Information, Computation, Communication

Computer Architecture

Question

- Now that we have created algorithms, how can we **construct systems that execute them**?

sum of first *n* integers
Input: n *Output: m*

s ← 0
while n > 0
  s ← s + n
  n ← n – 1
m ← s

From Algorithms to Computers

sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
    s ← s + n
    n ← n – 1
m ← s
```

sum of first *n* integers

Input: r1
Output: r2

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

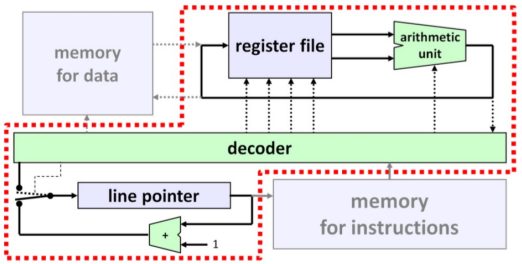
sum of first *n* integers

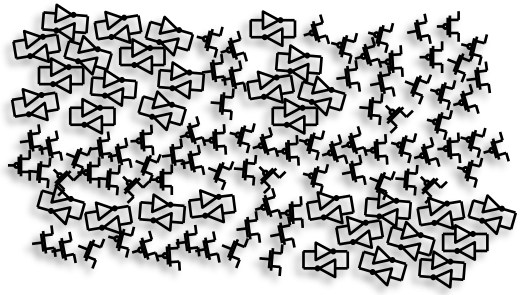
Input: r1
Output: r2

```
1: 0100010100000000
2: 0101100000001010
3: 0001001001011010
4: 0001010101000000
5: 0000101010101111
6: 0101000001100101
```

Software

Hardware





From Algorithms to Computers

Step 0

We will start with the algorithms that you have studied..

Step 1

… write them in terms of a few basic instructions..

Step 3

… and present them in a way a computer can understand.

Software

Hardware

Step 2

In parallel, we will create an abstract machine..

Step 4

…that we will implement with transistors.

From Algorithms to Computers (Step 0)

- In order to describe the idea of an algorithm, we use pseudo code.
- Let's consider an example

sum
Input: n *Output: m*

s ← 0
while n > 0
  s ← s + n
  n ← n – 1
m ← s

From Algorithms to Computers (Step 1)

sum of first *n* integers
Input: n *Output: m*

s ← 0
while n > 0
  s ← s + n
  n ← n – 1
m ← s

sum of first *n* integers
Input: r1 *Output: r2*

`1: copy r3, 0`
`2: jump_egz r1, 6`
`3: add r3, r3, r1`
`4: add r1, r1, -1`
`5: jump 2`
`6: copy r2, r3`

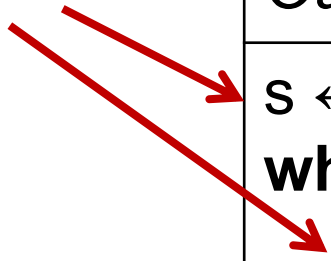
Software

Hardware

We will rewrite algorithm in terms of a few basic instructions, which will have physical counter parts (hardware to perform them).

Rewrite Algorithm using Basic Instructions

Our machine needs to be able to remember values, e.g., value for s



sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
    s ← s + n
    n ← n – 1
m ← s
```

- In all computers, values are store in so-called **registers**, which are physical implementations of variables with a fix number of bits (usually 32 or 64 bits)

Registers

- A **register** is the physical implementation of the notion of variable
- A machine usually has a small number of registers (a few dozen)
- For large data (array, list,..) we will use external memory (RAM)
- Registers are usually represented by **`r1`**, **`r2`**, **`r3`**,…
- We replace all arbitrary variable names with register names
  - **`n`** ➔ ***r1***
  - **`m`** ➔ ***r2***
  - **`s`** ➔ ***r3***

Step 1.1

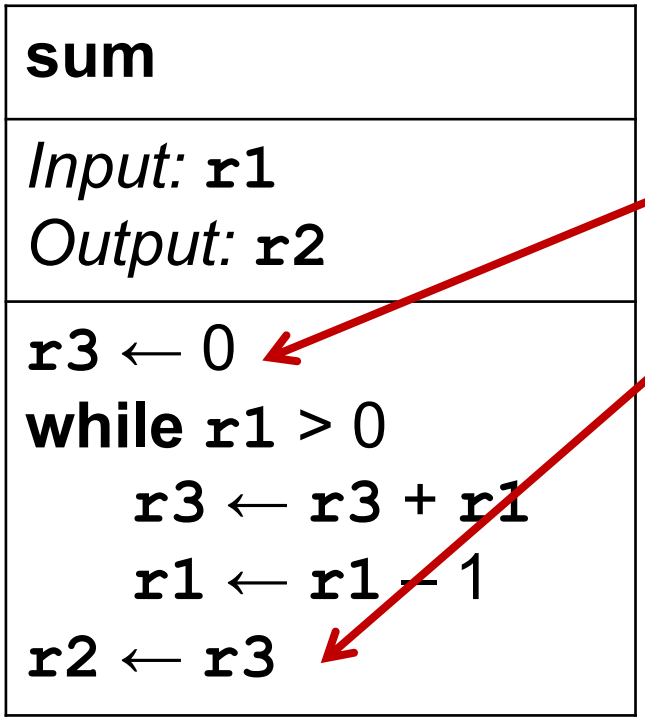
sum
Input: n ➔ `r1` *Output: m* ➔ `r2`

s ← 0 ➔ `r3`
while n > 0
  s ← s + n
  n ← n – 1
m ← s

sum
Input: `r1` *Output:* `r2`

`r3` ← 0
while `r1` > 0
  `r3` ← `r3` + `r1`
  `r1` ← `r1` – 1
`r2` ← `r3`

Next…



We will need to assign values to registers

We will use a basic instruction, e.g.,
"copy r3, 0"
for r3 ← 0
or
"copy r2, r3"
for r2 ← r3

Step 1.2

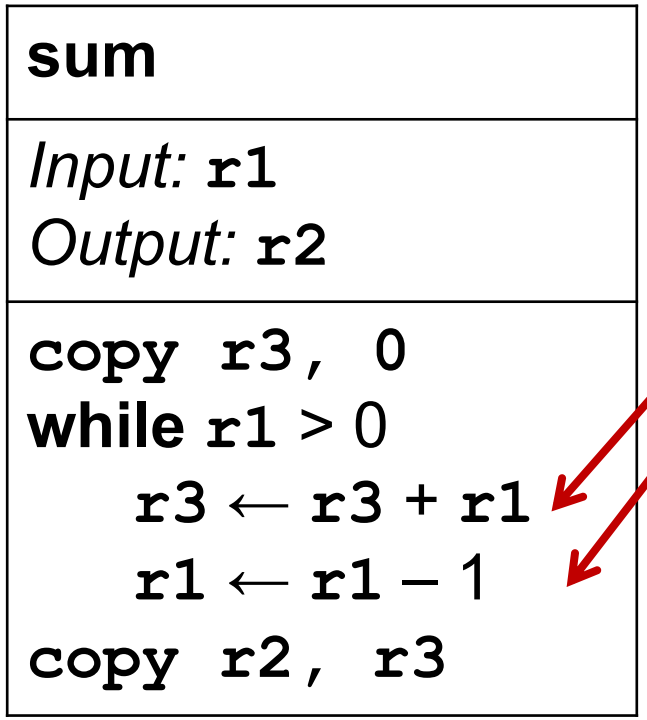
sum
Input: **r1** *Output:* **r2**

r3 ← 0
while r1 > 0
  r3 ← **r3** + **r1**
  r1 ← **r1** – 1
r2 ← **r3**

sum
Input: **r1** *Output:* **r2**

copy r3, 0
while r1 > 0
  r3 ← **r3** + **r1**
  r1 ← **r1** – 1
copy r2, r3

Next…



We will need to assign new values to registers after applying arithmetic operations

We will use basic instructions, e.g., "`add r3, r3, r1`" for `r3 ← r3 + r1`

Step 1.3

sum of first *n* integers

Input: `r1`
Output: `r2`

```
copy r3, 0
while r1 > 0
    r3 ← r3 + r1
    r1 ← r1 – 1
copy r2, r3
```

sum of first *n* integers

Input: `r1`
Output: `r2`

```
copy r3, 0
while r1 > 0
    add r3, r3, r1
    add r1, r1, -1
copy r2, r3
```

Basic Instructions

- There is a limited number of instructions, e.g.,
  - copy for assignment
  - add for addition
  - mul for multiplication
- All instructions (i) have a single result, (ii) take one or two registers (or constants) as operands
- Instructions are written in the following form:
  `name destination, operand1, operand2`
- Every computation in an algorithm is rewritten in these basic instructions, e.g.,
  a ← a * (b + c) with a in r1, b in r2 and c in r3 could be rewritten as.

```
add r2, r2, r3
mul r1, r1, r2
```

Next…

sum

Input: `r1`
Output: `r2`

```
copy r3, 0
while r1 > 0
    add r3, r3, r1
    add r1, r1, -1
copy r2, r3
```

- All control structures (if-conditions, while-loop, for-loops, ..) will be replaced by **jumps to labels**
- We will use **line numbers as labels**
- We will have a few different (conditional) jump instructions, e.g.,
  - `jump 2:`
    `always jump to line 2`
  - `jump_egz r1, 6:`
    `jump to line 6, if r1 is equal to zero`
  - `jump_eg r1, r2, 6:`
    `jump to line 6, if r1 is equal to r2`

Step 1.4

sum

Input: `r1`
Output: `r2`

```
copy r3, 0
while r1 > 0
    add r3, r3, r1
    add r1, r1, -1

copy r2, r3
```

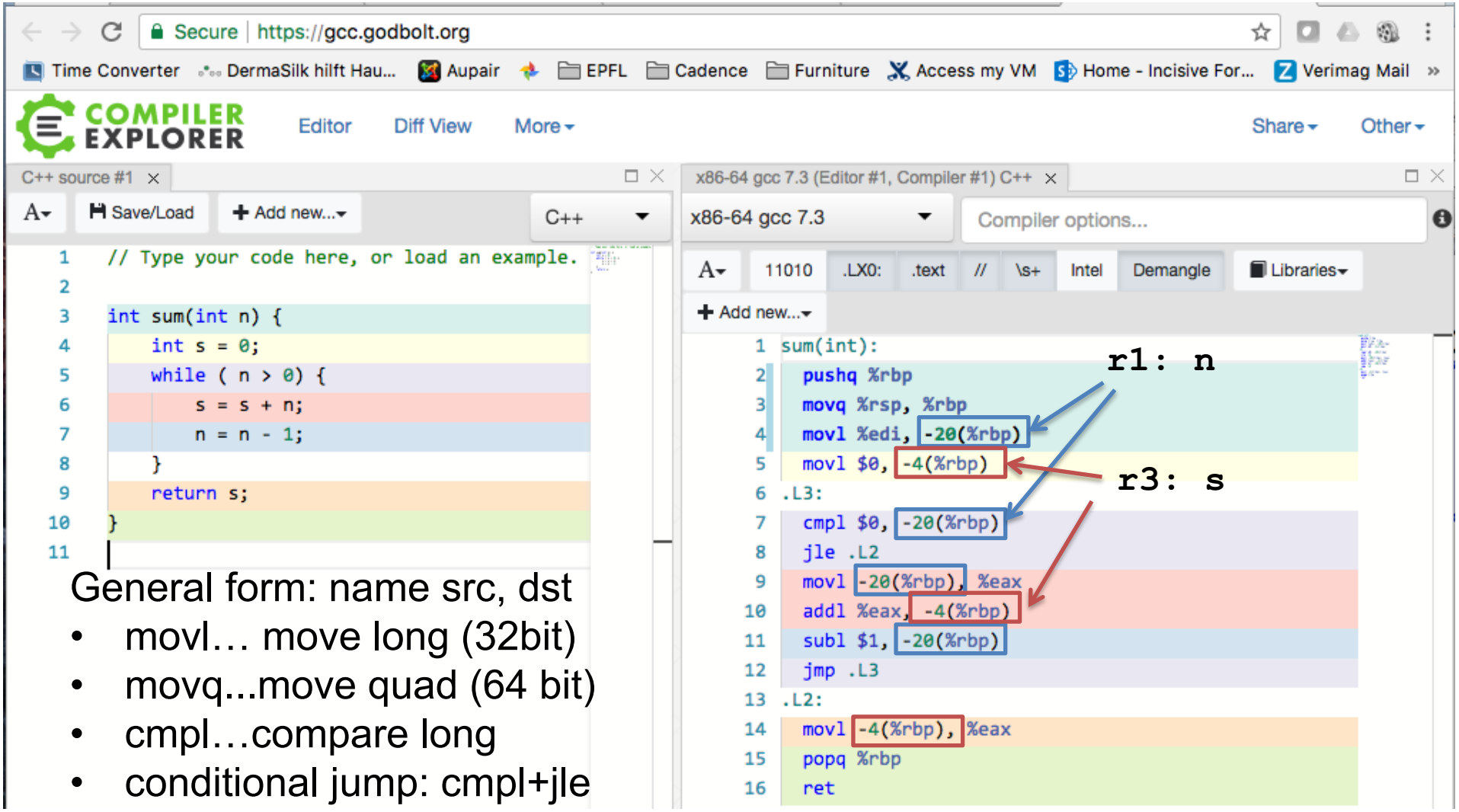
sum

Input: `r1`
Output: `r2`

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

This is a program in **assembly (or assembler) language**.

Example in x86 Assembly Language (Intel)



General form: name src, dst

- movl… move long (32bit)
- movq...move quad (64 bit)
- cmpl…compare long
- conditional jump: cmpl+jle

Try it yourself: g++ -S sum.cc -o sum.a

Example in x86 Assembly Language (Intel)

- Example in C

```
#include <stdio.h>

int main () {

  int a = 10;
  int b = 45;
  int add, mul;

  __asm__ ( "addl  %%ebx, %%eax;" : "=a" (add) : "
  __asm__ ( "imull %%ebx, %%eax;" : "=a" (mul) : "


  printf("Add = %d \n", add);
  printf("Mul = %d \n", mul);
}
```

```
Compile with: g++ assembly.c –o assembly –g
Run: ./assembly
Add = 55
Mul = 450
```

g++ -S assembly.cc -o assembly.a

```
_main:
        .cfi_startproc
## %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        pushq   %rbx
        subq    $24, %rsp
        .cfi_offset %rbx, -24
        movl    $10, -12(%rbp)
        movl    $45, -16(%rbp)
        movl    -12(%rbp), %eax
        movl    -16(%rbp), %ebx
        ## InlineAsm Start
        addl    %ebx, %eax

        ## InlineAsm End
        movl    %eax, -20(%rbp)
        movl    -12(%rbp), %eax
        movl    -16(%rbp), %ebx
        ## InlineAsm Start
        imull   %ebx, %eax
```

Summary

- We use **“registers”** to mimic variables in hardware
- We rewrite our program in terms of basic **“instructions”**
  - instructions to load/copy values into a registers
  - instructions for arithmetic (and binary) operations
  - instructions to jump to another instruction under some condition
- We use a restricted set of previously defined instructions
- E.g., ARM or Intel processors have their own set of instructions (x86, ARM, Risc-V ISA – instruction set architecture)

From Algorithms to Computers (Step 2)

sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
    s ← s + n
    n ← n – 1
m ← s
```

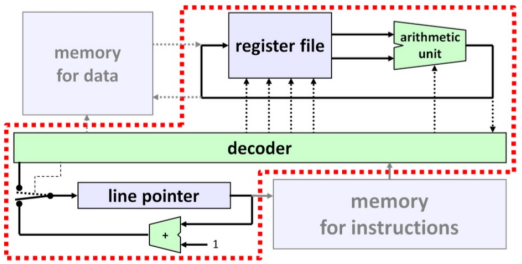
sum of first *n* integers

Input: r1
Output: r2

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

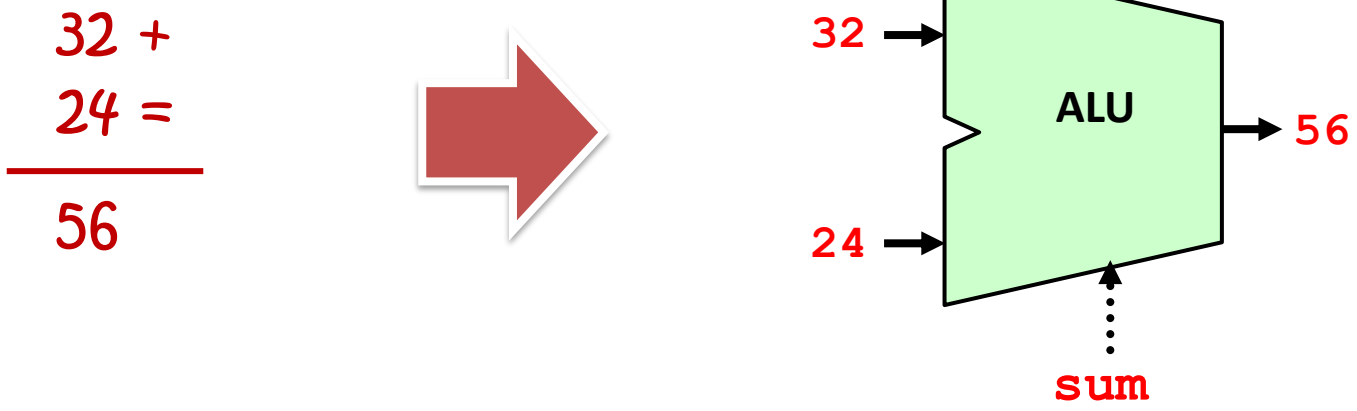
Software

Hardware



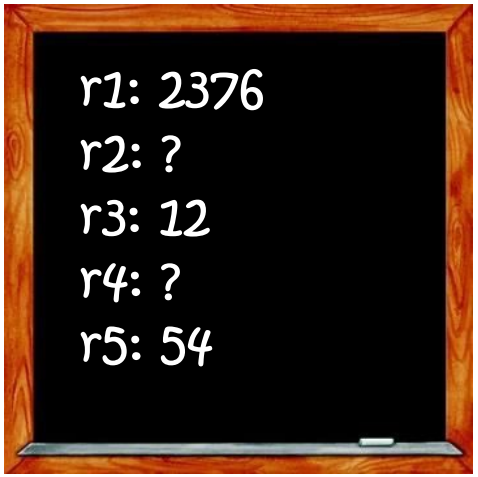
What do we need to calculate?

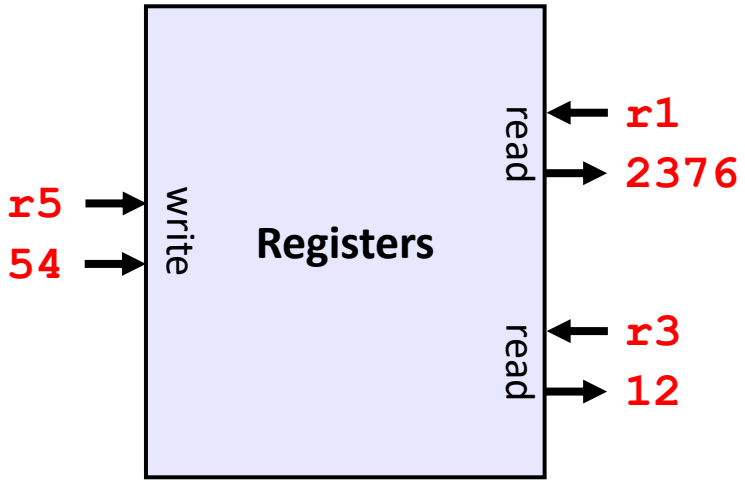
- **Arithmetic logic unit (ALU)** for arithmetic (and bitwise) operations (+,-,*,/,mod, &, |, ^,<<,>>,..)



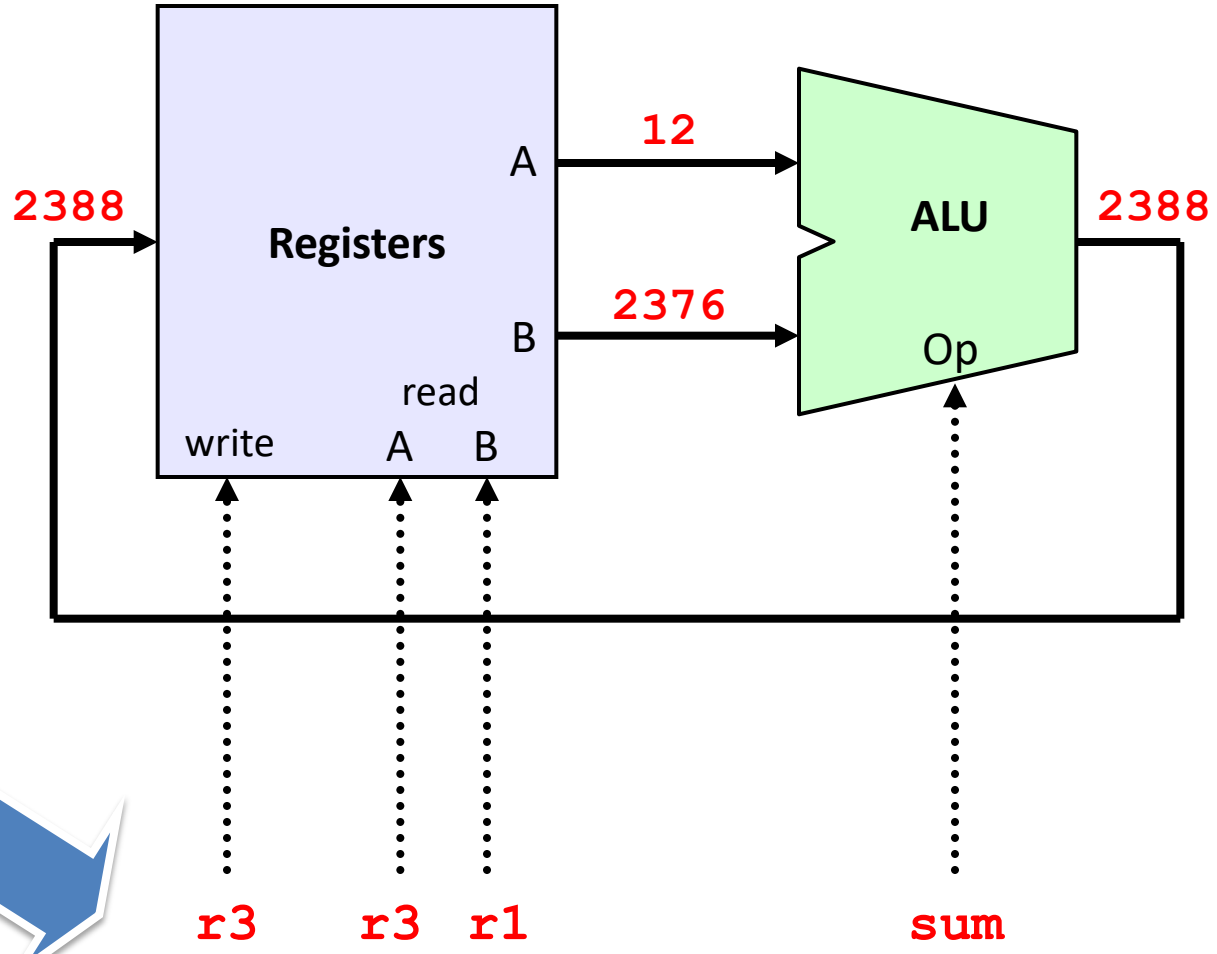
What do we need to calculate?

- **Registers** to save the values of the operands and the result





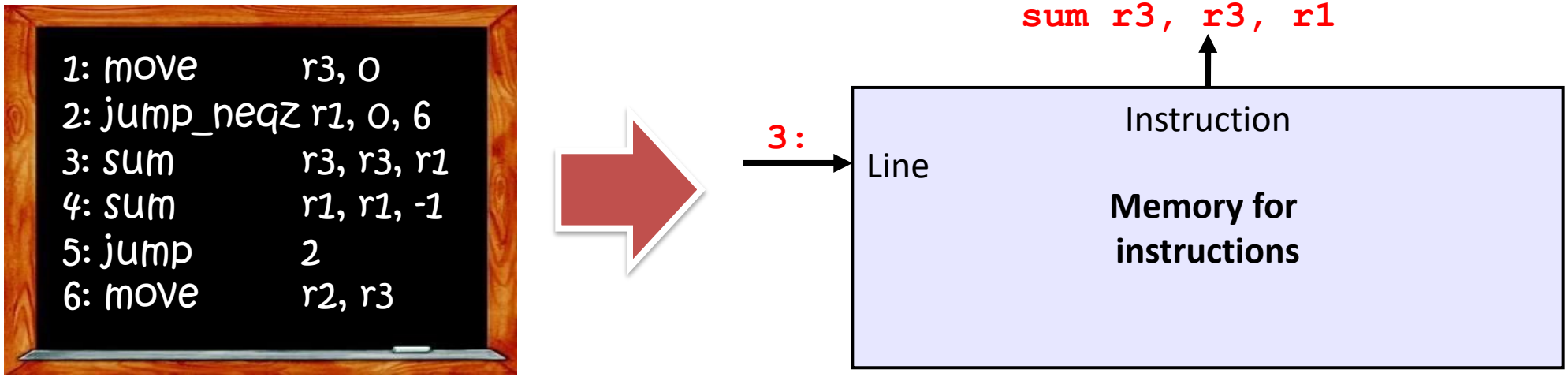
A Circuit to Calculate



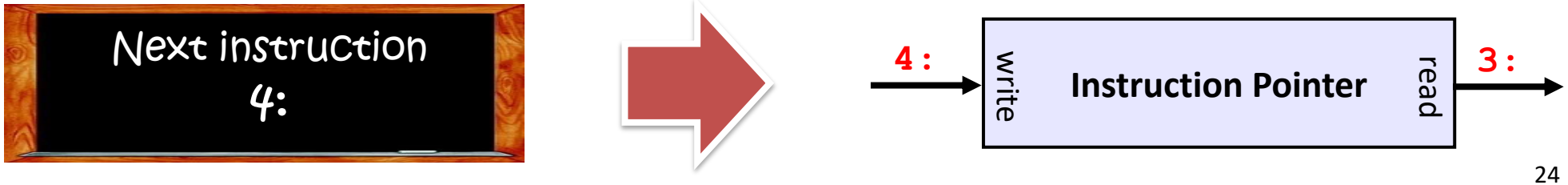

```
sum r3, r3, r1
```

What else do we need?

- Our algorithm or program needs to be stored somewhere

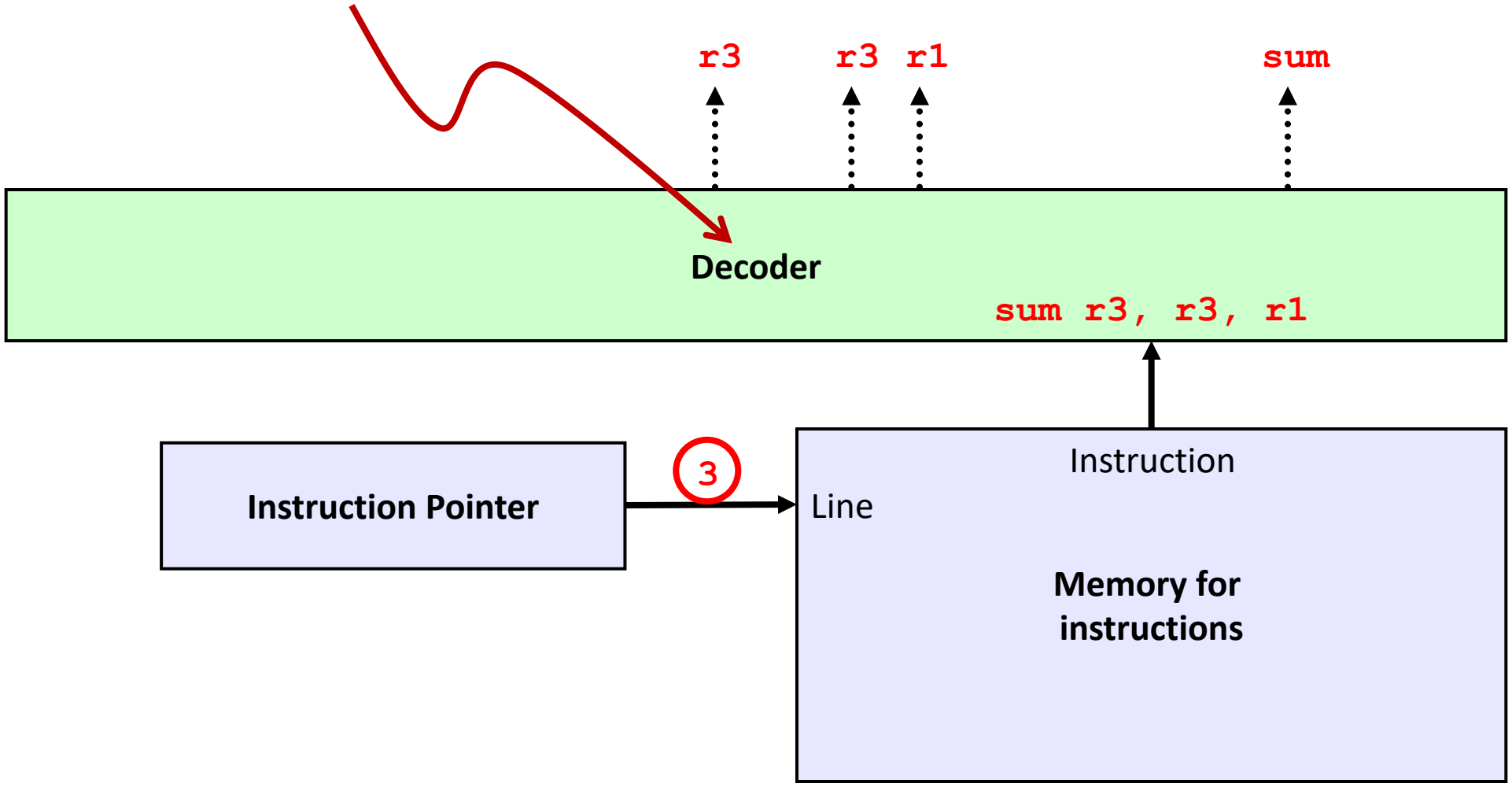


- We need a way to control where we are



How to control where we are?

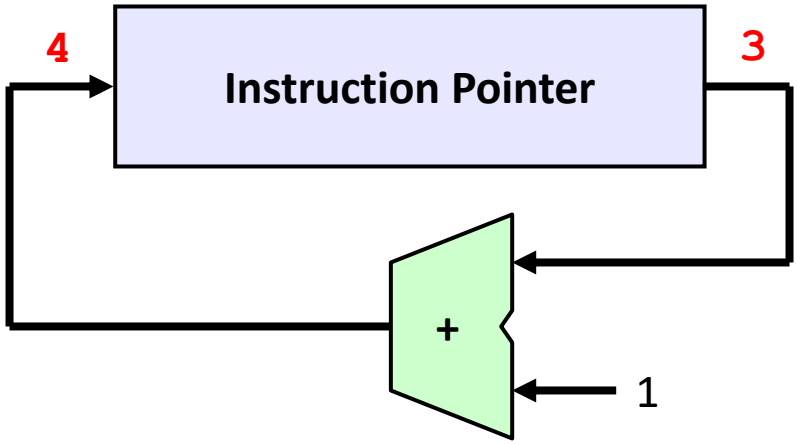
A simple circuit that separates the elements in an instruction



How to control where we are?

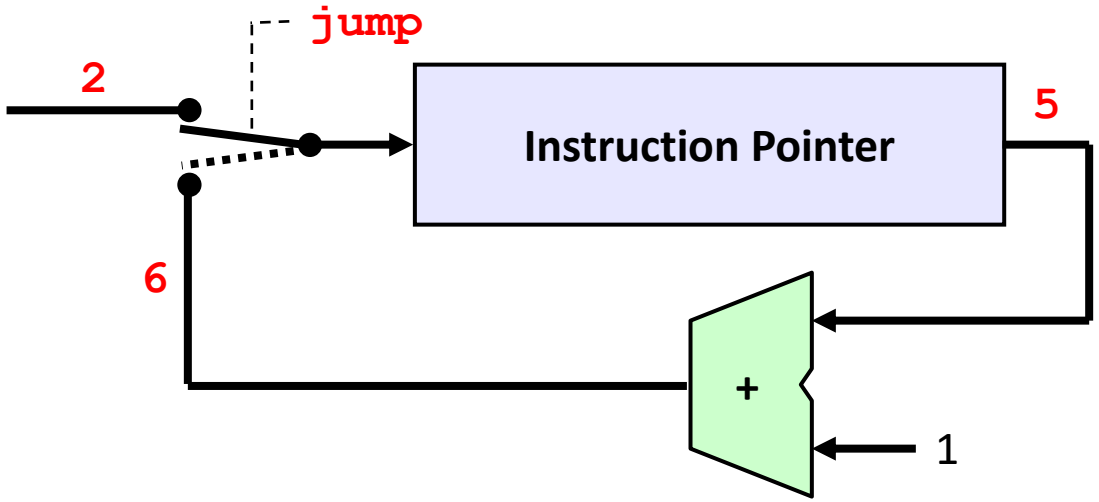
- Usually we pass to the next line

```
3: sum    r3, r3, r1
```

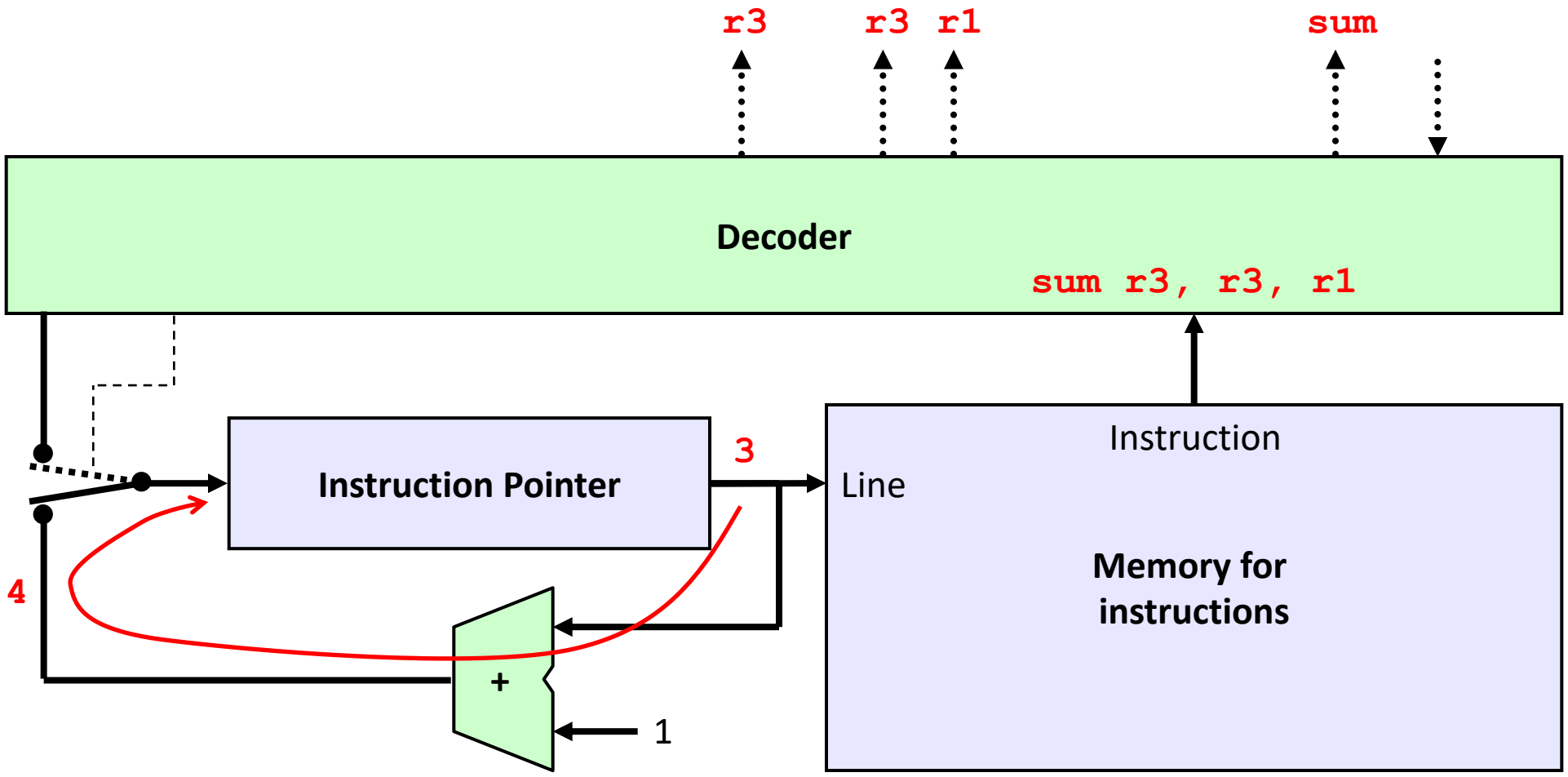


- If we get an instruction `jump`, we switch to the given line

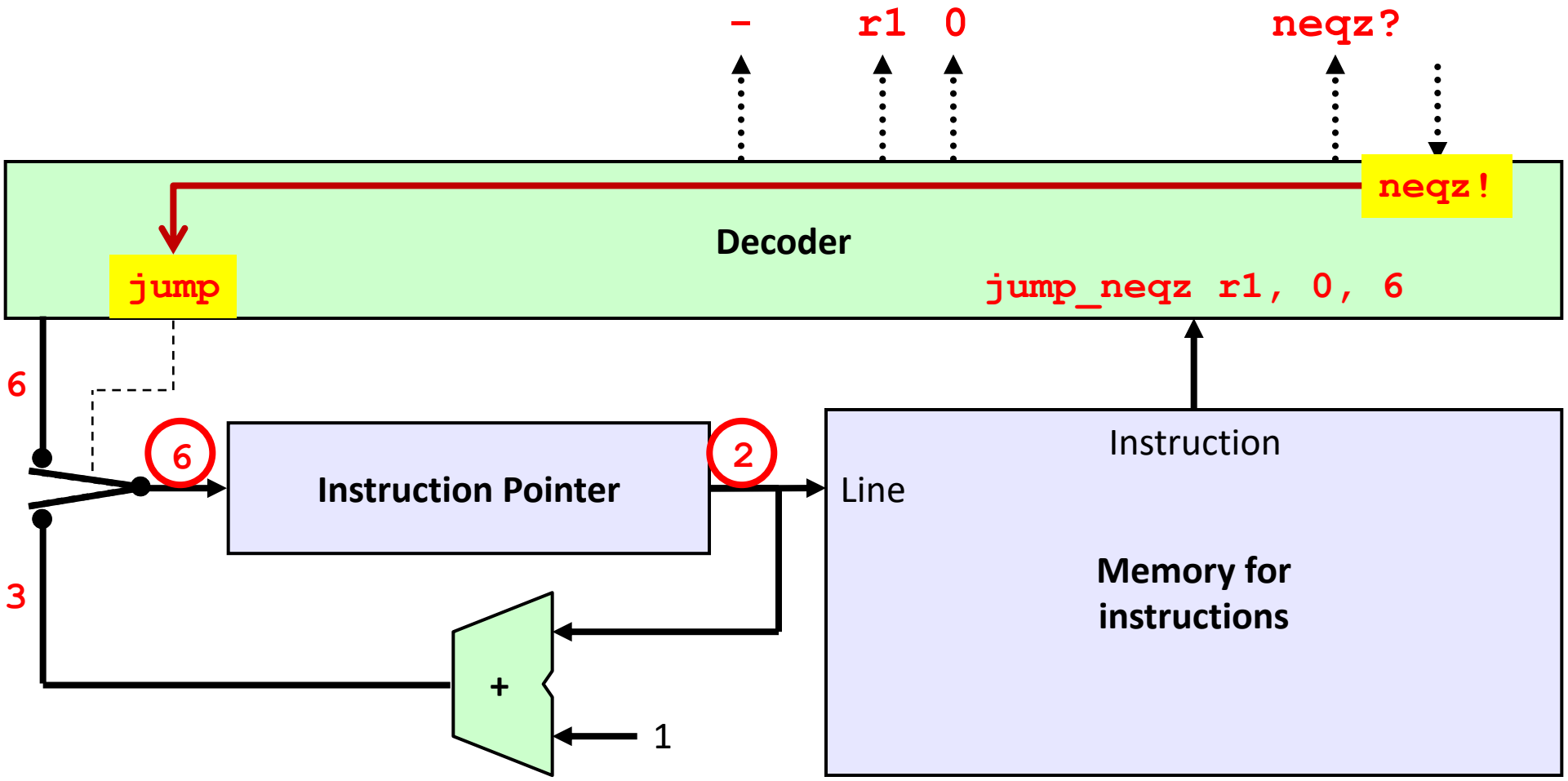
```
5: jump 2
```



A Circuit to Control the Instruction Pointer

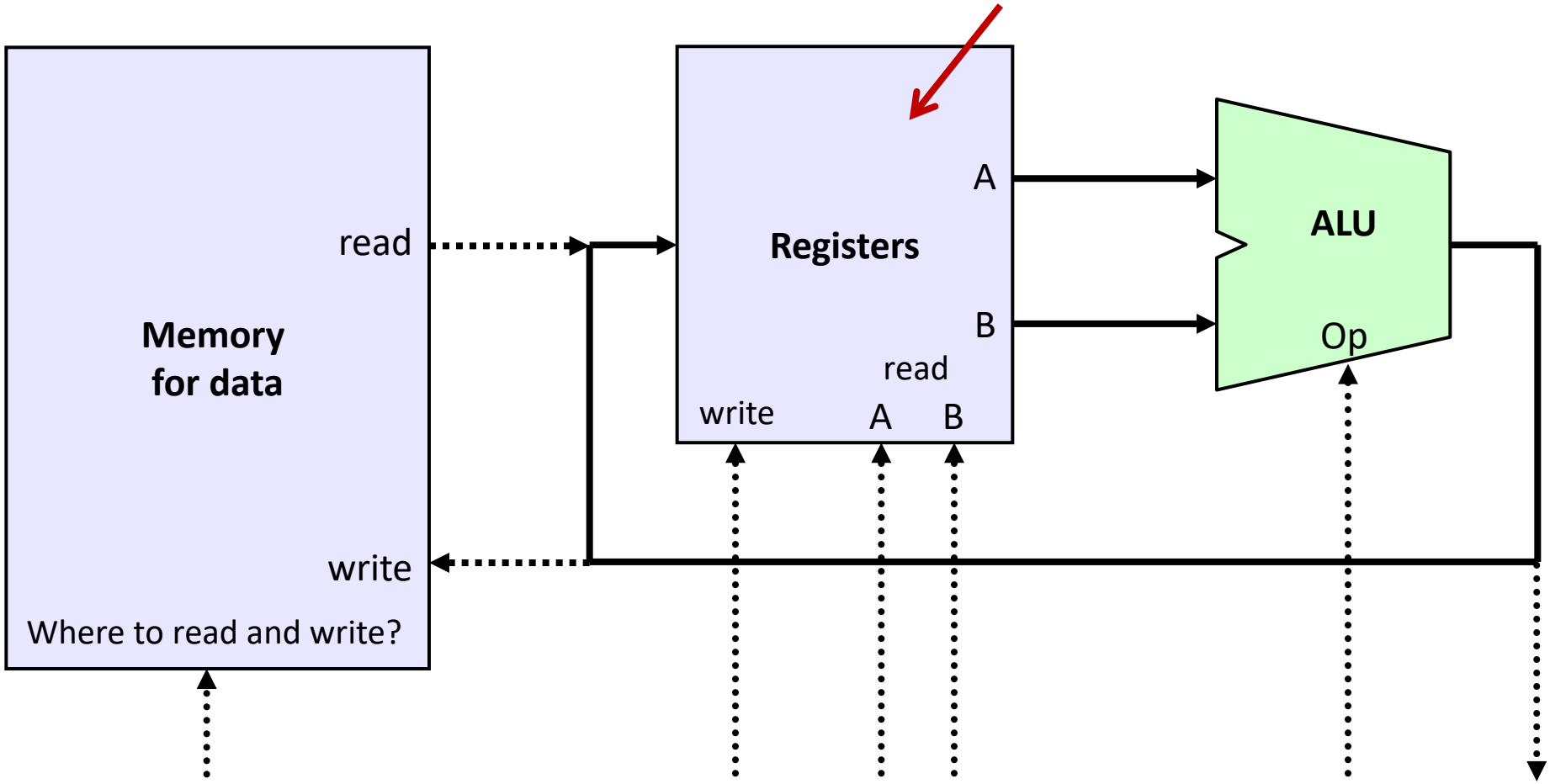


A Circuit to Control the Instruction Pointer

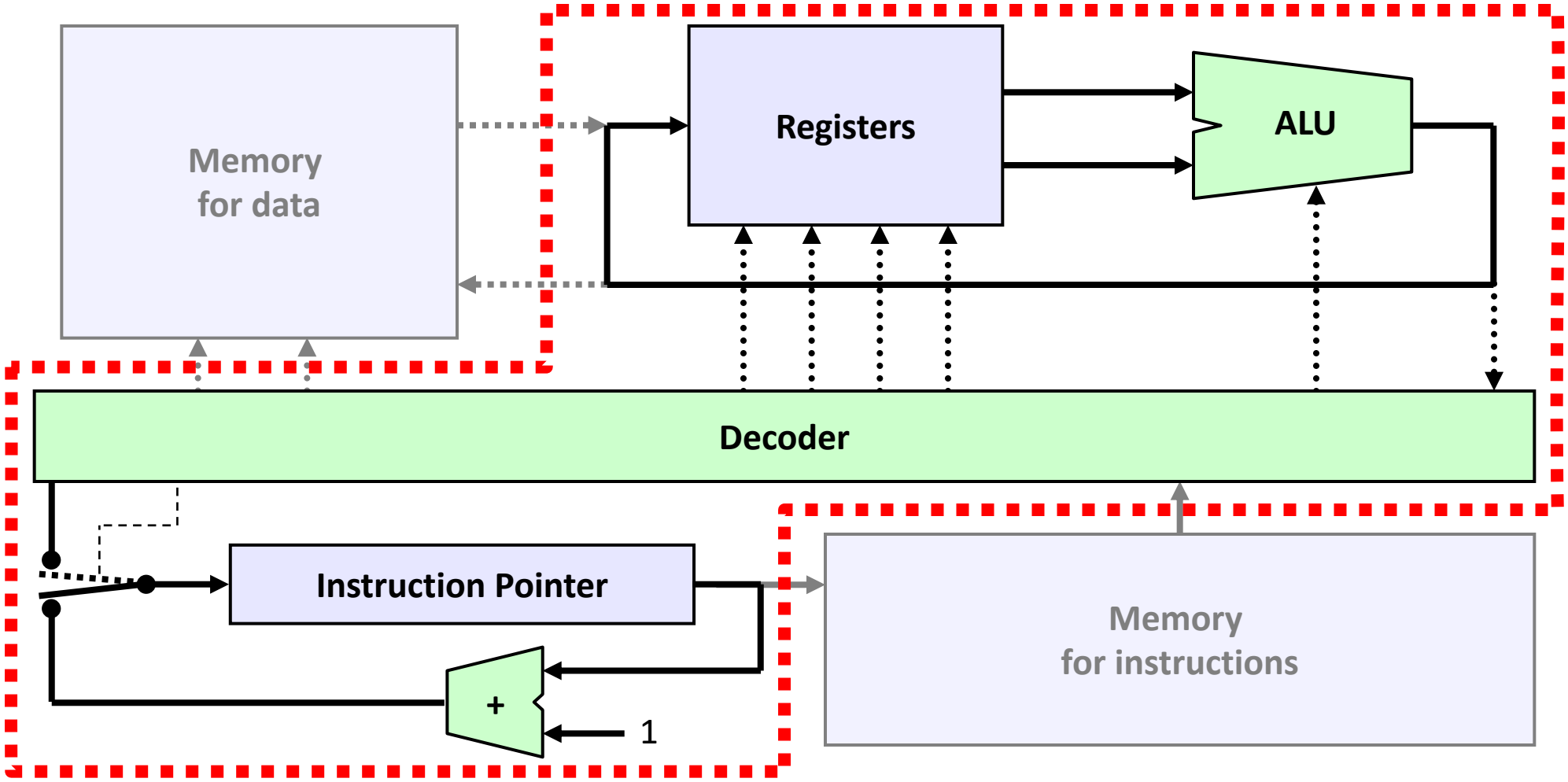


Memory to Store More Data

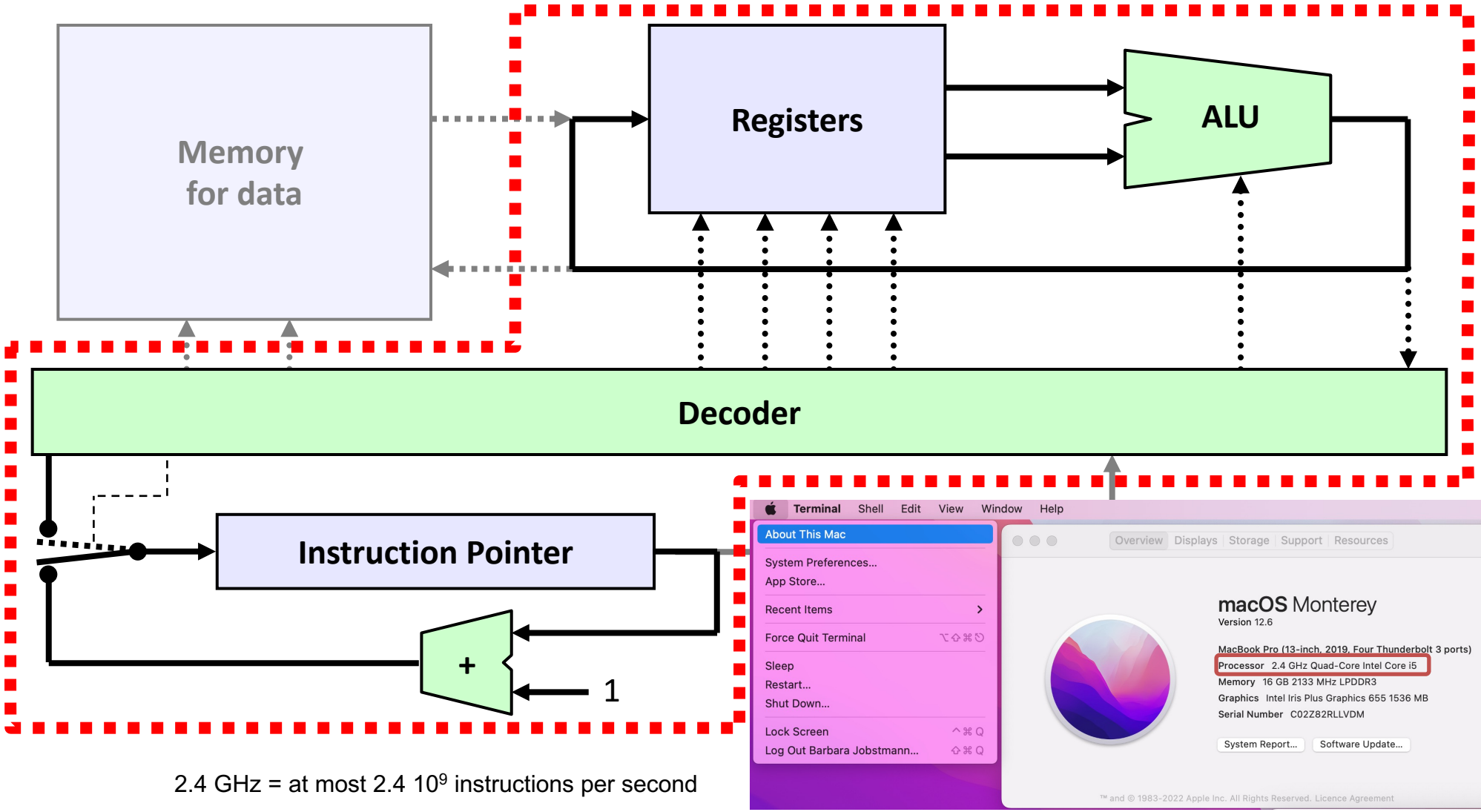
Relatively small: only
a few dozens registers



Processor or Central Processing Unit (CPU)



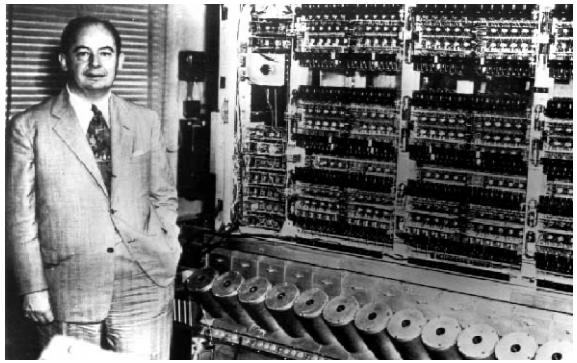
Processor or Central Processing Unit (CPU)

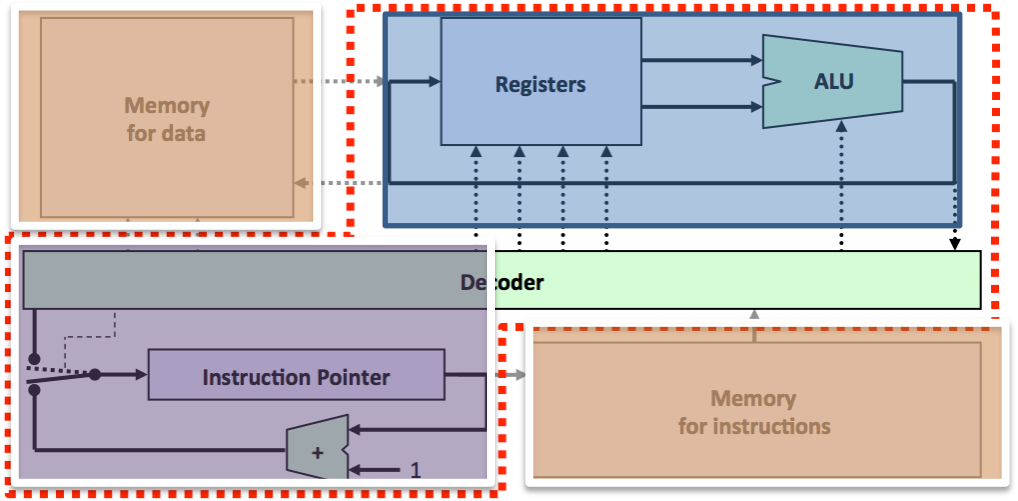


2.4 GHz = at most 2.4 10^9 instructions per second

von Neumann Architecture

- **Architecture** based on a 1945 description by **John von Neumann**
  - **Processing unit** that contains an **arithmetic logic unit** and **processor registers**
  - **Control unit** that contains an **instruction register** and **program counter**
  - **Memory** that stores **data** and **instructions**
  - External mass storage
  - Input and output mechanisms





General purpose computing device!

From Algorithms to Computers (Step 3)

sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
    s ← s + n
    n ← n – 1
m ← s
```

sum of first *n* integers

Input: r1
Output: r2

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

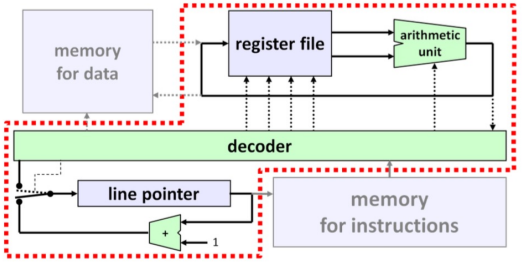
sum of first *n* integers

Input: r1
Output: r2

```
1: 0100010100000000
2: 0101100000001010
3: 0001001001011010
4: 0001010101000000
5: 0000101010101111
6: 0101000001100101
```

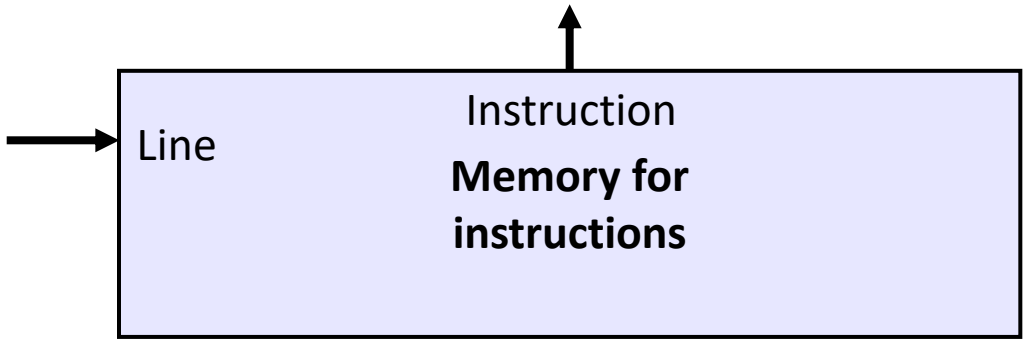
Software

Hardware

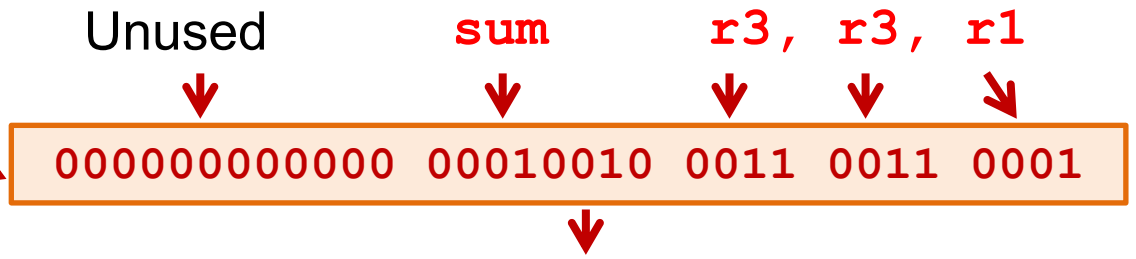


How to Encode Instructions ?

```
1: copy       r3, 0
2: jump_neqz  r1, 0, 6
3: sum        r3, r3, r1
4: sum        r1, r1, -1
5: jump       2
6: copy       r2, r3
```



- We can invent a simple encoding (cf. Lesson on “Information Representation”) :
  - A few bits for the name of the instruction (with 8 bits we can encode 256 different instructions)
  - A few bits for the registers (with 4 bits we can address 16 registers, so with 12 bits we can store two operands and one destination register)
  - And so on and so forth..
- So we can get by with 32 or 64 bits, as if to encode a typical integer



The value **592’993** represent the instruction `sum r3, r3, r1`

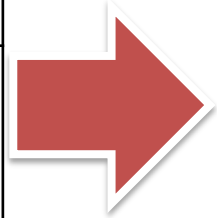
How to Encode Instructions ?

Assembly Language

sum

Input: **r1**
Output: **r2**

```
1: copy r3, 0
2: jump_negz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```



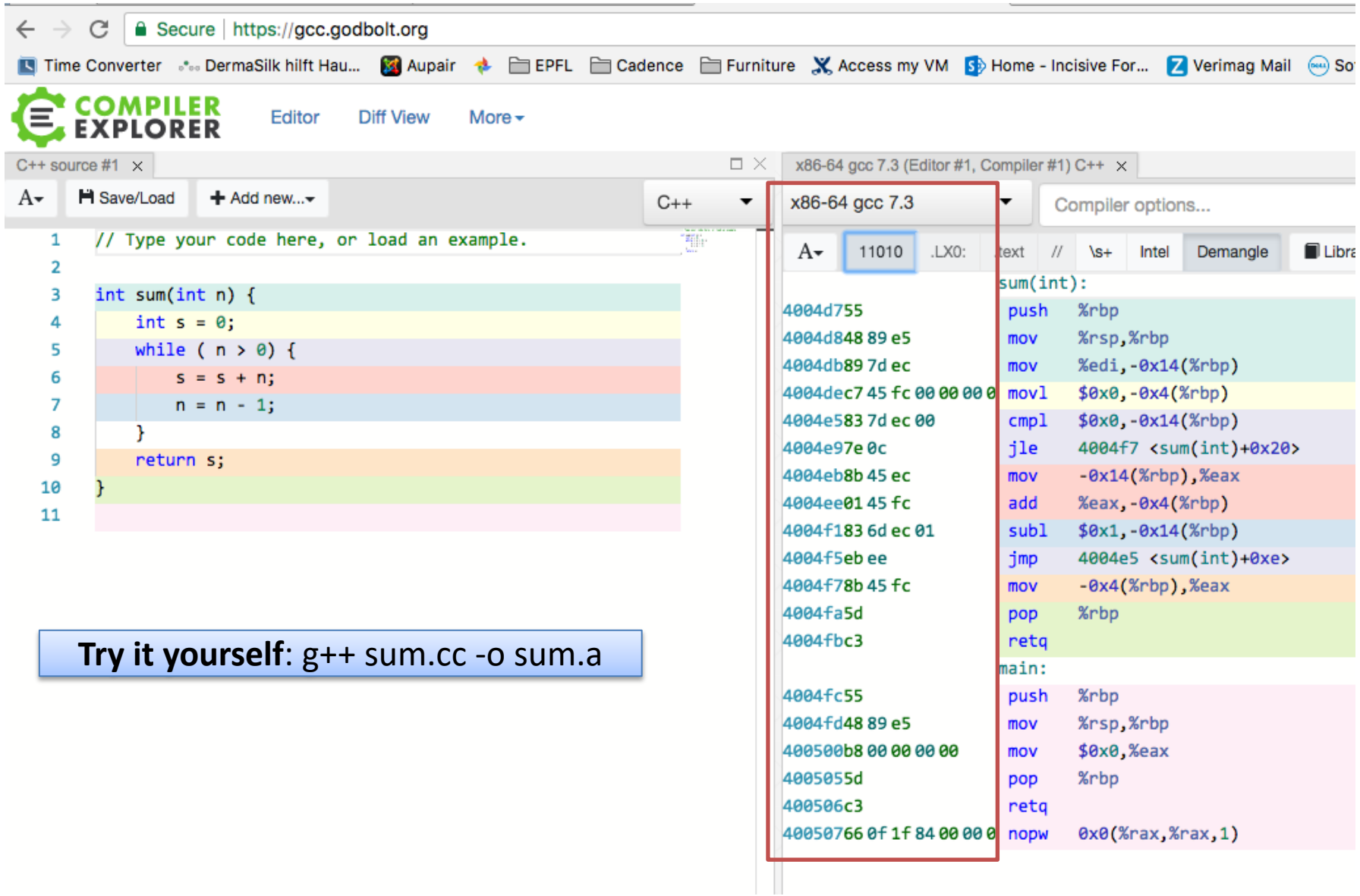
Machine Language in Binary

sum

Input: **r1**
Output: **r2**

```
1: 00000001001100000000
2: 00000101000101100000
3: 00000010001100110001
4: 00000010000100011111
5: 00000100001000000000
6: 00000001001000110000
```

Example in Machine Language (Intel)



Try it yourself: g++ sum.cc -o sum.a

From Algorithms to Computers

sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
    s ← s + n
    n ← n – 1
m ← s
```

sum of first *n* integers

Input: r1
Output: r2

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

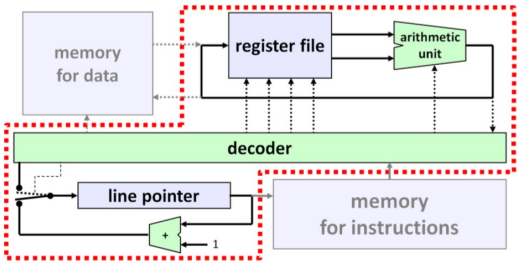
sum of first *n* integers

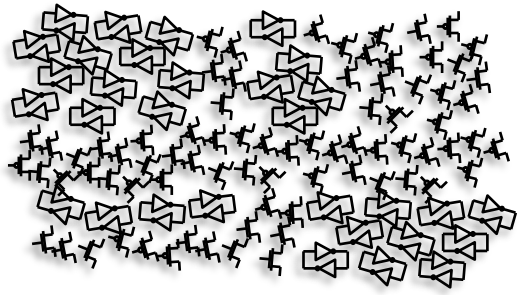
Input: r1
Output: r2

```
1: 0100010100000000
2: 0101100000001010
3: 0001001001011010
4: 0001010101000000
5: 0000101010101111
6: 0101000001100101
```

Software

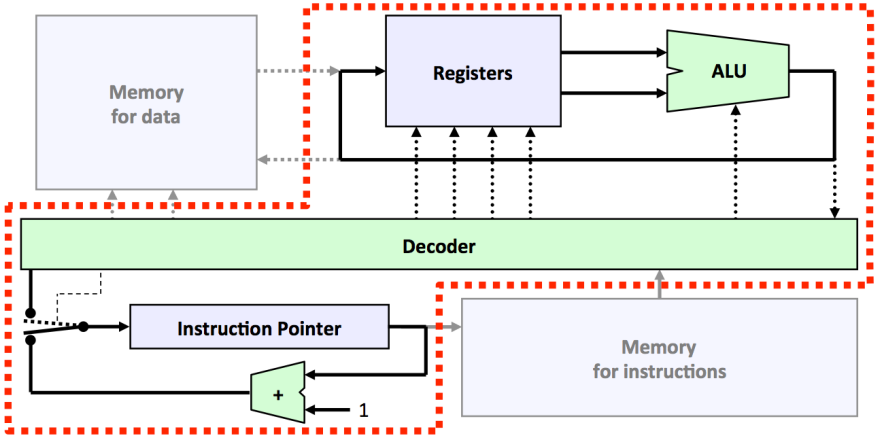
Hardware



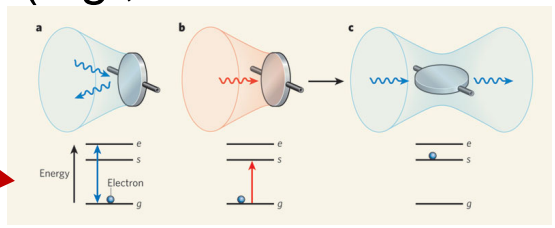


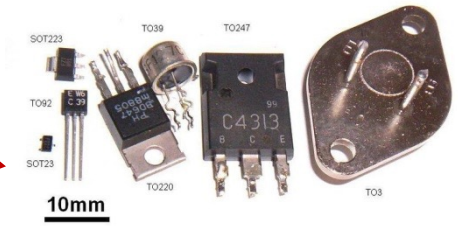
Technologies

- So far, our machine is abstract, completely independent of the underlying technology
- Even binary encoding is not a necessity

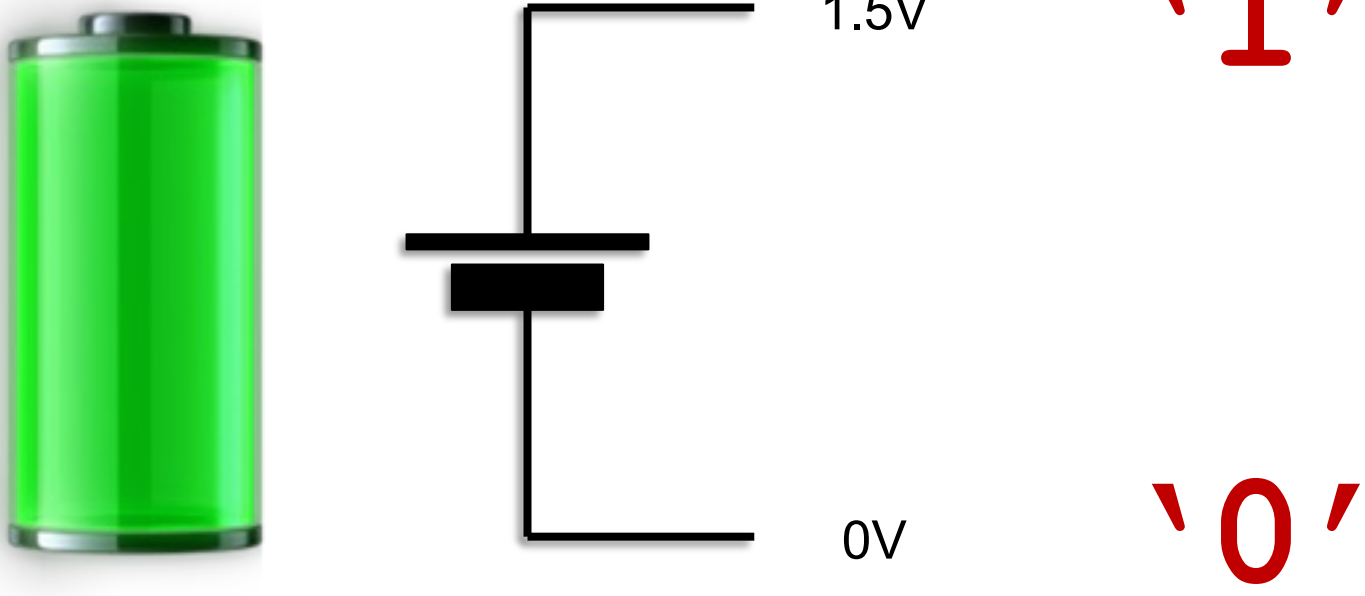


- Different technologies are possible:
  - Electro-mechanical (e.g., relays)
  - Electronic (e.g., tubes or transistors)
  - Optical





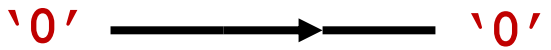
A Battery has Two Voltage Levels



- Very well suited to represent information in binary

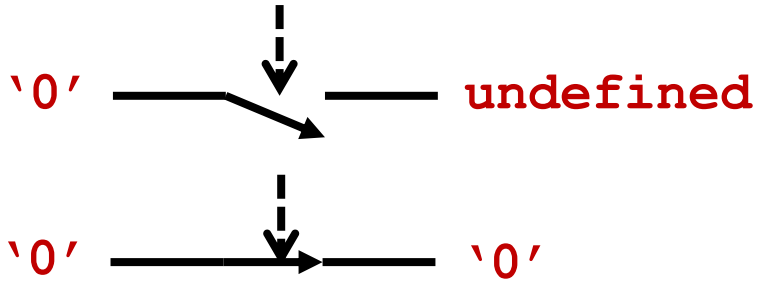
Switch (Interruptible Wire)

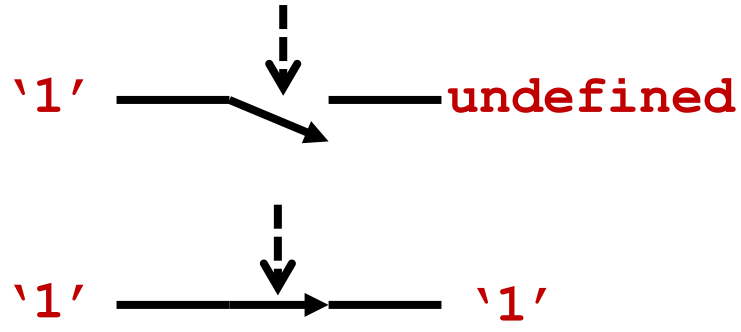
Does not propagate its state if the connection is open



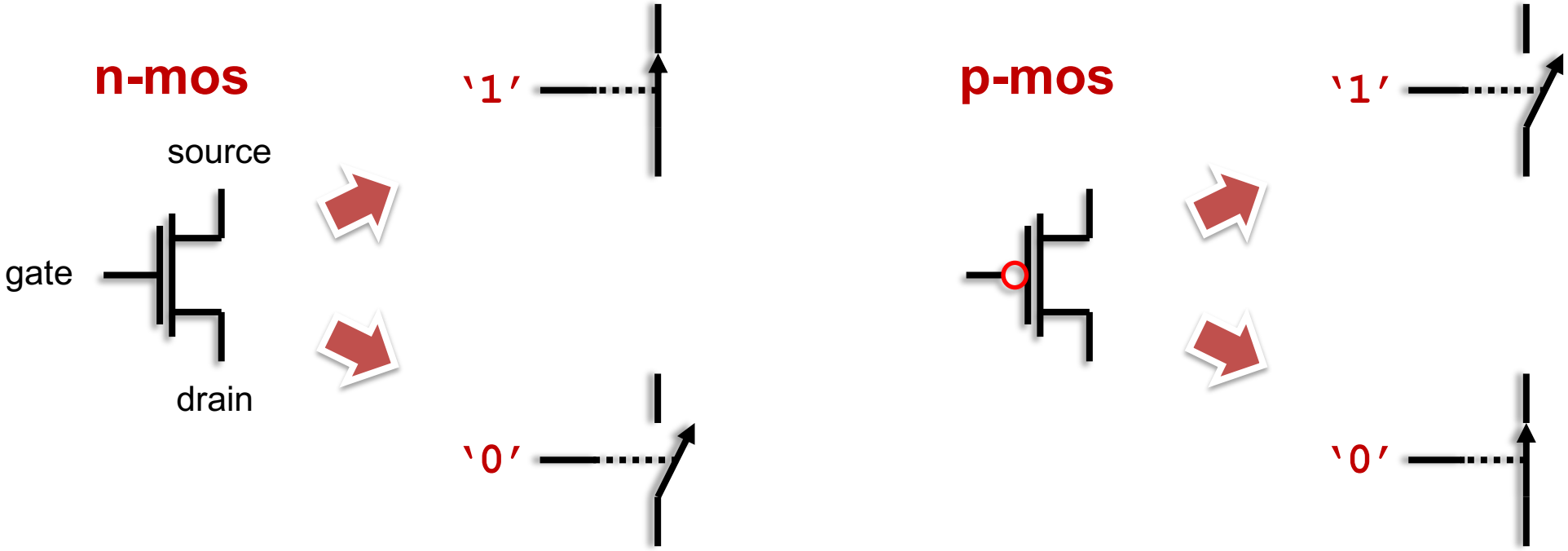
Propagates its states if the connection is closed

Transistor = Controllable Switch

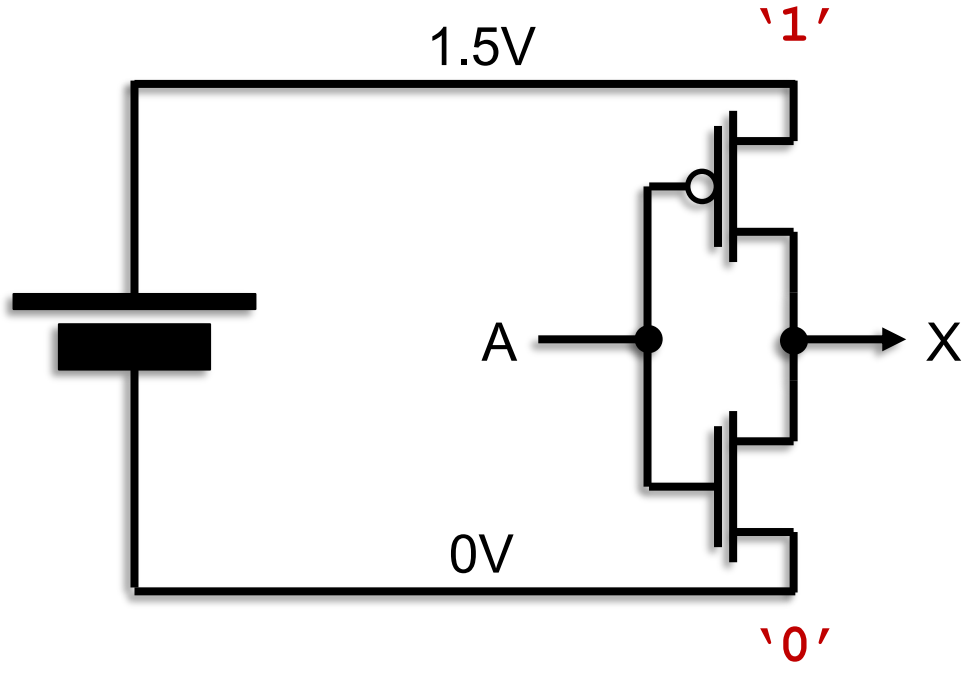




Transistor = Controllable Switch



Inverter (NOT Gate)

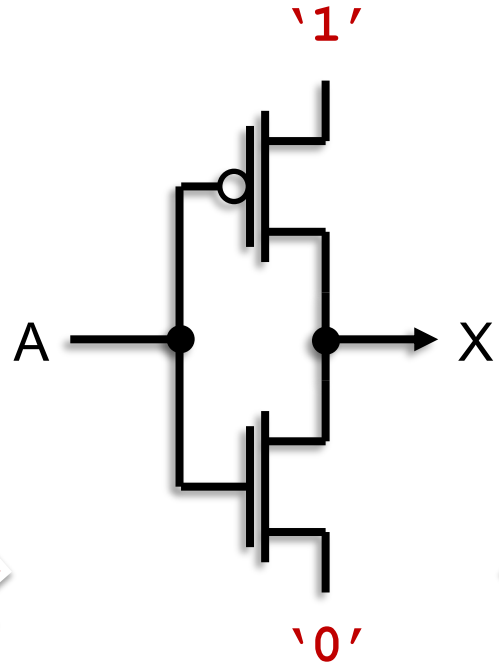


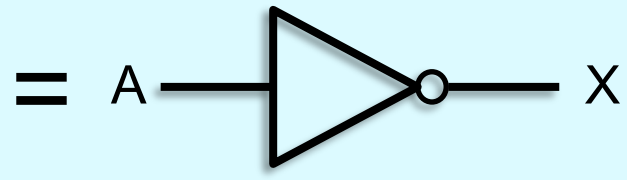
Gate = electronic circuit that can do basic computation

Inverter (NOT Gate)

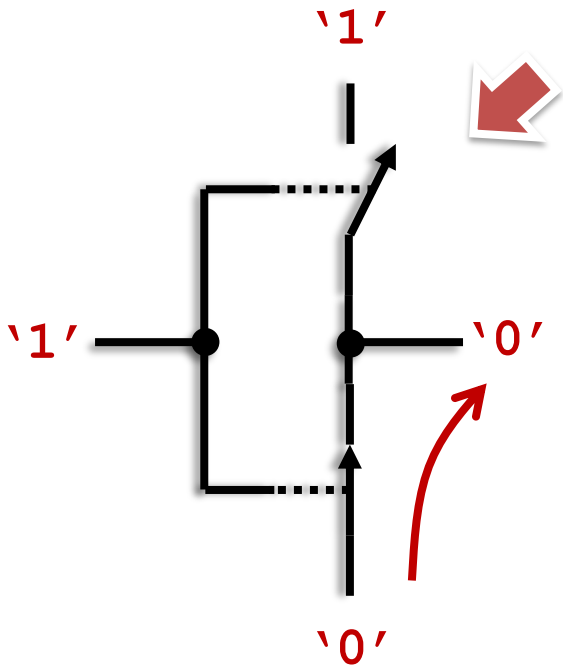
Truth table

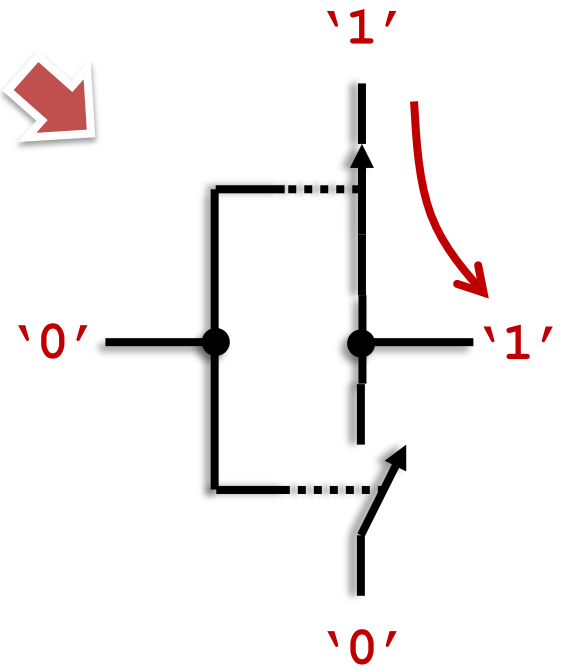
A	X
0	1
1	0





Logic Symbol





Truth Table of a Circuit

- Describes the function of a circuit
- For instance, consider a circuit with 2 inputs and 1 output
  - $2^2 = 4$ combinations (possible states)

Input 1	Input 2	Output
0	0	?
0	1	?
1	0	?
1	1	?

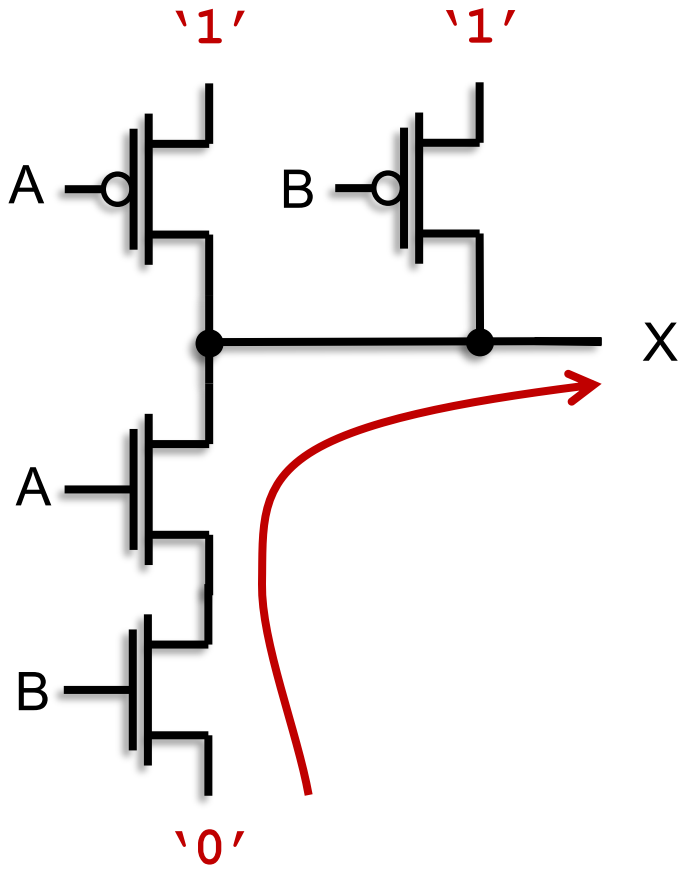
- Inverter from previous slide (1 input, 1 output)

Input	Output
0	1
1	0

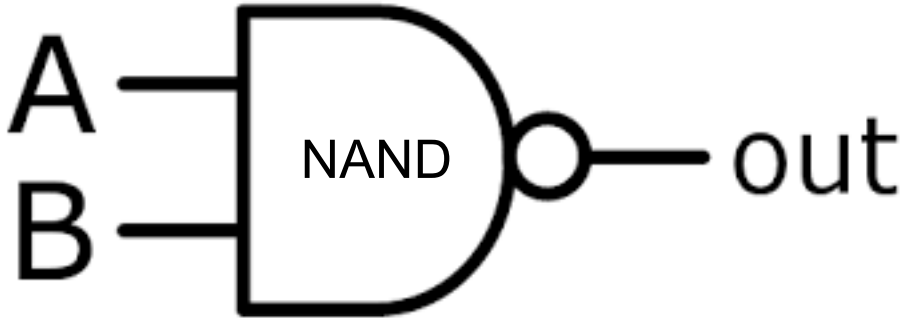
Drawing a Truth Table

Input 1	Input 2	Input 3	Output
0	0	0	0
0	0	1	1
0	1	0	…
0	1	1	…
1	0	0	…
1	0	1	…
1	1	0	…
1	1	1	…

A Circuit "NAND" (NOT AND)



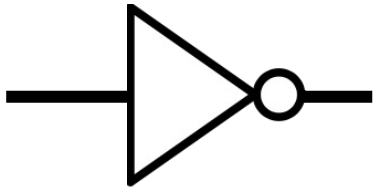
A	B	X = not (A and B)
0	0	1
0	1	1
1	0	1
1	1	0



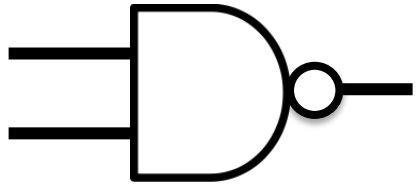
- The only way to get a `'0'` is to put two `'1'` at the inputs
  A NAND B: the output is `'0'` only if A **AND** B are `'1'`

Circuits for Other Functions

- We already know **NOT** and **NAND**



A	NOT A
0	1
1	0



Boolean function: $\mathbb{B}^n \rightarrow \mathbb{B}^m$

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

- NOT and NAND gates can implement any Boolean function.
  How many different Boolean functions with two inputs exist?

A	B	0	AND	F2	A	F4	B	XOR	OR
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

A	B	NOR	F9	NOT B	F11	NOT A	F13	NAND	1
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Do I have to know all of them? No, only NOT, AND, OR

The three correspond to negation (¬), conjunction (∧) and disjunction (∨).

AND and OR

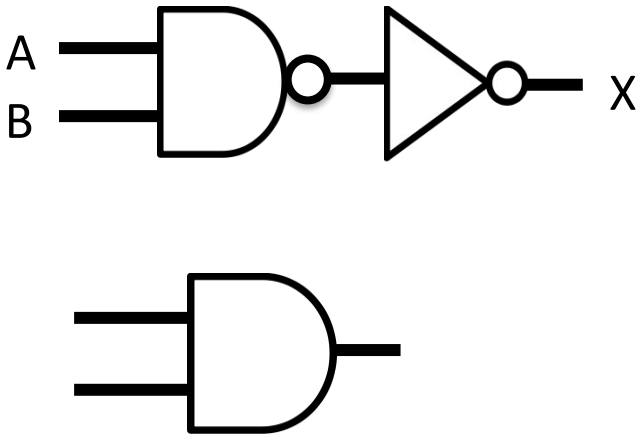
- We can build more functions using the known gates

A	B	X = A AND B
0	0	0
0	1	0
1	0	0
1	1	1

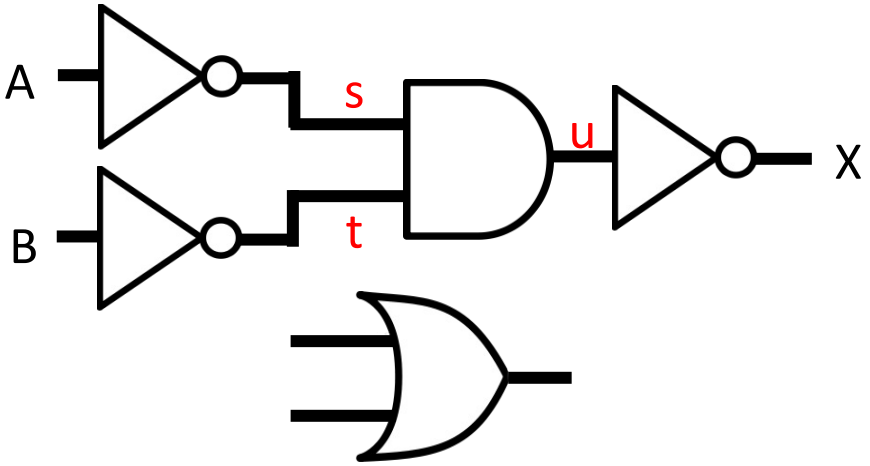
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

s	t	u
1	1	1
1	0	0
0	1	0
0	0	0

A **AND** B = **NOT** (A **NAND** B)



A **OR** B = **NOT** ((**NOT** A) **AND** (**NOT** B))



Circuits Can Implement Arbitrary Functions

Sum of 1s

A	B	C	Decimal	Carry	Sum
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

```
Sum = (NOT A AND NOT B AND     C) OR
      (NOT A AND     B AND NOT C) OR
      (    A AND NOT B AND NOT C) OR
      (    A AND     B AND     C)
```

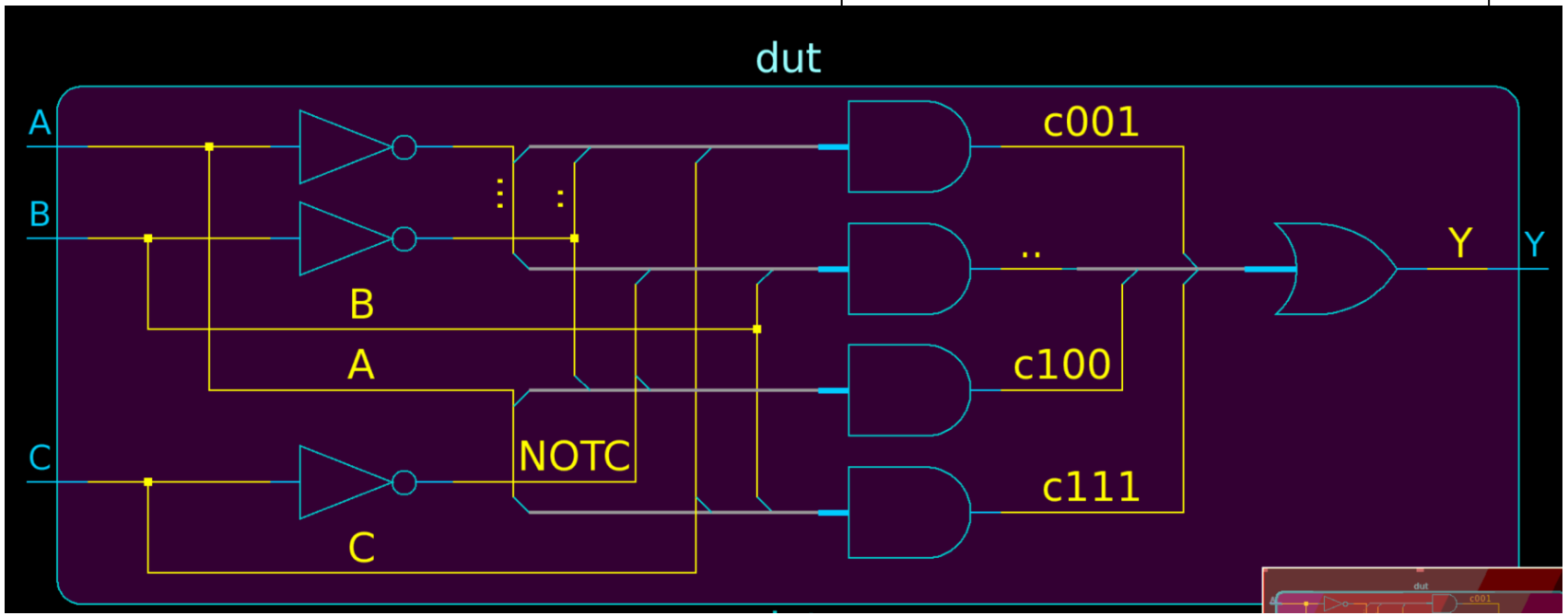
```
Carry = (NOT A AND     B AND     C) OR
        (    A AND NOT B AND     C) OR
        (    A AND     B AND NOT C) OR
        (    A AND     B AND     C)

Carry = (NOT A AND B AND C) OR (B OR C)
```

Example

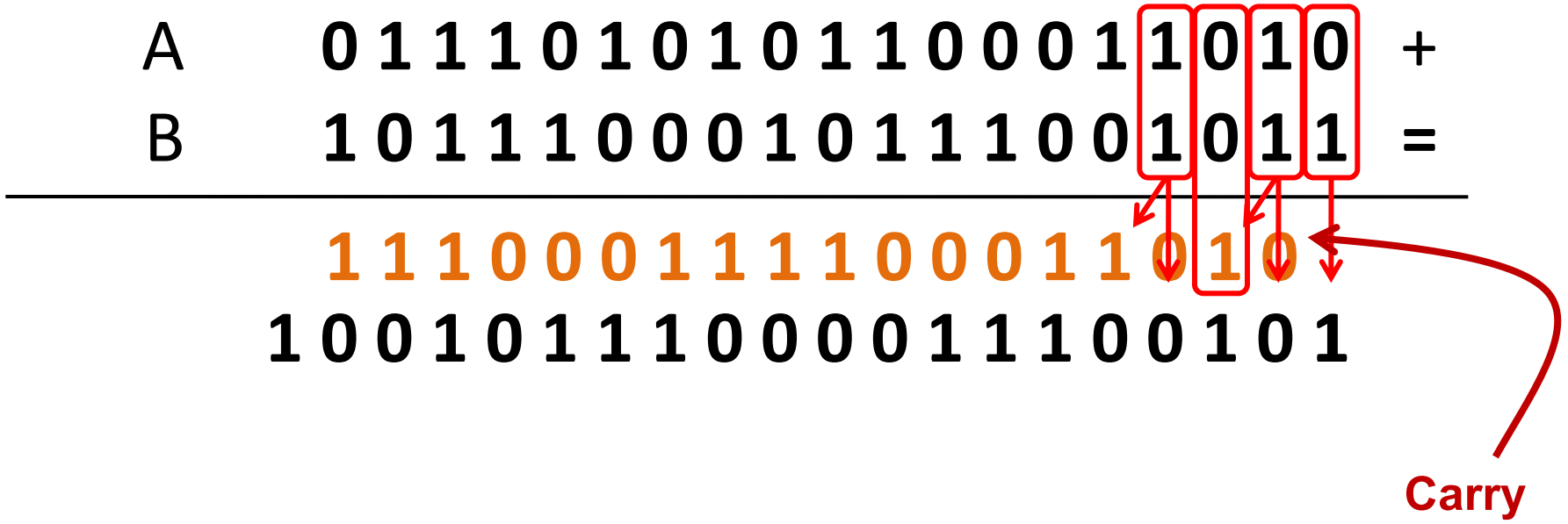
```
Y = (NOT A AND NOT B AND     C) OR
    (NOT A AND     B AND NOT C) OR
    (    A AND NOT B AND NOT C) OR
    (    A AND     B AND     C)
```

In Verilog (a hardware description language)

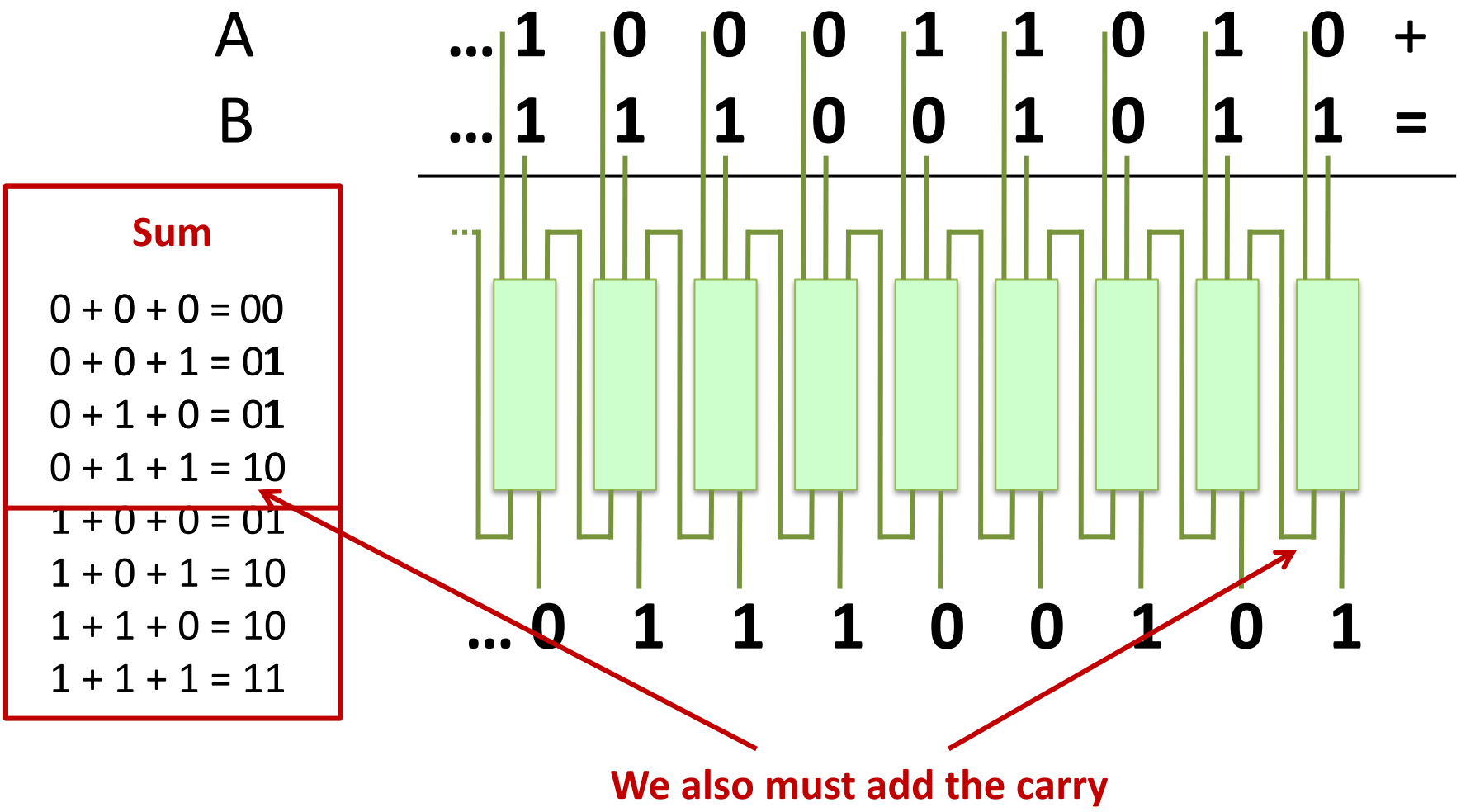
```
not n1(NOTA,A);
not n2(NOTB,B);
not n3(NOTC,C);

and a1(c001, NOTA, NOTB, C);
and a2(c010, NOTA, B, NOTC);
and a3(c100, A, NOTB, NOTC);
and a4(c111, A, B, C);

or o1(Y, c001, c010, c100, c111);
```



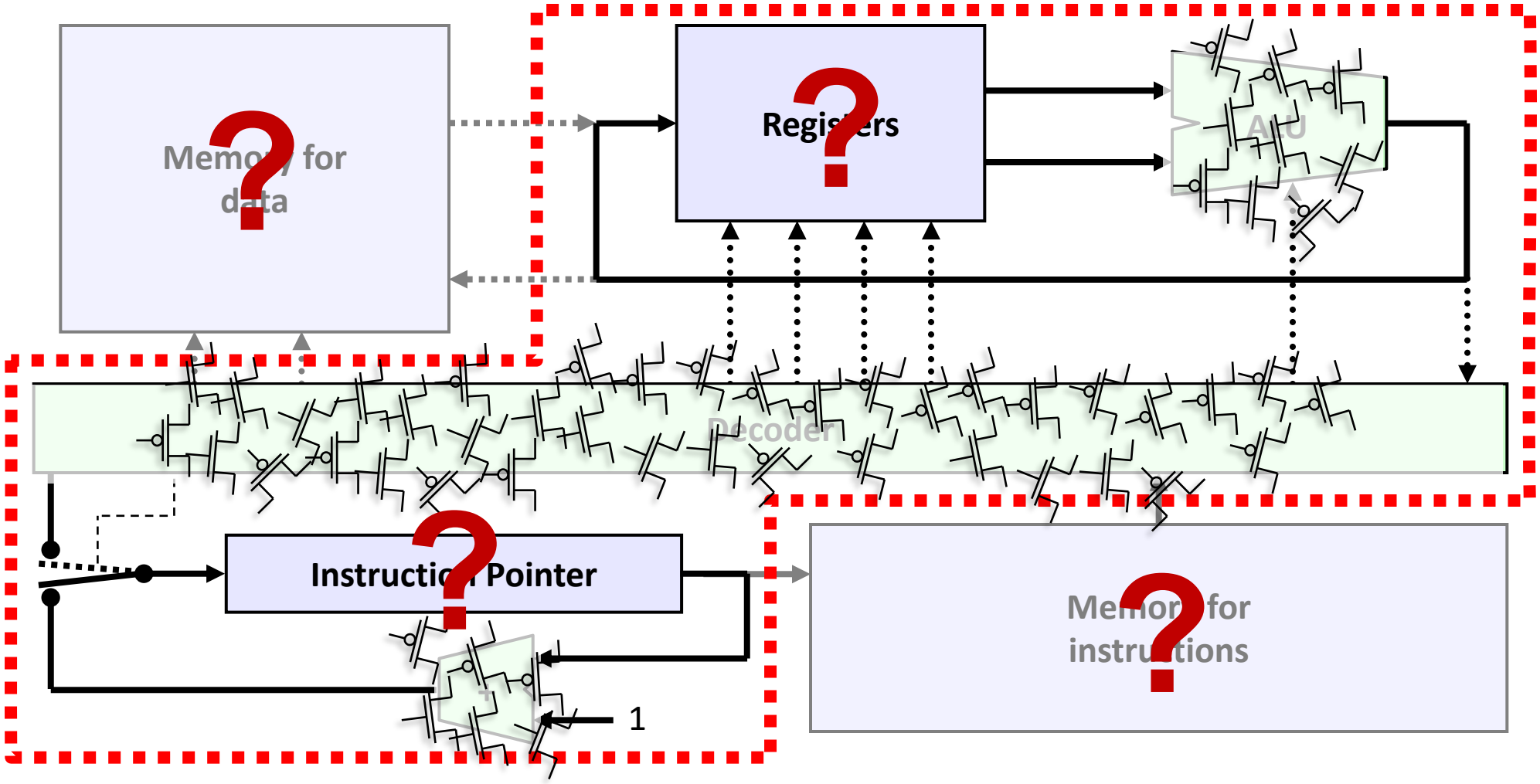
Adding is simple…



Making an Adder is simple…

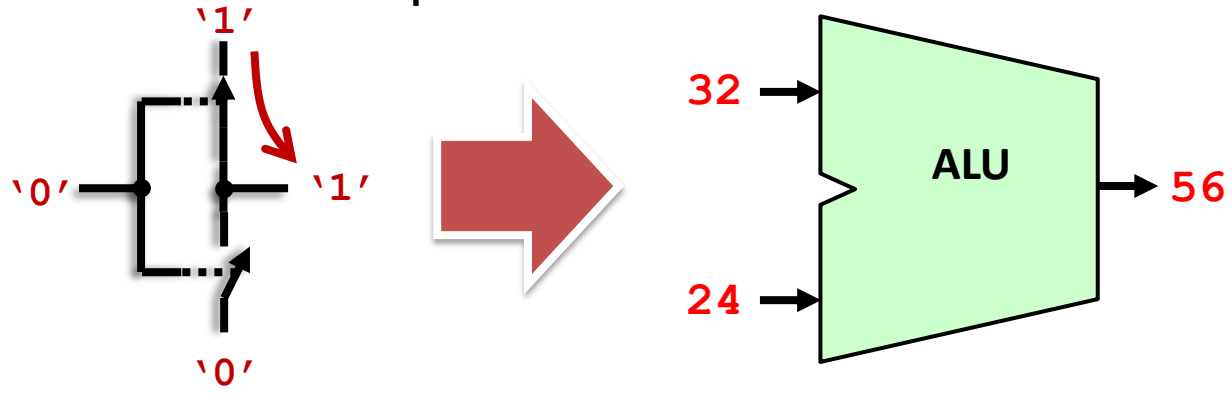


What about Registers and Memory?

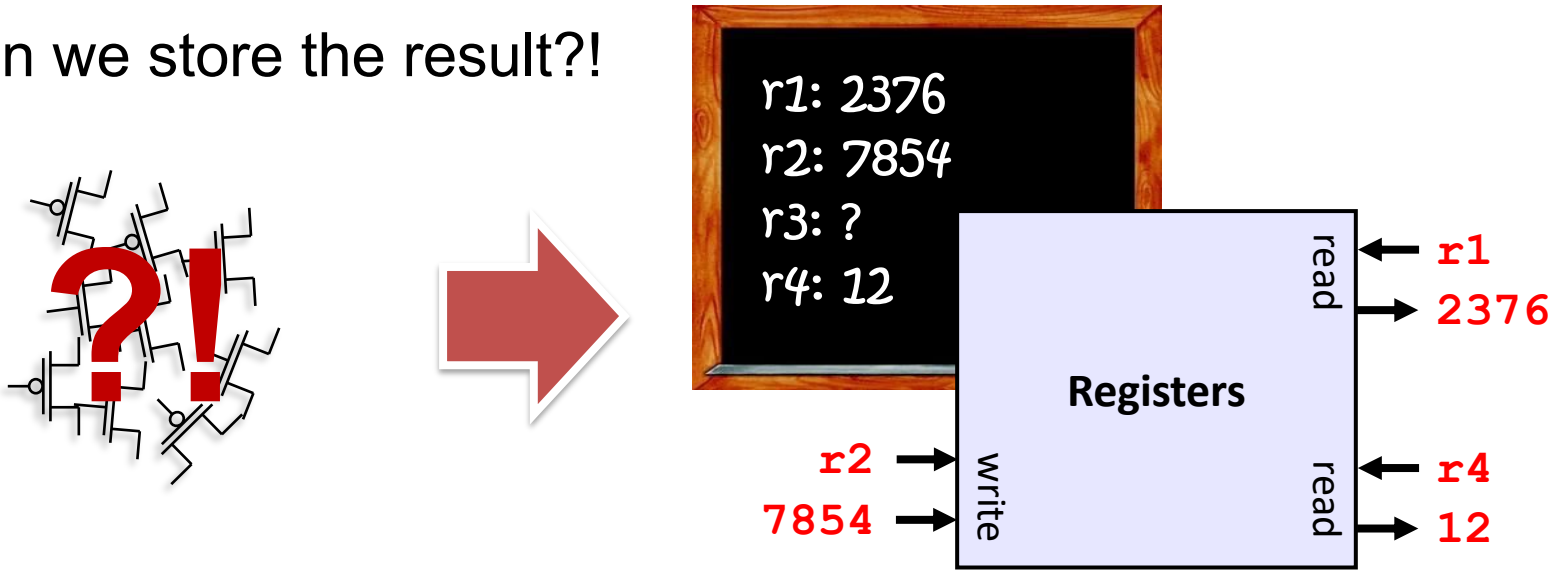


How to Store Information?

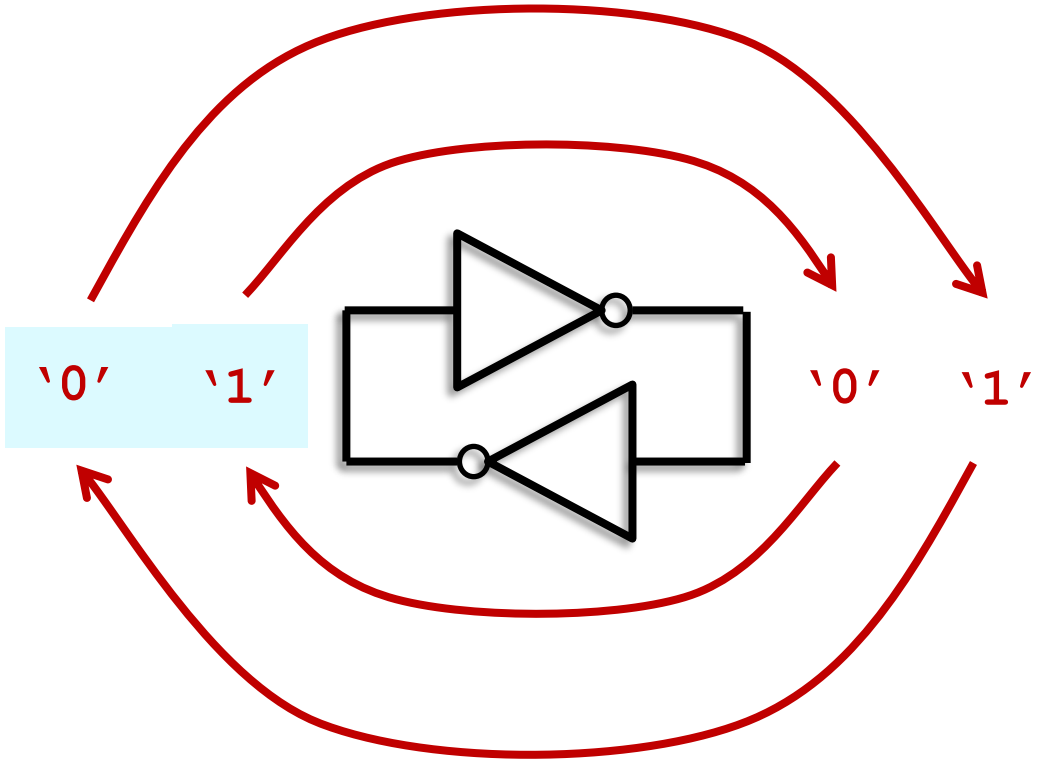
- We know now how to compute:



- How can we store the result?!

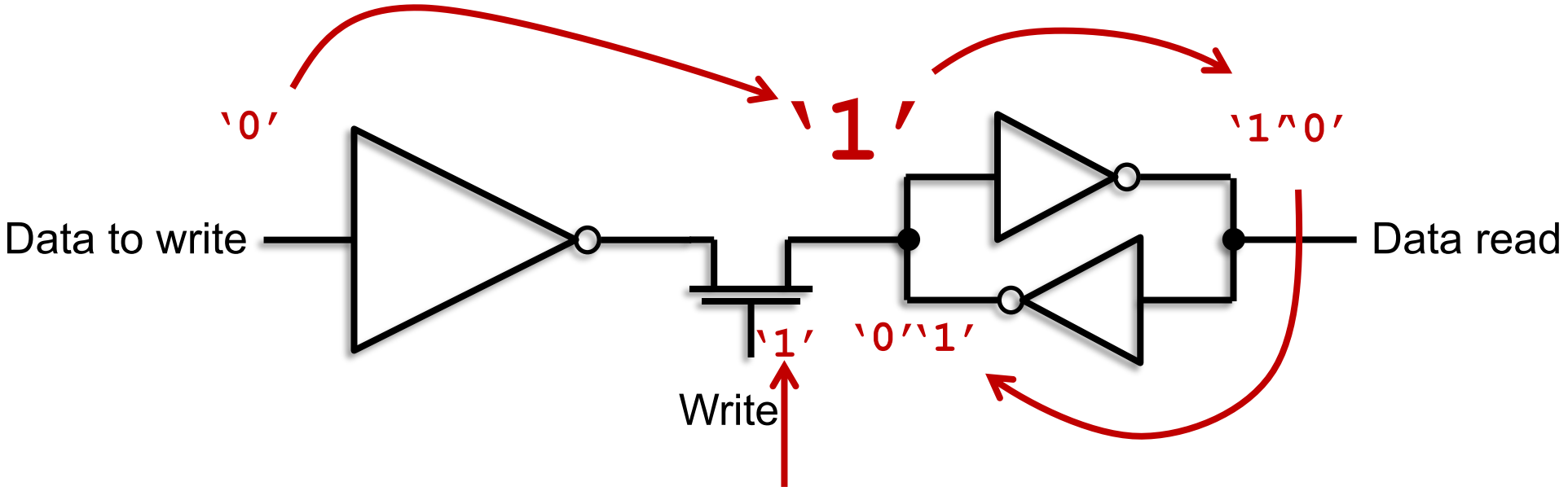


A Rather Special Circuit

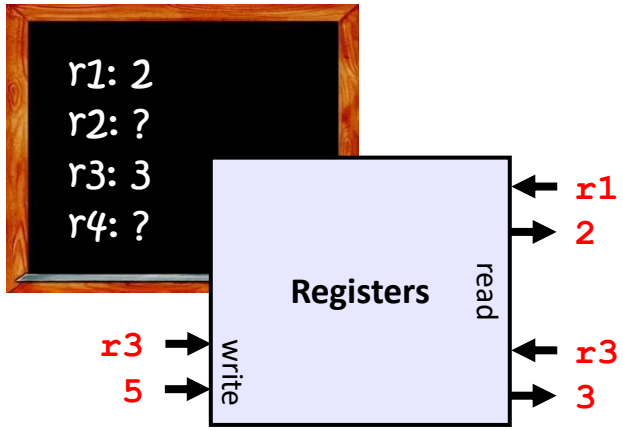


- A "bistable" circuit, i.e., it can be in one of two perfectly stable states. **A memory element of 1 bit!**

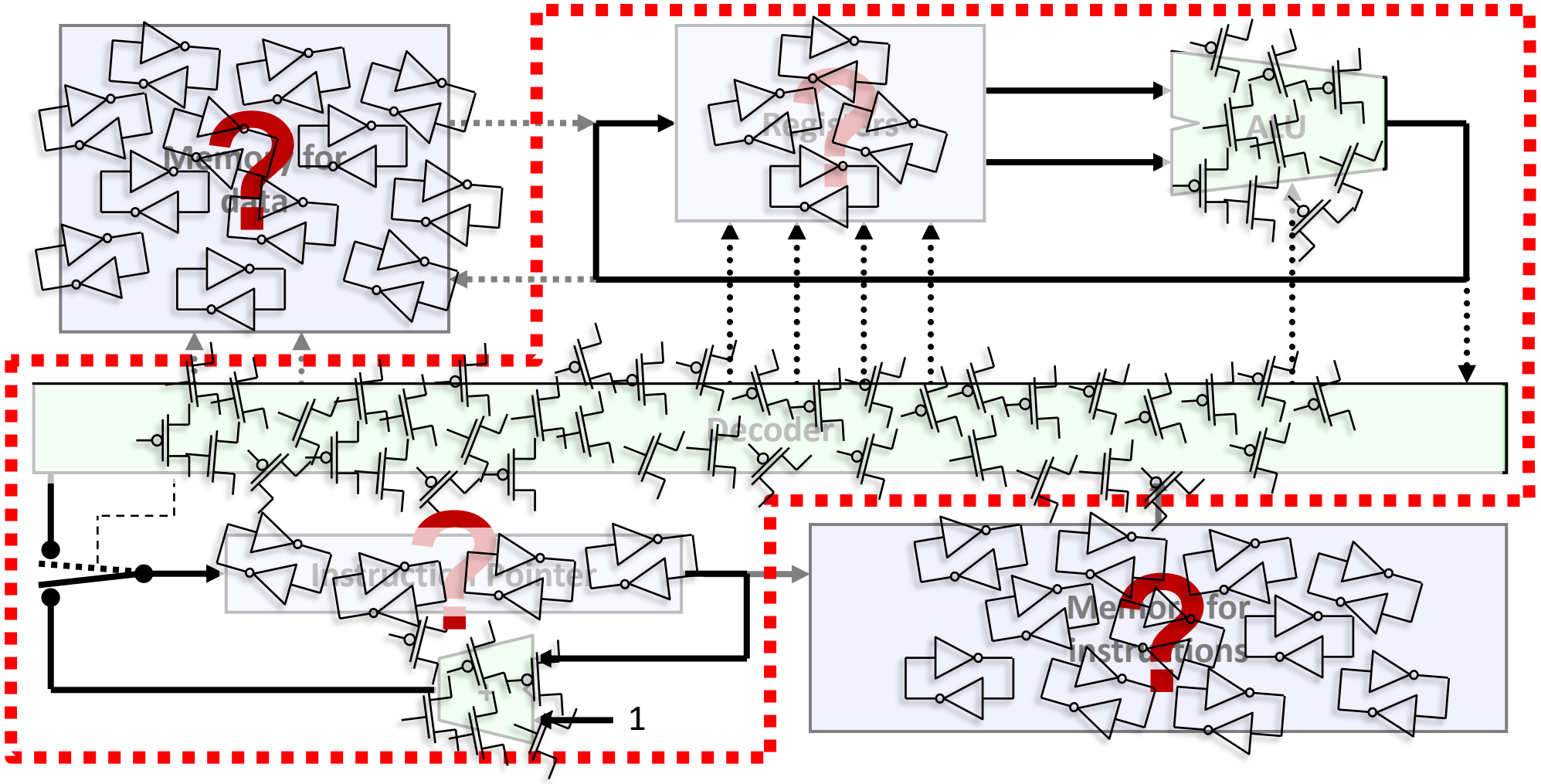
How to Write in this Memory?



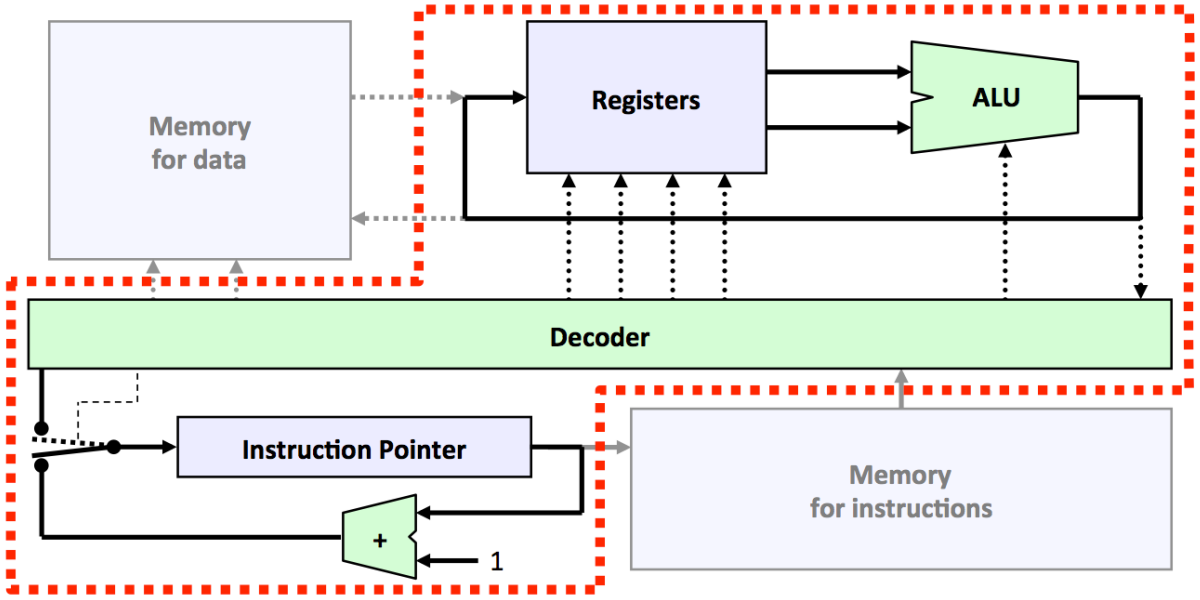
- With just a few transistors, we get a perfect memory circuit for all the bits of our registers and memories



What about Registers and Memory?

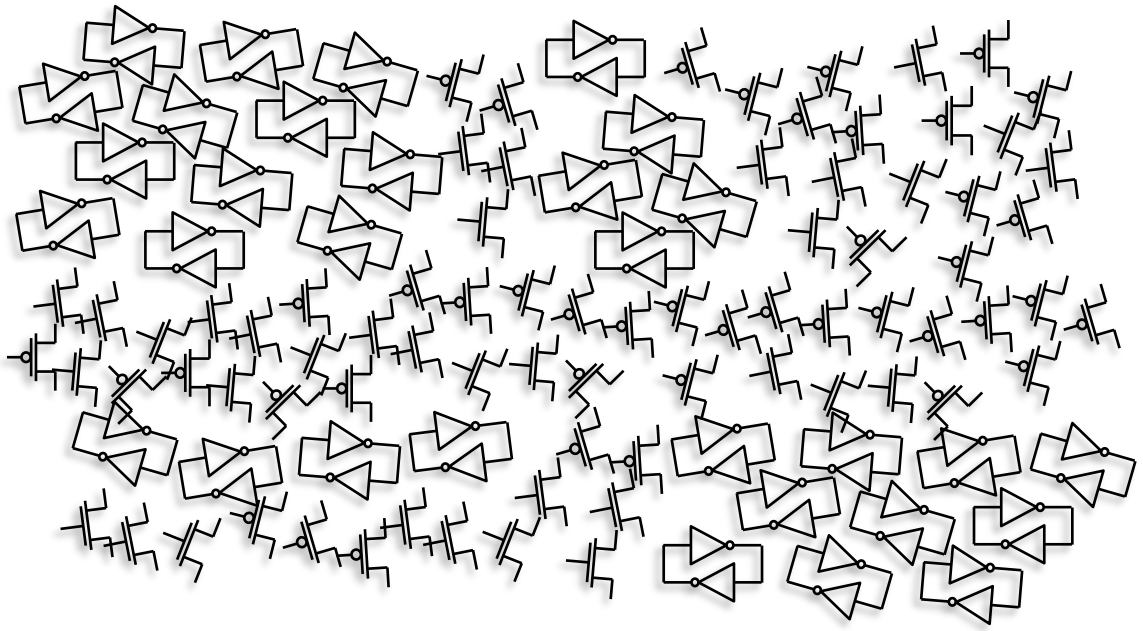


We have reached our goal!



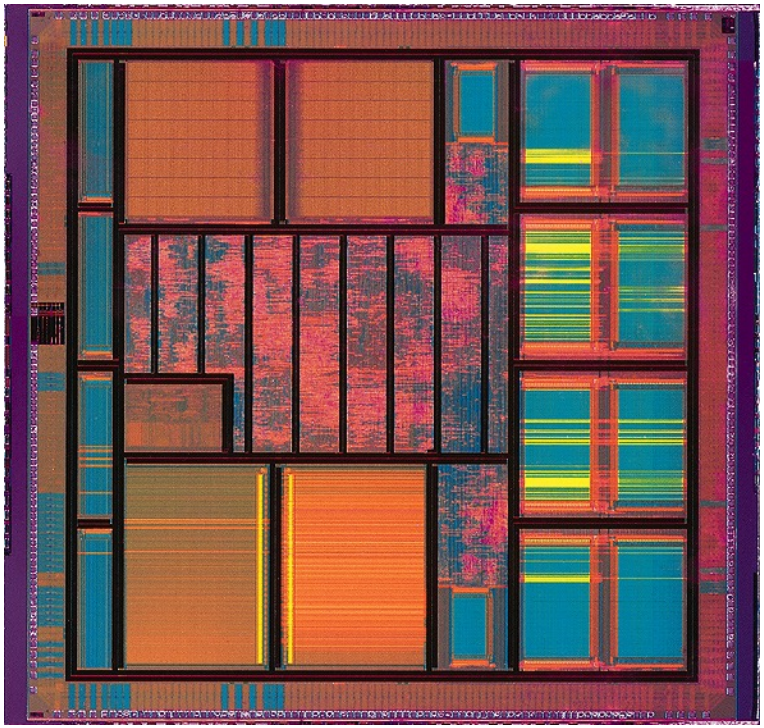
Architecture of a processor

We have reached our goal!



Electronic circuit

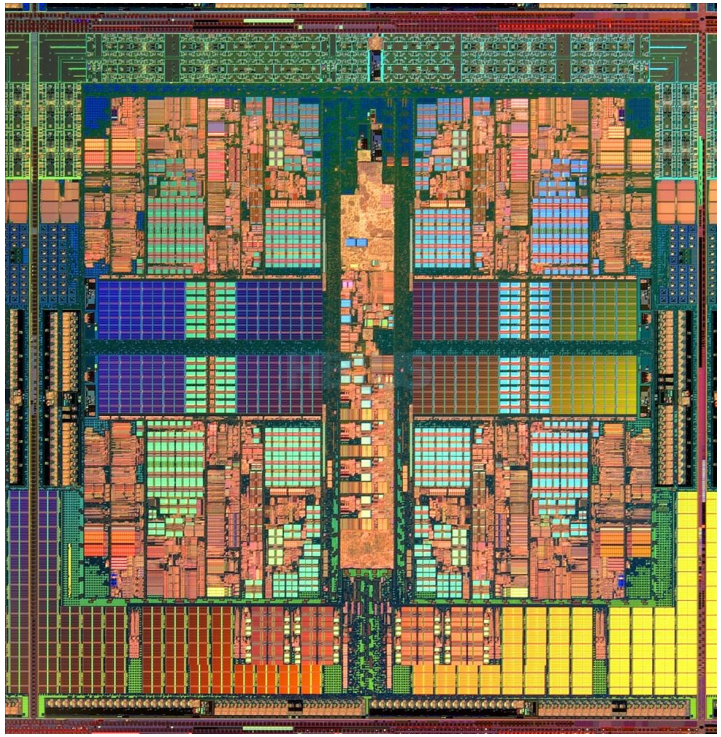
We have reached our goal!



A VLSI circuit can have nowadays around several 10^9 transistors

- Transistors can be extremely cheap and small:
  - A transistor costs as little as 10^{-8} and 10^{-7} cents (CHF/USD/EUR)
  - Modern device consists of Billion of transistors, e.g., Apple A17 SoC using in iPhone 15 Pro has about 19 Billion (10^9) transistors on about $100mm^2$

We have reached our goal!



From Algorithms to Computers

sum of first *n* integers

Input: n
Output: m

```
s ← 0
while n > 0
      s ← s + n
      n ← n – 1
m ← s
```

Programming Langages

C, C++, C#, Java, Scala, Python, Perl, PHP, SQL, Excel, etc.

sum of first *n* integers

Input: r1
Output: r2

```
1: copy r3, 0
2: jump_egz r1, 6
3: add r3, r3, r1
4: add r1, r1, -1
5: jump 2
6: copy r2, r3
```

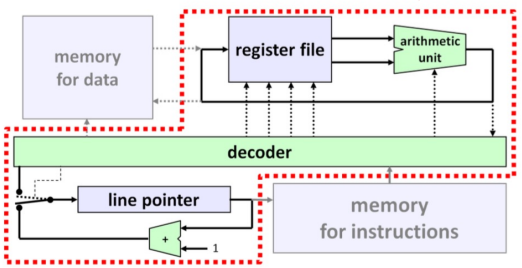
sum of first *n* integers

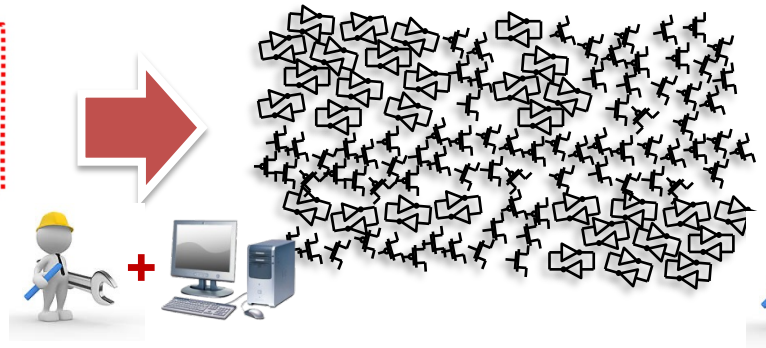
Input: r1
Output: r2

```
1: 0100010100000000
2: 0101100000001010
3: 0001001001011010
4: 0001010101000000
5: 0000101010101111
6: 0101000001100101
```

Software

Hardware



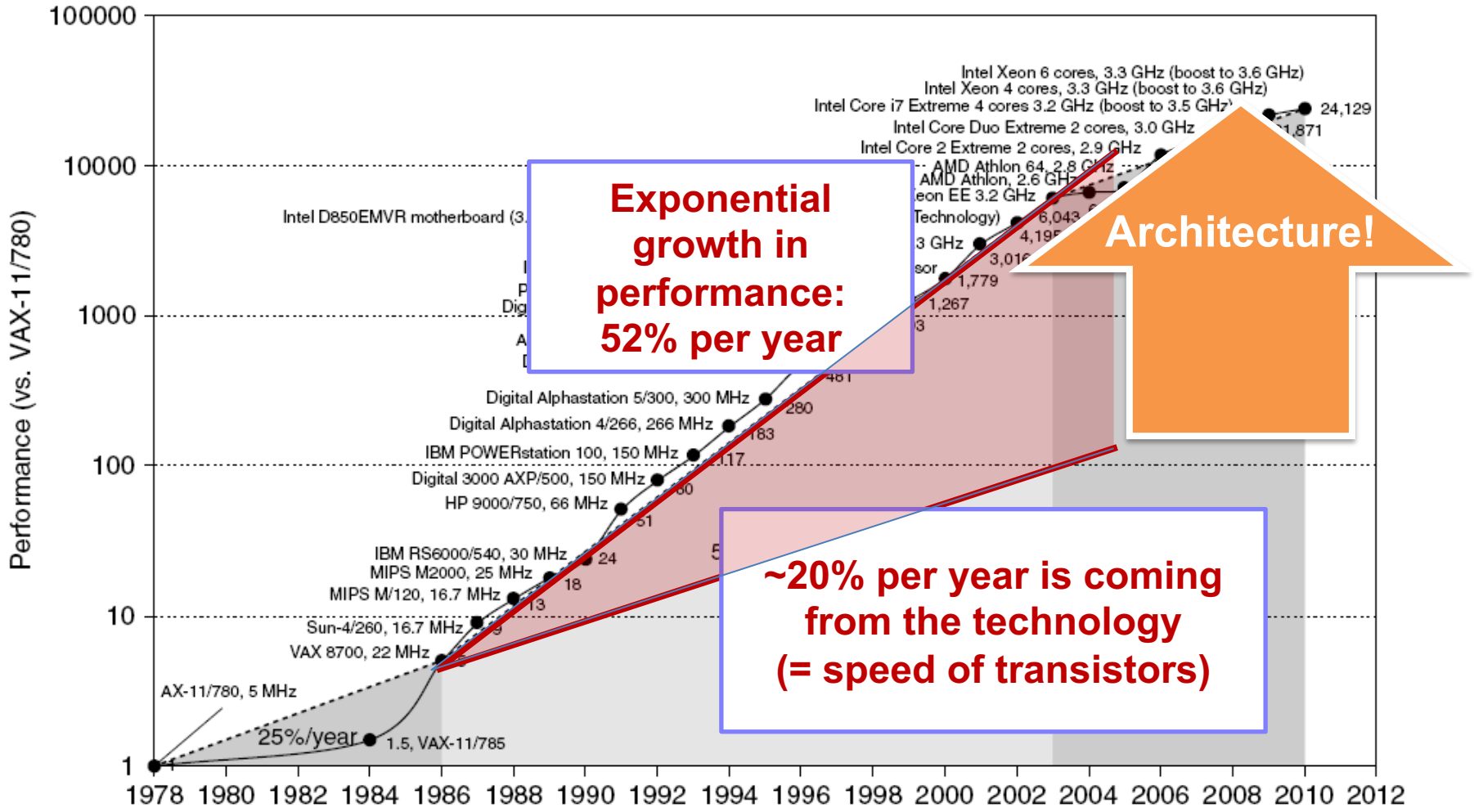


Summary

- From Algorithms to Computers
  - Assembly code = Basic instructions
  - Processor structure (registers, ALU, instruction counter, memories)
  - Transistors use to compute and store

Question

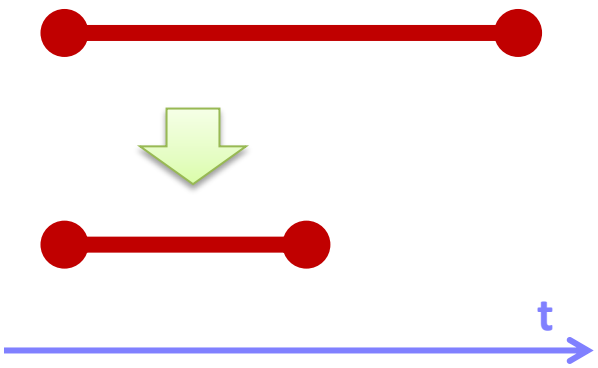
- How can we make these systems **faster** ?



How to increase performance?

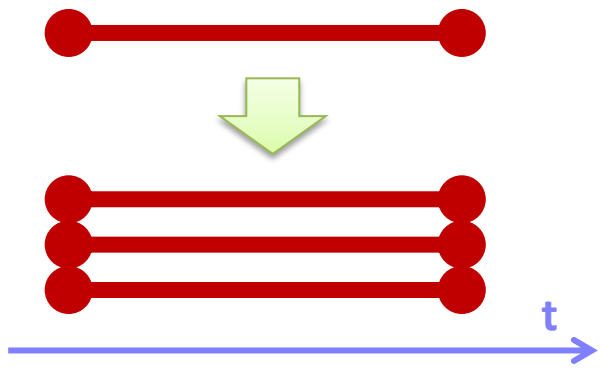
= Reduce delay

waiting time to get a result



= Increase the throughput

Number of results per time unit



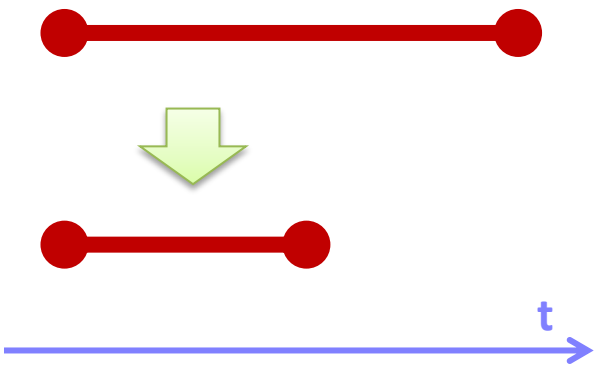
Two simple examples to improve performance:

1. At the level of the circuit:
   reduce the delay of an adder
2. At the level of the process structure:
   increase the throughput of instructions

How to increase performance?

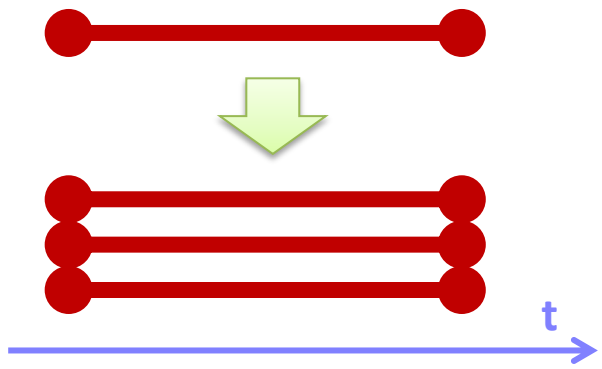
= Reduce delay

waiting time to get a result



= Increase the throughput

Number of results per time unit



Two simple examples to improve performance:

1. At the level of the circuit:
   reduce the delay of an adder
2. At the level of the process structure:
   increase the throughput of instructions

Adding is simple…

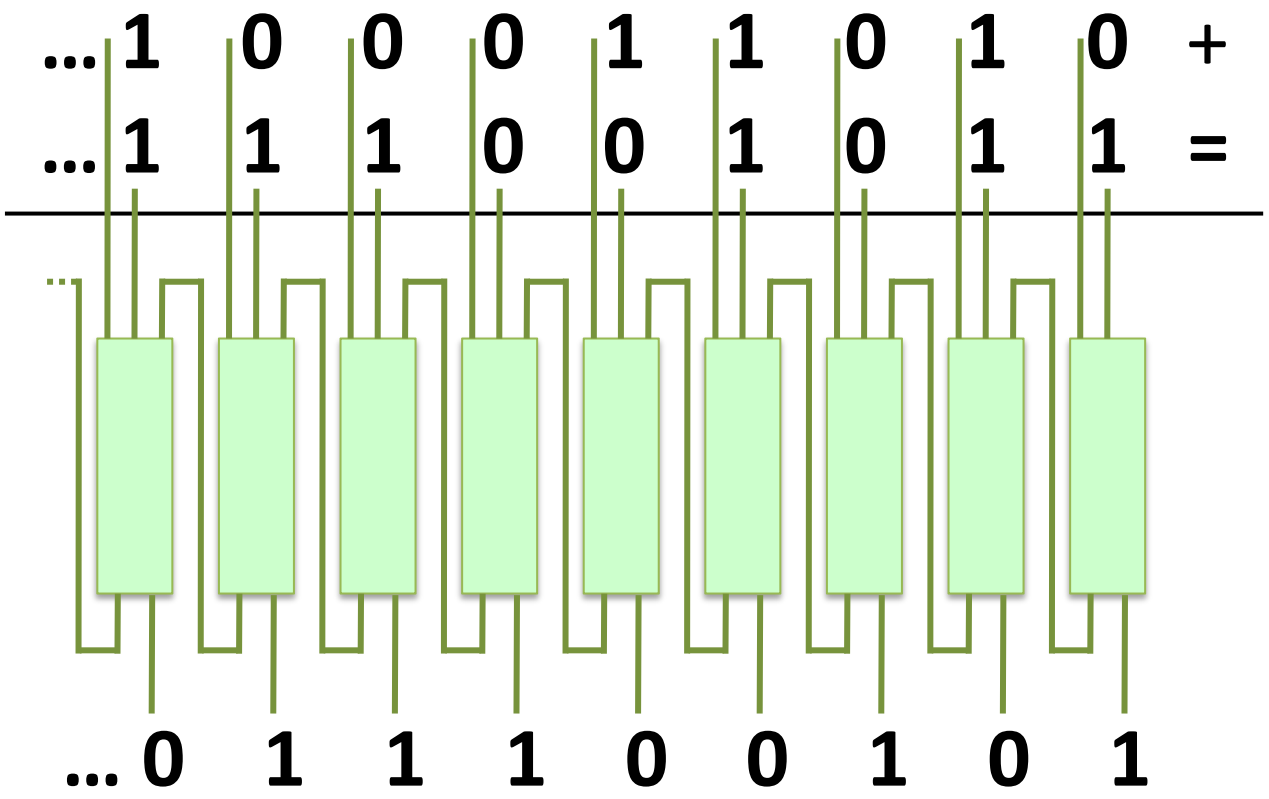
```
A    0 1 1 1 0 1 0 1 0 1 1 0 0 0 1 1 0 1 0  +
B    1 0 1 1 1 0 0 0 1 0 1 1 1 0 0 1 0 1 1  =
-------------------------------------------------
     1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 0
  1 0 0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1 0 1
```

Carry

Elementary sums:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 = $1 \cdot \mathbf{2^1} + 0 \cdot \mathbf{2^0} = 2_{10}$

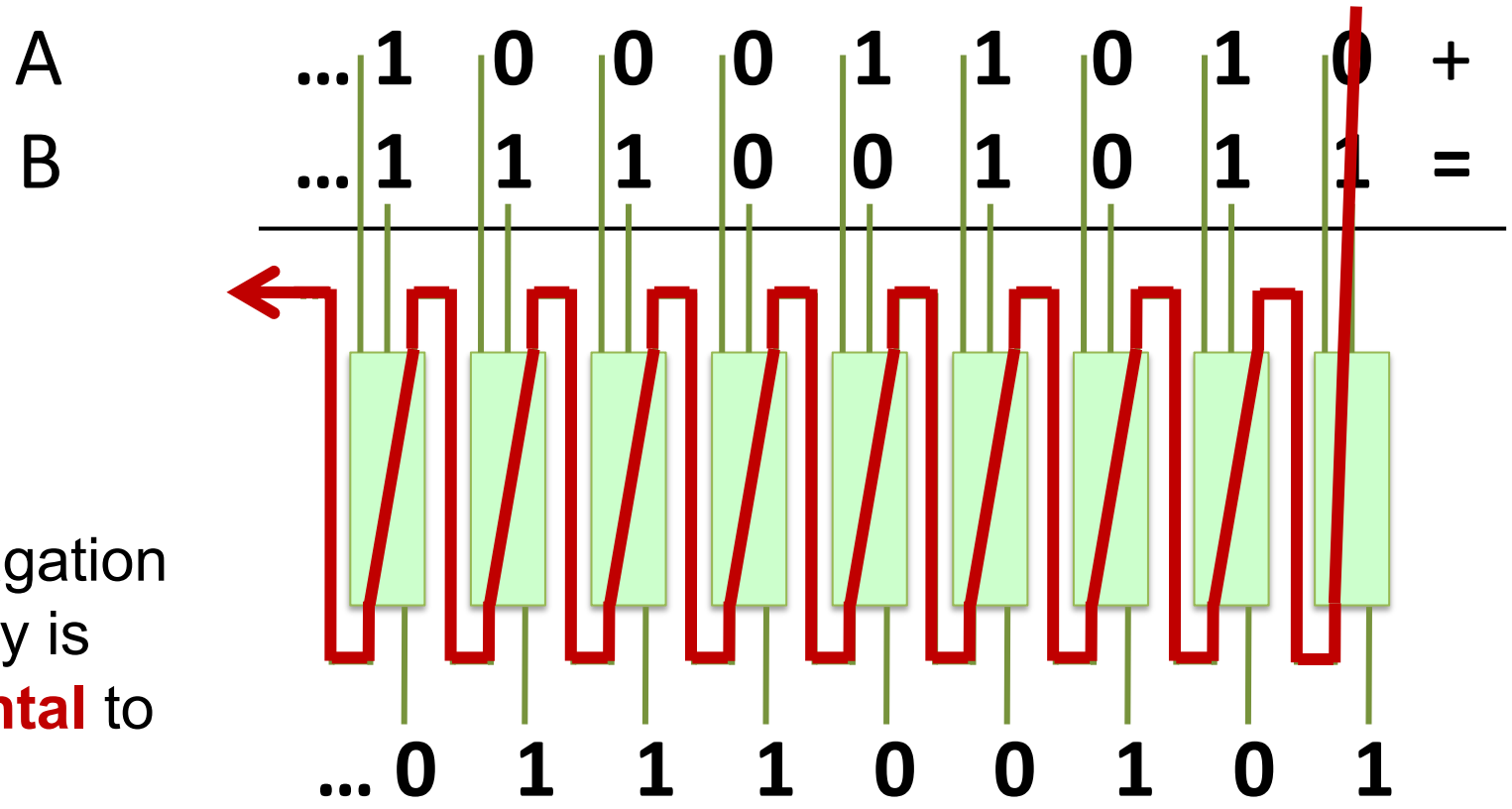
Making an Adder is simple…



Sum

0 + 0 + 0 = 00
0 + 0 + 1 = 01
0 + 1 + 0 = 01
0 + 1 + 1 = 10
1 + 0 + 0 = 01
1 + 0 + 1 = 10
1 + 1 + 0 = 10
1 + 1 + 1 = 11

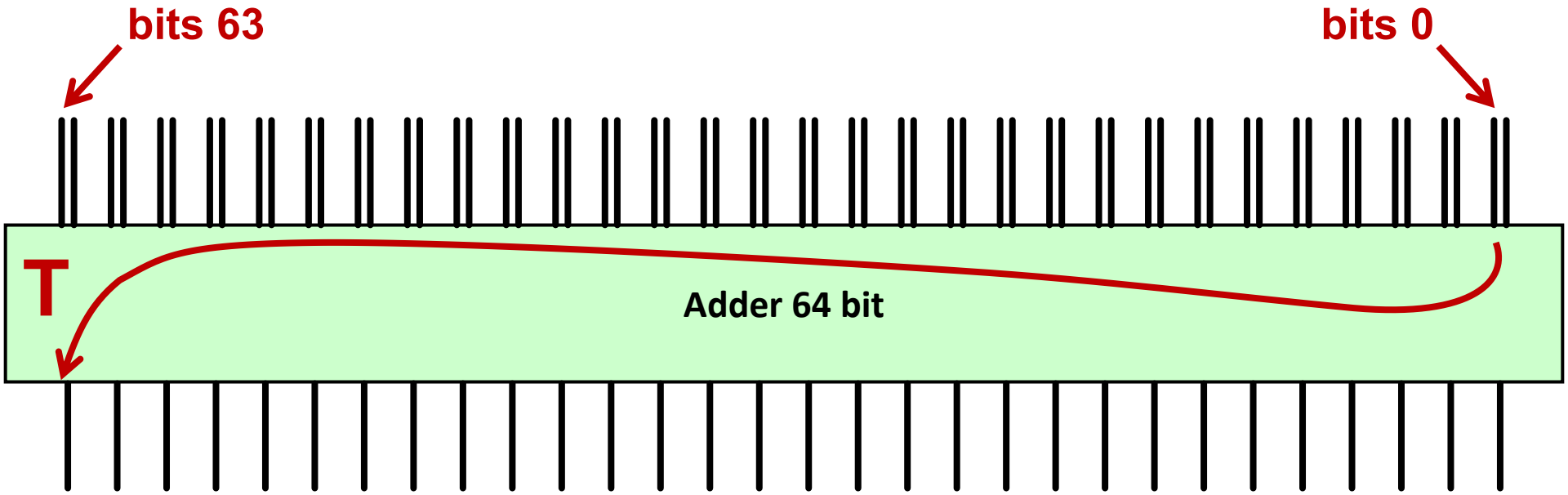
But this circuit is slow



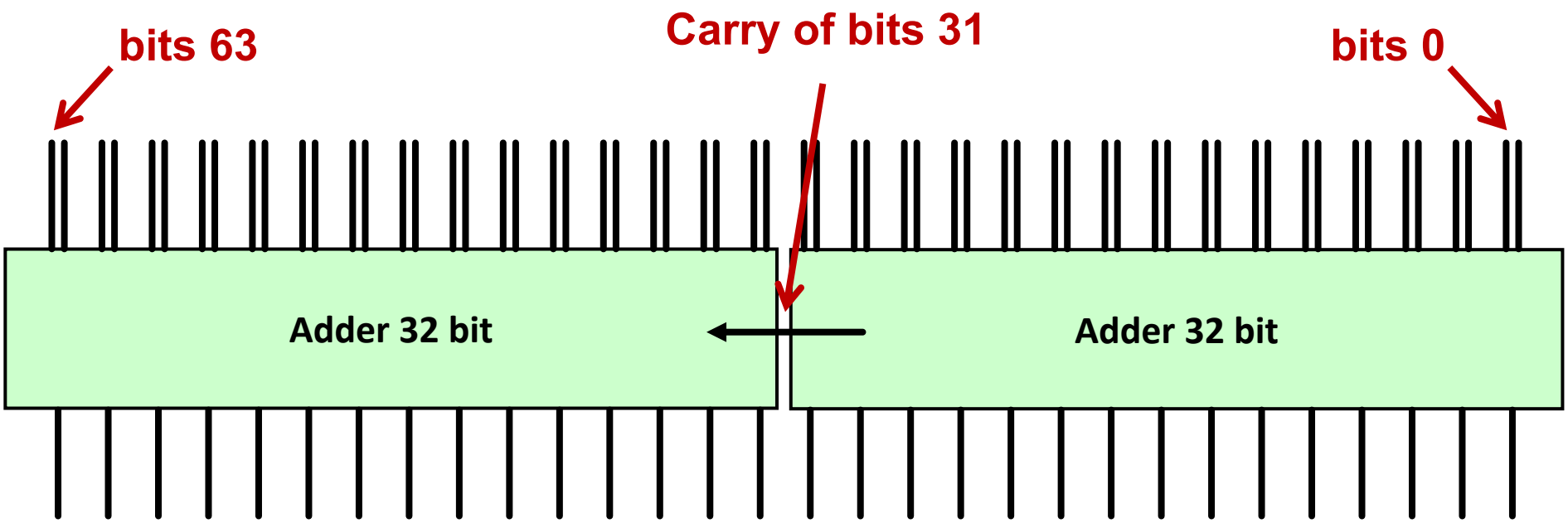
- The propagation of the carry is **fundamental** to an adder!

- By default, the **delay** of an adder is **proportional to the number of bits.**

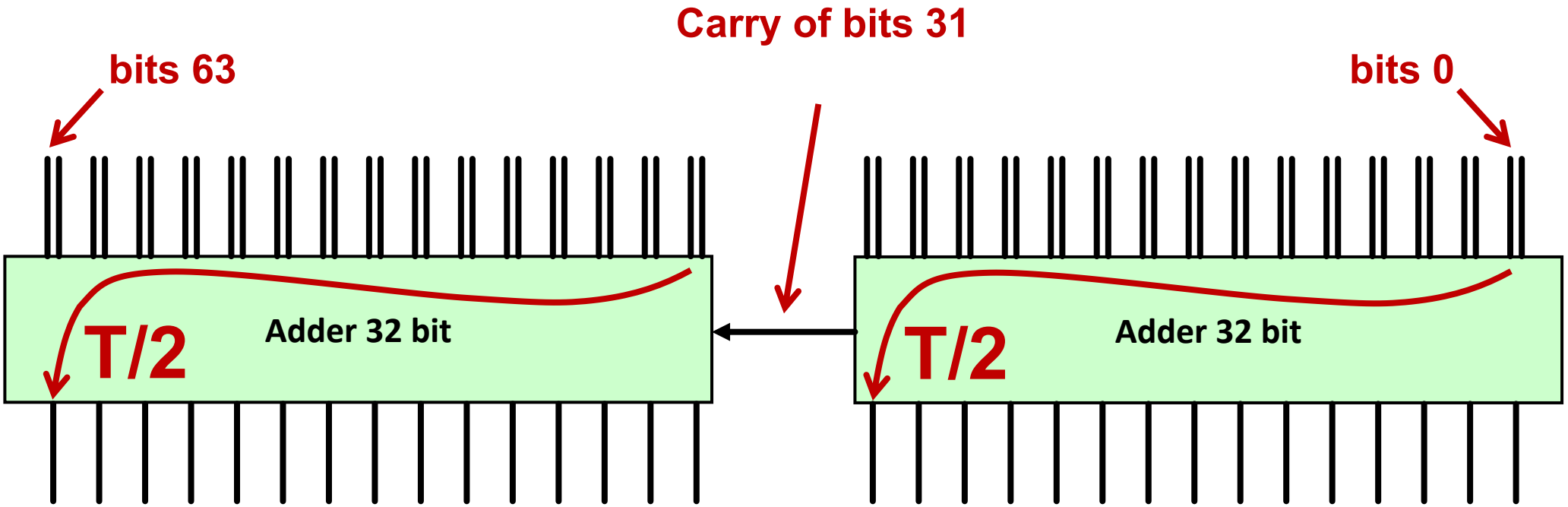
Can we do better?



Can we do better?

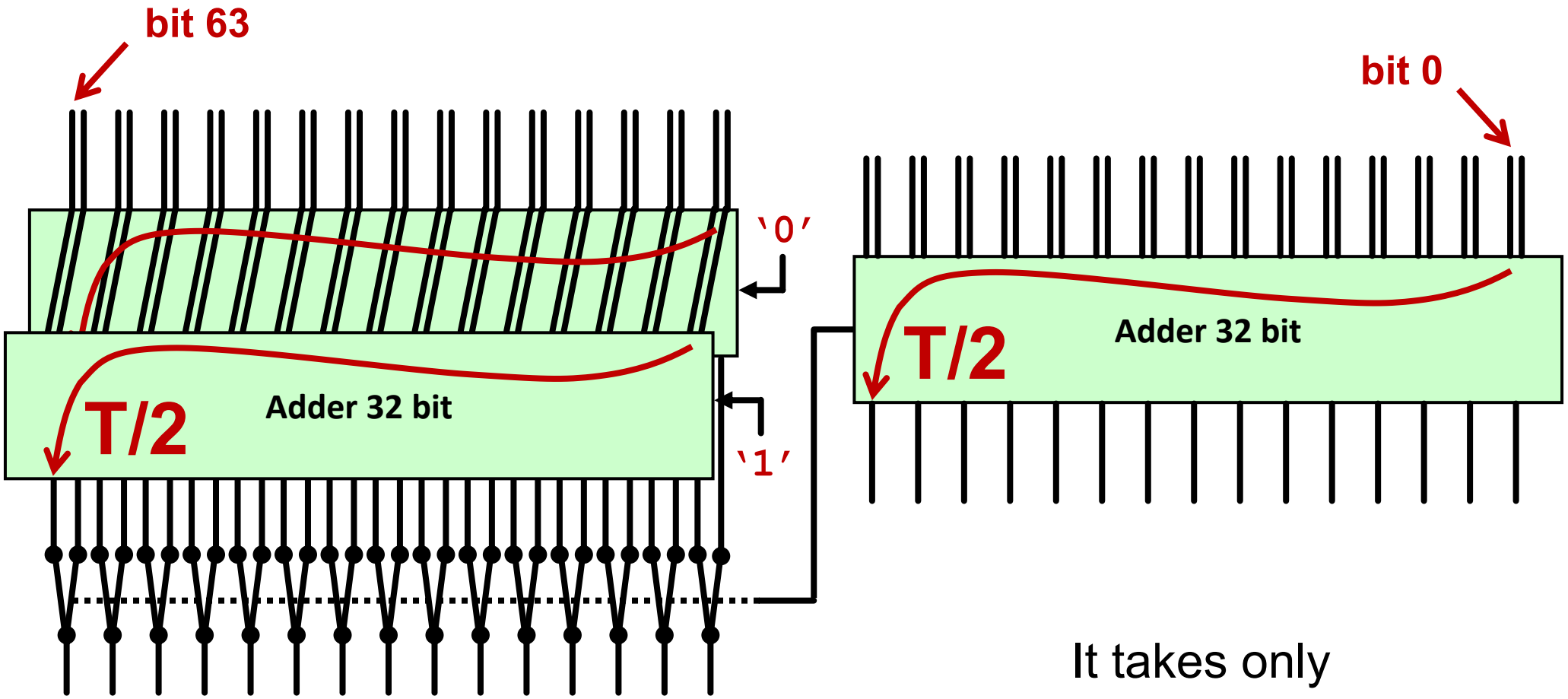


Can we do better?



We did not win **anything**…

Can we do better?



It takes only
half of the time!

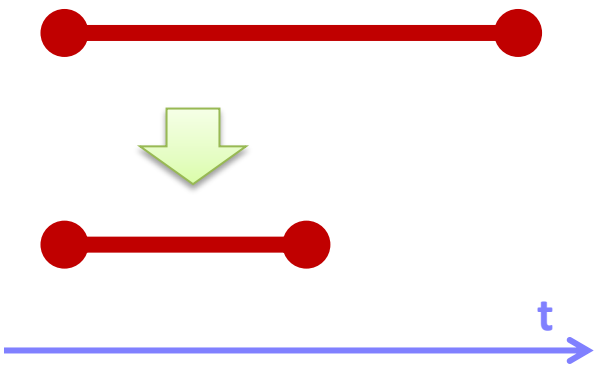
Computer Engineering

- We can change the performance of a circuit without changing the functionality.
- We can invest more transistors and more energy to get faster circuits.
- We can slow down circuits to save energy.

How to increase performance?

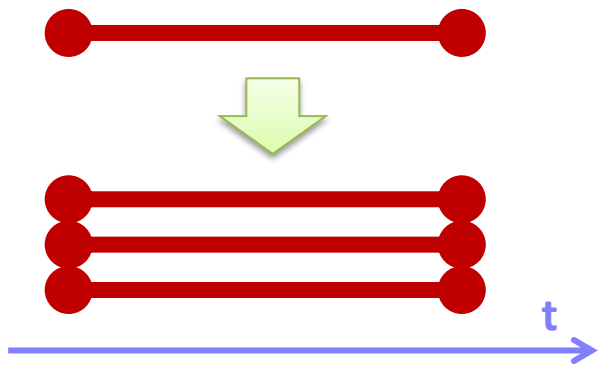
= Reduce delay

waiting time to get a result



= Increase the throughput

Number of results per time unit



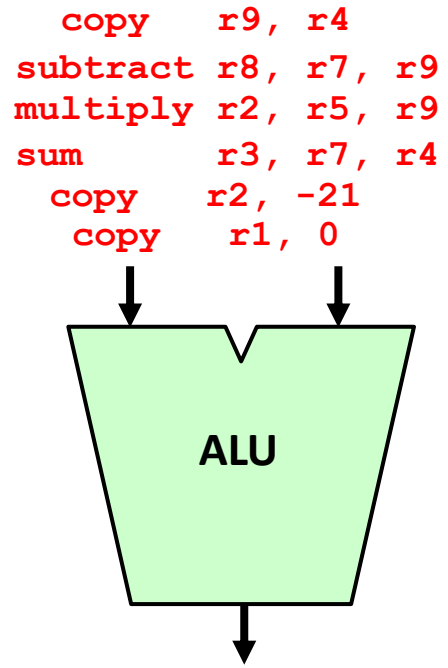
Two simple examples to improve performance:

1. At the level of the circuit:
   reduce the delay of an adder
2. At the level of the process structure:
   increase the throughput of instructions

Our Processor

```
103: copy     r1, 0
104: copy     r2, -21
105: sum      r3, r7, r4
106: multiply r2, r5, r9
107: subtract r8, r7, r9
108: copy     r9, r4
109: sum      r3, r2, r1
110: subtract r5, r3, r4
111: copy     r2, r3
112: sum      r1, r2, -1
113: sum      r8, r1, -1
114: divide   r4, r1, r7
115: copy     r2, r4
```

By default, we execute **one instruction** at a time



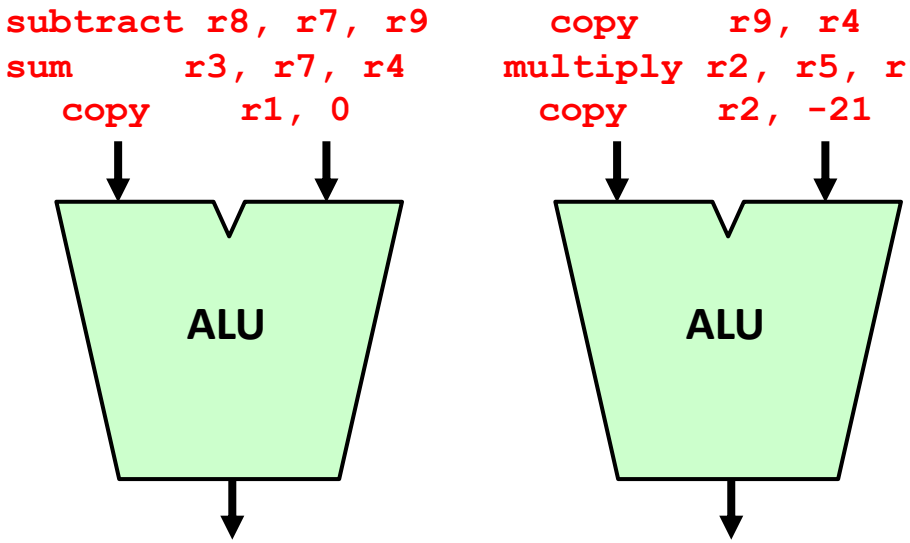
Can we do better?

Double Throughput of Processor

```
103: copy      r1, 0
104: copy      r2, -21
105: sum       r3, r7, r4
106: multiply  r2, r5, r9
107: subtract  r8, r7, r9
108: copy      r9, r4
109: sum       r3, r2, r1
110: subtract  r5, r3, r4
111: copy      r2, r3
112: sum       r1, r2, -1
113: sum       r8, r1, -1
114: divide    r4, r1, r7
115: copy      r2, r4
```

Problem?!

One could execute
two instructions at a time !

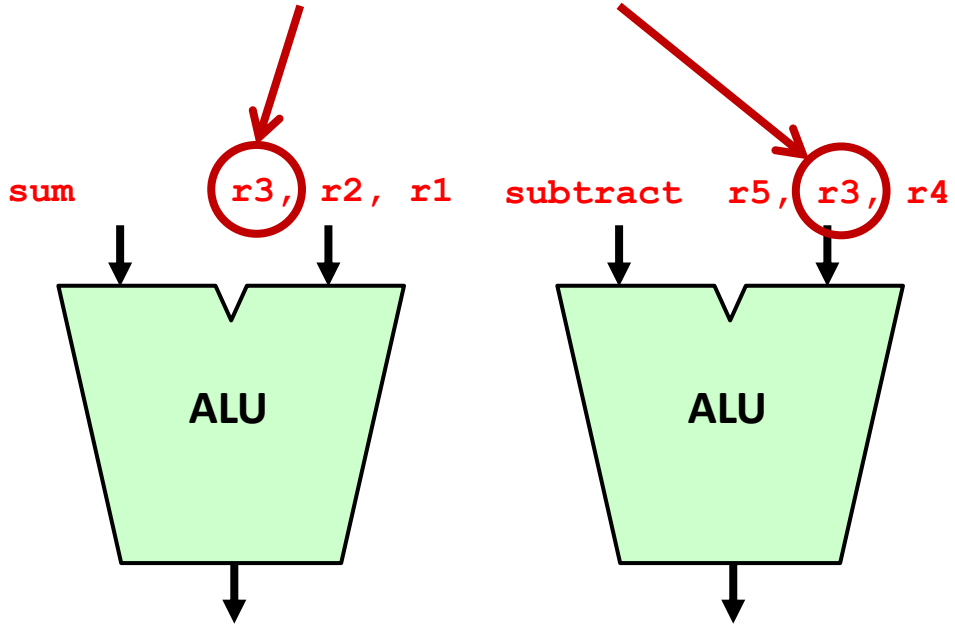


Double Throughput of Processor

```
103: copy     r1, 0
104: copy     r2, -21
105: sum      r3, r7, r4
106: multiply r2, r5, r9
107: subtract r8, r7, r9
108: copy     r9, r4
109: sum      r3, r2, r1
110: subtract r5, r3, r4
111: copy     r2, r3
112: sum      r1, r2, -1
113: sum      r8, r1, -1
114: divide   r4, r1, r7
115: copy     r2, r4
```

Problem?!

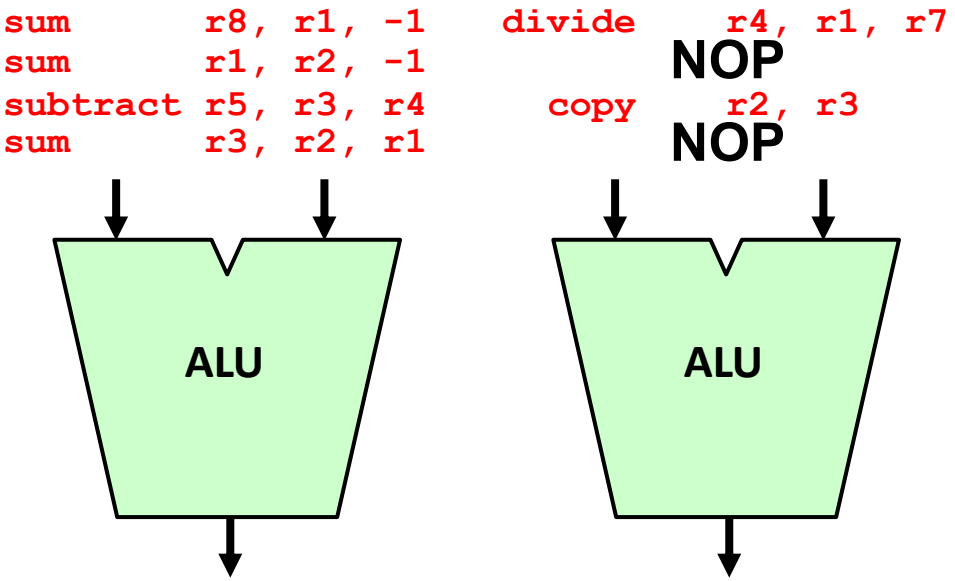
A problem occurs when the second instruction needs the result of the first computation!
If you are not careful,
the result is wrong!



Double Throughput of Processor

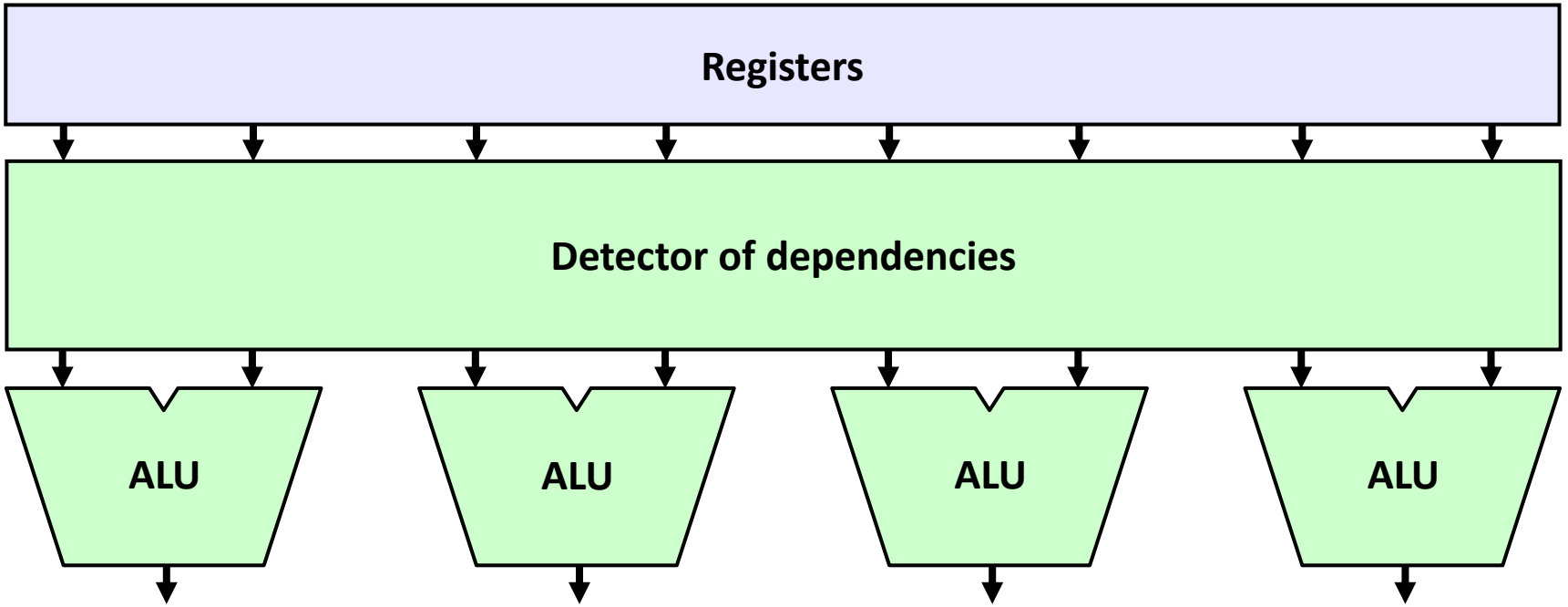
```
103: copy      r1, 0
104: copy      r2, -21
105: sum       r3, r7, r4
106: multiply  r2, r5, r9
107: subtract  r8, r7, r9
108: copy      r9, r4
109: sum       r3, r2, r1
110: subtract  r5, r3, r4
111: copy      r2, r3
112: sum       r1, r2, -1
113: sum       r8, r1, -1
114: divide    r4, r1, r7
115: copy      r2, r4
```

In pratice, one execute between **one and two instructions** at a time and the result is correct!



NOP … no operation

A Superscalar Processor



- All modern processors for laptops, phone, tablets, and for servers are of this type
- In addition, they reorder and execute the instructions before knowing if these instructions will be executed at all (e.g., they might be skipped due to an instruction like `jump`)

Computer Engineering

- We can change the system structure to run programs faster.
- We can add recourses to the processors to make them faster.
- We can use very basic processors to make them economical and energy efficient.

Summary

- From Algorithms to Computers
  - Assembly code = Basic instructions
  - Processor structure (registers, ALU, instruction counter, memories)
  - Transistors use to compute and store
- Performance
  - Reduce the delay of an adder
  - Increase the throughput of instructions