IMPLEMENTING A PIPELINED CPU

Unit 2

1

PIPELINED Y86 IMPLEMENTATION

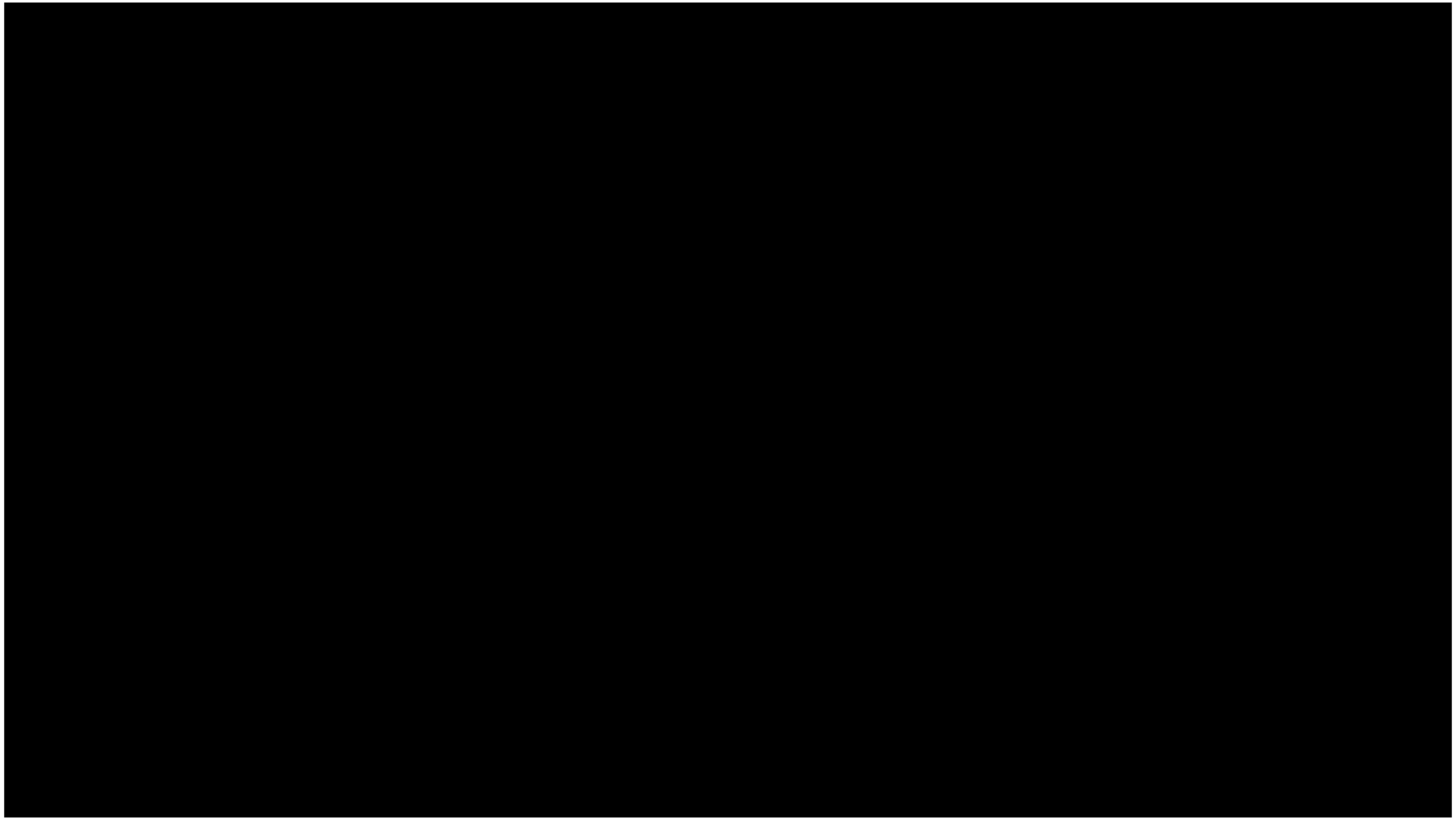
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

MOTIVATION

In a sequential Y86 implementation

Instr 1:	F	D	E	M	W							
Instr 2:						F	D	E	M	W		
Instr 3:											F	D

HUMAN PIPELINE



PIPELINE COMPUTATION

- Represent each stage as a module (fetch, decode, …)
  - Modules are ordered along the flow of computation
  - One module's output is the input of the next module
- Turn each module into a pipeline stage
  - Add pipeline registers before every stage except the 1st
  - These store the inputs for that stage
  - Stages execute in parallel working on different instructions
- As we will see later this introduces new problems

PIPELINE STAGES

How many stages should a pipeline have?

- If it has too few stages…
  - We are not exploiting the parallelism present in the program
- If it has too many…
  - There is high overhead and complexity
  - The program may not have enough parallelism to use them well

EXAMPLES

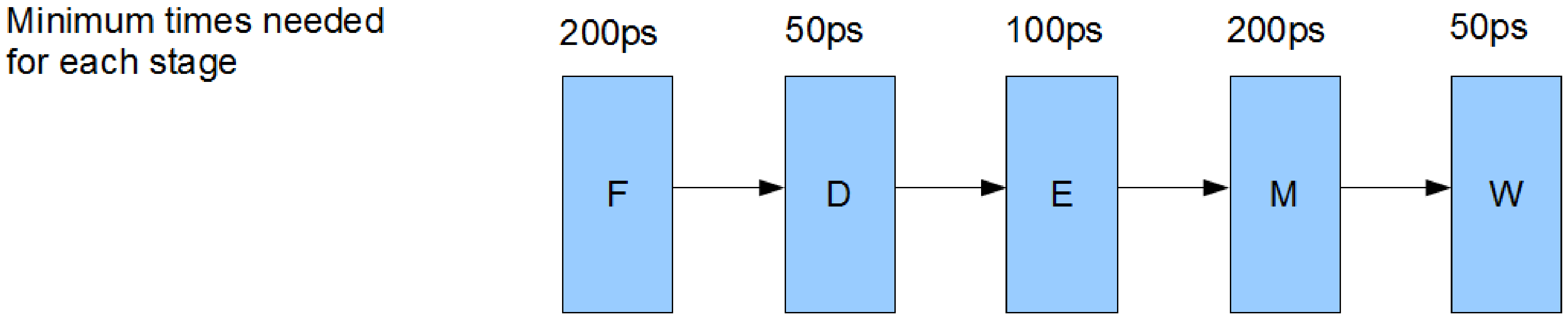
- MIPS processor (1985): first RISC processor, 5 stages
- Sparc, PowerPC processors: 9 pipeline stages
- Intel Pentium IV (late models): _______________
- Intel Core i7 processor: ___________________

MEASURING EFFICIENCY

- Latency:
  - How long it takes to execute one instruction from start to finish
  - Will usually not be reduced in a pipeline
- Throughput:
  - The number of instructions we can execute per unit of time
  - This is the only meaningful measure for pipelined CPUs

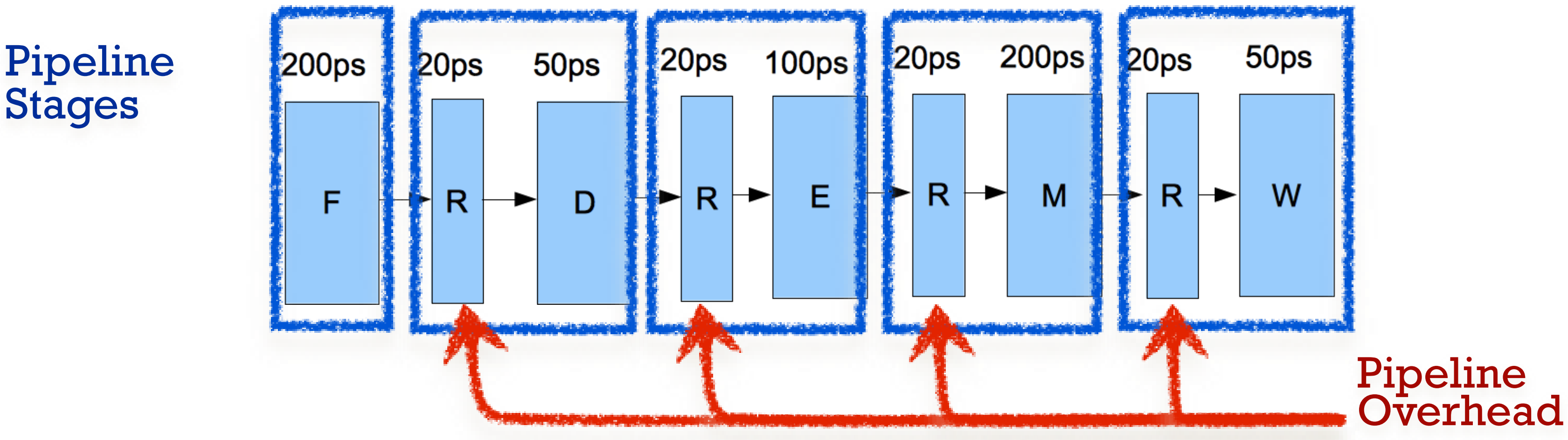
LATENCY & THROUGHPUT

- Sequential Implementation



- Latency: ______________
- Throughput: ____________

LATENCY AND THROUGHPUT



Assuming all stages are in use

- Throughput: ____________________
- Latency: ____________________

LATENCY AND THROUGHPUT

- Generalizing for pipelined CPUs:
  - Stages require an additional overhead
    - Storing and retrieving special registers
    - Latency for one instruction increases
  - New instruction can start executing once first stage is complete
    - Better throughput overall
  - All stages must run in same time slot
    - Can't move to the next instruction until slowest stage is free

PIPELINED Y86 IMPLEMENTATION

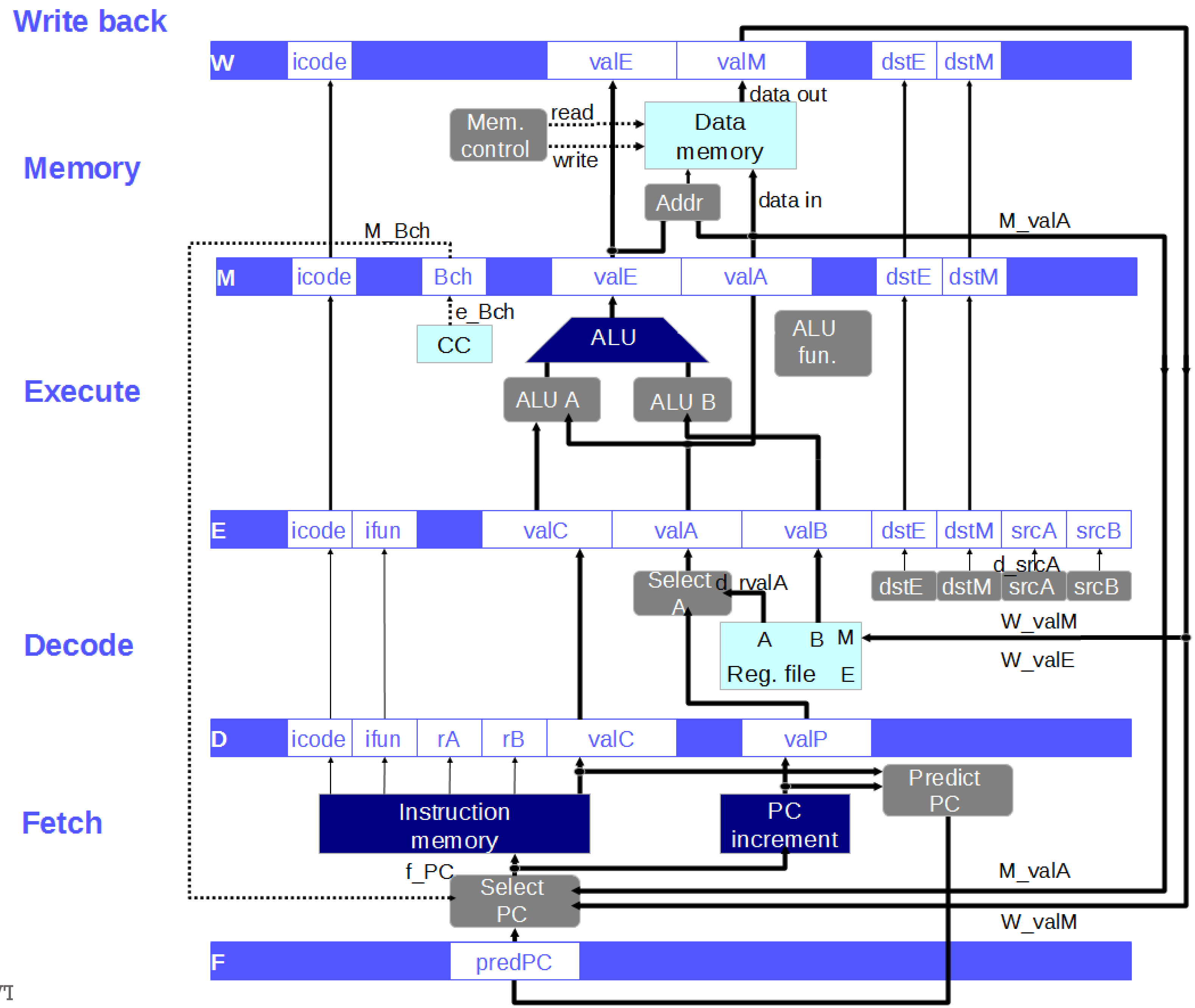
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

PIPELINED COMPUTATION

- divide execution into modules along flow of computation
  - classic RISC pipeline has five modules
  - modules arranged in order along computation flow
  - all inputs to a module must be computed by an earlier module in flow
- turn modules into pipeline stages
  - add pipeline registers between each stage
  - registers store inputs for that stage
  - each stage executes in parallel working on a different instruction
- observe that
  - a stage has less gate-propagation delay than overall circuit
  - what determines clock rate?

THE RISC PIPELINE

- How many stages?
  - enough to achieve sufficient parallelism
    - must have enough parallelism in the program
  - not too much to add undue overhead or complexity
- Which stages?
  - divide instructions into stages that instruction completes in order
  - then we can execute the stages in parallel on different instructions
- Example: rmmovq rA, D(rB)
  - what are the parts?
  - what order do they need to execute?

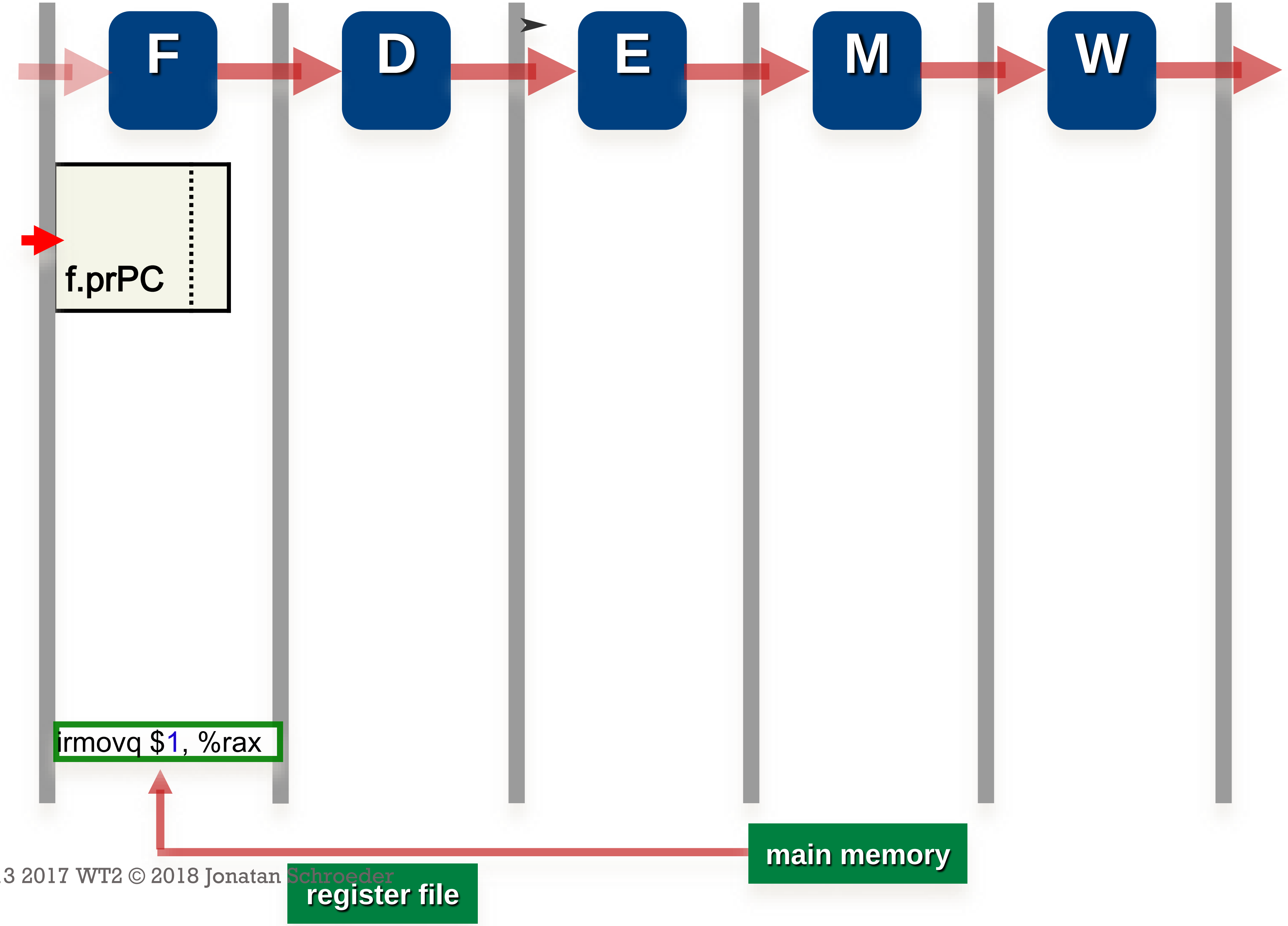


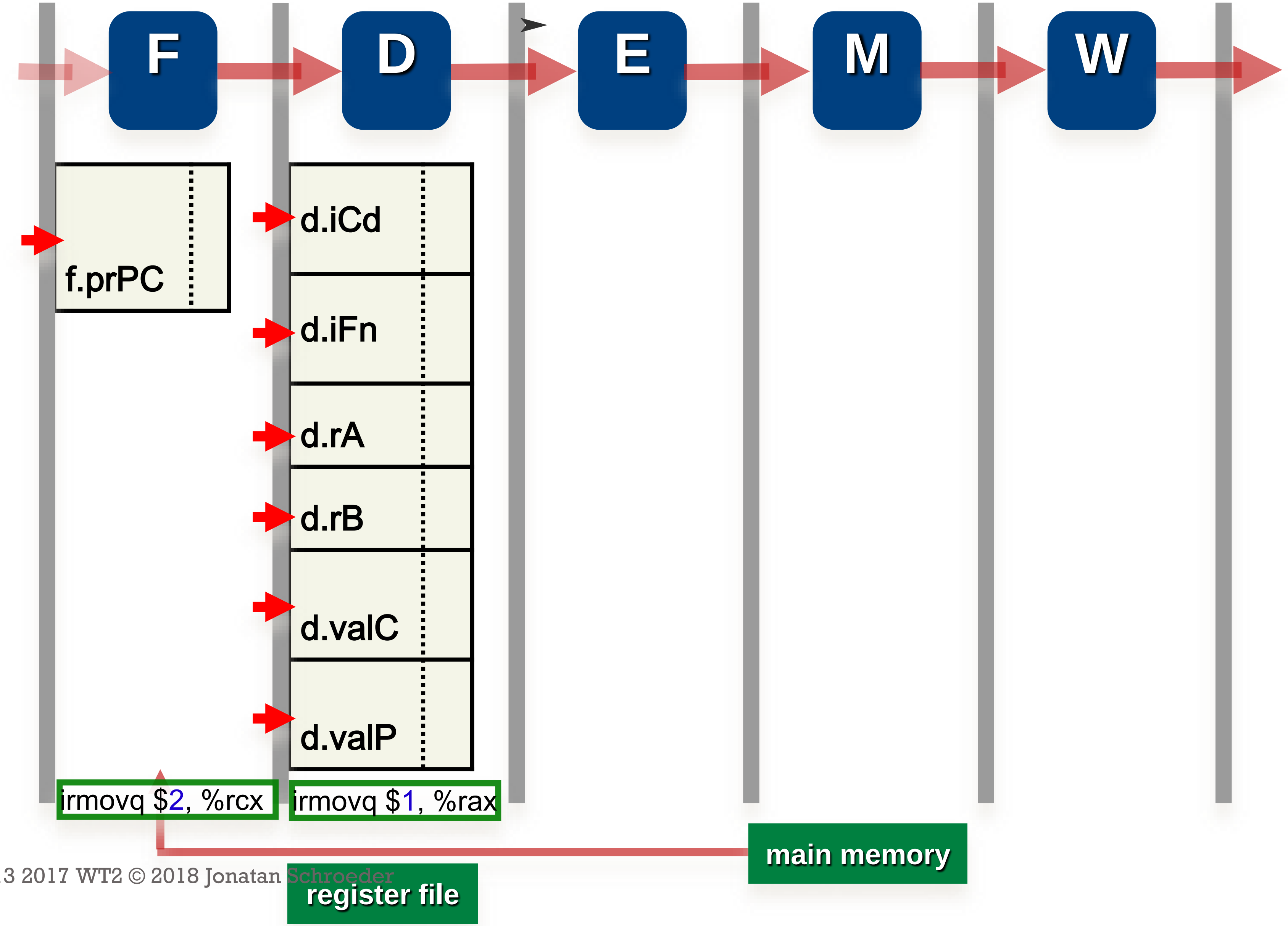
Write back
Memory
Execute
Decode
Fetch
W
icode
valE
valM
dstE
dstM
data out
Mem. control
read
write
Data memory
Addr
data in
M_valA
M_Bch
M
icode
Bch
valE
valA
dstE
dstM
e_Bch
CC
ALU
ALU fun.
ALU A
ALU B
E
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
Select A
d_rvalA
dstE
dstM
srcA
srcB
W_valM
A
B
M
Reg. file
E
W_valE
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
M_valA
Select PC
W_valM
F
predPC

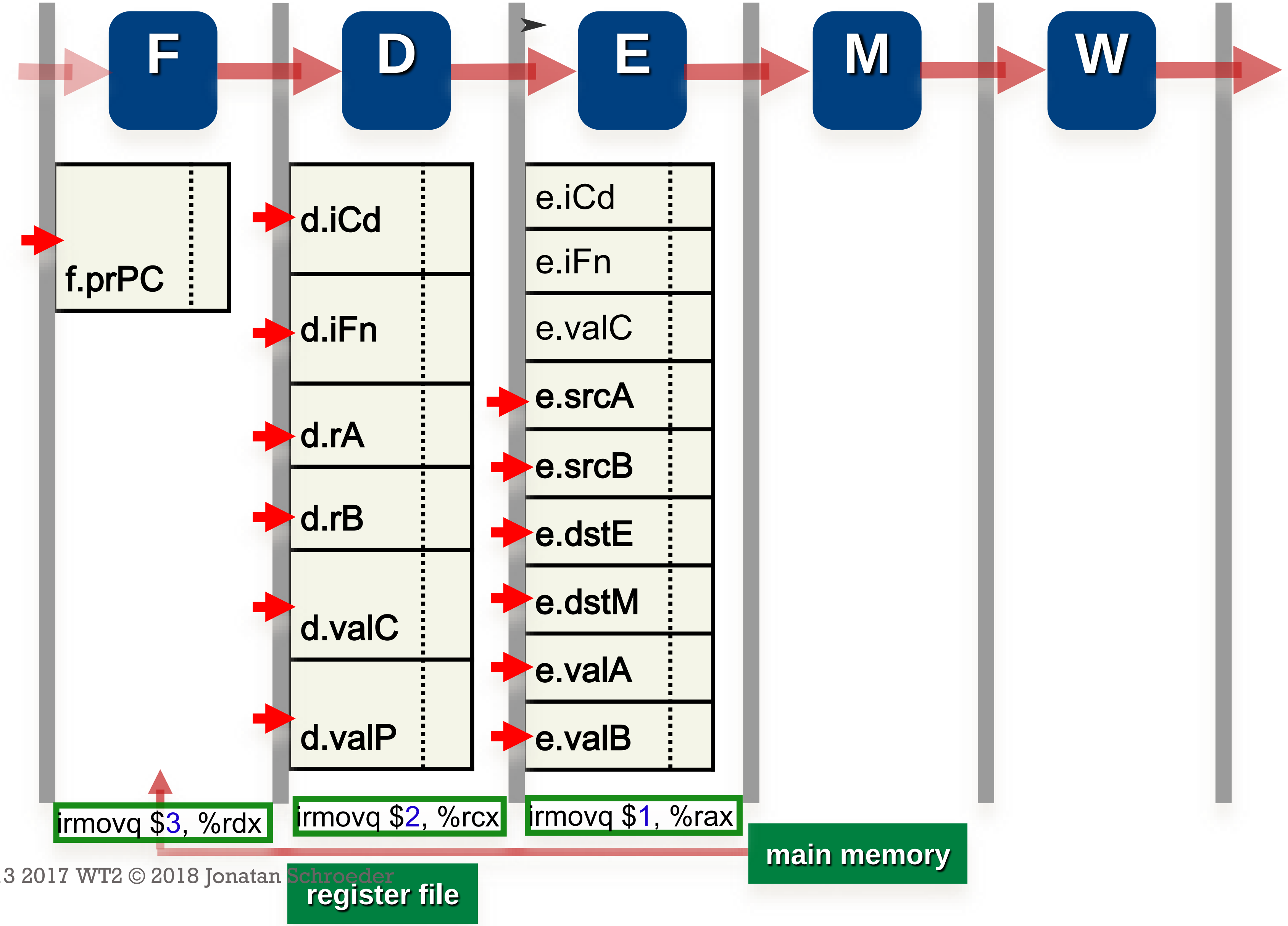
AN EXAMPLE PROBLEM

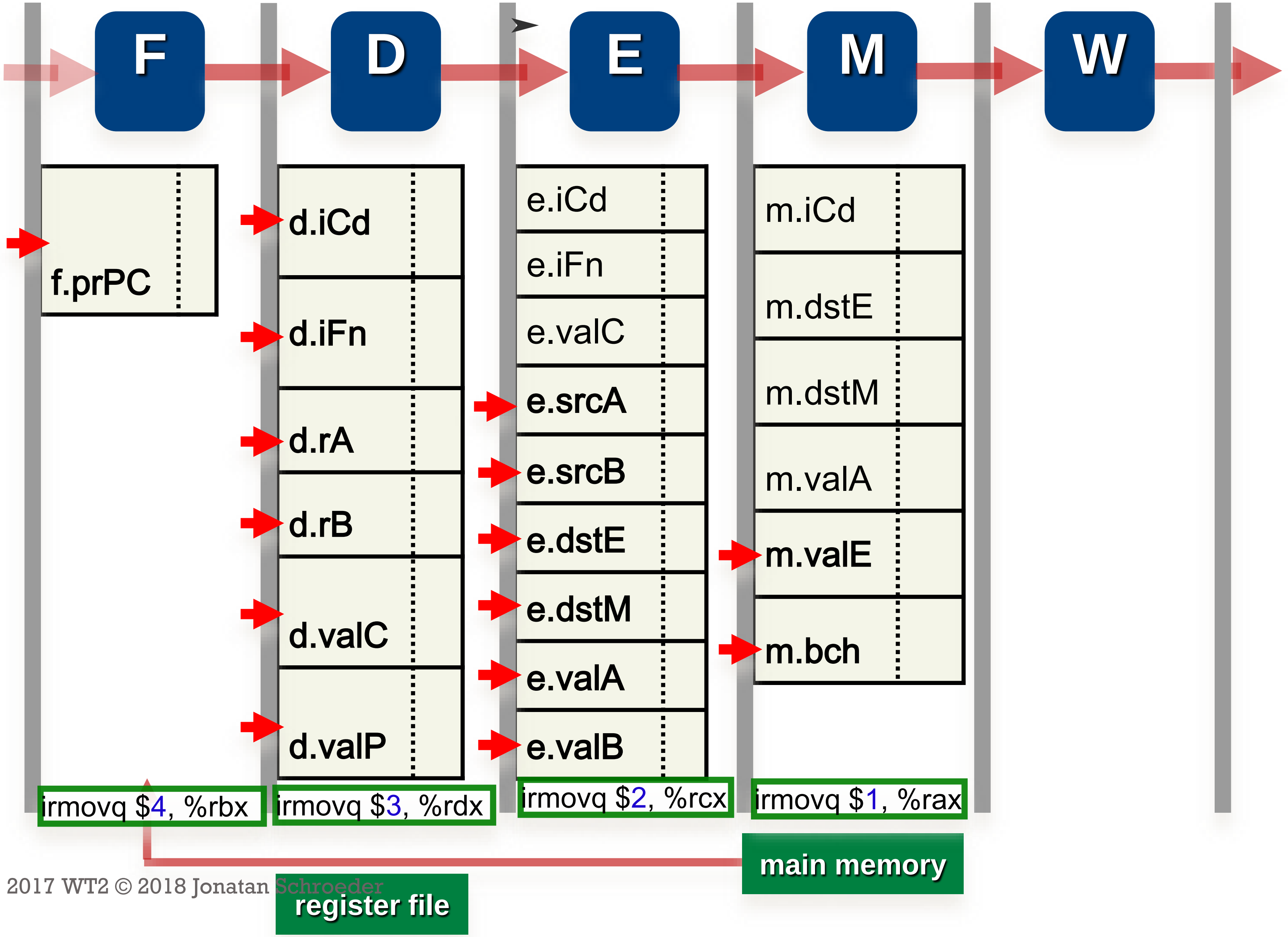
- What will each pipeline register contain when the last instruction in this sequence is entering the Fetch stage?

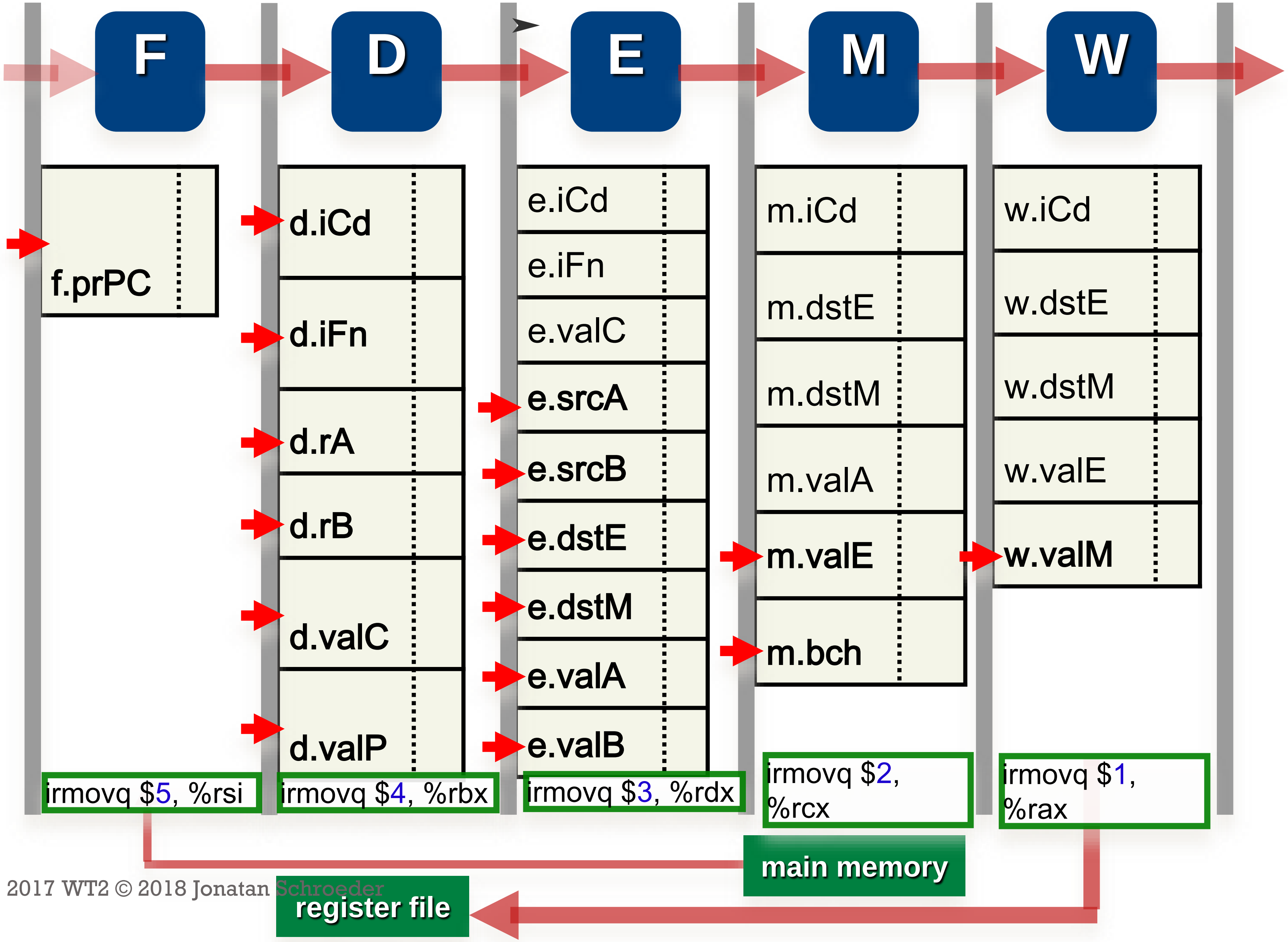
```
.pos 0x100
    irmovq $1, %rax
    irmovq $2, %rcx
    irmovq $3, %rdx
    irmovq $4, %rbx
    irmovq $5, %rsi
```


F
D
E
M
W
f.prPC
irmovq $1, %rax
main memory
register file


F
D
E
M
W
f.prPC
d.iCd
d.iFn
d.rA
d.rB
d.valC
d.valP
irmovq $2, %rcx
irmovq $1, %rax
main memory
register file


F
D
E
M
W
f.prPC
d.iCd
d.iFn
d.rA
d.rB
d.valC
d.valP
e.iCd
e.iFn
e.valC
e.srcA
e.srcB
e.dstE
e.dstM
e.valA
e.valB
irmovq $3, %rdx
irmovq $2, %rcx
irmovq $1, %rax
main memory
register file


F
D
E
M
W
f.prPC
d.iCd
d.iFn
d.rA
d.rB
d.valC
d.valP
e.iCd
e.iFn
e.valC
e.srcA
e.srcB
e.dstE
e.dstM
e.valA
e.valB
m.iCd
m.dstE
m.dstM
m.valA
m.valE
m.bch
irmovq $4, %rbx
irmovq $3, %rdx
irmovq $2, %rcx
irmovq $1, %rax
main memory
register file


F
D
E
M
W
f.prPC
d.iCd
d.iFn
d.rA
d.rB
d.valC
d.valP
e.iCd
e.iFn
e.valC
e.srcA
e.srcB
e.dstE
e.dstM
e.valA
e.valB
m.iCd
m.dstE
m.dstM
m.valA
m.valE
m.bch
w.iCd
w.dstE
w.dstM
w.valE
w.valM
irmovq $5, %rsi
irmovq $4, %rbx
irmovq $3, %rdx
irmovq $2, %rcx
irmovq $1, %rax
main memory
register file

TERMINOLOGY

- an instruction is ***in flight*** *when*
  - it is executing in the pipeline
- an instruction is ***retired*** when
  - it exits the pipeline; i.e., it completes
- on upcoming slides ...
  - ***exploiting*** and ***expressing*** parallelism
  - ***instruction-level parallelism***
  - instruction ***dependencies***
  - ***thread-level parallelism***
  - ***sequential consistency***
  - pipeline ***hazard, stall and bubble***

EXPRESSING VS EXPLOITING PARALLELISM

- Expressing parallelism: it's a mechanism where the programmer tells the system that two pieces of code can execute in parallel
- Exploiting parallelism: it's the system actually executing two pieces of code in parallel
- How do you express parallelism in C or Java?
- Is this mechanism useful for expressing ILP?

EXPLOITING INSTRUCTION-LEVEL PARALLELISM

The problem with instruction-level parallelism is:

- Programming languages like C, C++ and Java are based on the sequential consistency model:
  - The effect of executing the program must be the same as if instructions were executed one by one in the order they are written
- Programmers write code without thinking about parallelism
  - Example:
    a = b + c;   d = a + 1;   e = f + g;
    Can be rewritten as:
    a = b + c;   e = f + g;   d = a + 1;
- Compilers must find this on their own

PIPELINED Y86 IMPLEMENTATION

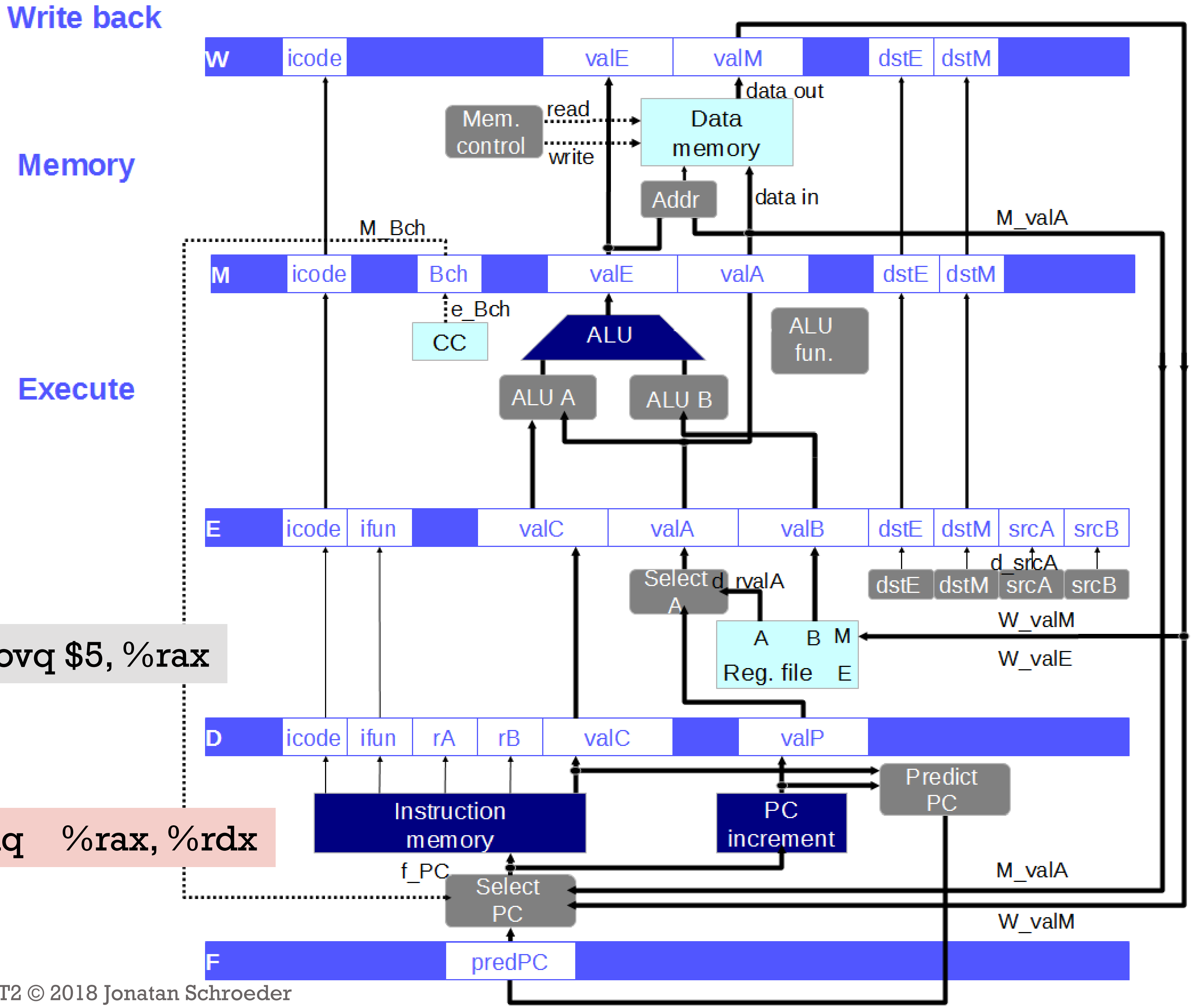
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

PIPELINE CONSIDERATIONS

- Consider this code:

```
irmovq $5, %rax
addq   %rax, %rdx
```

- At what stage is irmovq in when addq needs %rax?
  - At what stage is the value of %rax needed in addq?
  - At what stage is the value of %rax updated in irmovq?

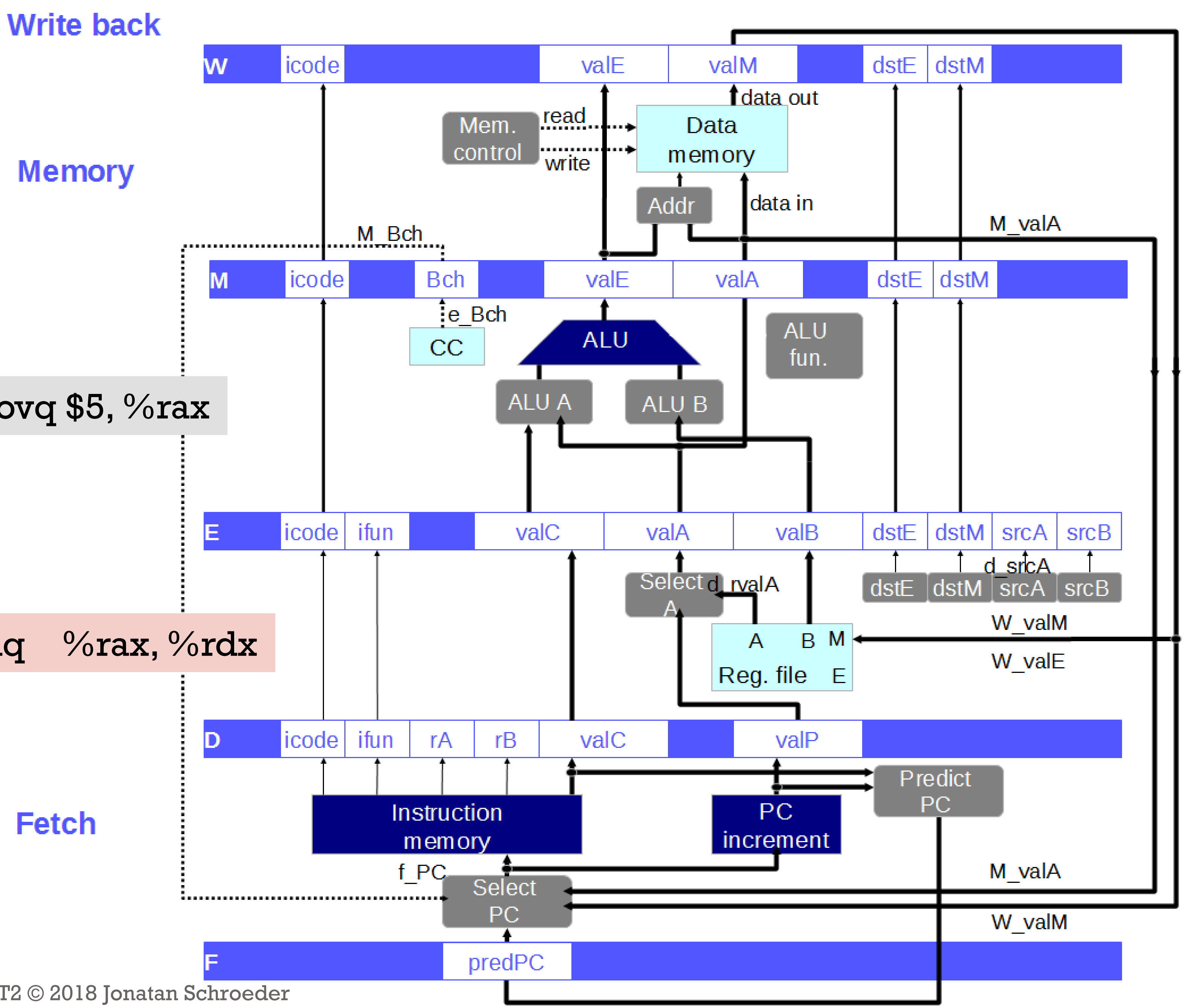


irmovq $5, %rax

addq %rax, %rdx

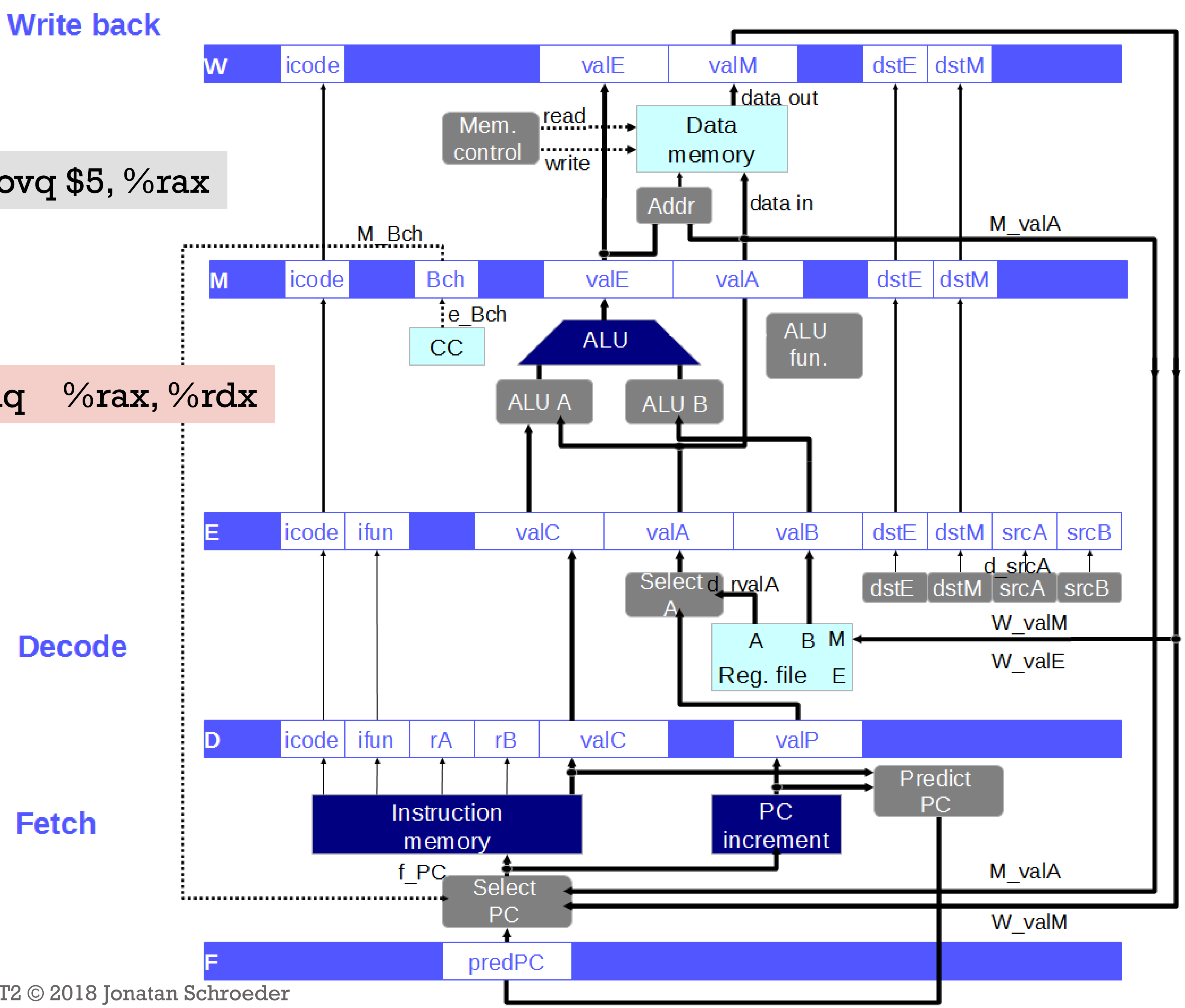
irmovq $5, %rax

addq %rax, %rdx



irmovq $5, %rax

addq %rax, %rdx



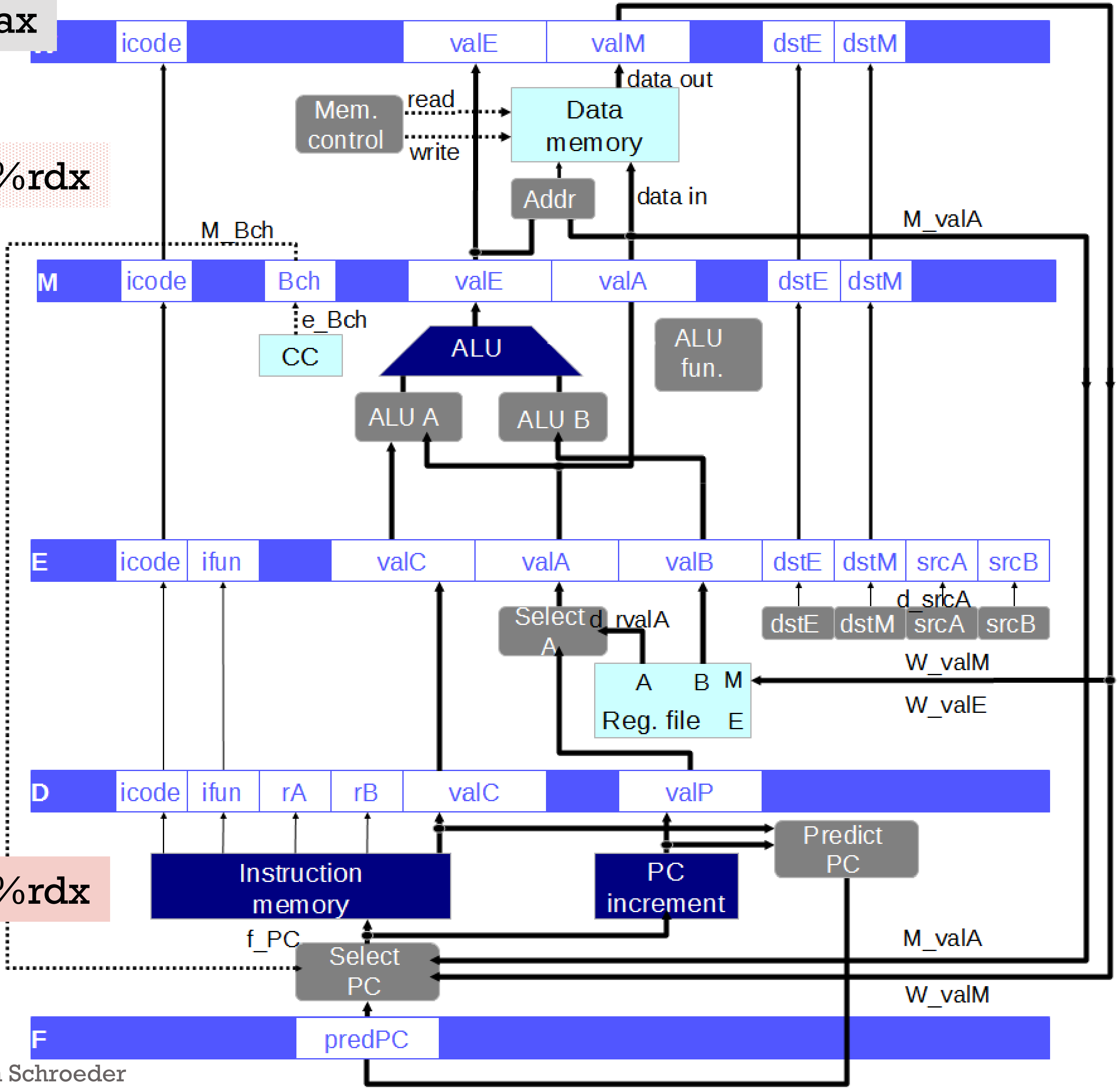
irmovq $5, %rax

addq %rax, %rdx

Execute

Decode

addq %rax, %rdx



INSTRUCTION-LEVEL PARALLELISM

- Pipeline requires some parallelism
  - multiple in-flight instructions run partly at the same time
  - may even run out-of-order
- However, the program **must** have the same output as if instructions were executed sequentially.

DEPENDENCIES CONSTRAIN PARALLELISM

- Execution of 2 or more instructions has to proceed in strict time order
  - Must be able to produce the same output as if one instruction were completely executed before the next instruction is started
- If no dependency, execution order doesn't matter
- Compilers try to expose as much instruction-level parallelism as they can
  - By separating dependent instructions
  - By grouping them with others on which they don't depend

DEPENDENCIES

- Example: how can we rewrite the following code to expose more parallelism?

```
addq %rax, %rbx
iaddq $5, %rbx
addq %rdx, %rcx
iaddq $9, %rcx
addq %rsi, %rdi
iaddq $3, %rdi
```

COMPILERS & INSTRUCTION PARALLELISM

- Compilers can reorder instructions to expose as much instruction-level parallelism as possible.
- However they cannot know every detail of the processor's pipeline (e.g. later Pentium IV's had more stages than earlier ones).
  - How many instructions should be kept between dependencies?
- So the **pipeline** must handle all dependencies correctly

DEPENDENCY TYPES

- Data dependencies
  - Involve dependencies between registers
  - Example: one instruction needs a register that is written by a previous instruction
- Control dependencies
  - Involve the flow of control (changes in PC)
  - Example: conditional jumps

CLASSIFYING DATA DEPENDENCIES

- if A and B are instructions, then $A \prec B$ means instruction B depends on instruction A
  - Causal: $A \prec B$ if B reads a value written by A
  - Output: $A \prec B$ if B writes to a location written by A
  - Alias (anti): $A \prec B$ if B writes to a location read by A

Read "$\prec$" as must happen before

TYPES OF DATA DEPENDENCY: EXERCISE

a) no dependency
b) casual dependency
c) anti-dependency
d) output dependency

1.
a = 1;
b = 2;
	3.
	a = 1;
	b = a;

2.
a = 1;
a = 2;
	4.
	a = b;
	b = 2;

TYPES OF DATA DEPENDENCY: EXERCISE

a) no dependency
b) casual dependency
c) anti-dependency
d) output dependency

1. irmovq $1, %rax rrmovq %rcx, %rax	3. rrmovq %rax, %rcx irmovq $1, %rax
2. irmovq $1, %rax rrmovq %rax, %rcx	4. irmovq $1, %rax irmovq $2, %rcx

CONTROL DEPENDENCIES

- Control dependencies determine what code is executed next
- Examples:
  - Whether a branch is taken or not taken (e.g., conditional jumps)
  - When the next instruction is obtained from a register or memory (e.g., return)
  - When an instruction writes to instruction memory (self-modifying code)
    - Will not be explored

WHEN DEPENDENCIES BECOME HAZARDS

- Dependencies are not always a problem
  - If there are enough instructions in between dependent instructions, there is no hazard
- Hazard happens when "normal" pipeline execution violates the dependency
- CPU must change its behaviour
  - By stalling instructions
  - By forwarding values from previous instructions
  - By speculating what instructions must run

PIPELINED Y86 IMPLEMENTATION

- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

LEARNING GOALS

- Explain how a pipelined processor uses stalling, data forward, and branch prediction to reduce or eliminate hazards.
- Define what is meant by a pipeline bubble and explain why/how a bubble is generated.
- For a sequence of machine language instructions, describe at an arbitrary execution point the state of the pipeline:
  - When there are no hazards
  - When hazards are handled using only stalling
  - When hazards are handled using stalling and/or data forwarding
  - When hazards are handled using stalling, data forwarding, and/or branch prediction

WHAT IS THE PROBLEM HERE?

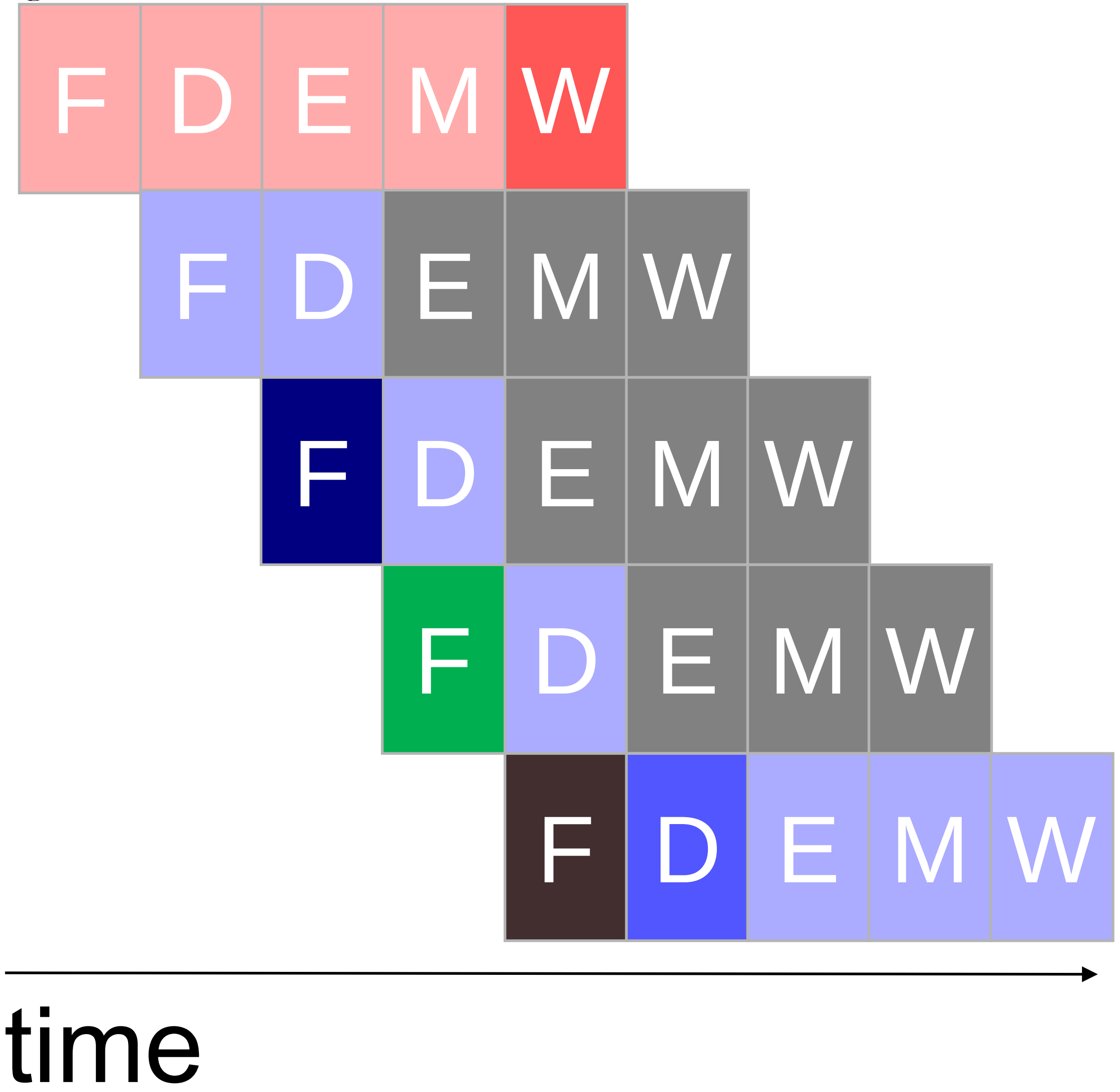
```
irmovq $1, %rax

addq %rax, %rbx

addq %rax, %rcx

addq %rax, %rdx

addq %rax, %rsi
```



STALLING

- pipeline stall: hold an instruction in a pipeline stage for an extra cycle

`irmovq $1, %rax`	F	D	E	M	W				
`addq %rax, %rbx`		F	D	D	D	D	E	M	W
`subq %rax, %rcx`			F	F	F	F	D	E	M
`xorq %rax, %rdx`							F	D	E

time →

BUBBLE

- The term pipeline bubble denotes a pipeline stage that is forced to do nothing to avoid a hazard, because of a stall in a previous stage
  - For example, if decode stalls, in the next instruction execute will have a bubble
- The bubble converts the instruction into a NOP

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**
	xorq	subq	addq	**bubble**

EXAMPLE

Fetch	Decode	Execute	Memory	Write-back
irmovq	?	?	?	?
addq	irmovq	?	?	?
subq	addq	irmovq	?	?
subq	addq	**bubble**	irmovq	?
subq	addq	**bubble**	**bubble**	irmovq
subq	addq	**bubble**	**bubble**	**bubble**
xorq	subq	addq	**bubble**	**bubble**
	xorq	subq	addq	**bubble**
		xorq	subq	addq

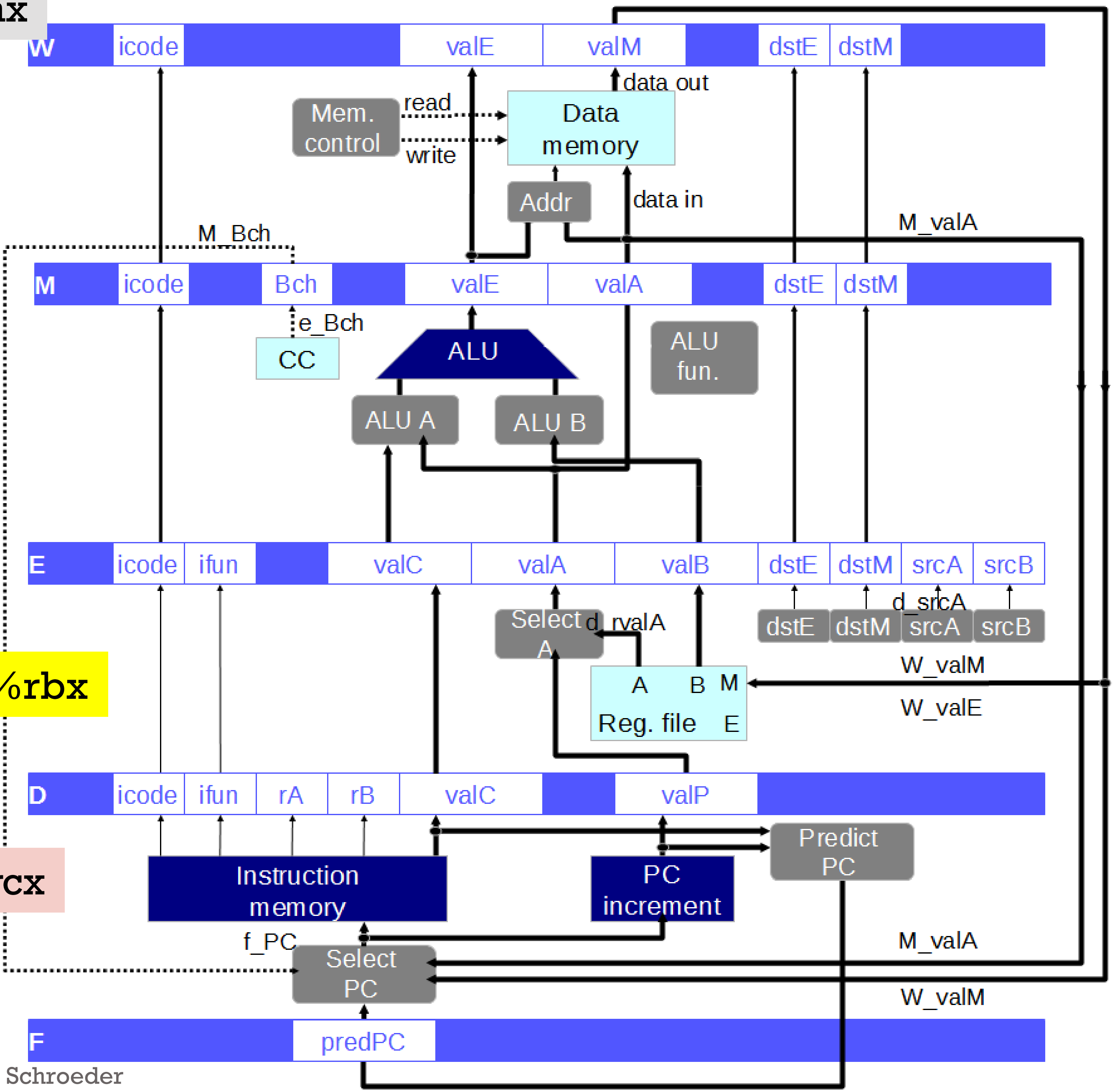
irmovq $5, %rax

Bubble/ NOP

Bubble/ NOP

addq   %rax, %rbx

subq %rax, %rcx



WHAT ABOUT ANTI AND OUTPUT DEPENDENCIES

- If instructions are executed in-order, they are not a problem
- Output dependency: first instruction's output can be ignored

```
irmovq   $1, %rax
irmovq   $2, %rax
addq     %rax, %rdx
irmovq   $1, %rax
```

Output

Causal

Anti

TYPES OF HAZARDS

For our pipelined Y86 implementation:

Anti-dependency:

addq %rax, %rdx

irmovq $1, %rax

F	D	E	M	W	
	F	D	E	M	W

Output dependency:

addq %rax, %rdx

irmovq $1, %rdx

F	D	E	M	W	
	F	D	E	M	W

As long as instructions execute in order, no problem

STALLING WORKS, BUT…

- Stalls should be avoided
  - every stall creates a pipeline bubble
  - every bubble results in a cycle when processor can't retire an instruction
  - retiring a bubble is of no value to program execution
- Bubbles decrease overall throughput
- Ideas needed to deal with
  - data dependencies
  - control dependencies

PIPE: RESOLVING HAZARDS BY STALLING SUMMARY

- data hazards
  - reading registers in decode that are written by instructions currently in execute, memory or write-back stages
  - stall instruction in decode until writer is retired
  - how many stall cycles? ______

PIPE: RESOLVING HAZARDS BY STALLING SUMMARY (CONT.)

- Conditional jump
  - At what stage do we know which PC?
  - At what stage is the PC needed?
  - how many stall cycles? ______
- Return
  - The next PC is available at the end of?
  - how many stall cycles? ______

PIPELINE CONTROL UNIT

- The pipeline-control module
  - is a hardware component (circuit) separate from the 5 stages
  - examines values across every stage
  - decides whether stage should stall or bubble

DEALING WITH HAZARDS

- Each stage register has a control input that determines what happens when the clock ticks:
  - Normal: register's new value is the input value.
  - Stall: register's new value is the same as its current value.
  - Bubble: register's new value is the same as for NOP.

PIPELINE CONTROL UNIT