# CS 61C:
# Great Ideas in Computer Architecture

## Lecture 2: *C Language*

Bernhard Boser & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

# Agenda

- **Numbers wrap-up**
- C Primer
- This is not on the exam!
- Break
- C Type declarations
- And in Conclusion, …

# Two's Complement
(8-bit example)

| Signed Decimal | Unsigned Decimal | Binary Two's Complement |
|---|---|---|
| -128 | 128 | 1 0 0 0 0 0 0 0 |
| -127 | 129 | 1 0 0 0 0 0 0 1 |
| … (+256) | … | … |
| -2 | 254 | 1 1 1 1 1 1 1 0 |
| -1 | 255 | 1 1 1 1 1 1 1 1 |
| 0 | 0 | 0 0 0 0 0 0 0 0 |
| 1 (+0) | 1 | 0 0 0 0 0 0 0 1 |
| … | … | . . . |
| 127 | 127 | 0 1 1 1 1 1 1 1 |

Note: Most significant bit (MSB) equals sign
But 2's complement is *not* sign-magnitude representation

# Unary Negation (Two's Complement)
## 4-bit Example ($-8_{ten}$ … $+7_{ten}$)

**Brute Force & Tedious**

"largest" 4-bit number + 1

$$\begin{array}{r} 16_{ten} \\ -\ 3_{ten} \\ \hline 13_{ten} \end{array} \qquad \begin{array}{r} 10000_{two} \\ -\ 0011_{two} \\ \hline 1101_{two} \end{array}$$

$$\begin{array}{r} 16_{ten} \\ -\ 13_{ten} \\ \hline 3_{ten} \end{array} \qquad \begin{array}{r} 10000_{two} \\ -\ 1101_{two} \\ \hline 0011_{two} \end{array}$$

**Clever & Elegant**

$$\begin{array}{r} 15_{ten} \\ -\ 3_{ten} \\ \hline 12_{ten} \\ +\ 1_{ten} \\ \hline 13_{ten} \end{array} \qquad \begin{array}{r} 01111_{two} \\ -\ 0011_{two} \\ \hline 1100_{two} \\ +\ 0001_{two} \\ \hline 1101_{two} \end{array}$$

*invert*

# Overflow
## 4-bit Example

**Unsigned**

$$\begin{array}{rr} 13_{ten} & 1101_{two} \\ +\ 14_{ten} & +\ 1110_{two} \\ \hline 27_{ten} & (1)1011_{two} \end{array}$$

carry-out and overflow

$$\begin{array}{rr} 7_{ten} & 0111_{two} \\ +\ 1_{ten} & +\ 0001_{two} \\ \hline 8_{ten} & (0)1000_{two} \end{array}$$

no carry-out and no overflow

***Carry-out → Overflow***

**Signed (Two's Complement)**

$$\begin{array}{rr} -3_{ten} & 1101_{two} \\ +\ -2_{ten} & +\ 1110_{two} \\ \hline -5_{ten} & (1)1011_{two} \end{array}$$

carry-out but no overflow

$$\begin{array}{rr} 7_{ten} & 0111_{two} \\ +\ 1_{ten} & +\ 0001_{two} \\ \hline 8_{ten} & (0)1000_{two} \end{array} \quad -8_{ten}$$

no carry-out but overflow

***Carry-out ↛ Overflow***

# Overflow Detection
## 4-bit Example

**Unsigned**

- Carry-out indicates overflow

**Signed (Two's Complement)**

- Overflow if
  - Signs of operands are equal

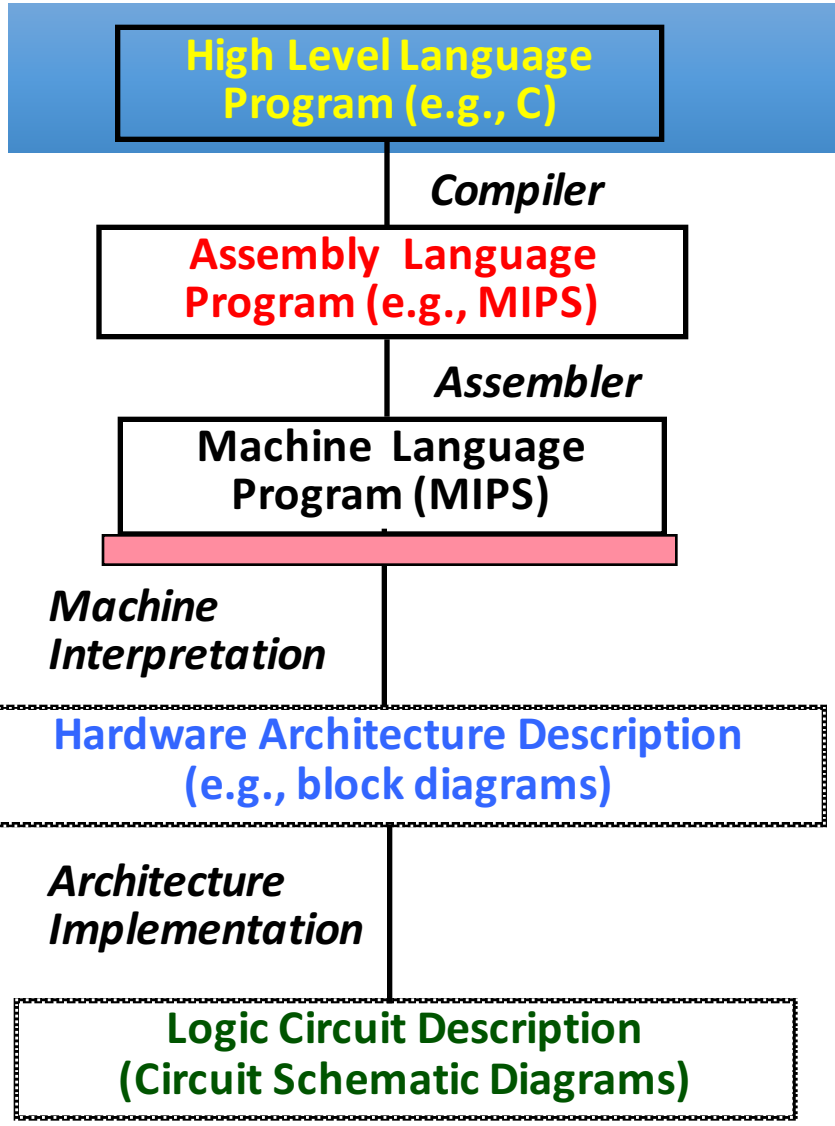
    *AND*
  - Sign of result differs from sign of operands
- No overflow when signs of operands differ

***Overflow rules depend on operands (signed vs unsigned)***

# Agenda

- Numbers wrap-up
- **C Primer**
- This is not on the exam!
- Break
- C Type declarations
- And in Conclusion, …

# Levels of Representation



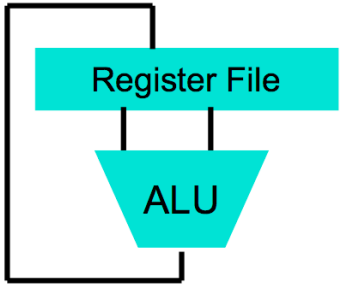

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

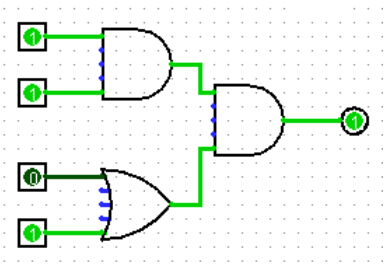
***Current Focus***

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

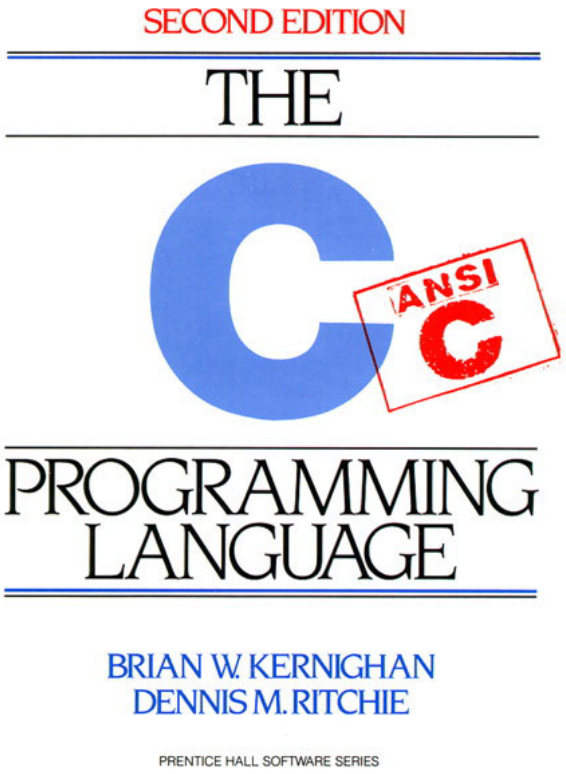
Anything can be represented as a *number*, i.e., data or instructions

```
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
```



# Introduction to C
# "The Universal Assembly Language"




- Languages used in 61C:
  - C
  - Assembly
  - Python used in two labs
- Prior programming experience helpful, e.g.
  - Java
  - C++

# Survey

Who has never written a program in

Java

*or*

C, C++, Objective C

?

# Hello World

**C**

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

---

**Java**

```java
public class L02_HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

# Compilation & Running

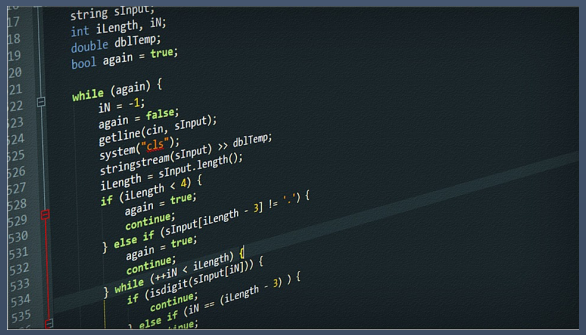
```
$ gcc HelloWorld.c
$ ./a.out
Hello World!
$
```

# Agenda

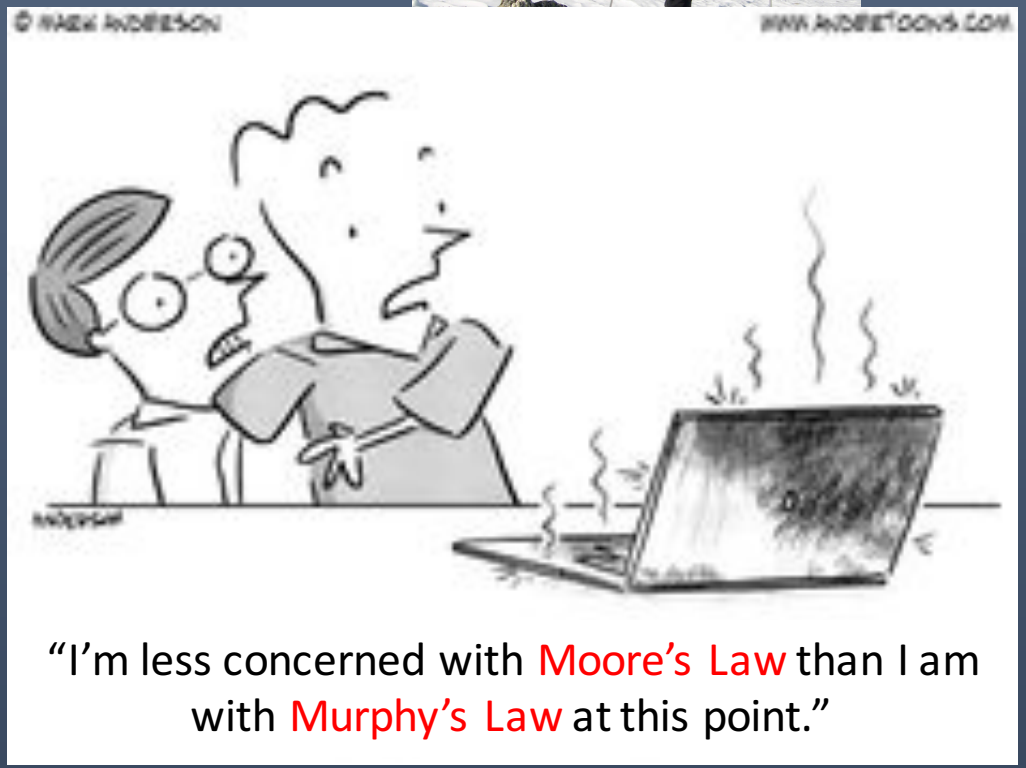
- Numbers wrap-up
- C Primer
- **This is not on the exam!**
- Break
- C Type declarations
- And in Conclusion, …

# Fun Stuff

Teardowns

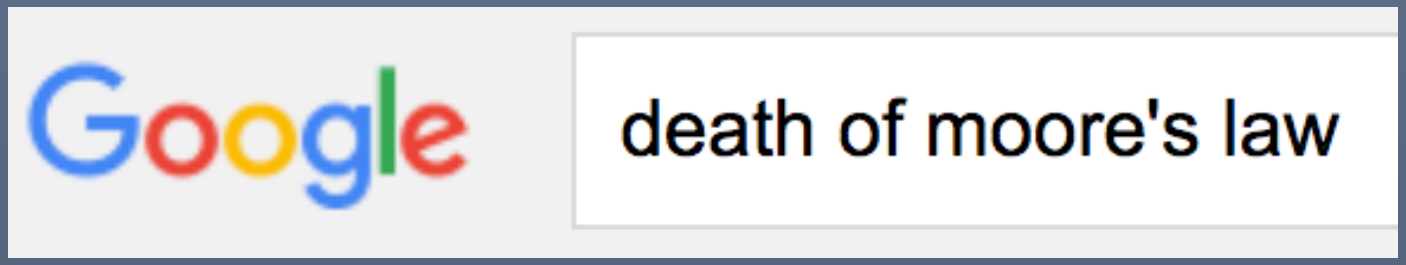



It's fun - never mind.




"I'm less concerned with Moore's Law than I am with Murphy's Law at this point."

Gossip. Murder. Suspense.




**Moore's law** has **died** at the age of 51 after an extended illness. In 1965, Intel co-founder Gordon **Moore** made an observation that the number of components in integrated circuits was doubling every 12 months or so. Feb 11, 2016

Moore's law really is dead this time | Ars Technica
arstechnica.com/information.../2016/.../**moores-law**-really-is-dead-this-time... Ars Technica

Moore's Law Is Finally Dead -- How Did This Happen? - fossBytes
fossbytes.com/**moores-law**-is-finally-dead-rest-in-peace-**moores-law**/
Feb 11, 2016 - Short Bytes: 2016 will be remembered as the year when **Moore's Law died**. This self-fulfilling law has managed to survive 51 years since Intel ...

Moore's Law – Not Dead – and Intel's Use of HPC to Keep it Alive
https://www.hpcwire.com/.../**moores-law**-not-dead-and-intels-use-of-hpc-to-k... HPCwire
Jan 11, 2016 - Editor's Note: It's taken as a given by many in HPC that **Moore's Law** is .... the industry have sounded the **death** knell for **Moore's Law**—as they ...

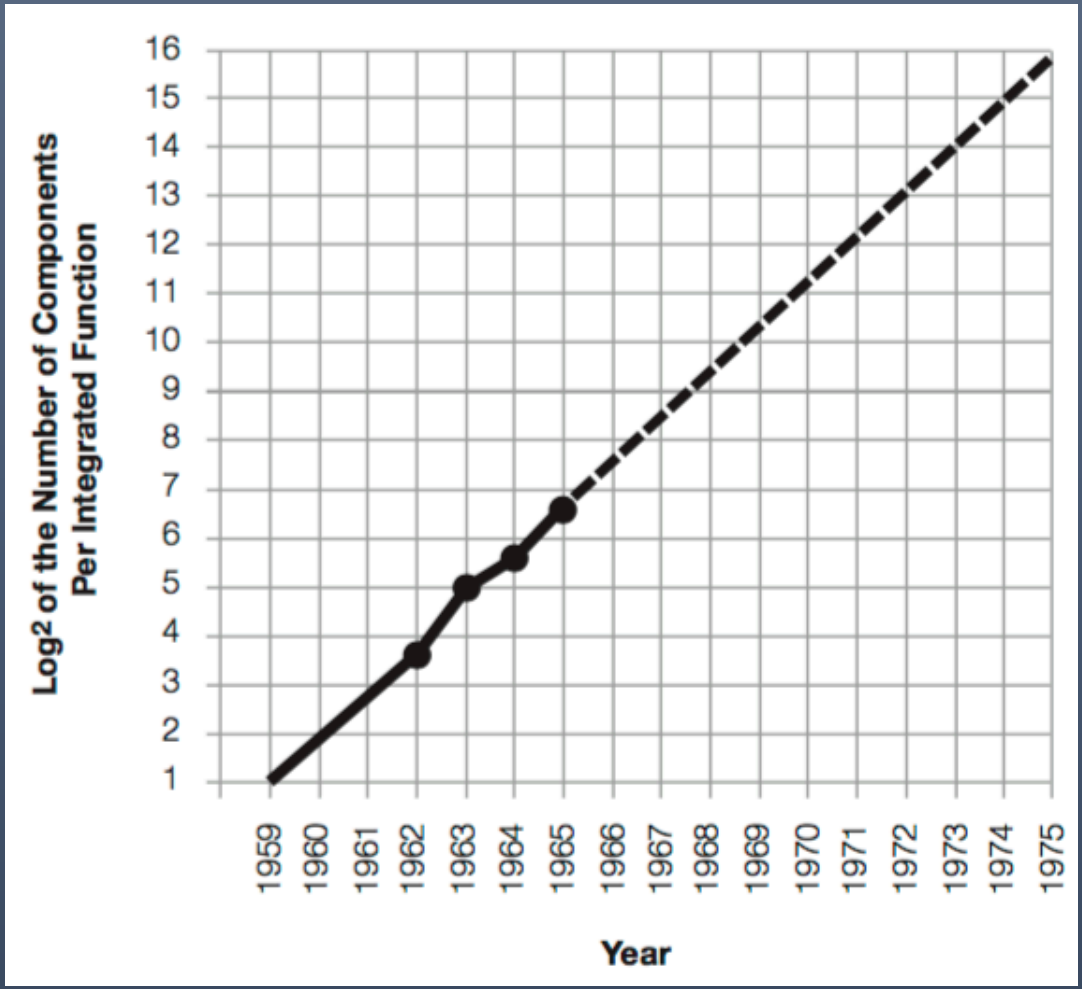
The Death of Moore's Law Will Spur Innovation - IEEE Spectrum
spectrum.ieee.org/.../the-**death-of-moores-law**-will-spur-innovation IEEE Spectrum
Mar 31, 2015 - The **Death of Moore's Law** Will Spur Innovation. As transistors stop shrinking, open-source hardware will have its day. By Andrew bunnie ...

# Who murdered Moore's Law?

- Killed what?
- Good riddance!?

- Boser's opinion 
  - Disagree and form your own!
- Not on test (dark background)
  - Legal statement: dark & bright → bright wins!
- Too nerdy?
  - Feel free to take a nap. We'll wake you when C restarts.

# Moore's Law: You mean this?




The experts look ahead

**Cramming more components onto integrated circuits**

**With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip**

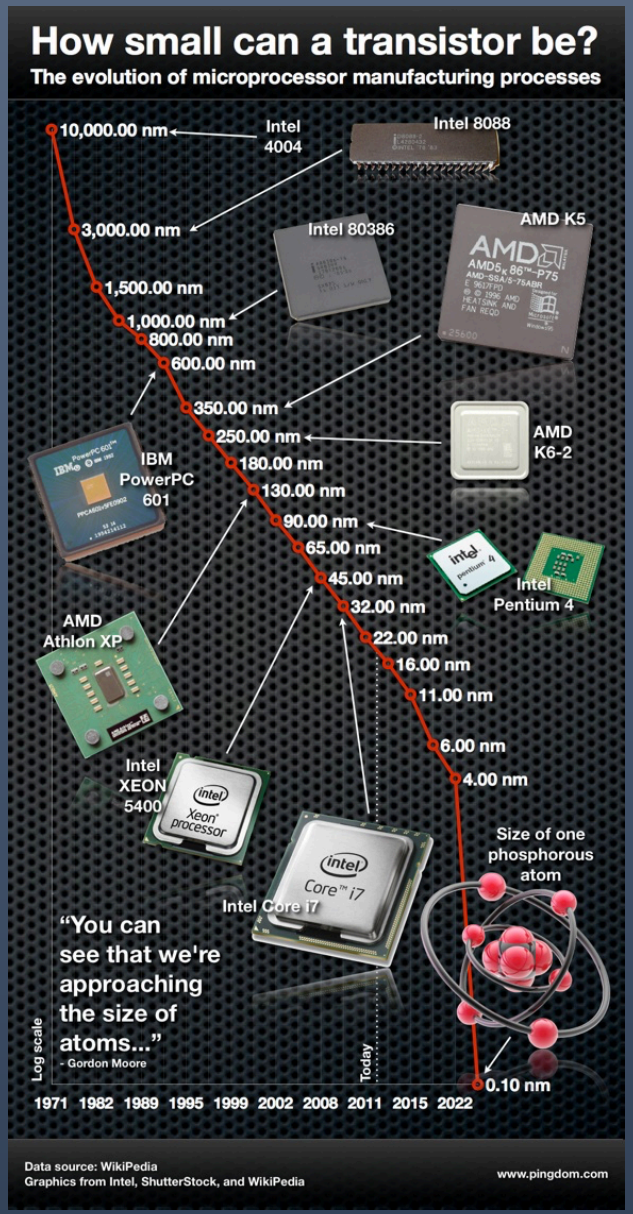
By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

**Electronics, Volume 38, Number 8, April 19, 1965**

**Popularly known as: "The number of transistors per chip doubles every two years."**

# Why is it over this time?




- More transistors per chip
  - Every transistor gets smaller
  - It was never easy
  - Splitting atoms presents entirely new challenges …
- In a few (10?) years, transistor scaling, as we know it, has to end
  - "You can see that we are approaching the size of atoms …," Gordon Moore, 2011

# But this hardly matters? Reason 1

- 1965:
  - 100 transistors max per chip
  - Barely a 4-bit adder …
- 2016:
  - > 10 Billion transistors on biggest chip (e.g. SPARC M7)
  - No chip in this room is limited by the number of transistors (?)
    - Plenty of opportunities for improving chips
    - Just adding transistors rarely cuts it
  - Few chips use the most advanced fabrication processes

# Reason 2

- This isn't even Dr. Moore's message …

**Number of transistors per chip**

**doubles every two years.**

(or 12 month or any other period)

- It is just what became popularly known as "Moore's Law"

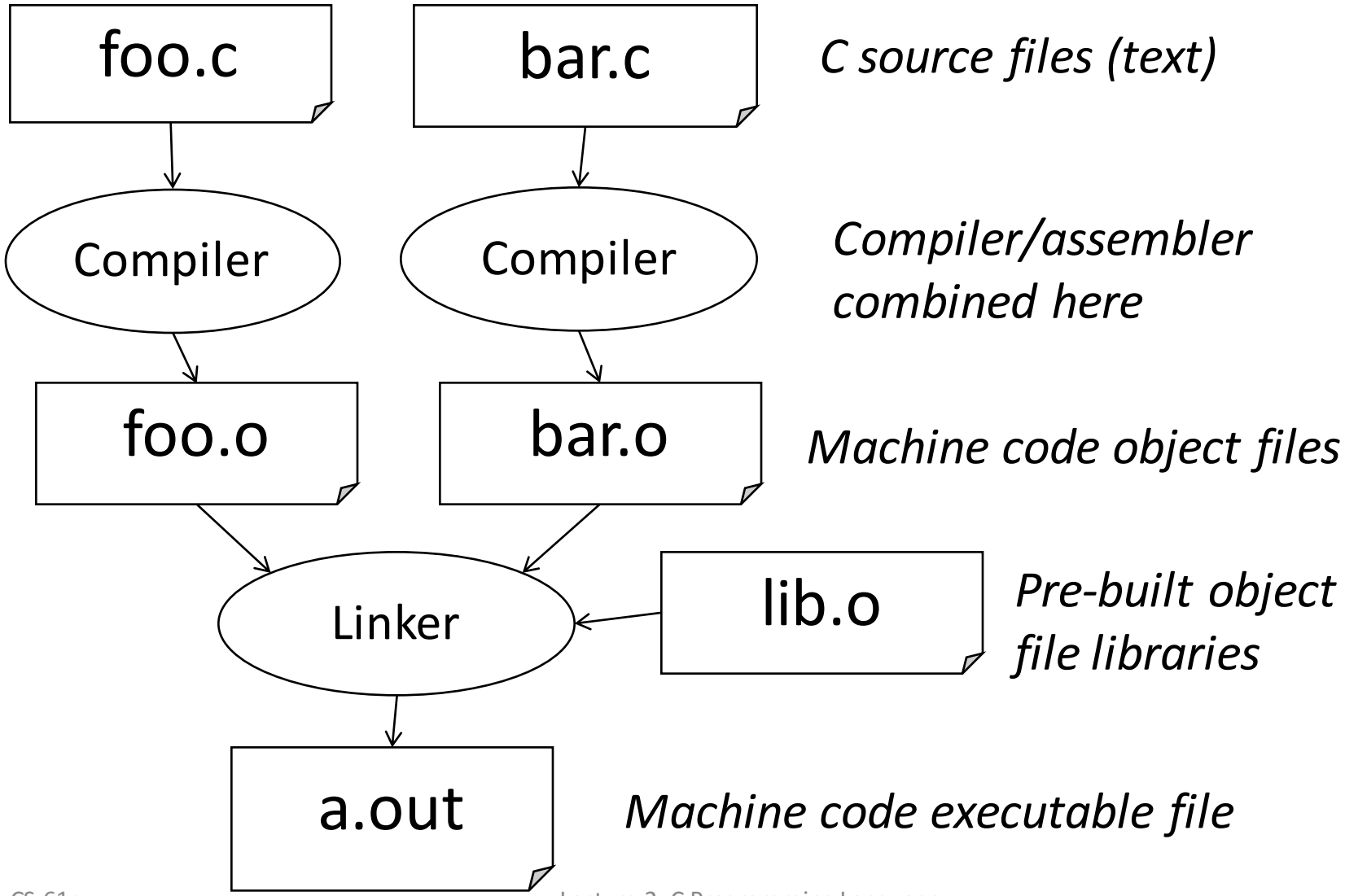
# What did Dr. Moore say, then?

Stay tuned!

# Agenda

- Numbers wrap-up
- C Primer
- This is not on the exam!
- **Compilation versus interpretation**
- Break
- C Type declarations
- And in Conclusion, …

# C Compilation Simplified Overview (more later in course)



# Compilation versus Interpretation

**C (compiled)**

- Compiler (& linker) translates source into machine language
- Machine language program is loaded by OS and directly executed by the hardware

**Python (interpreted)**

- Interpreter is written in some high level language (e.g. C) and translated into machine language
- Loaded by OS and directly executed by processor
- Interpreter reads source code (e.g. Python) and "interprets" it

# Java "Byte Code"

- Java compiler (javac) translates source to "byte code"
- "Byte code" is a particular assembly language
  - Just like i86, MIPS, ARM, …
  - Can be directly executed by appropriate machine
    - implementations exist(ed), not commercially successful
  - More typically, "byte code" is
    - interpreted on target machine (e.g. i86) by java program
    - compiled to target machine code (e.g. by JIT)
  - Program runs on any computer with a "byte code" interpreter (more or less)

# Compilation

- Excellent run-time performance:
  - Much faster than interpreted code (e.g. Python)
  - Usually faster than Java (even with JIT)
- Note: Computers only run machine code
  - Compiled application program, or
  - Interpreter (that interprets source code)

# Compilation: Disadvantages

- Compiled files, including the executable, are
  - architecture-specific, depending on processor type
    - e.g., MIPS vs. ARM
  - and the operating system
    - e.g., Windows vs. Linux
- Executable must be rebuilt on each new system
  - I.e., "porting your code" to a new architecture
- "Change → Compile → Run [repeat]" iteration cycle can be slow during development
  - Recompile only parts of program that have changed
  - Tools (e.g. make) automate this

# C Dialects

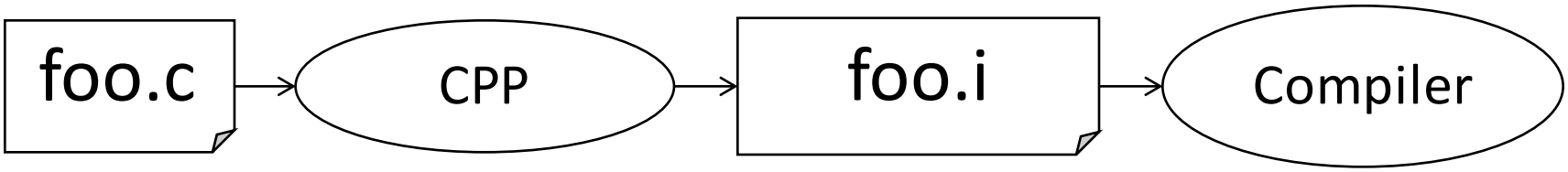
```
$ gcc x.c
```

```c
int main(void) {
    const int SZ = 5;
    int a[SZ];   // declare array
    for (int i=0; i<SZ; i++) a[i] = 0;
    return 0;
}
```

```
$ gcc –ansi –Wpedantic x.c
```

```c
#define SZ 5
int main(void) {
    int a[SZ];   /* declare array */
    int i;
    for (i=0; i<SZ; i++) a[i] = 0;
    return 0;
}
```

# C Pre-Processor (CPP)

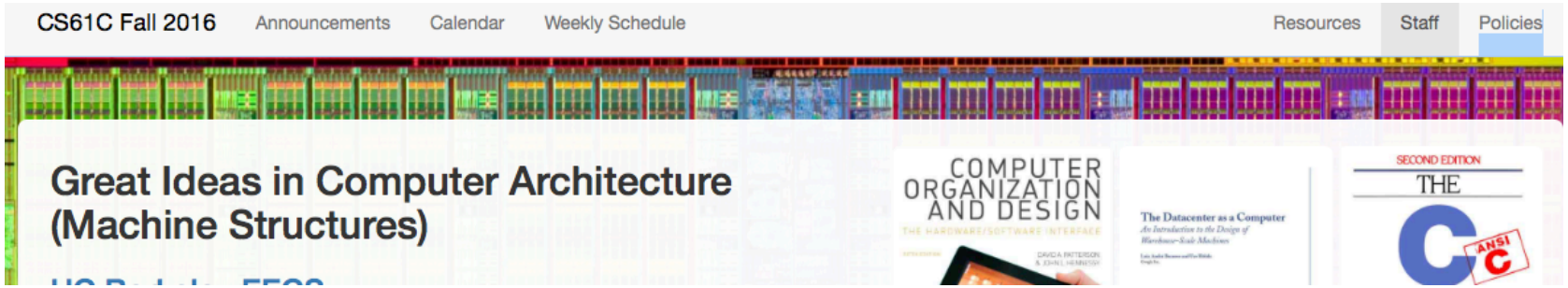



- C source files pass through macro processor, CPP, before compilation
- CPP replaces comments with a single space
- CPP commands begin with "#"
- `#include "file.h" /* Inserts file.h */`
- `#include <stdio.h> /* Loads from standard loc */`
- `#define M_PI (3.14159) /* Define constant */`
- `#if/#endif /* Conditional inclusion of text */`
- Use –save-temps option to gcc to see result of preprocessing
- Full documentation at: **`http://gcc.gnu.org/onlinedocs/cpp/`**

# Administrative

- Notify us of all known variances
  - DSP special accommodations
  - Exam conflicts
    (special accommodations can be made only in exceptional cases)

- **Notify head GSI by Tuesday, 9/6/2016**
  - see http://www-inst.eecs.berkeley.edu/~cs61c/fa16/



# Agenda

- Numbers wrap-up
- C Primer
- This is not on the exam!
- **Break**
- C Type declarations
- And in Conclusion, …

# Agenda

- Numbers wrap-up
- C Primer
- This is not on the exam!
- Break
- **C Type declarations**
- And in Conclusion, …

# Typed Variables in C

```
int a = 4;
float f = 1.38e7;
char c = 'x';
```

- Declare before use
- Type cannot change
- Like Java

| Type | Description | Examples |
|---|---|---|
| int | integers, positive or negative | 0, 82, -77, 0xAB87 |
| unsigned int | ditto, no negatives | 0, 8, 37 |
| float | (single precision) floating point | 3.2, -7.9e-10 |
| char | text character or symbol | 'x', 'F', '?' |
| double | high precision/range float | 1.3e100 |
| long | integer with more bits | 4279 |

# Constants and Enummerations in C

- Constants
  - Assigned in typed declaration, cannot change
  - E.g.
    - `const float pi = 3.1415;`
    - `const unsigned long addr = 0xaf460;`
- Enumerations

```c
#include <stdio.h>

int main() {
    typedef enum {red, green, blue} Color;
    Color pants = green;
    switch (pants) {
        case red:
            printf("red pants are hip\n");  break;
        case green:
            printf("green pants are weird\n");  break;
        default:
            printf("yet another color\n");
    }
    printf("pants = %d\n", pants);
}
```

# Integers: Python vs. Java vs. C

| Language | sizeof(int) |
|---|---|
| Python | >=32 bits (plain ints), infinite (long ints) |
| Java | 32 bits |
| C | Depends on computer; 16 or 32 or 64 |

- C: `int`
  - integer type that target processor works with most efficiently
- Only guarantee:
  - sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits
- Impacts portability between architectures

# Variable Sizes: Machine Dependent!

```c
#include <stdio.h>
int main(void) {
    printf("sizeof ... (bytes)\n");
    printf("char:          %lu\n",
           sizeof(char));
    printf("short:         %lu\n",
           sizeof(short));
    printf("int:           %lu\n",
           sizeof(int));
    printf("unsigned int:  %lu\n",
           sizeof(unsigned int));
    printf("long:          %lu\n",
           sizeof(long));
    printf("long long:     %lu\n",
           sizeof(long long));
    printf("float:         %lu\n",
           sizeof(float));
    printf("double:        %lu\n",
           sizeof(double));
}
```

```
sizeof ... (bytes)
char:           1
short:          2
int:            4
unsigned int:   4
long:           8
long long:      8
float:          4
double:         8
```

# Boolean

- No boolean datatype in C
  - Declare if you wish:

```
typedef int boolean;
const boolean false = 0;
const boolean true = 1;
```

- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (a special kind of *pointer*: more on this later)
- What evaluates to TRUE in C?
  - Anything that isn't false is true
  - Similar to Python:
    only 0's or empty sequences are false, everything else is true!

# Your Turn

```c
#include <stdio.h>

int main() {
    typedef enum { hard, easy, fun, sucks } CS61C;
    CS61C cs = sucks;

    if (cs = fun)
        printf("61C is my most exciting class\n");
    else if (cs = sucks)
        printf("61C sucks\n");
    else
        printf("Not sure about 61C\n");
}
```

| Answer | Output |
|---|---|
| A | `61C is my most exciting class` |
| B | `61C sucks` |
| C | `Not sure about 61C` |
| D | *- text not listed in A … C -* |
| E | *- no output -* |

# Functions in C

```
int number_of_people() {
  return 3;
}

void news() {
  printf("no news");
}

int sum(int x, int y) {
  return x + y;
}
```

- Like Java
- Declare return & argument types
- Void for no value returned
- Functions MUST be declared before they are used

# Uninitialized Variables

**Code**

```
#include <stdio.h>
#include <stdlib.h>

void undefined_local() {
    int x;  /* undefined */
    printf("x = %d\n", x);
}

void some_calc(int a) {
    a = a%2 ? rand() : -a;
}

int main(void) {
    for (int i=0;  i<5;  i++) {
        some_calc(i*i);
        undefined_local();
    }
}
```

**Output**

```
$ gcc test.c

$ ./a.out
x = 0
x = 16807
x = -4
x = 282475249
x = -16
```

# Struct's in C

- Struct's are structured groups of variables
- A bit like Java classes, but no methods
- E.g.

```c
#include <stdio.h>

int main(void) {
    typedef struct { int x, y; } Point;
    Point p1;
    p1.x = 0;    p1.y = 123;

    Point p2 = { 77, -8 };
    printf("p2 at (%d,%d)\n", p2.x, p2.y);
}
```

# More C …

- Lecture does not cover C completely
  - You'll still need your C reference for this course
  - K&R, *The C Programming Language*
  - & other references on the course website

- Lecture focus:
  - Pointers & Arrays
  - Memory management

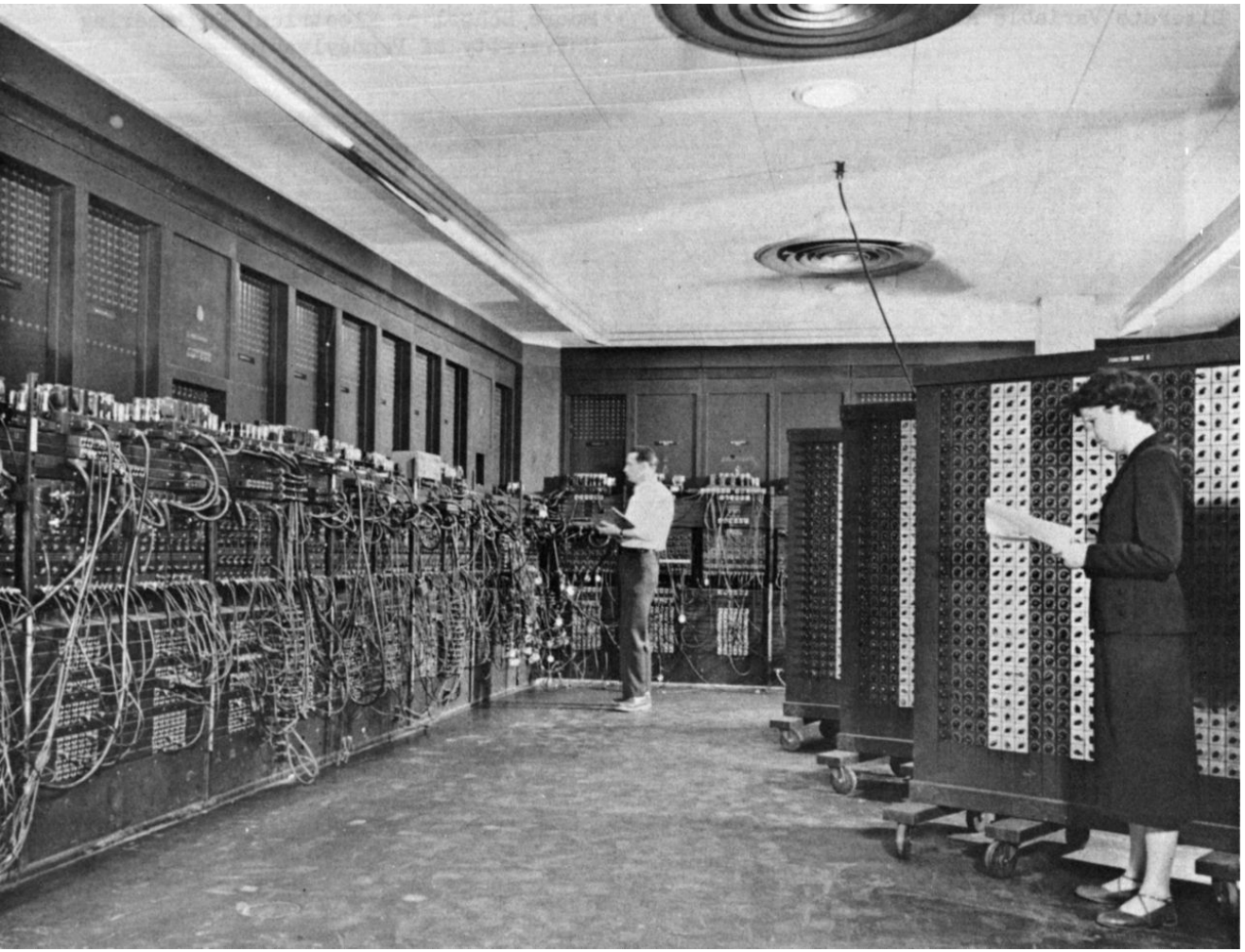
# Agenda

- Numbers wrap-up
- C Primer
- This is not on the exam!
- Break
- C Type declarations
- **And in Conclusion, …**

# Conclusion

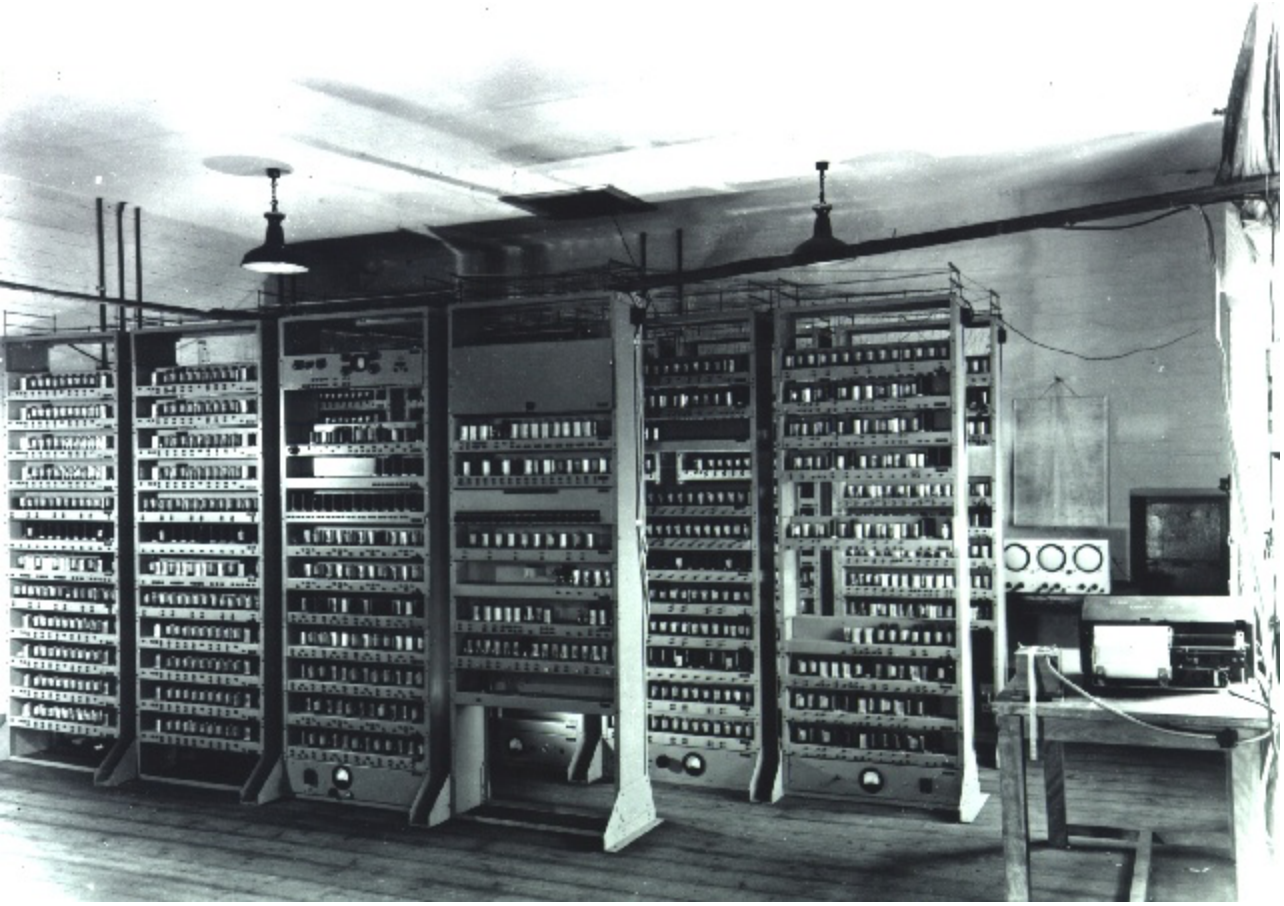
- C Programming Language
  - Popular (still!)
  - Similar to Java, but
    - no classes
    - explicit pointers (next lecture)
  - Beware
    - variables not initialized
    - variable size (# of bits) is machine & compiler dependent
- C is compiled to machine code
  - Unlike Python or Java, which are interpreted
  - Compilation is faster than interpretation

# ENIAC (U. Penn., 1946)
## First Electronic General-Purpose Computer



- 360 multiplies/sec
  - 10 decimal digits
  - ten's complement arithmetic
- Programmed with patch-cords
  - mostly women programmers!
  - no C
- Broke daily

# EDSAC (Cambridge, 1949)
## First General Stored-Program Computer



- 35 bit two's complement
- Programmed in assembler
  - paper tape
- David Wheeler (first CS PhD)
  - invented function call

# PDP-11 (1967)


- First computer to run C
  - Language invented by Dennis Ritchie at Bell Labs