Multicore and Parallelism

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 4.10, 1.7, 1.8, 5.10, 6

Administrivia

Next five weeks

- Week 11 (Apr 15): Proj3 release, Lab3 due Wed, HW2 due Sat
- Week 12 (Apr 22): Lab4 release and Proj3 due Fri
- Week 13 (Apr 29): Proj4 release, Lab4 due Tue, Prelim2
- Week 14 (May 6): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

- Week 15 (May 13): Proj4 due Wed

Today

Many ways to improve performance

- Instruction Level Parallelism
- Multicore
  - Performance in multicore

Next 2 lectures: synchronization

Next lecture: GPU

How to improve performance?

We have looked at

- Pipelining

- To speed up:
    - Deeper pipelining
    - Make the clock run faster
    - Parallelism
        - Not a luxury, a necessity

Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?

A: Deeper pipeline

- E.g. 250MHz 1-stage; 500Mhz 2-stage; 1GHz 4-stage; 4GHz 16-stage

Pipeline depth limited by…

- max clock speed
- min unit of work (less work per stage $\Rightarrow$ shorter clock cycle)
- dependencies, hazards / forwarding logic

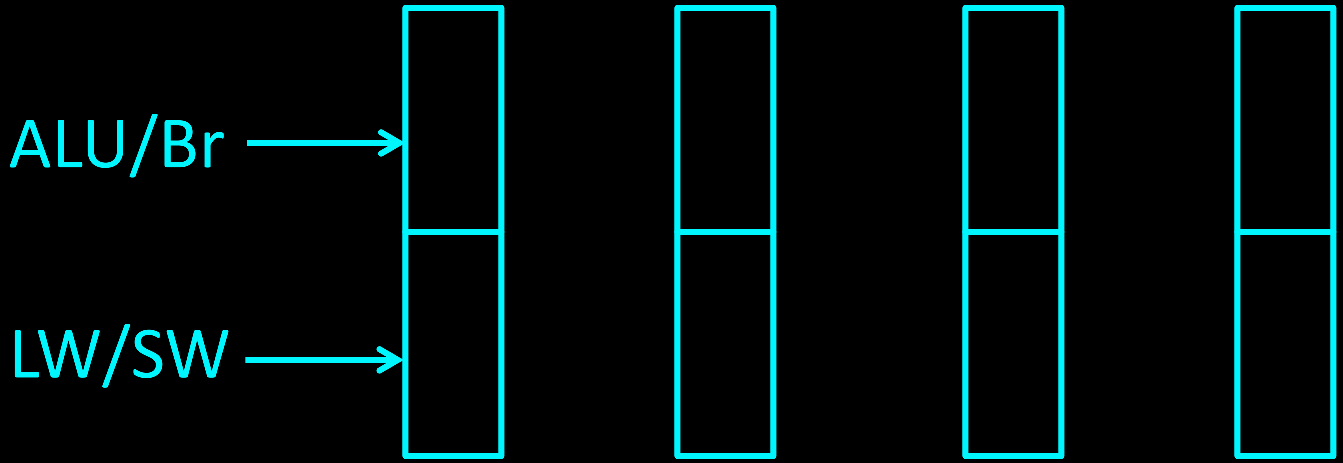
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?

A: Multiple issue pipeline

- Start multiple instructions per clock cycle in duplicate stages



Multiple issue pipeline

Static multiple issue

  aka Very Long Instruction Word

  Decisions made by compiler

Dynamic multiple issue

  Decisions made on the fly

Cost: More execute hardware

  Reading/writing register files: more ports

Static Multiple Issue

a.k.a. Very Long Instruction Word (VLIW)

Compiler groups instructions to be issued together

- Packages them into “issue slots”

Q: How does HW detect and resolve hazards?

A: It doesn’t

→ Simple HW, assumes compiler avoids hazards

Example: Static Dual-Issue 32-bit MIPS

- Instructions come in pairs (64-bit aligned)
  - One ALU/branch instruction (or nop)
  - One load/store instruction (or nop)

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
- Delay slot: 2 instructions (1 cycle)

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	M	WB	

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:			1
			2
			3
			4

Speculation

- Reorder instructions
  - To fill the issue slot with useful work
  - Complicated: exceptions may occur

Optimizations to make it work

Move instructions to fill in nops

    Need to track hazards and dependencies

Loop unrolling

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

5 instructions/4 cycles = IPC = 1.25

4 cycles/5 instructions = CPI = 0.8

Scheduling Example

Compiler scheduling for dual-issue MIPS…

```
Loop:  lw   $t0, 0($s1)          # $t0 = A[i]
       lw   $t1, 4($s1)          # $t1 = A[i+1]
       addu $t0, $t0, $s2        # add $s2
       addu $t1, $t1, $s2        # add $s2
       sw   $t0, 0($s1)          # store A[i]
       sw   $t1, 4($s1)          # store A[i+1]
       addi $s1, $s1, +8         # increment pointer
       bne  $s1, $s3, Loop       # continue if $s1!=end
```

8 cycles

```
      ALU/branch slot          Load/store slot      cycle
Loop: nop                      lw   $t0,  0($s1)      1
      nop                      lw   $t1,  4($s1)      2
      addu $t0, $t0, $s2       nop                    3
      addu $t1, $t1, $s2       sw   $t0,  0($s1)      4
      addi $s1, $s1, +8        sw   $t1,  4($s1)      5
      bne  $s1, $s3, Loop      nop                    6
```

delay slot

6 cycles

= CPI = 0.75

Scheduling Example

Compiler scheduling for dual-issue MIPS…

```
Loop:  lw   $t0, 0($s1)          # $t0 = A[i]
       lw   $t1, 4($s1)          # $t1 = A[i+1]
       addu $t0, $t0, $s2        # add $s2
       addu $t1, $t1, $s2        # add $s2
       sw   $t0, 0($s1)          # store A[i]
       sw   $t1, 4($s1)          # store A[i+1]
       addi $s1, $s1, +8         # increment pointer
       bne  $s1, $s3, Loop       # continue if $s1!=end
```

8 cycles

	ALU/branch slot	Load/store slot	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, +8	lw $t1, 4($s1)	2
	addu $t0, $t0, $s2	nop	3
	addu $t1, $t1, $s2	sw $t0, -8($s1)	4
	bne $s1, $s3, Loop	sw $t1, -4($s1)	5

5 cycles

= CPI = 0.625

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

```
lw   $t0, 0($s1)          # load A
addi $t0, $t0, +1         # increment A
sw   $t0, 0($s1)          # store A
lw   $t0, 0($s2)          # load B
addi $t0, $t0, +1         # increment B
sw   $t0, 0($s2)          # store B
```

ALU/branch slot	Load/store slot	cycle
nop	lw $t0, 0($s1)	1
nop	nop	2
addi $t0, $t0, +1	nop	3
nop	sw $t0, 0($s1)	4
nop	lw $t0, 0($s2)	5
nop	nop	6
addi $t0, $t0, +1	nop	7
nop	sw $t0, 0($s2)	8

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

```
lw   $t0, 0($s1)            # load A
addi $t0, $t0, +1           # increment A
sw   $t0, 0($s1)            # store A
lw   $t1, 0($s2)            # load B
addi $t1, $t1, +1           # increment B
sw   $t1, 0($s2)            # store B
```

ALU/branch slot	Load/store slot	cycle
nop	lw $t0, 0($s1)	1
nop	nop	2
addi $t0, $t0, +1	nop	3
nop	sw $t0, 0($s1)	4
nop	lw $t1, 0($s2)	5
nop	nop	6
addi $t1, $t1, +1	nop	7
nop	sw $t1, 0($s2)	8

Limits of Static Scheduling

Compiler scheduling for dual-issue MIPS…

```
lw   $t0, 0($s1)          # load A
addi $t0, $t0, +1         # increment A
sw   $t0, 0($s1)          # store A
lw   $t1, 0($s2)          # load B
addi $t1, $t1, +1         # increment B
sw   $t1, 0($s2)          # store B
```

```
ALU/branch slot         Load/store slot      cycle
nop                     lw   $t0,  0($s1)      1
nop                     lw   $t1,  0($s2)      2
addi $t0, $t0, +1       nop                    3
addi $t1, $t1, +1       sw   $t0,  0($s1)      4
nop                     sw   $t1,  0($s2)      5
```

Problem: What if $s1 and $s2 are equal (***aliasing***)? Won’t work

Dynamic Multiple Issue

a.k.a. SuperScalar Processor

CPU examines instruction stream and chooses multiple instructions to issue each cycle

- Compiler can help by reordering instructions….
- … but CPU is responsible for resolving hazards

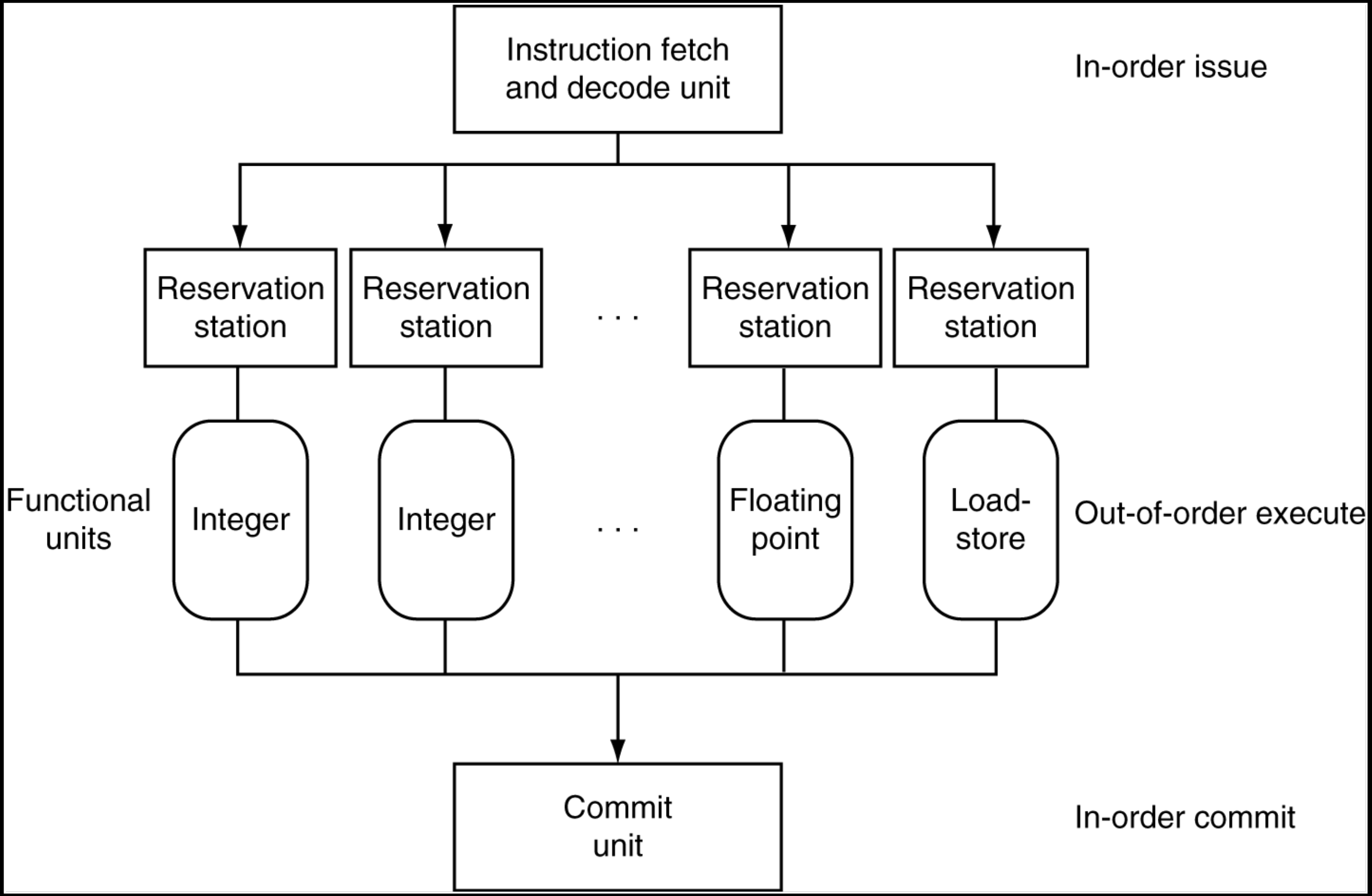
Dynamic Multiple Issue

a.k.a. SuperScalar Processor

Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insts. are retired

Dynamic Multiple Issue



Why dynamic scheduling?

To handle unpredictable stalls

- Like cache misses

Hides details of pipeline from applications

- Abstraction

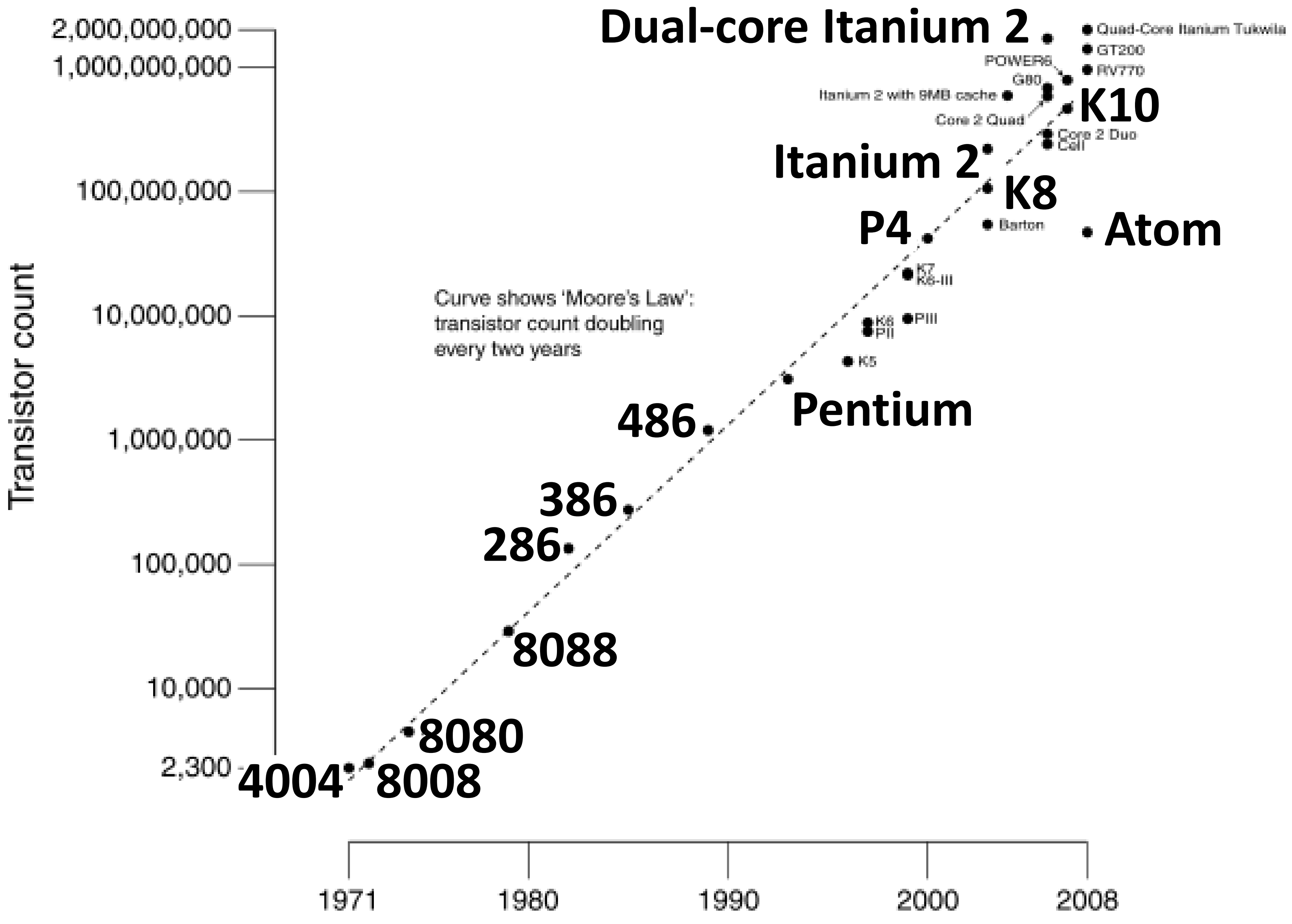
Does Multiple Issue Work?

Q: Does multiple issue / ILP work?

A: Kind of… but not as much as we'd like

Limiting factors?

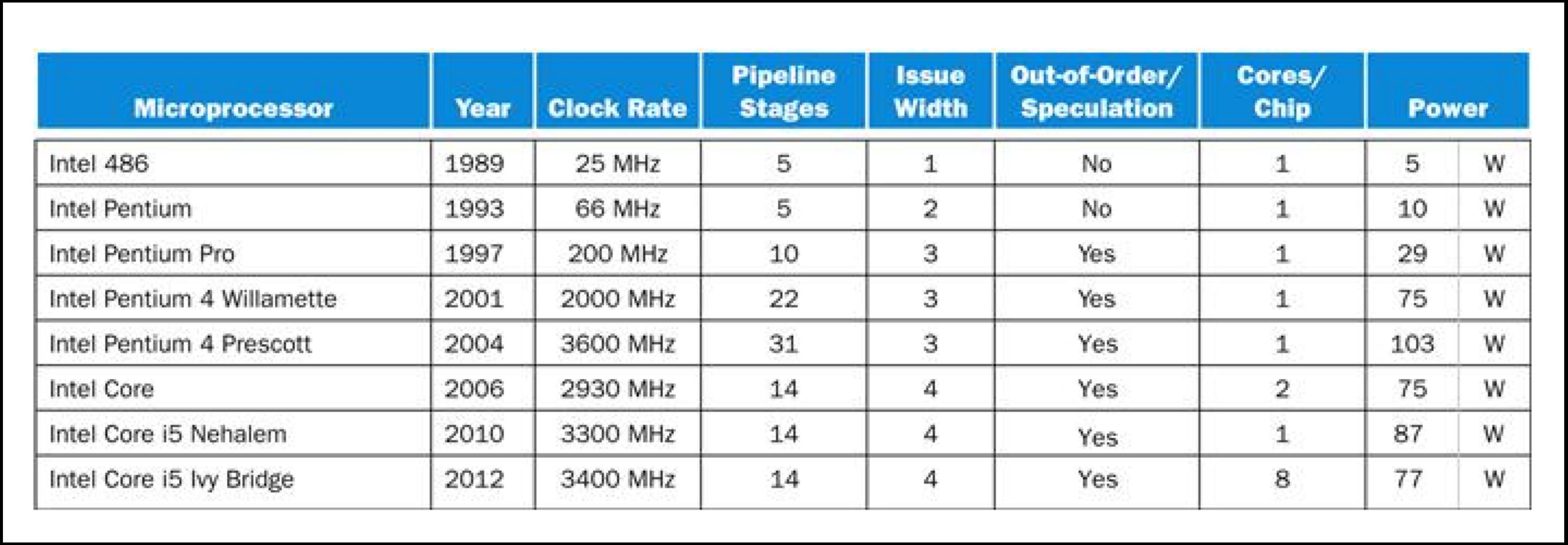
- Programs dependencies
- Hard to detect dependencies → be conservative
  - e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
- Hard to expose parallelism
  - Can only issue a few instructions ahead of PC
- Structural limits
  - Memory delays and limited bandwidth
- Hard to keep pipelines full


Dual-core Itanium 2
Quad-Core Itanium Tukwila
GT200
POWER6
RV770
G80
Itanium 2 with 9MB cache
K10
Core 2 Quad
Core 2 Duo
Cell
Itanium 2
K8
P4
Barton
Atom
K7
K6-III
Curve shows 'Moore's Law': transistor count doubling every two years
K6
PII
PIII
K5
Pentium
486
386
286
8088
8080
4004
8008
Transistor count
2,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2008

Power Efficiency

Q: Does multiple issue / ILP cost much?

A: Yes. Dynamic issue & speculation requires power



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	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

→ Multiple simpler cores may be better?

Why Multicore?

Moore’s law

- A law about transistors
- Smaller means more transistors per die
- And smaller means faster too

But: need to worry about power too…

Power Wall

Power = capacitance * voltage^2 * frequency

approx. capacitance * voltage^3

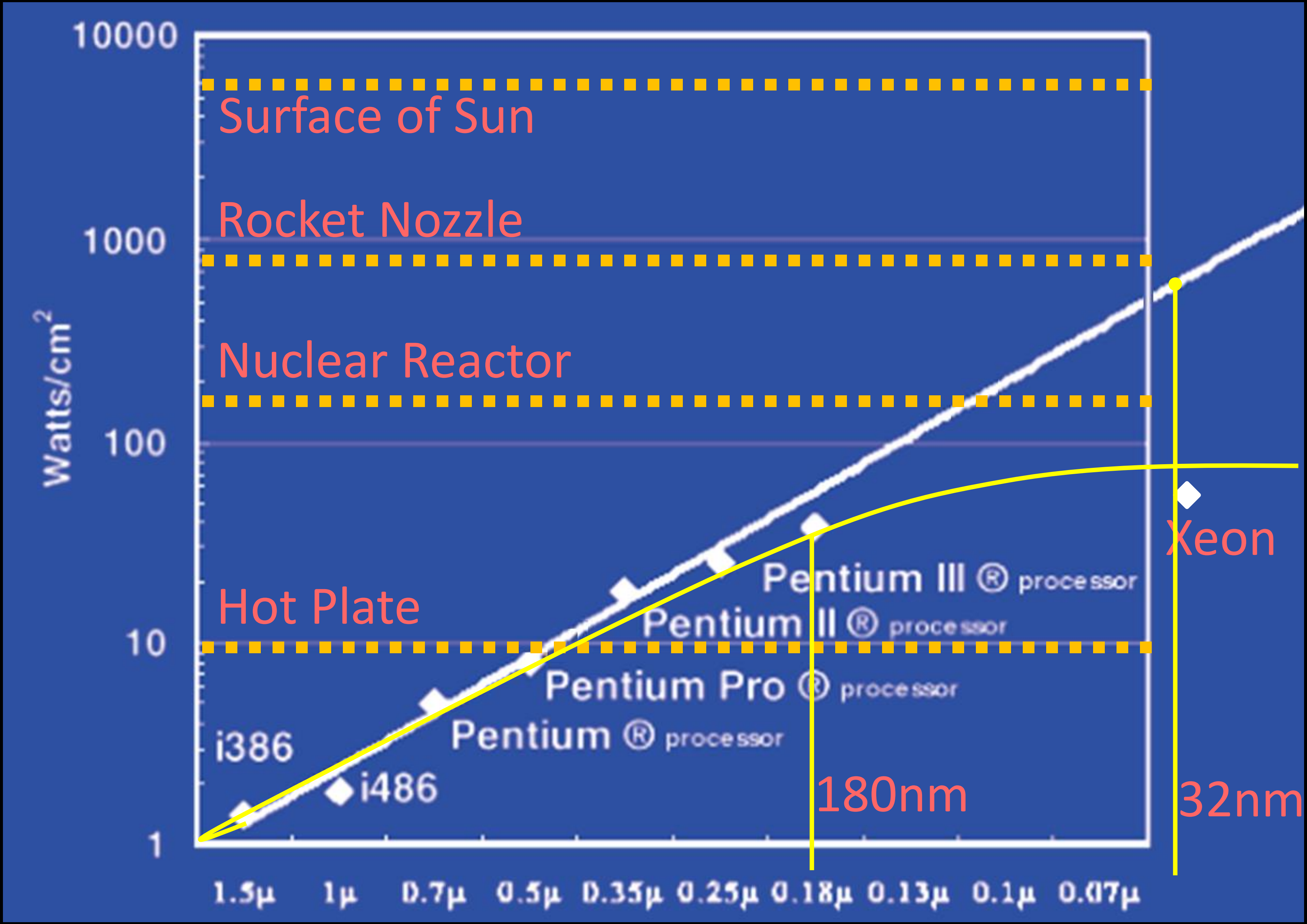
Reducing voltage helps (a lot)

Better cooling helps

The power wall

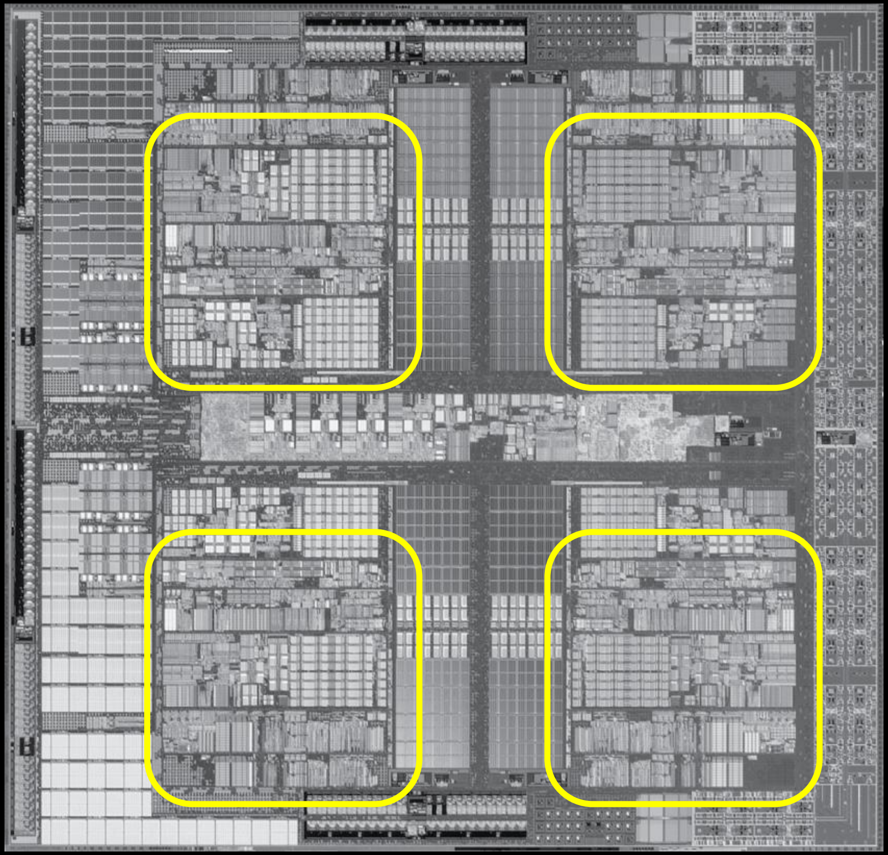
- We can’t reduce voltage further - leakage
- We can’t remove more heat

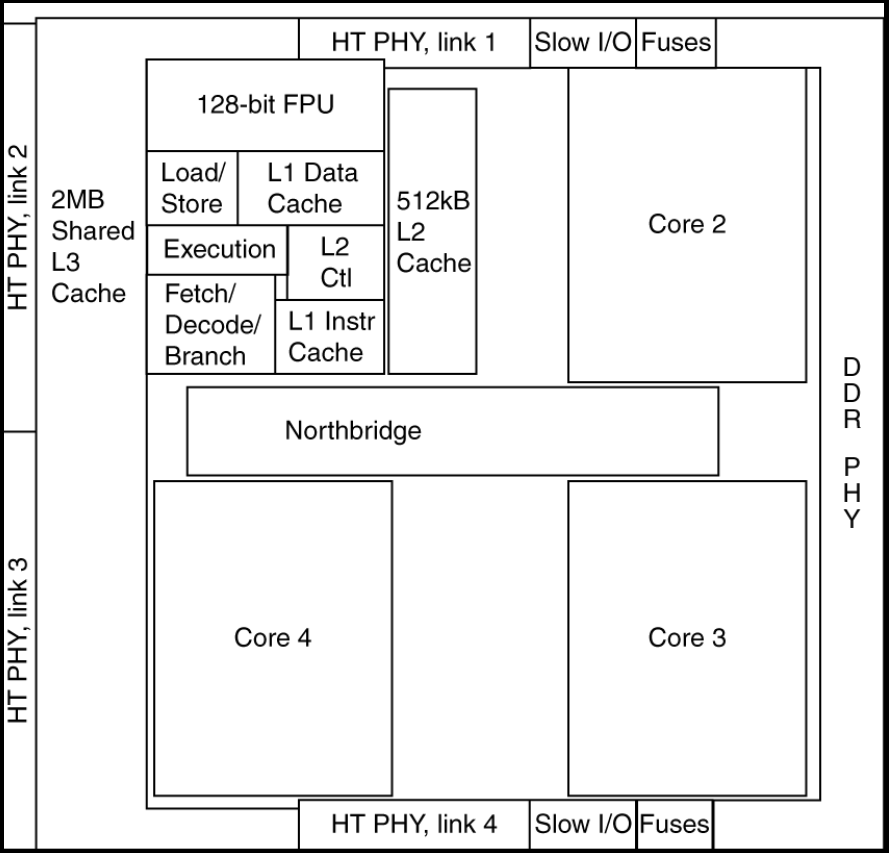
Power Limits



Inside the Processor

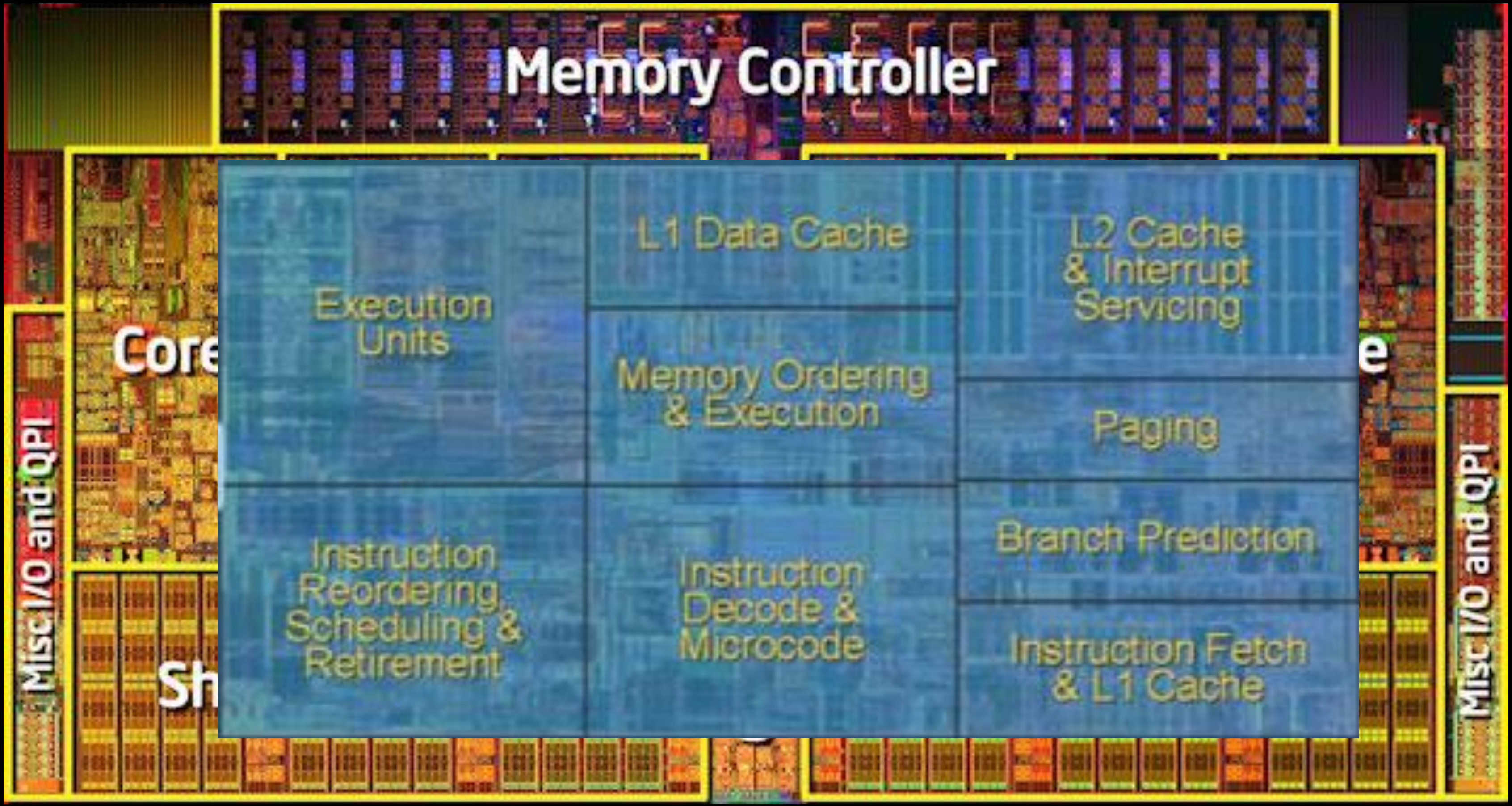
AMD Barcelona Quad-Core: 4 processor cores





Inside the Processor

Intel Nehalem Hex-Core


Memory Controller
Core
Sh
Misc I/O and QPI
Misc I/O and QPI
e
Execution Units
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
Branch Prediction
Instruction Reordering, Scheduling & Retirement
Instruction Decode & Microcode
Instruction Fetch & L1 Cache

Amdahl's Law

If tasks have a serial part and a parallel part…

Example:

step 1: divide input data into *n* pieces

step 2: do work on each piece

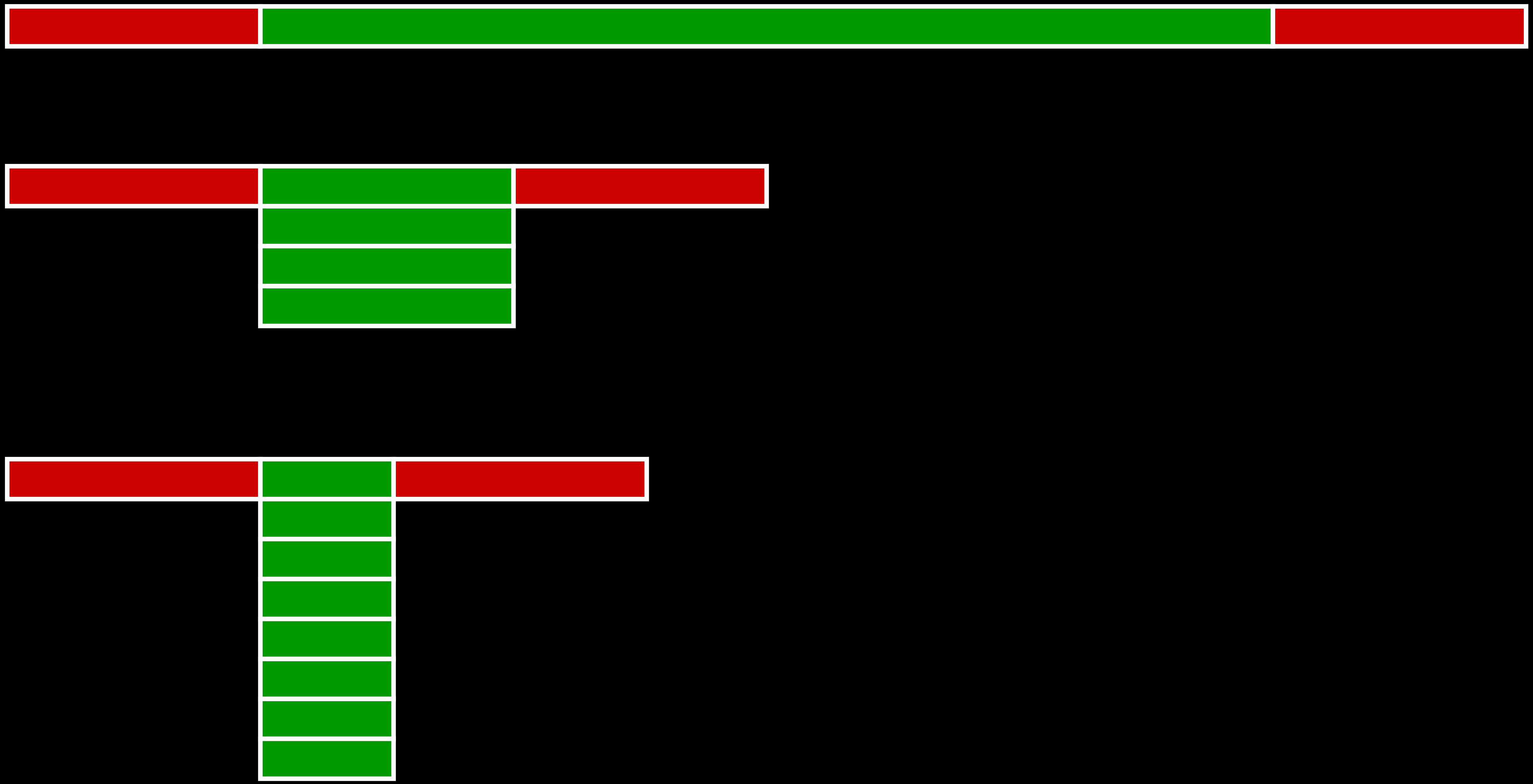
step 3: combine all results

Recall: Amdahl's Law

As number of cores increases …

- time to execute parallel part? goes to zero
- time to execute serial part? Remains the same
- *Serial part eventually dominates*

Amdahl's Law



Pitfall: Amdahl's Law

$$\text{Execution time after improvement} = \frac{\text{affected execution time}}{\text{amount of improvement}} + \text{execution time unaffected}$$

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Pitfall: Amdahl's Law

Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Example: multiply accounts for 80s out of 100s

- How much improvement do we need in the multiply performance to get 5× overall improvement?

$$20 = 80/n + 20$$ ***Can't be done!***

Scaling Example

Workload: sum of 10 scalars, and 10×10 matrix sum

- Speed up from 10 to 100 processors?

Single processor: Time = $(10 + 100) \times t_{add}$

10 processors

- Time = $100/10 \times t_{add} + 10 \times t_{add} = 20 \times t_{add}$
- Speedup = $110/20 = 5.5$

100 processors

- Time = $100/100 \times t_{add} + 10 \times t_{add} = 11 \times t_{add}$
- Speedup = $110/11 = 10$

Assumes load can be balanced across processors

Scaling Example

What if matrix size is 100×100?

Single processor: Time = $(10 + 10000) \times t_{add}$

10 processors

- Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
- Speedup = $10010/1010 = 9.9$

100 processors

- Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
- Speedup = $10010/110 = 91$

Assuming load balanced

Scaling

Strong scaling vs. weak scaling

Strong scaling: scales with same problem size

Weak scaling: scales with increased problem size