# The Von Neumann Model Internals –Computer Organization–

Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall-Winter 2024-25

# The micro-machine ISA (1/3)

In the first lab sessions, we'll look at a home-made processor.

- 8-bits instructions
- 8-bits signed integers only
- 2 8-bits registers, named `A` and `B`

## Computation instructions with 1 or 2 operands

```
B -> A          21 -> B
B + A -> A      B xor -42 ->
                A
not B -> A      lsr A -> A
A xor 12 -> A   B - A -> A;
```

**WARNING**: some instructions that would seem "intuitive" are actually forbidden... eg: `A+B -> B` is incorrect ... `B+A -> B` is correct.

# The micro-machine ISA (2/3)

## Memory reads and writes

```
*A -> A        *A -> B
A -> *A        B -> *A
*cst -> A      *cst -> B
A -> *cst      B -> *cst
```

**`*A` means: “the content of memory at the address contained in register A”.**

# The micro-machine ISA (3/3)

## Unconditional absolute branch

```
JA 42
```

continues execution at address 42.

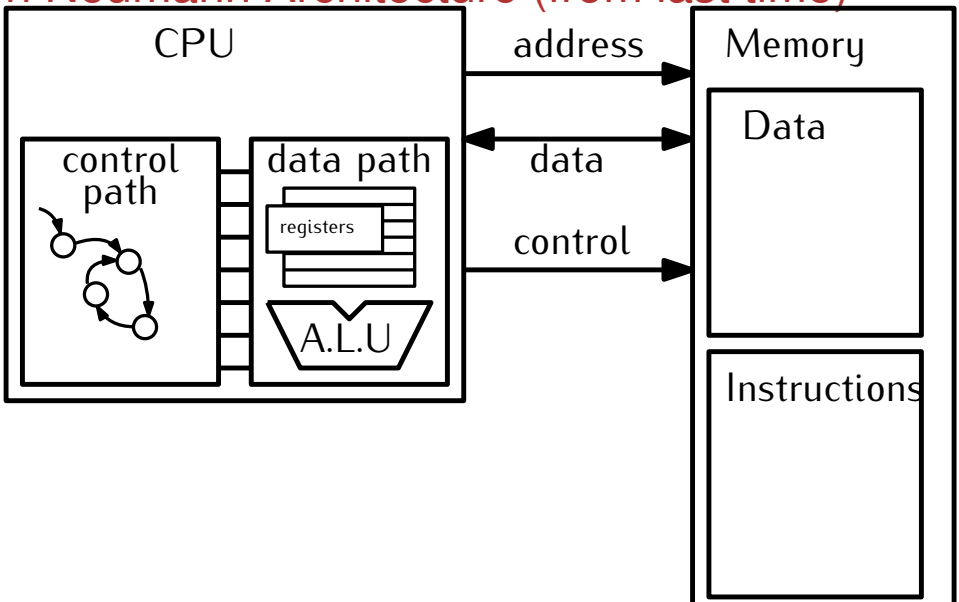
## Conditional relative branch

| | |
|---|---|
| `JR offset` | `JR offset IFZ` |
| | *(executed if Z=1)* |
| `JR offset IFC` | `JR offset IFN` |
| *(executed if C=1)* | *(executed if N=1)* |

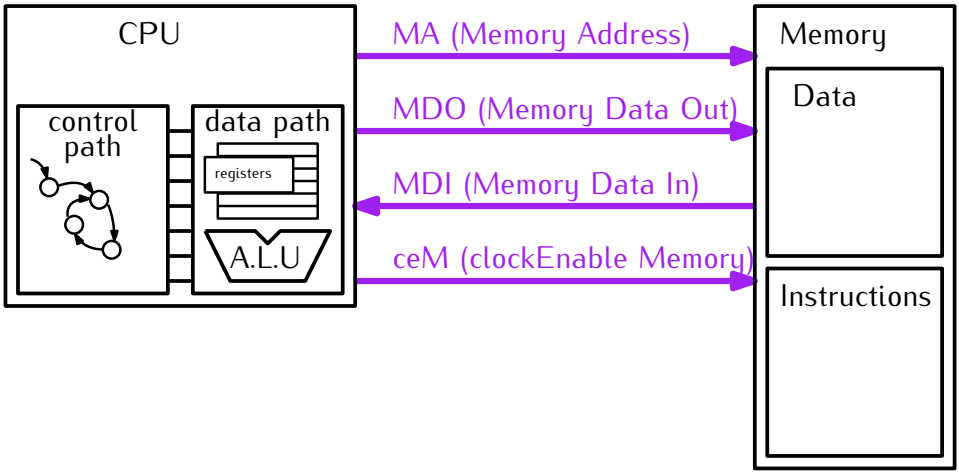
# The micro-machine ISA - example program

```
prog2: *21 -> A  ;; copy Mem[21] to register A
       *42 -> B  ;; copy Mem[42] to register B
       B+A -> B  ;; Compute B+A and put result
                 ;;    into register B
       B -> *43  ;; copy content of register B to Mem[43]
```

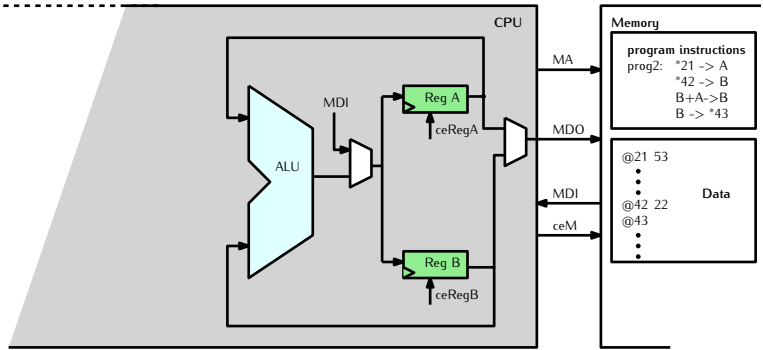
# Von Neumann Architecture (from last time)



# Von Neumann Architecture (refined)

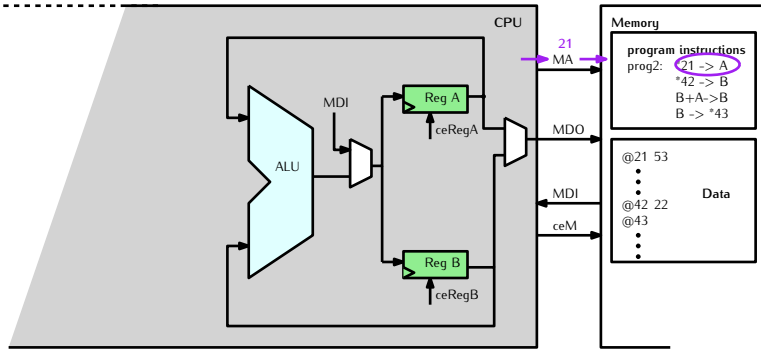


# 1- Execute Instructions



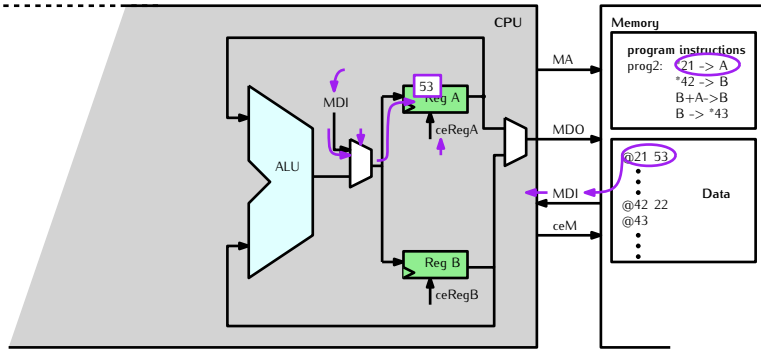

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



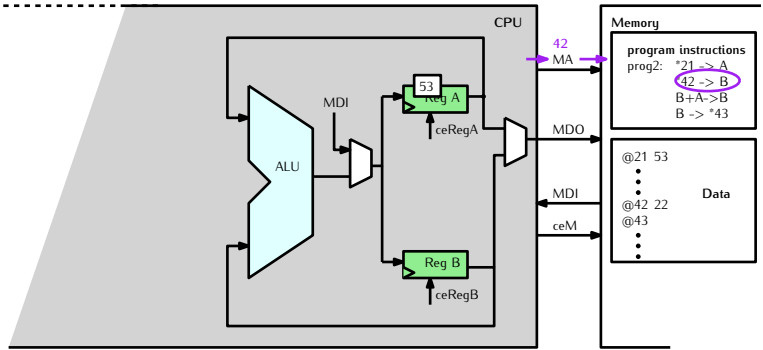

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



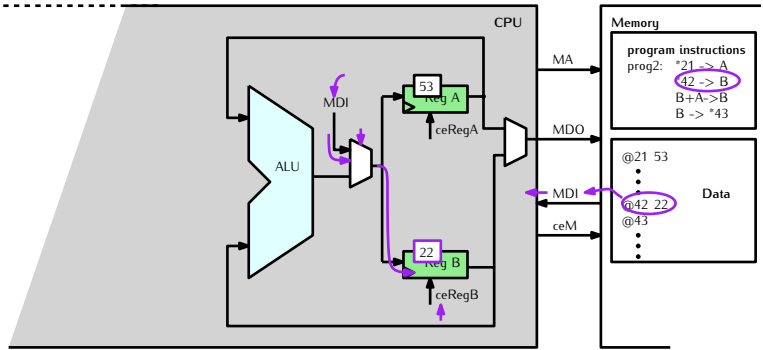

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



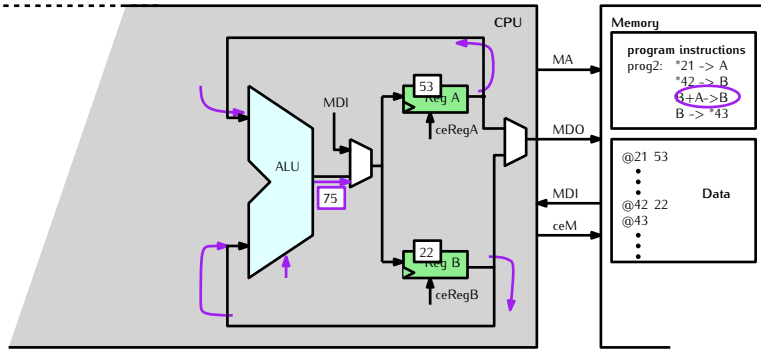

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



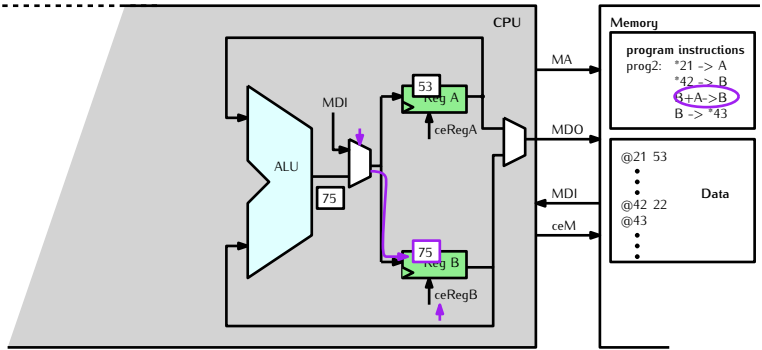

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



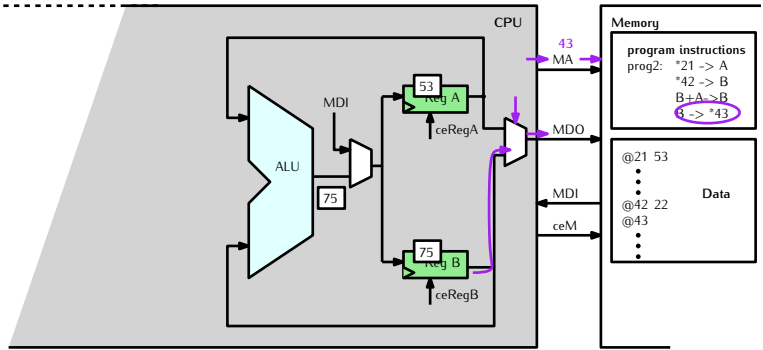

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



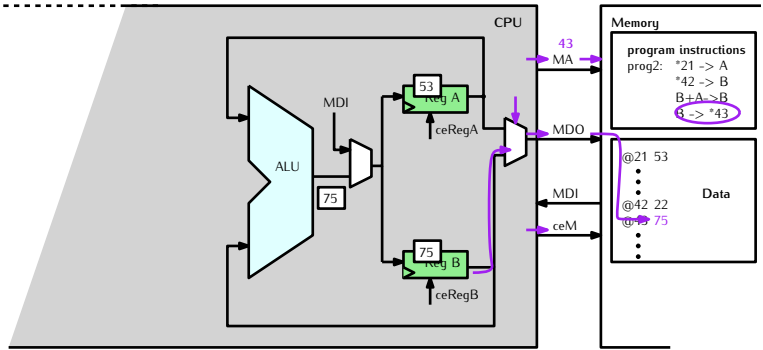

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



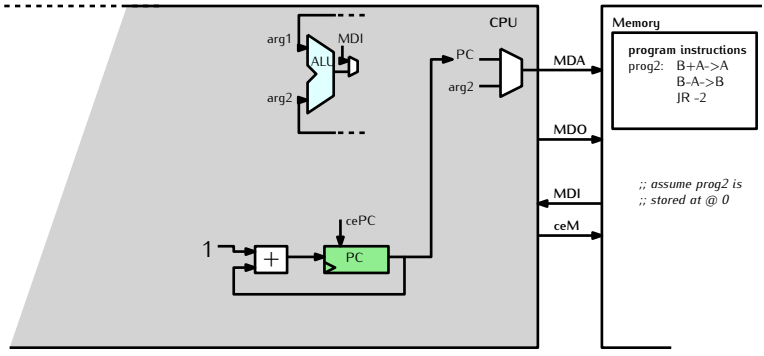

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 1- Execute Instructions



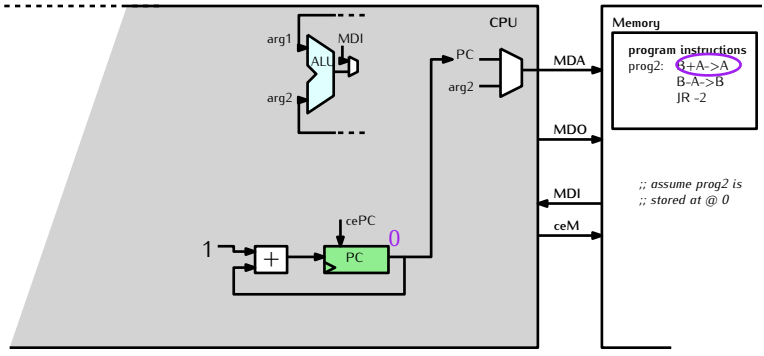

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 2- Advance in Program



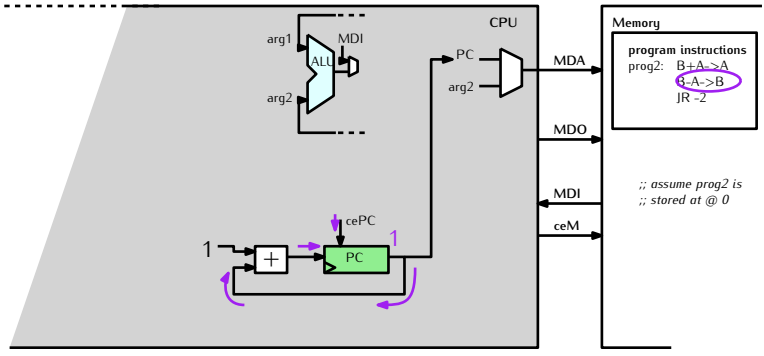

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 2- Advance in Program



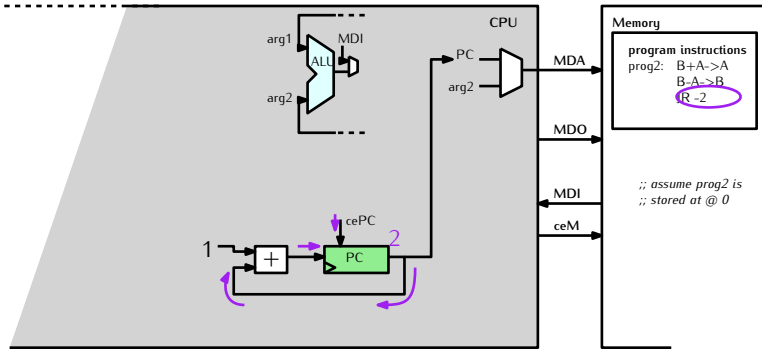

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 2- Advance in Program



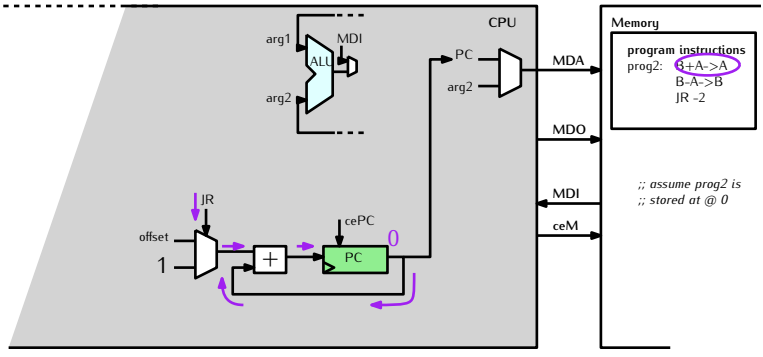

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 2- Advance in Program



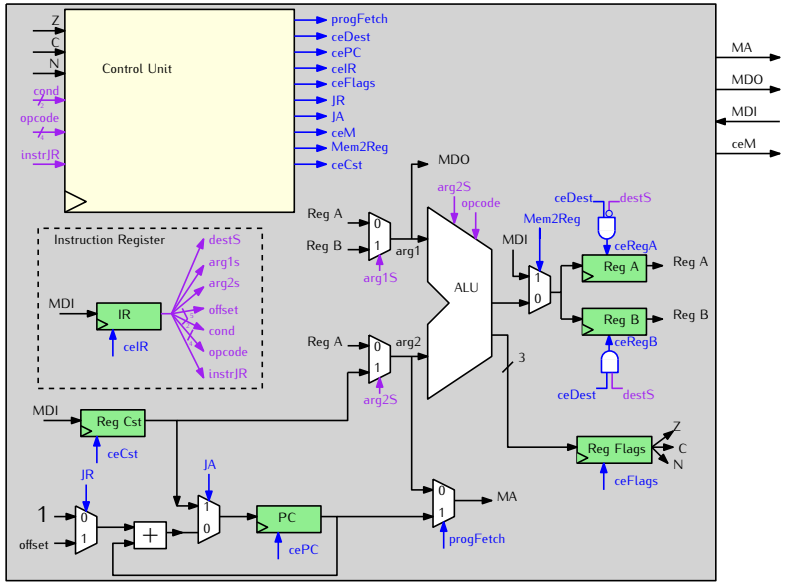

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# 2- Advance in Program



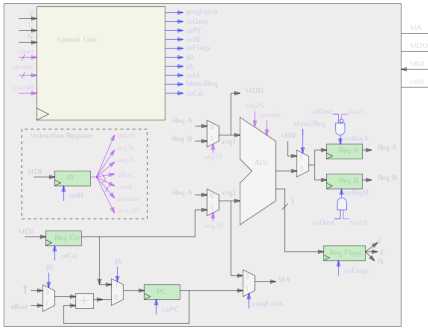

**MA**: Memory Address
**MDO**: Memory Data Output
**MDI**: Memory Data Input
**ceM**: clock-enable Memory (needs 1 when writing TO memory)

# DataPath - The Big Picture



# The DataPath

- Can store data for program:
  - Use General-Purpose Registers
  - Here, only 2: **RegA** and **RegB**
- Can compute on this data
- Talks to the memory through data and address buses, **MA**, **MDO** and **MDI**
- Dedicated Registers are used to track program execution:
  - **PC**: Program Counter
  - **IR**: Instruction Register
  - **Cst**: holds constants from Instructions
  - **Flags**: stores last values of ALU's flags: Z (zero), C (carry), N(egative)

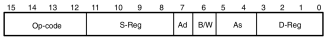


# Instruction Encoding

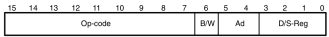
The **IR** register contains a **binary encoding** of the instruction being executed.

## msp430

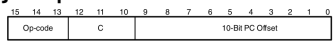
- 2 operands instructions




- 1 operand instructions



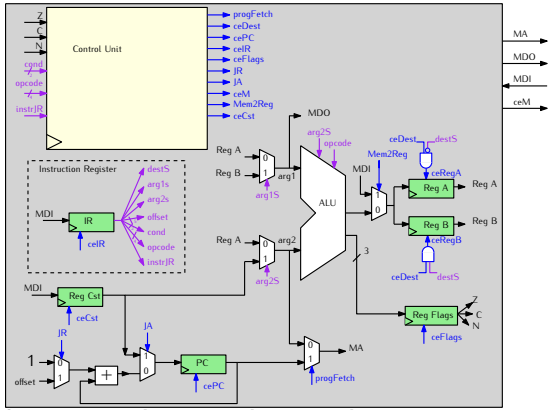

- jumps




## micro-machine

| bit | 7 | 6 | |
|---|---|---|---|
| instruction autres que JR | 0 | | co |
| saut relatif conditionnel | 1 | cond, vo | |

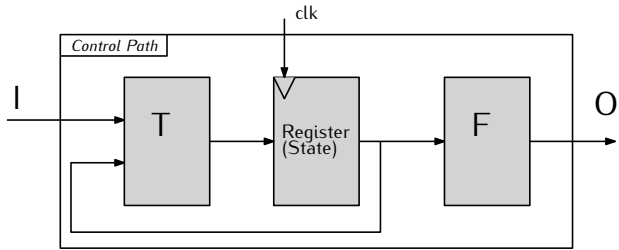
# The Control Unit




- Implements the passing from one instruction to the next
- Manipulates registers through “Chip Enable” commands: **ceRegA**, **ceRegB**, **cePC**, **ceIR**, etc.
- Manipulates memory through control signals: **ceM**
- Performs selection over data to be written to registers or buses: **JR**, **JA**, **progFetch**, etc.

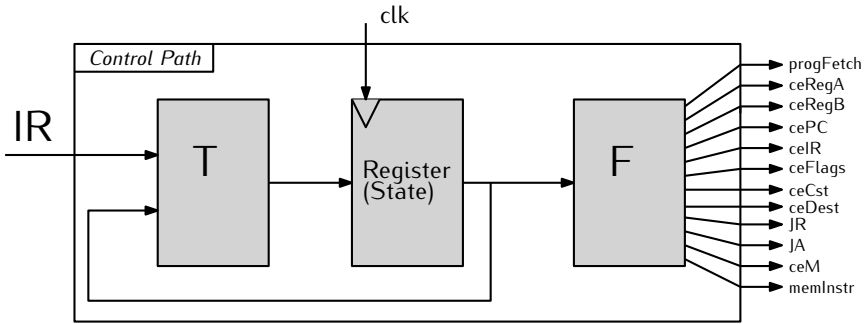
# Control Unit

The Control Unit is an Algorithmic State Machine (see Guillaume Beslon's lectures)




- F is the **Output** function
- T is the **Transition** function
- F and T are **combinatorial** circuits (no registers)

# Von Neumann's Control Unit - Interface



# Executing Programs - The Von Neumann Cycle

Given the structure of a program in a Von Neumann's machine, the algorithm to execute it is simple:

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

A Von Neumann machine is an ASM[1] that executes this **Von Neumann Cycle** such that:

Fetch Copy the current instruction bit-vector from the memory to the processor (register IR)

Decode Look at the instruction opcode to prepare the DataPath

Execute Process the data in the DataPath such that the processor **makes the behavior of the instruction happen**

!! **Don't forget** to move to the "**next**" instruction

[1] ASM = Abstract State Machine ... different from ASM/asm for "assembler"

# The Micro-machine

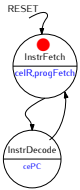
```
B -> A              21 -> B
B + A -> A          B - A?
not B -> A          lsr A -> A
A xor 12 -> A       B - A -> A
B xor -42 -> A
```

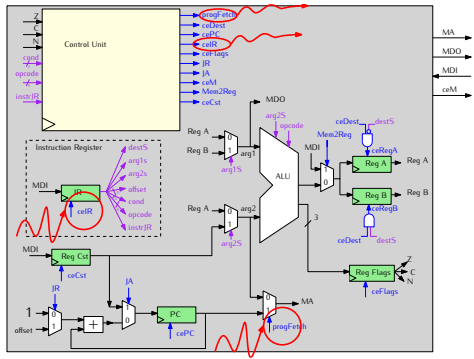
```
*A -> A             *A -> B
A -> *A             B -> *A
*cst -> A           *cst -> B
A -> *cst           B -> *cst
```

```
JA 42
```

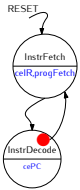
```
JR offset           JR offset IFZ
                    (executed if Z=1)
JR offset IFC       JR offset IFN
(executed if C=1)   (executed if N=1)
```

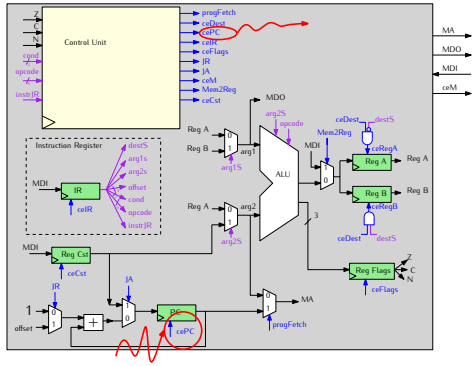
# The VN's automaton - Fetch-Decode



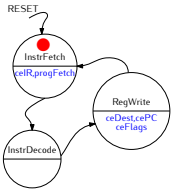



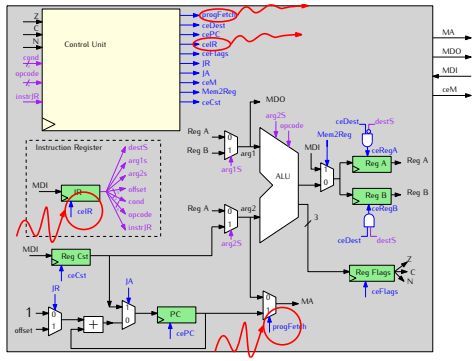
# The VN's automaton - Fetch-Decode



# The VN's automaton - Arithmetic/Logic ins on 1 byte

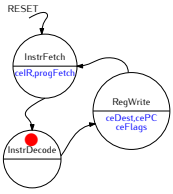


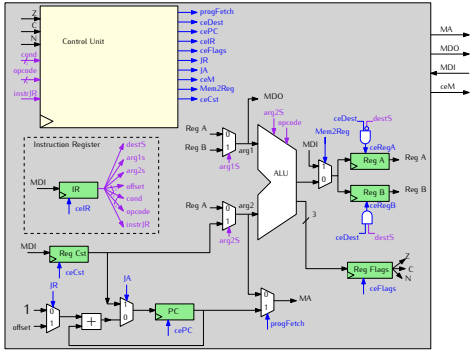




B + A -> A

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

# The VN's automaton - Arithmetic/Logic ins on 1 byte

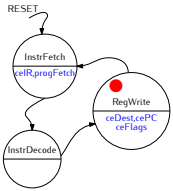


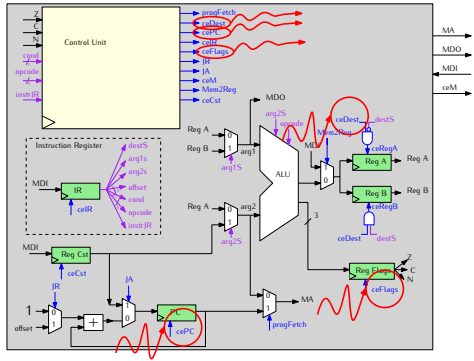




`B + A -> A`

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

# The VN's automaton - Arithmetic/Logic ins on 1 byte

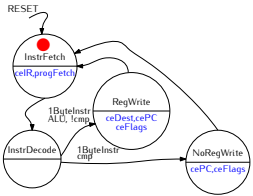


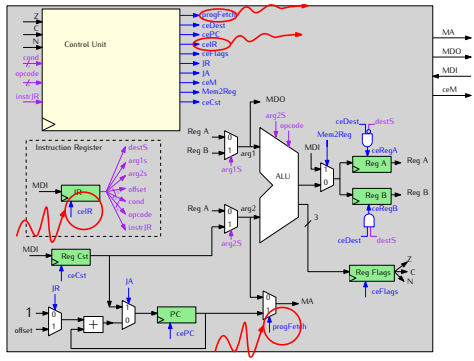




B + A -> A

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

# The VN's automaton - Comparison

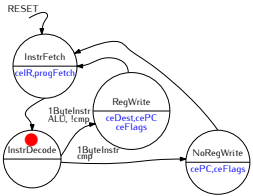


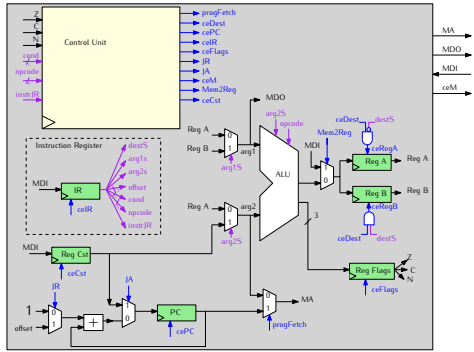




B - A?

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | $\alpha$ |

# The VN's automaton - Comparison

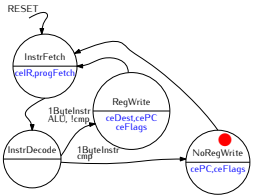


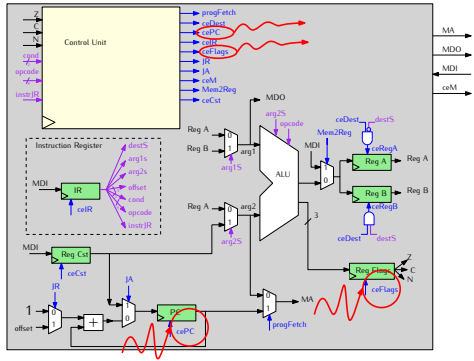




B - A?

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | $\alpha$ |

# The VN's automaton - Comparison

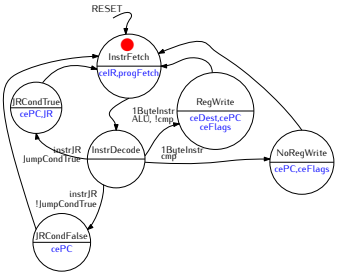


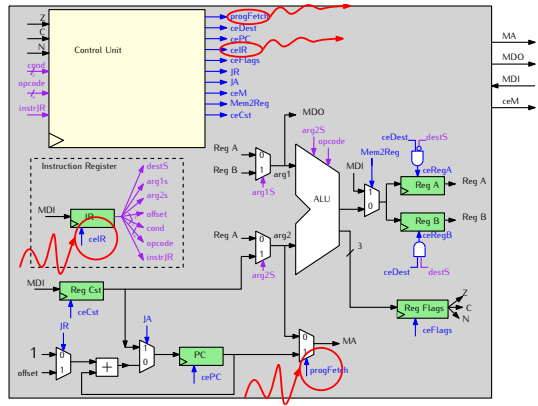




B - A?

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | $\alpha$ |

# The VN's automaton - Conditional relative jumps

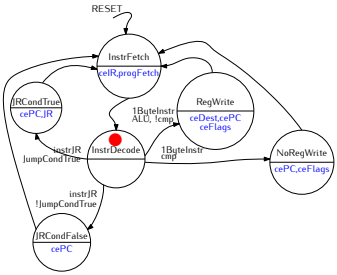


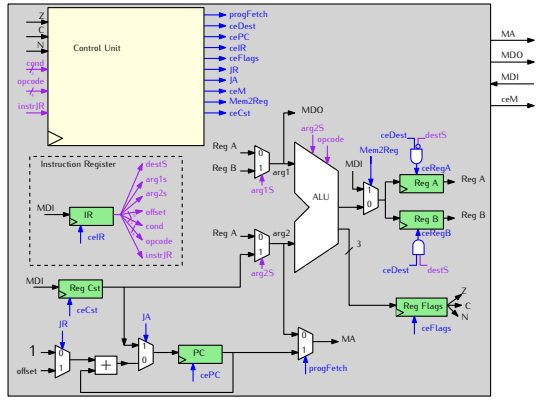




`JR -5 IFN`

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| saut relatif conditionnel | 1 | cond | | offset signé sur 5 bits | | | | |
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# The VN's automaton - Conditional relative jumps

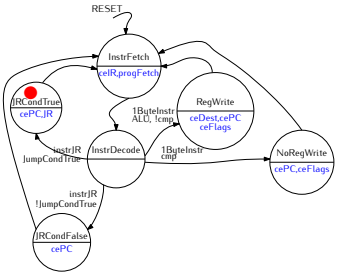


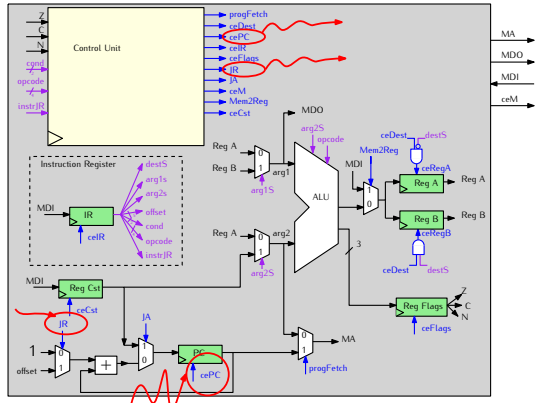

JR -5 IFN

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| saut relatif conditionnel | 1 | cond | | offset signé sur 5 bits | | | | |
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# The VN's automaton - Conditional relative jumps



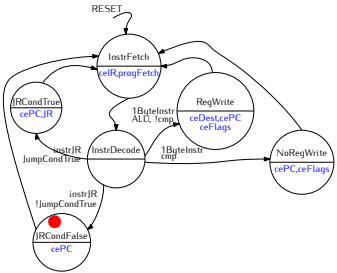

If condition is true (ie take the jump)



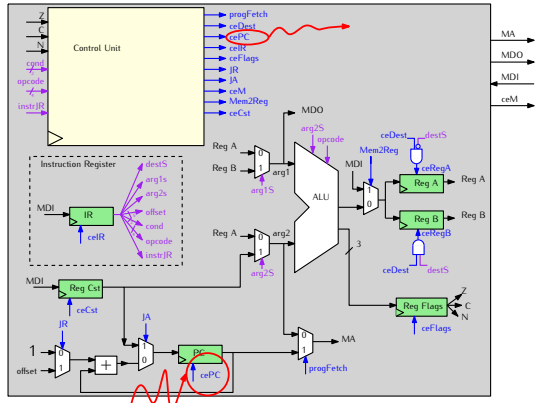

`JR -5 IFN`

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| saut relatif conditionnel | 1 | cond | | offset signé sur 5 bits | | | | |
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# The VN's automaton - Conditional relative jumps




If condition is false (ie DON'T take the jump)




`JR -5 IFN`

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| saut relatif conditionnel | 1 | cond | | offset signé sur 5 bits | | | | |
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

# Computing “conditions”

The automaton uses the `JumpCondTrue` for testing if a relative jump should be taken.
This signal needs to be computed from:

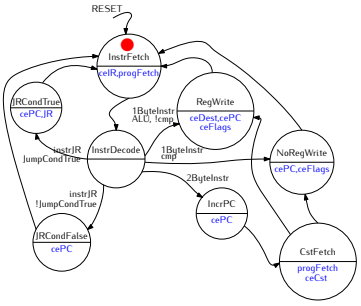
- flags (`Z`, `C`, `N`) which represent
- the instruction’s condition (`IFZ`, `IFC`, `IFN`, ... or none).

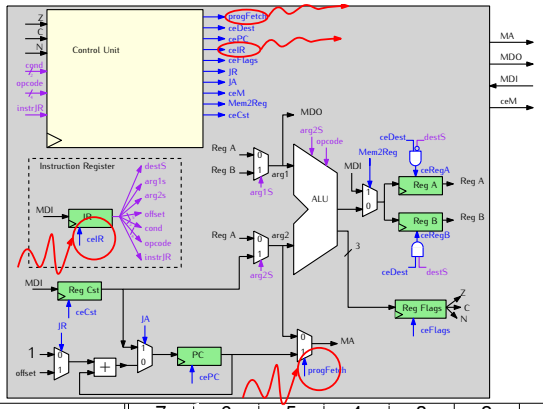
We want to **take the jump** if the “correct combination” occurs:

- `Z==1` and instruction’s condition is `IFZ` (`cond==01`)
- `C==1` and instruction’s condition is `IFC` (`cond==10`)
- `N==1` and instruction’s condition is `IFN` (`cond==11`)
- the instruction jumps **inconditionnally** (`cond==00`)

**TODO:** build the small combinatorial circuit that computes `JumpCondTrue` from `flags` and `cond`

# The VN's automaton - 2-bytes instructions

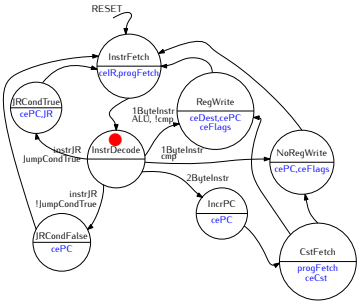


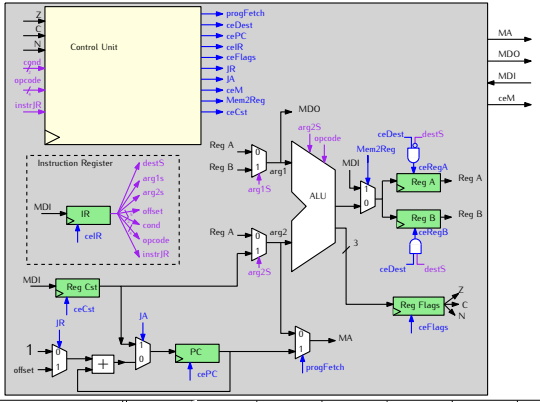




```
A xor 12 -> A
```

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# The VN's automaton - 2-bytes instructions

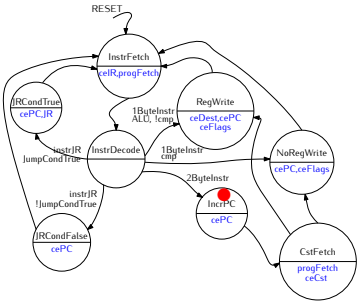


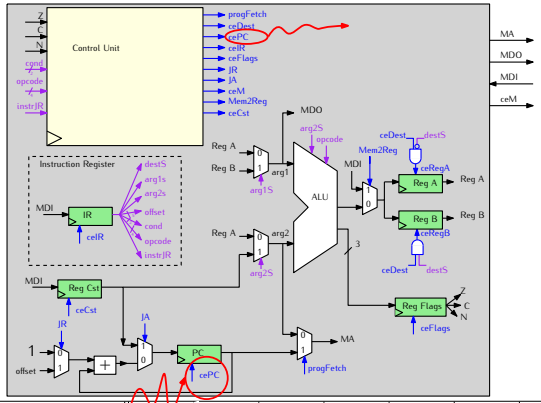




```
A xor 12 -> A
```

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# The VN's automaton - 2-bytes instructions

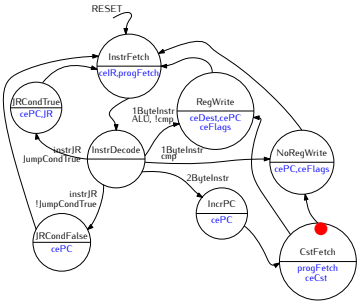


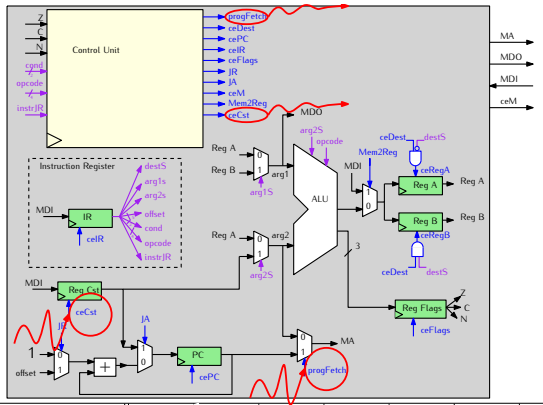




```
A xor 12 -> A
```

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# The VN's automaton - 2-bytes instructions

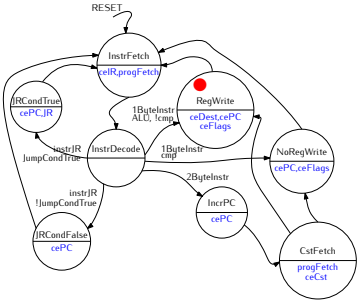


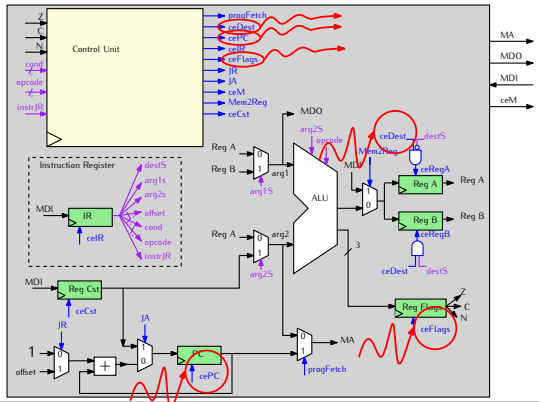




A xor 12 -> A

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# The VN's automaton - 2-bytes instructions

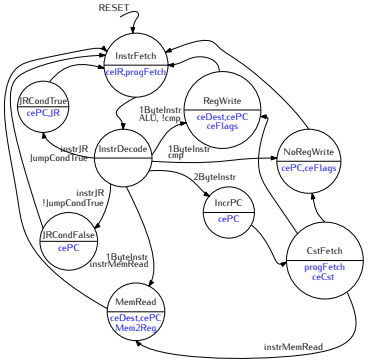


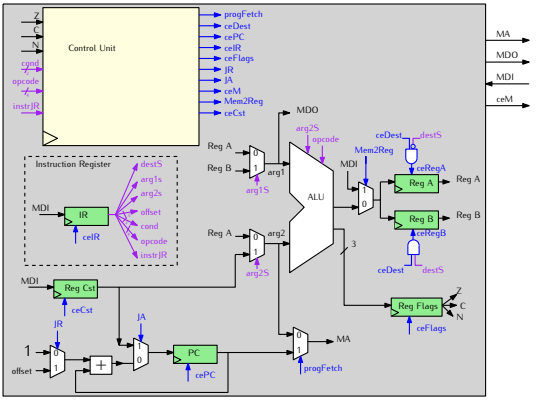




```
A xor 12 -> A
```

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

# The VN's automaton - Memory reads

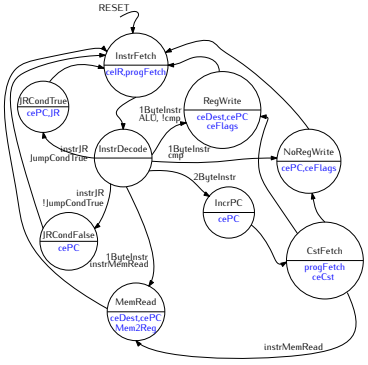


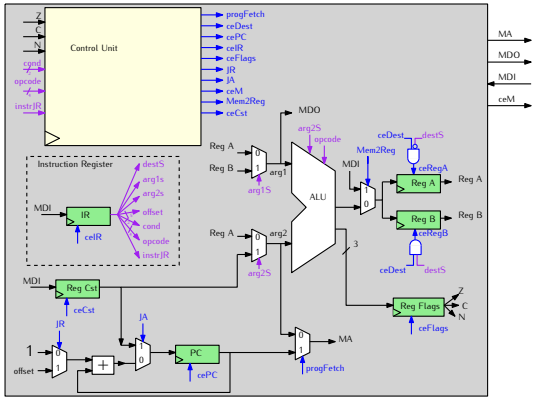




*A -> B

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

# The VN's automaton - Memory reads

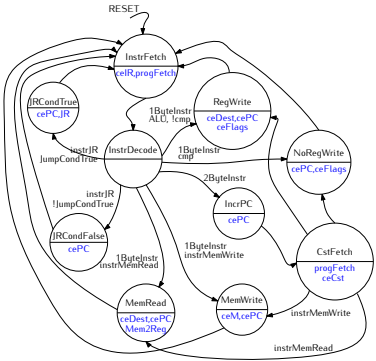


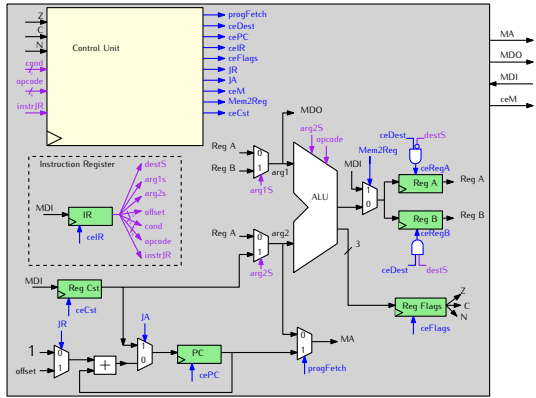




| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

```
*105 -> B
```

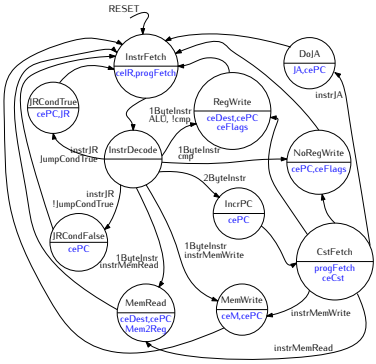
# The VN's automaton - Memory writes

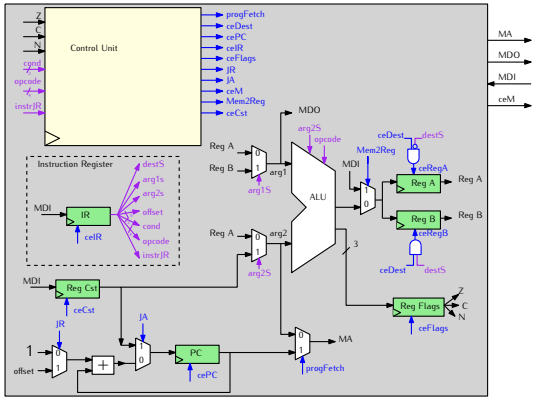



```
B -> *A
B -> *200
```

# The VN's automaton - Absolute Jumps



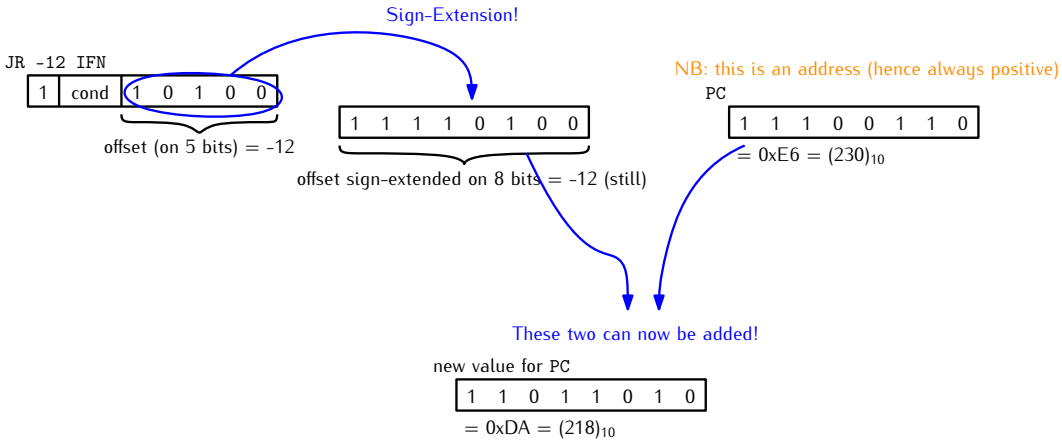

JA 42

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| instruction autres que JR | 0 | codeop | | | | arg2S | arg1S | destS |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| constante encodée sur 8 bits | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

# Outline

# Sign Extension

- Some instructions manipulate data encoded on different numbers of bits



# Outline

Sign Extension

Byte-order

(A bit) more advanced topics

# Byte-order: Definitions (1)

- By convention, a computer will store in memory bytes of a data word in a given order.

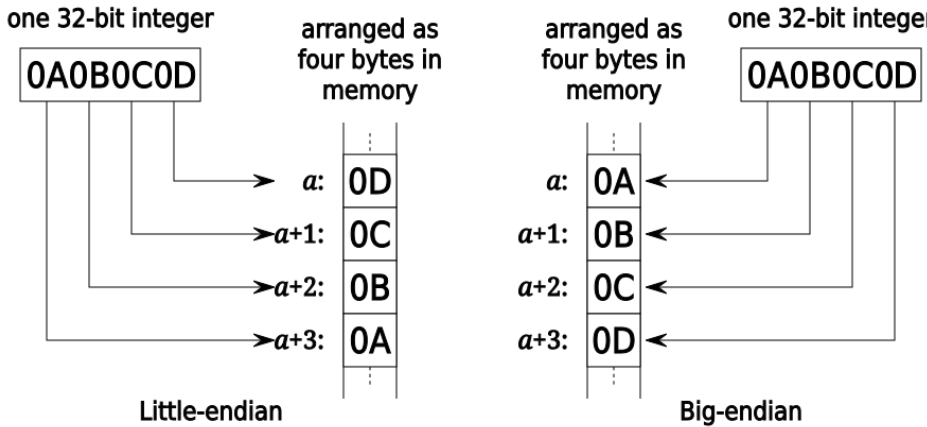
> **Definition 1: Big-endianness**
> $\triangleq$ *convention by which a computer stores bytes of a number in memory from the most-significant byte (at the smallest address) to the least-significant byte (at the highest address). Such a computer is called* ***big-endian****.*

# Byte-order: Definitions (2)

**Definition 2: Little-endianness**

$\triangleq$ *convention by which a computer stores bytes of a number in memory from the less-significant byte (at the smallest address) to the most-significant byte (at the highest address). Such a computer is called* ***little-endian****.*

# Byte-order: Example



# Outline

Sign Extension

Byte-order

(A bit) more advanced topics

# (A bit) more advanced topics

# Un-Pipelined Execution

The behavior of the processor (Fetch/Decode/execute) can be refined:

**IF** : Instruction Fetch

- Copy instruction word from memory to IR

**ID** : Instruction Decode

- Prepare operands
- `JA 10`: read constant from memory
- `JR +3 IFN`: extend 5-bits +3 to 8-bits

**EX** : Execute

- Send correct control signal to ALU
- Let arguments run towards ALU

**MEM** : Write/Read stuff to/from memory

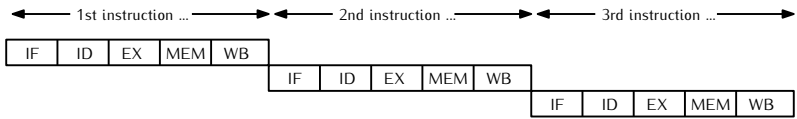
- `A -> *42`
- copy the current value of register `A` into memory at address 42.

**WB** : Writeback (result to register)

- Write value out of ALU into target register

# Un-Pipelined Execution

Until now, the execution of *n* instructions looks like this:




- If each step takes $\delta$ns,
- Then 1 instruction takes $5 * \delta$ns,
- Then **n instructions take** $\underline{n * 5 * \delta \textbf{ns}}$ to execute (ouch!).

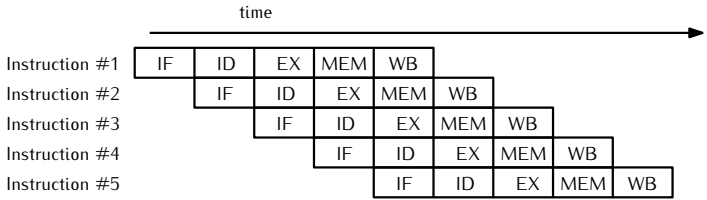
# Pipeline - Principle

- **Steps are independent** from one another
- Each step (IF, ID, EX, MEM, WB) can be realized with a **distinct part of the datapath**
- Data can move from one stage of the datapath to the next with **no delay**

The Processor datapath can work like an **Assembly Line**

# Pipelined Execution

Now, the execution of the same *n* instructions looks like this:



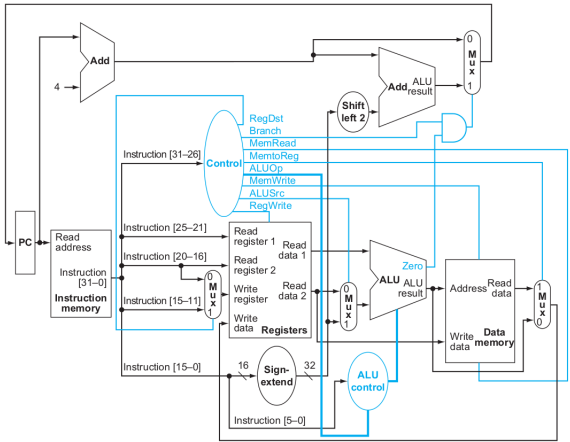

- If each steps takes $\delta$ns, then **n instructions take** $5 * \delta + (n - 1) * \delta$ = $(\mathbf{n} + \mathbf{4}) * \delta$ **ns** to execute.
- Expected gain: Processor gets D times faster (where D is the Depth of the pipeline).
- In nominal mode of course (see pipeline hazards)

# Pipelined Execution

What's the implication on the processor design?

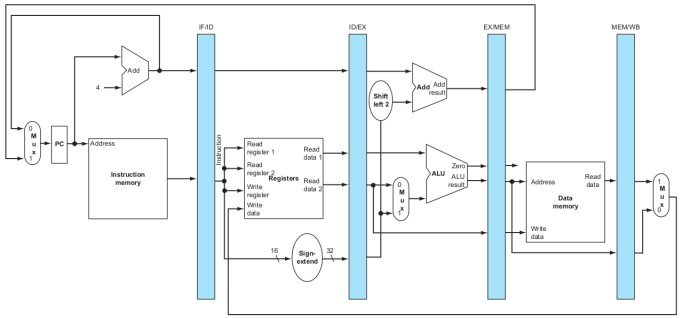
- The pipeline stages are separate parts of the CPU
- They are organized so that each stage takes one instruction at each new clock cycle
- An instruction is "passed" from one stage to the next

# Un-pipelined versus pipelined architecture (1/2)




NB: This is taken from D. A. Patterson, & J. L. Hennessy, *Computer Organization and Design*, p265. You should convince yourself that our micro-machine is **very close** to this. See also `https://en.wikipedia.org/wiki/MIPS_architecture`

# Un-pipelined versus pipelined architecture (2/2)




- Each **blue line** corresponds to a set of **stage-specific registers**
- eg IF/ID = info produced by Instruction Fetch, and needed by Instruction Decode.
- These are used to store data that needs to be passed from one stage to the next
- NB: ontrol signals are missing!!

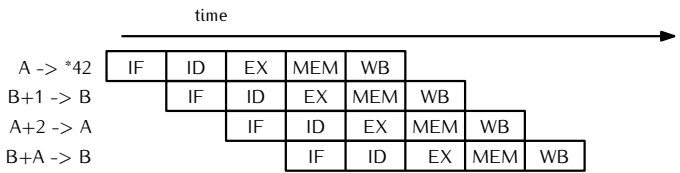
# Pipeline hazards

The pipeline's behavior may be disrupted by the following:

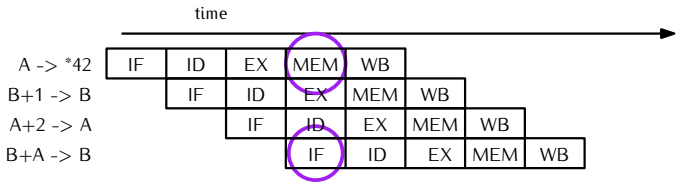
- **Structural hazard**: two instructions in different stages of the pipeline require the same resource.
- **Data hazard**: the data required by an instruction was not already produced by a preceeding instruction.
- **Control hazard**: an instruction execution is interrupted, typically by a branch.

- The general solution consists in inserting **bubbles** into the pipeline.
- This amounts to adding **NOPs**, ie "**do-nothing instructions**" into the program.

# Pipeline hazards



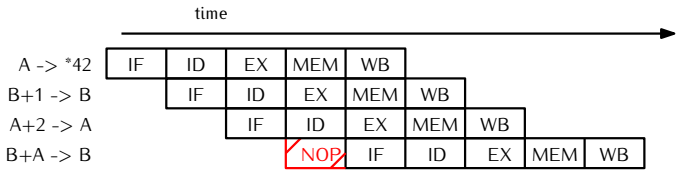
# Pipeline hazards




1. Structural Hazard

Both `A->*42` and `B+A->B` require memory simultaneously. We need to **delay 2nd instruction**.

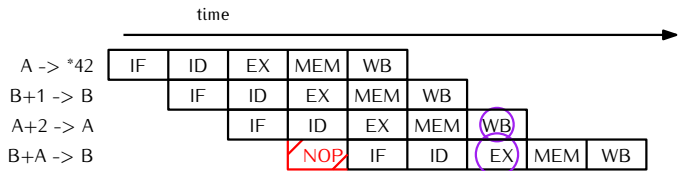
# Pipeline hazards




1. Structural Hazard

Both `A->*42` and `B+A->B` require memory simultaneously. We need to **delay 2nd instruction**.

# Pipeline hazards

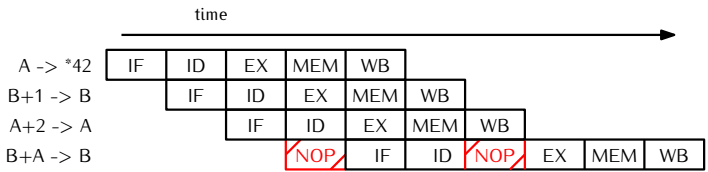



2. Data Hazard

`B+A->B` can only be executed correctly when `A+2->A` is fully finished, ie we need to wait for the value of `A` to be fully updated.

NB: no structural hazard here because `B+1->B` doesn't use the MEM stage.

# Pipeline hazards

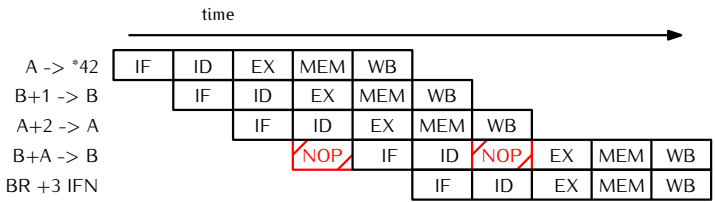



2. Data Hazard

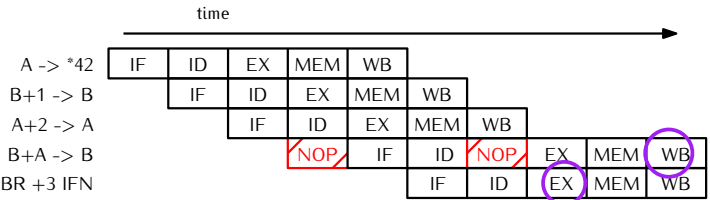
`B+A->B` can only be executed correctly when `A+2->A` is fully finished, ie we need to wait for the value of `A` to be fully updated.

NB: no structural hazard here because `B+1->B` doesn't use the MEM stage.

# Pipeline hazards



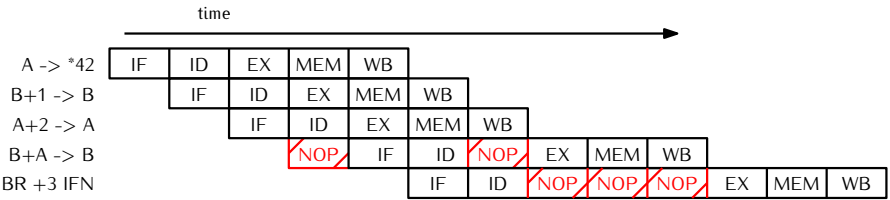
# Pipeline hazards




3. Control Hazard

The control flow associated to instruction `BR +3 IFN` can only be evaluated when **B** is fully updated by previous instruction.

# Pipeline hazards




3. Control Hazard

The control flow associated to instruction `BR +3 IFN` can only be evaluated when **B** is fully updated by previous instruction.

# Other sources of optimization (1/2)

## Inside the CPU

These are advanced topics, not addressed in AO. Got and check'em out if you are interested.

- Superscalar[2]: A **superscalar processor** executes multiple instructions in parallel by using multiple execution units (eg several ALUs), whereas the latter (pipeline) executes multiple instructions in the same execution unit in parallel by dividing the execution unit into different phases
- Very-Large Instruction Words[3]: A **VLIW processor** allows programs to explicitly specify instructions to execute in parallel
- support for multi-threading

[2] https://en.wikipedia.org/wiki/Superscalar_processor
[3] https://en.wikipedia.org/wiki/Very_long_instruction_word

# Other sources of optimization (2/2)

## Outside the CPU

- multi-core
- many-core, GPUs, etc.