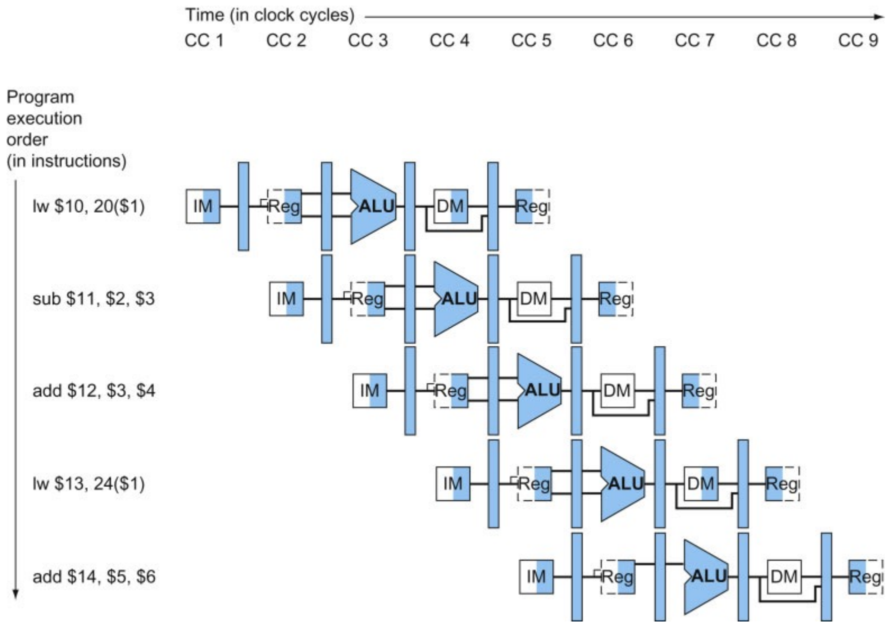
ILP (Instruction Level Parallelism) (section 4.11)

peak CPI of (single-issue) pipeline processor

- **n 개의 명령어**
- **p 개의 pipeline stages 일 때**
- **peak CPI (clock cycles per instruction) = 1**



- **IPC (Instructions Per Cycle) = 1/CPI**

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

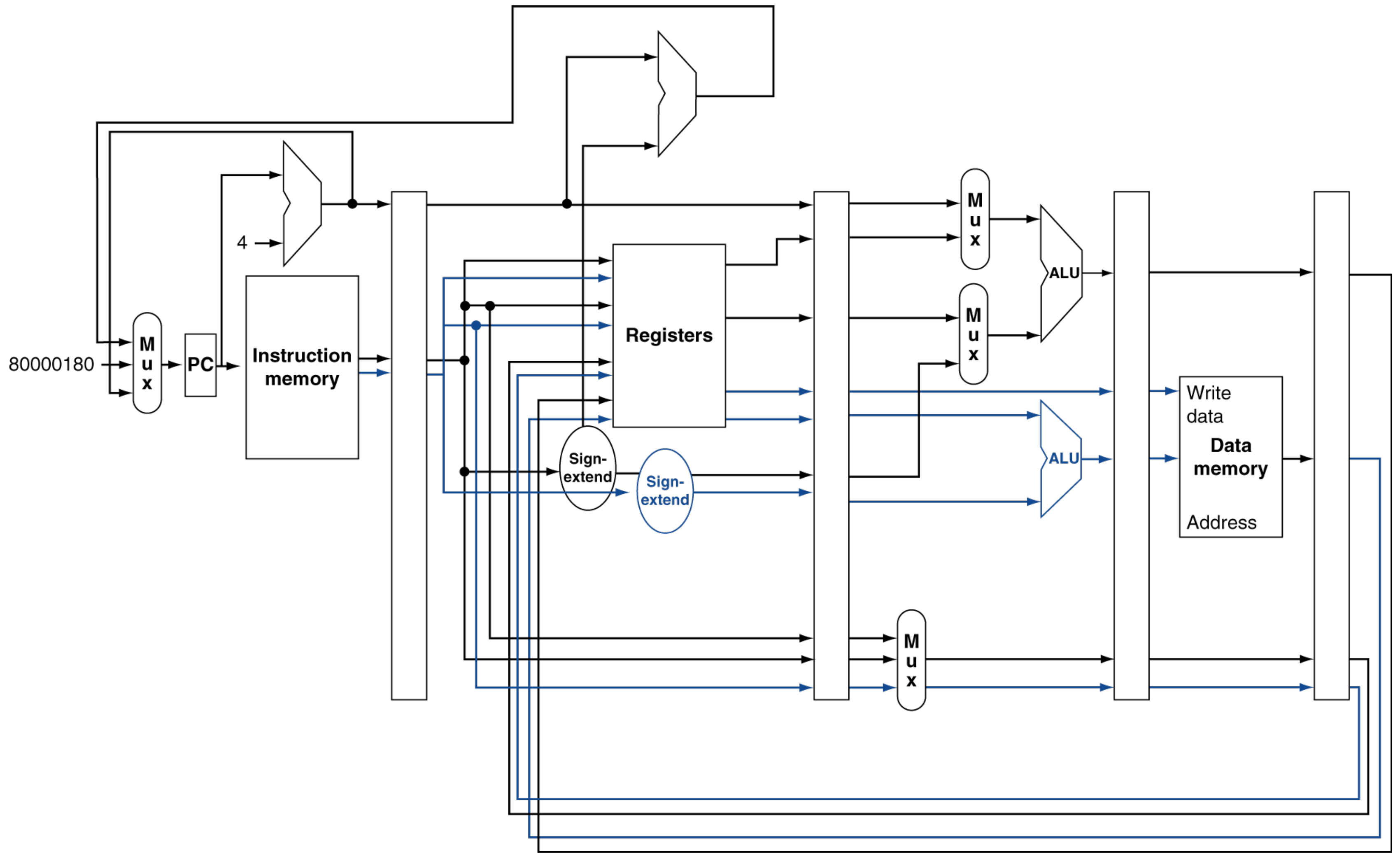
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Pad with nop if necessary

MIPS with Static Dual Issue



MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)        # $t0=array element
      addu $t0, $t0, $s2      # add scalar in $s2
      sw   $t0, 0($s1)        # store result
      addi $s1, $s1,-4        # decrement pointer
      bne  $s1, $zero, Loop   # branch $s1!=0
```

	ALU/branch	Load/store	cycle
`Loop:`	nop	`lw   $t0, 0($s1)`	1
	`addi $s1, $s1,-4`	nop	2
	`addu $t0, $t0, $s2`	nop	3
	`bne  $s1, $zero, Loop`	`sw   $t0, 4($s1)`	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

```
for (i=0;i<n;i++)
    a[i] += k;
```

$\longrightarrow$

```
for (i=0;i<n;i+=2){
    a[i] += k;
    a[i+1] += k;
}
```

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

Loop unrolling $\downarrow$

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      lw   $t0, -4($s1)
      addu $t0, $t0, $s2
      sw   $t0, -4($s1)
      lw   $t0, -8($s1)
      addu $t0, $t0, $s2
      sw   $t0, -8($s1)
      lw   $t0, -12($s1)
      addu $t0, $t0, $s2
      sw   $t0, -12($s1)
      addi $s1, $s1,-16
      bne  $s1, $zero, Loop
```

$\rightarrow$ Register Renaming

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      lw   $t1, -4($s1)
      addu $t1, $t1, $s2
      sw   $t1, -4($s1)
      lw   $t2, -8($s1)
      addu $t2, $t2, $s2
      sw   $t2, -8($s1)
      lw   $t3, -12($s1)
      addu $t3, $t3, $s2
      sw   $t3, -12($s1)
      addi $s1, $s1,-16
      bne  $s1, $zero, Loop
```

$\rightarrow$ Instruction Reordering

```
Loop: lw   $t0, 0($s1)
      addi $s1, $s1,-16
      addu $t0, $t0, $s2
      sw   $t0, 16($s1)
      lw   $t1, 12($s1)
      addu $t1, $t1, $s2
      sw   $t1, 12($s1)
      lw   $t2, 8($s1)
      addu $t2, $t2, $s2
      sw   $t2, 8($s1)
      lw   $t3, 4($s1)
      addu $t3, $t3, $s2
      sw   $t3, 4($s1)
      bne  $s1, $zero, Loop
```

Scheduling Loop Unrolled Code

```
Loop: lw   $t0, 0($s1)        # $t0=array element
      addu $t0, $t0, $s2      # add scalar in $s2
      sw   $t0, 0($s1)        # store result
      addi $s1, $s1,–4        # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
`Loop:`	`addi $s1, $s1,–16`	`lw $t0, 0($s1)`	1
	nop	`lw $t1, 12($s1)`	2
	`addu $t0, $t0, $s2`	`lw $t2, 8($s1)`	3
	`addu $t1, $t1, $s2`	`lw $t3, 4($s1)`	4
	`addu $t2, $t2, $s2`	`sw $t0, 16($s1)`	5
	`addu $t3, $t3, $s2`	`sw $t1, 12($s1)`	6
	nop	`sw $t2, 8($s1)`	7
	`bne $s1, $zero, Loop`	`sw $t3, 4($s1)`	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

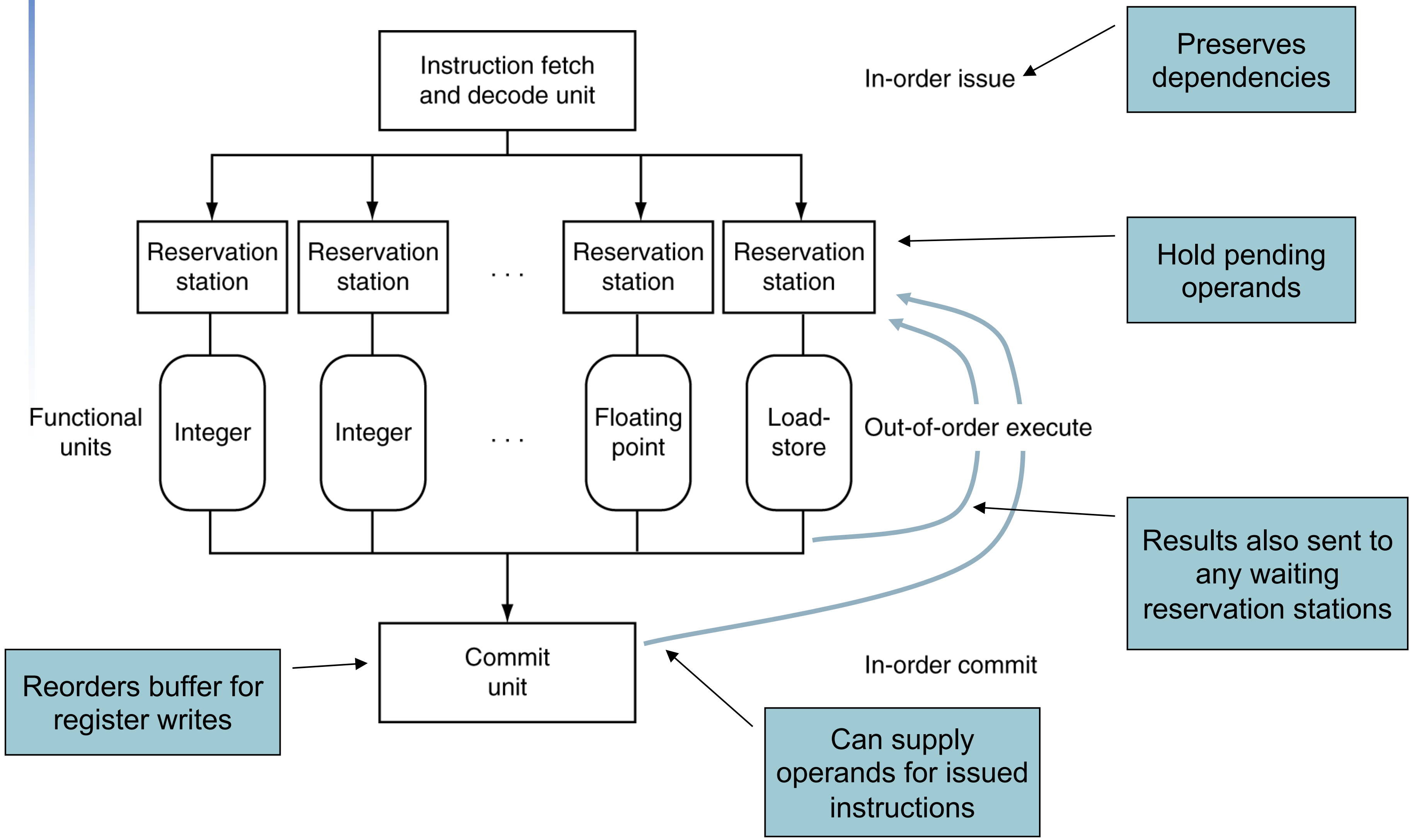
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
 lw     $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

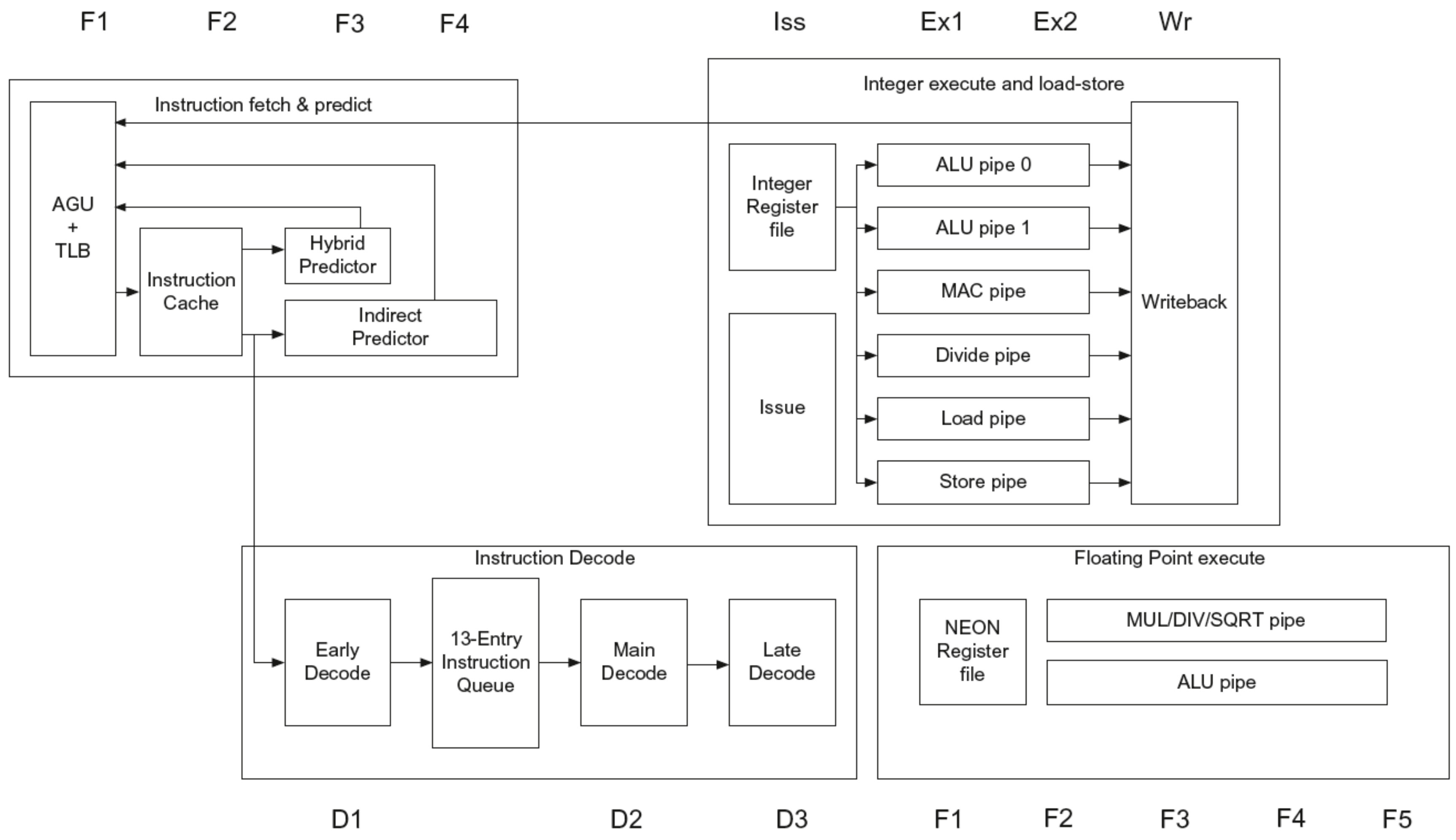
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	3000 MHz	14	4	Yes	2	75 W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87 W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130 W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130 W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140 W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165 W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185 W

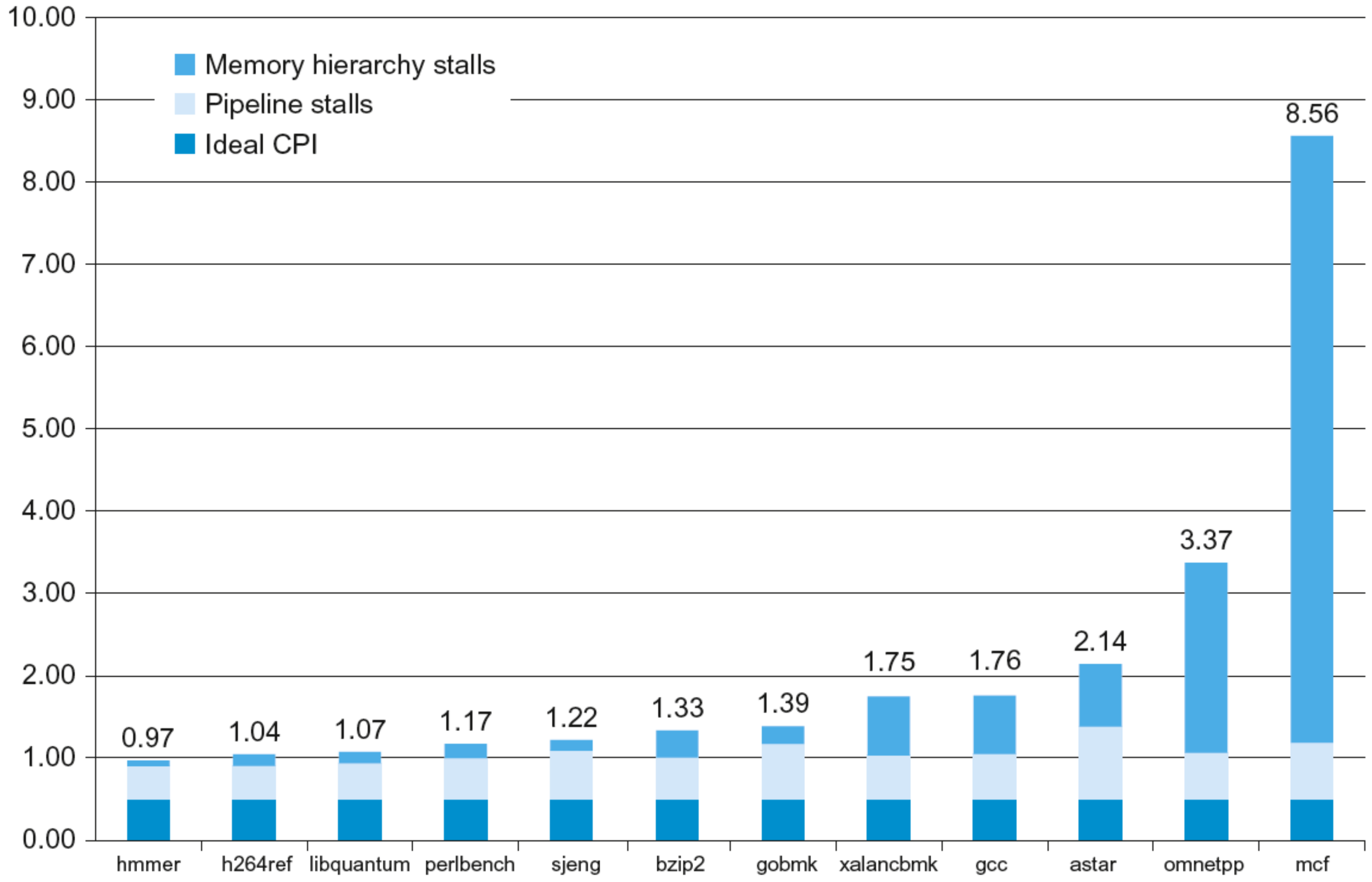
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

ARM Cortex-A53 Pipeline



ARM Cortex-A53 Performance



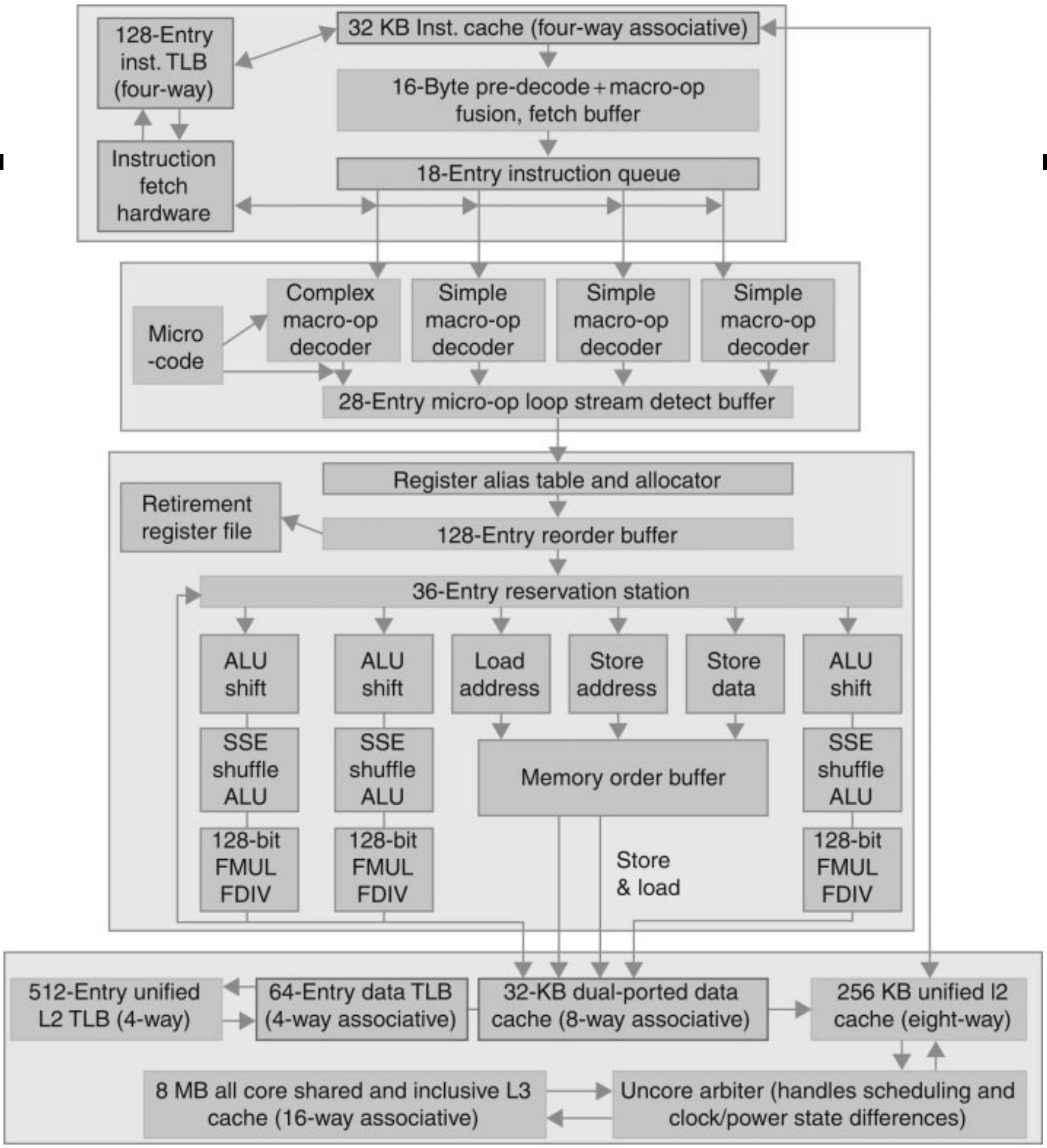

128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op 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)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

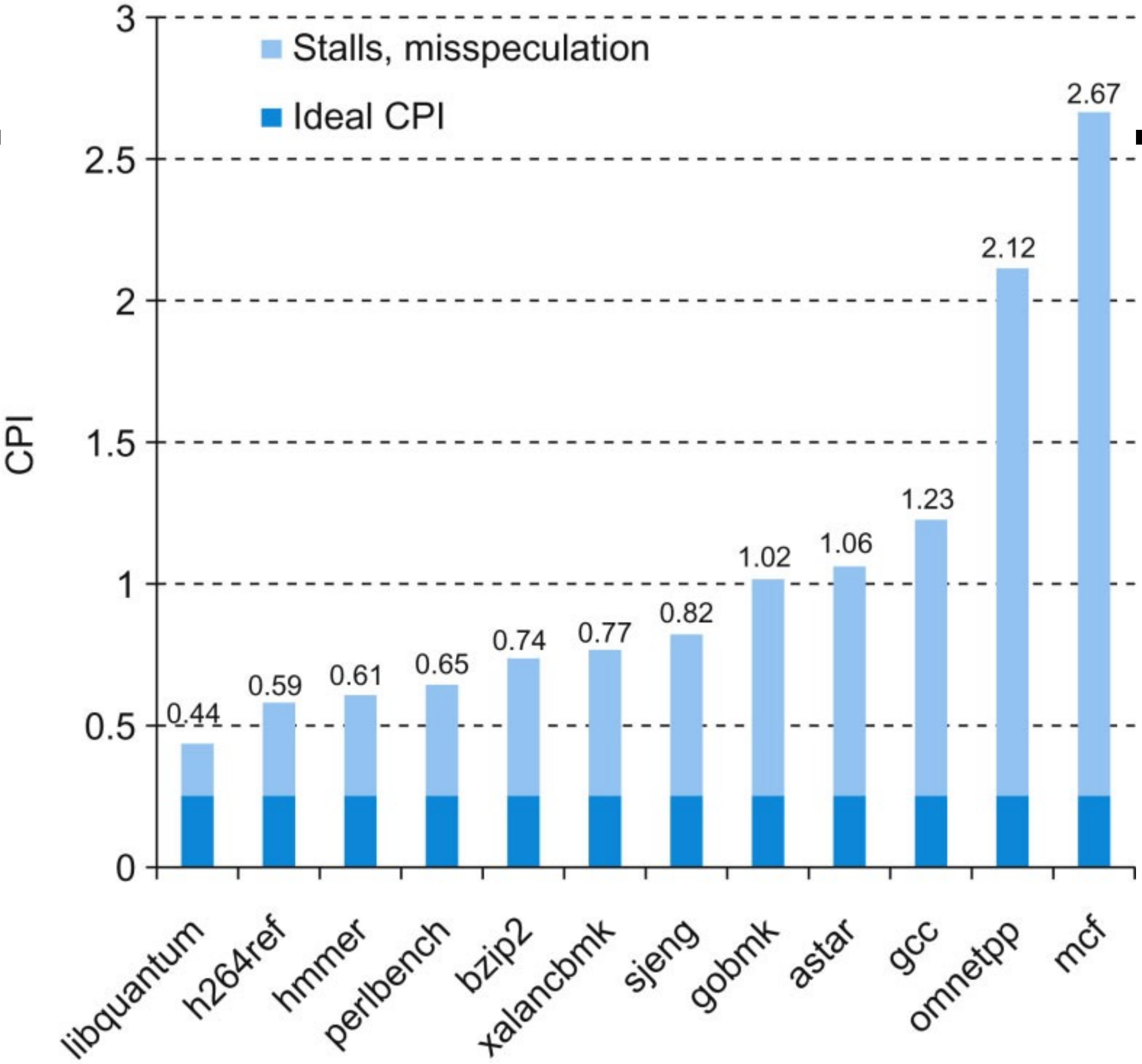


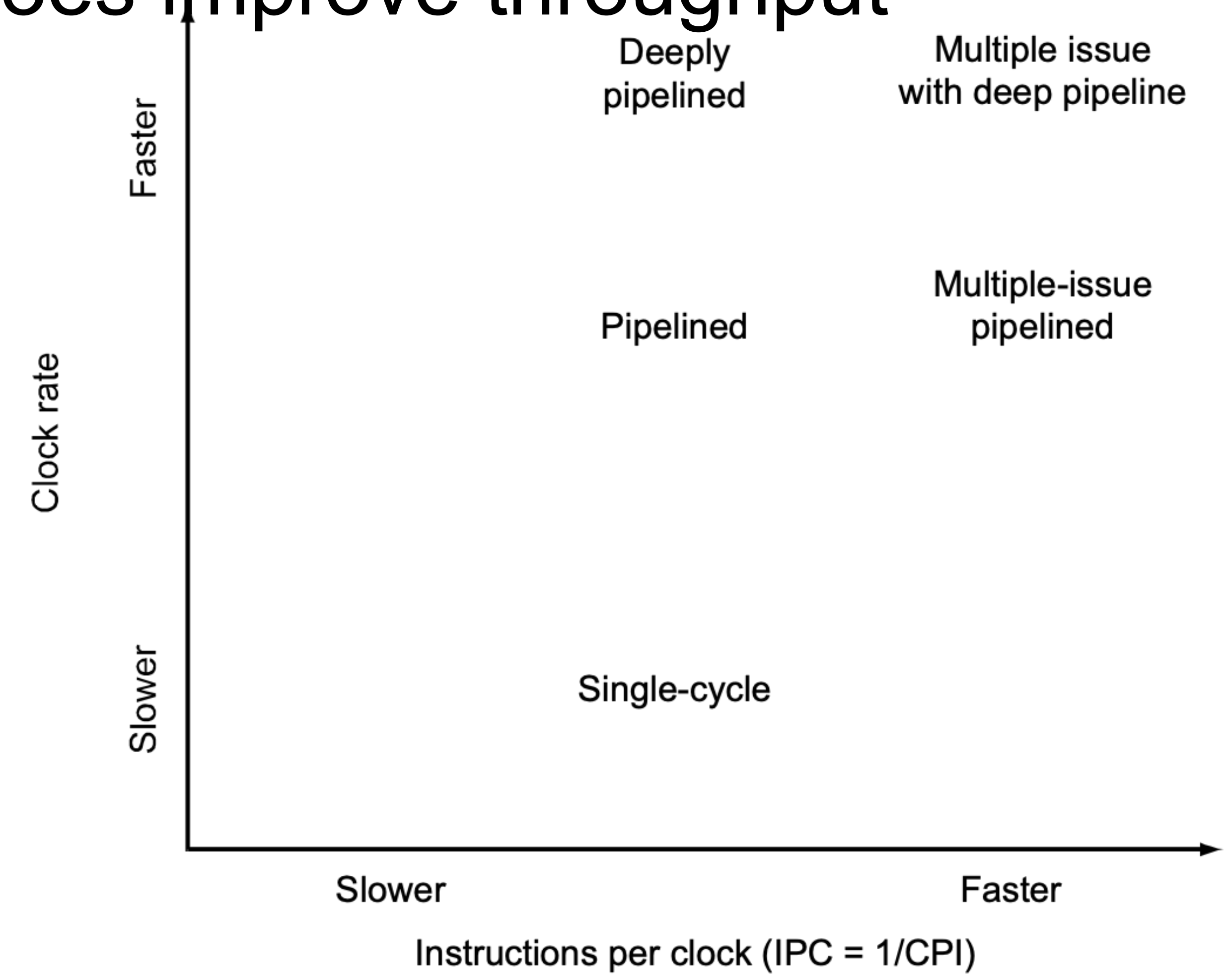
FIGURE 4.78 CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Summary

- Pipelining does not improve latency, but does improve throughput


Deeply pipelined
Multiple issue with deep pipeline
Pipelined
Multiple-issue pipelined
Single-cycle
Faster
Clock rate
Slower
Slower
Faster
Instructions per clock (IPC = 1/CPI)