# THE EVOLUTION AND ARCHITECTURE OF MODERN COMPUTERS

Professor Ken Birman
CS4414 Lecture 2

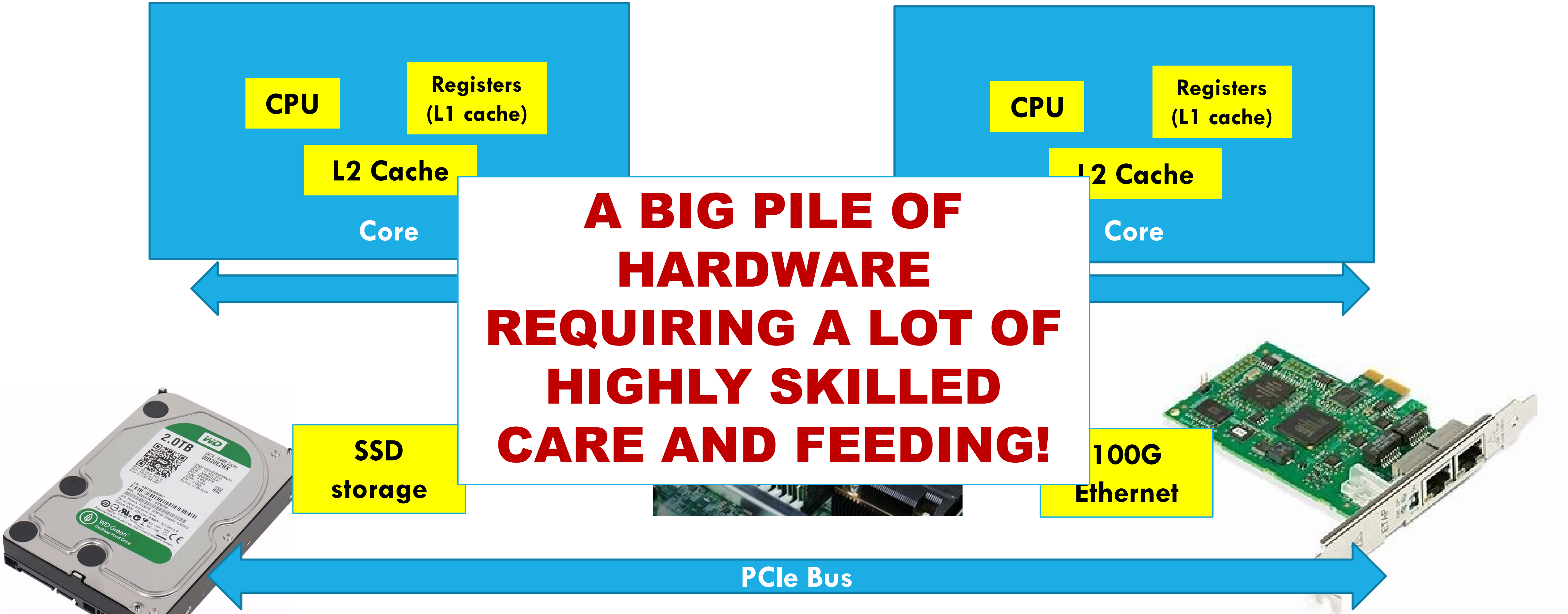
# IDEA MAP FOR TODAY

Computers are multicore NUMA machines capable of many forms of parallelism. They are extremely complex and sophisticated.

Individual CPUs don't make this NUMA dimension obvious. The whole idea is that if you don't want to know, you can ignore the presence of parallelism

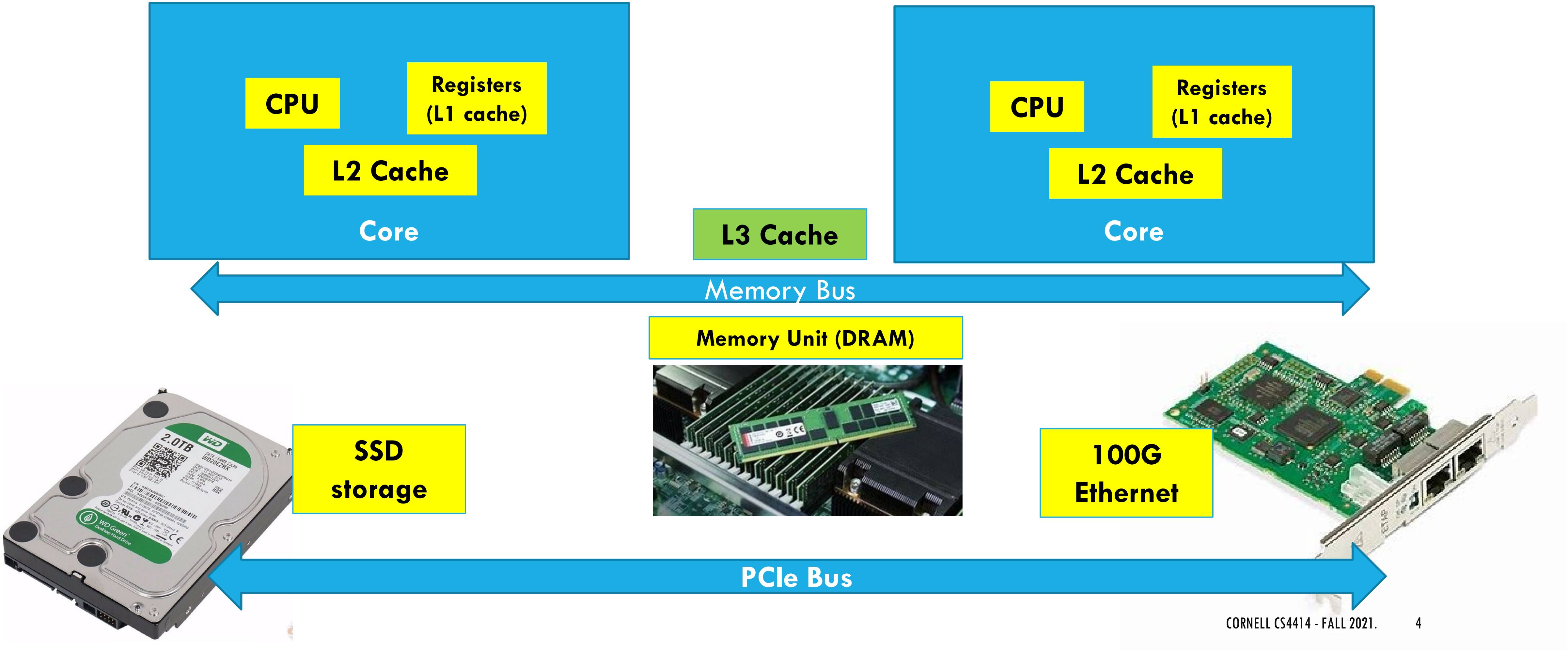
Compiled languages are translated to machine language. Understanding this mapping will allow us to make far more effective use of the machine.

# WHAT'S INSIDE? ARCHITECTURE = COMPONENTS OF A COMPUTER + OPERATING SYSTEM

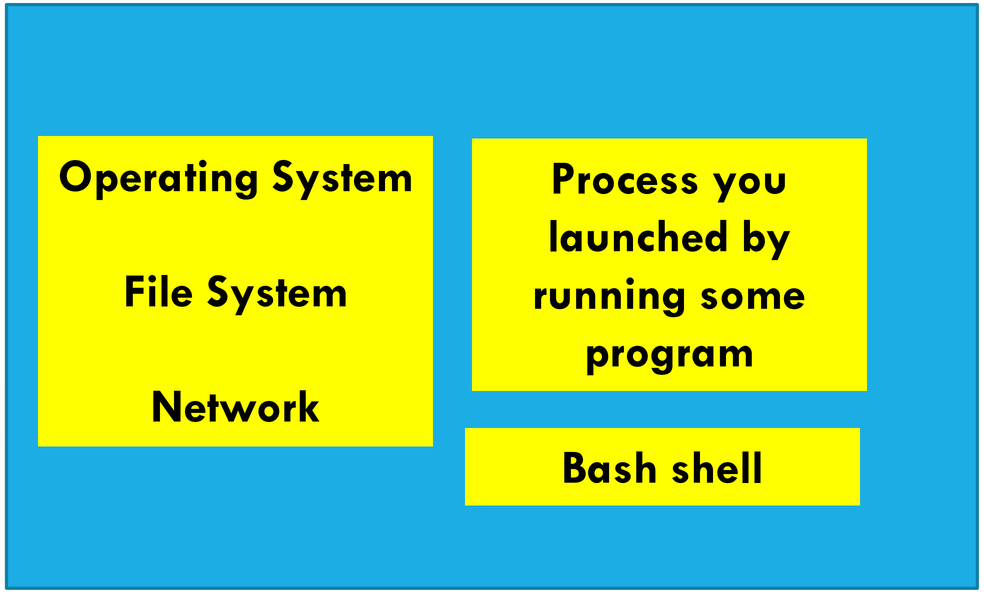



**A BIG PILE OF HARDWARE REQUIRING A LOT OF HIGHLY SKILLED CARE AND FEEDING!**

# WHAT'S INSIDE? ARCHITECTURE = COMPONENTS OF A COMPUTER + OPERATING SYSTEM



# WHAT'S INSIDE? ARCHITECTURE = COMPONENTS OF A COMPUTER + OPERATING SYSTEM




Job of the operating system (e.g. Linux) is to manage the hardware and offer easily used, efficient abstractions that hide details where feasible

# ARCHITECTURES ARE CHANGING RAPIDLY!

As an undergraduate (in the late 1970's) I programmed a DEC PDP 11/70 computer:

- A CPU (~1/2 MIPS), main memory (4MB)
- A storage device (8MB rotational magnetic disk), tape drive
- I/O devices (mostly a keyboard with a printer).

At that time this cost about $100,000

# ARCHITECTURES ARE CHANGING RAPIDLY!

Bill Gates:
"640K ought to be enough for anybody."

As … late 1970's) I programmed a DEC PDP…

- A CPU (~1/2 MIPS), main memory (4MB)
- A storage device (8MB rotational magnetic disk), tape drive
- I/O devices (mostly a keyboard with a printer).

At that time this cost about $100,000

# TODAY: MACHINE PROGRAMMING I: BASICS

History of Intel processors and architectures

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

C/C++, assembly, machine code

# MODERN COMPUTER: DELL R-740: $2,600

2 Intel Xenon chips with 28 “hyperthreaded” cores running at 1GIPS (clock rate is 3Ghz)

Up to 3 TB of memory, multiple levels of memory caches

All sorts of devices accessible directly or over the network

NVIDIA Tesla T4 GPU: adds $6,000, peaks at 269 TFLOPS

One CPU core actually runs two programs at the same time

# UTER: DELL R-740: $2,600

2 Intel Xenon chips with 28 “hyperthreaded” cores running at 1GIPS (clock rate is 3Ghz)

Up to 3 TB of memory, multiple levels of memory caches

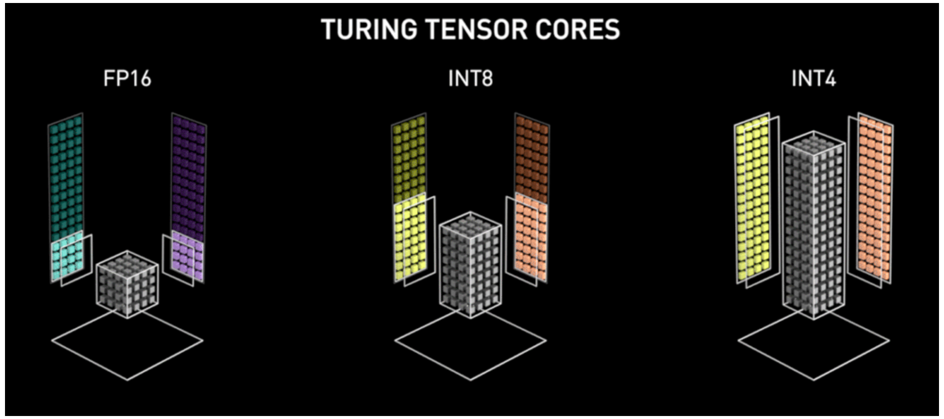
All sorts of devices accessible directly or over the network

NVIDIA Tesla T4 GPU: adds $6,000, peaks at 269 TFLOPS

# INTEL XENON

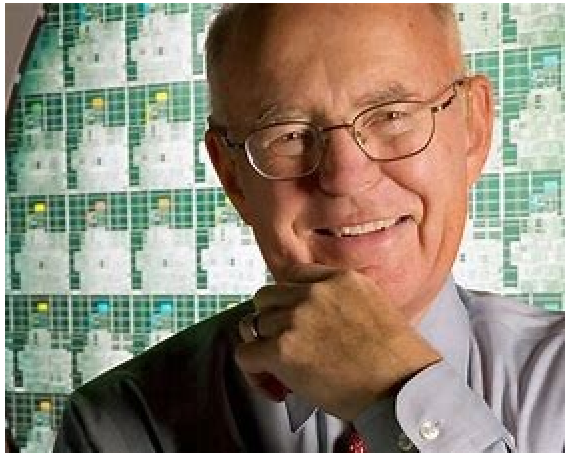
# NVIDIA TESLA


Each core is like a little computer, talking to the others over an on-chip network (the CMS)




The GPU has so many cores that a photo of the chip is pointless. Instead they draw graphics like these to help you visualize ways of using hundreds of cores to process a tensor (the "block" in the middle) in parallel!

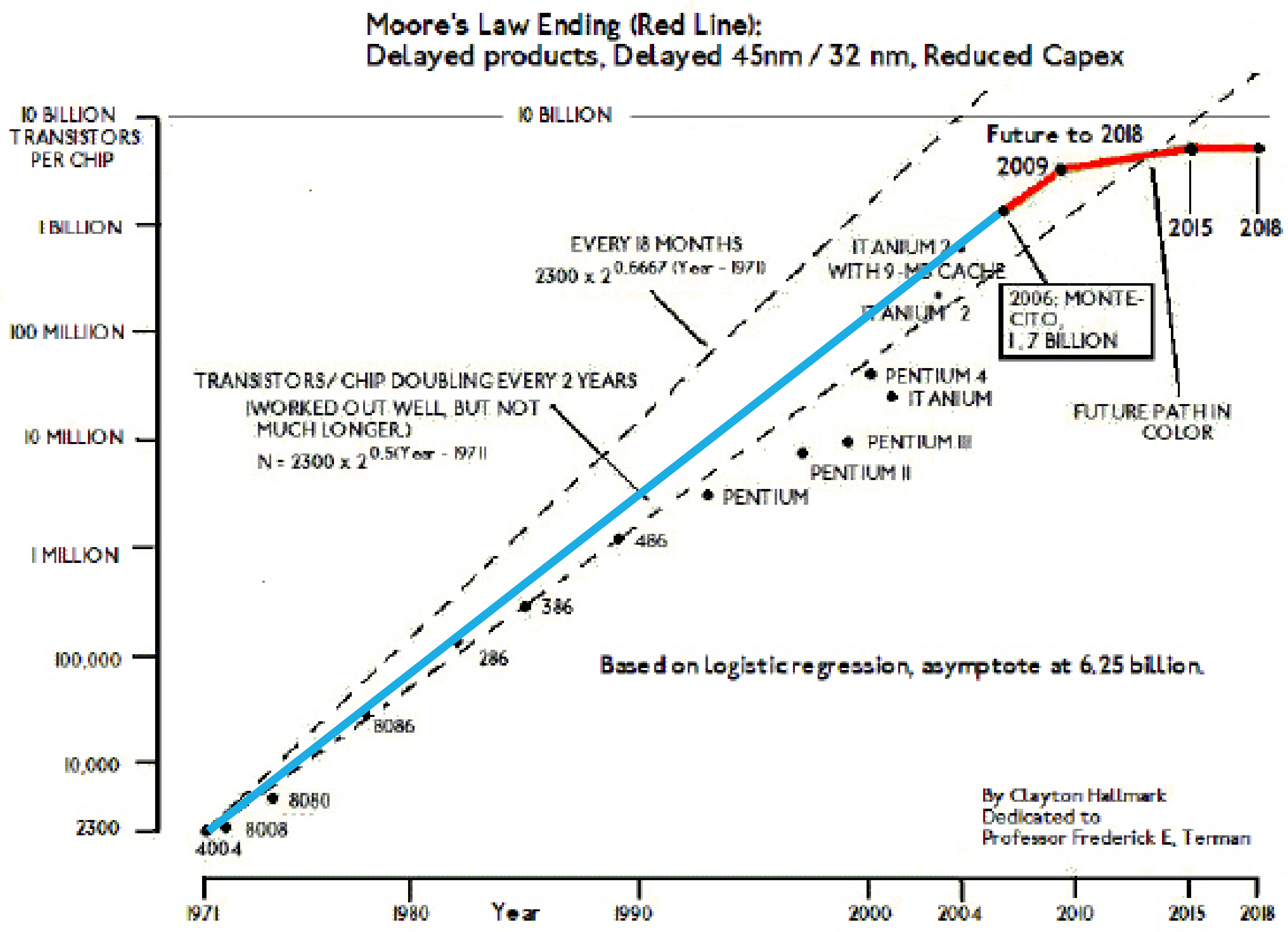
# HOW DID WE GET HERE?

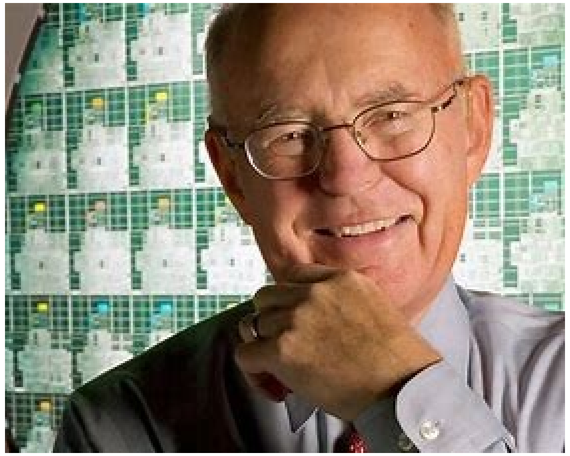


In the early years of computing, we went from machines built from distinct electronic components (earliest generations) to ones built from integrated circuits with everything on one chip.

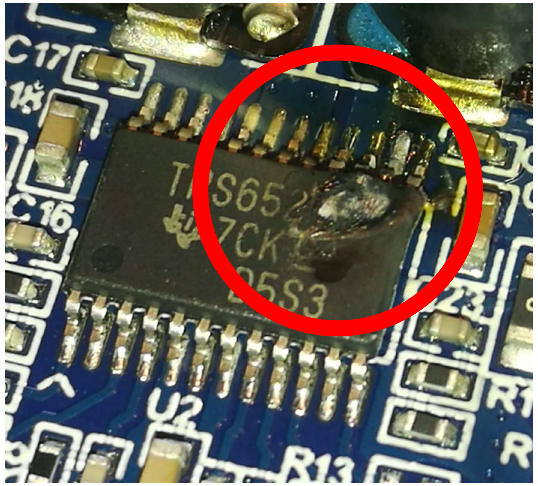
Quickly, people noticed that each new generation of computer had roughly double the capacity of the previous one and could run roughly twice as fast! Gordon Moore proposed this as a “law”.

# BUT BY 2006 MOORE'S LAW SEEMED TO BE ENDING



# WHAT ENDED MOORE'S LAW?




If you overclock your desktop this can happen…

To run a chip at higher and higher speeds, we use a faster clock rate and keep more of the circuitry busy.

Computing is a form of "work" and work generates heat… as roughly the square of the clock rate.

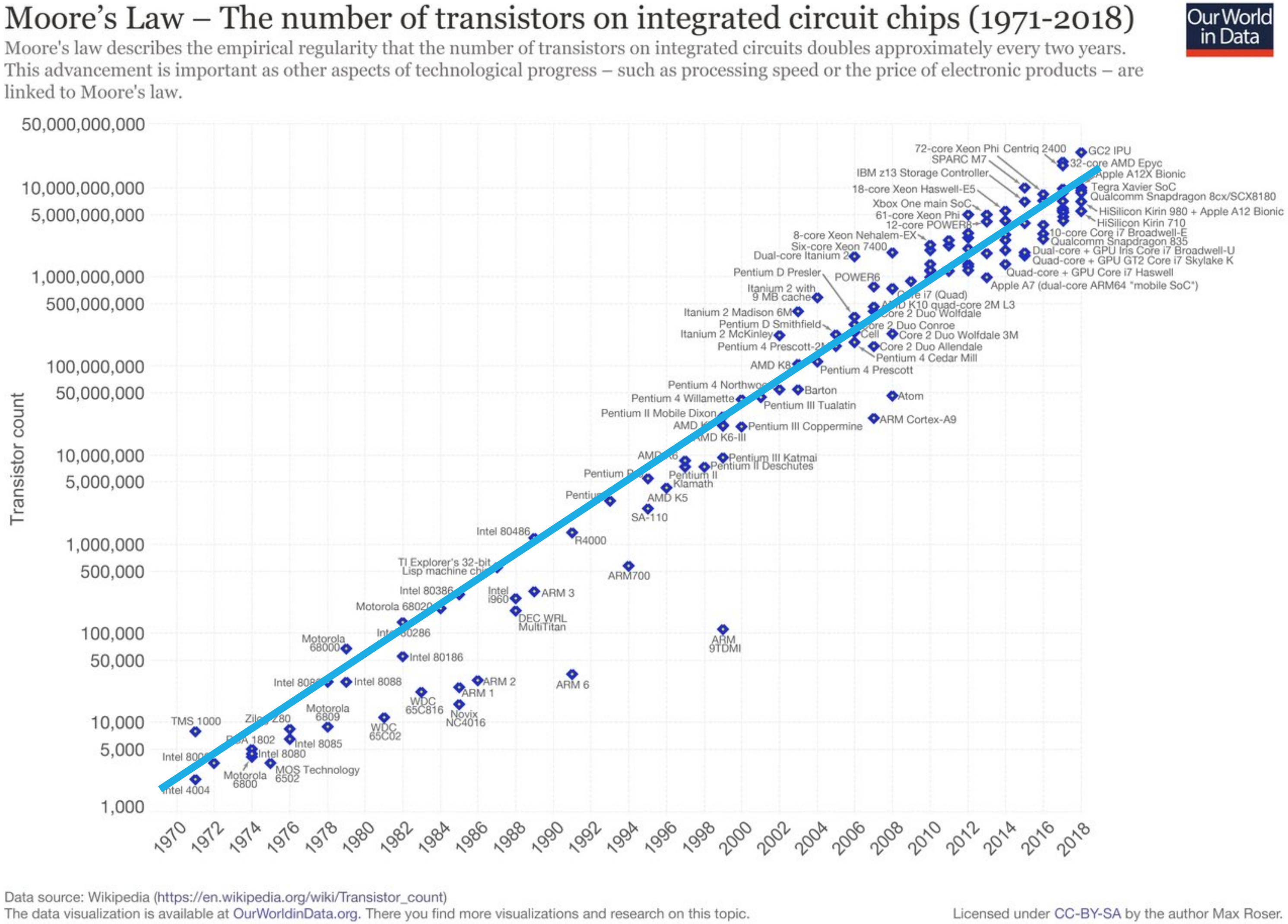
Chips began to fail. Some would (literally) melt or catch fire!

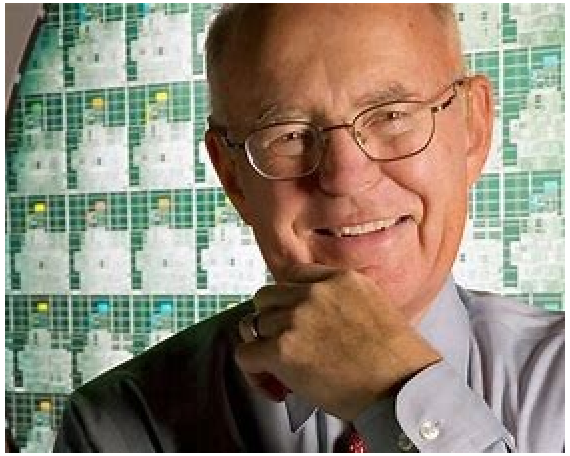
# BUT PARALLELISM SAVED US!

A new generation of computers emerged in which we ran the clocks at a somewhat lower speed (usually around 2 GHz, which corresponds to about 1 billion instructions per second), but had many CPUs in each computer.

A computer needs to have nearby memory, but applications needed access to “all” the memory. This leads to what we call a “non-uniform memory access behavior”: NUMA.

# MOORE'S LAW WITH NUMA



# … MAKING MODERN MACHINES COMPLICATED!

Prior to 2006, a good program

- Used the best algorithm: computational complexity, elegance
- Implemented it in a language like C++ that offers efficiency
- Ran on one machine

But the past decade has been disruptive! Suddenly even a single computer might have the ability to do hundreds of parallel tasks!

# THE HARDWARE SHAPES THE APPLICATION DESIGN PROCESS

We need to ask how a NUMA architecture impacts our designs.

If not all variables are equally fast to access, how can we "code" to achieve the fastest solution?

And how do we keep all of this hardware "optimally busy"?

# DEFINITIONS OF TERMS WE OFTEN USE

**Architecture:** (also ISA: instruction set architecture)
The parts of a processor design that one needs to understand for writing correct machine/assembly code

- Examples: instruction set specification, registers
- Machine Code: Byte-level programs a processor executes
- Assembly Code: Readable text representation of machine code

# DEFINITIONS OF TERMS WE OFTEN USE

**Microarchitecture: “drill down”.**

Details or implementation of the architecture

- Examples: memory or cache sizes, clock speed (frequency)

**Example ISAs:**

- Intel: x86, IA32, Itanium, x86-64
- ARM: Used in almost all mobile phones
- RISC V: New open-source ISA

# TODAY: MACHINE PROGRAMMING I: BASICS

History of Intel processors and architectures

Assembly Basics: Registers, operands, move

Arithmetic & logical operations

C/C++, assembly, machine code

# HOW A SINGLE THREAD COMPUTES

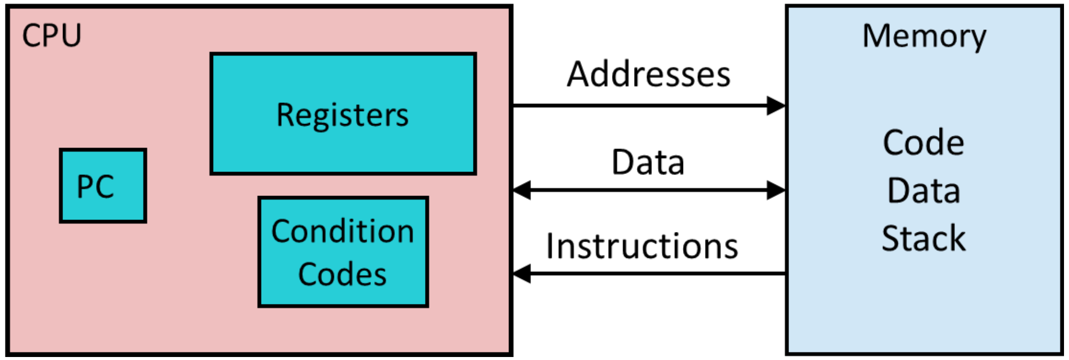
Common way to depict a single thread

In CS4414 we think of each computation in terms of a “thread”

A thread is a pointer into the program instructions. The CPU loads the instruction that the “PC” points to, fetches any operands from memory, does the action, saves the results back to memory.

Then the PC is incremented to point to the next instruction

# ASSEMBLY/MACHINE CODE VIEW




## Programmer-Visible State

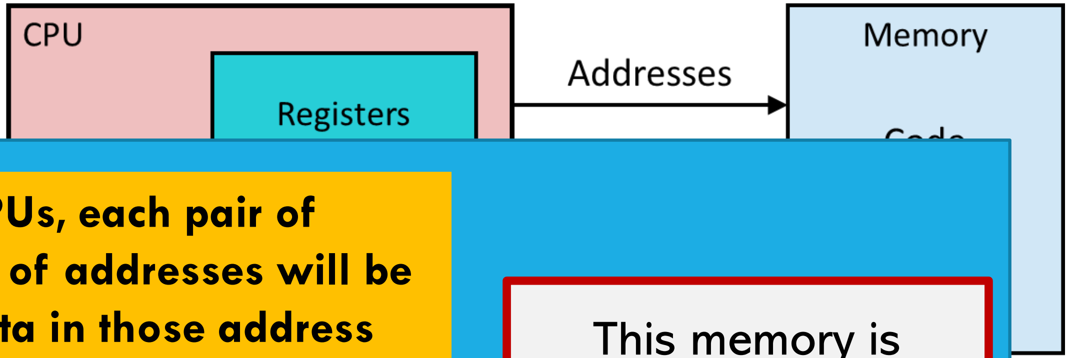
- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

## Memory

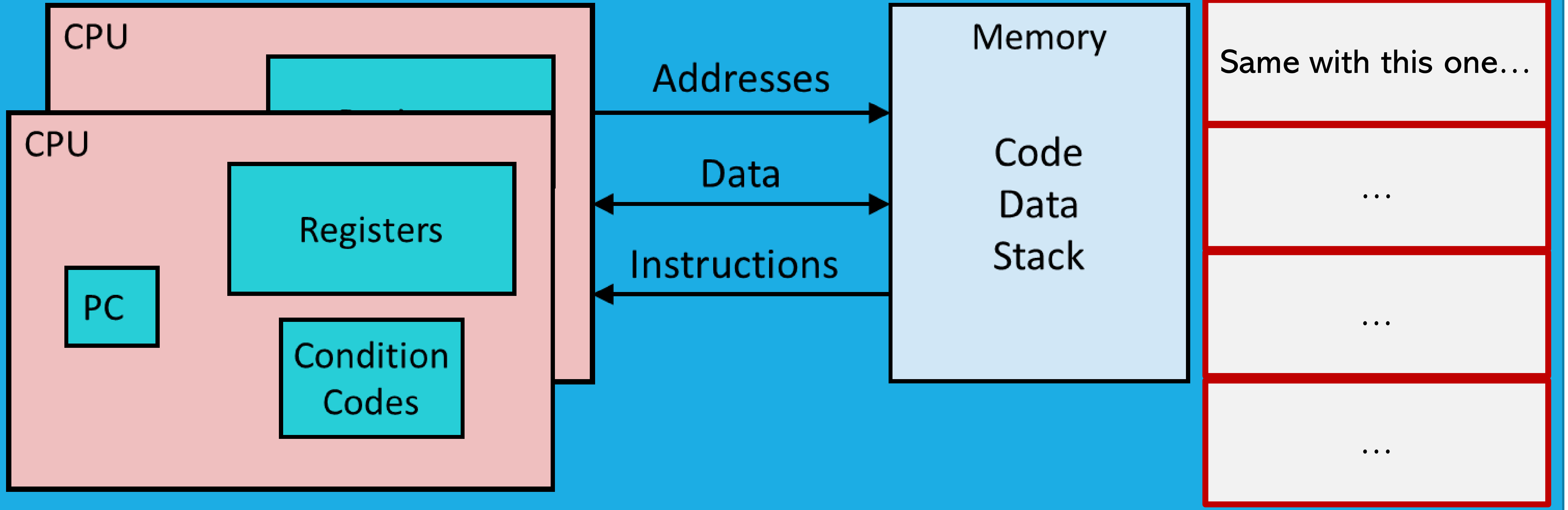
- Byte addressable array
- Code and user data
- Stack to support procedures

## Puzzle:

- *On a NUMA machine, a CPU is near a fast memory but can access* all *memory.*
- *How does this impact software design?*




**Example: With 6 on-board DRAM modules and 12 NUMA CPUs, each pair of CPUs has one nearby DRAM module. Memory in that range of addresses will be very fast. The other 5 DRAM modules are further away. Data in those address ranges is visible and everything looks identical, but access is slower!**



# LINUX TRIES TO HIDE MEMORY DELAYS

If it runs thread ***t*** on core ***k***, Linux tries to allocate memory for ***t*** (stack, malloc…) in the DRAM close to that ***k***.

Yet all memory operations work identically even if the thread is actually accessing some other DRAM. *They are just slower.*

*Linux doesn't even tell you which parts of your address space are mapped to which DRAM units.*

# MACHINE LANGUAGE

(We'll cover what we can but probably won't have time for all of this)

# THE HARDWARE UNDERSTANDS "PRIMITIVE" DATA TYPES

"Integer" data of 1, 2, 4, or 8 bytes

- Data values
- Addresses (untyped pointers)

Floating point data of 4, 8, or 10 bytes (new: 4-bit, 8-bit, 16-bit)

Code: Byte sequences encoding series of instructions

(SIMD vector data types of 8, 16, 32 or 64 bytes)

No aggregate types such as arrays or structures

- Just contiguously allocated bytes in memory
- **Example:** Raw images are arrays in a format defined by the camera or video, such as RGB, jpeg, mpeg. The *camera* understands the format. The host computer the camera is attached to just sees bytes

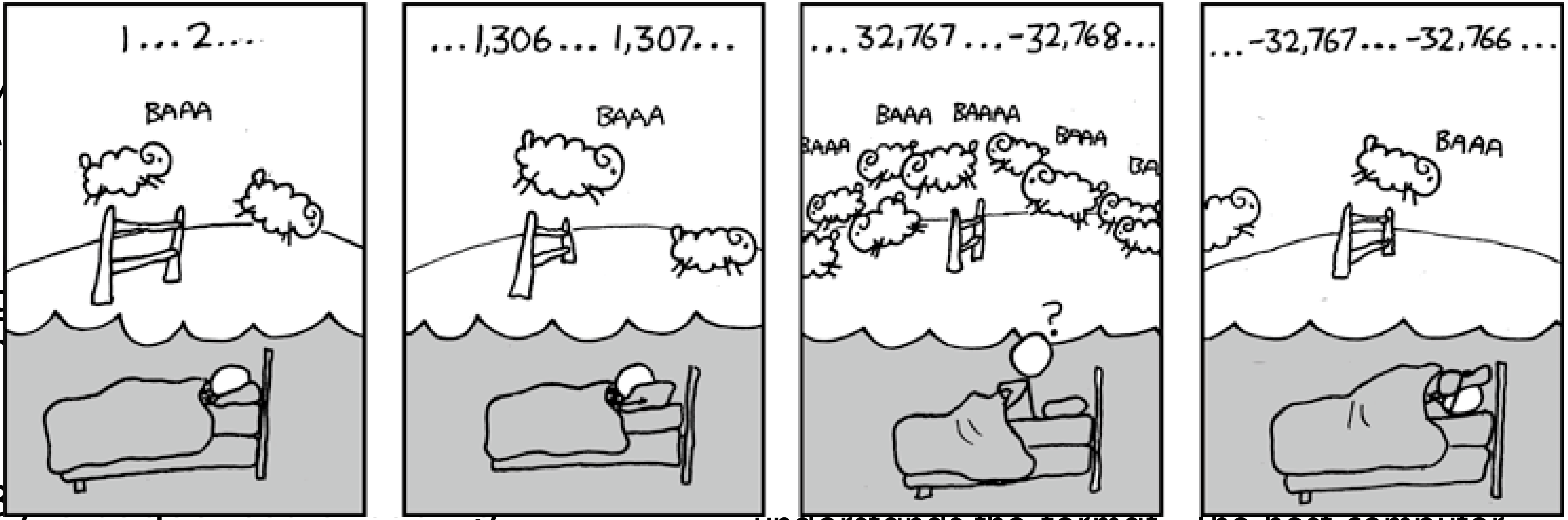
# THE HARDWARE UNDERSTANDS "PRIMITIVE" DATA TYPES

"Intege

- Data v
- Addre

Floating
bytes (r

Code: B
series of instructions



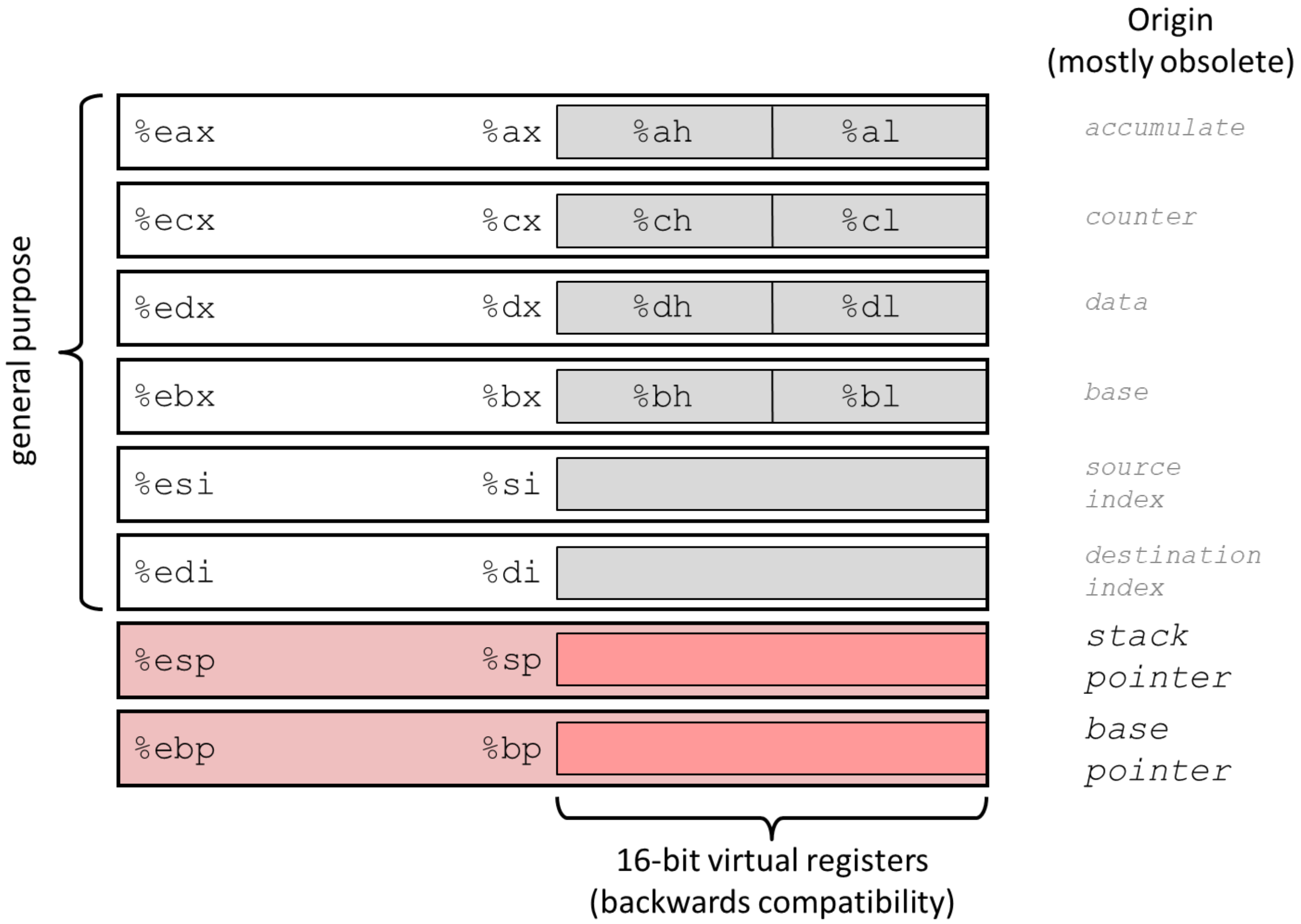

understands the format. The host computer
the camera is attached to just sees bytes

# X86-64 INTEGER REGISTERS

| | | | |
|---|---|---|---|
| %rax | %eax | %r8 | %r8d |
| %rbx | %ebx | %r9 | %r9d |
| %rcx | %ecx | %r10 | %r10d |
| %rdx | %edx | %r11 | %r11d |
| %rsi | %esi | %r12 | %r12d |
| %rdi | %edi | %r13 | %r13d |
| %rsp | %esp | %r14 | %r14d |
| %rbp | %ebp | %r15 | %r15d |

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

# SOME HISTORY: IA32 REGISTERS



# ASSEMBLY CHARACTERISTICS: OPERATIONS

Transfer data between memory and register
- Load data from memory into register
- Store register data into memory

Perform arithmetic function on register or memory data

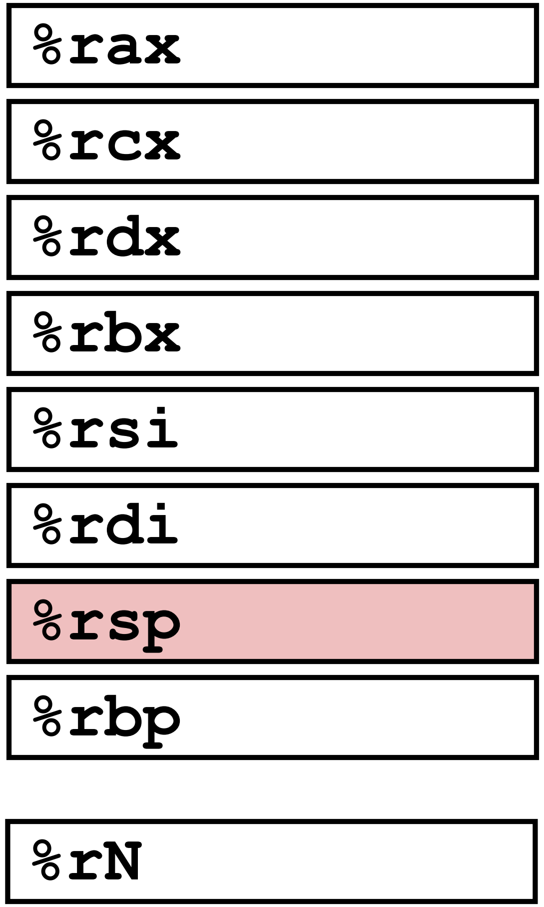
Transfer control
- Unconditional jumps to/from procedures
- Conditional branches
- Indirect branches

# Moving Data

- **Moving Data**

  **movq** *Source, Dest*

- **Operand Types**
  - ***Immediate:*** Constant integer data
    - Example: **$0x400, $-533**
    - Like C constant, but prefixed with **'$'**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **%rax, %r13**
    - But **%rsp** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at address given by register
    - Simplest example: **(%rax)**
    - Various other "addressing modes"




**Warning: Intel docs use mov *Dest, Source***

# `movq` Operand Combinations

| | Source | Dest | Src,Dest | C/C++ Analog |
|---|---|---|---|---|
| `movq` | *Imm* | *Reg* | `movq $0x4,%rax` | `temp = 0x4;` |
| | | *Mem* | `movq $-147,(%rax)` | `*p = -147;` |
| | *Reg* | *Reg* | `movq %rax,%rdx` | `temp2 = temp1;` |
| | | *Mem* | `movq %rax,(%rdx)` | `*p = temp;` |
| | *Mem* | *Reg* | `movq (%rax),%rdx` | `temp = *p;` |

***Cannot do memory-memory transfer with a single instruction***

# Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

# Example of Simple Addressing Modes

```
void
whatAmI(<type> a, <type> b)
{
      ????
}
```

%rdi → a

%rsi → b

```
whatAmI:
   movq     (%rdi), %rax
   movq     (%rsi), %rdx
   movq     %rdx, (%rdi)
   movq     %rax, (%rsi)
   ret
```

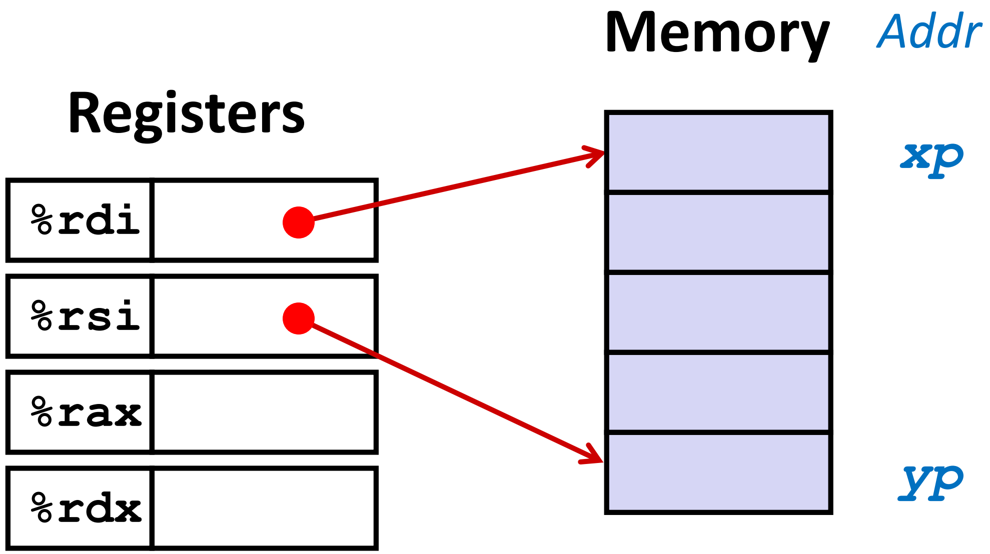
# Example of Simple Addressing Modes

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

# Understanding swap()

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```




| Register | Value |
|---|---|
| %rdi | xp |
| %rsi | yp |
| %rax | t0 |
| %rdx | t1 |

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

# Understanding swap()

**Registers**

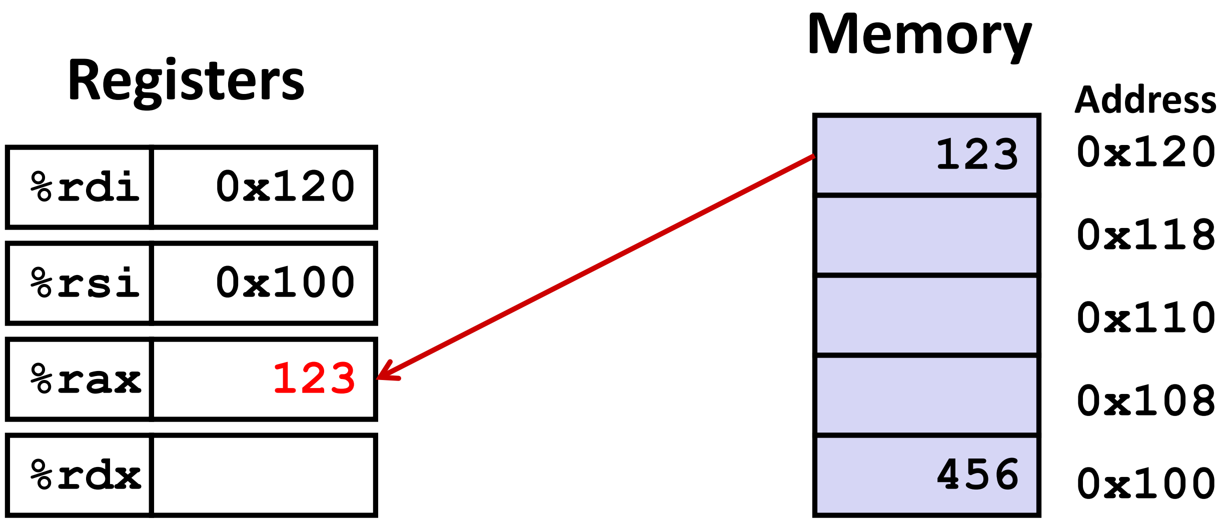
| | |
|---|---|
| %rdi | 0x120 |
| %rsi | 0x100 |
| %rax | |
| %rdx | |

**Memory**

| | Address |
|---|---|
| 123 | 0x120 |
| | 0x118 |
| | 0x110 |
| | 0x108 |
| 456 | 0x100 |

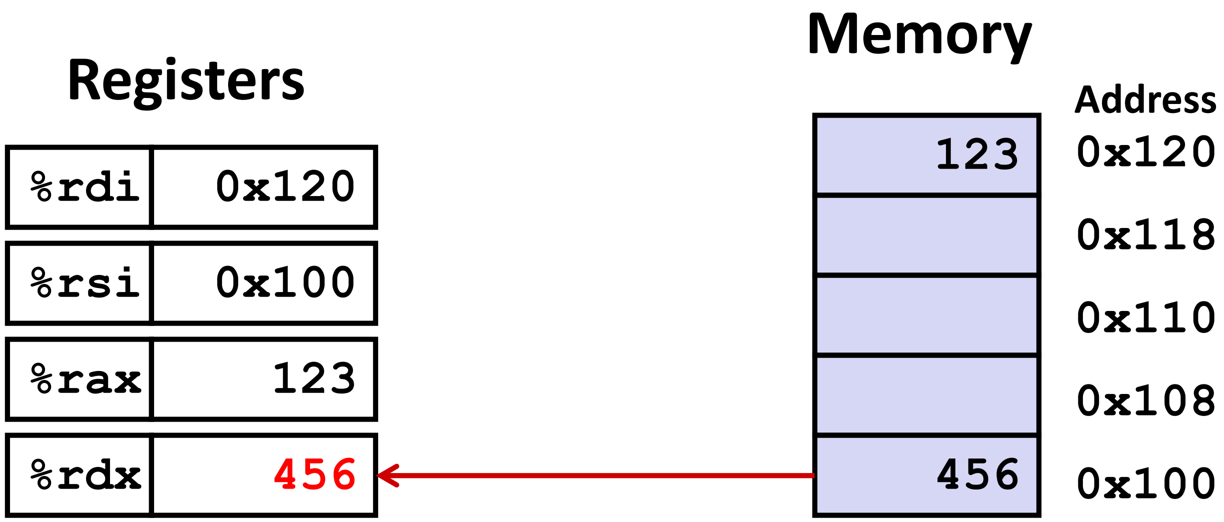
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

# Understanding swap()



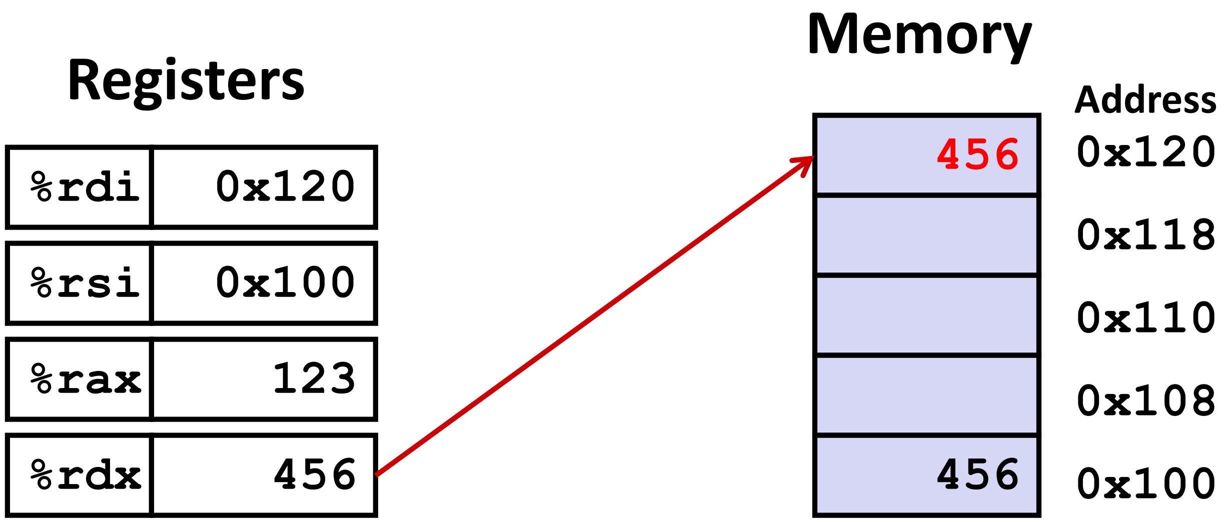

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

# Understanding swap()



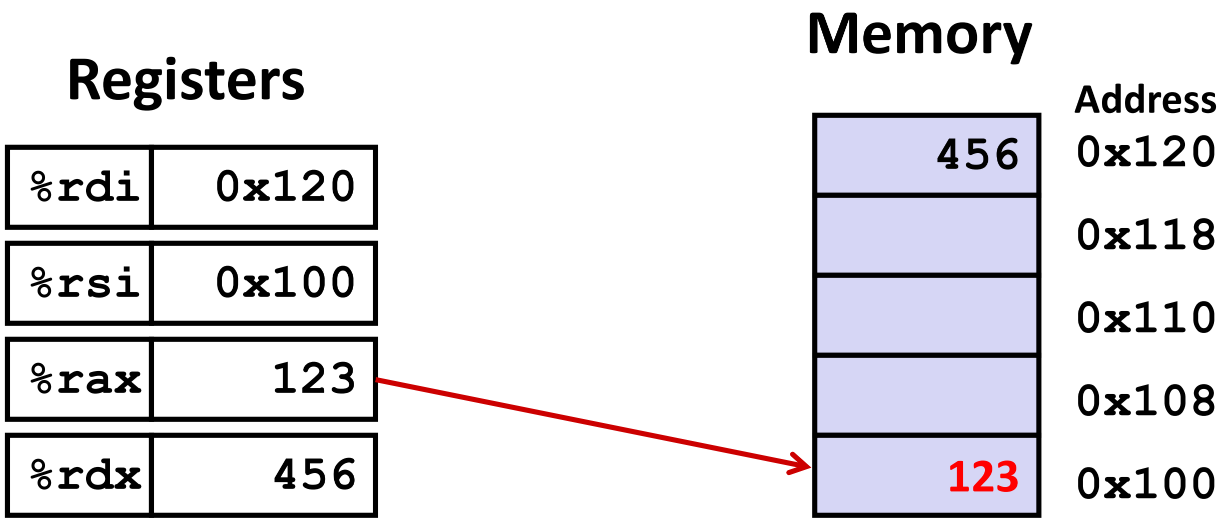

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

# Understanding swap()




```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

# Understanding swap()




```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

# Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

# Complete Memory Addressing Modes

- **Most General Form**

  **D(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  **(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]]**

  **D(Rb,Ri)** **Mem[Reg[Rb]+Reg[Ri]+D]**

  **(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]]**

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

**D(Rb,Ri,S)** **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for **`%rsp`**
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

| Expression | Address Computation | Address |
|---|---|---|
| `0x8(%rdx)` | | |
| `(%rdx,%rcx)` | | |
| `(%rdx,%rcx,4)` | | |
| `0x80(,%rdx,2)` | | |

# Address Computation Examples

| | |
|---|---|
| `%rdx` | `0xf000` |
| `%rcx` | `0x0100` |

| 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` |

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**
- C/C++, assembly, machine code

# Address Computation Instruction

- **`leaq` *Src, Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- **Example**

```
long m12(long x)
{
  return x*12;
}
```

**Converted to ASM by compiler:**

```
leaq (%rdi,%rdi,2), %rax   # t = x+2*x
salq $2, %rax              # return t<<2
```

# Some Arithmetic Operations

- **Two Operand Instructions:**

| *Format* | | *Computation* | |
|---|---|---|---|
| `addq` | *Src,Dest* | Dest = Dest + Src | |
| `subq` | *Src,Dest* | Dest = Dest – Src | |
| `imulq` | *Src,Dest* | Dest = Dest * Src | |
| `shlq` | *Src,Dest* | Dest = Dest << Src | ***Synonym: salq*** |
| `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! *Src,Dest***
  **(Warning: very old Intel docs use "op *Dest,Src*")**
- **No distinction between signed and unsigned int (why?)**

# Some Arithmetic Operations

- **One Operand Instructions**

| | | |
|---|---|---|
| `incq` | *Dest* | *Dest = Dest + 1* |
| `decq` | *Dest* | *Dest = Dest – 1* |
| `negq` | *Dest* | *Dest = – Dest* |
| `notq` | *Dest* | *Dest = ~Dest* |

- **See book for more instructions**
  - Depending how you count, there are 2,034 total x86 instructions
  - (If you count all addr modes, op widths, flags, it's actually 3,683)

# Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

## Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - Curious: only used once…

# Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

| Register | Use(s) |
|---|---|
| `%rdi` | Argument **x** |
| `%rsi` | Argument **y** |
| `%rdx` | Argument **z**, **t4** |
| `%rax` | **t1**, **t2**, **rval** |
| `%rcx` | **t5** |

# Evolution of Intel Instruction Set

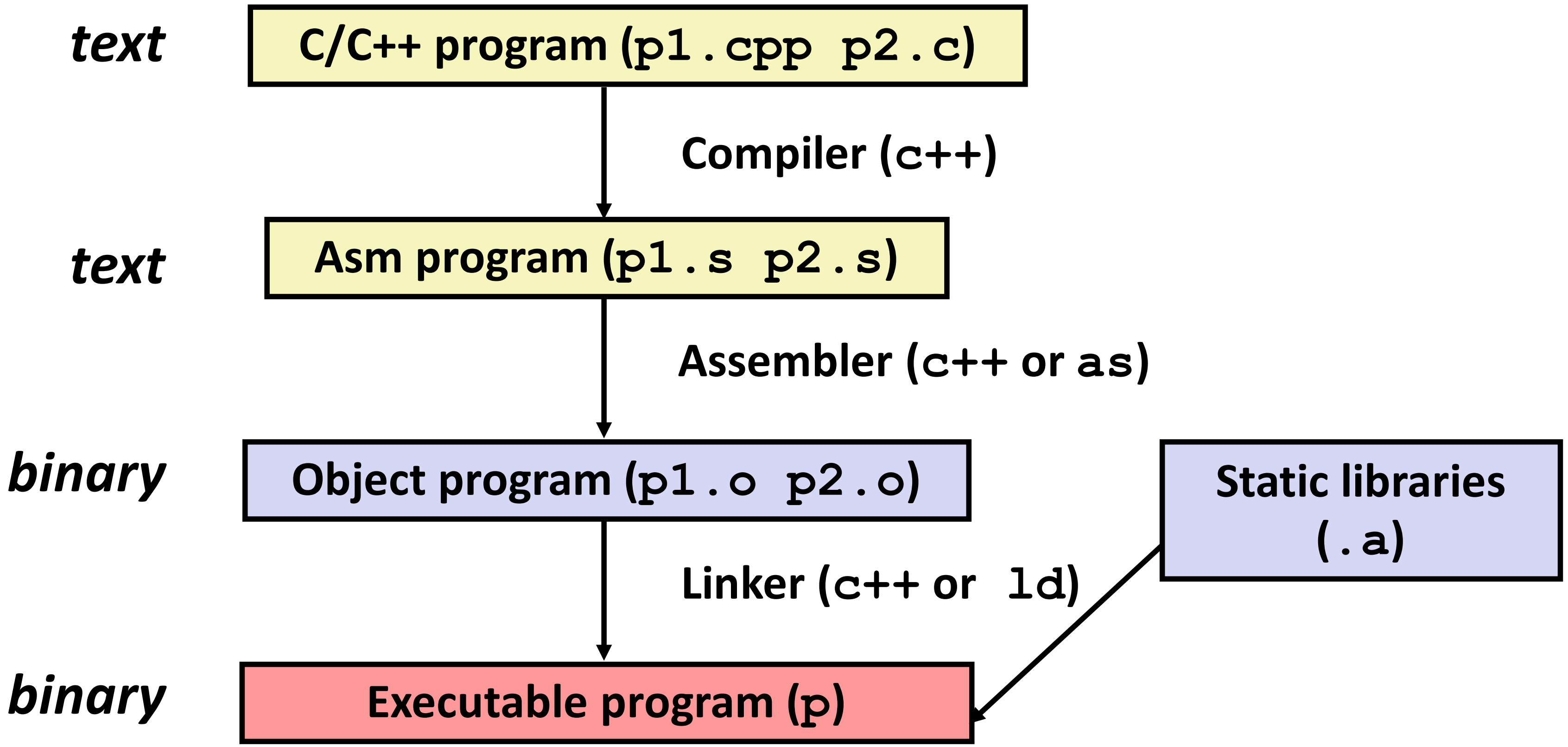
- **The Intel instruction set has changed over the decades since it was first introduced.**
- **Intel is a believer in the "CISC" model: complex instructions that are highly optimized**
- **Modern example: *vector parallel* instructions (also called SIMD: Single instruction, multiple data). Introduced to make the x86 more competitive with GPU accelerators**
  - Such as "Multiply these two vectors and put the result in this third vector", or "sum up the elements in this vector, and put the result *here*."
  - The underlying hardware uses parallel processing to do the job faster.
  - The C++ compiler can recognize many of these patterns and will emit vector parallel instructions (if the target computer supports them). You can also provide "hints" to the compiler, to do so.
- **There are many more examples; we will see a few later in the semester**

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- **C/C++, assembly, machine code**

# Turning C/C++ into Object Code

- Code in files `p1.cpp p2.c`
- Compile with command: `c++ pp1.cpp p2.c -o p`
  - There are often additional arguments such as –O3, -pg, -g…
  - Put resulting binary in file `p`



# Compiling Into Assembly

## C/C++ Code

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

## Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

This uses the "indirect" addressing mode: dest holds a memory address and *dest is a long integer at that address. We are using that location as a variable here!

**Obtain with command**

`C++ sum.c`

**Produces file** `sum.s`

# What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

# What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

**Things that look weird and are preceded by a '.' are generally directives.**

```
sumstore:
   pushq    %rbx
   movq     %rdx, %rbx
   call     plus
   movq     %rax, (%rbx)
   popq     %rbx
   ret
```

# Assembly Characteristics: Data Types

- **"Integer" data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

# Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

# Object Code

## Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x0400595`**

- **Assembler**
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - e.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

# Machine Instruction Example

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:   48 89 03
```

- **C Code**
  - Store value `t` where designated by `dest`
- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

    `t:` Register `%rax`

    `dest:` Register `%rbx`

    `*dest:` Memory **M[**`%rbx`**]**
- **Object Code**
  - 3-byte instruction
  - Stored at address `0x40059e`

# Disassembling Object Code

## Disassembled

```
0000000000400595 <sumstore>:
  400595:   53                  push   %rbx
  400596:   48 89 d3            mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff      callq  400590 <plus>
  40059e:   48 89 03            mov    %rax,(%rbx)
  4005a1:   5b                  pop    %rbx
  4005a2:   c3                  retq
```

- **Disassembler**

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

# Alternate Disassembly

## Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **Within gdb Debugger**
  - Disassemble procedure

```
gdb sum
disassemble sumstore
```

# Warning!


- Disassembly is useful when debugging but prohibited in many situations. A common and valid use is to understand what caused your own code to crash. With a complex piece of code knowing the line number isn't always enough.

- Hackers disassemble programs to look for coding errors that they can leverage to steal passwords or even take control by sending malformed inputs. This is why it is illegal to disassemble things like Microsoft Word.

- Cornell has harsh penalties for people who engage in hacking activities while enrolled in the university. A hacker could be suspended or expelled!