Understanding Interrupt Mechanisms

Preparing for RISC-V Interrupt Support

Learning Goal: Implement and understand key components for interrupt handling

Requirements: Verilator, GTKWave, CS200 Extension

---

1 Introduction

In this lab, you will implement and extend key modules that are fundamental to understanding interrupt mechanisms in RISC-V processors. While we won't be fully integrating interrupt support into the CPU in this lab, the work you do here will be crucial for implementing full interrupt support in the next lab. This lab is structured in three main parts:

1. **Extending the Buttons Controller**: You will modify the existing buttons controller to add interrupt generation capability. This will allow you to understand how hardware peripherals can signal the need for CPU attention.
2. **Implementing the CSR Controller**: You will create a Control and Status Registers (CSR) controller, which is essential for managing interrupt-related information in the RISC-V architecture. This module will help you understand how the CPU keeps track of and controls its interrupt state.
3. **Writing a Simple Interrupt Handler**: Finally, you will write a basic interrupt handler in assembly. This will demonstrate how software can respond to hardware interrupts even if the CPU is busy executing other tasks.

The goal of this lab is to provide you with a deep understanding of the components involved in interrupt handling, preparing you for full integration in the next lab.

2 Extending the Buttons Controller

2.1 Description

The Button Controller has been extended to support interrupt generation. It now provides an MMIO interface for software to retrieve the current states of buttons and switches, configure interrupt trigger modes, and handle interrupts. Figure 1 shows the layout of buttons and switches on the Gecko5 board that this controller interfaces with.

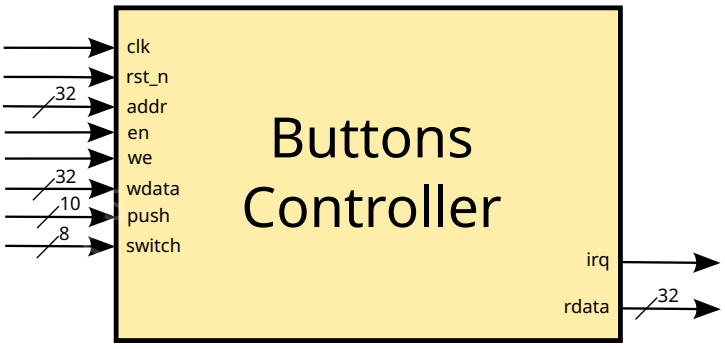

Figure 1: Diagram of new buttons controller

2.1.1 Interface Signals

The extended Button Controller has the following interface signals:

Table 1: Extended Button Controller Interface Signals

Name	Direction	Width	Description
clk	Input	1	Clock
rst_n	Input	1	Reset (active low)
en	Input	1	Enable
we	Input	1	Write enable
addr	Input	32	Address
wdata	Input	32	Write data
push	Input	10	States of push buttons
switch	Input	8	States of switches
rdata	Output	32	Read data
irq	Output	1	Interrupt request

2.1.2 MMIO Registers

The Button Controller now has four MMIO registers in the region starting at base address 0x70000000 and ending at 0x70000FFF:

Table 2: Extended Button Controller MMIO Registers

Name	Offset	Access	Description
VAL	0x0	Read-only	Value register
SRC	0x4	Read/Write	Interrupt source register
PTM	0x8	Write-only	Push-button trigger mode register
STM	0xC	Write-only	Switch trigger mode register

2.1.3 Register Descriptions

- **VAL Register**
  The VAL register remains unchanged, containing the current states of push buttons and switches.

- **SRC Register**

  The SRC register indicates which buttons or switches have triggered an interrupt. It has the following bit fields:

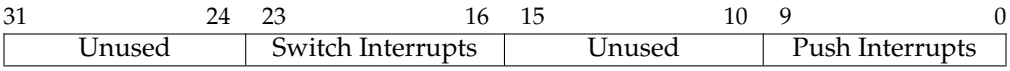

Figure 2: SRC Register Layout

  Important: The SRC register should keep the rising edge and falling edge interrupts set until they are cleared by software.

- **PTM Register**

  The PTM register configures the trigger mode for each push button. It uses 2 bits per button, allowing for 4 trigger modes.

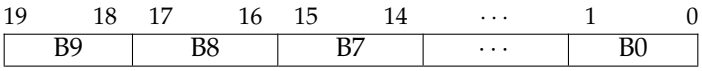

Figure 3: PTM Register Layout

  Where B9 is button_top, B8 is button_bottom, ..., and B0 is joystick_pressed.

- **STM Register**

  The STM register configures the trigger mode for each switch. It uses 2 bits per switch, allowing for 4 trigger modes.

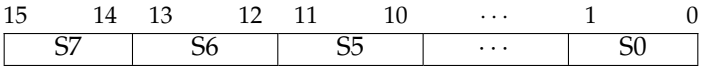

Figure 4: STM Register Layout

  Where S0 to S7 correspond to the 8 switches on the board.

2.1.4 Trigger Modes

Each button and switch can be configured with one of four trigger modes. These modes determine the conditions under which an interrupt is generated:

- **00: ACTIVE_LOW**

  In this mode, an interrupt is generated when the button or switch is in a low state (0). The interrupt remains active as long as the input stays low.

- **01: RISING_EDGE**

  This mode triggers an interrupt on the rising edge of the input signal, i.e., when the button or switch transitions from a low state (0) to a high state (1). It's useful for detecting when a button is pressed.

- **10: FALLING_EDGE**

  Opposite to RISING_EDGE, this mode generates an interrupt on the falling edge of the input signal, i.e., when the button or switch transitions from a high state (1) to a low state (0). It's typically used to detect when a button is released.

- **11: ACTIVE_HIGH (default)**

  In this mode, an interrupt is generated when the button or switch is in a high state (1). The interrupt remains active as long as the input stays high. This is the default mode for all buttons and switches upon reset.

Note: The choice of trigger mode depends on the specific requirements of your application. For example:

- Use RISING_EDGE or FALLING_EDGE when you want to detect the exact moment a button is pressed or released.
- Use ACTIVE_HIGH or ACTIVE_LOW when you want to continuously generate interrupts while a button is held down or a switch remains in a certain position.

Remember that the trigger mode for each button and switch can be configured independently by setting the appropriate bits in the PTM and STM registers.

2.1.5 Operation

The Button Controller operates with a one-cycle latency for read operations from both the VAL and SRC registers. The process works as follows:

1. When the controller receives a read operation from either the VAL or SRC register with the enable (en) signal asserted, it captures the current state of that register.
2. In the next clock cycle, it returns the captured state through the `rdata` output. This one-cycle delay ensures synchronous operation and allows time for the controller to properly sample and prepare the data.

Other operations include:

- Writing all zeros to the SRC register clears all falling and rising edge interrupts.
- Writing to PTM or STM registers configures the trigger modes.
- The controller continuously monitors button and switch states, comparing them with previous states to detect edges.
- When an interrupt condition is met, the corresponding bit in the SRC register is set.
- The irq_o signal is asserted if any bit in the SRC register is set.

Implementation Requirements:

- All unused bits in registers must be maintained at 0.
- The enable (en) signal must be high for any read or write operation to be performed.
- The write enable (we) signal must be high for any write operation to the SRC, PTM, or STM registers.
- Write operations are ignored if either en or we is low.
- Read operations are ignored if en is low.

2.1.6 Example Usage

To configure the first push button for rising edge detection:

1. Write 0x00000001 to address 0x70000008 (PTM register)

To clear all rising/falling edge pending interrupts:

1. Write 0x00000000 to address 0x70000004 (SRC register)

To read the current interrupt status:

1. Perform a read operation from address 0x70000004 (SRC register)
2. The interrupt status will be available in the next clock cycle

2.2 Exercise

- Import the Buttons Controller from Lab B to use it as a base.
- Extend the **Buttons Controller** in the file named `buttons.v` to support interrupt generation as described in this section.
- Implement the SRC, PTM, and STM registers, and add logic for interrupt generation based on trigger modes.
- Ensure that reading from both VAL and SRC registers has a one-cycle latency.
- When the component is implemented, create a testbench to verify its functionality before proceeding to the next step.

3 CSR Controller

3.1 Description

The Control and Status Registers (CSR) Controller is a crucial component in RISC-V processors for managing system control and status information, particularly for interrupt handling. This controller implements a subset of the RISC-V CSR registers relevant to basic interrupt handling.

3.1.1 Interface Signals

The CSR Controller has the following interface signals 3:

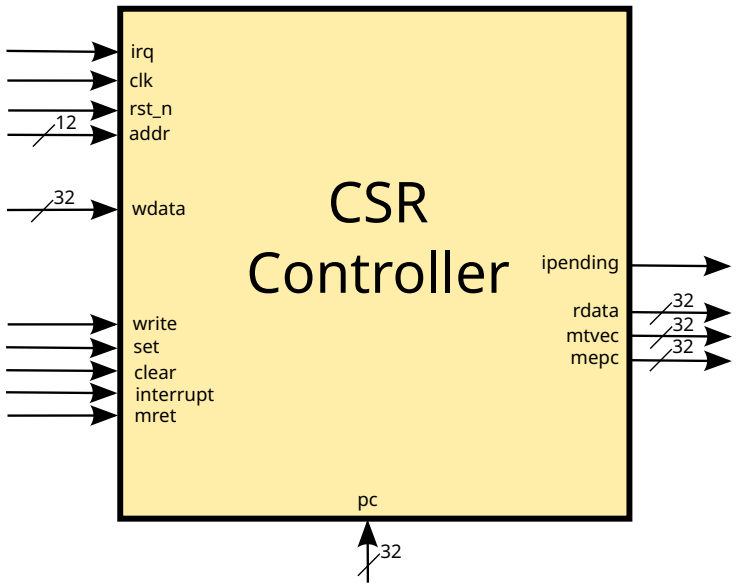

Figure 5: Diagram of CSR Controller

Table 3: CSR Controller Interface Signals

Name	Direction	Width	Description
clk	Input	1	Clock
reset_n	Input	1	Reset (active low and synchronous)
addr	Input	12	CSR address
wdata	Input	32	Write data
irq	Input	1	Interrupt request
pc	Input	32	Current program counter
write	Input	1	Write enable
set	Input	1	Set bits enable
clear	Input	1	Clear bits enable
interrupt	Input	1	Interrupt signal
mret	Input	1	Return from interrupt
rdata	Output	32	Read data
mtvec	Output	32	Trap vector base address
mepc	Output	32	Exception program counter
ipending	Output	1	Interrupt pending

Note: The write, set, and clear signals are mutually exclusive control signals - only one should be asserted at a time. If multiple signals are asserted simultaneously, the behavior is undefined.

3.1.2 CSR Registers

The CSR Controller implements the following registers:

3.1.3 Register Descriptions

General Register Requirements:

- All unused bits in any register must be maintained at 0

Table 4: CSR Registers

Name	Address	Description
mstatus	0x300	Machine status register
mie	0x304	Machine interrupt-enable register
mtvec	0x305	Machine trap-handler base address
mepc	0x341	Machine exception program counter
mcause	0x342	Machine trap cause
mip	0x344	Machine interrupt pending

- All registers are fully programmable (read/write).
- Write, set, and clear operations are mutually exclusive - only one should be asserted at a time
- **mstatus Register (0x300)**

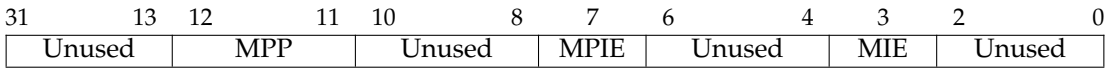

Figure 6: mstatus Register Layout

The mstatus register controls the CPU's current operating state:

  - MPP (Machine Previous Privilege): Indicate the privilege level before entering machine mode.
  - MPIE (Machine Previous Interrupt Enable): Stores the value of MIE before an interrupt occurs.
  - MIE (Machine Interrupt Enable): When set, interrupts are globally enabled in machine mode.

This register is crucial for interrupt handling. Unlike other registers, mstatus is initialized to 0x1800 on reset, which sets MPP to 3 (indicating machine mode) and all other bits to 0. This ensures the CPU starts in machine mode with interrupts disabled.

When an interrupt occurs:

  - The current MIE value is saved in MPIE
  - MIE is cleared, disabling further interrupts
  - MPP remains unchanged (always 3 in our implementation)

After handling the interrupt, the mret instruction restores MIE from MPIE, re-enabling interrupts if they were previously enabled and set the MPIE to 1.

- **mie Register (0x304)**

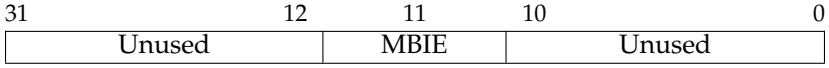

Figure 7: mie Register Layout

The mie register controls which interrupts are enabled:

  - MBIE: Machine Button Interrupt Enable. When set, button interrupts are enabled.

This register allows selective enabling of interrupt sources, providing fine-grained control over which events can interrupt the CPU.

- **mtvec Register (0x305)**

Figure 8: mtvec Register Layout

The mtvec register holds the base address of the interrupt handler:

  - All 32 bits store the base address for the interrupt handler.
  - The `mtvec` output of the CSR controller always reflects the current value of this register.

When an interrupt occurs, the CPU jumps to this address to begin executing the interrupt service routine.

- **mepc Register (0x341)**

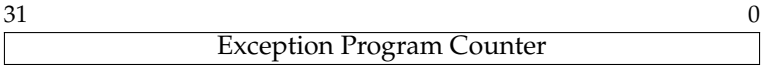

Figure 9: mepc Register Layout

The mepc register stores the program counter when an interrupt occurs:

  - All 32 bits hold the address of the instruction that was executing when the interrupt was taken.
  - The `mepc` output of the CSR controller always reflects the current value of this register.

This allows the CPU to return to the correct instruction after handling the interrupt.

- **mcause Register (0x342)**

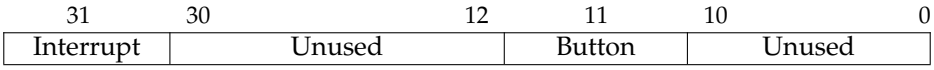

Figure 10: mcause Register Layout

The mcause register indicates the cause of an interrupt or exception:

  - Interrupt: Set to 1 for interrupts, 0 for exceptions. In our case, it will always be 1 during an interrupt.
  - Button: Set to 1 for button interrupts.

This register helps the interrupt handler determine how to respond to the event.

- **mip Register (0x344)**

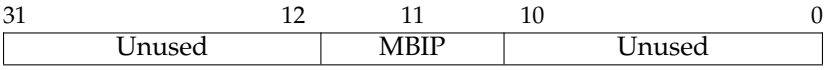

Figure 11: mip Register Layout

The mip register indicates which interrupts are currently pending:

  - MBIP: Machine Button Interrupt Pending. Set when a button interrupt is pending.

  This register allows the CPU to check which interrupts are waiting to be serviced, even if interrupts are currently disabled.

3.1.4 Operation

CSR (Control and Status Register) operations are designed with asynchronous read and synchronous write/set/clear functionality, similar to normal registers. This design allows for immediate reading of the current register value, even when a write, set, or clear operation is being performed. The updated value will be reflected in the next clock cycle. When a register is updated due to an interrupt handling process, this update is done synchronously like a normal write/set/clear operation would be.

- **Read Operation:** Reading a CSR is asynchronous. When `addr` selects a specific CSR, its current value is immediately output on `rdata`.
- **Write Operation:** Writing to a CSR is synchronous and occurs when `write` is asserted. The new value in `wdata` completely replaces the current register value on the next rising clock edge.
- **Set Operation:** The set operation performs a bitwise OR between the current register value and `wdata`. This operation allows setting specific bits without affecting others. For example, if the current value is 0x1010 and `wdata` is 0x0101, the result will be 0x1111.
- **Clear Operation:** The clear operation performs a bitwise AND between the current register value and the complement of `wdata`. This operation allows clearing specific bits without affecting others. For example, if the current value is 0x1111 and `wdata` is 0x0101, the result will be 0x1010.
- **Interrupt Request Handling:** The `irq` signal is an external interrupt request. When asserted, it updates the `mip` (Machine Interrupt Pending) register by setting the corresponding interrupt bit (MBIP). The `ipending` output is asserted if (1) interrupts are globally enabled (MIE bit of `mstatus` is set), and (2) any bit in `mip` is set while the corresponding interrupt enable bit in `mie` is set as well.
- **Interrupt Processing:** The `interrupt` signal, controlled by the CPU Controller, indicates that an interrupt is being processed. When asserted:
  - The current `pc` is saved to `mepc`
  - `mcause` is updated to indicate an interrupt (bit 31 set) and the cause (bit 11 for button interrupt)
  - `mstatus` is updated: the current MIE (bit 3) is saved to MPIE (bit 7), and MIE is cleared
- **Machine Return:** When the `mret` signal is asserted, indicating a return from an interrupt handler:
  - `mstatus` is updated: MIE (bit 3) is restored from MPIE (bit 7)
  - The CPU will use the value in `mepc` to return to the interrupted instruction

3.2 Exercise

- Implement the **CSR Controller** in the file named `csr.v` as described in this section.
- The six registers listed in Table 4 are already declared for you in the template, please use them without changing their names since they will be used in our testbenches to verify your implementation.

- Implement read, write, set, and clear operations, as well as interrupt handling logic.
- Ensure that the `mtvec` and `mepc` outputs always reflect their respective register values.
- After implementation, create a testbench to thoroughly test all features of the CSR Controller.

4 Interrupt Handler

4.1 Description

The interrupt handler is a crucial piece of software that responds to hardware interrupts. It's written in assembly language and works in conjunction with the CSR controller to manage interrupt processing. In this section, you'll implement a basic interrupt handler for button interrupts as part of a simple game.

4.2 Game Description

The goal is to implement a simple reaction game using interrupts. Here's how it works:

- The program enters a countdown (wait loop) after initialization.
- If a button is pressed during this countdown, triggering an interrupt, the player "wins". In the interrupt handler, you should set a memory location to 1 to indicate a win.
- If no interrupt occurs during the countdown, the player "loses".
- The game result is displayed using the LEDs: green for a win (Figure 13), red for a loss (Figure 12).

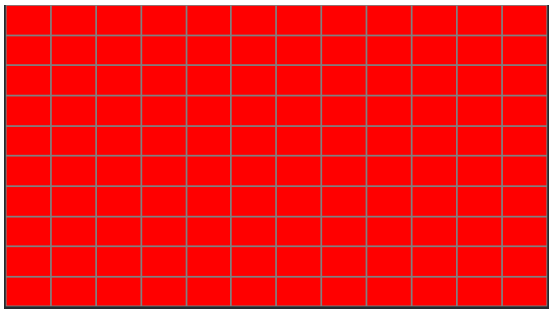

Figure 12: Losing state - Red LEDs

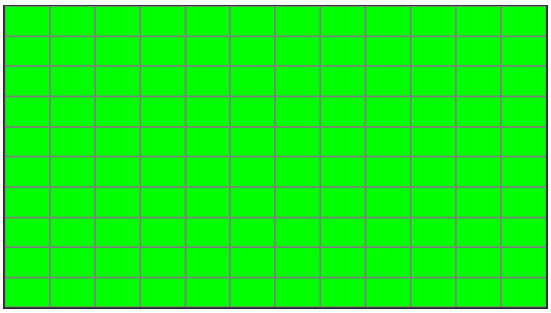

Figure 13: Winning state - Green LEDs

Figure 14: LED display for game outcomes

4.3 Memory Layout

The program uses the following memory regions:

- Stack: Growing down from 0x9FFFFFC0
- Game state: Stored at 0x90000000 (a single word)
  - Value 0: Indicates a loss
  - Value 1: Indicates a win

4.4 Interrupt Handler Structure and Key Points

Your interrupt handler should follow this general structure:

1. Save the context (registers) of the interrupted program
2. Identify the source of the interrupt (button press)
3. Handle the interrupt (set the win condition)
4. Clear the interrupt source
5. Restore the context
6. Return from the interrupt

When implementing your interrupt handler, keep these key points in mind:

- The interrupt handler address is already set up in the `mtvec` CSR in the provided template:

```
lui  t0, %hi(interrupt_handler)     # Load upper 20 bits of handler address
addi t0, t0, %lo(interrupt_handler) # Add lower 12 bits of handler address
csrw mtvec, t0                      # Set mtvec to handler address
```

- Initialize the stack pointer and enable interrupts.
- Configure the button controller for rising edge detection on the push buttons and switch buttons.
- In your interrupt handler:
  - Save the context by pushing all registers to the stack
  - Set the memory location 0x90000000 to 1 to indicate a "win"
  - Clear the interrupt source by writing 0 to the SRC register
  - Restore the context by popping all registers from the stack
  - Use `mret` to properly return from the handler

4.5 Exercise

- Complete the implementation of the interrupt handler in the file named `program.s`.
- Add the necessary interrupt setup code, including initializing the stack pointer and enabling interrupts.
- Implement the interrupt handler to set the win condition and clear the interrupt source.
- The template already includes the main game logic; integrate your interrupt handler with this existing code.
- Test your implementation by simulating button presses at different times and verifying the correct LED display for win and lose conditions.

5 Submission

Submit the following files for this lab:

- `buttons.v`: Extended Buttons Controller implementation
- `csr.v`: CSR Controller implementation
- `program.s`: Main program with game logic and interrupt handler implementation

Once you submit the files, you will receive a report describing the tests that were applied to your design and the results of those tests (success or failure). This is a preliminary submission so the result of the tests will not affect your grade and in case of failure, you will have some additional infos about what was the first test that failed of every testbench.