

Outline of Introduction

- Administrivia
- What is a computer?
- What is computer architecture?
- Why are there different types of computers?
- How do we tell computers what to do?

We Use High Level Languages

High Level Language Program

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

- There are many **high level languages (HLLs)**
 - Java, C, C++, C#, Fortran, Basic, Pascal, Lisp, Ada, Matlab, etc.
- HLLs tend to be English-like languages that are “easy” for programmers to understand
- In this class, we’ll focus on C/C++ as our running example for HLL code. Why?
 - C/C++ has pointers
 - C/C++ has explicit memory allocation/deallocation
 - Java hides these issues

HLL → Assembly Language

High Level Language Program

Compiler

Assembly Language Program

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
```

- Every computer architecture has its own **assembly language**
- Assembly languages tend to be pretty low-level, yet some actual humans still write code in assembly
- But most code is written in HLLs and **compiled**
 - **Compiler** is a program that automatically converts HLL to assembly

Assembly Language → Machine Language

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

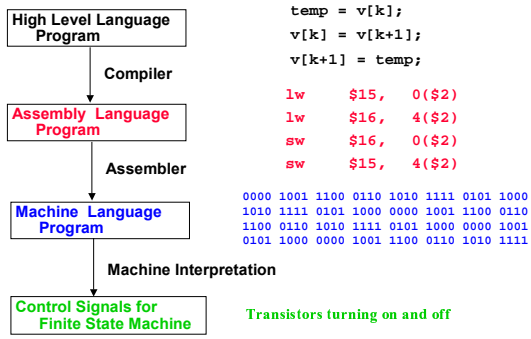
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $15, 0($2)
lw    $16, 4($2)
sw    $16, 0($2)
sw    $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

- **Assembler** program automatically converts assembly code into the binary **machine language** (zeros and ones) that the computer actually executes

Machine Language → Inputs to Digital System



Representing High Level Things in Binary

- Computers represent **everything** in binary
- Instructions are specified in binary
- Instructions must be able to describe
 - Data objects (integers, decimals, characters, etc.)
 - Memory locations
 - Operation types (add, subtract, shift, etc.)

Basic Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

- 4 bits is a **nibble**
- 8 bits is a **byte**
- 16 bits is a **half-word**
- 32 bits is a **word**
- 64 bits is a **double-word**

Character:

ASCII 7-bit code

Decimal: (binary coded decimal or BCD)

digits 0-9 encoded as 0000-1001 → 2 decimal digits packed per byte

Integers:

2's Complement (32-bit representation)

Floating Point:

- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended Precision (128-bit representation)

Issues for Binary Representation of Numbers

- There are many ways to represent numbers in binary
 - Binary representations are encodings → many encodings possible
 - What are the issues that we must address?
- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number
- **Choose representation that makes these issues easy for machine, even if it's not easy for humans**
 - Why? Machine has to do all the work!

Review: 2's Complement Integers

- Use large positives to represent negatives
- $(-x) = 2^n - x$
- This is 1's complement + 1
- $(-x) = 2^n - 1 - x + 1$
- So, just invert bits and add 1

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

6-bit examples:

$010110_2 = 22_{10}$; $101010_2 = -22_{10}$

$1_{10} = 000001_2$; $-1_{10} = 111111_2$

$0_{10} = 000000_2$; $-0_{10} = 000000_2 \rightarrow$ good!

Pros and Cons of 2's Complement

- Advantages:
 - Only one representation for 0 (unlike 1s comp): $0 = 000000$
 - Addition algorithm is much easier than with sign and magnitude
 - Independent of sign bits
- Disadvantage:
 - One more negative number than positive
 - Example: 6-bit 2's complement number
 $10000_2 = -32_{10}$; but 32_{10} could not be represented

All modern computers use 2's complement for integers

2's Complement Precision Extension

- Most computers today support 32-bit (int) or 64-bit integers
 - Specify 64-bit using gcc C compiler with `long long`
- To extend precision, use `sign bit extension`
 - Integer precision is number of bits used to represent a number

Examples

$14_{10} = 001110_2$ in 6-bit representation.

$14_{10} = 000000001110_2$ in 12-bit representation

$-14_{10} = 110010_2$ in 6-bit representation

$-14_{10} = 111111110010_2$ in 12-bit representation.

What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
 - Speed of light $\approx 3 \times 10^8$
 - Pi = 3.1415...
- Fixed number of bits limits range of integers
 - Can't represent some important numbers
- Humans use Scientific Notation
 - 1.3×10^4
- We'll revisit how computers represent `floating point numbers` later in the semester

What About Strings?

- Many important things stored as strings...
 - E.g., your name
- How should we store strings?

ASCII Character Representation

Oct.	Char	001	002	003	004	005	006	007
000	nul	soh	stx	etx	eot	enq	ack	bel
010	bs	ht	nl	vt	np	cr	so	si
020	dle	dcl	dc2	dc3	dc4	nak	syn	etb
030	can	em	sub	esc	fs	gs	rs	us
040	sp	!	"	#	\$	%	&	'
050	()	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[\]	^	_
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{		}	~	del

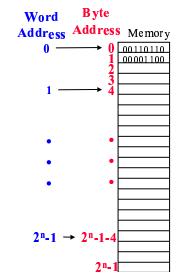
- Each character represented by 7-bit ASCII code.
- Packed into 8-bits

Computer Memory

- What is computer memory?
- What does it "look like" to the program?
- How do we find things in computer memory?

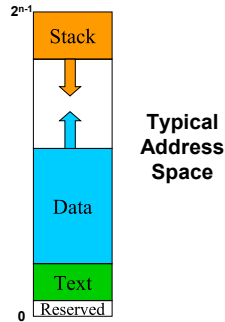
A Program's View of Memory

- What is memory? a bunch of bits
- Looks like a large linear array
- Find things by indexing into array
 - Index is unsigned integer
- Most computers support byte (8-bit) addressing
 - Each byte has a unique address (location)
 - Byte of data at address 0x100 and 0x101
 - Word of data at address 0x100 and 0x104
- 32-bit v.s. 64-bit addresses
 - We will assume 32-bit for rest of course, unless otherwise stated
 - How many bytes can we address with 32 bits? With 64 bits?



Memory Partitions

- **Text** for instructions
 - add dest, src1, src2
 - `mem[dest] = mem[src1] + mem[src2]`
- **Data**
 - Static (constants, globals)
 - Dynamic (heap, `new` allocated)
 - Grows upward
- **Stack**
 - Local variables
 - Grows down from top of memory
- Variables are names for memory locations
 - `int x; // x is a location in memory`

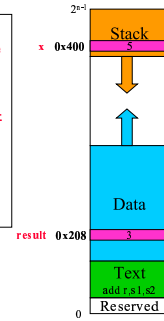


A Simple Program's Memory Layout

```

...
int result; // global variable
main()
{
  int x; // allocated on stack
  ...
  result = x + result;
  ...
}

```

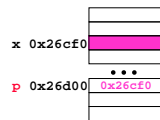


Pointers

- A pointer is a memory location that contains the address of another memory location
- "address of" operator & in C/C++
 - Don't confuse with bitwise AND operator (which is && operator)

Given
`int x; int* p;`
`p = &x; // p points to x (i.e., p is the address of x)`
Then
`*p = 2; and x = 2; produce the same result`

On 32-bit machine, p is 32-bits



Arrays

- In C++: allocate using array form of `new`

```
int* a = new int[100];
double* b = new double[300];
```
- `new[]` returns a pointer to a block of memory
 - How big? Where?
- Size of chunk can be set at runtime
- `delete[] a;` // storage returned
- In C:

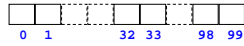

```
int* ptr = malloc(nbytes);
free(ptr);
```

Address Calculation

- x is a pointer, what is $x+33$?
- A pointer, but where?
 - What does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
- Result of subtracting two pointers is an int

$(d + 3) - d = \underline{\hspace{2cm}}$

```
int *a = new int[100]
```



$a[33]$ is the same as $*(a+33)$
 if a is $0x00a0$, then $a+1$ is $0x00a4$, $a+2$ is $0x00a8$
 (decimal 160, 164, 168)

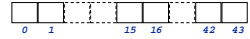
```
double *d = new double[200];
```



$*(d+33)$ is the same as $d[33]$
 if d is $0x00b0$, then $d+1$ is $0x00b8$, $d+2$ is $0x00c0$
 (decimal 176, 184, 192)

More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only
- what's at $*(begin+44)$?
- what does $begin++$ mean?
- how are pointers compared using $<$ and using $==$?
- what is value of $end - begin$?

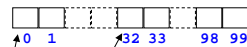


```
char* a = new char[44];  
char* begin = a;  
char* end = a + 44;
```

```
while (begin < end)  
{  
    *begin = 'z';  
    begin++;  
}
```

More Pointers & Arrays

```
int* a = new int[100];
```



a is a pointer
 $*a$ is an int
 $a[0]$ is an int (same as $*a$)
 $a[1]$ is an int
 $a+1$ is a pointer
 $a+32$ is a pointer
 $*(a+1)$ is an int (same as $a[1]$)
 $*(a+99)$ is an int
 $*(a+100)$ is trouble

Array Example

```
main()  
{  
    int* a = new int[100];  
    int* p = a;  
    int k;  
  
    for (k = 0; k < 100; k++)  
    {  
        *p = k;  
        p++;  
    }  
  
    cout << "entry 3 = " << a[3] << endl;  
}
```

Strings as Arrays

- A string is an array of characters with '\0' at the end
- Each element is one byte (in ASCII code)
- '\0' is null (ASCII code 0)

s	t			r	i	g	\0
0	1			15	16	42	43

Summary: Representing High Level in Computer

- Everything must be represented in binary!
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- We'll visit these topics again throughout semester

Outline of Introduction

- Administrivia
- What is a computer?
- What is computer architecture?
- Why are there different types of computers?
- How do we tell computers what to do?

What You Will Learn In This Course

- **The basic operation of a computer**
 - Primitive operations (instructions)
 - Computer arithmetic
 - Instruction sequencing and processing
 - Memory
 - Input/output
 - Doing all of the above, just faster!
- **Understand the relationship between abstractions**
 - Interface design
 - High-level program to control signals (SW → HW)

Course Outline

- Introduction to Computer Architecture
- **Instruction Sets & Assembly Programming (next!)**
- Central Processing Unit (CPU)
- Pipelined Processors
- Memory Hierarchy
- I/O Devices and Networks
- Performance Analysis (time permitting)
- Advanced Topics (time permitting)

The Even Bigger Picture

- ECE 52: Digital systems
- ECE 152: Basic computers
 - Finish 1 instruction every 1 very-long clock cycle
 - Finish 1 instruction every 5 short cycle
 - Finish 1 instruction every 1 short cycle (using pipelining)
- ECE 252: High-performance computers (plus more!)
 - Finish ~3-6 instructions every very-short cycle
 - Multiple cores each finish ~3-6 instructions every very-short cycle
 - Out-of-order instruction execution, power-efficiency, reliability, security, etc.