

Jin-Soo Kim  
(jinsoo.kim@snu.ac.kr)

Systems Software &  
Architecture Lab.

Seoul National University

Fall 2019

# Advanced Processor Architecture

Chap. 4.10 – 11, 4.14 – 15



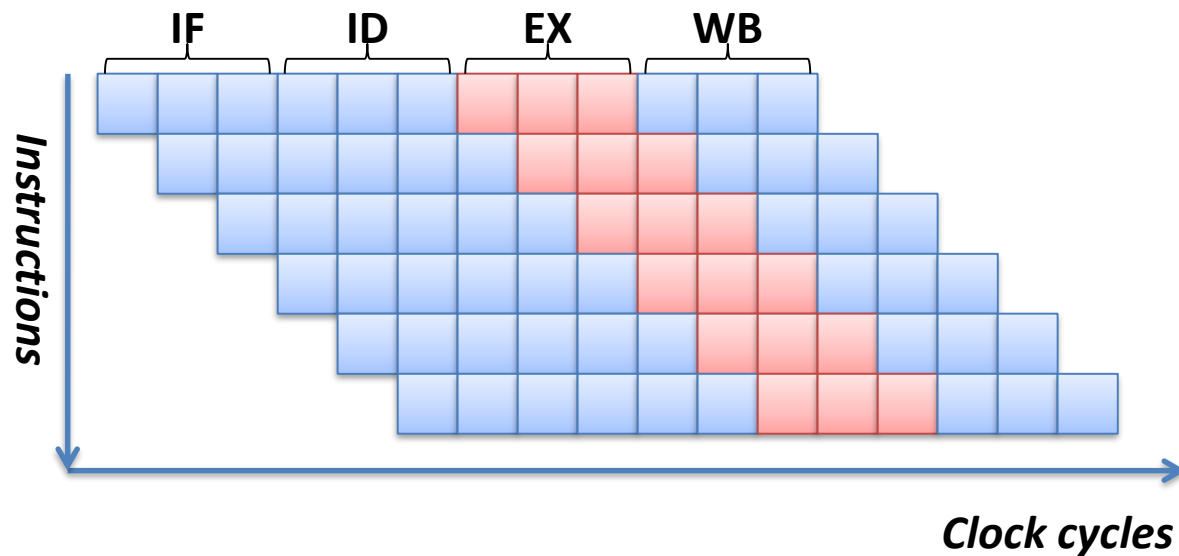
# Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- How to increase ILP?
- Deeper pipeline (“superpipelined”)
  - Less work per stage  $\Rightarrow$  shorter clock cycle
- Multiple issue
  - Replicate pipeline stages  $\Rightarrow$  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

# Superpipelined vs. Multiple-Issue

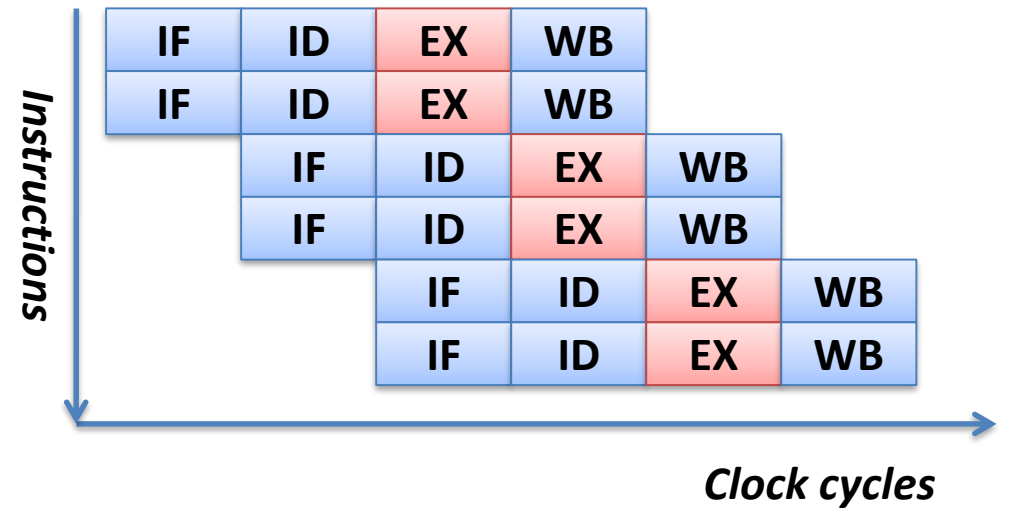
## ▪ Superpipelined

- Subdivide each pipeline stage
- Higher clock speed



## ▪ Multiple-issue

- Execute multiple instructions in parallel
- The EX stage has many functional units



# Multiple Issue

## ■ Static multiple issue

- Compiler groups instructions to be issued together
- Packages them into “issue slots”
- Compiler detects and avoids hazards
- **VLIW**(Very Long Instruction Word) processors

## ■ 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
- **Superscalar** processors

# 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

# Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

# Static Multiple Issue

- **Compiler groups instructions into “issue packets”**
  - Group of instructions that can be issued on a single cycle
  - Usually restricts what mix of instructions can be initiated in a clock cycle
  - Determined by pipeline resources required
  
- **Think of an issue packet as a very long instruction**
  - Specifies multiple concurrent operations
  - Very Long Instruction Word (VLIW)



# Scheduling Static Multiple Issue

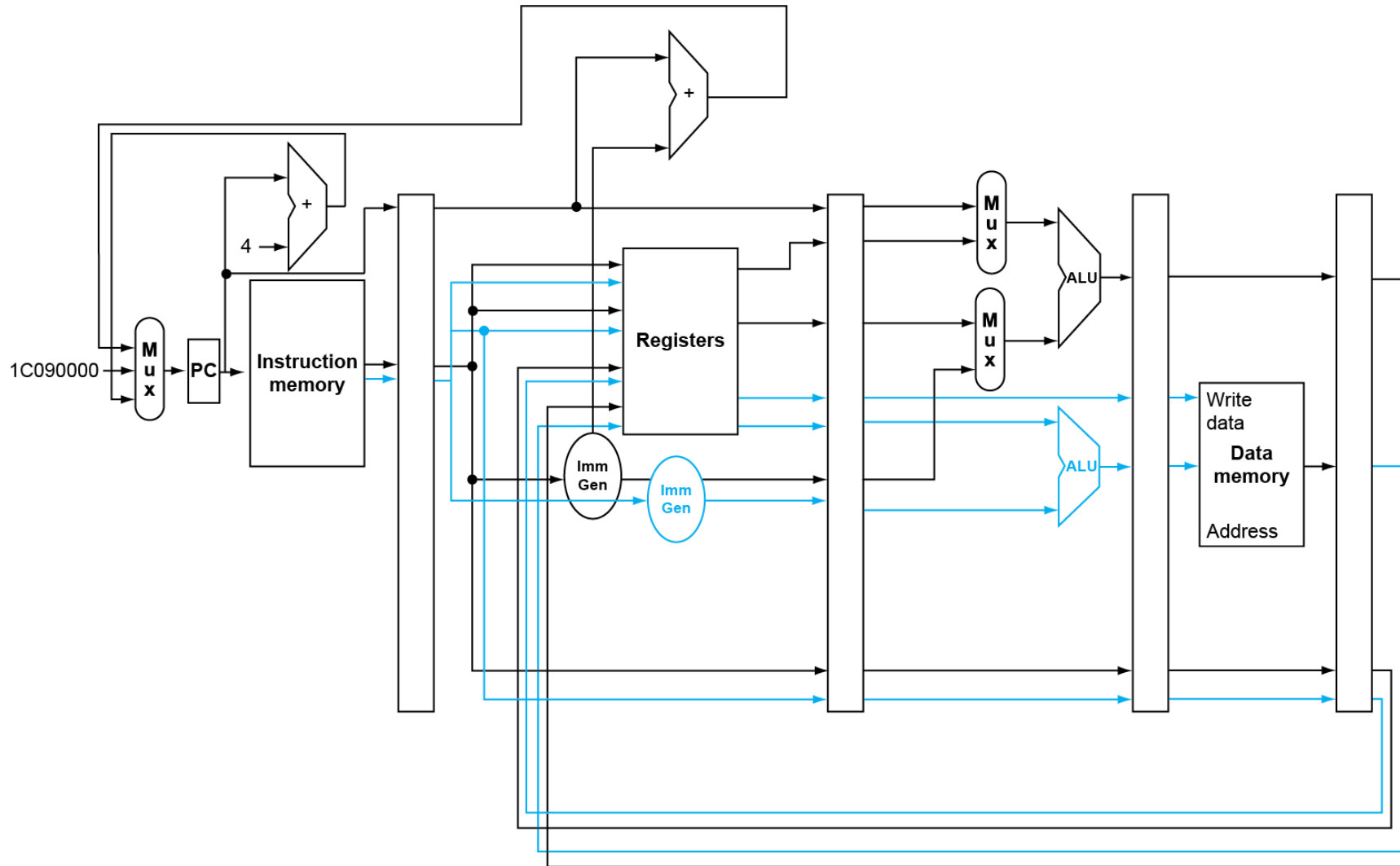
- Compiler must remove some/all hazards
- Reorder instructions into issue packets
- No dependencies within a packet
- Possibly some dependencies between packets
  - Varies between ISAs; compiler must know!
  - If all hazards are not removed, the hardware should detect hazards and generate stalls between two issue packets
- Pad with nop if necessary

# RISC-V with Static Dual Issue

- Two-issue packets
  - 64-bit aligned: One ALU/branch instruction + One load/store instruction
  - Pad an unused instruction with nop
  - Additional hardware:
    - +2 read / +1 write ports in register file
    - Separate adder for computing the effective address for memory

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

# RISC-V with Static Dual Issue



# Hazards in the Dual-Issue RISC-V

- 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
  - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

```
add    x10, x0, x13
ld     x2, 0(x10)
```

# Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld    x31, 0(x20)    // x31 = array element
      add   x31, x31, x21  // add scalar in x21
      sd    x31, 0(x20)    // store result
      addi  x20, x20, -8   // decrement pointer
      blt   x22, x20, Loop // branch if x22 < x20
```

	ALU/branch	Load/store	Cycle
Loop:	nop	ld x31, 0(x20)	1
	addi x20, x20, -8	nop	2
	add x31, x31, x21	nop	3
	blt x22, x20, Loop	sd x31, 0(x20)	4

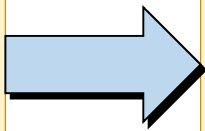
- $IPC = 5/4 = 1.25$  (cf. peak  $IPC = 2$ )

# Loop Unrolling

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

# Loop Unrolling Example

```
Loop: ld    x31, 0(x20)
      add  x31, x31, x21
      sd   x31, 0(x20)
      addi x20, x20, -8
      blt  x22, x20, Loop
```



```
Loop: ld    x31, 0(x20)
      add  x31, x31, x21
      sd   x31, 0(x20)

      ld   x31, -8(x20)
      add  x31, x31, x21
      sd   x31, -8(x20)

      ld   x31, -16(x20)
      add  x31, x31, x21
      sd   x31, -16(x20)

      ld   x31, -24(x20)
      add  x31, x31, x21
      sd   x31, -24(x20)

      addi x20, x20, -32
      blt  x22, x20, Loop
```

# Loop Unrolling Scheduled Example

	ALU/branch	Load/store	Cycle
Loop:	addi x20, x20, -32	ld x28, 0(x20)	1
	nop	ld x29, 24(x20)	2
	add x28, x28, x21	ld x30, 16(x20)	3
	add x29, x29, x21	ld x31, 8(x20)	4
	add x30, x30, x21	sd x28, 32(x20)	5
	add x31, x31, x21	sd x29, 24(x20)	6
	nop	sd x30, 16(x20)	7
	blt x22, x20, Loop	sd x31, 8(x20)	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
  - Through it may still help
  - Code semantics ensured by the CPU
- In-order vs. out-of-order (OOO)
  - Out-of-order processor analyzes the data flow structure of a program, and then executes instructions in some order that preserves the data flow order (Instruction execution order  $\neq$  program order)

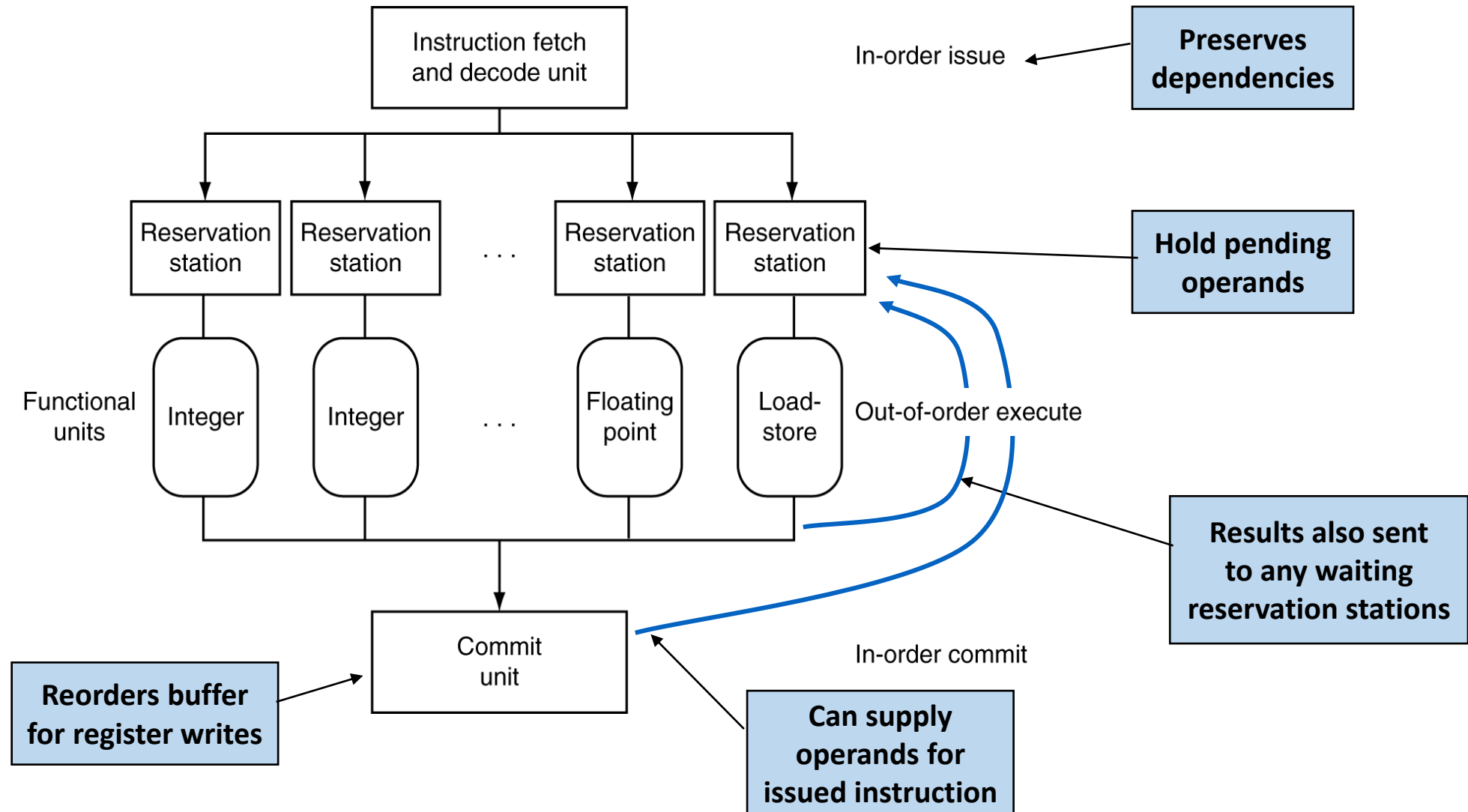
# Dynamic Pipeline Scheduling

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

```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x23, x23, x3
andi  x5, x23, x20
```

- Can start sub while add is waiting for ld

# Dynamically Scheduled CPU



# Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a functional unit
    - Register update may not be required

# Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
  
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don't commit load until speculation cleared

# Why Do Dynamic Scheduling?

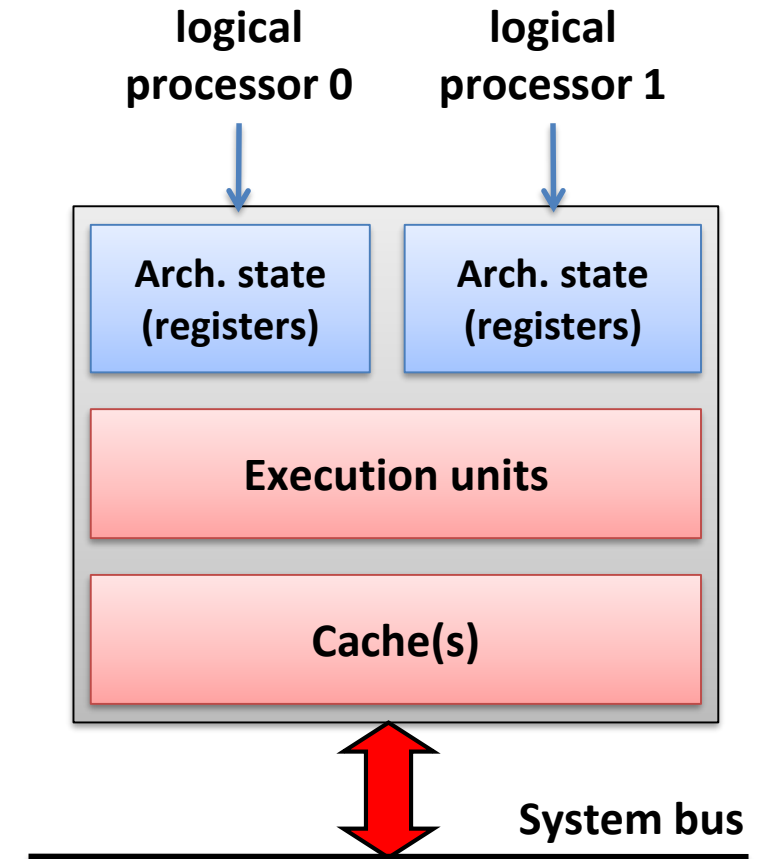
- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

# 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

# Hyper-Threading

- Simultaneous Multithreading Technology (SMT)
  - Utilizes thread-level parallelism
  - Fill pipelines with the instructions from
  - multiple threads running at the same time
  - An SMT processor appears as if it were multiple independent processors
  - Uses processor resources more effectively
  - Cost: <5% in added die area





# Power Efficiency

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

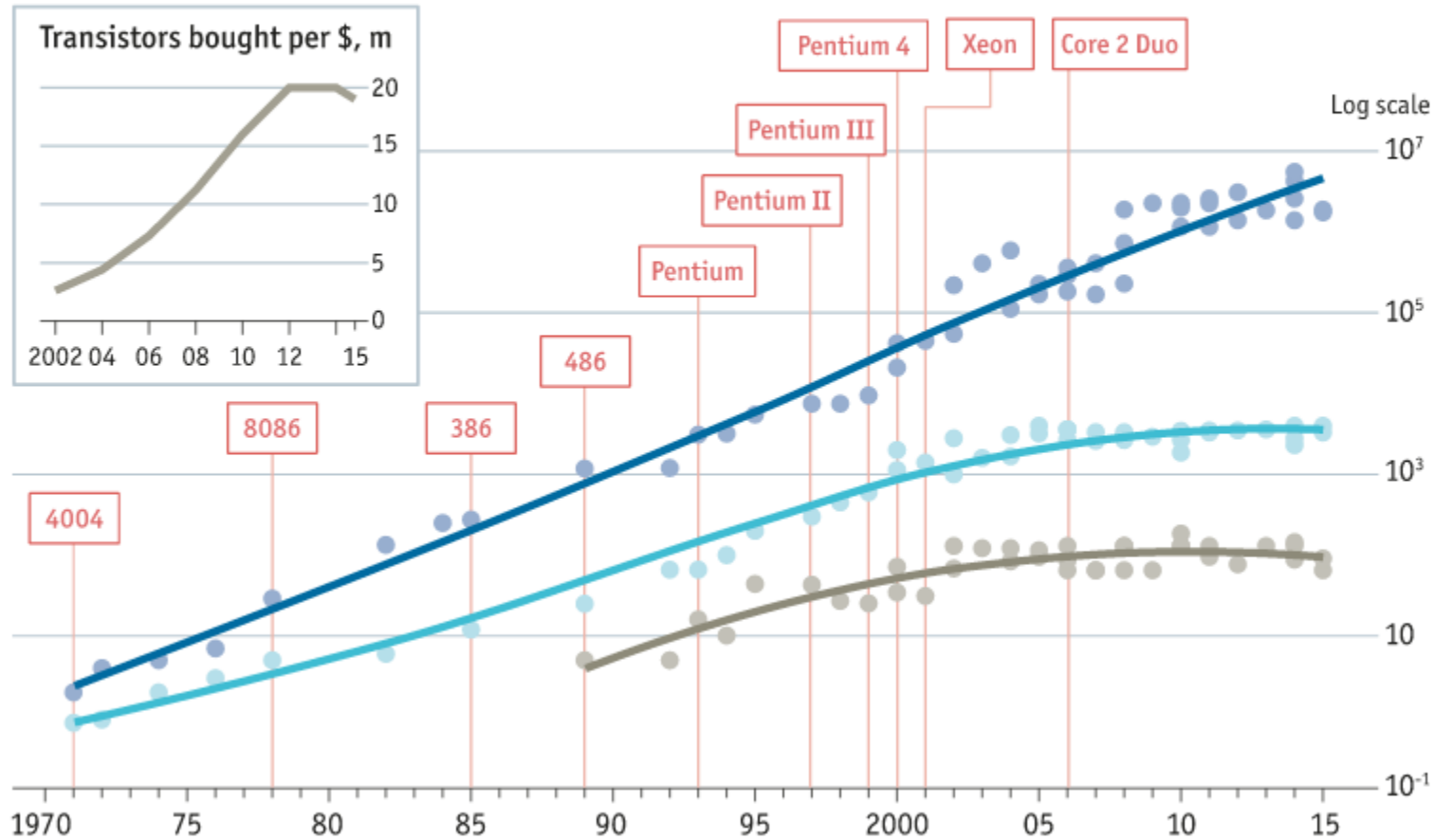
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-order/Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Core i5 Nehalem	2010	3300MHz	14	4	Yes	2-4	87W
Core i5 Ivy Bridge	2012	3400MHz	14	4	Yes	8	77W

# CPU Trends

## Stuttering

● Transistors per chip, '000 ● Clock speed (max), MHz ● Thermal design power\*, w

Chip introduction dates, selected



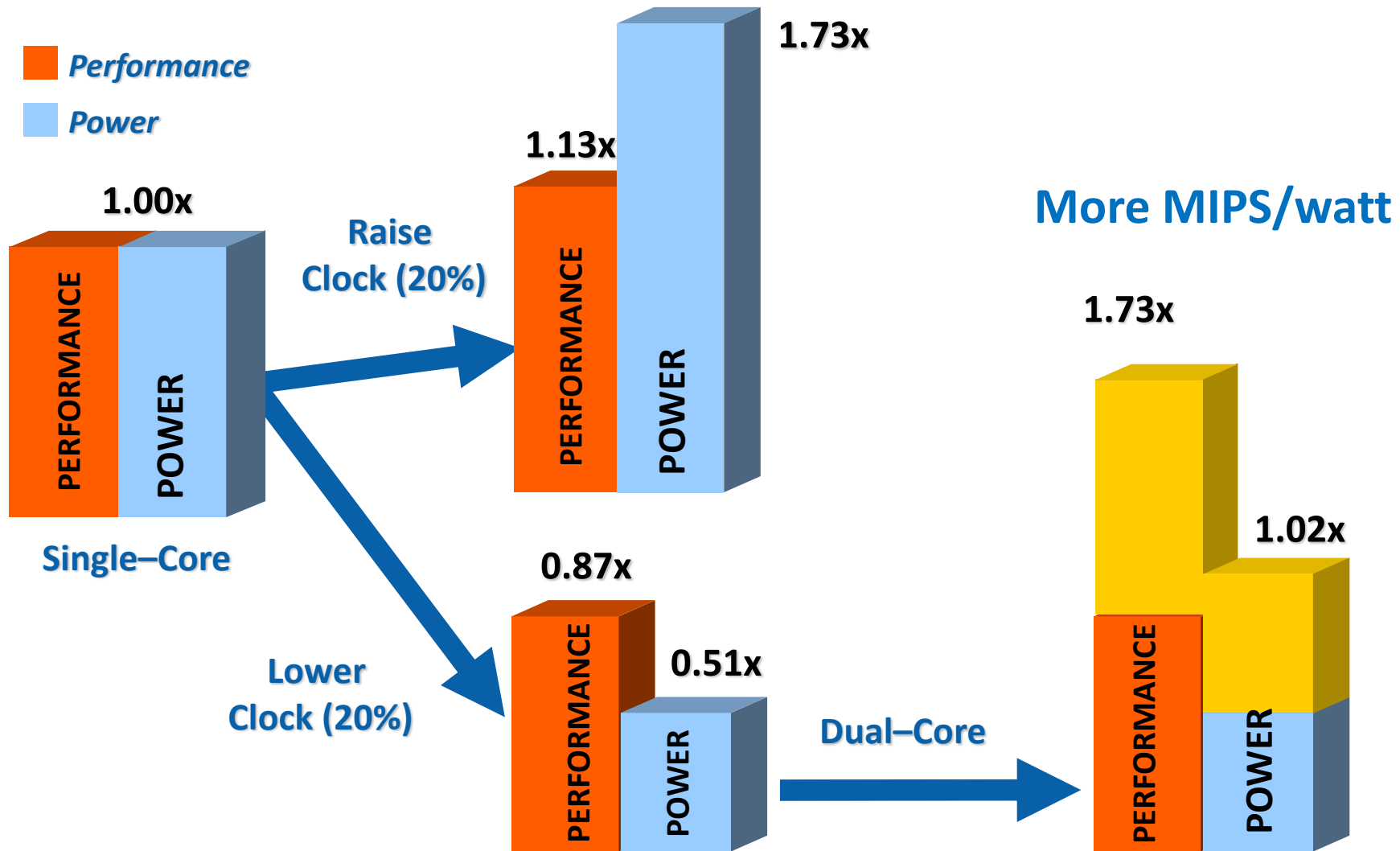
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; *The Economist*

\*Maximum safe power consumption

# Why Multi-core?

- **Memory wall**
  - CPU 55%/year, Memory 10%/year (1986 – 2000)
  - Caches show diminishing returns
- **ILP (Instruction Level Parallelism) wall**
  - Control dependency
  - Data dependency
- **Power wall**
  - Dynamic power  $\propto$  Frequency<sup>3</sup>
  - Static power  $\propto$  Frequency
  - Total power  $\propto$  The number of cores

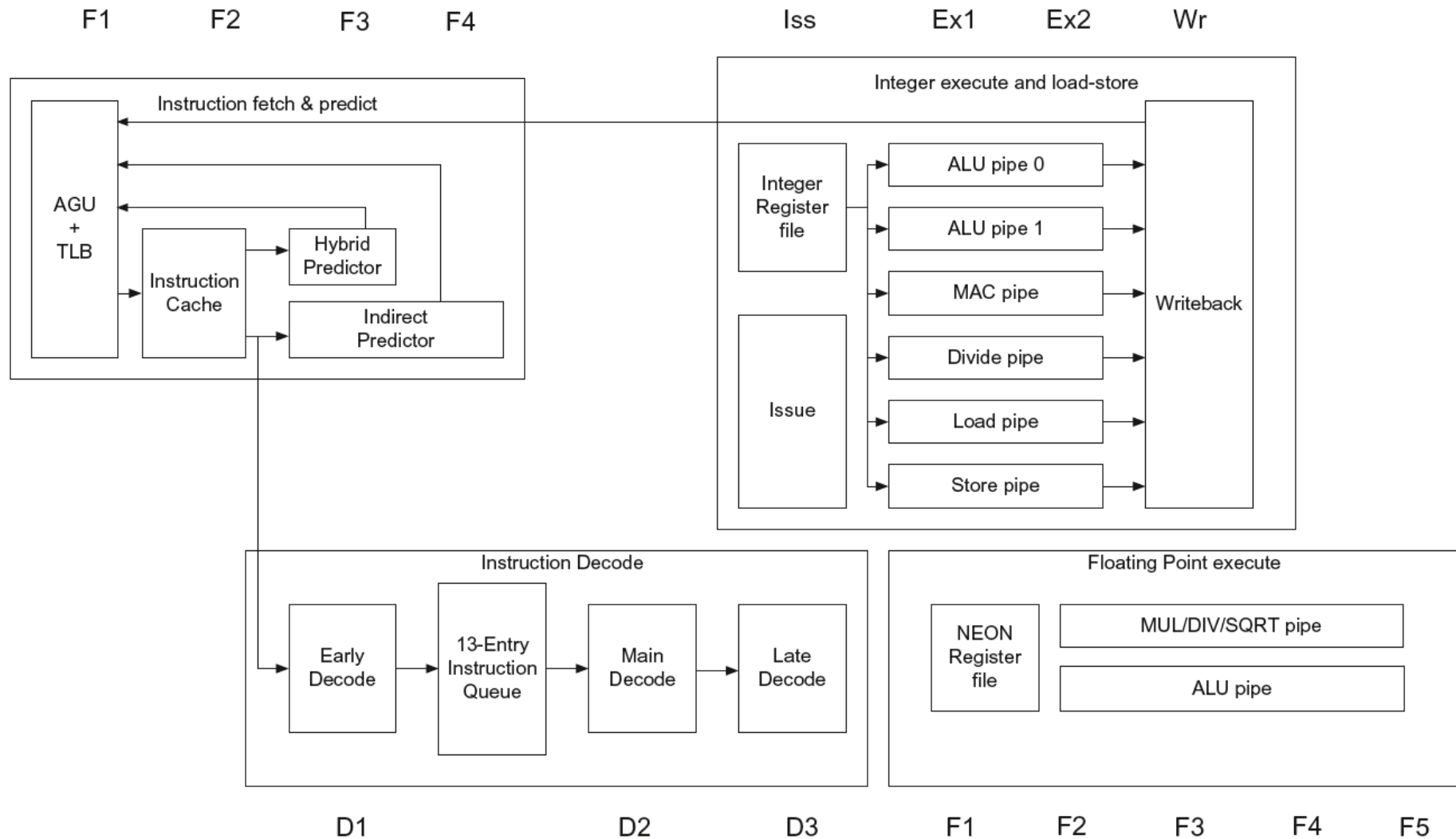
# Single-core vs. Multi-core



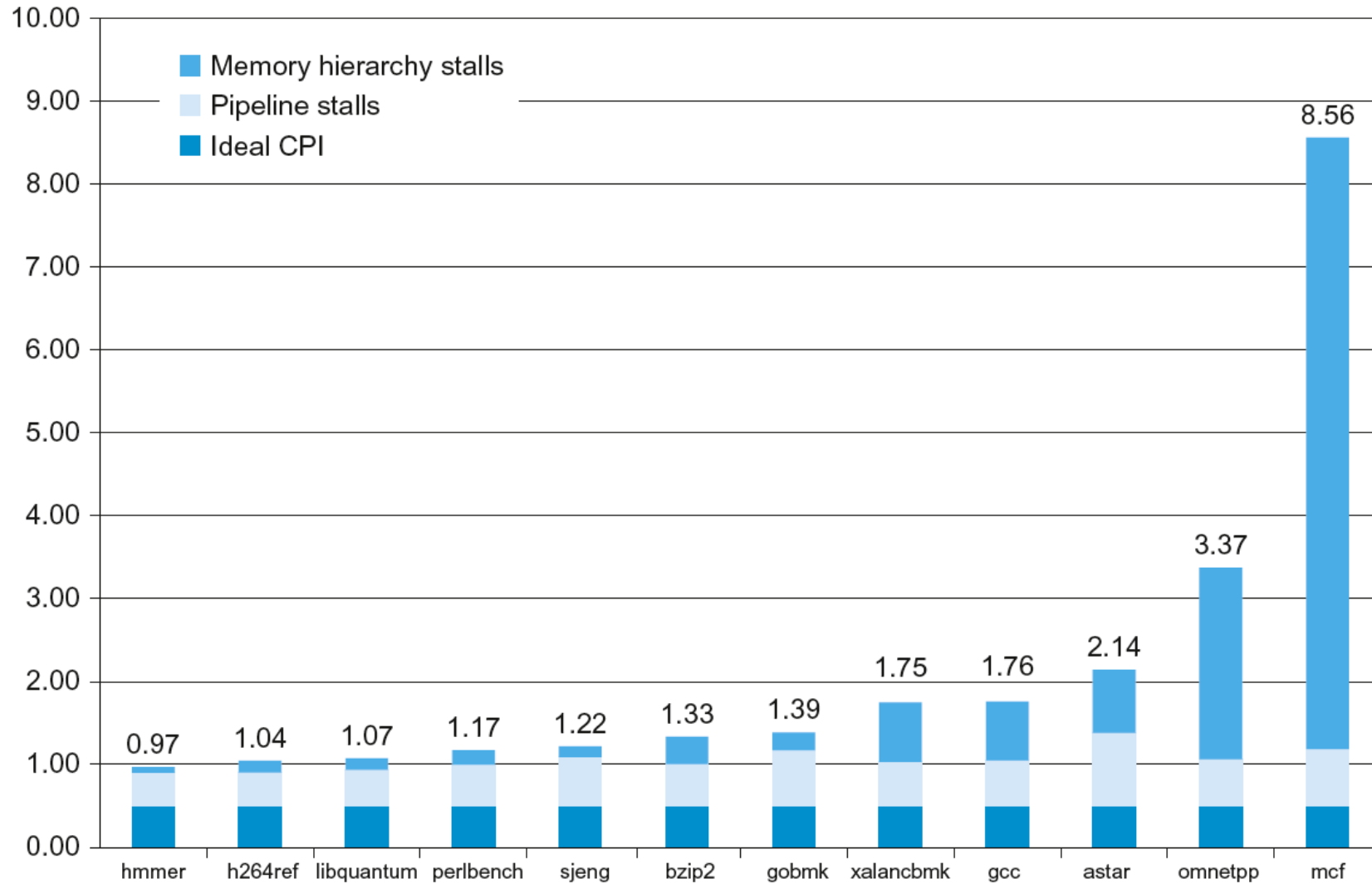
# Cortex A53 vs. Intel i7

Processor	ARM Cortex A53	Intel Core i7 920
Market	Personal mobile device	Server, cloud
Thermal design power (TDP)	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	16-64KiB I\$, 16-64 KiB D\$	32KiB 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

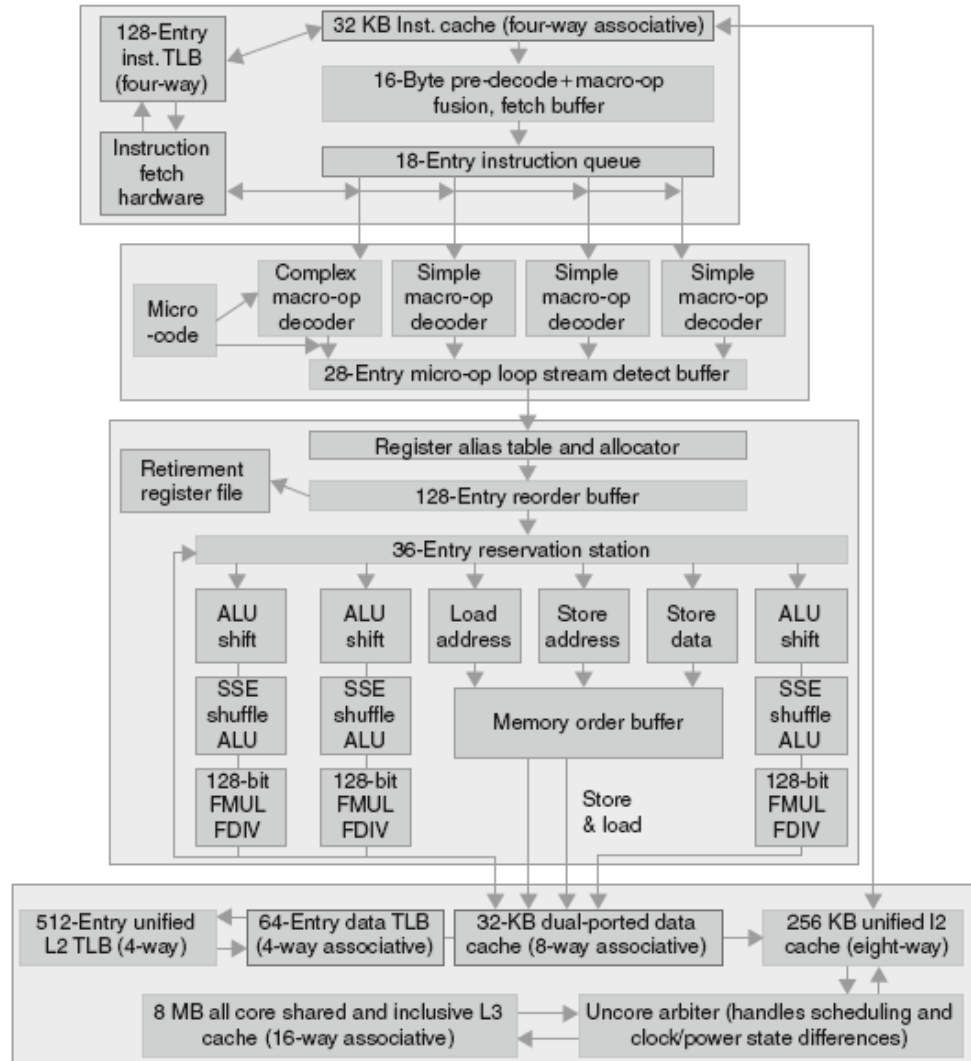


# ARM Cortex-A53 Performance

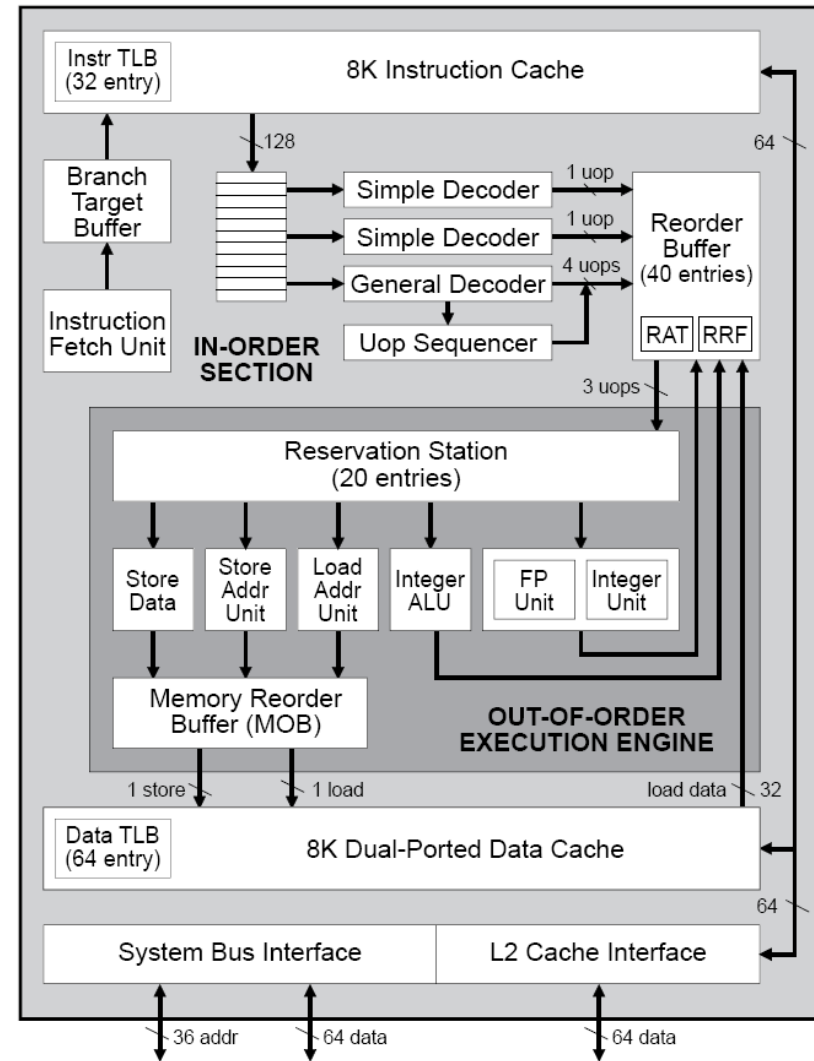


# Core i7 Pipeline

Intel Nehalem (2008)

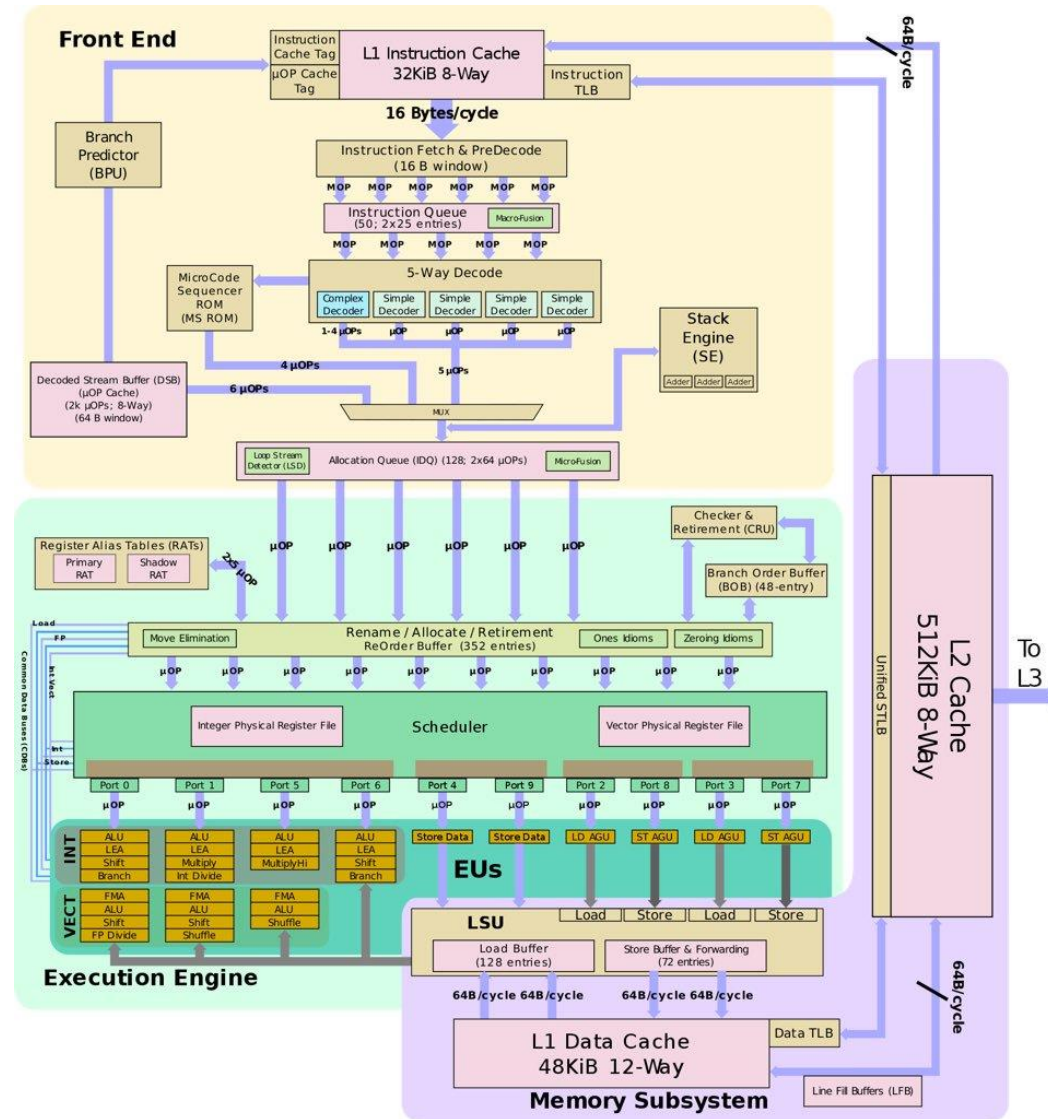


Intel P6 (1995)

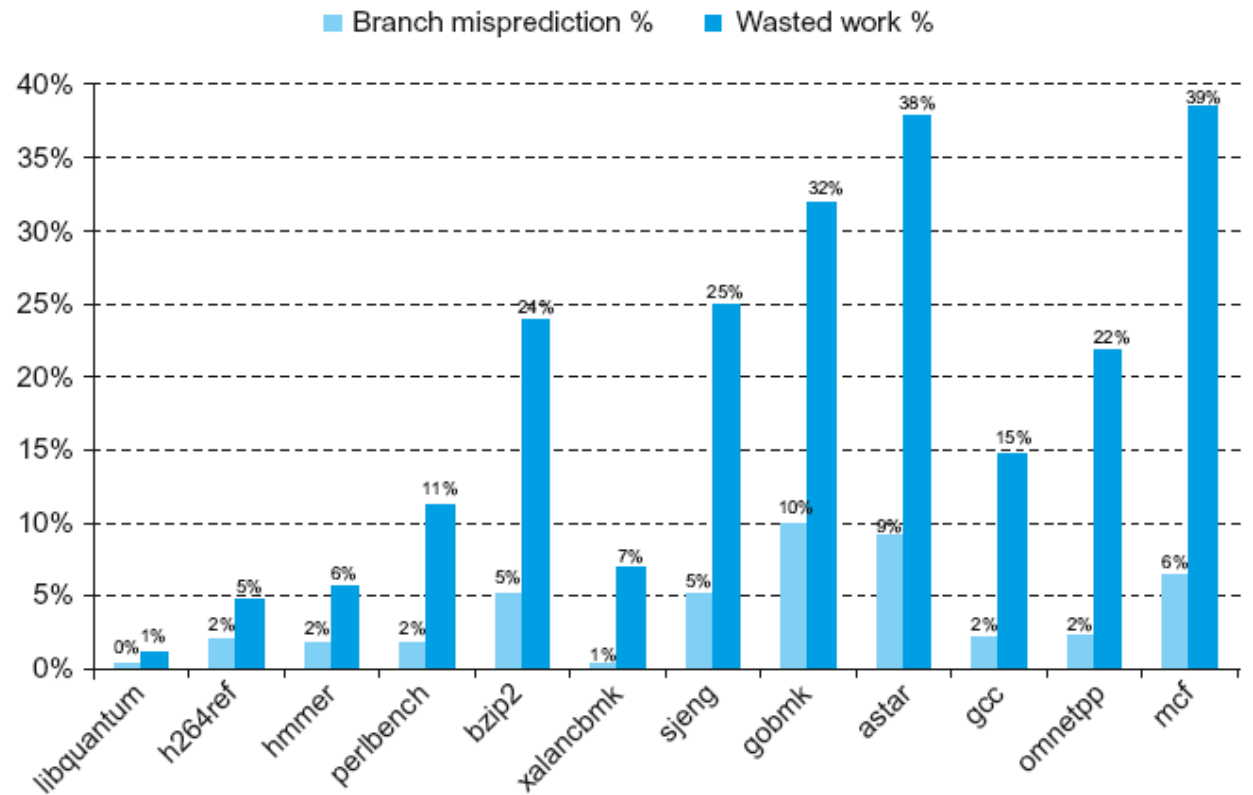
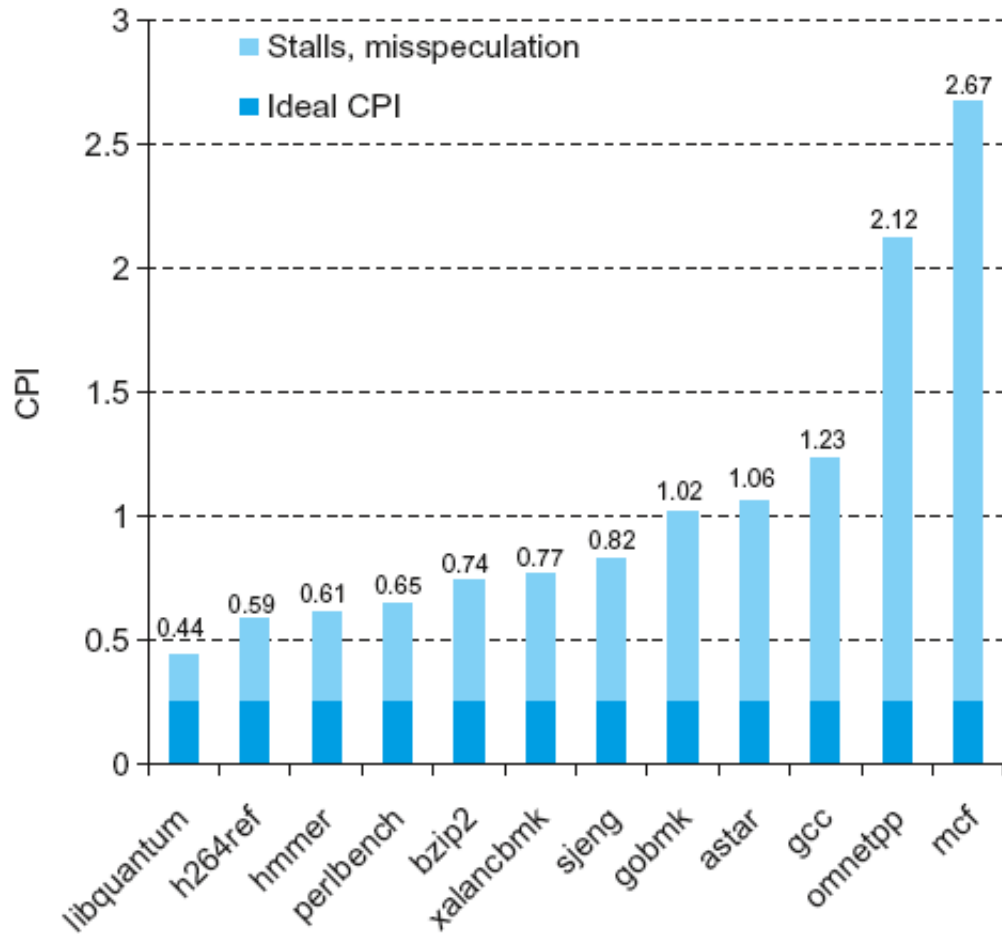




# Intel Sunny Cove (2019)



# Core i7 Performance



# Fallacies

- **Pipelining is easy**
  - The basic idea is easy
  - The devil is in the details (e.g., detecting data hazards)
- **Pipelining is independent of technology**
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends (e.g., predicated instructions)

# Pitfalls

- Poor ISA design can make pipelining harder
- e.g., complex instruction sets (VAX, IA-32)
  - Significant overhead to make pipelining work
  - IA-32 micro-op approach
- e.g., complex addressing modes
  - Register update side effects, memory indirection
- e.g., delayed branches
  - Advanced pipelines have long delay slots

# Summary

- ISA influences design of datapath and control
- Datapath and control influences 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