Procedure & STACK

Nahin Ul Sadad
Lecturer
CSE, RUET

Sample Code 1: Summation of Series (using Registers)

Pseudo Code

`COUNT = 5`
`NUM = 6`
 	
`WHILE NUM > COUNT`
`      SUM = SUM + NUM`
`      NUM = NUM + 1`
 	
`FINISH (NO OPERATION LEFT)`

Translating pseudo code to assembly code

Sample Code 1: Summation of Series (using Registers)

Assembly Code

```
XOR R0, R0   # Reset NUM (R0)
XOR R1, R1   # Reset SUM (R1)
XOR R2, R2   # Reset COUNT (R2)

ADD R2, 5    # COUNT (R2) = 5
ADD R0, 1    # NUM (R0) = 1

START:
    CMP R0, R2   # NUM (R0) > COUNT (R2)
    JG FINISH    # WHILE (NUM (R0) > COUNT (R2)), IF TRUE GOTO FINISH
    ADD R1, R0   # SUM (R1) = SUM (R1) + NUM (R0)
    ADD R0, 1    # NUM (R0) = NUM (R0) + 1
    JMP START    # GOTO START OF WHILE LOOP
FINISH:

Exit:           # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit        # PUT CPU INTO NOP (NO OPERATION)
```

Sample Code 1: Summation of Series (using Registers)

Main Memory (RAM)

Address	Instruction/Data
00	0000100000000
01	0000100010010
02	0000100100100
03	0101010100101
04	0101010000001
05	0010110000100
06	1001010001010
07	0001010010000
08	0101010000001
09	1000000000101
10	1000000001010

Register Set

Register No	Data
R0	0110
R1	1111
R2	0101
R3	0000
R4	0000
R5	0000
R6	0000
R7	0000

Figure: Data inside RAM and Registers after program execution.

Process

Process:
A process is an instance of an executing application.

An application is a file containing a list of instructions stored in the disk (often called an executable file), in flash memory, maybe in the cloud but it's not executing, it's a static entity.

When an application is launched, it is loaded into the memory and it becomes a process, so it is an active entity with a program counter specifying the next instruction to execute and a set of associated resources.

If the same program is launched more than once than multiple processes will be created executing the same program but will be having a very different state.

Process in Memory

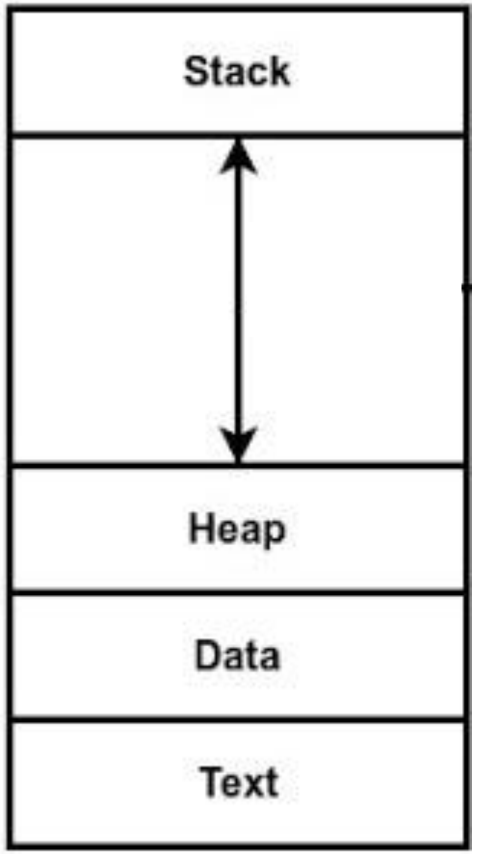


Figure: Process in Memory

1. **Stack:** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2. **Heap:** This is dynamically allocated memory to a process during its run time.
3. **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4. **Data:** This section contains the global and static variables.

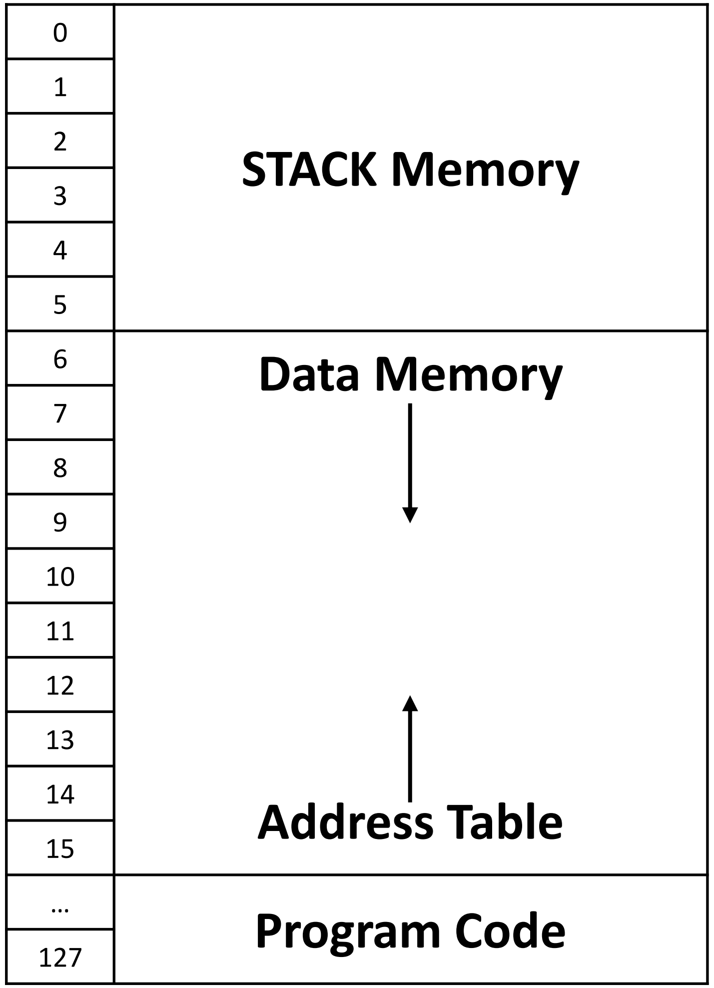
Process Address Space

1. **Logical Address Space:** The set of all logical addresses generated by a program is referred to as a logical address space.
2. **Physical Address Space:** The set of all physical addresses (in RAM) corresponding to these logical addresses is referred to as a physical address space.

Physical Address = Logical Address + Relocation Value

Implementing Simple Process Model in CPU

In order to use variables in program, we need to create simple process model for our CPU (Not real-life one). Here, all processes will share same STACK Memory, Data Memory and Address Table.



1. **Stack Memory:** This STACK memory is used for passing parameters to functions/procedures. (Address range 0-5)
2. **Data Memory:** Data memory is used to store local variables. It grows downward. (Address range 6-15)
3. **Address Table:** Address table is used to store addresses. It grows upward. (Address range 15-6)
4. **Program Code:** Program Code is the program itself. (Address range 16-127)

Implementing Simple Process Model in CPU

At first, we have to convert logical address to physical address. Following keywords will be used to set relocation value

```
START 16
```

Physical Address = Logical Address + Relocation Value

= Logical Address + 16

Program will be loaded at address 16 in RAM. It will fill all the previous addresses (0-15) with Zeroes (0).

Implementing Simple Process Model in CPU

Adding START in sample code 1.

Assembly Code

```
START 16      # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
              # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
              # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

XOR R0, R0    # Reset NUM (R0)
XOR R1, R1    # Reset SUM (R1)
XOR R2, R2    # Reset COUNT (R2)

ADD R2, 5     # COUNT (R2) = 5
ADD R0, 1     # NUM (R0) = 1

START:
    CMP R0, R2    # NUM (R0) > COUNT (R2)
    JG FINISH     # WHILE (NUM (R0) > COUNT (R2)), IF TRUE GOTO FINISH
    ADD R1, R0    # SUM (R1) = SUM (R1) + NUM (R0)
    ADD R0, 1     # NUM (R0) = NUM (R0) + 1
    JMP START     # GOTO START OF WHILE LOOP
FINISH:

Exit:            # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit     # PUT CPU INTO NOP (NO OPERATION)
```

Implementing Simple Process Model in CPU

Logical Address (Programmer's Perspective)

Logical Address	Instruction/Data
00	0000100000000
01	0000100010010
02	0000100100100
03	0101010100101
04	0101010000001
05	0010110000100
06	1001010001010
07	0001010010000
08	0101010000001
09	1000000000101
10	1000000001010

Physical Address (Stored in RAM/CPU's Perspective)

Physical Address	Instruction/Data
00	0000000000000
01	0000000000000
02	0000000000000
03	0000000000000
04	0000000000000
05	0000000000000
06	0000000000000
07	0000000000000
08	0000000000000
09	0000000000000
10	0000000000000
11	0000000000000
12	0000000000000
13	0000000000000
14	0000000000000
15	0000000000000
16	0000100000000
17	0000100010010
18	0000100100100
19	0101010100101
20	0101010000001
21	0010110000100
22	1001010001010
23	0001010010000
24	0101010000001
25	1000000000101
26	1000000001010

Conversion from Logical Address to Physical Address for sample code 1.

All the preceding addresses are filled with Zeroes.

Sample Code 2: Summation of Series (using Variables)

Pseudo Code

`COUNT = 5`
`NUM = 6`
 	
`WHILE NUM > COUNT`
`      SUM = SUM + NUM`
`      NUM = NUM + 1`
 	
`FINISH (NO OPERATION LEFT)`

Translating pseudo code to assembly code

Sample Code 2: Summation of Series (using Variables)

Assembly Code (Part 1)

```
START 16      # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
              # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
              # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

XOR R0, R0 # RESET R0 WHICH WILL BE USED INITIALIZE NUM, SUM AND COUNT

# ALL THE LOCAL VARIABLES WILL BE SAVED FROM ADDRESS 6
STORE [6], R0     # Reset NUM ([6])
STORE [7], R0     # Reset SUM ([7])
STORE [8], R0     # Reset COUNT ([8])

LOAD R2, [8]     # R2 = COUNT ([8]) = 0 USING R2 AS ITS REPLACEMENT
ADD R2, 5        # COUNT (R2) = 5
STORE [8], R2    # COUNT ([8]) = 5

LOAD R0, [6]     # R0 = NUM ([6]) = 0 USING R0 AS ITS REPLACEMENT
ADD R0, 1        # NUM (R0) = 1
STORE [6], R0    # NUM ([6]) = 1

LOAD R1, [7]     # R1 = SUM ([7]) = 0 USING R1 AS ITS REPLACEMENT
```

Sample Code 2: Summation of Series (using Variables)

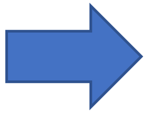
Assembly Code (Part 2)

```
STARTING:
    CMP R0, R2   # NUM (R0) > COUNT (R2)
    JG FINISH    # WHILE (NUM (R0) > COUNT (R2)), IF TRUE GOTO FINISH
    ADD R1, R0   # SUM (R1) = SUM (R1) + NUM (R0)
    ADD R0, 1    # NUM (R0) = NUM (R0) + 1
    JMP STARTING    # GOTO START OF WHILE LOOP
FINISH:

STORE [6], R0    # NUM ([6]) = 6
STORE [7], R1    # SUM ([7]) = 15

Exit:           # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit        # PUT CPU INTO NOP (NO OPERATION)
```

Sample Code 2: Summation of Series (using Variables)

Logical Address (Programmer's Perspective)

Logical Address	Instruction/Data
00	0000100000000
01	1100110000110
02	1100110000111
03	1100110001000
04	1100000101000
05	0101010100101
06	1100110101000
07	1100000000110
08	0101010000001
09	1100110000110
10	1100000010111
11	0010110000100
12	1001010100000
13	0001010010000
14	0101010000001
15	1000000011011
16	1100110000110
17	1100110010111
18	1000000100010
19	1000000100010



Physical Address (Stored in RAM/CPU's Perspective)

Physical Address	Instruction/Data
00	0000000000000
01	0000000000000
02	0000000000000
03	0000000000000
04	0000000000000
05	0000000000000
06	0000000000000
07	0000000000000
08	0000000000000
09	0000000000000
10	0000000000000
11	0000000000000
12	0000000000000
13	0000000000000
14	0000000000000
15	0000000000000

Physical Address	Instruction/Data
16	0000100000000
17	1100110000110
18	1100110000111
19	1100110001000
20	1100000101000
21	0101010100101
22	1100110101000
23	1100000000110
24	0101010000001
25	1100110000110
26	1100000010111
27	0010110000100
28	1001010100000
29	0001010010000
30	0101010000001
31	1000000011011
32	1100110000110
33	1100110010111
34	1000000100010
35	1000000100010

Conversion from Logical Address to Physical Address for sample code 2.
All the preceding addresses are filled with Zeroes.

Sample Code 2: Summation of Series (using Variables)

Physical Address (Stored in RAM/CPU's Perspective)

Physical Address	Instruction/Data	Memory Organization for Process
00	0000000000000	STACK MEMORY
01	0000000000000	
02	0000000000000	
03	0000000000000	
04	0000000000000	
05	0000000000000	
06	0000000000000	DATA MEMORY
07	0000000000000	
08	0000000000000	
09	0000000000000	
10	0000000000000	
11	0000000000000	
12	0000000000000	
13	0000000000000	
14	0000000000000	
15	0000000000000	ADDRESS MEMORY

Address 0-15 is reserved for process.

Physical Address	Instruction/Data
16	0000100000000
17	1100110000110
18	1100110000111
19	1100110001000
20	1100000101000
21	0101010100101
22	1100110101000
23	1100000000110
24	0101010000001
25	1100110000110
26	1100000010111
27	0010110000100
28	1001010100000
29	0001010010000
30	0101010000001
31	1000000011011
32	1100110000110
33	1100110010111
34	1000000100010
35	1000000100010

Sample Code 2: Summation of Series (using Variables)

Physical Address (Stored in RAM/CPU's Perspective)

Physical Address	Instruction/Data	Memory Organization for Process
00	0000000000000	STACK MEMORY
01	0000000000000	
02	0000000000000	
03	0000000000000	
04	0000000000000	
05	0000000000000	
06	0000000000000	NUM (DATA)
07	0000000000000	SUM (DATA)
08	0000000000000	COUNT (DATA)
09	0000000000000	
10	0000000000000	
11	0000000000000	
12	0000000000000	
13	0000000000000	
14	0000000000000	
15	0000000000000	ADDRESS MEMORY

Address 0-15 is reserved for process.

Physical Address	Instruction/Data
16	0000100000000
17	1100110000110
18	1100110000111
19	1100110001000
20	1100000101000
21	0101010100101
22	1100110101000
23	1100000000110
24	0101010000001
25	1100110000110
26	1100000010111
27	0010110000100
28	1001010100000
29	0001010010000
30	0101010000001
31	1000000011011
32	1100110000110
33	1100110010111
34	1000000100010
35	1000000100010

Sample Code 2: Summation of Series (using Variables)

Main Memory (RAM)

Physical Address	Instruction/Data	Memory Organization for Process
00	0000000000000	STACK MEMORY
01	0000000000000	
02	0000000000000	
03	0000000000000	
04	0000000000000	
05	0000000000000	
06	0000000000110	NUM (DATA)
07	0000000001111	SUM (DATA)
08	0000000000101	COUNT (DATA)
09	0000000000000	
10	0000000000000	
11	0000000000000	
12	0000000000000	
13	0000000000000	
14	0000000000000	
15	0000000000000	ADDRESS MEMORY

Register Set

Register No	Data
R0	0110
R1	1111
R2	0101
R3	0000
R4	0000
R5	0000
R6	0000
R7	0000

Figure: Data inside RAM and Registers after program execution.

Sample Code 3: Summation of Series (using Variables and Base Register)

Pseudo Code

```
COUNT = 5
NUM = 6

WHILE NUM > COUNT
     SUM = SUM + NUM
     NUM = NUM + 1

FINISH (NO OPERATION LEFT)
```

Translating pseudo code to assembly code

Sample Code 3: Summation of Series (using Variables and Base Register)

Assembly Code (Part 1)

```
START 16     # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
             # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
             # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

XOR R3, R3 # RESET BASE REGISTER (R3)
ADD R3, 6  # BASE REGISTER (R3) = 6
XOR R0, R0 # RESET R0 WHICH WILL BE USED INITIALIZE NUM, SUM AND COUNT

# ALL THE LOCAL VARIABLES WILL BE SAVED FROM ADDRESS 6
STORE [R3], R0      # Reset NUM ([6])
STORE [R3+1], R0    # Reset SUM ([7])
STORE [R3+2], R0    # Reset COUNT ([8])

LOAD R2, [R3+2]    # R2 = COUNT ([8]) = 0 USING R2 AS ITS REPLACEMENT
ADD R2, 5          # COUNT (R2) = 5
STORE [R3+2], R2   # COUNT ([8]) = 5

LOAD R0, [R3]    # R0 = NUM ([6]) = 0 USING R0 AS ITS REPLACEMENT
ADD R0, 1        # NUM (R0) = 1
STORE [R3], R0   # NUM ([6]) = 1

LOAD R1, [R3+1]    # R1 = SUM ([7]) = 0 USING R1 AS ITS REPLACEMENT
```

Sample Code 3: Summation of Series (using Variables and Base Register)

Assembly Code (Part 2)

```
STARTING:
    CMP R0, R2   # NUM (R0) > COUNT (R2)
    JG FINISH    # WHILE (NUM (R0) > COUNT (R2)), IF TRUE GOTO FINISH
    ADD R1, R0   # SUM (R1) = SUM (R1) + NUM (R0)
    ADD R0, 1    # NUM (R0) = NUM (R0) + 1
    JMP STARTING   # GOTO START OF WHILE LOOP
FINISH:

STORE [R3], R0      # NUM ([6]) = 6
STORE [R3+1], R1    # SUM ([7]) = 15

Exit:              # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit     # PUT CPU INTO NOP (NO OPERATION)
```

Hardware Limitations

For Direct Mode (`LOAD R0, [2]/STORE [3], R0`),

Opcode (6 bit)		Register 1	Address/Displacement
2 bits	4 bits	3 bits	4 bits
(11) Types of instruction	Operations	Ra(000-111)	Disp (0000-1111)

We can only directly access memory address from 0000(0) to 1111(15).

For `JMPREG` instruction,

Opcode (6 bit)		Register 1	Unused
2 bits	4 bits	3 bits	4 bits
Types of instruction	Operations	Ra (000-111)	XXXX

Since register can only save 4 bits of data, we can only directly access memory address from 0000(0) to 1111(15).

Sample Code 4: Function with no argument and no return value using Registers

Pseudo Code	

`main()`
`{`
`    add()`
`}`
	`add()`
	`{`
	`    R0 = 1`
	`    R1 = 2`
	`    R0 = R0+R1`
	`}`

Here, **`R0`** & **`R1`** are registers.

Translating pseudo code to assembly code

Sample Code 4: Function with no argument and no return value using Registers

Assembly Code

```
START 16     # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
             # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
             # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

Main: # MAIN FUNCTION WITH NO ARGUMENT
    JMP Add  # CALL Add FUNCTION
Return_From_Add:
    JMP Exit

Add: # ADD FUNCTION WITH NO ARGUMENT
    XOR R0, R0   # RESET (R0)
    XOR R1, R1   # RESET (R1)
    XOR R2, R2   # RESET (R2)

    ADD R0, 1    # R0 = 1
    ADD R1, 2    # R1 = 2
    ADD R0, R1   # R0 = R0 + R1
    JMP Return_From_Add # RETURN WITHOUT RETURN VALUE

Exit:             # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit      # PUT CPU INTO NOP (NO OPERATION)
```

STACK Memory

In order to send arguments to the called function, arguments are saved into STACK memory. Return values are also saved into STACK memory. Data inside Local variables/registers are also saved into STACK before calling function because that functions will also use local variables and registers.

STACK memory is called STACK memory because it works like a STACK/Last-In-First-Out (LIFO). It uses PUSH and POP operations to store and retrieve arguments. **STACK Pointer (SP) register** keep track of location of last data inside STACK.

1. **PUSH (X):** Push X value to STACK memory. It is implemented as follows:

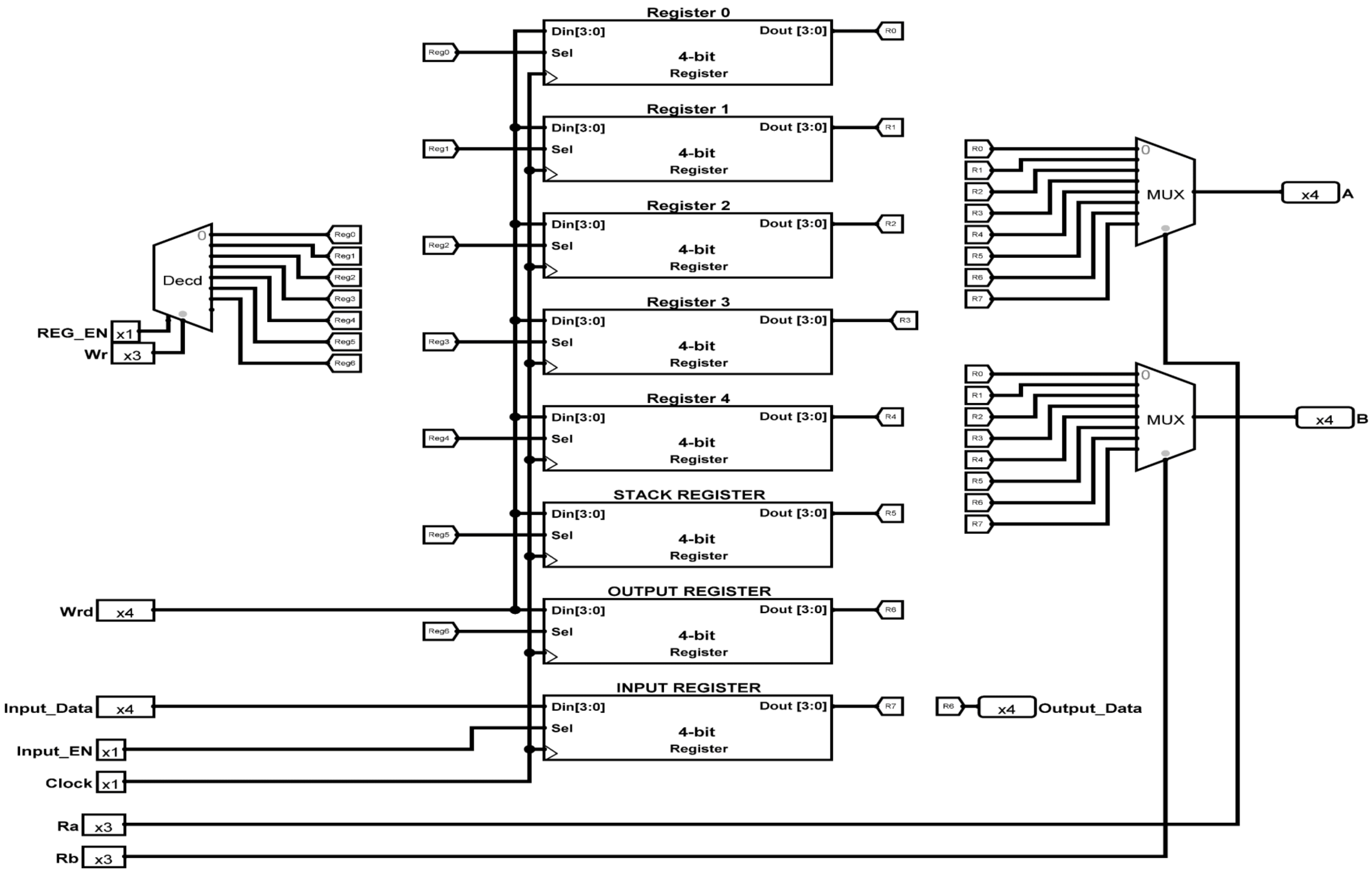
```
STORE [SP], R0        # PUSH ARGUMENT (STORED IN R0) TO STACK
ADD SP, 1             # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION
```

2. **POP (X):** Pop/Retrieve last stored data into X from STACK memory. It is implemented as follows:

```
LOAD R0, [SP]         # POP ARGUMENT (STORED IN R0)FROM STACK
SUB SP, 1             # SET STACK ADDRESS TO PREVIOUS MEMORY/STACK LOCATION
```

STACK Register

STACK Register: STACK Register (SP) keeps track of address of last data inside STACK.



Sample Code 5: Function with arguments and return values using Registers

Pseudo Code	

`main()`
`{`
`    D = 5`
`    E = add(1,2)`
`    D = D +1`
`}`
	`add(A,B)`
	`{`
	`  C = A+B`
	`  return C`
	`}`

Translating pseudo code to assembly code.

We will also optimize this assembly code without changing meaning of program.

Sample Code 5: Function with arguments and return values using Registers

Pseudo Assembly Code

```
START 16     # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
             # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
             # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

# ALL THE ARGUMENTS AND RETURN VALUES ARE STORED IN STACK
# WHICH STARTS FROM ADDRESS 0
XOR SP, SP # RESET STACK POINTER (SP) WHICH IS ORIGINALLY R5

Main: # MAIN FUNCTION WITH NO ARGUMENT
    # ALL THE LOCAL VARIABLES WILL BE SAVED FROM ADDRESS 6
    XOR R3, R3        # RESET (R3)
    ADD R3, 5         # R3 = 5
    STORE [6], R3     # D = [6] = R3 = 2

    XOR R0, R0   # RESET (R0)
    XOR R1, R1   # RESET (R1)

    ADD R0, 1   # R0 = 1 => IT SAVES FIRST ARGUMENT OF VALUE 1
    ADD R1, 2   # R1 = 2 => IT SAVES SECOND ARGUMENT OF VALUE 2
```

Sample Code 5: Function with arguments and return values using Registers

Pseudo Assembly Code

```
 # ARGUMENTS ARE ALWAYS PUSHED IN REVERSE ORDER INTO STACK
    PUSH(R3)   # PUSH D TO STACK

    PUSH(R1)   # PUSH 2 (SECOND ARGUMENT) TO STACK

    PUSH(R0)   # PUSH 1 (FIRST ARGUMENT) TO STACK

    JMP Add

Return_From_Add:
    POP(R2)    # POP RETURN VALUE FROM STACK
    STORE [7], R2    # SAVE E

    POP(R3)    # POP D FROM STACK
```

Sample Code 5: Function with arguments and return values using Registers

Pseudo Assembly Code

```
    ADD R3, 1          # D = D + 1
    STORE [6], R3      # SAVE D

    JMP Exit

Add: # ADD FUNCTION WITH A & B ARGUMENTS
    # SINCE ADD FUNCTION USES A, B AND C WHICH ARE LOCAL VARIABLES
    # OMITING THEM AND USING ONLY REGISTERS ARE ENOUGH
    # BECAUSE IT WILL NOT CHANGE MEANING OF PROGRAM

    POP(R0)      # POP A (R0) FROM STACK

    POP(R1)      # POP B (R1)FROM STACK

    ADD R0, R1         # A = A + B (R0 = R0 + R1)
```

Sample Code 5: Function with arguments and return values using Registers

Pseudo Assembly Code

`PUSH(R0)   # PUSH RETURN VALUE TO STACK`
 	
`JMP Return_From_Add   # RETURN WITH RETURN VALUE`
 	
`Exit:   # PUT CPU INTO NOP (NO OPERATION)`
`JMP Exit   # PUT CPU INTO NOP (NO OPERATION)`

Sample Code 5: Function with arguments and return values using Registers

Assembly Code

```
START 16     # PROGRAM WILL BE LOADED AT ADDRESS 16 IN RAM
             # IT WILL FILL ALL PREVIOUS ADDRESSES (0-15) WITH 0 AND
             # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (16)

# ALL THE ARGUMENTS AND RETURN VALUES ARE STORED IN STACK
# WHICH STARTS FROM ADDRESS 0
XOR SP, SP # RESET STACK POINTER (SP) WHICH IS ORIGINALLY R5

Main: # MAIN FUNCTION WITH NO ARGUMENT
    # ALL THE LOCAL VARIABLES WILL BE SAVED FROM ADDRESS 6
    XOR R3, R3        # RESET (R3)
    ADD R3, 5         # R3 = 5
    STORE [6], R3     # D = [6] = R3 = 2

    XOR R0, R0   # RESET (R0)
    XOR R1, R1   # RESET (R1)

    ADD R0, 1   # R0 = 1 => IT SAVES FIRST ARGUMENT OF VALUE 1
    ADD R1, 2   # R1 = 2 => IT SAVES SECOND ARGUMENT OF VALUE 2
```

Sample Code 5: Function with arguments and return values using Registers

Assembly Code

```
# ARGUMENTS ARE ALWAYS PUSHED IN REVERSE ORDER INTO STACK
    STORE [SP], R3   # PUSH D TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    STORE [SP], R1   # PUSH 2 (SECOND ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    STORE [SP], R0   # PUSH 1 (FIRST ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP Add

Return_From_Add:
    SUB SP, 1        # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R2, [SP]    # POP RETURN VALUE FROM STACK
    STORE [7], R2    # SAVE E

    SUB SP, 1        # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R3, [SP]    # POP D FROM STACK
```

Sample Code 5: Function with arguments and return values using Registers

Assembly Code

```
    ADD R3, 1          # D = D + 1
    STORE [6], R3      # SAVE D

    JMP Exit

Add: # ADD FUNCTION WITH A & B ARGUMENTS
    # SINCE ADD FUNCTION USES A, B AND C WHICH ARE LOCAL VARIABLES
    # OMITING THEM AND USING ONLY REGISTERS ARE ENOUGH
    # BECAUSE IT WILL NOT CHANGE MEANING OF PROGRAM

    SUB SP, 1          # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R0, [SP]      # POP A (R0) FROM STACK

    SUB SP, 1          # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R1, [SP]      # POP B (R1)FROM STACK

    ADD R0, R1         # A = A + B (R0 = R0 + R1)
```

Sample Code 5: Function with arguments and return values using Registers

Assembly Code

<pre>   STORE [SP], R0   # PUSH RETURN VALUE TO STACK
    ADD SP, 1   # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION
 	
    JMP Return_From_Add   # RETURN WITH RETURN VALUE
 	
Exit: # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit   # PUT CPU INTO NOP (NO OPERATION)</pre>

Sample Code 6: Calling Same Functions Twice

<table>
<tr><th colspan="2">Pseudo Code</th></tr>
<tr>
<td><pre>main()
{
    add(1,2)
    add(3,4)
}</pre></td>
<td><pre>add(A,B)
{
   C = A+B
   return C
}</pre></td>
</tr>
</table>

Translating pseudo code to assembly code.

Here, we cannot use JMP instruction directly to return from called function because it will not know from which it is being called. So, we have to save return address in STACK memory before calling function.

Sample Code 6: Calling Same Functions Twice

Assembly Code

```
START 13      # PROGRAM WILL BE LOADED AT ADDRESS 13 IN RAM
              # IT WILL FILL ALL PREVIOUS ADDRESSES (0-12) WITH 0 AND
              # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (13)

# ADDRESS TABLE
#==============================================================================
JMP Main                                                # ADDRESS 13 -> MAIN FUNCTION
JMP Return_From_Add1                                    # ADDRESS 14
JMP Return_From_Add2                                    # ADDRESS 15
#==============================================================================

Main: # MAIN FUNCTION WITH NO ARGUMENT
    # ALL THE ARGUMENTS AND RETURN VALUES ARE STORED IN STACK
    # WHICH STARTS FROM ADDRESS 0
    XOR SP, SP # RESET STACK POINTER (SP) WHICH IS ORIGINALLY R5
```

Sample Code 6: Calling Same Functions Twice

Assembly Code

```
#=======================  add function 1 ========================================
    XOR R0, R0   # RESET (R0)
    XOR R1, R1   # RESET (R1)
    XOR R2, R2   # RESET (R2)

    ADD R0, 1    # R0 = 1 => IT SAVES FIRST ARGUMENT OF VALUE 1
    ADD R1, 2    # R1 = 2 => IT SAVES SECOND ARGUMENT OF VALUE 2
    ADD R2, 14   # R2 = 14 => IT SAVES RETURN ADDRESS 14 WHICH WILL JUMP TO Return_From_Add1

    # ARGUMENTS ARE ALWAYS PUSHED IN REVERSE ORDER INTO STACK
    STORE [SP], R1   # PUSH 2 (SECOND ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    STORE [SP], R0   # PUSH 1 (FIRST ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    STORE [SP], R2   # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP Add
```

Sample Code 6: Calling Same Functions Twice

Assembly Code

```
Return_From_Add1:
    SUB SP, 1        # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R3, [SP]    # POP RETURN VALUE FROM STACK

#=======================  add function 2 ========================================
    XOR R0, R0   # RESET (R0)
    XOR R1, R1   # RESET (R1)
    XOR R2, R2   # RESET (R2)

    ADD R0, 3    # R0 = 3 => IT SAVES FIRST ARGUMENT OF VALUE 3
    ADD R1, 4    # R1 = 4 => IT SAVES SECOND ARGUMENT OF VALUE 4
    ADD R2, 15   # R2 = 15 => IT SAVES RETURN ADDRESS 15 WHICH WILL JUMP TO Return_From_Add2

    # ARGUMENTS ARE ALWAYS PUSHED IN REVERSE ORDER INTO STACK
    STORE [SP], R1   # PUSH 4 (SECOND ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    STORE [SP], R0   # PUSH 3 (FIRST ARGUMENT) TO STACK
    ADD SP, 1        # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION
```

Sample Code 6: Calling Same Functions Twice

Assembly Code

```
    STORE [SP], R2    # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1         # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP Add

Return_From_Add2:
    SUB SP, 1         # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R3, [SP]     # POP RETURN VALUE FROM STACK

    JMP Exit

Add: # ADD FUNCTION WITH A & B ARGUMENTS
    # SINCE ADD FUNCTION USES A, B AND C WHICH ARE LOCAL VARIABLES
    # OMITING THEM AND USING ONLY REGISTERS ARE ENOUGH
    # BECAUSE IT WILL NOT CHANGE MEANING OF PROGRAM

    SUB SP, 1         # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R2, [SP]     # POP RETURN_ADDRESS FROM STACK

    SUB SP, 1         # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R0, [SP]     # POP A (R0) FROM STACK
```

Sample Code 6: Calling Same Functions Twice

Assembly Code

```
    SUB SP, 1           # SET STACK ADDRESS TO CURRENT MEMORY/STACK LOCATION
    LOAD R1, [SP]       # POP B (R1)FROM STACK

    ADD R0, R1          # A = A + B (R0 = R0 + R1)

    STORE [SP], R0      # PUSH RETURN VALUE TO STACK
    ADD SP, 1           # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMPREG R2           # RETURN TO CALLED FUNCTION

Exit:                   # PUT CPU INTO NOP (NO OPERATION)
    JMP Exit            # PUT CPU INTO NOP (NO OPERATION)IS ORIGINALLY R5
```

Interrupts

An interrupt is a response by the processor to an event that needs attention from the software.

An interrupt condition alerts the processor and serves as a request for the processor to interrupt the currently executing code when permitted, so that the event can be processed in a timely manner.

If the request is accepted, the processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event.

This interruption is temporary, and, unless the interrupt indicates a fatal error, the processor resumes normal activities after the interrupt handler finishes.

There are two types of Interrupts:

1. Software Interrupts
2. Hardware Interrupts

Hardware Interrupts

Hardware Interrupt is caused by some hardware device such as request to start an I/O, a hardware failure or something similar. Hardware interrupts were introduced as a way to avoid wasting the processor's valuable time in polling loops, waiting for external events.

For example, when an I/O operation is completed such as reading some data into the computer from a tape drive.

Software Interrupts

Software Interrupt is invoked by the use of INT instruction. This event immediately stops execution of the program and passes execution over to the INT handler. The INT handler is usually a part of the operating system and determines the action to be taken. It occurs when an application program terminates or requests certain services from the operating system.

For example, output to the screen, execute file etc.

Software Input Interrupt Function in Assembly

```
HARDWARE_INTERRUPT_INPUT:         # HARDWARE INTERRUPT FOR INPUT WHICH WILL BE CALLED BY HARDWARE
                                  # ITSELF TO LET SOFTWARE KNOW THAT USER HAS ENTERED THE INPUT
    WAITING_FOR_INPUT:
        ACCEPT_INPUT
        JMP WAITING_FOR_INPUT         # WAITING FOR INPUT

    HARDWARE_INTERRUPT_INPUT_FOUND: # HARDWARE SAYS THAT INPUT IS FOUND
        JMP RETURN_FROM_HARDWARE_INTERRUPT_INPUT
```

Hardware Input Interrupt Function in Assembly

```
SOFTWARE_INTERRUPT_INPUT:         # SOFTWARE INTERRUPT FOR INPUT
                                  # IT WILL SAVE INPUT TO STACK
    JMP HARDWARE_INTERRUPT_INPUT

    RETURN_FROM_HARDWARE_INTERRUPT_INPUT:
        XOR R0, R0          # RESET (R0)
        ADD R0, INR         # SAVE INPUT DATA (INPUT REGISTER) TO R0

        SUB SP, 1           # SET STACK TO CURRENT STACK LOCATION
        LOAD R3, [SP]       # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3           # RETURN TO CALLED FUNCTION
```

Software Output Interrupt Function in Assembly

```
SOFTWARE_INTERRUPT_OUTPUT_PRINT:
    JMP HARDWARE_INTERRUPT_OUTPUT_PRINT
    RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT:

        SUB SP, 1        # SET STACK TO PREVIOUS STACK LOCATION
        LOAD R3, [SP]    # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3        # RETURN TO CALLED FUNCTION
```

Hardware Output Interrupt Function in Assembly

```
HARDWARE_INTERRUPT_OUTPUT_PRINT:
    XOR OUTR, OUTR
    ADD OUTR, R0
    PRINT_OUTPUT
    JMP RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT
```

Sample Code 7: Program takes INPUT from Keyboard and print it to TTY Display

Assembly Code

```
START 12      # PROGRAM WILL BE LOADED AT ADDRESS 12 IN RAM
              # IT WILL FILL ALL PREVIOUS ADDRESSES (0-11) WITH 0 AND
              # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (12)

# ADDRESS TABLE
#================================================================================
JMP MAIN                                  # ADDRESS 12 -> MAIN FUNCTION
JMP RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1 # ADDRESS 13
JMP RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT1 # ADDRESS 14
#================================================================================
JMP HARDWARE_INTERRUPT_INPUT_FOUND        # ADDRESS 15 -> FIXED ADDRESS
                                          # FOR HARDWARE INPUT INTERRUPT
#================================================================================

SOFTWARE_INTERRUPT_OUTPUT_PRINT:
    JMP HARDWARE_INTERRUPT_OUTPUT_PRINT
    RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT:

        SUB SP, 1       # SET STACK TO PREVIOUS STACK LOCATION
        LOAD R3, [SP]   # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3       # RETURN TO CALLED FUNCTION
```

Sample Code 7: Program takes INPUT from Keyboard and print it to TTY Display

Assembly Code

```
HARDWARE_INTERRUPT_OUTPUT_PRINT:
    XOR OUTR, OUTR
    ADD OUTR, R0
    PRINT_OUTPUT
    JMP RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT

SOFTWARE_INTERRUPT_INPUT:        # SOFTWARE INTERRUPT FOR INPUT
                                 # IT WILL SAVE INPUT TO STACK
    JMP HARDWARE_INTERRUPT_INPUT

    RETURN_FROM_HARDWARE_INTERRUPT_INPUT:
        XOR R0, R0       # RESET (R0)
        ADD R0, INR      # SAVE INPUT DATA (INPUT REGISTER) TO R0

        SUB SP, 1        # SET STACK TO CURRENT STACK LOCATION
        LOAD R3, [SP]    # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3        # RETURN TO CALLED FUNCTION
```

Sample Code 7: Program takes INPUT from Keyboard and print it to TTY Display

Assembly Code

```
HARDWARE_INTERRUPT_INPUT:   # HARDWARE INTERRUPT FOR INPUT WHICH WILL BE CALLED BY HARDWARE
                            # ITSELF TO LET SOFTWARE KNOW THAT USER HAS ENTERED THE INPUT
    WAITING_FOR_INPUT:
        ACCEPT_INPUT
        JMP WAITING_FOR_INPUT       # WAITING FOR INPUT

    HARDWARE_INTERRUPT_INPUT_FOUND: # HARDWARE SAYS THAT INPUT IS FOUND
        JMP RETURN_FROM_HARDWARE_INTERRUPT_INPUT

MAIN: # MAIN FUNCTION WITH NO ARGUMENT
    XOR SP, SP # RESET STACK POINTER (SP) WHICH IS ORIGINALLY R5

    XOR R0, R0              # RESET (R0)
    ADD R0, 13              # R0 = 13 => IT SAVES RETURN ADDRESS 13 WHICH
                            # WILL JUMP TO RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1

    STORE [SP], R0          # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1               # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP SOFTWARE_INTERRUPT_INPUT # WAITING FOR USER INPUT. RETURN VALUE WILL BE SAVED IN R0
```

Sample Code 7: Program takes INPUT from Keyboard and print it to TTY Display

Assembly Code

```
RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1:

    XOR R1, R1              # RESET (R1)
    ADD R1, 14              # R1 = 14 => IT SAVES RETURN ADDRESS 14 WHICH
                            # WILL JUMP TO RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT1

    STORE [SP], R1          # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1               # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP SOFTWARE_INTERRUPT_OUTPUT_PRINT     # OUTPUT DATA WILL BE IN R0
RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT1:
    JMP EXIT

EXIT:                   # PUT CPU INTO NOP (NO OPERATION)
    JMP EXIT            # PUT CPU INTO NOP (NO OPERATION)
```

Sample Code 8: Program takes two INPUTS from Keyboard and print their addition to TTY Display

Assembly Code

```
START 11      # PROGRAM WILL BE LOADED AT ADDRESS 11 IN RAM
              # IT WILL FILL ALL PREVIOUS ADDRESSES (0-10) WITH 0 AND
              # PHYSICAL ADDRESS = LOGICAL ADDRESS + RELOCATION VALUE (11)

# ADDRESS TABLE
#================================================================================
JMP MAIN                                            # ADDRESS 11 -> MAIN FUNCTION
JMP RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1           # ADDRESS 12
JMP RETURN_FROM_SOFTWARE_INTERRUPT_INPUT2           # ADDRESS 13
JMP RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT     # ADDRESS 14
#================================================================================
JMP HARDWARE_INTERRUPT_INPUT_FOUND         # ADDRESS 15 -> FIXED ADDRESS FOR
                                           # HARDWARE INPUT INTERRUPT
#================================================================================
SOFTWARE_INTERRUPT_OUTPUT_PRINT:
    JMP HARDWARE_INTERRUPT_OUTPUT_PRINT
    RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT:

        SUB SP, 1       # SET STACK TO PREVIOUS STACK LOCATION
        LOAD R3, [SP]   # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3       # RETURN TO CALLED FUNCTION
```

Sample Code 8: Program takes two INPUTS from Keyboard and print their addition to TTY Display

Assembly Code

```
HARDWARE_INTERRUPT_OUTPUT_PRINT:
    XOR OUTR, OUTR
    ADD OUTR, R0
    PRINT_OUTPUT
    JMP RETURN_FROM_HARDWARE_INTERRUPT_OUTPUT_PRINT

SOFTWARE_INTERRUPT_INPUT:        # SOFTWARE INTERRUPT FOR INPUT
                                 # IT WILL SAVE INPUT TO STACK
    JMP HARDWARE_INTERRUPT_INPUT

    RETURN_FROM_HARDWARE_INTERRUPT_INPUT:
        XOR R0, R0        # RESET (R0)
        ADD R0, INR       # SAVE INPUT DATA (INPUT REGISTER) TO R0

        SUB SP, 1         # SET STACK TO CURRENT STACK LOCATION
        LOAD R3, [SP]     # POP RETURN ADDRESS TO R3 FROM STACK

        JMPREG R3         # RETURN TO CALLED FUNCTION
```

Sample Code 8: Program takes two INPUTS from Keyboard and print their addition to TTY Display

Assembly Code

```
HARDWARE_INTERRUPT_INPUT:   # HARDWARE INTERRUPT FOR INPUT WHICH WILL BE CALLED BY HARDWARE
                            # ITSELF TO LET SOFTWARE KNOW THAT USER HAS ENTERED THE INPUT
    WAITING_FOR_INPUT:
        ACCEPT_INPUT
        JMP WAITING_FOR_INPUT       # WAITING FOR INPUT

    HARDWARE_INTERRUPT_INPUT_FOUND: # HARDWARE SAYS THAT INPUT IS FOUND
        JMP RETURN_FROM_HARDWARE_INTERRUPT_INPUT

MAIN: # MAIN FUNCTION WITH NO ARGUMENT
    XOR SP, SP # RESET STACK POINTER (SP) WHICH IS ORIGINALLY R5

    XOR R0, R0              # RESET (R0)
    ADD R0, 12              # R0 = 12 => IT SAVES RETURN ADDRESS 12 WHICH
                            # WILL JUMP TO RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1

    STORE [SP], R0          # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1               # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION
```

Sample Code 8: Program takes two INPUTS from Keyboard and print their addition to TTY Display

Assembly Code

```
    JMP SOFTWARE_INTERRUPT_INPUT # WAITING FOR USER INPUT. RETURN VALUE WILL BE SAVED IN R0
RETURN_FROM_SOFTWARE_INTERRUPT_INPUT1:
    XOR R1, R1              # RESET (R1)
    ADD R1, R0              # SAVE FIRST INPUT DATA (R0) TO R1

    XOR R0, R0              # RESET (R0)
    ADD R0, 13              # R0 = 13 => IT SAVES RETURN ADDRESS 13 WHICH
                            # WILL JUMP TO RETURN_FROM_SOFTWARE_INTERRUPT_INPUT2

    STORE [SP], R0          # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1               # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP SOFTWARE_INTERRUPT_INPUT # WAITING FOR USER INPUT. RETURN VALUE WILL BE SAVED IN R0
RETURN_FROM_SOFTWARE_INTERRUPT_INPUT2:

    ADD R0, R1              # R0 = R0 + R1

    XOR R2, R2              # RESET (R2)
    ADD R2, 14              # R2 = 14 => IT SAVES RETURN ADDRESS 14 WHICH
                            # WILL JUMP TO RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT
```

Sample Code 8: Program takes two INPUTS from Keyboard and print their addition to TTY Display

Assembly Code

```
    STORE [SP], R2          # PUSH RETURN ADDRESS TO STACK
    ADD SP, 1               # SET STACK ADDRESS TO NEXT MEMORY/STACK LOCATION

    JMP SOFTWARE_INTERRUPT_OUTPUT_PRINT      # OUTPUT DATA WILL BE IN R0
RETURN_FROM_SOFTWARE_INTERRUPT_OUTPUT_PRINT:
    JMP EXIT

EXIT:           # PUT CPU INTO NOP (NO OPERATION)
    JMP EXIT        # PUT CPU INTO NOP (NO OPERATION)
```

Thank you ☺