Stacks

- A stack is a list of data elements where elements are added/removed at top end only
- Also known as pushdown stack or last-in-first-out (LIFO) stack
- We push a new element on the stack top or pop the top element from the stack
- Programmer can create a stack in the memory
- There is often a special processor stack as well

Processor Stack

- Processor has stack pointer (SP) register that points to top of the processor stack
- Push operation involves two instructions:

```
Subtract    SP, SP, #4
Store       Rj, (SP)
```

- Pop operation also involves two instructions:

```
Load        Rj, (SP)
Add         SP, SP, #4
```

Subroutines

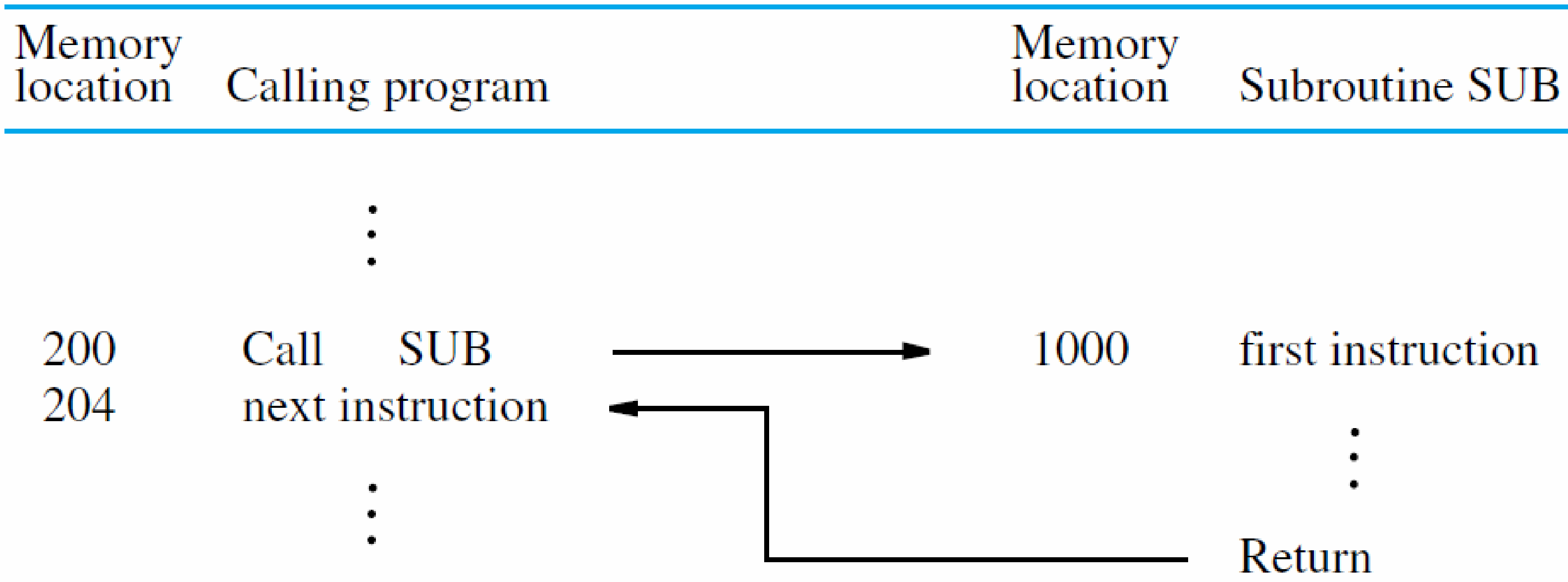
- In a given program, a particular task may be executed many times using different data
- Examples: mathematical function, list sorting
- Implement task in one block of instructions
- This is called a subroutine
- Rather than reproduce entire subroutine block in each part of program, use a subroutine call
- Special type of branch with Call instruction

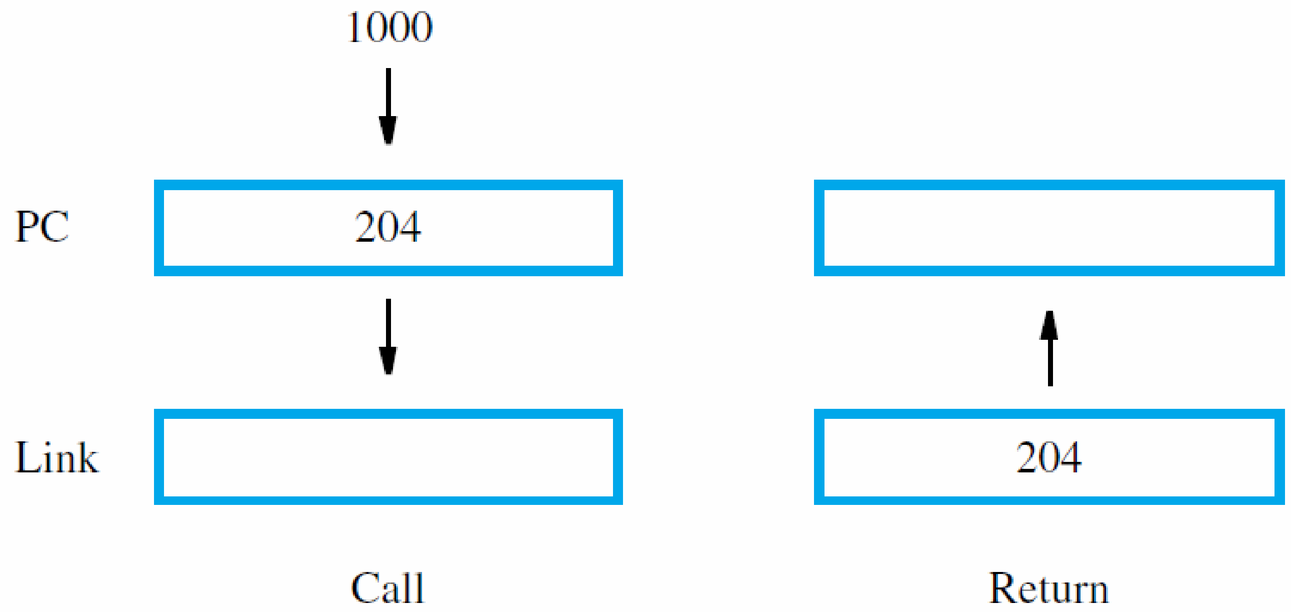
Subroutines

- Branching to same block of instructions saves space in memory, but must branch back
- The subroutine must return to calling program after executing last instruction in subroutine
- This branch is done with a Return instruction
- Subroutine can be called from different places
- How can return be done to correct place?
- This is the issue of subroutine linkage

Subroutine Linkage

- During execution of Call instruction,
  PC upated to point to instruction after Call
- Save this address for Return instruction to use
- Simplest method: place address in link register
- Call instruction performs two operations:
  store updated PC contents in link register,
  then branch to target (subroutine) address
- Return just branches to address in link register


Memory location
Calling program
Memory location
Subroutine SUB
200
Call SUB
204
next instruction
1000
first instruction
Return


1000
PC
204
Link
204
Call
Return

Subroutine Nesting and the Stack

- We can permit one subroutine to call another, which results in subroutine nesting
- Link register contents after first subroutine call are overwritten after second subroutine call
- First subroutine should save link register on the processor stack before second call
- After return from second subroutine, first subroutine restores link register

Parameter Passing

- A program may call a subroutine many times with different data to obtain different results
- Information exchange to/from a subroutine is called parameter passing
- Parameters may be passed in registers
- Simple, but limited to available registers
- Alternative: use stack for parameter passing, and also for local variables & saving registers

The Stack Frame

- Locations at the top of the processor stack are used as a private work space by subroutines
- A stack frame is allocated on subroutine entry and deallocated on subroutine exit
- A frame pointer (FP) register enables access to private work space for current subroutine
- With subroutine nesting, the stack frame also saves return address and FP of each caller

SP
(stack pointer)

saved [R4]
saved [R3]
saved [R2]
localvar3
localvar2
localvar1
saved [FP]
param1
param2
param3
param4

FP
(frame pointer)

Stack
frame
for
called
subroutine

Old TOS

ARM SUBROUTINES AND STACK

Instructions

- Subroutine linkage:

  BL SUBADDRESS

  Actions taken:

  1. The value of the updated PC is stored in R14 (LR), the Link register
  2. A branch is taken to SUBADDR

- Return from subroutine

  BX lr

  - By convention, registers r0 to r3 are used to pass parameters to subroutines, and r0 is used to pass a result back to the callers.
    - Calls between separately assembled or compiled modules => procedure call standard (*Procedure Call Standard for the ARM Architecture* specification)

Example

```
      AREA    subrout, CODE, READONLY     ; Name this block of code
              ENTRY                       ; Mark first instruction to execute
start         MOV     r0, #10                 ; Set up parameters
              MOV     r1, #3
              BL      doadd                   ; Call subroutine
stop
      More instructions


doadd         ADD     r0, r0, r1          ; Subroutine code (and prepare return value)
              BX      lr                  ; Return from subroutine
              END                         ; Mark end of file
```

Stack

- **Stack Instructions.** No specific instructions (push/pop)
  - Push and pop operations are performed by memory access instructions, with auto-increment addressing modes.
- **Stack pointer.** No special register, any general purpose register can be used (typically r13 also referred as SP)
- **Stack Types.**
  - Stack ascending and descendig
  - SP to last full or first empty

    -FA Full-Ascending -EA Empty-Ascending

    -FD Full-Descending -ED Empty-Descending

Implementing stacks with LDM and STM

Stack-oriented suffixes and equivalent addressing mode suffixes

Stack-oriented suffix	**For store or push instructions**	**For load or pop instructions**
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

Suffixes for load and store multiple instructions

Stack type	**Store**	**Load**
Full descending	STMFD (STMDB, Decrement Before)	LDMFD (LDM, increment after)
Full ascending	STMFA (STMIB, Increment Before)	LDMFA (LDMDA, Decrement After)
Empty descending	STMED (STMDA, Decrement After)	LDMED (LDMIB, Increment Before)
Empty ascending	STMEA (STM, increment after)	LDMEA (LDMDB, Decrement Before)

Stack usage: examples

```
STMFD sp!, {r0-r5}  ; Push onto a Full Descending
          ; Stack
LDMFD sp!, {r0-r5}  ; Pop from a Full Descending Stack
```

- The *Procedure Call Standard for the ARM Architecture* (AAPCS), and ARM and Thumb C and C++ compilers always use a full descending stack.
- The PUSH and POP instructions assume a full descending stack. They are the preferred synonyms for STMDB and LDM with writeback.

Stacking registers for nested subroutines

- At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.
- In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can be made safely without causing the return address to be lost.
  - If you do this, you can also return from a subroutine by popping pc off the stack at exit, instead of popping lr and then moving that value into pc. For example:

```
sub     PUSH {r5-r7,lr}       ; Push work registers and lr
        ; code
        BL somewhere_else
        ; code
        POP {r5-r7,pc}        ; Pop work registers and pc
```

Coldfire SUBROUTINES AND STACK

Subroutine Linkage

- Address register A7 is the stack pointer (SP)
- Subroutine call/return instructions use A7 to save/restore return address automatically
- JSR, BSR instructions for subroutine calls
- RTS instruction for returning from subroutines
- Pass parameters in registers or using stack
- If parameters pushed to or popped from stack, register A7 must always be a multiple of 4

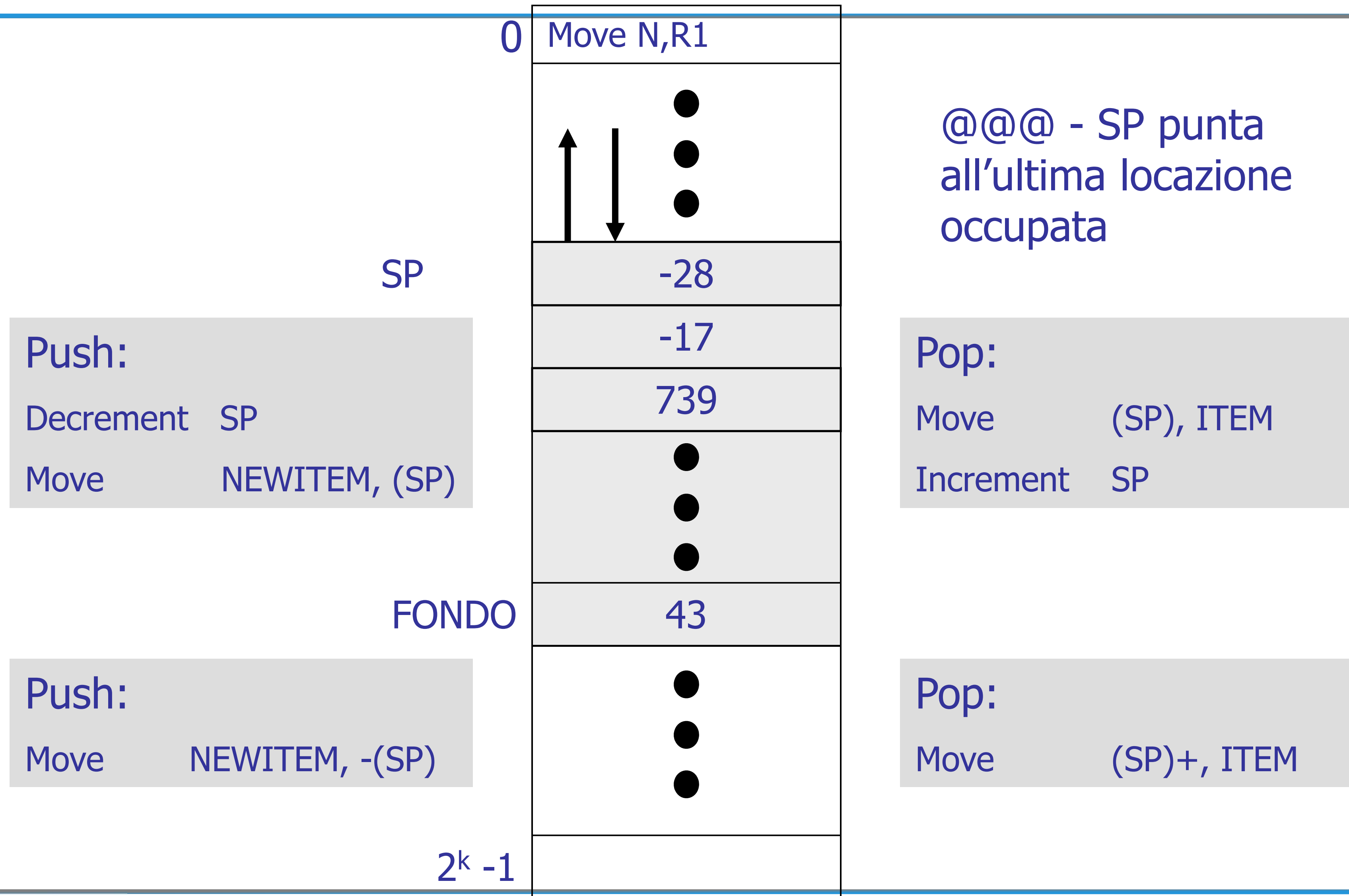
Calling program

```
        MOVEA.L   #NUM1, A2    Put the address NUM1 in A2.
        MOVE.L    N, D1        Put the number of elements n in D1.
        BSR       LISTADD      Call subroutine LISTADD.
        MOVE.L    D0, SUM      Store the sum in SUM.
        next instruction
        ⋮
```

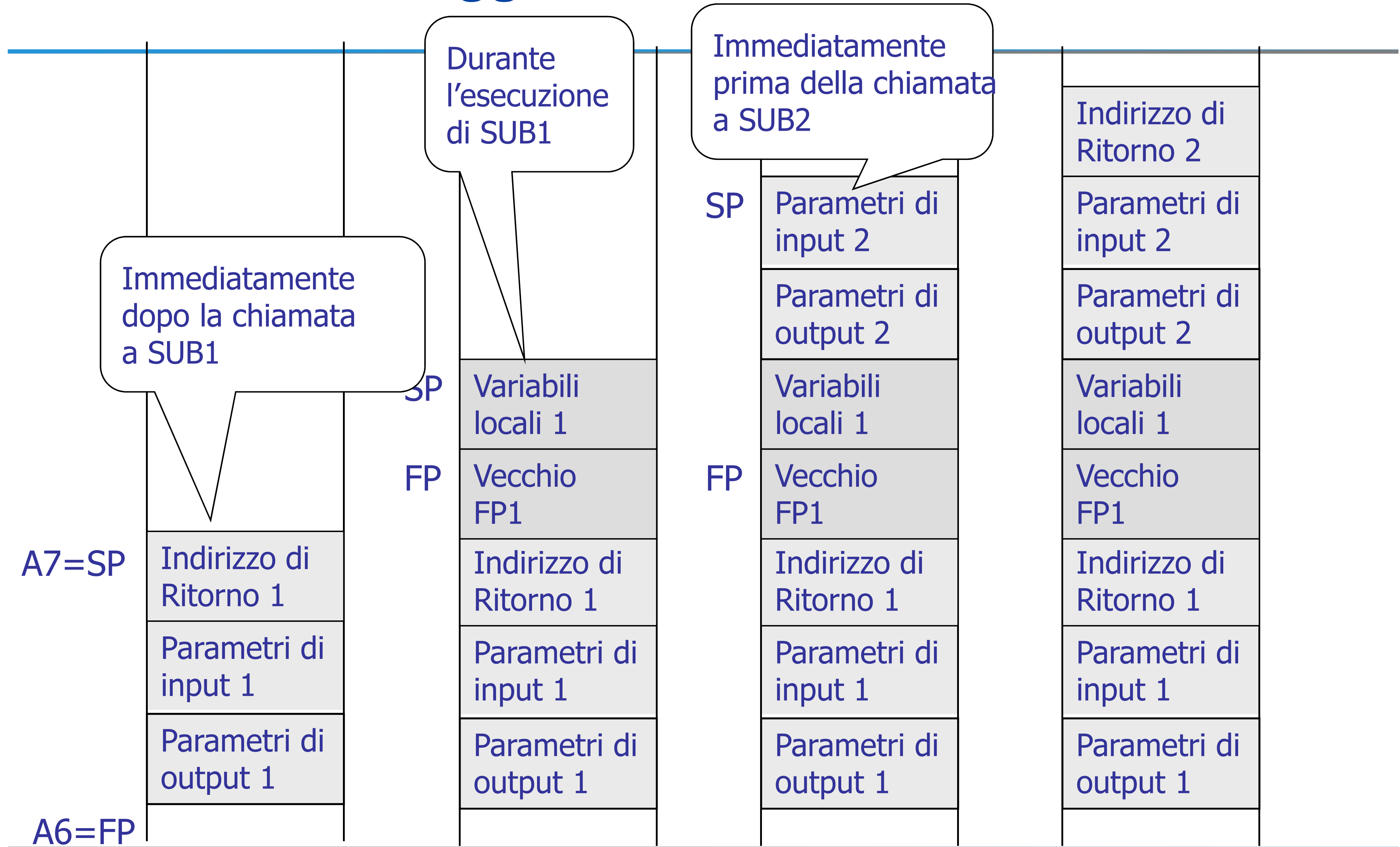
Subroutine

```
LISTADD: CLR.L    D0
LOOP:    ADD.L    (A2)+, D0    Accumulate sum in D0.
         SUBQ.L   #1, D1
         BGT      LOOP
         RTS
```

Pila



Passaggio Parametri su Stack



Link and allocate

- Syntassi:
  - LINK An,#<displacement>
- Funzionamento:

1. Esegue push su stack del contenuto del registro indirizzo specificato
2. Il registro indirizzo specificato viene caricato con il nuovo valore dello stack pointer
3. Il displacement viene esteso in segno e sommato a SP. Questo valore viene assegnato a SP.
4. Durante l'esecuzione SP varia, mentre FP rimane costante

Parametri in stack – div916.a68 - 1/2

```
DIVPQ       ADDA.L      #-10,SP
            MOVE.L      P,-(SP)
            CLR.L       -(SP)
            MOVE.L      Q,-(SP)
            JSR         DIVIDEL
            MOVE.L      (SP)+,PMODQ
            MOVE.L      (SP)+,PDIVQ
            TST.W       (SP)+
            BNE         DIVOVF

*Area Dati

P           DC.L        10
Q           DC.L        5
PDIVQ       DS.L        1
PMODQ       DS.L        1
```

Codice programma principale

Parametri in stack – div916.a68 - 2/2

```
DIVIDEL     LINK        A6,#-2
            MOVE.W      #32,CNT(A6)
            CLR.W       STATUS(A6)
            MOVE.L      HIDVND(A6),D0
            MOVE.L      LODVND(A6),D1
            CMP.L       DVSR(A6),D0
            BCS.S       DIVLUP
            MOVE.W      #1,STATUS(A6)
            BRA.S       CLNSTK
DIVLUP      LSL.L       #1,D1
            …
            …
            BGT.S       DIVLUP
            MOVE.L      D0,REM(A6)
            MOVE.L      D1,QUOT(A6)
CLNSTK      UNLK        A6
            MOVEA.L     (SP)+,A0
            ADDA.L      #STATUS-DVSR,SP
            JMP         (A0)
```

Codice subroutine

Parametri in stack – Esecuzione

ASIM - simple.cfg

File Proc_Unit View Simulation Window Tools Help

simple: Memoria 2

```
00008000  4E 56 FF FE 3D 7C 00 20 FF FE 42 6E 00 1C 20 2E 00 0C 22 2E 00 10
00008016  B0 AE 00 08 65 08 3D 7C 00 01 00 1C 60 20 E3 89 E3 90 65 06 B0 AE
```

simple: M68000 1

```
************************************************************************
* Wakerly - Table 9-16 Page 332
*
* input:
*
* output:
*
************************************************************************

                 ORG      $8000
```

```
D0:00000000 D4:00000000 A0:00000000 A4:00000000
D1:00000000 D5:00000000 A1:00000000 A5:00000000
D2:00000000 D6:00000000 A2:00000000 A6:00000000
D3:00000000 D7:00000000 A3:00000000 A7:00009000
| Cycles |      |T S  INT   XNZVC|  A7':00009200
|00000000|  |SR:0010011100000000|   PC:00000000
```

For Help, press F1

Ritorno – Esecuzione

ASIM - simple.cfg

File Proc_Unit View Simulation Window Tools Help

simple: Memoria 2

simple: M68000 1

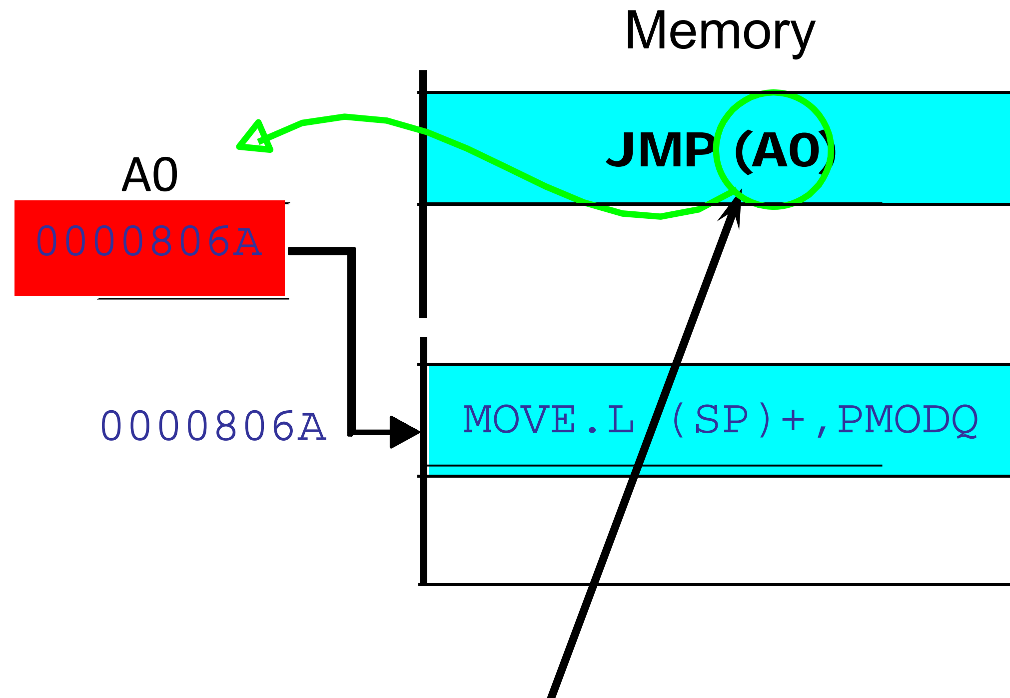
```
                MOVE.L   D1,QUOT(A6)
CLNSTK   UNLK            A6
                MOVEA.L (SP)+,A0
                ADDA.L   #STATUS-DUSR,SP
                JMP              (A0)
*****************************************************************************
* Main Program: COMPUTES PDIVQ:=P DIV Q; PMODQ:=P MOD Q
*
* Wakerly - Table 9-17 Page 336

DIVPQ           ADDA.L   #-10,SP
```

```
D0:00000000  D4:00000000  A0:0000806A  A4:00000000
D1:00000002  D5:00000000  A1:00000000  A5:00000000
D2:00000000  D6:00000000  A2:00000000  A6:00000000
D3:00000000  D7:00000000  A3:00000000  A7:00009000
| Cycles |      |T S  INT   XNZVC|  A7':000091EA
|00000B72|   |SR:0010011100000000|   PC:00008048
```

For Help, press F1

Ritorno - Schema



Questa istruzione significa: salta all'istruzione puntata dal registro indirizzo A0

L'istruzione specifica l'operando come (A0)

Ritorno – Mappa della memoria

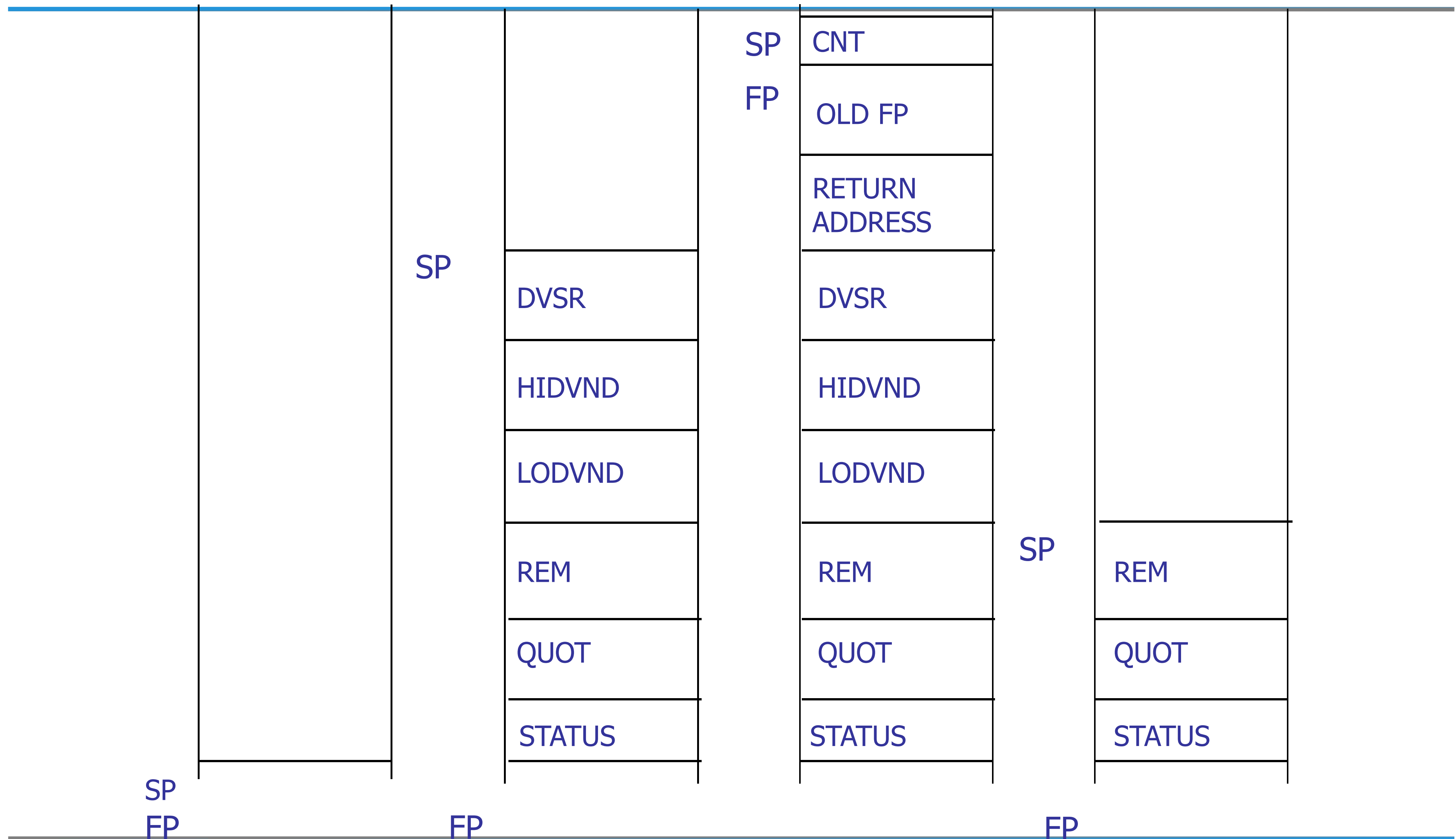
div916.lis - WordPad

File Modifica Visualizza Inserisci Formato ?

```
00008048  DFFC 00000014          44            ADDA.L     #STATUS-DVSR,SP
0000804E  4ED0                   45            JMP        (A0)
00008050                         46   ************************************
00008050                         47   * Main Program: COMPUTES PDIVQ:=P DIV (
00008050                         48   *
00008050                         49   * Wakerly - Table 9-17 Page 336
00008050                         50
00008050  DFFC FFFFFFF6          51   DIVPQ    ADDA.L     #-10,SP
00008056  2F39 00008500          52            MOVE.L     P,-(SP)
0000805C  42A7                   53            CLR.L      -(SP)
0000805E  2F39 00008504          54            MOVE.L     Q,-(SP)
00008064  4EB9 00008000          55            JSR        DIVIDEL
0000806A  23DF 0000850C          56            MOVE.L     (SP)+,PMODQ
00008070  23DF 00008508          57            MOVE.L     (SP)+,PDIVQ
00008076  4A5F                   58            TST.W      (SP)+
00008078  6600 9786              59            BNE        DIVOVF
0000807C                         60
0000807C                         61   *Area Dati
00008500                         62            ORG        $8500
00008500                         63
00008500  0000000A               64   P        DC.L       10
00008504  00000005               65   Q        DC.L       5
00008508                         66   PDIVQ    DS.L       1
0000850C                         67   PMODQ    DS.L       1
00008510                         68
00008510  =00001800              69   DIVOVF   EQU        $1800
00008510                         70
```

Per aprire la Guida, premere F1.

Evoluzione dello stack - div916.a68



Example Programs

- First example program computes:

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

- First elements of each array, $A(0)$ and $B(0)$, are stored at memory locations AVEC and BVEC
- Consider RISC and CISC versions of program
- Use Multiply instruction on pairs of elements and accumulate sum with Add instruction
- Some processors have MultiplyAccumulate

```
        Move              R2, #AVEC      R2 points to vector A.
        Move              R3, #BVEC      R3 points to vector B.
        Load              R4, N          R4 serves as a counter.
        Clear             R5             R5 accumulates the dot product.
LOOP:   Load              R6, (R2)       Get next element of vector A.
        Load              R7, (R3)       Get next element of vector B.
        Multiply          R8, R6, R7     Compute the product of next pair.
        Add               R5, R5, R8     Add to previous sum.
        Add               R2, R2, #4     Increment pointer to vector A.
        Add               R3, R3, #4     Increment pointer to vector B.
        Subtract          R4, R4, #1     Decrement the counter.
        Branch_if_[R4]>0  LOOP           Loop again if not done.
        Store             R5, DOTPROD    Store dot product in memory.
```

```
        Move        R2, #AVEC       R2 points to vector A.
        Move        R3, #BVEC       R3 points to vector B.
        Move        R4, N           R4 serves as a counter.
        Clear       R5              R5 accumulates the dot product.
LOOP:   Move        R6, (R2)+       Compute the product of
        Multiply    R6, (R3)+         next components.
        Add         R5, R6          Add to previous sum.
        Subtract    R4, #1          Decrement the counter.
        Branch>0    LOOP            Loop again if not done.
        Move        DOTPROD, R5     Store dot product in memory.
```

Tabella A2.5 Collegamento di sottoprogrammi NIOS II, ColdFire, ARM, IA-32

Caratteristica	NIOS II	ColdFire	ARM	IA-32
Meccanismo di collegamento	Registro di collegamento	In pila, implicito	Registro di collegamento	In pila, implicito
Registri:				
PC		PC	R15 | PC	EIP
SP	r27 | sp	A7	R13 | SP	ESP
FP	r28 | fp	A*j*	R12 | FP	EBP
LR	r31 | ra		R14 | LR	
Istruzioni di chiamata e di rientro	call *l*	BSR *l*	BL *l*	CALL *l*
	callr r*i*	JSR *a*		
	ret	RTS	LDMFD SP!, {*r*,PC}	RET
Altre istruzioni rilevanti		MOVEM.L *r*, (A7)	STMFD SP!, {*r*,LR}	PUSH *s*
				POP *d*
		LINK A*j*, #*p*		PUSHAD
		UNLK A*j*		POPAD

Legenda:

A*j* registro di indirizzo usato come puntatore all'area di attivazione ($0 \leq j \leq 6$);
l etichetta;
a indirizzo del sottoprogramma, modi: assoluto, indiretto da registro, con base, indice e spiazzamento, relativo a PC (con indice);
r lista di registri;
p spiazzamento;
s operando sorgente da impilare;
d destinazione di operando da spilare.

Example Programs

- Second example searches for 1st occurrence of pattern string *P* in target string *T*
- String *P* has length *m* and string *T* has length *n*
- Algorithm to implement in RISC/CISC styles:

$$
\begin{array}{l}
\textbf{for} \quad i \leftarrow 0 \ \textbf{to} \ n - m \ \textbf{do} \\
\qquad j \leftarrow 0 \\
\qquad \textbf{while} \ j < m \ \textbf{and} \ P[j] = T[i + j] \ \textbf{do} \\
\qquad \qquad j \leftarrow j + 1 \\
\qquad \textbf{if} \ j = m \ \textbf{return} \ i \\
\textbf{return} \ -1
\end{array}
$$

```
           Move                  R2, #T           R2 points to string T.
           Move                  R3, #P           R3 points to string P.
           Load                  R4, N            Get the value n.
           Load                  R5, M            Get the value m.
           Subtract              R4, R4, R5       Compute n − m.
           Add                   R4, R2, R4       The address of T(n − m).
           Add                   R5, R3, R5       The address of P(m).
LOOP1:     Move                  R6, R2           Use R6 to scan through string T.
           Move                  R7, R3           Use R7 to scan through string P.
LOOP2:     LoadByte              R8, (R6)         Compare a pair of
           LoadByte              R9, (R7)           characters in
           Branch_if_[R8]≠[R9]   NOMATCH            strings T and P.
           Add                   R6, R6, #1       Point to next character in T.
           Add                   R7, R7, #1       Point to next character in P.
           Branch_if_[R5]>[R7]   LOOP2            Loop again if not done.
           Store                 R2, RESULT       Store the address of T(i).
           Branch                DONE
NOMATCH:   Add                   R2, R2, #1       Point to next character in T.
           Branch_if_[R4]≥[R2]   LOOP1            Loop again if not done.
           Move                  R8, #–1          Write –1 to indicate that
           Store                 R8, RESULT         no match was found.
DONE:      next instruction
```

```
          Move          R2, #T          R2 points to string T.
          Move          R3, #P          R3 points to string P.
          Move          R4, N           Get the value n.
          Move          R5, M           Get the value m.
          Subtract      R4, R5          Compute n − m.
          Add           R4, R2          The address of T(n − m).
          Add           R5, R3          The address of P(m).
LOOP1:    Move          R6, R2          Use R6 to scan through string T.
          Move          R7, R3          Use R7 to scan through string P.
LOOP2:    MoveByte      R8, (R6)+       Compare a pair of
          CompareByte   R8, (R7)+         characters in
          Branch≠0      NOMATCH           strings T and P.
          Compare       R5, R7          Check if at P(m).
          Branch>0      LOOP2           Loop again if not done.
          Move          RESULT, R2      Store the address of T(i).
          Branch        DONE
NOMATCH:  Add           R2, #1          Point to next character in T.
          Compare       R4, R2          Check if at T(n − m).
          Branch≥0      LOOP1           Loop again if not done.
          Move          RESULT, #−1     No match was found.
DONE:     next instruction
```

Concluding Remarks

- Many fundamental concepts presented:
  - memory locations, byte addressability, endianness
  - assembly-language and register-transfer notation
  - RISC-style and CISC-style instruction sets
  - addressing modes and instruction execution
  - assembler to generate machine instructions
  - subroutines and the processor stack
- Later chapters build on these concepts