2. Processor micro-architecture: Implicit parallelism

Pipelining, scalar & superscalar execution

Advances in Computer Architecture

Motivation

- Pipeline-level parallelism is the weapon of architects to **increase throughput** and **tolerate latencies of communication** for **individual instruction streams** (i.e. sequential programs) **without participation from the programmer** (i.e. implicit)
- We will cover true and explicit parallelism later in the course

Processor performance

- Latency: expressed as **CPI** = cycles per instruction
  divide by frequency to obtain absolute latency
- Throughput: expressed as **IPC** = instructions per cycle
  multiply by frequency to obtain absolute throughput

- Pipelining objective: **increase IPC**, also decrease CPI
  As we will see ↘CPI and ↗IPC are conflicting requirements

Types of pipeline concurrency

- **Pipelined**: operations broken down in sub-tasks
  ⇒ different sub-tasks from different operations run in parallel

- **Scalar** pipelined: multiply the functional units
  ⇒ the same sub-task from different operations run in parallel

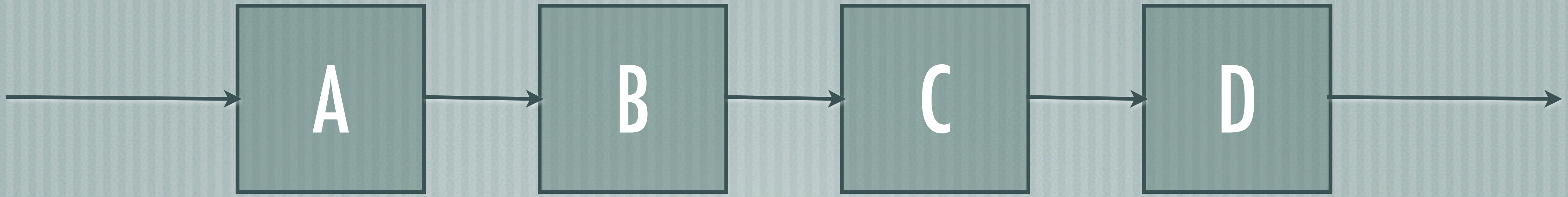
- **Superscalar** pipelined: multiply the issue units
  ⇒ multiple operations issued and completing simultaneously

Pipelining & hazards

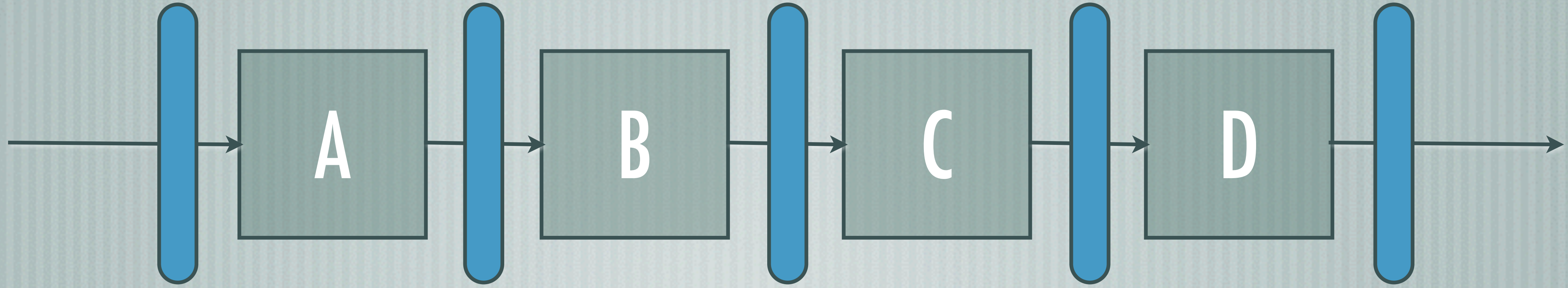
Pipeline basics

Pipelines

- Each instruction / operation can be decomposed in sub-tasks:
  Op = A ; B ; C ; D

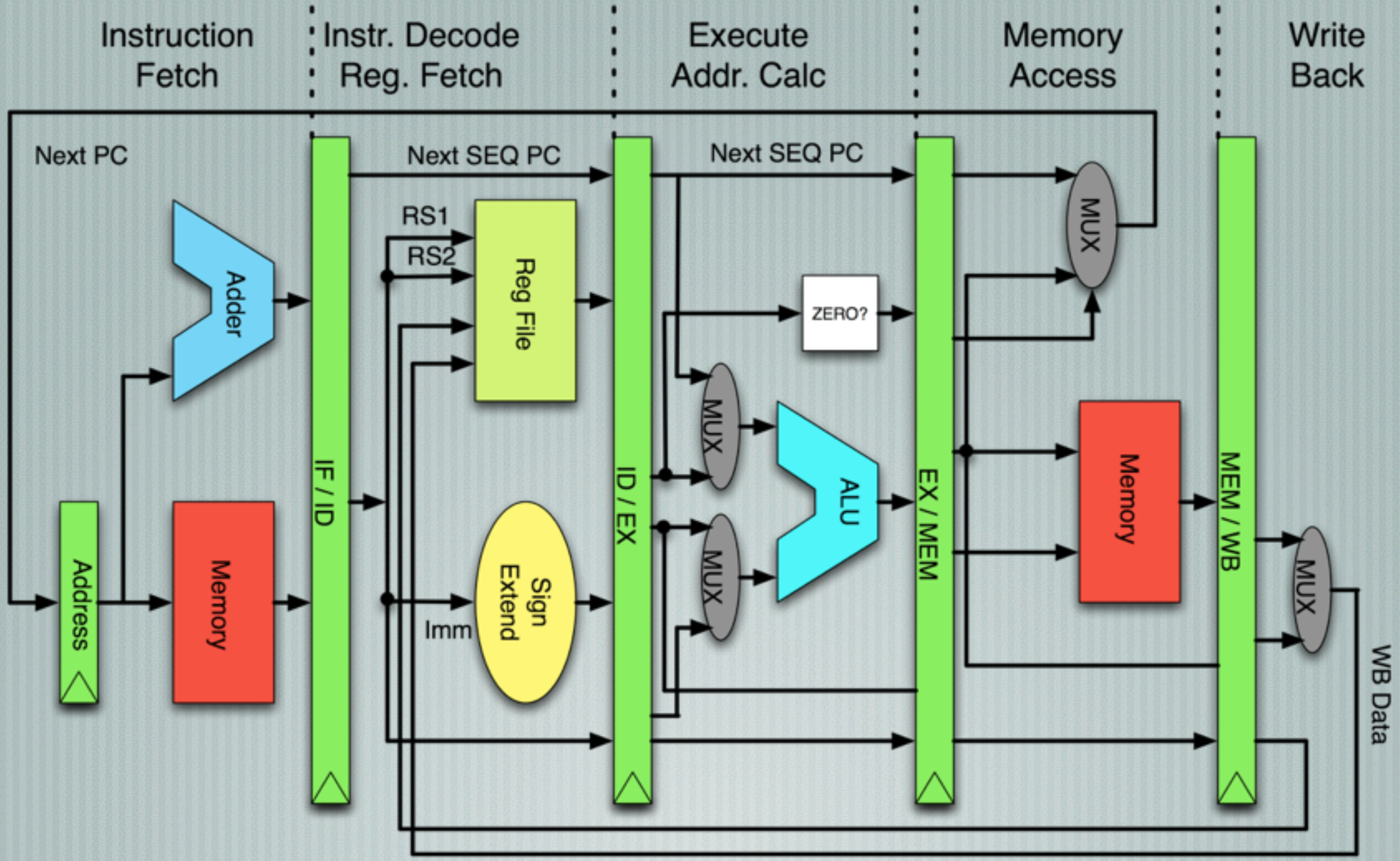


- Considering an instruction stream [Op$_1$; Op$_2$; ...]
  at each cycle n we can run in parallel: $A_{n+3} \parallel B_{n+2} \parallel C_{n+1} \parallel D_n$



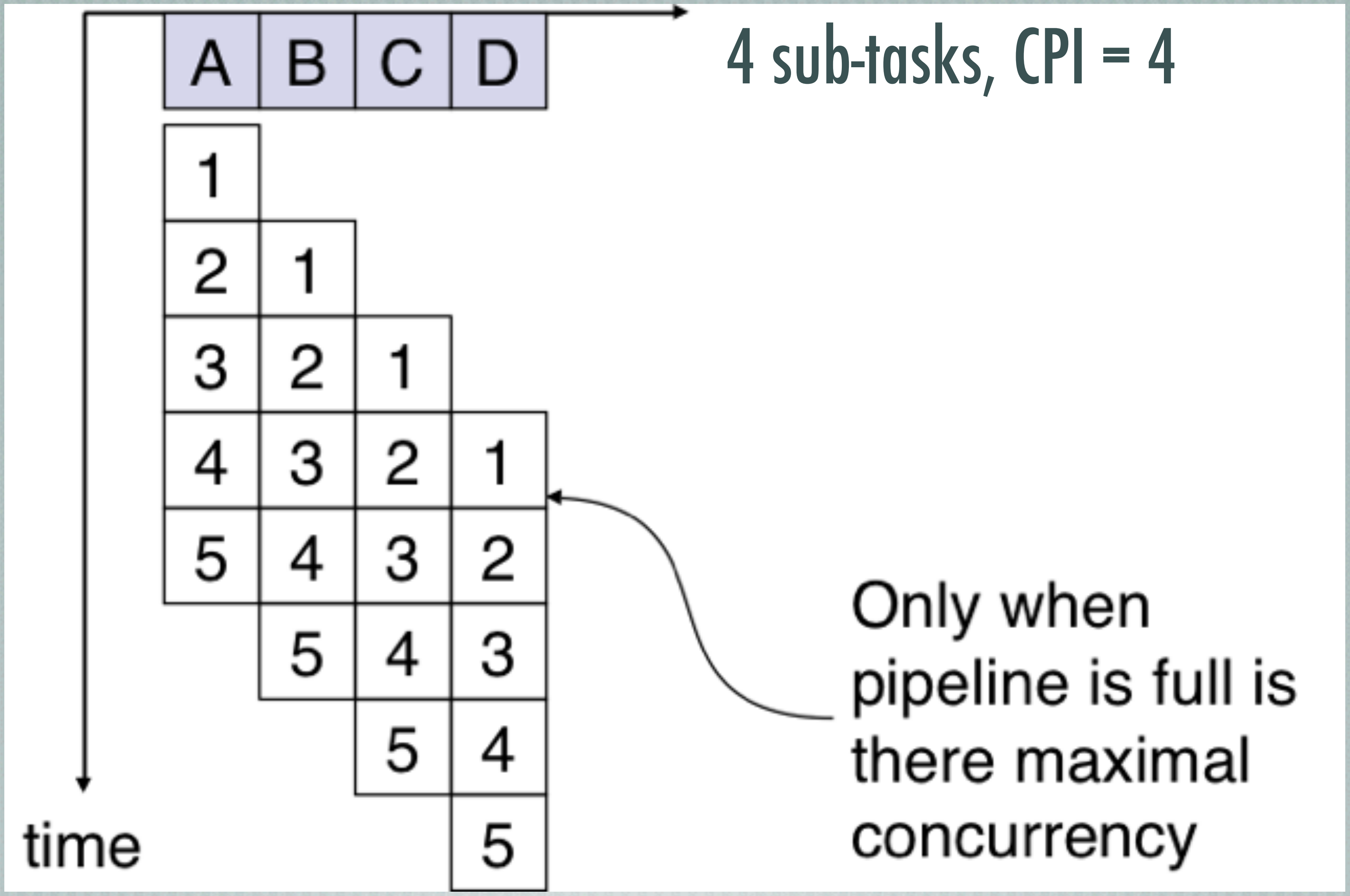
at start of cycle n: input A_{n+3} input B_{n+2} input C_{n+1}

Example: MIPS



Dynamic behavior

- 1 program, 5 operations:



4 sub-tasks, CPI = 4

(total execution time: 8 cycles)

Pipeline performance

- This pipeline has a length of 4 subtasks, assume each sub-task takes t seconds
  - for a single operation we get no speedup; it takes 4t seconds to complete all of the subtasks
  - this is the same as performing each sub task in sequence on the same hardware
- In the general case – for n operations – it takes 4t seconds to produce the first result and t seconds for each subsequent result

Pipeline performance

- For a pipeline of length L and cycle time t, the time T it takes to process n operations is:
  $\mathbf{T(n) = L \cdot t + (n\text{-}1) \cdot t = (L\text{-}1) \cdot t + n \cdot t}$
- We can characterise all pipelines by two parameters:
  - **startup time**: $S = (L\text{-}1) \cdot t$ (unit: seconds)
  - **maximum rate**: $r_\infty = 1/t$ (unit: instructions per second)

Optimization strategies

- Long instruction sequences suggest IPC = 1
- However there are problems with this:
  - some instructions require less sub-tasks than others
  - hazards: dependencies and branches
  - long-latency operations: can't fit the pipeline model
- What to do about these? The rest of the lecture covers this...

Trade-offs

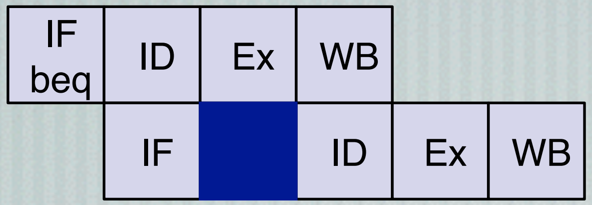
- Observations:
  - **more complexity** per sub-task requires **more time per cycle**
  - conversely, as the sub-tasks become simpler the cycle time can be reduced
  - so **to increase the clock rate** instructions must be **broken down into smaller sub-tasks**
  - ...but operations have a fixed complexity
  - **smaller sub-tasks mean deeper pipelines** = more stages $\Rightarrow$ more instructions need to be executed to fill the pipeline

Control hazards

Control hazards

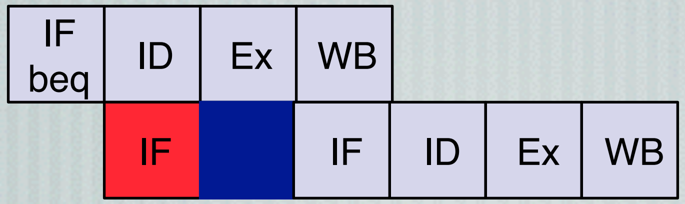
- Branches – in particular **conditional branches** – cause pipeline hazards
- the outcome of a conditional branch is not known until the end of the EX stage, but is required at IF to load another instruction and keep the pipeline full
  - A simple solution:
    assume by default that the branch falls through – i.e. is not taken – then continue speculatively until the target of the branch is known

Branch not taken



Continue to fetch but stall ID until branch target is known – one cycle lost

Branch is taken



Need to refetch at new target – two cycles lost

Wrong target

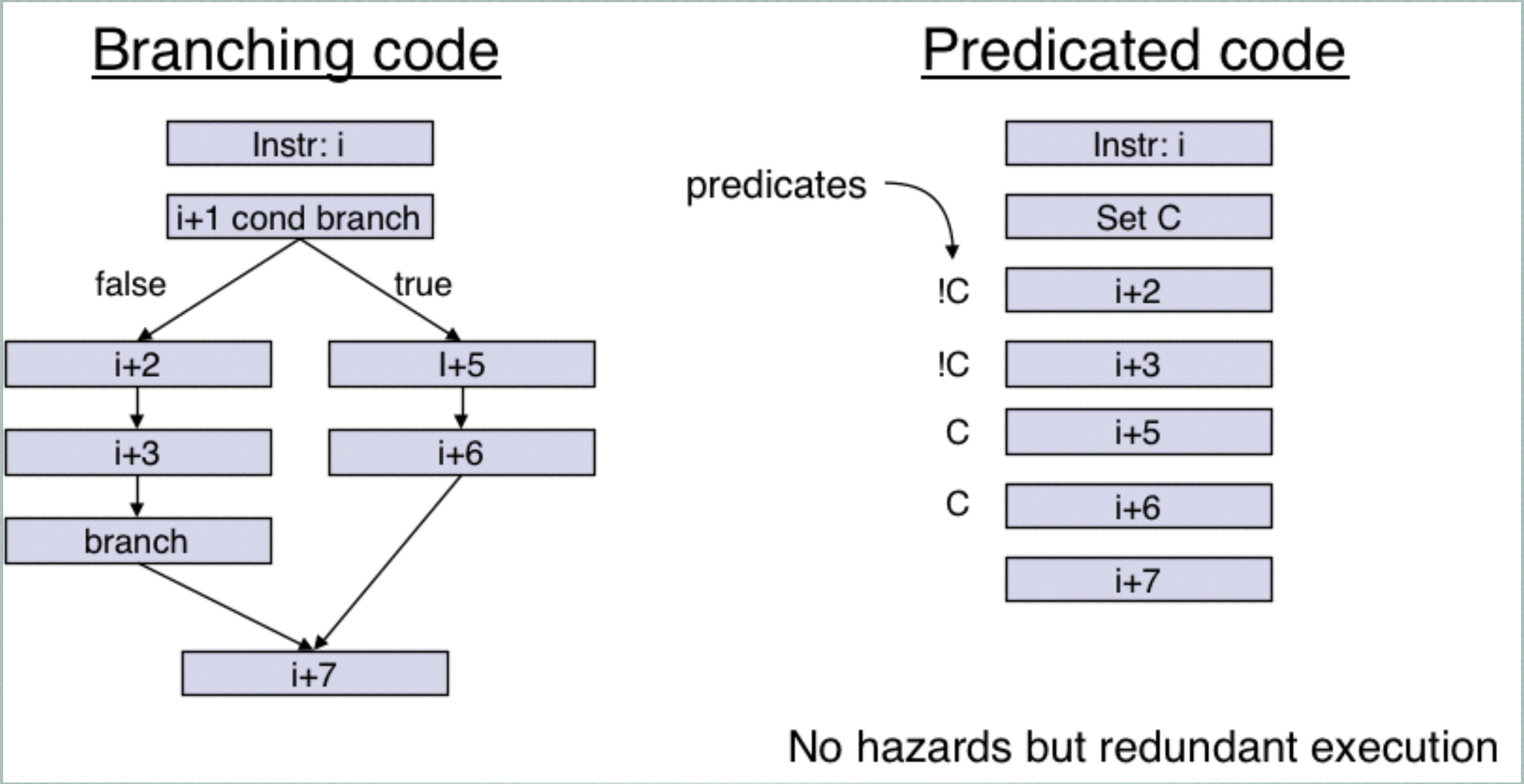
How to overcome

- Eliminate branches altogether via **predication** (most GPUs)
- Expose the branch delay to the programmer / compiler: **branch delay slots** (MIPS, SPARC, PA-RISC)
- Fetch from both targets, requires branch target address prediction
- Predict whether the branch is taken or not: **branch prediction** (cf later part of the lecture)
- Execute instructions from other threads: **hardware multithreading** (eg Niagara, cf next lecture)

Predication

- Control flow can (in some cases) be replaced by guarded or predicated instruction execution...
  - a condition sets a predicate register (boolean)
  - instructions are predicated on that register
  - any state change (WB or Mem write) only occurs if the predicate is true
- Useful in long pipelines where branch hazards can be costly,
  or to simplify the pipeline logic by not handling control hazards at all
- it removes a control hazard at the expense of **redundant instruction execution**

Predication – example



No hazards but redundant execution

Branch delay slots

- Specify in the ISA that a branch takes effect two instructions later, then let the compiler / programmer fill the empty slot

```
L1:
   lw a x[i]
   add a a a
   sw a x[i]
   sub i i 4
   bne i 0 L1
   nop
L2:
```

1 cycle wasted at each iteration

```
L1:
   lw a x[i]
   add a a a
   sw a x[i]
   bne i 4 L1
   sub i i 4
L2:
```

no bubble, but one extra sub at last iteration

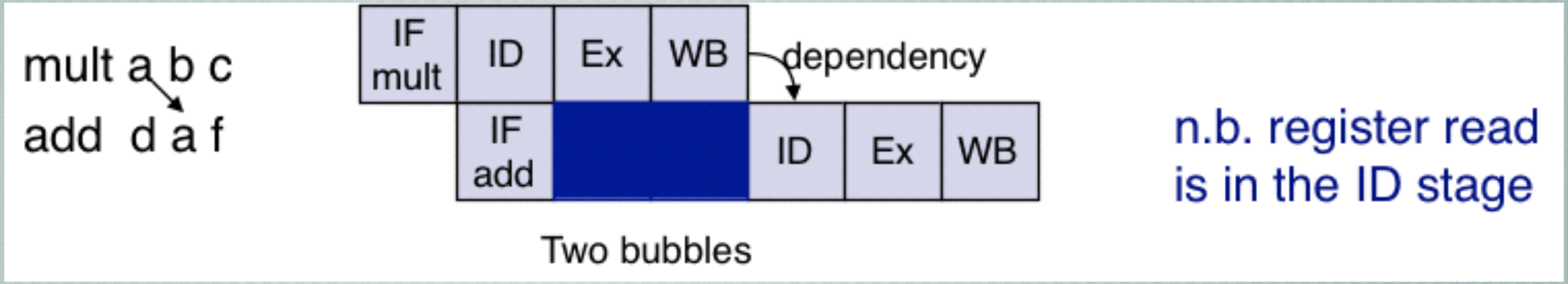
Fetch from both targets

- Using an additional I-cache port, both taken and not-taken are fetched
  - Then at EX a choice is made as to which is decoded
- When coupled with a branch delay slot this eliminates all wasted cycles, but...
  - longer pipelines, eg 20 stages, might contain several branches in the pipe prior to EX
  - **multiple conditional branches will break** this solution,
  - as every new branch doubles the number of paths fetched

Data hazards

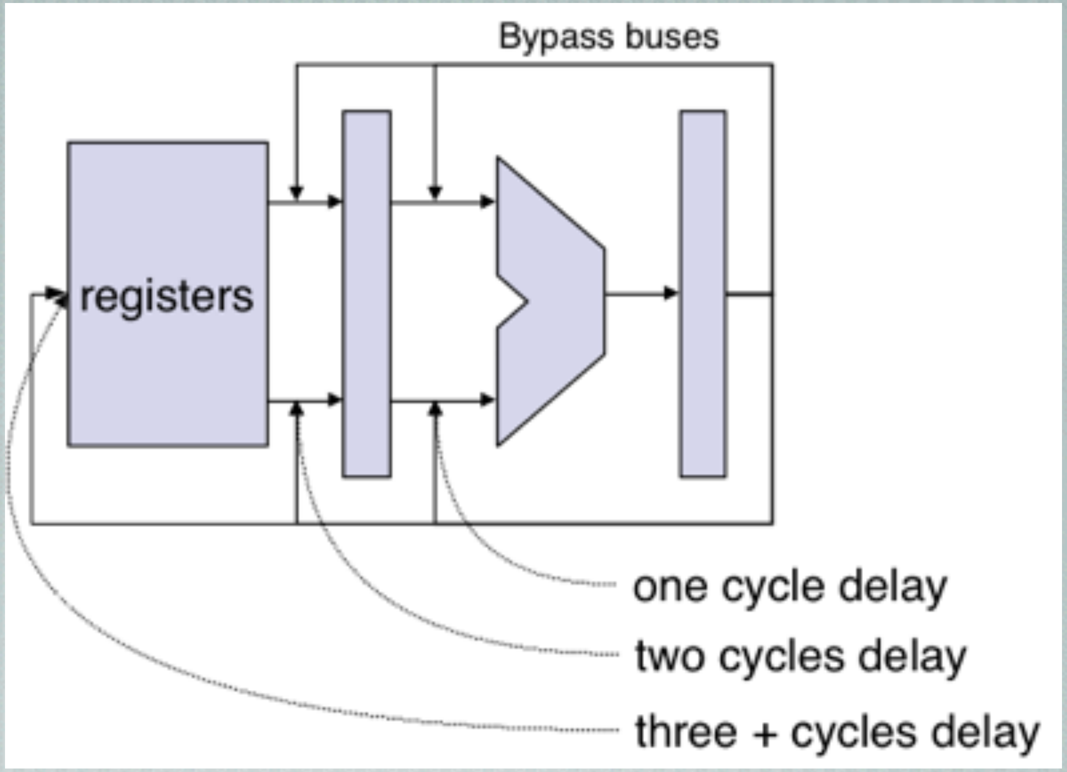
Data hazard

- Occurs when the output of one operation is the input of a subsequent operation
- The hazard occurs because of the latency in the pipeline
  - the **result** (output from one instruction) is not written back to the register file until the last stage of the pipe
  - the **operand** (input of a subsequent instruction) is required at register read – some cycles prior to writeback
  - the longer the RR to WB delay, the more cycles there must be between the writeback of the producer instruction and the read from the consumer
- Example:



How to overcome

- **Do nothing** i.e. expose to the programmer, e.g. MIPS 1
- **Bypass buses**:



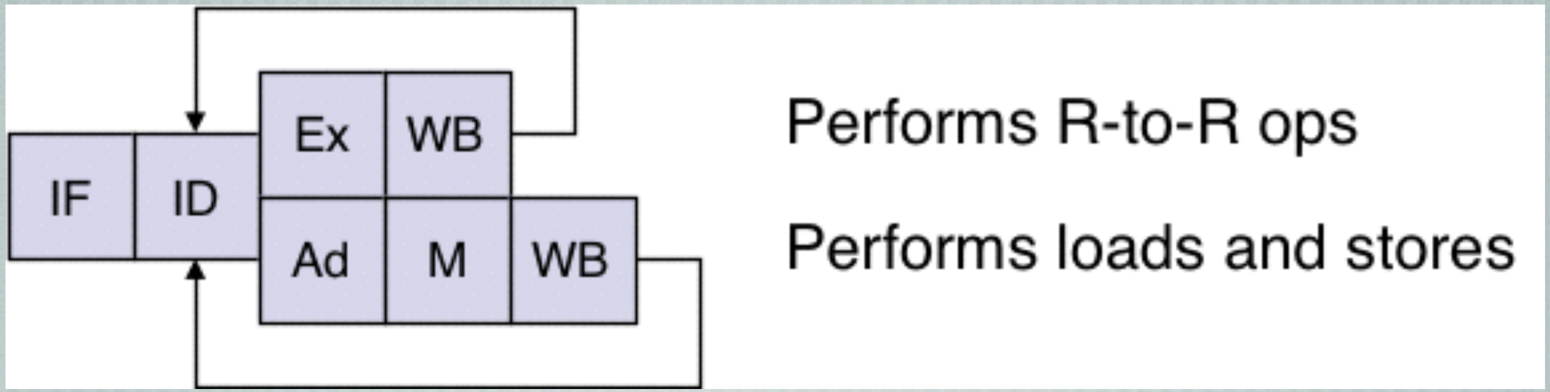
The operand is taken from the pipeline register and input directly to the ALU on the subsequent cycle

- Also: **reorder the instructions**

Structural hazards and scalar ILP

Scalar pipelines

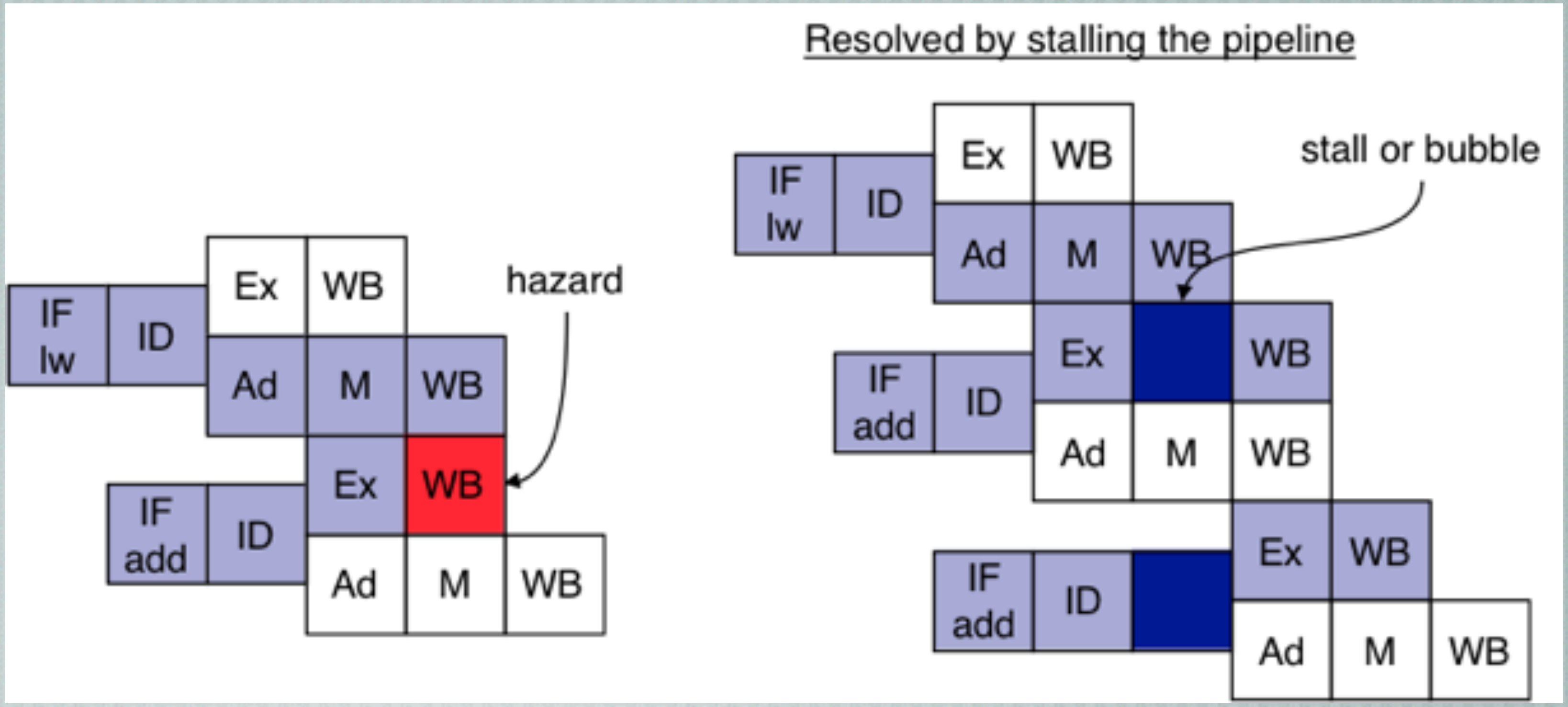
- In the simple pipeline, register-to-register operations have a wasted cycle
  - a memory access is not required, but this stage still requires a cycle to complete the operations
- Decoupling memory access and operation execution avoids this
  e.g. use an ALU plus a memory unit - this is **scalar ILP**



- ... note: either we need two write ports to the register file or arbitration on a single port

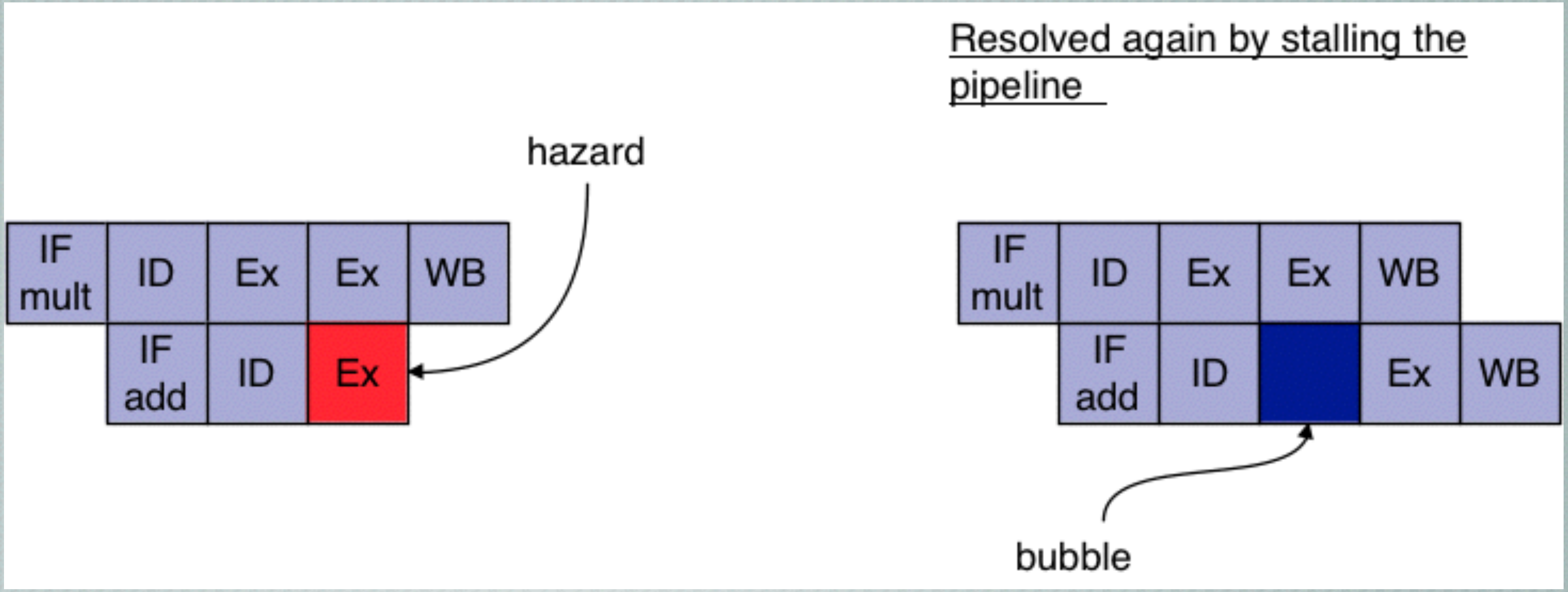
Structural hazard - registers

- A structural hazard occurs when **a resource in the pipeline is required by more than one instruction**
- a resource may be an execution unit or a register port
  - Example: **only one write port**
    lw a addr
    add b c d



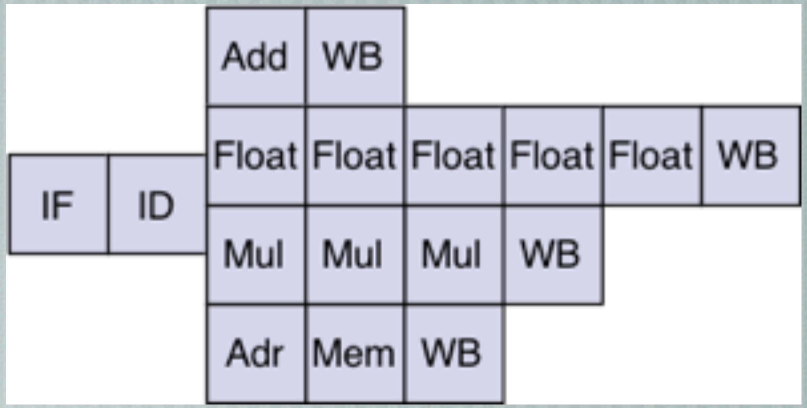
Structural hazard - execution units

- Some operations require more than one pipeline cycle
  - mult is more complex than add (often requires 2 cycles)
  - floating point still more complex still (~ 5 cycles)
- Example: 2-cycle multiply
  mult c d e
  add f g h



How to overcome

- They result from **contention**
  ⇒ they can be removed by **adding more resources**
  - register write hazard: add more write ports
  - execution unit: add more execution units
- Example: CDC 6600 (1963)
  10 units, 4 write ports, only FP div not pipelined



- Note:
  - **more resources = more cost** (area, power)

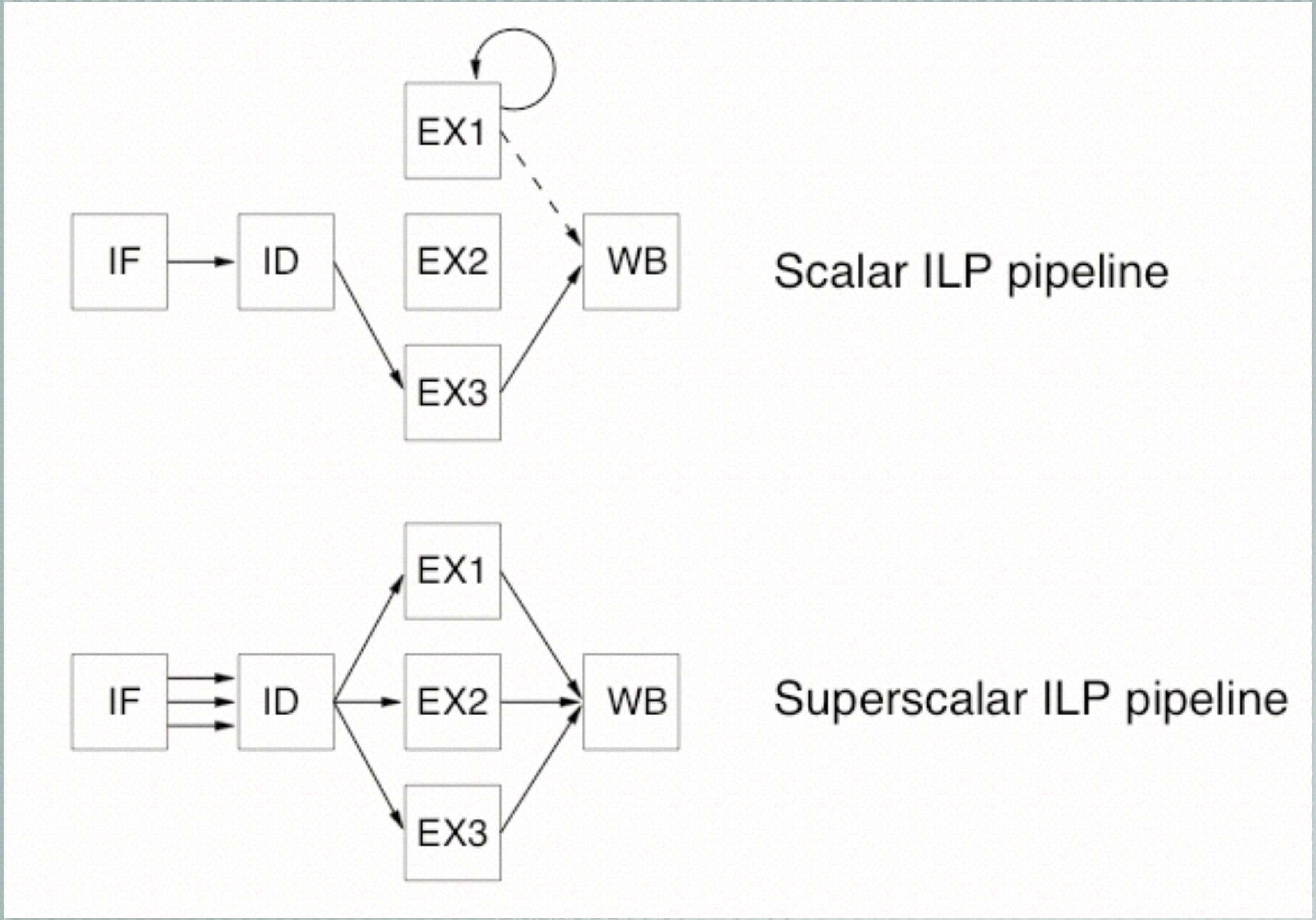
Superscalar processors

Introduction / overview

Pipelining - summary

- Depth of pipeline - **Superpipelining**
  - further dividing pipeline stages **increases frequency**
  - but introduces **more scope for hazards**
  - and higher frequency means **more power dissipated**
- Number of functional units - **Scalar pipelining -** avoids waiting for long operations to complete
  - instructions fetched and **decoded in sequence**
  - multiple operations **executed in parallel**
- Concurrent issue of instructions - **Superscalar ILP**
  - multiple instructions **fetched and decoded concurrently**
  - new **ordering issues** and **new data hazards**

Scalar vs. superscalar



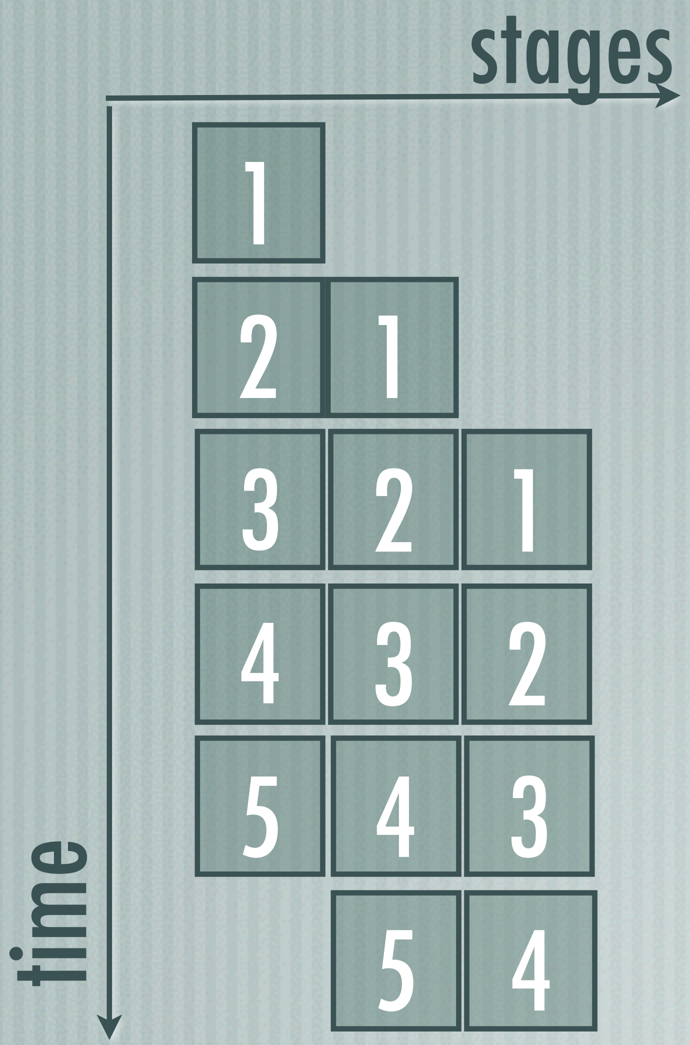
in-order issue

concurrent issue,
possibly out of order

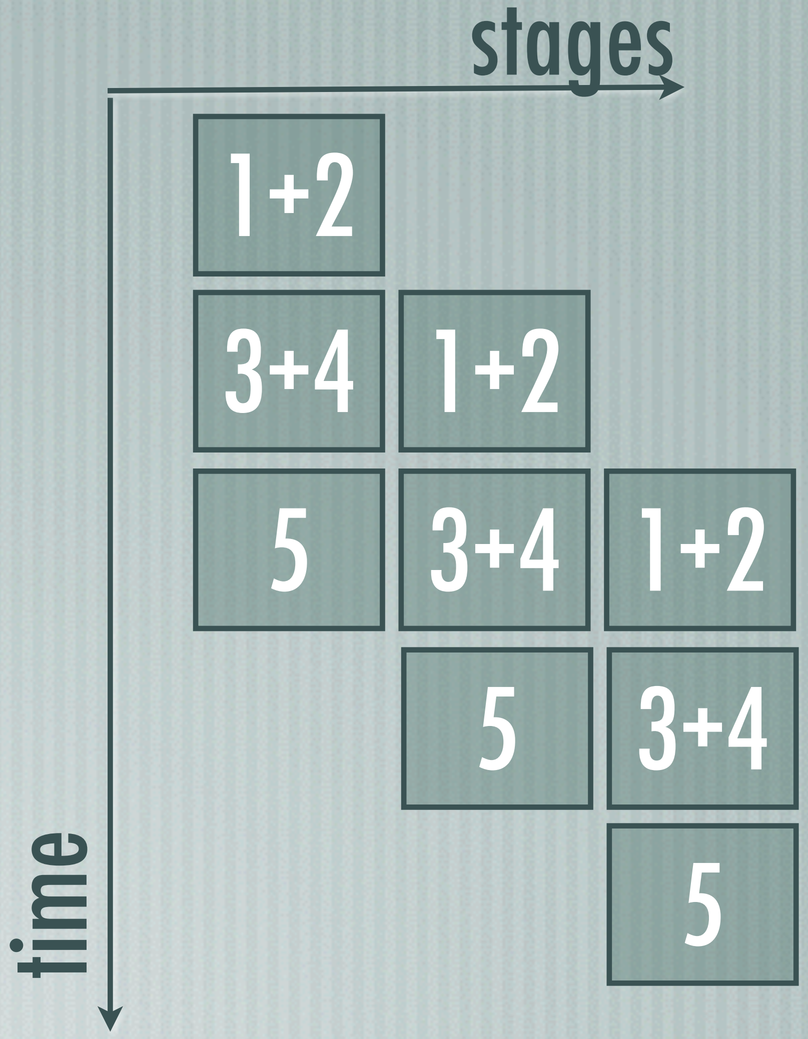
Most "complex" general-purpose processors are superscalar

Basic principle

Example based on simple 3-stage pipeline



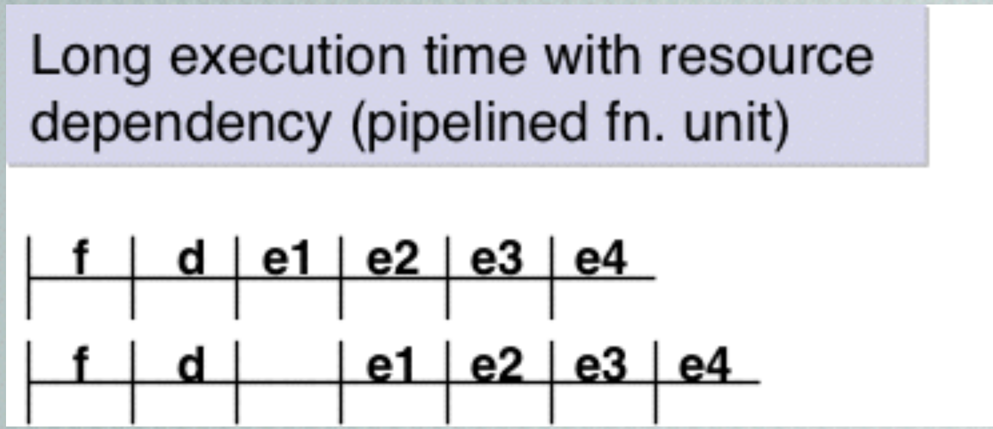
Scalar pipeline, max IPC = 1

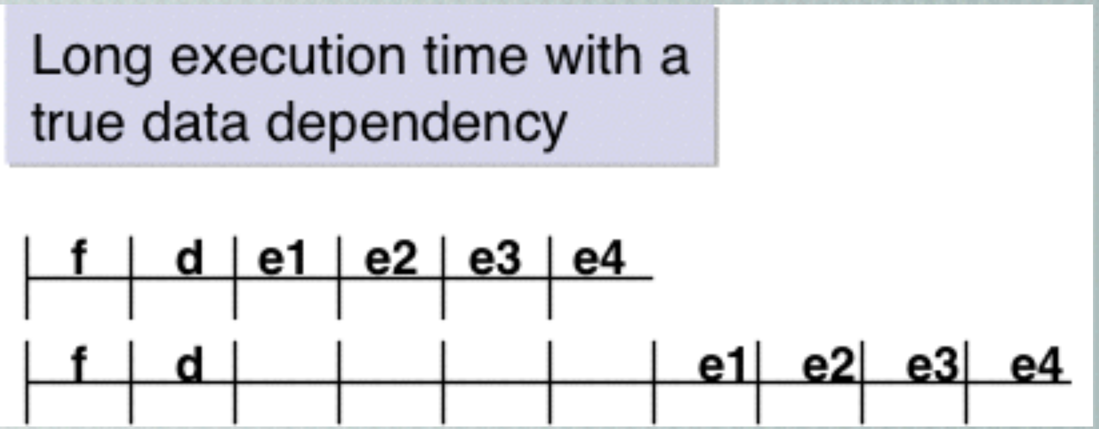


Superscalar, max IPC ≥ 1

Instruction-level parallelism

- **ILP** is the number of instructions issued per cycle (**issue parallelism / issue width**)
- **IPC** the number of instructions executed per cycle is limited by:
  - the ILP
  - the number of true dependencies
  - the number of branches in relation to other instructions
  - the latency of operations in conjunction with dependencies
- Current microprocessors: 4-8 max ILP, 12 functional units, however IPC of typically 2-3


Long execution time with resource dependency (pipelined fn. unit)
f
d
e1
e2
e3
e4
f
d
e1
e2
e3
e4



Aspects of superscalar execution

- parallel fetch decoding and issue
  - 100s of instructions in-flight simultaneously
- out-of-order execution and sequential consistency
  - Exceptions and false dependencies
- finding parallelism and scheduling its execution
- application specific engines, e.g. SIMD & prefetching

Instruction policies & related hazards

Instruction issue vs completion, new data hazards

Instruction issue basics

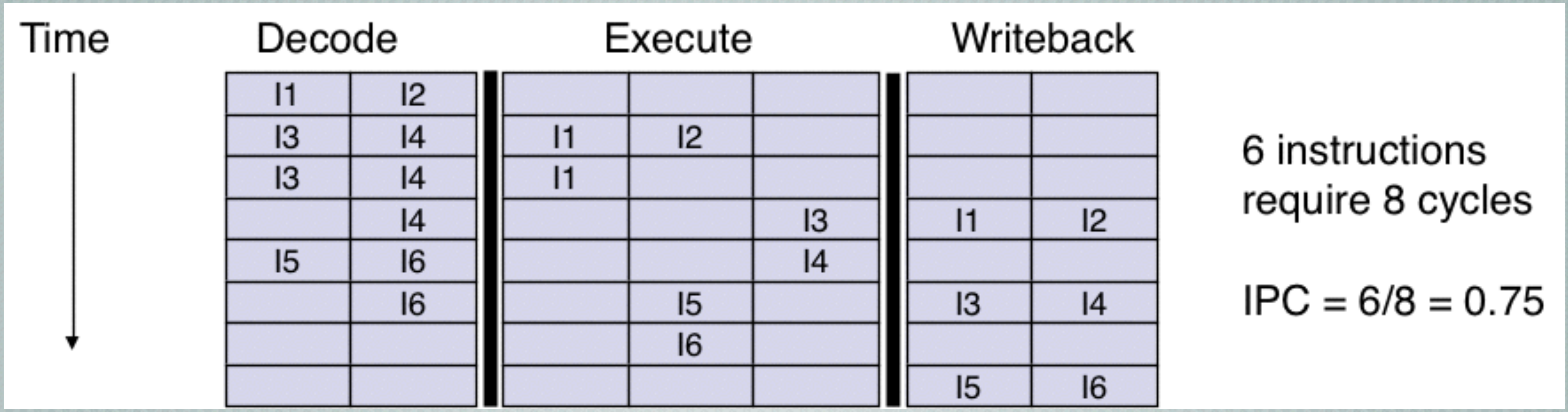
- Just widening of the processor's pipeline does not necessarily improve its performance
- The processor's **policy in fetching, decoding and executing instructions** also has a significant effect on its performance
- The instruction issue policy is determined by its **look-ahead capability** in the instruction stream
  - For example, with no look-ahead, if a resource conflict halts instruction fetching the processor is not able to find any further instructions until the conflict is resolved
  - If the processor is able to continue fetching instructions it may find an independent instruction that can be executed on a free resource out of programmed order
- Policies are characterized by **issue order** and **completion order**

In-order issue, in-order completion

- Simplest, unusual with superscalar designs
- Instructions issued in exact program order with results written in the same order
- This is shown here for comparison purposes only, as very few pipelines use in-order completion

In-order issue, in-order completion

- Assume a 3 stage execution in a pipeline that can issue two instructions, execute three instructions and write back two results every cycle... assume:
  - I1 requires 2 cycles to execute
  - I3 and I4 are in conflict for a functional unit
  - I5 depends on the value produced by I4
  - I5 and I6 are in conflict for a functional unit



Time

Decode		Execute			Writeback	
I1	I2					
I3	I4	I1	I2			
I3	I4	I1				
	I4			I3	I1	I2
I5	I6			I4		
	I6		I5		I3	I4
			I6			
					I5	I6

6 instructions require 8 cycles

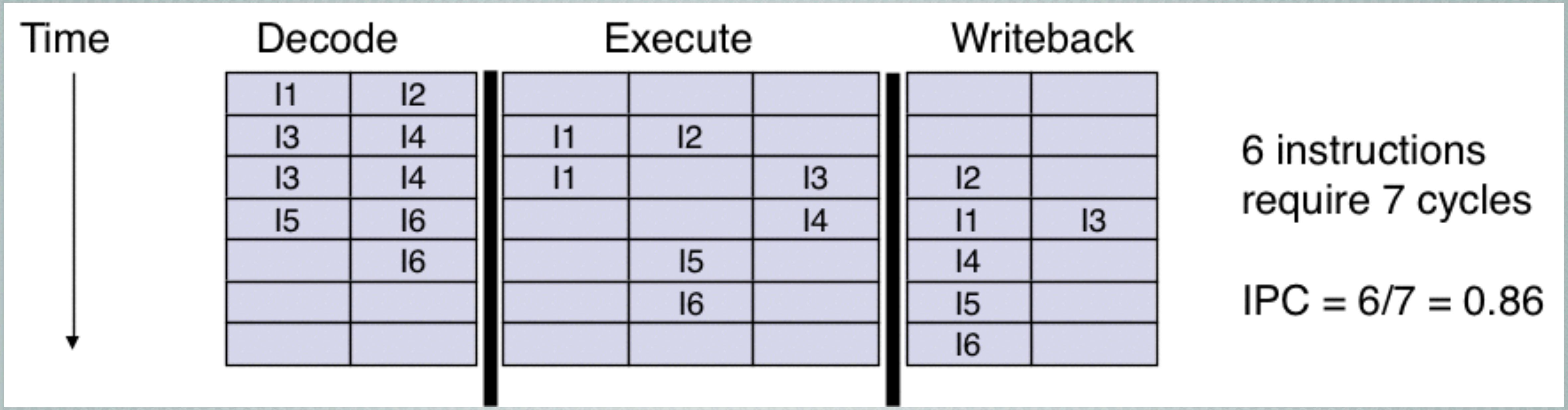
IPC = 6/8 = 0.75

In-order issue, out-of-order completion

- Out-of-order completion, improves performance of instructions with long latency operations, such as loads and floating point
- The modifications made to execution are:
  - any number of instructions allowed in the execution stage up to the total number of pipeline slots (stages × functional units)
  - instruction issue is not stalled when an instruction takes more than one cycle to complete

In-order issue, out-of-order completion

- Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle
  - I1 requires 2 cycles to execute
  - I3 and I4 are in conflict for a functional unit
  - I5 depends on the value produced by I4
  - I5 and I6 are in conflict for a functional unit



Time	Decode		Execute			Writeback	
↓	I1	I2					
	I3	I4	I1	I2			
	I3	I4	I1		I3	I2	
	I5	I6			I4	I1	I3
		I6		I5		I4	
				I6		I5	
						I6	

6 instructions require 7 cycles

IPC = 6/7 = 0.86

In-order issue, out-of-order completion

- In a processor with out-of-order completion, instruction issue is stalled when:
  - There is a **conflict** for a functional unit
  - An instruction depends on a result that is not yet computed - a **data dependency**
    - can use register specifiers to detect dependencies between instructions and logic to ensure synchronisation between producer and consumer instructions – e.g. scoreboard logic, cf CDC 6600
  - Also: a new type of dependency caused by out-of-order completion: the **output dependency (Write-after-Write dependency)**

Output dependencies

- Consider the code to the right:
  - the 1st instruction must be completed before the 3rd, otherwise the 4th instruction may receive the wrong result!
  - this is a new type of dependency caused by allowing out-of-order completion
  - the result of the 3rd instruction has an **output dependency** on the 1st instruction
  - the 3rd instruction must be stalled if its result may be overwritten by a previous instruction which takes longer to complete

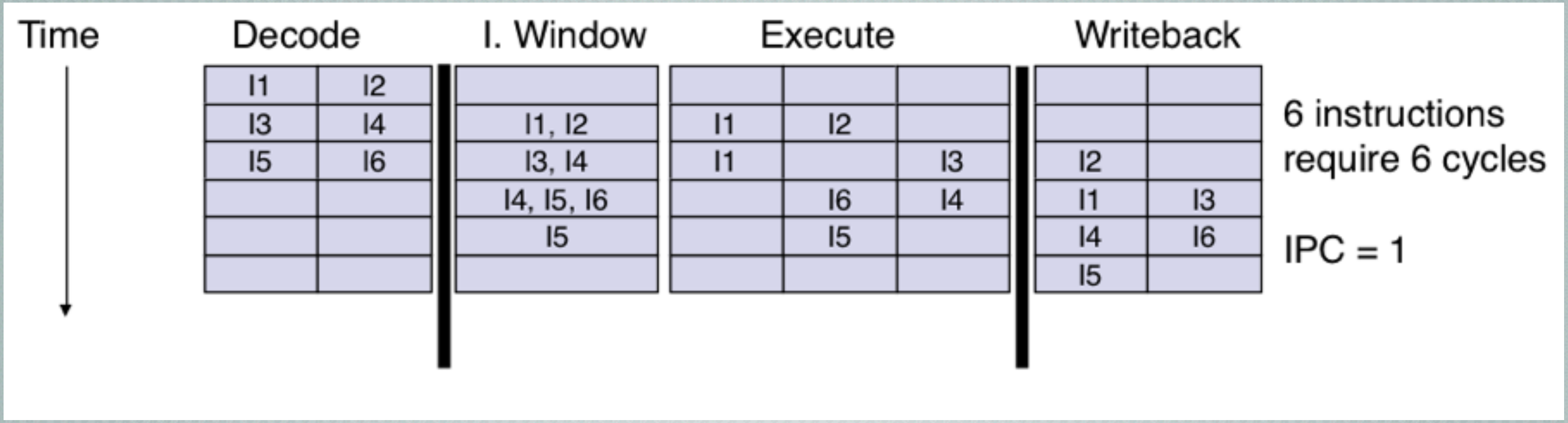
```
R3 := R3 op R5
R4 := R3  + 1
R3 := R5  + 1
R7 := R3 op R4
```

Out-of-order issue, out-of-order completion

- In-order issue stalls when the decoded instruction has:
  - a resource conflict, a true data dependency or an output dependency on an uncompleted instruction
  - this is true even if instructions after the stalled one can execute
  - to avoid stalling, decode must be decoupled from execution
- Conceptually out-of-order issue decouples the decode/issue stage from instruction execution
  - it requires an **instruction window** between the decode and execute stages
    to buffer decoded or part pre-decoded instructions
  - this buffer serves as a pool of instructions giving the processor a look-ahead facility
  - instructions are issued from the buffer in any order,
    provided there are no resource conflicts or dependencies with executing instructions

Out-of-order issue, out-of-order completion

- Again assume a processor issues two instructions, executes three instructions and writes back two results every cycle but now has a issue window of at least three instructions
  - I1 requires 2 cycles to execute
  - I3 and I4 are in conflict for a functional unit
  - I5 depends on the value produced by I4
  - I5 and I6 are in conflict for a functional unit



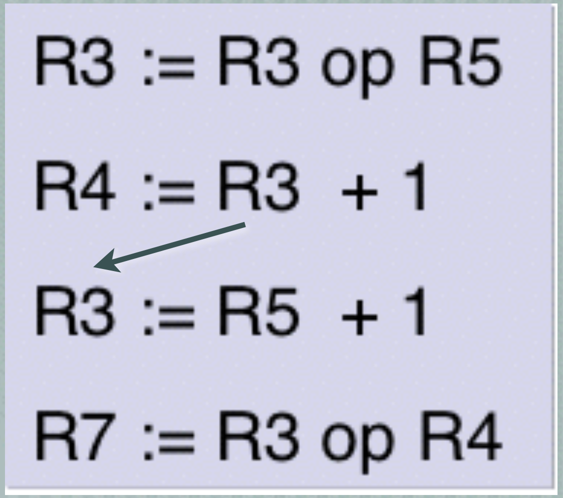
Decode		I. Window	Execute			Writeback	
I1	I2						
I3	I4	I1, I2	I1	I2			
I5	I6	I3, I4	I1		I3	I2	
		I4, I5, I6		I6	I4	I1	I3
		I5		I5		I4	I6
						I5	

6 instructions require 6 cycles

IPC = 1

Anti-dependencies

- Out-of-order issue introduces yet another dependency - called an anti-dependency (or Write-after-Read dependency)
  - the 3rd instruction can not be completed until the second instruction has read its operands
  - otherwise the 3rd instruction may overwrite the operand of the 2nd instruction
  - we say that the result of the 3rd instruction has an **anti-dependency** on the 1st operand of the 2nd instruction
  - this is like a true dependency but reversed



```
R3 := R3 op R5
R4 := R3  + 1
R3 := R5  + 1
R7 := R3 op R4
```

Summary of data hazards

- We have now have seen three kinds of dependencies
  - **True (data) dependencies** ... read after write (RAW)
  - **Output dependencies** ... write after write (WAW) - out of order completion
  - **Anti dependencies** ... write after read (WAR) - out of order issue
- Only true dependencies reflect the flow of data in a program and should require the pipeline to stall
  - when instructions are issued and completed out of order,
    the one-to-one relationship between registers and values at any given time is lost
  - new dependencies arise because registers hold different values from independent computations at different times – they are **resource dependencies**
- **Resource dependencies are really just storage conflicts** and can be eliminated by introducing new registers to re-establish the one-to-one relationship between registers and values at a given time

Register renaming

How resource dependencies are managed
in out-of-order issue or completion

Renaming – example

- Renaming dynamically rewrites the machine code using a larger register set
  - A renamed register is allocated somehow and remains in force until commit
  - Subsequent use of a register name as an operand uses the latest rename of it

$R_3 \rightarrow R_{3b} \rightarrow R_{3c}$

scope R_{3b}

$R3_b$:= R3 op R5

R4 := $R3_b$ + 1

$R3_c$:= R5 + 1

R7 := $R3_c$ op R4

Register renaming

- **Storage conflicts can be removed in out-of-order issue microprocessors by renaming registers**
  - requires additional registers e.g. a rename buffer or extended register file, not visible to the program
  - mapping between logical name and physical location is maintained in hardware while the instructions are executing
- Instructions are executed out of sequence from the instruction window using the renamed registers
  - new physical register allocated on multiple use of same target register name
  - mapping from instruction register to architectural register stored in hardware
  - instructions executing after a rename use the renamed register rather than instruction-specified register as an operand
- A **commit stage** is used to preserve sequential machine state by storing values to architectural registers **in program order**

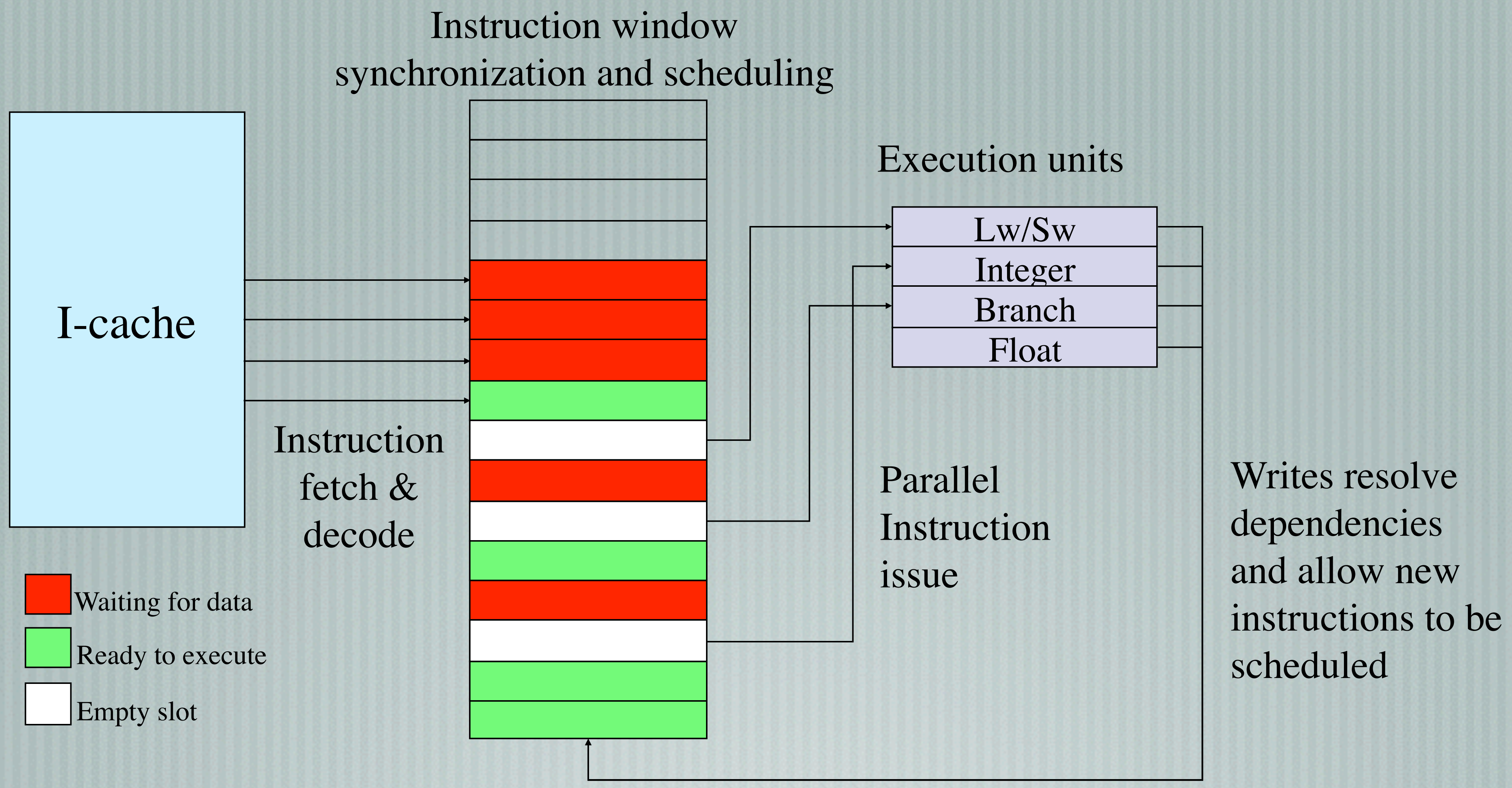
Strategies renaming

- Can either rename at **instruction issue**
  - explicit renaming maps architectural register to physical register used in conjunction with a **scoreboard** to track dependencies
- Can remap implicitly using **reservation stations** at the execute stage and a **reorder buffer** on instruction completion
  - reservation stations use dataflow to manage true dependencies and implicitly rename registers
  - the reorder buffer holds the data until all previous instructions have completed then write it to the architectural register specified

Dynamic scheduling

In out-of-order issue pipelines

Abstract problem


Instruction window
synchronization and scheduling
I-cache
Instruction fetch & decode
Execution units
Lw/Sw
Integer
Branch
Float
Parallel Instruction issue
Writes resolve dependencies and allow new instructions to be scheduled
🟥 Waiting for data
🟩 Ready to execute
⬜ Empty slot

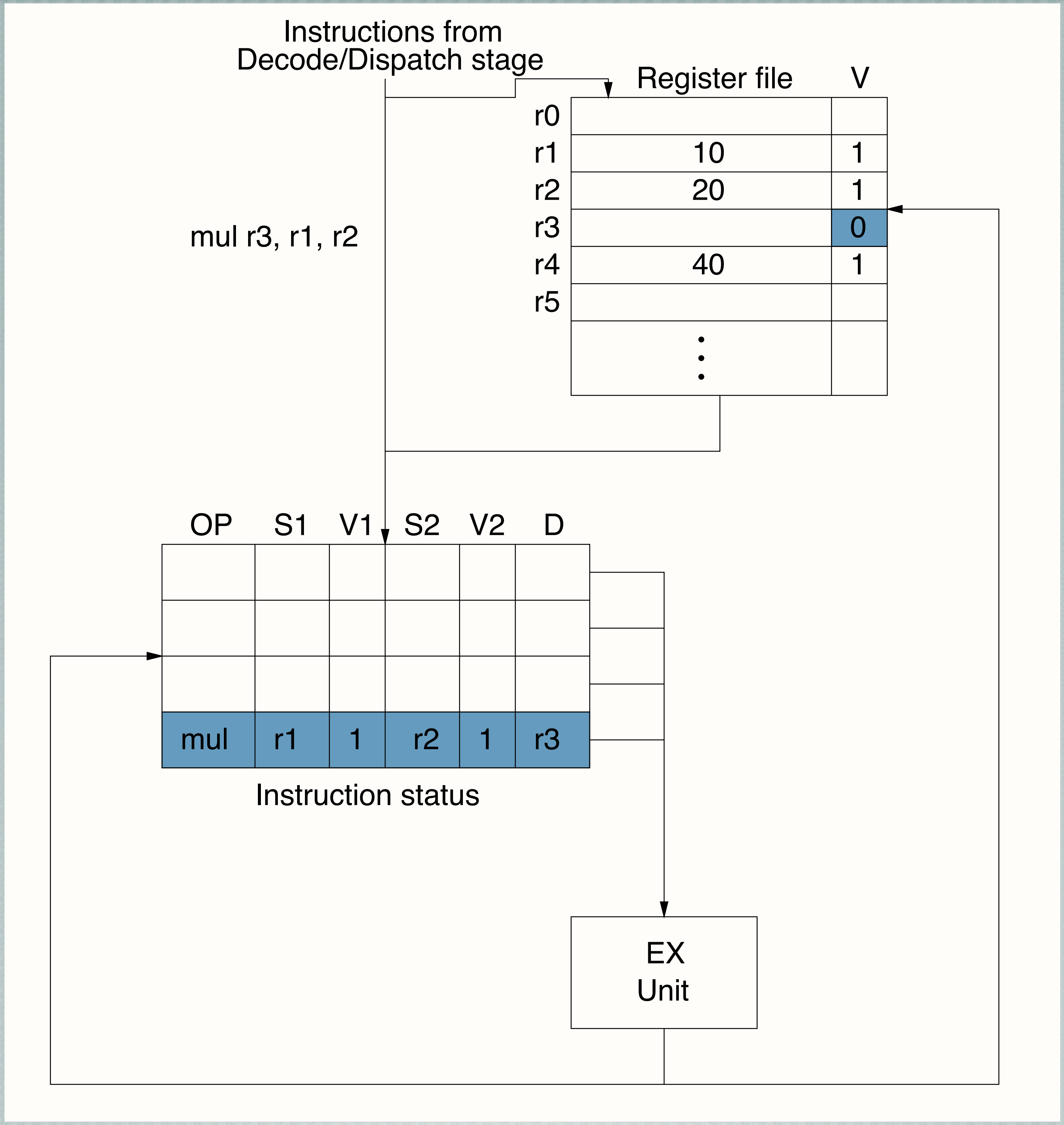
When to issue the instructions?

How to preserve order of side effects?

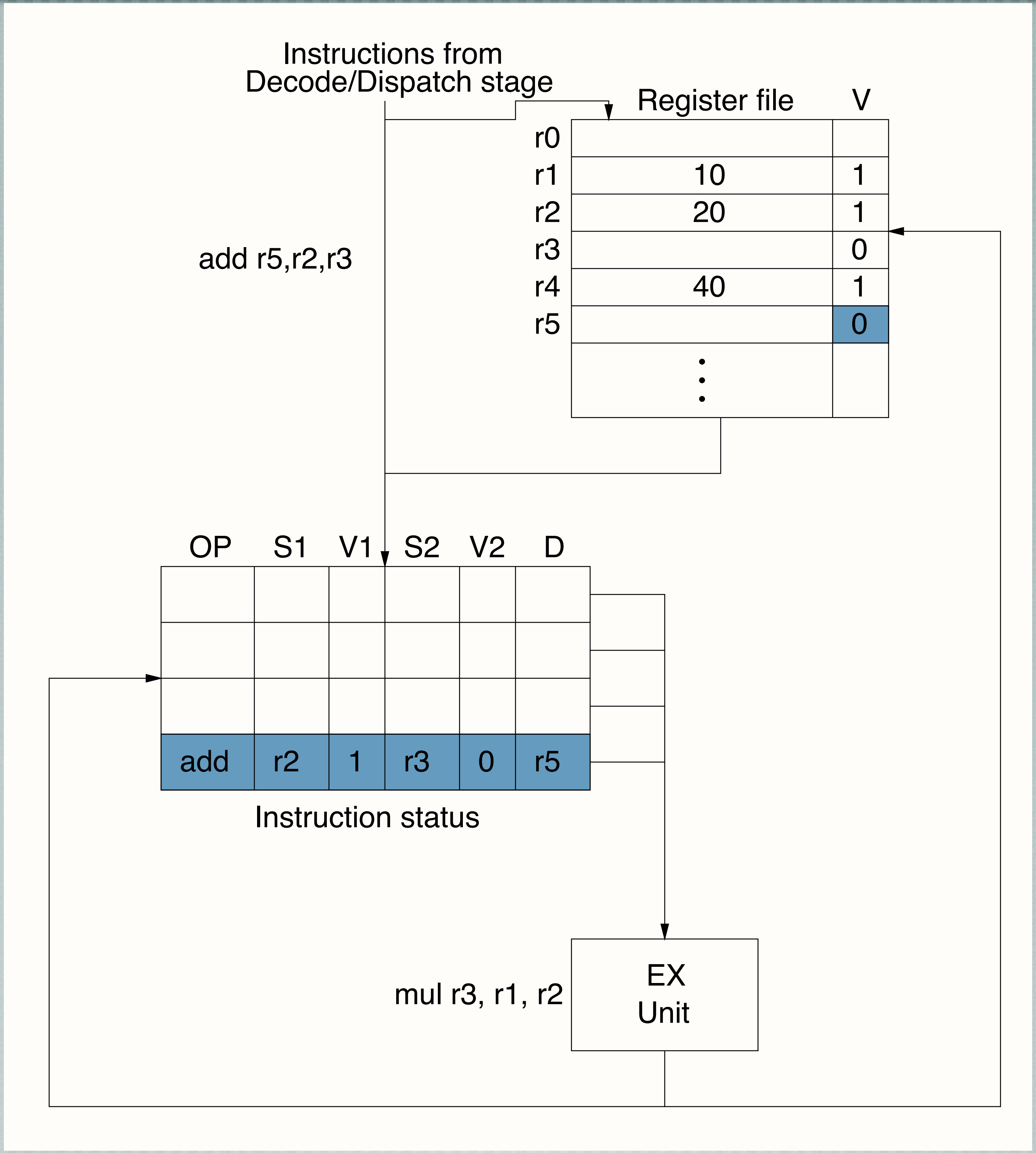
Scoreboarding

- 1 data structure - **centralized** approach
- Scoreboard monitors ready instructions, registers and functional units
  - Issues instructions when source operands and functional units are available
  - Registers include bit indicating their validity
    - At issue, if destination reg. is valid, then mark it as invalid. Otherwise block (WAW hazard). Validate bit at WB while checking for WAR hazard
    - If source reg. is invalid, then block (RAW hazard)
  - Explicit register renaming to avoid WAW and WAR hazards

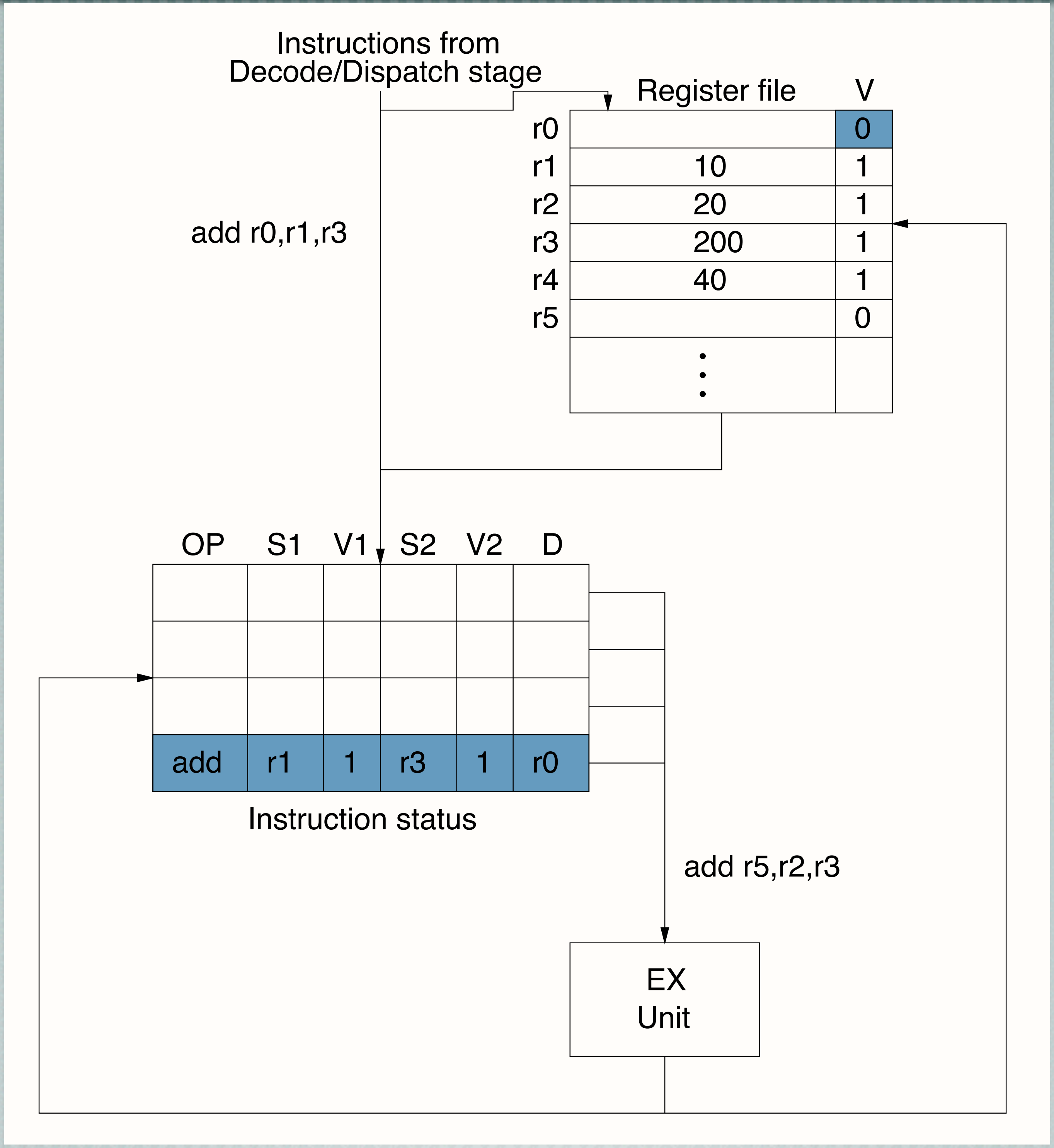
Scoreboarding



Scoreboarding



Scoreboarding



Scoreboarding

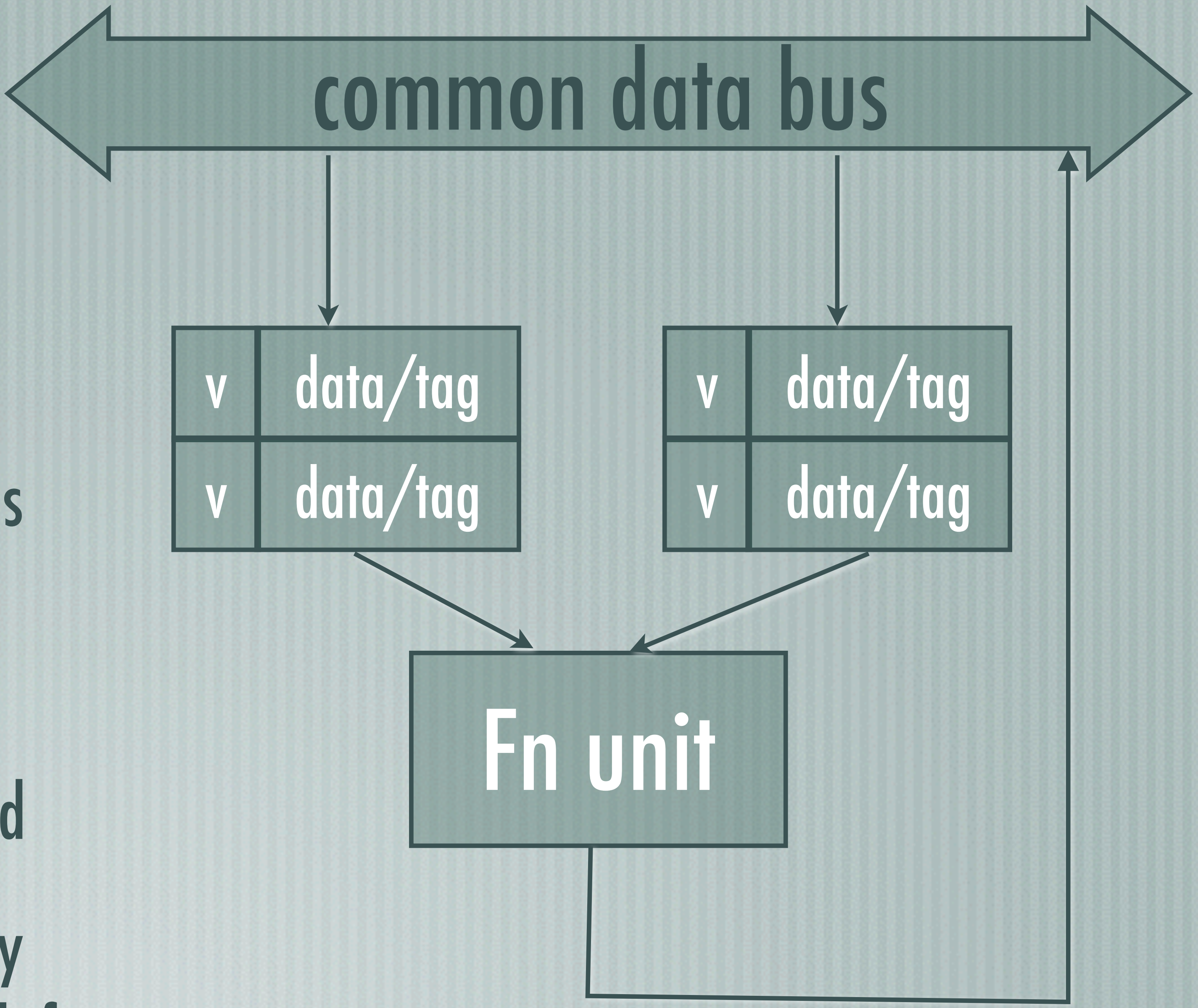
- More information:
  - http://www.cs.umd.edu/class/fall2001/cmsc411/projects/dynamic/example1-2.html
  - H&P, Appendix A.7 - Dynamic Scheduling with a Scoreboard

Reservation stations & Tomasulo

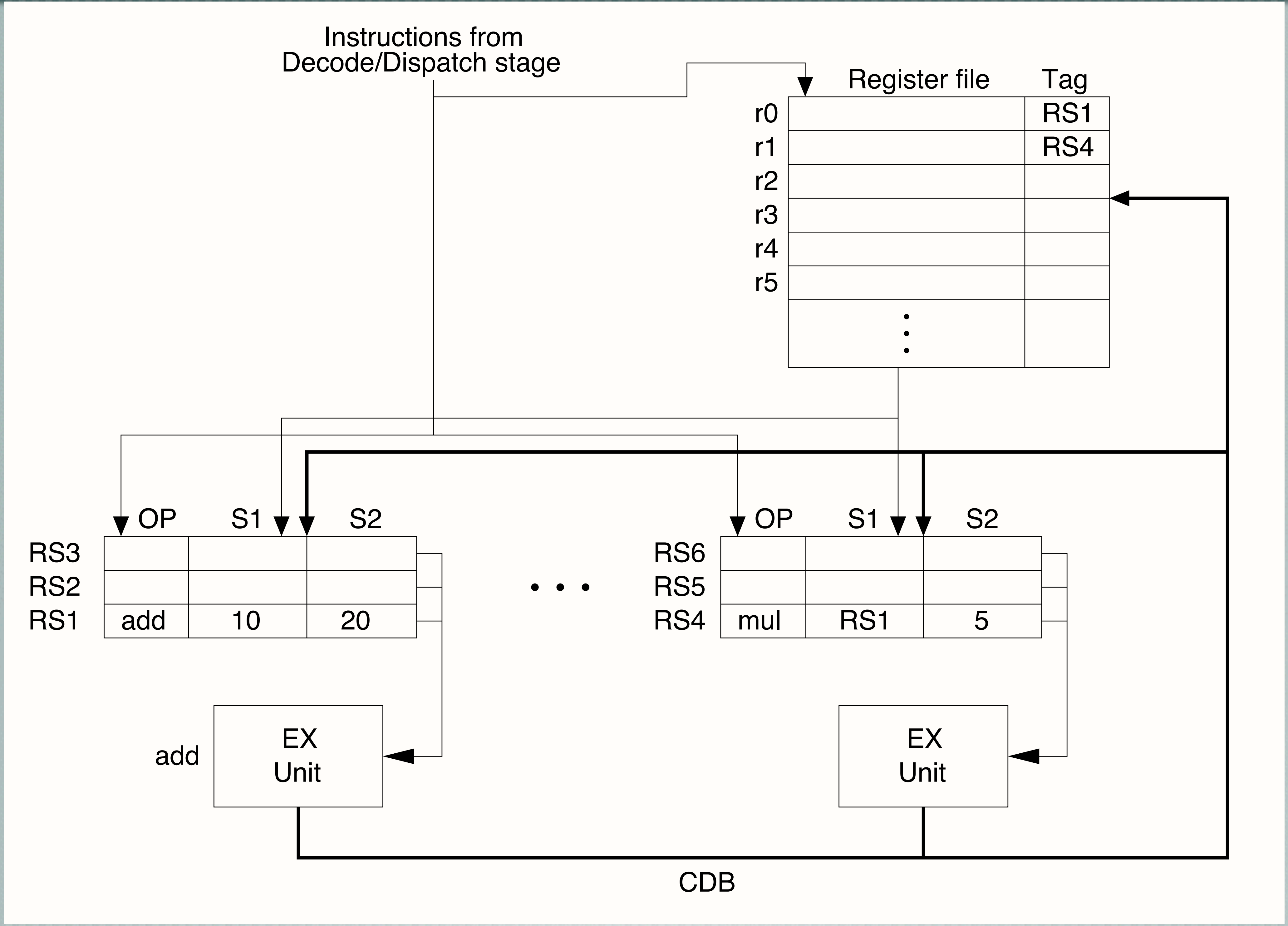
- Basic idea: dataflow read-blocking, write-resume
- **Distributed control structure using reservation stations**
- **implicit** renaming of registers
  - if operand not available: pointer to producing reservation station (**tag**) is stored instead of register ID
  - these tags are matched with result tags using a common data bus
  - results broadcast to all reservation stations for RAW
- **WAR and WAW hazards eliminated by register renaming**

Reservation stations

- tag identifies the **source** of the data – i.e. which reservation station entry
- v is a **valid** bit
- Instructions can be issued before data is available, but valid bit is set to 0
- Only when an instruction has two valid operands the functional unit is activated
- Every result matches all l-r tags in every reservation station and data is grabbed if a match occurs



Tomasulo's algorithm



Tomasulo’s algorithm

- Issue - get instruction from Op Queue
  - if reservation station is free (no structural hazard) control issues instruction and operands (renames registers)
- Execution - operate on operands (EX)
  - when both operands ready then execute; if not ready, watch Common Data Bus for result
- Write result - finish execution (WB)
  - write on Common Data Bus to all awaiting reservation stations; free reservation station entry
- Normal data bus = data + destination; **Common data bus = data + source**
  - e.g. 64 bits of data + 4 bits of reservation station ID

More information

- Demo using web applet University of Edinburgh
  - http://www.dcs.ed.ac.uk/home/hase/webhase/demo/tomasulo.html
- Another demo from University of Massachusetts
  - http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo/AppletTomasulo.html

Reordering

In out-of-order issue pipelines

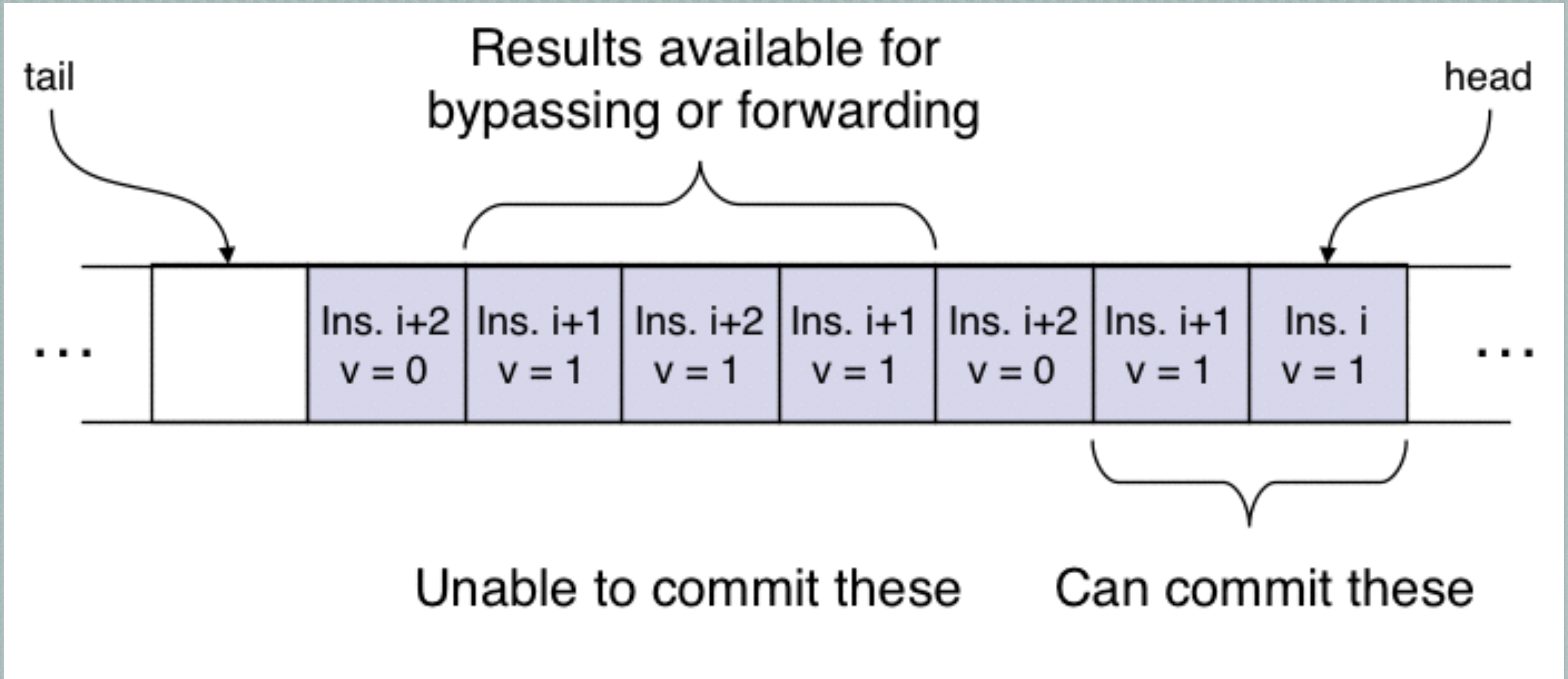
Machine state

- Issuing instructions out of order: sequential-order machine state is lost
  - this can cause problems when exceptions occur
  - how can we define the machine state if many instructions are in flight - not necessarily in program order?
- To checkpoint the sequential state we must **retire** or **commit** instructions only **when all previous instructions have finished**
  - only at this stage the result of an instruction can be written into the architectural register
- This is managed in a **reorder buffer** (ROB) that provides sequential state at the end of the pipeline

Reorder buffer

- The reorder buffer stores information about all instructions executing between issue and retire stages
  - it can also store the results of those instructions pending a write to the architectural register file, which is another implicit form of register renaming
- The reorder buffer is a **queue or FIFO (typically circular)**
  - instructions are written to it at the tail in program order at the issue stage
  - instructions are removed from it at the head but only when they have completed execution
  - at this stage, the results can be safely written to the architectural register file: the instruction is then said to be **retired** or **committed**

Reorder buffer



Flag v determines whether result has been written or not

Memory load/store reordering

- Note that although a consistent register state may be identified using a reorder buffer, the memory state is a different matter
  - this is because of memory delays, cache writeback strategies etc.
- Modern microprocessors hold memory reads and writes in buffers similar to reservation stations
  - these will match reads with writes, and also bypass data
    so that a read to a location in the buffer that has not yet been written can provide its value to the memory read
- This allows load/store re-ordering and can improve locality of memory accesses

Memory load/store reordering

- Can safely allow a **load to bypass a store** as long as the addresses of load and store are different
- If addresses are not know then either
  - **do not allow bypassing**, or
  - **speculatively bypass** the store but **squash the load** if the address turns out to be the same
- Can also allow **loads to bypass loads on cache misses**
  - This is called a **lock-up free cache** but it can **complicate the cache coherence protocol**

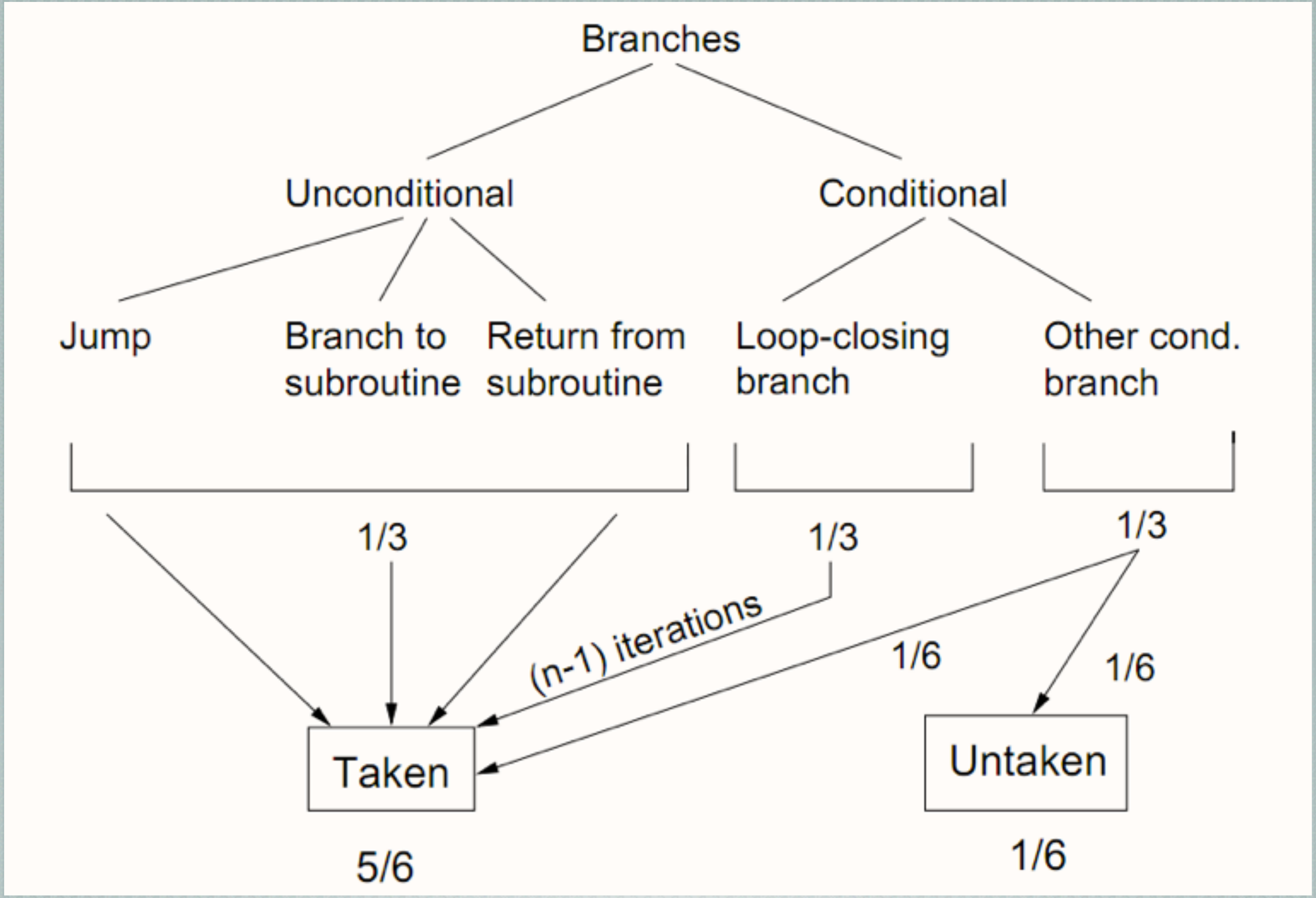
Branch prediction

Dealing with control hazards

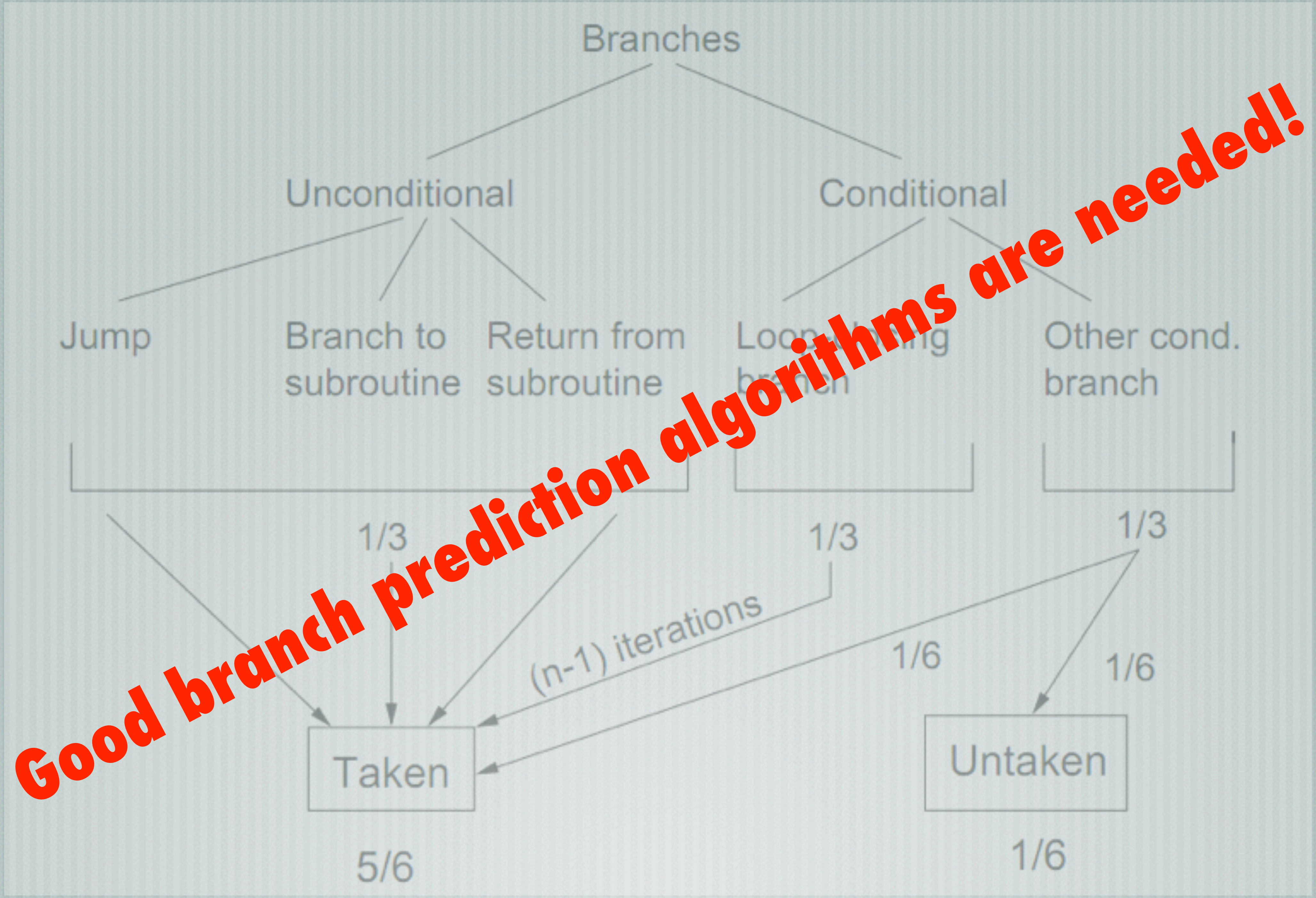
Control hazards revisited

- Superscalar pipelines are typically super-pipelined and have many stages ≈ 10-20
- They also have wide issue widths ≈ 4-8
  - we may therefore have 40-160 instructions in flight
- The latency to resolve a branch condition is large
  - 15 cycles for a conditional branch in Intel Core i7!
  - hence many pipeline slots will be filled with instructions from the wrong target
  - this has to be cleaned up when the wrong target is chosen including register renames

Grohoski's estimate



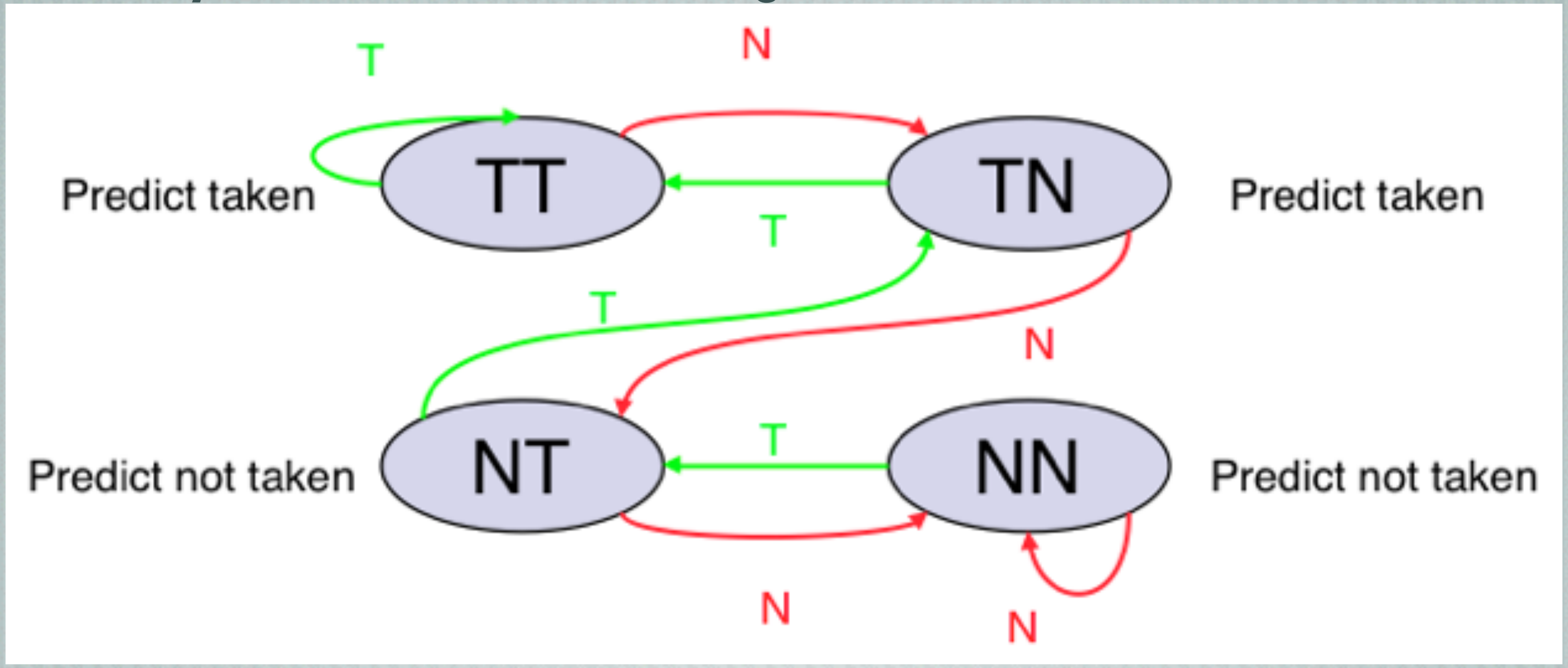
Grohoski's estimate



Good branch prediction algorithms are needed!

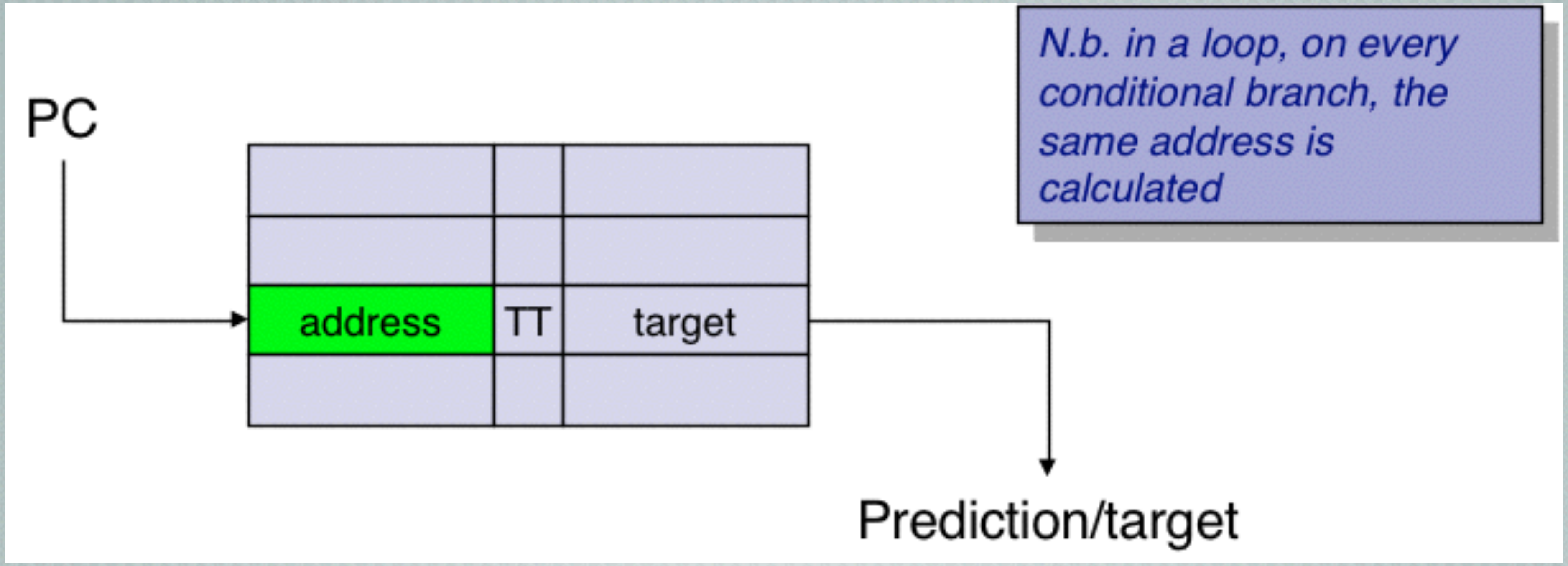
Bimodal predictors

- Use two bits to represent the last two attempts (≈ 90% accurate) ... there are various schemes
  - TT TN NT NN are the predictor states
  - E.g. change prediction only if miss-predict twice but return in one step
    - this is only one of several strategies



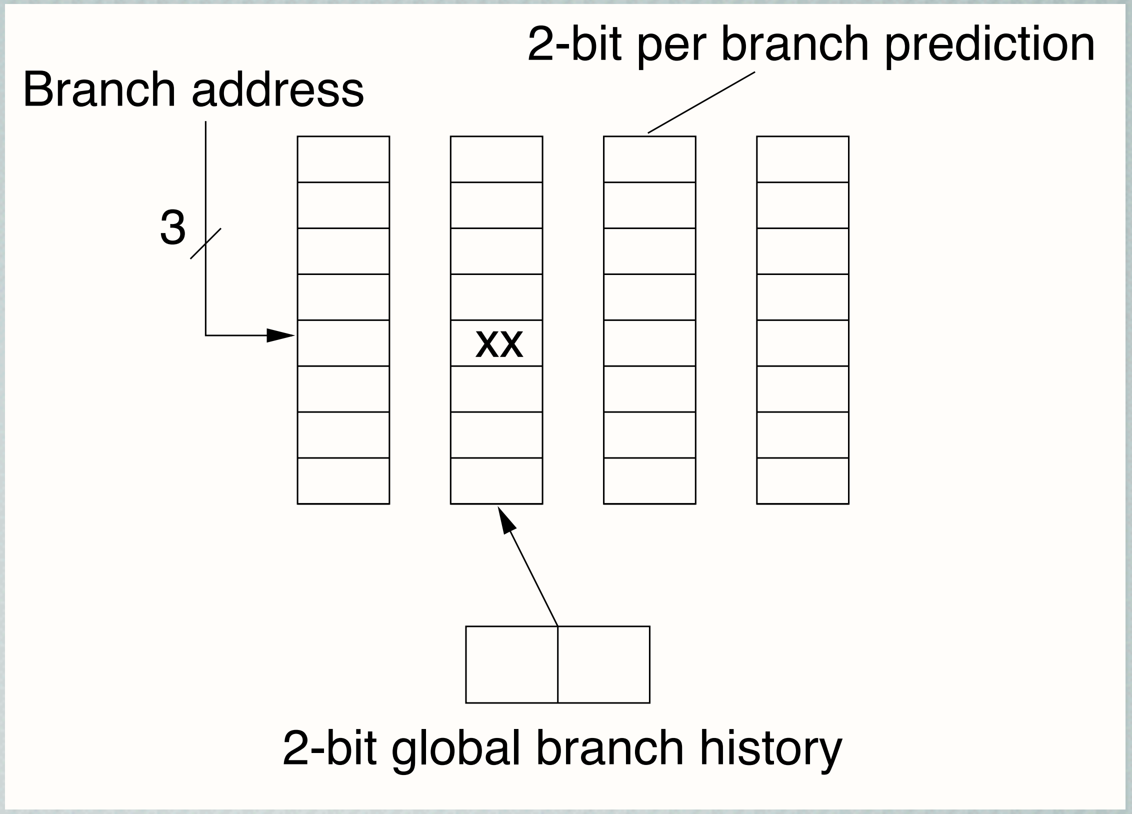
Branch history buffers

- Stores the **prediction state in a table**, either associatively addressed or indexed on small number of address bits
  - Can also store branch target if it is associative
  - Get prediction at IF stage and update prediction when condition is resolved


PC
address
TT
target
Prediction/target
N.b. in a loop, on every conditional branch, the same address is calculated

Correlated or global predictors

- There may be correlation between different branches
- Normally predictors are indexed on address bits of the branch instruction
- Correlation can be tracked by so-called global predictors that maintain a register of the history of recent branches taken and use this to address the prediction

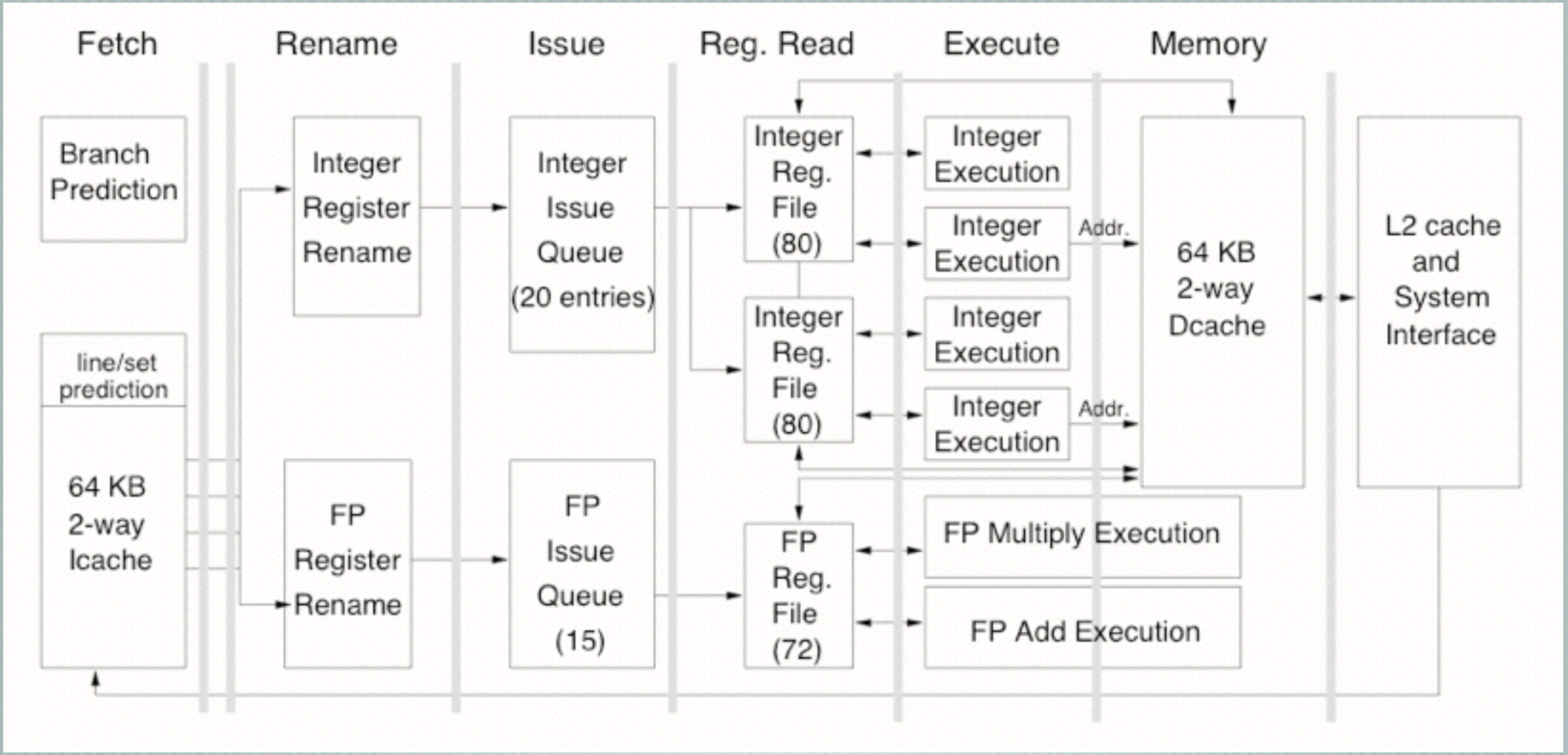


Example processors

DEC Alpha 21264 (late 1990's)

- The EV6 was the last Alpha microprocessor to be manufactured
  - Alpha has a very clean RISC ISA that uses separate integer and floating-point register files
  - Alpha was unique in supporting a high clock rate and short pipeline through good ISA and silicon design
- EVA6 uses out-of-order issue in a 7 stage pipeline
  - 4 instructions per cycle can be fetched (speculatively)
    and up to 6 instructions issued out of order
  - sophisticated branch predictor
  - uses scoreboarding and explicit renaming techniques to track dependencies and avoid false dependencies

Alpha 21264 pipeline

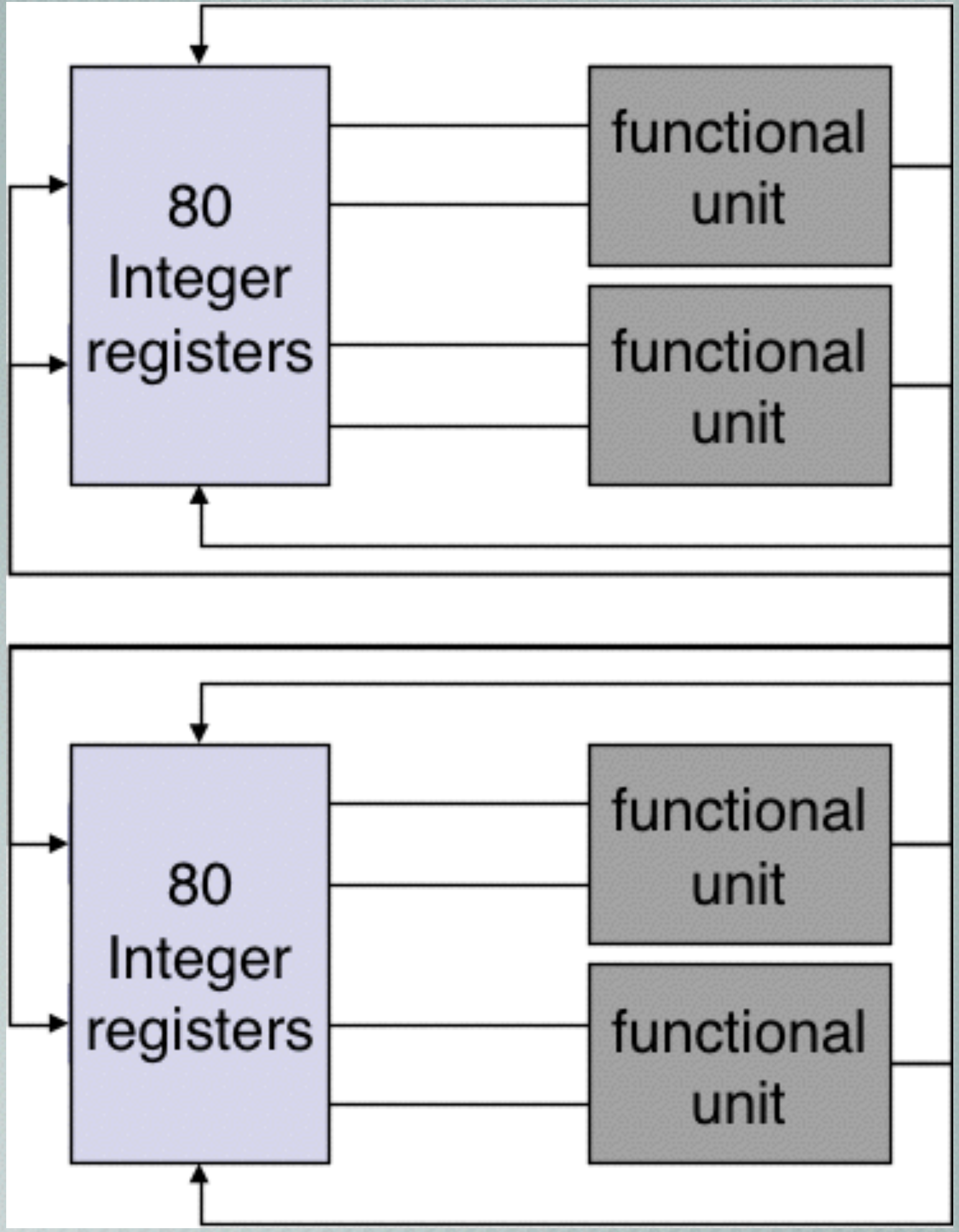


Dual RF to save ports

- Later we will see that the register file area grows quadratically with number of ports
  - number of ports proportional to the number of functional units allowed to write in one cycle
  - multiple issue processors requires two read and one write port per concurrent functional unit
- Ports can be minimized by
  - separating floating point and integer operations as long as data can be moved between the two
  - duplicating registers
- The Alpha uses both techniques: 72 FP registers and 80 integer registers which are duplicated to minimize ports

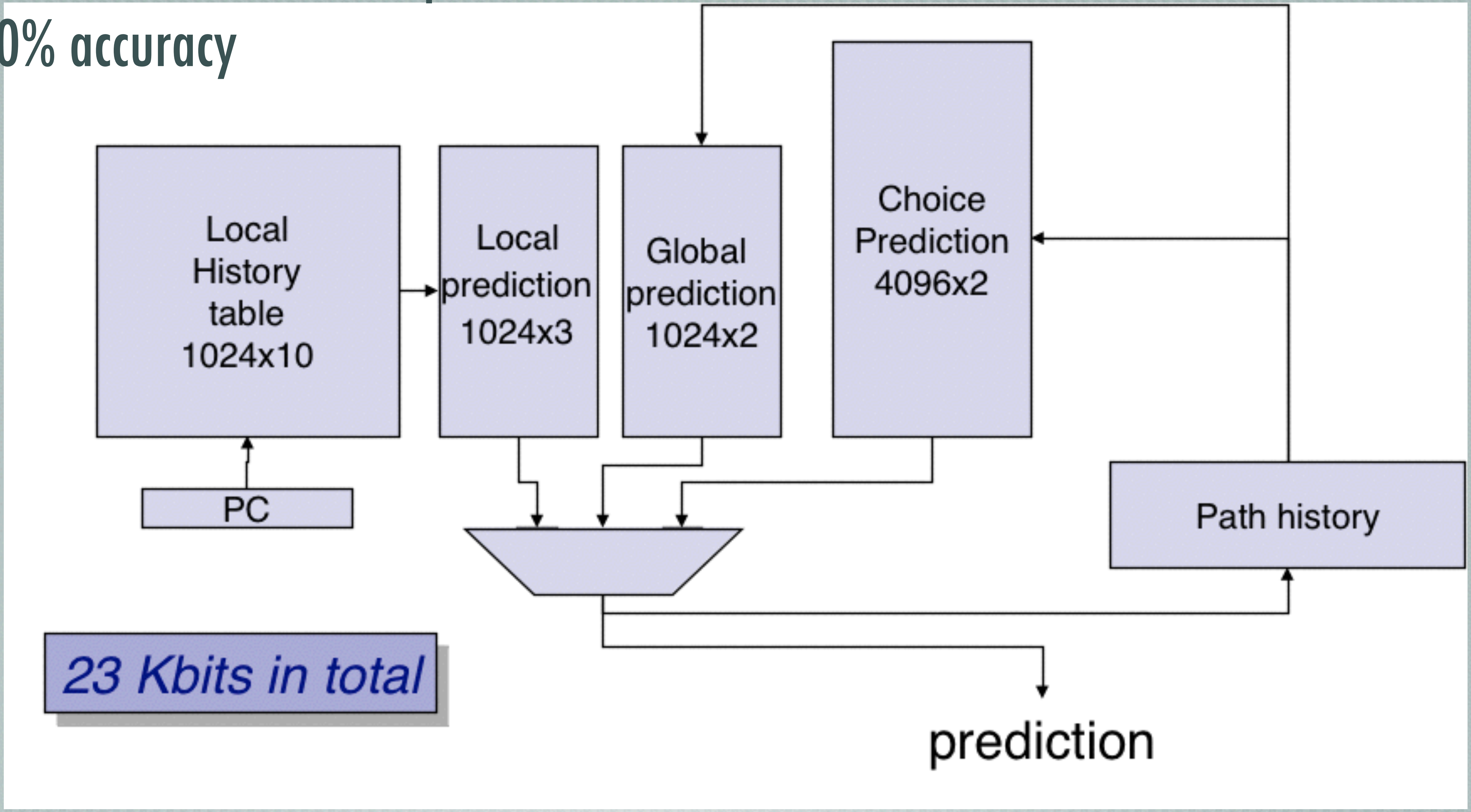
Dual RF to save ports

- Alpha groups 2 functional units with each of 2 register files
  - 4 read and 4 write ports
  - Area = $2 \cdot c \cdot 8^2$ =128c
- Instead of having one file (8 read and 4 write ports)
  - Area = $c \cdot 12^2$ = 144c
- Only a small saving in area,
  but this also aids locality of signals on the critical read-to-functional unit path
- c is a constant based on line width/spacing



Alpha branch prediction

- Hybrid tournament branch predictor
- 90-100% accuracy



Alpha instruction fetch

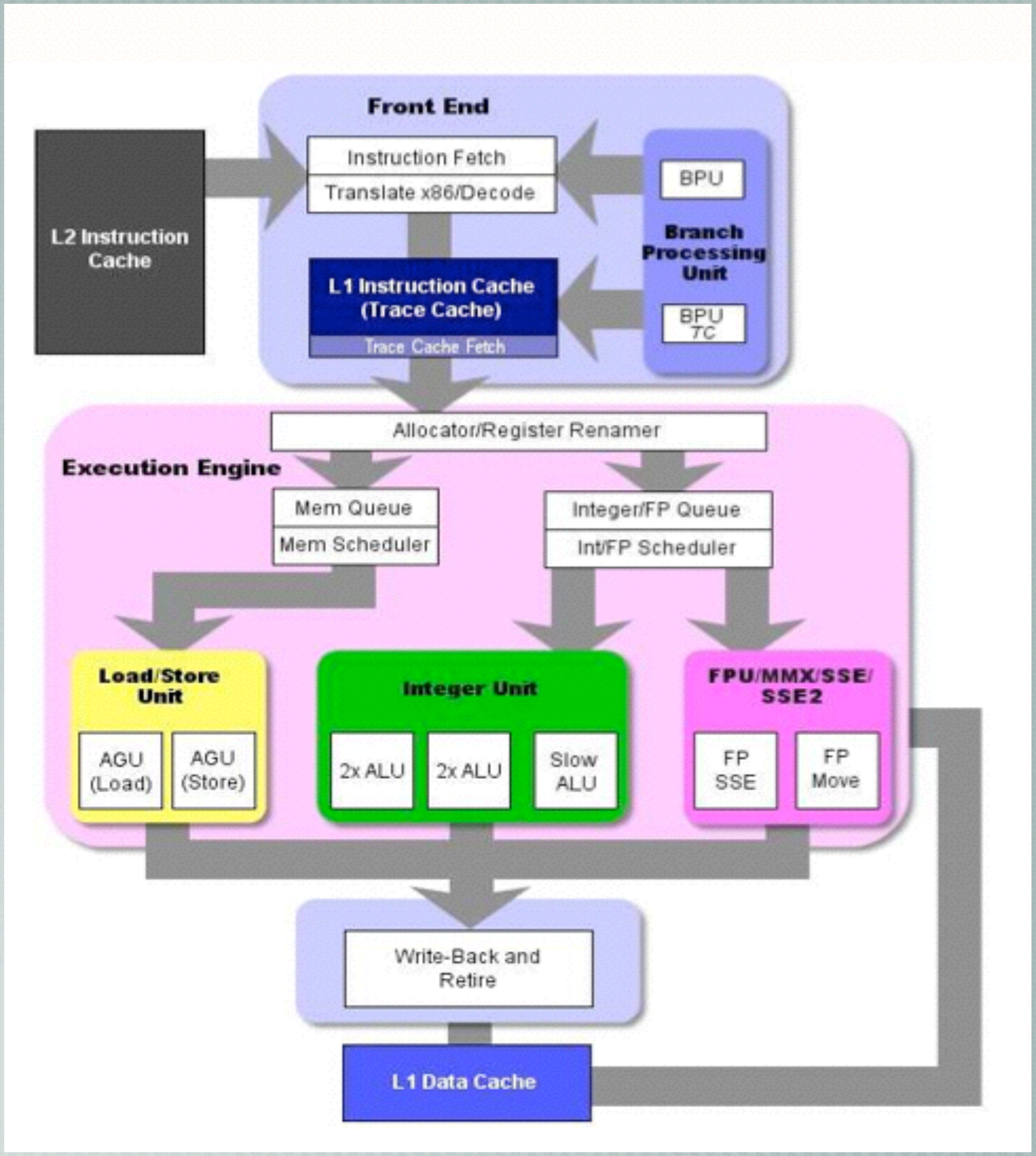
- I-cache is a 2-way set associative cache, 16 bytes cache lines
  - a cache block fetch is four instructions
  - it uses **line and set prediction**
    - it predicts where to fetch the next block from
  - **accuracy of 85%**
  - line miss-prediction cost typically 1 cycle bubble

Intel Pentium 4 (early 2000's)

- This has a very deep pipeline and hence high clock rate - 4GHz
- Problems exacerbated due to the X86 CISC instruction set, which is not suitable for pipelining
- X86 instructions are translated into µops
  - these are regular and uniform - like a traditional RISC ISA
  - trace cache caches translated µop sequences along program execution paths

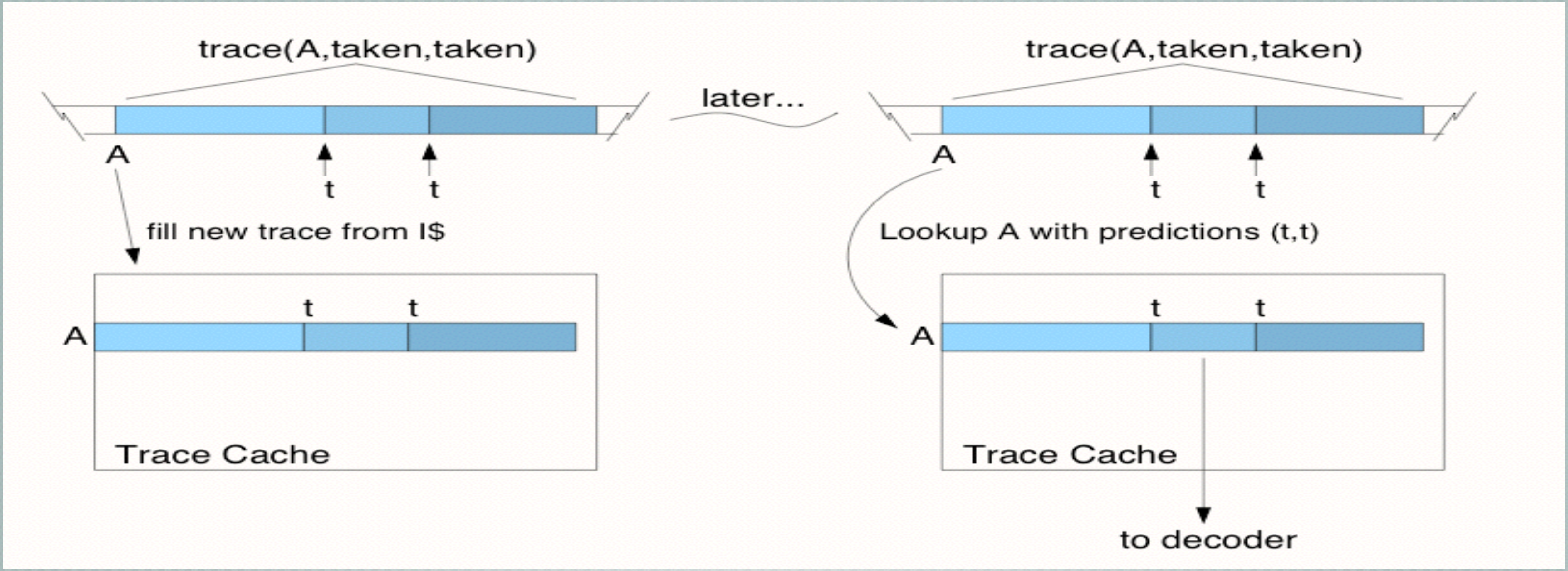
Pentium 4 pipeline overview

- Decode stage translates X86 instructions into μops
- Trace cache stores μop traces
  - i.e. a cache of instructions as executed rather than as stored in memory
- Branch history table + branch target buffer + static prediction
- Adopts simultaneous multi-threading - hyperthreading (fetches instructions simultaneously from 2 threads, cf later)



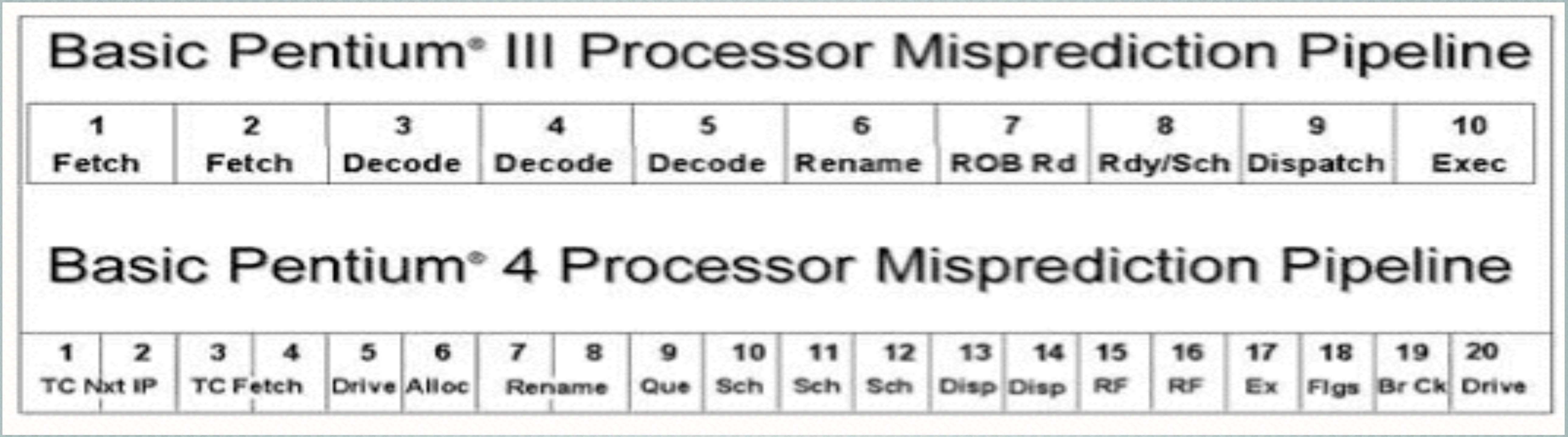
Pentium 4's trace caches

- Cache **actual instruction sequences** rather than program sequences
  - i.e. the sequence of instructions executed rather than their order in memory



Pentium 4’s pipeline stages

- 6-way out-of-order execution 20 stage pipeline
  - 2 generations compared below (note the superpipelining)
- 126 entry reorder buffer (registers and result status)



Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium® 4 Processor Misprediction Pipeline

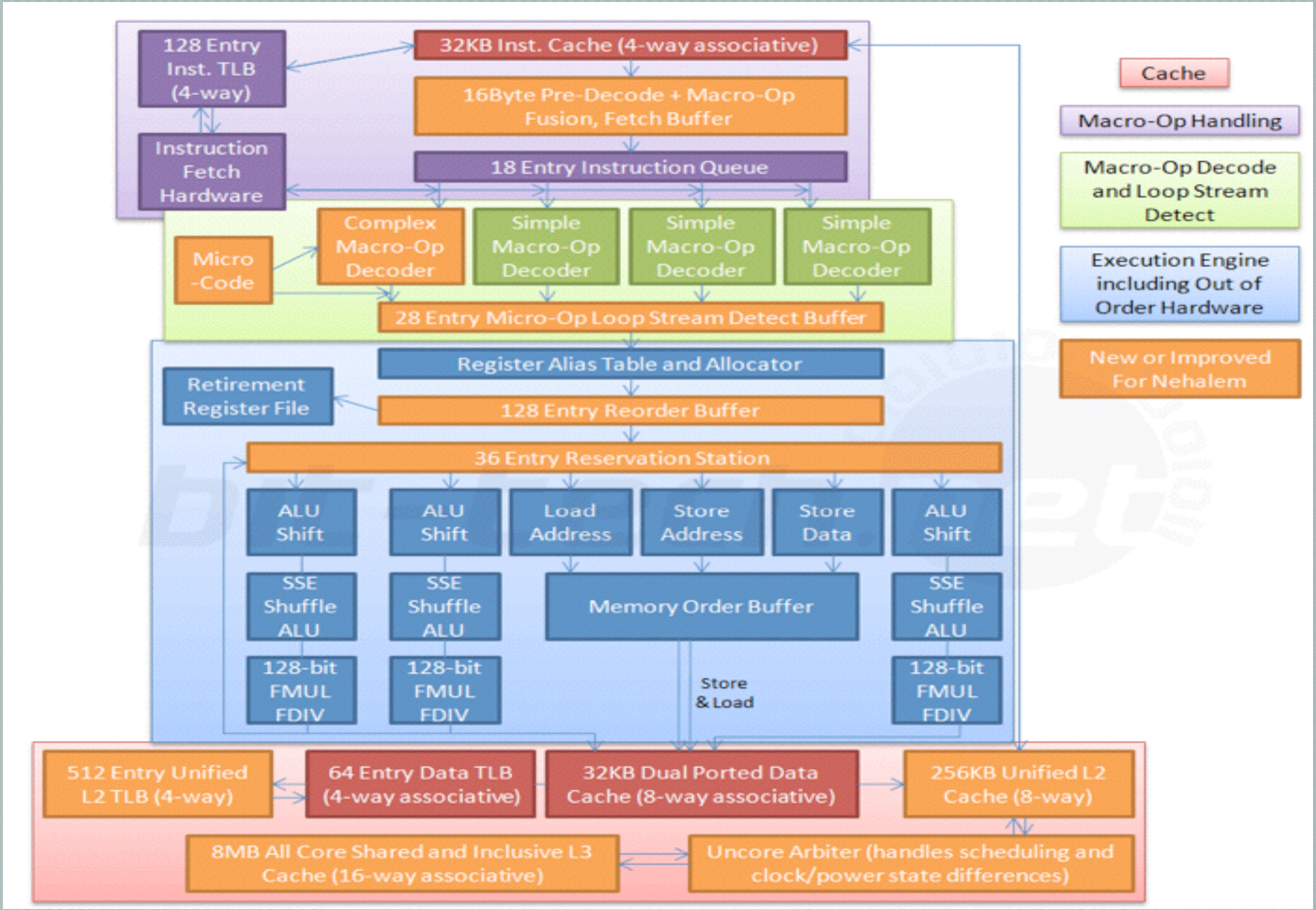
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

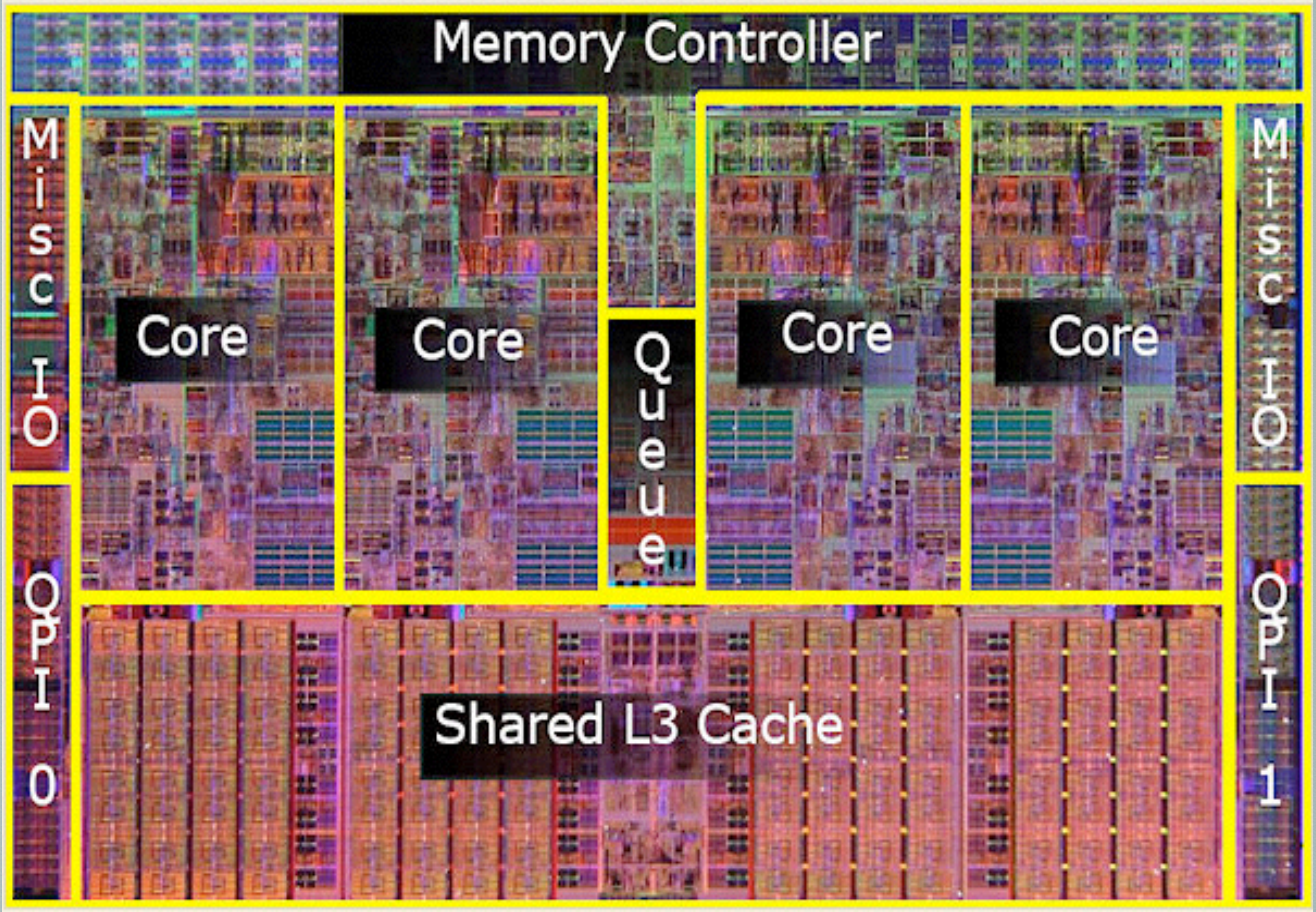
Intel moving to multi-core

- Pentium 4D increased the pipeline length to 31 stages in an attempt to push clock speeds to 4GHz
- Since then Intel has moved to multi-core - shorter pipelines - slower clocks
  - 2006 Core 2 duo 2.93 GHz, 291M transistors, 65nm
  - 2007 Quad core Xeon 2.66GHz, 582M transistors, 65nm
  - 2007 Quad core Xeon Penryn >3GHz, 820M transistors, 45nm
  - 2008 Core i7 Nehalem <3GHz, 731M transistors, 45nm
  - 2011 Core i7 Sandy Bridge ≈3GHz, 995M transistors, 32nm
  - 2012 Core i7 Ivy Bridge, >3GHz, 1.4B transistors, 22nm

Intel Core i7 - Nehalem ('08-'10)

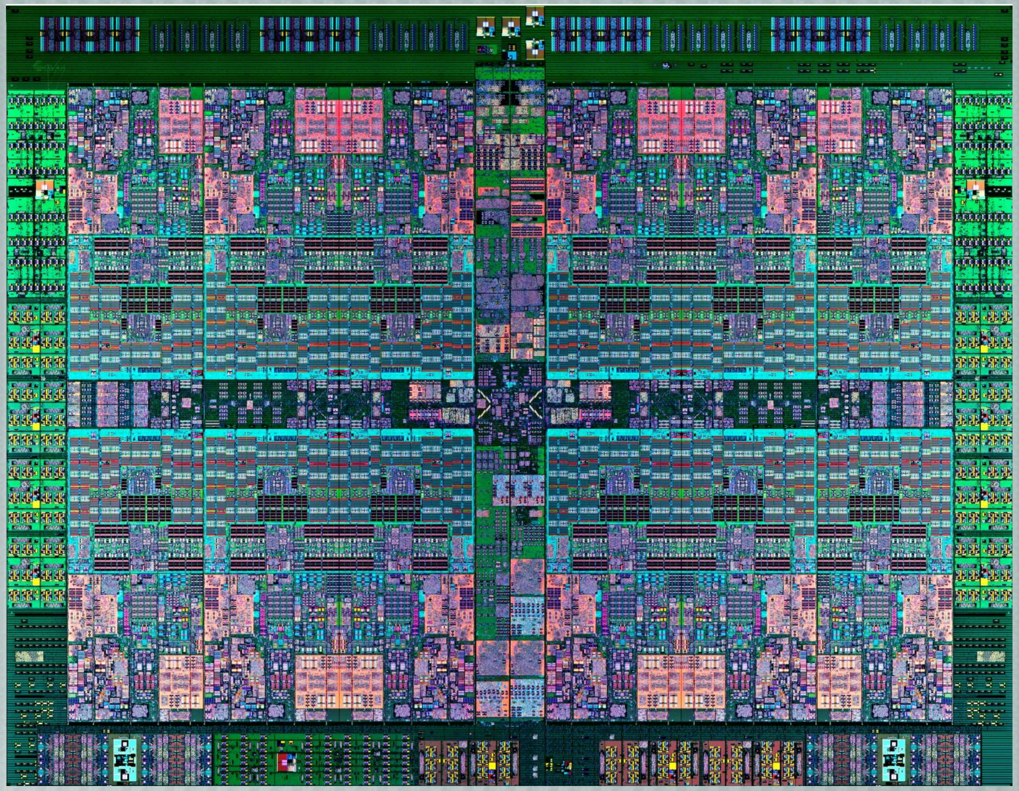
- 2 or 4 core up to 3GHz
- 14 stage pipeline with stream prefetching
- Return of hyper-threading
- 32 KB L1 I-cache & 32 KB L1 D-cache (8-way set associative)
- 256 KB L2 cache per core (8-way set associative)
- Shared 8 MB L3 cache (16-way set associative)

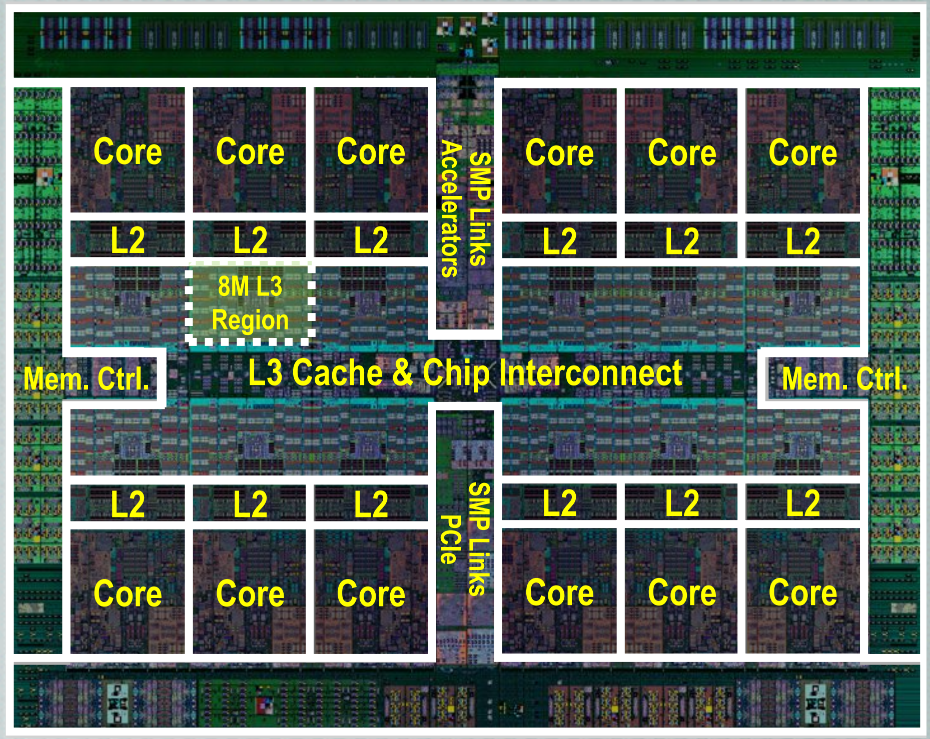

128 Entry Inst. TLB (4-way)
32KB Inst. Cache (4-way associative)
16Byte Pre-Decode + Macro-Op Fusion, Fetch Buffer
Instruction Fetch Hardware
18 Entry Instruction Queue
Micro -Code
Complex Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
Simple Macro-Op Decoder
28 Entry Micro-Op Loop Stream Detect Buffer
Register Alias Table and Allocator
Retirement Register File
128 Entry Reorder Buffer
36 Entry Reservation Station
ALU Shift
ALU Shift
Load Address
Store Address
Store Data
ALU Shift
SSE Shuffle ALU
SSE Shuffle ALU
Memory Order Buffer
SSE Shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & Load
128-bit FMUL FDIV
512 Entry Unified L2 TLB (4-way)
64 Entry Data TLB (4-way associative)
32KB Dual Ported Data Cache (8-way associative)
256KB Unified L2 Cache (8-way)
8MB All Core Shared and Inclusive L3 Cache (16-way associative)
Uncore Arbiter (handles scheduling and clock/power state differences)
Cache
Macro-Op Handling
Macro-Op Decode and Loop Stream Detect
Execution Engine including Out of Order Hardware
New or Improved For Nehalem


Memory Controller
Misc IO
Misc IO
Core
Core
Queue
Core
Core
QPI 0
QPI 1
Shared L3 Cache

IBM POWER8 (2014)

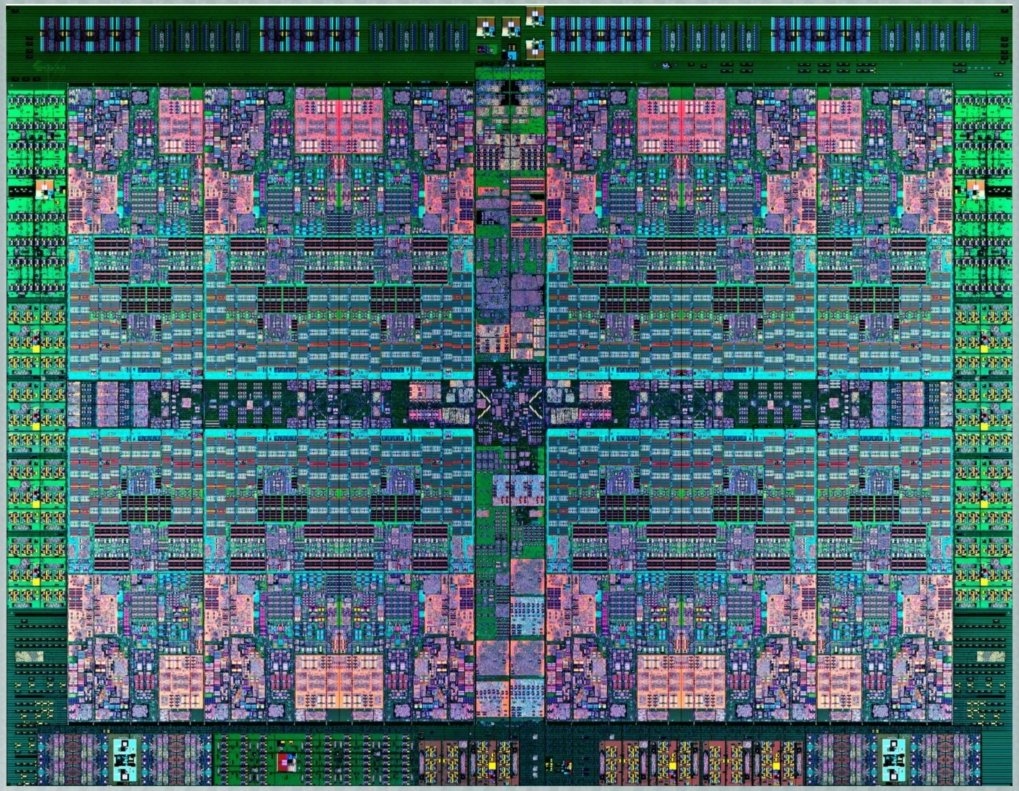
- 12 cores on a chip
- Each core can execute up to 8 HW threads (SMT)
- Fetch, decode, dispatch and commit 8 instructions/cycle
- Issue 10 instruction/cycle to 16 execution units

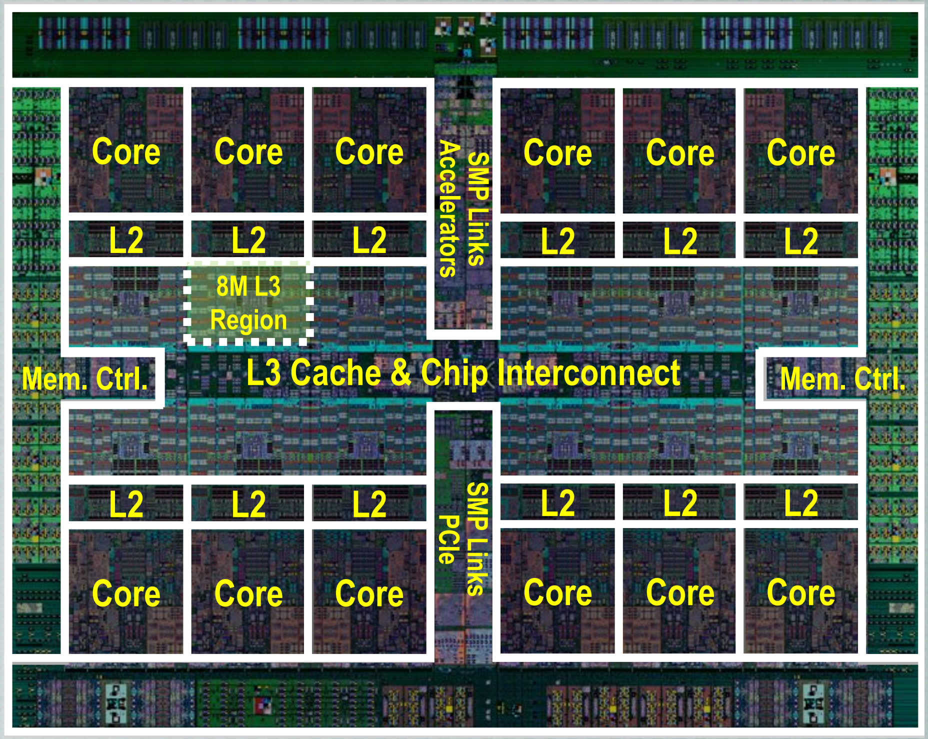




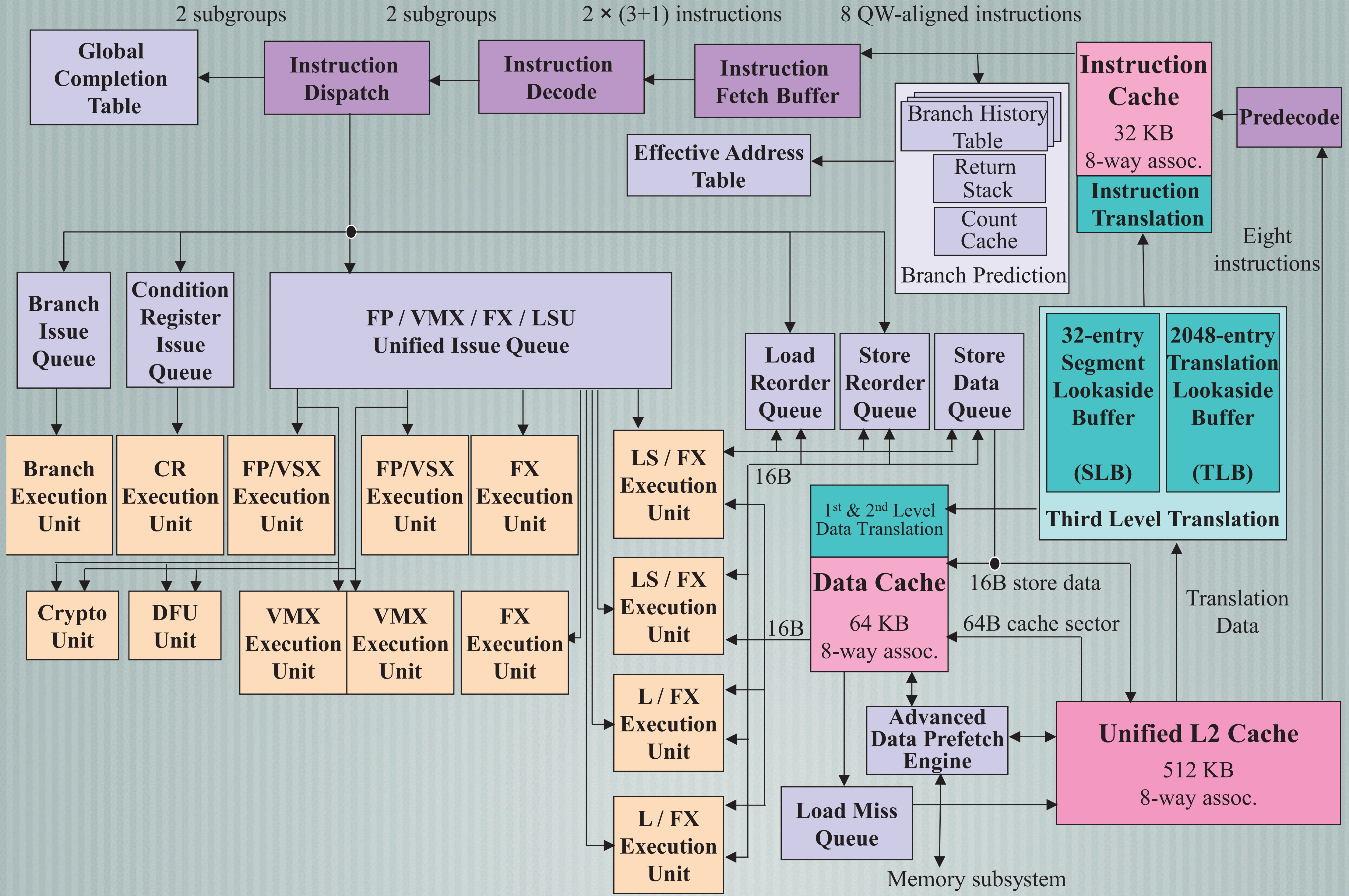
IBM POWER8 (2014)

- Can perform 4 loads (or 2 loads+2 stores) per cycle
- Uses combination of reservation stations and ROB
  - Max. 224 in-flight instructions after dispatch
- HW-support for Transactional Memory (TM)
- >2000 physical registers to support renaming and TM checkpointing
- 64KB L1 D-cache and 32KB L1 I-cache
- 512KB L2 cache, 96MB L3 cache
- Up to 8 outstanding L1 cache misses
- HW-initiated instruction and data prefetch
- Uses hybrid branch predictor like Alpha





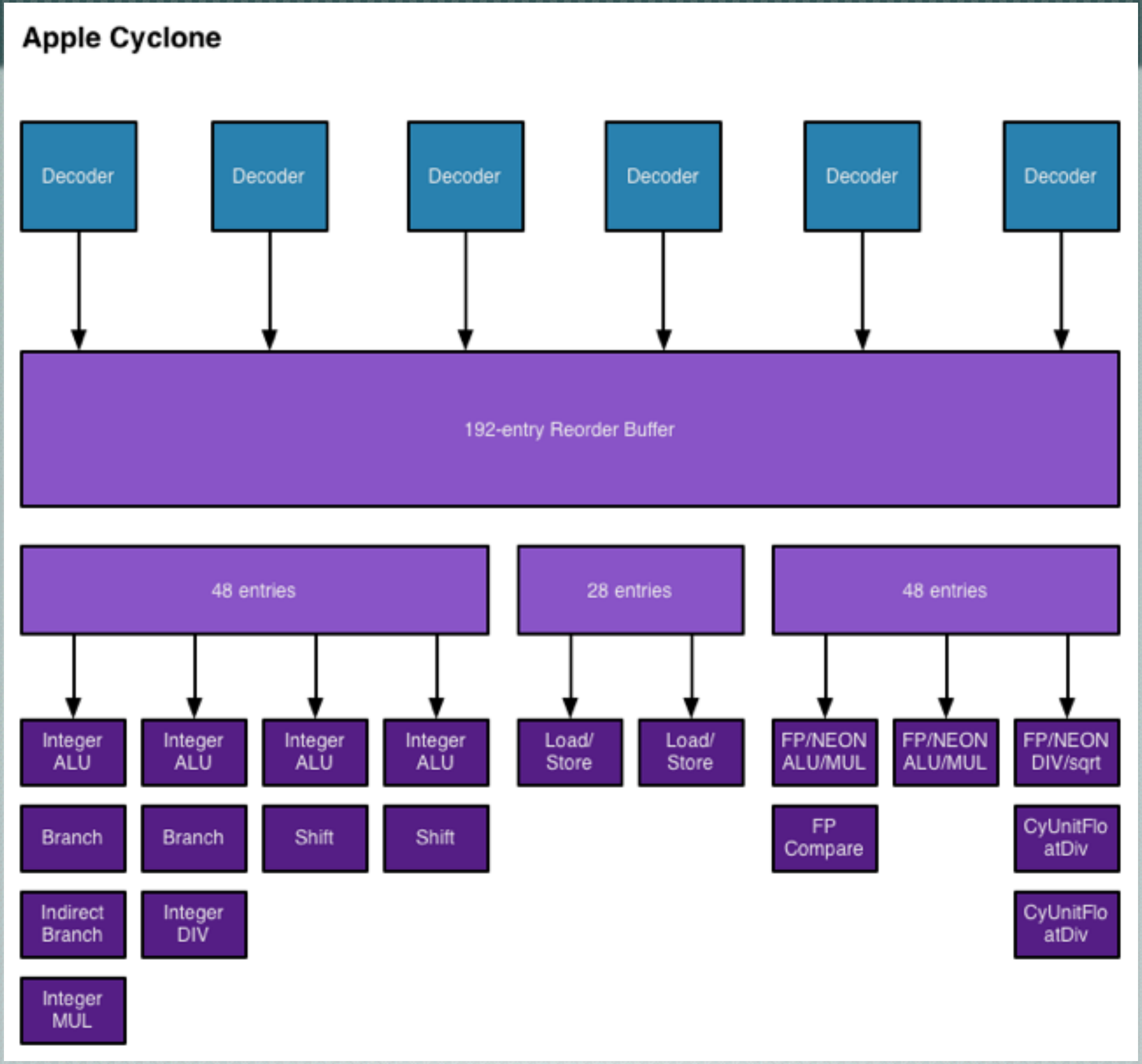
IBM POWER8 (2014)



Apple A7/A8 for iPhone ('13/'14)

- Clock 1.4 GHz, dual-core
- 6 instruction-wide out-of-order execution core
- 12 execution units, 192-entry ROB
- 64KB L1 I/D-caches, 1MB L2 cache and 4MB L3 cache
- Quad-core GPU on chip

Apple A7/A8 for iPhone ('13/'14)



Summary

Summary

- Out-of-order issue exploits **instruction-level concurrency from a sequential instruction stream** (implicit concurrency)
  - it attempts to achieve a large number of instructions executing simultaneously in multiple functional units
  - instructions are **dynamically scheduled** at the issue by **executing out of order** while **honouring dependencies**
  - dependencies introduced by completing and issuing instructions out of order are removed by **register renaming**
- Out-of-order issue can be
  - **costly** and **only appropriate for low levels of concurrency**
  - **inefficient on irregularly branching code**
- Difficult to get an IPC of much more than 2 even for 8-way issue