S J P N Trust's

**Hirasugar Institute of Technology, Nidasoshi.**

Inculcating Values, Promoting Prosperity

Approved by AICTE and Affiliated to VTU Belgaum.

# Computer Organization

Prof. Ravindra R. Patil

Assistant Professor, CSE

25-Aug-18

## 2.6.1 Assembler Directives

| | | | |
|---|---|---|---|
| | 100 | Move | N,R1 |
| | 104 | Move | #NUM1,R2 |
| | 108 | Clear | R0 |
| LOOP | 112 | Add | (R2),R0 |
| | 116 | Add | #4,R2 |
| | 120 | Decrement | R1 |
| | 124 | Branch>0 | LOOP |
| | 128 | Move | R0,SUM |
| | 132 | | |
| | | ⋮ | |
| SUM | 200 | | |
| N | 204 | 100 | |
| NUM1 | 208 | | |
| NUM2 | 212 | | |
| | | ⋮ | |
| NUM$n$ | 604 | | |

**Figure 2.17** Memory arrangement for the program in Figure 2.12.

If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

| | Memory address label | Operation | Addressing or data information |
|---|---|---|---|
| Assembler directives | 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 |

**Figure 2.18** Assembly language representation for the program in Figure 2.17.

25-Aug-18

# Basic Input / Output Operations

- The data on which the instructions operate are not necessarily already stored in memory.
- Data need to be transferred between processor and outside world (disk, keyboard, etc.)
- I/O operations are essential, the way they are performed can have a significant effect on the performance of the computer.

# Program-Controlled I/O

- Read in character input from a keyboard and produce character output on a display screen.
  - Rate of data transfer (keyboard, display, processor)
  - Difference in speed between processor and I/O device creates the need for mechanisms to synchronize the transfer of data.
  - A solution: on output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character. Input is sent from the keyboard in a similar way

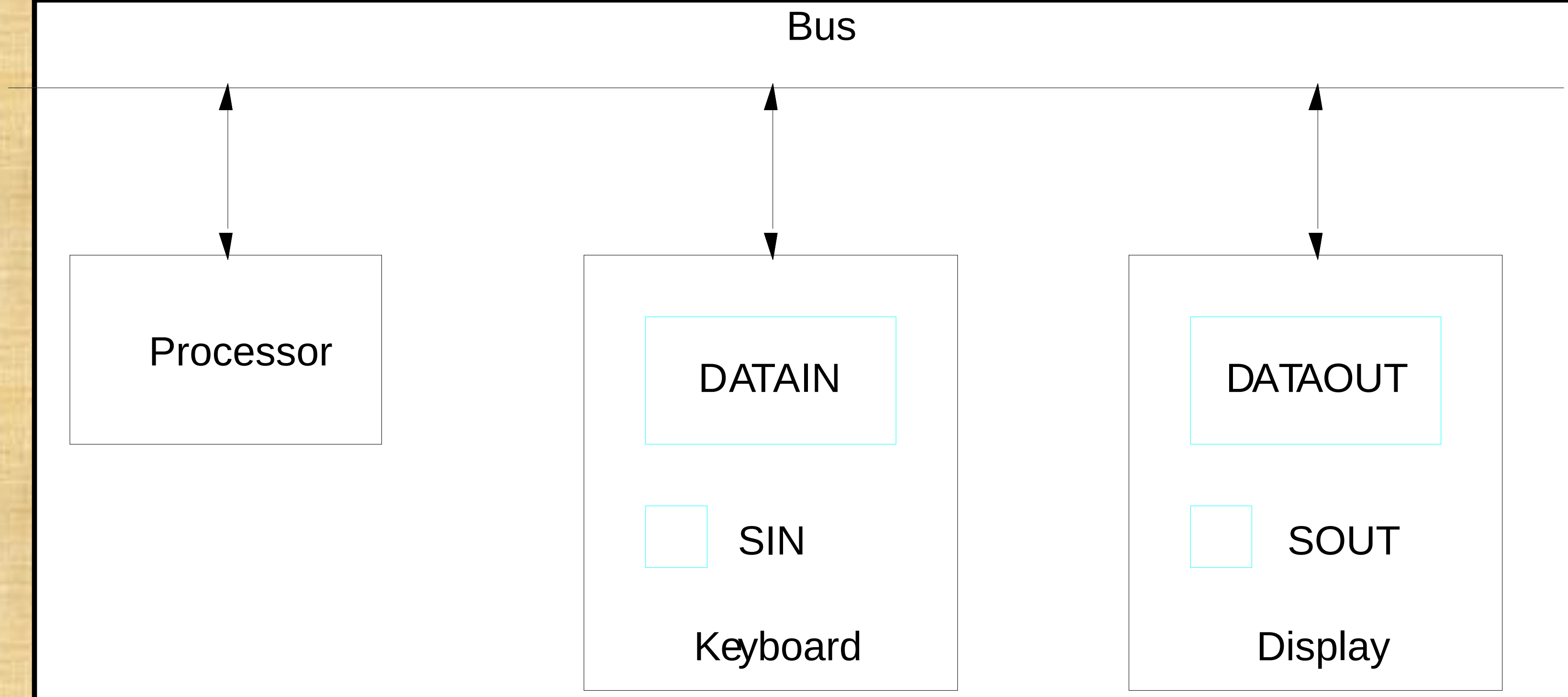



Figure 2.19 Bus connection for processor , keyboard, and display

# Program-Controlled I/O Example

Machine instructions that can check the state of the status flags and transfer data:

*READWAIT* Branch to *READWAIT* if SIN = 0
Input from DATAIN to R1

*WRITEWAIT* Branch to *WRITEWAIT* if SOUT = 0
Output from R1 to DATAOUT

# Memory-Mapped I/O

- Some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers.

```
READWAIT  Testbit   #3, INSTATUS
          Branch=0  READWAIT
          MoveByte  DATAIN, R1
```

Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.

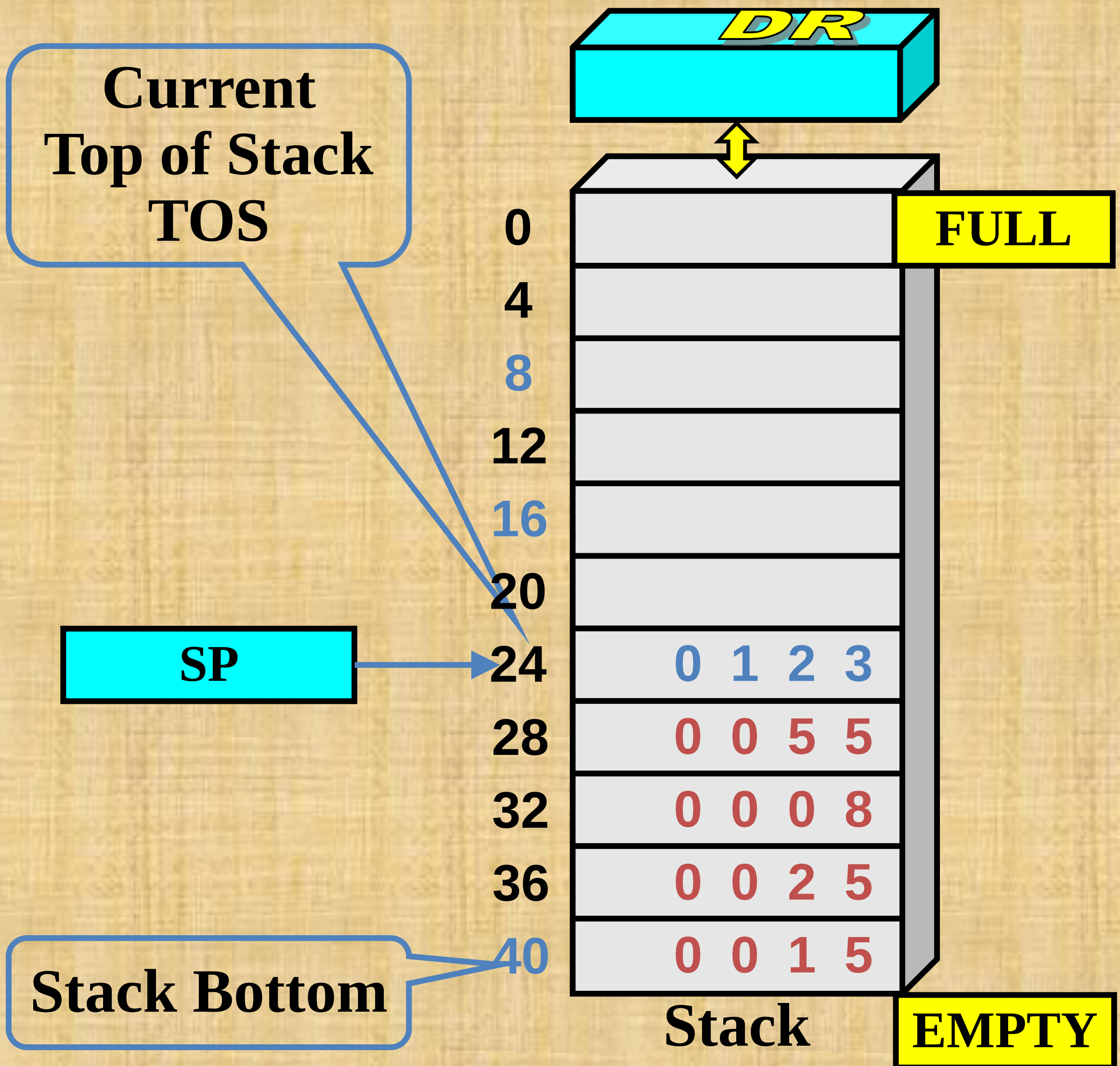
## Program that reads a line of characters & displays it unit carriage return key to be pressed.

```
      Move #Loc,R0
READ  TestBit #3,INSTATUS
      Branch=0 READ
      MoveByte DATAIN,(R0)
ECHO  TestBit #3,OUTSTATUS
      Branch=0 ECHO
      MoveByte (R0),DATAOUT
      Compare #CR,(R0)+
      Branch!=0 READ
```

# Stack and Queues

- LIFO

  *Last In First Out*



# Stack and Queues

**Push operation can be implemented as:**

Substract #4,SP

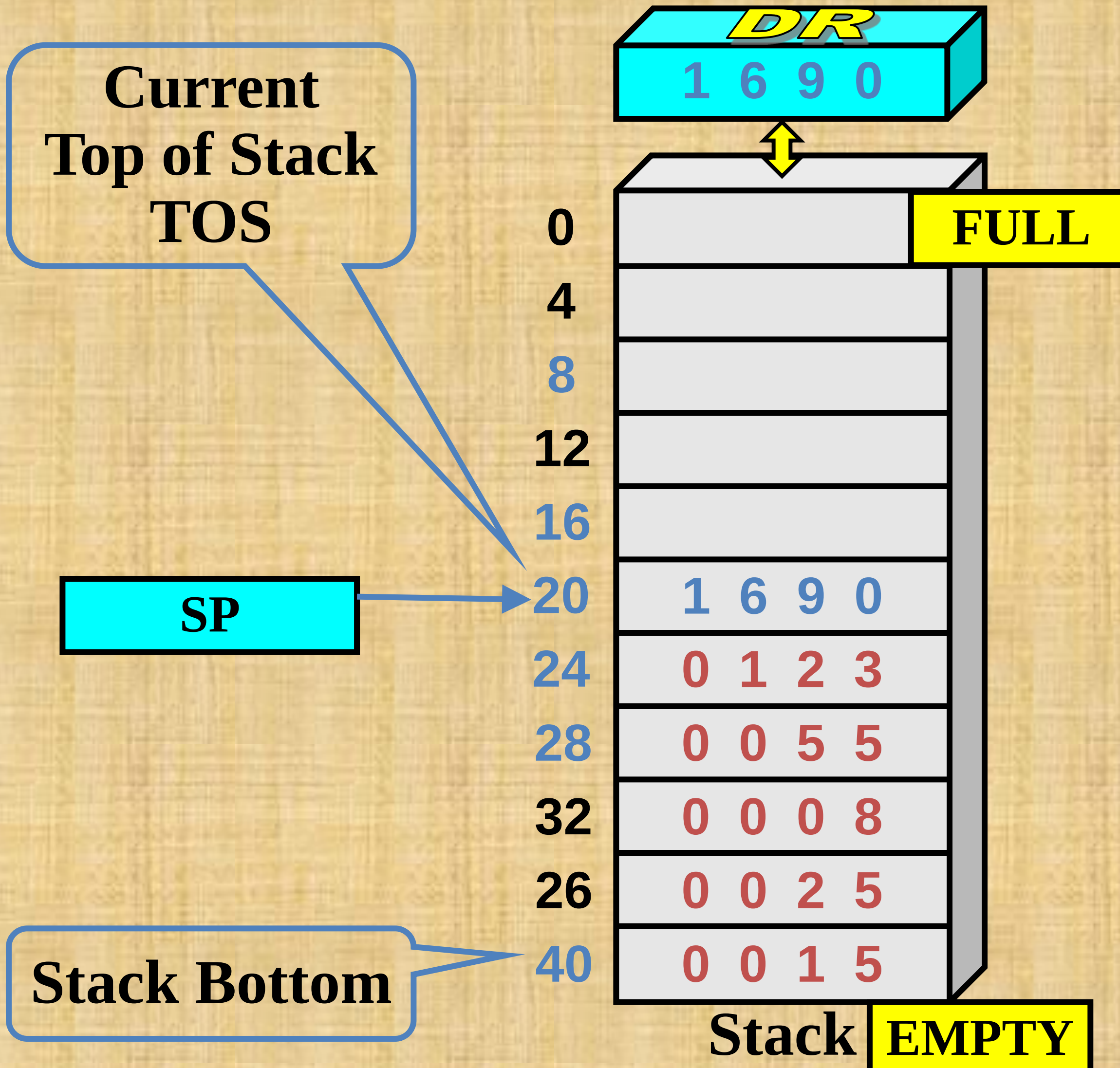
Move NEWITEM,(SP)

**Pop operation can be Implemented as:**

Move (SP),ITEM

Add #4,SP

# Stack after Push of one element



# PUSH and POP USING AUTOINCREMENT AND AUTODECREMENT

PUSH can implemented using auto decrement:

Move NEWITEM,-(SP)

POP can implemented using auto increment:

Move (SP)+,ITEM

# STACK SIZE WITH address 2000 to 1500 PUSH AND POP

```
SAP   Compare #2000,SP
      Branch>0   EMTERROR       Routine for POP
      Move (SP)+,ITEM

SAH   Compare #1500,SP
      Branch<=0 FULLERROR       Routine for PUSH
      Move NEWITEM,-(SP)
```

# Subroutines

- Perform a particular subtask many times on different data values:- subroutine .
- To save the space in memory, only one copy of such instructions are stored in memory.
- Any program that requires this subroutine will simply branch to staring location of this.
- After subroutine execution, the calling program must resume execution, after the calling instruction in the called program.

# Subroutines

- The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage*.
- The call instruction is special type of branch instruction:
  - Store the content of PC in the link Register
  - Branch to the target address specified by the instruction.

# Subroutines

- The Return Instruction is a special branch instruction:
  - Branch to the address contained in the link register.

# Subroutines

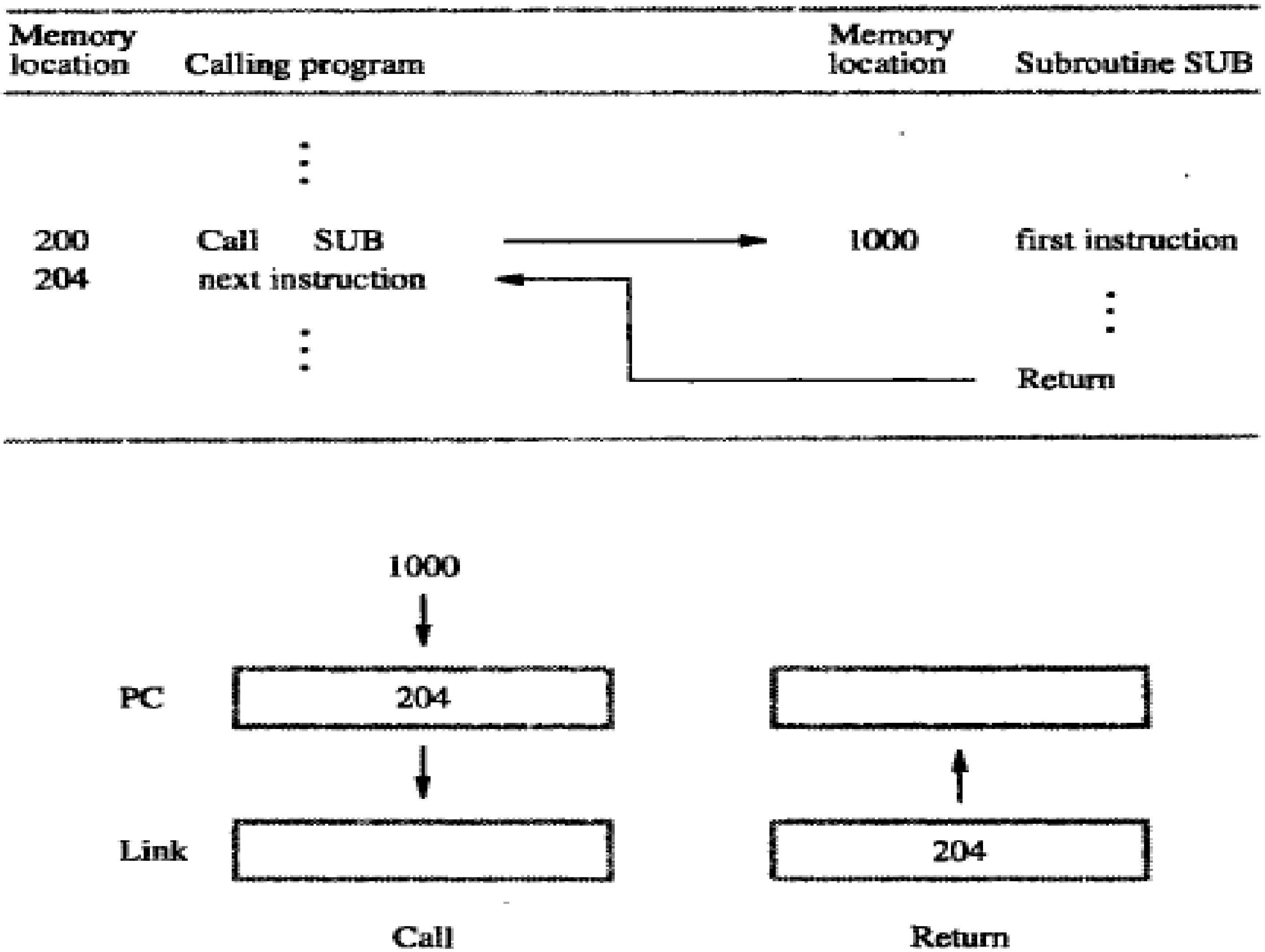



**Figure 2.24** Subroutine linkage using a link register.

# Parameter Passing

- When calling a subroutine, a program must provide to the subroutine the parameters, the operands or addresses, to be used in the computation.
- The exchange of information between a calling program and a subroutine is referred as *parameter passing.*
- There are two ways :
    - Placed in registers
    - Placed in memory locations: *processor stack*

# Parameter Passing

- Passing parameter through processor registers is straightforward and efficient.

Example:

Calling program

```
Move N,R1
Move #NUM1,R2
Move R0,SUM
.
.
```

Subroutine

```
LISTADD   Clear R0
LOOP      Add (R2)+,R0
          Decrement R1
          Branch>0  LOOP
          Return
```

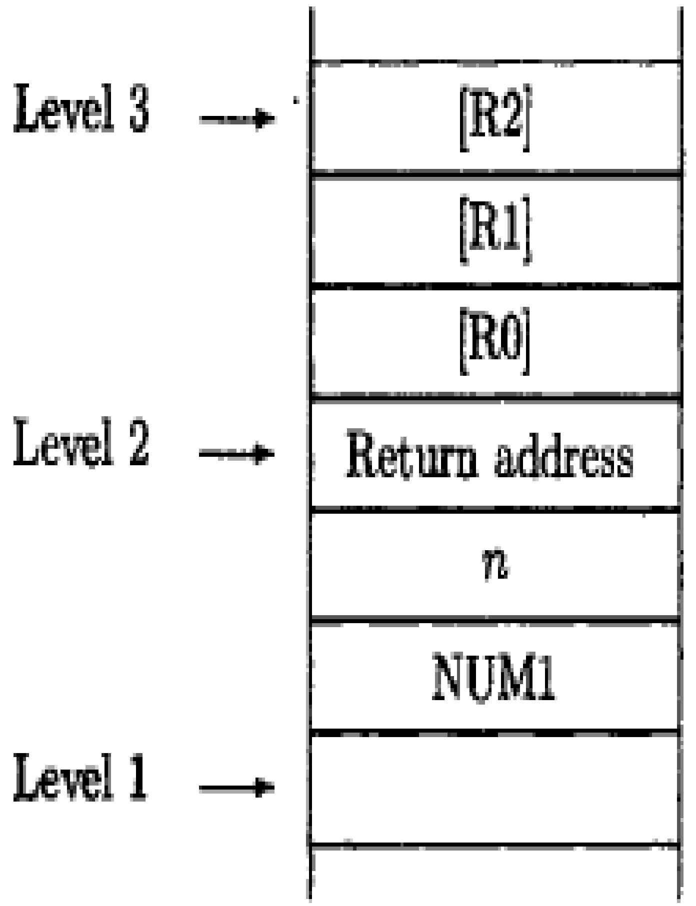
# Parameter Passing

- Passing parameter as address.

```
Move #Num1.-(SP)
Move N,-(SP)
Call LISTADD
Move 4(SP),SUM
Add #8,SP
```

```
LISTADD  MoveMultiple R0-R2,-(SP)
         Move 16(SP),R1
         Move 20(SP),R2
         Clear R0
LOOP     Add (R2)+,R0
         Decrement R1
         Branch>0 LOOP
         Move R0,20(SP)
         MoveMultiple (SP)+,R0-R2
         Return
```

# Parameter Passing




(b) Top of stack at various times

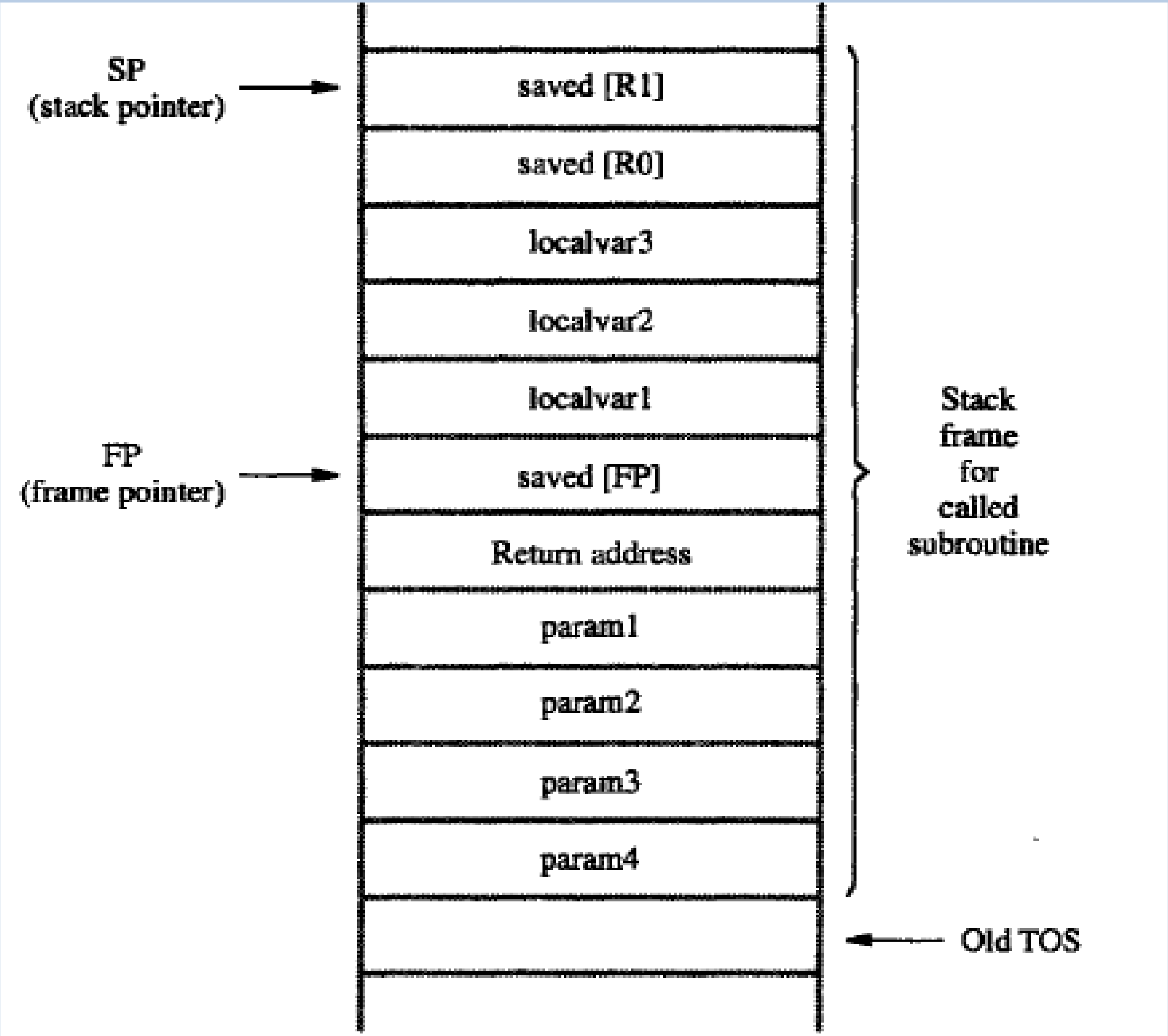
# The STACK Frame

- The *stack frame*, also known as *activation record* is the collection of all data on the stack associated with one subprogram call.

The stack frame generally includes the following components:

- The return address.
- Argument variables passed on the stack.
- Local variables.
- Saved copies of any registers modified by the subprogram that need to be restored.

# The STACK Frame



# The STACK Frame

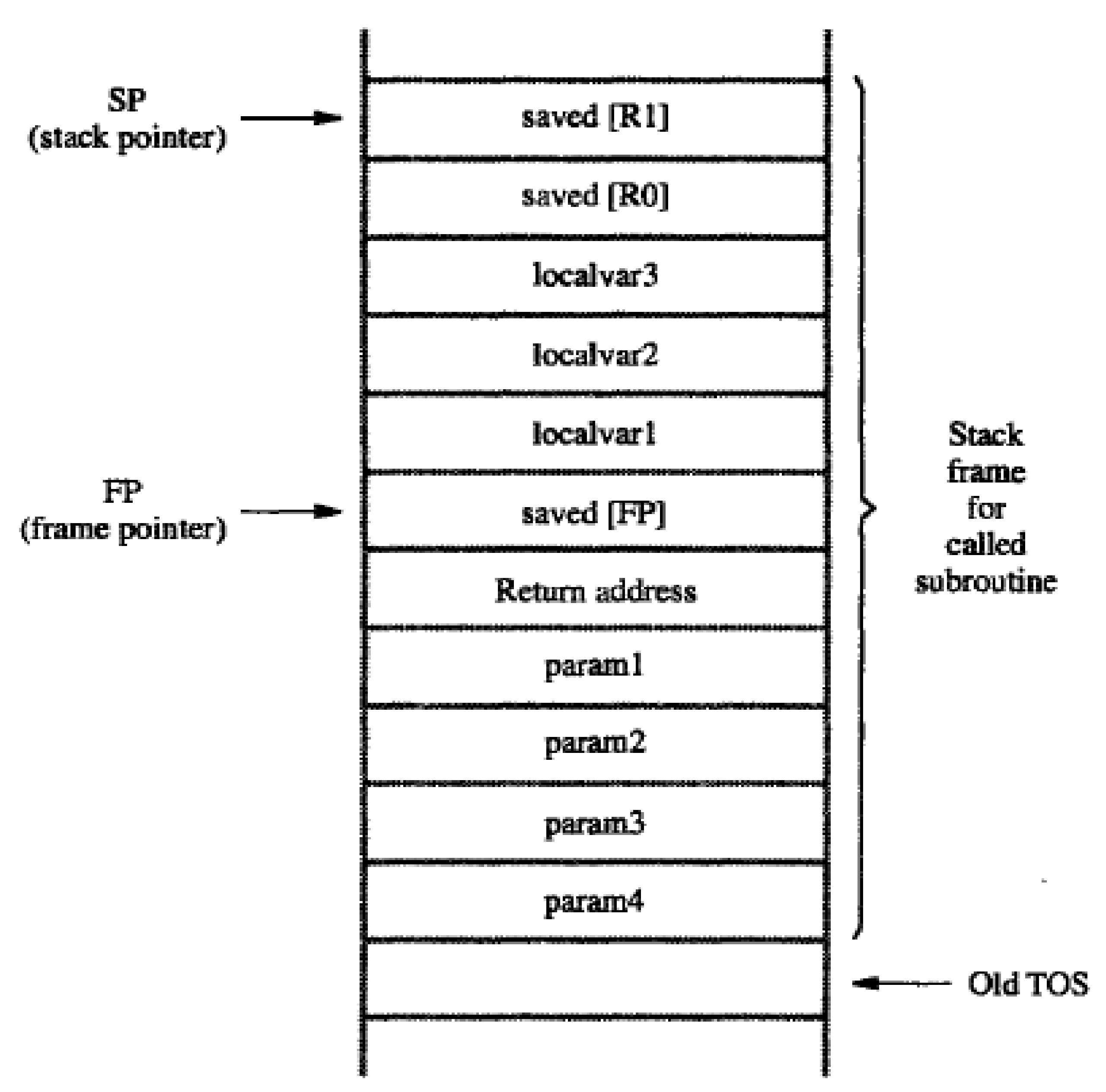
- The stack pointer will change when a subprogram does a push or pop operation.
- When this happens, the offset addresses representing local automatic variables such as 4(SP) are no longer valid.
- One way to alleviate this problem is by using the *frame pointer*.

# The STACK Frame

## The Frame Pointer(FP):

- The frame pointer is another register that we set to the address of the stack frame when a subprogram begins executing.
- If the code refers to local variables as offsets from the frame pointer instead of offsets from the stack pointer, then the program can use the stack pointer without complicating access to auto variables.
- We would then refer to something in the stack frame as offset(FP) instead of offset(SP)

# Subroutine STACK Frame Example



# Subroutine STACK Frame Example

- The first 2 instructions executed in subroutine are:

Move FP,-(SP)

Move SP,FP

- Space for the 3 Local variables is now allocated on the stack by executing:

Subtract #12,SP

- The contents of R0 & R1 are pushed on to the Stack
- After the completion of Task by Subroutine it brings SP back to FP value.

Add #12,SP

# Subroutine STACK Frame Example

- The calling program is responsible for removing the parameters from the stack frame.
- The stack pointer now pointes to the old TOS, and we are back to where we have started.

# STACK Frame for Nested Subroutine

- The stack is the proper data structure for holding return addresses when subroutines are called.
- Stack frames for nested subroutines build up on the processor stack as they are called.
- Note that the saved contents of FP in the current frame at the top of the stack are the frame pointer contents for the stack frame of the subroutine that called the current subroutine.

# Nested Subroutine Example

| Memory location | | Instructions |
| --- | --- | --- |
| **Main program** | | |
| | ⋮ | |
| 2000 | Move | PARAM2,−(SP) |
| 2004 | Move | PARAM1,−(SP) |
| 2008 | Call | SUB1 |
| 2012 | Move | (SP),RESULT |
| 2016 | Add | #8,SP |
| 2020 | next instruction | |

**First subroutine**

| | | | |
| --- | --- | --- | --- |
| 2100 | SUB1 | Move | FP,−(SP) |
| 2104 | | Move | SP,FP |
| 2108 | | MoveMultiple | R0−R3,−(SP) |
| 2112 | | Move | 8(FP),R0 |
| | | Move | 12(FP),R1 |
| | | ⋮ | |
| | | Move | PARAM3,−(SP) |
| 2160 | | Call | SUB2 |
| 2164 | | Move | (SP)+,R2 |
| | | ⋮ | |
| | | Move | R3,8(FP) |
| | | MoveMultiple | (SP)+,R0−R3 |
| | | Move | (SP)+,FP |
| | | Return | |

## Second subroutine

```
3000  SUB2  Move          FP,-(SP)
            Move          SP,FP
            MoveMultiple  R0-R1,-(SP)
            Move          8(FP),R0
            ⋮
            Move          R1,8(FP)
            MoveMultiple  (SP)+,R0-R1
            Move          (SP)+,FP
            Return
```