1. Hardware design of the DE2-115 Board
   1. Create a Quartus II project - new project(Top level design)
   2. and start a new design - **.bdf**
   3. Platform Designer and Nios II system - Nios II processor instantiated - **.qsys**
      1. Add Nios II processor and all componenets
      2. Renaming and Connecting components
      3. Resolve conflicts of base addresses - addresses and IRQ num of each components do not conflict each other
      4. Checking the module names
      5. More about Nios II processor settings and generation of the system
      6. **Generating the CPU (**Nios II sys) **- .bsf**
      7. Adding the Nios II processor**(add “.qip” - Quartus II IP File)** to Quartus
   4. Integrating the Nios II system with other hardware components - create inputs and outputs on the top level design and interconnections between the nios2\_system and its environment. - **pin\_assignments --- Start Compilation**
   5. Download the lab1 digital design to the FPGA chip(**USB Blaster)**
2. Software implementation on the DE2 with dedicated hardware design
   1. Developing software with Nios II IDE - **Eclipse (Quartus Prime 18.1)**
   2. Construct an application

* software created based on the same hardware configurations are stored in the same workspace
* To create an application, “**File->New->Nios II Application and BSP from Template”**
* For “**SOPC Information File Name**” click and browse to “nios2\_system.sopcinfo
* \_bsp” contains information about the hardware
  1. Access modules added in the SOPC **Builder -** building bsp library

Lab 02

**Avalon Bus and Components:**

* Platform Designer peripherals/components are compatible with the Avalon bus.
* Avalon bus interconnects components with arbitrators and multiplexers.
* Components are categorized as master or slave.
* Nios II processor is an example of a master component.
* LEDs, switches, push buttons, timers, and UART are examples of slave components.
* Masters can send and receive commands, while slaves respond to commands.

**Registers in Slave Components:**

* Slaves often have internal registers for communication, configuration, coefficients, and debugging.
* Configuration registers control a slave device's behavior.
* Coefficient registers store frequently accessed data.
* Registers aid in debugging by storing messages and internal status.

**Memory Mapping and Access:**

* Registers in slave components are memory-mapped to the Nios II processor.
* Memory mappings use addresses specified in Platform Designer.
* Registers match the processor's data bus width (usually 32 bits).
* Each register occupies 4 bytes of memory space.
* First register uses the component's address; subsequent registers follow in 4-byte increments.

**Accessing Registers:**

* Use IORD and IOWR functions for reading and writing registers.
* These functions are macros that simplify memory-mapped register operations.
* Variants available for different data widths (8, 16, 32 bits).

**Example Code:**

cCopy code

#include <io.h> int main(void) { int value = 0; value = IORD(0x00001000, 0); // Read from register-1 of peripheral-1 IOWR(0x00002000, 1, value); // Write to register-2 of peripheral-2 }

**System.h and HAL APIs:**

* System.h contains important system and component information.
* HAL APIs provide a high-level interface for accessing memory-mapped components.
* Use system.h macros for portability across different hardware configurations.

**File-Based Peripherals:**

* File pointers can be used to interact with peripherals as files.
* fopen opens the peripheral as a file and returns a file pointer.
* Use fprintf to write data to the peripheral.
* Use fgetc to read data from the peripheral.
* File descriptors offer similar functionality with read() and write() methods.

**Button and Timer ISRs:**

* Requirements for writing ISRs include understanding interrupts and providing an ISR function.
* Register ISRs using alt\_irq\_register.
* Button ISR example provided for handling button presses.
* Timer ISR example demonstrates timer-based interrupts.

**Timer ISR and Polling:**

* Timer ISRs return the timeout value for the next iteration.
* Use alt\_alarm\_start to start a timer with an ISR.
* Polling is an alternative to interrupts, continuously checking peripheral values.

**Pointers:**

* Pointers store memory locations of other values.
* Dereferencing pointers retrieves the value stored at a memory address.

This cheat sheet summarizes the essential information related to memory-mapped registers, Platform Designer components, HAL APIs, ISRs, and pointers.

* 1. **Memory mapped registers and Platform Designer**
  2. **System.h and HAL APIs**

Access Component Registers: int button\_state = **IORD(KEYS\_BASE, 0);**

HAL APIs - use HAL APIs like **alt\_read\_word** and **alt\_write\_word:**

**int button\_state = alt\_read\_word(KEYS\_BASE);**

**alt\_write\_word(LED\_BASE, 0x01);**

System Configuration: nios2\_**system.sopcinfo** file contains system info and generates system.h.

* 1. **Using File Pointers**

1. **Create File Pointer**: - **fopen()**

* Syntax: **FILE\* fopen(const char\* filename, const char\* mode);**
* Example: **FILE\* uart\_file = fopen(UART\_NAME, "r+");**

1. **Modes for Opening Files**:

* "r": Read-only access.
* "w": Write-only access.
* "r+": Read and write access.

1. **Write to File Pointer**: - **fprintf()**.

* Syntax: **int fprintf(FILE\* stream, const char\* format, ...);**
* Example: **fprintf(uart\_file, "Hello, UART!");**

1. **Read from File Pointer**: Utilize standard C file-reading functions like **fgetc()**.  
   read a single character. Return nect char in receive buffer. if there's no pending character, it **will block** until the next character arrives.

* Syntax: **int fgetc(FILE\* stream);**
* Example: **int received\_char = fgetc(uart\_file);**

1. **Blocking for Input**: When using **fgetc()**, if there's no pending character, it will block until the next character arrives.
2. **Close File Pointer**: Don't forget to close the file pointer when done.

* Syntax: **int fclose(FILE\* stream);**
* Example: **fclose(uart\_file);**

1. **Error Handling**: Check the return values of functions for error handling and consider using try-catch blocks if available in your programming environment.
   1. **Using File Descriptors**

* **Access UART Registers**:
  + **Status Register**: **IORD\_ALTERA\_AVALON\_UART\_STATUS(base)** - Returns the status register. Use **ALTERA\_AVALON\_UART\_STATUS\_RRDY\_MASK** to check for waiting data.
  + **UART Read**: **IORD\_ALTERA\_AVALON\_UART\_RXDATA(base)** - Returns the current received value.
  + **UART Write**: **IOWR\_ALTERA\_AVALON\_UART\_TXDATA(base, data)** - Writes a value to the UART.
* **Interrupt Handling**:
  + **Create ISR**: You can create an Interrupt Service Routine (ISR) to handle incoming UART data.
  + **Register ISR**: Use **alt\_irq\_register()** to register the ISR for UART interrupts.
  + **Enable ISR**: Use **IOWR\_ALTERA\_AVALON\_PIO\_IRQ\_MASK()** to enable the UART interrupt handler.
  + **Clear Interrupts**: Use **IOWR\_ALTERA\_AVALON\_PIO\_EDGE\_CAP()** when necessary to clear interrupts.
  1. **Using Status Registers**
* Access UART registers in Nios II using associated helper functions with UART\_BASE as the argument.
* Status Register:
  + Use IORD\_ALTERA\_AVALON\_UART\_STATUS(base) to retrieve the status register.
  + Pre-defined mask for waiting data: ALTERA\_AVALON\_UART\_STATUS\_RRDY\_MASK.
* UART Read:
  + Fetch the current received value with IORD\_ALTERA\_AVALON\_UART\_RXDATA(base).
* UART Write:
  + Send data to UART using IOWR\_ALTERA\_AVALON\_UART\_TXDATA(base, data).
* Interrupt Service Routine (ISR):
  + Create an ISR for handling UART interrupts.
  + Register the ISR with alt\_irq\_register().
  + Enable the interrupt handler with IOWR\_ALTERA\_AVALON\_PIO\_IRQ\_MASK().
  + Clear interrupts using IOWR\_ALTERA\_AVALON\_PIO\_EDGE\_CAP().
  1. **Accessing other peripherals with HAL APIs**
     1. **Requirements of writing ISRs**
        1. IORD\_ALTERA\_AVALON\_PIO\_DATA(BASE\_NAME) **to read values.**
        2. IOWR\_ALTERA\_AVALON\_PIO\_DATA(BASE\_NAME, value) **to write values.**
        3. IORD\_ALTERA\_AVALON\_PIO\_IRQ\_MASK(BASE\_ADDR) **to get interrupt mask status**. - *identify which pins are set to generate interrupts and which are not*
        4. IOWR\_ALTERA\_AVALON\_PIO\_IRQ\_MASK(BASE\_ADDR, value) **to enable/disable interrupts by setting the interrupt mask for a PIO component..**
        5. IORD\_ALTERA\_AVALON\_PIO\_EDGE\_CAP(BASE\_NAME) **to check edge-capture status. -** ***Checks the edge-capture status of a specific PIO component to determine which inputs triggered an interrupt.***
        6. IOWR\_ALTERA\_AVALON\_PIO\_EDGE\_CAP(BASE\_NAME, value) **to write edge-capture status. *- clear interrupt flags and prepare for the next interrupt*.**

Requirements for Writing ISRs:

* Understand how to initialize and handle interrupts, which are assigned IRQ numbers.
* Provide an ISR function to be executed when an interrupt occurs.
* Within the ISR, perform necessary housekeeping like clearing interrupt flags. re-enable interrupt if required (sometimes the interrupt is disabled once the control enters the ISR).
* Enable the interrupt for the device generating it.
* Register the ISR so the system knows where to transfer control.
  + 1. **The body of the ISR**
* ISR function prototype: **void name\_of\_the\_isr\_function(void\* context, alt\_u32 ID)**.
  + Context can be any pointer passed to the ISR.
  + ID is the interrupt ID.
  + Housekeeping tasks and handling the interrupt logic are done in the ISR.
    1. **Registering the ISR**
* **int alt\_irq\_register ( alt\_u32 id, void\* context, void (\*isr)(void\* , alt\_u32))**
  + ID is the IRQ number.
  + Context is a pointer anything to data you want to pass.If Global vari, then NULL.
  + **function pointer -** ISR function is the ISR to execute. -- “**sys/alt\_irq.h**”
    1. **Timer ISR**
* Timer ISR prototype: **alt\_u32 timer\_isr\_function(void\* context)**.
* Timer ISR returns the next timeout value or 0 to stop the timer.
* timer ISR does not need to be registered,
* Include **"sys/alt\_alarm.h"** for timer functions.
* Timers are started using

**int alt\_alarm\_start( alt\_alarm\* timer\_pointer, alt\_u32 time\_to\_run,**

**alt\_u32 (\*function\_pointer\_to\_the\_timer\_isr) (void\* context),**

**void\* context);**

* The first argument is a **timer pointer** that specifies or points to a timer.
* The second argument - **time duration** before the timer times out (countdown timer).
* The third argument is a **function pointer** pointing to the Timer ISR (Interrupt Service Routine).
* The last argument is **the context**, a data pointer passed to the Timer ISR.
* If the timer fails to start, the function **returns a negative value**.
  1. **Programming technique**
     1. **Polling**
     2. **Pointers**
  2. **Exercise**
  3. **SCCharts**
     1. **Opening SCCharts**
     2. **Creating a Project**
     3. **ABRO**
     + HandleA
     + ABO
     + Complete ABRO
     1. **Generating Code**
  4. **Executing SCCharts on Nios II**
     1. **Implementation in Nios II Build Tools**
     2. Exercise

#include <system.h>

#include <altera\_avalon\_pio\_regs.h>

#include <alt\_types.h>

#include <sys/alt\_irq.h>

#include <unistd.h>

#include <stdio.h>

volatile int counter = 0;

volatile int key1\_pressed = 0; // Flag to track Key1 press and hold

// Function to increment the counter

void incrementCounter() {

counter++;

}

// Timer ISR

alt\_u32 timer\_isr\_function(void\* context) {

int\* timeCount = (int\*) context;

// Check if Key1 is pressed and held down

if (key1\_pressed) {

incrementCounter();

printf("Count: %i\n", counter);

}

return 500; // Set the timer to trigger every 500ms

}

// Button ISR

void key\_button\_interrupts(void\* context, alt\_u32 id) {

int\* temp = (int\*) context;

(\*temp) = IORD\_ALTERA\_AVALON\_PIO\_EDGE\_CAP(KEYS\_BASE);

IOWR\_ALTERA\_AVALON\_PIO\_EDGE\_CAP(KEYS\_BASE, 0);

if (!((\*temp) & 0x01)) {

// Key0 pressed

incrementCounter();

printf("Count: %i\n", counter);

}

if (!((\*temp) & 0x02)) {

// Key1 pressed

key1\_pressed = 1; // Set the Key1 press and hold flag

} else {

// Key1 released

key1\_pressed = 0; // Clear the flag

}

}

int main(void) {

alt\_alarm timer;

int timeCountMain = 0;

void\* timerContext = (void\*) &timeCountMain;

alt\_alarm\_start(&timer, 500, timer\_isr\_function, timerContext);

int counterValue = 0;

void\* context\_going\_to\_be\_passed = (void\*) &counterValue;

IOWR\_ALTERA\_AVALON\_PIO\_EDGE\_CAP(KEYS\_BASE, 0);

IOWR\_ALTERA\_AVALON\_PIO\_IRQ\_MASK(KEYS\_BASE, 0x3);

alt\_irq\_register(KEYS\_IRQ, context\_going\_to\_be\_passed, key\_button\_interrupts);

// // Function to send a command to the LCD

// void sendCommand(unsigned char command) {

// IOWR(LCD\_RS, 0, 0); // Set RS to 0 for command mode

// IOWR(LCD\_RW, 0, 0); // Set RW to 0 for write mode

// IOWR(LCD\_DATA, 0, command); // Send the command on the data bus

// // Pulse the enable signal (E) to latch the command

// IOWR(LCD\_EN, 0, 1);

// usleep(LCD\_DELAY\_US);

// IOWR(LCD\_EN, 0, 0);

// usleep(LCD\_DELAY\_US);

// }

// // Function to send data to the LCD

// void sendData(unsigned char data) {

// IOWR(LCD\_RS, 0, 1); // Set RS to 1 for data mode

// IOWR(LCD\_RW, 0, 0); // Set RW to 0 for write mode

// IOWR(LCD\_DATA, 0, data); // Send the data on the data bus

// // Pulse the enable signal (E) to latch the data

// IOWR(LCD\_EN, 0, 1);

// usleep(LCD\_DELAY\_US);

// IOWR(LCD\_EN, 0, 0);

// usleep(LCD\_DELAY\_US);

// }

while (1);

return 0;

}