MICROPROCESSOR SYSTEMS

Lecture 7

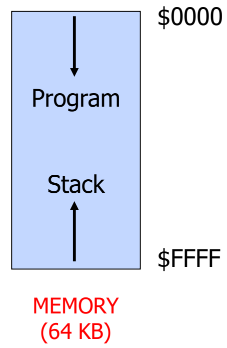
Stack, Subroutine, Interrupt

Topics

- Stack
- Subroutines
- Interrupts

Stack

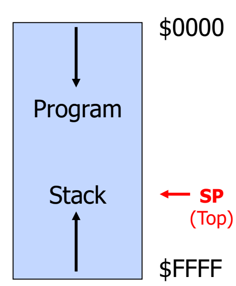
- System Stack is a **temporary storage area** in memory.
- Programmer can define the base address of the stack.
- Base of the stack is often the **highest** available memory address ($FFFF).
  - (In EDU-CPU, default base address of stack is $7FFF.)
- Program codes are placed from address $0000 (increasingly).
- Stack data are placed from address $FFFF (decreasingly).



3

Stack Pointer Register

- **Stack Pointer (SP)** register points to **Top** of stack (first empty available memory location in stack).
- SP goes upwards,
  from higher addresses to lower addresses.
- Stack is used in two ways
  - Data can be stored and removed in stack.
  - While branching to a subroutine, the **return addresses (PC=Program Counter)** is stored in stack automatically.



4

Operation of Stack

- **<u>Push instruction (PSH):</u>**
  - Stores a data to top of stack.
  - Data is placed on stack first.
  - Then stack pointer (top) is decremented.

- **<u>Pop instruction (PUL) :</u>**
  - Removes a data from top of stack.
  - Stack pointer (top) is incremented first.
  - Then data is retrieved from top of stack.

Only 8-bit accumulator registers (A or B) can be used as operands for PSH and PUL.

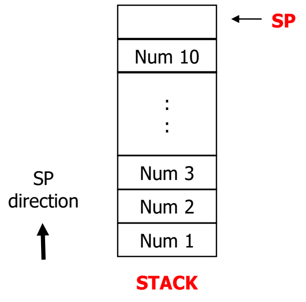
LIFO (Last In First Out)

- Stack is also known as Last-In-First-Out data structure.
- Lastly added data is removed firstly.
- Push and pop operations must be written **in reverse order**, in order **to restore data** back into its original value.
- Unbalanced PSH and PUL instructions can overwrite the program codes.

```
PSH A
PSH B
...
...
PUL B
PUL A
```

Example : Reversing an array

- Suppose there is an **Array** of ten elements (1 byte each) starting from address $1000 in memory.
- Write a program to store the numbers in the reverse order starting at the **same array address.**
- The stack will be used to store the numbers temporarily.
  - In first phase, all elements will be read from Array and stored to Stack.
  - In second phase, all elements will be read from Stack and stored back to Array.



7

Example : Reversing an array

```
*Initialize the Array at address $1000 with ten numbers.
        ORG  $1000
        DAT  10,20,30,40,50
        DAT  60,70,80,90,100

START     LDA    YG,$FFFF     ;Stack pointer
          LDA    SK,$1000     ;Array address
          LDA    B, 10        ;Loop counter
*-----------------
*Read all elements from array and add them to stack.
LOOP1     LDA    A,<SK+0>   ;Load from array
          PSH    A            ;Add to stack
          INC    SK          ; Increment SK
          DEC    B           ;Decrement counter
          CMP    B, 0        ;Compare B to 0
          BNE    LOOP1       ;Branch to Loop1

*(continued on next page)
```

8

Example : Reversing an array (continued)

***Remove all elements from stack, and store them back to array.**

```
         LDA   SK,$1000
         LDA   B, 10      ;Loop counter

LOOP2    PUL   A          ;Remove from stack
         STA   A,<SK+0>   ;Store to array

         INC   SK         ; Increment SK
         DEC   B          ;Decrement counter
         CMP   B, 0       ;Compare B to 0
         BNE   LOOP2      ;Branch to Loop2

         INT
```


Topics

- Stack
- Subroutines
- Interrupts

Subroutine

- A subroutine is a group of instructions that will be used repeatedly in different locations of program.
- Similar to functions in C language.

- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately after the main program.

Subroutine Instructions

- BSR and JSR instructions are used for calling a subroutine unconditionally.
- RTS instruction is used to return back to main program.

- Syntax of Instructions :
  - **BSR ADDRESS**
    - Branch to Subroutine
    - Uses relative addressing
  - **JSR ADDRESS**
    - Jump to Subroutine
    - Uses absolute addressing
  - **RTS**
    - Return from Subroutine

Conditional Calling of Subroutine

- The following instructions are used to conditionally call a subroutine.

- **BSC** : Branch to Subroutine Conditionally
  Syntax : ***BSC*** *status_flag_bit_name, Address*

- **JMC** : Jump to Subroutine Conditionally
  Syntax: ***JMC*** *status_flag_bit_name, Address*

- Status flag bit name can be one of the followings.
  - S : Sıfır (Zero)
  - N : Negatif (Negative)
  - E : Elde (Carry)
  - T : Taşma (Overflow)

Calling and returning from Subroutine

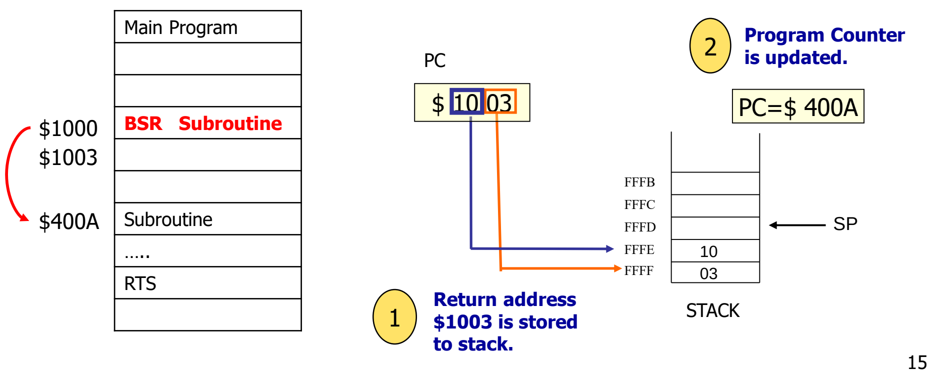
- When a subroutine is called, the return address in PC (Program Counter) is saved to Stack automatically.

- Then, PC is assigned to the target address (subroutine address) automatically.
  PC ← TARGET ADDRESS

- When RTS instruction is executed, the return address is automatically taken from Stack into PC.
  PC ← RETURN ADDRESS

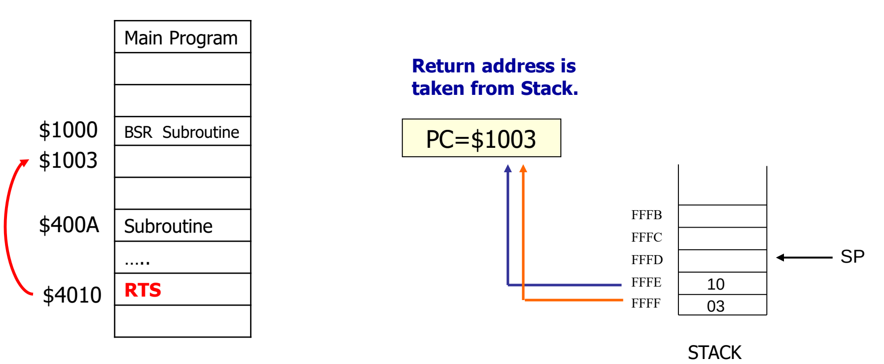
Calling: BSR Instruction

- When BSR (Branch to Subroutine) is executed, the followings are done by CPU automatically:
  - Push address of next instruction immediately following the **BSR** onto stack. (Low byte and high byte of address)
  - Load program counter with the 16-bit target address supplied with the **BSR** instruction.



Returning: RTS Instruction

- **RTS** (Return from subroutine) returns to main program.
  - Retrieve the return address from top of stack.
  - Load program counter with the return address.



Passing Data to a Subroutine

- **Method1:**
- Data (parameter) can be passed to a subroutine through **registers**.
  - Data is stored in a register by the calling program.
  - Subroutine uses value from the register.

- **Method2:**
- Use agreed upon **memory locations**.
  - The calling program stores data in a memory location (i.e. variable).
  - Subroutine retrieves the data from the location.

17

Call-by Value vs. Call-by Reference

- **Call by value:** The values of parameters are transferred to subroutine, usually in accumulator registers.

- **Call by reference:** The memory addresses of parameters are transferred to subroutine, usually in Index Register. (Similar to a pointer variable in the C language.)

18

Example1 : Call-by-value using Accumulator

- Main program calls subroutine to find **absolute value** of a number.
- Accumulator A contains the value of a signed number.

Main program

```
START   LDA   A, -87        ;number
        BSR   ABSVAL        ;call subroutine
        STA   A,<$1000>     ;result
        INT
```

Subroutine

```
ABSVAL  BIT   A, %10000000  ;Bit test (filtering the leftmost sign bit)
        BEQ   END           ;If zero flag is 1
        NEG   A             ;Negate A
END     RTS
```


Example2 : Call-by-reference using Index Register

- Address of number is sent to subroutine using the **Index Register** (SK).
- Subroutine writes absolute value over original value.

Main program

```
START  STA -87, $1000   ;number
       LDA   SK, $1000   ;address of number
       BSR   ABSVAL      ;call subroutine
       INT
```

Subroutine

```
* SK contains address of data
ABSVAL LDA   A, <SK+0>  ;load data
       BIT   A, $80          ; Bit test A and %1000 0000
       BEQ   END             ; If Zero flag is 1
       NEG   A               ;Negate A
       STA   A,<SK+0>        ;store result
END RTS
```

Using the Stack for Parameter Passing to a Subroutine

- Stack can be used also for **parameter passing.**
  - Main program writes parameters (value or address) into stack (push).
  - Subroutine reads parameters from stack (pull).
  - Stack is a shared memory by all subroutines.
  - Value of Stack Pointer (SP) is known by main program and by subroutines.

21

Example3 : Call-by-value using Stack

- Parameters (values of two numbers) are passed to Subroutine via stack.
- **Subroutine adds two numbers.**
- The result is in accumulator A.

```
START   LDA   YG, $FFFF
        LDA   A, 10 ; First number
        PSH   A
        LDA   A, 20 ; Second number
        PSH   A
        BSR   ADD_NUMBERS
        STA   A, $3000
        PUL A
        PUL A
END     INT
```

STACK	
	$FFFB ← SP
Return Address (High byte)	$FFFC
Return Address (Low byte)	$FFFD
Number2 (20)	$FFFE
Number1 (10)	$FFFF

STACK

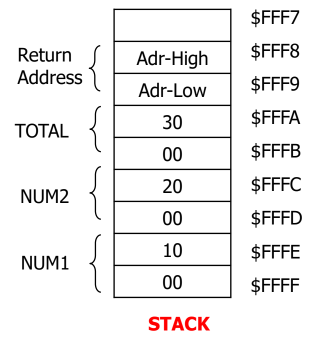
```
ADD_NUMBERS
        LDA   A,<YG+4> ; Retrieve first number
        ADD   A,<YG+3>; Retrieve second number
        RTS
```

22

Example4: Call-by-reference using Stack

```
START   LDA   YG,$FFFF
        STA   $A9, $1000 ; First number
        LDA   SK, $1000
        MOV   AB, SK     ; Using SK
        PSH   B
        PSH   A

        STA   $75, $2000 ; Second number
        LDA   A, $00     ; Not using SK
        PSH   A
        LDA   A,$20
        PSH   A

        LDA   SK, $3000  ; Result address
        MOV   AB, SK     ; Using SK
        PSH   B
        PSH   A

        BSR   SUM
        PUL   A
        PUL   A
        PUL   A
        PUL   A
        PUL   A
        PUL   A
END     INT
```

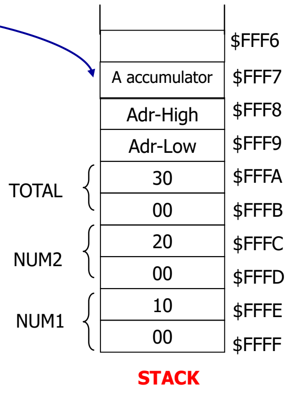
- The addresses of two numbers and address of the result are transferred to the subroutine.
- In the subroutine SUM, two numbers are added, the result (total) is stored in the destination location.



SUM Subroutine

The CD register is used as address register.

```
SUM                          ;Name label of subroutine
     PSH   A                 ;Backup A

     MOV   SK, YG    ;Copy Stack Pointer to Index Register

     LDA   CD, <SK+08>       ;Num1 address
     LDA   A, <CD>           ;Retrieve Num1

     LDA   CD, <SK+06>       ;Num2 address
     ADD   A, <CD>           ;Retrieve Num2, and add

     LDA   CD, <SK+04>       ;Total address
     STA   A, <CD>           ;Total stored

     PUL   A                 ;Restore A
     RTS                     ;Return from subroutine
```



Topics

- Stack
- Subroutines
- Interrupts

25

Interrupt

- An interrupt is a signal coming from an Input or Output device.
- When microprocessor receives an interrupt signal, it suspends the currently executing program (after executing its current instruction).
- Then jumps to an **Interrupt Service Routine (ISR)** to respond to the incoming interrupt.
- Each interrupt will have its own ISR.

26

I/O Interrupt Processing

CPU TASKS : Implemented as hardware

- **Device Controller** chip sends an interrupt signal to **CPU**
- ↓ CPU finishes the execution of current instruction
- ↓ CPU signals acknowledgement of interrupt
- ↓ CPU pushes CCR (status flags) and PC (Program Counter) onto stack
- ↓ CPU loads new PC value based on Interrupt Service Routine address

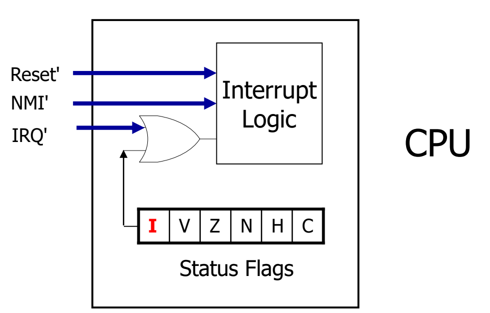
Interrupt Service Routine (ISR) :
Implemented as software

- Save all registers to stack (optional)
- ↓ Process the interrupt
- ↓ Restore all registers from stack (optional)
- ↓ Return from interrupt (CPU restores old CCR and PC from stack)

27

Interrupt signals in CPU

- **Reset:** Brings the computer into the startup state.
- **Non-maskable interrupt (NMI) :** CPU responses the interrupt always.
- **Maskable interrupt (IRQ):** CPU responses the interrupt request based on the interrupt (I) bit of status flags register.
  - If the I flag=1, then Blocking the interrupt
  - If the I flag=0, then CPU responds to interrupt request
- Priorities : **Reset > NMI > MI**



28

Interrupt vector

- An Interrupt Service Routine can be called automatically with two methods:
  Vectored or **Non-vectored**

- If user has to provide address of a ISR subroutine, then it is known as **vectored interrupt.**
  - An **interrupt vector** is a memory address which contains the address of an ISR subroutine in memory.
    (Similar to a pointer-to-function in C language.)

Interrupt Vector table

- **Interrupt Vector Table:**
  An array of interrupt vectors, containing multiple ISR addresses.

Example

Reset Routine	(H)	$FFFF
	(L)	$FFFE
Non-maskable Interrupt Routine	(H)	$FFFD
	(L)	$FFFC
Software Routine	(H)	$FFFB
	(L)	$FFFA
Maskable Interrupt Routine	(H)	$FFF9
	(L)	$FFF8

H: high byte
L: low byte

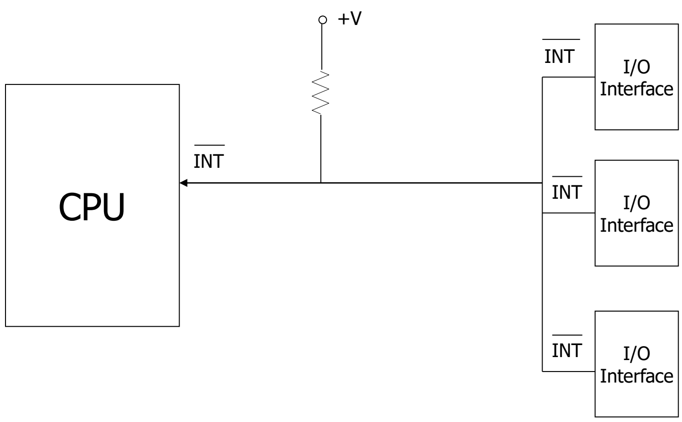
Interrupt Processing

- When an interrupt is received, Microprocessor needs to determine where the interrupt is coming from.
- Microprocessor executes the Interrupt Service Routine (ISR) associated with the interrupt source.
- There are two handling methods.
  - Polling
  - Interrupt Controller Chip

31

Polling

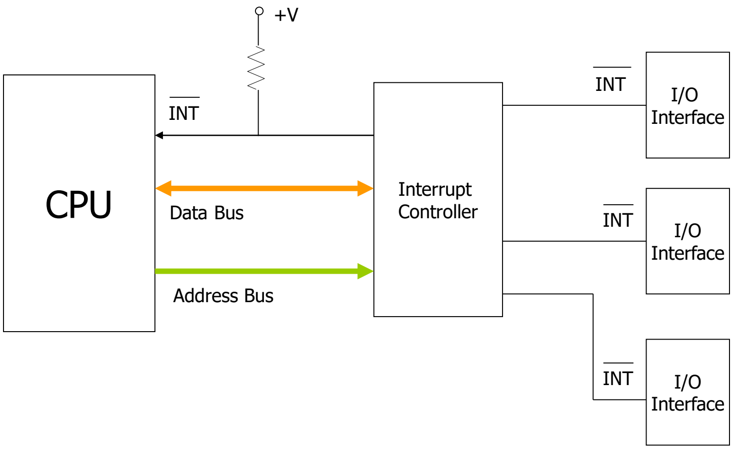
- The interrupt line is lowered if there is an interrupt.
- CPU reads status registers of Input/Output interfaces one by one, to determine which interrupt service routine will be executed.
- Polling order determines the interrupt priority.



32

Interrupt Controller chip

- When I/O interrupt occurs, interrupt controller sends it to CPU.
- CPU reads the register of the interrupt controller to determine which interrupt routine will be executed.

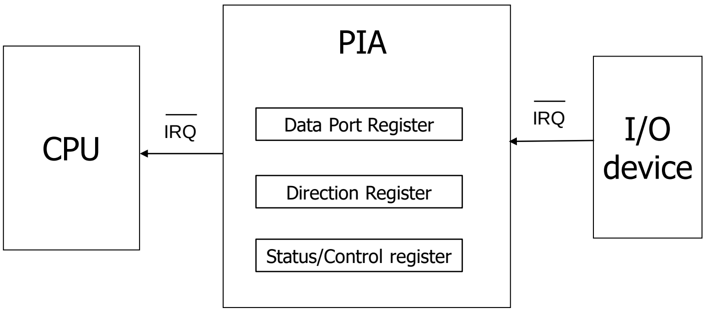


Branching to ISR

- **Interrupt Service Routines** (ISR) are like normal subroutines.
- When CPU accesses the ISR, it stores the current value of program counter (PC) onto stack.
- Interrupt routine should return to main program with the **RTI** (Return from Interrupt) instruction.
- Upon return from the interrupt routine, CPU continues with where it left.

Example : Counting the interrupt events

- **PIA** (Peripheral Interface Adapter) chip is connected to CPU and I/O device.
- I/O device (such as a **button**) can send interrupt signal to PIA, then PIA can send it to CPU.
- **Purpose :** Count **how many times** interrupt signal is received by CPU.



IRQ : Interrupt Request signal

35

Example (continued)

- Default interrupt vector address in Educational CPU is $FFF8 and $FFF9.
- The vector can be assigned to address of an ISR (Interrupt Service Routine).

- **Main program tasks:**
  - Store the address (`$C000`) of the ISR in the vector.
  - Configure the PIA status/control register for interrupts.
  - Enable the CPU interrupt.
  - In an endless loop, program waits for an interrupt.

- **ISR routine tasks:**
  - Increment the event counter variable.
  - Return back to main program.

36

Example (continued)

Note: Interrupts are not implemented in EDU-CPU.

Main program

```
*Interrupt events counter variable
COUNTER   RMB 1

START
*ISR address $C000 is stored in interrupt vector ($FFF8 and $FFF9)
        STA $C0, $FFF9        ; High byte of ISR location
        STA $00, $FFF8        ; Low byte of ISR location

*Configure the status/control register of PIA.A
*(Allow IRQ input of PIA)
        STA %00001010, <DURDEN.A>

        CLR  <COUNTER> ; Clear counter variable
        EIN            ; Enable IRQ input of CPU
WAIT    BRA WAIT       ; Endless loop
```

ISR

```
* Interrupt Service Routine (ISR) is called automatically when an interrupt occurs
        ORG $C000  ; Address of the ISR
        INC <COUNTER>  ; Increment counter
        RTI  ; Return from interrupt
```

37