Corso di Architettura dei Sistemi a Microprocessore

Gestione dell'I/O

Luigi Coppolino

Contact info

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento di Ingegneria

Centro Direzionale di Napoli, Isola C4
V Piano lato SUD - Stanza n. 512

Tel: +39-081-5476702
Fax: +39-081-5476777

References

- Textbook
  - Chapter, 3

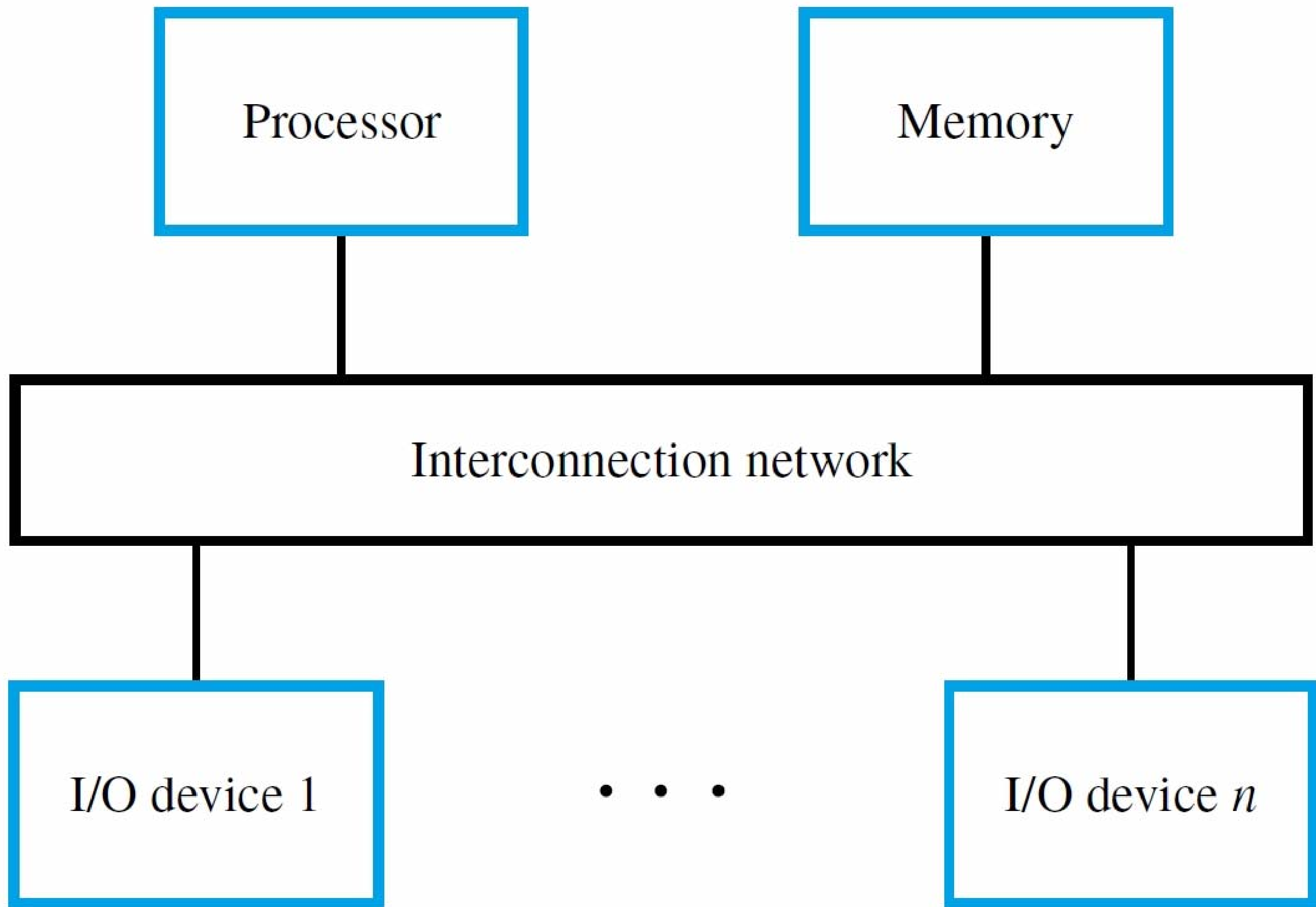
- Ambienti didattici di supporto:
  - Easy68K
  - ARMSim

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

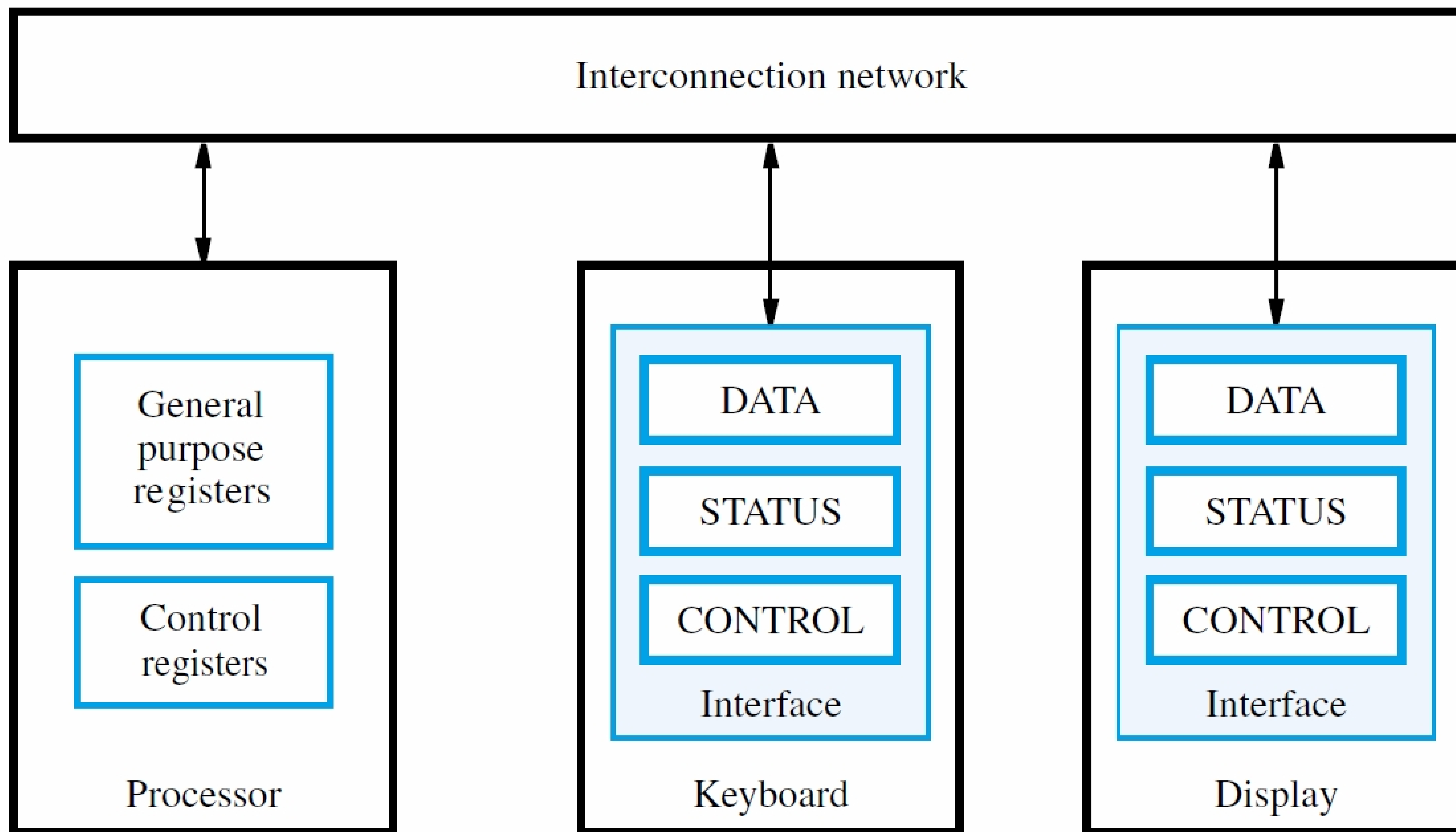
Accessing I/O Devices

- 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

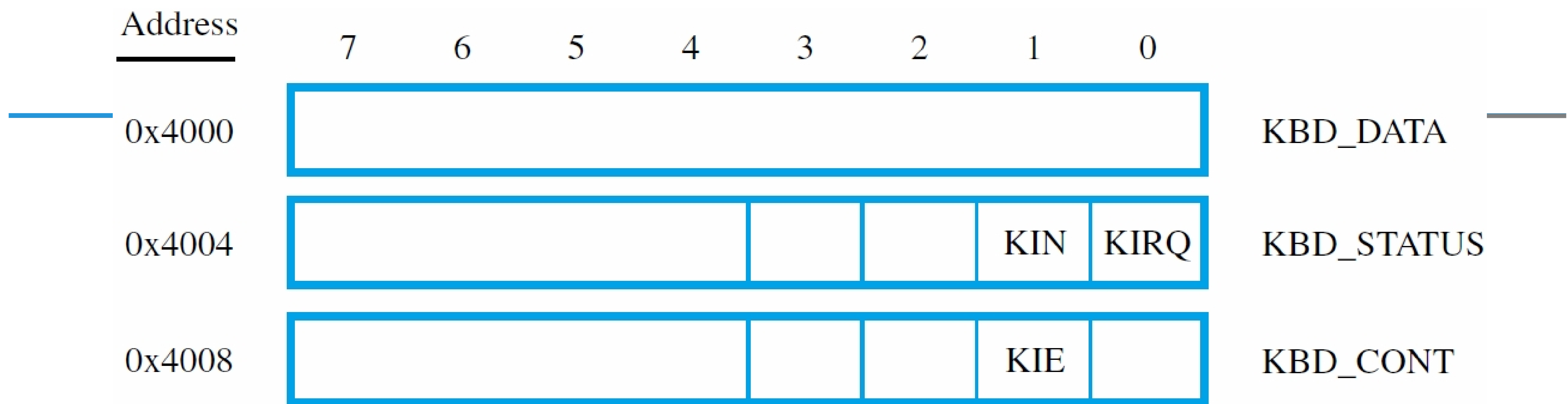
- Discuss I/O issues using keyboard & display
- Read keyboard characters, store in memory, and display on screen
- Implement this task with a program that performs all of the relevant functions
- This approach called program-controlled I/O
- How can we ensure correct timing of actions and synchronized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

Signaling Protocol for I/O Devices

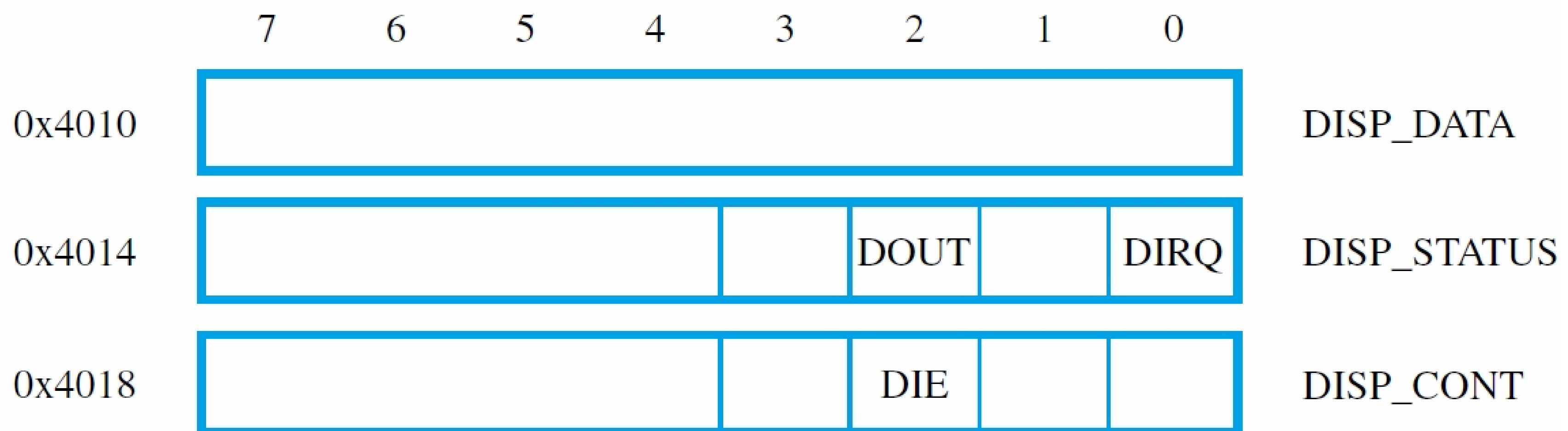
- Assume that the I/O devices have a way to send a '*ready*' signal to the processor
- For keyboard, indicates character can be read so processor uses Load to access data register
- For display, indicates character can be sent so processor uses Store to access data register
- The '*ready*' signal in 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_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, CompareByte instructions as 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.

```
        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.
```

	ARM	Commenti
KBD_DATA	EQU &4000	Indirizzi dei registri dati di tastiera e schermo
DISP_DATA	EQU &4010	
	LDR R0, =LOC	Locazione dove sarà immagazzinata la linea
	LDR R1, =KBD_DATA	Puntatore al registro dati della tastiera
	LDR R2, =DISP_DATA	Puntatore al registro dati dello schermo
CR	EQU &0D	Codice ASCII di Ritorno Carrello (CR)
LEGGI	LDRB R3, [R1, #4]	Leggi il registro di stato della tastiera
	TST R3, #2	Controlla se c'è un carattere
	BEQ LEGGI	Attendi finché arriva
	LDRB R3, [R1]	Leggi il carattere dalla tastiera (ciò azzera KIN)
	STRB R3, [R0], #1	Immagazzina il carattere in memoria
		Incrementa il puntatore
ECO	LDRB R4, [R2, #4]	Leggi il registro di stato dello schermo
	TST R4, #4	Controlla se lo schermo è pronto
	BEQ ECO	Attendi finché non lo è
	STRB R3, [R2]	Invia il carattere allo schermo (ciò azzera DOUT)
	TEQ R3, #CR BNE LEGGI	Se il carattere non è CR reitera la lettura di caratteri

e ARM di lettura e visualizzazione di una linea di caratteri.

	ColdFire	Commenti
	MOVEA.L #LOC, A2	Inizializza il puntatore alla prima locazione di memoria dove immagazzinare i caratteri
	MOVEA.L #KBD_STATUS, A3	Inizializza il puntatore all'indirizzo del registro di stato della tastiera
	MOVEA.L #DISP_STATUS, A4	Inizializza il puntatore all'indirizzo del registro di stato dello schermo
	CLR.L D0	Azzera il registro dati destinato ai caratteri
LEGGI:	BTST.B #1, (A3)	Attendi la lettura di un carattere nel buffer KBD_DATA della tastiera
	BEQ LEGGI	
	MOVE.B KBD_DATA, D0	Trasferisci il carattere al registro (ciò azzera KIN)
	MOVE.B D0, (A2)+	Immagazzina il carattere in memoria e incrementa il puntatore
ECO:	BTST.B #2, (A4)	Attendi che lo schermo sia pronto
	BEQ ECO	
	MOVE.B D0, DISP_DATA	Trasferisci il carattere appena letto al buffer dello schermo (ciò azzera DOUT)
	CMPI.L #CR, D0	Controlla se il carattere letto sia CR
	BNE LEGGI	Se non lo è, reitera la lettura dei caratteri

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

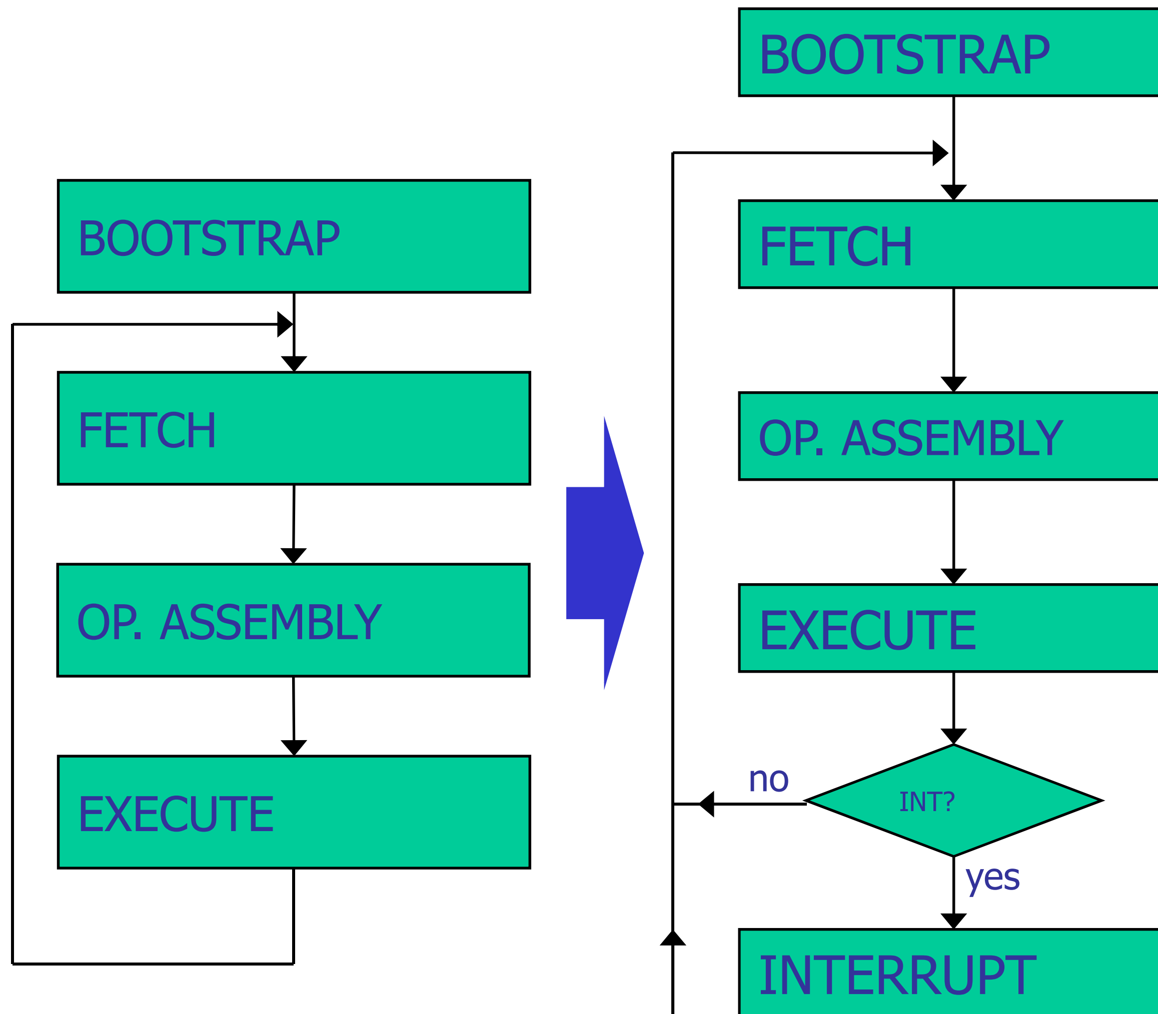
$i + 1$

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 $i + 1$

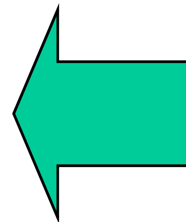
Managing Asyncronous Events



- Multitasking
- OS operations
- I/O management
- …

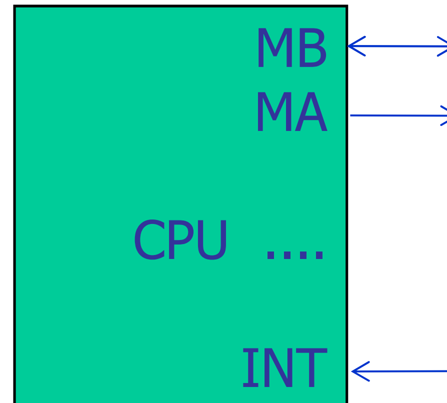
Activation

```
while (true) {
  Fetch();
  Decode();
  Execute();
  CheckForInterrupt();
}
```



Check for **exceptional** events, if none continue

The INTERRUPT stage (1/2)



- Starts when INT signal is on
- Some events are «pending» and wait to be «managed»
- Many causes can have generated the events
- The interrupt starts the service that will serve such events

The INTERRUPT stage (2/2)

INTERRUPT

- Processor mode: “Supervisor”
- During this stage the processor does not execute a program
  - To exececute a program the processor should be in the normal cycle (*fetch, execute)*
- During the *interrupt* consiste some *hardware* steps happen to “prepare” the processor to managing the interrupt

Exceptions

- Reset
  - Resets the system to the initial state
  - Generated by error conditions that cannot be recovered
- Traps
  - Allow the programmer to get the processor in supervisor status
  - Synchronous events (wrt normal execution)
- Interrupts
  - Requests from external devices (typically I/O ones)
  - Asynchronous events (wrt normal execution)

Interrupt management

- When an Interrupt happens:
  - Step 1: temporary copy of SR and set SR to manage interrupts
  - Step 2: Identify the device requesting attention and the address for the Interrupt Service Routine (ISR)
  - Step 3: save the execution context
  - Spet 4: prepare the new context and start the ISR
  - Step 5: restore the saved context and continue the normal execution

Interrupt latency

- Max time between interrupt request and the first useful instruction of the ISR