嵌入式微處理機系統

04_CPUs

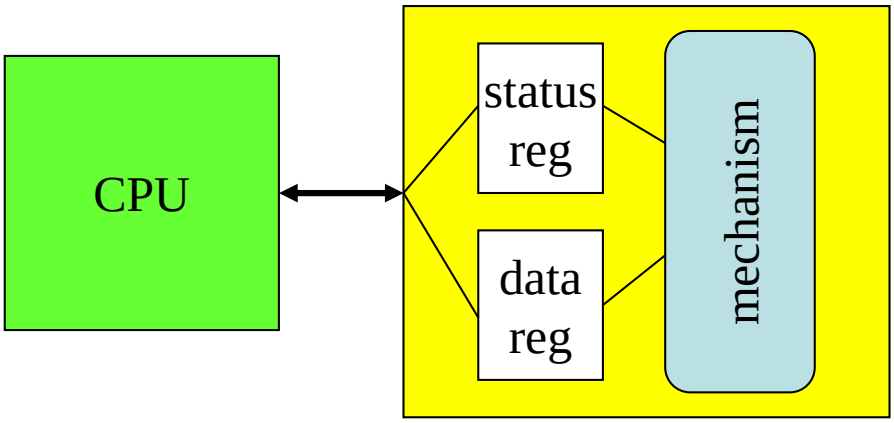
國立台灣科技大學電機系
王乃堅老師
Feb. 2024

CPUs

- Input and output
- Supervisor mode, exceptions, traps
- Co-processors
- Memory management and address translation
- Cache
- How architecture affects program performance
- How architecture affects program power consumption

I/O devices

- Usually includes some non-digital component.
- Typical digital interface to CPU:



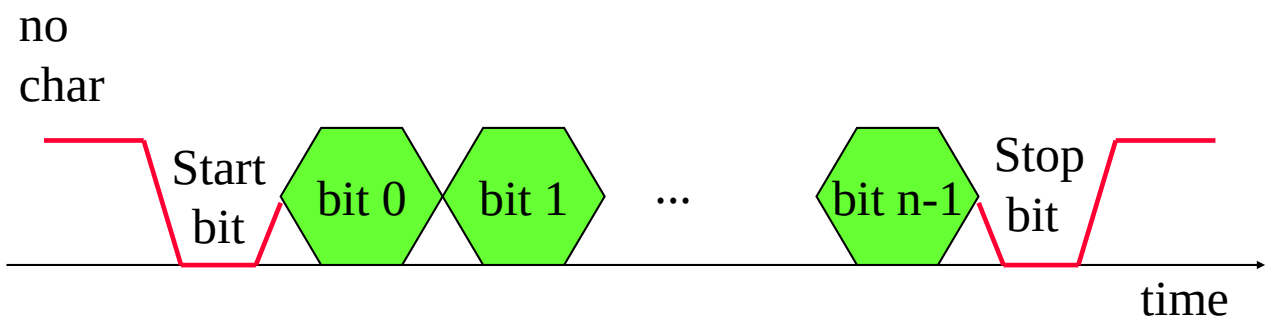
- Data registers: hold values that are treated as data by the device, such as the data read or written by a disk
- Status registers: provide information about the device's operation, such as whether the current transaction has completed

Application: 8251 UART

- Universal Asynchronous Receiver Transmitter (UART): provides serial communication.
- 8251 functions are integrated into standard PC interface chip.
- Allows many communication parameters to be programmed.

Serial communication

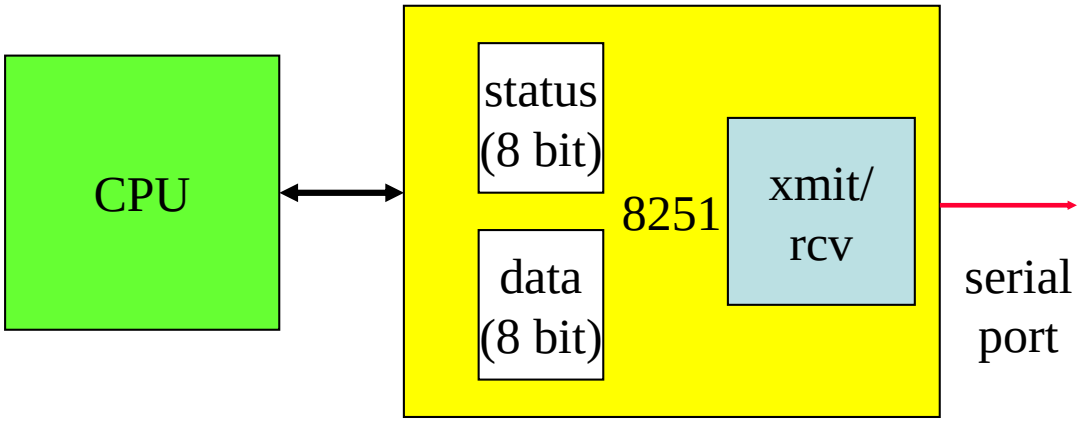
- Characters are transmitted separately:



Serial communication parameters

- Baud (bit) rate (bits/sec)
- Number of bits per character (5~8)
- Parity/no parity (even or odd)
- Even/odd parity
- Length of stop bit (1, 1.5, 2 bits)

8251 CPU interface



- Transmitter Ready: it is ready to accept a data character
- Transmitter Empty: when the UART has no character to send
- Receiver Ready: when the UART has a character to be read by CPU

Programming I/O

- Two types of instructions can support I/O:
  - special-purpose I/O instructions; (I/O mapped I/O)
  - memory-mapped load/store instructions.
- Intel x86 provides in, out instructions.
- Most other CPUs use memory-mapped I/O.
- I/O instructions do not preclude memory-mapped I/O.

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

- Read/write code:

```
LDR r1, #DEV1 ; set up device address
LDR r0, [r1]      ; read DEV1

LDR r0, #8        ; set up value to write
STR r0, [r1]      ; write value to device
```


SHARC memory mapped I/O

- Device must be in external memory space (above 0x400000).
- Use DM to control access:

```
I0 = 0x400000;
M0 = 0;
R1 = DM(I0,M0);
```

Peek and poke

- Traditional HLL (high-level language) interfaces:
- The traditional names for functions that read and write arbitrary memory locations are peek and poke.
- Peek function in C:

```
int peek(char *location) {
        return *location; }        ; /* de-reference location pointer */
```

  - To read a device register

```
#define DEV1 0x1000
…
dev_status = peek(DEV1)          ; /* read device register  */
```

- Poke function in C:

```
void poke(char *location, char newval) {
        (*location) = newval; }    ; /* write to location    */
```

  - To write to status register

```
poke(DEV1,8)                     ; /* write 8 to device register  */
```


Busy-wait input and output

- Devices are typically slower than CPU
- Polling:
  - asking an I/O device whether it is whether it is finished by reading its status register
  - If CPU is writing several characters to an output device, then it must wait for one operation to complete before starting the next one.
    - If we try to start writing the second character before the device has finished with the first one, for example, the device will probably never print the first character.
  - Simplest way to program device.
    - Use instructions to test when device is ready.

Busy-wait I/O

- Busy-wait routine in C (Ex. 3-3)

```
current_char = mystring;            /* point to head of string  */
while (*current_char != '\0') {     /* until null character  */
poke(OUT_CHAR,*current_char); /* send character to device */
while (peek(OUT_STATUS) != 0); /* keep checking status */
current_char++; /* update character pointer */
}
```

  - Outer while loop sends the characters one at a time
  - Inner while loop checks the device status
  - By repeatedly checking the device status until the status changes to 0

Simultaneous busy/wait input and output

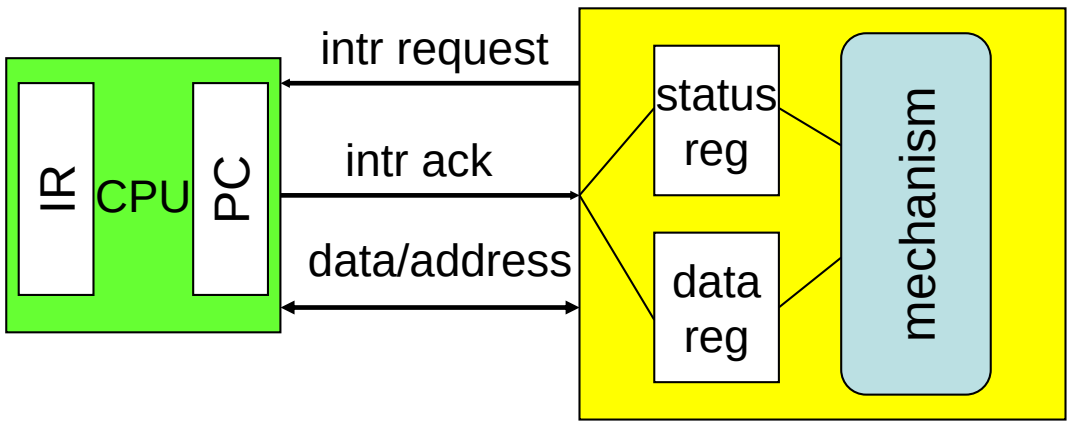
- *Ex.3-4 Copying characters from input to output using busy-wait I/O*

```
while (TRUE) {                          /* perform operation forever */
    /* read */
    while (peek(IN_STATUS) == 0); /* wait until ready  */
    achar = (char)peek(IN_DATA);   /* read character  */
    /* write */
    poke(OUT_DATA,achar);
    poke(OUT_STATUS,1);             /* turn on device  */
    while (peek(OUT_STATUS) != 0);          /* wait until done  */
}
```

Interrupt I/O

- Busy/wait is very inefficient. (Polling)
  - CPU can't do other work while testing device
  - Hard to do simultaneous I/O → parallel processing
- CPU can do useful work in parallel with I/O transaction, such as computation, and control of other I/O
- Interrupts allow a device to change the flow of control in the CPU.
  - Causes subroutine call to handle device.

Interrupt interface



- PC → interrupt handler routine (device driver)
- Interrupt request: when it wants service from CPU
- Interrupt acknowledgement: when it is ready to handle the I/O device's request

Interrupt behavior

- Based on subroutine call mechanism.
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
  - Return address is saved to resume executing foreground program.
- Interrupts allow the flow of control in the CPU to change easily between different contexts, such as a foreground computation and multiple I/O devices

Interrupt physical interface

- CPU and device are connected by CPU bus.
- CPU and device handshake:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.

Ex. 3-5: character I/O handlers

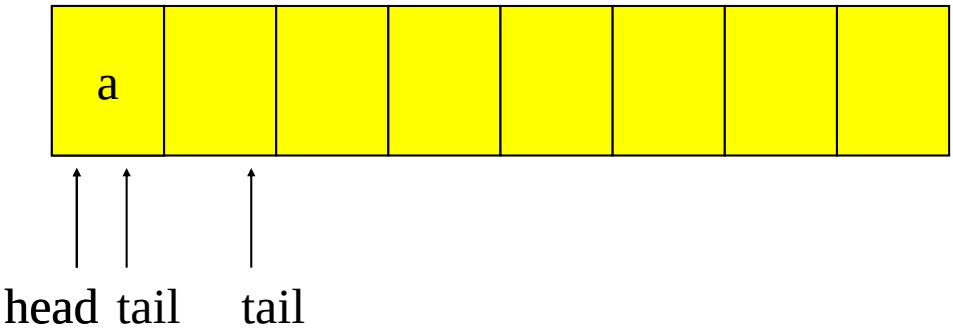
- Coping characters from input to output with basic interrupt

```
void input_handler() {              /*  get a char. and put it  in globle */
    achar = peek(IN_DATA);          /* get character   */
    gotchar = TRUE;                 /* signal to main program    */
    poke(IN_STATUS,0);          /* reset status to initiate next transfer    */
}
void output_handler() {             /* react to character being sent  */
}
```


Ex. 3-5: interrupt-driven main program

```
main() {
    while (TRUE) {                  /* read then write forever  */
        if (gotchar) {              /* write a character  */
            poke(OUT_DATA,achar);  /* put character in device */
            poke(OUT_STATUS,1);    /* set status to initiate write */
            gotchar = FALSE;            /* reset flag */
            }
        }
    }
```

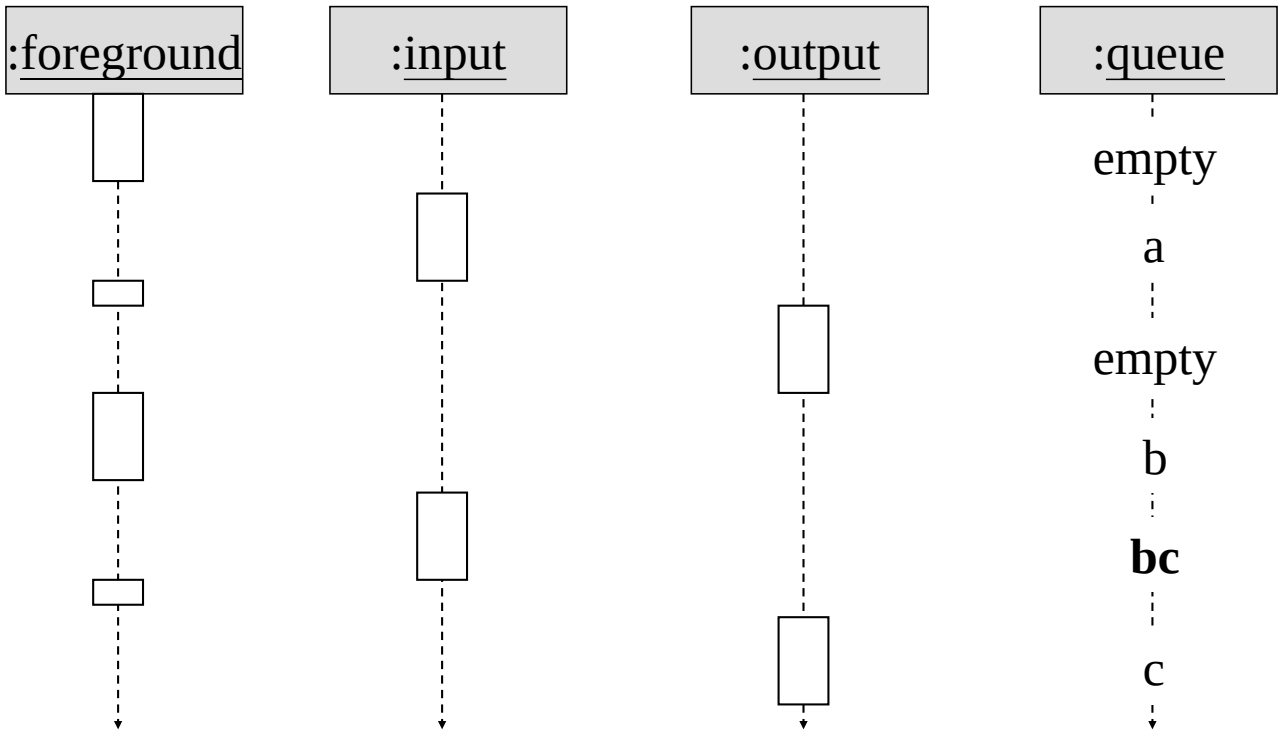
- Queue for characters:



Buffer-based input handler

```
void input_handler() {
    char achar;
    if (full_buffer()) error = 1;
    else { achar = peek(IN_DATA);
    add_char(achar); }
    poke(IN_STATUS,0);
    if (nchars == 1)
        { poke(OUT_DATA,remove_char();
    poke(OUT_STATUS,1); }
    }
```

I/O sequence diagram



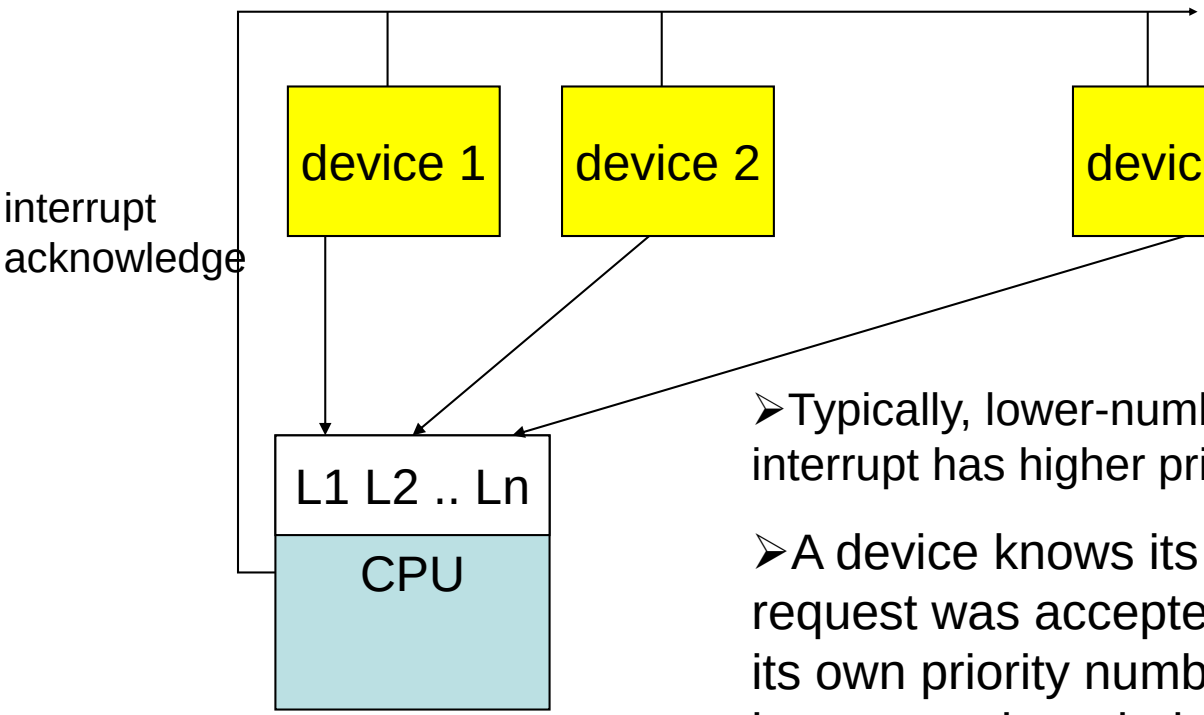
Debugging interrupt code

- What if you forget to change registers?
  - Foreground program can exhibit mysterious bugs.
  - Bugs will be hard to repeat---depend on interrupt timing.

Priorities and vectors

- Two mechanisms allow us to make interrupts more specific:
  - Priorities determine what interrupt gets CPU first.
  - Vectors determine what code is called for each type of interrupt.
- Typically, lower-numbered interrupt has higher priority
- A device knows its interrupt request was accepted by seeing its own priority number on the interrupt acknowledge lines.
- Mechanisms are orthogonal: most CPUs provide both.

Prioritized interrupts

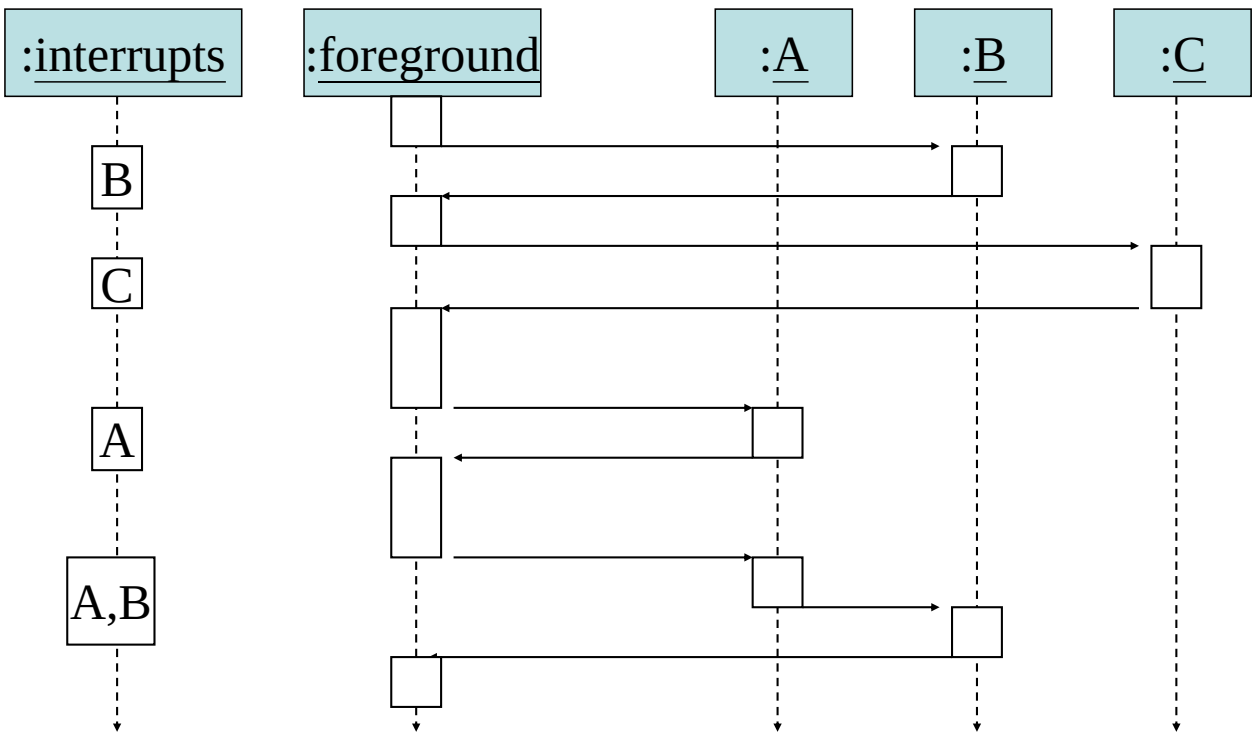


- Typically, lower-numbered interrupt has higher priority
- A device knows its interrupt request was accepted by seeing its own priority number on the interrupt acknowledge lines.

Interrupt prioritization

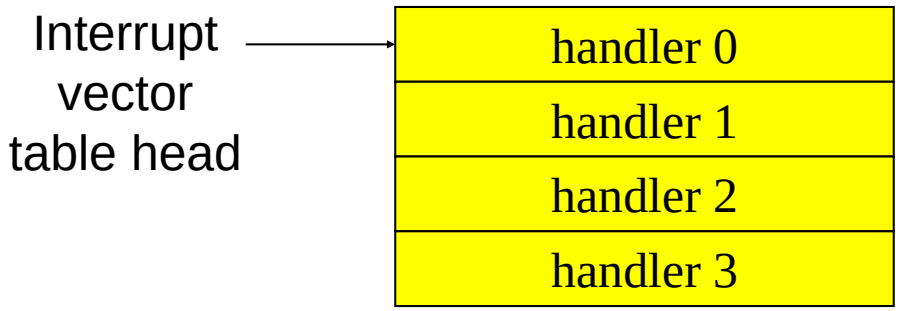
- Masking: interrupt with priority lower than current priority is not recognized until pending interrupt is complete.
- Non-maskable interrupt (NMI): highest-priority, never masked.
  - Often used for power-down.
- We can combine polling with prioritized interrupts to efficiently handle the devices. (see Fig. 3-4)

Example 3-8: Prioritized I/O

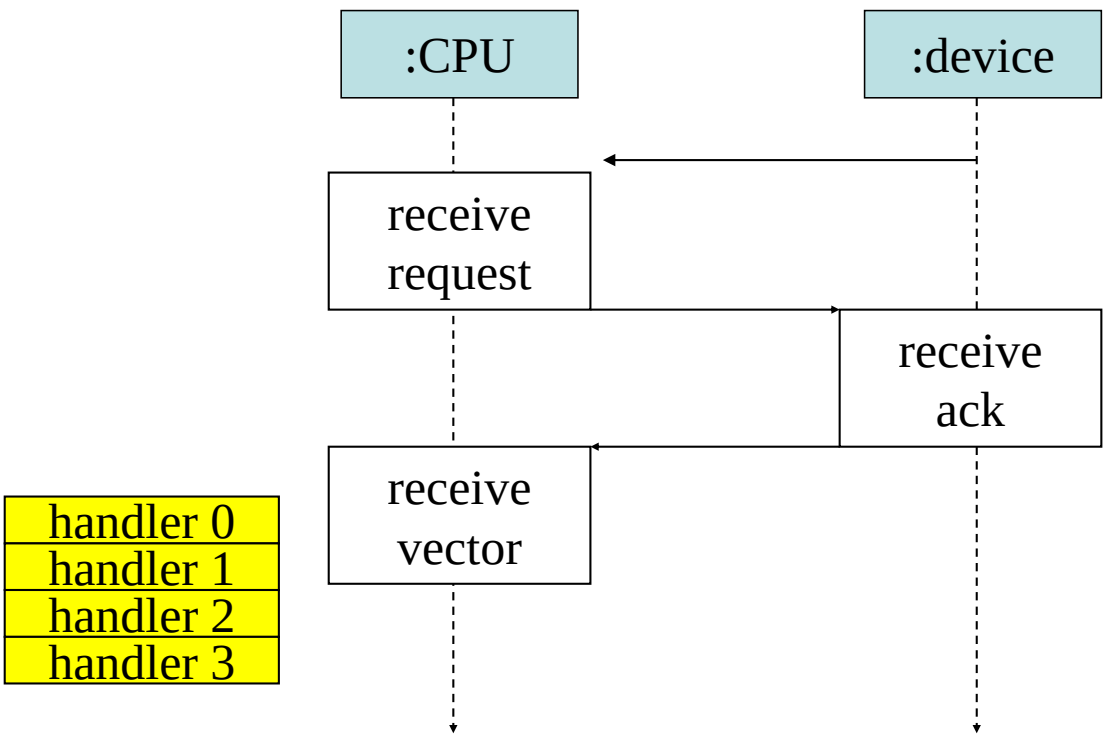


Interrupt vectors

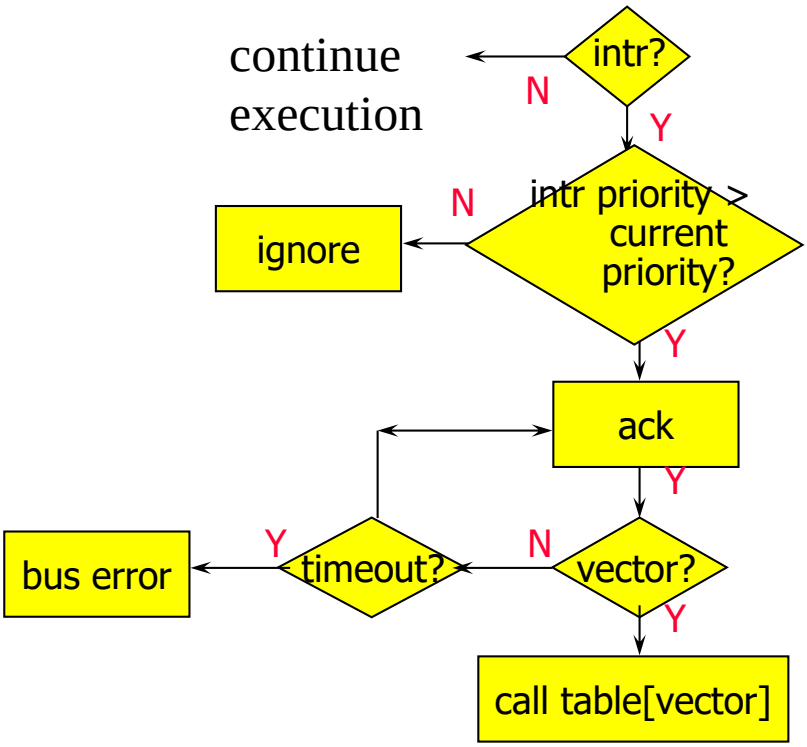
- Allow different devices to be handled by different code.
- Interrupt vector table:



Interrupt vector acquisition



Generic interrupt mechanism



Assume priority selection is handled before this point.

Interrupt sequence

1. CPU: check pending interrupts at the beginning of an instruction. It answer the highest priority interrupt by acknowledging request.
2. Device: receives acknowledgment and sends vector.
3. CPU calls handler by LUT. A subroutine-like mechanism is used to save the PC and CPU's state.
4. Software: the device driver may save additional CPU state
   - Processes request
   - Restore the saved state
   - Interrupt return
5. CPU restores PC and state to foreground program.

Sources of interrupt overhead

- Handler execution time (similar to a subroutine)
  - Branch penalty
  - Store CPU register
- Interrupt mechanism overhead
  - Acknowledge the interrupt
  - Obtain the vector from the device
- Register save/restore.
- Pipeline-related penalties.
- Cache-related penalties.

ARM interrupts

- ARM7 supports two types of interrupts:
  - Fast interrupt requests (FIQs).
  - Interrupt requests (IRQs).
- Interrupt table starts at location 0.

ARM interrupt procedure

- CPU actions:
  - Save PC. Copy CPSR to SPSR.
  - Force bits in CPSR to record interrupt.
  - Force PC to vector.
- Handler responsibilities:
  - Restore proper PC.
  - Restore CPSR from SPSR.
  - Clear interrupt disable flags.

ARM interrupt latency

- Worst-case latency to respond to interrupt is 27 cycles:
  - Two cycles to synchronize external request.
  - Up to 20 cycles to complete current instruction.
  - Three cycles for data abort.
  - Two cycles to enter interrupt handling state.

Supervisor mode

- May want to provide protective barriers between programs.
  - Avoid memory corruption.
- Need supervisor mode to manage the various programs.
- SHARC does not have a supervisor mode.

ARM supervisor mode

- Use SWI instruction to enter supervisor mode, similar to subroutine:

  SWI CODE_1
- Sets PC to 0x08.
- The bottom 5 bits of the CPSR are all set to 1
- Argument to SWI is passed to supervisor mode code.
- Saves CPSR in SPSR.

Exception

- Exception: internally detected error.
- Exceptions are synchronous with instructions but unpredictable.
- Build exception mechanism on top of interrupt mechanism.
- Exceptions are usually prioritized and vectorized.

Trap

- Trap (software interrupt): an exception generated by an instruction.
  - Call supervisor mode.
- ARM uses SWI instruction for traps.
- SHARC offers three levels of software interrupts.
  - Called by setting bits in IRPTL register.

Co-processor

- Co-processor: added function unit that is called by instruction.
  - Floating-point units are often structured as co-processors.
- ARM allows up to 16 designer-selected co-processors.
  - Floating-point co-processor uses units 1 and 2.