IMPLEMENTING A PIPELINED CPU

Unit 2

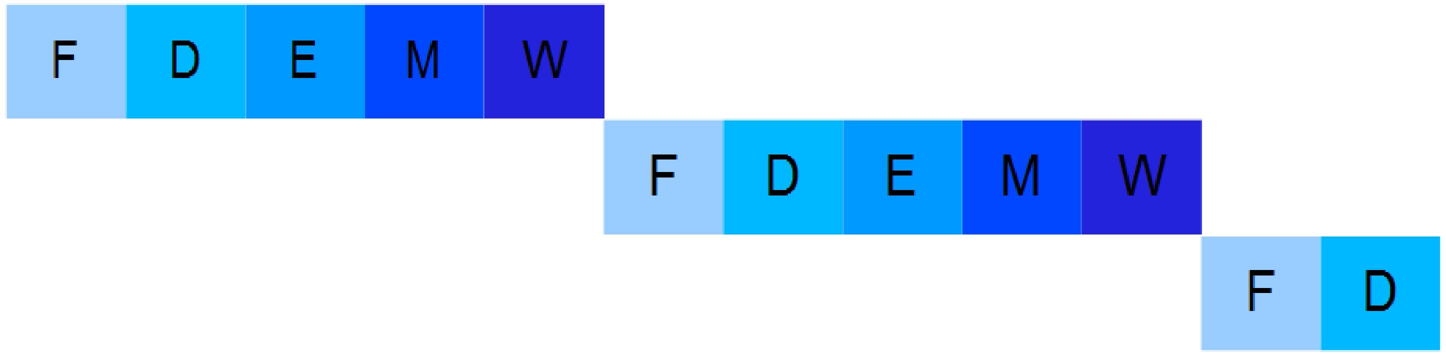
PIPELINED Y86 IMPLEMENTATION

- 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:
- Instr 2:
- Instr 3:



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: ________________

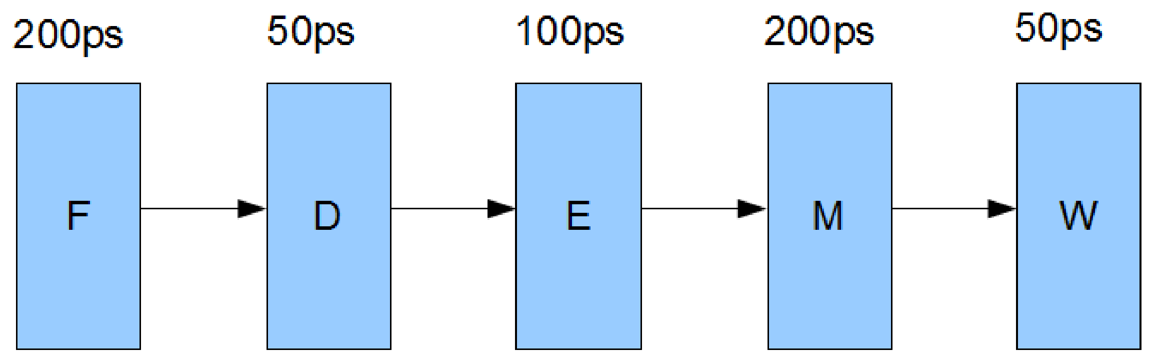
MEASURING EFFICIENCY

- 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

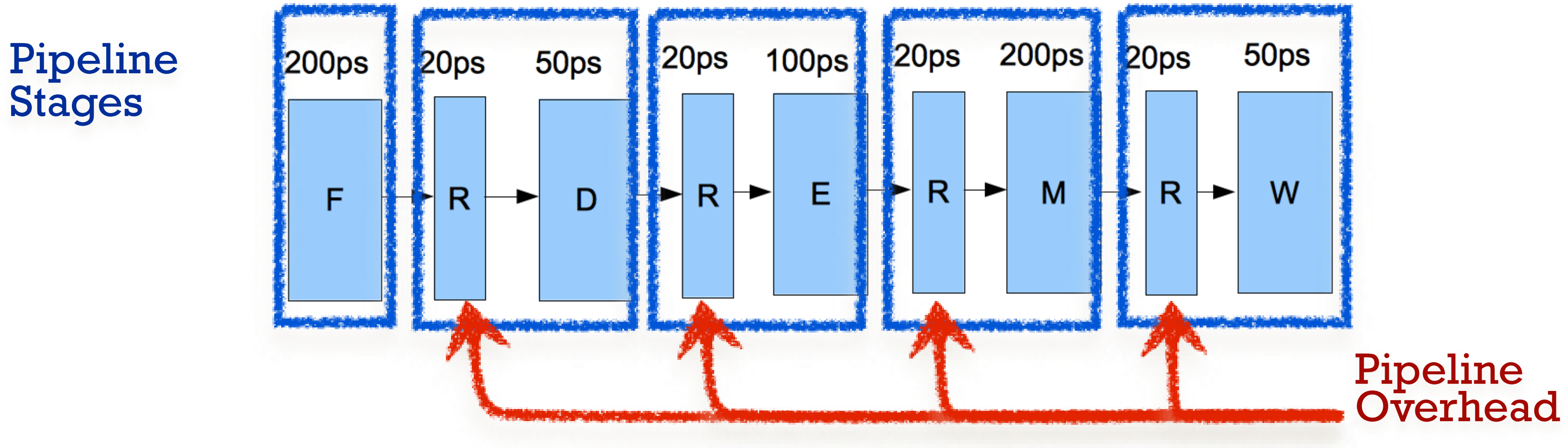
- Sequential Implementation

Minimum times needed
for each stage



- 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

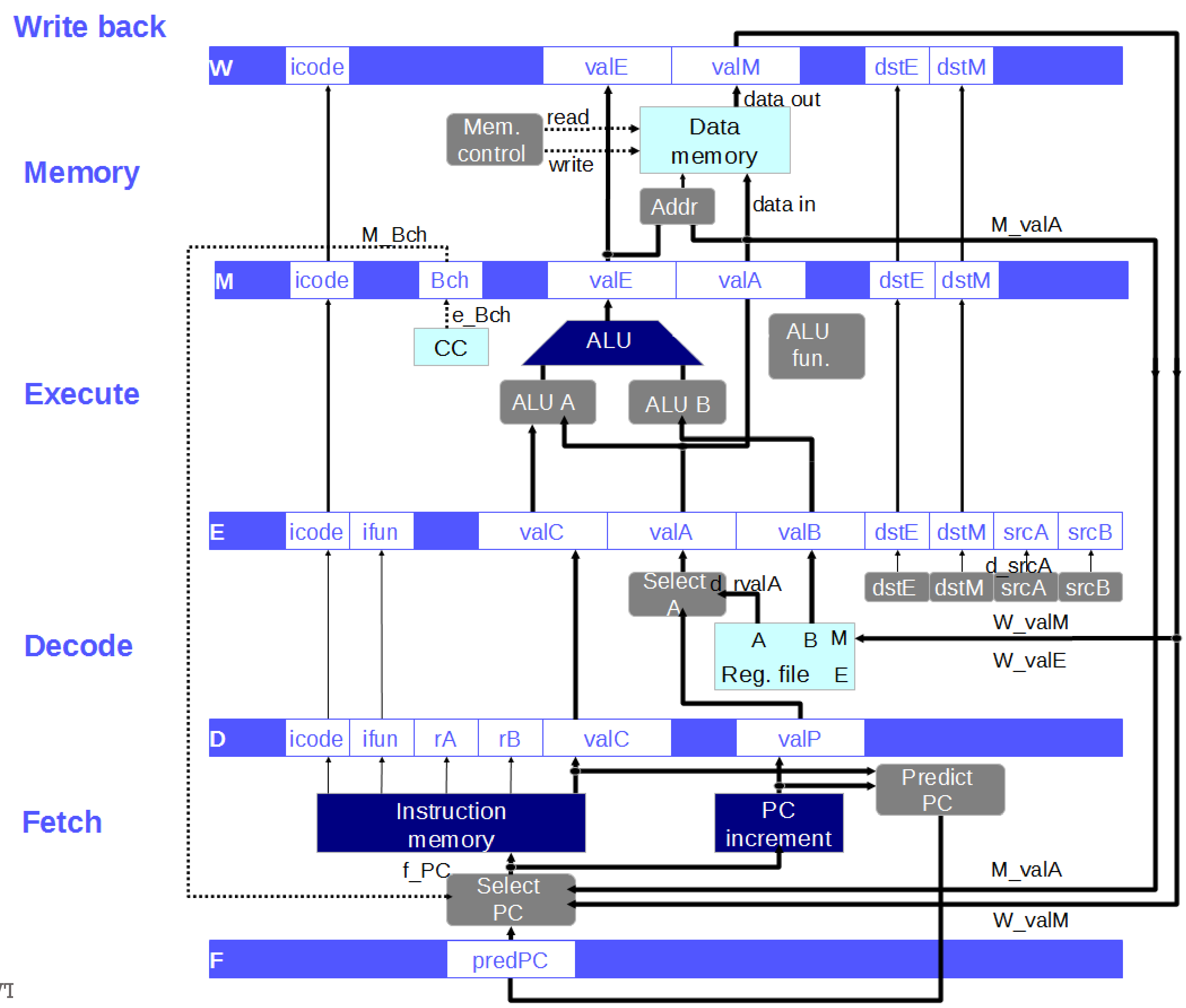
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

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

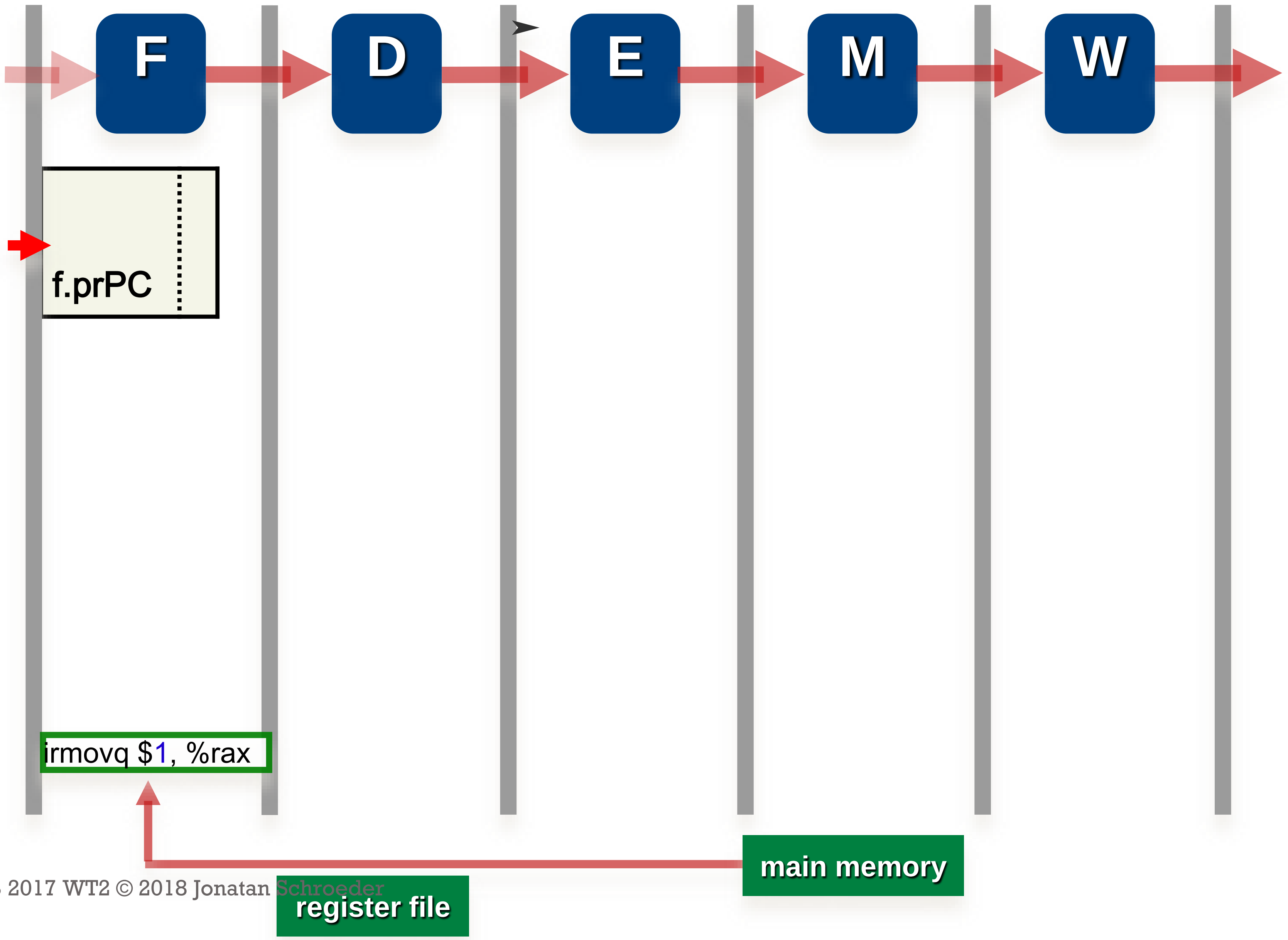
- How many stages?
  - enough to achieve sufficient parallelism
    - must have enough parallelism in the program
  - not too much to add undue overhead or complexity
- Which stages?
  - divide instructions into stages that instruction completes in order
  - then we can execute the stages in parallel 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_Bch
M_valA
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
dstE
dstM
srcA
srcB
Select A
d_rvalA
A
B
M
Reg. file
E
W_valM
W_valE
D
icode
ifun
rA
rB
valC
valP
Predict PC
Instruction memory
PC increment
f_PC
Select PC
M_valA
W_valM
F
predPC

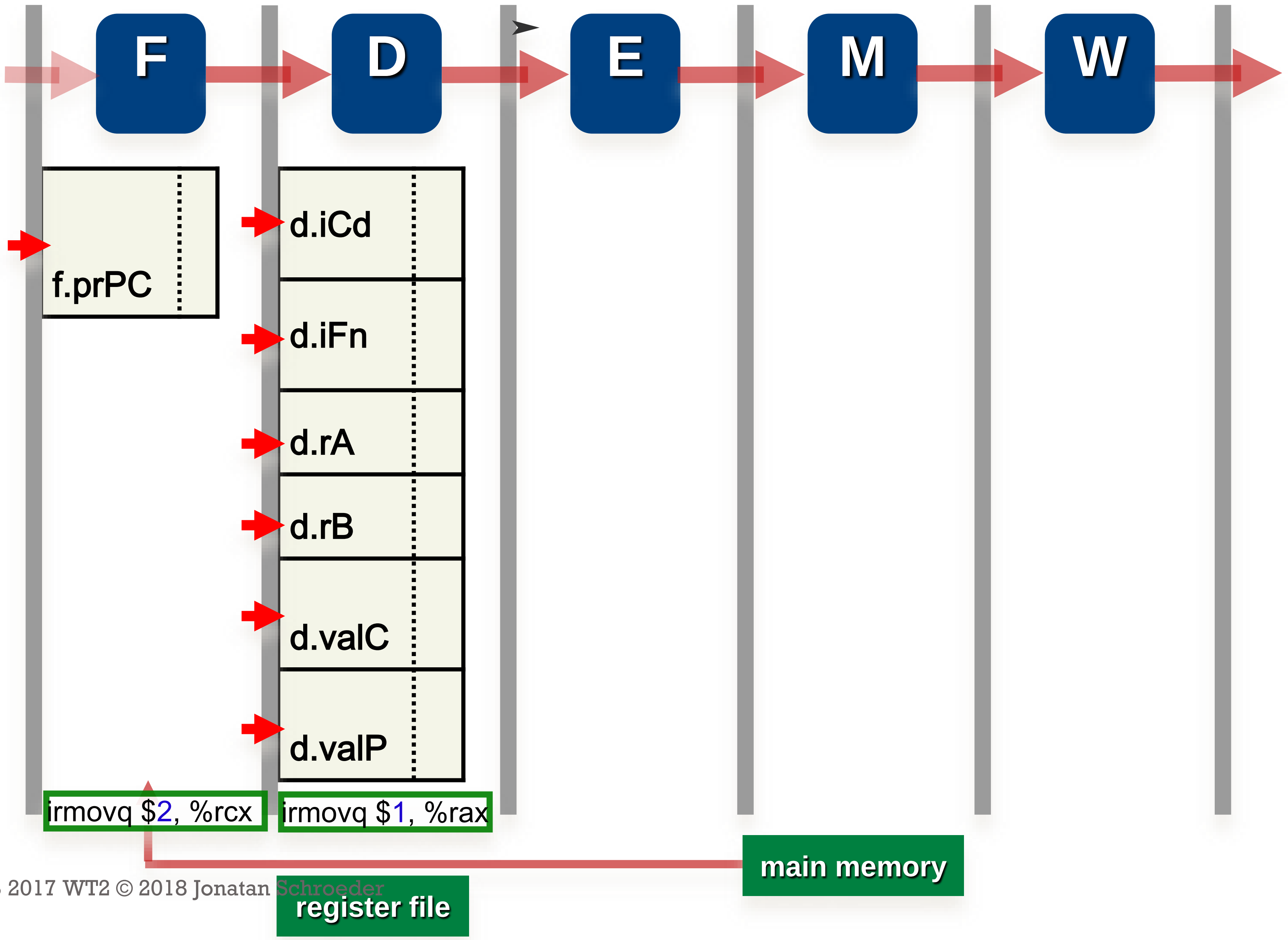
AN EXAMPLE PROBLEM

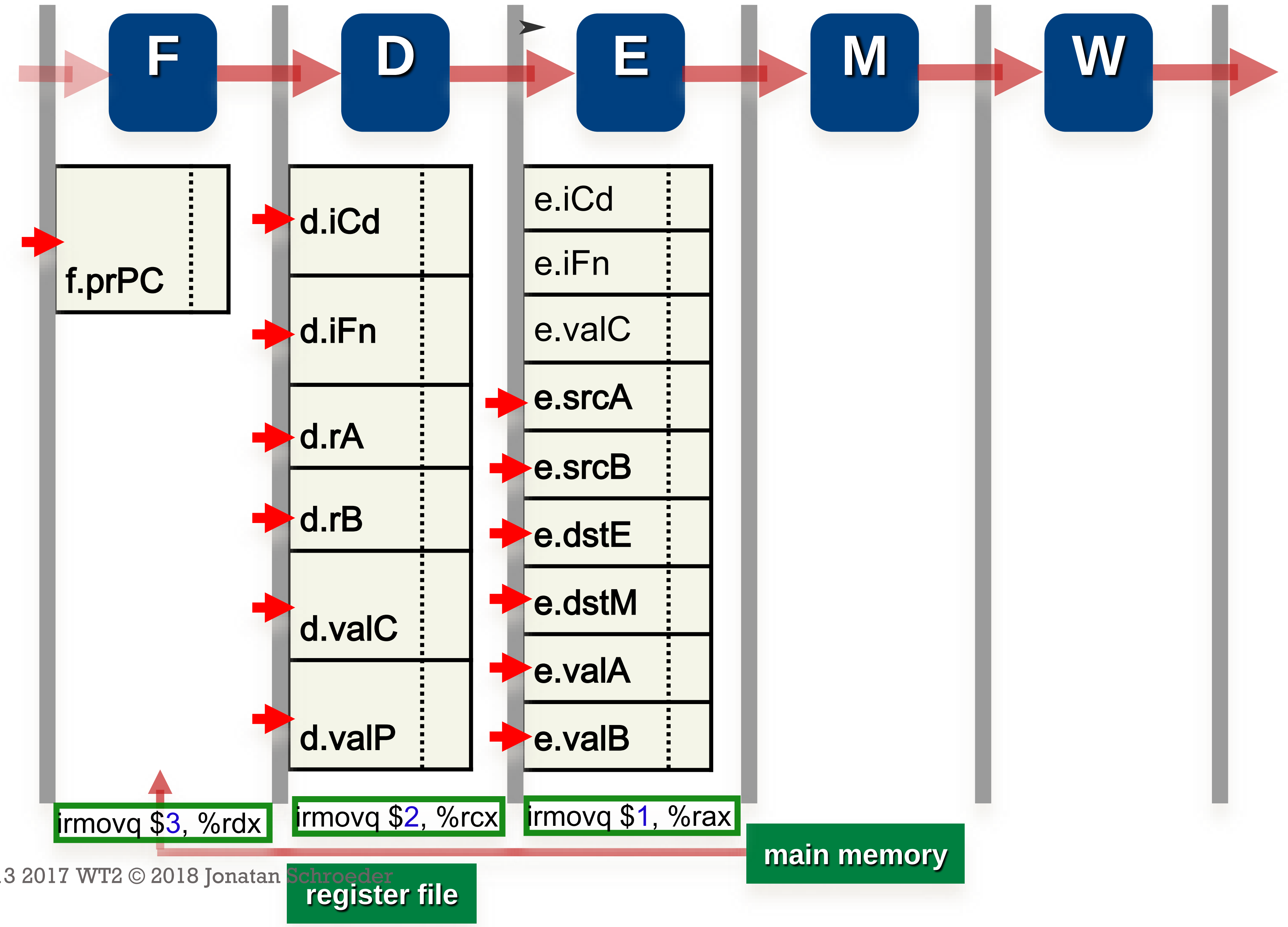
- What will each pipeline register contain when the last instruction in this sequence is entering the Fetch stage?

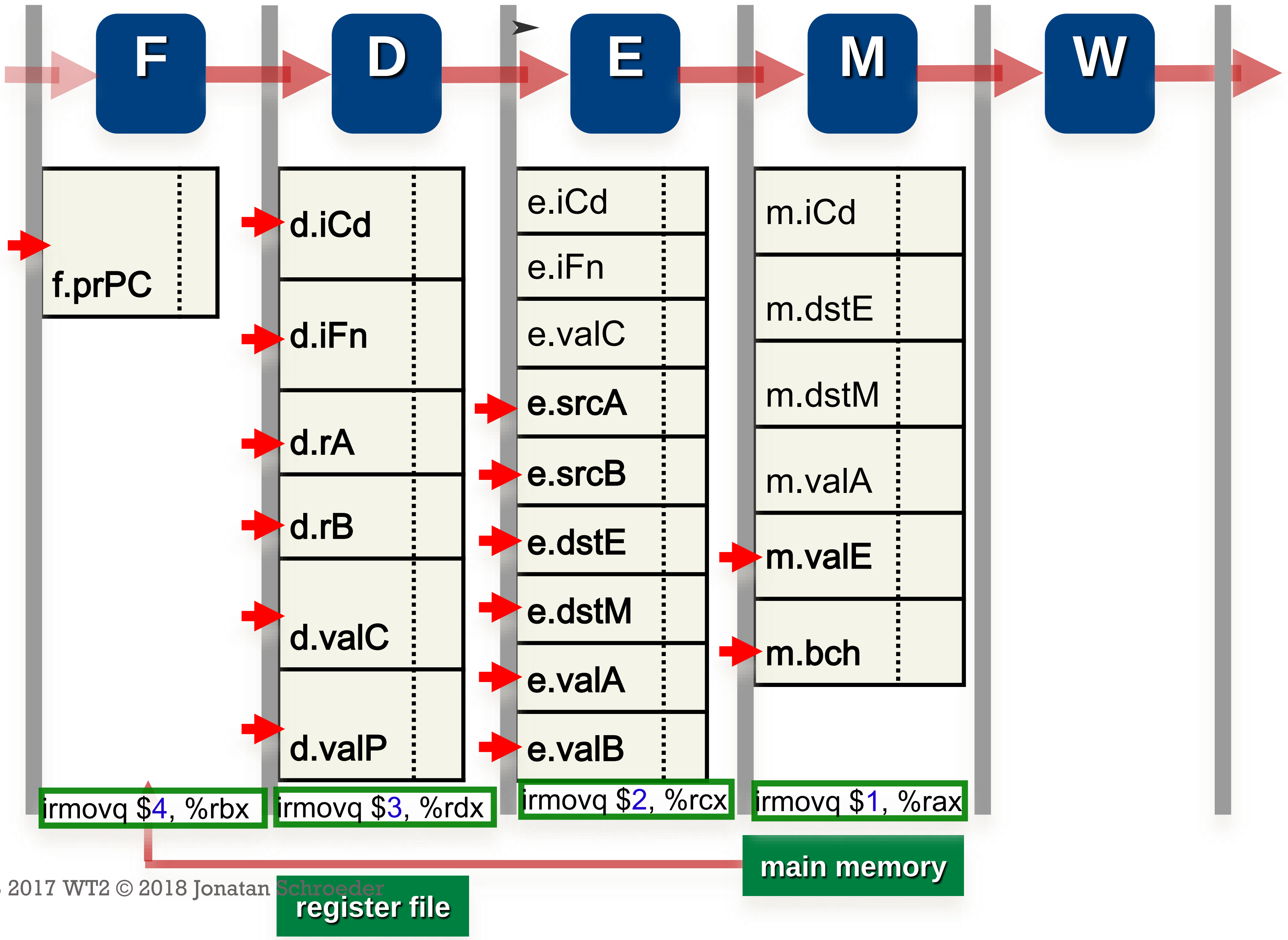
```
.pos 0x100
    irmovq $1, %rax
    irmovq $2, %rcx
    irmovq $3, %rdx
    irmovq $4, %rbx
    irmovq $5, %rsi
```

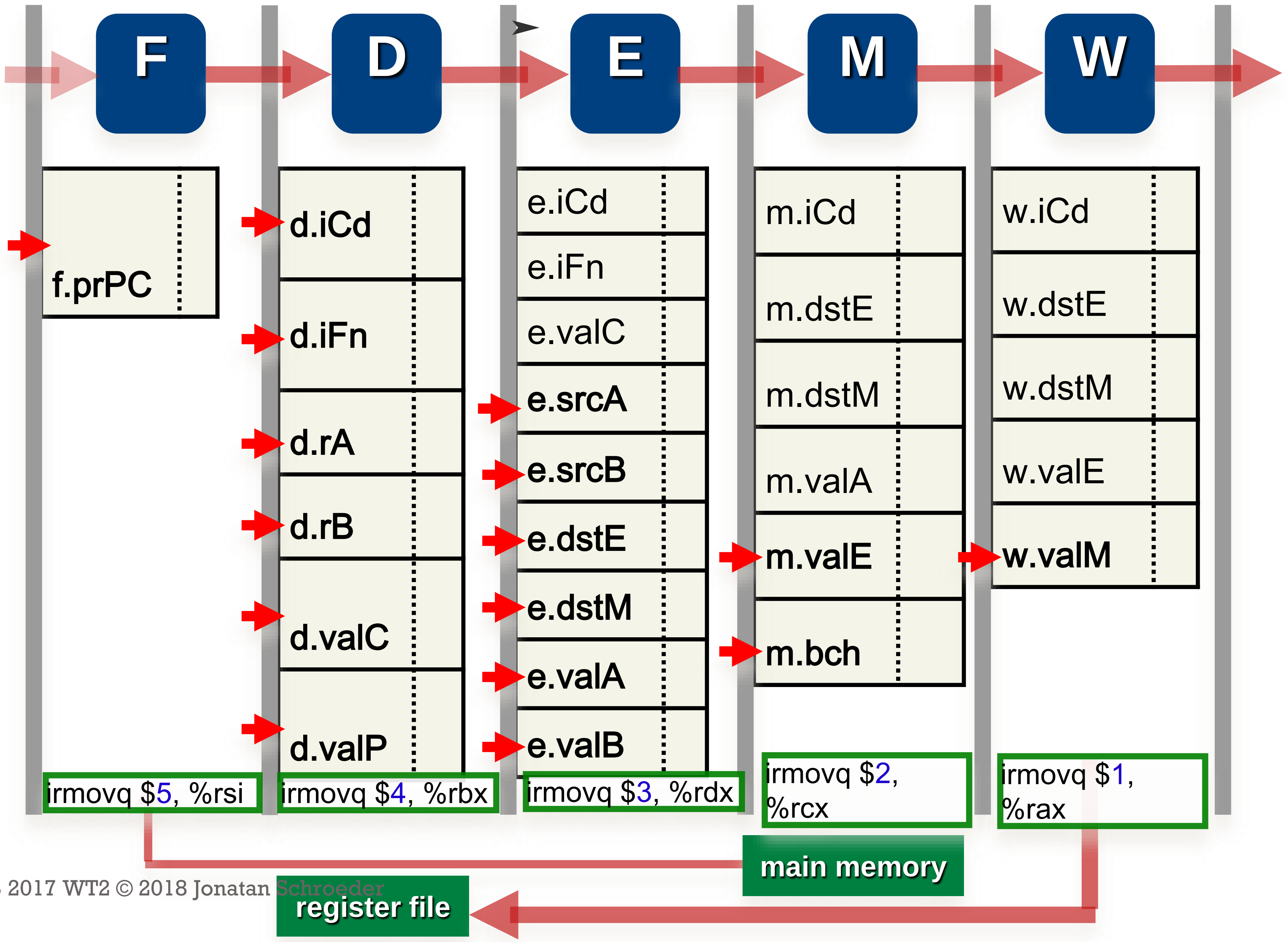


irmovq $1, %rax


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

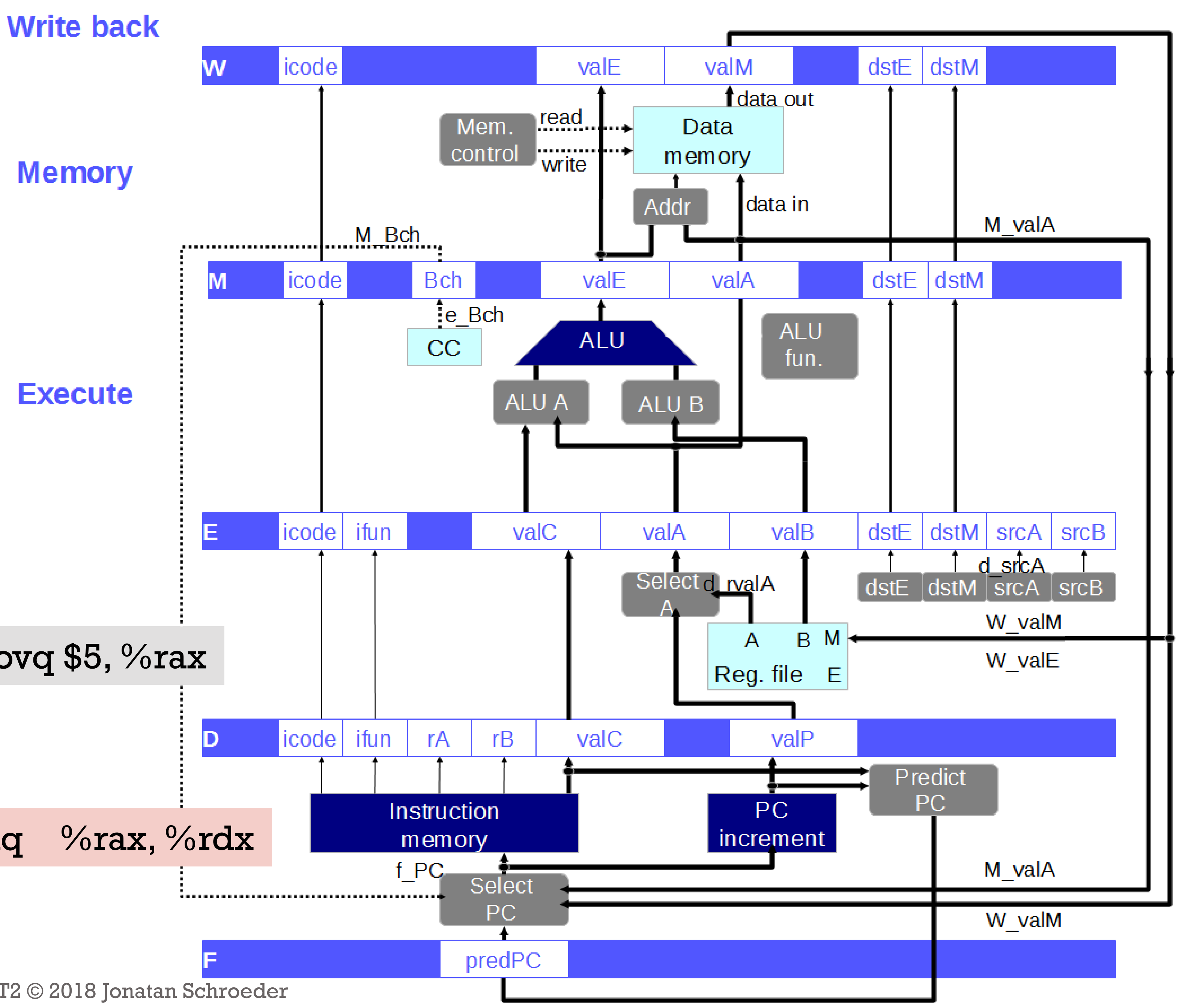
- Unit outline
  - Motivation and basic concepts
  - Initial Implementation
  - Hazards
    - Types of hazards
    - Dealing with hazards by stalling
    - Data hazards: forwarding to avoid stalling
    - Control hazards: branch prediction
    - Indirect jumps
  - Performance analysis

PIPELINE CONSIDERATIONS

- Consider this code:

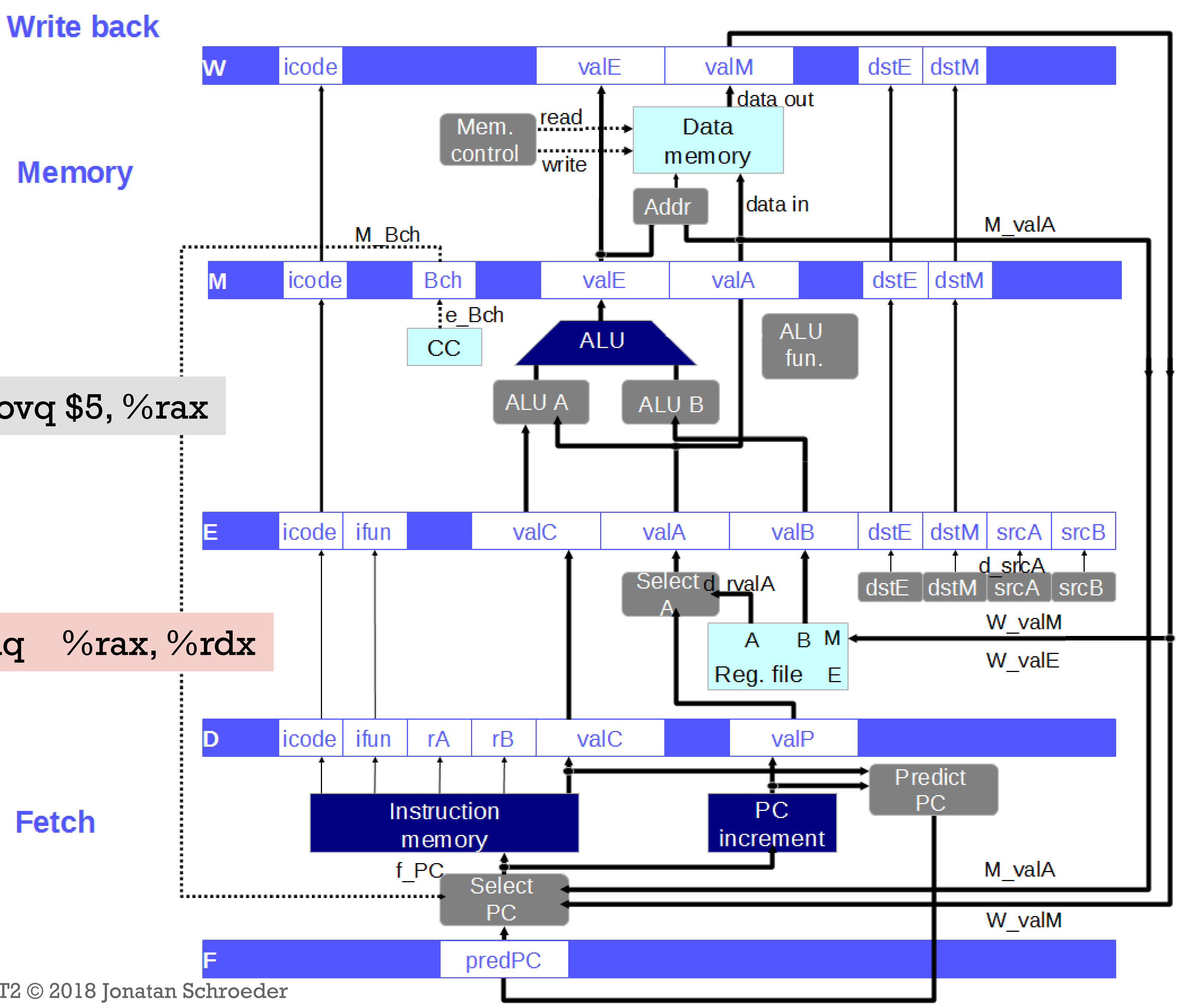
```
    irmovq $5, %rax
    addq   %rax, %rdx
```

- 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

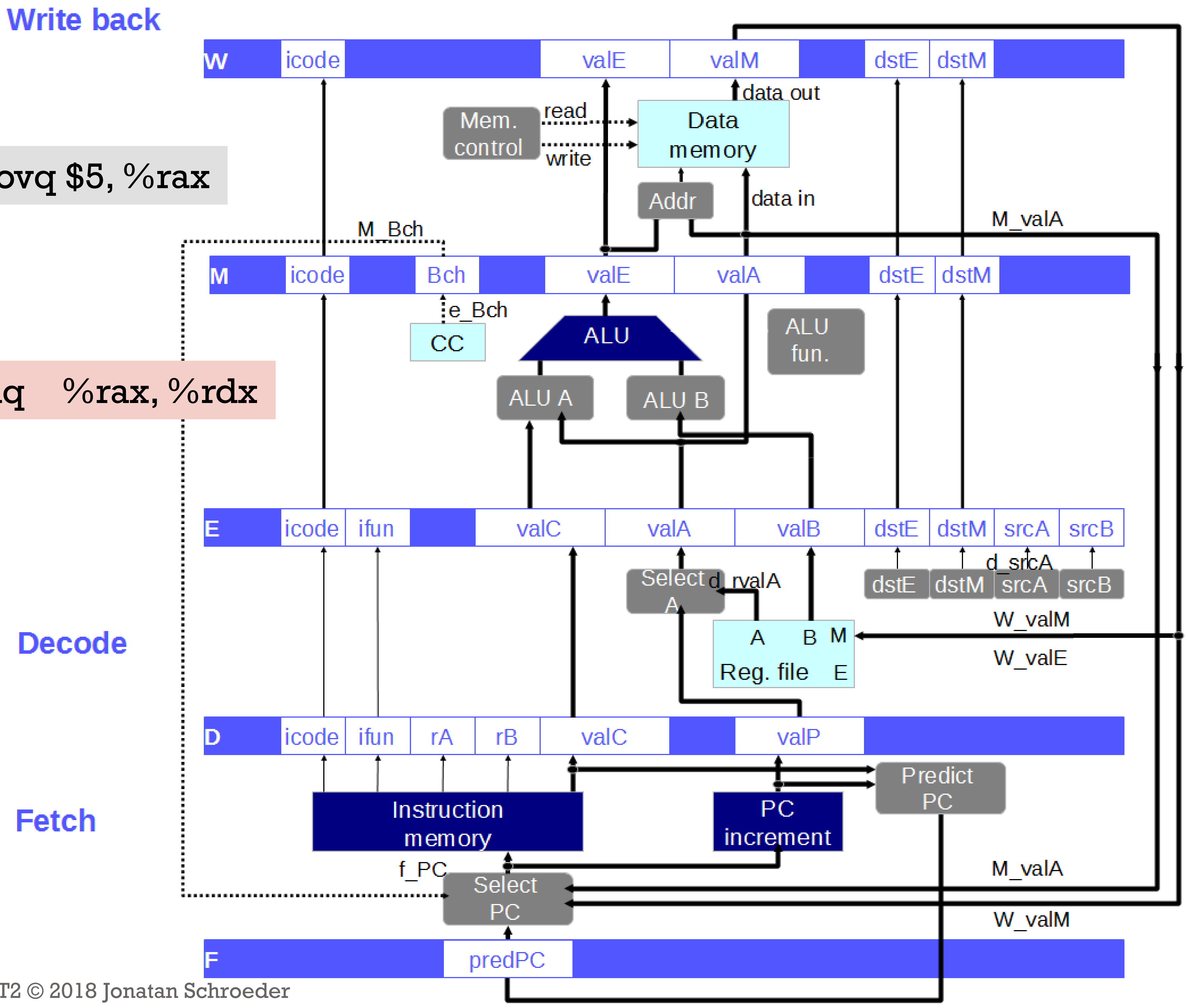


irmovq $5, %rax

addq %rax, %rdx

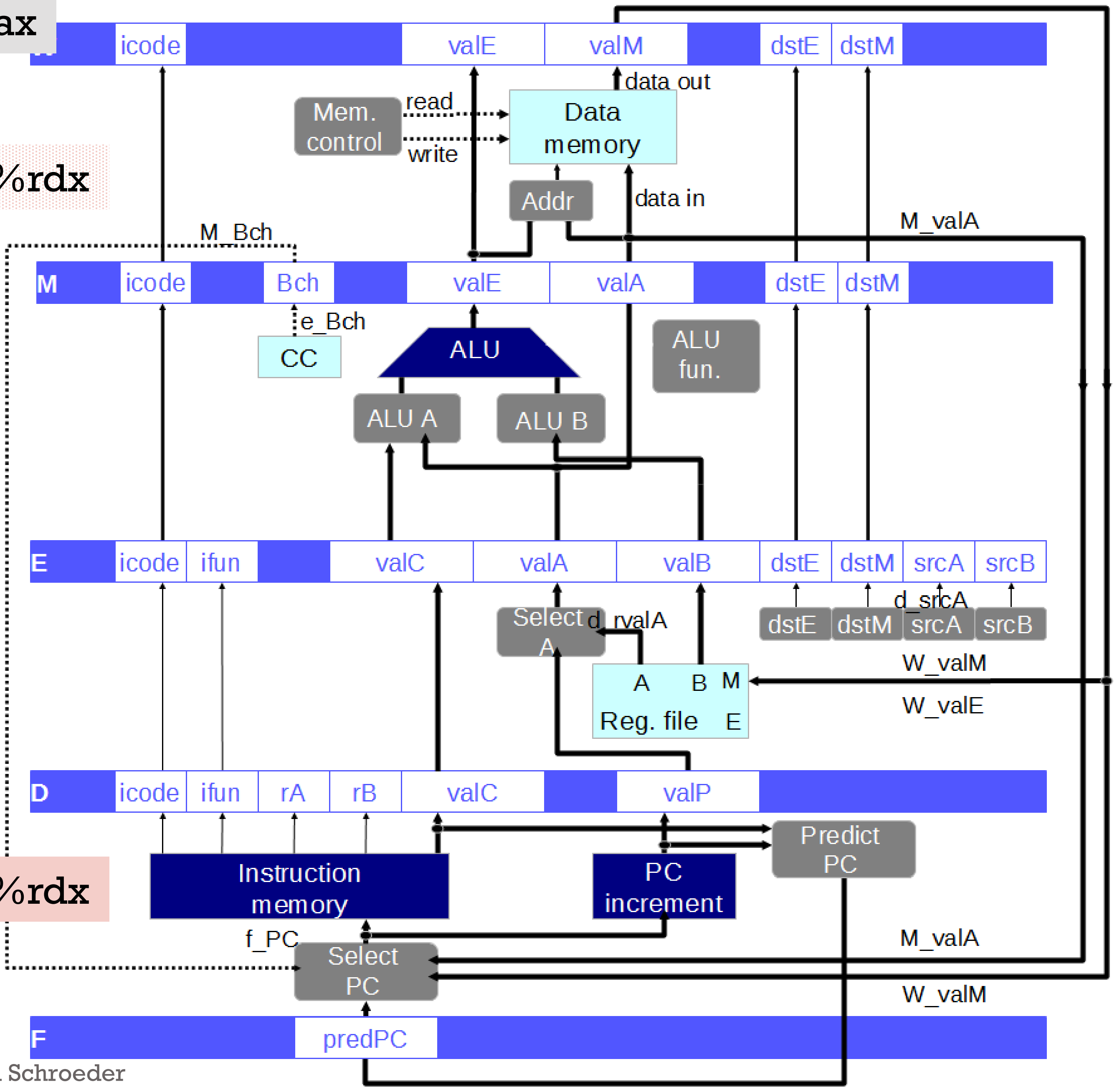
irmovq $5, %rax

addq %rax, %rdx



irmovq $5, %rax

addq %rax, %rdx



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	c) anti-dependency
b) casual 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

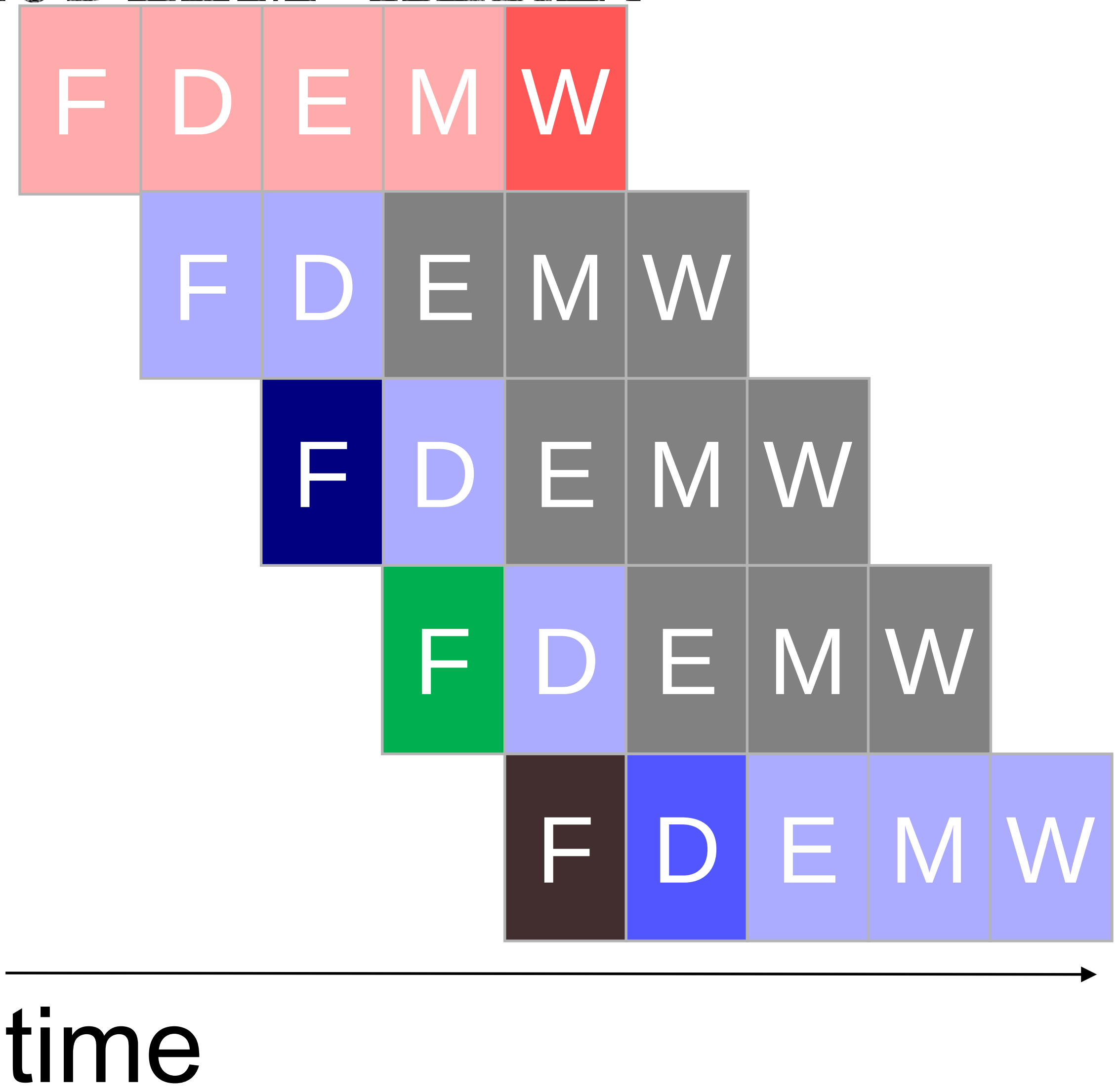
- Unit outline
- Motivation and basic concepts
- Initial implementation
- Hazards
  - Types of hazards
  - Dealing with hazards by stalling
  - Data hazards: using data forwarding to avoid stalling
  - Control hazards: branch prediction
  - Indirect jumps
- Performance analysis.

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 $\longrightarrow$

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

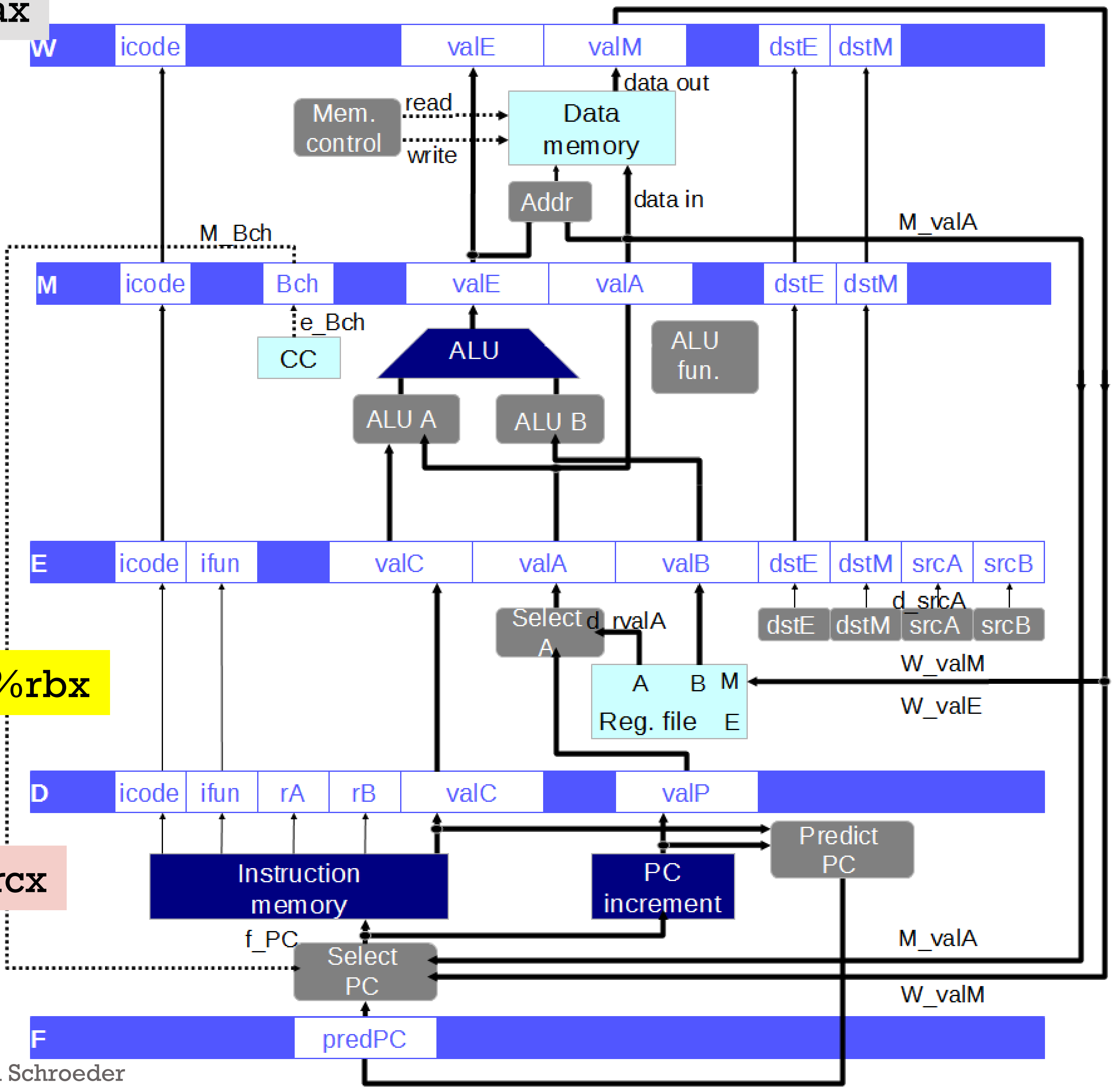
irmovq $5, %rax

Bubble/ NOP

Bubble/ NOP

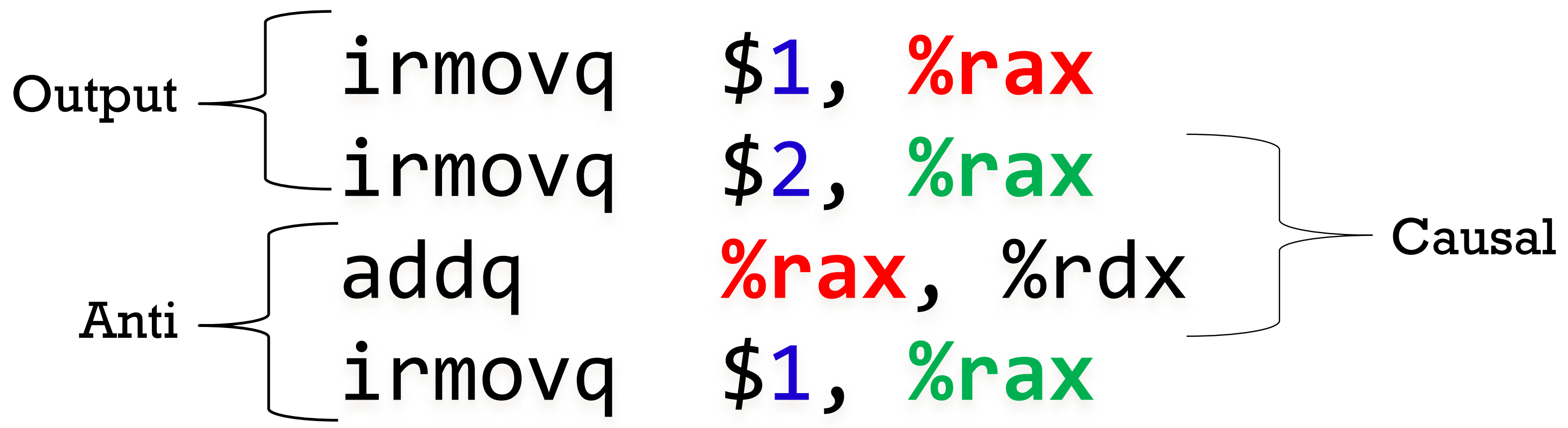
addq %rax, %rbx

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



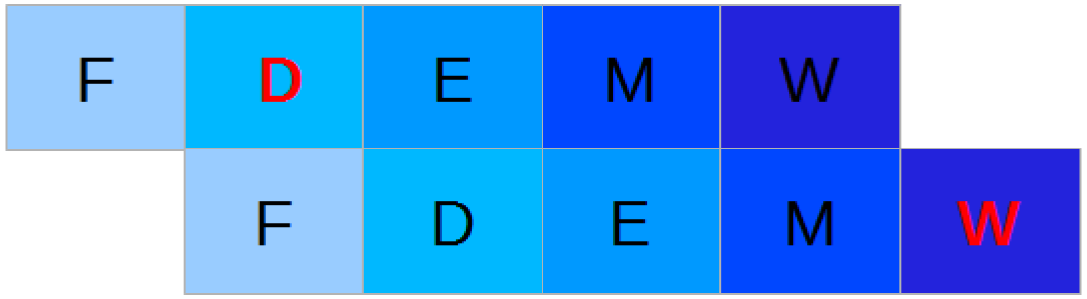
```
irmovq  $1, %rax
irmovq  $2, %rax
addq    %rax, %rdx
irmovq  $1, %rax
```

TYPES OF HAZARDS

For our pipelined Y86 implementation:

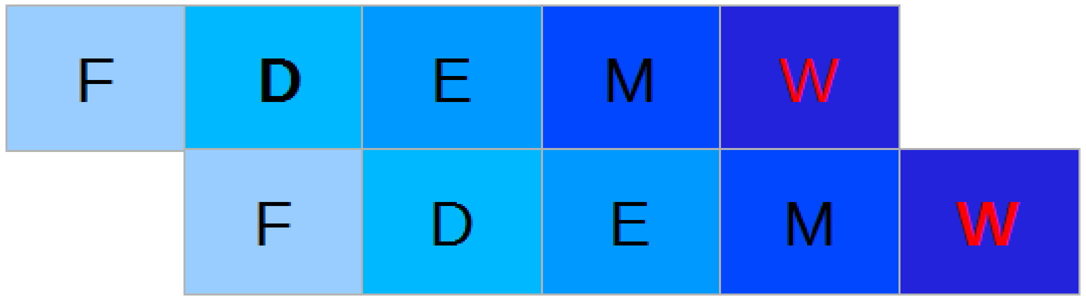
Anti-dependency:

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



Output dependency:

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



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? ______

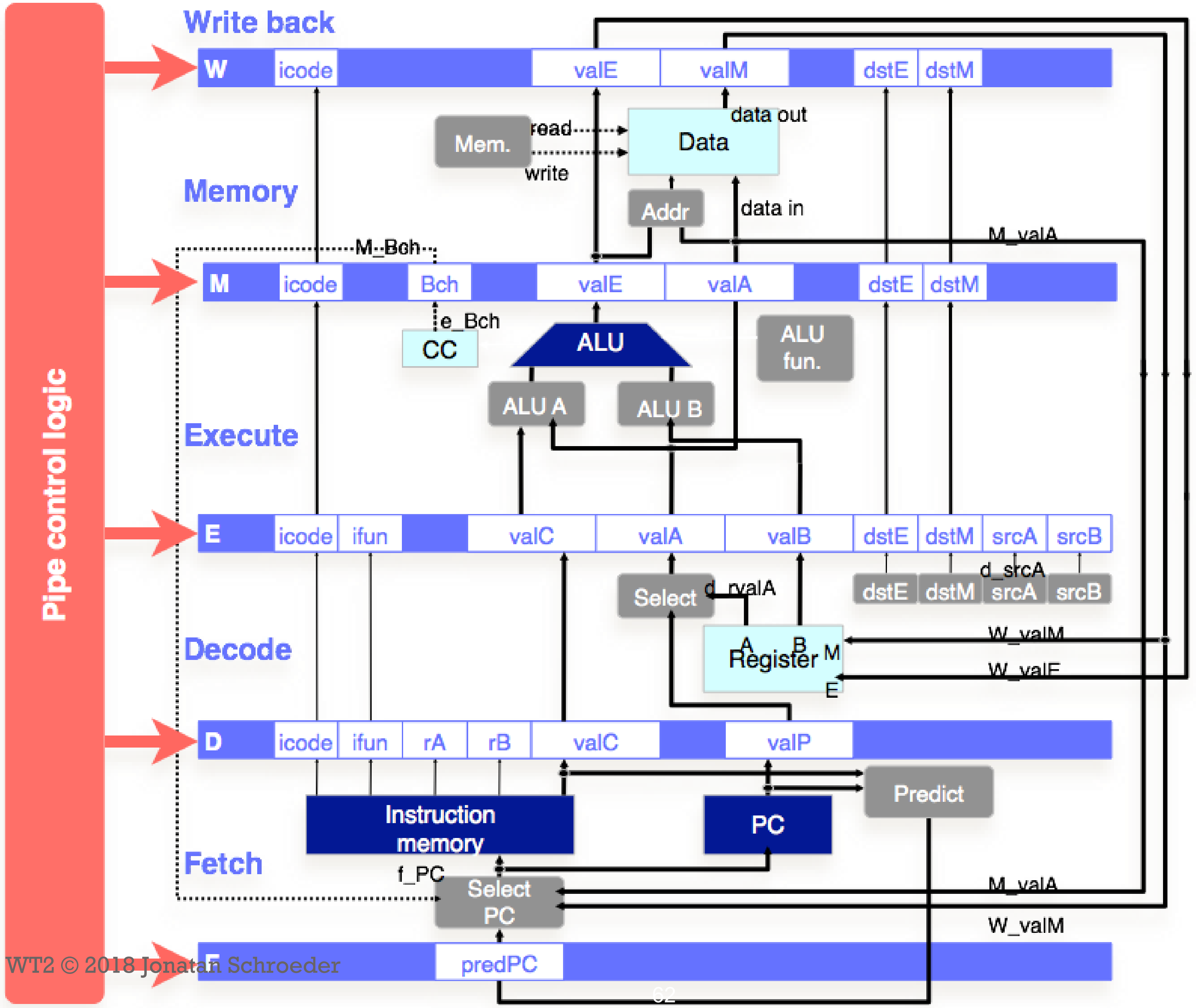
PIPELINE CONTROL UNIT

- 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



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.

DEALING WITH CAUSAL DEPENDENCIES

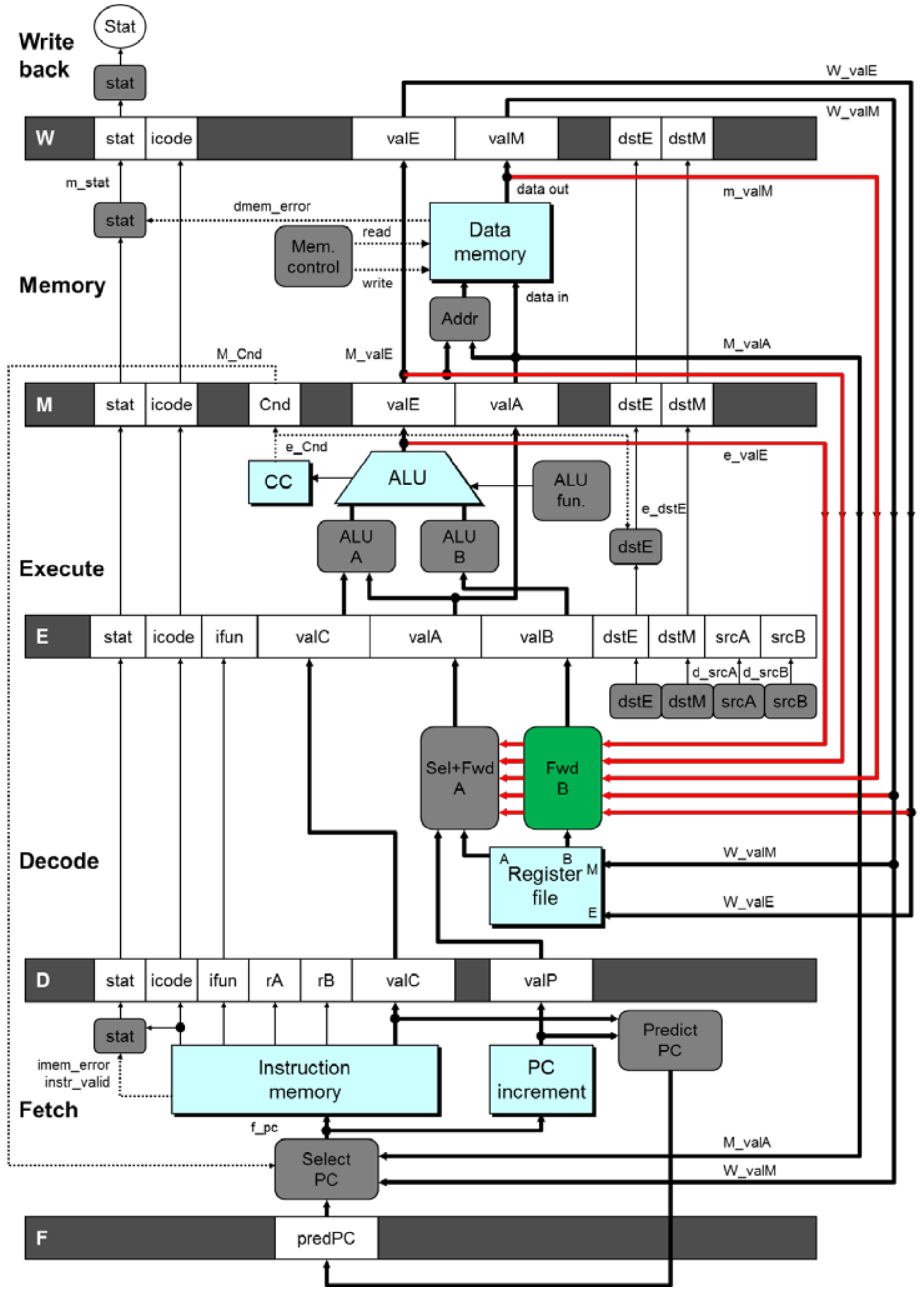
- may have to stall
  - an instruction can't read a value that the processor doesn't know yet
- but in most cases processor has computed value, it just hasn't written it to the register file yet
- so data forwarding will resolve hazard (in these cases)
  - is mechanism that forwards values from later pipeline stages to earlier ones so that an instruction can read value before it is written back to register file

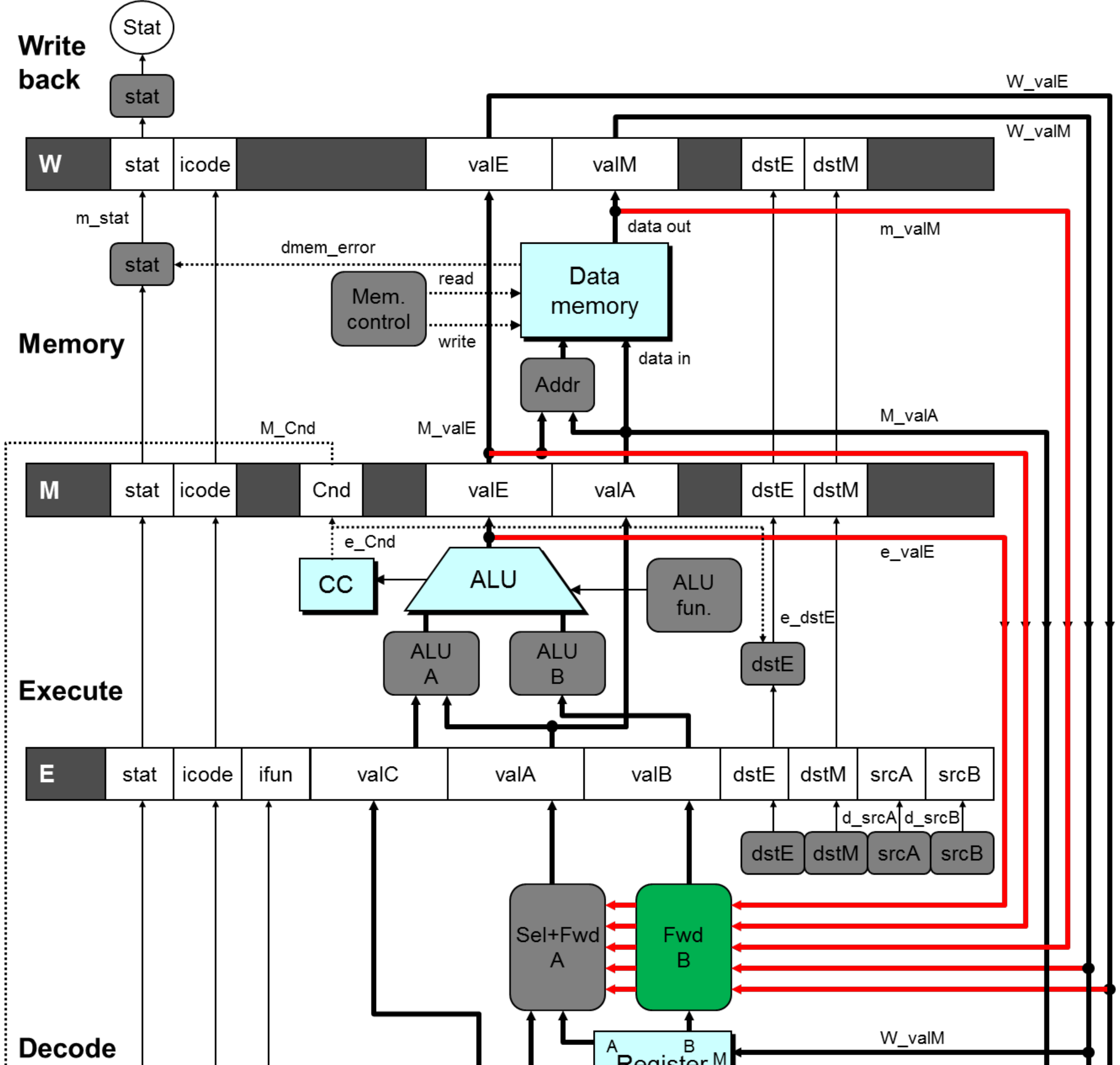
DATA FORWARDING

- In y86, at the end of what stage is output known by CPU?
  - irmovq, OPq, pushq, popq (rsp), call (rsp), ret (rsp)
  - mrmovq, popl (target reg)
- in y86, at the end of what stage is register value needed?

PIPE (WITH FORWARDING)

- **To** Decode
- **From**
  - W: new value to register file
  - M: new value read from memory
  - E: new value from ALU
- **By** sending value and name from all three stages back to forwarding logic in decode stage





Write back
Stat
stat
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
stat
dmem_error
read
write
Mem. control
Data memory
data out
m_valM
data in
Addr
Memory
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_valE
e_dstE
dstE
ALU A
ALU B
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
A
B
Register
M
W_valM
Decode

REGISTER TO REGISTER HAZARD WITH DATA FORWARDING

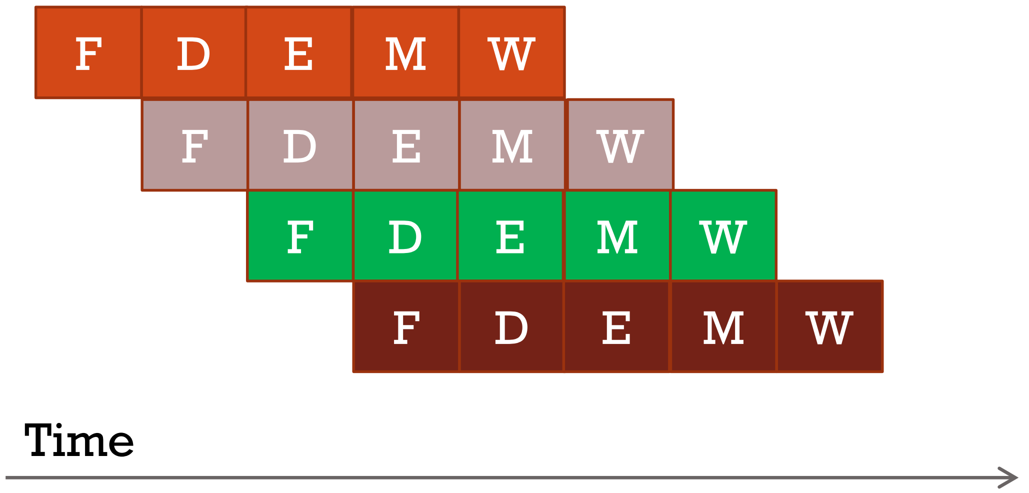
- Instructions:
  - irmovq, addq, subq, andq, xorq
  - pushq, popq, call, ret - all depend upon %rsp
- Forward from ______ to ________

irmovq $10, %rax

addq %rax, %rbx

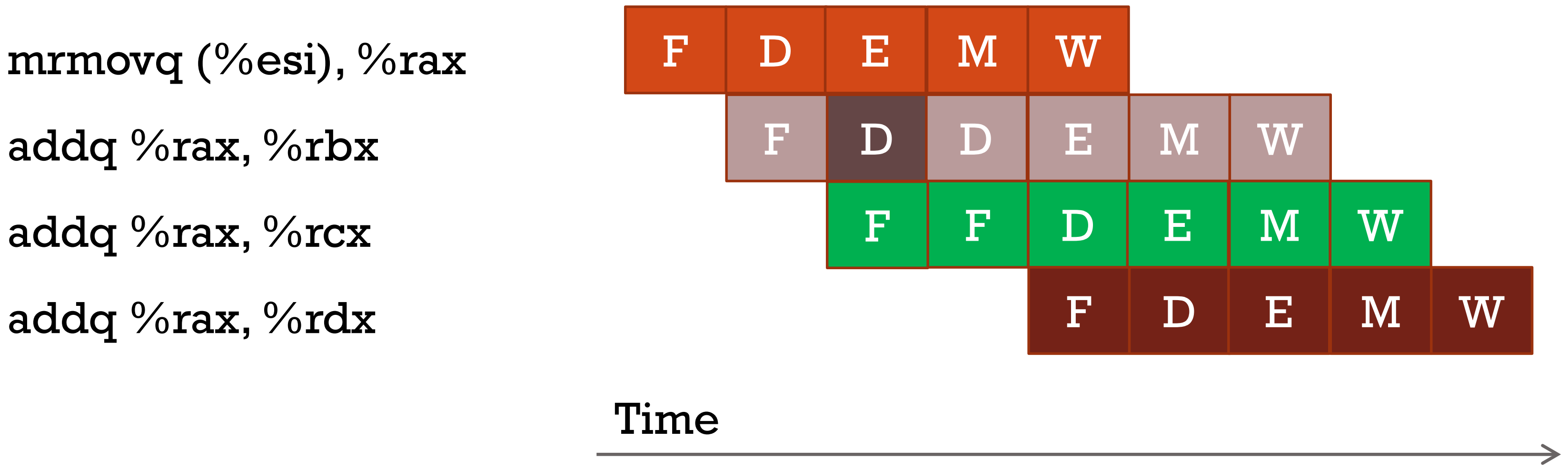
addq %rax, %rcx

addq %rax, %rdx

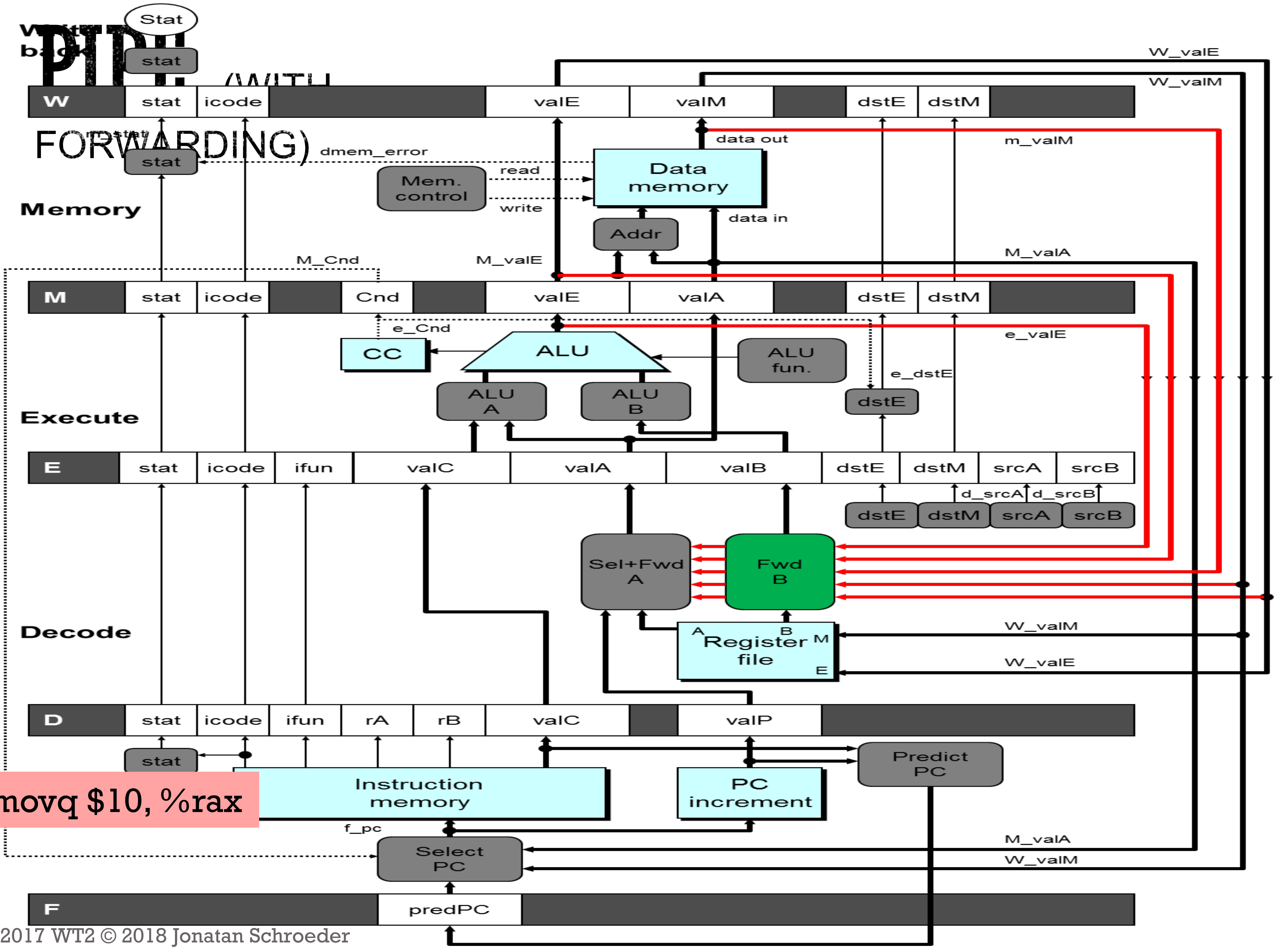


LOAD USE HAZARD WITH DATA FORWARDING

- Instructions: mrmovq, popq
- Stall 1 cycle
- Forward from _____ to _______

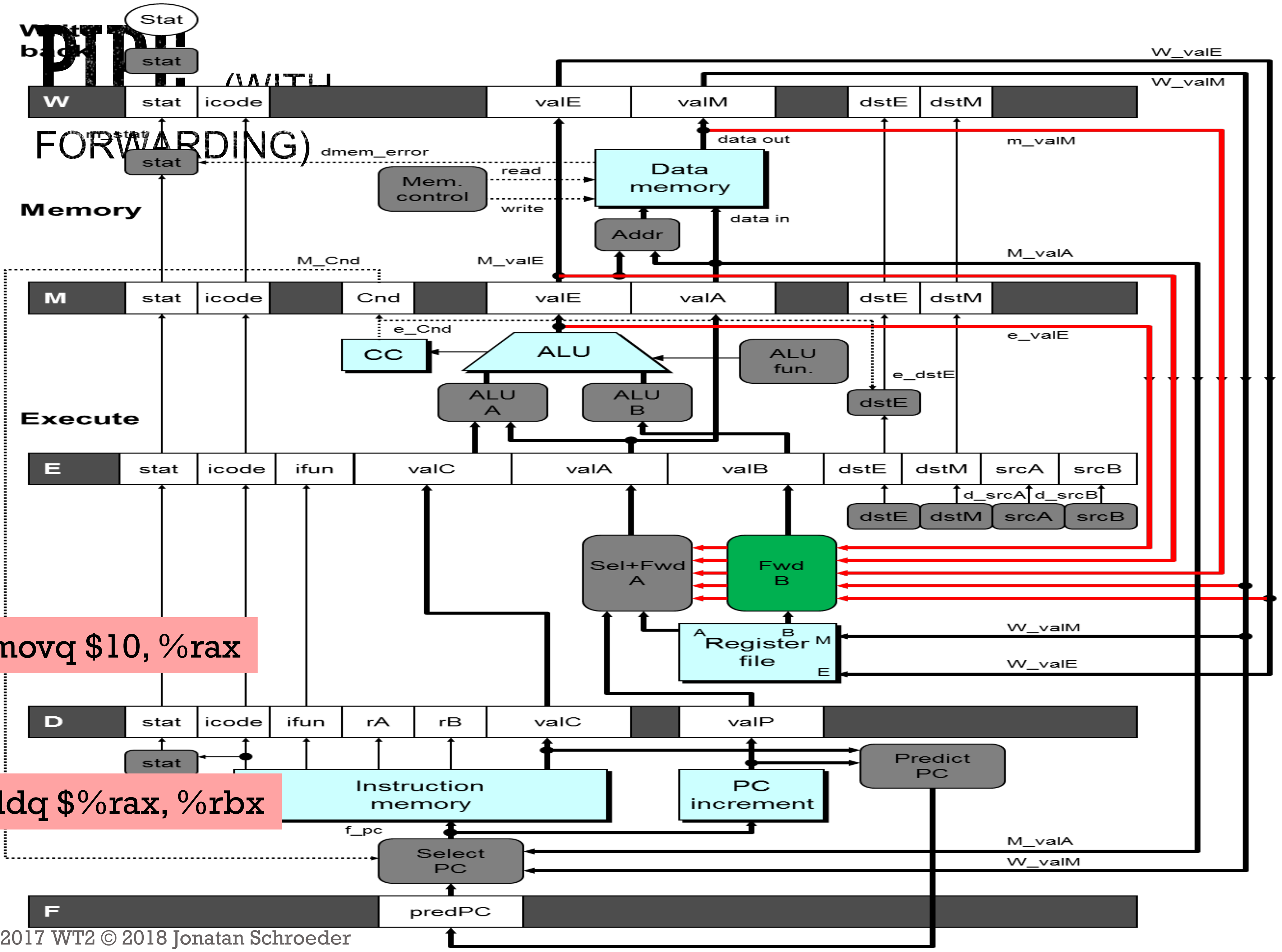


PIPE (WITH FORWARDING)



irmovq $10, %rax

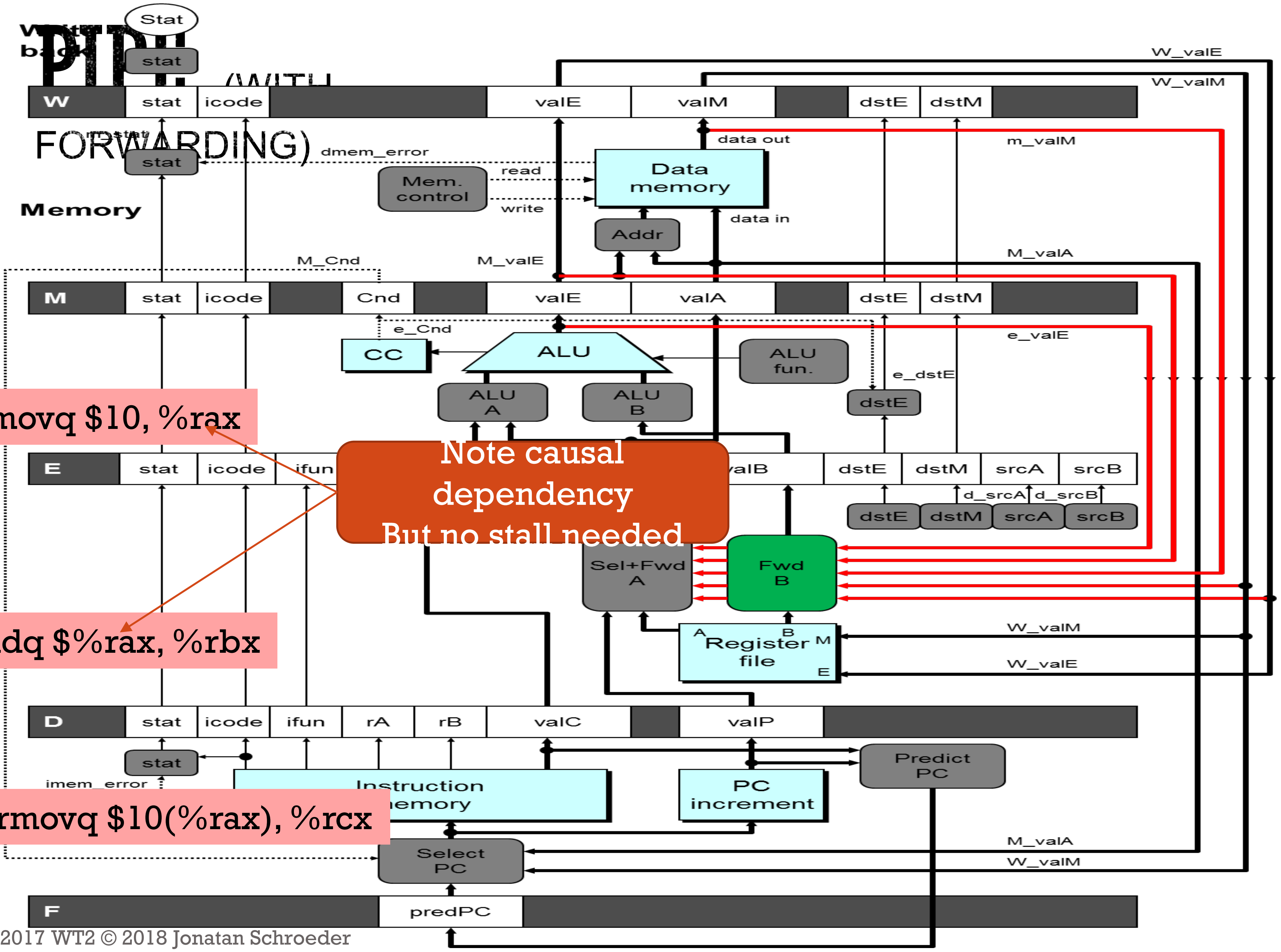
PIPE (WITH FORWARDING)



irmovq $10, %rax

addq $%rax, %rbx

PIPE (WITH FORWARDING)



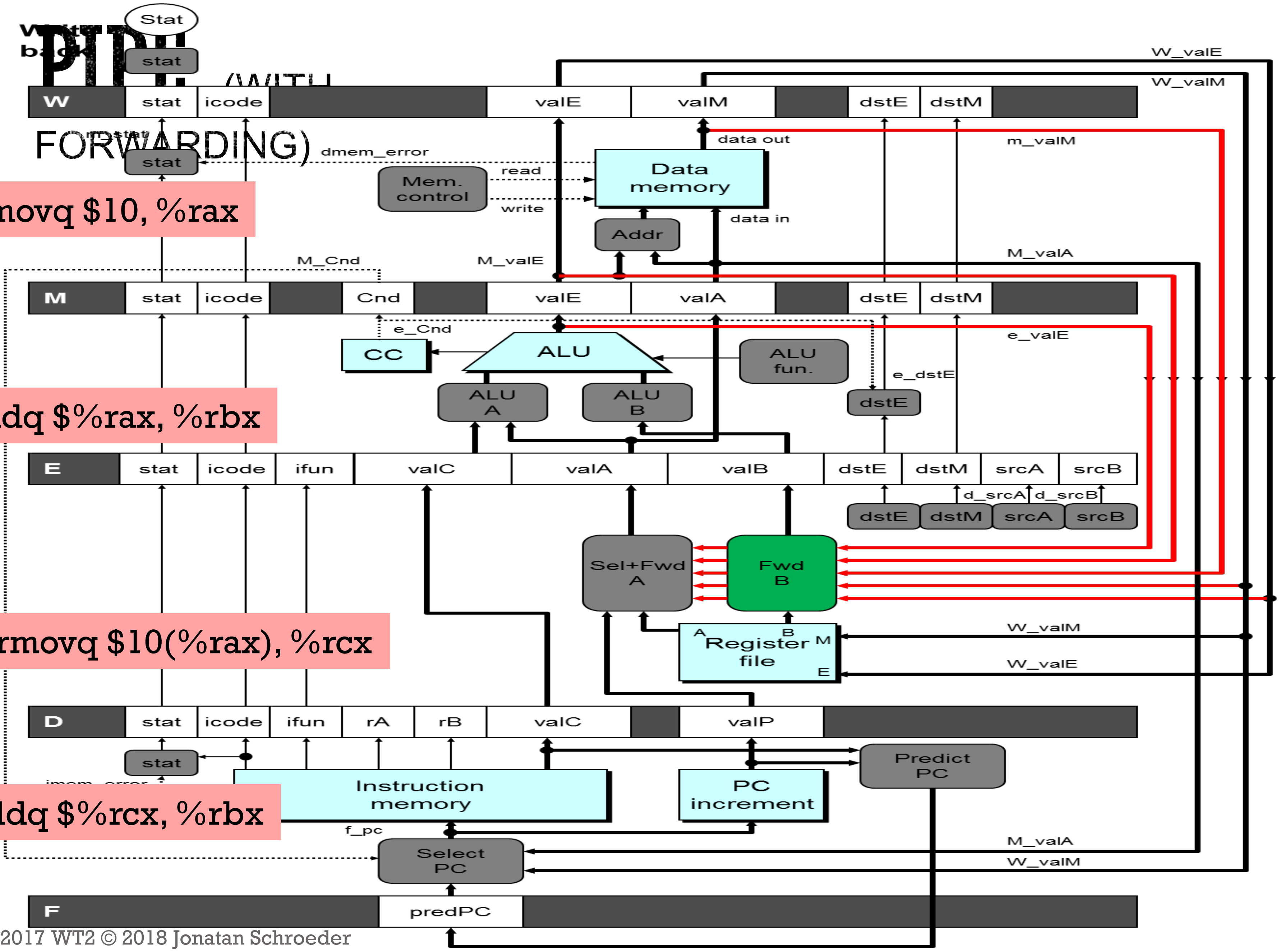
irmovq $10, %rax

Note causal dependency
But no stall needed

addq $%rax, %rbx

mrmovq $10(%rax), %rcx

PIPE (WITH FORWARDING)



irmovq $10, %rax

addq $%rax, %rbx

mrmovq $10(%rax), %rcx

addq $%rcx, %rbx

irmovq $10, %rax

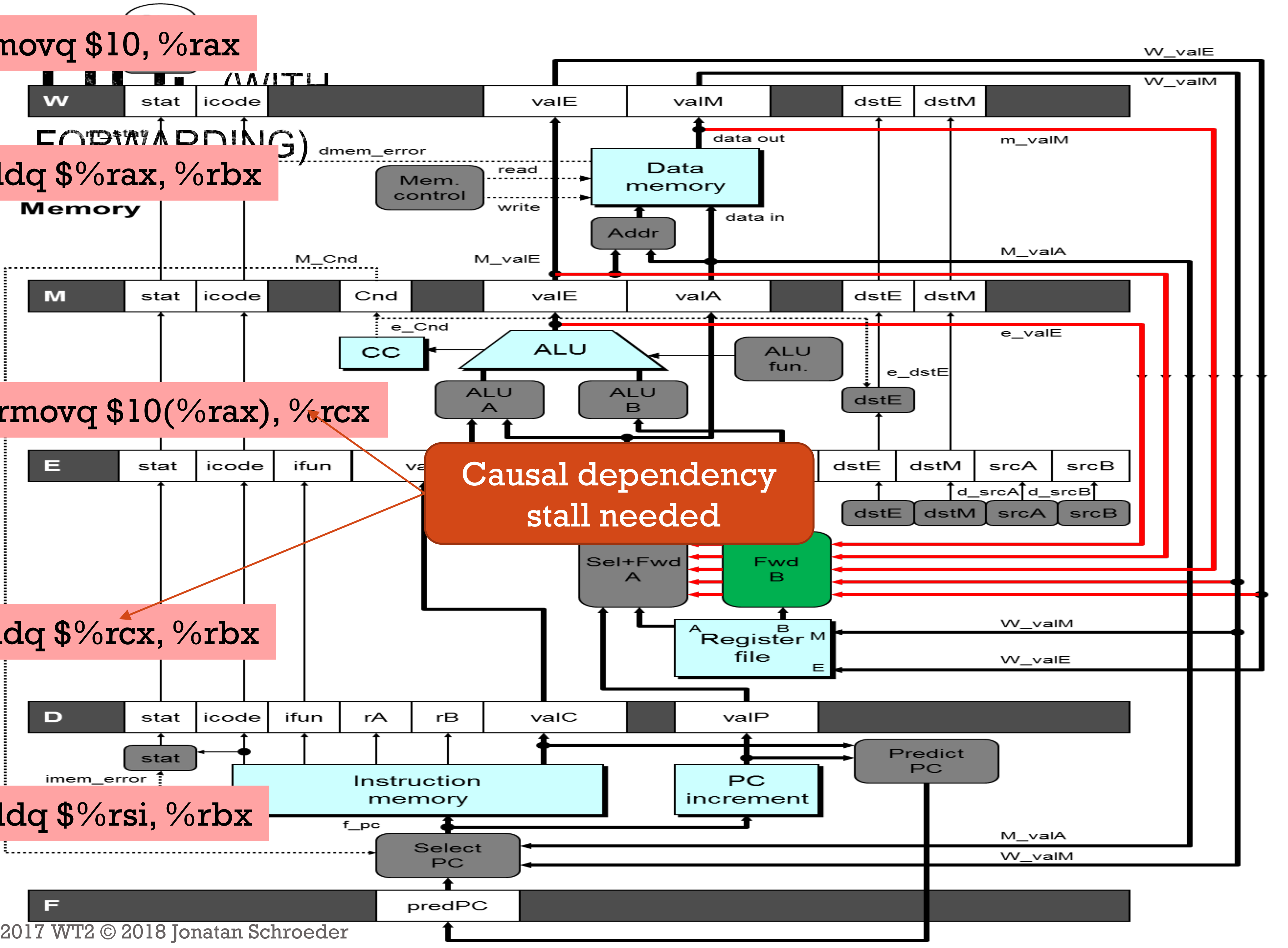
addq $%rax, %rbx

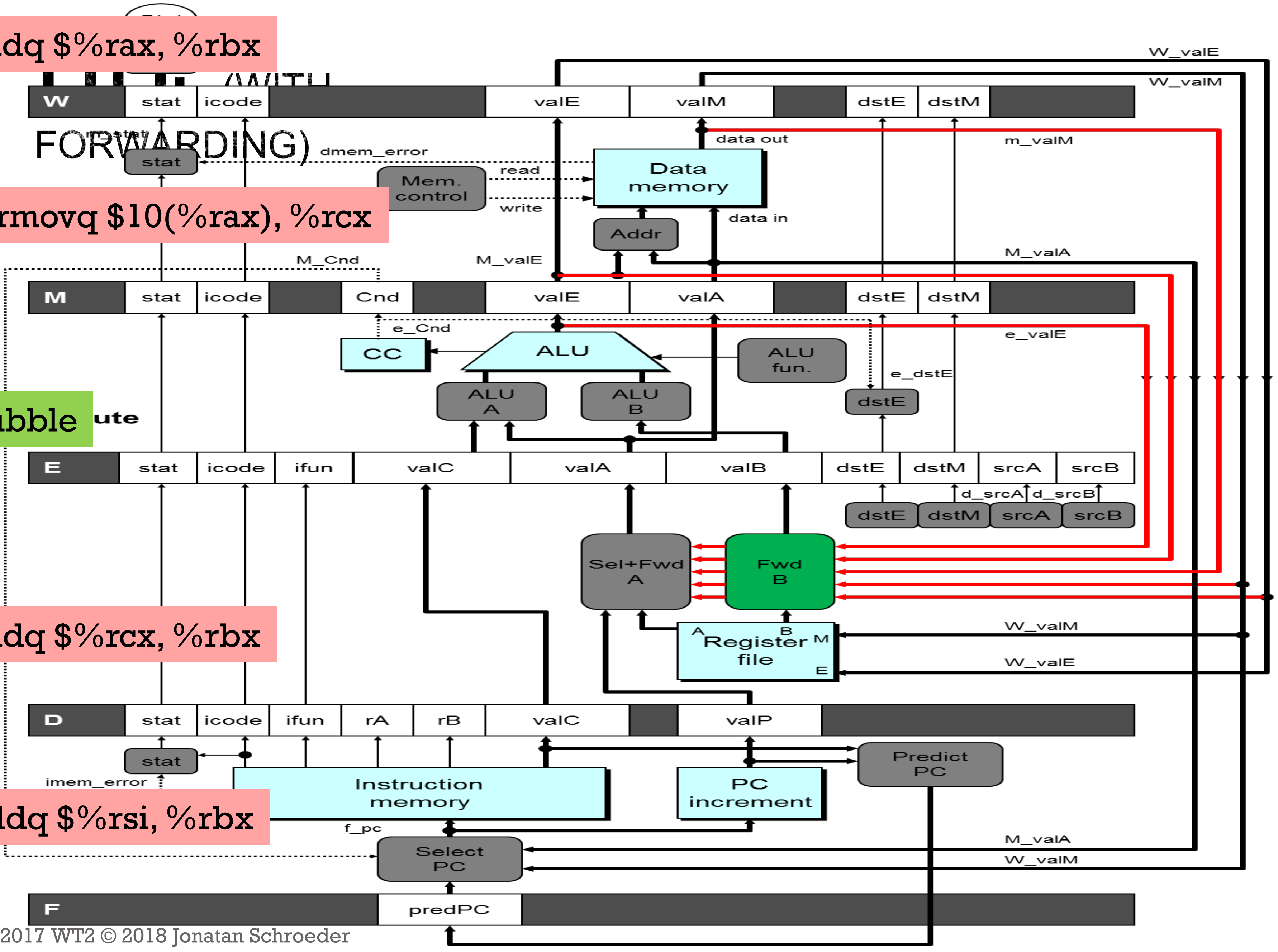
mrmovq $10(%rax), %rcx

Causal dependency stall needed

addq $%rcx, %rbx

addq $%rsi, %rbx





addq $%rax, %rbx
mrmovq $10(%rax), %rcx
Bubble
addq $%rcx, %rbx
addq $%rsi, %rbx
(WITH
FORWARDING)
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
stat
dmem_error
Mem. control
read
write
Data memory
data out
data in
m_valM
Addr
M_Cnd
M_valE
M_valA
M
Cnd
valA
e_Cnd
CC
ALU
ALU fun.
e_valE
e_dstE
dstE
ALU A
ALU B
ute
E
ifun
valC
valB
srcA
srcB
d_srcA
d_srcB
Sel+Fwd A
Fwd B
Register file
A
B
M
E
D
rA
rB
valP
imem_error
Instruction memory
PC increment
Predict PC
f_pc
Select PC
F
predPC

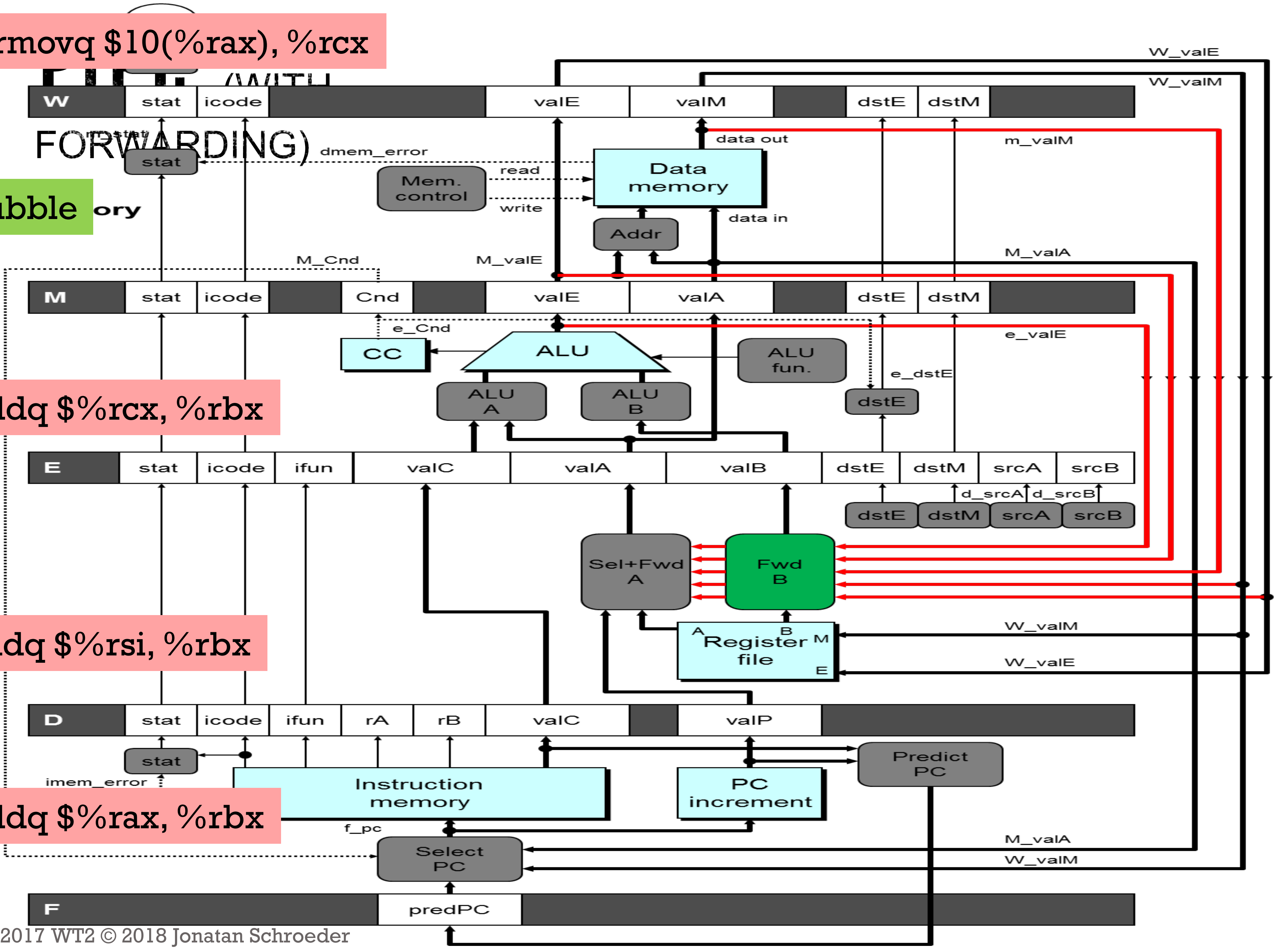
mrmovq $10(%rax), %rcx

Bubble

addq $%rcx, %rbx

addq $%rsi, %rbx

addq $%rax, %rbx



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.

CONTROL DEPENDENCIES

- Unconditional jumps and procedure calls
  - Hazard? (e.g. jmp foo, call bar)
- Conditional jumps
  - Hazard? (e.g. jle foo)
  - What is the problem?
  - At what stage does the CPU know the answer?
- Return
  - Hazard? (e.g. ret)
  - What is the problem?
  - At what stage does the CPU know the answer?

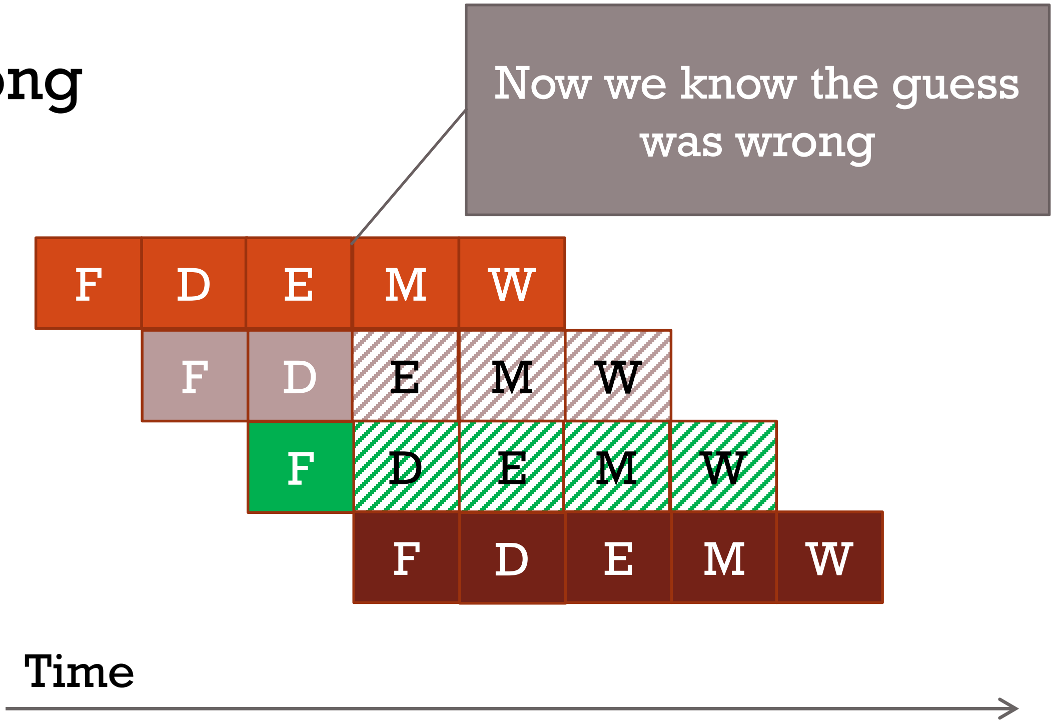
BRANCH PREDICTION

- Problem:
  - We won’t know whether or not to jump until end of Execute stage
  - In the fetch stage (i.e. when setting the next PC) we don’t know if we should use valP (branch not taken) or valC (branch taken)
- Idea:
  - Guess whether or not branch taken
  - Start speculative execution of instructions
  - None will reach M or W stage before we know if the guess was wrong
- Guess wrong:
  - Shoot down all speculative instructions by turning into bubbles

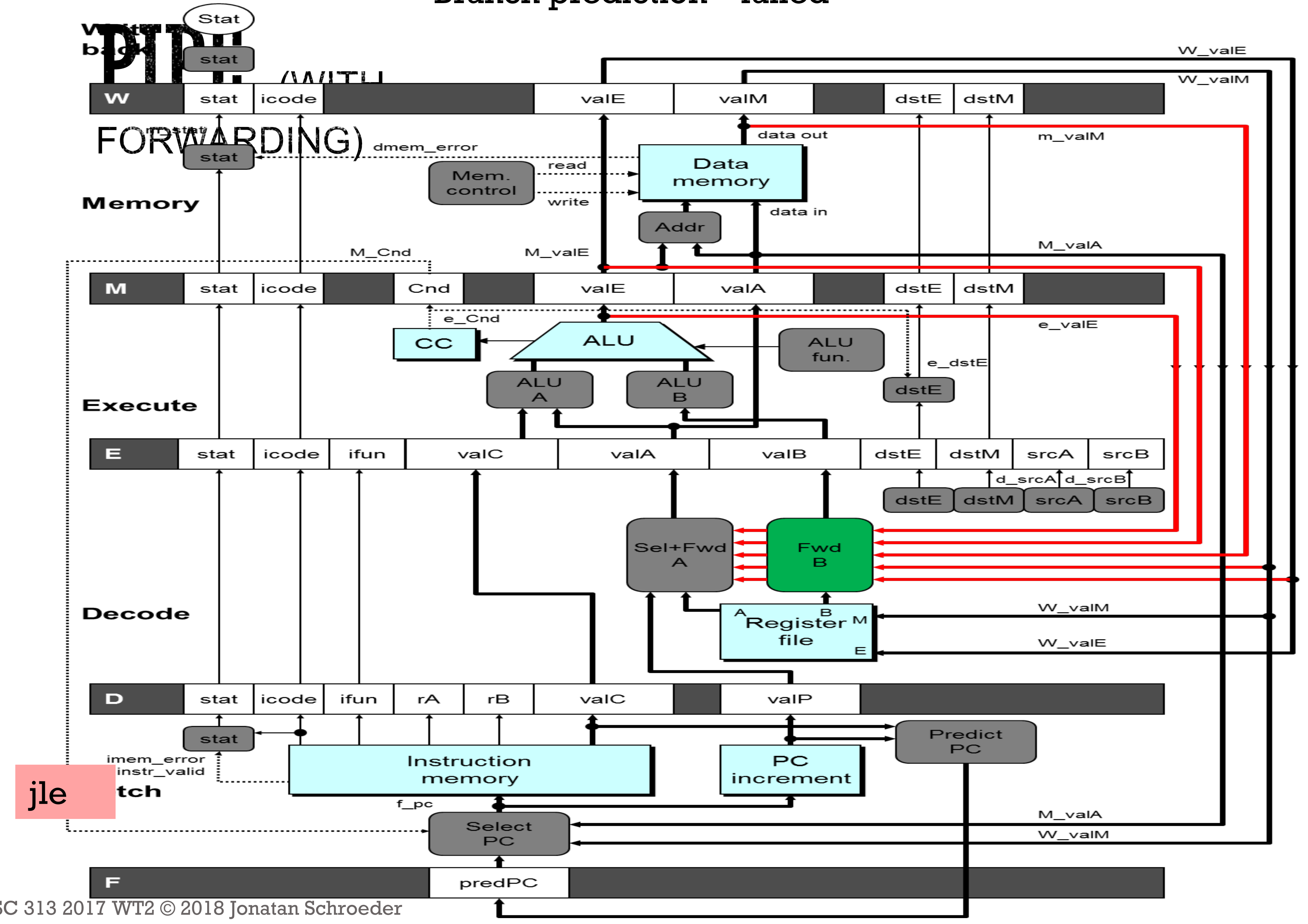
BRANCH PREDICTION

- For conditional jumps:
  - No bubble if guess right
  - Two bubbles if guess wrong

```
0x100          jle $0xbadbeef
0xbadbeef      addq %rax, %rbx
0xbadbeef+2    addq %rax, %rcx
0x109          irmovq $0, %rax
```

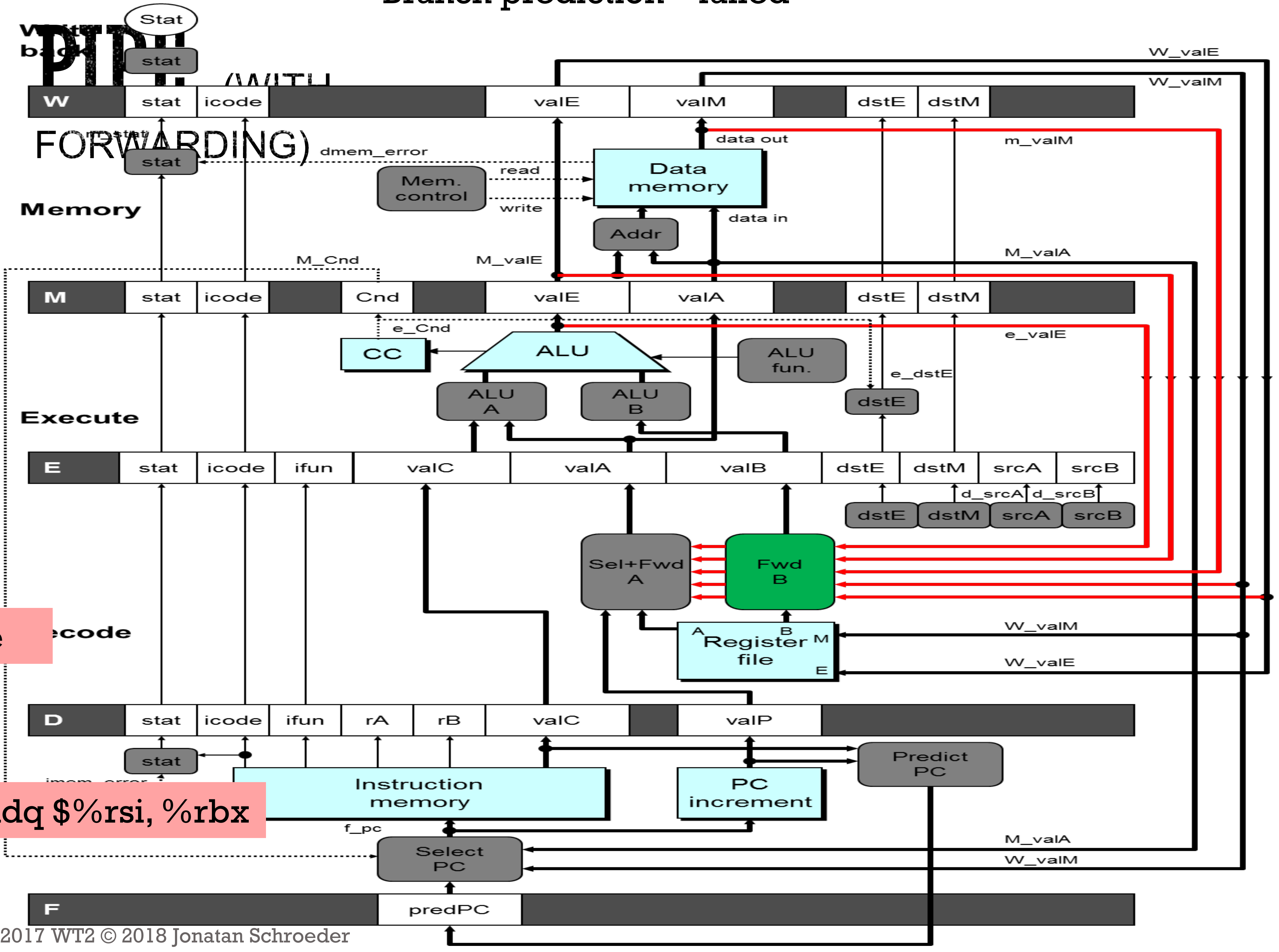


Branch prediction – failed



Branch prediction – failed

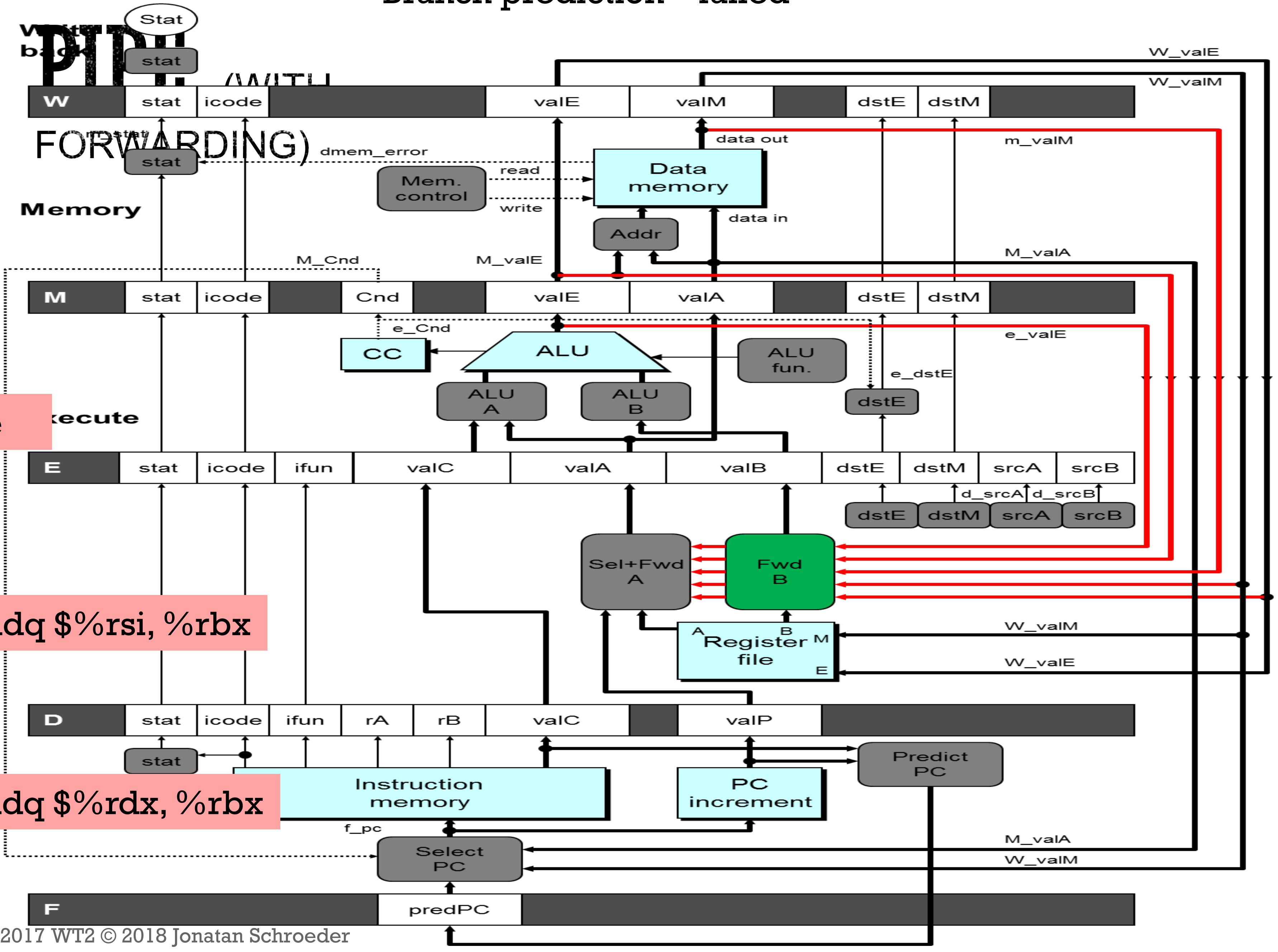
PIPE (WITH FORWARDING)



jle

addq $%rsi, %rbx

Branch prediction – failed

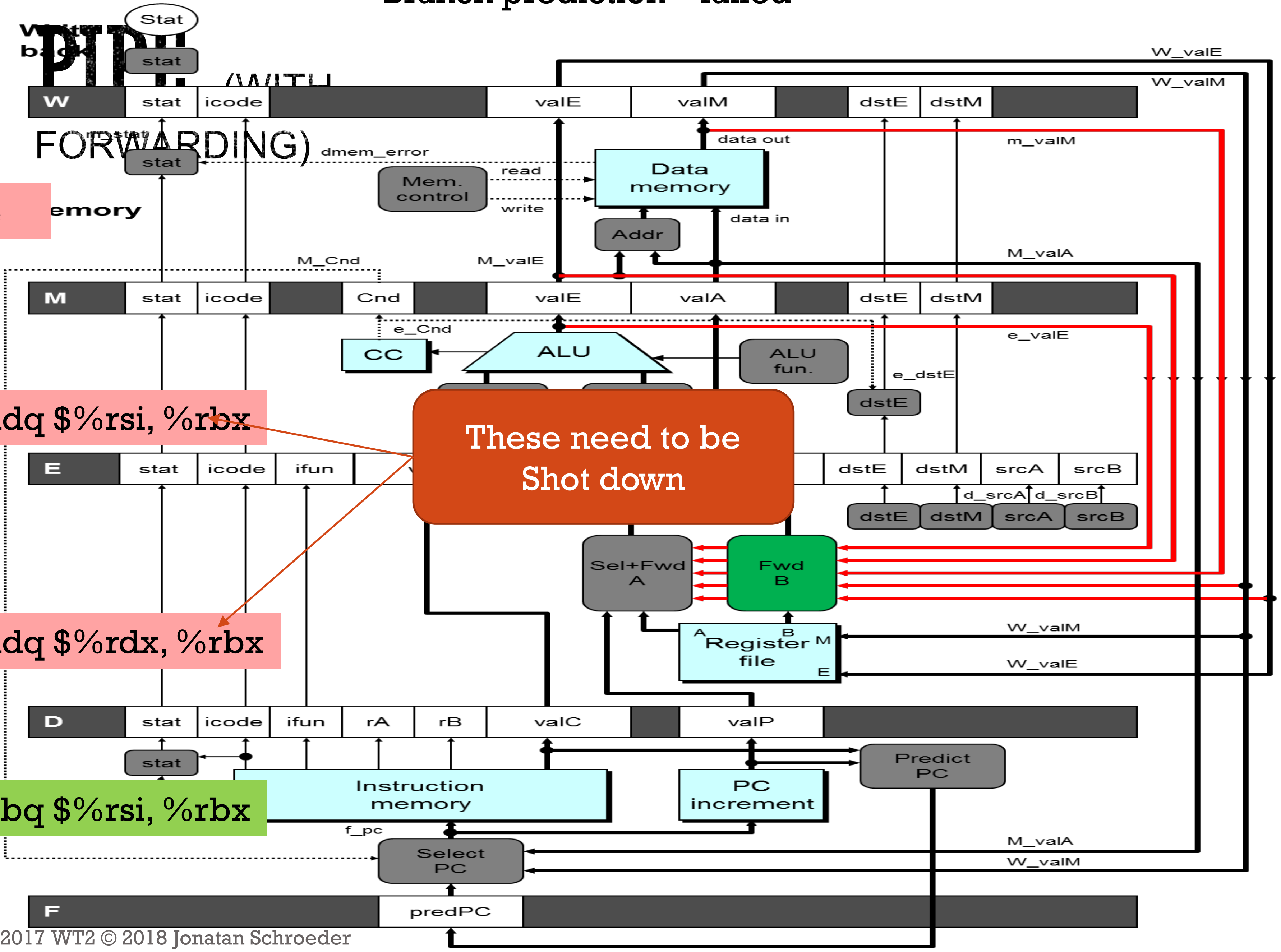


jle

addq $%rsi, %rbx

addq $%rdx, %rbx

Branch prediction – failed



jle

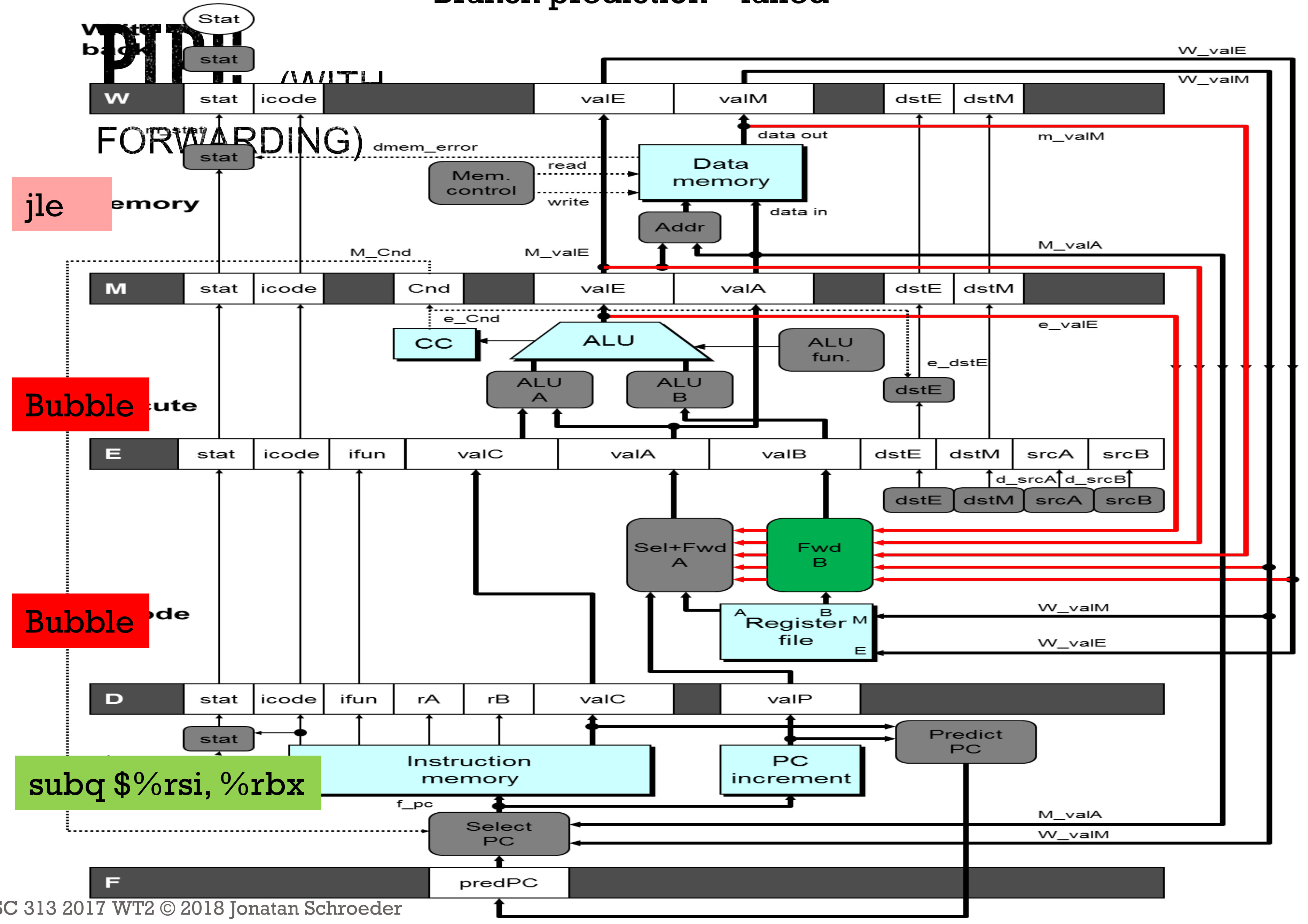
addq $%rsi, %rbx

These need to be Shot down

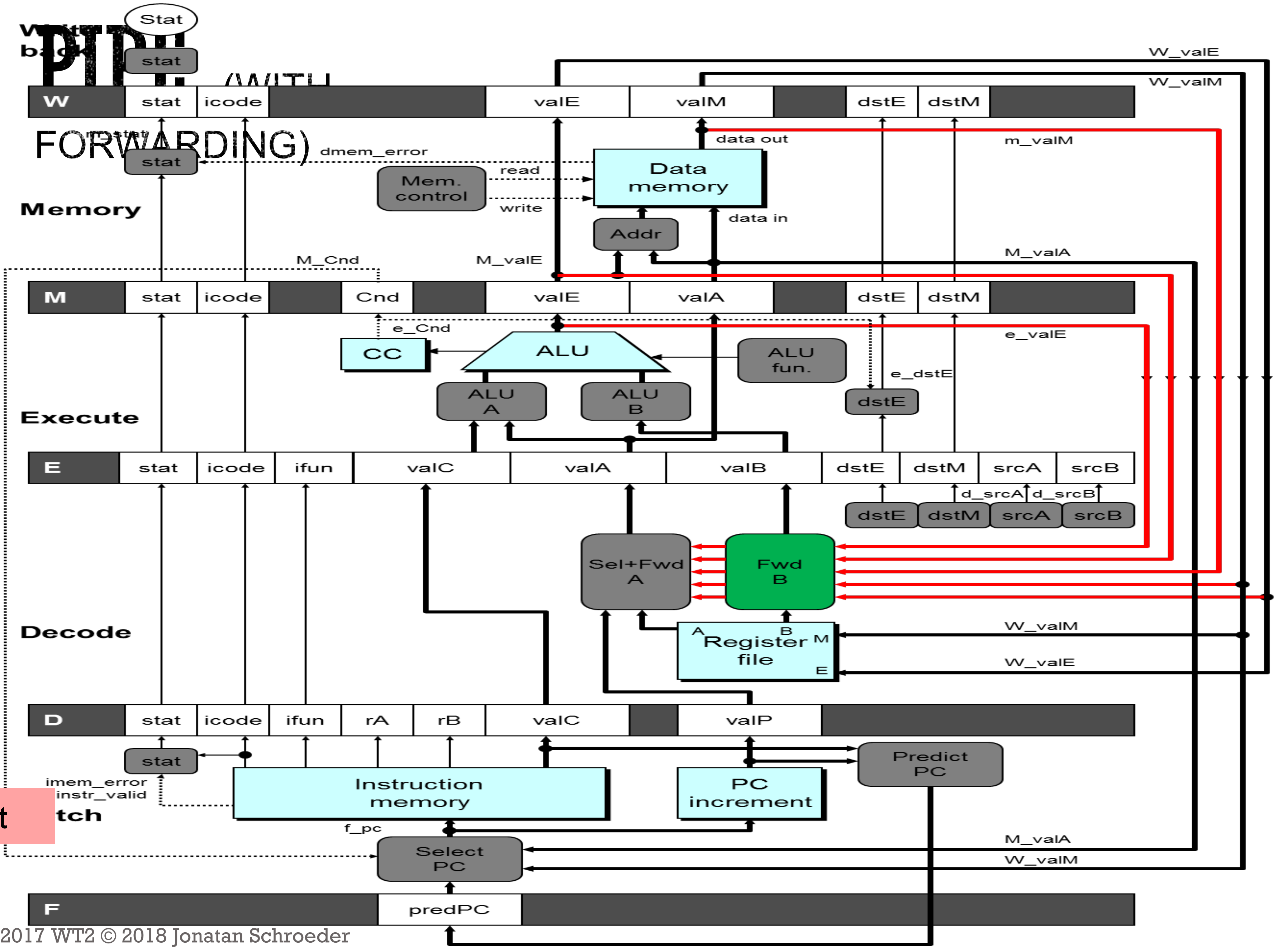
addq $%rdx, %rbx

subq $%rsi, %rbx

Branch prediction – failed

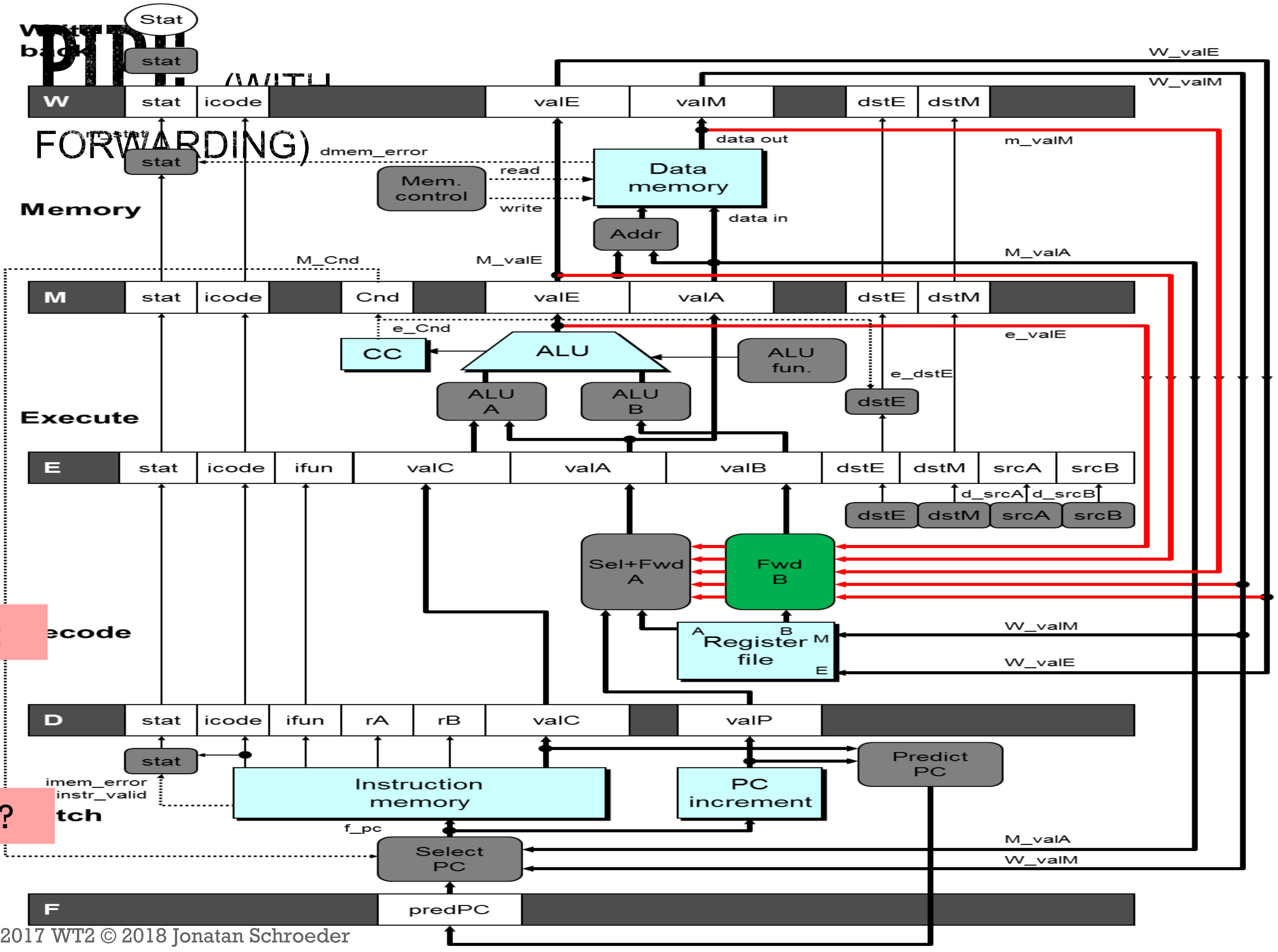


Return


PIPE (WITH FORWARDING)
Stat
stat
W
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
stat
dmem_error
Mem. control
read
write
Data memory
data out
data in
m_valM
Addr
Memory
M_Cnd
M_valE
M_valA
M
Cnd
valA
e_Cnd
CC
ALU
ALU fun.
e_valE
e_dstE
dstE
ALU A
ALU B
Execute
E
ifun
valC
valB
srcA
srcB
d_srcA
d_srcB
Sel+Fwd A
Fwd B
Register file
A
B
M
E
Decode
D
rA
rB
valP
Predict PC
Instruction memory
PC increment
imem_error
instr_valid
f_pc
Select PC
F
predPC
Fetch

ret

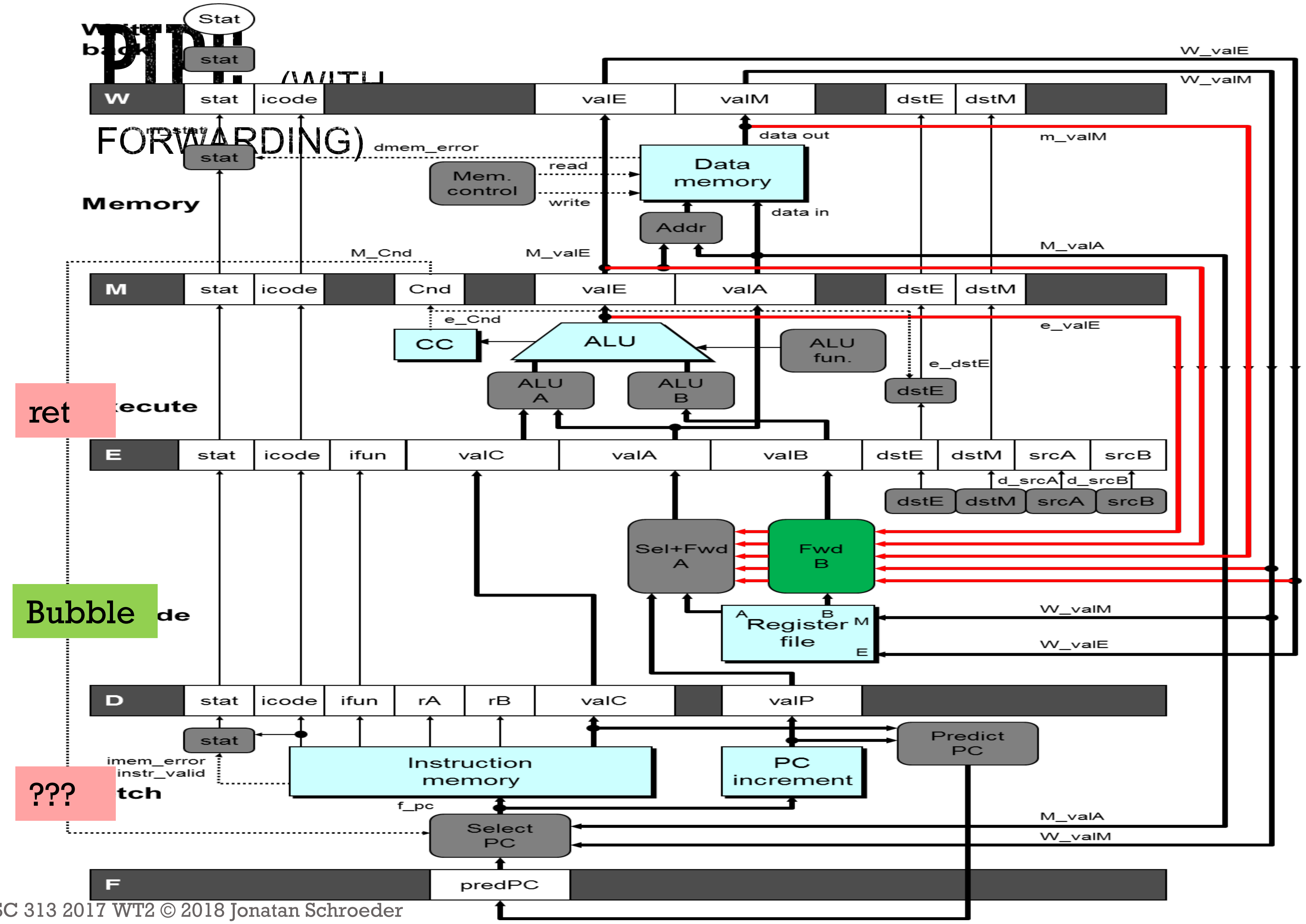
Return



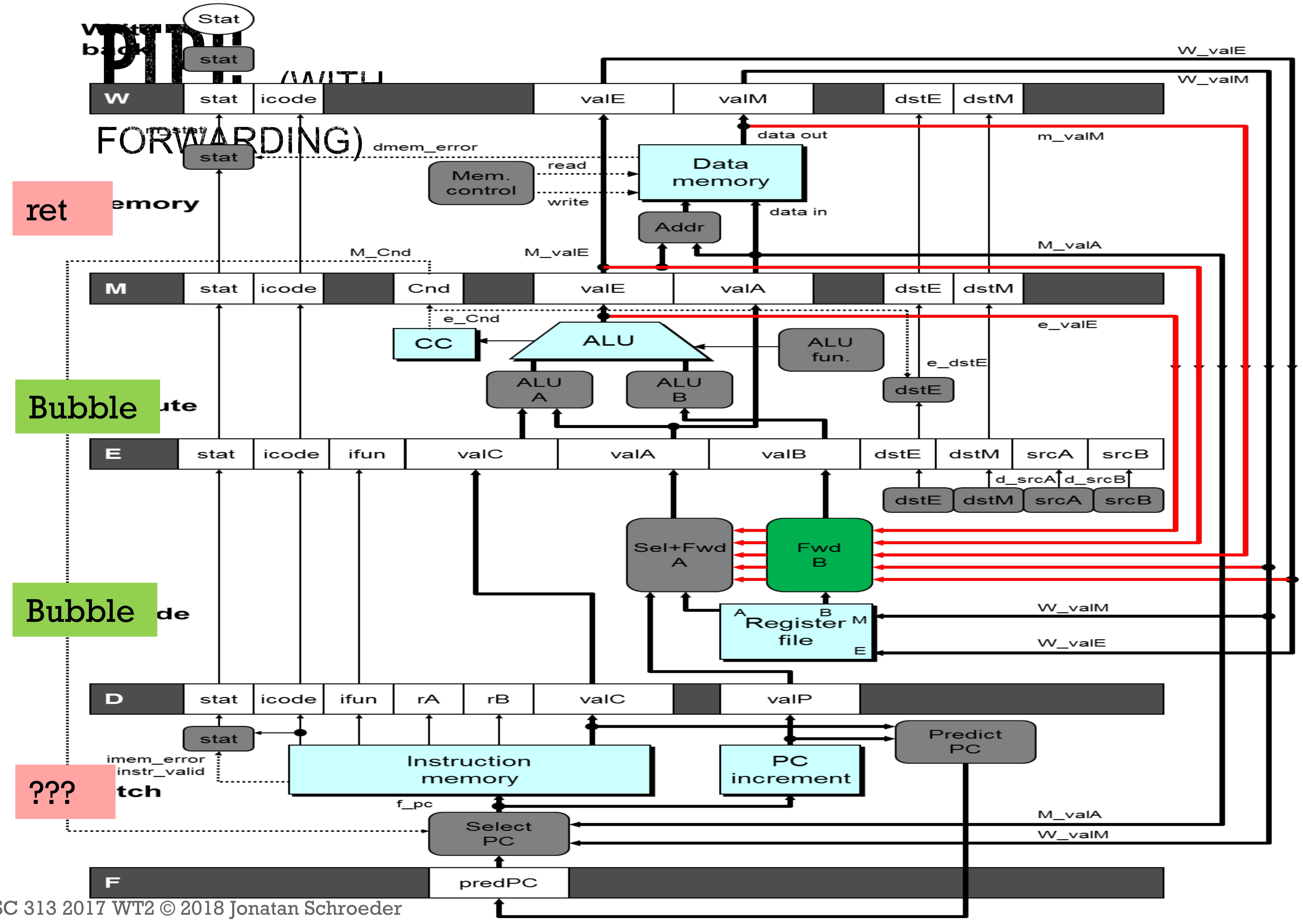
PIPE (WITH FORWARDING)
Stat
stat
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
dmem_error
data out
m_valM
Data memory
Mem. control
read
write
Memory
Addr
data in
M_valA
M_Cnd
M_valE
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
e_valE
CC
ALU
ALU fun.
e_dstE
dstE
ALU A
ALU B
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
ret
ecode
A
B
M
Register file
E
W_valM
W_valE
D
stat
icode
ifun
rA
rB
valC
valP
stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
???
tch
f_pc
Select PC
M_valA
W_valM
F
predPC

Return

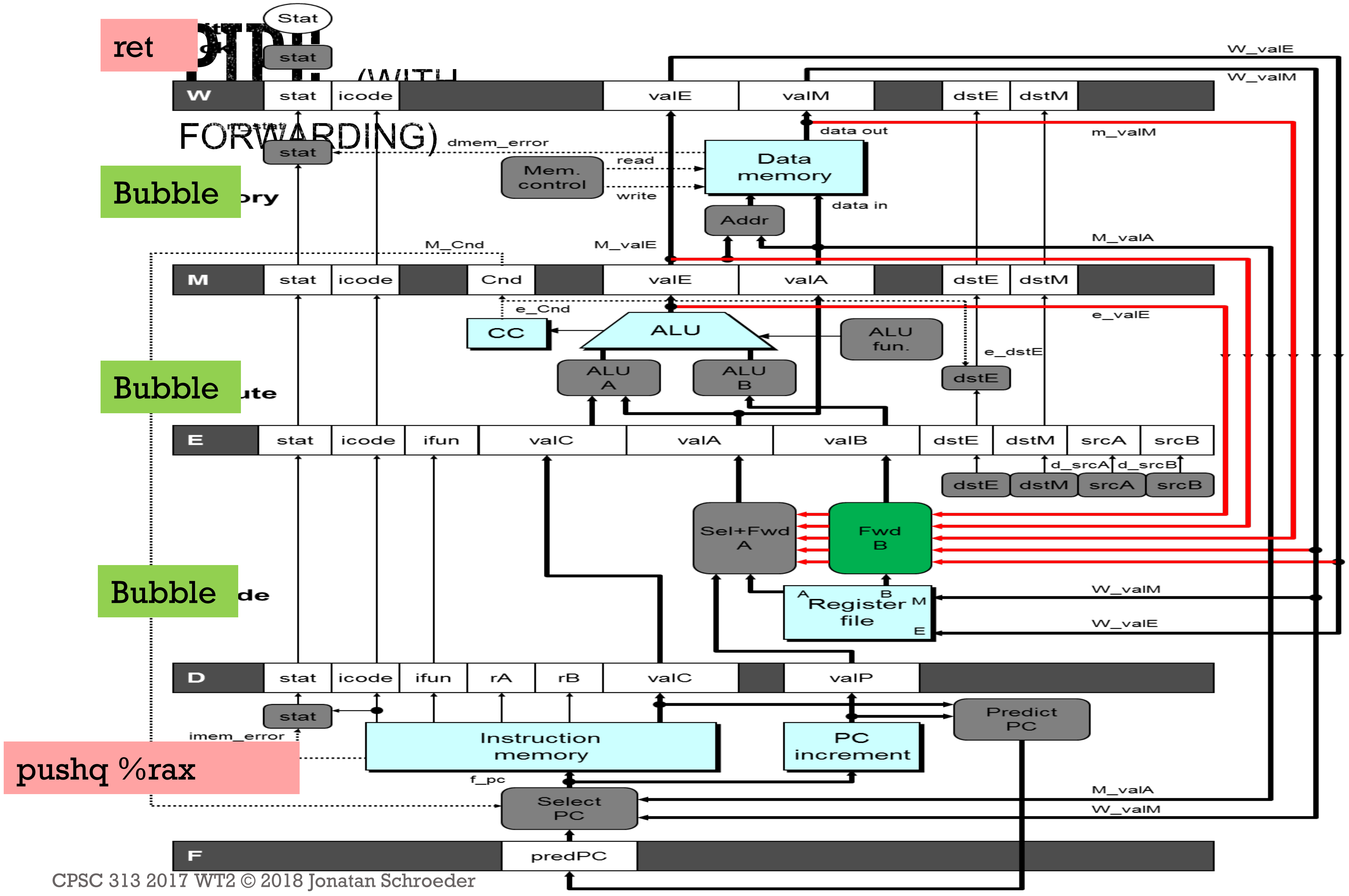
PIPE (WITH FORWARDING)



Return


PIPE (WITH FORWARDING)
Write back
Stat
stat
W
stat
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_stat
stat
dmem_error
data out
m_valM
Data memory
Mem. control
read
write
data in
Addr
ret
Memory
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_valE
e_dstE
dstE
ALU A
ALU B
Bubble
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
A
B
Register file
M
E
W_valM
W_valE
Bubble
Decode
D
stat
icode
ifun
rA
rB
valC
valP
stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
???
Fetch
f_pc
Select PC
M_valA
W_valM
F
predPC

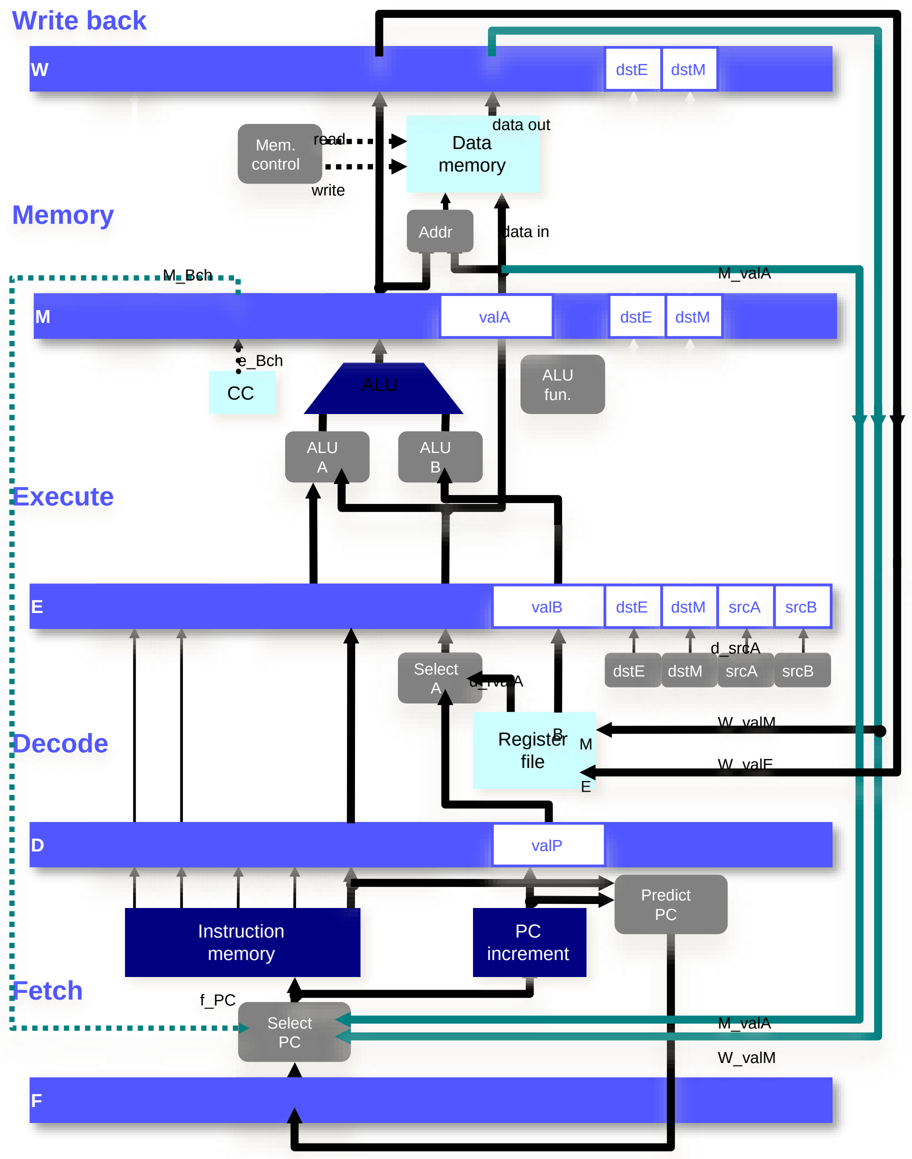
Return



ret
Bubble
Bubble
Bubble
pushq %rax
Stat
stat
W
icode
valE
valM
dstE
dstM
W_valE
W_valM
m_valM
data out
dmem_error
read
write
Mem. control
Data memory
data in
Addr
M_Cnd
M_valE
M_valA
M
Cnd
valA
e_Cnd
CC
ALU
ALU fun.
e_valE
e_dstE
ALU A
ALU B
E
ifun
valC
valB
srcA
srcB
d_srcA
d_srcB
Sel+Fwd A
Fwd B
Register file
A
B
M
E
D
rA
rB
valP
Predict PC
imem_error
Instruction memory
PC increment
f_pc
Select PC
F
predPC
FORWARDING)

PREDICTING THE NEXT PC IN FETCH STAGE

- unconditional jumps (call, jmp)
  - predPC = valC, available in F
- conditional jumps
  - know if branch taken in E (bch)
  - jump prediction in F
    - e.g., predPC = valC (i.e., taken)
  - misprediction control hazard
    - handled by "Select PC" in F
    - feedback from M
  - no stall if prediction is correct
- return from procedure call
  - know target in M
    - feedback from W to "Select PC" in F
  - y86 stalls 3 cycles



IS IT JUMP PREDICTION OR GUESSING?

- Are jumps usually 50% likely to be taken, and 50% not?
  - If so might as well just guess at random.
- Consider how jumps are used in a program:
  - loops:
    - continue condition at bottom of loop is normally taken.
    - exit condition at top of loop is normally not taken.
  - ifs:
    - if testing for error condition, error handling code normally skipped.
    - if testing for recursion base case, base case normally skipped.

OBSERVATIONS

- Most jumps tend to go one way more often
  - Taken or not taken for that line of code predominates
  - Supported by empirical evidence
- If we can figure this out we can improve a program’s performance
  - By guessing the most often direction

WHAT THE COMPILER KNOWS

- In many cases the compiler can make a good prediction because:
  - It creates the code
  - For loops it knows if the jump means exit or continue
  - *Might* be able to spot error test for ifs
  - Only has the program text to make decisions on
  - Cannot (usually) use dynamic information from execution

WHAT COMPILER WANTS FROM ISA

- Conditional jumps to be predicted one way or the other
- If ISA rules are explicit, compiler can adjust code
  - Compiler can change type of jump based on its prediction
  - Example: convert
    `if (a < b) c = 1; else d = 2;`
    to
    `if (a >= b) d = 2; else c = 1;`
- Complicated prediction rules are OK as long as they are well-defined

EXAMPLE OF COMPLEX PREDICTION

- predict *backward* jumps are taken and *forward* jumps are not taken
  - backward jumps (that just go a short distance) are almost always loop-continue jumps
  - so they will be mostly taken
  - forward jumps could be anything, so ISA might predict not taken to add flexibility
- what's most important is that the compiler knows what the processor will do

DYNAMIC JUMP PREDICTION

- Sometimes the compiler can’t tell, but jump still has a strong tendency one way or another
  - e.g., its hard for compilers to tell which if branch tends to be executed
- Dynamic Jump Prediction is done by CPU Hardware
  - this is a type of jump where the compiler does not know what will happen
  - the hardware bases its prediction on the past behaviour it observes of the jump

IMPLEMENTING DYNAMIC JUMP PREDICTION

- CPU hardware maintains an on-chip cache of recent jump results
  - Caches jump address to bch
- When jump is in Execute stage (bch is computed) it updates this cache
- When jump is in Fetch state, use history as basis for prediction
  - Example: if last jump was taken, assume it's taken again

IMPLEMENTING DYNAMIC JUMP PREDICTION

- Better implementation: remember not just the last execution, but the last few results
  - Example: store branch history as a sequence of bits with 1 indicating taken
  - e.g., 1010 means the jump's recent taken history was: no, yes, no yes
- Use the history as predictor to future jumps
  - Option 1: use the count of the majority of bits as prediction
  - Option 2: try to observe a pattern

DYNAMIC PREDICTION & COMPILER

- Compiler needs to know if jump prediction is used and how the decision is made
  - Compiler can change code to make better use of predictor
- One option
  - One set of branch instructions that use static prediction
  - One set that uses dynamic prediction
- Another option: delay slots
  - Instructions that execute regardless of jump direction
  - Found in older RISC processors (e.g., MIPS)

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.

INDIRECT JUMPS

- Y86-64 only has direct jumps (destination in instruction)
  - Special case: return (discussed later)
- Indirect jumps are common
  - Indirect jump: jmp D(%rax)
    - PC ← D + R[%rax]
  - Double indirect jump: jmp *D(%rax)
    - PC ← M[D + R[%rax]]
  - necessary to support modern programming languages (e.g., polymorphic dispatch in OO languages)

PROBLEM

- Unlike direct jumps, we do not know the target address in Fetch
  - for indirect we know it at the end of Execute
  - for double-indirect we know it at the end of Memory
  - Consequently, we cannot predict it in Fetch, too many targets
- This is really bad
  - virtually every procedure call in object-oriented language is double-indirect

INDIRECT JUMPS: RETURN

- Problem
  - Can't "guess" where return moves to
  - Must read return address from memory
  - but that doesn't happen until M
- Result
  - leads to seemingly inevitable 3 bubbles for each return
  - *returns are common*, because procedures are small
  - a good idea for program readability, but with bad performance consequences

SOLUTION

- Maintain a small stack of return addresses on the CPU Chip
    - Add address on call instruction
    - Retrieve address on return instruction
    - Run retrieved address speculatively
- note that this stack has a finite size, so return address may not be there
    - in this case, we follow the old procedure and stall three times

POLYMORPHIC DISPATCH

- To generate code for the call object.method() (e.g. b.foo())
  - object contains pointer to table with method addresses
  - Index into the table is determined by the method name
  - Assuming %rax is object address, D is method index:
    - mrmovq (%rax), %rbx
      // %rbx is now table address
    - call *D(%rbx)
      // jump to method address stored in table

```
class Base {
   void foo () { ... }
   void bar () { ... }
}

class Sub extends Base {
   void foo () { ... }
}

void zot (Base b) {
   b.foo ();
}
```

This is bonus material. You will not be examined on this. You many ignore it.

PREDICTION FOR INDIRECT JUMPS

- Key observation
  - while compiler does not actually know the class of object, it knows only that the object implements the static type of the variable
  - many variables tend to store objects of the same type over time
- Key idea
  - the hardware (or Java Virtual Machine) remembers object type and target address used the last time call was executed. If next call is to same type, it uses cached target address instead of reading it from memory.
- Implementation
  - maintain a cache of call instructions with address of procedure last called (ID a call instruction by its address)
    - predict that call will call this same procedure again when the same call instruction is executed

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.

PERFORMANCE ANALYSIS

Don’t count NOPs

- cycles per instruction (CPI)
  - measures pipeline efficiency
  - Inversely related to throughput

-
  ```
  CPI = totalCycles / instructionRetiredCycles
      = 1 + lp + mp + rp
  ```
- bubble penalties

load/use	`lp`	= 1 bubble * prob. of occurrence
branch misprediction	`mp`	= 2 bubbles * prob. of occurrence
return	`rp`	= 3 bubbles * prob. of occurrence

PERFORMANCE ANALYSIS EXAMPLE

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Penalty
load/use	lp	25%	20%	1	?
mispredict	mp	20%	40%	2	?
return	rp	2%	100%	3	?
total					?

```
CPI = ?
```

LIMITS TO PIPELINE DEPTH

- Goal is improved throughput
  - retired instructions / second
  - 1 / (CPI * clock-period)
- Are deeper pipelines better?
  - Pros
  - Cons