CS 221 Computer Architecture

Week 3: Information Representation (2)

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Devices & Interrupts	(Chapter 8)
W10*	**Nov 13- Nov 16**	**Storage Management**	**(Chapter 9)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

Week 3: Plan

- **Hexadecimal Representation**
- **Character Representation**
- **Floating Point Representation**
- **Introduction to PEP/6**
- **Readings**
  - Chapter 3 of book
  - Next week: Chapter 4 (please start reading!)

I.
Hexadecimals and Character Representations

Hexadecimal Representation

- **Binary numbers are hard to read for humans.**
- **Hexadecimal and octal notations are just two different choices of the base in a weighted radix position number system.**
- **Hexadecimal is base 16, and octal base 8.**

Remember the Octal System

0	**10 (8)**	**70 (7*8 = 56)**	**… 100 (8*8 = 64)**
1	**11**	**71**	
2	**12**	**72**	
3	**13**	**73**	
4	**14**	**74**	
5	**15**	**75**	
6	**16**	**76**	
7	**17 (15)**	**77**	**This is called the octal system of counting or base 8**

101 =

$1 \times 8^2 + 0 \times 8^1 + 1 \times 8^0 = 65$

Hexadecimal System

0	10 (16)	...	F0 (15*16 = 240)	… 100 (16*16 = 256)
1	11			
2	12		72	
3	13		73	
4	14		74	
5	15		75	
6	16		76	
7	17			
8	18		For decimal 10 through 15, we use A to F	
9	19		This is called the hexadecimal system	
A	1A		or base 16	
B	1B			
C	1C		$101 = 1 \times 16^2 + 0 \times 16^1 + 1 \times 16^0 = 257$	
D	1D			
E	1E			
F	1F (31)			

So Why Hexadecimal?

- **Hexadecimal and Octal are useful because**
  - They are easier for people to deal with than a strict binary representation: a shorthand.
  - Because the bases are powers of two, the digits in hexadecimal and octal representations map straight onto groups of bits in the binary representation.
- **Each hexadecimal digit corresponds to 4 binary bits or a *nibble***
  - Each octal digit corresponds to 3 binary bits.

    0111 1111 binary 1010 1000 binary

    7 F hex A 8 hex
- **It is easy to remember the binary nibble patterns that correspond to a hexadecimal character!**

Example

- **F00D is a hexadecimal number**
- **F00D**

  $15 * 16^3 + 0 * 16^2 + 0 * 16^1 + 13 * 16^0 =$

  $15*4096 + 0*256 + 0*16 + 13 = 61453$
- **However, to convert hexadecimal into binary, just convert each hex digit into 4 bits:**
  - F00D is 1111 0000 0000 1101
  - The sign of a hex constant in 2's complement is indicated by the value of the most significant bit.
  - FOOD is negative because 'f' represents a bit pattern greater than or equal to 8.

From Decimal to Hex

- **Divide the number by 16, the base, and write down the remainder.**

4313 / 16	**remainder**	
269	**9**	
16	**d**	by now we have divided by 16^2
1	**0**	by now we have divided by 16^3
0	**1**	by now we have divided by 16^4

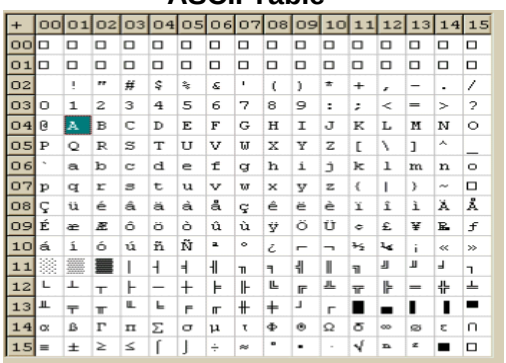
- **Now read back in reverse order: 10d9!**
- **Use hex as intermediate when converting to binary**

  10d9 hex is 0001 0000 1101 1001 in binary!

Representing Characters

- **Since all data are represented as bits, we need to encode characters as binary numbers.**
- **Assignment of values to the chars can be arbitrary**
  - But use of arithmetic comparisons of the codes should be possible to do alphabetical sorting.
- **ASCII (American Standard Code for Information Interchange) is the most common code.**
  - Encodes letters used in English, punctuation marks, and some control characters into 7 bit numbers.
  - However, typically a full byte is used for convenience, leaving the msb empty.

ASCII Table



Other Coding Schemes

- **There are standards to extend ASCII to include characters from more alphabets:**
  - Characters are commonly stored one per 8-bit byte, which gives you another bit to play with.
  - ISO-8859-x series codes the first 128 characters the same as ASCII, and the upper 128 characters are different sets of special symbols and accented characters from various Roman alphabets.
  - ISO 8859-1 is called "Latin One".
- **Unicode is a 16-bit encoding that represents scripts for all the alphabets in the world.**
  - Char types in Java use Unicode.
  - The least significant 8 bits of Unicode encodings are compatible with ISO-8859 and thus ASCII.

II.
Floating Point Representations

Announcements

- **There will be an extra tutorial on PEP/6 this Friday**
- **Details will follow**

So we haven't discussed fractions!

- **So how do we represent fractions in binary?**
  - We can't use any of the schemes we've used so far.
  - Because they can only represent powers of 2.
- **But instead of interpreting binary numbers using positive exponents like 2^n**
- **We could interpret them as negative exponents: 2^{-n} !!**

$1010 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 0*2^{-4}$

$1 * 1/2 + 0* 1/4 + 1*1/8 + 0*1/16 = .625$

Fractions in Decimal

- **Note that this works like decimal scientific notation**

  12.4568 (decimal) means

  $1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3} + 8*10^{-4} =$

  $1*10 + 2*1 + 4/10 + 5/100 + 6/1000 + 8/10000$
- **In scientific notation:**

  $12.4568 =$

  $1245680 * 10^{-5} =$

  $124568 * 10^{-4} =$

  $1245.68 * 10^{-2} =$

  $124.568 * 10^{-1} =$

  $1.24568 * 10^1$ *(this is called normalized notation)*

Negative Fractions in Binary

- **So how do we represent negative numbers?**
  - Simple: use 2^0 (=1).
  - It's the MSB, and we can use it as a sign bit.
  - So our original bit pattern should have been:

$01010 = - 0*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 0*2^{-4}$

$0 + 1 * 1/2 + 0* 1/4 + 1*1/8 + 0*1/16 = .625$

or

$11010 = -1 + 1 * 1/2 + 0* 1/4 + 1*1/8 + 0*1/16 = - .375$

- **Our notation now is called *Fixed Point* notation**

Floating Point Notation

- **Note that we could have put the decimal point at any arbitrary location and make it a standard.**
  - However, it would be fixed after that.
  - Which is why it is called fixed point
- **However, scientific calculations often involve very small as well as very large numbers.**
  - Would be neat if one number could represent
    The mass of the sun: $2 * 10^{33}$ grams
    The mass of an electron: $9 * 10^{-28}$ grams
  - We would need a lot of bits for this in fixed point.

To write them both out as complete decimal fractions that you could line up for addition would require 62 decimal digits (34 before the decimal point and 28 after it). Yet each number has only one significant digit, so the 64 digit representation would store a great many 0s.

Floating Point Notation (2)

- **The alternative would be to have a limited precision: we could not represent all numbers.**
  - Note that we can never represent all fractions!
- **Or have a floating point, where we could adjust precision according to our needs.**
  - Scientific decimal notation actually does this.
  - A number is represented by a fraction or mantissa which corresponds to the precision of the value.
  - The fraction is multiplied by ten raised to the power of a specified exponent (e below):

$$x = f * 10^e$$

*where f is the fraction or **mantissa** and e is the exponent*

So what exponent do we use?

- **Where do we put the decimal point?**
- **In scientific notation, fractions are normalized**
  - the exponent is adjusted so that there is one digit to the left of the decimal point.

12.4568 =
$1245680 * 10^{-5}$ =
$124568 * 10^{-4}$ =
$1245.68 * 10^{-2}$ =
$124.568 * 10^{-1}$ =
$1.24568 * 10^{1}$ (normalized notation)

- **So write $2.345678 * 10^4$ rather than $23.45678 * 10^3$**

Floating Point in Binary (2)

- **Floating Points in binary work the same as in decimal, except the base is *2* rather then *10***

$x = f * 2^e$ *where f is the fraction and e is the exponent*

23456.78 ($2.345678 * 10^4$) decimal corresponds to

23456 . 78

$101101110100000.110001111010 * 2^0$ =
and then we normalize by shifting right
$1.01101110100000110001111010 * 2^{14}$=
$2^{14} *(0* 2^{-1} + 1*2^{-2} … + 0* 2^{-27})$

Conversion to Binary: Whole part

- **How do you convert from decimal fractions to binary ones?**

Convert the whole number part using familiar method of repeated divisions & remainders

Quotients:	Remainders:
23456	
11728	0
5864	0
2932	0
1466	0
733	0
366	1
183	0
91	1
45	1
22	1
11	0
5	1
2	1
1	0
0	1
0	0
Result:	101101110100000

Conversion to Binary: Fraction

- **How do you convert the fractional part?**

Repeatedly multiply the fraction by two. When the result is greater than or equal to one, subtract one from the product and record a one in the corresponding bit position.

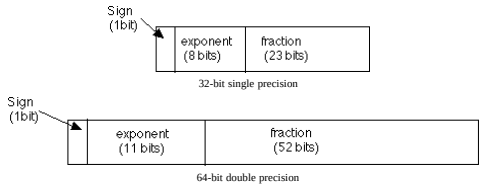
Fractions:	Whole parts:
0.78	
1.56	1
1.12	1
0.24	0
0.48	0
0.96	0
1.92	1
1.84	1
1.68	1
1.36	1
0.72	0
1.44	1
0.88	0
Result:	.110001111010

Conversion to Binary: Fraction (2)

- **This counts how many many halves are in the fraction, then how many quarters, how many eighths, how many sixteenths, etc.**
- **You stop until you have reached your precision (run out of bits).**
  - Or get exactly zero after subtracting one.
- **The resulting is the closest binary approximation for that fraction.**

So What Do Floating Points Look Like?

- **Remember x= f*2^e**
- **So how do bits represent the fraction & exponent?**
  - IEEE 754 Standard Floating Point Format.
  - One for 32 bit precision and 64 bit precision.
  - Corresponds to float and double in java.



What Do Floating Points Look Like (2)

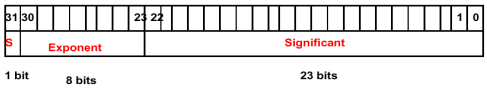
- **Sign bit different from integers or fixed point**
  - Simply 1 for negative numbers and 0 for positive numbers: thus 2 representations of zero (-0, +0)
- **The fractional part thus always represents an unsigned fraction**
- **The exponent bits need also represent negative.**
  - Do not use two's complement because we want 000000 to be the smallest and 111111 the largest.
  - Thus, we can more easily compare small and large numbers only by comparing their exponents.

Exponent = value of bits - (2^{n-1} -1)

If exponent has 8 bits, value is bits - 127 (range -127 to +127)

If exponent has 11 bits, value is bits - 1023 (range -1023 to +1023)

The Fraction, Mantissa, Significant



- **23 bits in single precision, 52 in double precision.**
- **When a binary fraction is normalized, there will always be a 1 to the left of the binary point.**
  - It therefore isn't represented in the IEEE format
  - The word significand is used to refer to the combination of the implied 1, the binary point, and the explicit 23 or 52 bits.
- **So we can actually represent 24 or 53 bits**

*x = sign * 1. fraction * 2^{exponent}*

Examples

- **Decimal fraction that is easy to convert to binary.**
  - 0.375 is a quarter plus one eighth
    *0.011(binary) * 2-0 or 1.1(binary) * 2-2.*
  - The sign bit is 0 for a positive number.
  - The exponent bits equivalent to 7D(hex) or 125(dec) represent an exponent of 125 - 127 = -2.
  - The remaining bits represent the significand 1.1(binary): remember that the leading 1 is not stored.

1	0111 1101	1000 0000 0000 0000 0000 000

IEEE 754 Single-precision floating point representing 0.375

Floating Point Overflow & Underflow

- **With integers we have the problem of overflow when the absolute value of a number is too great to fit into the number of bits**
  - With floating point, there is the possibility of underflow as well: an absolute value too small to be represented.
- **Take single-precision values. When the fraction is normalized, we can't express negative values which are less than:**

- 1.11111111111111111111111 (binary) * $2^{254\text{ -}127}$
(~ -2 * 2^{127} = -2^{128}, or -1.7 * 10^{38})

or positive values greater than

+1.11111111111111111111111(binary) * $2^{254\text{ - }127}$
(~ 2 * 2^{127} = 2^{128} or 1.7 * 10^{38}).

- **Numbers whose absolute values are greater than these limits constitute an overflow.**

Floating Point Overflow & Underflow

- **We also are unable to represent non-zero negative numbers greater than:**

  $-1.00000000000000000000000_{(binary)} * 2^{1-127}$ (-2^{-126}, approx $-1.2 * 10^{-38}$)

- **i.e. negative fractions with tiny absolute values, or non-zero positive numbers less than:**

  $1.00000000000000000000000_{(binary)} * 2^{1-127}$ (2^{-126}, approx $1.2 * 10^{-38}$)

- **Numbers whose absolute values are less than these limits constitute an underflow.**

Special Values

- **In addition to the normalized numbers, there are 4 special cases:**
  - If the exponent is 0 and the fraction is 0: the number represents + or - 0.
  - If the exponent bits are all 1s and the fraction bits are all 0s: + or - infinity (e.g., division by zero)
  - If the exponents are all 1s and the fractional bits are anything but all zeros this is *not a number (NaN)*, the value used to represent undefined results like infinity divided by infinity. Also + or -
  - If the exponent is 0 and the fractional bits are anything but all zeros this is a 'denormalized' number, with 0 instead of 1 as the implicit bit before the binary point. Used to represent smaller numbers, up to + 2^{-150}.

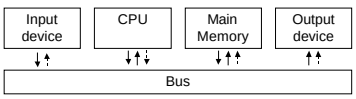
II.
Introduction to PEP/6

What is PEP/6

- **PEP/6 is a virtual computer running on your real computer as a software program**
- **It was designed to teach you the essence of programming machine code and assembly**
  - Without becoming too hardware dependent.
  - Without having to crash a unix box.
- **We will use the PEP/6 simulator throughout the course as an instructional tool**
- **It's up to you to relate this to real processors.**

Simulated Hardware

- **PEP/6 Hardware consists of four major components at the machine level:**
  - The central processing unit (CPU)
  - The main memory
  - The input device
  - The output device
- **Communicate with each other via a bus.**



The CPU

- **The CPU contains seven direct onboard memory locations called registers**
  - The 4-bit status register (NZVC)
    - Negative Zero oVerflow Carry
  - The 16-bit accumulator (A)
    - Contains result of operation
  - The 16-bit index register (X)
    - Contains result of operation
  - The 16-bit base register (B)
    - Contains start address of array
  - The 16-bit program counter (PC)
    - Contains address of current instruction
  - The 16-bit stack pointer (SP)
    - Contains address of a stack
  - The 24-bit instruction register (IR)
    - Holds instruction after fetch from memory

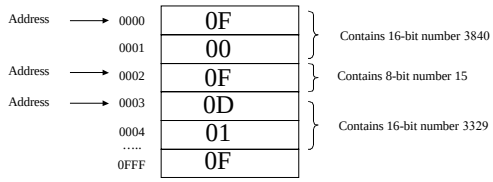
The CPU also holds all the circuitry to execute instructions

Main Memory

- **This is the Random Access Memory of PEP/6**
  - It contains 4K (4096 bytes)
  - It's cell size is 8 bits or 1 byte
- **Each byte has an address like a mail box**
- **Addresses range from 0 to 4095, 0000 to 0FFF hex**
  - Each address indicates location of a byte
  - Addresses indicate location of a value!
  - Not the value itself (remember pointers?)!!
- **Bytes with small address are top of memory**
- **Bytes with large address are bottom**

Main Memory

- **Word size in PEP/6 are 2 bytes or 16-bit**
  - Corresponds to registers: a 16-bit computer
  - A word occupies 2 memory locations!
  - You are responsible for knowing data types!

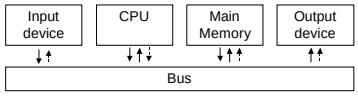


Input and Output Devices

- **PEP/6 Simulates two input devices:**
  - A text file
  - The keyboard
  - You must specify one or the other!
- **PEP/6 also simulates two output devices:**
  - A text file
  - The screen
  - You must specify one or the other!

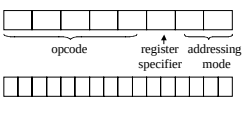
Data and Control

- **Data can flow from input device on bus to memory**
- **It can also flow from main memory to CPU**
- **But not directly to CPU from input device!**
- **Same is true for output: sent to memory first!**
- **Dashed lines indicate control: originate from CPU**
- **CPU sends signals to fetch/store data from memory**
- **CPU thus controls all parts of computer**



PEP/6 CPU Instruction Set

- **Instructions are the binary codes that indicate CPU operations such as addition or left shift.**
  - Are sent to CPU to tell it what to do
  - They are stored in main memory by programmer
- **PEP/6 has 32 instructions in its instruction set**
- **Each instruction consists of two parts:**
  - Instruction specifier: *specifies the operation*
  - Operand specifier: *specifies input data*

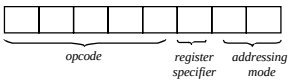


instruction

16-bit operand

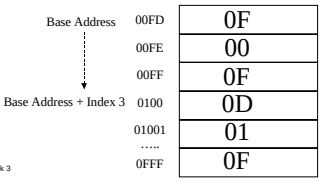
Instruction Specifier

- **Instruction specifier is divided into three fields:**
  - The actual five-bit operation code (opcode)
  - The one-bit register specifier
    - 0 = Perform operation with operand on accumulator (A)
    - 1 = Perform operation with operand on index register (X)
  - The two-bit input addressing mode specifier
    - 00 = Immediate (i) *operand specifier contains the value itself*
    - 01 = Direct (d) *operand spec. contains address of value in memory*
    - 10 = Stack relative (s) *value at index in memory relative to stack address*
    - 11 = Indexed (x) *value at index relative to address in base register*



Indexed Addressing Modes

- **Value of the operand added to one of two registers:**
  - The stack register (SP), which contains the address in memory where stack is located
  - The base register (B), which can contain any address in memory
- **In these modes, the operand is used as an index into an array of bytes in memory: handy!**



Opcodes

- **These specify which operation should be done**
  - An ADD, an AND or whatever
- **The opcode is decoded by the CPU into a sequence of logic gate operations**
- **In PEP/6, the opcode is 5 bits, total of 32 opcodes**
  - Some are left for future
  - Table 4.1 page 126 in book shows all opcodes
- **We will go through some of them**

Unary vs. Binary Instructions

- **Some opcodes require an operand, some not**
  - Depends on whether operation is unary or binary.
  - Remember: NOT is unary, ADD is binary.
- **Unary opcodes have the next instruction one byte further in memory**
  - There is no operand specifier
  - The addressing mode specifier is ignored
  - Operate on registers A or X
- **Binary opcodes have the next instruction three bytes further in memory**
  - The Program Counter is used to maintain the location of the next instruction.

Unary vs. Binary Instructions (2)

In the below example you see instruction located in main memory. The first instruction is binary and therefore occupies 3 locations. The next is unary and occupies only one location.

binary	**Instruction: ADD**	00FD	0001 1010
	Operand Spec. MSB	00FE	0000 0011
	Operand Spec. LSB	00FF	0100 1110
unary	**Instruction: ASR**	0100	0100 1000
binary	**Instruction: ADD**	0101	0001 1010
	Operand Spec. MSB	0102	0000 0011
	Operand Spec. LSB	0103	0100 1110
unary	**Instruction: ASR**	0104	0100 1000

Instruction Opcodes

00000 *Stop Execution*

- Makes computer stop.
- Ignores addressing mode and register specifier

00001 *Load Operand into Register R*

- Loads one word (2 bytes) from main memory into either the index register or accumulator
- Affects N and Z status bits, but not V or C

Example

- 00001 0 01 — 0000 0000 0100 1010

  Load — A — direct addressing — operand address 004A (h) in main memory

Before A: 036D (h) *After* A: 92EF (h) N=1
Mem[004A]: 92EF (h) Mem[004A]: 92EF (h) Z=0

Store

00010 *Store Register R to Operand*

- Stores one word from either the index register or accumulator into main memory
- Affects no status bits

Example

- 00010 1 01 — 0000 0000 0100 1010

  Store — X — direct addressing — operand address 004A (h) in main memory

Before X: 92BC (h) *After* X: 92BC (h)
Mem[004A]: F082 (h) Mem[004A]: 92BC (h)

ADD

00011 *Add Operand to Register R*

- Adds one word from main memory to either the index register or accumulator
- Affects all status bits

Example

- 00011 0 01 0000 0000 0100 1010

  Add A direct addressing operand address 004A (h) in main memory

Before A: FFB6 (h) *After* A: FFd9 (h)
Mem[004A]: 0023 (h) Mem[004A]: 0023 (h)

NZVC: 1000 result is negative

Subtract

00100 *Subtract Operand to Register R*

- Subtracts one word from main memory from contents of either the index register or accumulator
- Affects all status bits

Example

- 00100 1 01 0000 0000 0100 1010

  Subtract X direct addressing operand address 004A (h) in main memory

Before X: 000d (h) 13 (d) *After* X: FFFE (h) -2 (d)
Mem[004A]: 000F (h) 15 (d) Mem[004A]: 000F (h) 15 (d)

NZVC: 1001 result is negative and carry because we borrowed into the msb

AND

00101 *AND Operand to Register R*

- Performs logical AND operation to either the index register or accumulator with a word from memory
- Affects N and Z bits
- Handy for stripping away bits: masking
  - *AND with 0 strips bits, AND with ones keeps bits*

Example: Strip most significant 12 bits (3 nibbles)

- 00101 1 01 0000 0000 0100 1010

  And X direct addressing operand address 004A (h) in main memory

Before X: 6B5D (h) *After* X: 000D (h)
Mem[004A]: 000F (h) Mem[004A]: 000F (h)

NZVC: 0000

OR

00110 *OR Operand to Register R*

- Performs logical OR operation to either the index register or accumulator with a word from memory
- Affects N and Z bits
- Handy for inserting one bits into bit patterns: fill
  - *OR with 1 always sets bit*

Example: Set most significant 12 bits to ones (3 nibbles)

- 00110 1 01 0000 0000 0100 1010

  Or X direct addressing operand address 004A (h) in main memory

Before X: OC4D (h) *After* X: FFFD (h)
Mem[004A]: FFF0 (h) Mem[004A]: FFF0 (h)

NZVC: 1000 because register turned negative

Questions?