[Chap.5-2] Optimizing Program Performance

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

SKK UNIVERSITY

Contents

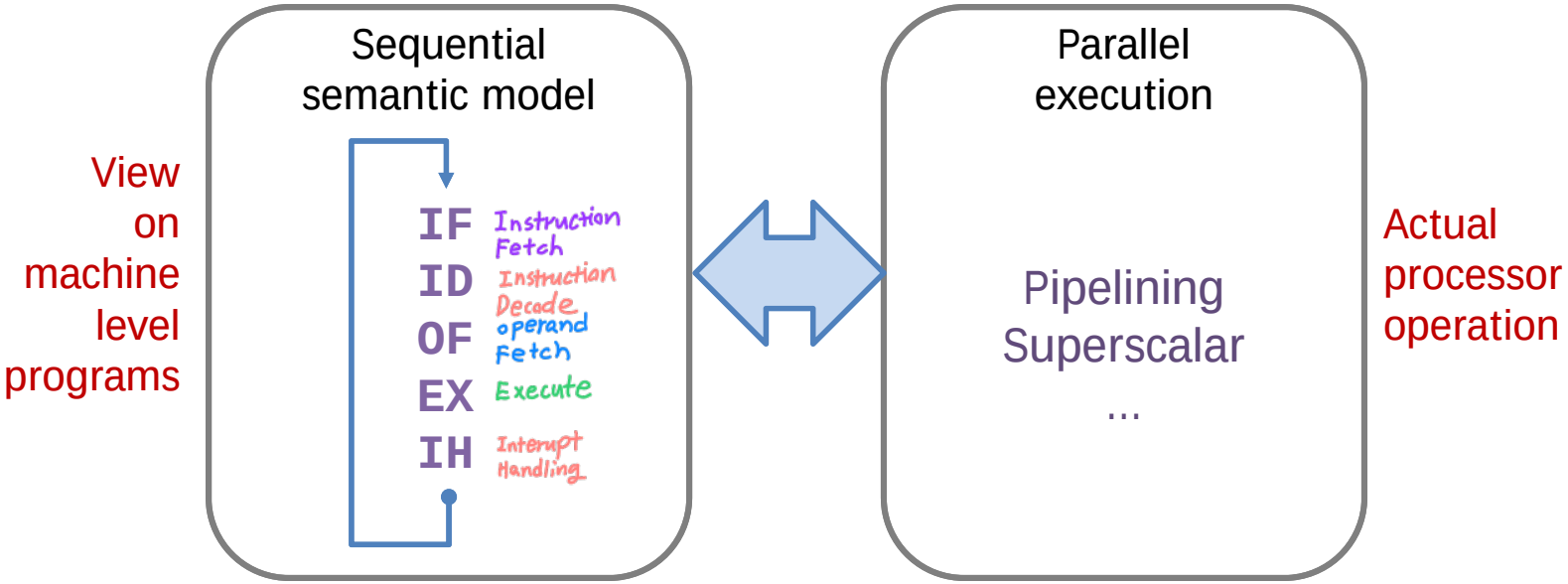
- Introduction
- Optimizing compilers
- Expressing program performance
- Benchmark example
- Loop optimization
- Reducing procedure calls
- Reducing memory references
- Understanding modern processors
- Loop unrolling
- Enhancing parallelism
- …

Understanding Modern P's

- **Further optimizations**
  - Exploiting the microarchitecture of the processor
  - Focusing on general principles of optimization that can be applied on a wide variety of machines (may not be able to applied on some machines)

Understanding Modern P's

- **Modern processors**
  - Have complex hardware that attempts to maximize program performance
    - Instruction-level parallelism
      - ✓ 100+ instructions "in flight"
        (100+instructions being executed in parallel)



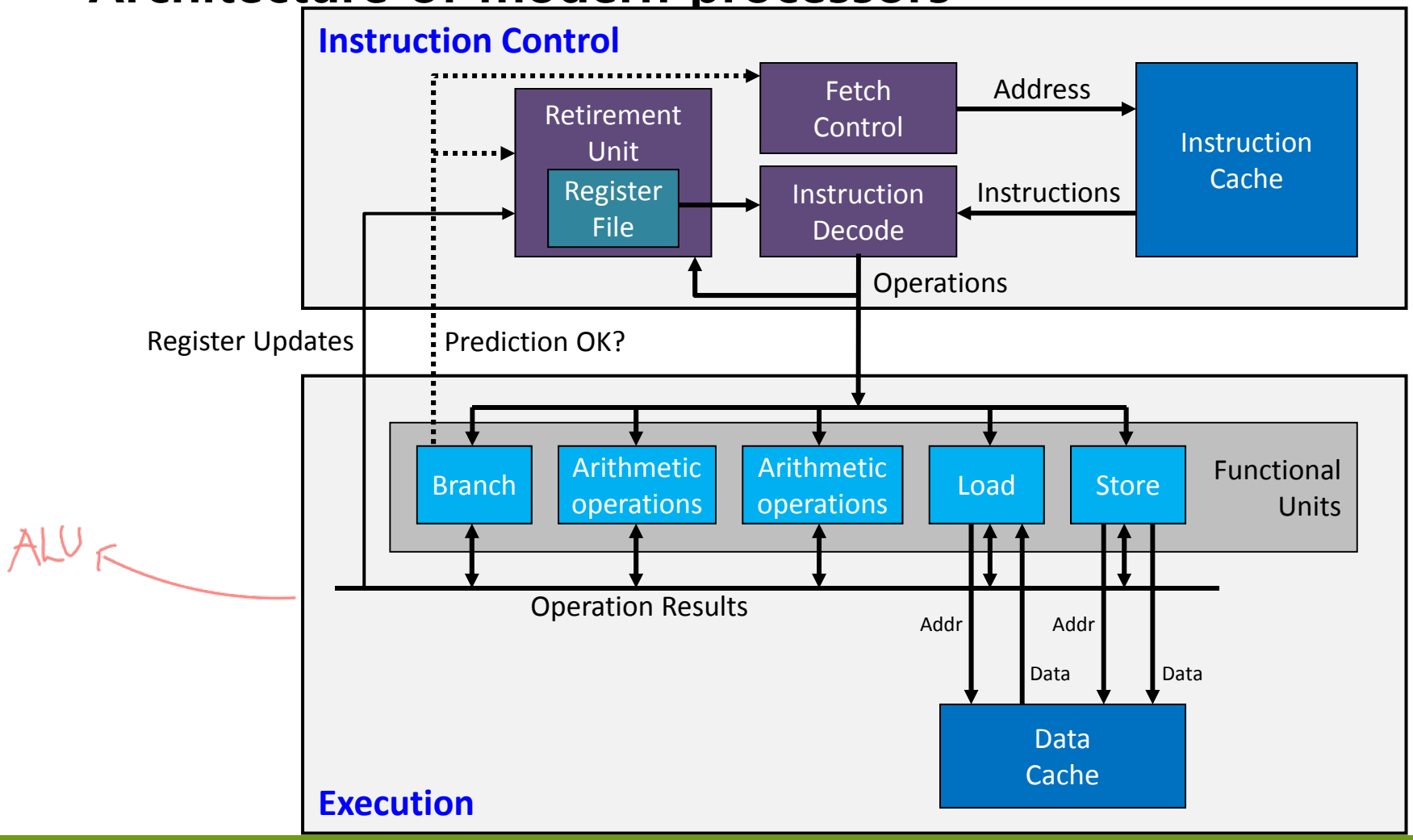
Understanding Modern P's

Modern processors

- Two lower bounds
  that characterize the maximum program performance
  - Latency bound
    - ✓ Encountered when a series of operations must be performed in strict sequence
    - ✓ Limits the performance of a program when data dependencies exist
  - Throughput bound
    - ✓ Characterizes the raw computing capacity
    - ✓ Ultimate limit on program performance

Understanding Modern P's

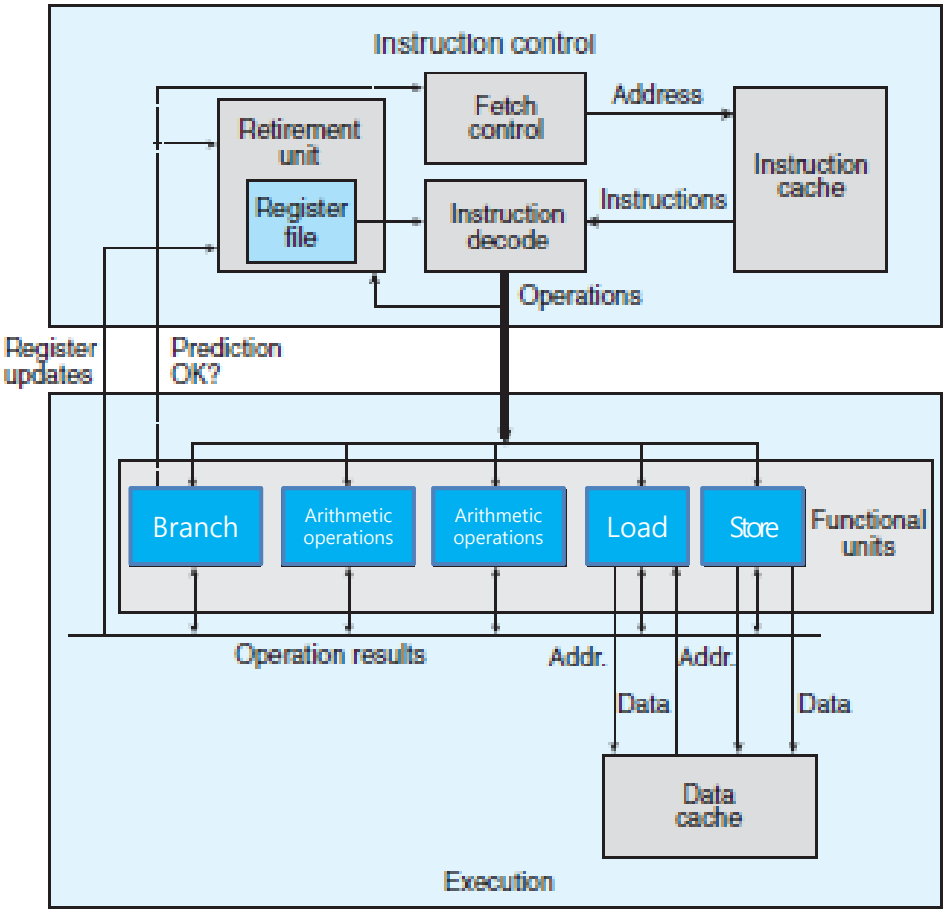
- Architecture of modern processors



Understanding Modern P's

Architecture of modern processors

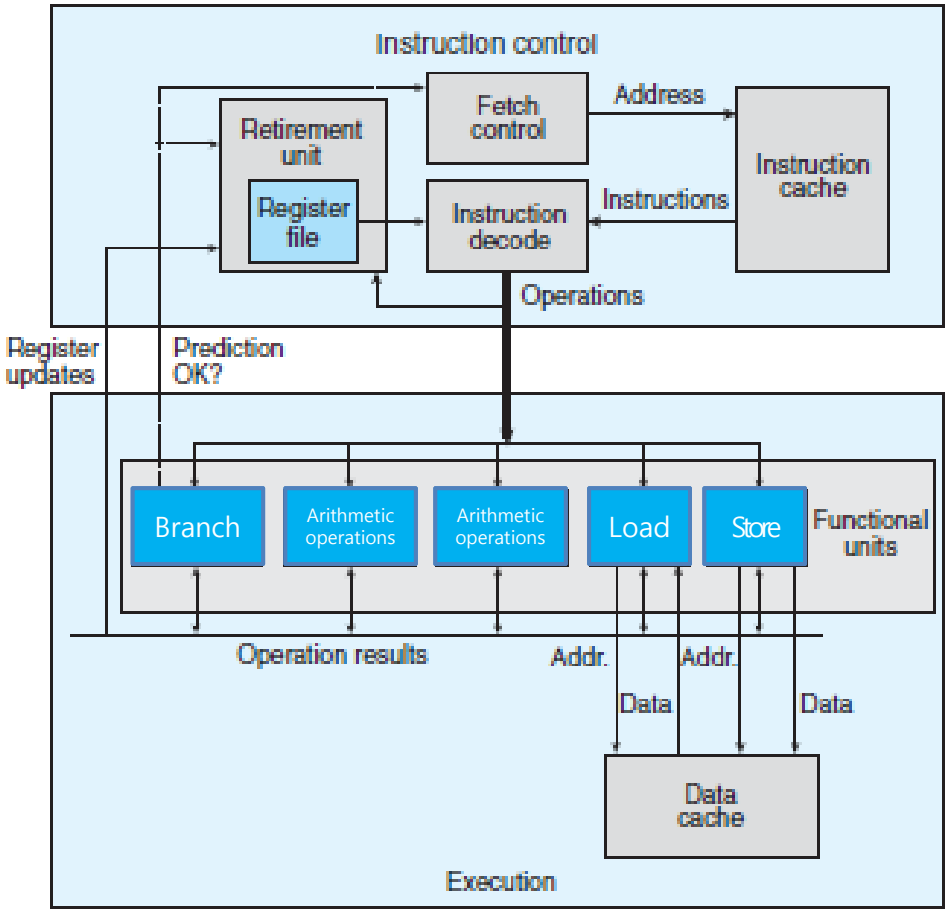
- Intel Core i7 model
- Haswell μarchitecture
- Superscalar
  - Can perform multiple operations on every clock cycle
- Out-of-order
  - Instruction execution sequence can be different from their ordering in machine-level program



Understanding Modern P's

Architecture of modern processors

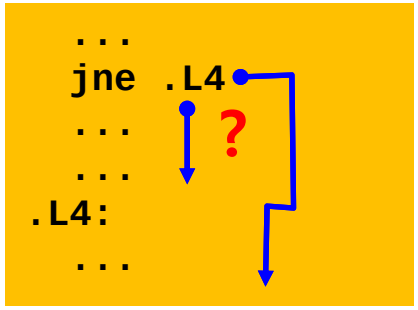
- ICU
  - Reading instructions from memory and generating primitive operations (micro-operations)
- EU
  - Executing the operations received from ICU



Understanding Modern P's

Architecture of modern processors

- ICU
  - Reads the instructions from the **instruction cache**, that contains the recently accessed instructions
  - Key technologies
    - ✓ **Prefetching**
      - Fetches next instruction ahead of finishing current instruction, and decodes and sends operations to EU
    - ✓ **Branch prediction** (by **fetch control** unit)
      - Guesses whether or not a branch will be taken
      - Predicts the target address for the branch
      - Loop 에서의 prediction은 루프 안쪽으로 수행



    - ✓ **Speculative execution**
      - Prefetching, decoding, and even executing instructions before determining the correctness of the branch prediction

Understanding Modern P's

Architecture of modern processors

- ICU
  - Instruction decoding
    - ✓ In CISC machines such as x86 processors, an instruction can be decoded into a variable # of operations
    - ✓ Example)
      - `addq %rax,%rdx` Converted into a single operation
      - `addq %rax,8(%rdx)` Converted into a multiple operations

        - 3 operations for loading, adding, and storing

Understanding Modern P's

Architecture of modern processors

- EU
  - Receives a number of operations on each clock cycle
  - The operations are dispatched to a set of functional units, that perform the actual operations
  - Load and store units access memory via **data cache**, that contains recently accessed data values

  - **Speculative execution**
    - ✓ Pre-execution, storing the results in temporary area
    - ✓ Examining branch instruction in EU for mis-prediction, in which case, discarding the results and signaling with the correct branch destination

Understanding Modern P's

Architecture of modern processors

- EU
  - Multiple different functional units
    - ✓ Each functional unit is intentionally designed to be able to perform a variety of different operations
  - For Intel Core i7 Haswell,
    - ✓ 8 functional units

0. Integer arithmetic, FP multiplication, integer and FP division, branches
1. Integer arithmetic, FP addition, integer multiplication, FP multiplication
2. Load, address computation
3. Load, address computation
4. Store
5. Integer arithmetic
6. Integer arithmetic, branches
7. Store, address computation

4 units for integer arithmetic
1 unit for integer multiplication
1 unit for FP addition
2 units for FP multiplication
2 units for load operation

Understanding Modern P's

Architecture of modern processors

- ICU (retirement unit)
  - Keeps track of ongoing processing and makes sure that it obeys the sequential semantics of the machine-level program
    - ✓ Makes the instruction **retire**, when the operations of the instruction have completed and any branch points to the instruction are confirmed as correctly predicted
      - Updates the program registers
    - ✓ Makes the instruction to be **flushed**, in case mis-prediction has occurred
      - Discard any results that may have been computed
      - Do not alter the program state

Understanding Modern P's

■ Architecture of modern processors

- EU
  - Register renaming
    - ✓ Mechanism for controlling the communication of operands among the execution units

When an instruction that updates register ***r*** is decoded, a ***tag t*** is generated giving a unique identifier to the result of the operation. An entry ***(r, t)*** is added to a **table** maintaining the association between program register ***r*** and tag ***t*** for an operation that will update this register. When a subsequent instruction using register ***r*** as an operand is decoded, the operation sent to the execution unit will contain ***t*** as the source for the operand value. When some execution units complete the first operation, it generates a result ***(v, t)*** indicating that the operation with tag ***t*** produced value ***v***. Any operation waiting for ***t*** as a source will then use ***v*** as the source value, a form of data forwarding. By this mechanism, values can be forwarded directly from one operation to another, rather than being written to and read from the register file, enabling the second operation to begin as soon as the first has completed. The **renaming table** only contains entries for registers having pending write operations. When a decoded instruction requires a register ***r***, and there is no tag associated with this register, the operand is retrieved directly from the register file. With register renaming, an entire sequence of operations can be performed speculatively, even though the registers are updated only after the processor is certain of the branch outcomes.

Understanding Modern P's

Functional unit performance

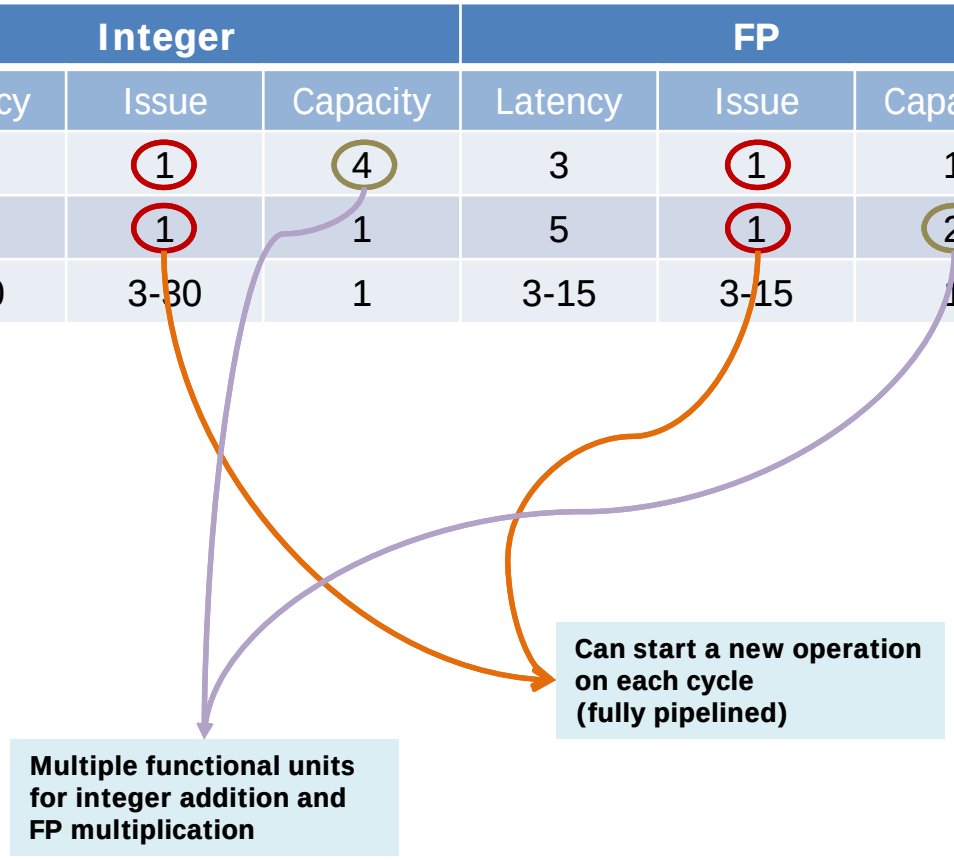
- Performance characteristics of Intel Core i7
  - Latency
    - ✓ The total time (# of clock cycles) required to perform the operation
  - Issue-time
    - ✓ The minimum # of clock cycles between two successive operations of the same type
  - Capacity
    - ✓ # of functional units capable of performing that operation
  - Max throughput = Capacity $\times \frac{1}{\text{Issue-time}}$ (operations per cycle)

Understanding Modern P's

Functional unit performance

- Performance characteristics of Intel Core i7 Haswell

Operation	Integer			FP		
	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3-30	3-30	1	3-15	3-15	1



Understanding Modern P's

- **Functional unit performance**
  - Performance characteristics of Intel Core i7 Haswell
    - Notes)
      - ✓ Latencies increase
        - For more complex data types
        - For more complex operations
      - ✓ Fully pipelined functional units
        - Issue time of 1
      - ✓ Divider
        - Not pipelined (Issue-time = Latency)
        - Comparatively costly operation

Understanding Modern P's

Functional unit performance

- Performance of **combineX()** functions
  - Latency bound
    - ✓ Minimum value for the CPE for any function that must perform the combining operation in a strict sequence
  - Throughput bound
    - ✓ Minimum bound for the CPE based on the maximum rate at which the functional units can produce results

Bound	Integer		FP	
	+	*	+	*
Latency	1.00	3.00	3.00	5.00
Throughput	0.50	1.00	1.00	0.50

- The processor can potentially sustain a rate of 4 operations per cycle
- But, the need to read elements from memory creates an additional throughput bound
  - The 2 load units limit the processor to reading at most 2 data values per clock cycle, yielding a throughput bound of 0.50

Understanding Modern P's

- **Data-flow representation (of programs)**
  - A tool for analyzing the performance of a machine-level program
    - Shows how the data dependencies between the different operations affect (constrain) their execution order
    - Can get a lower bound on the # of clock cycles required to execute a program, by finding **critical paths** in the **data-flow graph**

Understanding Modern P's

Data-flow representation

- Example) CPE measurements of the function **combine4**

Function	Method	Integer		FP	
		+	*	+	*
combine4	Accumulate in local var acc	1.27	3.01	3.01	5.01
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

- Computing the sum or product of **n** elements requires around (**L**·**n** + **K**) clock cycles,
  where **L** is the latency of the combining operation
  and **K** represents the overhead
- CPE ≡ **L** (latency bound)

Understanding Modern P's

- **Data-flow graphs**
  - Visualize how the data dependencies in a program dictate its performance
  - Example)
    - **combine4()**
    - Double-precision FP data, multiplication

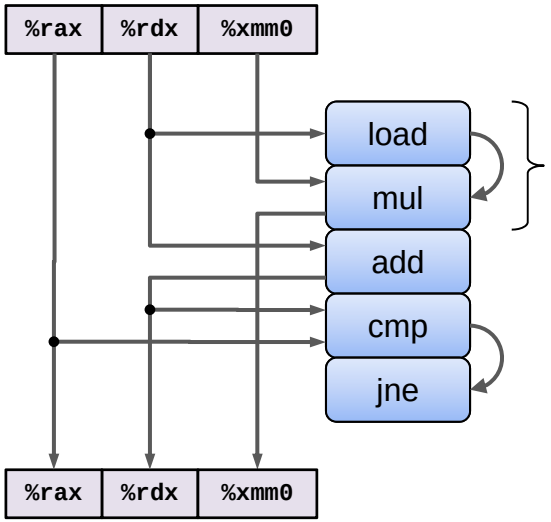
```
void combine4(v_p v, data_t *dest)
{
   long i;
   long length = v_length(v);
   data_t *data = get_v_start(v);
   data_t acc = IDENT;

   for (i = 0; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

```
[Inner loop of combine4()]
   data_t = double, OP = *
   acc in %xmm0, data+i in %rdx, data+length in %rax

.L25:                                   loop:
   vmulsd (%rdx),%xmm0,%xmm0               Multiply acc by data[i]
   addq $8,%rdx                            Increment data+i
   cmpq %rax,%rdx                          Compare to data+length
   jne .L25                                If !=, goto loop
```

Understanding Modern P's

Data-flow graphs: graphical representation

- Example) **combine4()**
  - 4 instructions → 5 operations




```
vmulsd (%rdx),%xmm0,%xmm0
addq $8,%rdx
cmpq %rax,%rdx
jne loop
```

Loop registers ⟶ %rdx, %xmm0

- Both used as source and destination R's in the loop
- Values generated in one iteration are used in another

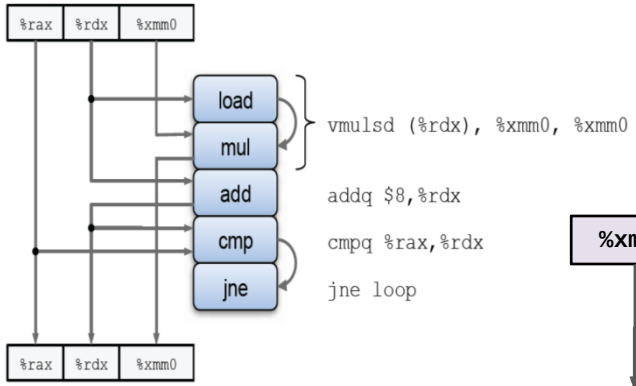
Understanding Modern P's

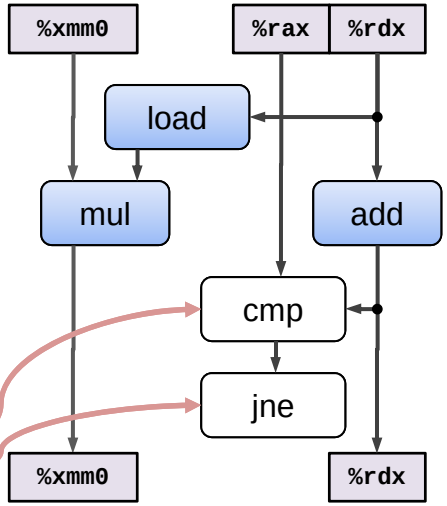
- **Data-flow graphs: graphical representation**
  - Classification on registers
    - Read-only registers (**%rax**)
      - ✓ Used as source values, not modified in the loop
    - Write-only registers
      - ✓ Used as destinations
    - Local registers (CCs)
      - ✓ Updated and used in the loop
      - ✓ No dependency from one iteration to another
    - **Loop registers** (**%rdx**, **%xmm0**)
      - ✓ Used as source values and as destinations, with iteration dependency

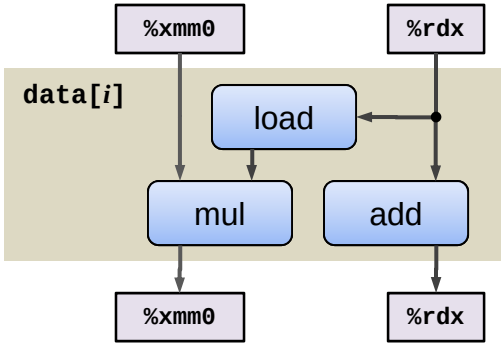
The chains of operations between loop registers determine the performance-limiting data dependencies

Understanding Modern P's

- **Data-flow graphs**
  - Example) **combine4()**
    - Further refinements





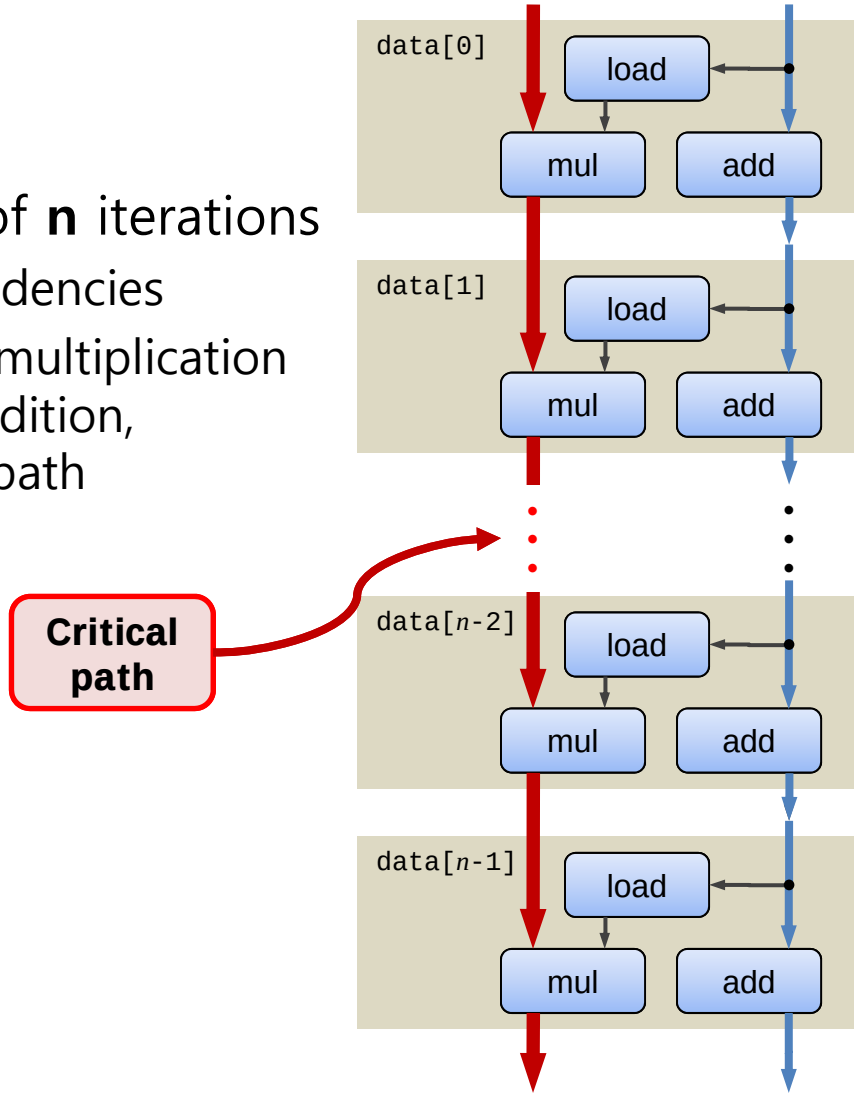


Not related to the dependencies between loop R's

Only loop R's and related operations

Understanding Modern P's

- **Data-flow graphs**
  - Example) **combine4()**
    - Data-flow representation of **n** iterations
      - ✓ Two chains of data dependencies
      - ✓ Assuming 5 cycles for FP multiplication and 1 cycle for integer addition, left chain forms a critical path
        - **5·n** cycles to execute



Loop Unrolling

Loop unrolling

- Program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration

```
void psum1(float a[], float p[], long n)
{
   long i;
   p[0] = a[0];
   for (i=1; i<n; i++)
      p[i] = p[i-1] + a[i];
}
```

```
void psum2(float a[], float p[], long n)
{
   long i;
   p[0] = a[0];
   for (i=1; i<n-1; i+=2) {
      float mv = p[i-1] + a[i];
      p[i] = mv;
      p[i+1] = mv + a[i+1];
   }
   if (i<n)
      p[i] = p[i-1] + a[i];
}
```

Loop Unrolling

Loop unrolling

- Can improve performance in 2 ways
  - Reduces the number of operations that do not contribute directly to the program results
    - ✓ Loop indexing, conditional branching, …
  - Exposes ways that can further transform the code to reduce the number of operations in the critical paths

  - Loop unrolling by a factor of **k**
    - ✓ **k-way loop unrolling (k × 1)**
    - ✓ Computations on **k** elements in one iteration

Loop Unrolling

Loop unrolling

- Example) No loop unrolling

```
void combine4(v_p v, data_t *dest)
{
   long i;
   long length = v_length(v);
   data_t *data = get_v_start(v);
   data_t acc = IDENT;

   for (i = 0; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

Loop Unrolling

Loop unrolling

- Example) 2-way loop unrolling on the combining code

```
void combine5(v_p v, data_t *dest)
{
    long i;
    long length = v_length(v);
    long limit = length - 1;
    data_t *data = get_v_start(v);
    data_t acc = IDENT;

    for (i = 0; i < limit; i += 2) {
        acc = (acc OP data[i]) OP data[i+1];
    }

    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Loop Unrolling

Loop unrolling

- Performance of the unrolled code for k = 2 and k = 3

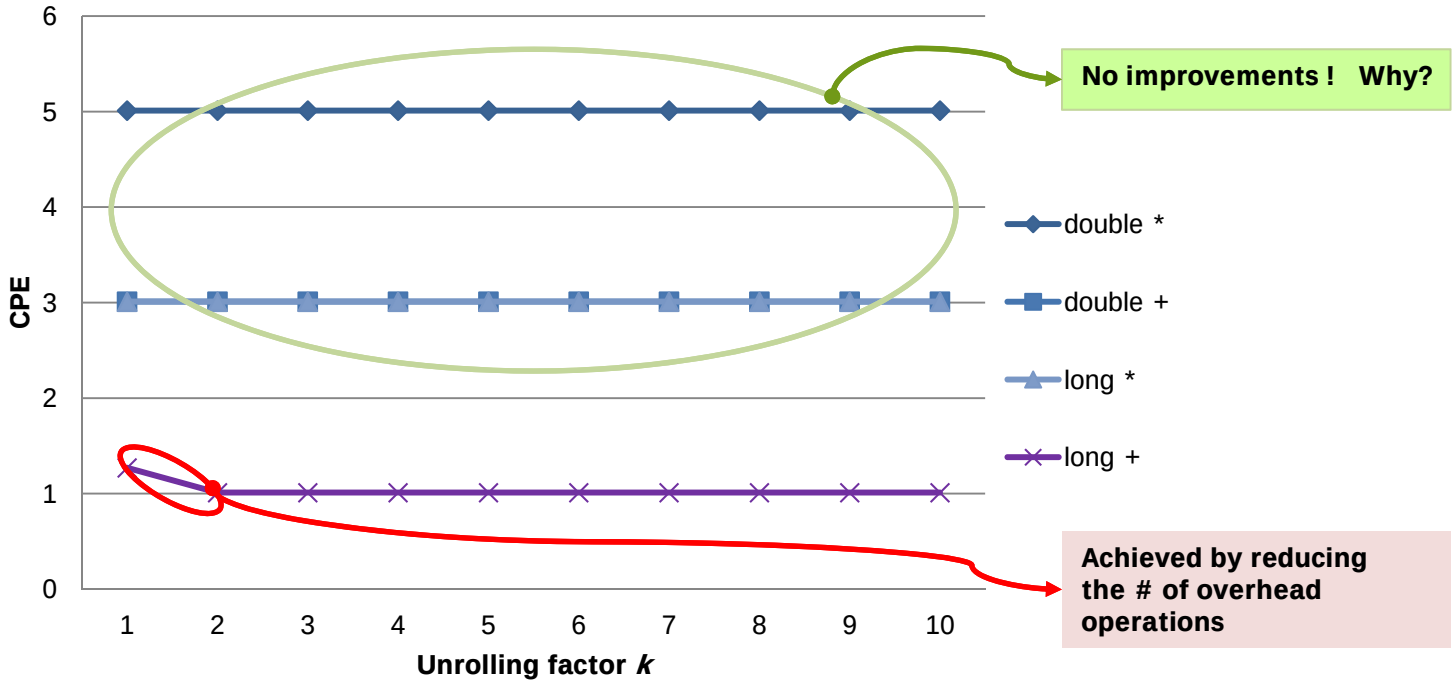
Function	Method	Integer		FP	
		+	*	+	*
combine4	Accumulate in temp	1.27	3.01	3.01	5.01
combine5	2×1 unrolling	1.01	3.01	3.01	5.01
	3×1 unrolling	1.01	3.01	3.01	5.01
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

- Note)
  - CPE for integer addition improves (by reducing OH operations), while CPEs for other operations do not improve
  - Why ?

Loop Unrolling

Loop unrolling

- Performance of the unrolled code for k = 1 ~ 10

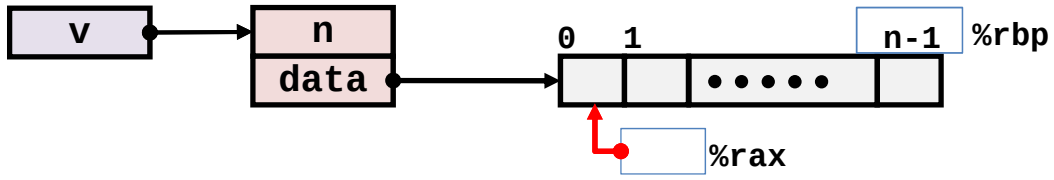


Loop Unrolling

- **Loop unrolling**
  - Why no improvements ?
    - Assembly code (**combine5()**)

```
void combine5(v_p v, data_t *dest)
{
   ...
   for (i = 0; i < limit; i+=2) {
      acc = (acc OP data[i]) OP data[i+1];
   }

   for (; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

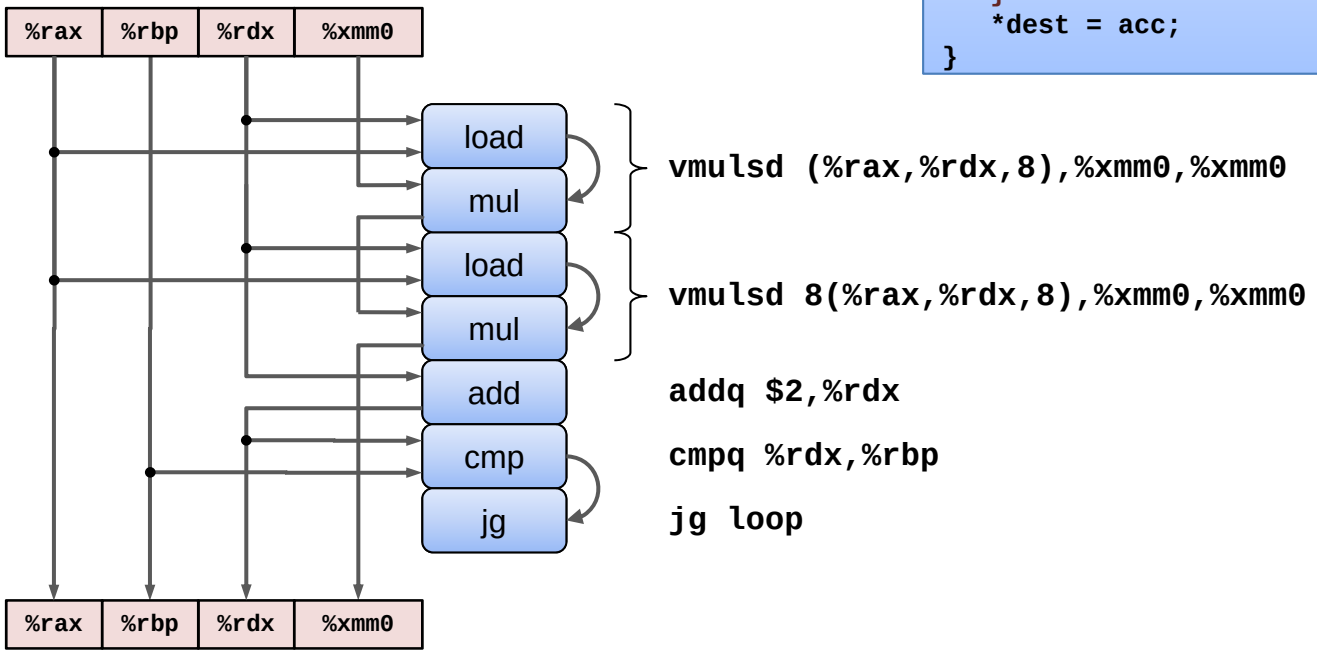
```
[Inner loop of combine5()]
   data_t = double, OP = *
   i in %rdx, data in %rax, limit in %rbp, acc in %xmm0

.L35:                                  loop:
   vmulsd (%rax,%rdx,8),%xmm0,%xmm0    Multiply acc by data[i]
   vmulsd 8(%rax,%rdx,8),%xmm0,%xmm0   Multiply acc by data[i+1]
   addq $2,%rdx                        Increment i by 2
   cmpq %rdx,%rbp                      Compare limit:i
   jg .L35                             If >, goto loop
```



Loop Unrolling

- **Loop unrolling**
  - Why no improvements ?
    - Graphical representation (**combine5()**)

```
void combine5(v_p v, data_t *dest)
{
   ...
   for (i = 0; i < limit; i+=2) {
      acc = (acc OP data[i]) OP data[i+1];
   }

   for (; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

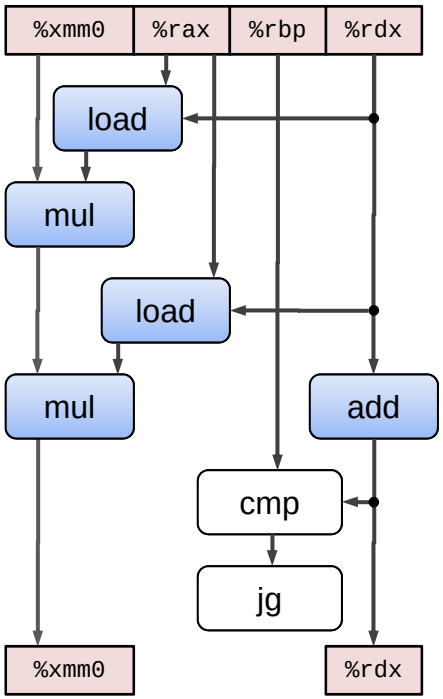


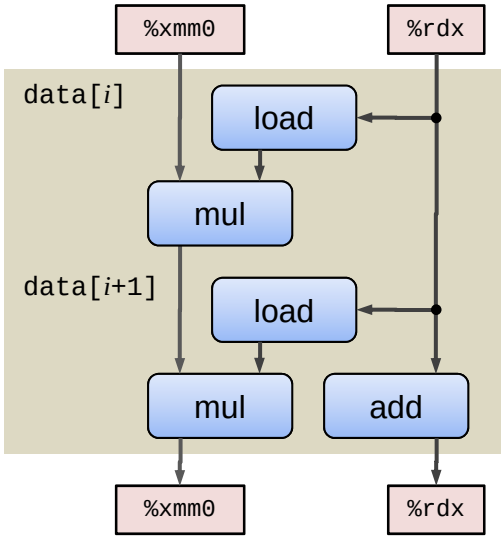

```
vmulsd (%rax,%rdx,8),%xmm0,%xmm0

vmulsd 8(%rax,%rdx,8),%xmm0,%xmm0

addq $2,%rdx

cmpq %rdx,%rbp

jg loop
```

Loop Unrolling

Loop unrolling

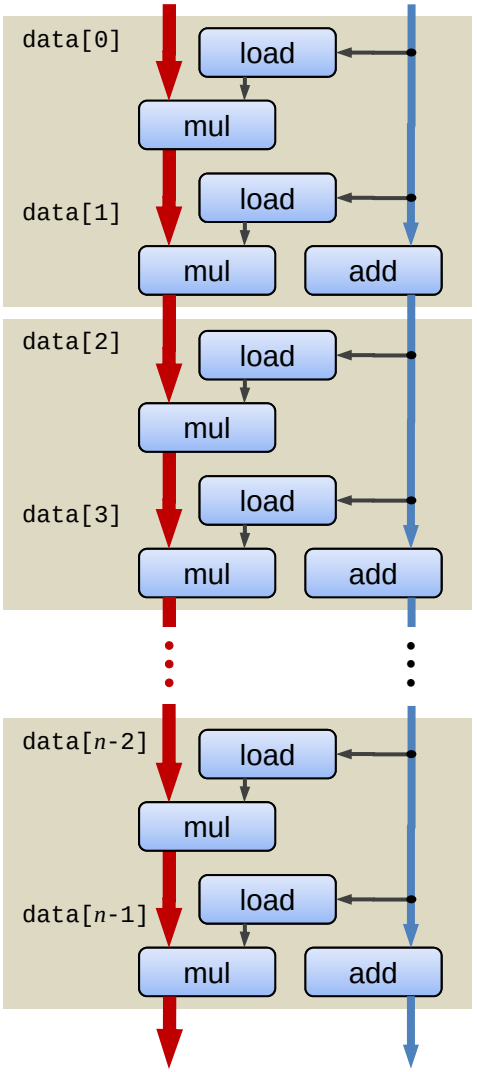
- Why no improvements ?
  - Refinements (**combine5()**)





Loop Unrolling

- **Loop unrolling**
  - Why no improvements ?
    - Data-flow representation (**combine5()**)
    - There are still a critical path of **n mul** operations in the graph
      - ✓ Half as many iterations, but, each iteration has 2 **mul** operations in sequence
    - No differences with the case of no loop unrolling



Loop Unrolling

Loop unrolling

- Note) Getting the compiler to unroll loops

```
linux> gcc … -funroll-loops …
```

  - Now, **gcc** will perform some forms of loop unrolling when invoked with optimization level 3 or higher

Enhancing Parallelism

Multiple accumulators

- For our combining operations,
  on integer addition and multiplication,
  where the operations are commutative and associative

  - $P_n = \prod_{i=0}^{n-1} a_i$

  - $PE_n = \prod_{i=0}^{n/2-1} a_{2i}$, $PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}$, $P_n = PE_n \times PO_n \times \delta$

    where $\delta = 1$, if n is even
    a_{n-1}, if n is odd

Enhancing Parallelism

Multiple accumulators

- 2-way loop unrolling (2×1 unrolling)

```
void combine5(v_p v, data_t *dest)
{
   long i;
   long length = v_length(v);
   long limit = length - 1;
   data_t *data = get_v_start(v);
   data_t acc = IDENT;

   for (i = 0; i < limit; i+=2) {
      acc = (acc OP data[i]) OP data[i+1];
   }

   for (; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

Enhancing Parallelism

Multiple accumulators

- 2-way loop unrolling + 2-way parallelism (2×2 unrolling)

```
void combine6(v_p v, data_t *dest)
{
    long i;
    long length = v_length(v);
    long limit = length - 1;
    data_t *data = get_v_start(v);
    data_t acc0 = IDENT; data_t acc1 = IDENT;

    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i]
        acc1 = acc1 OP data[i+1];
    }
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Enhancing Parallelism

Multiple accumulators

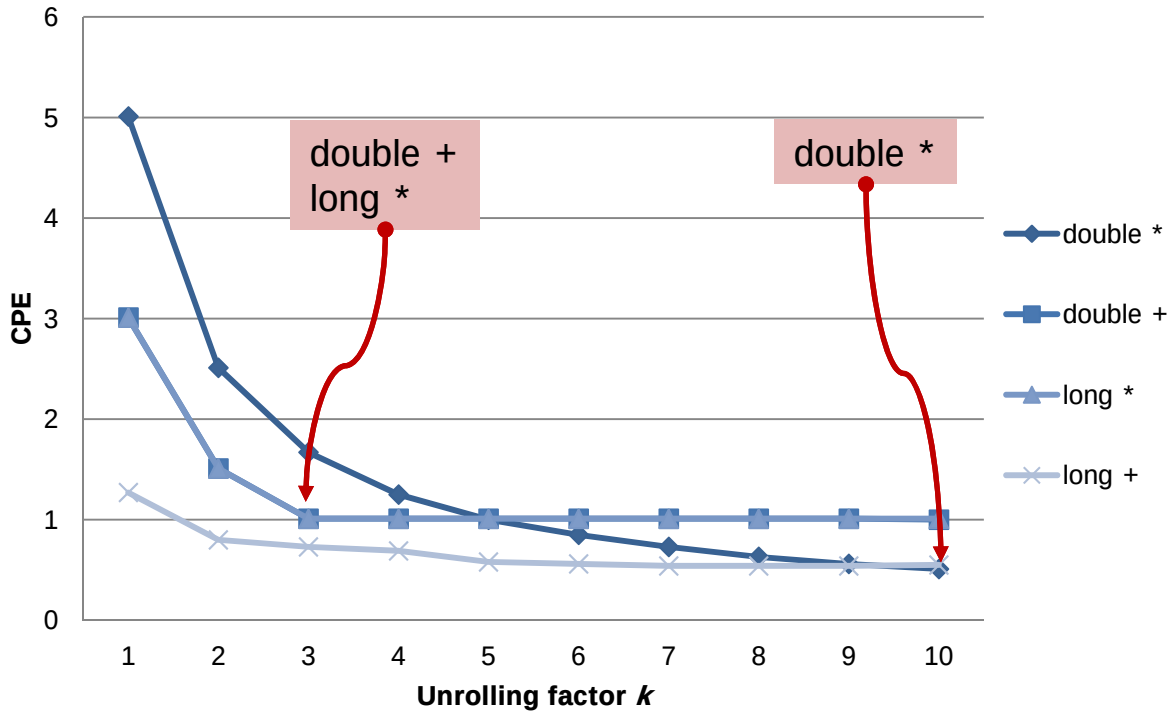
- Performance of 2×2 unrolling

Function	Method	Integer		FP	
		+	*	+	*
combine4	Accumulate in temp	1.27	3.01	3.01	5.01
combine5	2×1 unrolling	1.01	3.01	3.01	5.01
combine6	2×2 unrolling	0.81	1.51	1.51	2.51
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

Enhancing Parallelism

Multiple accumulators

- Performance of **k**×**k** unrolling for **k** = 1 ~ 10

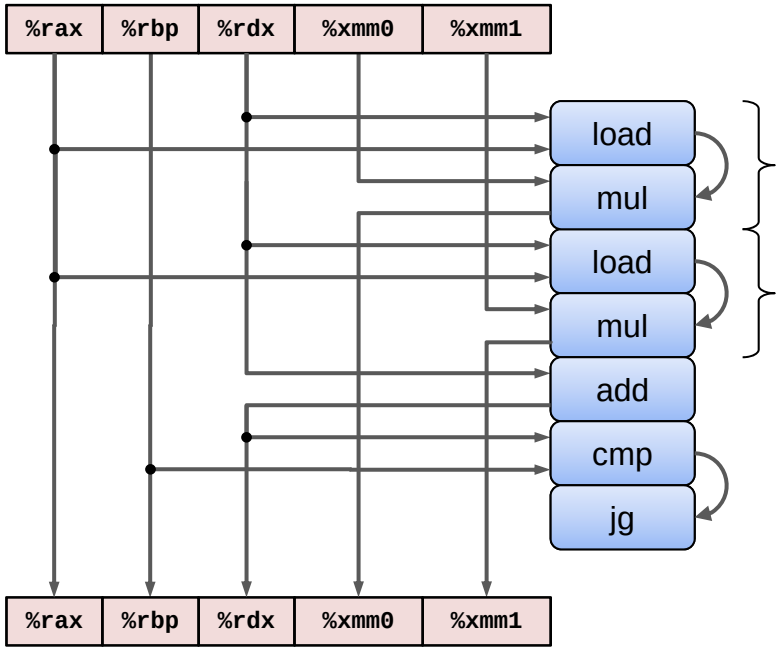


- CPEs for all of our combining cases improve with increasing values of **k**

Enhancing Parallelism

Multiple accumulators

- Graphical representation (**combine6()**)

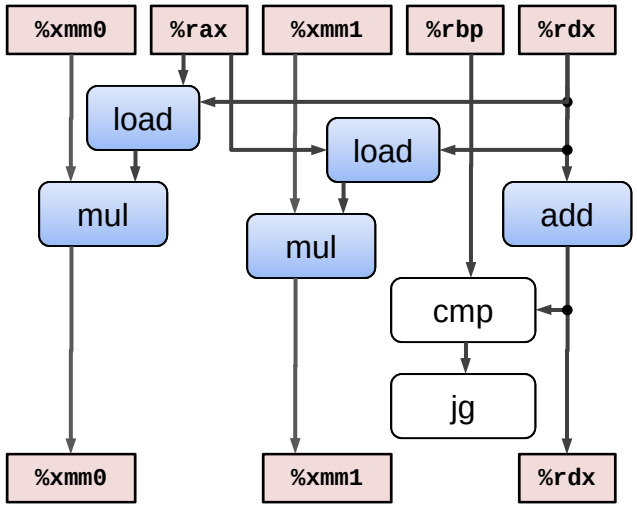


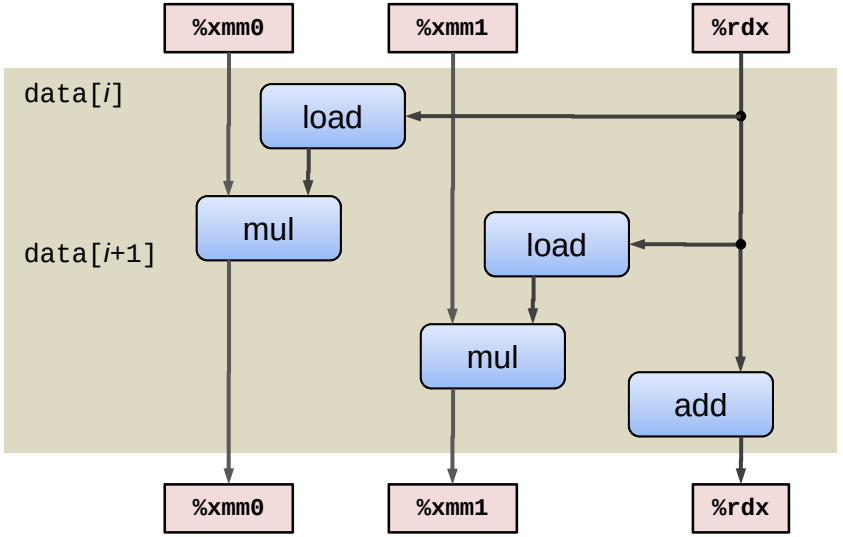

```
vmulsd (%rax,%rdx,8),%xmm0,%xmm0

vmulsd 8(%rax,%rdx,8),%xmm1,%xmm1

addq $2,%rdx
cmpq %rdx,%rbp
jg loop
```

Enhancing Parallelism

Multiple accumulators

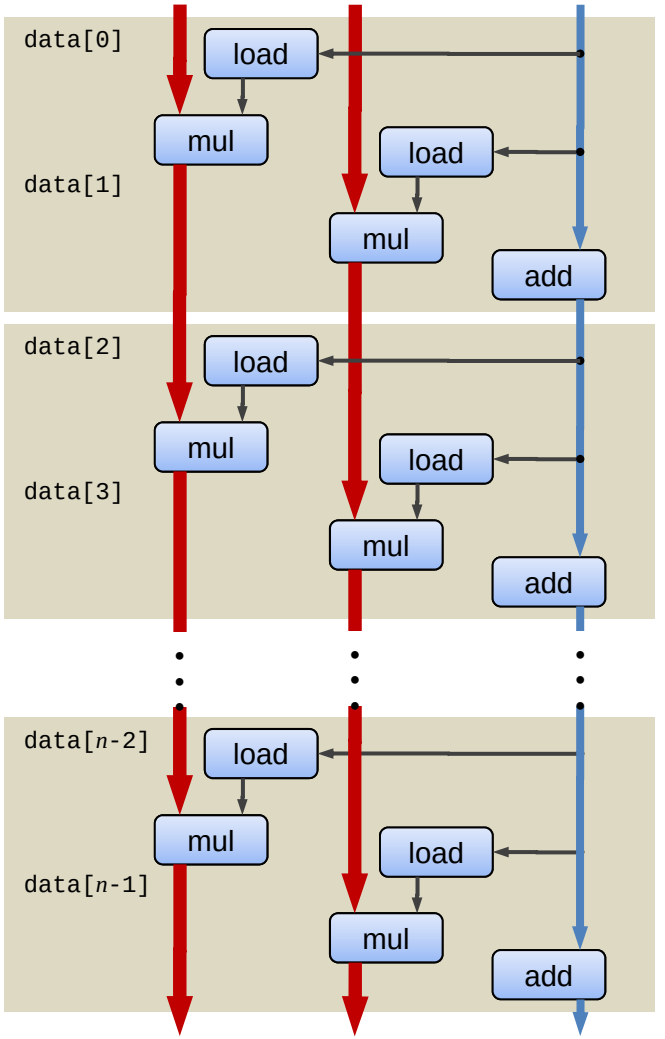
- Refinements (**combine6()**)





Enhancing Parallelism

- **Multiple accumulators**
  - Data-flow representation (**combine6()**)
  - 2 critical paths
    - 1 for even-numbered elements
    - 1 for odd-numbered elements
  - CPE of **L**/2
    for operations with latency **L**



Enhancing Parallelism

Multiple accumulators

- Note)
  - FP addition and multiplication are not associative
  - So, **combine5()** and **combine6()** may result in different results due to the rounding and overflow
    - ✓ In most real-life applications, producing such results is unlikely
  - But, program developer should check whether there can be such conditions

Enhancing Parallelism

- **Reassociation transformation**
  - Utilizes associativity law
  - Improves performance by changing the combining order
  - Example) In **combine5()**,

```
acc = (acc OP data[i]) OP data[i+1];
```

```
acc = acc OP (data[i] OP data[i+1]);
```

Enhancing Parallelism

Reassociation transformation

- 2×1 unrolling

```
void combine5(v_p v, data_t *dest)
{
   long i;
   long length = v_length(v);
   long limit = length - 1;
   data_t *data = get_v_start(v);
   data_t acc = IDENT;

   for (i = 0; i < limit; i+=2) {
      acc = (acc OP data[i]) OP data[i+1];
   }

   for (; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

Enhancing Parallelism

Reassociation transformation

- 2×1a unrolling

```
void combine7(v_p v, data_t *dest)
{
    long i;
    long length = v_length(v);
    long limit = length - 1;
    data_t *data = get_v_start(v);
    data_t acc = IDENT;

    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }

    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Enhancing Parallelism

Reassociation transformation

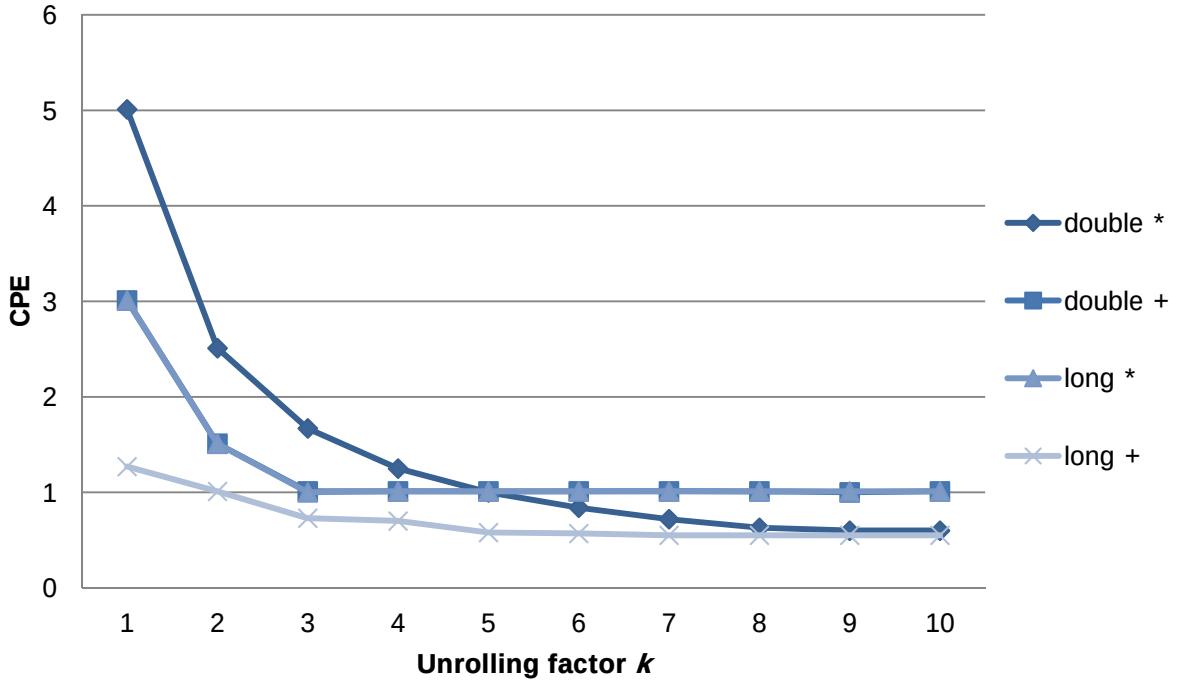
- Performance of 2×1a unrolling

Function	Method	Integer		FP	
		+	*	+	*
combine4	Accumulate in temp	1.27	3.01	3.01	5.01
combine5	2×1 unrolling	1.01	3.01	3.01	5.01
combine6	2×2 unrolling	0.81	1.51	1.51	2.51
combine7	2×1a unrolling	1.01	1.51	1.51	2.51
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

Enhancing Parallelism

■ Reassociation transformation

- Performance of **k**×1a unrolling



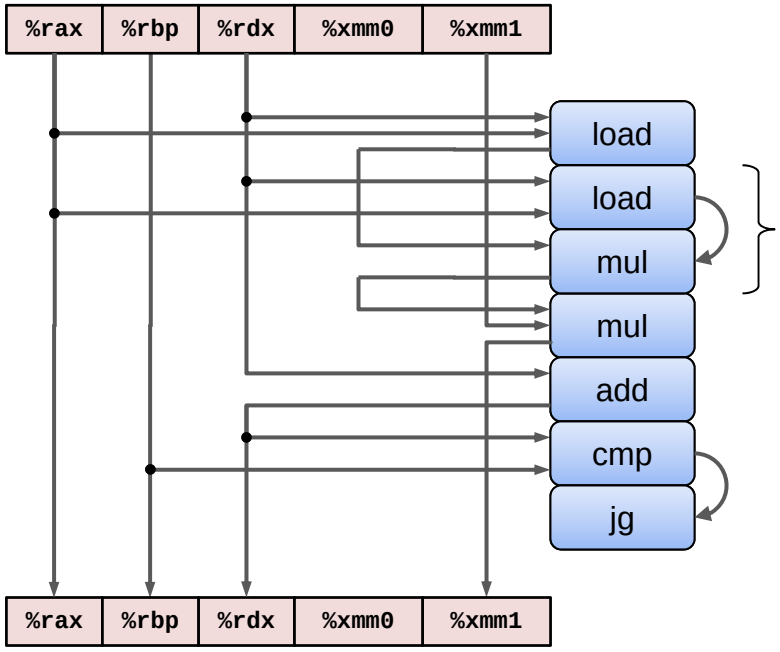
- Similar to the case of **k**×**k** unrolling

Enhancing Parallelism

Reassociation transformation

- Graphical representation (**combine7()**)

```
void combine7(v_p v, data_t *dest)
{
   ...
   for (i = 0; i < limit; i+=2) {
      acc = acc OP (data[i] OP data[i+1]);
   }

   for (; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

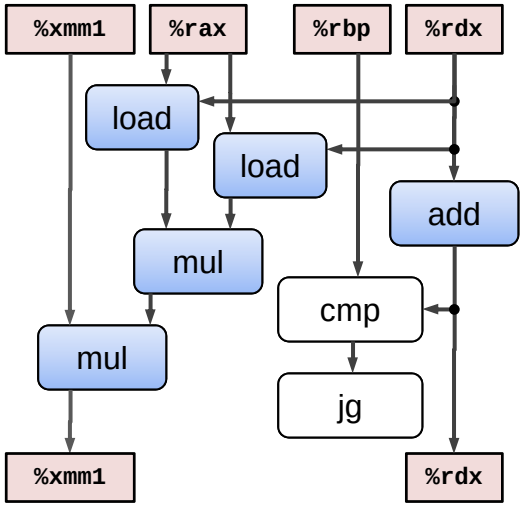


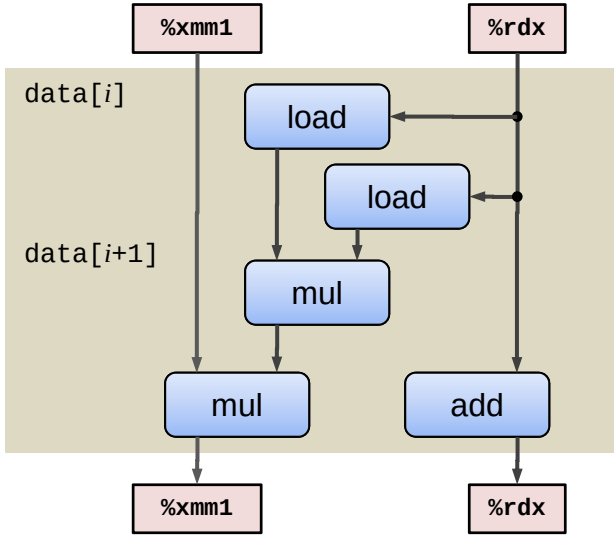

```
vmovsd (%rax,%rdx,8),%xmm0

vmulsd 8(%rax,%rdx,8),%xmm0,%xmm0

vmulsd %xmm0,%xmm1,%xmm1
addq $2,%rdx
cmpq %rdx,%rbp
jg loop
```

Enhancing Parallelism

Reassociation transformation

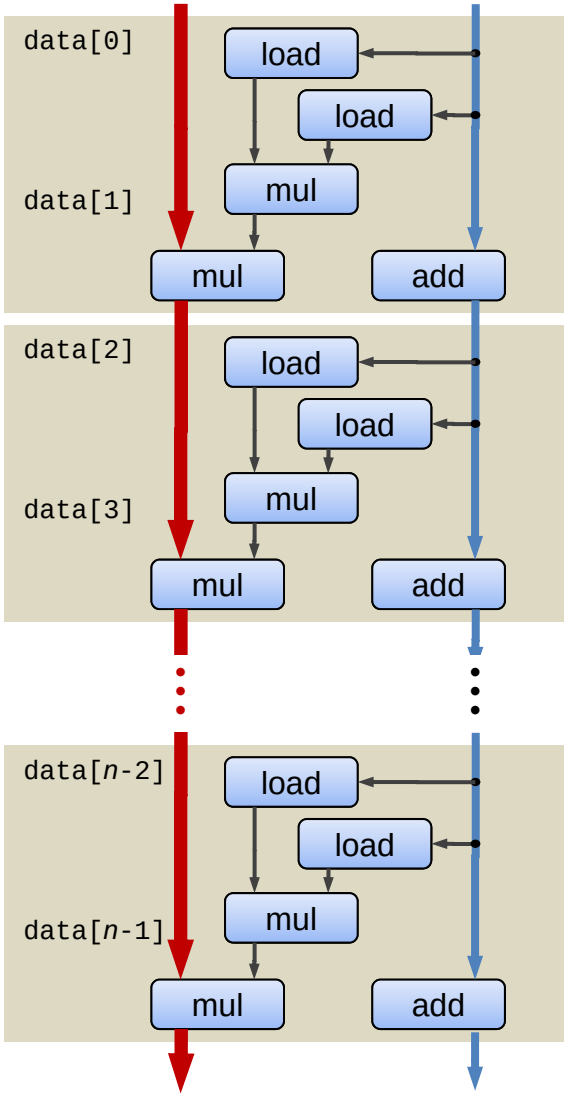
- Refinements (**combine7()**)





Enhancing Parallelism

- **Reassociation transformation**
  - Data-flow representation (**combine7()**)
  - Only 1 **mul** operation forms a data-dependency chain between loop registers
  - **n**/2 operations along the critical path



Enhancing Parallelism

Reassociation transformation

- Note)
  - Integer addition and multiplication is associative
    - ✓ Current versions of **gcc** do perform reassociations for integer operations
  - FP addition and multiplication are not associative
    - ✓ Most compilers do not attempt any reassociations for FP operations
  - In general, unrolling a loop and accumulating multiple values in parallel is a more reliable way to achieve improved program performance

Summary: Combining Code

Without using SIMD parallelism

Function	Method	Integer		FP	
		+	*	+	*
combine1	Abatract –O1	10.12	10.12	10.17	11.14
combine6	2×2 unrolling	0.81	1.51	1.51	2.51
	10×10 unrolling	0.55	1.00	1.01	0.52
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

Summary: Combining Code

- **Using SIMD parallelism**
  - Can get additional performance gains of nearly 4× or 8×
    - In case of FP multiplication,
      the CPE drops from 11.14 to 0.06
      (In this case, the performance gain of over 180×)

Summary: Combining Code

- **SIMD**
  - Single-Instruction Multiple-Data
  - Used by SSE or AVX instructions
  - **XMM**(**YMM**) registers can hold multiple values
    - 4(8) integer or single-precision FP numbers or 2(4) double-precision FP numbers
  - SSE/AVX instructions can perform vector operations on these registers in parallel
    - Ex) `vmulps (%rcx),%xmm0` 4 multiplications

Summary