UNIT-1.
Basic Structure of Computers

Dr. A. M. Chandrashekhar
Dept. of CS & E
SJCE - Mysore

SYLLABUS

- **Basic Structures of Computers, Machine Instructions & Programs:** Computer Types, Functional Units, Basic Operational Concepts, Number Representation, and Arithmetic Operations.
- **Instruction Set Architecture:** Memory Locations and Addresses, Memory Operations, Instructions and Instruction Sequencing, Addressing Modes, Assembly Language, Stacks Subroutines, Additional Instructions, RISC and CISC Styles

Classes of Computers

- Desktop/laptop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Workstations
  - More computing power used in engg. applications, graphics etc.
- Enterprise System/ Mainframes
  - Used for business data processing
- Server computers (Low End Range)
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- Supercomputer (High End Range)
  - Large scale numerical calculation such as weather forecasting, aircraft design
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

Functional Units

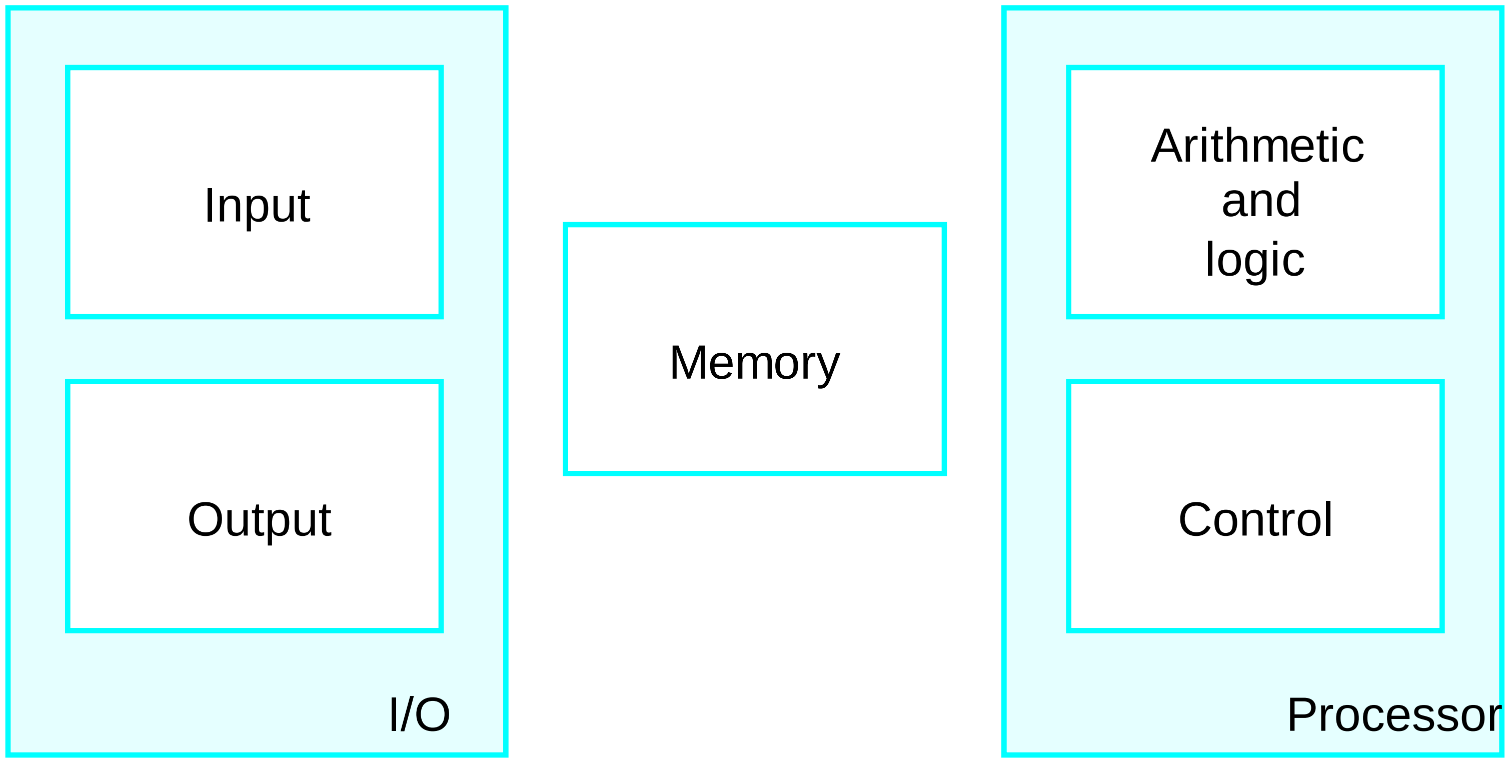


Figure 1.1. Basic functional units of a computer.

Information Handled by a Computer

- Instructions/machine instructions
  - Govern the transfer of information within a computer as well as between the computer and its I/O devices
  - Specify the arithmetic and logic operations to be performed
  - Program
- Data
  - Used as operands by the instructions
  - Source program
- Encoded in binary code – 0 and 1

Memory Unit

- Store programs and data

- Two classes of storage: Primary & secondary
  - Primary storage
  - Fast
  - Programs must be stored in memory while they are being executed
  - Large number of semiconductor storage cells
  - Processed in words
  - Address
  - RAM and memory access time
  - Memory hierarchy – cache, main memory

  - Secondary storage
    - Larger in size
    - Slow and cheaper

Arithmetic and Logic Unit (ALU)

- Most computer operations are executed in ALU of the processor.
- Load the operands into memory
  - bring them to the processor
  - perform operation in ALU
  - store the result back to memory or retain in the processor.
- Registers

Control Unit

- All computer operations are controlled by the control unit.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Control unit is usually distributed throughout the machine instead of standing alone.

Operations of a computer

- Accept information in the form of programs and data through an input unit and store it in the memory
- Fetch the information stored in the memory, under program control, into an ALU, where the information is processed
- Output the processed information through an output unit
- Control all activities inside the machine through a control unit

A Typical Instruction execution

- Add LOCA, R0 ; R0 ← R0+[LOCA)]
  - Add the operand at memory location LOCA to the operand in a register R0 in the processor.
  - Place the sum into register R0.
  - The original contents of LOCA are preserved.
  - The original contents of R0 is overwritten.
  - Steps
    - Instruction is fetched from the memory into the processor –
    - the operand at LOCA is fetched and added to the contents of R0
    - the resulting sum is stored in register R0.

Connection Between the Processor and the Memory

- Instruction register (IR)
- Program counter (PC)
- General-purpose register (R_0 – R_{n-1})

- Memory address register (MAR)
- Memory data register (MDR)

Typical Operating Steps

- Programs reside in the memory through input devices
- PC is set to point to the first instruction
- The contents of PC are transferred to MAR
- A Read signal is sent to the memory
- The first instruction is read out and loaded into MDR
- The contents of MDR are transferred to IR
- Decode and execute the instruction

Typical Operating Steps (Cont')

- Get operands for ALU
  - General-purpose register
  - Memory (address to MAR – Read – MDR to ALU)
- Perform operation in ALU
- Store the result back
  - To general-purpose register or
  - To memory (address to MAR, result to MDR – Write)
- During the execution, PC is incremented to the next instruction

Interrupt

- Normal execution of programs may be preempted if some device requires urgent servicing.

- The normal execution of the current program must be interrupted – the device raises an *interrupt* signal.

- Interrupt-service routine(ISR)

- Current system information
  - backup and restore (PC, general-purpose registers, control information, specific information)

Bus Structures

- There are many ways to connect different parts inside a computer together.
- A group of lines that serves as a connecting path for several devices is called a *bus*.
- Address/data/control

- Single-bus

Speed Issue

- Different devices -- different transfer/operate speed.
- If the speed of bus is bounded by the slowest device connected to it, the efficiency will be very low.
- How to solve this?
  - A common approach – use buffers.

CISC and RISC

- Complex Instruction Set Computer (CISC):
  - A computer with large no. of instructions (100 or more).

- Reduced Instruction Set Computer (RISC):
  - Fewer instructions with simple constructs.

Characteristics of CISC

- A large no. of instructions (100 to 250)
- A large variety of addressing modes (5 to 20)
- Variable length instruction format
- Instructions that manipulate operands in memory.
- Some instruction that perform specialized tasks and are used infrequently.
- Support more complex HLLs
- Improve execution efficiency
  - Complex operations in microcode (Micro-programmed CU)

Characteristics of RISC

- Relatively few instructions
- Relatively few addressing modes
- Memory access limited to LOAD & STORE
- All operations done within the registers of the CPU.
- Fixed-length, easily decoded instruction format
- Single-cycle instruction execution (few Steps)
- Hardwired rather than micro-programmed CU

Arithmetic Operations

Number representation

SIGNED INTEGER

- 3 major representations:

  Sign and magnitude

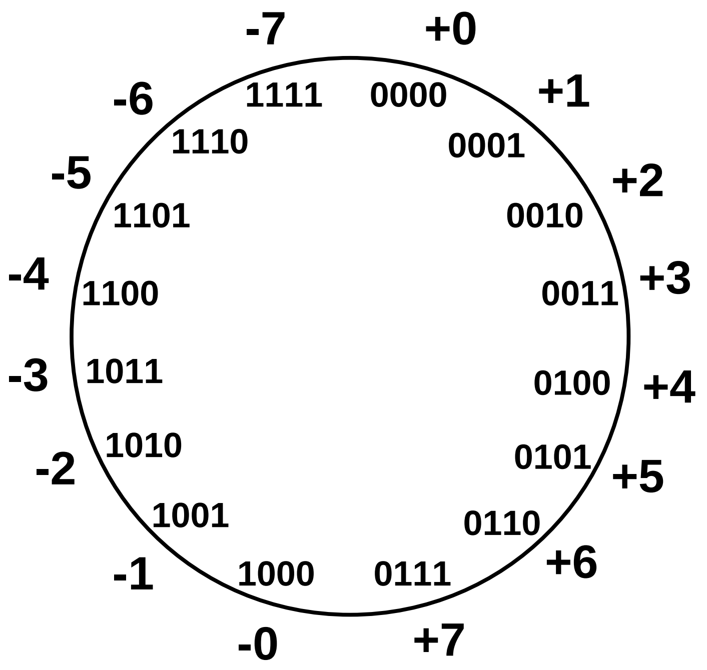
  One's complement

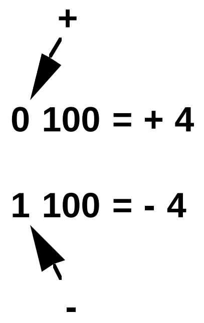
  Two's complement

- Assumptions:

  4-bit machine word → 16 different values can be represented, Roughly half are positive, half are –ve

Sign and Magnitude Representation





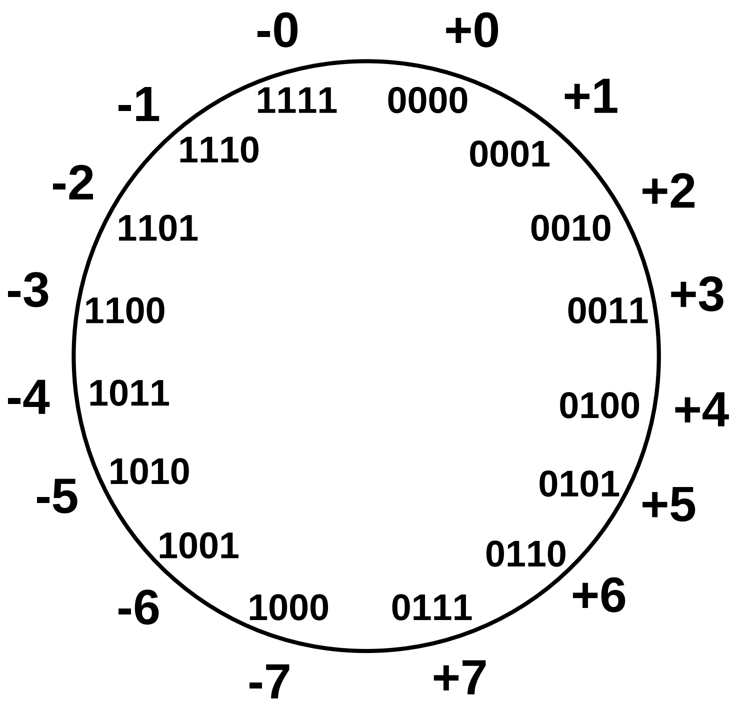
High order bit is sign: 0 = positive (or zero), 1 = negative

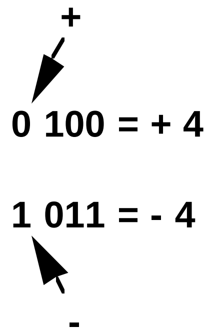
Three low order bits is the magnitude: 0 (000) thru 7 (111)

Number range for n bits = +/- 2^{n-1} -1

Two representations for 0

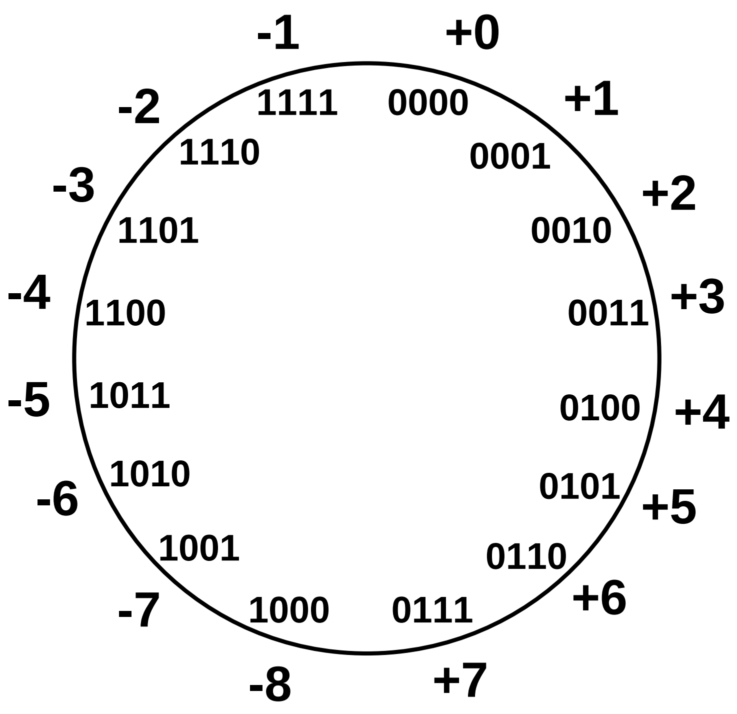
One's Complement Representation





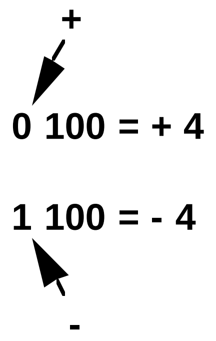
- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

Two's Complement Representation



1's comp + 1

Is 2's Complement



- **Only one representation for 0**
- **One more negative number than positive number**

Binary, Signed-Integer Representations

Page 28

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Figure 2.1. Binary, signed-integer representations.

Addition and Subtraction – 2's Complement

If carry-in to the high order bit =carry-out then Ignore carry

```
  4     0100        -4      1100
+ 3     0011     + (-3)     1101
----    ----     ------    -----
  7     0111        -7     11001
```

if carry-in differs from carry-out then overflow

```
  4     0100        -4      1100
- 3     1101       + 3      0011
----    -----      ---      ----
  1    10001        -1      1111
```

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

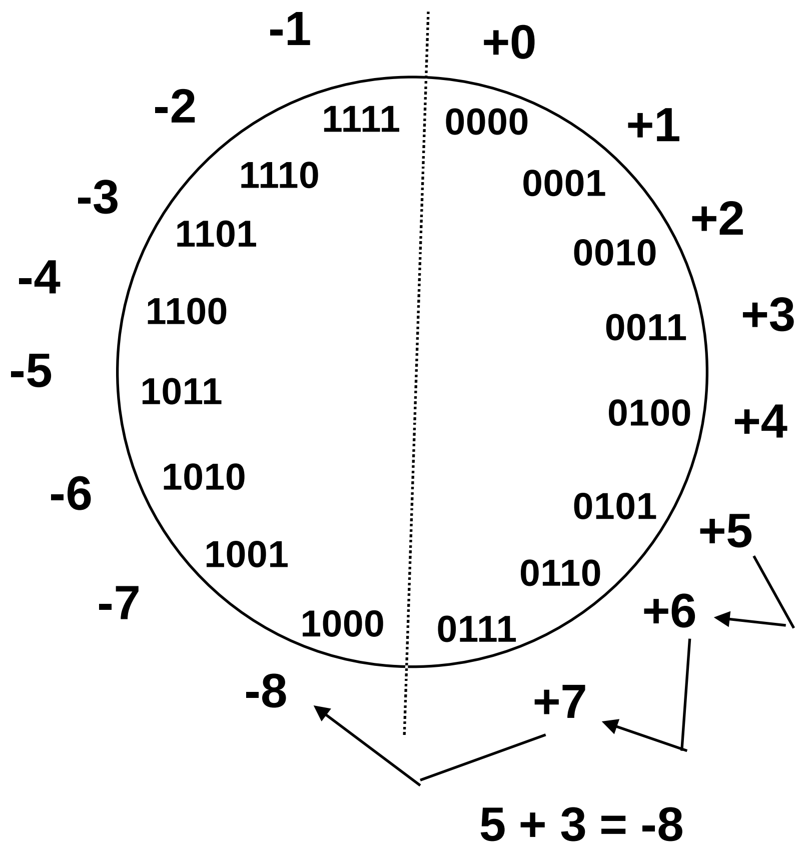
2's-Complement Add and Subtract Operations

Page 14

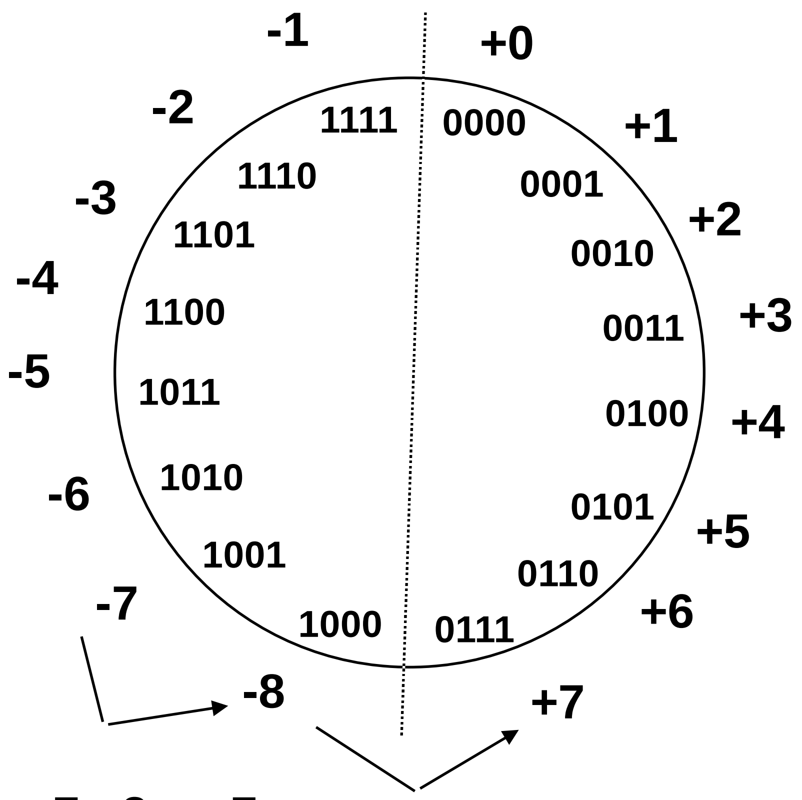
```
(a)     0 0 1 0    (+2)          (b)       0 1 0 0    (+4)
      + 0 0 1 1    (+3)                  + 1 0 1 0    (- 6)
      ---------    ----                  ---------    -----
        0 1 0 1    (+5)                    1 1 1 0    (- 2)

(c)     1 0 1 1    (- 5)         (d)       0 1 1 1    (+7)
      + 1 1 1 0    (- 2)                 + 1 1 0 1    (- 3)
      ---------    -----                 ---------    -----
        1 0 0 1    (- 7)                   0 1 0 0    (+4)

(e)     1 1 0 1    (- 3)                   1 1 0 1
      - 1 0 0 1    (- 7)      ⇒          + 0 1 1 1
      ---------    -----                 ---------    -----
                                           0 1 0 0    (+4)

(f)     0 0 1 0    (+2)                    0 0 1 0
      - 0 1 0 0    (+4)       ⇒          + 1 1 0 0
      ---------    ----                  ---------    -----
                                           1 1 1 0    (- 2)

(g)     0 1 1 0    (+6)                    0 1 1 0
      - 0 0 1 1    (+3)       ⇒          + 1 1 0 1
      ---------    ----                  ---------    -----
                                           0 0 1 1    (+3)

(h)     1 0 0 1    (- 7)                   1 0 0 1
      - 1 0 1 1    (- 5)      ⇒          + 0 1 0 1
      ---------    -----                 ---------    -----
                                           1 1 1 0    (- 2)

(i)     1 0 0 1    (- 7)                   1 0 0 1
      - 0 0 0 1    (+1)       ⇒          + 1 1 1 1
      ---------    ----                  ---------    -----
                                           1 0 0 0    (- 8)

(j)     0 0 1 0    (+2)                    0 0 1 0
      - 1 1 0 1    (- 3)      ⇒          + 0 0 1 1
      ---------    -----                 ---------    -----
                                           0 1 0 1    (+5)
```

Figure 2.4. 2's-complement Add and Subtract operations.

Overflow - Add two positive numbers to get a negative number or two negative numbers to get a positive number



5 + 3 = -8



-7 - 2 = +7

Overflow Conditions

- When the actual result of an arithmetic operation is outside the representable range, an *arithmetic overflow has occurred.*

- However, this is not true when adding using 2's-complement representation.

- the value of the carry-out bit from the sign-bit position is not an indicator of overflow. Overflow may occur only if both summands have the same sign.

- The addition of numbers with different signs cannot cause overflow because the result is always within the representable range.

- when adding two numbers in 2's-complement representation. Examine the signs of the two summands and the sign of the result. When both summands have the same sign, an overflow has occurred when the sign of the sum is not the same as the signs of the summands.

Overflow Conditions

Overflow when carry-in to the high-order bit does not equal carry out

```
            0 1 1 1
   5          0 1 0 1
   3          0 0 1 1
  ---       ---------
  -8          1 0 0 0
```

Overflow

```
                1 0 0 0
  -7              1 0 0 1
  -2              1 1 0 0
  ---           ---------
   7            1|0 1 1 1
```

Overflow

```
            0 0 0 0
   5          0 1 0 1
   2          0 0 1 0
  ---       ---------
   7          0 1 1 1
```

No overflow

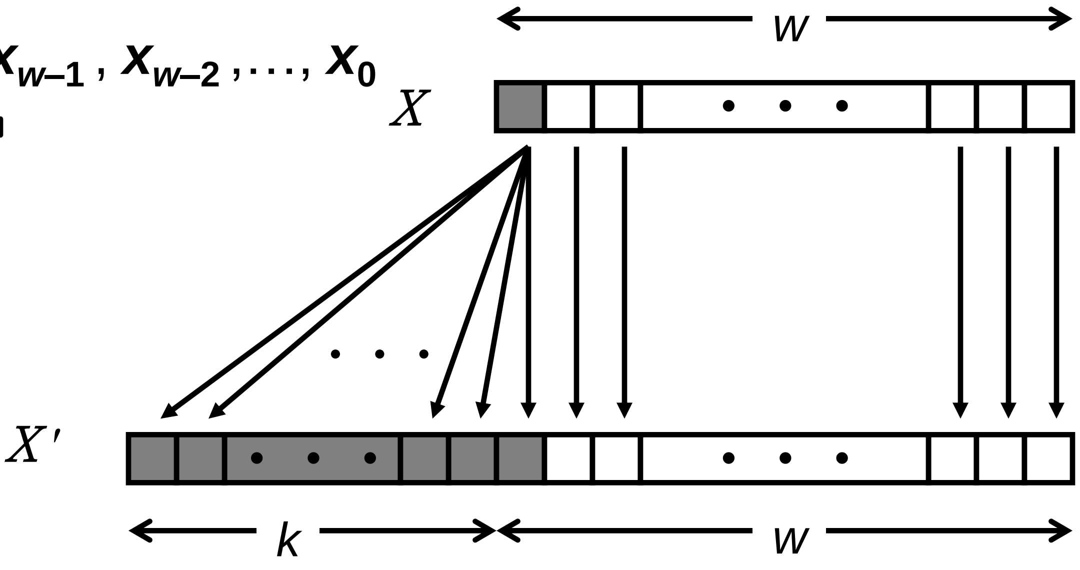
```
                1 1 1 1
  -3              1 1 0 1
  -5              1 0 1 1
  ---           ---------
  -8            1|1 0 0 0
```

No overflow

Sign Extension

To represent a signed number in 2's-complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called *sign extension*.

- Task:
  - Given *w*-bit signed integer *x* , Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Floating point Numbers

➢In integers, The implied binary point at the right end of the number, just after bit *b0.*

➢In the case of floating point numbers, the binary point is said to *float.*

➢A binary floating-point number can be represented by:
• a sign for the number
• some significant bits
• a signed scale factor exponent for an implied base of 2

We will discuss in detail in Unit4: Arithmetic operations

Character Representation

The most common encoding scheme for characters is ASCII (American Standard Code for Information Interchange).
Alphanumeric characters, operators, punctuation symbols, and control characters are represented by 7-bit codes

Memory Locations, Addresses, and Operations

Memory Location, Addresses, and Operation

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.

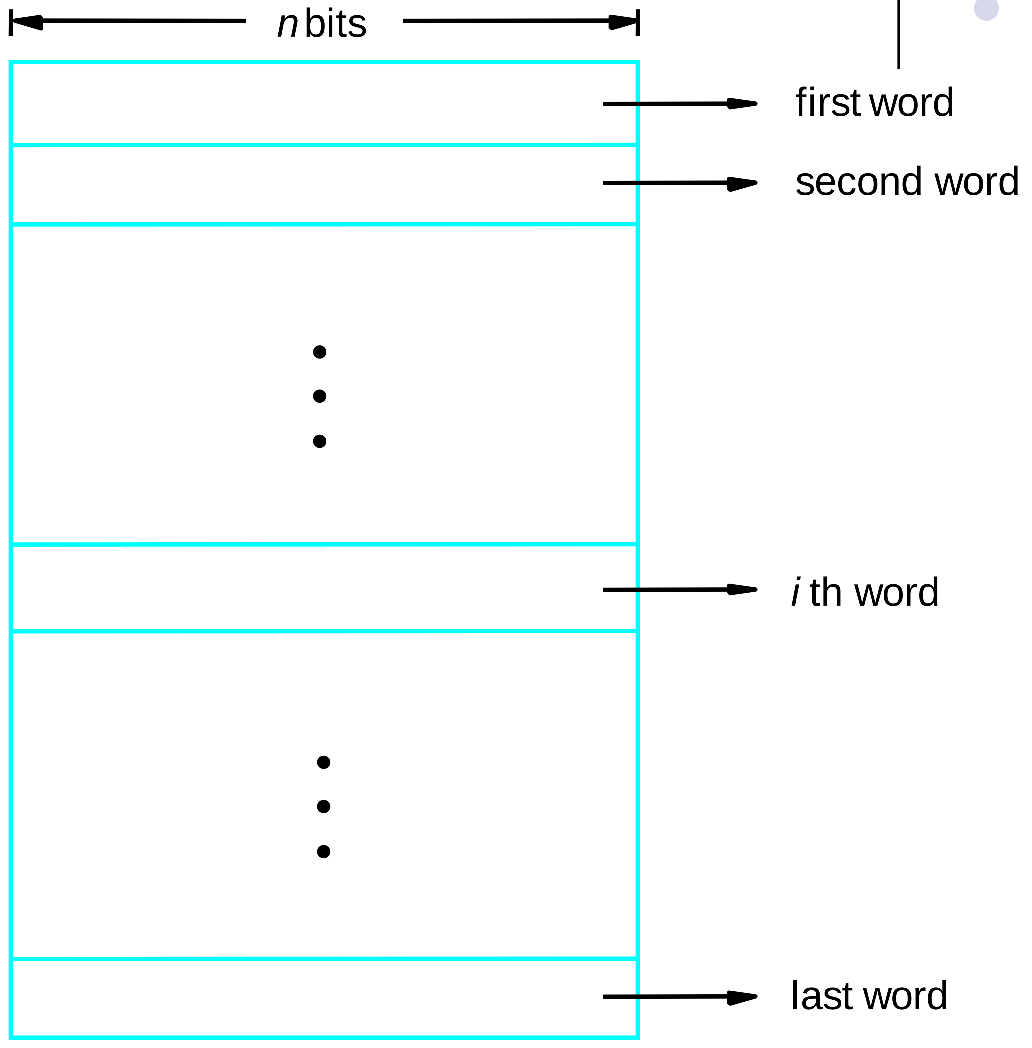
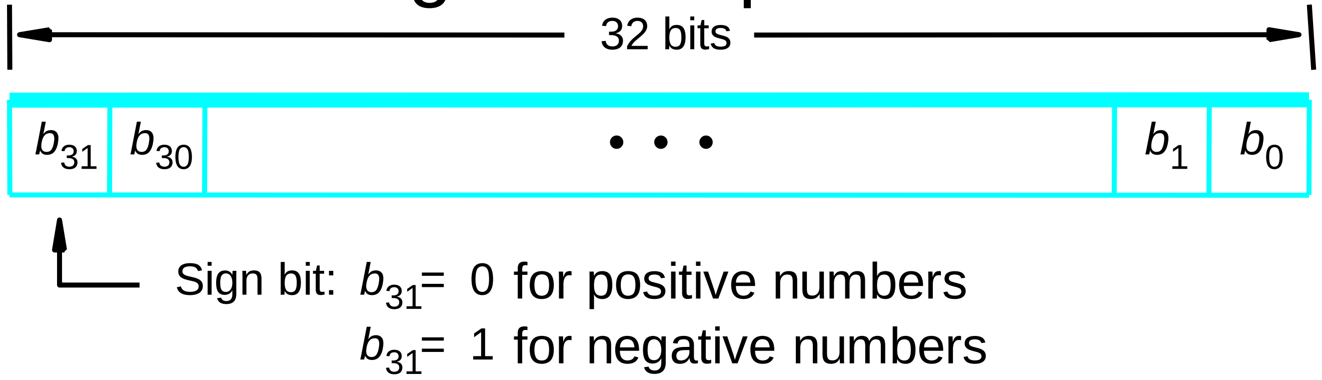


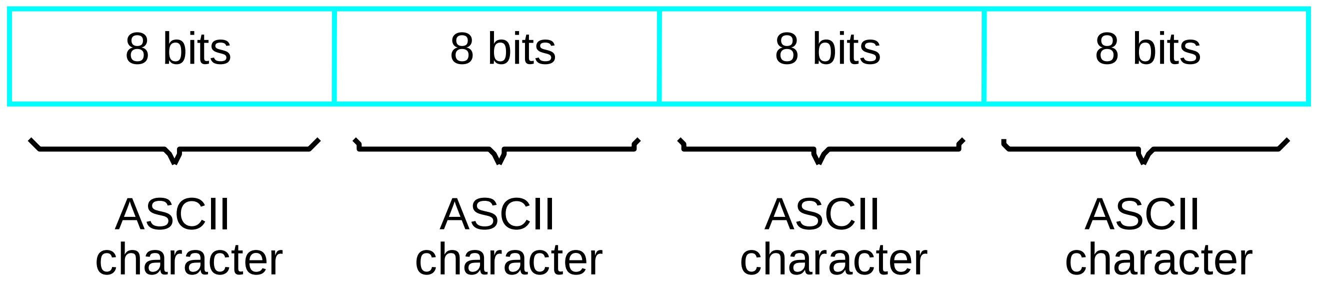
Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

- 32-bit word length example


32 bits
b_{31}
b_{30}
• • •
b_1
b_0
Sign bit: b_{31} = 0 for positive numbers
b_{31} = 1 for negative numbers

(a) A signed integer



(b) Four characters

Memory Location, Addresses, and Operation ……

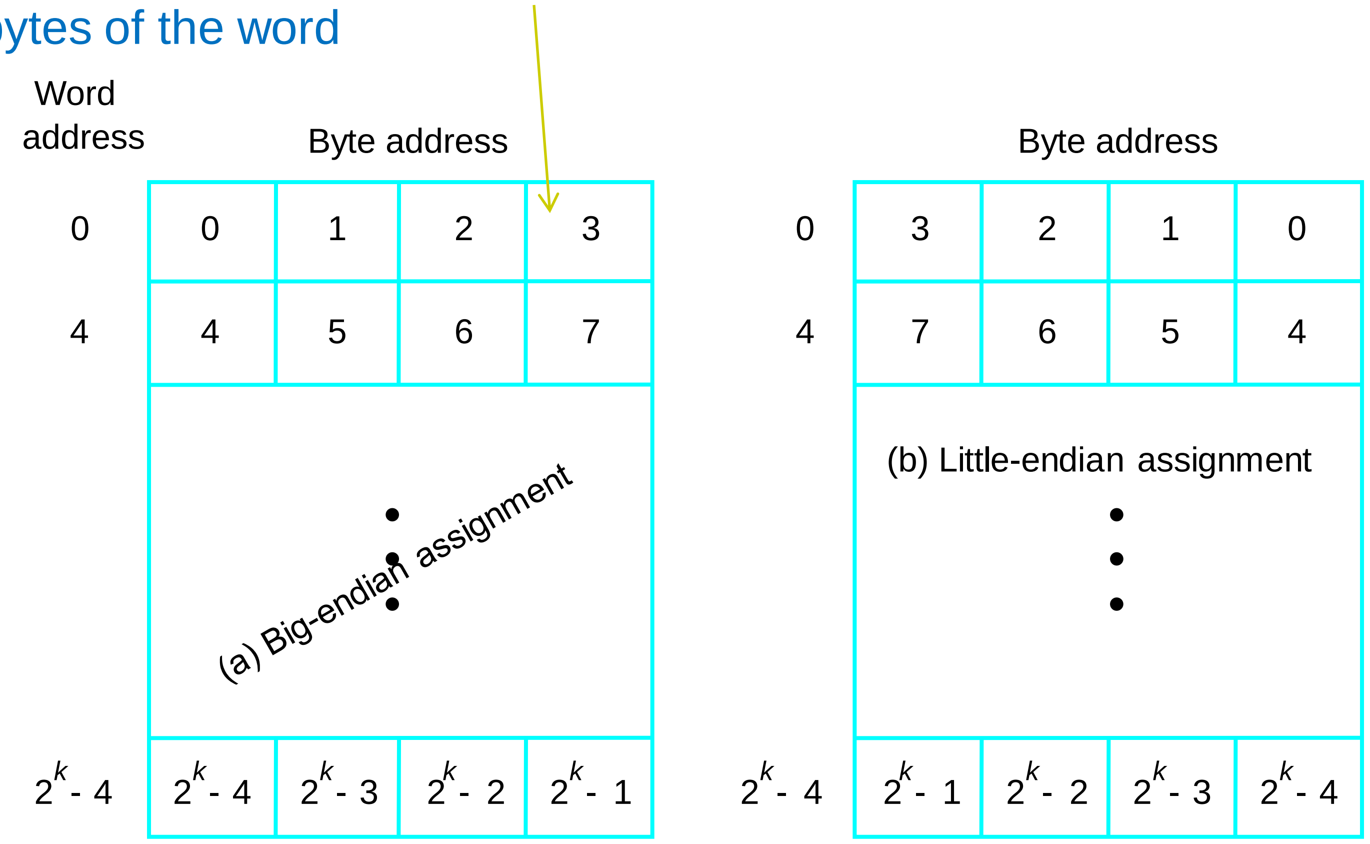
- To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.
- A *k*-bit address memory has 2^k memory locations, namely $0 – 2^k-1$, called memory space.
- 24-bit memory: $2^{24} = 16,777,216 = 16M$ ($1M=2^{20}$)
- 32-bit memory: $2^{32} = 4G$ ($1G=2^{30}$)
- 1K(kilo)=2^{10}
- 1T(tera)=2^{40}

Memory Location, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- byte-addressable memory - Byte locations have addresses 0, 1, 2, …
- word-addressable- If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word


Word address
Byte address
0
4
$2^k - 4$
0 1 2 3
4 5 6 7
$2^k - 4$ $2^k - 3$ $2^k - 2$ $2^k - 1$
(a) Big-endian assignment
Byte address
3 2 1 0
7 6 5 4
$2^k - 1$ $2^k - 2$ $2^k - 3$ $2^k - 4$
(b) Little-endian assignment

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word

Memory Operation

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn’t change.
  - What to supply to load →Address
  - Registers can be used

- Store (or Write)
  - Overwrite the content in memory
  - What to supply to Store → Address and Data
  - Registers can be used

Instruction and Instruction Sequencing

“Must-Perform” Operations

- Data transfers -- between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

Register Transfer Notation(RTN)

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)

- Contents of a location are denoted by placing square brackets around the name of the location ($R1 \leftarrow [LOC]$, $R3 \leftarrow [R1]+[R2]$)

- Register Transfer Language (RTL)

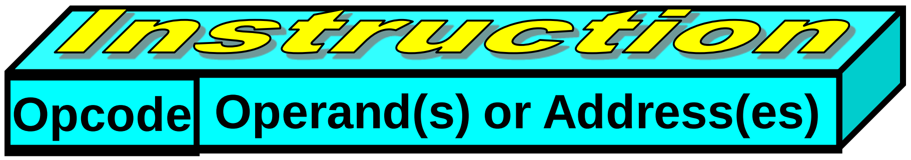
Assembly Language Notation

- Represent machine instructions and programs.
- Move LOC, R1 ➔ R1←[LOC]
- Add R1, R2, R3 ➔ R3 ←[R1]+[R2]

CPU Organization

- Single Accumulator
  - Result usually goes to the Accumulator
  - Accumulator has to be saved to memory quite often
- General Register
  - Registers hold operands thus reduce memory traffic
  - Register bookkeeping
- Stack
  - Operands and result are always in the stack

Instruction Formats



- Three-Address Instructions
  - ADD R1, R2, R3 $R1 \leftarrow R2 + R3$
- Two-Address Instructions
  - ADD R1, R2 $R1 \leftarrow R1 + R2$
- One-Address Instructions
  - ADD M $AC \leftarrow AC + [M]$
- Zero-Address Instructions
  - ADD $TOS \leftarrow TOS + (TOS – 1)$

Instruction Formats

RISC Instructions

Lots of registers. Memory is restricted to Load & Store

Example: Evaluate $(A+B) * (C+D)$

- Three-Address
  1. ADD R1, A, B ; $R1 \leftarrow M[A] + M[B]$
  2. ADD R2, C, D ; $R2 \leftarrow M[C] + M[D]$
  3. MUL X, R1, R2 ; $M[X] \leftarrow R1 * R2$

Instruction Formats

Example: Evaluate $(A+B) * (C+D)$

- Two-Address

1.	MOV	R1, A	; $R1 \leftarrow M[A]$
2.	ADD	R1, B	; $R1 \leftarrow R1 + M[B]$
3.	MOV	R2, C	; $R2 \leftarrow M[C]$
4.	ADD	R2, D	; $R2 \leftarrow R2 + M[D]$
5.	MUL	R1, R2	; $R1 \leftarrow R1 * R2$
6.	MOV	X, R1	; $M[X] \leftarrow R1$

Instruction Formats

Example: Evaluate $(A+B) * (C+D)$

- One-Address

1. LOAD A	; $AC \leftarrow M[A]$
2. ADD B	; $AC \leftarrow AC + M[B]$
3. STORE T	; $M[T] \leftarrow AC$
4. LOAD C	; $AC \leftarrow M[C]$
5. ADD D	; $AC \leftarrow AC + M[D]$
6. MUL T	; $AC \leftarrow AC * M[T]$
7. STORE X	; $M[X] \leftarrow AC$

Instruction Formats

Example: Evaluate (A+B) $*$ (C+D)

- Zero-Address

1. PUSH A	; TOS $\leftarrow$ A
2. PUSH B	; TOS $\leftarrow$ B
3. ADD	; TOS $\leftarrow$ (A + B)
4. PUSH C	; TOS $\leftarrow$ C
5. PUSH D	; TOS $\leftarrow$ D
6. ADD	; TOS $\leftarrow$ (C + D)
7. MUL	; TOS $\leftarrow$ (C+D)$*$(A+B)
8. POP X	; M[X] $\leftarrow$ TOS

Instruction Formats

Example: Evaluate $(A+B) * (C+D)$

- RISC
  1. LOAD R1, A ; $R1 \leftarrow M[A]$
  2. LOAD R2, B ; $R2 \leftarrow M[B]$
  3. LOAD R3, C ; $R3 \leftarrow M[C]$
  4. LOAD R4, D ; $R4 \leftarrow M[D]$
  5. ADD R1, R1, R2 ; $R1 \leftarrow R1 + R2$
  6. ADD R3, R3, R4 ; $R3 \leftarrow R3 + R4$
  7. MUL R1, R1, R3 ; $R1 \leftarrow R1 * R3$
  8. STOREX, R1 ; $M[X] \leftarrow R1$

Using Registers

- Registers are faster : Potential speedup
- Shorter instructions
  - The number of registers is smaller (e.g. 32 registers need 5 bits)
- Minimize the frequency with which data is moved back and forth between the memory and processor registers.

Instruction Execution and Straight-Line Sequencing

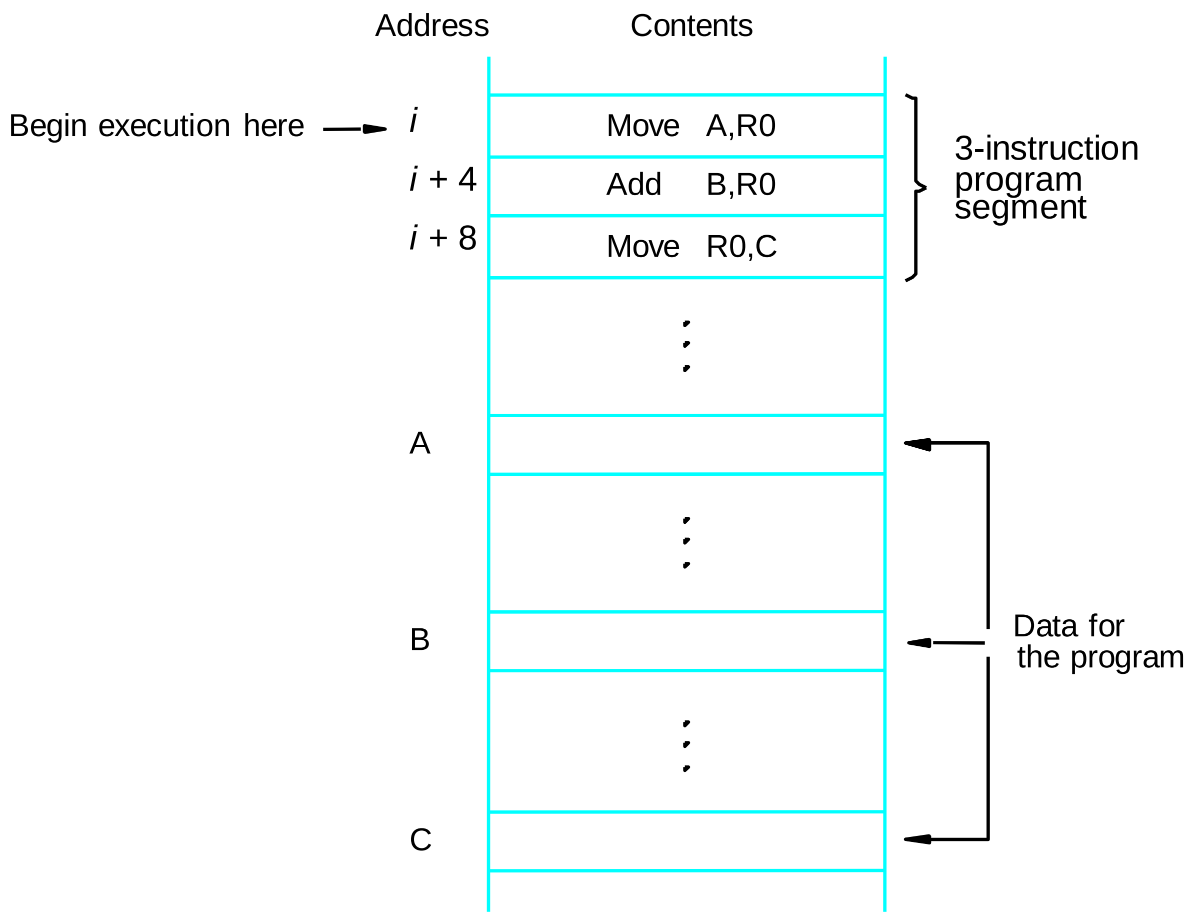
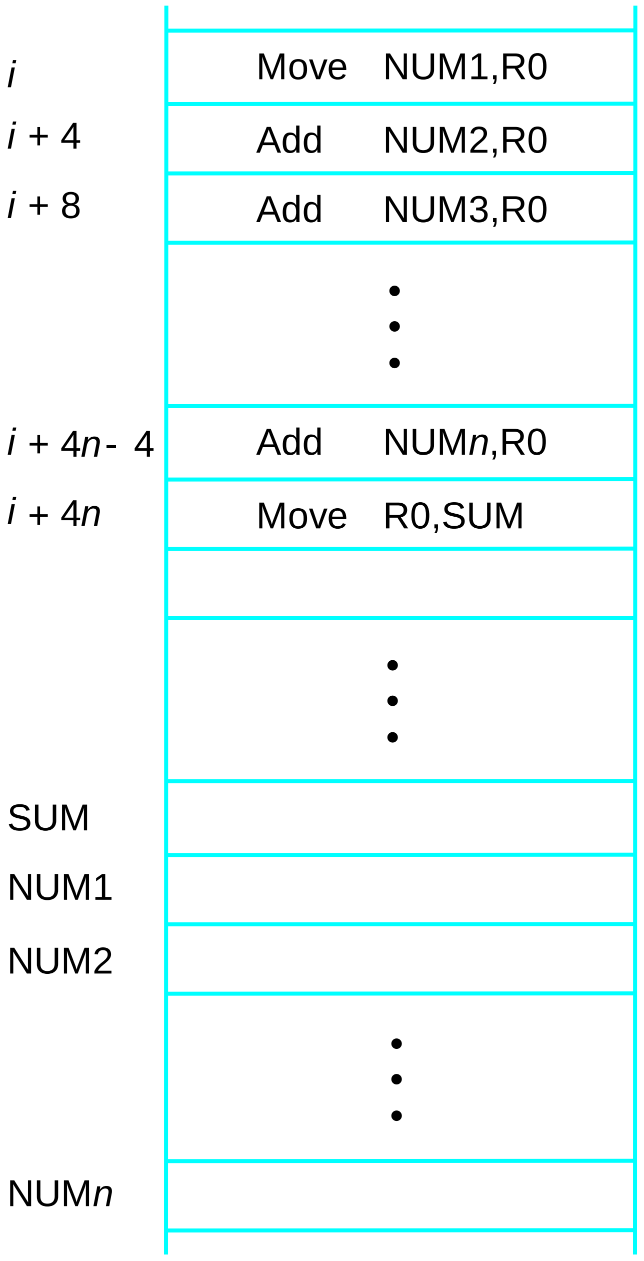


Figure 2.8. A program for C ← [A] + [B].

Assumptions:
- One memory operand per instruction
- 32-bit word length
- Memory is byte addressable
- Full memory address can be directly specified in a single-word instruction

Two-phase procedure
-Instruction fetch
-Instruction execute

A straight-line program for adding *n* numbers



```
i             Move   NUM1,R0
i + 4         Add    NUM2,R0
i + 8         Add    NUM3,R0
                 ⋮
i + 4n - 4    Add    NUMn,R0
i + 4n        Move   R0,SUM

                 ⋮
SUM
NUM1
NUM2
                 ⋮
NUMn
```

Branching

Branch target

Conditional branch

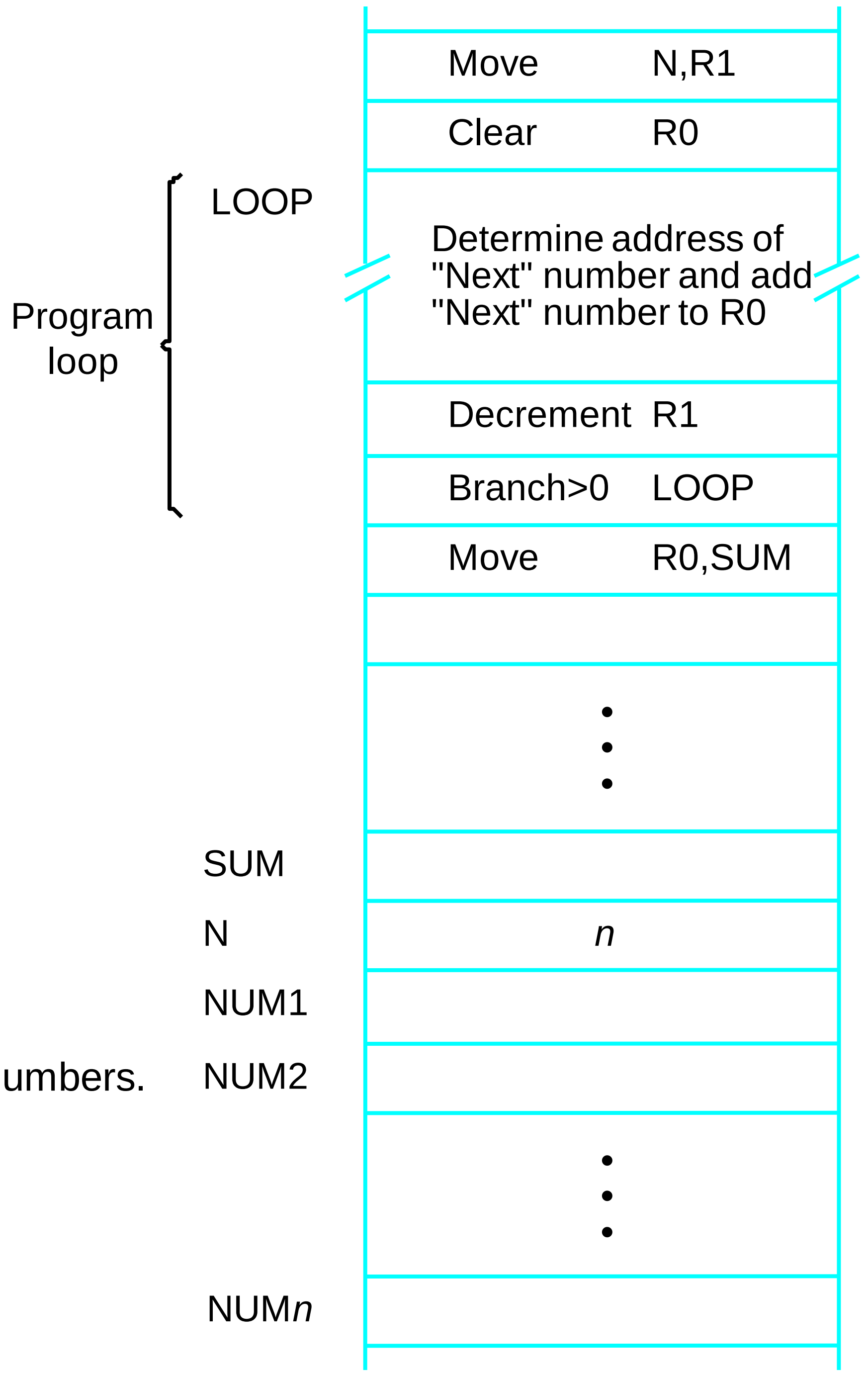


Figure 2.10. Using a loop to add *n* numbers.

Condition Codes

- Condition code flags
- Condition code register / status register

- N (negative)
- Z (zero)
- V (overflow)
- C (carry)

- Different instructions affect different flags

Conditional Branch Instructions

- Example:
  - A: 1 1 1 1 0 0 0 0
  - B: 0 0 0 1 0 1 0 0

```
A:       1 1 1 1 0 0 0 0
+(−B):   1 1 1 0 1 1 0 0
------------------------
         1 1 0 1 1 1 0 0
```

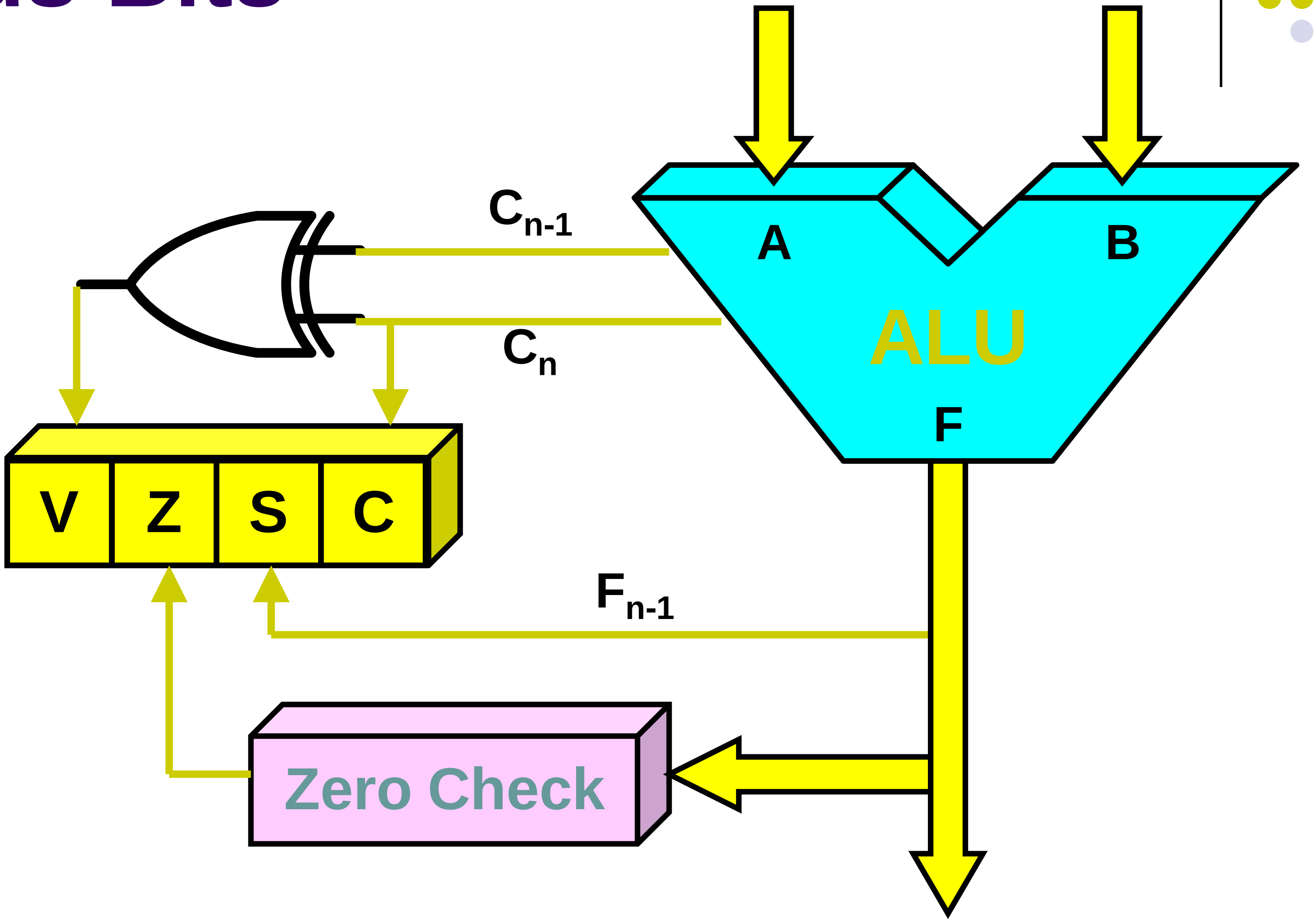
C = 1

Z = 0

S = 1

V = 0

Status Bits



Addressing Modes

The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Addressing Modes

- EA= Effective address
  ie
  actual address of the operand

Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC

Indirect	(Ri)
	(LOC)
		EA = [Ri]
		EA = [LOC]

Index	X(Ri)	EA = [Ri] + X
Basewith index	(Ri,Rj)	EA = [Ri] + [Rj]
Basewith index and offset	X(Ri,Rj)	EA = [Ri] + [Rj] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri] ; Increment Ri
Autodecrement	−(Ri)	Decrement Ri ; EA = [Ri]

Addressing Modes

- Immediate
  - Instruction contains the one of the operand
  - Ex “MOV R1, #5”, i.e. R1$\leftarrow$ 5
- Register / register direct
  - Indicate which register holds the operand
  - Ex: Load R1
- Direct Addressing / Absolute AM
  - Instruction contains the memory address of the operand
  - Ex: Load A

Addressing Modes

- Indirect Addressing Mode :
  - Instruction contains address of address of operand

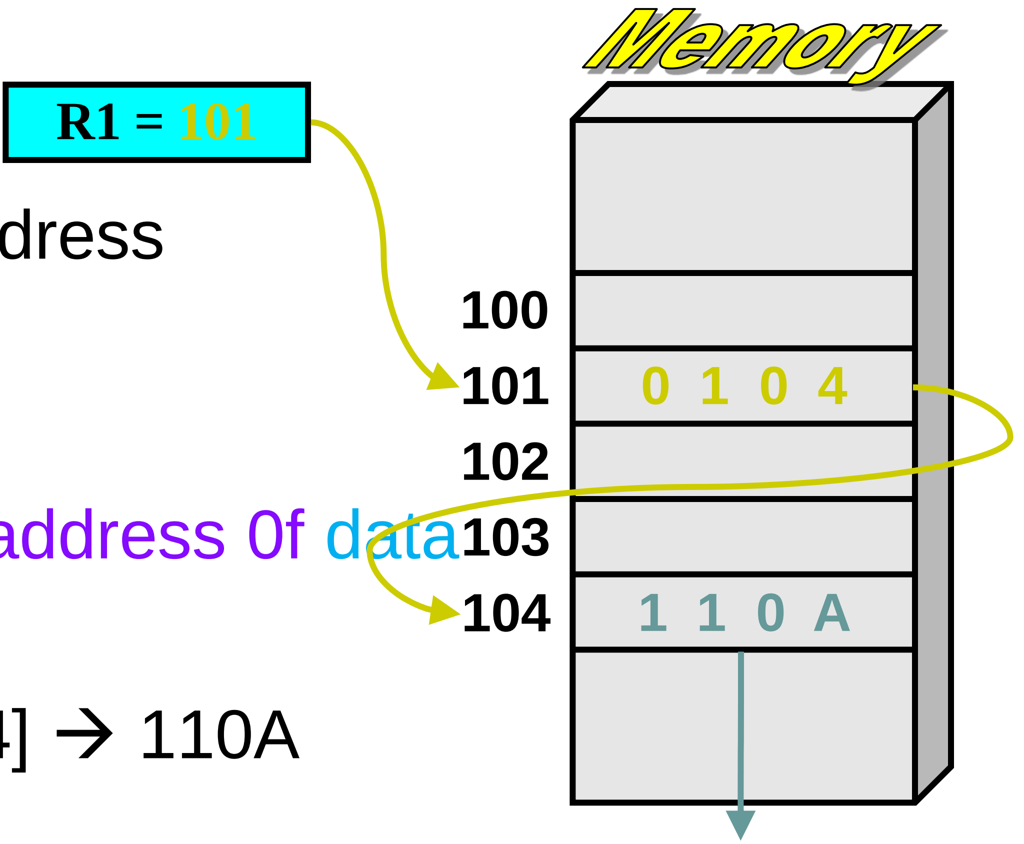
Load (R1)

- Register holds address

Of address of data

[[R1]]→address of address 0f data

[[101]] → [104] → 110A

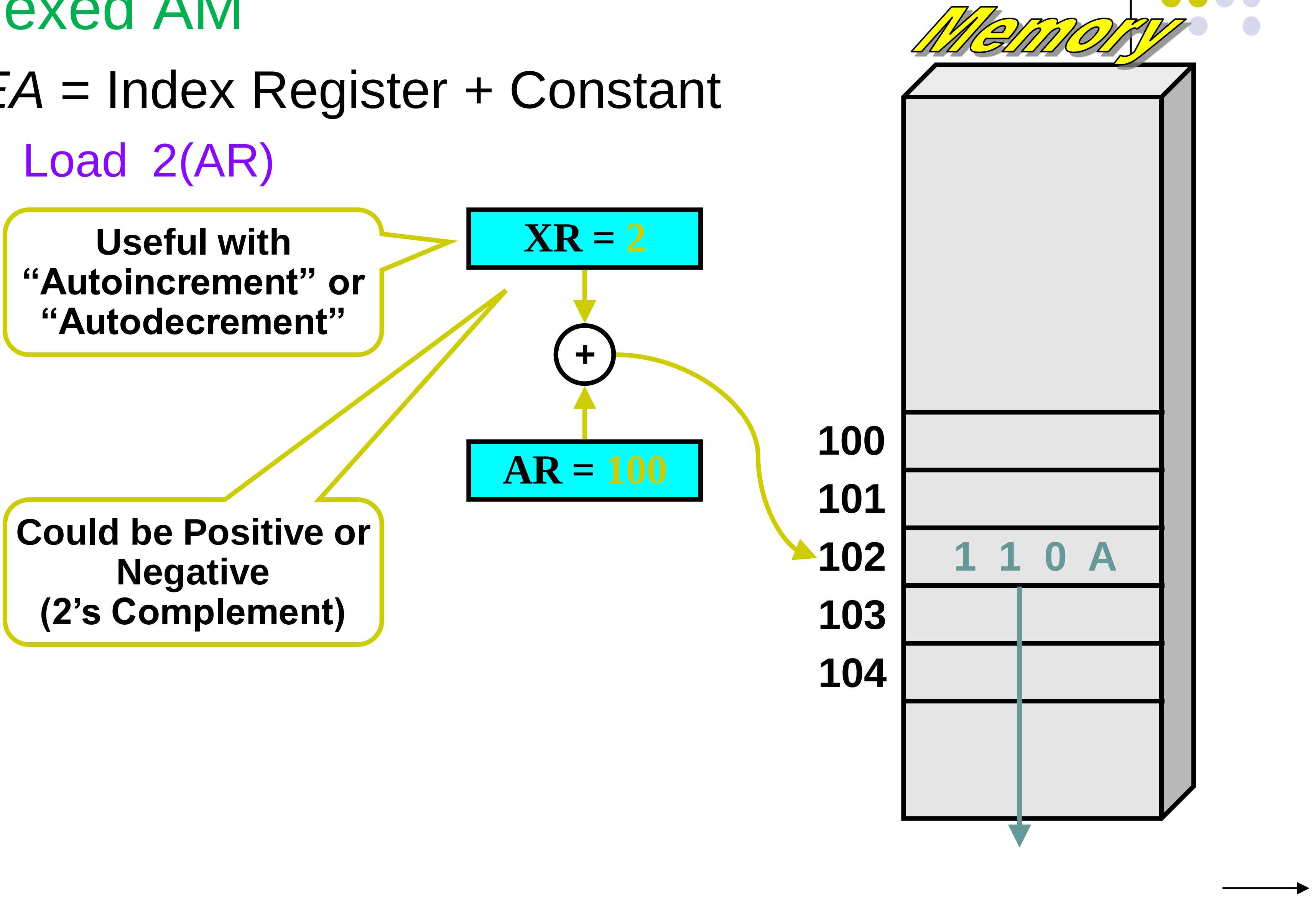


Indexing and Arrays

- Index addressing mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register : R_i
- $X(R_i)$: $EA = X + [R_i]$; X is constant
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
  - EX1 : $EA = 200 + [R_i]$ EX2 : $EA = MAX + [R_i]$

Addressing Modes

- Indexed AM
  - *EA* = Index Register + Constant
    - Load 2(AR)

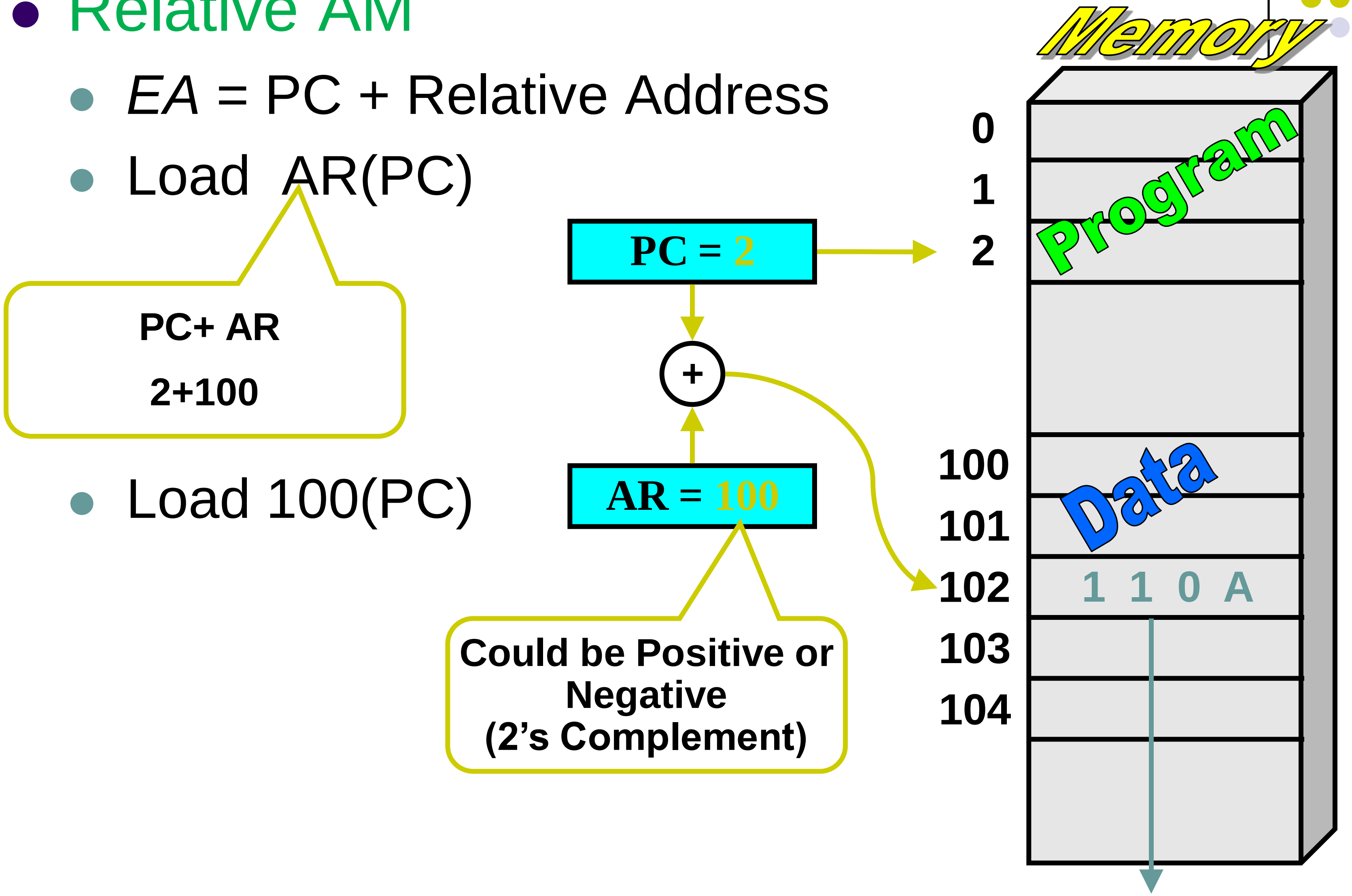

Useful with “Autoincrement” or “Autodecrement”
Could be Positive or Negative (2’s Complement)
XR = 2
+
AR = 100
Memory
100
101
102
1 1 0 A
103
104

Relative Addressing

- Relative Addressing mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.
- X(PC) – note that X is a signed number
- This location is computed by specifying it as an offset from the current value of PC.
- Ex: Branch>0 LOOP
- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

Addressing Modes

- Relative AM
  - *EA* = PC + Relative Address
  - Load AR(PC)
  - Load 100(PC)



Addressing Modes

EA= Effective address i.e actual address

Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		EA = [LOC]

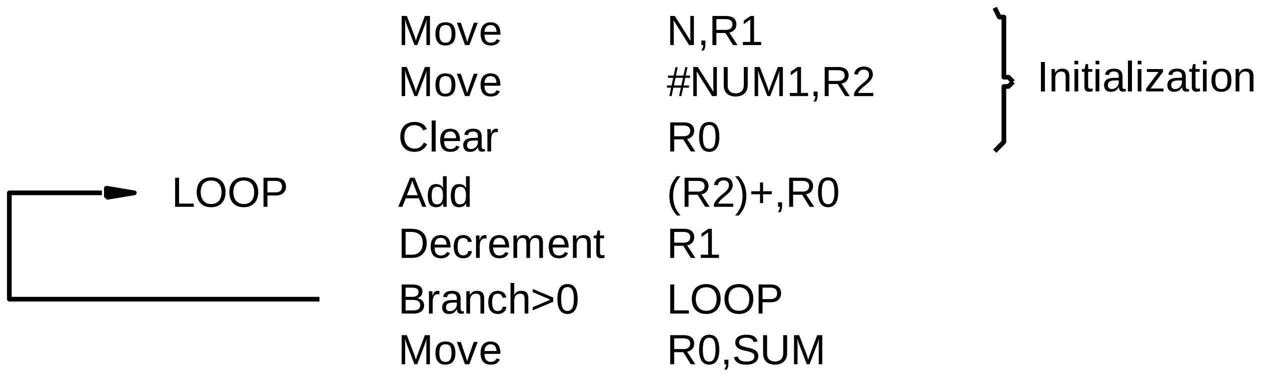
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Additional Modes

- Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.

- **$(R_i)+$.** The increment is 1 for byte-sized operands, 2 for 16-bit operands, and 4 for 32-bit operands.

- Autodecrement mode: **$-(R_i)$** – decrement first



```
                Move          N,R1        }
                Move          #NUM1,R2    } Initialization
                Clear         R0          }
        LOOP    Add           (R2)+,R0
                Decrement     R1
                Branch>0      LOOP
                Move          R0,SUM
```

Figure 2.16. The Autoincrement addressing mode used in the program of Figure 2.12.

A L P

Assembly language programming And Assembler Directives

Overview of Assembly Language

- ❑ Advantages:
  - ✓ Faster as compared to programs written using HLL
  - ✓ Efficient memory usage
  - ✓ Control down to bit level
- ❑ Disadvantages:
  - × Need to know detail hardware implementation
  - × Not portable
  - × Slow to development and difficult to debug
- ❑ Basic components in assembly Language:

  Instruction, Directive, Label, and Comment

Assembly language Instruction Format

- General Format of Instructions

[label:]   mnemonic/Opcode   [operands]   [;comment]

- **Label**: It provides a symbolic address
- **Opcode:** It specifies the type the operation (e.g. **`add`**, **`sub`**, etc.)
- **Operands**: Specify the data required by the operation
- **Comments**: Only for programmers' reference, Explain the program's purpose

- Machine Code Format

Opcode	Mode	Operand1	Operand2

$$\text{MOV AL, BL} \quad \Longrightarrow \quad \underbrace{1\ 0\ 0\ 0\ 1}_{\text{MOV}}\ 0\ 0\ \underbrace{0\ 1}_{\text{Register mode}}\ 1\ 0\ 0\ 0\ 0\ 1\ 1$$

Example of Assembly Language Program

```
;NUMOFF.ASM: Turn NUM-LOCK indicator off.
        .MODEL SMALL
        .STACK
        .CODE
        .STARTUP
        MOV     AX,40H      ;set AX to 0040H
D1:     MOV     DS,AX       ;load data segment with 0040H
        MOV     SI,17H      ;load SI with 0017H
        AND     BYTE PTR [SI],0DFH   ;clear NUM-LOCK
bit
        .EXIT
        END
```

Comments

Assembly directive

Instructions

Assembly directive

Label

Assembler directives

- Provide information / direction to the assembler while translating a program

- Assembler directives are pseudo instructions
  - They will not be translated into machine instructions.
  - Non-executable: directives are not part of the instruction set

Assembler directives…..

- They can be used to declare variables, to declare constants, and to label locations in the code to be used as branch destinations.
- create storage space for results, Used to define segments, allocate memory
- Use of Assembler directives makes an assembly language easier to understand.

Assembler Directives…

- Basic assembler directives
  - START :
    - Specify name and starting address for the program
  - END :
    - Indicate the end of the source program
  - BYTE :
    - Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant.
  - WORD :
    - Generate one-word integer constant
  - RESB :
    - Reserve the indicated number of bytes for a data area
  - RESW :
    - Reserve the indicated number of words for a data area

.DATA, .TEXT, & .GLOBL Directives

- .DATA directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- .TEXT / .CODE directive
  - Defines the code segment of a program containing instructions
- .GLOBL directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files

ORG (origin)

- Tells the assembler where to place your code in memory.
  - ORG   value

- Operand given is the desired memory location. Ex: ORG IE00; sets origin to be IE00
- Value can be: constant /symbol / expression
-
```
      CODE
                          ORG     8000H
         TOTAL:   MOV     AX,7000H
                          MOV     DS,AX
                          MOV     AL,0
```

Assembler Directive:(END)

END ; end of code

- No operands / labels
- END directive is placed after the last statement of a program
- Tells the assembler to stop looking for more code.
- The assembler will ignore any statement after an END directive

END - End Program

ENDP - End Procedure

ENDS - End Segment

Assembler directive (EQU)

- In ALP, constants are defined through EQU
- **EQU -** used to give a name to some value or to a symbol.
- Each time the assembler finds the name in the program, it will replace the name with the value or symbol.

➢ Syntax: Name EQU constant

```
➢ Ex:  MIN  EQU 5    ; sets the value of MIN to 5
       MAX   EQU  100
       NAME    EQU   'SJCE'
```

Variables

Pseudo-op	type	size	range
DB	unsigned	1 byte	0 to 255.
	signed	1 byte	-128 to +127.
DW	unsigned	2 bytes	0 to 65,535 (64K).
	signed	2 bytes	-32,768 to +32,767.
DD	unsigned	4 bytes	0 to 4,294,967,295 (4 Mbytes).
	signed	4 bytes	-2,147,483,648 to +2,147,483,647.

Assembler Directive –DB(byte Variable)

- **DB -** DB directive is used to declare a byte type variable or to store a byte in memory location.
- Syntax: Name DB initial value

  Examples: ALPHA DB 4

- **DW -** is used to define a variable of type word or to reserve storage location of type word in Memory
- Syntax: Name DW initial value

  Example: WRD DW -2

Data Directives

- **.BYTE** Directive
  - Stores the list of values as 8-bit bytes
- **.HALF** Directive
  - Stores the list as 16-bit values aligned on half-word
- **.WORD** Directive
  - Stores the list as 32-bit values aligned on a word boundary
- **.FLOAT** Directive
  - Stores the listed values as single-precision floating point
- **.DOUBLE** Directive
  - Stores the listed values as double-precision floating point

Example of An Assembly Language Programme segmet

```
        ORG   $001000
        MOVE VAL1, D0
        MOVE VAL2, D1
        MUL  D0, D1
        ADD  #CONST, D1
        END

        ORG   $002000
VAL1    DC    3
VAL2    DC    12
CONST   EQU   5
```

Example of Complete ALP

```
        ORG     $001000
START   CLR.L   D0                  ; sum = 0;
        MOVEA.L         #ARRAY,A0       ; ptr points to the 1st
array element
        MOVE.W  #SIZE,D1                ; counter = SIZE;
LOOP    ADD.W   (A0)+,D0                ; sum += *ptr++;
        SUBQ.W  #1,D1               ; counter -= 1;
        BGT     LOOP                ; repeat while counter > 0;
        MOVE.W  D0,SUM              ; Save the sum in memory
        MOVE.W  #228,D7                 ; Magic exit code
        TRAP    #14                 ; more magic exit code
;--------DATA AND CONSTANTS------------------------------
        ORG     $002000
SIZE    EQU     9                       ; the size of the array
ARRAY   DC.W    3,7,-5,12,23,15,-12,82,5   ; the array to
sum
SUM     DS.W    1                       ; the final sum
        END
```

Assembly Language

Instructions in the Instruction set

Types of Instructions

- Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data value is not modified

Data Transfer Instructions

Mode	Assembly	Register Transfer
Direct address	LD ADR	$AC \leftarrow M[ADR]$
Indirect address	LD @ADR	$AC \leftarrow M[M[ADR]]$
Relative address	LD $ADR	$AC \leftarrow M[PC+ADR]$
Immediate operand	LD #NBR	$AC \leftarrow$ NBR
Index addressing	LD ADR(X)	$AC \leftarrow M[ADR+XR]$
Register	LD R1	$AC \leftarrow R1$
Register indirect	LD (R1)	$AC \leftarrow M[R1]$
Autoincrement	LD (R1)+	$AC \leftarrow M[R1]$, $R1 \leftarrow R1+1$

Data Manipulation Instructions

- Arithmetic
- Logical & Bit Manipulation
- Shift

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate	NEG

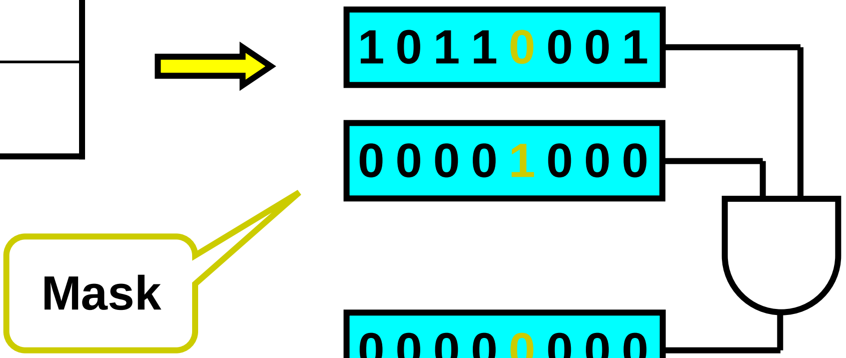
Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (Subtract)	CMP
Test (AND)	TST

Subtract A – B but don't store the result



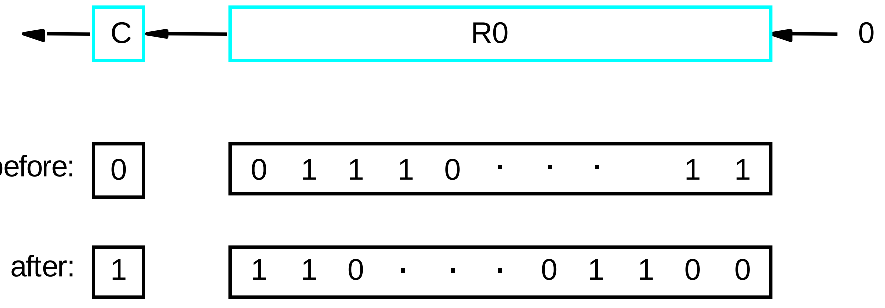
Conditional Branch Instructions

Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0

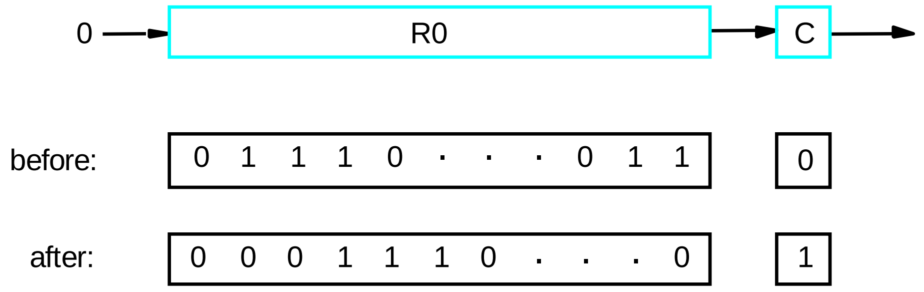
Additional Instructions

Logical Shifts

- Logical shift – shifting left (LShiftL) and shifting right (LShiftR)

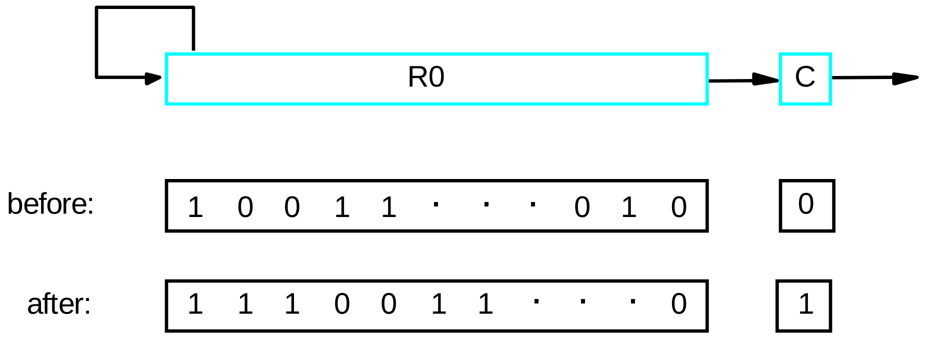


(a) Logical shift left LShiftL #2,R0



(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts



(c) Arithmetic shift right AShiftR #2,R0

Rotate

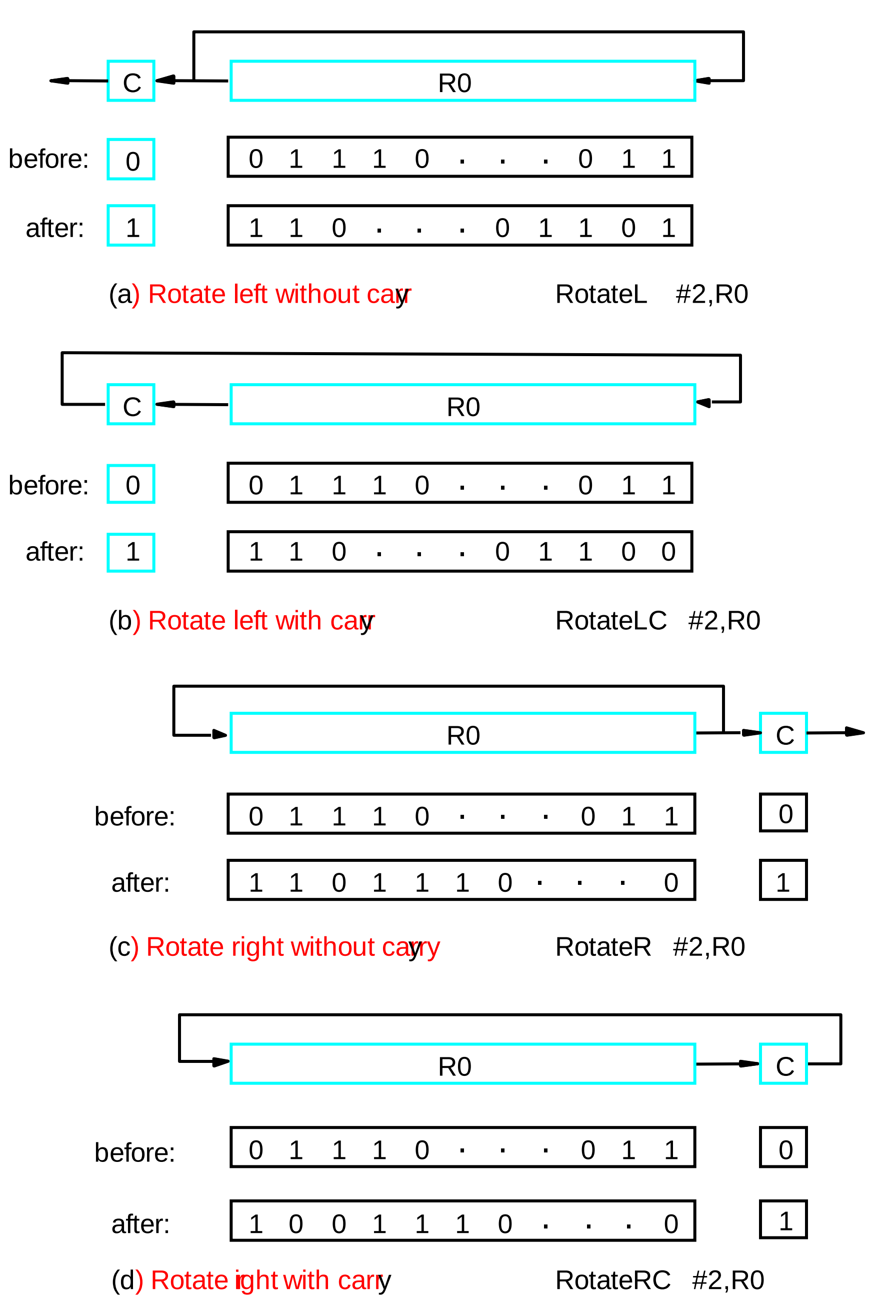


Figure 2.32. Rotate instructions.

Multiplication and Division

- Not very popular (especially division)

- Multiply R_i, R_j
  $R_j \leftarrow [R_i] \times [R_j]$
- 2n-bit product case: high-order half in R(j+1)

- Divide R_i, R_j
  $R_j \leftarrow [R_i] / [R_j]$
  Quotient is in Rj, remainder may be placed in R(j+1)

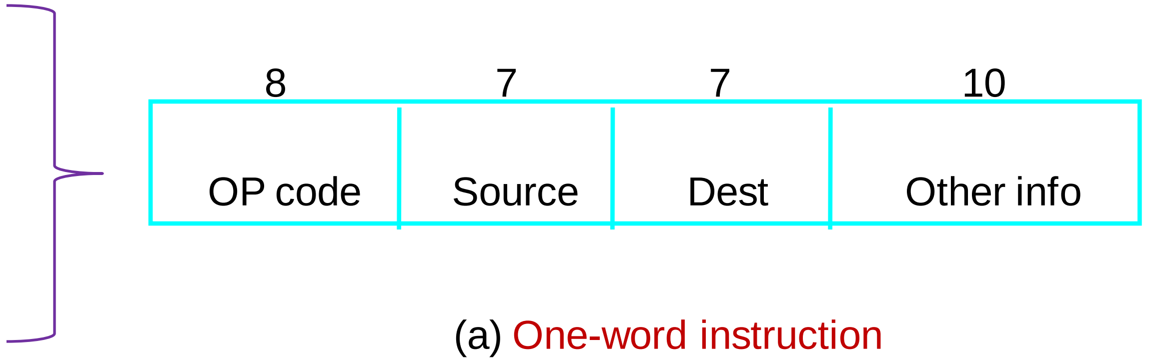
Encoding of Machine Instructions

Converting ALP instructions into Machine level instruction ….. i. e. into set of 0’s and 1’s

Encoding of Machine Instructions

- Assembly language program needs to be converted into machine instructions. (ADD = 0100)
- OP code and the type of operands used may be specified using an encoded binary pattern
- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.

- Add R1, R2
- Move 24(R0), R5
- LshiftR #2, R0
- Move #$3A, R1
- Branch>0 LOOP



(a) One-word instruction

Encoding of Machine Instructions

- What happens if we want to specify a memory operand using the Absolute addressing mode?

  Move R2, LOC

- 14-bit for LOC – insufficient

- Solution – use two words

OP code	Source	Dest	Other info
Memory address/Immediate operand			

(b) Two-word instruction

Encoding of Machine Instructions

- Then what if an instruction in which two operands can be specified using the Absolute addressing mode?

Move LOC1, LOC2

- Solution – use two additional words
- This approach results in instructions of variable length.
- Complex instructions can be implemented – (CISC)

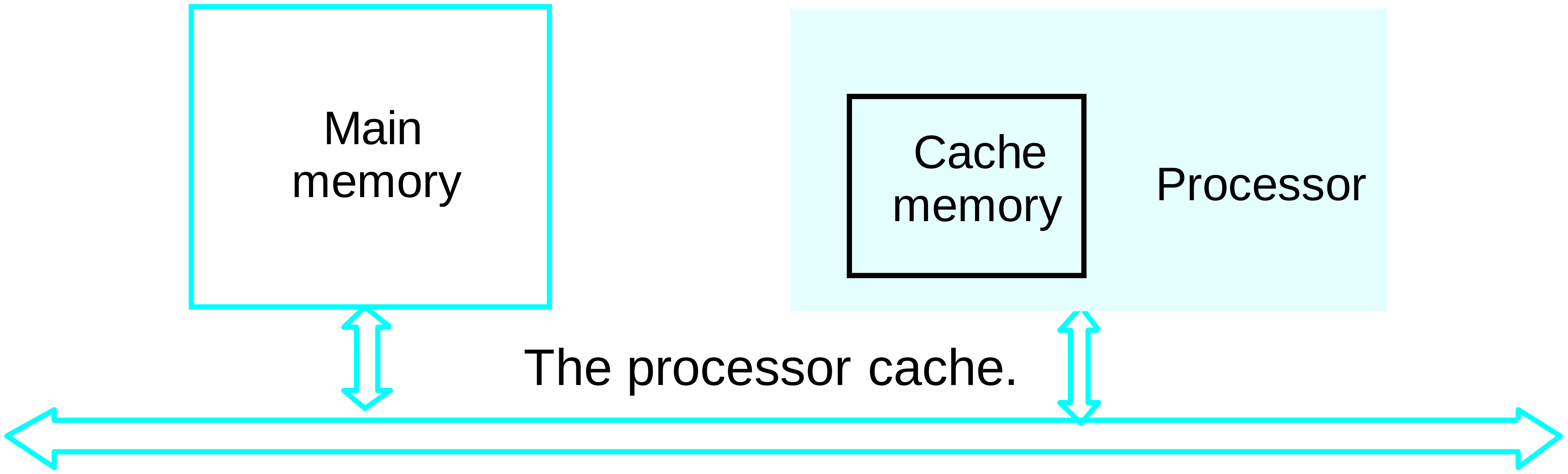
Encoding of Machine Instructions

- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.
- It is still possible to define a highly functional instruction set, which makes extensive use of the processor registers.
- Add R1, R2 ----- yes
- Add LOC, R2 ----- no
- Add (R3), R2 ----- yes

Performance Issues….

- The most important measure of a computer is how quickly it can execute programs.
- Three factors affect performance:
  - Hardware design > Instruction set > Compiler

- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.



The processor cache.

Performance …..

- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip.
- Speed
- Cost
- Memory management

Processor Clock

- Clock,
- Clock cycle, and
- clock rate
- The execution of each instruction is divided into several steps, each of which completes in one clock cycle.
- Hertz – cycles per second

Basic Performance Equation

$$T = \frac{N \times S}{R}$$

- T – processor time required to execute a program that has been prepared in high-level language
- N – number of actual machine language instructions needed to complete the execution (note: loop)
- S – average number of basic steps needed to execute one machine instruction. Each step completes in one clock cycle
- R – clock rate
- **Note**: these are not independent to each other

How to improve T?

Pipeline and Superscalar Operation

- Instructions are not necessarily executed one after another.
- The value of S doesn’t have to be the number of clock cycles to execute one instruction.
- **Pipelining** – overlapping the execution of successive instructions.
- Add R1, R2, R3
- **Superscalar operation** – multiple instruction pipelines are implemented in the processor.
- Goal – reduce S (could become <1!)

Clock Rate

- How to Increase clock rate
  - Improve the integrated-circuit (IC) technology to make the circuits faster
  - Reduce the amount of processing done in one basic step (however, this may increase the number of basic steps needed)
- Increases in R that are entirely caused by improvements in IC technology affect all aspects of the processor's operation equally except the time to access the main memory.

CISC and RISC

- Tradeoff between N and S

- A key consideration is the use of pipelining
  - S is close to 1 even though the number of basic steps per instruction may be considerably larger
  - It is much easier to implement efficient pipelining in processor with simple instruction sets

- Reduced Instruction Set Computers (**RISC**)
- Complex Instruction Set Computers (**CISC**)

Compiler

- A compiler translates a high-level language program into a sequence of machine instructions.
- To reduce N, we need a suitable machine instruction set and a compiler that makes good use of it.
- Goal – reduce N×S
- A compiler may not be designed for a specific processor; however, a high-quality compiler is usually designed for, and with, a specific processor.

Performance Measurement

- T is difficult to compute.
- Measure computer performance using benchmark programs.
- **System Performance Evaluation Corporation (SPEC)** selects and publishes representative application programs for different application domains, together with test results for many commercially available computers.
- Compile and run (no simulation)
- Reference computer

$$SPEC\ rating = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

$$SPEC\ rating = \left(\prod_{i=1}^{n} SPEC_i\right)^{\frac{1}{n}}$$

Multiprocessors and Multicomputers

- Multiprocessor computer
  - Execute a number of different application tasks in parallel
  - Execute subtasks of a single large task in parallel
  - All processors have access to all of the memory – shared-memory multiprocessor
  - Cost – processors, memory units, complex interconnection networks

- Multicomputers
  - Each computer only have access to its own memory
  - Exchange message via a communication network – message-passing multicomputers

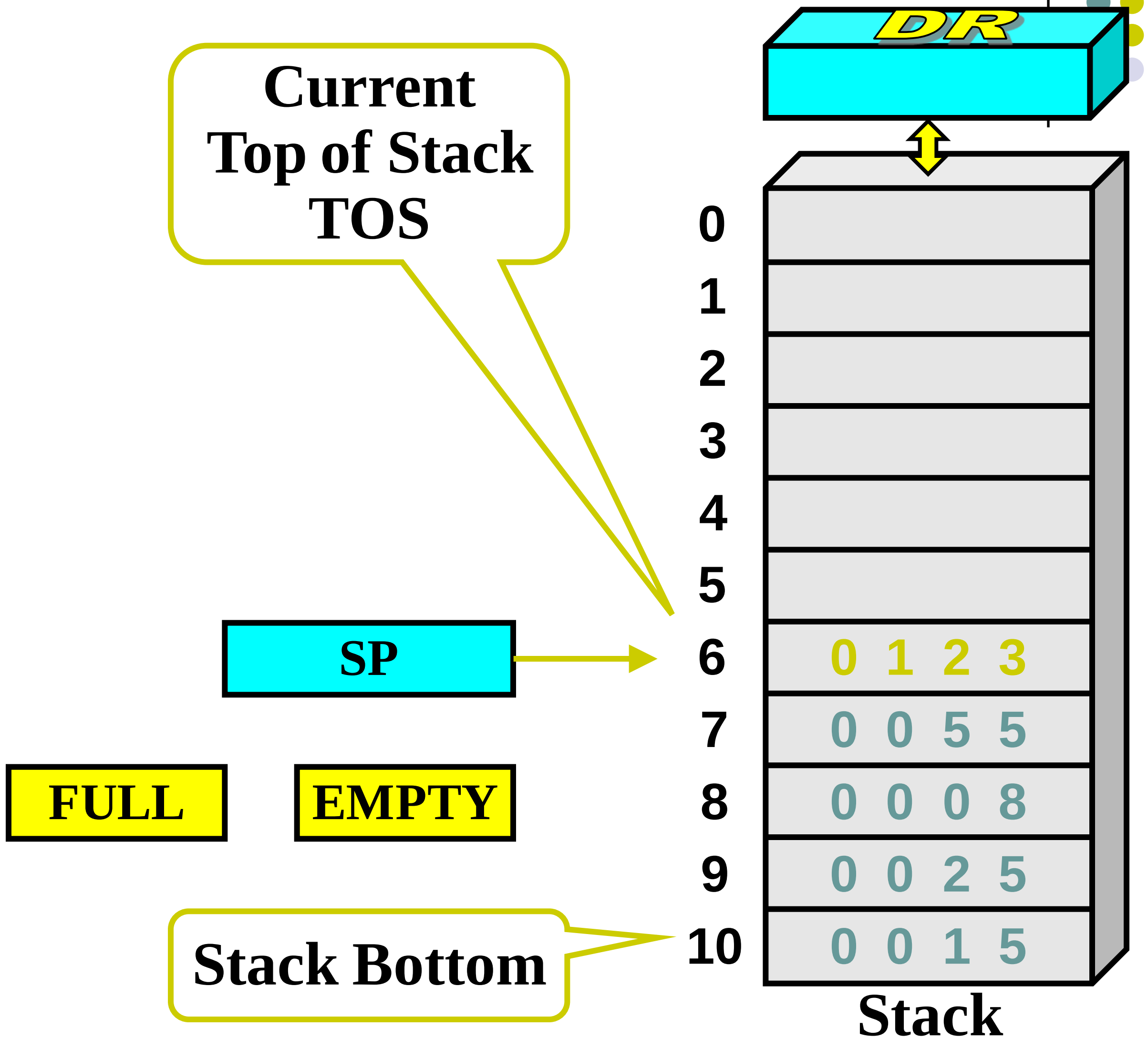
Stacks

It is a data structure that works in LIFO (*Last In First Out*) *fashion*

The Stack: Introduction

- Register R7 generally used as SP
- Since our program usually starts at a low memory address and grows upward, we start the stack at a <u>high memory address</u> and <u>work downward</u>.
- We push values onto the stack using *Auto-decrement mode* MOVE.W DR2,-(SP)
- We pop values from the stack using Auto-*increment mode* MOVE.W (SP)+, DR3
- Some instructions affect the stack directly

Stack Organization



Stack Organization

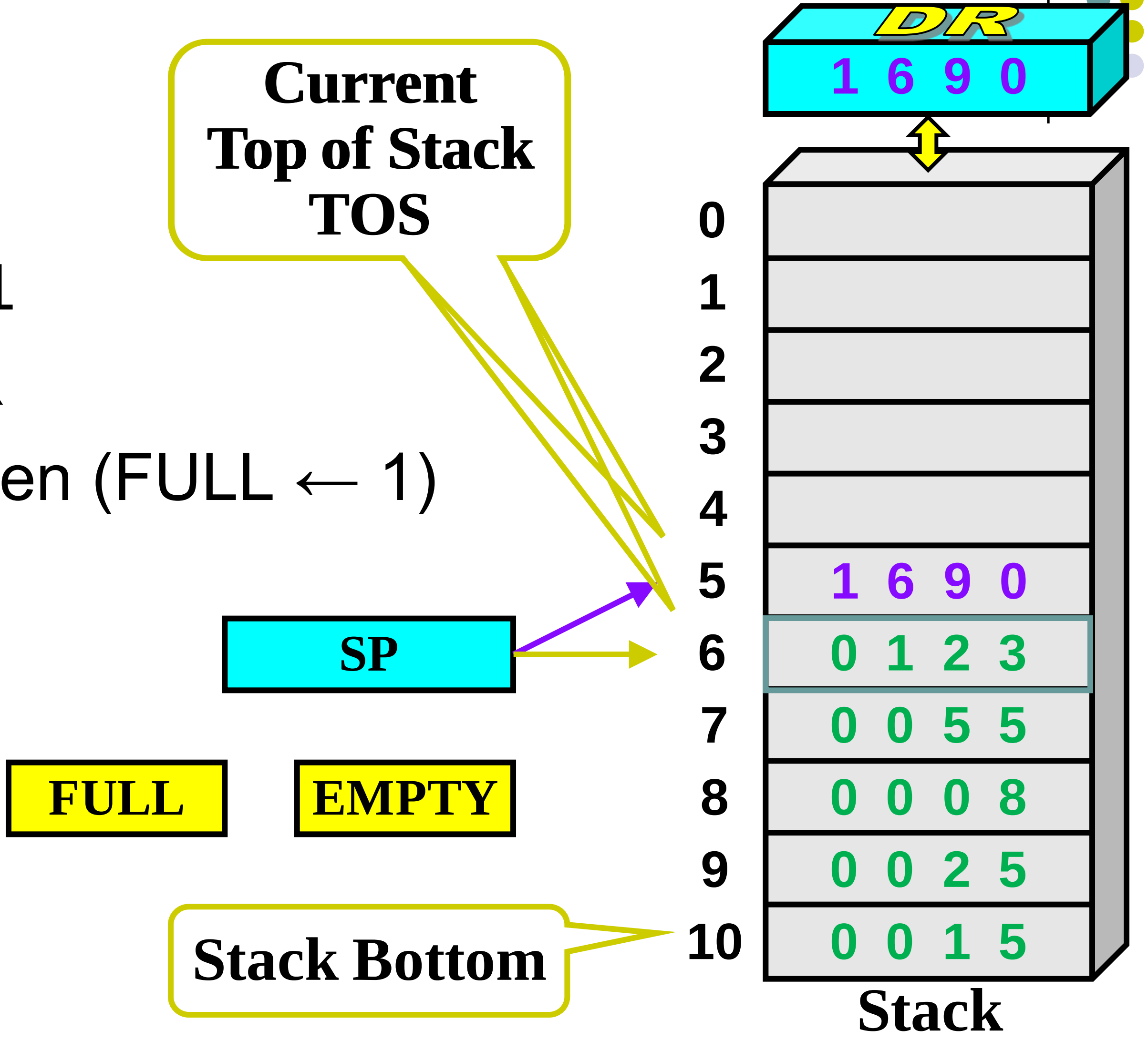
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

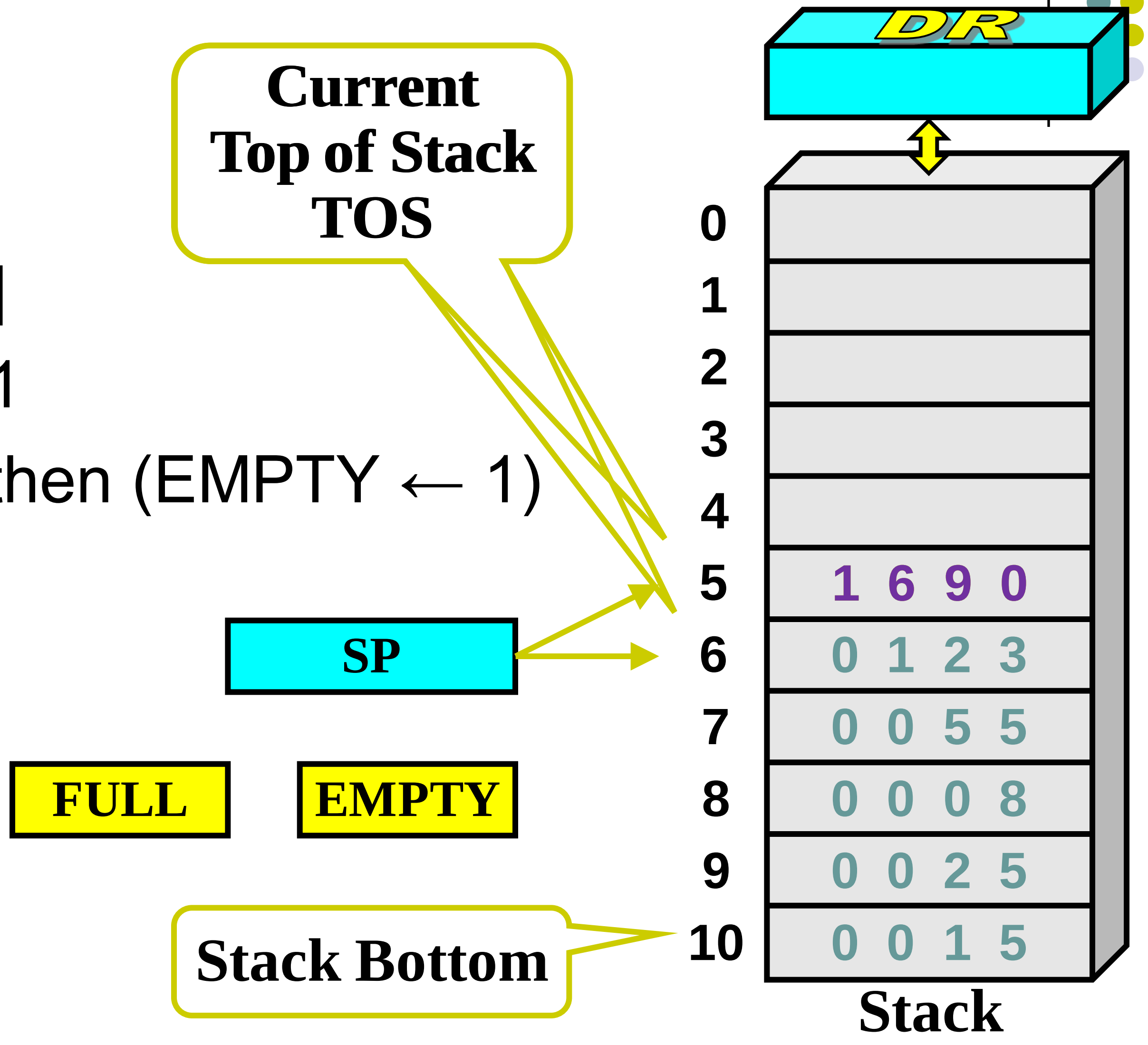
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

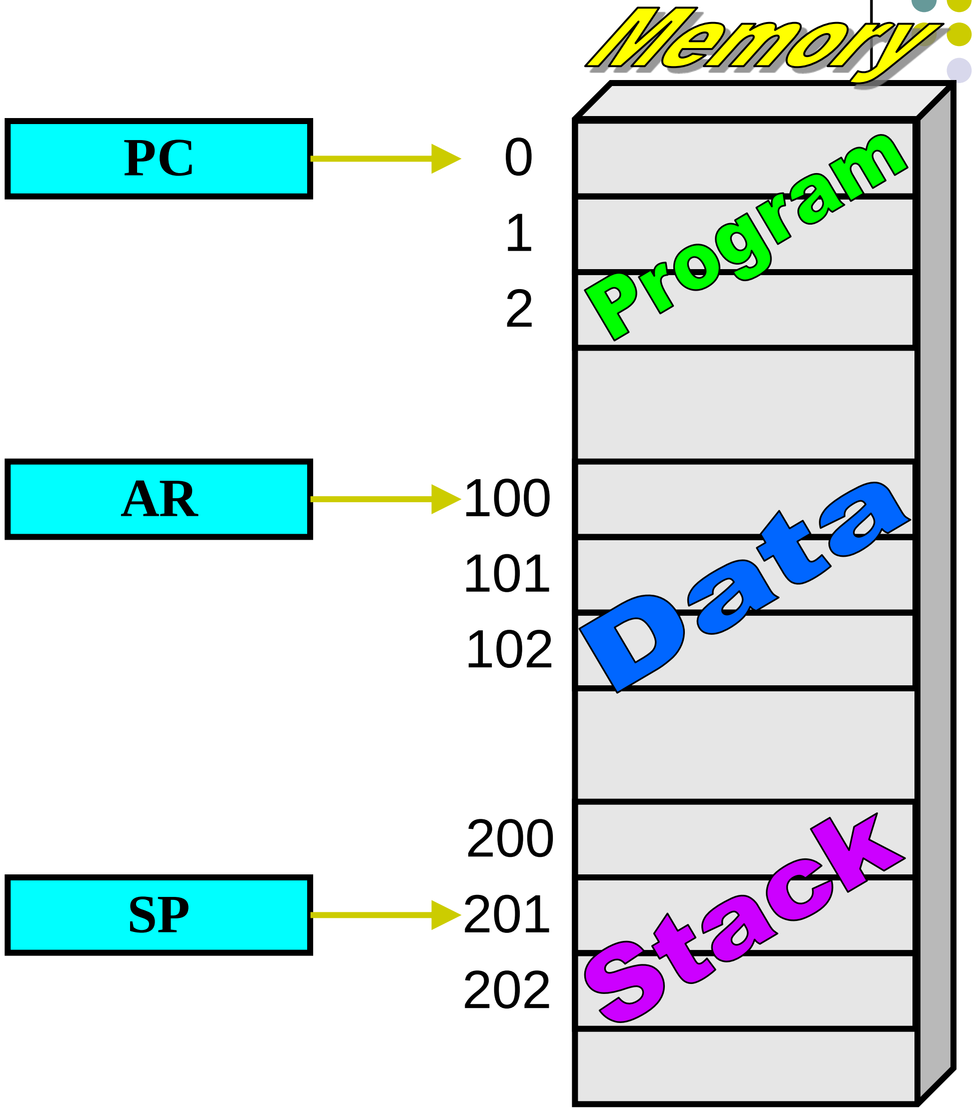
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



Purposes of the Stack

- Temporary storage of variables
- Temporary storage of program addresses
- Communication with *subroutines*
  - Push variables on stack
  - Jump to subroutine & Return back to calling program

Reverse Polish Notation

- Infix Notation

  $A + B$

- Prefix or Polish Notation

  $+ A B$

- Postfix or Reverse Polish Notation (RPN)

  $A B +$

$$A * B + C * D \xrightarrow{\text{RPN}} A\ B * C\ D * +$$

(2) (4) * (3) (3) * +

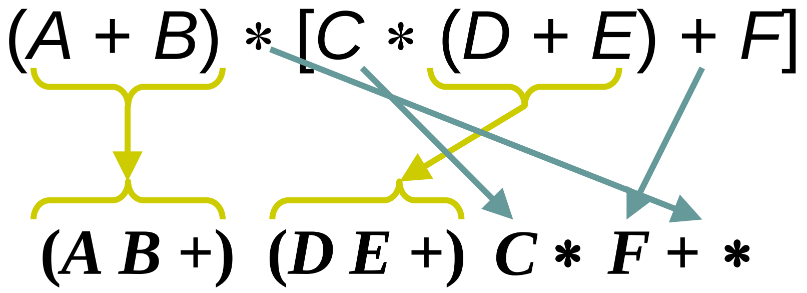
(8) (3) (3) * +

(8) (9) +

17

Reverse Polish Notation

- Example



$(A + B) * [C * (D + E) + F]$

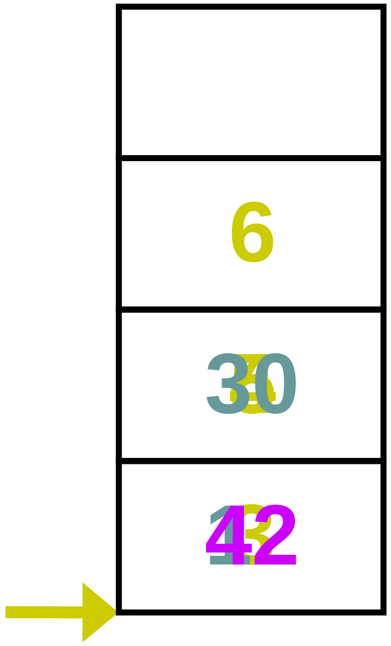
$(A\ B\ +)\ (D\ E\ +)\ C\ *\ F\ +\ *$

Reverse Polish Notation

- Stack Operation

  (3) (4) $*$ (5) (6) $*$ +

```
PUSH    3
PUSH    4
MULT
PUSH    5
PUSH    6
MULT
ADD
```



SUBROUTINE

- Self contained sequence of instructions.
- Why use subroutines?
  - Code re-use
  - Easier to understand code (readability)
  - Divide and conquer
    - Complex tasks are easier when broken down into smaller tasks
- How do we call a subroutine in assembly?
  - Place the parameters *somewhere known*
  - CALL /JSR or BSR to jump to the subroutine
  - Return/ RTS to return from subroutine

C Assembly

```
main() {
    int a, b;
    a = 5;
    b = sqr(a);
    printf("%d\n" b);
}
/* subrtn sqr */
int sqr(int val) {
    int sqval;
    sqval = val * val;
    return sqval;
}
```

```
main    MOVE.W   A,D1
        JSR      sqr
        MOVE.W   D0,B
        move.w   #228,D7
        TRAP     #14

;*** subroutine sqr ***
sqr     MUL.W    D1,D1
        MOVE.W   D1,D0
        RTS
        ORIGIN   $2000
A       DC.W     5
B       DS.W     1
        end
```

CALL instruction: JSR / BSR In

- Call instruction causes a branch to the first instruction of a subroutine.
  - The address of the next instruction available in the PC is saved to a temp. location (buil-in Stack)
  - Control is transferred to the beginning of the subroutine.

- JSR *label*
  - MOVE PC, –(SP)
  - LEA *address-of-label*, PC

  In other words:
  - SP ← [SP] – 4
  - [SP] ← [PC]
  - PC ← *address-of-label*

- BSR *label*
  - Same, but offset from the current PC is stored instead of the absolute address

Return Instruction : RTS

- Return instruction causes a branch back to the calling program.
  - Transfers the contents of the temp. location (return address) back to the PC.
- *RTS is a zero address instruction*
- *Pop the address from the top of the stack back into the PC*

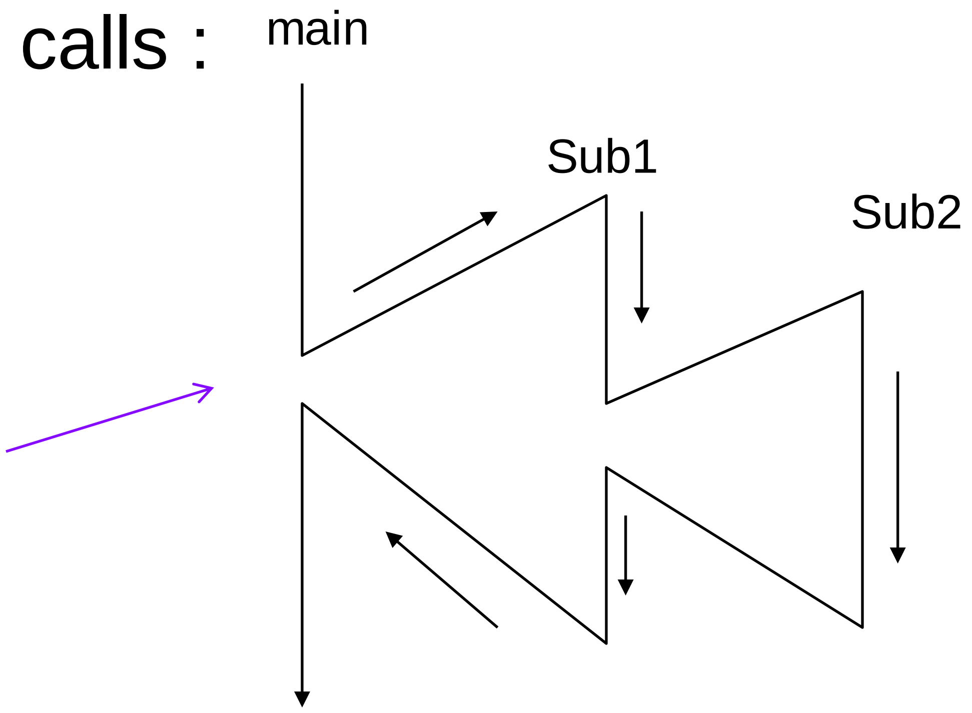
  - MOVE (SP)+, PC

In other words:

- PC $\leftarrow$ [SP]
- [SP] $\leftarrow$ [SP] + 4

Subroutine Call and Return

- Return address can be stored in-
  - Fixed location in memory
  - Processor register
  - Memory stack
- Recursive subroutine calls :
  - Memory stack
    - – most efficient
  - Nested subroutine



Interrupts & Program Interrupt

- an **interrupt** is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- The processor responds by suspending its current activities, saving its state, and executing a small program called an interrupt service routine (ISR) to deal with the event.
- Transfer of program control from a currently running program to another service program as a result of an external or internal generated request.

Subroutine Vs Interrupt

Subroutine	Interrupt
• Initiated by execution of some instruction	• Initiated by some external or internal signal
• Address of the subroutine determined from the address part of the instruction	• Address of the interrupt service routine determined by hardware
• Only the value of the PC stored before branching to the subroutine	• Interrupt procedure stores all information to describe the state of the CPU

Parameter Passing Mechanism

- Parameter passing in assembly language is different
  - More complicated than that used in a high-level language

- In assembly language
  - Place all required parameters in an accessible storage area
  - Then call the procedure

Parameter Passing ---

- Two types of storage areas used
    - Registers: general-purpose registers are used (register method)
    - Memory: stack is used (stack method)

- Two common mechanisms of parameter passing
    - Pass-by-value: parameter value is passed
    - Pass-by-reference: address of parameter is passed

Stack Parameters

- Consider the following max procedure

```
int max ( int x, int y, int z ) {
    int temp = x;
    if (y > temp) temp = y;
    if (z > temp) temp = z;
    return temp;
}
```

Calling procedure: `mx = max(num1, num2, num3)`

Register Parameters

```
mov   ax, num1
mov   bx, num2
mov   cx, num3
call max
mov   mx,  ax
```

Stack Parameters

```
push num3  }
push num2  }  Reverse Order
push num1  }
call max
mov   mx, ax
```

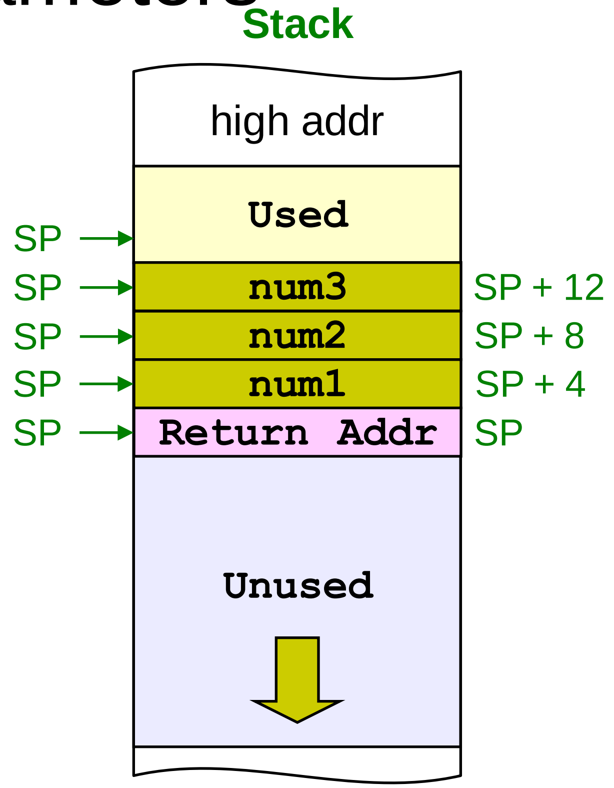
Passing Parameters thru Stack

- Calling procedure pushes parameters on the stack
- Procedure max receives parameters on the stack

Passing Parameters on the stack

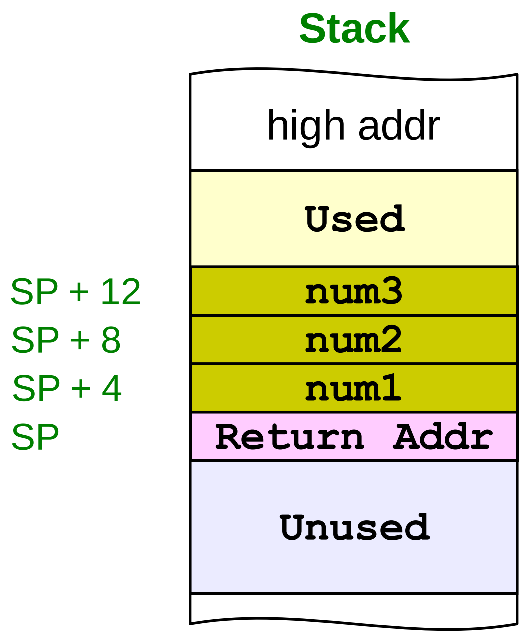
```
push num3
push num2
push num1
call max
mov  mx,AX
Add  SP,12
```

```
max PROC
    mov AX,[SP+4]
    cmp AX,[SP+8]
    jge @1
    mov AX,[SP+8]
@1: cmp AX,[SP+12]
    jge @2
    mov AX,[SP+12]
@2: ret
max ENDP
```



Accessing Parameters from Stack

- When parameters are passed on the stack
  - Parameter values appear after the return address
- We can use SP to access parameter values
  - [SP+4] for num1,
  - [SP+8] for num2, and
  - [SP+12] for num3



Parameters: Register V/S Stack

- Passing Parameters in Registers
  - Pros: Convenient, easier to use, and faster to access
  - Cons: Only few parameters can be passed
    - **A small number of registers are available**
    - **Often these registers are used and need to be saved on the stack**
    - **Pushing register values on stack negates their advantage**
- Passing Parameters on the Stack
  - Pros: Many parameters can be passed
    - **Large data structures and arrays can be passed**
  - Cons: Accessing parameters is not simple
    - **More overhead and slower access to parameters**

End of UNIT-1