How to Create and Program Interrupt-Based Systems

MicroBlaze-SoC + Interrupts

By Jason Agron

What is an Interrupt?

- Definition [Wikipedia]:
  - "… An interrupt is an asynchronous signal from hardware indicating the need for attention or a synchronous event in software indicating the need for a change in execution."
- They can be caused by…
  - Hardware signals.
    - Interrupt request lines.
  - Software invocations.
    - CPU instructions that purposefully "cause" an interrupt.

How Are Interrupts Handled?

- When an interrupt "goes off"…
  - First, interrupts are disabled.
    - You don't want to interrupt your interrupt handler.
  - CPU state must be saved.
    - So that you can return to the EXACT same program point.
  - Interrupt handler is invoked.
    - A program that "finds" which interrupts "fired".
    - Services each active interrupt.
    - "Clears" interrupts that have been serviced.
  - Return to normal processing…
    - CPU state is restored.
    - Interrupts are re-enabled.

Interrupt Lines

- A processor has a SET of interrupt lines.
  - They are often prioritized.
- Each line's behavior can be configured.
  - Sensitivity (Edge/Level).
  - Polarity (Active High/Low).
- Additional interrupt lines can be added by multiplexing.
  - Multiplex multiple lines onto a single request line.
  - Requires extra info find out which interrupt occurred.
  - Often accomplished by using an "interrupt controller".

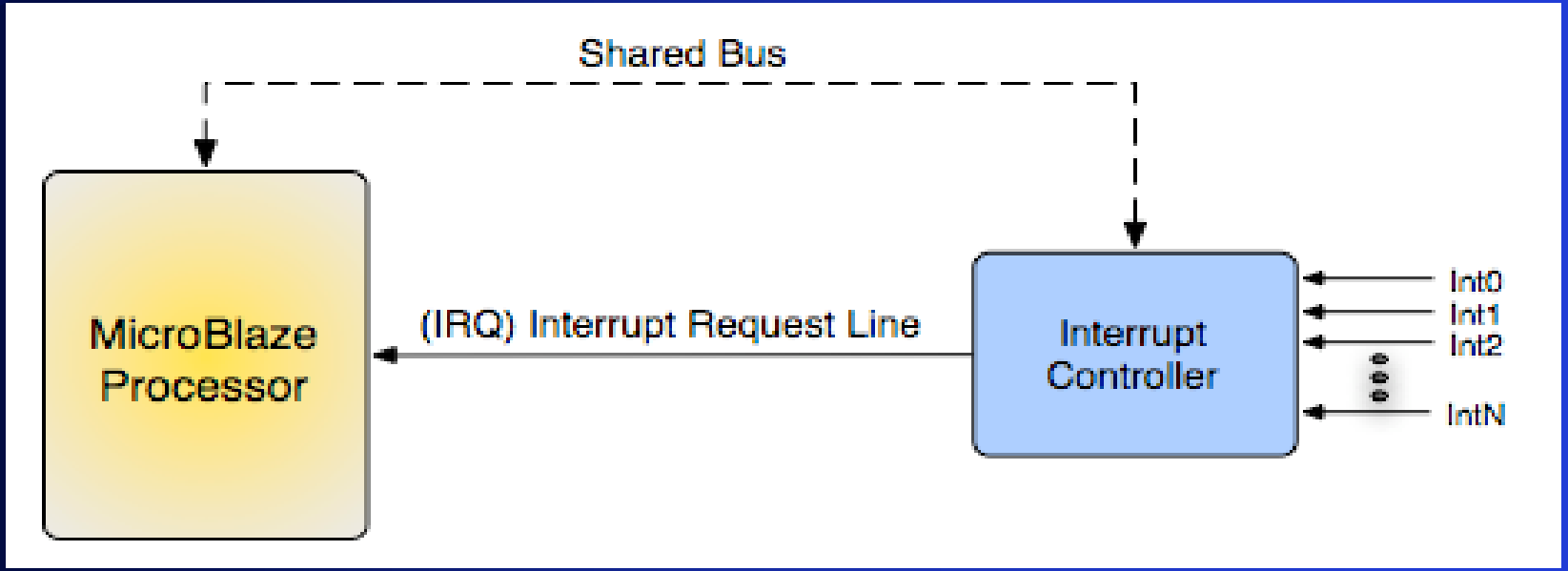
MicroBlaze Interrupt Semantics

- MB has a single interrupt request line.
  - Called the *interrupt* port.
- The processor only reacts to interrupts…
  - If the *interrupt enable* (IE) bit in the *machine status register* (MSR) is set to ‘1’.
- The processor will also ignore interrupts…
  - If the *break in progress* (BIP) bit in the *machine status register* (MSR) is set to ‘1’.

MicroBlaze Interrupt Semantics

- Interrupt Vector Address:
  - 0x00000010 - 0x00000014.
  - Branch target to get to interrupt handler.
  - The value stored here is usually a branch to the interrupt handler routine.
- Register File Return Address:
  - R14.
  - Storage of where to return after interrupt handler routine is complete.
  - The value stored here is the address of the instruction that was in the decode stage when interrupt occurred.

MicroBlaze Interrupt Configuration



- Multiple interrupt sources multiplexed into one IRQ.
- Interrupt controller holds state of interrupts.
  - Which are enabled (IER).
  - Which have fired (ISR).
  - Which are acknowledged (IAR).
- Interrupt state is accessed by CPU over the bus.

What Hardware Is Needed?

- A single interrupt line can be directly connected to the processor.
  - Done via port modifications of the MB in…
    - The .mhs file
    - The “System Assembly View” - Port Tab.
- If multiple interrupts are needed…
  - An interrupt controller is very useful
    - AKA “PIC”, Priority Interrupt Controller.
    - Allows for multiple interrupt lines to be connected.
  - Use the OPB_INTC (Xilinx IP Core).

Our Goal

- To create a interrupt-based system with at least 2 interrupt sources.
  - One source must be periodic.
  - One source must be external.
- A timer will be used to generate a periodic interrupt signal.
  - Use the OPB_TIMER (Xilinx IP Core).
- The push buttons or dip switches can be used to generate the external interrupt.

HW Design - IP Addition

- First, add in the OPB_TIMER and OPB_INTC peripherals.
  - Make sure to connect them to the OPB bus.
- Now, the system is interrupt-capable but…
  - Interrupt lines must still be routed.
    - Connect interrupt controller to the CPU.
    - Connect interrupt lines to the interrupt controller.
  - Software still needs to be implemented.
    - Functions to enable/handle interrupts.

HW Design - IRQ Connection (1)

- GOAL: Connect the interrupt controller's *IRQ* line to the CPU's *interrupt* port.
- Open up your system's .mhs file.
- First, connect the MB's *interrupt* port to a signal called *myIRQ*.

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER HW_VER = 5.00.c
 PARAMETER C_USE_FPU = 0
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_NUMBER_OF_PC_BRK = 2
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 BUS_INTERFACE DOPB = mb_opb
 BUS_INTERFACE IOPB = mb_opb
 PORT DBG_CAPTURE = DBG_CAPTURE_s
 PORT DBG_CLK = DBG_CLK_s
 PORT DBG_REG_EN = DBG_REG_EN_s
 PORT DBG_TDI = DBG_TDI_s
 PORT DBG_TDO = DBG_TDO_s
 PORT DBG_UPDATE = DBG_UPDATE_s
 PORT Interrupt = myIRQ
END
```

HW Design - IRQ Connection (2)

```
BEGIN opb_intc
 PARAMETER INSTANCE = opb_intc_0
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0x80200200
 PARAMETER C_HIGHADDR = 0x802002ff
 BUS_INTERFACE SOPB = mb_opb
 PORT Irq = myIRQ
 PORT Intr = periodic_interrupt & external_interrupt
END
```

- Second, connect the OPB_INTC's *IRQ port* to the *myIRQ* signal.
- This completes the connection between CPU and interrupt controller.

HW Design - Interrupt Lines (1)

```
BEGIN opb_timer
 PARAMETER INSTANCE = opb_timer_1
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_COUNT_WIDTH = 32
 PARAMETER C_ONE_TIMER_ONLY = 0
 PARAMETER C_BASEADDR = 0x80200000
 PARAMETER C_HIGHADDR = 0x802000ff
 BUS_INTERFACE SOPB = mb_opb
 PORT OPB_Clk = sys_clk_s
 PORT Interrupt = periodic_interrupt
END
```

- GOAL: Connect interrupt lines to interrupt controller.
- First, connect the OPB_TIMER's *Interrupt* line to a signal called *periodic_interrupt*.

HW Design - Interrupt Lines (2)

```
BEGIN opb_intc
 PARAMETER INSTANCE = opb_intc_0
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0x80200200
 PARAMETER C_HIGHADDR = 0x802002ff
 BUS_INTERFACE SOPB = mb_opb
 PORT Irq = myIRQ
 PORT Intr = periodic_interrupt
END
```

- Now, connect the *periodic_interrupt* line to the *Intr* port of the OPB_INTC.
- This completes the connection of the periodic interrupt.

HW Design - Interrupt Lines (3)

```
PORT external_interrupt = external_interrupt, DIR = I, SIGIS = INTERRUPT, SENSITIVITY = EDGE_RISING
```

- Now, we must connect an external interrupt line to the interrupt controller.
- First, start by defining an external interrupt port named *external_interrupt*.
  - It must be declared as an interrupt line (SIGIS = INTERRUPT).
  - It must be declared as an input (DIR = I).
  - It's sensitivity must be set (SENSITIVITY = EDGE_RISING).

HW Design - Interrupt Lines (4)

```
BEGIN opb_intc
 PARAMETER INSTANCE = opb_intc_0
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0x80200200
 PARAMETER C_HIGHADDR = 0x802002ff
 BUS_INTERFACE SOPB = mb_opb
 PORT Irq = myIRQ
 PORT Intr = periodic_interrupt & external_interrupt
END
```

- Now, we must connect the newly defined *external_interrupt* port to the *Intr* port on the interrupt controller.
- HOLD ON!!! We already have an interrupt signal connected to the *Intr* port.
  - True, but many signals can be connected.
  - This is done via signal concatenation (the '&' operator).
  - This signal is now a vector of signals with priorities.
    - LSB is highest priority, MSB is lowest priority.

HW Design - Interrupt Lines (5)

```
#### Module DIPSWs_4Bit constraints

Net external_interrupt LOC = AC11;
Net external_interrupt IOSTANDARD = LVCMOS25;
```

- One last step…
  - The newly defined *external_interrupt* port needs to be connected to an external button or switch.
- This is specified in the .ucf file.
  - Ports are associated with external pins on the FPGA.
- Within the .ucf file (located in <yourProject>/data/)
  - Associate the *external_interrupt* port with a location (LOC) and an IO voltage standard.

Important Note: LOC Constraints

- When modifying the .ucf file…
  - Make sure that only a single "net" (or signal) is connected to a certain location.
  - If this rule is violated, then synthesis will fail.
    - The toolset is not capable of splitting I/O pads automatically.
- EXAMPLE:
  - If using the dip switches as inputs.
  - First, remove the OPB_GPIO core for the dip switches.
    - Comment out or delete all .mhs file entries for the IP core.
    - Comment out or delete all .ucf file entries for the IP core.
  - This effectively "frees" the FPGA's input pins at these locations.

HW Design - Complete

- Now, your hardware platform is fully setup for interrupts.
  - Interrupt lines are connected to the interrupt controller.
  - Interrupt controller's IRQ line is connected to the CPU.
- Additional parameters for the OPB_TIMER and OPB_INTC can be configured from within XPS if needed.

SW Design

- Now, the system just needs an application program to setup/handle interrupts.
- This requires:
  - An interrupt handler to be written.
  - The interrupt handler must be registered.
  - Interrupts must be enabled on the system.
    - Must be enabled on the processor.
    - Must be enabled on the interrupt controller.
    - Must be enabled on any interrupt-based peripherals.
      - HINT - OPB TIMER.

Enabling/Disabling Interrupts

- Documentation located in XAPP778.
- *void microblaze_enable_interrupts(void)*
  - "This function enables interrupts on the MicroBlaze system. When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on using this function."
- *void microblaze_disable_interrupts(void)*
  - "This function disables interrupts on the MicroBlaze system. This function may be called when entering a critical section of code where a context switch is undesirable."

Registering Handlers

- Documentation located in XAPP778.
- *void microblaze_register_handler(*
  XInterruptHandler Handler,
  *void *DataPtr)*
  - *"This function allows one to register the interrupt handler for the interrupt on the MicroBlaze processor. This handler will be invoked by the first level interrupt handler that is present in the BSP. The first level interrupt handler takes care of saving and restoring registers, as necessary for the interrupt handling and hence the function that you register with this handler can concentrate on the other aspects of the interrupt handling, without worrying about saving registers."*

Main Program: Steps

- Register interrupt handler routine.
  - Places the correct branch instruction in memory location 0x10 to branch to your interrupt handling routine upon an interrupt.
- Initialize interrupt-based components.
  - Peripherals that generate interrupts (i.e. timers).
  - Interrupt controller (PIC).
- Enable interrupts on processor.
  - Now interrupts are allowed to occur.
  - Main program can be interrupted by interrupt handler.

Interrupt Handlers: Steps

- Figure out which interrupts have occurred.
  - Read in ISR (interrupt service vector).
- Service interrupts that have occurred.
  - Inspect ISR “one bit at a time”.
- Clear interrupts that have occurred.
  - Write to IAR (interrupt acknowledge vector).
    - Clears interrupt at the PIC.
  - Clear interrupt at the source if needed.
    - When does this need to be done (HINT: level triggered?).
- Voila!
  - Interrupt service is complete, just return to main program.

Example: Main Program

```
// Main program
int main()
{
  // Register an interrupt handler
  microblaze_register_handler((XInterruptHandler)myIntHandler,(void *)0);

  // Enable timer and interrupt generation
  *timerLoad = timerPeriod;              // Set time of period
  *timerControl = timerEnableMask;       // Enable timer

  // Enable interrupts on PIC
  *ier = ierEnableTimerMask;             // Enable timer interrupt
  *mer = merInitMask;                    // Setup MER

  // Enable interrupts on MB
  microblaze_enable_interrupts();

  // Infinitely loop...
  while(1)
    {
      xil_printf("**** x = %d\r\n",x);
    }

  return 0;
}
```

Example: Interrupt Handler

```
// Interrupt handler
//      * Spawned on an interrupt
//      * Performs service routine(s)
//      * Clears serviced interrupts
void myIntHandler(void)
{
  int curISR;

  // Check which interrupt(s) occurred and service them
  curISR = *isr;

  // Check Int1 (Periodic Timer Interrupt)
  if (curISR & ierEnableTimerMask)
    {
      // Do something
      x+=2;
      xil_printf("\tTimer Interrupt!!! %d\r\n",*isr);

      // Clear timer interrupt at timer
      *timerControl = timerClearMask;

      // Clear timer interrupt at PIC
      *iar = iarClearTimerMask;
    }
}
```

Important Registers: PIC

- OPB_INTC
  - ISR - Interrupt Service Register.
    - Offset (0x0)
    - Read/write register used to show which active interrupts are pending (which have “fired”).
  - IER - Interrupt Enable Register.
    - Offset (0x8)
    - Read/write register used to enable selected interrupts.
  - IAR - Interrupt Acknowledge Register.
    - Offset (0xC)
    - Write-only location used to clear interrupt requests.
  - MER - Master Enable Register.
    - Offset (0x1C)
    - Read/write register used to enable the IRQ output of the PIC.

Important Registers: Timer

- OPB_TIMER
  - TCSR0 - Timer Control/Status Register 0.
    - Offset (0x0)
    - Timer mode control.
      - Timer enables, interrupt setup, etc.
  - TLR0 - Timer Load Register 0.
    - Offset (0x4)
    - Timer re-load value (period).
  - TCR0 - Timer Counter Register 0.
    - Offset (0x8)
    - Timer counter value (the timer's "clock").