R·I·T

Input/Output Programming

Week 5 Lecture 1

Reminders

- IO Devices
- Lab 2 is due this week

- Quiz due Sunday
  - **IO Devices, Custom Components**

- Demo due Friday
  - **Custom IP Demo**

- Exam 1 next Wednesday!!!
  - **In person during lecture**

I/O Devices

- What are they?
  - **Provide information to the processor or receive information from the processor**
    - I/O devices can be subdivided into input & output
    - Ex: Keyboard, display, switches, LEDs
- I/O Interfaces
  - **Enable processor to receive or transmit information**
  - **Provide a mechanism for communication**
    - Transmission medium may be wired or wireless
    - Communication interface may be serial or parallel
    - Control, status, and data can be exchanged
- We will use two types of I/O devices
  - **External: devices not included on FPGA**
  - **Internal: IP cores that interface to NIOS II on the FPGA**

External I/O Devices

- External I/O devices in embedded systems are often mechanical
  - **Solenoids, relays, etc.**
  - **Their response time is often characterized by the time required for some physical movement**
    - Orders of magnitude slower than a typical processor
- Other I/O devices are analog-to-digital converters or digital-to-analog converters
  - **Commonly translate a voltage to/from a digital value**
  - **Still slower than typical processor**
- What's the point?
  - **The typical I/O data transfer rate is limited by the speed of the external device, not the processor**
  - **A programmer must be certain a processor does not attempt to transfer data at a rate faster than the device can process it.**

External I/O Devices

- External I/O devices operate asynchronously with respect to the processor
- Most I/O programming requires some “handshaking” mechanism between the device and the processor to coordinate reliable data transfer
- The time behavior of I/O data transfer rates vary significantly from one device to another
  - **From low data rate, random occurrence (keyboard)**
  - **To high data rate, periodic occurrence (DMA device)**

Time Behavior in I/O Data Transfer

Data Rate	Occurrence	Increasing Time →
Low	Random	* * * * * * * * * *
	Periodic	* * * * * * * * * * * * * *
High	Random	******** ****** ********** ********** ************
	Periodic	****** ****** ****** ****** ****** ****** ****** ****** ******

I/O Devices

- Accessing I/O Devices
  - **I/O mapping provides dedicated memory map for I/O**
    - Requires special instructions for access I/O devices
  - **Memory mapping uses same memory map used for memory**
    - Allows normal instructions to access I/O devices
    - I/O devices treated like memory
    - Each I/O device has it's own address

Accessing memory-mapped I/O Devices

- Accessing I/O Devices with C
  - **Use C pointers to access device**
    - Create pointer and assign pointer the address of I/O device
    - Pointer type should match size of device data
  - **Ex: Assume we need to access a 16-bit I/O device**

```
#define BASE_ADDRESS  0x12345678
uint16* my_device = (uint16*)BASE_ADDRESS
```

    - my_device is pointer to uint16 piece of data
    - Pointer is initialized to base address of the device
      - ***BASE_ADDRESS is a symbol constant for address***
      - ***Use type casting (uint16 *) to tell compiler how to handle BASE_ADDRESS***
    - Use #define to define base address
      - ***Nios II SBT automatically creates system.h with #defines***

Accessing Memory-mapped I/O Devices

- Reading from a device in C
  - **Create pointer**
  - **Use VOLATILE Type Qualifiers**
    - Tells compiler data may change outside scope of the compiler
  - **Ex: Create pointer to 16-bit I/O device**

```
#define BASE_ADDRESS  0x12345678
volatile uint16* dev = (uint16*)BASE_ADDRESS

// read device and store in variable
data = *dev;
```

Accessing Memory-Mapped I/O Devices

- Writing to Device in C
  - **Create pointer**
  - **VOLATILE Type Qualifiers may not be needed**
    - Remember if the data can change w/o compiler knowing then use VOLATILE type qualifier
  - **Ex: Create pointer to 16-bit I/O device**

```
#define BASE_ADDRESS  0x12345678
uint16* dev = (uint16*)BASE_ADDRESS

// write data to device
*dev = data;
```

Accessing Memory-Mapped I/O Devices

- Accessing Registers in I/O Device in C
  - **Suppose you have an I/O device with 3 16-bit registers and you want to write the 2nd register**
  - **Ex:**

```
#define BASE_ADDRESS  0x12340000
#define REG2_OFFSET   1
uint16* dev = (uint16 *)BASE_ADDRESS

// write data to device
*(dev + REG2_OFFSET) = data;
```

    - Since the pointer is 16-bits, the pointer address will be incremented by one pointer unit or 2 bytes in this case
      - ***Address of dev is 0x12340000***
      - ***Address (dev + 1) of is 0x12340002***

Accessing Memory-Mapped I/O Devices

- When using IP with the NIOS II in QSYS (Platform Designer), it is important to look at the data sheet so you know the width of the registers internal to the IP
  - **Google PIO Core NIOSII – Use most recent version**
  - **Example: PIO Register map**

23.5.2. Register Map

An Avalon-MM master peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown below. The table assumes that the PIO core's I/O ports are configured to a width of n bits.

Table 214. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
								continued...
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1), (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set. Outset value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				
5	outclear		W	Specifies which output bit to clear. Outclear value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				

Accessing Memory-Mapped I/O devices

- How does a device get the processors attention?
  - **Processor ask device if it requires servicing**
    - Called Programmed Controlled I/O or Polling
  - **Device tells processor that it requires servicing**
    - Called Interrupt Controlled I/O

Polling

- Programmed Control I/O Devices
  - **Program directly controls I/O device access**
    - I/O is handle under control of program
  - **Polling is the process of repetitively checking device status**

```
do forever
  if (device1 needs service)
    service device1
```

  - **Advantages**
    - Simple to implement
    - All control is under program control
    - Order of devices are serviced/controlled by program
      - ***Program sets priority***
  - **Disadvantage**
    - Wasted time checking status of devices that don't need servicing

Interrupts

- Interrupt Control I/O Devices
  - **Device has ability to alert processor when servicing is needed**
    - Device has signal to request processor to service device
    - Request Signal must be connected pin that causes interrupt
      - ***dedicated interrupt pin (INT)***
      - ***generic GPIO pins can generate interrupt***
      - ***Internal on-chip peripherals***
  - **Interrupt Service Routine (ISR)**
    - Executed when device asserts interrupt request
    - ISR is a function that checks status of device
    - Only performs the minimum amount of servicing of the device

I/O Devices

- Interrupt Control I/O Devices
  - **Steps of handling interrupt control I/O device**
    - Device asserts interrupt request
    - Processor suspends current instruction
    - Processor saves current context to stack
    - Interrupt disabled
      - ***May only disable lower priority interrupts***
    - Processor jumps to ISR to perform required device service
    - ISR clears interrupt condition in I/O device
      - ***Programmer decides if this is the first or last step***
    - Processor restores previous context and continues where it left off

I/O Devices

- Interrupt Service Routine (ISR)
  - **Also known as interrupt handler**
  - **Similar to a subroutine call**
  - **Should be kept short**
    - Interrupt may be disabled or blocking other interrupts
      - ***Prevents real-time response***
    - Performs minimal (critical) processing
      - ***Set flag or schedule task to finish work***
  - **Each ISR handles one specific device**
  - **First thing done is verify the source of the interrupt**
    - Are we in this ISR for correct reason?
  - **The last step should be to clear the interrupt event**
    - Allows device to issue another interrupt
    - If done first, then another interrupt could interrupt the ISR