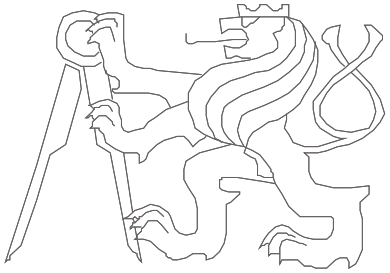
QtMips – Simulator for Education

CPU Core, Pipeline and Cache Visualization [*1] for Computer Architecture Courses [*2]



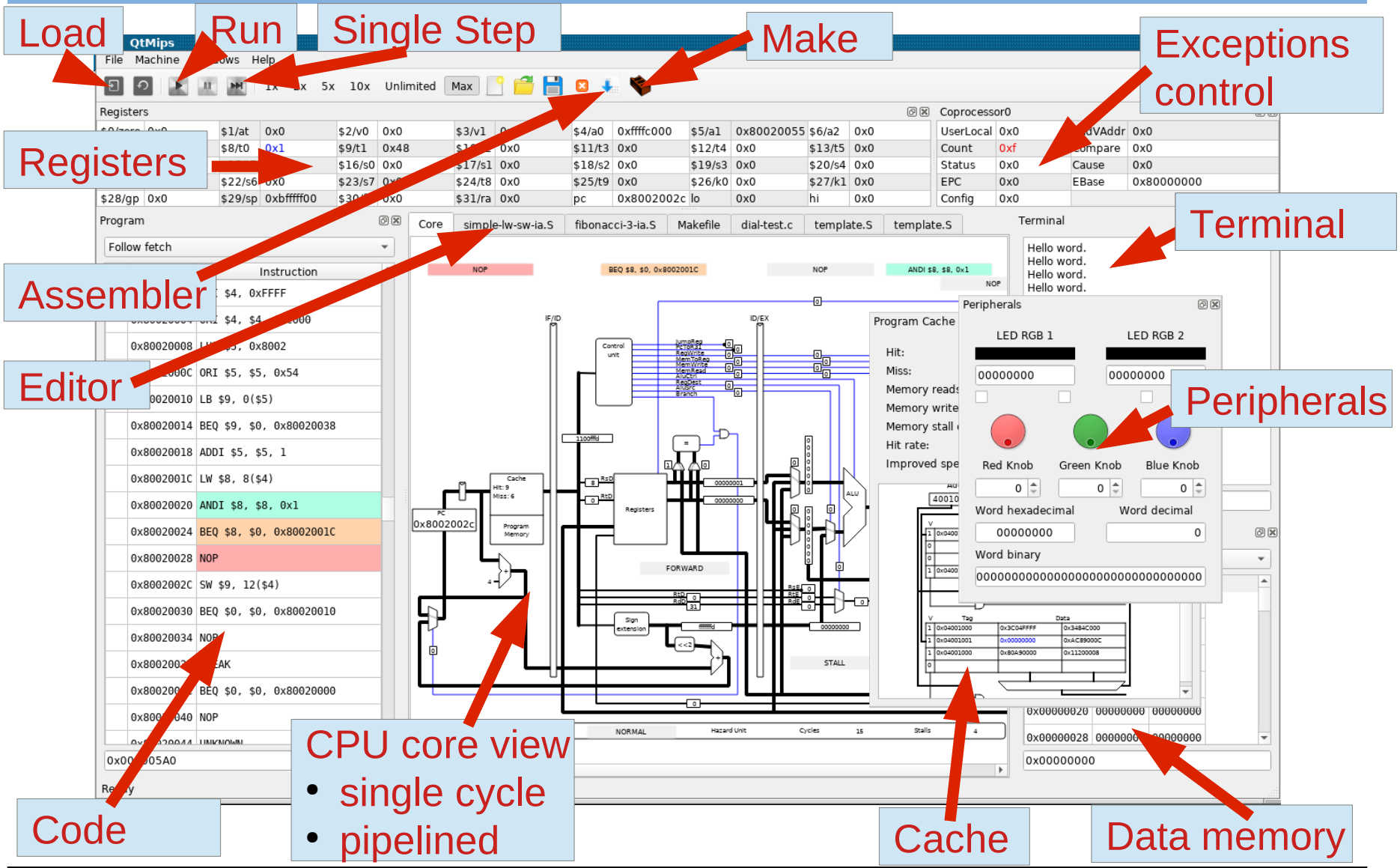
Czech Technical University in Prague

Karel Kočí, **Pavel Píša**, Michal Štepanovský

[*1] https://github.com/cvut/QtMips/

[*2] https://cw.fel.cvut.cz/wiki/courses/b35apo/en/start

QtMips – MIPS Architecture Emulator



QtMips – Download

- Windows, Linux, Mac
  https://github.com/cvut/QtMips/releases
- Ubuntu

  https://launchpad.net/~ppisa/+archive/ubuntu/qtmips
- Suse, Fedora and Debian
  https://software.opensuse.org//download.html?project=home%3Appisa&package=qtmips
- Suse Factory

  https://build.opensuse.org/package/show/Education/qtmips
- Online version

  http://cmp.felk.cvut.cz/~pisa/apo/qtmips/qtmips_gui.html
- MIPS-ELF binutils and GCC for Linux, MAC OS and Windows

  http://cmp.felk.cvut.cz/~pisa/apo/qtmips/

QtMips – Origin and Development

- **MipsIt** used in past for Computer Architecture course at the Czech Technical University in Prague, Faculty of Electrical Engineering
- Diploma theses of Karel Kočí mentored by Pavel Píša

  Graphical CPU Simulator with Cache Visualization

  https://dspace.cvut.cz/bitstream/handle/10467/76764/F3-DP-2018-Koci-Karel-diploma.pdf
- Switch to QtMips in the 2019 summer semester
- Fixes, extension and partial internals redesign by Pavel Píša

- Alternatives:
- SPIM/QtSPIM: A MIPS32 Simulator

  http://spimsimulator.sourceforge.net/
- MARS: IDE with detailed help and hints

  http://courses.missouristate.edu/KenVollmar/MARS/index.htm
- EduMIPS64: 1x fixed and 3x FP pipelines

  https://www.edumips.org/

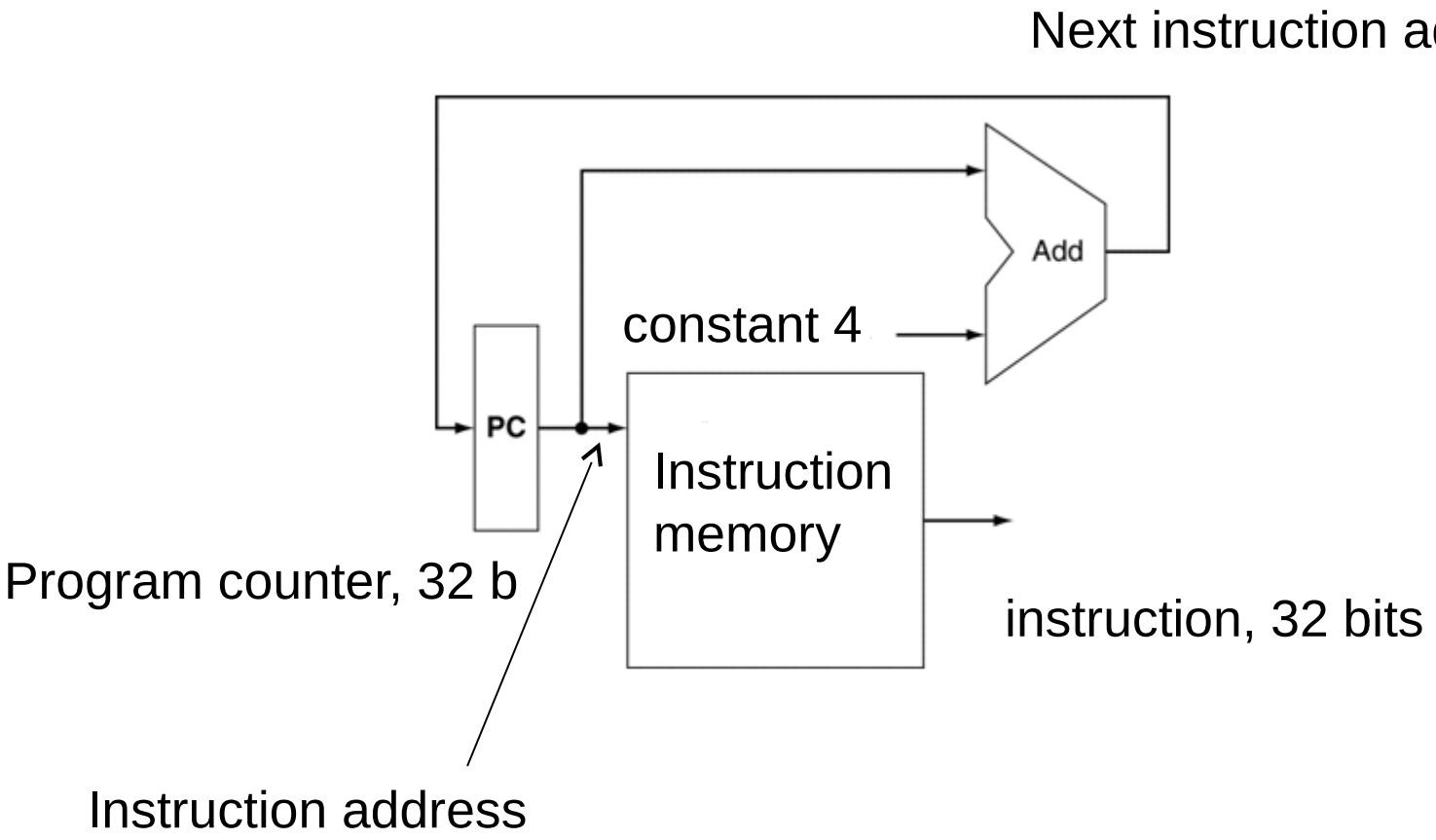
Compilation: C ➞ Assembler ➞ Machine Code

```
int pow = 1;
int x = 0;

while(pow != 128)
{
  pow = pow*2;
  x = x + 1;
}
```

```
addi s0, $0, 1        // pow = 1
addi s1, $0, 0        // x = 0
addi t0, $0, 128      // t0 = 128

while:
  beq  s0, t0, done  // if pow==128, go to done
  sll  s0, s0, 1      // pow = pow*2
  addi s1, s1, 1      // x = x+1
  j    while
done:
```

```
8001FFF4  00 00 00 00          NOP
8001FFF8  00 00 00 00          NOP
8001FFFC  00 00 00 00          NOP
80020000  20 10 00 01  start() ADDI  $16, $00, 0x1
80020004  20 11 00 00          ADDI  $17, $00, 0x0
80020008  20 08 00 80          ADDI  $08, $00, 0x80
8002000C  12 08 00 04  while:  BEQ   $08, $16, 0x4
80020010  00 00 00 00          NOP
80020014  00 10 80 40          SLL   $16, $16, 1
80020018  08 00 80 03          J     0x8003
8002001C  22 31 00 01          ADDI  $17, $17, 0x1
80020020  00 00 00 00  done:   NOP
80020024  00 00 00 00          NOP
80020028  00 00 00 00          NOP
```

Hardware realization of basic (main) CPU cycle



The goal of this lecture

- To understand the implementation of a simple computer consisting of CPU and separated instruction and data memory
- Our goal is to implement following instructions:
  - Read and write a value from/to the data memory

    `lw` – load word, **`sw`** – store word
  - Arithmetic and logic instructions

    `add`, **`sub`**, **`and`**, **`or`**, **`slt`**
  - Program flow change/jump instruction **`beq`**
- CPU will consist of control unit and ALU.
- Notes:
  - The implementation will be minimal (single cycle CPU – all operations processed in the single step/clock period)
  - The lecture 4 focuses on more realistic pipelined CPU implementation

The instruction format and instruction types

- The three types of the instructions are considered:

Type	31…					0
R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0		
J	**opcode**(6), 31:26	**address**(26), 25:0				

- the R type instructions → opcode=000000, funct – operation
- rs – source, rd – destination, rt – source/destination
- shamt – for shift operations, immediate – direct operand

- 5 bits allows to encode 32 GPRs ($0 is hardwired to 0/discard)

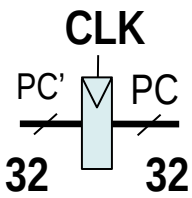
Opcode encoding

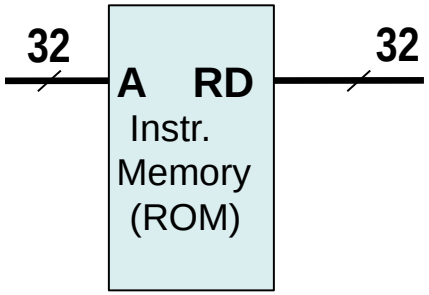
Decode opcode to the ALU operation
- Load/Store (I-type): F = add – add offset to the address base
- Branch (I-type): F = subtract – used to compare operands
- R-type: F depends on funct field

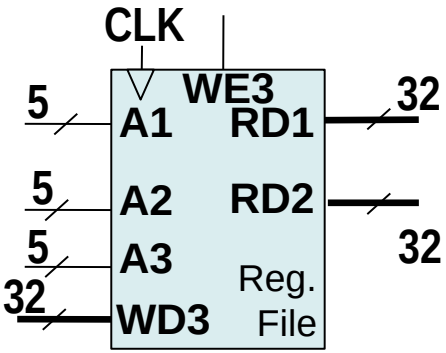
There are more I-type operations which use ALU in the real MIPS ISA

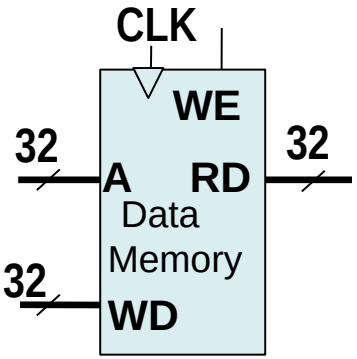
Instruction	Opcode	Func	Operation	ALU function	ALU control
lw	100011	XXXXXX	load word	add	0010
sw	101011	XXXXXX	store word	add	0010
beq	000100	XXXXXX	branch equal	subtract	0110
add	000000 R-type	100000	add	add	0010
sub		100010	subtract	subtract	0110
and		100100	AND	AND	0000
or		100101	OR	OR	0001
slt		101010	set-on-less-than	set-on-less-than	0111

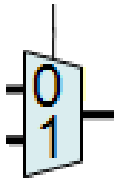
CPU building blocks







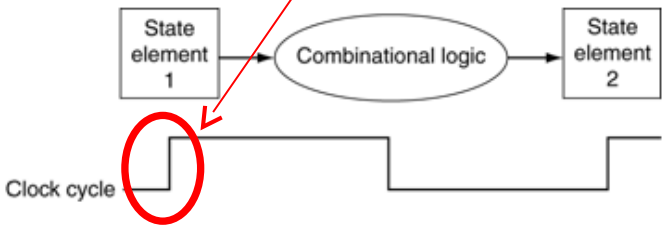




Multiplexer

Write at the rising edge of CLK when WE = 1

Read after “enough time” for data propagation



The load word instruction

`lw` – **load word** – load word from data memory into a register

Description	A word is loaded into a register from the specified address
Operation:	$t = MEM[$s + offset];
Syntax:	lw $t, offset($s)
Encoding:	`1000 11ss ssst tttt iiii iiii iiii iiii`

Example: Read word from memory address 0x4 into register number 11:
lw $11, 0x4($0)

```
1000 11ss ssst tttt iiii iiii iiii iiii
1000 1100 0000 1011 0000 0000 0000 0100
```

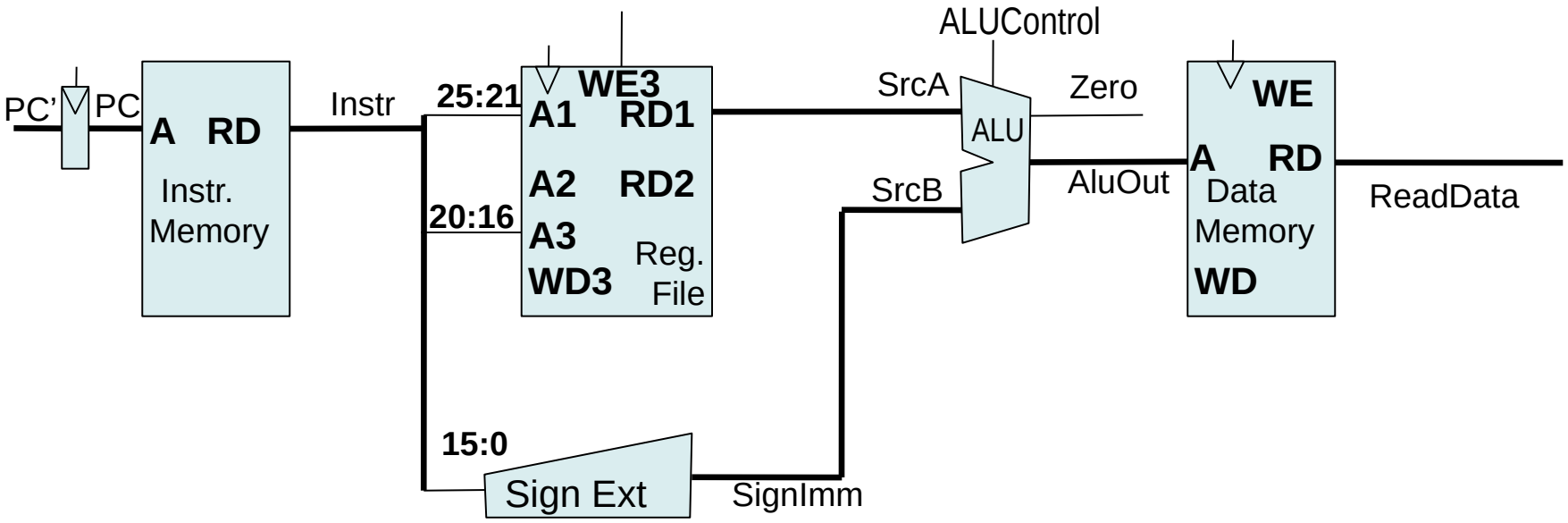
0 11 4

0x 8C 0B 00 04 – machine code for instruction lw $11, 0x4($0)
Note: Register $0 is hardwired to the zero

Single cycle CPU – implementation of the load instruction

`lw`: type I, rs – base address, imm – offset, rt – register where to store fetched data

I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0

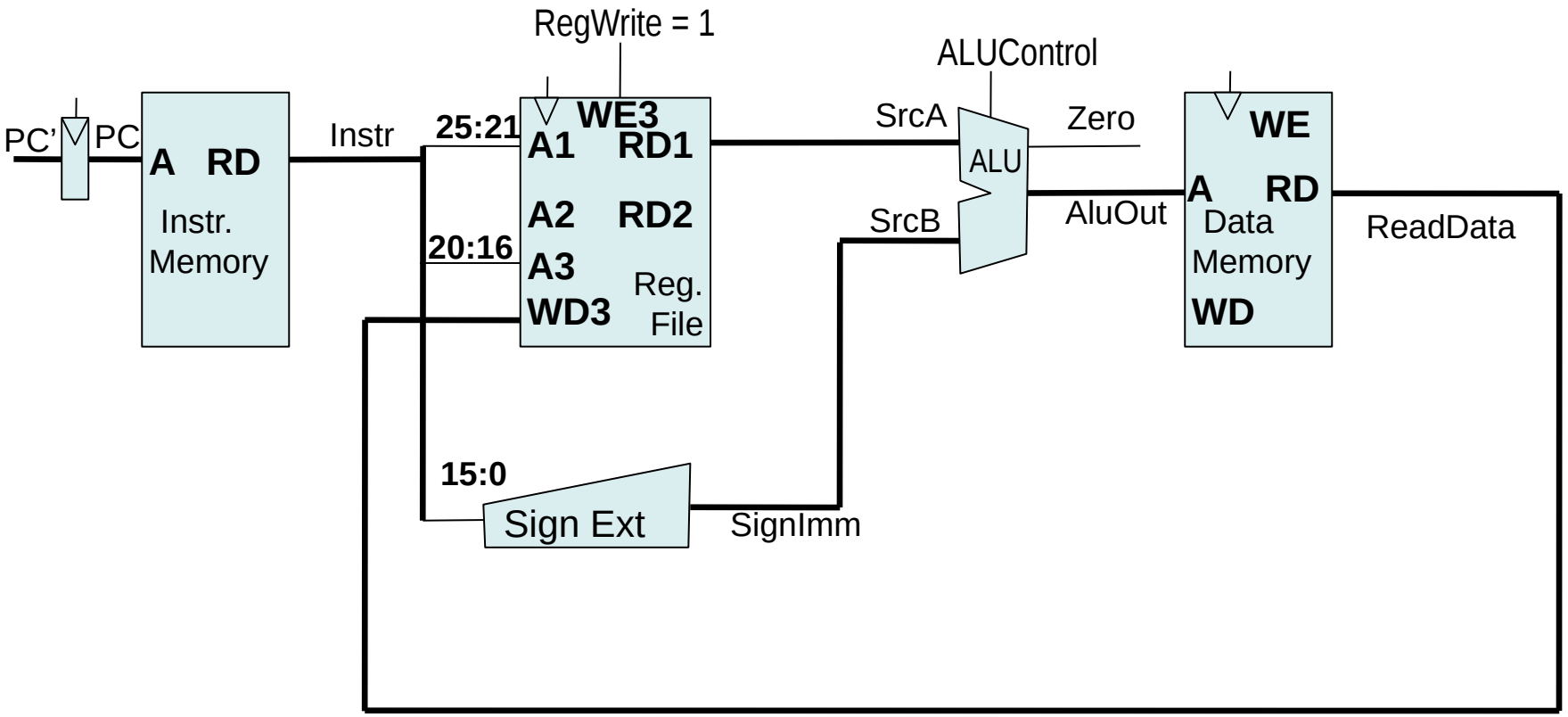


Single cycle CPU – implementation of the load instruction

lw: type I, rs – base address, imm – offset, rt – register where to store fetched data

I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0

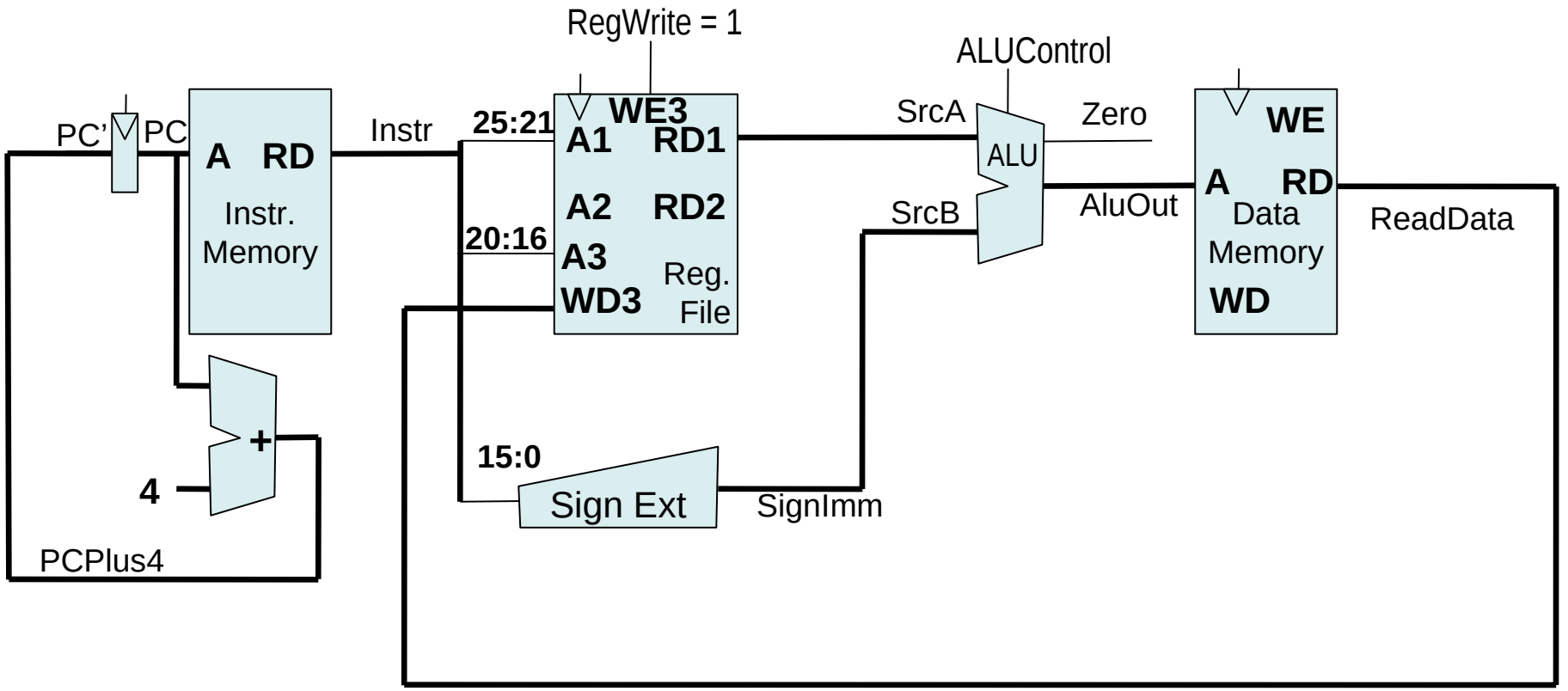
Write at the rising edge of the clock



Single cycle CPU – implementation of the load instruction

lw: type I, rs – base address, imm – offset, rt – register where to store fetched data

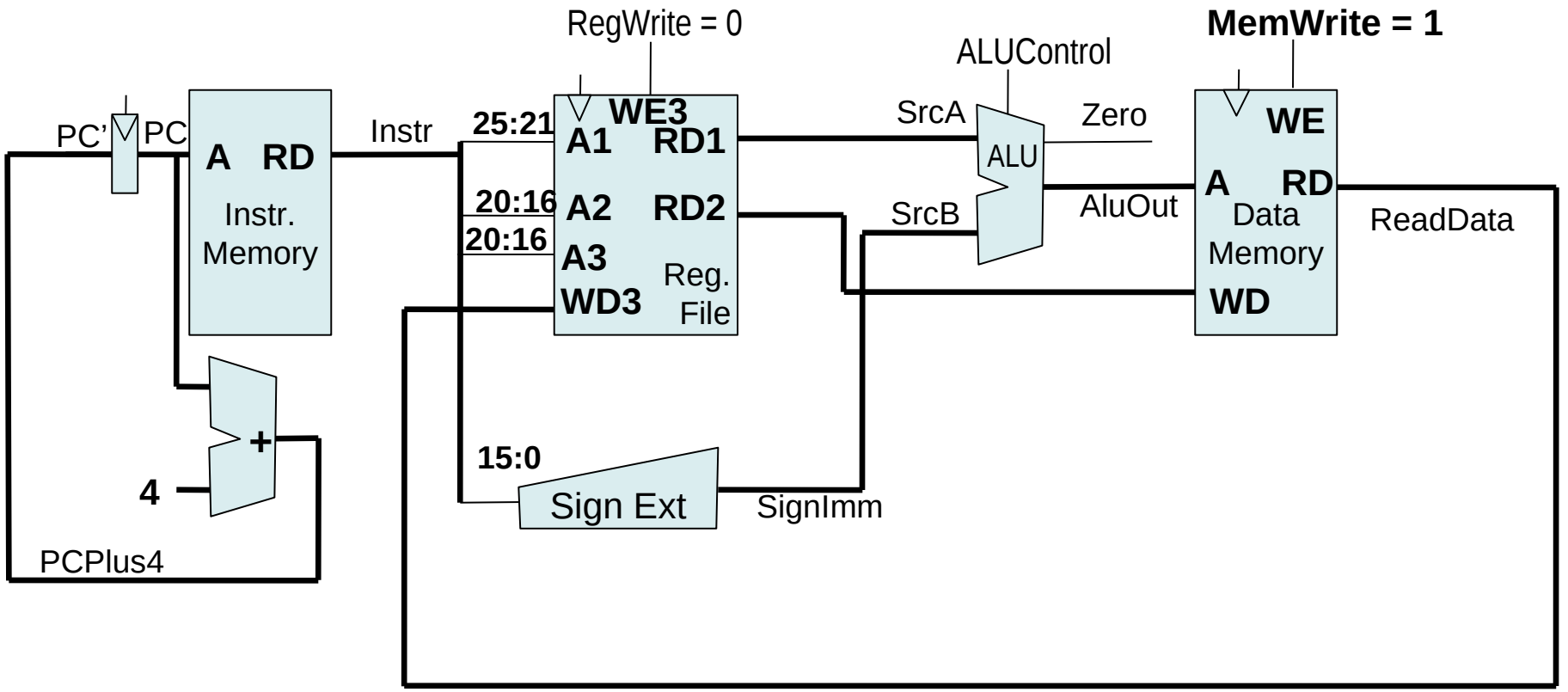
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Single cycle CPU – implementation of the store instruction

sw: type I, rs – base address, imm – offset, rt – select register to store into memory

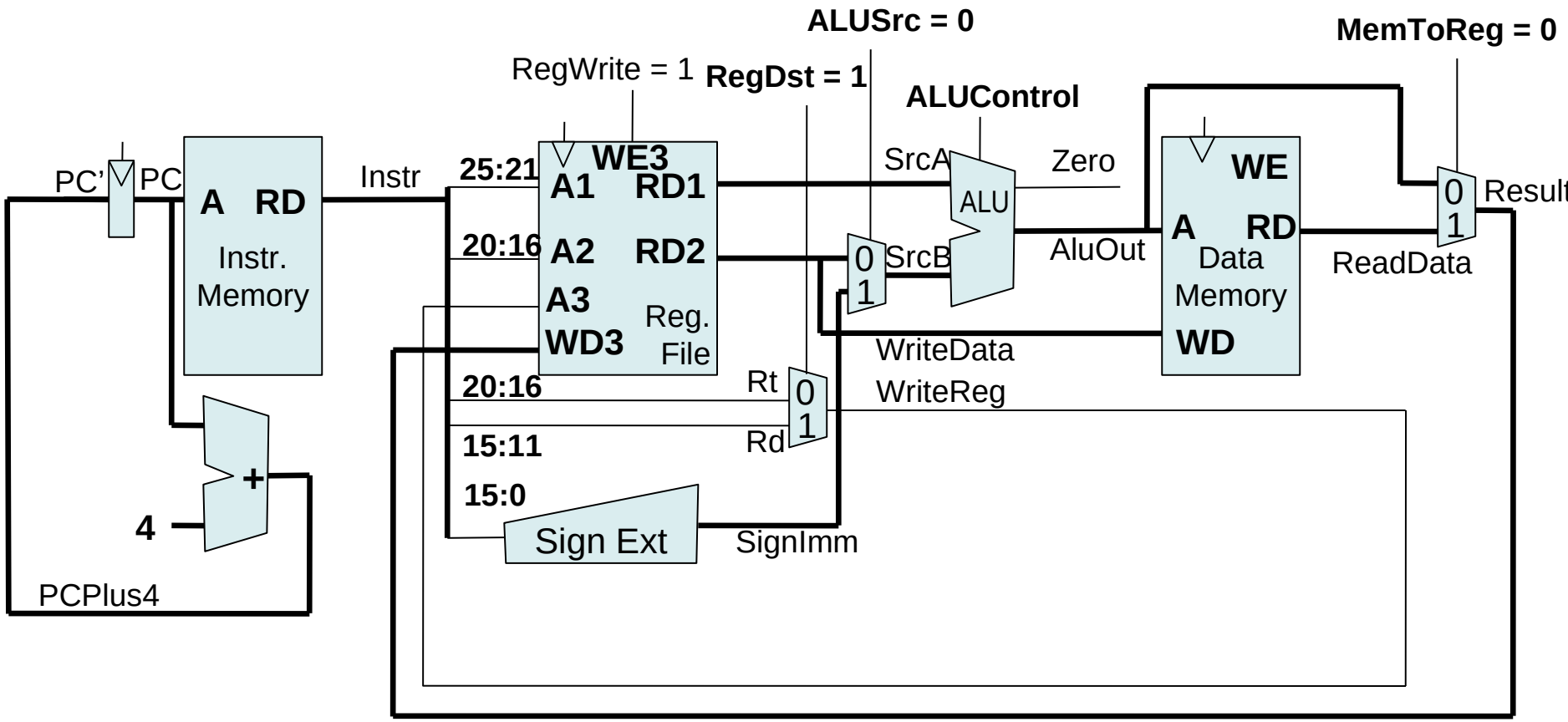
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Single cycle CPU – implementation of the **add** instruction

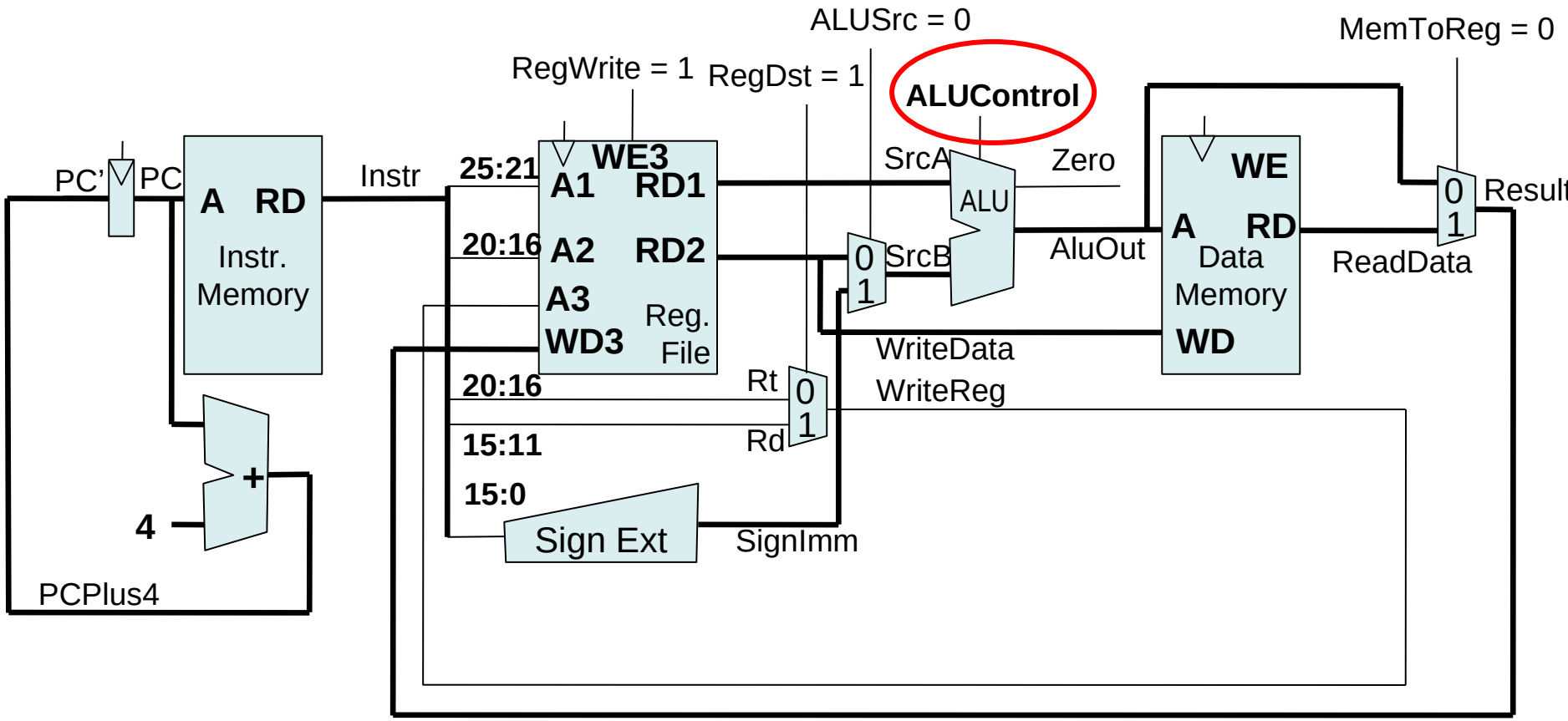
add: type R, rs, rt – source, rd – destination, funct – select ALU operation = add

R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0



Single cycle CPU – **sub**, **and**, **or**, **slt**

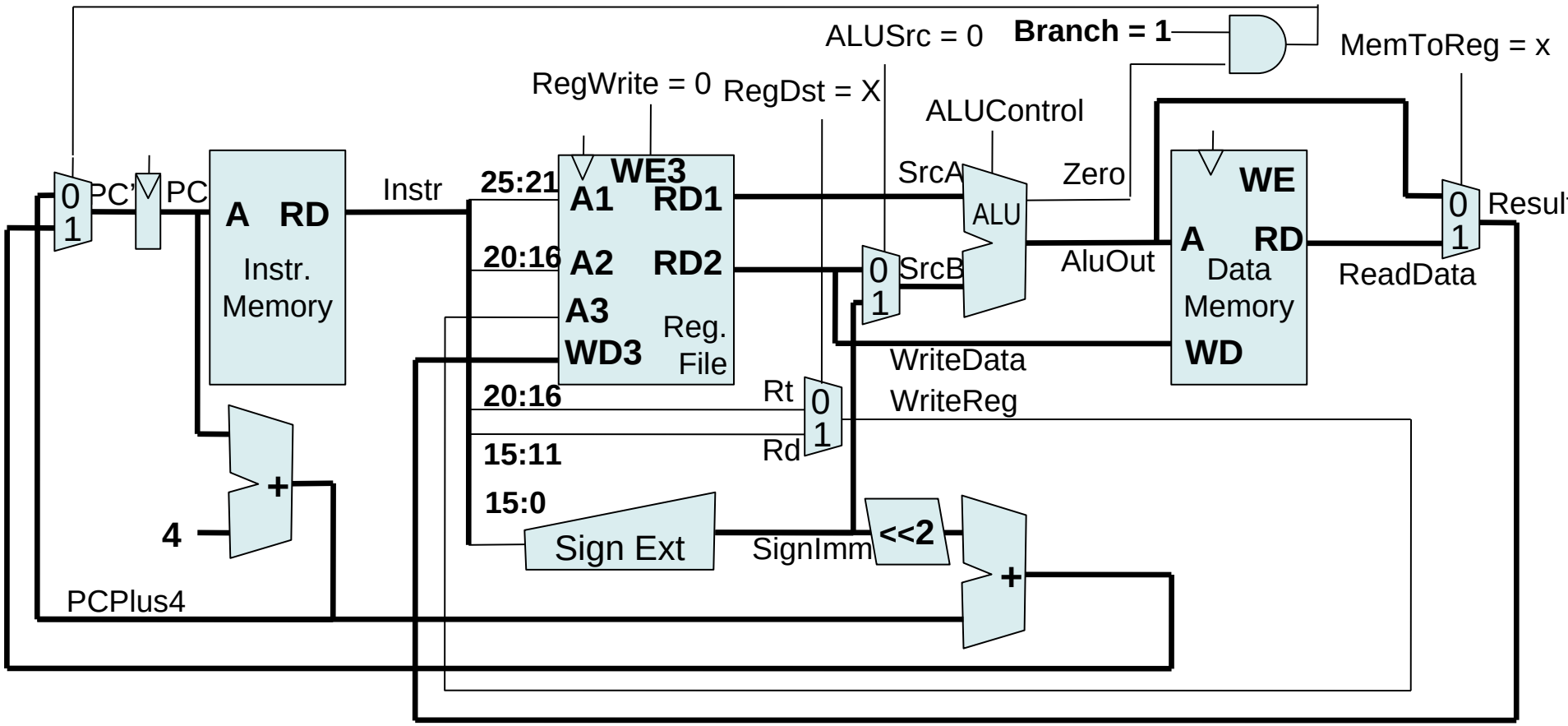
Only difference is another ALU operation selection (ALUcontrol). The data path is the same as for **add** instruction



Single cycle CPU – implementation of **beq**

beq – branch if equal; imm–offset; PC´ = PC+4 + SignImm*4

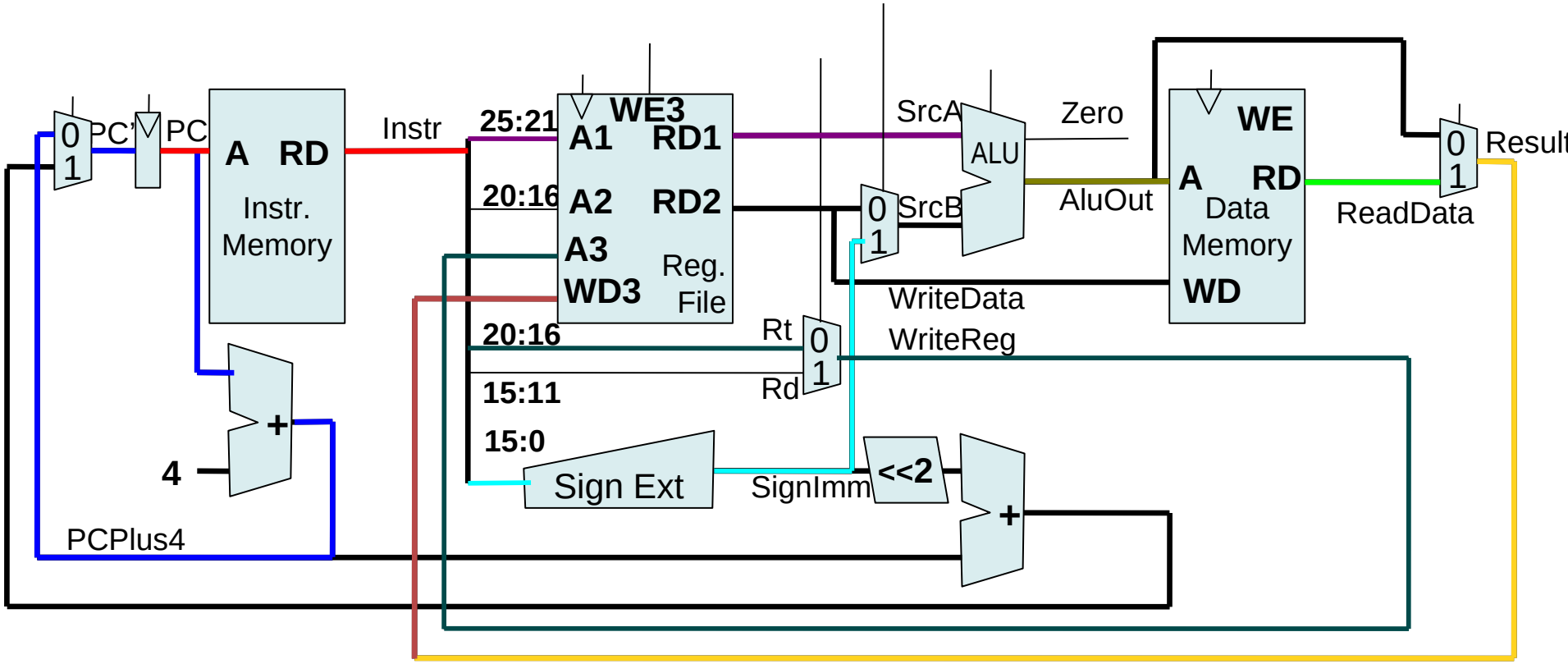
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0



Single cycle CPU – Throughput: IPS = IC / T = $IPC_{str}.f_{CLK}$

- What is the maximal possible frequency of the CPU?
- It is given by latency on the critical path – it is **`lw`** instruction in our case:

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$



Single cycle CPU – Throughput: IPS = IC / T = $IPC_{str}.f_{CLK}$

- What is the maximal possible frequency of the CPU?
- It is given by latency on the critical path – it is **`lw`** instruction in our case:

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$

Consider following parameters:

- t_{PC} = 30 ns
- t_{Mem} = 300 ns
- t_{RFread} = 150 ns
- t_{ALU} = 200 ns
- t_{Mux} = 20 ns
- $t_{RFsetup}$ = 20 ns

Then T_c = 1020 ns → f_{CLK} max = 980 kHz,

IPS = 980e3 = 980 000 instructions per second

Notes

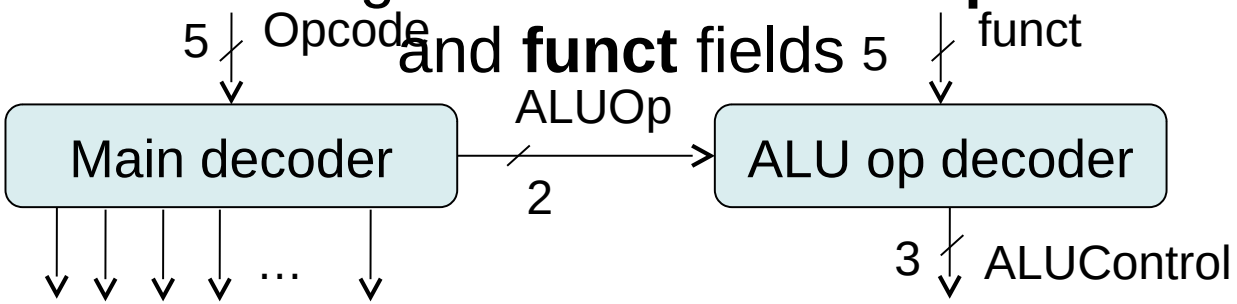
- Remember the result, so you can compare it with result for pipelined CPU during lecture 4
- You should compare this with actual 30e9 IPS per core, i.e. total 128 300 MIPS for today high-end CPUs
- How many clever enhancements in hardware and programming/compilers are required for such advance!!!
- After this course you should see behind the first two hills on that road.

- We will continue with control unit implementation and its function

Single cycle CPU – Control unit

R	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**rd**(5), 15:11	**shamt**(5)	**funct**(6), 5:0
I	**opcode**(6), 31:26	**rs**(5), 25:21	**rt**(5), 20:16	**immediate** (16), 15:0		
J	**opcode**(6), 31:26	**address**(26), 25:0				

Control signals values reflect **opcode** and **funct** fields



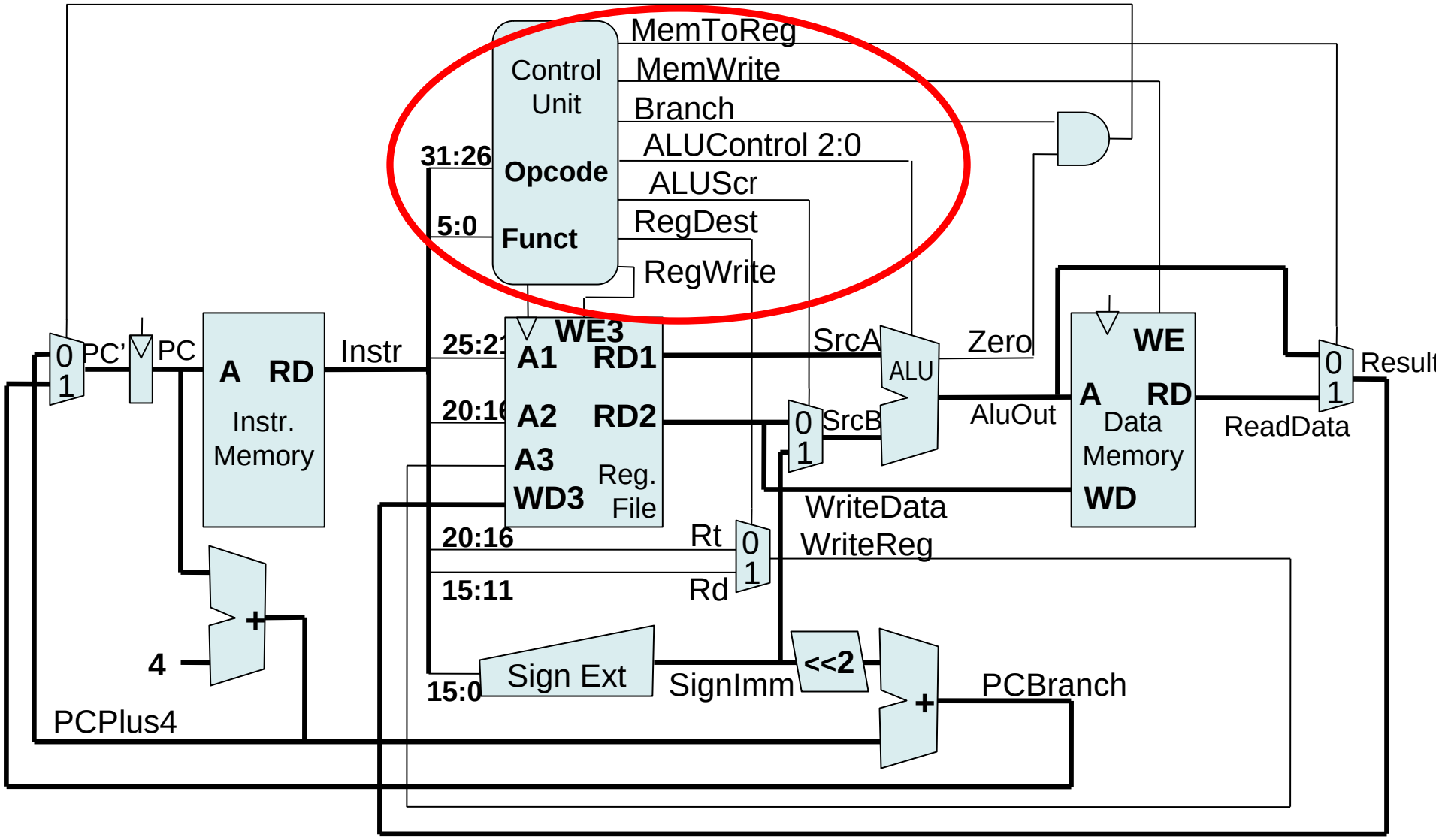
ALUOp	
00	addition
01	subtraction
10	according to funct
11	-not used-

	Opcode	Reg Write	RegDst	ALUSrc	ALUOp	Branch	Mem Write	MemTo Reg
R-type	000000	1	1	0	10	0	0	0
lw	100011	1	0	1	00	0	0	1
sw	101011	0	X	1	00	0	1	X
beq	000100	0	X	0	01	1	0	X

ALU Control (ALU function decoder)

ALUOp (selector)	Funct	ALUControl
00	X	010 (add)
01	X	110 (sub)
1X	add (100000)	010 (add)
1X	sub (100010)	110 (sub)
1X	and (100100)	000 (and)
1X	or (100101)	001 (or)
1X	slt (101010)	111 (set les than)

The control unit of the single cycle cpu



Pipelined instructions execution

Suppose that instruction execution can be divided into 5 stages:

IF – Instruction Fetch, ID – Instruction decode (and Operands Fetch),
EX – Execute, MEM – Memory Access, WB – Write Back

and $\tau = \max \{ \tau_i \}_{i=1}^{k}$, where τ_i is time required for signal propagation (*propagation delay)* through *i*-th stage.

IF – setup PC for memory and fetch pointed instruction. Update PC = PC+4

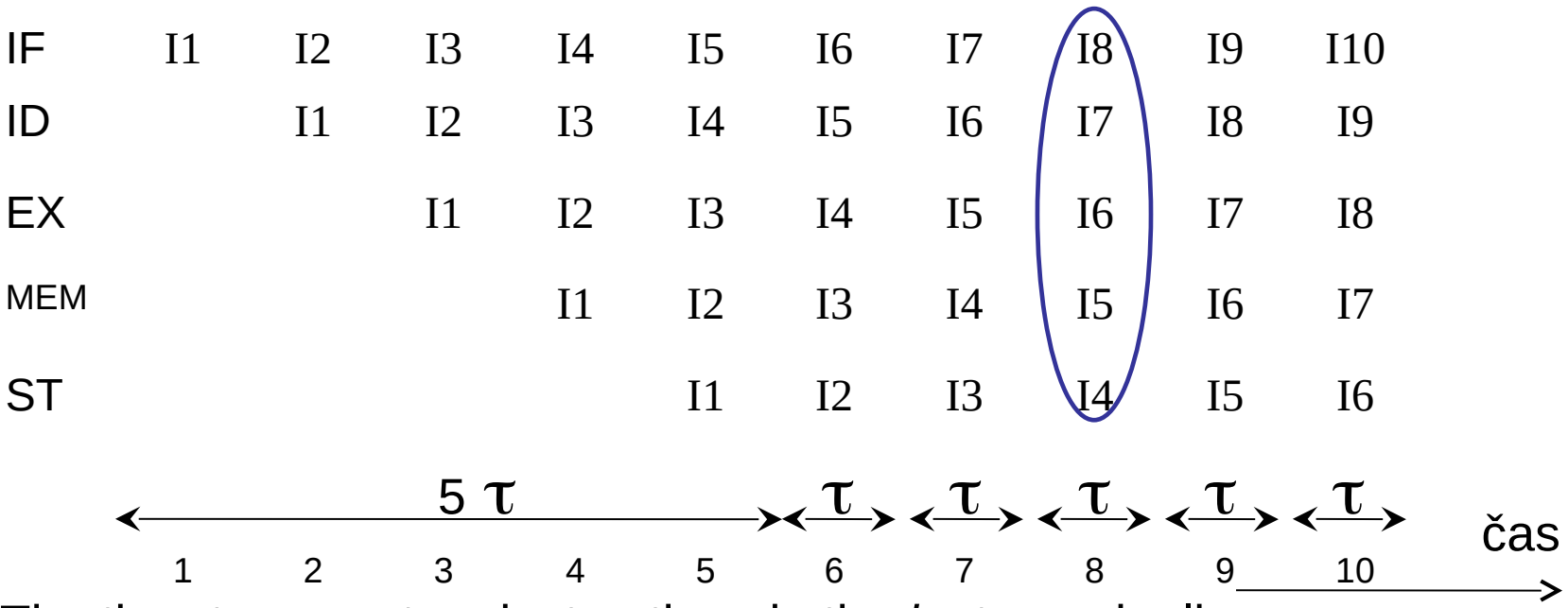
ID – decode the opcode and read registers specified by instruction, check for equality (for possible `beq` instruction), sign extend offset, compute branch target address for branch case (this is means to extend offset and add PC)

EX – execute function/pass register values through ALU

MEM – read/write main memory for *load/store* instruction case

WB – write result into RF for instructions of register-register class or instruction *load* (result source is ALU or memory)

Instruction-level parallelism - pipelining



- The time to execute *n* instructions in the *k*-stage pipeline:

$$T_k = k.\tau + (n-1)\,\tau$$

- Speedup: $S_k = \frac{T_1}{T_k} = \frac{nk\,\tau}{k\tau + (n-1)\tau}$ $\lim_{n\to\infty} S_k = k$

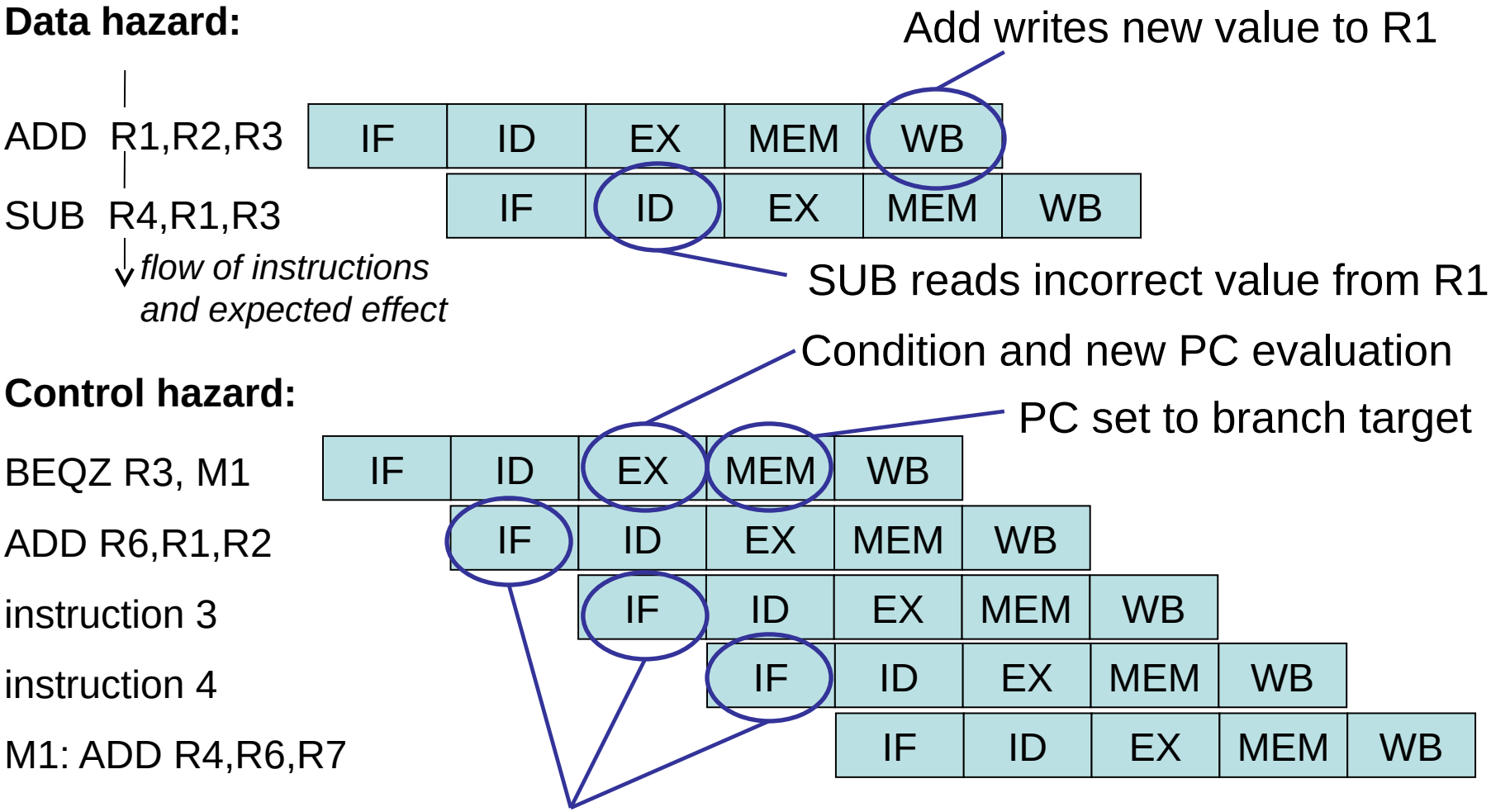
Prerequisite: pipeline is optimally balanced, circuit can arbitrarily divided

Instruction-level parallelism - pipelining

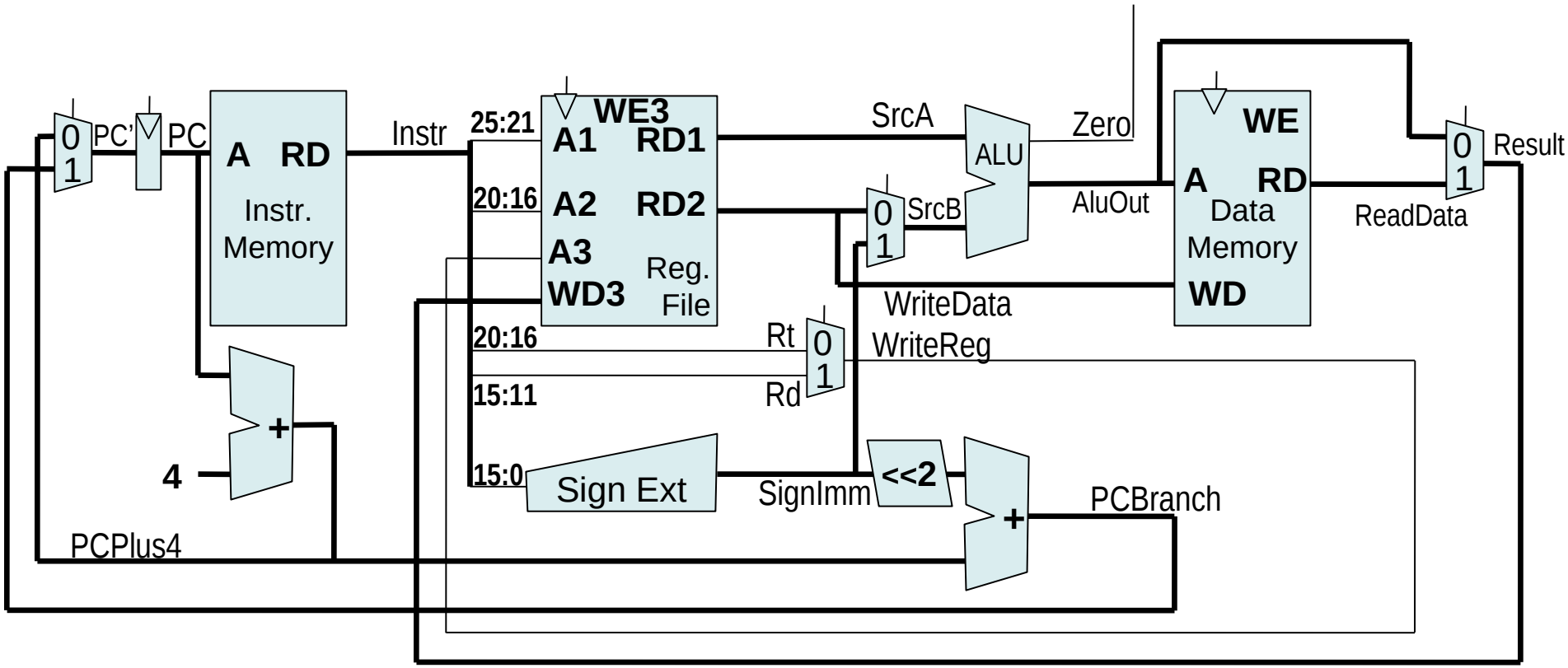
- Does not reduce the execution time of individual instructions, effect is just the opposite...
- Hazards:
  - structural (resolved by duplication),
  - data (result of data dependencies: RAW, WAR, WAW)
  - control (caused by instructions which change PC)...
- Hazard prevention can result in pipeline stall or pipeline flush
- Remark : Deeper pipeline (more stages) results in shorter sequences of gates in each stage which enables to increase the operating frequency of the processor…, but more stages means higher overhead (demand to arrange better instructions into pipeline and result in more significant lag in the case of stall or pipeline flush)

Instruction-level parallelism – Semantics violations

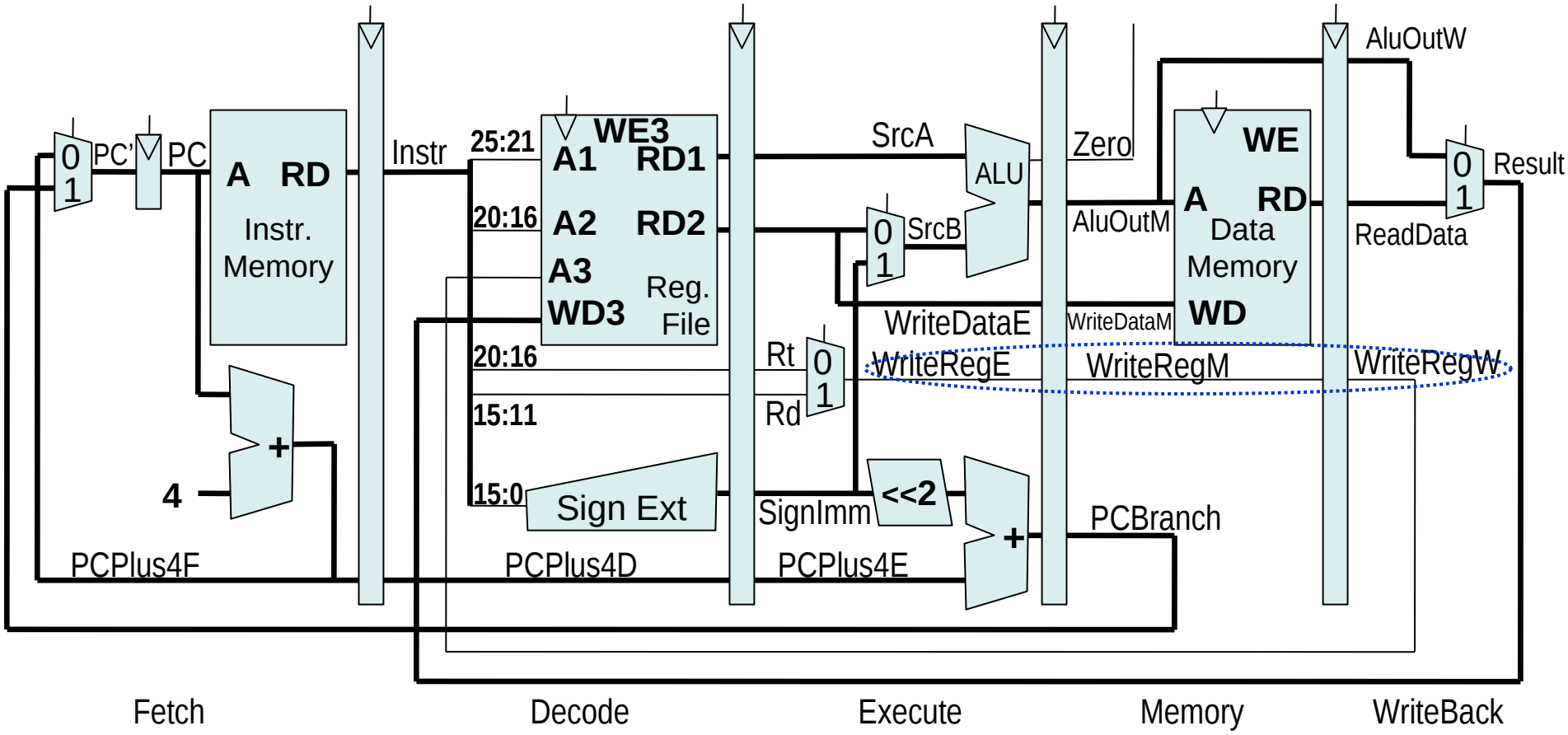
Data hazard:



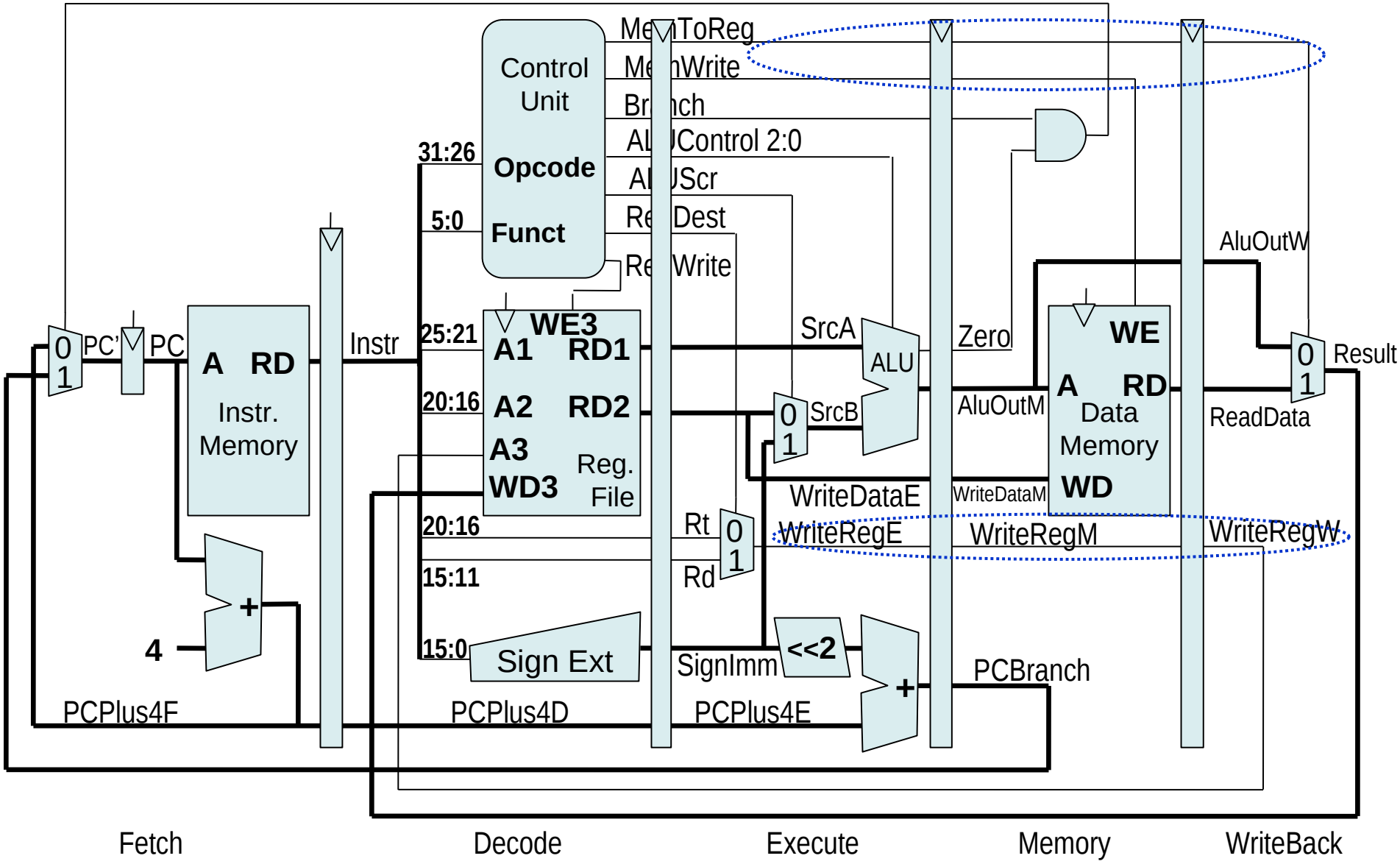
Non-pipelined execution



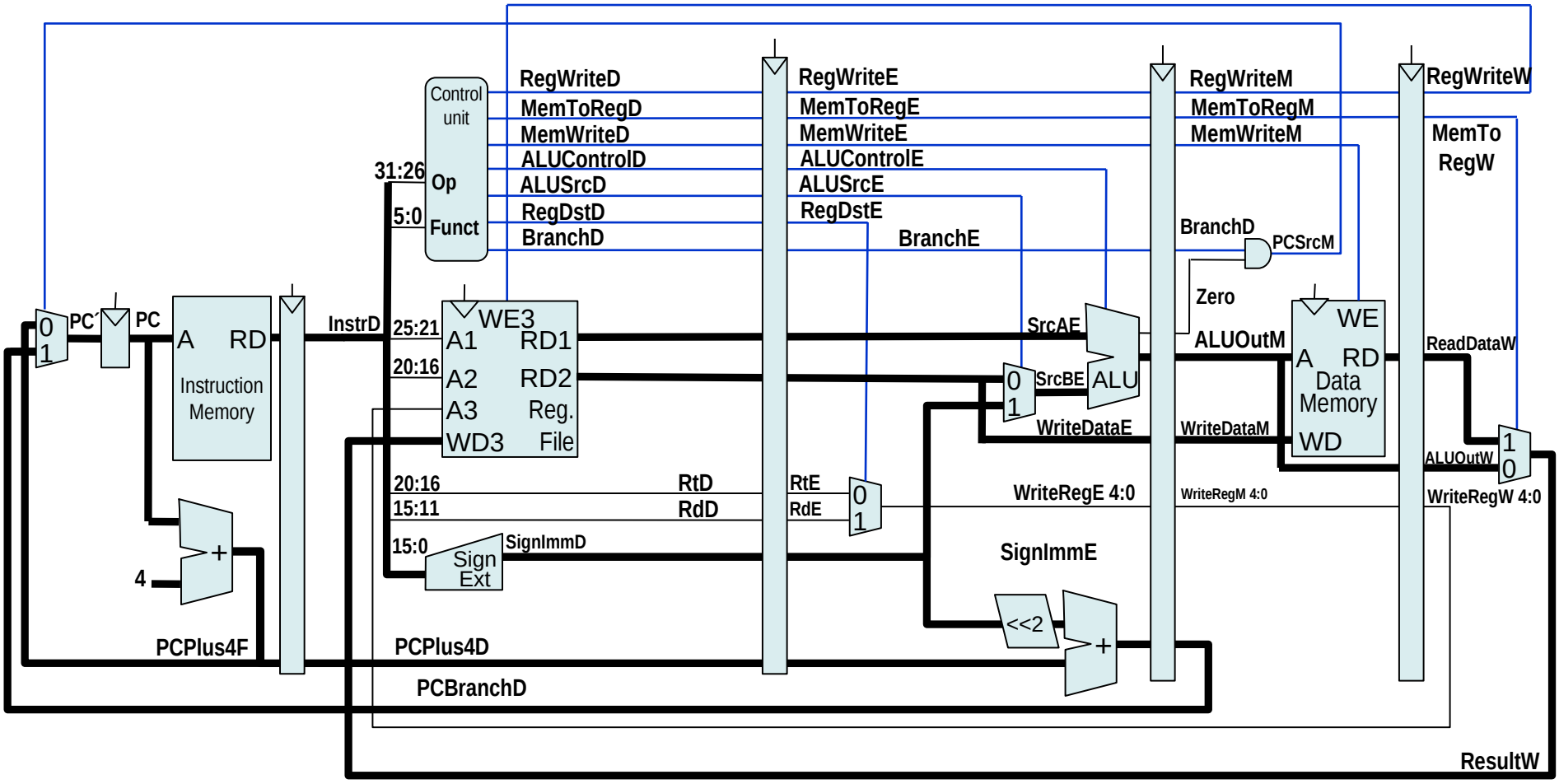
Pipelined execution



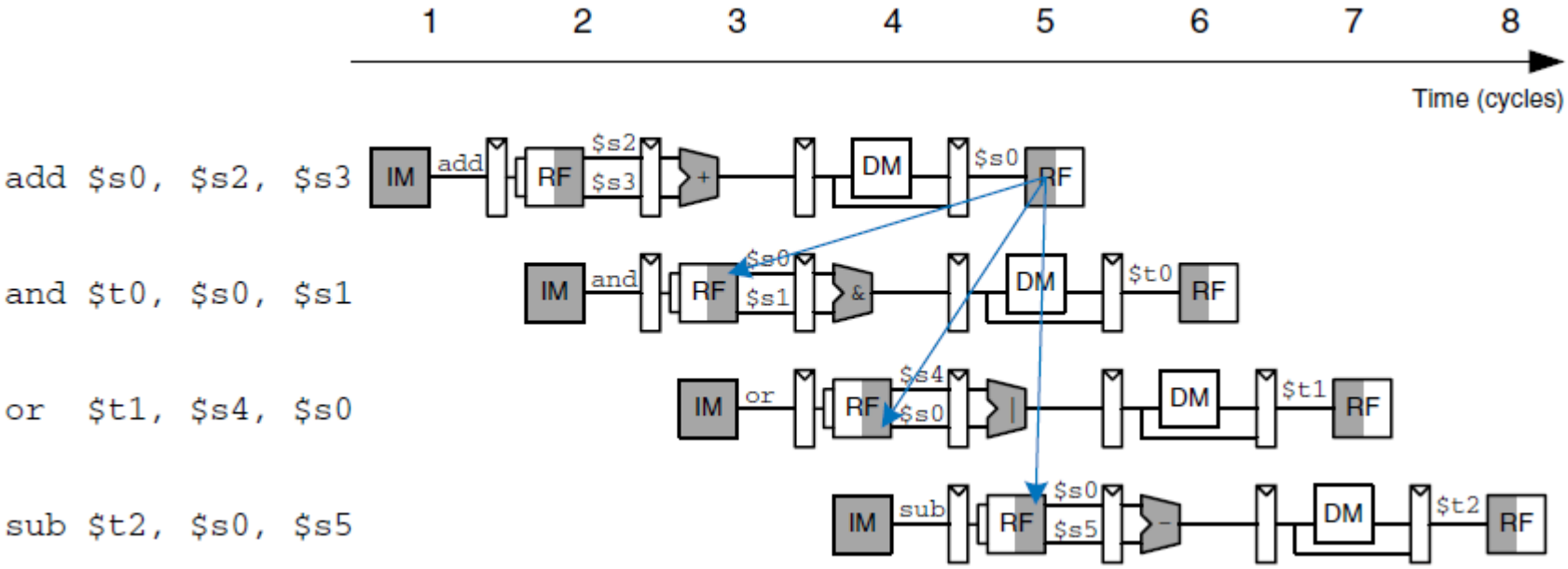
Pipelined execution



The same design but drawn scaled down…

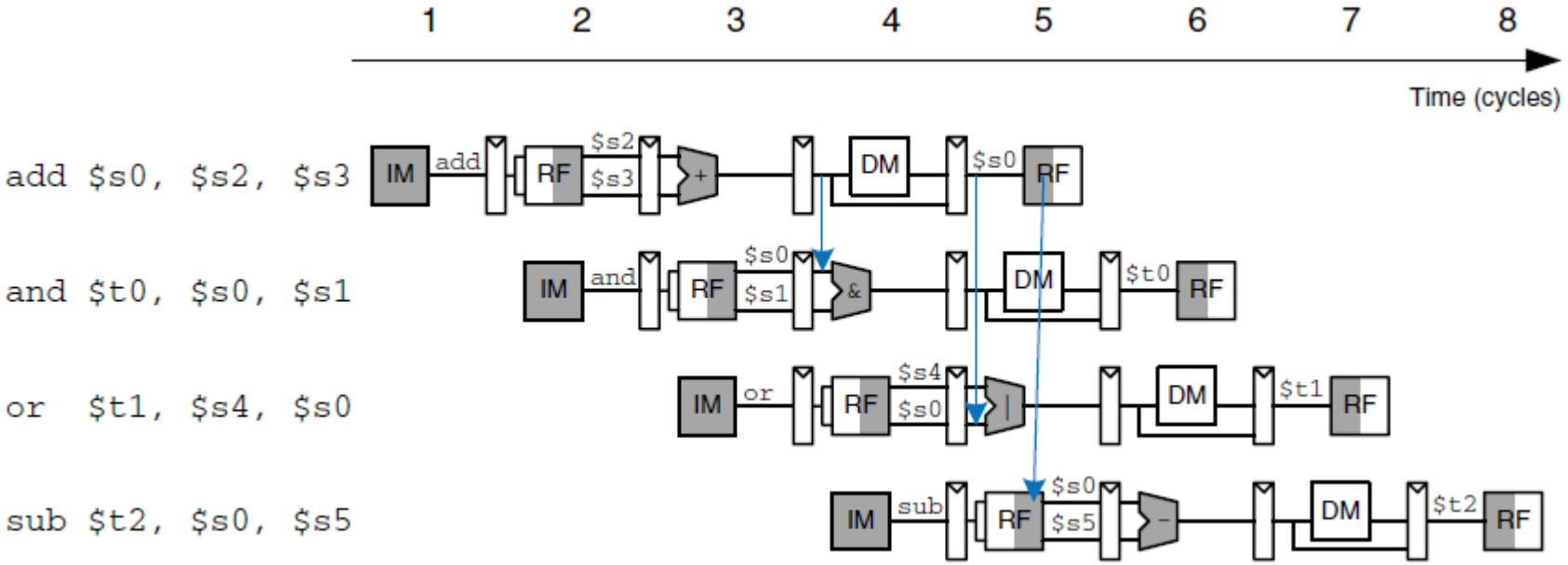


Cause of the data hazards



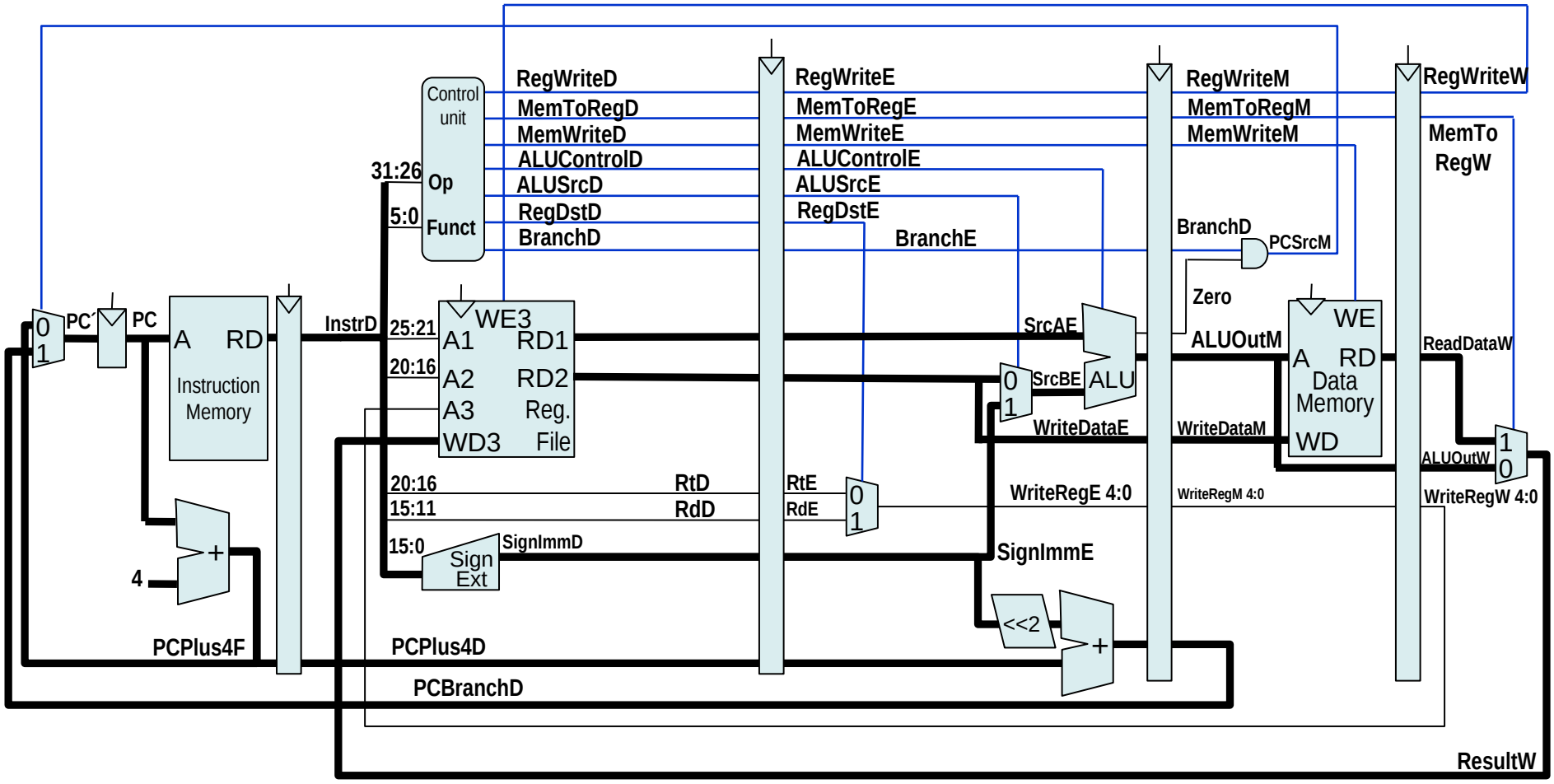
- Register File – access from two pipeline stages (Decode, WriteBack) – actual write occurs at the first half of the clock cycle, the read in the second half ⇒ there is no hazard for `sub $s0` input operand
- RAW (Read After Write) hazard – `and` (`or`) requires `$s0` in 3 (4)
- How can such hazard be prevented without pipeline throughput degradation?

Forwarding to avoid data hazards

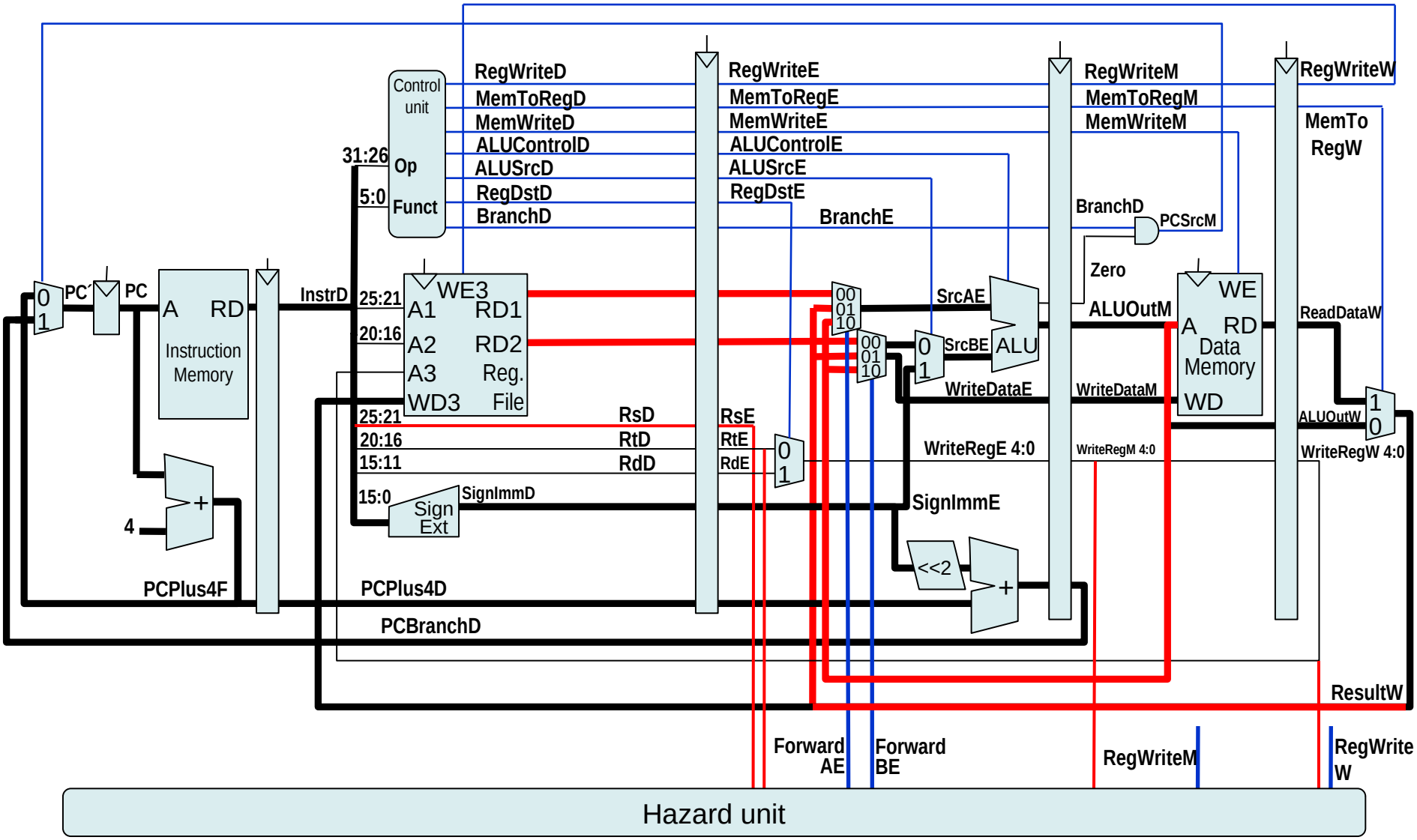


- If a result is available (computed) before subsequent instruction(s) requires the value then data hazard can be avoided by forwarding
- Hazard case is indicated when some of source registers in EX stage is the same as destination register in stage MEM or WB
- The register numbers are fed to the Hazard Unit
- The RegWrite signal from MEM and WB stage has to be monitored as well to check that register number on WriteReg lines takes effect – lw / sw etc.

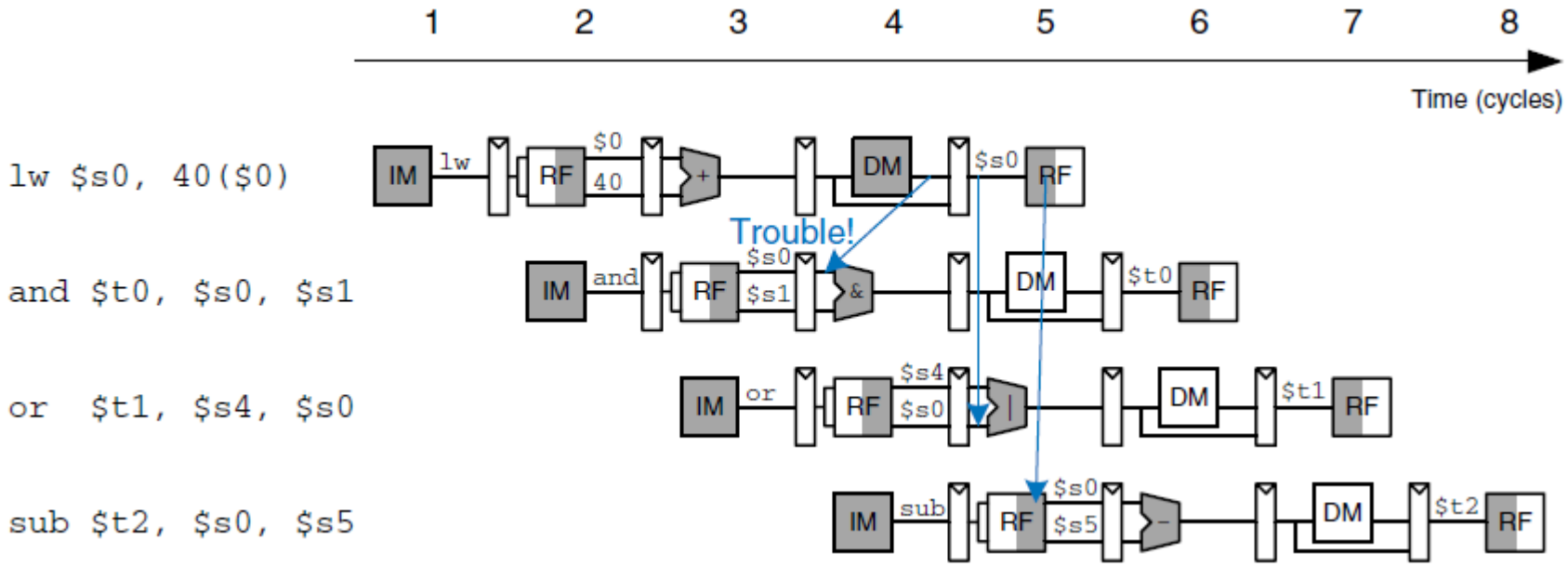
CPU after previous design steps



Data hazards solved by forwarding

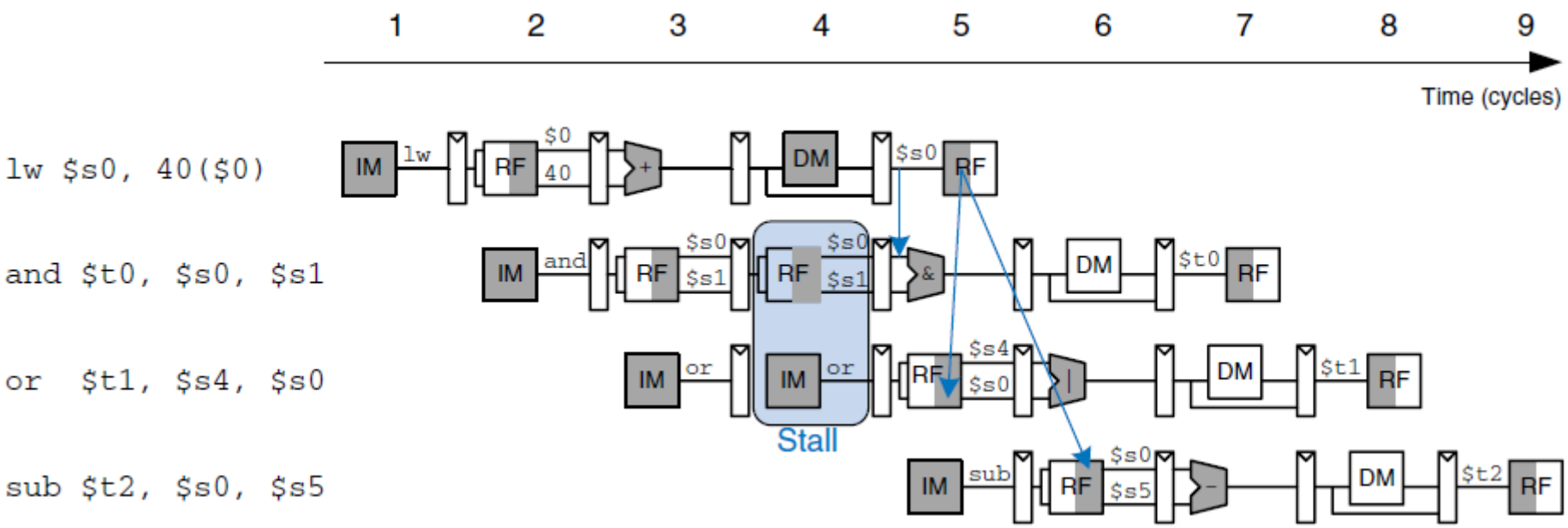


Data hazard avoided by pipeline stall



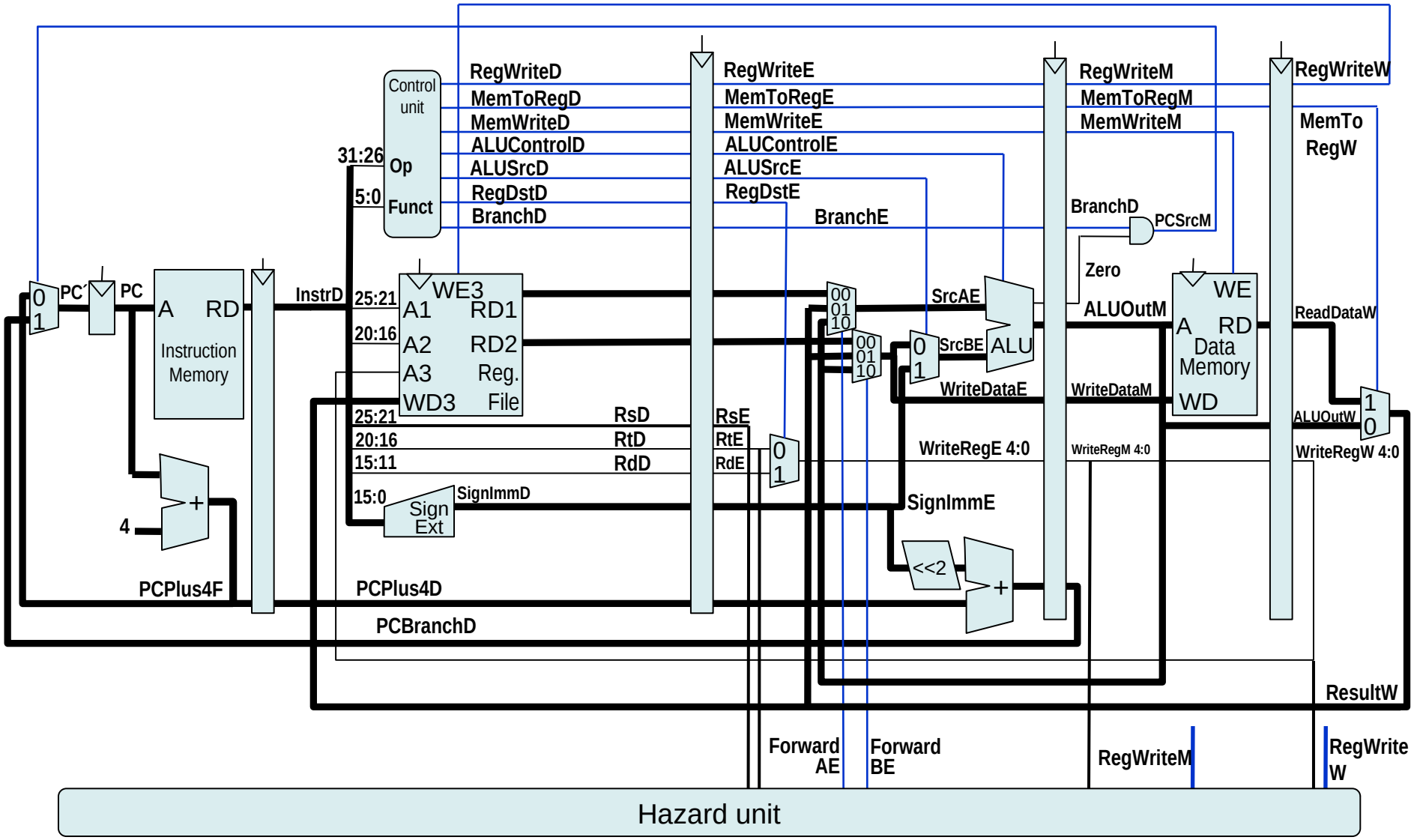
- If subsequent instructions require result before it is available in CPU then the pipeline has to be stalled (stall state inserted)
- The stall is mean to solve hazard but affect system throughput
- Pipeline stages preceding that one which is affected by the hazard are stalled until all results required by subsequent instructions are available – results are forwarded to the sink which required their value

Data hazard avoided by pipeline stall

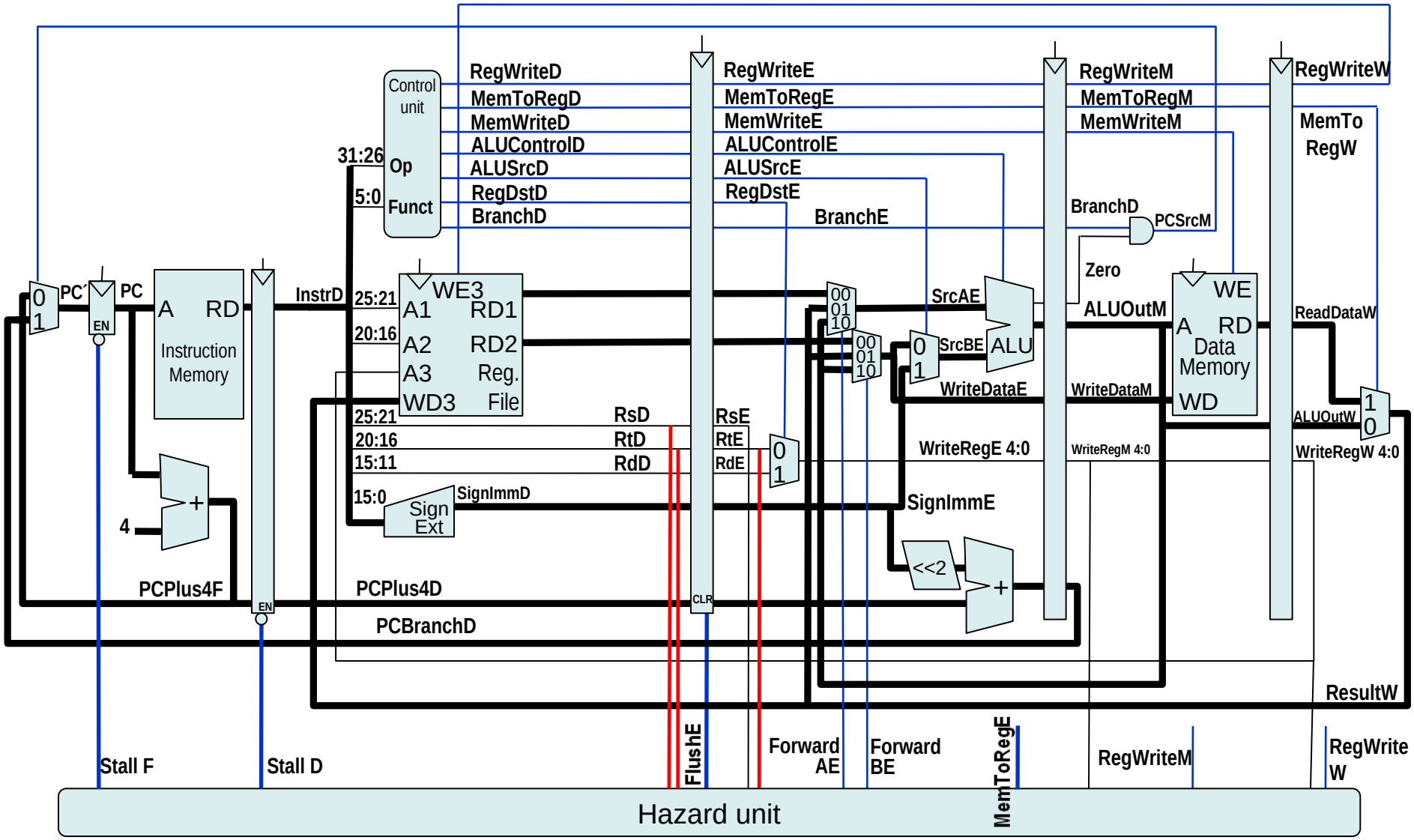


- The stall is realized by the holding content of the inter-stage registers (gating their clocks or blocking their latch enable signals)
- Results from colliding stages have to be „discarded“ – certain control signals in CPU (RF or memory write enable, branch gating) are reset (held low)
- Both is achieved by introduction of control signals to hold and/or reset inter-stages registers

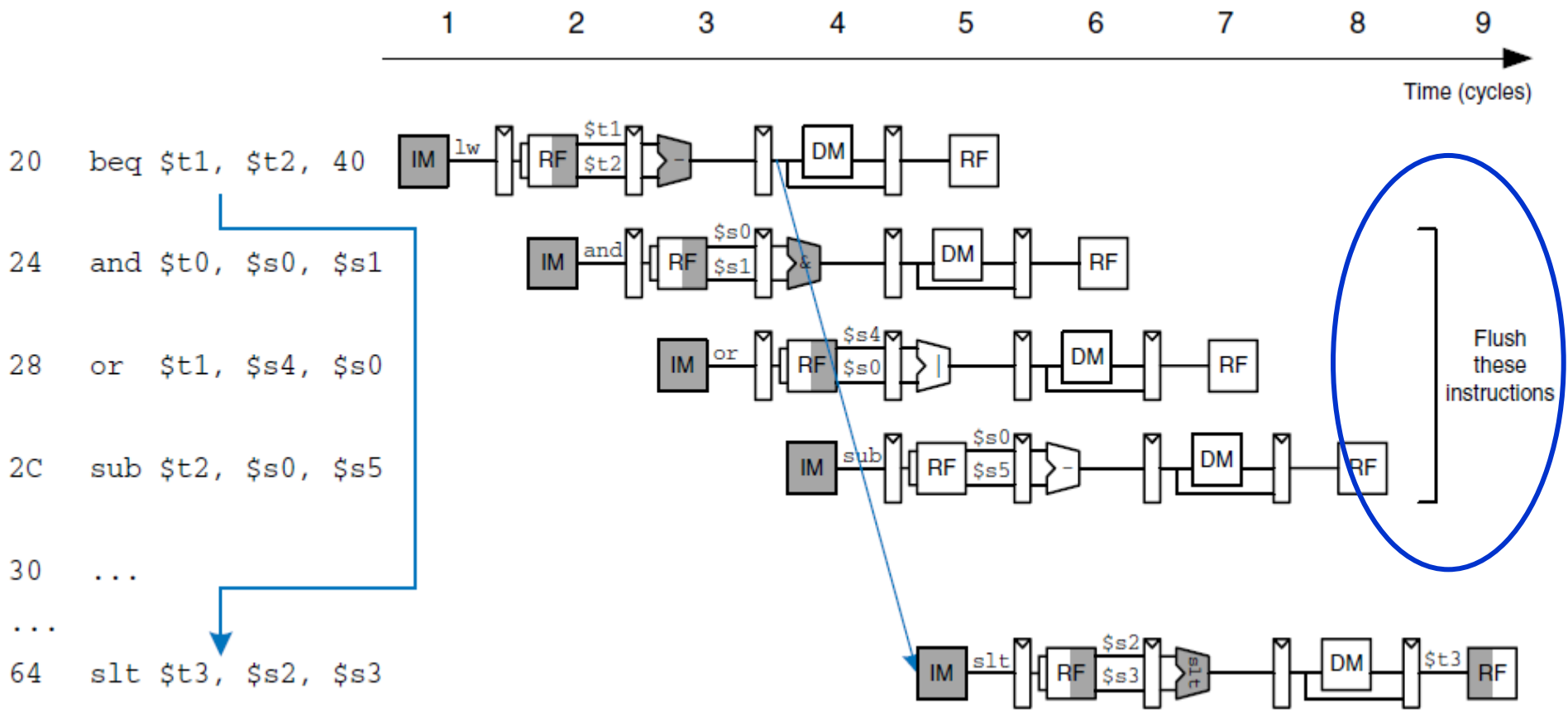
Processor design build till now



Processor with data hazards avoided by stall

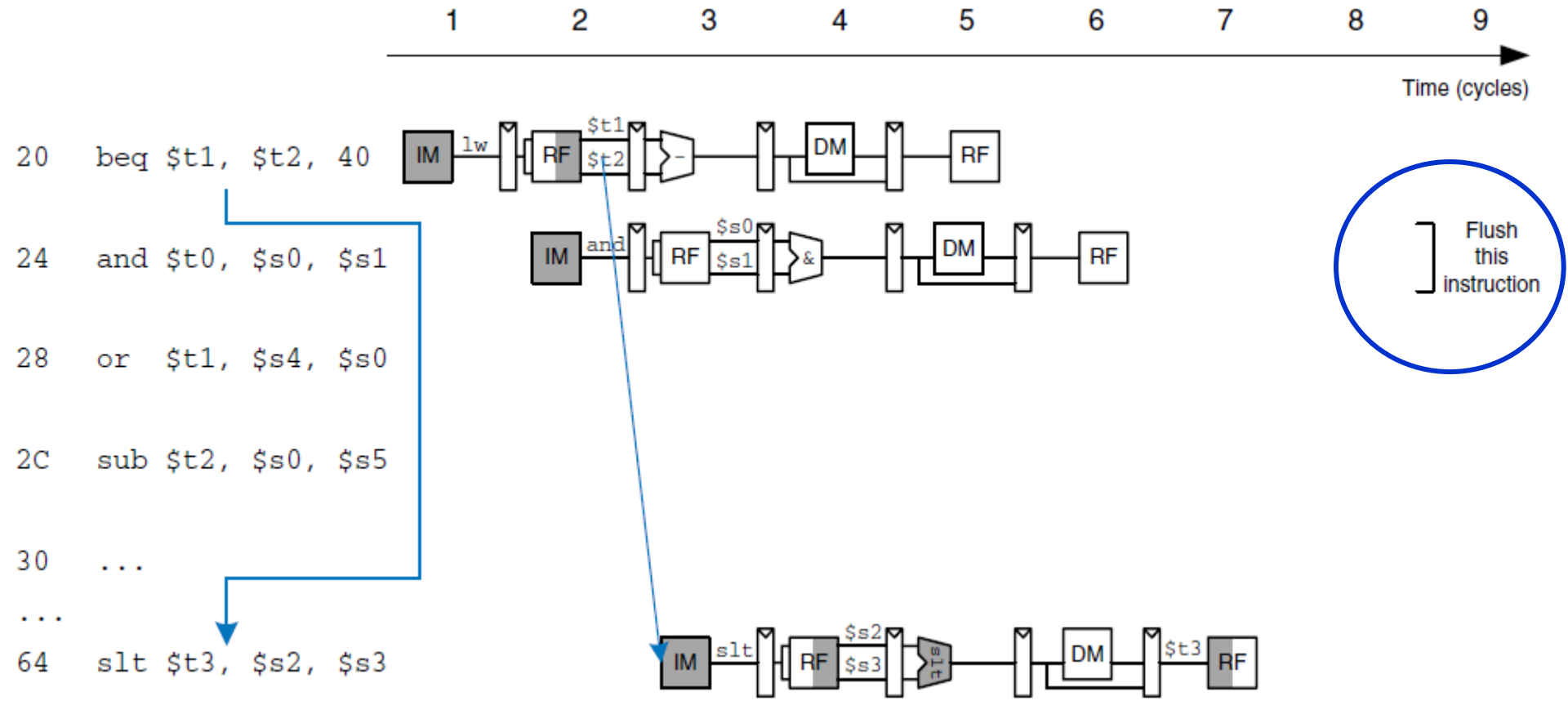


Control hazards (branch and jump)



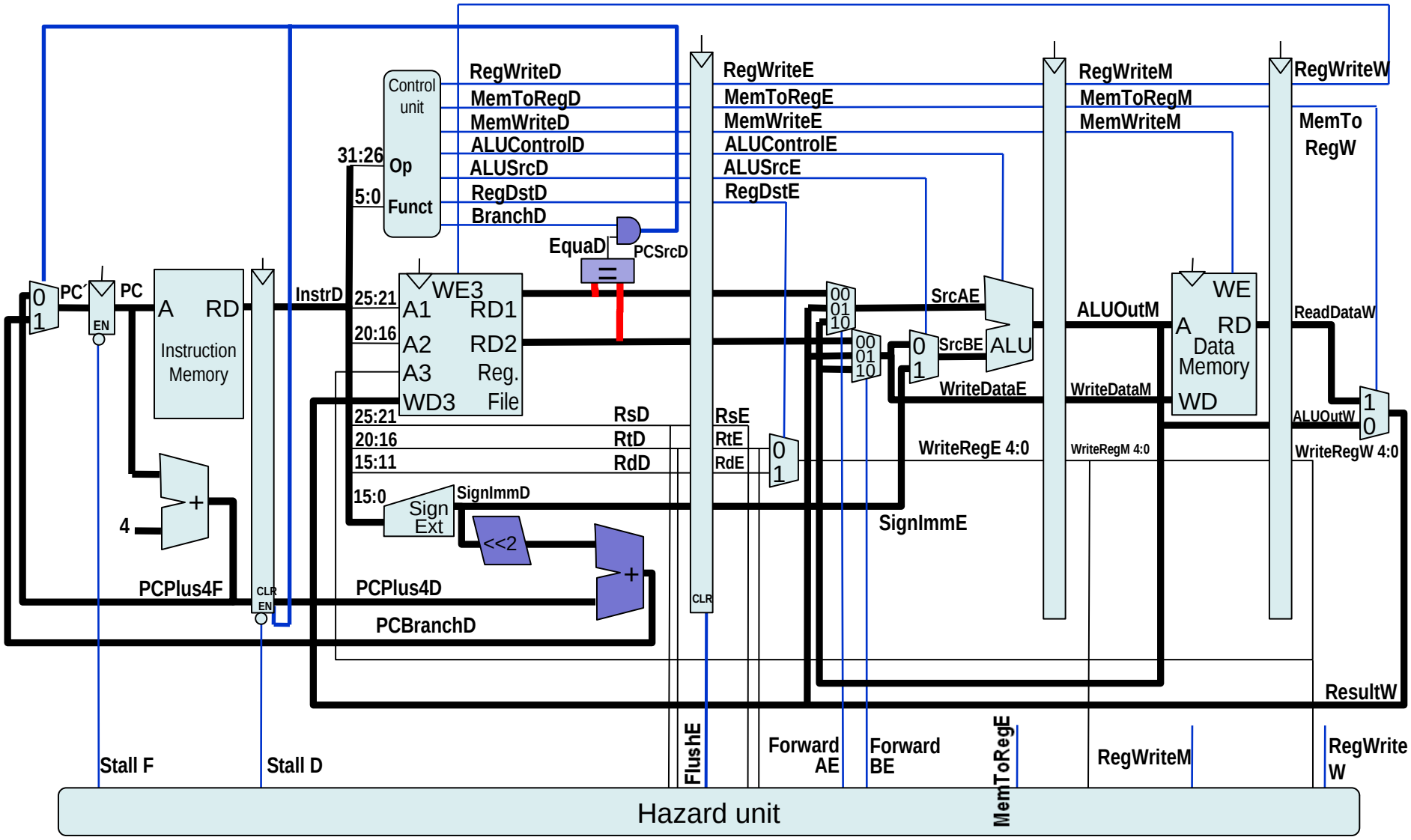
- Result is not known before 4th cycle. Why?

Control hazards – better to know result earlier…

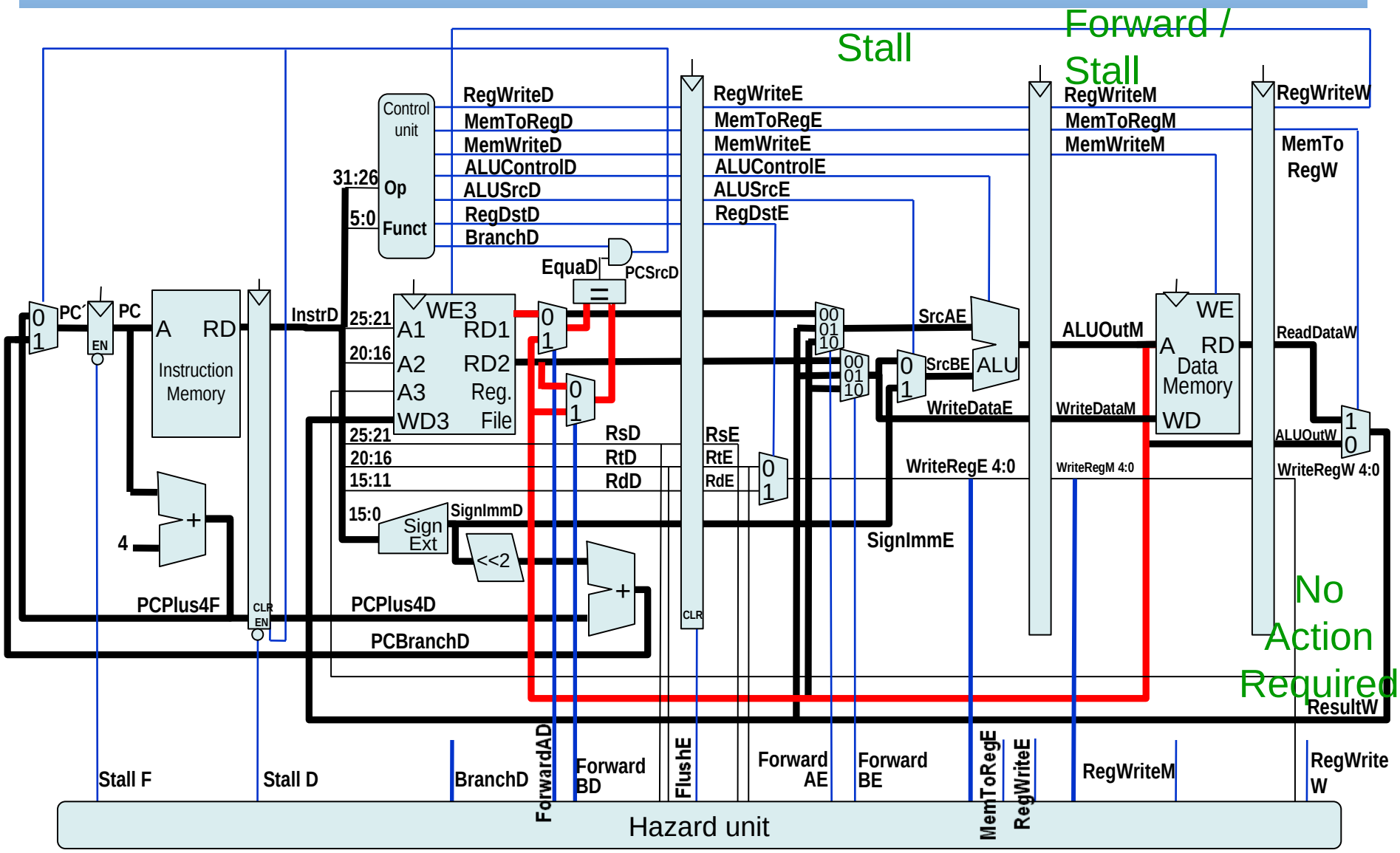


- If the result of comparison can be evaluated in the 2nd cycle **misprediction penalty** can be reduced
- But the processing of the comparison at earlier stage can induce new RAW hazards..!!!

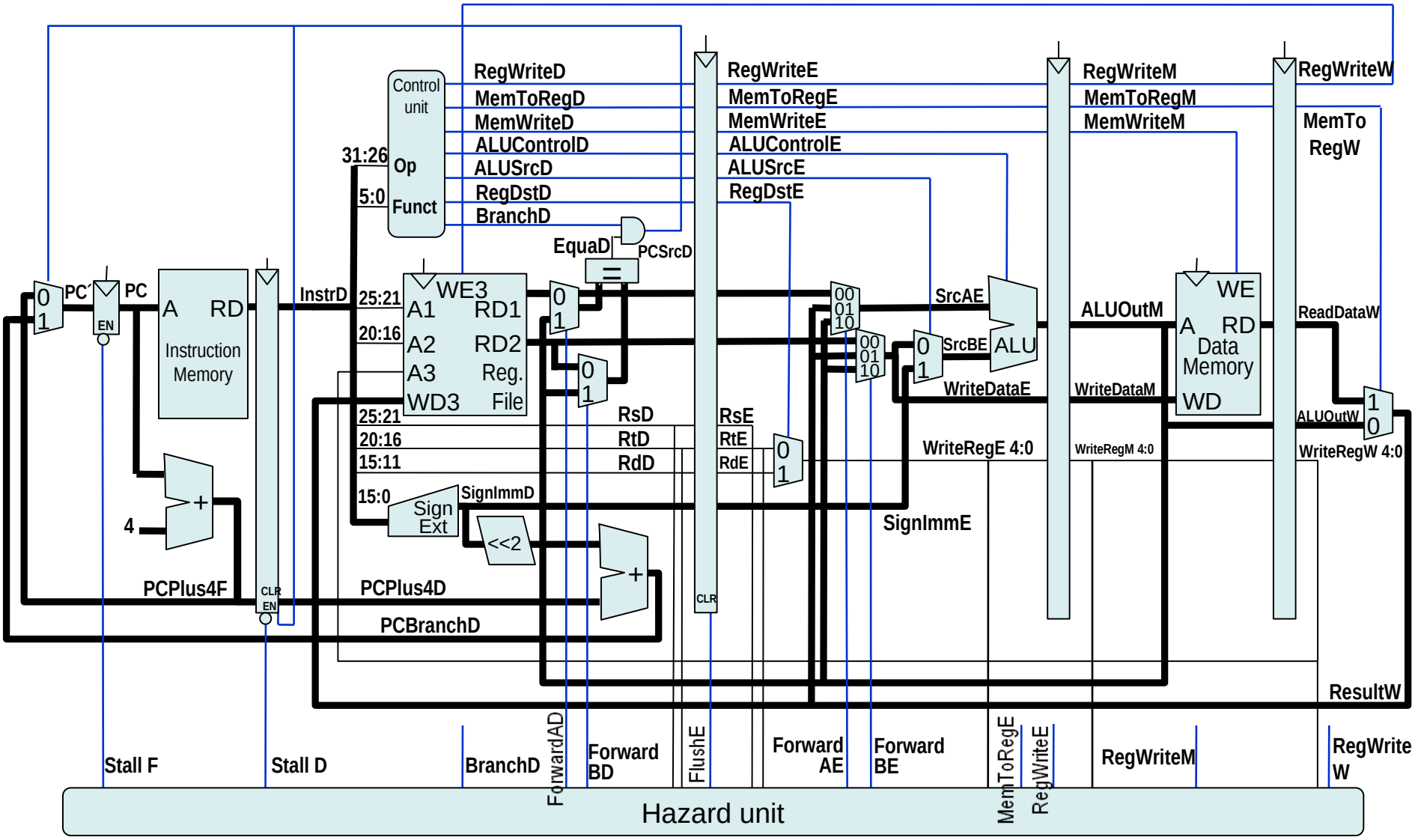
Resolve control hazards by early evaluate and flush



Resolve RAW hazards by forwarding or stalling



We are finished – pipelined processor is designed



Pipelined CPU – performance: IPS = IC / T = $IPC_{avg}.f_{CLK}$

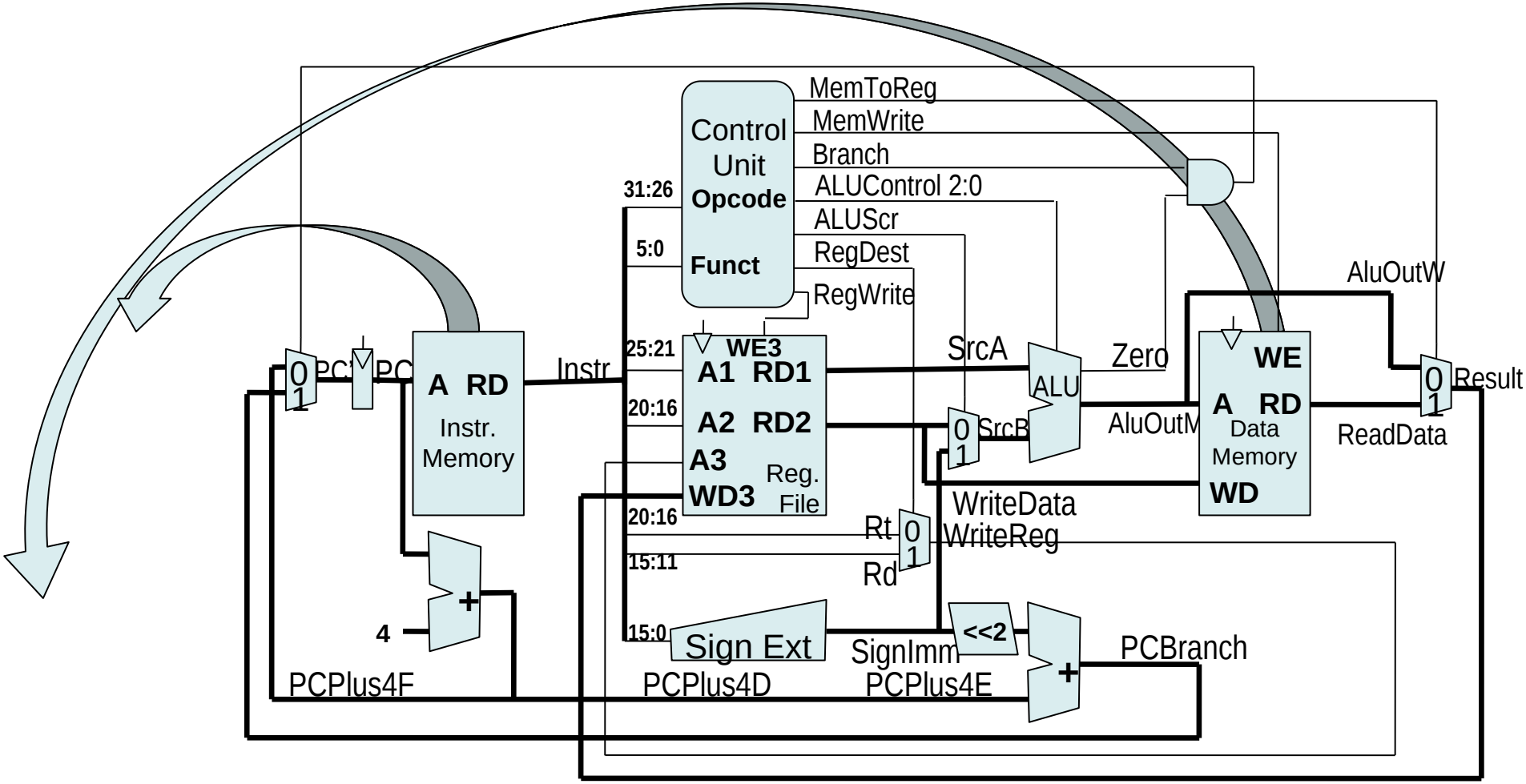
- What is maximal acceptable frequency for the CPU?
- Which stage is the slowest one?
- The cycle time is determined by the slowest stage
- For our case:
  Tc = 300 ns --> 3 333 kHz

  If the pipeline fill overhead is neglected (i.e. no pipeline stalls and flushes are considered) then ideal IPC = 1.
  IPS = 1 • 3 333e3 = 3 333 000 instructions per second

- Introduction of the 5-stage pipeline increases performance (throughput) 3 333 000/ 980 000 = 3.4 times! (considering IPC=1)

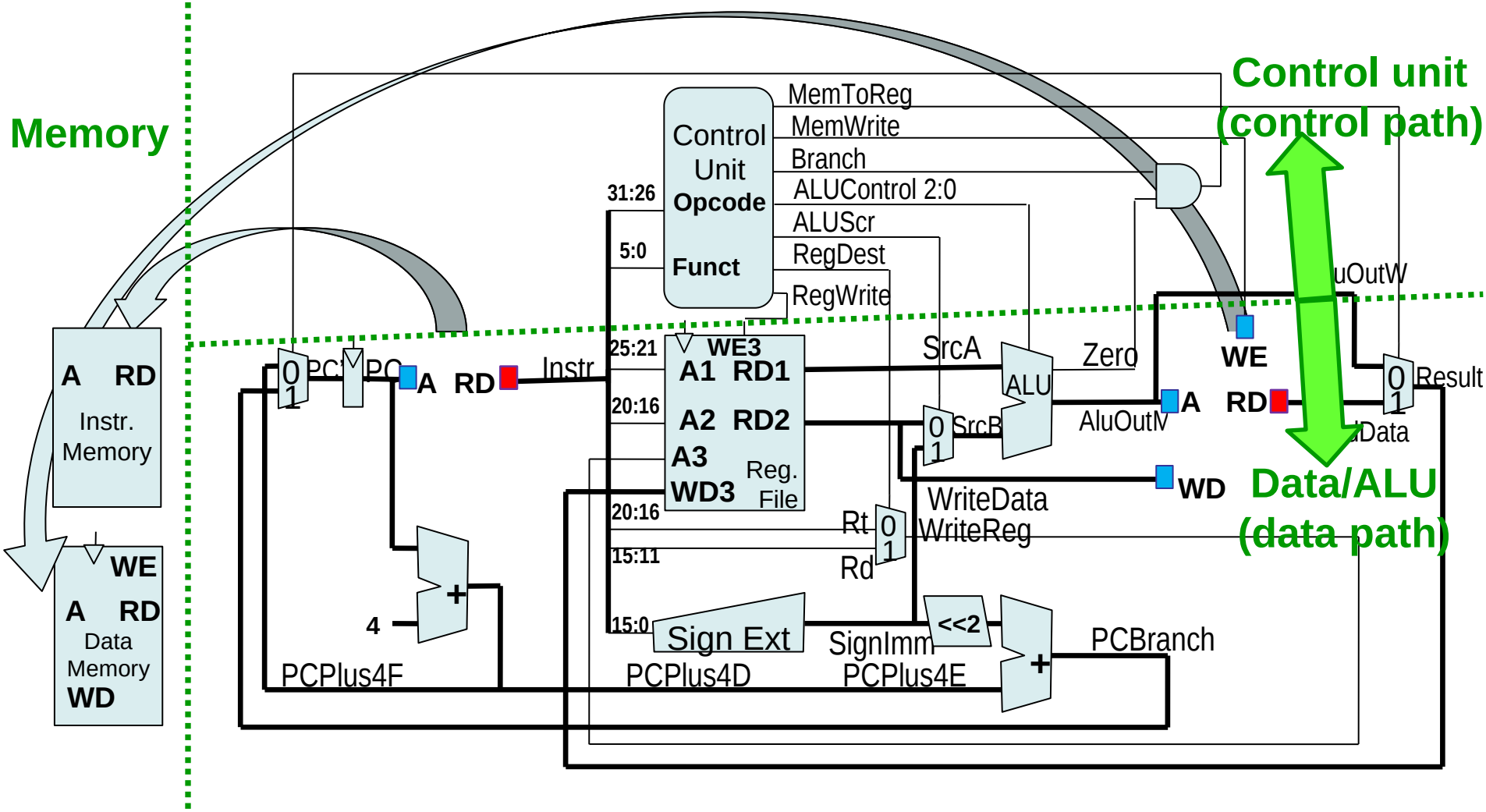
What is result of the design?

Return back to non-pipelined CPU version

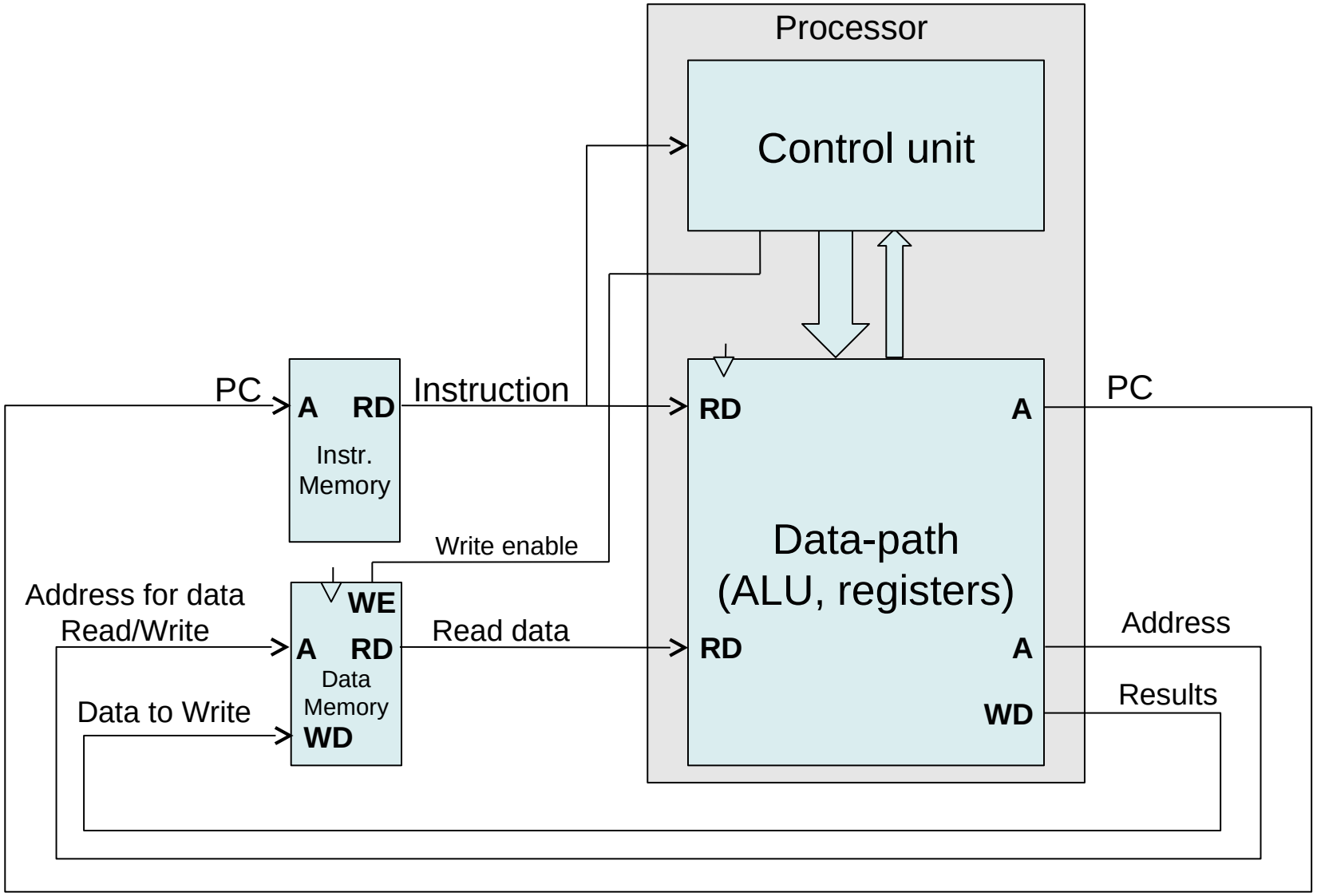


What is result of the design?

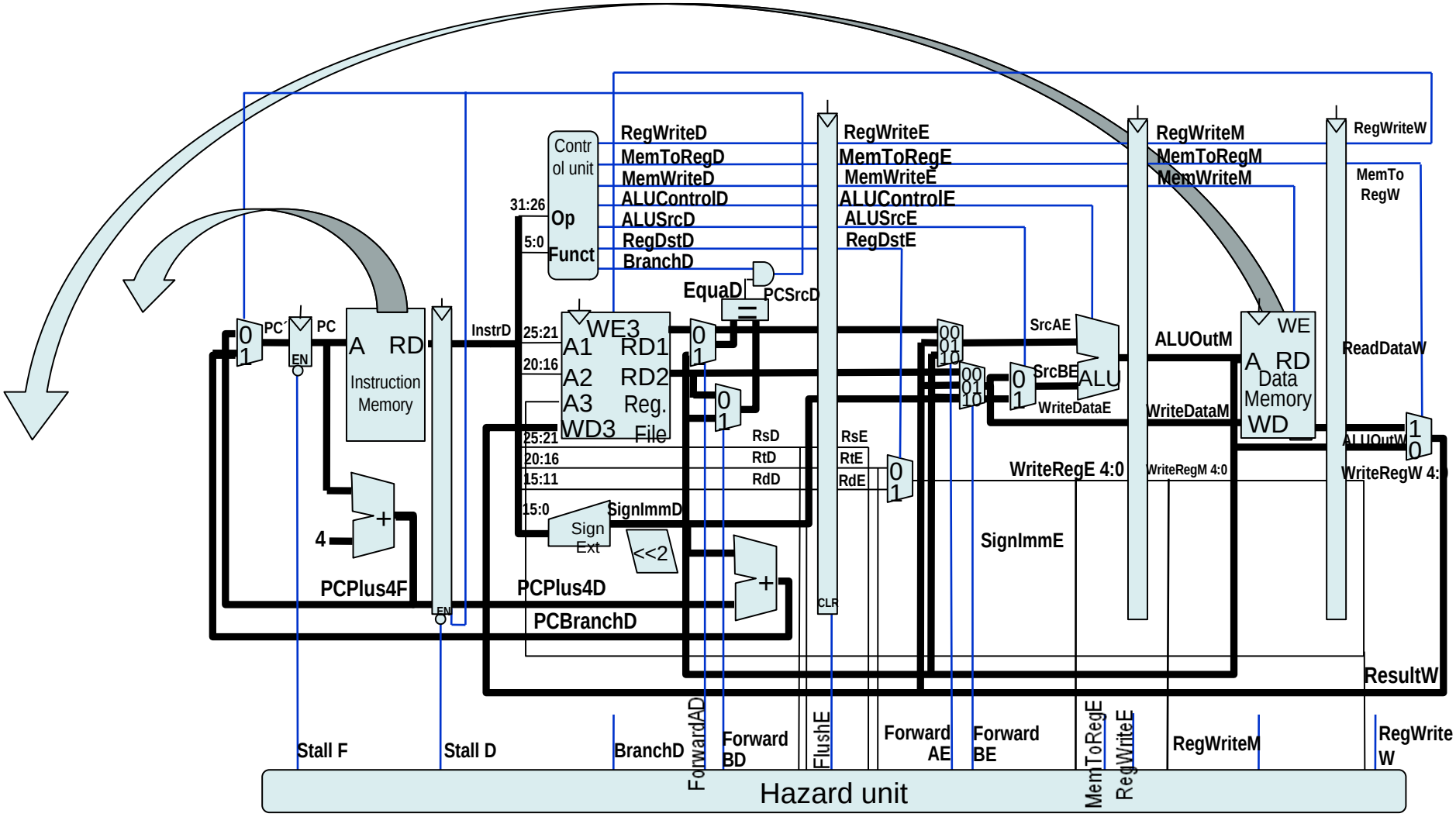
Return back to non-pipelined CPU version



What is result of the design?



CPU design result – pipelined version



Literature and resources

- Hennesy, J. L., Patterson, D. A.: Computer Organization and Design, The HW/SW Interface
- Hennesy, J. L., Patterson, D. A.: Computer Architecture : A Quantitative Approach, Third Edition, San Francisco, Morgan Kaufmann Publishers, Inc., 2002
- Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2004

Motivation and Mottos

- QtMips Home Page https://github.com/cvut/QtMips

  Implemented for Computer Architectures
  https://cw.fel.cvut.cz/wiki/courses/b35apo/start

  and Advanced Computer Architectures
  https://cw.fel.cvut.cz/wiki/courses/b4m35pap/start

  courses at Czech Technical University in Prague,
  Faculty of Electrical Engineering,
  Department of Control Engineering
- Come and meet with us, robotics, makers automotive etc. projects
- Come and teach with us, teaching is the best way to deeper understanding the subjects, no simulator can generate so much perturbations as students
- Talk is cheap. Show me the code. Linus Torvalds

  Reply https://www.openhub.net/accounts/ppisa
- Talk is cheap, show me your happiness. Michal Sojka

  Reply https://ppisa.rajce.idnes.cz/selected/