Ch.3 Input and Output

Topics

- Introduction: input and output (I/O)
  - Gaming industry devices
- 3.1.1: I/O device – system interface
- 3.1.2: Program-controlled I/O transfer
- 3.2: Interrupt-based I/O
- 3.2.6: Exceptions

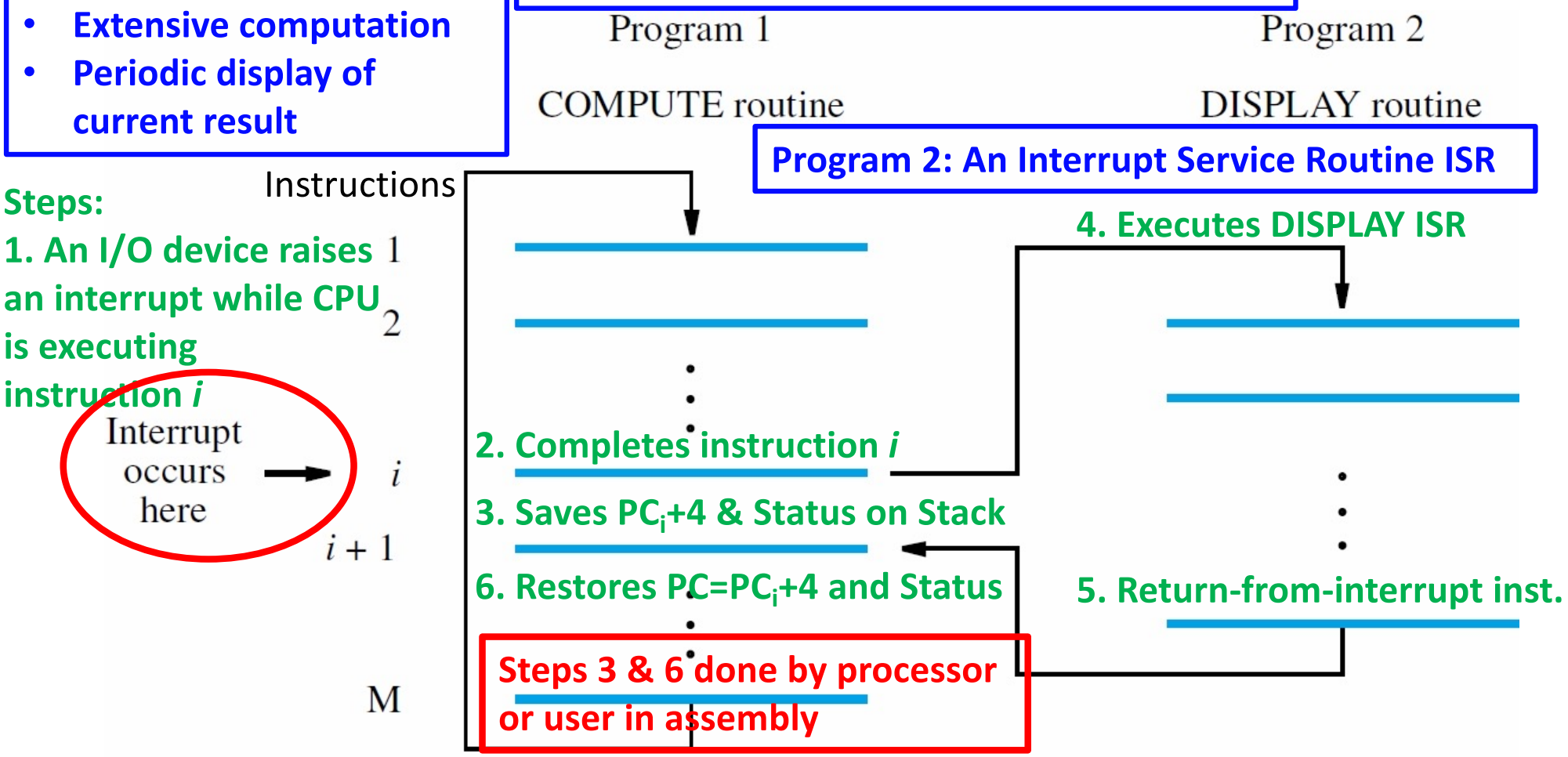
3.2 Interrupt I/O Fig. 3.6

Interrupt is external or internal?

Timer is an IO device ?

Program 1
- **Extensive computation**
- **Periodic display of current result**

Set a Timer; raise an interrupt when timeout



Interrupts Handling/Handshaking-1

Processor | **I/O Device**

1. Processor is executing a program

2. An I/O device *T* raises an interrupt

How? Send a hardware-based ***Interrupt-request*** signal

3. Processor suspends current program execution

When? After current instruction is processed

4. Processor identifies the specific interrupt *T*

How? By a) polling or b) from the interrupt vector

5. Processor acknowledges the interrupt has been recognized

How? Send a hardware-based ***Interrupt-acknowledge INTA*** signal

Interrupts Handling/Handshaking-2

Processor | **I/O Device**

6. I/O device *T* de-activates ***Interrupt-request***

7. Processor saves PC and status

Why? To allow for its next IRQ

Where? On Stack

8. Processor branches to and executes ***ISR T***

How? Get ISR address from Interrupt Table

9. Processor executes ISR *T* …till Return

10. Processor returns from ISR *T* and restores PC and status

Where? From Stack

11. Processor continues current program execution or ***process next IRQ from some device***

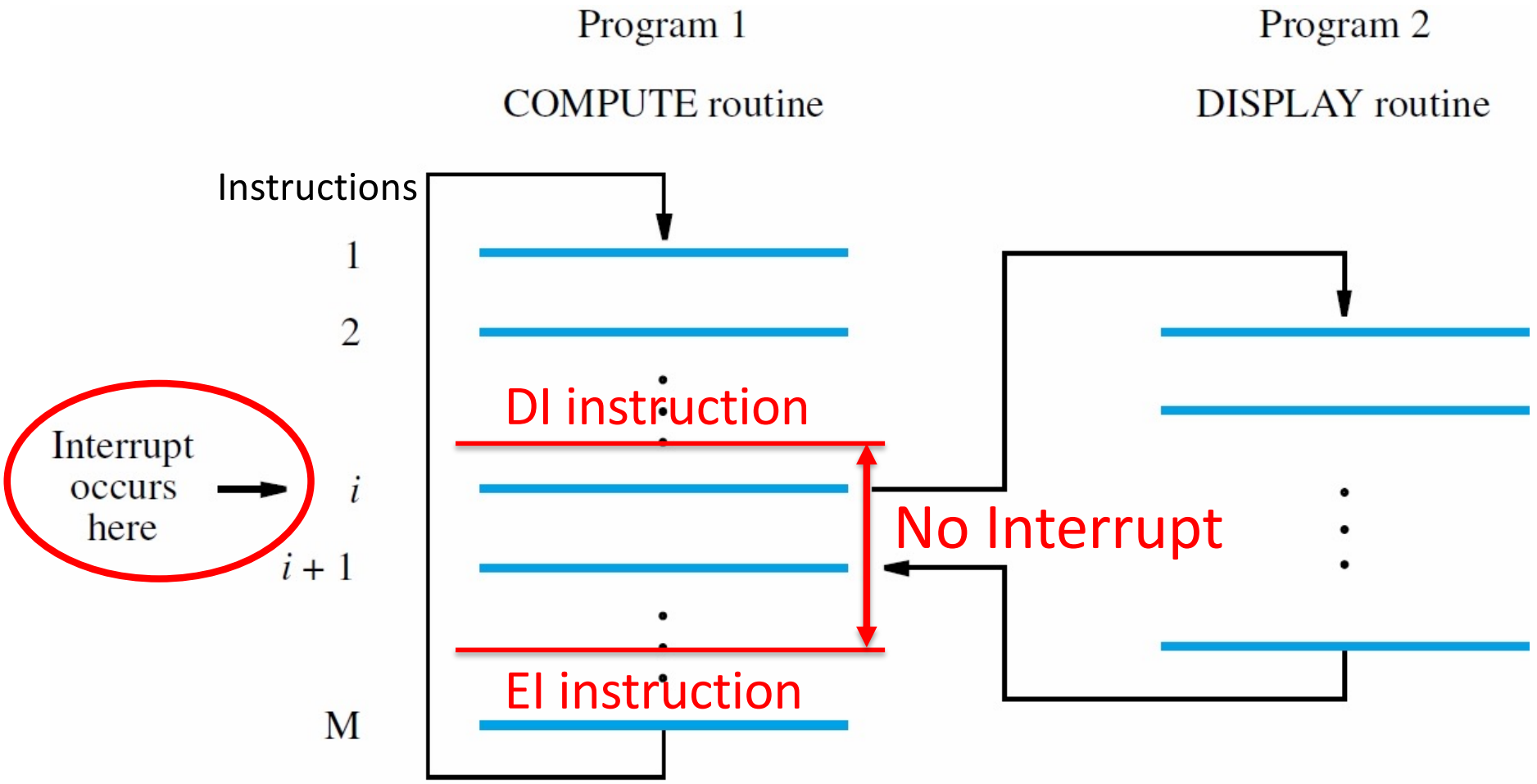
Disable and Enable Interrupts

- Some tasks cannot tolerate interrupt latency

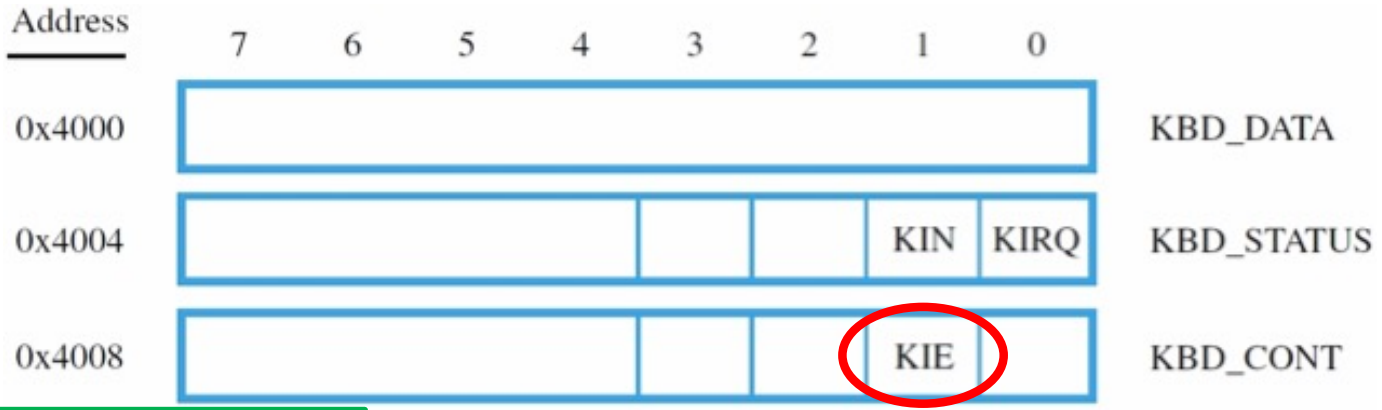
Interrupt Latency ?

  - In both main program and interrupt service routine
  - Processor must complete a task without interruption
- How?
  - Need to disable interrupts temporarily
  - Then enable interrupt again when the task is done
- Normal case:
    - **DI**: disable all interrupts
    - **EI**: enable all interrupts

DI & EI Interrupt



Device Level: EI in I/O Control Registers



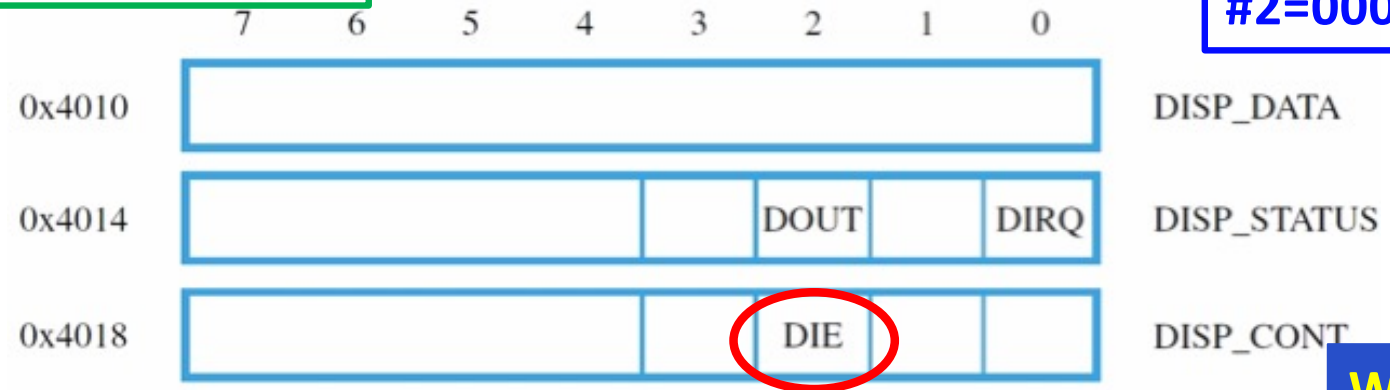
(a) Keyboard interface

Device Level:
EI or DI to that device only

Move #2, 0x4008

Why #2 ?

#2=0000 0010



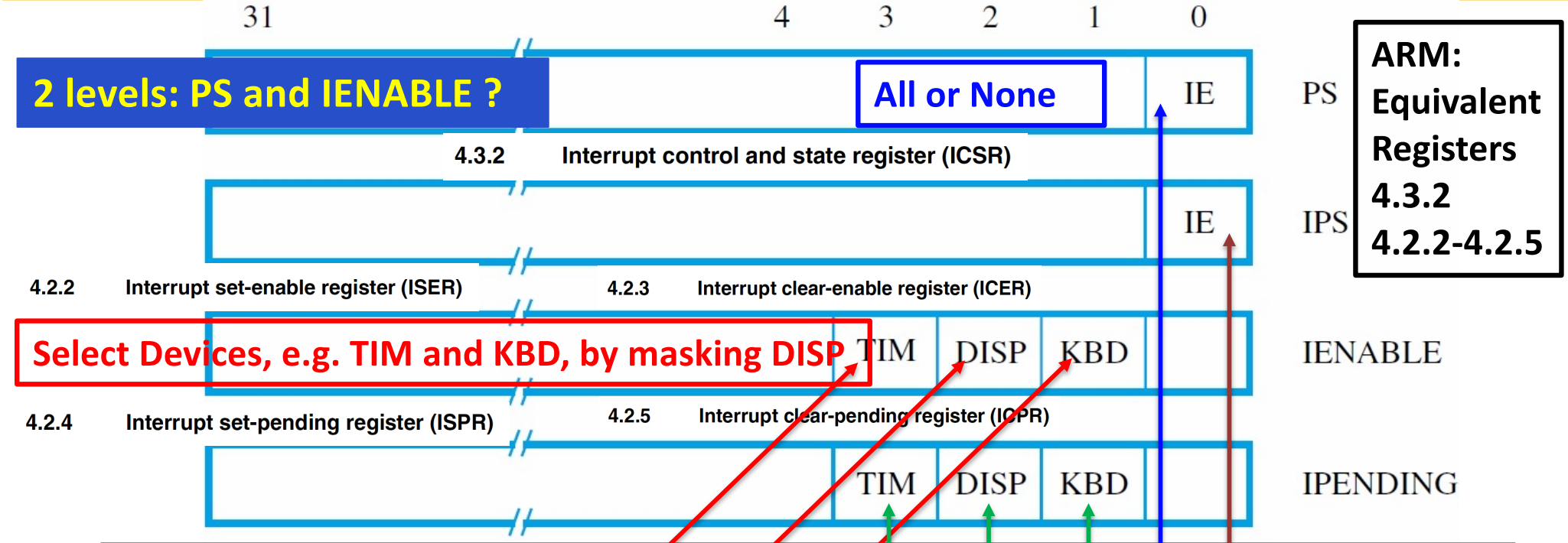
(b) Display interface

OR 0x4018, #4

Why #4 ?

#4=0000 0100

Processor Level: EI/DI in Control Registers



ARM: Equivalent Registers 4.3.2 4.2.2-4.2.5

PS (Processor Status) register: IE bit setting affects **all** devices
IPS (Interrupt Processor Status) register: content of PS when serving interrupt
IENABLE: one bit per device to enable/disable individual device (interrupt masking)
IPENDING: one bit per device to indicate active interrupt requests (not served yet)

Single Interrupt: Event Sequence

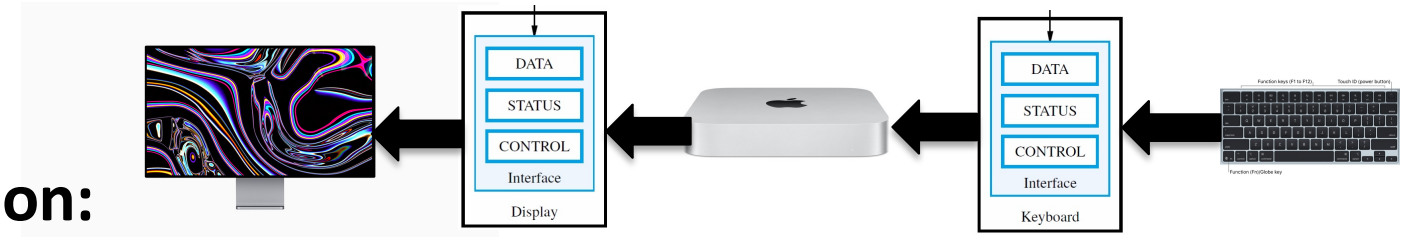
1. Program sets IE (e.g., in PS to 1) to enable interrupts (assuming interrupt enabled at the other two levels: Device and IENABLE)

PS (Processor Status) register: IE bit setting affects **all** devices

2. When an interrupt request IR_x is recognized, processor saves program counter and status register
3. IE bit is cleared to 0 so that there is no further interruptions
4. After servicing interrupt IR_x, restores saved state, and sets IE to 1 again

3.2.5 Processor-Keyboard-Display

Example application:



- Keyboard Input: interrupt to read entered characters
- Echo each keyboard character to Display
- Display Output: polling within Key-ISR
- Done when carriage return (CR) is entered and detected

Fig 3.8 Main Program

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
```

How many levels of interrupt control?

Why the OR ?

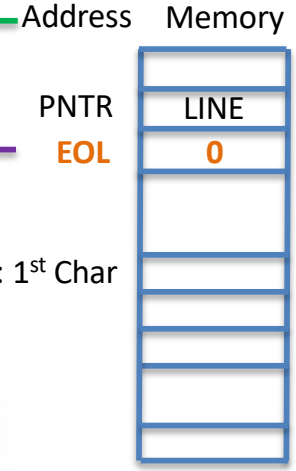


Fig. 3.8 Keyboard ISR

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		

Initialization
Save Reg's

Input Key-Data

Output when
Display ready

CR ?

If CR, done

Restore
Registers Used

Note: no parameter passed?

Fig 3.8 Main (Assignment)

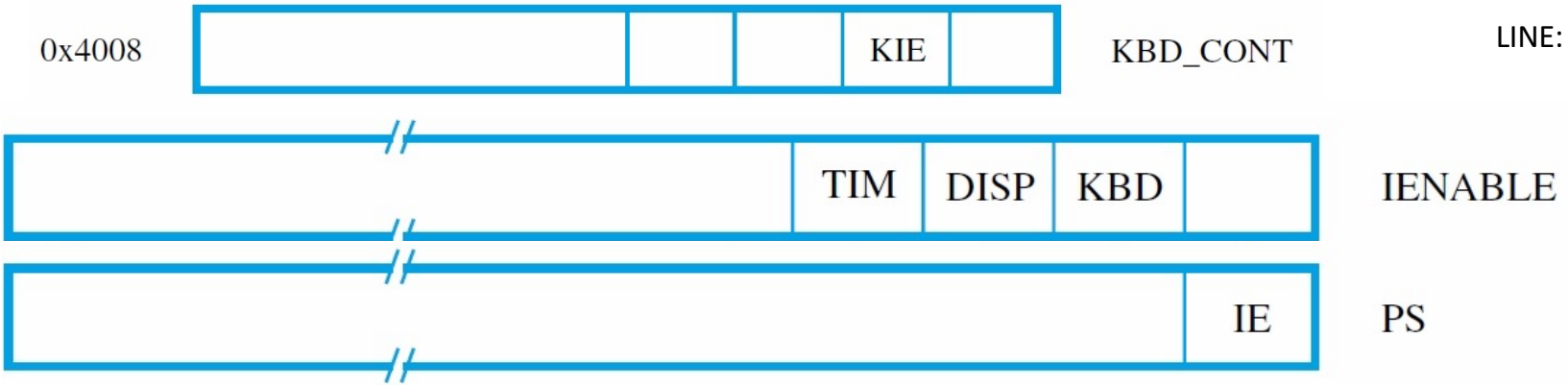
Main program

```
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
```

1. What is the purpose of EOL?

2. Why these EIs use different instructions?

3. Why these two Writes to Control Registers use different instructions?



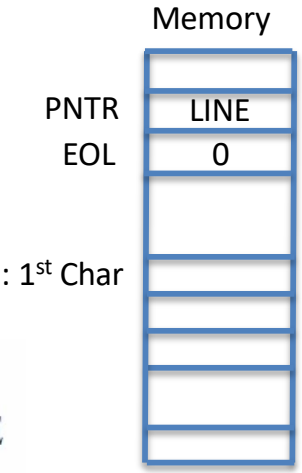


Fig. 3.8 ISR (Assignment)

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
```

4. Where is the Frame Pointer?

5. Why the PTR is incremented by only one address location?

6. Why do we wait-loop to echo, while there is no wait in getting the Keyboard input?

7. Why DI at keyboard interface, and not in PS or IENABLE registers?

8. Comment on the difference in saving/restoring registers in ISR vs Call Subroutine?