtb for this lab is located:
http://projects.ece.utexas.edu/courses/spring_22/ee382n-16775/arch/labs/SP22_LAB_2/

CDMA Data Transfer Measurement Procedure:

This program will use the code that was developed in LAB_1 as a baseline. The code needs to be converted from polling to interrupt driven. The program will also use a "fork-join" construct to spawn DMA transactions. The child process starts the DMA transaction. In this application the parent process waits until the DMA is complete. In other applications the parent process can continue executing

Here is the basic code flow for the fork-join construct that can be used in this lab.

```
// ----------------------------------------------------------------------------------
// This for loop performs "lp_cnt" # DMA Transfers with "data_cnt" words transferred
// ----------------------------------------------------------------------------------
for(cnt = 0; cnt < lp_cnt; cnt++)
{
    // ------------------------------------------------------------------------
    // Fork off a child process to set up the DMA and start the transfer
    // ------------------------------------------------------------------------
    childpid = vfork();
    if (childpid >=0)    // Fork succeeded
    {
      // --------------------------------------------------------------------
      // This code runs in the child process as the childpid == 0
      // --------------------------------------------------------------------

       if (childpid == 0)
       {
           pm(CDMA+DA, BRAM1);                 // Write destination address
           pm(CDMA+SA, OCM);                   // Write source address
           pm(CDMA+CDMACR, 0x1000);            // Enable interrupts
           pm(CDMA+BTT, data_cnt*4);           // Start transfer
           exit(0);                            // Exit the child process
       }
      // --------------------------------------------------------------------
      // This code runs in the parent process as the childpid != 0
      // --------------------------------------------------------------------
      else
       {
           // ------------------------------------------------------------------
           // Wait for child process to terminate before checking
           // for the interrupt
           // ------------------------------------------------------------------
           waitpid(childpid, &status, WCONTINUED);

           while (!det_int);
           det_int = 0;                        // Clear interrupt detected flag
           pm(CDMA+CDMACR, 0x0000);            // Disable interrupts

           // ------------------------------------------------------------------
           // Check to make sure transfer was completed correctly
           // ------------------------------------------------------------------

           for(int i=0; i < data_cnt; i++)
           {
```

```
                    if(BRAM0_VA[i] != ocm[i])
                    {
                        printf("test failed!!\n");
                        goto exit_test;
                    }
                }

            } // if childpid ==0
        } // if childpid >=0

        else  // fork failed
        {
            perror("Fork failed");
            exit(0);

        } // end if childpid >=0

    }  //  end for loop
```

The ***CAPTURE_TIMER*** must be configured in capture mode. The timer is enabled when the DMA transfer starts. The timer is halted when the `cdma_introut` signal from the **axi_cdma** module is asserted. The value in the `cap_timer_out[31:0]` register contains the number of `axi_aclk` clock cycles that DMA transfer took. The `cdma_introut` signal is connected to the GIC interrupt pin on the PS. The interrupt handler will acknowledge the interrupt and set the `det_int` flag for the test program.

The test program will report the following information:

```
Test status  ---  # loops and # words transferred
Minimum Latency
Maximum Latency
Average Latency
Standard Deviation
```

Here is an example output of the CDMA transfer latency showing debug information

```
***************************************************************
Memory test passed --- 500 loops and 1000 words
Minimum DMA Latency: 2689
Maximum DMA Latency: 9067
Average DMA Latency: 3166.000000
Standard Deviation: 372.000000

Number of samples: 500
Number of interrupts detected = 500
Interrupt #53: 9400000 GICv2 125 Edge cdma-controller
***************************************************************
Memory test passed --- 500 loops and 1000 words
Minimum DMA Latency: 2653
Maximum DMA Latency: 6765
Average DMA Latency: 3224.000000
Standard Deviation: 433.000000
```

```
Number of samples: 500
Number of interrupts detected = 500
Interrupt 53: 9400500 GICv2 125 Edge cdma-controller
**************************************************************
```

These values are for the kernel when it is not busy doing other tasks. You will need to open additional terminal windows and start other tasks to see what the impact is on the "Maximum Latency". An interesting task is to do a continuous recursive directory listing of the flash drive to stress the OS and the AMBA bus:

```
while (cd /) do ls -algR; done
```

The maximum latency and standard deviation will both increase:

```
**************************************************************
Memory test passed --- 500 loops and 1000 words
Minimum DMA Latency: 2578
Maximum DMA Latency: 1312215
Average DMA Latency: 7919.000000
Standard Deviation: 61545.000000

Number of samples: 500
Number of interrupts detected = 500
53: 9424119 GICv2 125 Edge cdma-controller
**************************************************************
Memory test passed --- 500 loops and 1000 words
Minimum DMA Latency: 2663
Maximum DMA Latency: 170834
Average DMA Latency: 4496.000000
Standard Deviation: 8762.000000

Number of samples: 500
Number of interrupts detected = 500
53: 9424619 GICv2 125 Edge cdma-controller
**************************************************************
```

Notice that the maximum latency has increased 194 times.

Interrupt Latency Measurement Procedure:

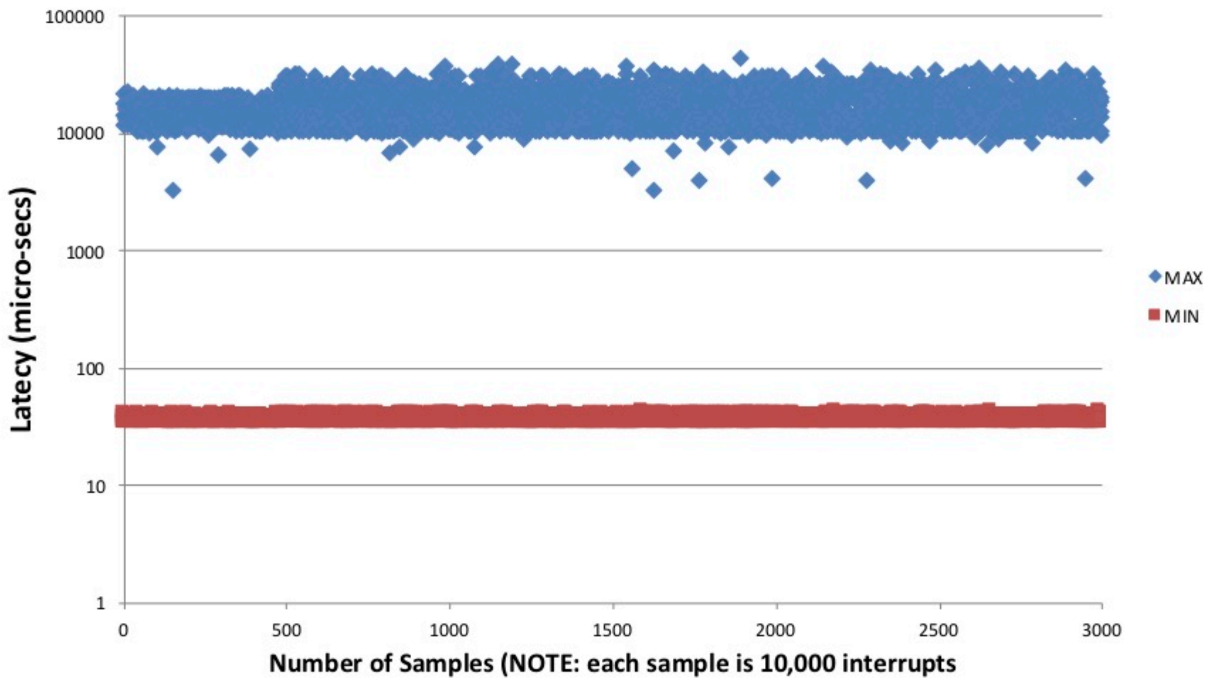
1) Assert the "`interrupt_out`" pin (`slv_reg1[0])` on the Capture Timer module. This pin is connected to the GIC interrupt input on the PS.
2) At the same time assert the `slv_reg1[1]` pin. This will start the timer counter.
3) Wait for interrupt to be detected and handled in the kernel. Read the timer counter value.
4) Negate the interrupt and disable the timer counter.
5) The value in the timer is read by the application program and the following data needs to be displayed:

```
Minimum Latency
Maximum Latency
Average Latency
Standard Deviation
Number of Samples
Number of Interrupts registered in the Kernel using /proc/interrupts
```

6) Here is an example output showing some debug information:

```
***************************************************************
Minimum Latency: 17
Maximum Latency: 2893
Average Latency: 25.540000
Standard Deviation: 49.710000
Number of samples: 10000
48: 433717
***************************************************************
Minimum Latency: 17
Maximum Latency: 1071
Average Latency: 27.470000
Standard Deviation: 11.150000
Number of samples: 10000
48: 443717
***************************************************************
Minimum Latency: 20
Maximum Latency: 335
Average Latency: 27.440000
Standard Deviation: 4.960000
Number of samples: 10000
48: 453717
***************************************************************
```

7) The number of samples per test needs to be programmable. The maximum number of samples is approximately 10,000. Here is what you should see if you ran around 30 million interrupts:



Lab procedure:

In this Lab you will reuse the code from Lab #1. The CDMA code needs to be converted from polling to interrupt driven. You will leave the frequency dithering code in place. Use the 9 combinations shown in the table to the left:

PS (CPU) Clock Freq	PL (FPGA) Clock Freq
1499 MHz	300 MHz
999 MHz	187.5 MHz
416.6 MHz	100 MHz

The capture timer block will be used to measure DMA transfer time and interrupt latency time using a counter clocked by the PL clock. Theoretically the number of clock cycle should be close to the same independent of the PL frequency. Minor differences may be caused by the cross-clock domain logic in the AMBA bus.

You will need to write two kernel modules for this lab. The first kernel module will handle the interrupts for the CDMA unit and second will handle the interrupts for the Capture-Timer unit.

TEST #1: Refer to CDMA Transfer Measurement Procedure outlined above for this test. The memory will be tested using the following sequence:

1. Load a memory page (i.e., 4K bytes) in the OCM with a 2048 random 32 bit data values using the Linux `srand(time(0))` and `rand()` routines. Use 0xFFFC_0000 as the starting address of this page.
2. Transfer the random data in the OCM (0xFFFC_0000) to the BRAM (0xA002_8000) using the CDMA unit.
3. Measure the number of cycles it took to do CDMA transfer using the Capture-Timer unit as described above.
4. Transfer the random data in the BRAM (0xA002_8000) back to the OCM at a different address (0xFFFC_2000) using the CDMA unit.
5. Measure the number of cycles it took to do CDMA transfer using the Capture-Timer unit as described above.
6. Once the steps above are complete, the OCM data at address 0xFFFC_0000 is compared to the OCM data at address 0xFFFC_2000. This confirms that DMA traffic to/from the OCM & BRAM works.
7. Change the PS and PL clock frequencies and repeat until all 9 combinations have been tested.

TEST #2: Refer to the Interrupt Latency Measurement Procedure outlined above to measure interrupt latency in the kernel.

1. Measure the interrupt latency for a statically large number of interrupts using all 9 frequency combinations.
2. Generate a plot similar to what is shown above.

Do NOT use the Vivado SDK development tools. They are for bare-metal implementations. We will not be doing any bare-metal implementations in this class.

General Tutorials:

Setting up Baseline Ultra96 Xilinx Environment
BIT File Generation Flow

Ultra-96 Documentation:

Setting up Ultra-96 Board
Getting Started
Ultra-96 HW User Guide
Ultra-96 Base TRD
Ultra-96 Building the Base TRD
Ultra-96 Schematic
Ultra96 Assembly Drawings

Xilinx Zynq UltraScale+ Tutorials and Documentation

ZYNQ UltraScale+ Register Map
ZYNQ UltraScale+ MPSoC Base Targeted Reference Design
Zynq UltraScale+ All Programmable SoC Technical Reference Manual
Repository of useful Vivado, Zynq & Petalinux Documentation

DMA Documentation & Tutorials

Xilinx AXI CDMA Manual
Xilinx AXI Timer Manual
Xilinx Wiki page on DMA
Using the AXI DMA in Vivado