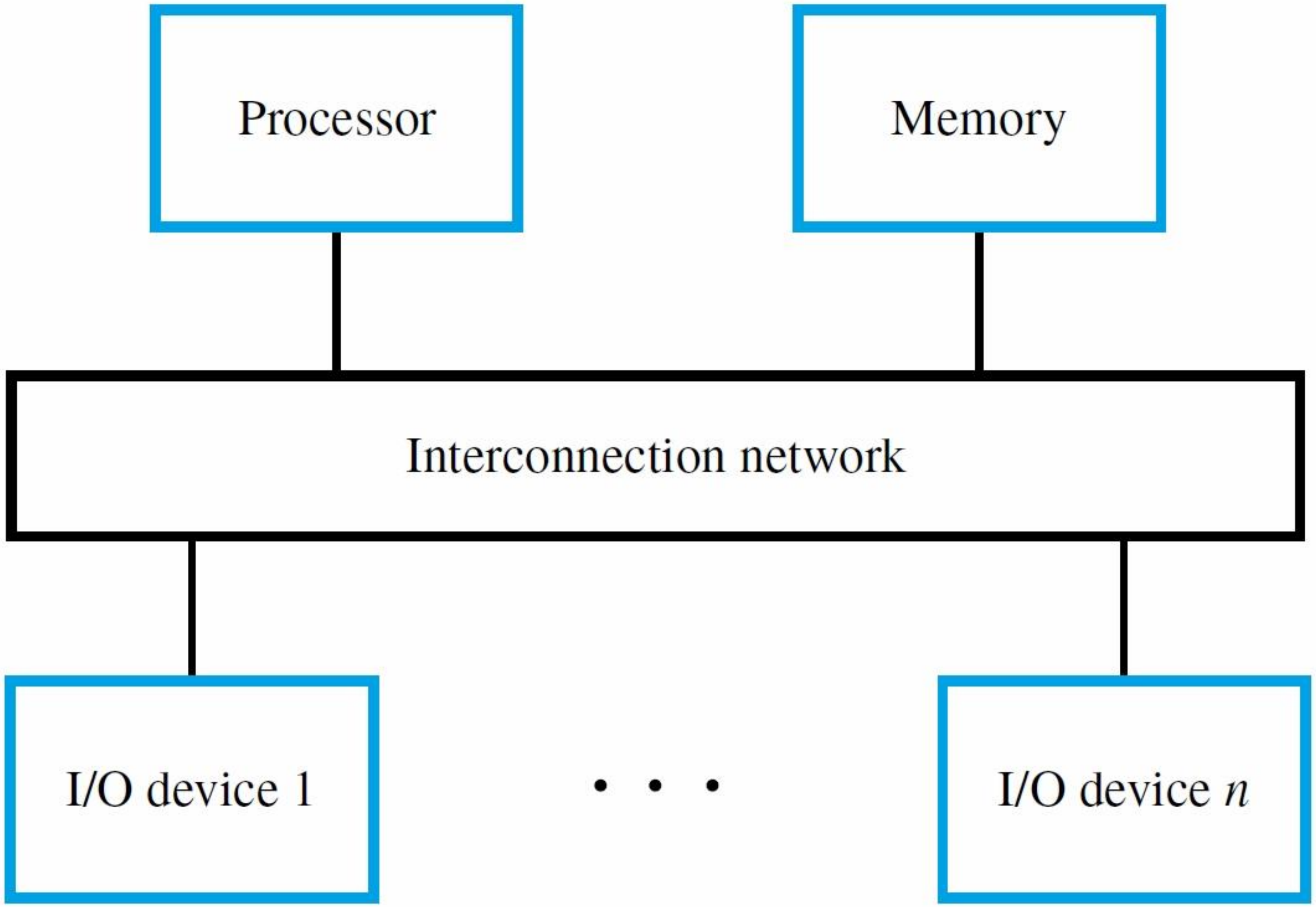
Chapter 3
Basic Input/Output

Chapter Outline

- Basic I/O capabilities of computers
- I/O device interfaces
- Memory-mapped I/O registers
- Program-controlled I/O transfers
- Interrupt-based I/O
- Exceptions

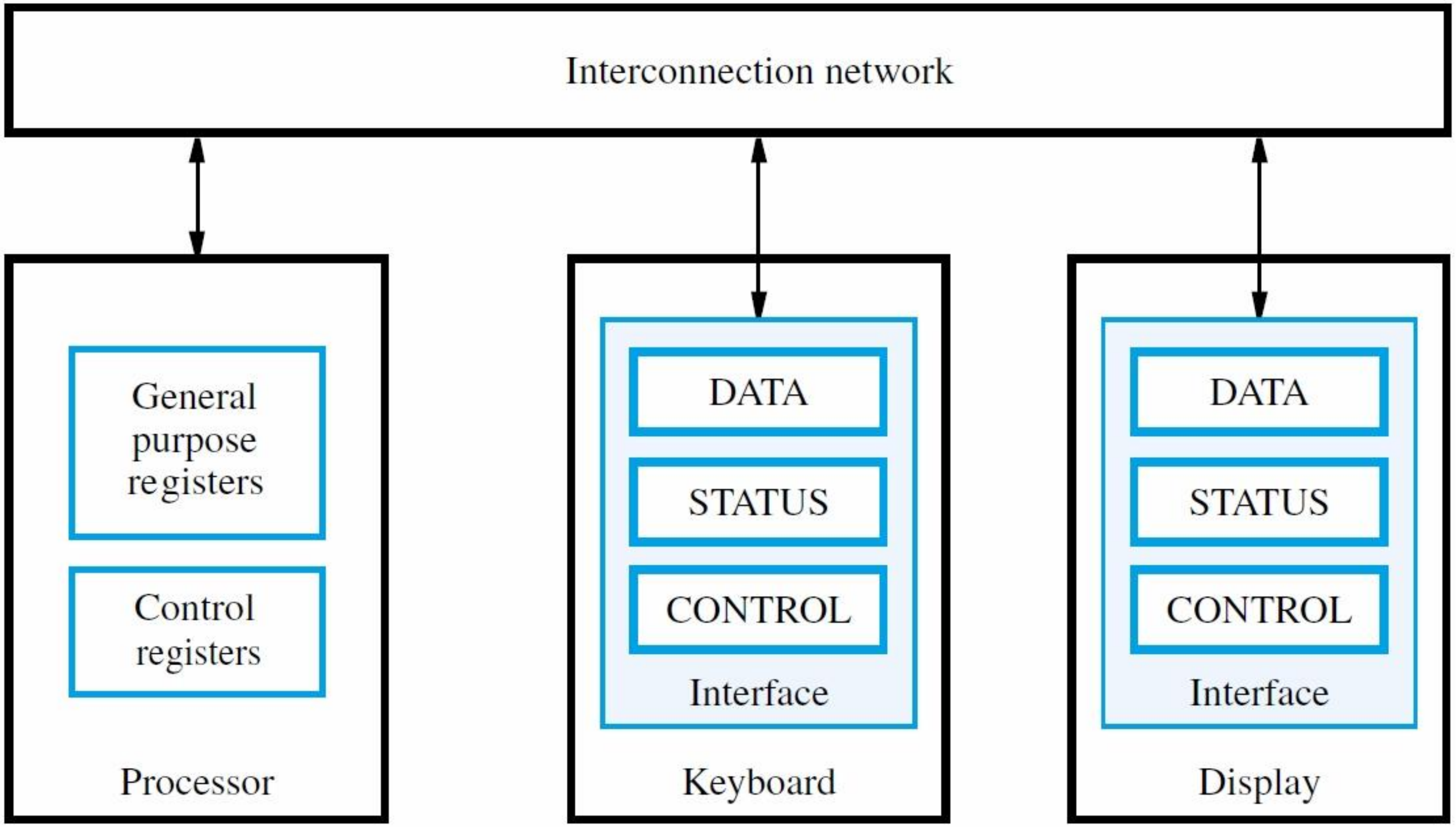
Accessing I/O Devices

- Computer system components communicate through an interconnection network
- Address space and memory access concepts from preceding chapter also apply here
- Locations associated with I/O devices are accessed with Load and Store instructions
- Locations implemented as I/O registers within same address space $\rightarrow$ memory-mapped I/O


Processor
Memory
Interconnection network
I/O device 1
• • •
I/O device *n*

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


Interconnection network
General purpose registers
Control registers
Processor
DATA
STATUS
CONTROL
Interface
Keyboard
DATA
STATUS
CONTROL
Interface
Display

Program-Controlled I/O

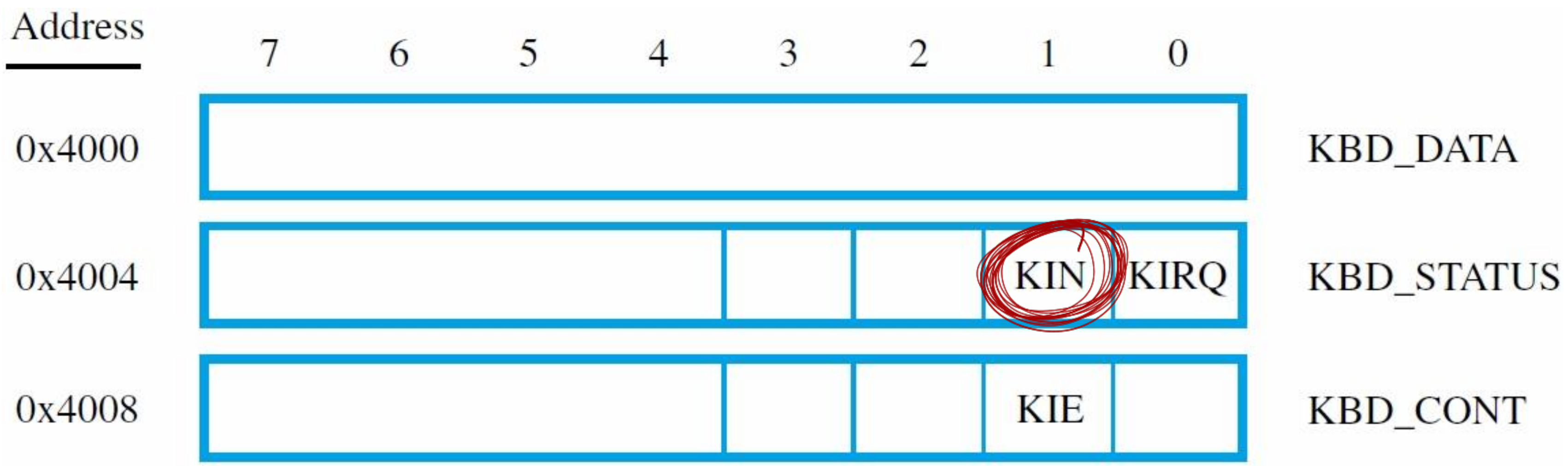
- Discuss I/O issues using keyboard & display
- Read keyboard characters, store in memory, and display on screen
- Implement this task with a program that performs all of the relevant functions
- This approach called program-controlled I/O
- How can we ensure correct timing of actions and synchronized transfers between devices?

Signaling Protocol for I/O Devices

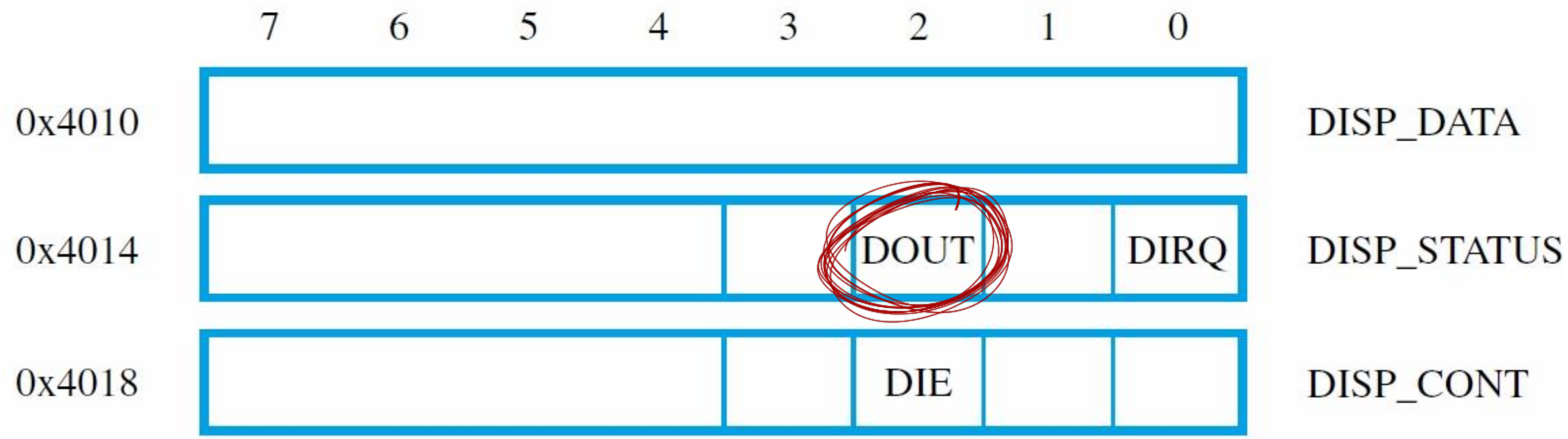
- Assume that the I/O devices have a way to send a '*ready*' signal to the processor
- For keyboard, indicates character can be read so processor uses Load to access data register
- For display, indicates character can be sent so processor uses Store to access data register
- The '*ready*' signal in each case is a status flag in status register that is polled by processor

Example I/O Registers

- For sample I/O programs that follow, assume specific addresses & bit positions for registers
- Registers are 8 bits in width and word-aligned
- For example, keyboard has KIN status flag in bit b_1 of KBD_STATUS reg. at address 0x4004
- Processor polls KBD_STATUS register, checking whether KIN flag is 0 or 1
- If KIN is 1, processor reads KBD_DATA register



(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_DATA is read

KIN =1 means key was pressed, data is ready in KBD_DATA.

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_DATA register is written

DOUT = 1 → display is ready to accept new data
= 0 → display is showing previous data

RISC-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, CompareByte instructions as well as auto-increment addressing mode

where we will start storing the typed characters

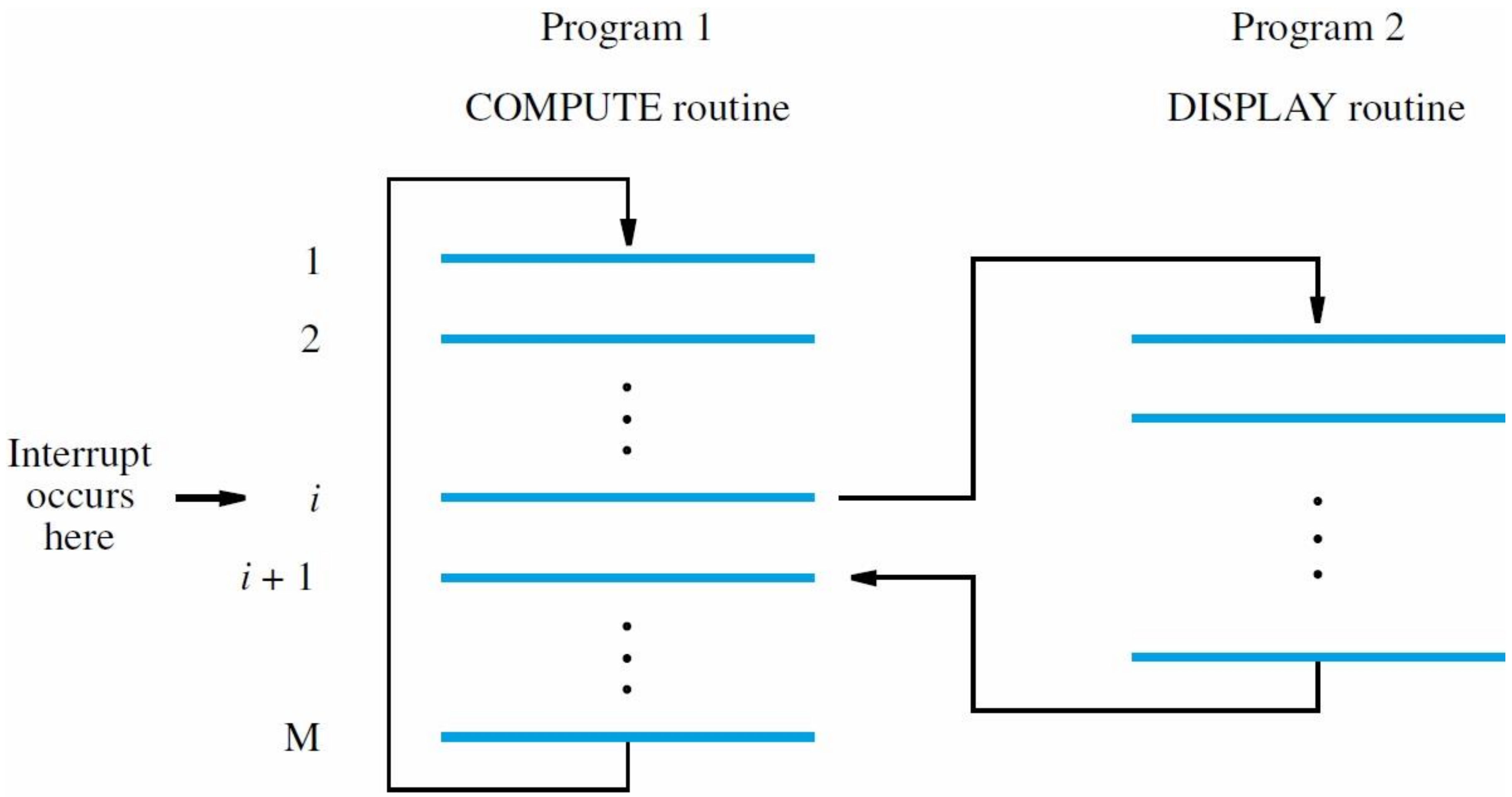
```
        Move                 R2, #LOC          Initialize pointer register R2 to point to the
                                                 address of the first location in main memory
                                                 where the characters are to be stored.
        MoveByte             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.
```

Interrupts

- Drawback of a wait loop: processor is busy
- With long delay before I/O device is ready, cannot perform other useful computation
- Instead of using a 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
- 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
- Acknowledgement can be done by accessing status or data register in device interface
- *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, both in processor and in device interfaces
- Provides flexibility to programmers
- Use control bits in processor and I/O registers

Event Sequence for an Interrupt

- Processor status (PS) register has 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

Handling Multiple Devices

- Which device is requesting service?
- How is appropriate service routine executed?
- Should interrupt nesting be permitted?
- How are two simultaneous requests handled?
- For 1st question, poll device status registers, checking if *IRQ* bit for each device is set
- For 2nd question, call device-specific routine for first set IRQ bit that is encountered

Vectored Interrupts

- Vectored interrupts reduce service latency; no instructions executed to poll many devices
- Let requesting device identify itself directly with a signal or a binary code
- 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

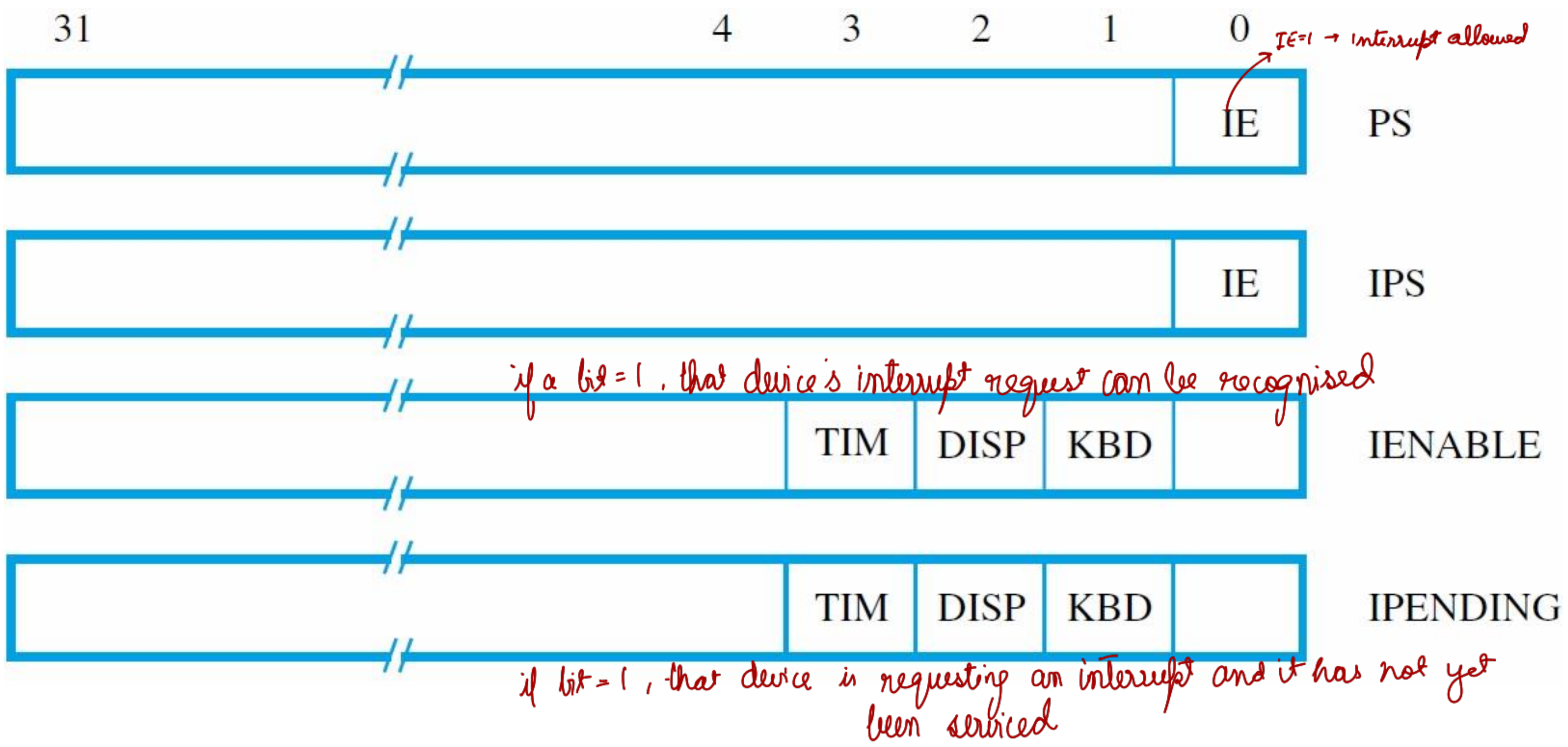
- Two or more devices request at the same time
- Arbitration or priority resolution is required
- With software polling of I/O status registers, service order determined by polling order
- With vectored interrupts, hardware must select only one device to identify itself
- Use arbitration circuits that 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 separate 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 can be recognized
- IPENDING has one bit per device to indicate if interrupt request has not yet been serviced


31
4
3
2
1
0
IE=1 → interrupt allowed
IE
PS
IE
IPS
if a bit = 1, that device's interrupt request can be recognised
TIM
DISP
KBD
IENABLE
TIM
DISP
KBD
IPENDING
if bit = 1, that device is requesting an interrupt and it has not yet been serviced

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 programs
- We assume that predetermined location *ILOC* is address of 1st instruction in service routine

Store X, Y → Store X in Y

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
```

पहले उसे R2 में डाला, Change value, then R2 को उसमें डाला

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*: Uses interrupts to communicate with and control the execution of user programs

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

Concluding Remarks

- Two basic I/O-handling approaches: *program-controlled* and *interrupt-based*
- 1st approach has direct control of I/O transfers
- Drawback: wait loop to poll flag in status reg.
- 2nd approach suspends program when needed to service I/O interrupt with separate routine
- Until then, processor performs useful tasks
- Exceptions cover all interrupts including I/O

Sections to Read (From Hamacher’s Book)

- Chapter on Basic Input/Output
  - All sections and sub-sections except
    - Sub-section 3.1.4 and any other CISC style programs