

CSSE2010/CSSE7201

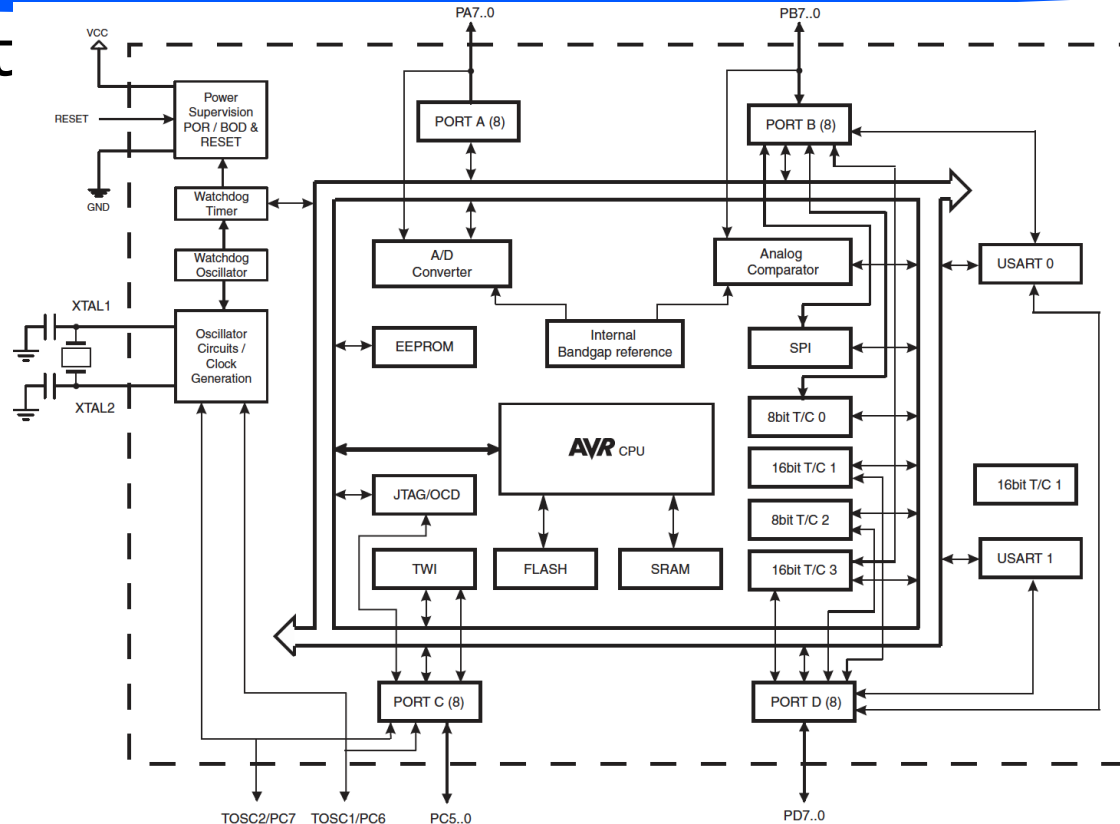
Lecture 17

Interrupts

School of Information Technology and Electrical Engineering
The University of Queensland

Today

- Today – Input/Output
 - Stacks revisited
 - Polling
 - Interrupts



AVR Stack

- Instructions:
 - **push Rr**
 - Value in register Rr place on top of stack
 - $SP \leftarrow SP - 1$ (top of stack pointer adjusted)
 - **pop Rd**
 - Value on top of stack placed in register Rd
 - $SP \leftarrow SP + 1$ (stack pointer adjusted)
- Procedure call: **call, rcall, icall, ret** instructions
 - e.g. call instruction: **call label**
 - Program counter value (return address) placed on top of stack
 - $SP \leftarrow SP - 2$ (since 16 bits needed for return address)
 - $PC \leftarrow$ label value

Function call - example

What values would be in r17 and r18 after this code?

17% **A.** 17: 0 r18: 20

17% **B.** r17: 0 r18: 33

17% **C.** r17: 20 r18: 20

17% **D.** r17: 20 r18: 33

17% **E.** r17: 33 r18: 20

17% **F.** r17: 33 r18: 33

```
ldi r17, 20
ldi r18, 33
push r17
push r18
pop r17
clr r17
pop r17
```

CPU Interacting with I/O

- CPU operates faster speed than I/O and I/O can be sometimes asynchronous
- CPU needs to know
 - When data is available from input devices
 - When output devices are ready to accept data
- Two options:
 - **Polling** driven I/O
 - **Interrupt** driven I/O

Polling the Input(s)

- Processor periodically asks devices and waits for responses
- Inefficient (wasting time asking and waiting!)
 - But may be OK if the processor has nothing else to do

Interrupts

- Device(s) tells CPU when it is ready
- Examples
 - Device ready to accept more output
 - Serial transmit buffer is empty
 - Device has input ready
 - Key pressed on keyboard
 - Regular interrupt
 - Timer generates interrupt every e.g. 20ms
- More efficient for CPU
- Changes the flow of control

Interrupts

- CPU receiving interrupt causes it to execute software to handle the interrupt
 - Software routine is called
 - **interrupt handler**, or
 - **interrupt service routine** (ISR)
 - When the handler completes, control must be returned where we left off
- Issues
 1. How does the CPU know where the ISR is?
 2. All registers (*including status register*) must have their values preserved – How?
 3. What happens if another interrupt arrives whilst we're servicing an interrupt?

Finding the Interrupt Handler

- Based on interrupt source, determine address of ISR for that interrupt
 - Look in **interrupt vector table** (often at beginning of memory).
Table contains:
 - table of addresses, or
 - table of jump instructions (AVR does it this way)
- Jump to address
- Note: vector sometimes refers to the interrupt number (index into the vector table) and sometimes to the address of the interrupt handler itself

Finding the Interrupt Handler

AVR Interrupts

- For the ATmega324A (from datasheet, pages 69-70)

<i>Vector</i>	<i>Program Addr</i>	<i>Source</i>	<i>Interrupt Definition</i>
1	0x0000	RESET	RESET, Watchdog Reset, etc.
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
	...		
11	0x0014	TIMER2_COMPB	Timer/Counter2 Compare Match B
12	0x0016	TIMER2_OVF	Timer/Counter2 Overflow
	...		
20	0x0026	SPI_STC	SPI Serial Transfer Complete
21	0x0028	USART0_RX	USART0 Receive Complete
22	0x002A	USART0_UDRE	USART0 Data Register Empty
23	0x002C	USART0_TX	USART0 Transmit Complete
24	0x002E	ANALOG_COMP	Analog Comparator
	...		

Preserving Register Values

- If an ISR uses registers
 - save copies of those registers before using them
 - restore values afterwards (before returning from interrupt)
- In particular, status register must be saved
 - Many CPUs do this automatically
 - AVR does not – must do so manually (if status register would be changed by handler)
- Called **transparency**
 - When ISR finishes, computer is in same state as it was before the interrupt

Interrupt Summary – Sequence of Actions

- Hardware
 1. Device asserts interrupt line
 2. CPU determines source of interrupt
 3. Push PC (return address) onto stack
 - Some CPUs push status register as well, but not AVR
 4. Determine Address of ISR (eg. From Interrupt Vector Table)
 5. CPU puts address of ISR into PC
 6. CPU disables interrupts

Interrupt Summary – Sequence of Actions

- Software (ISR)
 7. Save all registers to be used
 8. Determine source of interrupt (if this is not done by hardware)
 9. Handle the interrupt by executing the ISR code (eg. Do required input or output)
 10. Restore Registers
 11. Execute return from interrupt instruction

Interrupt Summary – Sequence of Actions

- Back to Hardware
 - 12. Interrupts are re-enabled
 - 13. CPU pops PC off the stack and restores it
 - 14. Execution continues from where it was “interrupted”

Instructions which change the Status Register

- How do you know which instructions change the status register?
 - See the Instruction Set Reference

Template AVR ISR

- `isr_label:`
 - `; Save any registers used`
 - `push r0`
 - `in r0, SREG`
 - `push r0`
 -
 - `; ... perform operation`
 - `; ... (usually Input/output)`
 -
 - `; Restore registers`
 - `pop r0`
 - `out SREG, r0`
 - `pop r0`
 - `; return from interrupt`
 - `reti`

CSSE2010

Which of these is a suitable interrupt service routine which reads a byte from port B pins and outputs it to port C?

(a)

```
.def temp = r16
int0: push temp
      in temp, SREG
      push temp
      in temp, PINB
      out PORTC, temp
      reti
```

(b)





```
.def temp = r16
int0: push temp
      in temp, SREG
      push temp
      in temp, PINB
      out PORTC, temp
      pop temp
      pop temp
      out SREG, temp
      reti
```

(c)

```
.def temp = r16
int0: push temp
      in temp, SREG
      push temp
      in temp, PORTB
      out PINC, temp
      pop temp
      out SREG, temp
      pop temp
      reti
```

(d)

```
.def temp = r16
int0: push temp
      in temp, PINB
      out PORTC, temp
      pop temp
      reti
```

0%	0%	0%	0%
			
(a)	(b)	(c)	(d)

10

AVR Interrupts in C

- `#include <avr/interrupt.h>`
- Interrupt handler can be written as a C function, using a special ISR macro, e.g.

```
ISR(INT0_vect) {  
    ... /* code here */  
}
```
- This is the handler for the INT0 external interrupt
 - Interrupt vector table setup and register saving/restoring happen automatically
 - Use source name from datasheet followed by `_vect`
- See AVR LibC manual – `avr/interrupt.h` documentation

Setting up interrupts

Besides the handler, also need to:

- Set up conditions for interrupt
- Enable that particular interrupt
 - An I/O register bit for each interrupt controls whether that interrupt is enabled or not
- Clear the specific interrupt flag (to ensure interrupt doesn't trigger immediately)
 - Usually done by writing **1** to some I/O register bit
- Turn on interrupts globally
 - `sei()` macro (same as `sei` assembly language instruction)

Volatile variables in C

- Example:

```
volatile uint8_t count;
```

- Where a variable can be changed outside the normal flow of a program (e.g. in an interrupt handler) it should be declared as volatile
 - Prevents code optimiser from assuming variable value doesn't change and optimising code away

Nested Interrupts

- Chance that another device interrupts during ISR execution
 - Could simply disable subsequent interrupts during the ISR
 - Usually, the global interrupt enable flag is one of the ones restored by the return from interrupt instruction
 - Better to prioritise interrupts, and allow higher priority interrupts to interrupt lower priority ones (and not vice versa, or at the same level)
 - Many CPUs support this (not AVR)

AVR Interrupt Priority

- Determined by vector: lowest vector is highest priority

<i>Vector</i>	<i>Program Addr</i>	<i>Source</i>	<i>Interrupt Definition</i>
1	0x0000	RESET	RESET, Watchdog Reset, etc.
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
	...		
11	0x0014	TIMER2_COMPB	Timer/Counter2 Compare Match B
12	0x0016	TIMER2_OVF	Timer/Counter2 Overflow
	...		
20	0x0026	SPI_STC	SPI Serial Transfer Complete
21	0x0028	USART0_RX	USART0 Receive Complete
22	0x002A	USART0_UDRE	USART0 Data Register Empty
23	0x002C	USART0_TX	USART0 Transmit Complete
24	0x002E	ANALOG_COMP	Analog Comparator
	...		

AVR Interrupt Priority (cont.)

- Priorities only matter when two (or more) interrupts are waiting to be serviced
 - Highest priority (lowest vector) will be serviced first
- Interrupts are disabled during ISR
 - re-enabled by RETI instruction
- Possible to enable interrupts during ISR
 - Priorities not relevant here
 - Any incoming (or pending) interrupt may be serviced - even if lower priority

Traps

- Traps are *software interrupts*, i.e. caused by events in software rather than hardware
 - *Exceptional* events
- Examples:
 - Overflow
 - Divide by zero
 - Undefined opcode
- Traps save continually checking for errors
- Trap handlers (service routines) don't always return to the original program
- AVR doesn't provide traps
 - but is possible to generate hardware interrupts from software