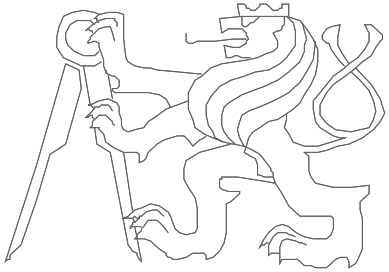
Advanced Computer Architectures

03

Superscalar Techniques –
Data flow inside the processor as a result of instructions execution
(Register Data Flow)



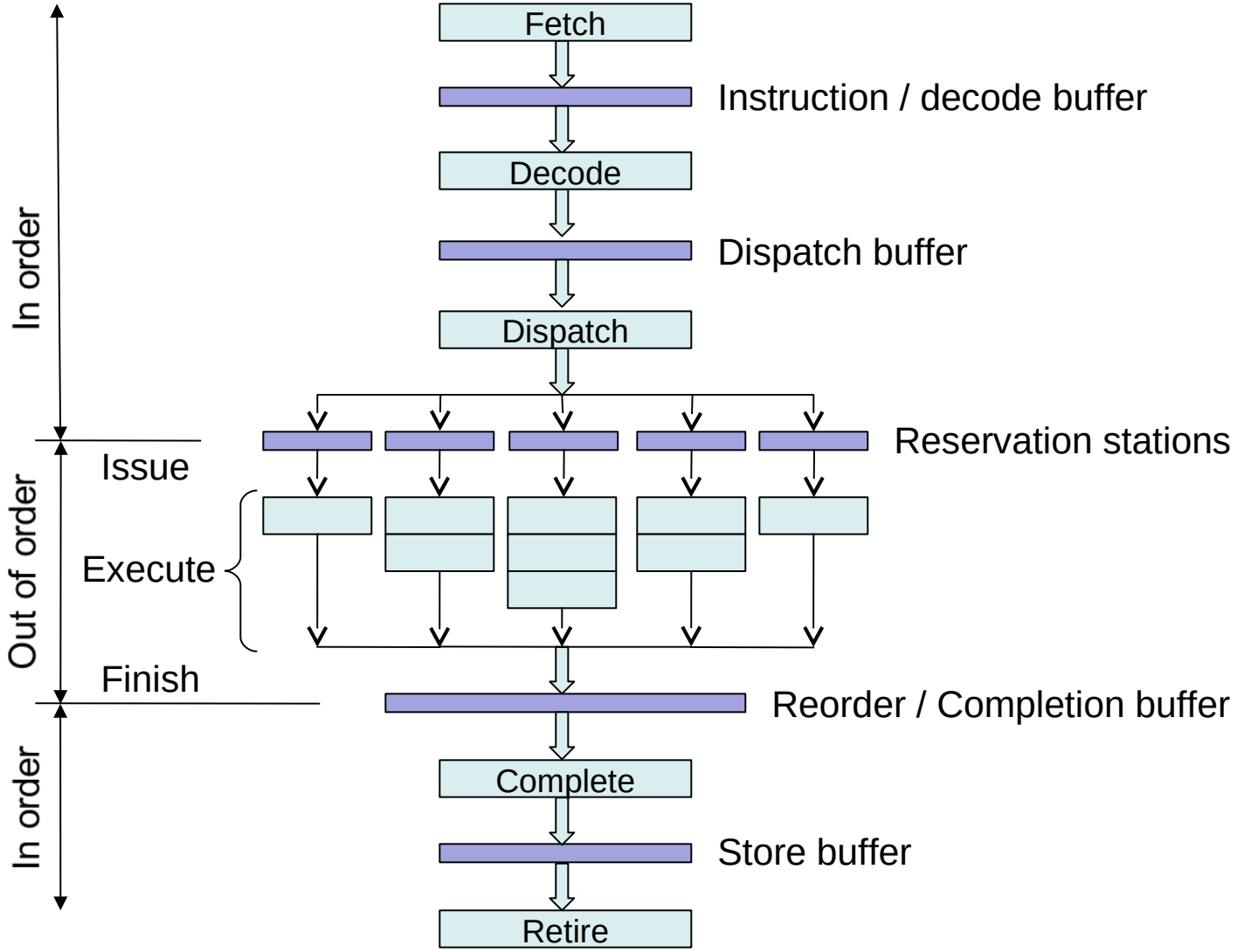
Czech Technical University in Prague, Faculty of Electrical Engineering
Slides authors: Michal Štepanovský, update Pavel Píša

Superscalar Technique – see previous lesson

- The goal is to achieve a **maximum throughput of the instruction processing**
- Instruction processing can be analyzed as instructions flow or data flow, more precisely:
  - **register data flow –** data flow between processor registers
  - instruction flow through the pipeline
  - memory data flow – to/from memory
- It roughly matches to:
  - Arithmetic-logic (ALU) and other computational instructions (FP, bit-field, vector) processing
  - Branch instruction processing
  - Load/store instruction processing
- maximizing the throughput of these three flows (or complete flow) correspond to the minimizing penalties and latencies of above three instructions types

Today's lecture topic

Superscalar pipeline – see previous lesson



Register data flow

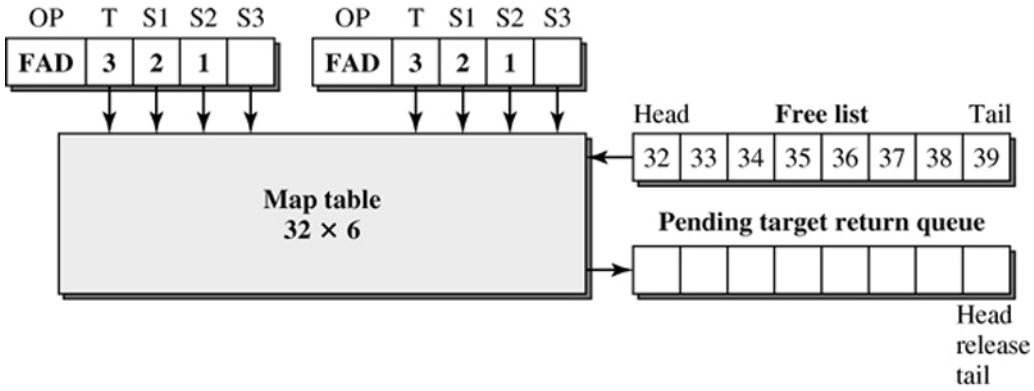
- load/store architecture – lw,sw,... instructions without other data processing (ALU, FP, …) or modifications
- We start with **register-register instruction** type processing (all instructions performing some operation on source registers and using a destination register to store the result of that operation)
- ***Register recycling*** – The reuse of registers. It has two forms:
  - **static** – due to optimization performed by the compiler during *register allocation*. In the first step, the compiler generates *single-assignment code*, supposing infinity number of registers. In the second step, due to a limited number of registers in ISA, it attempts to keep as many of the temporary values in registers as possible. The goal is to minimize moves of data to/from memory.
  - **dynamic** – at run time

```
for(i=0; i<10; i++){
        R1 = R1+R2;
}
```

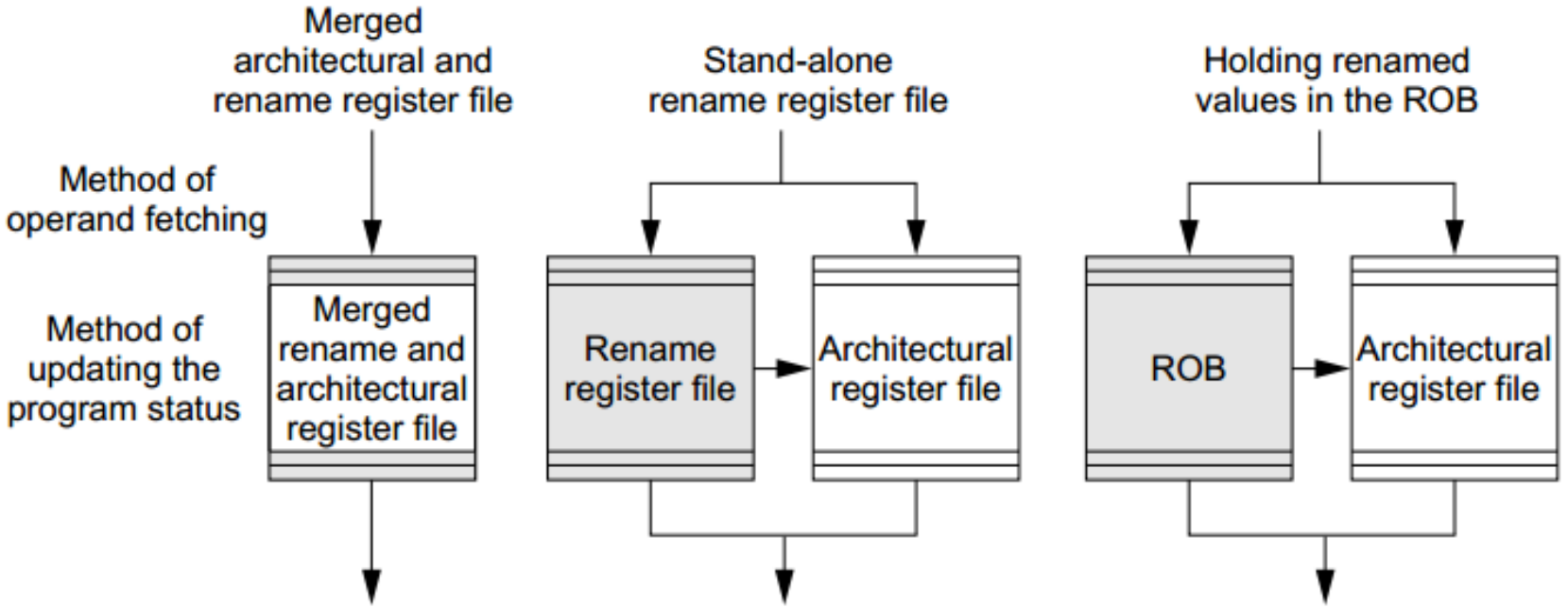
next cycle waits for the R1 update from the previous cycle. The operation cannot be dispatched to reservation station if an only the single representation of R1 exists. Solved by register renaming

Register Renaming Techniques in HW

- Register Renaming can use
  - Two separated register files – a *rename register file* (RRF) as addition to the *architected register file* (ARF)
  - Single larger registers pool – *pooled/merged/shared register file* – arbitrary register can be set as architectural (defined by ISA)
    - advantage: no copying between non-architectural and the architectural register is needed
    - disadvantage: additional HW resources are needed when compared to the first option. During context switch, we have to identify which registers correspond to architectural ones…
    - example: IBM RS/6000:



Register Renaming Techniques in HW



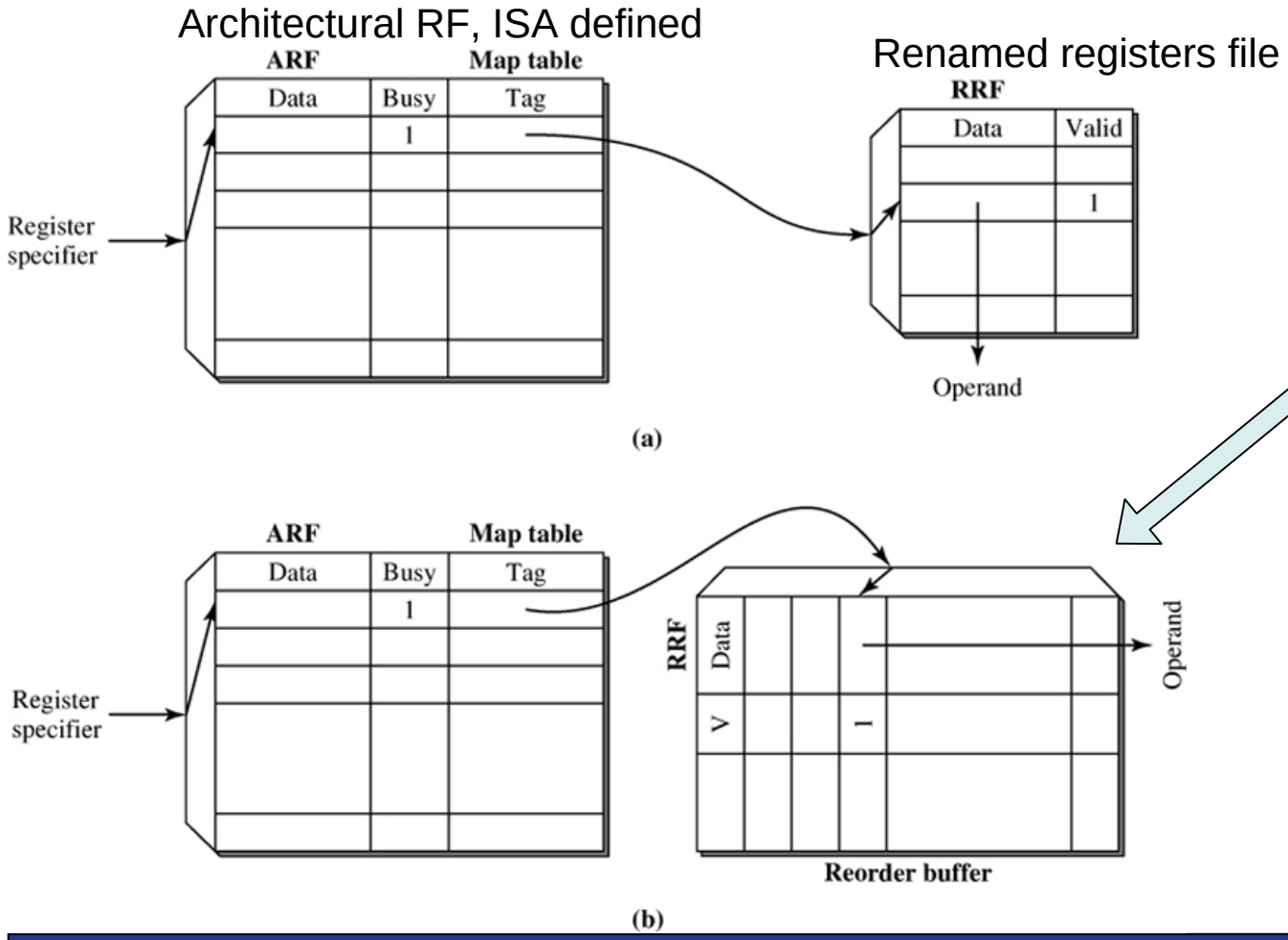
Power1 (1990)
Power2 (1993)
ES/9000 (1992)
Nx586 (1994)
PMI (Sparc64, 1995)
R10000 (1996)
R12000 (1999)
Alpha 21264 (1998)
Pentium 4 (FP) (2000)
K7 (FP) (1999)
K8 (FP) (2003)

PowerPC 603 (1993)
PowerPC 604 (1995)
PowerPC 620 (1996)
Power3 (1998)
PA 8000 (1996)
PA 8200 (1997)
PA 8500 (1999)

Am290000 superscalar (1995)
K5 (1995)
M1 (1995)
K6 (1997)
Pentium Pro (1995)
Pentium II (1997)
Pentium III (1999)
Pentium 4 (FX) (2000)
Pentium M (2003)
Core (2006)

Register Renaming Techniques in HW

- Let's look at the second and third approach from the previous slide.
- Implementation of the second case with two separated registers files



RRF as a part of Reorder buffer

Disadvantage:
- inefficient since not every instruction defines/modifies a register

Advantage:
- ROB already contains ports to receive data from functional units and to update the ARF

What we have to do?

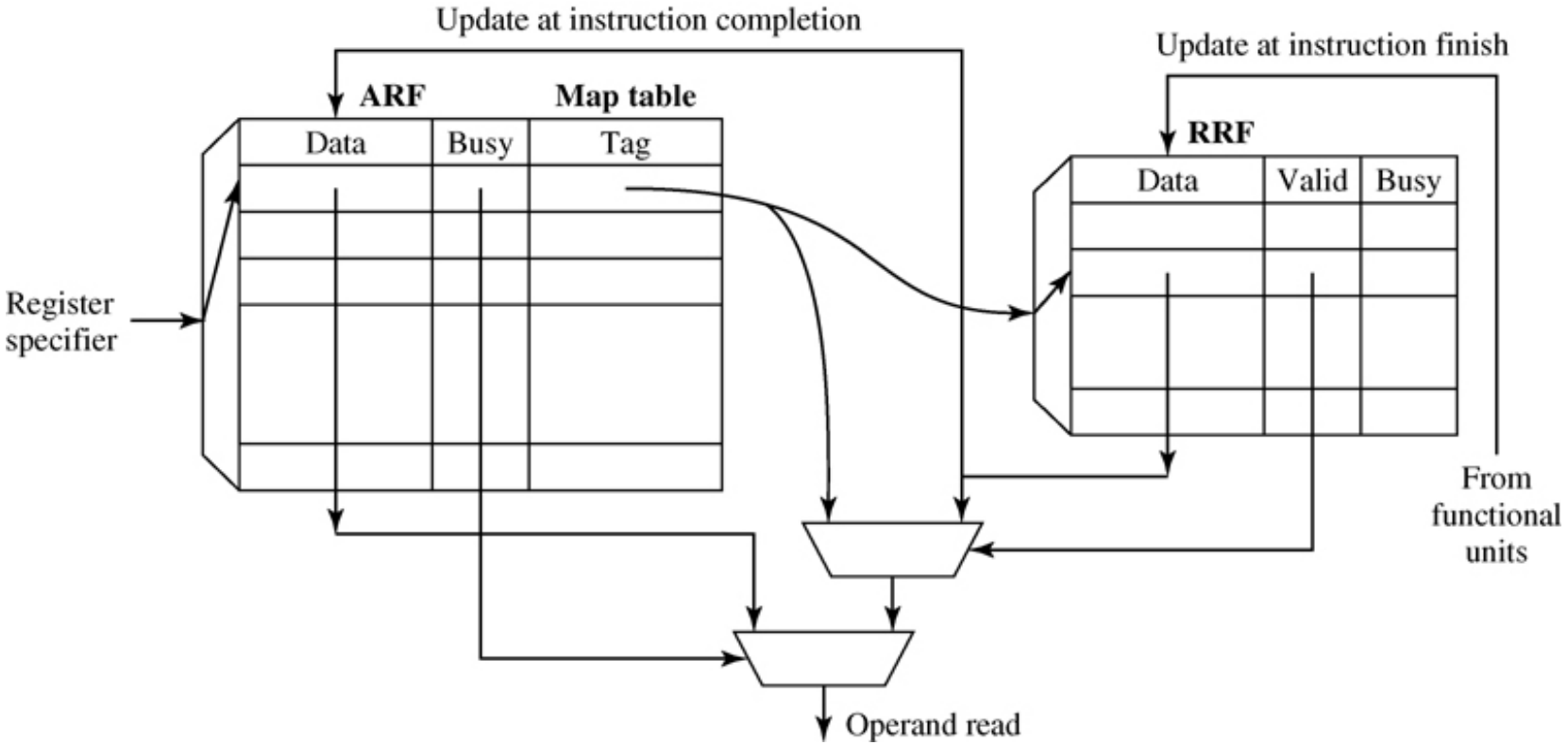
- Register Renaming involves:
  1. **Source read** (typically during decoding or dispatching phase) Possibilities:
     - The value is in architectural register – we can use it
     - The value is NOT in ARF (Busy flag is set)
       - In RRF the Valid bit is set (instruction finished execution, but still waiting for the completion) – we can use the value
       - In RRF the Valid bit is NOT set – we can NOT use the value, receiving Tag is used instead of data in place reservation station input operand
  2. **Destination allocate** (typically during decoding or dispatching phase) – it requires:
     - set Busy bit in ARF and RRF (choose free register in RRF: Busy==0)
     - Assign the Tag
     - Update the Map table

     This task is performed only for instructions writing into the register.
  3. **Register update** (next slide)

What we have to do?

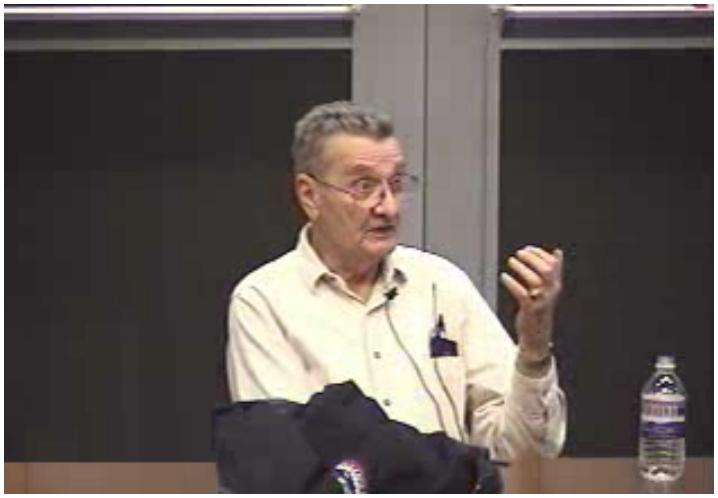
3. <u>Register update</u>

- when instruction finishes execution, the result is written into the RRF according to its tag. The ARF is updated by copy from RRF later when the instruction is completed. Corresponding RRF slot/register is deallocated.



Who is Tomasulo?

- **Robert Tomasulo**: Lecture: Out-of-order processing-History of the IBM System/360 Model 91, University of Michigan College of Engineering, 1/30/2008.

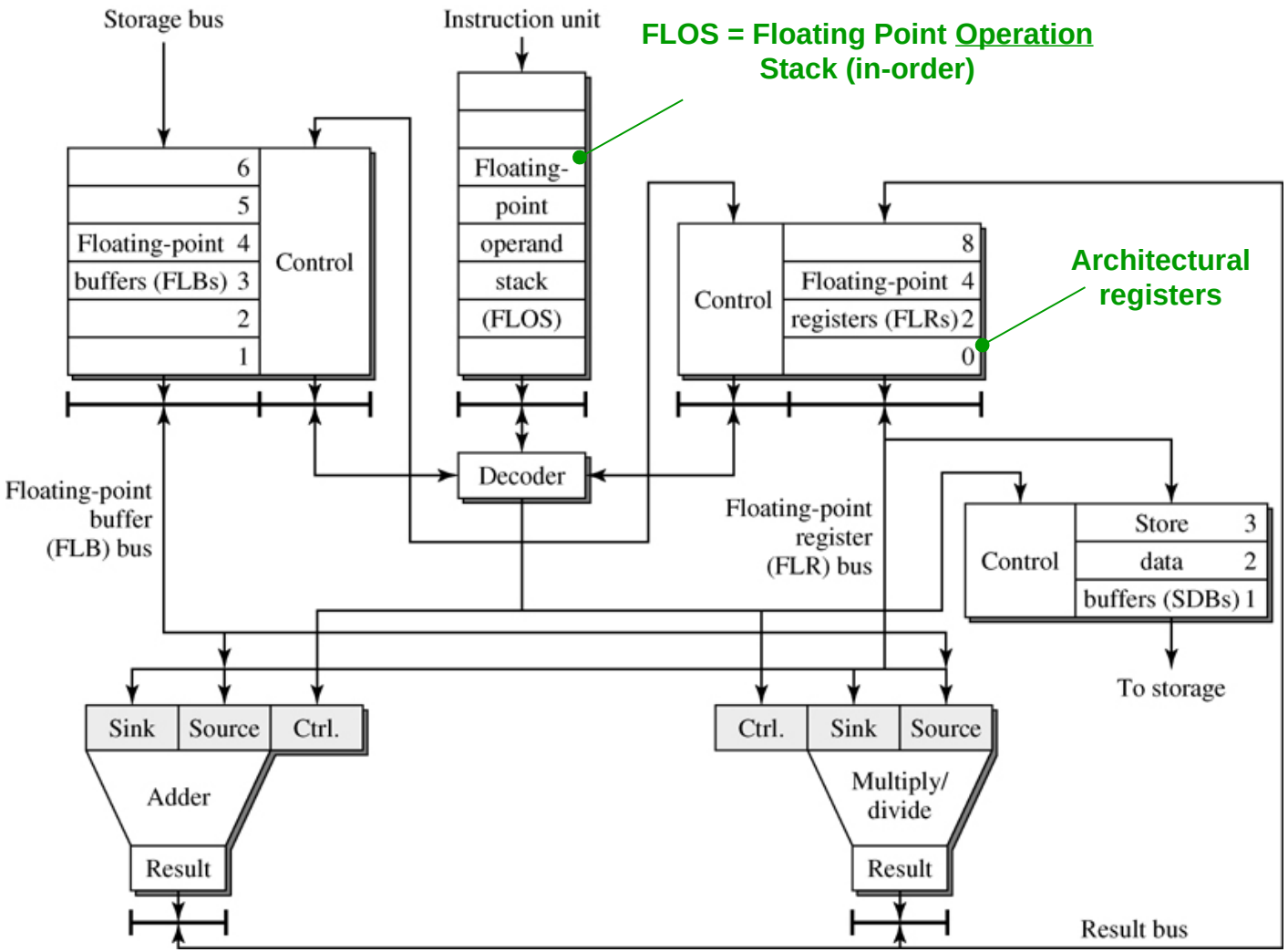


- was the recipient of the 1997 Eckert–Mauchly Award for the Tomasulo's algorithm (invented in 1967)
- Key paper: An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal of Research and Development, 11(1):25-33, January 1967.

Original IBM System/360 – the year 1964

- a clear distinction between architecture and implementation – cheaper (and slower) models, or faster (and expensive) models
- Speed from 0.0018 to 0.034 MIPS, Model 91 up to 16.6 MIPS (500x faster)
- successful in the market
- one of the most successful computers in history – **great impact on following computers design**
- The chief architect was **Gene Amdahl**
- 16 32-bit general purpose registers, 24-bits for addressing
- introduced a number of standards :
  - Why has the **byte** a fixed size of 8 bits?
  - Why every byte in the memory has their own address? (versus bit, word)
  - From where the EBCDIC is coming from? (Extended Binary Coded Decimal Interchange Code)
  - What was before **IEEE 754**-1985 and what influenced this standard?

The original design of the IBM System/360 Floating Point Unit



FP instructions:
- internally as register-register machine thanks FLB and SDB

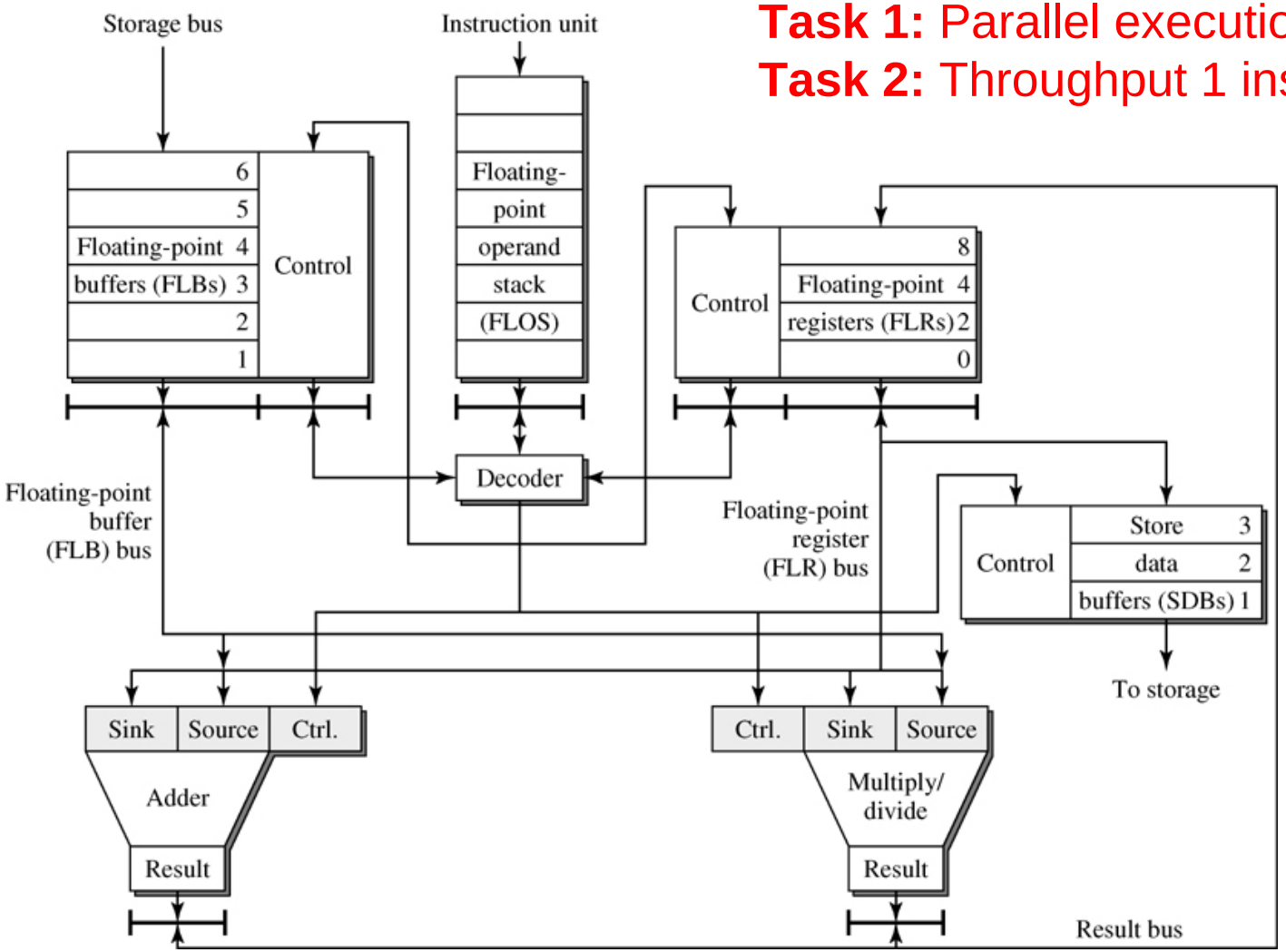
Functional units:
- nonpipelined
- Adder: 2 cycles
- Mult/Div: 3 or 12 cycles

Result bus:
- used for architectural registers update

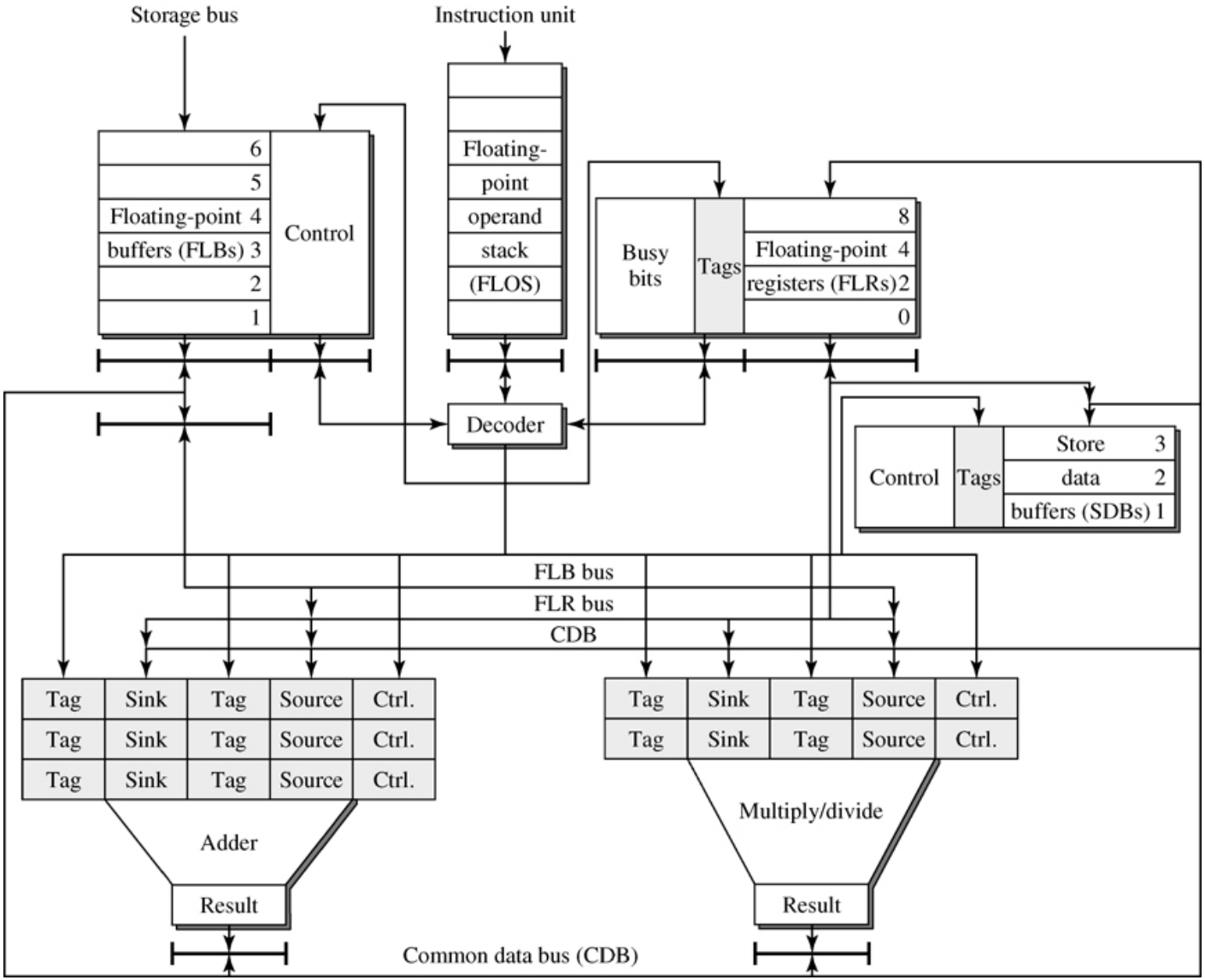
X = X + A;
Y = Y/X;

Imagine to be an IBM designer !!!

Task 1: Parallel execution in both units
Task 2: Throughput 1 instruction/cycle



Solution…


Storage bus
Instruction unit
6
5
Floating-point 4
buffers (FLBs) 3
2
1
Control
Floating-
point
operand
stack
(FLOS)
Busy
bits
Tags
8
Floating-point 4
registers (FLRs)2
0
Decoder
Control
Tags
Store 3
data 2
buffers (SDBs) 1
FLB bus
FLR bus
CDB
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Adder
Result
Tag Sink Tag Source Ctrl.
Tag Sink Tag Source Ctrl.
Multiply/divide
Result
Common data bus (CDB)

The Classic Tomasulo Algorithm

- Instruction has two operands. The first one is source and destination simultaneously. Nevertheless, let's consider following program with two sources and a separate destination:

```
w:  R4 = R0 + R5      // R0 == 6.0,   R5 == 7.8
x:  R2 = R0*R4
y:  R4 = R4 + R5
z:  R5 = R4*R2
```

The initial state of architectural registers

FLR - architectural reg.

	Busy	Tag	Data
0			6.0
1			
2			
3			
4			
5			7.8

Cycle 1.:

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
	2				
	3				
w-start	Adder				

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
	5				
	Mult / Div				

FLR - architectural reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	1	
	5			7.8

Instructions are dispatched in order!

First, instruction w: Read R0, Read R5 (data are valid); in R4 Set Busy and Tag (Tag=Reservation Station Number)

Second, instruction x: Read R0 (success), Read R4 (bit Busy is set, so Tag is used instead); in R2 set Busy and Tag

The Classic Tomasulo Algorithm

w: R4 = R0 + R5
x: R2 = R0*R4
y: R4 = R4 + R5
z: R5 = R4*R2

Cycle 2:
- instruction y: Read R4 (Tag==1 is read instead of data), Read R5 (data are valid); in R4 set Busy and Tag (Tag=Reservation Station Num of this instruction == 2)
- instruction z: Read R4 (Tag==2), Read R2 (Tag==4); in R5 set Busy and Tag
... on the end of the cycle, Adder produces the result and propagates it (together with its Tag) on the Common Data bus. Reservation Stations updates their Data (Tag match)

Cycle 2.:

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	---	0	7.8
	3				
w-finishin		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
z	5	2	---	4	---
		Mult / Div			

FLR - architectural reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

instruction w finishes execution and broadcasts its ID (Tag = reservation station 1) and its result into CDB 6.0+7.8=13.8

Cycle 3.:

	RS #	Tag	Sink	Tag	Source
w	1				
y	2	0	13.8	0	7.8
	3				
		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	---	4	---
		Mult / Div			

FLR - architectural reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

Deallocation of reservation station #1

How will it continue? Try at home...

The Classic Tomasulo Algorithm

- The Classic Tomasulo Algorithm in this form **does not** support the precise exception
- **Why?**

- Notes (Classic Tomasulo Algorithm):
  - FLR holds the Tag (ID) of that reservation station, which is responsible for producing the result.
  - Whenever any functional unit finishes execution (the result is known), the result has to be propagated together with Tag on CDB (Common Data Bus). All remaining reservation stations and FLR have to snoop for Tag match and eventually update their data.

Precise exception/Interrupts – general view

- What is an exception? What is an interrupt?
- An **interrupt** is an event emitted by hardware or software indicating the needs of immediate attention of CPU. The processor interrupts the sequential execution semantics, stores at least part of its state, it invokes interrupt service routine and then returns to continue interrupted instruction sequence if the cause is resolved.
- **Hardware interrupt** – asynchronous from an external device
- **Exception (Software interrupt) –** caused by an unexpected exceptional condition during instruction processing (div by 0), or by undefined, special (syscall, …) instruction, failed memory access, etc. => internal interrupt, synchronous
- Precise Exception
  - Exception in classic von Neuman computer (sequential uniprocessor) is always precise => well-defined state before calling an interrupt handler, in case of exception corresponds to the sequential state before causing instruction

Exceptions/interrupts examples

- **I/O Device Request** caused by the need of external event processing (character, packed, … arrival, sending buffer released, …) – asynchronous state fully restored
- **Supervisor Call SVC** – request service from OS kernel – initial CPU state is used as arguments and returned one is modified to deliver results and success/fail/error code indication
- **Breakpoint** – usually invokes debugger which can modify a state
- **ALU/FP exceptions** (arithmetic overflow/underflow FP, division by zero, …) – usually invoke signal handler in program context or aborts program
- **Page Fault** – is resolved by OS (full state restore) or signal/abort
- **Misaligned Memory Access** – OS resolved or signal/abort
- **Undefined Opcode –** emulated by OS or signal/abort
- **Error in HW** (parity, ECC, low voltage,…)

Exceptions/interrupts classification examples

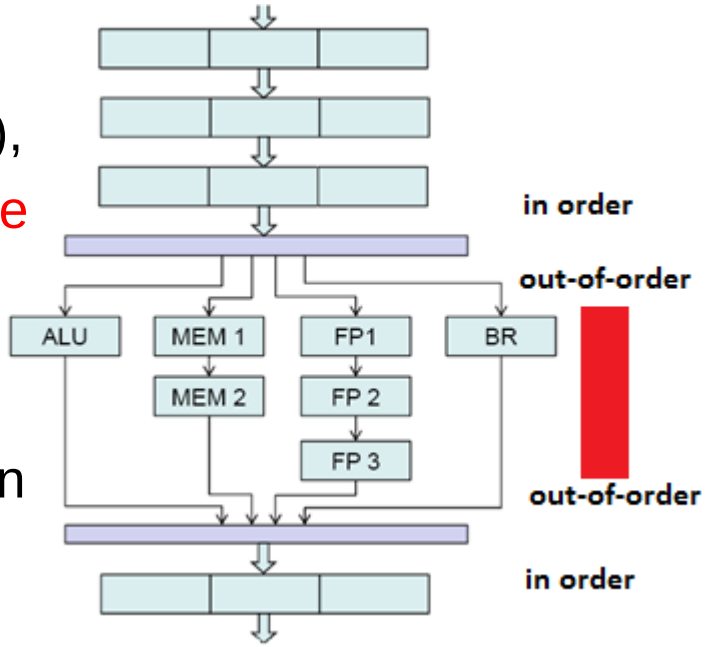
- Internal vs. external – sometimes distinguished as interrupt vs. exception, sometimes (by some ISA or manufacturer) interrupt or exception used for both cases
- Hardware vs. software
- Synchronous vs. asynchronous
- User invoked (predictable) vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume vs. terminate

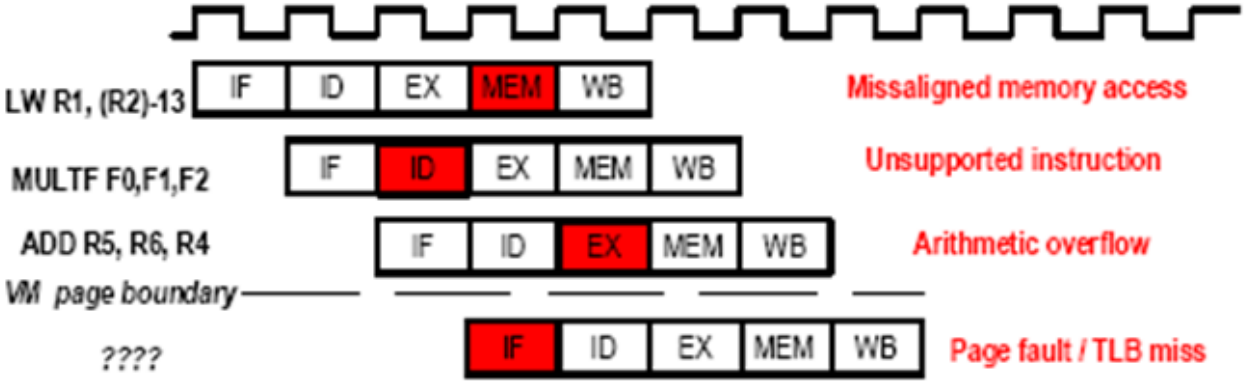
Exceptions/interrupts classification table

Exception type	Synchronous	User requested	Maskable	Within instructions	Resume
I/O device request	No	No	OS	No	Yes
Invoke operating system	yes	Yes	No	No	Yes
Tracing instruction execution	yes	Yes	No	No	Yes
Breakpoint	yes	Yes	OS	No	Yes
Integer arithmetic overflow	yes	No	User	Yes	Yes
Floating-point arithmetic overflow or underflow	yes	No	User	Yes	Yes
Page fault	yes	No	No	Yes	Yes
Misaligned memory accesses	yes	No	No/ OS	Yes	Yes
Memory protection violation	yes	No	No	Yes	Yes
Using undefined instruction	yes	No	No	Yes	No
Hardware malfunction	No	No	No	Yes	No
Power failure	No	No	No	Yes	No

Exceptions processing in the pipelined processor

- Can arise in every cycle,
- More exceptions at once (different sources),
- Ordering of exceptions according to the time may not correspond to program order of instructions,
- Processor state may not be consistent with sequential semantic of program execution in every cycle.





Two exceptions arise in the same cycle. Up to four can appear simultaneously for this example! Solution?

Is it a **precise** exception? Yes, iff

- 1. Processor handles exceptions in program order
  - Not in the order as they arise in the pipeline
- 2. Processor state is sequentially consistent before calling exception handler
  - All instructions before the source of exception are completed (or retired) and no instruction after the source of exception changed the processor state or the state of the memory

The simple way how to resolve precise exception – Commit

- **Commit stage -** is the last stage in the pipeline in which the exception can arise
- After **Commit** – the instruction will not generate the exception

Conditions for precise exception:

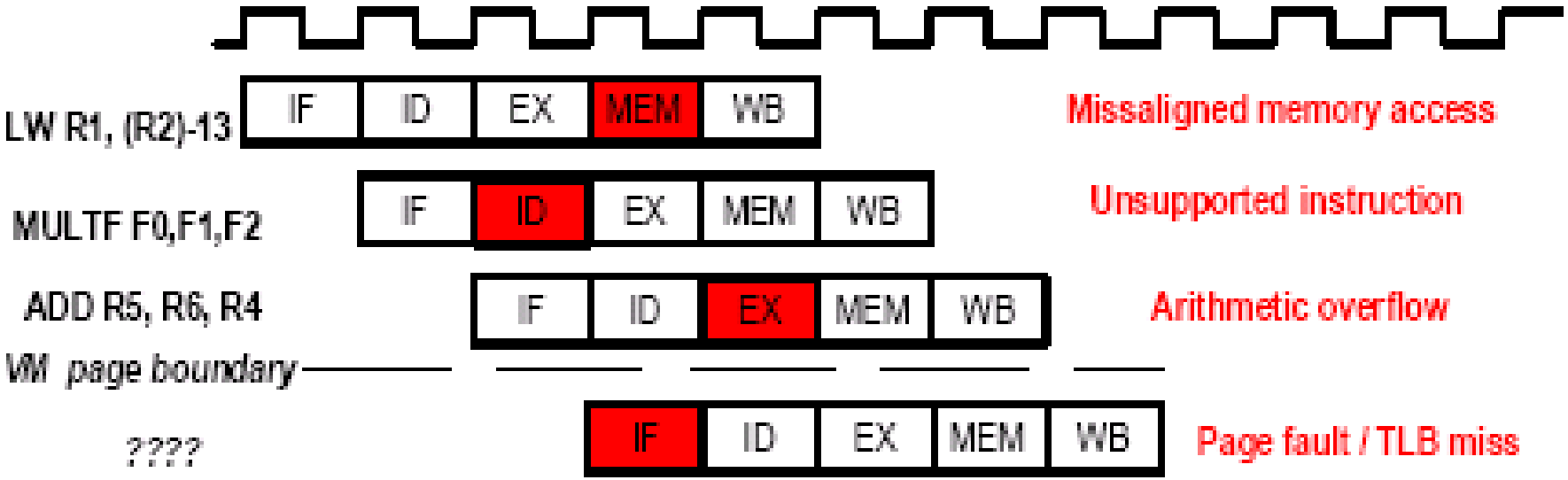
- All instructions arrive into the Commit in program order
- No instruction is allowed to change processor state or memory before Commit

Implementation of precise exception handling

- Requests (exceptions) are accumulated in hardware and propagated (with instruction corresponding Program Counter) into the Commit stage
- Processor tests every instruction in the Commit stage whether the exception was introduced, exceptions within instruction are serviced in order of their arising.

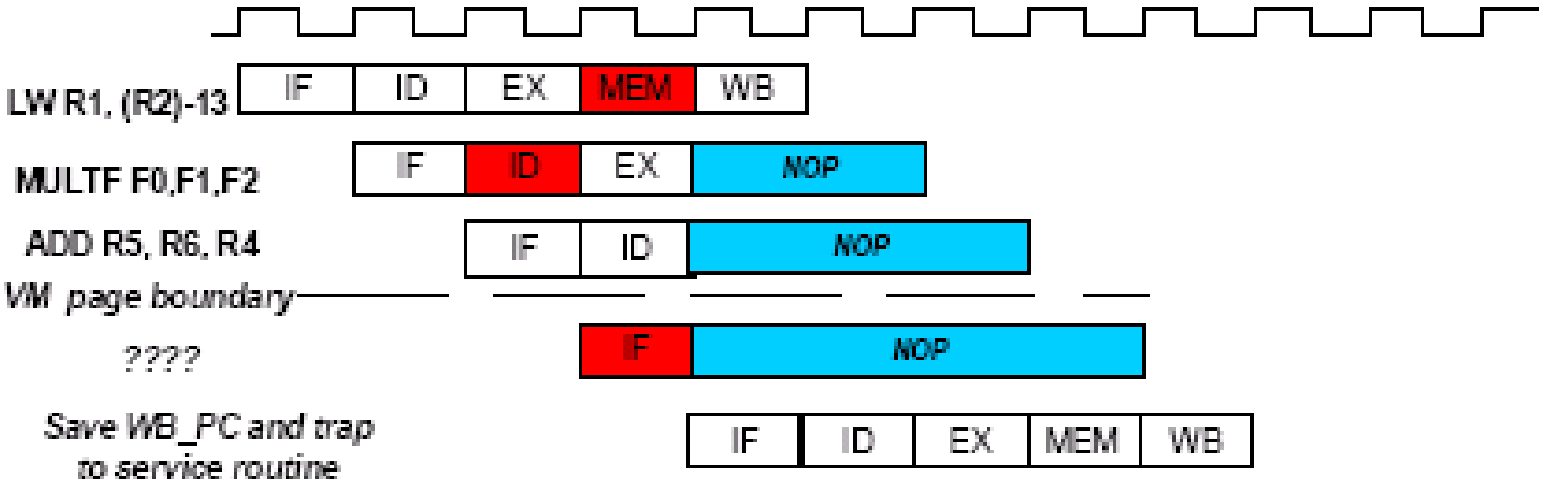
  For recoverable exceptions, saved architectural state correspond to the state before causing instruction and after OS resolve cause, the instruction is reissued (restored PC points to the causing instruction)
- All instructions in the pipeline before Commit stage are discarded and all instruction already passing Commit stage stage are finished/retired

What is the Commit stage in the pipeline example?



If MEM stage is chosen as **Commit stage** for this processor then …

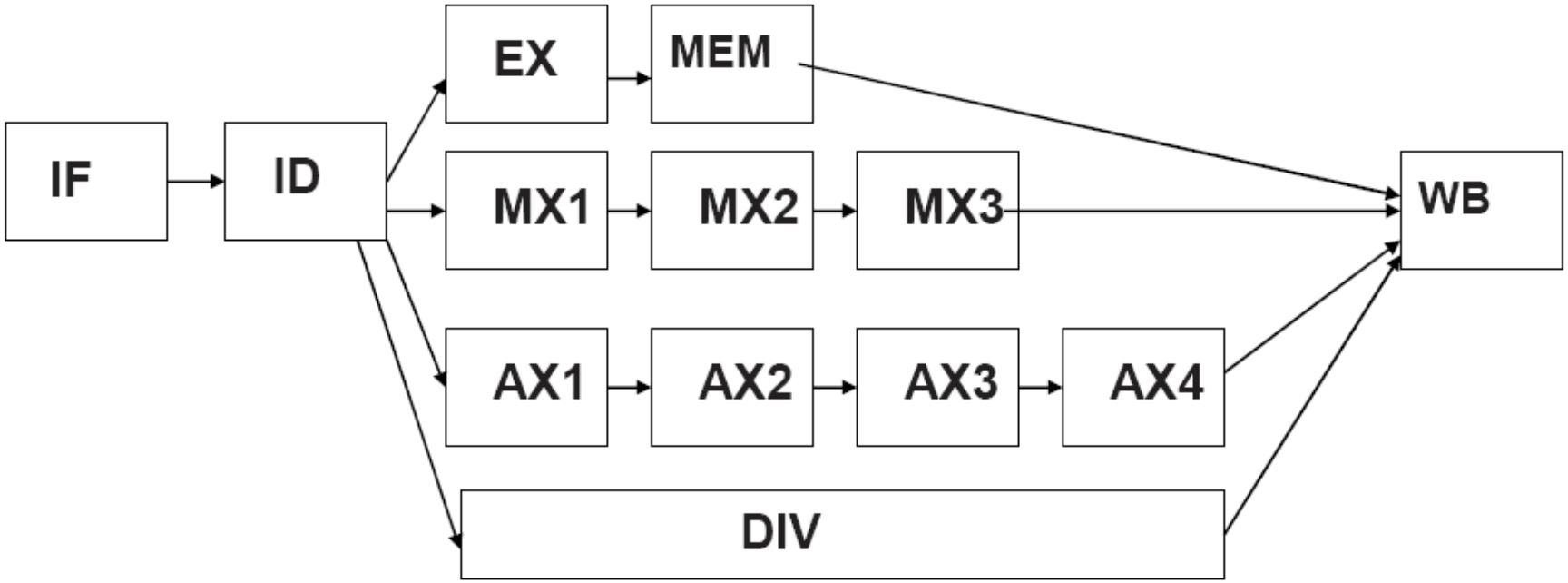
What is the Commit stage in this pipeline?



- Yes, it is MEM. This processor supports precise exceptions because:
  - All instructions arrive into the MEM in program order (all instructions need 4 cycles to reach the MEM and instructions are fetched and processed in program order)
  - Processor state (register files) are written in WB, memory in MEM, there is no state change before the MEM stage
  - (exception hander in commit)

Problem solved?

- No, there are more complicated CPU implementations
- Different latencies of parallel pipelines and out-of-order instructions processing ... (superscalar CPU)
- Return to the Tomasulo algorithm …



Recall the Classic Tomasulo Algorithm – reached final state

w: R4 = R0 + R5 // R0 = 6.0, R5 = 7.8

x: R2 = R0*R4

y: R4 = R4 + R5

z: R5 = R4*R2

Cycle 2.:

	RS #	Tag	Sink	Tag	Source
w	1	0	6.0	0	7.8
y	2	1	---	0	7.8
	3				
w-finishin		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	1	---
z	5	2	---	4	---
		Mult / Div			

FLR - architectural reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

instruction w finishes execution and broadcasts its ID (Tag = reservation station 1) and its result into CDB 6.0+7.8=13.8

Cycle 3.:

	RS #	Tag	Sink	Tag	Source
w	1				
y	2	0	13.8	0	7.8
	3				
		Adder			

	RS #	Tag	Sink	Tag	Source
x	4	0	6.0	0	13.8
z	5	2	---	4	---
		Mult / Div			

FLR - architectural reg.

		Busy	Tag	Data
	0			6.0
	1			
x	2	yes	4	
	3			
w	4	yes	2	
z	5	yes	5	7.8

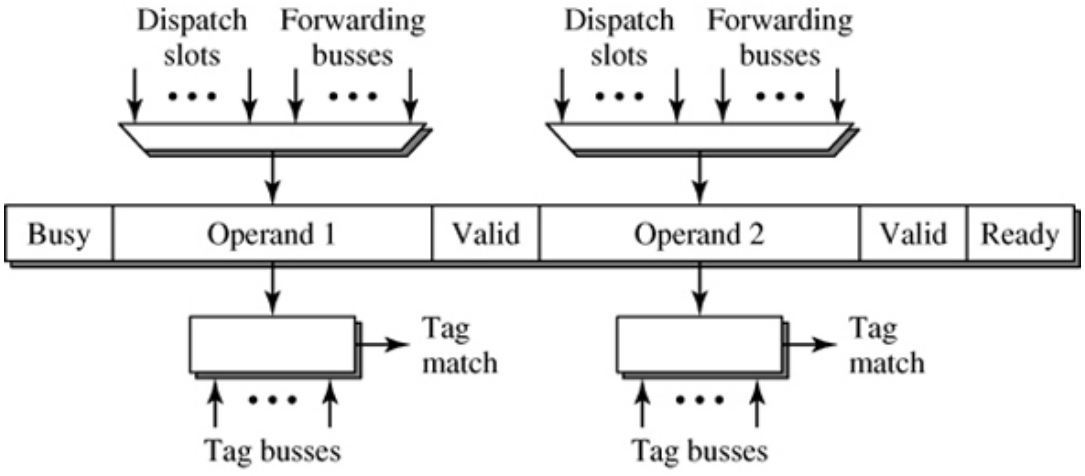
Deallocation of reservation station #1

How will it continue? Try at home...

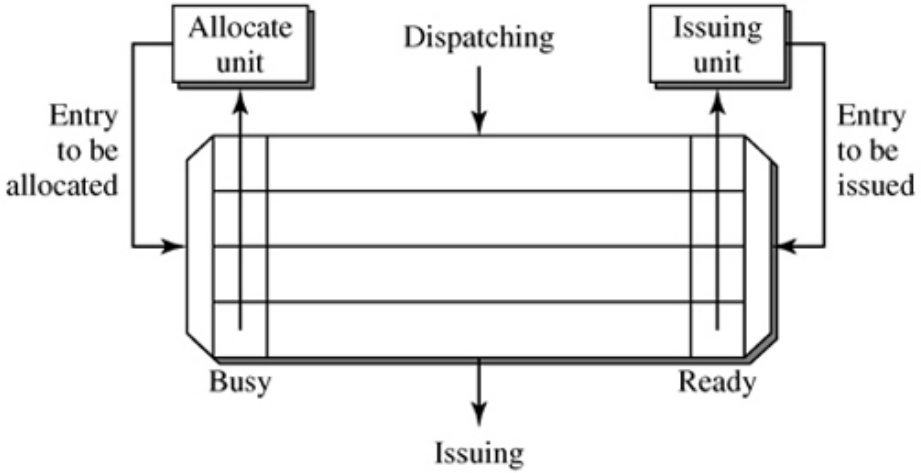
Improved Tomasulo Algorithm

- **Issue / Dispatch**: Load instruction (in program order) from instructions queue (prefetch buffer)
  - If a reservation station (RS) is free and Reorder Buffer (ROB) is not full (structural hazard detection), dispatch instruction into RS and allocate the entry in ROB
  - Actions: 1. instruction decode, 2. RS and ROB entries are allocated, 3. destination register rename read values/tags from RF, 4. dispatch decoded instruction to RS and ROB
- **Issue:**
  - Issue to the functional unit when all source operands are available, if not snoop on the common data bus for missing operands and wait in RS
- **Execute** (EX): functional unit processes operands and compute result
- **Write result**: finish execution
  - Write result &Tag on the common data bus to RS and ROB, deallocate RS
- **Commit**: Architectural register (or memory) updated from ROB
  - If the oldest instruction (at the head) in ROB has a result (is executed), complete (or retire) the instruction = update register with the result in reorder buffer or do a store; remove instruction from reorder buffer (free ROB slot)

Reservation station. How does it look?



(a)



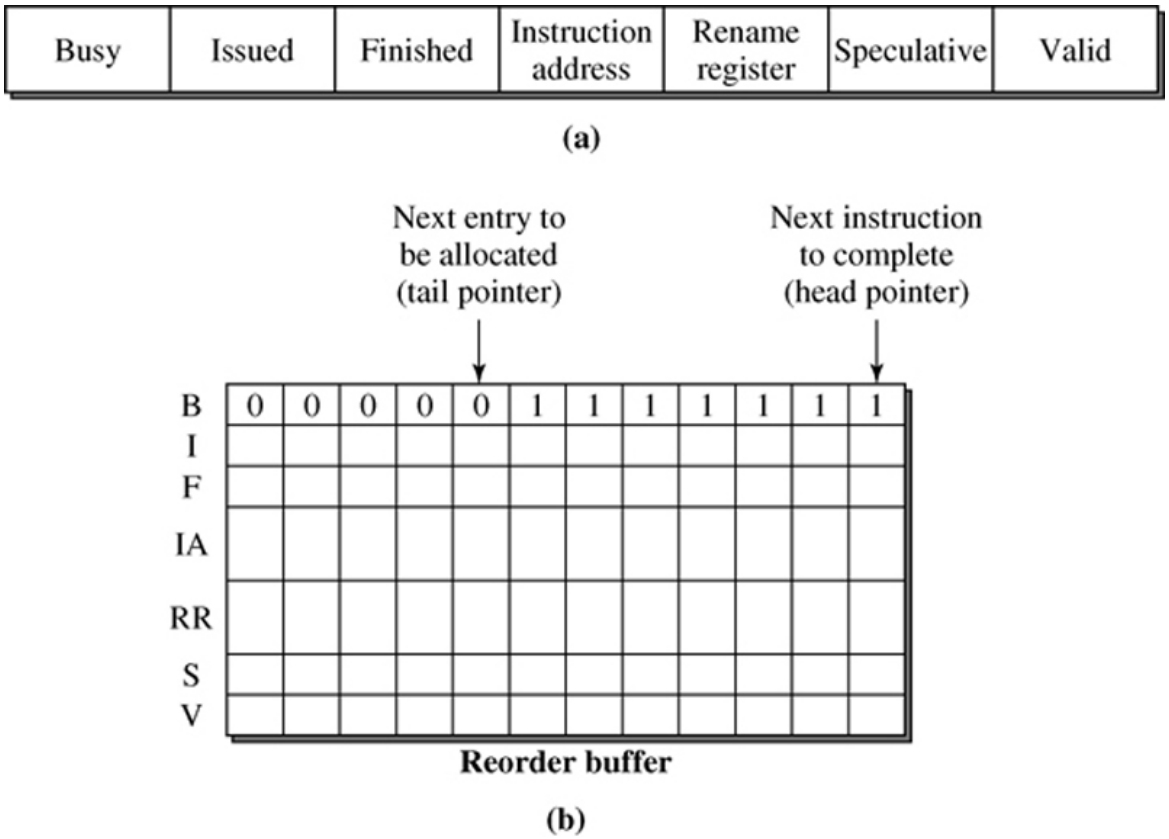
(b)

Reservation station entry with required match logic monitoring tag buses (can be CDB, ROB, …) for input operand matching tags when valid=0. In match case operand value is stored and set valid=1, when both operands valid set ready=1

Notes:

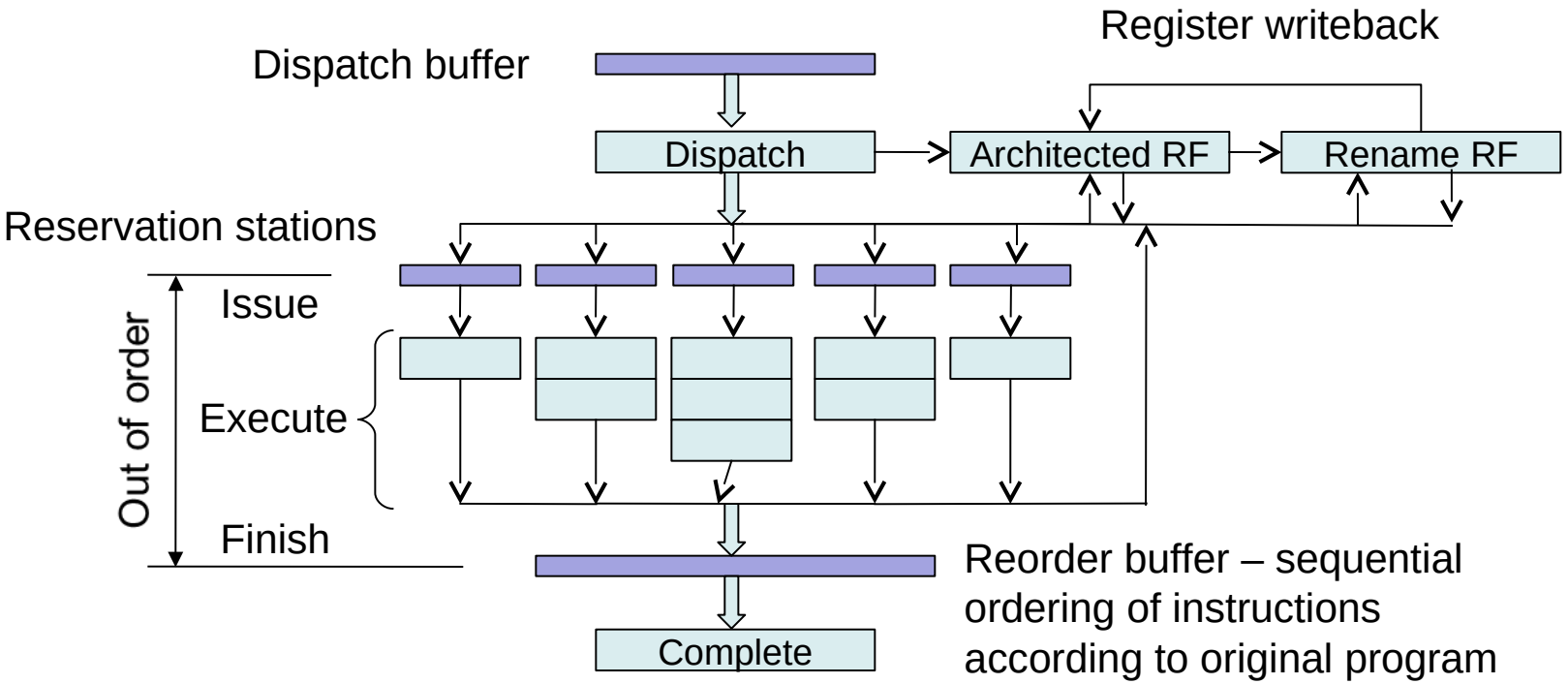
- Busy – allocation
- Valid – operands update
- Ready – instruction wake up
- Issue – instruction select

Reorder buffer. How does it look?



- Contain all *in-flight* instructions.. (dispatched, not completed)
- Circular queue..
- Rename register file (slide 7)

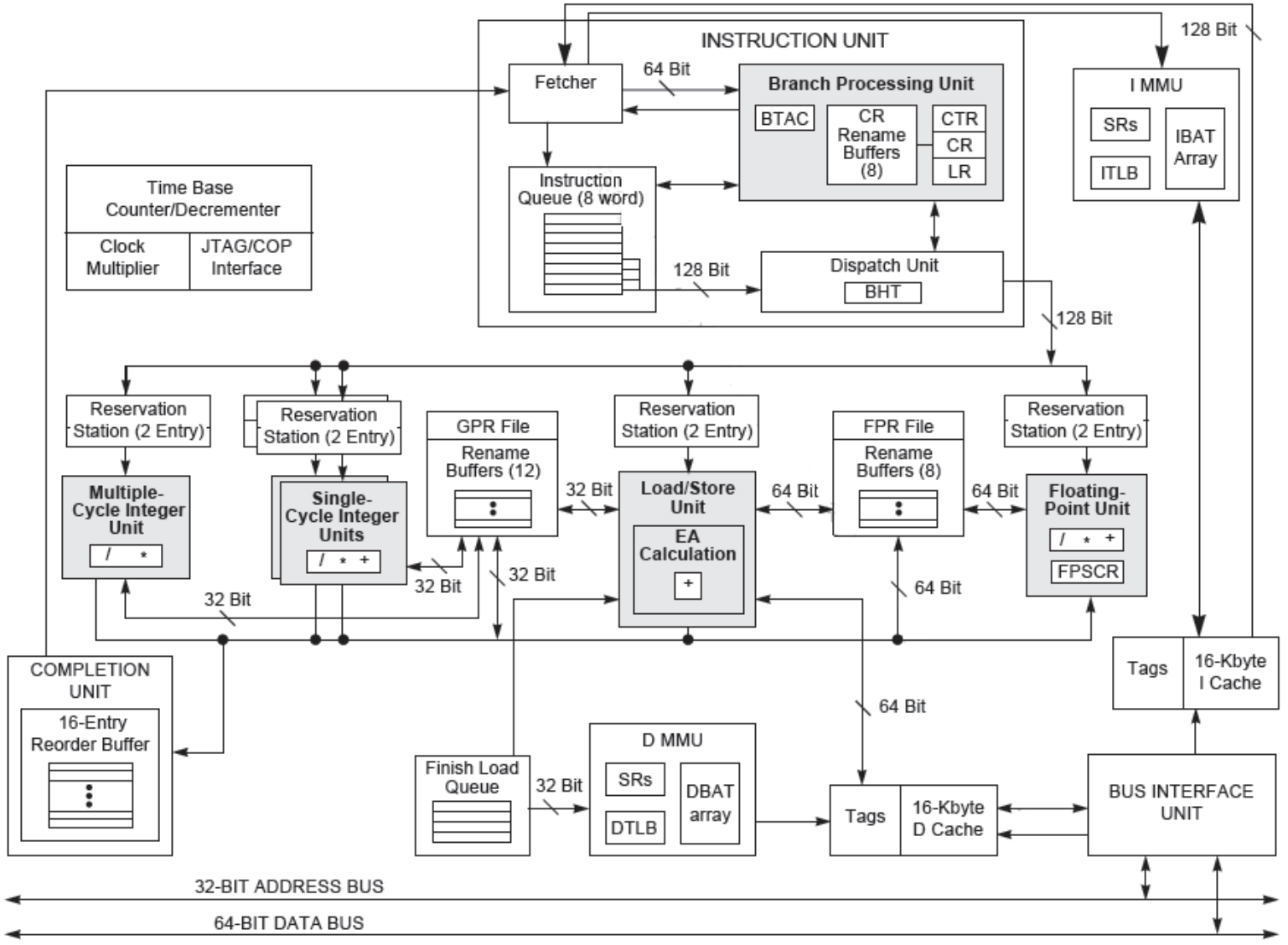
The Improved Tomasulo Algorithm



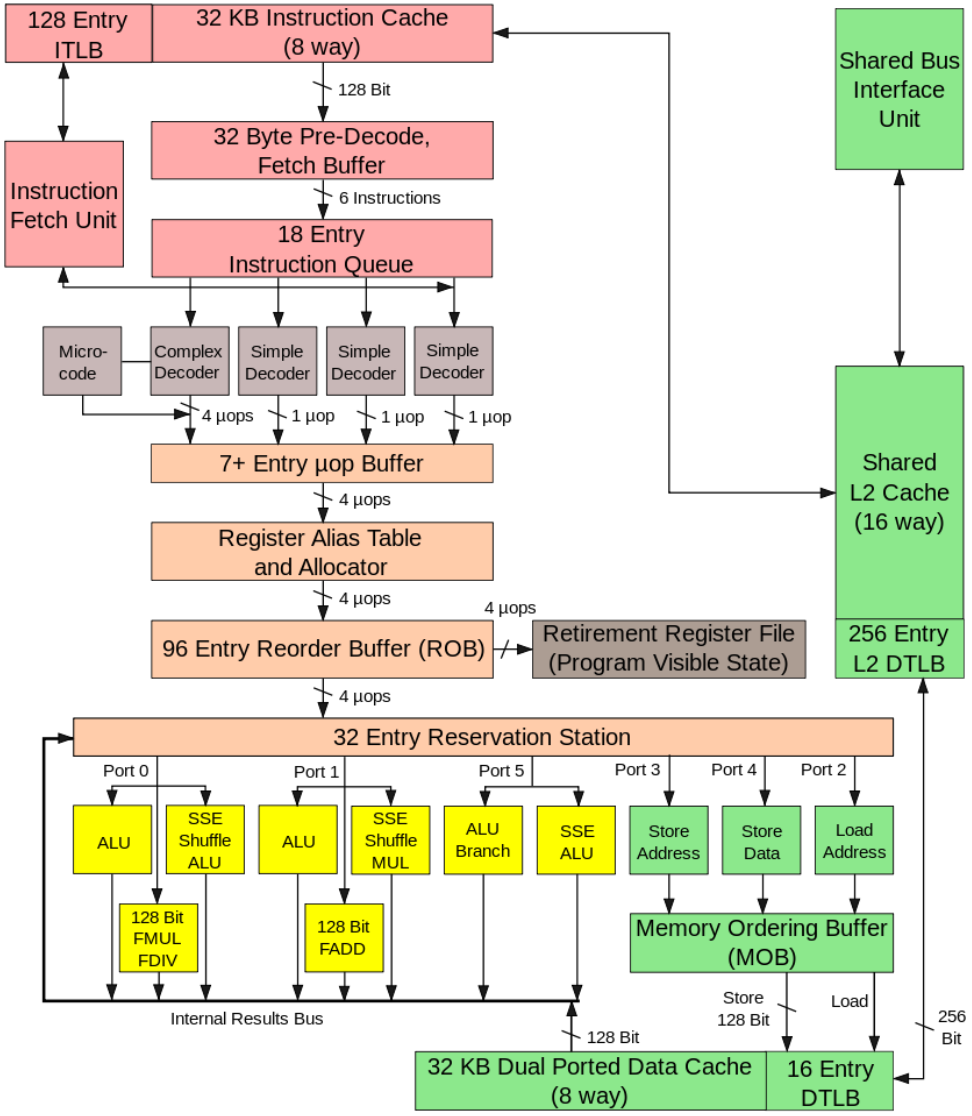
Three approaches (register renaming support):

- Merged register file (any register may be an architectural one)
- Architected register file + Rename register file (RRF), where RRF:
  - standalone
  - part of ROB (see slides 4 - 8)

PowerPC 604… The year 1994



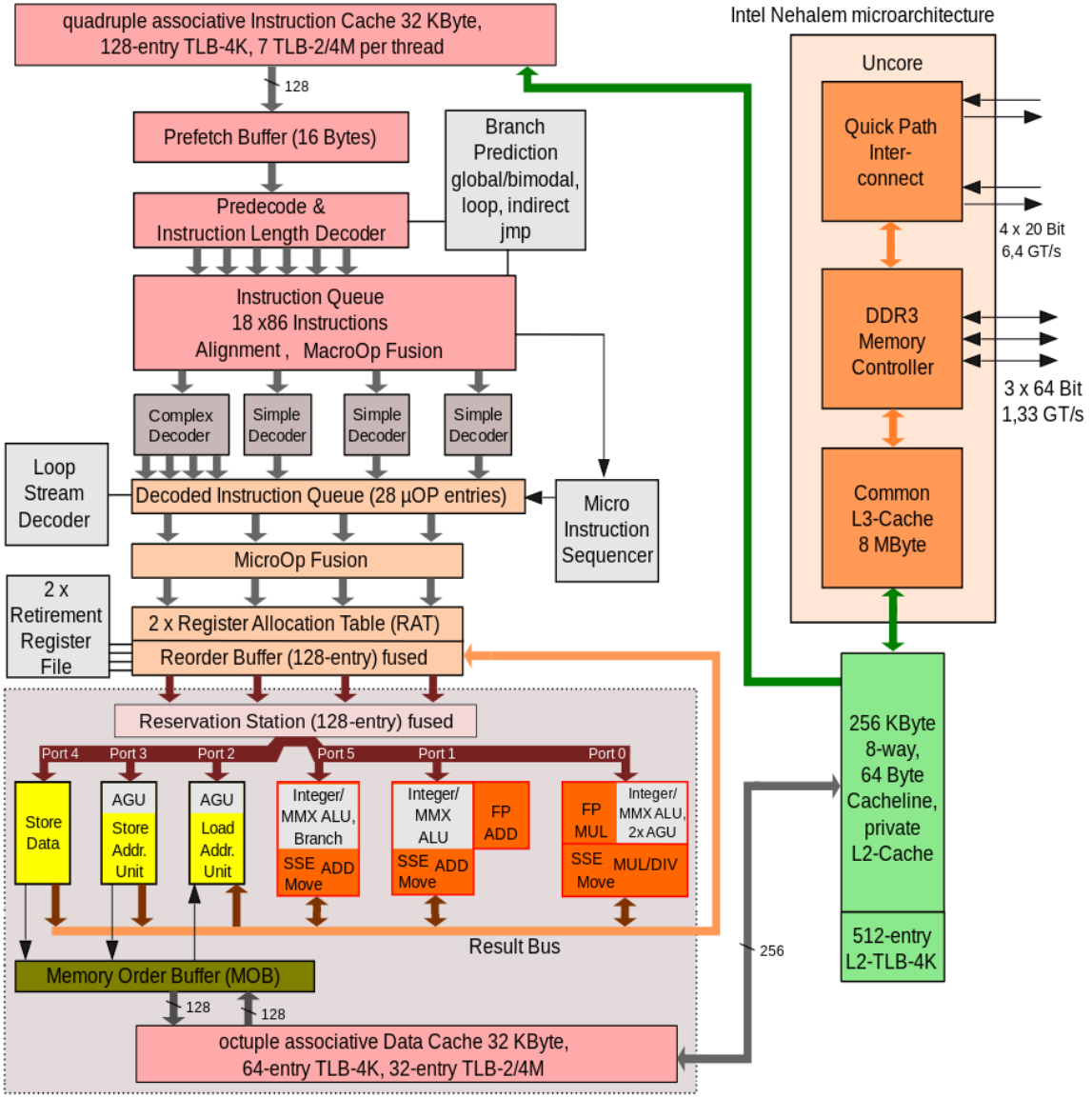
Intel **Core** microarchitecture… The year 2006



Intel Core 2 Architecture

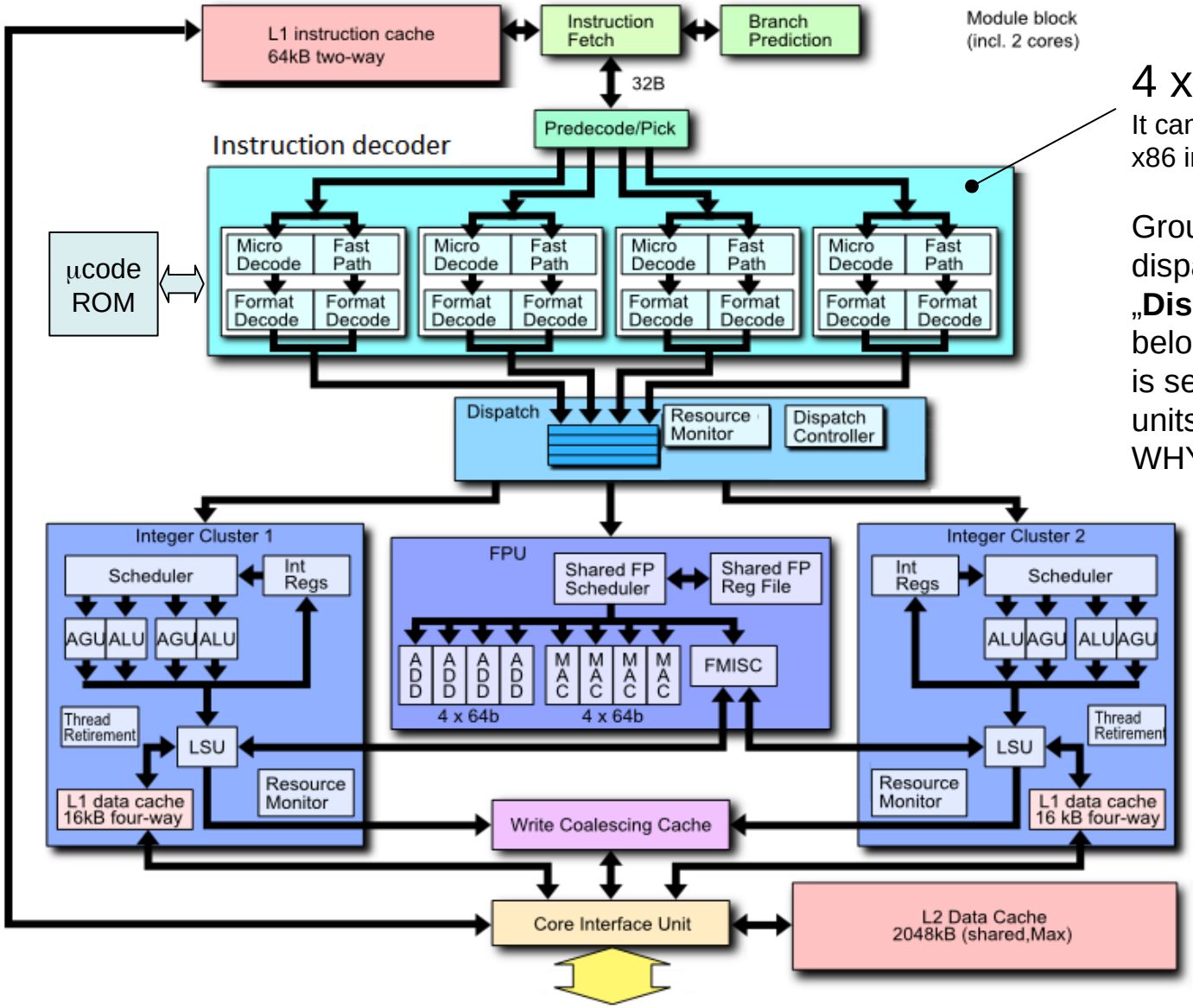
http://en.wikipedia.org/wiki/Intel_Core_(microarchitecture)

Intel **Nehalem** (Core i7) - 2008



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

AMD **Bulldozer** 15h (FX, Opteron) - 2011



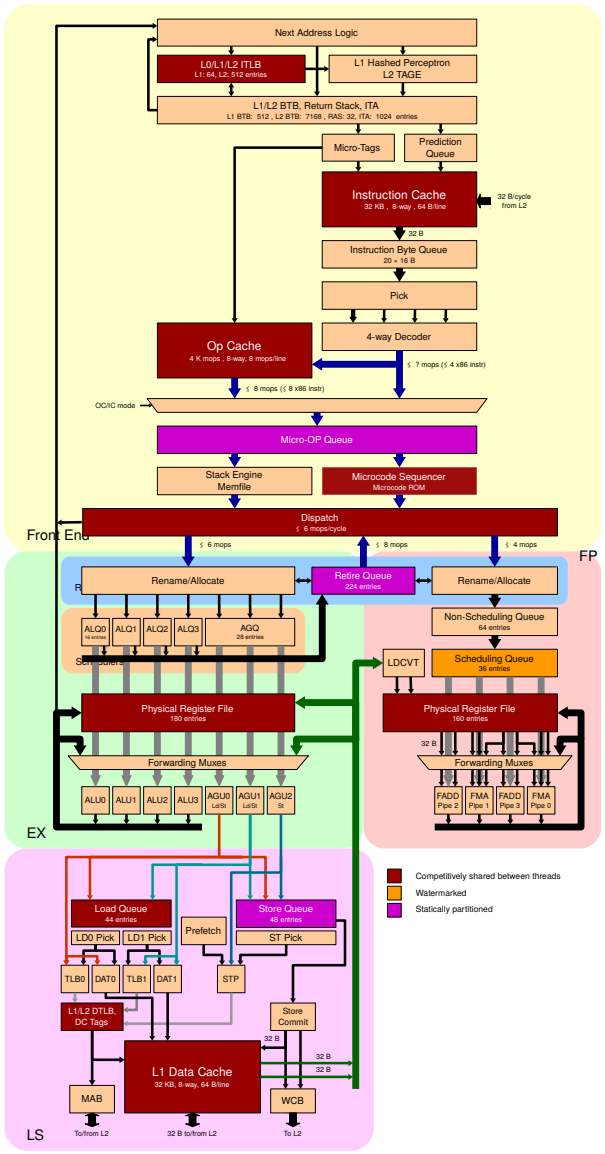
4 x86 decoders…

It can fetch and decode up to four x86 instructions per clock.

Group of µops, which are dispatched together -> „**Dispatch group**“ – always belongs to only one thread and is sent to one of two integer units or to FPU as a whole. WHY?

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

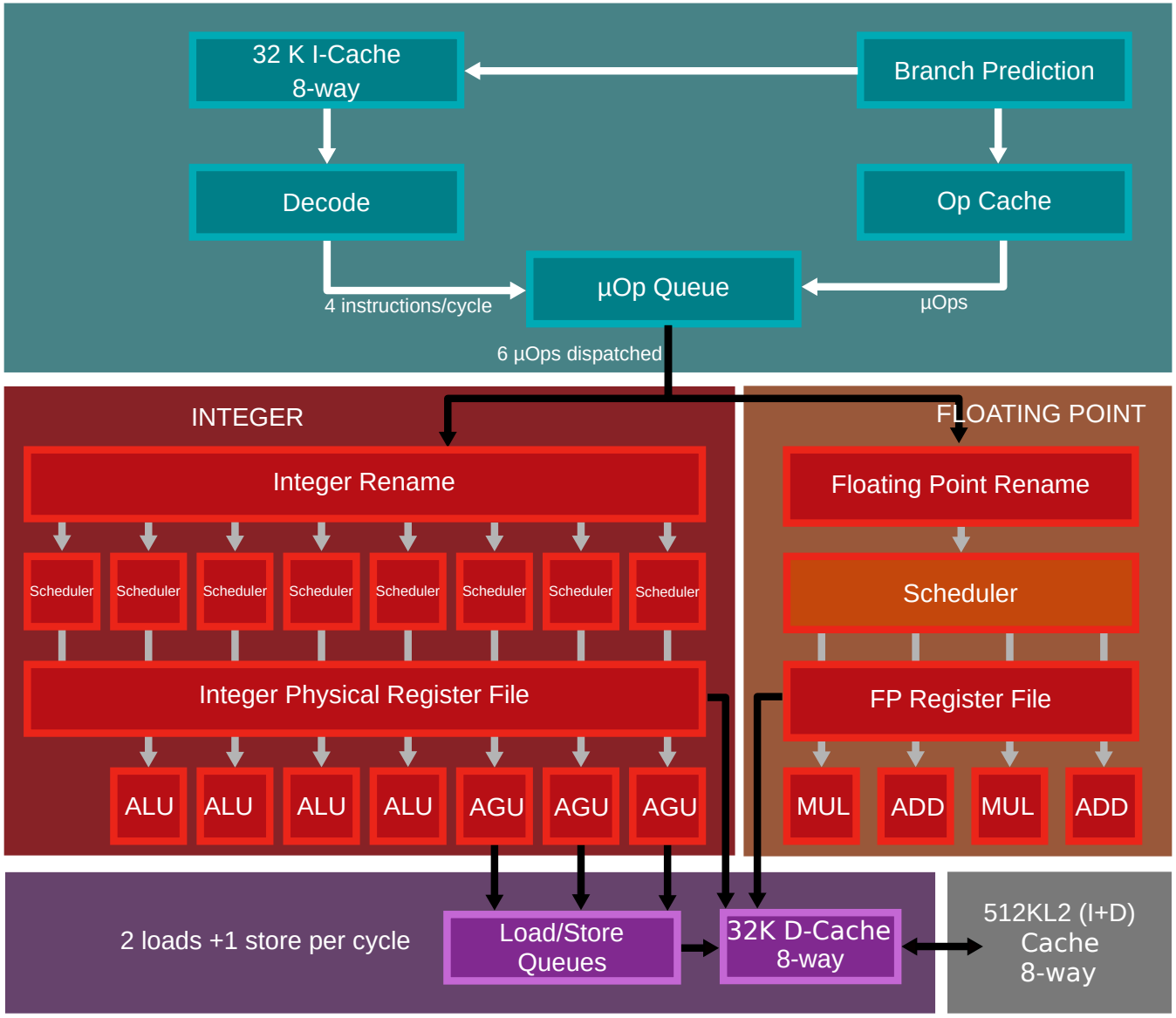
AMD **Zen 2** - Microarchitecture



- 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



What is important …

- “All” modern processors implement some form of register renaming (based on Tomasulo algorithm) to remove anti-dependencies and output dependencies (WAR, WAW)
- Register renaming is typically realized during decoding phase and may require the renaming of registers in all instructions in the **fetch group** (dependencies may arise not only in relation to dispatched, issued and executed instructions, but also between instructions in the fetch group)
- Register renaming removes WAW and WAR dependencies, what is the key difference between **scoreboarding**, which only monitors data dependencies and allows to schedule instructions out-of-order only when all dependencies have been resolved (otherwise: stall)

Additional notes

- Slots have to be allocated in reservation station and reorder buffer for each dispatched instruction

- The idea to combine reservation stations and reorder buffer together.

  → instruction window

  - Future file – fast access to last values
  - Architectural file – precise exception

- Instructions are dispatched into instruction window in this case; its entries monitor for matching input operands (denoted by tag) on result buses

- Size of instruction window determines the number of instructions which can be executed in parallel – maximal degree of parallelism (DOP)

“Another” solution?

- Idea 1: Perform the architectural register file update when an instruction finishes execution, but in the case of exception be able to rollback changes. We need some storage to do this → **History Buffer (HB)**
- History Buffer:
  - During decoding, an entry in History Buffer is reserved
  - When an instruction finishes execution, the value from the architectural register is stored in HB
  - When the instruction is oldest one, and no exception was generated, an entry in HB can be discarded
  - When the instruction is the oldest one and exception was generated, values from HB are copied into architectural registers

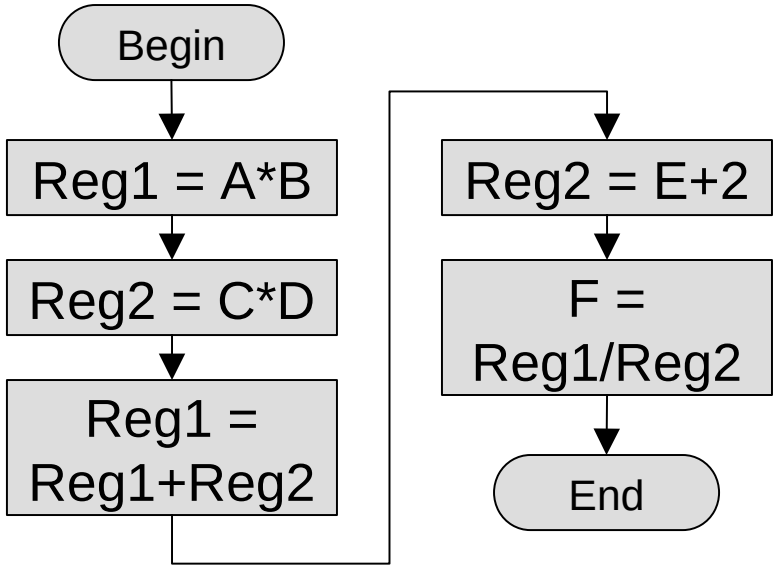
- Idea 2: Perform the architectural register file update in program order. Nevertheless, use another “invisible” registers instead. These “invisible” registers are immediately updated when instruction finishes execution (they hold the most current results) → **Future File (FF)**
- Future File:
  - Future file – fast access to last values
  - Architectural file – precise exception

And what is a difference between Future File and Rename Register File ? (slides 6 and 7 of this lecture) FF entries match ARF

Recall the first lecture: General view

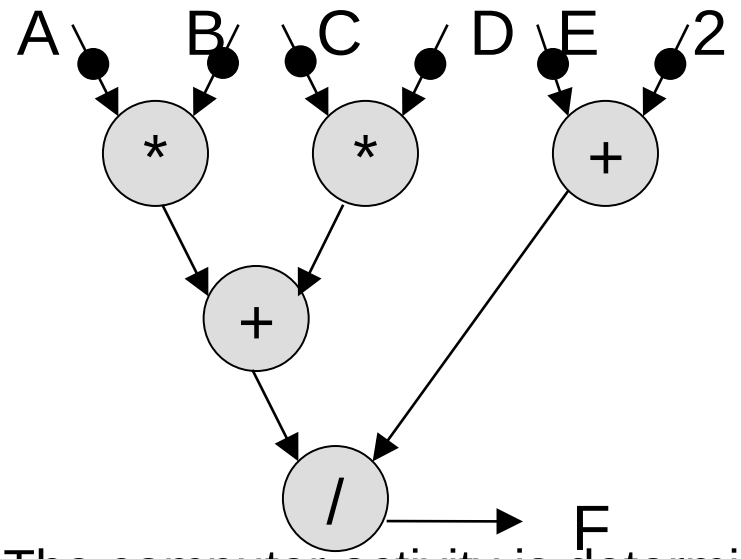
Considering: F = (A*B + C*D) / (E+2)

Control Flow approach:



The computer activity is determined by the sequence of instructions…

Data Flow approach:



The computer activity is determined by demands for results (demand driven computer) or by the presence of operands (data driven)…

Recall the first lecture: Control Flow vs. Data Flow

- Instructions are processed in the order as they are written (according to Program Counter: PC' = PC+1).
- Special control flow instructions (jumps, branch) are used to alleviate this rule

→ Imperative programming

- Out-of-order execution have to produce the same results as the program in their original form -> amount of HW resources to satisfy this

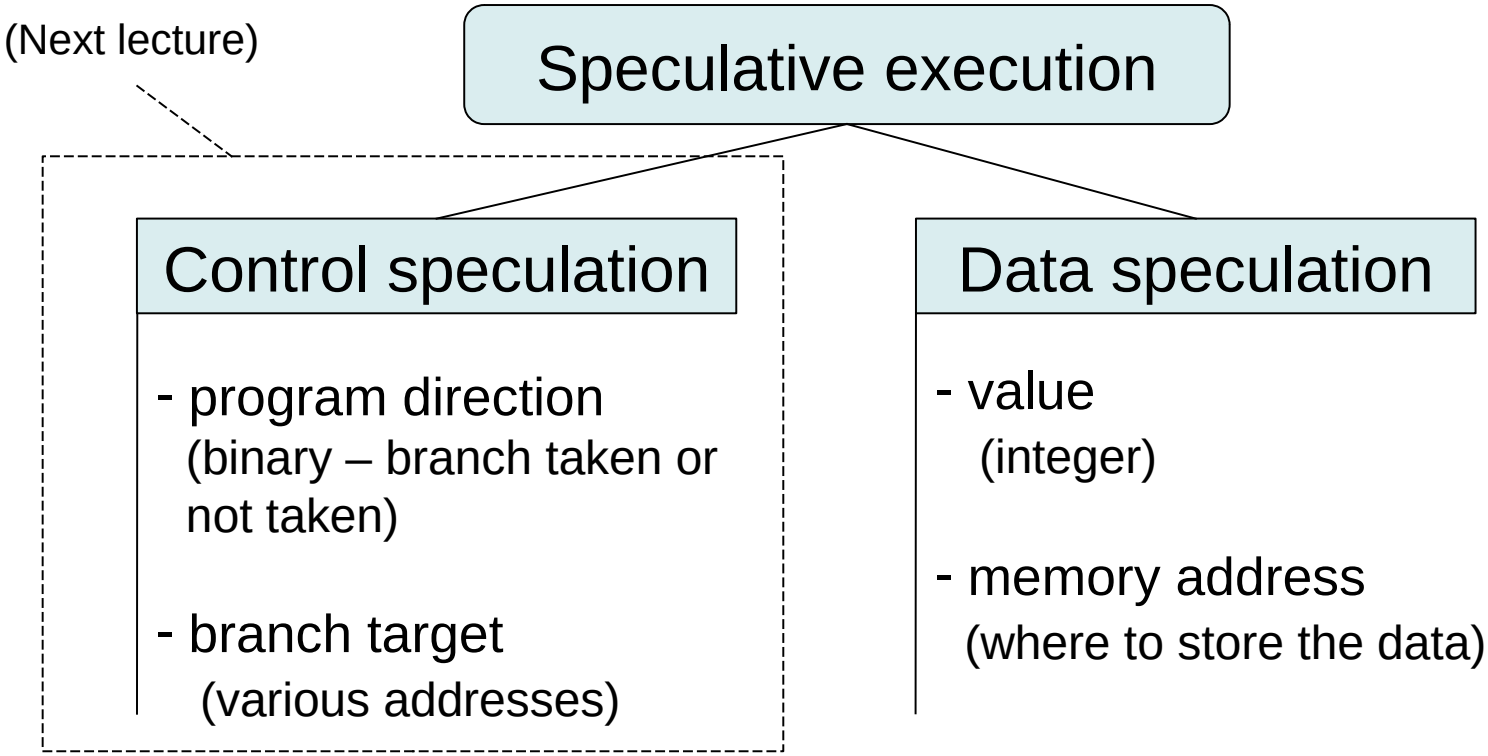
- Instruction in DF computer is represented as the {operation; source(s); destination}
- Instruction is executed when it has a valid source(s)
- DF computer does not have a program counter
- The ordering of instructions during program execution is given by the instruction interdependencies and by the presence of inputs

Data flow limit

- Out-of-order execution in combination with register renaming realizes “Data flow computer” inside CPU
- In this case, an only a small part of the whole program is parallelized (tens … hundred of instructions)
- However, if we imagine an unlimited number of hardware resources (functional units, data paths, etc.), we can achieve the maximal degree of parallelism (all WAW and WAR dependencies will be removed, only RAW dependencies remain)

- What is the **Data flow limit**?
- Suppose any technique for getting faster CPU!!!

Speculation?

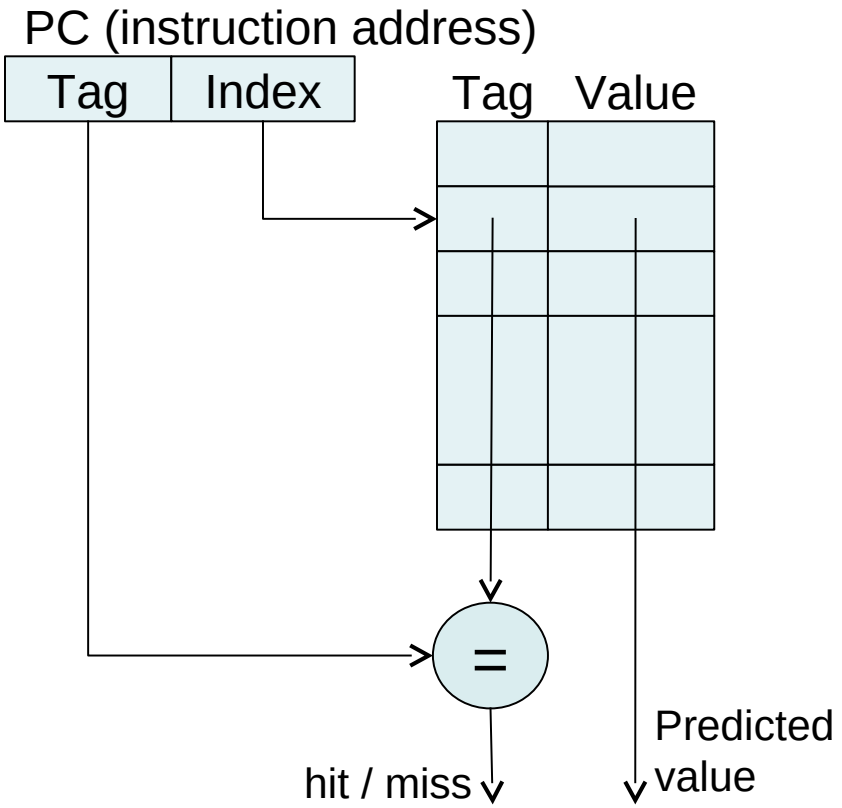


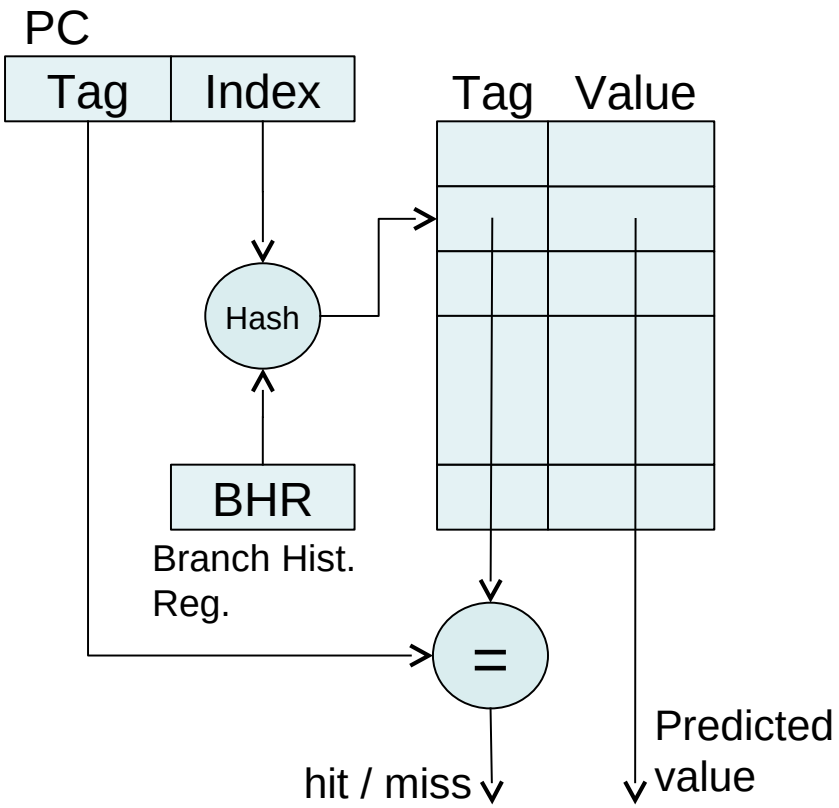
- What is the **Data flow limit**?
- If we want the higher speed-up, we can try to "remove " RAW dependencies… But how?

Data speculation – value prediction

- Last-value predictor.

Motivation: n=scanf(„%d“,&x), etc.

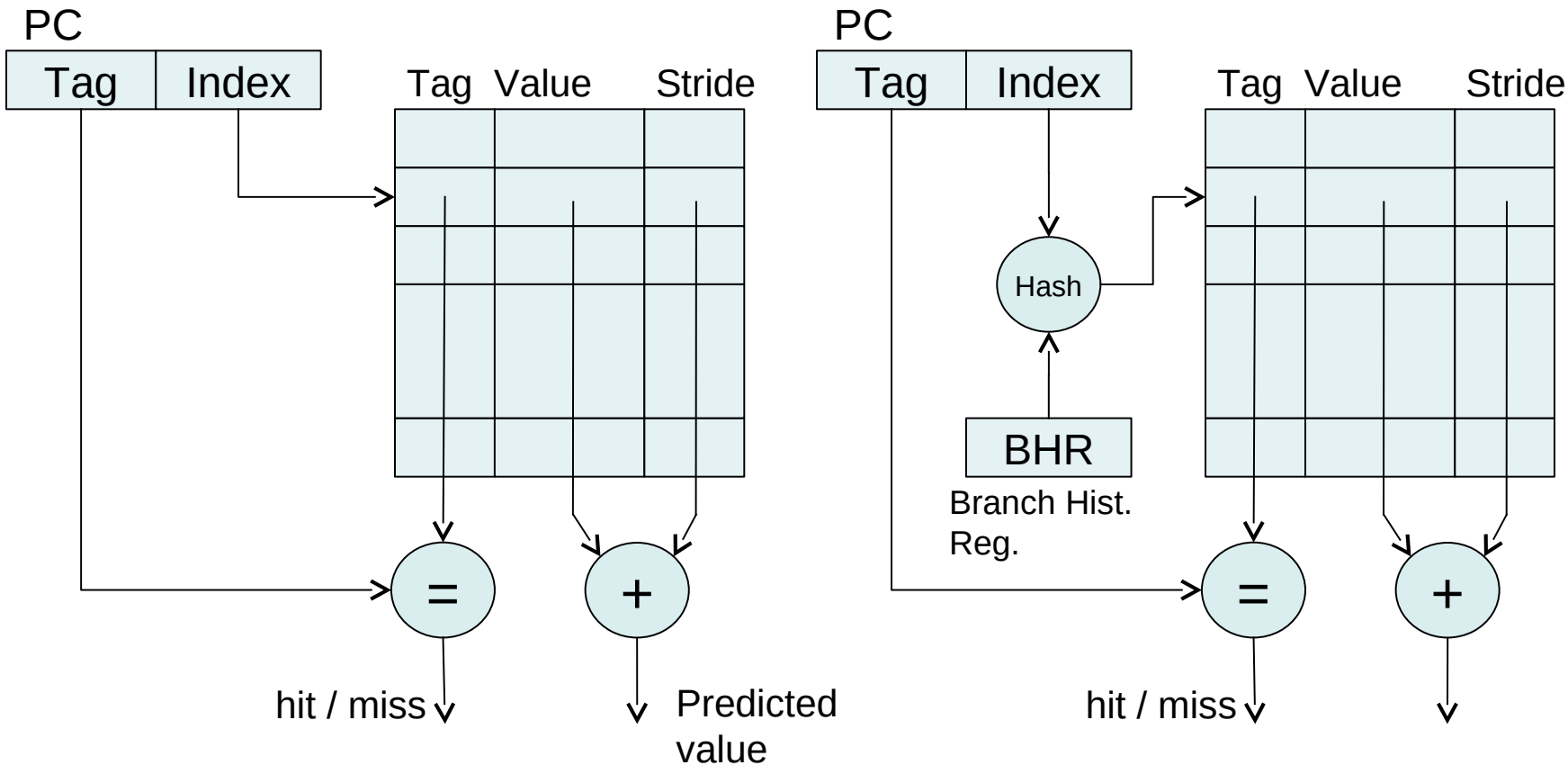




Data speculation – value prediction

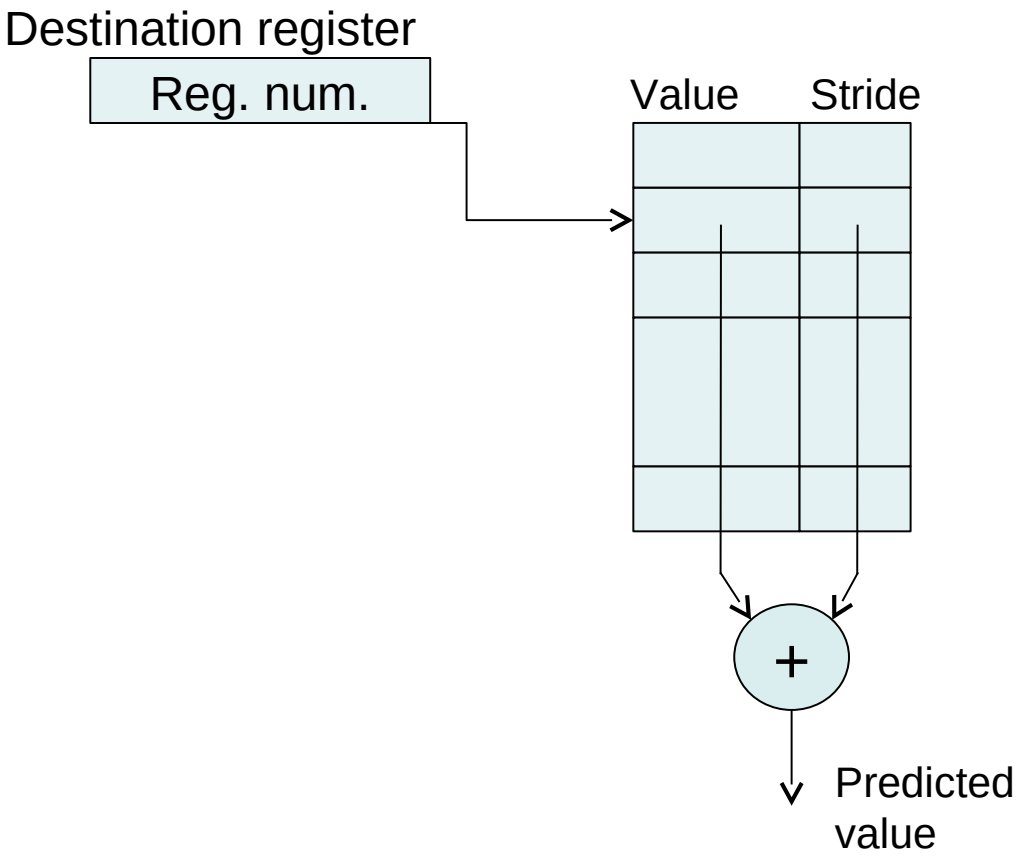
- Stride predictor.

Motivation: for(i=0;i<100;i++), p=malloc(16), etc.



Data speculation – value prediction

- Register-file predictor



What is next?

- Is there another way how to resolve WAW and WAR dependences?
- Is there another way how to resolve RAW dependences?

- Yes… „we can remove not only edges, but even vertices“

Literature

References:

1. D. Sima: The design space of register renaming techniques Design-space-of-register-renaming-techniques.pdf
2. http://en.wikipedia.org/wiki/IBM_System/360
3. Onur Mutlu: Computer Architecture 15-740/18-740, Lecture 9: More on Precise Exceptions. Carnegie Mellon University. 2011
4. **Shen, J.P., Lipasti, M.H.: Modern Processor Design : Fundamentals of Superscalar Processors, First Edition, New York, McGraw-Hill Inc., 2005**
5. http://www.dia.eui.upm.es/Asignatu/Arq_par/articulos/AMDBulldozer.pdf
6. http://www.cs.cmu.edu/afs/cs/academic/class/15213-f01/docs/mpr-p6.pdf