Lecture 3

Assembly Language

Previously ..

- Number representation for computing
- Memory location and Address
  - Bit Address, Address Space
  - Big & Little Endian
- Instruction Sequencing
  - Assembly Notation: ADD R1, R2
- Instruction Sets
  - CISC vs RISC
- Branch Concept
  - Branch Code (Flag)
- Addressing Mode
  - Immediate, Direct, Indirect mode …

Assembly Language (1)

- Assembly language is used to represent machine instructions in a way that human can understand.
- Words such as move, load, and branch are used as commands. These keywords are normally replaced by mnemonics in assembly language

- Ex. move $\longrightarrow$ Mov

  branch $\longrightarrow$ Br

Assembly Language (2)

- Program written in Assembly language can be translated into a sequence of machine language (op-code)
- In some Assembly languages, a different mnemonics will be used for different addressing mode
- Ex ADD #5,R3 Ξ ADDI 5,R3

Assembly Language (3)

- Assembly language allows different ways to specify numerical values

Decimal	ADD #93,R1
Binary	ADD #%01011101,R1
Hexadecimal	ADD #$5D,R1

Statements in Assembly Language

Label: Operation   Operand(s)   Comment

- 4 fields need one or more blank or tab characters
- Label is associated with the memory address where the instruction will be loaded or the addresses of data.
- Operation contains mnemonic of instruction or assembly directive เป็นโน้ตให้ assembly สั่งให้ทำงาน ทุกบรรทัดที่เขียนจะแปลงเป็น instruction เลย เช่น add
- Operand contains the addressing information

Assembly Directive (1)

- This statement does not an instruction
- For an assembler to produce an object code, it must know the following:
  1. How to interpret the name (sum = 200)
  2. Where to place the instructions in the memory
  3. Where to place the data operands in the memory

Assembly Directive (2)

	Memory Address Label	Operation	Addressing Or data information
Assembler directive	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that	START	MOVE	N,R1
generate		MOVE	#NUM1,R2
machine		CLR	R0
instructions	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Equal (*EQU*) Directive

- If we need to assign a numerical value to any name, we can use the *EQU* directive.

  EX SUM EQU 200 (assign 200 to SUM)

- The assembler uses the value when translating the source program to the object program.

Origin Directive

- The *Origin* directive tells the assembler where, in the memory, to place the data block that follows.
- From the previous example, memory location 204 is to be loaded with the next data block.

Dataword Directive

- The *Dataword directive* states that the data value 100, is to be placed in the memory (at address 204).
- Statement that involves placing data/instruction into a memory may be given a label (*N* in this Example)
- *N* = 204 (address location).
- We can say that the *Dataword* statement has the label *N.*

Reserve Directive

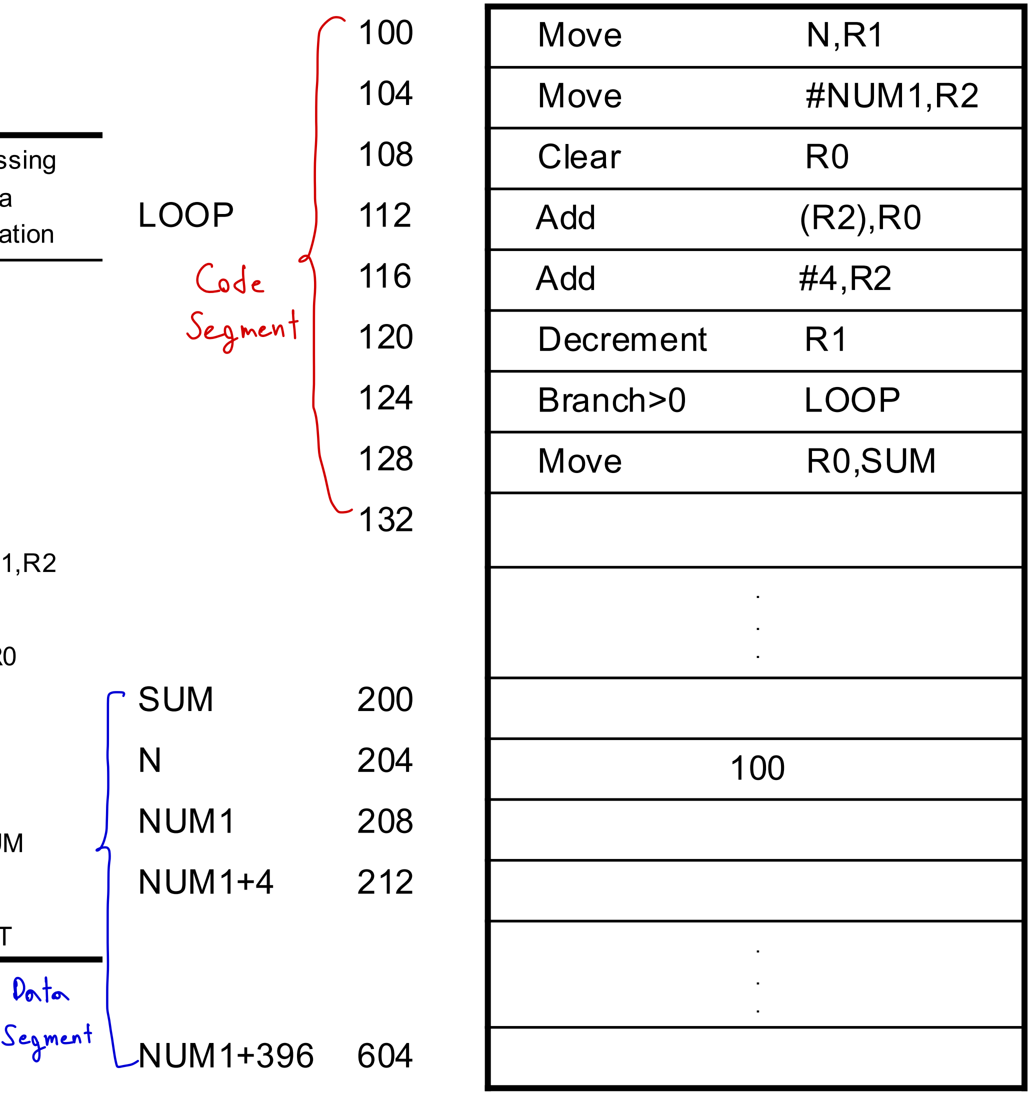
- The *Reserve* directive declares that a memory block of 400 bytes, is to be reserved for data.
- From example
  - ‘NUM1 RESERVE 400’ means that the next 400 bytes of memory are reserved for data and the first byte is labeled with *NUM1*
- Thus, NUM1 = 208

Return and *End* Directives

- The *Return* directive indicates where the execution of the program should be terminated, here.
- The *End* directive tells the assembler where is the end of the source program.

Again, Assembly Directive

	Memory Address Label	Operation	Addressing Or data information
Assembler directive	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START



Assembler

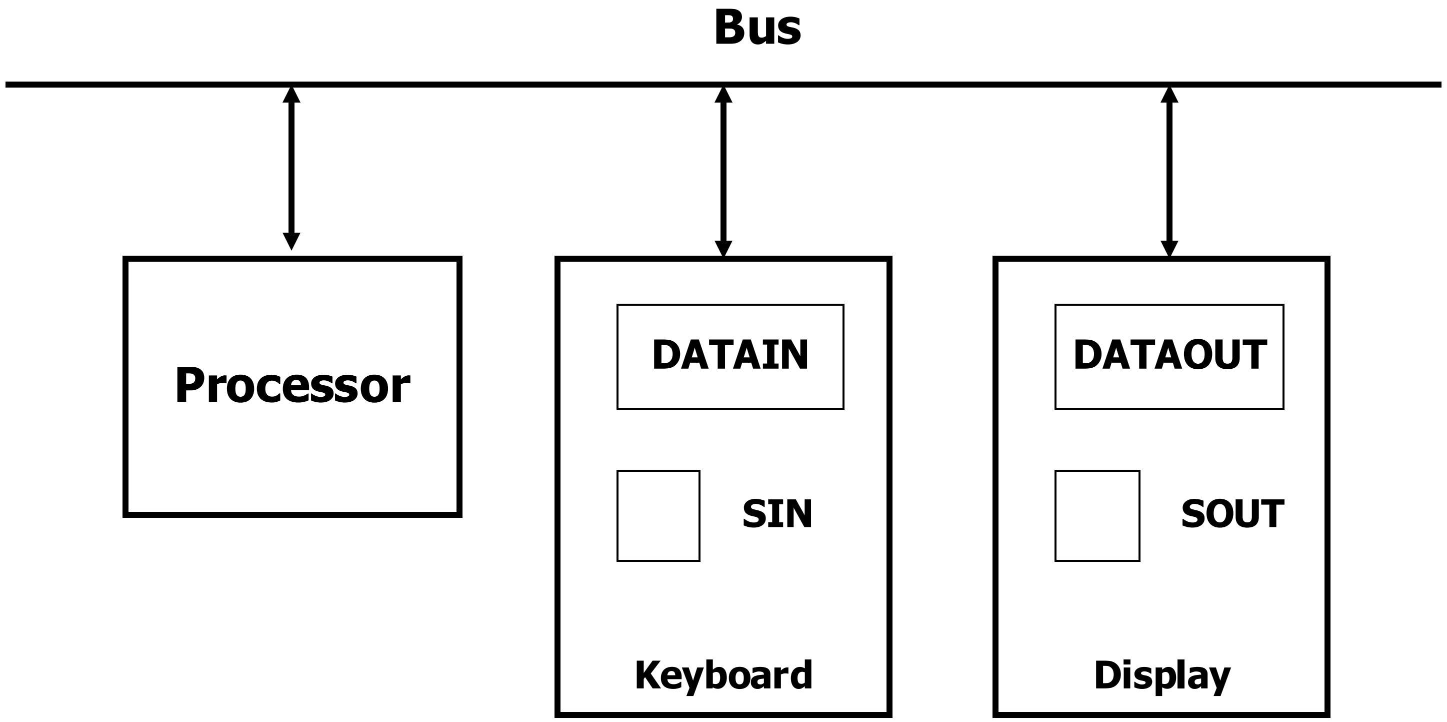
- The assembler is a program that replaces all symbols in the source code with the binary codes (used at a machine level).
- The assembler also replaces names and labels with the actual values.
- The assembler scans through a source program, keep tracks of all names and the numerical values in a symbol-table. When names appear, they are replaced with values from this table.

Map label ว่าตรงกับ address ที่ทำไหม

I/O operations (1)

- Speed of any operator to enter characters via a keyboard, is a few character/sec.
- Speed of characters to be transferred to a display device, are several thousands character/sec.
- Speed of a CPU to process the characters, is in the scale of many millions character/sec.
- A major difference in speed, requires a mechanism to synchronize the data transfer.

I/O operations (2)



I/O operations: DATAIN

- 'DATAIN' is a register that stores an input from a keyboard. 'SIN' is a control flag that informs a processor that a valid character is in DATAIN.
- When SIN = 1, a processor reads from DATAIN. SIN is cleared (SIN = 0), when the reading is finished.
- Ex. Assembly code

```
READWAIT     Branch to READWAIT if SIN = 0
           input from DATAIN to R1
```

Loop
เรื่อยๆ จนกว่า
false

I/O operations: DATAOUT

- 'DATAOUT' and 'SOUT' are used in the same manner (for display).
- When SOUT = 1, a processor fills DATAOUT, and then it clears SOUT. When a display device is finished, SOUT is set again.
- Ex. Assembly code

```
WRITEWAIT     Branch to WRITEWAIT if SOUT = 0
              output from R1 to DATAOUT
```

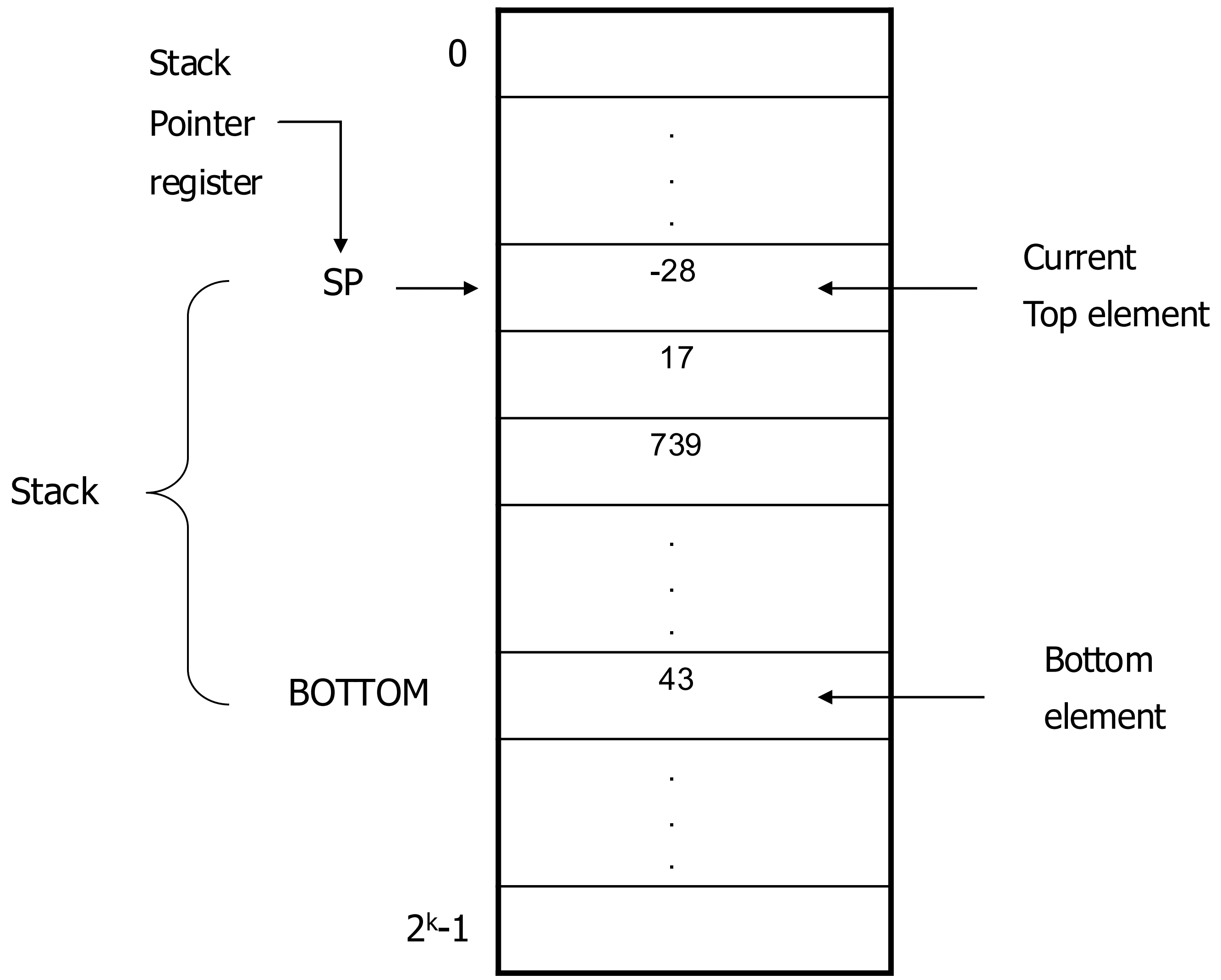
I/O operations: Example

```
          Move        #LOC, R0        Initialize pointer register R0 to point to
                                      the address  of first location in memory
                                      where the characters are to be stored.
READ      TestBit     #3,INSTATUS     Wait for a character to be entered
          Branch=0    READ            in the keyboard buffer DATAIN .
          Move        DATAIN,(R0)     Transfer the character from DATAIN into
                                      the memory (this clears SIN to 0).
ECHO      TestBit     #3,OUTSTATUS    Wait for the display to become ready.
          Branch=0    ECHO
          Move        (R0),DATAOUT    Move the character just read to the
                                      display buffer resister (this clears SOUT
                                      to 0).
          Compare     #CR,(R0)+       Check if the character just read is CR
          Branch ≠ 0  READ            (carriage return). If  it is not CR, then
                                      branch back and read another character.
                                      Also, increment  the pointer to store the
                                      next character.
```

Stacks (1)

- Stack is a list of data elements that can be added or removed at one end only (top). It is LIFO (Last in first out)
- Stack is usually used to handle control between a main program and subroutines

Stacks (2)


Stack
Pointer
register
SP
0
-28
17
739
43
2^k-1
Current
Top element
Stack
BOTTOM
Bottom
element

PUSH and *POP* (1)

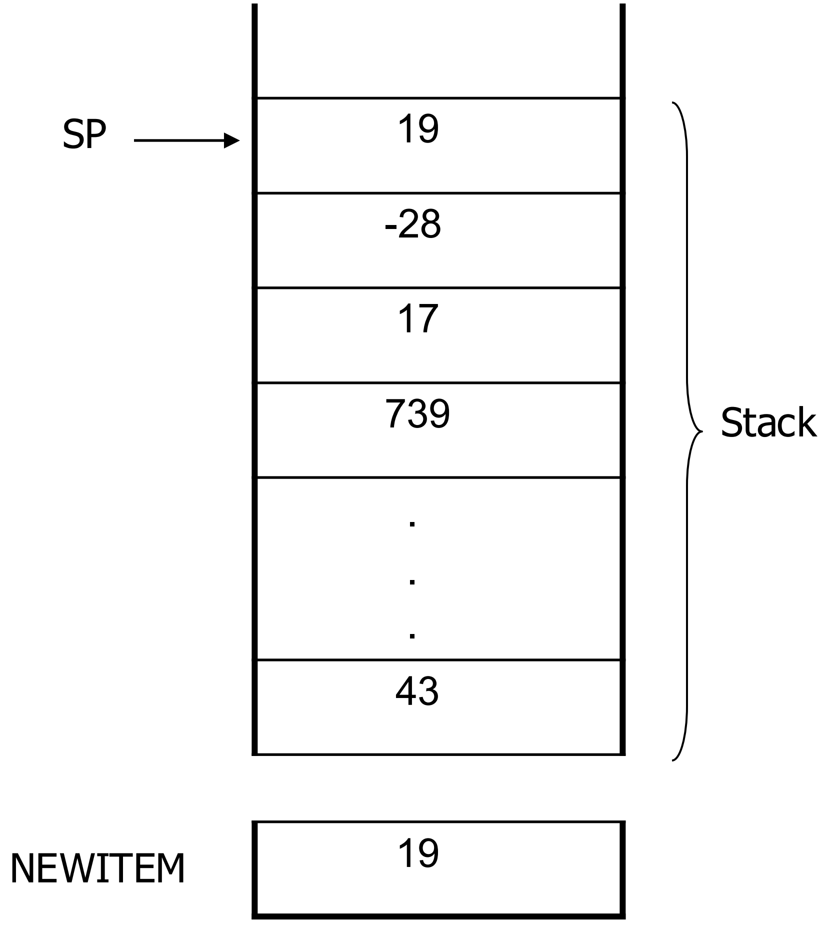
```
PUSH    SUB     #4,SP
        Move    NEWITEM,(SP)

POP     Move    (SP), ITEM
        ADD     #4,SP
```

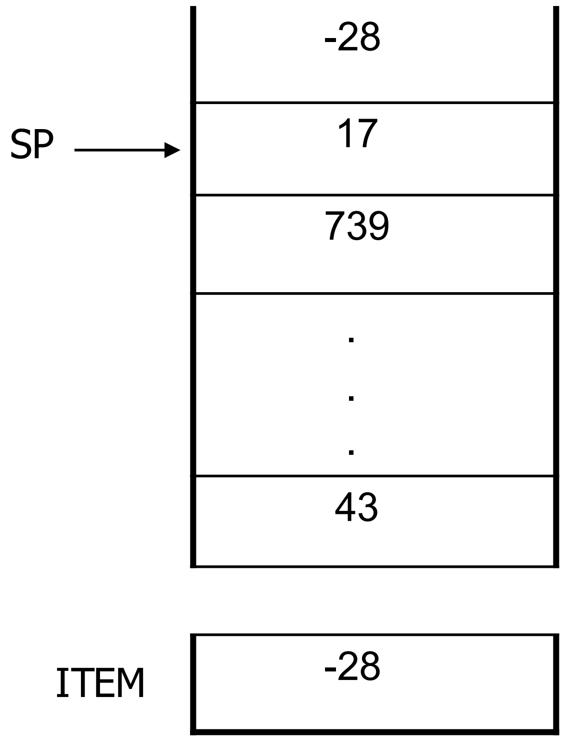
PUSH and *POP* (2)

```
PUSH    SUB     #4,SP
        Move    NEWITEM,(SP)

POP     Move    (SP), ITEM
        ADD     #4,SP
```



(a) After push from NEWITEM



(b) After pop into ITEM

PUSH and *POP* (3)

- For Auto decrement & Auto increment
  - Push $\longrightarrow$ Move NEWITEM, -(SP)
  - Pop $\longrightarrow$ Move (SP)+, ITEM
- Stack is usually allocated before the program run with a fixed amount of space

PUSH and *POP* (4)

SAFEPOP	Compare Branch>0 Move	#2000,SP EMPTYERROR (SP)+,ITEM	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action. Otherwise , pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch ≤ 0 Move	#1500,SP FULLERROR NEWITEM,-(SP)	Check to see if the stack pointer contains an address value equal to or less than 1500.If it does, the stack is full. Branch to the routine FULLERROR for appropriate action. Otherwise , push the element in memory location NEWITEM onto the stack.

(b) Routine for a safe push operation

Queues

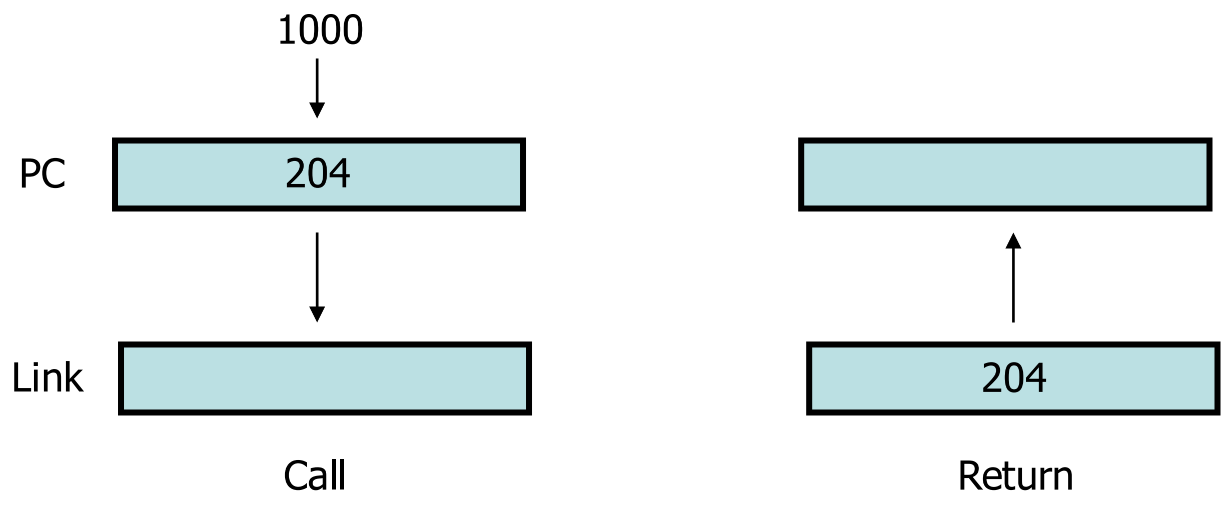
- A queue is similar to a stack, except for the fact that it is FIFO (First in first out) and it uses a circular buffer with 2 pointers (head & tail).
- A circular buffer is used to fix the location of a queue in a memory.

Subroutines (1)

- Only one copy of subroutine is store in a memory to save space. Any program that want to use a subroutine simply branches to its starting location.
- When a subroutine returns, the assembler has to make sure that it returns to the appropriate location. The simplest way is to use a link register to save the content of the PC when a subroutine is called.

Subroutines (2)

Memory Location	Calling Program	Memory Location	Subroutine SUB
	. . .		
200	Call SUB	1000	first instruction
204	next instruction		. . .
	. . .		Return



Subroutines (3)

- The link register cannot support the nested subroutine calls (a subroutine calling another subroutine).
- Using a stack concept, nested subroutine calls can be carried out at any depth.
  - Call instruction : pushes the content of the PC to the stack and loads the beginning of subroutine address to the PC.
  - Return instruction : pops the return address from a stack and copy it to a PC.

Parameter Passing (1)

1. It can be done by using registers. The called program can place a set of parameters in a set of registers before calling a subroutine. The subroutine then uses those values and returns the result via another register.

Parameter Passing (2)

Calling program

```
            Move        N,R1        R1 serves as a counter.
            Move        #NUM,R2     R2 points to the list.
            Call        LISTADD     Call subroutine.
            Move        R0,SUM      Save result.
```

SMT, Not Enough Regs

Subroutine

```
LISTADD     Clear       R0          Initialize sum to 0.
LOOP        Add         (R2)+,R0    Add entry from list.
            Decrement   R1
            Branch > 0  LOOP
            Return                  Return to calling program.
```

Parameter Passing (3)

2. It can also be done by placing parameters in the stack together with the return address.

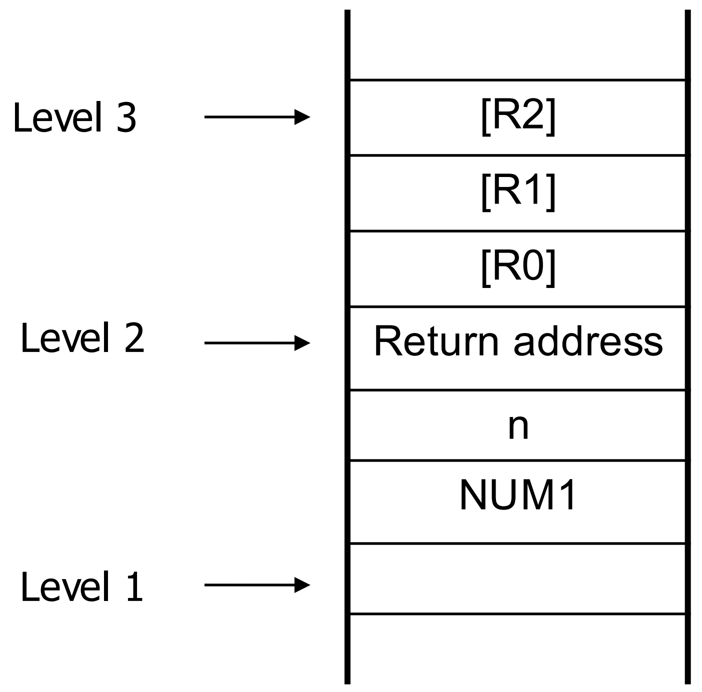
Parameter Passing (4)

Assume top of stack is at level 1 below.

```
            Move          #NUM1,-(SP)    Push parameters onto stack.
            Move          N,-(SP)
            Call          LISTADD        Call subroutine
                                         (top of stack at level 2).
            Move          4(SP),SUM      Save result.
            Add           #8,SP          Restore top of stack
                                         (top of stack at level 1).
            .
            .
            .
LISTADD     MoveMultiple  R0-R2,-(SP)    Save registers
                                         (top of stack at level 3).
            Move          16(SP),R1      Initialize counter to n.
            Move          20(SP),R2      Initialize pointer to the list .
            Clear         R0             Initialize sum to 0.
LOOP        Add           (R2)+,R0       Add entry from list.
            Decrement     R1
            Branch > 0    LOOP
            Move          R0,20(SP)      Put result on the stack.
            MoveMultiple  (SP)+,R0-R2    Restore registers.
            Return                       Return to calling program.
```

(a) Calling program and subroutine

Parameter Passing (5)



(b) Top of stack at various times

Study More on ..

- Additional instructions
  - Logic instructions
  - Shift and rotate instructions
  - Multiplication and division instructions
- Dealing a 32-bit value with two 16-bit registers