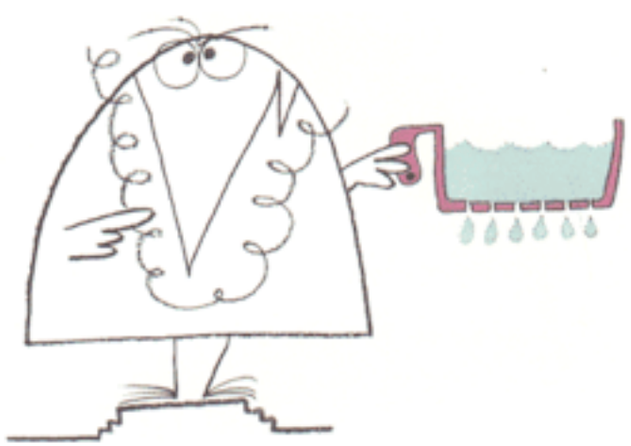
ELEC-H-473

Microprocessor architectures



Lecture 02,03
Dragomir Milojevic
dmilojev@ulb.ac.be

Outline

1. Basic computer architecture concepts
2. Minimum system architecture
3. Instruction execution cycle
4. ISA
5. RISC vs. CISC
6. Instruction execution cycle — physical aspects
7. Pipeline execution and benefits

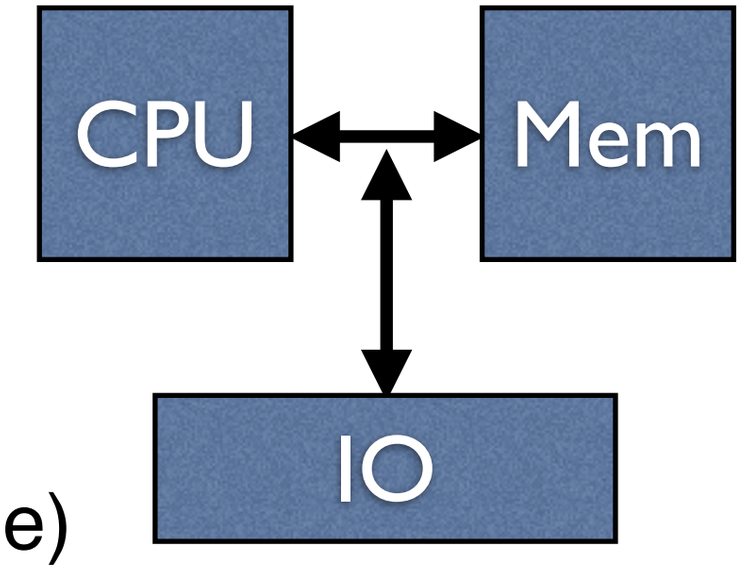
1. Basic computer architecture concepts

Background

- What defines a computer?
  - ✓ The difference with electronic calculator :
    - ✦ Computer stores electronically the information that controls the computational process
    - ✦ This is the same as in the Turing machine (the “action table” even if hardwired, is somehow “stored” in the memory)
- This concept was known to **John von Neumann**, the author of : *First Draft of a Report on the EDVAC,* 101 pages of unfinished, unpublished report (subject to controversy …)
- This is considered to be the basis for the designs ENIAC (USA) and Colossus (UK) : first electronic calculators to be considered **computers**

(Very) simplified model of computer arch.

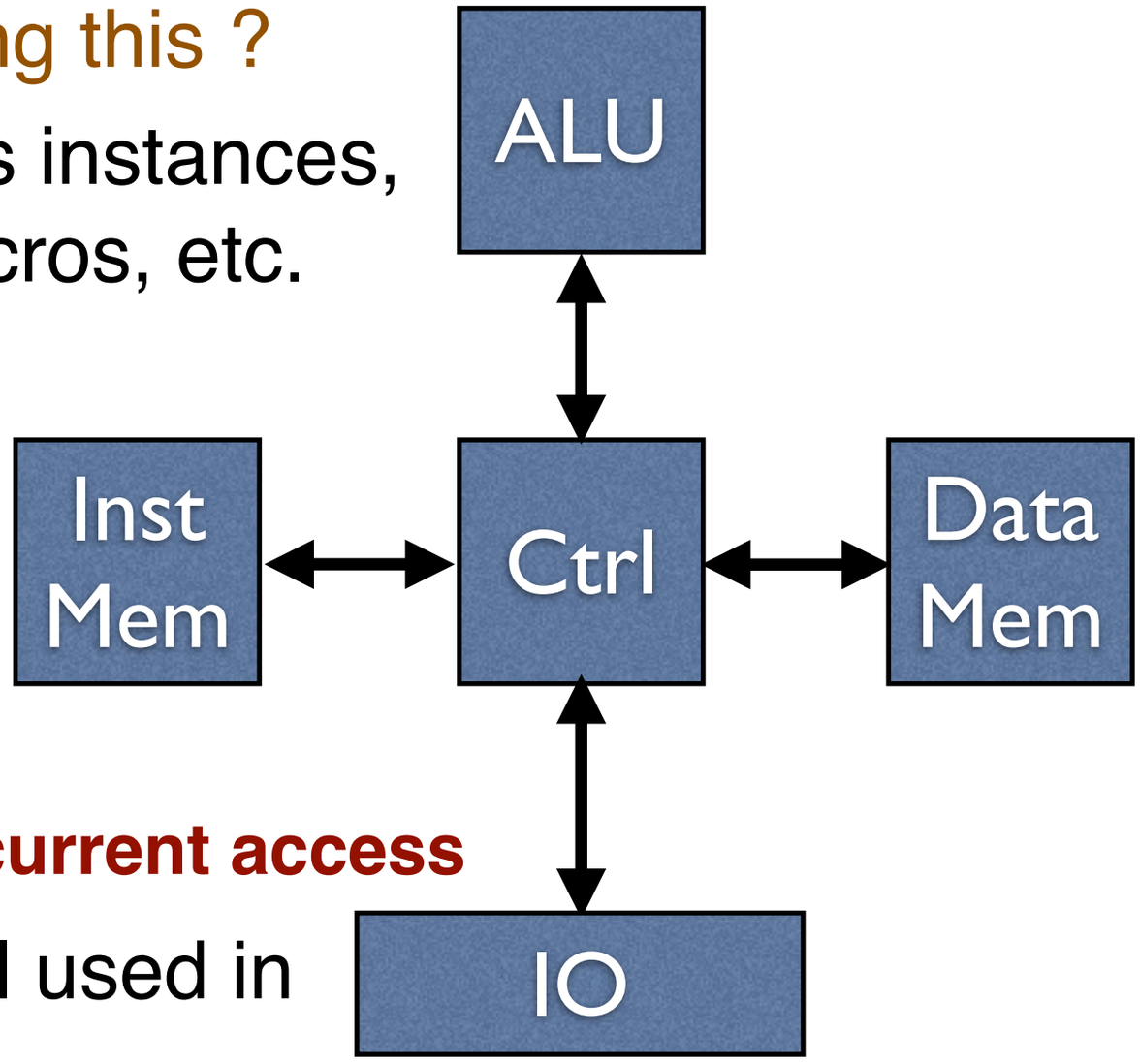
- First draft → **von Neumann architecture** (computer model) :
  - ✓ Instructions for the **Central Processor Unit** (CPU) as defined by the program stored in the **memory**
  - ✓ and the data for computation (also stored in the memory)
  - ✓ **Both memories are the SAME !** (What is the consequence to the system operation?)



- CPU :
  - ✓ Arithmetic-Logic Unit (ALU) + Control Unit + Registers (Register File)
- Input/output block interfaces to the outside world
- Communication between different units — a bus : a collection of wires (What is the ≠ between Point-to-Point connection and a bus?)

Harvard architecture — a variant

- **Harvard architecture**
  - ✓ **Data and Instruction memories are NOT the SAME !**
- What is the advantage in doing this ?
- These can be separate RAMs instances, physically different chips, macros, etc.
- Important consequence
  - ✓ Physically different means:
    - ✦ diff electrical properties
    - ✦ timing and/or sizes
    - ✦ **but most importantly concurrent access**
- Pure Harvard architecture still used in DSPs and micro-controllers



Modified Harvard architecture

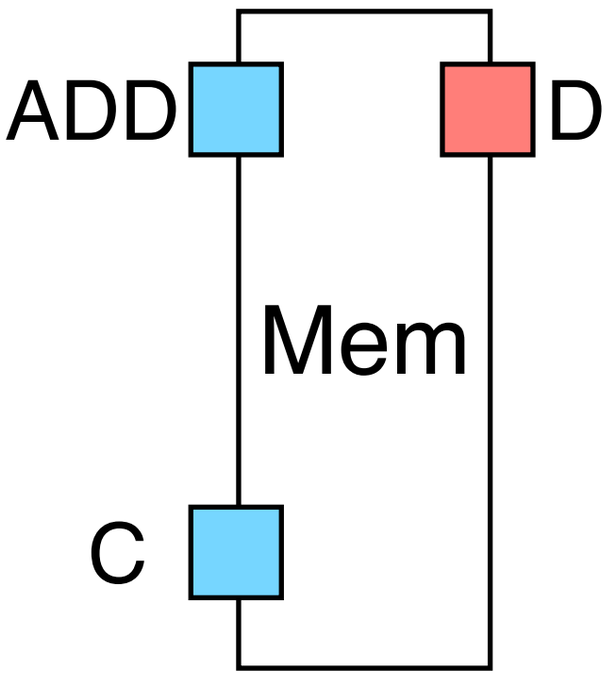
- Combines the advantages of the von Neumann architecture :
  - ✓ **Instructions are treated as data**
    - ✦ Stored at same storage space (e.g. HDD, or central memory)
    - ✦ You can do compilation at runtime
    - ✦ You can write self-modifying code
- … and the advantages of the Harvard architecture
  - ✓ Concurrent instruction/data access
- Today computers are most of the time built using **modified Harvard model** because of the cache memory hierarchy introduced to overcome the problem of (S,D)RAM/Logic scaling discrepancy (separate L1 for Instructions/Data)

Memory

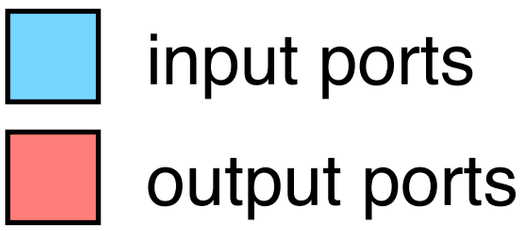
- Stores data and instructions (separately or not, depending on the memory hierarchy level, more on this later)
- Both data & instructions are binary data (computer doesn't have the awareness of what is what, i.e. it can't make a difference between program and data)
- Memories are like Look-up Tables (LUTs), defined using :
  - ✓ **Entry** — defined as address A (location where do you want to read/write)
  - ✓ **Control** — defined as signals C, used to drive the control logic of the memory (typically just ENABLE and R/W for SRAMs, note that this can be more complicated for DRAMs)
  - ✓ **Data** — the actual data
    - ✦ **stored** at address A and **accessed** when **reading**
    - ✦ **stored** at address A when **writing**

Memory and port view of the module

- **Entry** → ADDRESS [n-1 : 0]
  - ✓ The number of bits of address determines the number of available locations (here 2^n)
  - ✓ Typical “location” stores one bit of info
  - ✓ Multiple bits are concatenated to make a **word**
  - ✓ This organization can vary, but typically:
    - ✦ one address → points to one byte
    - ✦ $2^{something}$ bytes makes a larger data word that could be accessed in 1 cycle

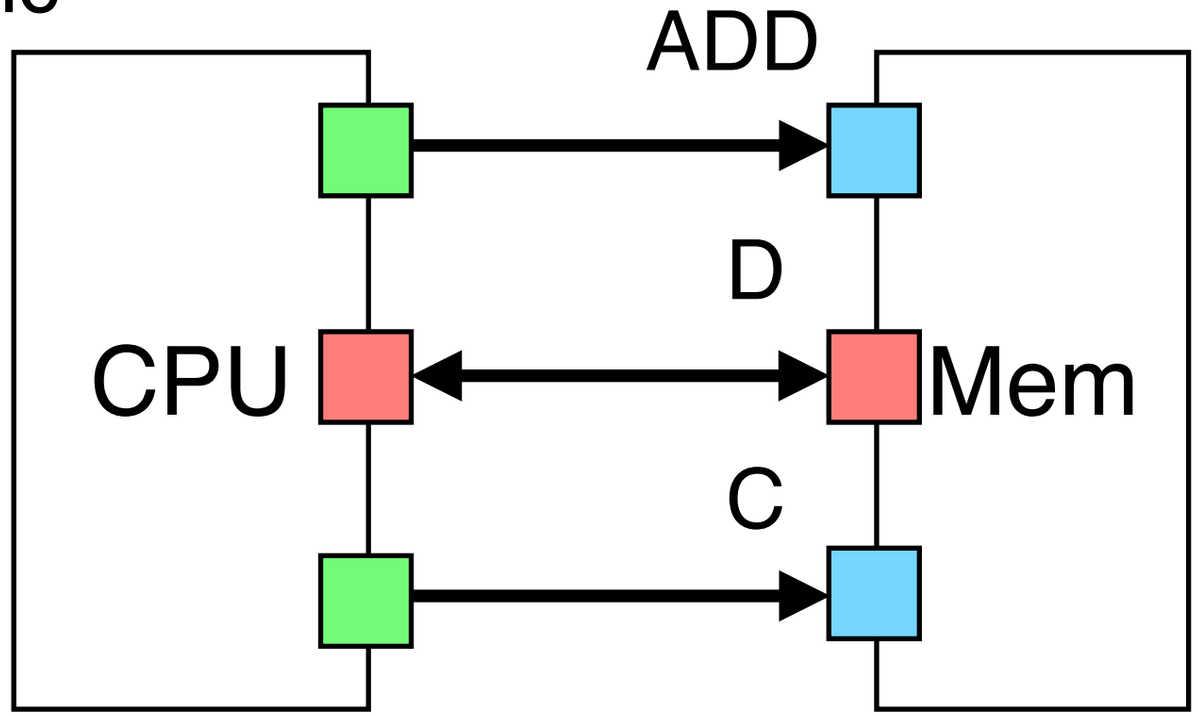


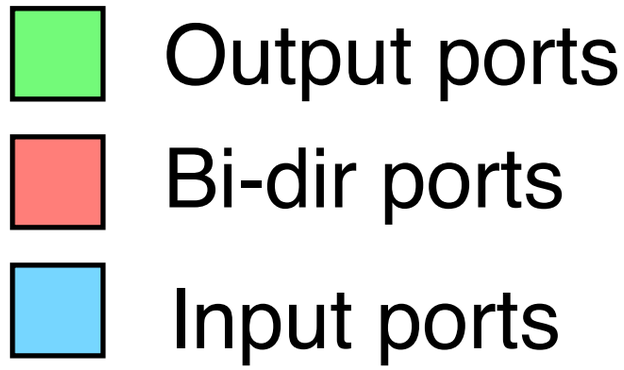
- **Control** → CTRL
- **Data** → D [m-1 : 0]
  - ✓ m wires → m bits that are **transferred in parallel**
  - ✓ This is also known as **bit-level parallelism**
  - ✓ Important because this determines CPU to memory bandwidth



Ports view (and why arrows have meaning)

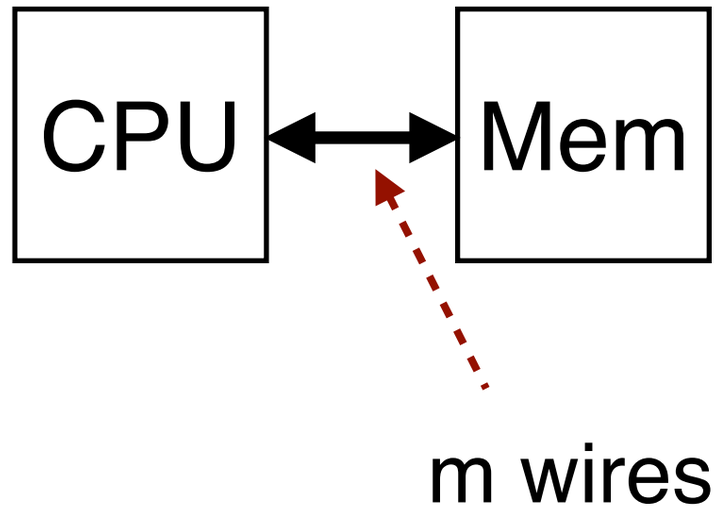
- Since Memory can be READ/WRITE the DATA port is typically **bi-directional** meaning it can be input **or** output depending on the control logic
  - ✓ It is input on WRITE
  - ✓ It is output on READ
- If this is the case, you have similar (mirror) situation on the CPU side:
  - ✓ CPU acts like a **master**
  - ✓ Memory acts like a **slave**
  - ✓ DATA is bi-directional
- CPU translates R/W instructions





System performance

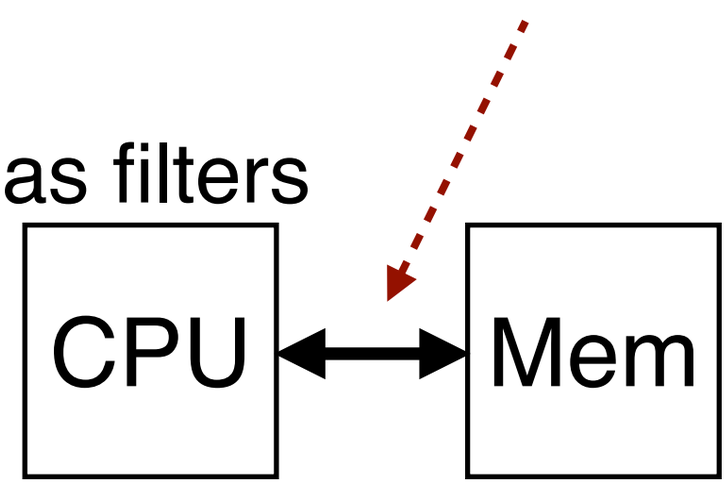
- First view :
  - ✓ If CPU can process N instructions per sec,
  - ✓ the memory should deliver the data/instructions to the CPU, so that he can effectively execute N instructions.
  - ✓ If this is not the case, the **CPU is stalled** (we waist CPU cycles)



- Any system can be in one of these 2 situations:
  - ✓ it can be **bounded by computation**,
  - ✓ or by the **memory access**
  - ✓ Good operating point is when we are break-even
- CPU/memory bandwidth depends on the m but also on something else? What influences the system bandwidth?

CPU — MEM

- CPU and MEM are interconnected with wires
- Wires (Cu conductors) have parasitic resistances/ capacitances/ inductance (RLC)
- At higher frequencies parasitics will act as filters
- For any given connection there will be a **max F** that we can have …



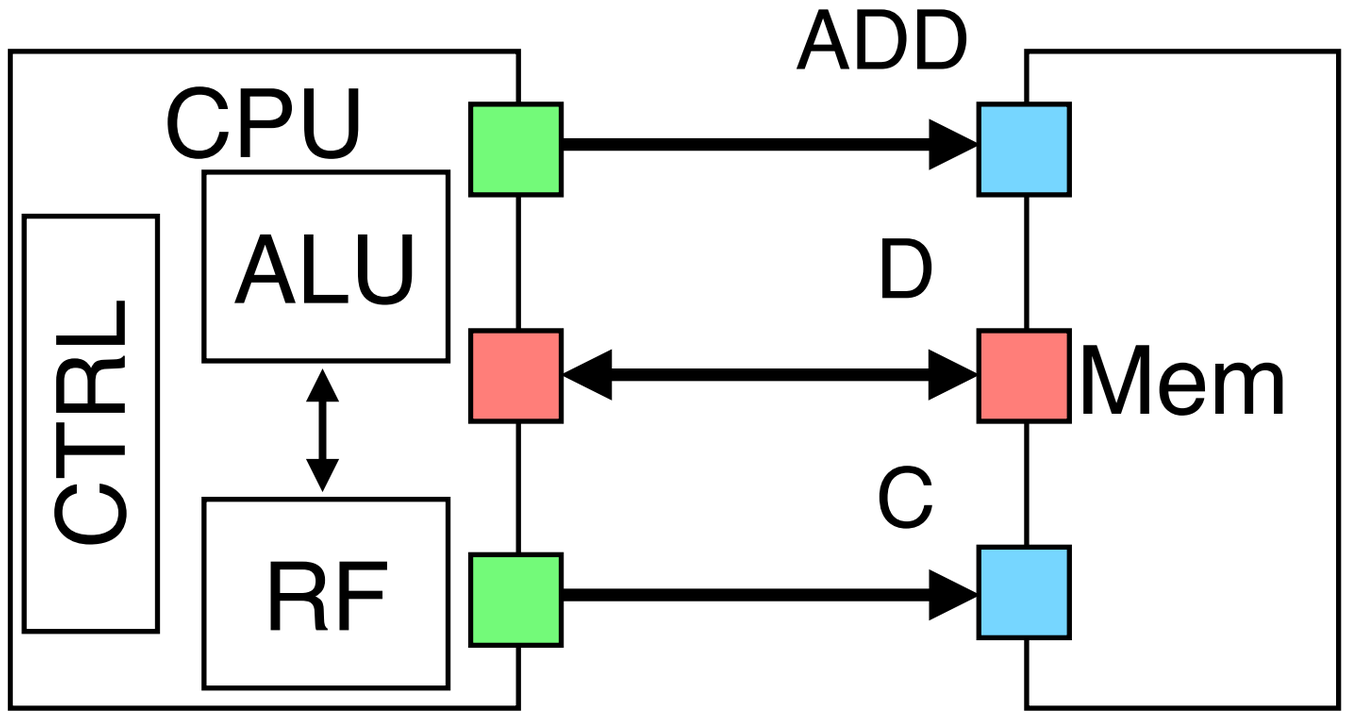
- It is a function of the wire geometry & distance : closer the better (this is why DRAMs are so closed to the CPU)
- THIS IS TRUE AT WHATER SCALE YOU ARE LOOKING AT (at the scale of the PCB but also IC)
- **Bandwidth = Frequency X number of bits**

On masters and slaves

- **Master** can initiate operations on one or more slaves
- **Slave** can only **accept**, or **refuse** the operation; it can't initiate the transfer by himself
- In our model the CPU is the master, the memory is the slave, and this means that the CPU is initiating the memory access
- CPU needs **time** to initiate the access, and this time is the time that the CPU will not use to compute our DATA …
  → It is a wasted time in a way
- In today's machine the actual cores don't do that (more on this later)

Closer look into the CPU : RF + CTRL

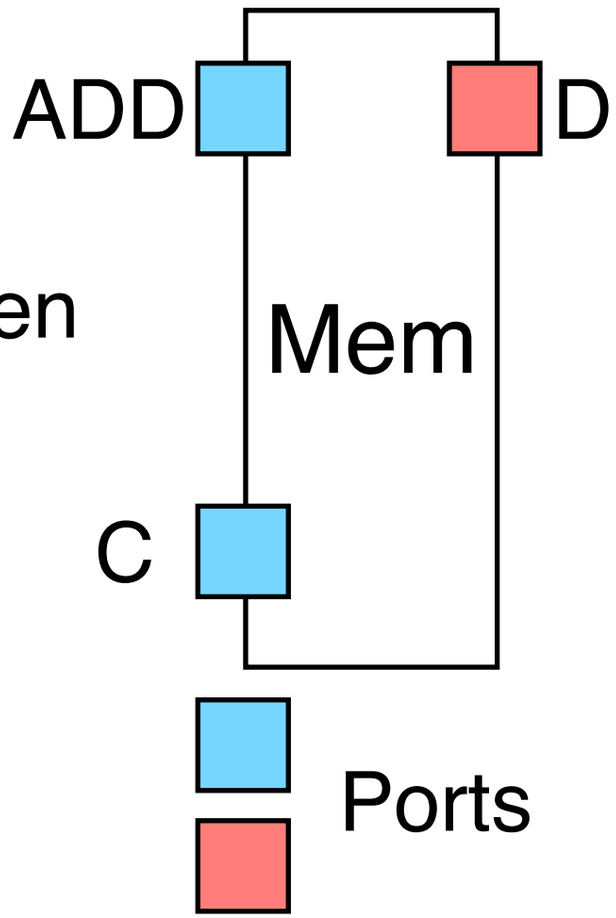
- ALU — unit that actually performs computations
- **RF — Register file** : set of named registers that store **operands** = variables on which the ALU does computation
- They can be **source** (SRC) – when they are arguments, or **destination** (DST) when they target function evaluation



- Both ALU & RF need **control signals** (inputs) to steer their operation and these need to be properly orchestrated
- These signals are generated by **CTRL unit**

Register Files (RF)

- Typical RFs are SRAMs
- Typical SRAMs are **single ported devices**, i.e. 1 SRAM can read **OR** write through that port → only one operation performed at a time (in 1clk cycle)
- Typical RFs are **multi-port** access memories, i.e. they allow multiple data to be read or written in the same time
- RFs : few read ports & only one write port to avoid data coherency problems
  - ✓ **What happens when we have two simultaneous writes at the same location?**
- Multiple-access ports allow to speed up access to the data that needs to be processed



Computer architecture: not just a HW model

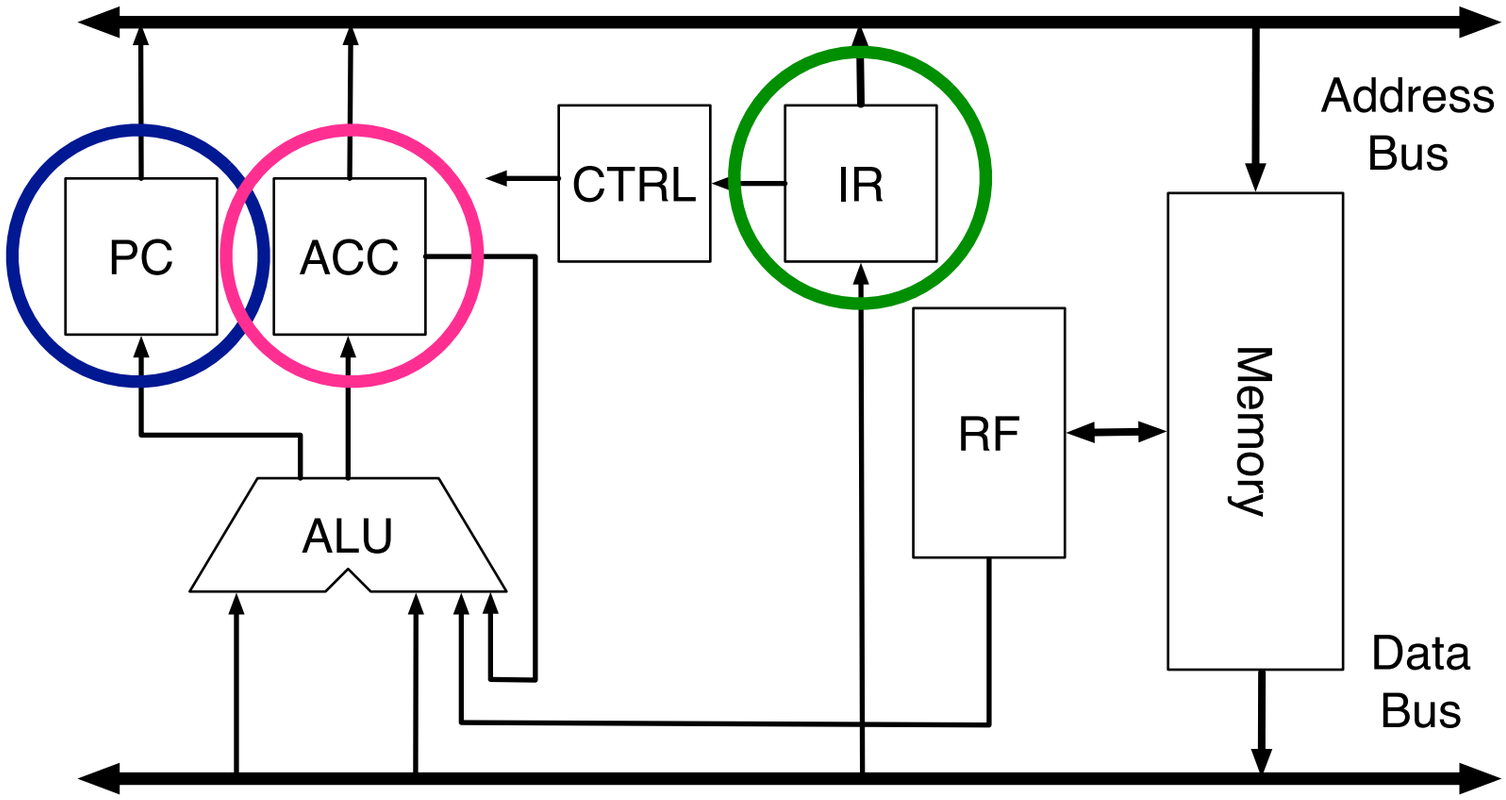
- ALU, RF, CTRL logic and the Memory are not enough to make a computer
- We still miss :
  - ✓ some **architectural** details necessary to make a real computer (even though minimalist)
  - ✓ the **instruction execution model** — or, how the instructions in the context of a computer program are handled
  - ✓ the actual specification of instructions, i.e. **Instruction Set Architecture**
- These 3 notions combined together are necessary to build a computer

2. Minimum system architecture

Minimum system architecture

- We start with simple system **micro - (*μ* or u) architecture** : a description of all components in the system and the way they interact

To ALU + CTRL we add few new components :

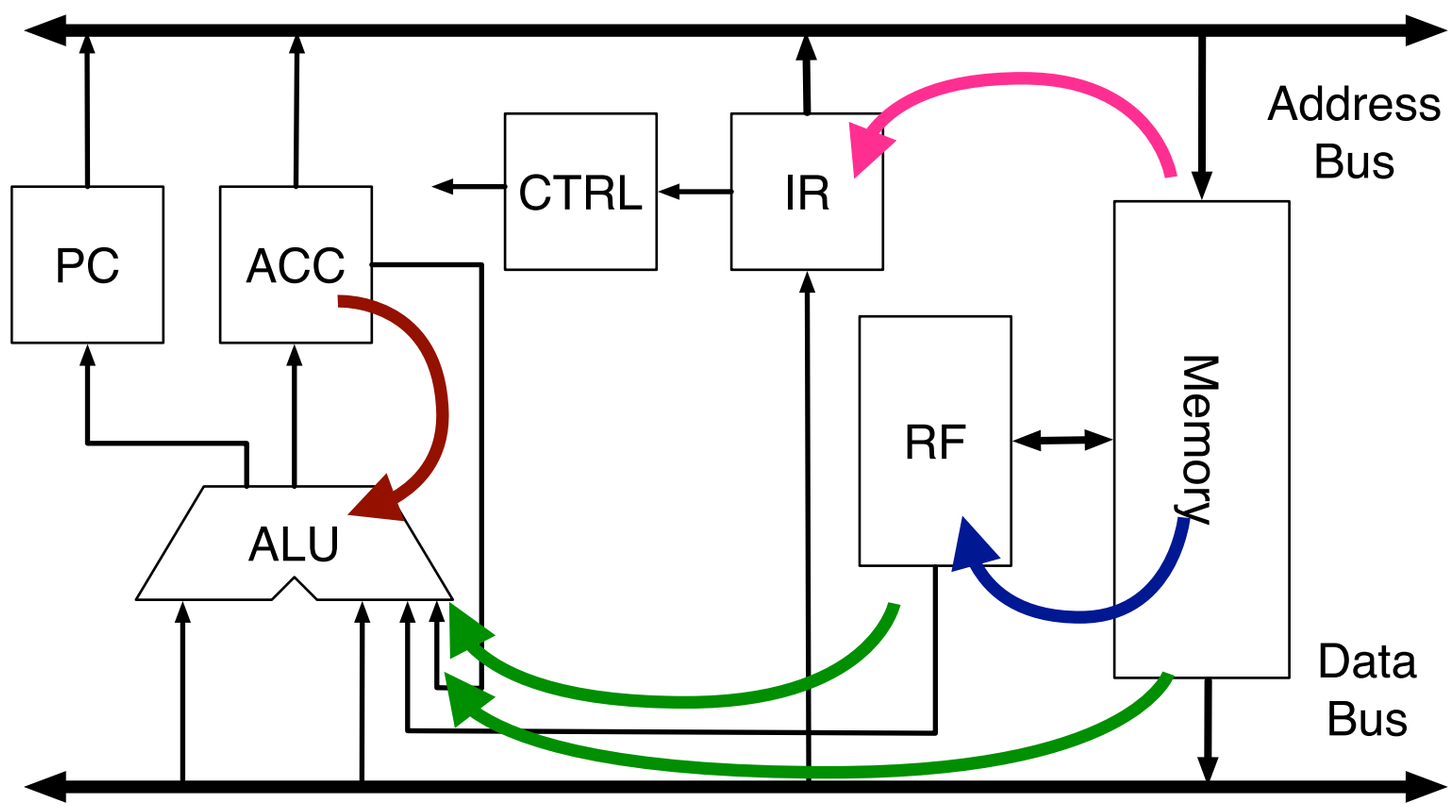


- **PC — Program counter** : holds the address of the current instruction (register)
- **IR — Instruction register** : holds the code of the current instruction **(opcode)**
- **ACC — Dedicated register** that is R/W directly from the ALU

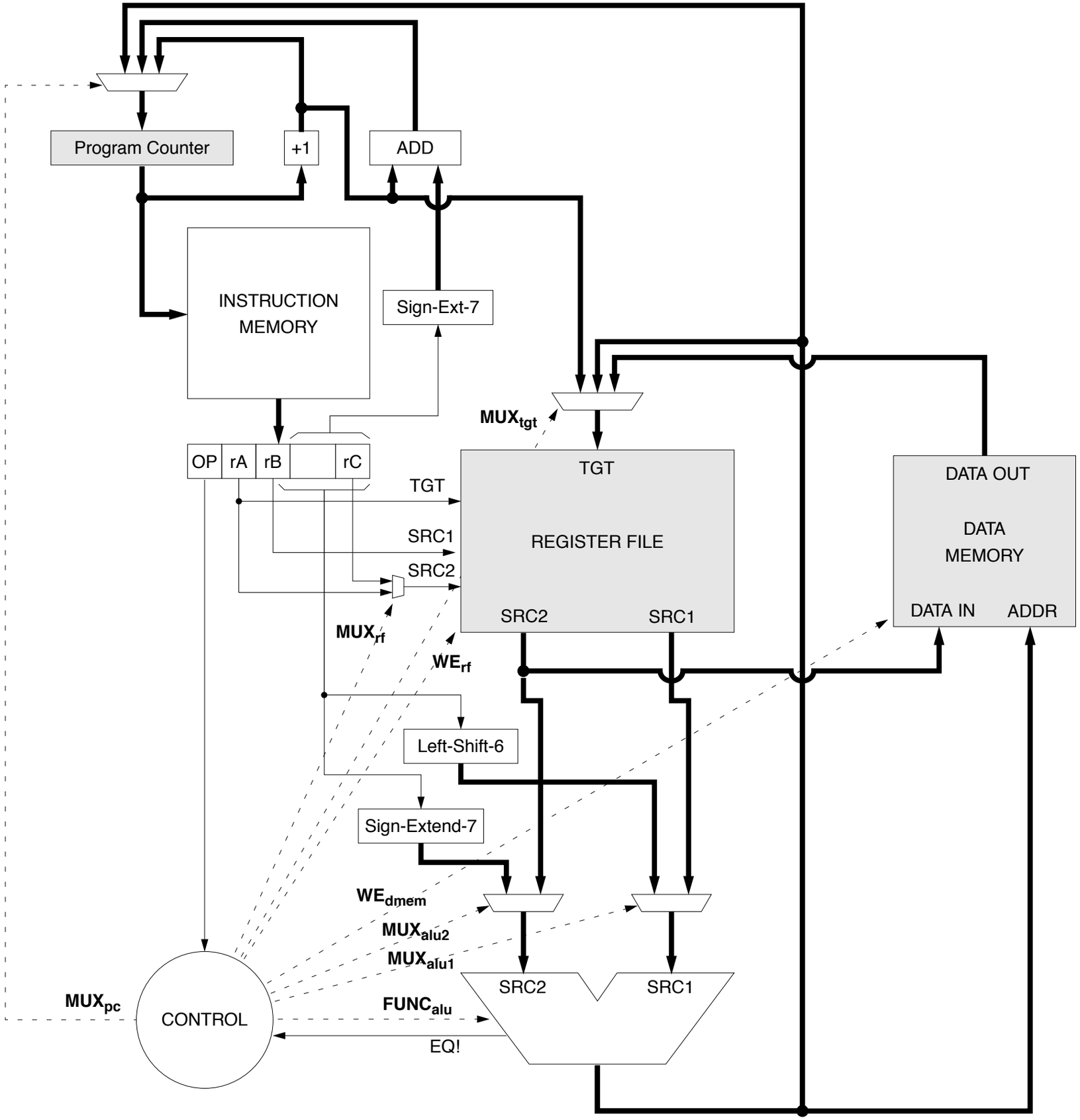
Minimum system architecture

- Note the data paths :
  - ✓ **From memory to Register File (RF)**
  - ✓ **From memory to Instruction Register (IR)**
  - ✓ **ALU can compute from RF, but also from the Memory**
  - ✓ **ALU can compute on result from the previous operation using ACC**

- How this can be achieved ?
- Could you draw a simplified schematic circuit ?
- How ALU choses the operand ?



RISC 16

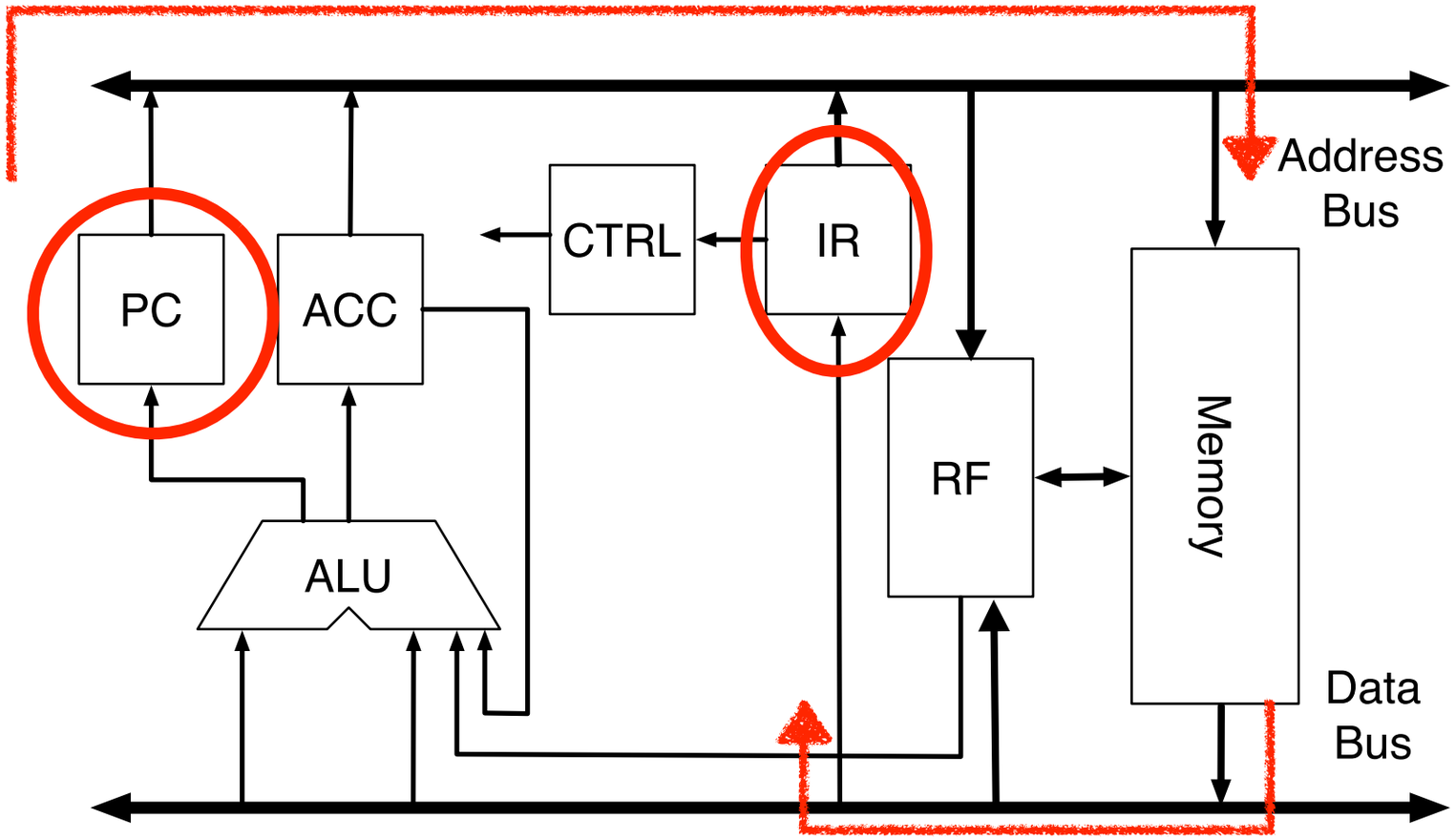

Program Counter
+1
ADD
INSTRUCTION MEMORY
Sign-Ext-7
MUXtgt
OP
rA
rB
rC
TGT
SRC1
SRC2
REGISTER FILE
TGT
SRC2
SRC1
DATA OUT
DATA MEMORY
DATA IN
ADDR
MUXrf
WErf
Left-Shift-6
Sign-Extend-7
WEdmem
MUXalu2
MUXalu1
SRC2
SRC1
MUXpc
CONTROL
FUNCalu
EQ!

3. Instruction execution cycle

Instruction execution model

- A program is a list of instructions that is read and executed one by one in a sequential manner
- The execution model (also part of the von Neumann arch.)
  - ✓ **Fetch instruction** — Current instruction is fetched from the memory, i.e. the opcode is transported to IR
  - ✓ **Decode instruction** — Internal logic interprets the instructions, that is : it generates the appropriate ctrl signals for all associated logic that will be involved in the execution of this operation
  - ✓ **Fetch operands** — Memory read for the data (not always necessary, if data is already in RF — thus we omit this one)
  - ✓ **Execute instruction** — Does the actual operation on the operand
  - ✓ **Write result** — in the memory or in the RF
- Simplified : F, D, Ex, W

1. Instruction fetch

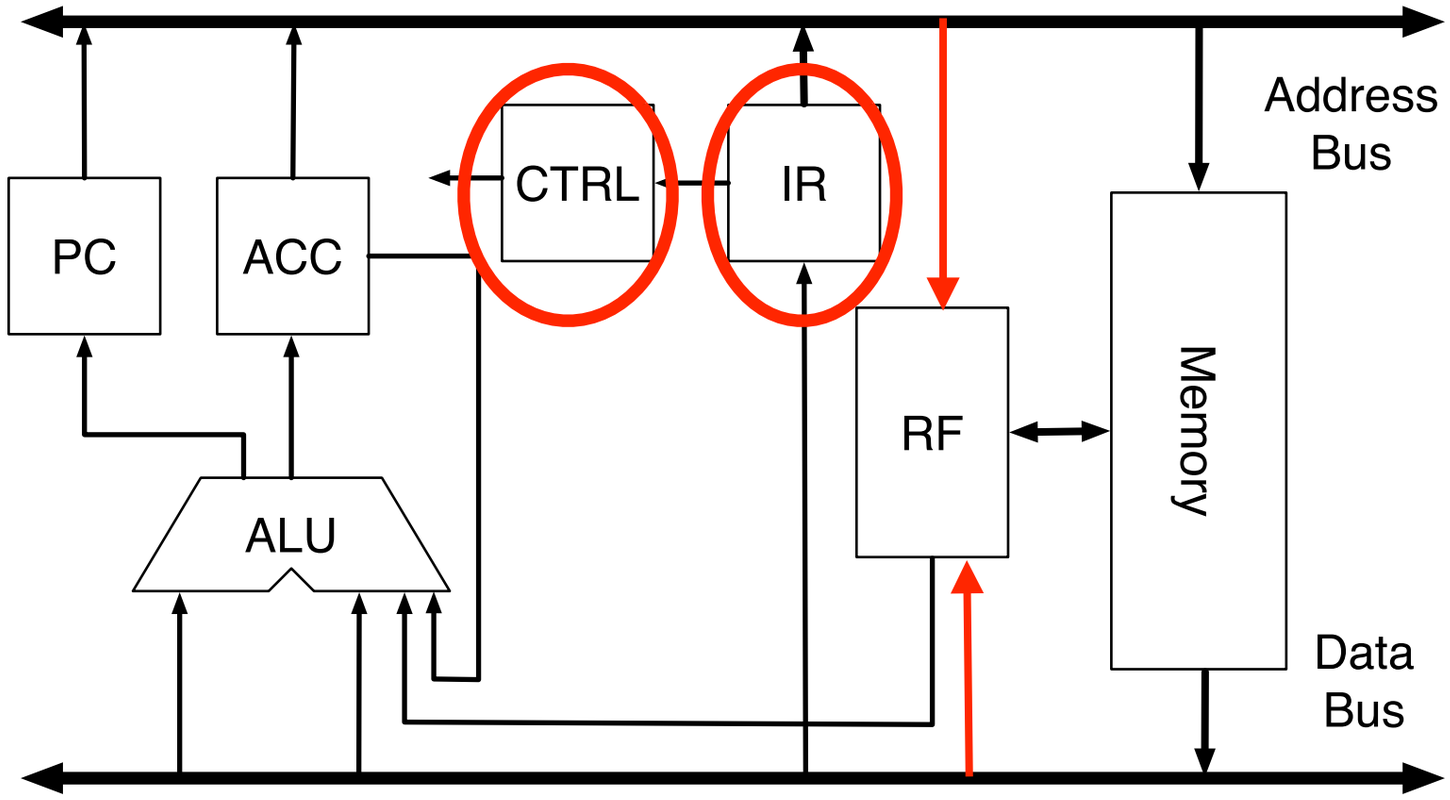


a. CTRL logic places the value of the PC on the ADDRESS bus

b. This generates READ memory access & the data that is read is placed on the data bus; the data read is the opcode of the instruction stored on a given address

c. IR recognises that the data on the bus is for him and takes the opcode value from the bus & stores it for further processing

2. Instruction Decode

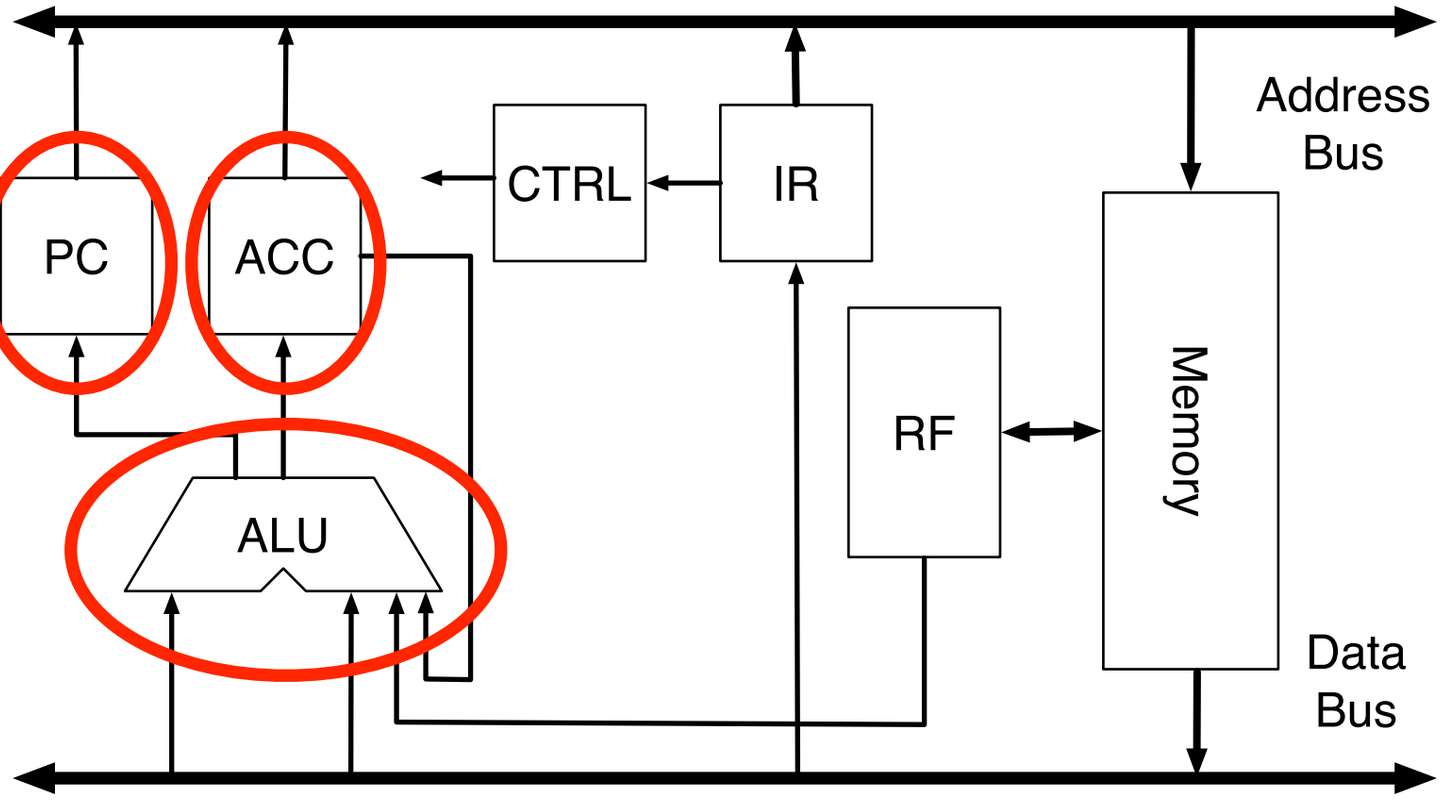


a. CTRL logic takes the instruction from the IR

b. Based on the instruction opcode it generates the appropriate control signals for all other units in the system, namely ALU → operation selection and operands preparation

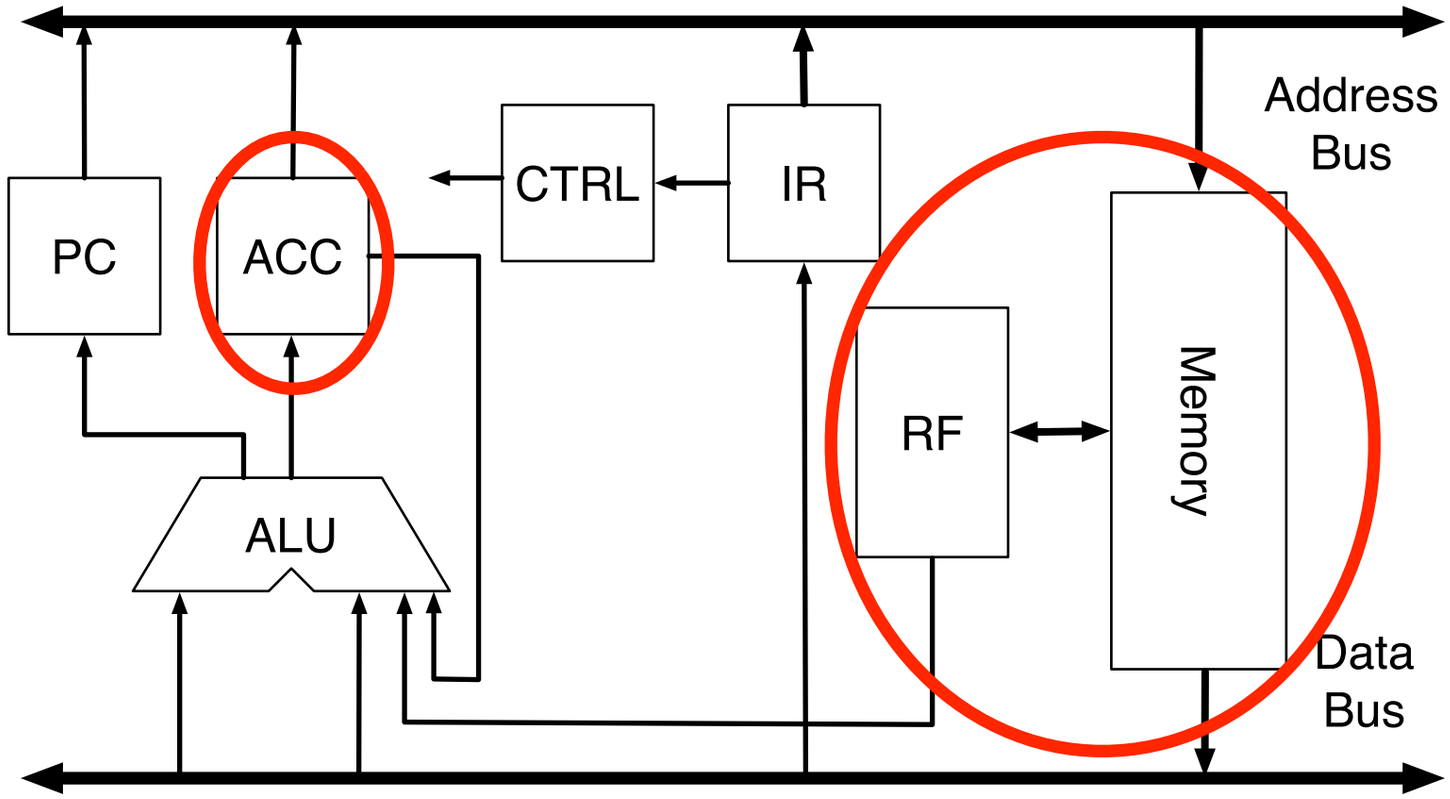
c. In the simplest of its forms it is just a n:m decoder
(can be combinatorial circuit — propose an example of the decoder?)

3. Instruction Execute



- ALU performs the operation (logical AND, arithmetical + etc.)
- The result is written in accumulator
- PC is updated according to the executed instruction (typically PC += 1, but something else is possible depending on the instruction)

4. Write



- Depending on the architecture and the instruction used (specified result destination) the result is written
- It can be RF or memory

Architectures depending on memory R/W

- Depending on what could be a possible combination of the source/destination operands and/or result we can have different type of architectures :

  - ✓ **Load/Store architecture** — operands **must** be in the RF, any computation is always preceded with explicit R/W operation from/to memory to/from RF

  - ✓ **Register/Memory architecture** — operands can be in either in RF **or** memory (this is an exclusive or, both RF and mem is not possible)

  - ✓ **Register + Memory** — any operation can have operands being in memory **and/or** RF

4. ISA

Instruction Set Architecture — ISA

- ISA — the link between the micro-architecture and the programmer (assembly or compiler)
- Until now we defined an *μ*-architecture and the way it operated (instruction execution) but we didn't said anything on WHAT this machine can really do
- WHAT the machine can do, as opposed on HOW, makes ISA :
  - ✓ defines native data types (integer and floating point sizes),
  - ✓ register number (names), sizes and types,
  - ✓ instructions,
  - ✓ addressing modes,
  - ✓ memory architecture,
  - ✓ interrupt and exception handling,
  - ✓ and external I/O
- **Computer = *μ*-architecture + ISA**

Instructions — typical operations

- **Data movement**
  - ✓ Load (from memory) or Store (to memory)
  - ✓ move memory-to-memory or register-to-register
  - ✓ move Input/output (from/to I/O device)
- **Arithmetic**
  - ✓ Integer
  - ✓ Floating point
- **Shifts** : left, right
- **Logical** : and, not, set, clear
- **Control** (jump/branch) : conditional or not
- **Subroutine handling** : call, return
- **Interrupt**

Addressing modes

Not all the modes are always implemented

← used to note the assignment of the result

Addressing mode	Example	Meaning
Register	Add R4,R3	R4 ← R4 + R3
Immediate	Add R4,#3	R4 ← R4 + 3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2] R2 ← R2 + d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Classification of Instruction Sets

- Depending on the way we implement instruction set we can have different (not to say many) **flavors of ISA**
- In computing, most of the time we do the same type of operations (memory move, some basic logic/arithmetic operation etc.)
- The idea behind the classification is to look into number of occurrences of instructions in typical programs
- We can plot the instruction histogram : (and this is not a very representative piece of code)
- How would you plot your own histo?

◦ **Rank instruction Integer Average Percent total executed**

1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

◦ **Simple instructions dominate instruction frequency**

5. RISC vs. CISC

Classification of Instruction Sets

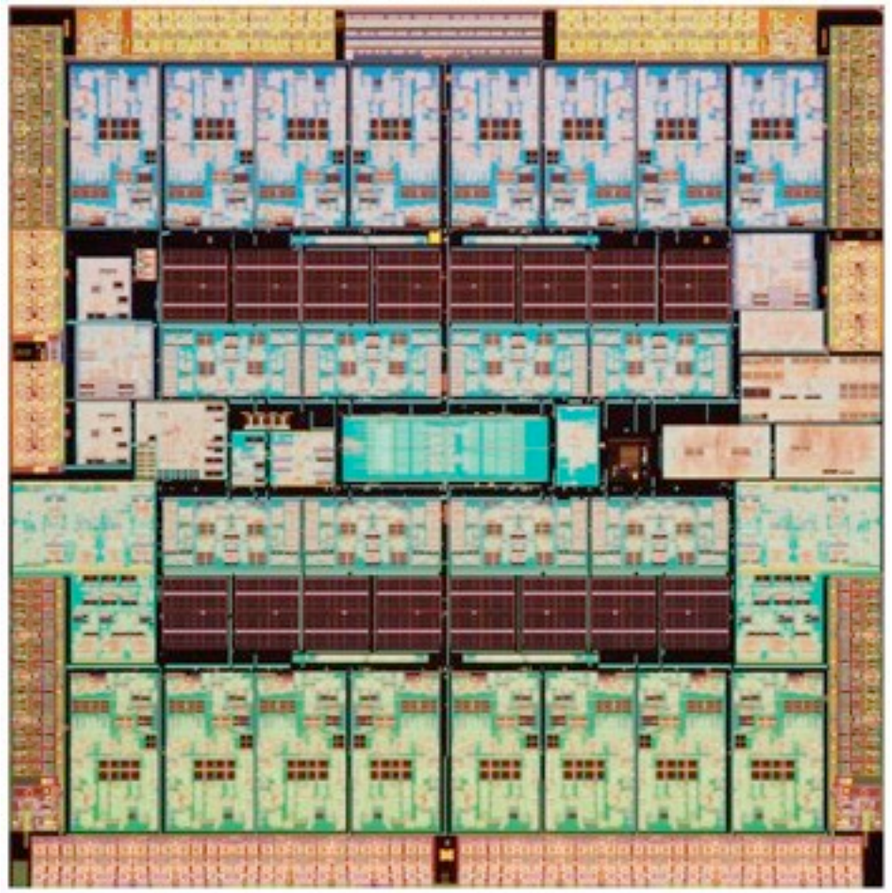
- Conclusion :
  - ✓ **NOT all instructions are used all the time**
  - ✓ Some instructions are used very frequently, some rarely
- This drives the idea of two different approaches to ISA :
  - ✓ **Reduced Instruction Set Computer — RISC**
  - ✓ **Complex Instruction Set Computer — CISC**
- RISC/CISC war is around for quite some time;
- ... in reality both are cool and whether RISC is to be chosen over CISC will depend on application … more general purpose CPU will tend towards CISC architectures (“one architecture fits all applications”)

RISC philosophy

- Instructions are simplified in content and therefore can be implemented more efficiently in HW
- Typically all RISC instructions would require minimum execution time : one cycle
- Complex instructions are made as “subroutines”
  - ✓ This is typically a load/store system : if operation is to be done on memory, it will have to be split into separate load, execute and store instructions
- Efficient instructions will have higher throughput and the system as a whole would be more performant (if 99% of instructions are fast, we do not care about the other 1%)
- Instruction set appears as SMALL but they are HIGHLY optimised

RISC today

- RISC highly popular in academia, but ended-up in the real world
  - ✓ Berkley — Berkley RISC → **SPARC**
  - ✓ Stanford — **MIPS**
- RISC is still on menu:
  - ✓ For high-performance (super-computers) & servers
    - ✦ Sun Microsystems (acquired by Oracle)
      - ✳ SPARC line of CPUs from T1 to T5 (T5 released in 2013) targets servers 1CPU = 16 cores
    - ✦ Fujitsu instance of SPARC architectures — K-computer
      - ✳ most powerful super-computer in 2011
      - ✳ 10 Penta-flops using 80k 8-core processors @ GHz
  - ✓ but also in mobile applications (ARM) for tablets and smart phones ...



SPARC T3

CISC

- As opposed to RISC, CISC has many instructions and some of them can be complex : **instructions are heterogeneous** (in terms of complexity, and hence execution time)

- Example :

  ✓ you could possibly load from memory, compute and store in a single instruction

  ✓ obviously this can not be made in the same amount of time as direct computation from RF, where the operands are ready …

- **Heterogeneous instructions** → in fact means variable N° of cycles to execute (this will have a strong impact on performance, as we will se this later ...)

RISC/CISC today

- It is not black & white : there is no pure CISC architecture, more like CISC/RISC combo

- Take advantage of RISC for a sub-set of instructions, but still have complex instructions around to enable **General Purpose Application** (gaming or scientific computation)

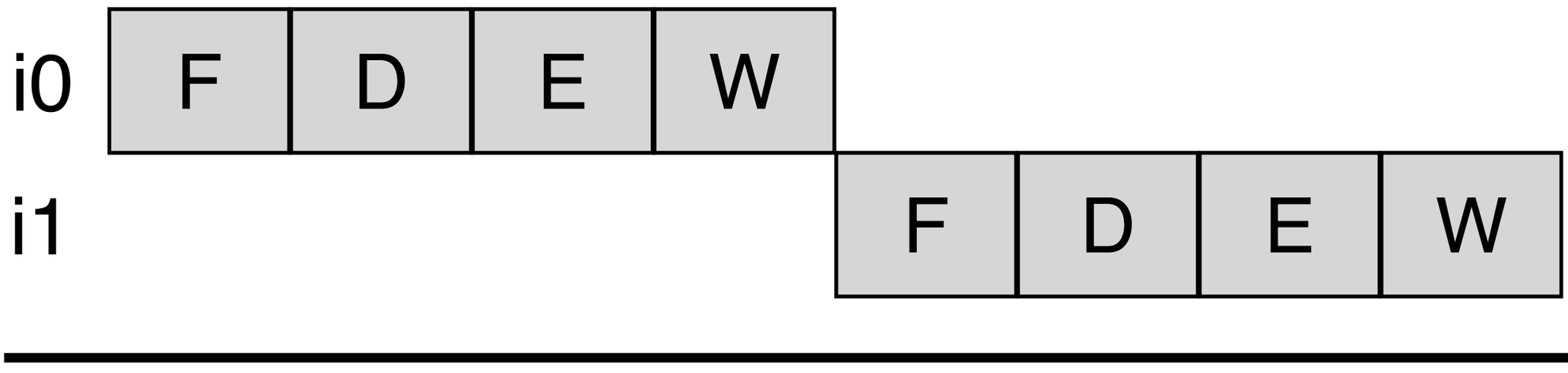
- This is what you will find in X86 processor architectures (Intel/AMD)

  ✓ You should be able to explain why?

6. Instruction execution cycle — physical aspects

How to execute these 4 instruction steps?

- All 4 steps in one big step, so that the next instruction can finish only after the complete instruction cycle have finished?



- Not ideal, because next instruction will have to wait !
  - assuming that each cycle last time t,
    we have to wait for 4 x t before starting next instruction
- This is known as **Single Issue Base Machine**
  (no computer works like this !)

Before going any further ...

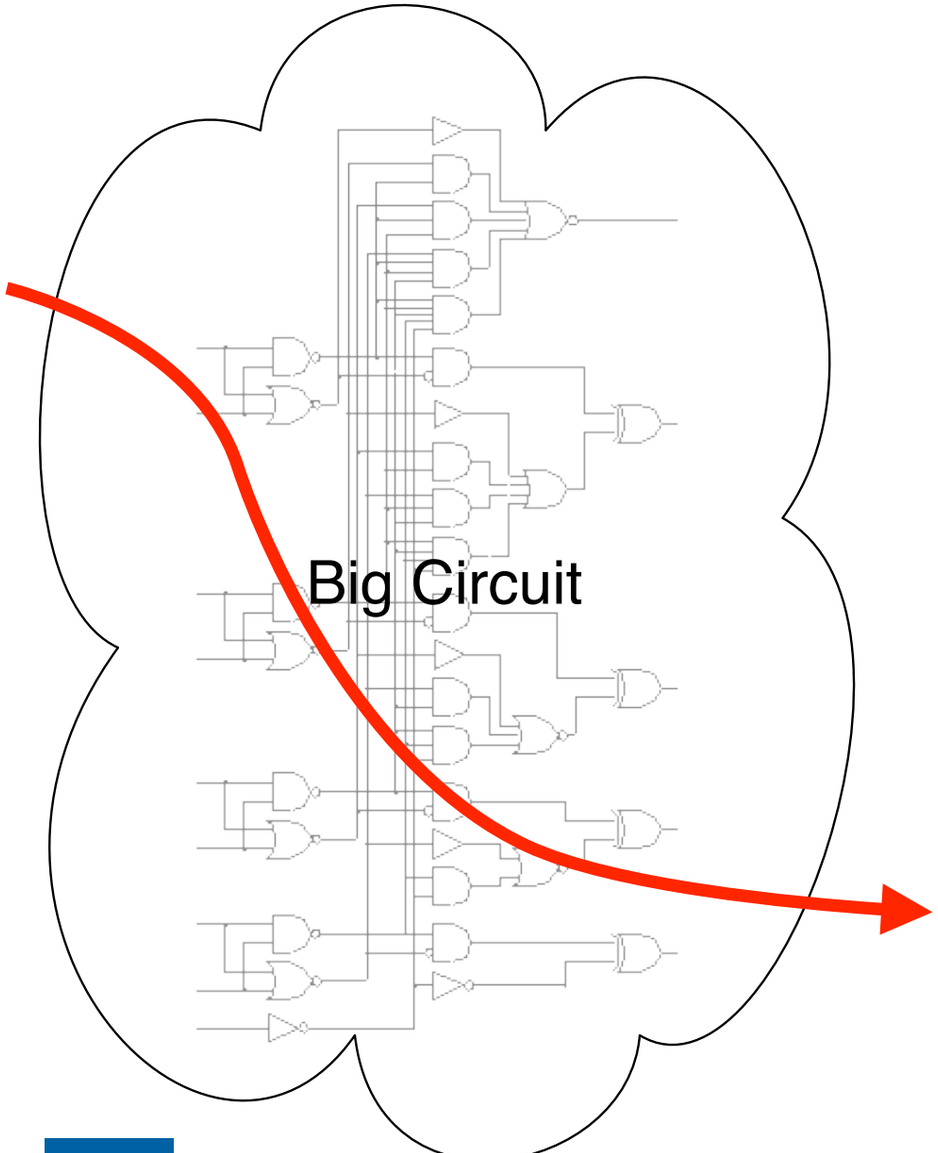
- The absolute value of t will depend on what?

- CPU is a logic circuit (at this stage it doesn't matter if it is combinatorial or sequential)
- Any logic circuit has what we call a **critical path** — the path that exhibits biggest delay (sum of all gate and interconnect delays)

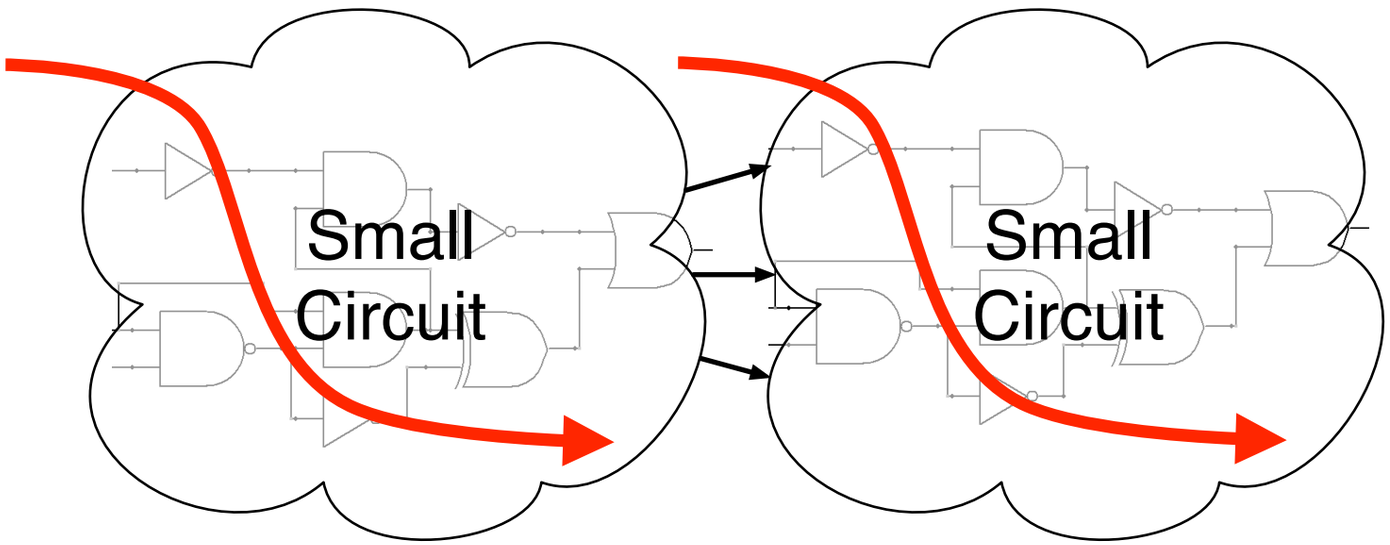
- Bigger the circuit → better chances are that the critical path is bigger (has bigger delay)
- Smaller the circuits → smaller critical path
- Consequence (roughly) :
  - ✓ Small circuit — **FASTER**
  - ✓ Bigger circuits — **SLOWER**

How to reduce critical path delay ?

- The idea is to split the big-circuit into 2 smaller sub-circuits
- If the cut is made in such a way that 1/2 of the critical path is in the left sub-circuit and the other 1/2 is on the right ...



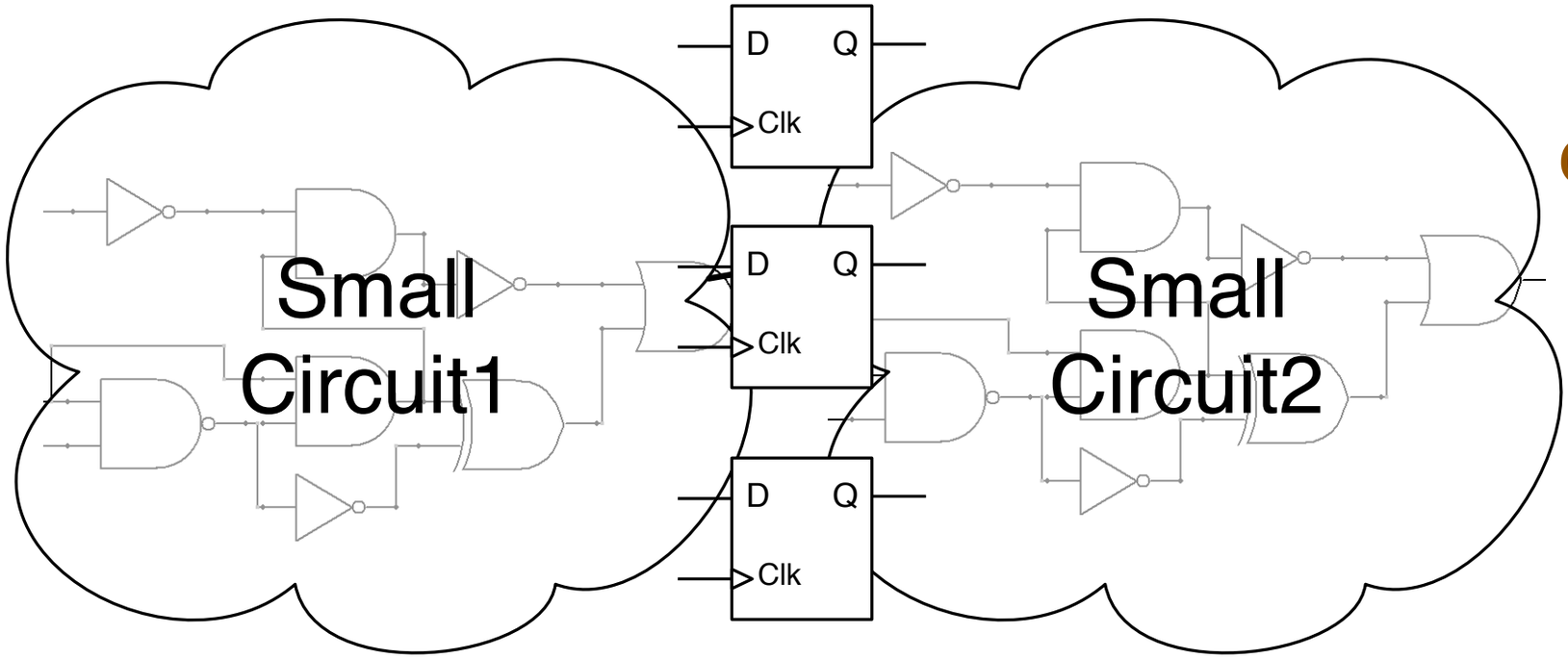
- But does this solve our problem?



Not yet, we need to add something ...

How to reduce critical path delay ?

- For each wire that connects 2 sub-circuits we need to insert a **Flip-Flop** (FF)
- **FF** is a synchronous memory element, driven by a raising (or a falling) edge of a clock
- FF will store the result of the operation of the SmallCircuit1 !

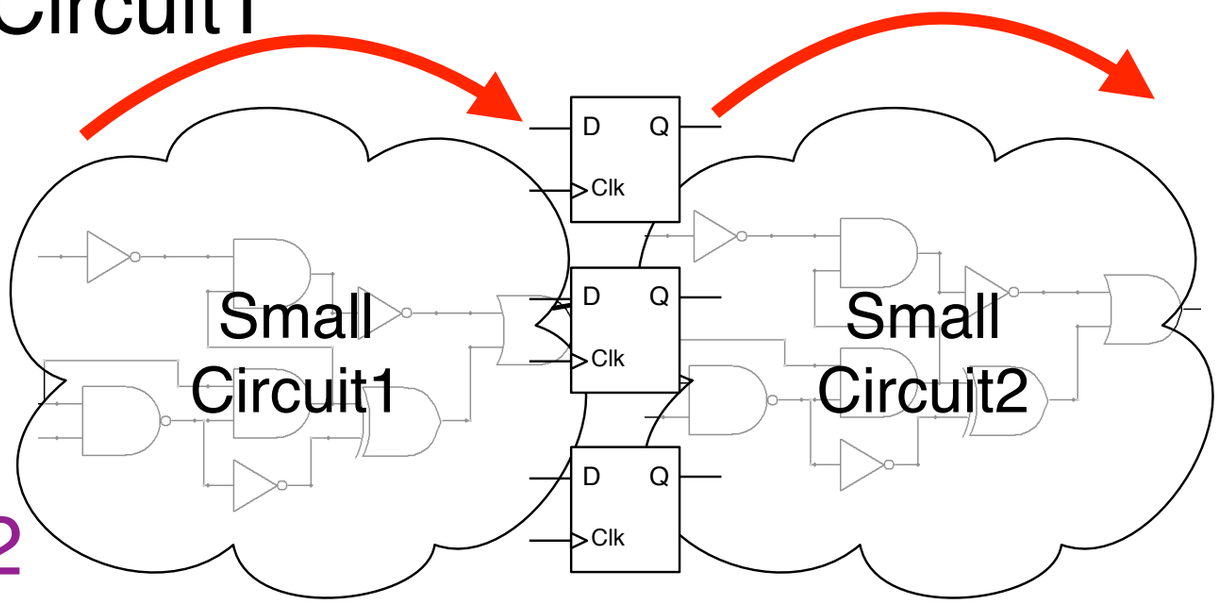


What is the consequence of this?

Two circuits are 1 cycle away.
What does this mean?

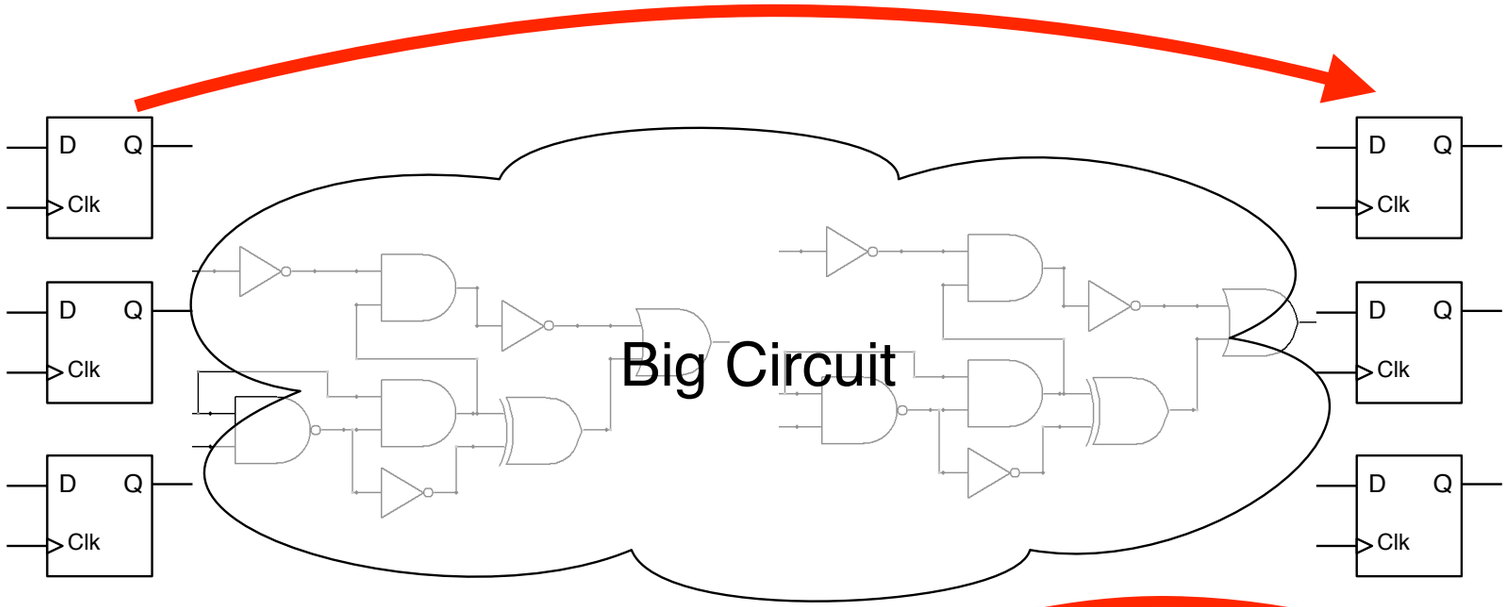
Inserted FFs → Two independent circuits !

- With FFs in between the SmallCircuit1 and SmallCircuit2 can work **independently** (in parallel) !
- Also the **“time distance”** between the two circuits is now 1 Clock cycle
- In time T1 SmallCircuit1 calculates a result out of input data1(SmallCircuit2 has to wait)
- In time T2 SmallCircuit2 will receive on the input the result of the computation of the SmallCircuit1
- SINCE THIS IS STORED IN FF WE CAN DO SOMETHING ELSE IN THE SmallCircuit1
  → start computation on data2



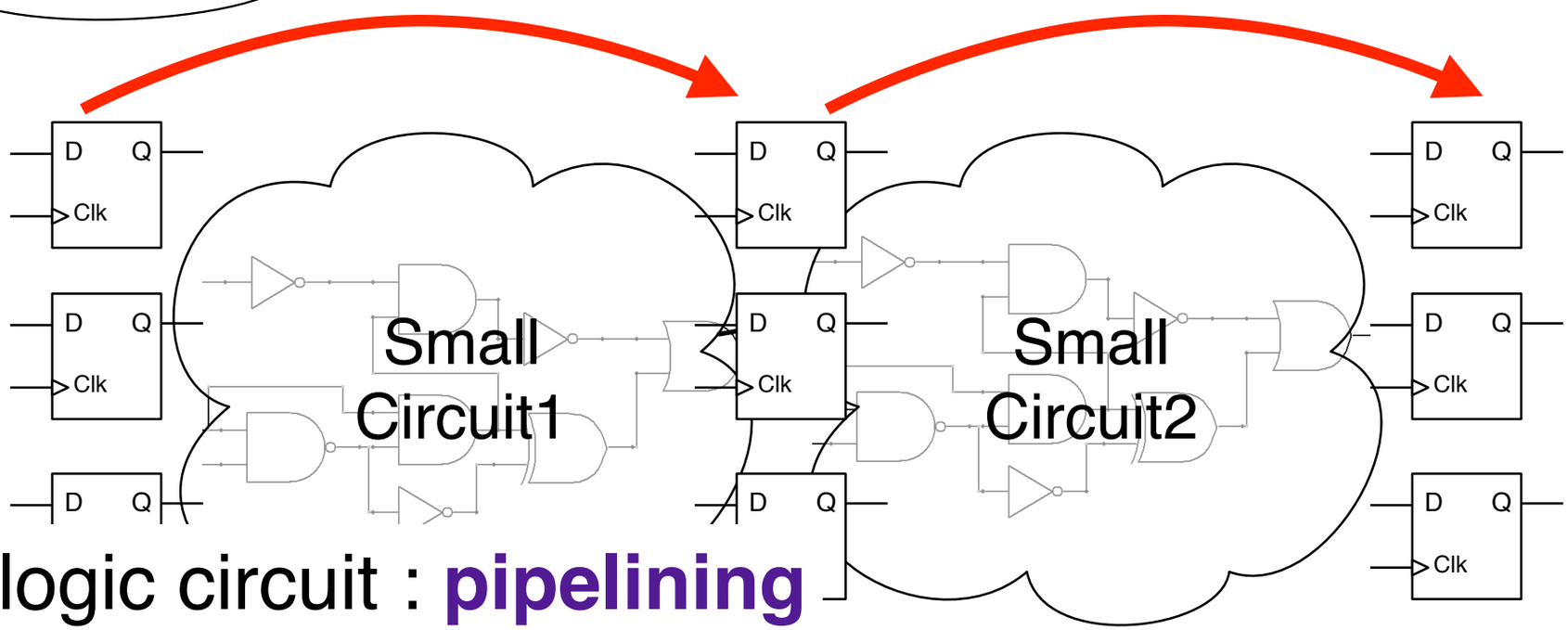
Typical logic circuits are sequential …

- ...maning they have already FFs to avoid race conditions (the operation paradigm is From Register-to-Register)



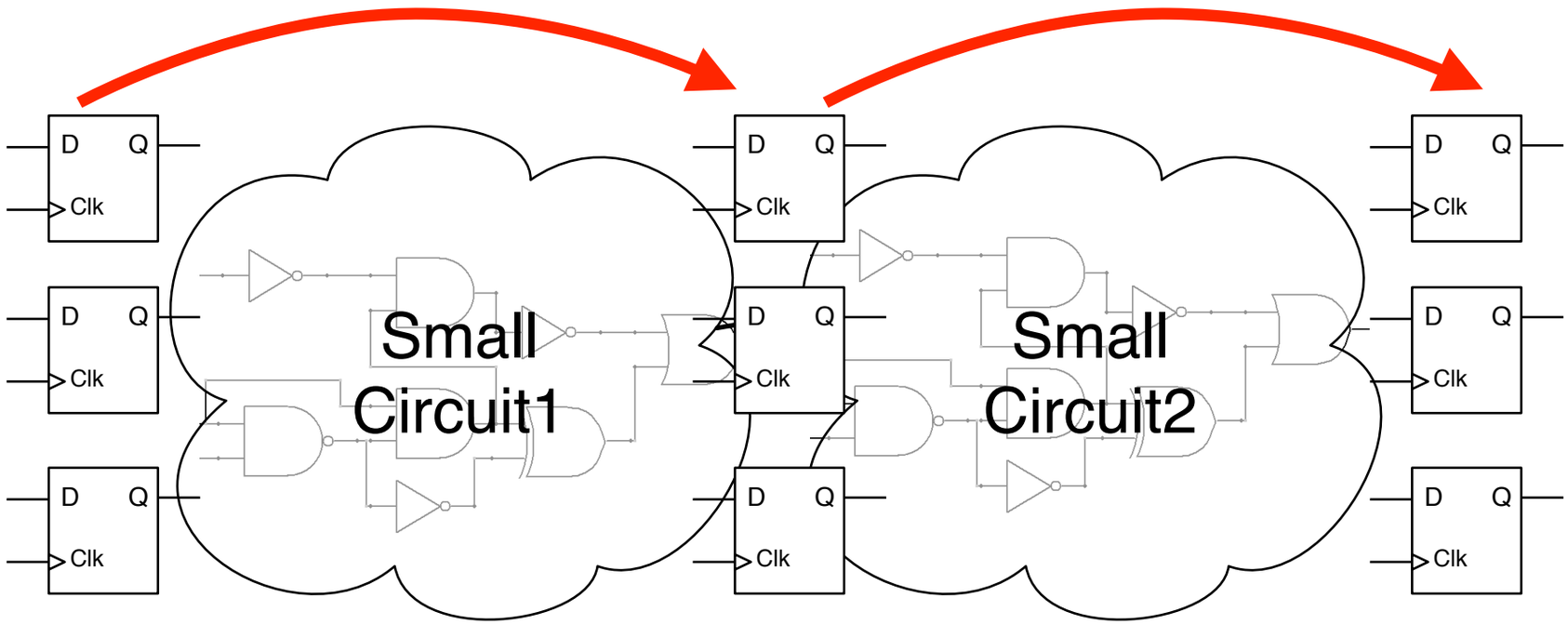
Big circuit, one cycle away ...

Two smaller circuits, two cycles away ...



This is true for any logic circuit : **pipelining**

Pipeline stage insertion : the consequence



When inserting a pipeline stage :

- We hopefully break the critical path & **reduce delay** ☺
  (or increase F; Can you explain the link delay/F?)
- But we do insert one clock **cycle delay** (known as **latency**) ☹

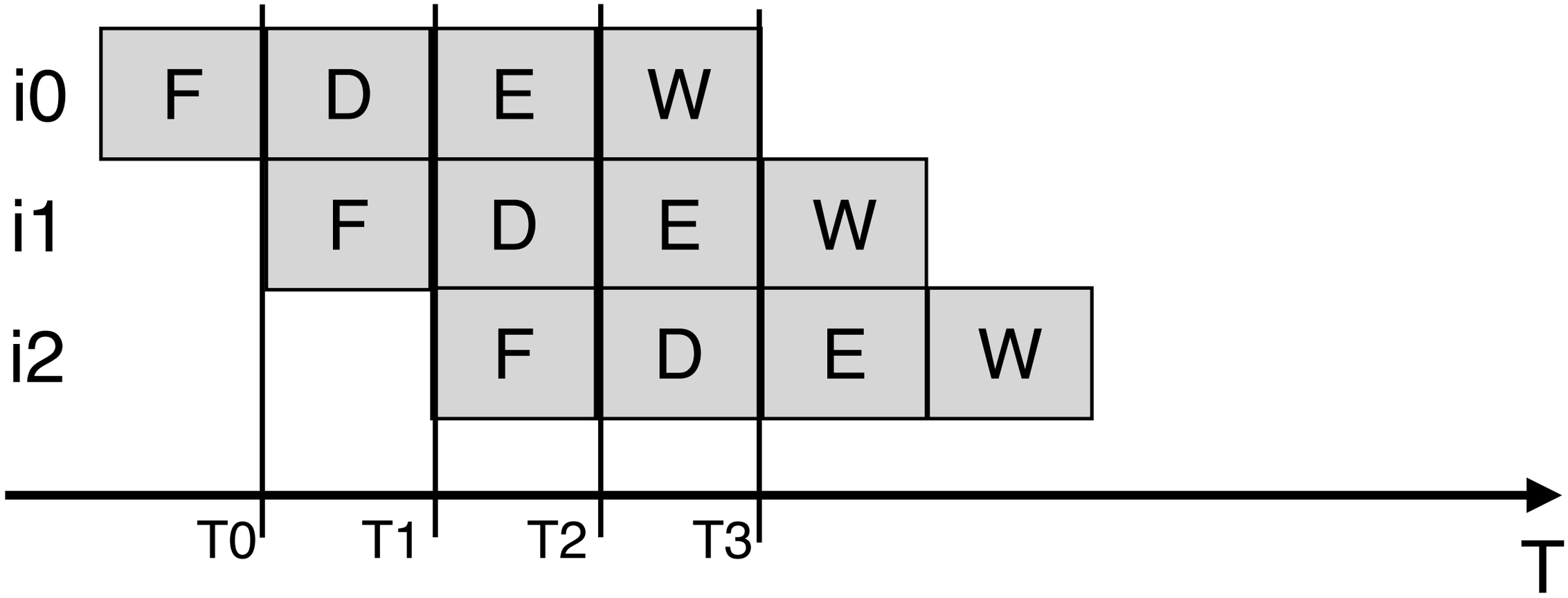
7. Pipeline execution

Pipelining applied to CPU

- Each step of the execution model (F, D, Ex, W) becomes a distinct logic sub-circuit !
- Because F, D, Ex, W are sub-circuits, they are smaller (and therefore faster — hopefully we cut critical path in 4 ...)
- All steps are linked using FF that pass the results from one stage to another

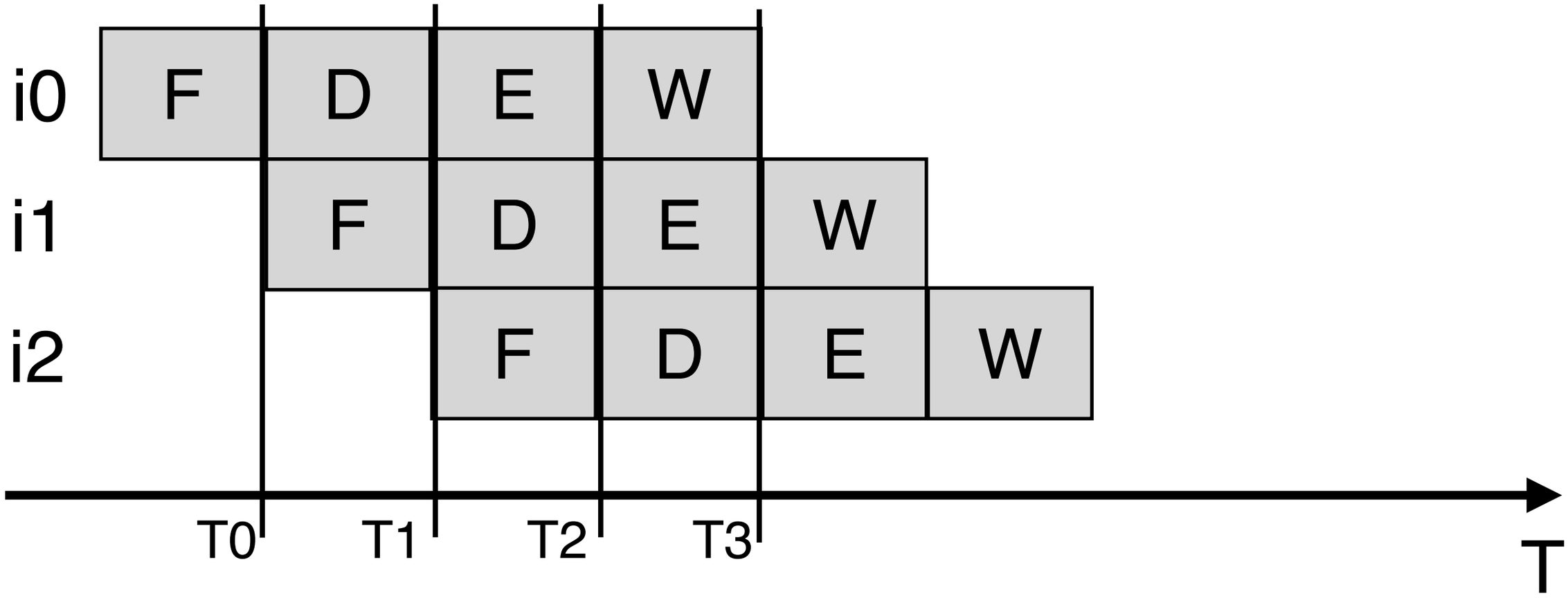
- The latency of the system is therefore **4 cycles** :
  - ✓ One cycle delay per pipeline stage
  - ✓ When instruction enters the “instruction pipeline“, the result will pop-up at the output 4 cycles later
  - ✓ N° of cycles difference between PC on the Address bus and the operand written in the RF or in the memory …

Pipeline Execution



- In T0 — we fetch first instruction
- In T1 — fetch of the i0 is done, i1 can be fetched & i0 decoded
- In T2 — we fetch i2, decode i1 & execute i0
- In T3 — we fetch i3, decode i2, execute i1 & i0 write

Pipeline Execution



- T3 — fetch i3 & i2 decode & i1 execute & i0 write
  (& means that thing work in parallel)
- After T3
  - ✓ On every clock cycle, 1 new instruction is fetched,1 decoded, 1 executed and 1 written (terminated)

Pipeline Speed-up vs. Single Issue

- For N instructions and n pipeline stages, the total cycle count is :
  - ✓ Single Issue Base machine →Cs = N x n x t
  - ✓ n-stage pipeline CPU → Cp=n + N x t
- The acceleration is
  - ✓ Cp/Cn
- If N >> n :
  - ✓ we get 1/n
- **The acceleration is proportional to the number of pipeline stages !**
- What do you do with this?
  → Do we insert infinite number of pipeline stages ?

Pipeline & Super-Pipeline

- What do you do with this?
  → Do we insert infinite number of pipeline stages ?
- **NEVER !**
  - ✓ Each pipeline stage adds an extra-cycle of latency !
  - ✓ Acceleration of n is possible only if you execute lot's of instructions that have to be perfectly equal in length (RISC could potentially use pipeline better)
- In practice : trade-off between
  - ✓ Number of pipeline stages and the size of critical path
  - ✓ Some CPUs are super-pipelined (the number of stages is much, much bigger then the number of instruction execution steps)
    - ✦ X86 heavily pipelined → good marketing argument since F goes up !
  - ✓ Other CPUs are less pipelined ...

Pipeline & Execution model

- Pipeline executions introduces something very special
  → **parallelism** : things happen at the same time
- This particular type of parallelism (pipelining) is one of the few techniques said to exploit : **Instruction Level Parallelism**, i.e. the possibility to execute multiple instructions at the same time (here the overlap is possible between different stages of different instructions)
- By definition we want more of this since it means faster execution (better instruction throughput)
- Pipeline execution is handled automatically for you by the HW of the CPU, you do not see this !
- In the beginning everything was made so that the programers do not have to think about this … but as we will see this is not always the case, especially if you target high-performance apps

Things to take

- Simple 1 ALU based architecture is already somehow parallel
  - ✓ Bit-level parallelism (word/operand level)
  - ✓ Pipeline for F, D, Ex, W operation of the instruction
- Pipeline :
  - ✦ enables parallel execution of instructions
    → higher instruction throughput
  - ✦ acceleration proportional to the number of pipeline stages
  - ✦ but trades-off throughput with latency
  - ✦ enables higher clock rates, because of the smaller critical paths of corresponding
- Today's machines : modified Harvard architecture, a blend of CISC/RISC that more or less pipeline the execution of the instructions