DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

COMPUTER ARCHITECTURES

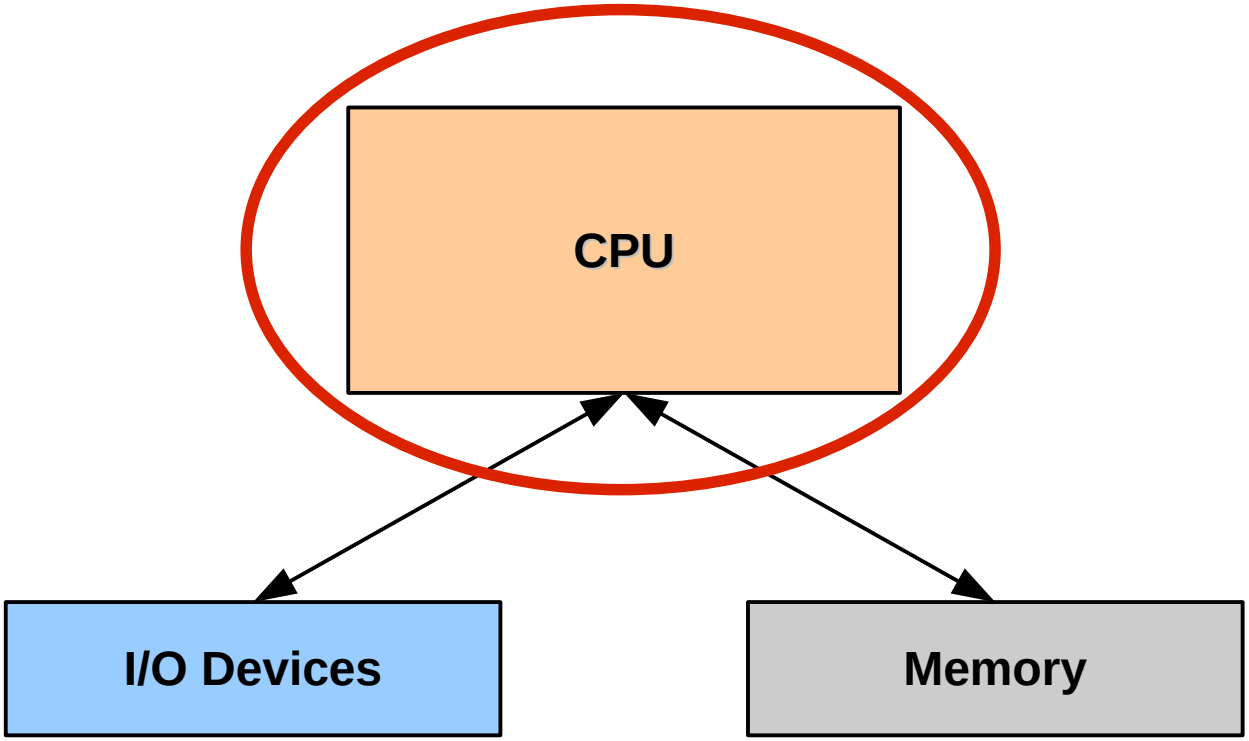
Instruction Pipeline

Gábor Horváth, ghorvath@hit.bme.hu

Budapest,
2023. 05. 02.

MŰEGYETEM 1782

WHERE ARE WE?



EXECUTING INSTRUCTIONS

- Our CPU in the examples: RISC
- Instructions:
  - Load / Store
    `R1 ← MEM[R0+42]` or `MEM[R0+42] ← R1`
    - Addressing mode: indirect with offset
  - Arithmetic / Logic
    `R1 ← R2 + R3` or `R1 ← 42 * R3` or `R4 ← R1 & R5`
  - Control
    - Branch, jump
    - Conditional jump
      `JUMP -24 IF R2 == 0`
  - Only Load and Store are allowed to access the memory

EXECUTING INSTRUCTIONS

- Stages of the execution of instructions:
  1) Instruction Fetch, **IF**
     + increase program counter (if it is not a jump)
  2) Instruction decode/register fetch, **ID**
     - Decoding the binary encoded instruction
       - Determine what kind of instruction it is
         → also provides control signals for ALU
       - Operands located in registers are retrieved from the register file
  3) Execution, **EX** – ALU executes an operation

     In case of
     - Load/Store: it calculates the memory address (`R1←MEM[R0+42]`)
     - Arithmetic/Logic: it calculates the result (`R1 ← R2*R3`)
     - Conditional jump: it evaluates the condition and also calculates branch target address (`JUMP [PC]-24 IF R2==0`)

EXECUTING INSTRUCTIONS

… cont'ed:

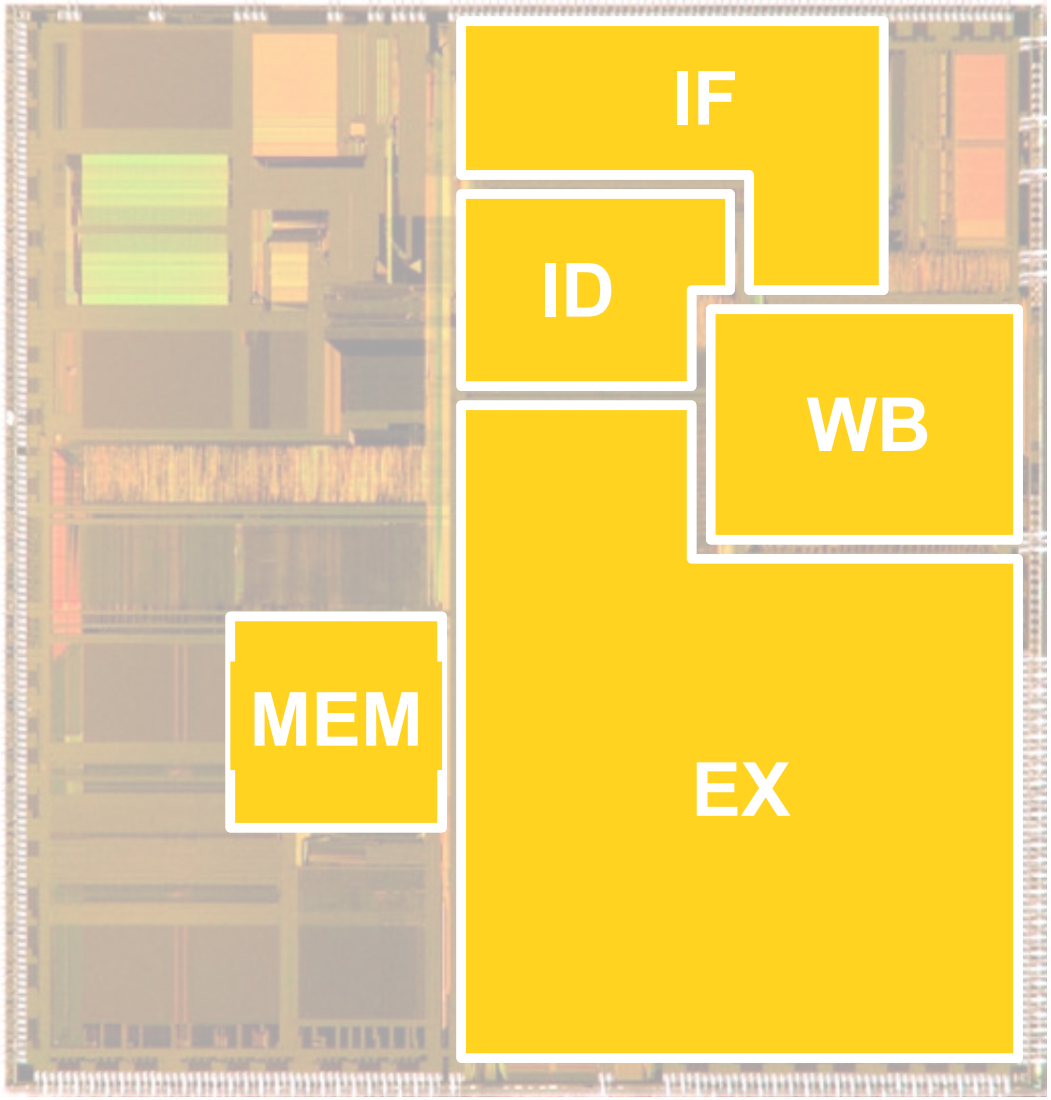
4) Memory access, **MEM**
   In case of
   - Load/Store: the memory operation is accomplished
   - Arithmetic/logic, jumps: this phase does nothing

5) Write-back, **WB**
   In case of
   - Arithmetic/logic: stores the result to the register file (**R1 ← R2*R3**)
   - Load: stores the memory content retrieved to register file (**R1 ← MEM[R0+42]**)
   - Store, jumps: this phase does nothing

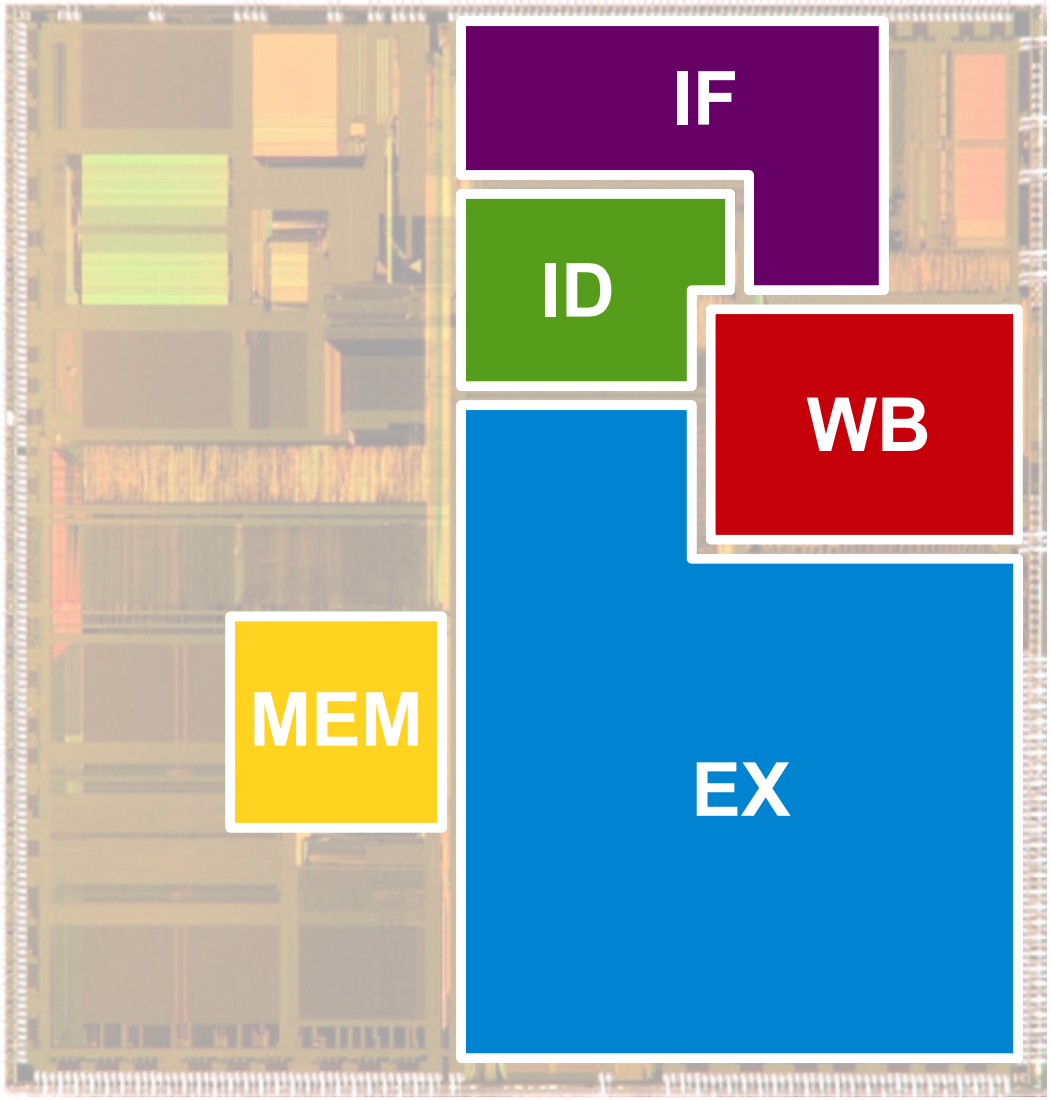
EXECUTING INSTRUCTIONS

- Stages to execute:
  - Arithmetic instructions: IF, ID, EX, WB
  - Store instructions: IF, ID, EX, MEM
  - Load instructions: IF, ID, EX, MEM, WB
  - Jumps: IF, ID, EX
  - Etc.

EXECUTING INSTRUCTIONS

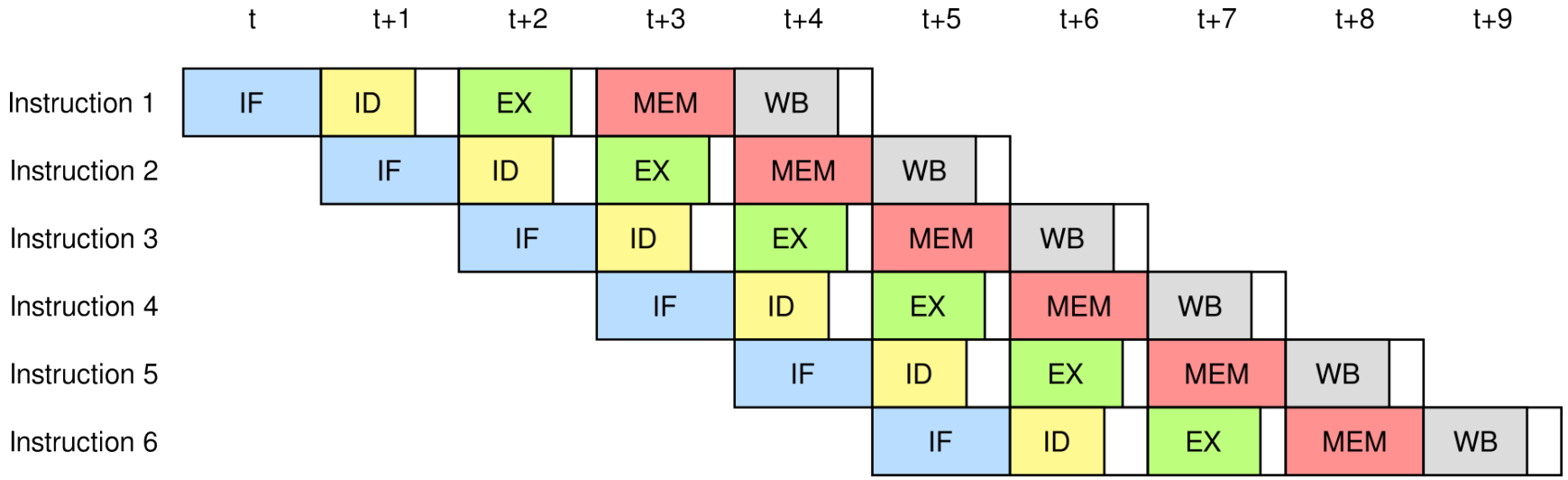


EXECUTING INSTRUCTIONS



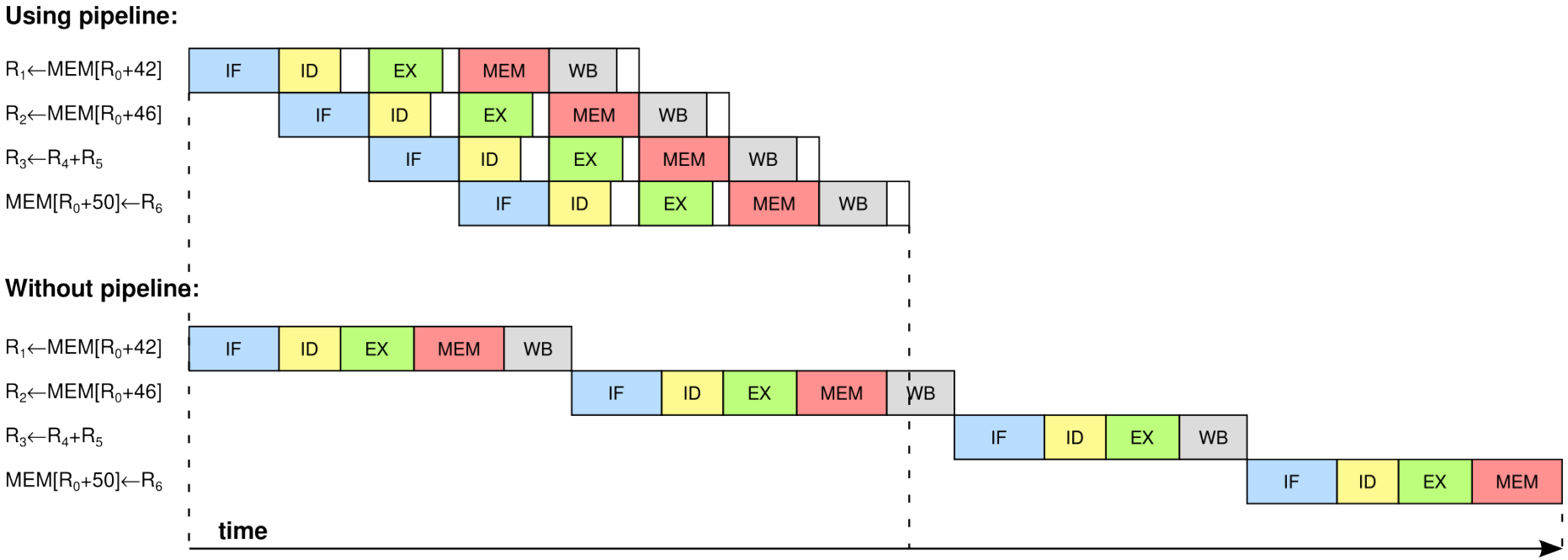
INSTRUCTION PIPELINE

- This was the concept of instruction pipeline
- Stages of instruction executions are overlapped:
  - Every stage needs to be given the same amount of time: ***cycle time***



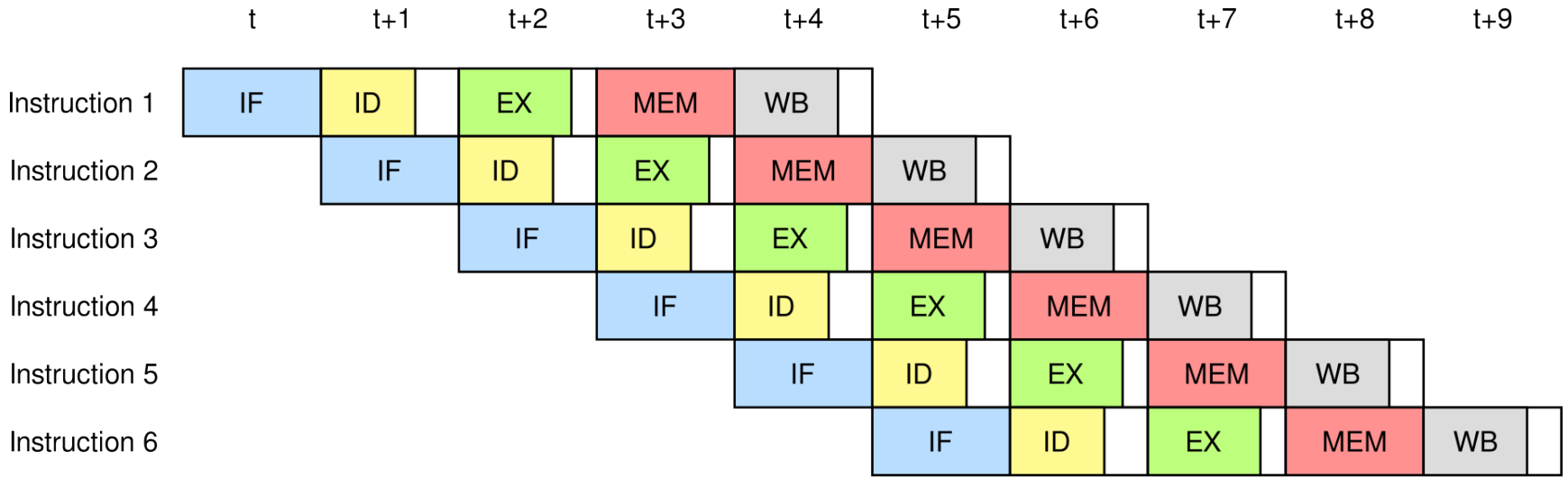
INSTRUCTION PIPELINE

- How much do we gain?
  - On the one hand: we overlapped the execution of several instructions → gain
  - On the other hand: there are idle time intervals (some stages are not needed by all types of instructions, some stages are faster than the cycle time, etc.)
- As long as the instructions are independent, the gain is significant



INSTRUCTION PIPELINE

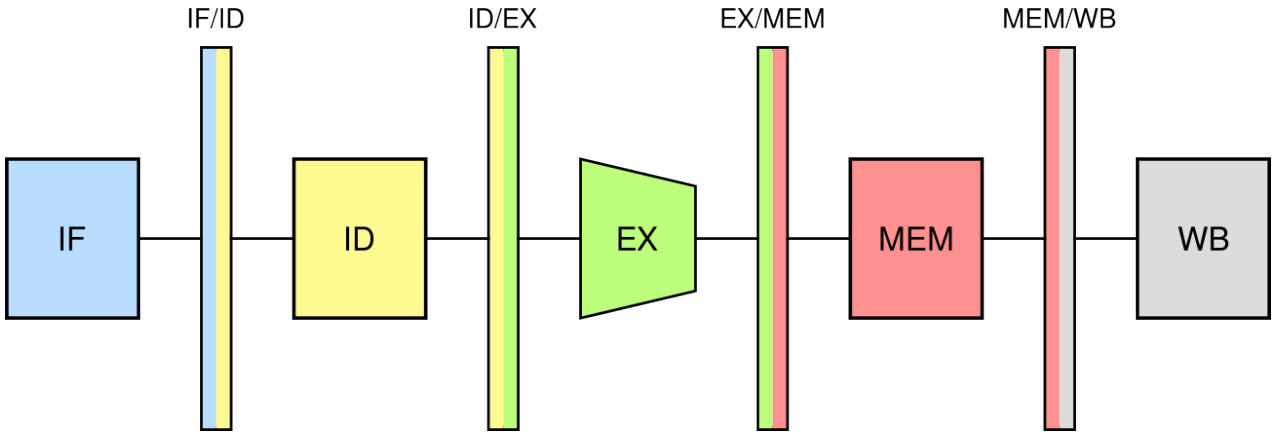
- Quantities:
  - Depth: number of stages
  - Latency: the execution time of an instruction
  - Throughput: executed instructions / sec



PIPELINE REGISTERS

- Each stage needs to exchange some information with the neighboring stages about the current instruction

  → these are stored in **pipeline registers**



- Example:
  - IF puts the fetched instruction to register IF/ID
  - ID takes it out from IF/ID, and puts the ALU control signals and operands to register ID/EX
  - EX takes it out from ID/EX, executes the arithmetic operation, puts the result to register EX/MEM
  - Etc... (later)

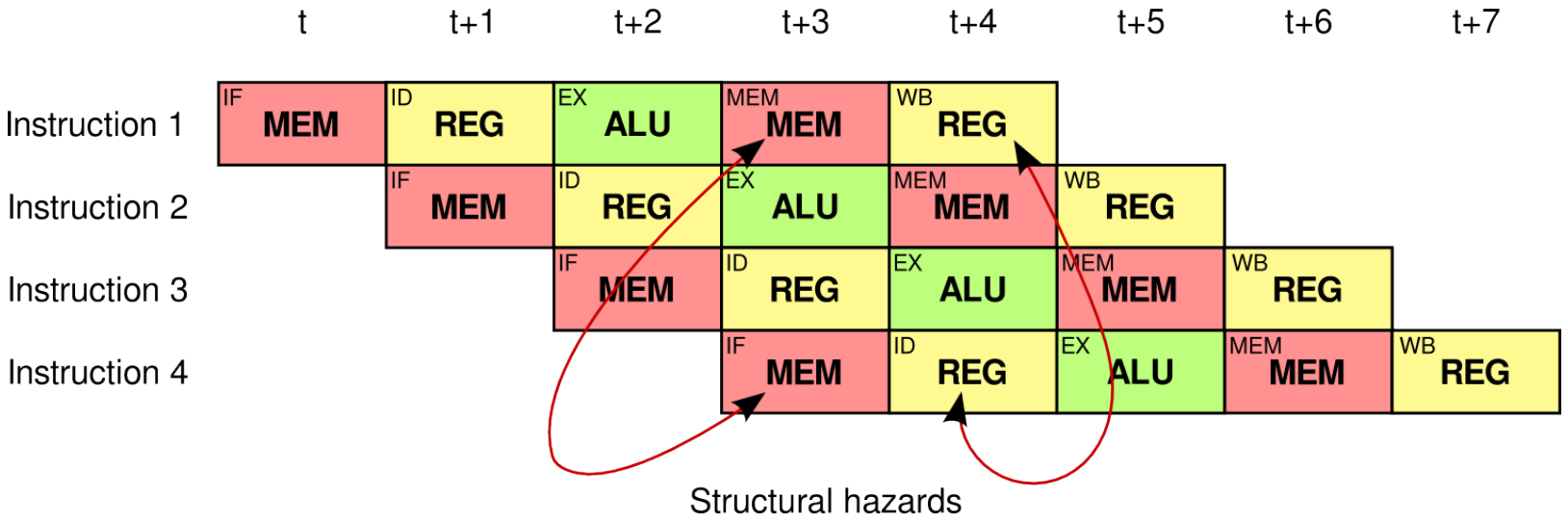
HAZARDS

- There is a serious problem
- The instructions are not independent!
  - Sometimes they need the same system resources at the same time
    → ***Structural hazard***
  - An operand of an instruction is the result of the previous instruction
    → ***Data hazard***
  - Conditional jumps: we do not know where the execution of the program continues until the branch condition is evaluated
    → ***Control hazard***

  How to treat them:
  - We either resolve them by using some smart trick
  - ...or stop the pipeline till the hazard persists
    → degrades efficiency

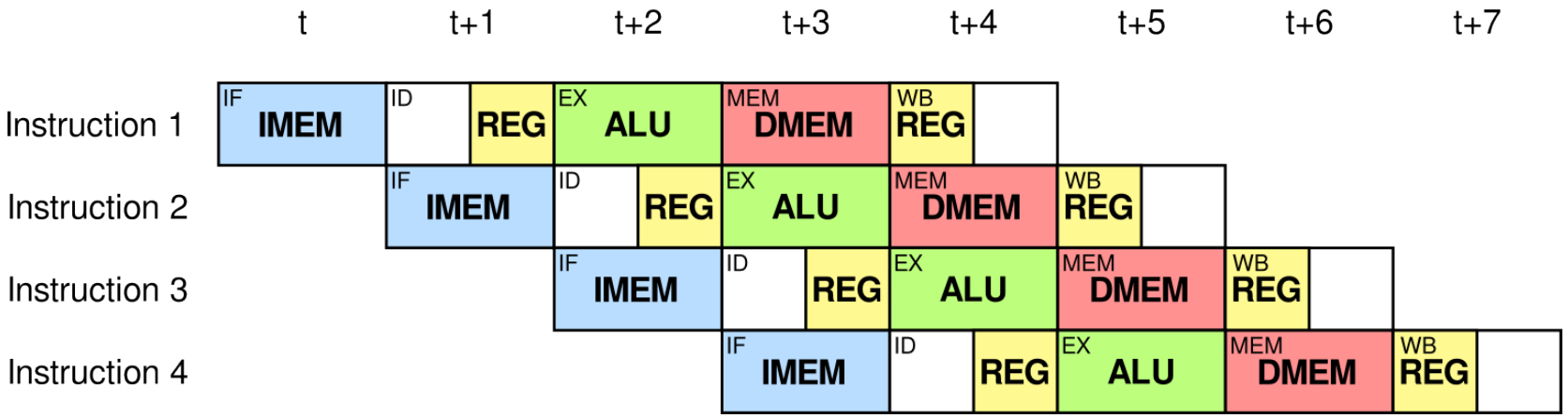
STRUCTURAL HAZARDS

- Some system resources are used in several stages as well
  - IF: Memory
  - ID: Register file
  - EX: ALU
  - MEM: Memory
  - WB: Register file



STRUCTURAL HAZARD

- Resolving it:
  - In case of the „memory” resource:
    - By using separate instruction and data cache
  - In case of the „register file” resource:
    - One of the stages uses the register file in the first half of the cycle time, the other stage uses it in the second half



DATA HAZARD

- Caused by data dependencies
- Data hazard:

  Multiple instructions operate on the same data address
  (on the same register, or on the same memory address)

- E.g. an operand of an instruction is the result of a previous instruction

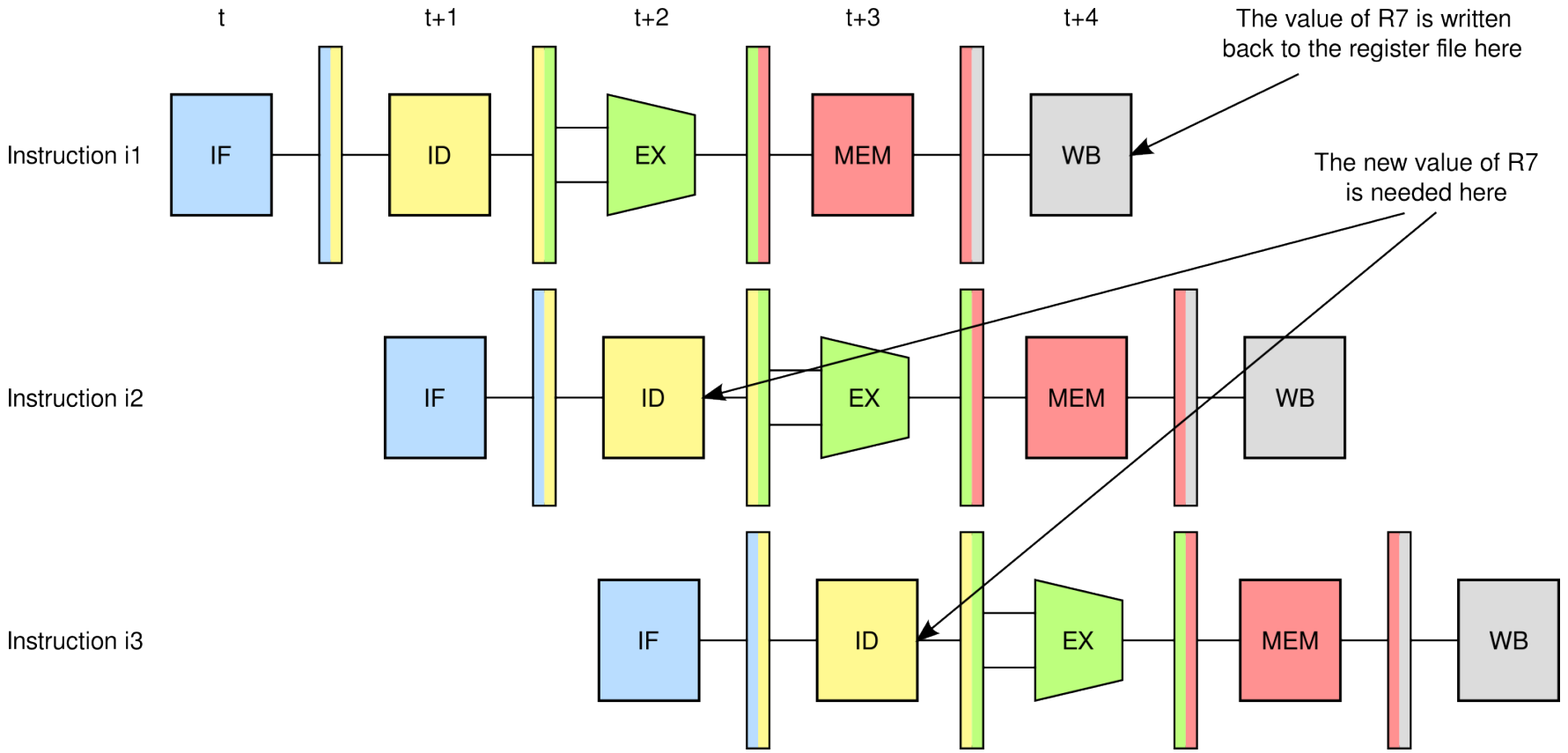
  → ***RAW dependency*** (Read After Write)

```
i1:  R3 ← MEM[R2]
i2:  R1 ← R2 * R3
i3:  R4 ← R1 + R5
i4:  R5 ← R6 + R7
i5:  R1 ← R8 + R9
```

- RAW dependencies:

  i1 ↔ i2, i2 ↔ i3

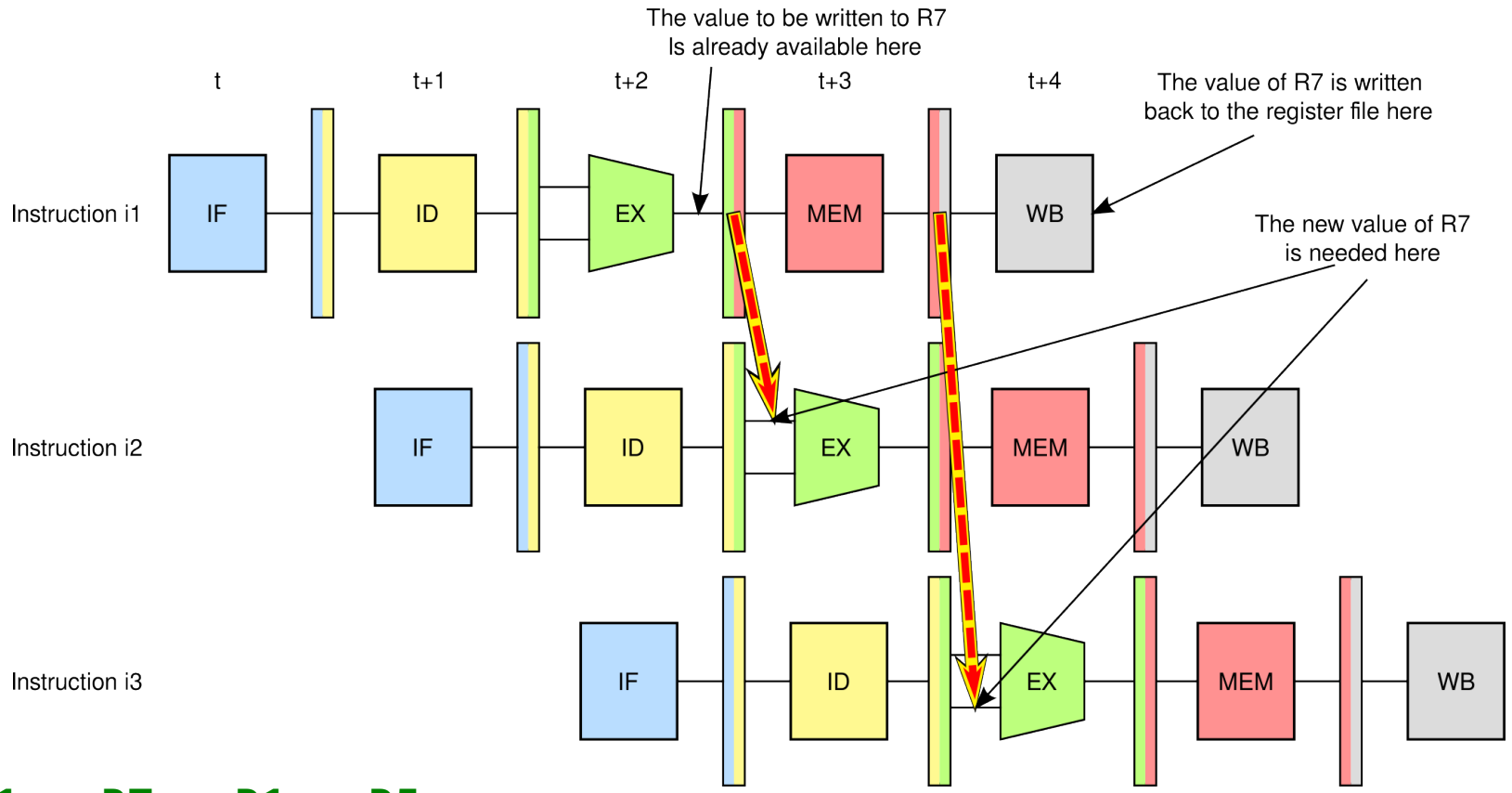
RAW DEPENDENCIES




```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

RAW DEPENDENCIES

- The result is available, but it is not yet stored to the register file
- Let us read it out from the appropriate pipeline register → ***forwarding***



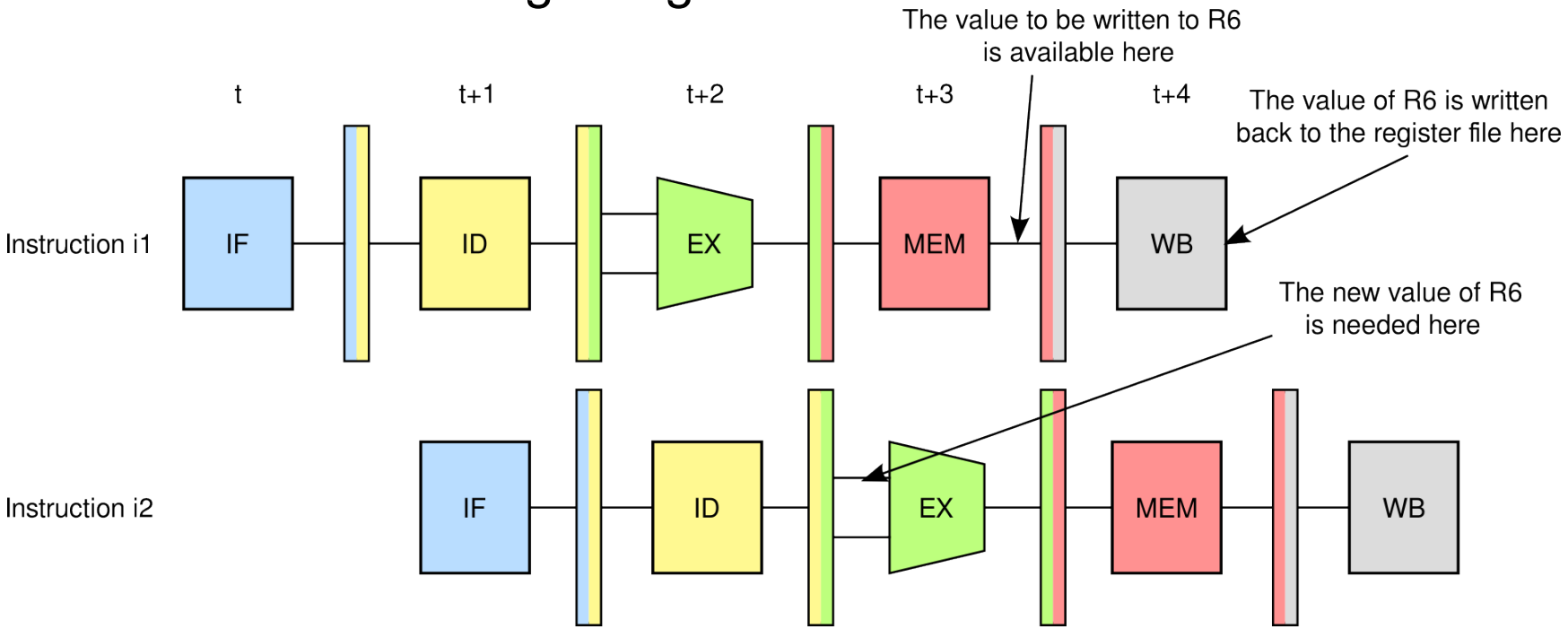

```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

RAW DEPENDENCIES

- Forwarding: there are situations when it does not help

```
i1:  R6 ← MEM[R2]
i2:  R7 ← R6 + R4
```

- The value of R6 is available at the end of MEM stage
- i2 needs it at the beginning of EX

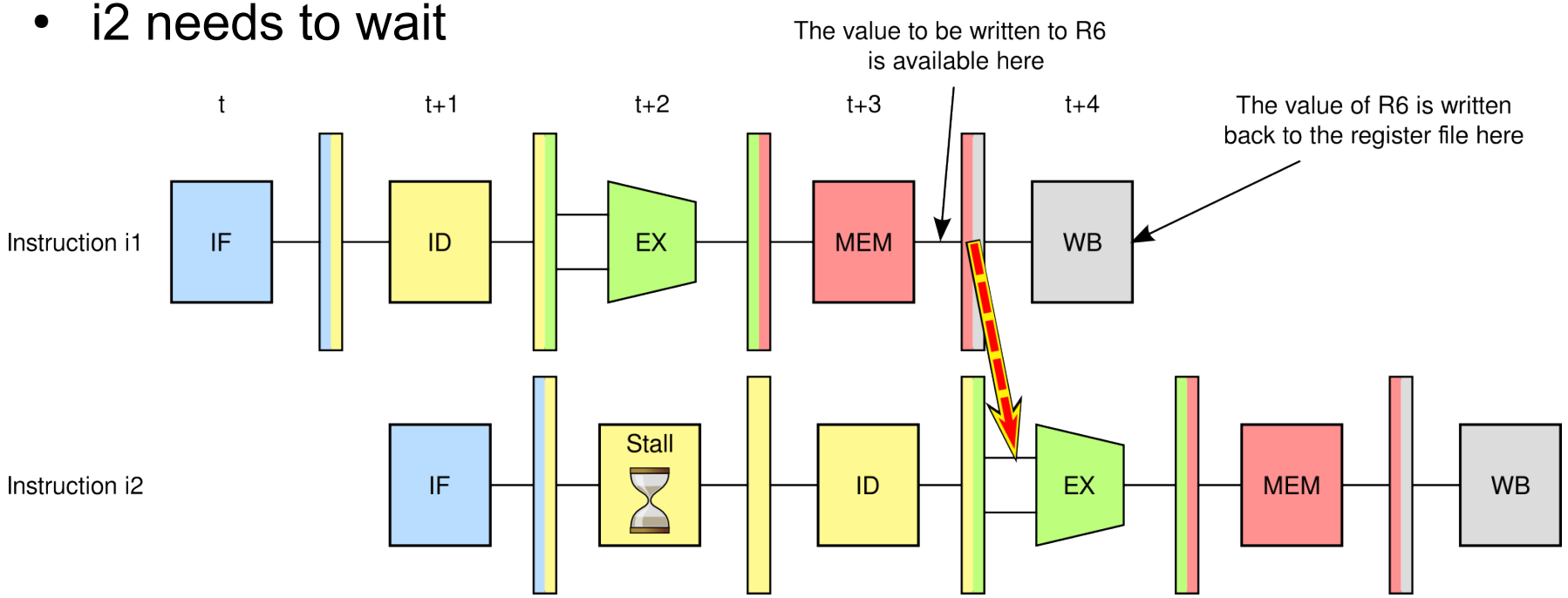


RAW DEPENDENCIES

- Forwarding: there are situations when it does not help

```
i1:  R6 ← MEM[R2]
i2:  R7 ← R6 + R4
```

- The value of R6 is available at the end of MEM stage
- i2 needs it at the beginning of EX
- i2 needs to wait



DATA HAZARD

- Data dependencies:
  - **RAW**: an instruction reads the register/address written by an earlier instruction
    - Solution: forwarding/waiting
  - WAR: an instruction writes the register/address read by an earlier instruction
    - Not a problem in this simple pipeline
  - WAW: multiple instructions write the same register/address
    - Not a problem in this simple pipeline
  - RAR: multiple instructions read the same register/address
    - Never a problem

- Example:

```
i1:  R3 ← MEM[R2]
i2:  R1 ← R2 * R3
i3:  R4 ← R1 + R5
i4:  R5 ← R6 + R7
i5:  R1 ← R8 + R9
```

- WAW: i2 ↔ i5 (write after write)
- WAR: i3 ↔ i4 (write after read)

CONTROL HAZARDS

- There is a problem with the conditional jump instructions:
  - IF fetches them
  - The jump condition and the jump target address is calculated only by stage EX!
  - Where to fetch the subsequent instruction from till it completes the EX stage?
- Solutions:
  - Stop and wait till the jump instruction completes the EX stage → used by Intel 80386
  - Predict the jump condition and the address immediately

    → ***branch prediction***

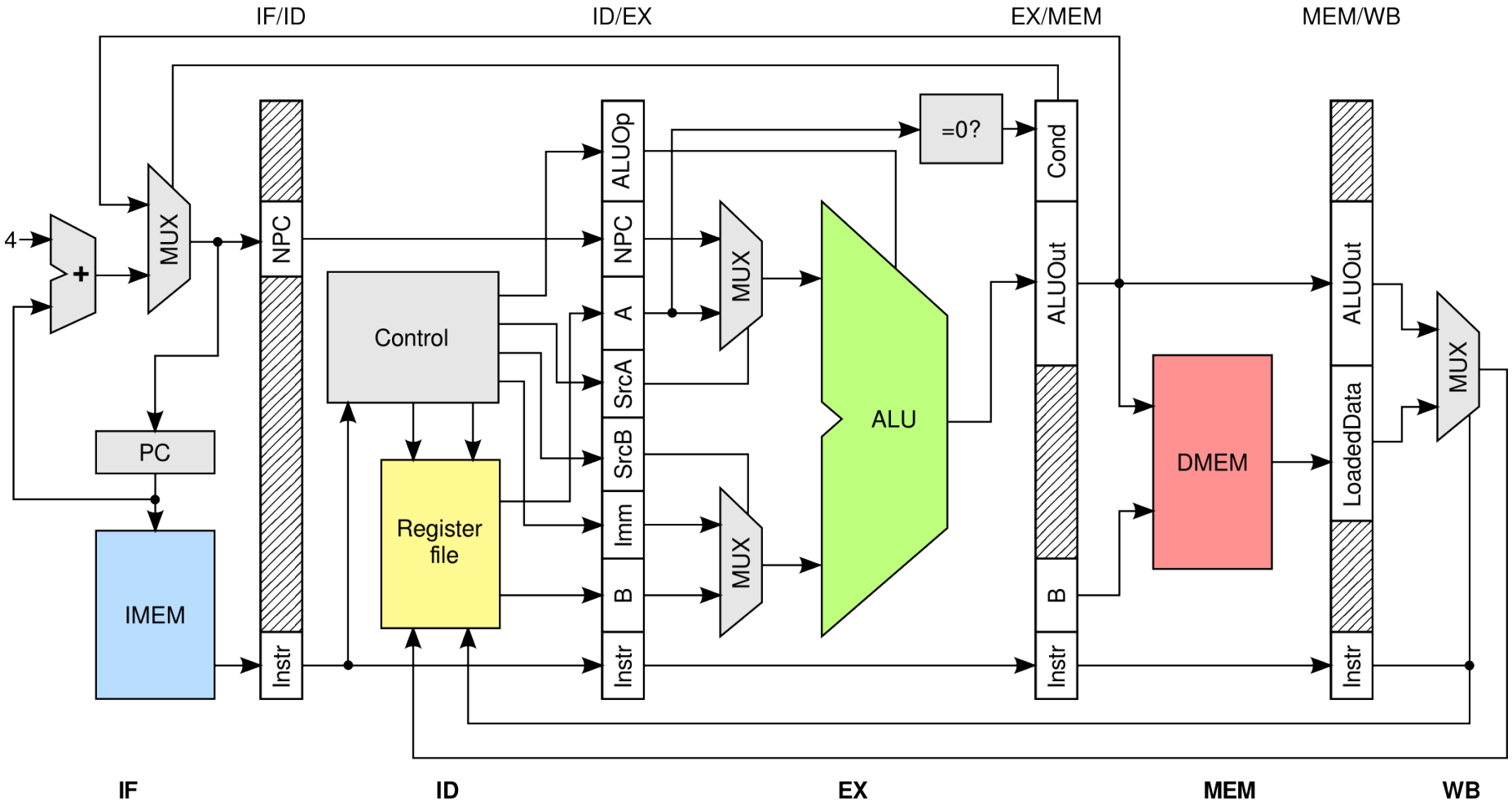
CONTROL HAZARDS

- **Static branch prediction** → not adaptive
  There are methods which
  - Always predicts that the jump won't be taken
    - It costs nothing
    - In case of misprediction, the instructions fetched by mistake are invalidated (by Intel 80486)
    - However, conditional jumps are more often taken than not taken (loops)
  - Always predicts that the jump will be taken
    - OK, but how do we know the jump target address then?
    - It works well if the target address is calculated well before the condition
  - Jump forward: not taken, jump backward: taken
    - Motivation: loops usually do jump back, except, when they are finished
- **Dynamic branch prediction**
  - Adapts the behavior of the program (it learns which jumps are taken and which are not)
  - It is much better than the static one
  - All modern CPUs use dynamic branch prediction

CONCLUSION

- What did we learn up to this point?
  - The concept of instruction pipeline
  - That the pipeline stages communicate by using the pipeline registers
  - How to cope with hazards
- Let us design our own instruction pipeline!

IMPLEMENTATION OF THE PIPELINE



THE ALU

- ALU: has a central role
- It does computations:

```
R1 ← R1 * 42
R1 ← R2 & R3
PC ← PC + 42          (JUMP +42)
R1 ← MEM [R2 + 42]
```

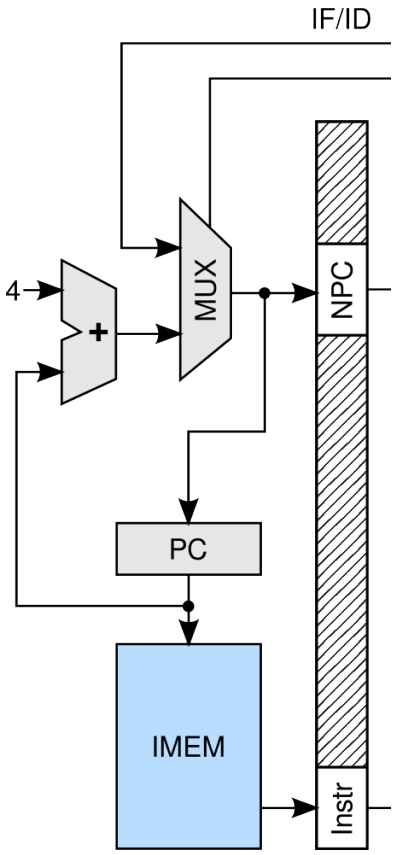
  - First operand: **register** or **PC**
  - Second operand: **register** or **constant** (immediate)
- Evaluates jump conditions:

```
JUMP -28 IF R1==0
```

- The task of the ID phase:
  - Preparing operands for the ALU
  - Preparing operand selection signals for the ALU
  - Preparing the operation code (+, -, *, /, &, etc.) for the ALU

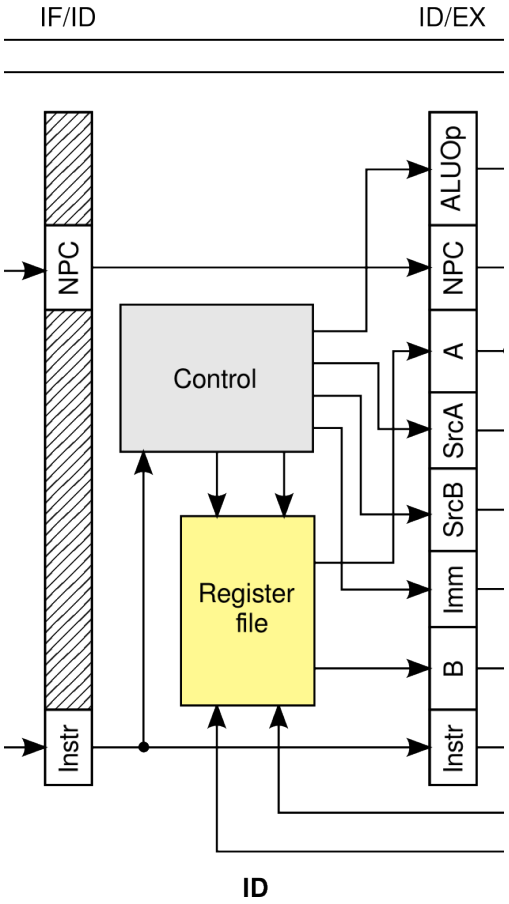
STAGE IF

- Updates the program counter:
  - If jump (**`EX/MEM.Instr.Opcode==branch`**), and if it will be taken (**`EX/MEM.Cond==TRUE`**):
    - **`PC ← EX.MEM.ALUOut`**
  - Otherwise:
    - **`PC ← PC+4`**
- Passing the instruction word and the new program counter value forward:
  - **`IF/ID.NPC ← PC`**
  - **`IF/ID.Instr ← IMEM[PC]`**



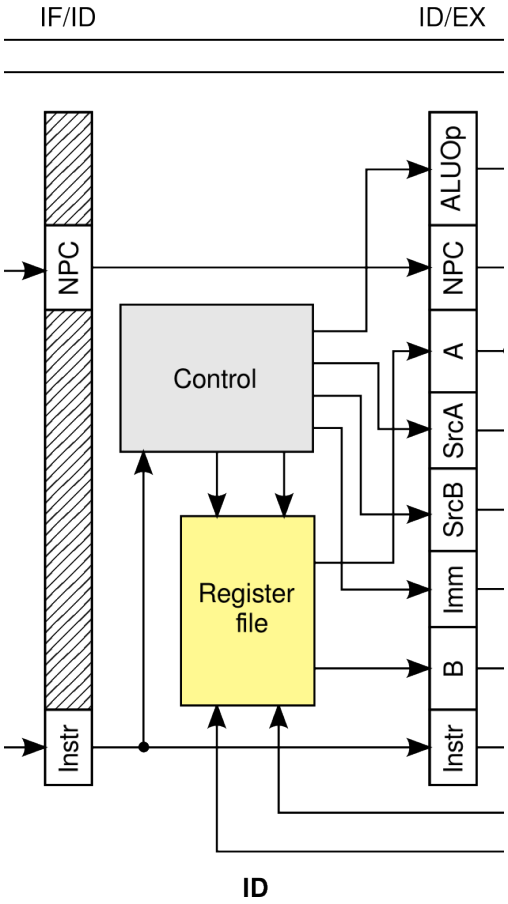
STAGE ID (PART 1)

- Prepares the operands:
  - First operand:
    - `ID/EX.NPC ← IF/ID.NPC`
    - `ID/EX.A ← Reg [IF/ID.Instr.ra]`
  - Selection signal for the first operand:
    - In case of jump instruction (`IF/ID.Instr.Opcode == branch`):
      - `ID/EX.SrcA ← npc`
    - Otherwise:
      - `ID/EX.SrcA ← regA`
  - Second operand:
    - `ID/EX.Imm ← IF/ID.Instr.imm`
    - `ID/EX.B ← Reg [IF/ID.Instr.rb]`
  - Selection signal for the second operand:
    - If constant (`IF/ID.Instr.HasImm`):
      - `ID/EX.SrcB ← imm`
    - Otherwise:
      - `ID/EX.SrcB ← regB`
- `...`



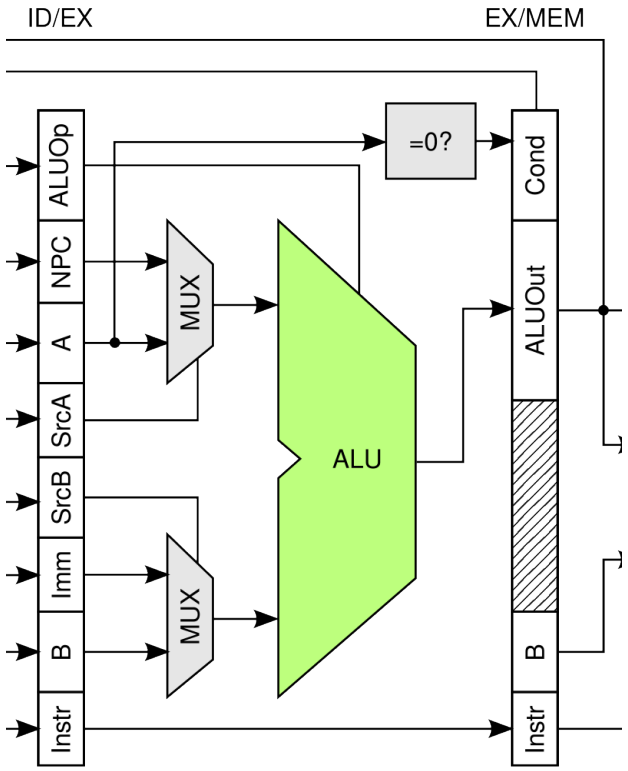
STAGE ID (PART 2)

- `...`
- Prepares operation code:
  - If we have an arithmetic instruction (`IF/ID.Instr.Opcode == arithm`):
    - `ID/EX.ALUOp ← IF/ID.Instr.Func`
  - Otherwise (PC update or address calculation):
    - `ID/EX.ALUOp ← „+”`
- Passing the instruction word forward:
  - `ID/EX.Instr ← IF/ID.Instr`



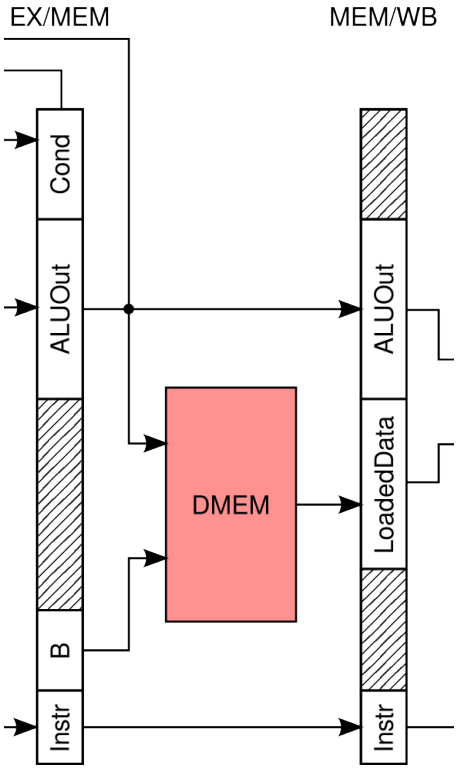
STAGE EX

- ALU inputs:
  - If `ID/EX.SrcA == npc`
    - `ALU.A ← ID/EX.NPC`
  - If `ID/EX.SrcA == regA`
    - `ALU.A ← ID/EX.A`
  - If `ID/EX.SrcB == imm`
    - `ALU.B ← ID/EX.Imm`
  - If `ID/EX.SrcB == regB`
    - `ALU.B ← ID/EX.B`
  - Operation code:
    - `ALU.Op ← ID/EX.ALUOp`
- Storing the result:
  - `EX/MEM.ALUOut ← ALU.Out`
- Comparison unit:
  - `EX/MEM.Cond ← ID/EX.A == 0`
- Forwarding B in case of „Store” instructions
  - `EX/MEM.B ← ID/EX.B` (This will be the data to store)
- Forwarding the instruction word:
  - `EX/MEM.Instr ← ID/EX.Instr`



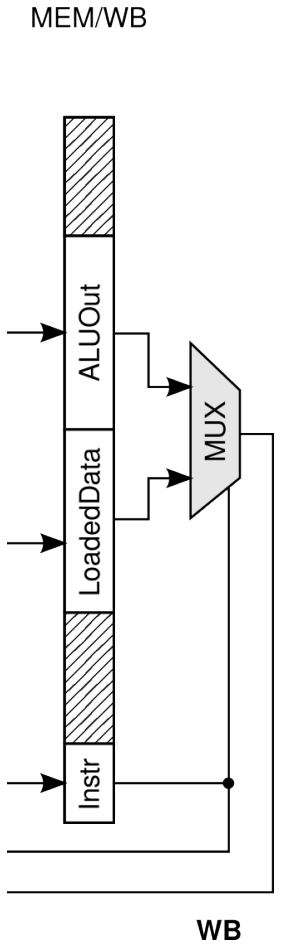
STAGE MEM

- Memory address
  - **`ALUOut`**
- In case of “Store” (**`ID/EX.Instr.Opcode == Store`**)
  - **`MEM[EX/MEM.ALUOut] ← EX/MEM.B`**
    (Data to write: B)
- In case of “Load” (**`ID/EX.Instr.Opcode == Load`**)
  - **`MEM/WB.LoadedData ← MEM[EX/MEM.ALUOut]`**
- In case of arithmetic instruction:
  - **`MEM/WB.ALUOut ← EX/MEM.ALUOut`**
    (Forwarding the result)
- Forwarding the instruction word:
  - **`MEM/WB.Instr ← EX/MEM.Instr`**

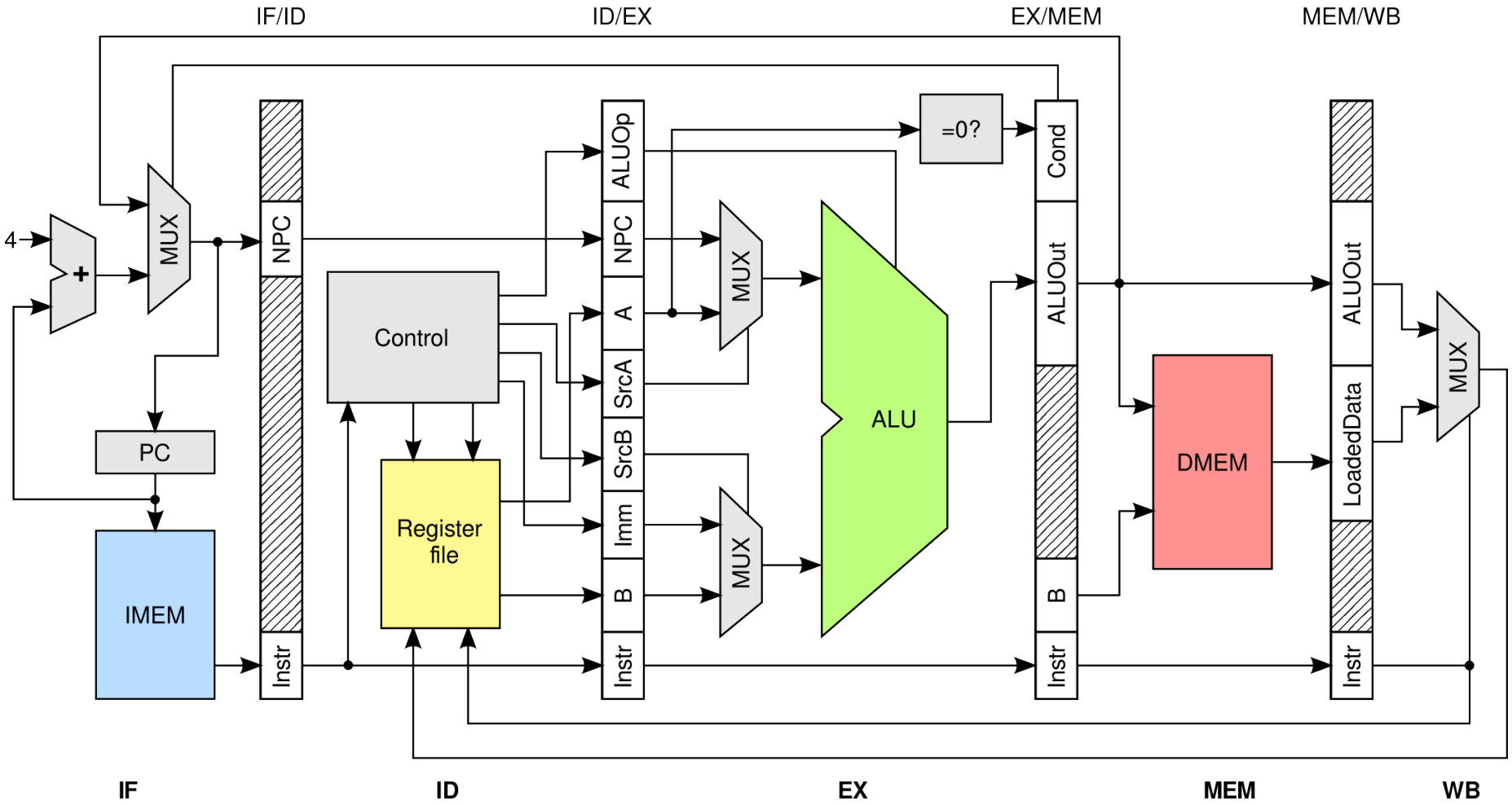


STAGE WB

- Updating the register file (`Reg[ ]`)
  - In case of arithmetic instructions:
    (`MEM/WB.Instr.Opcode == arithm`)
    - `Reg[MEM/WB.Instr.rd] ← MEM/WB.ALUOut`
  - In case of “Load” instruction:
    (`MEM/WB.Instr.Opcode == Load`)
    - `Reg[MEM/WB.Instr.rd] ← MEM/WB.LoadedData`

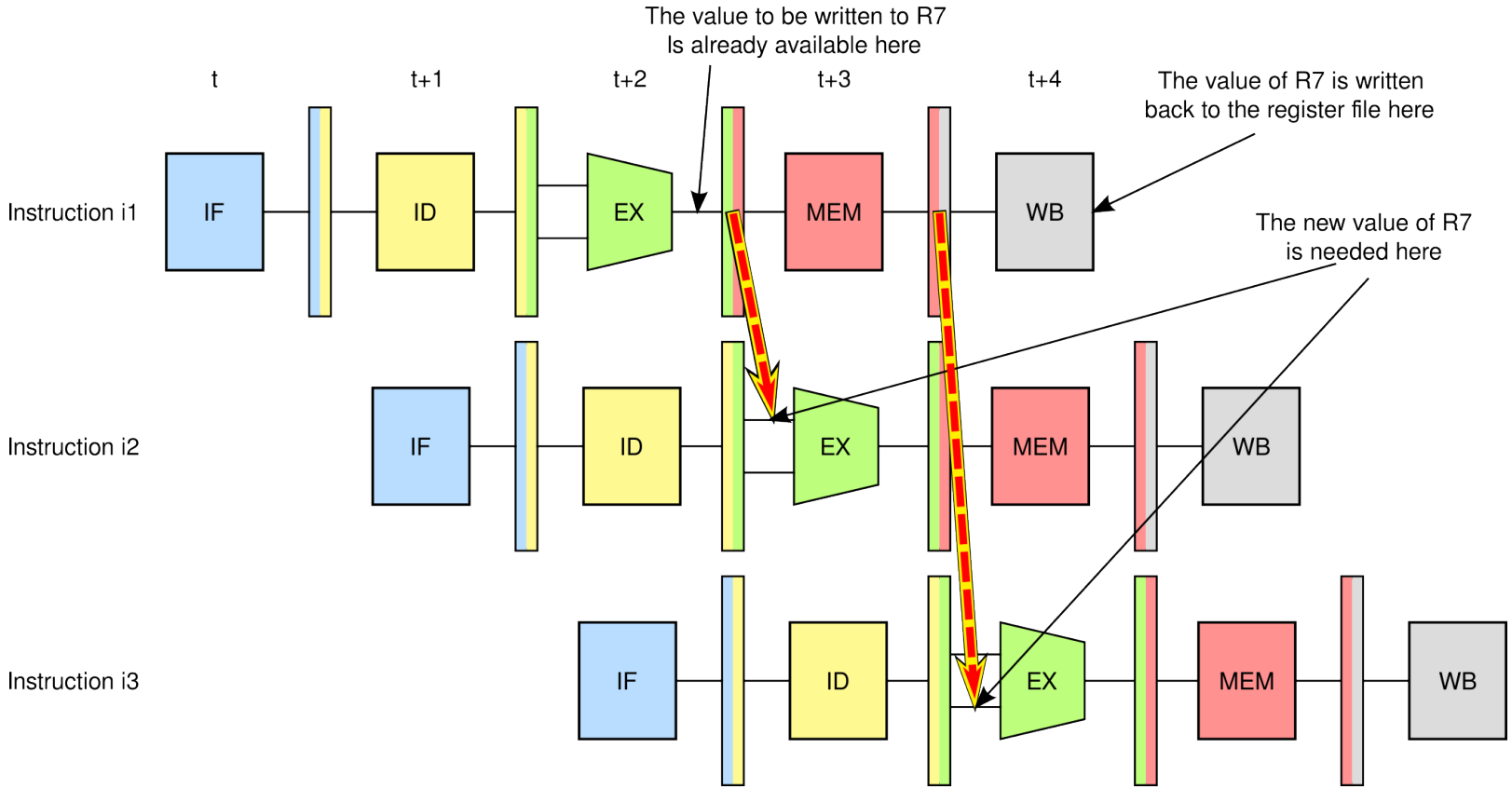


IMPLEMENTATION OF THE PIPELINE



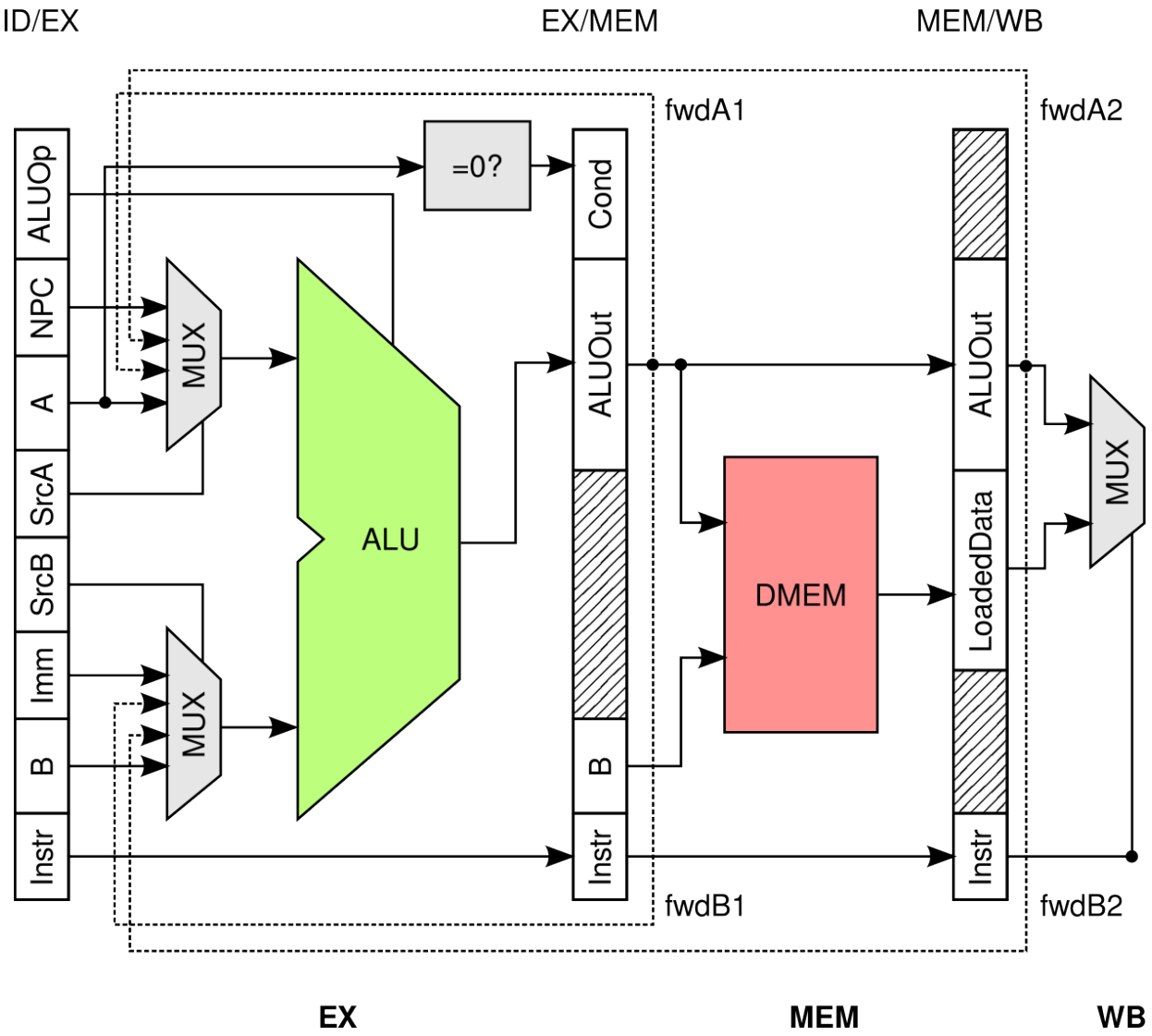
RAW DEPENDENCIES

- Recall: we need forwarding




```
i1:  R7 ← R1 + R5
i2:  R8 ← R7 + R2
i3:  R5 ← R8 + R7
```

RAW DEPENDENCIES



RAW DEPENDENCIES

- How does stage ID know what the operands of ALU are?
  - There are two operands
  - Each of them can have two different sources (NPC↔register, constant↔register)
  - Register operands can came from three locations:
    - From the decoder, ID
    - From the ALU output
    - From the ALU output of the previous step
- Operand selection logic (in stage ID):
  - For example, operand 1, **`ID/EX.SrcA`** can be **`0, 1, 2, 3:`**
    - **`npc: IF/ID.Instr.Opcode == branch`**
    - **`fwdA1:`** if **`IF/ID.Instr.Opcode==arithm && IF/ID.Instr.ra == ID/EX.Instr.rd`**
    - **`fwdA2:`** if **`IF/ID.Instr.Opcode==arithm && IF/ID.Instr.ra == EX/MEM.Instr.rd`**
    - **`regA:`** if **`IF/ID.Instr.Opcode==arithm,`** otherwise
  - Similarly for operand 2

COPING WITH OTHER HAZARDS

- If the resolution of a RAW dependency needs to stall the pipeline:
  - Stage ID can detect it (as it detects RAW as well)
  - Stops stage IF
- Control hazards
  - Stop & wait method: detect if a conditional jump arrived and stop stage IF for 2 cycles
  - Static prediction that always predicts „jump not taken”
    - Goes on with fetching the next instructions
    - If the jump turns out to be taken, the instructions fetched by mistake need to be invalidated:
      - Instructions are carrying a „valid” flag
      - If an instruction needs to be invalidated, „valid” is set to 0
      - Instructions with Valid=0 do nothing in stages MEM and WB (thus they leave no traces at all)

CONCLUSION

- What did we learn up to this point?
  - The concept of instruction pipelines
  - The role of pipeline registers
  - How to cope with hazards
  - How to implement an instruction pipeline
  - How to implement hazard detection and resolution
- Everything is nice till something extraordinary happens
- What can happen?
  - Exceptions!
    - An interrupt from an I/O device
    - Page fault
    - Protection fault
    - Invalid instruction
    - Etc.

PRECISE EXCEPTIONS

- Ideal behavior
  - Given that the exception raises during instruction i.
  - We want to ensure a state where
    - All instructions before i are completed
    - Instructions after i have not been started at all

  → ***precise exceptions***
- Not easy to implement if we have a pipeline
- When an exception happens, we want to prevent the started but not yet completed instructions from leaving a trace
- … but sometimes it can not be prevented

  (a later instruction might have already changed the content of the memory)

PRECISE EXCEPTIONS

- What can happen in the various stages:
  - Stage IF: page fault, protection fault
  - Stage ID: invalid instruction
  - Stage EX: arithmetic error (integer overflow, division by zero, etc.)
  - Stage MEM: page fault, protection fault
  - Stage WB: no exceptions can happen

	1	2	3	4	5	6
Rk ← Rm + Rn	IF	ID	EX	MEM	WB	
Ri ← MEM [Rj]		IF	ID	EX	MEM	WB

- Instruction 1.: Integer overflow in stage EX
- Instruction 2.: page fault in stage IF

  → The order of events is not correct! The exception of a later instruction occurs sooner!

PRECISE EXCEPTIONS

- A possible solution:
  - Exceptions are not handled immediately
  - The CPU just marks the instruction with a flag
  - Handling of exception: only in stage WB

    → the order of exceptions will be correct this way

	1	2	3	4	5	6
Rk ← Rm + Rn	IF	ID	EX	MEM	WB	
Ri ← MEM [Rj]		IF	ID	EX	MEM	WB

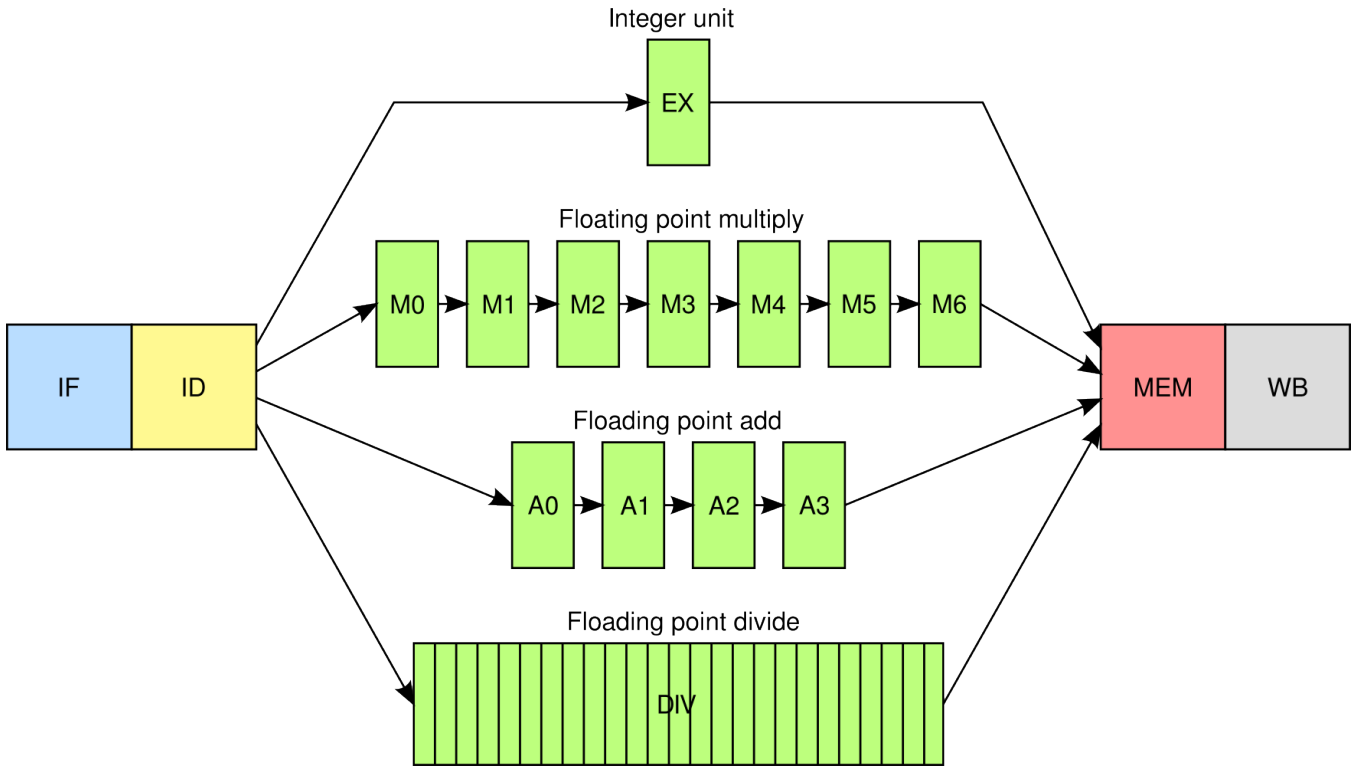
PIPELINE WITH MULTI-CYCLE OPERATIONS

- Until now all arithmetic operations were executed in 1 cycle
- But different arithmetic operations may have different **latencies**
  - Floating point operations need more cycles
  - Multiplication needs more cycles than addition
  - Division is the slowest operation
- The arithmetic functional units can be pipelined or unpipelined internally
  - E.g. the floating point addition might have a latency of 4 cycles,
  - ...but can start a new addition operation every cycle

  → ***Iteration interval*** = 1

PIPELINE WITH MULTICYCLE OPERATIONS

Functional unit	Latency	Iteration interval
Integer ALU	1	1
FP add	4	1
FP multiply	7	1
FP divide	25	25



PIPELINE WITH MULTICYCLE OPERATIONS

- Observe:
  - Instructions are started ***in-order***, but completed ***out-of-order***

D4 ← D1 * D5	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
D2 ← D1 + D3		IF	ID	A0	A1	A2	A3	MEM	WB		
D0 ← MEM [R0+4]			IF	ID	EX	MEM	WB				
MEM [R0+8] ← D5				IF	ID	EX	MEM	WB			

- In this example this was not a problem
- ...but it can introduce problems in other situations
- Stage ID has new tasks:
  - To keep the semantics of the program
  - To cope with new kinds of hazards and dependencies

STRUCTURAL HAZARDS INTRODUCED

Instruction	1	2	3	4	5	6	7	8	9	10	11
D4 ← D1 * D5	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
D2 ← D1 + D3				IF	ID	A0	A1	A2	A3	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
D0 ← MEM [R0+4]							IF	ID	EX	MEM	WB

- Problem:
  - 10.: there are 3 instructions in stage MEM at the same time
  - 11.: there are 3 instructions in stage WB at the same time
- Solution:
  - Stage ID knows exactly the latency of the instructions → it can detect such situations
  - If ID thinks that an instruction will cause structural hazard later → it does not let it move forward in the pipeline (stall)

MUCH MORE RAW DEPENDENCIES

- Data hazards are worse than before
- Problem: pipeline is longer
  - The result of an operation is calculated late
    → if an instruction uses that result, it has to wait a lot
  - There are more instructions under execution at the same time
    → the chance of having a RAW dependency among them is larger (compared to our simple 5-stage pipeline)

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	14	14	15	16	17
D4 ← MEM [R0+4]	IF	ID	EX	ME	**WB**												
D0 ← D4 * D6		IF	ID	**Stop**	M0	M1	M2	M3	M4	M5	M6	ME	**WB**				
D2 ← D0 + D8			IF	**Stop**	ID	**Stop**	**Stop**	**Stop**	**Stop**	**Stop**	**Stop**	A0	A1	A2	A3	ME	**WB**
MEM [R0+4] ← D2					IF	**Stop**	**Stop**	**Stop**	**Stop**	**Stop**	**Stop**	ID	EX	**Stop**	**Stop**	**Stop**	ME

WAW DEPENDENCIES

- WAW dependency:
  - Two instructions write their results to the same register
- If instructions are allowed to complete out-of-order → might introduce problems!
  - It might happen that the later instruction writes its result to the target register sooner
  - Stage ID has to detect this situation and handle it by inserting stalls

Instruction	1	2	3	4	5	6	7	8
D2 ← D1 + D3	IF	ID	A0	A1	A2	A3	MEM	WB
...		IF	ID	EX	MEM	WB		
D2 ← MEM [R0+4]			IF	ID	EX	MEM	WB	

- Clearly bad
  - Solution: Stage ID introduces 2 stalls at the last instruction

CONCLUSION

- Instruction pipeline is a nice, beautiful concept
- Difficulties:
  - Hazards / data dependencies
  - Exceptions
  - Multi-cycle operations

→ The longer the pipeline is, the more serious these difficulties are