Code Optimization II: Machine Dependent Optimizations

Topics

- Machine-Dependent Optimizations
  - Pointer code
  - Unrolling
  - Enabling instruction level parallelism
- Understanding Processor Operation
  - Translation of instructions into operations
  - Out-of-order execution of operations

Announcements

- Buffer Lab Grading this week
- Performance Lab available on moodle, due Monday Nov 17 by 8 am
  - TA's introduced the performance lab in recitation
  - In this lab, you will apply transformations to speed up an image processing loop
- Midterm #2 is probably ~Tuesday Nov 11
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 5, all sections

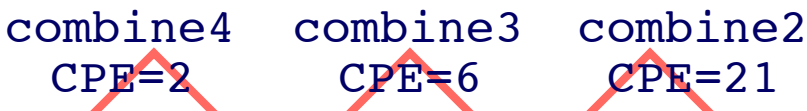
Recap of Machine-independent Optimizations…

Improving software performance

- Compilers must be cautious in their optimizations
  - Memory aliasing
  - Functional side effects
- Optimizations
  - Code motion
  - Reduction in strength
  - Share common sub-expressions
- CPE
  - Cycles/element

Optimization Example

- combine1 CPE=32

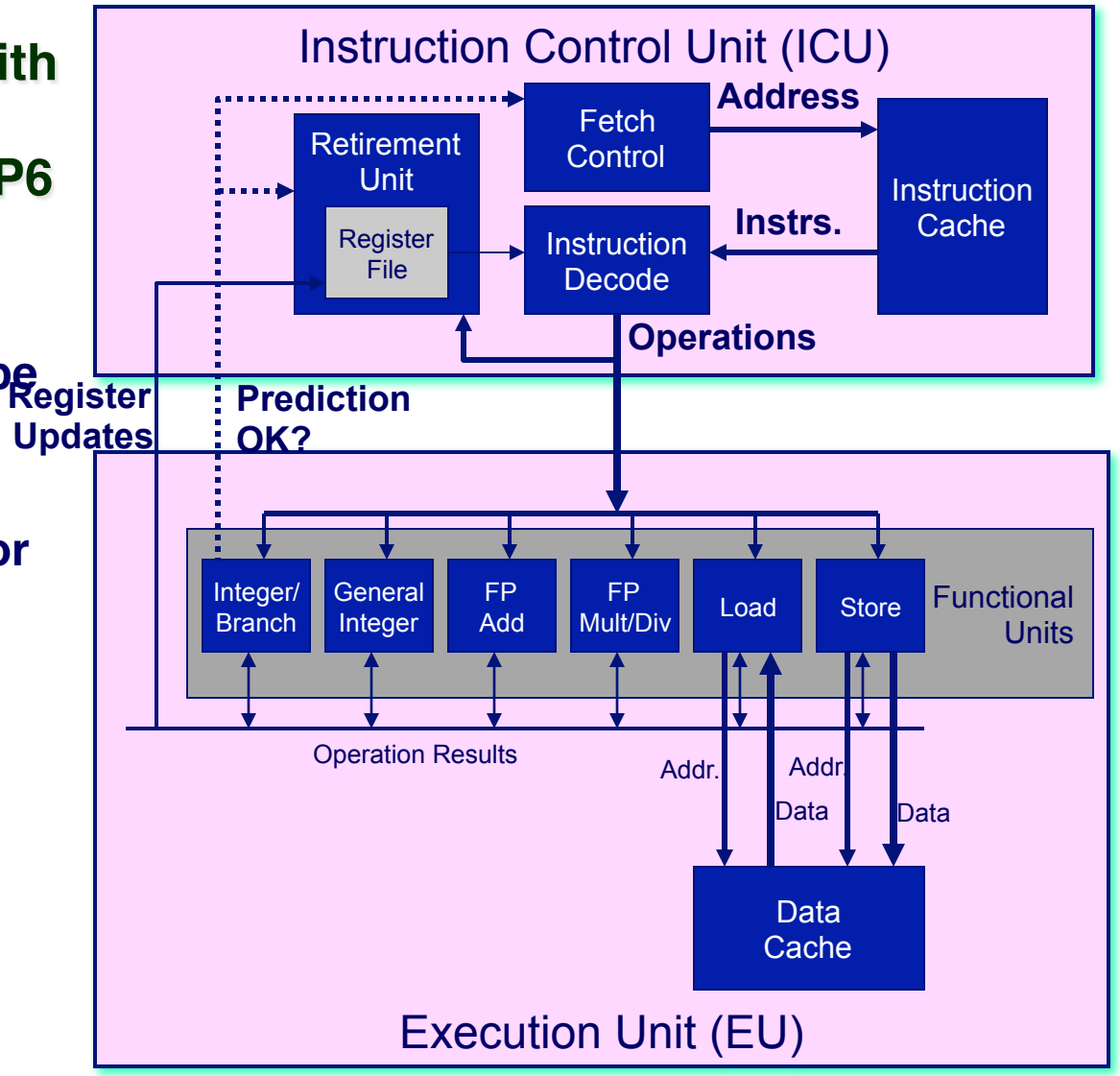



```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Modern CPU Design

Extends Y86 Design with multiple Functional Units in a Pentium P6

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

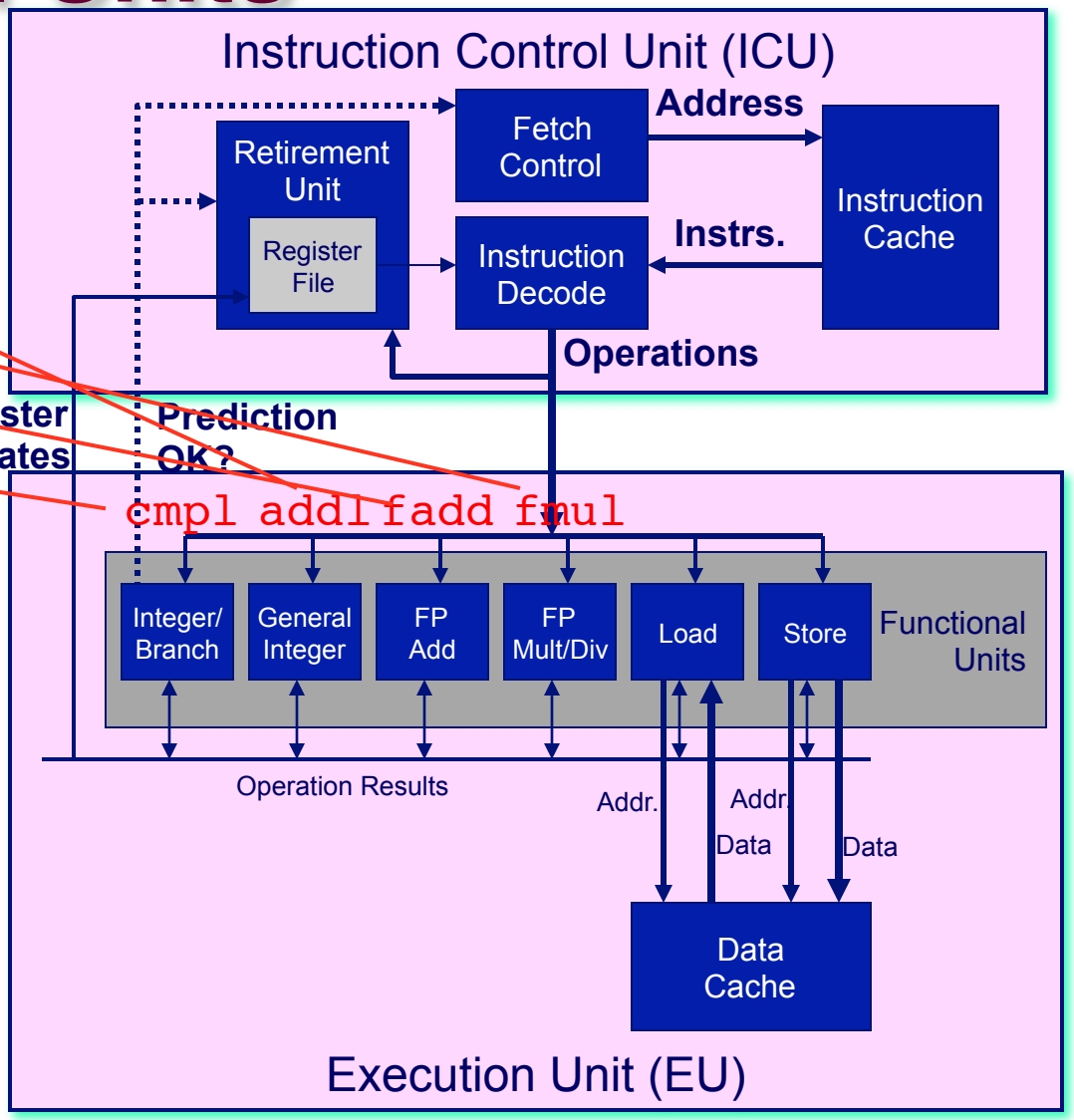


Parallel Execution across Functional Units

Code Example:

```
addl %eax, %ebx
sall $4, %esi
fmul …
fadd …
cmpl %ecx, %edx
jle …
```

- **Instructions can run at the same time in parallel on different functional units**
  - **This is called *instruction-level parallelism***
- **Instruction *re-ordering* may occur**
  - **In this example, `fmul` and `fadd` can execute before `sall` if independent**



Pipelined Execution Within Functional Units

Each Individual Functional Unit may be pipelined

Example: Integer Multiply

- Delay = 4 cycles, but
- Throughput = 1 instruction/cycle, i.e. can issue a new instruction to it every 1 cycle, i.e. Issue time = 1 cycle
  - This is because of multiple stages of pipelining internal to the Integer Multiply unit

Pipelined Execution Within Functional Units

Capabilities of a Pentium P6 CPU:

Instruction	Latency	Cycles/Issue
Integer Add	1	0.5
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

Pipelined Execution Within Functional Units

Capabilities of an Intel Core i7:

	Integer		Single-precision		Double-precision	
Operation	Latency	Issue	Latency	Issue	Latency	Issue
Addition	1	0.33	3	1	3	1
Multiplication	3	1	4	1	5	1
Division	11–21	5–13	10–15	6–11	10–23	6–19

Figure 5.12 **Latency and issue time characteristics of Intel Core i7 arithmetic operations.** Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two operations. The times for division depend on the data values.

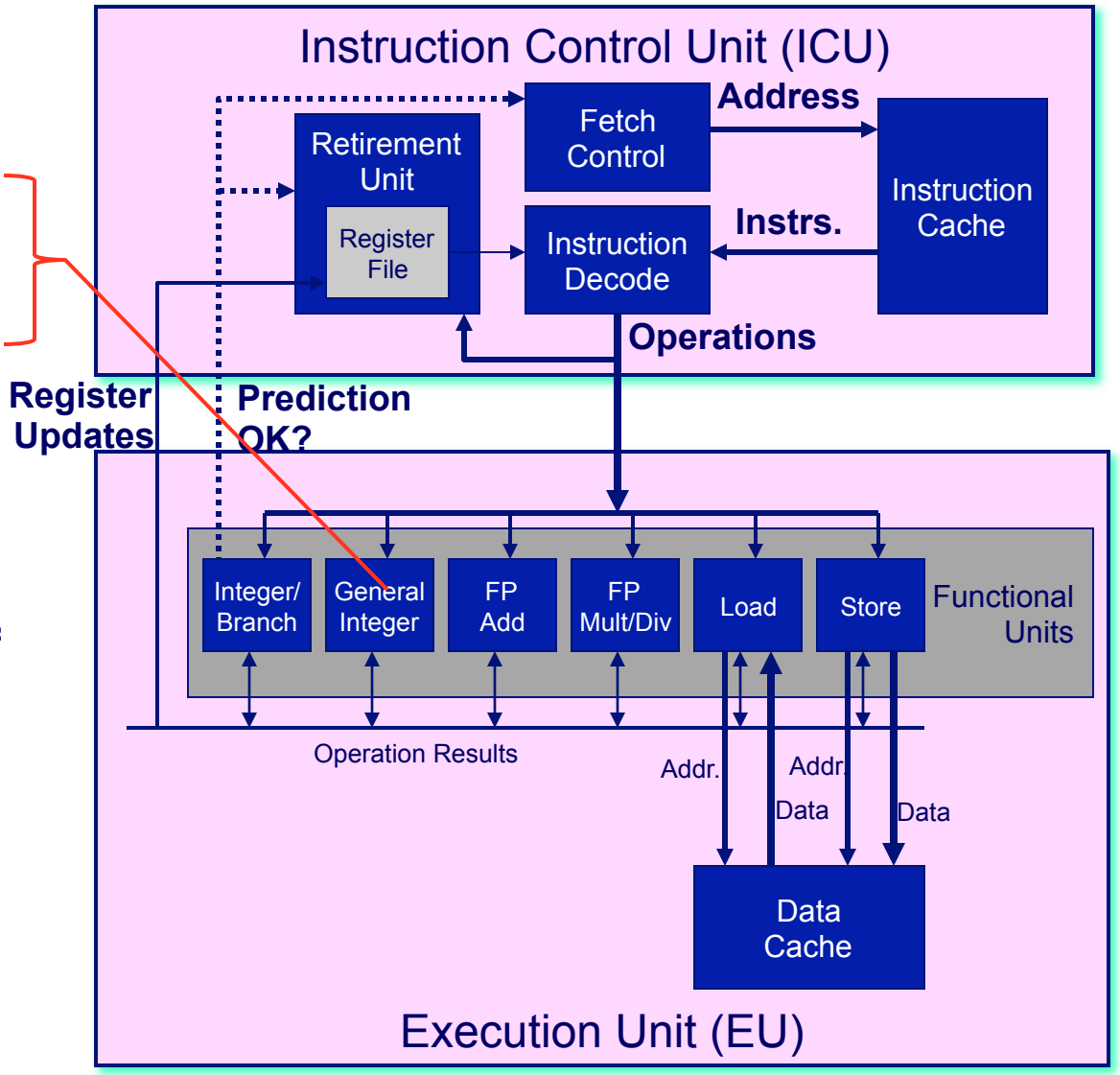
Pipelined Execution

Code Example:

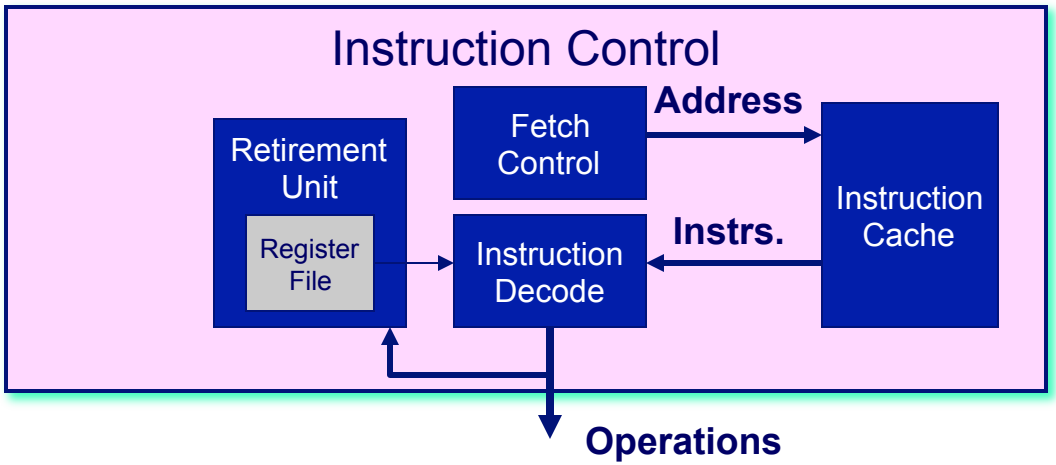
```
...
imull %eax, %ebx
immul %ecx, %edx
immul %edi, %esi
...
```

- **All 3 instructions can be pipelined in an integer multiply at different stages of execution**

Generally, can mix both parallel and pipelined execution inside CPU



Instruction Control



Grabs Instruction Bytes From Memory

- Based on current PC + predicted targets for predicted branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

Translates Instructions Into *Operations*

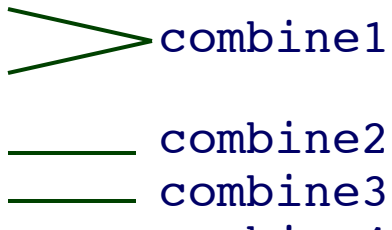
- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*

- Abstract identifier linking destination of one operation with sources of later operations

Previous Best Combining Code

Method	Integer +
Abstract -g	42.06
Abstract -O2	31.25
Move vec_length	20.66
Move function call	6.00
Accum. in temp	2.00




```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

Translation Example

Version of Combine4

- Integer data, add operation

```
.L24:                                  # Loop:
  addl (%eax,%edx,4),%ecx              # sum += data[i]
  incl %edx                            # i++
  cmpl %esi,%edx                       # i:length
  jl .L24                              # if < goto Loop
```

Translation of First Iteration

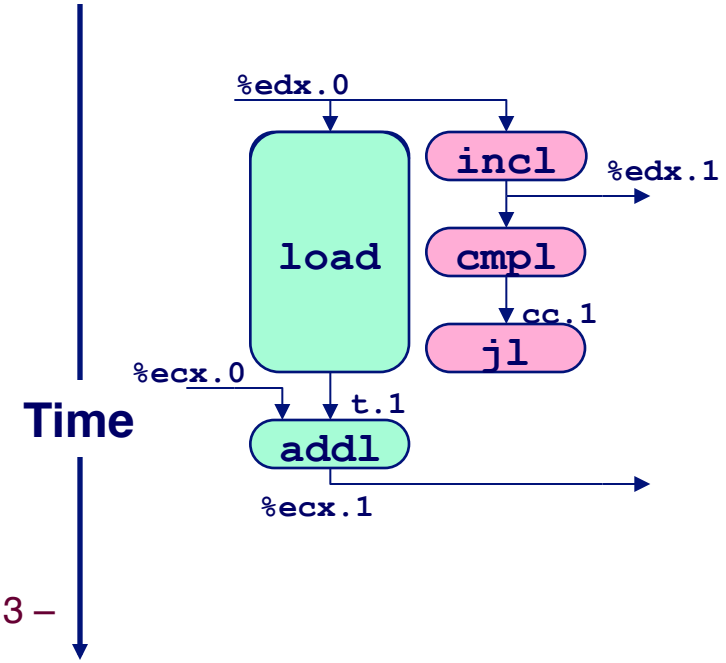
```
.L24:

  addl (%eax,%edx,4),%ecx
  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4)  ➔ t.1
iaddl t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl-taken cc.1
```

- ICU fetches `add` instruction & translates it to 2 operations for functional units, 1 for the load unit, 1 for an integer unit

Visualizing Operations

```
load (%eax,%edx.0,4) ➔ t.1
iaddl t.1, %ecx.0    ➔ %ecx.1
incl %edx.0          ➔ %edx.1
cmpl %esi, %edx.1    ➔ cc.1
jl-taken cc.1
```



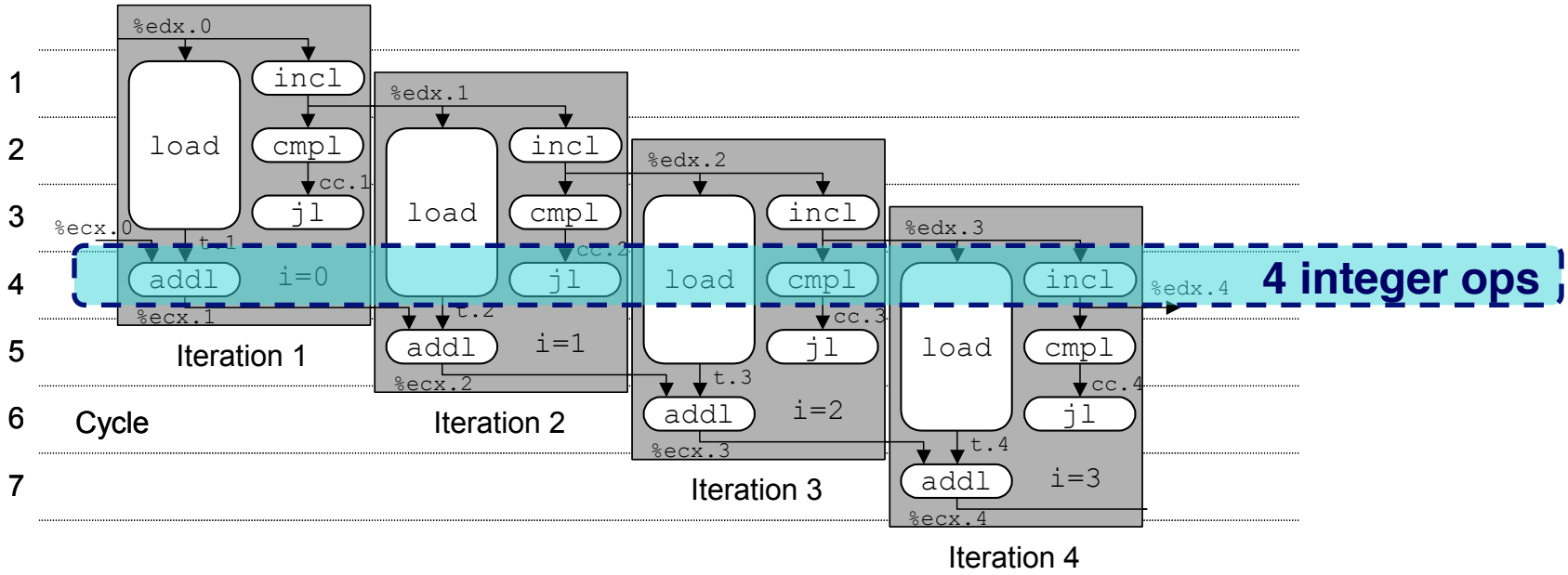
Operations

- Note `add` depends on `load` completing first, but `inc`, `cmp` and `jl` are independent, so can execute in parallel!
- We see *register renaming* inside the functional units, e.g. %edx.1 (see text)
  - Many instructions are spread over different functional units and are operating on different versions of registers

Data-flow graph

- `load`'s latency = 3 clock cycles, but can be pipelined
- `add`'s latency = 1 clock cycle

4 Iterations of Combining Sum

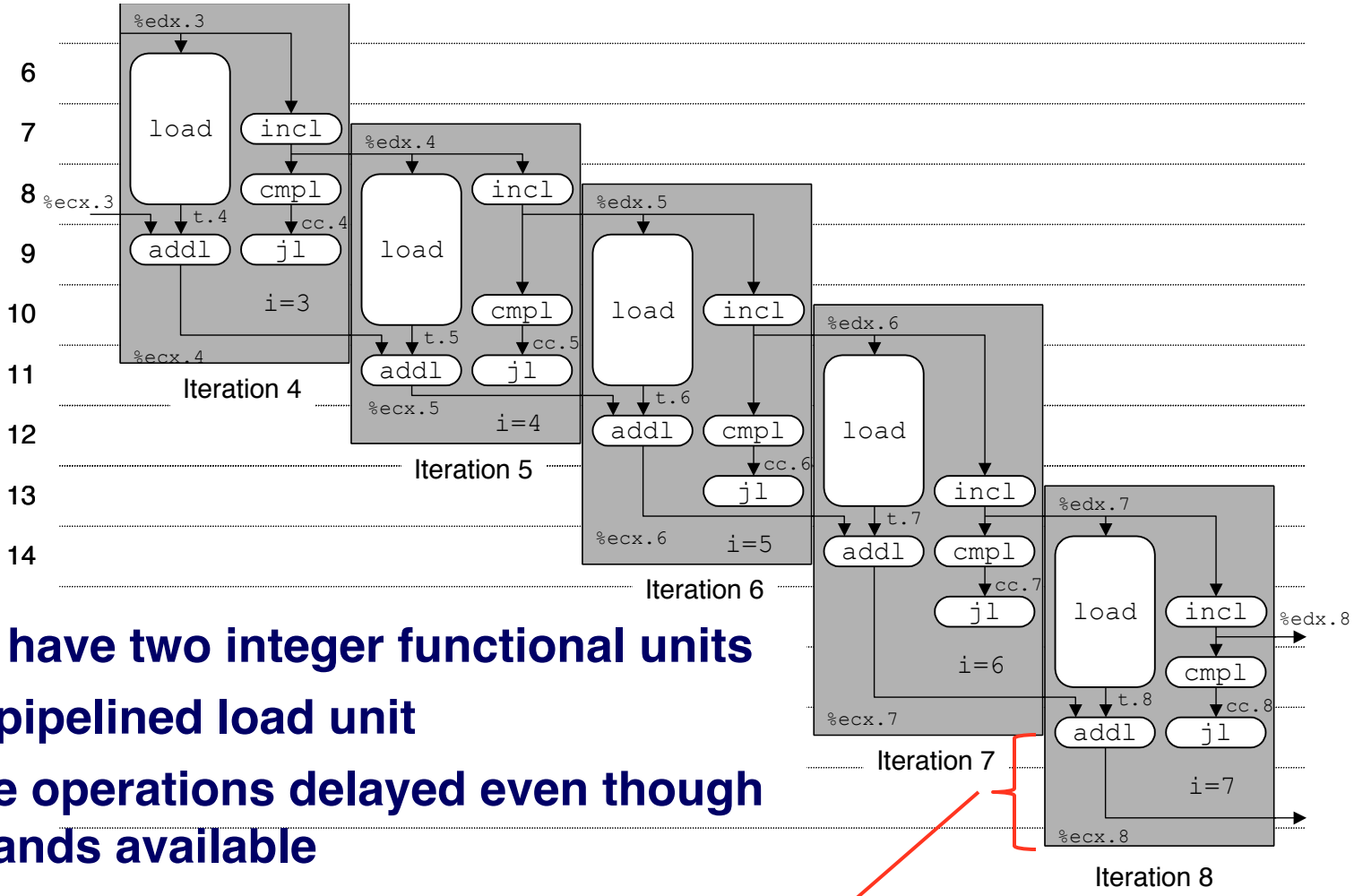


Unlimited Resource Analysis, e.g. at least 4 loads & 4 integer units

Performance

- Can begin a new iteration on each clock cycle
- Should give CPE of 1.0
- Would require executing 4 integer operations in parallel

Combining Sum: Resource Constraints



- Only have two integer functional units
- One pipelined load unit
- Some operations delayed even though operands available
- Set priority based on program order

Performance

- Delay = 4-5 cycles, but CPE = 2.0

Translation Example

Version of Combine4

- Integer data, multiply operation

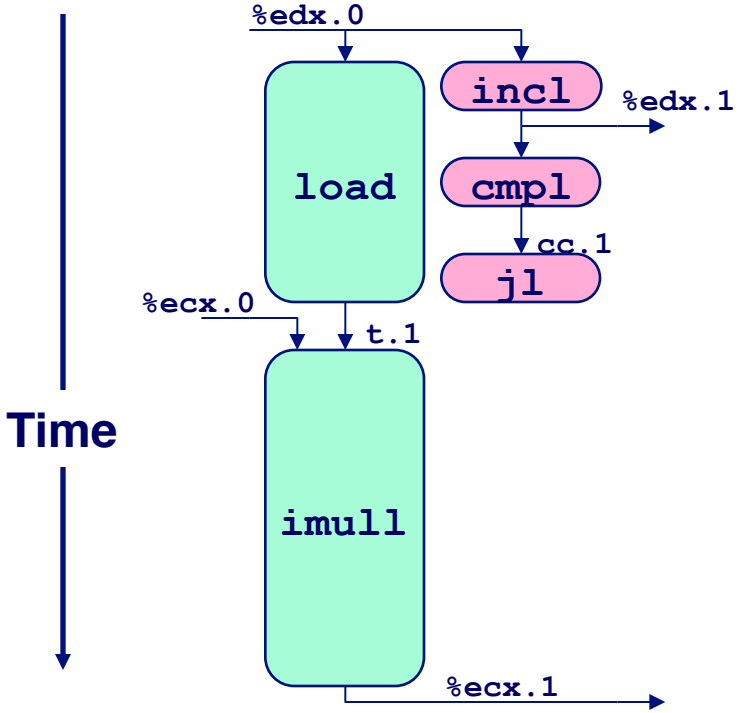
```
.L24:                              # Loop:
  imull (%eax,%edx,4),%ecx         # t *= data[i]
  incl %edx                        # i++
  cmpl %esi,%edx                   # i:length
  jl .L24                          # if < goto Loop
```

Translation of First Iteration

```
.L24:

  imull (%eax,%edx,4),%ecx
  incl %edx
  cmpl %esi,%edx
  jl .L24
```

```
load (%eax,%edx.0,4)  ➔ t.1
imull t.1, %ecx.0     ➔ %ecx.1
incl %edx.0           ➔ %edx.1
cmpl %esi, %edx.1     ➔ cc.1
jl-taken cc.1
```

Visualizing Operations




```
load (%eax,%edx,4)  ➔ t.1
imull t.1, %ecx.0   ➔ %ecx.1
incl %edx.0         ➔ %edx.1
cmpl %esi, %edx.1   ➔ cc.1
jl-taken cc.1
```

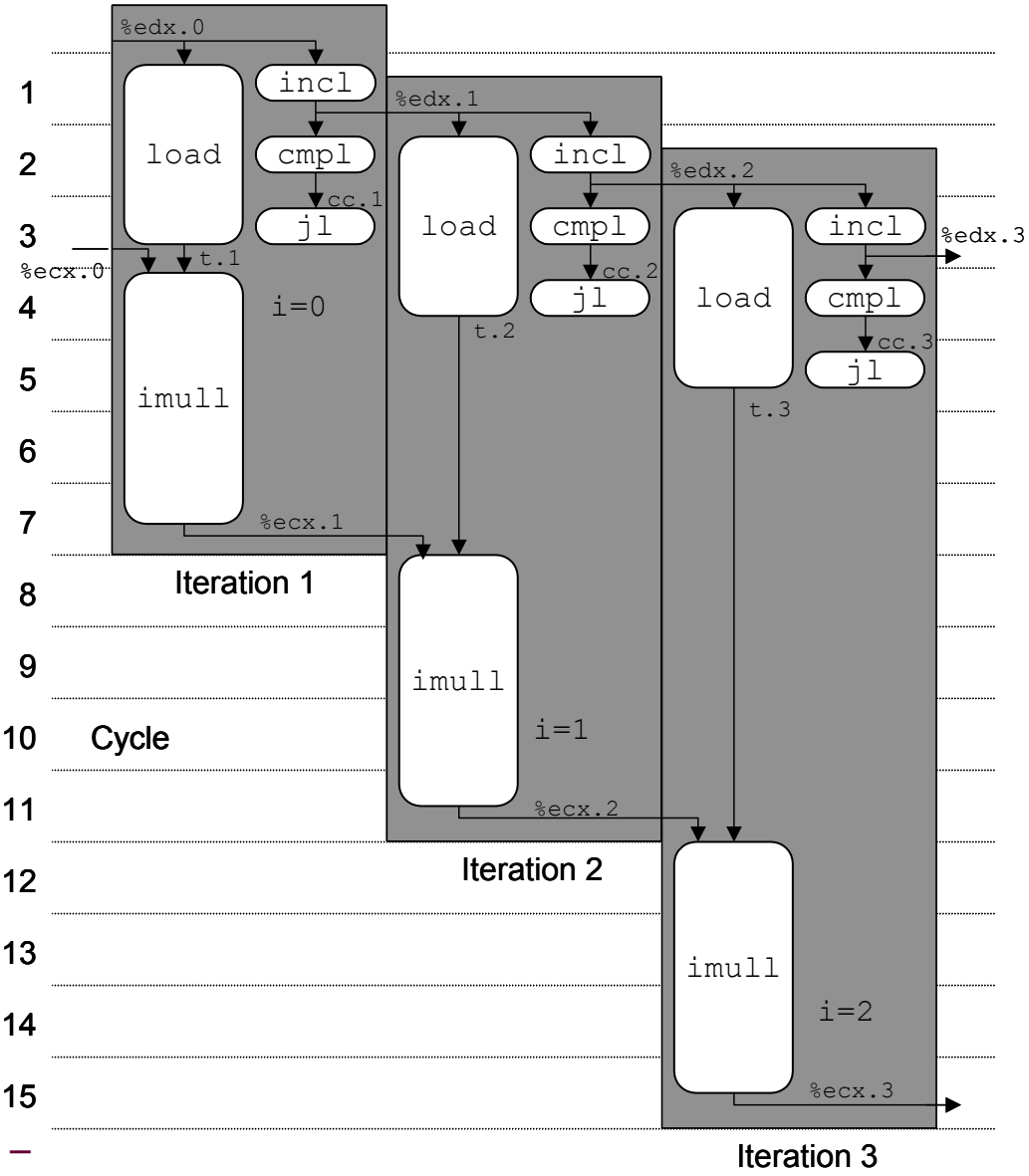
Operations

- Vertical position denotes time at which executed
  - Cannot begin operation until operands available
- Height denotes latency, so imull takes 4 clock cycles

Operands

- Arcs shown only for operands that are passed within execution unit

3 Iterations of Combining Product



Unlimited Resource Analysis

Performance

- Can't use imull pipeline due to data dependency
- Limiting factor becomes latency of integer multiplier
- Gives CPE of 4.0
- Even with infinite # of integer units, cannot do better than this

Loop Unrolling

```
void combine5(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-2;
  int *data = get_vec_start(v);
  int sum = 0;
  int i;
  /* Combine 3 elements at a time */
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    sum += data[i];
  }
  *dest = sum;
}
```

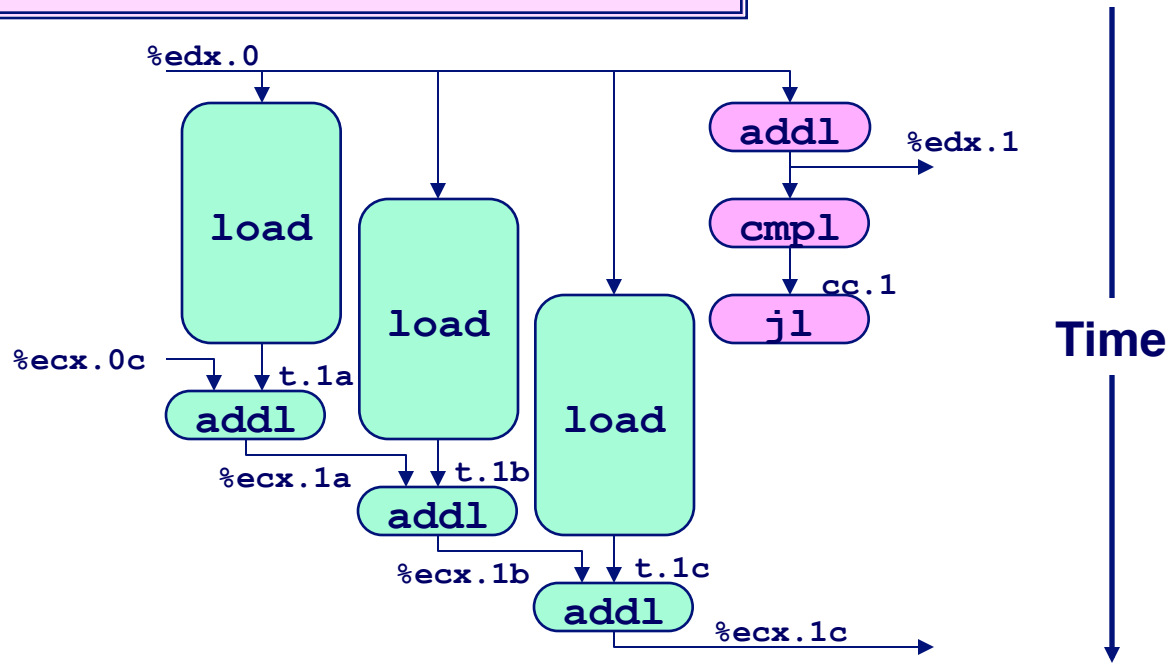
Optimization

- Combine multiple iterations into single loop body
- Amortizes loop overhead across multiple iterations
- Finish extras at end
- Measured CPE = 1.33
- This is a machine-dependent optimization

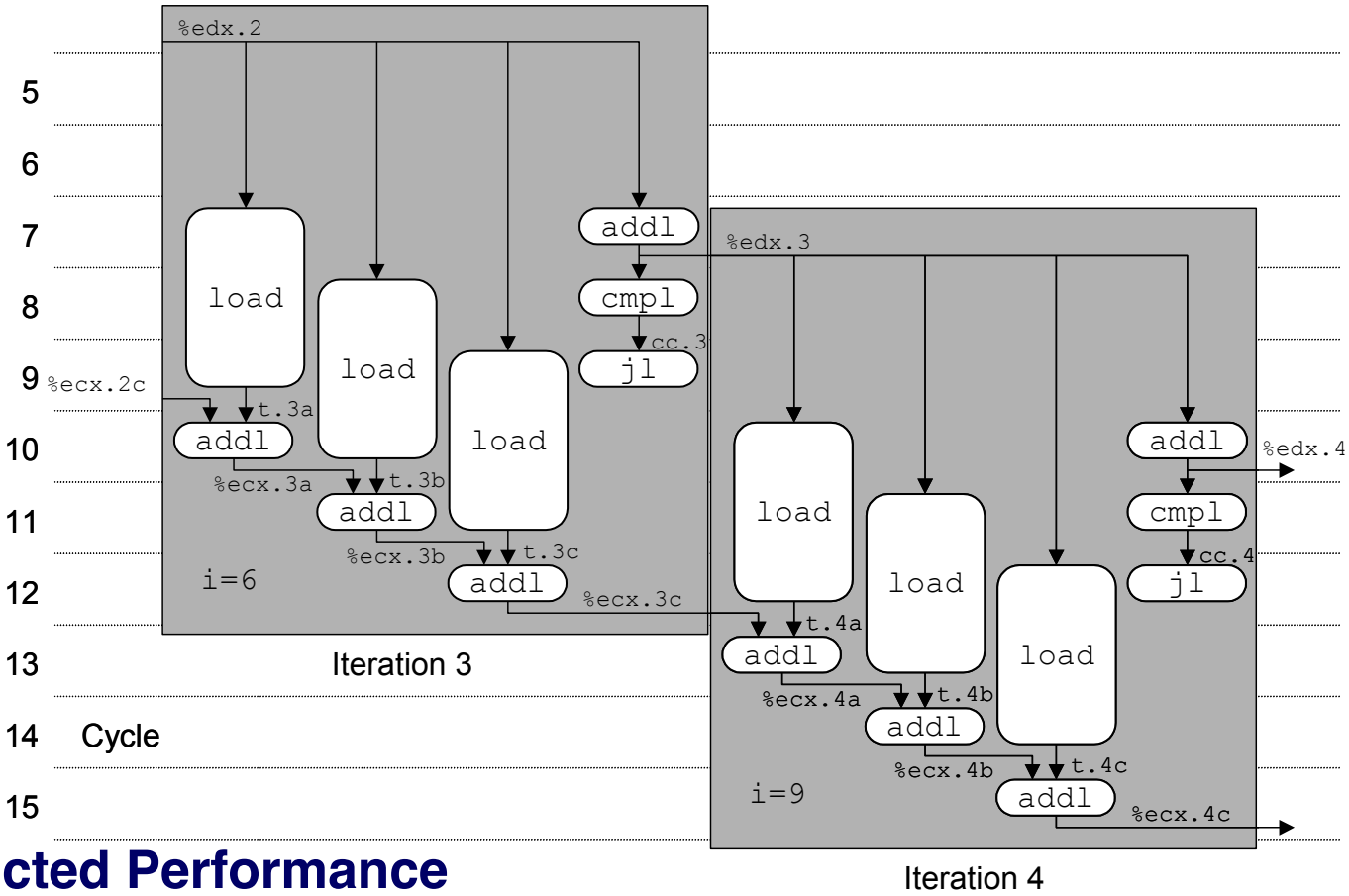
Visualizing Unrolled Loop

```
load (%eax,%edx.0,4)    ➔ t.1a
iaddl t.1a, %ecx.0c     ➔ %ecx.1a
load 4(%eax,%edx.0,4)   ➔ t.1b
iaddl t.1b, %ecx.1a     ➔ %ecx.1b
load 8(%eax,%edx.0,4)   ➔ t.1c
iaddl t.1c, %ecx.1b     ➔ %ecx.1c

iaddl $3,%edx.0         ➔ %edx.1
cmpl %esi, %edx.1       ➔ cc.1
jl-taken cc.1
```

- Loads can pipeline, since don't have dependencies
- Only one set of loop control operations



Executing with Loop Unrolling



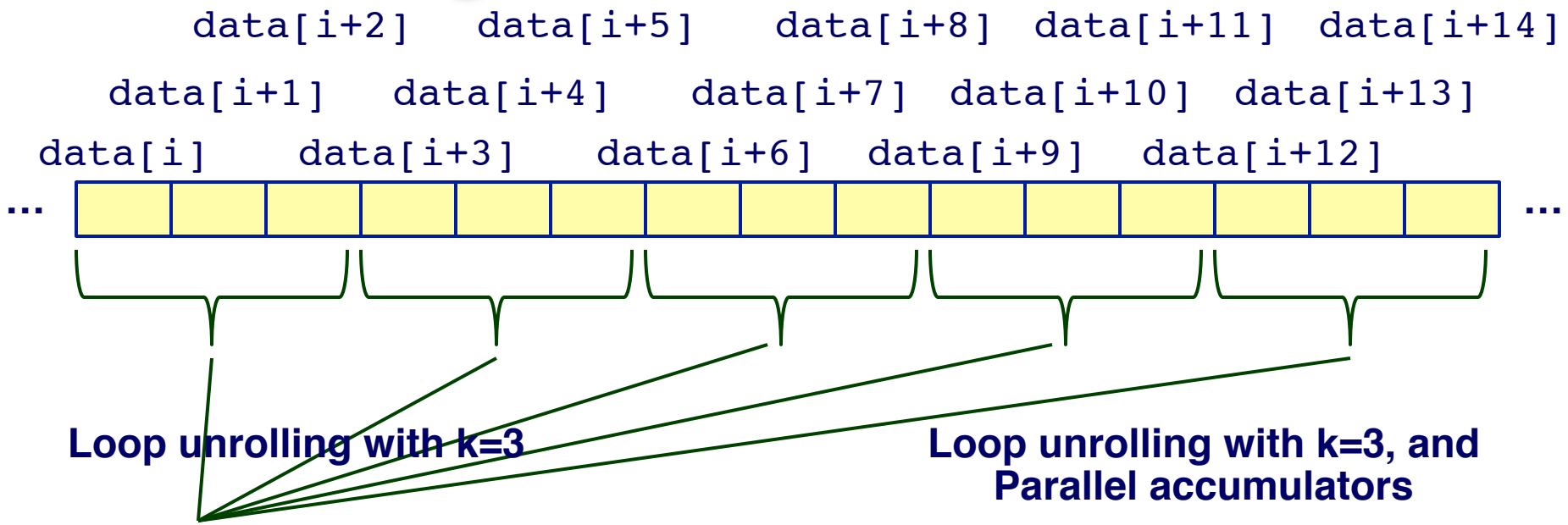
- **Predicted Performance**
  - **CPE = 1.0. One iteration in 3 cycles, and 3 elements per iteration**
  - **Loop unrolling allowed add/cmp/jl to be shifted in time so as not to conflict with the add after the load, (only 2 int units)**
- **Measured Performance**
  - **CPE of 1.33, One iteration every 4 cycles**

Effect of Unrolling

Unrolling Degree		1	2	3	4	8	16
Integer	Sum	2.00	1.50	1.33	1.50	1.25	1.06
Integer	Product	4.00					
FP	Sum	3.00					
FP	Product	5.00					

- Only helps integer sum for our examples
  - Other cases constrained by functional unit latencies
- Effect is nonlinear with degree of unrolling
  - Many subtle effects determine exact scheduling of operations

Visualizing Parallelism



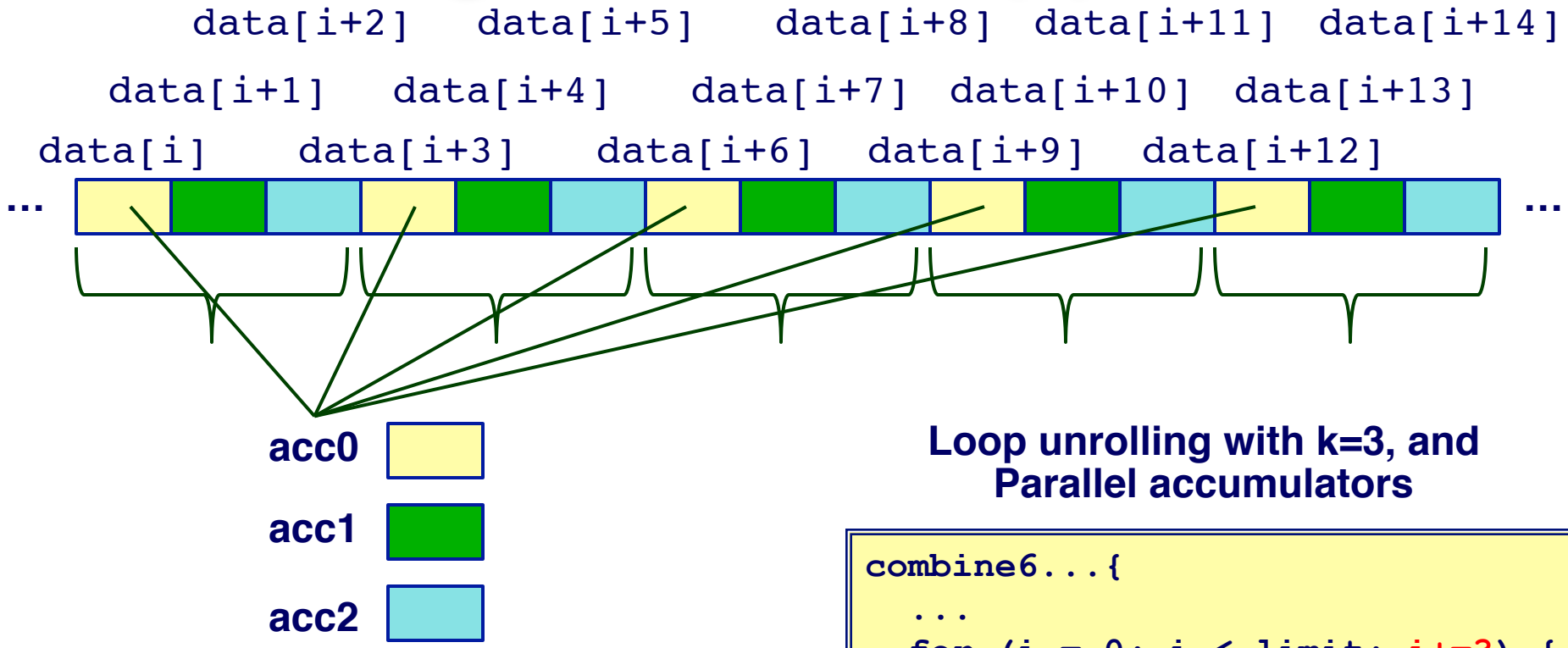
Loop unrolling with k=3

Loop unrolling with k=3, and Parallel accumulators

```
combine5...{
  ...
  for (i = 0; i < limit; i+=3) {
    sum += data[i] + data[i+1]
           + data[i+2];
  }
  ...
}
```

```
combine6...{
  ...
  for (i = 0; i < limit; i+=3) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
    acc2 = acc2 + data[i+2];
  }
  sum = acc0 + acc1 + acc2;
  ...
}
```

Visualizing Parallelism (2)



Loop unrolling with k=3, and Parallel accumulators

```
combine6...{
  ...
  for (i = 0; i < limit; i+=3) {
    acc0 = acc0 + data[i];
    acc1 = acc1 + data[i+1];
    acc2 = acc2 + data[i+2];
  }
  sum = acc0 + acc1 + acc2;
  ...
}
```

- Each accumulator is independent of the other accumulators, so summing can occur in parallel
- Sum together accumulators at the last step

Parallel Loop Unrolling

```
void combine6(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x0 = 1;
  int x1 = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x0 *= data[i];
  }
  *dest = x0 * x1;
}
```

Code Version

- Integer product

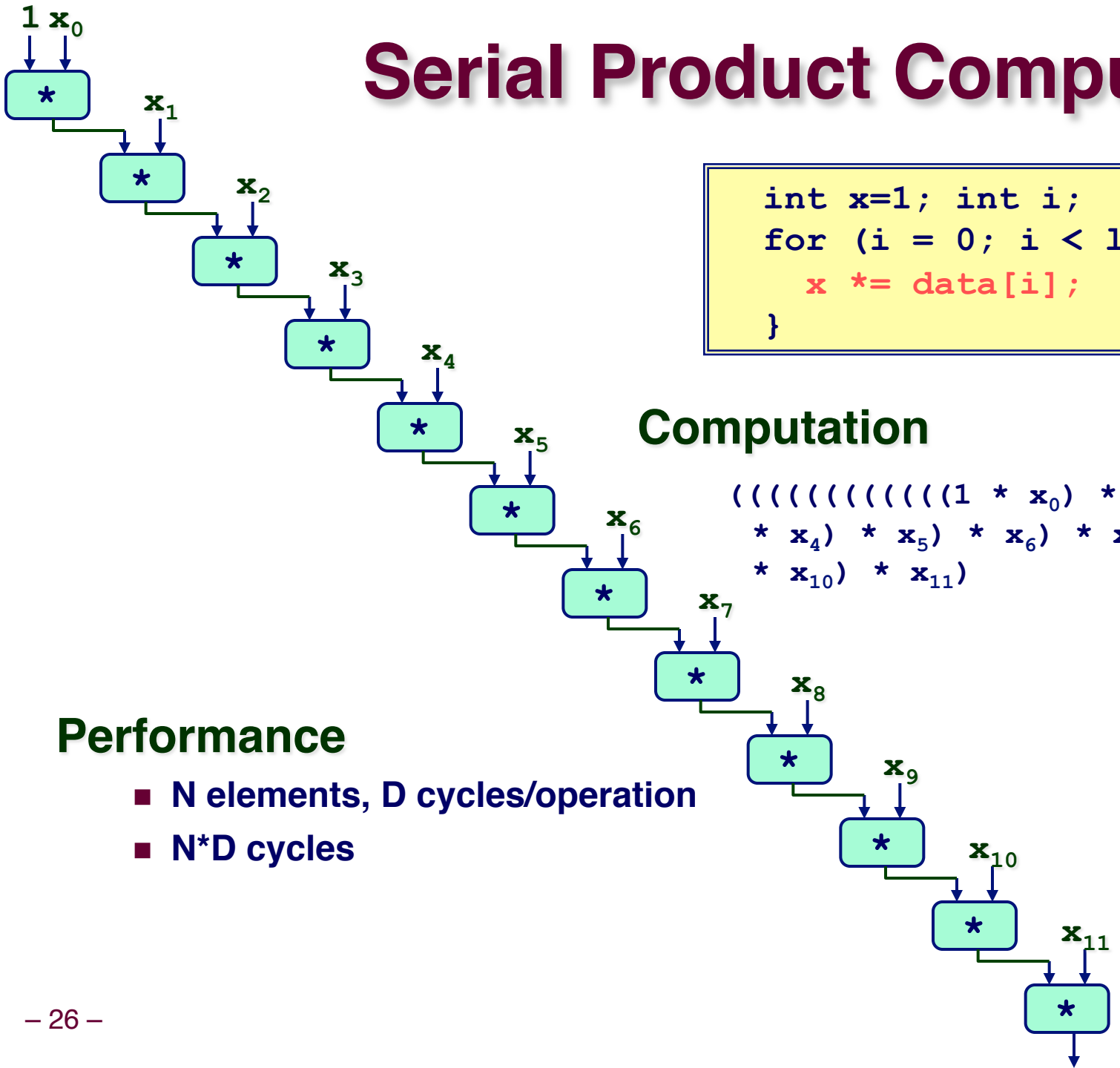
Optimization

- Accumulate in two different products
  - Can be performed simultaneously
- Combine at end

Performance

- CPE = 2.0
- 2X performance

Serial Product Computation




```
int x=1; int i;
for (i = 0; i < limit; i++) {
  x *= data[i];
}
```

Computation

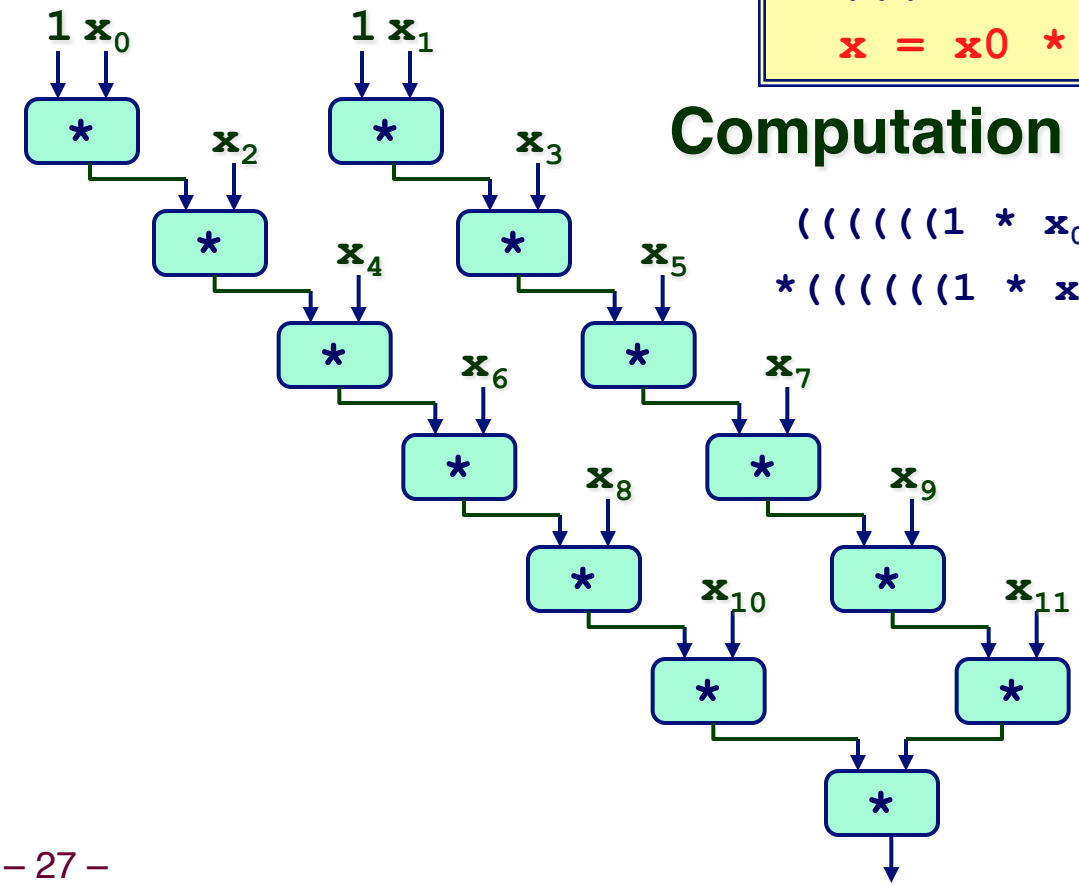
$$((((((((((((1 * x_0) * x_1) * x_2) * x_3) * x_4) * x_5) * x_6) * x_7) * x_8) * x_9) * x_{10}) * x_{11})$$

Performance

- N elements, D cycles/operation
- N*D cycles

Dual Parallel Product Computation

```
int x0 = 1; int x1 = 1; int i,x;
for (i = 0; i < limit; i+=2) {
  x0 *= data[i];
  x1 *= data[i+1];
}
...
x = x0 * x1;
```



Computation

$$((((((1 * x_0) * x_2) * x_4) * x_6) * x_8) * x_{10})$$
$$*((((((1 * x_1) * x_3) * x_5) * x_7) * x_9) * x_{11})$$

Performance

- N elements, D cycles/ operation
- (N/2+1)*D cycles
- ~2X performance improvement

Requirements for Parallel Computation

Mathematical

- Combining operation must be associative & commutative
  - OK for integer multiplication and addition
  - Not strictly true for floating point
    - » We saw some pathological cases in the textbook where the order of floating point multiplication can cause overflow
      - e.g. in Ch 2, for single precision floating point, "(1e20*1e20)*1e-20 evaluates to +∞, while 1e20*(1e20*1e-20) evaluates to 1e20."
    - » OK for most applications

Hardware

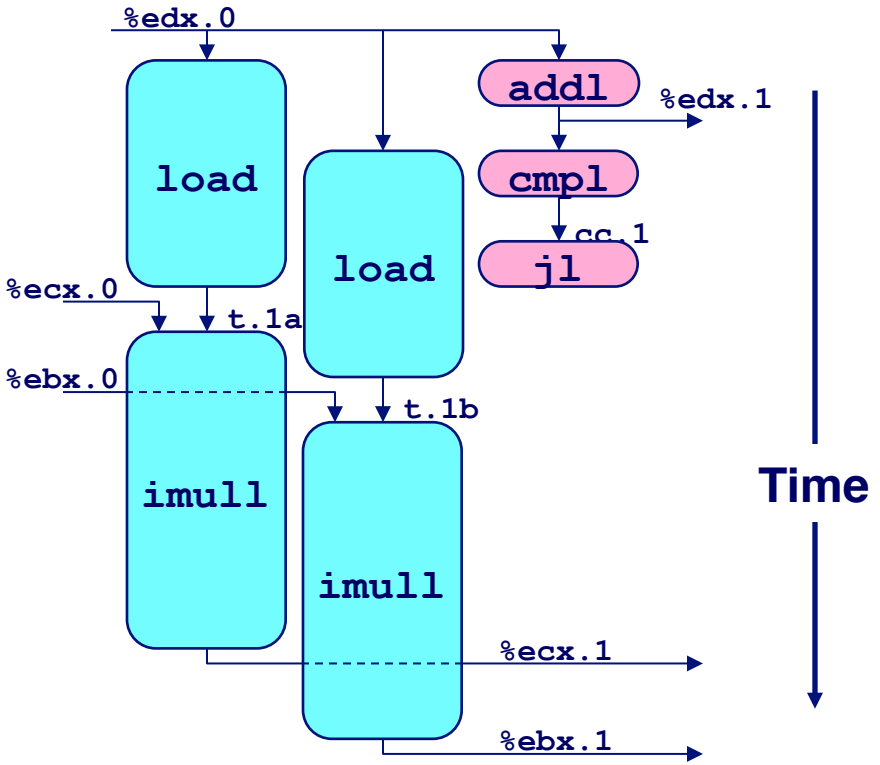
- Pipelined functional units
- Ability to dynamically extract parallelism from code

Visualizing Parallel Loop

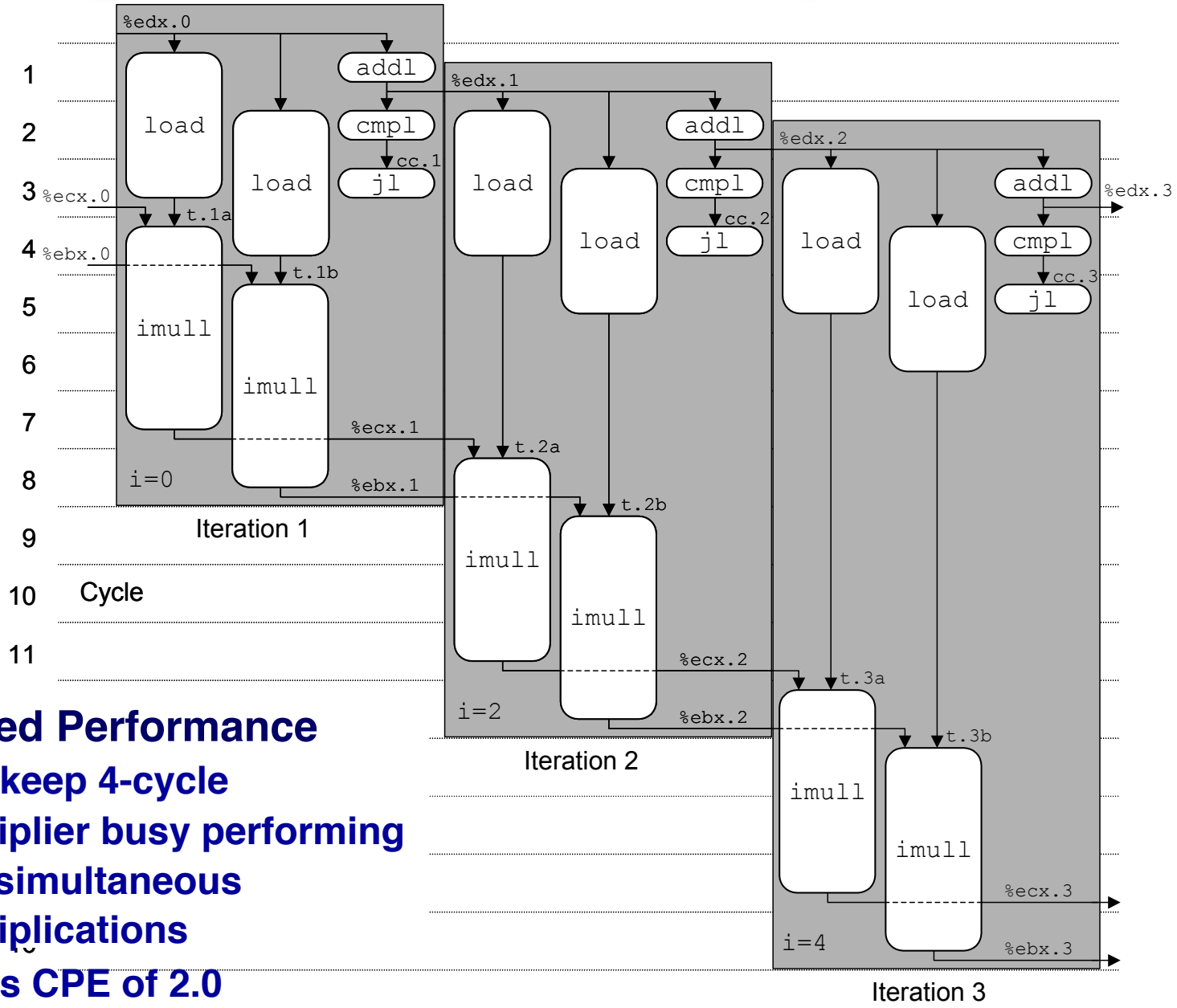
```
load (%eax,%edx.0,4)   → t.1a
imull t.1a, %ecx.0     → %ecx.1
load 4(%eax,%edx.0,4)  → t.1b
imull t.1b, %ebx.0     → %ebx.1

iaddl $2,%edx.0        → %edx.1
cmpl %esi, %edx.1      → cc.1
jl-taken cc.1
```

```
  ...
  for (i = 0; i < limit;
i+=2) {
    x0 *= data[i];
    x1 *= data[i+1];
  }
  ...
  *dest = x0 * x1;
```

- Two multiplies within loop no longer have data dependency
- Allows them to pipeline (issue time of integer multiply is 1 clock cycle)



Executing with Parallel Loop



- **Predicted Performance**
  - **Can keep 4-cycle multiplier busy performing two simultaneous multiplications**
  - **Gives CPE of 2.0**

Optimization Results for Combining

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
Move function call	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Pointer	3.00	4.00	3.00	5.00
Unroll 4	1.50	4.00	3.00	5.00
Unroll 16	*1.06*	4.00	3.00	5.00
Unroll 2 X Parallel 2	1.50	2.00	2.00	2.50
4 X 4	1.50	*2.00*	*1.50*	2.50
8 X 4	1.25	*1.25*	1.50	*2.00*
Theoretical Opt.	1.00	1.00	1.00	2.00
Worst : Best	*39.7*	*33.5*	*27.6*	*80.0*

Parallel Unrolling: Reassociation

```
void combine6aa(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int limit = length-1;
  int *data = get_vec_start(v);
  int x = 1;
  int i;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2) {
    x *= (data[i] * data[i+1]);
  }
  /* Finish any remaining elements */
  for (; i < length; i++) {
    x *= data[i];
  }
  *dest = x;
}
```

Code Version

- Integer product

Optimization

- Multiply pairs of elements together
- And then update product
- "Tree height reduction"

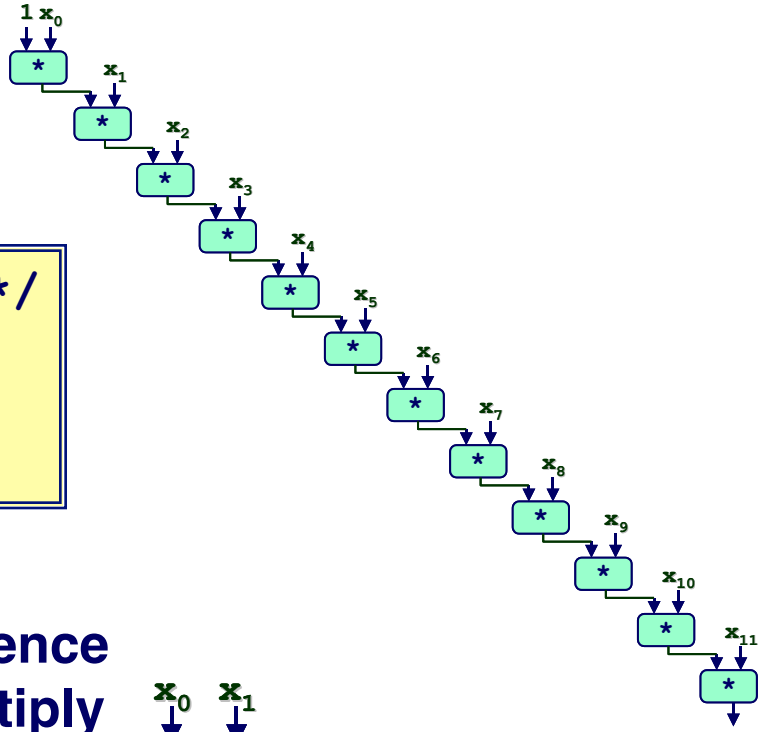
Performance

- CPE = 2.5

Understanding Parallelism

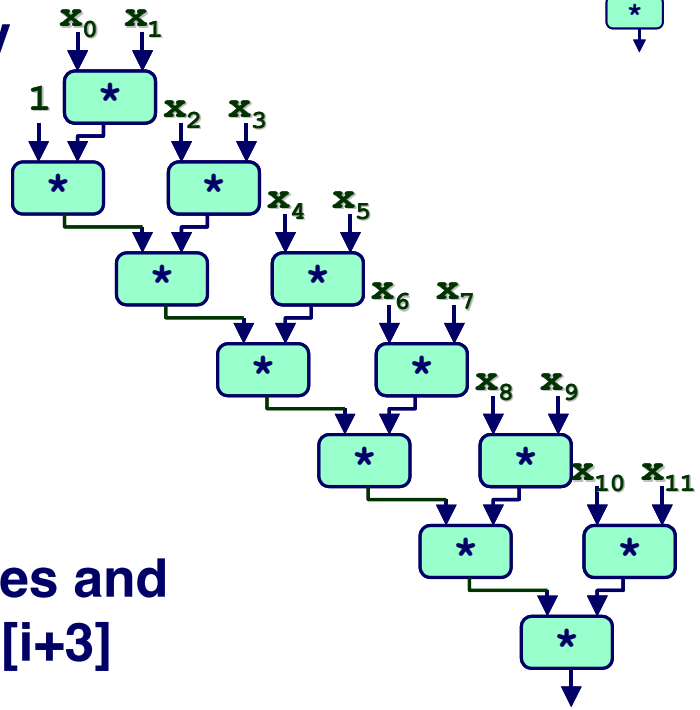
```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
  x = (x * data[i]) * data[i+1];
}
```

- CPE = 4.00
- All multiplies performed in sequence and depend on the previous multiply




```
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
  x = x * (data[i] * data[i+1]);
}
```

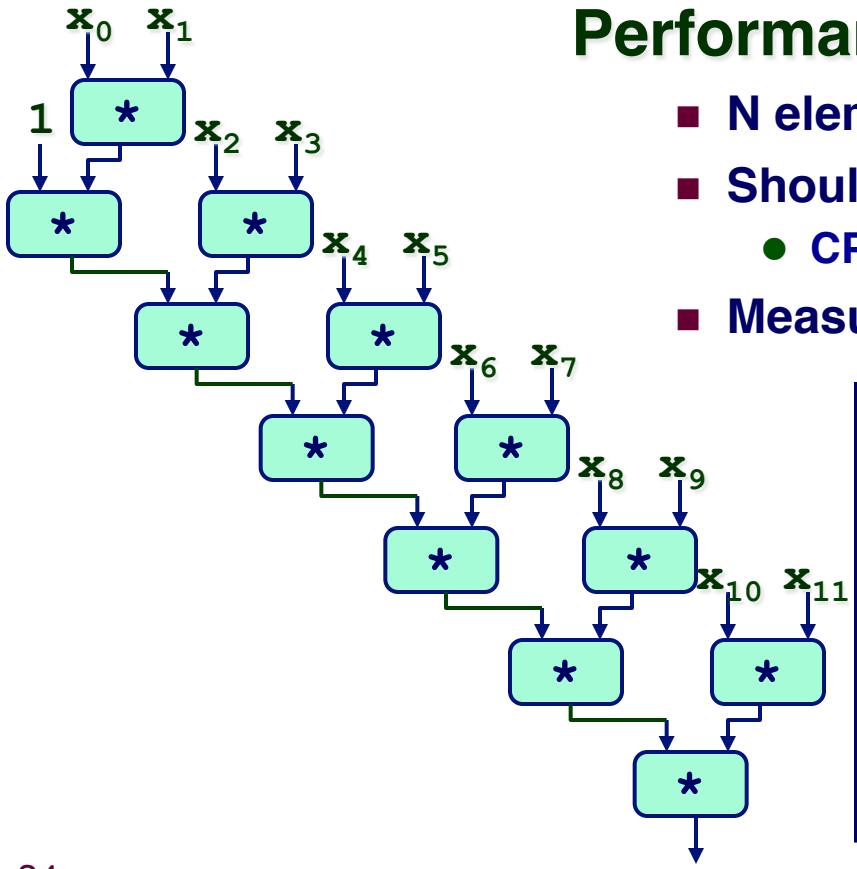
- CPE = 2.50
- Multiplies overlap: data[i]*data[i+1] independent of accumulator multiplies and next iteration multiply data[i+2]*data[i+3]



Reassociation (Method #2) Computation

Computation

$$((((((1 * (x_0 * x_1)) * (x_2 * x_3)) * (x_4 * x_5)) * (x_6 * x_7)) * (x_8 * x_9)) * (x_{10} * x_{11}))$$



Performance

- N elements, D cycles/operation
- Should be (N/2+1)*D cycles
  - CPE = 2.0
- Measured CPE worse

Unrolling	CPE (measured)	CPE (theoretical)
2	2.50	2.00
3	1.67	1.33
4	1.50	1.00
6	1.78	1.00

Summary: Results for Pentium III

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
Move function call	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00
Unroll 4	1.50	4.00	3.00	5.00
Unroll 16	1.06	4.00	3.00	5.00
4 X 2	1.50	2.00	*1.50*	2.50
8 X 4	*1.25*	*1.25*	1.50	*2.00*
8 X 8	1.88	1.88	1.75	2.00
Worst : Best	*39.7*	*33.5*	*27.6*	*80.0*

- Biggest gain doing basic optimizations
- But, last little bit helps