Interrupt Controller and Integration

Learning Goal: Add interrupt capability to the multicycle RISC-V CPU

Requirements: Verilator, GTKWave, CS200 Extension

1 Introduction

In this lab you will extend the multicycle RISC-V CPU with the capability to handle interrupts. The following schematic 1 shows the current state of the CPU.

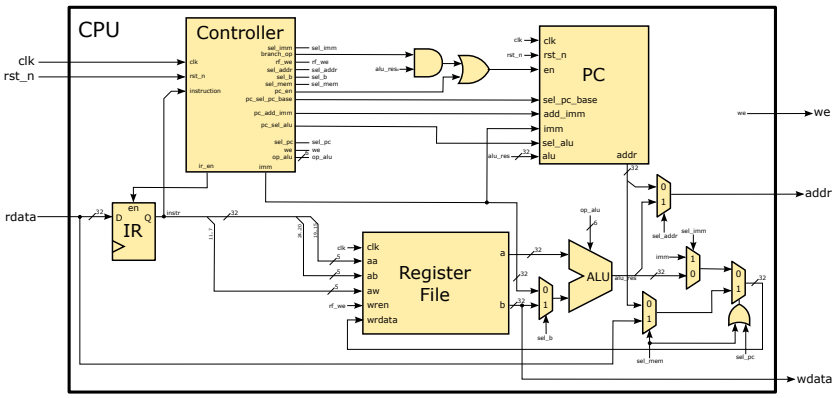

Figure 1: Original multicycle RISC-V CPU

The following figure 2 shows the modifications required for the architecture to handle interrupts. For handling interrupts, a new unit, called **CSR Controller**, is added to the architecture. The irq input signal captures the interrupt event. The Program Counter and Controller need to be modified in order to handle the interrupt events. When an interrupt occurs, the normal program flow should be stopped and the interrupt should be serviced by jumping to the appropriate interrupt service routine.

At the end of this lab, you will have a functional RISC-V CPU with interrupt support, and you will be able to run the debugger on `program.s` that you developed in the last part of the lab using this CPU. In case you did not manage to implement some components from Lab B, we have provided these components on Moodle. You are free to use any of the components we provide, but you are encouraged to use your own implementation.

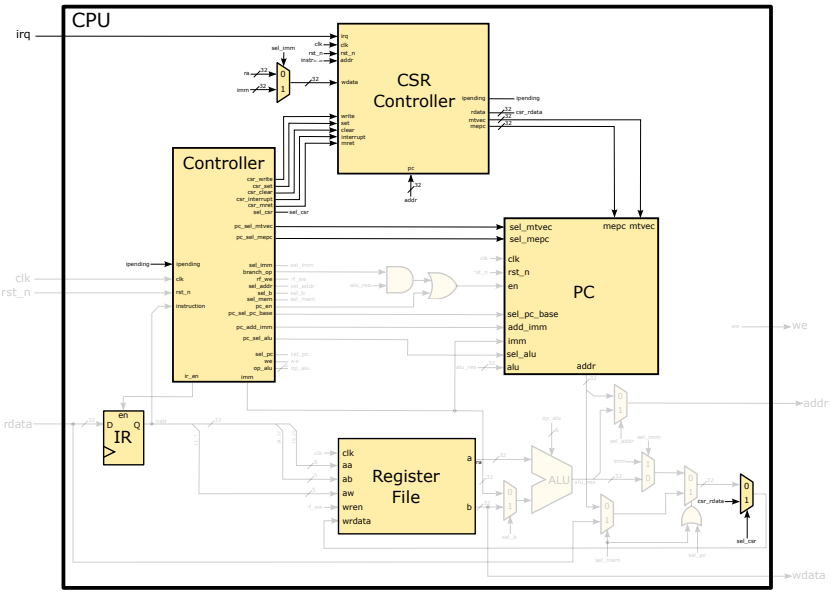

Figure 2: Multicycle RISC-V CPU with interrupt handling

2 Adding CSR Instructions

To support interrupt handling, we need to add Control and Status Register (CSR) instructions to our RISC-V CPU. The following schematic shows the signals of interest for this implementation:

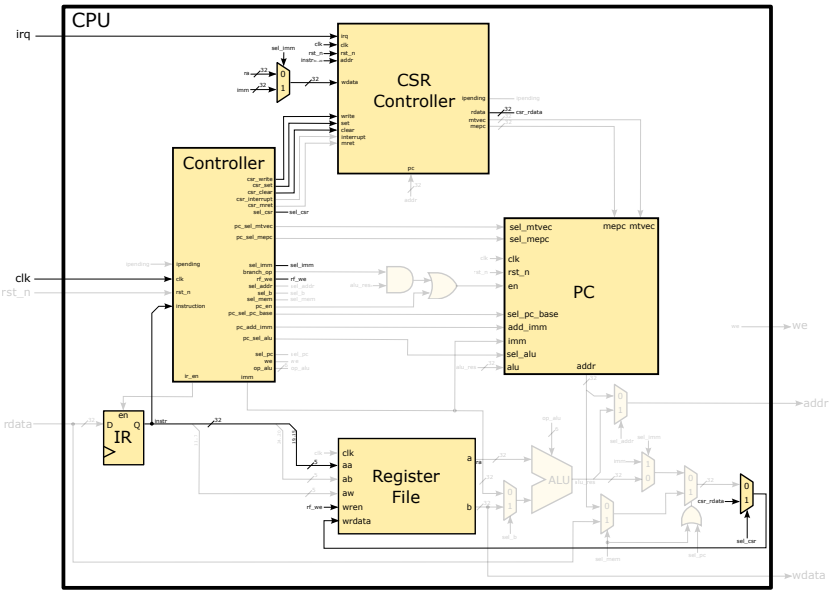

Figure 3: CPU schema focusing on CSR-related signals

CSR instructions are part of the SYSTEM instruction type in RISC-V. The layout for these instructions is as follows:

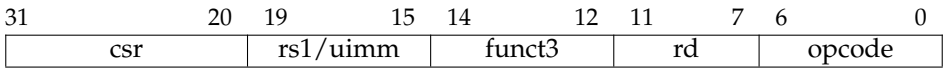

Figure 4: The SYSTEM instruction format for CSR operations in RV32I.

As a reminder, the opcode for SYSTEM instructions is 0x73 like EBREAK. The `funct3` field determines the specific CSR operation to be performed:

Instruction	funct3	Description
csrrw rd, csr, rs1	001	CSR Read/Write
csrrs rd, csr, rs1	010	CSR Read and Set Bits
csrrc rd, csr, rs1	011	CSR Read and Clear Bits
csrrwi rd, csr, uimm	101	CSR Read/Write Immediate
csrrsi rd, csr, uimm	110	CSR Read and Set Bits Immediate
csrrci rd, csr, uimm	111	CSR Read and Clear Bits Immediate

Table 1: CSR instructions in RV32I.

2.1 Register Write-back in CSR Instructions

It's important to note that CSR instructions always write to a destination register, even when that register is x0. This behavior differs from other RISC-V instructions where writes to x0 are typically ignored. The reasons for this are:

1. **Consistency:** CSR instructions always read the old value of the CSR before potentially modifying it. This old value is written to the destination register, providing a consistent way to access the previous state of the CSR.
2. **Side-effect free reads:** For the CSRs we're implementing (mtvec, mepc, mcause, etc.), reading does not cause any side effects. This means it's safe to always perform the read operation, even if the result might be discarded when writing to x0.
3. **Simplicity in hardware:** Always writing the old CSR value to the destination register simplifies the hardware implementation, as it doesn't need to check whether the destination is x0 before deciding to perform the write.

During the DECODE stage, if a SYSTEM type instruction is detected (different from a **ebreak** instruction), the controller should transition to a new CSR state in the next cycle. In the controller (`controller.v`), the following signals should be toggled during the CSR state:

- `sel_csr`: Set to 1 to select CSR data for writeback.
- `rf_we`: Set to 1 to enable writing to the register file.
- `csr_write`, `csr_set`, or `csr_clear`: Set based on the `funct3` field.
- `sel_imm`: Set to 1 if the instruction is an immediate type, 0 otherwise.
- `imm`: The immediate value for CSR instructions, **zero-extended** to 32 bits. Should be set to 0 if the instruction is not an immediate type.

After executing the CSR instruction, the controller should transition back to the FETCH2 state to continue normal execution as shown in Figure 5.

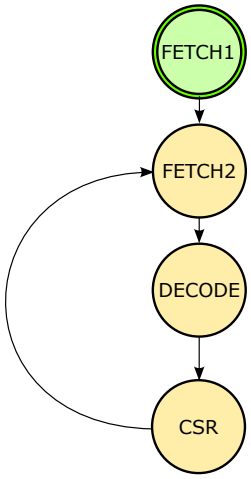

Figure 5: State machine for CSR instructions

2.2 Exercise

In this exercise, you will extend your RISC-V CPU implementation to support CSR instructions. This will involve modifying several modules from your previous work.

- Import the following modules from your previous labs:
  - `add_sub.v`
  - `alu.v`
  - `buttons.v`
  - `comparator.v`
  - `controller.v`
  - `csr.v`
  - `decoder.v`
  - `ir.v`
  - `leds.v`
  - `logic_unit.v`
  - `mem.sv`
  - `mux2x32.v`
  - `mux4x32.v`
  - `pc.v`
  - `register_file.v`
  - `seven_seg_lcd.v`
  - `shift_unit.v`
- Modify the `controller.v` module:
  - Add a new CSR state to the state machine.
  - Implement the logic to transition to the CSR state when a SYSTEM instruction (except `ebreak`) is detected.
  - In the CSR state, set the appropriate control signals as described earlier.
- **Important:** No testbenches are provided for this exercise. You must create your own testbenches to verify your implementation. You can also create some simple assembly programs to test the CSR instructions.

3 Program Counter

3.1 Description

The Program Counter (PC) module has been extended with four new inputs to support interrupt handling and returning from interrupts:

- `sel_mtvec` and `mtvec`: When `sel_mtvec` is set (and the PC is enabled), the PC should be updated with the value of `mtvec`. It's crucial to ensure that the two least significant bits (LSBs) of this value are set to zero to maintain memory alignment.
- `sel_mepc` and `mepc`: These operate similarly to `sel_mtvec` and `mtvec`. When `sel_mepc` is set (and the PC is enabled), the PC should be updated with the value of `mepc`. Since `mepc` stores a previous PC value, it's already properly aligned and doesn't require LSB adjustment.

These additions allow the PC to handle interrupt vector addresses and return addresses for interrupt service routines.

3.2 Exercise

- Extend the `pc` interface to include the new signals: `sel_mtvec`, `mtvec`, `sel_mepc`, and `mepc`.
- Implement the logic to update the PC based on these new inputs, ensuring proper alignment for `mtvec`.
- **Important:** No testbenches are provided for this exercise. You must create your own testbenches to verify your implementation.

4 Interrupt Handling

4.1 Description

When an interrupt occurs, the CPU needs to stop its current execution, save the necessary state, and jump to the interrupt handler. This process is primarily handled in the FETCH2 state of the controller.

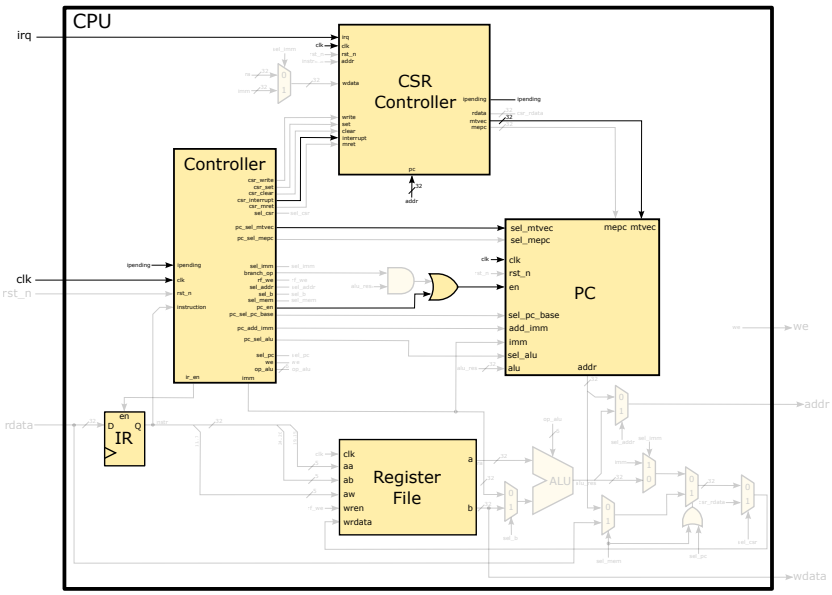

Figure 6: CPU schema focusing on interrupt handling signals

The interrupt handling process is triggered when the `ipending` signal is set. In the FETCH2 state, the controller checks this signal and, if set, initiates the interrupt handling sequence.

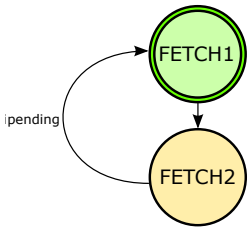

Figure 7: State machine for interrupt handling

When an interrupt is detected, the following signals are toggled in the controller:

- `csr_interrupt`: Set to 1 to indicate an interrupt is being handled.
- `pc_sel_mtvec`: Set to 1 to select the interrupt vector address.
- `pc_en`: Set to 1 to enable updating the program counter.

These signals cause the CPU to save the current program counter to the `mepc` CSR, update the `mcause` CSR with the interrupt cause, and jump to the interrupt handler address specified in the `mtvec` CSR.

4.2 Exercise

- Modify the `controller.v` module to handle the interrupt sequence in the FETCH2 state.
- Implement the logic to set the appropriate signals when an interrupt is detected.
- **Important:** Create your own testbench to verify the interrupt handling implementation.

5 Returning from an Interrupt

5.1 Description

To return from an interrupt, the RISC-V architecture provides the `mret` (Machine-mode Return) instruction. This instruction is part of the SYSTEM instruction type and is encoded as an I-type instruction.

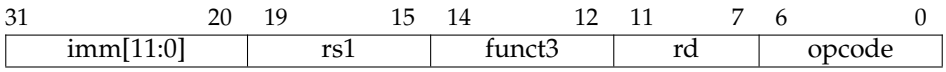

Figure 8: The `mret` instruction format in RV32I.

For the `mret` instruction, the fields have the following specific values:

Instruction	Immediate	funct3	opcode
`mret`	0x302	0x0	0x73

Table 2: Field values for the `mret` instruction.

It's important to note that while `mret` shares the same opcode (0x73) as CSR and EBREAK instructions, it should be distinguished from these during the decode phase. This can be done by checking the specific immediate value (0x302) and ensuring that funct3 is all zeros. This distinction allows the controller to correctly identify and handle the `mret` instruction separately from other SYSTEM instructions.

When the `mret` instruction is decoded, the controller transitions to the INTERRUPT_RETURN state.

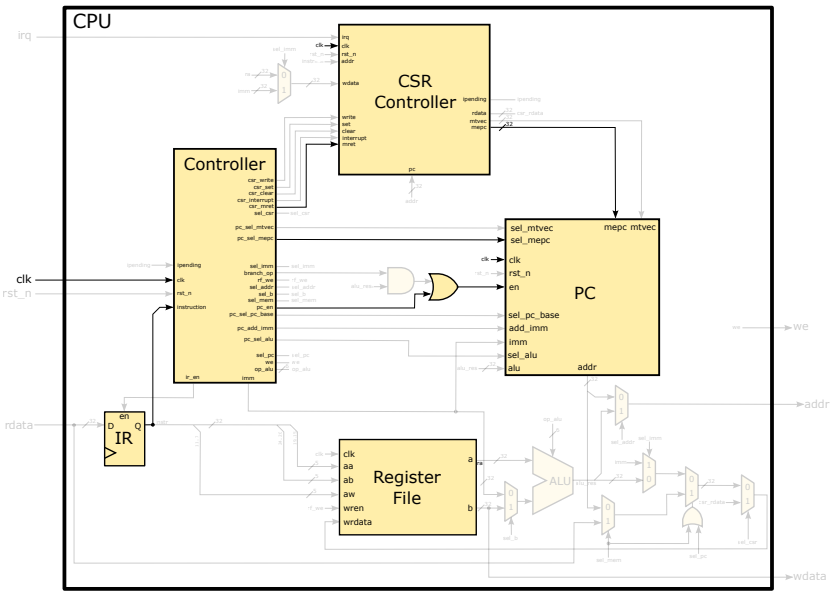

Figure 9: CPU schema focusing on interrupt return signals

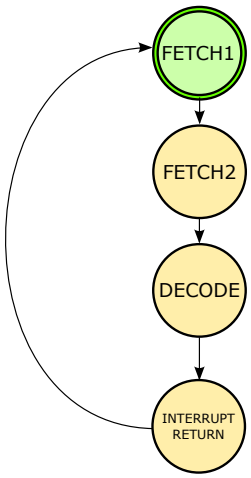

Figure 10: State machine for returning from an interrupt

In the INTERRUPT_RETURN state, the following signals are toggled in the controller:

- `csr_mret`: Set to 1 to indicate an `mret` instruction is being executed.
- `pc_sel_mepc`: Set to 1 to select the return address stored in `mepc`.
- `pc_en`: Set to 1 to enable updating the program counter.

These signals cause the CPU to restore the program counter from the `mepc` CSR, restore the interrupt enable bit in the `mstatus` CSR, and resume execution at the instruction that was interrupted.

After executing the `mret` instruction, the controller transitions back to the FETCH1 state to continue normal execution.

5.2 Exercise

- Extend the `controller.v` module to include the INTERRUPT_RETURN state.
- Implement the logic to detect the `mret` instruction and set the appropriate signals.
- Like the previous lab, you can debug the file `gecko.v` to run the `program.s` file with your own program to help you debug your own implementation.
- **Important:** Create your own testbench to verify the interrupt return implementation.

6 Submission

Submit all Verilog files related to the exercises in this lab, including:

- `add_sub.v`
- `alu.v`
- `buttons.v`
- `comparator.v`
- `controller.v`
- `cpu.sv`
- `csr.v`
- `decoder.v`
- `ir.v`
- `leds.v`
- `logic_unit.v`
- `mem.sv`
- `mux2x32.v`
- `mux4x32.v`
- `pc.v`
- `register_file.v`
- `seven_seg_lcd.v`
- `shift_unit.v`

Additionally, submit the `program.s` file from the previous assignment containing the interrupt handler implementation.

Ensure that all files are included in your submission, as they are necessary for the complete system evaluation. Also, remember to check linter errors on Verilog files either locally or through the preliminary grader. Failing linter test will result in the final grade of the lab being **halved**.

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 submission is a final one, so you will not receive any other feedback. At then end of the results you will get a score that will be used to grade your submission.