From Sequential Circuits to “Real” Computers

Lecturer: Guillaume Beslon
(Lecture adapted from Lionel Morel)

Computer Science and Information Technologies - INSA Lyon

Fall 2026

Introduction

- What we have done so far is implementing “simple” FSM by using Moore Machines
- BUT FSM cannot manipulate complex data (e.g., integers) because this would require too many states...
- Hum ... but digital circuits (and of course computers) DO deal with data!
- We need a methodology to have both:
  - The "security" of FSM (formal description of the behavior),
  - The ability to manipulate complex data.

  to build circuits manipulating data (typically integers) and ultimately real computers.

$\Rightarrow$ **Algorithmic State Machines**, aka control-data separation

From FSM to ASM

- FSM can have a VERY large number of states (typically larger than 2^{32})
- Conceiving such an FSM with a Moore machine is theoretically possible but practically impossible
- All machines dealing with numerical values typically have a very large number of states
- ASMs (Algorithmic State Machines) devide this large number of states between two machines:
  - A *datapath* dealing with *numerical values* $\rightarrow$ large number of states, simple flow
  - A *controller* dealing with *control flow* $\rightarrow$ low number of states, complex flow
- Both systems are synchronized on the same Clock
- **Control-Data separation principle**

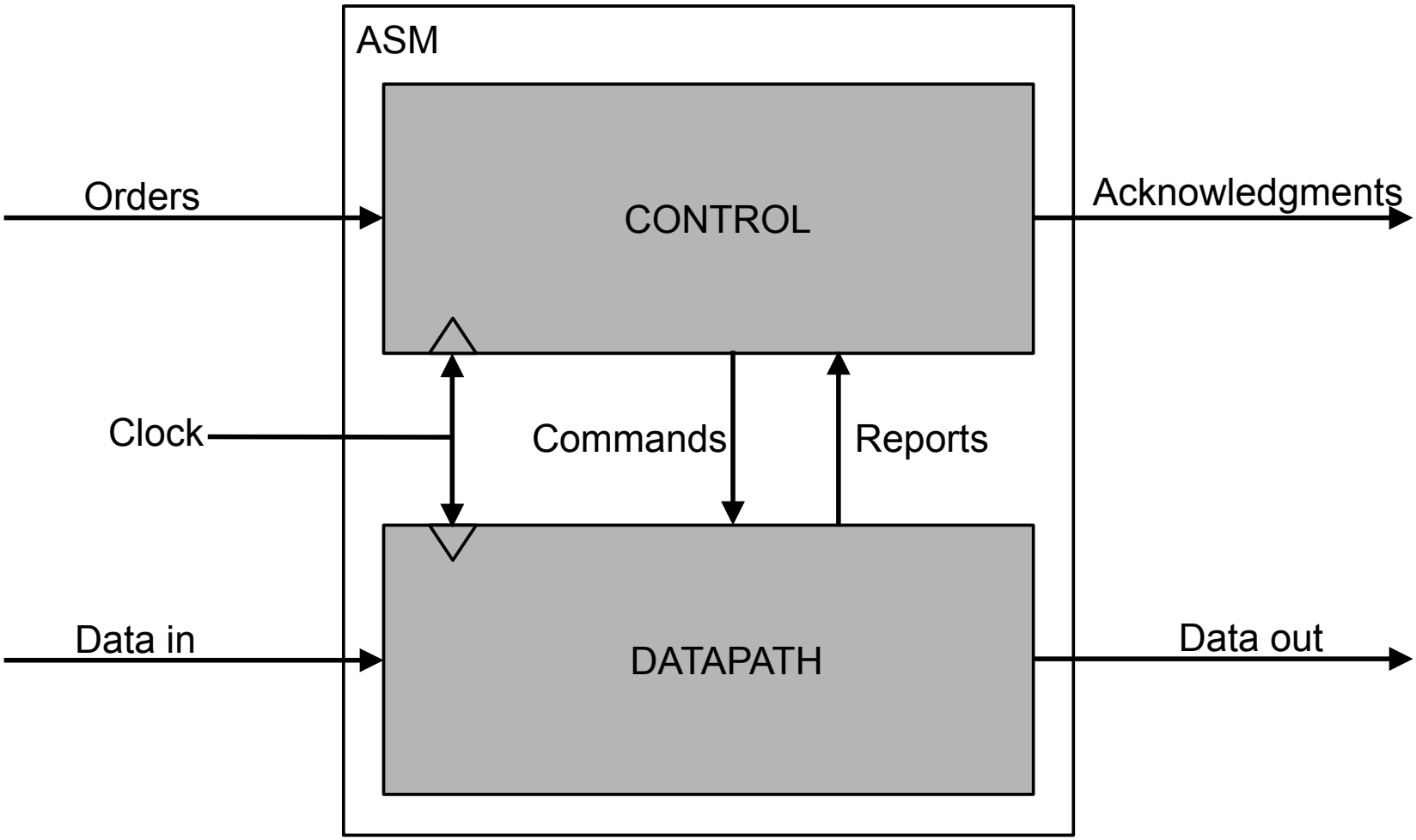
The Control-Data separation principle



From FSM to ASM: a simple example, the stopwatch



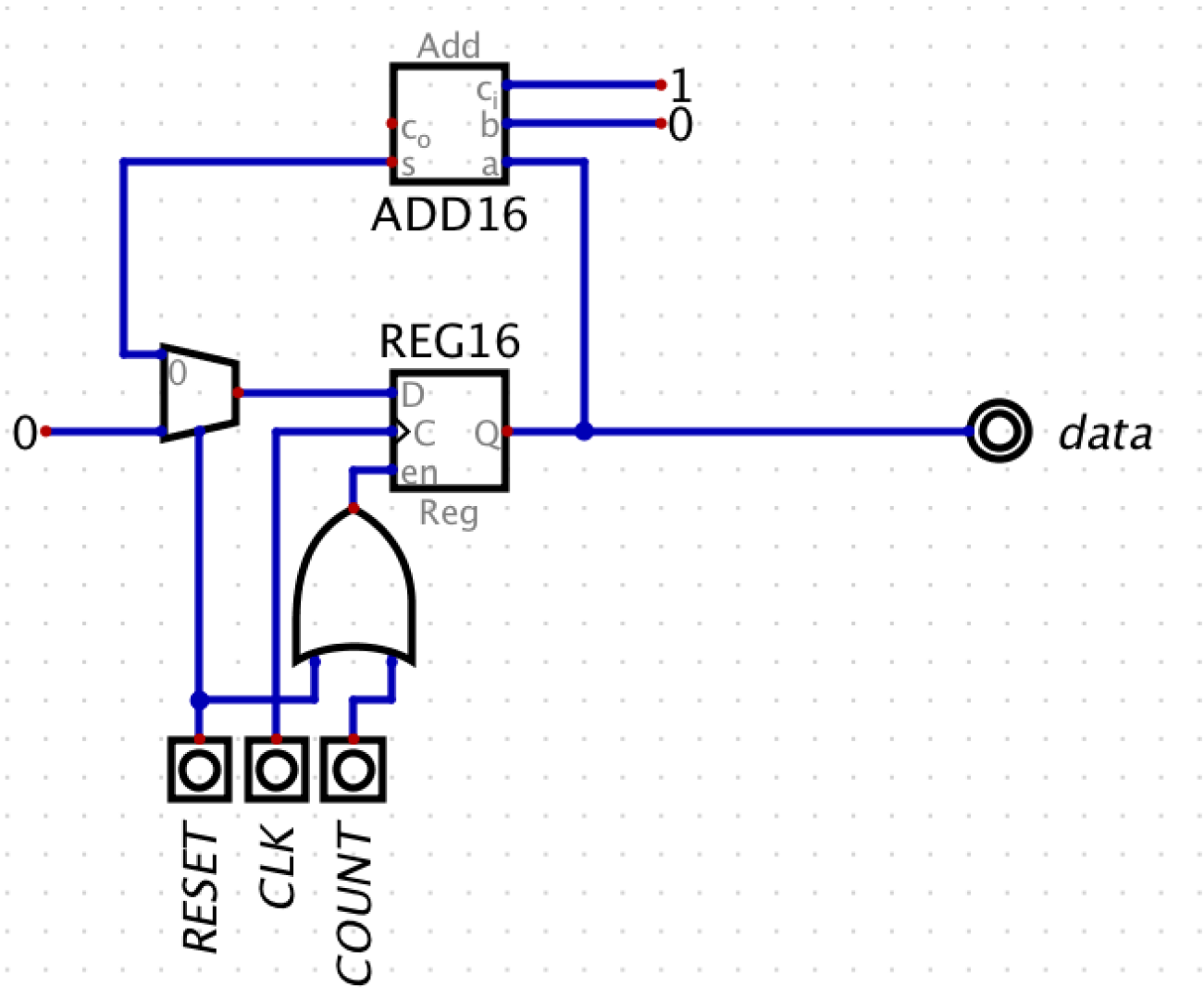
A Sequential Circuit within the Control-Data Separation Scheme



Datapath

- Offers **computational ressources** needed for the operations to be implemented
- Typically includes arithmetic and logical components (possibly integrated into an ALU – Arithmetic and Logic Unit) and **registers** connected by **buses** and **multiplexers**
- Exchanges data (in/out) with the outside of the circuit
- Performs all operations on **data**
- But typically doesn't know **which** operation to perform and **when** to perform it
- **Clock** drives registers (synchronous circuits)

The stopwatch datapath



Control

- Knows **which** operations to perform and **when**
- Doesn't deal with data directly (doesn't no **how** to do the operations)
- Typically implemented as a **Finite State Machine**, i.e., an **automaton** (see lecture 5)

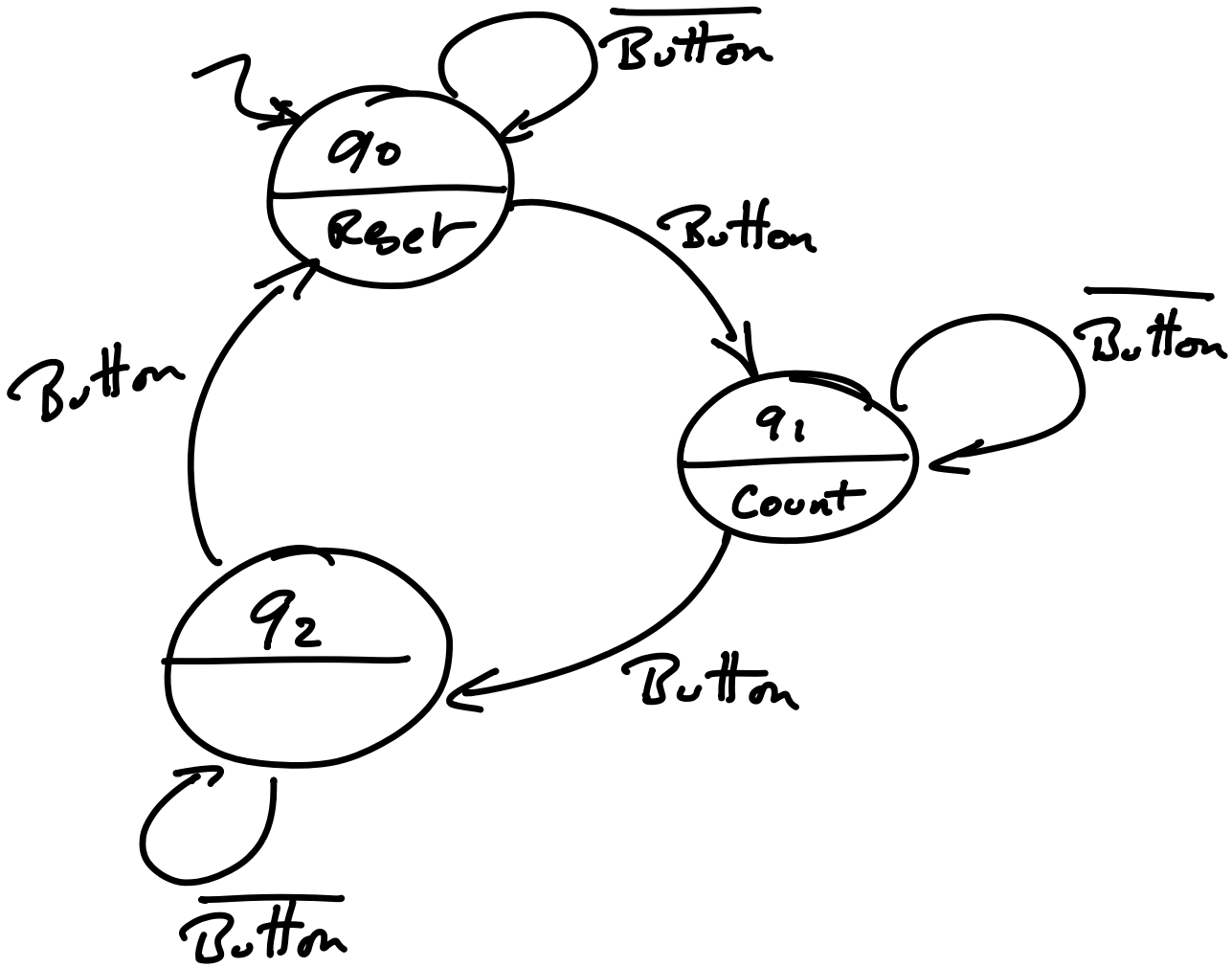
  Input alphabet: **Orders** (from the outside of the circuit) and **Reports** (from the datapath)

  Output alphabet: **Acknowledgements** (to the outside of the circuit) and **Commands** (to the datapath)
- **Clock** drives automaton state changes (synchronous circuits)

Control of the stopwatch

$I = \{BUTTON\}$

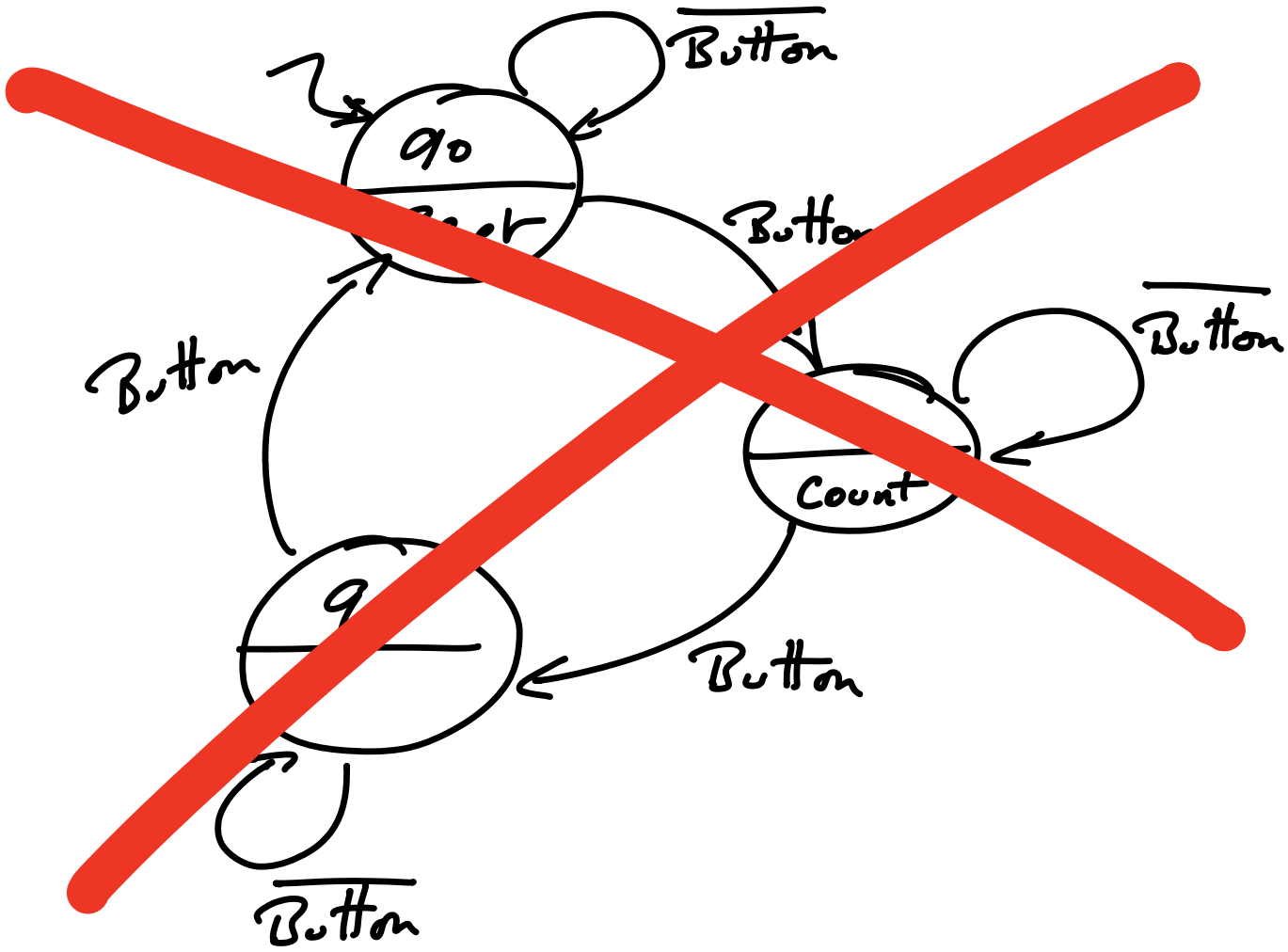
$O = \{COUNT, RESET\}$



Control of the stopwatch

$I = \{BUTTON\}$

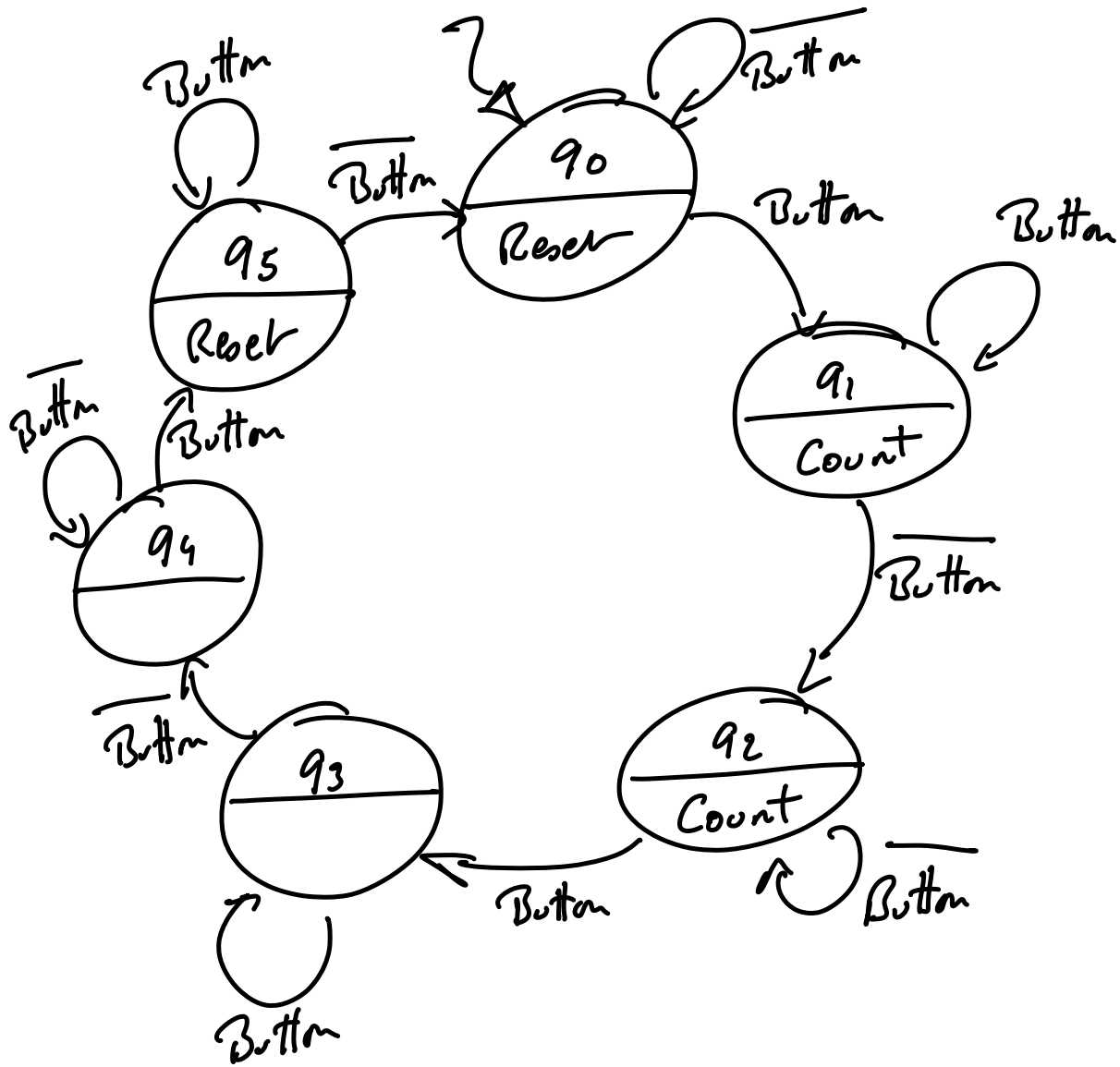
$O = \{COUNT, RESET\}$



Control of the stopwatch

$I = \{BUTTON\}$

$O = \{COUNT, RESET\}$



Control (cont'd)

Control is just about implemeting a Moore machine (no more, no less !!). Biggest difficulty is to **not forget any control signal**:

- Between Control and outside world (Orders, Acknowledgments)
- Between Control and Datapath (Commands, Reports)

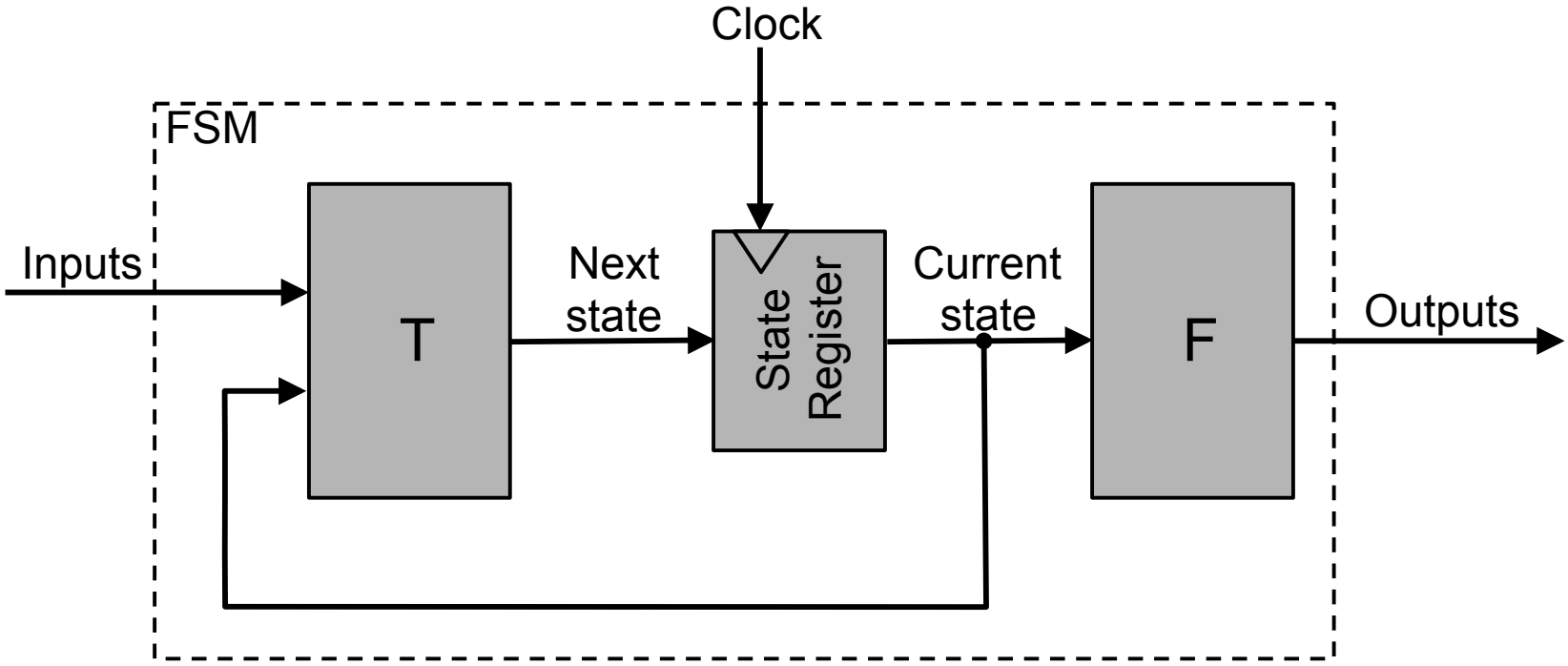
VERY IMPORTANT

- **Commands** will control datapath registers through their **enable** pin (NOT by modifying the clock signal!!!!)
- **Commands** control datapath routing through **mutiplexers**
- Control "never" has access to the data. It only receives **Reports computed by the datapath**. Reports are used to choose automaton transitions.

Control (cont'd)

Then control is implemented as a classical FSM (remember lecture 5)

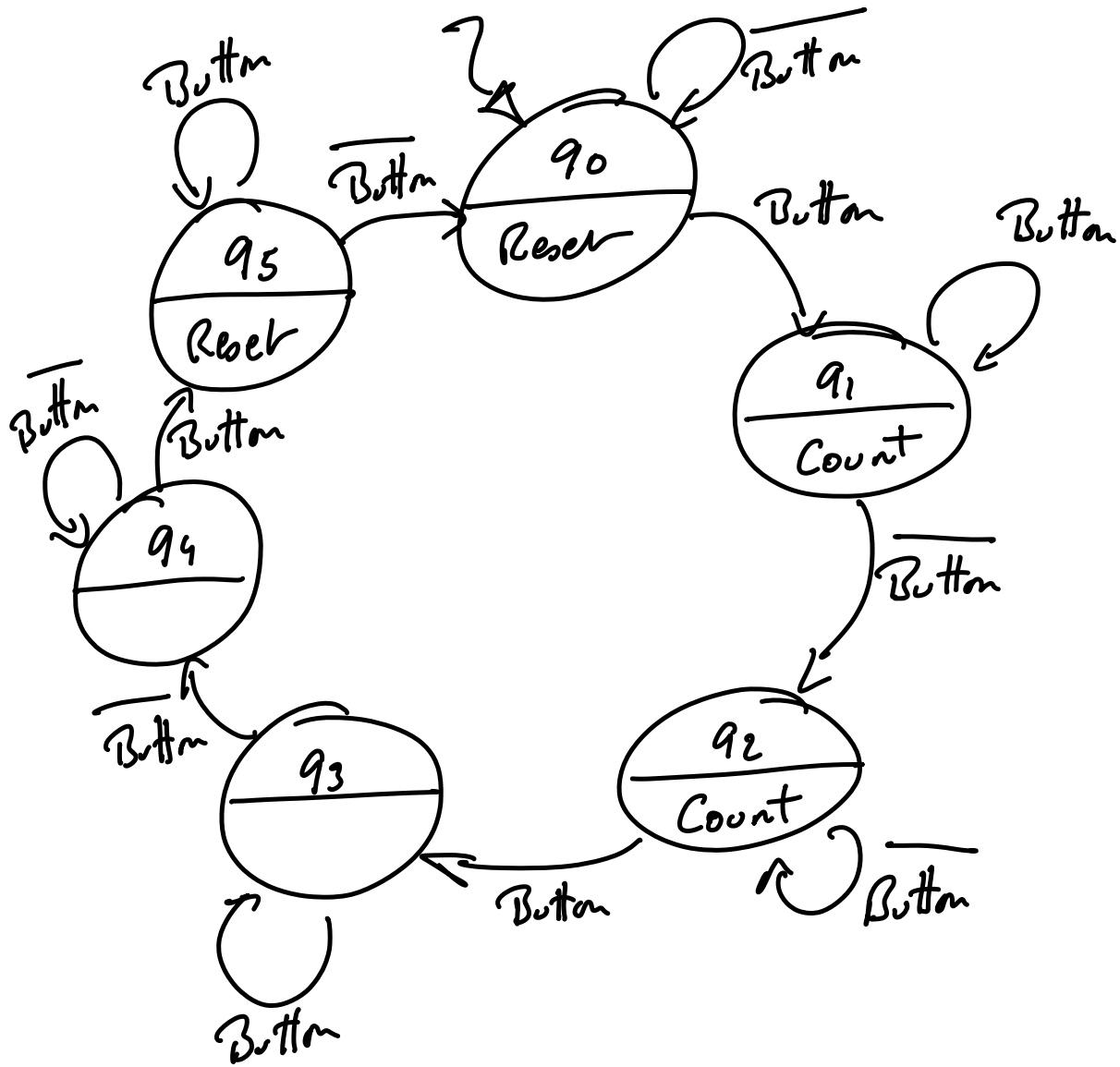
- $I = \text{Orders} \cup \text{Reports}$
- $O = \text{Acknowledgements} \cup \text{Commands}$
- $Q =$ set of states
- $T = Q \times (\text{Orders} \cup \text{Reports}) \rightarrow Q$ (transition function)
- $F = Q \rightarrow (\text{Acknowledgements} \cup \text{Commands})$ (output function)



Reminder: Control of the stopwatch

$I = \{BUTTON\}$

$O = \{COUNT, RESET\}$



Transition function for the stopwatch control

State	Button	Next State
q_0	0	q_0
q_0	1	q_1
q_1	0	q_2
q_1	1	q_1
q_2	0	q_2
q_2	1	q_3
q_3	0	q_4
q_3	1	q_3
q_4	0	q_4
q_4	1	q_5
q_5	0	q_0
q_5	1	q_5

$\Rightarrow$

s_2	s_1	s_0	Button	s_2'	s_1'	s_0'
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	1
0	1	1	0	1	0	0
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	0	0	0
1	0	1	1	1	0	1

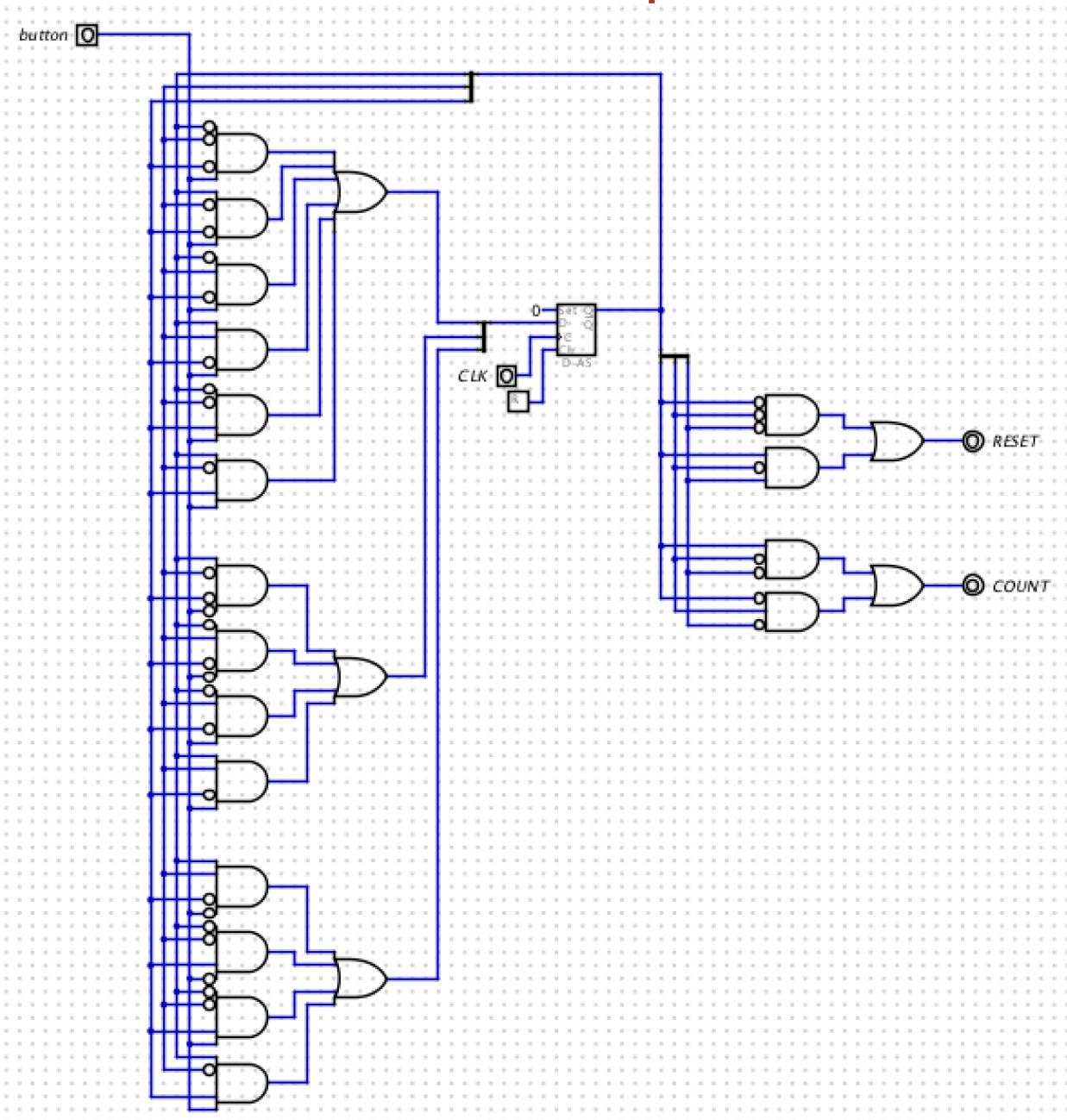
Output function for the stopwatch control

State	Reset	Count
q_0	1	0
q_1	0	1
q_2	0	1
q_3	0	0
q_4	0	0
q_5	1	0

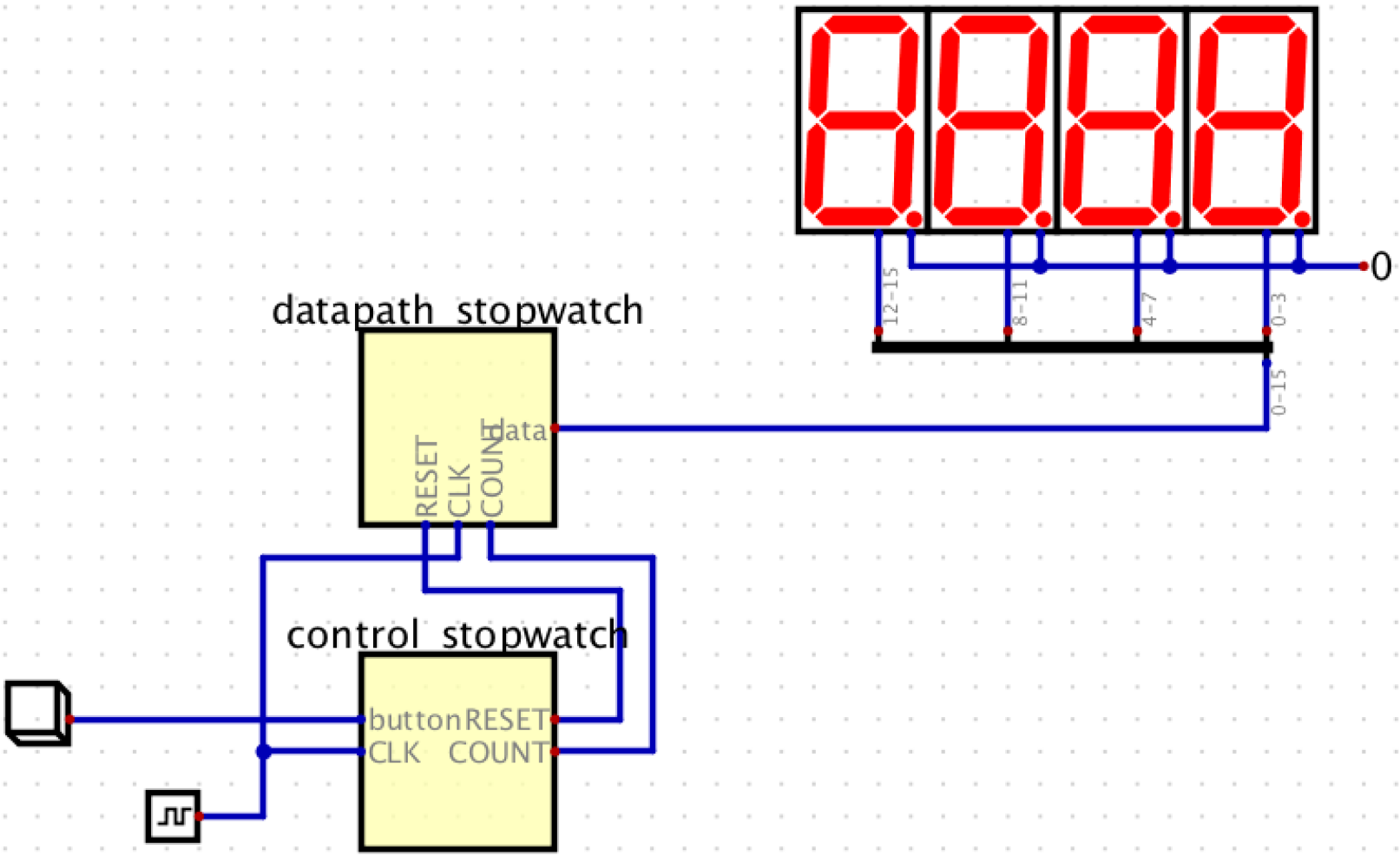
$\Rightarrow$

s_2	s_1	s_0	Reset	Count
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	0

Control circuit for the stopwatch



Stopwatch final circuit



Demo Time!

Conception Methodology

1. Start with:
   - The algorithm describing the expected behavior
   - The general scheme of an ASM
2. Using knowledge about circuit's environment and expected functionalities, identify **Orders** and **Acknowledgements**.
3. Build the Datapath:
   - Identify registers and computational resources (ALU)
   - Connect them such that all computations **can** be performed (including reports computation)
4. Design Datapath/Control interface (**Commands** and **Reports** signals). Interface will connect:
   - **Commands**: Outputs of the automaton to control the datapath (registers, plexers, ALU...)
   - **Reports**: Synthetic indicators of datapath state (e.g. ALU Flags). Sent to control
5. Transform the (unformal) algorithm into a Moore machine:
   - Identify states and transitions
   - Associate **Acknowledges** and **Commands** to each state.

Example: computing Syracuse sequence

Algorithm: Syracuse Sequence

1: **while** $n \neq 1$ **do**
2: **if** n is even **then**
3: $n \leftarrow n/2$
4: **else**
5: $n \leftarrow 3n + 1$
6: **end if**
7: **end while**

But this algorithm does not take important things into account:

- When does the computation start, how is n entered in the machine? $\rightarrow Go/\overline{Go}$
- What is the computation produces an oVerflow? $\rightarrow V/\overline{V}$

Example: computing Syracuse sequence (ASM-Friendly)

```
1: while forever do
2:     while $\overline{Go}$ do
3:         Input $n$
4:         Reset $V$
5:     end while
6:     while $GO$ AND $n \neq 1$ AND $\overline{V}$ do
7:         Print $n$
8:         if $n_0$ then
9:             $n \leftarrow n/2$
10:        else
11:            $n, V \leftarrow 3n + 1$
12:        end if
13:    end while
14:    if $V$ then
15:        while $GO$ do
16:            Print ERROR
17:        end while
18:    end if
19:    if $n = 1$ then
20:        while $GO$ do
21:            Print $n$
22:        end while
23:    end if
24: end while
```

Where are numerical data?

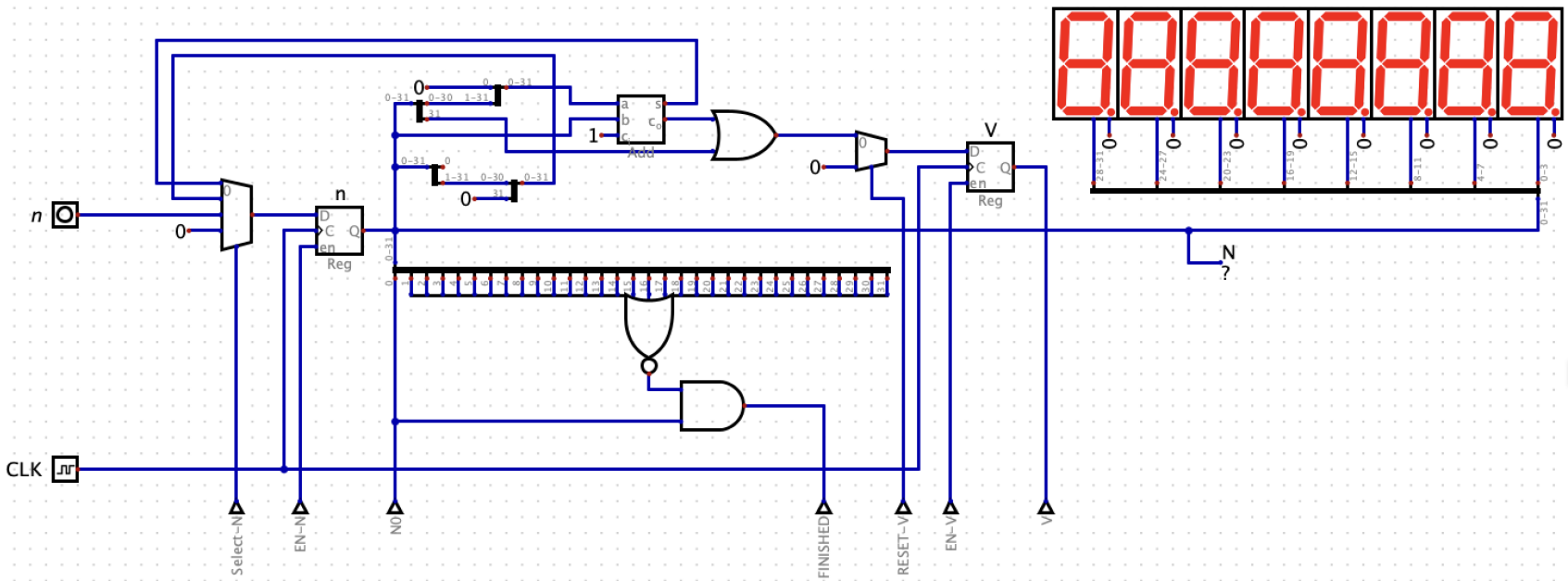
```
1: while forever do
2:     while \overline{Go} do
3:         Input n
4:         Reset V
5:     end while
6:     while GO AND n ≠ 1
   AND \overline{V} do
7:         Print n
8:         if n_0 then
9:             n ← n/2
10:        else
11:            n, V ← 3n + 1
12:        end if
13:    end while
14:    if V then
15:        while GO do
16:            Print ERROR
17:        end while
18:    end if
19:    if n = 1 then
20:        while GO do
21:            Print n
22:        end while
23:    end if
24: end while
```

Conceive the datapath

Remember: the datapath is in charge of ALL the computations...

- Here, the datapath needs to be able to:
  - store n on 32 bits (**Data in**) and V on 1 bit
  - compute $n/2$
  - compute $3n+1$
  - compute `V` (overflow of $3n+1$)
  - compute `FINISHED` when $n=1$
  - display n (in hexadecimal)
- The Datapath will send n_0, `V` and `FINISHED` to the Control Unit (these are **Reports**)
- The Datapath is controlled by the Control Unit through **commands**
  - `EN-N` to ask to write n
  - `EN-V` to ask to write V
  - `RESET-V` to reset V
  - `SELECT-N` to choose what to write in n among the three possibilities (data-in, $n/2$ or $3n+1$). `SELECT-N` is a vector of two bits.

Datapath for the Syracuse ASM

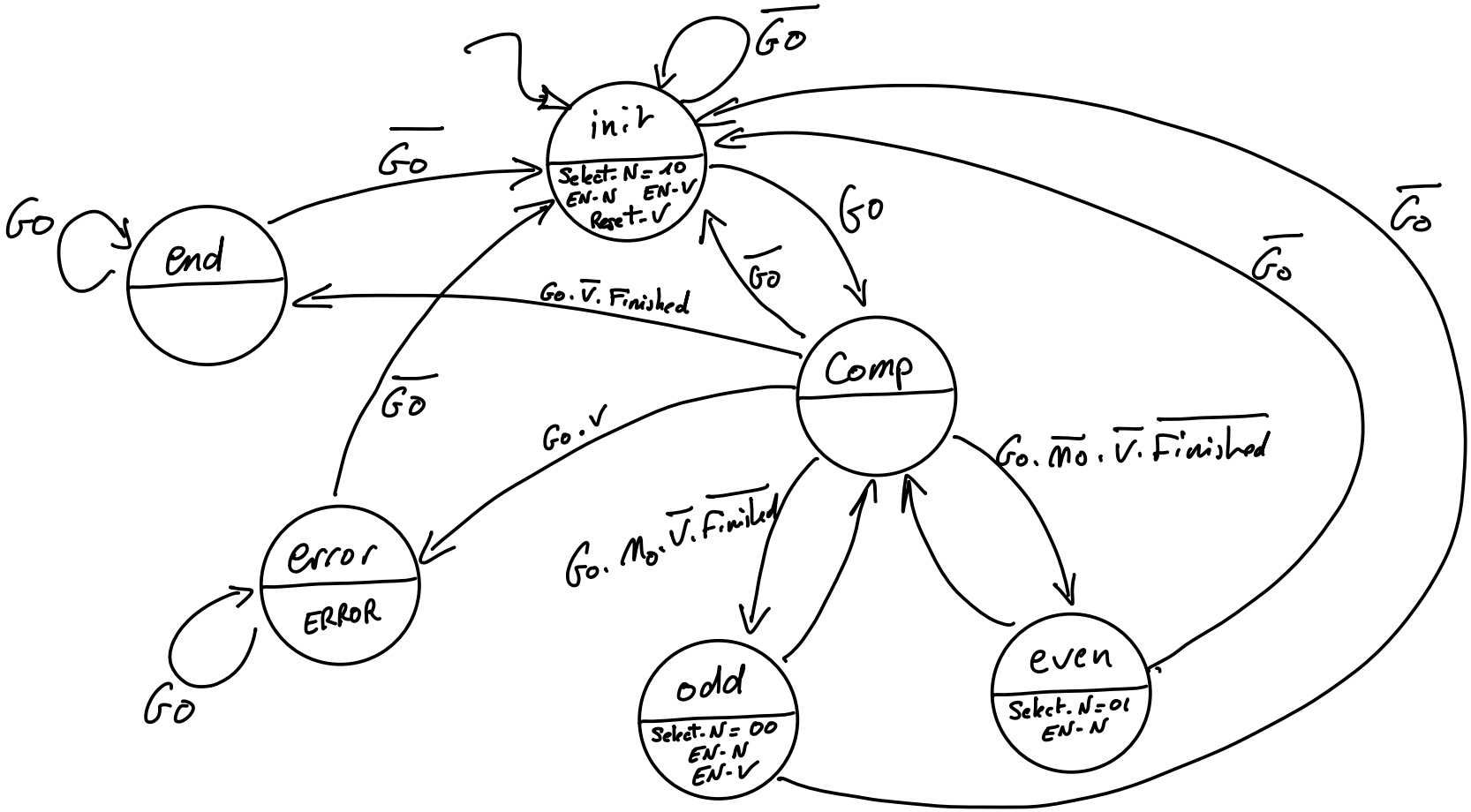


Simpler solutions are possible. This one strictly separates the control unit from the datapath...

Conceiving the Control Unit as an FSM

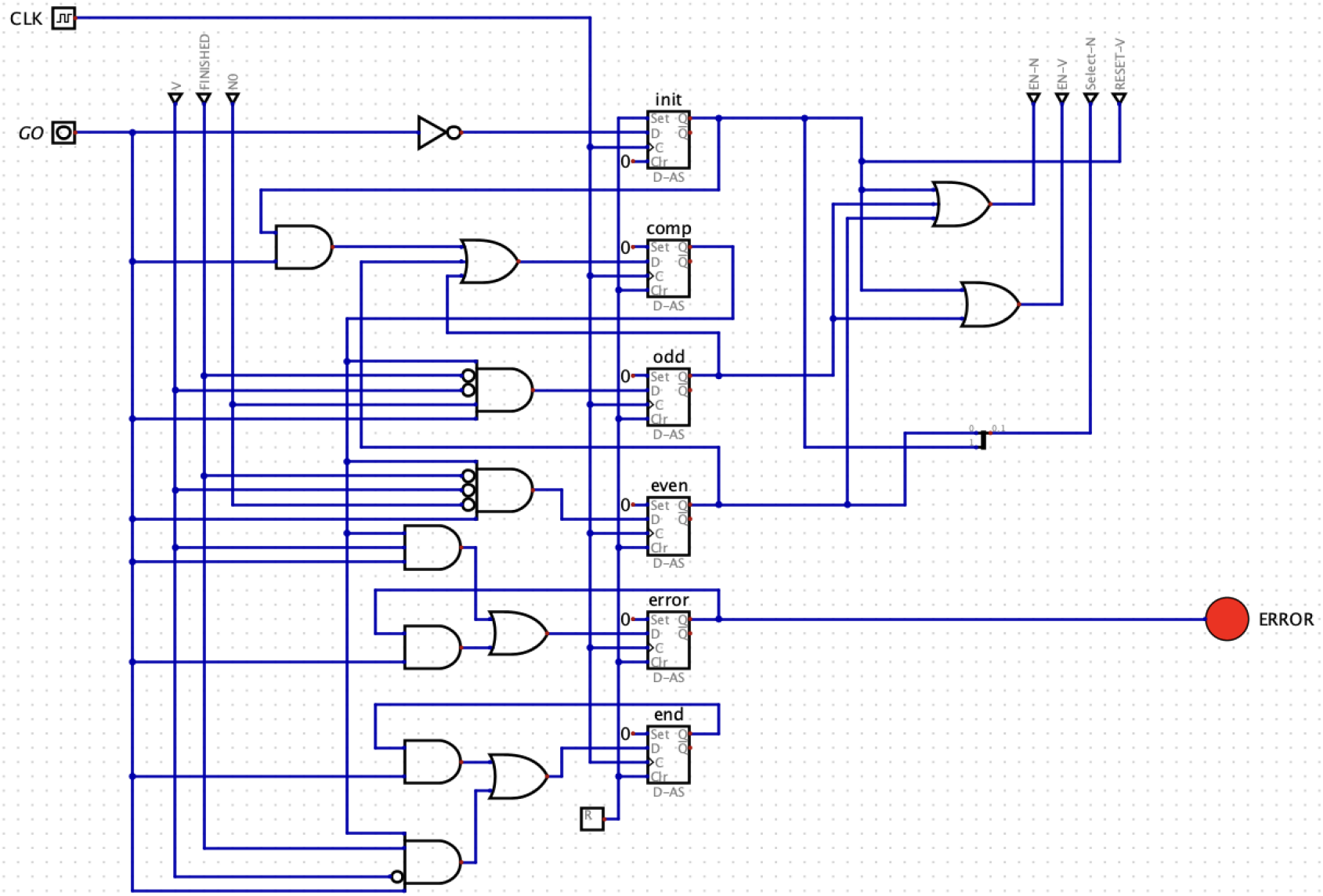
```
1:  while forever do
2:      while Go do
3:          Input n
4:          Reset V
5:      end while
6:      while GO AND n ≠ 1
        AND V do
7:          Print n
8:          if n_0 then
9:              n ← n/2
10:         else
11:             n, V ← 3n + 1
12:         end if
13:     end while
14:     if V then
15:         while GO do
16:             Print ERROR
17:         end while
18:     end if
19:     if n = 1 then
20:         while GO do
21:             Print n
22:         end while
23:     end if
24: end while
```

Control Unit for the Syracuse ASM (FSM)



Simpler solutions are possible. This one strictly separates the control unit from the datapath...

Control Unit for the Syracuse ASM (Circuit)



Demo Time!

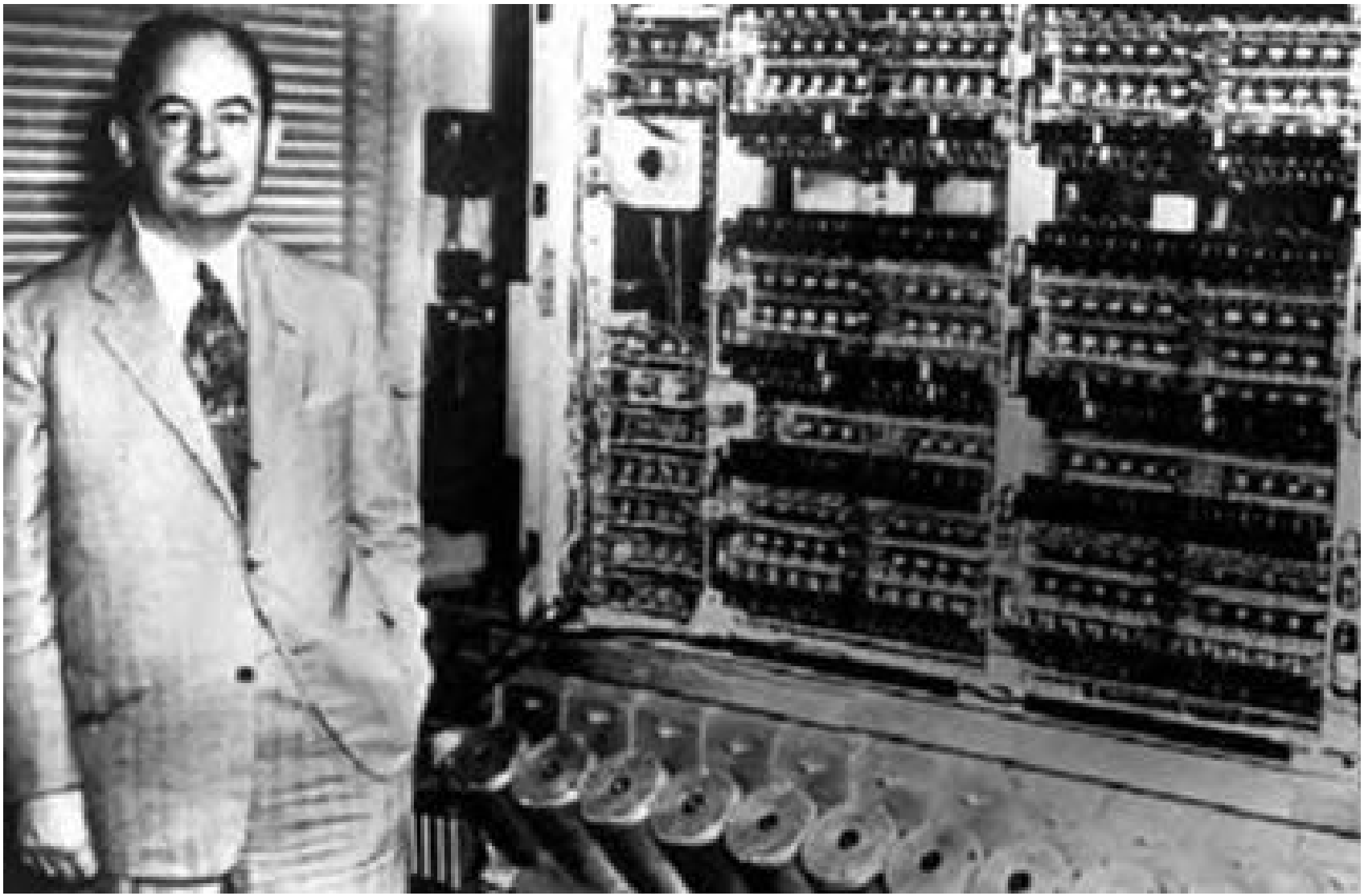
Final sprint: building a real (but simple) computer

- So far, we have implemented “simple” circuits that are able
  - to efficiently compute any binary function on one or several binary words;
  - to store large amounts of information in a memory hierarchy.
- The next step is to **assemble these circuits** into a versatile computer:
  - *i.e.*, a machine capable of running any program.
- To achieve this, we need to introduce the notions of **instruction** and **instruction set** (*jeu d'instructions*).
- **We will use Control-Data separation to build a sequential circuit which function will be to compute the execution of a sequence of instructions**

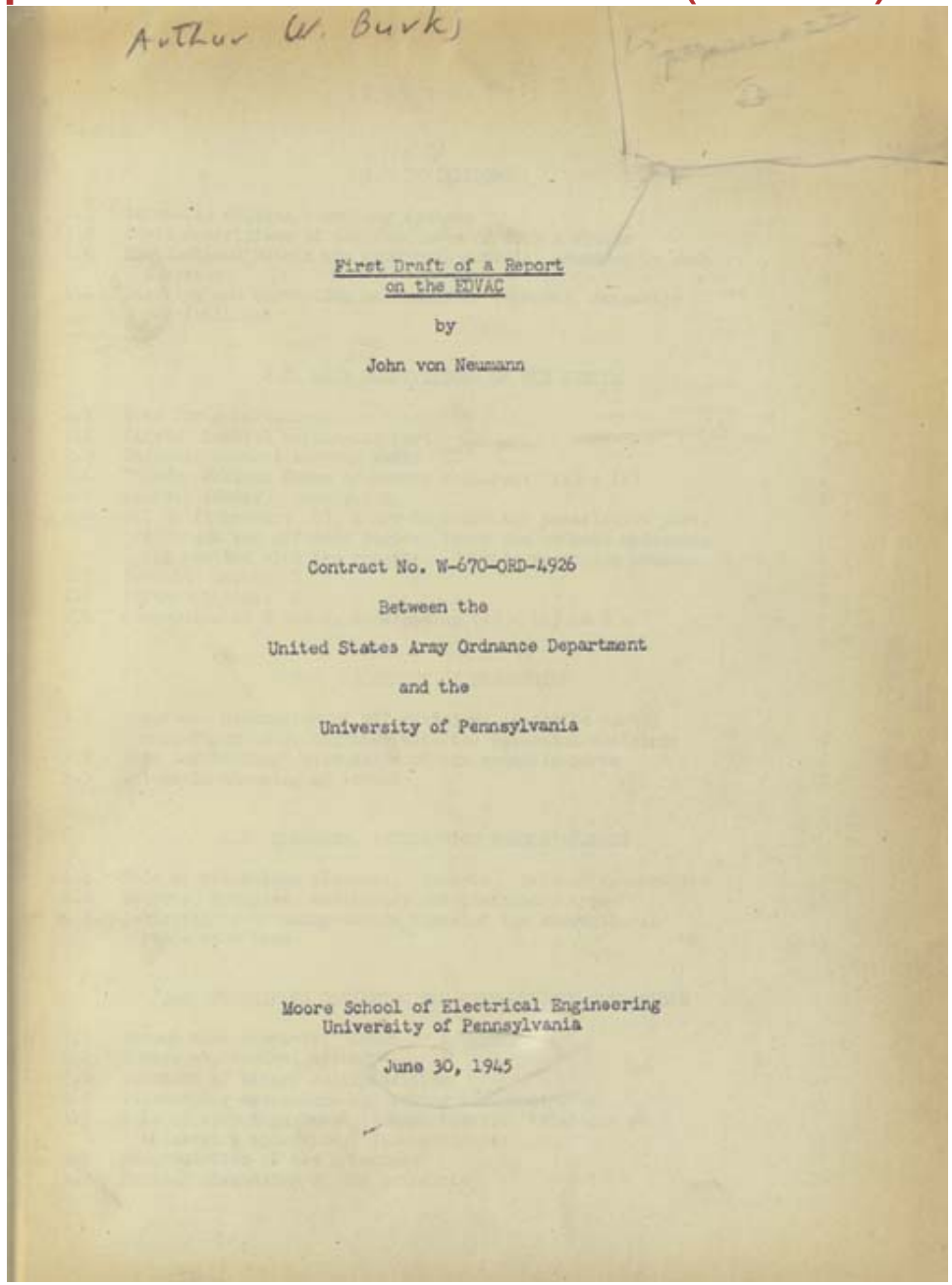
This idea is at the heart of “modern computers”...

$\Rightarrow$ von Neumann architecture

Von Neumann and the EDVAC



First Draft Report on the EDVAC (1945)



Arthur W. Burks

First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania

Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

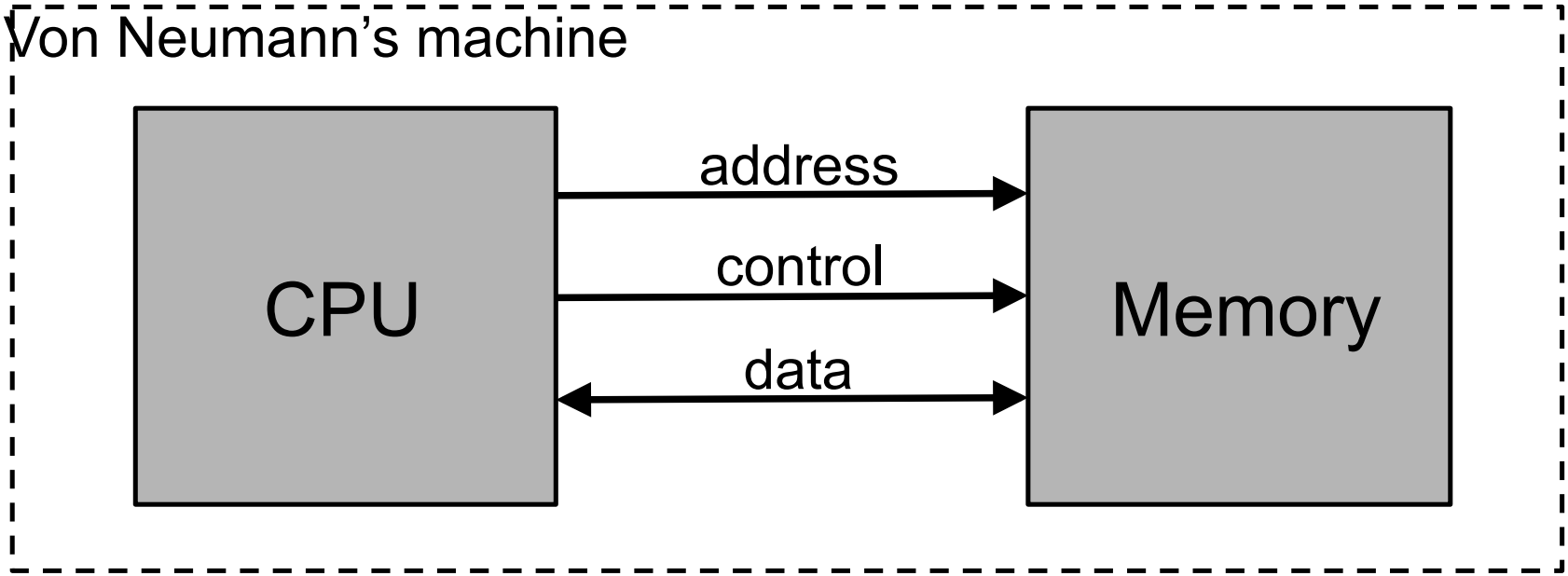
*The considerations which follow deal with the structure of a very high speed **automatic digital computing** system, and in particular with its logical control.*

*An automatic computing system is a (usually highly composite) device, which can **carry out instructions** to perform calculations of a considerable order of complexity—e.g. to solve a non-linear partial differential equation in 2 or 3 independent variables numerically.*

---

bold faces added by me :)

Basic concepts in von Neumann's architecture



A **Von Neumann's machine** is made of two main elements:

- a **memory** that stores a **sequence of instructions** (the **program**) and **data**;
- a **Central Processing Unit** (CPU) that fetches one instructions at a time from memory, decodes and executes it, and then moves on to the next instruction

  $\Rightarrow$ fetch–decode–execute cycle (aka Von Neumann's cycle).

What is an instruction?

- An **instruction** is a binary vector that tells the processor **what operation to perform and on which operands**.
- So an instruction typically specifies:
  - the **operation** to perform (e.g., add, subtract, load, store);
  - the **operands** involved in the operation;
  - possibly, where to **store the result**.
- For example, an instruction could be: `ADD R1,R2,R3`
  - Add the contents of registers `R1` and `R2`;
  - Store the result in register `R3`.
- Each instruction of the program is stored in the computer memory as a **binary vector** composed of its **opcode** (what it does) and of its **operands**
- The operands can be located at different places in the computer (in the memory, in registers...).
  - The way an operand is located is called the **addressing mode**. Typical computers have *many* addressing modes!
  - The opcode typically indicates both the instruction and its addressing mode

More on Addressing Modes

- Typical computers support different **addressing modes** that are useful for different purposes:

  Immediate The instruction contains the operand value itself (a constant).

  Direct The instruction contains the operand location (*e.g.*, a memory address or a register).

  Indirect The instruction specifies a location that contains the address of the operand.

  Indexed `address = base_address + offset`. `offset` is provided by the instruction while `base_address` is stored in a register.

  Implicit The operand is implied by the instruction itself.
- Modern processors support many addressing modes!
- The instruction encoding specifies the **operation** and how its operands are accessed.
- In assembly code, addressing modes are indicated by writing conventions (e.g., `$0x64`, `%eax`, `-0x8(%rbp)`, ...).

What is an instruction set?

- An **instruction set** is the collection of all instructions that a processor can understand and execute.
- It defines the **language of the machine (Instruction Set Archirecture – ISA)**:
  - which operations are available;
  - which operands can be used;
  - how instructions are encoded in binary;
  - how data and memory are accessed $\rightarrow$ addressing mode.
- Typical instructions include:
  - **Arithmetic**: `ADD`, `SUB`, `MUL`, ...
  - **Logic**: `AND`, `OR`, `XOR`, ...
  - **Memory**: `LOAD`, `STORE`, ...
  - **Flow Control**: `JUMP`, `BRANCH`, `CALL`, ...

The instruction set is the interface between software and hardware.

From Instructions to Programs

- A **program** is simply a sequence of instructions.
- The processor executes these instructions one after another:

$$I_1 \longrightarrow I_2 \longrightarrow I_3 \longrightarrow \cdots$$

- For example, computing $c = a + b$ could be done with the following sequence:

```
LOAD    a,R1        # a → R1
LOAD    b,R2        # b → R2
ADD     R1,R2,R3    # R1+R2 → R3
STORE   R3,c        # R3 → c
```

- Both instructions and data (a, b, and c) are represented as binary information and stored in **memory**.
- By changing the sequence of instructions, we can change what the computer does. A programmable computer fetch and execute instructions stored in memory.

Recapitulation: Basic Concepts in von Neumann's Architecture

- A von Neumann machine executes **instructions**.
- A program is a sequence of instructions ordered **sequentially**.
  - This sequence defines the **control flow**.
- All instructions supported by the processor form the **instruction set**.
- There are **three main types of instructions**:
  - Data management (`LOAD`, `MOVE`, ...)
  - Arithmetic and logic (`ADD`, `MUL`, `NOT`, `SHL`, `SUB`, ...)
  - Control flow (`JUMP`, `JGE`, `JLE`, ...)
- Each instruction is stored in memory as a **binary vector** composed of an **opcode** and one or more **operands**.
- Operands can be located in different places, such as **registers** or **memory**.
  - The way an operand is located is its **addressing mode**.
  - The instruction encoding specifies the operation and how its operands are accessed.

Basic concepts in von Neumann's architecture

```
1       int main()
2       {
3         int x,i;
4         x = 0;
5         i = 0;
6         for (i = 1; i<100;i++)
7           {
8             x = x+1;
9           }
10        return x;
```

Basic concepts in von Neumann's architecture

```
1       int main()
2       {
3         int x,i;
4         x = 0;
5         i = 0;
6         for (i = 1; i<100;i++)
7           {
8             x = x+1;
9           }
10      return x;
```



```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```

Basic concepts in von Neumann's architecture

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```



```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

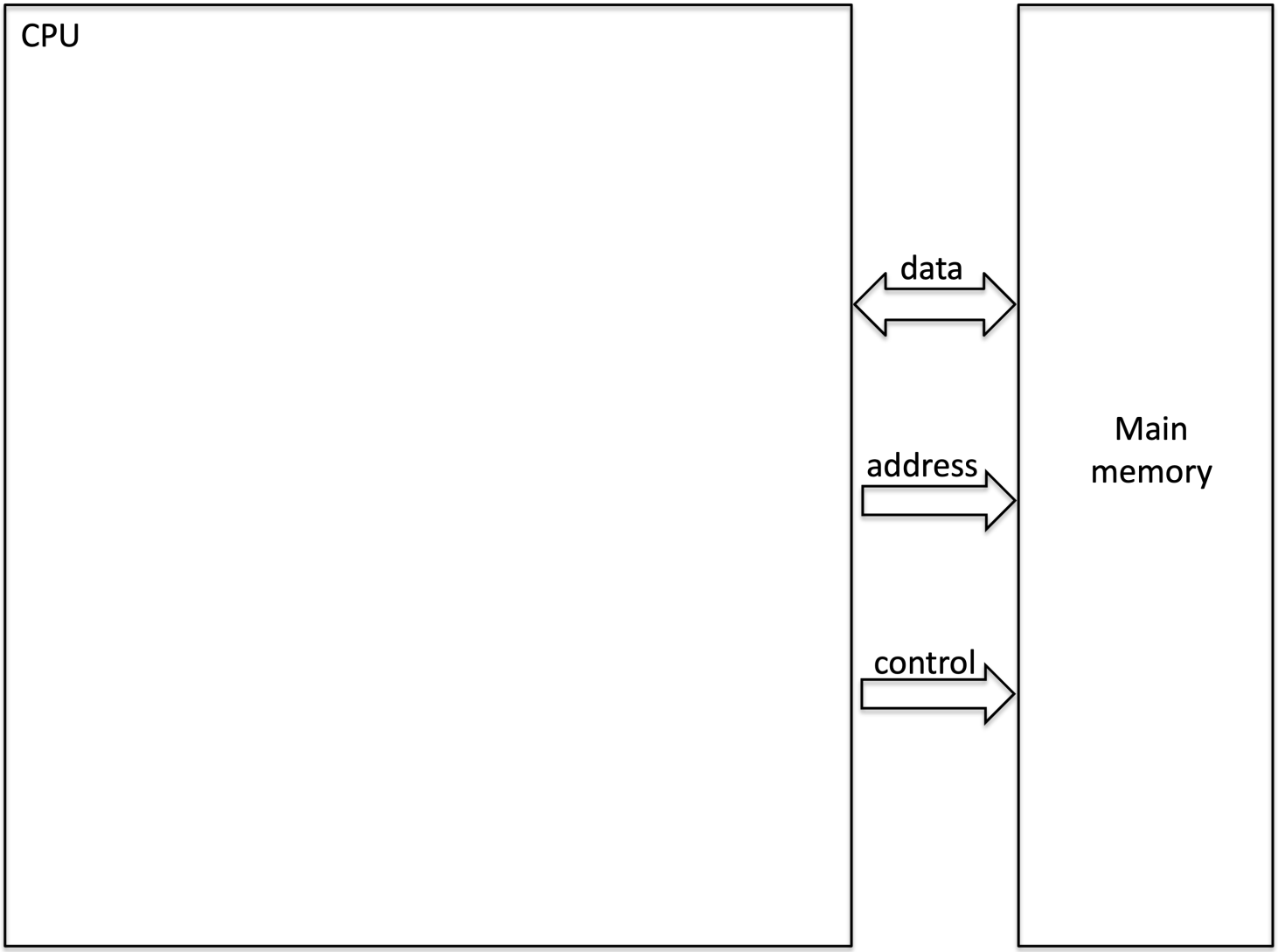
Basic concepts in von Neumann's architecture

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```

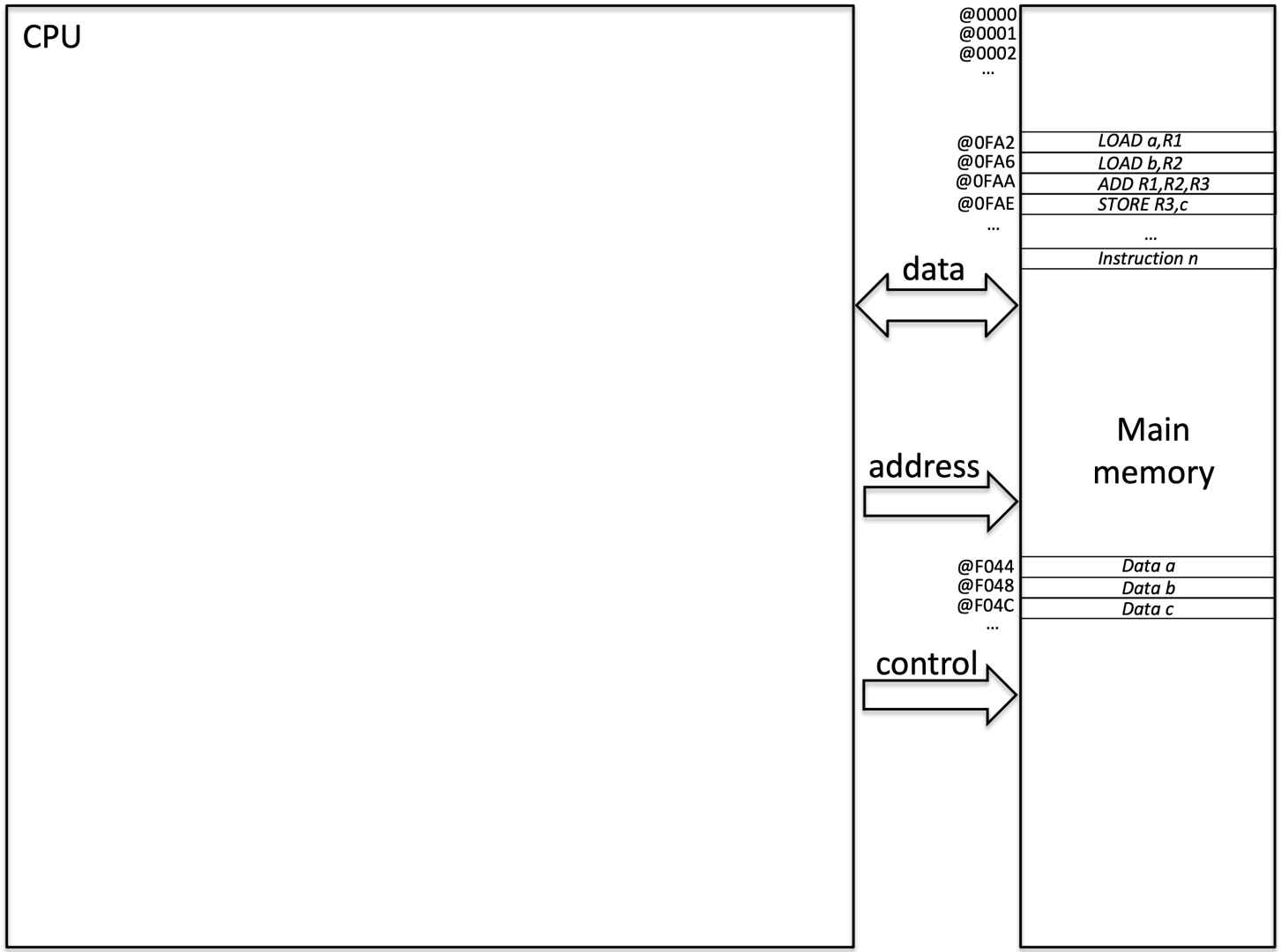


```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

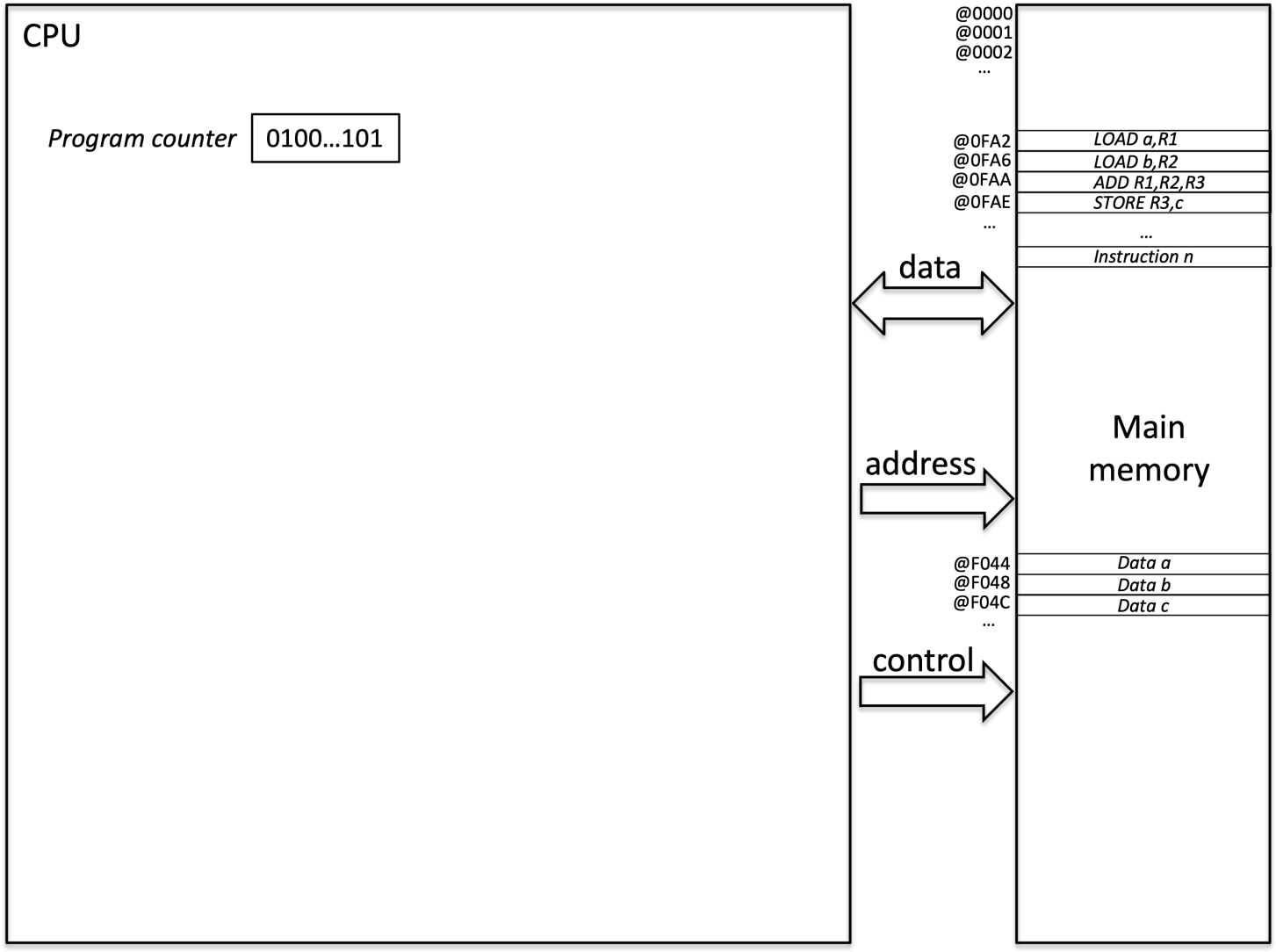
What does a CPU need to be able to execute a program?



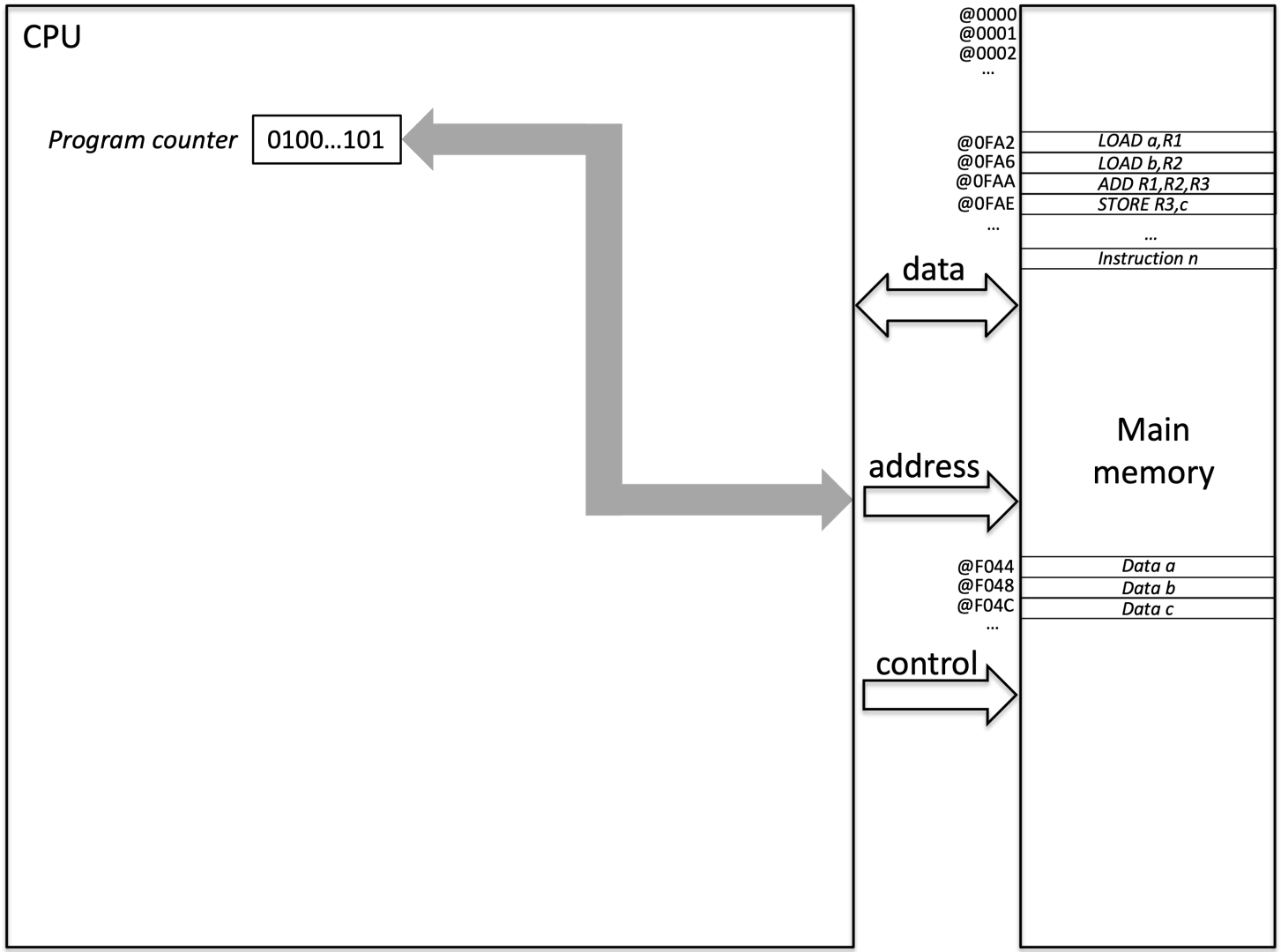
It needs a memory with the program and its data in two separate places...



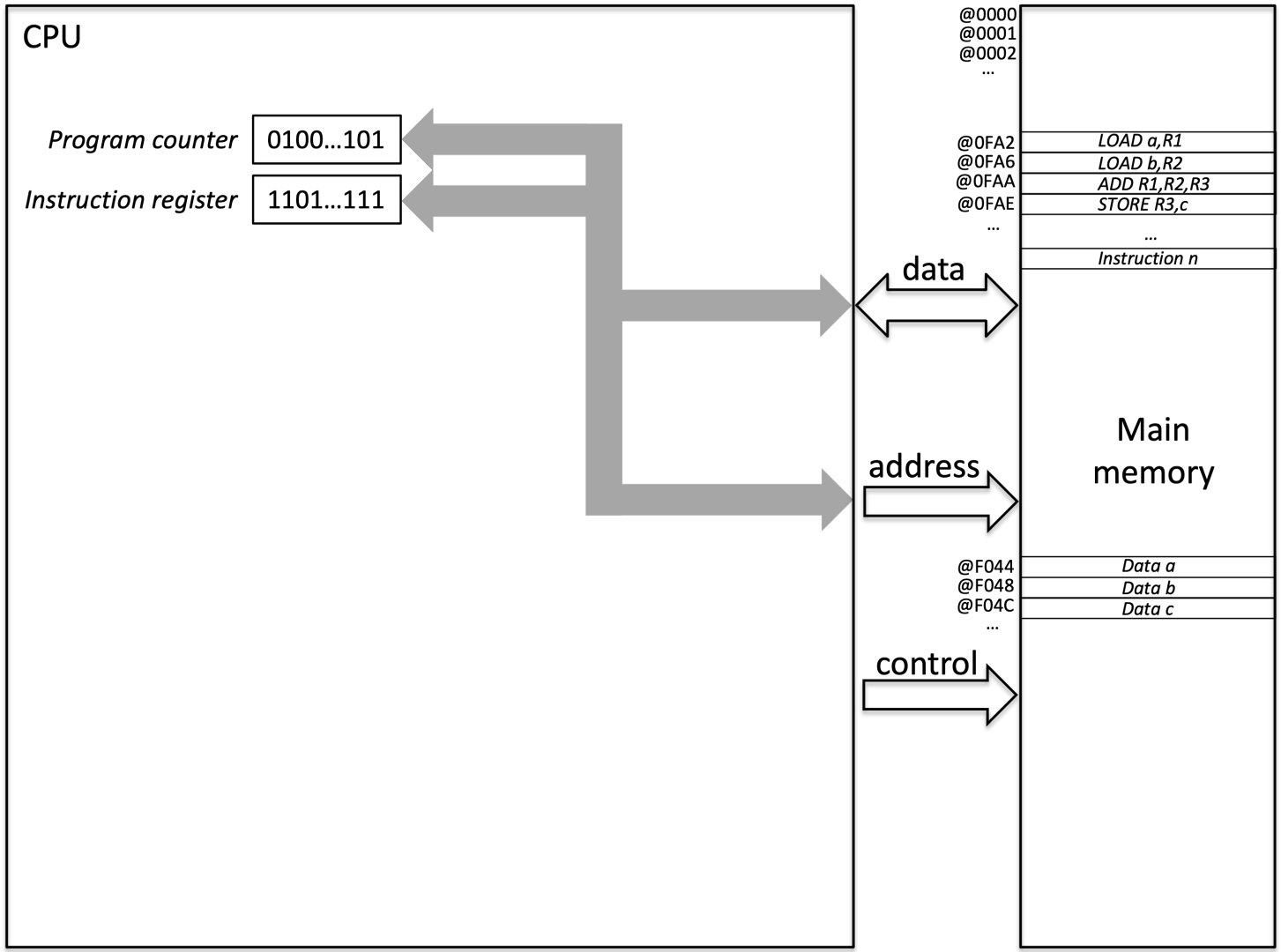
It needs a place to store the address of the instruction
→ **program counter**



...and some wires to send this information to the memory address bus

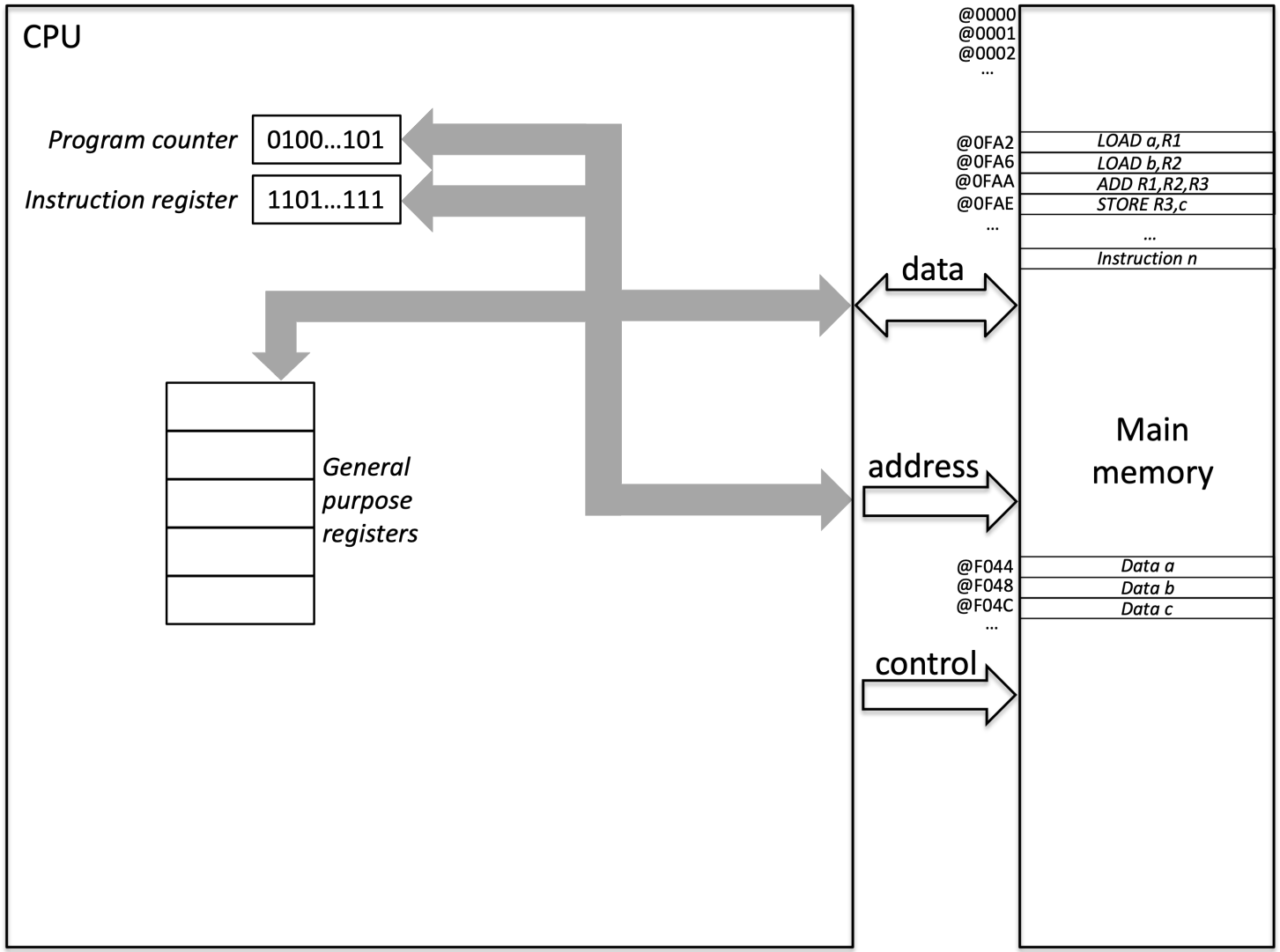


The memory then sends the current instruction binary code... The CPU stores it in its **instruction register**

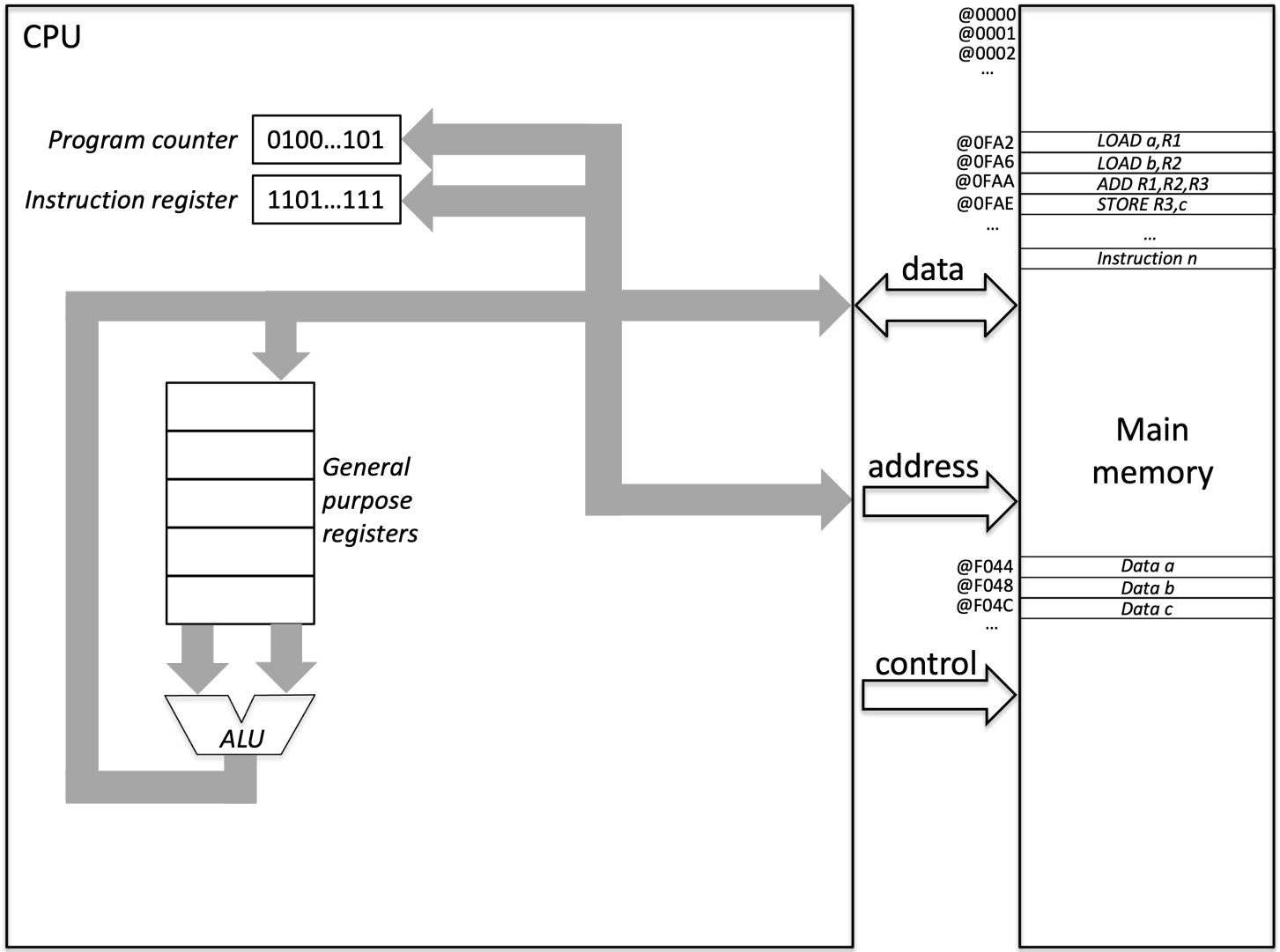


End of the FETCH cycle...
The CPU now knows what it has to do...
Let’s now EXECUTE the instruction

MEMORY instructions: the CPU reads or writes some data in the memory. This data is stored in a **General Purpose Register**



ARITHMETIC and LOGIC instructions: the content of two general purposed registers is sent to the **ALU**; the result is stored in a general purpose register

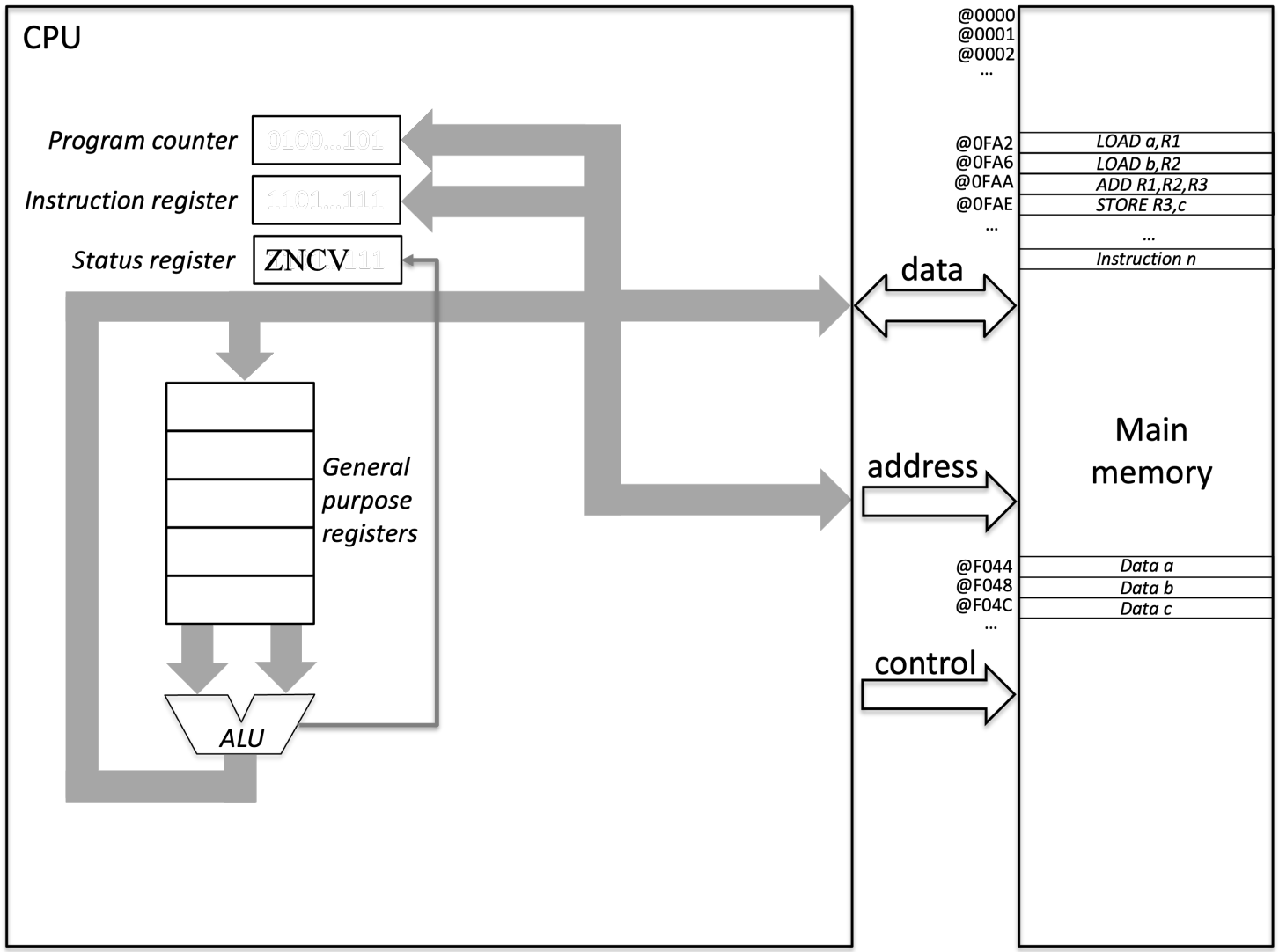


The **Arithmetic and Logic Unit (ALU)**

- The **ALU** (Arithmetic and Logic Unit) is the part of the CPU responsible for performing **arithmetic and logical operations**.
- The ALU takes **operands** as input and produces a **result** as output.
- An **opcode** specifies which operation the ALU must perform.
- Typical operations include:
  - arithmetic: `ADD`, `SUB`, `SHL`, `SHR` ...
  - logic: `AND`, `OR`, `XOR`, ...
  - comparisons: $=$, $<$, $>$, ...
- In most architectures the ALU also outputs binary **Flags** (`Z`, `N`, `C`,`V`) that are stored in the `Status Register`.

The ALU is the part of the CPU that “does the computation”.

The ZNCV flags and the status register



A decisive question: where is the Next Instruction?

- For a memory, arithmetic, or logic instruction, the next instruction normally immediately follows the current one in memory:

$$\text{address}_{\text{next}} = \text{address}_{\text{current}} + \text{length}_{\text{current}}.$$

- The **ALU** can therefore be used to compute the address of the next instruction (which will be written in the **Program Counter**).
- If this were always the case, the sequence of executed instructions would always be the same: no tests, no loops, no conditional execution.
- To implement **tests, loops, and other changes in the execution sequence**, the instruction set contains **control-flow instructions**.
- These instructions can modify the address of the next instruction to be executed.

Control Flow Instructions

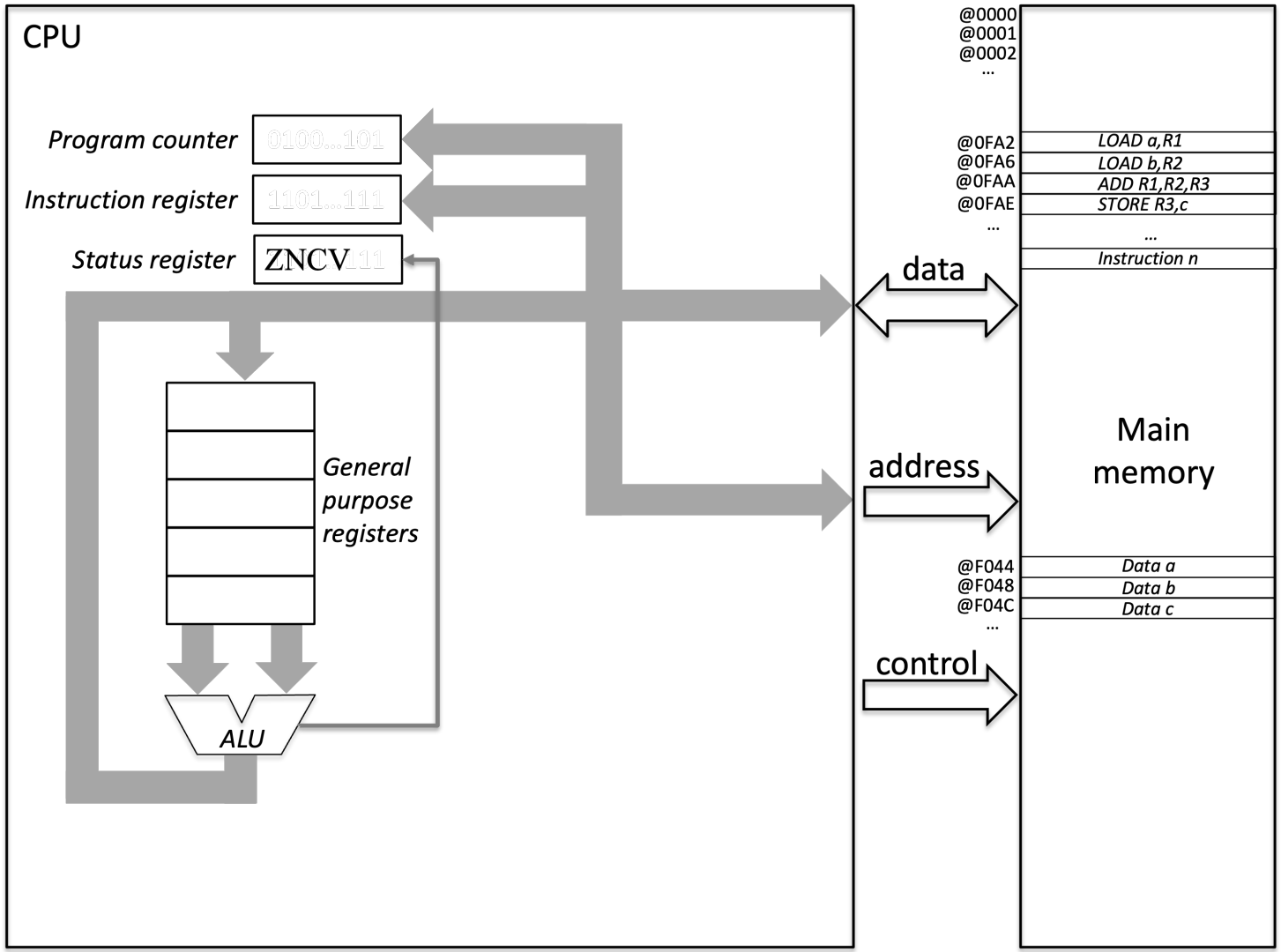
- **Control-flow instructions** change the address of the next instruction to be executed.
  - **unconditional jump** (`JUMP address`): transfer execution to a specified target address, which may be encoded as an absolute address or a relative offset.
  - **conditional branch** (`JUMP-IF condition, address`) changes the control flow only if a condition is satisfied:
    - For example:

```
CMP   R1,R2    # compare R1 and R2
JGE   -16      # jump 16 bytes backward if R1 ≥ R2
```

    - Conditional branches generally use **relative addresses**
- These instructions allow us to implement:
  - **tests** (`if`, `else`, `...`)
  - **loops** (`while`, `for`, `...`)

Key concept: Control-flow instructions determine which instruction is executed next. Otherwise, execution proceeds sequentially, with the next instruction being the one immediately following the current instruction in memory.

All these registers + ALU + Buses constitutes the **datapath** of the CPU



von Neumann architecture – The datapath

In a von Neumann architecture, the DataPath contains some FUNDAMENTAL elements:

- The **Program Counter** (`PC`) stores the address of the current/next instruction
- The **Instruction Register** (`IR`) stores the binary vector of the (opcode of the) instruction that is being executed
- **General Purpose Registers** that temporarily store numerical data in the processor
- The **Arithmetic and Logic Unit**, a combinatorial circuit that is able to perform various computations (Add, Sub, SHL...) on one or two operands. It has two outputs:
  - The result of the computation
  - A series of **Flags** that indicates whether the result is Zero (`Z`) or Negative (`N`) and if the computation has produced a Carry (`C`) or an oVerflow (`V`)
  - These flags are stored in a specific register (`SR` – Status Register) and used by **conditional jump** instructions.

The datapath: a musical analogy



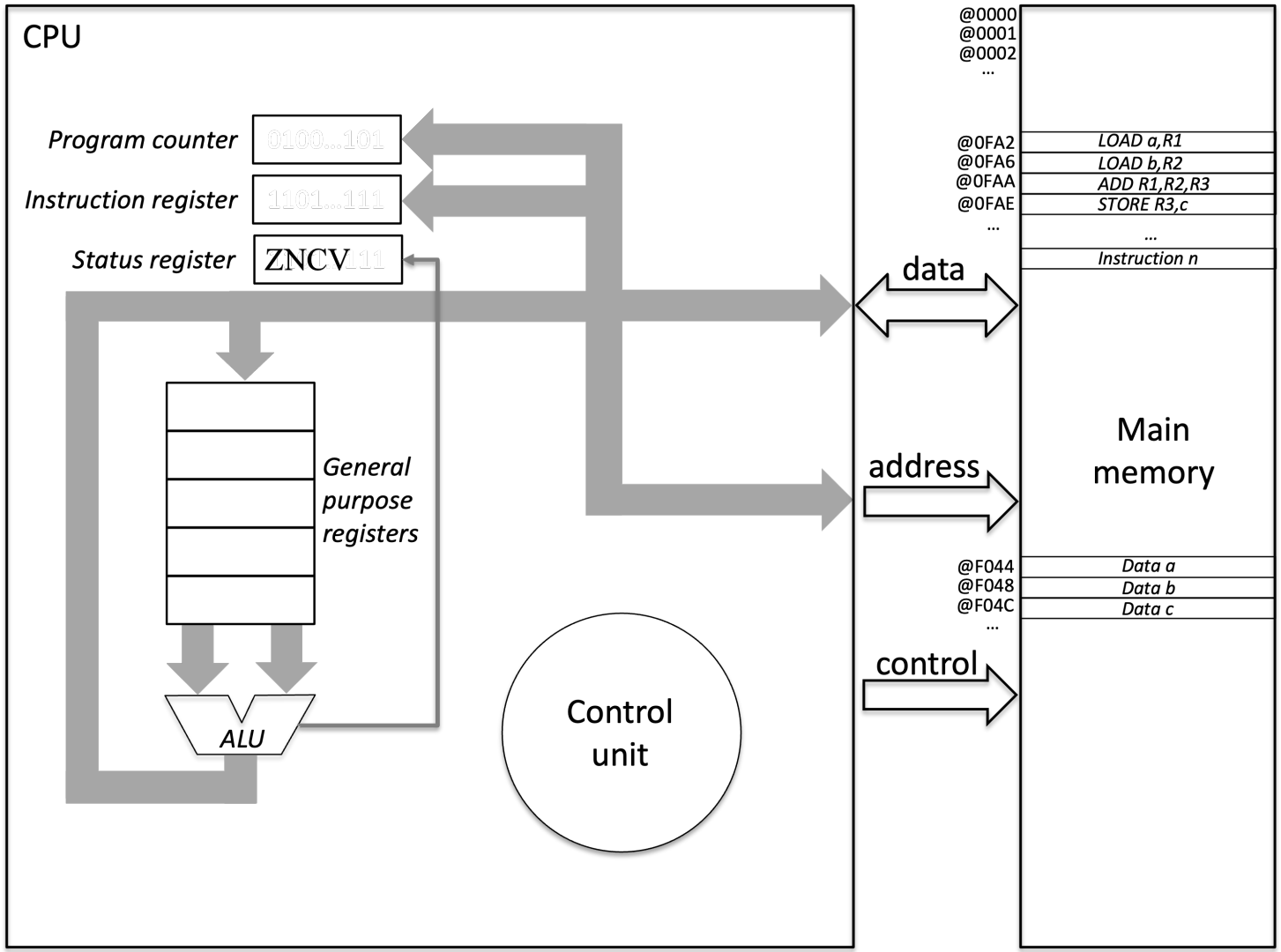
The datapath is composed of many specialized circuits. It is able to perform all the operations required to execute any instruction in the instruction set, but it does not know what to do or when to do it.

But who is the conductor?

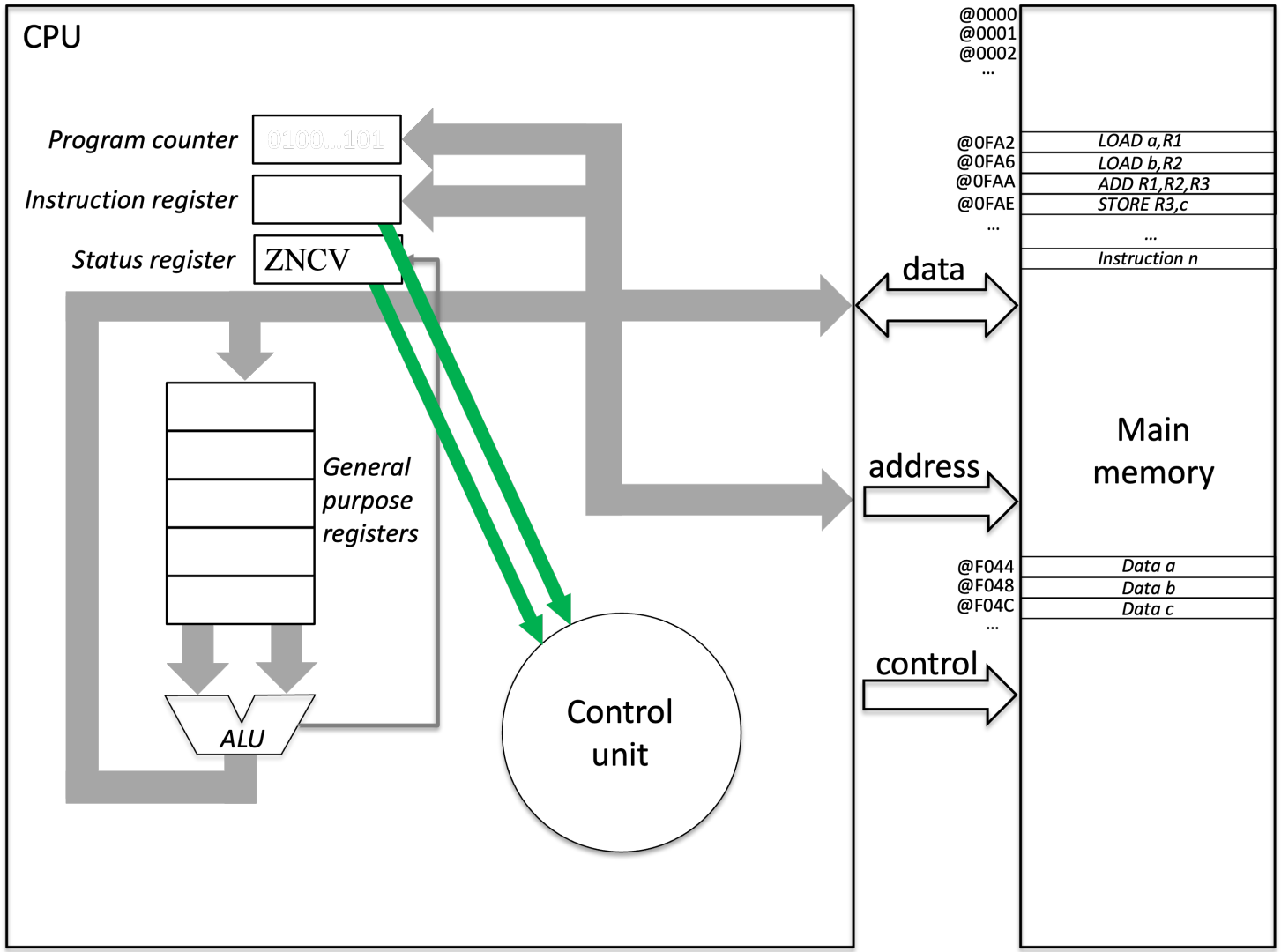


The datapath needs a **conductor** to tell its different circuits what to do and when to do it.

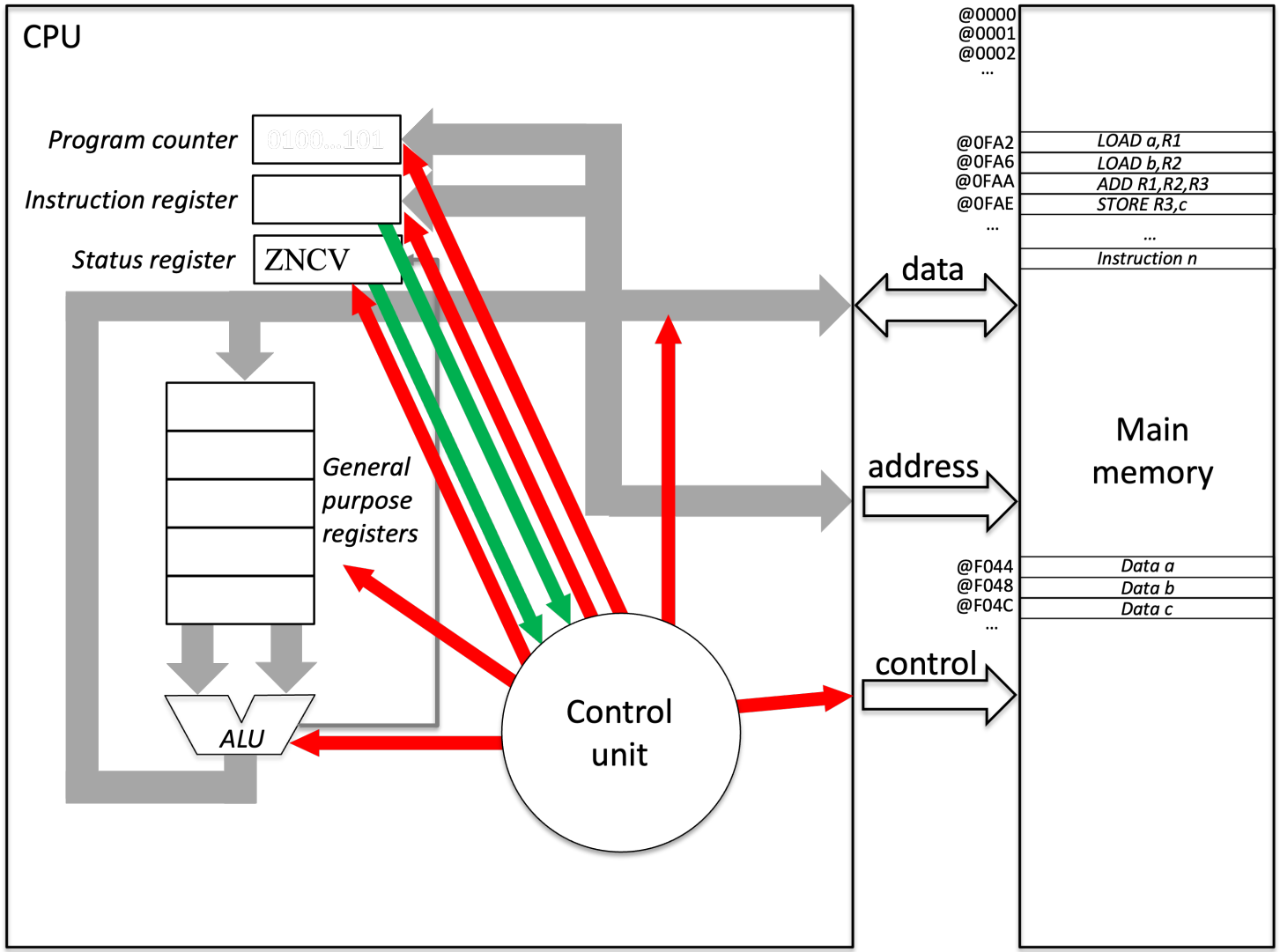
The **Control Unit** is the conductor that decides who does what in the CPU



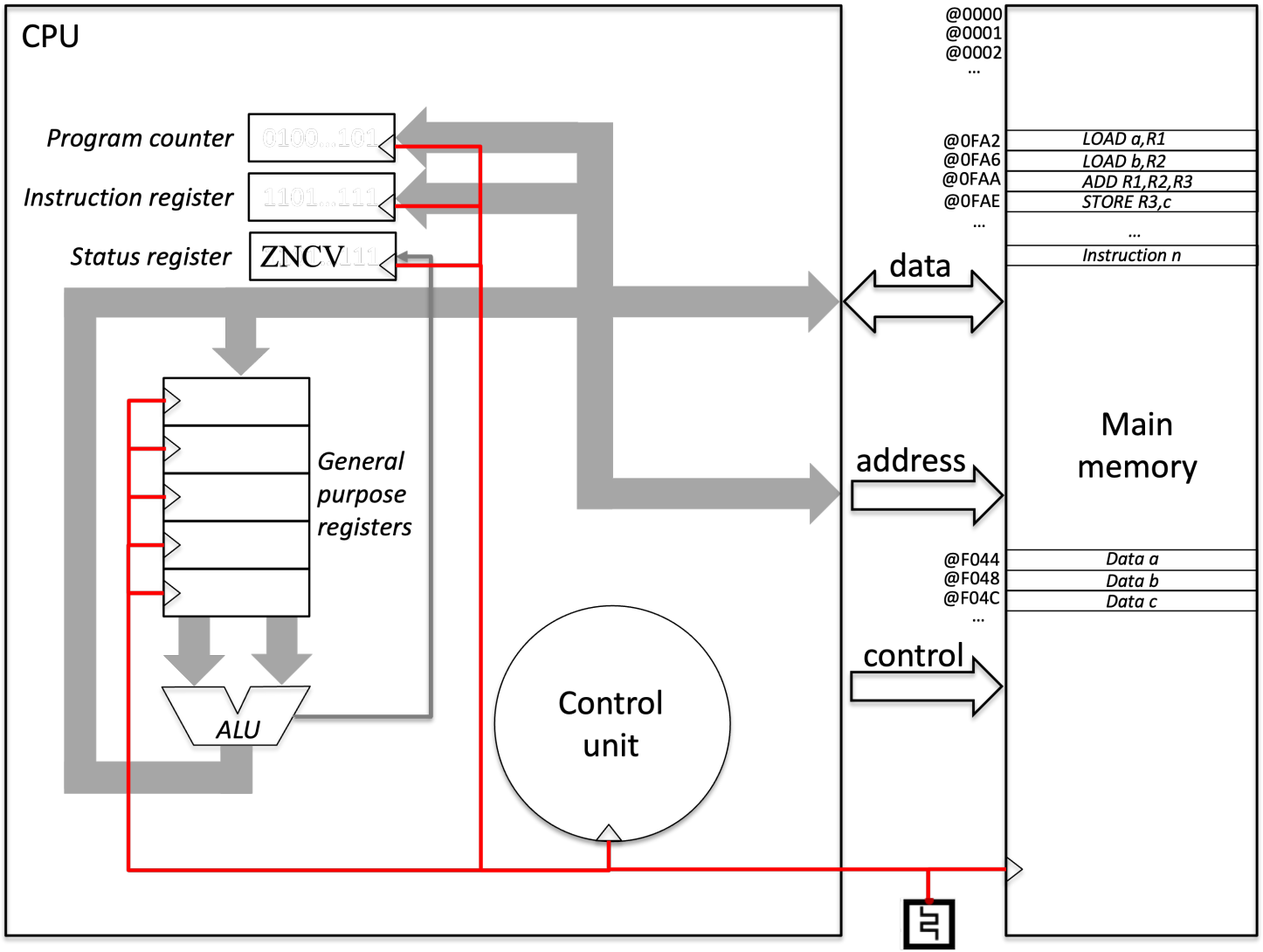
The **Control Unit** receives the binary code of the instruction and the content of the status register



At each time step the **Control Unit** sends signal to control data exchanges (including with the memory) and register write operations



Remember that this machine is synchronous: its sequential components (control unit, registers, and memory) are synchronized by a common clock.



Executing programs – The Von Neumann Cycle

Given the organization of a von Neumann's machine, the algorithm executed by its Control Unit is (astonishingly) simple:

```
Do forever:
   Fetch Instruction
   Decode Instruction
   Execute Instruction
```

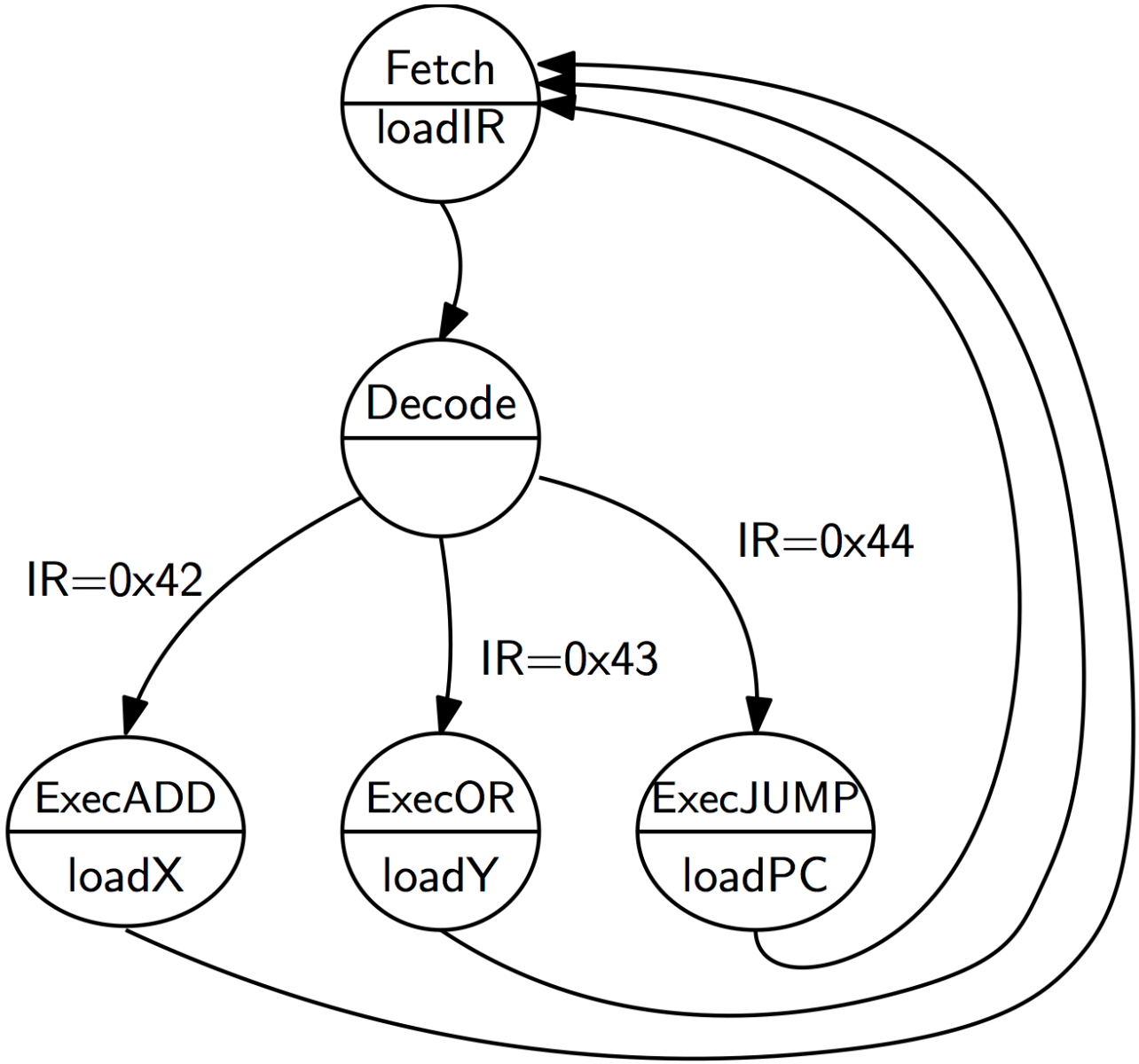
A von Neumann's machine an ASM that executes this algorithm (the "von Neumann cycle") such that:

Fetch Copy the current instruction bit-vector from the memory to the processor and compute the address of the next one

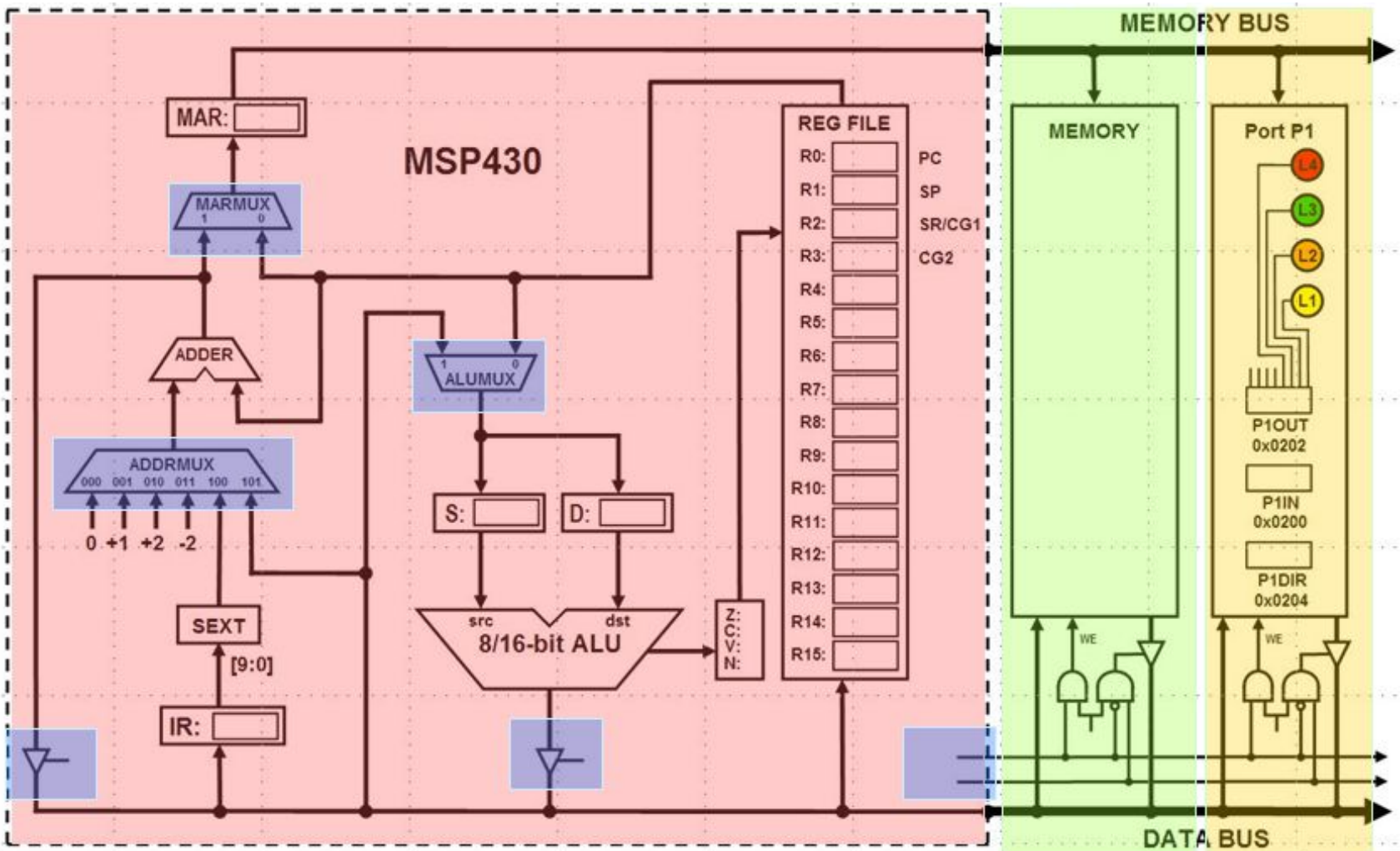
Decode Look at the instruction opcode to prepare the DataPath

Execute Process the data in the DataPath such that the instruction does what is has to do (possibly in several time steps)

von Neumann architecture – the control automaton



Von Neumann Architecture – the architecture of a real (but simple) computer: the MSP430



How is this code executed on that machine?

```
0x0000000100000f70 <+0>:     push   %rbp
0x0000000100000f71 <+1>:     mov    %rsp,%rbp
0x0000000100000f74 <+4>:     movl   $0x0,-0x4(%rbp)
0x0000000100000f7b <+11>:    movl   $0x0,-0x8(%rbp)
0x0000000100000f82 <+18>:    movl   $0x0,-0xc(%rbp)
0x0000000100000f89 <+25>:    movl   $0x1,-0xc(%rbp)
0x0000000100000f90 <+32>:    cmpl   $0x64,-0xc(%rbp)
0x0000000100000f94 <+36>:    jge    0x100000fb1 <main()+65>
0x0000000100000f9a <+42>:    mov    -0x8(%rbp),%eax
0x0000000100000f9d <+45>:    add    $0x1,%eax
0x0000000100000fa0 <+48>:    mov    %eax,-0x8(%rbp)
0x0000000100000fa3 <+51>:    mov    -0xc(%rbp),%eax
0x0000000100000fa6 <+54>:    add    $0x1,%eax
0x0000000100000fa9 <+57>:    mov    %eax,-0xc(%rbp)
0x0000000100000fac <+60>:    jmpq   0x100000f90 <main()+32>
0x0000000100000fb1 <+65>:    mov    -0x8(%rbp),%eax
0x0000000100000fb4 <+68>:    pop    %rbp
```



```
0000f70 55 48 89 e5 c7 45 fc 00 00 00 00 c7 45 f8 00 00
0000f80 00 00 c7 45 f4 00 00 00 00 c7 45 f4 01 00 00 00
0000f90 83 7d f4 64 0f 8d 17 00 00 00 8b 45 f8 83 c0 01
0000fa0 89 45 f8 8b 45 f4 83 c0 01 89 45 f4 e9 df ff ff
0000fb0 ff 8b 45 f8 5d c3 90 90 01 00 00 00 1c 00 00 00
0000fc0 00 00 00 00 1c 00 00 00 00 00 00 00 1c 00 00 00
0000fd0 02 00 00 00 70 0f 00 00 34 00 00 00 34 00 00 00
```

Demo Time!

That all folks!

In this course, we followed a bottom-up approach:

- ✓ How information is coded $\rightarrow$ **binary**
- ✓ How we can deal with this information to compute other information from it $\rightarrow$ **boolean algebra**
- ✓ How to build **combinatorial circuits** implementing simple mathematical functions
- ✓ How to deal with time and describe **sequential behaviors**
- ✓ How to build a small **programmable machine**

$\Rightarrow$ The “Computer Architecture” course will further this discussion towards “real” computers.