Chapter 1. Basic Structure of Computers

Functional Units

Functional Units

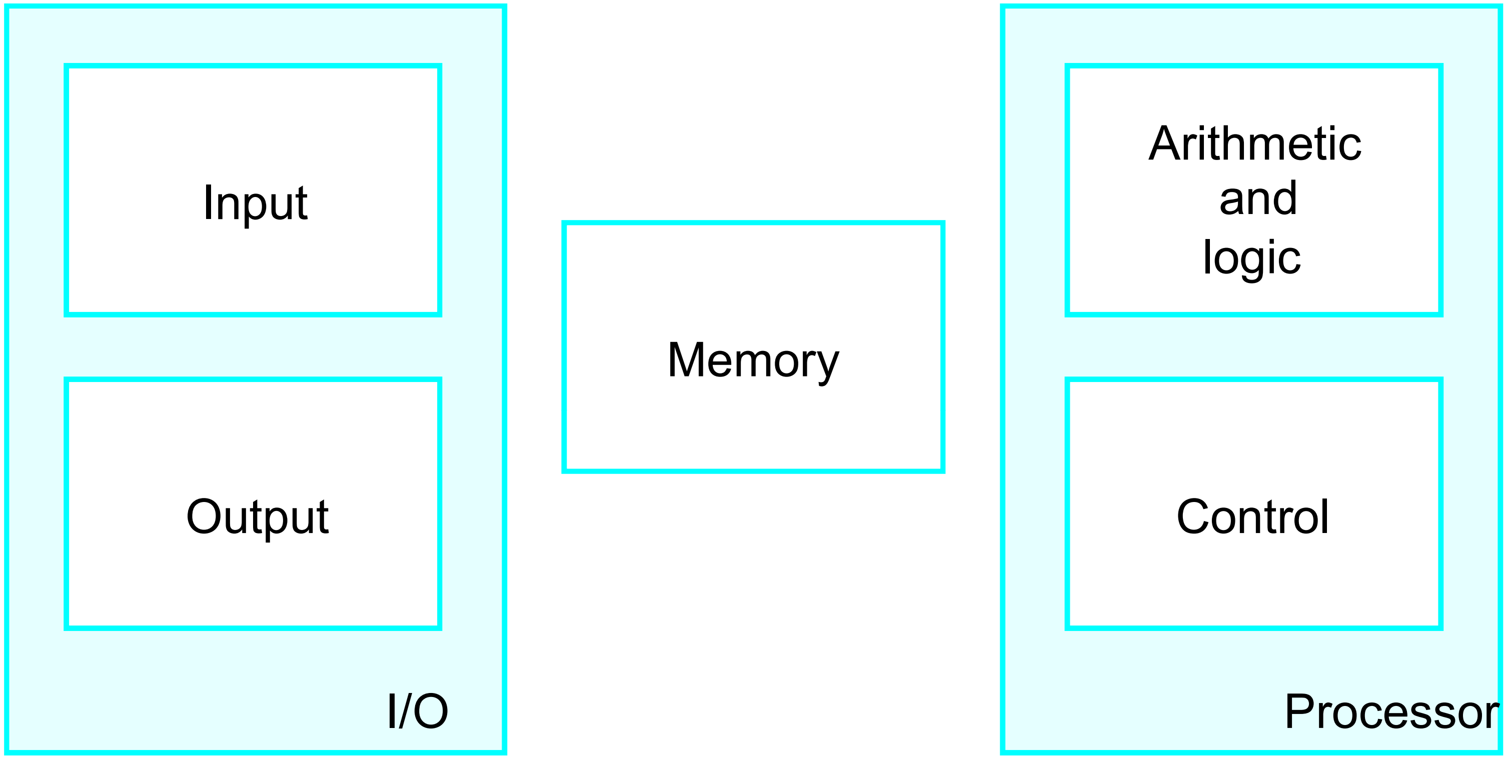


Figure 1.1. Basic functional units of a computer.

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 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 and cheaper

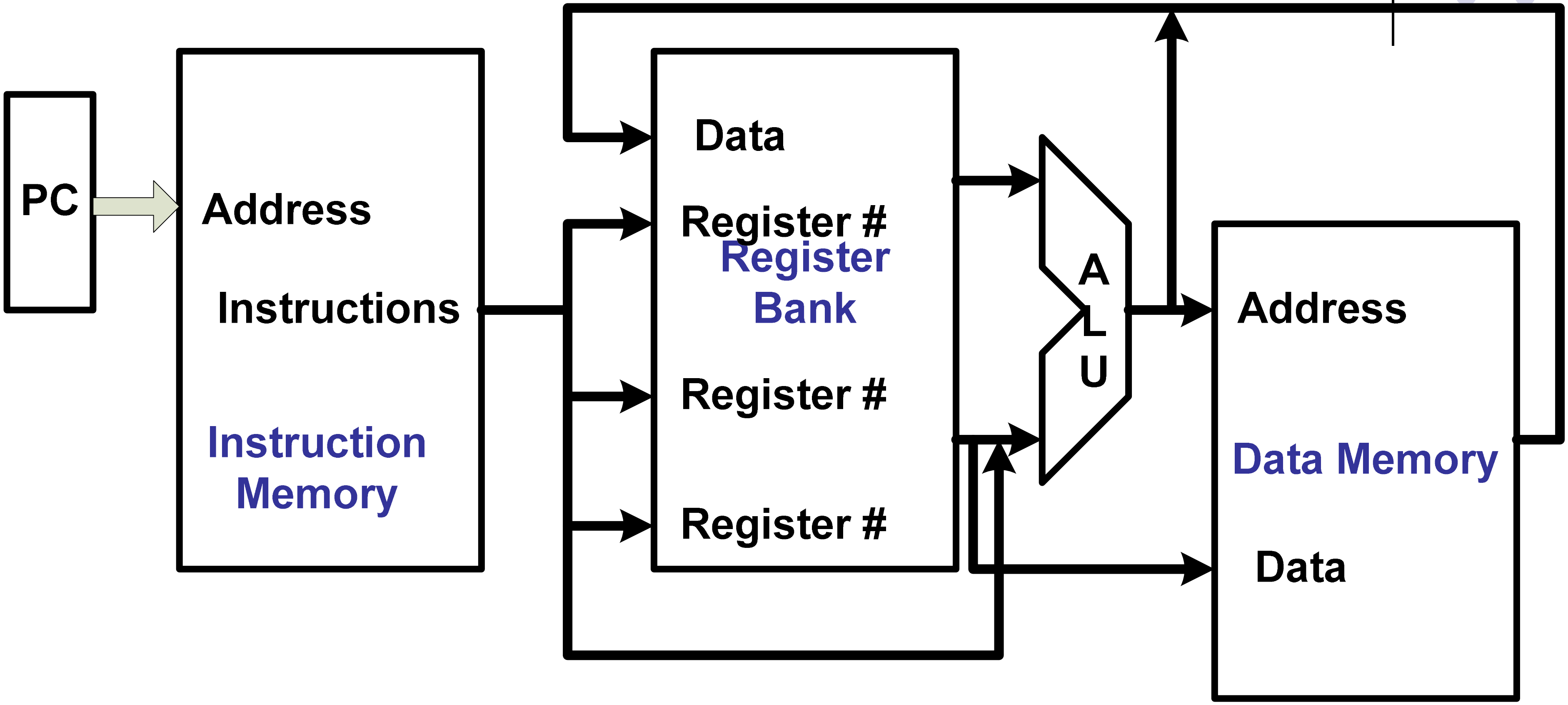
Arithmetic and Logic Unit (ALU)

- 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
- Fast control of ALU

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

The processor : Data Path and Control



- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (state elements)

Five Execution Steps

Step name	Action for R-type instructions	Action for Memory-reference Instructions	Action for branches	Action for jumps

Instruction fetch	IR = MEM[PC]
	PC = PC + 4			

Instruction decode/ register fetch	A = Reg[IR[25-21]]
	B = Reg[IR[20-16]]
	ALUOut = PC + (sign extend (IR[15-0])<<2)			

Execution, address computation, branch/jump completion	ALUOut = A op B	ALUOut = A+sign extend(IR[15-0])	IF(A==B) Then PC=ALUOut	PC=PC[31-28]||(IR[25-0]<<2)
Memory access or R-type completion	Reg[IR[15-11]] = ALUOut	Load:MDR =Mem[ALUOut] or Store:Mem[ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Basic Operational Concepts

Review

- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

A Typical Instruction

- Add LOCA, R0
- 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.
- 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.

Separate Memory Access and ALU Operation

- Load LOCA, R1
- Add R1, R0
- Whose contents will be overwritten?

Connection Between the Processor and the Memory

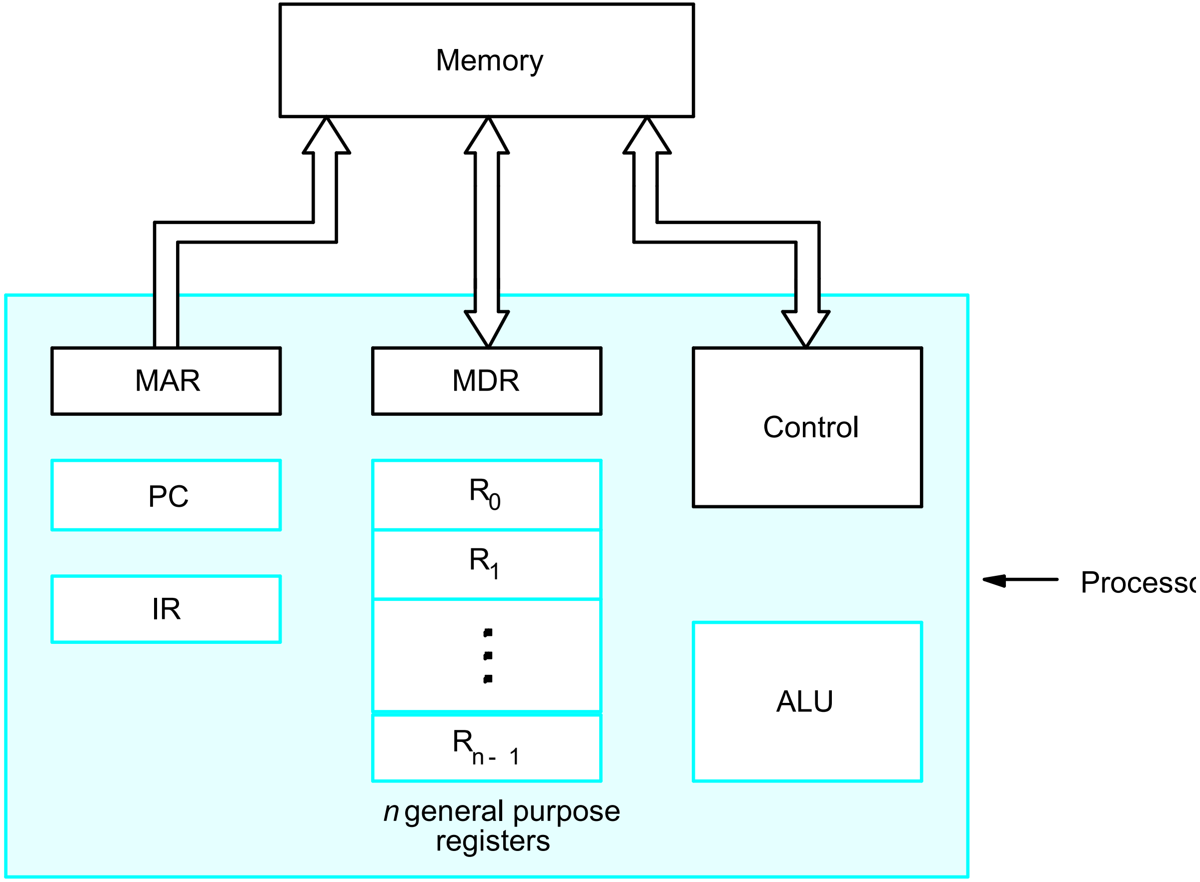


Figure 1.2. Connections between the processor and the memory.

Registers

- 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
  - 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
- 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

Bus Structure

- Single-bus

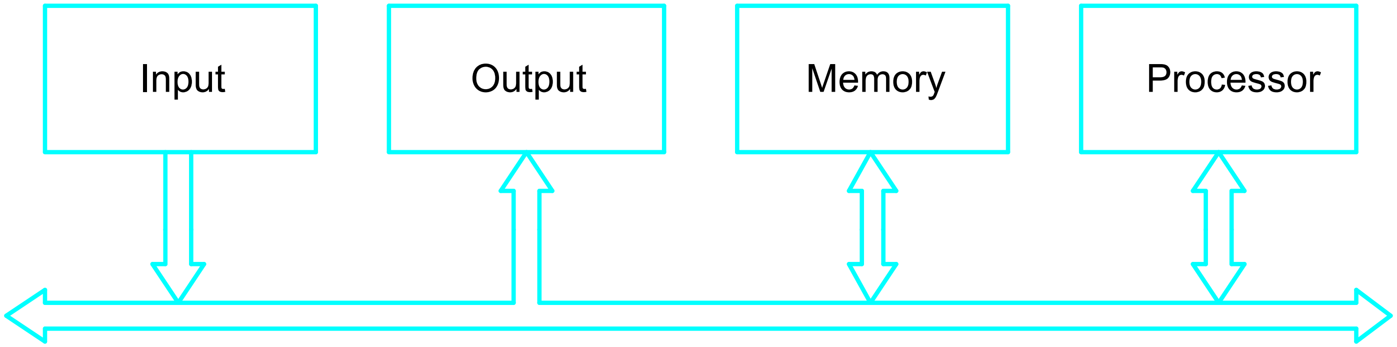


Figure 1.3. Single-bus structure.

Speed Issue

- Different devices have 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.

Performance

Performance

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

Performance

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

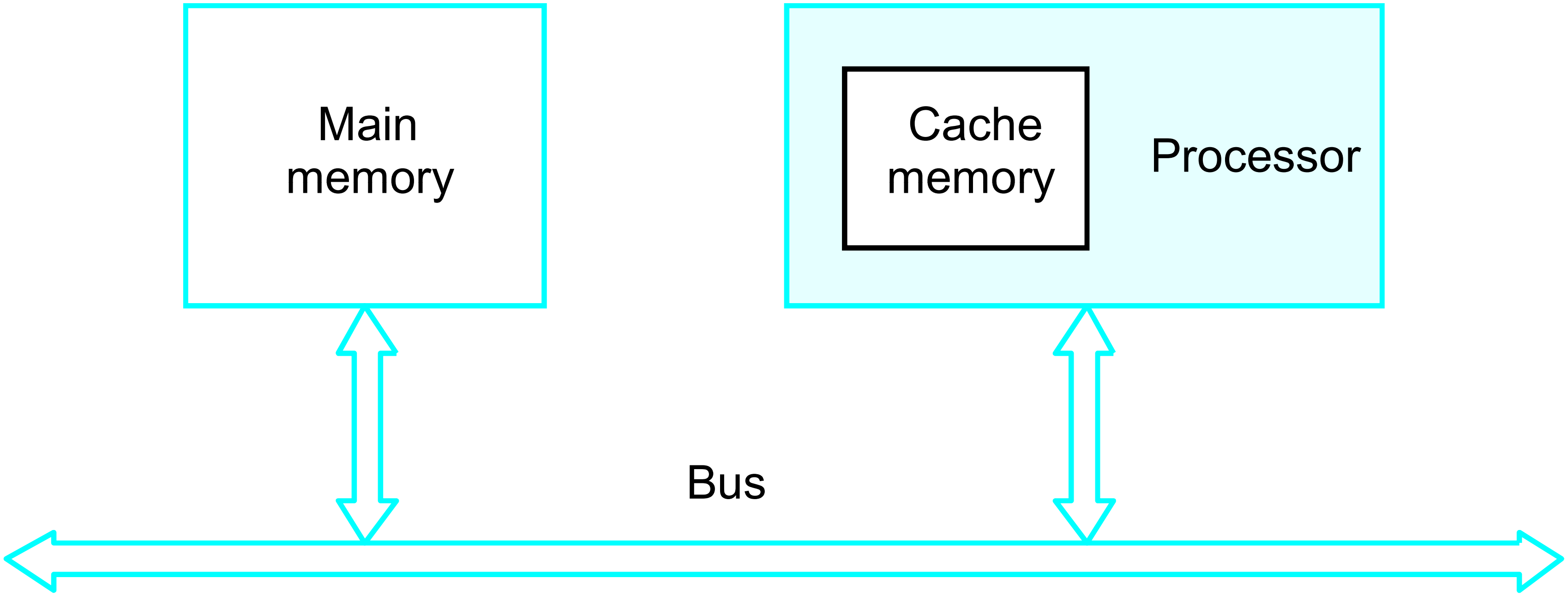


Figure 1.5. The processor cache.

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 – 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

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

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

- 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

Chapter 2. Machine Instructions and Programs

Objectives

- Machine instructions and program execution, including branching and subroutine call and return operations.
- Number representation and addition/subtraction in the 2’s-complement system.
- Addressing methods for accessing register and memory operands.
- Assembly language for representing machine instructions, data, and programs.
- Program-controlled Input/Output operations.

Number, Arithmetic Operations, and Characters

Signed Integer

- 3 major representations:

  Sign and magnitude

  One’s complement

  Two’s complement

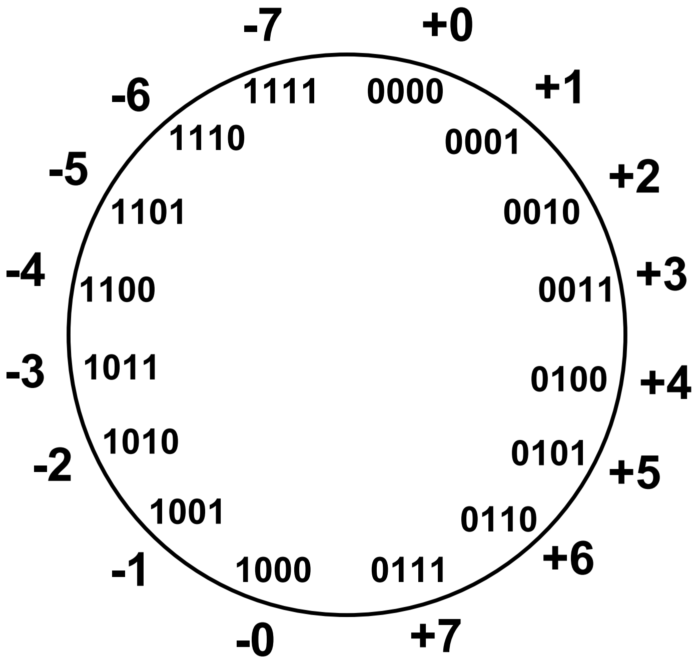
- Assumptions:

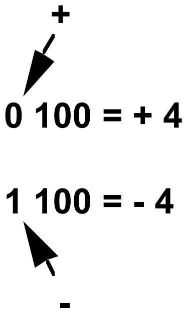
  4-bit machine word

  16 different values can be represented

  Roughly half are positive, half are negative

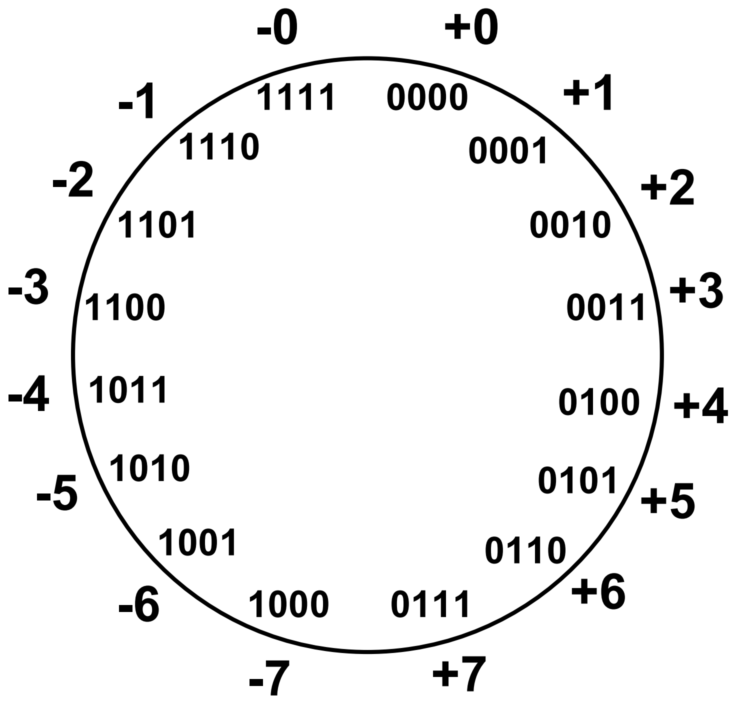
Sign and Magnitude Representation

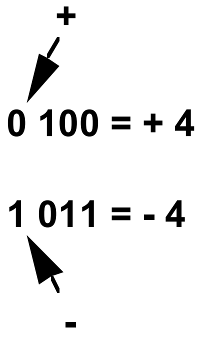




High order bit is sign: 0 = positive (or zero), 1 = negative
Three low order bits is the magnitude: 0 (000) thru 7 (111)
Number range for n bits = +/-2^{n-1} -1
Two representations for 0

One’s Complement Representation

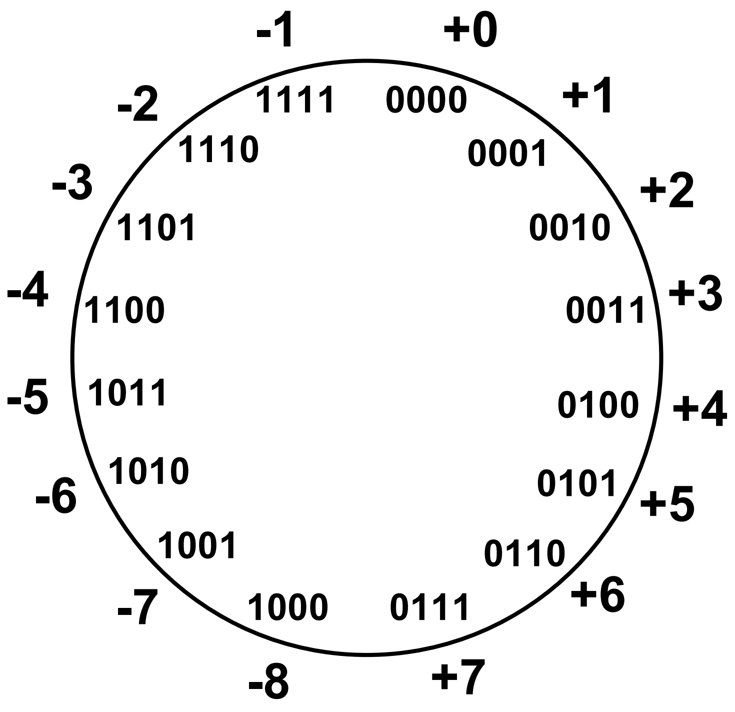


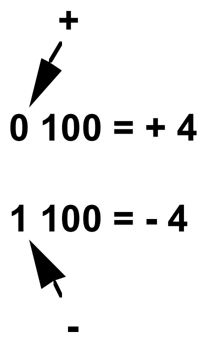


- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

Two's Complement Representation

like 1's comp except shifted one position clockwise





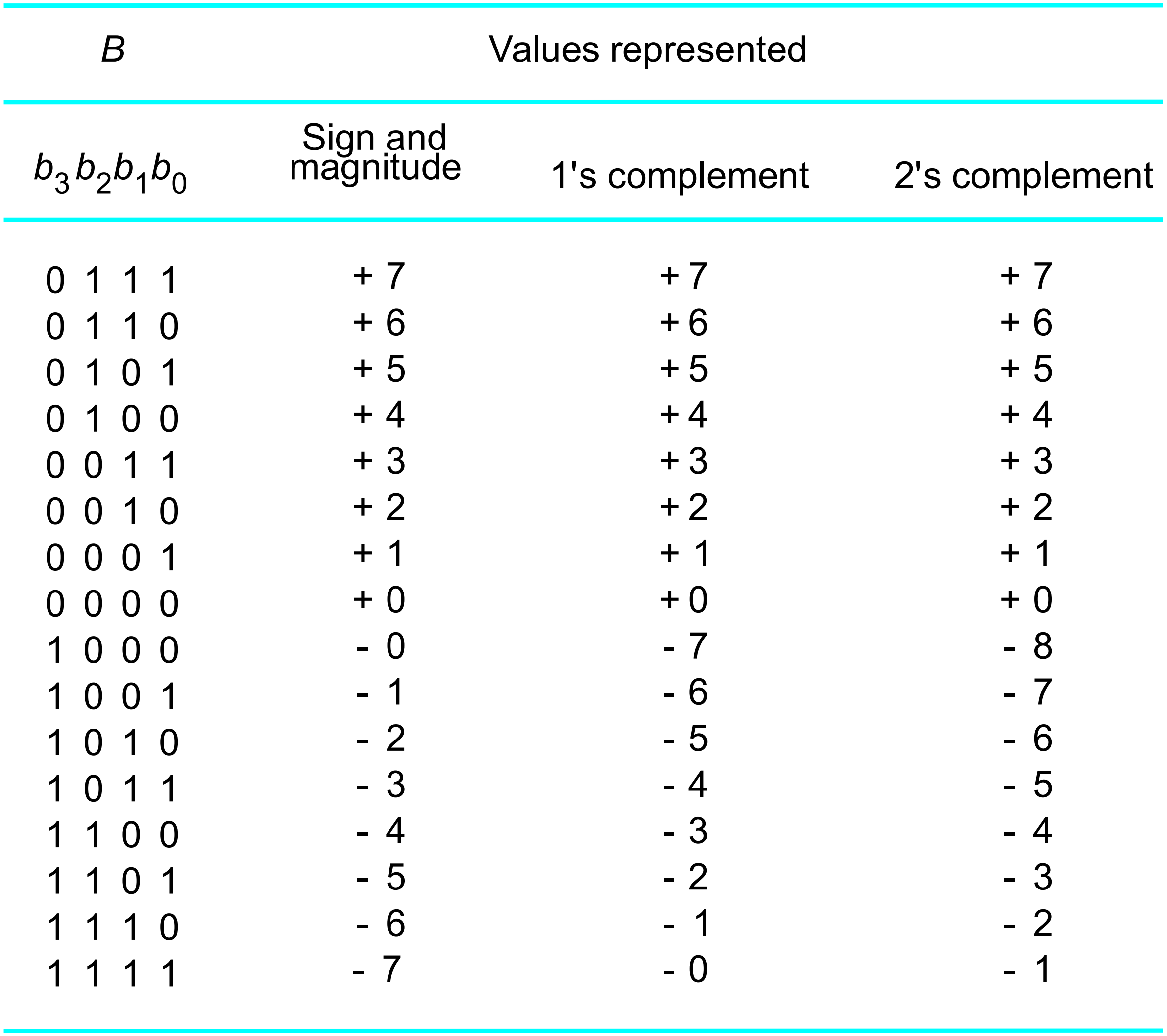
0 100 = + 4

1 100 = - 4

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

Binary, Signed-Integer Representations

Page 28



B	Values represented		
$b_3 b_2 b_1 b_0$	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

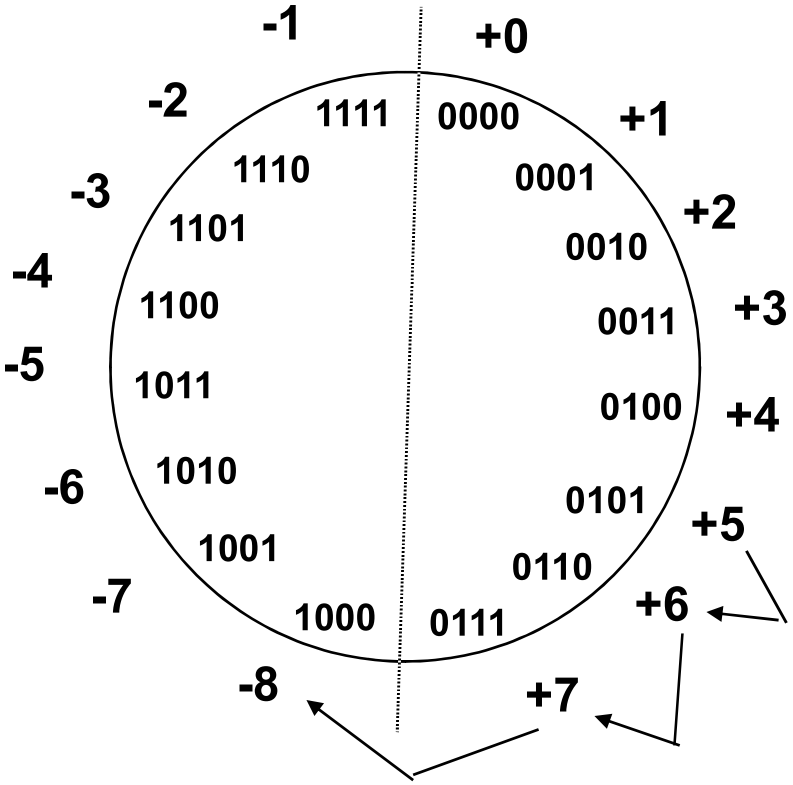
2’s-Complement Add and Subtract Operations

Page 31

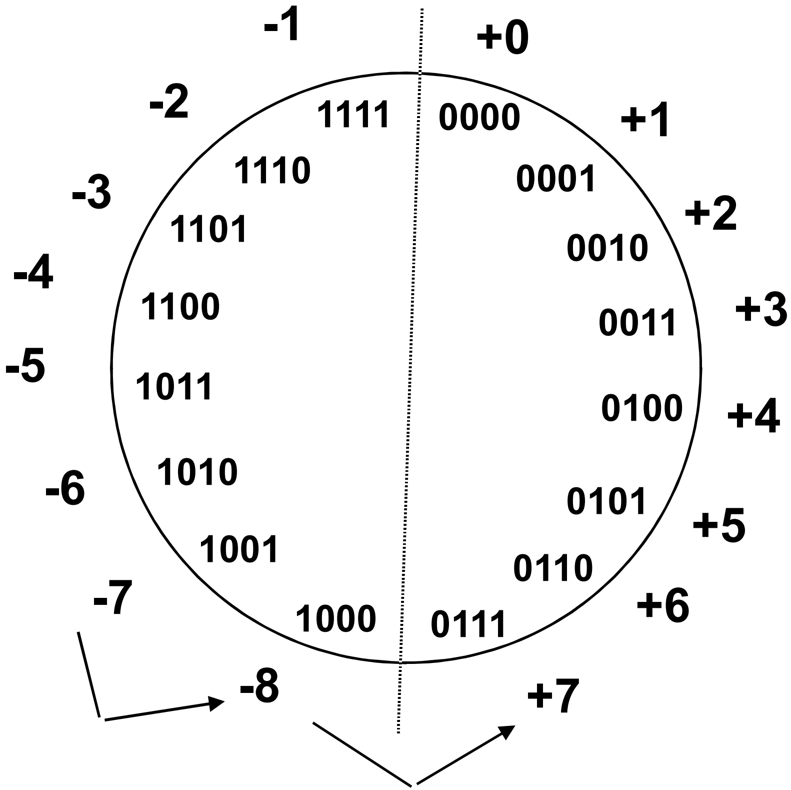
```
(a)      0 0 1 0     (+2)                   (b)      0 1 0 0     (+4)
       + 0 0 1 1     (+3)                          + 1 0 1 0     (- 6)
       ---------     ----                          ---------     ----
         0 1 0 1     (+5)                            1 1 1 0     (- 2)

(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

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

Overflow

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

No overflow

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

Overflow

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

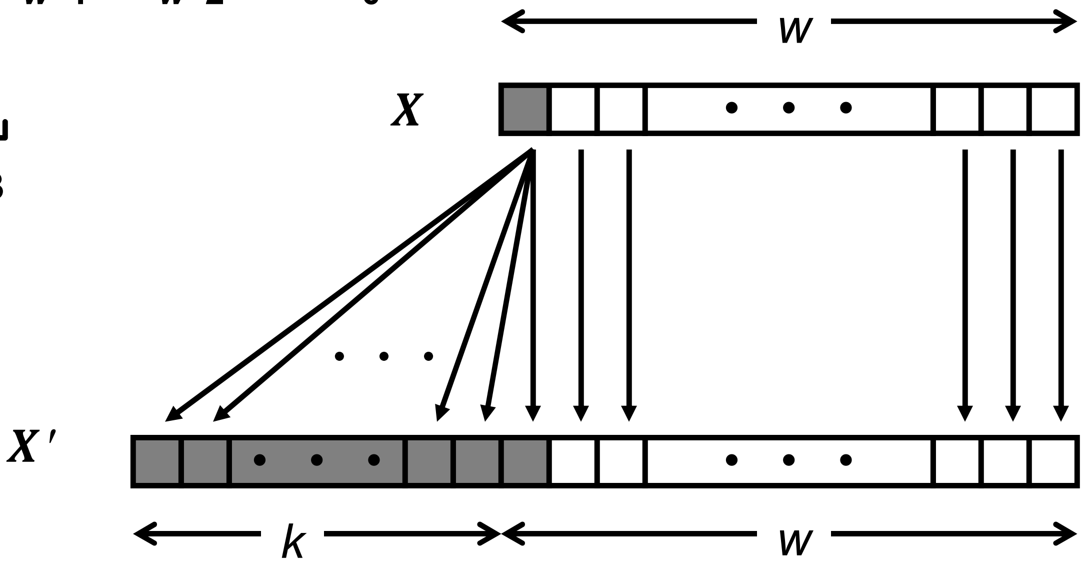
No overflow

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

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:
  - $\boldsymbol{X'} = \boldsymbol{x_{w-1}}, \ldots, \boldsymbol{x_{w-1}}, \boldsymbol{x_{w-1}}, \boldsymbol{x_{w-2}}, \ldots, \boldsymbol{x_0}$

k copies of MSB



Sign Extension Example

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 C4 92	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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.

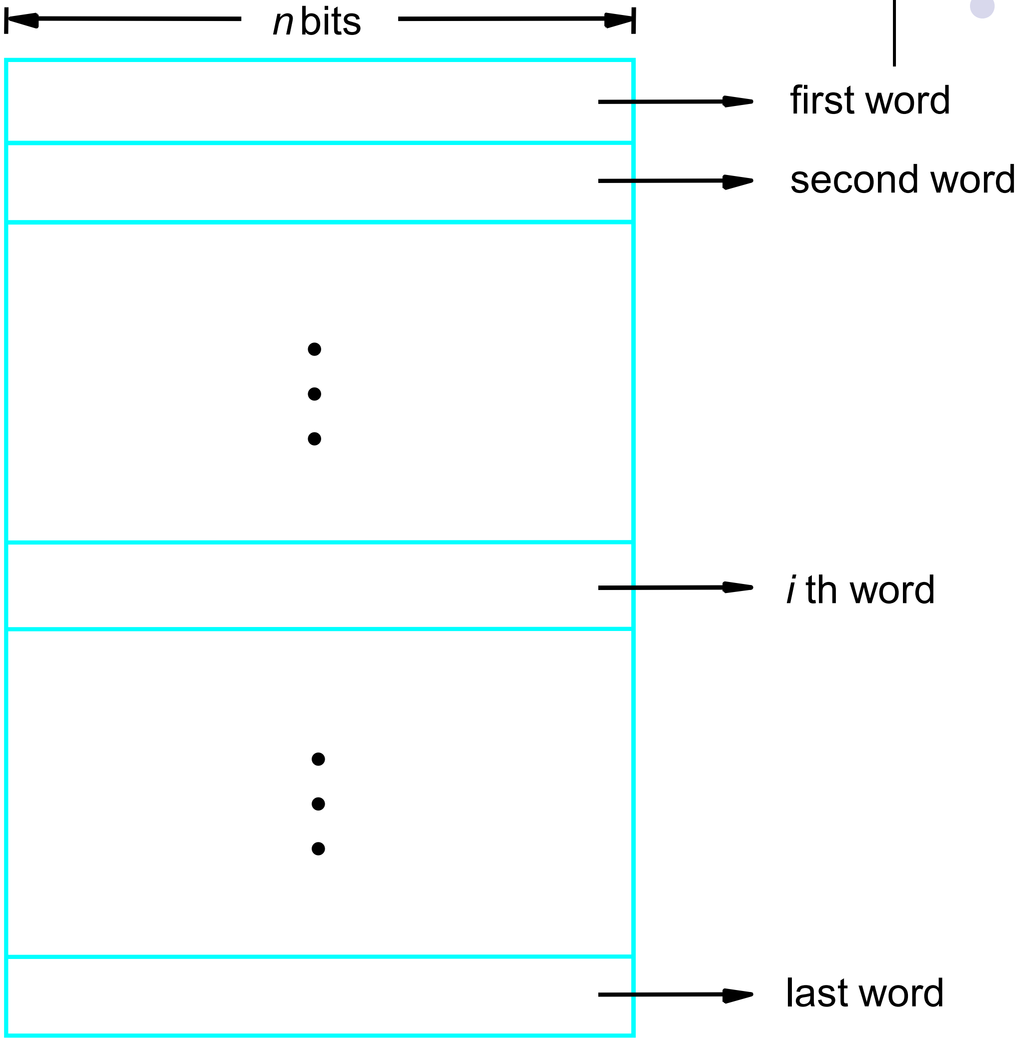
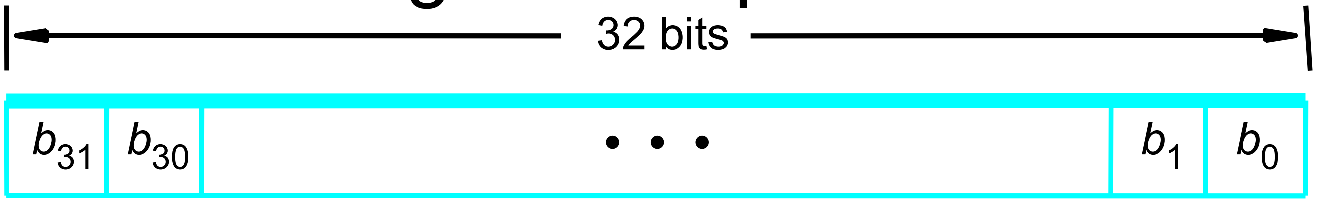


Figure 2.5. Memory words.

Memory Location, Addresses, and Operation

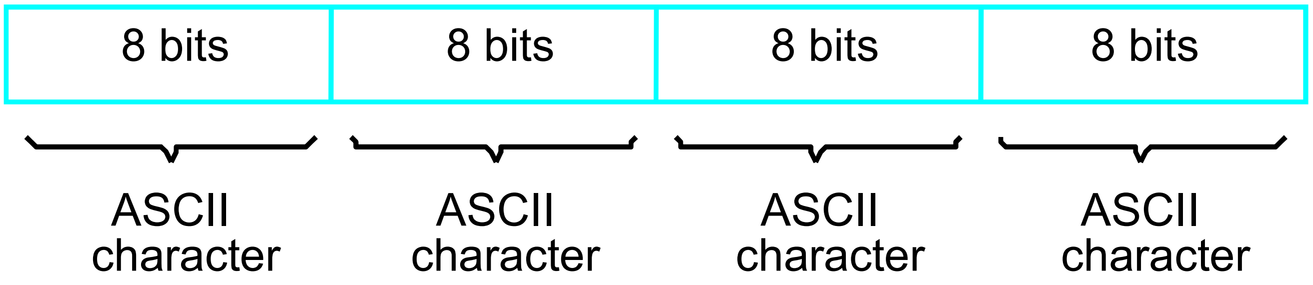
- 32-bit word length example



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: $2^{32} = 4G$ ($1G=2^{30}$)
- $1K(kilo)=2^{10}$
- $1T(tera)=2^{40}$

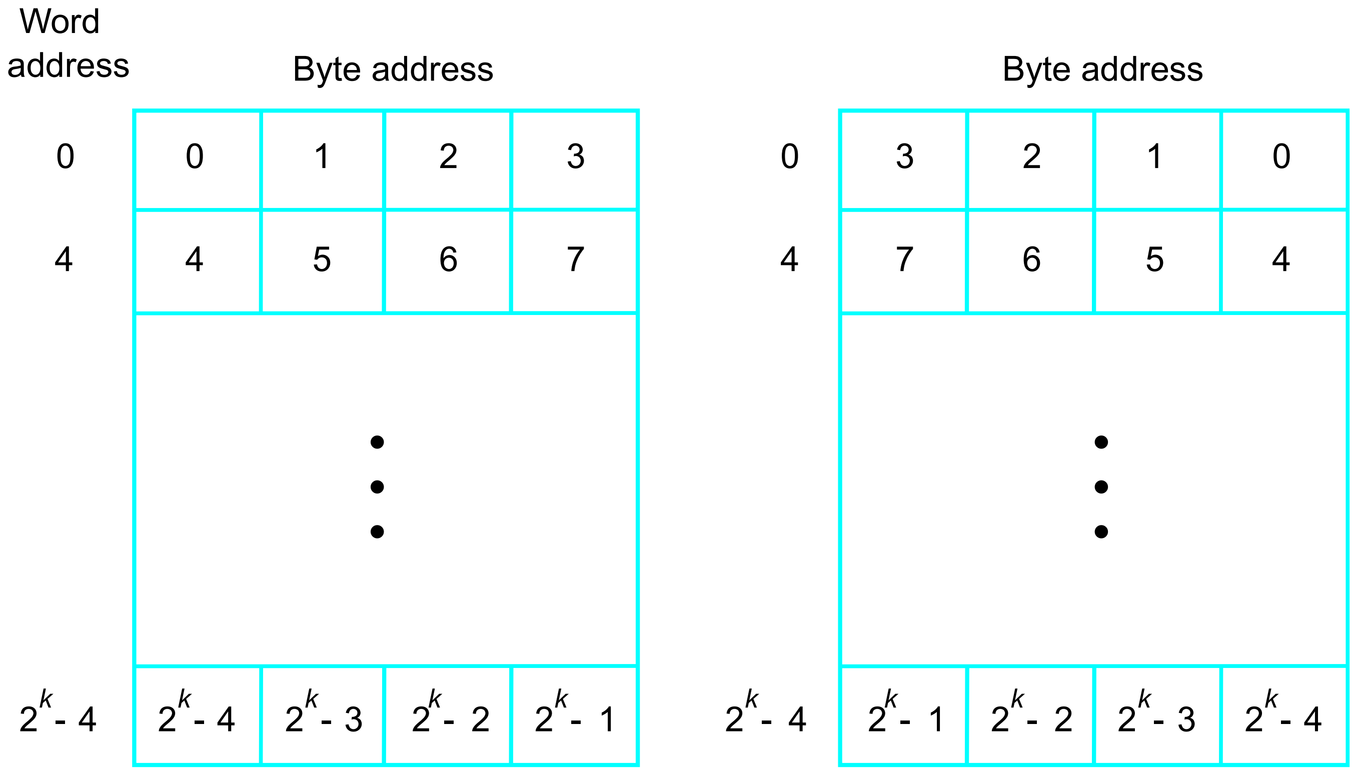
Memory Location, Addresses, and Operation

- It is impractical to assign distinct addresses to individual bit locations in the memory.
- The most practical assignment is to have successive addresses refer to successive byte locations in the memory – byte-addressable memory.
- Byte locations have addresses 0, 1, 2, … If word length is 32 bits, they successive words are located at addresses 0, 4, 8,…

Big-Endian and Little-Endian Assignments

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

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



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	⋮			
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

	Byte address			
0	3	2	1	0
4	7	6	5	4
	⋮			
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment

Figure 2.7. Byte and word addressing.

Memory Location, Addresses, and Operation

- Address ordering of bytes
- Word alignment
  - Words are said to be aligned in memory if they begin at a byte addr. that is a multiple of the num of bytes in a word.
    - 16-bit word: word addresses: 0, 2, 4,….
    - 32-bit word: word addresses: 0, 4, 8,….
    - 64-bit word: word addresses: 0, 8,16,….
- Access numbers, characters, and character strings

Memory Operation

- Load (or Read or Fetch)
  - Copy the content. The memory content doesn’t change.
  - Address – Load
  - Registers can be used
- Store (or Write)
  - Overwrite the content in memory
  - Address and Data – Store
  - 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

- 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←[LOC], R3 ←[R1]+[R2])
- Register Transfer Notation (RTN)

Assembly Language Notation

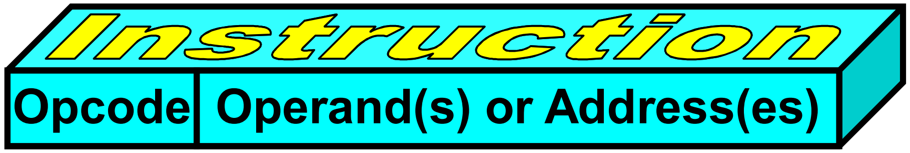
- Represent machine instructions and programs.
- Move LOC, R1 = R1←[LOC]
- Add R1, R2, R3 = R3 ←[R1]+[R2]

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 ← R2 + R3
- Two-Address Instructions
  - ADD R1, R2 R1 ← R1 + R2
- One-Address Instructions
  - ADD M AC ← AC + M[AR]
- Zero-Address Instructions
  - ADD TOS ← TOS + (TOS – 1)
- RISC Instructions
  - Lots of registers. Memory is restricted to Load & Store



Instruction Formats

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

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

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Two-Address

```
1.  MOV   R1, A        ; R1 ← M[A]
2.  ADD   R1, B        ; R1 ← R1 + M[B]
3.  MOV   R2, C        ; R2 ← M[C]
4.  ADD   R2, D        ; R2 ← R2 + M[D]
5.  MUL   R1, R2       ; R1 ← R1 ∗ R2
6.  MOV   X, R1        ; M[X] ← R1
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- One-Address

```
1. LOAD  A        ; AC ← M[A]
2. ADD   B        ; AC ← AC + M[B]
3. STORET         ; M[T] ← AC
4. LOAD  C        ; AC ← M[C]
5. ADD   D        ; AC ← AC + M[D]
6. MUL   T        ; AC ← AC ∗ M[T]
7. STOREX         ; M[X] ← AC
```

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

- Zero-Address

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

Instruction Formats

Example: Evaluate (A+B) ∗ (C+D)

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

Using Registers

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

Instruction Execution and Straight-Line Sequencing

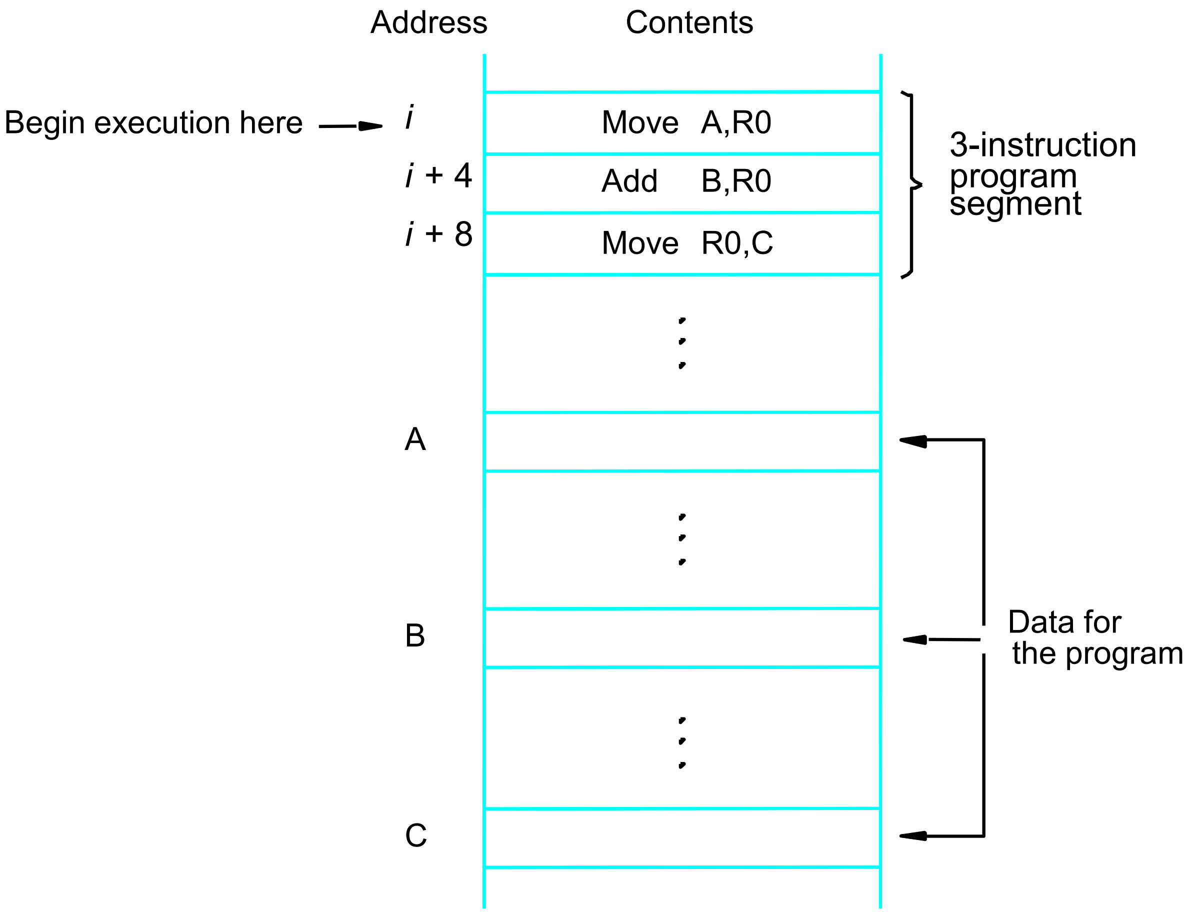


Figure 2.8. A program for $C \leftarrow [A] + [B]$.

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

Page 43

Branching

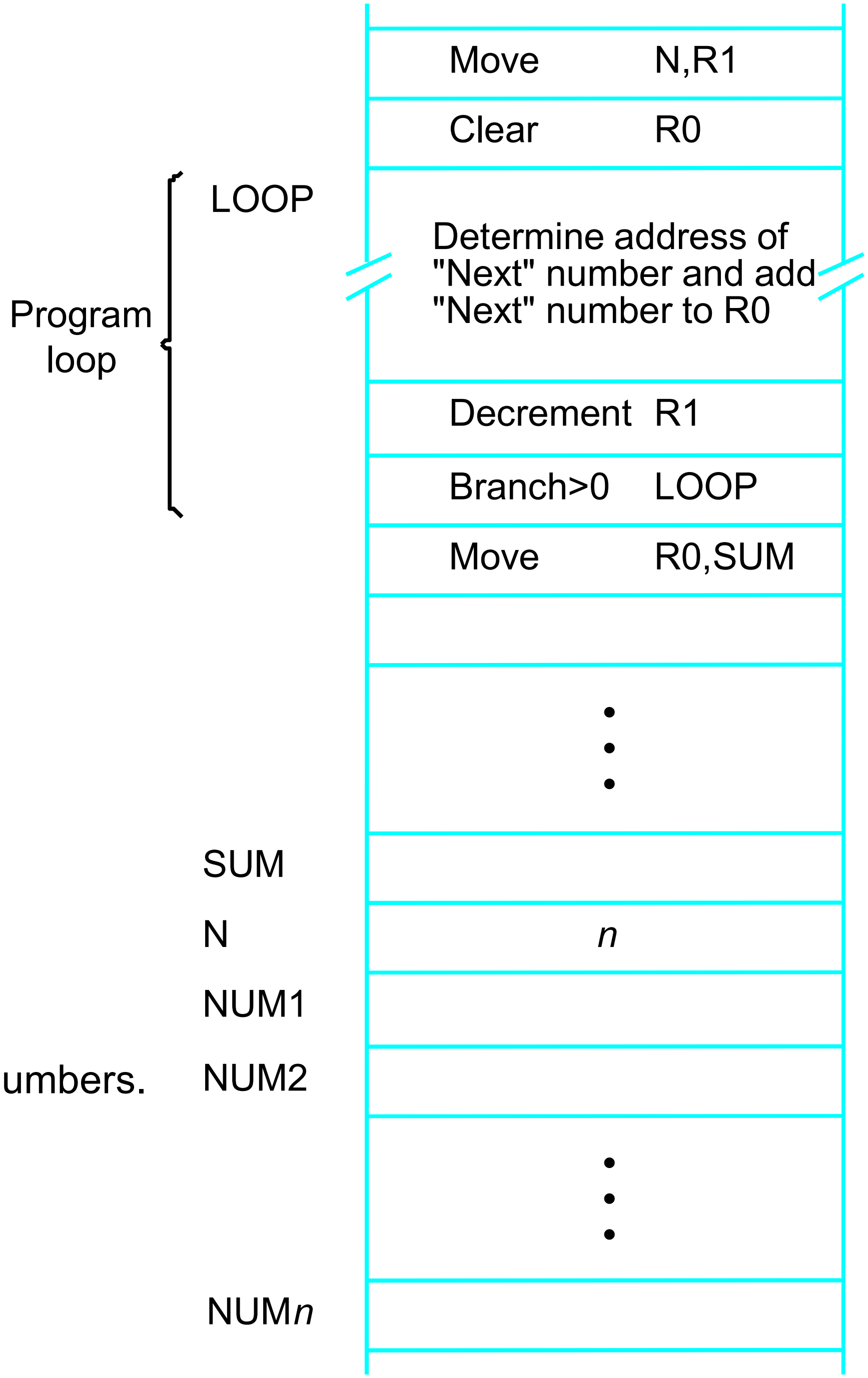
i	Move	NUM1,R0
i + 4	Add	NUM2,R0
i + 8	Add	NUM3,R0
	⋮	
i + 4*n*- 4	Add	NUM*n*,R0
i + 4*n*	Move	R0,SUM
	⋮	
SUM		
NUM1		
NUM2		
	⋮	
NUM*n*		

Figure 2.9. A straight-line program for adding *n* numbers.

Branching

Branch target

Conditional branch



	Move N,R1
	Clear R0
LOOP	Determine address of "Next" number and add "Next" number to R0
	Decrement R1
	Branch>0 LOOP
	Move R0,SUM
	⋮
SUM	
N	*n*
NUM1	
NUM2	
	⋮
NUM*n*	

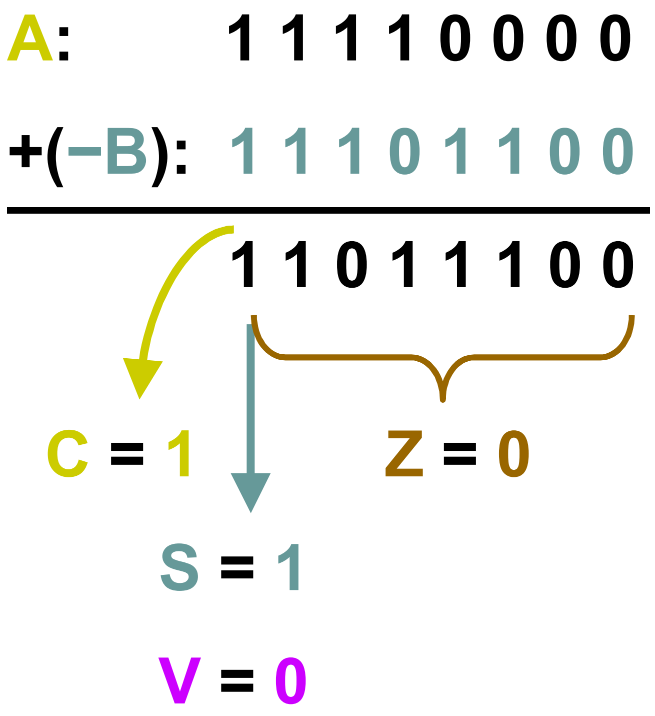
Figure 2.10. Using a loop to add *n* numbers.

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

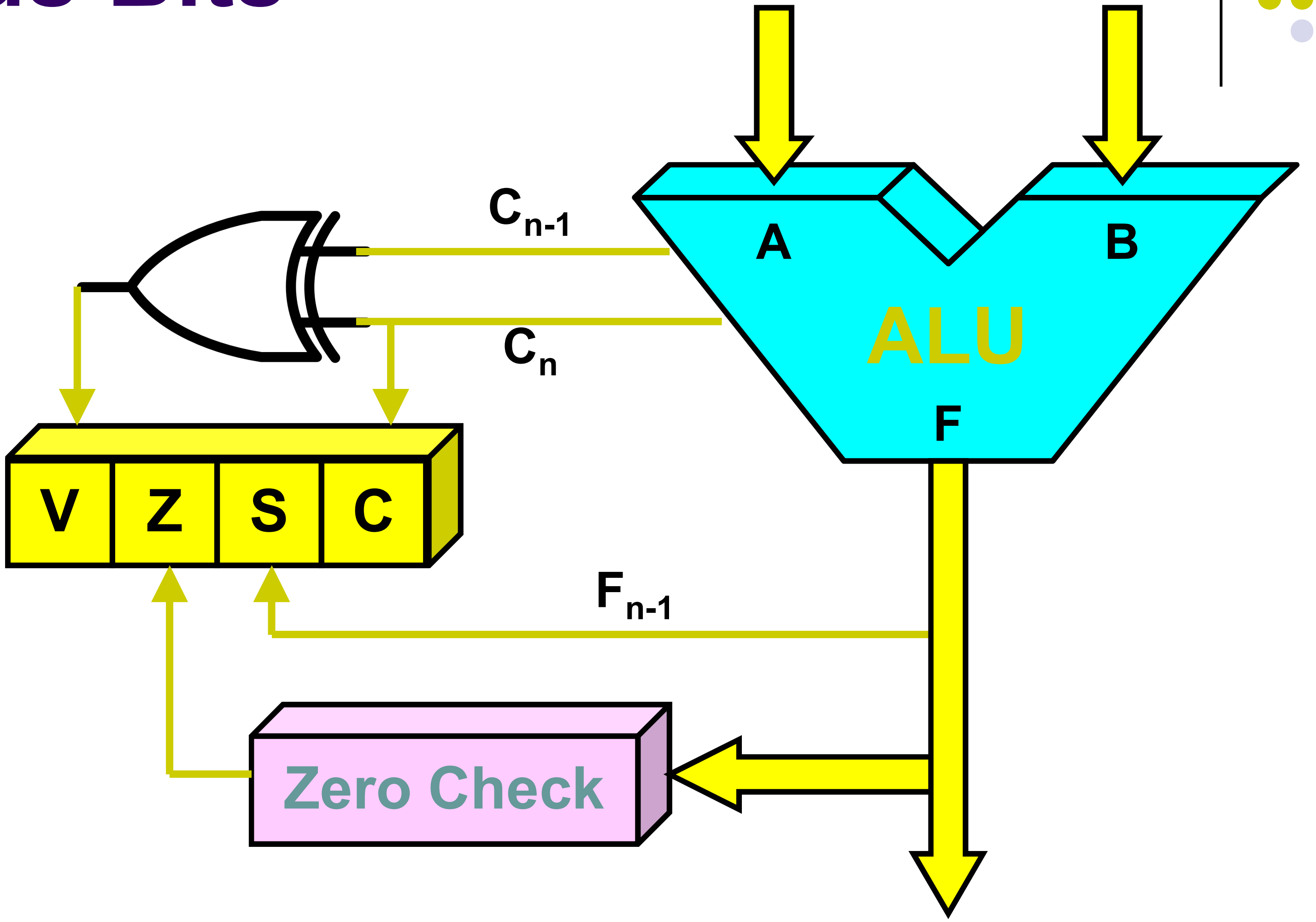
C = 1

Z = 0

S = 1

V = 0

Status Bits

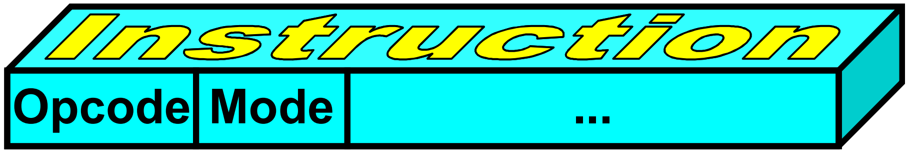


Addressing Modes

Generating Memory Addresses

- How to specify the address of branch target?
- Can we give the memory operand address directly in a single Add instruction in the loop?
- Use a register to hold the address of NUM1; then increment by 4 on each pass through the loop.

Addressing Modes

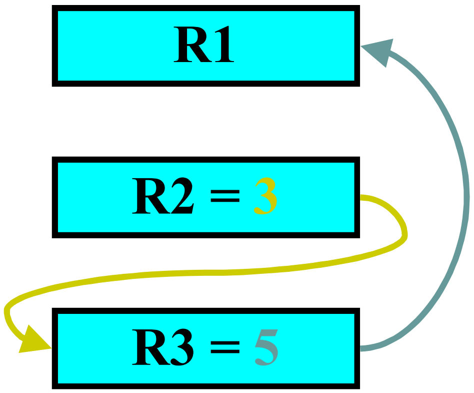


- Implied
  - AC is implied in “ADD M[AR]” in “One-Address” instr.
  - TOS is implied in “ADD” in “Zero-Address” instr.
- Immediate
  - The use of a constant in “MOV R1, 5”, i.e. R1 ← 5
- Register
  - Indicate which register holds the operand

Addressing Modes

- Register Indirect
  - Indicate the register that holds the number of the register that holds the operand

    MOV R1, (R2)



R1

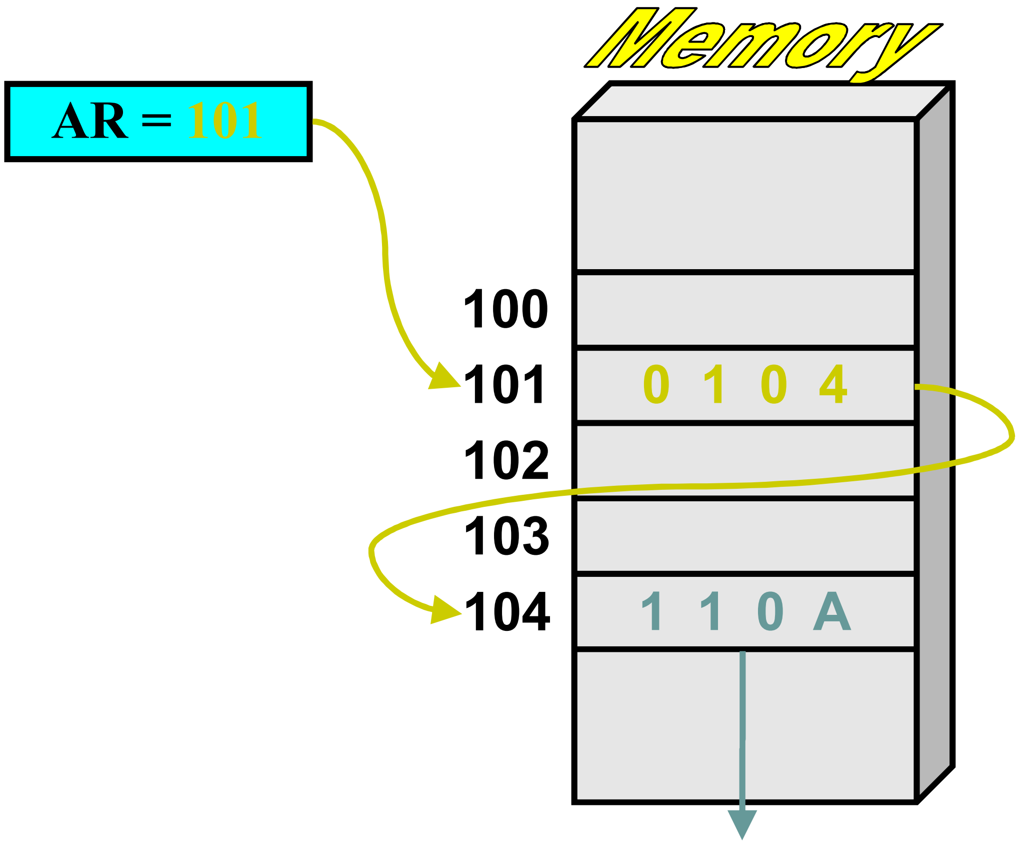
R2 = 3

R3 = 5

- Autoincrement / Autodecrement
  - Access & update in 1 instr.
- Direct Address
  - Use the given address to access a memory location

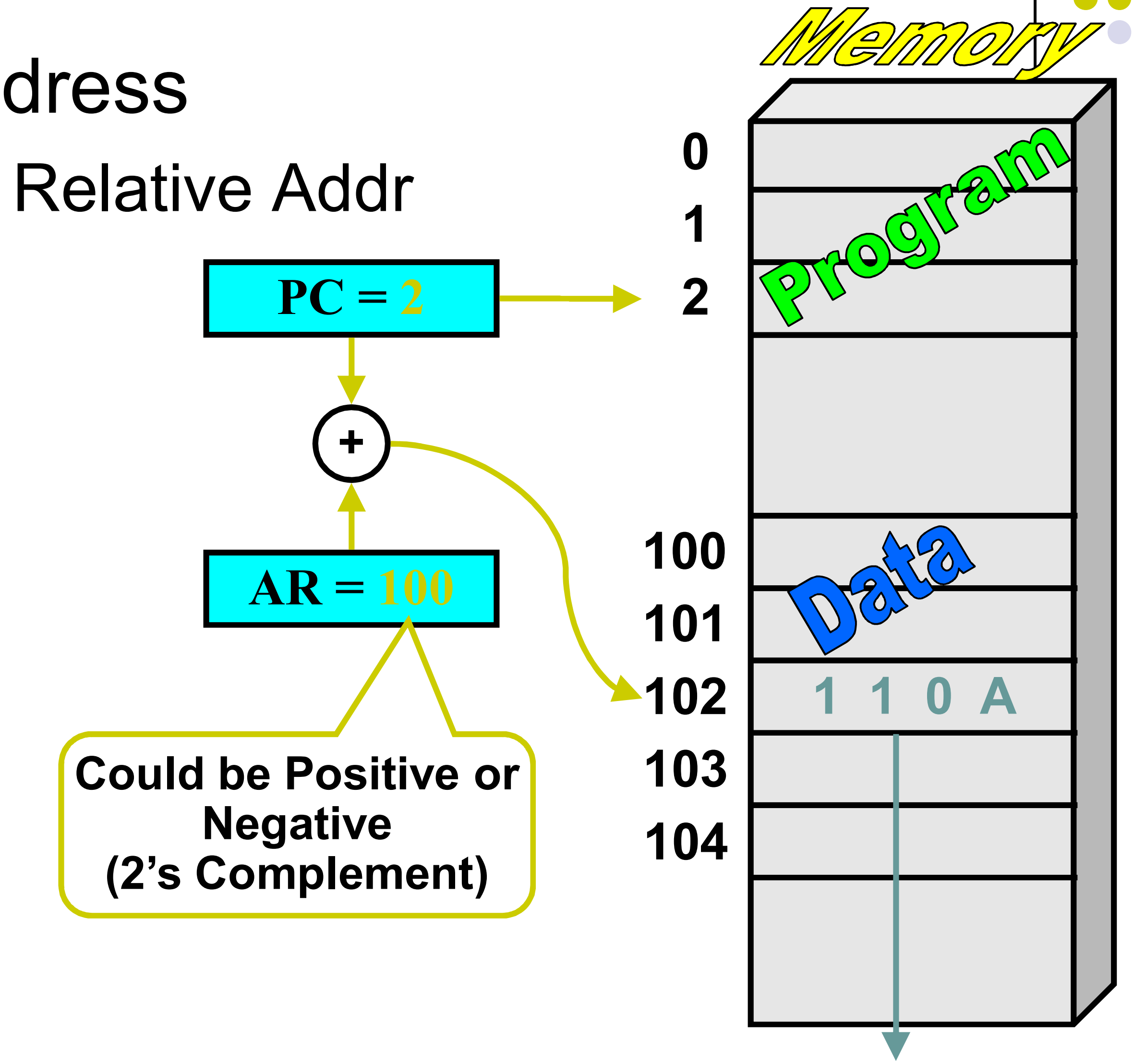
Addressing Modes

- Indirect Address
  - Indicate the memory location that holds the address of the memory location that holds the data



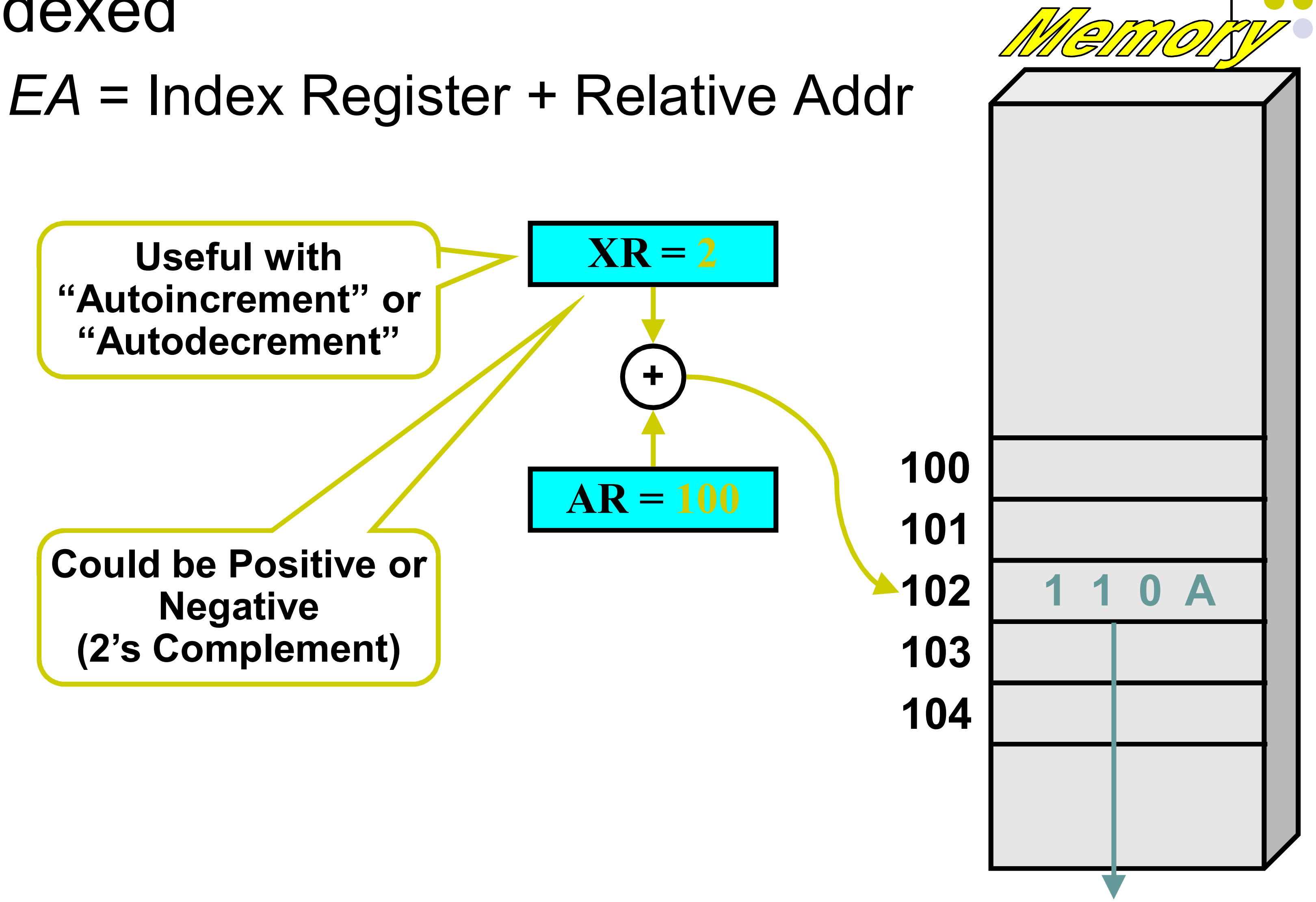
Addressing Modes

- Relative Address
  - *EA* = PC + Relative Addr



Addressing Modes

- Indexed
  - *EA* = Index Register + Relative Addr

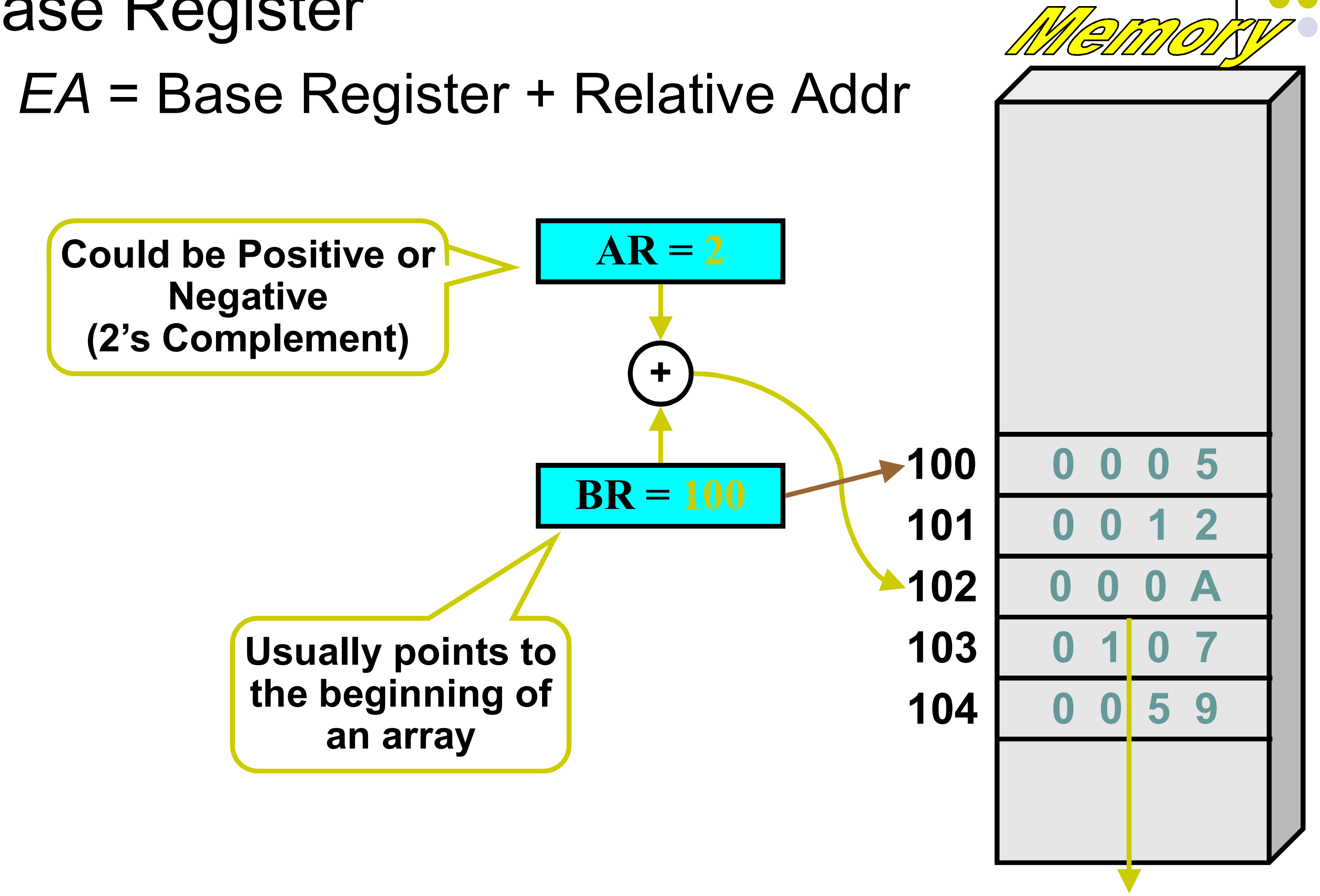


Useful with "Autoincrement" or "Autodecrement"

Could be Positive or Negative (2's Complement)

Addressing Modes

- Base Register
  - *EA* = Base Register + Relative Addr



Could be Positive or Negative (2’s Complement)

Usually points to the beginning of an array

100	0 0 0 5
101	0 0 1 2
102	0 0 0 A
103	0 1 0 7
104	0 0 5 9

Addressing Modes

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

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
Basewith index	(R*i* ,R*j*)	EA = [R*i*] + [R*j*]
Basewith 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*]

Indexing and Arrays

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.
- Index register
- $X(R_i)$: $EA = X + [R_i]$
- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.
- If X is shorter than a word, sign-extension is needed.

Indexing and Arrays

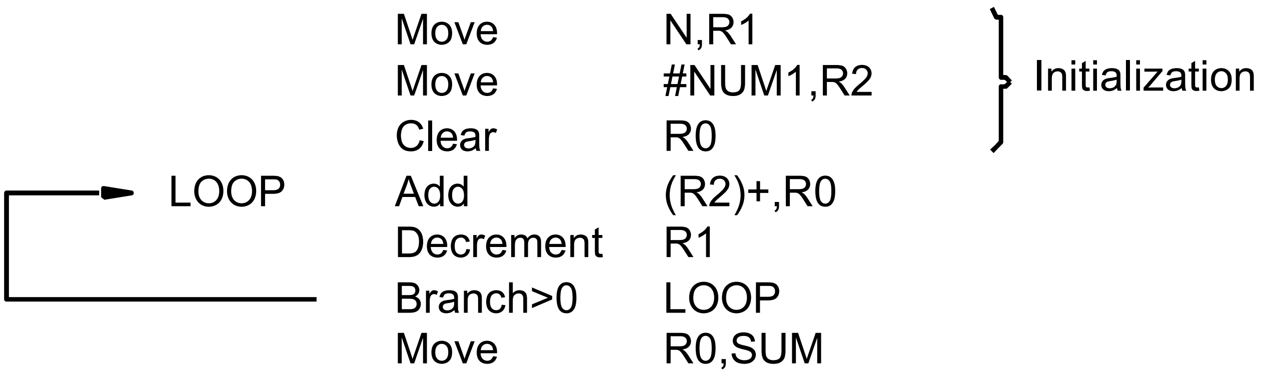
- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.
- Several variations:
  (R_i, R_j): $EA = [R_i] + [R_j]$
  $X(R_i, R_j)$: $EA = X + [R_i] + [R_j]$

Relative Addressing

- Relative 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
- Branch>0 LOOP
- This location is computed by specifying it as an offset from the current value of PC.
- Branch target may be either before or after the branch instruction, the offset is given as a singed num.

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.

Assembly Language

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	$AC \leftarrow \mathrm{NBR}$
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

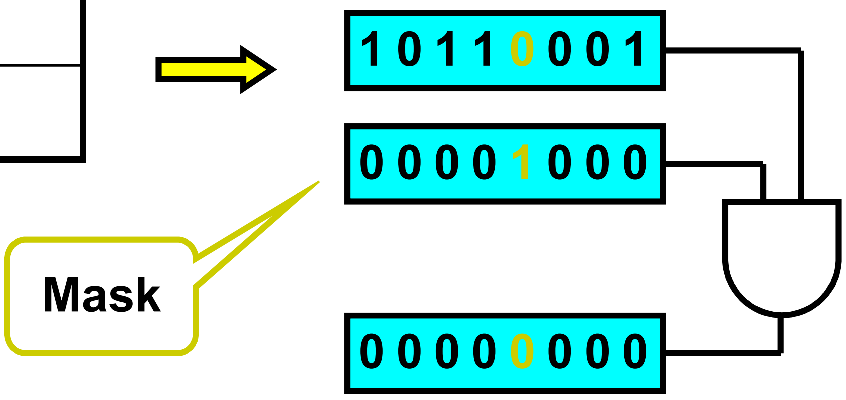
Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

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



1 0 1 1 0 0 0 1

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0

Mask

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

Basic Input/Output Operations

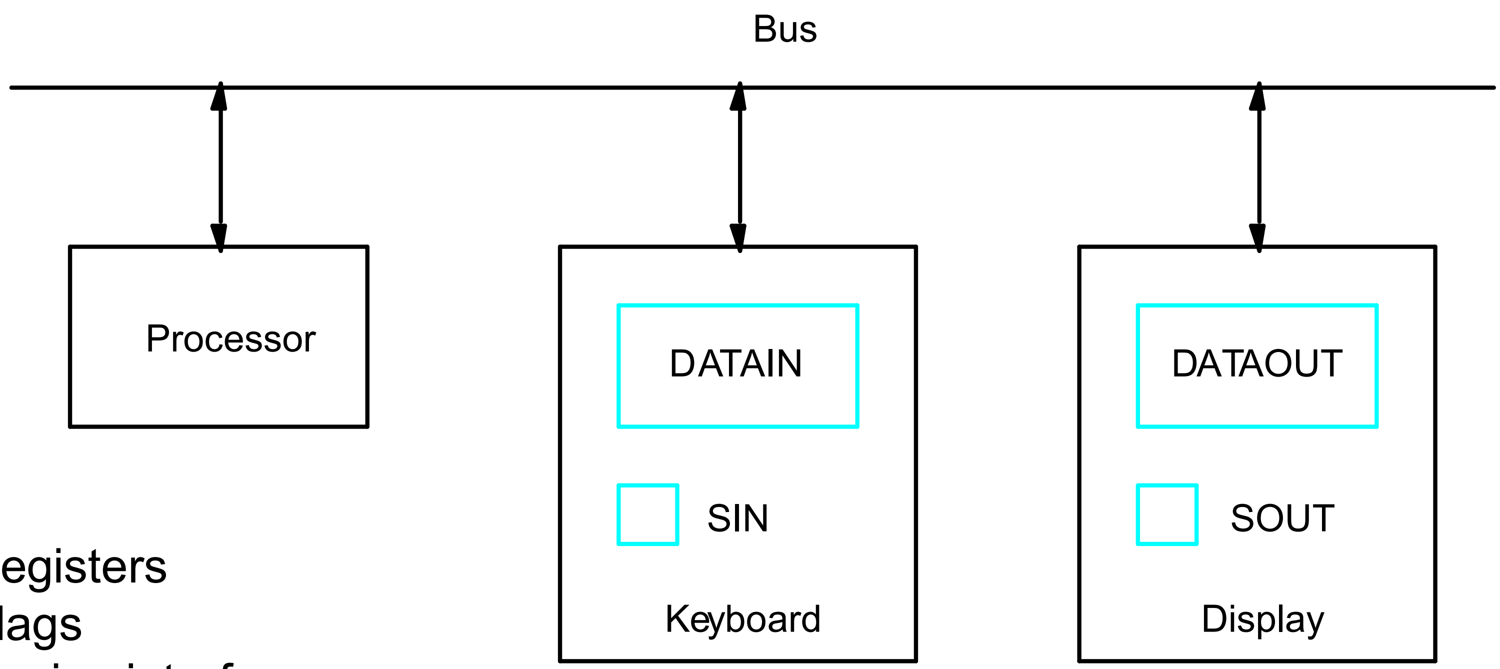
I/O

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

Program-Controlled I/O Example

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

Program-Controlled I/O Example



- Registers
- Flags
- Device interface

Figure 2.19 Bus connection for processor, keyboard, and display

Program-Controlled I/O Example

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

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

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

Program-Controlled I/O Example

- Memory-Mapped I/O – some memory address values are used to refer to peripheral device buffer registers. No special instructions are needed. Also use device status registers.

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

Program-Controlled I/O Example

- Assumption – the initial state of SIN is 0 and the initial state of SOUT is 1.
- Any drawback of this mechanism in terms of efficiency?
  - Two wait loops→processor execution time is wasted
- Alternate solution?
  - Interrupt

Stacks

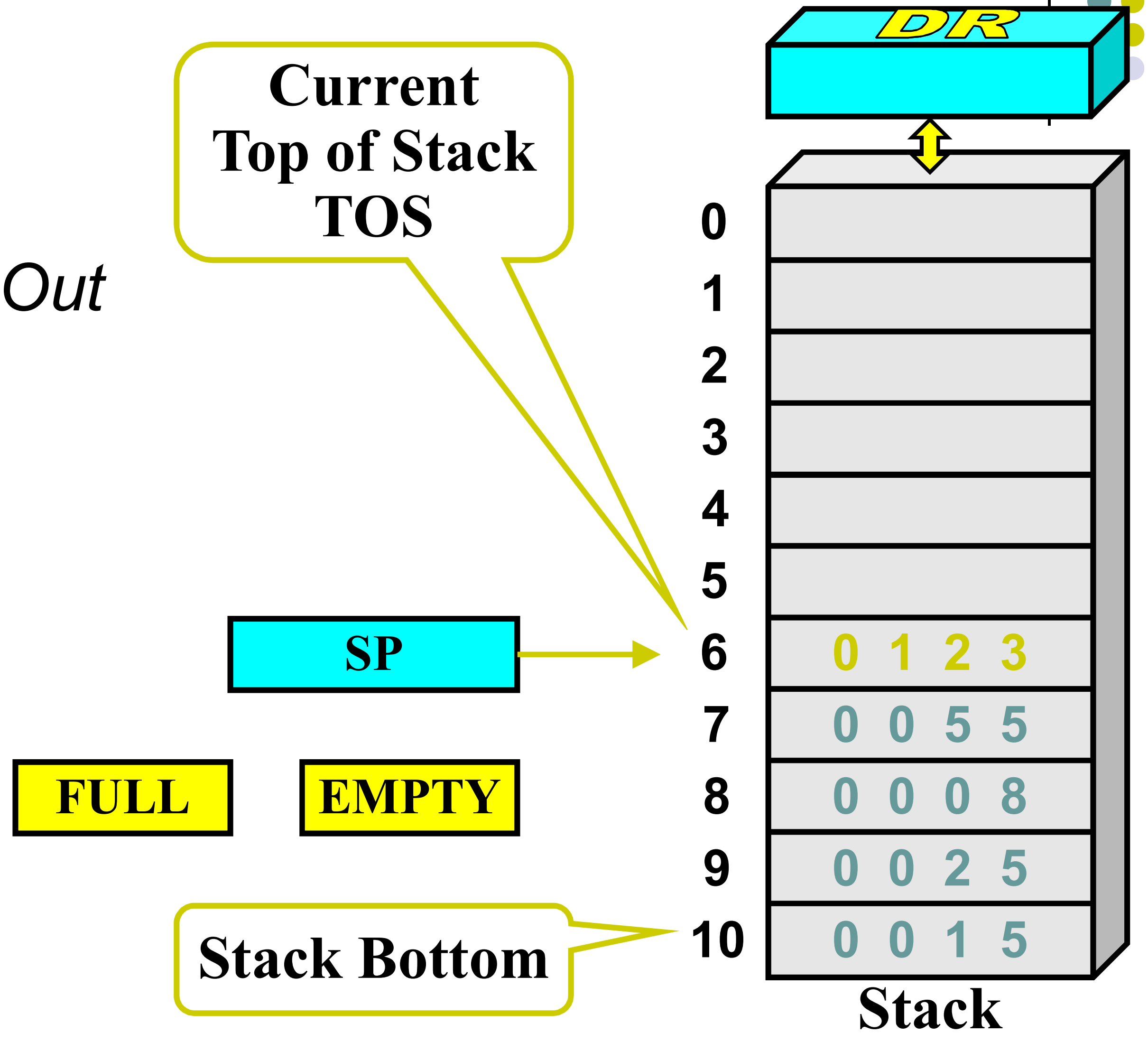
Home Work

- For each Addressing modes mentioned before, state one example for each addressing mode stating the specific benefit for using such addressing mode for such an application.

Stack Organization

- LIFO

  Last In First Out



DR

Current Top of Stack TOS

SP

FULL

EMPTY

Stack Bottom

0

1

2

3

4

5

6 0 1 2 3

7 0 0 5 5

8 0 0 0 8

9 0 0 2 5

10 0 0 1 5

Stack

Stack Organization

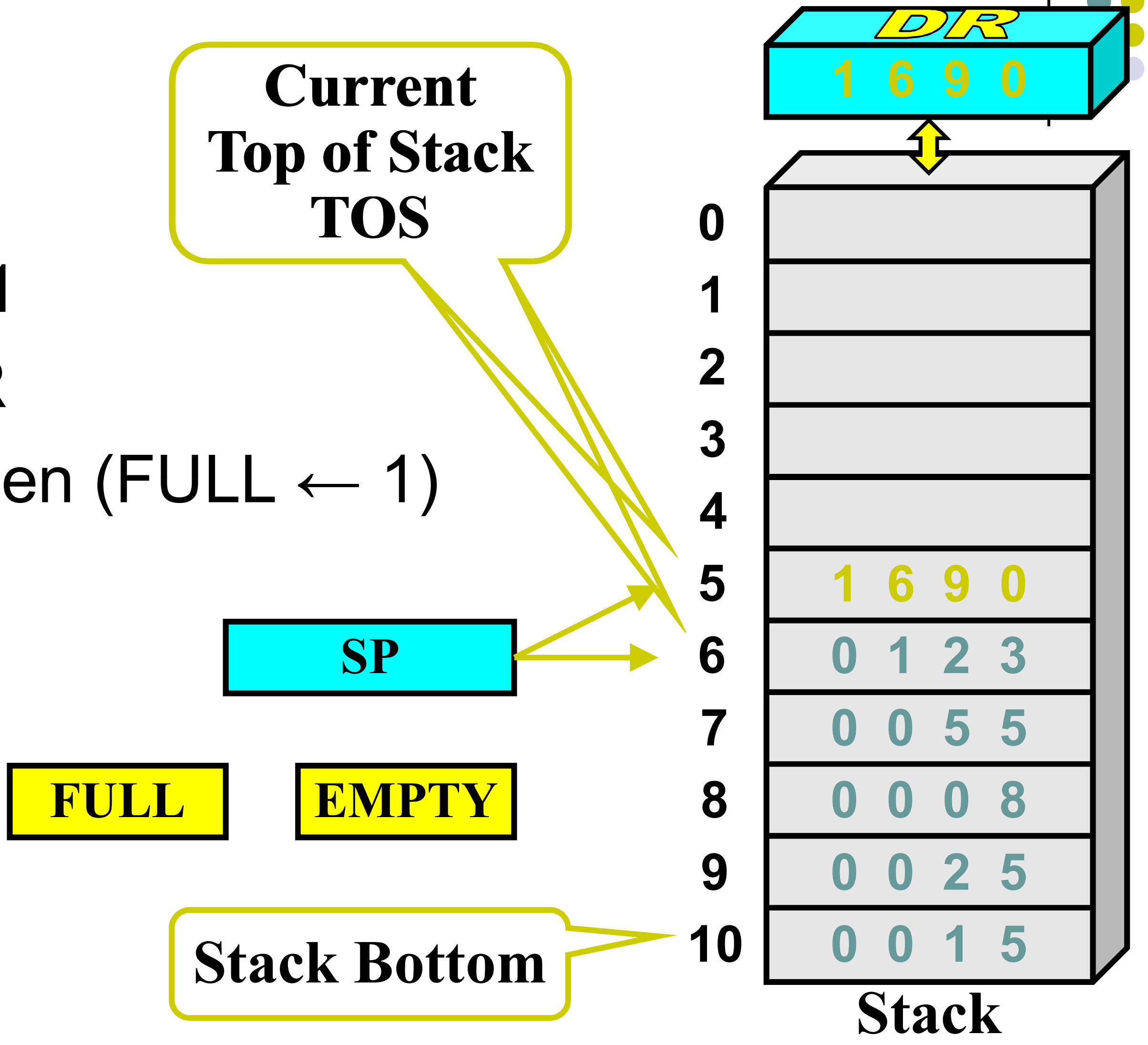
- PUSH

  SP ← SP – 1

  M[SP] ← DR

  If (SP = 0) then (FULL ← 1)

  EMPTY ← 0



Stack Organization

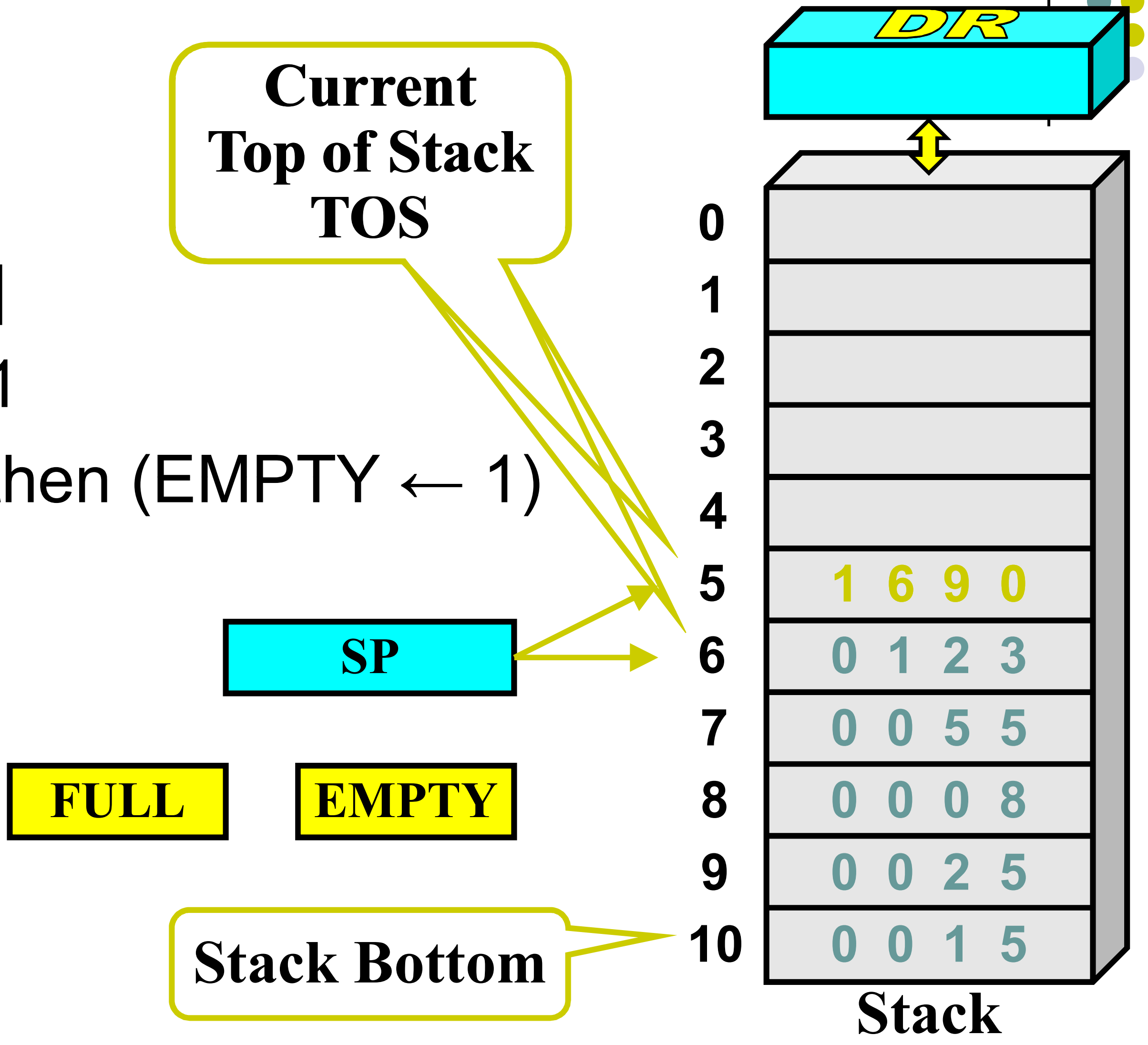
- POP

  DR ← M[SP]

  SP ← SP + 1

  If (SP = 11) then (EMPTY ← 1)

  FULL ← 0



Stack Organization

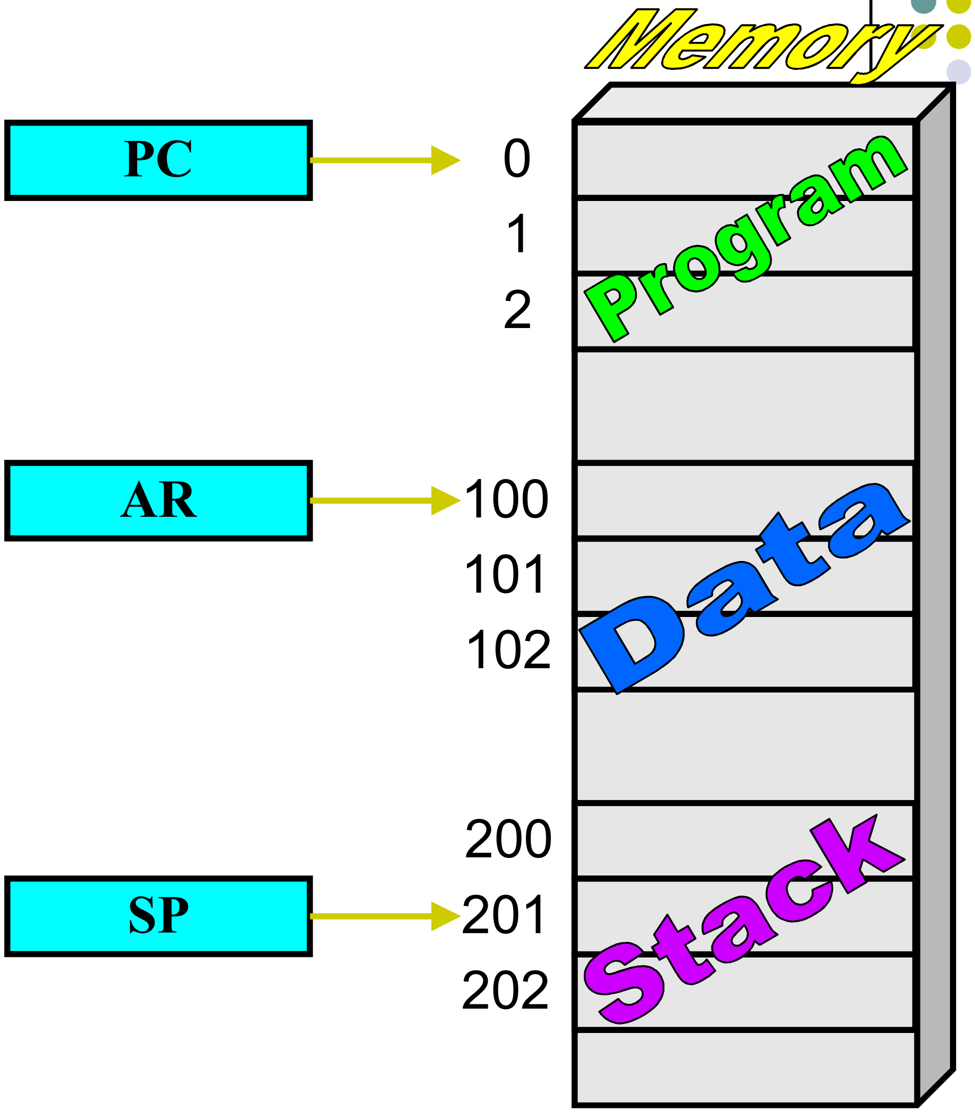
- Memory Stack
  - PUSH

    SP ← SP – 1

    M[SP] ← DR
  - POP

    DR ← M[SP]

    SP ← SP + 1



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) * +

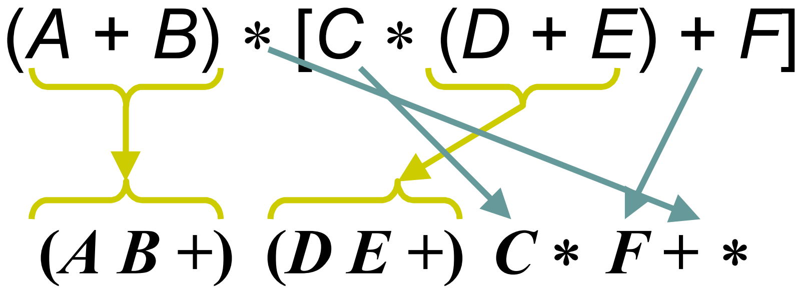
(8) (3) (3) * +

(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

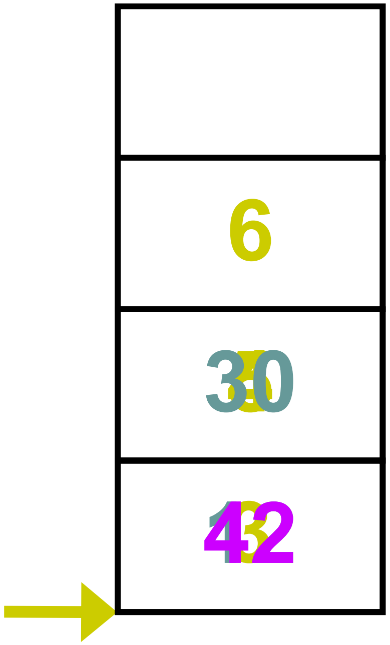
MULT

PUSH 5

PUSH 6

MULT

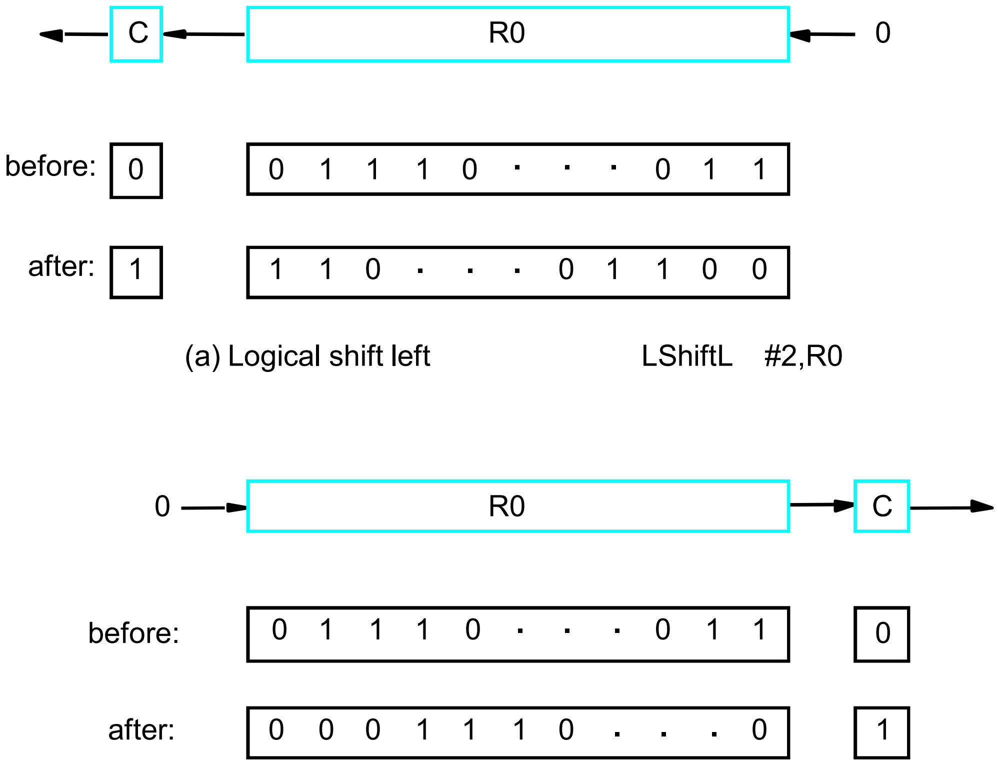
ADD



Additional Instructions

Logical Shifts

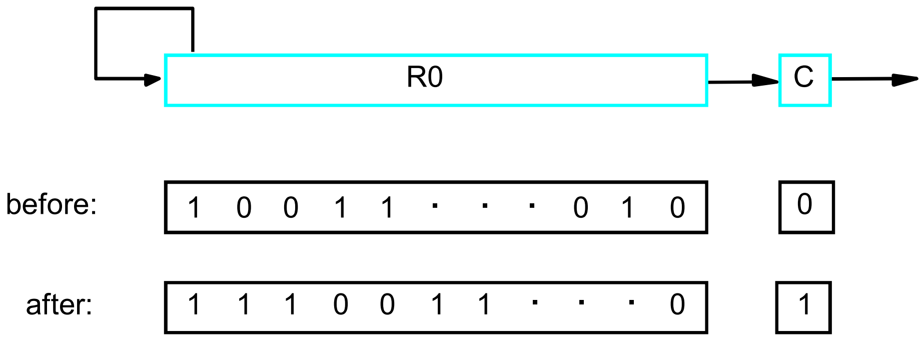
- Logical shift – shifting left (LShiftL) and shifting right (LShiftR)



(a) Logical shift left LShiftL #2,R0

(b) Logical shift right LShiftR #2,R0

Arithmetic Shifts



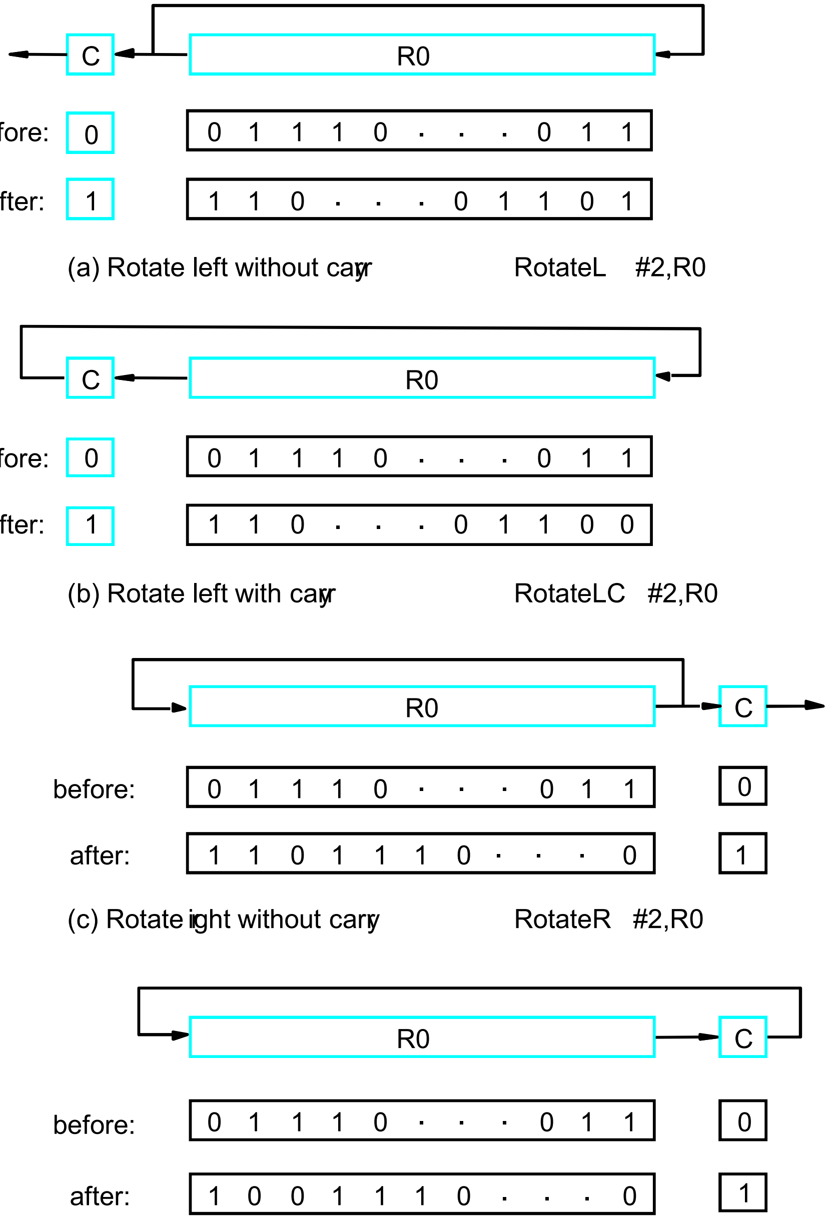
R0 C

before: 1 0 0 1 1 · · · 0 1 0 0

after: 1 1 1 0 0 1 1 · · · 0 1

(c) Arithmetic shift right AShiftR #2,R0

Rotate



(a) Rotate left without cayr RotateL #2,R0

(b) Rotate left with cayr RotateLC #2,R0

(c) Rotate ight without carry RotateR #2,R0

(d) Rotate ight with carry RotateRC #2,R0

Figure 2.32. Rotate instructions.

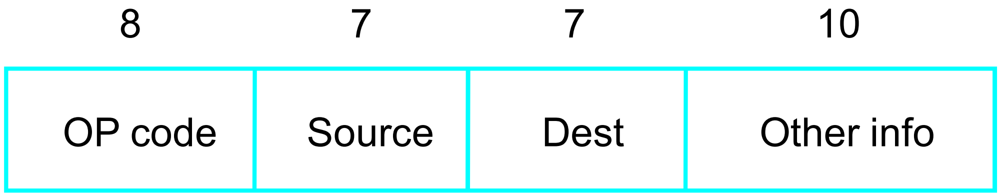
Multiplication and Division

- 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

Encoding of Machine Instructions

- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)
- In the previous section, an assumption was made that all instructions are one word in length.
- OP code: the type of operation to be performed 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

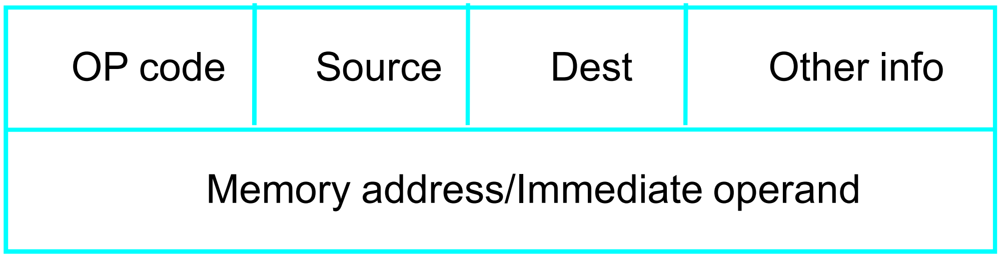


8	7	7	10
OP code	Source	Dest	Other info

(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, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (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

Chapter 7. Basic Processing Unit

Overview

- Instruction Set Processor (ISP)
- Central Processing Unit (CPU)
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executed by carrying out a sequence of more rudimentary operations.

Some Fundamental Concepts

Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

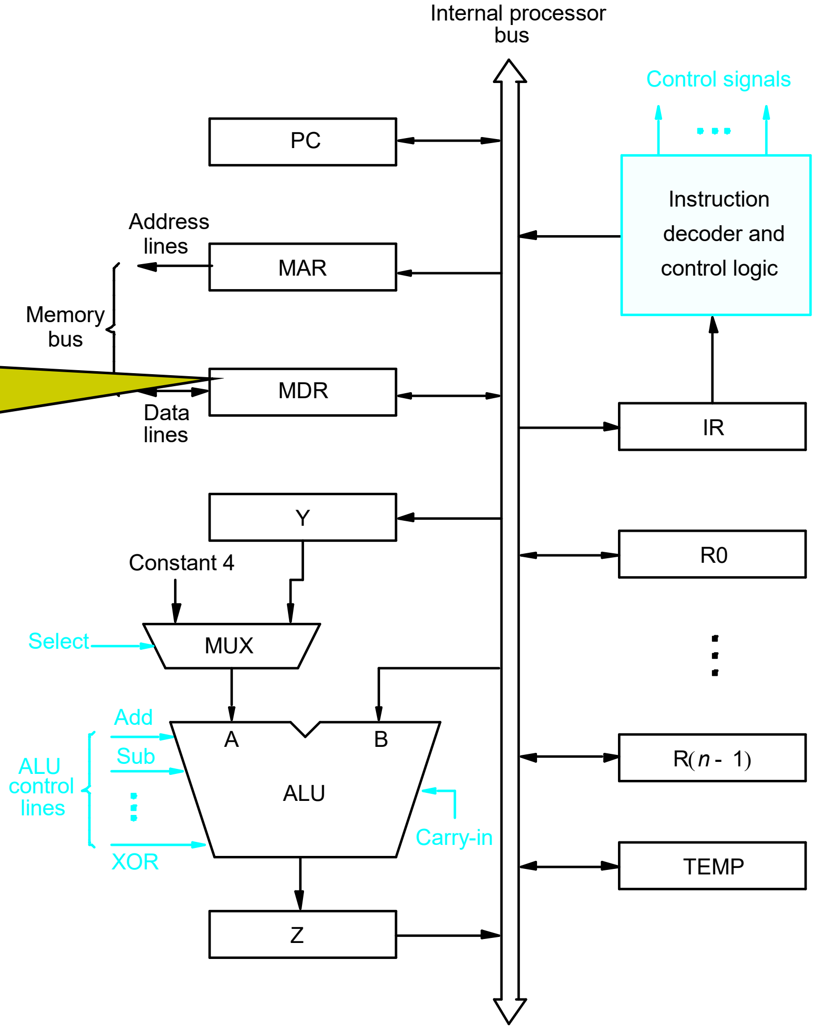
$$\text{IR} \leftarrow [[\text{PC}]]$$

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

$$\text{PC} \leftarrow [\text{PC}] + 4$$

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization



MDR HAS TWO INPUTS AND TWO OUTPUTS

Datapath

Textbook Page 413

Figure 7.1. Single-bus organization of the datapath inside a proc

Executing an Instruction

- Transfer a word of data from one processor register to another or to the ALU.
- Perform an arithmetic or a logic operation and store the result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store a word of data from a processor register into a given memory location.

Register Transfers

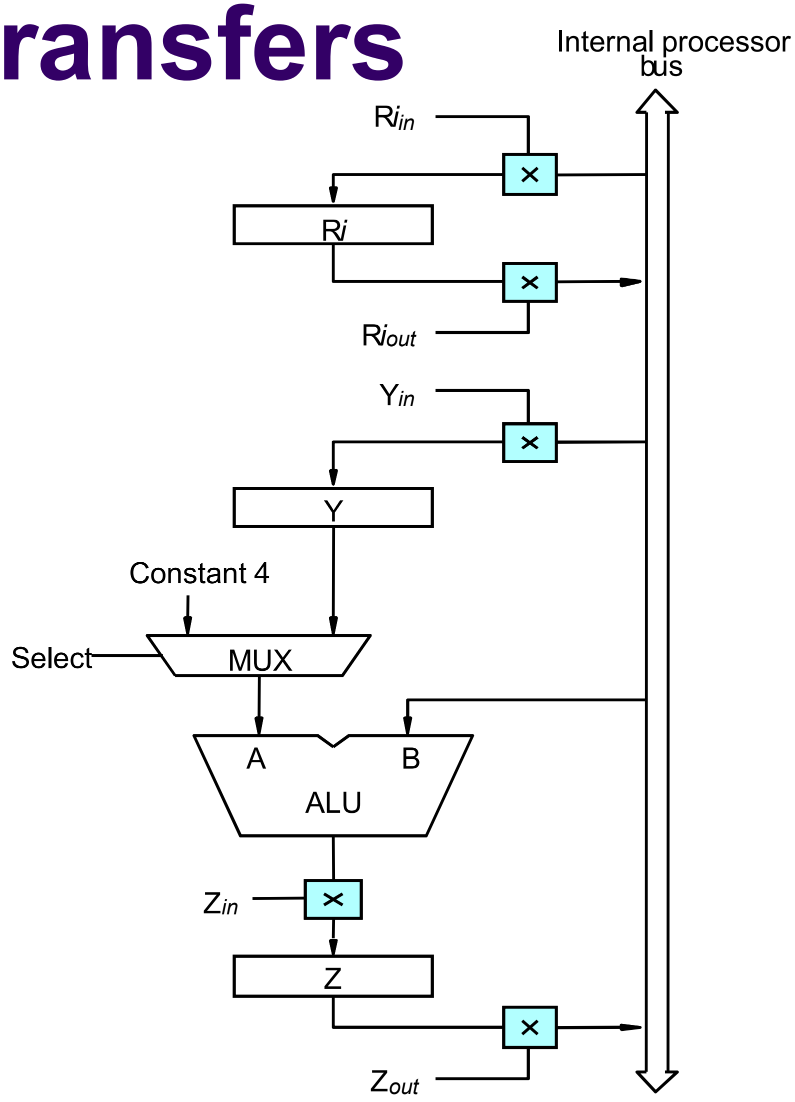


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Register Transfers

- All operations and data transfers are controlled by the processor clock.

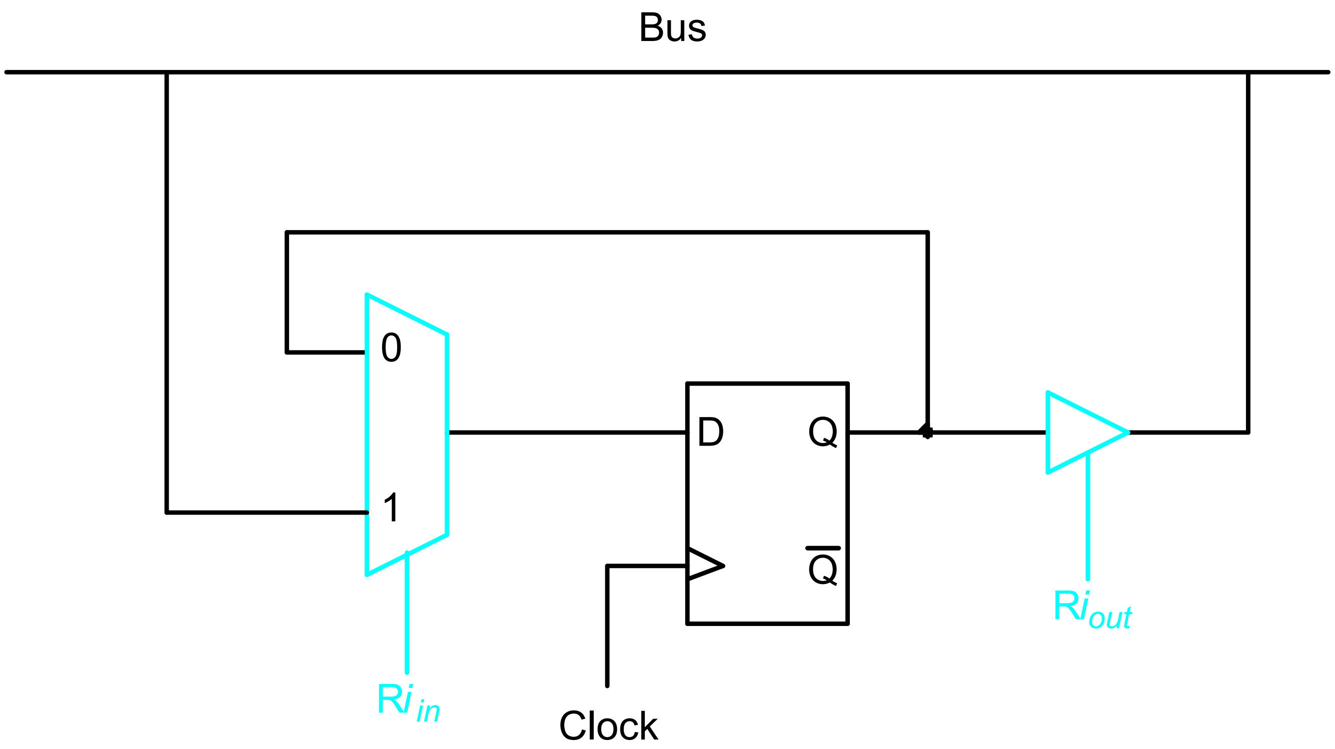


Figure 7.3. Input and output gating for one register bit.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
  1. R1out, Yin
  2. R2out, SelectY, Add, Zin
  3. Zout, R3in

Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.

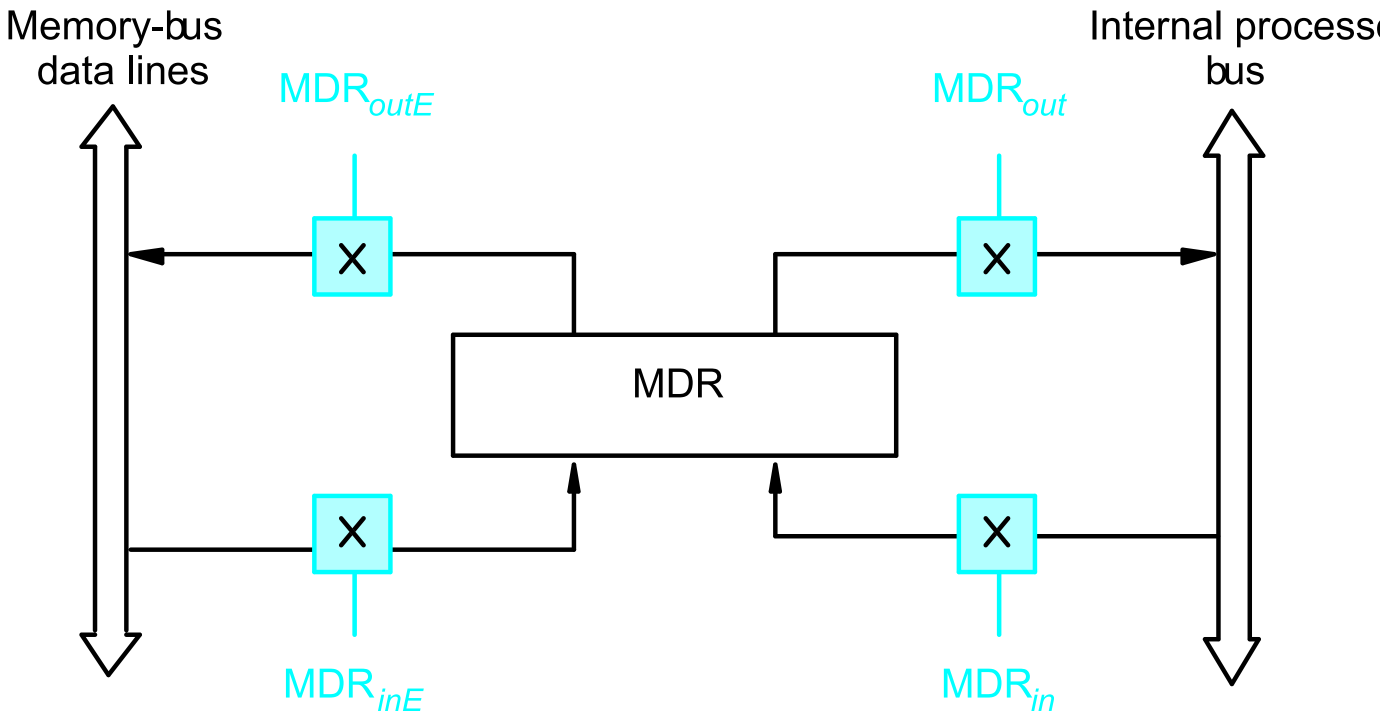


Figure 7.4. Connection and control signals for register MDR.

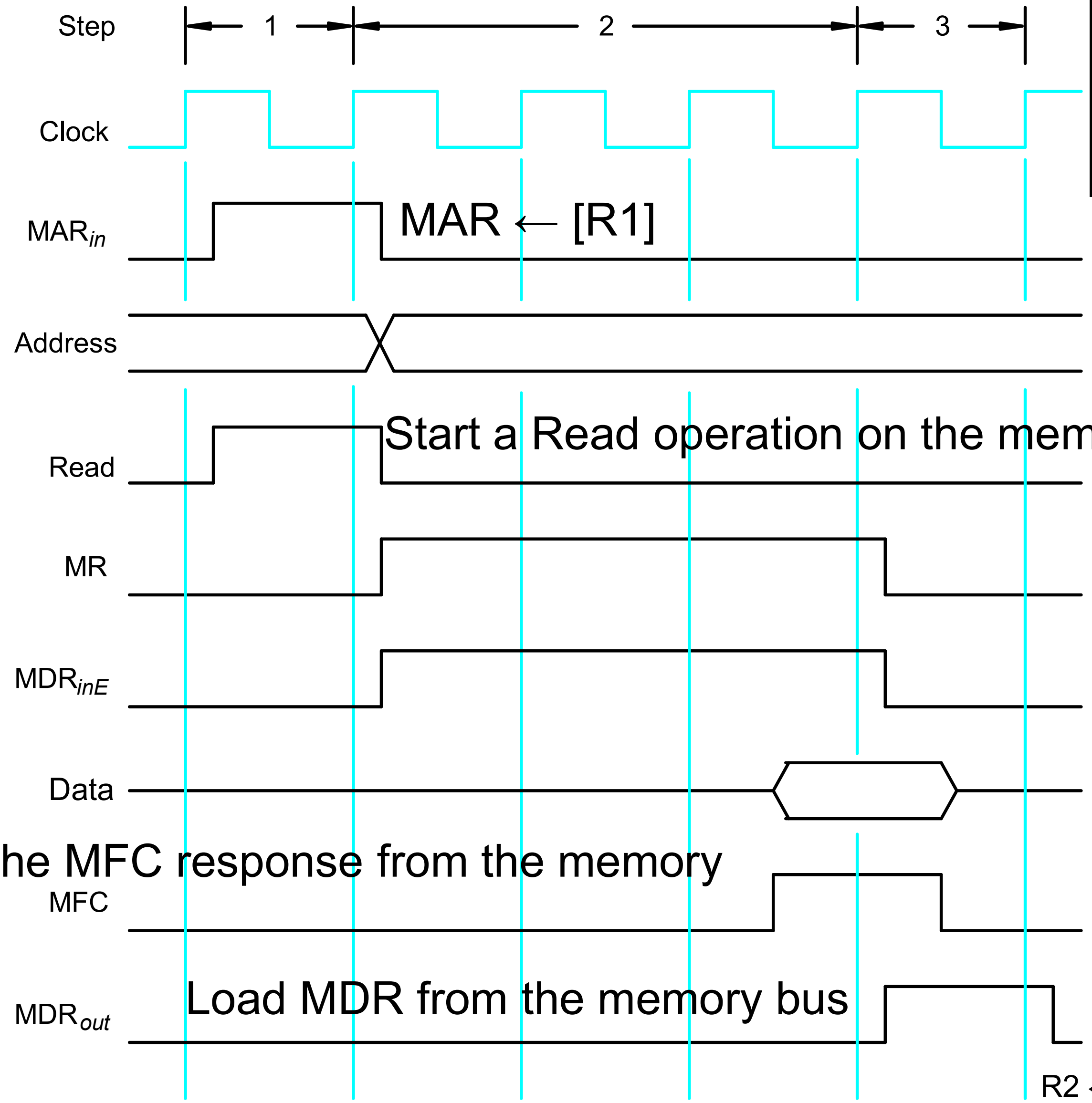
Fetching a Word from Memory

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- Move (R1), R2

  - MAR ← [R1]
  - Start a Read operation on the memory bus
  - Wait for the MFC response from the memory
  - Load MDR from the memory bus
  - R2 ← [MDR]

Timing

Assume MAR
is always available
on the address lines
of the memory bus.



MAR ← [R1]

Start a Read operation on the memory bus

Wait for the MFC response from the memory

Load MDR from the memory bus

R2 ← [MDR]

Figure 7.5. Timing of a memory Read operation.

Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

Architecture

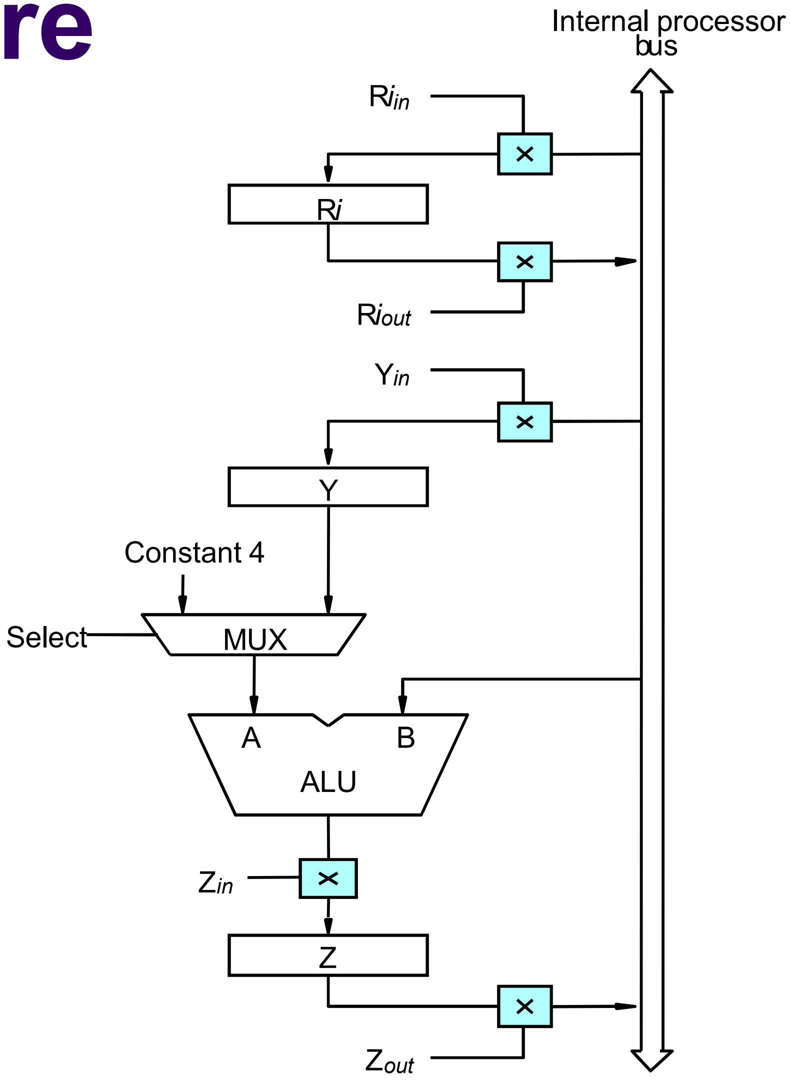


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Execution of a Complete Instruction

Add (R3), R1

Step	Action
1	PC$_{out}$, MAR$_{in}$, Read, Select4,Add, Z$_{in}$
2	Z$_{out}$, PC$_{in}$, Y$_{in}$, WMF C
3	MDR$_{out}$, IR$_{in}$
4	R3$_{out}$, MAR$_{in}$, Read
5	R1$_{out}$, Y$_{in}$, WMF C
6	MDR$_{out}$, SelectY, Add, Z$_{in}$
7	Z$_{out}$, R1$_{in}$, End

Figure 7.6. Control sequencefor execution of the instruction Add (R3),R1.

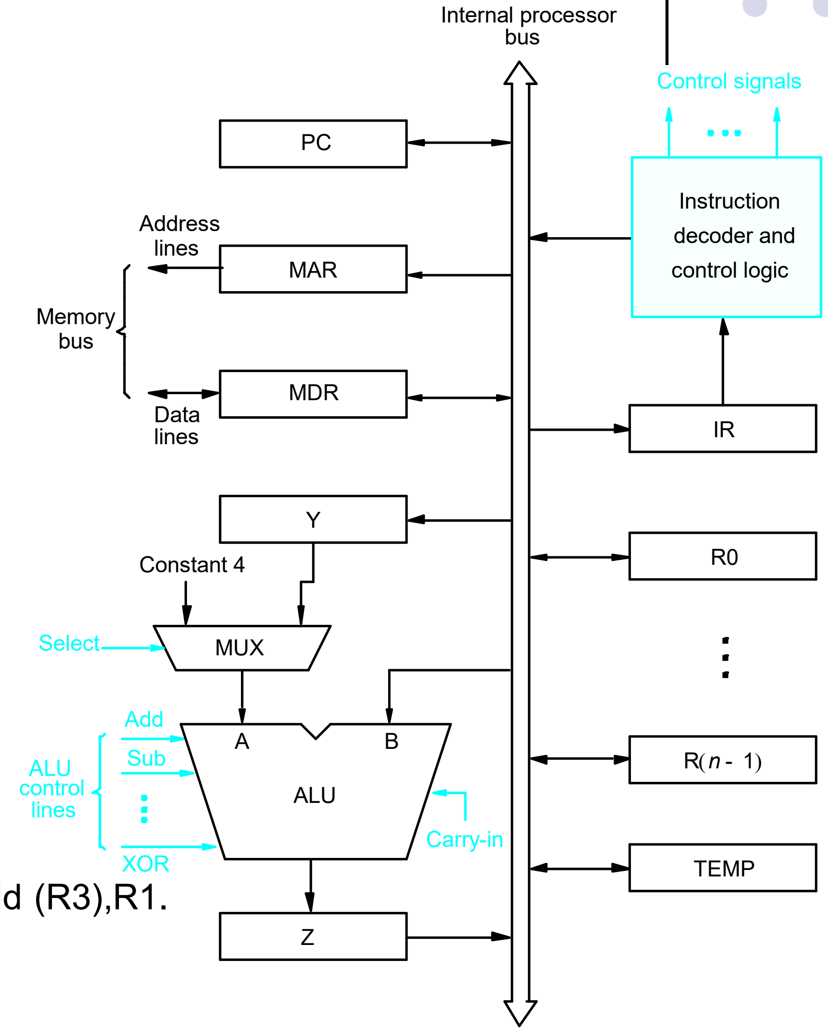


Figure 7.1. Single-bus organization of the datapath inside a proc

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

Execution of Branch Instructions

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4,Add, Z_{in}
2	Z_{out}, PC_{in} , Y_{in}, WMF C
3	MDR_{out} , IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in} , End

Figure 7.7. Control sequence for an unconditional branch instruction.

Multiple-Bus Organization

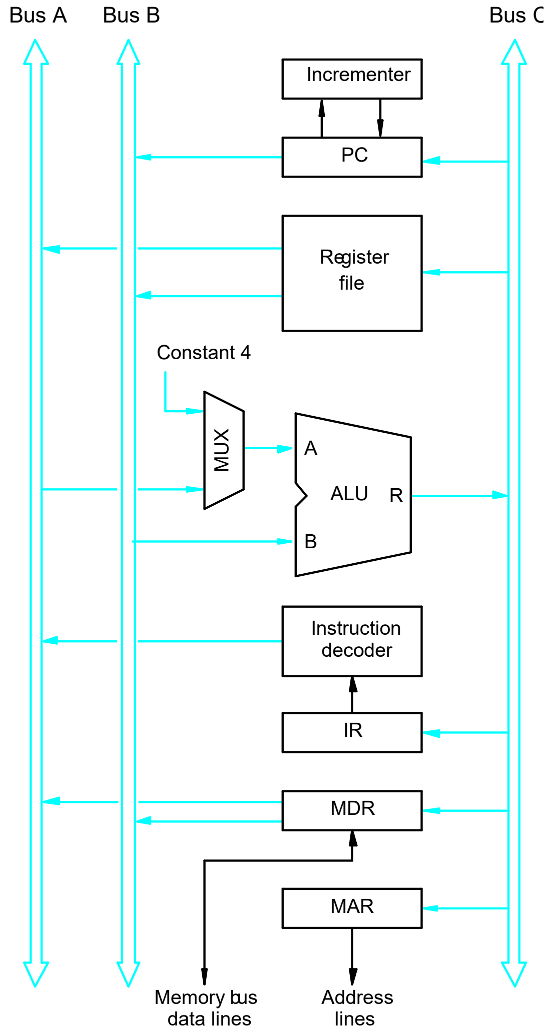


Figure 7.8. Three-bus oganization of the datapath.

Multiple-Bus Organization

- Add R4, R5, R6

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR^{in}
4	$R4^{outA}$, $R5^{outB}$, SelectA, Add, $R6^{in}$, End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6, for the three-bus organization in Figure 7.8.

Quiz

- What is the control sequence for execution of the instruction

  Add R1, R2

  including the instruction fetch phase? (Assume single bus architecture)

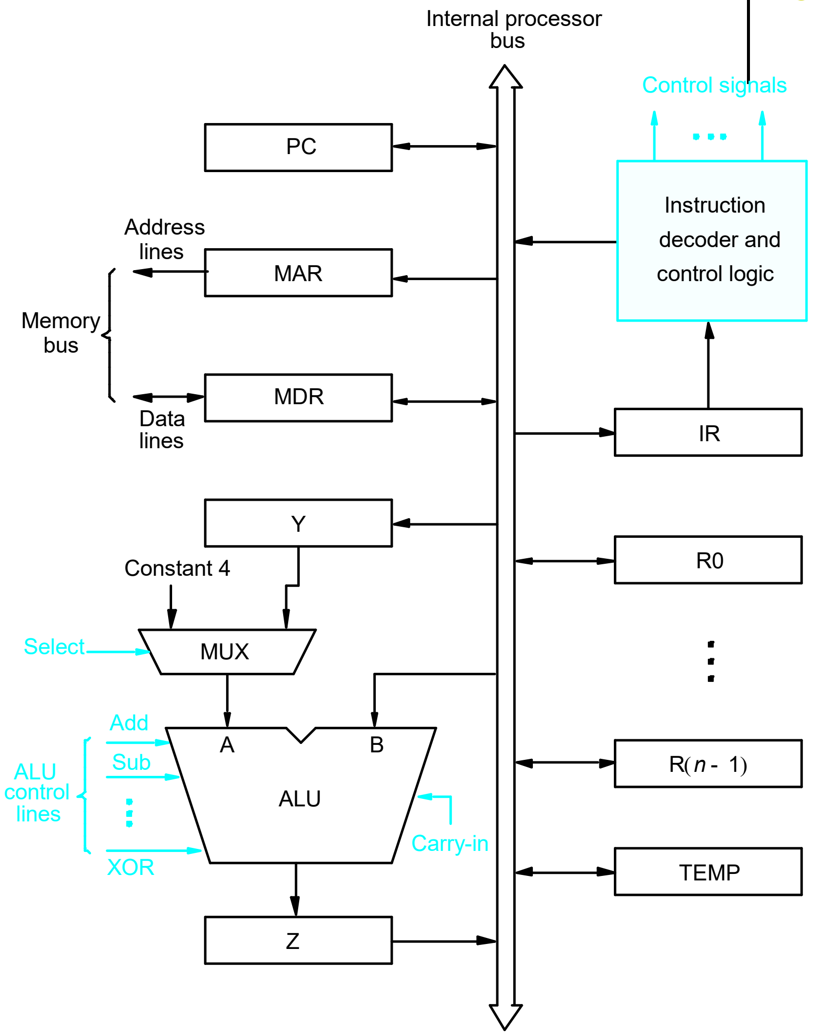


Figure 7.1. Single-bus organization of the datapath inside a proc

Hardwired Control

Overview

- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and microprogrammed control
- Hardwired system can operate at high speed; but with little flexibility.

Control Unit Organization

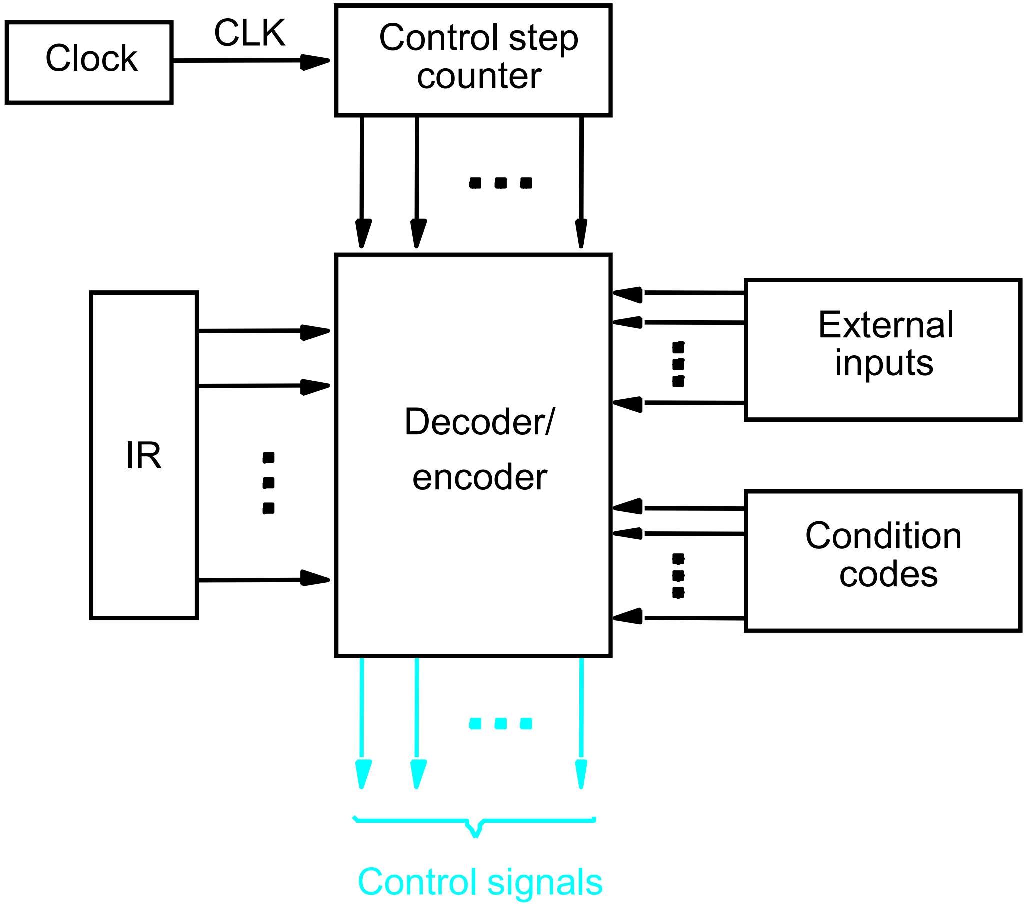


Figure 7.10. Control unit organization.

Detailed Block Description

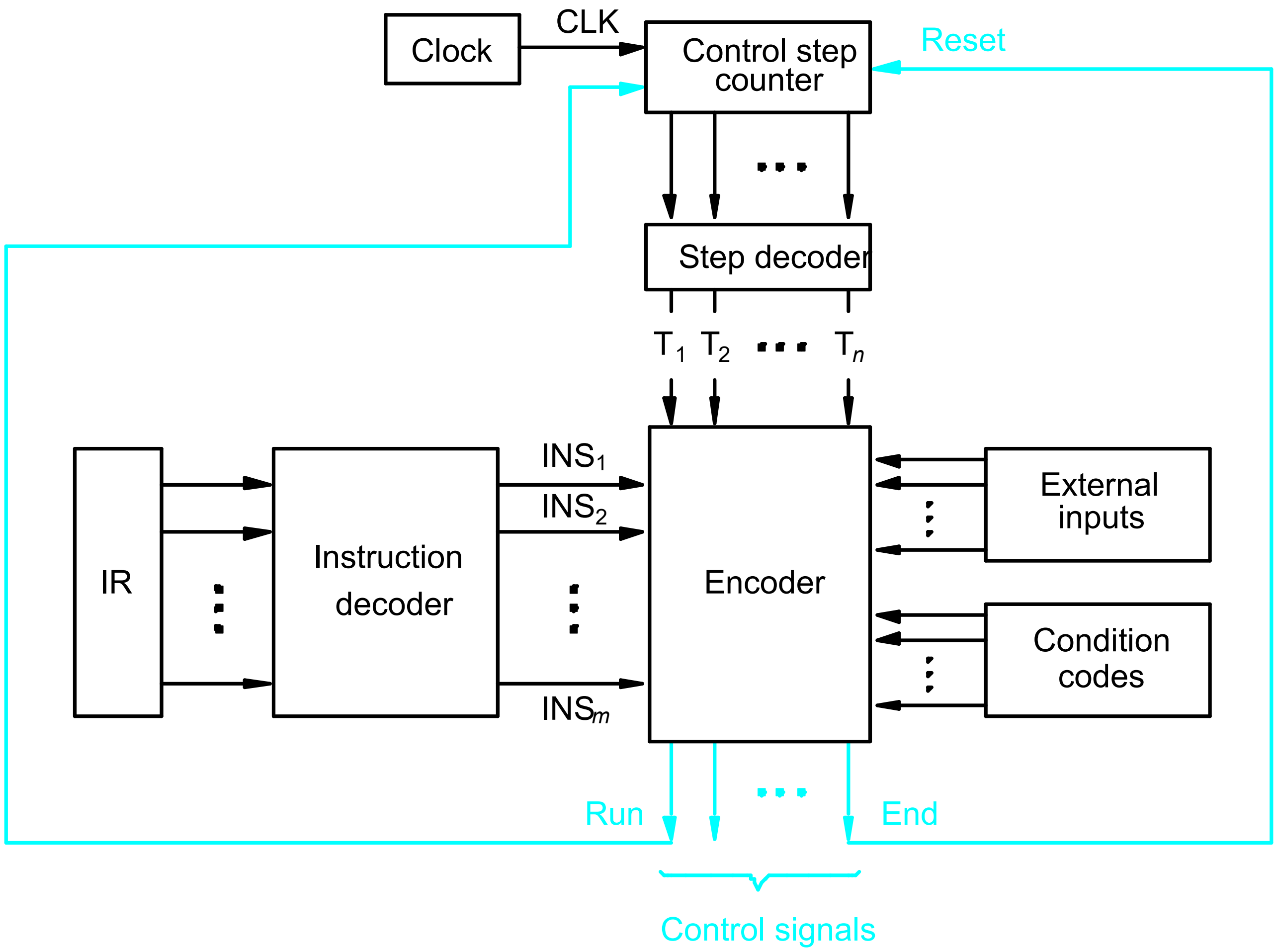


Figure 7.11. Separation of the decoding and encoding functio

Generating Z_{in}

- $Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \ldots$

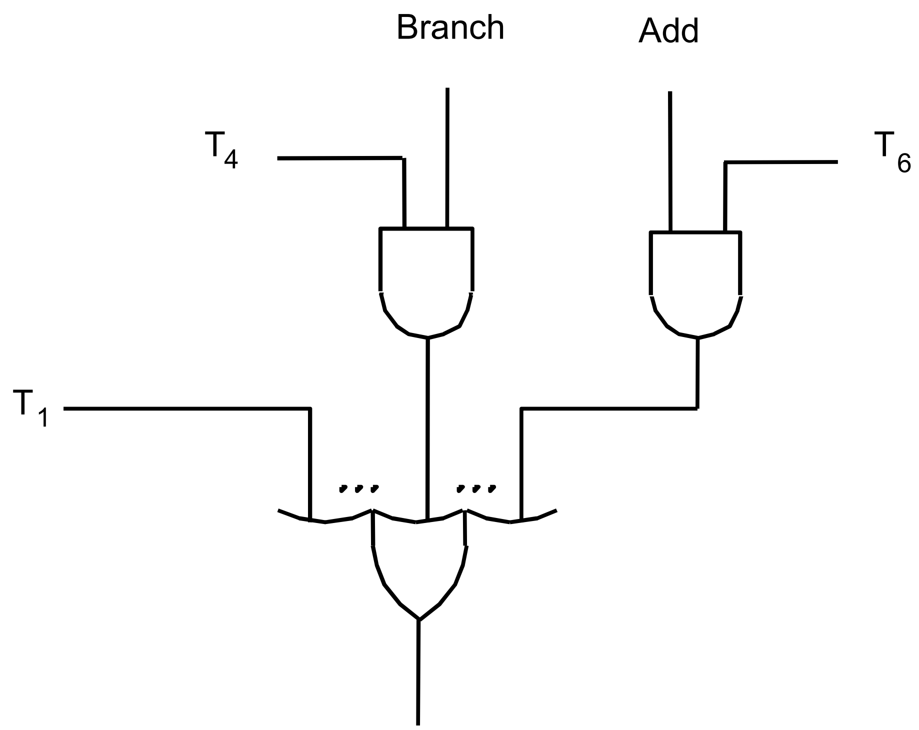


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- $\text{End} = T_7 \bullet \text{ADD} + T_5 \bullet \text{BR} + (T_5 \bullet \text{N} + T_4 \bullet \overline{\text{N}}) \bullet \text{BRN} + \ldots$

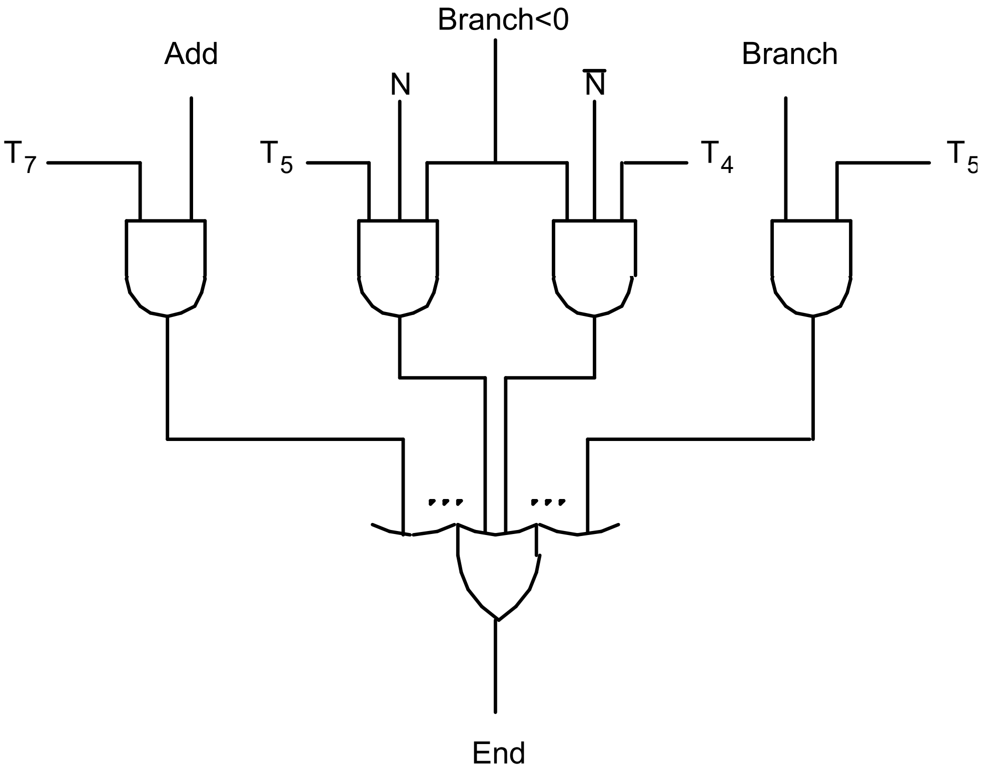


Figure 7.13. Generation of the End control signal.

A Complete Processor

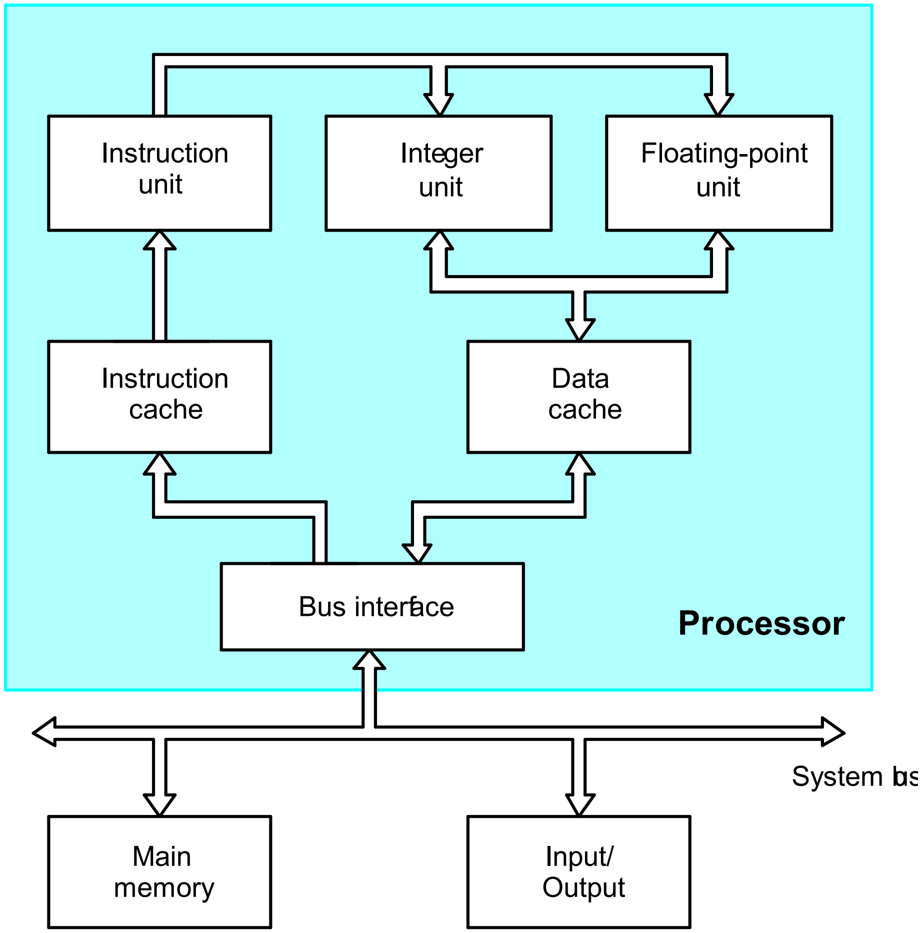


Figure 7.14. Block diagram of a complete processor

Microprogrammed Control

Overview

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	..
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

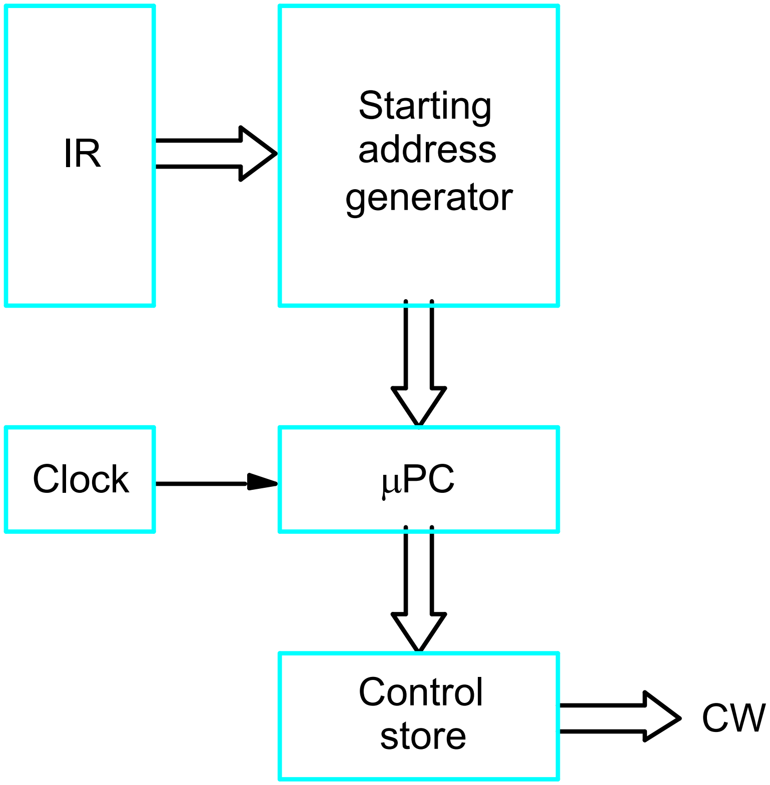
Overview

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4,Add, Z_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMF C
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMF C
6	MDR_{out} , SelectY, Add, Z_{in}
7	Z_{out} , $R1_{in}$, End

Figure 7.6. Control sequencefor execution of the instruction Add (R3),R1.

Overview

- Control store



One function cannot be carried out by this simple organization.

Figure 7.16. Basic organization of a microprogrammed control

Overview

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.

Address	Microinstruction
0	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
1	Z_{out} , PC_{in} , Y_{in} , WMFC
2	MDR_{out} , IR_{in}
3	Branch to starting address of appropriate microroutine
. . . .	
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out} , SelectY, Add, Z_{in}
27	Z_{out} , PC_{in} , End

Figure 7.17. Microroutine for the instruction Branch<0.

Overview

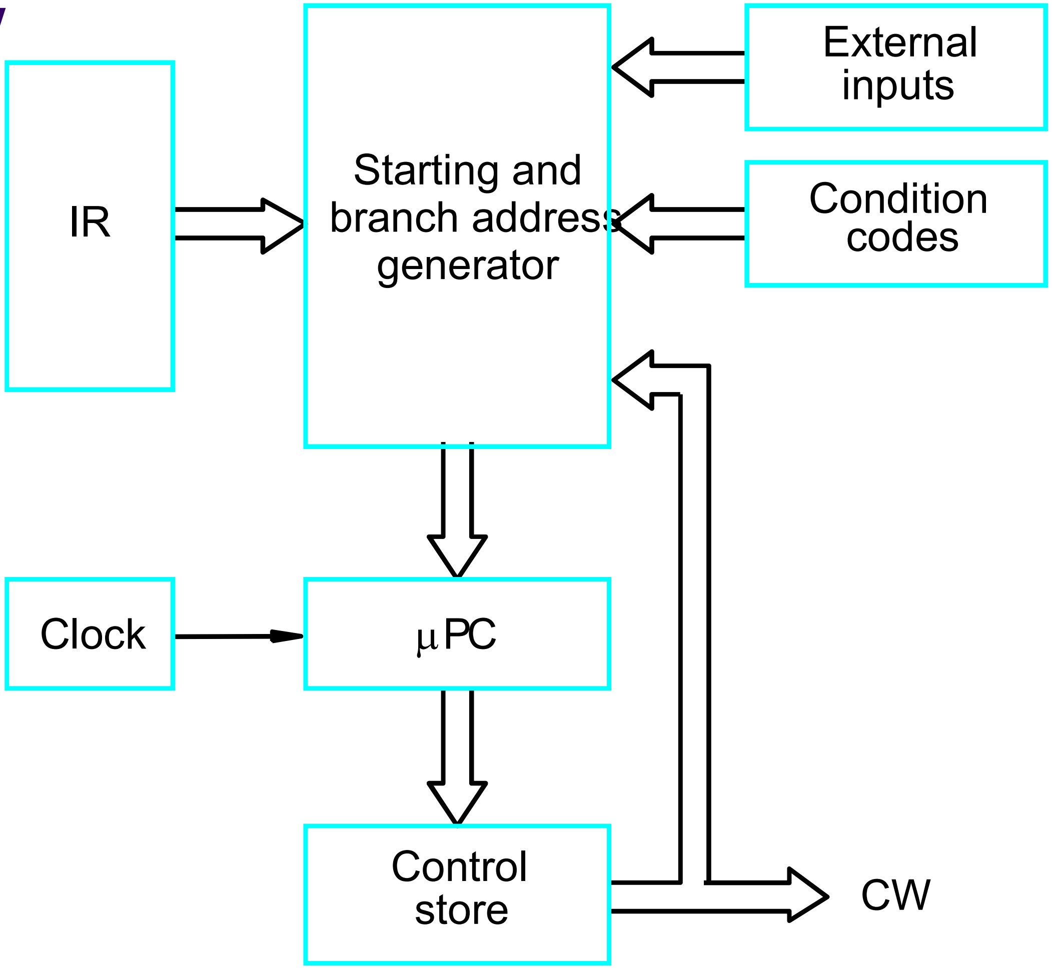
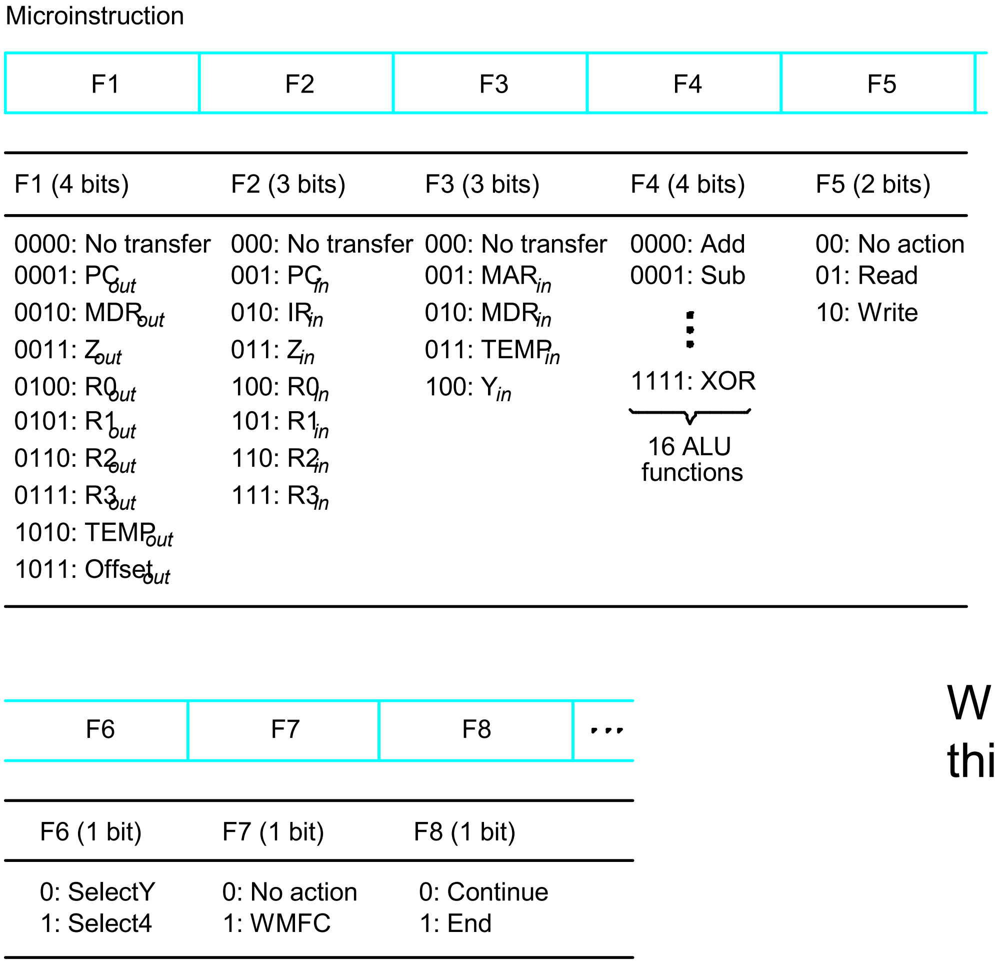


Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Partial Format for the Microinstructions



Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functions	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

F6	F7	F8	...

F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0: SelectY	0: No action	0: Continue
1: Select4	1: WMFC	1: End

Figure 7.19. An example of a partial format for field-encoded microinstruction

What is the price paid for this scheme?

Further Improvement

- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

Microprogram Sequencing

- If all microprograms require only straightforward sequential execution of microinstructions except for branches, letting a μPC governs the sequencing would be efficient.
- However, two disadvantages:
  - Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.
  - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).

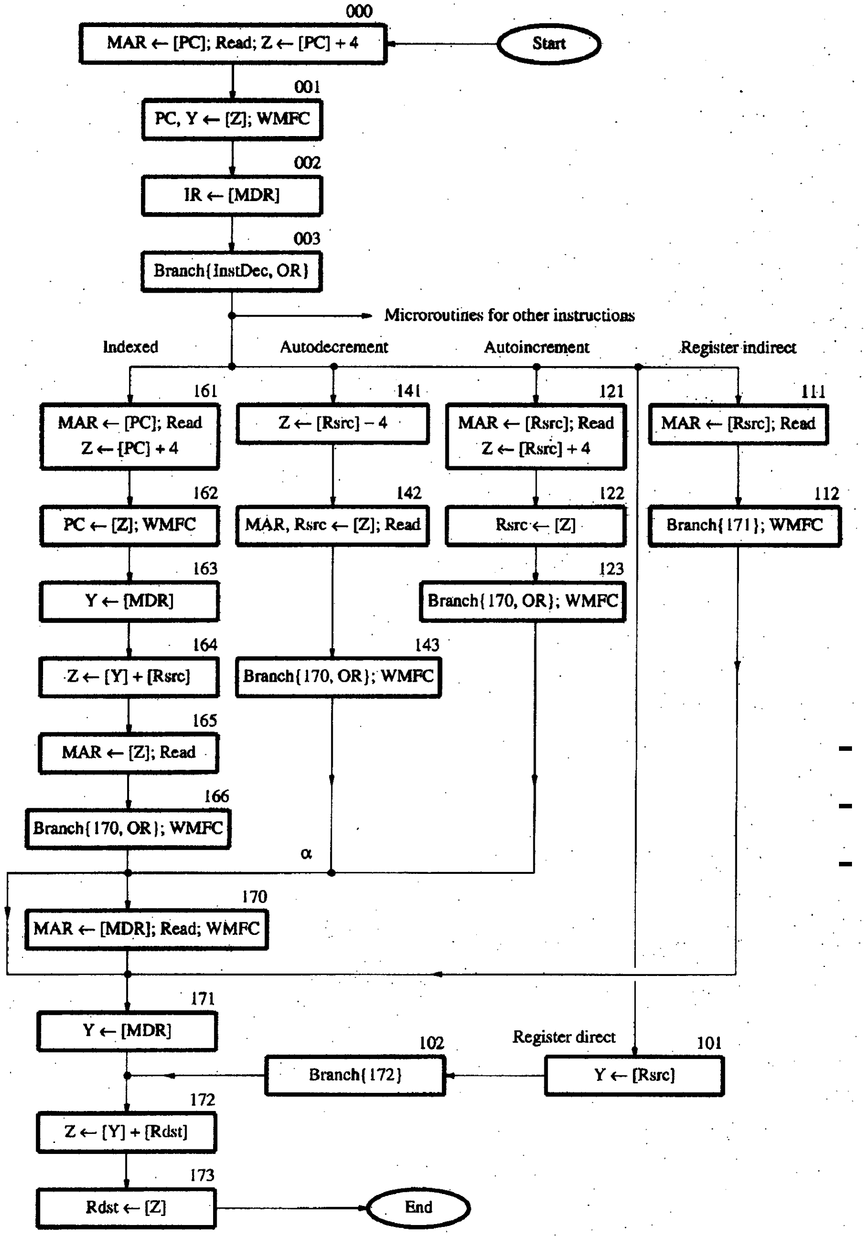
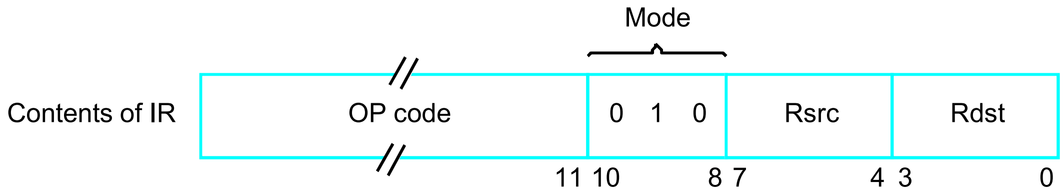


Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.

- Bit-ORing
- Wide-Branch Addressing
- WMFC



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	µBranch {µPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	µBranch {µPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21. Microinstruction for Add (Rsrc)+,Rdst.

*Note:*Microinstruction at location 170 is not executed for this addressing mode.

Microinstructions with Next-Address Field

- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

Microinstructions with Next-Address Field

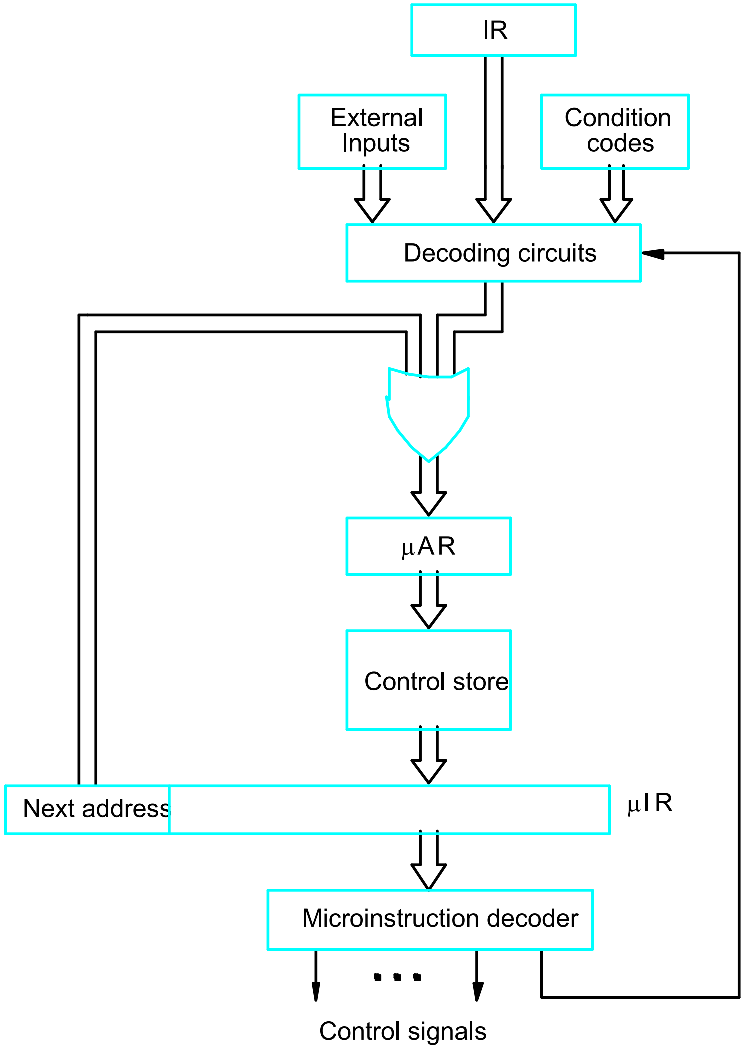


Figure 7.22. Microinstruction-sequencing organization.

Microinstruction

F0	F1	F2	F3

F0 (8 bits)	F1 (3 bits)	F2 (3 bits)	F3 (3 bits)
Address of next	000: No transfer	000: No transfer	000: No transfer
microinstruction	001: PC$_{out}$	001: PC$_{in}$	001: MAR$_{in}$
	010: MDR$_{out}$	010: IR$_{in}$	010: MDR$_{in}$
	011: Z$_{out}$	011: Z$_{in}$	011: TEMP$_{in}$
	100: Rsrc$_{out}$	100: Rsrc$_{in}$	100: Y$_{in}$
	101: Rdst$_{out}$	101: Rdst$_{in}$	
	110: TEMP$_{out}$		

F4	F5	F6	F7

F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)
0000: Add	00: No action	0: SelectY	0: No action
0001: Sub	01: Read	1: Select4	1: WMFC
⋮	10: Write		
1111: XOR			

F8	F9	F10

F8 (1 bit)	F9 (1 bit)	F10 (1 bit)
0: NextAdrs	0: No action	0: No action
1: InstDec	1: OR$_{mode}$	1: OR$_{indsrc}$

Figure 7.23. Format for microinstructions in the example of Section 7

Implementation of the Microroutine

Octal address	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
0 0 0	0 0 0 0 0 0 0 1	0 0 1	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
0 0 1	0 0 0 0 0 0 1 0	0 1 1	0 0 1	1 0 0	0 0 0 0	0 0	0	1	0	0	0
0 0 2	0 0 0 0 0 0 1 1	0 1 0	0 1 0	0 0 0	0 0 0 0	0 0	0	0	0	0	0
0 0 3	0 0 0 0 0 0 0 0	0 0 0	0 0 0	0 0 0	0 0 0 0	0 0	0	0	1	1	0
1 2 1	0 1 0 1 0 0 1 0	1 0 0	0 1 1	0 0 1	0 0 0 0	0 1	1	0	0	0	0
1 2 2	0 1 1 1 1 0 0 0	0 1 1	1 0 0	0 0 0	0 0 0 0	0 0	0	1	0	0	1
1 7 0	0 1 1 1 1 0 0 1	0 1 0	0 0 0	0 0 1	0 0 0 0	0 1	0	1	0	0	0
1 7 1	0 1 1 1 1 0 1 0	0 1 0	0 0 0	1 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 2	0 1 1 1 1 0 1 1	1 0 1	0 1 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0
1 7 3	0 0 0 0 0 0 0 0	0 1 1	1 0 1	0 0 0	0 0 0 0	0 0	0	0	0	0	0

Figure 7.24. Implementation of the microroutine of Figure 7.21 usi
next-microinstruction address field. (See Figure 7.23 for encoded sign

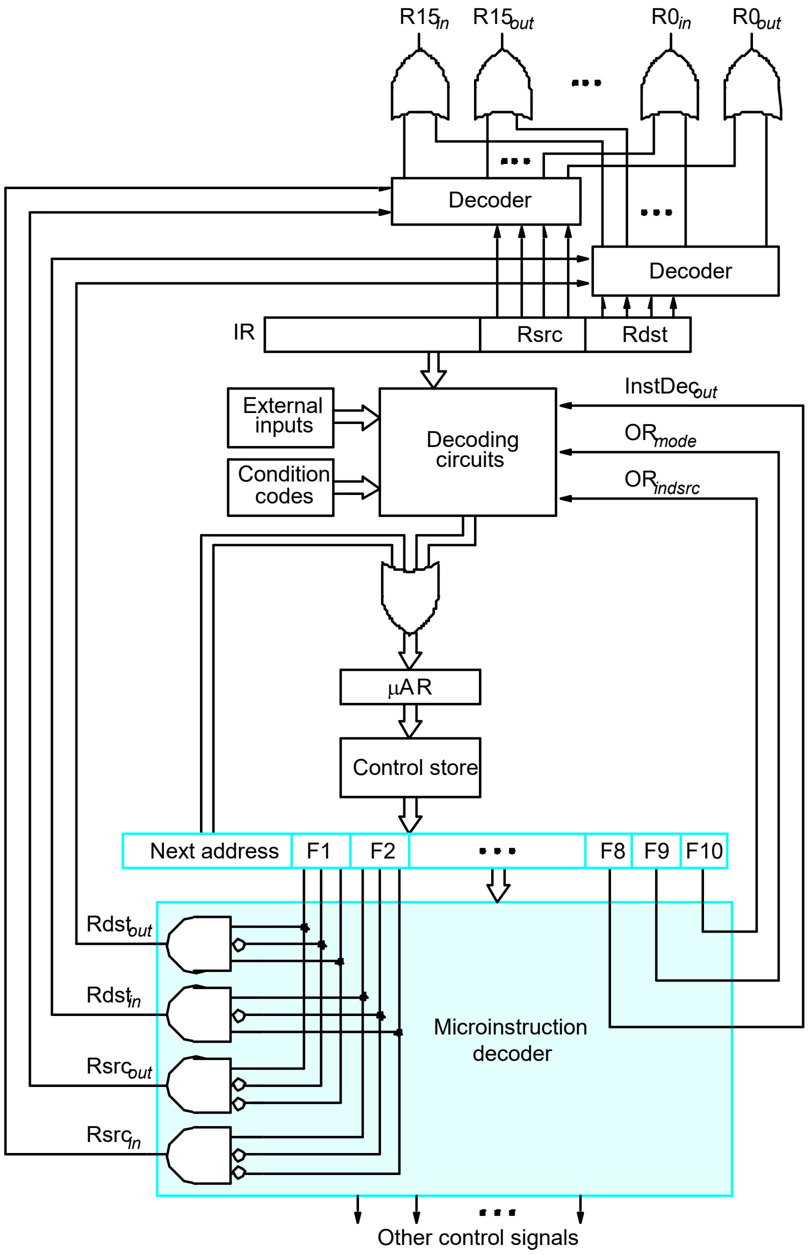


Figure 7.25. Some details of the control-signal-generating circuitry.

bit-ORing

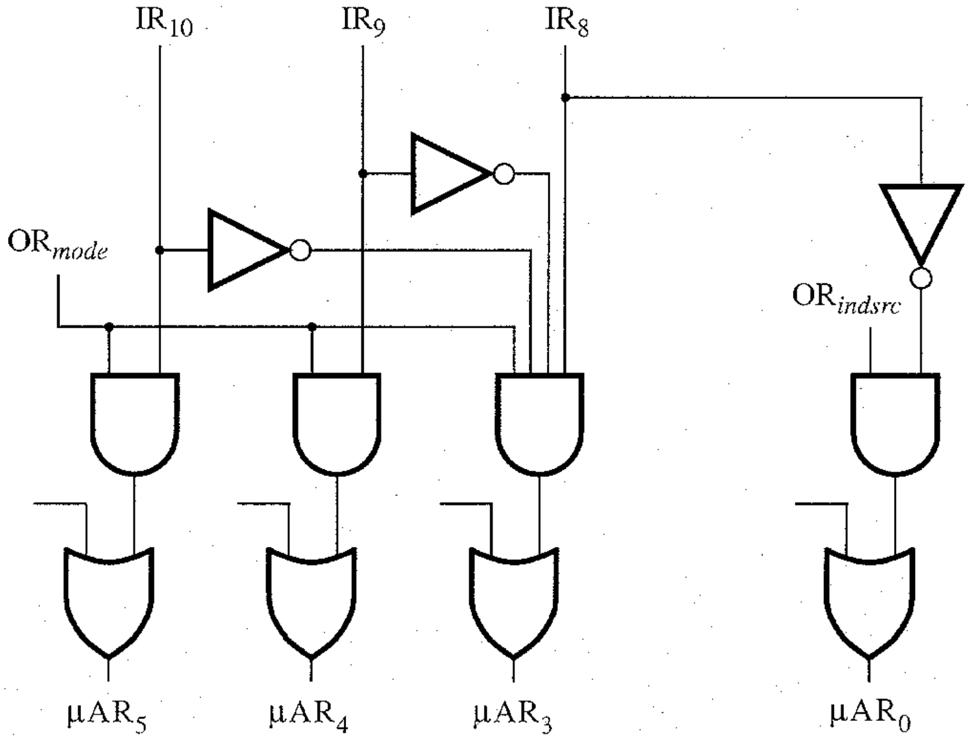


Figure 7.26. Control circuitry for bit-ORing

(part of the decoding circuits in Figure 7.25).

Further Discussions

- Prefetching
- Emulation