ColdFire addressing modes

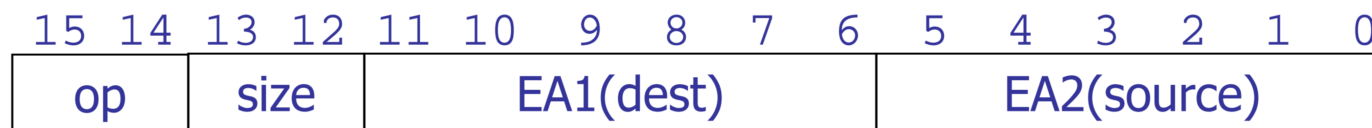
Instructions

- One, two, or three consecutive words
- *OP-code* word is first – it specifies operation
- Also provides some addressing information; one or two *extension* words provide more
- Most arithmetic and data-transfer instructions have source/destination operands:
  OP src, dst
- .L, W., or .B suffix for OP code specifies size

Modi di Indirizzamento

- Register Direct
  - Data-register Direct
  - Address-register Direct
- Immediate (or Literal)
- Absolute
  - Short
  - Long
- Address-register Indirect
- Auto-Increment
- Auto-Decrement
- Indexed short
- Based
- Based Indexed
  - Short
  - Long
- Relative
- Relative Indexed
  - Short
  - Long

Encoding for the MOVE instruction



MOVE

- the EA field is 6 bit long and is organized in two sub-fields (3 bit each)

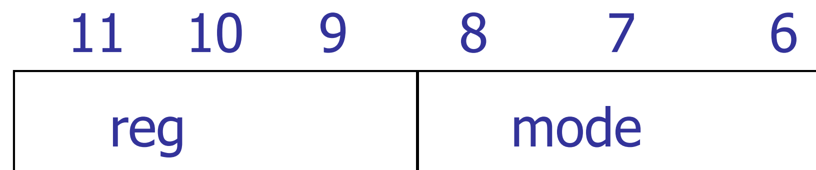
alcuni modi possibili:

mode	reg	syntax	EA	name	#e.w.
0	0-7	Dn	Dn	Data-register direct	0
1	0-7	An	An	Address-register direct	0
2	0-7	(An)	MEM[An]	Address-register indirect	0
7	0	addr	MEM[addr]	Absolute short	1
7	4	#data	data	Immediate	1o 2

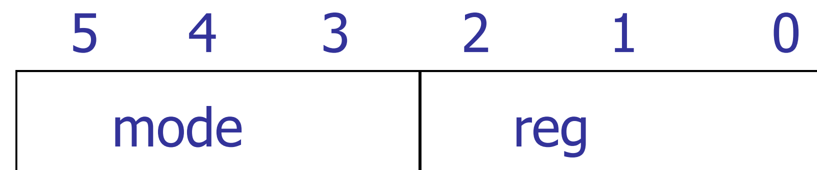
- all the addressing modes can be used for both sourse and destination (with the exception of the immediate for the destination)
- The MOVE instruction is full ortogonal (ortogonal ISA => all the instruction are ortogonal)

Addressing Mode encoding

- EA is encoded over 6 bits in the first word of the instruction (opcode word)
- The MOVE instruction has two of such a filed (one for each operand)
- Some addressing modes need more information given in additional words (extension words)

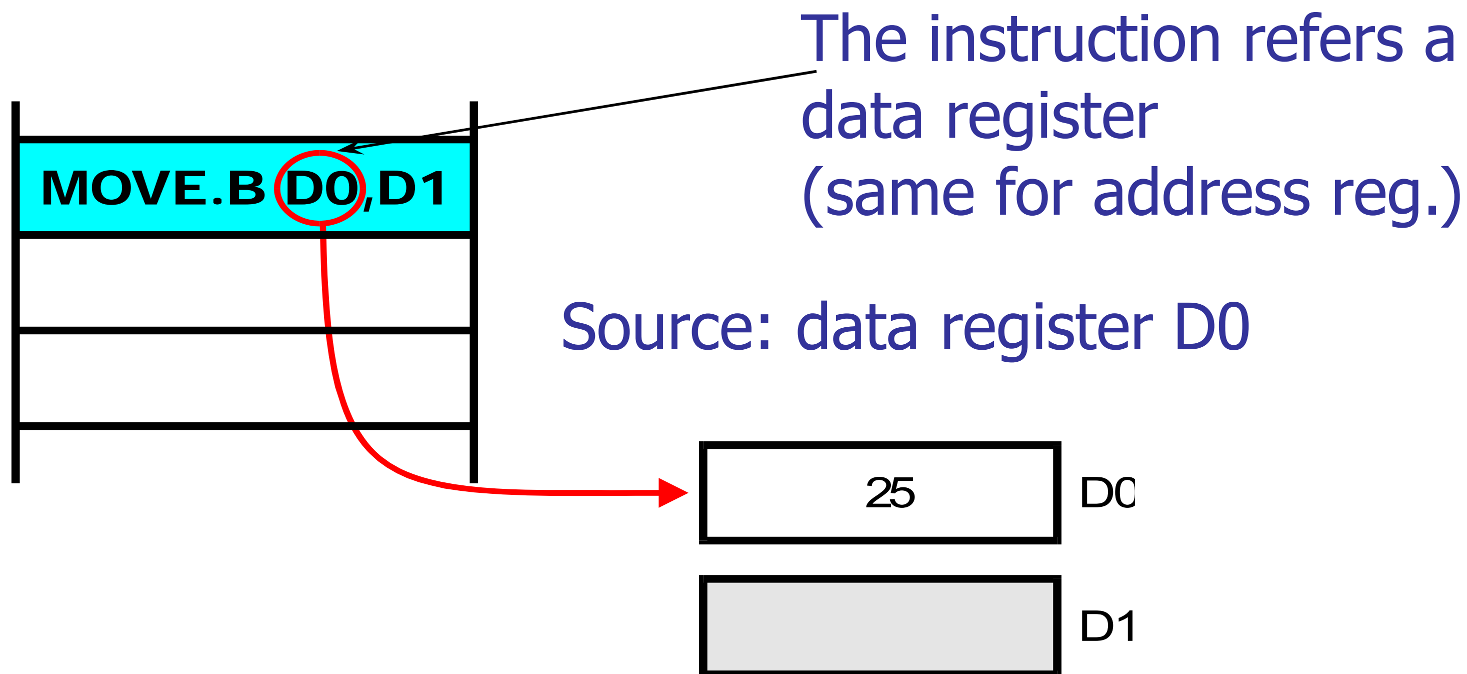


Destination operand MOVE



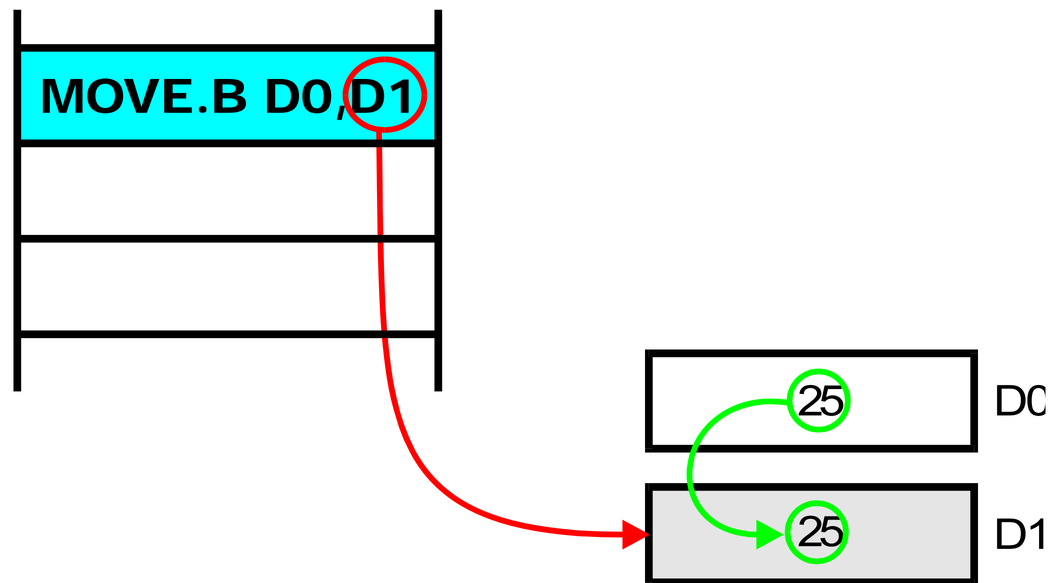
EA field for all the other cases

Register Direct Addressing



The MOVE.B D0,D1 instruction has data registers for both source and destination

Register Direct Addressing



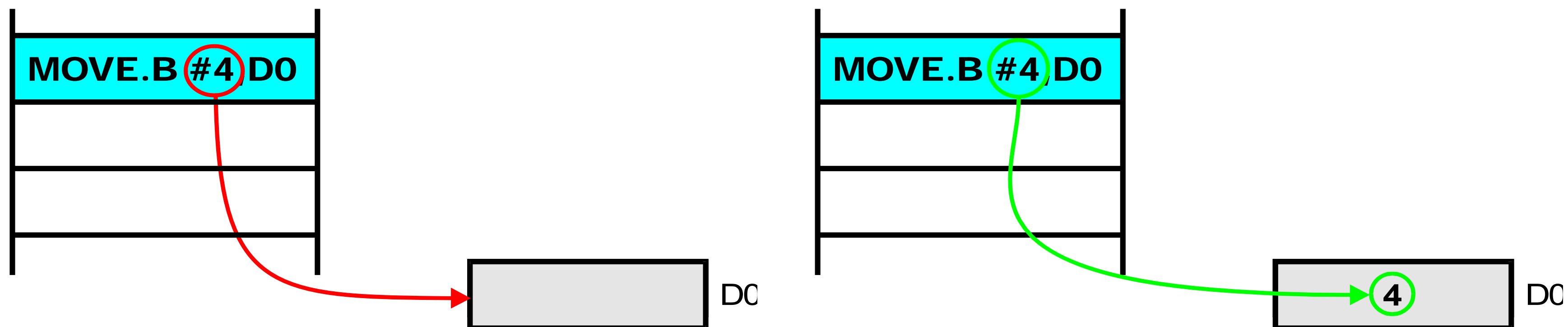
The final result is that D0 content is copied to D1

Register Direct Addressing

- No access to external memory: fast
- One word only instructions (only 6 bits per operand)
  - **Mode = 0, reg = 0-7 per Dn**
  - **Mode = 1, reg = 0-7 per An**
- Used to store frequently used variable (scratchpad storage)

Immediate Addressing

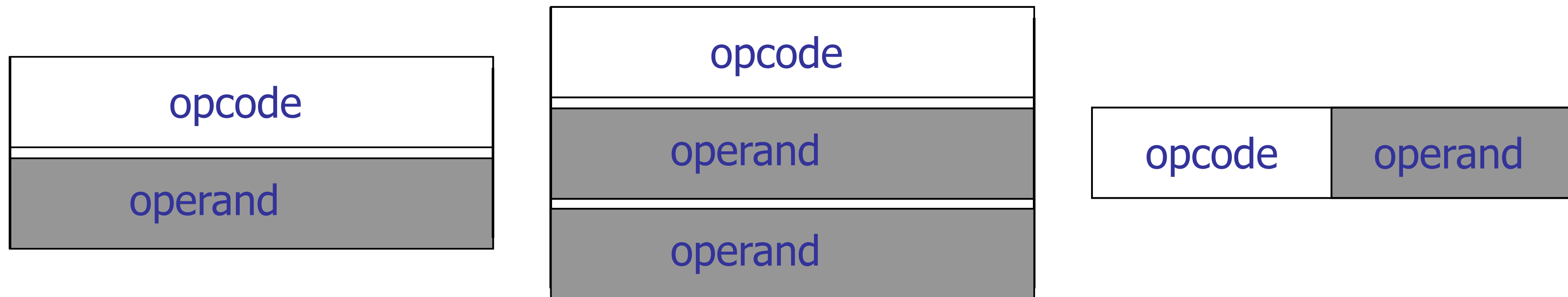
- The real operand is made available as part of the instruction
- Only used for sourse operand
- The symbol # used ahead of the value
- The immediate operand is also said «literal»



The MOVE.B #4,D0 has a literal as source operand and makes use of register direct for destination

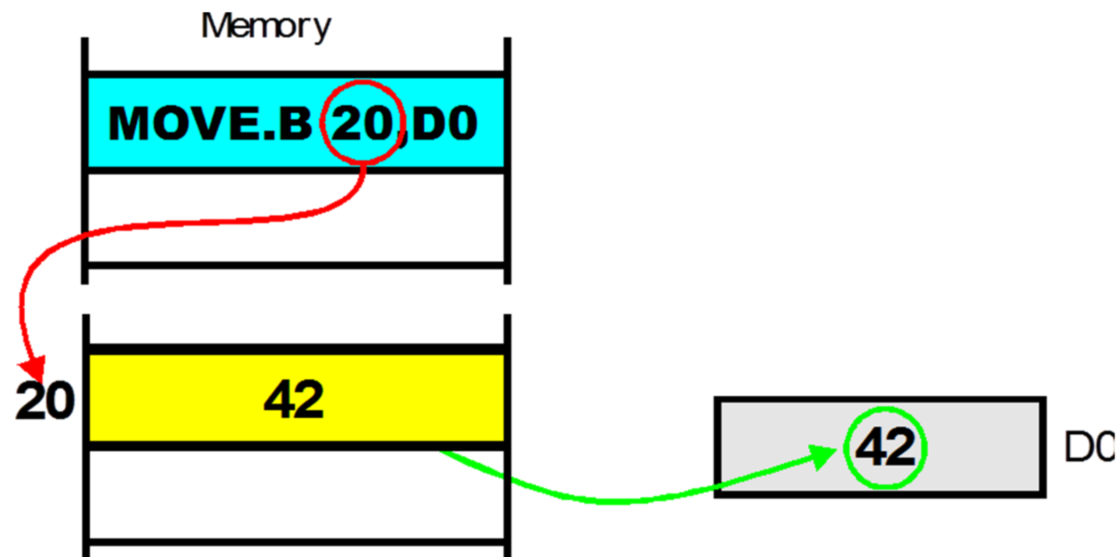
ImmediateAddressing - Encoding

- May use extension words for the operand
  - **Mode = 7, reg = 4**



Absolute Addressing (or Direct Addressing)

- The instruction refers a memory address that contains the actual operand
- Two memory accesses:
  - Instruction fetch
  - Operand assembly



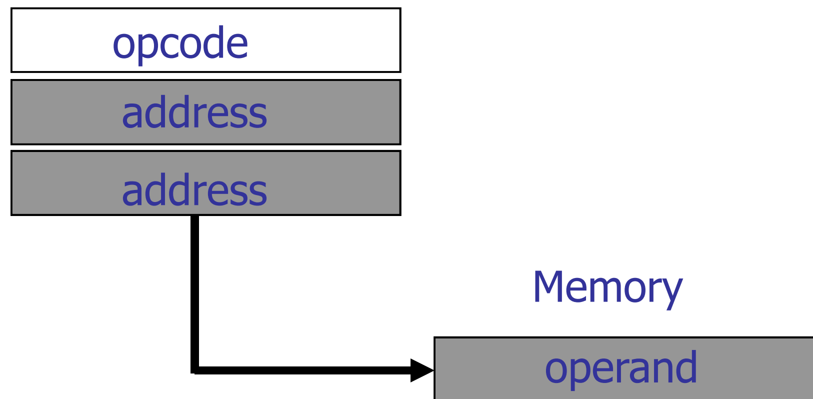
Absolute Addressing - Codifica

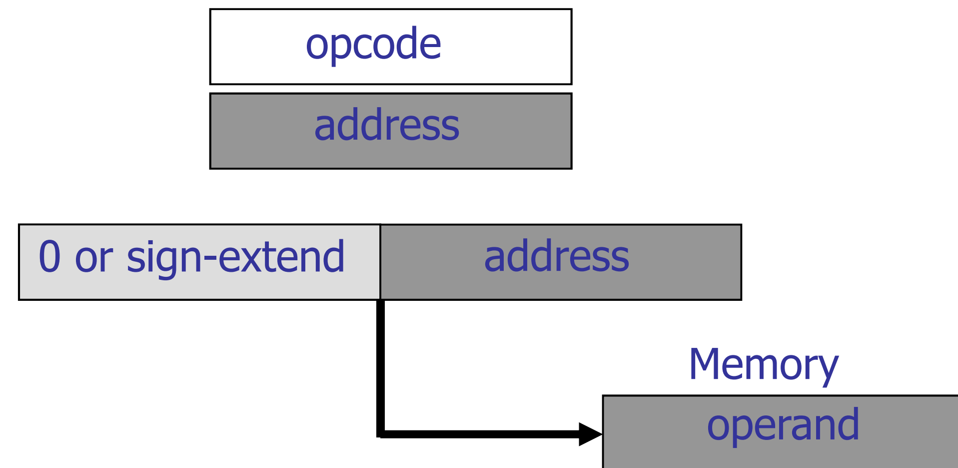
Absolute Long:

mode = 7, reg = 1

Absolute Short:

mode = 7, reg = 0





Example for basic modes

Lets consider the high level statement

Z = Y + 24

It can be performed by the following assembly program

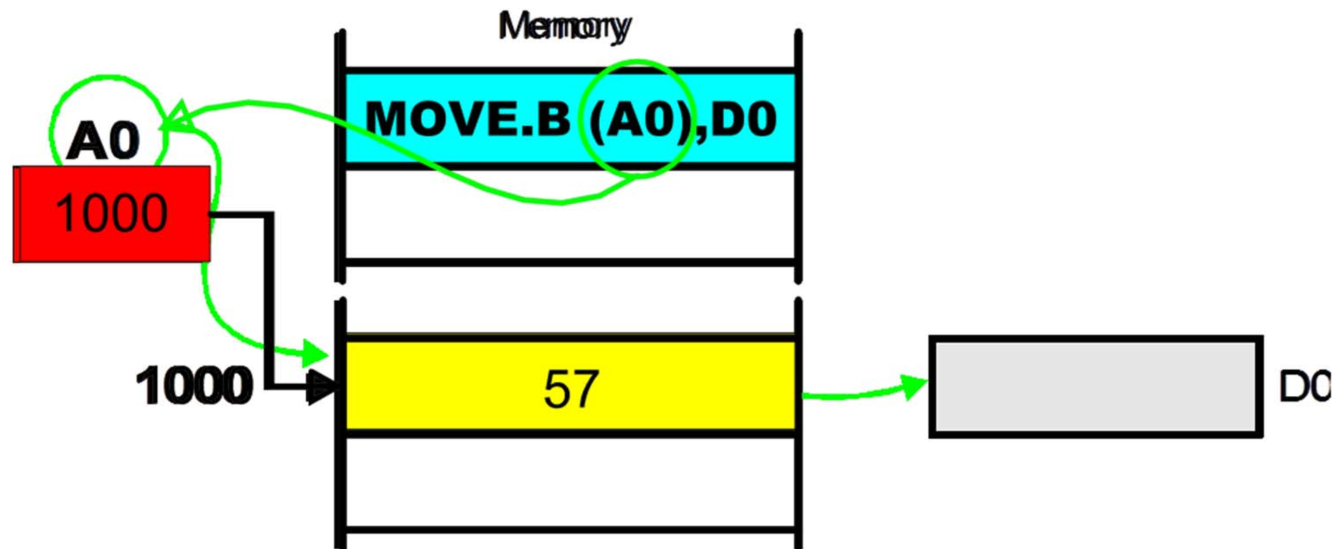
```
        ORG       $400   code section
        MOVE.B    Y,D0
        ADD       #24,D0
        MOVE.B    D0,Z

        ORG       $600   data section
Y       DC.B      27     store a constant
Z       DS.B      1      reserve a byte for Z
```

Assembled code

```
1     00000400                         ORG  $400
2     00000400 103900000600            MOVE.B   Y,D0
3     00000406 06000018                ADD.B  #24,D0
4     0000040A 13C000000601            MOVE.B   D0,Z
5     00000410 4E722700                STOP    #$2700
6              *
7     00000600                         ORG     $600
8     00000600 1B               Y:     DC.B    27
9     00000601                  Z:     DS.B    1
10    00000400                         END     $400
```

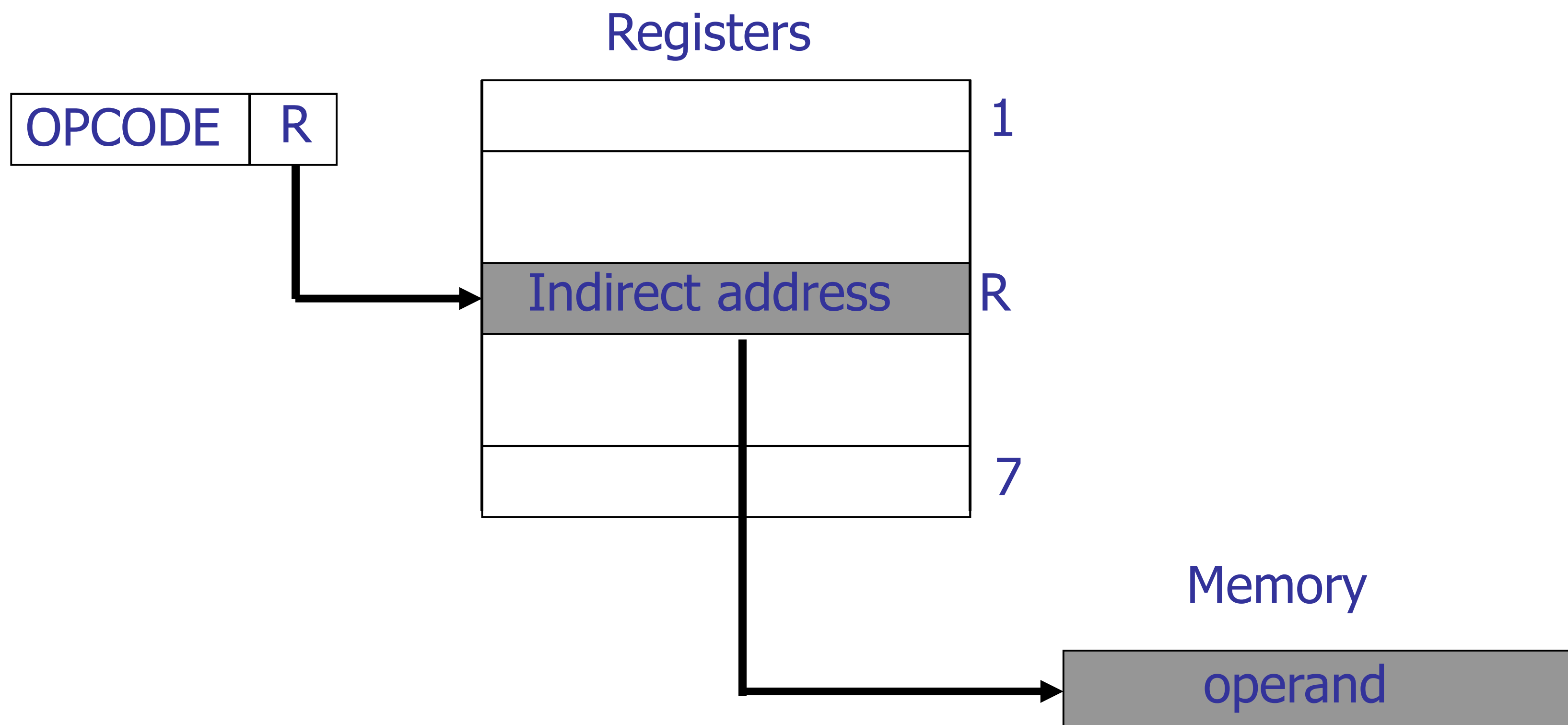
Address Register Indirect Addressing

- The EA is the content of the specified address register



Address Register Indirect Addressing - Encoding

- mode = 2; reg = 0-7



Auto-increment

- As address indirect but the after the instruction the content of the address is updated by increasing its value according to the data size
- Example:
  - MOVE.W (A7)+, D0   Pop to D0 from A7 stack
- Encoding
  - mode = 3, reg = 0-7

Auto-decrement

- As address indirect but before the instruction the content of the address is updated by decreasing its value according to the data size

- Example:
  - MOVE.W D0,-(A7) Push of D0 to A7 stack

- Encoding
  - mode = 4, reg = 0-7

Example

```
* File: autoinc.a68 - Sum up consecutive numbers

          ORG      $8000
          MOVE.B   #5,D0
          LEA      Table,A0      A0 points the list
          CLR.B    D1            clear the accumulator
Loop      ADD.B    (A0)+,D1      add up next element
          SUB.B    #1,D0
          BNE      Loop
          ORG      $8100
Table     DC.B     1,2,3,4,5     Sample vector
```

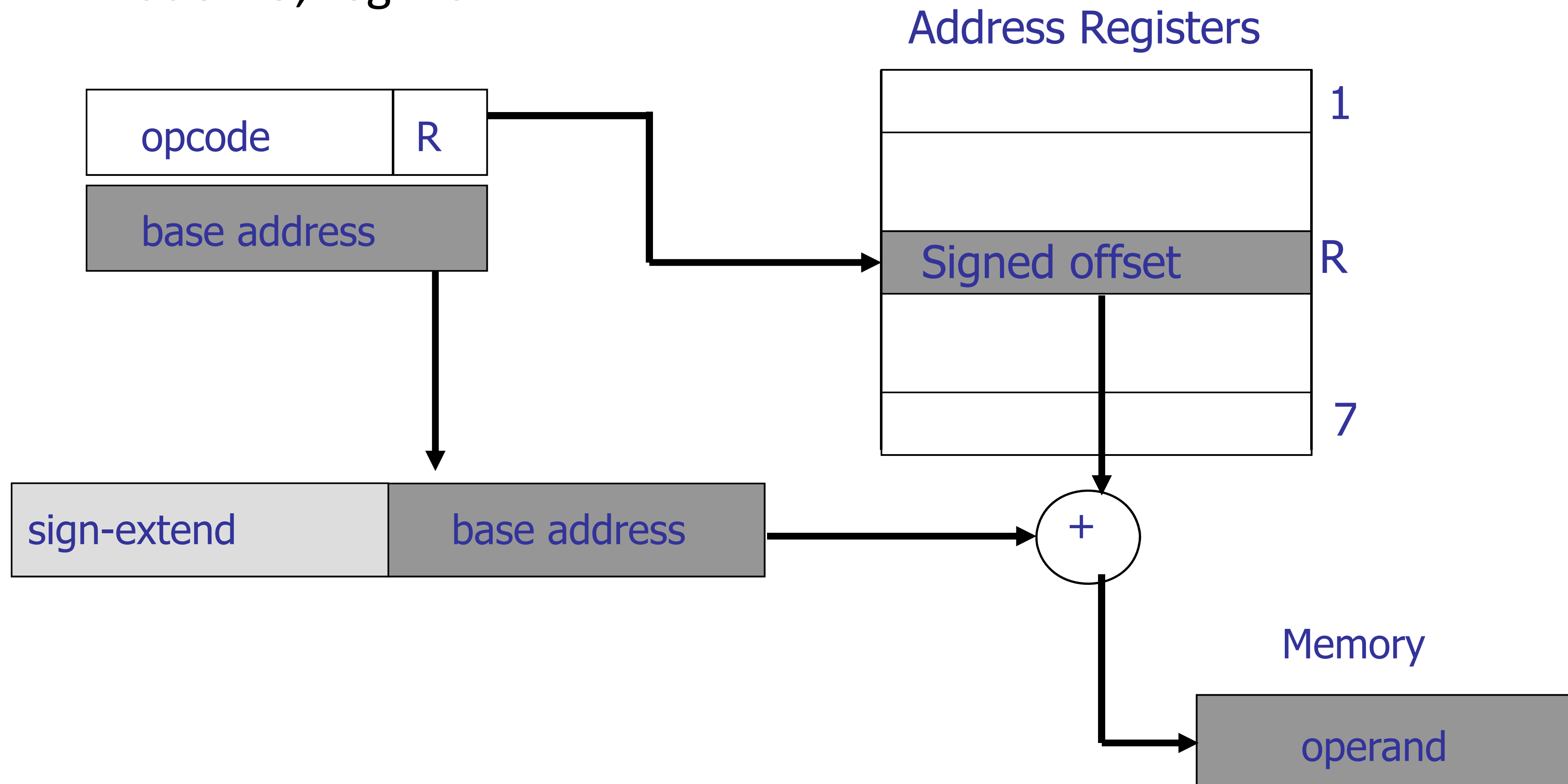
Indexed

- In generale, l'Indexed Addressing combina due componenti mediante somma, per formare l'EA
  - Il primo componente è detto *base address* ed è specificato come parte dell'istruzione (come nell'absolute addressing)
  - Il secondo componente è detto *index register* e contiene il valore da sommare al base address per ottener l'EA
- È adatto per accedere ai valori di array e di tabelle
- Il processore MC68000 non supporta esplicitamente l'Indexed Addressing. Tuttavia, è possibile usare l'Indexed Short Addressing nei (32+32)Kbyte agli estremi dei 4GB dello spazio di memoria
- Esempio:

```
MOVEA.W     I,AO
MOVE.B      CLIST-1(AO),D1        Leggi clist[i]
```

Indexed Short Addressing - Codifica

- mode = 5, reg = 0-7

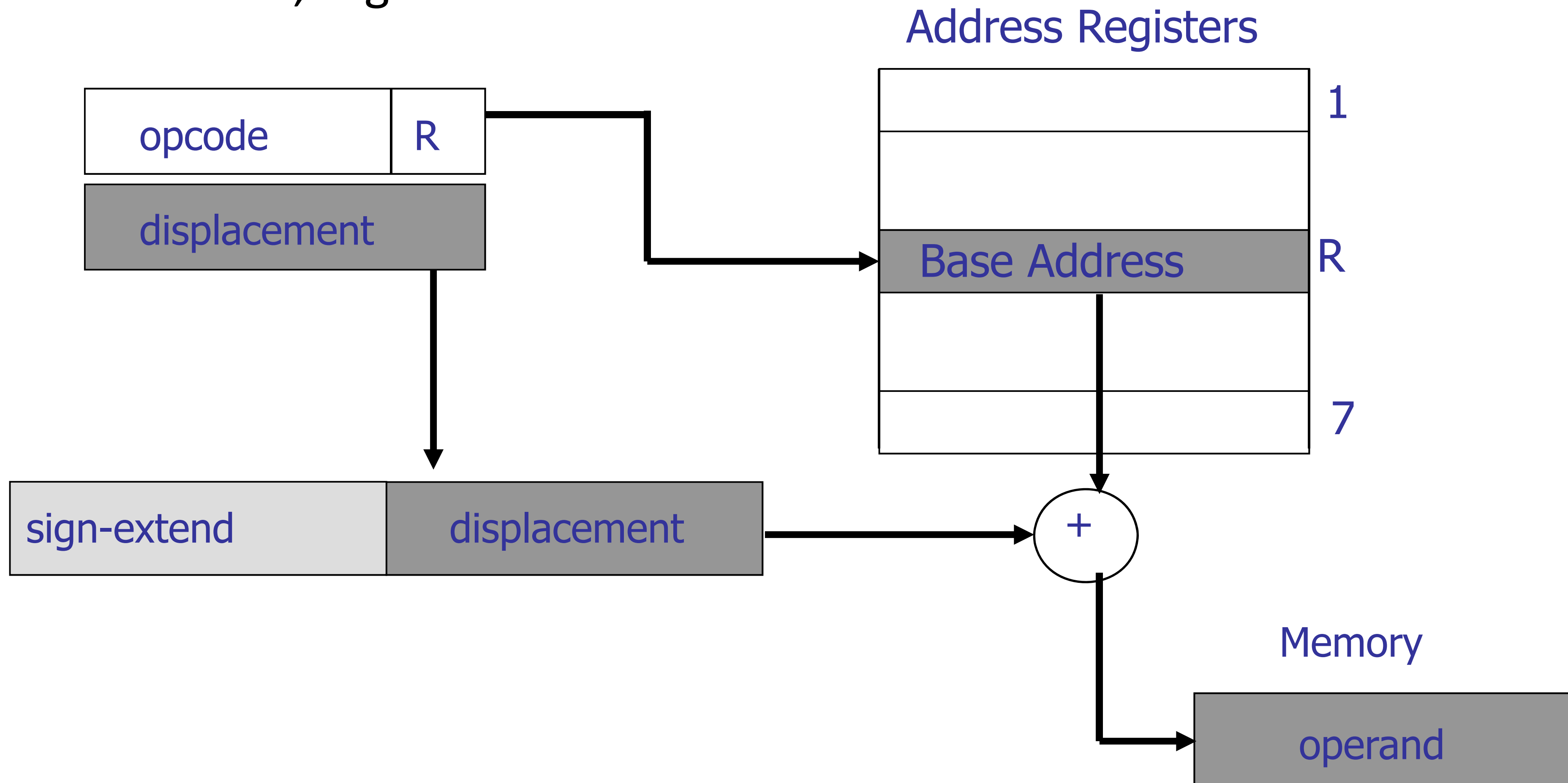


Based Addressing

- Based Addressing è esattamente l'inverso dell'Indexed Addressing, in quanto combina due componenti mediante somma, per formare l'EA, ma:
  - Il primo componente è detto *displacement* ed è specificato come parte dell'istruzione (come nell'absolute addressing)
  - Il secondo componente è detto *base address* ed è contenuto in un registro
- È adatto per accedere ai valori di array e di tabelle di cui siconosca la posizione relativa ad assembly time, ma non quella iniziale
- Il processore MC68000 supporta il Based Addressing come l'Indexed

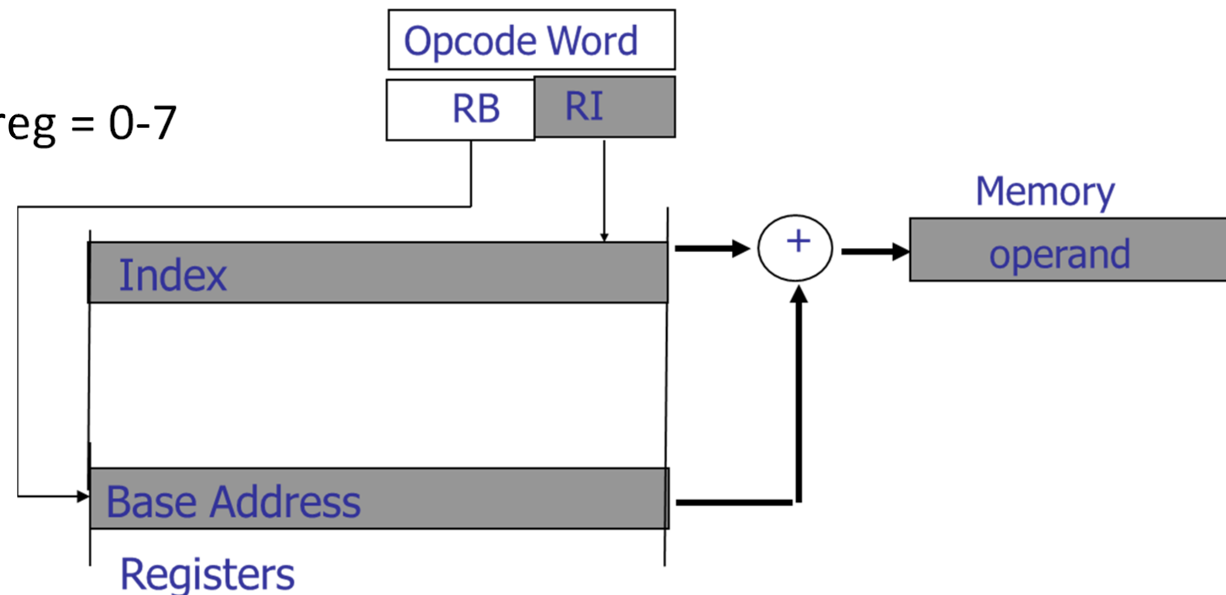
Based Addressing - Codifica

- mode = 5, reg = 0-7



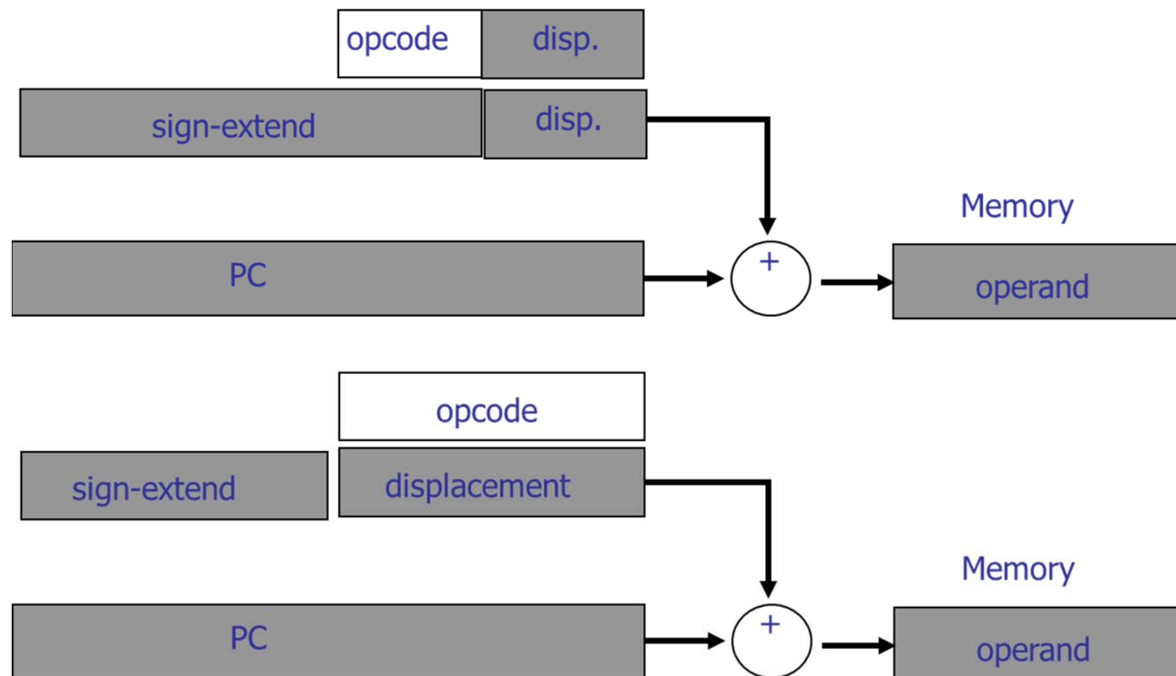
Based Indexed

- Based Indexed Addressing: EA given by the sum of two components:
  - *base address*
  - *displacement*
- Useful for array and tables
- Coldfire (MC68000) supports both Short Based Indexed and Long Based Indexed
- Encoding:
  - mode = 6, reg = 0-7



Relative Addressing

- (Relative to the PC)
- The EA is given by adding the displacement to the PC value
- Often small displacements, 8 or 16 bits, to point to an instruction next to the current one (instead of absolute 32 bits addresses)
- Encoding
  - mode = 7, reg = 2



Relative Indexed Addressing

- Similar to the Based Indexed but the base register is substituted by PC
- Encoding
  - mode = 7, reg = 3

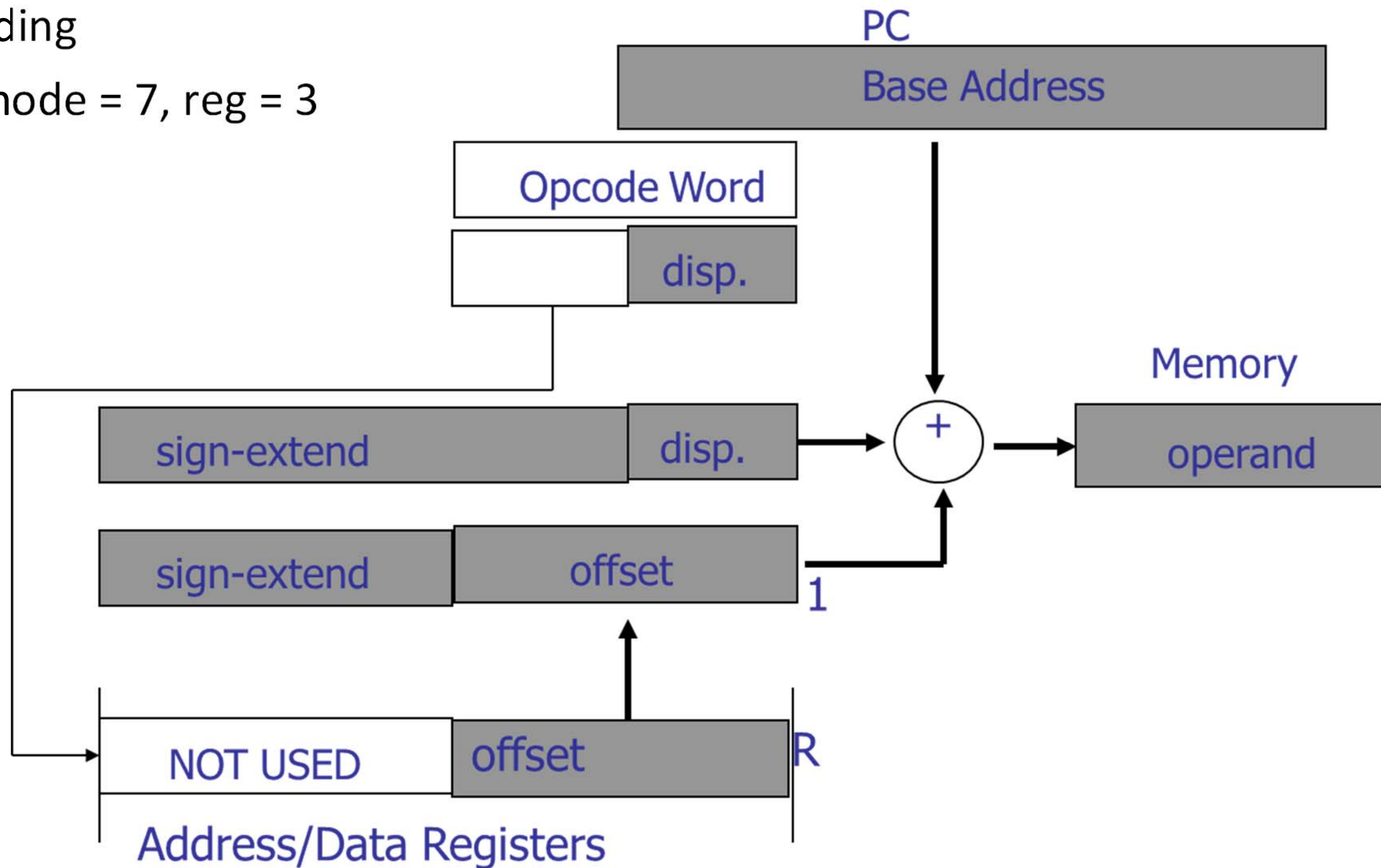


Tabella A2.2 Modi di indirizzamento e relativa notazione simbolica nelle ISA NIOS II, ColdFire, ARM e IA-32

Modo di indirizzamento	NIOS II	ColdFire	ARM	IA-32
Immediato	Valore	#Valore	#Val	Valore
Assoluto o diretto	LOC(r0)	Valore	Val	LOC
Di registro	r*i*	R*i*	R*i*	R
Indiretto di registro	(r*i*)	(A*i*)	[R*i*]	[R]
Con base e spiazzamento	X(r*i*)	W(A*i*)	[R*i*,#Val]	[R+X]
Con indice e spiazz.				[R_x*S+X]
Con base e indice			[R*i*,±R*j*,s]	[R+R_x*S]
Con base, indice e spiazz.		B(A*i*,R*j*)		[R+R_x*S+X]
Con autoincremento		(A*i*)+		
Con autodecremento		–(A*i*)		
Relativo a PC		W(PC)	L	L
Relativo a PC con indice		B(PC,R*i*)		
Indiretto da memoria				*LOC oppure [R_x*S+X]
Con pre-base e spiazz.			[R*i*,#Val]!	
Con post-base e spiazz.			[R*i*],#Val	
Con pre-base e indice			[R*i*,±R*j*,s]!	
Con post-base e indice			[R*i*],±R*j*,s	

Legenda:

Valore numero con segno (a 16 bit in NIOS II, a 8 o 32 bit in IA-32) rappresentato esplicitamente o da etichetta;

Val numero rappresentato in valore assoluto e segno a 9 bit nel modo immediato, a 13 bit nei modi assoluto e con spiazzamento;

LOC indirizzo assoluto (a 16 bit in NIOS II, a 32 bit in IA-32);

R, R_x uno degli otto registri generali IA-32, ma non si può usare il registro ESP (puntatore alla pila) come registro indice R_x; R*i*, R*j*, ISA ColdFire: registro A*i* o D*i* (rispettivamente A*j* o D*j*);

X spiazzamento: numero con segno (a 16 bit in NIOS II, a 8 o 32 bit in IA-32, ma solo a 32 bit nel modo con indice e spiazzamento);

S fattore di scala (IA-32): 1, 2, 4 o 8;

s scorrimento logico (ARM): ds #vs dove ds ∊ {LSL, LSR}: direzione dello scorrimento e vs: valore dello scorrimento (numero a 5 bit);

W Valore a 16 bit;

B Valore a 8 bit;

L Etichetta.

Instructions and Sequencing

- Instructions for a computer must support:
  - data transfers to and from the memory
  - arithmetic and logic operations on data
  - program sequencing and control
  - input/output transfers
- First consider data transfer & arithmetic/logic
- Control and input/output examined later
- Introduce notation to facilitate discussion

RISC and CISC Instruction Sets

- Nature of instructions distinguishes computer
- Two fundamentally different approaches
- Reduced Instruction Set Computers (RISC) have one-word instructions and require arithmetic operands to be in registers
- Complex Instruction Set Computers (CISC)
  have multi-word instructions and allow operands directly from memory

RISC Instruction Sets

- Focus on RISC first because it is simpler
- RISC instructions each occupy a single word
- A load/store architecture is used, meaning:
  - only Load and Store instructions are used to access memory operands
  - operands for arithmetic/logic instructions must be in registers, or one of them may be given explicitly in instruction word

RISC Instruction Sets

- Instructions/data are stored in the memory
- Processor register contents are initially invalid
- Because RISC requires register operands, data transfers are required before arithmetic
- The Load instruction is used for this purpose:
  Load *procr_register*, *mem_location*
- Addressing mode specifies memory location; different modes are discussed later

RISC Instruction Sets

- Consider high-level language statement:

  $$C = A + B$$

- A, B, and C correspond to memory locations
- RTN specification with these symbolic names:

  $$C \leftarrow [A] + [B]$$

- Steps: fetch contents of locations A and B, compute sum, and transfer result to location C

RISC Instruction Sets

- Sequence of simple RISC instructions for task:

```
Load    R2, A
Load    R3, B
Add     R4, R2, R3
Store   R4, C
```

- Load instruction transfers data to register
- Store instruction transfers data to the memory
- Destination differs with same operand order

A Program in the Memory

- Consider the preceding 4-instruction program
- How is it stored in the memory?
  (32-bit word length, byte-addressable)
- Place first RISC instruction word at address i
- Remaining instructions are at $i + 4$, $i + 8$, $i + 12$
- For now, assume that Load/Store instructions specify desired operand address directly;
  this issue is discussed in detail later

Address	Contents
Begin execution here → i	Load R2, A
$i + 4$	Load R3, B
$i + 8$	Add R4, R2, R3
$i + 12$	Store R4, C
	⋮
A	
	⋮
B	
	⋮
C	

4-instruction program segment

Data for the program

Instruction Execution/Sequencing

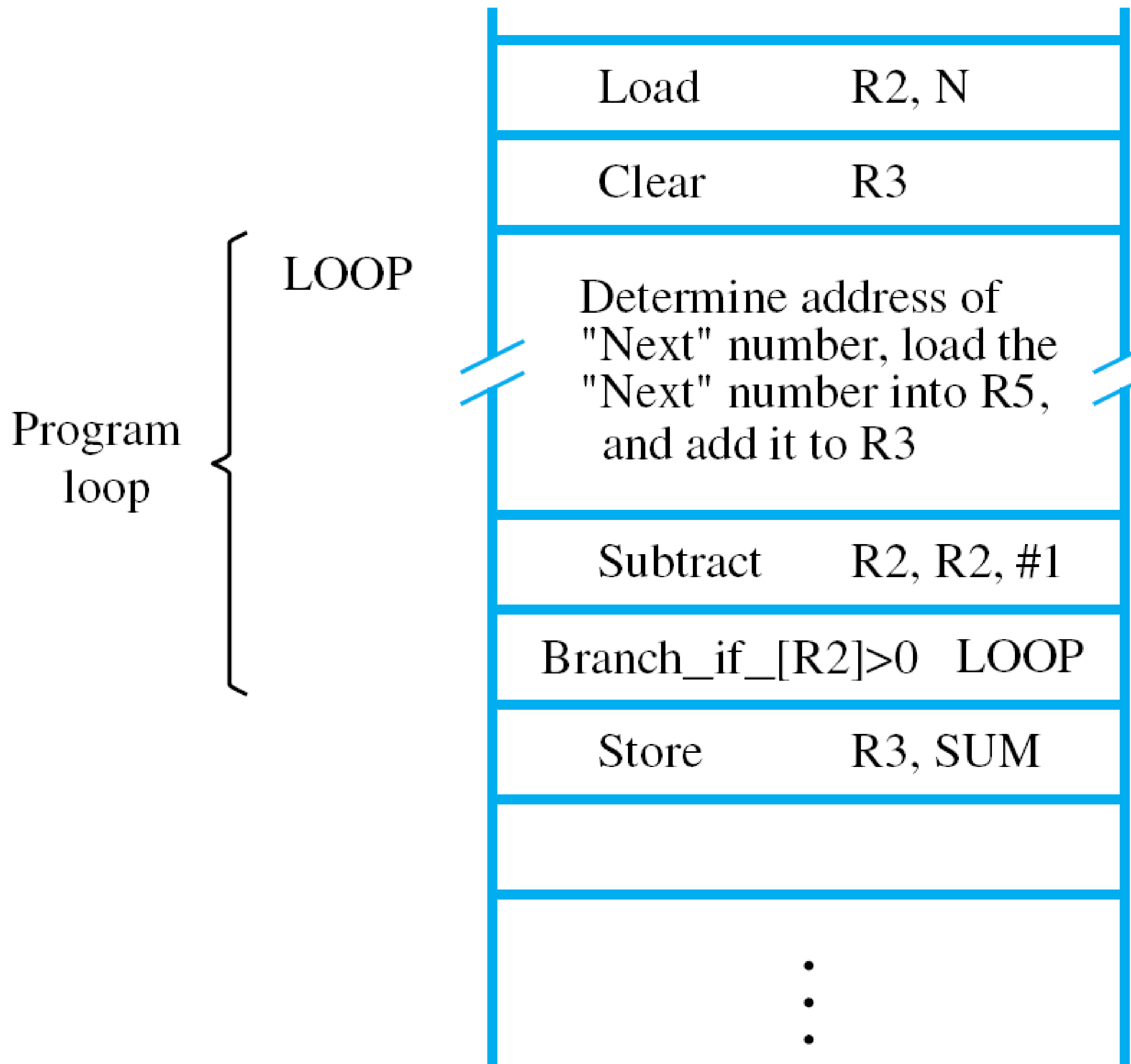
- How is the program executed?
- Processor has program counter (PC) register
- Address *i* for first instruction placed in PC
- Control circuits fetch and execute instructions, one after another $\rightarrow$ straight-line sequencing
- During execution of each instruction, PC register is incremented by 4
- PC contents are $i + 16$ after Store is executed

Details of Instruction Execution

- Two-phase procedure: fetch and execute
- Fetch involves *Read* operation using PC value
- Data placed in procr. instruction register (IR)
- To complete execution, control circuits examine encoded machine instruction in IR
- Specified operation is performed in steps, e.g., transfer operands, perform arithmetic
- Also, PC is incremented, ready for next fetch

Branching

- We can illustrate the concept of branching with a program that adds a list of numbers
- Same operations performed repeatedly, so the program contains a loop
- Loop body is straight-line instruction sequence
- It must determine address of next number, load value from the memory, and add to sum
- Branch instruction causes repetition of body


Load R2, N
Clear R3
LOOP
Program loop
Determine address of "Next" number, load the "Next" number into R5, and add it to R3
Subtract R2, R2, #1
Branch_if_[R2]>0 LOOP
Store R3, SUM

Branching

- Assume that size of list, *n*, stored at location N
- Use register R2 as a counter, initialized from N
- Body of loop includes the instruction

  Subtract R2, R2, #1

  to decrement counter in each loop pass
- Branch_if_[R2]>0 goes to branch target LOOP as long as contents of R2 are greater than zero
- Therefore, this is a conditional branch

Branching

- Branches that test a condition are used in loops and various other programming tasks
- One way to implement conditional branches is to compare contents of two registers, e.g.,

  Branch_if_[R4]>[R5]   LOOP
- In generic assembly language with mnemonics the same instruction might actually appear as

  BGT   R4, R5, LOOP

Generating Memory Addresses

- Loop must obtain next number in each pass
- Load instruction cannot contain full address since address size (32 bits) = instruction size
- Also, Load instruction itself would have to be modified in each pass to change address
- Instead, use register R*i* for address location
- An example of addressing modes (next topic)
- Initialize to NUM1, increment by 4 inside loop

Assembly Language

- Mnemonics (LD/ADD instead of Load/Add) used when programming specific computers
- The mnemonics represent the OP codes
- Assembly language is the set of mnemonics and rules for using them to write programs
- The rules constitute the language syntax
- Example: suffix 'I' to specify immediate mode

  ADDI   R2, R3, 5   (instead of #5)

Program Assembly & Execution

- From source program, assembler generates machine-language object program
- Assembler uses ORIGIN and other directives to determine address locations for code/data
- For branches, assembler computes ±offset from present address (in PC) to branch target
- Loader places object program in memory
- Debugger can be used to trace execution

Number Notation

- Decimal numbers used as immediate values:

```
ADDI    R2, R3, 93
```

- Assembler translates to binary representation
- Programmer may also specify binary numbers:

```
ADDI    R2, R3, %01011101
```

- Hexadecimal specification is also possible:

```
ADDI    R2, R3, 0x5D
```

- Note that $93 = 1011101_2 = 5D_{16}$

Logic Instructions

- AND, OR, and NOT operations on single bits are basic building blocks of digital circuits
- Similar operations in software on multiple bits
- Using RISC-style instructions, all operands are in registers or specified as immediate values:

```
Or      R4, R2, R3
And     R5, R6, #0xFF
```

- 16-bit immediate is zero-extended to 32 bits

Shift and Rotate Instructions

- Shifting binary value left/right = mult/div by 2
- Arithmetic shift preserves sign in MS bit
- Rotate copies bits from one end to other end
- Shift amount in register or given as immediate
- Carry flag (discussed later) may be involved
- Examples:

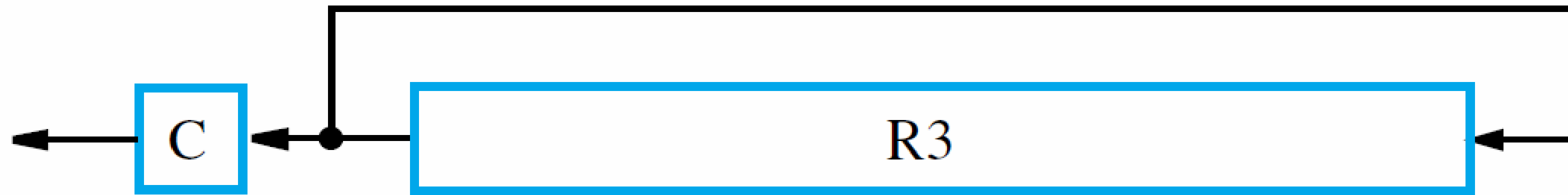
  LShiftL R3, R3, #2 (mult by 4)

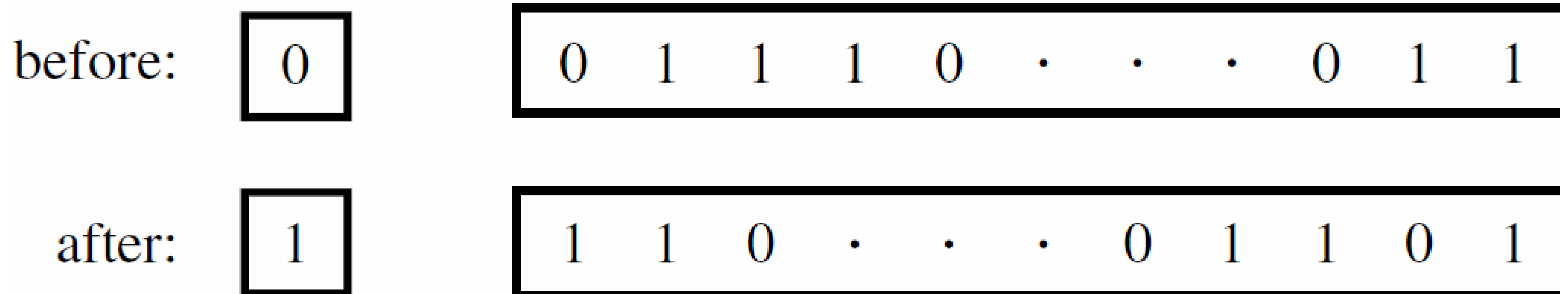
  RotateL R3, R3, #2 (MS bits to LS bits)

before: 0   0 1 1 1 0 · · · 0 1 1

after: 1   1 1 0 · · · 0 1 1 0 0

(a) Logical shift left LShiftL R3, R3, #2





(a) Rotate left without carry RotateL R3, R3, #2

Example Program: Digit Packing

- Illustrate shift, logic, byte-access instructions
- Memory has two binary-coded decimal digits
- Pointer set to 1st byte for index-mode access to load 1st digit, which is shifted to upper bits
- Upper bits of 2nd digit are cleared by ANDing
- ORing combines 2nd digit with shifted 1st digit for result of two packed digits in a single byte
- 32-bit registers, but only 8 lowest bits relevant

```
Move        R2, #LOC          R2 points to data.
LoadByte    R3, (R2)          Load first byte into R3.
LShiftL     R3, R3, #4        Shift left by 4 bit positions.
Add         R2, R2, #1        Increment the pointer.
LoadByte    R4, (R2)          Load second byte into R4.
And         R4, R4, #0xF      Clear high-order bits to zero.
Or          R3, R3, R4        Concatenate the BCD digits.
StoreByte   R3, PACKED        Store the result.
```

Multiplication and Division

- Signed integer multiplication of *n*-bit numbers produces a product with as many as 2*n* bits
- Processor truncates product to fit in a register:
  Multiply Rk, Ri, Rj (R$k \leftarrow$ [Ri] $\times$ [Rj])
- For general case, 2 registers may hold result
- Integer division produces quotient as result:
  Divide Rk, Ri, Rj (R$k \leftarrow$ [Ri] / [Rj])
- Remainder is discarded or placed in a register

32-bit Immediate Values

- To construct 32-bit immediates or addresses, use two instructions in sequence:

```
OrHigh R2, R0, #0x2000
Or     R2, R0, #0x4FF0
```

- Result is 0x20004FF0 in register R2
- Useful pseudoinstruction:

```
MoveImmediateAddress    R2, LOC
```

- Assembler can substitute OrHigh & Or

CISC Instruction Sets

- Not constrained to load/store architecture
- Instructions may be larger than one word
- Typically use two-operand instruction format, with at least one operand in a register
- Implementation of C = A + B using CISC:

Move R*i*, A
Add R*i*, B
Move C, R*i*

CISC Instruction Sets

- Move instruction equivalent to Load/Store
- But also can transfer immediate values and possibly between two memory locations
- Arithmetic instructions may employ addressing modes for operands in memory:

```
Subtract    LOC, Ri
Add         Rj, 16(Rk)
```

Additional Addressing Modes

- *Autodecrement* mode: before accessing operand, register contents are decremented, then new contents provide effective address
- Notation in assembly language:

  Add Rj, –(Ri)

- Use autoinc. & autodec. for stack operations:

  Move –(SP), NEWITEM (push)

  Move ITEM, (SP)+ (pop)

Condition Codes

- Processor can maintain information on results to affect subsequent conditional branches
- Results from arithmetic/comparison & Move
- Condition code flags in a status register:
  - N (negative) 1 if result negative, else 0
  - Z (zero) 1 if result zero, else 0
  - V (overflow) 1 if overflow occurs, else 0
  - C (carry) 1 if carry-out occurs, else 0

Branches using Condition Codes

- CISC branches check condition code flags
- For example, decrementing a register causes N and Z flags to be cleared if result is *not* zero
- A branch to check logic condition $N + Z = 0$:

  Branch>0 LOOP
- Other branches test conditions for $<, =, \neq, \leq, \geq$
- Also Branch_if_overflow and Branch_if_carry
- Consider CISC-style list-summing program

Sum of the Elements in an Array

```
        Move       R2, N       Load the size of the list.
        Clear      R3          Initialize sum to 0.
        Move       R4, #NUM1   Load address of the first number.
LOOP:   Add        R3, (R4)+   Add the next number to sum.
        Subtract   R2, #1      Decrement the counter.
        Branch>0   LOOP        Loop back if not finished.
        Move       SUM, R3     Store the final sum.
```

RISC and CISC Styles

- RISC characteristics include:
  simple addressing modes
  all instructions fitting in a single word
  fewer total instructions
  arithmetic/logic operations on registers
  load/store architecture for data transfers
  more instructions executed per program
- Simpler instructions make it easier to design faster hardware (e.g., use of pipelining)

RISC and CISC Styles

- CISC characteristics include:
  - more complex addressing modes
  - instructions spanning more than one word
  - more instructions for complex tasks
  - arithmetic/logic operations on memory
  - memory-to-memory data transfers
  - fewer instructions executed per program
- Complexity makes it somewhat more difficult to design fast hardware, but still possible

Assembler Directives

- Other information also needed to translate source program to object program
- How should symbolic names be interpreted?
- Where should instructions/data be placed?
- Assembler directives provide this information
- ORIGIN defines instruction/data start position
- RESERVE and DATAWORD define data storage
- EQU associates a name with a constant value

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Tabella A2.4 Direttive di assemblatore GAS, NIOS II, ColdFire, ARM e MASM

GAS	NIOS II	ColdFire	ARM	MASM
.org	.org	.org		ORG
.equ | =	.equ	.equ | =	EQU | =	EQU | =
.space|.skip	.skip	.space|ds.*t*	SPACE	D*t* *n* DUP(*v*)
.byte	.byte	.byte|dc.b	DCB	DB
.short|.hword|.word	.hword	.short|dc.w	DCW*a*	DW
.long|.int|.word	.word	.long|dc.l	DCD*a*	DD
.quad			DCQ*a*	DQ
.ascii	.ascii	.ascii	DCB	
.asciz|.string	.asciz	.asciz	DCB "*s*",0	
.data	.data	.data	AREA *s* DATA	.DATA
.text	.text	.text	AREA *s* CODE	.CODE
		entry	ENTRY	
		.textequ		TEXTEQU
.req			RN	
.title			TTL	TITLE
.end	.end		END	END *e*

Legenda:
t suffisso del codice mnemonico: tipo (dimensione) di ciascun elemento ColdFire: $t \in \{b,w,l\}$; MASM: $t \in \{B,W,D,Q\}$;
n numero di elementi;
v valore iniziale di ogni elemento, "?" se dati non inizializzati;
a suffisso del codice mnemonico: nessun allineamento se *a* = U, altrimenti *a*, assente e allineamento a indirizzo pari se DCW, multiplo di 4 se DCD o DCQ;
s stringa di caratteri ASCII (nella direttiva AREA: nome del segmento);
e etichetta (opzionale): indirizzo di inizio dell'esecuzione del programma.

ARM Instructions

Instructions

- Load and Store:

  LDR and STR for words

  LDRH and STRH for half words (zero-extended on a Load)

  LDRB and STRB for bytes (zero-extended on a Load)

  LDRSH and LDRSB are used for sign-extended Loads

  (Half words and bytes are positioned at the low-order end of a register)

Instructions

- Multiple-word Load and Store:
  Any subset of the processor registers can be loaded or stored with the Block Transfer instructions LDM and STM
  Example: LDMIA R10!, [R0, R1, R6, R7]

  If [R10] = 1000, words at 1000, 1004, 1008, and 1012 are loaded into the registers, and R10 contains 1016 after all transfers

Instructions

- Arithmetic:
  Assembly language format is
  OP R*d*, R*n*, R*m* or #offset

  ADD R0, R2, R4
  performs
  R0 ← [R2] + [R4]

  SUB R0, R3, #17
  performs
  R0 ← [R3] – 17

  (immediates are unsigned values in the range 0 to 255)

Instructions

- Arithmetic: The second source operand can be shifted or rotated before being used

  ADD R0, R1, R5, LSL #4

  performs

  $R0 \leftarrow [R1] + 16 \times [R5]$

  Shifts and rotations available:

  LSL Logical shift left

  LSR Logical shift right

  ASR Arithmetic shift right

  ROR Rotate right

Instructions

- Shifting/rotation of the second source operand in arithmetic instructions:

  The last bit shifted (or rotated) out is written into the C flag
  A second rotation operation, labelled RRX (Rotate right extended), includes the C flag in the bits being rotated; only rotates by 1 bit
  (If the second source operand is an immediate value, a limited form of rotation is provided)

Instructions

- Arithmetic:

  MUL R0, R1, R2

  performs

  $R0 \leftarrow [R1] \times [R2]$

  The low-order 32 bits of the 64-bit product are written into R0
  For 2’s-complement numbers, the value in R0 is correct if the product fits into 32 bits

Instructions

- Arithmetic:

  MLA R0, R4, R5, R6

  performs

  $R0 \leftarrow ([R4] \times [R5]) + [R6]$

  This Multiply-Accumulate instruction is useful in signal-processing applications

  Other versions of MUL and MLA generate 64-bit products

Instructions

- Move:

  MOV Rd, Rm

  performs

  $Rd \leftarrow [Rm]$

  MOV Rd, #value

  performs

  $Rd \leftarrow value$

  (The second operand can be shifted/rotated)

Instructions

- Move:

  MVN R*d*, R*m* or #value

  loads the bit-complement of [R*m*] or value into R*d*

Instructions

- Implementing Shift and Rotate instructions:

  MOV R*i*, R*j*, LSL #4

  achieves the same result as the generic instruction:

  LShiftL R*i*, R*j*, #4

Instructions

- Logic:

  AND Rd, Rn, Rm

  performs the bit-wise logical AND of the operands in registers R*n* and R*m* and writes the result into register R*d*

  ORR (bit-wise logical OR)
  EOR (bit-wise logical XOR)

  are also provided

Instructions

- Logic:

  The Bit Clear instruction, BIC, is closely related to the AND instruction

  The bits of R*m* are complemented before they are ANDed with the bits of R*n*

  If R0 contains the hexadecimal pattern 02FA62CA, and R1 contains 0000FFFF,

  ```
  BIC     R0, R0, R1
  ```

  results in 02FA0000 being written into R0

Instructions

- Test:

  TSTR*n*, R*m* or #value

  performs bit-wise logical AND of the two operands, then sets condition code flags

  TSTR3, #1

  sets Z = 1 if low-order bit of R3 is 0

  sets Z = 0 if low-order bit of R3 is 1

  (useful for checking status bits in I/O devices)

Instructions

- Test:

  TEQ Rn, Rm or #value

  performs bit-wise logical XOR of the two operands, then sets condition code flags

  TEQ R2, #5

  sets Z = 1 if R2 contains 5

  sets Z = 0 otherwise

Instructions

- Compare:

  CMP R*n*, R*m*
  performs
  [R*n*] – [R*m*]
  and updates condition code flags based on the result

Instructions

- Setting condition code flags

CMP, TST, and TEQ, always update the condition code flags

Arithmetic, Logic, and Move instructions do so only if S is appended to the OP code

ADDS updates flags, but ADD does not

Instructions

- Adding 64-bit operands

```
 ADC         R0, R1, R2      (Add with carry)
performs     R0 ← [R1] + [R2] + [C]

If pairs R3,R2 and R5,R4 hold 64-bit operands,
 ADDS     R6, R2, R4
 ADC      R7, R3, R5
writes their sum into register pair R7,R6
```

Instructions

- Branch:

  B{condition}   LOCATION

  branches to LOCATION if the settings of the condition code flags satisfy {condition}

  BEQ   LOCATION
  branches if $Z = 1$

1000

BEQ LOCATION

1004

updated [PC] = 1008

Offset = 92

LOCATION = 1100

Branch target instruction

Condition field encoding in ARM instructions.

Condition field $b_{31} \ldots b_{28}$	Condition suffix	Name	Condition code test
0 0 0 0	EQ	Equal (zero)	$Z = 1$
0 0 0 1	NE	Not equal (nonzero)	$Z = 0$
0 0 1 0	CS/HS	Carry set/Unsigned higher or same	$C = 1$
0 0 1 1	CC/LO	Carry clear/Unsigned lower	$C = 0$
0 1 0 0	MI	Minus (negative)	$N = 1$
0 1 0 1	PL	Plus (positive or zero)	$N = 0$
0 1 1 0	VS	Overflow	$V = 1$
0 1 1 1	VC	No overflow	$V = 0$
1 0 0 0	HI	Unsigned higher	$\overline{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
1 0 1 0	GE	Signed greater than or equal	$N \oplus V = 0$
1 0 1 1	LT	Signed less than	$N \oplus V = 1$
1 1 0 0	GT	Signed greater than	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	Always	
1 1 1 1		not used	

Program

- An assembly-language program for adding numbers stored in the memory is shown in the next slide

  The instruction
  LDR R2, =NUM1
  is a pseudoinstruction that loads the 32-bit address value NUM1 into R2
  It is implemented using actual instructions

Sum the Elements in an Array

```
        LDR   R1, N          Load count into R1.
        LDR   R2, =NUM1      Load address NUM1 into R2.
        MOV   R0, #0         Clear accumulator R0.
LOOP    LDR   R3, [R2], #4   Load next number into R3.
        ADD   R0, R0, R3     Add number into R0.
        SUBS  R1, R1, #1     Decrement loop counter R1.
        BGT   LOOP           Branch back if not done.
        STR   R0, SUM        Store sum.
```

Assembly language

- An assembly language program for adding numbers is given in the next slide
- Comments:

1. The AREA directive specifies the start of instruction (CODE) and data (DATA) areas

2. The ENTRY directive specifies the start point for program execution

	Memory address label	Operation	Addressing or data information
Assembler directives		AREA	CODE
		ENTRY	
Statements that generate machine instructions		LDR	R1, N
		LDR	R2, POINTER
		MOV	R0, #0
	LOOP	LDR	R3, [R2], #4
		ADD	R0, R0, R3
		SUBS	R1, R1, #1
		BGT	LOOP
		STR	R0, SUM
Assembler directives		AREA	DATA
	SUM	DCD	0
	N	DCD	5
	POINTER	DCD	NUM1
	NUM1	DCD	3, –17, 27, –12, 322
		END	

Assembly language

- Comments (continued)

3. The combination of the instruction

```
        LDR     R2, POINTER
```

and the data declaration

```
POINTER    DCD    NUM1
```

implements the pseudoinstruction

```
        LDR    R2, =NUM1
```

Pseudoinstructions

- Operations specified by pseudoinstructions are implemented with actual machine instructions by the assembler
- Example: An immediate is an 8-bit unsigned value
  The pseudoinstruction
  ```
  MOV     R0, #–5
  ```
  is implemented with the actual instruction
  ```
  MVN     R0, #4
  ```
  (the bit-complement of 4 = 00000100
  –5 = 11111011)

Pseudoinstructions

- Loading 32-bit values:

  The pseudoinstruction
  LDR   R*d*, =value
  loads a 32-bit value into R*d*

  LDR   R3, =127
  is implemented with
  MOV   R3, #127

  (used for “short” values)

Pseudoinstructions

- Loading 32-bit values:

```
        LDR     R3, =&A123B456
is implemented with
        LDR     R3, MEMLOC    (instruction)
   MEMLOC  DCD     &A123B456      (data)
```

(used for “long” values, including addresses)

Pseudoinstructions

- Loading 32-bit address label values:
  If the address is “close” to the current value of the program counter (R15), the ADR pseudoinstruction can be used

  ```
  ADR     Rd, LOCATION
  ```
  is implemented with
  ```
  ADD     Rd, R15, #offset,     or
  SUB     Rd, R15, #offset
  ```
  (offset is calculated by the assembler)

Esempio di programma ARM

Si consideri il seguente codice (Algoritmo di Euclide per il Massimo Comun Divisore):

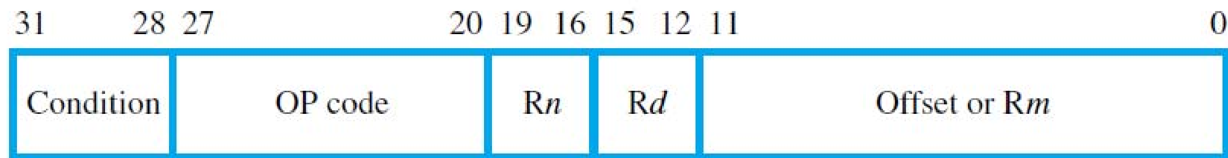
```
function gcd (integer a, integer b): result is integer
    while (a<>b) do
     if (a > b) then
          a = a - b
     else
          b = b - a
     endif
    endwhile
    result = a
```

Assembly ARM

```
gcd
 CMP    r0,r1
 BEQ    end
 BLT    less
 SUB    r0,r0,r1
 BAL    gcd
less
 SUB    r1,r1,r0
 BAL    gcd
end
```

Conditional Execution

- Almost every ARM instruction can be executed conditionally on the state of the ALU status flags
- The instructions that can be conditional have an optional condition code



- The conditioned instruction is only executed if the condition code flags meet the specified condition
- Example:

```
ADD r0, r1, r2    ; r0 = r1 + r2, don't update flags
ADDS r0, r1, r2   ; r0 = r1 + r2, and update flags
ADDSCS r0, r1, r2 ; If C flag set then r0 = r1 + r2, and
; update flags
```

Assembly ARM with conditioned instructions

Same algorithm as before

```
gcd
 CMP r0,r1
 SUBGT r0,r0,r1
 SUBLT r1,r1,r0
 BNE gcd
```

Conditional Vs Non Conditional

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	BEQ end	1 (not executed)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1 (not executed)
1	1	BNE gcd	1 (not executed)
			Total = 10

case where r0 equals 1 and r1 equals 2

Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

COLDFIRE INSTRUCTIONS

Instructions

- One, two, or three consecutive words
- *OP-code* word is first – it specifies operation
- Also provides some addressing information; one or two *extension* words provide more
- Most arithmetic and data-transfer instructions have source/destination operands:

  OP src, dst

- .L, W., or .B suffix for OP code specifies size

MOVE Instruction

- Used to perform transfers between memory, I/O interfaces, and registers
- Value being transferred affects N and Z flags in status register
- Byte, word, and longword sizes are permitted
- All addressing modes valid for source operand
- Some modes not permitted for destination
- Some source/dest. mode pairings not valid

Address	Contents	
i	OP-code word	
$i + 2$	002A	Upper 16 bits of immediate value
$i + 4$	4C80	Lower 16 bits of immediate value
$i + 6$		

The instruction MOVE.L #$2A4C80 in memory.

Arithmetic Instructions

- Most permit only longword size,
  and most require at least one register operand
- Addition, subtraction, comparison, negation:
  ADD.L, SUB.L, CMP.L, NEG.L
- ADDI.L, ADDQ.L for immediate operands
- ADDA.L, SUBA.L, CMPA.L for address registers
- Arithmetic operations affect condition codes
- ADDX.L, SUBX.L, NEGX.L for numbers > 32 bits

```
MOVE.L   #$A72C10F8, D2   D2 contains A72C10F8.
MOVE.L   #$10, D3         D3 contains 10.
MOVE.L   #$5C00FE04, D4   D4 contains 5C00FE04.
MOVE.L   #$4A, D5         D5 contains 4A.
ADD.L    D2, D4           Add low-order 32 bits; carry-out sets X and C flags.
ADDX.L   D3, D5           Add high-order bits with X flag as carry-in bit.
```

Program to add numbers larger than 32 bits using the ADDX instruction.

Multiplication and Division

- Signed/unsigned multiply of 16-bit numbers, or 32-bit numbers (where result is truncated)
- N and Z flags affected; V and C flags cleared
- Signed/unsigned division where divisor is either 16 or 32 bits (dividend is always 32 bits)
- For 16-bit divisor, remainder placed in register, but for 32-bit divisor, remainder is discarded
- Special instruction gives remainder separately

```
MOVE.W   #$FFFF, D2   The low-order word of D2 is treated as –1.
MOVE.W   #$0001, D3   The low-order word of D3 contains 1.
MULS.W   D2, D3       The signed longword result in D3 is –1
                        or $FFFFFFFF, hence the N flag is set.
```

(a) Signed computation of $-1 \times 1 = -1$

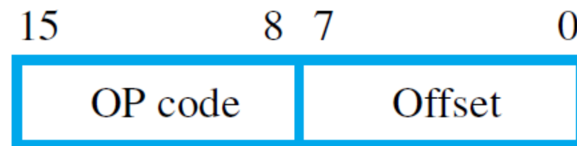
```
MOVE.W   #$FFFF, D2   The low-order word of D2 is treated as 65535.
MOVE.W   #$0001, D3   The low-order word of D3 contains 1.
MULU.W   D2, D3       The unsigned longword result in D3 is 65535
                        or $0000FFFF, hence the N flag is cleared.
```

(b) Unsigned computation of $65535 \times 1 = 65535$

Branch and Jump Instructions

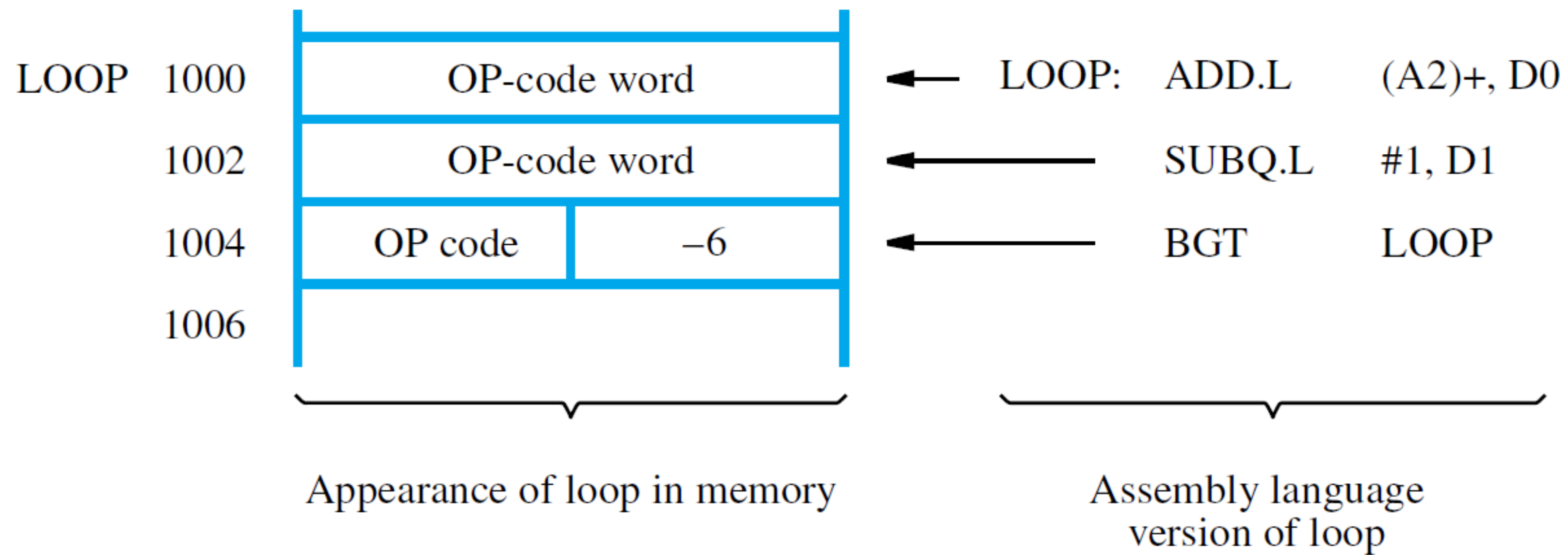
- Conditional branches test combinations of condition code flags; e.g., BEQ checks if Z=1
- BRA is unconditional branch
- Target address for branch computed by adding signed offset to program counter value
- Offset is part of branch instruction encoding
- May be 8, 16, or 32 bits in size
- Unconditional JMP instruction can specify target address with an addressing modes

Condition suffix ***cc***	**Name**	**Test condition**
HI	High	$C \vee Z = 0$
LS	Low or same	$C \vee Z = 1$
CC	Carry clear	$C = 0$
CS	Carry set	$C = 1$
NE	Not equal	$Z = 0$
EQ	Equal	$Z = 1$
VC	Overflow clear	$V = 0$
VS	Overflow set	$V = 1$
PL	Plus	$N = 0$
MI	Minus	$N = 1$
GE	Greater or equal	$N \oplus V = 0$
LT	Less than	$N \oplus V = 1$
GT	Greater than	$Z \vee (N \oplus V) = 0$
LE	Less or equal	$Z \vee (N \oplus V) = 1$



Branch address = [updated PC] + offset

(a) Short-offset branch instruction format



[PC] = 1006 when branch address is computed

Branch address = 1006 – 6 = 1000

(b) Example of using a branch instruction

```
        MOVEA.L   #NUM1, A2     Put the address NUM1 in A2.
        MOVE.L    N, D1         Put the number of entries n in D1.
        CLR.L     D0
LOOP:   ADD.L     (A2)+, D0     Accumulate sum in D0.
        SUBQ.L    #1, D1
        BGT       LOOP
        MOVE.L    D0, SUM       Store the result when finished.
```

```
        MOVEA.L   #LIST, A2     Get the address LIST.
        CLR.L     D3
        CLR.L     D4
        CLR.L     D5
        MOVE.L    N, D6         Load the value n.
LOOP:   ADD.L     4(A2), D3     Add current student mark for Test 1.
        ADD.L     8(A2), D4     Add current student mark for Test 2.
        ADD.L     12(A2), D5    Add current student mark for Test 3.
        ADDA.L    #16, A2       Increment the pointer.
        SUBQ.L    #1, D6        Decrement the counter.
        BGT       LOOP          Loop back if not finished.
        MOVE.L    D3, SUM1      Store the total for Test 1.
        MOVE.L    D4, SUM2      Store the total for Test 2.
        MOVE.L    D5, SUM3      Store the total for Test 3.
```

Logic and Shift Instructions

- AND.L, OR.L, EOR.L, NOT.L instructions
  require at least one data register operand
- ANDI.L, ORI.L, EORI.L – immediate src operand
- N and Z flags affected;   V and C flags cleared

- LSL.L, LSR.L, ASL.L, ASR.L instructions require immediate or data register for shift amount
- Operand to be shift is in a data register

```
MOVEA.L   #LOC, A0      A0 points to two consecutive bytes.
MOVE.B    (A0)+, D0     Load first byte into D0.
LSL.L     #4, D0        Shift left by 4 bit positions.
MOVE.B    (A0), D1      Load second byte into D1.
ANDI.L    #$F, D1       Clear all high-order bits in D1.
OR.L      D0, D1        Concatenate the digits.
MOVE.B    D1, PACKED    Store the result.
```

Tabella A2.7a Istruzioni NIOS II, ColdFire, ARM e IA-32

Tipo di istruzioni	NIOS II	ColdFire	ARM	IA-32
(a) Istruzioni di trasferimento, di controllo				
Trasferimento				
Load	ld*b* r*i*, X(r*j*)		LDR*b*	
Store	st*b* r*i*, X(r*j*)		STR*b*	
Move	mov*a*	MOVE*a*.*b*	MOV, MVN	MOV, LEA, PUSH, POP
Multiplo		MOVEM.*b*	LDM*w*, STM*w*	POPAD, PUSHAD
Controllo				
Salto incondiz.	br *l*, jmp r*i*	JMP	B *l*	JMP
Salto condiz.	bc r*i*, r*j*, *l*	Bc *l*	Bc *l*	Jc *l*
				LOOP *l*
Chiamata e rientro: si veda Tabella A2.5				

Legenda:

b suffisso del codice operativo, dimensione del dato in memoria, estensione a 32 bit: NIOS II: $b \in$ {w,b,h,bu,hu}, ColdFire: $b \in$ {B,W,L}, ARM: $b \in$ {B,H,SB,SH} opzionale;

a suffisso opzionale del codice operativo, modo di indirizzamento: NIOS II: $a \in$ {i,ui,ia}, indirizzamento immediato, ColdFire: $a \in$ {A,Q}, se la dest. è un registro indirizzo o dati, rispettivamente;

w suffisso del codice operativo, progressione del registro di base, $w \in$ {IA,DA,IB,DB}; *l*, etichetta; *c*, suffisso del codice operativo, condizione aritmetico-logica di salto: NIOS II: $c \in$ {eq,ne,ge,geu,gt,gtu,le,leu,lt,ltu}, ColdFire, ARM, IA-32: si veda Tabella A2.8

Tabella A2.7b Istruzioni NIOS II, ColdFire, ARM e IA-32

Tipo di istruzioni	NIOS II	ColdFire	ARM	IA-32
(b) Istruzioni aritmetiche, di confronto, logiche, di scorrimento				
Aritmetiche				
Addizione	add*m*	ADD*m*.L	ADD*f*, ADC	ADD, ADC
Sottrazione	sub*m*	SUB*m*.L	SUB*f*, SBC	SUB, SBB
		NEG*m*.L		NEG
Moltiplicazione	mul*m*	MUL*s*.*b*	MUL, MLA	IMUL
Divisione	div, divu	DIV*s*.*b*		IDIV
Resto		REM*s*.L		
Altre		EXT.*b*		INC, DEC
		CLR.*b*		
Confronto	cmp*cm*	CMP*m*.L	CMP, CMN	CMP
Logiche				
Congiunzione	and*m*, andhi	AND*m*.L	AND, TST	AND
Disgiunzione	or*m*, orhi	OR*m*.L	ORR	OR
Disg. esclusiva	xor*m*, xorhi	EOR*m*.L	EOR, TEQ	XOR
Negazione		NOT*m*.L		NOT
Altre	nor		BIC	
Scorrimento				
Logico	srl*m*	LSR.L	*t*, LSR	SHR
	sll*m*	LSL.L	*t*, LSL	SHL
Aritmetico	sra*m*	ASR.L	*t*, ASR	SAR
		ASL.L		SAL
Rotazione	ror		*t*, ROR	ROR, RCR
	rol*m*			ROL, RCL

Legenda:

m suffisso opzionale del codice operativo, NIOS II: $m \in \{i\}$, secondo operando sorgente immediato, ColdFire: $m \in \{I,A,X\}$, sorgente immediato (sola opzione nelle istruzioni logiche), o dest. registro indirizzo, o riporto in ingresso da bit di esito X (sola opzione in NEG, esclusa in CMP);

f suffisso opzionale del codice operativo, imposta i bit di esito C, V se f = S; *s*, suffisso del codice operativo, operandi con o senza segno, $s \in \{S,U\}$;

b suffisso del codice operativo, dimensione dell'operando sorgente o di destinazione; ColdFire: $b \in \{W,L\}$ in MUL, DIV; $b \in \{B,W,L\}$ in EXT, CLR;

c suffisso del codice operativo, condizione di confronto (come nelle istruzioni di salto); *t*, istruzione MOV o ADD, lo scorrimento si applica al secondo operando sorgente.

FITNESS

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

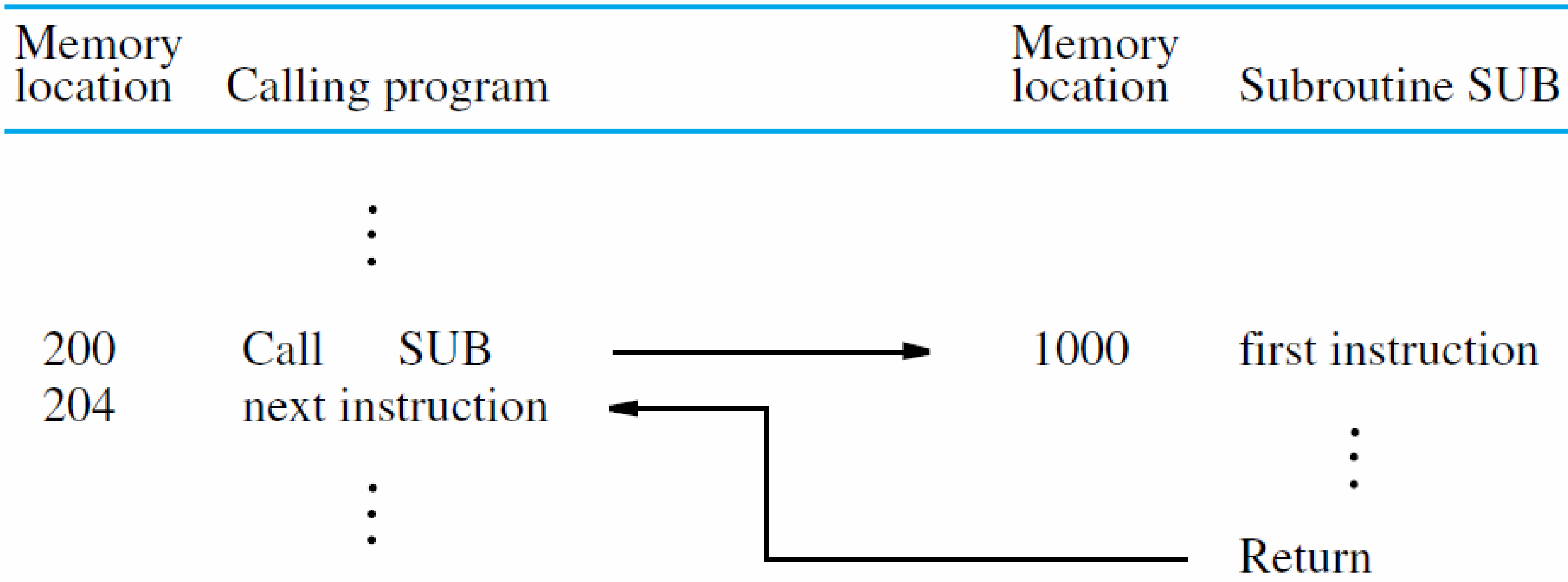
- In a given program, a particular task may be executed many times using different data
- Examples: mathematical function, list sorting
- Implement task in one block of instructions
- This is called a subroutine
- Rather than reproduce entire subroutine block in each part of program, use a subroutine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

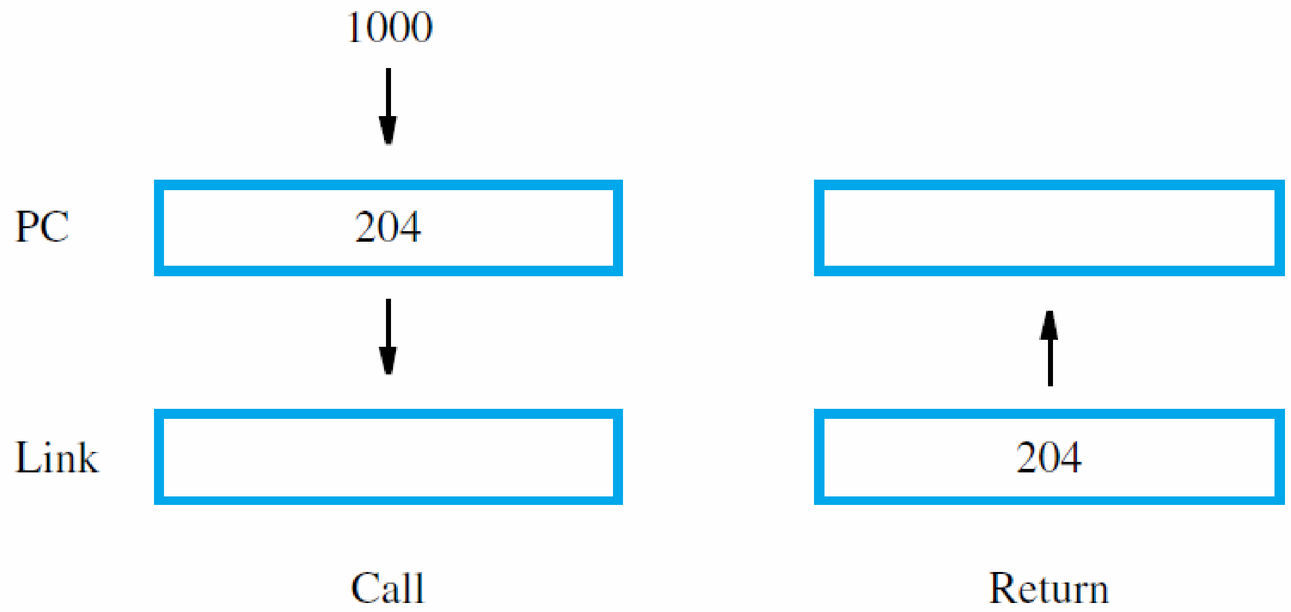
Subroutines

- 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

FITNESS

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

FP
(frame pointer)

saved [FP]

param1

param2

param3

param4

Stack
frame
for
called
subroutine

Old TOS