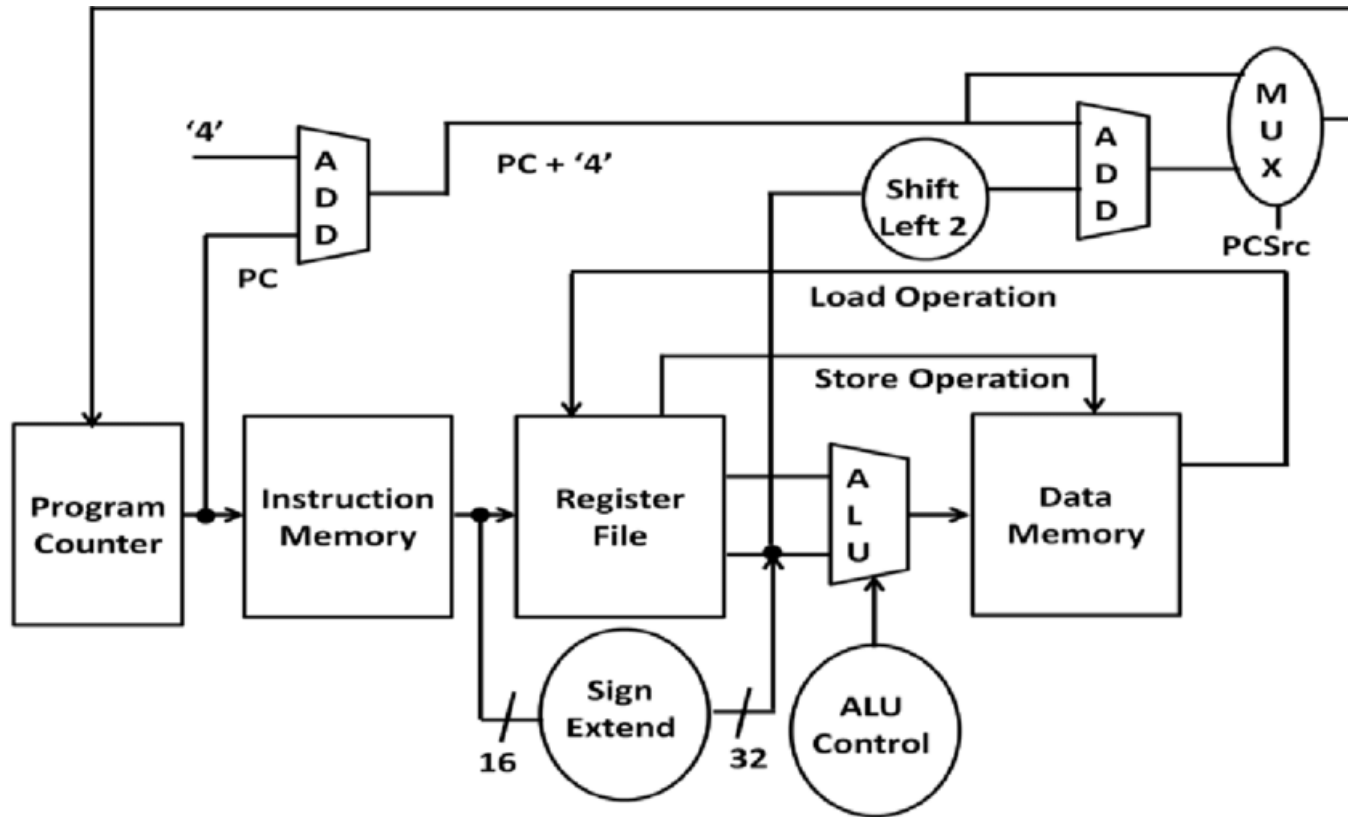


Interrupt

Recap: How CPU works?



```
fact:    addi $1, $0, 1
        beq $4, $0, return1
        bne $4, $1, continue
return1:    addi $2, $1, 0
            jr $31
```

```
# initialize reg. 1 to 1
# if (n == 0) then goto return1:
# if (n != 1) then goto continue:
# assign result = 1
# return
```

```
continue: addi $29, $29, -12
          sw $30, 8($29)
          sw $31, 4($29)
          addi $30, $29, 8
          sw $4, 0($29)
          addi $4, $4, -1
          jal fact
          lw $4, 0($29)
          mult $2, $4
          mflo $2
          lw $31, 4($29)
          lw $30, 8($29)
          addi $29, $29, 12
          jr $31
```

```
# allocate stack space for fp, ra, n
# save frame pointer
# save return address
# update frame pointer
# save n
# make n-1
# recursive call to fact(n-1)
# restore n
# fact (n-1) * n
# put product in result reg.
# restore return address
# restore frame pointer
# restore stack pointer
# return
```

Polling vs Interrupt

In the past few weeks, we usually need to continuously monitor (poll) the status of some devices or some register bits. This process wastes MCU cycle and may prevent us from handle other events on time.

```
// wait for button press
while (PINC & (1 << PINC0)); ← Polling


// do something

// wait for button release
while (!(PINC & (1 << PINC0))); ← Polling
```

Polling vs Interrupt

In the interrupt method, devices/peripherals inform the MCU via an interrupt flag or an interrupt signal through an interrupt pin. The MCU jumps to execute the interrupt service routine (ISR) and continue from where it was stopped after finished execute the ISR.

```
ISR (INT0_vect) {  
    // some instructions  
}  
  
int main() {  
    a();  
    b();  
    c();  
}
```



The diagram illustrates the execution flow of an interrupt. An orange line starts from the `a();` line in the `main()` function, goes right, then up, then left, ending with an arrow pointing to the `// some instructions` line inside the `ISR (INT0_vect) {` block. Another orange line starts from the closing brace of the `ISR` block, goes left, then down, then left, ending with an arrow pointing to the `b();` line in the `main()` function, indicating the return from the interrupt service routine.

AVR Interrupt: Enable/Disable

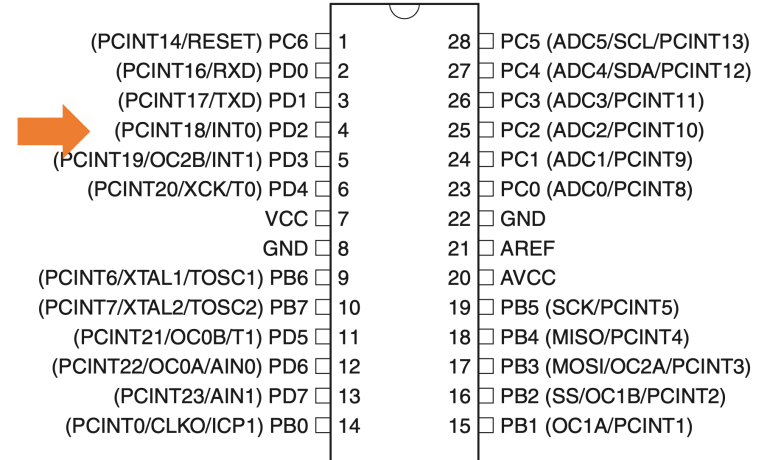
- All interrupts are disabled after reset.
- Interrupts can be enabled or disabled by
 1. set or clear the **I** bit (Global Interrupt Enable) in the SREG (Status Register) using the sei() and cli() command
 2. set or clear the enable bit of each interrupt e.g. INT[1:0], SPIE, RXCIE, TXCIE, ADIE
 3. If the **I** bit is 0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high

AVR Interrupt: Sample Code

```
ISR (INT0_vect) {  
    // some instructions  
}
```

```
int main() {  
    PORTD |= (1 << PORTD2);  
  
    // enable external interrupt 0  
    EIMSK |= (1 << INT0);  
    // enable interrupt  
    sei();  
  
    while (1) { // do other things }  
}
```

External Interrupt 0 Pin



ATMega328P Interrupt Vector

Table 12-6. Reset and Interrupt Vectors in ATmega328 and ATmega328P

VectorNo.	Program Address ⁽²⁾	Source	Interrupt Definition
1	0x0000 ⁽¹⁾	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI_STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART, Data Register Empty
21	0x0028	USART_TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE_READY	EEPROM Ready
24	0x002E	ANALOG_COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM_Ready	Store Program Memory Ready

← Highest priority

← Lowest priority

How AVR handle interrupt?

1. The CPU finishes the current executing instruction and push the current PC address (address of the next instruction) to the stack
2. The MCU jumps to a fixed location in the interrupt vector table (inside of the program memory) which contain an instruction to jump to the interrupt service routine (ISR)
3. The MCU executes the ISR until it reaches the RETI instruction
4. The MCU pops the PC address from the stack and starts execute from that address

Sample AVR Interrupt Assembly Code

```
.INCLUDE "M32DEF.INC"
.ORG 0                                ;location for reset
    JMP    MAIN
.ORG 0x02                             ;location for external interrupt 0
    JMP    EX0_ISR
MAIN: LDI    R20,HIGH(RAMEND)
    OUT     SPH,R20
    LDI     R20,LOW(RAMEND)
    OUT     SPL,R20                ;initialize stack
    LDI     R20,0x2                ;make INT0 falling edge triggered
    OUT     MCUCR,R20
    SBI     DDRC,3                ;PORTC.3 = output
    SBI     PORTD,2              ;pull-up activated
    LDI     R20,1<<INT0         ;enable INT0
    OUT     GICR,R20
    SEI                                ;enable interrupts
HERE: JMP    HERE
EX0_ISR:
    IN      R21,PORTC
    LDI     R22,0x08             ;00001000 for toggling PC3
    EOR     R21,R22
    OUT     PORTC,R21
    RETI
```

AVR Interrupt: Characteristics

- Several interrupt sources e.g. pin state change, SPI transfer complete, ADC conversion complete
- The individual interrupt enable bit and the **I** bit must be set to enable interrupt
- When an interrupt occurs, the **I** bit is cleared and all interrupts are disabled. The **I** bit will automatically be set when the RETI instruction is executed.
- **I** bit can be set in the ISR to enable nested interrupts

AVR Interrupt: Characteristics

- When the **I** bit is set while one or more interrupt flags are set, the corresponding ISR will be executed by order of priority
- When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served.
- When using the CLI instruction, the interrupts will be immediately disabled. No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with the CLI instruction