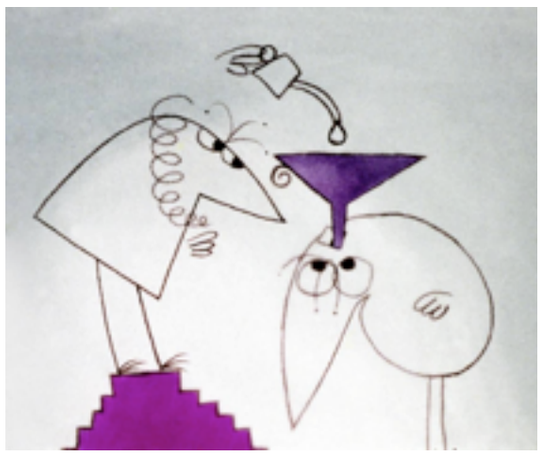
ELEC-H-473

Microprocessor architectures



Lecture 04

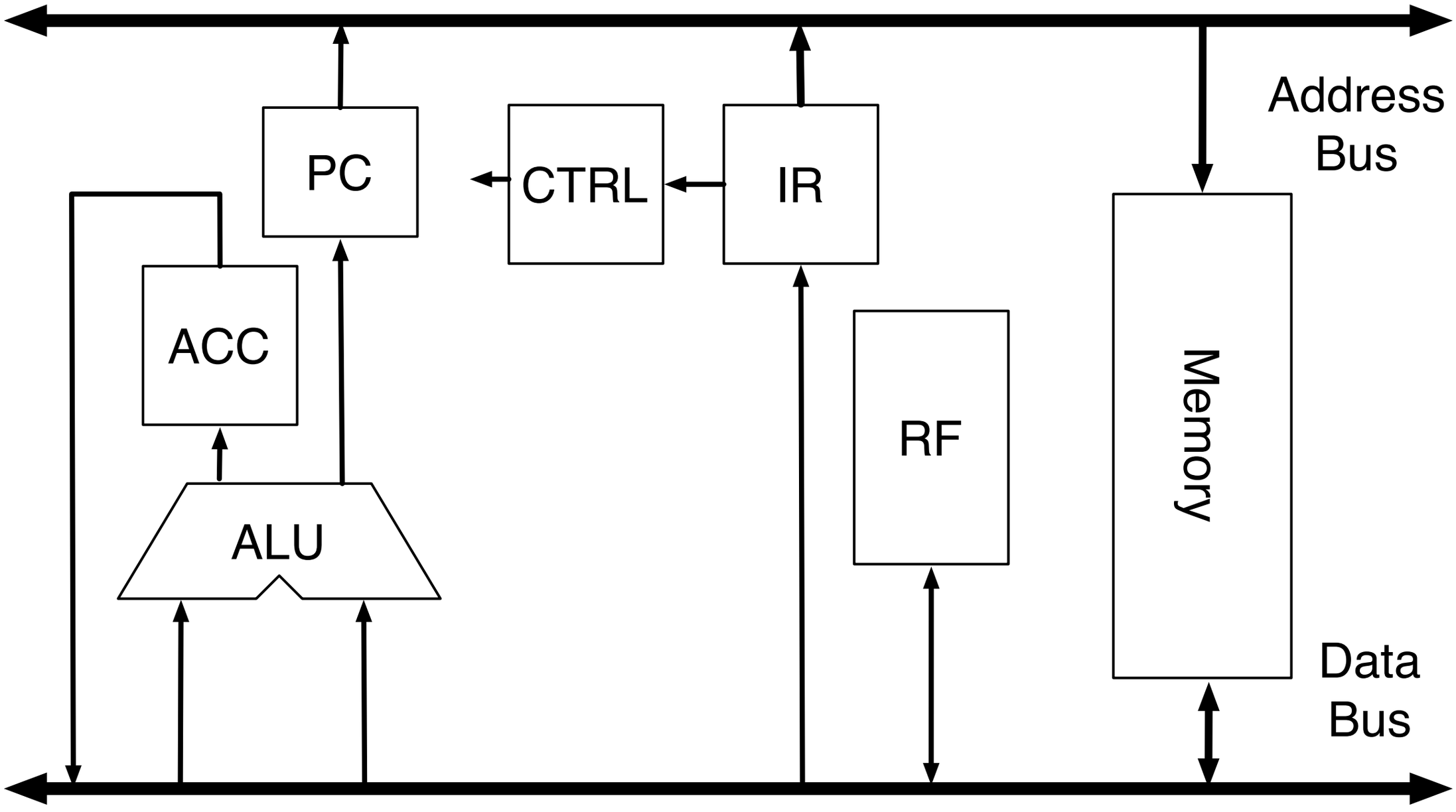
Dragomir Milojevic

dmilojev@ulb.ac.be

Previously on ELEC-H-473

- Basic architecture : combination of Von Neumann and Harvard architecture
- Instruction execution model — different views :
  - ✓ **Programmer** — F, D, Ex, W
  - ✓ **IC designer views** — Critical path
- Pipelined execution trade-off :
  delay reduction (frequency increase) vs. latency
- Latency depends on the pipeline depth (i.e. the N° of pipeline stages)
- CISC/RISC approach to ISA

Minimum system architecture



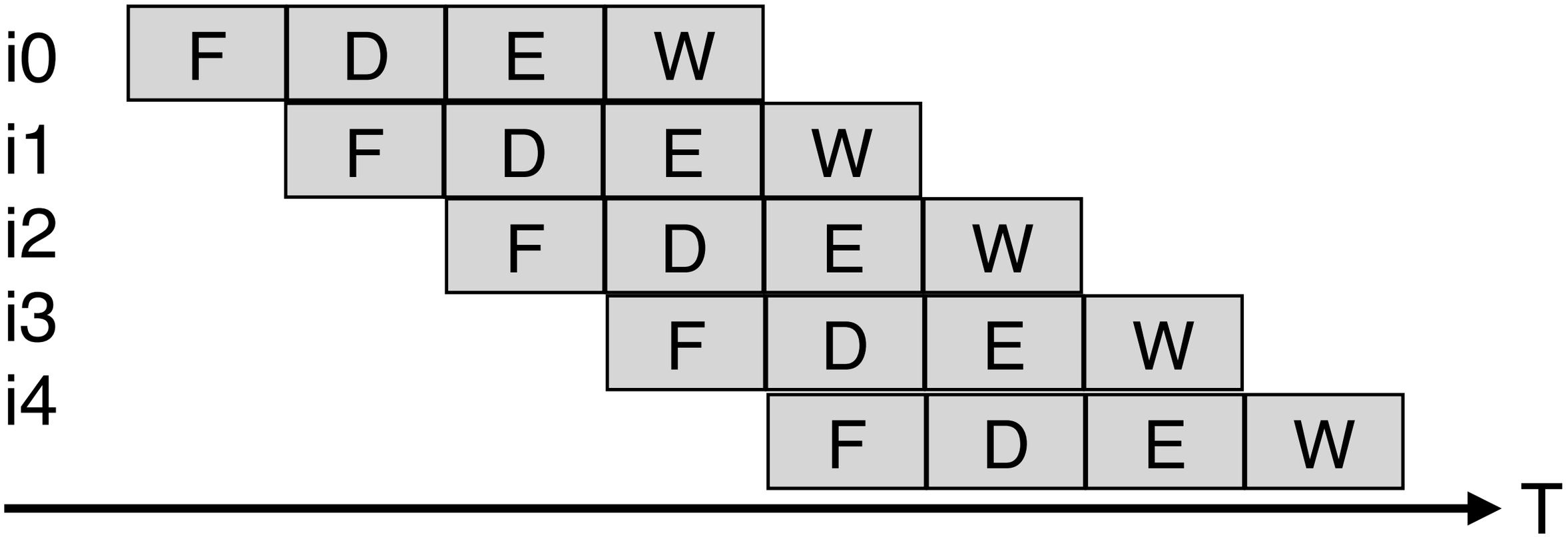
Simplified instruction set

- We will introduce few assembly instructions necessary to write some pseudo-code necessary to illustrate different concepts i.e. we add **Instruction Set Architecture** to the minimum system architecture
- Instructions: few basic instructions, more as we move along
  - ✓ **Data movement** – from memory-to-register (and inverse!) and from register-to-register
    - ✦ `MOV dst , src`
  - ✓ **Arithmetic**
    - ✦ `ADD, DIV, MUL` — simple + , / , *
- Register file – set of registers of a certain size with names
  - ✓ AX → JX (10 registers of whatever word size)

Today

- Pipeline hazards
- Name dependency
- Data conflicts
  - ✓ Out-of-order execution
- Branch conflicts
- Resource conflicts
- Loop unrolling

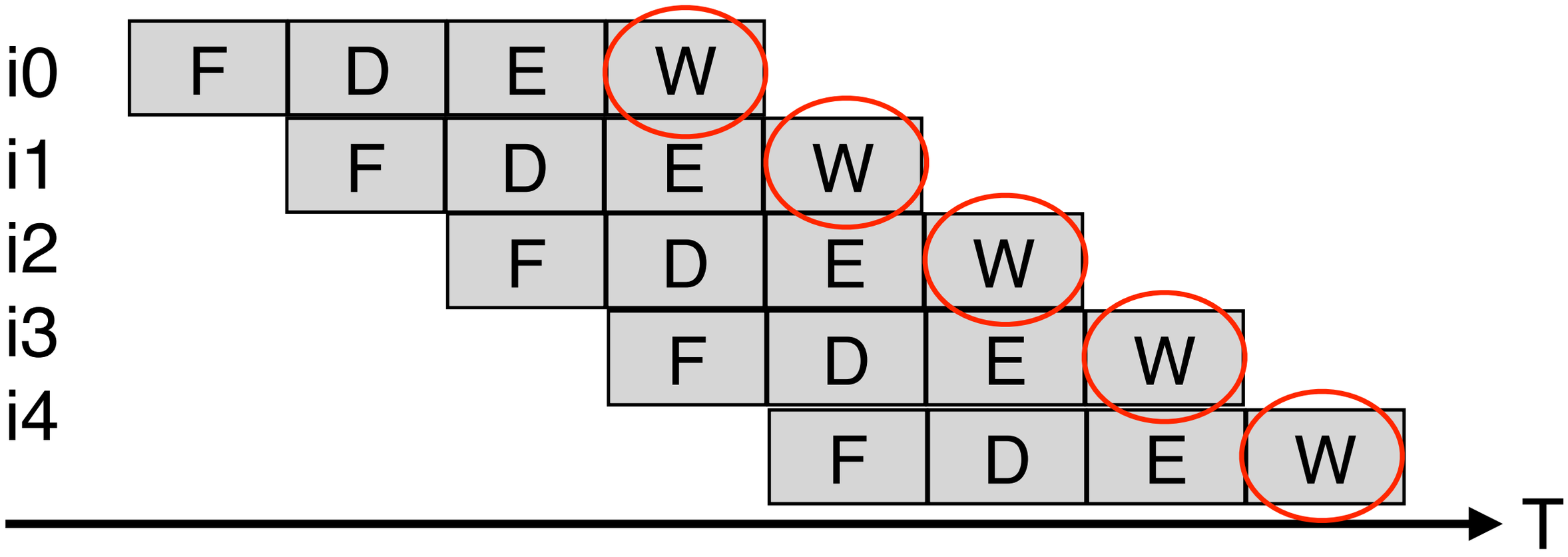
Pipeline acceleration



- Speed-up proportional to N° of pipeline stages is possible only if :
  - ✓ all stages take the **same**, fixed, **predictable** amount of **time**
  - ✓ there is no **dependency** between consecutive instructions
- **These hypothesis are EXTREMELY optimistic !!!**
- If this happens in the real world, then we are :
  - ✓ either very lucky ... **or we understand computer architecture well !**

Pipeline acceleration

- Even if multiple *μ*-ops executed in parallel, at their best, the pipeline can achieve an IPC of 1
  - ➡ 1 instruction could be done for each system cycle



In the real world ...

- What happens if, for whatever reason, we can not maintain the pace of the ideal pipeline execution ?
  (remember C.Chaplin in "*Modern Times*" :
  he is too tired to follow the others
  in the pipeline and misses his turns …)

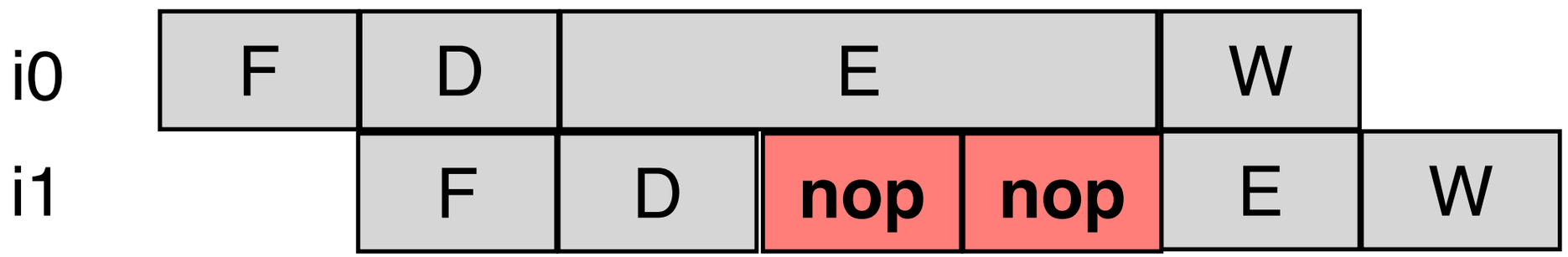
- Pipeline chain is broken, normal operation is altered …
- This is called :
  - ✓ **Pipeline stall**
  - ✓ **Broken pipeline**
  - ✓ **Bubbling**
    (a bubble enters the pipeline)

Bubbling in computer architecture

- This happens because we have **execution hazards**
- Execution hazard will occur :
  - ✓ **When instructions do not take the same time**
  - ✓ **When some unpredictable situation happen**
  - ✓ **When there is computational dependency between instructions**
    Often, there is dependency between instructions
    → after all the computing is :
    "*About how do we serialize the computations ?*"
    (cf Turing machine)
- Example :
  - ✓ Arithmetic operation currently in execution reference memory location (instead of register)
  - ✓ Since operand fetch from memory will take more cycles, the following instruction need to wait the completition of the first

Pipeline stall — the execution mechanism

- How the CPU handles the pipeline stall :
  - ✓ Instruction is fetched
  - ✓ Control logic determines if there is a potential hazard
  - ✓ If this is the case, the control logic inserts **NOPs** (AKA wait state) in the instruction execution pipeline of the following instruction → that **waits** the fist one to complete !
  - ✓ The number of NOPs is proportional to the difference in execution time
  - ✓ If the number of NOPs > then the N° of pipeline stages, the pipeline is fully emptied (it is said that the pipeline is **flushed**)



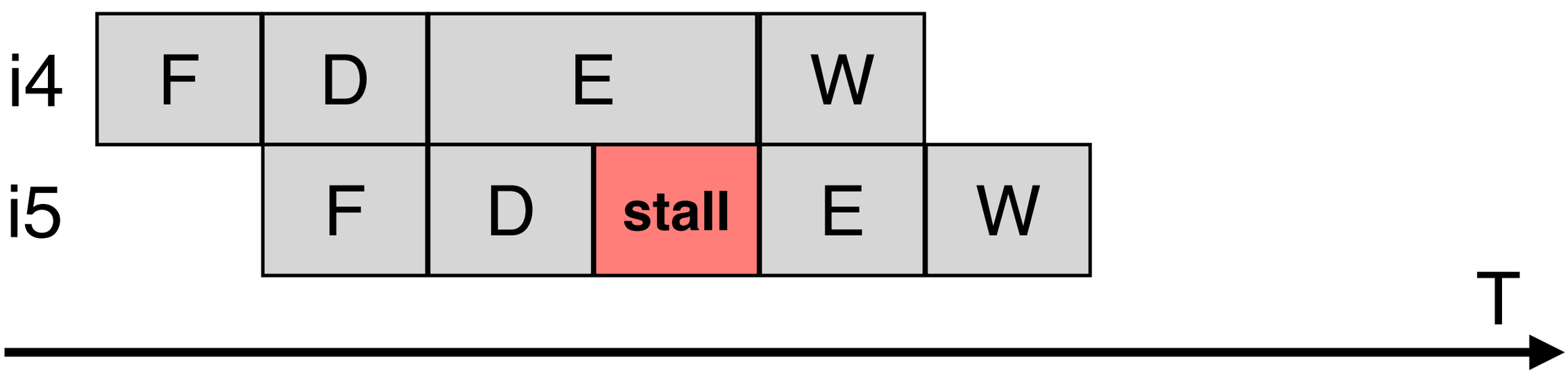
Bubble example : instruction exec time diff

- Let's tray to compute the following :

  `C=(A/B) + C`

- It is reasonable to believe that the arithmetic division (instruction 4.) is going to take more time then addition (instruction 5.), if there is no HW divider
- If these two instruction are to be pipelined: addition will have to wait until the division ends first !

```
--
1. MOV     AX, A
2. MOV     BX, B
3. MOV     CX, C
4. DIV     AX, BX
5. ADD     AX, CX
```

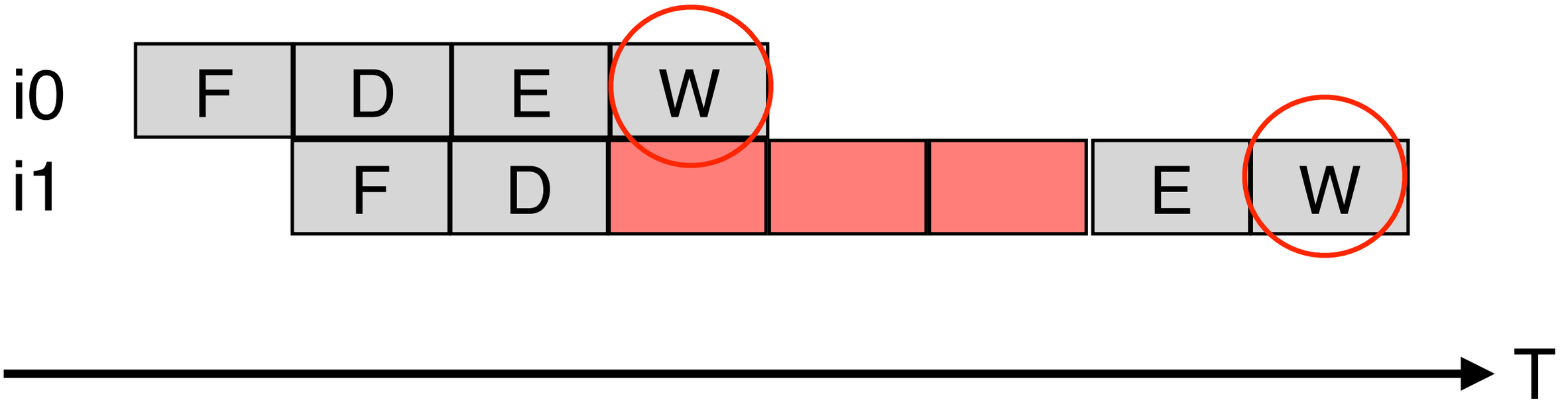


Pipeline acceleration when bubble

- In real world IPC < 1, because of the stalls (you can not get a WRITE on every system cycle)
- The question is :

  ✓ How we can improve this?

- First thing to we need to understand are the pipeline stall causes ...



Possible causes of the pipeline break

- You can possibly enumerate all hazard causes based on computer architecture and programming models :

  - ✓ **Name dependency**

  - ✓ **Data dependency**

  - ✓ **Control dependency**

  - ✓ **Resource conflicts**

Name dependency

Name dependency

- **RegFile** — limited set of registers storing operands
- We say that there is register **name dependency** if the instruction operands target the same names of src/dst registers, but if there is no actual data flow between the instructions !
  - ✓ Anti dependence — instructions i+1 writes to register used by i; the good order needs to be preserved
  - ✓ Output dependence — instructions i, i+1 write to the same output register :
- Name dependence is not true dependence, there is no real computational sequence between instructions (false dependency)
- If this is the case, the register names can be changed any time!

Register renaming — avoid unnecessary serial reuse of registers due to their limited number
Propose an example ?

ISA vs. Physical registers

- Register File — multi-ported SRAM of a limited size that trade-off size for fast access time → **physical registers**
- These registers can be mapped to ISA, or **architecture registers** (e.g. AX, BX, CX etc.)
- The mapping can be done in 1-to-1 fashion, once for all (hardwired in HW) :
  for each ISA register there is one corresponding physical register
- Or, we can do differently :
  - ✓ ISA registers map to physical registers using **dynamically** allocated table (Register Alias Table — RAT)
  - ✓ So program access physical registers through RAT
  - ✓ If this allocation is done smartly, you can do register renaming online

Who can do register renaming ?

- **Statically** — so, not at run time but during programming or compile time :
  - ✓ Programmer — when allocating registers for operands
  - ✓ Compiler — when translating high-level (C/C++) code into assembly
- **Dynamically** — at run-time, during execution
  - ✓ CPU — looks for the register renaming opportunities
- Dynamic register renaming can work out quite well, but the good idea in general is to check if this is done

Register renaming was possible because there was no data dependency, but what happens when we have one ?

Name dependency — Example

```
--
1. MOV     CX, [Mem1]
2. ADD     CX, BX
3. MOV     CX, [Mem2]
4. ADD     CX, BX
```

```
--
1. MOV     CX, [Mem1]
2. ADD     CX, BX
3. MOV     JX, [Mem2]
4. ADD     JX, BX
```

Both additions target the same destination register CX :
there is no possible overlap of instructions
→ poor possibility for pipelining

Second destination register is modified to target another free RegFile physical slot : possible overlap of instructions → pipelining

If done online the initial code could still contain reg names on the left, the change will happen on the fly

Data dependencies

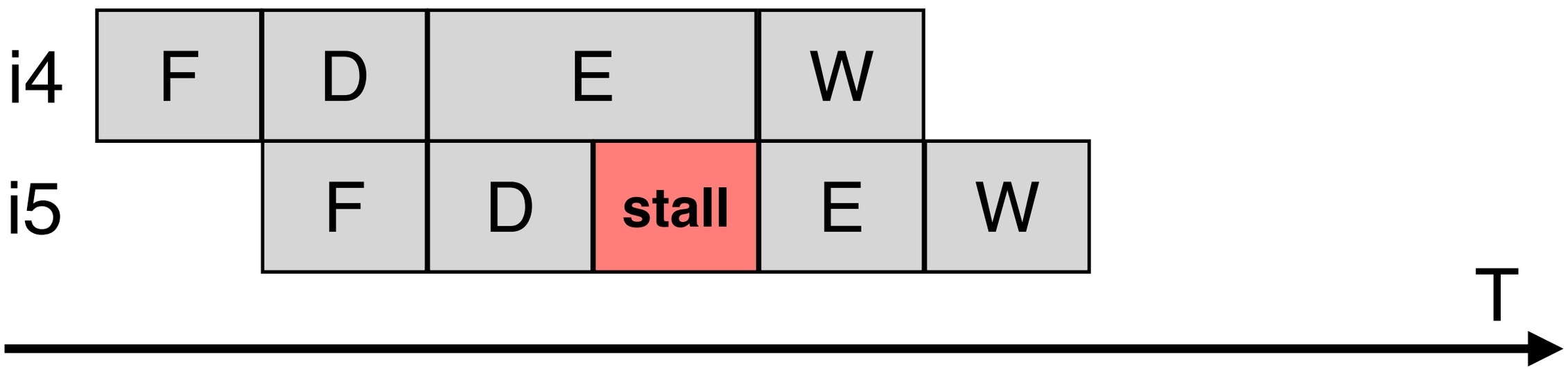
Example of data dependency

- Let's tray to compute the following :

  `C=(A/B)` and `E=(A+D)`

- Let's suppose that the arithmetic division (instruction 4.) is going to take more time then addition (instruction 5.) — this might not be true for high perf CPUs today ...
- If these two instruction are to be pipelined: addition will have to wait until the division ends first !

```
--
1. LDA     CX, A
2. LDA     BX, B
3. LDA     AX, C
4. DIV     AX, BX
5. ADD     AX, CX
6. MOV     D , AX
```



Types of data hazards

- Read after Write (any pipeline is subject to this)

```
i1. ADD     AX, CX
i2. ADD     BX, AX
```

- Write after Read — `i2` writes before `i1` uses it

```
i1. ADD     AX, CX
i2. ADD     CX, AX
```

  (this could happen if multiple ALUs)

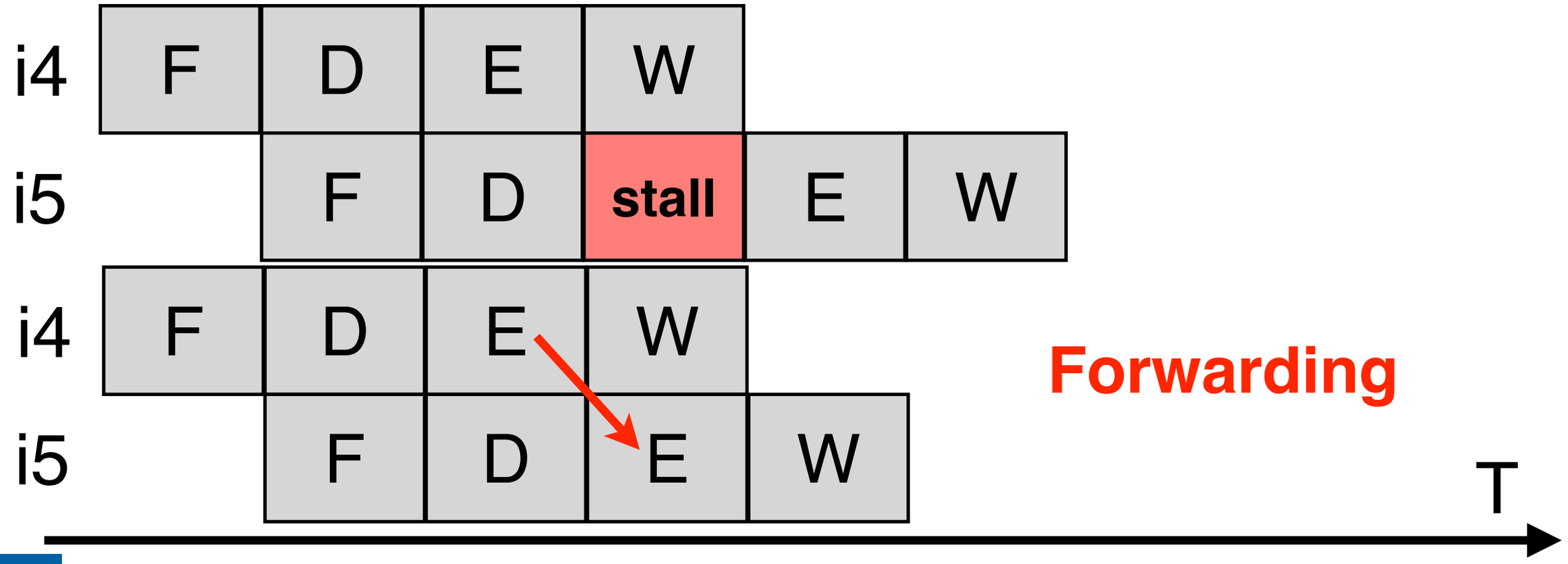
- Write after Write — both instructions write to same destination

```
i1. ADD     AX, CX
i2. ADD     AX, BX
```

  (this could happen if multiple ALUs)

Simple solution

- The same is valid even if the i4 took only one cycle, if the data needs to be written to the RegFile before we can use it
- In that case i5 needs to wait for data to be written to RegFile (execution can proceed only after W stage)
- Or computed data is already available after E of i4 : so if it can be passed directly to the i5 we do not need to wait for W of the i4



Instruction dependency

- Two instructions are **independent** if they can **execute simultaneously** (if we have multiple ALU: e.g. one general and one dedicated)
- If the instructions are independent they will never cause a stall in the pipeline of an arbitrary depth, if executed one after another
- The idea then is to put as close as possible, and as much as possible independent instructions together !
- On the opposite : you want to put data dependent instructions just enough far away so that the result is available when needed
  - ✓ This means that there exist order of the instruction execution that will maximize the instruction throughput and minimize pipeline stalls
    → **this sounds like instructions scheduling** !
- Similar thing happens at another abstraction level — OS and task scheduling—, but what is different is the granularity and the time we have to schedule instructions (ms vs. ns time frames)
  What is the impact of the small amount of time?

Who does instruction scheduling ?

- Proposed minimal architecture follows the execution sequence imposed by the program itself : this is called **in-order** execution
- For in-order architectures, correct scheduling,that will depend on the ISA, can be done by the **compiler**, or by the **programmer**
- The drawback is that both need to be ISA aware (compiler/ program are then architecture dependent and will not run elsewhere with the same perf)

- Idea : make a HW block that will perform **out-of-order** execution
- At run-time, this block will try to pack instructions in such a way to minimize potential pipeline stalls
- Advantages : compiler/user independent
- Disadvantages : extra area cost, complexity, low run-time decision cycle (or general scheduling problem is complex) → even though automatic, should be checked

Out-of-order (OoO) example

```
--
DIV R2, R1
ADD R3, R2
MUL R4, R3
SUB R8, R7
SUB R10, R9
SUB R12, R11
```

First 3 instructions : data dependent, no scheduling opportunity on DIV, ADD, MUL

But SUBs are independent !

One of the SUBs could be executed earlier …

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	Fetch	Decode	Execute			Write		
Add R3, R2		Fetch	Decode	Wait		Execute	Write	
Sub R8, R7			Fetch	Decode	Execute	Write		
Mul R4, R3				Fetch	Decode	Wait	Execute	Write
Sub R10, R9					Fetch	Decode	Execute	Write
Sub R12, R11						Fetch	Decode	Execute

OoO : How it is done ?

- Make a window of instructions and retire the instruction from the window whenever the operand is available :
  - ✓ Instruction fetch
  - ✓ Instruction dispatch to an **instruction queue**
  - ✓ The instruction waits in the queue until the **input operands are available**
  - ✓ The instruction is then allowed to leave the queue before earlier, older instructions
  - ✓ The instruction is issued to the appropriate execution unit and executed by that unit
  - ✓ The results are queued
  - ✓ Only after all older instructions have their results written back to the register file, the end result will be retired

Control dependency

It is useful to alter instruction flow

- Occurs whenever we have a branch in the instruction flow
- Branch:
  - ✓ Unconditional — equivalent to *goto* statements
  - ✓ Conditional — based on Boolean value of an arbitrary algebraic or logical expression (false, true)
- At branching point the control flow is altered, and we start executing another instruction sequence
- Let's add this feature to our ISA
  - ✓ Jump
    - ✦ `label` — marks the place in the code
    - ✦ `JZ, JNZ` — jumps if equal to zero or if not equal to zero to the label

```
--
1. LDA    BX, B
2. label :
3.    ...
4.    SUB    BX, 1
5.    JNZ label
```

Why programs avoid unconditional jumps ?

- Uncontrolled usage of unconditional jumps create
  - ✓ unreadable code, difficult to debug and possibly inefficient
- In literature known as spaghetti code
  (because you can not figure out which part of the spaghetti goes where)

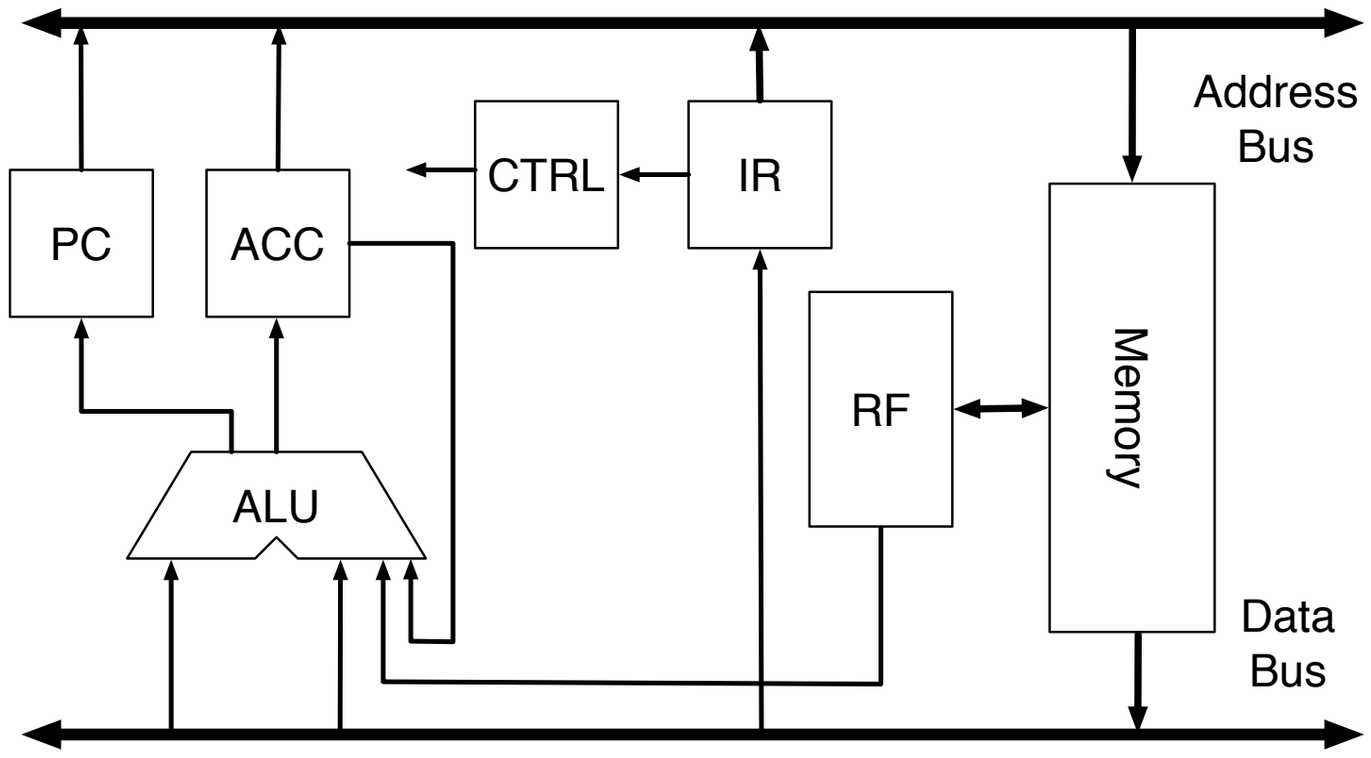
- **Structured programming**
  - ✓ Force usage of functions, not jumps
  - ✓ Enables better readability
  - ✓ C/C++ against BASIC

Typical branching structure

```
if (a==35) then
    /* instruction sequence1*/

/* normal sequence*/
```

- Depending on the condition :
  - ✓ Branch is taken or
  - ✓ not taken
- If the branch is not taken, then we do not have to change PC
- If yes the PC will have to update the value



Impact of the branch on pipeline execution

- Branching has a very strong impact on the pipeline execution obviously …
- We need to compute condition (using general purpose ALU) and then look into the result of this computation
- Next instruction **can not be fetched before** we know which of the two paths we will have to follow !

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	**Fetch**	**Decode**	**Execute**			**Write**		
Add R3, R2		**Fetch**	**Decode**	**Wait**		**Execute**	**Write**	
Branch			**Fetch**	**Decode**	**Wait**		**Execute**	**Write**
Instr 4								**Fetch**

Waits →

Condition statistics

- Important question is :
  How often the condition is TRUE or FALSE ?
- Three possible situations :
  - ✓ Often TRUE
  - ✓ Often FALSE
  - ✓ 50/50

- Depending on HW and the actual statistics on a==35 you could write your code differently
  - ✓ Alternative
    (a != 35)

```
if (a==35) then
    /* instruction sequence1*/

/* normal sequence*/
```

Solution 1

- We need to compute condition (one ALU remember) and then look into the result of this computation
- Next instruction can not be fetched before we know which of the two paths we will have to follow !
- The simplest scheme to handle branches is to **flush the pipeline**, holding or deleting any instructions after the branch, until the branch destination is known
- Simple both for hardware and software, but as you would expect not that efficient...

Solution 2

- Treat **every branch will not be taken** and allow HW to continue execution as if there was no branch
- The pipeline looks like there is no branch
- If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address → **pipeline is flushed**
- This is a bet : either we succeed or we fail
- If you know your HW branch strategy you can adapt your code accordingly
- More on this later ...

Resource conflicts

Super-scalar architectures

- Until now in our simple architecture we assumed only one **execution unit** — one ALU
- Integer ALUs are not that expensive in HW (at least compared with other components that we introduced)
- Idea : replicate ALUs not necessarily with same functionalities (common subset + some specialized functions)
- If there are independent instructions in the program they can be executed on different **execution units** (new term for ALU) and thus avoid pipeline stall
- The problem is only how to redirect executions on multiple execution units
- You can de-multiply the complete execution pipeline ...

Super-scalar execution pipeline

- With multiple execution pipelines the IPC can now be > 1
- But this is still not guaranteed
- Extraction of the ILP **could** be better assuming better automatic instruction scheduling, or better code (compiler or written)

Cycle	1	2	3	4	5	6	7	8
Div R2, R1	Fetch	Decode	Execute			Write		
Add R3, R2	Fetch	Decode	Wait			Execute	Write	
Sub R8, R7		Fetch	Decode	Execute	Write			
Mul R4, R3		Fetch	Decode	Wait			Execute	Write
Sub R10, R9			Fetch	Decode	Execute	Write		
Sub R12, R11			Fetch	Decode	Execute	Write		
Sub R14, R13				Fetch	Decode	Execute	Write	
Add R5, R4				Fetch	Decode	Wait		Execute

Super-scalar architectures

- If ALUs are different : e.g. we can have HW multiplier in one ALU and not in the other
- If instructions are ADD and then MUL this is OK, both ALUs can be used at the same time (work in parallel)
- If instructions are ADD OK too
- But what happens if both are MUL?
  - ✓ **Resource conflict** — both instructions try to use the same execution unit at the same time
  - ✓ Stall
  - ✓ We loose cycles as before
- Extra constraints on instruction scheduler …(more difficult to extract ILP)

Loop unrolling

Loop unrolling

- Goal : increase the probability of possible overlapping instructions to help scheduler avoid pipeline stalls
- In loops this is very frequent situation, look at the code :

```
for(i=0; i<1000; i++)
    A[i] = A[i] + B[i];
```

- If there is a delay on `A[i] = A[i] + B[i]` computation, there will be a pipeline stall on every loop iteration (X 1000 times) !
- Consider the following loop transformation :

```
for(i=0; i<250; i++) {
      A[i + 0*250] = A[i + 0*250] + B[i + 0*250];
      A[i + 1*250] = A[i + 1*250] + B[i + 1*250];
      A[i + 2*250] = A[i + 2*250] + B[i + 2*250];
      A[i + 3*250] = A[i + 3*250] + B[i + 3*250]; }
```

- Four computations are now independent : we removed data dependency on subsequent instructions

Loop unrolling — gains

- Transformed loop :

```
for(i=0; i<250; i++) {
        A[i + 0*250] = A[i + 0*250] + B[i + 0*250];
        A[i + 1*250] = A[i + 1*250] + B[i + 1*250];
        A[i + 2*250] = A[i + 2*250] + B[i + 2*250];
        A[i + 3*250] = A[i + 3*250] + B[i + 3*250]; }
```

- Four computations :
  - ✓ Can do register renaming
  - ✓ Could order instructions to minimise the pipeline stall
  - ✓ If super-scalar, the system could use multiple execution units at the same time

Example : loop operation + not scheduled

```
for(i=0; i<1000; i++)
  A[i] = A[i] + B[i]
```

```
MOV  DX, [A];                -- load pointer to A
MOV  CX, 1000;               -- load loop counter (i)
loop:
   MOV    AX, [DX];          -- load A(i)
   ADD    BX, AX;            -- ADD
   MOV    Mem2, BX;          -- store result
   ADD    DX, 8;             -- move to next arr. elem.
   SUB    CX, 1;             -- dec loop counter
JNZ loop;
```

Example : scheduled

- ld ADD takes 3 cycles, how can you improve the overall execution time ? (modify the following code)

```
MOV  DX, [A];                -- load pointer to A
MOV  CX, 1000;               -- load loop counter (i)
loop:
   MOV    AX, [DX];            -- load A(i)
   ADD    BX, AX;              -- ADD
   MOV    Mem2, BX;            -- store result
   ADD    DX, 8;               -- move to next arr. elem.
   SUB    CX, 1;               -- dec loop counter
JNZ loop;
```

Example : loop unrolling 2X

- Unroll the loop twice using the following template ?
- What is the maximum number of unrolled iterations you can do?

```
MOV  DX, [A];                -- load pointer to A
MOV  CX, 500;                -- load loop counter (i)
loop:
   MOV    AX, [DX];           -- load A(i)
   MOV    EX, [DX+500];       -- load A(i+500)
...
JNZ loop;
```