Computer Architectures

Branch Prediction and Speculative Execution

Pavel Píša, Richard Šusta

Michal Štepanovský, Miroslav Šnorek

Czech Technical University in Prague, Faculty of Electrical Engineering

Control Hazards

- Jump and Branch processing and decision is significant obstacle for pipelined execution.
- **Jump instruction** needs only the jump target address
- Branch instruction depends on two sources:

• Branch Result	**Taken** or **Not Taken**
• Branch Target Address	

Example of RISC-V **beq** and **bne**:

• PC + 4	If Branch is NOT Taken
• PC + 2 × immediate	If Branch is Taken

Benchtests of Branch Statistics

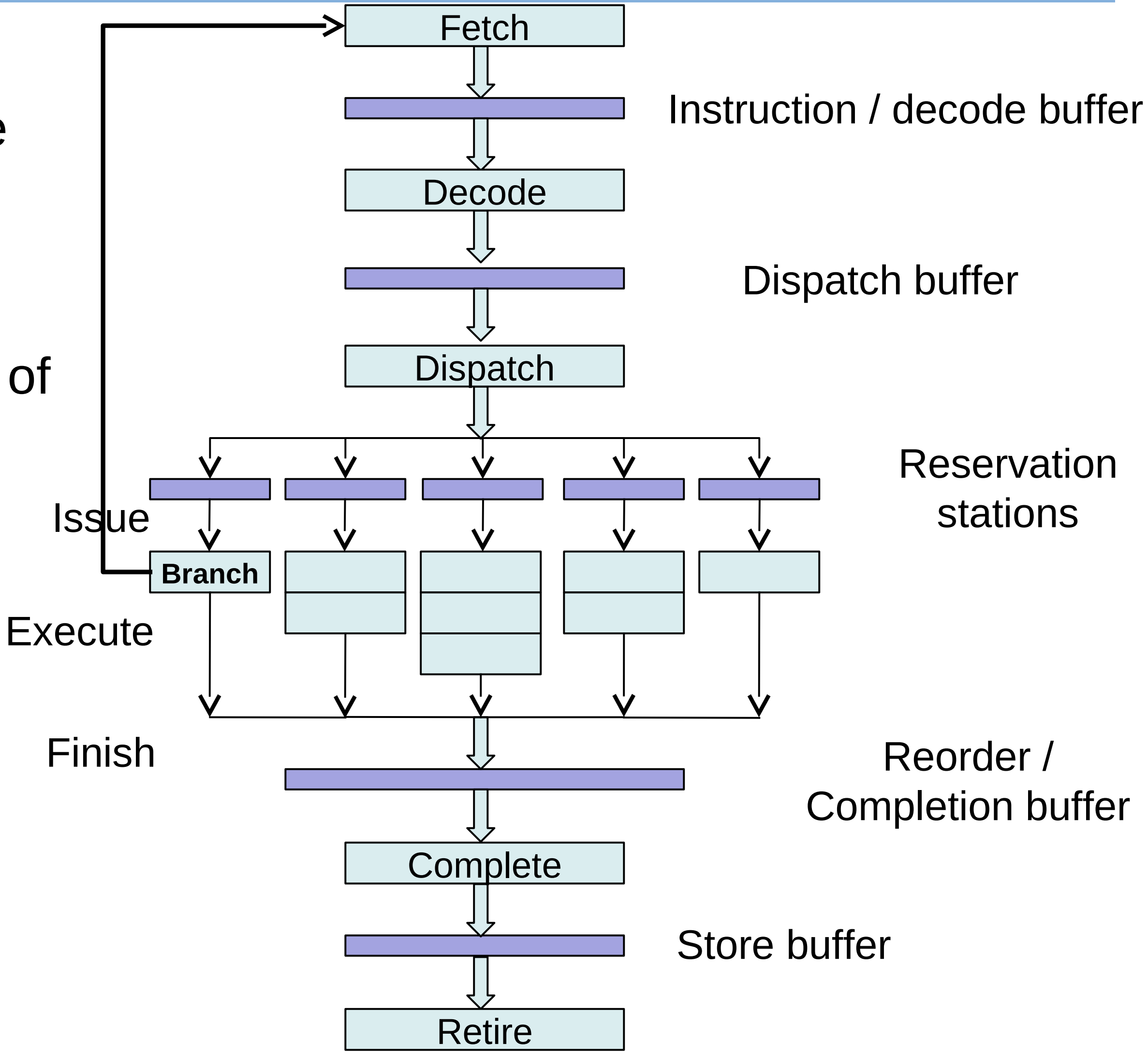
- Branches occur every **4-7 instructions** on average in integer programs, commercial and desktop applications;
  somewhat less frequently in scientific ones :-)
- **Unconditional** branches : approx. **20%** (of branches)
- **Conditional** branches approx. **80%** (of branches)
  - **66%** is forward. 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%**.

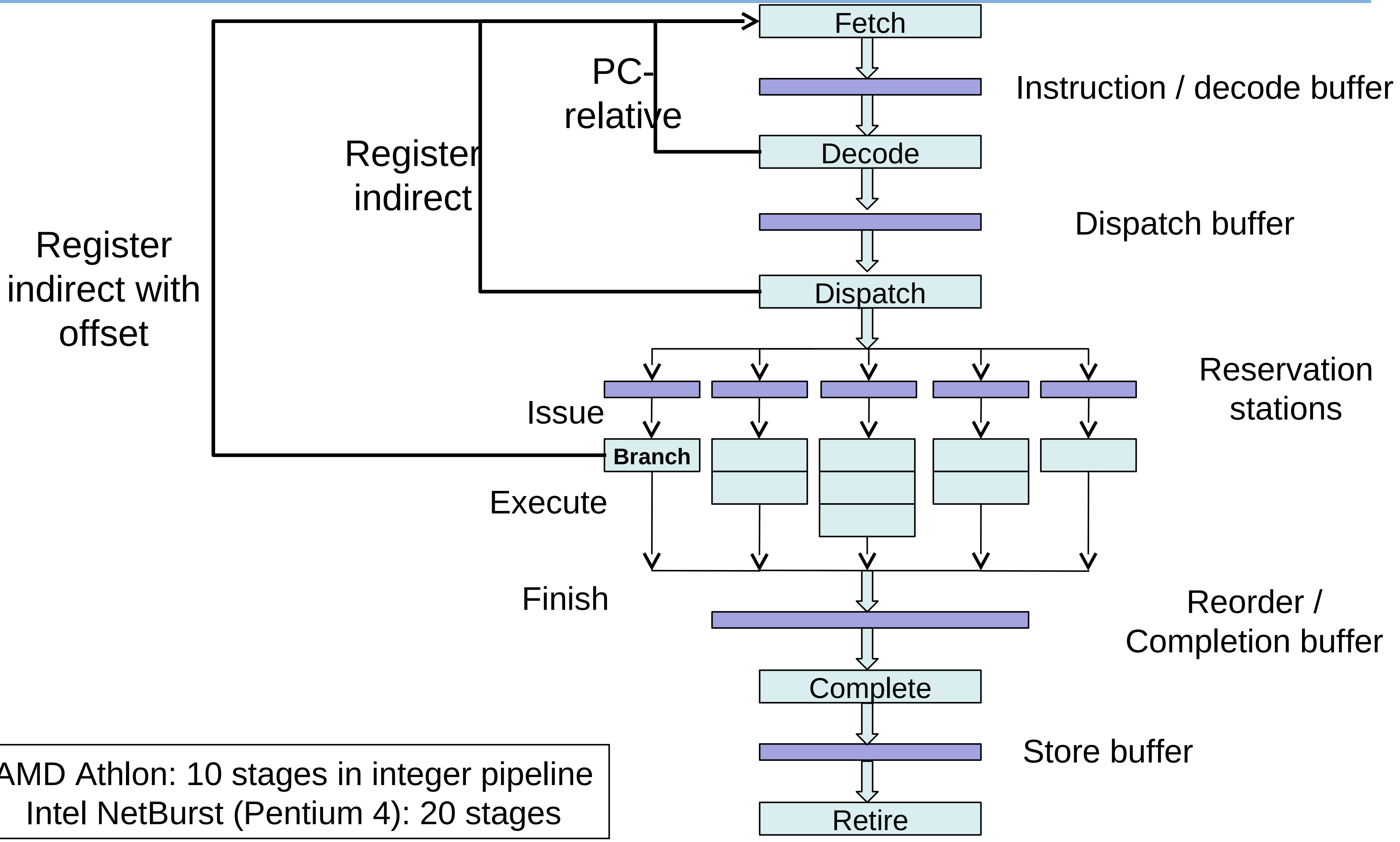
See: Lizy Kurian John, Lieven Eeckhout:
Performance Evaluation and Benchmarking, *CRC Press 2018*

Branch prediction – Motivation

- the penalty of three cycles in fetching the next instruction;
- number of empty instruction slots multiplied by the width of the superscalar machine;
- Amdahl's law..



Branch prediction – Motivation



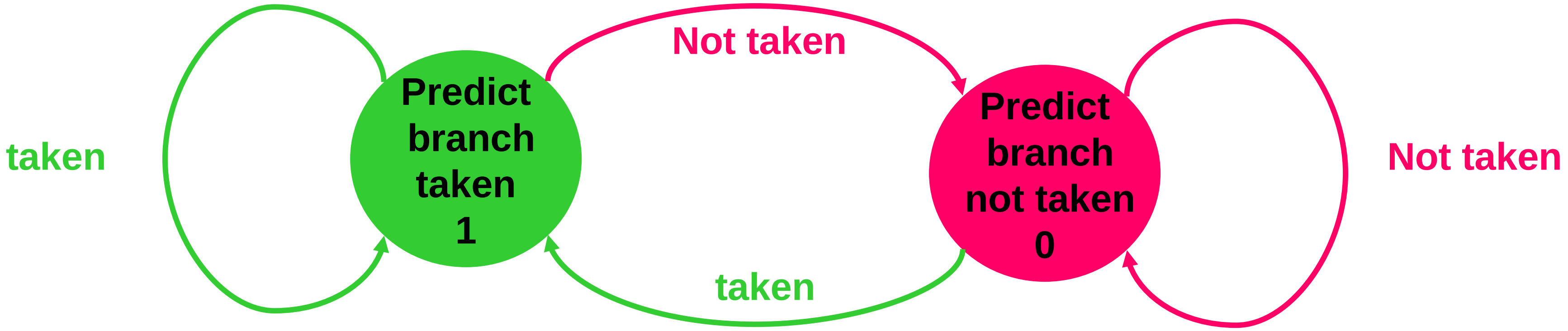
AMD Athlon: 10 stages in integer pipeline
Intel NetBurst (Pentium 4): 20 stages

Branch prediction

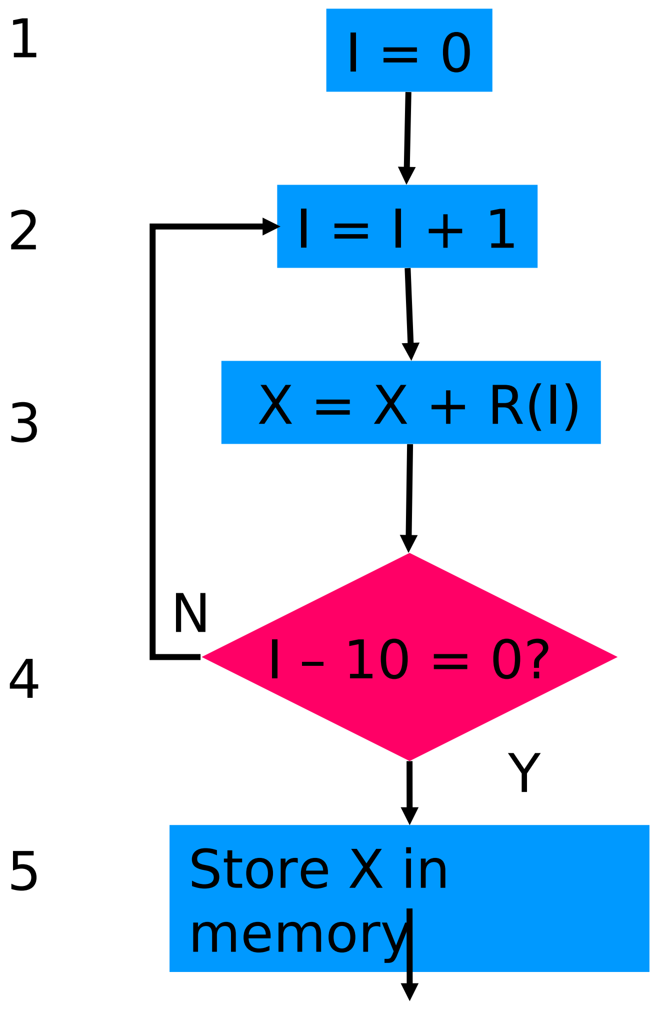
- Two fundamental components:
  - branch target speculation (where is next instruction),
  - branch condition speculation (if the branch is taken).
- Branch target speculation:
  - BTB (Branch Target Buffer) – cache (associative memory) with two fields: BIA (Branch Instruction Address) and BTA (Branch Target Adress) – accessed during the instruction fetch using the instruction fetch address (PC)
  - When BIA matches with current PC, the corresponding BTA is accessed and if the branch instruction is predicted to be taken, BTA is used to modify PC

One-bit Branch Prediction (usually local)

- 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 – One Bit predictor



Execution of Instruction 4

Execu-tion seq.	Old hist. bit	Next instr.			New hist. bit	Prediction
		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

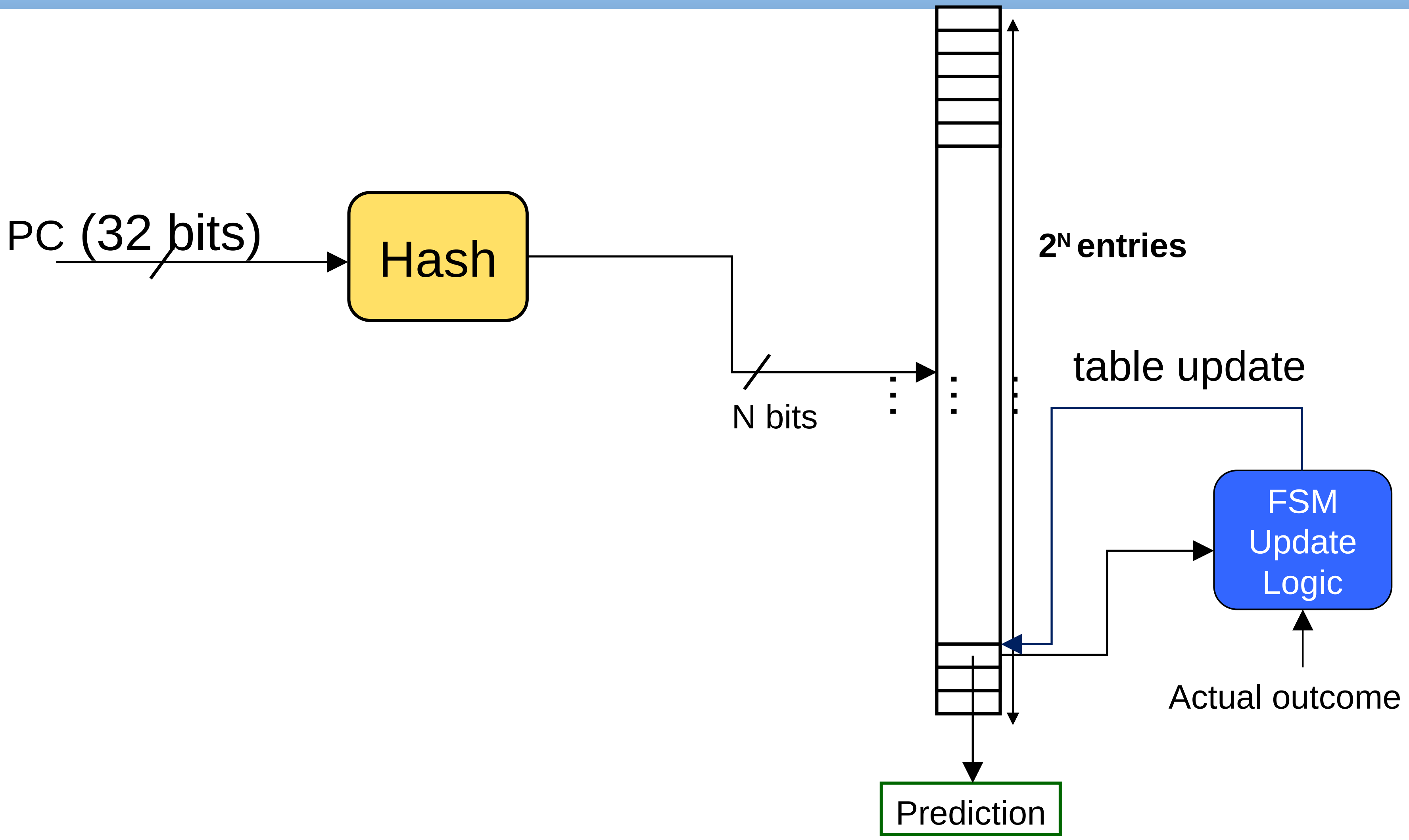
bit = 0 *branch not taken*, bit = 1 *branch taken*.

Demonstration in RISC-V RARS Simulator

- RARS (RISC-V Assembler and Runtime Simulator)
  https://github.com/TheThirdOne/rars

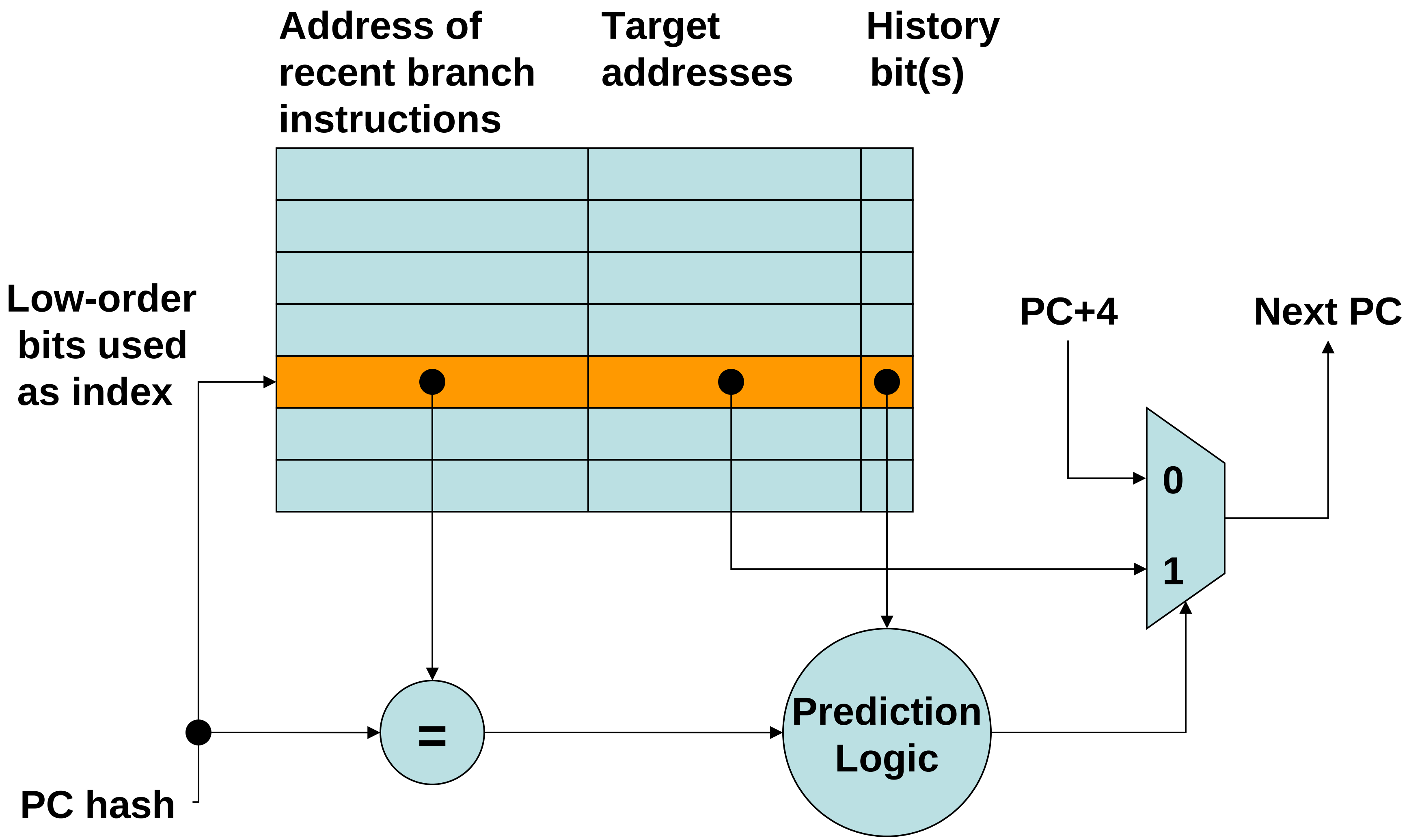
```
start:  addi a0, zero, 0      # cycles = 0
        addi s1, zero, 3
L1: # for (i = 3; i !=0; i--)
        addi s2, zero, 5      # i = 3
L2: # for (j = 5; j !=0; j--)
        addi s3, zero, 4
L3: # for (k = 4; k !=0; k--)
        addi a0, a0, 1        # cycles++
        addi s3, s3, -1       # k--
        bne  s3, zero, L3     # if (k != 0) goto L3

        addi s2, s2, -1       # j--
        bne  s2, zero, L2     # if (j != 0) goto L2

        addi s1, s1, -1       # i--
        bne  s1, zero, L1     # if (i != 0) goto L1

        ebreak
```

Typical Organization of Branch Prediction Table



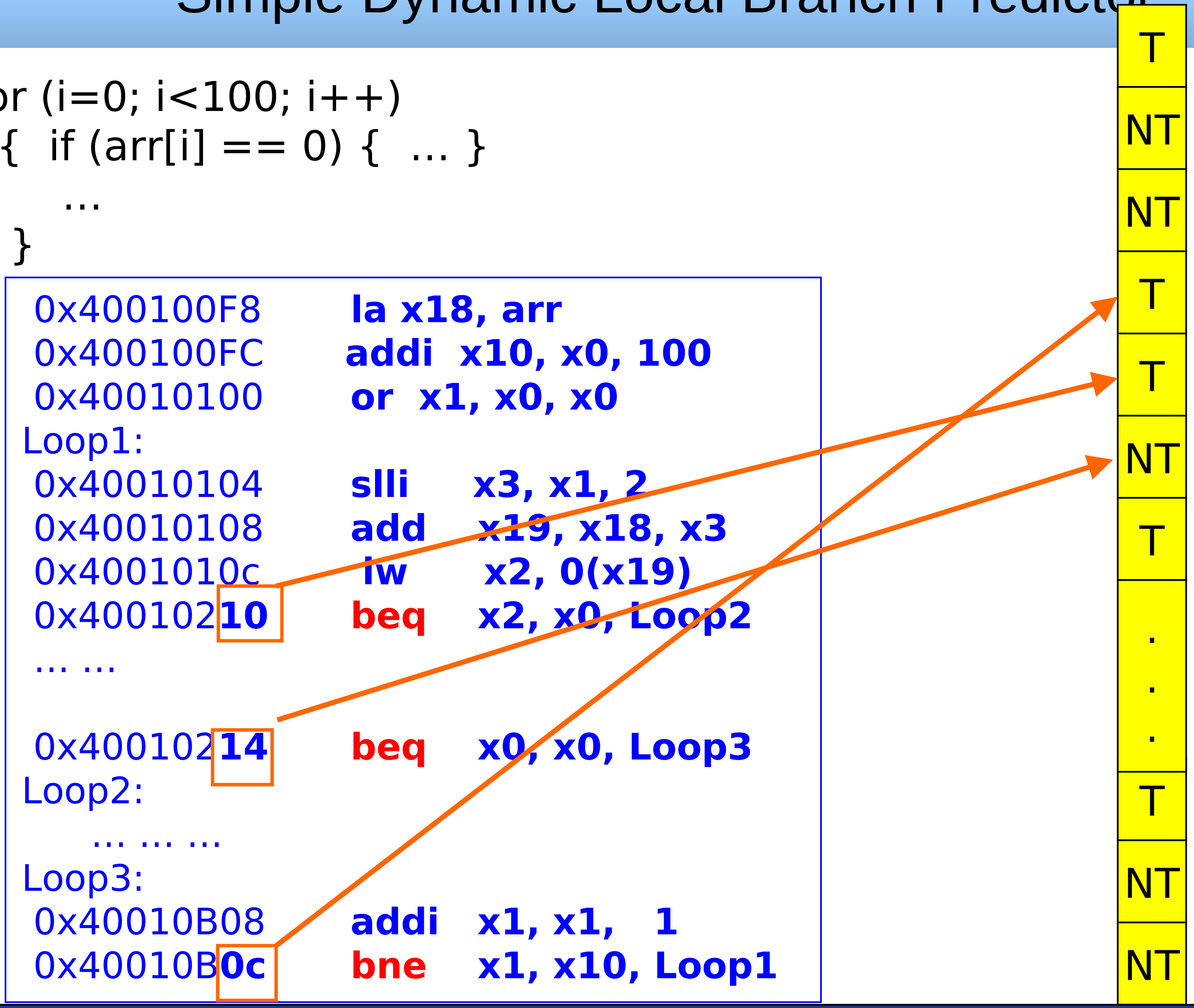
Note: FSM - Finite State Machine (cz: konečný automat)

Branch Prediction



Simple Dynamic Local Branch Predictor

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

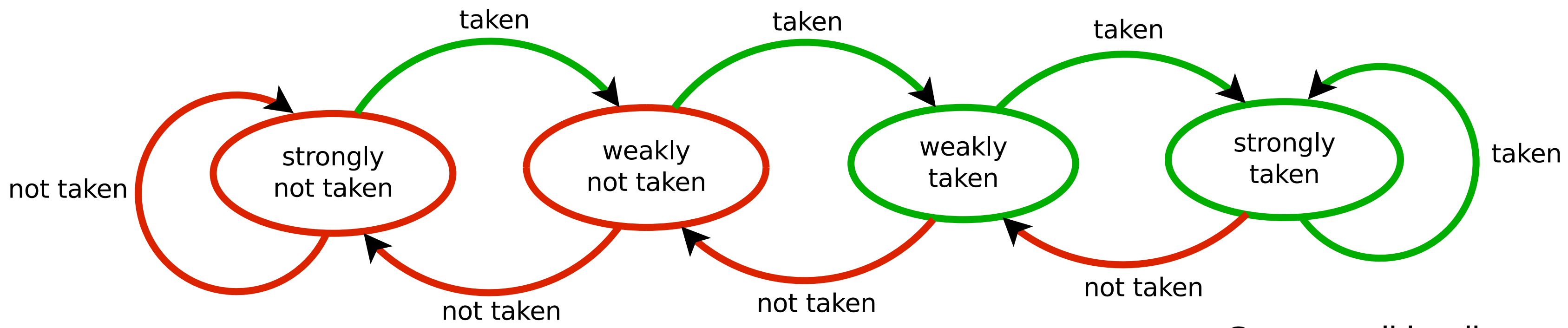


```
 0x400100F8     la x18, arr
 0x400100FC     addi  x10, x0, 100
 0x40010100     or  x1, x0, x0
Loop1:
 0x40010104     slli    x3, x1, 2
 0x40010108     add     x19, x18, x3
 0x4001010c      lw      x2, 0(x19)
 0x40010210     beq     x2, x0, Loop2
 … …

 0x40010214     beq     x0, x0, Loop3
Loop2:
      … … …
Loop3:
 0x40010B08     addi   x1, x1,   1
 0x40010B0c     bne    x1, x10, Loop1
```

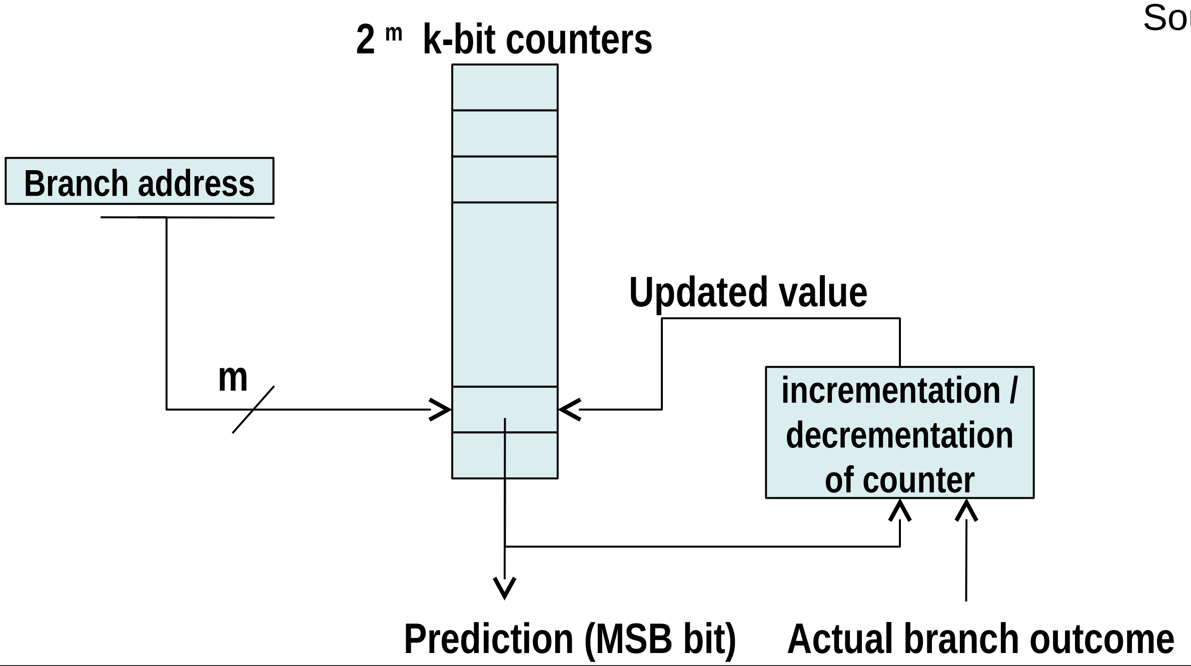
1-bit Branch History Table

Two-Bit Prediction Buffer Type I

- Smith's algorithm (2-bit saturating counter)

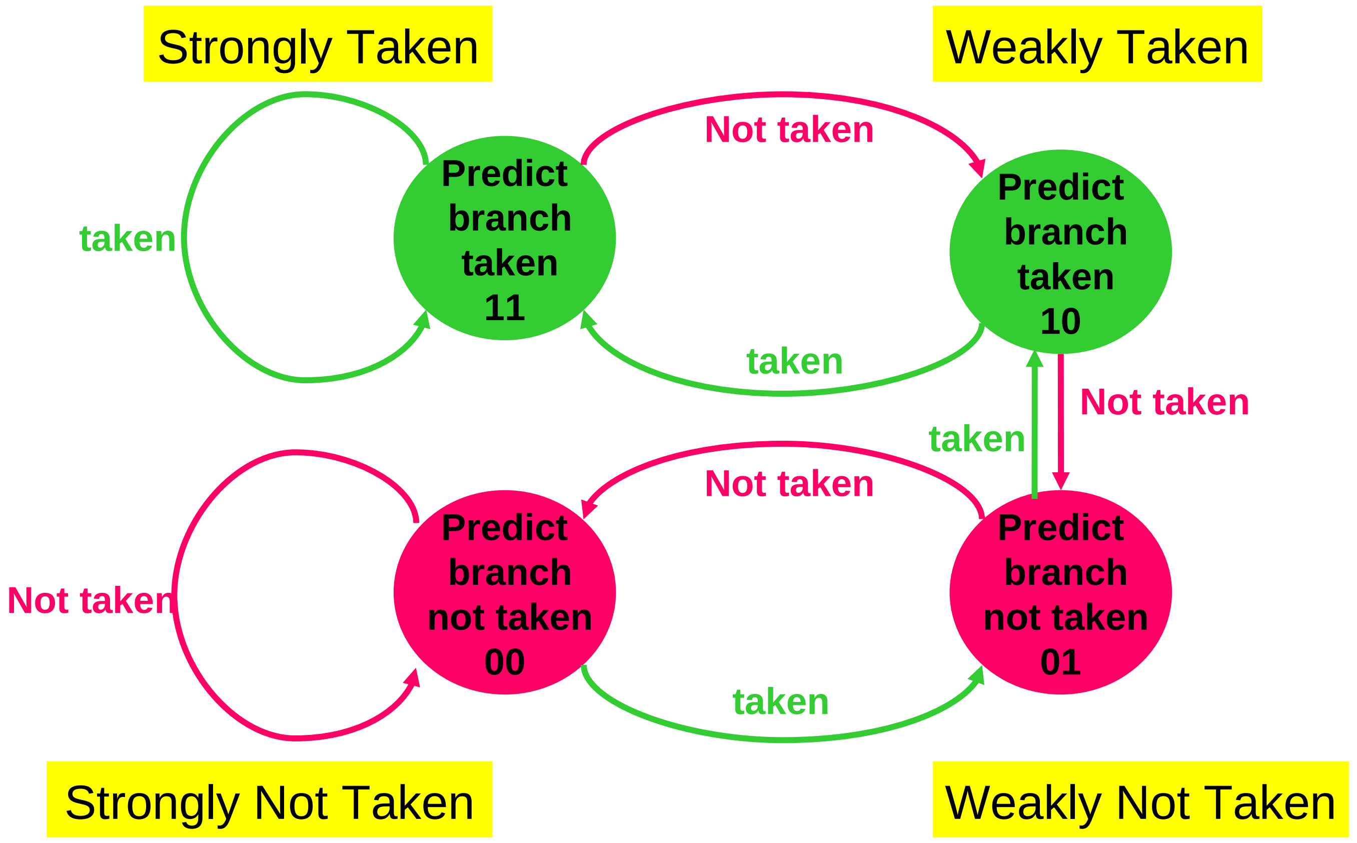


Source: wikipedia

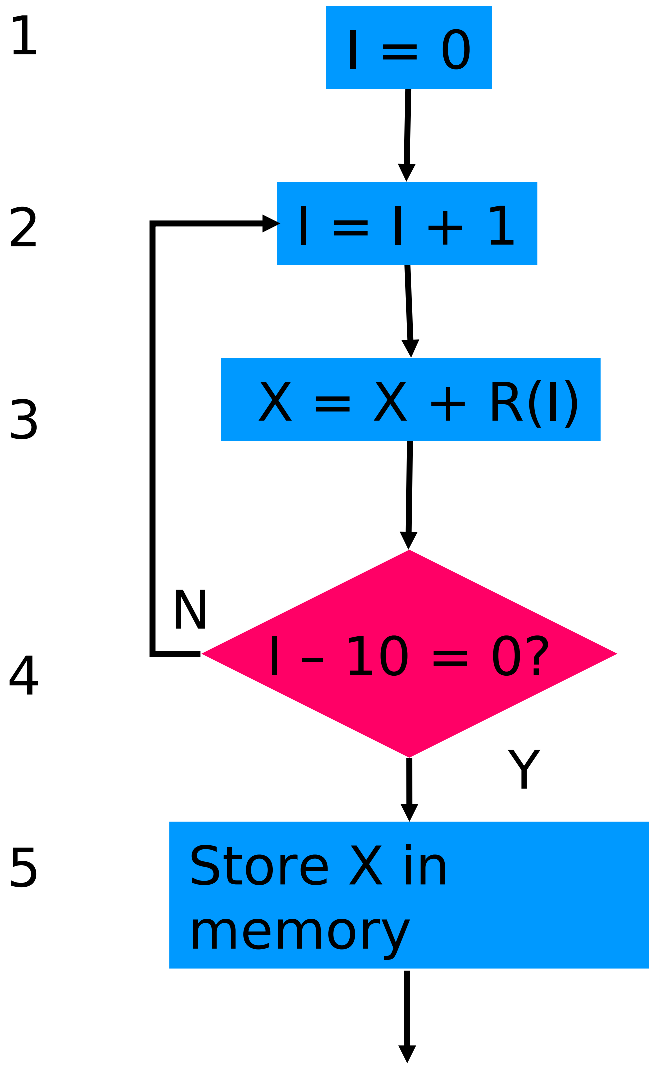


Two-Bit Prediction Buffer Type I

- Smith’s algorithm (2-bit saturating counter). This one has no hysteresis.



Branch Prediction for a Loop – Smith's Predictor

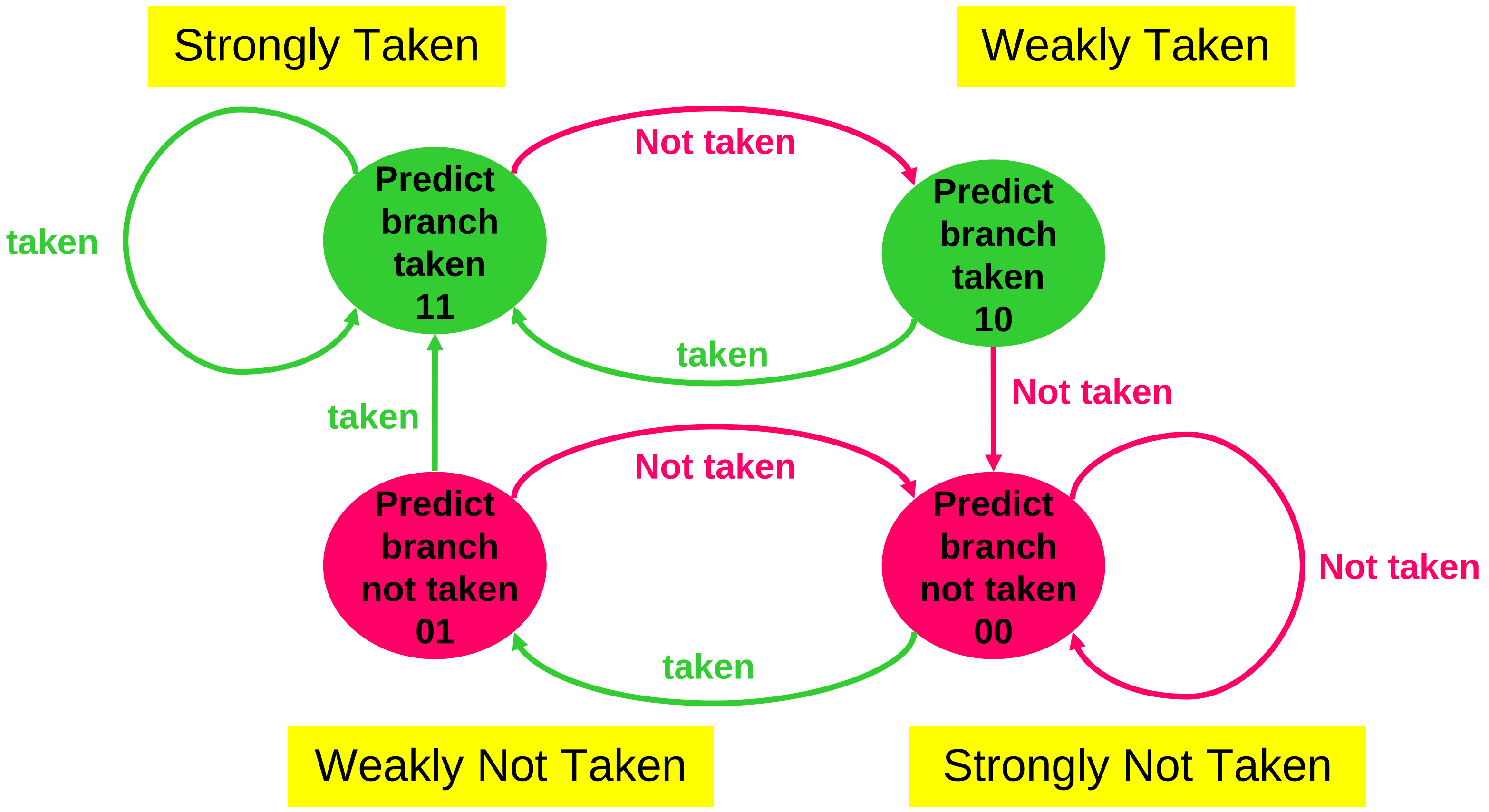


Execution of Instruction 4

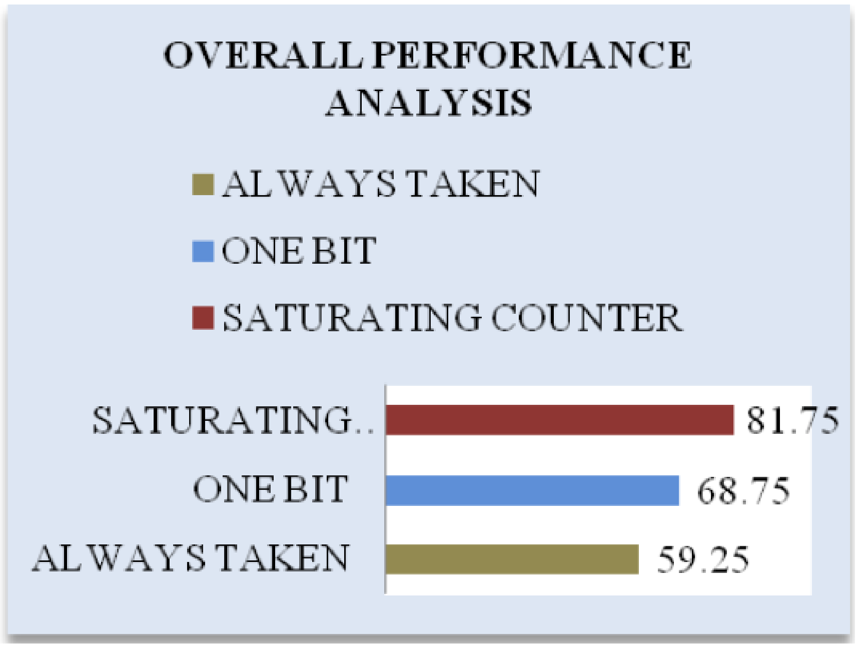
Execu-tion seq.	Old Pred. Buf	Next instr.			New pred. Buf	Prediction
		Pred.	I	Act.		
1	10	2	1	2	11	Good
2	11	2	2	2	11	Good
3	11	2	3	2	11	Good
4	11	2	4	2	11	Good
5	11	2	5	2	11	Good
6	11	2	6	2	11	Good
7	11	2	7	2	11	Good
8	11	2	8	2	11	Good
9	11	2	9	2	11	Good
10	11	2	10	5	10	Bad

Two-Bit Prediction Buffer Type II.

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



Example of Results of Benchtest



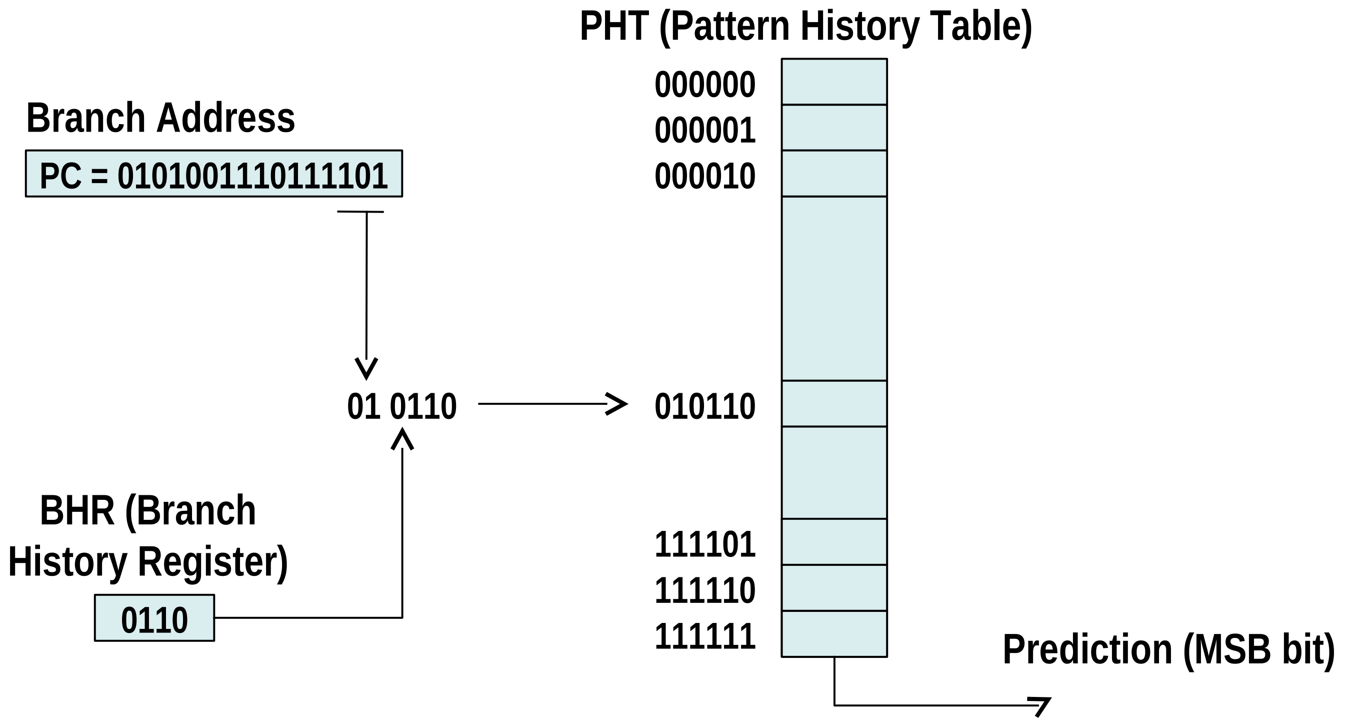
Here, a higher number means the better prediction

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).

Branch Condition Prediction – Global Predictor

- **Global-History** Two-Level Branch Predictor with a 4-bit Branch History Register



What is the optimal number of bits for BHR a for BA?

Branch Condition Prediction – Global Predictor

- **Global-History** Two-Level Branch Predictor with a 4-bit Branch History Register

- Why use global history for PHT indexation?

```
a=0;
if(condition #1)   a=3;
if(condition #2)   b=10;
if(a <= 0)  F();
```

- The behavior of a branch may be connected (or correlated) with a different branches conditions evaluation in the past.
- In our example, execution of function F() depends on the condition #1. The condition #2 is irrelevant. Predictor must be able to learn this behavior (distinguish these branch conditions).

Correlating Predictors with Local 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.**

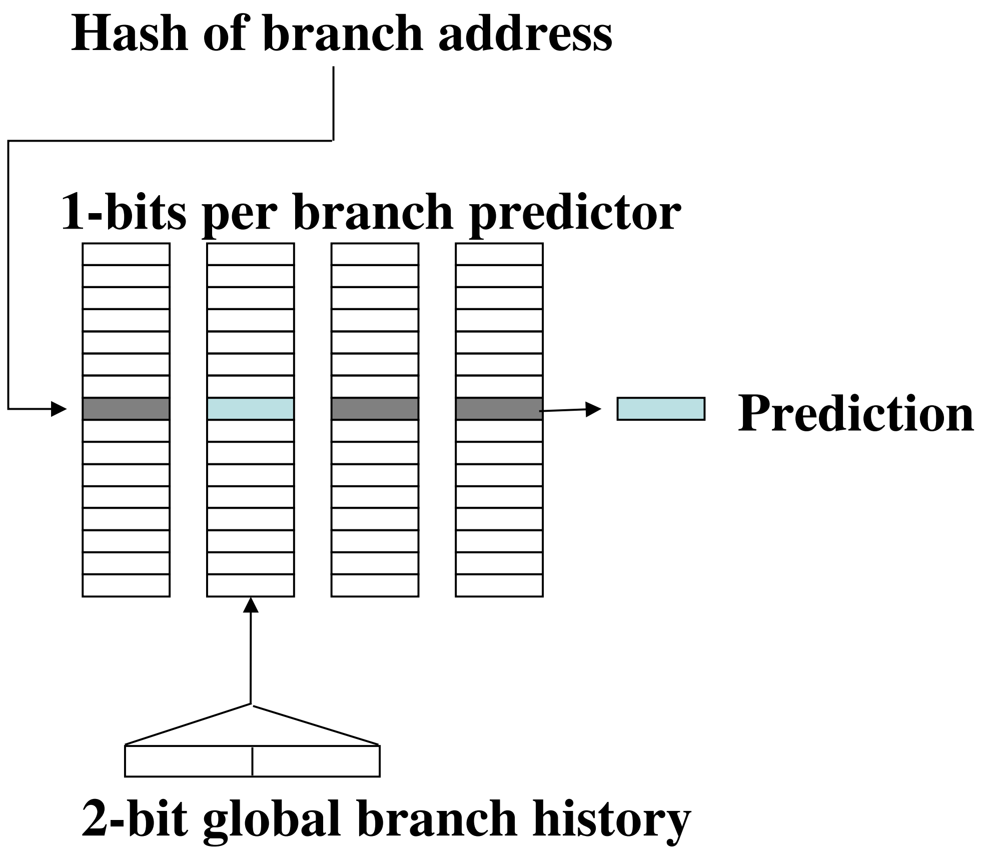
P11
This predictor is used if the previous 2 branches in the program have both status **Taken**.

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.

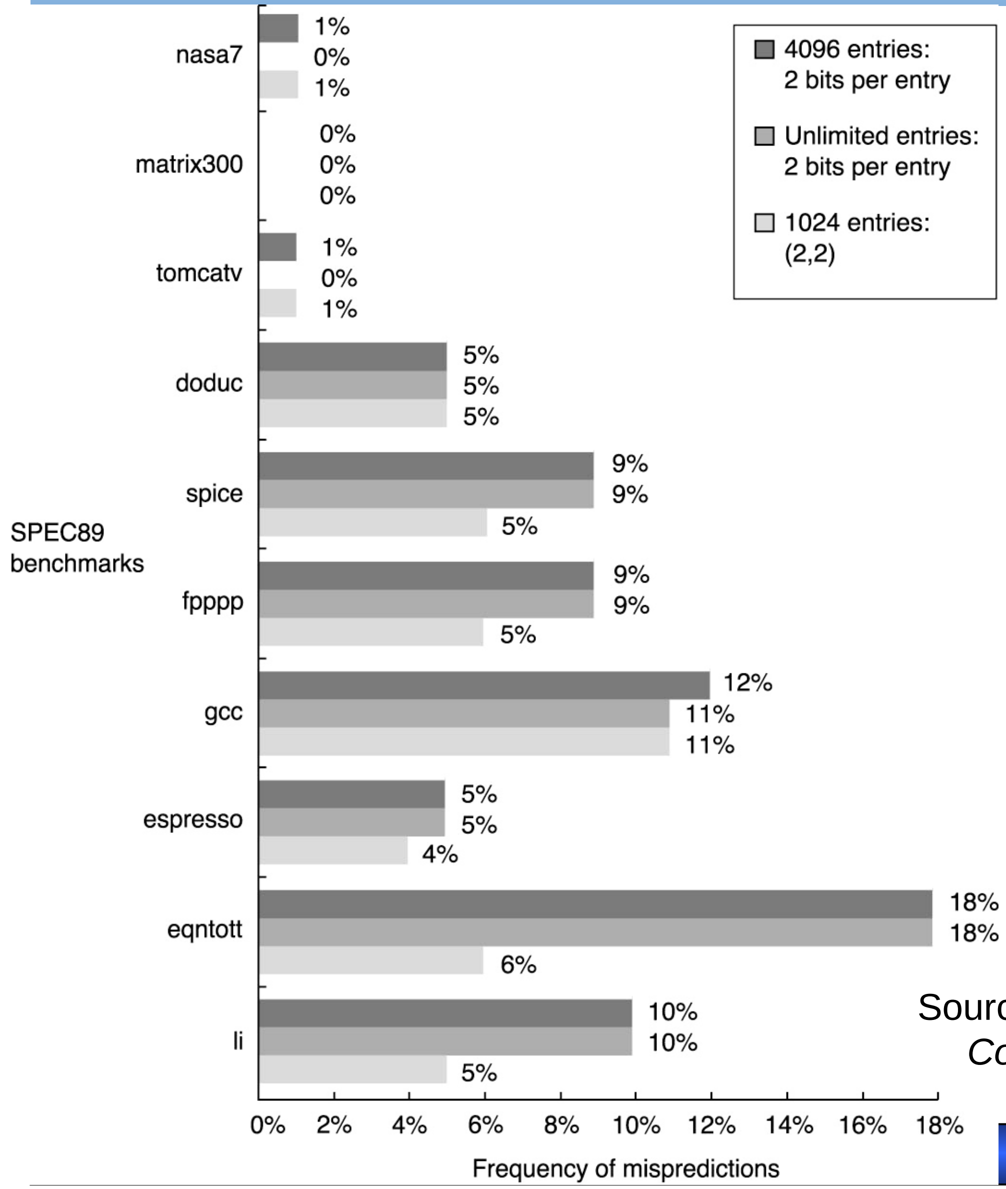
Correlating Predictors

- 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:

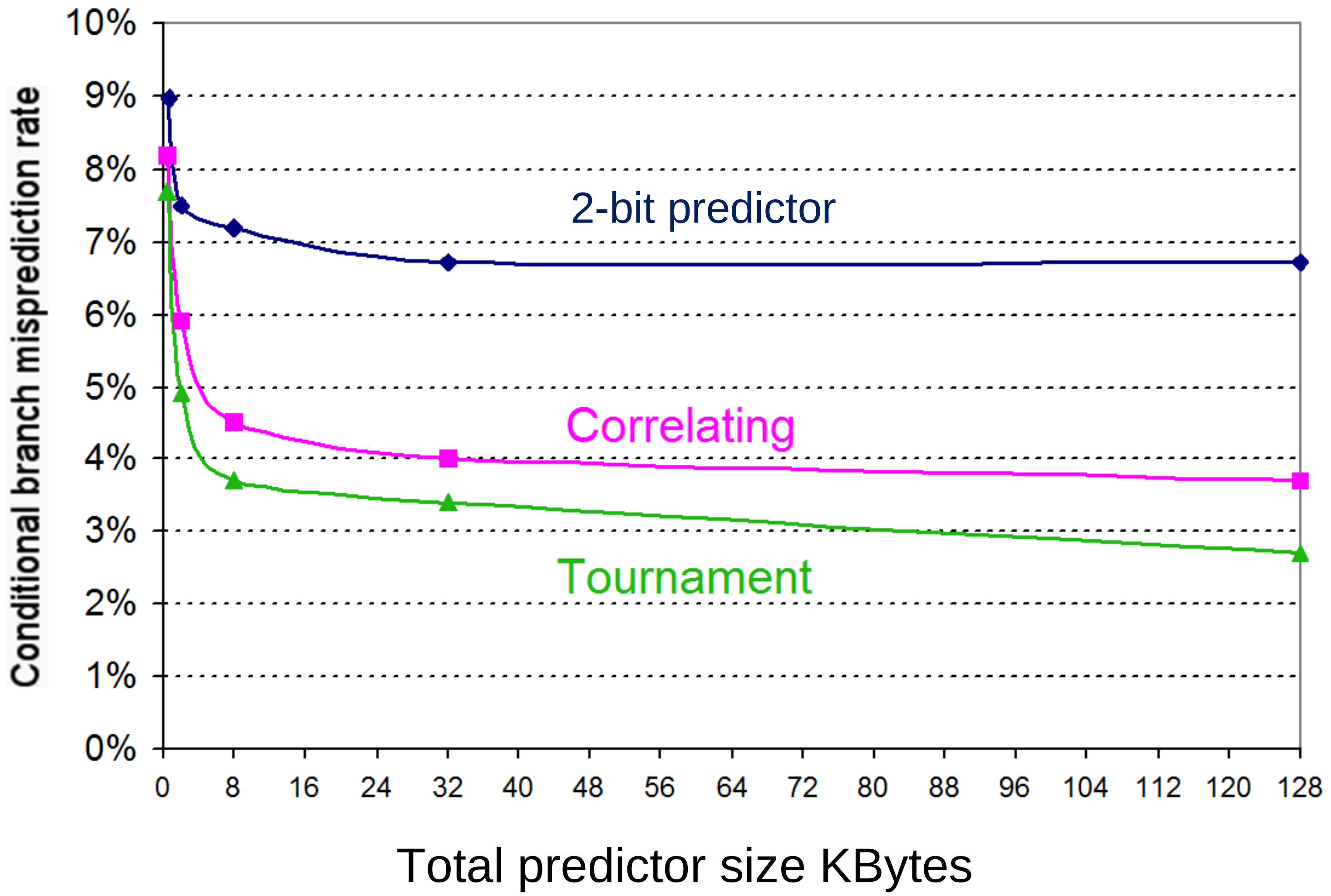
- **gcc** INT1 GNU C compiler
- **espresso** INT PLA optimizing tool
- **spice2g6** FP2 Circuit simulation and analysis
- **doduc** FP Monte Carlo simulation
- **nasa7** FP Seven floating-point kernels
- **li** INT LISP interpreter
- **eqntott** INT Conversions of equations to truth table
- **matrix300** 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 Accuracy – Test for Tournament Predictor



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	**ASM, x in s2**	**Comment**
if(x<0) x=-x;	slt s1, s2, zero	// tmp = x<0 ? 1 : 0
	beq s1, zero, Skip1	// if(tmp==0) goto Skip
	sub s2, zero, s2	// x = - x;
Skip1:	…	

Fast C code	**ASM, x in s2**	**Comment**
int tmp = x>>31;	srai s1, s2, 31	// tmp = x<0 ? -1 : 0
x ^= tmp;	xor s2, s2, s1	// 1st compliment of x, if tmp=-1
x -= tmp;	sub s2, s2, s1	// 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.

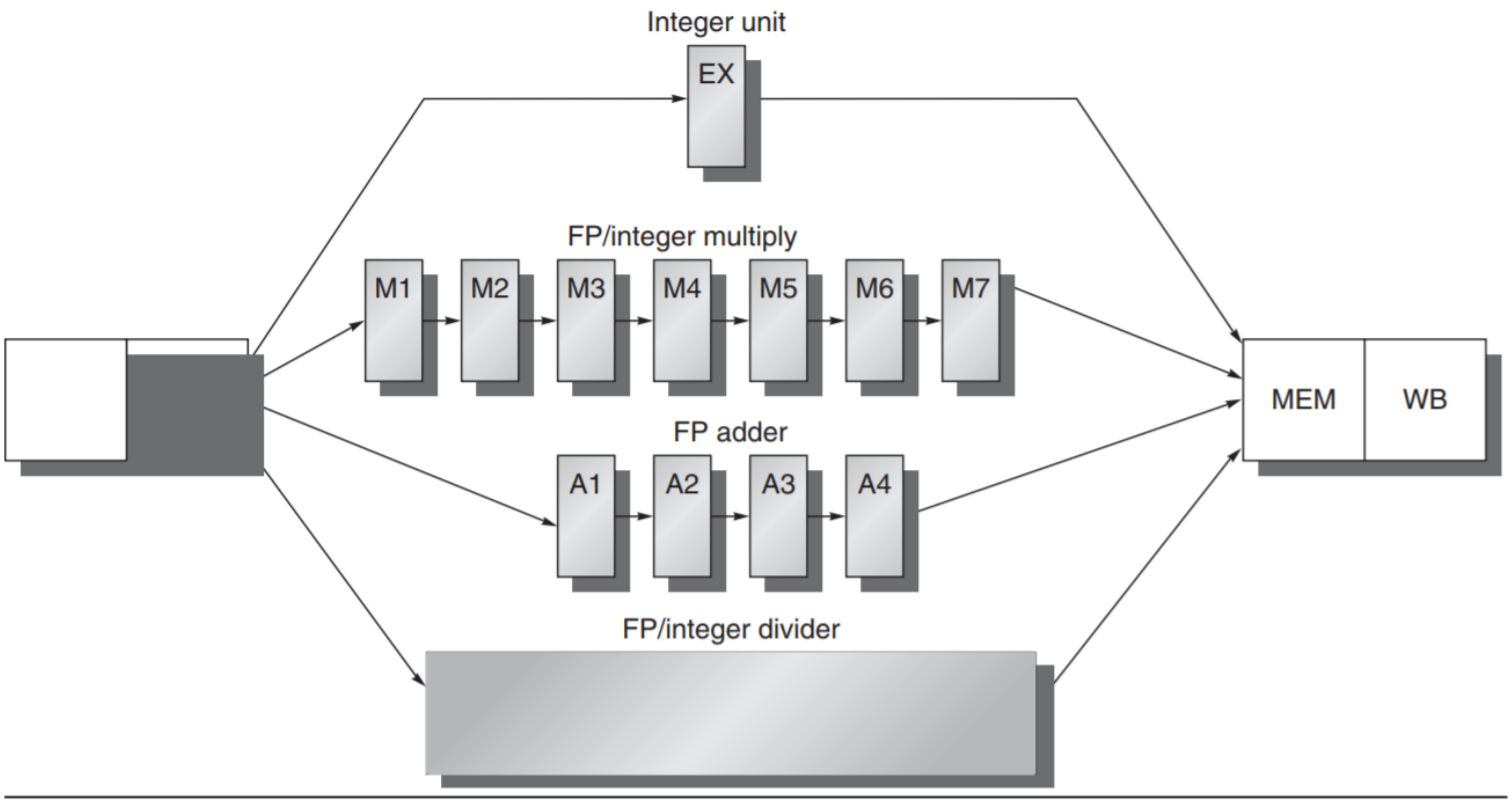
What are Dynamic multiple-issue processors aka Superscalar processors ?

Definition of Superscalar Processor

Wikipedia:

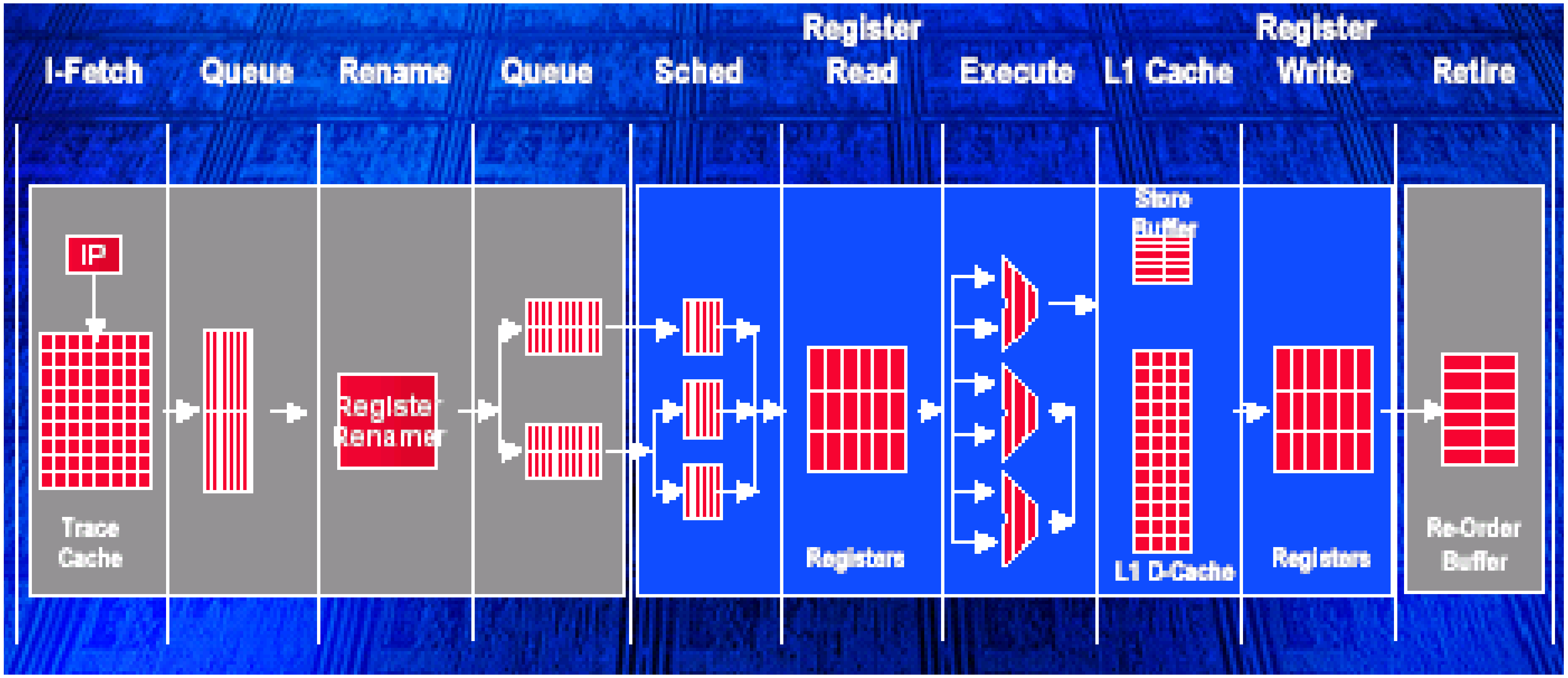
- In contrast to a **scalar processor** that can execute at most **one single instruction per clock cycle**, a **superscalar processor** can execute **more than one** instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor.
- Q: What does it actually mean "more than one"?

A Pipeline That Supports Multiple Outstanding FP Operations



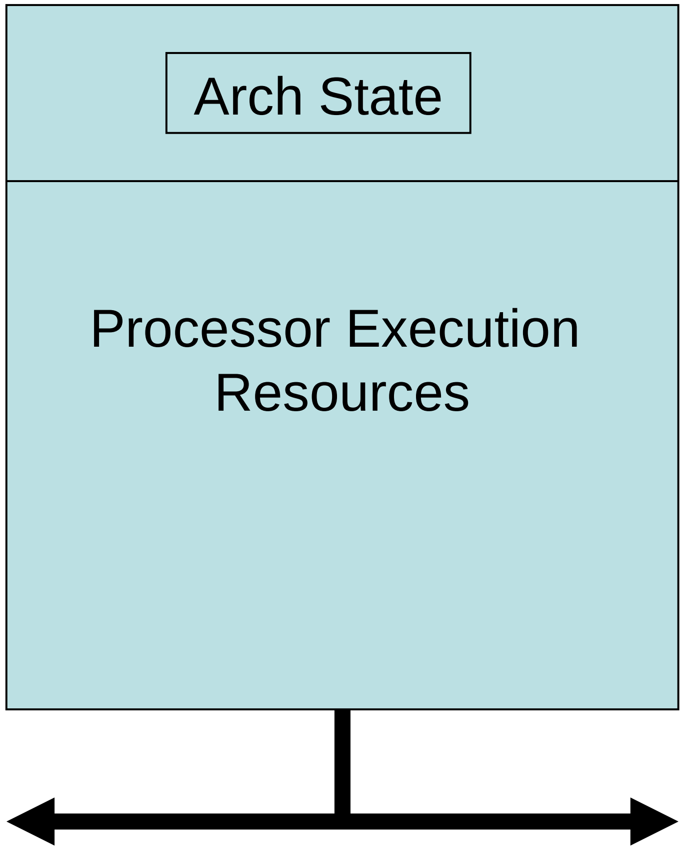
Source of picture: J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.*

Pentium 4 - Out-of-order Execution pipeline

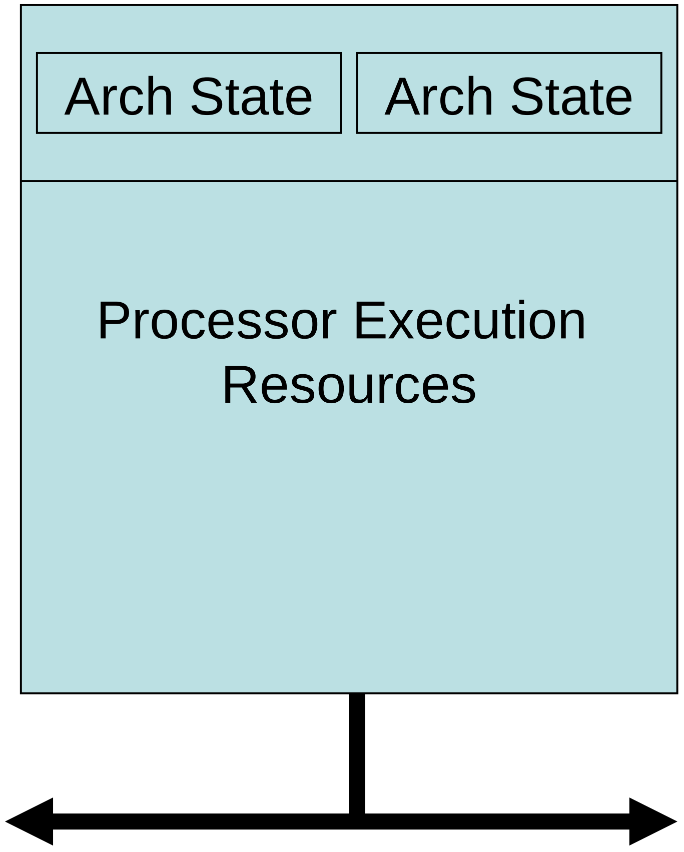


[Source: Intel]

Hyper-Threading



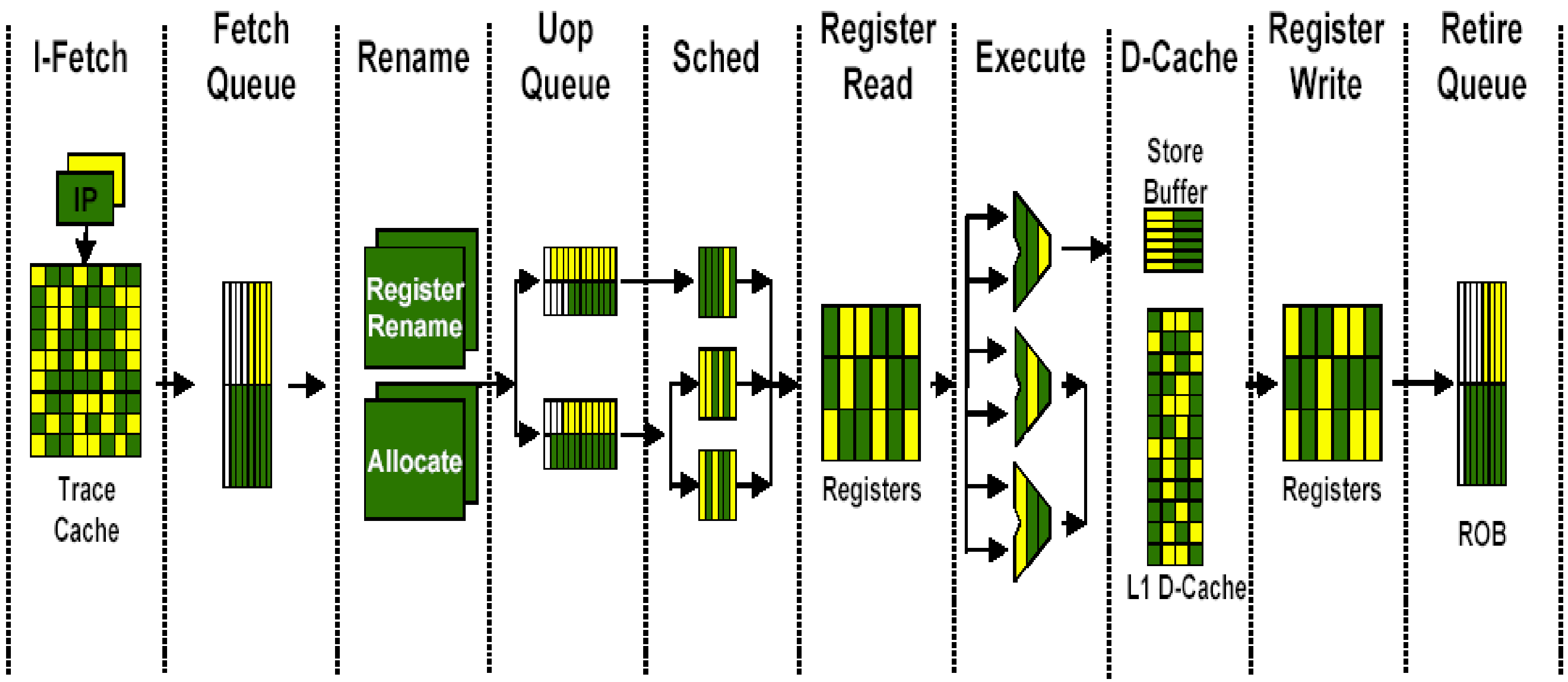
Processor with out Hyper-Threading Technology



Processor with Hyper-Threading Technology

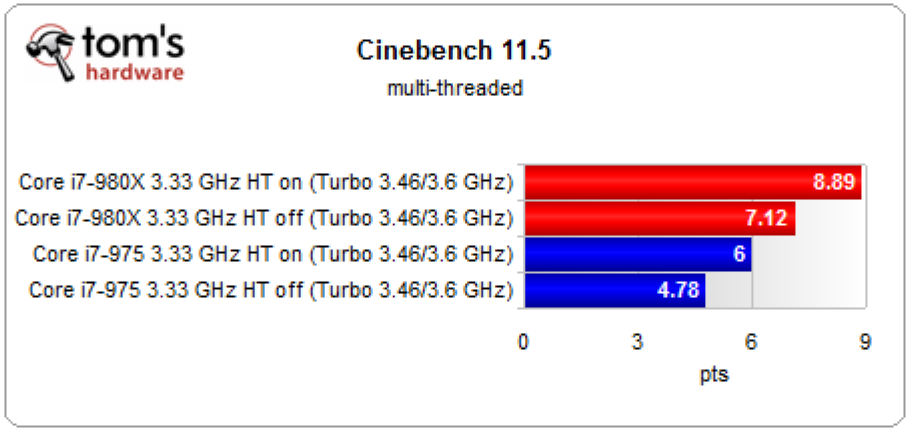
Ref: Intel Technology Journal, Volume 06 Issue 01, February 14, 2002

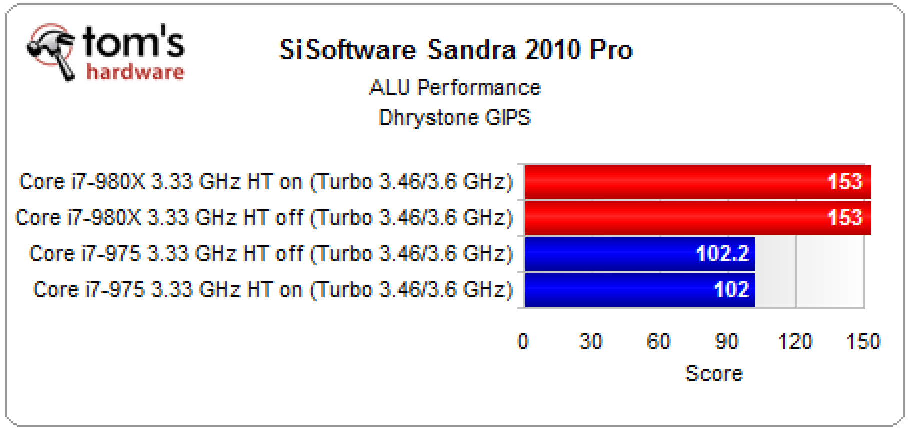
Pentium 4: Netburst Microarchitecture’s execution pipeline



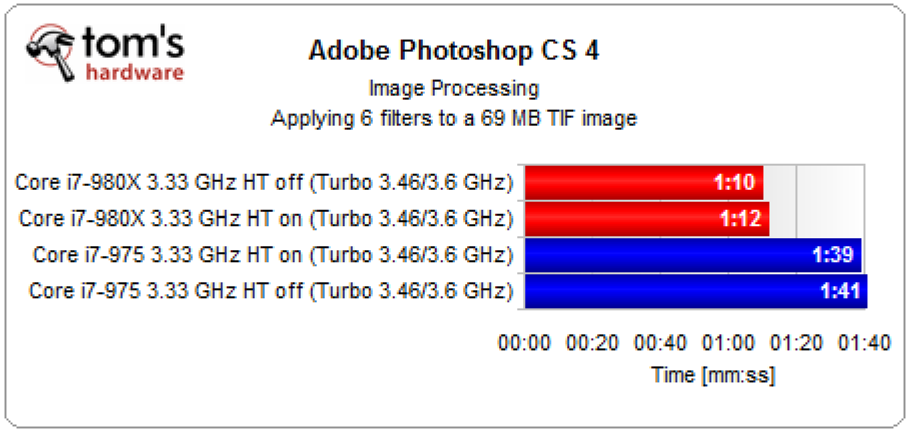
Picture is simplified because the pipeline has actually 20 steps.
The branch miss prediction penalty is here extremely high.

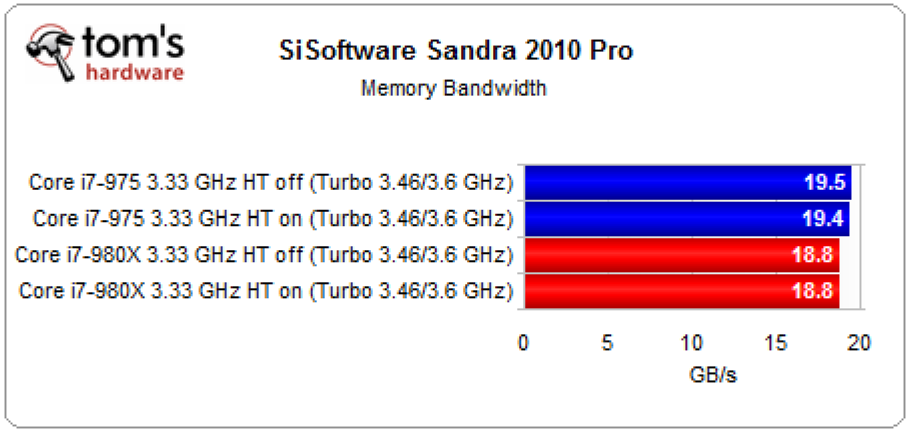
Sample from: Hyper-Threading Benchtest



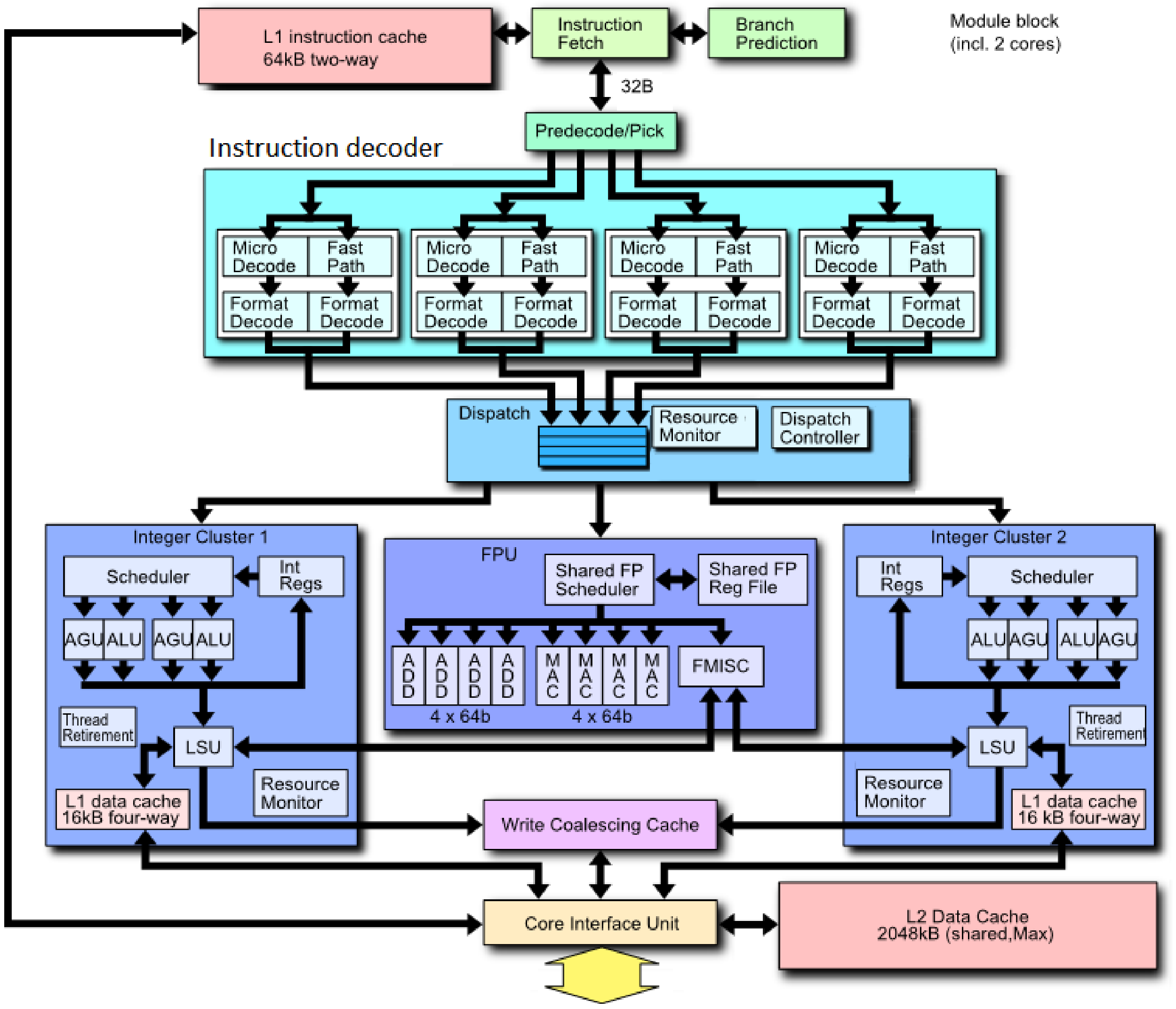


No influence on integer arithmetic performance or memory bandwidth! Why?



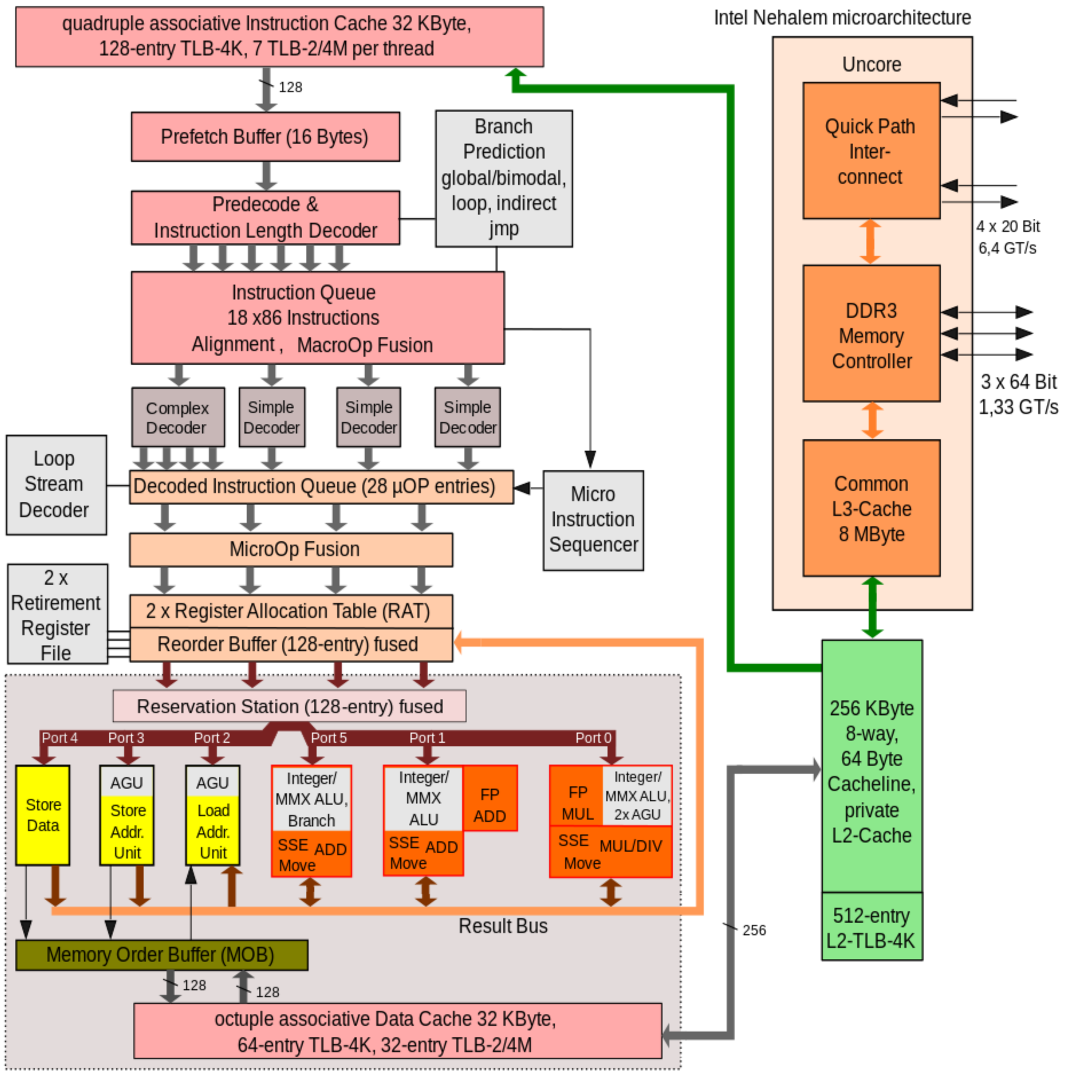


AMD Bulldozer 15h (FX, Opteron) - 2011



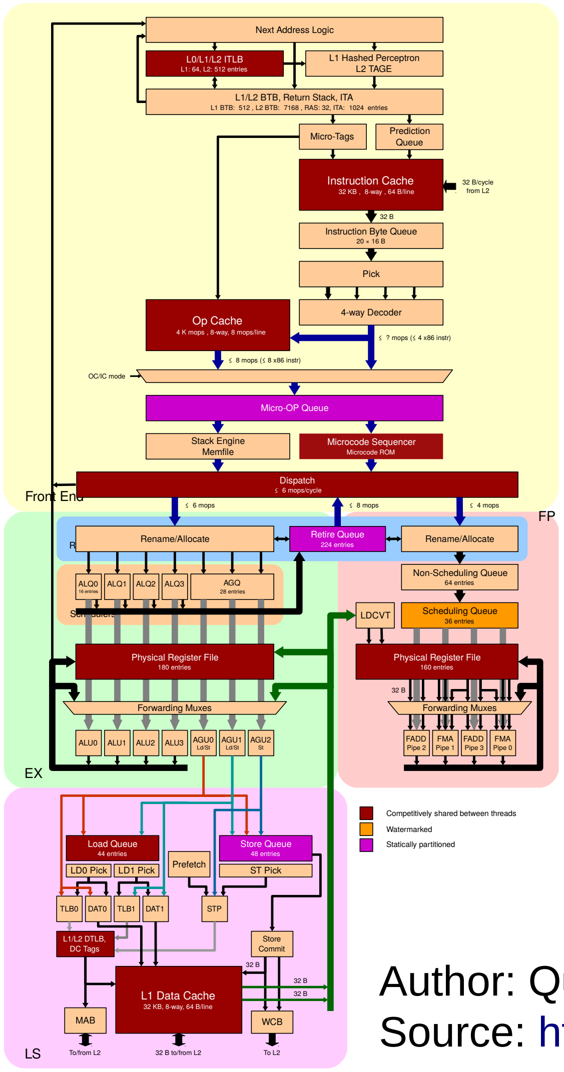
- http://en.wikipedia.org/wiki/File:AMD_Bulldozer_block_diagram_(CPU_core_bloack).PNG

Intel Nehalem (Core i7) - 2008



Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment, MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
Result Bus
Memory Order Buffer (MOB)
128
128
octuple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K

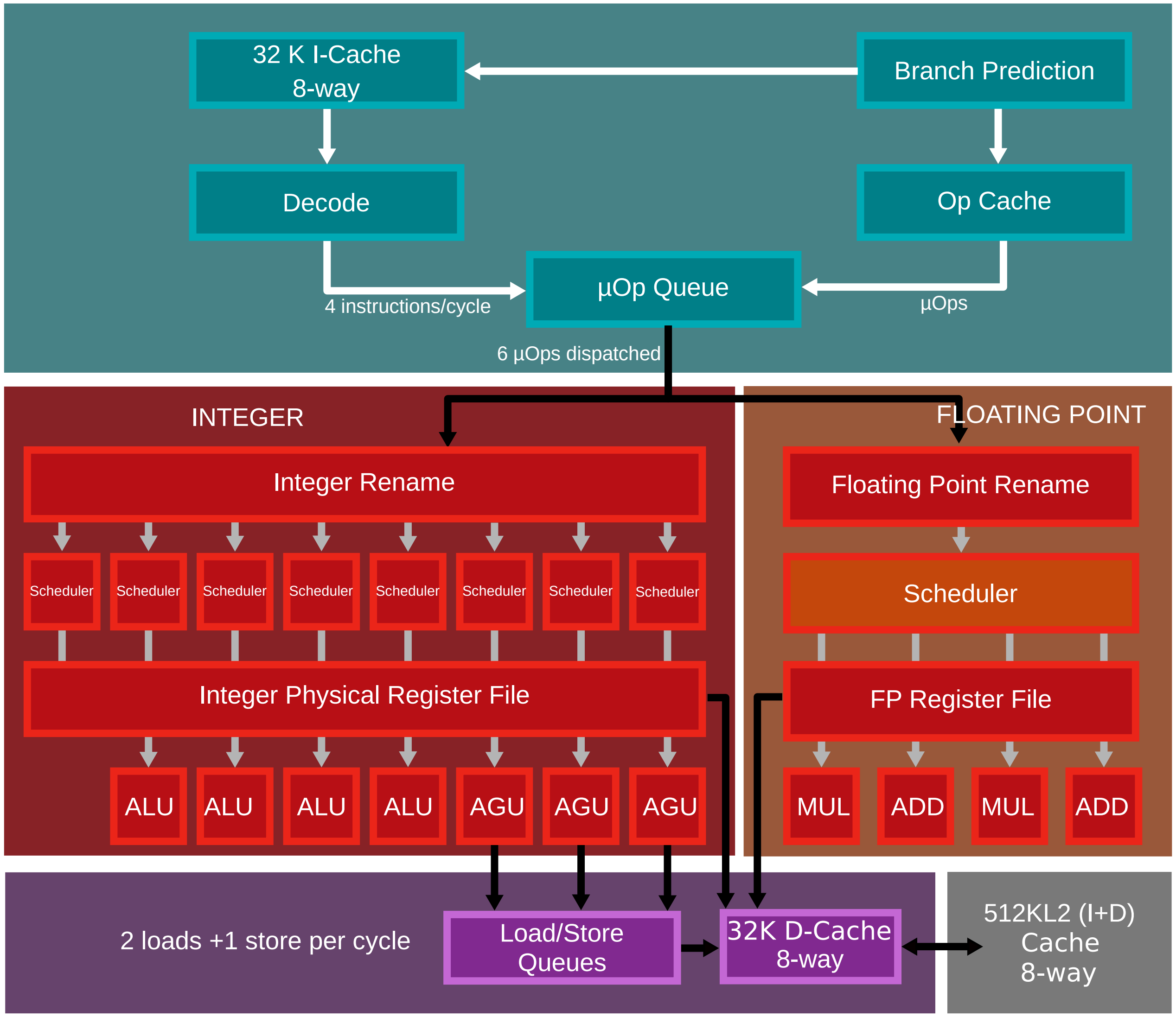
AMD Zen 2 - Microarchitecture



- 7 nm process (from 12 nm), I/O die utilizes 12 nm
- Core (8 cores on CPU chiplet), 6/8/4 µOPs in parallel
  - Frontend, µOP cache (4096 entries)
  - FPU, 256-bit Eus (256-bit FMAs) and LSU (2x256-bit L/S), 3 cycles DP vector mult latency
  - Integer, 180 registers, 3x AGU, scheduler (4x16 ALU + 1x28 AGU)
  - Reorder Buffer 224 entries
- Memory subsystem
  - L1 i-cache and d-cache, 32 KiB each, 8-way associative
  - L2 512 KiB per core, 8-way,
  - L2 DTLB 2048-entry
  - 48 entry store queue
- CCX
  - L3, slices, 2x 16 MiB
  - L3 latency (~40 cycles)
- In-silicon Spectre enhancements
- I/O, PCIe 4.0, Infinity Fabric 2, 25 GT/s

Author: QuietRub
Source: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2

AMD Zen 2 - Microarchitecture



32 K I-Cache 8-way
Branch Prediction
Decode
Op Cache
4 instructions/cycle
µOp Queue
µOps
6 µOps dispatched
INTEGER
FLOATING POINT
Integer Rename
Floating Point Rename
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Integer Physical Register File
FP Register File
ALU
ALU
ALU
ALU
AGU
AGU
AGU
MUL
ADD
MUL
ADD
2 loads +1 store per cycle
Load/Store Queues
32K D-Cache 8-way
512KL2 (I+D) Cache 8-way