Computer Architectures

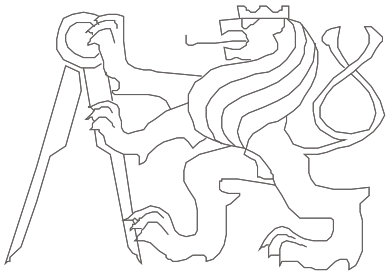
Pipelined Instruction Execution

Hazards, Stages Balancing, Super-scalar Systems

Pavel Píša, Richard Šusta

Michal Štepanovský, Miroslav Šnorek

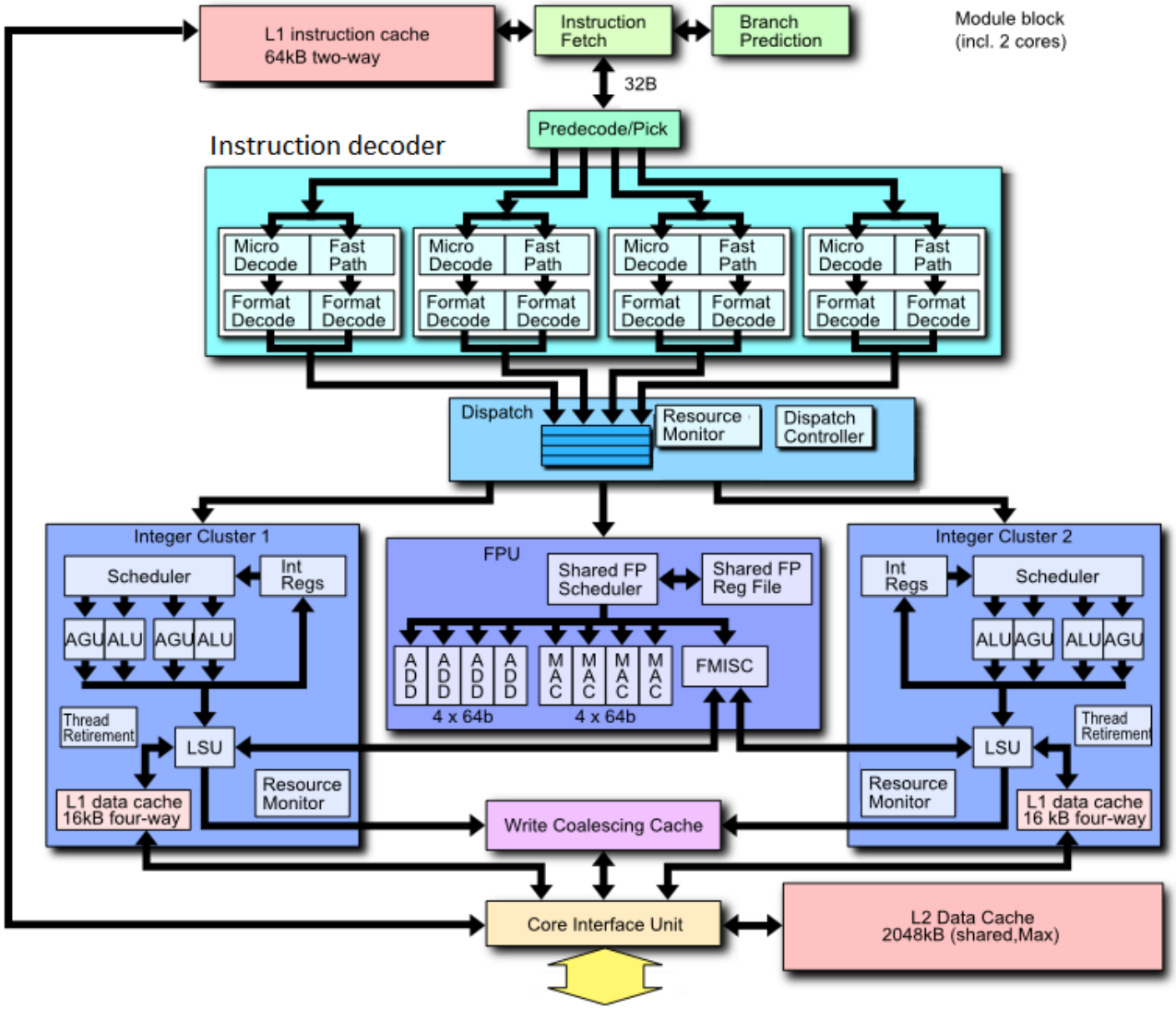
Main source of inspiration: Patterson and Hennessy



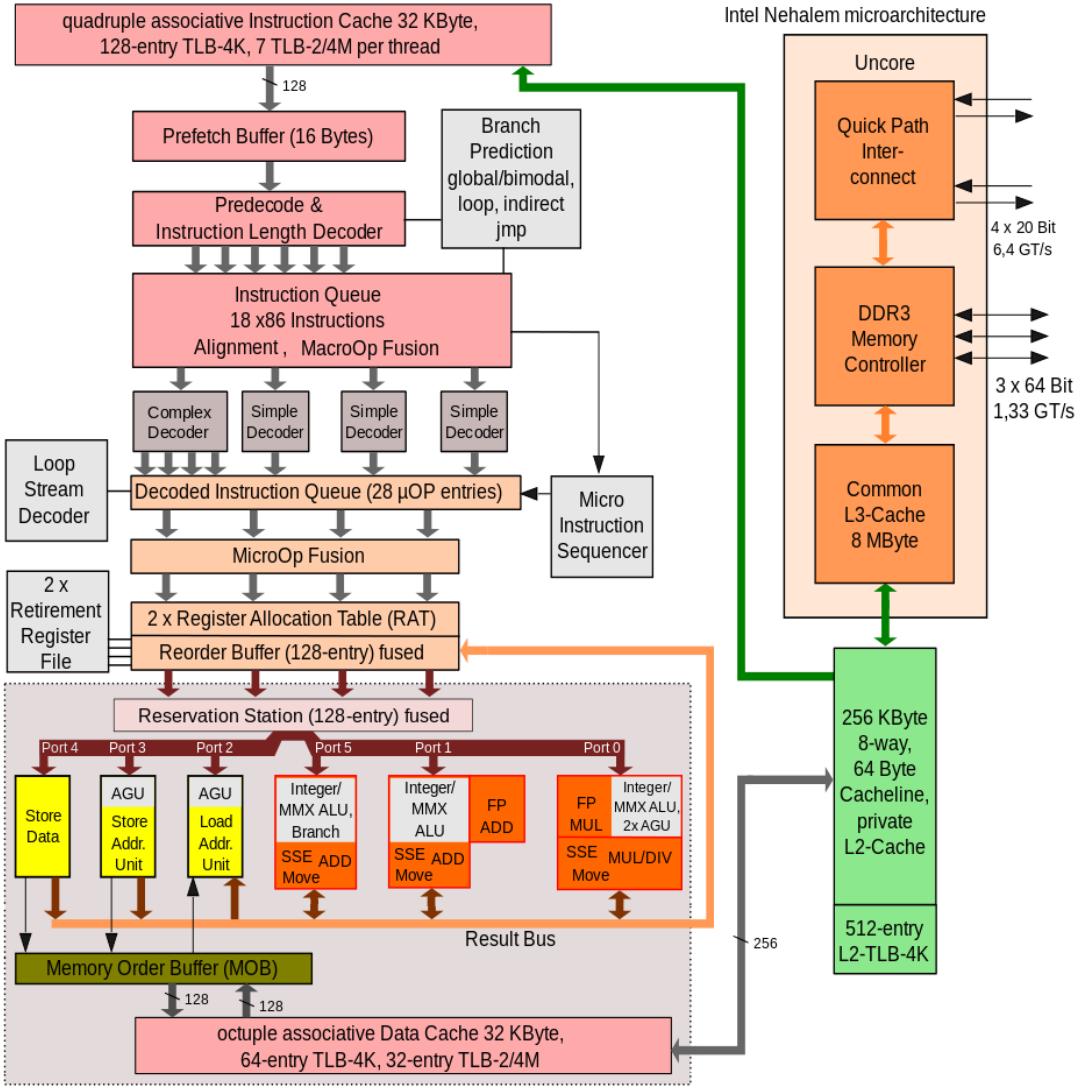
Czech Technical University in Prague, Faculty of Electrical Engineering

English version partially supported by:
European Social Fund Prague & EU: We invests in your future.

Motivation – AMD Bulldozer 15h (FX, Opteron) - 2011



Motivation – Intel Nehalem (Core i7) - 2008

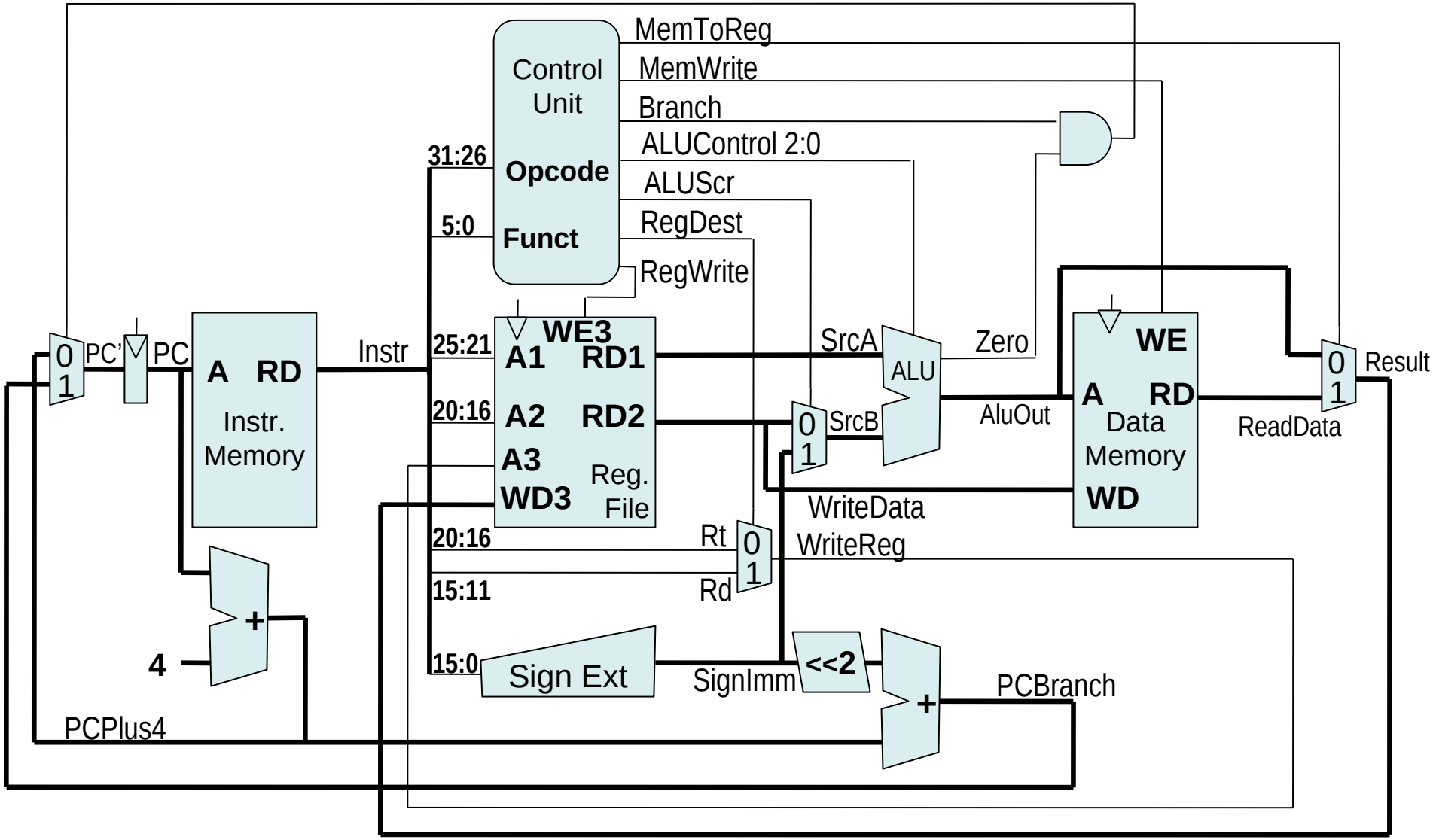


The Goal of Today Lecture

- Convert/extend CPU presented in the lecture 2 to the pipelined CPU design.
- The following instructions are considered for our CPU design:
  `add, sub, and, or, slt, addi, lw, sw` and `beq`

Typ	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				

Single Cycle CPU Together with Memories

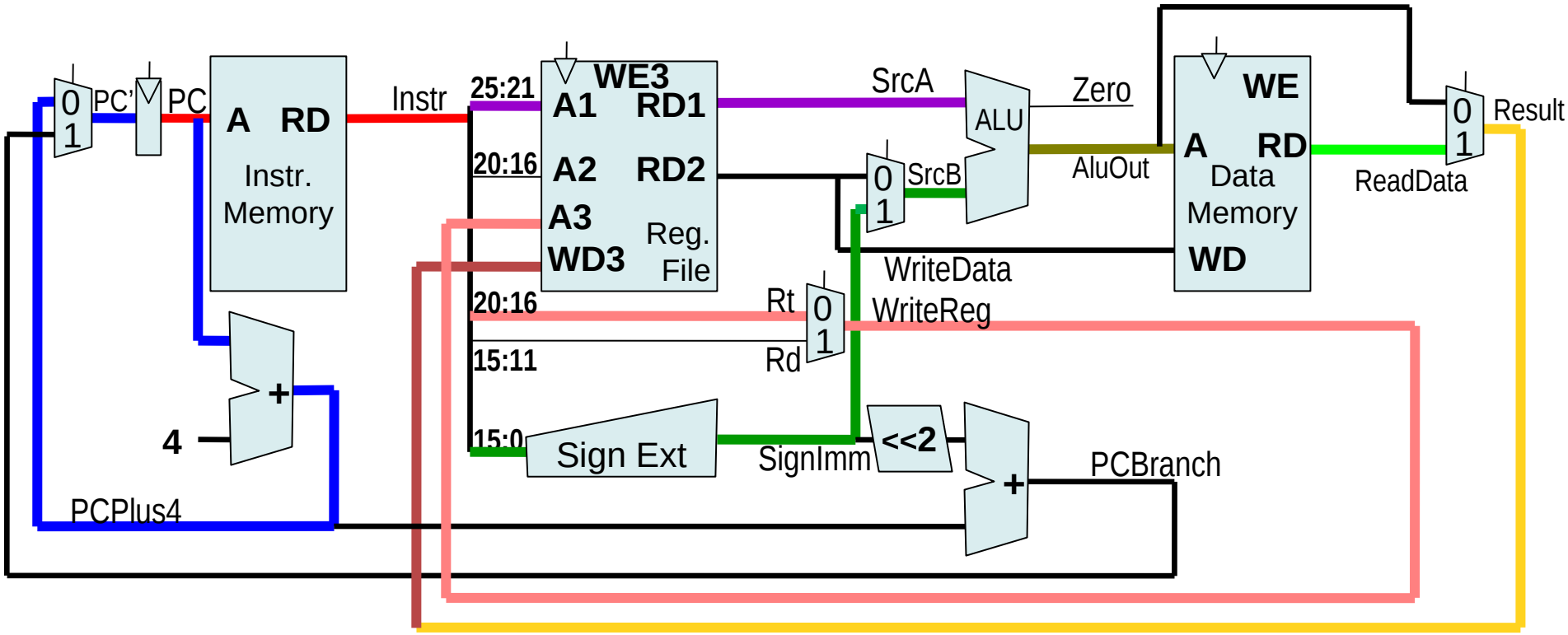


From lecture 2

Single Cycle CPU – Performance: IPS = IC / T = $IPC_{avg}.f_{CLK}$

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

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



From lecture 2

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

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

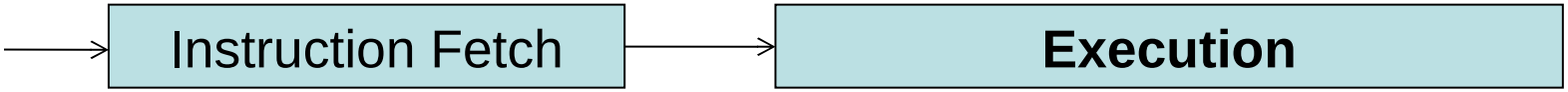
  Then Tc = 1020 ns --> $f_{CLK\ max}$ = 980 kHz,

  IPS = 1 • 980e3 = 980 000 instructions per second

From lecture 2

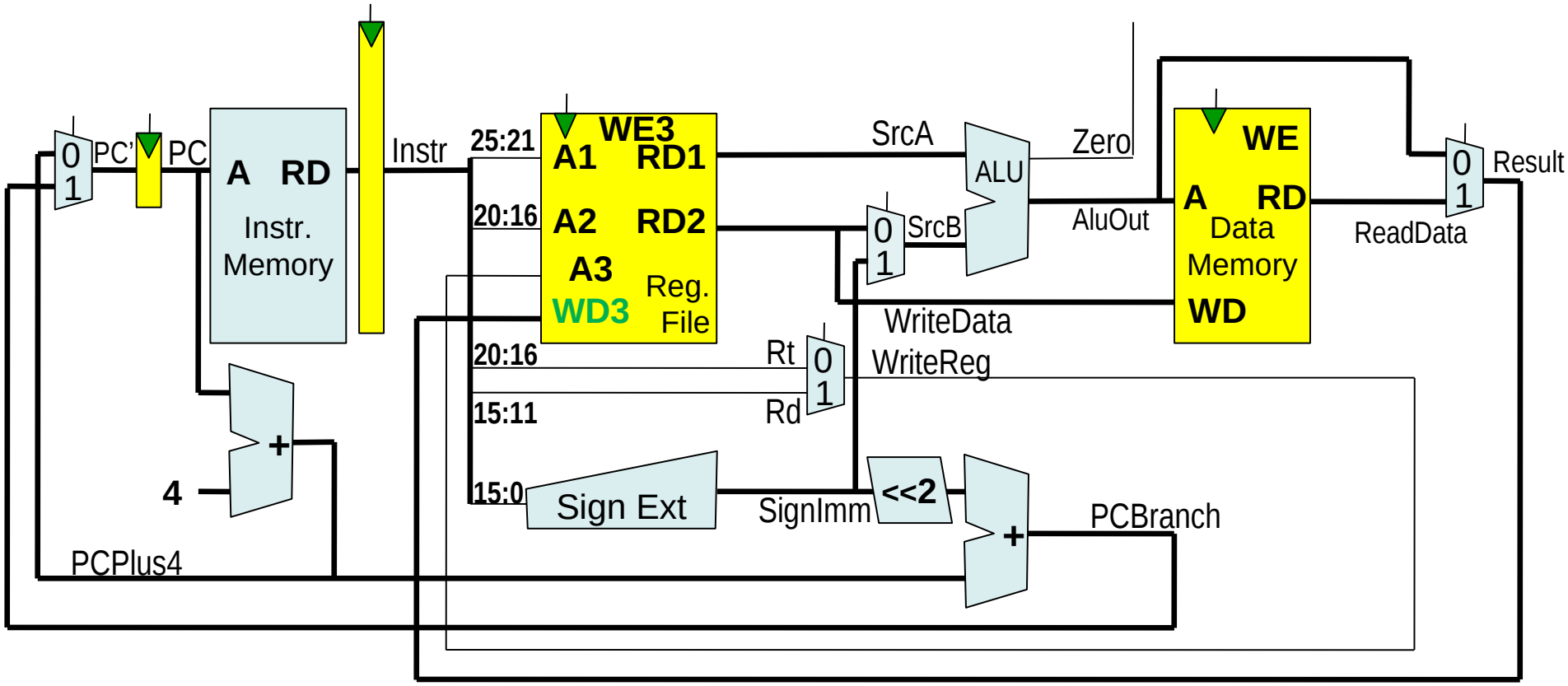
Separate Instruction Fetch and Execution

Even non-pipelined processors separate instruction execution into stages :



1. Instruction Fetch – setup PC for memory and fetch pointed instruction. Update PC = PC+4
2. The actual execution of the instruction

Non-Pielined Execution with Instructions Prefetching



in the figure, it indicates the clock input responding to the rising edge

Single Cycle CPU – Throughput: IPS = IC / T = $IPC_{str} \cdot f_{CLK}$

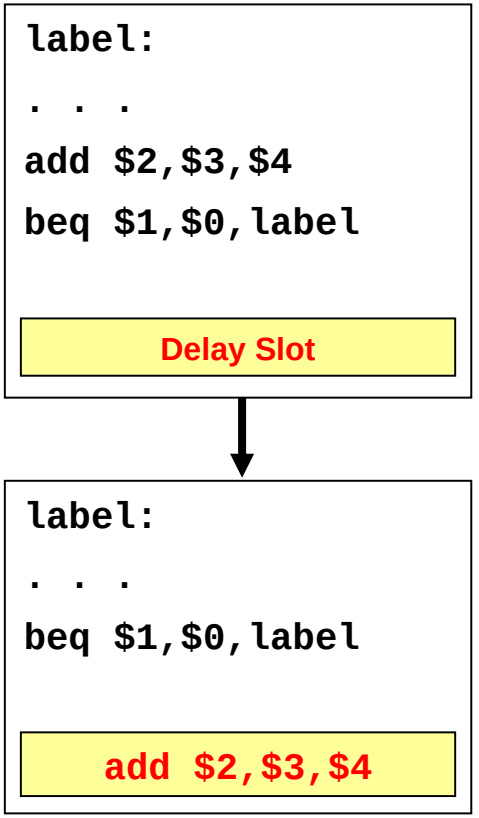
Consider following parameters:

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

If Tc_{fetch} is executed paralel with Tc_{proc},
then $Tc_{fetch} < Tc_{proc}$, and $Tc_p = 150+200+300+20+20$
= 690 ns → 1.45 MHz → **IPS = 1 450 000**

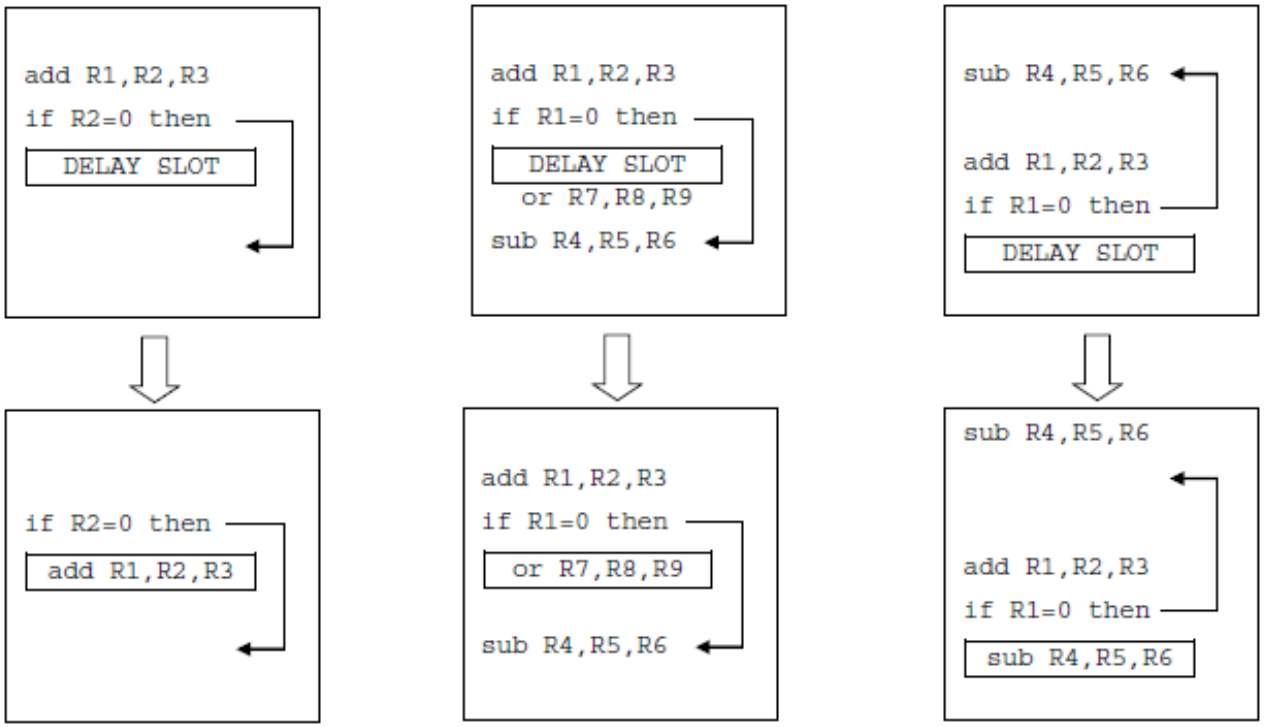
Delay Slot

- Prefetched instruction is executed unconditionally even when the proceeding instruction is a taken branch
  - Branch takes place after the next instruction
- MIPS architecture defines execution of one delay slot
- Compiler fills the branch delay slot
  - By selecting an independent instruction from the sequence before the branch
  - It has to be to execute instruction in the delay slot whether branch is taken or not
- If no suitable instruction is found
  - then the compiler fills delay slot with a NOP



Delay Slot

The task of the compiler is therefore to ensure that the following instructions in the branch delay slot are valid and useful. Three alternatives illustrating how the delay slot can be filled are depicted in the figures below.



The easiest way (at the same time the least efficient) is to fill the delay slot with a blank instruction - nop.

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

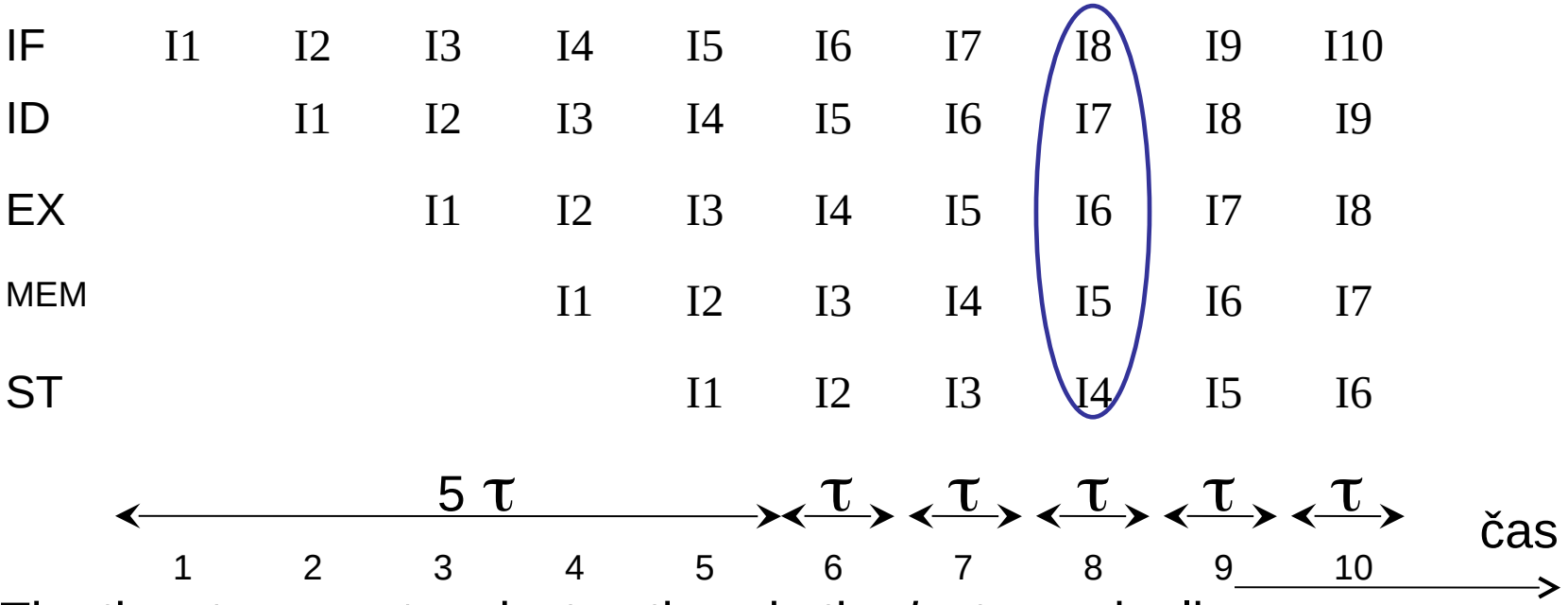
ID – decode the opcode and read registers specified by instruction, check for equality (for possible `beq` instruction), sign extend offset, compute branch target address for branch case (this is means to extend offset and add PC)

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



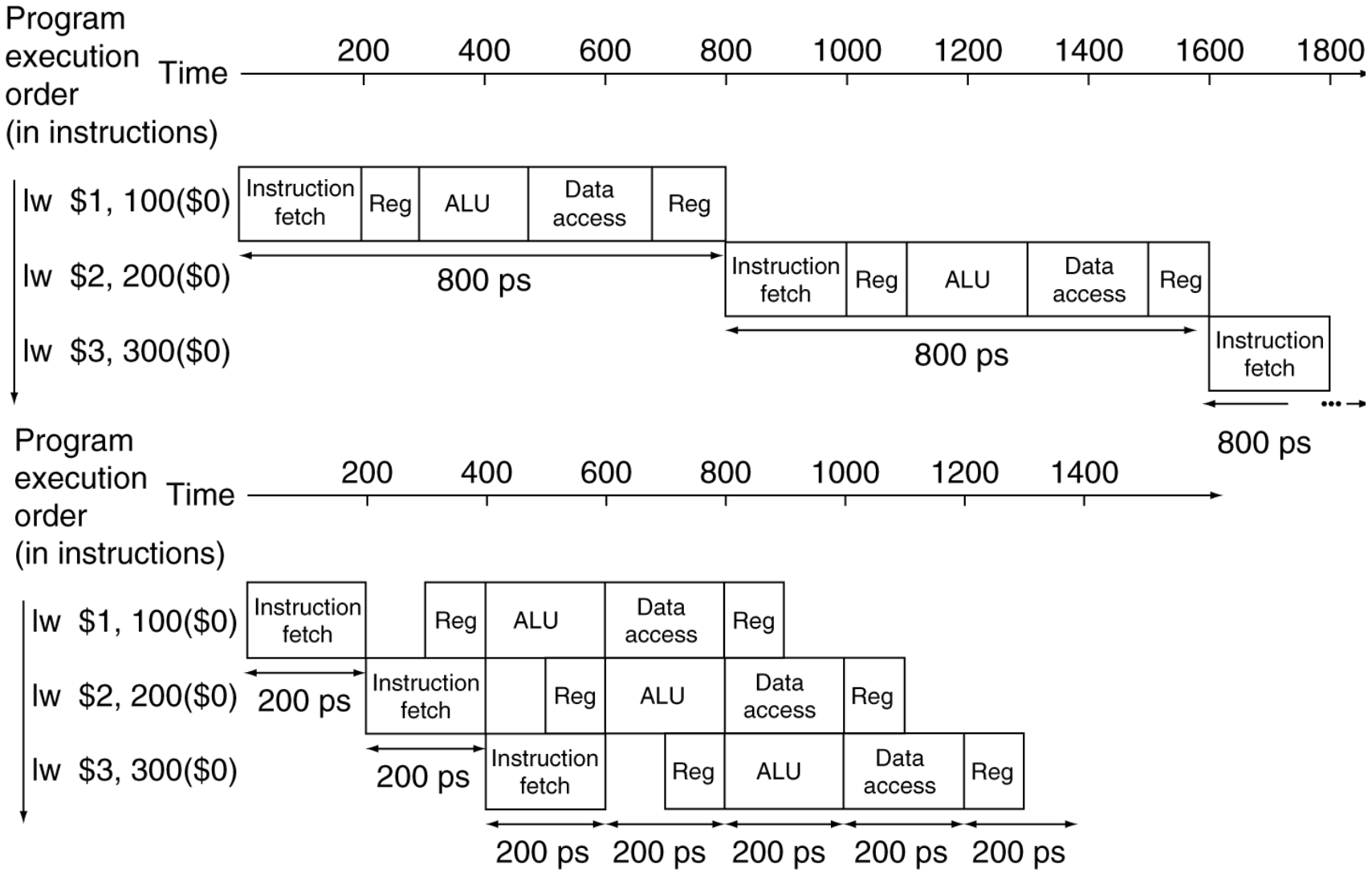
- The time to execute *n* instructions in the *k*-stage pipeline:

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

- 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

Pipeline Loads Example



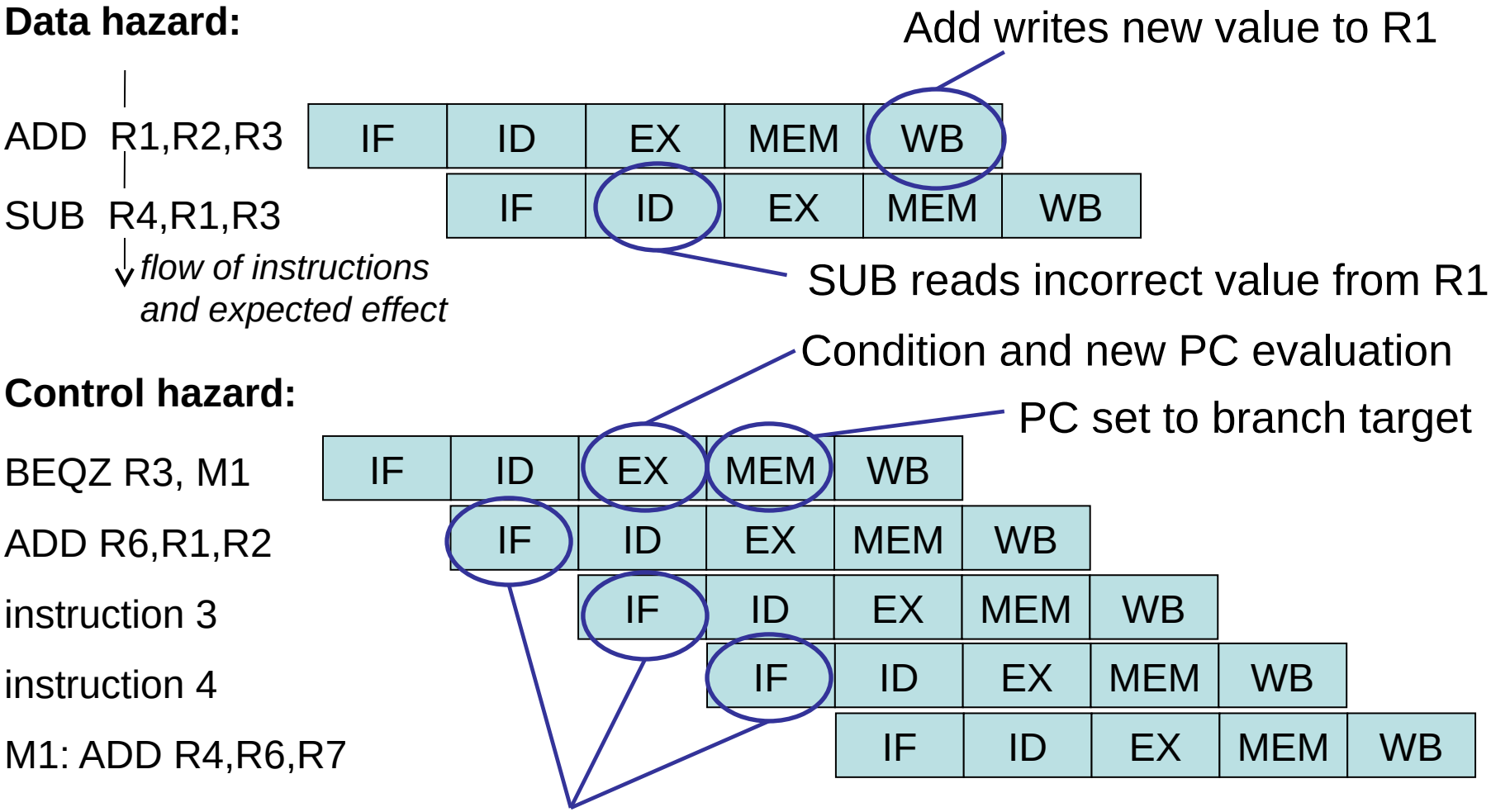
source: Gedare Bloom and Mary Jane Irwin

Instruction-level Parallelism - Pipelining

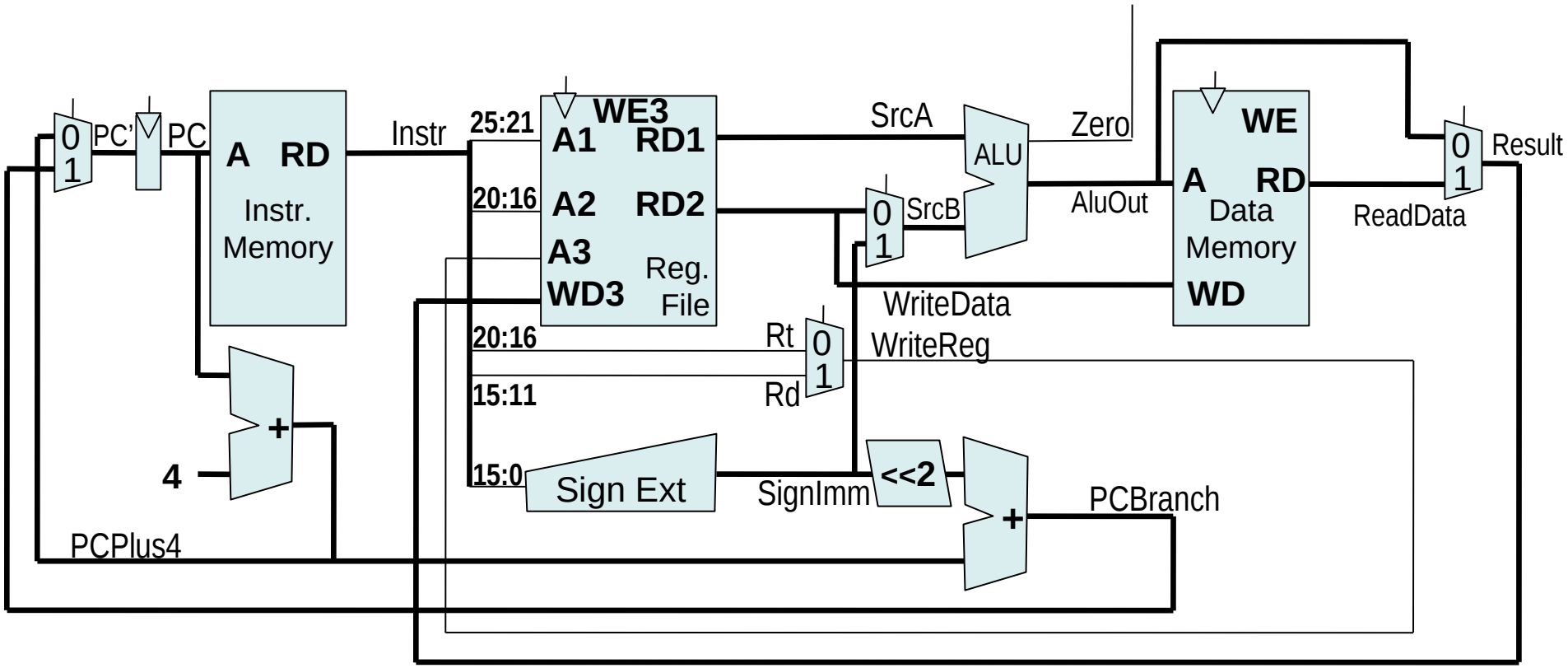
- Does not reduce the execution time of individual instructions, effect is just the opposite...
- Hazards:
  - structural (resolved by duplication),
  - data (result of data dependencies: RAW, WAR, WAW)
  - control (caused by instructions which change PC)...
- Hazard prevention can result in pipeline 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:

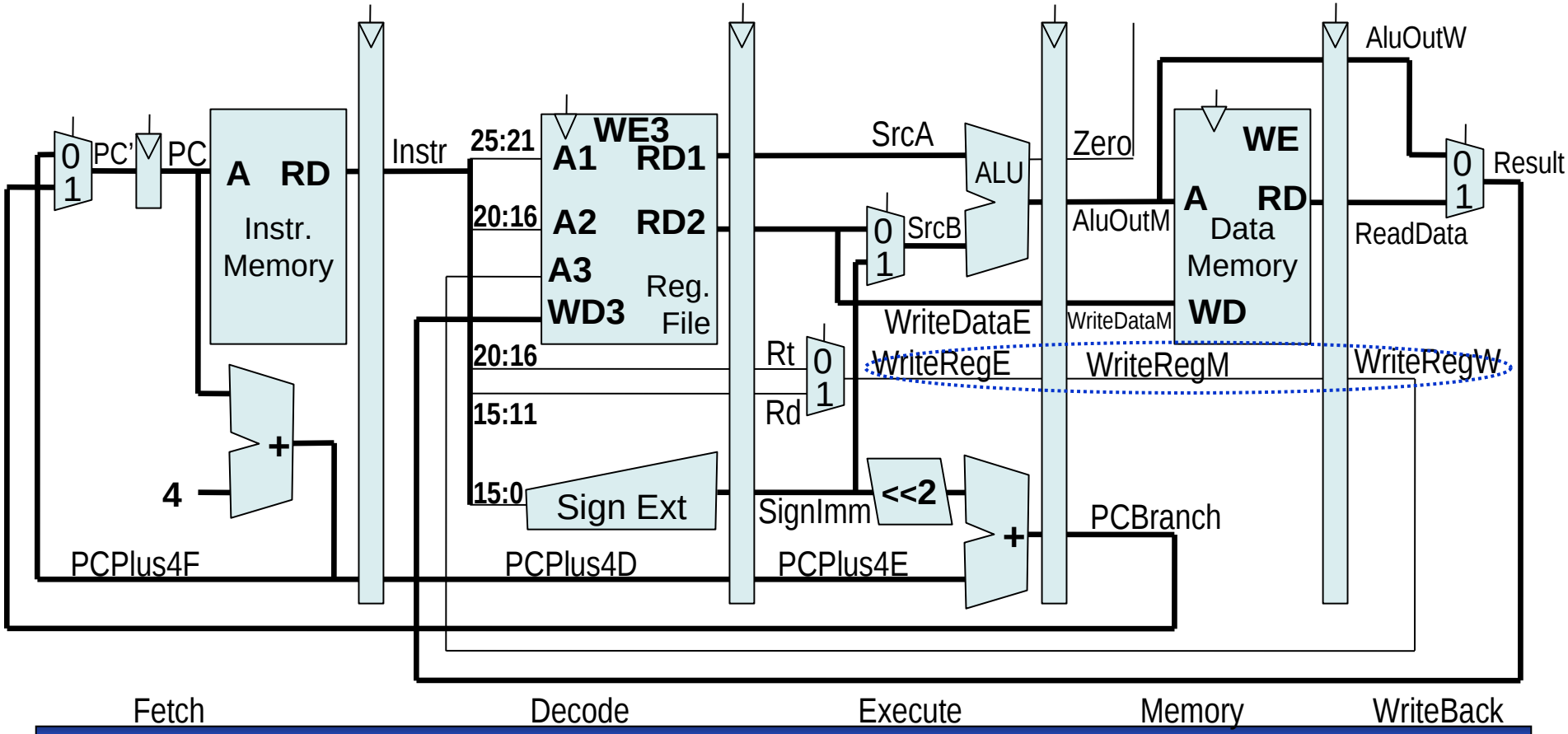


Non-pipelined Execution

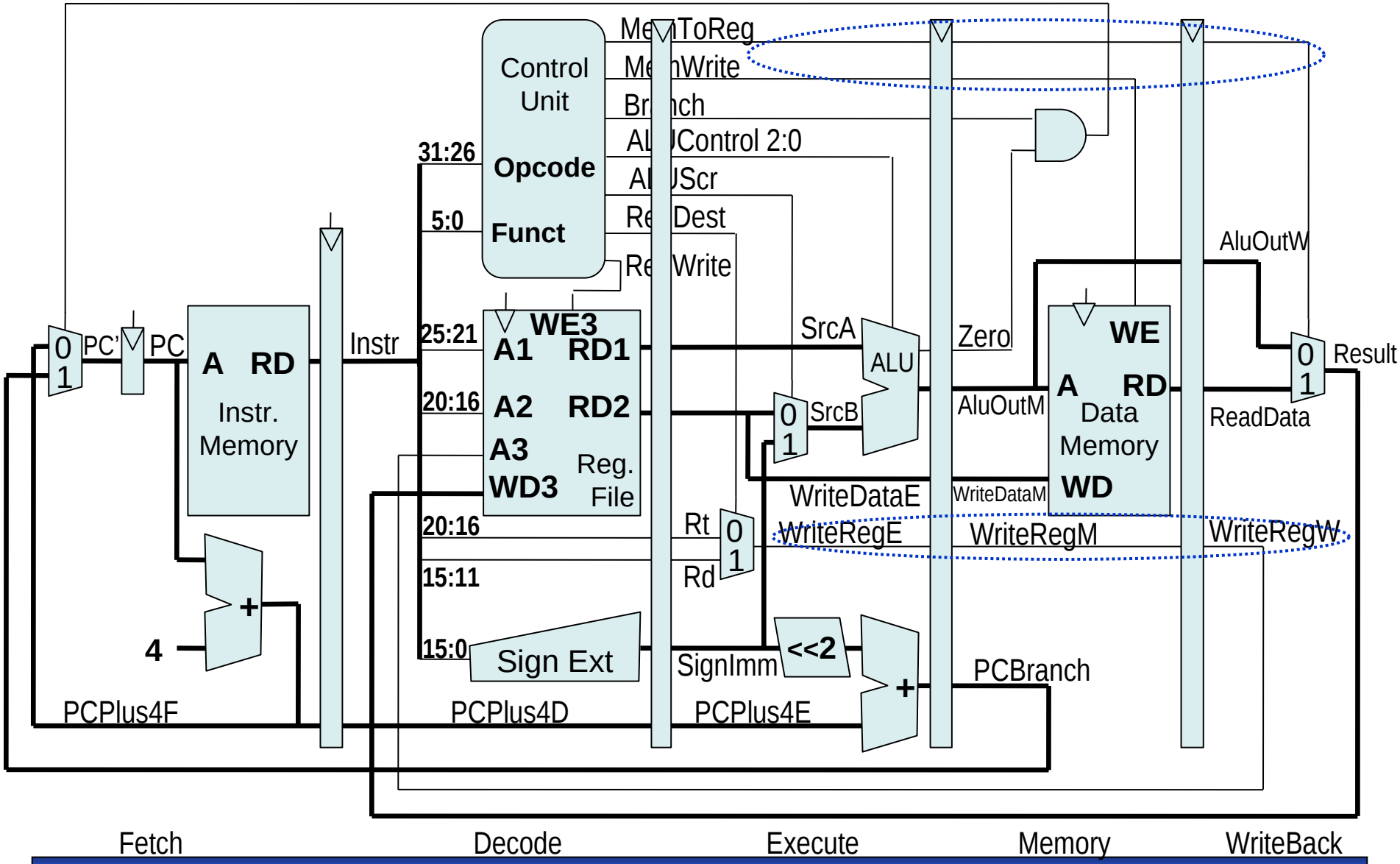


From lecture 2

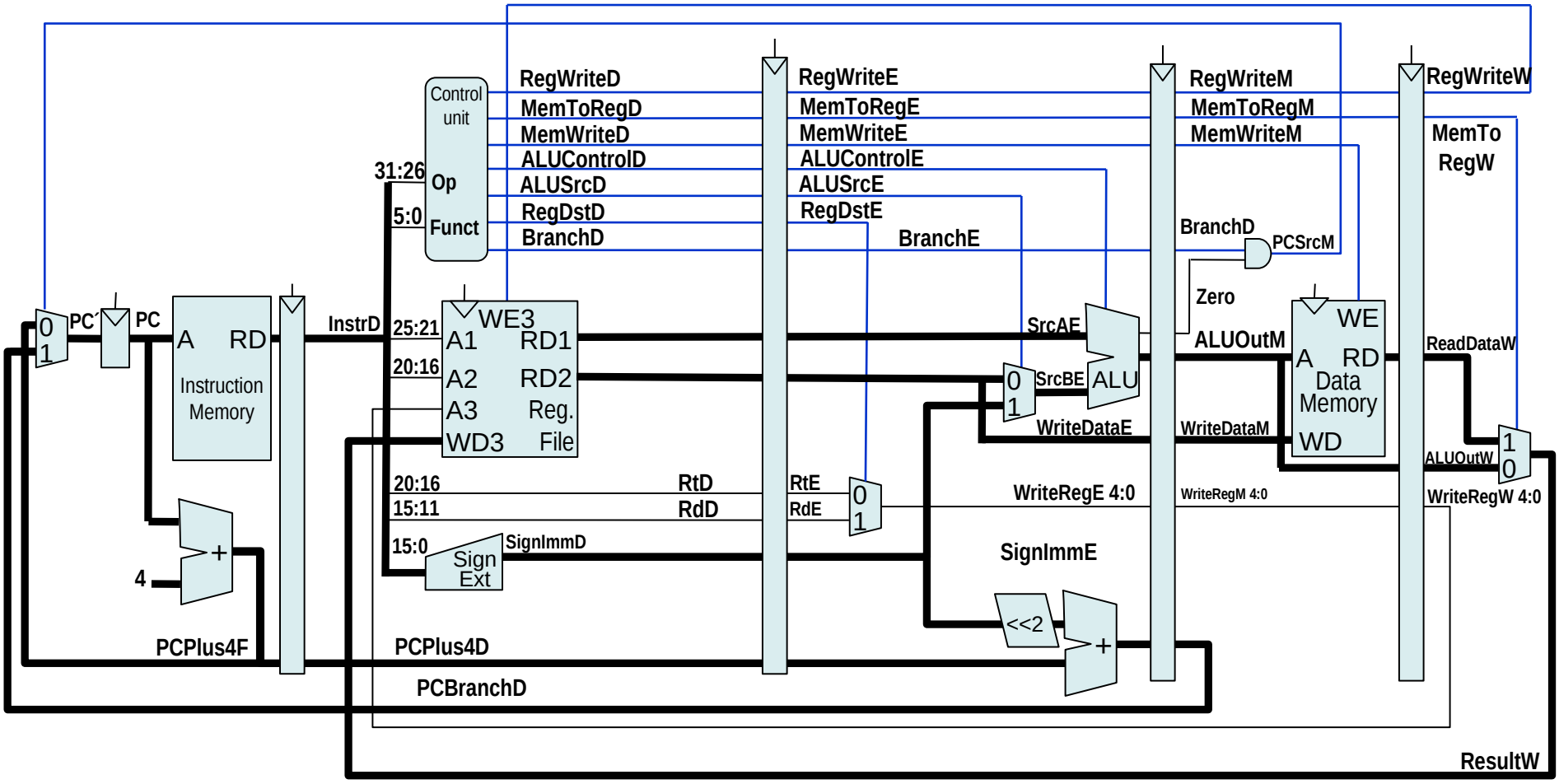
Pipelined Execution



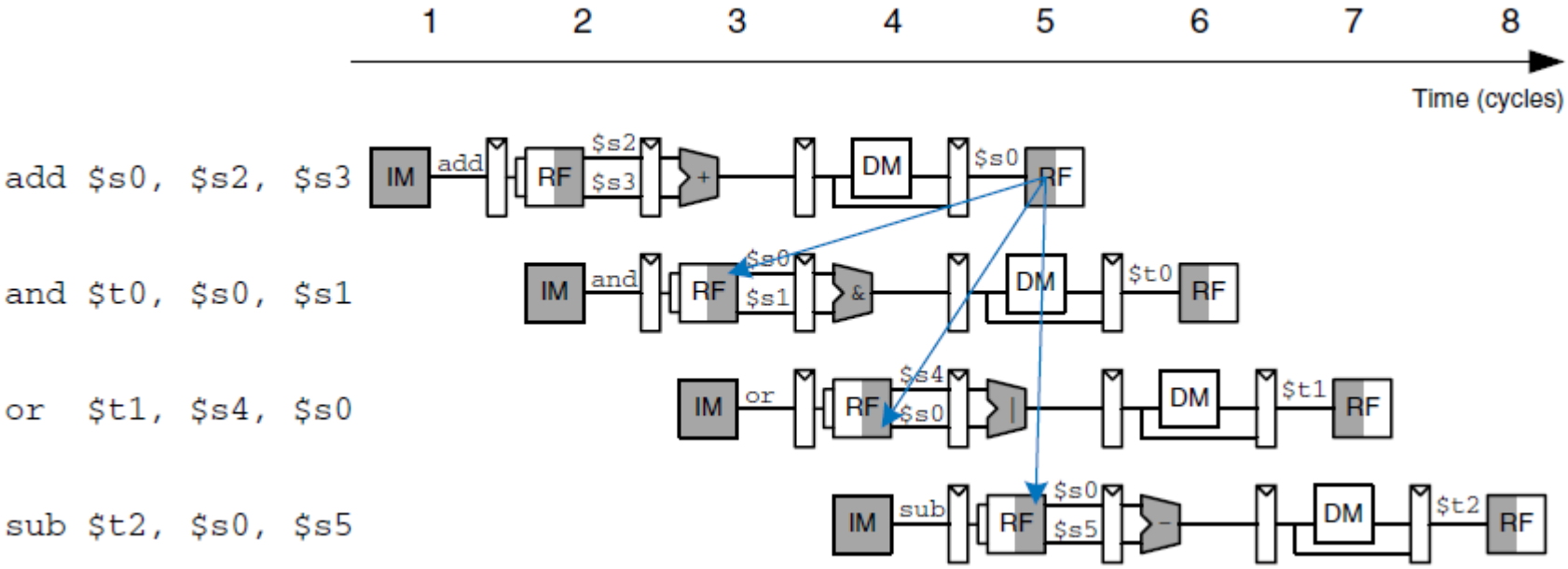
Pipelined Execution



The Same Design but Drawn Scaled Down…

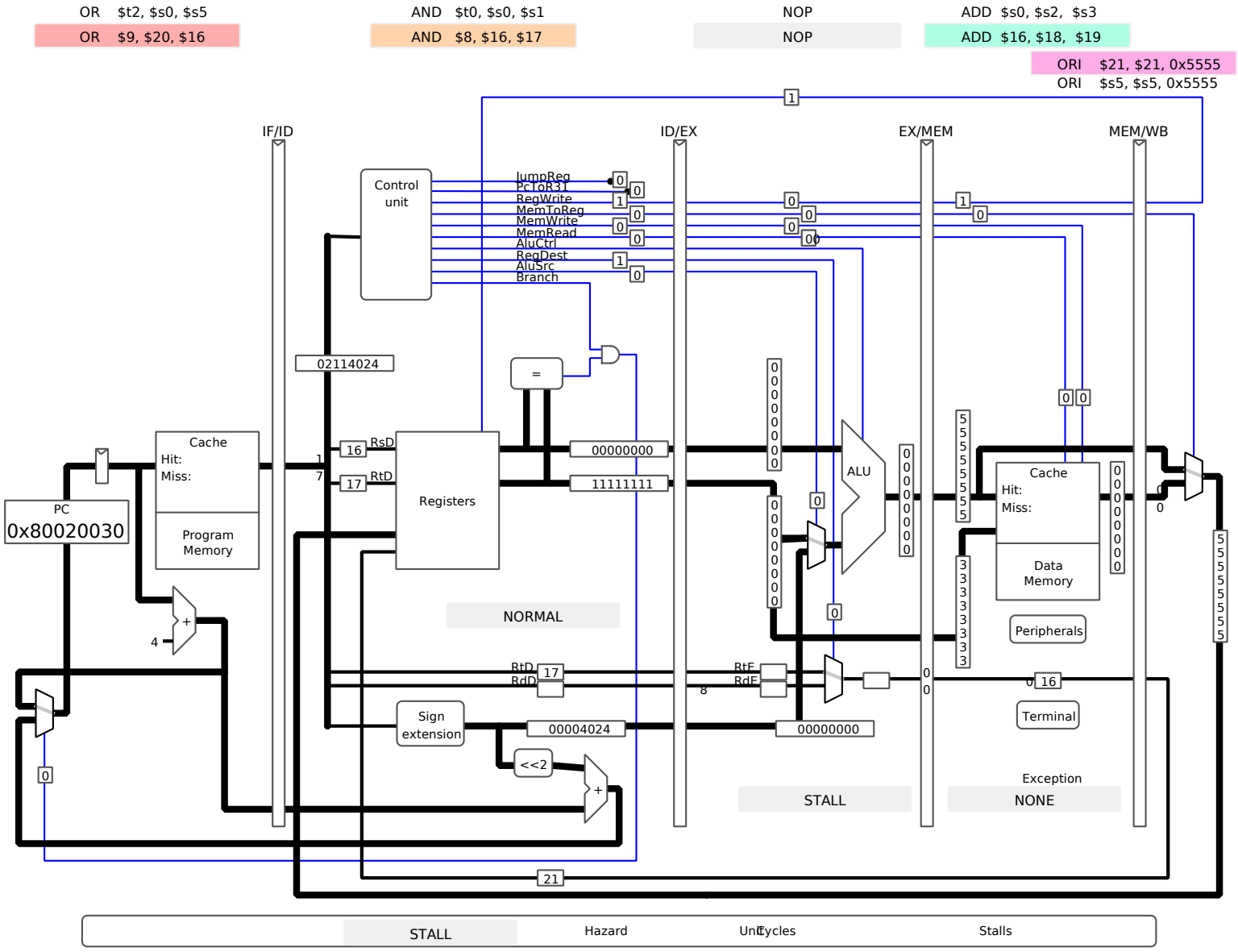


Cause of the Data Hazards

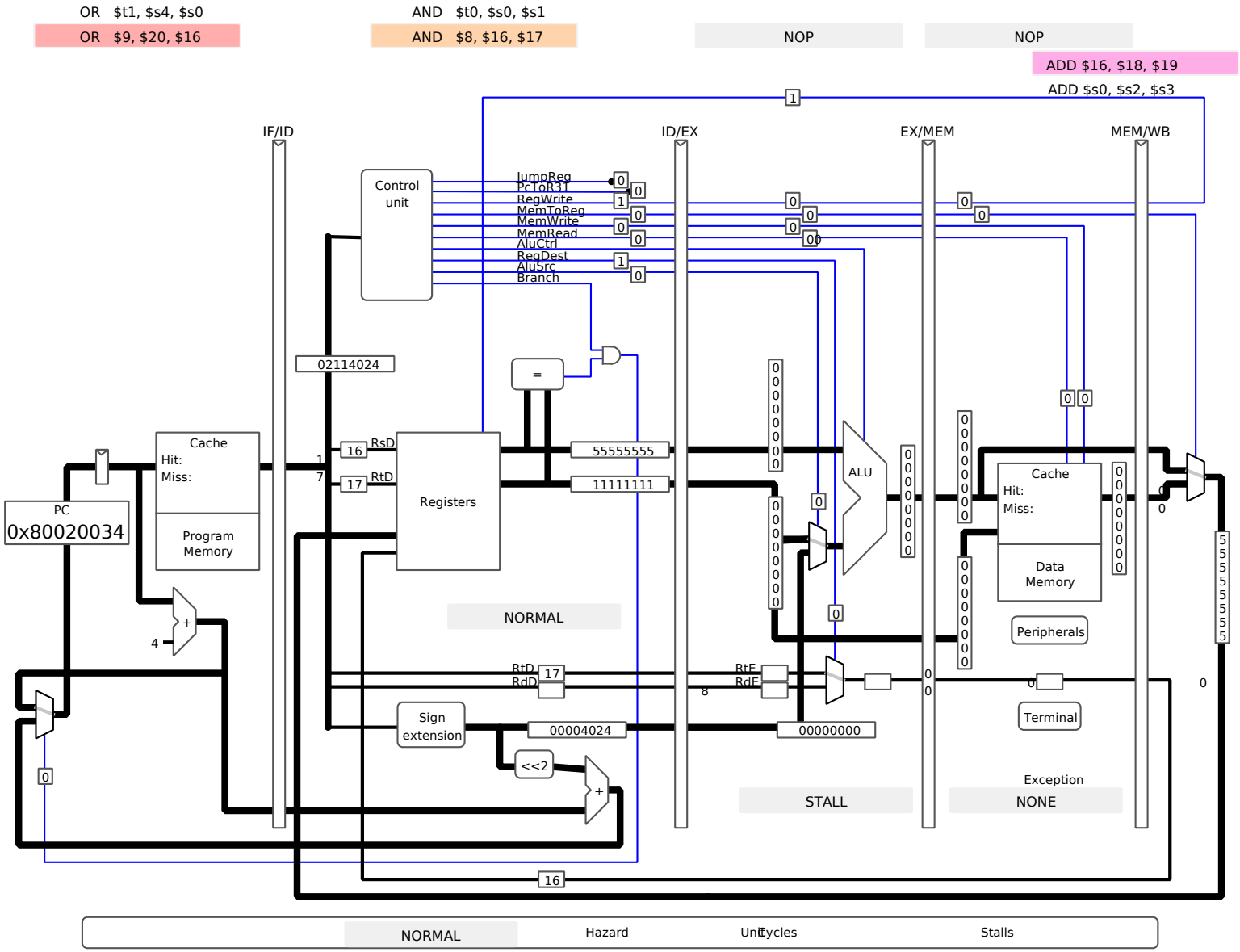


- Register File – access from two pipeline stages (Decode, WriteBack) – actual write occurs at the first half of the clock cycle, the read in the second half ⇒ there is no hazard for `sub $s0` input operand
- RAW (Read After Write) hazard – `and` (`or`) requires `$s0` in 3 (4)
- How can such hazard be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

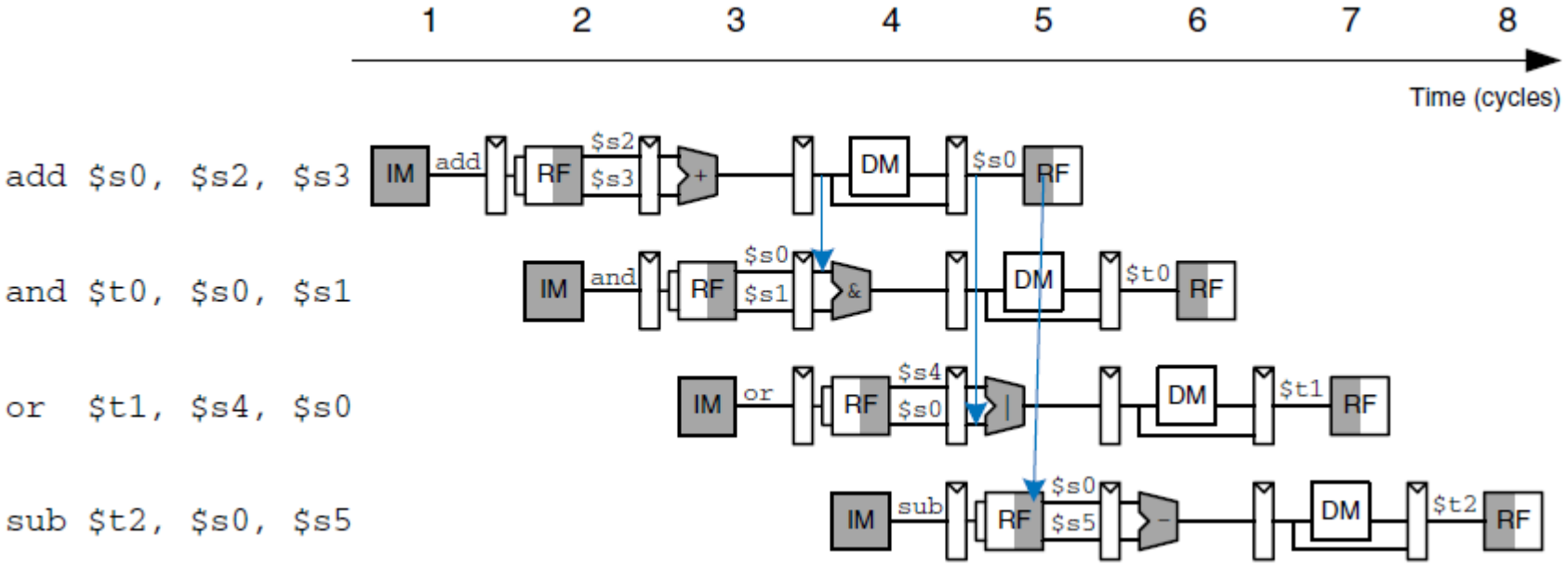
QtMips – Resolve Data Hazard by Stall



QtMips – Resolve Data Hazard by Stall

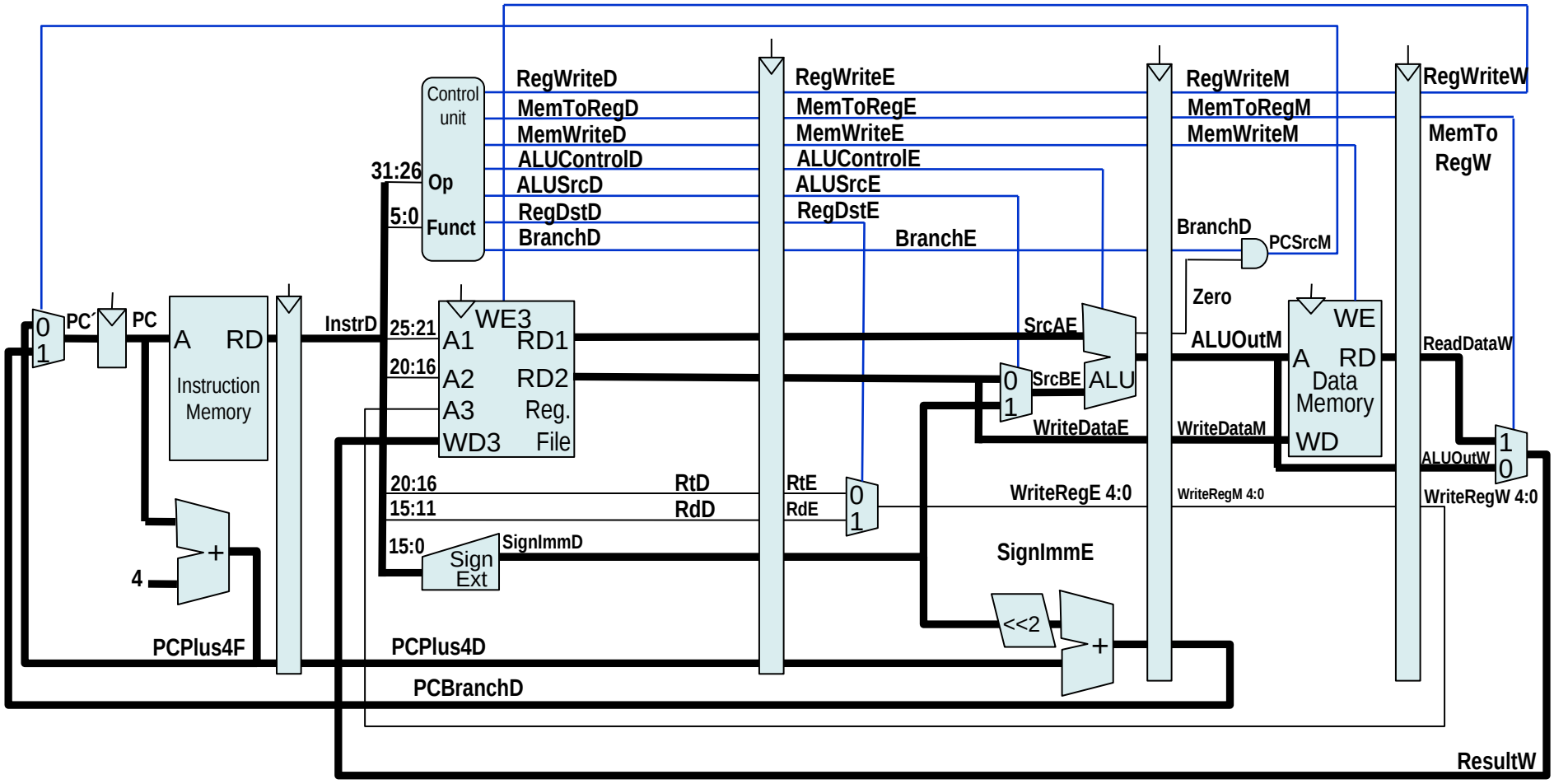


Forwarding to Avoid Data Hazards

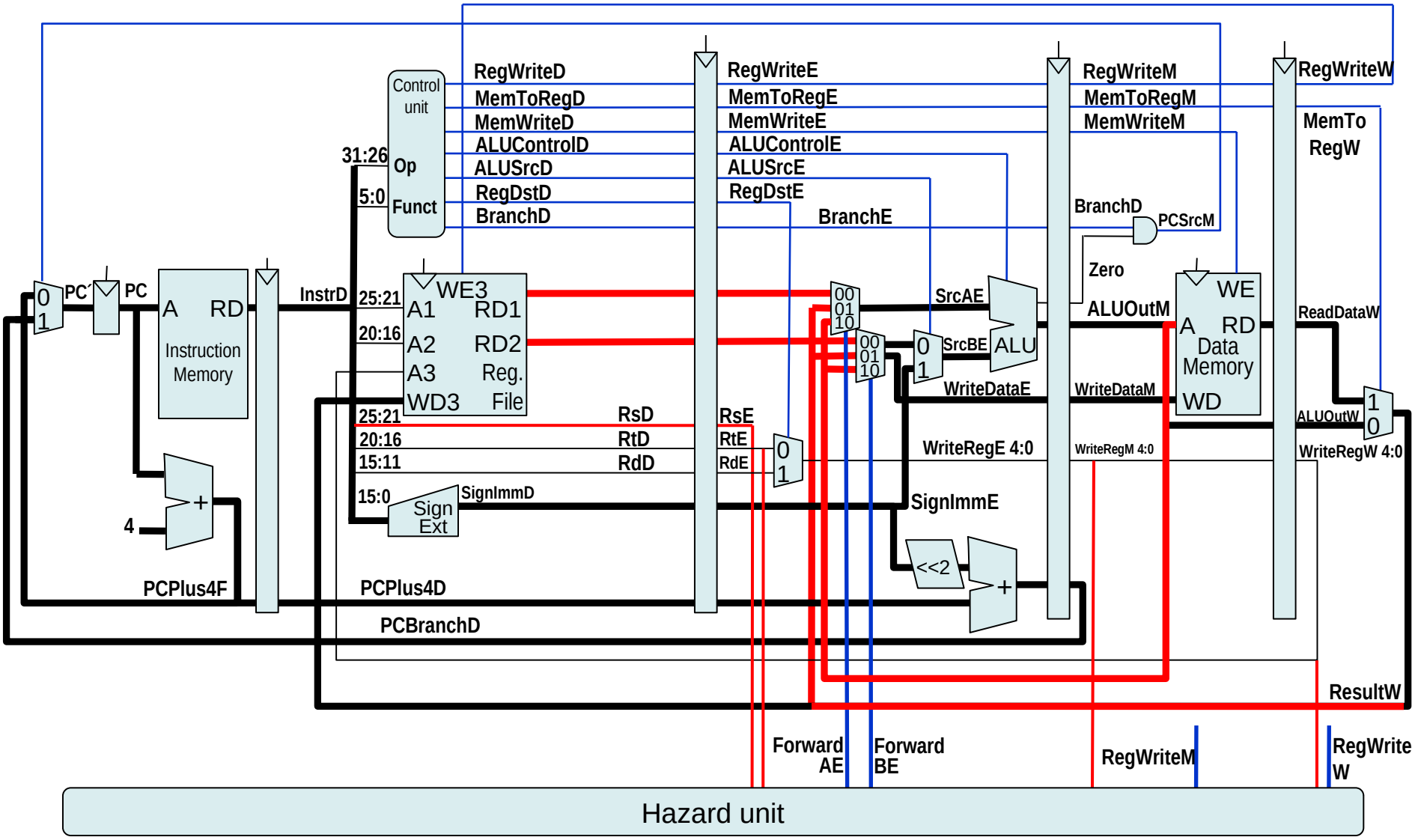


- If a result is available (computed) before subsequent instruction(s) requires the value then data hazard can be avoided by forwarding
- Hazard case is indicated when some of source registers in EX stage is the same as destination register in stage MEM or WB
- The register numbers are fed to the Hazard 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

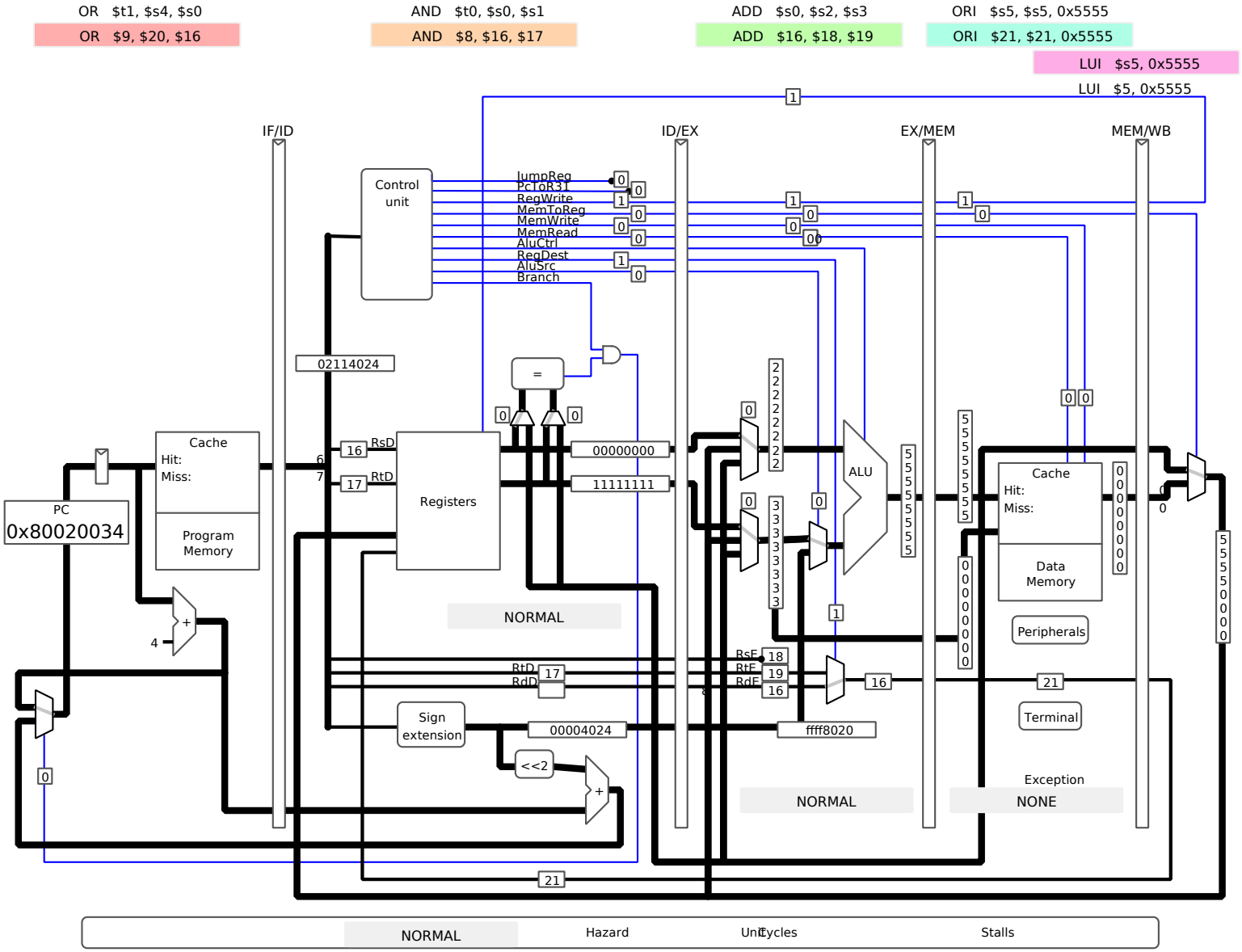
CPU after previous design steps



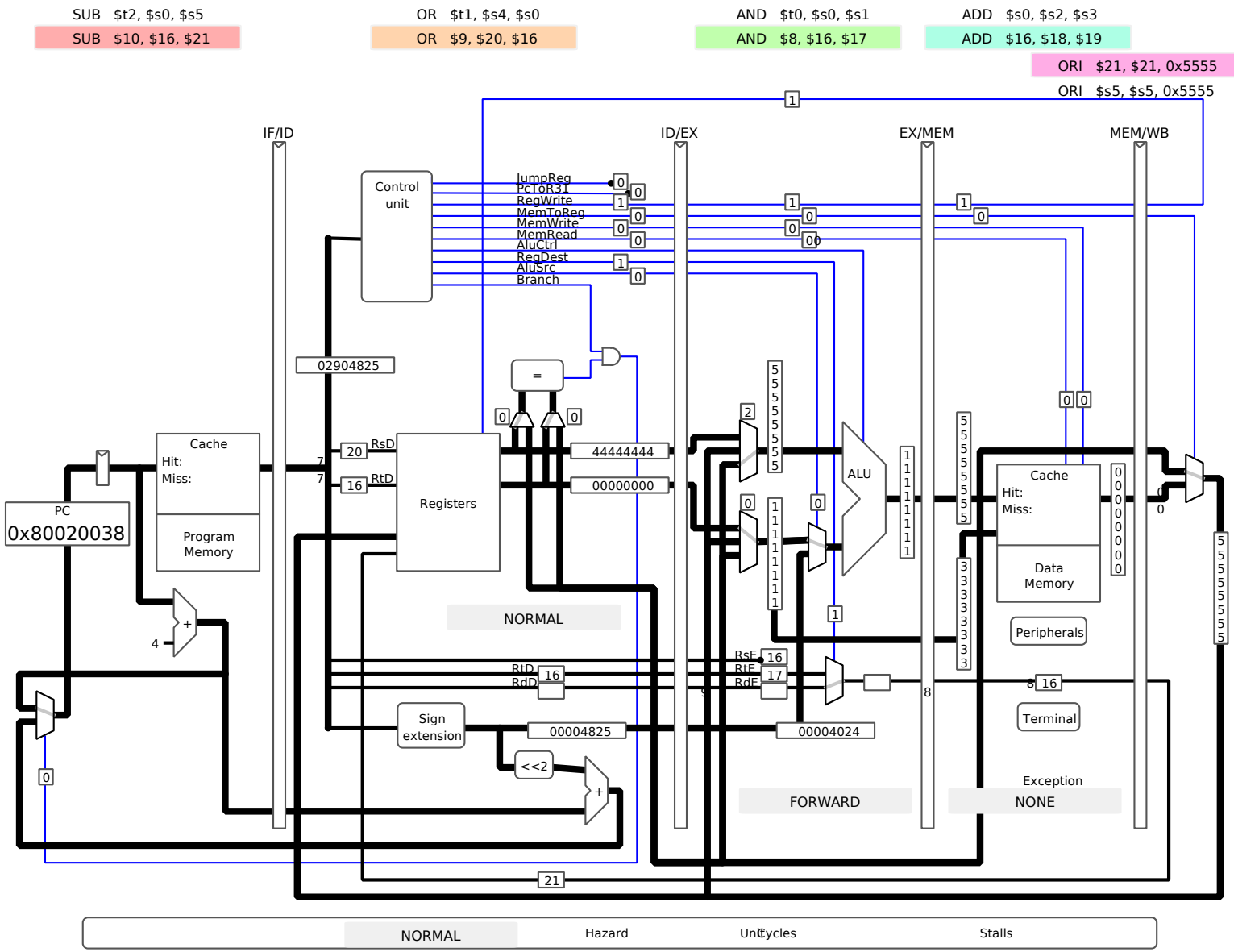
Data Hazards Solved by Forwarding



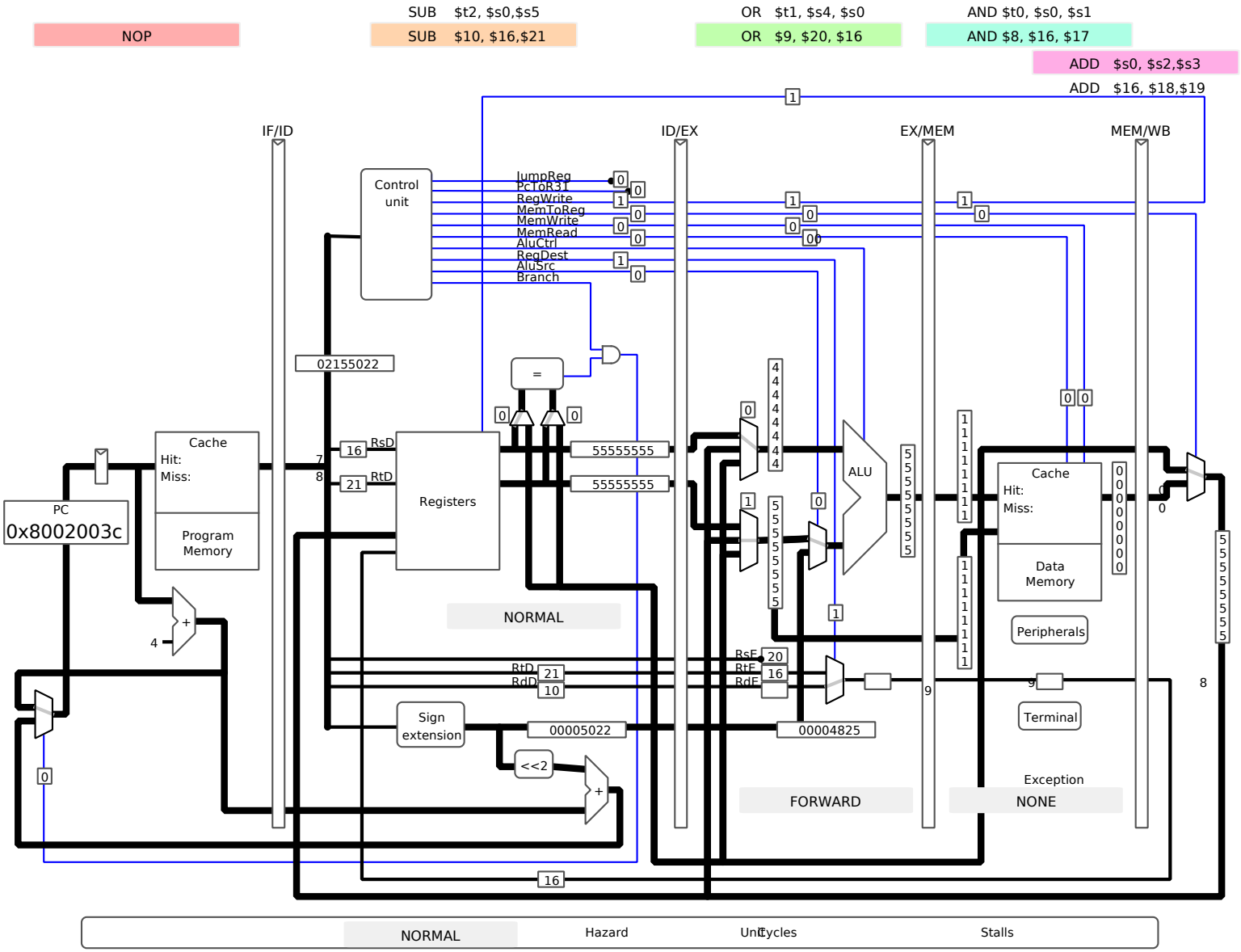
QtMips – Resolve Data Hazard by Forwarding



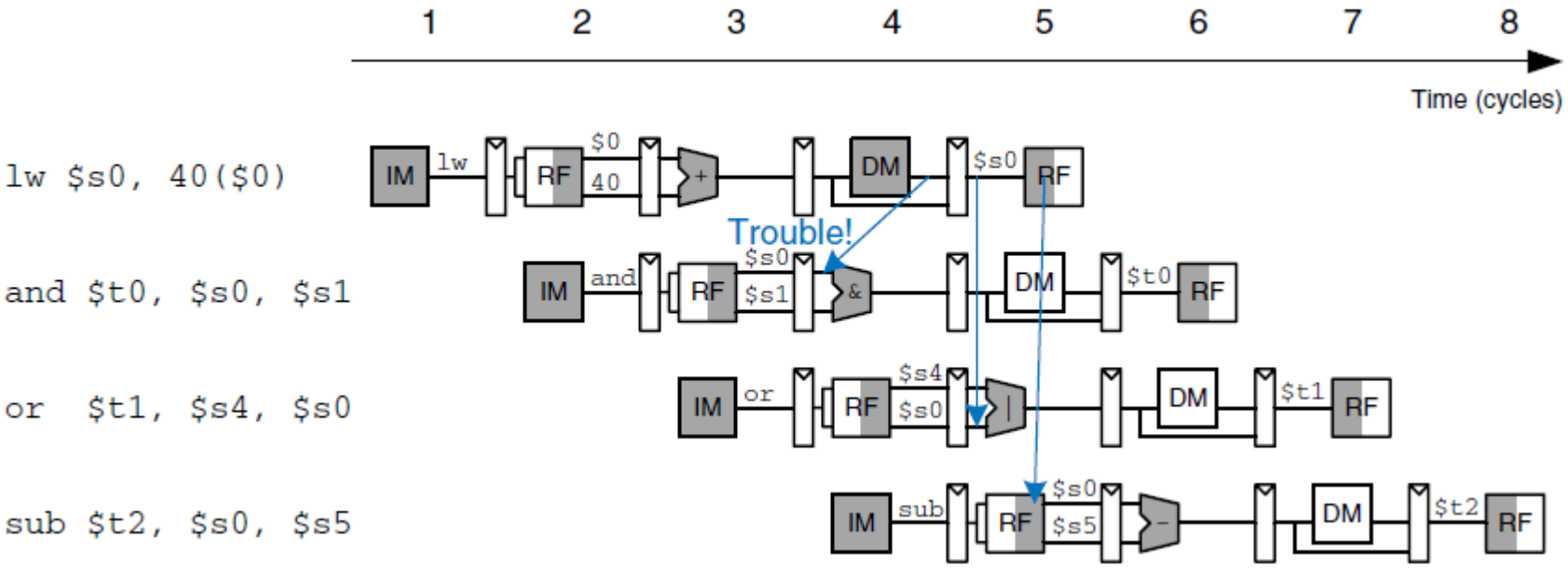
QtMips – Resolve Data Hazard by Forwarding



QtMips – Resolve Data Hazard by Forwarding

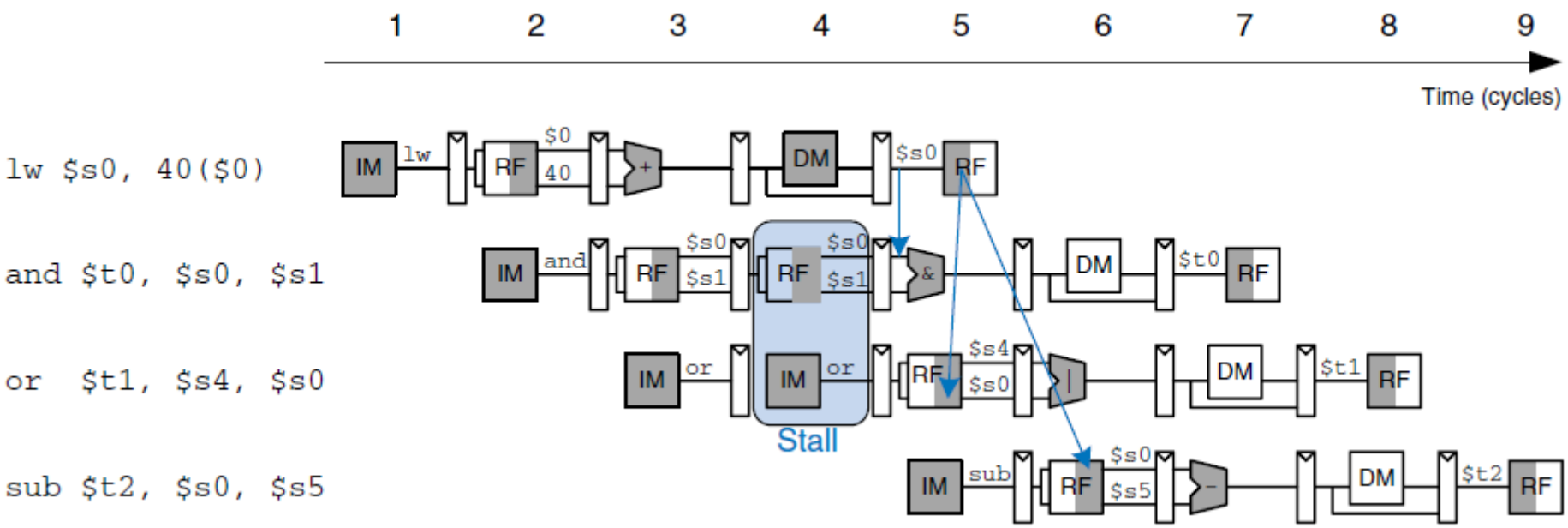


Data Hazard Avoided by Pipeline Stall



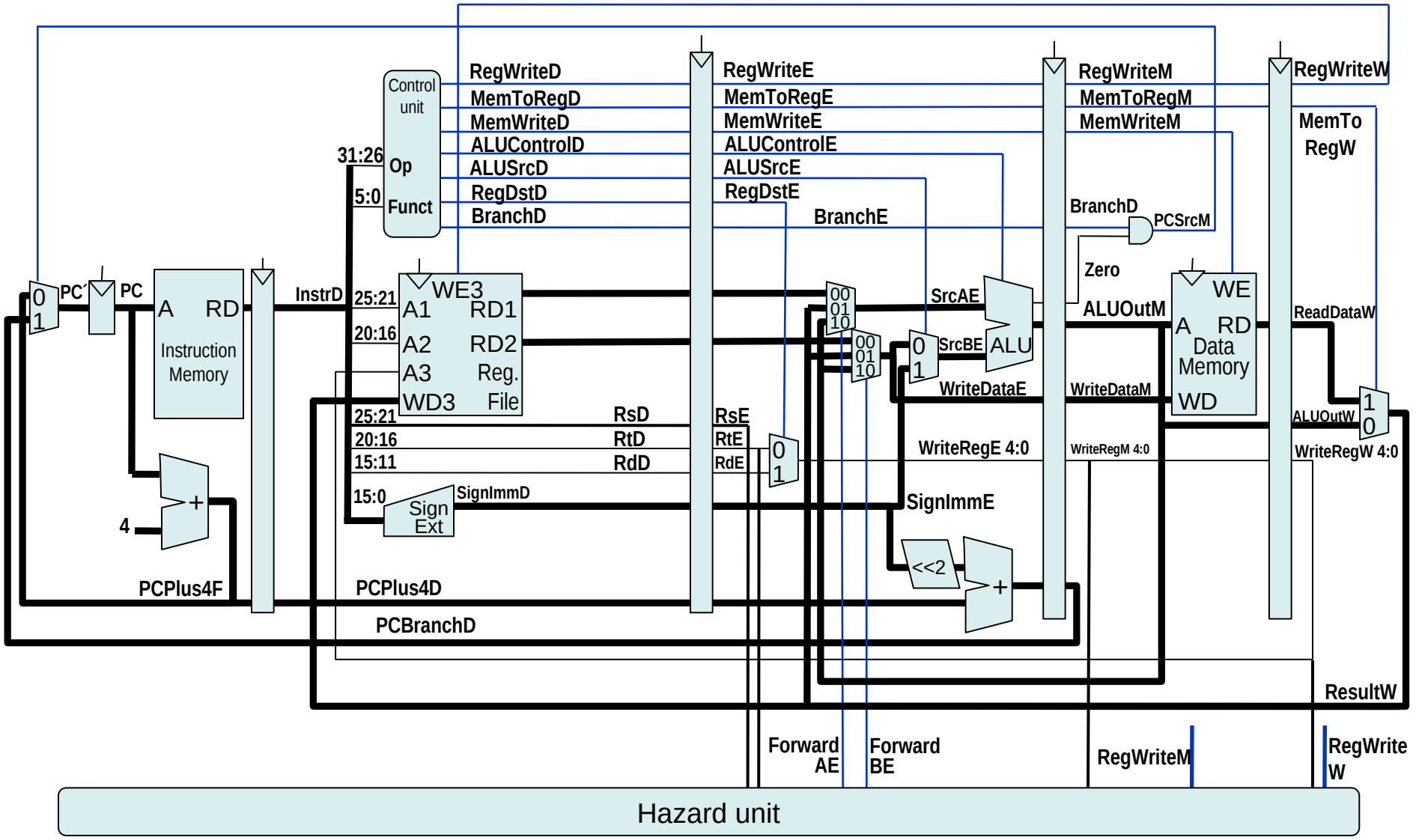
- If subsequent instructions require result before it is available in CPU then the pipeline has to be stalled (stall state inserted)
- The stall is mean to solve hazard but affect system throughput
- Pipeline stages preceding that one which is affected by the hazard are stalled until all results required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

Data Hazard Avoided by Pipeline Stall

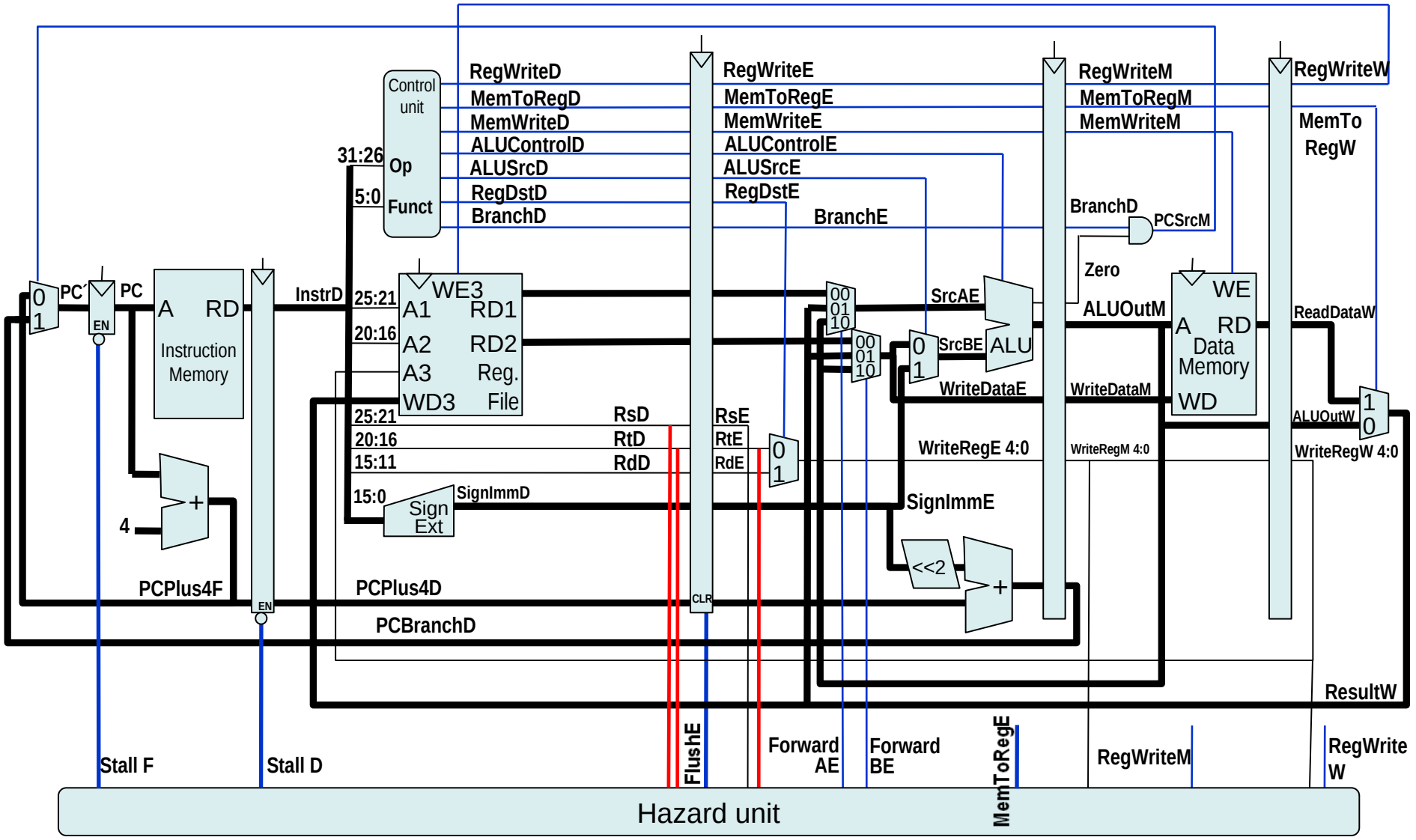


- The stall is realized by the holding content of the inter-stage registers (gating their clocks or blocking their latch enable signals)
- Results from colliding stages have to be „discarded“ – certain control signals in CPU (RF or memory write enable, branch gating) are reset (held low)
- Both is achieved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

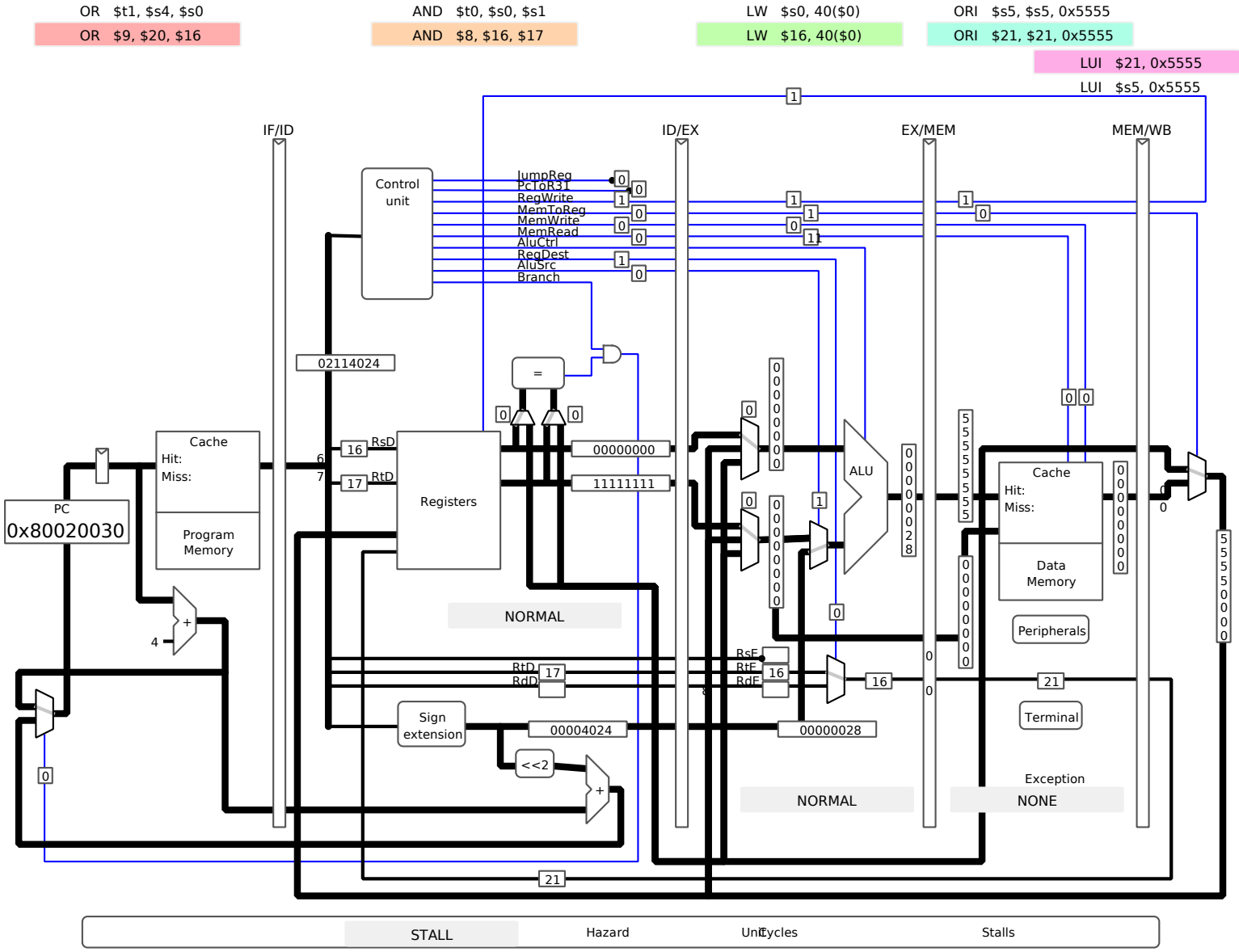
Processor Design Build Till Now



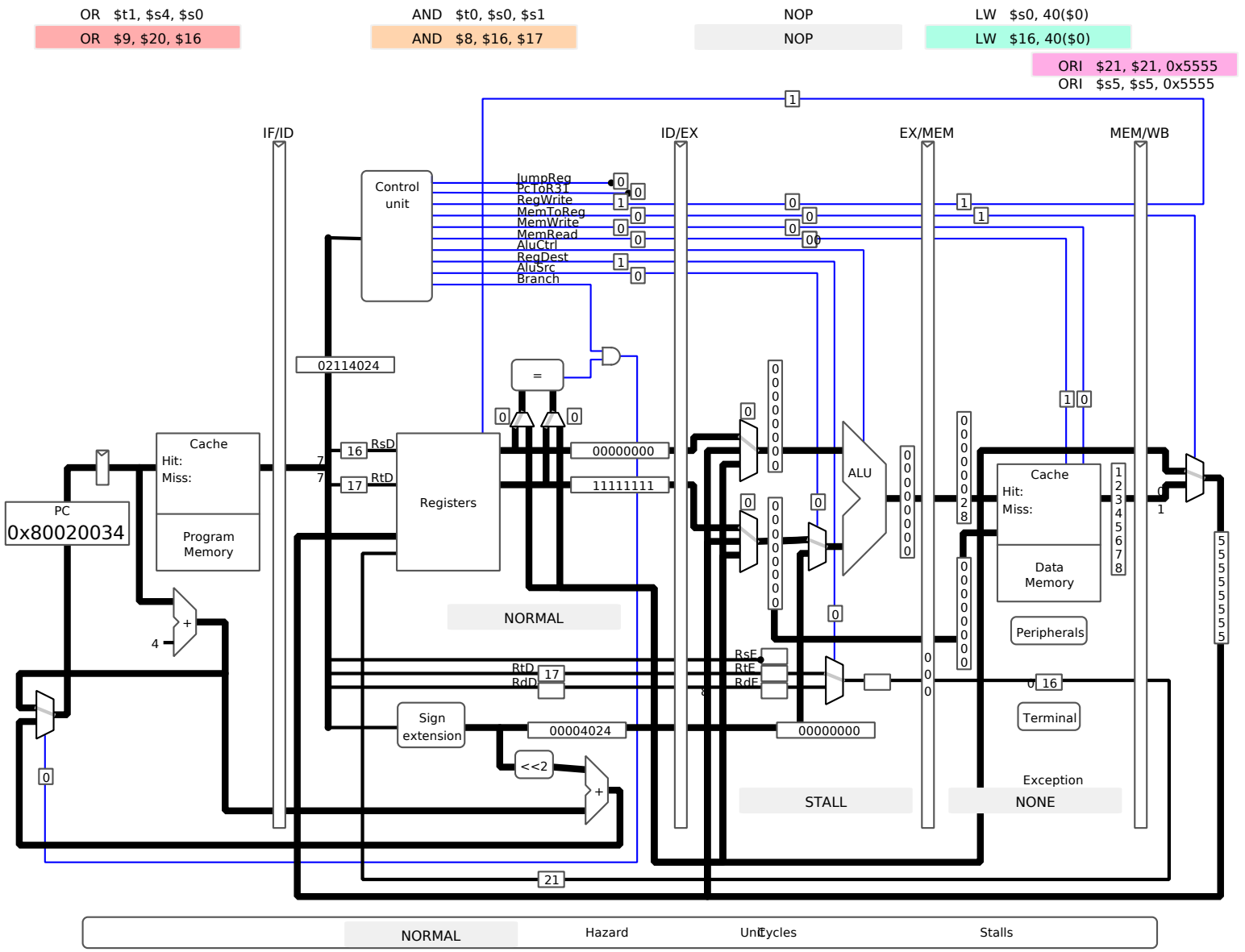
Processor with Data Hazards Avoided by Stall



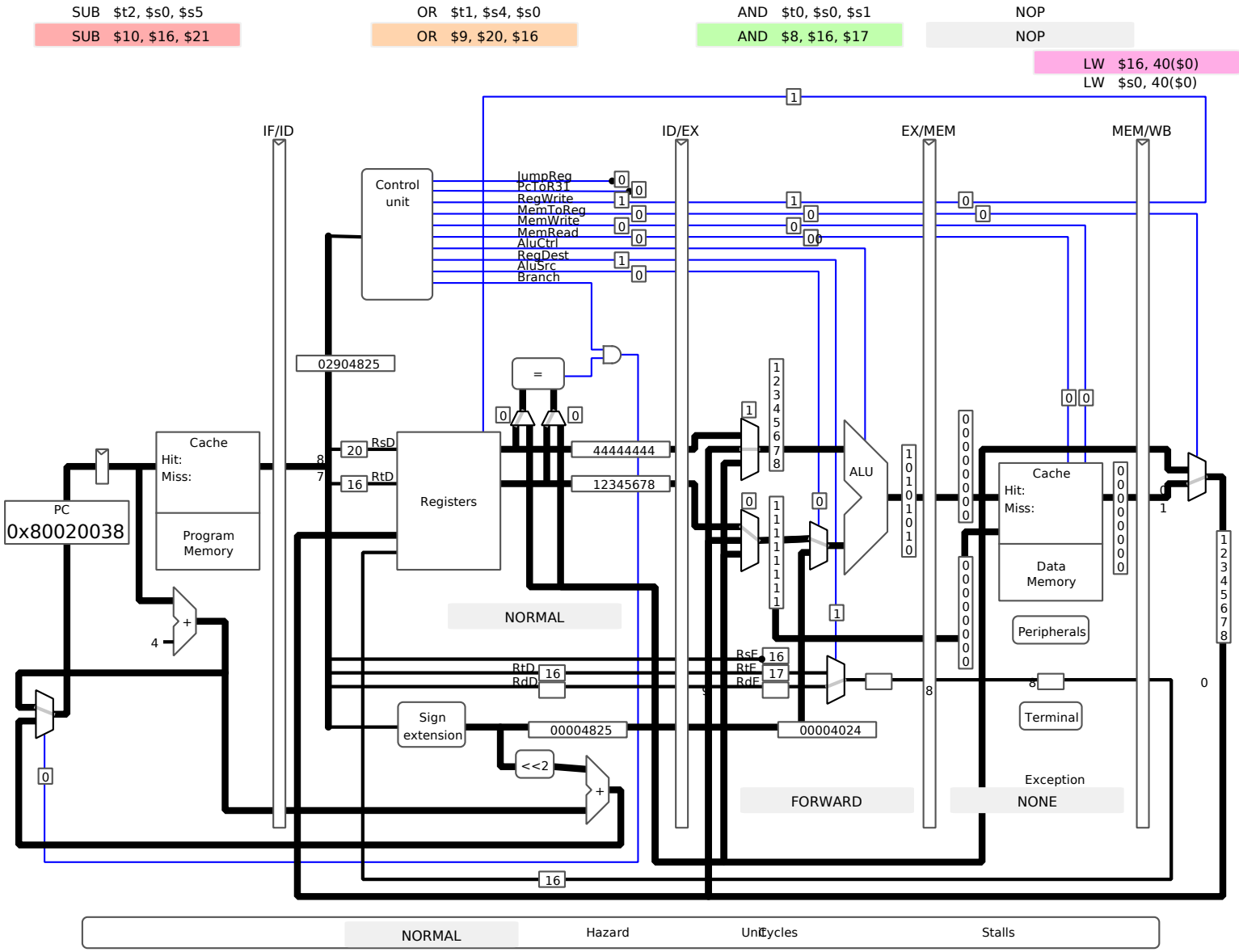
QtMips – Resolve LW Hazard by Stall



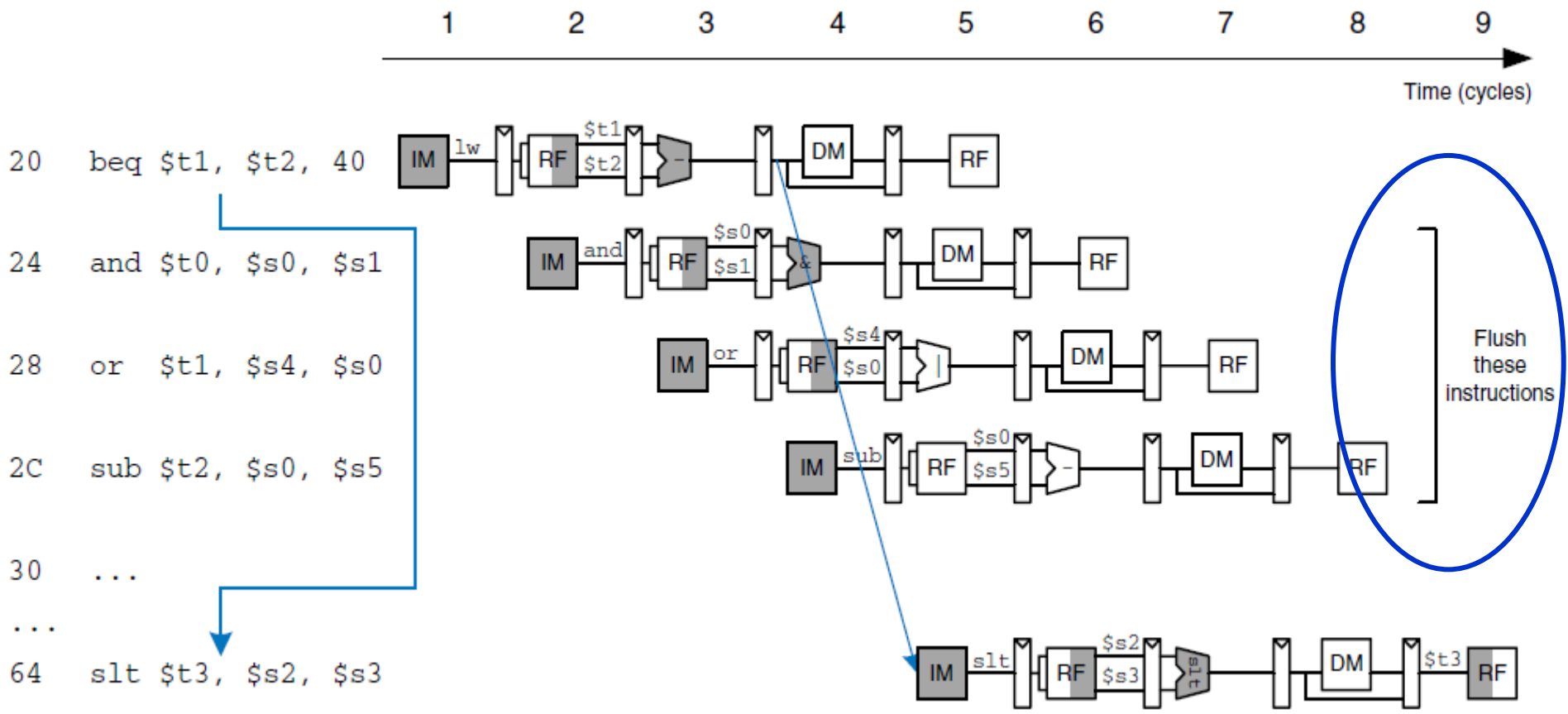
QtMips – Resolve LW Hazard by Stall



QtMips – Resolve LW Hazard by Stall

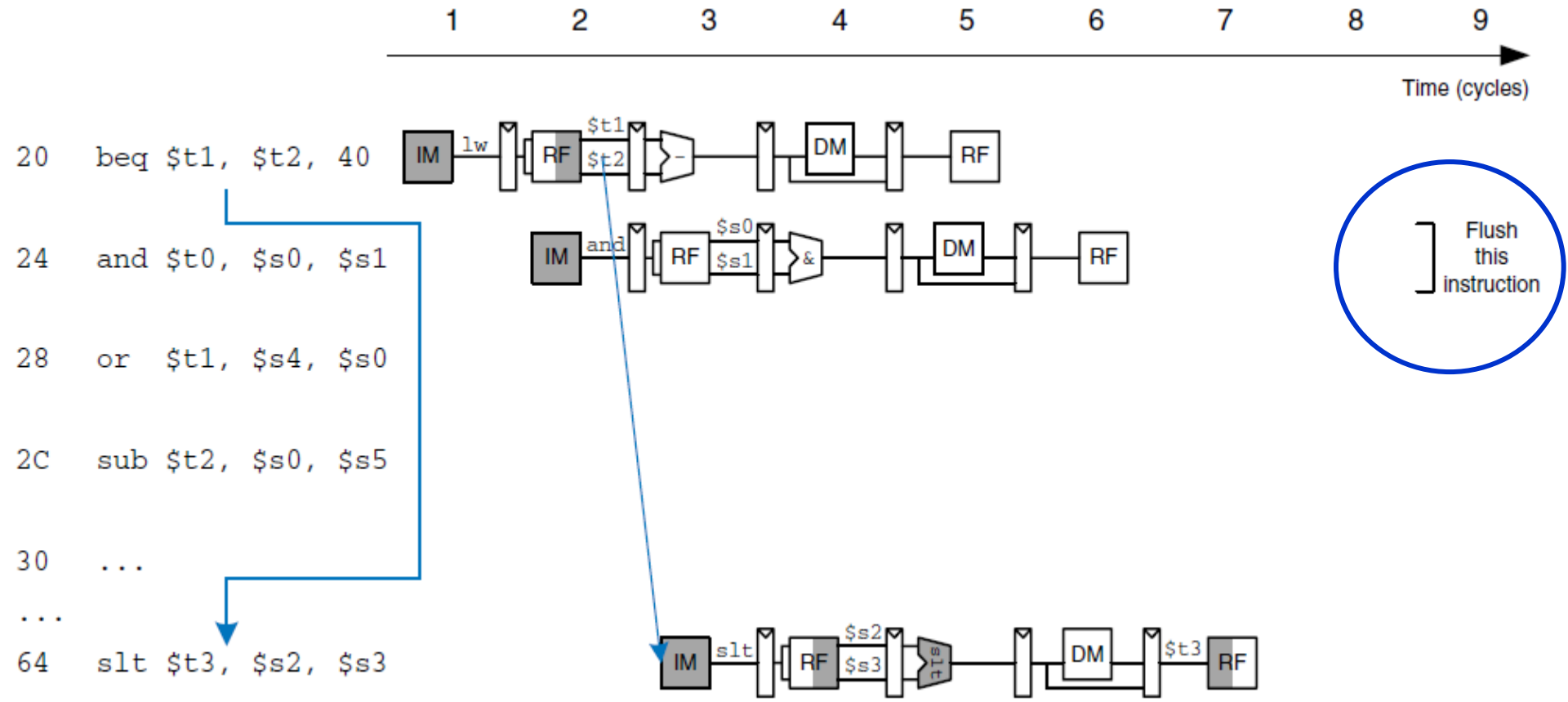


Control Hazards (Branch and Jump)



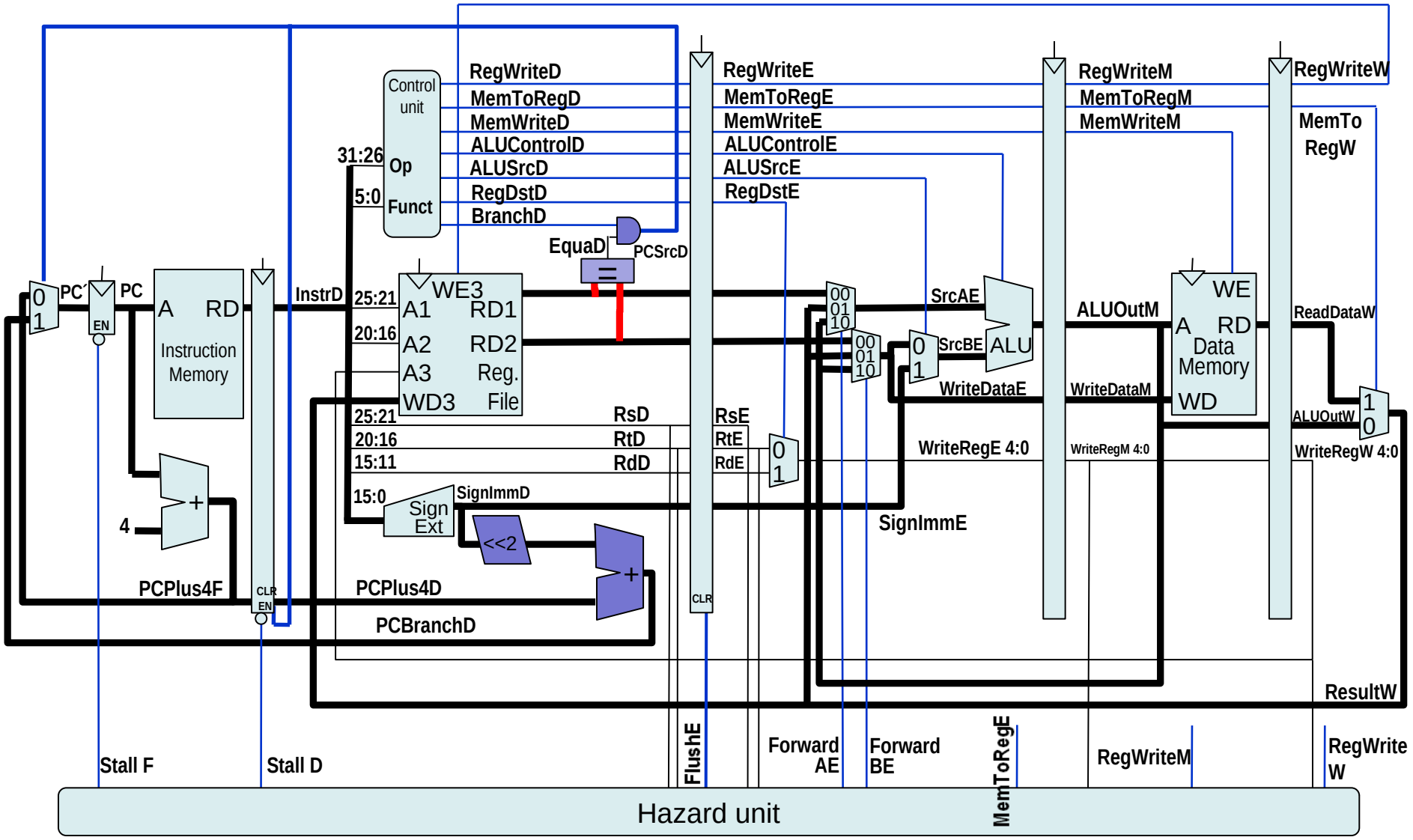
- Result is not known before 4th cycle. Why?

Control Hazards – Better to Know Result Earlier…

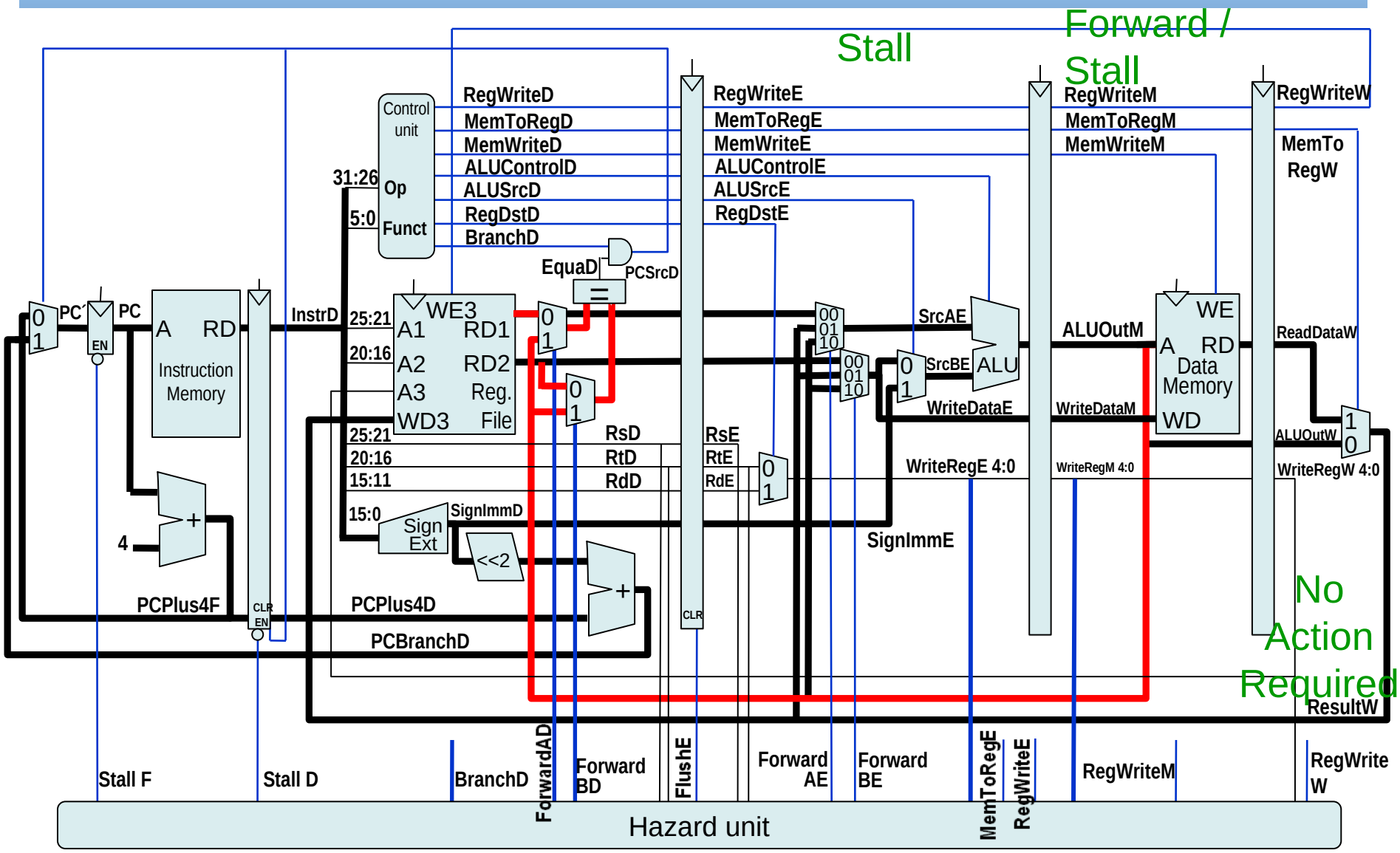


- If the result of comparison can be evaluated in the 2nd cycle **misprediction penalty** can be reduced
- But the processing of the comparison at earlier stage can induce new RAW hazards..!!!

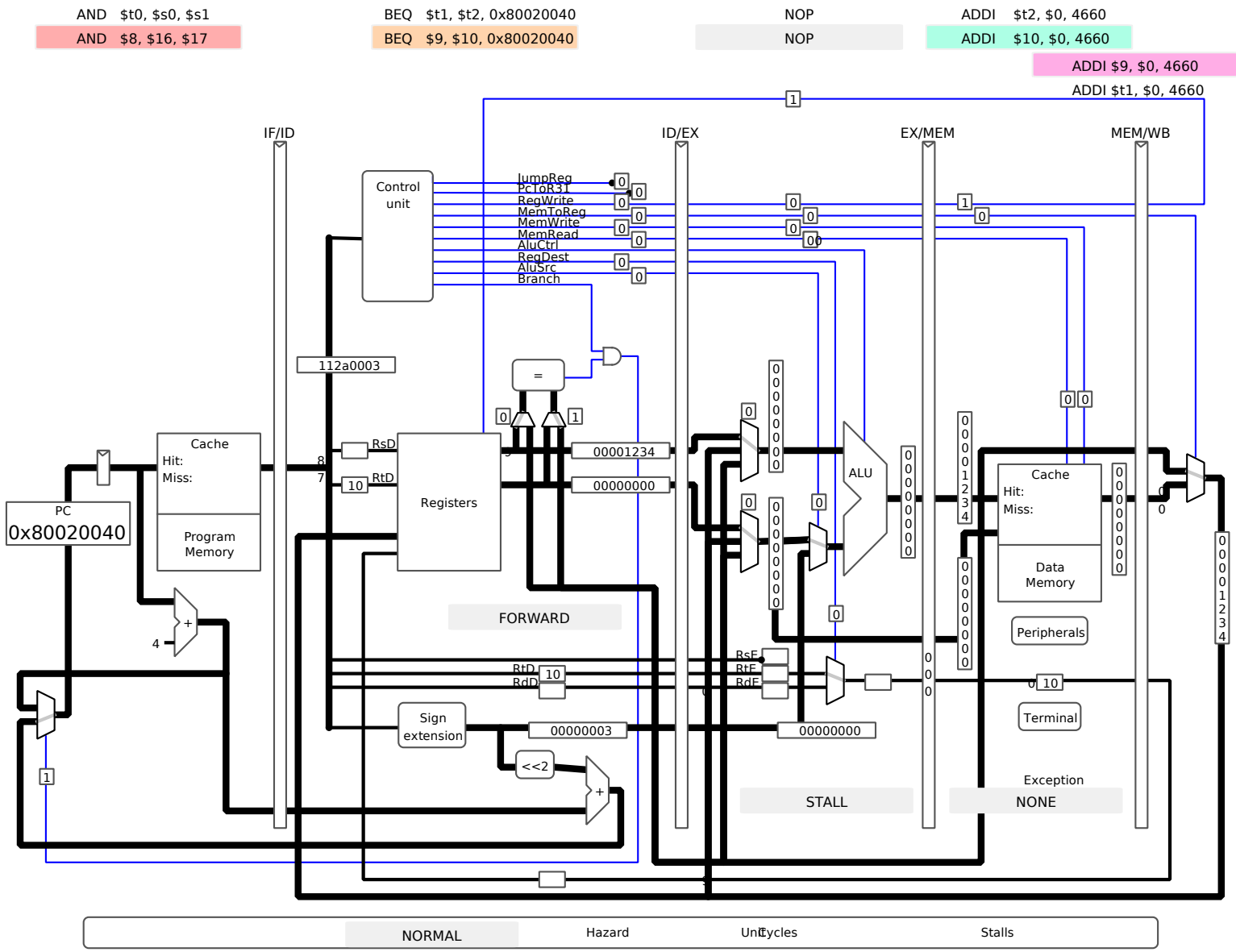
Resolve Control Hazards by Early Evaluate and Flush



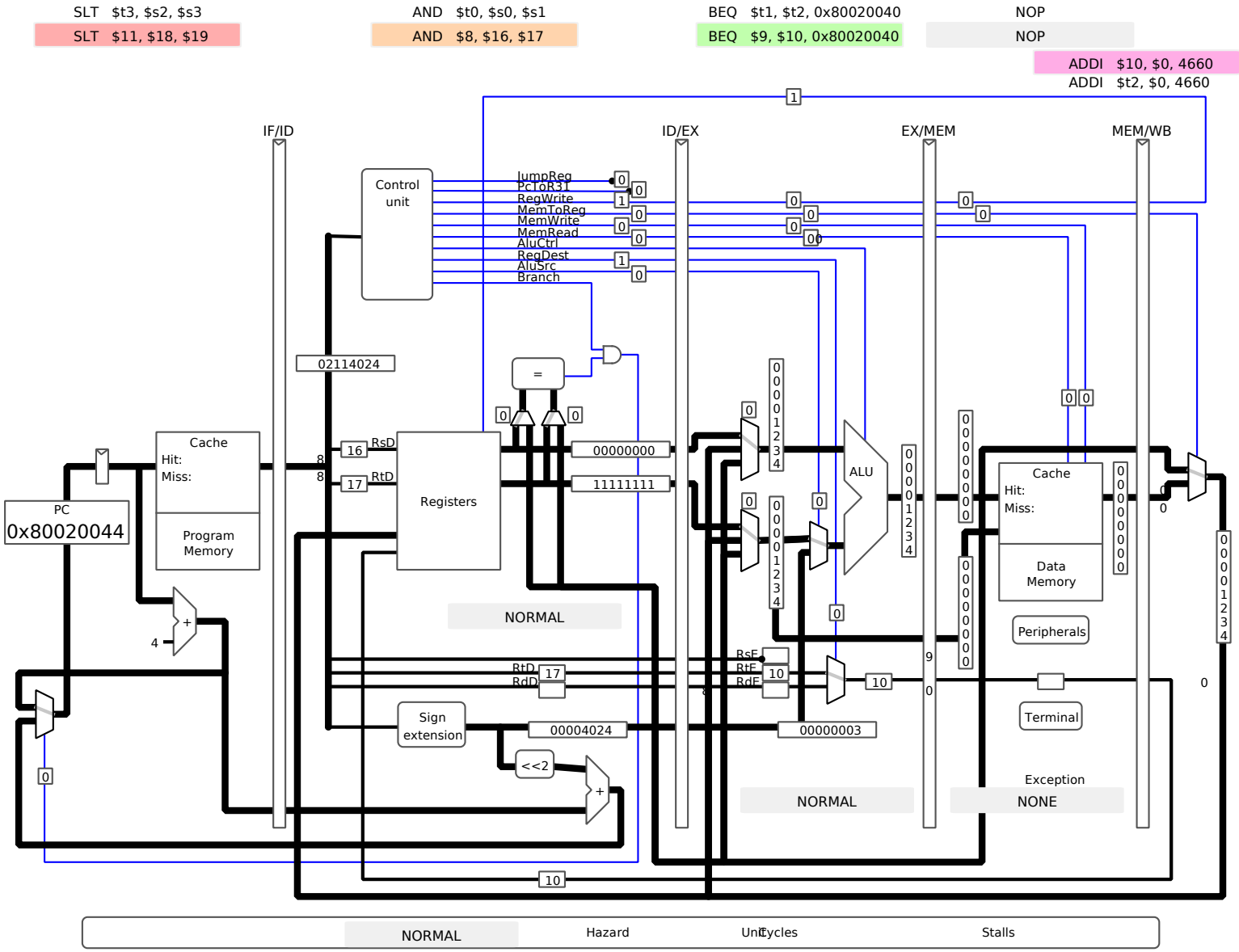
Resolve RAW Hazards by Forwarding or Stalling



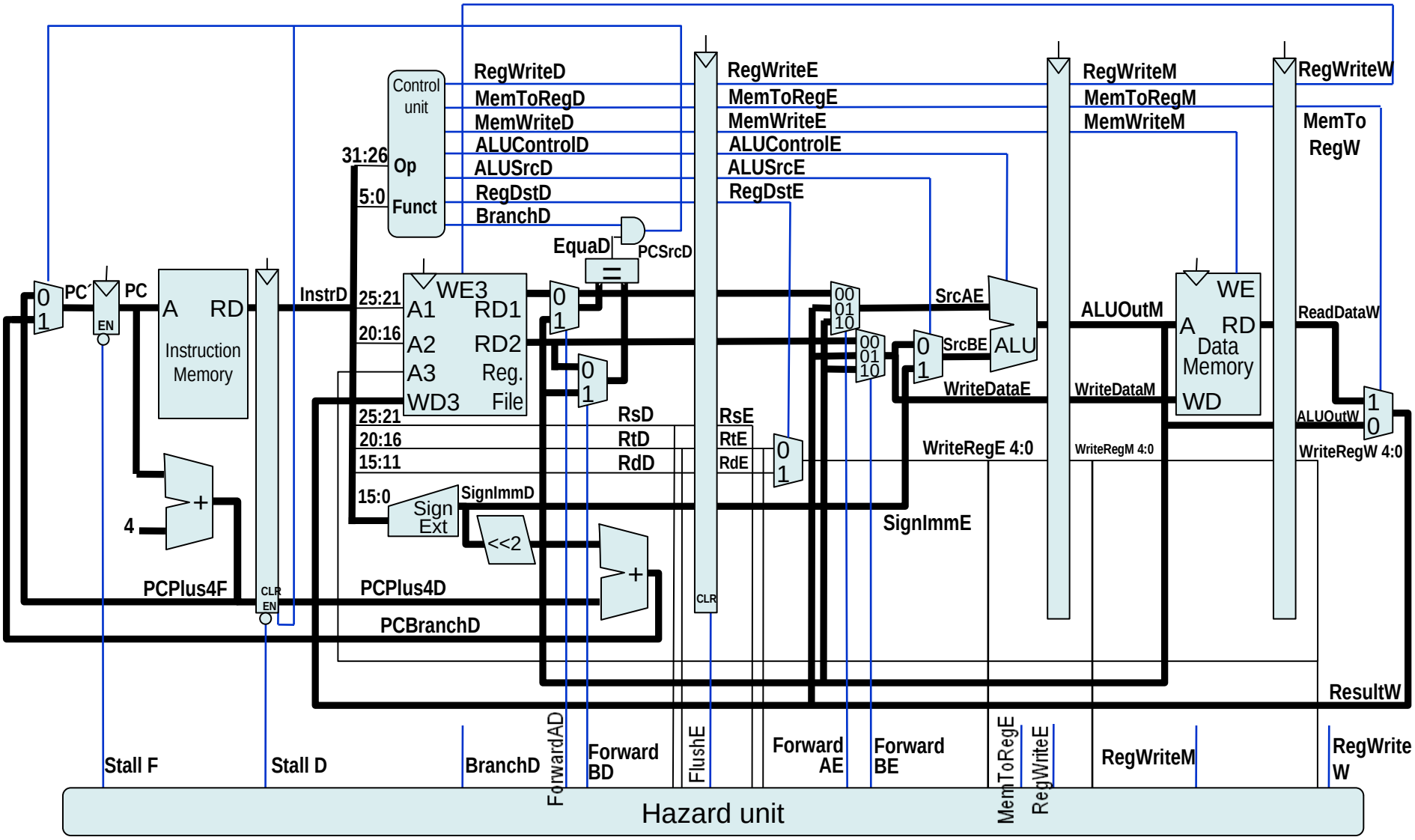
QtMips – Resolve BEQ Source Hazard by Stall



QtMips – Resolve BEQ Control Hazard



We are Finished – Pipelined Processor is Designed



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

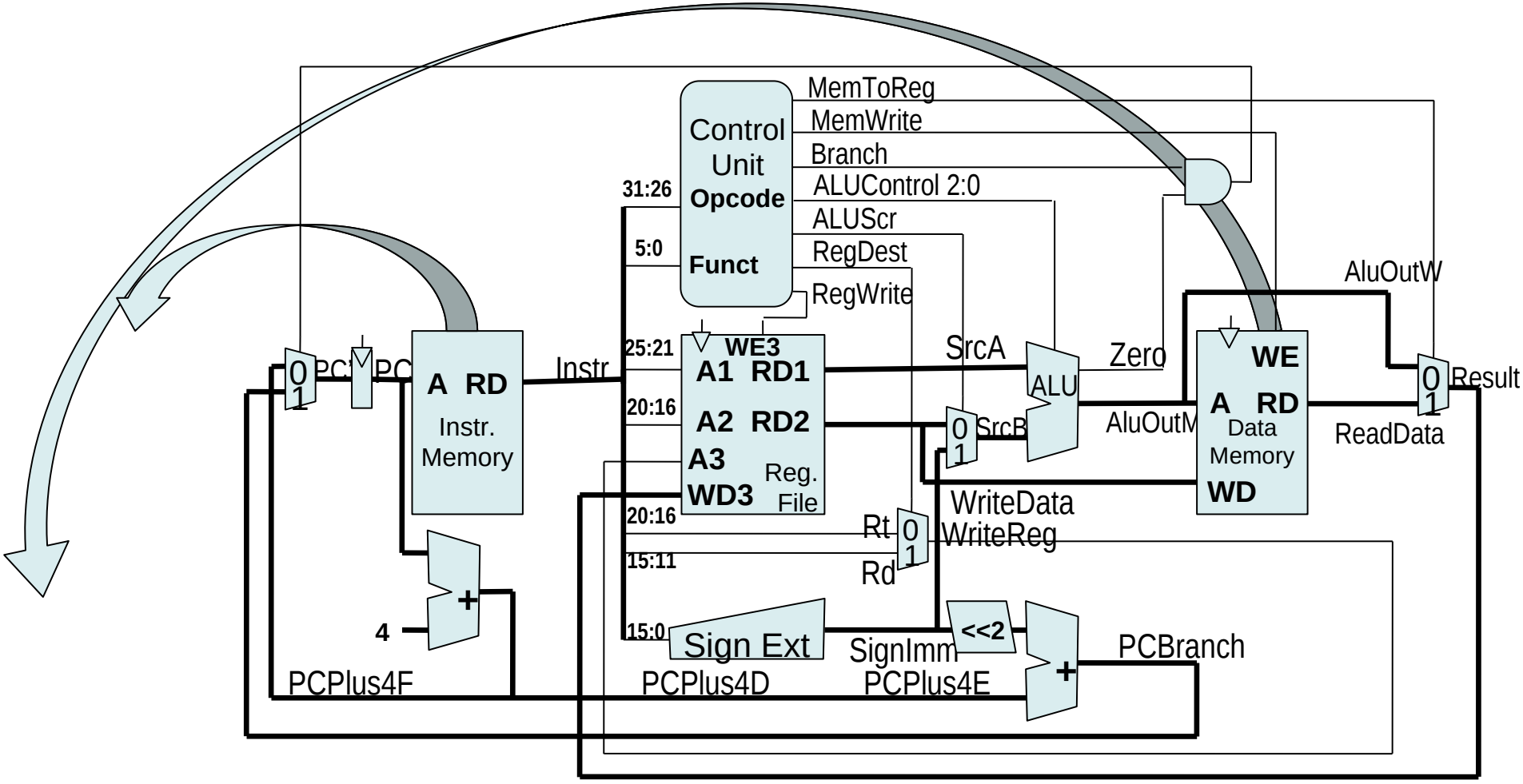
- What is maximal acceptable frequency for the CPU?
- Which stage is the slowest one?
- The cycle time is determined by the slowest stage
- For our case:
  Tc = 300 ns --> 3 333 kHz

  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)

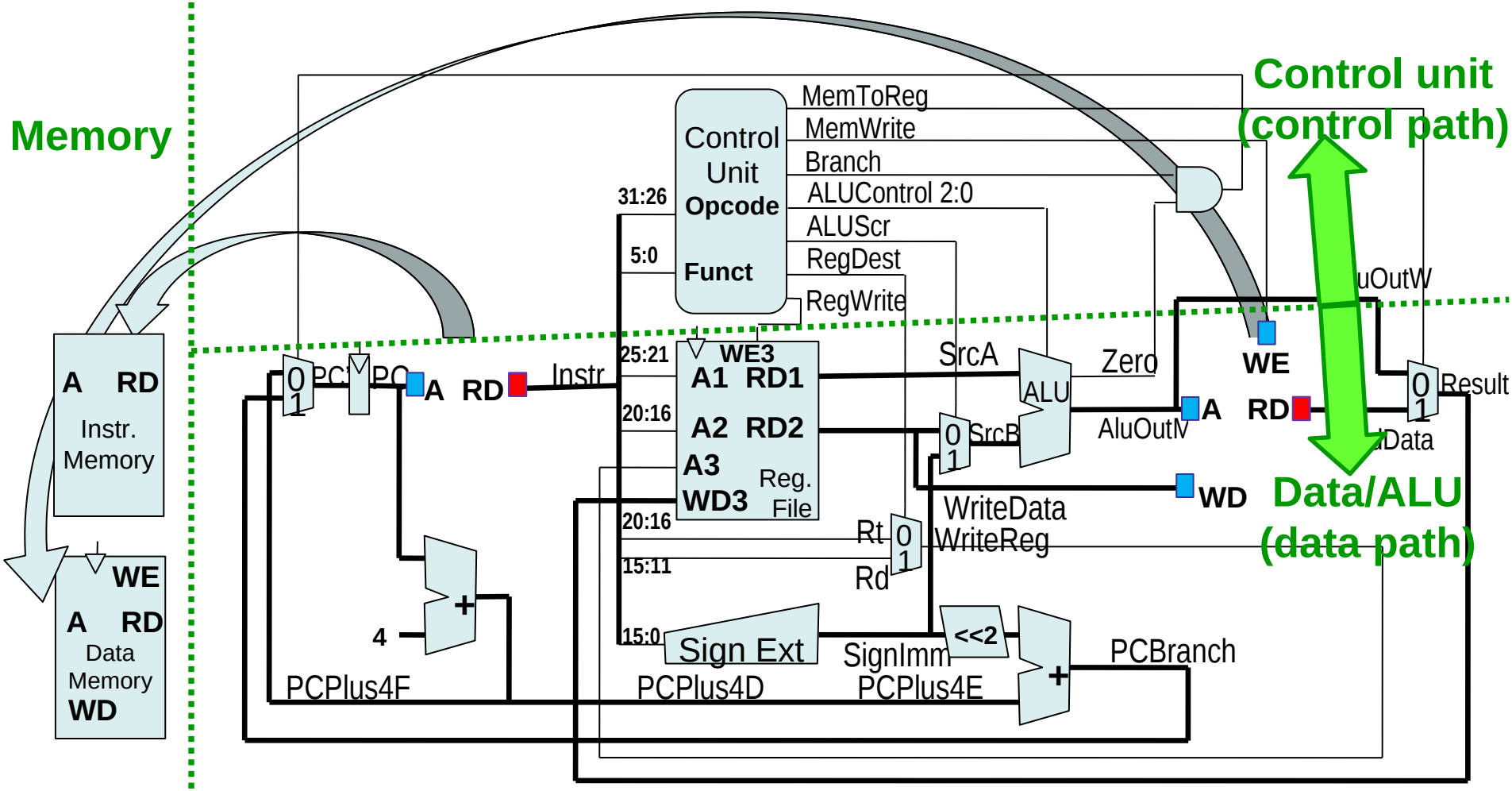
What is Result of the Design?

Return back to non-pipelined CPU version

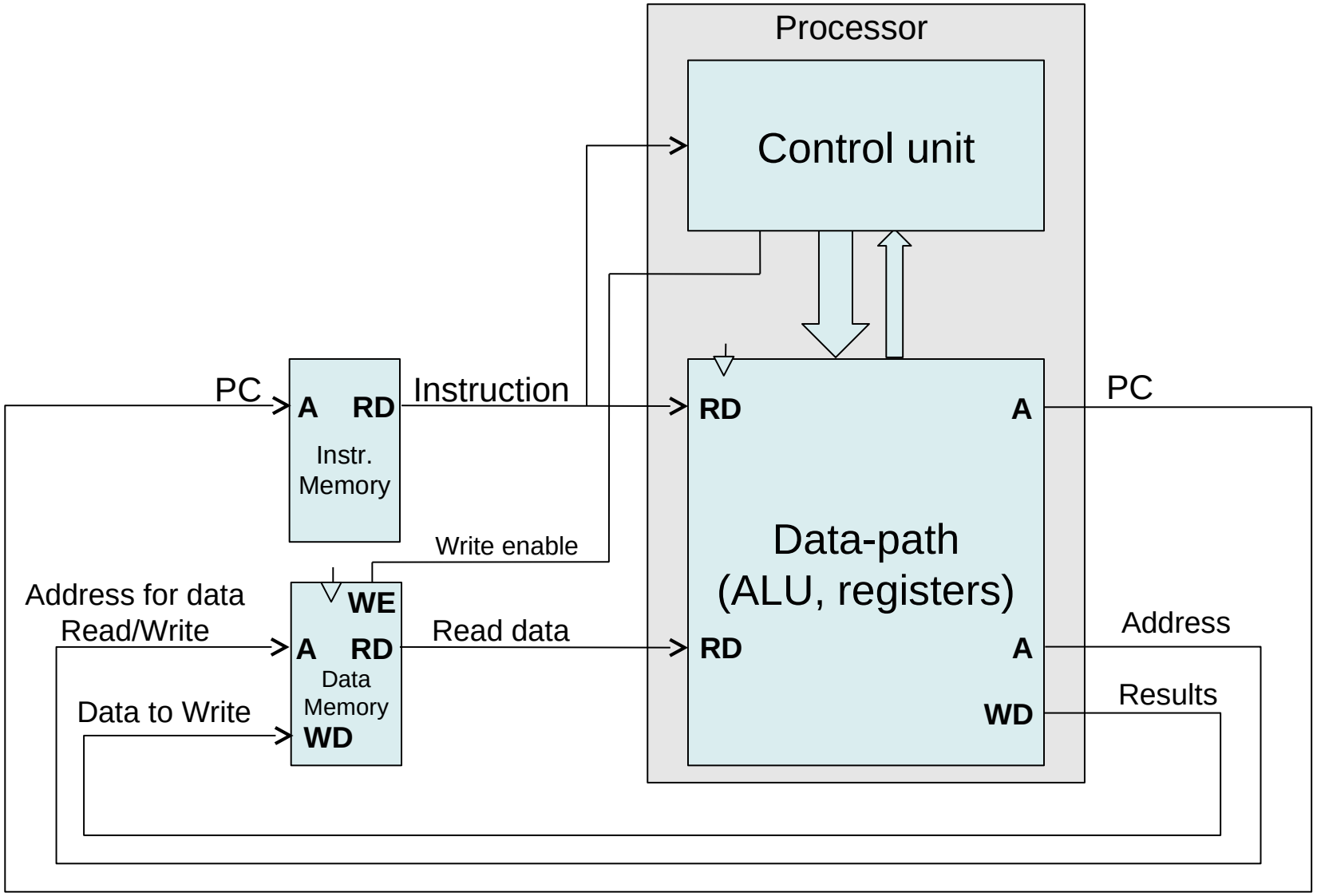


What is Result of the Design?

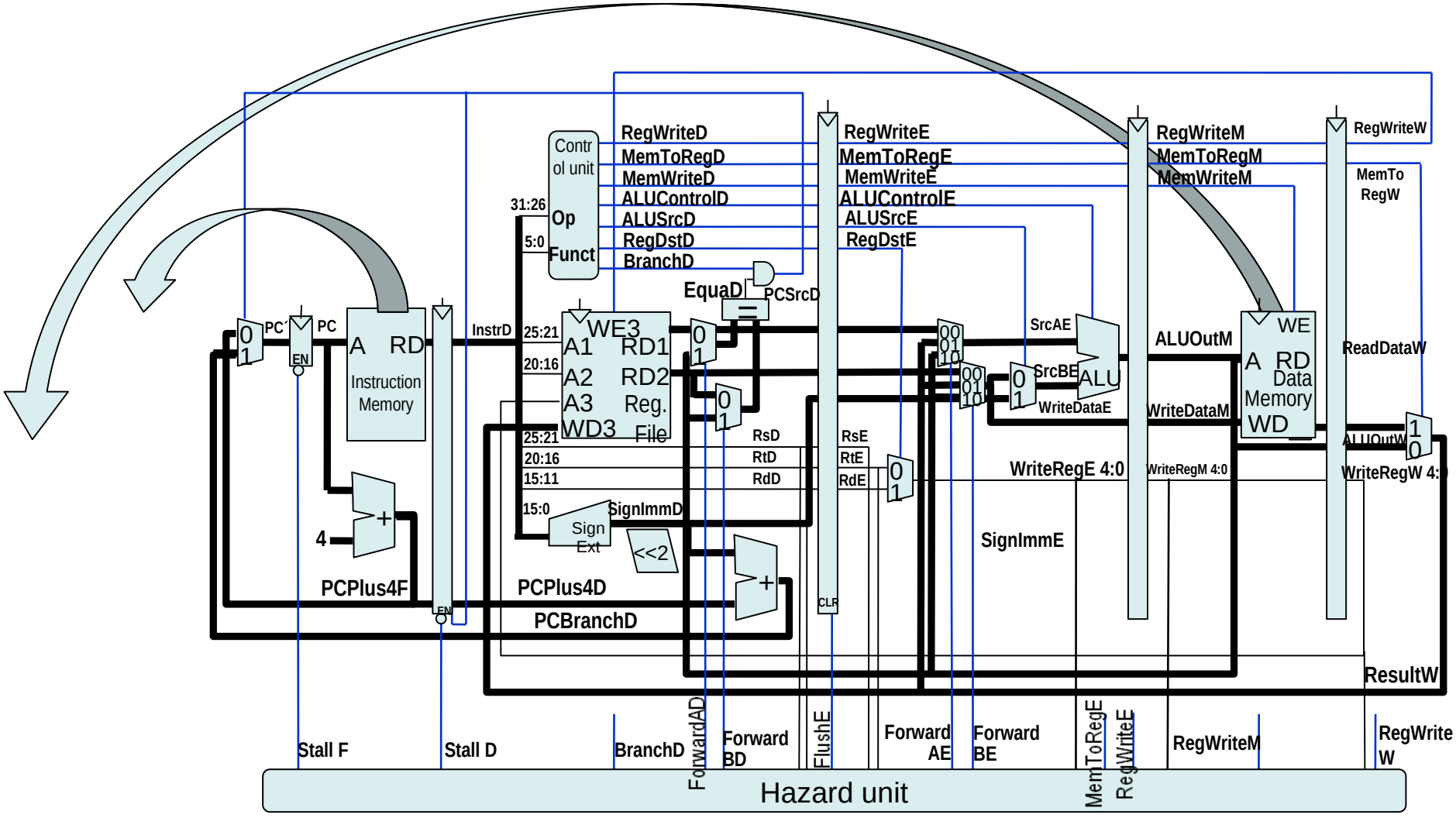
Return back to non-pipelined CPU version



What is Result of the Design?

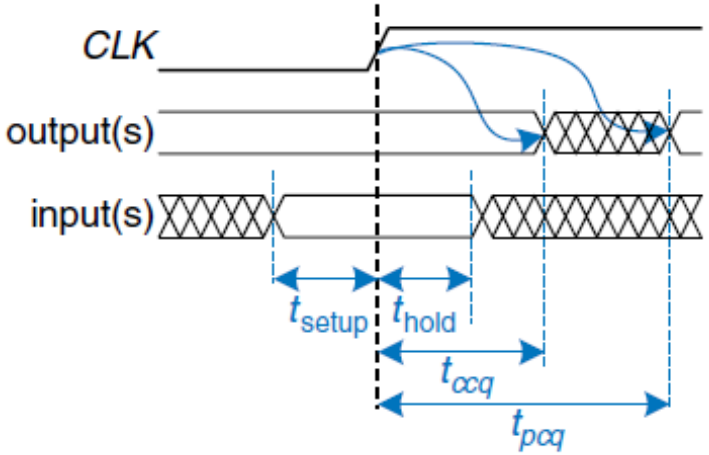


CPU Design Result – Pipelined Version



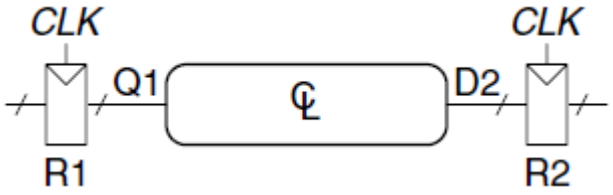
Pipelined CPU – Timing

- The timing/AC characteristics of synchronous sequential circuit :

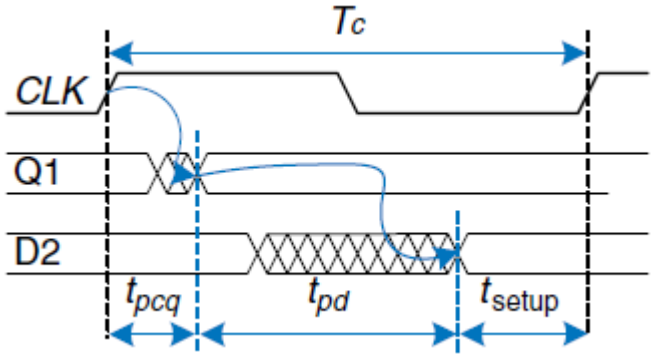


t_{setup} – inputs setup time
t_{hold} – inputs hold time

- Signal integrity constrain for the setup time before the clock:



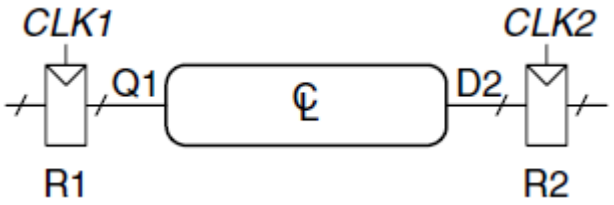
$$Tc >= t_{pcq} + t_{pd} + t_{setup}$$

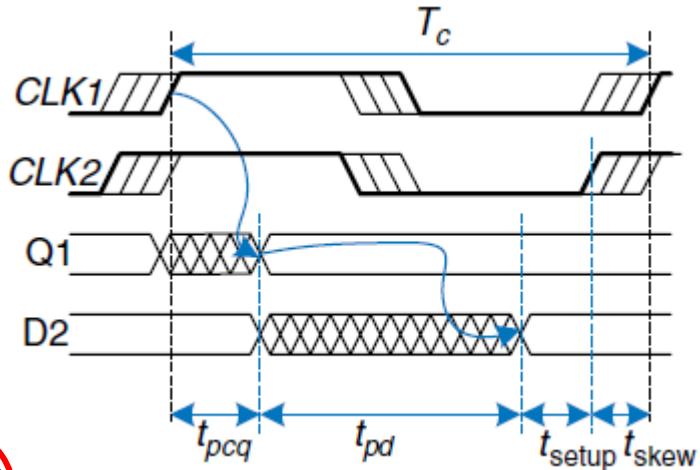


t_{pd} – combinatorial logic propagation delay

Pipelined Processor – Timing

- Constraint for the setup time (consider the clock distribution jitter):

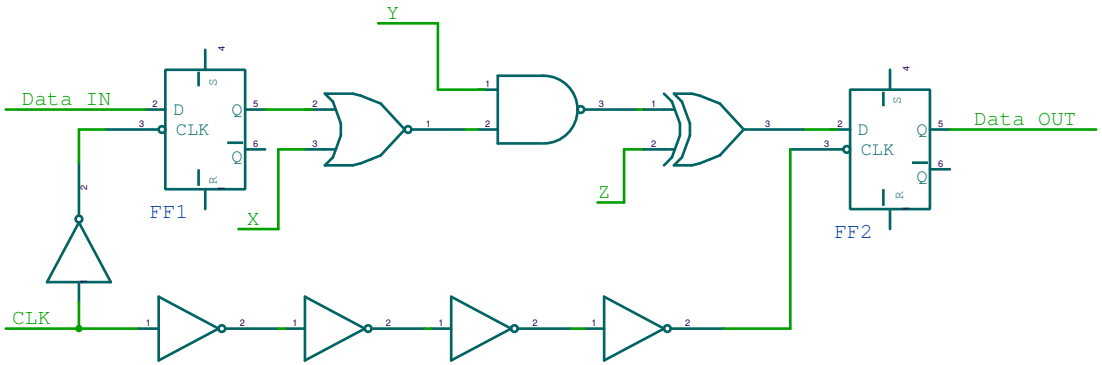




$$Tc >= t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

Clock distribution jitter is limiting factor,
if it reaches or exceeds value of t_{pd}
(too deep pipeline / too many stages…)

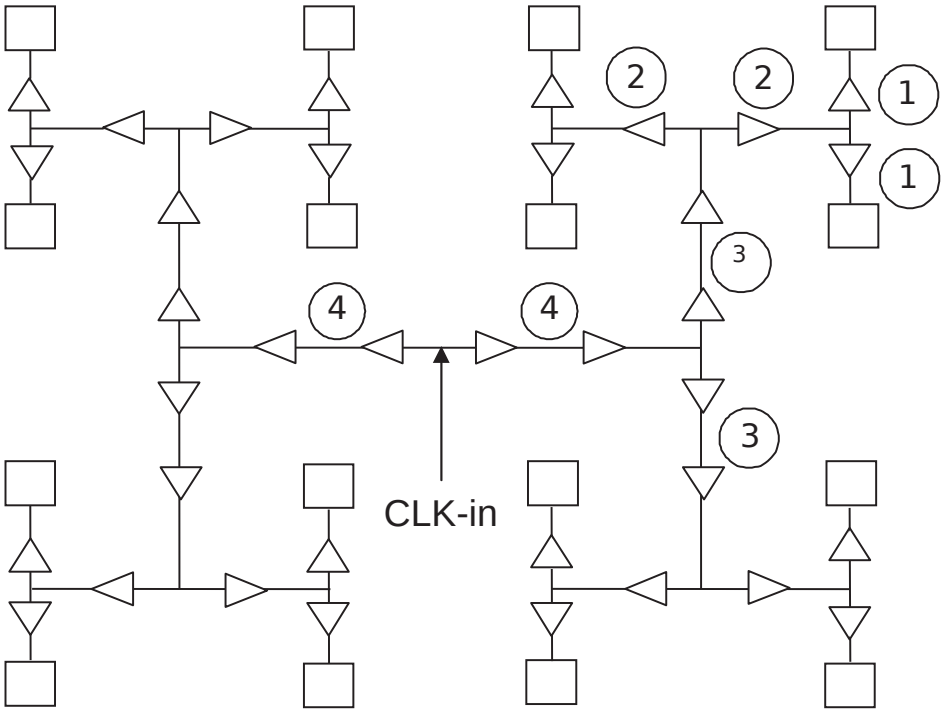
Clock Distribution Network Skew



- **Positive Clock Skew** – clock arrives at the capturing sequential later than it arrives at the launching sequential
- **Negative Clock Skew** – clock arrives at the launching sequential later than it arrives at the capturing sequential
- **Local Clock Skew** – skew between any two sequentials with a valid timing path between them.
- **Global Clock Skew** – clock skew between any two sequentials in the design irrespective of whether a timing paths exists between them

diagram source: peda clock-tree-skew

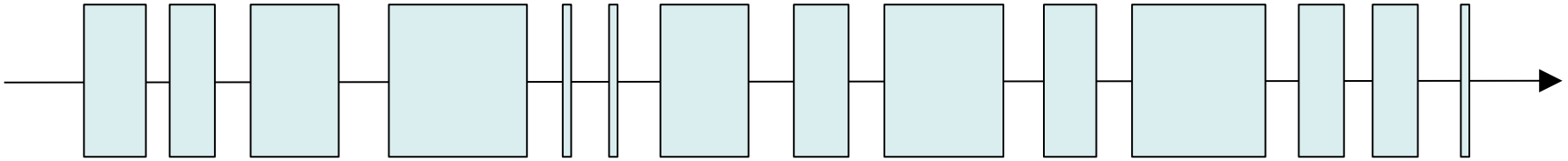
Clock Distribution Network – H-tree



source: Tawfik, S., Kursun, V.: Clock Distribution Networks with Gradual Signal Transition Time Relaxation for Reduced Power Consumption.

Pipeline Stages Balancing

Linear pipelining:

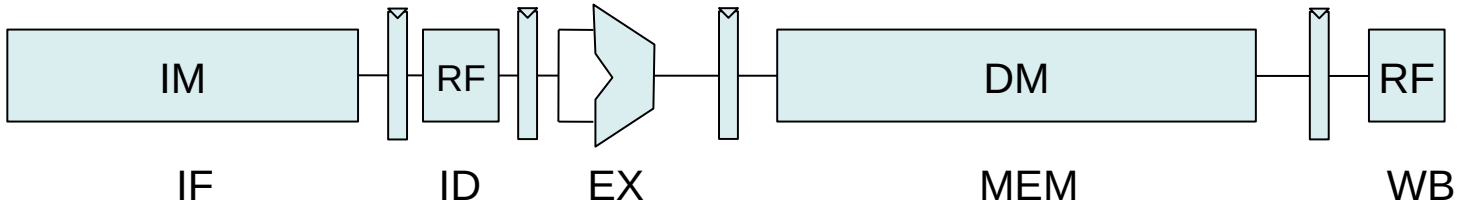


(applies to tree based adder, multiplier, (unrolled) iterative divider..)

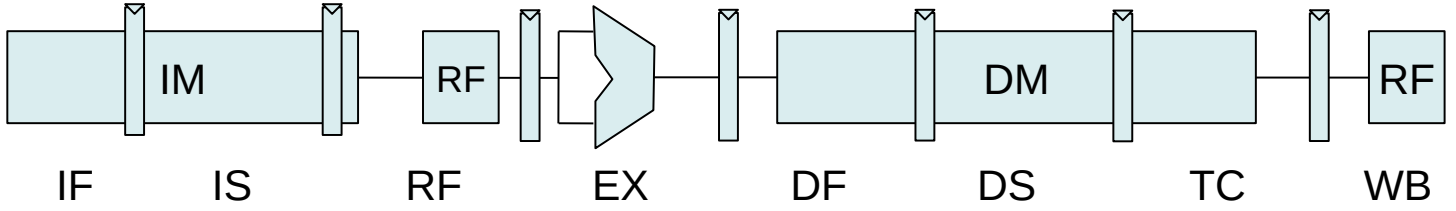
- **Balancing**: the goal is to divide the processing into *N* stages in such way, that stage propagation delays are roughly the same…
- The number of stages reflects preference of performance (throughput) versus latency.

Superpipeline and Beyond

- Not well balanced 5-stage pipeline:



- Deeper pipeline is result of decomposing stages into more stages



- It allows CPU to work at higher frequencies but introduces many problems as well..
- Complex forwarding, more pipeline stalls, hazards need to be solved by complex logic

Typical Pipeline Depths in Today's CPUs

P5 (Pentium) : **5**
P6 (Pentium 3): **10**
P6 (Pentium Pro): **14**
NetBurst (Willamette, 180 nm) - Celeron, Pentium 4: **20**
NetBurst (Northwood, 130 nm) - Celeron, Pentium 4, Pentium 4 HT: **20**
NetBurst (Prescott, 90 nm) - Celeron D, Pentium 4, Pentium 4 HT, Pentium 4 ExEd: **31**
NetBurst (Cedar Mill, 65 nm): **31**
NetBurst (Presler 65 nm) - Pentium D: **31**
Core : **14**
Bonnell: **16**

Haswell **14**-**19**
Cooper Lake **14**-19

K7 Architecture - Athlon : **10**-**15**
K8 - Athlon 64, Sempron, Opteron, Turion 64: **12**-**17**

AMD Zen **19**
AMD Zen2 **19**

ARM 8-9: **5**
ARM 11: **8**
Cortex A7 2-wide **8**-**10**
Cortex A8 2-wide **13**
Cortex A15 3-wide **15**-**25**

Cortex-A35 2-wide **8**
Cortex-A53 2-wide **8**
Cortex-A57 3-wide **15**
Cortex-A77 4-wide **11**–**13**
Denver 2/7-wide **13**
M4 6-wide **15**
Lightning 7-wide **16**

SiFive FE310-G000 **5**
SiFive FU540-C000 **5**

- The Optimum Pipeline Depth for a Microprocessor:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4333&rep=rep1&type=pdf

Branch Stall Discussion and Delay Slots

- The instruction memory read and fetch is expensive and result of condition evaluation in branch instructions (even worse target in indirect branch instructions) has to be evaluated before next fetch and execute. The **stall** state is waste of cycles. Options to use that cycle(s) are:
  - Start fetch and execution of instruction(s) following branch and flush/discard results if it is resolved that it should not be executed
  - Extend above by adding condition results/branch predictor (**taken**/**not-taken**) and branch target cache (BTB)
  - Execute one or more instructions after branch unconditionally in (so called) **delay slot**
- Delay slots unconditional execution is common for many DSP (digital signal processor) and some RISC architectures (MIPS, SPARC)