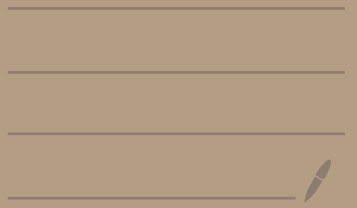
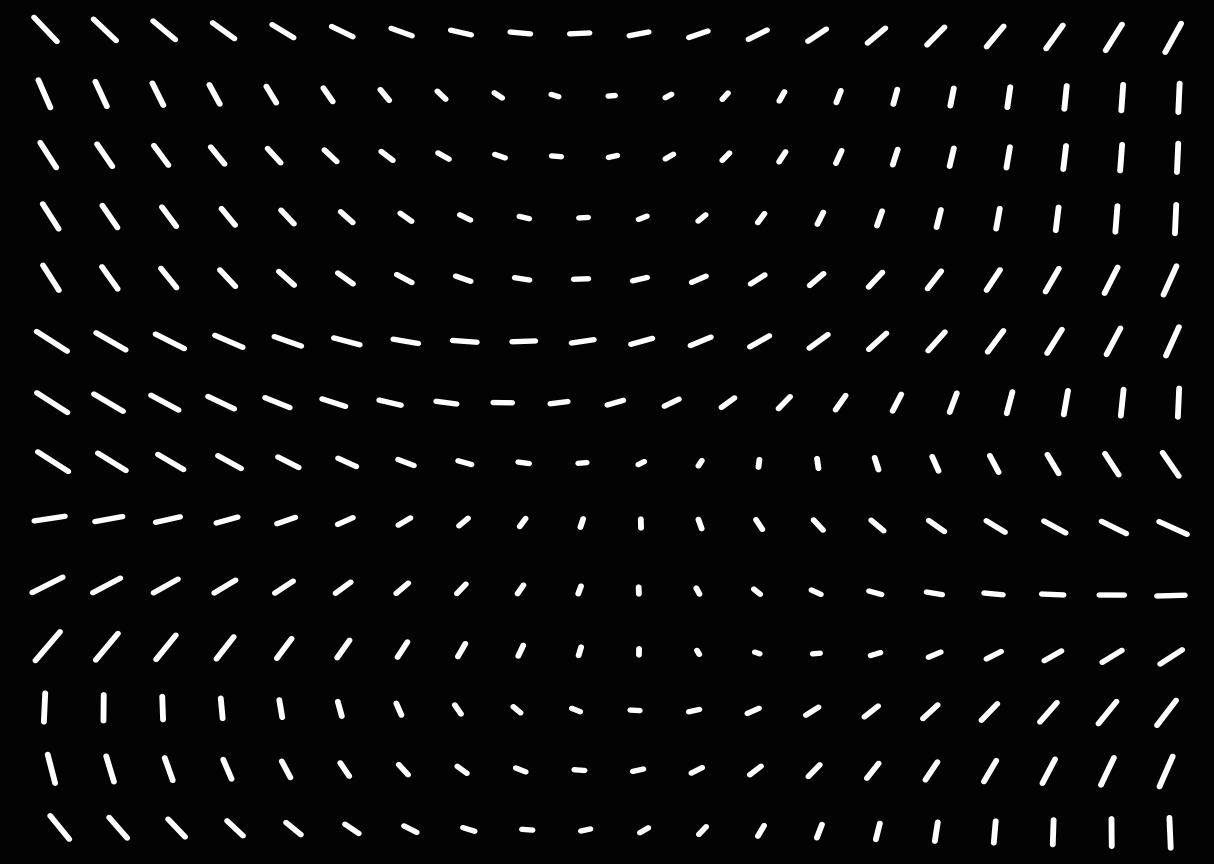
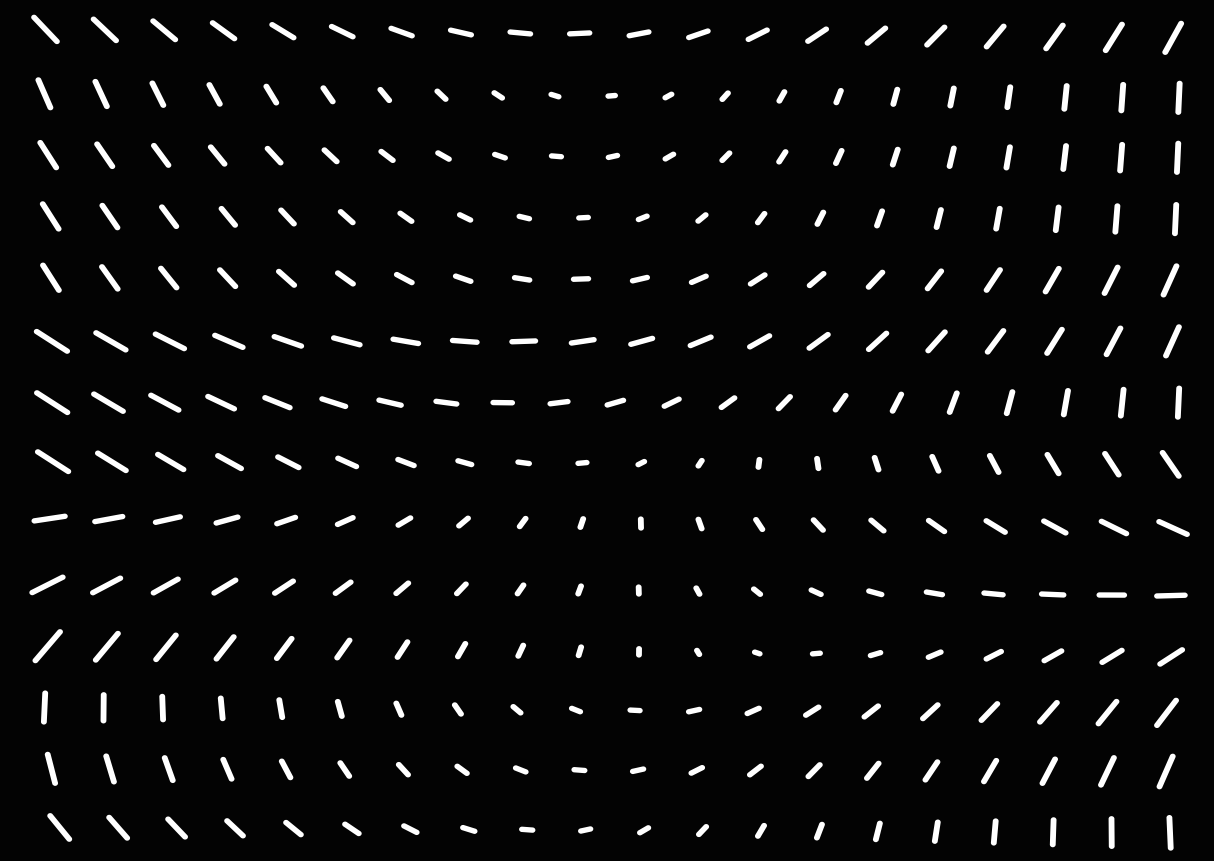
CISC 221



Lecture Notes



Unit 1



Lecture 1 - Introduction

Bit about C - language

- C - structural programming language
- files end in .c
- main function is required
- Must be compiled before they run
- lower level language, communicates with assembly and machine code.

- C program, compiler translates to assembly code, which processor understands as machine code.

- C has effecient compiler, preferred choice for system level programming

Assembly Advantages

- Debugging
- Performance Tuning
- Reverse Engineering

Slides 8 - 10

Interesting Notes

- Computers can only store numbers in a limited discrete format
- Common Problems with integer overflow and float precision

Computer arithmetic:

1) Finiteness - Numbers stored using fixed bits
2) Discreteness - operations are restricted to ranges

Memory Matters:

- Memory is not infinite or uniform

1) Cache and RAM - Data is easier for fast acess
2) Memory Bugs - Acessing invalid memory, causes crashes

- C and C++ does not provide memory protection.
- Memory access Pattern significantly affect performance

Holistic Interview:

High Level programming → Assembly → Machine Code → Execution

Memory Acess:

- Hierarchical structure (cache, RAM)
- Speed effeciency

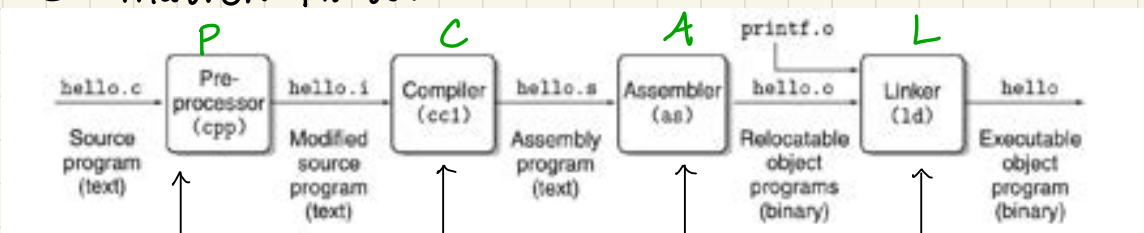
Lecture 2 - Introduction to C

Refresher: Life Cycle of a C Program

1) Write source code in C file
2) Compiling: Translates source code to machine readable instructions
3) Linking: Combine your code with libraries to executable file
4) Execution: Run Program as binary on operating system

From Source to Binary Code

Compilation Phase:

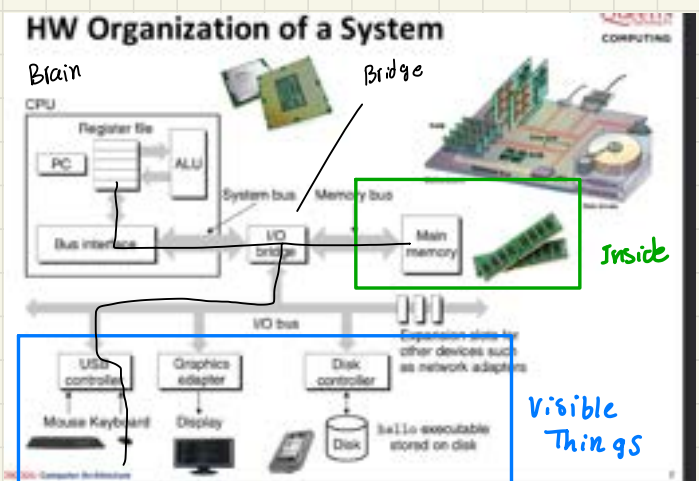


#Includes directives macros and comments

Converts C into Assembly code

Converts Assembly to machine Code

Combines object files into an executable file



For Hello: Read Command (Shell captures input, writes to memory, triggers program execution)

For Hello: Binary Command (DMA - Direct Memory Acess loads Binary File into memory for execution)

Course Focus

- Hardware Management: OS prevents misuse, Simplify app dev, manage concurrency and parallelism.

C Refresher

- Strongly Typed Language, has strict data type rules
- Basic Data Types: char, int, float, double, short, long
  (Big 4)
- Strings are arrays of char
- Type casting:

```
int height = 2;
float fheight = (float)height
```

Structs:

- A compound datatype for grouping related variables
- Acess elements using dot operator

Ex

Pointers

- Variable storing the memory adress of another variable
- Dereference a pointer using *: *ptr = 10; assigns 10 to x
- Be carefull, for memory leak

Arrays

- Contains multiple elements of same type
- Size is specified at decleration
- Squvare brackets for indexing

```
arrSize= 3;
int arr[3];
arr[0] = 12;
arr[2] = 34;

int arr2[] = {12, 34, 56};
```

Operators

All derived from C: +, -, /, *, ++, --, <, >, ||, and symbol

- Malle sure to cast correctly with these

Precedence and Associativity

Precedence: Determines order of evaluation (Bedmas)

Associativity: Determines order when precedence is equal

Flow Control

- Flow statements: if, if/else, while, do, for, switch/case
- C Does not have boolean types, instead 0 means false, else true
- Equality is ==

Functions

- use functional decomposition
- take arguments / return values
- variable declared inside functions have local scope.

```
int square (int i) {
    return i * i;
}
```

Program File Structure

.c - Code files

header files (.h) - has functions, prototypes, macros

- use # include, so files can see each other
- C has a large number of built in library files (.h files) and compiled code

Lecture 3- Bits and Bytes

Hardware Organization Recap

- Memory is organized hierarchically (registers, cache, RAM)
- Instructions and data are fetched from memory to be executed

Bits

- A bit is the smallest unit of data in a computer (represents 0, 1)
- Bits are building blocks for encoding data (numbers, characters, instructions)
- Bits are easy to implement using bistable elements (0 = low voltage, 1 = high voltage)
- Bits enable universal representation for everything

EX

2 Bits can represent 4 combinations

00, 01, 10, 11

Text: Each letter A or B is turned into a series of Bits

"A" in Binary: 01000001

Bits and Binary

- A bit is one binary digit (0 or 1)
- Binaly can encode numbers:

Each bit is a place $(2^2) + (2^1) + 2^0$

in binary system $4 + 0 + 1 = 5$

Memory

- Memory is structured by Adresses and corresponding data
- Adress: unique location
- Data: Stored information (binary)

Data Reperentation and Intepretation

- Binary data represents different types of information based on context. (could be Numerical, Alphabetical, other)

EX

Binaly could mean:

010 → 2
→ C
→ 0.2

Binary and Friends

- Radix (Base) - The base determines the number of unique digits to use.

Binary (base 2) → 0, 1

Decimal (base 10) → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimal (base 16) → 0-9, A-F

Binary and Friends - Representation

- Each position in a number corresponds to a power of the base

Value = $d_0 \times b^0 + d_1 \times b^1 + d_2 \times b^2$

where d_i is the digit and b is the base.

Conversion of Bases

Ex 1 Convert 1001011011 to decimal (Binary to decimal)

$1 \times 2^9 + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1$

$= 603$

Note: 1 hex digit = 4 bits

Ex 2 Convert 50 to Binary (integer to Binary)

$50 \rightarrow 2^5 + 2^4 + 2^1$

$32 + 16 + 2$

$= 110010$ in Binary

Ex 3 Hexadecimal to decimal

25 B → decimal

$2 \times 16^2 + 5 \times 16 + 11 \times 16^0 = 603$

Ex 4 Binary Fraction to decimal

10.0101

$1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$

$= 2.3125$

Ex 5 Hexadecimal to Binary Conversion

2F → Binary

2 in hex = 0010

F in hex = 1111

2F = 00101111

Ex 6 Convert Binary to Hexadecimal

1101101

1101 = D

1101 = D

Hex = DD

Bytes

- A byte is 8 bits

Represents values from:

Binary: 00000000 to 11111111

Decimal: 0 to 255

Hexadecimal: 00 to FF

Hexadecimal Representation in C:

- 0xFA1D37B represents the hex value
- Each digit corresponds to 4 binary bits
  F = 1111, A = 1010

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Byte-Oriented Memory Organization

- Each memory has an adress and content
- Pointers: store memory adresses and have a specific type associated with them.

Machine Words

- Word size is the nominal size of an integer-valued data (determines the size of virtual memory adress space.)
  [Typical values: 32-bit/64 bit)

Ex

- 32 bit system can adress 2^{32} bytes = 4 GB
- 64 bit system supports 2^{64} = 18 exabytes

Data Reperesentation

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	-	-	10/16
pointer	4	8	8

Note: Even if a variable is undefined it takes up the same amount of space in memory

Strings in C

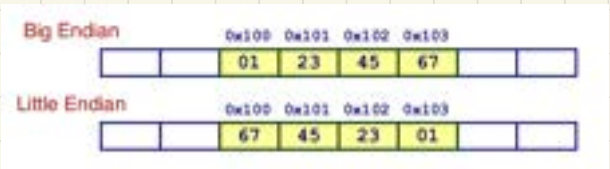
- Strings are arrays of characters
- Each character is encoded in ASCII (7-bit Standard)
- Strings are null terminated

Word-Oriented Memory Organization:

- A word is a fixed size unit of data used by a computer (4 bytes on 32bit)
- Memory is structured in bytes, but CPU reas and processes data in words.
- Each byte memory has uniaue adress

Byte Ordering

- Big Endian: Stores most significant byte (MSB) at smallest adress.
- Little Endian: Stores least significant byte (LSB) at the smallest adress



Reperesenting Integers

- Integers can be stored in binary, hexadecimal or decimal

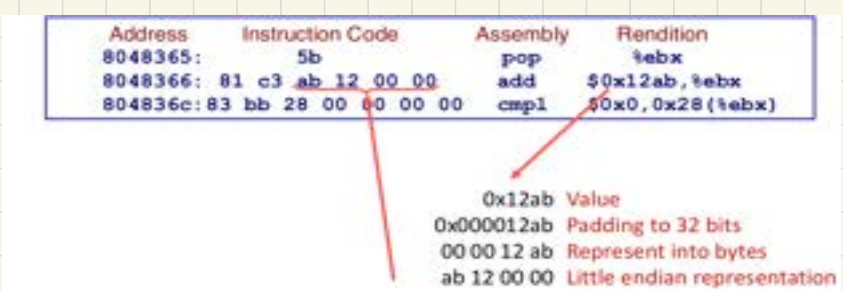
  Little Endian: 6D 3B 00 00
  Big Endian: 00 00 3B 6D
  } For systems

- Dissasembly is translating machine code into human readable assembly code.

Ex A value 0x12AB in little endian

raw memory layout: AB 12 00 00

This means least significant byte AB comes first



How Memory layout is displayed

Representing Pointers

- A pointer stores the memory adress of a variable
- Different compilers assign diff locations to objects.

Representing Strings

- String are array of characters, represented by 1 byte (4 bits)(ASCII encoding)
- Byte ordering doesn't affect strings

Little Endian and Big-Endian Store Numbers

- When a number is stored in memory, its binary representation is broken in to bytes (8 bit groups)

Ex Decimal: 15213 → Binary: 0011 1011 0110 1101

Break into bytes:

First byte: 0110 1101 → hex: 6D

: 0011 1011 → hex: 3D

: 0000 0000 → 00

0000 0000 → 00

Big endian: 00 00 3B 6D

Little endian: 6D 3B 00 00

Storage: 0x16ee6f31c

0x16ee6f31d

Memories in adress

Steps in Endian Conversion

1) convert to binary
2) Break into bytes
3) rearrange bytes depending on Big/Little Endian

Lecture 4 - Integers

Quick Recap

xp - Points to memory location
*xp - Points to actual value

- Variable like int x, long y are stored in memory with specific adresses
- Pointers (int *xp, long *yp) Stores adress of variables.
- If x starts at an adress 0x100, next int would stored next 4 bytes at 0x104.

Logical operators

- AND (&): True if both inputs are 1 (0 and 1 = 0)
- OR (|) : True if at least one input is 1.
- NOT (~) : Inverts the bit 0 to become 1.
- XOR (^) : True if inputs are different
- Note: (1) Means True, (0) Means False
- Can be reperesented using switches

Bit Level Operations in C

- C supports bitwise operations to integral datatypes (long, int, short, char)
- Operations are applied bit by bit.

Ex

1) ~0x41 : Inverts each bit 01000001 → 10111110
2) 0x69 and 0x55

Logical vs. Bitwise Operations

- Logical operators (&&, ||, !)
  - Treat 0 as false, non zero as true.
  - return 0 or 1
- Bitwise operators (&, |, ~, ^)
- Operate on the binary level

UnSigned Integers / Encoding Integers

- Unsigned integers only represent non-negative numbers
- Stored as binary

- Signed integers use sign bit to rep positive/negative
  - 0: Positive
  - 1: Negative

Unsigned: $B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$

- x_i: The bit at position i
- w: Total num of bits

Two's Complement

$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$

- Key idea is representing neg values by flipping bits adding 1

- Different data types have varying ranges, depending on signed/unsigned

Binary Number Property

- Binary follows base 2 positional system
- unsigned: 0 to $2^w - 1$
- Signed: -2^{w-1} to $2^{w-1} - 1$

Twos Complement

when u have neg to pos

- How negative numbers are stored:
  - Flip all bits
  - Add 1 to result

Ex 5: 00000101

-5: 11111010 Flip

Add 1: 11111011 Add 1

Ex 2

Positive: 00111011 01101101 → 15213

Negative: 11 000100 10010010 → -15213

Note: Larger word sizes allow for larger ranges

More bits means more range

Modular Arithmetic

- Numbers wrap around after reaching max vawe (2^w)

Unsigned Arithmetic

- operates within 0 to $2^w - 1$
- Adding / subtracting beyond this range, wraps around

Signed Arithmetic (Two's complement)

- operates within -2^{w-1} to $2^{w-1} - 1$

Shifting Operations

Left Shift (x << y)

- Moves bits of x to the left by y positions
- Fills right side bits with 0
- Multiplies the number by 2^y (if no overflow)

Ex
Binary: 0010 (2)
Shift 2 Left → 1000 (8)

Right Shift (x >> y):

- Move bits of x to right by y positions
- Logical Shift:
  - Fills left bits with 0 1010 (2) → 0010
- Arithmetic Shift:
  - Fills left bits with Sign bit 1110 >>2 → 1111

Special case: when the amount K > w (where w is word size)
behaviour is undefined in C but usually wraps around K mod w.

Left Shifts

x << 3

- Moves all bits in binary to left by 3 position
- Fills right with 0's

0xC3 → 1100 0011 → 00011000 → 0x18
For x << 3

Right Shift

- moves bits to right by h positions
- Logical fills left most bits with 0's
- Arithmetic fills with MSB

Lecture 5 - Integer Arithmetic

B2U(x) - Unsigned
B2T(x) - Signed
MSB - Most Significant bits

Recap - Numeric Datatypes

- Signed Integers: represent positive/negative values
  - Ex: int (4 bytes), short (2 bytes)
- Unsigned Integers: represent positive values, and 0
  - Ex: unsigned int (4 bytes)
- Real Numbers:
  - Ex: Floating-point values like float (4 bytes) and double (8 bytes)

Example:

- 8 bit binary: 11111010
  - Unsigned: B2U = 250
  - Signed: B2T = -6

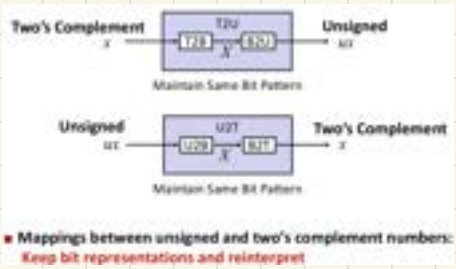
Conversion and Casting

- Key Ideas: Bit patterns are the same for both signed and unsigned numbers, but interpretation varies.

Mapping Between Signed/Unsigned

1.) Signed to Unsigned (T2U)
   - Retain bit pattern, interpret as unsigned

2.) Unsigned to Signed (U2T)
   - Retain bit pattern, interpret as signed



Mapping Signed / Unsigned

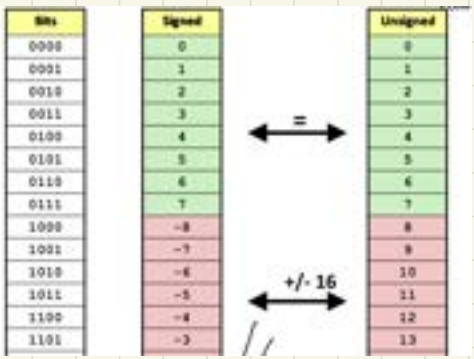
If you had 4 bits, instead of 8,
Unsigned values, would go up to 15, (1111)
Signed values, would go from −8 to 7.

Ex

To find range, use 1111 = −8 (twos compliment)
0111 = 7 (Positive)

Neg range: 1000 to 1111 (−8 to −1)
Pos range: 0000 to 0111 (0 to 7)

- A large unsigned number maps to a negative Signed number



Note: largest negative weight in Signed, becomes largest positive weight in unsigned.

Signed vs. Unsigned in C

- Constants by default are signed integers unless specified with U at end

- Casting is explicit, manually done to convert between signed / unsigned

int tx = (int) ux; cast Unsigned → Signed
unsigned uy = (unsigned) ty; cast Signed → Unsigned

Sign Extension

- extending a smaller integer to a larger type, while preserving its value
- Rule: For signed integers (Pos or neg):
  ① Extend the MSB to fill extra bits

Ex Positive number extension:
short value: 15213 → 00 111011 01101101
int value: Extended → 00000000 00000000 00111011 01101101

Same applies to negative value

Truncation

- Reducing a larger integer to a smaller type by dropping the leading bits
- Behaviour is equivalent to taking mod 2^k of number for unsigned

Summary Expanding / Truncating

- Expanding:
  - unsinged: Fills with zeroes
  - Signed: Extends with the sign bit
- Truncating:
  - Unsigned: Drops leading bits, mod 2^k
  - Signed: Similar to unsigned but reinterprets

Note: If truncation removes bits critical to determining the value, there will be data loss.

Unsigned Addition

- Operates in modular arithmetic:
  - Sum is calculated as $x + y \mod 2^w$, where w is bit width
  - True Sum: Calculated as $x+y$ with $w+1$ bits.
  - Modular Sum: Keeps only the lower w bits, discarding carry bit

Example

$x = 1111$, $y = 0001$

True Sum: 10000

Modular Sum: 0000 ($16 \mod 2^4$) ?

- Wrap around behaviour; if $x+y \geq 2^w$, sum wraps around $x+y-2^w$.

True Sum; Values exceed 2^w, but aren't stored fully

Modular Sum; Ranges from 0 to 2^w-1

Overflow when true sum exceeds w bit range

Mathematical Properties

- **Modular addition forms an *Abelian Group***
  - **Closed** under addition
    $0 \le x +^{u}_{w} y \le 2^{w}-1$
  - **Commutative**
    $x +^{u}_{w} y = y +^{u}_{w} x$
  - **Associative**
    $(x +^{u}_{w} y) +^{u}_{w} z = x +^{u}_{w} (y +^{u}_{w} z)$
  - **0** is additive identity
    $x +^{u}_{w} 0 = x$
  - Every element x has an additive **inverse**
    $2^{w} - x$

Two's Complement Addition

- Signed addition works the same as unsigned at the bit level
  - Operands are w-bits
  - True sum is w+1 bits
- result is truncated to w bits by discard carry

Overflow in Two's complement Addition

- Overflow happens when the true sum exceeds the signed range $(-2^{w-1}$ to $2^{w-1}-1)$
- Positive overflow: $x+y > 2^{w-1}-1$, wraps around to negative
- Negative overflow: $x+y < -2^{w-1}$, wraps to positive
- $-x = \sim x + 1$
  (flip all bits, add on)

Multiplication Basics

- Multiplying two w-bit numbers can produce a result that requires up to 2w bits
- Unsigned Multiplication:
  $0 \le x \cdot y \le (2^{w}-1)^{2} = 2^{2w} - 2^{w+1} + 1$
- Two's Complement Multiplication:
  $x \cdot y$ ranges from $-(2^{2w-2})$ to $(2^{2w-2}-1)$

Unsigned Multiplication in C

- True product is calculated as:
  $x \cdot y \bmod 2^{w}$
- The high order bits are discarded.

Signed Multiplication in C

- discards high-order bits but treat numbers as twos complement
- Lower bits are the same for signed and unsigned multiplication at bit level.

Power of 2 Multiplications

- Multiplication by power of 2 can be done using left shifts: $x \cdot 2^K = x << K$

Ex
$x << 3 = x \cdot 8$
$(x << 5) - (x << 3) = x \cdot 24$

- Shifting is faster than multiplication

Power of 2 Division

- Division by powers of 2 is performed using logical right shifts; $x \div 2^K = x >> K$

Ex
$x = 15213$
$x >> 1 = 7606$
$x >> 4 = 950$

Summary: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^n
    - Mathematical addition + possible subtraction of 2^n
  - Signed: modified addition mod 2^n (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^n
- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^n
  - Signed: modified multiplication mod 2^n (result in proper range)

Types of overflow

- Positive overflow, if $x > 0, y > 0$ and result $x+y$ is negative.

Ex

$x = 01100$
$y = 00100$

$x+y = 10000$
$= -16$

- Negative overflow, happens when two negative numbers are added, result becomes smaller than -2^{w-1}

$x = 10100$, $y = 10001$
(-12) (-15)

$-12 + -15 = -27$

Binary: $100010 1 \rightarrow 0\ 0101$, negative overflow.

Lecture 6/7 Floating Points

- Floating point numbers represented in the form:

$$V = (-1)^s \times M \times 2^E$$

  - S: Sign bit [0 = pos, 1 = neg]
  - M: Mantissa [frac part 1.0 – 2.0 usually]
  - Exponents

Normalized Numbers

- E = exp-biased (the exponent is not all 0 or 1's)
- 1. Frac

Quiz 1 Study guide

High Level programming → Assembly → Machine Code → Execution

Pointers

- Variable storing the memory adress of another variable
- Dereference a pointer using *: *ptr = 10; assigns 10 to x
- Be carefull, for memory leak
- *p = 20; Assignes value to memory location

C Refresh

- Strongly Typed Language, has strict data type rules
- Basic Data Types: char, int, float, double, short, long
  (Big 4)
- Strings are arrays of char
- Type casting:

```
int height = 2;
float fheight = (float)height
```

Bits

- A bit is the smallest unit of data in a computer (represents 0, 1)
- Bits are building blocks for encoding data (numbers, characters, instructions)
- Bits are easy to implement using bistable elements (0 = low voltage, 1 = high voltage)
- Bits enable universal representation for everything

Bits and Binary

- A bit is one binary digit (0 or 1)
- Binary can encode numbers:

Each bit is a place in binary system: $(2^2) + (2^1) + 2^0$

$4 + 0 + 1 = 5$

Binary and Friends

- Radix (Base) - The base determines the number of uniave digits to use.

Binary (base 2) → 0, 1

Decimal (base 10) → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimal (base 16) → 0-9, A-F

A = 10, B = 11 ... F = 16

Always needs to be 16 bits / 2 bytes

Bytes

- A byte is 8 bits

Represents values from:

Binary: 00000000 to 11111111

Decimal: 0 to 255

Hexadecimal: 00 to FF

Short - 32 bits
long - 64 bits

Hexadecimal Representation in C:

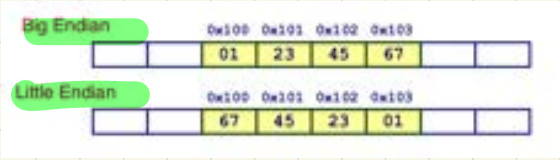
- 0xFA1D37B represents the hex value
- Each digit corresponds to 4 binary bits
  F = 1111, A = 1010

Strings in C

- Strings are arrays of characters
- Each character is encoded in ASCII (7-bit Standard)
- Strings are null terminated

Byte Ordering

- Big Endian: Stores most significant byte (MSB) at smallest address.
- Little Endian: Stores least significant byte (LSB) at the smallest address
- Endianess: determines how multibyte data is stored in memory



Little Endian and Big-Endian Store Numbers

- When a number is stored in memory, its binary representation is broken in to bytes (8 bit groups)

Ex Decimal: 15213 → Binary: 0011 1011 0110 1101

Break into bytes:

First byte: 0110 1101 → hex: 6D

: 0011 1011 → hex: 3D

: 0000 0000 → 00

0000 0000 → 00

Big endian: 00 00 3B 6D

Little endian: 6D 3B 00 00

Steps in Endian Conv

1) Convert to binary
2) Break into bytes
3) rearrange bytes depend. on Big/Little Endian

Bitwise operators

- AND (&): True if both inputs are 1 (0 and 1 =0)
- OR (|) : true if at least one input is 1.
- NOT (~) : Inverts the bit 0 to become 1.
- XOR(^ : True if inputs are different
- Note: (1) Means True, (0) Means False
- Can be represented using switches

Bit Level Operations in C

- C supports bitwise operations to integral datatypes (long, int, short, char)
- Operations are applied bit by bit.

Ex

1) ~0x41 : Inverts each bit 01000001 → 10111110
2) 0x69 and 0x55

Logical Operations

- Logical operators (&&, ||, !) → use double signs
  - Treat 0 as false, non zero as true.
  - return 0 or 1
  - Uses ! instead of ~

Unsigned Integers / Encoding Integers

- Unsigned integers only represent non-negative numbers
- Stored as binary
- Signed integers use sign bit to rep positive/negative
  - 0: Positive
  - 1: Negative

Unsigned: $B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$

- x_i: The bit at position i
- w: Total num of bits

Two's Complement

$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$

- Key idea is representing neg values by flipping bits adding 1

Binary Number Property

- Binary follows base 2 positional system
- unsigned: 0 to $2^w - 1$
- Signed: -2^{w-1} to $2^{w-1} - 1$

Twos complement

when u have neg to pos

- How negative numbers are stored:
  - Flip all bits
  - Add 1 to result

Ex
5: 00000101
-5: 11111010 Flip
Add 1: 11111011 Add 1

Ex 2
Positive: 00111011 01101101 → 15213
Negative: 11 000100 10010010 → -15213

Note: Larger word sizes allow for larger ranges

More bits means more range

Modular Arithmetic

- Numbers wrap around after reaching max value (2^w)

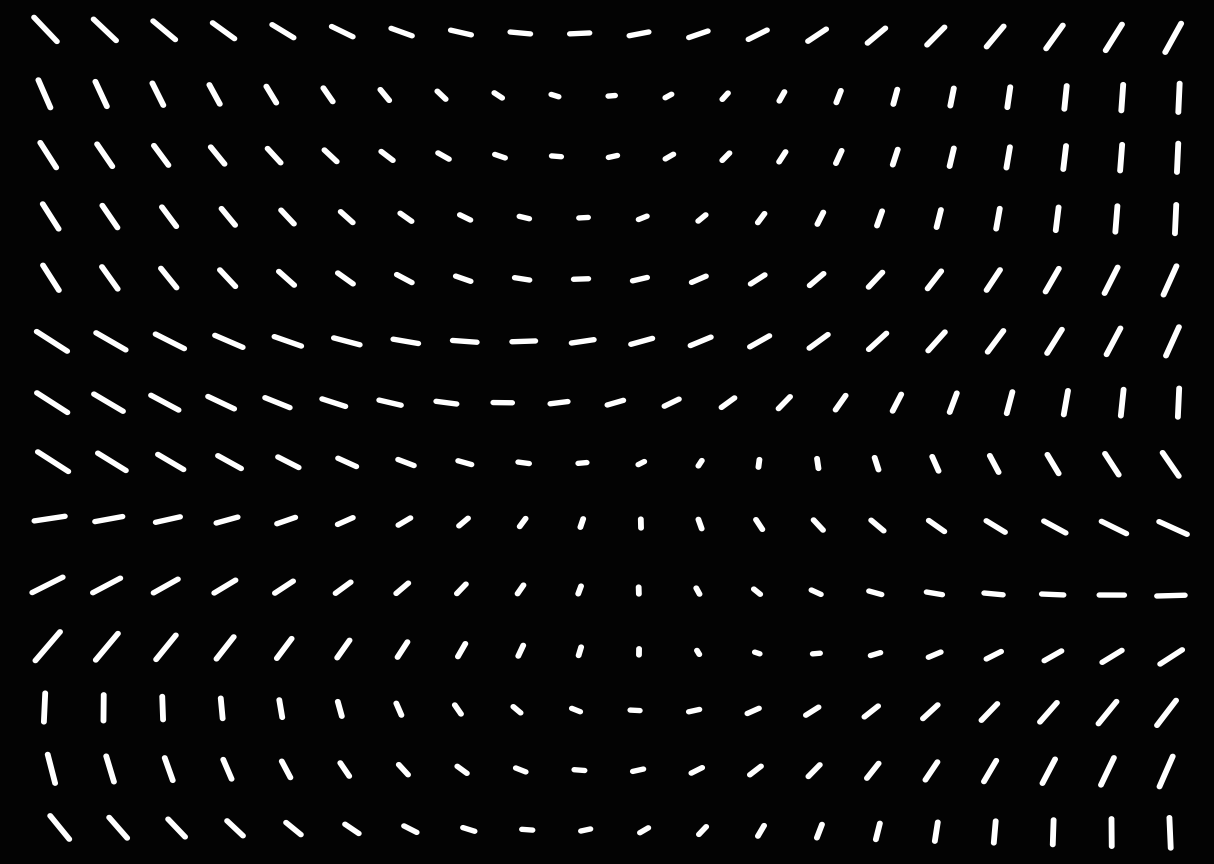
Unsigned Arithmetic

- operates within 0 to $2^w - 1$
- Adding / subtracting beyond this range, wraps around

Signed Arithmetic (Two's complement)

- operates within -2^{w-1} to $2^{w-1} - 1$

Unit 2



Lecture 9 - Machine Fundamentals

Importance of Machine Code

1) Optimize efficiency, understanding how compilers generate code
2) Analyze run time behaviour, see low level execution
3) Improve security, identify vulnerabilities

C, Assembly, Machine Code

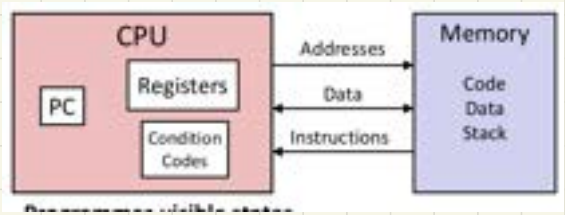
High Level C → Assembly → Machine code

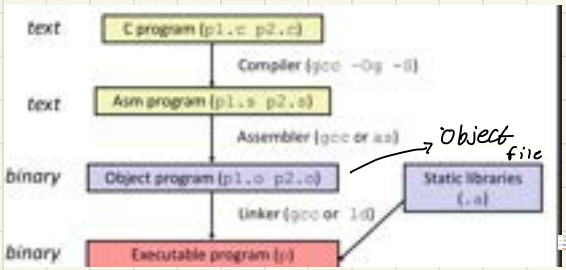
- Starts with high level C-code
- Then compiler translates C to assembly language (human readable version of machine code)
- Machine code (Binary) → Assembly is converted into Binary machine code executed by CPU.

- Architecture - known as ISA, defines parts of processor needed to write assembly/machine code
- Micro-architecture - physical implementation of ISA

Basic CPU Components

- PC (Program Counter) - holds address of next instruction to be executed, automatically increments
- Registers - small fast storage locations, used to hold temporary data. (many diff types)
- Condition Codes (Flags) - special registers, store info on recent arithmetic/logic





Compilation Process

- To convert .c file to an executable

1) Compilation: Translates C into assembly using gcc
2) Assembly: Converts to machine code
3) Linking: Combines object code with libraries to make executable program

Compiling to Assembly

- assembly is annotated by passing parameters, functions
- details size and data type, specifies size in memory

Assembly Operations

- **Data transfer:** moving data between memory / registers
- **Arithmetic Functions:** Perform calculations using data / memory
- **Control Transfer:** managing flow of execution with in code (branches / conditionals)

- **Assembler:** Translates assembly instructions into binary object code
- **Linker:** Finalizes creation of an executable

C declaration	Intel data type	Assembly code suffix	x86-64 size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long int	Quad word	q	8
long long int	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

sses (untyped pointers)

Assembler

- Translates .s into .o, which is binary representation of assembly code.
- Includes binary encoding
- Linking, process of creating executable

Assembler / Machine Code Role

- Assembler Role: Translates assembly code into binary object code, this machine code is what the computer's CPU executes.
- Machine code: Holds the binary data that instructs the CPU on what to do (hingly optimized for architecture it's meant to run on)

Machine Instruction Ex

`*dest = t;` stores value t, designated by dest

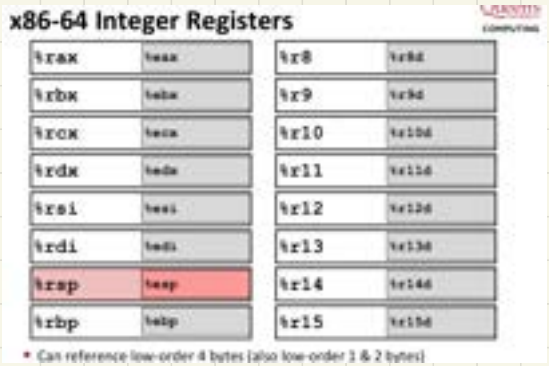
`movq %rax, (%rbx)` Highlights storing value from one register (%rax) into memory location pointed by %rbx.

`0x400597: 48 89 03`

- Object code → disassembled back into assembly code

Registers, Operands and Move

- White side shows full 64 bit size
  %rax is the accumulator register
- Gray part: shows register 32-bit, 16-bit, 8 bit.
- Backwards compatability, each 32-bit register



Moving Data

- mova: instruction used to move data from one place to another
- Could move registers, constants or memory locations

Operand Types

- Immediate: Direct numerical values prefixed with a $. Constants like 0x400
- Register: using registers like %rax or %r13 to store/retrieve values
- Memory: Refers to data in memory specified by register.
- Source/Destination. Source is where it comes from, Destination is where the data is moved to.

Operand Combinations

- Immediate → register, Same as assigning constant value to variable-e.
- Register → Register, copying one variable to another
- Memory → memory, similiar to dereferencing pointer in C
- Register → Register, similiar assigning value to a dereferenced pointer in C

Note: Memory → memory transfer is not possible, first move data from memory to register then register to another memory.

Memory Addressing Modes

- Normal Addressing, register is used to hold the adress, of the data in memory. (not data itself)
- In C, its dereferencing a pointer to access value it Points to
- Displacement, involves register, constant. Where register contains base memory adress and constant is an offset added to that adress.
- In C, its like accessing array element, base = start, displacement = index

Understanding swap()

- Registers Contain addresses, which point to a Value in memory
- %rax, (%rsi)
  - takes value at this register
  - Points to memory adress of this and Puts value there
- swap() uses registers to temporarily hold values in memory adresses given by %rdi and %rsi

Lecture 10- Machine Fundamentals

- adresses should always be 64 bits
- rax is for returning values
- General Format: D(Rb, Ri, S)

D - displacement, constant valve added to adress

Rb - Base register, holds the base memory adress

Ri - Index Register, used to calculate offset from base

S - scale, multiplier used with Ri

Adress Computations

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Arithmetic and Logic Operations

- leaq Src, Dst ; Calculates an adress using given addressing mode, stores it without accessing memory
- Usefull for complex adresses, leaq (%rdi, %rsi, 2), %rax → adress = val in rdi + 2* val in rsi

Two-operand instructions:

Format	Computation		
addq	Src, Dest	Dest = Dest + Src	
subq	Src, Dest	Dest = Dest - Src	
imulq	Src, Dest	Dest = Dest * Src	
salq	Src, Dest	Dest = Dest << Src	Also called shlq
sarq	Src, Dest	Dest = Dest >> Src	Arithmetic
shrq	Src, Dest	Dest = Dest >> Src	Logical
xorq	Src, Dest	Dest = Dest ^ Src	
andq	Src, Dest	Dest = Dest & Src	
orq	Src, Dest	Dest = Dest | Src	

Watch out for argument order!

- Shift operations, multiplies register values by multiples of two.
  - imulq - multiplies two integer values
  - Salq - Shift left, multiplies a value of 2 raised to power of shift amt.

Note: Compiler transform's statements/expressions Compiler uses many instruction combinations.

Register Usage

- %rdi, %rsi, %rdx used for function arguments
- %rax, %rdx, %rcx used for storing intermediate results

Lecture 11 - Machines Control

Condition Codes

- Condition codes are special **single bit flags**, that **store the result** of **arithmetic operations**. These codes **control conditional flow**, like jumps and stuff.

Condition Code Registers

CF (carry flag) - set if there is carry from msb. Used for arithmetic
ZF (zero flag) - set if result is 0, Used for equality comparisons (x == y)
SF (Sign flag) - Set if result is negative, Used when signed arithmetic
OF (overflow flag) - Set if two's compliment arithmetic, signed arithmetic overflow detection

Note: Condition codes are automatically updated after an operation like addq or subq

Explicitly Setting Condition Codes

- cmpq - compares two values by seeing if (a-b) == 0 `cmpq %rsi, %rdi` (rdi - rsi)
- testq - Computes bitwise AND (A & B) `testq %rax, %rax`

Reading Condition Codes

1) Setting single byte in register
2) Conditional branching (jX instructions)
3) Conditional moves (set X instructions)

SetX Instructions

- SetX instructions store condition results into a register
- only affects lowest byte of a register

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF) &~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF) | ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

EX

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

if x > y then %al = 1, else = 0

Conditional Branches (jx instructions)

- Conditional jumps that execute based on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF) & ~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) | ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)

Ex

```
cmpq %rsi, %rdi     # compare x, y
jg greater          # jump if x > y
```

- JLE jumps if $x <= y$

Loops

- Loops in Assembly language control repeated execution of code using conditional branches (jx instructions)

1) Do while loop

- do while loop executes at least once before checking condition

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x) goto loop;
  return result;
}
```

```
  movl   $0, %rax    #  result = 0
.L2:                 #  loop:
  movq   %rdi, %rdx  #  copy x to rdx
  andl   $1, %rdx    #  t = x & 0x1
  addq   %rdx, %rax  #  result += t
  shrq   %rdi        #  x >>= 1
  jne    .L2         #  if (x) goto loop
  rep; ret
```

Loop

t = x & 1

- Key point: always runs atleast once before if(x) go to loop.

2) While Loops

- while loops check the condition before running the loop

- Uses jump to middle translation

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

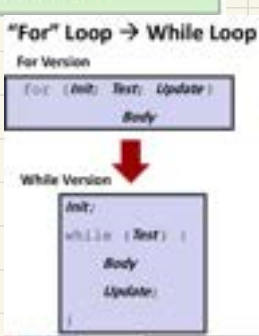
Jump to Middle

```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
loop:
  result += x & 0x1;
  x >>= 1;
test:
  if(x) goto loop;
  return result;
}
```

3) For Loops

- When you know the amt of times, you want to loop.

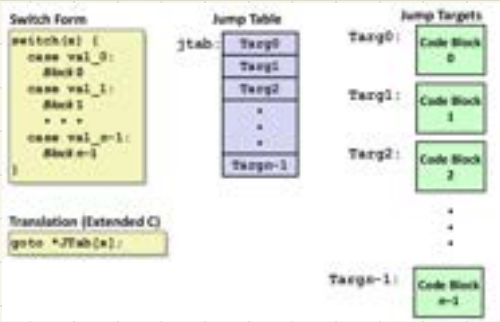
for(initialize, test, update)



Switch Statements

- allow branching execution based on a variable's value
- In assembly they are implemented using jump table
- large switch statements use a jump table

- Multiple case labels → diff case, same execution
- Fall through behaviour → next case
- Missing cases → unreachable?



Approach

1) looks up jump table using x as index
2) Jumps directly to case block
3) Executes code for that case

- use jmp * to execute switch case
- Jump table stores addresses of case labels
  - Base address
  - Direct jmp adress
  - Indirect jmp *

- Case 1: w = y * z

```
.L3:
    movq    %rsi, %rax      # y
    imulq   %rdx, %rax      # y * z
    ret
```

- Cases can have identical behaviour, sharing assembly code
- Each case corresponds to a .L2

Lecture 12 Machine Procedures

Mechanisms in Procedures

- Passing control - moving from procedure to procedure
- Passing data - arguments / return values
- memory management - allocate / deallocate memory during execution
- Machine instruction implementation - only uses necessary mechanisms from x86-64

Stack Structure

- Stack grows towards lower memory addresses
- RSP register points to top of stack (lowest numerical address)
- when data is pushed onto stack, rsp is decremented to make room for data at top of stack. → Push, Src
- when data is popped it reads data where rsp points to then rsp is incremented → POPa Dest

Passing Control

C-Code

- Multistore function computes multiplication of x/y using mult 2 then stores results in dest pointer

Assembly Code:

Push/POP commands save and restore rbx value

Calla 400550 <mult2> calls mult 2 function places return adress then jumps

- Calla decreases stack pointer to make space
- Caller - prepares function call by pushing to stack
- Callee - executes stack correctly to ensure return adress is preserved

Procedure Data Flow

- Registers have first 6 args passed
- return values stored in %rax
- Stack used for beyond first 6

1) arguments set up in registers
2) mult is called with 2 registers returns in rax
3) moves result from rax to memory located by original pointer

Stack Discipline

- manage data for limitid time
- Each Procedure makes new Stack frame

to allow space for its arguments, local variables, pointers

Call Chain Examples

- Yoo() → who() → amI()
- important for recursive functions

Stack Frames

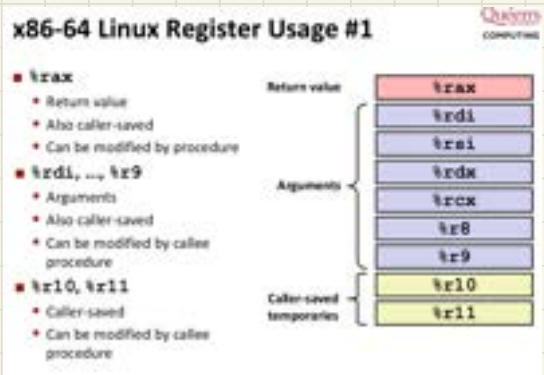
- Contents of Stack Frames:
  - return information, return address
  - local storage, Procedures using local vars' are stored here
  - Temporary space, used for temp calculations
- Allocation: When a procedure is called new stack frame is made includes space for all of the above (Push by Call)
- De allocation: Stack frame is removed
- Note: In nested procedure calls, a new stack frame is created for each function. %rsp and %rbp gets repointed.
- When procedures return, the stack is deallocated
  - ret uses return function jump back to caller

Typical x86-64 Stack Frame

- Current Stack Frame: Contains local variable, parameters for function
- Caller Stack Frame: Consists of return adress and arguments for the function

Register Saving Conventions

- Caller is the function making the call, Callee is one being called
- registers can be for temp storage
- Caller saved registers - caller expects may be modified by the callee and saves them if needed before call
- Callee saved registers - Callee is responsible for saving/restoring



% rsp is special as it points to top of stack.

Caller saved (volatile)
Callers responsibility to save

non volatile
must be saved restored

Illustration of Recursion

* Results return in %rax

- return (x & 1) + pcount_r(x>>1)

adds the lsb of x to result of a recursive recursive call with x shifted right by 1.

- recursion unwinds, modifies %rax for return vals
- Functions cleans up stack, returns total count of 1 bits

Summary

- Each call has private storage on stack including saved register/local variables
- Register saving conventions, crucial to ensure no overwriting.
- Call and return disciplice, LIFO, if P calls Q, then Q must complete and return before P
- Mutual recursion.

Lecture 13 - Machine Arrays

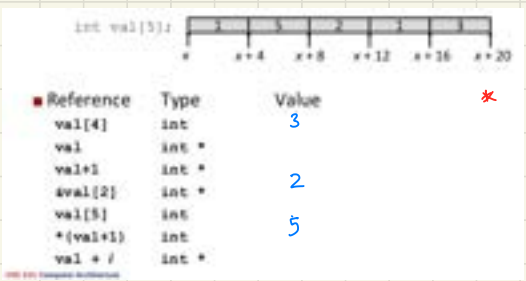
One-Dimensional Arrays

- Sequence of elements of same type, stored in contigously memory
- 1-D array `T A[L]`, consists of of a series of data type T, length L
- Memory allocation would be `L * sizeof(T)` bytes.

Ex

Char String[12] - 12 bytes
int val[5] - spans 20 bytes
double a[3] - uses 24 bytes

- Indentifier of an array could be used as pointer to first element of array. Base adress: A[0]



- typedef int zip_dig[5] - defines new type reps array of 5 ints
- zip_dig cmu = {1, 5, 3, 1, 2}; initializes array of type "zip_dig".

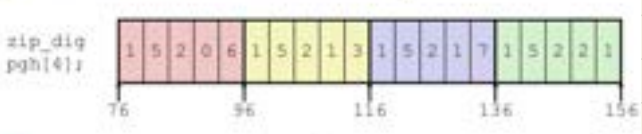
- Register %rdi contains starting address of array
- Register %rsi contains array index

- Array loops have incremental each element. Uses jmps to add to vars and compares each iteration.

Multi-Dimensional Array

- Declaration: `T A[R][C];`
  - R Rows, C columns
  - Type T elements require K bytes
  - Array Size: R * C * K bytes
  - all elements in row, stored contigously

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- "zip_dig pgh[4]" equivalent to "int pgh[4][5]"
  - Variable pgh: array of 4 elements, allocated contiguously
  - Each element is an array of 5 int's, allocated contiguously

Accessing Elements

- To access rows A[i]
- Starting address A + i*(C*K)
  Based on row index and size of each row
- Address Calculation: A + (i*C + j)*K for [i][j]

- For nxn matrixes, can handle easier element access, dynamic resizing
- Fixed 16 by 16 matrix
  a[i][j] has memory address
  A + i*(C*K) + j*K, where C=16, K=4

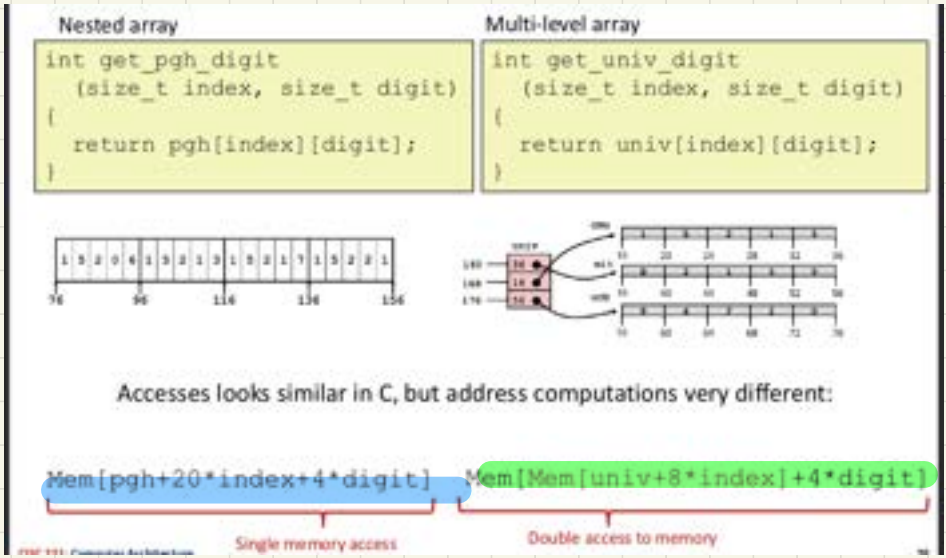
Multi-Level Arrays

- Pointers that point to other arrays
- dynamic/complex ds

```
int *univ[UCount] = {mit, cmu, ucb};
```

- each element in univ points to another array

- Memory Access: involves many dereferencing steps
  1) access Pointer from base array
  2) use Pointer to access actual integer



Quiz 2 Prep

- Callee saved registers: %rbx, %rbp, %r12, r15 (must save/restore if modified)
- Caller saved registers: %rax, %rcx, %rdx. (caller assumes will be altered by callee)
- Stack grows from higher memory addresses to lower
- addq → increases stack pointer, but moves it up, shrinking overall stack
- subq → decreases stack pointer, moves to lower adress, means stack expansion
- leaq is used to compute memory addresses

Q2 Stuff

Callee Saved: %rbx, %rbp, r12, r15

Caller Saved: %rax, %rcx, %rdx

Key adressing modes

$value → value

%register → value @ reg

(%register) → value @ memory adress → value

4(%rax) → adds 4 to value in rax, then use memory adress to find result

(%base, %index, scale) → base + index * scale gives address then find value

displacement(%base, %index, scale) → address is base + index *scale + displacement, use adress to get value

---

1)

movq 1(%rcx, %rdx, 8), %rdi

movq = move 8 bytes from memory to register

1(%rcx, %rdx, 8)

= %rcx + 1 + (%rdx * 8)

= 0xFF + 1 + (0x01 * 8)

= 0xFF + 1 + 0x08

= 0x108 → 0x7011, dest → %rdi

2) addq %rdx, 6(%rax, %rsi)

- adds 0x01 to →

6(%rax, %rsi)

= 6(0x110, 0x0A)

= 6 + 0x110 + 0x0A

= 0x120 ⟶ add %rdx

Dest: 0x120, 0x220A + 0x01 = 0x220B

3) subq -16(%rax), %rcx

dest: %rcx

= %rax - 16

= 0x110 - 16

= 0x100 → 0x0055

%rcx - 0x0055

0xFF - 0x0055 = 0xAA

4) leaq 0x30(%rax, %rdx, 8), %rsi

0x110 + 0x01 * 8

0x30(0x118)

dest=rsi, 0x30 + 0x118

= 0x148

5) andq $0x0F, (%rax)

dest: 0x110

perform Bitwise AND

0x0144 and 0x000F = 0x0004

memory at 0x110 is 0x0004

Q2 Practise

- if register is being modified update register value
- if memory is being modified, update memory location

movq → Loads from memory → register
addq → adds to memory or register
Subq → Subtracts from memory or register
leaq → Compute adress
andq → Bitwise AND

If Operation is in form of:

movq %register, memory address — Dest = memory address
addq memory, %register — Dest = Register

orq $0x0F, %rax - Bitwise or with rax
orq $0x0F, (%rax) - Bitwise or with value in memory of rax

1) movq %rdi, 1(%rcx, %rdx, 8)

1(%rcx, %rdx, 8) — value moved %rdi = 0x08

= 1 + (0xFF + 0x08)
= 0x108

Dest: 0x108, value: 0x08

5) orq $0x0F, 0x10(%rax)

or 0x10 + 0x110

0000 0000 0000 1111 = 0x120
0010 0010 0000 1010 0 x220A
0010 0010 0000 1111

value = 0x220F
Dest: 0x120

4) leaq 0x50(%rcx, %rdx, 8), %rsi

dest: %rsi
value: 0x50 + 0xFF + 0x01 * 8
: 0x157

Computes memory address

2) addq 6(%rax, %rsi), %rsi ← add

Dest: %rsi
value: 6(%rax, %rsi)
= 6 + 0x110 + 0x02
= 0x118
= 0x00AB + add %rsi value
↳ 0x00AB + 0x02
= 0x00AD

3) subq %rdx, -16(%rax) ← Subtract

dest: -16(%rax)
-16 + 0x110
= 0x100 contains 0x0055
0x0055 - 0x01 = 0x0064

Dest: 0x100
value: 0x0064

Memory Address	Value
0x200	0x0050
0x204	0x0040

Register	Value
%rax	0x200
%rbx	0x5

1) addq %rbz, 4(%rax)

Dest is a memory address
Content is going to be %rbx + 4(%rax)

Value = 4 + 0x200 — Dest: 0x5
= 0x204 + %rbx
↳ = 0x0040 + 0x5
= 0x0045

Q2 - Practise

Memory Address	Content
0x100	0x0030
0x104	0x0040
0x108	0x0050
0x10C	0x0060

Register	Content
%rax	0x100
%rbx	0x3
%rcx	0x4
%rdx	0x1

1) addq %rbx, (%rax)

Dest: 0x100

Value: 0x3 + 0x0030

= 0x0033

2) movq %rcx, 4(%rax)

Dest: 0x104

Value: 0x4

3) subq %rdx, 8(%rax)

Dest: 0x108

Val: 0x1

4) movq %rcx, (%rax)

Dest = 0x110

Value = 0xFF

5) addq $0x03, (%rax)

value: 0x03 + 0x110

= 0x113

Dest: 0x110

6) subq %rsi, -8(%rax)

Dest: 0x108 -8 + 0x110

= 0x108

value: 0x7011 - 0x0A

7) leaq 12(%rcx, %rdx, 4), %rdi

12 + 0xFF + 0x04

= C + 0xF + 0x04

= 0x1F

Dest: %rdi

value: 0x1F

8) andq $0xF0, (%rax)

Dest: 0x110

Perform and with

0 1111 0000

1 0001 0000

0 0000 0000

value: 0x10

1) movq 1(%rcx, %rdx, 8), %rdi

Dest: %rdi

value: 0xFF + (0x01 * 8) + 1

: (0x108)

= 0x7011

2) addq %rdx, 6(%rax, %rsi)

Dest: 0x120 0x110 + 0x0A + 6

= 0x120

Value: 0x01 + 0x220A

= 0x220B

3) subq -16(%rax), %rcx

Dest: %rcx

Value: 0x110 - 0x10

= 0x100

= 0x0055

= 0xFF - 0x0055

= 0xAA

4) leaq 0x30(%rax, %rdx, 8), %rsi

Dest: %rsi

value: 0x30 + 0x110 + 0x8

= 0x148

5) subq 0x10(%rax), %rsi

Dest: %rsi

Value: 0x10 - 110

= 0x100

= 0x0055

6) addq (%rsi), %rax

=%rax

0x02 + 0x110

= 0x112

7) movq 4(%rax, %rdx, 4), %rcx

dest: %rcx

= 0x110 + 0x04 + 4

= 0x118

= 0x00AB

Q3

mxn matrix: 1280 bytes

salq $6, %rdi , shift left %rdi (i) by 6 bits

↳ %rdi = i x 2^6 = i x 64

↳ Each row takes up 64 bytes

↳ Each row takes 64 bytes, each element has 4 bytes, $\frac{64}{4}$ = 16 is each row has 16 elements. # of cols is 16

Elements in Matrices

char A[M][N] - 1 byte

short A[M][N] - 2 byte

int A[M][N] - 4 bytes

double A[M][N] - 8 bytes

long A[M][N] - 8 bytes

movl A(%rdi, %rsi, 4), %eax , loading 4 byte (1=long) element into %eax

2560 bytes

A + (%rdi) + (%rsi * 4)

A + 64i + 4·j

%eax = A + 64i + 4j

= A + 4(16i + j)

Confirms each row has 16 cols

$\frac{1280}{16 \times 4}$ = # of rows

= 20

%rdi = i x 2^5 = 32i

B + 32i + 8j

= B + 8(4i + j)

movq B(%rdi, %rsi, 8), %rax

↳ # of bytes per elem

# of rows = $\frac{\text{memory}}{\text{cols} \times \text{elem size}}$

$N = \frac{\text{row size}}{\text{elem size}}$

= $\frac{2560}{4 \times 8}$ = 10

rows = 80

cols = 4

elem = 8

Q3 v2

1280 bytes

```
salq $6, %rdi        # %rdi = Shift 2^6 · i = 64i
movq A(%rdi, %rsi, 8), %eax   # %eax = A + 64i + 8j
                              #      = A + 8(8i + j)
ret
```

Longs are 8 bytes: $\frac{64}{8} = 8$

# of cols = 8

# of rows = 20 $\quad \frac{1280}{8 \times 8} = 20$

# of size elem = 8

Num of rows = $\frac{\text{Total memory}}{\text{row size}} = \frac{\text{Total memory}}{\#\text{num cols} \times \text{elem size}}$

%rdi = $i \times 2^5 = 32i$

%eax = $A + 32i + 4j$

$= A + 4(8i + j)$

elem size = 4

$\frac{1280}{16 \cdot 4} = 20$

Q4 - Interpreting C vs Assembly

C code

```
long loop_1(long x, long y)
{
long z = 1      # z is default set to 1   z = 1
while (x ≤ y){  # condition
  z = z*(x+y)   # z is calculated, using x and y, assigned new val
  x = x + 1     # x is incremented
}
return z;
}
```

For x = 1, y = 3

z = 1 * 4
z = 4, x = 2
z = 4 * (5)
z = 20, x = 3
z = 20 * (6)
z = 120

Note: z does not change and only is returned once

Assembly code

```
loop_1:                  # Function name
  movl $1, %eax          rax = z = 1        # moves 1 to %eax register which holds z initially
  jmp .L2                jump to check condition   # checks condition, due to while loop behaviour
.L3:                     Starting point for loop
  leaq (%rdi, %rsi), %rdx    rdx = x + y      # remember leaq just computes address
  imulq %rdx, %rax           rax = z * (x+y)  # refresher on imulq  multiplies signed numbers  # rax = rax * rdx
  addq $1, %rdi              rdi = x + 1      # increment x
.L2:
  cmpq %rsi, %rdi        Compare x and y (x-y)   # compares values in registers rsi and rdi and sees if x - y == 0
  jle .L3                Loop when x ≤ y         # if x - y ≥ 0, jump to .L3, jle means jump if less than or equal to
  ret                    returns function
```

Q.5 - More advanced Translation

C Code

```
long branch(long x, long y, long z){
    long val = x+y+z
    if ( x<5 )
      if( y<z )
        val = x*y ;
      else
        val = y*z ;
    }elseif( x>10 )
      val = x*z
    return val;
}
```

cmpq $8, %rdi
jge .L2 means If x<8 continue

order matters in your operand

Assembly Code (x in %rdi, y in %rsi, z in %rdx)

```
branch:
    leaq (%rdi,%rsi), %rax      longval=%rax = x+y   ✓
    addq %rdx, %rax             longval = %rax = x+y+z   ✓
    cmpq $5, %rdi               If x≥5 then jump to L2 or x<5 continue   ✓
*   jge .L2
    cmpq %rdx, %rsi             If y≥z, then jump to L3 or y<z continue   ✓
*   jge .L3
    movq %rdi, %rax             rax = x
    imulq %rsi, %rax            rax = x * y
    ret
.L3:
    movq %rsi, %rax             longval = y
    imulq %rdx, %rax            longval = longval * z
    ret
.L2:
    cmpq $10, %rdi              If x ≤ 10, jump to L4   x>10, continue
    jle .L4
    movq %rdi, %rax             else: longval = x
    imulq %rdx, %rax            longval = longval * z
.L4:
    ret
```

```
branch:
  leaq (%rdi,%rsi), %rax
  addq %rdx, %rax
  cmpq $10, %rdi
  jge  .L2
  cmpq %rdx, %rsi
  jge  .L3
  movq %rdi, %rax
  imulq %rsi, %rax
  ret
.L3:
  movq %rsi, %rax
  imulq %rdx, %rax
  ret
.L2:
  cmpq $15, %rdi
  jle .L4
  movq %rdi, %rax
  imulq %rdx, %rax
.L4:
```

(a in %rdi, b in %rsi, c in %rdx)

%rax = a+b ✓

%rax = %rax + c ✓

If a<10, the continue, else jump to L2 ✓

If b < c, then continue, else jump to .L3 ✓

%rax = a ✓

%rax = %rax * b ✓

return

%rax = b ✓

%rax = %rax * c ✓

return

If a > 15 (10?), continue, else jump to L4

%rax = a ✓

%rax = %rax * c ✓

return

```
long branch(long a, long b,
long c){
long val= a+b+c ;
if ( a < 10 ){
    if ( b < c )
     val= a*b;
    else
     val= b*c;
} elseif ( a > 15 )
    val= a*c ;
return val;
}
```

Assembly code
(x in %rdi, y in %rsi)

```
Loop_1:
  movl $1, %eax
  jmp .L2
L3:
  leaq(%rdi, %rsi
  imulq %rdx , %r
  addq $1, %rdi
L2:
  cmpq %rsi, %rdi
  jle .L3
  ret
```

(x in %rdi, y in %rsi)

%eax = 1

Jump to L2 (loop condition)

%rdx = x + y ✓

%rax = %rax (x+y) ✓

x = x + 1 ✓

Compare x and y

loop when x ≤ y

return

```
long Loop_1(long x, long y)
{
long z= _____;
while (_____){
z= _____;
x = _____;
}
return z;
}
```

z = 1 ; 1

(x ≤ y)

z * (x+y)

x = x+1

2 , 4

z = 1,

z = 1 * (6)
z = 6
x = 3
z = 6 * (7)
z = 42
x = 4
z = 42 * (8)
z = 336

Q3 / a4 Loops

C code

```
long Loop_2(long x, long y){
    long z = 1
    for ( x = 1 ; x≤y ; x++ ){
        z = z * (x+y)
    }
    return z
}
```

Assembly code (x in %rdi, y in %rsi)

```
%eax = 1
Compare x > y
    loop when x>y, .L2
.L3:        %rdx = x+y
            %rax = %rax * (x+y)
            x = x+1
            Condition: x ≤ y
                go back up to .L3
.L2:
ret   This is return code or the stop
      condition for the for loop.
```

Inside for loop

C Code

```
long Loop_3(long x, long y){
    long z=1
    do {
        z = z*(x+y)
        x = x+1
    } while ( x ≤ y );
    return z;
}
```

Assembly Code

```
%eax = 1
.L3:
    %rdx = x+y
    %rax = z*(x+y)
    x = x+1
    Compare x and y
        Jump to .L3
        if x ≤ y
```

Add/subtract operations

- case 1: dest is register
  - value is added/subtracted to registers content, stored back
  - FF - 55 = AA
- case 2: Dest is memory
  - Find memory, read old value, add/subtract content to it and store back result

add/sub value, register

This +/- that

1) Dest: %rdi ✓

value: 1 + 0xFF + 0x01 * 8

: 0x108 → 0x7011 ✓

2) Val. 0x01 x

Dest: 6 + 0x110 + 0x0A

= 0x120 dont add content to address → Value

=

@ 0x120 = 0x220A

Then add 0x01 to that get 0x220B

3) subw -16(%rax), %rcx

Dest: %rcx ✓

Val: -16 + 0x110

= 0x100 - 0xFF

= 0x01 x → 0x0055, FF - 55 = AA

4) leaq 0x30(%rax, %rdx, 8), %rsi

Dest: %rsi ✓

Val: 0x30 + 0x110 + 0x08

: 0x148 vs. 0x17C ✓

5) andq $0x0F, (%rax)

0x0144

0000 0000 1111

0001 0100 0100

0000 0000 0100

= 0x04 → 0x004 OK ✓

Dest: 0x110 ✓

1) dest:

Val: 0x08 ✓

1 + 0xFF + (0x01 * 8)

= 0x108

= 0x7011 ✓

Dest: 0x108

Val: 0x08

2) Dest: %rsi

Val: 6 + 0x110 + 0x02

= 0x118 ✓

= 0x00AB + 02

= 0x00AD ✓

3) Dest: -16 + 0x110

Dest: = 0x100 ✓

= 0x0055

0x0055 - 0x01

= 0x0054 ✓

4) Dest: %rsi ✓

Val: 0x50 + 0xFF + 0x08

= 0x157 ✓

5) Dest: 0x0144 + 0x10

= 0x154 x

0x120 → 0x220A

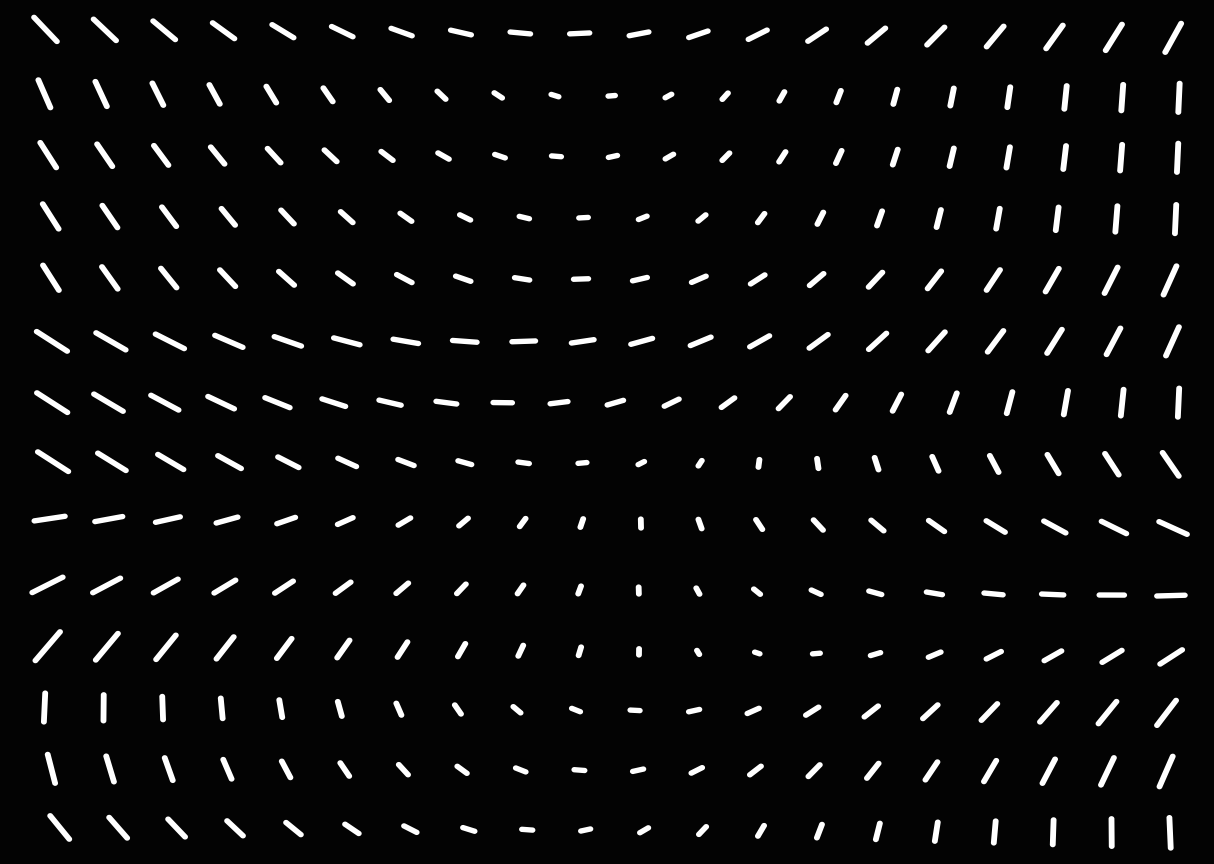
Val: 0000 0000 0000 1111

0010 0010 0000 1010

0010 0010 0000 1111

= 0x220F

Unit 3



Lecture 15 - Machine Structure

Structure

- a Struct in C is a block of memory that contains multiple fields
- Fields are ordered as declared
- Compiler determines size and positions of fields

Memory Alignment

- Aligned memory speeds up access and avoids CPU ineffeciencies

Ex

```
struct {
  char c;      1 byte   - compiler adds padding to satisfy alignment
  int i[2];    8 bytes
  double v;    8 bytes
} *p;
```

- For primitive datatype requiring K bytes the adress must be multiple of K (2,4,8)

Arrays of Structures

- When creating an array of structures, each structure must be properly aligned
- Each structure in the array is 48 bytes as compiler adds padding
- Compiler calculates offsets based on alignment, alignment affects memory layout
- Save space by reordering Struct initializers

Unions

- Union allows different variables to share same memory (but only one field at a time)

Byte Ordering

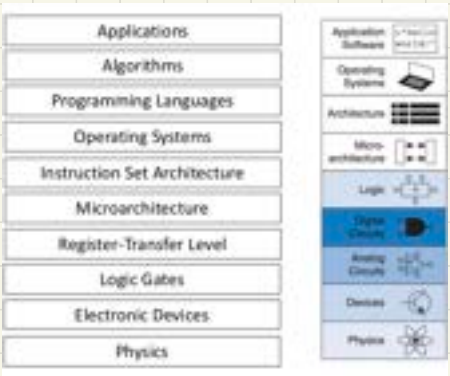
- Big Endian (MSB First) 12 34 56 78 (Sparc)
- Little Endian (LSB First) 78 56 34 12
- modern CPU's use little-endian

Floating Point Operations

- Floating Point Registers: XMM0 to XMM15 (can store 4 single-precision/2 sp floats)
- FP memory referencing, arguments passed in regular registers
- FP values passed in XMM registers

Lecture 16 - Digital logic : logic gates

General Intro



- Computers are built on multiple abstraction layers
- Highest level (software side)
- Lowest level (hardware side)
- Each layer depends on one below it.
- ISA (Instruction Set Archeticcture)
  - ↓ The set of instructions a CPU understands
- Micro Archetecure → design of actual CPU that executes those instructions

- Hardware/Software must evolve together
- Digital logic gates dictate limits of hardware design.
- as transistors shrink, we hit physical limits.
- Logic gates (And, or, Not) are foundation of binary computation
- CPU's built using millions logic gates are complex circuits
- effeciency of gates directly impact processor speed and power consumption

Digital Logic Circuits (DLC)

- DLC are built using transistors in chips
- Transistors act as a simple switch that creak logic gates (And, or, not)
- Boolean expression describe circuits function
- Complex circuits are built using networks of simple logic gates
- all digital devices are built using billions of transistors to form logic gates

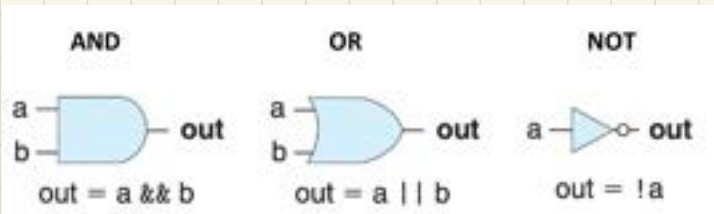
Primitive logic Functions

- logic gates take binary inputs (0 or 1) and produce binary output
- gates are represented using symbols

Truth tables -> all possible input/output combos

boolean expressions -> math version of logic

∧ - and
∨ - or
¬ - negation (not)



NAND and NOR Gates

- NAND - Not and, negates output of A and B
- NOR - Not Or, negates output of A or B
- Drawn with small circle (inverter) at output
- Note: Every other gate can be made using just NANDS

Multiple Input Gates

- Some gates take more than 2 inputs: (a, b, c)
- NOR3 gate outputs 1, when a, b, c is 0.
- more inputs -> more control over logic operations

Bit Adder

- bit adder is a digital circuit used to add binary numbers
- Full adder take 3 inputs, two outputs
- $s_1 = a_1 \oplus b_1 \oplus c_1$
- $c_{1+1} = (a_1 \wedge b_1) \vee (b_1 \wedge c_1) \vee (a_1 \wedge c_1)$
- if both bits and carry are 1, results in carry out 1.

a_1 = first bit
b_1 = second bit
c_1 = carry in (previous carry)
s_1 = Sum bit
c_{1+1} = Carry out bit

Lecture 17 - Combinational Circuits

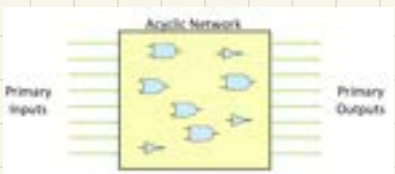
Combinational Circuits Overview

- Combinat circuits are LC, where output determind by current input values
- functional specification: describes relationship between inputs and outputs (using boolean expressions)
- timing specification: defines delays between input changes and output updates
- no memory / prev state
- can use diff combinations to get same result
- Single / line means number indicates a bus

Rules of Combinational Circuit Composition

- Circuit is combinational if:
  1) Each circuit is itself combinational
  2) Each node in circuit is either input or connected to only one output
  3) No feedback loops

Note: Loops create Sequential Circuits, which need memory elements, which is Sequential logic



How to build Combinational Circuit

1) create truth table
2) Boolean expressions based on T.T
3) Build circuit using logic gates to implement boolean functions

---

- **Terminology**
  - *Literal*: a variable or its complement (inverse)
  - *Product* (or *implicant*): the AND of one or more literals
  - *Minterm*: a product involving all inputs to the function
  - *Sum*: the OR of one or more literals
  - *Maxterm*: a sum involving all inputs to the function

EX

$Y = \overset{\text{minterm}}{m}AB + A'B$

Means A=1, B=1, or A=0, B=1

SOP Form

- way to express boolean functions using Anded minters that or OR-ed together
- write terms with output as Y=1 as minterms

EX

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

midterms: where Y=1 are

$A'B$
AB'
AB

$\therefore Y = A'B + AB' + AB$

$Y = \Sigma(m_1, m_2, m_3)$

From Logic to Circuits

- any SOP equation can be drawn as a circuit
- Steps:
  1) Draw input variables
  2) Add NOT gates for complemented variables
  3) And gates for minterms
  4) Or gate to combine all minterms.

- Note: Not always easiest circuit

Boolean Algebra Axioms

- Axioms define how binary variables behave

	Axiom		Dual	Name
A1	$B=0$ if $B\neq 1$	A1'	$B=1$ if $B\neq 0$	Binary field
A2	$\bar{0}=1$	A2'	$\bar{1}=0$	NOT
A3	$0 \cdot 0=0$	A3'	$1+1=1$	AND/OR
A4	$1 \cdot 1=1$	A4'	$0+0=0$	AND/OR
A5	$0 \cdot 1=1 \cdot 0=0$	A5'	$1+0=0+1=1$	AND/OR

↓ Allows us to simplify logical expressions

	Theorem	Dual	Name
T1	$B \cdot 1=B$	T1'	$B+0=B$	Identity
T2	$B \cdot 0=0$	T2'	$B+1=1$	Null Element
T3	$B \cdot B=B$	T3'	$B+B=B$	Idempotency
T4		$\bar{\bar{B}}=B$		Involution
T5	$B \cdot \bar{B}=0$	T5'	$B+\bar{B}=1$	Complements

De Morgan Theorem

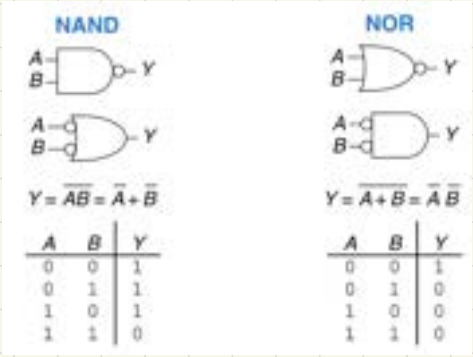
- Helps convert between and and or operations using inverters (not gates)

1.) NAND → OR with inversion

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

2.) NOR → And with inversion

$$\overline{A+B} = \bar{A} \cdot \bar{B}$$



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Simplifying Boolean Expressions

- goal is to reduce number of terms and logic gates
- SOP expression is minimized, with fewest terms
- require fewer logic gates

Ex: $\bar{A}\bar{B}C + A\bar{B}C + ABC + \bar{A}\bar{B}\bar{C}$

$= \bar{A}\bar{B}C + A\bar{B}C + ABC$

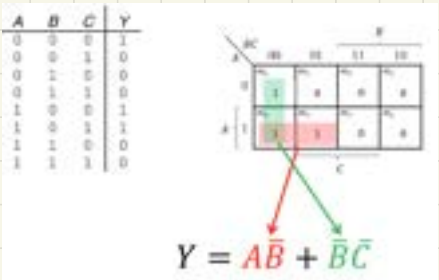
$= \bar{B}C(1) + AB(1)$

$= \bar{B}C + AB$

Karnaugh Maps

- Visual for boolean expression, best up to 4 variables
- arranges minterm in grid
- Default setup:

m0	m1	m3	m2
m4	m5	m7	m6



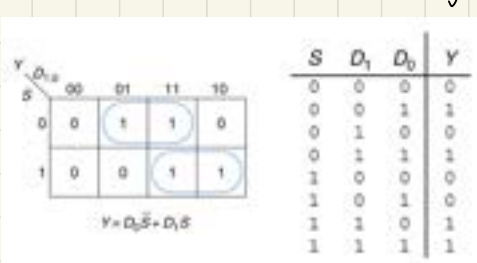
1) Find minterms, put them on grid
2) group adjacent 1's together (group size, 1,2,4)
(bigger group minimize more terms)
3) Each group is a product term

Dont Care Conditions

- Some inputs do not affect the output
- marked as x
- Simplifies logic, reduces number of logic gates
- priority circuit is a combinational circuit, determines which input has highest priority
- only highest Priority active input determines output

Multiplexors Introduction

- A multiplexor is a combinational circuit that selects one of multiple input signals and forwards it to output
- Uses a select signal (S) to decide which input to pass
- effectively control data routing

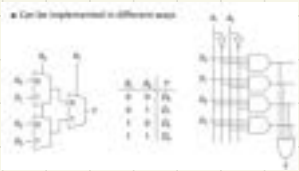


2 Data inputs D_0, D_1
1 Select line S
output y

$$Y = D_0\bar{S} + D_1S$$

Note: Not gate to invert S

4:1 Multiplexor



- 4 data inputs
- 2 select lines (S_1, S_0)
- 1 output (Y)

$$Y = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1 S_0 + D_2 S_1\bar{S}_0 + D_3 S_1 S_0$$

Decoder

- A decoder has N inputs and 2^N outputs
- it activates only one output based on binary input
  - used in address decoding in memory
  - instruction decoding in CPU's
- decoder generates minterms

Enabled Decoder

- Enable (E) input controls decoder activation
- When $E=1$, decoder works normally
- When $E=0$, all outputs are 0.

Timing

- Circuits don't change instantly, output has delay for input change
- measure timing at 50% transition point
  - Rising edge (Low → High transition)
  - Falling edge (High → Low transition)
- Propagation delay (T_{pd}) = Maximum time an output takes to stabalize
- Contamination delay (T_{cd}) = Minimum time before any output starts to change
  - to calculate T_{pd} and T_{cd} of gates looking into internals
  - Delays are caused by charging / discharging capacitance, different rise/falling delays
  - Critical Path:
    - the longest (slowest) path is the critical path
    - The critical path delay determines how fast circuit operates
  - Shortes path:
    - related to contamination delay, how fast any change in input start affecting outputs
  - Glitches:
    - Signal takes multiple paths to output, temporaly incorrect outputs
    - to solve glitches, adding delay buffers or rearranging logic gates
    - Faster processors minimize critical path delays

Lecture 18 - Digital Logic ALU

Building Combinational Circuits

- CC produce outputs only based on current inputs

Process:

1) Truth table
2) Boolean Expression
3) Logic Circuit
4) K-map Simplify
5) gate optimization & best timing (fewer gates doesn't directly mean fastest due to prop delays)

ALU (Arithmetic Logic Unit)

- type of combinational circuit that performs arithmetic (add/subtract, and, or)
- Heart of CPU

Full Adder vs 1 bit Adder

- 1 bit adder computes sum and carry out
- Full adder, adds Two inputs A and B, carry in, generates Sum and carry out.
- Circuit diagrams use xor gates for Sum.

Carry Propogate Adder

- used for multi bit numbers
- carry out of one stage propogates to next bit

- rippler carry adder (Simpler but slow)
- carry-lookahead adder (faster) → pre computes bits (generates $G_i = A_i B_i$, $P_i = A_i + B_i$ (passes)
- prefix adder (optimized)

Comparator

- determines whether two binary numbers are equal or which is larger
  - Equality: $A == B$
  - Magnitude: $A < B$

Shifter

- Shifter moves bits left/right

1) Logical Shift: Fills Shifted positions with zeroes
2) Arithmetic Shift: preserves Sign bit (MSB)
3) Rotator

- Decoder - Circuit changes code to a set of signals
- multiplexor - several inputs to share one device

Multiplier

- multiplication is done using repeated addition and shift operations

Reading ALU

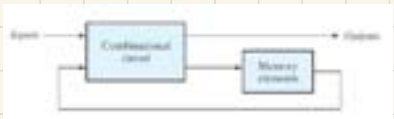
- inputs A, B
- operation selected using control signal $F_{2:0}$
- inverter suggests subtraction

Random Notes

- Latch : Level Sensitive (output changes when clk is at 1)
- Flip-Flops : Edge Triggered (output only changes on a rising edge of clk)

Lecture 19 - Sequential Logic

- Output depends on both current inputs and past inputs
- Uses some kind of memory, sequence of inputs → sequence of outputs



Synchronous data processing

- CLC operations are asynchronous
- Synchronize data transfer/processing by timed registers → ensures stabilized input to CLC
- Clock period should be greater than th CLC: → ensure valid outputs

$T_c > T_{pd}$

$\text{Period} = T_c = \frac{1}{\text{Freaquency}}$

- updates only on the clock edge

Bi-Stable Devices

- bi-stable device has two stable states, switches between them
- remains in one state until external force/energy applied to change it

Cross Coupled Inverter Pair

- CCIP is an electrical circuit exhibits bi-stability. (Part of memory circuits)
- Two outputs q and $\bar{q}$ (output of one is input to other)

SR Latch

- An SR (Set-Reset) Latch is bi stable circuit used to store single bit.
- Two inputs S(set), R(Reset). Controls q and $\bar{q}$
- Built using Cross coupled NOR gates
- Avoid state is (S=1, R=1) causes q, $\bar{q}$ to be 0

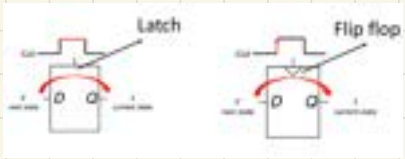
D-Latch

- D latch is modified SR Latch that eliminates invalid States
- Takes in Data (0/1) input and clk input, Controls when value stored
- latch is level sensitive, Changes Stat when signal is High

Level Trigger vs Edge Trigger

- Level triggered: Enable is High, latch continously updates
- Edge triggered: latch only changes, when clock signal transitions
  - positive edge: 0 → 1
  - Negative edge 1 → 0

D Flip-Flop



- Stores single bit of information
- has data and clk input
- Unlike d-latch, D flip flop updates, only on rising edge of clock
- D flip flop? – edge triggered, SI-18
- Commonly used in State machines

Register

- Collection of N flip-flops (share common clk)
- Registers store multi-bit data update Simultaneously
- Key components of Sequential circuits

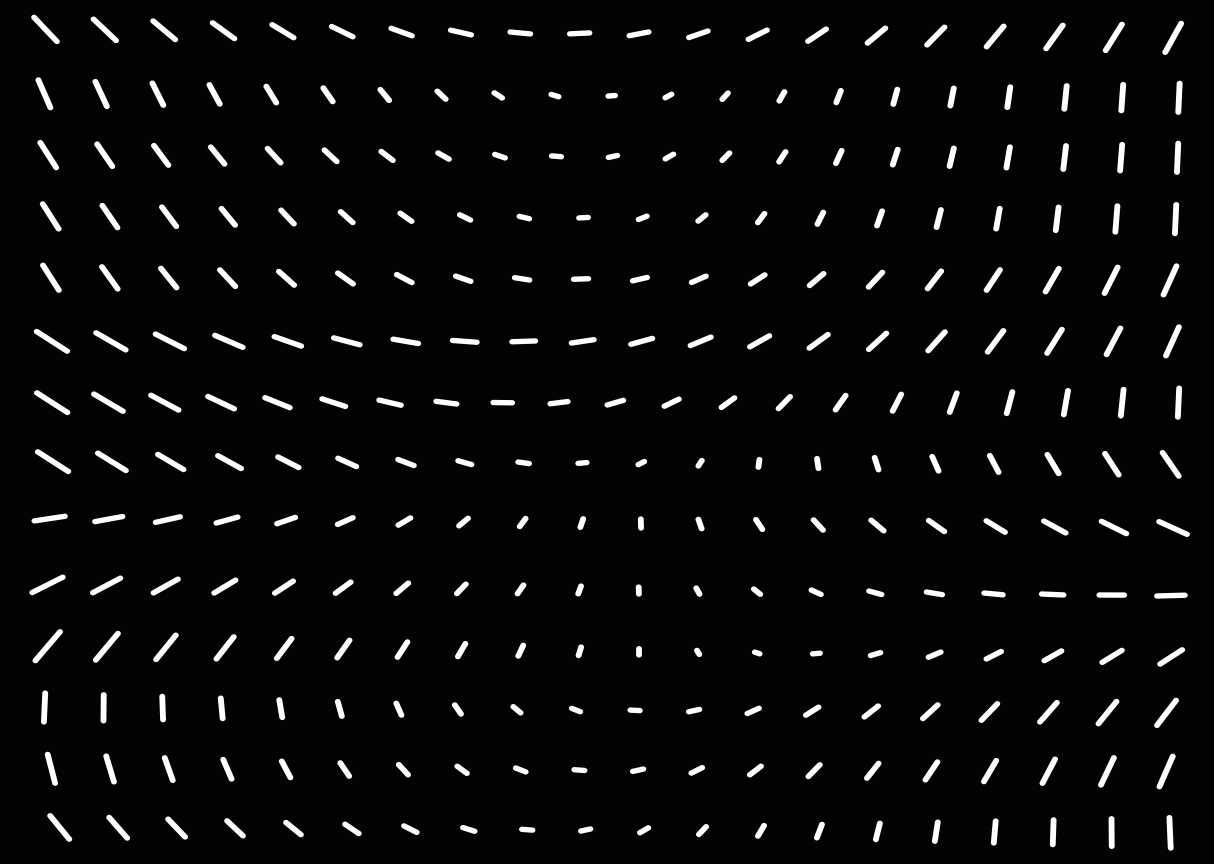
Counter

- Binary counter is Sequential circuit for Counting
- clk input, reset input, N bit outputs

Shift Register

- Stores multiple bits and shifts through data
- on rising edge, new bit shifts in all prev bits move forward

Practise Problems



Please Contact me for more info