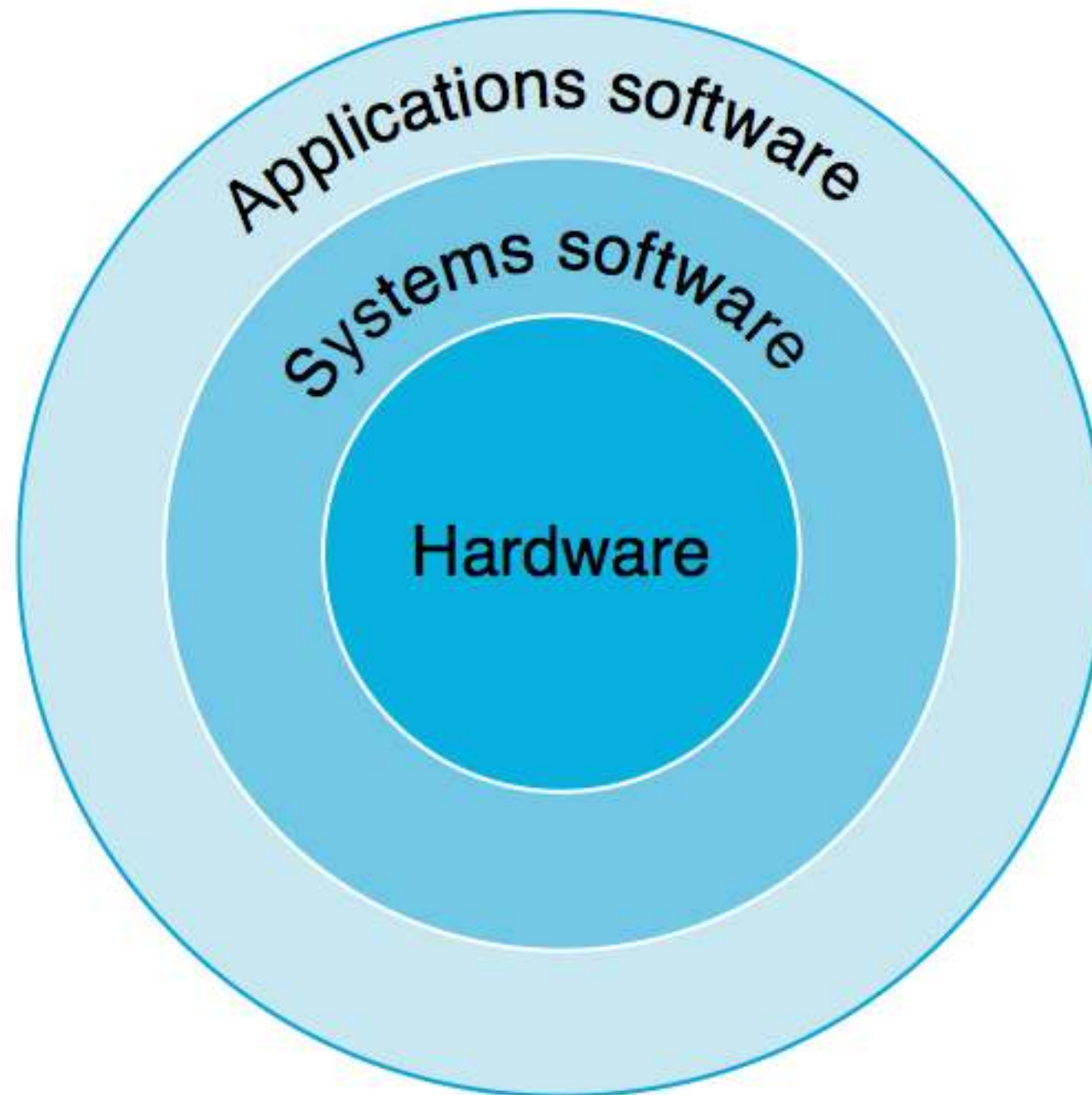
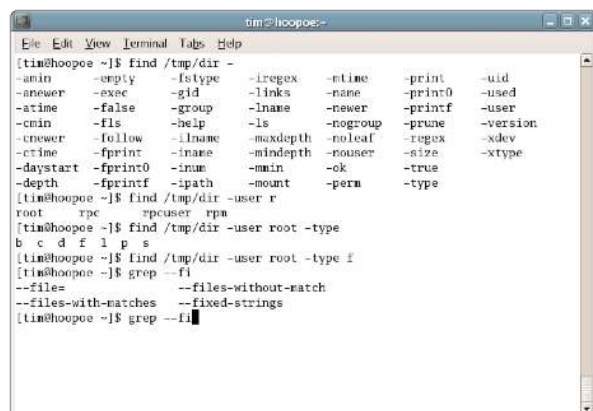


The Fundamentals of Computer Architecture

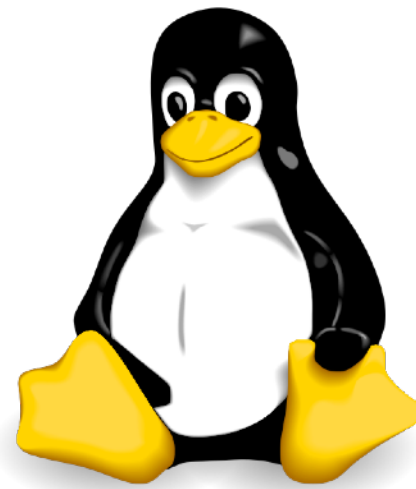


Application Software



Systems Software

Operating Systems



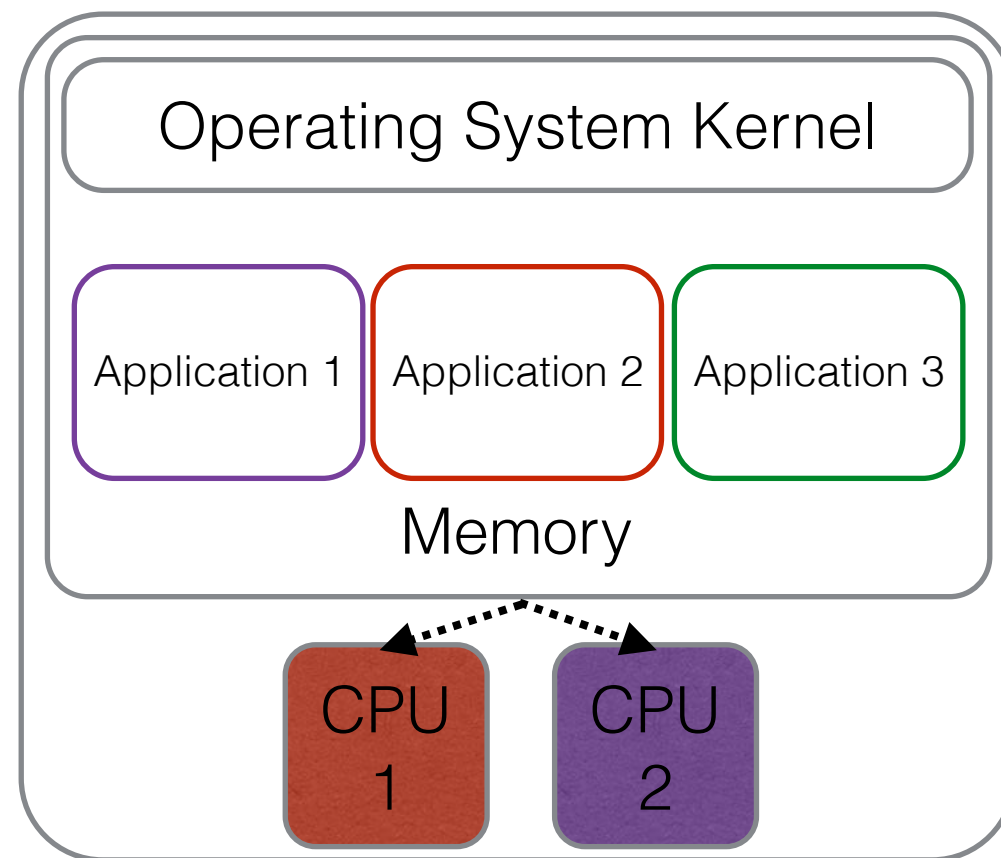
Compilers, Linkers, Assemblers



GCC:
GNU Compiler
Collection

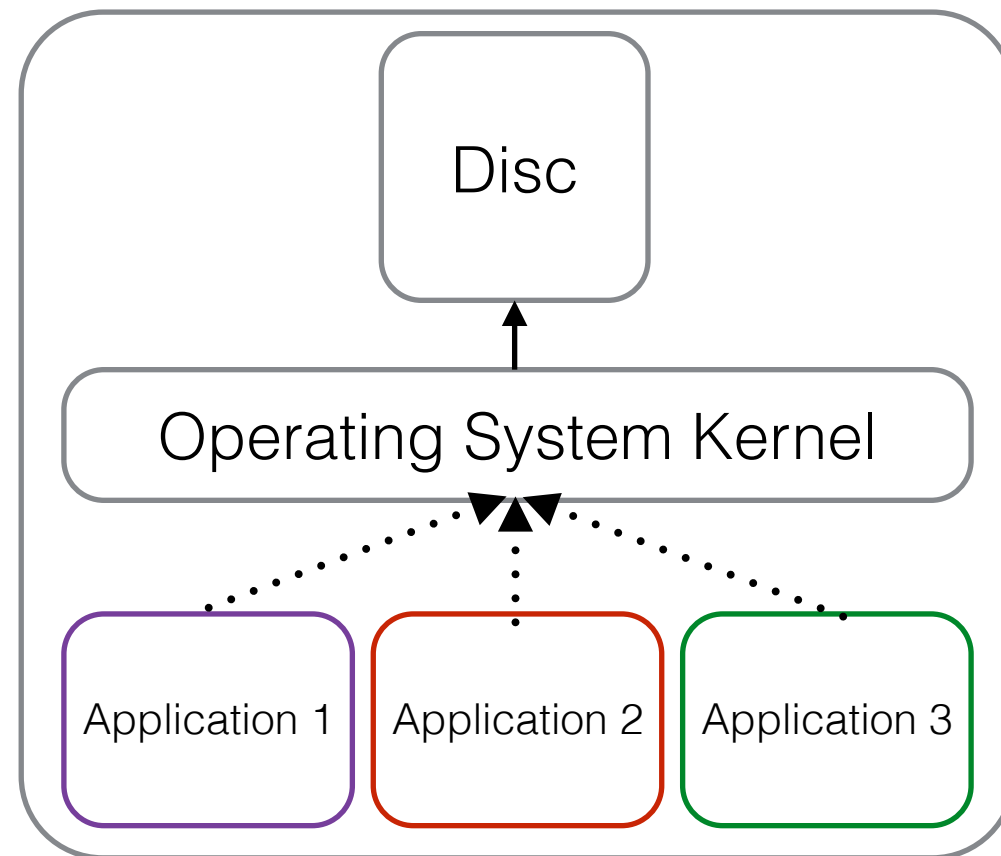
Systems Software

Operating Systems, ctd.



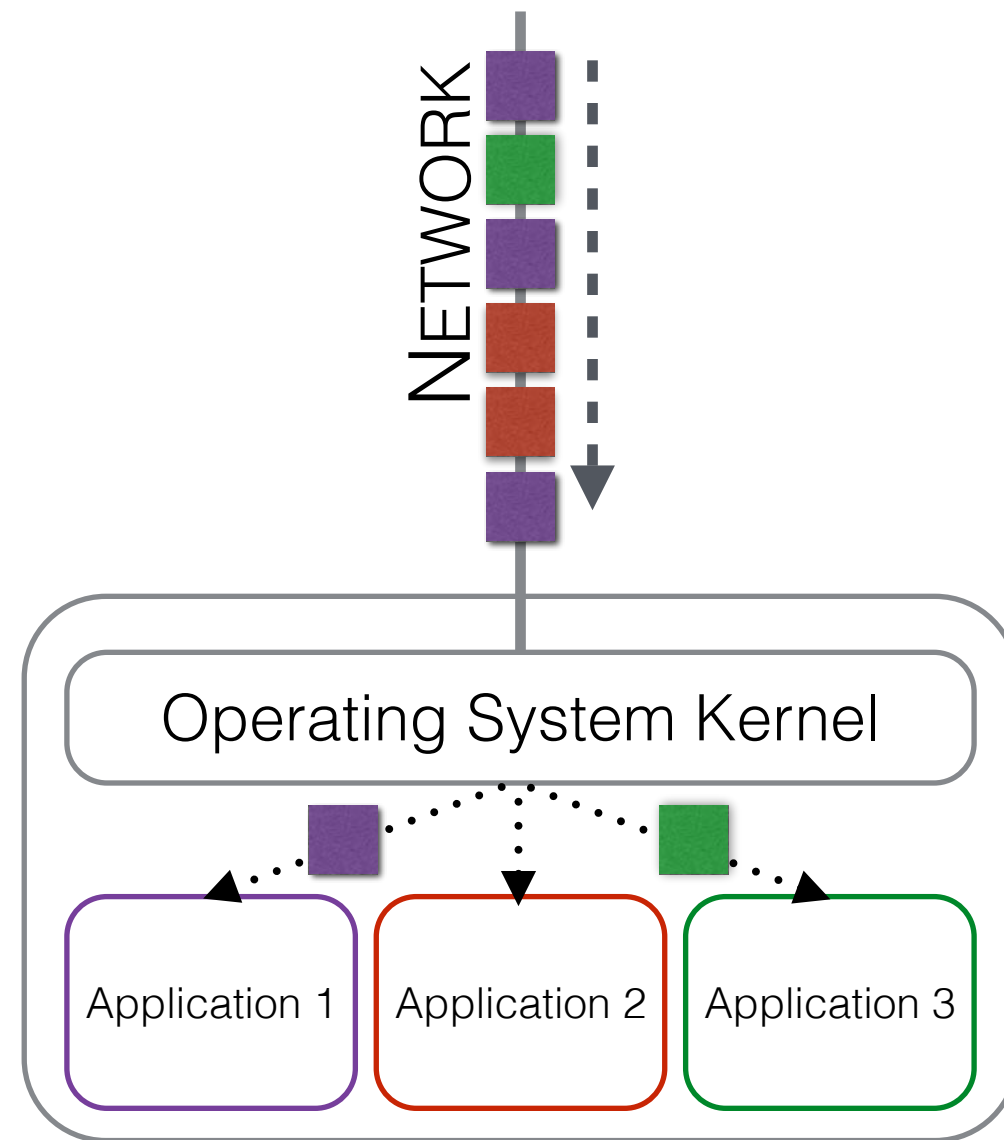
Systems Software

Operating Systems, ctd.



Systems Software

Operating Systems, ctd.



The kernel provides a common environment for applications

Systems Software

Compilers, Linkers, Assemblers

High-level Language

```
swap(int v[], int k)
{
    int temp;
    temp  = v[k];
    v[k]  = v[k+1];
    v[k+1] = temp;
}
```

Assembly Language

```
swap:
    multi $2, $5, 4
    add   $2, $4, $2
    lw    $15, 0($2)
    lw    $16, 0($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
    jr    $31
```

Machine Language

```
000000001010001000000000100011000
00000000100000100001000001000010
10001101111000100000000000000000
100011100001001000000000000000100
101011100001001000000000000000000
101011011110001000000000000000100
000000111110000000000000000001000
```

Compiler

Assembler

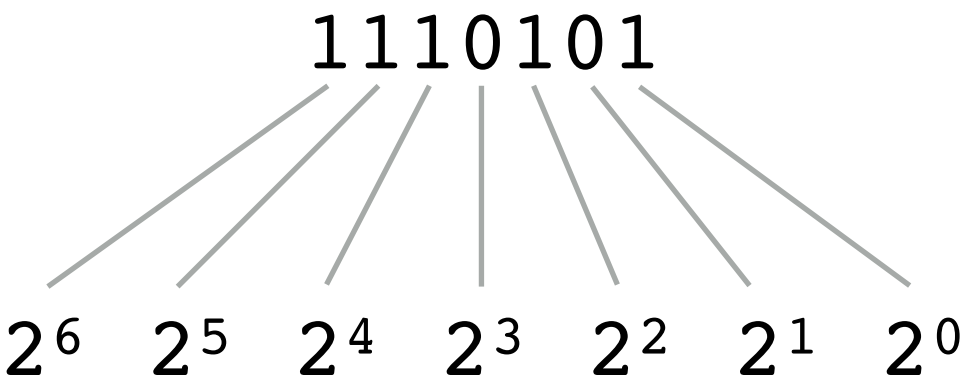


Compilers abstract the hardware

Base 10 (decimal) versus Base 2 (binary)

0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000
17	10001
18	10010
19	10011
20	10100
...	...
99	1100011
100	1100100
101	1100101

Base 10 (decimal) versus Base 2 (binary)



Convert to Decimal:

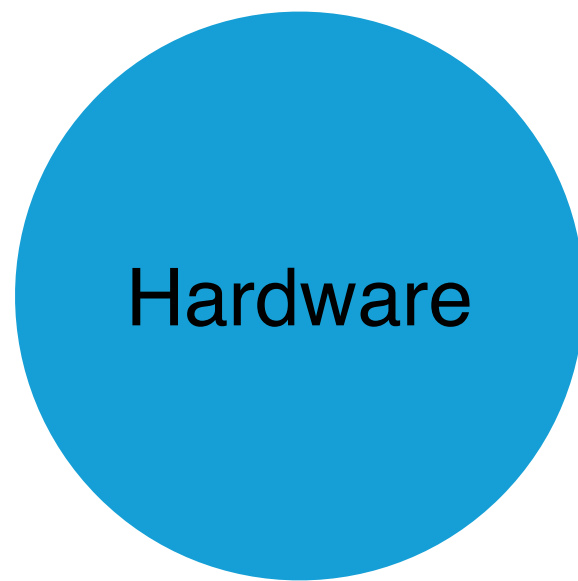
$$1x2^6 + 1x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 1x2^0$$

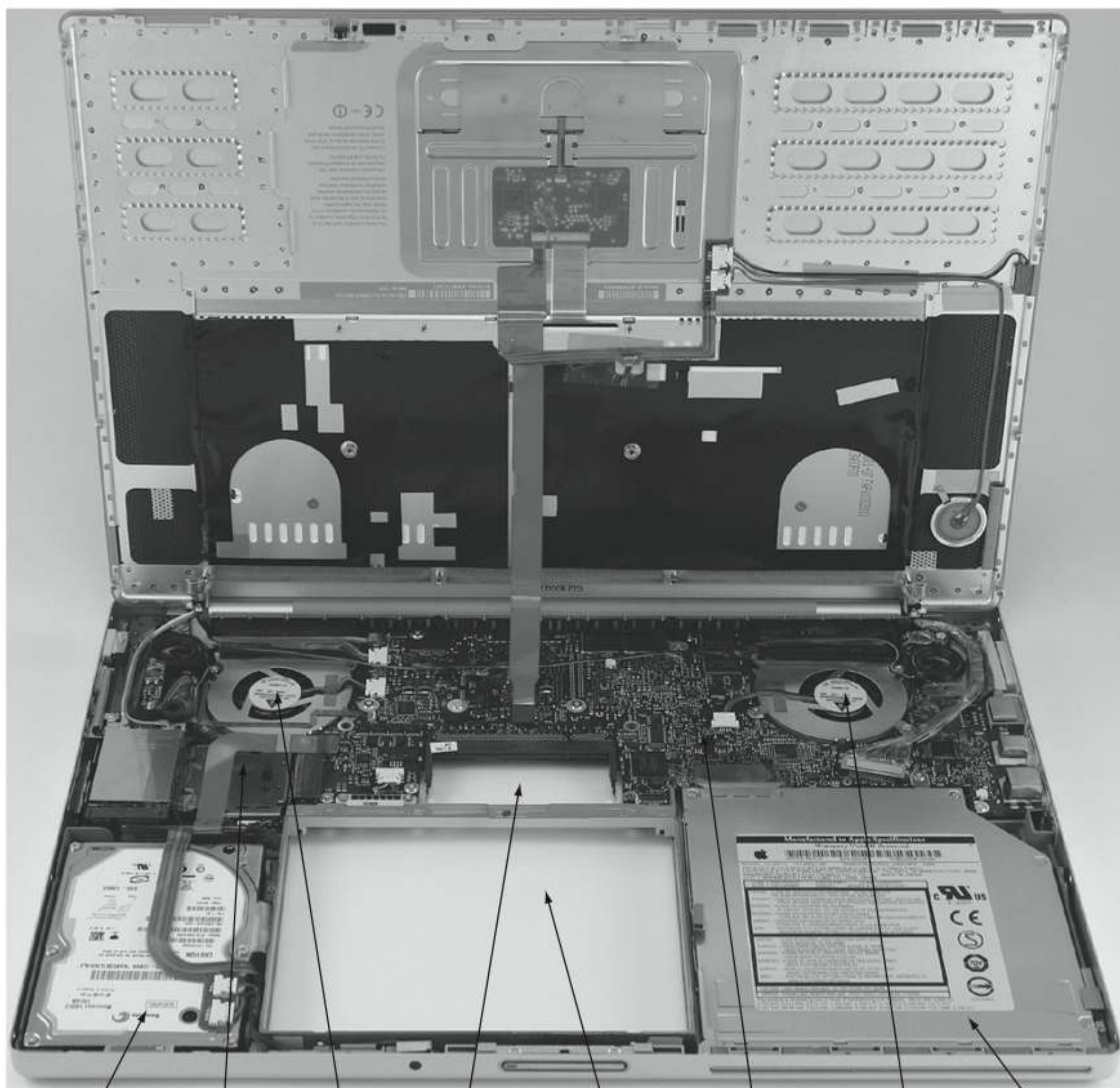
$$64 + 32 + 16 + 0 + 4 + 0 + 1$$

117

Base 10 (decimal) versus Base 2 (binary)

0	000000000000000000000000000000000000
1	0000000000000000000000000000000000001
2	0000000000000000000000000000000000010
3	0000000000000000000000000000000000011
4	00000000000000000000000000000000000100
5	00000000000000000000000000000000000101
6	00000000000000000000000000000000000110
7	00000000000000000000000000000000000111
8	000000000000000000000000000000000001000
9	000000000000000000000000000000000001001
10	000000000000000000000000000000000001010
11	000000000000000000000000000000000001011
12	000000000000000000000000000000000001100
13	000000000000000000000000000000000001101
14	000000000000000000000000000000000001110
15	000000000000000000000000000000000001111
16	0000000000000000000000000000000000010000
17	0000000000000000000000000000000000010001
18	0000000000000000000000000000000000010010
19	0000000000000000000000000000000000010011
20	0000000000000000000000000000000000010100
...	
99	000000000000000000000000000000000001100011
100	000000000000000000000000000000000001100100
101	000000000000000000000000000000000001100101





Hard drive

Processor

Fan with
cover

Spot for
memory
DIMMs

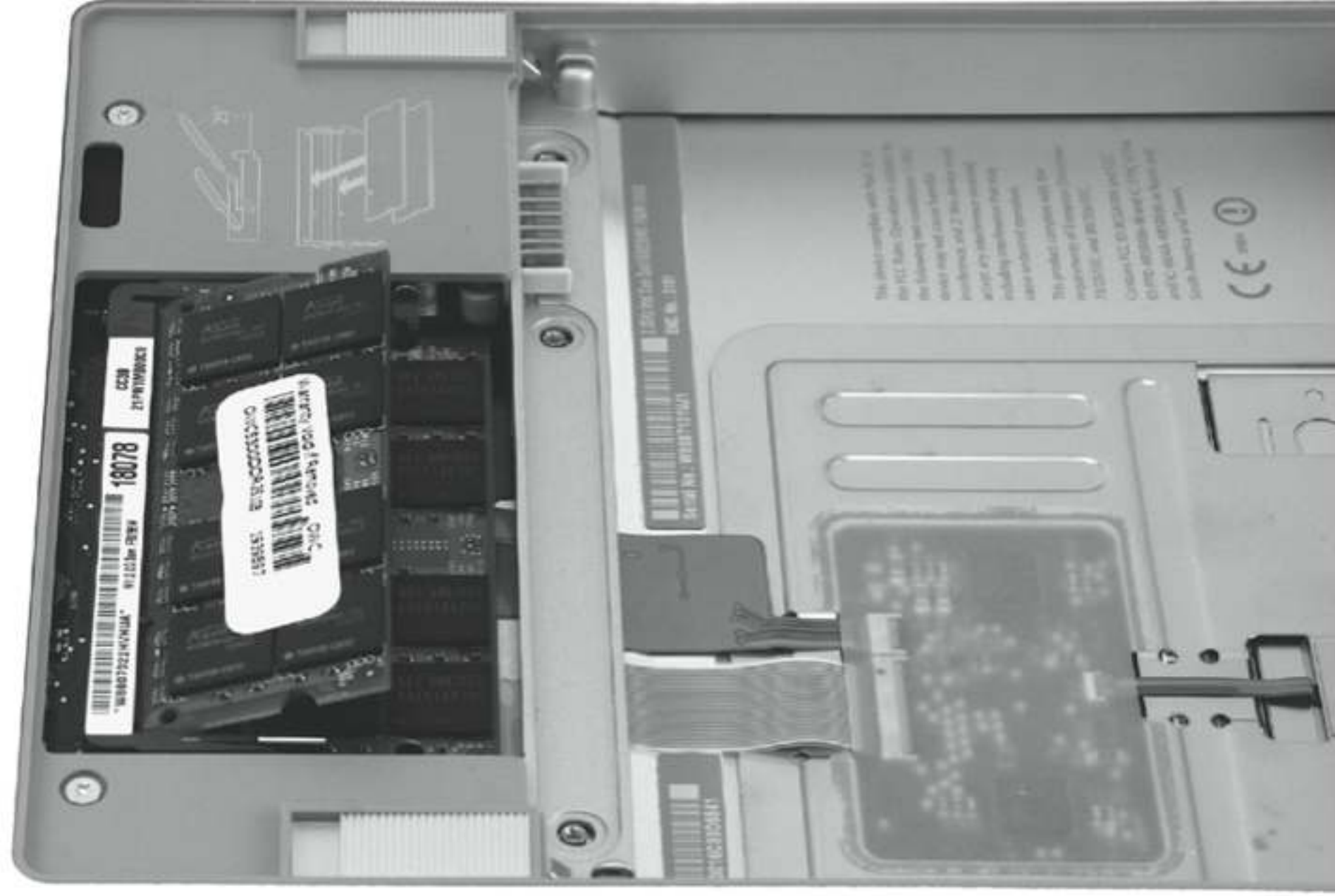
Spot for
battery

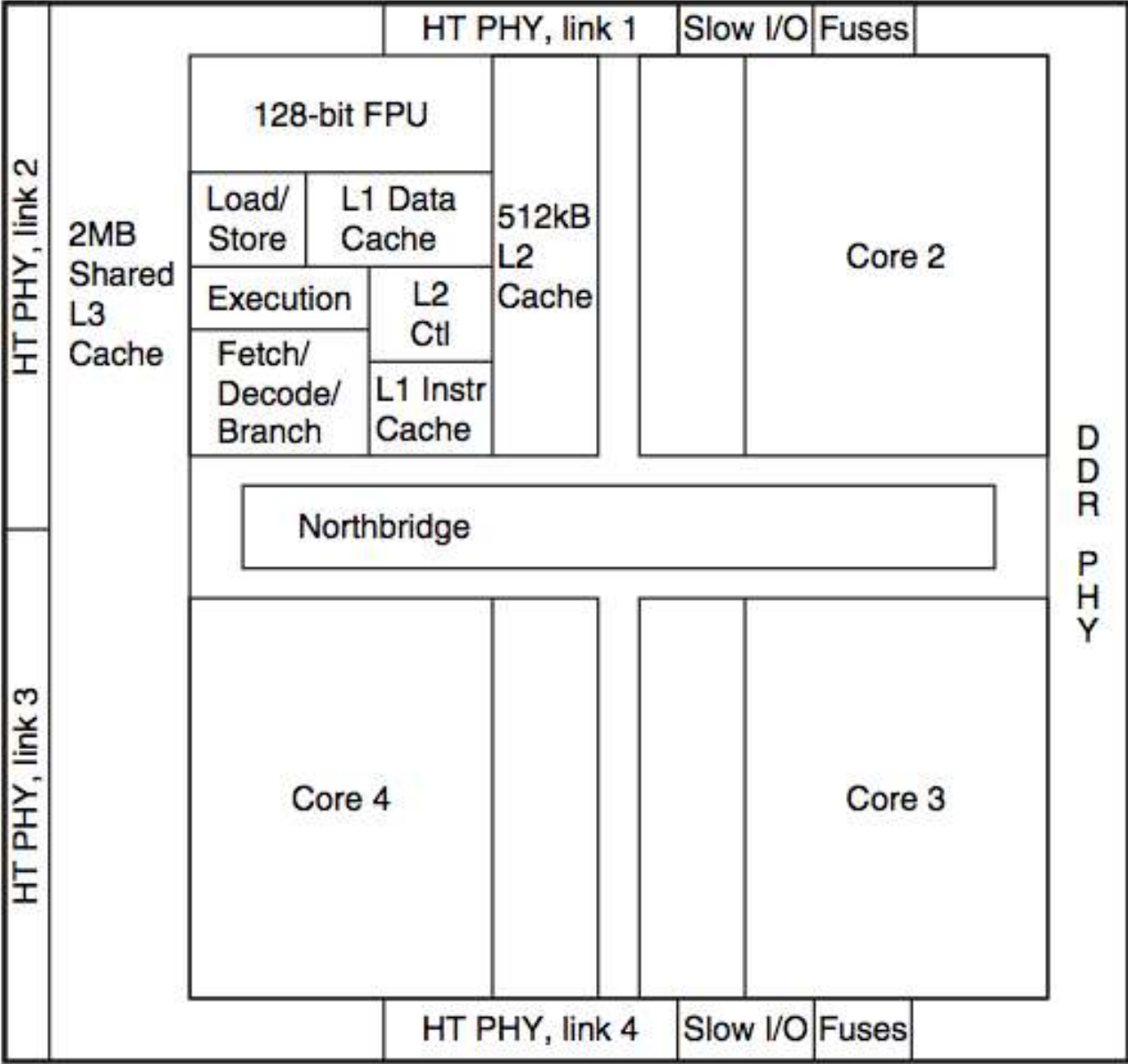
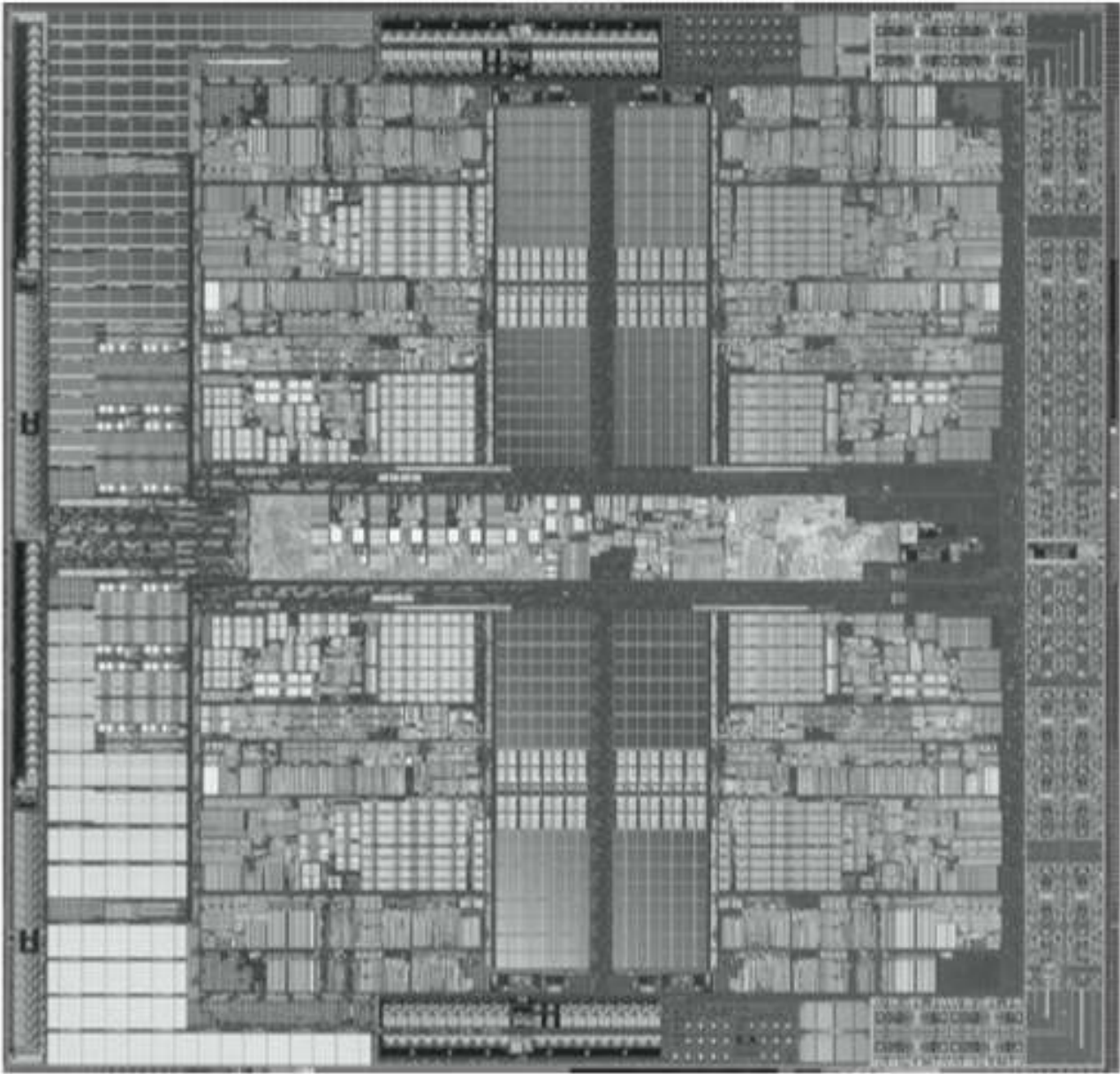
Motherboard

Fan with
cover

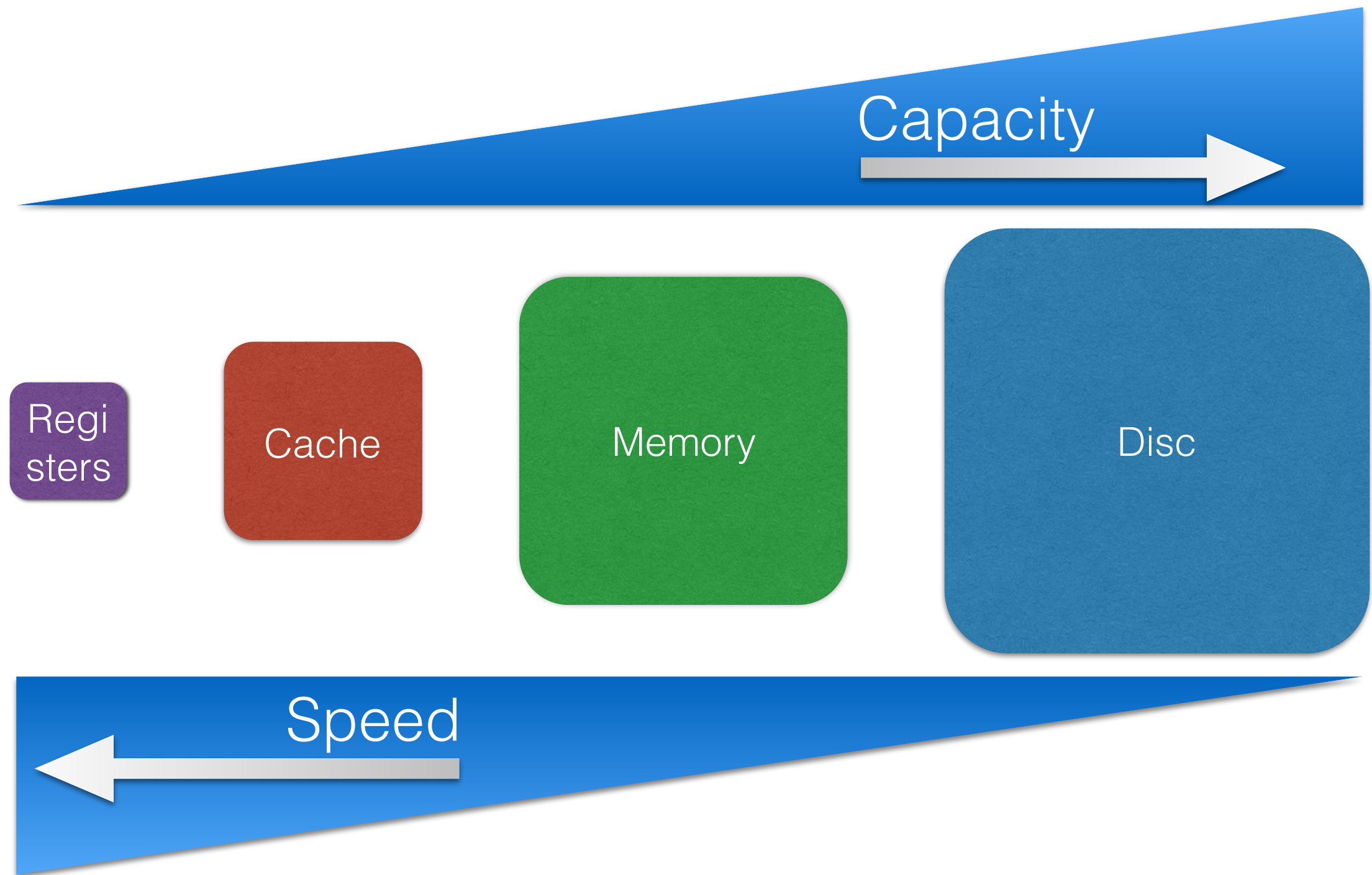
DVD drive



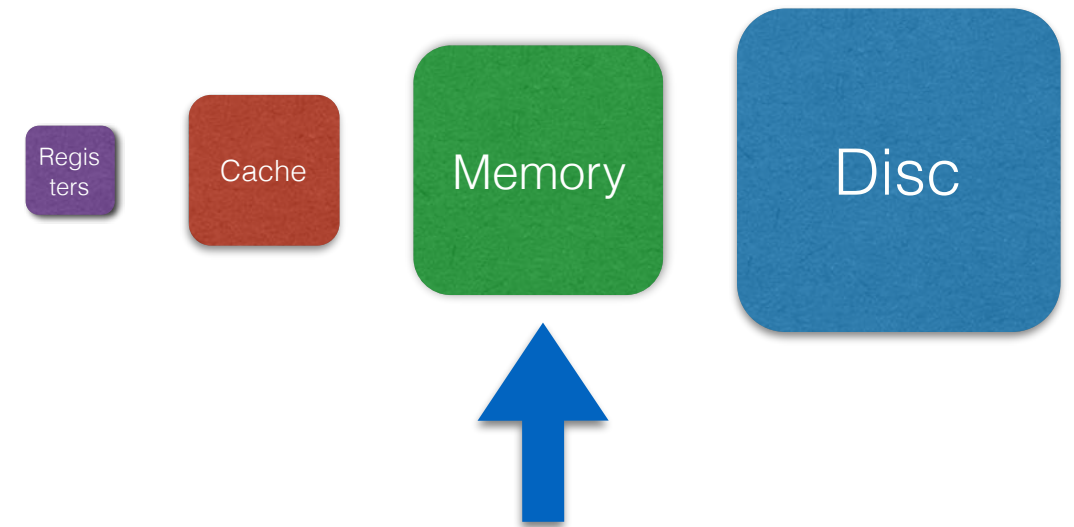




Storing code and data



- Disk access time: 5–20 milliseconds
- Memory access time: 50–70 nanoseconds— 100,000 times faster
- The cost per gigabyte of disk is 30 to 100 times less expensive than memory



1011010101010000

10110101010100000110100100100101011000101011010101010000011010010010010101100010...

Memory: RAM - Random Access Memory

32-bit computer: address fields are 32 bits wide

allows 2^{32} possible addresses, or 4Gb

machines are now 64-bit, allowing memories up to 2^{64}

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

<http://www.cs.mcgill.ca/~jvybihal/Courses/CS273/8086%20Registers.JPG>



10110101010100001011010101010000

Registers have a fixed size and a fixed count

Some registers are 'general purpose' some are specialized

an instruction format

Translating a MIPS Assembly Instruction into a Machine Instruction

Let's do the next step in the refinement of the MIPS language as an example. We'll show the real MIPS language version of the instruction represented symbolically as

```
add $t0,$s1,$s2
```

first as a combination of decimal numbers and then of binary numbers.

The decimal representation is

0	17	18	8	0	32
---	----	----	---	---	----

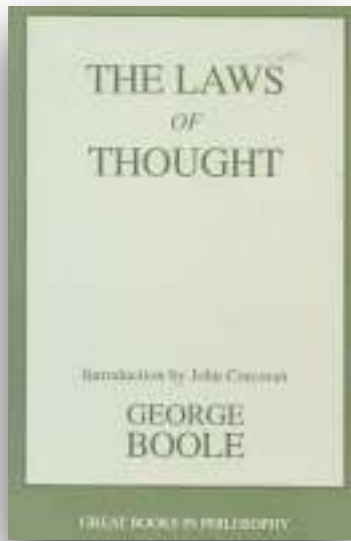
Each of these segments of an instruction is called a *field*. The first and last fields (containing 0 and 32 in this case) in combination tell the MIPS computer that this instruction performs addition. The second field gives the number of the register that is the first source operand of the addition operation ($17 = \$s1$), and the third field gives the other source operand for the addition ($18 = \$s2$). The fourth field contains the number of the register that is to receive the sum ($8 = \$t0$). The fifth field is unused in this instruction, so it is set to 0. Thus, this instruction adds register $\$s1$ to register $\$s2$ and places the sum in register $\$t0$.

This instruction can also be represented as fields of binary numbers as opposed to decimal:

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Base 10 (decimal) versus Base 2 (binary)

[illegible]



1854, “a Mathematical language dealing with the questions of logic”

Boolean Logic / Boolean Algebra

True = 1
False = 0

AND operation: * or &

A * B

1 * 1 = True
1 * 0 = False
0 * 1 = False
0 * 0 = False

OR operation: + or |

A + B

1 + 1 = True
1 + 0 = True
0 + 1 = True
0 + 0 = False

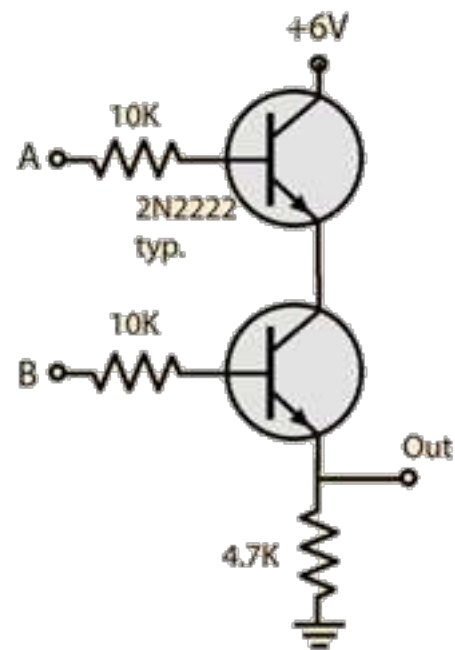
NOT operation: ^

$\bar{A}$

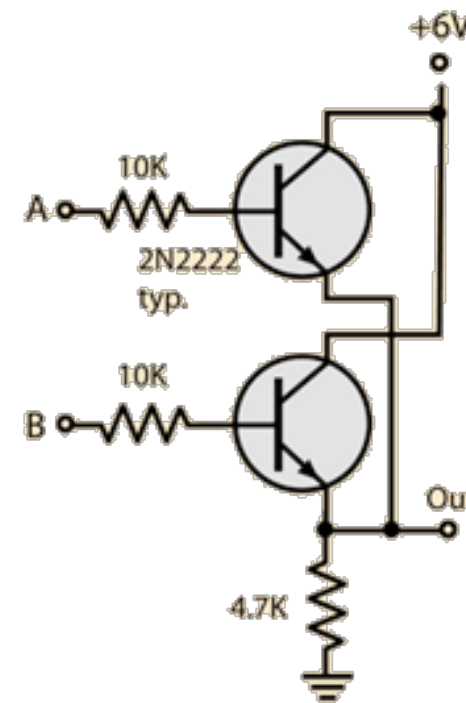
1 = False
0 = True

Boolean logic, ctd.

AND gate



OR gate



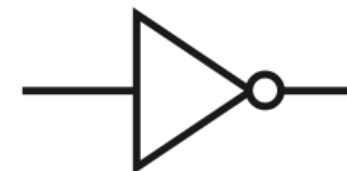
AND



OR



NOT



How does computation happen?

Addition: 6 + 7

C code

```
i = 6  
j = 7  
g = i + j;
```

Assembly

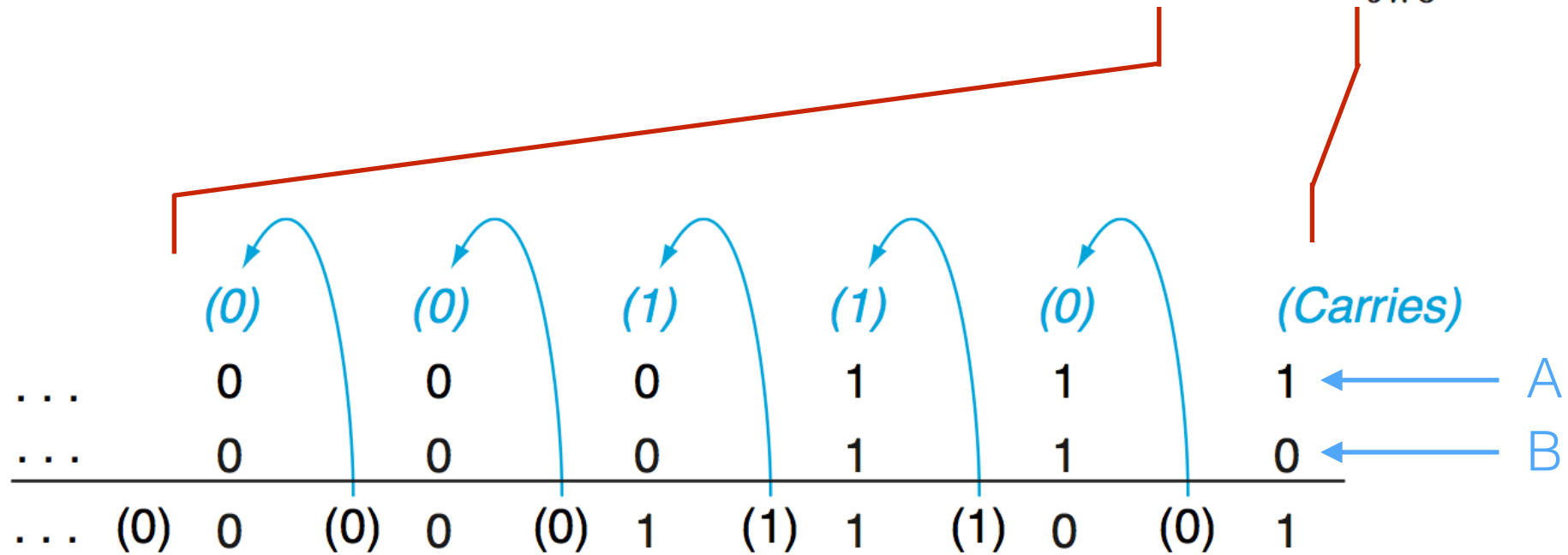
```
lw    $t0, 4($s3)  
lw    $t1, 8($s3)  
add   $t0, $t1, $t2  
sw    $t2, 48($s3)
```

Machine code

```
10100001110000100010101000001100  
10101001010000000010101000001001  
01100001100010000010100001101000  
00011001110000010010101011001000
```

Binary Addition

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\
 + \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\
 \hline
 = \quad 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}}
 \end{array}$$



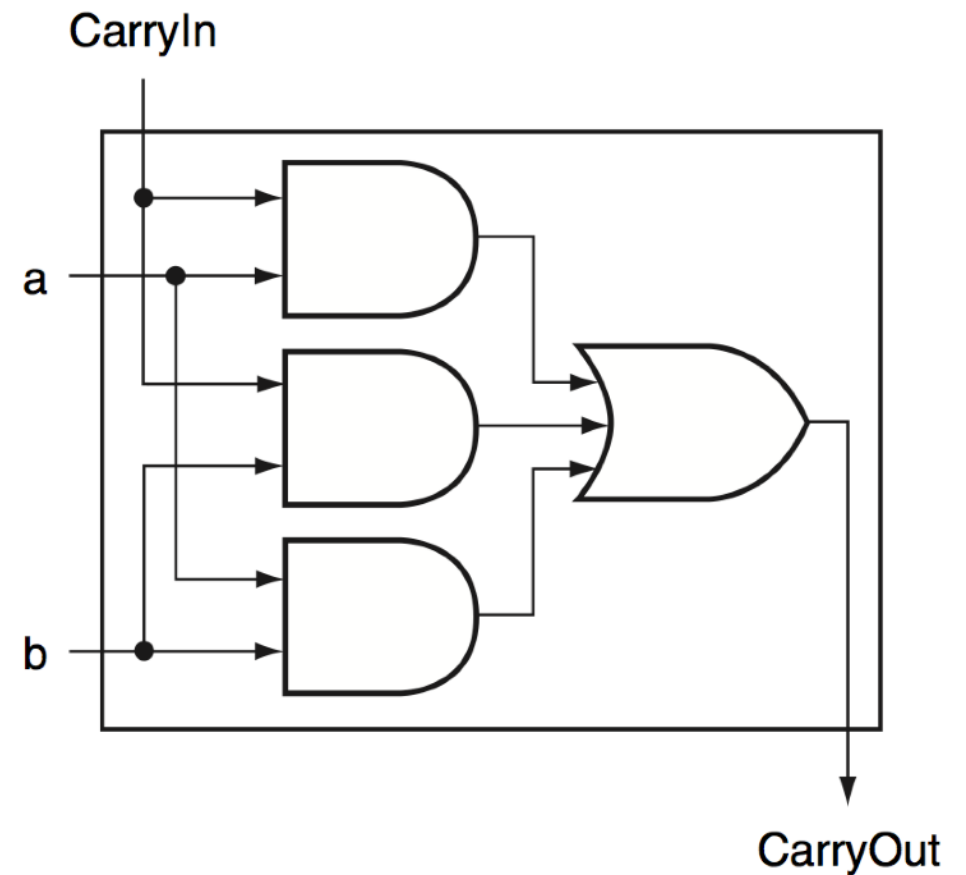
Inputs			Outputs	
A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Binary Addition, ctd.



Inputs			Outputs	
A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\text{CarryOut} = (b * \text{CarryIn}) + (a * \text{CarryIn}) + (a * b)$$



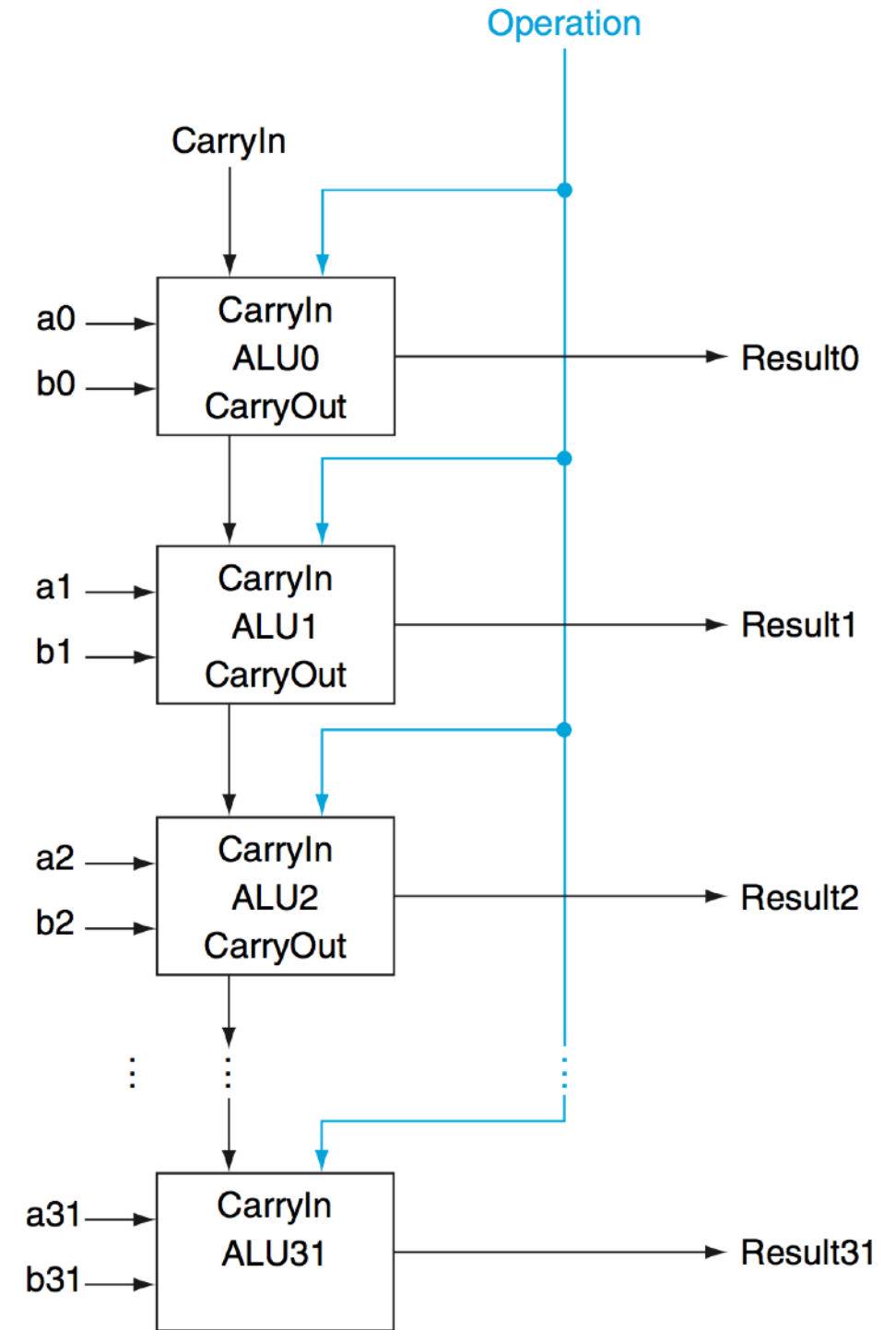
Binary Addition, ctd.

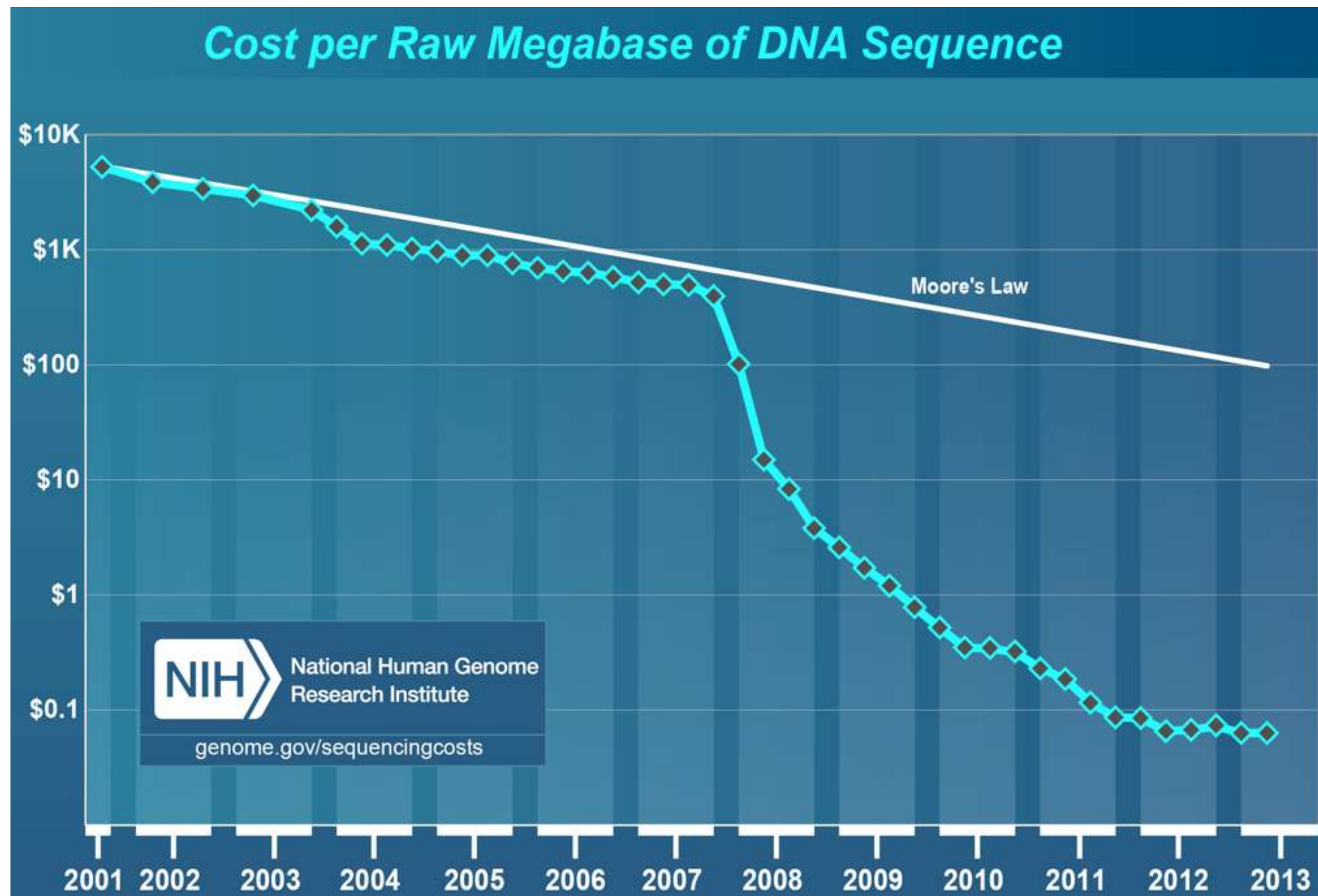
Inputs			Outputs	
A	B	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

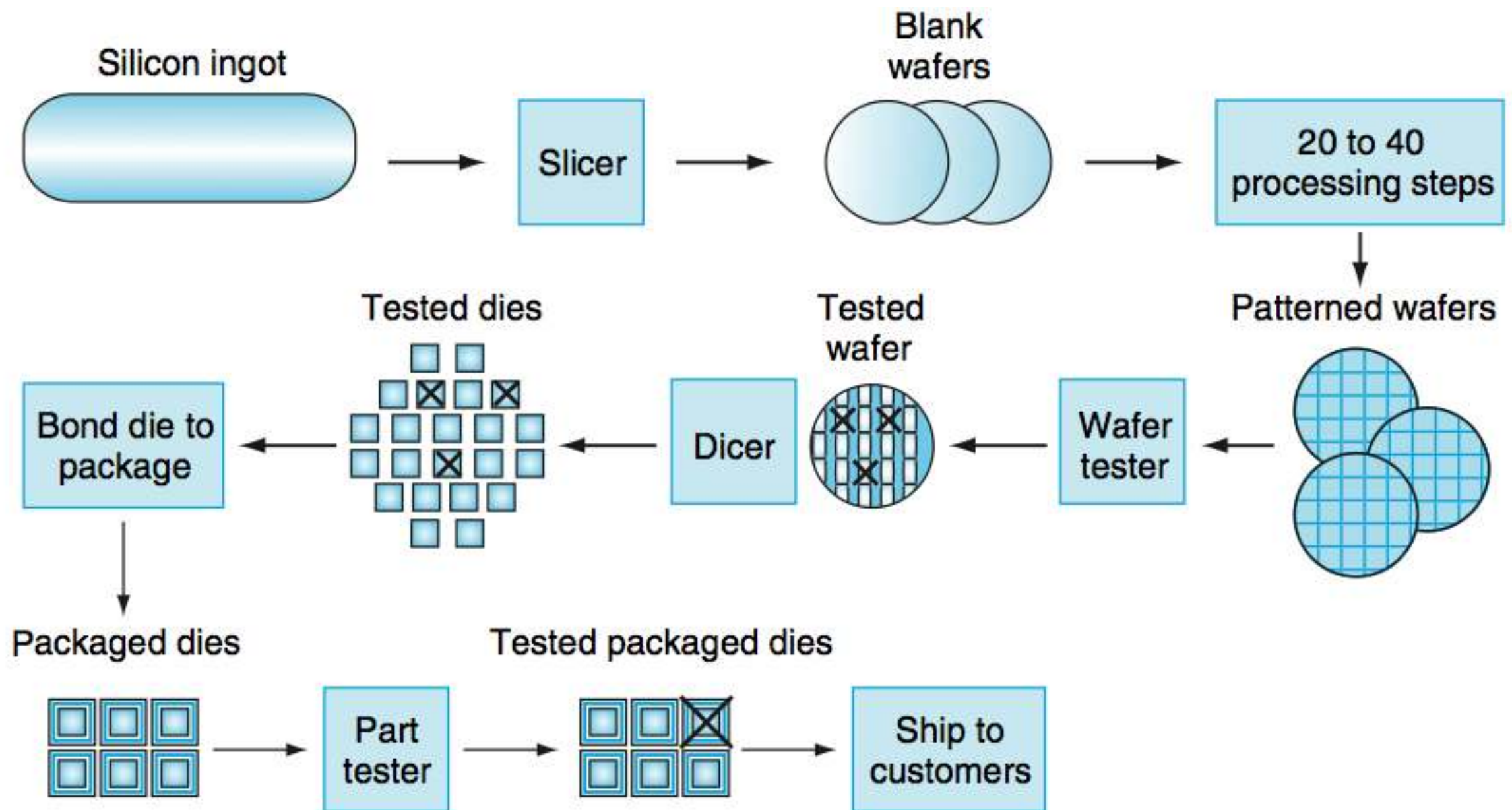
Binary Addition, ctd.

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{\text{two}} = 7_{\text{ten}} \\ +\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{\text{two}} = 6_{\text{ten}} \\ \hline =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{\text{two}} = 13_{\text{ten}} \end{array}$$

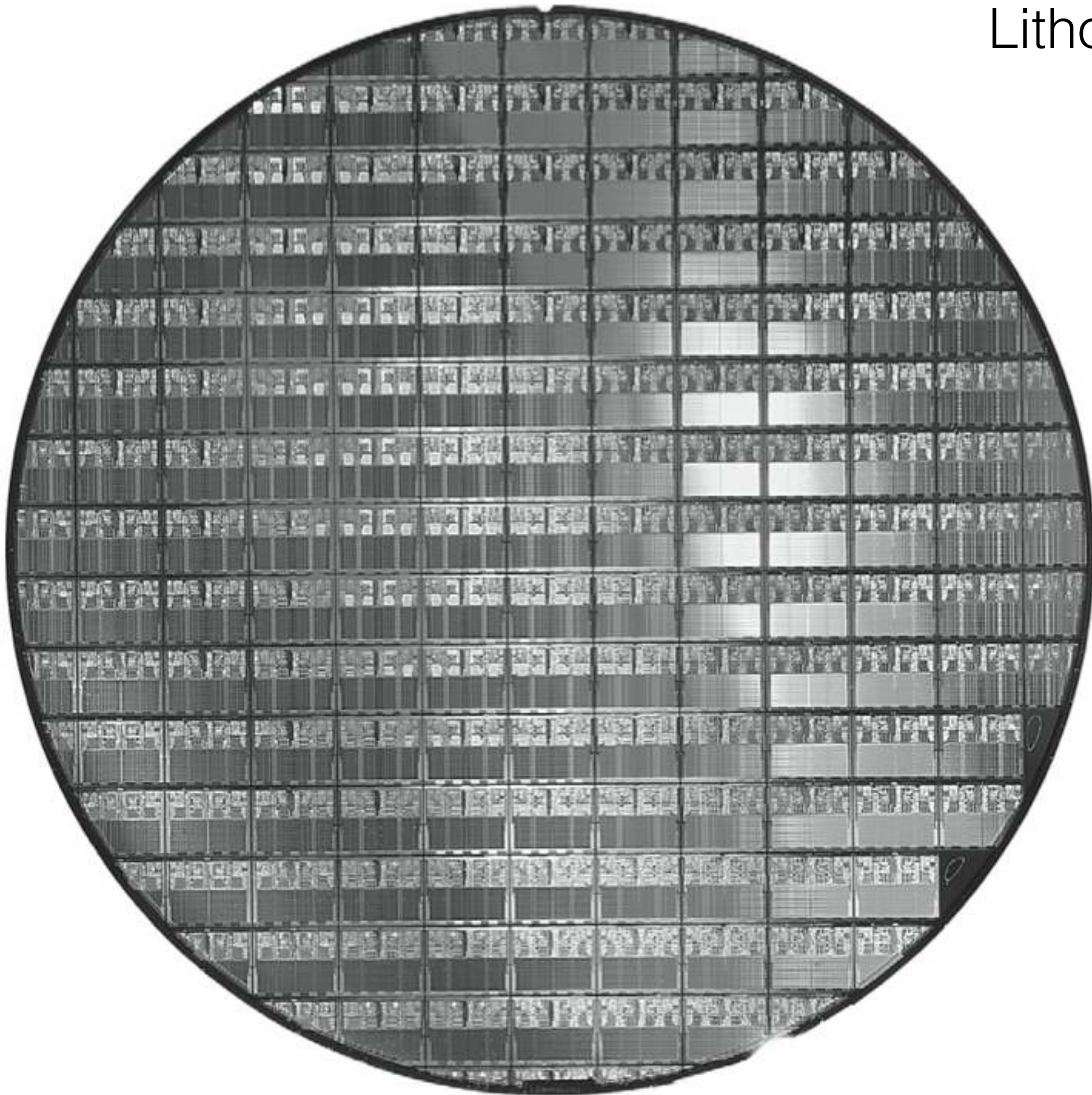




- Some people say it takes 18 months and others say 24
- Some interpret the law to be about the doubling of processing power, not the number of transistors.
- A self-fulfilling prophecy than an actual law, principle or observation: power constraints / parallelism



Lithography



Developments in High Throughput Sequencing

