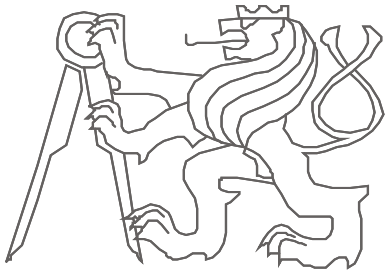
Architektury počítačů

Predikce skoků + Hyper-Threading



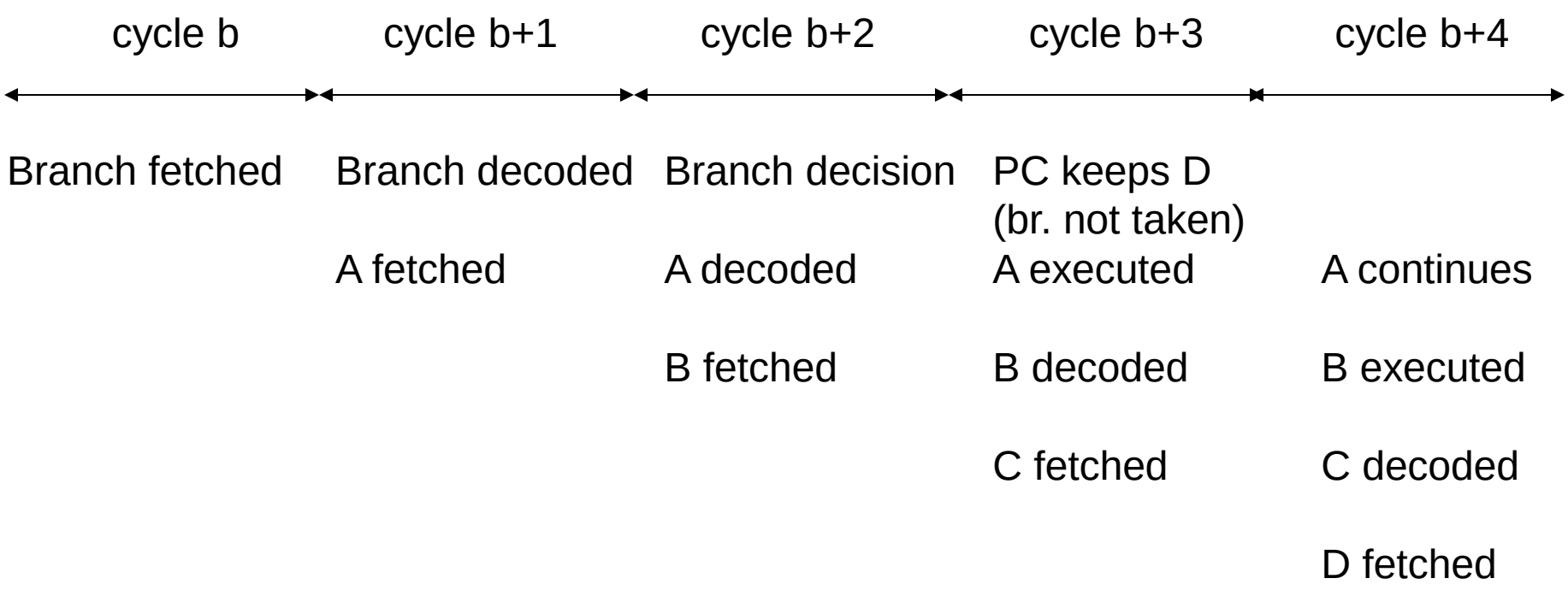
České vysoké učení technické, Fakulta elektrotechnická

Control Hazards

- Jump and Branch are great performance losses.
- **Jump instruction** needs only the jump target address
- Branch instruction requires 2 operations:
  - Branch Result — **Taken** or **Not Taken**
  - Branch Target Address
    - PC + 4 — If Branch is NOT Taken
    - PC + 4 + 4 × immediate — If Branch is Taken

Branch Not Taken

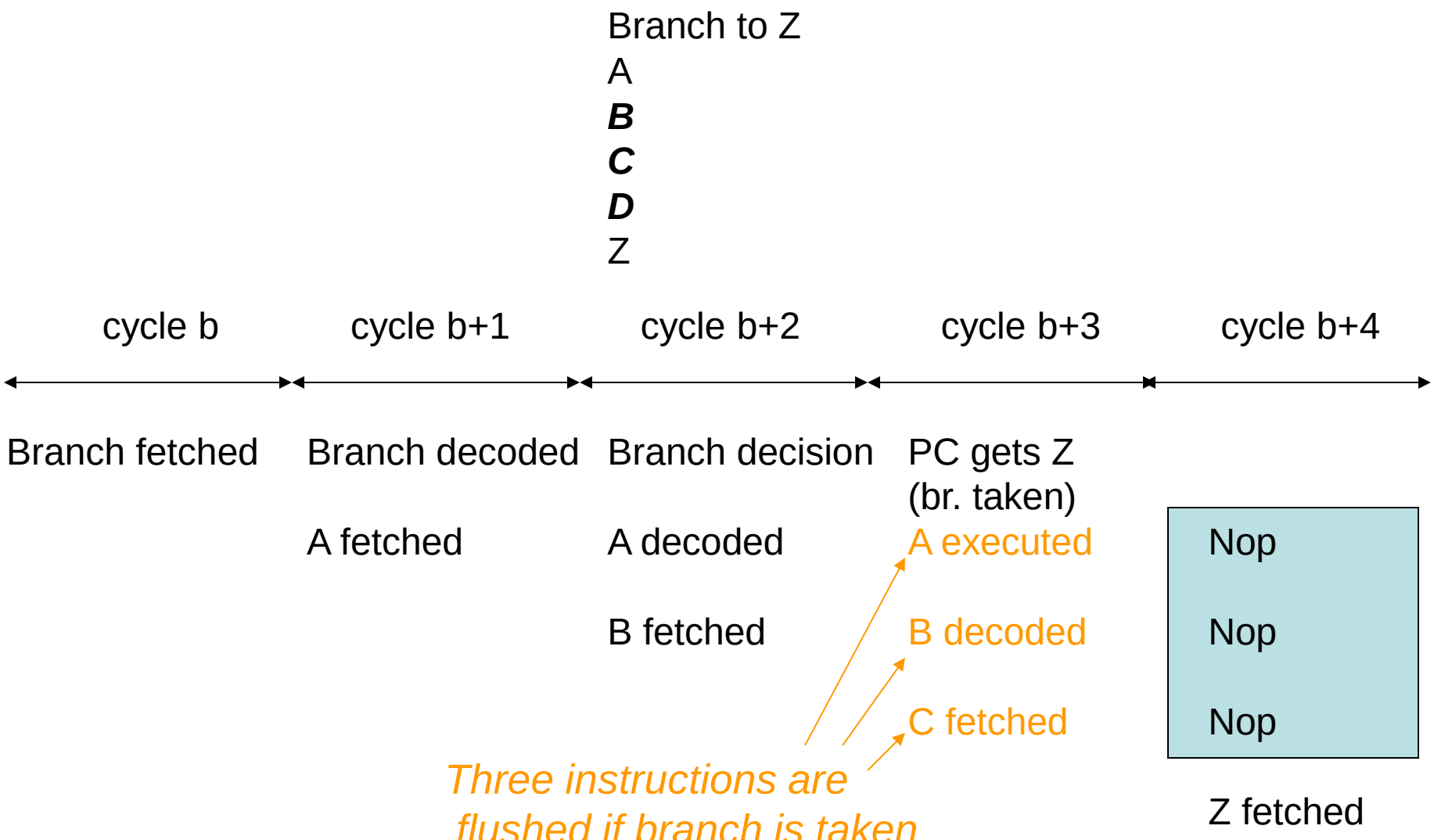
Branch to Z
A
B
C
D
Z



Branch Hazard

- Consider heuristic – branch **Not taken**.
- Continue fetching instructions in sequence following the branch instructions.
- If branch is taken (indicated by *zero* output of ALU):
  - Control generates *branch* signal in ID cycle.
  - *branch* activates *PCSource* signal in the MEM cycle to load PC with new branch address.
  - *Instructions in the pipeline must be flushed if branch is taken – can this penalty be reduced?*

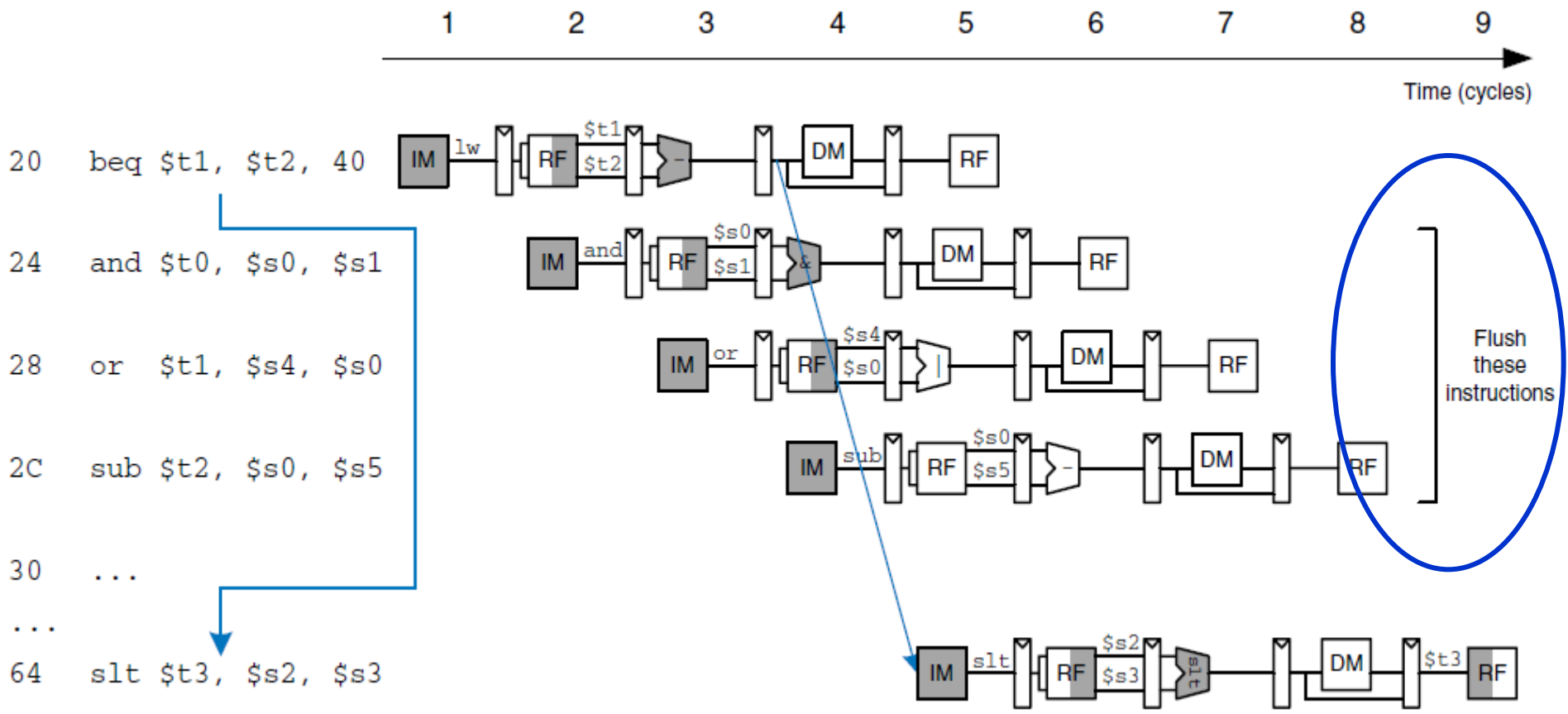
Branch Taken



Pipeline Flush

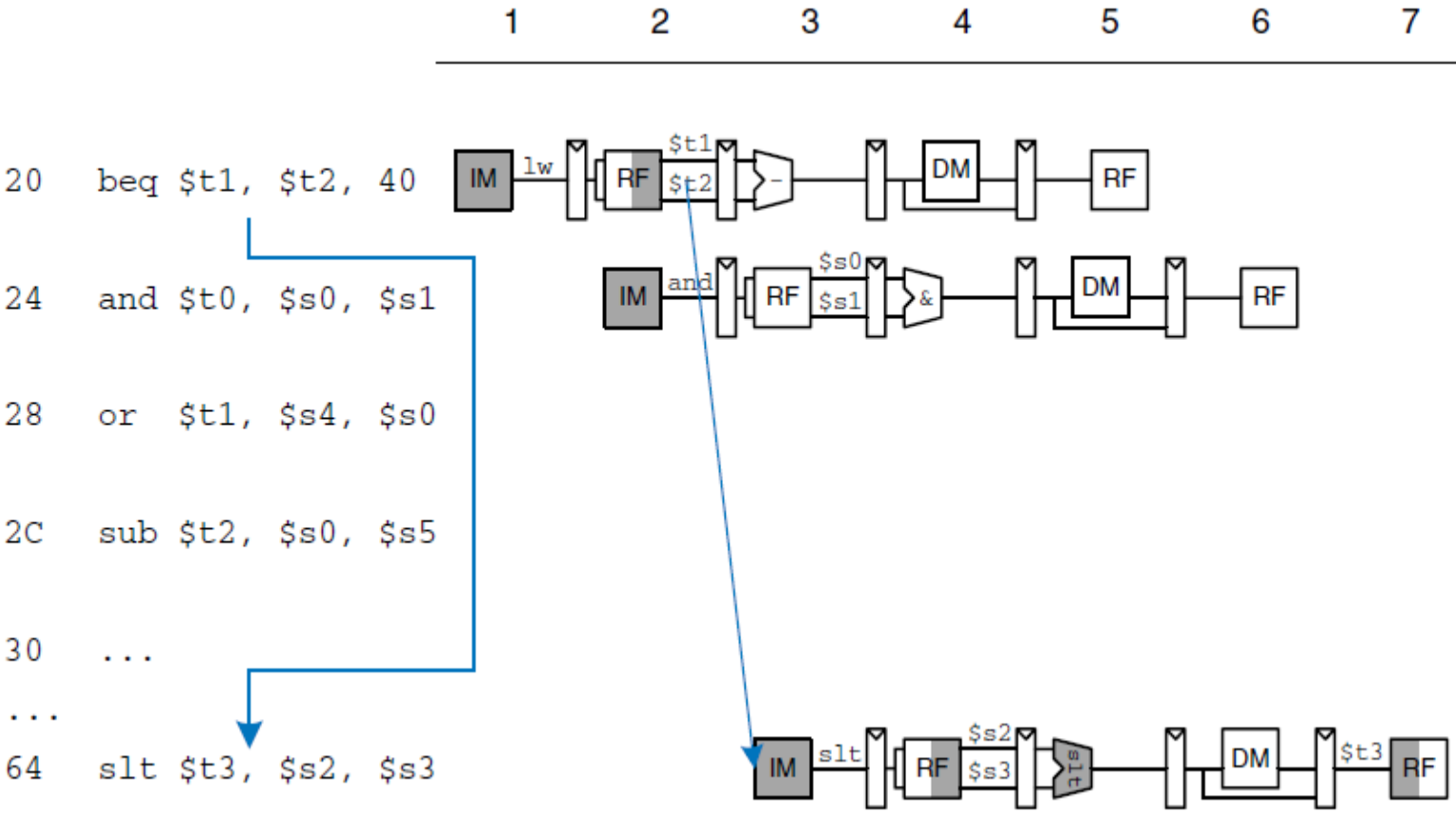
- If branch is taken (as indicated by *zero*), then control does the following:
  - Change all control signals to 0, similar to the case of stall for data hazard, i.e., insert bubble in the pipeline.
  - Generate a signal *IF.Flush* that changes the instruction in the pipeline register IF/ID to 0 (nop).
- Penalty of branch hazard is reduced by
  - Adding branch detection and address generation hardware in the decode cycle – **one bubble needed** – *a next address generation logic in the decode stage writes PC+4, branch address, or jump address into PC.*
  - Using branch prediction.

Řídicí hazardy



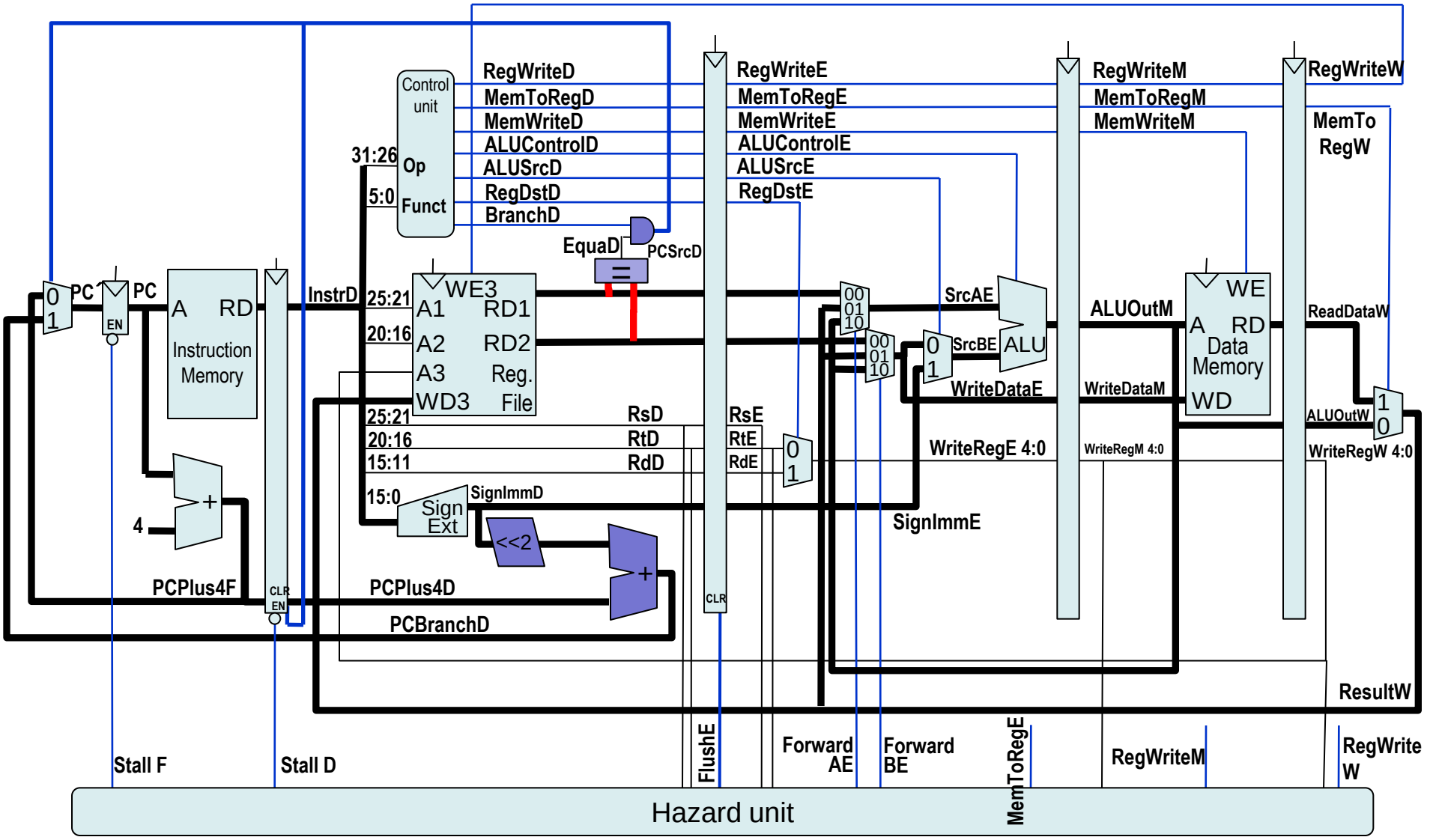
- výsledek porovnání je znám až v 4. cyklu.. Proč?

Řídicí hazardy – raději znát výsledek dříve..



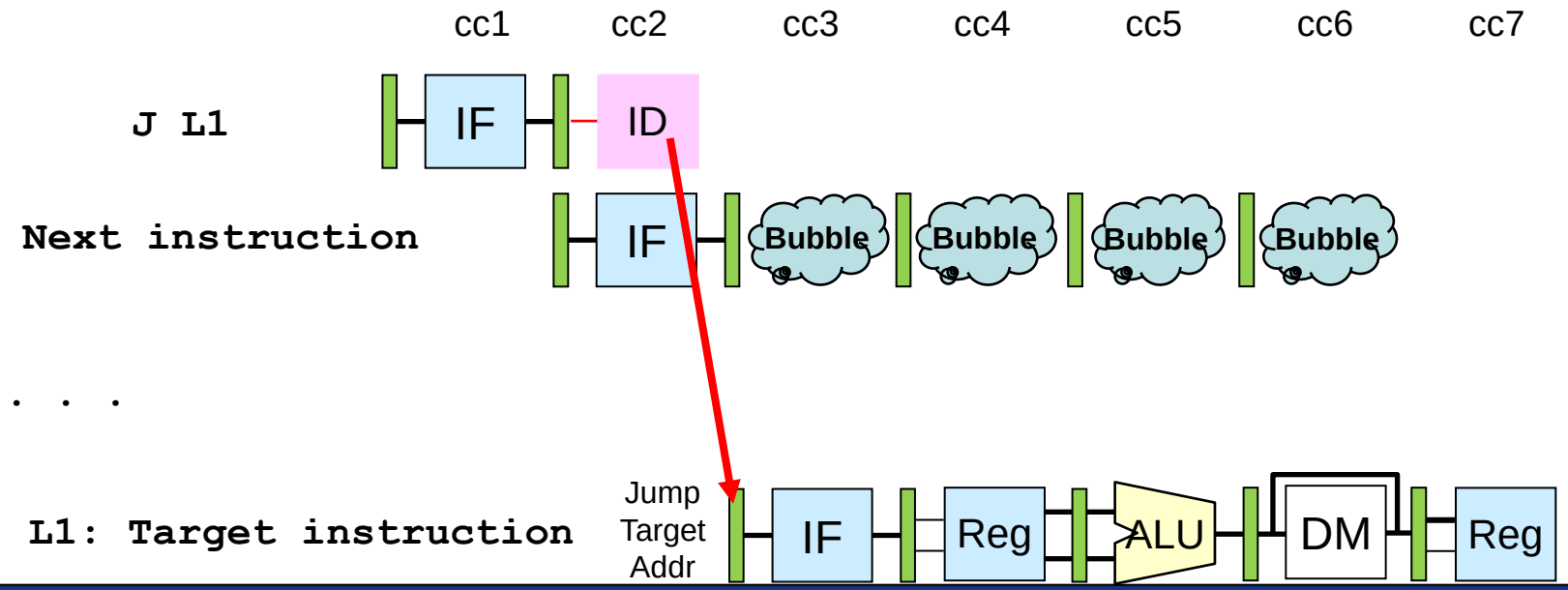
- pokud dokážeme stanovit výsledek porovnání už v 2. cyklu, můžeme redukovat tzv. „misprediction penalty“
- přesun rozhodování dopředu může zavést nové RAW hazardy..!!!

Řešení řídicích hazardů vyprázdněním (flush)

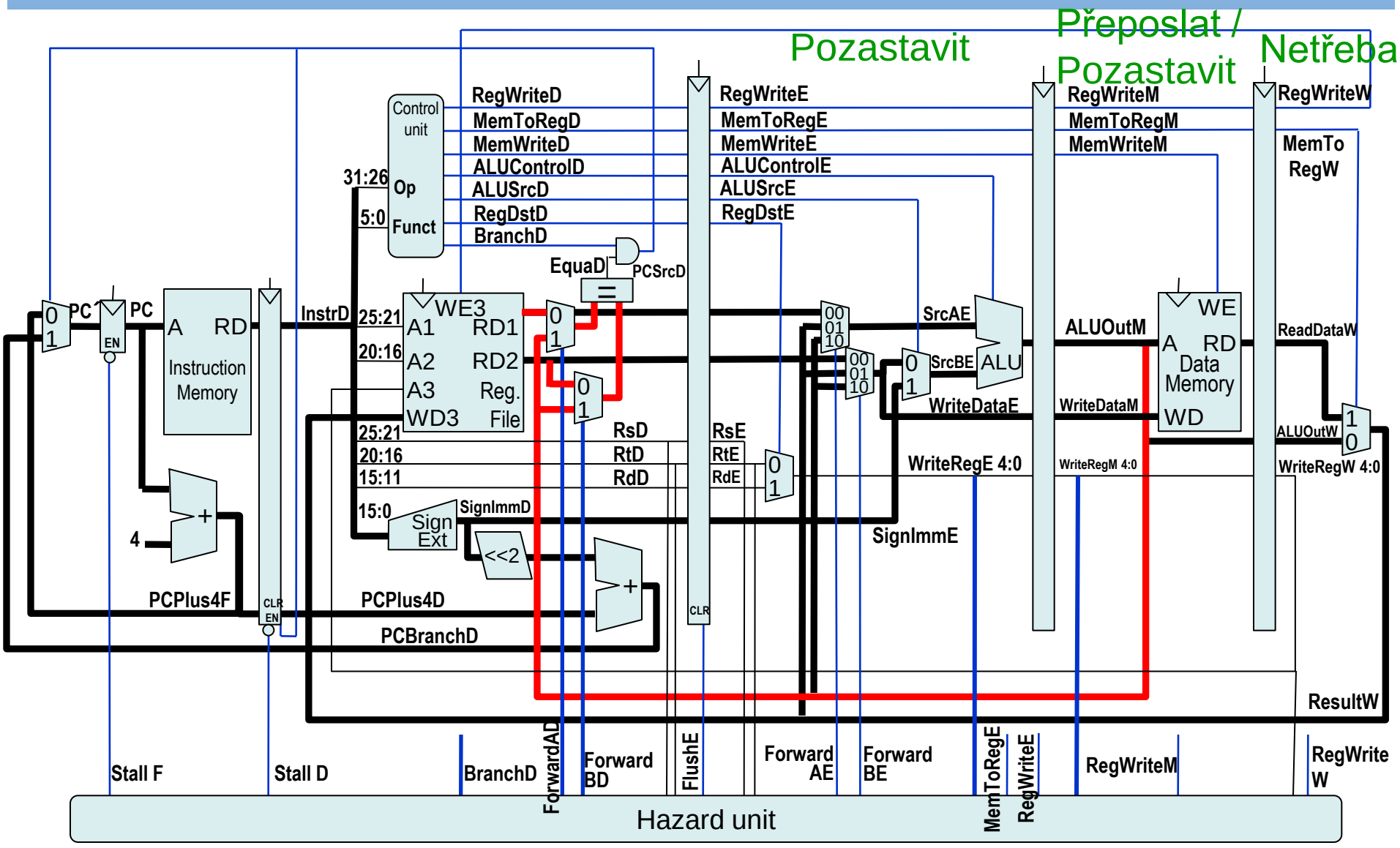


1-Cycle Jump Delay

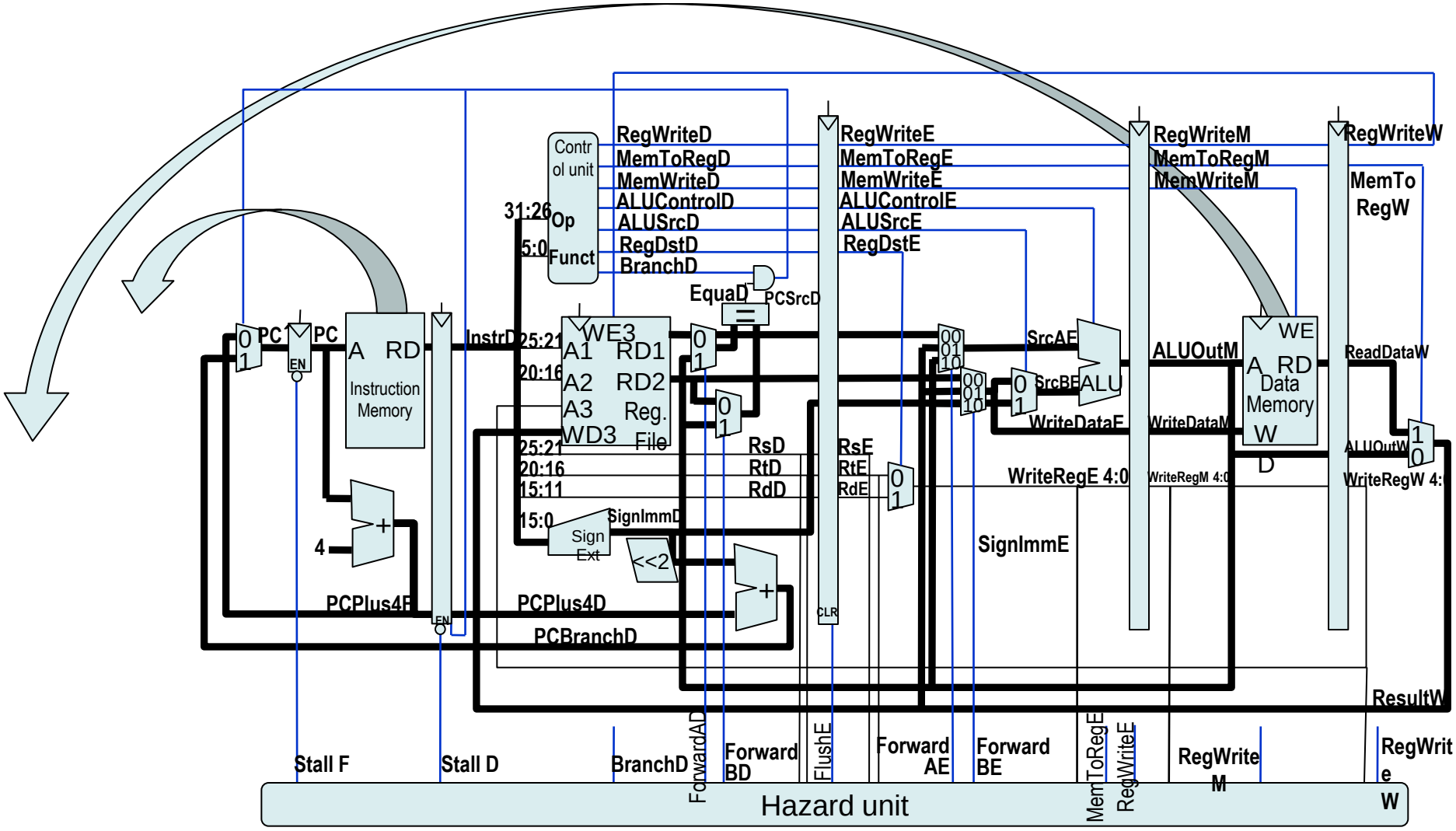
- If the control logic detects a Jump instruction in the 2nd Stage, then Next instruction is fetched anyway.
- We flush only with one instruction.



Řešení vzniklých RAW hazardů přeposíláním nebo pozastavením

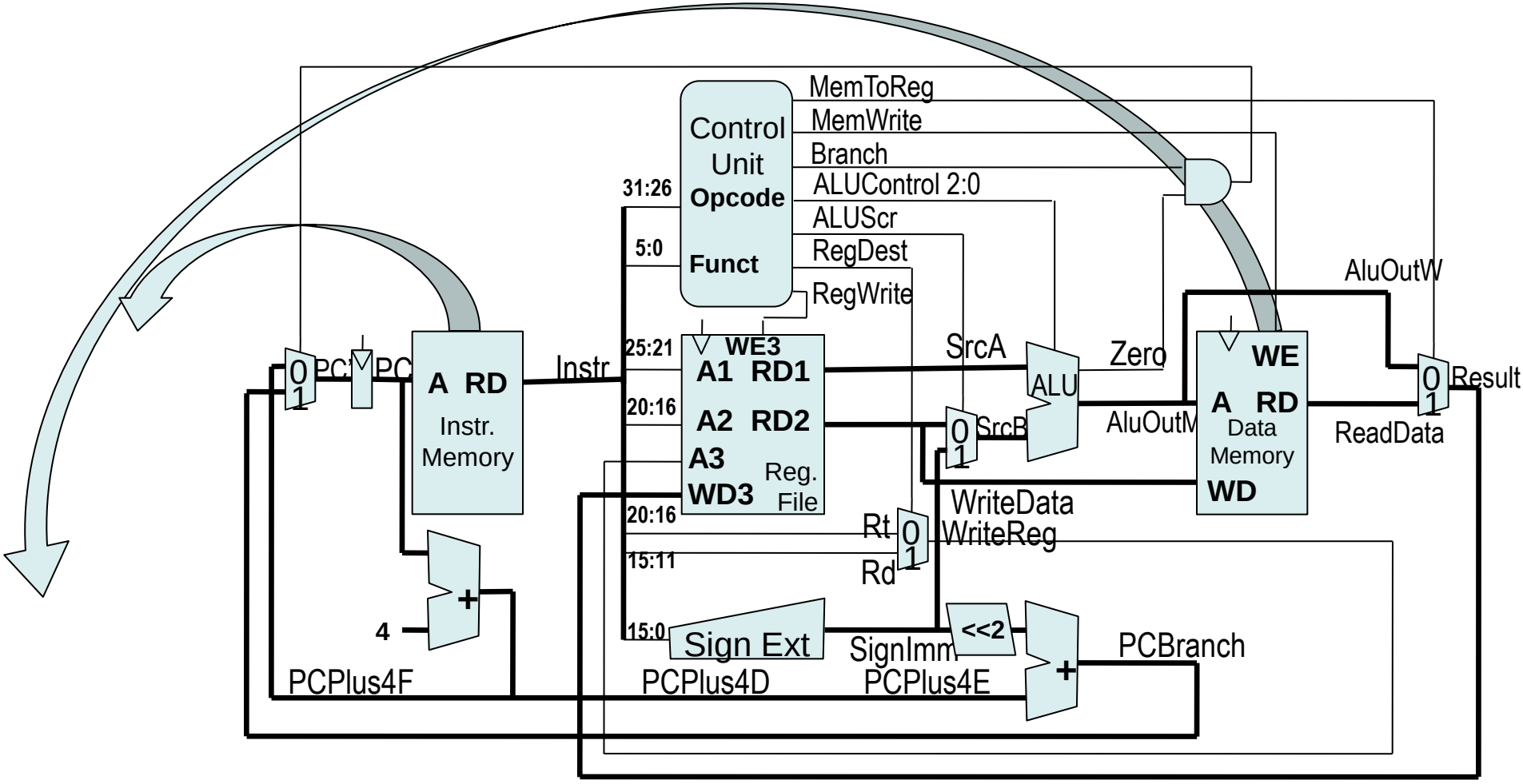


Co jsme navrhli? – zřetězená verze



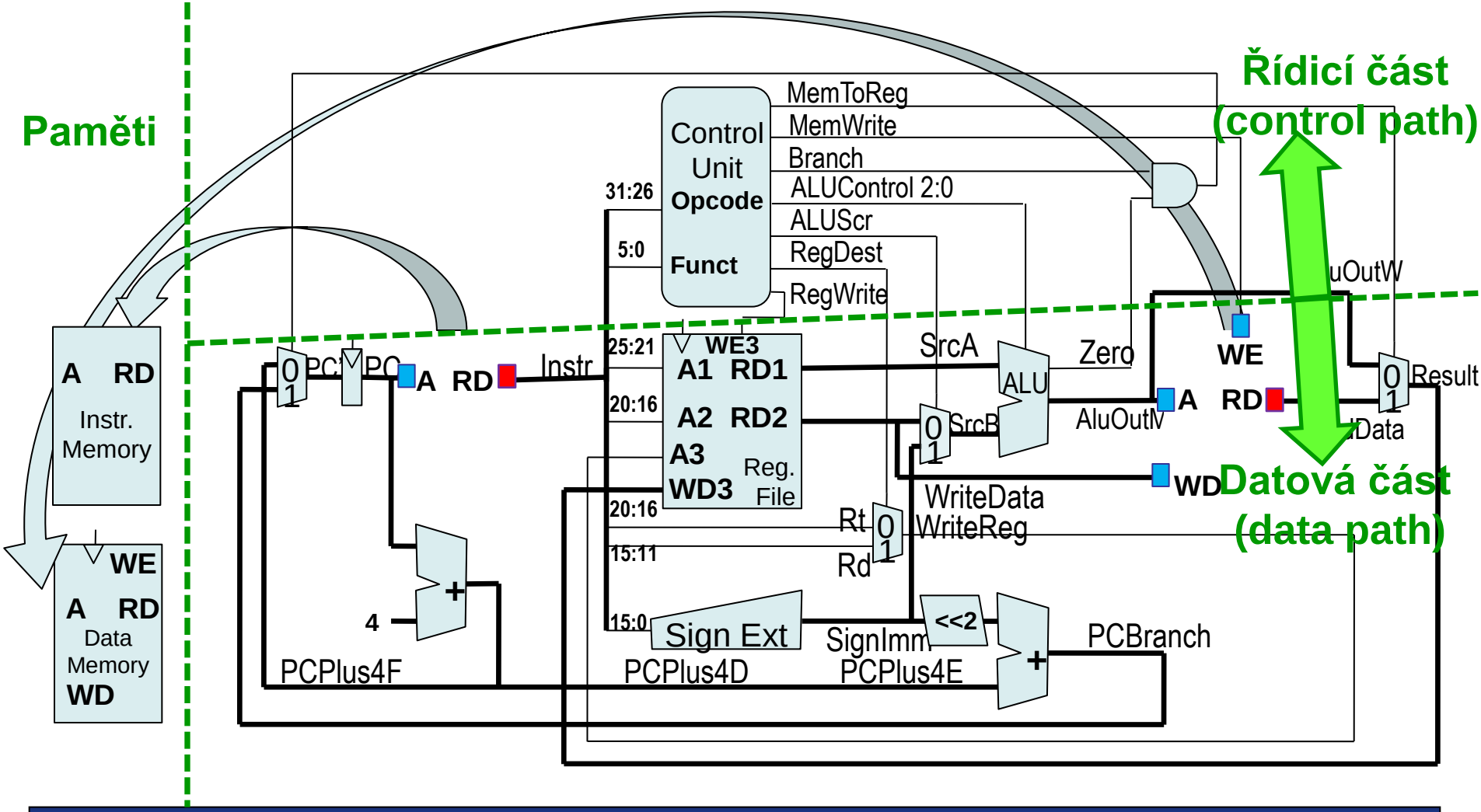
Co jsme navrhli?

Návrat k nezřetězenému procesoru

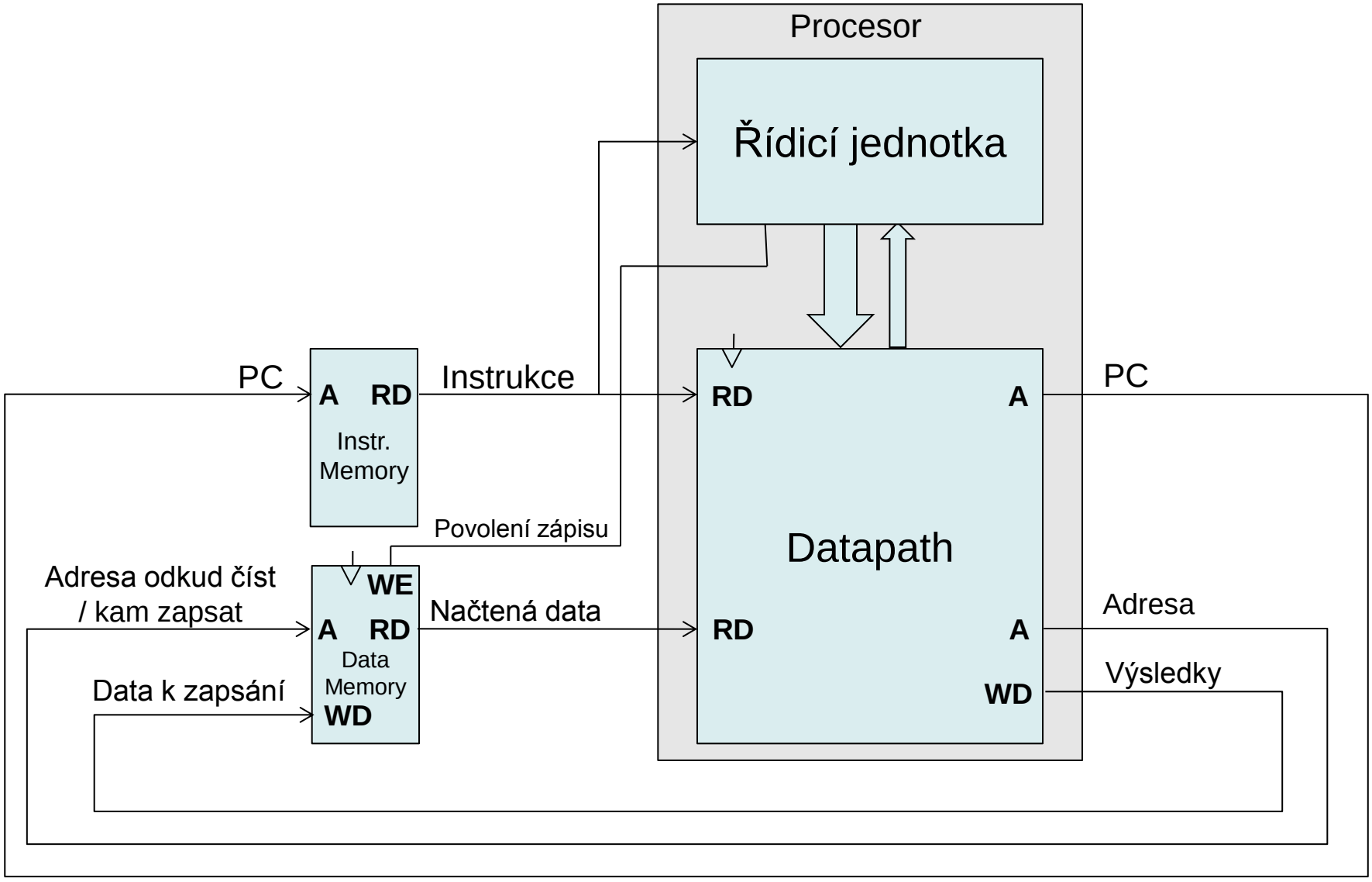


Co jsme navrhli?

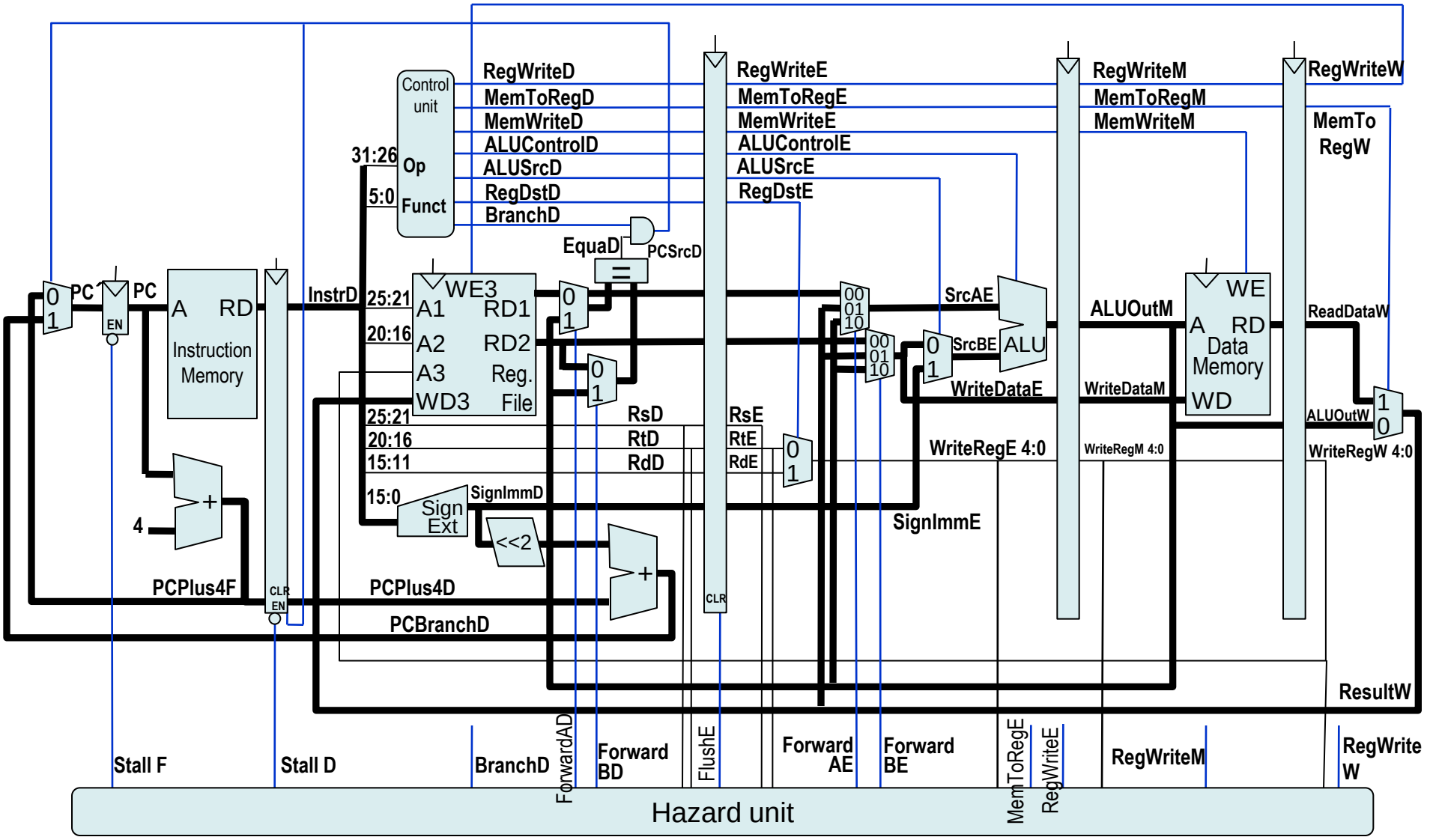
Návrat k nezřetězenému procesoru



Co jsme navrhli?



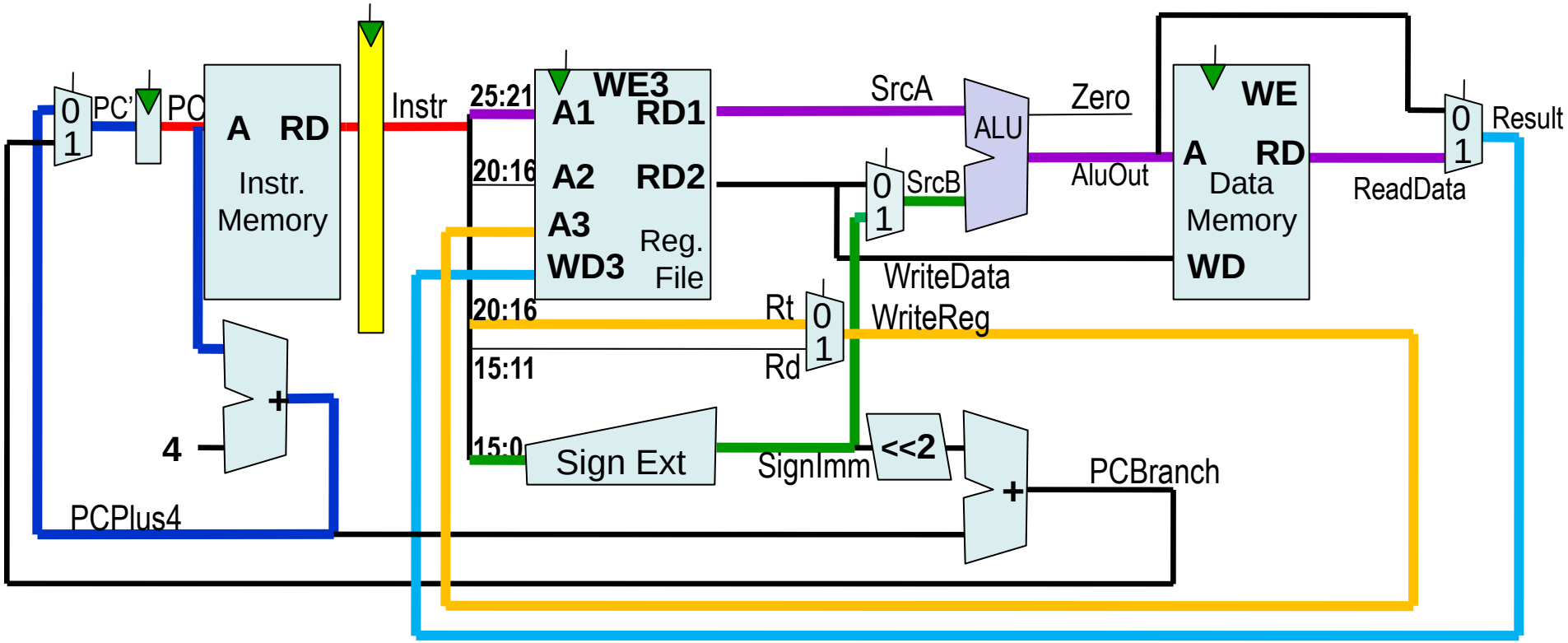
Hotovo – navržený zřetězený procesor



Opakování: Jedno-cyklový procesor

- Jaká může být maximální frekvence procesoru?
- Zpoždění na kritické cestě – instrukce `lw`:

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



Opakování: Jedno-cyklový procesor – výkon

- Předpokládejme:

t_{PC}	= 30 ns	t_{Mem}	= 300 ns
t_{RFread}	= 50 ns	t_{ALU}	= 200 ns
t_{Mux}	= 20 ns	$t_{RFsetup}$	= 20 ns

Při T_{fetch} prováděném paralelně s $T_{processor}$,
jelikož je vždy 30+300=T_{fetch} < $T_{processo}$ = 50+200+300+20+20 = 590 ns = 1.69 MHz ->
IPS = 1 690 000 [Instructions per second]

Zřetězený procesor – výkon

Má-li zřetězený procesor

T_c dobu taktu

P počet stupňů pipeline

N označuje počet instrukcí v programu, pak

$$\mathbf{T_{program} = (P + (N-1)) * T_c}$$

protože 1. instrukce potřebuje P taktů k naplnění pipeline,
ale každá další instrukce přidá už pouze 1 takt navíc.

Zřetězený procesor – výkon: IPS = IC / T

Dobu cyklu určuje nejpomalejší stupeň

- V našem případě jde o paměť:
  T_{men} = 300 ns --> T_{cmin} = 300 ns --> 3 333 kHz
- Neuvažujme-li stavy stall a flush pipeline,
  pak lze říct, že program s větším počtem instrukcí N
  vykoná jednu instrukci za jeden cyklus.
  IPS = $1/T_{cmin}$ = 3333333 instrukcí za sekundu
- Zavedením 5-stupňového zřetězení jsme zlepšili propustnost
  3 333333/ **1 690 000** = 1,97 = ~2 krát!

Proč tak málo? Naše jednoduchá pětistavová pipeline příliš závisí na době přístupu do paměti.

Predikce skoků

Benchtests of Branch Statistics

- Branches occur every **4-7 instructions** on average in integer programs, commercial and desktop applications; somewhat less frequently in scientific ones :-)
- **Unconditional** branches : approx. **20%** (of branches)
- **Conditional** branches approx. **80%** (of branches)
  - **66%** is forward. Most of them (~60%) are often **Not Taken**.
  - **33%** is backward. Almost all of them are **Taken**.

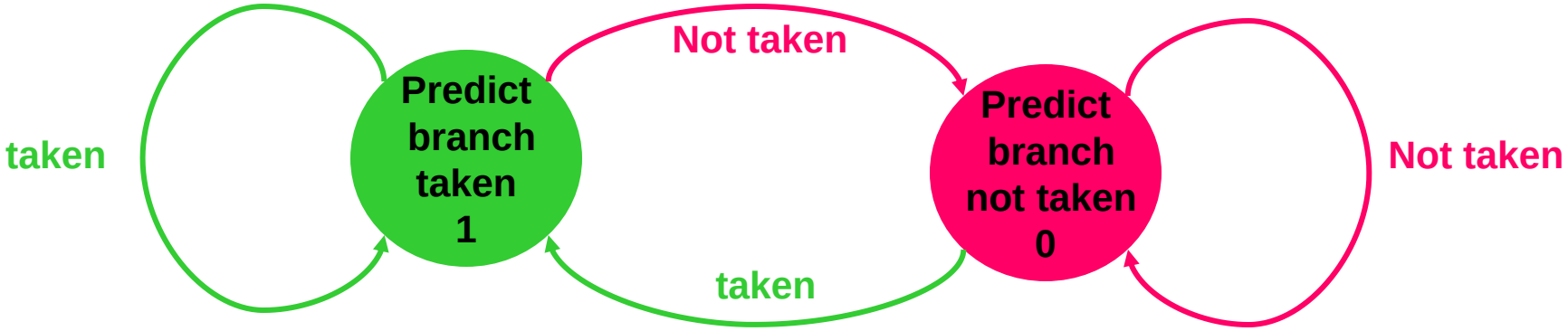
- **We can estimate the probability that a branch is taken**
  $p_{taken} = 0.2 + 0.8* (0.66 * 0.4 + 0.33) = 0.67$

  In fact, many simulations show that p_{taken} is **from 60 to 70%**.

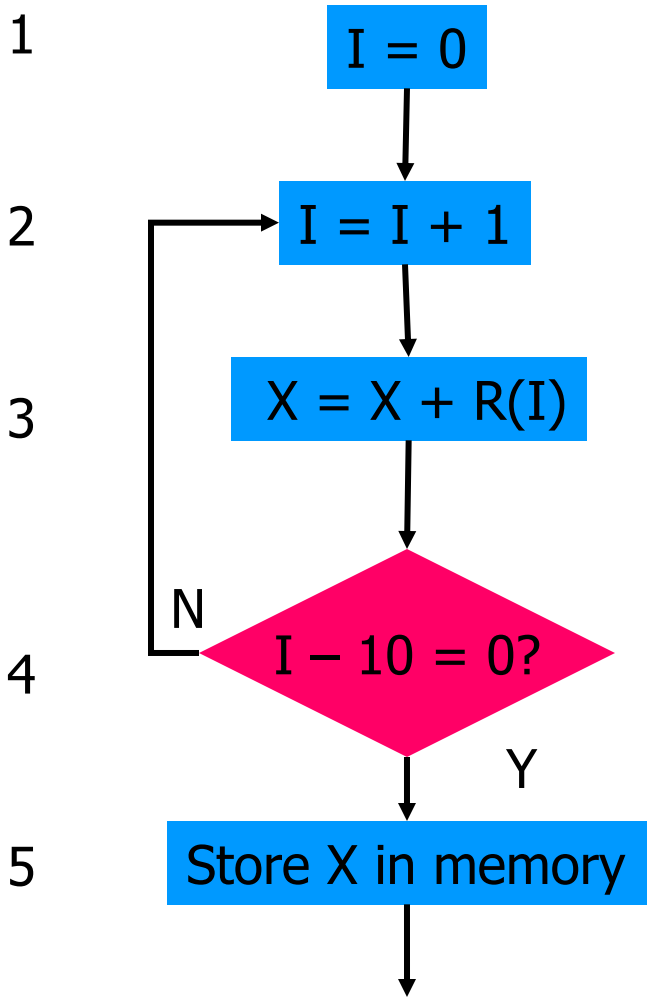
See: Lizy Kurian John, Lieven Eeckhout:
Performance Evaluation and Benchmarking, *CRC Press 2018*

One-bit Branch Prediction

- A one-bit prediction scheme:
  a one “history bit” tells what happened on the last branch instruction execution:
  - History bit = 1, branch was previously **Taken**
  - History bit = 0, branch was previously **Not taken**



Branch Prediction for a Loop

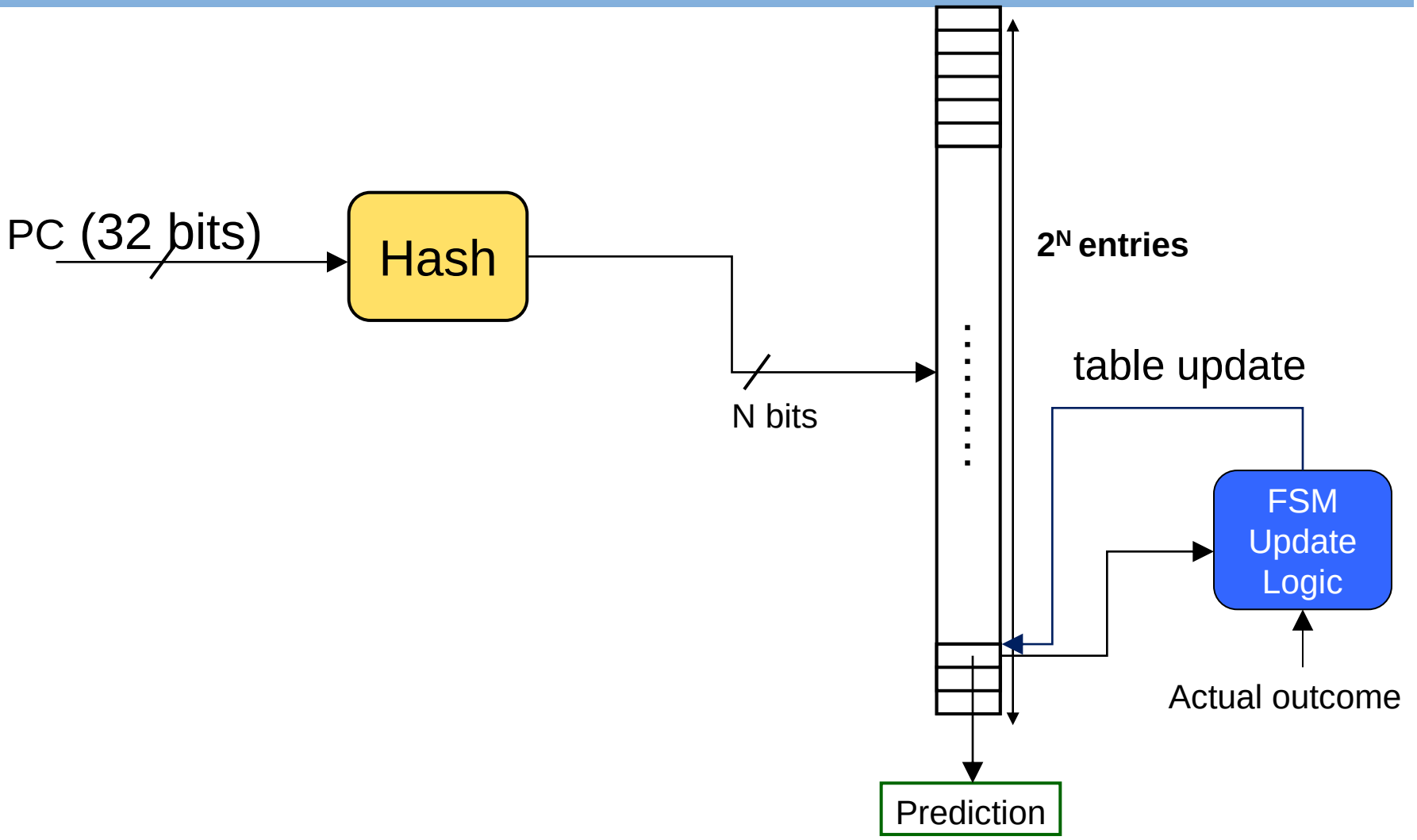


Execution of Instruction 4

Execu-tion seq.	Old hist. bit	Next instr.			New hist. bit	Predi ction
		Pred.	I	Act.		
1	0	5	1	2	1	Bad
2	1	2	2	2	1	Good
3	1	2	3	2	1	Good
4	1	2	4	2	1	Good
5	1	2	5	2	1	Good
6	1	2	6	2	1	Good
7	1	2	7	2	1	Good
8	1	2	8	2	1	Good
9	1	2	9	2	1	Good
10	1	2	10	5	0	Bad

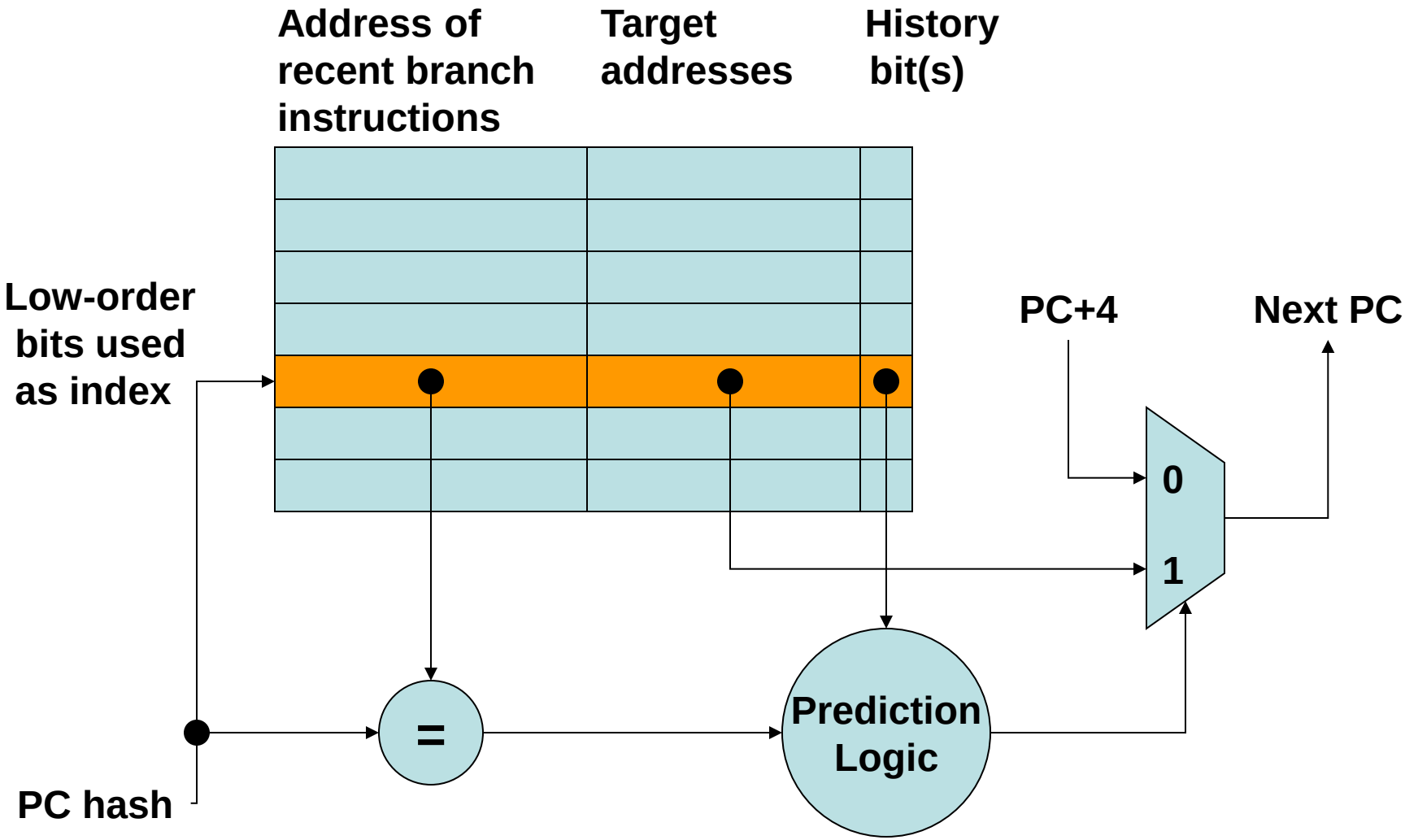
bit = 0 *branch not taken*, bit = 1 *branch taken*.

Typical Organization of Branch Prediction Table



Note: FSM - Finite State Machine (cz: konečný automat)

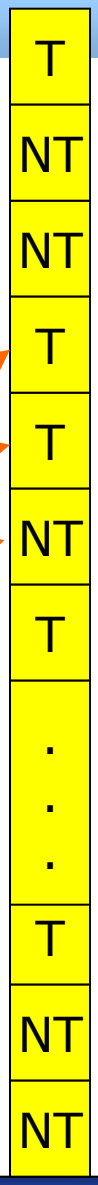
Branch Prediction



Simplest Dynamic Branch Predictor

```
for (i=0; i<100; i++)
  {  if (arr[i] == 0) {  … }
       …
  }
```

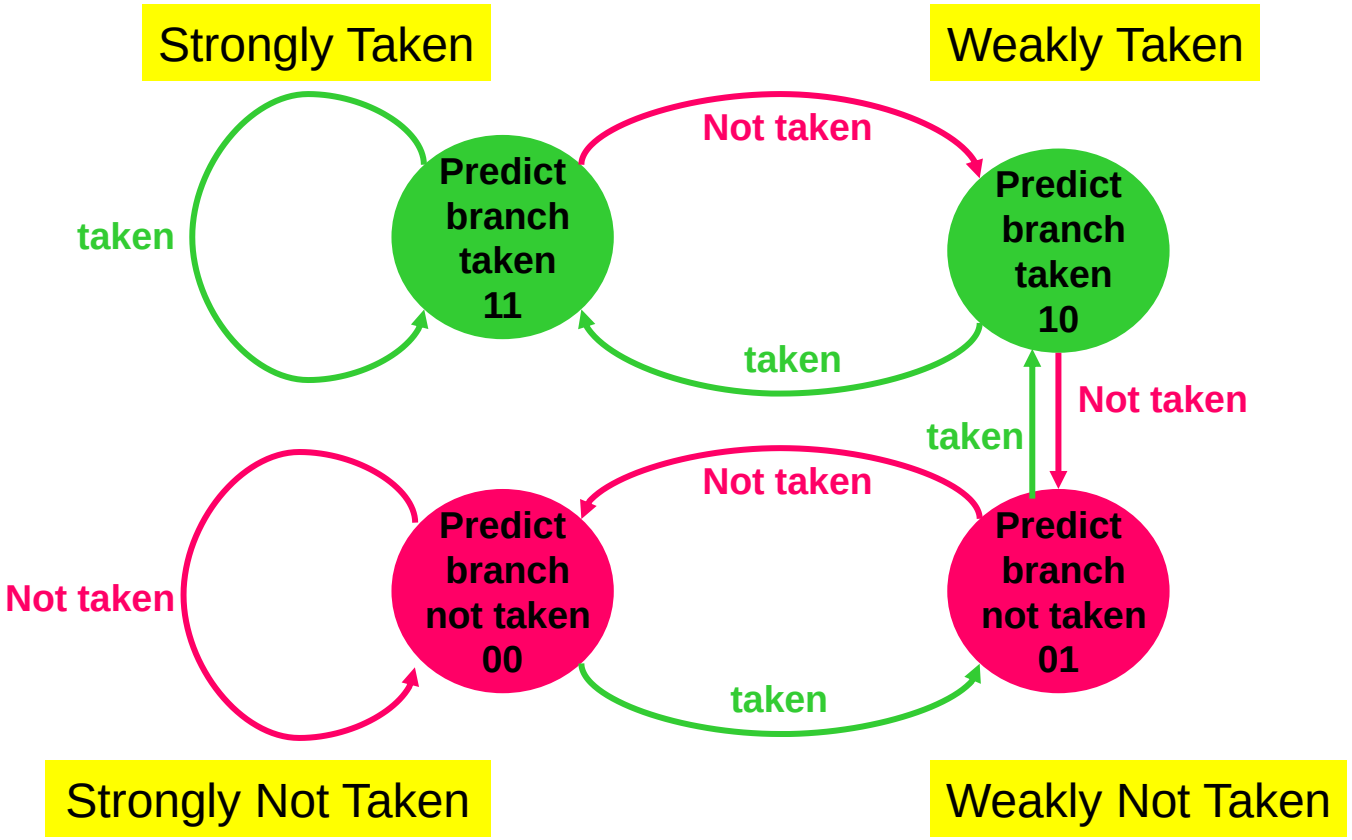
```
 0x400100F8      la $18, arr
 0x400100FC      addi  $10, $0, 100
 0x40010100      or  $1,  $0,  $0
Loop1:
 0x40010104      sll     $3, $1, 2
 0x40010108      add     $19, $18, $3
 0x4001010c      lw      $2, ($19)
 0x400102 10     beq     $2, $0, Loop2
 … …

 0x400102 14     beq     $0, $0, Loop3
Loop2:
      … … …
Loop3:
 0x40010B08      addi    $1, $1,   1
 0x40010B 0c     bne     $1, $10, Loop1
```



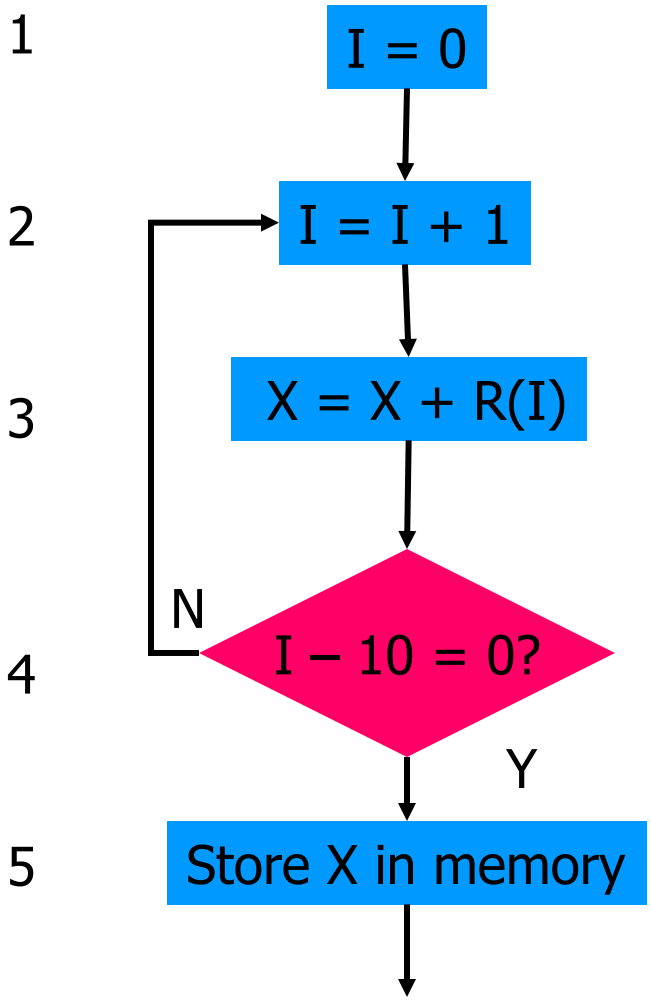
1-bit Branch History Table

Two-Bit Prediction Buffer Type I

- It is called 2-bit saturating counter. This one has no hysteresis.



Branch Prediction for a Loop

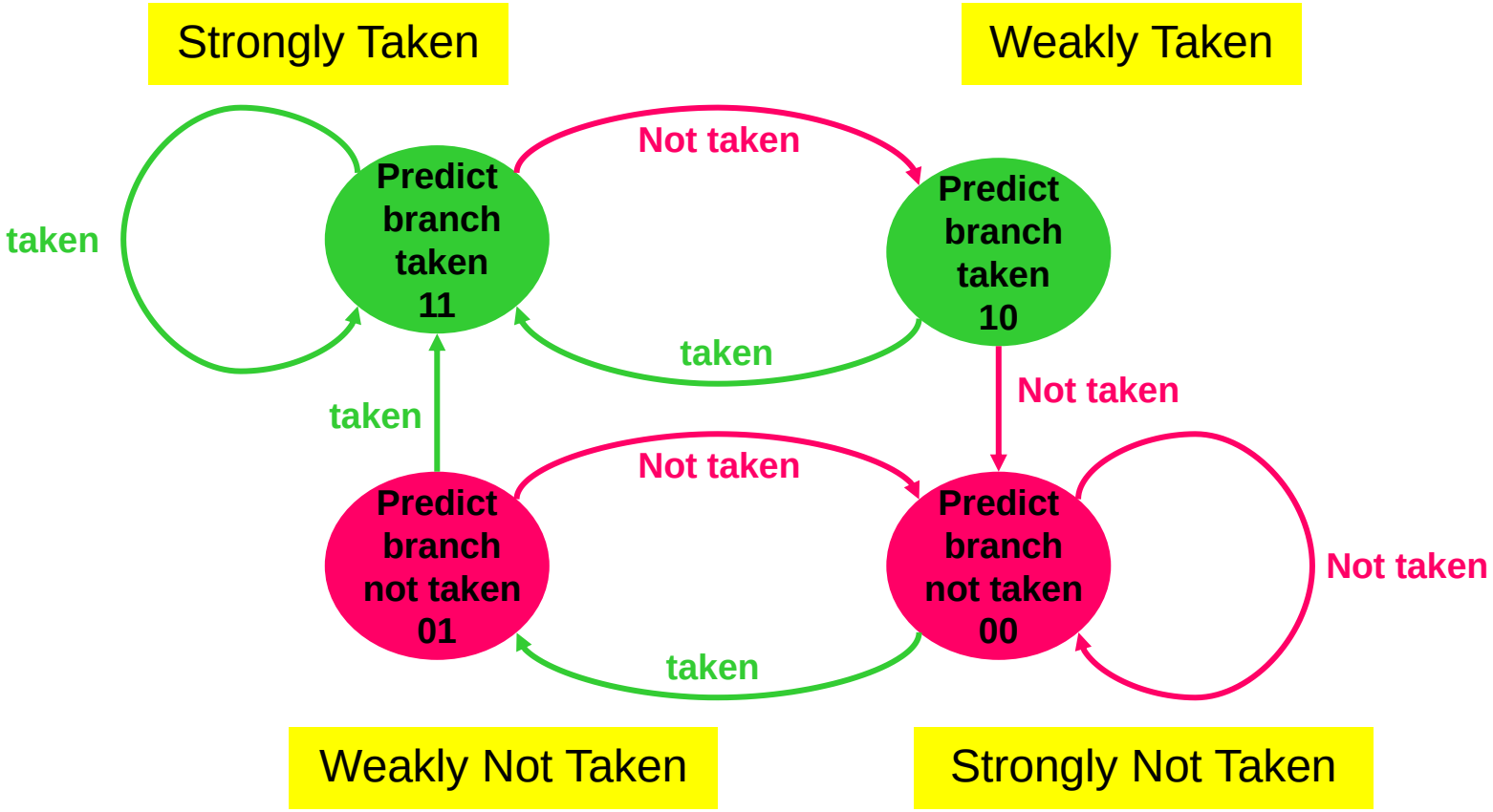


Execution of Instruction 4

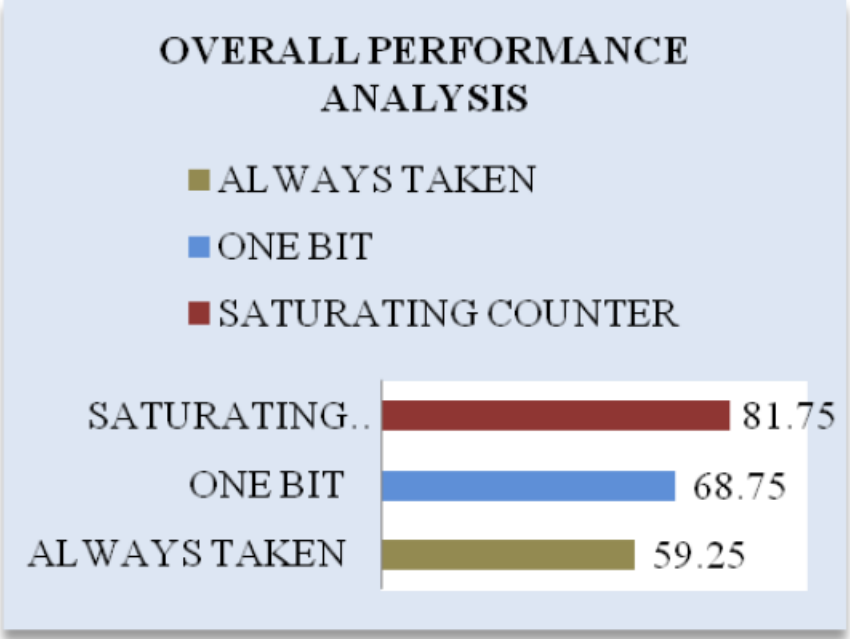
Execu-tion seq.	Old Pred. Buf	Next instr.			New pred. Buf	Predi ction
		Pred.	I	Act.		
1	10	2	1	2	11	Good
2	11	2	2	2	11	Good
3	11	2	3	2	11	Good
4	11	2	4	2	11	Good
5	11	2	5	2	11	Good
6	11	2	6	2	11	Good
7	11	2	7	2	11	Good
8	11	2	8	2	11	Good
9	11	2	9	2	11	Good
10	11	2	10	5	10	Bad

Two-Bit Prediction Buffer Type II.

This 2-bit saturating counter was modified by adding hysteresis.
Prediction must miss twice before it is changed.



Some result of benchtest



Here, a higher number means the better prediction

Source: https://ieeexplore.ieee.org/document/6918861
H. Arora, S. Kotecha and R. Samyal, "Dynamic Branch Prediction Modeller for RISC Architecture," *2013 International Conference on Machine Intelligence and Research Advancement*, Katra, 2013, pp. 397-401.

Note: This study has used saturating counter with hysteresis (type II).

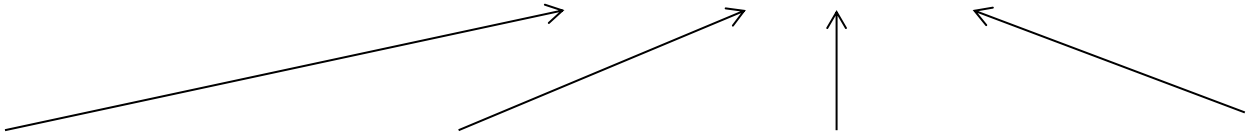
Correlating Predictors

We can look at other branches for clues

```
if (x==2)                 // branch b1
                          …
if (y==2)                 // branch b2
                          …
if(x!=y)  { … }           // branch b3 depends on the
                             results of b1 and b2
```

(2,1) Correlated predictor

We use 4 predictors: **P00 | P01 | P10 | P11**



P00
This predictor is used if the previous 2 branches in the program have both status **Not taken**.

P01
This predictor is used if the previous 2 branches have history: 2nd last branch **Not taken**, and the last branch **Taken**

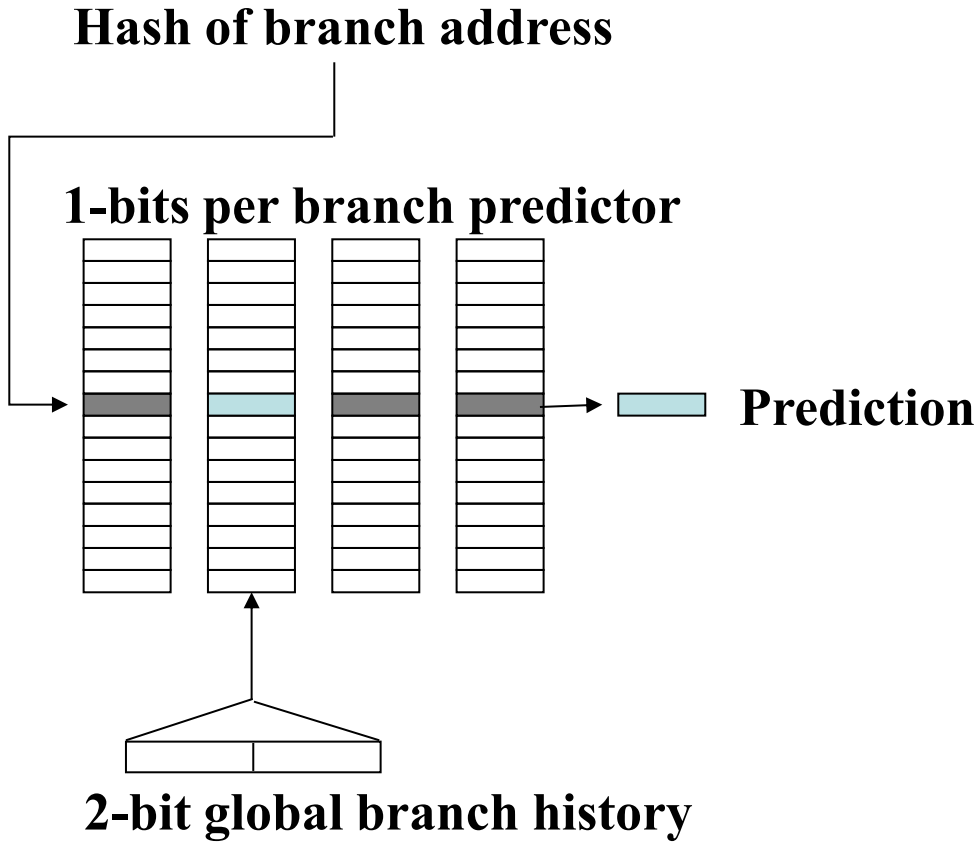
P10
This predictor is used if the previous 2 branches have history: 2nd last branch **Taken**, and the last branch **Not taken.**

P11
This predictor is used if the previous 2 branches in the program have both status **Taken**.

A (2,1) correlated branch predictor
- (**2**,**1**) means 2^{2} =4 predictors buffers each contains **1** bit
- and uses the behavior of the last **2** branches to choose from 2^{2} predictors.

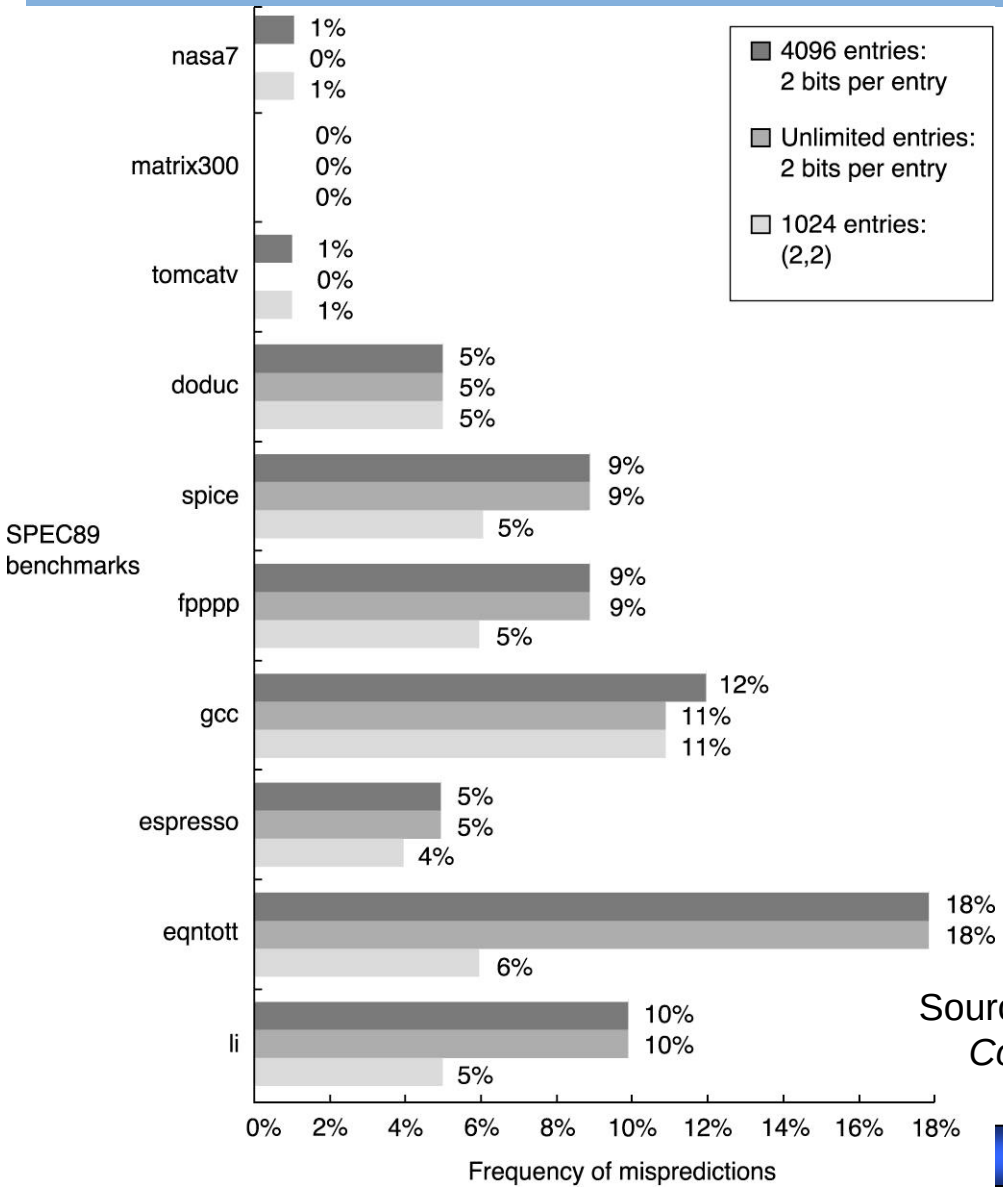
Correlating Predictors

- Example (2,1) predictor



- 2 bits of global history means that we look at T/NT behavior of last 2 branches to determine the behavior of THIS branch.
- The buffer can be implemented as an one dimensional array.
- (**m**,n) predictor uses behavior of last **m** branches to choose from 2^{m} predictor each of them is n-bit predictor.

Correlating Predictors in SPEC89



Note: **SPEC89** is older SPEC CPU benchmark suite that is nowadays replaced by newer sets. It contained:

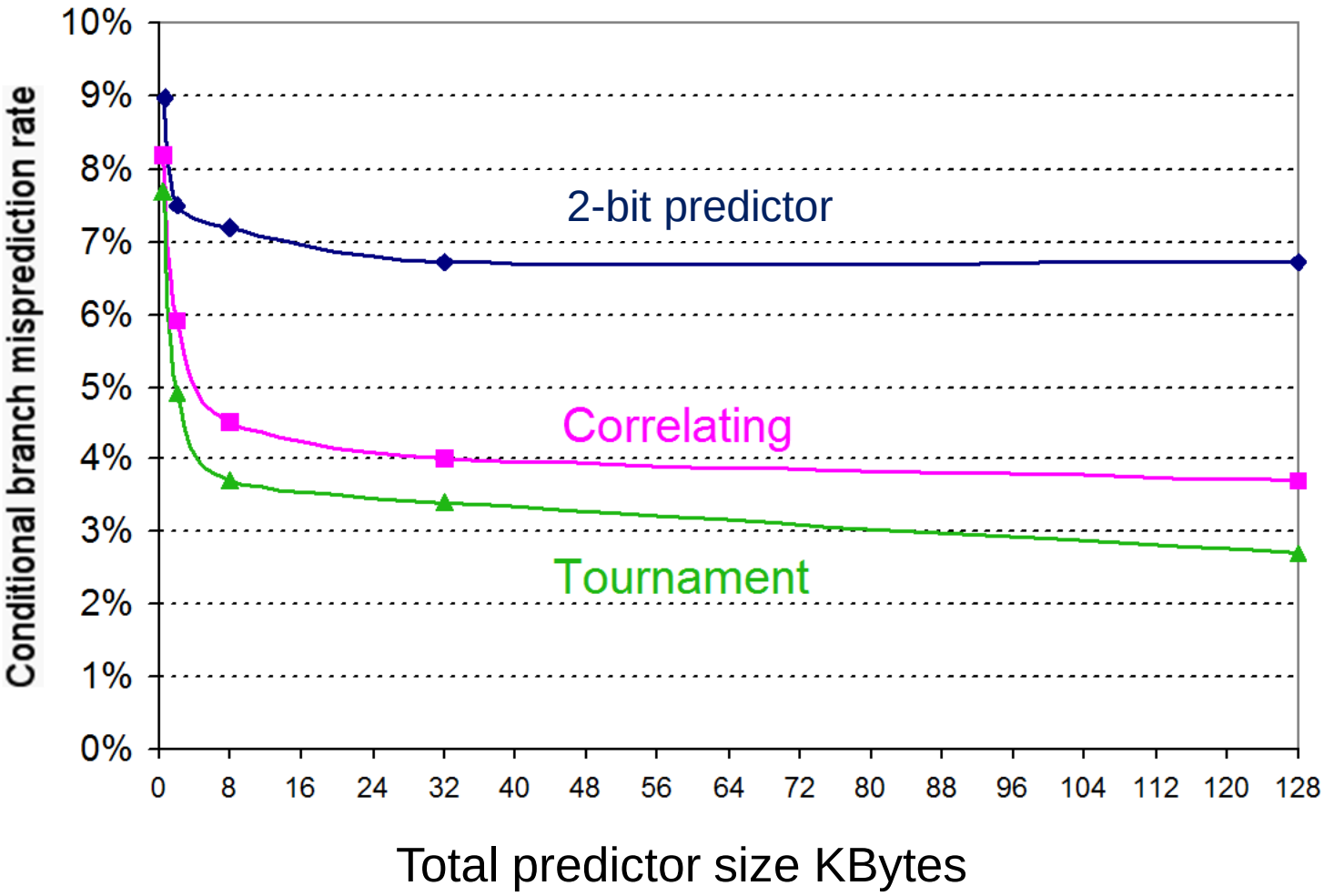
- **gcc** INT1 GNU C compiler
- **espresso** INT PLA optimizing tool
- **spice2g6** FP2 Circuit simulation and analysis
- **doduc** FP Monte Carlo simulation
- **nasa7** FP Seven floating-point kernels
- **li** INT LISP interpreter
- **eqntott** INT Conversions of equations to truth table
- **matrix300** FP Matrix solutions
- **fpppp** FP Quantum chemistry application
- **tomcatv** FP Mesh generation application

Source of picture: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.*

Tournament Predictors

- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance was improved.
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information *(local branch was taken, not taken)*, and combine them with a selector.
- They use n-bit saturating counter to choose between predictors.
- Hopes to select right predictor for right branch.

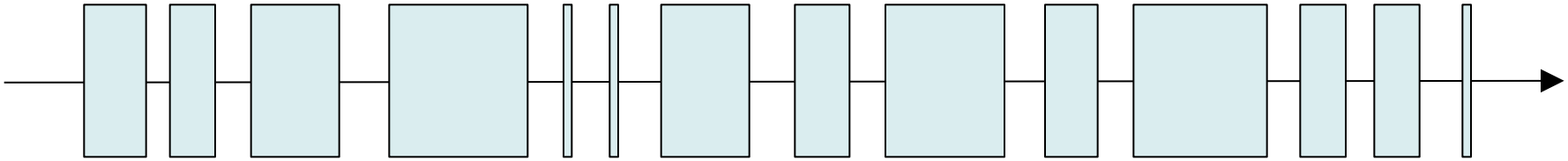
Benchtest of Accurancy



More pipeline steps

Vyvažování stupňů zřetězení

Lineární zřetězení:

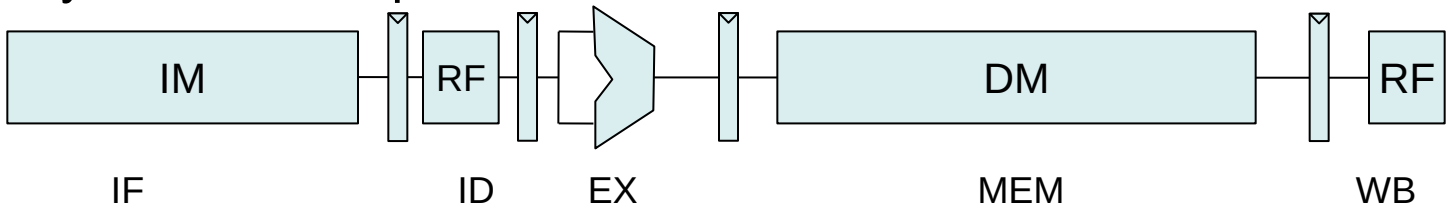


(též: používá se též ve stromovém sumátoru, násobičce, iterační děličce..)

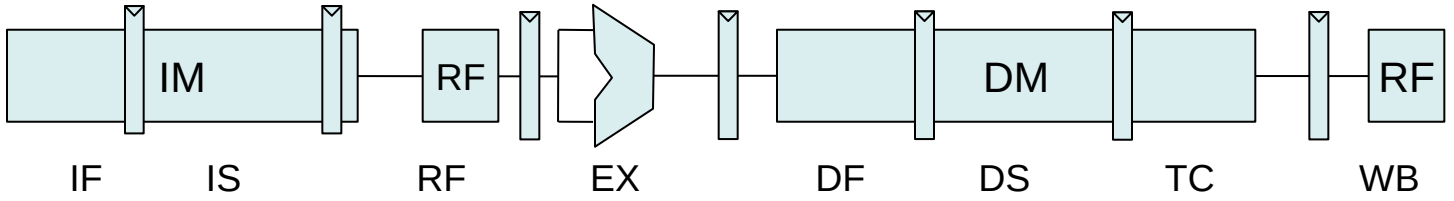
- **Vyvažování**: cílem je rozdělit jednotlivé bloky do *N* stupňů tak, aby ve zpoždění ve všech stupních bylo pokud možno stejné…
- Volba počtu stupňů závisí od preference: propustnost vs. latence

Superzřetězení

- nevyvážené 5-stupňové zřetězení:



- hlubší zřetězení vzniklá další dekompozicí, přináší možnost dalšího zvýšení pracovní frekvence, avšak také řadu dalších problémů jako další forwarding, nárůst pozastavení pipeline, hazardy a nárůst ceny chybné predikce skou.



IF	First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
IS	Second half of instruction fetch, complete instruction cache access.
RF	Instruction decode and register fetch, hazard checking, and instruction cache hit detection.
DF	Data fetch, first half of data cache access.
DS	Second half of data fetch, completion of data cache access.
TC	Tag check, to determine whether the data cache access hit.

A small example how to **Avoid Branches**

On web, you can found out many tricks suitable for time critical loops. This example present how to calculate absolute value of 32 bit signed integer **x** without branches.

Code with *unpredictable branch* dependable on data

C code	**MIPS if x in $2**	**Comment**
if(x<0) x=-x;	slt $1, $2, $0	// tmp = x<0 ? 1 : 0
	beq $1, $0, Skip1	// if(tmp==0) goto Skip
	nop	// delay slot
	sub $2, $0, $2	// x = - x;
Skip1:	…	

Fast C code	**MIPS if x in $2**	**Comment**
int tmp = x>>31;	sra $1, $2, 31	// tmp = x<0 ? -1 : 0
x ^= tmp;	xor $2, $2, $1	// 1st compliment of x, if tmp=-1
x -= tmp;	sub $2, $2, $1	// add 1 if tmp = 1

Note: On MIPS with static prediction, we save just 1 instruction. If we compile the C code for an Intel processor with longer pipeline, then a branch miss-prediction is more expensive.

Jaká je délka zřetězení? – orientačně…

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**

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

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

- The Optimum Pipeline Depth for a Microprocessor: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4333&rep=rep1&type=pdf

What are Dynamic multiple-issue processors aka Superscalar processors ?

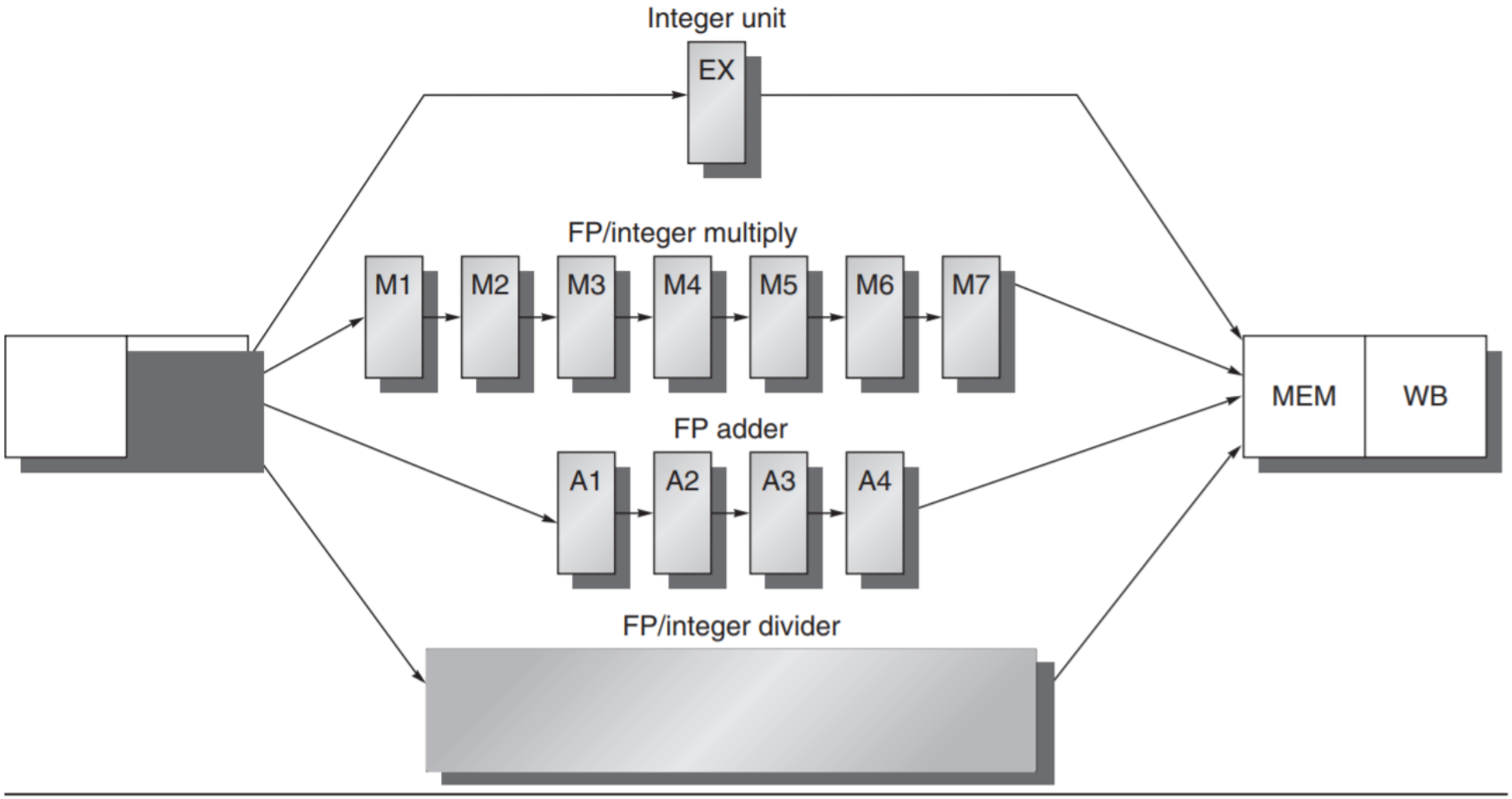
Definition

Wiki:

- *In contrast to a **scalar processor** that can execute at most **one single instruction per clock cycle**, a **superscalar processor** can execute **more than one** instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor.*

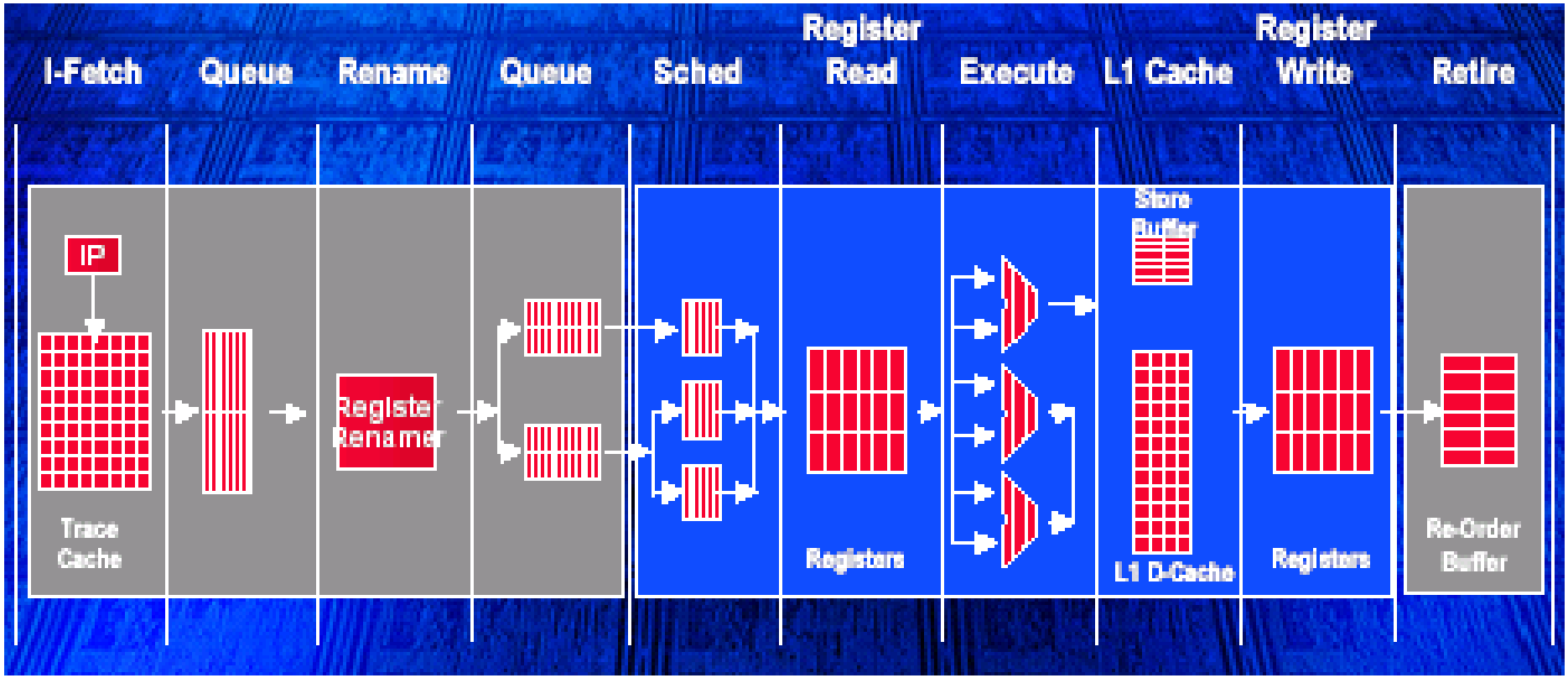
Q: What does it actually mean "more than one"?

A pipeline that supports multiple outstanding FP operations



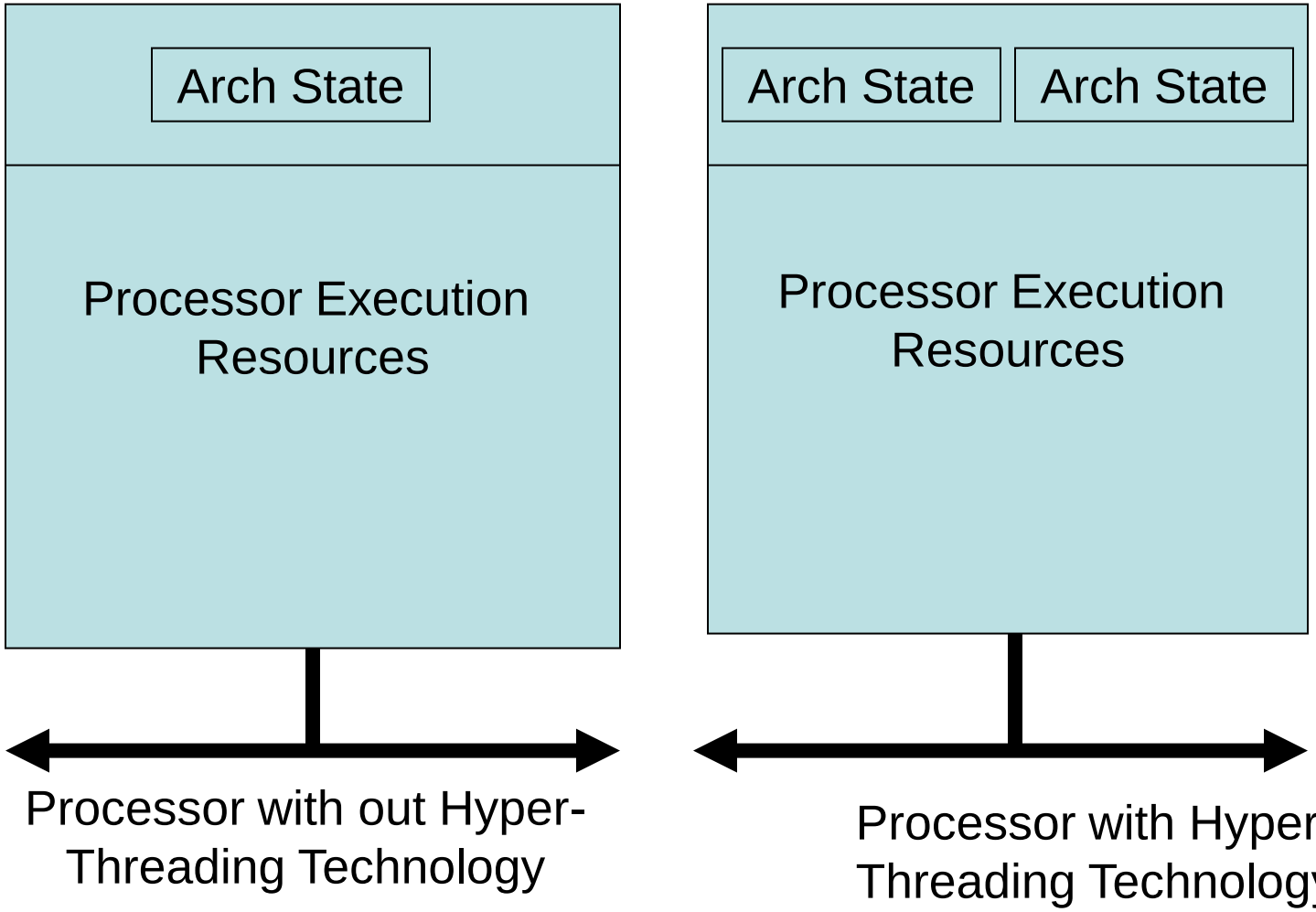
Source of picture: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.*

Pentium 4 - Out-of-order Execution pipeline



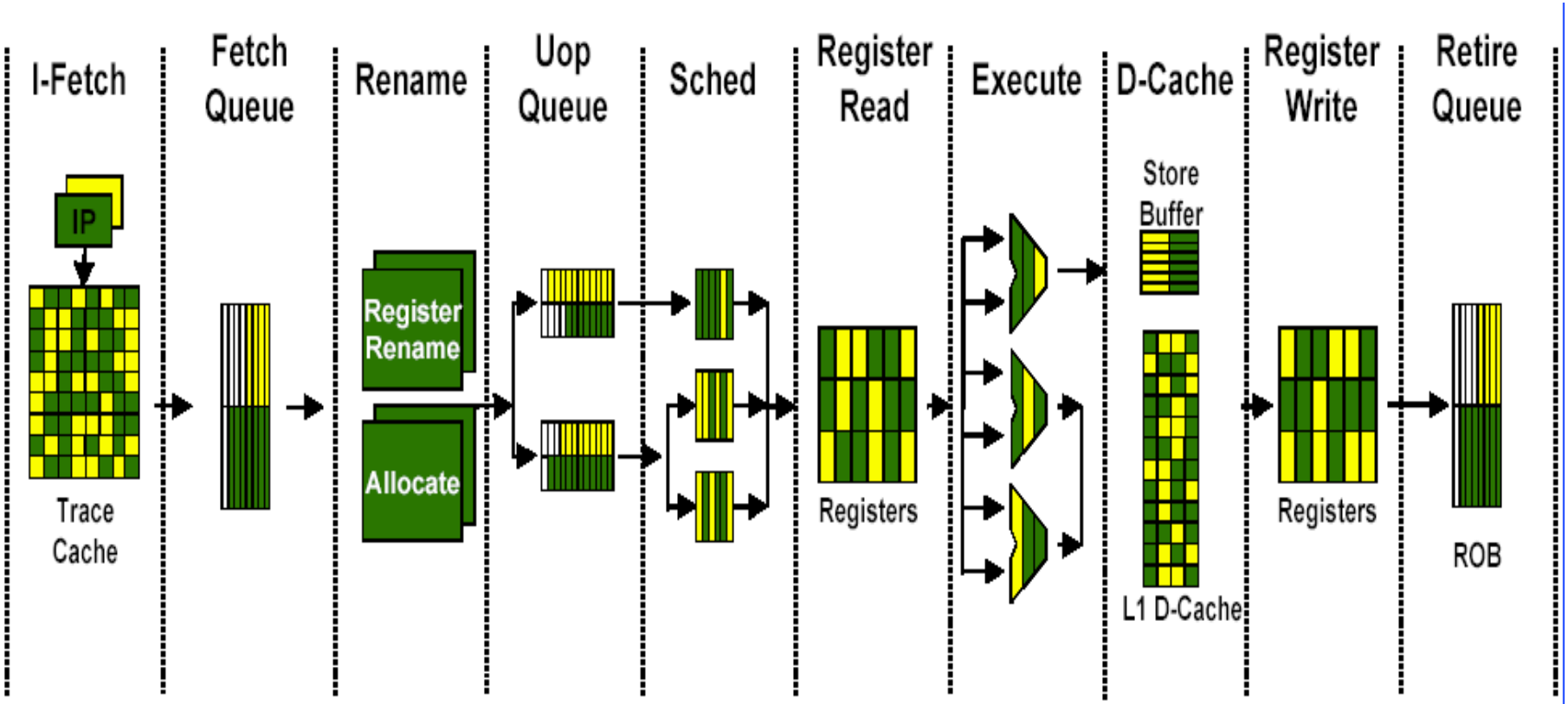
[Source: Intel]

Hyper-Threading



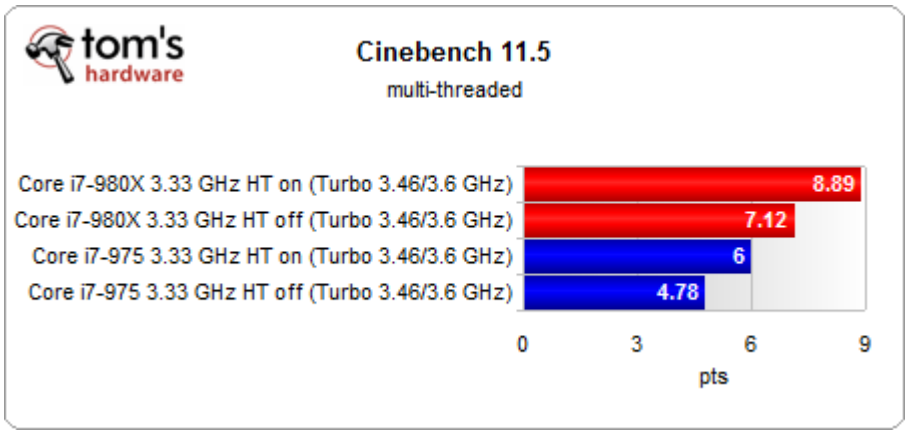
Ref: Intel Technology Journal, Volume 06 Issue 01, February 14, 2002

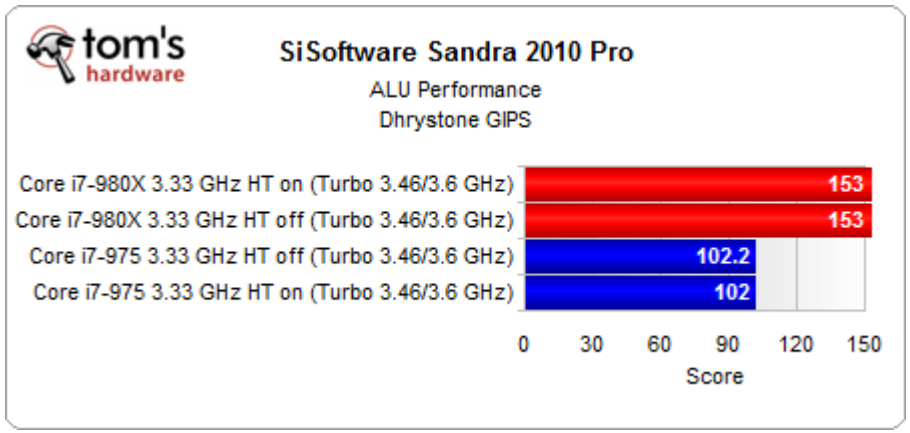
Pentium 4: Netburst Microarchitecture's execution pipeline



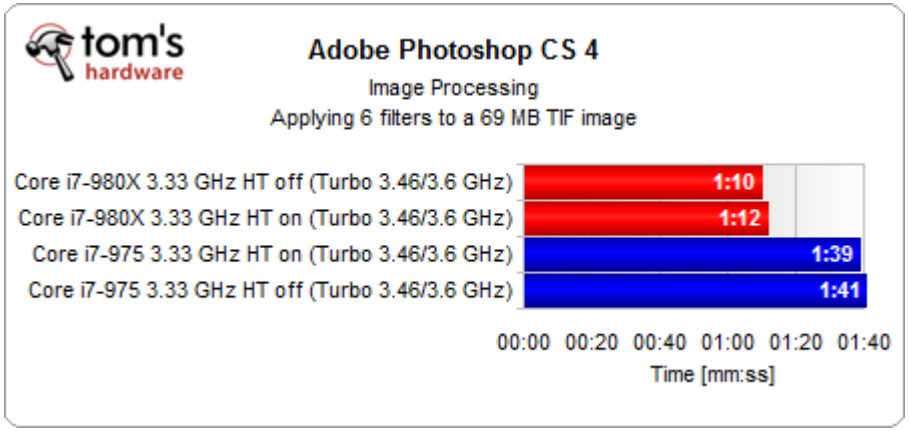
Picture is simplified because the pipeline has actually 20 steps.
The branch miss prediction penalty is here extremely high.

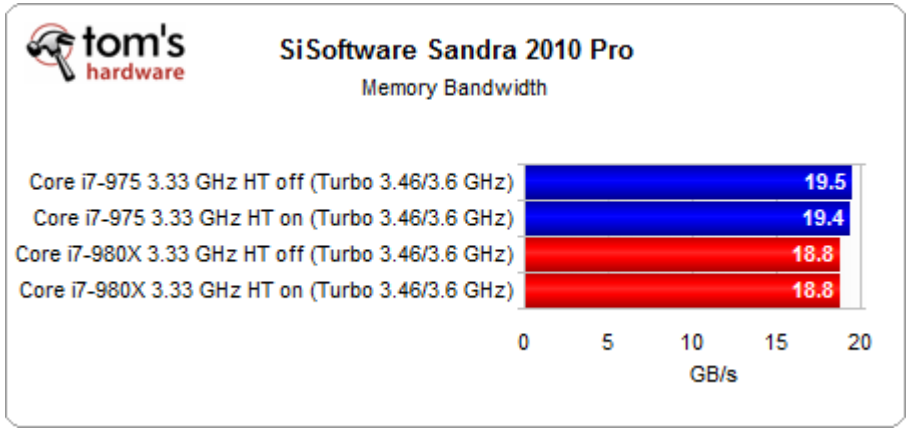
Sample from: Hyper-Threading Benchtest



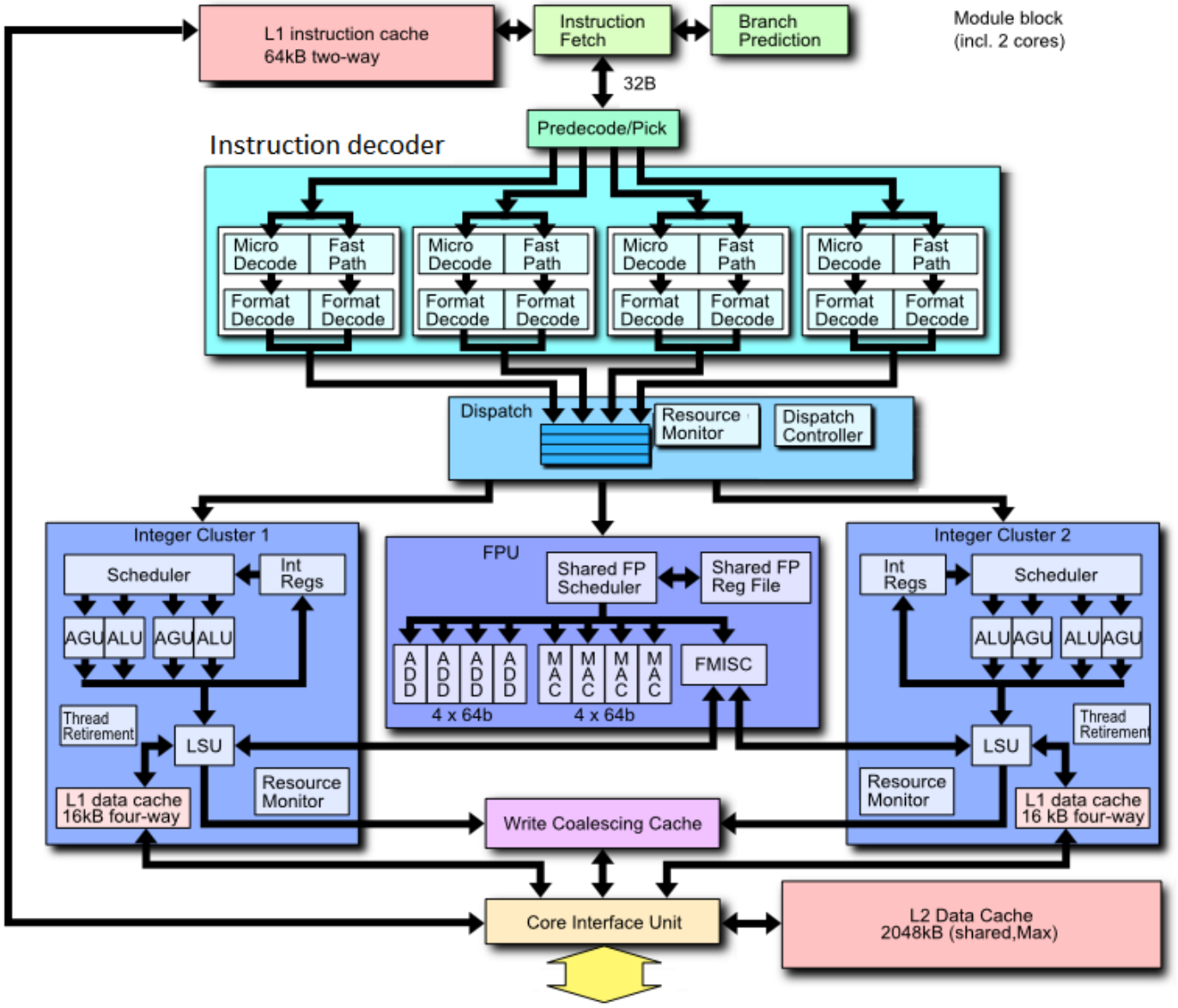


No influence on integer arithmetic performance or memory bandwidth! Why?



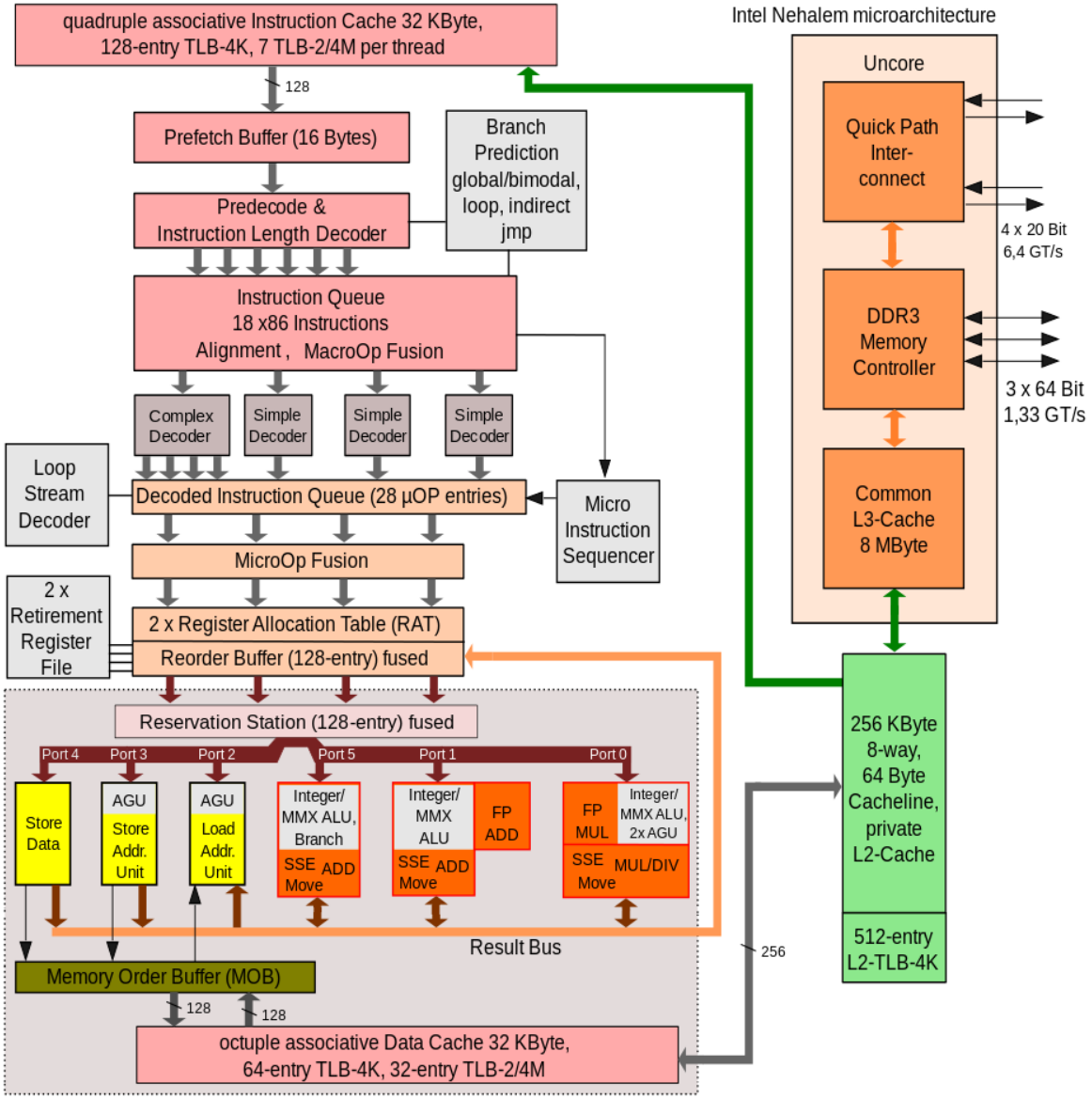


Motivace k přednášce – AMD **Bulldozer** 15h (FX, Opteron) - 2011



- http://en.wikipedia.org/wiki/File:AMD_Bulldozer_block_diagram_(CPU_core_bloack).PNG

Motivace k přednášce – Intel **Nehalem** (Core i7) - 2008



- http://en.wikipedia.org/wiki/File:Intel_Nehalem_arch.svg