

Computational Optimization

ISE 407

Lecture 1

Dr. Ted Ralphs

Reading for this Lecture

- “How Computers Work,” R. Young
- “The Elements of Computing Systems,” N. Nisan and S. Schocken
- “Introduction to High Performance Computing”, V. Eijkhout, Chapter 1.
- “Introduction to High Performance Computing for Scientists and Engineers,” G. Hager and G. Wellein, Chapter 1.

What is a Computer System?

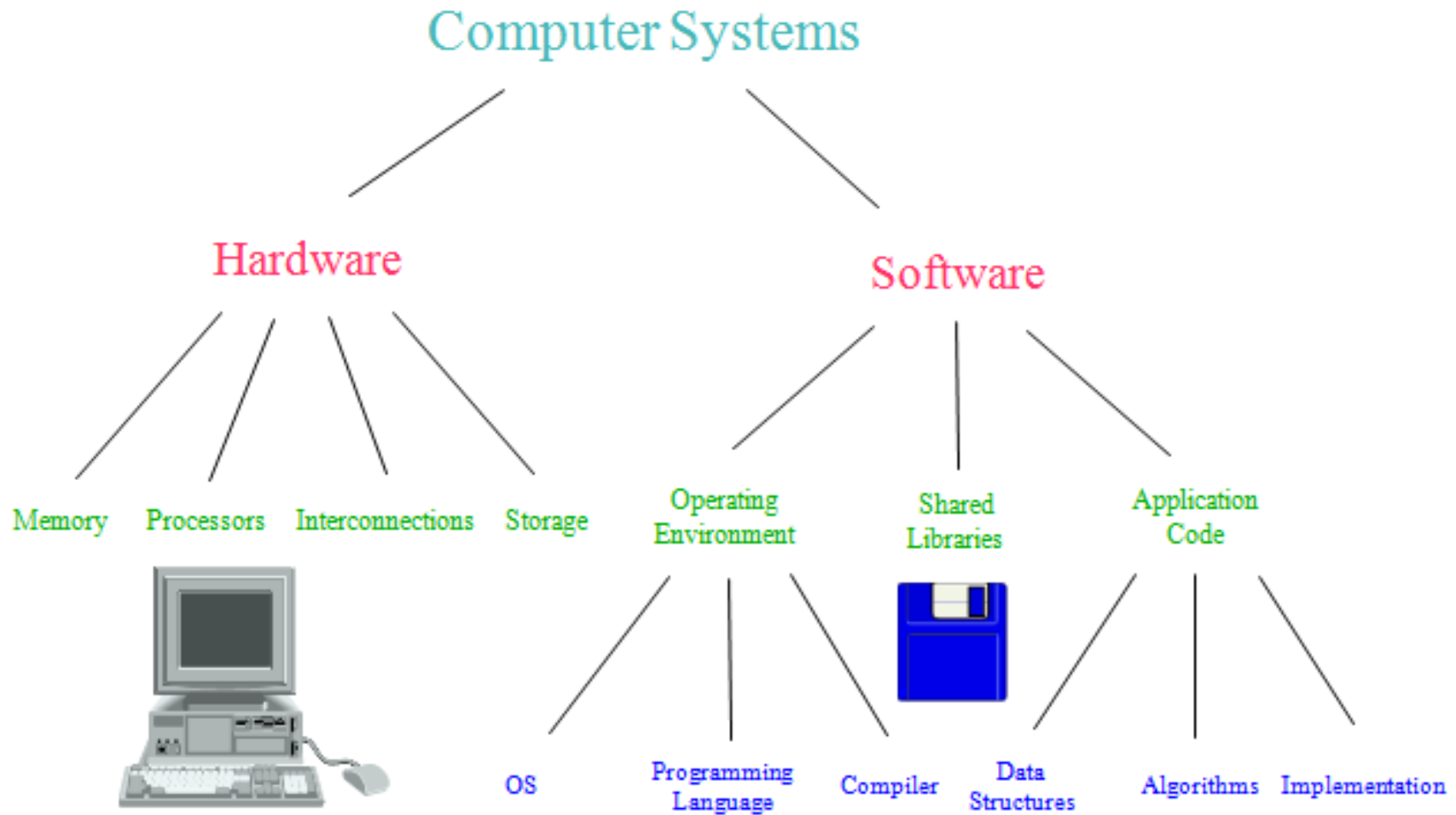


Figure 1: High Level View of a Computer System

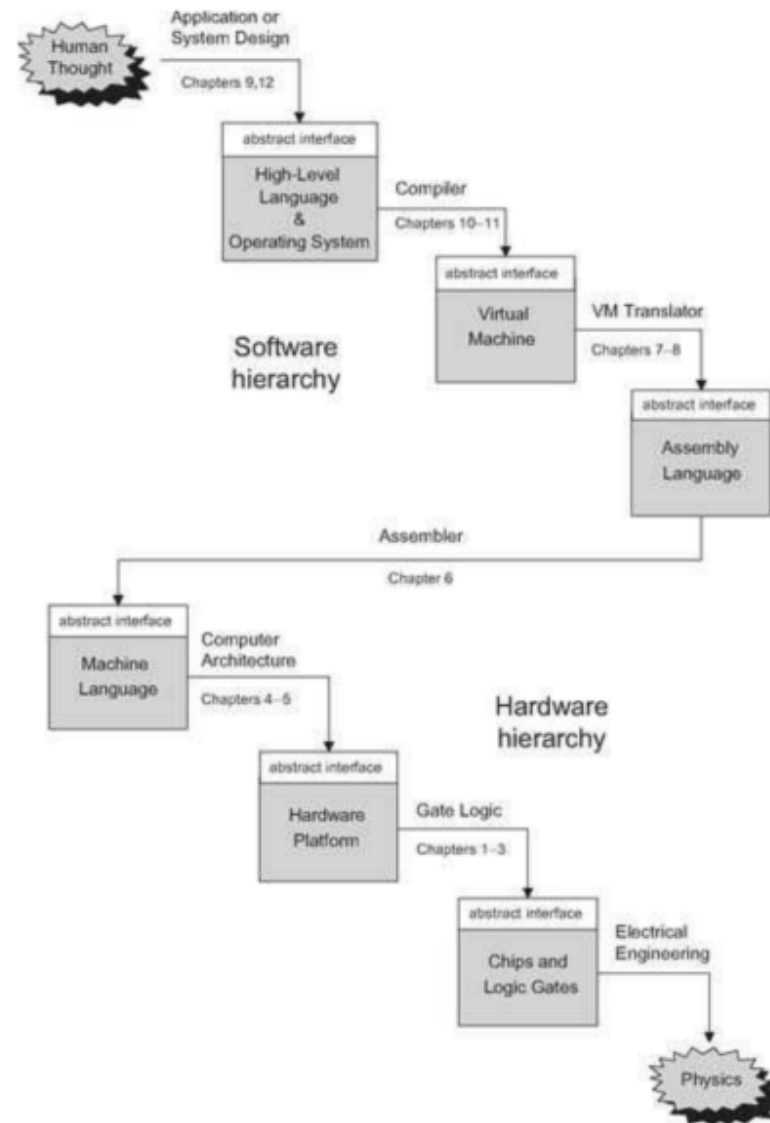
What Are Computational Methods?

- Computational methods are
 - Algorithms based on logical procedures rooted in mathematics.
 - Generally speaking, they are meant to be implemented on a computer.
- Such methods are usually stated initially in the high-level language of mathematics.
- This class is about translating mathematics into abstract procedures, then a programming language, and finally into machine instructions.
- The translation to a programming language must be done (primarily) by a human, whereas later stages are performed automatically.
- Doing this well requires an understanding of how computer systems work.

What Does a Computer System Really Do?

- A computer system
 - connects to physical input sources (keyboard, sensors, and other peripherals) to obtain data (instructions as well as raw input).
 - based on the instructions, manipulates the raw input through a sequence of logical operations to produce output.
 - sends the output to peripherals (screen, printer) that convert the output back into a physical source.
- Two basic components form the core of the system
 - The *central processing unit (CPU)* performs the logical operations.
 - The *storage system* connects to input and output devices and to the CPU to provide the input to and receive the output from the CPU.
- Loosely speaking, the *operating system* provides a range of “services.”
- These allow human programmers and users to interact with and control the computer in a more natural fashion.
- Naturally, this is all somewhat over-simplified.

Hierarchy of Abstractions




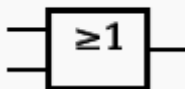

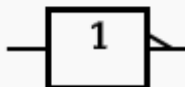


Source: Nisan and Schocken

Boolean Logic


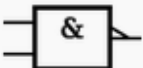

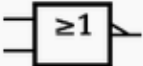

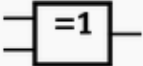

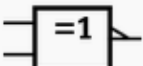
- The basic computational unit of a computer is the *logic gate*.
- An electronic version of such a gate can be built a small number (between 2 and 5) transistors.
- Starting from basic logic gates, it is possible to build chips that perform ever more sophisticated calculations.
- Underlying all of this computation is simple Boolean logic.
- Basic logic gates
 - AND
 - OR
 - NOT

Basic Logic Gates

Type	Distinctive shape	Rectangular shape	Boolean algebra between A & B	Truth table																		
AND			$A \cdot B$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A AND B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	INPUT		OUTPUT	A	B	A AND B	0	0	0	0	1	0	1	0	0	1	1	1
INPUT		OUTPUT																				
A	B	A AND B																				
0	0	0																				
0	1	0																				
1	0	0																				
1	1	1																				
OR			$A + B$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A OR B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	INPUT		OUTPUT	A	B	A OR B	0	0	0	0	1	1	1	0	1	1	1	1
INPUT		OUTPUT																				
A	B	A OR B																				
0	0	0																				
0	1	1																				
1	0	1																				
1	1	1																				
NOT			\overline{A}	<table><tr><th>INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>NOT A</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	INPUT	OUTPUT	A	NOT A	0	1	1	0										
INPUT	OUTPUT																					
A	NOT A																					
0	1																					
1	0																					

Source: https://en.wikipedia.org/wiki/Logic_gate

Composite Gates

NAND			$\overline{A \cdot B}$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A NAND B</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	INPUT		OUTPUT	A	B	A NAND B	0	0	1	0	1	1	1	0	1	1	1	0
INPUT		OUTPUT																				
A	B	A NAND B																				
0	0	1																				
0	1	1																				
1	0	1																				
1	1	0																				
NOR			$\overline{A + B}$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A NOR B</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	INPUT		OUTPUT	A	B	A NOR B	0	0	1	0	1	0	1	0	0	1	1	0
INPUT		OUTPUT																				
A	B	A NOR B																				
0	0	1																				
0	1	0																				
1	0	0																				
1	1	0																				
XOR			$A \oplus B$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A XOR B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	INPUT		OUTPUT	A	B	A XOR B	0	0	0	0	1	1	1	0	1	1	1	0
INPUT		OUTPUT																				
A	B	A XOR B																				
0	0	0																				
0	1	1																				
1	0	1																				
1	1	0																				
XNOR			$\overline{A \oplus B}$ or $A \odot B$	<table><tr><th colspan="2">INPUT</th><th>OUTPUT</th></tr><tr><th>A</th><th>B</th><th>A XNOR B</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	INPUT		OUTPUT	A	B	A XNOR B	0	0	1	0	1	0	1	0	0	1	1	1
INPUT		OUTPUT																				
A	B	A XNOR B																				
0	0	1																				
0	1	0																				
1	0	0																				
1	1	1																				

Boolean Arithmetic

- From basic logic gates, we can straightforwardly build chips that do arithmetic.
- The addition of two binary numbers can be reduced to sequences of additions of three bits (the third bit is the carry).
- Most other arithmetic operations can be reduced to sequences of additions.
- Thus, from these basic elements, we can build a chip that does most of the things we want it to do.

CPU

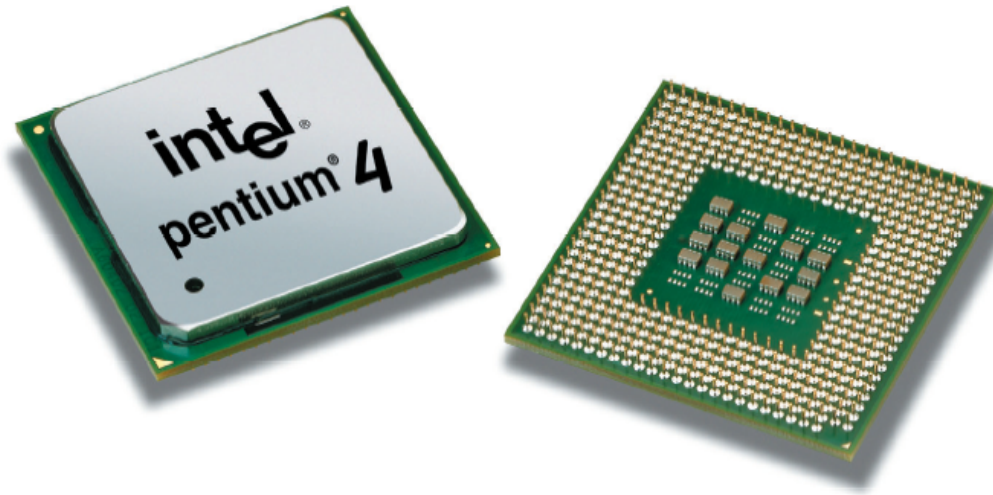
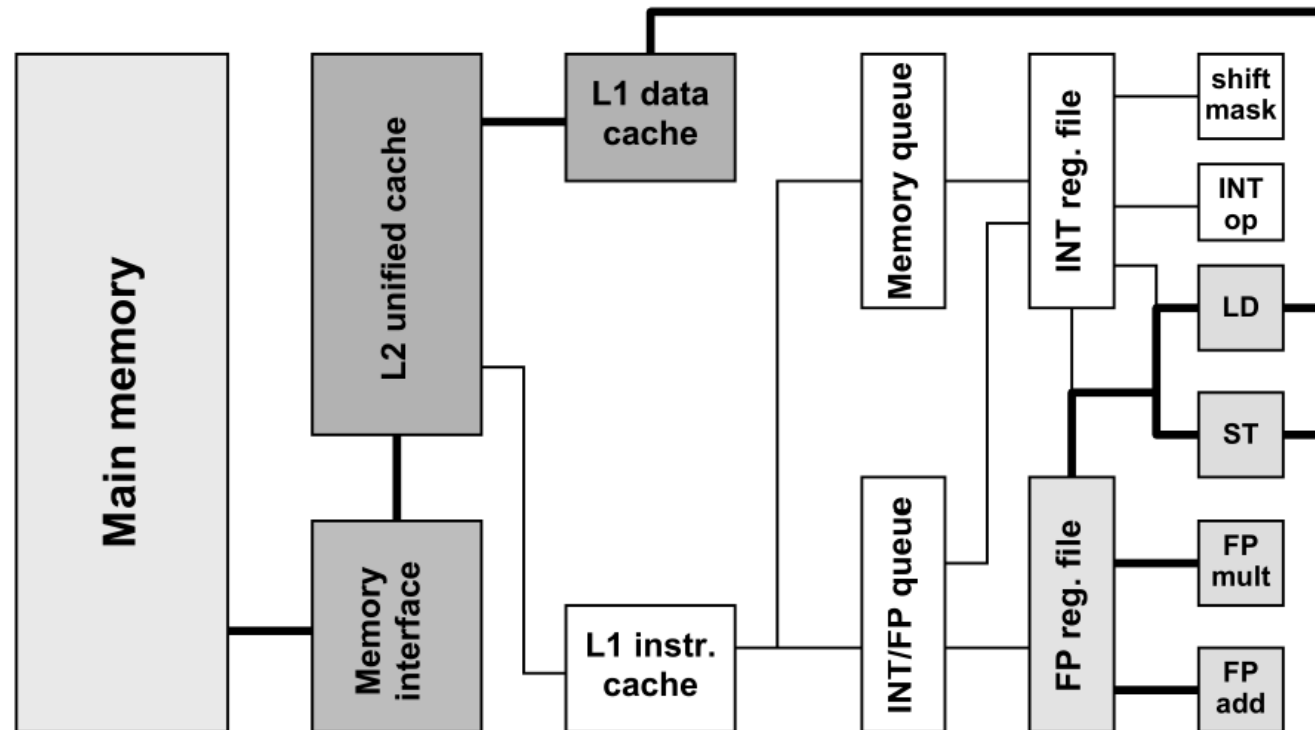


Figure 1 Central Processing Unit

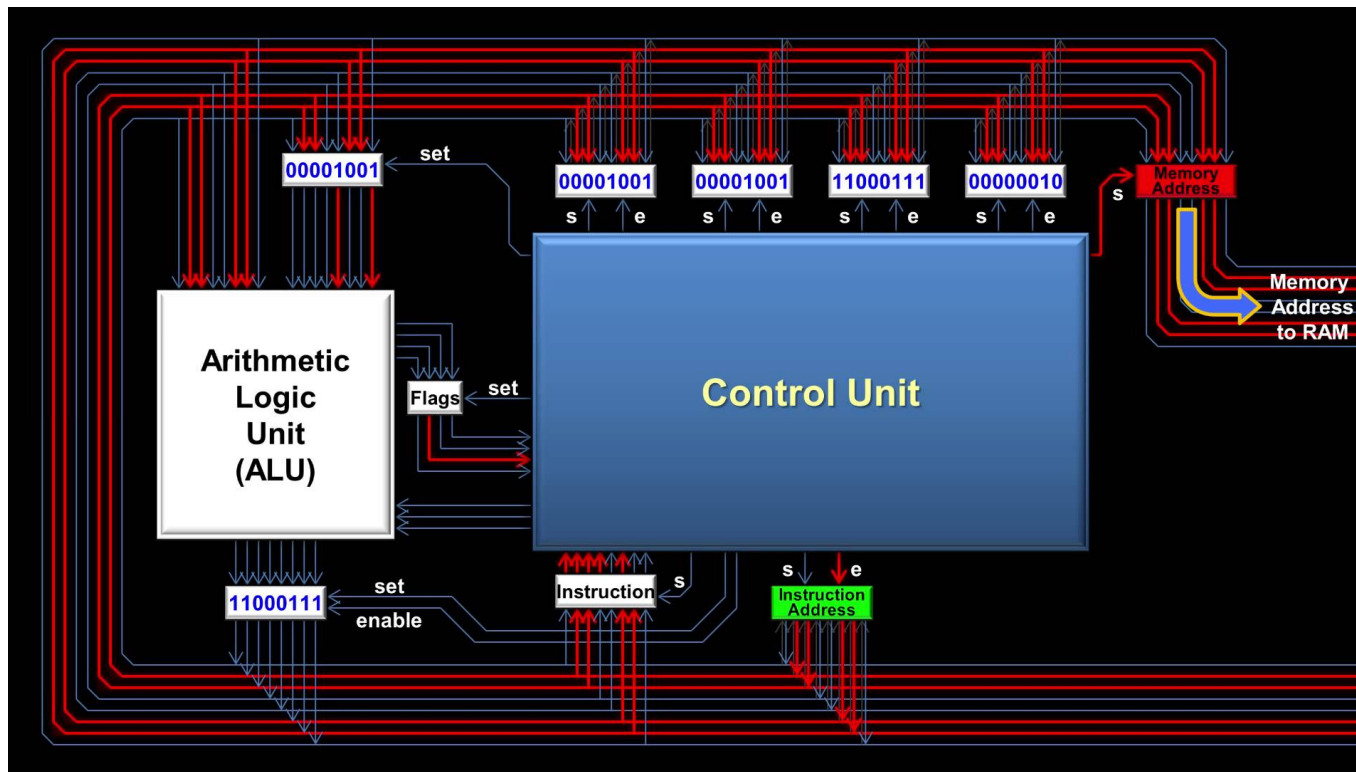
- Modern CPUs combine billions of such logic gates in a complex physical manifestation of these ideas.
- CPU stands for “central processing unit,” but what we call a CPU is more like a “CPU chip.”
- The chip may have multiple “compute cores” for doing computations in parallel.
- The CPU has other sub-parts, such as a control unit, arithmetic logic unit, memory management unit, clock, etc.

Basic CPU Architecture



Source: Hager and Wellein, Figure 1.2

How a CPU Works



Watch this: https://www.youtube.com/watch?v=cNN_tTXABUA

Basic Architecture of a Computer

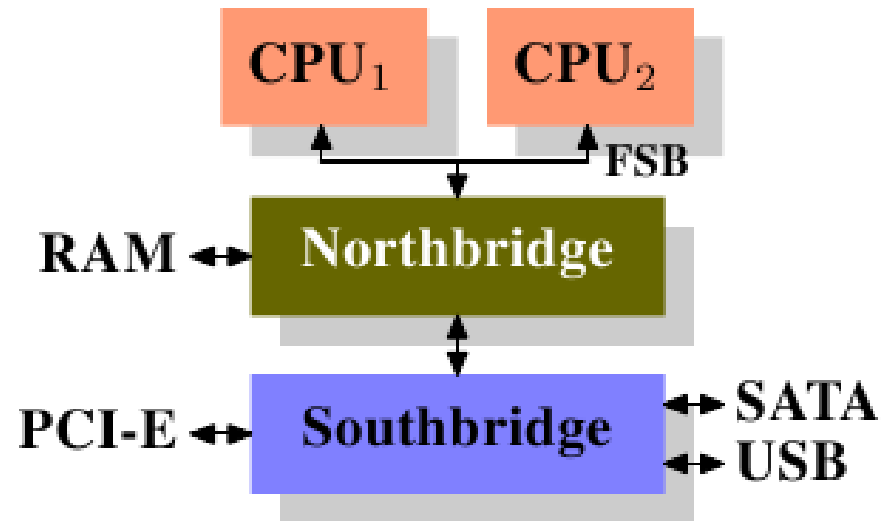


Figure 2: Basic architecture of a modern computer

Basic elements include

- CPU (Central processing unit)
- RAM (Random access memory)
- Storage
- Peripherals

RAM



- RAM stands for random access memory (as opposed to older style memory, like magnetic tape, that had to be accessed linearly).
- Typical “burst” communication rate is around 10 GB/s
- Size varies from 4-32 GB
- It is temporary memory
- It keeps data and instructions for CPU

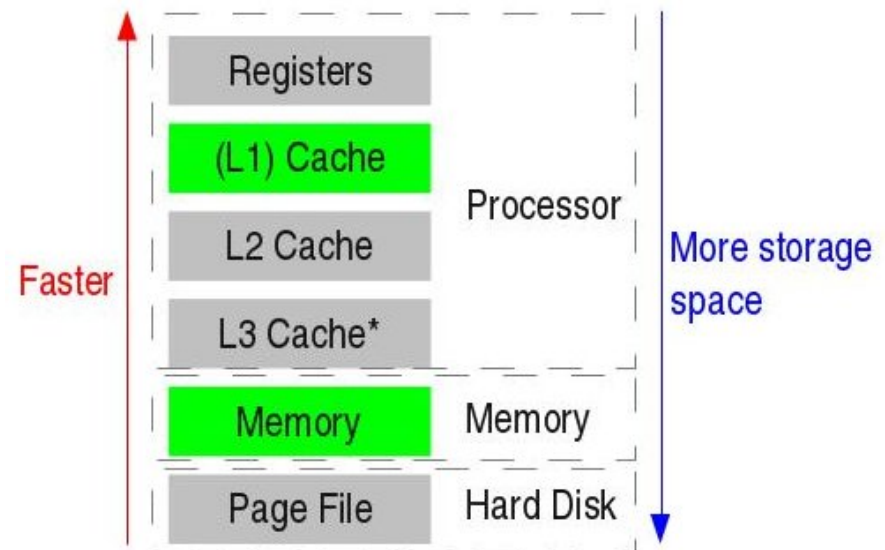
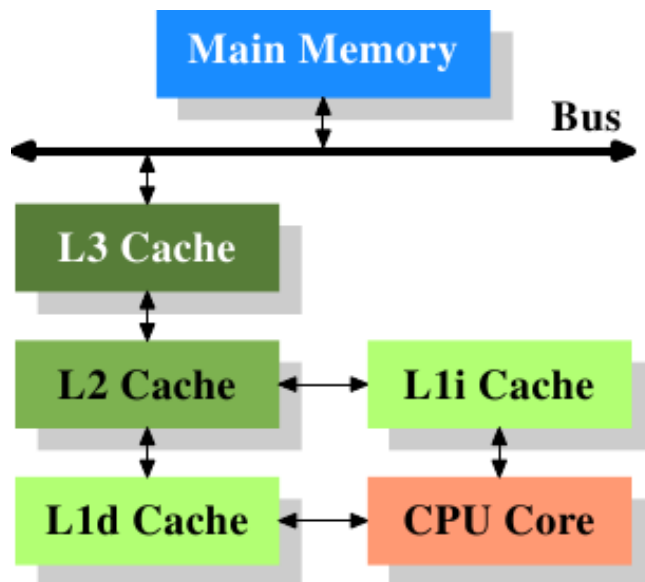
How RAM works

- Memory is divided into slots that store a fixed number of bits.
- Each slot gets an address.
- To retrieve data, you must have the address.
- We'll see later that RAM is supplemented by smaller, faster blocks of memory called *cache*.

Address	Value
0x00	01001010
0x01	10111010
0x02	01011111
0x03	00100100
0x04	01000100
0x05	10100000
0x06	01110100
0x07	01101111
0x08	10111011
...	...
0xFE	11011110
0xFF	10111011

Storage Hierarchy

- There is a large gap between processor speeds and memory speeds.
- It is possible to produce faster memory, but it's expensive and takes much more physical space.
- As a compromise, we add small fast memory, called *cache*, for storing the most important data.
- This a crucial driver of performance and we'll delve further into it in Lecture 2.

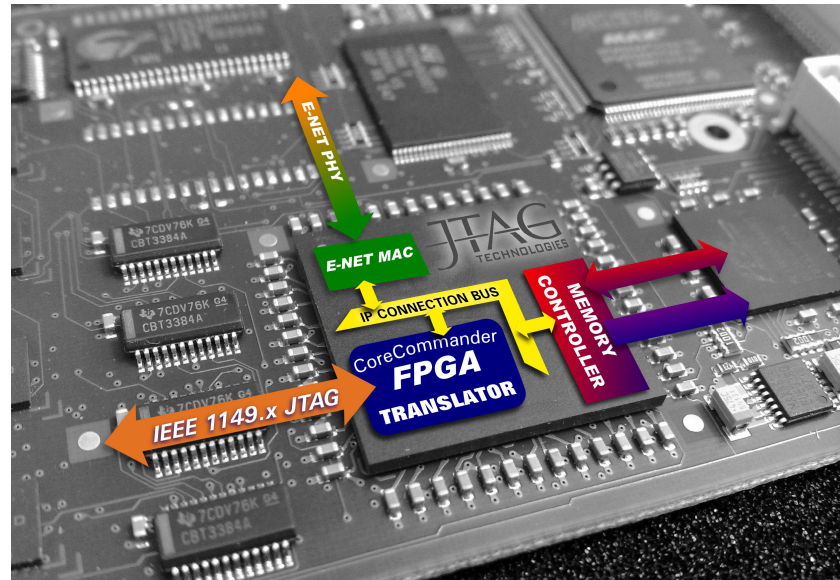


Other Technologies: GPU



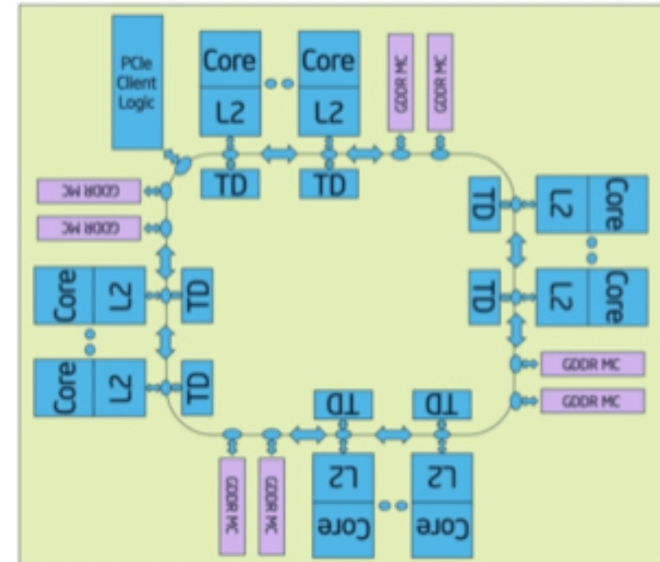
- GPUs are not new, but are now more general-purpose.
- They can be programmed in C (with some restrictions).
- Provide the ability to run blocks of many synchronized parallel threads executing the same “kernel.”
- Blocks have access to small, fast shared memory and slow global memory.
- Amount of global memory per thread is much smaller than CPU.

Other Technologies: FPGA



- Field-programmable gate arrays (FPGAs) are integrated circuits designed to be configured by a customer or a designer after manufacturing.
- They essentially CPUs custom-designed for certain workloads.

Other Technologies: Co-processors



- Xeon Phi is similar to a GPU, but with a more general instruction set.
- “Essentially a 60-core SMP chip where each core has a dedicated 512-bit wide SSE (Streaming SIMD Extensions) vector unit” –Dr. Dobbs
- Got it?

Machine Language

- The native language spoken by a computer is a sequence of 0s and 1s.
- These are divided into chunks that can be interpreted as instructions, memory addresses, numbers, or other data units.
- The chip provides a basic set of instructions that it can understand.
- This is called *machine language*.
- It can be translated into a more human readable form known as *assembly language*.
- In the early history of computing, all programming was done in assembly language.
- This is the first of many layers that exist to translate human thought into action by the computer.

Simple Example

```
1 function f(x::Int)
2     x += 5
3 end
```

```
julia> @code_native debuginfo=:none f(10)
```

```
1     .text
2     leaq    5(%rdi), %rax
3     retq
4     nopw    %cs:(%rax,%rax)
5     nop
```

The instruction set used here is the **x86-64** instruction set that is most commonly in use on modern computers.

Registers

- The codes beginning with with % on the last slide are names for the available registers.
- In the x86-64 architecture, the registers hold 64-bit, but the lower bits can be used as 32-, 16-, or 8-bit registers.
- Certain registers play special roles, mainly by convention.
 - `%rax` is used to store a function's return value.
 - `%rsp` is the stack pointer.
 - `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are the first six integer or pointer arguments to a function.

Registers

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Source: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

More Complex Example

```
1  function f(x::Array{Float64,1}, stride::Int, limit::Int)
2      s = 0
3      i = 0
4      while(true)
5          i += stride
6          if (i > limit)
7              break
8          end
9          @inbounds s=x[i]
10     end
11     return s
12 end
```

More Complex Example (cont'd)

```
julia> @code_native debuginfo=:none f(x, 2, length(x))
```

```
1      .text
2      cmpq    %rcx, %rdx
3      jle     L17
4      movq    $0, (%rdi)
5      movb    $2, %dl
6      xorl    %eax, %eax
7      retq
8  L17:
9      xorl    %r8d, %r8d
10     nopw    %cs:(%rax,%rax)
11     nop
12  L32:
13     addq    %rdx, %r8
14     leaq    (%rdx,%r8), %rax
15     cmpq    %rcx, %rax
16     jle     L32
17     movq    (%rsi), %rax
18     movq    -8(%rax,%r8,8), %rax
19     movq    %rax, (%rdi)
20     movb    $1, %dl
21     xorl    %eax, %eax
22     retq
23     nopl    (%rax)
```

How Numbers Are Represented

- Recall from 418 that numbers are represented in the floating point system.
- The floating-point numbers F are a subset of the real numbers.
- A particular floating-point number system F is characterized by four parameters:
 - the base β ,
 - the precision t ,
 - the exponent range $[L, U]$.
- Each floating-point number $x \in F$ has a value

$$x = \pm \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \beta^e,$$

where the integers $0 \leq d_i \leq \beta - 1$ for $1 \leq i \leq t$ and $L \leq e \leq U$.

- Only numbers of this form can be represented and this can have strange consequences.

IEEE 754

- The modern implementation of the floating point number system is specified in IEEE 754.
- There are a lot of technical details in the actual storage and these differ slightly from the above description.
- We always have $\beta = 2$ (everything is stored in binary format).
 - Single Precision (32 bit, 4 bytes): $t = 24$, exponent range $\approx [-8, 8]$
 - Double Precision (64 bits, 8 bytes): $t = 53$, exponent range $\approx [-11, 11]$
- The set F is not a continuum, or even an infinite set.
- The numbers are not equally spaced throughout their range.



Fig. 2.1. The floating-point number system for $\beta = 2$, $t = 3$, $L = -1$, $U = 2$.

Numbers in Julia

Julia has a very robust set of numerical types, including complex and rational.

- Integer types:

Type	Signed?	Number of bits	Smallest value	Largest value
<code>Int8</code>	✓	8	-2^7	$2^7 - 1$
<code>UInt8</code>		8	0	$2^8 - 1$
<code>Int16</code>	✓	16	-2^{15}	$2^{15} - 1$
<code>UInt16</code>		16	0	$2^{16} - 1$
<code>Int32</code>	✓	32	-2^{31}	$2^{31} - 1$
<code>UInt32</code>		32	0	$2^{32} - 1$
<code>Int64</code>	✓	64	-2^{63}	$2^{63} - 1$
<code>UInt64</code>		64	0	$2^{64} - 1$
<code>Int128</code>	✓	128	-2^{127}	$2^{127} - 1$
<code>UInt128</code>		128	0	$2^{128} - 1$
<code>Bool</code>	N/A	8	false (0)	true (1)

- Floating-point types:

Type	Precision	Number of bits
<code>Float16</code>	half	16
<code>Float32</code>	single	32
<code>Float64</code>	double	64

Machine Epsilon

- It is simple to obtain the machine epsilon value for a given float type.

```
julia> eps(Float32)  
1.1920929f-7
```

```
julia> eps(Float64)  
2.220446049250313e-16
```

- This is just computed as the smallest difference between any two floating point numbers of the specified type.

Execution of a Program

- In a simple architecture, one instruction is executed per time step (called a CPU *cycle*).
- The speed of computation is (partially) determined by the CPU *frequency*, which is the number of cycles per second.
- The frequency is limited by the physics of the device (heat dissipation, etc.).
- The simplest instructions amount to
 - “Move this data from here to there” (possibly masking certain bits or rotating the bits in the process).
 - “Look for the next instruction here”.
- From this basic kind of instruction, we can derive all the things a modern computer can do.
- Most modern architectures have complex instruction sets and some instructions take multiple cycles to execute.

Higher-Level Instructions

- Data handling and memory operations
 - Put a value in a register
 - Move data from memory to a register or vice versa
 - Read or write data from hardware devices
- Arithmetic and logic
 - Add, subtract, multiply, divide.
 - Bitwise operations (conjunction/disjunction).
 - Comparison
- Control flow
 - Branch
 - Conditionally branch
 - Indirectly branch

Source: en.wikipedia.org/wiki/Instruction_set

Modern Processors, Pipelining, and Vectorization

- On a modern computer, instructions typically take multiple cycles.
- However, the CPU may be able to overlap execution, starting a new instruction before the old one is finished.
- This process is called *pipelining*.
- Certain instruction can also operate simultaneously on 4 64-bit integers/floats at a time.
- This creates an additional source of parallelism.
- Therefore, we have to take into account both
 - latency (how many cycles the instruction takes to complete) and
 - (reciprocal) throughput (average number of instructions per cycle)

Source: <https://biojulia.net/post/hardware/>

Instruction Speeds in Practice

Instruction	Latency	Reciprocal throughput
move data	1	0.25
and/or/xor	1	0.25
test/compare	1	0.25
do nothing	1	0.25
int add/subtract	1	0.25
bitshift	1	0.5
float multiplication	5	0.5
vector int and/or/xor	1	0.5
vector int add/sub	1	0.5
vector float add/sub	4	0.5
vector float multiplic.	5	0.5
lea	3	1
int multiplic	3	1
float add/sub	3	1
float multiplic.	5	1
float division	15	5
vector float division	13	8
integer division	50	40

Source: <https://biojulia.net/post/hardware/>

Moving Data

- As we have seen, CPUs can only operate on data that resides in a limited number of *registers*.
- We must therefore be constantly moving data from where it resides into the register (and then move the result of computation back out).
- One of the most important drivers of the speed of computation is how efficiently this can be done.
- There is a complex hierarchy of hardware devices whose goal is to move data as efficiently as possible to the registers and back out again.