CMP 1203

Lecture 4

Excess-M representation of Signed numbers

- Define a set of values, split it into ranges and use one range to represent positive numbers, another for negative numbers.
- Maps decimal numbers
- Bias (M) is selected such that the range is split equally
- If using **n** bits to represent numbers, choose bias of $2^{n-1} - 1$. E.g. For 4 bits, bias is $2^{4-1} - 1 = 7$
- Then excess-M representation simply adds M to that integer.
- Examples:

1. $0_{10} = 0 + 7 = 7_{10} = 0111_2$
2. $-7 = -7 + 7 = 0_{10} = 0000_2$
3. $1111_2 = 15_{10}$, 15-7 =8. So $1111_2 = +8_{10}$

Recap

Integer		Binary strings representing the signed integer			
Decimal	Binary	Signed Magnitude	1's complement	2's complement	Excess-127
2	00000010	00000010	00000010	00000010	10000001
-2	00000010	10000010	11111101	11111110	01111101
100	01100100	01100100	01100100	01100100	11100011
-100	01100100	11100100	10011011	10011100	00011011

Unsigned Vs. Signed Numbers

- Unsigned numbers are used to represent values which are guaranteed to be positive e.g. a memory address
- As a computer engineer/programmer, you must be able to manage both ➔ declare variable as a specific type e.g. ***int*** , **unsigned int** in C programming

Question

- What happens if we try to store values larger than specified number of bits?
- Unsigned numbers simply wrap around and start over at zero: **return to zero** e.g. if using 4 bits and add 1 to 1111, we get 0000
- Signed numbers use half their space for +ve numbers and the other for –ve numbers. Adding 1 to the largest positive 2's complement number (+7) – 0111 results in 1000 (-8). Wraps around with change in sign!
- Need to anticipate these scenarios in program

Booth's Algorithm

- To speed up multiplication, need more techniques e.g. those which use the presence of consecutive 0's or 1's
- Inspect 2 bits of the multiplier at a time Q(i)Q(i-1), $0 \leq I \leq n-1$
- If 01, add multiplicand and shift
- If 10, subtract multiplicand and shift
- If 00 or 11, only shift

Examples

Example Consider the multiplication of the two positive numbers $M = 0111$ (7) and $Q = 0011$ (3) and assuming that $n = 4$. The steps needed are tabulated below.

M	A	Q	$Q(-1)$		
0111	0000	0011	0	Initial value	
0111	1001	0011	0	$A = A - M$	
0111	1100	1001	1	ASR	End cycle #1
0111	1110	0100	1	ASR	End cycle #2
0111	0101	0100	1	$A = A + M$	
0111	0010	1010	0	ASR	End cycle #3
0111	0001	0101	1	ASR	End cycle #4

+21 (correct result)

*2's complement
*arithmetic right shift
*1st bit of shifted A is same as previous bit

Floating Point Representation (Scientific Notation)

- A FP number can be expressed in the form

$$\pm m * b^{e}$$

- m = mantissa /significand , e = exponent , b = base(radix) of the exponent

Figure 4.11 Representation of a floating-point number

Example

- $17 = 17.0 \times 10^0 = 1.7 \times 10^1 = 0.17 \times 10^2$
- $17_{10} = 10001_2 \times 2^0 = 1000.1_2 \times 2^1 = 100.01_2 \times 2^2 = 10.001_2 \times 2^3 = 0.10001_2 \times 2^5$ etc.
- If we use 1 sign bit, 5 exponent bits and 8 bits for the significand then $0.10001_2 \times 2^5$ is:

0 0 0 1 0 1 1 0 0 0 1 0 0 0

sign exponent significand

Example

- Use excess-M representation for exponent to account for negative exponents
- e.g. using excess-15:

17_{10} becomes : e =15+5 =20

0 1 0 1 0 0 1 0 0 0 1 0 0 0

sign exponent significand

Normalization

- Various representations for a single number
- 0 10101 10001000 = 0.10001×2^6
- 0 10110 01000100 = 0.010001×2^7
- 0 10111 00100010 =?
- 0 11000 00010001 =?

Are all equivalent.

Normalization: The MSB must always be 1

Question: Express 0.0625 in normalized floating point form using the simple model with excess-15 bias (assume 5 and 8 bits for exponent and significand respectively)

Floating point arithmetic

- 1. Re-write both as raised to same power
- 2. Add
- 3. Maintain bigger exponent and truncate low-order bit

	17-15= 2nd power	
0	10001	11001000
0	01111	10011010
	15-15=0 power	
	11.001000	
+	0.10011010	
	11.10111010	*normalise andmaintain 2nd power
0	**10001**	**11101110**

HOMEWORK: HOW DO WE REPRESENT ZERO IN THIS FORMAT?

Floating point Arithmetic

0	10010	11001000	$= 0.11001000 \times 2^3$
X 0	10000	10011010	$= 0.10011010 \times 2^1$

- 0.11001000 X 0.10011010 = 0.0111100001010000
- Multiplying by $2^3 \times 2^1 = 2^4$ gives 111.10000101
- Renormalizing gives:
- **0 10010 11110000**

- **Errors:**
- Arise because computers have limited storage so cannot represent entire range of infinite real numbers
- e.g. in these examples, we cant represent bigger than 128 because they are bigger than 8 bits wide hence round off or drop low order bit
- Compute relative error as ratio (error/true value) *100

IEEE- 754 Floating Point Standard

- Single precision standard uses excess-127 bias and an 8-bit exponent.
- Significand assumes an implied 1 to the left of the radix point and is 23 bits
- Implied one is called “hidden bit or hidden 1”
- e.g: 5.5 = 101.1 = 0.1011×2^3 is represented as 1.011×2^2
- Exception to normalization rule!

TABLE 4.2 Characteristics of the IEEE Single and Double Floating-Point Formats

Characteristic	Single-precision	Double-precision
Length in bits	32	64
Fraction part in bits	23	52
Hidden bits	1	1
Exponent length in bits	8	11
Bias	127	1023
Approximate range	$2^{128} \approx 3.8 \times 10^{38}$	$2^{1024} \approx 9.0 \times 10^{307}$
Smallest normalized number	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$

Range, Precision and Accuracy

- Range: interval from the smallest to the largest value e.g. for 8 bit 2's complement, range is -128 to + 128
- Accuracy: how close a number is to its true value e.g. if cant represent 0.01 within our range but we can find a number that is close to it or reasonably accurate
- Precision: signifies how much information we have about a value and the amount of information used to represent the value e.g. 1.666 and 1.6660 have 3 and 4 decimal digits of precision respectively but none is more accurate than the other
- ** accuracy and precision are usually confused

Assembly Language Programming

- Necessary in some cases irrespective of presence of compiler
- Can shorten and fasten machine code than that generated by a compiler ➔ useful in embedded systems or portable apps where resources are limited; write some part of code in assembly language
- Directly maps machine code into a form humans can understand

Simple Machine

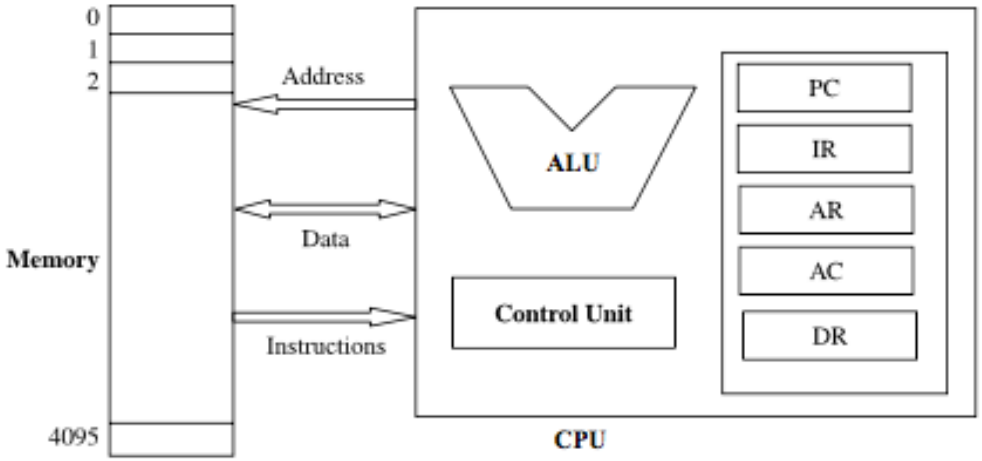

Figure 3.1 A simple machine

- Word size is 16 –bits
- Memory size is 4096 words
- 5 16-bit registers

•. Program counter (PC) – stores address of next instruction to be executed
•Instruction Register – contains operation code portion of the instruction being executed
•Address Register – contains the address portion of the instruction being executed
•Accumulator (AC) –serves as source and destination for data
•Data register – used to hold data

Instruction Set

TABLE 3.1 Instruction Set of the Simple Processor

Operation code	Operand	Meaning of instruction
0000		Stop execution
0001	*adr*	Load operand from memory (location adr) into AC
0010	*adr*	Store contents of AC in memory (location adr)
0011		Copy the contents AC to DR
0100		Copy the contents of DR to AC
0101		Add DR to AC
0110		Subtract DR from AC
0111		And bitwise DR to AC
1000		Complement contents of AC
1001	*adr*	Jump to instruction with address adr
1010	*adr*	Jump to instruction adr if AC = 0

Processor supports 3 instruction types:

- data transfer – load, store and move data between registers AC and DR
- data processing – add, subtract , and, not
- program control – jump and conditional jump

Example

- Machine code to add contents of memory location 12, initialized to 350 and memory location 14 initialized to 96 and store the result in location 16, initialized to 0.

TABLE 3.2 Simple Machine Language Program in Binary (Example 1)

Memory location (bytes)	Binary instruction	Description
0000 0000 0000	0001 0000 0000 1100	Load the contents of location 12 in AC
0000 0000 0010	0011 0000 0000 0000	Move contents of AC to DR
0000 0000 0100	0001 0000 0000 1110	Load the contents of location 14 into AC
0000 0000 0110	0101 0000 0000 0000	Add DR to AC
0000 0000 1000	0010 0000 0001 0000	Store contents of AC in location 16
0000 0000 1010	0000 0000 0000 0000	Stop
0000 0000 1100	0000 0001 0101 1110	Data value 350
0000 0000 1110	0000 0000 0110 0000	Data value is 96
0000 0001 0000	0000 0000 0000 0000	Data value is 0

Column 1: memory location in binary for each instruction and operand
Column 2: lists contents of memory locations
*** observe difficulty to understand and debug

Example

- Same program in HEX
- Reduces number of digits

TABLE 3.3 Simple Machine Language Program in Hexadecimal (Example 1)

Memory location (bytes)	Hex instruction
000	100C
002	3000
004	100E
006	5000
008	2010
00A	0000
00C	015E
00E	0060
010	0000

Instruction Mnemonics and Syntax

- Assembly programs are written in short abbreviations called mnemonics
- A mnemonic is an abbreviation that represents an actual machine instruction

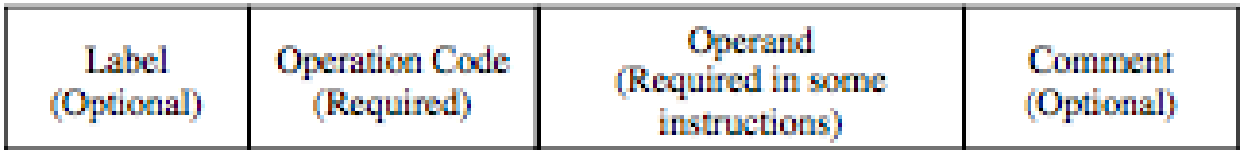

Figure 3.2 Assembly instruction format

-Label – provides symbolic name for a memory address
-Opcode – contains symbolic abbreviation of a given operation
-Operand – consists of additional information or data required by the opcode
-Comments: space for documentation for debugging and maintenance

Assembly Language

- For this example, comments are began with \

  START LD X \copy contents of location X into AC

TABLE 3.4 Assembly Language for the Simple Processor

Mnemonic	Operand	Meaning of instruction
STOP		Stop execution
LD	*x*	Load operand from memory (location x) into AC
ST	*x*	Store contents of AC in memory (location x)
MOVAC		Copy the contents AC to DR
MOV		Copy the contents of DR to AC
ADD		Add DR to AC
SUB		Subtract DR from AC
AND		And bitwise DR to AC
NOT		Complement contents of AC
BRA	*adr*	Jump to instruction with address *adr*
BZ	*adr*	Jump to instruction *adr* if AC = 0

Assembly Language

- Pseudo instructions/ assembler directives: commands understood by assembler but don't correspond to actual machine instructions
- In these examples W will be used to reserve a word in memory e.g.
- X W 350 \reserve a word initialized to 350

Assembly Language

- Write an assembly program to perform perform: Z ← X*Y ; X,Y and Z are memory locations

Assembly Language

```
        LD X            \ Load X in AC
        ST N            \ Store AC (X original value) in N
LOOP    LD N            \ AC ← N
        BZ EXIT         \ Go to EXIT if AC = 0 (N reached 0)
        LD ONE          \ AC ← 1
        MOVAC           \ DR ← AC
        LD N            \ AC ← N
        SUB             \ subtract 1 from N
        ST N            \ store decrements N
        LD Y            \ AC ← Y
        MOVAC           \ DR ← AC
        LD Z            \ AC ← Z (partial product)
        ADD             \ Add Y to Z
        ST Z            \ store the new value of Z
        BRA LOOP
EXIT    STOP
X       W       5       \ reserve a word initialized to 5
Y       W       15      \ reserve a word initialized to 15
Z       W       0       \ reserve a word initialized to 0
ONE     W       1       \ reserve a word initialized to 1
N       W       0       \ reserve a word initialized to 0
```

What do you notice?

Assembly Language

- Write an assembly program to perform perform: Z ← X*Y ; X,Y and Z are memory locations
- -No multiplication but added Y X times (simple CPU doesn't have multiplication operation)
- N used as a counter initialized to X and then decremented by one after each addition
- BZ instruction is used to test for when N reaches 0
- Memory location is used to store N but needs to be loaded into AC before BZ is executed
- Also use a memory location (ONE) to store the constant 1
- Memory location Z contains partial products and eventually final result

Assembly Language

```
        LD X            \ Load X in AC
        ST N            \ Store AC (X original value) in N
LOOP    LD N            \ AC ← N
        BZ EXIT         \ Go to EXIT if AC = 0 (N reached 0)
        LD ONE          \ AC ← 1
        MOVAC           \ DR ← AC
        LD N            \ AC ← N
        SUB             \ subtract 1 from N
        ST N            \ store decrements N
        LD Y            \ AC ← Y
        MOVAC           \ DR ← AC
        LD Z            \ AC ← Z (partial product)
        ADD             \ Add Y to Z
        ST Z            \ store the new value of Z
        BRA LOOP
EXIT    STOP
X       W       5       \ reserve a word initialized to 5
Y       W       15      \ reserve a word initialized to 15
Z       W       0       \ reserve a word initialized to 0
ONE     W       1       \ reserve a word initialized to 1
N       W       0       \ reserve a word initialized to 0
```

Assembler Directives and Commands

- Some practical issues need to be considered when writing assembly language

1. Assembler directives
2. Use of symbols : used to represent numbers
3. Use of synthetic operations: enables programmers to use instructions not supported by the architecture – translated by the assembler to those supported
4. Assembler syntax: some conventions are defined to refer to hardware components e.g. prefixes attached to names or values
5. Interaction with the operating system: hardware in some machines cant be accessed by a program directly but controlled by OS. Utilize system calls to do functions that are part of the OS

How do we go from assembly language to execution?

- 1. Assembler reads program and generates machine code
- 2. Passed to linker which checks for calls to procedures in the link library and combines them with the object program to form executable program
- 3.Loader loads the program into memory and branches CPU to the starting address
- 4.Program begins execution

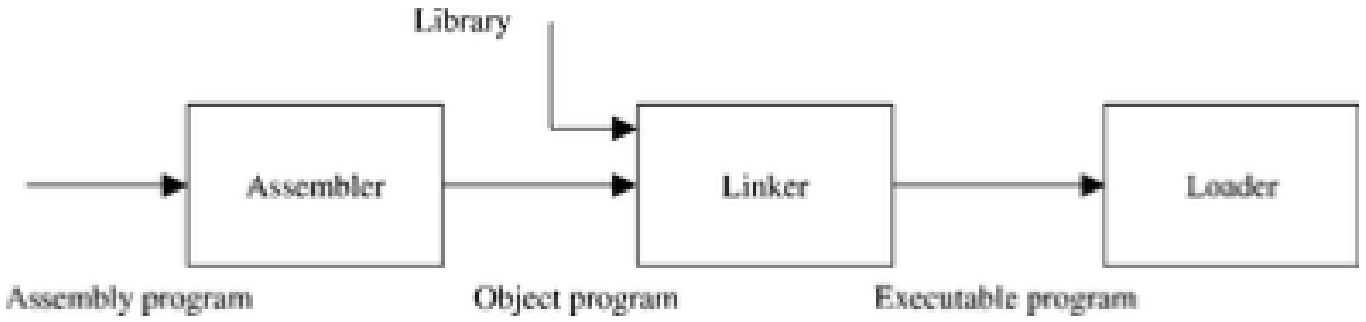

Figure 3.3 Assembly and execution process

Handout

- Linda Null and Julia Lobur, The Essentials of Computer Organization and Architecture , 4th Ed, Chapter 2 and 4
- Mostafa Abd-El-Barr, Hesham El-Rewini, Essentials of Computer Organization and Architecture, Chapter 3 and 4

References/ Handout

- Work out as many possible examples as you can!
- Linda Null and Julia Lobur, The Essentials of Computer Organization and Architecture , 4th Ed, Chapter 2
- Mostafa Abd-El-Barr, Hesham El-Rewini, Essentials of Computer Organization and Architecture, Chapter 4