Spring 2018

EE 382N-4: Advanced Micro-Controller Systems

Lab Assignment #2

Due March 28th, 2018

Overview:

This lab measures the interrupt latency through the Linux kernel running on the Zed Board. This will require the development of two programs. One is the application level program that measures the latency and the second is the kernel module that is invoked when the interrupt occurs. You can find an example of both programs at: FPGA_INTERRUPTS   These programs were designed to work with Linux 2.6 and need to be updated to the new 4.6 interrupt mechanism in the Zinq-7000 (covered in Lecture 10).

You can find an example of how to measure interrupt latency at: example_measuring_interrupt_latency

Additionally if you are interested in how kernel modules work, you can find an example at: KERNEL_MODULE_TEST

Schematic Generation Procedure:

1) Use the Vivado schematic from Lab 1 and remove the AXI_GPIO module.

2) Implement a new GPIO module using an AXI_SLAVE. This involves creating a new IP block similar to what was done with the LFSR. The block will contain 16 registers. Add inputs and outputs as shown in the block diagram below.

3) Modify the Processor System (PS) to add a PL-PS interrupt input.

Interrupt Latency Measurement Procedure:

1) The "interrupt_out" pin on the "int_latency" module is driven by the slv_reg0[0] port. This pin is connected to the interrupt input on the PS. The application software asserts and negates the pin. The time from when the "interrupt_out" pin is asserted until when it is negated is measured by the application program. The application program will use the techniques in example_measuring_interrupt_latency to determine the time the Linux kernel handles the interrupt.

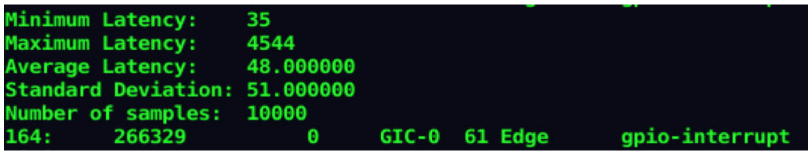
2) The LEDS[7:0] output port will be driven by slv_reg1[7:0]. Each time the "interrupt_out" pin on the "int_latency" module is ***asserted*** the LEDS[0] output pin is ***asserted***. Each time the "interrupt_out" pin on the "int_latency" module is ***negated*** the LEDS[0] output pin is ***negated***. The user can hook a scope to the LEDS[0] pin to measure the amount of jitter. The remaining "LEDS[7:1]" pins can be used for diagnostics.

3) The number of samples per loop needs to be programmable. Additionally the SWS[0] switch input should be used to cause an interrupt to be registered in the kernel. This switch does NOT need to be debounced. The user should toggle this switch a number of times to see if additional spurious interrupts are detected by the kernel.

4) The application program reports the following:

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

Use the example code from: example_measuring_interrupt_latency to determine how to structure the application program.
Here is an example output:



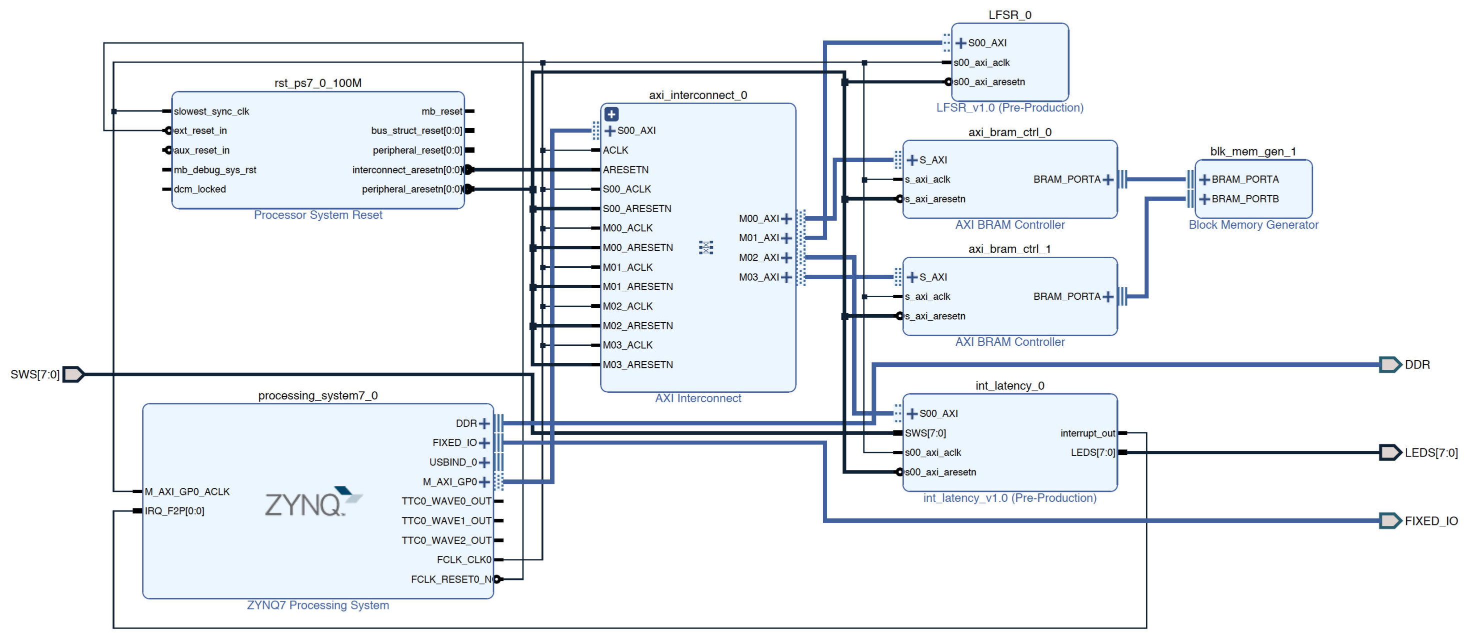
```
Minimum Latency:    35
Maximum Latency:    4544
Average Latency:    48.000000
Standard Deviation: 51.000000
Number of samples:  10000
164:     266329          0     GIC-0  61 Edge      gpio-interrupt
```

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:

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

Additionally the values from each loop need to be captured in a .CSV file for review.

Block Diagram:



Implementation Details

1) The new int_latency block will need to be implemented in Verilog. It will have 16 registers of which 3 will be used for this Lab.

2) Register port slv_reg0[0] will be used to assert or negate the "interrupt_out" port pin. Additionally SWS[0] can cause an interrupt.

```
assign interrupt_out = slv_reg0[0] | SWS[0];
```

3) Register port slv_reg1[7:0] will be used to drive the "LEDS[7:0]" port pins.

```
assign LEDS[0] = slv_reg0[0] | slv_reg1[0]; // Toggle LEDS[0] when interrupt_out is toggled
assign LEDS[7:1] = slv_reg1[7:1];
```

4) The "SWS[7:0]" inputs can be read by the software at reg2[7:0]. This is accomplished by directly connecting the SWS pins to reg_data_out as shown in this code snippet from the AXI_SLAVE

```
assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
always @(*)
begin
    // Address decoding for reading registers
    case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
       4'h0 : reg_data_out <= slv_reg0;
       4'h1 : reg_data_out <= slv_reg1;
       4'h2 : reg_data_out <= {24'h0,SWS[7:0]};   // Note: need to zero fill upper 24 bits
       4'h3 : reg_data_out <= slv_reg3;
       default : reg_data_out <= 0;
    endcase
end
```

5) Use the following address map for the peripherals:

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
LFSR_0	S00_AXI	S00_AXI_reg	0x43C0_0000	4K	0x43C0_0FFF
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_2000	8K	0x4000_3FFF
int_latency_0	S00_AXI	S00_AXI_reg	0x43C1_0000	4K	0x43C1_0FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF

Deliverables:

1) Download your bit file to the ZedBoard and demonstrate your code running on the ZedBoard.

2) Upload your C-code and Vivado directories to Canvas.

3) Write a README summarizing the design and testing process

Tutorials:

An FPGA Tutorial using the ZedBoard
Zynq Design from Scratch
Zynq Development
Free Electrons
Zynq Training

Environmental Details:

1) Vivado is available on the Linux machines. Refer to this web page ECE-Linux-Machines for details on accessing the Linux machines remotely. NOTE: VNC is no longer supported.

2) All software will be compiled on the ZedBoard using the GNU tool suite. **Do not use the Vivado SW development tools. These are for bare-metal implementations**.