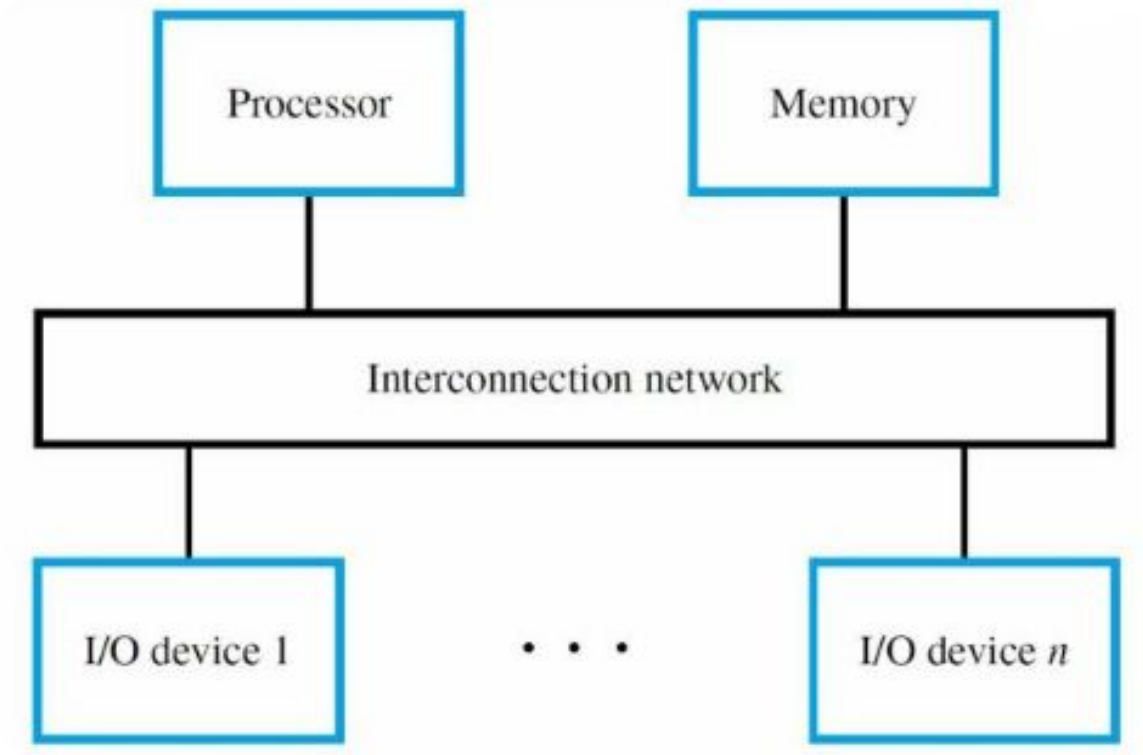
# Basic Input/Output

## Accessing I/O Devices

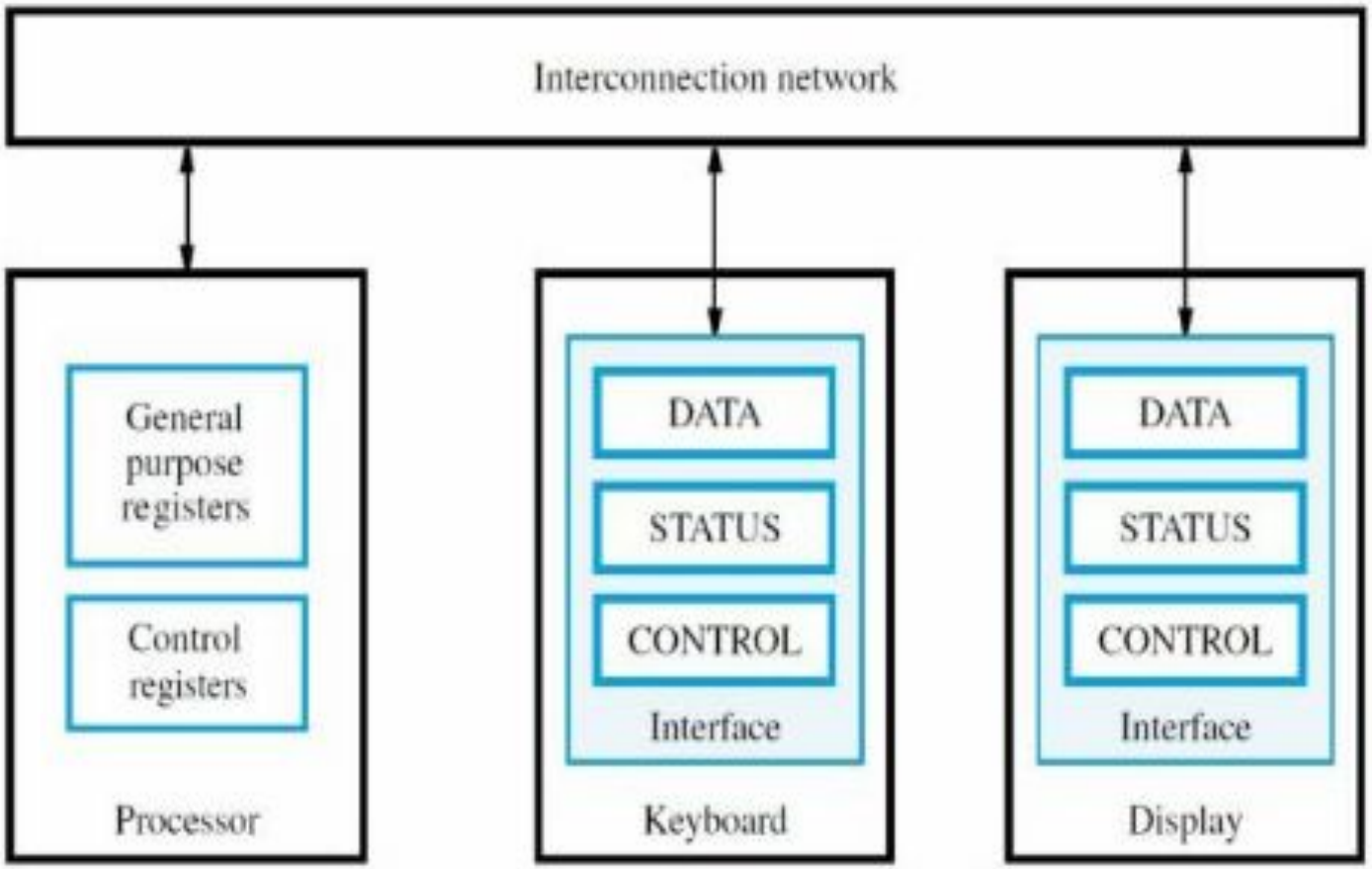
- Computer system components communicate through an interconnection network
- Memory-mapped I/O allows I/O registers to be accessed as memory locations.
- As a result, these registers can be accessed using only Load and Store instructions




**A computer system**.

## I/O Device Interface

- An I/O device interface is a circuit between a device and the interconnection network
- Provides the means for data transfer and exchange of status and control information
- Includes data, status, and control registers accessible with Load and Store instructions
- Memory-mapped I/O enables software to view these registers as locations in memory



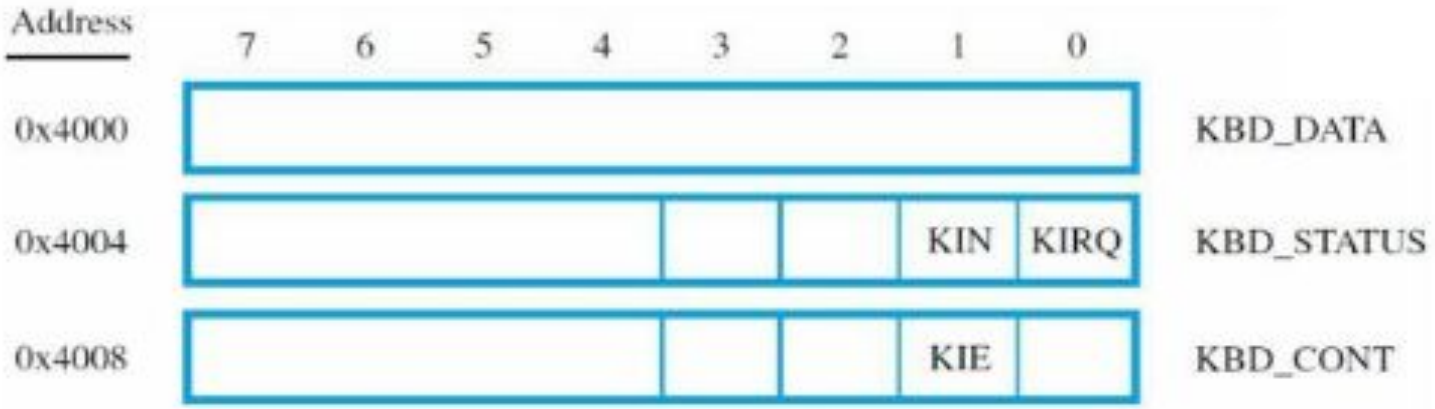

**The connection for processor, keyboard, and display.**

## Program-Controlled I/O

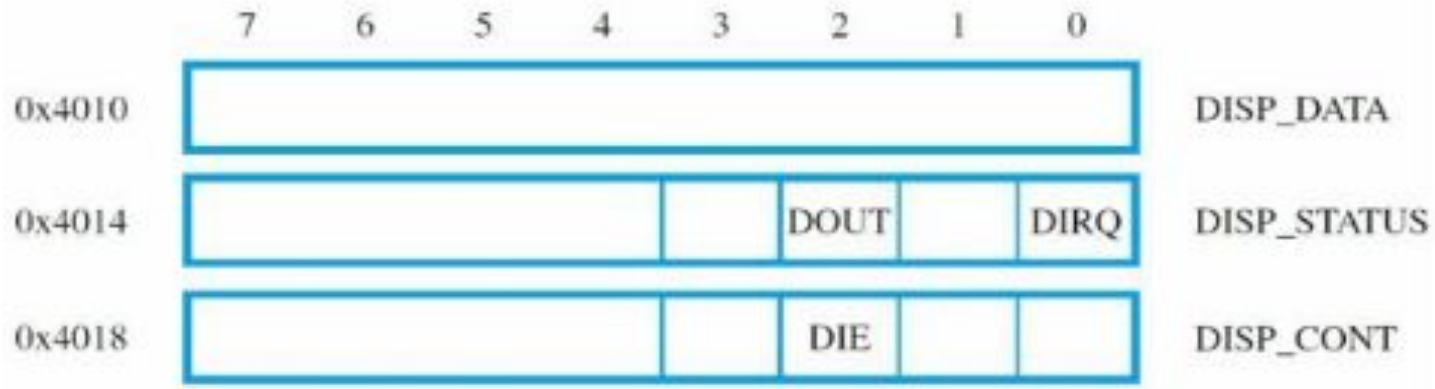
- Discuss I/O issues using keyboard & display
- Read keyboard characters, store in memory, and display on screen
- Implement this task with a program that performs all of the relevant functions
- This approach called program-controlled I/O

## Signaling Protocol for I/O Devices

- Assume that the I/O devices have a way to send a ‘READY’ signal to the processor
- For keyboard, it indicates that the character is ready to be read.
- For display, it indicates the display is ready to receive the character.
- The ‘READY’ signal in each case is a status flag in status register that is polled by processor




(a) Keyboard interface




(b) Display interface

## Wait Loop for Polling I/O Status

- Program-controlled I/O implemented with a wait loop for polling keyboard status register:

```
READWAIT:
    LoadByte R4, KBD_STATUS
    And R4, R4, #2
    Branch_if_[R4]=0 READWAIT
    LoadByte R5, KBD_DATA
```

- Keyboard circuit places character in KBD_DATA and sets KIN flag in KBD_STATUS

- Circuit clears KIN flag when KBD_STATUS read

## Wait Loop for Polling I/O Status

- Similar wait loop for display device:

```
WRITEWAIT:
    LoadByte R4, DISP_STATUS
    And R4, R4, #4
    Branch_if_[R4]=0 WRITEWAIT
   StoreByte R5, DISP_DATA
```

- Display circuit sets DOUT flag in DISP_STATUS after previous character has been displayed

- Circuit automatically clears DOUT flag when DISP_STATUS register is read

## RISC- and CISC-style I/O Programs

- Consider complete programs that use polling to read, store, and display a line of characters
- Each keyboard character echoed to display
- Program finishes when carriage return (CR) character is entered on keyboard
- LOC is address of first character in stored line
- CISC has TestBit, Compare Byte  well as auto-increment addressing mode

```
        Move                R2, #LOC        Initialize pointer register R2 to point to the
                                              address of the first location in main memory
                                              where the characters are to be stored.
        MoveByte            R3, #CR         Load ASCII code for Carriage Return into R3.
READ:   LoadByte            R4, KBD_STATUS  Wait for a character to be entered.
        And                 R4, R4, #2      Check the KIN flag.
        Branch_if_[R4]=0    READ
        LoadByte            R5, KBD_DATA    Read the character from KBD_DATA
                                              (this clears KIN to 0).
        StoreByte           R5, (R2)        Write the character into the main memory and
        Add                 R2, R2, #1        increment the pointer to main memory.
ECHO:   LoadByte            R4, DISP_STATUS Wait for the display to become ready.
        And                 R4, R4, #4      Check the DOUT flag.
        Branch_if_[R4]=0    ECHO
        StoreByte           R5, DISP_DATA   Move the character just read to the display
                                              buffer register (this clears DOUT to 0).
        Branch_if_[R5]≠[R3] READ            Check if the character just read is the
                                              Carriage Return. If it is not, then
                                              branch back and read another character.
```

A RISC-style program that reads a line of characters and displays it.

| | | | |
|---|---|---|---|
| | Move | R2, #LOC | Initialize pointer register R2 to point to the address of the first location in main memory where the characters are to be stored. |
| READ: | TestBit | KBD_STATUS, #1 | Wait for a character to be entered |
| | Branch=0 | READ | in the keyboard buffer KBD_DATA. |
| | MoveByte | (R2), KBD_DATA | Transfer the character from KBD_DATA into the main memory (this clears KIN to 0). |
| ECHO: | TestBit | DISP_STATUS, #2 | Wait for the display to become ready. |
| | Branch=0 | ECHO | |
| | MoveByte | DISP_DATA, (R2) | Move the character just read to the display buffer register (this clears DOUT to 0). |
| | CompareByte | (R2)+, #CR | Check if the character just read is CR (carriage return). If it is not CR, then |
| | Branch≠0 | READ | branch back and read another character. Also, increment the pointer to store the next character. |

A CISC-style program that reads a line of characters and displays it.

## Interrupts

- Drawback of a programmed I/O: BUSY-WAIT LOOP
- Due to the time needed to poll if I/O device is ready, the processor cannot often perform useful computation
- Instead of using a BUSY-WAIT LOOP, let I/O device alert the processor when it is ready
- Hardware sends an interrupt-request signal to the processor at the appropriate time, much like a phone call.
- Meanwhile, processor performs useful tasks

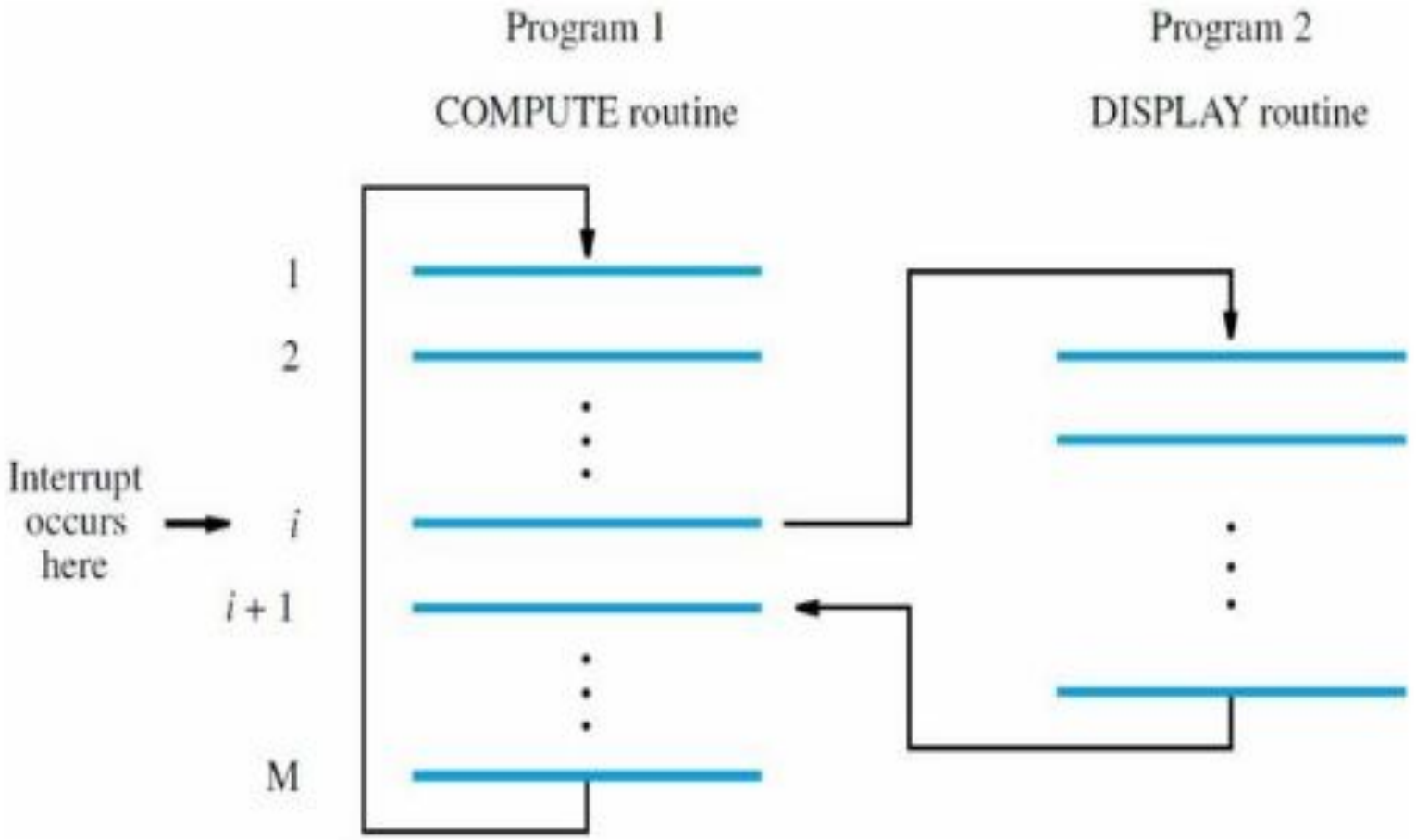
## Example of Using Interrupts

- Consider a task with extensive computation and periodic display of current results  Timer circuit can be used for desired interval, with interrupt-request signal to processor

- Two software routines: COMPUTE & DISPLAY

- Processor suspends COMPUTE execution to execute DISPLAY on interrupt, then returns

- DISPLAY is short; time is mostly in COMPUTE


Program 1
COMPUTE routine
Program 2
DISPLAY routine
1
2
Interrupt
occurs
here
i
i + 1
M

## Interrupt-Service Routine

- DISPLAY is an interrupt-service routine
- Differs from subroutine because it is executed at any time due to interrupt, not due to Call
- For example, assume interrupt signal asserted when processor is executing instruction I
- Instruction completes, then PC saved to temporary location before executing DISPLAY
- Return-from-interrupt instruction in DISPLAY restores PC with address of instruction i + 1

## Issues for Handling of Interrupts

- Save return address on stack or in a register
- Interrupt-acknowledge signal from processor tells device that interrupt has been recognized
- In response, device removes interrupt request
- Saving/restoring of general-purpose registers can be automatic or program-controlled

## Enabling and Disabling Interrupts

- Must processor always respond immediately to interrupt requests from I/O devices?

- Some tasks cannot tolerate interrupt latency and must be completed without interruption

- Need ways to enable and disable interrupts --Provides flexibility to programmers

- Use control bits in processor and I/O registers

## Event Sequence for an Interrupt

- Processor status (PS) register has Interrupt Enable (IE) bit
- Program sets IE to 1 to enable interrupts
- When an interrupt is recognized, processor saves program counter and status register
- IE bit cleared to 0 so that same or other signal does not cause further interruptions
- After acknowledging and servicing interrupt, restore saved state, which sets IE to 1 again

## Vectored Interrupts

- Vectored interrupts reduce service latency; no instructions executed to poll many devices
- Let requesting device identify itself directly with a special signal or a unique binary code (like different ringing tones for different callers)
- Processor uses info to find address of correct routine in an interrupt-vector table  Table lookup is performed by hardware.
- Vector table is located at fixed address, but routines can be located anywhere in memory

## Interrupt Nesting

- Service routines usually execute to completion
- To reduce latency, allow interrupt nesting by having service routines set IE bit to 1 Acknowledge the current interrupt request before setting IE bit to prevent infinite loop
- For more control, use different priority levels
- Current level held in processor status register
- Accept requests only from higher-level devices

## Simultaneous Requests

- Two or more devices request at the same time
- Arbitration or priority resolution is required
- With software polling of I/O status registers, service order determined by polling order With vectored interrupts, hardware must select only one device to identify itself
- Use arbitration circuits that enforce desired priority or fairness across different devices

## Controlling I/O Device Behavior

- Processor IE bit setting affects all devices
- Desirable to have finer control with IE bit for each I/O device in its control register Such a control register also enables selecting the desired mode of operation for the device
- Access register with Load/Store instructions
- For example interfaces, setting KIE or DIE to 1 enables interrupts from keyboard or display

## Processor Control Registers

- In addition to a processor status (PS) register, other control registers are often present
- IPS register is where PS is automatically saved when an interrupt request is recognized
- IENABLE has one bit per device to control if requests from that source are recognized
- IPENDING has one bit per device to indicate if interrupt request has not yet been serviced


Control registers in the processor.

## Accessing Control Registers

- Use special Move instructions that transfer values to and from general-purpose registers

- Transfer pending interrupt requests to R4:

```
MoveControl R4, IPENDING
```

- Transfer current processor IE setting to R2:

```
MoveControl R2, PS
```

- Transfer desired bit pattern in R3 to IENABLE:

```
MoveControl IENABLE, R3
```

## Examples of Interrupt Programs

- Use keyboard interrupts to read characters, but polling within service routine for display
- Illustrate initialization for interrupt programs, including data variables and control registers
- Show saving of registers in service routine
- Consider RISC-style and CISC-style programs
- We assume that predetermined location ILOC is address of 1st instruction in service routine

**Interrupt-service routine**

```
ILOC:   Subtract        SP, SP, #8          Save registers.
        Store           R2, 4(SP)
        Store           R3, (SP)
        Load            R2, PNTR            Load address pointer.
        LoadByte        R3, KBD_DATA        Read character from keyboard.
        StoreByte       R3, (R2)            Write the character into memory
        Add             R2, R2, #1            and increment the pointer.
        Store           R2, PNTR            Update the pointer in memory.
ECHO:   LoadByte        R2, DISP_STATUS     Wait for display to become ready.
        And             R2, R2, #4
        Branch_if_[R2]=0 ECHO
        StoreByte       R3, DISP_DATA       Display the character just read.
        Move            R2, #CR             ASCII code for Carriage Return.
        Branch_if_[R3]≠[R2] RTRN            Return if not CR.
        Move            R2, #1
        Store           R2, EOL             Indicate end of line.
        Clear           R2                  Disable interrupts in
        StoreByte       R2, KBD_CONT          the keyboard interface.
RTRN:   Load            R3, (SP)            Restore registers.
        Load            R2, 4(SP)
        Add             SP, SP, #8
        Return-from-interrupt
```

**Main program**

```
START:  Move            R2, #LINE
        Store           R2, PNTR            Initialize buffer pointer.
        Clear           R2
        Store           R2, EOL             Clear end-of-line indicator.
        Move            R2, #2              Enable interrupts in
        StoreByte       R2, KBD_CONT          the keyboard interface.
        MoveControl     R2, IENABLE
        Or              R2, R2, #2          Enable keyboard interrupts in
        MoveControl     IENABLE, R2           the processor control register.
        MoveControl     R2, PS
        Or              R2, R2, #1
        MoveControl     PS, R2              Set interrupt-enable bit in PS.
        next instruction
```

A RISC-style program that reads a line of characters using interrupts, and displays the line using polling.

**Interrupt handler**

```
ILOC:       Subtract          SP, SP, #12          Save registers.
            Store             LINK_reg, 8(SP)
            Store             R2, 4(SP)
            Store             R3, (SP)
            MoveControl       R2, IPENDING         Check contents of IPENDING.
            And               R3, R2, #4           Check if display raised the request.
            Branch_if_[R3]=0  TESTKBD              If not, check if keyboard.
            Call              DISR                 Call the display ISR.
TESTKBD:    And               R3, R2, #2           Check if keyboard raised the request.
            Branch_if_[R3]=0  NEXT                 If not, then check next device.
            Call              KISR                 Call the keyboard ISR.
NEXT:       ...                                    Check for other interrupts.

            Load              R3, (SP)             Restore registers.
            Load              R2, 4(SP)
            Load              LINK_reg, 8(SP)
            Add               SP, SP, #12
            Return-from-interrupt
```

**Main program**

```
START:      ...                                    Set up parameters for ISRs.
            Move              R2, #2               Enable interrupts in
            StoreByte         R2, KBD_CONT           the keyboard interface.
            Move              R2, #4               Enable interrupts in
            StoreByte         R2, DISP_CONT          the display interface.
            MoveControl       R2, IENABLE
            Or                R2, R2, #6           Enable interrupts in
            MoveControl       IENABLE, R2            the processor control register.
            MoveControl       R2, PS
            Or                R2, R2, #1
            MoveControl       PS, R2               Set interrupt-enable bit in PS.
            next instruction
```

**Keyboard interrupt-service routine**

```
KISR:       ...
            .
            .
            Return
```

**Display interrupt-service routine**

```
DISR:       ...
            .
            .
            Return
```

A RISC-style program that initializes and handles interrupts.

**Interrupt-service routine**

```
ILOC:   Move          -(SP), R2          Save register.
        Move          R2, PNTR           Load address pointer.
        MoveByte      (R2), KBD_DATA     Write the character into memory
        Add           PNTR, #1             and increment the pointer.
ECHO:   TestBit       DISP_STATUS, #2    Wait for the display to become ready.
        Branch=0      ECHO
        MoveByte      DISP_DATA, (R2)    Display the character just read.
        CompareByte   (R2), #CR          Check if the character just read is CR.
        Branch≠0      RTRN               Return if not CR.
        Move          EOL, #1            Indicate end of line.
        ClearBit      KBD_CONT, #1       Disable interrupts in keyboard interface.
RTRN:   Move          R2, (SP)+          Restore register.
        Return-from-interrupt
```

**Main program**

```
START:  Move          PNTR, #LINE        Initialize buffer pointer.
        Clear         EOL                Clear end-of-line indicator.
        SetBit        KBD_CONT, #1       Enable interrupts in keyboard interface.
        Move          R2, #2             Enable keyboard interrupts in
        MoveControl   IENABLE, R2          the processor control register.
        MoveControl   R2, PS
        Or            R2, #1
        MoveControl   PS, R2             Set interrupt-enable bit in PS.
        next instruction
```

A CISC-style program that reads a line of characters using interrupts, and displays the line using polling.

## Multiple Interrupt Sources

- To use interrupts for both keyboard & display, call subroutines from ILOC service routine
- Service routine reads IPENDING register
- Checks which device bit(s) is (are) set to determine which subroutine(s) to call
- Service routine must save/restore Link register
- Also need separate pointer variable to indicate output character for next display interrupt

**Interrupt handler**

```
ILOC:       Move            -(SP), R2          Save registers.
            Move            -(SP), LINK_reg
            MoveControl     R2, IPENDING       Check contents of IPENDING.
            TestBit         R2, #2             Check if display raised the request.
            Branch=0        TESTKBD            If not, check if keyboard.
            Call            DISR               Call the display ISR.
TESTKBD:    TestBit         R2, #1             Check if keyboard raised the request.
            Branch=0        NEXT               If not, then check next device.
            Call            KISR               Call the keyboard ISR.
NEXT:       ---                                Check for other interrupts.

            Move            LINK_reg, (SP)+    Restore registers.
            Move            R2, (SP)+
            Return-from-interrupt
```

**Main program**

```
START:      ---                                Set up parameters for ISRs.
            SetBit          KBD_CONT, #1       Enable interrupts in keyboard interface.
            SetBit          DISP_CONT, #2      Enable interrupts in display interface.
            MoveControl     R2, IENABLE
            Or              R2, #6             Enable interrupts in
            MoveControl     IENABLE, R2          the processor control register.
            MoveControl     R2, PS
            Or              R2, #1
            MoveControl     PS, R2             Set interrupt-enable bit in PS.
            next instruction
```

**Keyboard interrupt-service routine**

```
KISR:       ---
            .
            .
            .
            Return
```

**Display interrupt-service routine**

```
DISR:       ---
            .
            .
            .
            Return
```

A CISC-style program that initializes and handles interrupts.

## Exceptions

- An exception is any interruption of execution
- This includes interrupts for I/O transfers
- But there are also other types of exceptions
- Recovery from errors: detect division by zero, or instruction with an invalid OP code
- Debugging: use of trace mode & breakpoints
- Operating system: software interrupt to enter

## Recovery from Errors

- After saving state, service routine is executed
- Routine can attempt to recover (if possible)or inform user, perhaps ending execution
- With I/O interrupt, instruction being executed at the time of request is allowed to complete
- If the instruction is the cause of the exception, service routine must be executed immediately
- Thus, return address may need adjustment