

EECS 388: Embedded Systems

6. Interrupt

Heechul Yun

Agenda

- Polling vs. interrupt
- Interrupt processing
- Interrupt related issues

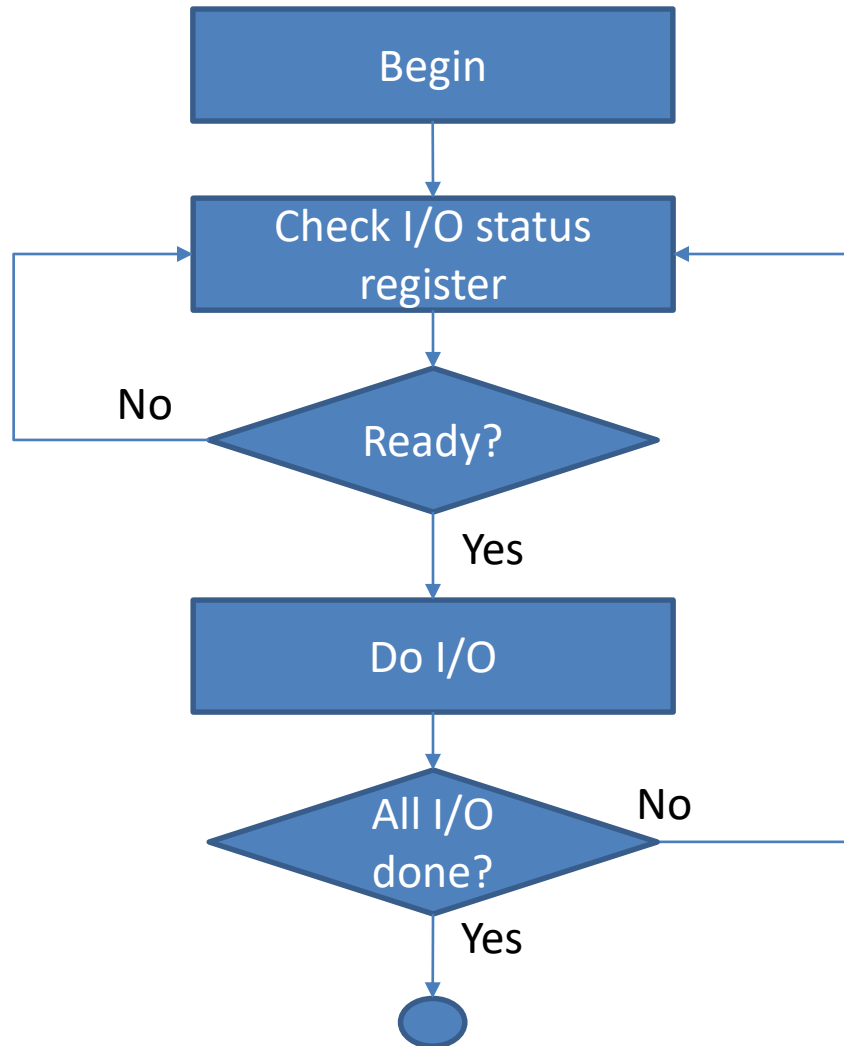
Polling

- Keep checking the status to see if I/O is ready

```
void ser_write(char c)
{
    uint32_t regval;
    /* busy-wait if tx FIFO is full */
    do {
        regval = *(volatile uint32_t *) (UART0_CTRL_ADDR + UART_TXDATA);
    } while (regval & 0x80000000);

    /* write the character */
    *(volatile uint32_t *) (UART0_CTRL_ADDR + UART_TXDATA) = c;
}
```

Polling



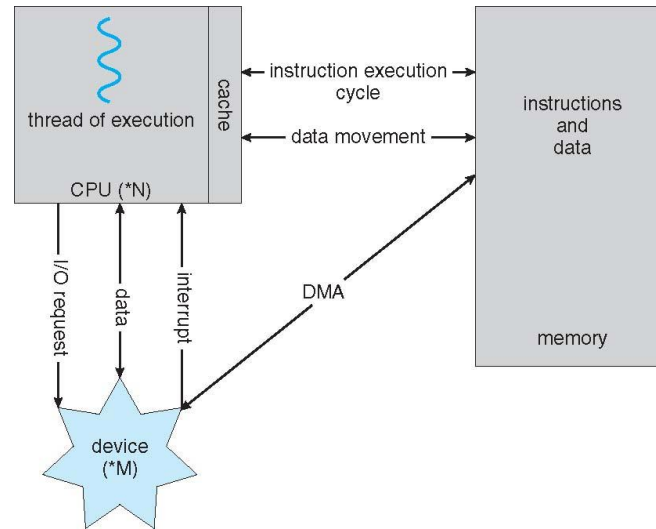
Polling

- The problem
 - CPU can't do any useful things in the meantime.
- Improvements
 - Check multiple I/O devices rather than one
 - Instead of keep checking the status (busy waiting), do some other useful work and then check again
 - Q. when to check again?
 - Q. what if the I/O must occur within a bound time?
- Can we do I/O immediately when it is ready?

Interrupt

- What is an interrupt?
 - A signal to the processor telling “do something now!”
- Hardware interrupts
 - Devices (timer, disk, keyboard, ...) to CPU
- Software interrupts
 - special instructions (e.g., int 0x80)
- Exceptions
 - Divide by zero, segmentation fault, ...

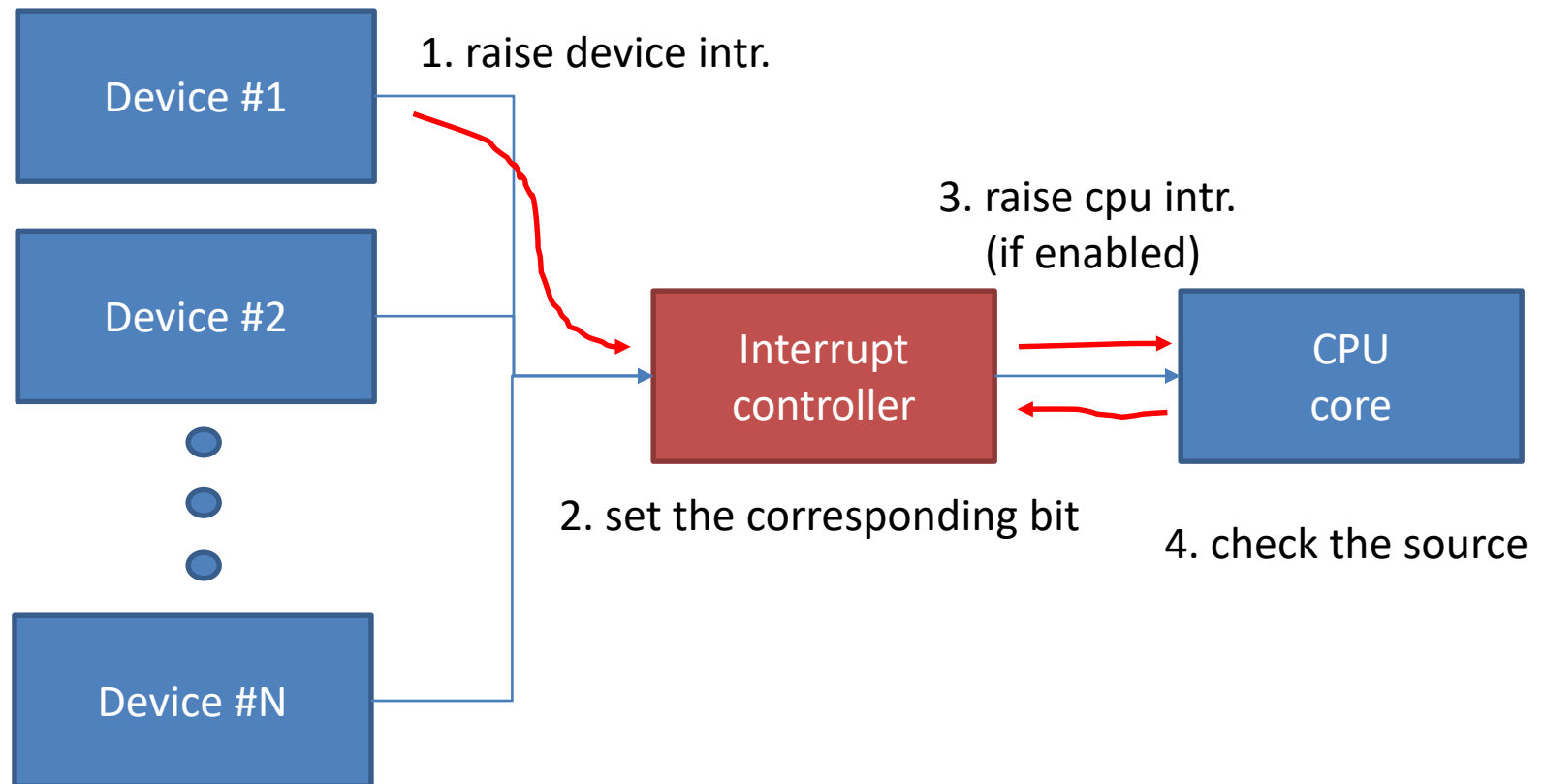
Interrupt Handling



- Step 1: deliver an interrupt to the CPU
- Step 2: save CPU states (registers)
- Step 3: execute the associated interrupt service routine (ISR)
- Step 4: restore the CPU states
- Step 5: return to the interrupted program

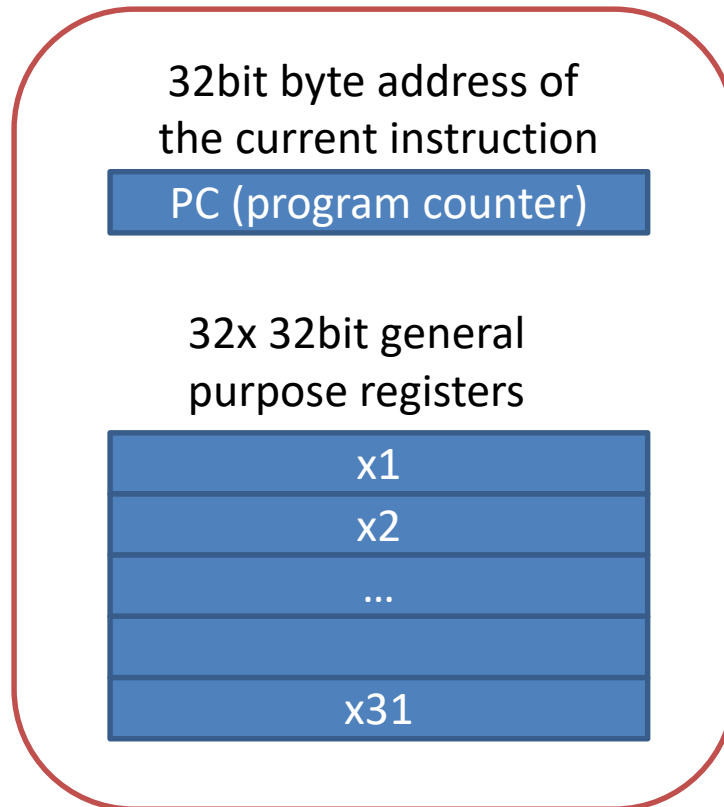
Interrupt Handling

- Interrupt delivery



Interrupt Handling

- Saving the CPU state



32bit address space
($2^{32} \times 8\text{bit} = 4\text{GB}$)



Interrupt Handling

- Interrupt Service Routine (ISR)

Disable interrupt

Save CPU registers

Irq_no = Find out the interrupt requester

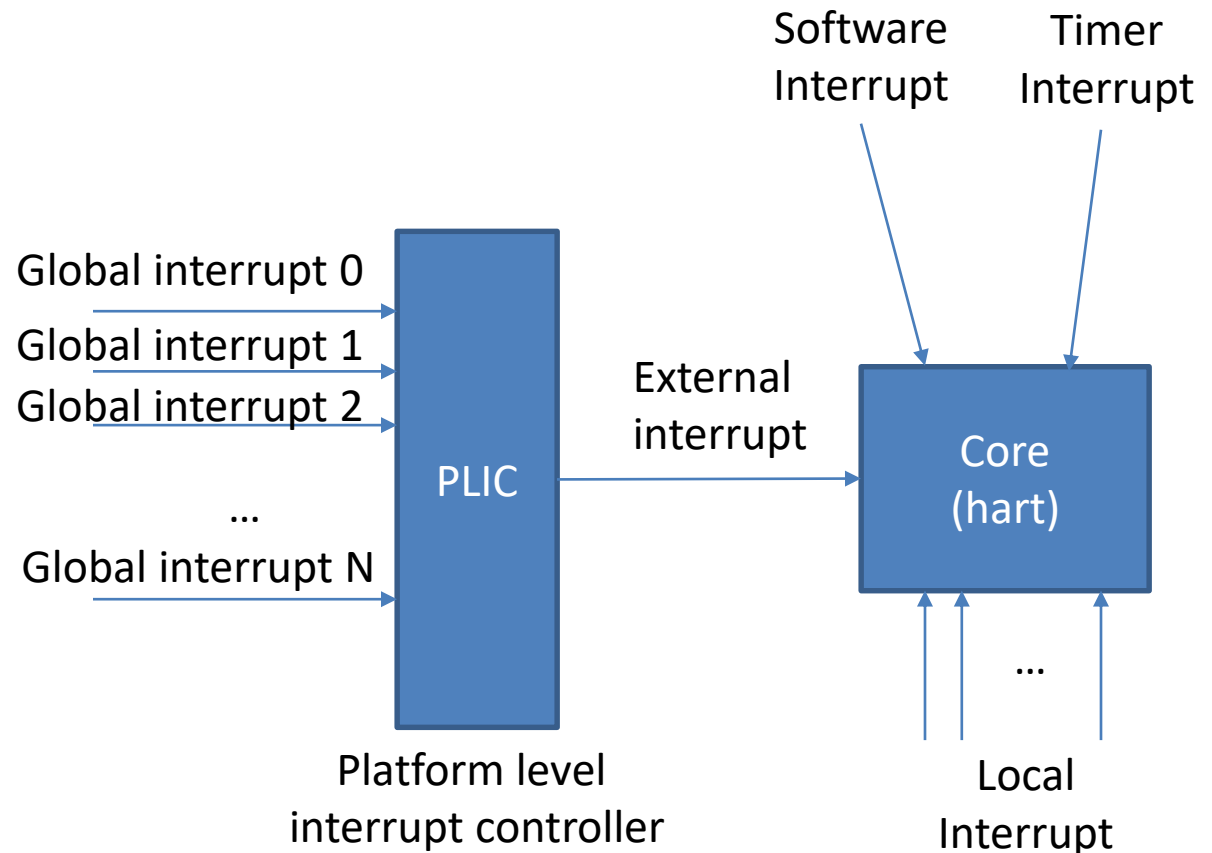
Jump to the interrupt_vector[irq_no]

Restore CPU registers

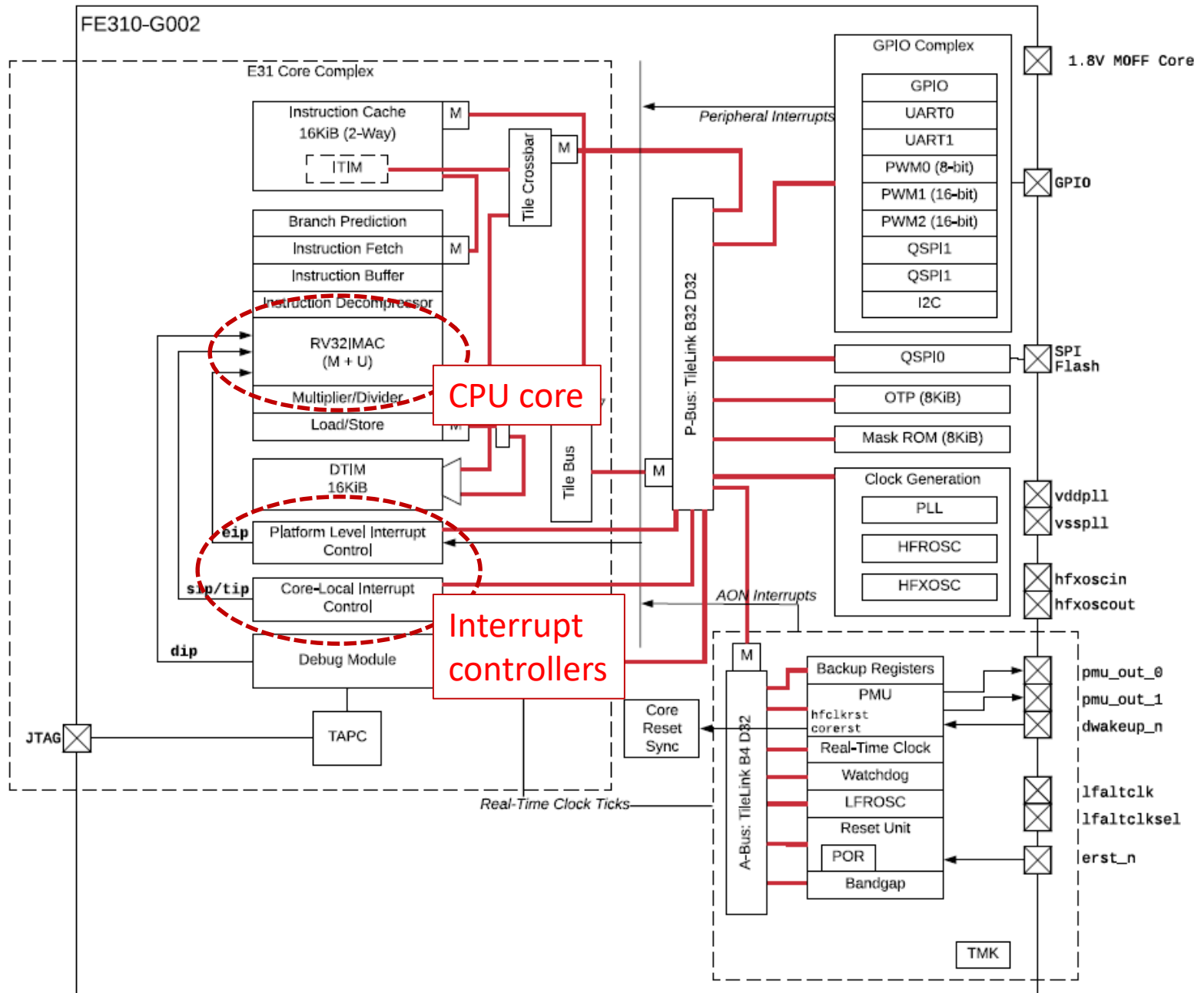
Re-enable interrupt

Case Study: RISC-V Interrupts

- Software
- Timer
- Local
- External



[Drew Barbier, "An Introduction to the RISC-V Architecture," 2019](#)



SiFive FE310 RISC-V CPU

Interrupt Related CSRs

- Control and status registers (CSRs)
 - Registers for software/hardware communication
 - Use special instructions to read/write
- mstatus
 - Global interrupt enable/disable
- mcause
 - Identify the cause of the interrupt
- mvec
 - Base address of interrupt handler(s)
- mtime
 - Architecturally defined constant speed time
- mtimecmp
 - Trigger interrupt when $mtime > mtimecmp$

Interrupt Related CSRs

- *mcause* CSR
 - Interrupt [31]
 - 1 = interrupt
 - 0 = exception
 - Code [30:0]
 - Exception code

Interrupt = 1 (interrupt)	
Exception Code	Description
0	User Software Interrupt
1	Supervisor Software Interrupt
2	Reserved
3	Machine Software Interrupt
4	User Timer Interrupt
5	Supervisor Timer Interrupt
6	<i>Reserved</i>
7	Machine Timer Interrupt
8	User External Interrupt
9	Supervisor External Interrupt
10	<i>Reserved</i>
11	Machine External Interrupt
12 - 15	<i>Reserved</i>
≥16	Local Interrupt X

Interrupt = 0 (exception)	
Exception Code	Description
0	Instruction Address Misaligned
1	Instruction Access Fault
2	Illegal Instruction
3	Breakpoint
4	Load Address Misaligned
5	Load Access Fault
6	Store/AMO Address Misaligned
7	Store/AMO Access Fault
8	Environment Call from U-mode
9	Environment Call from S-mode
10	Reserved
11	Environment Call from M-mode
12	Instruction Page Fault
13	Load Page Fault
14	Reserved
15	Store/AMO Page Fault
≥16	Reserved

Trap Handler Entry and Exit

- On entry
 - Save the current state
 - PC, privilege, interrupt enable
 - Disable interrupt ($mstatus.MIE = 0$)
 - Jump to **trap handler**
- On exit
 - Restore saved state
 - PC, privilege, interrupt enable
 - Jump to the stored PC

```
Push Registers
...
interrupt = mcause.msb
if interrupt
    branch isr_handler[mcause.code]
else
    branch exception_handler[mcause.code]
...
Pop Registers
MRET
```

Trap Handler

```
.align 2
.global trap_entry
trap_entry:
    addi sp, sp, -16*REGBYTES
    //store ABI Caller Registers
    STORE x1, 0*REGBYTES(sp)
    STORE x5, 2*REGBYTES(sp)
    ...
    STORE x30, 14*REGBYTES(sp)
    STORE x31, 15*REGBYTES(sp)
    //call C Code Handler
    call handle_trap
    //restore ABI Caller Registers
    LOAD x1, 0*REGBYTES(sp)
    LOAD x5, 2*REGBYTES(sp)
    ...
    LOAD x30, 14*REGBYTES(sp)
    LOAD x31, 15*REGBYTES(sp)
    addi sp, sp, 16*REGBYTES
    mret
```

```
void handle_trap()
{
    unsigned long mcause = read_csr(mcause);
    if (mcause & MCAUSE_INT) {
        //mask interrupt bit and branch to handler
        isr_handler[mcause & MCAUSE_CAUSE] ();
    } else {
        //branch to handler
        exception_handler[mcause]();
    }
}
//write trap_entry address to mtvec
write_csr(mtvec, ((unsigned long)&trap_entry));
```


PLIC External Interrupt Handler

```
void handle_trap(void) __attribute__((interrupt));
void handle_trap()
{
    unsigned long mcause = read_csr(mcause);
    if (mcause & MCAUSE_INT) {
        //mask interrupt bit and branch to handler
        isr_handler[mcause & MCAUSE_CAUSE] ();
    } else {
        //synchronous exception, branch to handler
        exception_handler[mcause & MCAUSE_CAUSE]();
    }
}

//install PLIC handler at MEIP Location
isr_handler[11] = machine_external_interrupt;
//write trap_entry address to mtvec
write_csr(mtvec, ((unsigned long)&handle_trap));

void machine_external_interrupt()
{
    //get the highest priority pending PLIC interrupt
    uint32_t int_num = plic.claim_complete;
    //branch to handler
    plic_handler[int_num]();
    //complete interrupt by writing interrupt number
    //back to PLIC
    plic.claim_complete = int_num;
}
```

PLIC Interrupts on FE310 (HiFive1)

Source Start	Source End	Source
1	1	AON Watchdog
2	2	AON RTC
3	3	UART0
4	4	UART1
5	5	QSPI0
6	6	SPI1
7	7	SPI2
8	39	GPIO
40	43	PWM0
44	47	PWM1
48	51	PWM2
52	52	I2C

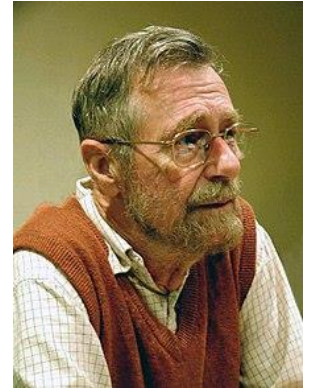
Table 26: PLIC Interrupt Source Mapping

Problems with Interrupts

- Timing
- Concurrency

Interrupts are Evil

“[I]n one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer’s grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature.”



(Dijkstra, “The humble programmer” 1972)

<https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html>

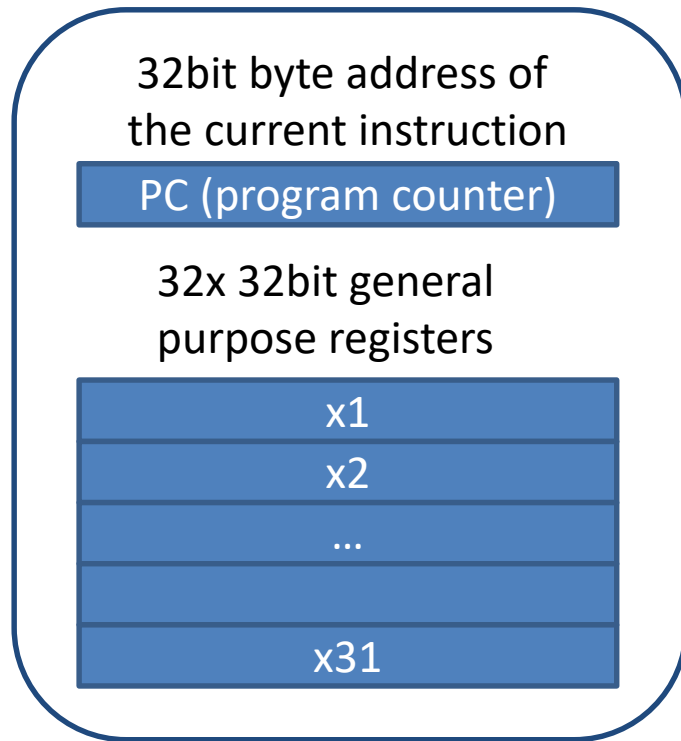
Timing Issues

- When to occurs?
- For how long?
- How many interrupts over time?
- Generally the answers are all “don’t know”
- What if the interrupted code has real-time requirements?

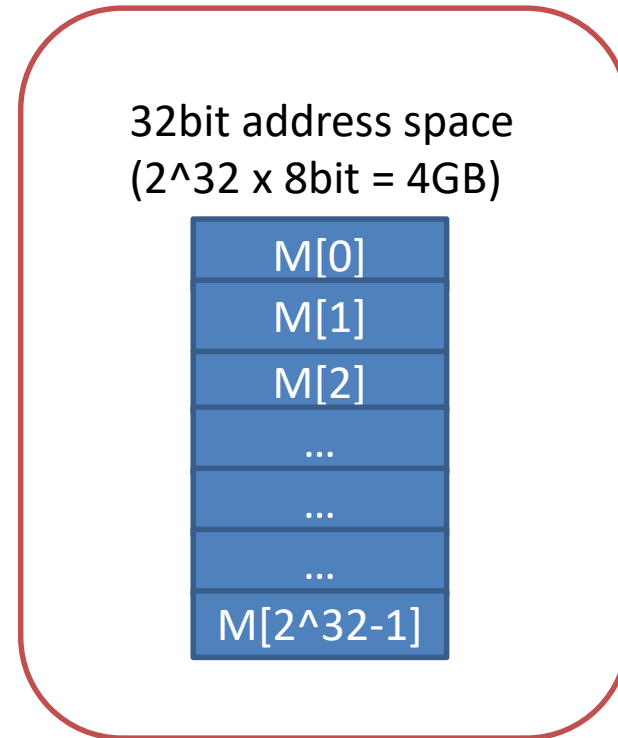
Timing Issues

- General guideline
 - ISR should be kept as small as possible.
 - E.g., move data from the device's buffer to memory
 - Heavy duty work should be done later
 - Know your context switching overhead
 - Direct overhead: register save/restore
 - Indirect overhead: cache pollution (if caches are used)
 - Polling can be better sometimes
 - If interrupts occur too frequently, polling can reduce context switching overhead and improve throughput

Concurrency



Private: No problem



Shared: Big problem

- Memory is shared between the ISR and the interrupted code: they can mess with each other

Example

```
int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

Q. What does this program intend to do?

Example

```
int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

Q. Will it work?

Recall: Volatile in I/O

```
void ser_write(char c)
{
    uint32_t regval;
    /* busy-wait if tx FIFO is full */
    do {
        regval = *(volatile uint32_t *) (UART0_CTRL_ADDR + UART_TXDATA);
    } while (regval & 0x80000000);

    /* write the character */
    *(volatile uint32_t *) (UART0_CTRL_ADDR + UART_TXDATA) = c;
}
```



```
ser_write:
    li    a4,268513280
.L17:
    lw    a5,0(a4)
    bltz  a5,.L17
    sw    a0,0(a4)
    ret
```

a4 = 0x10013000

a5 = *a4

Branch to .L17 if a5 < zero

Needed **volatile** to inform the compiler that the value of the variable may be changed by the hardware

Volatile

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

Need volatile to inform the compiler that the value of the variable may be changed “externally” (by the interrupt handler)

Improved Example


```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

Q. Is this now correct?

Example

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

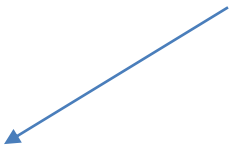
What happen if interrupt 10 occurs immediately after interrupt 9 finishes?



Example

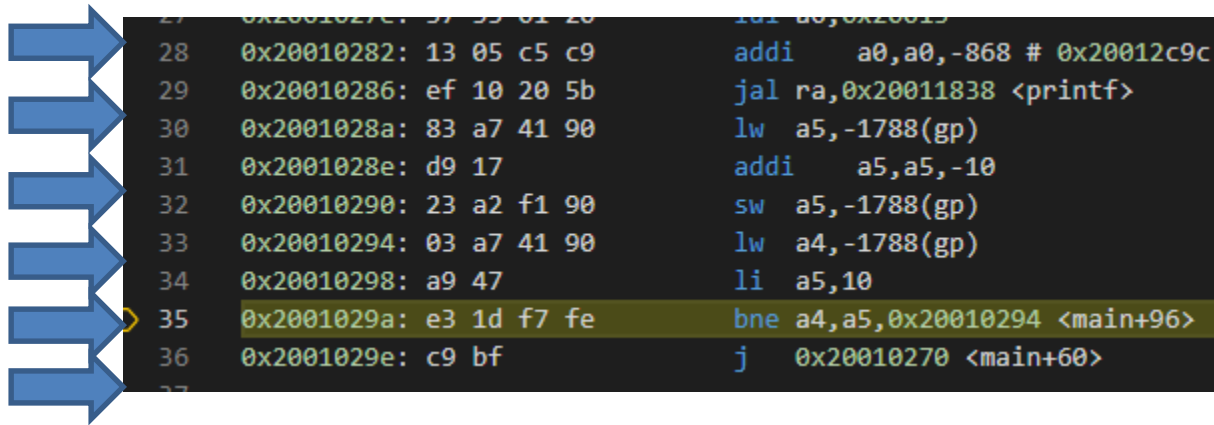
```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

What happen if an interrupt occurs here?



Concurrency

- An interrupt can occur at any instruction



```
27 0x2001027c: 57 55 01 20      lui    a0,0x2001027c
28 0x20010282: 13 05 c5 c9      addi   a0,a0,-868 # 0x20012c9c
29 0x20010286: ef 10 20 5b      jal   ra,0x20011838 <printf>
30 0x2001028a: 83 a7 41 90      lw    a5,-1788(gp)
31 0x2001028e: d9 17           addi   a5,a5,-10
32 0x20010290: 23 a2 f1 90      sw    a5,-1788(gp)
33 0x20010294: 03 a7 41 90      lw    a4,-1788(gp)
34 0x20010298: a9 47           li    a5,10
35 0x2001029a: e3 1d f7 fe      bne   a4,a5,0x20010294 <main+96>
36 0x2001029e: c9 bf           j     0x20010270 <main+60>
```

Example

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

What happen if interrupt 10 occurs immediately after interrupt 9 finishes?

..., 9, 10, 11, ...

Non-deterministic outcome

Example

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        if (count == 10){
            printf("reset the counter");
            count -= 10;
        }
    }
}
```

What happen if an interrupt occurs here?

0, 1, ..., 10
1, 2, ..., 10

...

Non-deterministic outcome

Atomicity

- Single C statement can be translated into multiple assembly instructions
- Single instruction (e.g., LDM/STM in ARM) can be multiple atomic operations in hardware
- Interrupt can occur between any two atomic operations

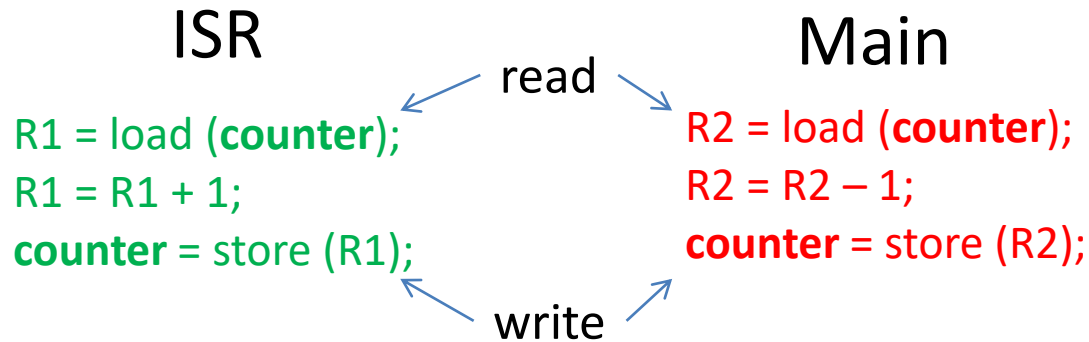
```
count64 = 0;          li  a5, 0
                      sw  a5, -1784(gp)
                      sw  a6, -1780(gp)
                      li  a6, 0
```

```
count++;             lw   a5, %lo(count) (a4)
                      addiw a5, a5, 1
                      sw   a5, %lo(count) (a4)
```

```
STMDB sp!, {r4, r5, r6, r7, r8, r9, sl, lr}
```

Race Condition

- A situation when two or more threads **read and write** shared data at the same time
- Correctness depends on the execution order



- How to prevent race conditions?

Synchronization

- How to protect shared variables?
 - Between ISRs and the main program
- Solutions
 - Be very careful 😊
 - Single writer (write at only ISR or Main)
 - Disable interrupt
 - When the main reads/writes the shared variables

Improved Example

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        disable_interrupt();
        if (count >= 10){
            printf("reset the counter");
            count = 0;
        }
        enable_interrupt();
    }
}
```

Interrupt in Interrupt Handler?

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        disable_interrupt();
        if (count >= 10){
            printf("reset the counter");
            count = 0;
        }
        enable_interrupt();
    }
}
```

What happen if an interrupt occurs here?

```
lw      a5,%lo(count) (a4)
addiw   a5,a5,1
sw      a5,%lo(count) (a4)
```

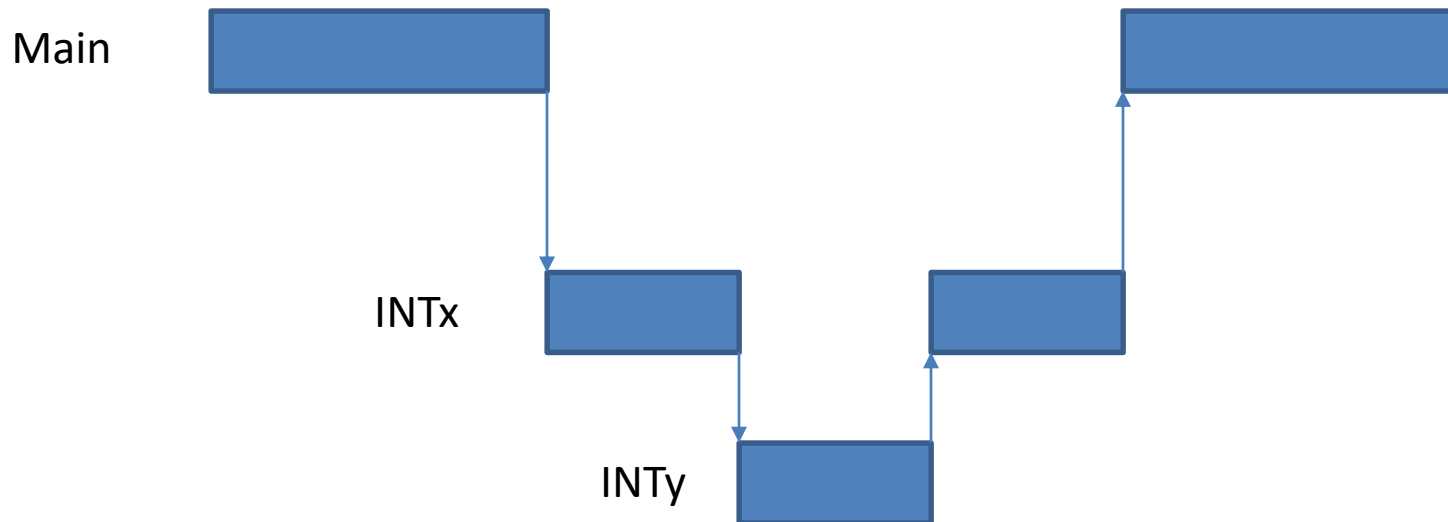
Example

```
volatile int count = 0;
void ISR(void) {
    count++;
    print("count=%d\n", count)
}
int main(void) {
    // install ISR
    ...
    // main code
    while(1){
        disable_interrupt();
        if (count >= 10){
            printf("reset the counter");
            count = 0;
        }
        enable_interrupt();
    }
}
```

What happen if an interrupt occurs here?

In general, when an ISR is executed the hardware/OS already disabled the interrupt... BUT

Nested Interrupt



- High priority interrupt preempts low priority one. But not the other way around.
- Not all HW platforms support nested interrupts.

Interrupt on RISC-V/HiFive1

- Priority
 - External interrupts (highest)
 - Software interrupts
 - Timer interrupts (lowest)
 - (doesn't preempt but serviced in priority order)
- Latency
 - Signal → first instruction of the handler: 4 cycles
 - PLIC routing: 3 cycles
 - Total = 4 + 3 = 7 cycles (best-case scenario)

Summary

- Interrupts
 - “Do something right now!!!”
 - A hardware mechanism to handle urgent matters
 - Preempt whatever the CPU was currently doing
 - Introduce **concurrency** in sequential code
 - Evil (= because concurrency is hard to do it right)
 - Synchronization between the main program and the interrupt service routines are needed.

Acknowledgements

- Some slides were adopted from the materials originally developed by
 - Edward A. Lee and Prabal Dutta (UCB) for EECS149/249A