CENG 3420
Computer Organization & Design
Lecture 15: Instruction-Level Parallelism

Textbook: Chapter 4.11

Zhengrong Wang

CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

Introduction

Extracting Yet More Performance

Deep pipelining:

- Increase the depth of the pipeline to increase the clock rate
  - Balance the latency of each stage.
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)

Multiple-Issue:

- Fetch (and execute) more than one instructions at one time
  - Expand every pipeline stage to accommodate multiple instructions

Example on Multiple-Issue

- The instruction execution rate, CPI, will be less than 1.
- Instead we use IPC: instructions per clock cycle
  - E.g., a 3 GHz, four-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4

Question: If the datapath has a five-stage pipeline, how many instructions are active in the pipeline at any given time?

ILP & Machine Parallelism

Instruction-level parallelism (ILP)

- A measure of the average number of instructions in a program that a processor might be able to execute at the same time
- Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions

Machine Parallelism

- A measure of the ability of the processor to take advantage of the ILP of the program
- Determined by the number of instructions that can be fetched and executed at the same time.

To achieve high performance, need both ILP and Machine Parallelism

Multiple-Issue Processor Styles

Static multiple-issue processors (aka VLIW)

- Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
- Example: Intel Itanium and Itanium 2 for the IA-64 ISA
  - EPIC (Explicit Parallel Instruction Computer)
  - 128-bit "bundles" containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
  - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
  - Extensive support for speculation and predication

Dynamic multiple-issue processors (aka superscalar)

- Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
- Example: IBM Power series, Pentium 4, MIPS R10K, AMD Barcelona

Static vs. Dynamic

Static: “let’s make our compiler take care of this”

- Fast runtime
- Limited performance (variable values available when is running)

Dynamic: “let’s build some hardware that takes care of this”

- Hardware penalty
- Complete knowledge on the program

Dependencies

Dependencies

- Structural Hazards – Resource conflicts
  - A SS/VLIW processor has a much larger number of potential resource conflicts
  - Functional units may have to arbitrate for result buses and register-file write ports
  - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource
- Data Hazards – Storage (data) dependencies
  - Limitation more severe in a SS/VLIW processor
- Control Hazards – Procedural dependencies
  - Ditto, but even more severe
  - Use dynamic branch prediction to help resolve the ILP issue

Resolved through combination of hardware and software.

Data Hazards

- True dependency (RAW)
  - Later instruction using a value (not yet) produced by an earlier instruction.
- Anti-dependencies (WAR)
  - Later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later).
- Output dependency (WAW)
  - Two instructions write the same register or memory location.

```
R3 := R3 * 5   # R3 -> R3 True dependency (RAW)
R4 := R3 + 1   # R3 -> R3 Anti-dependency (WAR)
R3 := R5 + 1   # R3 -> R3 Output dependence (WAW)
```

Question

- Question: Find all data dependences in this instruction sequence.

```
I1: ADD R1, R2, R1
I2: LW R2, 0(R1)
I3: LW R1, 4(R1)
I4: OR R3, R1, R2
```

Data Hazards

```
R3 := R3 * 5  # R3 -> R3 True dependency (RAW)
R4 := R3 + 1  # R3 -> R3 Anti-dependency (WAR)
R3 := R5 + 1  # R3 -> R3 Output dependence (WAW)
```

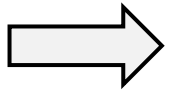
- True dependencies (RAW) represent the flow of data and information through a program
- Anti-dependencies (WAR) and output dependencies (WAW) arise because the limited number of registers, i.e., programmers reuse registers for different computations leading to storage conflicts
- Storage conflicts can be reduced (or eliminated) by
  - Increasing or duplicating the troublesome resource
  - Providing additional registers that are used to re-establish the correspondence between registers and values
  - Allocated dynamically by the hardware in SS processors

Resolve Storage Conflicts

- Register Renaming
  - The processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * 5   # R3 -> R3
R4 := R3 + 1   # R3 -> R3
R3 := R5 + 1   # R3 -> R3
```

Rename



```
R3b := R3a * 5   # R3b -> R3b
R4a := R3b + 1   #
R3c := R5a + 1   #
```

  - The hardware that does renaming assigns a "replacement" register from a pool of free registers
  - Releases it back to the pool when its value is superseded and there are no outstanding references to it
  - Rename from "architectural register" to "physical register" – like virtual memory?

Resolve Control Dependency

Speculation

- Allow execution of future instr's that (may) depend on the speculated instruction:
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (memory order speculation)

- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
  - Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

VLIW

Static Multiple Issue Machines (VLIW)

Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously

- Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
- The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
- The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards

- VLIW has
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

An Example VLIW RISC-V Machine

- The ALU and data transfer instructions are issued at the same time.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 4.69 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

An Example VLIW RISC-V Machine

- Consider a 2-issue RISC-V with a 2 instr bundle (8B)

ALU or Branch Inst (4B)				Load or Store Inst (4B)			

- Instructions are always fetched, decoded, and issued in pairs
- If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

A RISC-V VLIW (2-issue) Datapath

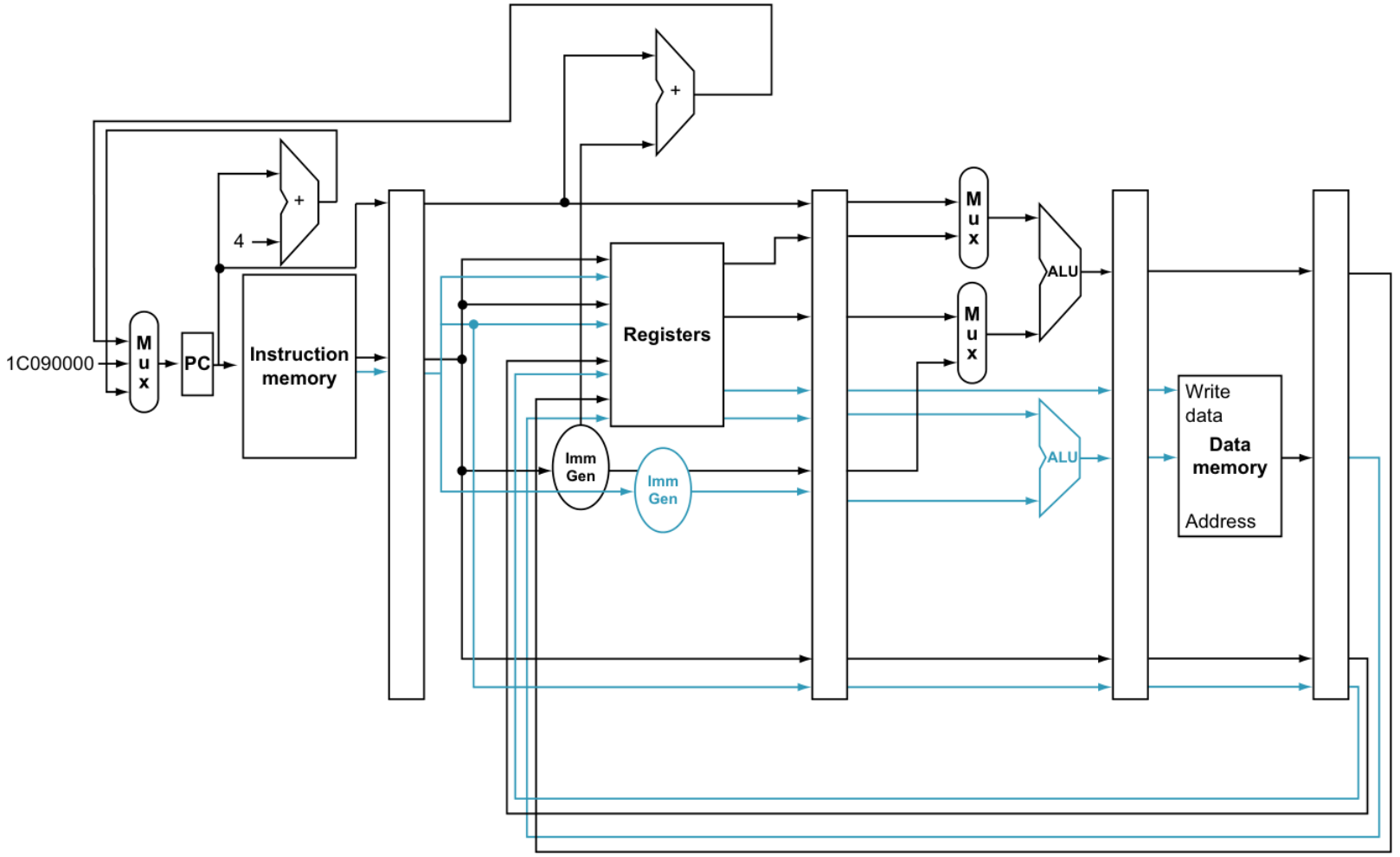


FIGURE 4.70 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

Compiler Techniques for Exposing ILP

- Instruction Scheduling
- Loop Unrolling

Instruction Scheduling

- Consider the following loop code

```
Loop: lw   x31, 0(x20)     // x31=array element
      add  x31, x31, x21   // add scalar in x21
      sw   x31, 0(x20)     // store result
      addi x20, x20, -4    // decrement pointer
      blt  x22, x20, Loop  // compare to loop limit,
                           // branch if x20 > x22
```

- Reorder to avoid pipeline stalls as much as possible
  - Instructions in one bundle must be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

Instruction Scheduled

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw x31, 0(x20)	1
	addi x20, x20, -4		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sw x31, 4(x20)	4

FIGURE 4.71 The scheduled code as it would look on a two-issue RISC-V pipeline. The empty slots are no-ops. Note that since we moved the `addi` before the `sw`, we had to adjust `sw`'s offset by 4.

Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
  - Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards
- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies

Loop Unrolling Scheduled

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20, x20, ~~-32~~ -16	lw x28, 0(x20)	1
		lw x29, 12(x20)	2
	add x28, x28, x21	lw x30, 8(x20)	3
	add x29, x29, x21	lw x31, 4(x20)	4
	add x30, x30, x21	sw x28, 16(x20)	5
	add x31, x31, x21	sw x29, 12(x20)	6
		sw x30, 8(x20)	7
	blt x22, x20, Loop	sw x31, 4(x20)	8

FIGURE 4.72 The unrolled and scheduled code of Figure 4.71 as it would look on a static two-issue RISC-V pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements x20 by 16, the addresses loaded are the original value of x20, then that address minus 4, minus 8, and minus 12.

Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle – Done by code re-ordering (trace scheduling)
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts

Superscalar

Dynamic Multiple-Issue Processors

Use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously

- Instruction-fetch and issue – fetch instructions, decode them, and issue them to a FU to await execution
- Defines the instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated
- Defines the processor lookahead capability – complete execution of issued instructions beyond the current instruction
- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state)

In-Order Superscalar

Instruction Fetch and Decode Units

- are required to issue instructions in-order so that dependencies can be tracked

Commit Unit

- is required to write results to registers and memory in program fetch order so that
  - If exceptions occur the only registers updated will be those written by instructions before the one causing the exception
  - If branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)

Out-of-Order Superscalar

- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order
- FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
- Allowing out-of-order execution increases the amount of ILP

In-Order vs Out-of-Order

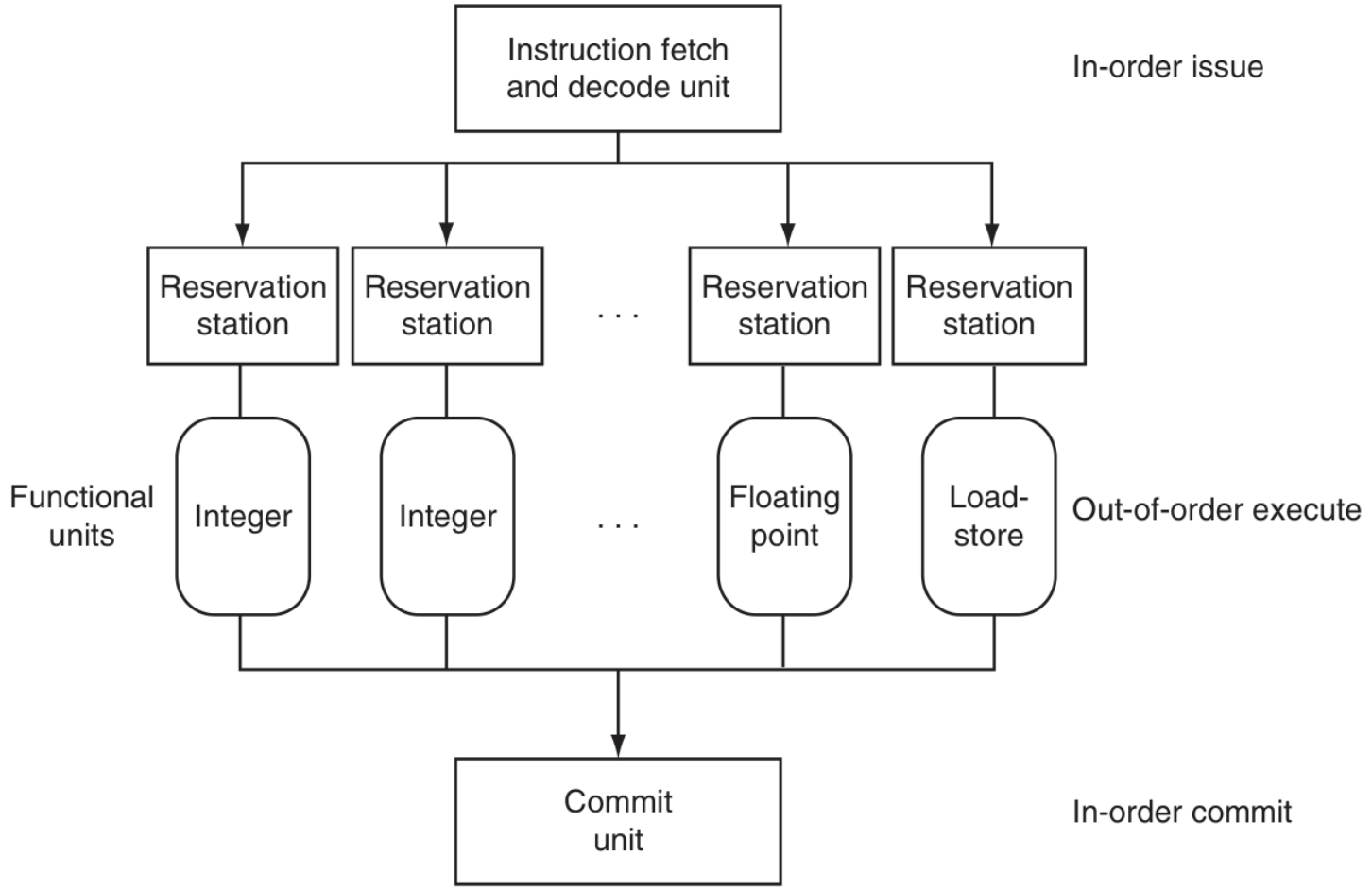


FIGURE 4.73 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

Out-of-Order Execution

With out-of-order execution, a later instruction may execute before a previous instruction so the hardware needs to resolve both write after read (WAR) and write after write (WAW) data hazards.

```
lw      $t0, 0($s1)
addu    $t0, $t1, $s2
. . .
sub     $t2, $t0, $s2
```

- If the lw write to $t0 occurs after the addu write, then the sub gets an incorrect value for $t0
  - The addu has an output dependency on the lw – write after write (WAW)
  - The issuing of the addu might have to be stalled if its result could later be overwritten by an previous instruction that takes longer to complete
  - Hardware register renaming could solve this.

Kernel-memory-leaking Intel Processor Design Flaw

- Intel's OOO CPU allows a speculative user load to access OS kernel data.
  - Fixed with Forcefully Unmap Complete Kernel With Interrupt Trampolines, aka FUCKWIT ☺
  - But slows down your system.
  - https://www.theregister.com/2018/01/02/intel_cpu_design_flaw/

Kernel-memory-leaking Intel processor design flaw forces Linux, Windows redesign

Speed hits loom, other OSes need fixes

Chris Williams — Tue 2 Jan 2018 // 19:29 UTC

FINAL UPDATE A fundamental design flaw in Intel's processor chips has forced a significant redesign of the Linux and Windows kernels to defang the chip-level security bug.

Programmers are scrambling to overhaul the open-source Linux kernel's virtual memory system. Meanwhile, Microsoft is expected to publicly introduce the necessary changes to its Windows operating system in an upcoming Patch Tuesday: these changes were seeded to beta testers running fast-ring Windows Insider builds in November and December.

Crucially, these updates to both Linux and Windows will incur a performance hit on Intel products. The effects are still being benchmarked, however we're looking at a ballpark figure of five to 30 per cent slow down, depending on the task and the processor model. More recent Intel chips have features – such as PCID – to reduce the performance hit. Your mileage may vary.

Summary

Does ILP 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, e.g., limited window size during instruction issue
- Memory delays and limited bandwidth: hard to keep pipelines full
- Speculation can help if done well

Does ILP 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, e.g., limited window size during instruction issue
- Memory delays and limited bandwidth: hard to keep pipelines full
- Speculation can help if done well (But may have security issue)
  - Spectre/Meltdown attack.

Summary: Extracting More Performance

To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by

- Deep pipelining
- Static multiple-issue (VLIW)
- Dynamic multiple-issue (superscalar)

- A processor's instruction issue and execution policies impact the available ILP
- Register renaming can solve these storage dependencies

Evolution of CPU Pipeline

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

FIGURE 4.74 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better (next lecture)