

Signed binary values

- Values that may be positive or negative are stored in *twos-complement* format
- The leftmost bit is reserved for the sign: 1 = negative, 0 = non-negative
- Negative values are stored in such a way that binary addition yields a correct result

$$\begin{array}{r} 0101_2 = 5 \\ + 1100_2 = 4 \\ \hline 0001_2 = 1 \end{array}$$

- To convert to 2's complement, flip bits add 1

David Keil 9/03 7

Integer overflow

- If a computation produces a value that exceeds the capacity of its storage, the effect is *overflow*, producing an incorrect computed value
- *Example (4 bits):*
1000 + 1000 = 10000 (stored as 0000)
- To avoid overflow, use numeric data types of sufficient capacity

David Keil 9/03 8

Floating-point representation

- Concept is same as that used in exponential (scientific) notation; e.g., $3200 = 3.2 \times 10^3$
- Values are stored using a *binary point*
- For precision and flexibility, 3 fields are used: sign bit; normalized binary fraction (NBF); exponent
- The NBF results from moving the binary point just to the left of the leftmost 1 bit
- Multiplying NBF by 2^{exponent} compensates for shifting the binary point
- Significant bits may be lost in floating-point representation




David Keil 9/03 9

Hexadecimal notation

- 16 digits: '0' to '9' and 'a'..'f'
- Compactness
- Easy conversion with binary
- Each digit is 4 bits
- Memory addresses are normally expressed in hex
- Examples:
 $0a_{16} = 10_{10}$
 $12_{16} = 18_{10}$




David Keil 9/03 10

Logic gates

- Used to manipulate binary data
- 1 or 2 bit input, 1-bit output
- Specified using truth tables
- NOT (negation) 
- AND (conjunction) 
- OR (disjunction) 
- Used as components of combinational circuits:
NAND, NOR, XOR, adders, etc.

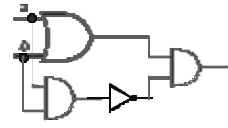
David Keil 9/03 11

Gates and truth tables

Gate	Schematic	Truth table															
NOT		<table border="1"> <thead> <tr> <th>a</th> <th>not a</th> </tr> </thead> <tbody> <tr><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td></tr> </tbody> </table>	a	not a	1	0	0	1									
a	not a																
1	0																
0	1																
OR		<table border="1"> <thead> <tr> <th>a</th> <th>b</th> <th>a or b</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	a	b	a or b	0	0	0	0	1	1	1	0	1	1	1	1
a	b	a or b															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
AND		<table border="1"> <thead> <tr> <th>a</th> <th>b</th> <th>a and b</th> </tr> </thead> <tbody> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </tbody> </table>	a	b	a and b	0	0	0	0	1	0	1	0	0	1	1	1
a	b	a and b															
0	0	0															
0	1	0															
1	0	0															
1	1	1															

David Keil 9/03 12

XOR and NOR circuits



<i>a</i>	<i>b</i>	<i>a xor b</i>
0	0	0
0	1	1
1	0	1
1	1	0

- *Problem:* Can you build a NOR gate from NOT, AND, and OR gates?

<i>a</i>	<i>b</i>	<i>a nor b</i>
0	0	1
0	1	0
1	0	0
1	1	0

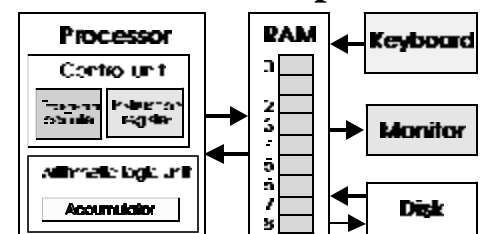
David Keil 9/03 13

Bitwise operations

<i>Operation</i>	<i>C/C++ operator</i>	<i>Example</i>
Complement	<code>~</code>	<code>~ 10000000₂ = 01111111₂</code>
OR	<code> </code>	<code>1100₂ 1001₂ = 1101₂</code>
AND	<code>&</code>	<code>1100₂ & 1001₂ = 1000₂</code>
Left shift	<code><<</code>	<code>1101₂ << 1 = 11010₂</code>
Right shift	<code>>></code>	<code>11000₂ >> 2 = 110₂</code>
XOR	<code>^</code>	<code>1001₂ ^ 1010₂ = 0011₂</code>

David Keil 9/03 14

A model computer



- RAM (random-access memory) contains programs and data

David Keil 9/03 15

An assembler program

```

// echo.asm; echoes input
Data      input a
statement print a
with label stop
          data 0
          a
    
```

← Input statement
← Output statement
← Termination

Sample input/output:
 [Input:] 26
 [Output:] 26

David Keil 9/03 16

The fetch-execute cycle

PC ← 0

Repeat

- Copy instruction from MEM(PC) to IR
- Increment PC
- Execute operation in IR

until operation is STOP

- PC = Program Counter register
- IR = Instruction Register
- ← = gets

David Keil 9/03 17

The language of the model processor

- Device I/O: *input, print, sinput, sprint*
- Copy data between ACC and RAM: *load, store*
- Calculate in ACC: *add, sub*
- Control execution: *stop, jump, jump0, jump-*
- Allocate and label data space: *data, sdata*

David Keil 9/03 18

The assembler

- An *assembler* program converts assembly-language code to machine language

Assembler-language program (mnemonics and labels)

Assembler

Machine-language program (binary)

David Keil 9/03 19

A program to add

```

// ADD.ASM: Displays sum of 2 inputs
input  input1
input  input2
load   input1
add    input2
store  sum
print  sum
stop
input1 data 0
input2 data 0
sum    data 0
    
```

$sum \leftarrow input1 + input2$

- input* copies data from keyboard to RAM
- load* and *add* alter contents of ACC
- store* copies from ACC to memory

David Keil 9/03 20

Mnemonics, operands, and labels

- In an assembler program, a mnemonic is an easily remembered abbreviation for a machine-language operation
- An operand specifies the address on which an instruction is to operate
- A label is a name for an address
- In the model-processor assembler, column 1 is for labels, column 2 for mnemonics, column 3 for operand labels

David Keil 9/03 21

A program to find absolute value

```

// abs.asm
input  n
load  n
jump- negate
jump  end
negate sub  n
sub   n
end   store abs
print abs
stop
n    data 0
abs  data 0
    
```

$input\ n$
if $n < 0$
 display $(-n)$
else
 display n

- Mnemonic *jump-* triggers jump to operand address (label *negate*) if negative value in ACC

David Keil 9/03 22

Program to count down from 10

```

// count.asm
repeat load  count
print  count
jump0  exit
sub    one
store  count
jump  repeat
exit   stop
count data 10
one    data 1
    
```

$count \leftarrow 10$
repeat
 display $count$
 decrement $count$
until $count = 0$

- Mnemonic *jump0* triggers jump to operand address (label *exit* here) if ACC contains the value 0
- Mnemonic *jump* triggers unconditional jump to operand address (label *repeat* here)

David Keil 9/03 23

Machine and assembler languages

- Operations are simple
- Each operation has a mnemonic abbreviation in assembler language
- Each processor has its own machine language and assembler language
- Most operations act on a data item in RAM
- Programmer may give names to addresses, using labels
- Hard to read

David Keil 9/03 24