Microprocessor Systems

Dr. Gökhan İnce

Topics

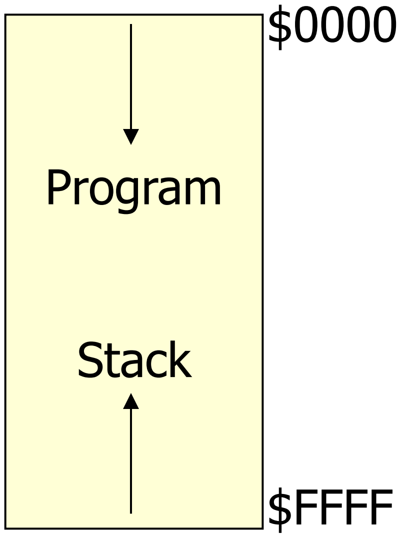
- Stack
- Subroutine
- Interrupt

Stack

- Stack is a temporary storage area in memory identified by the programmer
  - Variables are stored temporaly
  - Program addresses are stored temporaly
  - Connection to subroutine
- Last in first out (LIFO) structure
- The stack normally grows backwards into memory.
  - Programmer defines the bottom address of the stack and the stack grows up into reducing address range.

Stack

- The stack grows from higher addresses to lower so that the top of the stack grows upwards.
- Because of this, the bottom of the stack is often located at the highest available memory address.
- It is used in two ways
  - Data is stored in the program
    - By `PUSH(PSH)` and `POP` instructions
  - While branching to the subroutine or branching to interrupt service routine the return addresses (PC) and parameters are pushed onto stack



Stack in Educational CPU

- There is a 16-bit stack pointer (SP) in the Educational CPU
- Stack pointer points the next available address in the stack
- The accumulator contents can be pushed to stack.
- The stack contents can be pulled to the accumulators
- The stack grows from higher addresses to lower

Saving Information on the Stack

- Stack pointer is set to the suitable address

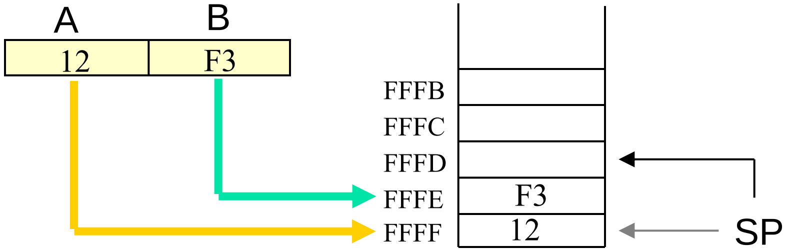
```
LDA SP, $FFFF
```

- To store data to the stack `PSH A`
  - ACC A is stored at the address pointed with SP

    <SP> ← A

  - SP is decremented by one to point the next available address in the stack

    SP ← SP-1



The POP Instruction

- In order to retrieve data from stack:

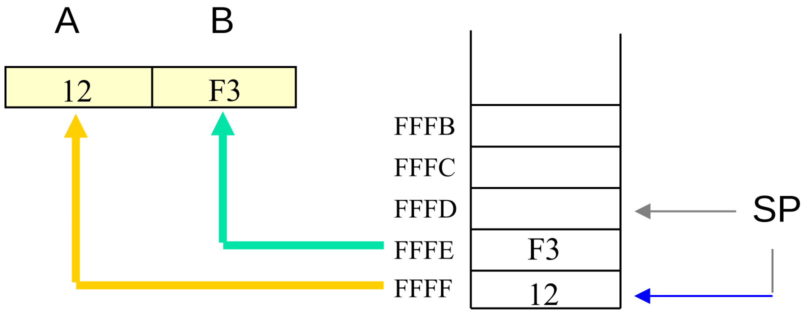
```
POP B
```

  - Increment SP

```
SP ← SP + 1
```

  - Copy the contents of the memory location pointed by the SP to accumulator B

```
B ← <SP>
```



Operation of the Stack

- During pushing, the stack operates in a “store then decrement” style.
  - the information is placed on the stack first, then the stack pointer is decremented.

- During popping, the stack operates in a “increment then use” style.
  - The pointer is incremented and then the information is retrieved from the top of the stack

- The SP always points to “the top of the stack”.

LIFO

- Push and pop operations must be done in reverse order in order to retrieve information back into its original location.

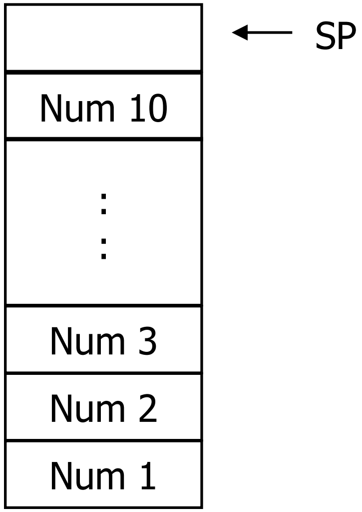
```
PUSH A
PUSH B
:
:
POP B
POP A
```

- Push too much without popping and you can overwrite your program! It's your responsibility not to do this.

Example-1

There is an array of ten elements starting from address $1000 in memory.
They will be stored in reverse order starting at the same address.

```
START    LDA   SP,$2000
         LDA   IX,$1000
         LDA   B,$0A
BACK     BEQ   NEXT
         LDA   A,<IX+0>
         INC   IX
         PSH   A
         DEC   B
         BR    BACK
NEXT     LDA   IX,$1000
         LDA   B,$0A
LOOP     BEQ   END
         POP   A
         STA   A,<IX+0>
         INC   IX
         DEC   B
         BR   LOOP
END      SWI
```



Topics

- Stack
- Subroutine
- Interrupt

Subroutine

- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.
  - Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.
- In Assembly language, a subroutine can exist anywhere in the code.
  - However, it is customary to place subroutines separately from the main program.

Subroutine- Instructions

- The instructions that is used for calling subroutine and returning from subroutine in educational CPU:
  - BSR ADDRESS: Calling the subroutine in the ADDRESS.

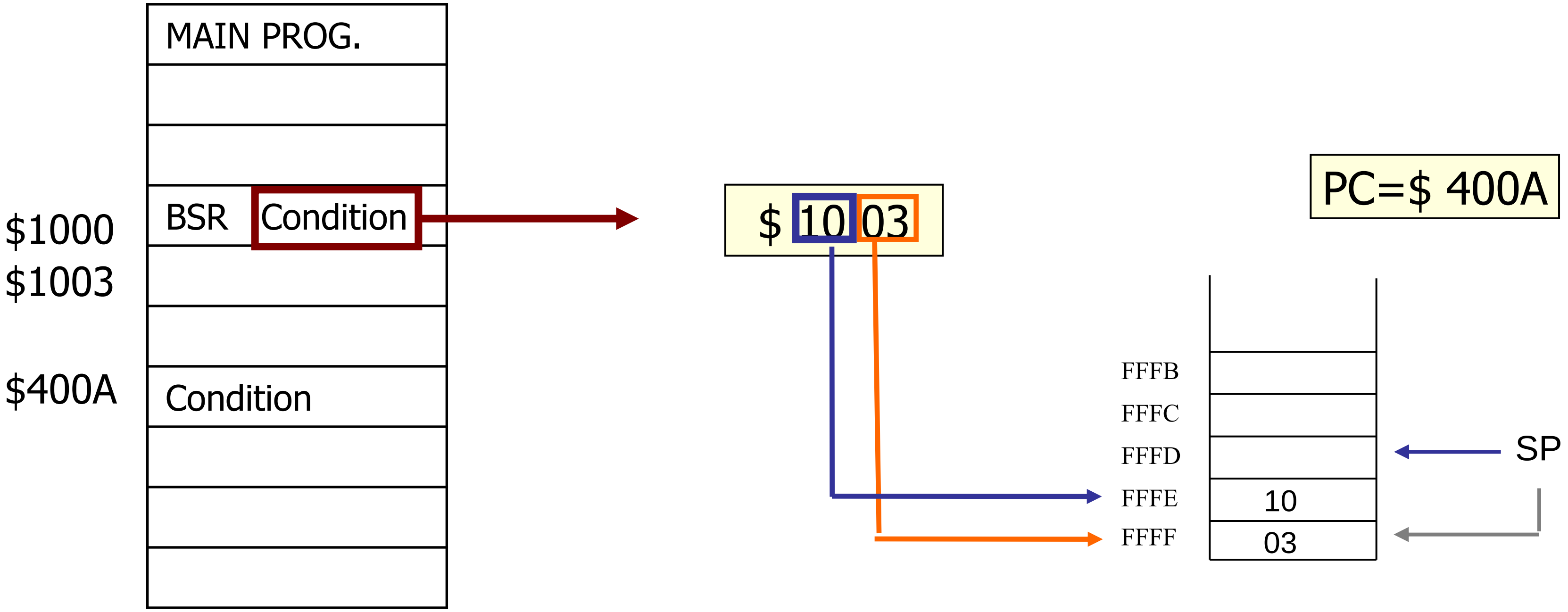
    <SP> ← PC(low), SP ← SP-1
    <SP> ← PC(high), SP ← SP-1
    PC ← ADDRESS
  - BSR STEP: Calling the subroutine that has address of PC+STEP

    <SP> ← PC(low), SP ← SP-1
    <SP> ← PC(high), SP ← SP-1
    PC ← PC + STEP
  - RTS: Return back to main program.

    SP ← SP + 1, PC(high) ← <SP>
    SP ← SP + 1, PC(low) ← <SP>

BSR Instruction

- When BSR is executed:
  - Push the address of the instruction immediately following the **`BSR`** onto the stack
  - Load the program counter with the 16-bit address supplied with the **`BSR`** instruction.



MAIN PROG.

$1000 BSR Condition

$1003

$400A Condition

$ 10 03

PC=$ 400A

FFFB

FFFC

FFFD

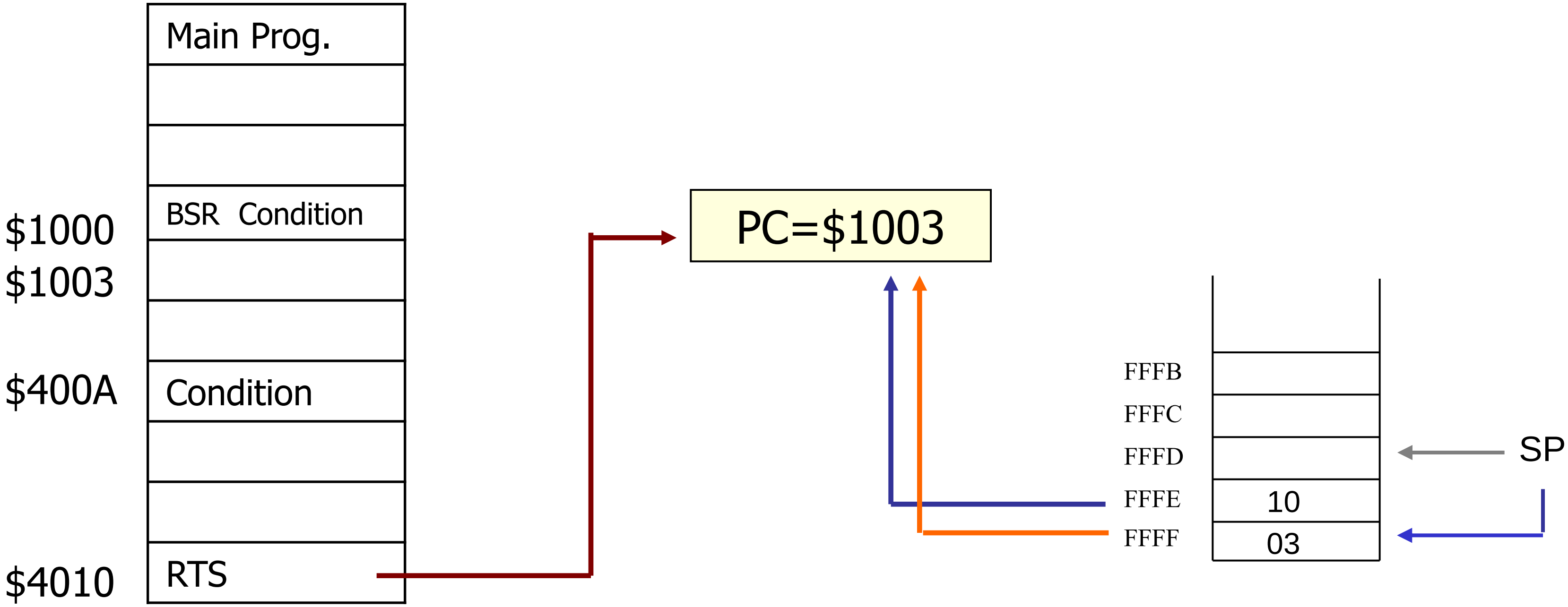
FFFE 10

FFFF 03

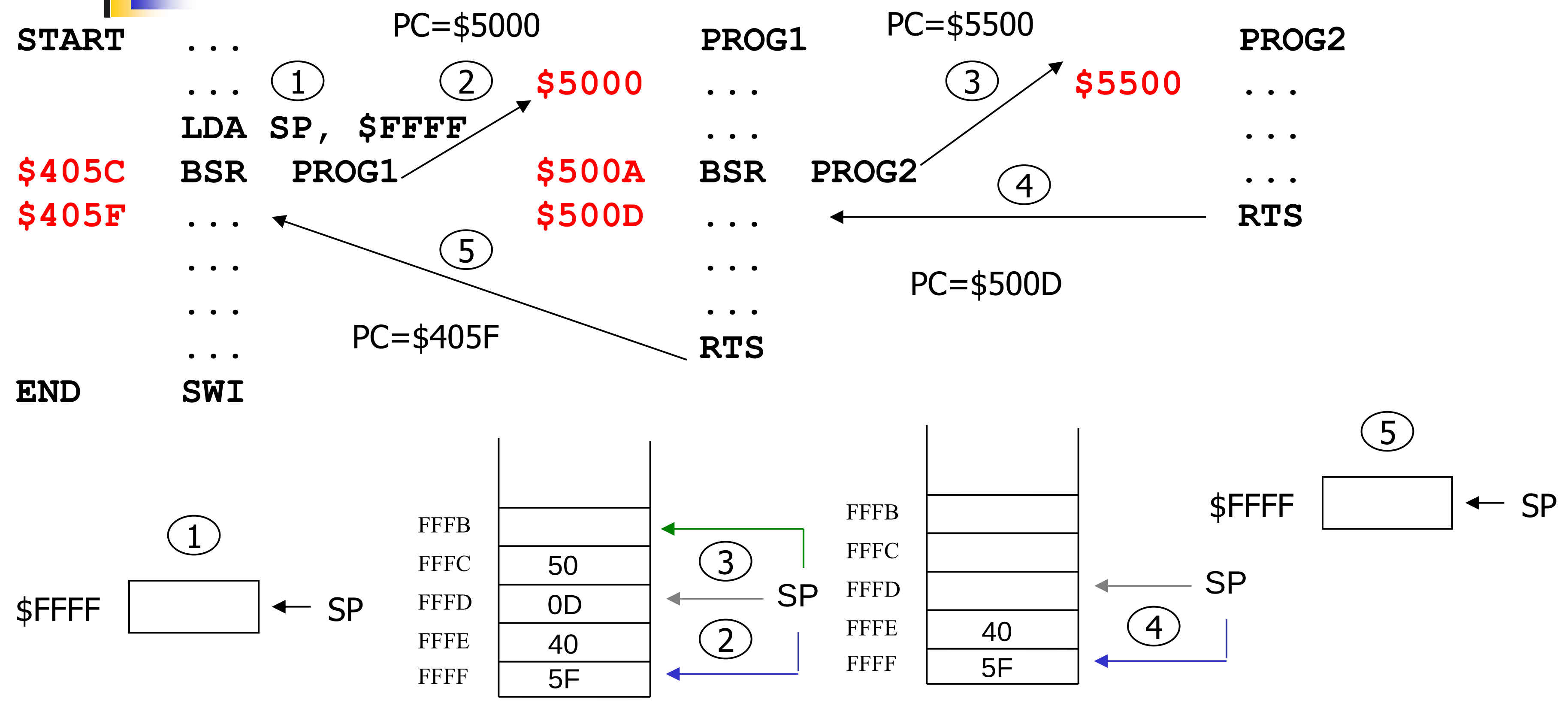
SP

RTS Instruction

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



Subroutine



```
START   ...
        ...
        LDA SP, $FFFF
$405C   BSR  PROG1
$405F   ...
        ...
        ...
        ...
END     SWI
```

```
        PROG1
$5000   ...
        ...
$500A   BSR  PROG2
$500D   ...
        ...
        ...
        RTS
```

```
        PROG2
$5500   ...
        ...
        ...
        RTS
```

① ② PC=$5000

③ PC=$5500

④ PC=$500D

⑤ PC=$405F

①

$FFFF ← SP

②③ SP

FFFB	
FFFC	50
FFFD	0D
FFFE	40
FFFF	5F

④ SP

FFFB	
FFFC	
FFFD	
FFFE	40
FFFF	5F

⑤

$FFFF ← SP

You must initialize the stack in any program that calls subroutines. If you don't, return addresses may overwrite other important data, or not be recorded because the memory at the address the SP points to is either ROM or just empty space (i.e., not installed).

Example-2

A program to find the absolute value of numbers.

Compute magnitude (absolute value) of single-byte signed number in A. Return unsigned result in A.

```
START     LDA    SP,$FFFF
          LDA    A,$A9
          STA    A,<$1000>
          LDA    A,$75
          STA    A,<$100A>
          BSR    ABSVAL
          LDA    A,<$1000>
          BSR    ABSVAL
          STA    A,<$1000>
END       SWI
----------------------------
ABSVAL    TST    A,$80
          BEQ    END
          NEG    A

END       RTS
```

Passing Data to a Subroutine

- In Assembly Language data is passed to a subroutine through registers.
  - The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.
- The other possibility is to use agreed upon memory locations.
  - The calling program stores the data in the memory location and the subroutine retrieves the data from the location and uses it.
- Stack can be used for parameter passing
  - Data are pushed to stack before calling the subroutine.

Call by Reference– Call by Value

- Call by reference: The addresses of the parameters are transferred to the subroutine.

- Call by value: The values of parameters are transferred to the subroutine.

Values of the parameters are transferred to subroutine in Example 2

If the address of a parameter is passed to the subroutine, the contents of this memory location (original value of the parameter) can be modified by the subroutine.

Example-3

We write the previous example again. This time the address of the number is sent to the subroutine using the X register. The subroutine writes the result over original value.

```
START     LDA    SP,$FFFF
          LDA    A,$A9
          STA    A,<$1000>
          LDA    A,$75
          STA    A,<$100A>
          LDA    IX,$100A
          BSR    ABSVAL
          LDA    A,<$1000>
          BSR    ABSVAL
END       SWI
---------------------------------
ABSVAL    PSH    A
          LDA    A,<IX+0>
          TST    A,$80
          BEQ    FNSHD
          NEG    A
          STA    A,<IX+0>
FNSHD     POP    A
          RTS
```

Call by reference using index register

The role of the stack in subroutine calls

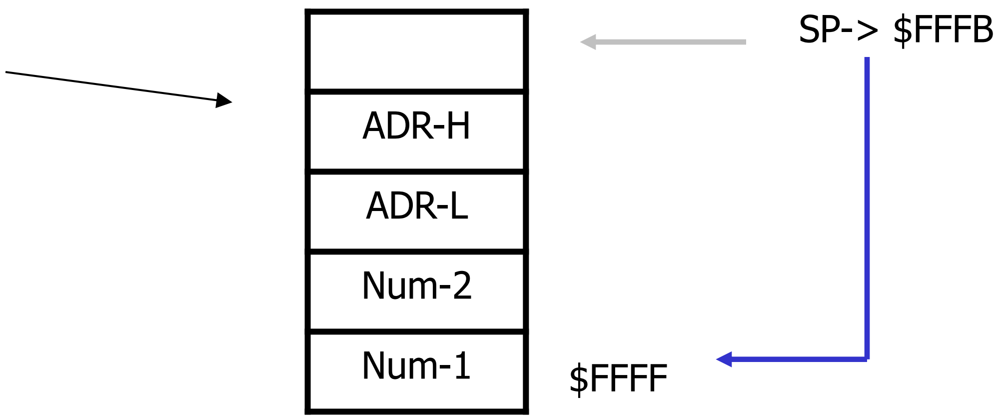
- Stack saves the return address for the PC
- Stack can be used for parameter passing
  - The main program writes parameters (value or address) into stack (push) and the subroutine reads these parameters from stack (pull). The stack is a shared memory. The value of SP is known by main and the subroutine.
- Stack can be used to store the register values
- Stack can be used to allocate local variables for the subroutine

Example-4

A subroutine to add two numbers. Parameters are passed via stack, using call-by-value technique. Return value is in A.

```
START     LDA    SP,$FFFF
          LDA    A,$59
          STA    A,<$1000>
          PSH    A
          LDA    A,$75
          STA    A,<$1005>
          PSH    A
          BSR    ADDTN
```

```
ADDTN     LDA    A,<SP+4>
          ADD    A,<SP+3>
```



SP-> $FFFB

ADR-H

ADR-L

Num-2

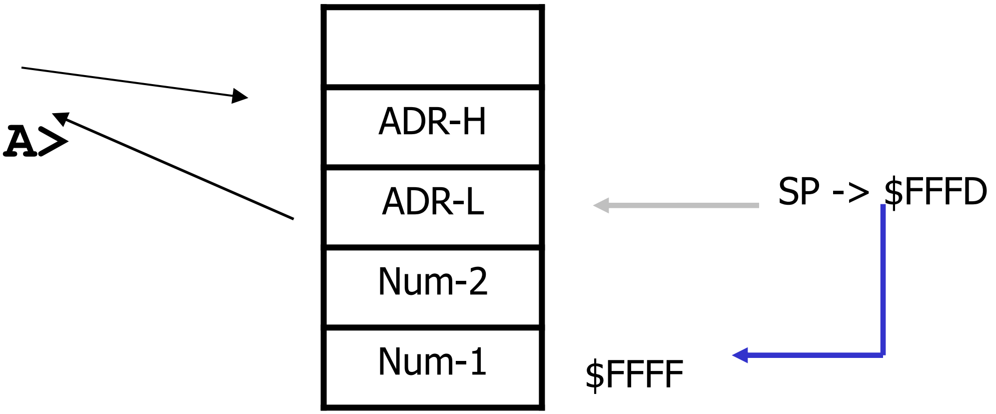
Num-1

$FFFF

Example-4

A subroutine to add two numbers. Parameters are passed via stack, using call-by-value technique. Return value is in AccA.

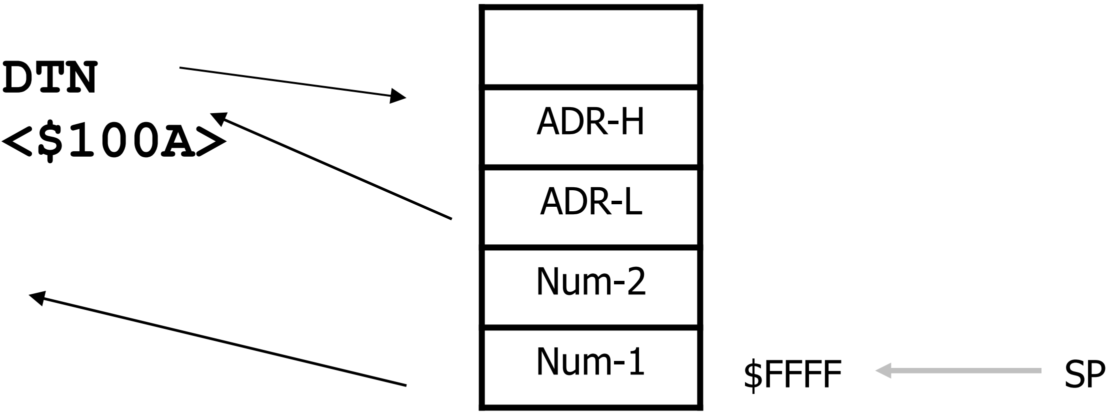
```
START   LDA   SP,$FFFF          ADDTN   LDA   A,<SP+4>
        LDA   A,$59                     ADD   A,<SP+3>
        STA   A,<$1000>                 RTS
        PSH   A
        LDA   A,$75
        STA   A,<$1005>
        PSH   A
        BSR   ADDTN
        STA   A,<$100A>
```



Example-4

A subroutine to add two numbers. Parameters are passed via stack, using call-by-value technique. Return value is in AccA.

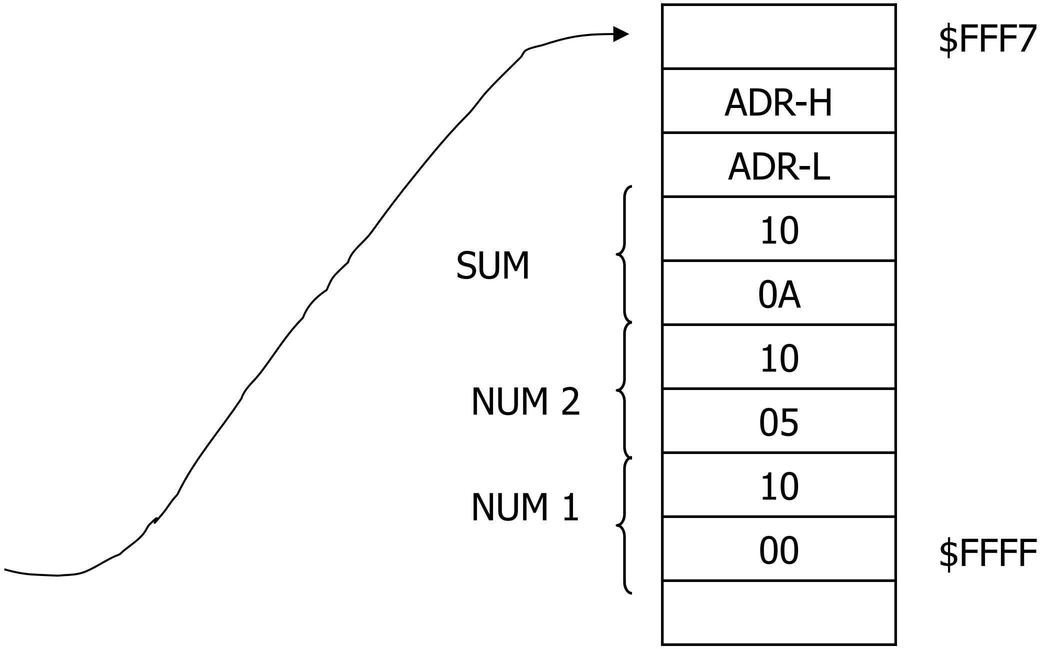
```
START    LDA   SP,$FFFF          ADDTN    LDA   A,<SP+4>
         LDA   A,$59                      ADD   A,<SP+3>
         STA   A,<$1000>                  RTS
         PSH   A
         LDA   A,$75
         STA   A,<$1005>
         PSH   A
         BSR   ADDTN
         STA   A,<$100A>
         POP   A
         POP   A
END      SWI
```



Example-5

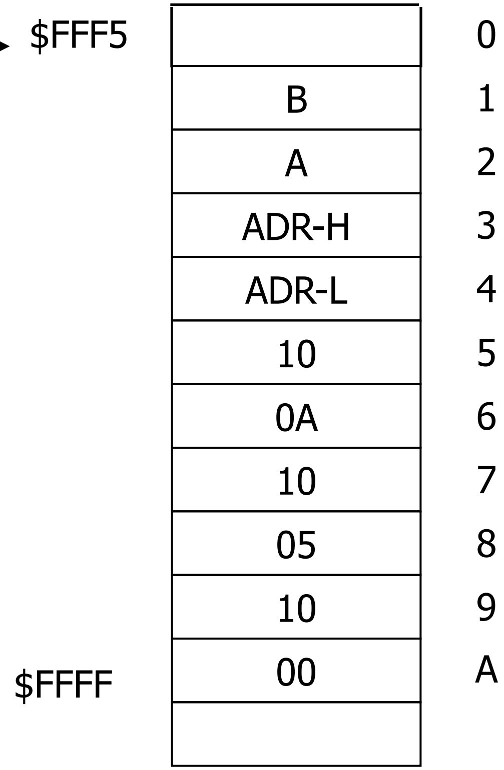
The addresses of two numbers to be added and the addresses of the result are transferred to the subroutine. In the subroutine (SUMCLR) two numbers are added, these numbers are cleared and the result is put in the destination location.

```
START   LDA    SP,$FFFF
        LDA    A,$59
        STA    A,<$1000>
        LDA    IX,$1000
        MOV    AB,IX
        PSH    B
        PSH    A
        LDA    A,$75
        STA    A,<$1005>
        LDA    A,$05
        PUSH   A
        LDA    A,$10
        PSH    A
        LDA    IX,$100A
        MOV    AB,IX
        PSH    B
        PSH    A
        BSR    SUMCLR
END     SWI
```



Example-5

```
SUMCLR   PSH   A
         PSH   B
         MOV   IX,SP
         CLR   B
         LDA   CD,<IX+09>    *Num1 address*
         LDA   A,<CD>        *ACCA <- Num1*
         STA   B,<CD>         *Clear Num1*
         LDA   CD,<IX+07>    *Num2 address*
         ADD   A,<CD>        *ACCA <- Num1+Num2*
         STA   B,<CD>        *Clear Num2 *
         LDA   CD,<IX+05>    *Sum address*
         STA   A,<CD>        *Sum stored*
         POP   B
         POP   A
         RTS
```



$FFF5		0
	B	1
	A	2
	ADR-H	3
	ADR-L	4
	10	5
	0A	6
	10	7
	05	8
	10	9
$FFFF	00	A

Local Variables

- Local variables are kept in stack by the subroutine. The subroutine must allocate and deallocate necessary memory space for local variables.

Example: Swap the high and low bytes of a 16-bit number stored in memory.

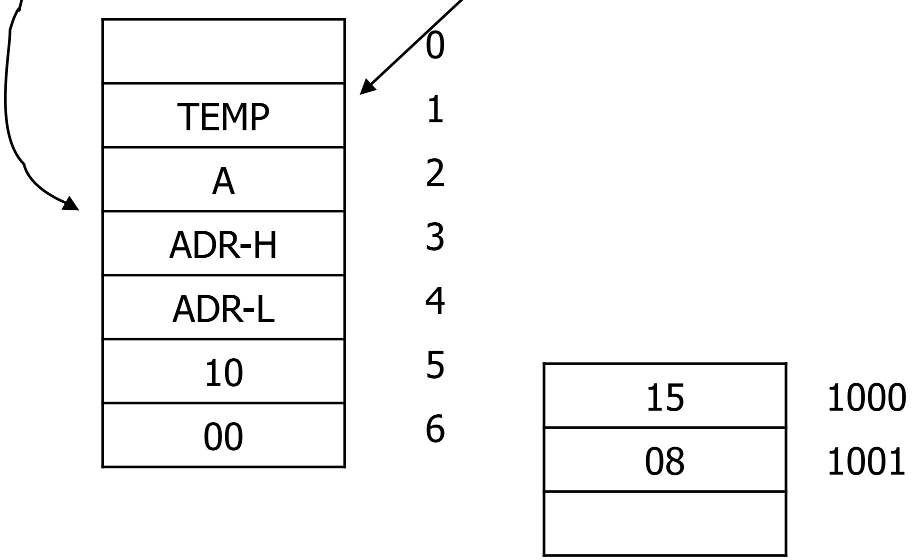
C:

```
void swap(short int *number)
{
   char temp;
   temp = *number;
   *number = *(number + 1);
   *(number + 1) = temp;
}
```

Example-6

```
START    LDA   SP,$FFFF
         LDA   IX,$1000
         MOV   AB,IX
         PSH   B
         PSH   A
         LDA   A,$44
         BSR   SWAP
```

```
SWAP     PSH   A
         DEC   SP        *place for local variable *
```



	0
TEMP	1
A	2
ADR-H	3
ADR-L	4
10	5
00	6

15	1000
08	1001

```
void swap(short int *number)
{
        char temp;
```

Example-6

```
START    LDA    SP,$FFFF
         LDA    IX,$1000
         MOV    AB,IX
         PSH    B
         PSH    A
         LDA    A,$44
         BSR    SWAP
```

```
              SWAP   PSH   A
                     DEC   SP           *place for local variable *
IX  ← FFF9           MOV   IX,SP
CD ← 1000            LDA   CD,<IX+05>   *address for high byte*
A ← 15               LDA   A,<CD>       *ACCA <- High byte*
```

	0
TEMP	1
A	2
ADR-H	3
ADR-L	4
10	5
00	6

15	1000
08	1001

```
void swap(short int *number)
{
      char temp;
```

Example-6

```
START     LDA    SP,$FFFF
          LDA    IX,$1000
          MOV    AB,IX
          PSH    B
          PSH    A
          LDA    A,$44
          BSR    SWAP
```

```
          SWAP    PSH    A
                  DEC    SP          *place for local variable *
IX ←FFF9          MOV    IX,SP
CD← 1000          LDA    CD,<IX+05>  *address for high byte*
A← 15             LDA    A,<CD>      *ACCA <- High byte*
                  STA    A,<IX+1>    *temp <- High byte *
CD←1001           INC    CD          *address for low byte*
A← 08             LDA    A,<CD>      *ACCA <- Low byte*
CD←1000           DEC    CD
```

	0
15	1
A	2
ADR-H	3
ADR-L	4
10	5
00	6

15	1000
08	1001

```
void swap(short int *number)
{
        char temp;
        temp = *number;
```

Example-6

```
START      LDA    SP,$FFFF
           LDA    IX,$1000
           MOV    AB,IX
           PSH    B
           PSH    A
           LDA    A,$44
           BSR    SWAP
```

```
             SWAP      PSH    A
                       DEC    SP            *place for local variable *
IX ←FFF9               MOV    IX,SP
CD← 1000               LDA    CD,<IX+05>    *address for high byte*
A← 15                  LDA    A,<CD>        *ACCA <- High byte*
                       STA    A,<IX+1>      *temp <- High byte *
CD←1001                INC    CD            *address for low byte*
A← 08                  LDA    A,<CD>        *ACCA <- Low byte*
CD←1000                DEC    CD
                       STA    A,<CD>        *swap*
A← 15                  LDA    A,<IX+1>      *ACCA <- temp*
CD←1001                INC    CD
```

	0
15	1
A	2
ADR-H	3
ADR-L	4
10	5
00	6

08	1000
08	1001

```
void swap(short int *number)
{
        char temp;
        temp = *number;
        *number = *(number + 1);
```

Example-6

```
START     LDA    SP,$FFFF
          LDA    IX,$1000
          MOV    AB,IX
          PSH    B
          PSH    A
          LDA    A,$44
          BSR    SWAP
END       SWI
```

```
             SWAP     PSH    A
                      DEC    SP           *place for local variable *
SK ← FFF9             MOV    IX,SP
CD ← 1000             LDA    CD,<IX+05>   *address for high byte*
A ← 15                LDA    A,<CD>       *ACCA <- High byte*
                      STA    A,<IX+1>     *temp <- High byte *
CD ← 1001             INC    CD           *address for low byte*
A ← 08                LDA    A,<CD>       *ACCA <- Low byte*
CD ← 1000             DEC    CD
                      STA    A,<CD>       *swap*
A ← 15                LDA    A,<IX+1>     *ACCA <- temp*
CD ← 1001             INC    CD
                      STA    A,<CD>       *high byte to <$1001>*
                      INC    SP *deallocate temp*
                      POP    A
                      RTS
```

	0
15	1
A	2
ADR-H	3
ADR-L	4
10	5
00	6

08	1000
15	1001

```
void swap(short int *number)
{
        char temp;
        temp = *number;
        *number = *(number + 1);
        *(number + 1) = temp;
}
```

Attention!!

- In **`PUSH`** and **`POP`** instructions, the variables should be in reverse order (LIFO)
- # of **`PUSH`** instructions should equal the # of **`POP`** instructions
  - Otherwise wrong return address is retrieved from stack by RTS instruction. So program gives error.
- It is not recommended to have **`PUSH`** and **`POP`** within loops
- Return from subroutine with RTS instruction

Topics

- Stack
- Subroutine
- Interrupt

Interrupt

- Interrupt can be considered to be an emergency signal.
  - The Microprocessor should respond to it as soon as possible.
  - When the Microprocessor receives an interrupt signal, it suspends the currently executing program (after executing its current instruction) and jumps to an Interrupt Service Routine (ISR) to respond to the incoming interrupt.
  - Each interrupt will most probably have its own ISR.

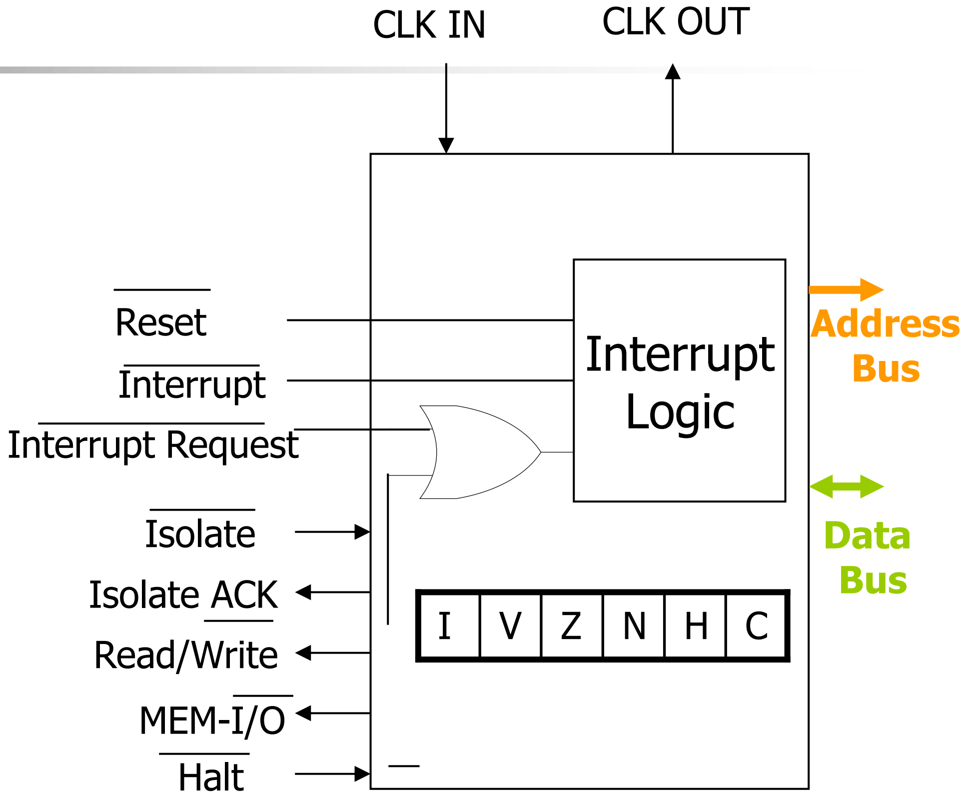
Interrupt

- Non-maskable interrupt: CPU responses the interrupt
- Maskable interrupt:

CPU responses the interrupt based on the interrupt bit of status register.

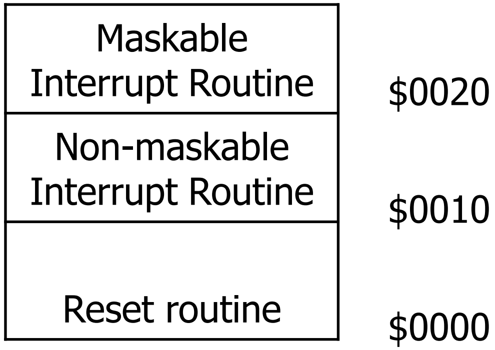
  - I=1: Blocking the interrupt
  - I=0 CPU responds to interrupt request

- Reset: Brings the computer into the startup state.
- Priorities: Reset>NMI>MI



Interrupt

- Responding to an interrupt may be immediate or delayed depending on whether the interrupt is maskable or non-maskable and whether interrupts are being masked or not.
- There are two ways of redirecting the execution to the ISR depending on whether the interrupt is vectored or non-vectored.
  - The vector is already known to the Microprocessor
  - The device will have to supply the vector to the Microprocessor



Maskable Interrupt Routine	$0020
Non-maskable Interrupt Routine	$0010
Reset routine	$0000

Interrupt Vector

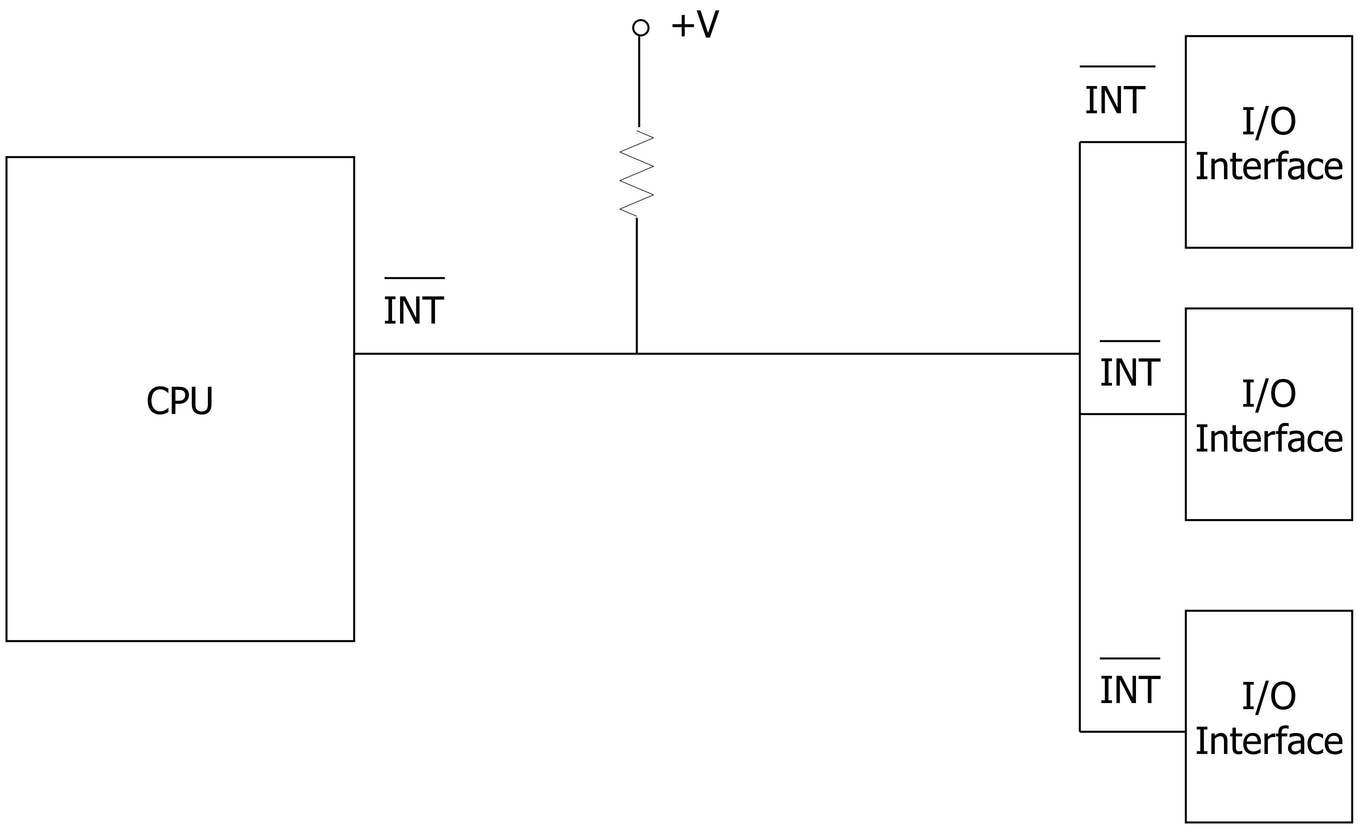
- An interrupt vector is a pointer to where the ISR is stored in memory.
- Interrupt Vector Table: A table of interrupt vectors

Reset	(H)	$FFFF
Routine	(L)	$FFFE
Non-maskable	(H)	$FFFF
Interrupt Routine	(L)	$FFFE
Software	(H)	$FFFF
Routine	(L)	$FFFE
Maskable	(H)	$FFFF
Interrupt Routine	(L)	$FFFE

Interrupt in Educational CPU

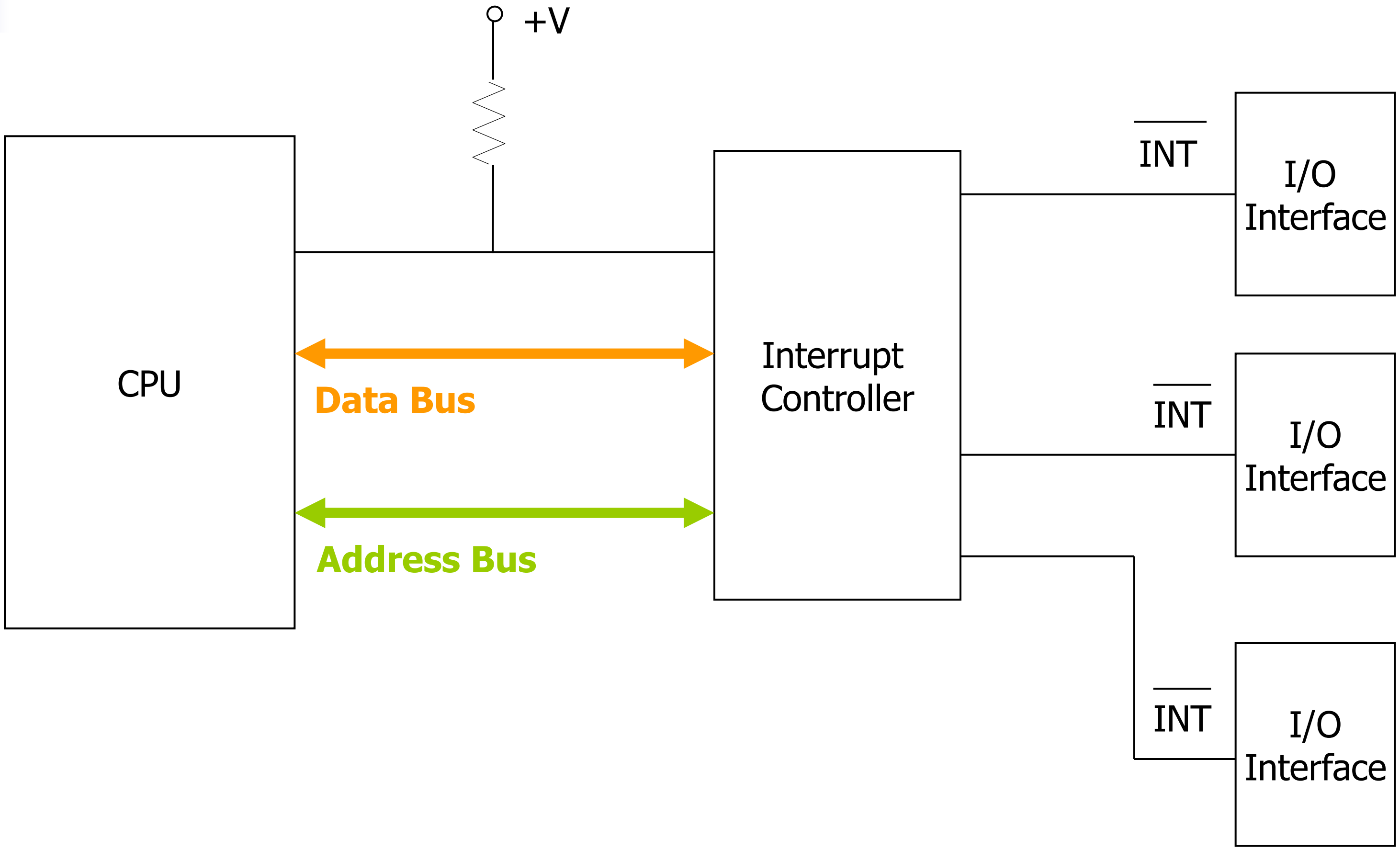
- When a hardware NMI or IRQ is received:
  - Need to understand where the interrupt is coming from with:
    - Polling : Check the PIA and ACIA status bits. Polling order determines the interrupt priority.
    - Interrupt Controller Chip : Configures interrupt priorities and indicates interrupt source
  - Execute the interrupt routine associated with the interrupt source

Polling



- The interrupt line is lowered if there is an interrupt.
- CPU reads status registers of I/Os to determine which interrupt service routine will be executed

Interrupt Controller



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

Branching to ISR

- Interrupt service routines are like subroutines.
- When the CPU accesses the ISR, it pushes the current value of the program counter (PC) onto stack
- Upon return from the interrupt, the CPU should continue with where it left from with the same register contents
- The interrupt routine returns to the main program with the RTI (Return from Interrupt) instruction

I/O Interrupt Processing

Device Controller issues an interrupt

CPU finishes of execution of current instruction

CPU signals acknowledgement of interrupt

CPU pushes CCR and PC onto stack

CPU loads new PC value based on interrupt

save the CPU's registers

process interrupt

Restore CPU's registers

Restore old CCR and PC