# From Sequential Circuits to “Real” Computers

Lecturer: Guillaume Beslon
(Lecture adapted from Lionel Morel)

Computer Science and Information Technologies - INSA Lyon

Fall 2025

# 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

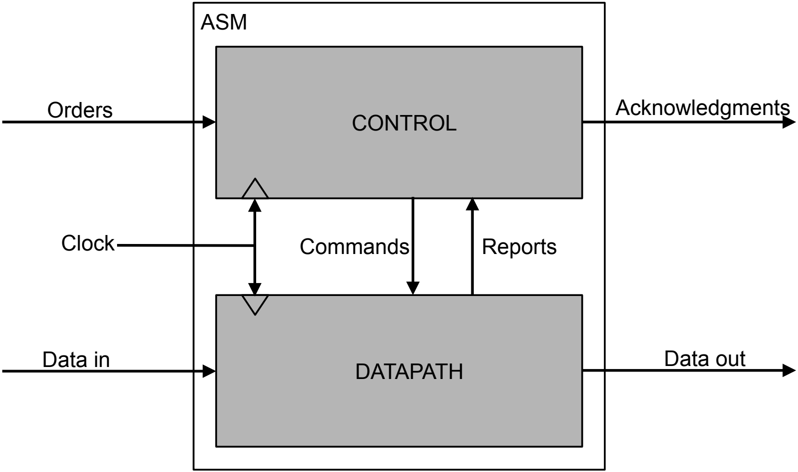
# From FSM to ASM

- 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

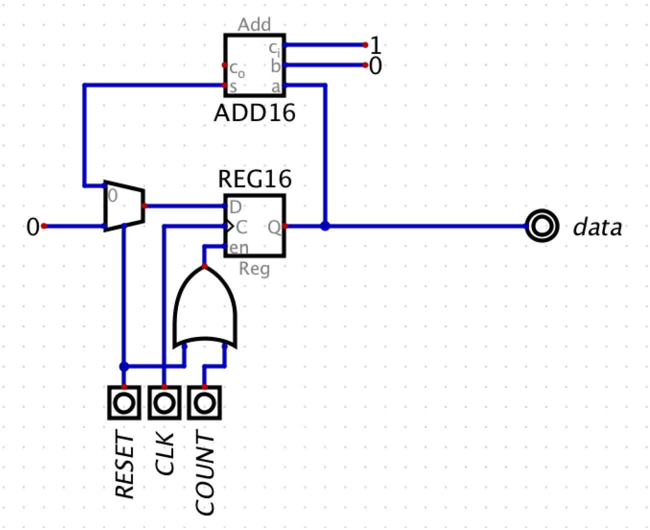
# A Sequential Circuit within the Control-Data Separation Scheme



# 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)

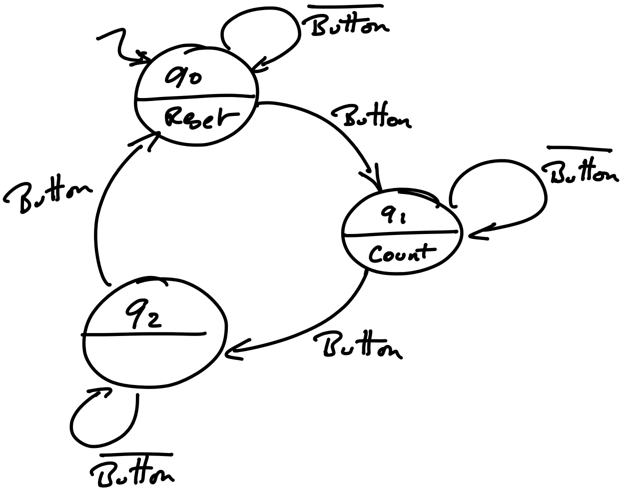
  Input alphabet: **Orders** (from the outside of the circuit) and **Reports** (from the datapath)

  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

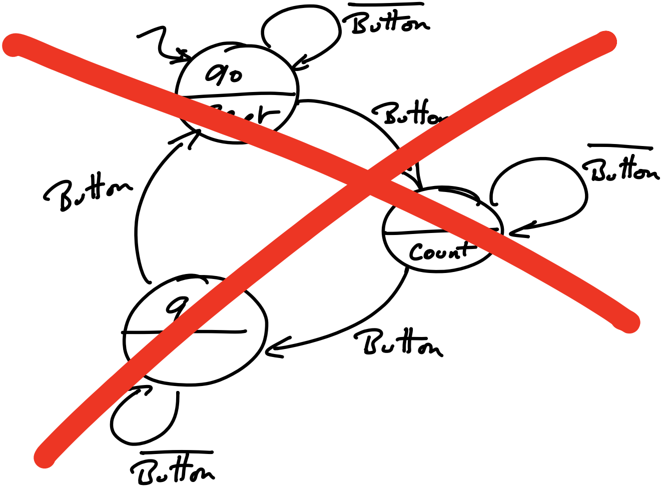
$I = \{BUTTON\}$

$O = \{COUNT, RESET\}$



# Control of the stopwatch

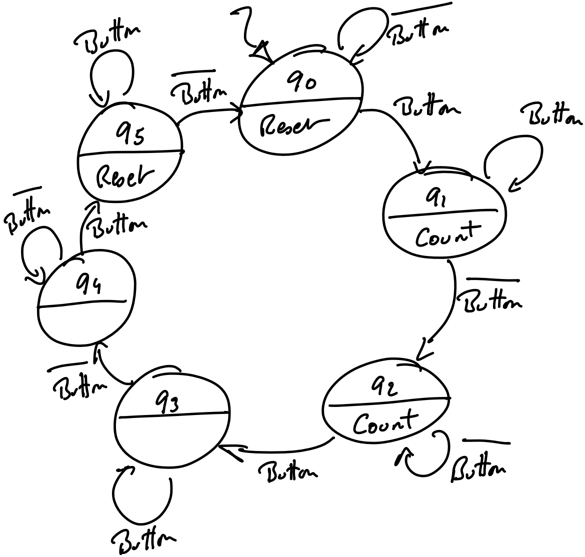
$I = \{BUTTON\}$
$O = \{COUNT, RESET\}$



# 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)

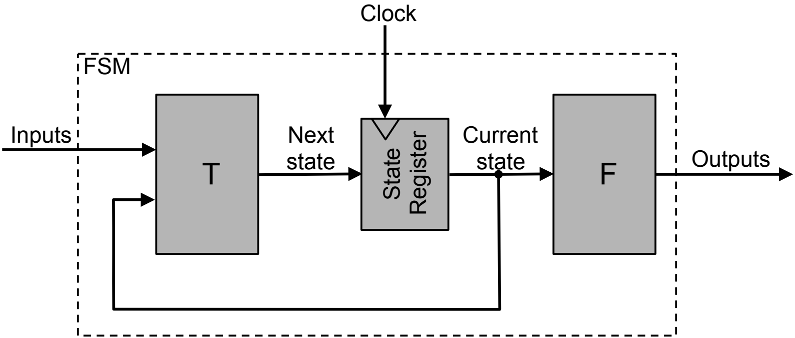
## VERY IMPORTANT

- **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)

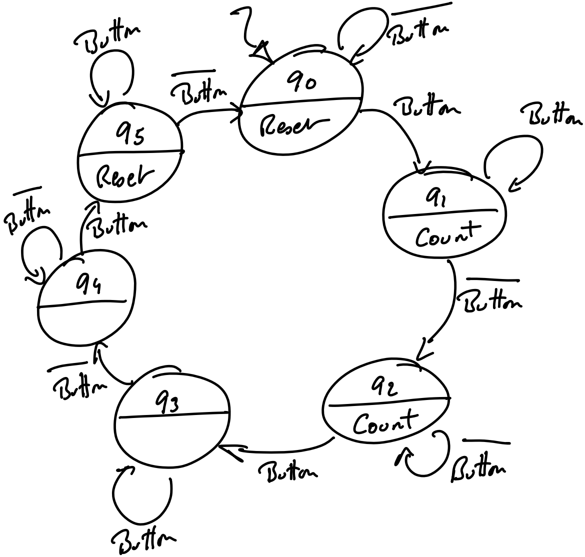
- $I$ = Orders ∪ Reports
- $O$ = Acknowledgements ∪ Commands
- $Q$ = set of states
- $T = Q \times$ (Orders ∪ Reports) $\rightarrow Q$ (transition function)
- $F = Q \rightarrow$ (Acknowledgements ∪ Commands) (output function)



# 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 |

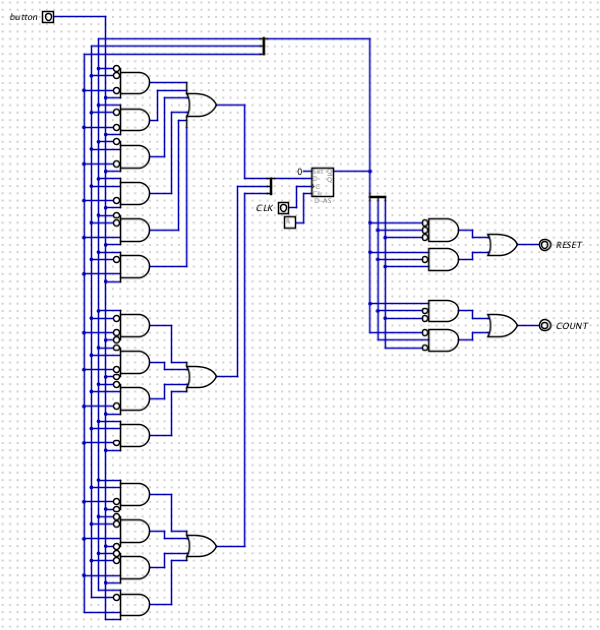
# Output function for the stopwatch control

| 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 |

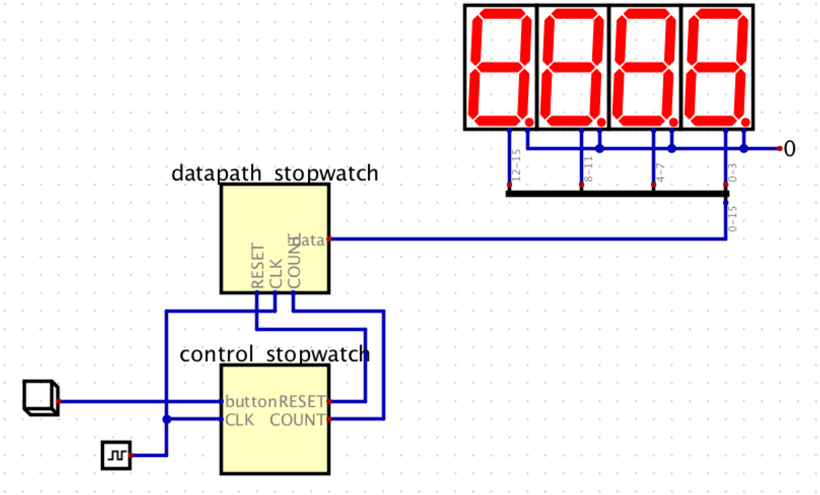
$\Rightarrow$

| $s_2$ | $s_1$ | $s_0$ | 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: a telemeter

Let's build a telemeter with digital display.

Usage:

- User presses a button
- Telemeter emits an ultrasound impulse
- Measures the echo travel time
- Travel time is translated into a distance
- Distance is displayed on screen

# Telemeter: definition of input/output signals

Inputs are:

- **GO**: triggers a new measure. Telemeter waits for GO to be `1` to start a new measure.
- **Receive**: `0` when the ultrasound sensor hears “nothing”, `1` when sensor hears an echo.

Outputs are:

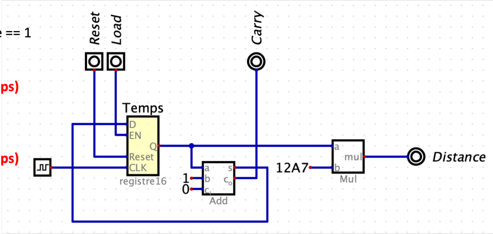
- **Emit**: Needs to be set to `1` during one clock cycle to emit an ultrasound impulse.
- **Distance**: unsigned, 16 bits precision (but maximum value can be different from 65,535) due to time→distance convertion); `0` until **Receive**.
- **OK**: `0` whenever the telemeter counts, `1` as soon as Distance is valid. Stays `1` until we ask for a new measure
- **ERR**: `1` if echo “never” comes back, `0` otherwise.

# Telemeter: Algorithm

```
tant_que Go==0
fin_tant_que
tant_que 1
        temps = 0
        emit
        tant_que receive==0 and carry==0
                (temps, carry) = temps + 1
        fin_tant_que
        si carry == 1
                tant_que Go == 1
                    error
                fin_tant_que
                tant_que Go == 0
                    error
                fin_tant_que
        fin_si
        si carry == 0 and receive == 1
                tant_que Go == 1
                    OK
                    distance = f(temps)
                fin_tant_que
                tant_que Go == 0
                    OK
                    distance = f(temps)
                fin_tant_que
        fin_si
fin_tant_que
```

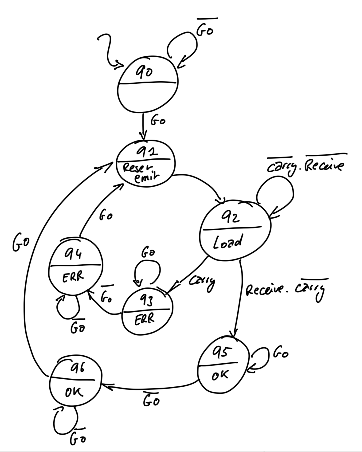
# Telemeter: from algorithm to datapath

```
tant_que Go==0
fin_tant_que
tant_que 1
      temps = 0
      emit
      tant_que receive==0 and carry==0
            (temps, carry) = temps + 1
      fin_tant_que
      si carry == 1
            tant_que Go == 1
               error
            fin_tant_que
            tant_que Go == 0
               error
            fin_tant_que
      fin_si
      si carry == 0 and receive == 1
            tant_que Go == 1
               OK
               distance = f(temps)
            fin_tant_que
            tant_que Go == 0
               OK
               distance = f(temps)
            fin_tant_que
      fin_si
fin_tant_que
```



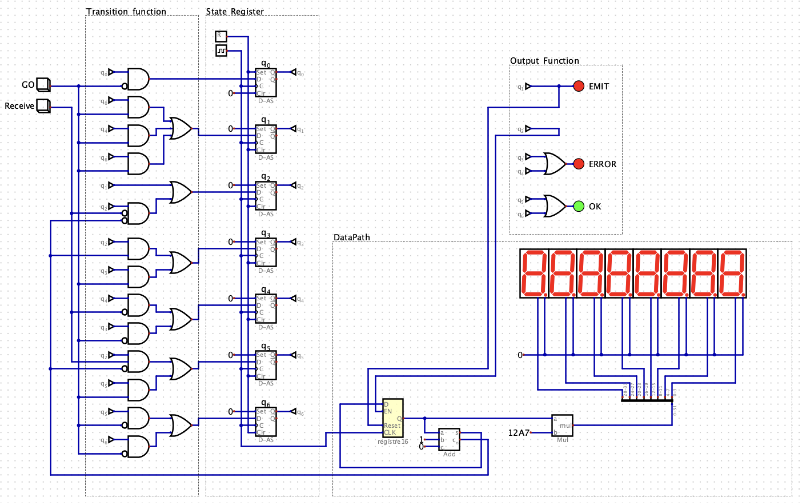
# Telemeter: from algorithm to control

```
tant_que Go==0                                     } q0
fin_tant_que
tant_que 1
    temps = 0                                      } q1
    emit
    tant_que receive==0 and carry==0               } q2
        (temps, carry) = temps + 1
    fin_tant_que
    si carry == 1
        tant_que Go == 1                           } q3
            error
        fin_tant_que
        tant_que Go == 0                           } q4
            error
        fin_tant_que
    fin_si
    si carry == 0 and receive == 1
        tant_que Go == 1                           } q5
            OK
            distance = f(temps)
        fin_tant_que
        tant_que Go == 0                           } q6
            OK
            distance = f(temps)
        fin_tant_que
    fin_si
fin_tant_que
```



**Demo Time!**

# Telemeter: final circuit

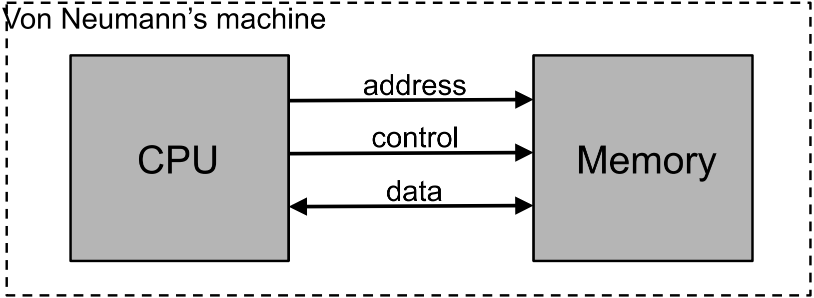


# Final sprint: building a real (but simple) computer

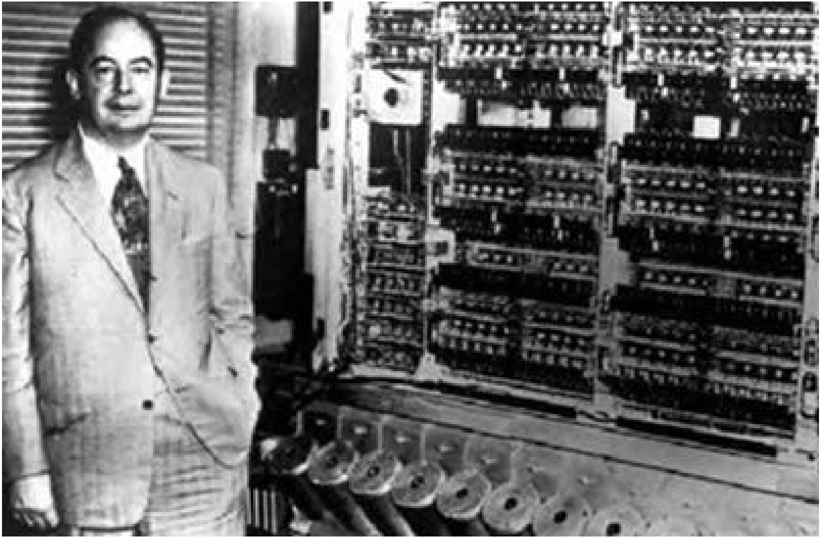
- Great, we have all the necessary elements to build a “Real” computer
- The only thing we still need is a way to **organize** things in order to execute **any program** rather than always the same computation...
- But executing a program can be “simply” viewed as a computation (i.e. always the same !)
  - read an instruction
  - execute it
  - go to the next one
- **We will use Control-Data separation to build a sequential circuit which function will be to compute the execution of a sequence of instructions**

$\Rightarrow$ **von Neumann architecture**

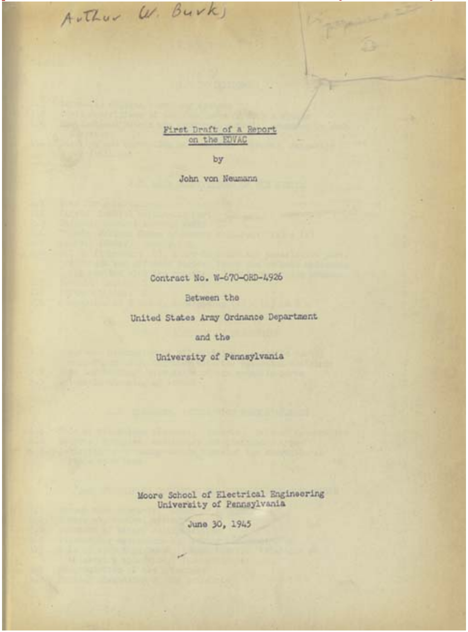
# Von Neumann's computer



# 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 machine executes **instructions**
- A program is a list of instructions ordered **sequentially**
  - This sequence is the **control flow**
- All possible instructions form the **instruction set**
- There are **three main types of instructions**
  - Data management (load, move...)
  - Arithmetic and Logic (Add, Mul, Not, SHL, Sub...)
  - Flow control (Jump, JGE, JLE...)
- 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 indicates both the instruction and its addressing mode

# More on addressing modes

- Typical computers have many **addressing modes**
- Different addressing modes used for different purposes

| | |
|---|---|
| Immediate | Operand is a constant directly specified in the instruction |
| Direct | Instruction specifies the location (memory address or register) of the operand |
| Indirect | Instruction specifies a location (memory address or register) that contains the address of the operand |
| Indexed | Effective address = base address (stored in a register) + displacement (constant) |
| Implicit | Operand is implied by the instruction itself |
| ... | |

- The opcode specifies both the instruction and the addressing mode.
- In assembly code, addressing modes are indicated by writing conventions (ex: `$0x64, %eax, -0x8(%rbp), ...`)

# Basic concepts in von Neumann's architecture

```
int main()
{
  int x,i;
  x = 0;
  i = 0;
  for (i = 1; i<100;i++)
    {
      x = x+1;
    }
  return x;
```

# Basic concepts in von Neumann's architecture

```c
int main()
{
  int x,i;
  x = 0;
  i = 0;
  for (i = 1; i<100;i++)
    {
      x = x+1;
    }
  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
```

# Executing programs – The Von Neumann Cycle

Given the structure of a program in a von Neumann's machine, the algorithm to execute it is (astonishingly) simple:

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

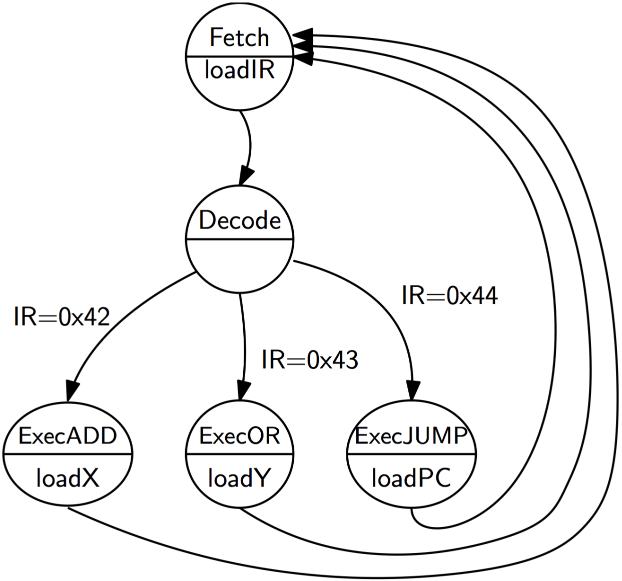
A von Neumann's machine an ASM that executes this algorithm (the "von Neumann cycle") such that:

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

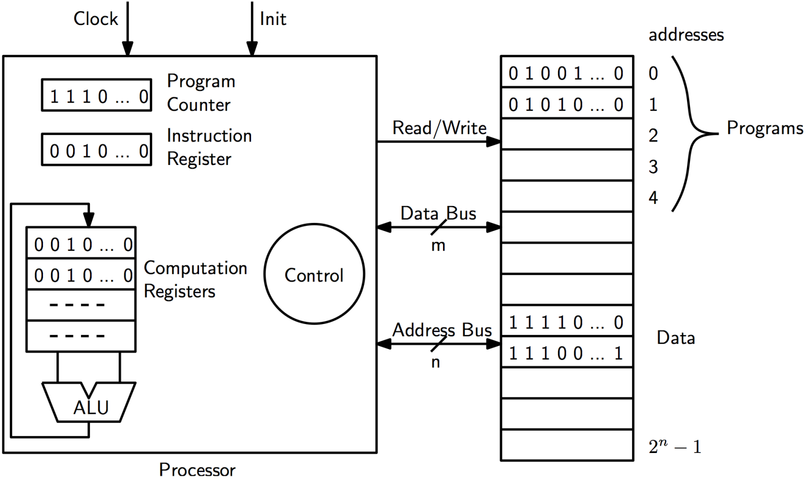
**Decode** Look at the instruction opcode to prepare the DataPath

**Execute** Process the data in the DataPath such that the instruction does what is has to do

# von Neumann architecture – the control automaton



# von Neumann architecture – The datapath

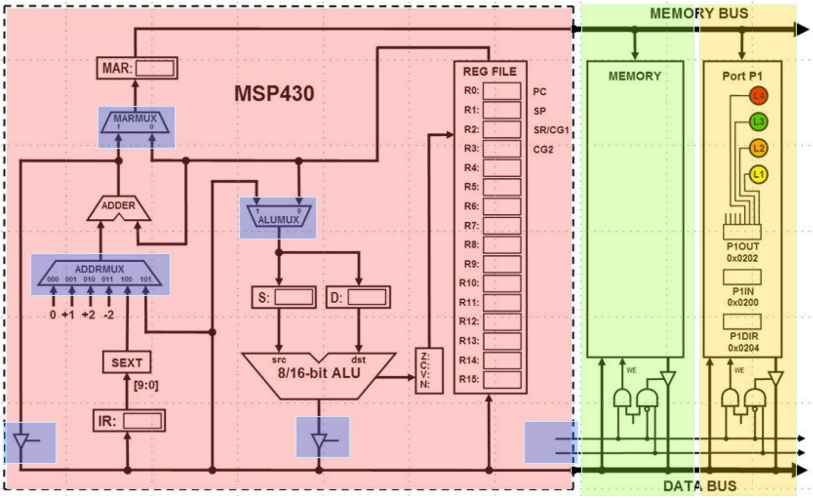


# von Neumann architecture – The datapath

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
- **Registers** 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.

# Von Neumann Architecture – the datapath of a real computer



# How is this code executed?

```
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.