

Instruction Set Architecture

Corso di
Architettura dei Sistemi a Microprocessore

Luigi Coppolino

Dipartimento di Ingegneria

Università degli Studi di Napoli “Parthenope”



Fault and Intrusion Tolerant NEtworked Systems

The Fault and Intrusion Tolerant NEtworked SystemS (FITNESS) Research Group
<http://www.dit.uniparthenope.it/FITNESS/>



Contact info

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento per le Tecnologie

Centro Direzionale di Napoli, Isola C4
V Piano lato SUD - Stanza n. 512

Tel: +39-081-5476702
Fax: +39-081-5476777

Roadmap

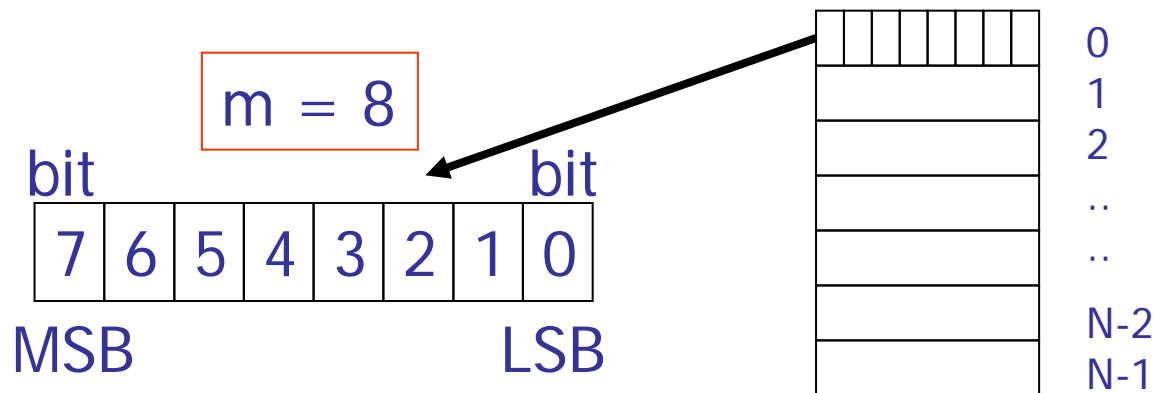
- Organizzazione della memoria
- Esempi di organizzazione della memoria
- I principali modi di indirizzamento
- Esempi d'impiego
- Riepilogo modi base
- Altri modi di indirizzamento
- Istruzioni ARM E COLDFIRE
- Uso dello Stack e chiamata di funzioni

Sources

- Textbook (chapter 2)
- Manuale Freescale
(http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf) (http://www.freescale.com/files/dsp/doc/ref_manual/C0PRM.pdf)
- Manuale ARM
(http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/DUI0204J_rvct_assembler_guide.pdf)
- Quick Guides:
 - ARM:
http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001I/QRC0001_UAL.pdf
 - Coldfire (m68000):
<http://home.anadolu.edu.tr/~sgorgulu/micro2/2008/68KISx1.pdf>

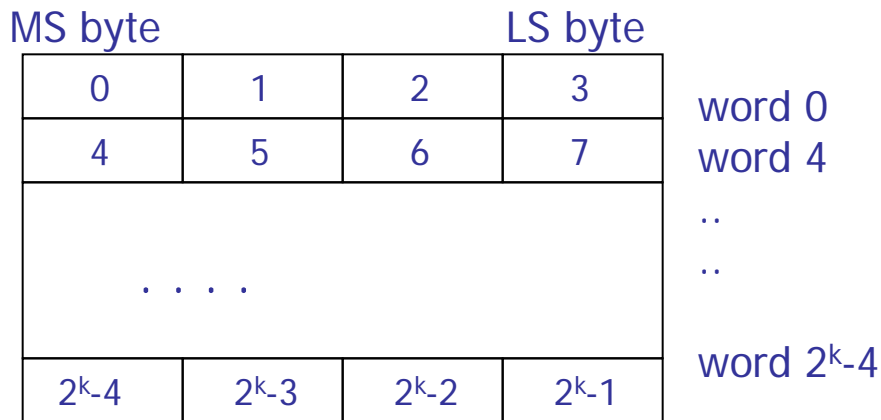
The main memory (central memory)

- The main memory of a computer is organized as an array of sequences of m bits, each of them is said *word* ($m = \text{WORD LENGTH (LUNGHEZZA DI PAROLA)} \Rightarrow$ typical values for m are 16, 32, or 64)
- Any read or write operation from/to the main memory accesses to a whole word
- Each word has an *address*, that is an integer number between 0 and $N-1$ (ADDRESS SPACE – SPAZIO DI INDIRIZZAMENTO), being $N = 2^c$

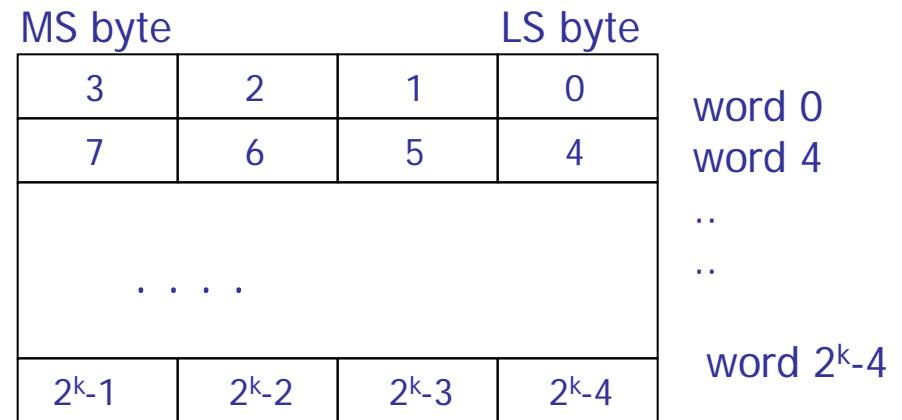


Memory structure

- A word is typically composed of 16 bit, 32 bit, or 64 bit
 - It is impractical to give an address to each bit
- Memory is typically *byte-addressable*, the smallest unit of memory that can be referenced is a byte
- Since a word is composed of four byte, addresses of bytes within a word can be ordered in two alternative ways: **big endian** e **little endian**



BIG-ENDIAN ordering

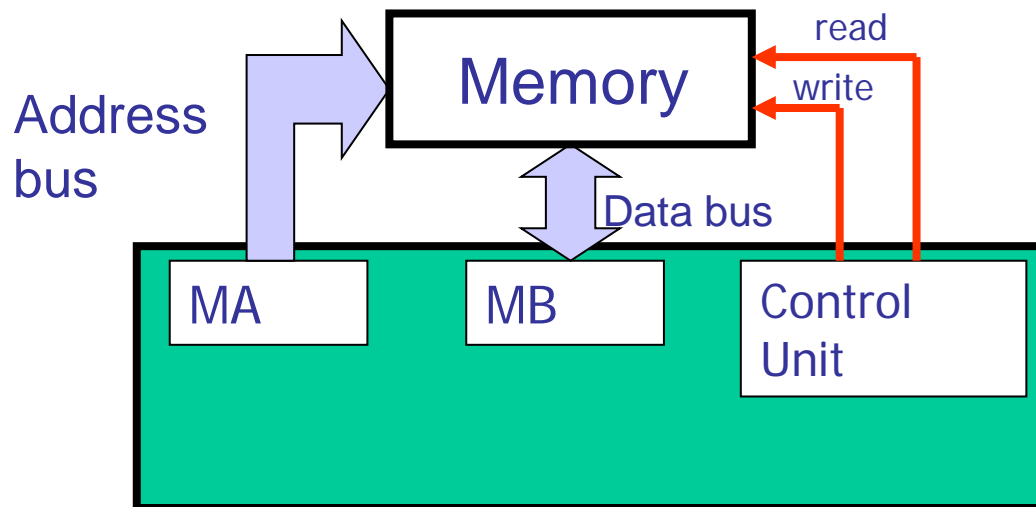


LITTLE-ENDIAN ordering

Address and Register size

- Address size:
 - Number of bits composing an address
 - If the Address size is m the address space of the memory is 2^m
- Register size:
 - Number of bits composing a register
- Typically Register Size is a multiple (or equal) of Memory Size

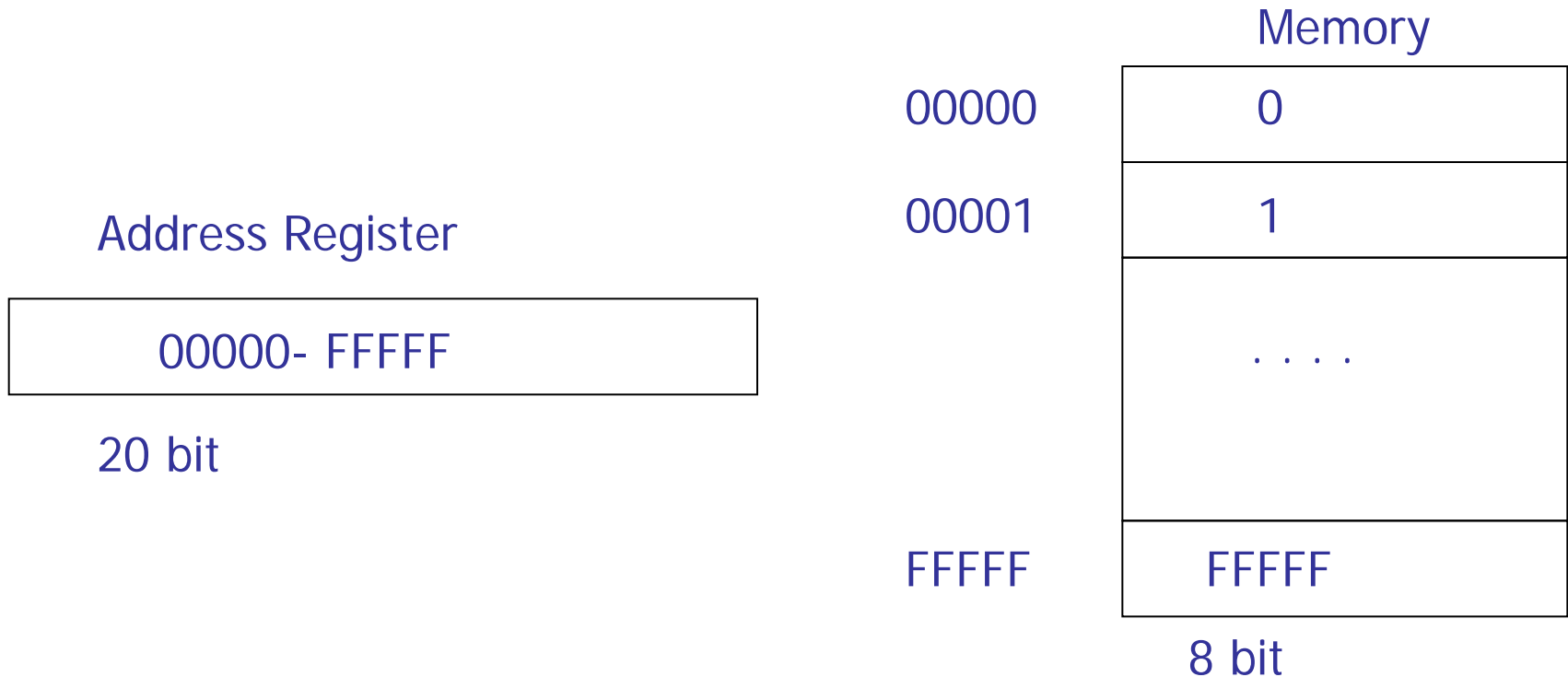
- Not all the MA register bits must be connected to the bus, thus:
 - Logical address space can be lower than physical address space => “aliasing”



Example

- Draw the schema of a memory architecture with the following characteristics:
 - Logical address space: 1MB
 - Physical address space: 1MB
 - Word length: 1 byte
 - Accessibility: byte addressable

Solution

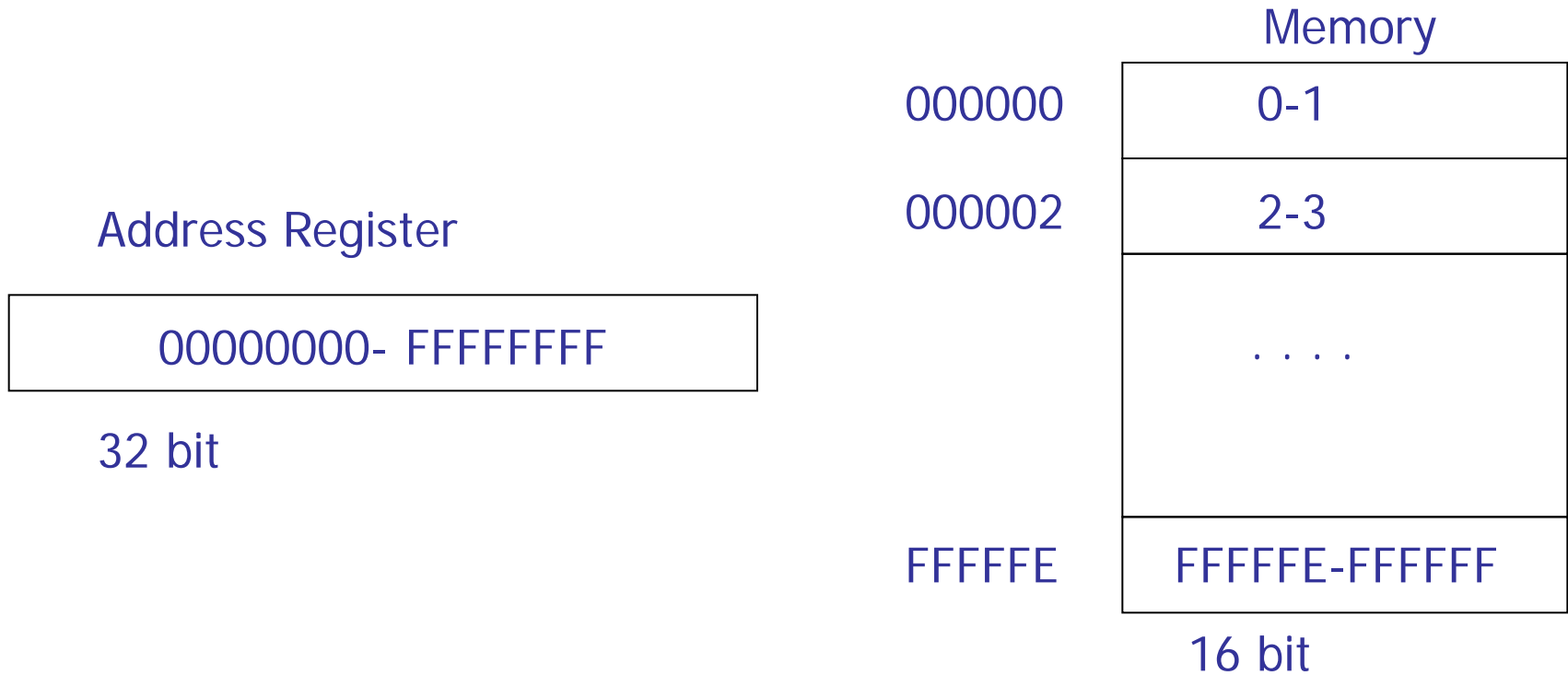


- The MC68008 is organised in this way

Example

- Draw the schema of a memory architecture with the following characteristics:
 - Logical address space: 4GB
 - Physical address space: 16MB
 - Word length: 2 byte
 - Accessibility: byte addressable

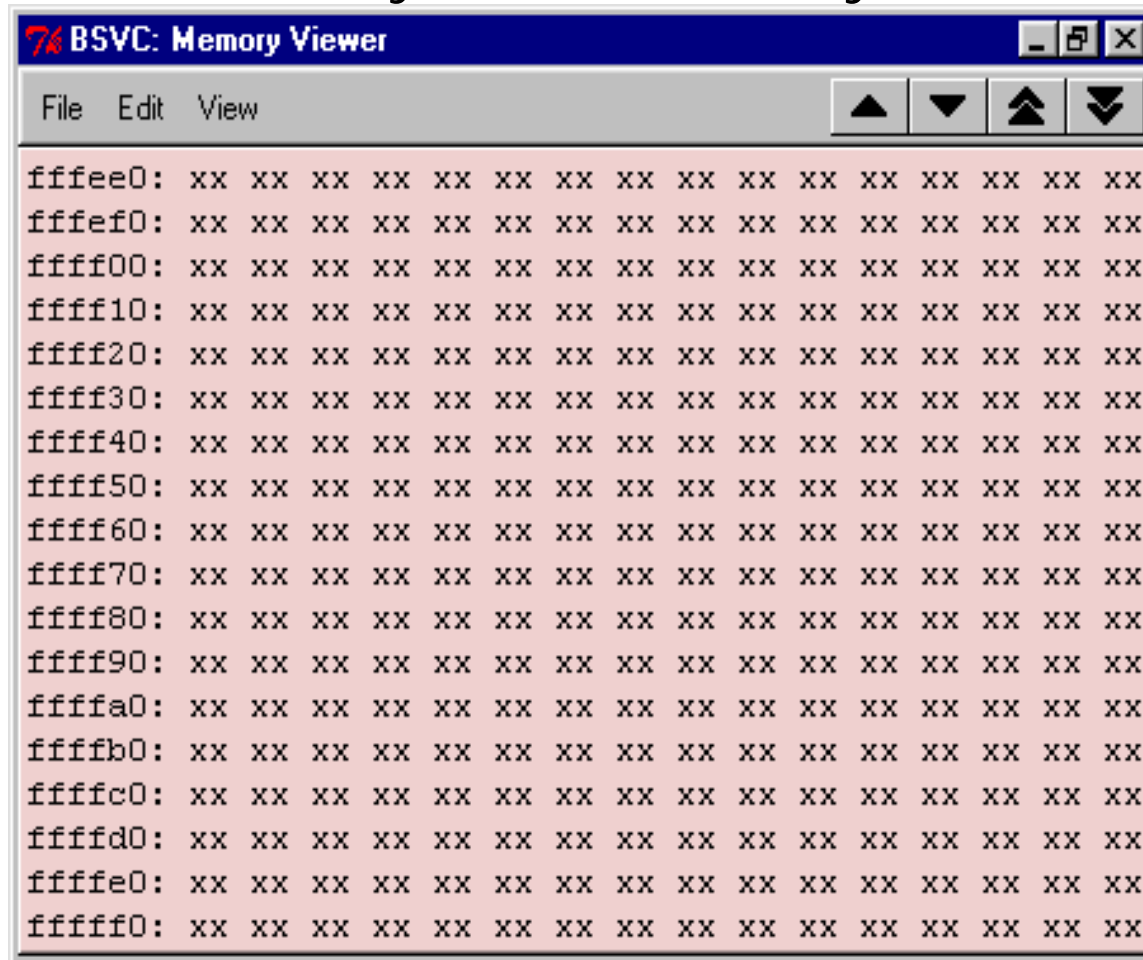
Solution



- The MC68000 and MC68010 are organized in this way

BSVC memory

- The MC68000 memory view offered by a simulator



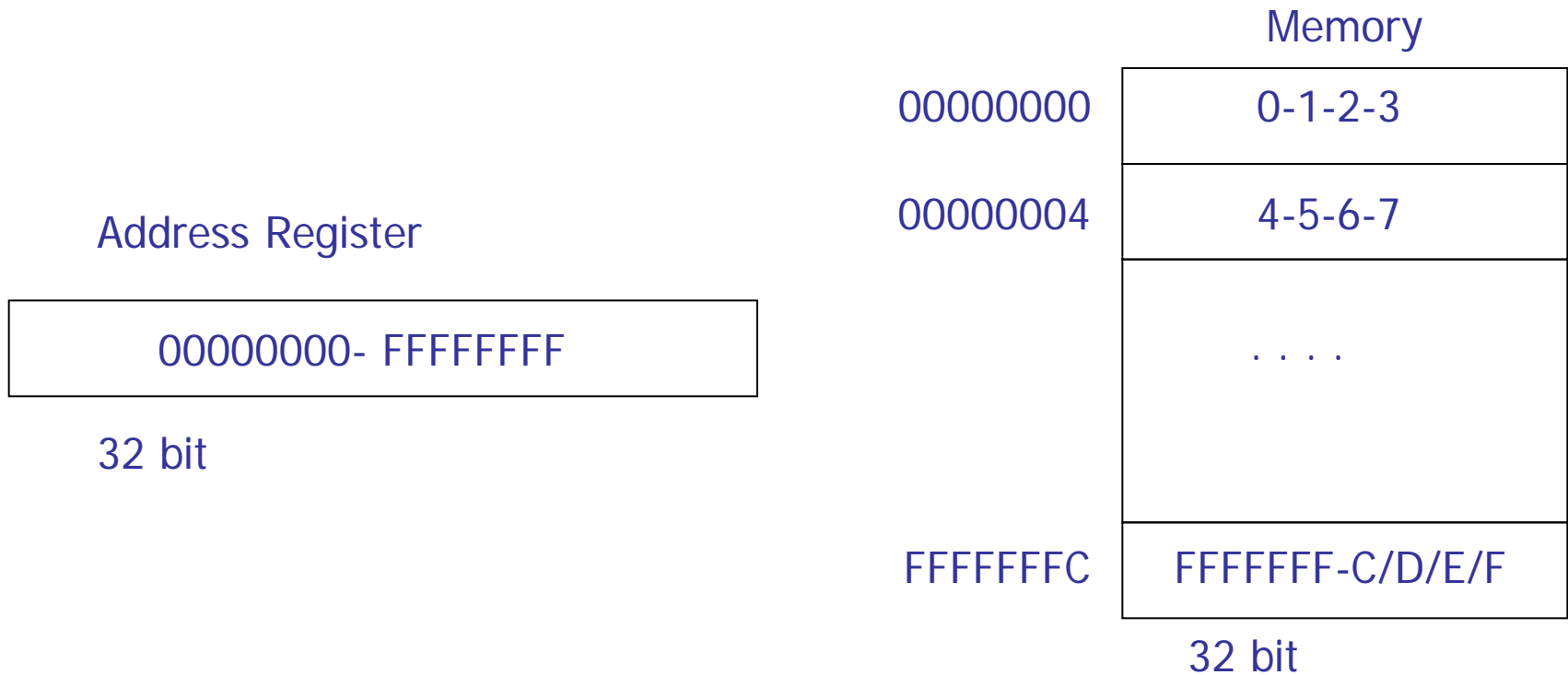
The screenshot shows a window titled "BSVC: Memory Viewer" with a menu bar containing "File", "Edit", and "View". Below the menu bar are four navigation buttons: a left arrow, a right arrow, an up arrow, and a down arrow. The main area of the window displays a memory dump with 17 rows of hexadecimal addresses and their corresponding data. Each row consists of a 16-bit hexadecimal address followed by 16 pairs of "xx" characters, representing 16 bytes of memory.

```
ffffe0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
fffff0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff00: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff10: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff20: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff30: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff40: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff50: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff60: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff70: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff80: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffff90: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffffa0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffffb0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffffc0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffffd0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
ffffe0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
fffff0: xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
```

Example

- Draw the schema of a memory architecture with the following characteristics:
 - Logical address space: 4GB
 - Physical address space: 4GB
 - Word length: 4 byte
 - Accessibility: byte addressable

Soluzione



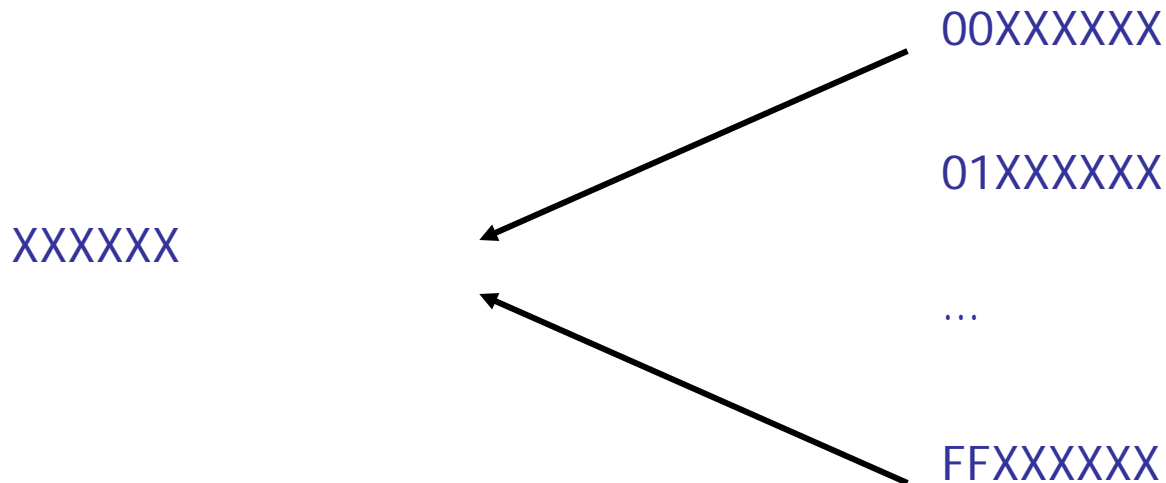
- MC68020 and following architectures are organised this way

Example

- The MC68000 memory architecture has the following characteristics:
 - Logical address space: 4GB
 - Physical address space: 16MB
- The MC68020 memory architecture has the following characteristics:
 - Logical address space: 4GB
 - Physical address space: 4GB
- Show the aliasing regions among the two processors

Solution

- For each address of the MC68000 there are 256 distinct addresses in the MC68020 processor.
- The aliasing regions are identified by the following pattern:

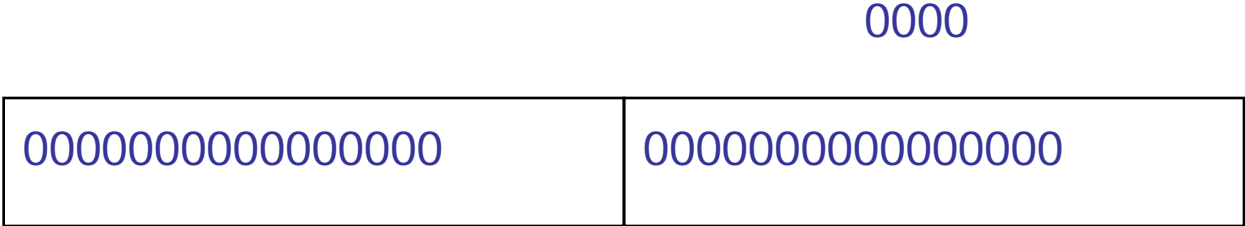


Example

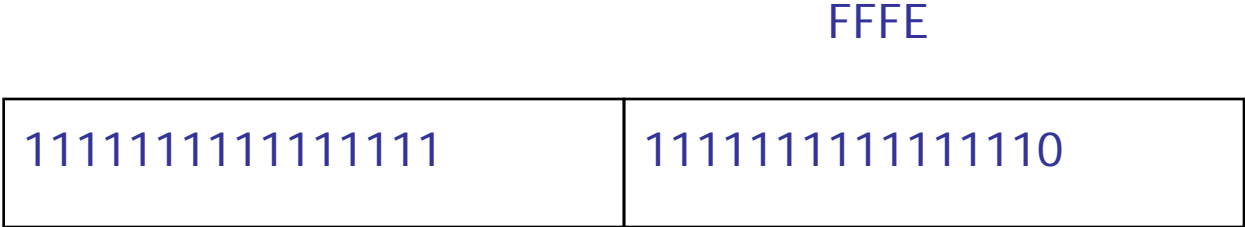
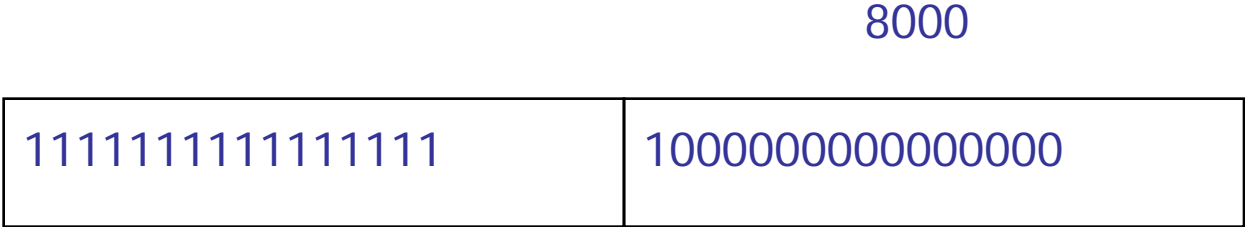
- By extending a 16 bit address with its sign bit, to show the memory area addressed in a 32-bit architecture

Soluzione

- Addresses between 0000 and 7FFE are mapped on the first 32KB of the 4GB memory



- Addresses between 8000 and FFFE are mapped on the last 32KB of the 4GB memory



Register Transfer Notation

- Register transfer notation is used to describe hardware-level data transfers and operations
- Predefined names for procr. and I/O registers
- Arbitrary names for locations in memory
- Use [...] to denote contents of a location
- Use \leftarrow to denote transfer to a destination
- Example: $R2 \leftarrow [LOC]$
(transfer from LOC in memory to register R2)

Register Transfer Notation

- RTN can be extended to also show arithmetic operations involving locations
- Example: $R4 \leftarrow [R2] + [R3]$
(add the contents of registers R2 and R3,
place the sum in register R4)
- Right-hand expression always denotes a value, left-hand side always names a location

Assembly-Language Notation

- RTN shows data transfers and arithmetic
- Another notation needed to represent machine instructions & programs using them
- **Assembly language** is used for this purpose
- For the two preceding examples using RTN, the assembly-language instructions are:

Load R2, LOC

Add R4, R2, R3

Assembly-Language Notation

- An instruction specifies the desired operation and the operands that are involved
- We will use English words for the operations (e.g., Load, Store, and Add) when they are not related to a specific architecture
- Commercial processors use **mnemonics**, usually abbreviations (e.g., LD, ST, and ADD)
- Mnemonics differ from processor to processor
 - Will use mnemonics to report specific processors related code

Memory Operations

- Memory contains data & program instructions
- Control circuits initiate transfer of data and instructions between memory and processor
- *Read* operation: memory retrieves contents at address location given by processor
- *Write* operation: memory overwrites contents at given location with given data

Addressing Modes

- Programs use data structures to organize the information used in computations
- High-level languages enable programmers to describe operations for data structures
- Compiler translates into assembly language
- **Addressing modes** provide compiler with different ways to specify operand locations
- Consider modes used in RISC-style processors

RISC-type addressing modes.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R_i	$EA = R_i$
Absolute	LOC	$EA = LOC$
Register indirect	(R_i)	$EA = [R_i]$
Index	$X(R_i)$	$EA = [R_i] + X$
Base with index	(R_i, R_j)	$EA = [R_i] + [R_j]$

EA = effective address

Value = a signed number

X = index value

Addressing Modes

- We have already seen examples of the *register* and *absolute* addressing modes
- RISC-style instructions have a fixed size, hence absolute mode information limited to 16 bits
- Usually sign-extended to full 32-bit address Absolute mode is therefore limited to a subset of the full 32-bit address space
- Assume programs are limited to this subset

Variables

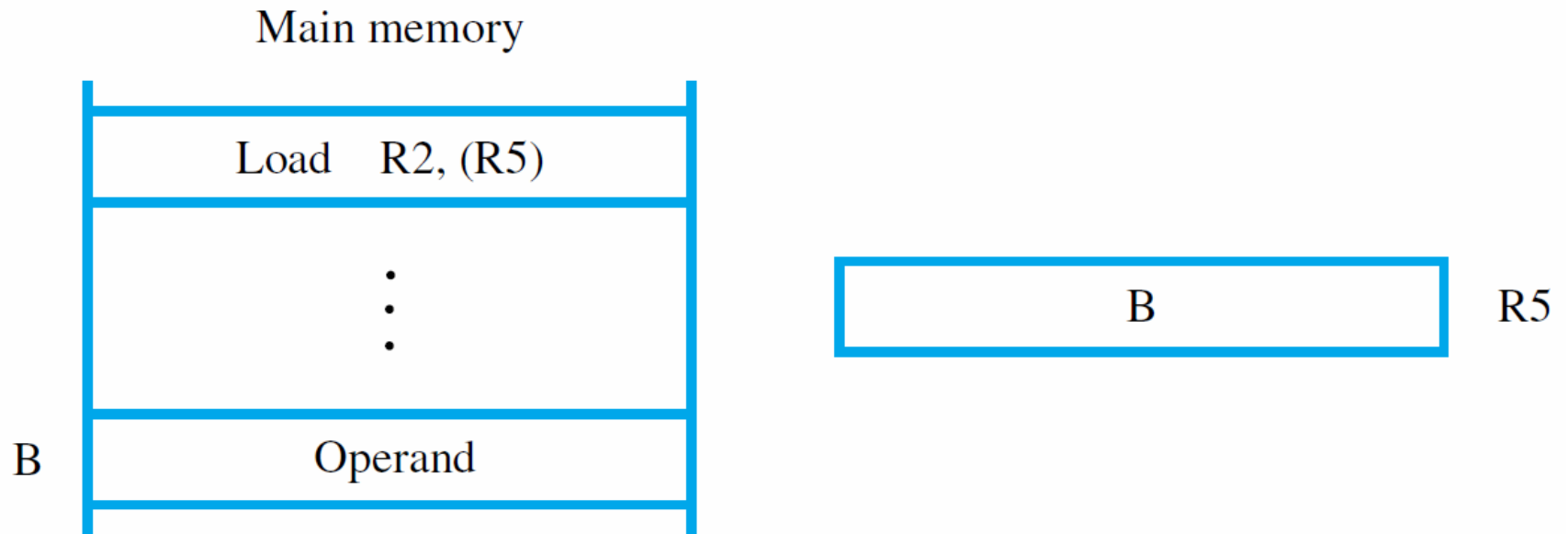
- Variable declaration in high-level language:
Integer NUM1, NUM2, SUM;
- Allocates storage to locations in the memory
- When referenced by high-level statements, compiler translates to assembly language:
Load R2, NUM1
- Absolute mode (in subset of address space) enables access to variables in the memory

Constants

- Assume constant 200 is added to a variable
- *Immediate* mode enables use of constants in assembly-language instructions
- One approach for specification:
Add R4, R6, 200_{immediate}
- Not practical to use subscripts in this manner
- Alternative approach uses special character:
Add R4, R6, #200

Indirection and Pointers

- Register, absolute, and immediate modes directly provide the operand or address
- Other modes provide information from which the **effective address** of operand is derived
- For program that adds numbers in a list, use register as **pointer** to next number
- *Indirect* mode provides address in register:
Load R2, (R5)



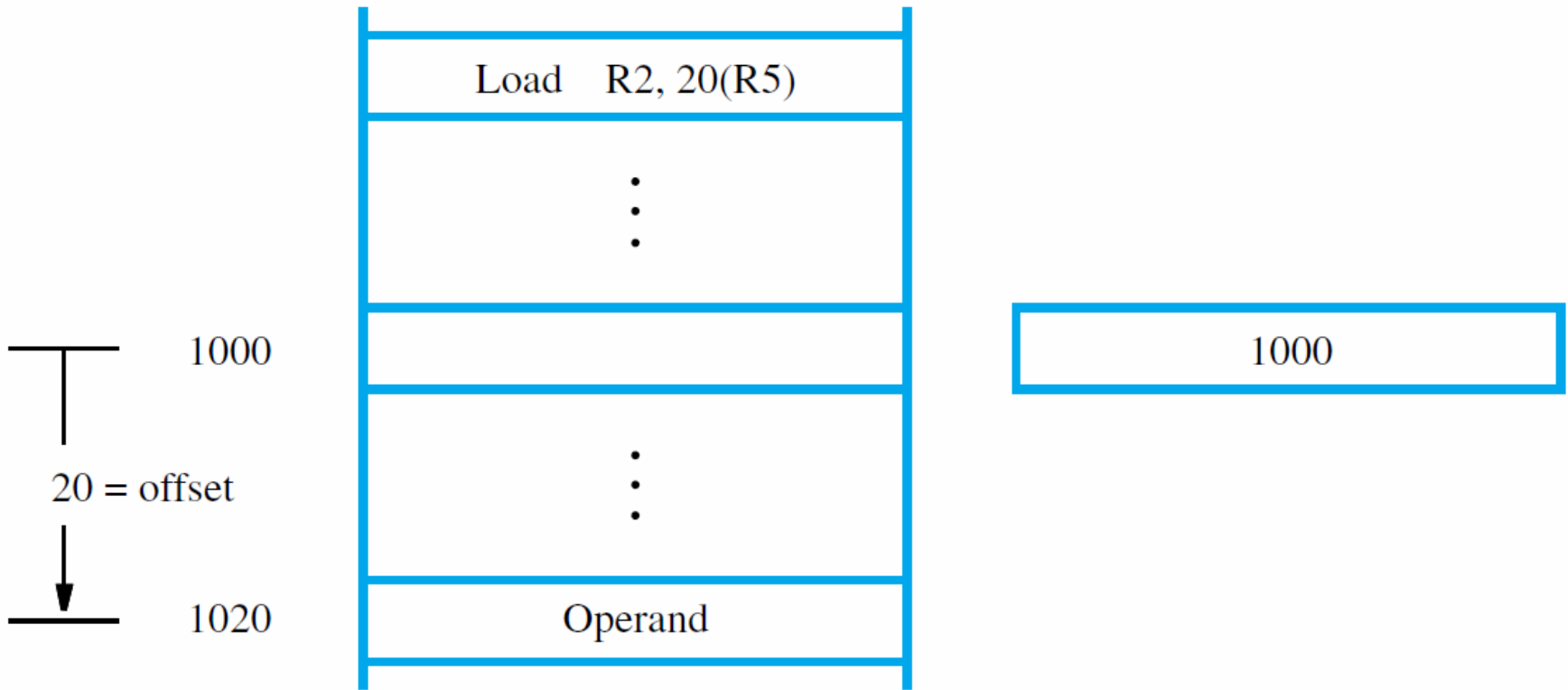
Indirection and Pointers

- Body of loop can now use register as pointer
- To initialize the pointer, use the instruction:
Move R4, #NUM1
- In RISC-style processors, R0 is usually always 0
- Implement using Add and immediate mode:
Add R4, R0, #NUM1
- Move is a convenient **pseudoinstruction**
- We now have complete list-addition program

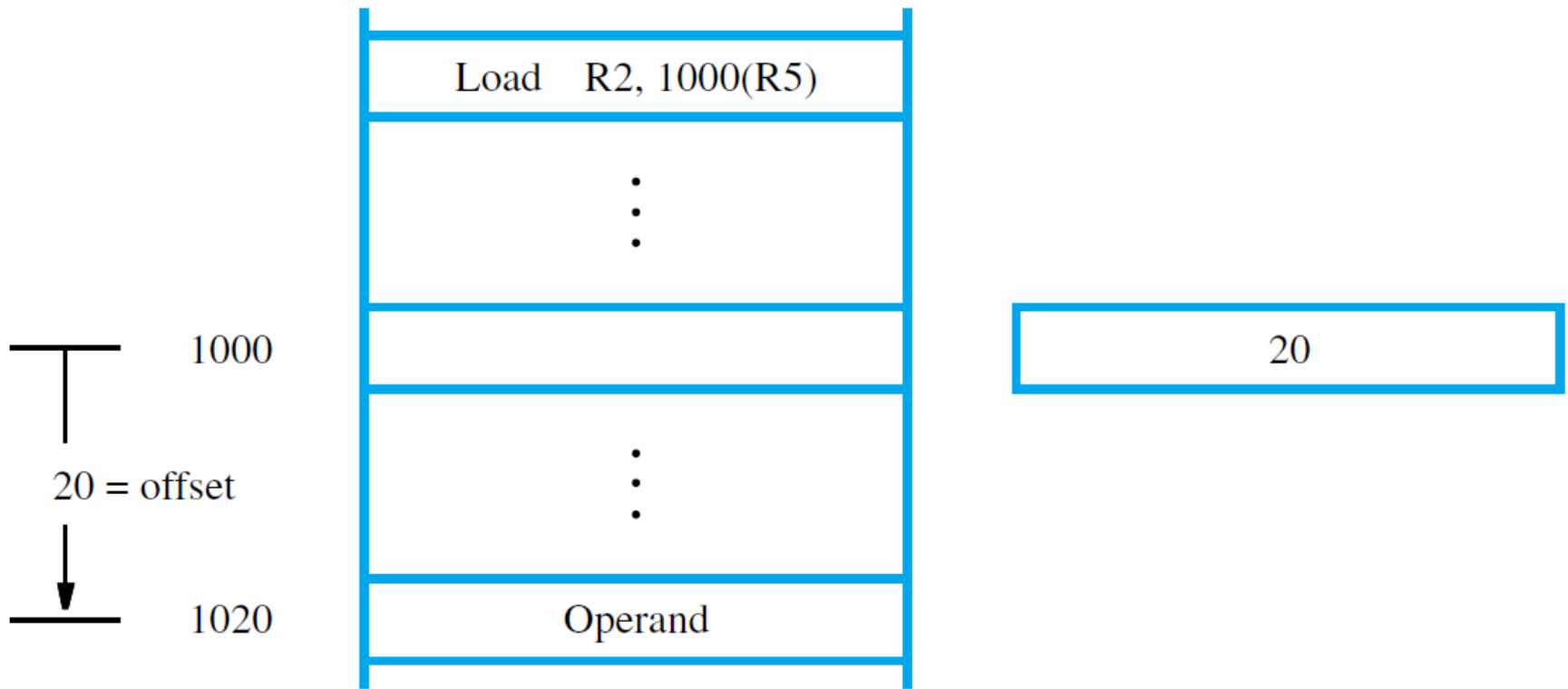
	Load	R2, N	Load the size of the list.
	Clear	R3	Initialize sum to 0.
	Move	R4, #NUM1	Get address of the first number.
LOOP:	Load	R5, (R4)	Get the next number.
	Add	R3, R3, R5	Add this number to sum.
	Add	R4, R4, #4	Increment the pointer to the list.
	Subtract	R2, R2, #1	Decrement the counter.
	Branch_if_ $[R2]>0$	LOOP	Branch back if not finished.
	Store	R3, SUM	Store the final sum.

Indexing

- Consider *index* mode in: Load R2, $X(R5)$
- Effective address is given by $[R5] + X$
- For example, assume operand address is 1020, 4 words (20 bytes) from start of array at 1000
- Can put start address in R5 and use $X=20$
- Alternatively, put offset in R5 and use $X=1000$
- *Base with index* mode: Load Rk, $X(Ri, Rj)$
- Effective address is given by $[Ri] + [Rj] + X$



(a) Offset is given as a constant



(b) Offset is in the index register

Additional Addressing Modes

- CISC style has other modes not usual for RISC
- *Autoincrement* mode: effective address given by register contents; after accessing operand, register contents incremented to point to next
- Useful for adjusting pointers in loop body:
 - Add SUM, (R_i)⁺
 - MoveByte (R_j)⁺, R_k
- Increment by 4 for words, and by 1 for bytes

ARM MEMORY AND REGISTER STRUCTURE, AND ADDRESSING MODES

ARM – Advanced RISC Machine

- 32-bit RISC-processor core (32-bit instructions)
- 37 internal registers of 32-bit (16)
- Pipeline (ARM7: 3 stadi)
- Cache (depends on implementation)
- Von Neuman-type bus structure (ARM7), Harvard (ARM9)
- Data types 8 / 16 / 32 -bit
- 7 modalità (usr, fiq, irq, svc, abt, sys, und)
- Struttura semplice → buon rapporto fra velocità / consumo

RISC: Reduced Instruction Set Computing

- Instructions: simpler but more efficient
- High clock frequency
- More complex compiling and debugging
- Higher number of registers

Data types

- byte
- halfword (2bytes aligned)
- word (4byte aligned)

Memory structure

- Byte addressable
- Half and full words (16 or 32 bits) can be organized as both big-endian and little-endian

Processor Modes

ARM seven processing modes, depending on the code being executed:

User (usr)

modalità standard di esecuzione del processo

FIQ (fiq)

modalità privilegiata per gestione di flussi dato ad alta velocità

IRQ (irq)

modalità privilegiata per la gestione degli interrupt

Supervisor (svc)

modalità privilegiata per l'esecuzione del Sistema Operativo

Abort (abt)

implementa la memoria virtuale e la protezione della memoria

System (sys)

modalità privilegiata per l'esecuzione dei task del S.O.

und

serve per il supporto all'emulazione software dei coprocessori

Registers

37 registers

- 31 general purpose
- 6 status registers

At every time 15 general purpose registers and two status registers are in use

Registers

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Registers

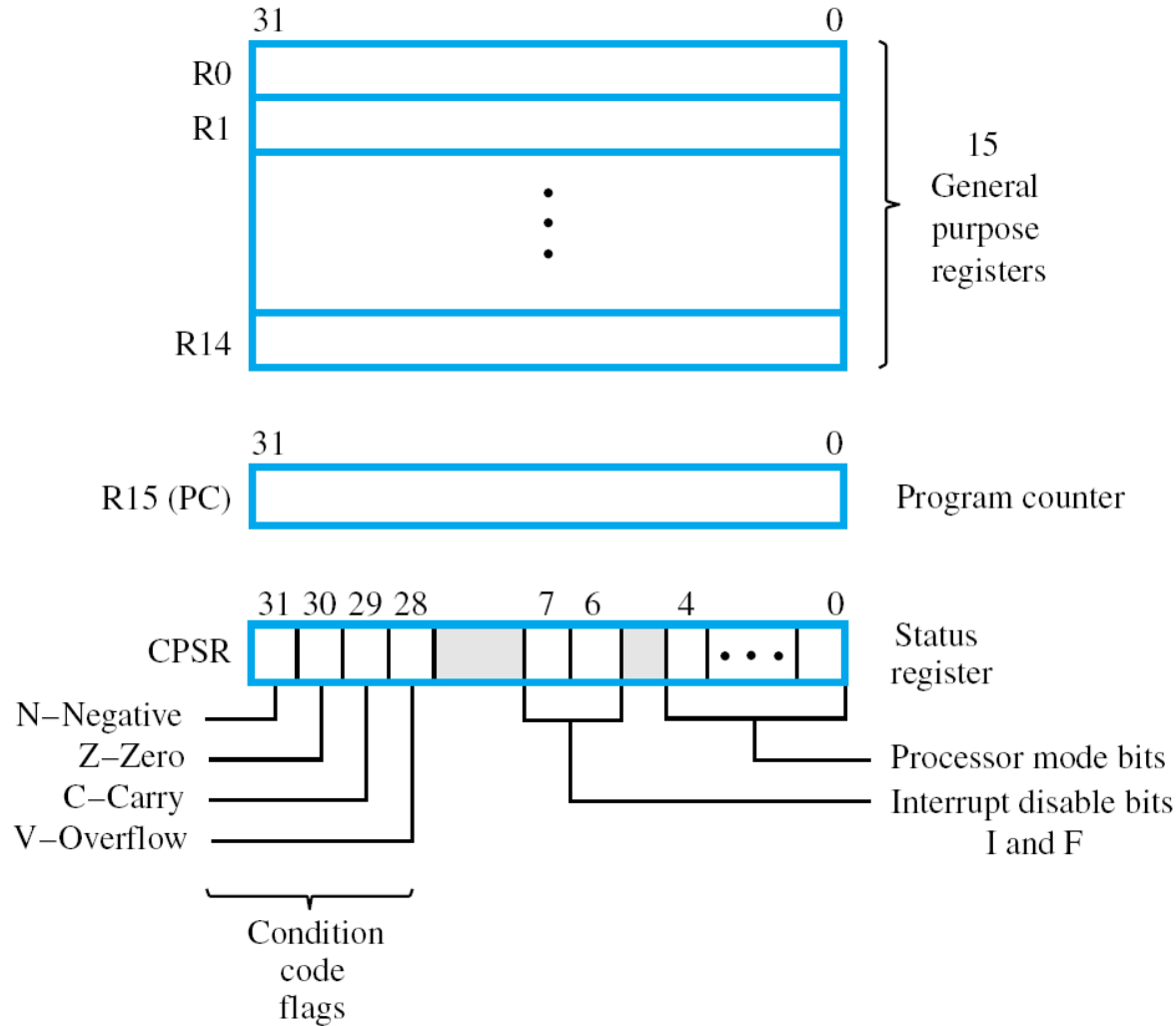
Modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

Registers

- The first 7 registers (R0-R7) are unbanked (physical location shared among all the processor modes)
- Registers from R8 to R14 are banked (depending on the specific processor mode they point to a specific physical location)
 - R8-R12 banked only for the FIQ mode (quick context switch for executing high ISR)
 - R13, R14 ed R15 normally used as Stack Pointer, Link Register, and Program Counter
- A status register (CPSR) holds the condition code flags (N, Z, C, V), two interrupt-disable bits, and five processor mode bits

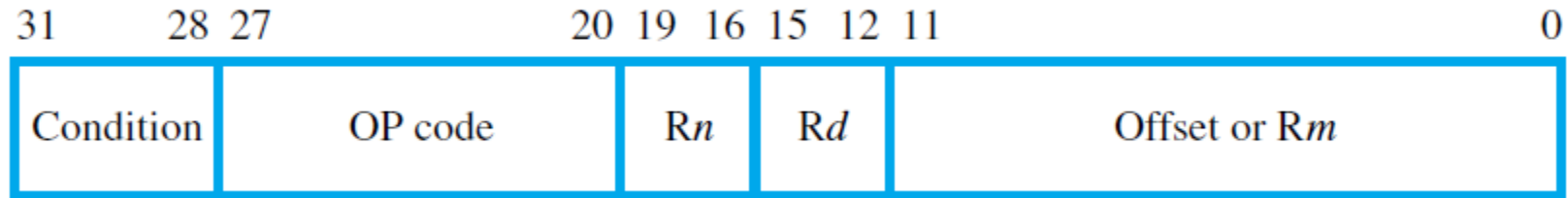
Status Register



Addressing modes

- All modes are derived from a basic form of **indexed addressing**
- The **effective address** of a memory operand is the sum of the contents of a **base** register R_n and a signed **offset**
- The offset is either a 12-bit immediate value in the instruction or the contents of a second register R_m
- Examples of addressing modes can be shown by using the Load instruction LDR, whose format is given in following slide
- The store instruction STR has same format
- Both LDR and STR access a word location

LOAD Instruction Encoding



Instruction Structure

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Data processing immediate shift	cond [1]	0	0	0	opcode			S	Rn				Rd				shift amount			shift	0	Rm												
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x																	0	x						
Data processing register shift [2]	cond [1]	0	0	0	opcode			S	Rn				Rd				Rs			0	shift	1	Rm											
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x																	0	x	x	1	x			
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x																	1	x	x	1	x								
Data processing immediate [2]	cond [1]	0	0	1	opcode			S	Rn				Rd				rotate			immediate														
Undefined instruction [3]	cond [1]	0	0	1	1	0	x	0	0	x																								
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask				SBO				rotate			immediate													
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn				Rd				immediate																
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn				Rd				shift amount			shift	0	Rm											
Undefined instruction	cond [1]	0	1	1	x																	1	x											
Undefined instruction [4,7]	1	1	1	1	0	x																												
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn				register list																				
Undefined instruction [4]	1	1	1	1	1	0	0	x																										
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																												
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset																									
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn				CRd				cp_num			8-bit offset													
Coprocessor data processing	cond [5]	1	1	1	0	opcode1			CRn				CRd				cp_num			opcode2	0	CRm												
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1			L	CRn				Rd				cp_num			opcode2	1	CRm											
Software interrupt	cond [1]	1	1	1	1	swi number																												
Undefined instruction [4]	1	1	1	1	1	1	1	1	x																									

Addressing modes

- Pre-indexed mode:

LDR $Rd, [Rn, \#offset]$

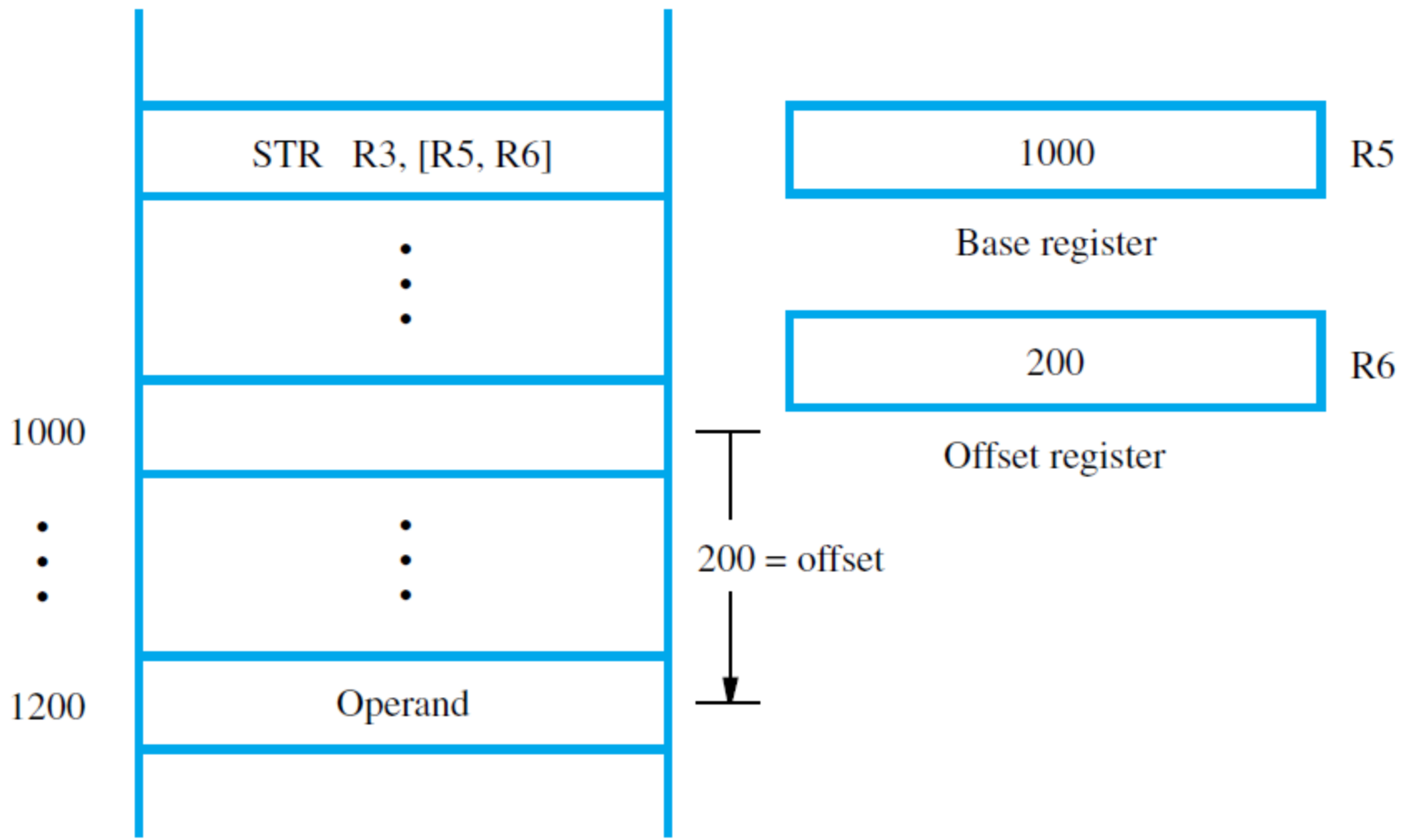
performs

$Rd \leftarrow [[Rn] + offset]$

LDR $Rd, [Rn, Rm]$

performs

$Rd \leftarrow [[Rn] + [Rm]]$



Addressing modes

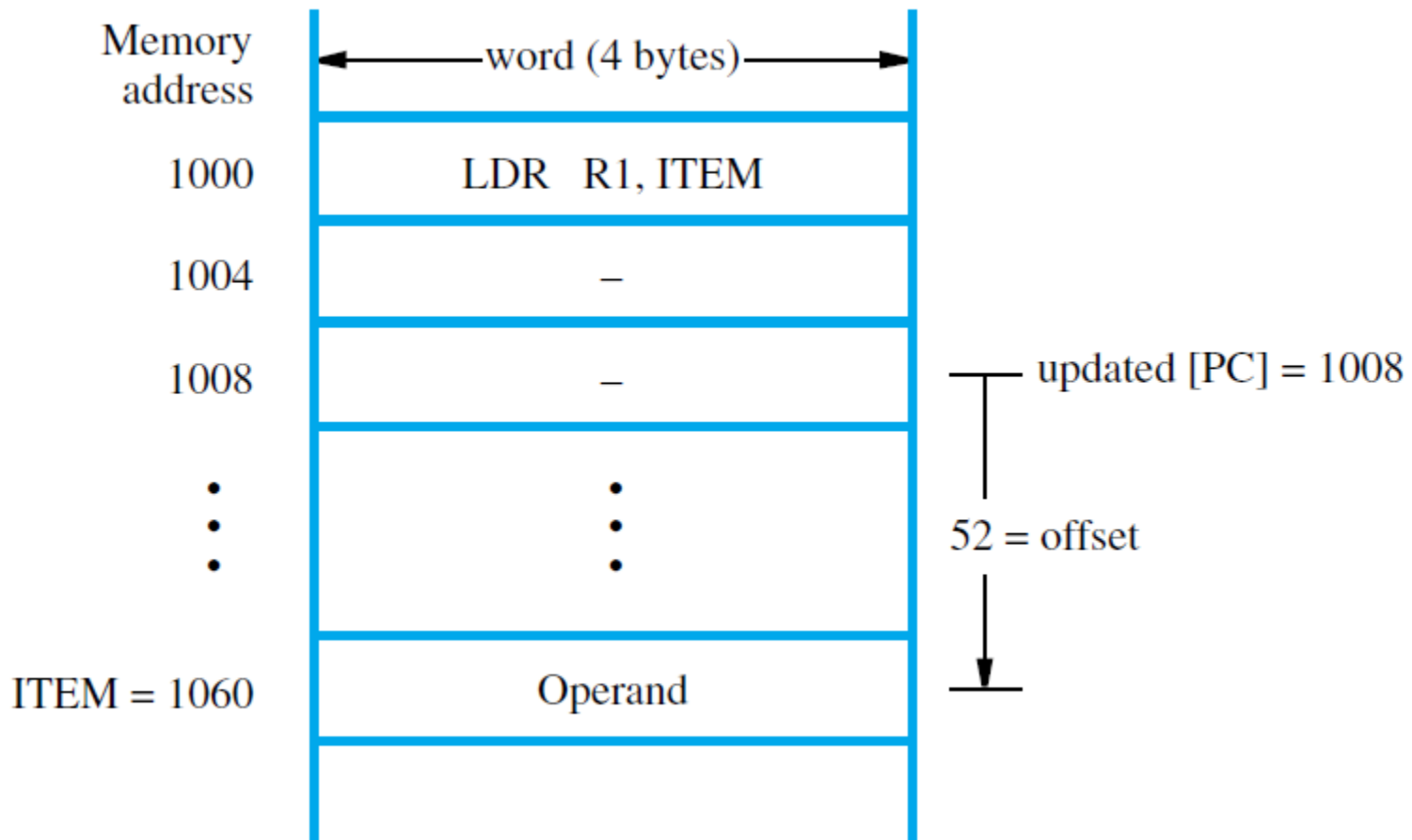
- Relative mode:

LDR Rd , ITEM

performs

$Rd \leftarrow [[PC] + \text{offset}]$

where offset is calculated by the assembler



Addressing modes

- Pre-indexed with writeback (a generalization of the autodecrement mode):

LDR $Rd, [Rn, \#offset]!$

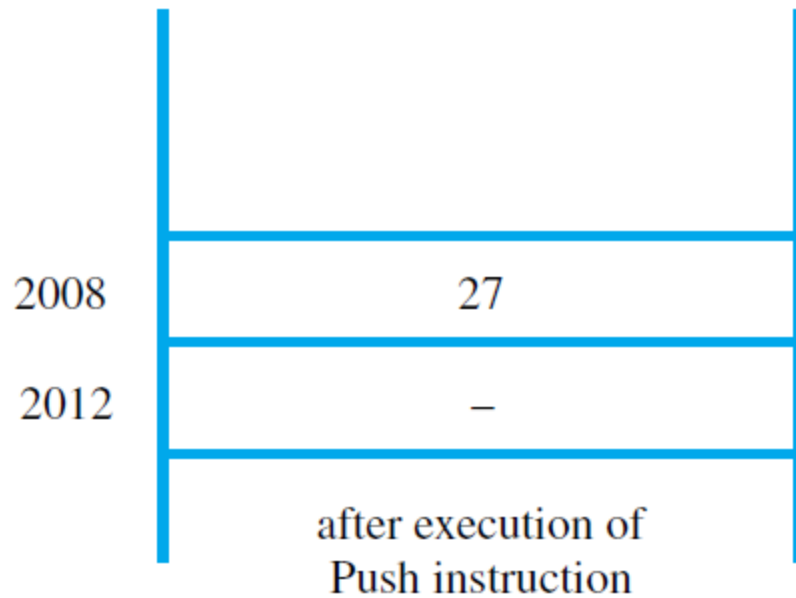
performs

$Rd \leftarrow [[Rn] + offset]$

followed by

$Rn \leftarrow [Rn] + offset$

(Rm can be used instead of $\#offset$)



Push instruction:
STR R0, [R5, #-4]!

Addressing modes

- Post-indexed mode (a generalization of the autoincrement mode):

LDR $Rd, [Rn], \#offset$

performs

$Rd \leftarrow [[Rn]]$

followed by

$Rn \leftarrow [Rn] + offset$

(Rm can be used instead of $\#offset$)

Addressing modes

- If the offset is given as the **contents of R_m** , it can be **shifted** before being used

Example:

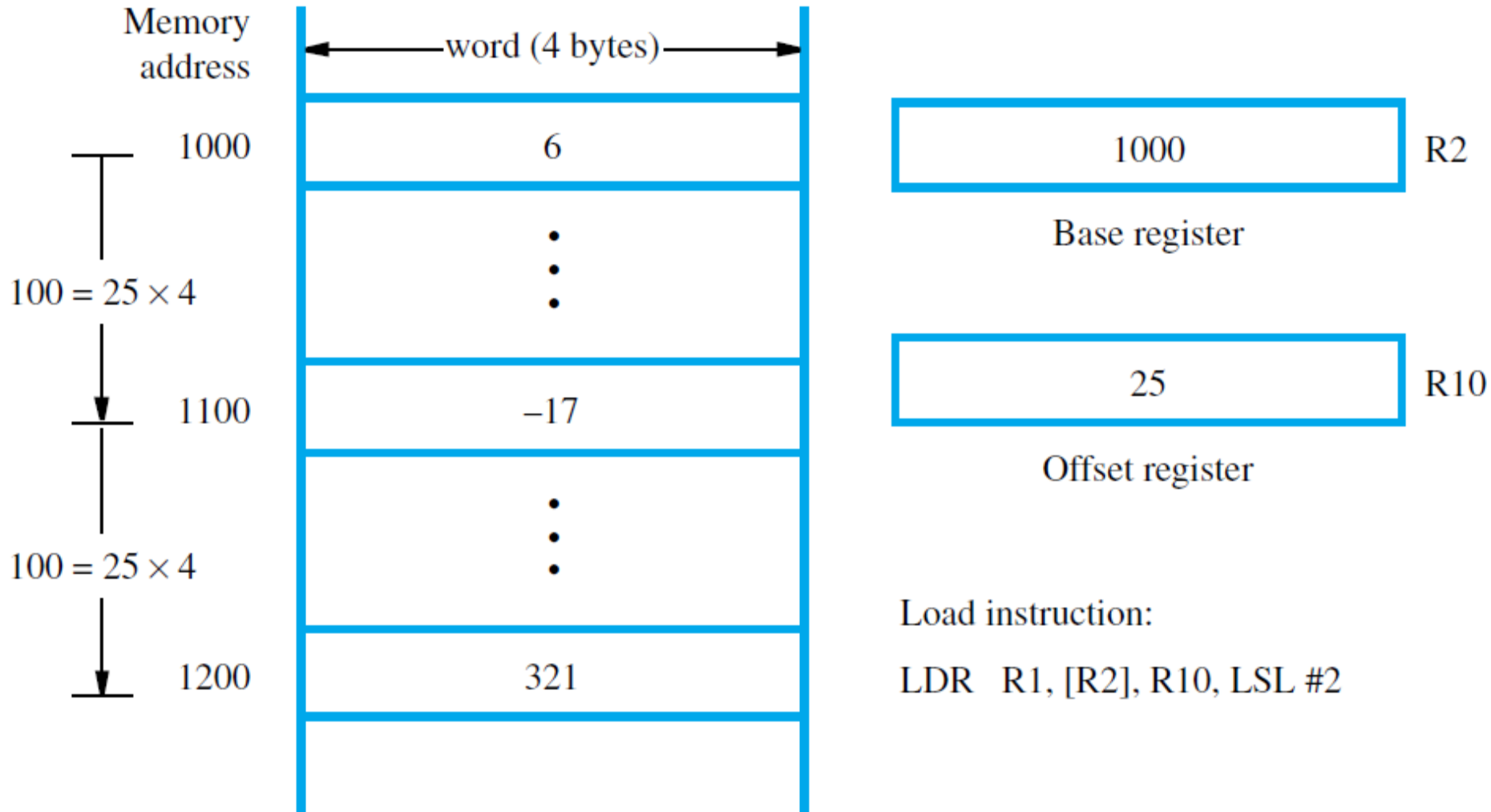
```
LDR R0, [R1, -R2, LSL #4]!
```

performs

$$R0 \leftarrow [[R1] - 16 \times [R2]]$$

followed by

$$R1 \leftarrow [R1] - 16 \times [R2]$$



ARM indexed addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	$[Rn, \#offset]$	$EA = [Rn] + offset$
Pre-indexed with writeback	$[Rn, \#offset]!$	$EA = [Rn] + offset;$ $Rn \leftarrow [Rn] + offset$
Post-indexed	$[Rn], \#offset$	$EA = [Rn];$ $Rn \leftarrow [Rn] + offset$
With offset magnitude in Rm :		
Pre-indexed	$[Rn, \pm Rm, shift]$	$EA = [Rn] \pm [Rm] \text{ shifted}$
Pre-indexed with writeback	$[Rn, \pm Rm, shift]!$	$EA = [Rn] \pm [Rm] \text{ shifted};$ $Rn \leftarrow [Rn] \pm [Rm] \text{ shifted}$
Post-indexed	$[Rn], \pm Rm, shift$	$EA = [Rn];$ $Rn \leftarrow [Rn] \pm [Rm] \text{ shifted}$
Relative (Pre-indexed with immediate offset)	Location	$EA = \text{Location}$ $= [PC] + offset$

EA = effective address

offset = a signed number contained in the instruction

shift = direction #integer

where direction is LSL for left shift or LSR for right shift; and

integer is a 5-bit unsigned number specifying the shift amount

$\pm Rm$ = the offset magnitude in register Rm can be added to or subtracted from the contents of base register Rn

COLDFIRE MEMORY AND REGISTER STRUCTURE

Memory Organization

- Byte-addressable, 32-bit address space
- Big-endian addressing scheme
- Longword (32-bit), word (16-bit), and byte (8-bit) sizes for integer data

Word
address

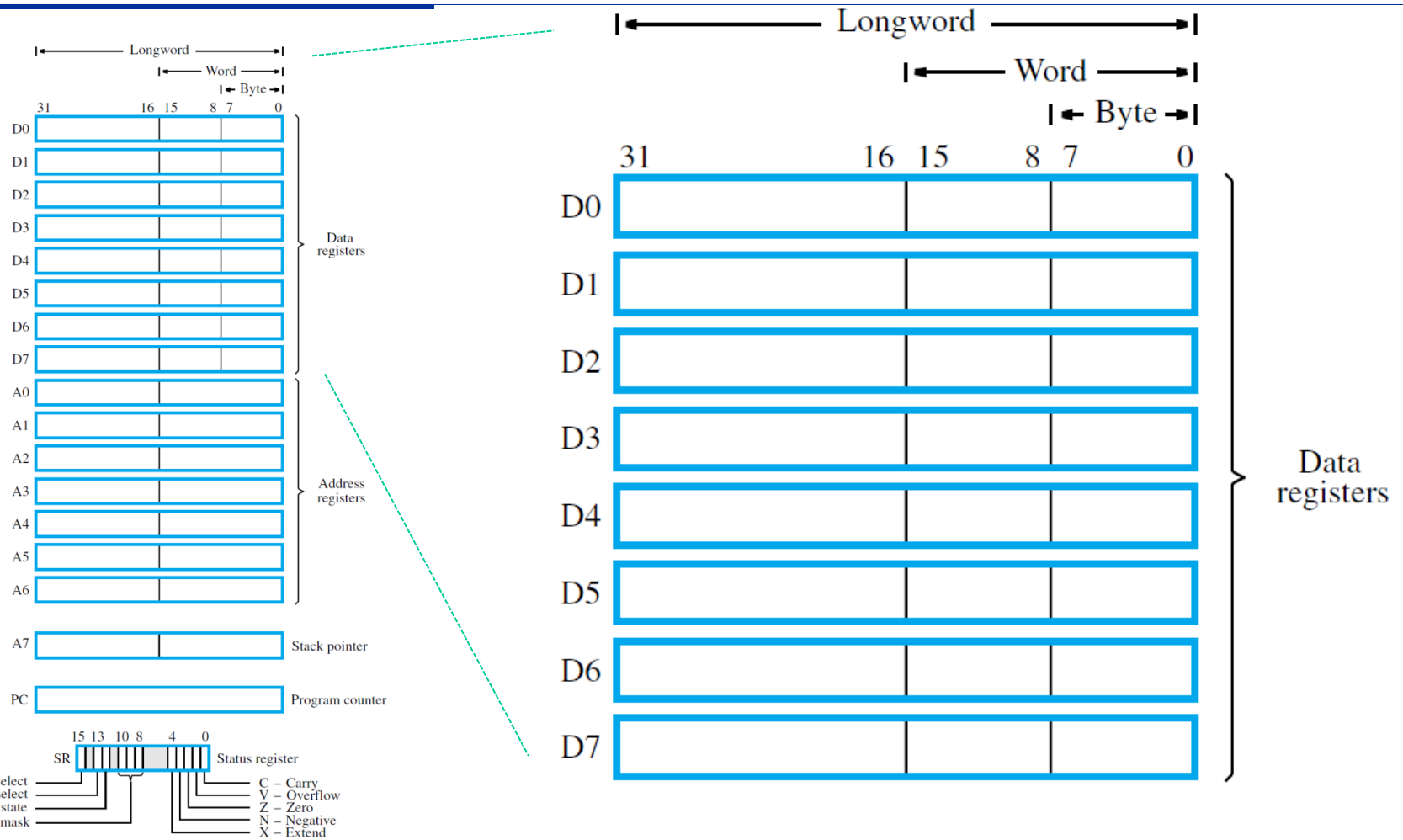
Contents

0	byte 0	byte 1	} Longword 0 (byte 0 is the high-order byte)
2	byte 2	byte 3	
	⋮	⋮	
<i>i</i>	byte <i>i</i>	byte <i>i</i> + 1	} Longword <i>i</i> (byte <i>i</i> is the high-order byte)
<i>i</i> + 2	byte <i>i</i> + 2	byte <i>i</i> + 3	
	⋮	⋮	
$2^{31} - 2$	byte $2^{31} - 2$	byte $2^{31} - 1$	

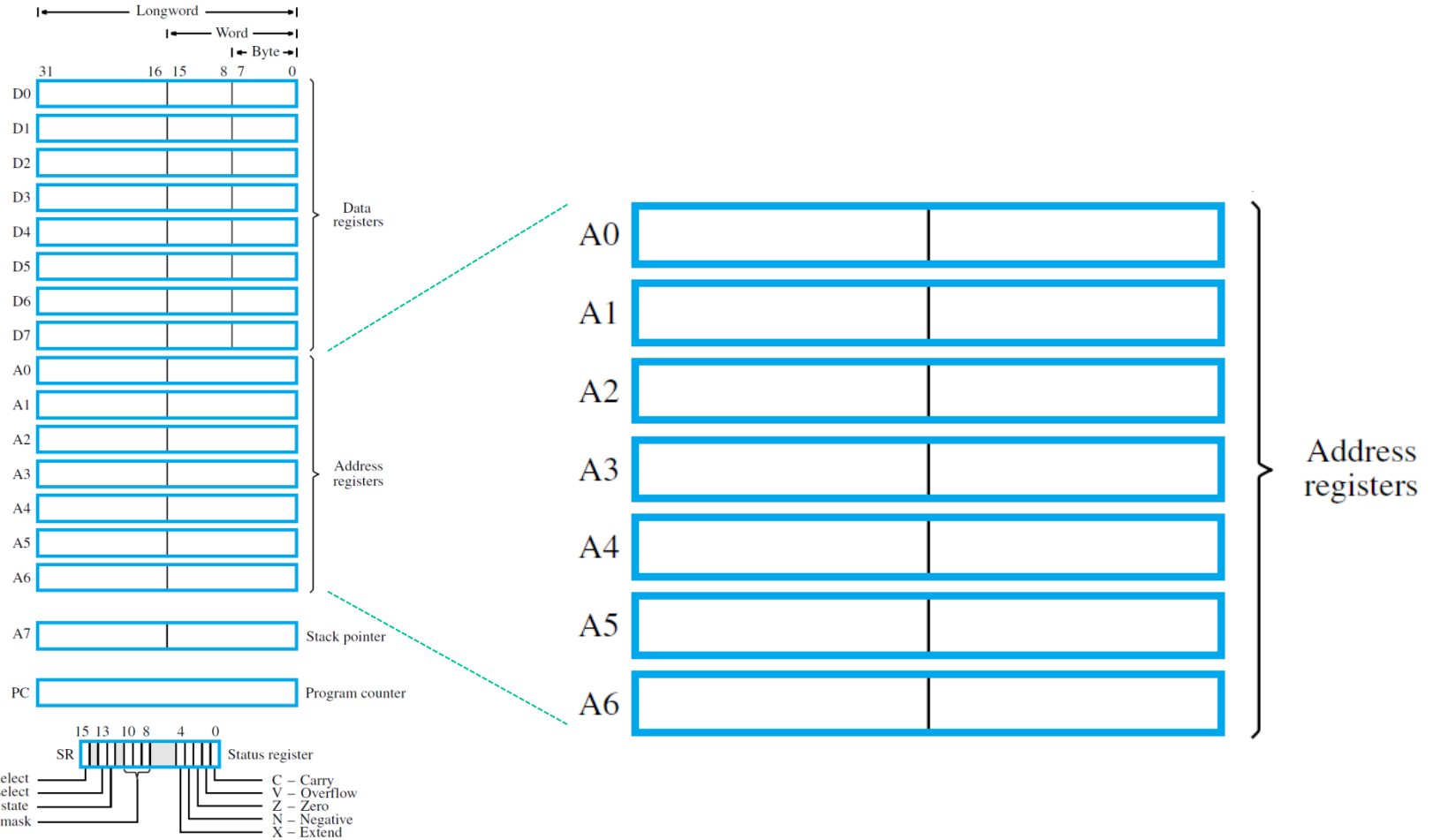
Register Structure

- Eight data registers, D0 to D7
- Eight address registers, A0 to A7, and register A7 is the stack pointer (SP)
- Status register (SR) with condition codes

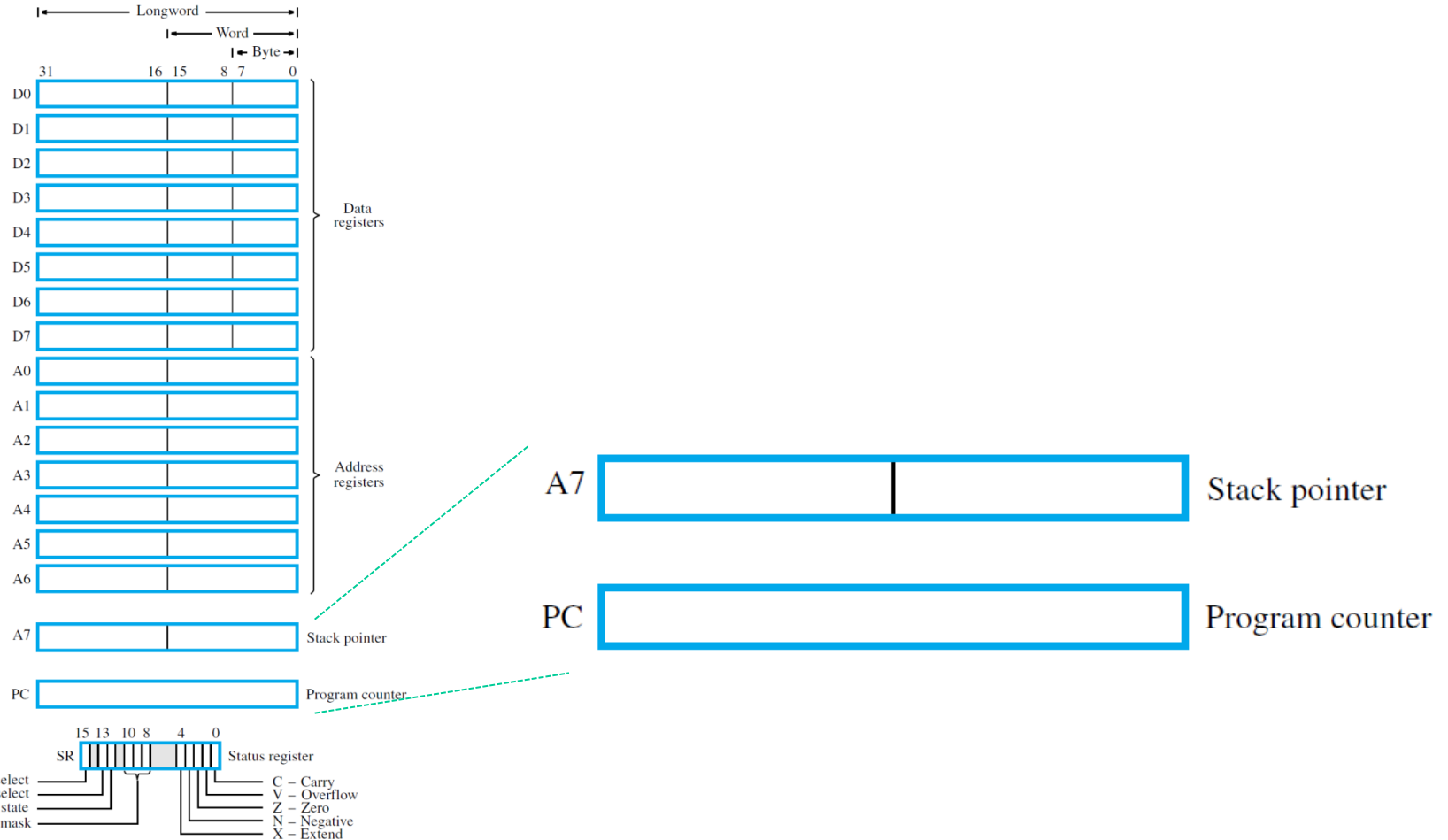
Data Registers



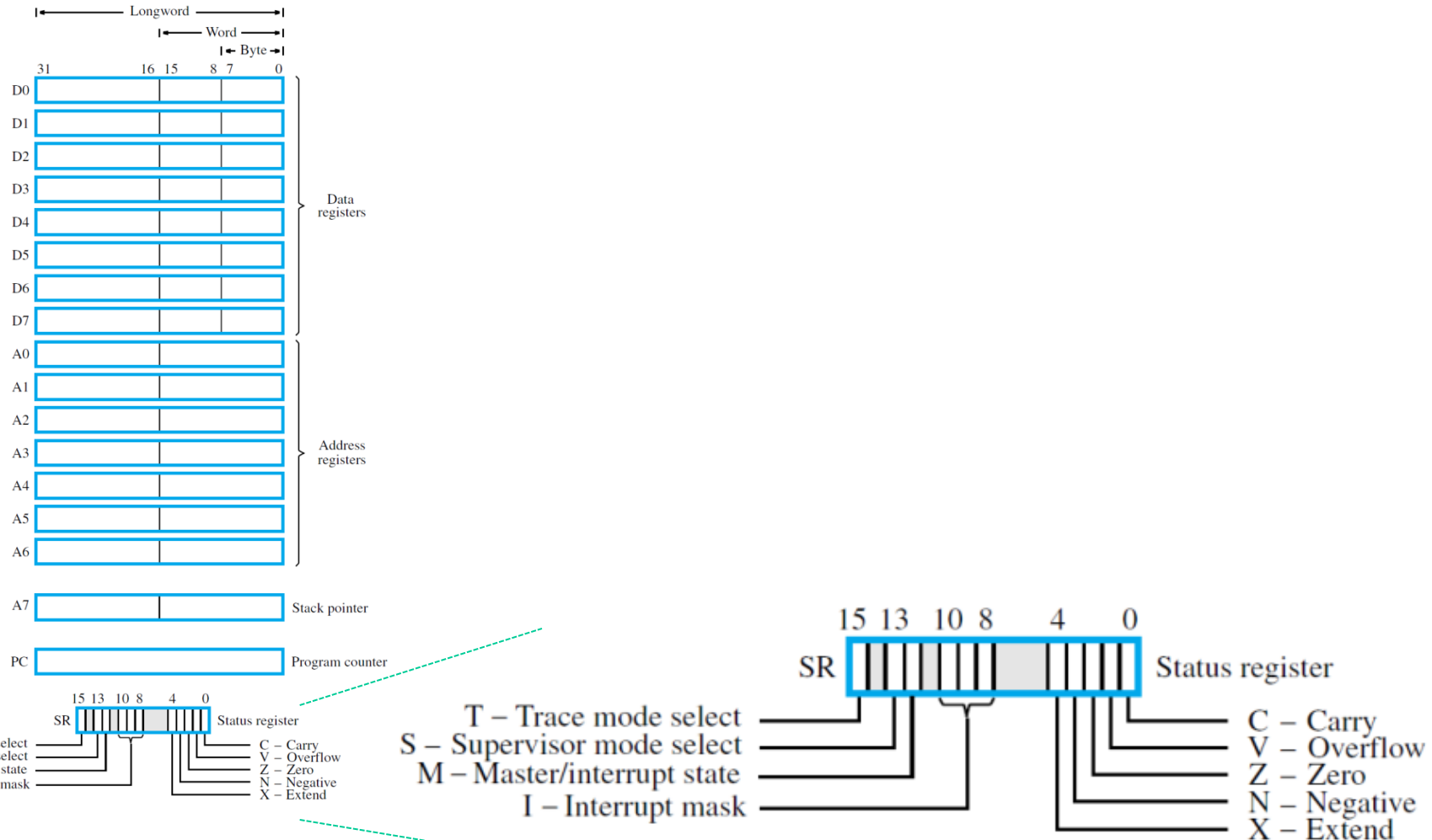
Address Registers



SP and PC



Status Register



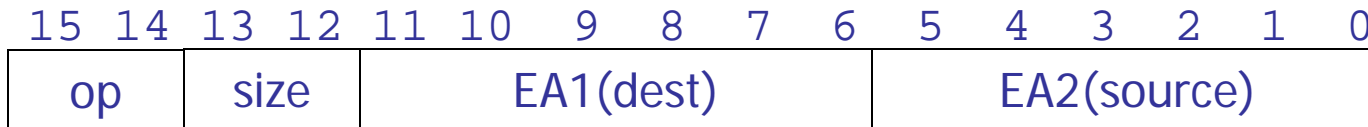
Instructions

- 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

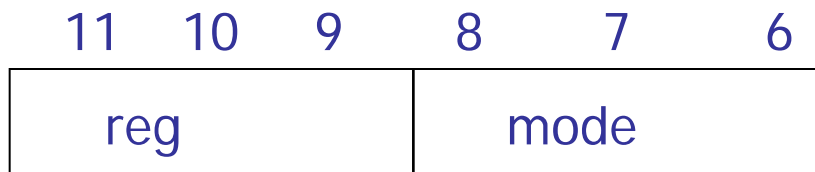
- all the addressing modes can be used for both source and destination (with the exception of the immediate for the destination)

- The MOVE instruction is full orthogonal (orthogonal ISA => all the instructions are orthogonal)

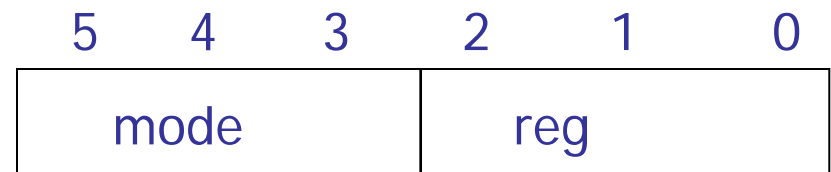


Addressing Mode encoding

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

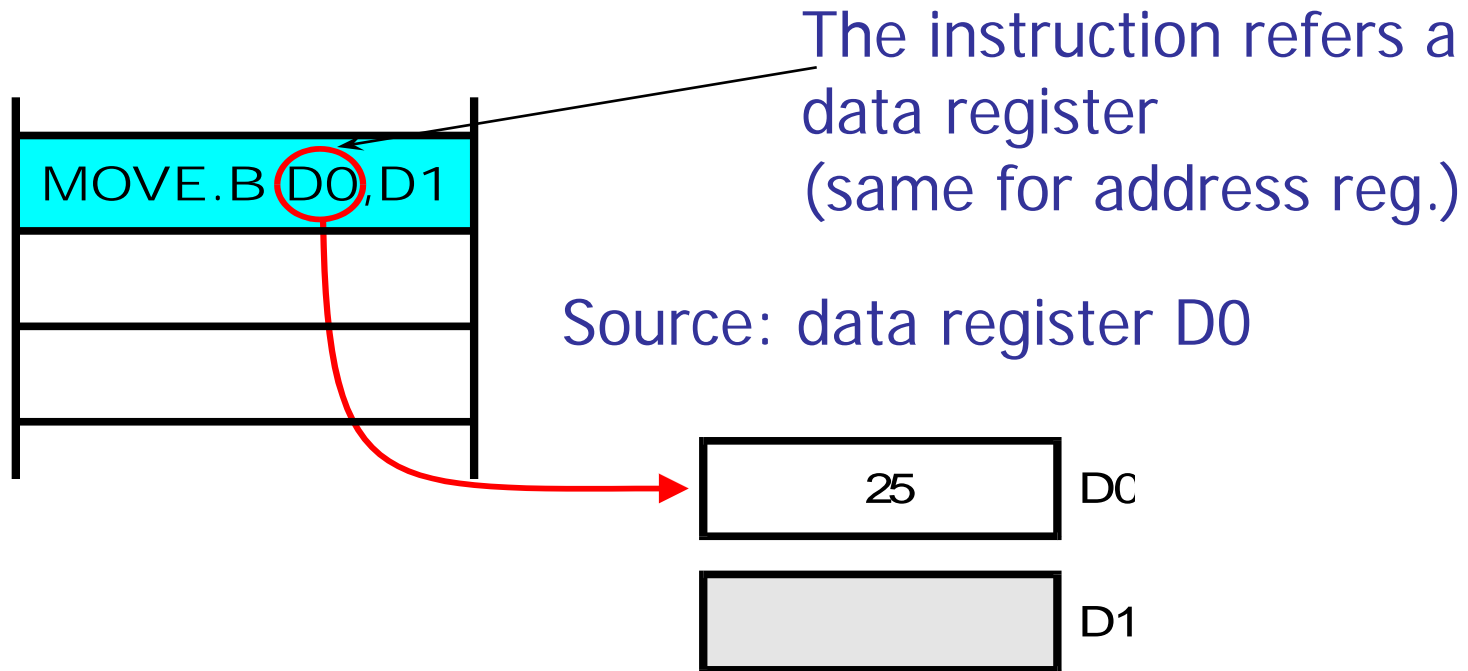


Destination operand
MOVE



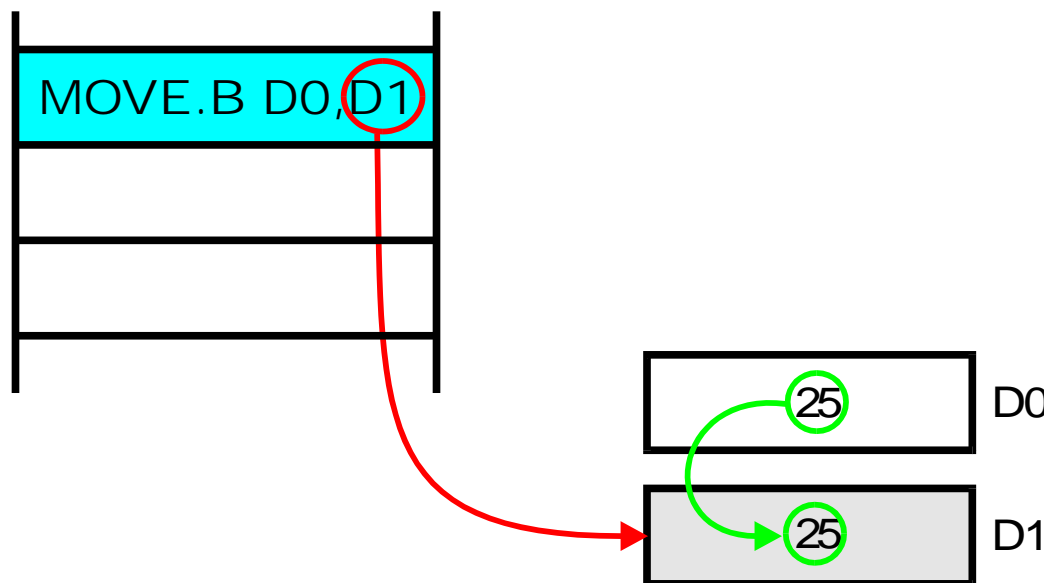
EA field for all the other
cases

Register Direct Addressing



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

Register Direct Addressing



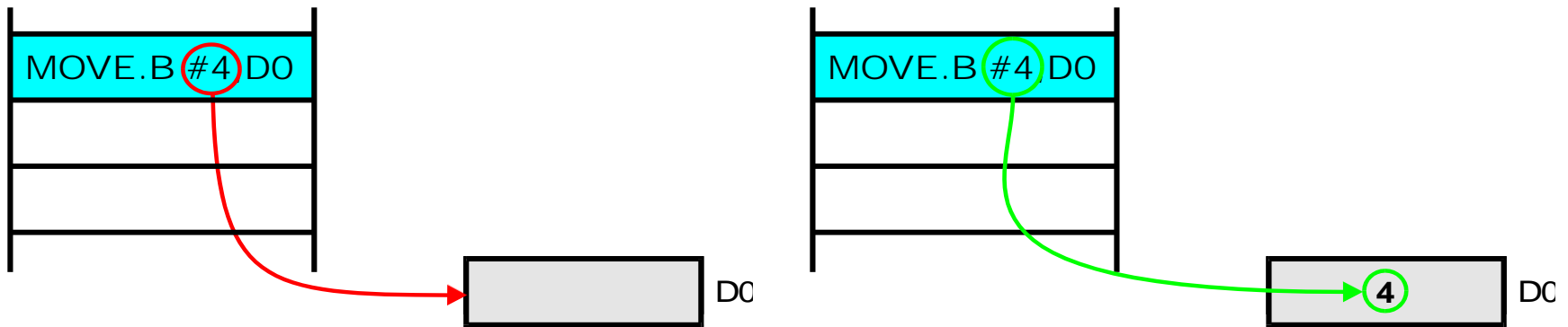
The final result is that D0 content is copied to D1

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

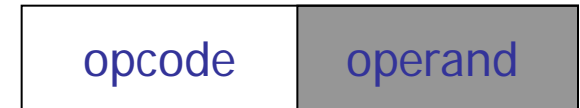
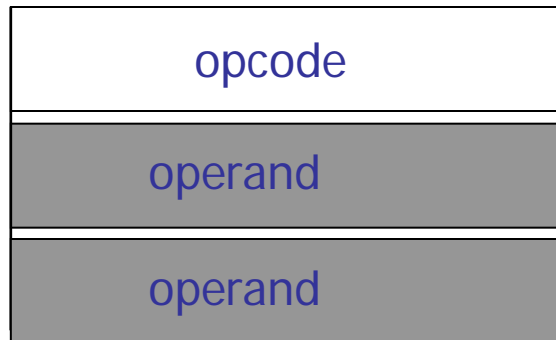
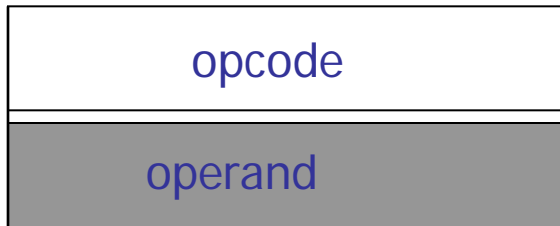
- The real operand is made available as part of the instruction
- Only used for source operand
- The symbol # used ahead of the value
- The immediate operand is also said «literal»



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

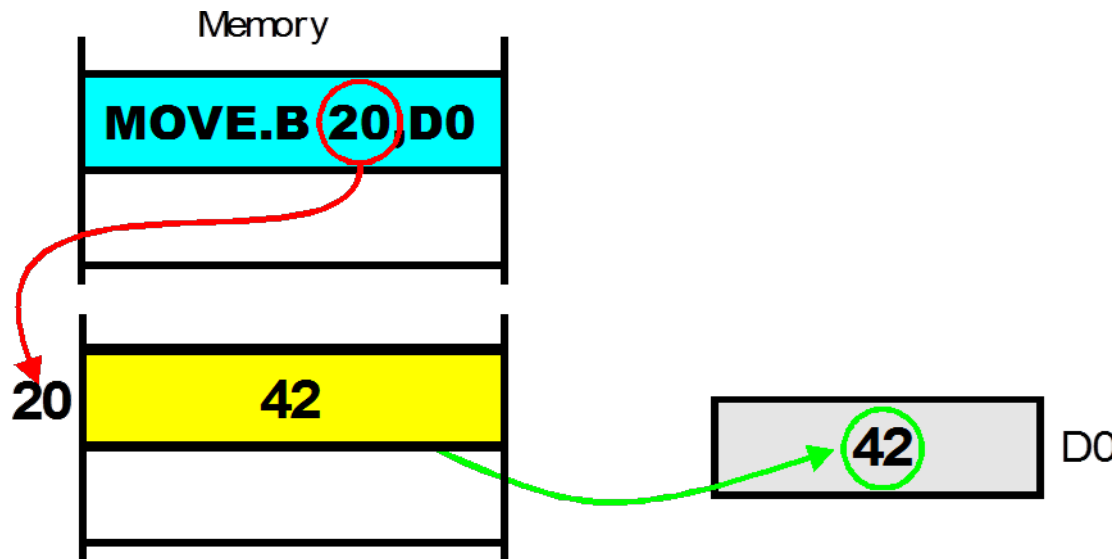
ImmediateAddressing - Encoding

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



Absolute Addressing (or Direct Addressing)

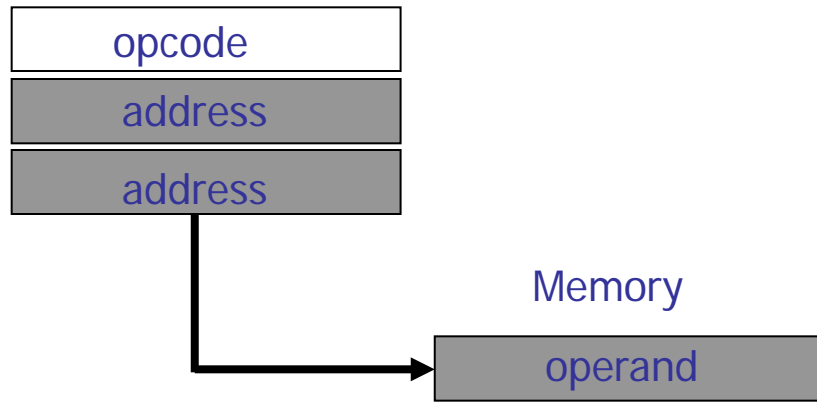
- The instruction refers a memory address that contains the actual operand
- Two memory accesses:
 - Instruction fetch
 - Operand assembly



Absolute Addressing - Codifica

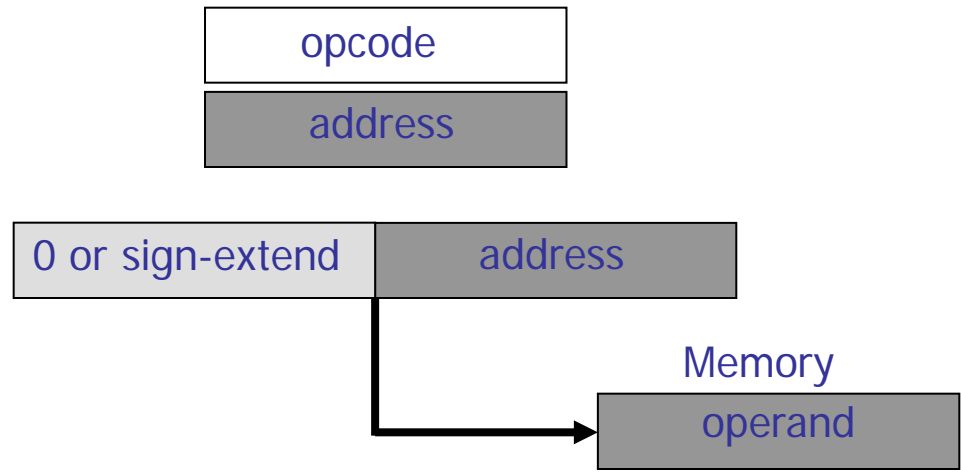
Absolute Long:

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

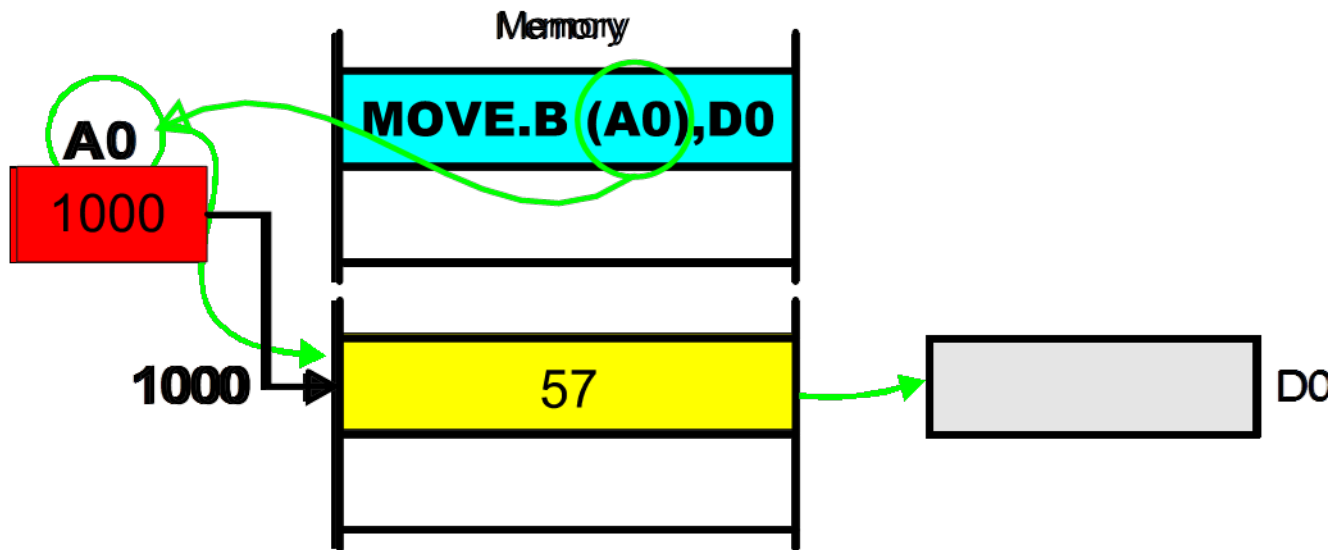
ORG    $600           data section
Y      DC.B 27        store a constant
Z      DS.B 1         reserve a byte for Z
```

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 00000001      Z: DS.B 1
10     00000400      END $400
```

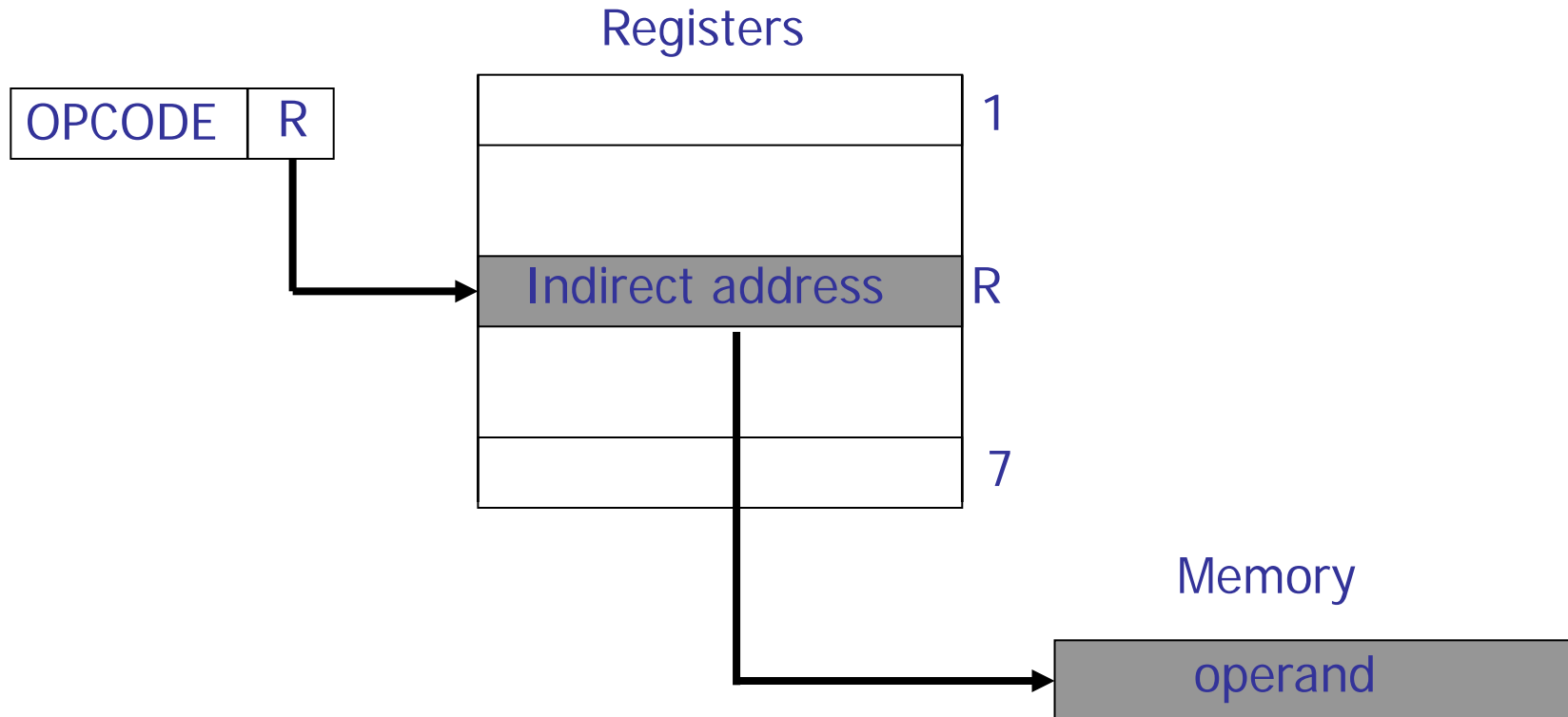
Address Register Indirect Addressing

- 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 ottenere 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

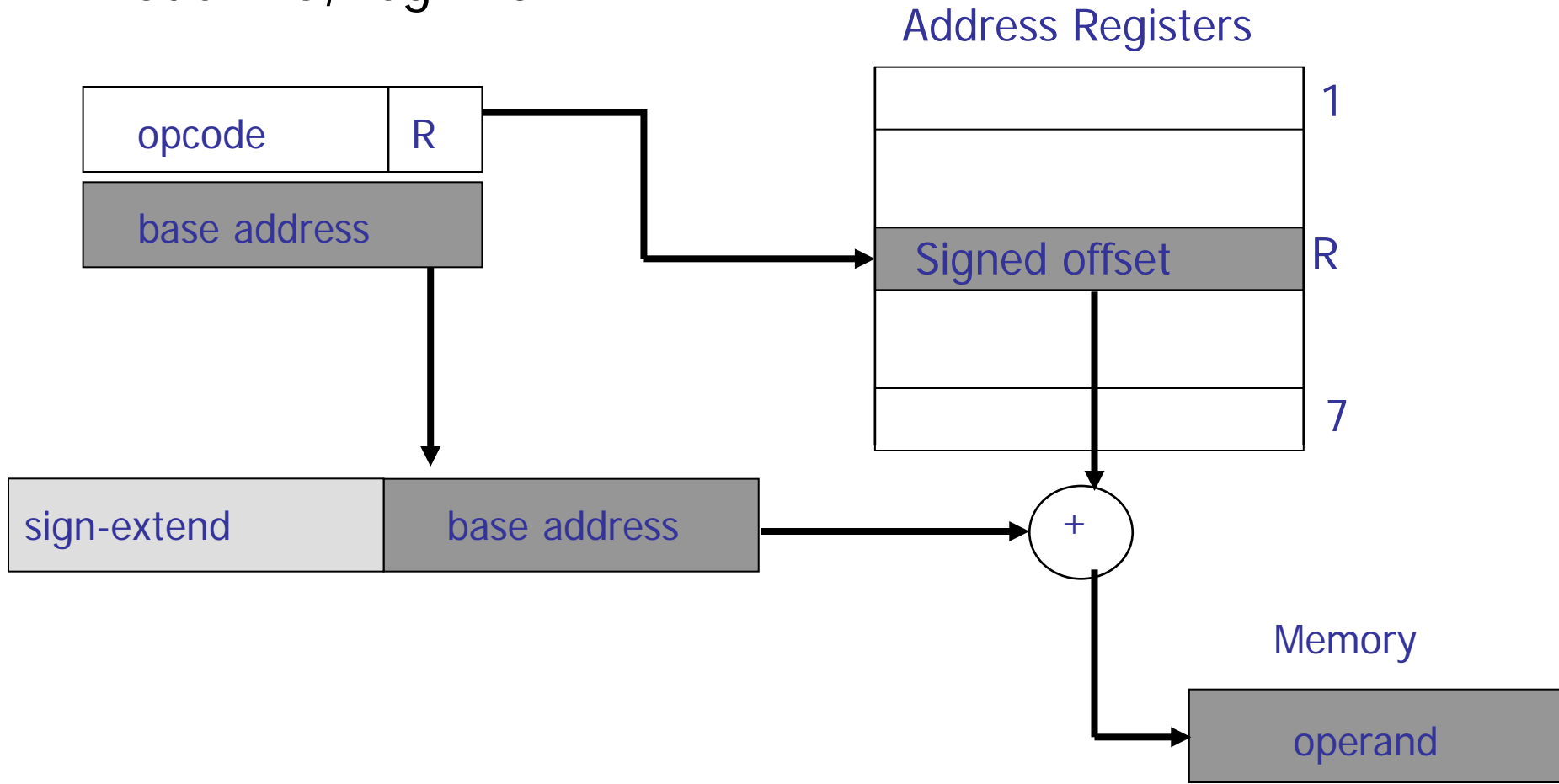
MOVE.B

CLIST-1(AO),D1

Leggi clist[i]

Indexed Short Addressing - Codifica

- mode = 5, reg = 0-7

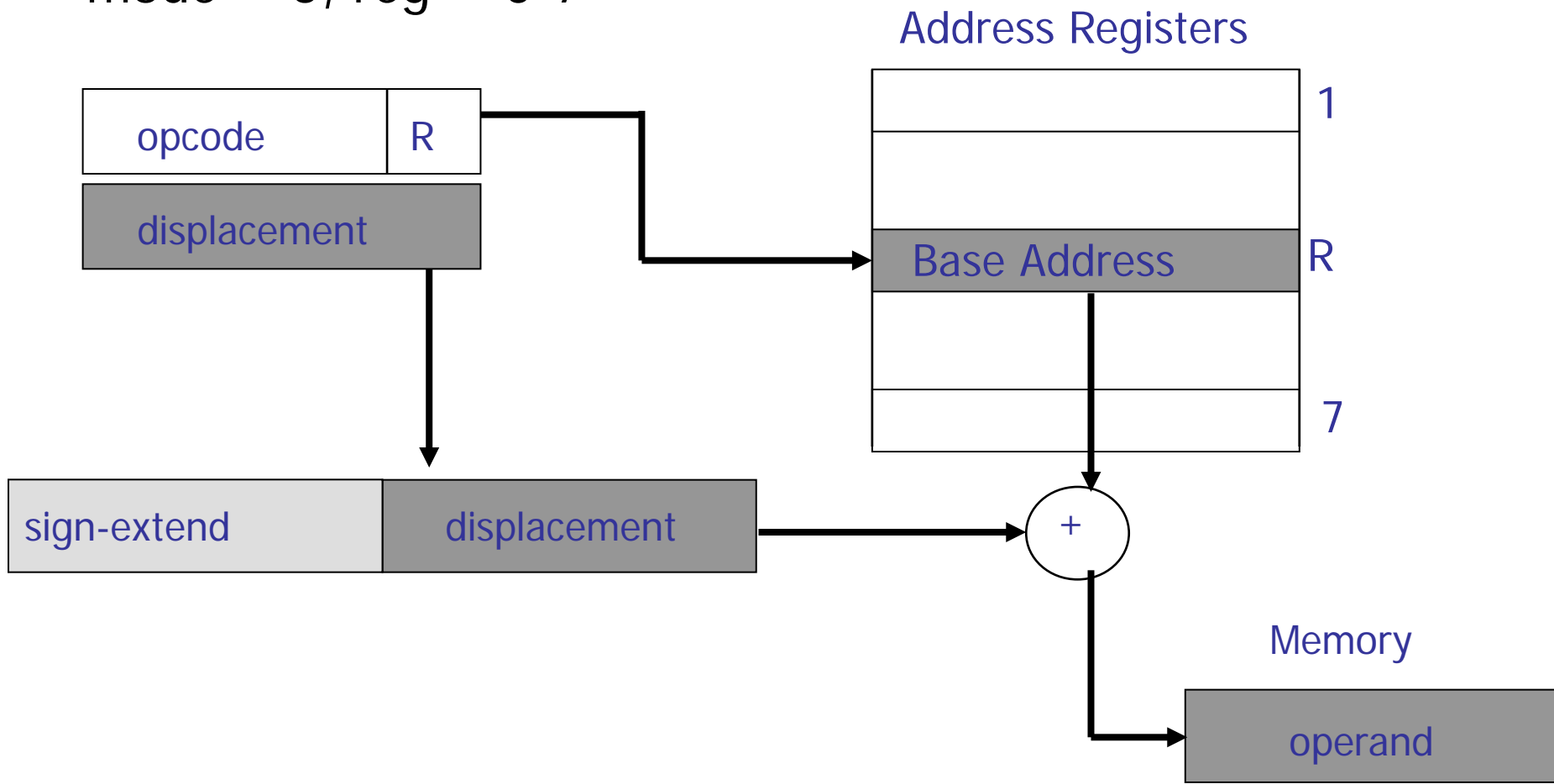


Based Addressing

- 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

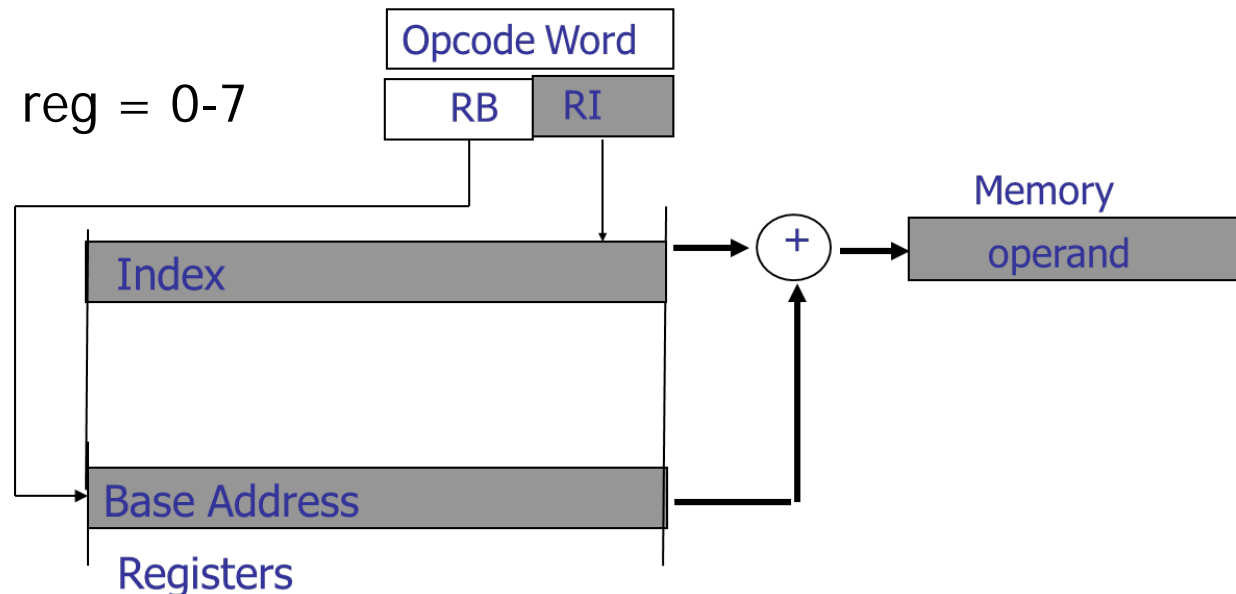
Based Addressing - Codifica

- mode = 5, reg = 0-7



Based Indexed

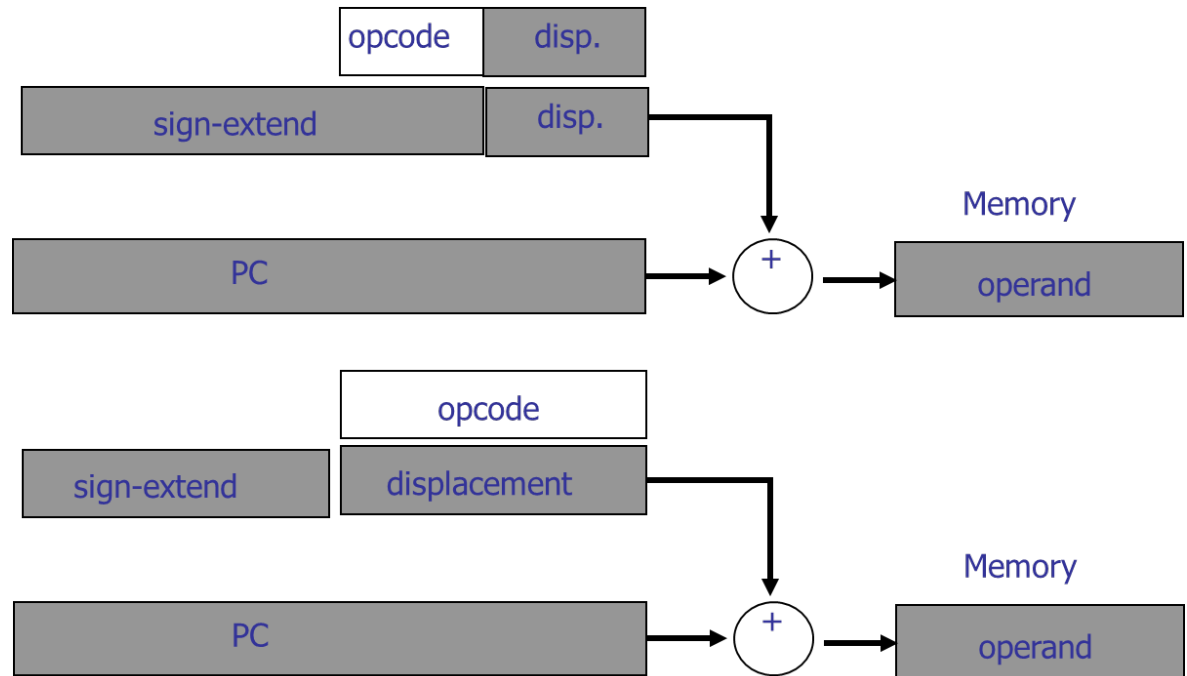
- Based Indexed Addressing: EA given by the sum of two components:
 - *base address*
 - *displacement*
- Useful for array and tables
- Coldfire (MC68000) supports both Short Based Indexed and Long Based Indexed
- Encoding:
 - mode = 6, reg = 0-7



Relative Addressing

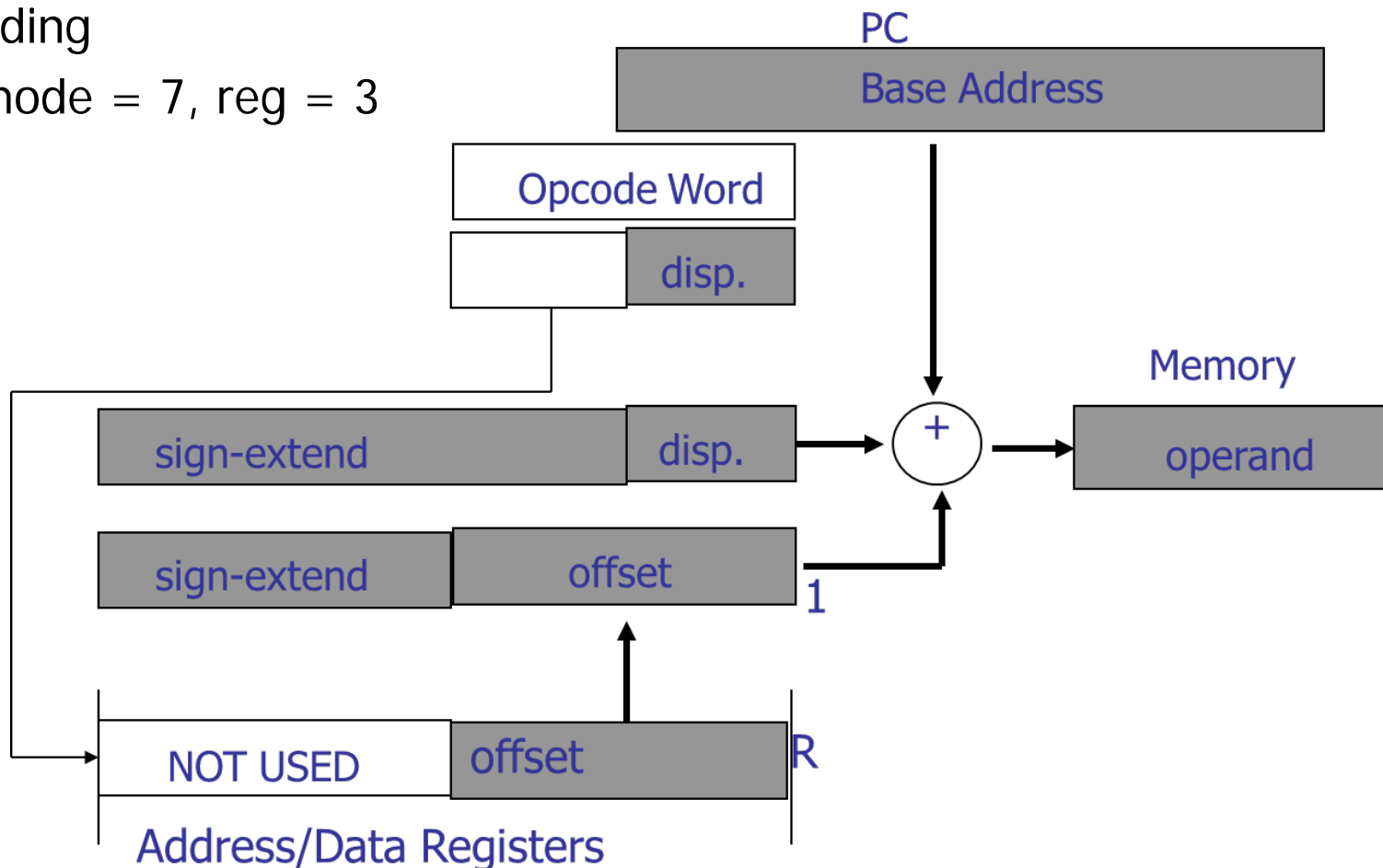
- (Relative to the PC)
- The EA is given by adding the displacement to the PC value
- Often small displacements, 8 or 16 bits, to point to an instruction next to the current one (instead of absolute 32 bits addresses)
- Encoding

– 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



Processor Structure

Tabella A2.1 Memoria e registri nei processori NIOS II, ColdFire, ARM e IA-32

Caratteristica	NIOS II	ColdFire	ARM	IA-32
Architettura	RISC	CISC	RISC	CISC
Lunghezza di parola	32 bit	16 bit	32 bit	32 bit
Lunghezza d'istruzione	1 parola	1 ÷ 3 parole	1 parola	1 ÷ 12 byte
Spazio degli indirizzi	2^{32} byte	2^{32} byte	2^{32} byte	2^{32} byte
Lunghezze di dati (byte)	1,2,4	1,2,4	1,2,4	1,2,4,8,16
Ordinamento dei byte	decresc.	crescente	opzione	decresc.
Registri	r0-r31	A0-A7,PC, D0-D7,SR	R0-R15, CPSR	si veda Caso di Studio C2

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 spiazamento	$X(r_i)$	$W(A_i)$	$[R_i, \#Val]$	$[R+X]$
Con indice e spiaz.				$[R_x * S + X]$
Con base e indice			$[R_i, \pm R_j, s]$	$[R + R_x * S]$
Con base, indice e spiaz.		$B(A_i, R_j)$		$[R + R_x * S + X]$
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 spiaz.			$[R_i, \#Val]!$	
Con post-base e spiaz.			$[R_i], \#Val$	
Con pre-base e indice			$[R_i, \pm R_j, s]!$	
Con post-base e indice			$[R_i], \pm R_j, s$	

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 spiazamento;
- 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 spiazamento: 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 spiazamento);
- S fattore di scala (IA-32): 1, 2, 4 o 8;
- s scorrimento logico (ARM): ds #vs dove ds \in {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

Begin execution here →

Address

Contents

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 → **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

Program
loop

LOOP

Load R2, N

Clear R3

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)

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	LOOP:	LD CLR MOV LD ADD ADD SUB BGT ST	R2, N R3 R4, #NUM1 R5, (R4) R3, R3, R5 R4, R4, #4 R2, R2, #1 R2, R0, LOOP R3, SUM
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Tabella A2.4 Direttive di assembler 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	Dt n DUP(v)
.byte	.byte	.byte dc.b	DCB	DB
.short .hword .word	.hword	.short dc.w	DCWa	DW
.long .int .word	.word	.long dc.l	DCDa	DD
.quad			DCQa	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.

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 \pm 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:
 - LShiftLR3, 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:

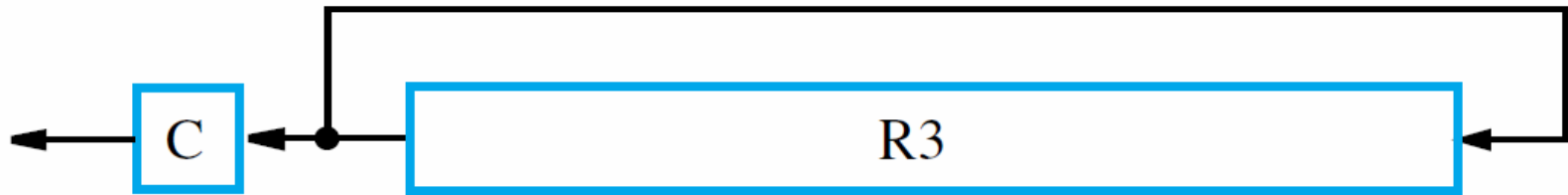
1

1	1	0	·	·	·	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---

(a) Logical shift left

LShiftL R3, R3, #2





before:

0

0	1	1	1	0	·	·	·	0	1	1
---	---	---	---	---	---	---	---	---	---	---

after:

1

1	1	0	·	·	·	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---

(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 $2n$ bits
- Processor truncates product to fit in a register:
Multiply R_k, R_i, R_j ($R_k \leftarrow [R_i] \times [R_j]$)
- For general case, 2 registers may hold result
- Integer division produces quotient as result:
Divide R_k, R_i, R_j ($R_k \leftarrow [R_i] / [R_j]$)
- 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, R_i

Add $R_j, 16(R_k)$

Additional Addressing Modes

- *Autodecrement* mode: before accessing operand, register contents are decremented, then new contents provide effective address
- Notation in assembly language:
Add $R_j, -(R_i)$
- 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

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 *Rd*, *Rn*, *Rm* or #offset

ADD R0, R2, R4

performs

$R0 \leftarrow [R2] + [R4]$

SUB R0, R3, #17

performs

$R0 \leftarrow [R3] - 17$

(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 Rd , Rm or #value

loads the bit-complement of [Rm] or value
into Rd

Instructions

- Implementing **Shift** and **Rotate** instructions:

MOV $R_i, R_j, \text{LSL } \#4$

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 Rn and Rm and writes the result into register Rd

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 Rm are complemented before they are ANDed with the bits of Rn

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

BIC R0, R0, R1

results in 02FA0000 being written into R0

Instructions

- Test:

TST Rn, Rm or #value
performs bit-wise logical AND of the two operands, then
sets condition code flags

TST $R3, \#1$
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 Rn, Rm

performs

$[Rn] - [Rm]$

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

ADD**S** updates flags, but ADD does not

Instructions

- Adding 64-bit operands

ADC R0, R1, R2 (Add with carry)
performs $R0 \leftarrow [R1] + [R2] + [C]$

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

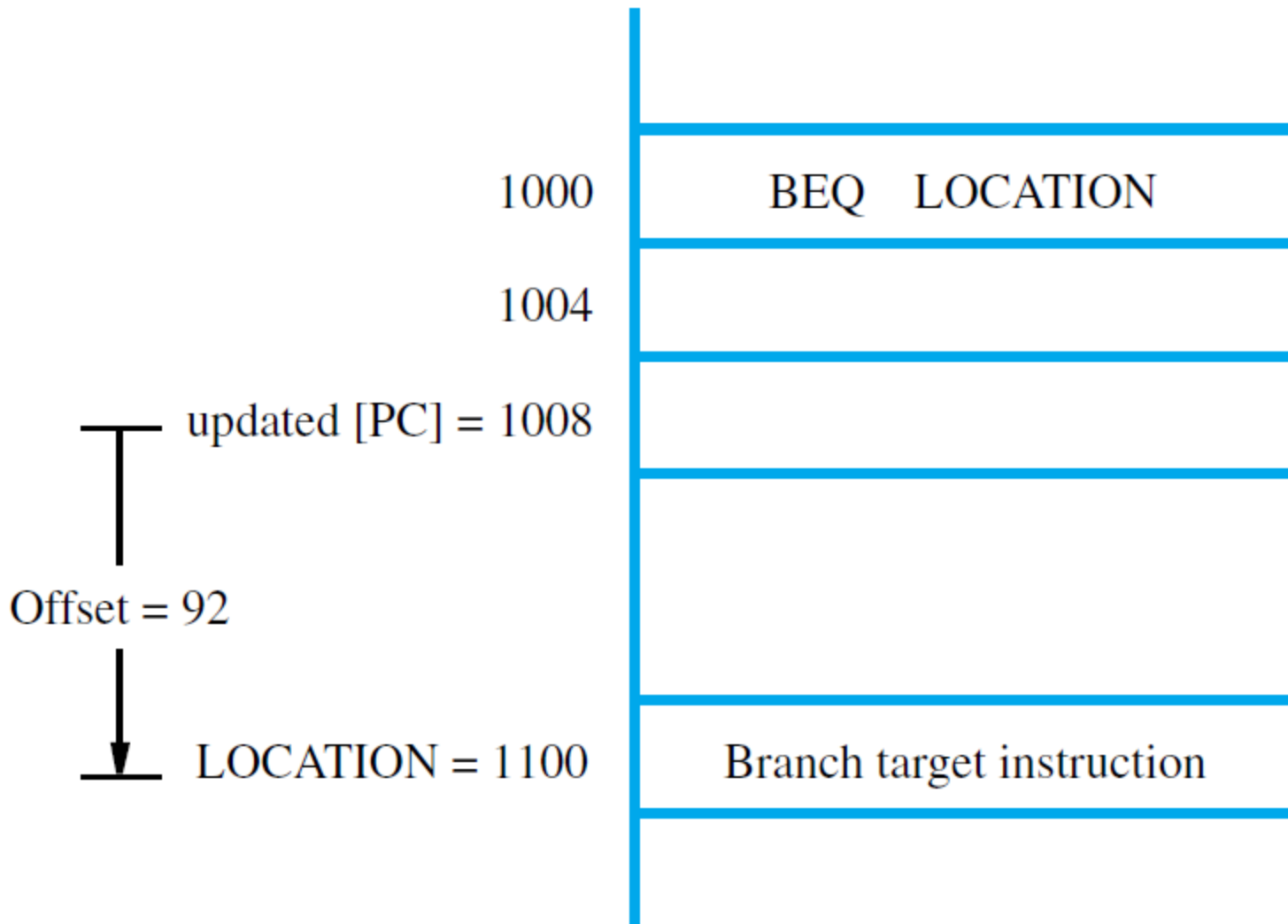
- Branch:

B{condition} LOCATION

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

BEQ LOCATION

branches if $Z = 1$



Condition field encoding in ARM instructions.

Condition field $b_{31} \dots b_{28}$	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	$\bar{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\bar{C} \vee Z = 1$
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 ENTRY	CODE
Statements that generate machine instructions	LOOP	LDR LDR MOV LDR ADD SUBS BGT STR	R1, N R2, POINTER R0, #0 R3, [R2], #4 R0, R0, R3 R1, R1, #1 LOOP R0, SUM
Assembler directives	SUM N POINTER NUM1	AREA DCD DCD DCD DCD END	DATA 0 5 NUM1 3, -17, 27, -12, 322

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    Rd, =value
```

loads a 32-bit value into *Rd*

```
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

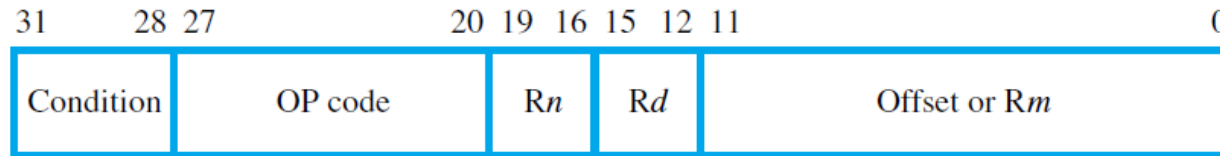
SUB r1,r1,r0

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 \geq)
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 \leq)
GE	N and V the same	Signed \geq
LT	N and V differ	Signed $<$
GT	Z clear, N and V the same	Signed $>$
LE	Z set, N and V differ	Signed \leq
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 $65535 \times 1 = 65535$

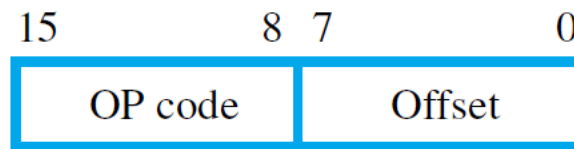
Branch and Jump Instructions

- 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

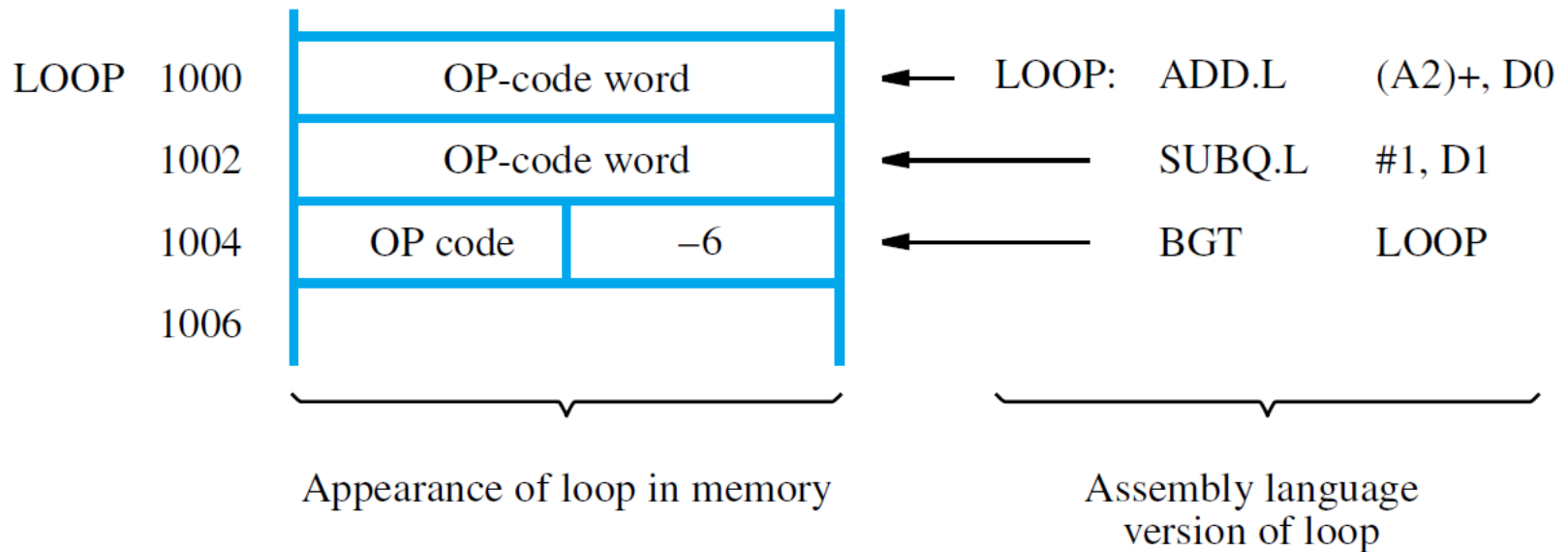
<i>cc</i>	Name	Test condition
HI	High	$C \vee Z = 0$
LS	Low or same	$C \vee Z = 1$
CC	Carry clear	$C = 0$
CS	Carry set	$C = 1$
NE	Not equal	$Z = 0$
EQ	Equal	$Z = 1$
VC	Overflow clear	$V = 0$
VS	Overflow set	$V = 1$
PL	Plus	$N = 0$
MI	Minus	$N = 1$
GE	Greater or equal	$N \oplus V = 0$
LT	Less than	$N \oplus V = 1$
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 <i>n</i> .
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	<i>ldb ri, X(rj)</i>		LDR <i>b</i>	
Store	<i>stb ri, X(rj)</i>		STR <i>b</i>	
Move	<i>mova</i>	MOVE <i>a.b</i>	MOV, MVN	MOV, LEA, PUSH, POP
Multiplo		MOVEM. <i>b</i>	LDM <i>w</i> , STM <i>w</i>	POPAD, PUSHAD
Controllo				
Salto incondiz.	<i>br l, jmp ri</i>	JMP	B <i>l</i>	JMP
Salto condiz.	<i>bc ri, rj, l</i>	Bc <i>l</i>	Bc <i>l</i>	Jc <i>l</i> LOOP <i>l</i>
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	<i>addm</i>	ADD <i>m.L</i>	ADD <i>f</i> , ADC	ADD, ADC
Sottrazione	<i>subm</i>	SUB <i>m.L</i> NEG <i>m.L</i>	SUB <i>f</i> , SBC	SUB, SBB NEG
Moltiplicazione	<i>mulm</i>	MUL <i>s.b</i>	MUL, MLA	IMUL
Divisione	<i>div, divu</i>	DIV <i>s.b</i>		IDIV
Resto		REMs.L		
Altre		EXT. <i>b</i> CLR. <i>b</i>		INC, DEC
Confronto	<i>cmpcm</i>	CMP <i>m.L</i>	CMP, CMN	CMP
Logiche				
Congiunzione	<i>andm, andhi</i>	AND <i>m.L</i>	AND, TST	AND
Disgiunzione	<i>orm, orhi</i>	OR <i>m.L</i>	ORR	OR
Disg. esclusiva	<i>xorm, xorhi</i>	EOR <i>m.L</i>	EOR, TEQ	XOR
Negazione		NOT <i>m.L</i>		NOT
Altre	<i>nor</i>		BIC	
Scorrimento				
Logico	<i>srlm</i> <i>sllm</i>	LSR.L LSL.L	<i>t</i> , LSR <i>t</i> , LSL	SHR SHL
Aritmetico	<i>sram</i>	ASR.L ASL.L	<i>t</i> , ASR	SAR SAL
Rotazione	<i>ror</i> <i>rolm</i>		<i>t</i> , ROR	ROR, RCR 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.



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 $R_j, (SP)$
- Pop operation also involves two instructions:
 - Load $R_j, (SP)$
 - Add $SP, SP, \#4$

Subroutines

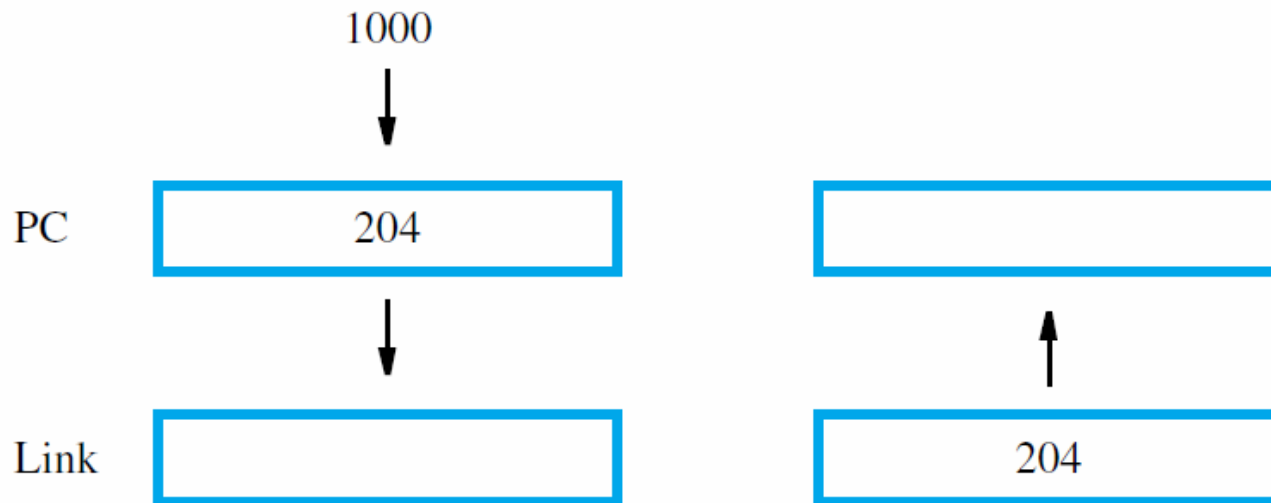
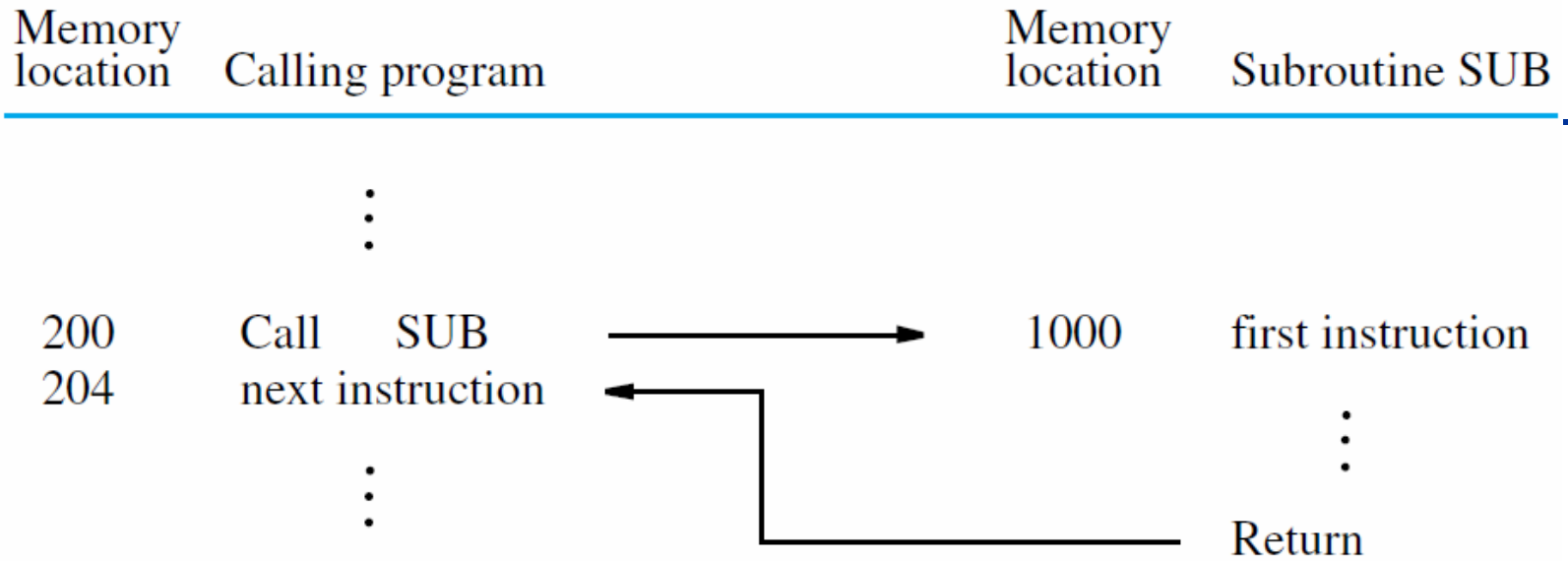
- In a given program, a particular task may be executed many times using different data
- Examples: mathematical function, list sorting
- Implement task in one block of instructions
- This is called a **subroutine**
- Rather than reproduce entire subroutine block in each part of program, use a subroutine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



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

FP
(frame pointer)



saved [FP]

param1

param2

param3

param4

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 descending
 - 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

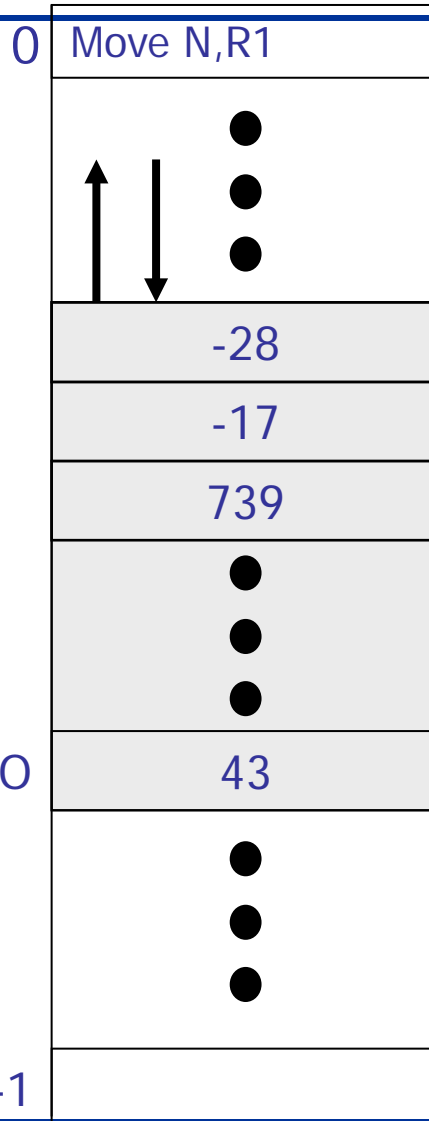
Calling program

MOVEA.L	#NUM1, A2	Put the address NUM1 in A2.
MOVE.L	N, D1	Put the number of elements <i>n</i> 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



@@@ - SP punta all'ultima locazione occupata

Push:

Decrement SP

Move NEWITEM, (SP)

Pop:

Move (SP), ITEM

Increment SP

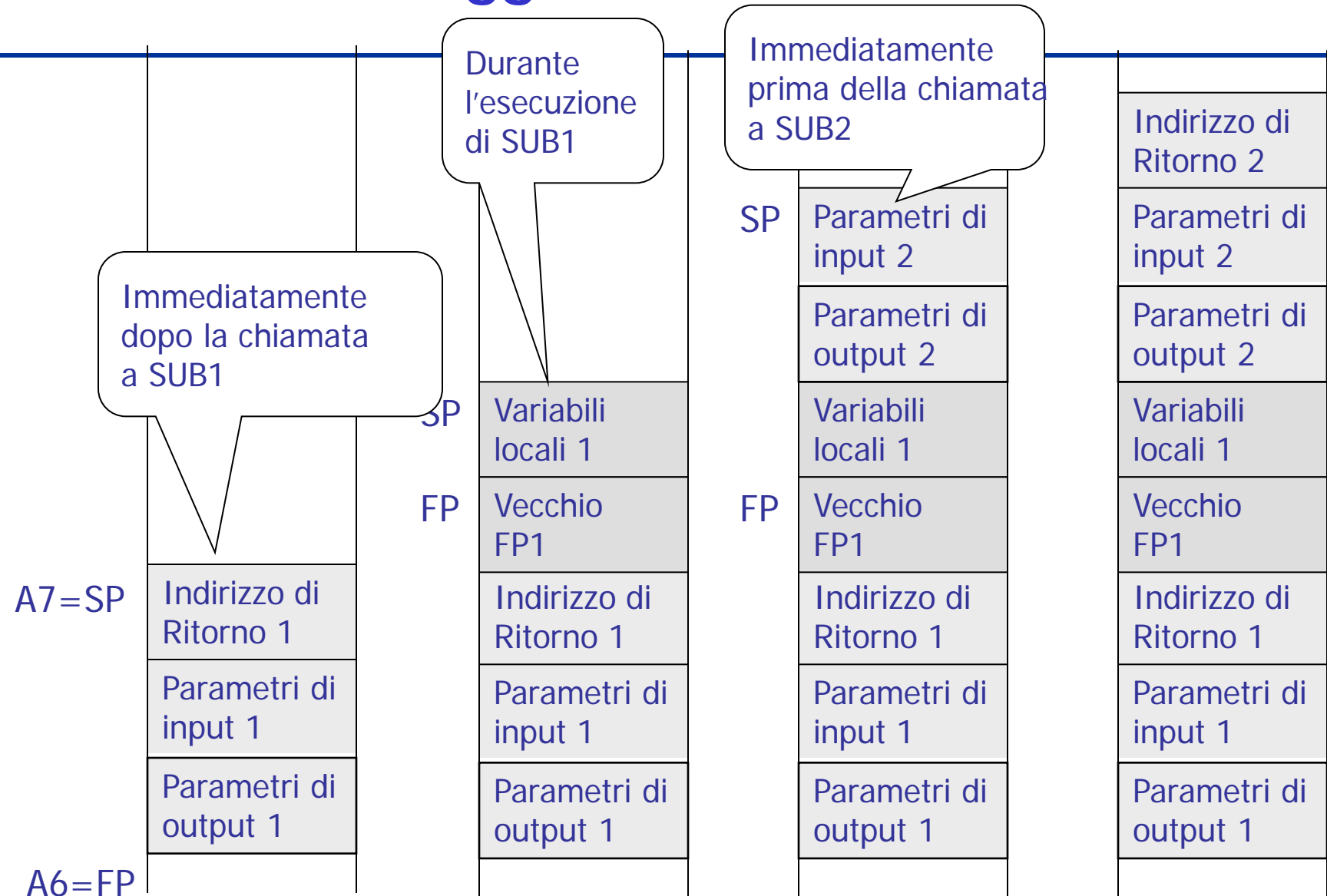
Push:

Move NEWITEM, -(SP)

Pop:

Move (SP)+, ITEM

Passaggio Parametri su Stack



Link and allocate

- Sintassi:
 - 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
```

Codice programma principale

*Area Dati

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

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

The screenshot shows the ASIM simulation environment. The main window displays a memory dump with addresses and hex values. A sub-window titled 'simple: Memoria 2' is active, showing a memory dump starting at 00008000. Below this, a window titled 'simple: M68000 1' is open, displaying assembly code and stack parameters. The assembly code includes 'ORG \$8000' and 'OUT EQU 0'. The stack parameters are listed as follows:

```
*****  
* Wakerly - Table 9-16 Page 332  
*  
* input:  
*  
* output:  
*  
*****  
ORG      $8000  
OUT      EQU      0  
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  XNZVCI  A7':00009200  
|00000000| | ISR:0010011100000000 | PC:00000000
```

For Help, press F1



Ritorno – Esecuzione

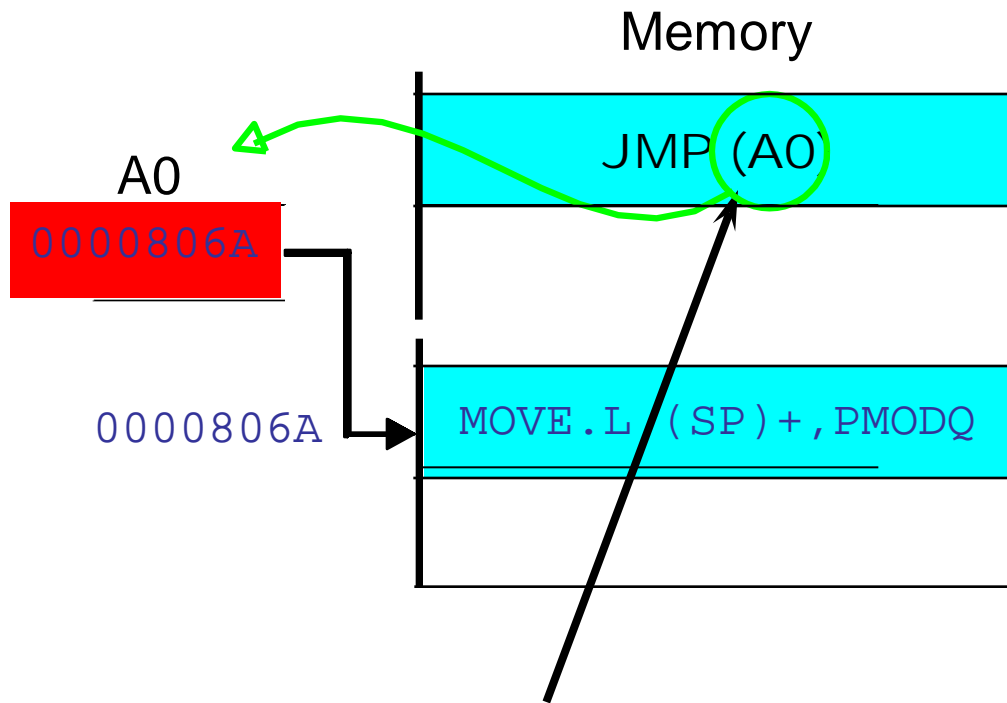
The screenshot displays the ASIM simulator window titled "ASIM - simple.cfg". The main window shows assembly code for "simple: Memoria 2" and "simple: M68000 1". The code includes instructions like "MOVE.L D1,QUOT(A6)", "CLNSTK UNLK", "MOVEA.L (SP)+,A0", "ADDA.L #STATUS-DUSR,SP", and "JMP (A0)". A comment block follows: "*****", "* Main Program: COMPUTES PDIVQ:=P DIV Q; P MOD Q;", "* Wakerly - Table 9-17 Page 336". Below the code, a table of register values is shown:

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	IT S INT	XNZVCI	A7':000091EA
100000B721	ISR:00100111000000001	PC:00008048	

At the bottom left, it says "For Help, press F1".



Ritorno - Schema



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

L'istruzione specifica l'operando come (A0)

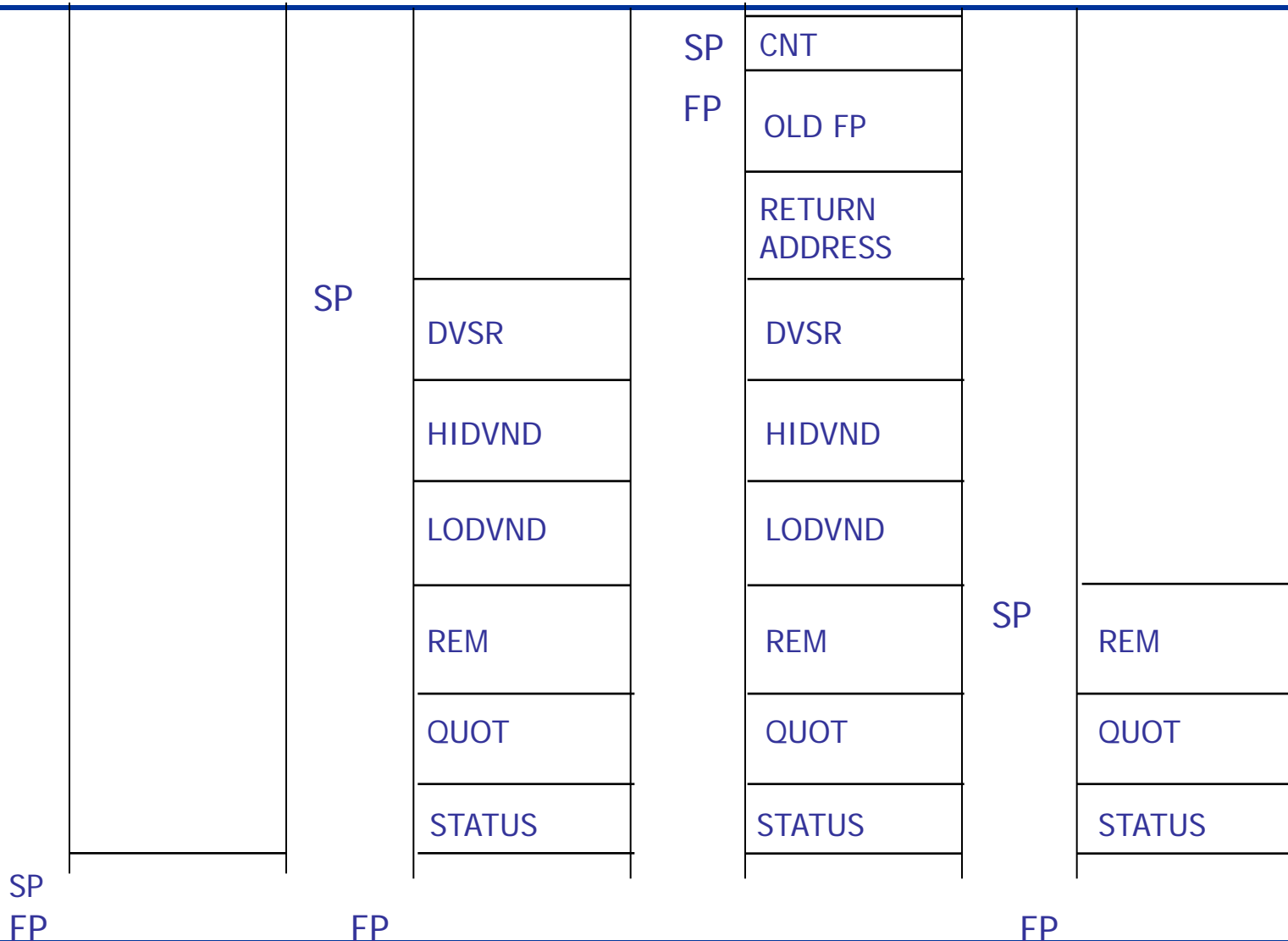
Ritorno – Mappa della memoria

```
div916.lis - WordPad
File Modifica Visualizza Inserisci Formato ?
00008048 DFFC 00000014 44 ADDA.L #STATUS-DVSR,SI
0000804E 4ED0 45 JMP (AO)
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	Aj	R12 FP	EBP
LR	r31 ra		R14 LR	
Istruzioni di chiamata e di rientro	call / callr ri ret	BSR / JSR a RTS	BL / LDMFD SP!, {r,PC}	CALL / RET
Altre istruzioni rilevanti		MOVEM.L r, (A7) LINK Aj, #p UNLK Aj	STMFD SP!, {r,LR}	PUSH s POP d PUSHAD POPAD

Legenda:

- Aj* 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:

```
for    $i \leftarrow 0$  to  $n - m$  do  
       $j \leftarrow 0$   
      while  $j < m$  and  $P[j] = T[i + j]$  do  
         $j \leftarrow j + 1$   
      if  $j = m$  return  $i$   
return  $-1$ 
```

	Move	R2, #T	R2 points to string <i>T</i> .
	Move	R3, #P	R3 points to string <i>P</i> .
	Load	R4, N	Get the value <i>n</i> .
	Load	R5, M	Get the value <i>m</i> .
	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 <i>T</i> .
	Move	R7, R3	Use R7 to scan through string <i>P</i> .
LOOP2:	LoadByte	R8, (R6)	Compare a pair of
	LoadByte	R9, (R7)	characters in
	Branch_if_ $[R8] \neq [R9]$	NOMATCH	strings <i>T</i> and <i>P</i> .
	Add	R6, R6, #1	Point to next character in <i>T</i> .
	Add	R7, R7, #1	Point to next character in <i>P</i> .
	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 <i>T</i> .
	Branch_if_ $[R4] \geq [R2]$	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 $\neq 0$	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