RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

Recitation 5

Jae Woo Joo

One’s Complement

- Represent negative numbers by complementing positive numbers

- It has two zero representation

000	001	010	011	100	101	110	111
0	1	2	3	-3	-2	-1	-0

Two's Complement

- Advantages – only 1 zero & convenient for arithmetic computation

- Flip the bits and add 1 (One's complement + 1)
- Ex)

40 = 0010 1000
-> Flip 1101 0111
-> Add1 1101 1000 (-40 in two's complement form)

-40 = 1101 1000
-> Flip 0010 0111
-> Add1 0010 1000 (two's complement of -40)

Two's complement

- What is the range that can represent with n bits?

$$[-2^{n-1}, 2^{n-1}-1]$$

- More negative numbers than positive numbers
  - Since we only have 1 zero

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

Examples

- Convert these negative decimal into negative binary using 2’s complement

- -192
- -16
- -1
- -0

Examples

- Convert these negative decimal into negative binary using 2's complement

- -192 => 0100 0000
- -16 => 1111 0000
- -1 => 1111 1111
- -0 => 0000 0000

Arithmetic of two's complement

- Arithmetic addition

+ 6	0000 0110		- 6	1111 1010
+13	0000 1101		+13	0000 1101
+19	0001 0011		+ 7	0000 0111

Arithmetic of two's complement

- Arithmetic subtraction

-5	1111 1011	-5	1111 1011
- 6	0000 0110	- -6	1111 1010
-11	1111 0101	+ 1	0000 0001

Two's complement overflow

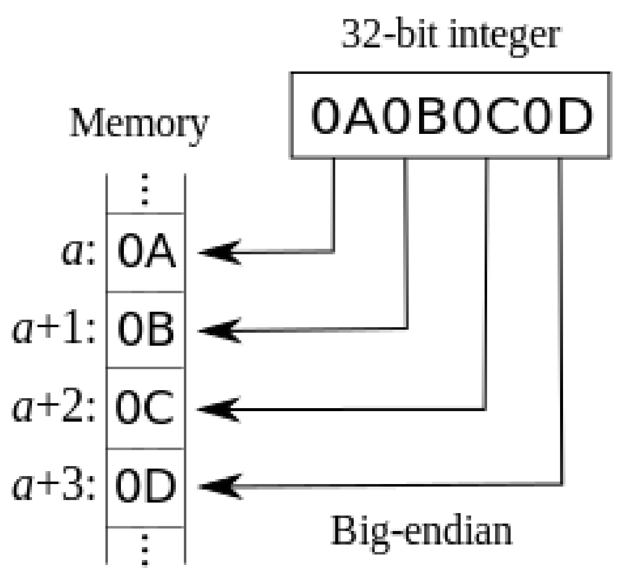
- It needs one extra bit but the sign bit will be wrong

- How to detect an overflow?
  - Adding 2 positive numbers -> But negative result
  - Adding 2 negative numbers -> But positive result

6	0110	-6	1010
+ 5	0101	+ -6	1010
-5	1011	4	0100

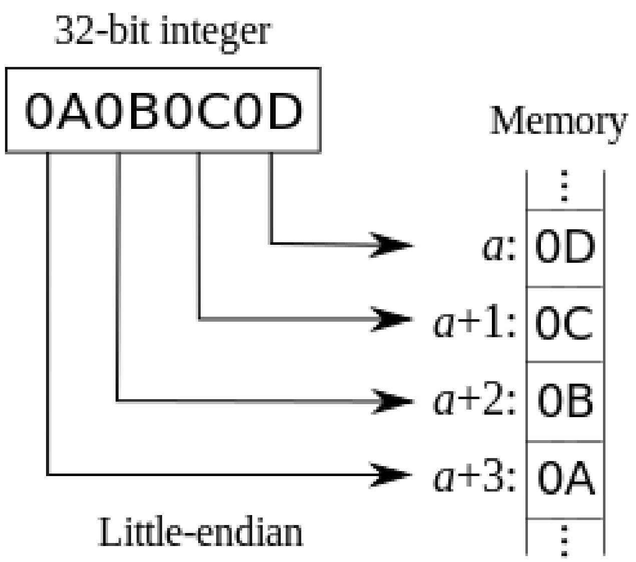
Endianness

- The order of the bytes stored in memory
- Big endian
  - MSB is stored at a particular address and the subsequent bytes are stored in the following higher memory addresses
  - LSB is stored at the highest memory address



Endianness

- Little endian
  - LSB is stored at the lower memory address and the subsequent bytes a re stored in the following higher memory addresses
  - MSB is stored at the highest memory address



Endianness (Byte ordering example)

- Consider the following word (32 bit) of memory

Little Endian LSB
Big Endian MSB

Little Endian MSB
Big Endian LSB

	AB	CD	00	00
Memory address	0	1	2	3

- Big Endian interprets as
  - AB CD 00 00 (2882338816)
- Little Endian interprets as
  - 00 00 CD AB (52651)

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
  - Encode 1 as “True” and 0 as “False”

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
  - Operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
__________    __________    __________    __________
  01000001      01111101      00111100      10101010
```

- All of the properties of Boolean Algebra Apply

Example: Representing & Manipulating Sets

- Representation
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$
    - **01101001** { 0, 3, 5, 6 }
    - *76543210*
    - **01010101** { 0, 2, 4, 6 }
    - *76543210*
- Operations
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

Bit-Level Operations in C

- Operations %, |, ~, ^ are available in C
  - Apply to any “integral” data type
  - View arguments as bit vectors
  - Arguments applied bit-wise

C expression	Binary expression	Binary result	Hexadecimal result
~0x41	~[0100 0001]	[1011 1110]	0xBE
~0x00	~[0000 0000]	[1111 1111]	0xFF
0x69 & 0x55	[0110 1001] & [0101 0101]	[0100 0001]	0x41
0x69 | 0x55	[0110 1001] | [0101 0101]	[0111 1101]	0x7D

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1

C expression	Result
!0x41	0x00 = "False"
!0x00	0x01 = "True"
0x69 && 0x55	0x01 = "True"
0x69 || 0x55	0x01 = "True"

Shift Operations

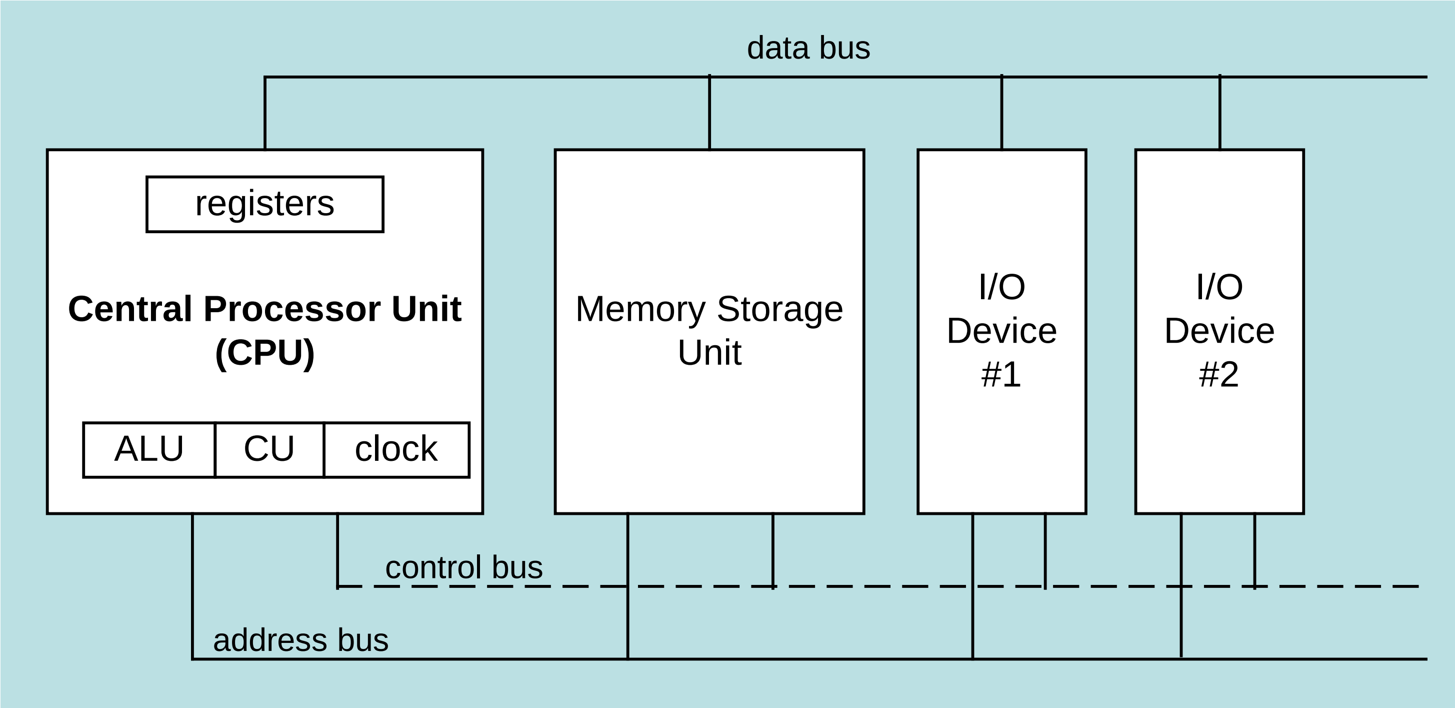
- Left Shift: x << y
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: x >> y
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	**01100010**
<< 3	**00010*000***
Log. **>> 2**	***00*011000**
Arith. **>> 2**	***00*011000**

Argument **x**	**10100010**
<< 3	**00010*000***
Log. **>> 2**	***00*101000**
Arith. **>> 2**	***11*101000**

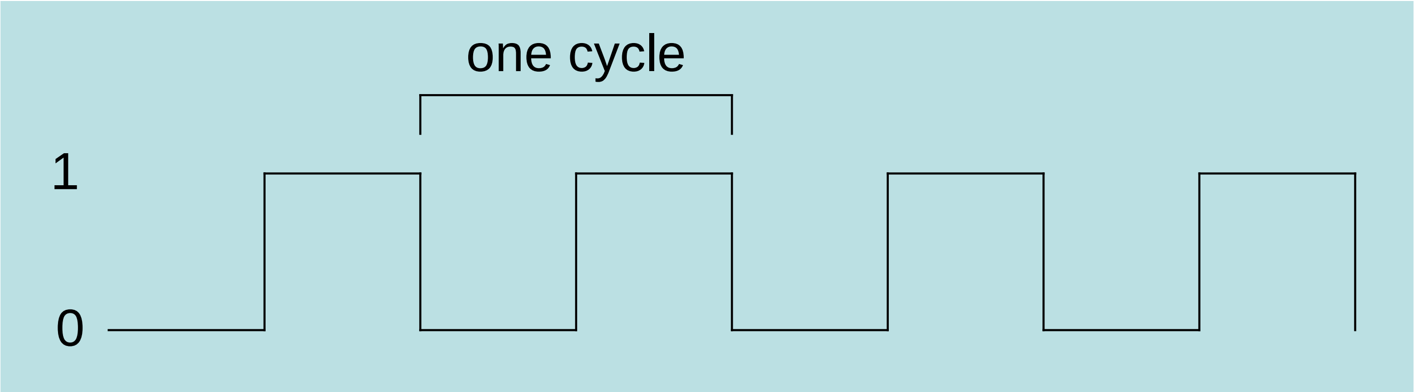
Basic Hardware Organization

- Clock synchronizes CPU operations
- Control Unit coordinates sequence of execution steps
- ALU performs arithmetic and bitwise processing



Clock

- Clock synchronizes all CPU and BUS operations
- Clock cycle measures time of a single operation
- Clock is used to trigger events



Instruction Execution Cycle

- Basic operation cycle of a computer
  - **Fetch**: The next instruction is fetched from the memory that is currently stored in the program counter
  - **Decode**: The encoded instruction present in the IR is interpreted
  - **Execute**: The control unit passes the instruction to the ALU to perform mathematical or logic functions and writes the result to the register.

Instruction Execution Cycle

Loop

- **fetch** next instruction
- advance the program counter (PC)
- **decode** the instruction
- if memory operand needed read from memory
- **execute** the instruction
- if result is memory operand, write to memory

Continue loop

CISC and RISC

- CISC – Complex instruction set computer
  - Large instruction set
  - High-level operations
  - Requires microcode interpreter
- RISC – Reduced instruction set computer
  - Simple, atomic instructions
  - Small instruction set
  - Directly executed by hardware

What is Assembly Language

- It is used to write programs in terms of the basic operations of a processor
- A processor understands only machine language instructions
- Machine language is too obscure and complex
- So low-level assembly language is designed for the processors

jaewoo.joo@cs.rutgers.edu

Advantages of Assembly Language

- Requires less memory and execution time
- Allows hardware-specific complex jobs in an easier way
- Suitable for time-critical jobs

Brief structure

High-Level Language Program

```
#include <stdio.h>
int main() {
    int x, y, temp;
    x=1; y=2;
    temp =x; x=y;  y=temp;
    printf("%d %d %d\n",x,y,temp);
}
```

Compiler

Assembly Language Program

```
movl    $1, -8(%ebp)
movl    $2, -12(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -16(%ebp)
movl    -12(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -16(%ebp), %eax
movl    %eax, -12(%ebp)
movl    -16(%ebp), %eax
movl    %eax, 12(%esp)
movl    -12(%ebp), %eax
movl    %eax, 8(%esp)
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
```

ISA

Assembler

Machine Language Program

```
7f 45 4c 46 01 01 01
 00 00 00 00 00 00 00
 00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```

Assembly Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Parts
  - Label (optional)
  - Opcode (also called as mnemonic)
  - Operand
  - Format
    - [label:]

      opcode operands
  - Example)

    movl %eax, %ebx

Labels

- Act as place markers
  - Marks the address of code and data

- Code label
  - Target of jump or loop instructions
  - Ex) L1: (followed by colon)

Opcode

- Instruction opcode
  - MOV
  - ADD
  - SUB
  - MUL
  - JMP
  - CALL

jaewoo.joo@cs.rutgers.edu

Operands

- Constant (immediate value)
  - Ex) 96
- Constant expression
  - Ex) 2 + 4
- Register
  - Ex) EAX

Registers

- Registers are CPU components that hold data and address
- Much faster to access than memory
- It is used to speed up CPU operations
- Categories
  - General registers
    - Data registers
    - Pointer registers
    - Index registers
  - Control registers
  - Segment registers

Q & A

- Any questions?