Week 3
Dr. David Ward
Hybrid Embedded Systems
Today’s Agenda

- Discuss Homework and Labs
  - HW #2 due September 17
- Interrupt Systems
  - NIOS II, HAL, PIO
- Lab #3 Overview
  - PS/2 Keyboard design
Expanding the I/O System

- Interrupt system: Allow devices to get processor’s attention

![Diagram showing the I/O system with Processor, Memory-I/O Bus, Main Memory, I/O Controllers, Disk, Graphics, and Network connections.](image-url)
Interrupt Terms

- **Hardware interrupt**
  - an exception caused by a signal from hardware (request for action)
- **IRQ**- “Interrupt ReQuest”
- **ISR**- Interrupt Service Routine
  - Code executed on the main processor in service of the device
- **Interrupt vector**
  - commonly referred to term for memory location holding the address of the interrupt service routine (ISR)
- **Interrupt mask**
  - used to block interrupts
- **Software Exception**
  - Any condition or signal that interrupts normal program execution.
    - Software: divide by 0, illegal opcode, etc.
Interrupt Terminology (Nios II)

- **application context**: the status of the Nios II processor and the HAL during normal program execution, outside of the exception handler.
- **context switch**: the process of saving the Nios II processor’s registers on an exception, and restoring them on return from the interrupt service routine.
- **exception handler**: the complete system of software routines, which service all exceptions and pass control to ISRs as necessary.
- **exception overhead**: additional processing required by exception processing.
  - The exception overhead for a program is the sum of all the time occupied by all context switches.
- **implementation-dependent instruction**: a Nios II processor instruction that is not supported on all implementations of the Nios II core. For example, the mul and div instructions are implementation-dependent, because they are not supported on the Nios II/e core.
- **interrupt context**: the status of the Nios II processor and the HAL when the exception handler is executing.
- **interrupt request (IRQ)**: a signal from a peripheral requesting a hardware interrupt.
- **interrupt service routine (ISR)**: a software routine that handles an individual hardware interrupt.
- **invalid instruction**: an instruction that is not defined for any implementation of the Nios II processor.
- **unimplemented instruction**: an implementation-dependent instruction that is not supported on the particular Nios II core implementation that is in your system. For example, in the Nios II/e core, mul and div are unimplemented.
- **other exception**: an exception which is not a hardware interrupt nor a trap.
Difference between Interrupts and Software Exceptions

- **Interrupt** is a hardware-based event
- **Software exceptions** originate from the program currently running on the processor
  - Handled in a similar way as interrupts: each designed cause will change the program to an “exception handler” routine.

### Table 9.2: Nios II Exception Cause Codes

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause Code</th>
<th>Cause Symbol (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0</td>
<td>NIOS2_EXCEPTION_RESET</td>
</tr>
<tr>
<td>Processor-only Reset Request</td>
<td>1</td>
<td>NIOS2_EXCEPTION_CPU_ONLY_RESET_REQUEST</td>
</tr>
<tr>
<td>Interrupt</td>
<td>2</td>
<td>NIOS2_EXCEPTION_INTERRUPT</td>
</tr>
<tr>
<td>Trap Instruction</td>
<td>3</td>
<td>NIOS2_EXCEPTION_TRAP_INST</td>
</tr>
<tr>
<td>Unimplemented Instruction</td>
<td>4</td>
<td>NIOS2_EXCEPTION_UNIMPLEMENTED_INST</td>
</tr>
<tr>
<td>Illegal Instruction</td>
<td>5</td>
<td>NIOS2_EXCEPTION_ILLEGAL_INST</td>
</tr>
<tr>
<td>Misaligned Data Address</td>
<td>6</td>
<td>NIOS2_EXCEPTION_MISALIGNED_DATA_ADDR</td>
</tr>
<tr>
<td>Misaligned Destination Address</td>
<td>7</td>
<td>NIOS2_EXCEPTION_MISALIGNED_TARGET_PC</td>
</tr>
<tr>
<td>Division Error</td>
<td>8</td>
<td>NIOS2_EXCEPTION_DIVISION_ERROR</td>
</tr>
<tr>
<td>Supervisor-only instruction Addr</td>
<td>9</td>
<td>NIOS2_EXCEPTION_SUPERVISOR_ONLY_INST_ADDR</td>
</tr>
<tr>
<td>Supervisor-only Instruction</td>
<td>10</td>
<td>NIOS2_EXCEPTION_SUPERVISOR_ONLY_INST</td>
</tr>
<tr>
<td>Supervisor-only Data Address</td>
<td>11</td>
<td>NIOS2_EXCEPTION_SUPERVISOR_ONLY_DATA_ADDR</td>
</tr>
<tr>
<td>Translation lookaside buffer (TLB) Miss</td>
<td>12</td>
<td>NIOS2_EXCEPTION_TLB_MISS</td>
</tr>
<tr>
<td>TLB Permission Violation (execute)</td>
<td>13</td>
<td>NIOS2_EXCEPTION_TLB_EXECUTE_PERM_VIOLATION</td>
</tr>
<tr>
<td>TLB Permission Violation (read)</td>
<td>14</td>
<td>NIOS2_EXCEPTION_TLB_READ_PERM_VIOLATION</td>
</tr>
<tr>
<td>TLB Permission Violation (write)</td>
<td>15</td>
<td>NIOS2_EXCEPTION_TLB_WRITE_PERM_VIOLATION</td>
</tr>
<tr>
<td>MPU Region Violation (instruction)</td>
<td>16</td>
<td>NIOS2_EXCEPTION_MPU_INST_REGION_VIOLATION</td>
</tr>
<tr>
<td>MPU Region Violation (data)</td>
<td>17</td>
<td>NIOS2_EXCEPTION_MPU_DATA_REGION_VIOLATION</td>
</tr>
<tr>
<td>Cause unknown (2)</td>
<td>-1</td>
<td>NIOS2_EXCEPTION_CAUSE_NOT_PRESENT</td>
</tr>
</tbody>
</table>

---

**Notes to Table 9.2:**

1. Cause symbols are defined in `sys/alt_exceptions.h`.
2. This value is passed to the exception handler if the `cause` argument if the cause is not known; for example, if the cause register is not implemented in the Nios II processor core.
NIOS II Interrupt

- Many devices
- Interrupt controller is generated by SOPC builder
  - Controlled by IRQ number assignment
  - 32 level-sensitive interrupt request lines
    - irq0 – irq31
- Software or Hardware priority
  - Software (32 slave IRQs)
    - Priority encoded IRQ[0] is highest priority
  - Hardware (64 slave IRQs)
    - Passes highest priority to the master port
    - Lesser priority IRQs undetectable until higher priority IRQ is cleared
- Supports nested interrupts
NIOS II Interrupt Controller

Figure 2-12. IRQ Mapping Using Software Priority
Up to 64 slave IRQ signals
Exception Handling in Nios II

- Saves the status register in `estatus`
  - This means that if hardware interrupts are enabled, the EPIE bit of `estatus` is set.
- Disables hardware interrupts.
- Saves the next execution address in `ea (r29)` - the effective address register.
- Transfers control to the Nios II processor exception address
- Nios II exceptions and interrupts are not vectored.
  - The same exception address receives control for all types of interrupts and exceptions.
  - The exception handler at that address must determine the type of exception or interrupt.
Hardware Abstraction Layer (HAL)

- Lightweight runtime environment
  - provides simple device driver interface
  - allows access to devices with common C library functions
HAL Services

- Integration with C standard library
- Device drivers
- HAL API
- System initialization
- Device initialization
Interfacing with HAL

- Application developers
  - using HAL API or C standard library
  - do not consider the underlying hardware
- Device driver developers
  - making drivers for low-level hardware
  - interface newly written drivers with HAL
Generic Devices in HAL

- Character-mode devices
- Timer devices
- File subsystems
- Ethernet devices
- DMA devices
- Flash memory devices
Exception Handling in HAL

- Top level exception handler
  - creates private stack
  - stores register values onto the stack
  - determines the type of exception and invoke the right SW or HW handler
- Software exception handler
- Hardware interrupt handler
  - ISRs for peripherals
1. Save the status registers
2. Disable interrupts
3. Save the EA (effective address register)...where the program was executing before interrupt
4. Transfer control to the interrupt vector

EA is also known as PC (Program Counter)
Exception Handling in HAL

- Check `estatus` to see if hardware interrupt is enabled
  - if yes, check to see if the interrupt is hardware by checking `ipending`
    - if yes, corresponding ISR is called
    - if no, call the software exception handler
- Hardware interrupts have higher priority than software exceptions
Exception Handling in HAL

- Upon returning from exceptions, the top-level handler
  - restores the stack pointer
  - restores the registers from the stack
  - exits by issuing an *eret* instruction
Exception Handling in HAL

- HW Handlers
  - 32 hardware interrupts (0 to 31, with 0 as highest priority)
  - priority is HAL specific and not NIOS II
Exception Handling in HAL

- SW handlers
  - used for unimplemented instructions
    - e.g. running multiplication on NIOS II with no multipliers
  - traps
Interrupt Diagram Steps

1. **I/O interrupt**
2. **Save EA**
3. **Get interrupt service address and change NIOS EA to that address**
4. **Advantage:**
   - User program progress is only halted during actual transfer
5. **Disadvantage, special hardware is needed to:**
   - Cause an interrupt (I/O device)
   - Detect an interrupt (processor)
   - Save the proper states to resume after the interrupt (processor)
Get Interrupt Service Address

- (3) Get interrupt service addr and change NIOS EA to that address

- For each IRQ #, there is a memory address story in an interrupt vector table (also in memory).

- Upon that hardware selecting which IRQ, a LOAD to the corresponding vector table entry is performed
  - The result of that loaded value (4 bytes) is placed in the EA (effective address) of the NIOS Processor
  - Thus the next instruction executed is the code for the Interrupt Service Routine (ISR)
PIO (Parallel I/O)

Edge capture register
- Synchronously capture
- Rising edge
- Falling edge
- Ether edge
- Enable bit-clearing for edge capture register

Interrupt
- Generate IRQ
- Level
  (Interrupt CPU when any unmasked I/O pin is logic true)
- Edge
  (Interrupt CPU when any unmasked bit in the edge-capture register is logic true)

Warning: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs.
To Do: cpu_0: No reset vector has been specified for this CPU. Please parametrize the CPU to resolve this issue.
To Do: cpu_0: No exception vector has been specified for this CPU. Please parametrize the CPU to resolve this issue.
Warning:pio_0: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.
PIO Core Types: Output, Input w/ Interrupt, Bi-directional
PIO Device Map

- Base address assigned from SOPC
- But device is 4 register locations (4 bytes each location)
- Registers are: DATA, Direction, InterruptMask, Edgecapture

<table>
<thead>
<tr>
<th>Table 1. PIO Register Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset in bytes</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
 PIO CORE Data

**Data Input and Output**

The PIO core I/O ports can connect to either on-chip or off-chip logic. The core can be configured with inputs only, outputs only, or both inputs and outputs. If the core is used to control bidirectional I/O pins on the device, the core provides a bidirectional mode with tristate control.

The hardware logic is separate for reading and writing the data register. Reading the data register returns the value present on the input ports (if present). Writing data affects the value driven to the output ports (if present). These ports are independent; reading the data register does not return previously-written data.
Edge Capture

The PIO core can be configured to capture edges on its input ports. It can capture low-to-high transitions, high-to-low transitions, or both. Whenever an input detects an edge, the condition is indicated in the edge capture register. The types of edges detected is specified at system generation time, and cannot be changed via the registers.
The PIO core can be configured to generate an IRQ on certain input conditions. The IRQ conditions can be either:

- **Level-sensitive**—The PIO core hardware can detect a high level. A NOT gate can be inserted external to the core to provide negative sensitivity.

- **Edge-sensitive**—The core’s edge capture configuration determines which type of edge causes an IRQ.

Interrupts are individually maskable for each input port. The interrupt mask determines which input port can generate interrupts.
PIO CORE Configurations

Figure 9-2. PIO Core with Input Ports, Output Ports, and IRQ Support

Figure 9-3. PIO Core with Bidirectional Ports
### Expanded PIO Device Map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register Name</th>
<th>R/W</th>
<th>(n-1)</th>
<th>...</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>data</td>
<td>R</td>
<td>Data value currently on PIO inputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W</td>
<td>New value to drive on PIO outputs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>direction (1)</td>
<td>R/W</td>
<td>Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>interruptmask (1)</td>
<td>R/W</td>
<td>IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>edgecapture (1), (2)</td>
<td>R/W</td>
<td>Edge detection for each input port.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>outset</td>
<td>W</td>
<td>Specifies which bit of the output port to set.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>outclear</td>
<td>W</td>
<td>Specifies which output bit to clear.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes to Table 9-2:**

1. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.
2. Writing any value to edgecapture clears all bits to 0.
**Interruptmask Register**

Setting a bit in the interruptmask register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See “Interrupt Behavior” on page 9–7.

The interruptmask register only exists when the hardware is configured to generate IRQs. If the core cannot generate IRQs, reading interruptmask returns an undefined value, and writing to interruptmask has no effect.

After reset, all bits of interruptmask are zero, so that interrupts are disabled for all PIO ports.
Expanded PIO Device Map

**edgecapture Register**

Bit \( n \) in the edgecapture register is set to 1 whenever an edge is detected on input port \( n \). An Avalon-MM master peripheral can read the edgecapture register to determine if an edge has occurred on any of the PIO input ports. If the option **Enable bit-clearing for edge capture register** is turned off, writing any value to the edgecapture register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

The type of edge(s) to detect is fixed in hardware at system generation time. The edgecapture register only exists when the hardware is configured to capture edges. If the core is not configured to capture edges, reading from edgecapture returns an undefined value, and writing to edgecapture has no effect.

- Rising Edge
- Falling Edge
- Either Edge
outset and outclear Registers

You can use the outset and outclear registers to set and clear individual bits of the output port. For example, to set bit 6 of the output port, write 0x40 to the outset register. Writing 0x08 to the outclear register clears bit 3 of the output port.

These registers are only present when the option Enable individual bit set/clear output register is turned on.
Hardware Abstraction Layer (HAL).

- System software that allows programmer to control the hardware devices using a series of function calls
  - Control how or whether interrupts should be handled
  - i.e. turn off interrupts when code executing on NIOS II CPU is critical (time constraint), don’t allow serial input cable to get processor’s attention
HAL Interface

- alt_irq_register()
- alt_irq_disable()
- alt_irq_enable()

- alt_irq_disable_all()
- alt_irq_enable_all()
- alt_irq_interruptible()
- alt_irq_non_interruptible()
- alt_irq_enabled()
NIOS II Interrupt Setup

- Declare variable for holding information that will be passed from device to ISR
- Enable the device
  - Set the control registers on the device
- Register the ISR with the HAL system
  - Done in main()
- Develop code for ISR
Enable Device and Register ISR

Declare variable to hold state of device that will be passed to ISR

Set the controls on the PIO device to capture and cause interrupts

Register the ISR with the HAL interface: assign address of the ISR to the Interrupt vector table.

```c
#include "sys/alt_irq.h"
#include "system.h"

... /* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...

/* Initialize the button_pio. */
static void void init_button_pio()
{
    /* Recast the edge_capture pointer to match the alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;
    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALONPIO_IRQ_MASK(BUTTON_PIO_BASE, 0x0);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALONPIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    /* Register the ISR. */
    alt_irq_register( BUTTON_PIO_IRQ,
                     edge_capture_ptr,
                     handle_button_interrups );
}
```
Enable Device and Register ISR

```
#include "sys/alt_irq.h"
#include "system.h"

...

/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...

/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
       alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;

    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALONPIO_IRQ_MASK(BUTTONPIO_BASE, 0xf);

    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALONPIO_EDGE_CAP(BUTTONPIO_BASE, 0x0);

    /* Register the ISR. */
    alt_irq_register(BUTTONPIO_IRQ, 
                     edge_capture_ptr, 
                     handle_button_interrupts);
}
```

This is decided at SOPC builder time, user can define which IRQ # [0 up to 31]

Memory-mapped device address assigned at SOPC builder time
#include “system.h”
#include “altera_avalon_pio_regs.h”
#include “alt_types.h”

static void handle_button_interrupts(void* context, alt_u32 id)
{
    // cast the context pointer to an integer pointer.
    volatile int* edge_capture_ptr = (volatile int*) context;

    //Read the edge capture register on the button PIO & Store value.
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTONS_BASE);

    //Write to the edge capture register to reset it.
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTONS_BASE, 0);

    // reset interrupt capability for the Button PIO.
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTONS_BASE, 0xf);
}
static void handle_button_interrupts(void* context, alt_u32 id)
{
    // cast the context pointer to an integer pointer.
    volatile int* edge_capture_ptr = (volatile int*) context;

    // Read the edge capture register on the button PIO & Store value.
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTONS_BASE);

    // Write to the edge capture register to reset it.
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTONS_BASE, 0);

    // reset interrupt capability for the Button PIO.
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTONS_BASE, 0xf);
    if (*edge_capture_ptr == 1) // Check if the first button was pressed
    {
        LCDwrite("hello world\n");
    }

    if (*edge_capture_ptr == 4) // Check if the 4th button was pressed
    {
        IOWR(LEDS_BASE, 0, (1 & 0xF)); // Write 1 to the LEDs
        usleep(1000000);
        IOWR(LEDS_BASE, 0, (0 & 0xF)); // Write 0 to the LEDs
    }
}
NIOS II Interrupt Handling Performance

Because the Nios II processor is highly configurable, there is no single typical number for each metric. This section provides data points for each of the Nios II cores under the following assumptions:

- All code and data is stored in on-chip memory.
- The ISR code does not reside in the instruction cache.
- The software under test is based on the Altera-provided HAL exception handler system.
- The code is compiled using compiler optimization level -03, that is, high optimization.

Table 8–1 lists the interrupt latency, response time, and recovery time for each Nios II core.

<table>
<thead>
<tr>
<th>Core</th>
<th>Latency</th>
<th>Response Time</th>
<th>Recovery Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nios II/f</td>
<td>10</td>
<td>105</td>
<td>62</td>
</tr>
<tr>
<td>Nios II/s</td>
<td>10</td>
<td>128</td>
<td>130</td>
</tr>
<tr>
<td>Nios II/e</td>
<td>15</td>
<td>485</td>
<td>222</td>
</tr>
</tbody>
</table>

Note to Table 8–1:
(1) The numbers indicate time measured in CPU clock cycles.
Lab # 3 Overview

PS/2 Keyboard
Lab #3 Files on class web site

- **ASCII-Map.pdf**
  - Keyboard code to ASCII code mapping

- **KB_ScanCode.v**
  - Verilog file to read serial keyboard data to 8 bit parallel data
  - Implementation details in Chapter 7.6 (textbook)

- **Scan2ASCII.mif**
  - Memory initialization file
    - Initialize ROM
Keyboard Interface Logic

Synchronous serial

Bidirectional data and clock
  * Converted to 8-bit ASCII

- Data transmission is synchronized with clock
- Transmission starts when data is 0
- Transmission stop bit (1) terminates transfer

Keyboard

Data transmission

Odd-Parity bit
Keyboard Communication Modes

Keyboard samples clock line every 60 uSecs

- Checks the status of clock (if 0 cannot send) before sending

- Forces the clock signal to 0 for > 60 uSec
  - Done while preparing data
  - Then allows keyboard to drive clock to 1

- When system sends data to the keyboard, it forces the data line to 0 until the keyboard starts to clock the data stream
# Keyboard Commands

### System Commands to Keyboard

<table>
<thead>
<tr>
<th>Command</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set/Reset Status Indicators</td>
<td>ED</td>
</tr>
<tr>
<td>Echo</td>
<td>EE</td>
</tr>
<tr>
<td>Invalid Command</td>
<td>EF</td>
</tr>
<tr>
<td>Select Alternate Scan Codes</td>
<td>F0</td>
</tr>
<tr>
<td>Invalid Command</td>
<td>F1</td>
</tr>
<tr>
<td>Read ID</td>
<td>F2</td>
</tr>
<tr>
<td>Set Typematic Rate/Delay</td>
<td>F3</td>
</tr>
<tr>
<td>Enable</td>
<td>F4</td>
</tr>
<tr>
<td>Default Disable</td>
<td>F5</td>
</tr>
<tr>
<td>Set Default</td>
<td>F6</td>
</tr>
<tr>
<td>Set All Keys – Typematic</td>
<td>F7</td>
</tr>
<tr>
<td>- Make/Break</td>
<td>F8</td>
</tr>
<tr>
<td>- Make</td>
<td>F9</td>
</tr>
<tr>
<td>- Typematic/Make/Break</td>
<td>FA</td>
</tr>
<tr>
<td>Set Key Type – Typematic</td>
<td>FB</td>
</tr>
<tr>
<td>- Make/Break</td>
<td>FC</td>
</tr>
<tr>
<td>- Make</td>
<td>FD</td>
</tr>
<tr>
<td>Respond</td>
<td>FE</td>
</tr>
<tr>
<td>Reset</td>
<td>FF</td>
</tr>
</tbody>
</table>

### Keyboard Commands to System

<table>
<thead>
<tr>
<th>Command</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Detection Error/Overrun</td>
<td>00</td>
</tr>
<tr>
<td>Keyboard ID</td>
<td>83AB</td>
</tr>
<tr>
<td>BAT Completion Code</td>
<td>AA</td>
</tr>
<tr>
<td>BAT Failure Code</td>
<td>FC</td>
</tr>
<tr>
<td>Echo</td>
<td>EE</td>
</tr>
<tr>
<td>Acknowledge (ACK)</td>
<td>FA</td>
</tr>
<tr>
<td>Resent</td>
<td>FE</td>
</tr>
<tr>
<td>Key Numb</td>
<td>Make Code</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>01</td>
<td>0E</td>
</tr>
<tr>
<td>02</td>
<td>16</td>
</tr>
<tr>
<td>03</td>
<td>1E</td>
</tr>
<tr>
<td>04</td>
<td>26</td>
</tr>
<tr>
<td>05</td>
<td>25</td>
</tr>
<tr>
<td>06</td>
<td>2E</td>
</tr>
<tr>
<td>07</td>
<td>36</td>
</tr>
<tr>
<td>08</td>
<td>3D</td>
</tr>
<tr>
<td>09</td>
<td>3E</td>
</tr>
<tr>
<td>10</td>
<td>46</td>
</tr>
<tr>
<td>11</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>4E</td>
</tr>
<tr>
<td>13</td>
<td>55</td>
</tr>
<tr>
<td>14</td>
<td>66</td>
</tr>
<tr>
<td>15</td>
<td>UD</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>1D</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>2C</td>
</tr>
<tr>
<td>22</td>
<td>35</td>
</tr>
<tr>
<td>23</td>
<td>3C</td>
</tr>
<tr>
<td>24</td>
<td>44</td>
</tr>
<tr>
<td>25</td>
<td>45</td>
</tr>
<tr>
<td>26</td>
<td>4D</td>
</tr>
<tr>
<td>27</td>
<td>54</td>
</tr>
<tr>
<td>28</td>
<td>5B</td>
</tr>
<tr>
<td>29</td>
<td>5D</td>
</tr>
<tr>
<td>30</td>
<td>56</td>
</tr>
<tr>
<td>31</td>
<td>1C</td>
</tr>
<tr>
<td>32</td>
<td>1B</td>
</tr>
<tr>
<td>33</td>
<td>23</td>
</tr>
</tbody>
</table>

- **Xmits** Make code when key pressed
- **Break code** is 0xF0 followed by Make code when key is released

**Standard 104-key Keyboard**

**Keyboard Scan Codes & ASCII Characters (ASCII-Map.pdf)**

![Keyboard Scan Codes & ASCII Characters](ASCII-Map.pdf)
module KB_ScanCode(input KBclk, KBdata, ResetKB, SYNclk, 
output ScanRdy, output KeyReleased, 
output reg [7:0] ScanCode);

reg KBClock; // KB Synchronized Clock

always @(posedge SYNclk) KBClock <= KBclk;

//Detect Start and Stop
//Collect data bite
//Wait for key release

endmodule
Detect Start and Stop

- Uses fast system clock (~50Mhz)
- Waits for KBData to become 0 then issues StartBitDetected
- Remains 1 until Completed signal is received
  - Completed signal comes from Data Bit Collection module

```verilog
wire Completed;
reg StartBitDetected;

always @(posedge SYNclk)
  if (ResetKB) StartBitDetected <= 1'b0;
  else
    case (StartBitDetected)
      0: if (KBdata) StartBitDetected <= 1'b0;
        else StartBitDetected <= 1'b1;
      1: if (~Completed) StartBitDetected <= 1'b1;
        else StartBitDetected <= 1'b0;
    endcase
```
Data Bit Collection

```verilog
reg [3:0] p_state, n_state;
parameter [3:0] Idle=0,
    Bit1=1, Bit2=2, Bit3=3, Bit4=4,
    Bit5=5, Bit6=6, Bit7=7, Bit8=8,
    OPar=9, Stop=10;
always @(posedge KBClock)
    if (ResetKB) p_state <= Idle;
    else p_state <= n_state;
always @(p_state, StartBitDetected, KBdata) begin
    case (p_state)
        Idle: if (~StartBitDetected) n_state <= Idle;
             else n_state <= Bit1;
        Bit1,Bit2,Bit3,Bit4,Bit5,Bit6,Bit7,Bit8, OPar:
            n_state <= p_state + 1;
        Stop: n_state <= Idle;
        default: n_state <= Idle;
    endcase
end
assign Completed = (p_state == Stop) ? 1 : 0;
assign ScanRdy = (p_state == Stop);
always @(posedge KBClock)
    if (p_state > Bit1 & p_state <= Bit8)
        ScanCode <= {KBdata, ScanCode[7:1]};
```

- Shift register that clocks data bits as they appear on KBData on rising edge of KBClock
- Uses KBClock for this state machine
- Waits for StartBitDetected, then cycles through Bit1-Bit8 states
  - Data bits appear on KBData on rising edge of KBClock
- When all bits have been detected, it goes to stop state
Waiting For Key Release

```verilog
reg [1:0] p_release, n_release;
parameter [1:0] WaitF0 = 0, WaitRPT = 1, Released = 2;
always @(posedge KBClock) p_release <= n_release;
always @(p_release, ScanCode, ScanRdy)
    case (p_release)
        WaitF0 : if (ScanCode == 8'hF0) && (ScanRdy))
                    n_release = WaitRPT;
                    else n_release = WaitF0;
        WaitRPT : if (ScanRdy) n_release = Released;
                    else n_release = WaitRPT;
        Released: n_release = WaitF0;
        default: n_release = WaitF0;
    endcase
assign KeyReleased = (p_release == Released) ? 1 : 0;
```

*Waits for 0xF0 on ScanCode*
ASCII Conversion Memory Initialization File (scan2ASCII.mif)

DEPTH = 128;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = DEC;
% Keyboard Scan Code to ASCII %
CONTENT
BEGIN
% Set 2: ASCII ; Key Char %
%----------------------------------%  
  0D :  09 ; % 16     Tab %
  0E :  96 ; %  1    ' %
  11 :  00 ; %  60   Alt %
  12 :  00 ; %  44   Shift %
  14 :  00 ; %  58   Ctrl %
  15 :  81 ; %  17    Q %
  16 :  49 ; %   2    L %
  1A :  90 ; %  46    Z %
  1B :  83 ; %  32    S %
  1C :  65 ; %  31    A %
  . .
  66 :  08 ; %  15   BS %
END;
Questions?