Recitation 1:

ILP, SIMD, and Thread Parallelism

15-418 Parallel Computer Architecture and Programming

CMU 15-418/15-618, Spring 2020

Goals for today

- Topic is parallelism models: ILP, SIMD, threading

- Solve some exam-style problems
- Walk through example code

- Most of all,

<u>ANSWER YOUR QUESTIONS!</u>

Big lessons from today

- *Focus your effort on the performance bottleneck*
- Usually, you do **not** need to model processor in detail to find the performance bottleneck

Recall: Taylor expansion of $\sin(x)$

```
void sinx(int N, int terms, float * x,
          float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

- How fast is this code?
- Where should we focus optimization efforts?
- What is the bottleneck?

Recall: Taylor expansion of $\sin(x)$

```
void sinx(int N, int terms, float * x,
          float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

- How fast is this code?
- On ghc machines: 5 ns / element ≈ 23 cycles / element
- Not very good ☹

Recall: Taylor expansion of $\sin(x)$

```
void sinx(int N, int terms, float * x,
          float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

- Where should we focus optimization efforts?
- A: Where most of the time is spent

Recall: Taylor expansion of $\sin(x)$

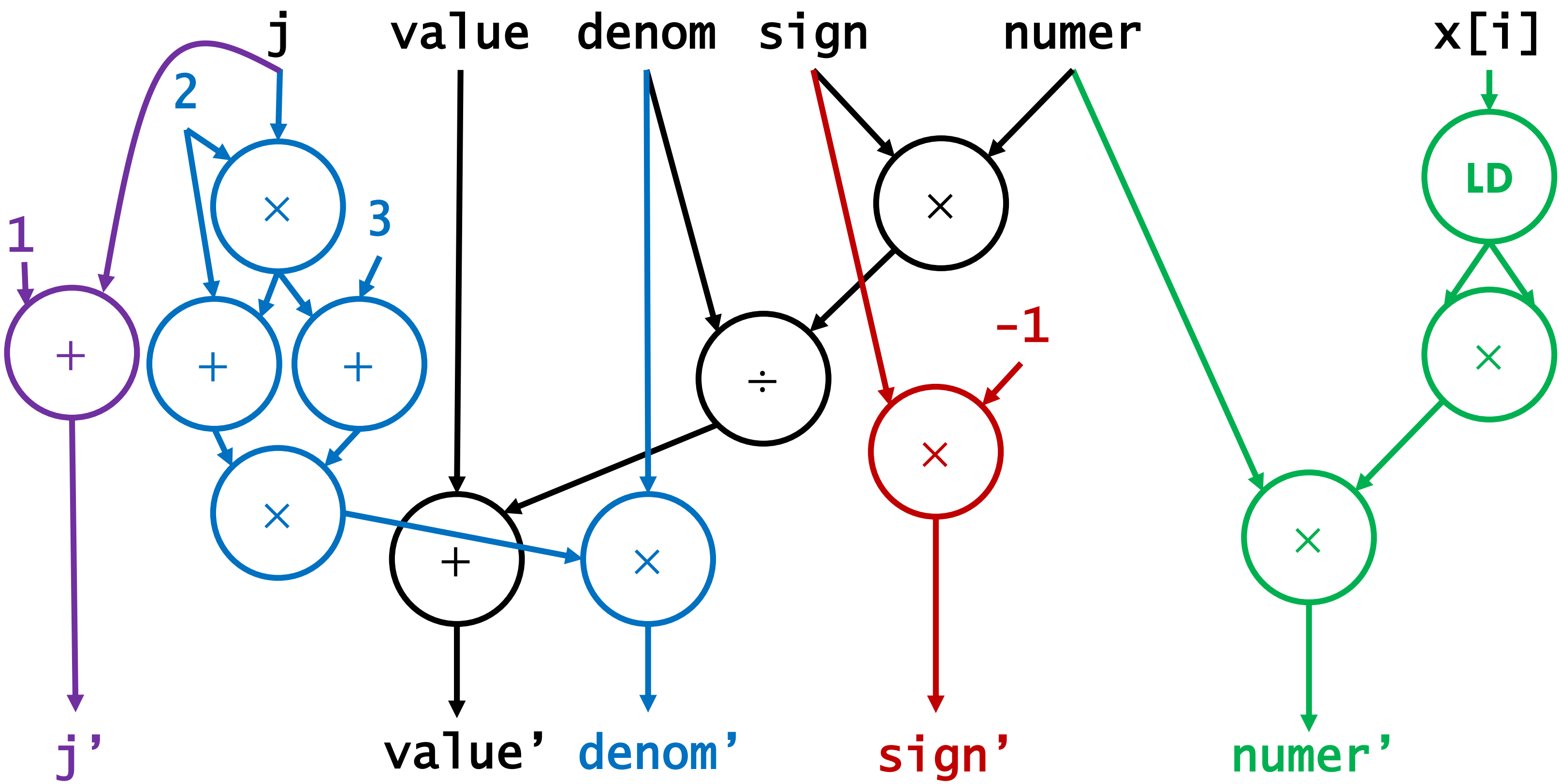
```
void sinx(int N, int terms, float * x,
          float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

- What is the bottleneck?

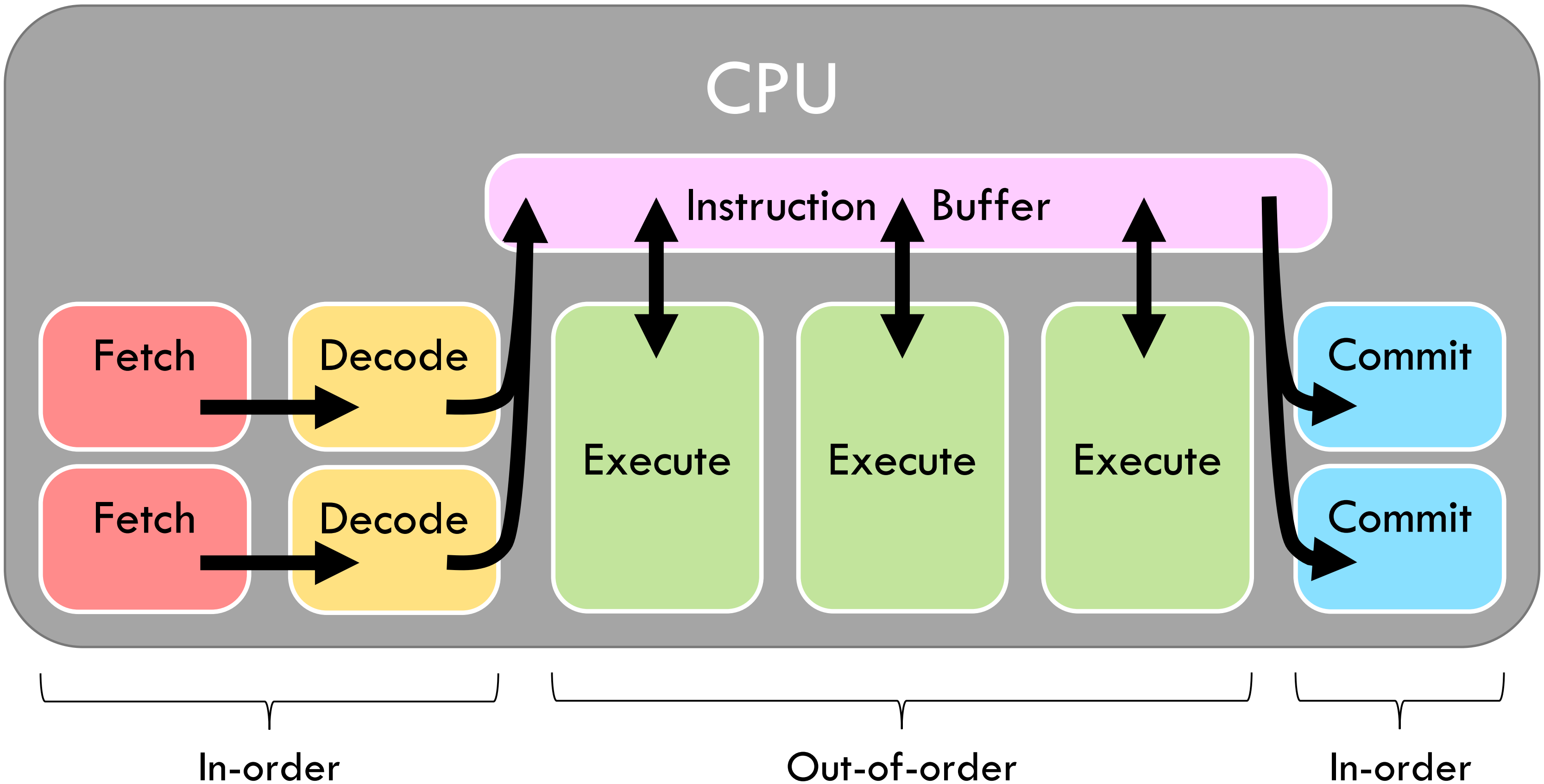
Dataflow for a single iteration



OK, but how does this perform on a real machine?

Superscalar OOO Processor

- What in microarchitecture should we worry about?



GHC Machine Microarchitecture

- What in microarchitecture should we worry about?
  - Fetch & Decode? **NO.** Any reasonable machine will have sufficient frontend throughput to keep execution busy + all branches in this code are easy to predict (n<u>ot</u> always the case!).
  - Execution? **YES.** This is where dataflow + most structural hazards will limit our performance.
  - Commit? **NO.** Again, any reasonable machine will have sufficient commit throughput to keep execution busy.

Intel Skylake (GHC machines) Execution Microarchitecture

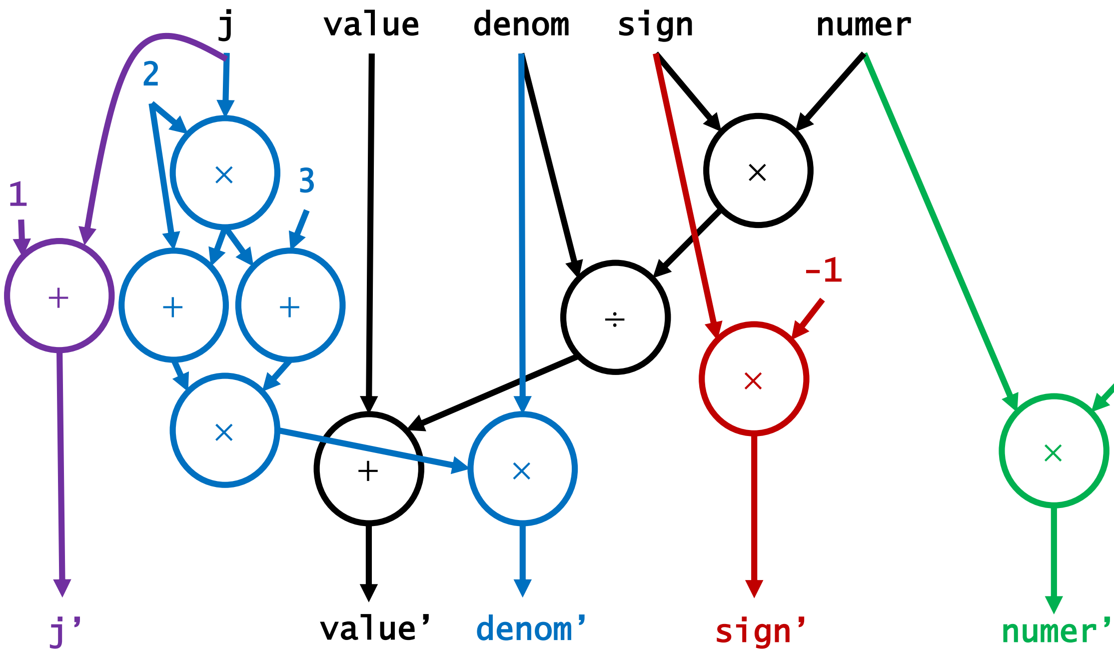
	Integer			Floating Point		
	Latency	Pipelined?	Number	Latency	Pipelined?	Number
Add	1	✓	4	4*	✓	2
Multiply	3	✓	1	4	✓	2
Divide	21-83	✗	1	13-14	✗**	1
Load	2	✓	2			

* 3 cycles if using x87 instructions

** Can issue another operation after 4 cycles

Source: Search for "Skylake" in
https://www.agner.org/optimize/microarchitecture.pdf
https://www.agner.org/optimize/instruction_tables.pdf

What is our **throughput bound?**



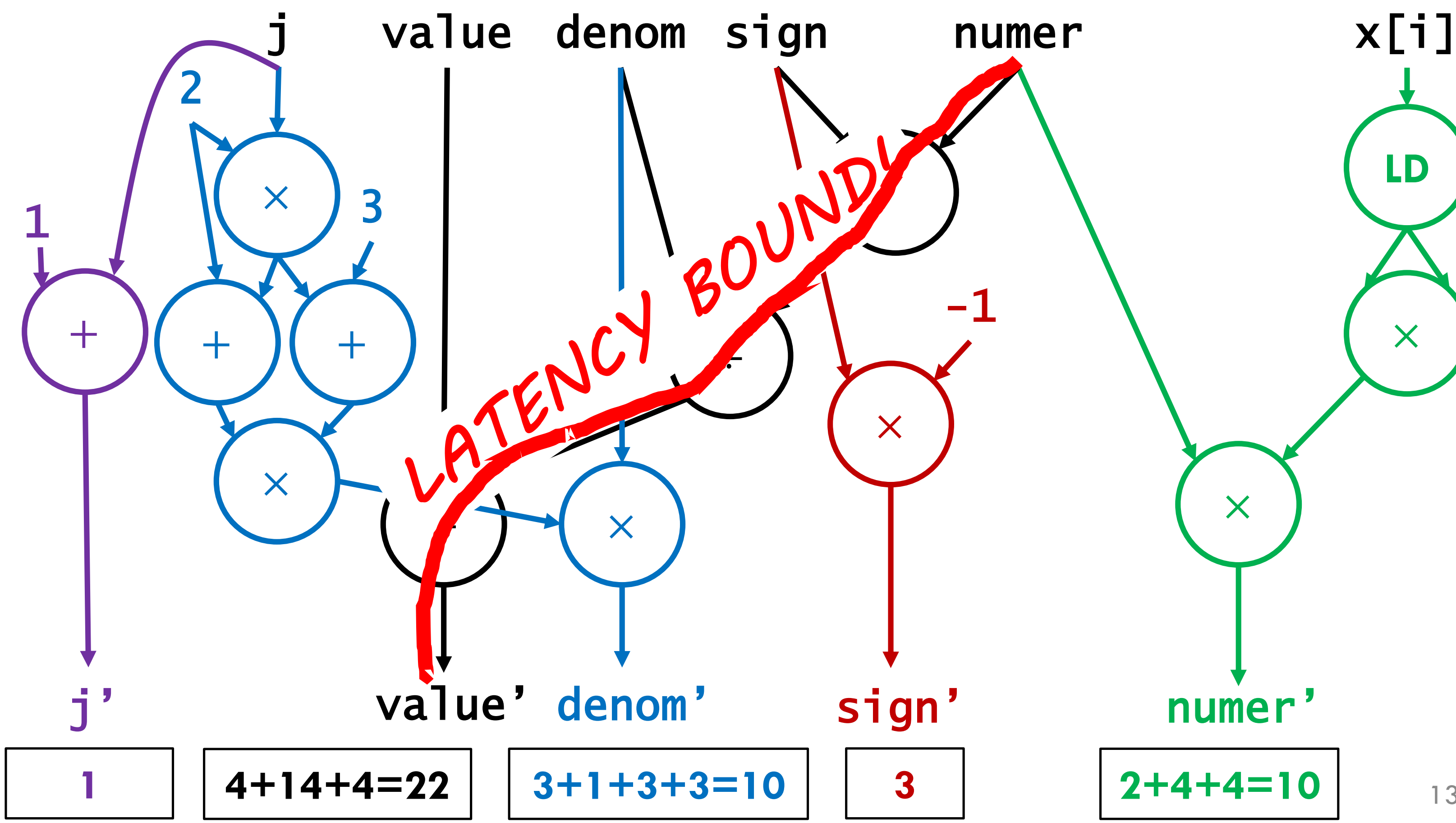
Op	# Code	μArch	Thput bound
Int Add			
Int Mul			
Int Di			
FP Div			
Load			

THPUT BOUND!

Throughput bound: Ignore data hazards, think *only* about max issue rate due to <u>structural hazards</u>

What is our **latency bound?**

- **Latency bound:** Ignore structural hazards, think *only* about the critical path through data hazards



Takeaways

- Observed 23 cycles / element

- Latency bound dominates throughput bound
  ➔ We are latency bound!

- Notes
  - This analysis can often be "eyeballed" w/out full dataflow
  - Actual execution is more complicated, but latency/thput bounds are good approximation
  - (Also: avoid division!!!)

Speeding up $\sin(x)$: Attempt #1

- What if we eliminate unnecessary work?

```
void sinx_better(int N, int terms, float * x,
                 float *result) {
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = x[i]*x[i];
        float numer = x2*x[i];
        int denom = 6; // 3!
        int sign = -1;

        for (int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x2;
            denom *= (2*j+2) * (2*j+3);
            sign = -sign;
        }

        result[i] = value;
    }
}
```

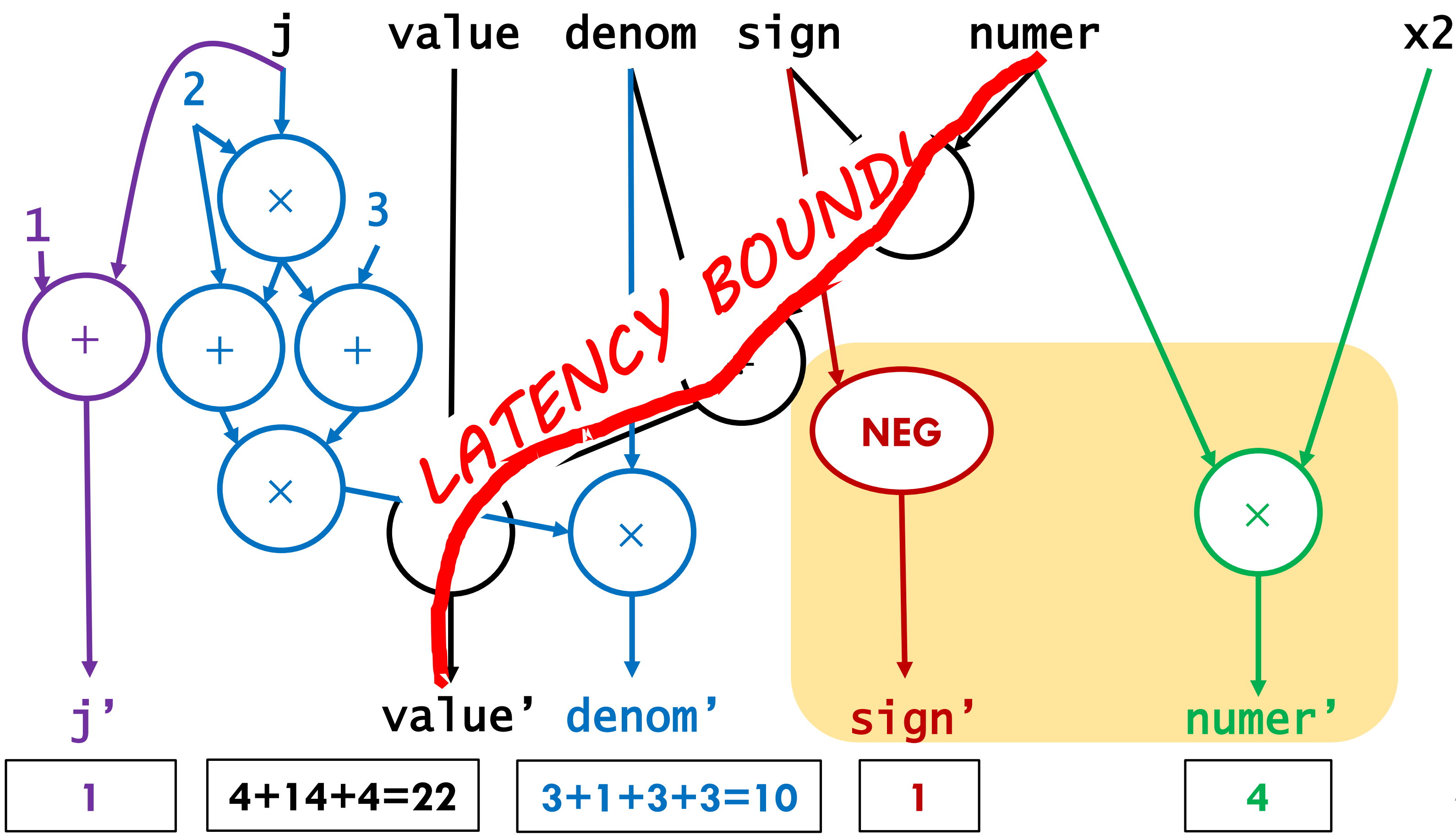
A: Little to no improvement.

5ns / element ≈ 23 cycles / element

Why not better?

What is our **latency bound?**

- Find the critical path in the dataflow graph



Attempt #1 Takeaways

- Optimizations did not improve performance!

- To get real speedup, we need to *focus on the performance bottleneck*

Speeding up $\sin(x)$: Attempt #2

- Let’s focus on that pesky division…

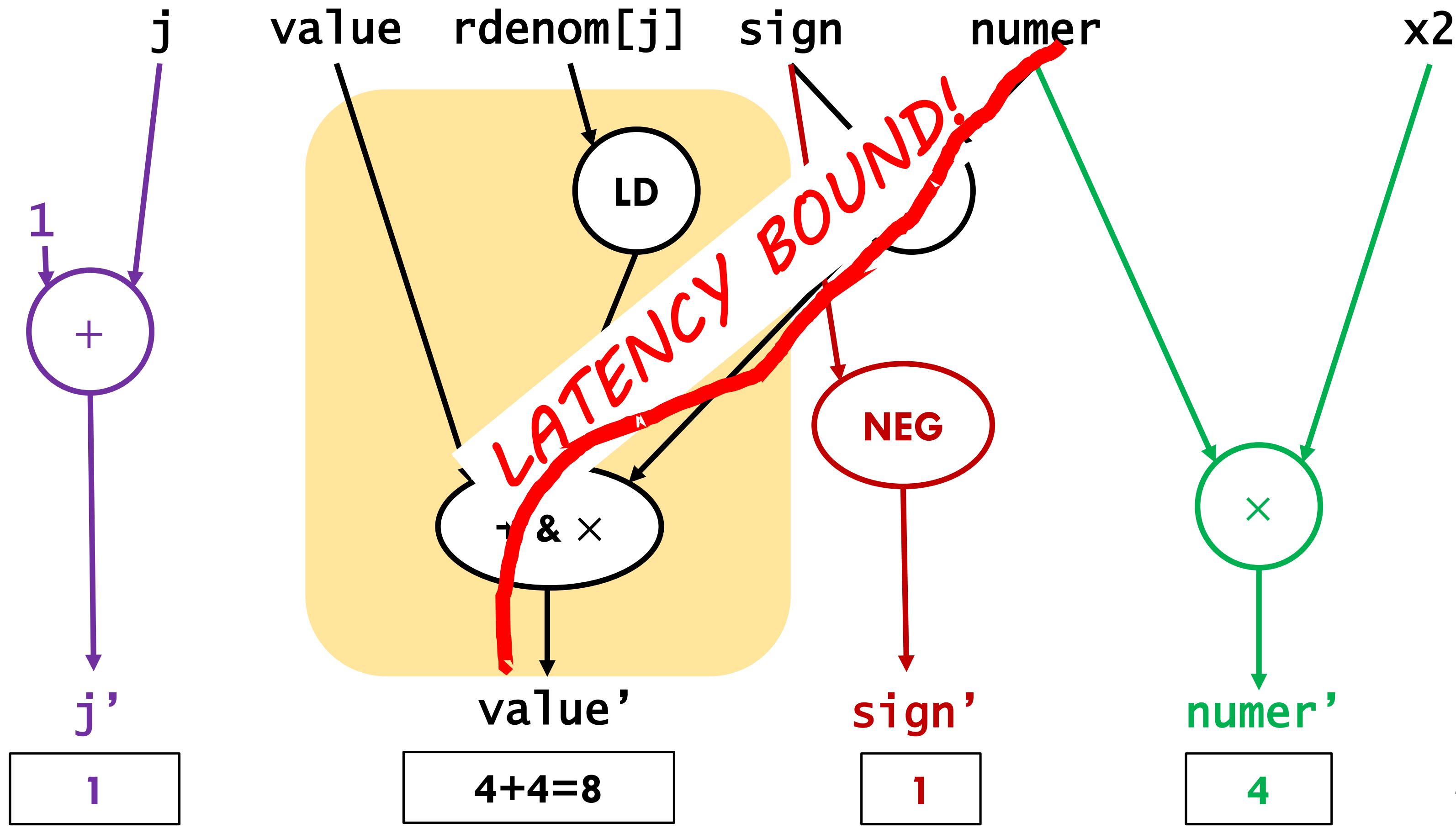
```
void sinx_predenom(int N, int terms, float * x, float *result) {
    float rdenom[MAXTERMS];
    int denom = 6;
    for (int j = 1; j <= terms; j++) {
        rdenom[j] = 1.0/denom;
        denom *= (2*j+2) * (2*j+3);
    }
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float numer = x2 * value;
        int sign = -1;
        for (int j=1; j<=terms; j++) {
            value += sign * numer * rdenom[j];
            numer *= x2;
            sign = -sign;
        }
        result[i] = value;
    }
}
```

A: Big improvement!

2.3ns / element ≈ 10.8 cycles/element

What is our **latency bound?**

- Find the critical path in the dataflow graph



Attempt #2 Takeaways

- Here we go! Attacking the bottleneck got $> 2\times$!
- Gap between observed performance and latency bound widens (10.8 cycles vs 8 cycles)
  - This is normal! Latency/throughput bounds are a simplification; often you will not achieve them
  - It is usually **not** worth the effort to model execution in detail to understand why!
- …But performance is still near the latency bound, can we do better?

Speeding up $\sin(x)$: Attempt #3

- Don't need sign in inner-loop either

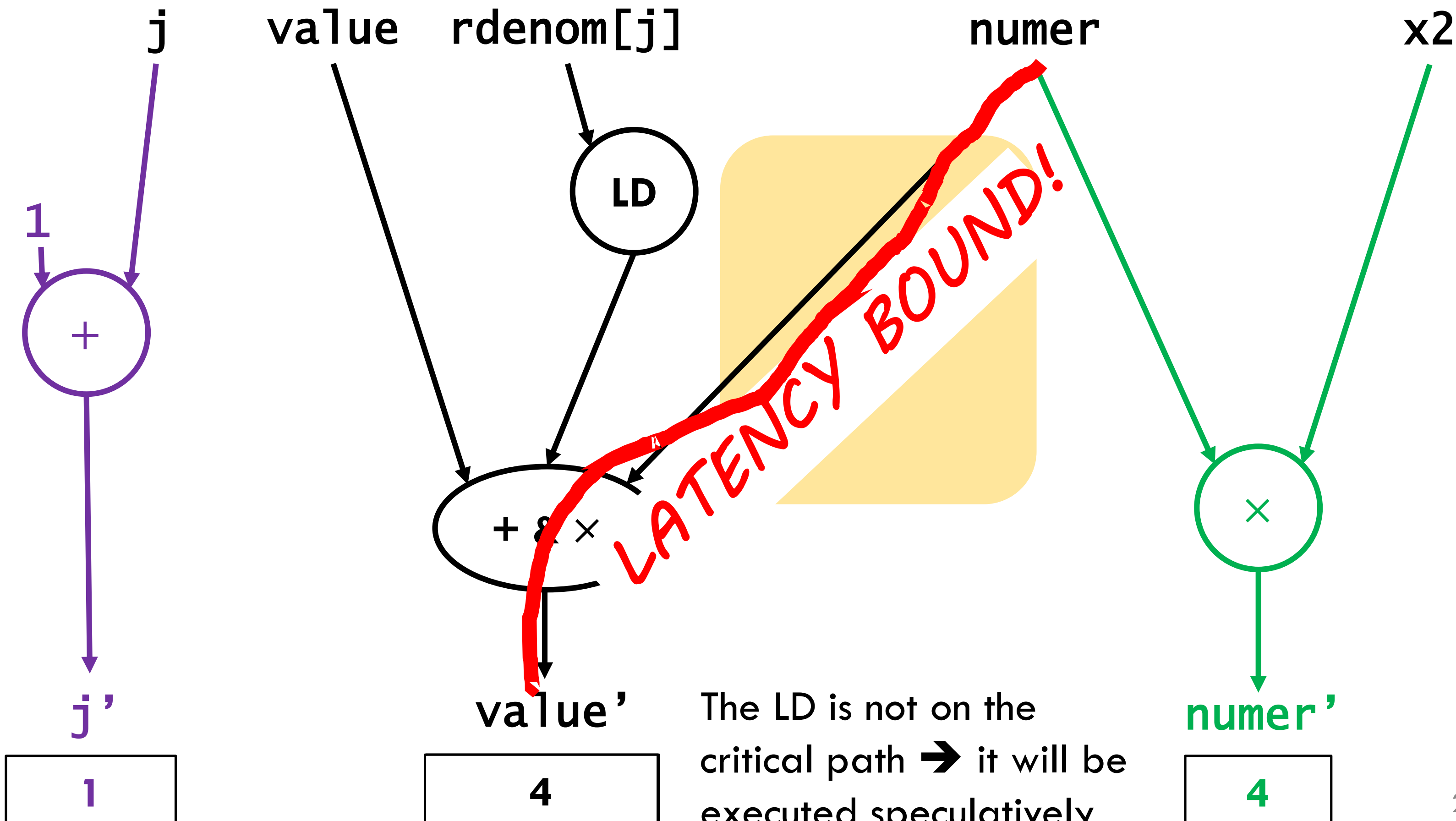
```
void sinx_predenoms(int N, int terms, float * x, float *result) {
    float rdenom[MAXTERMS];
    int denom = 6;
    float sign = -1.0;
    for (int j = 1; j <= terms; j++) {
        rdenom[j] = sign/denom;
        denom *= (2*j+2) * (2*j+3);
        sign = -sign;
    }
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float numer = x2 * value;
        for (int j=1; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.78 ns / element ≈

3.8 cycles / element

What is our **latency bound?**

- Find the critical path in the dataflow graph



The LD is not on the critical path ➔ it will be executed speculatively

Attempt #3 Takeaways

- We're down to the latency of a single, fast operation per iteration
- \+ Observed performance is very close to this latency bound, so throughput isn't limiting
- ➔ We're done optimizing individual iterations

- How to optimize multiple iterations?
  - Eliminate dependence chains across iterations
  - A) Loop unrolling (ILP)
  - B) Explicit parallelism (SIMD, threading)

Speeding up $\sin(x)$: Loop unrolling

- Compute multiple elements per iteration

```
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before…
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        for (int j=1; j<=terms; j+=2) {
            value += numer * rdenom[j];
            value += numer * x2 * redom[j+1];
            numer *= x4;
        }
        result[i] = value;
    }
}
```

Correct? Not yet…

Speeding up $\sin(x)$: Loop unrolling

- Compute multiple elements per iteration

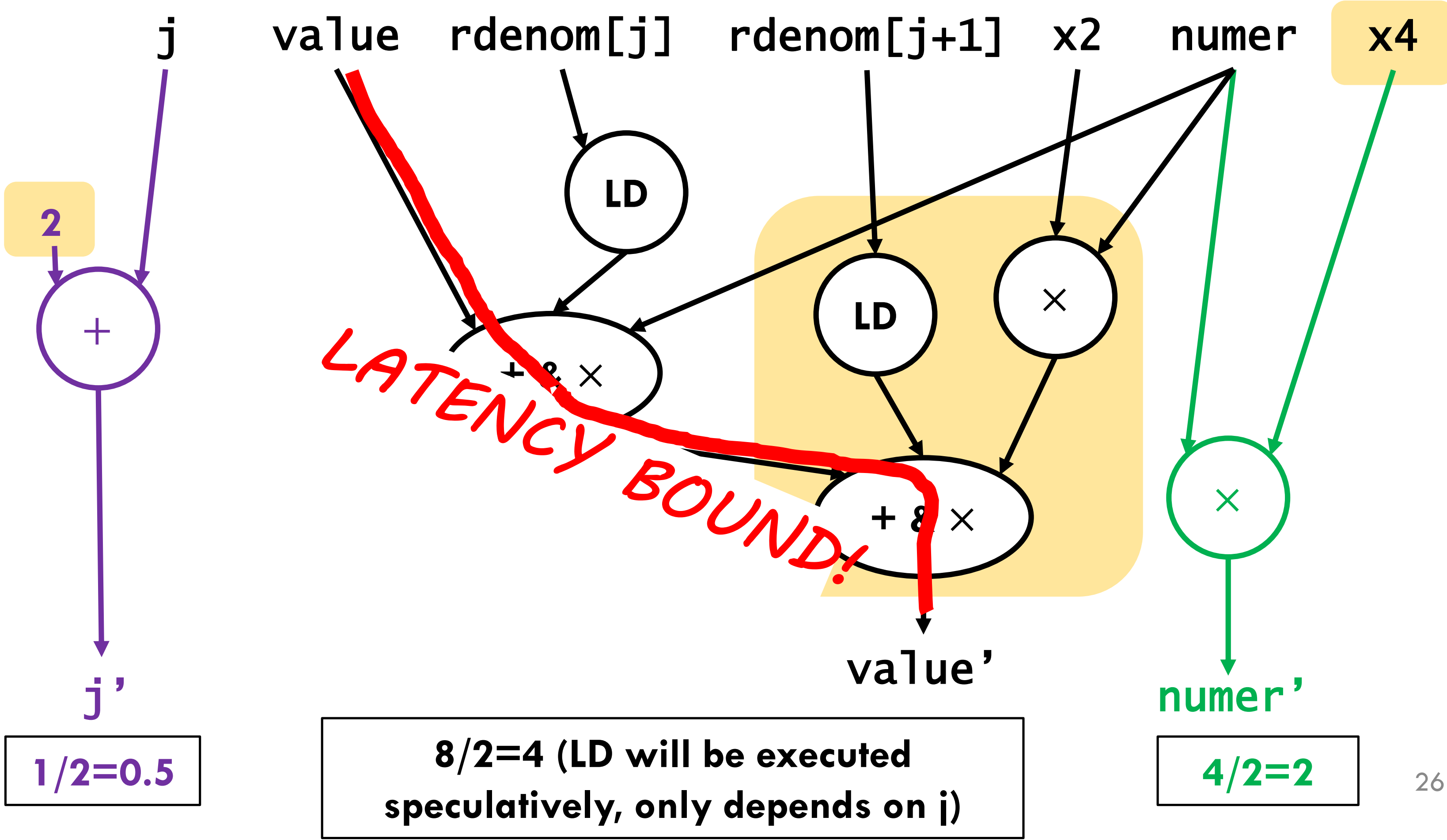
```
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before…
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        int j;
        for (j=1; j<=terms-1; j+=2) {
            value += numer * rdenom[j];
            value += numer * x2 * redom[j+1];
            numer *= x4;
        }
        for (; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.7 ns / element ≈ 3.3 cycles / element

☹

What is our **latency bound?**

- Find the critical path in the dataflow graph



Speeding up $\sin(x)$: Loop unrolling #2

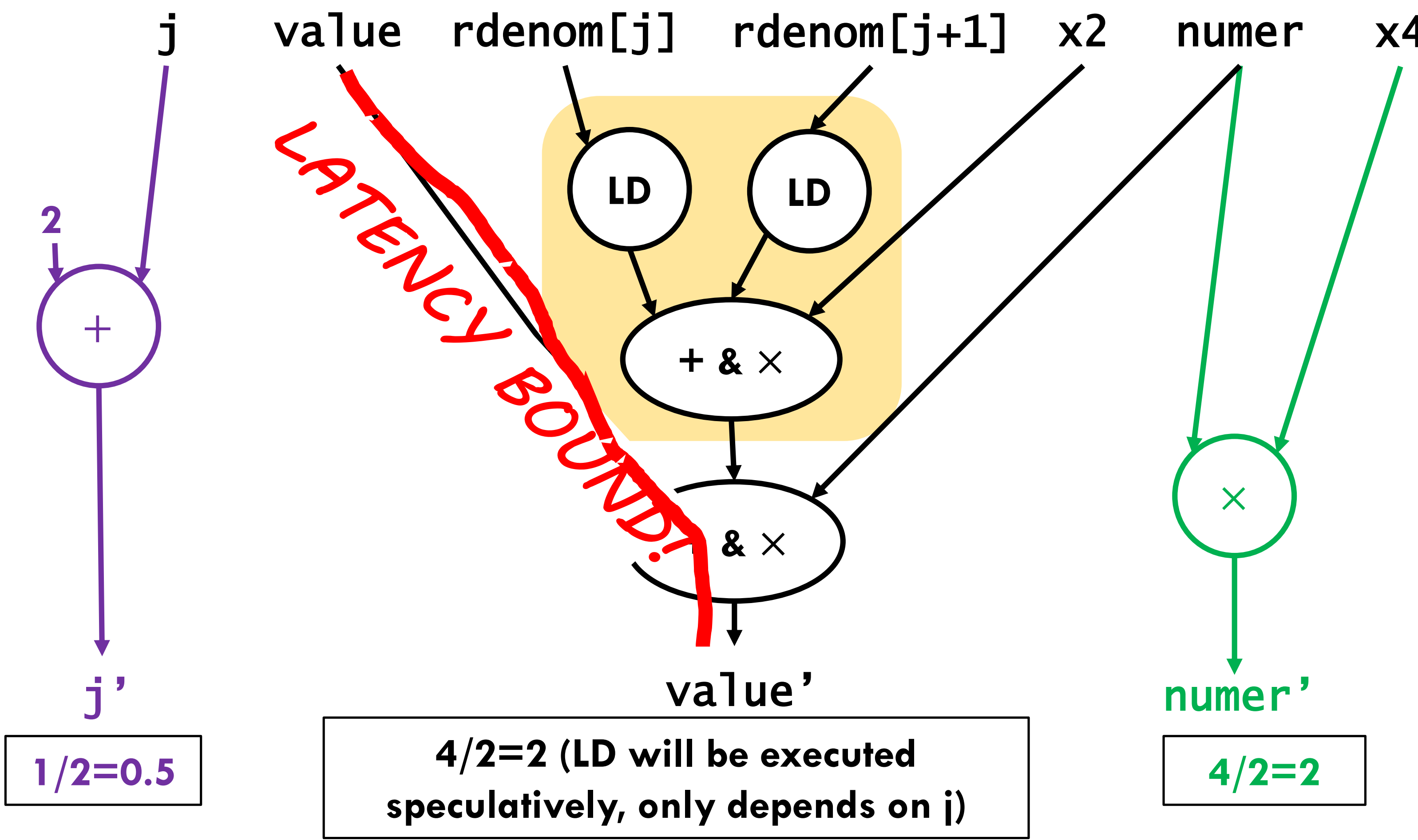
- What if floating point associated + distributed?

```
void sinx_unrollx2(int N, int terms, float * x, float *result) {
    // same predom stuff as before…
    for (int i=0; i<N; i++) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        int j;
        for (j=1; j<=terms-1; j++) {
            value += numer * (rdenom[j] + x2 * redom[j+1]);
            numer *= x4;
        }
        for (; j<=terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
}
```

0.55 ns / element ≈
2.6 cycles / element

What is our **latency bound?**

- Find the critical path in the dataflow graph



Loop unrolling takeaways

- Need to break dependencies across iterations to get speedup
  - Unrolling by itself doesn't help

- We are now seeing throughput effects
  - Latency bound = 2 vs. observed = 2.6

- Can unroll loop 3x, 4x to improve further, but…
- …Diminishing returns (1.5 cycles / element at 5x)

Speeding up $\sin(x)$: Going parallel (explicitly)

- Use ISPC to vectorize the code

```
export void sinx_reference(uniform int N, uniform int terms,
                           uniform float x[],
                           uniform float result[]) {
   foreach (i=0 ... N) {
        float value = x[i];
        float numer = x[i]*x[i]*x[i];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom;
            numer *= x[i] * x[i];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }

        result[i] = value;
    }
}
```

0.26 ns / element ≈ 1.25 cycles / element

Speeding up $\sin(x)$: Going parallel (explicitly) + optimize

```
export void sinx_unrollx2a(uniform int N, uniform int terms,
                           uniform float x[],
                           uniform float result[]) {
    uniform float rdenom[MAXTERMS];
    uniform int denom = 6;
    uniform float sign = -1;
    for (uniform int j = 1; j <= terms; j++) {
        rdenom[j] = sign/denom;
        denom *= (2*j+2) * (2*j+3);
        sign = -sign;
    }
    foreach (i=0 ... N) {
        float value = x[i];
        float x2 = value * value;
        float x4 = x2 * x2;
        float numer = x2 * value;
        uniform int j;
        for (j=1; j<=terms-1; j+=2) {
            value +=
              numer * (rdenom[j] +
                       x2 * rdenom[j+1]);
            numer *= x4;
        }
        for (; j <= terms; j++) {
            value += numer * rdenom[j];
            numer *= x2;
        }
        result[i] = value;
    }
```

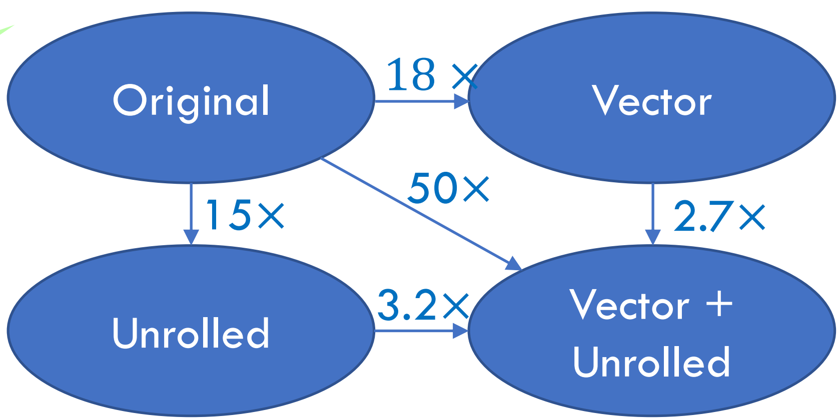
0.096 ns / element ≈
0.46 cycles / element

SIMD takeaways

- Well, that was easy! (Thanks ISPC)
- Cycles per element:

	Scalar	Vector
Unoptimized	23	1.25
Unrolled	1.5	0.46

- Speedup



Maximum speedup requires hand tuning + explicit parallelism!

What if? #1
Impact of structural hazards

- Q: What would happen to $\sin(x)$ if we only had a single, unpipelined floating-point multiplier?

- A1: Performance will be much worse
- A2: We will hit throughput bound much earlier
- A3: Loop unrolling will help by reducing multiplies

What if? #2
Impact of structural hazards

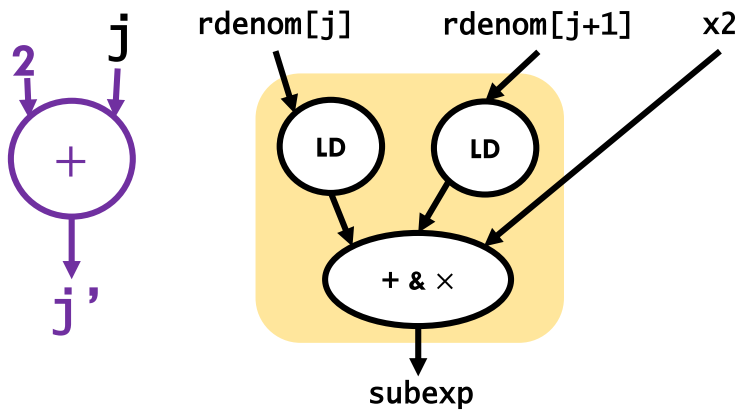
- Q: What would happen to $\sin(x)$ if LDs (cache hits) took 3 cycles instead of 2 cycles?

- A: Nothing. This program is latency bound, and LDs are not on the critical path.

Loads do not limit $\sin(x)$

- Consider just the slice of the program that generates the subexpression: $(\mathrm{rdenom[j]} + \mathrm{x2} \times \mathrm{rednom[j+1]})$

- What is this program's latency + throughput bound?



- Latency bound: **1 cycle / iteration!**
  - Through j′ computation, not the subexpression computation – there is no cross-iteration dependence in the subexpression!)
- Throughput bound: also 1 cycle / iteration
  - 1 add / 4 adders; 2 LDs / 2 LD units; 1 FP FMA / 1 FP unit
  - (This will change to 2 cycles if we add the `value` FMA)

What if? #3
Vector vs. multicore

- Q: What would happen to $\sin(x)$ if the vector width was doubled?
- A1: *If we're using ISPC,* we would expect roughly 2× performance (slightly less would be realized in practice).

- Q: Can we do this forever & expect same results?
- A: No. Computing rdenom will limit gains (Amdahl's Law).

- Q: For this $\sin(x)$ program, would you prefer larger vector or more cores?
- A: Either should give speedup, but this program maps easily to SIMD, and adding vector lanes is much cheaper (area + energy) than adding cores. (Remember GPU vs CPU pictures.)

What if? #4
Benefits(?) of SMT

- Q: How should we schedule threads on a dual-core processor with SMT, running these two apps, each of which have 2 threads?
  - The $\sin(x)$ function
  - A program that is copying large amounts of data with very little computation

- (Note: There are four "cores" and four threads)

- A: We want to schedule one $\sin(x)$ thread and one memcpy() thread on each core, since SMT is most beneficial when threads use different execution units

What if? #5
Limits of speculation

- Q: What will limit the “performance” of this (silly) program on a superscalar OOO processor?

```
int foo() {
   int i = 0;
   while (i < 100000) {
       // assume single-cycle rand instruction
       if (rand() % 2 == 0) {
           i++;
       } else {
           i--;
       }
   }
}
```

- A: Unpredictable branch in if-else will cause frequent pipeline flushes

What if? #6
Benefits(?) of SMT

- Q: Would the previous program benefit from running on multiple SMT threads on a single core?

- A: Yes! Its performance is limited by the CPU frontend, which is replicated in SMT