From Sequential Circuits to “Real” Computers

Lecturer: Guillaume Beslon
Original Author: Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall 2019
What we have done so far is implementing “simple” FSM by using Moore Machines.

No (complex) data is manipulated (e.g., integers) because this will require too many states...

But digital circuits (and of course computers) deal with data.

We need a methodology that will gather:
- combinatorial functions,
- automata,
- registers,

to build functions such as counters, dividers, speedometers... and general computers.

⇒ **Algorithmic State Machines**
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
- Such machines typically have a large number of states to deal with numerical values
- ASM 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
From FSM to ASM: a simple example, the stopwatch

Blackboard!
A Sequential Circuit within the Control-Data Separation Scheme
Datapath

- Offers **computational resources** 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)
A typical (simplistic) Datapath
Control

- Knows **which** operations to perform and **when**
- Doesn’t deal with data directly (doesn’t know **how** to do the operations)
- Typically implemented as a **Finite State Machine**, i.e., an **automaton** (see lectures 4 and 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

Control is designed as a classical FSM (remember lectures 4 and 5)
Control (cont’d)

Then control is implemented as a classical FSM (remember lectures 4 and 5)

- $I = \text{Orders} \cup \text{Reports}$
- $O = \text{Acknowledgements} \cup \text{Commands}$
- $Q = \text{set of states}$
- $T = Q \times (\text{Orders} \cup \text{Reports}) \rightarrow Q$ (transition function)
- $F = Q \rightarrow (\text{Acknowledgements} \cup \text{Commands})$ (output function)
Control (cont’d)

Control is just about implementing 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 **multiplexers**
- Control “never” has access to the data. It only receives **Reports computed by the datapath**. Reports are used to choose automaton transitions.
Conception Methodology

1. Start with:
   - The algorithm describing the expected behavior
   - The general scheme of a sequential circuit

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.
Running 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
Running Example: 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 conversion); 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.
Running Example: Algorithm

tant_que GO = 0
fin_tant_que

tant_que 1
   tant_que GO = 1
   temps = 0
   Emit
fin_tant_que

tant_que Receive = 0 and carry = 0
   (temps,carry) = temps + 1
   distance = f(temps)
fin_tant_que

si carry
   tant_que GO = 0
   ERR
   fin_tant_que
sinon
   tant_que GO = 0
   OK
   fin_tant_que
fin_si
fin_tant_que
Running Example: Circuit interface
Running Example: Global architecture

[Diagram showing a block diagram with nodes labeled 'Control', 'Datapath', 'Distance', 'Go', 'Receive', 'OK', 'ERR', 'Emit', and 'OK']
Running Example: counting distance
Running Example: bonus, dealing with time and overflow

Blackboard!
Running Example: moore machine
Demo Time!
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.

⇒ von Neumann architecture
Von Neumann’s computer
Von Neumann and the EDVAC
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 :)


At any rate a *central arithmetical part* of the device *will probably have to exist*, and this constitutes the first specific part: CA.

⇒ Datapath and ALU
If the device is to be **elastic**, that is as nearly as **possible all purpose**, then a distinction must be made between the specific instructions given for and defining a particular problem, and the general control **organs** which see to it that these instructions—no matter what they are—are carried out.

By the **central control** we mean this latter function only, and the **organs which perform it** form the second specific part: **CC**.
At any rate the total memory constitutes the third specific part of the device: $M$.

⇒ Memory
The three specific parts CA, CC (together C) and M correspond to the associative neurons in the human nervous system.

Von Neumann’s Architecture

https://en.wikipedia.org/wiki/Von_Neumann_architecture
What is a program?

High-level language program (in C)

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

Assembly language program (for MIPS)

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

Binary machine language program (for MIPS)

```
00000000010100010000000100011000
00000000100000010000000100000100
10001101111001000000000000000000
10001100001001000000000000000000
10101110000110000000000000000000
10101101100010000000000000000000
0000001111110000000000000000000100
```
 Programs

.signed .init9
main:
    mov.b #2, &0x32
loop:
    mov.b #2, &0x31
    mov #20000, R4
wait1:
    dec R4
    jnz wait1
    mov.b #0, &0x31
    mov #20000, R4
wait2:
    dec R4
    jnz wait2
    jmp loop
Programs are executed instruction by instruction:

```
loop{
    Fetch Current Instruction
    Decode Instruction
    Execute (use datapath)
    Write Results (to registers / memory)
    Go to Next Instruction
}
```

A computer an ASM (i.e. by a controller and a datapath) that executes this algorithm (the “von Neumann cycle”)
von Neumann architecture – The datapath

The diagram illustrates the key components of a von Neumann architecture, focusing on the datapath. The components include:

- Clock
- Program Counter
- Instruction Register
- Computation Registers
- ALU
- Control
- Data Bus
- Address Bus
- Read/Write

The diagram shows the flow of signals and data through these components, emphasizing the interaction between the program counter, instruction register, computation registers, and the data bus. The control logic manages the flow of data and instructions, ensuring the correct execution of operations.
von Neumann architecture – the control automaton
Von Neumann Architecture – the datapath of a real computer
Demo Time!
That all folks!

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

✓ How information is coded → binary
✓ How we can deal with this information to compute other information from it → 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**

⇒ The “Computer Architecture” course will further this discussion towards “real” computers.