# Program Optimization

**(Chapter 5)**

# Outline

- **Generally Useful Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
  - Removing unnecessary procedure calls
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Exploiting Instruction-Level Parallelism**
- **Dealing with Conditionals**
  - Branch Prediction

# Performance Realities

***There's more to performance than asymptotic complexity.***

- **Constant factors matter too!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

# Optimizing Compilers

- **Provide efficient mapping of program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Don’t (usually) improve asymptotic efficiency**
  - up to programmer to select best overall algorithm
  - Big-O savings are (often) more important than constant factors
    - but constant factors also matter
- **Have difficulty overcoming “optimization blockers”**
  - potential memory aliasing
  - potential procedure side-effects

# Aliasing

*“When data in memory can be acessed in more than one way”*

Example: Is it safe to keep x in a register?

```
int x;
int *p;
…
*p = 123;
…
```

What if p points to x?

In general, we cannot know the answer to this question with out running the program.

# Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
  - Often prevents it from making optimizations when would only affect behavior under pathological conditions.
- **Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles**
  - e.g., Data ranges may be more limited than variable types suggest
- **Most analysis is performed only within procedures**
  - Whole-program analysis is too expensive in most cases
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs

**When in doubt, the compiler must be conservative!**

# Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler

**Machine Independent Optimizations:**

- Code Motion
- Reduction in Strength
- Using Registers for frequently accessed variables
- Share Common Subexpressions

# Code motion

## Reduce frequency that a computation is performed

**IF it will always produce the same result**

**THEN move it out of inner loop**

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

# Code motion

Most compilers do a good job with array code and simple loop structures

- Code Generated by GCC

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  int *p = a+ni;
  for (j = 0; j < n; j++)
    *p++ = b[j];
}
```

```
  imull %ebx,%eax          # i*n
  movl 8(%ebp),%edi        # a
  leal (%edi,%eax,4),%edx  # p = a+i*n (scaled by 4)
.L40:                      # Inner Loop
  movl 12(%ebp),%edi       #   b
  movl (%edi,%ecx,4),%eax  #   b+j  (scaled by 4)
  movl %eax,(%edx)         #   *p = b[j]
  addl $4,%edx             #   p++  (scaled by 4)
  incl %ecx                #   j++
  jl .L40                  #   loop back if j<n
```

# Reduction in strength

- Replace costly operations with simpler ones
  - Example: Replace multiply & divide with shifts & adds

    `17*x` → `(x << 4) + x`

    - Depends on cost of multiply or divide instruction
    - Is it worth it? This is “machine dependent”
- Recognize sequence of products and replace with addition

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

# Using registers

*Reading and writing registers is much faster than reading/writing memory!*

**Limitations**

- Compiler not always able to determine whether variable can be held in register
- Possibility of Aliasing

  "Multiple ways of naming/accessing a variable or data item."

  There could be a pointer to this variable.

  Putting it in a registers could be risky.

  RISKY! It might change the behavior of the program!!!

**The Performance gain is huge!**

# Share common subexpressions

Want to reuse computations where possible

- But compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up+down+left+right;
```

**3 multiplications: i*n, (i–1)*n, (i+1)*n**

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up+down+left+right;
```

**1 multiplication: i*n**

```
  leal -1(%edx),%ecx # i-1
  imull %ebx,%ecx    # (i-1)*n
  leal 1(%edx),%eax     # i+1
  imull %ebx,%eax       # (i+1)*n
  imull %ebx,%edx    # i*n
```

# Example: Convert a string to lower case

A function to convert string to lower case:

```
void lower(char *s) {
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

If length of string is n, how does the run-time of this function grow with n?

- Linear, Quadratic, Cubic, Exponential?

# Strlen

```
int lencnt = 0;
size_t strlen(const char *s)
{
     size_t length = 0;
     while (*s != '\0') {
        s++; length++;
     }
     lencnt += length;
     return length;
}
```

*First call:*

Time required = n (proportional to string length)

*Second call:*

Another n

*Number of times called:*

n

*Total time:*

$n + n + n + \ldots n \;=\; n^2 + \ldots \;=\; O(n^2)$

# Example: Convert a string to lower case

A function to convert string to lower case:

```
void lower(char *s) {
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Notice: strlen is executed every iteration

- Must scan string until finds '\0'
- strlen is linear in length of string

Loop itself is linear in length of string

Overall performance is quadratic… $O(n^2)$

# Example: Convert a string to lower case

```
void lower(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

## Let’s apply code motion

Consider the call to **strlen…**

Result does not change from one iteration to another.

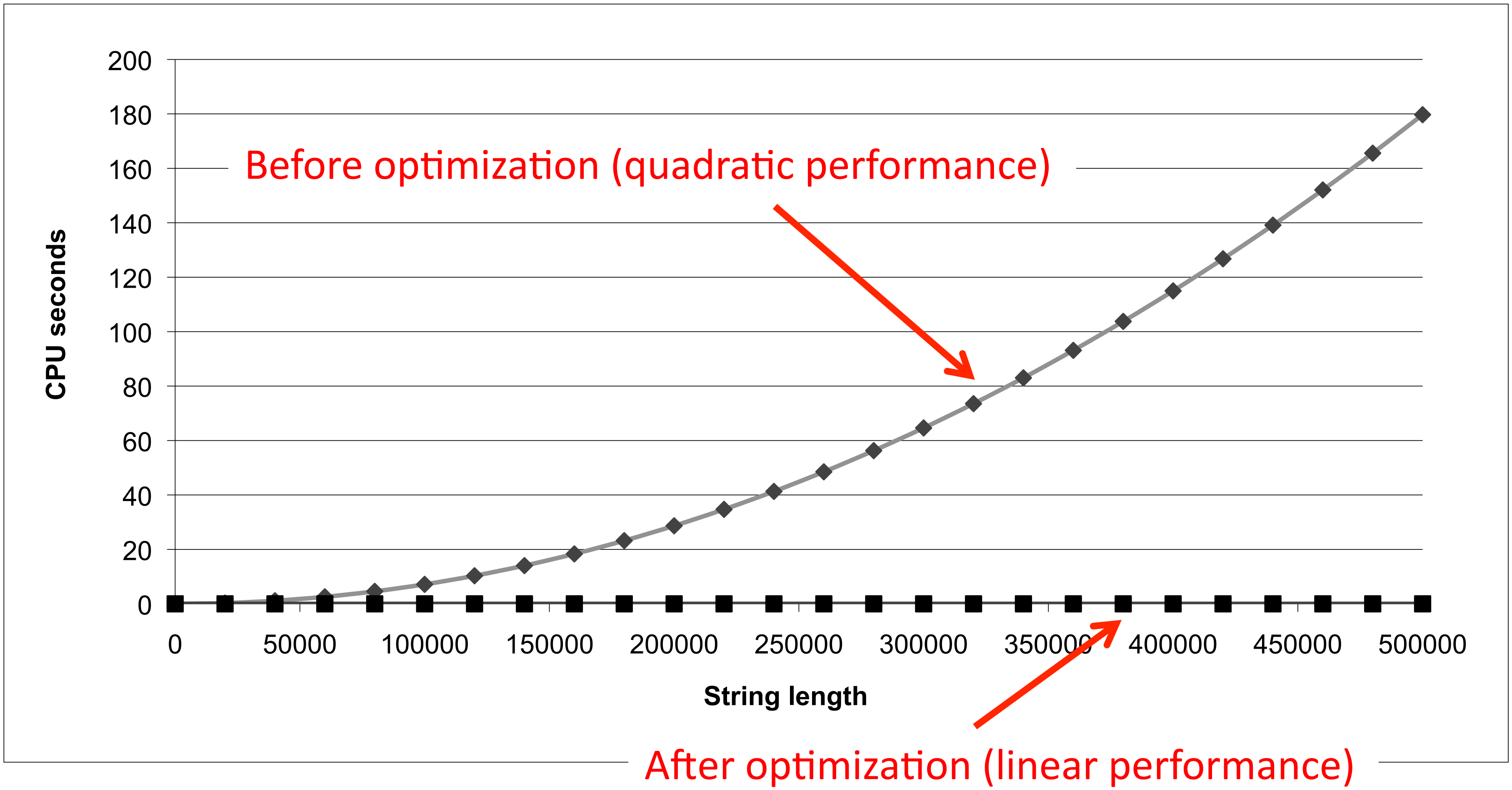
Compiler does not know this, though.

Move call to **strlen** outside of loop.

# Example: Convert a string to lower case

Linear Performance O(n): Time doubles when string length doubles

Quadratic Performance $O(n^2)$: Time qadruples when length doubles



# Optimization Blocker: Procedure Calls

***Why couldn't compiler move `strlen` out of inner loop?***

- Procedure may have side effects
  - Alters global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`

- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Use of `inline` functions
    - GCC does this with –O2
  - Do your own code motion

# Memory Aliasing

```
/* Sum the rows in a n X n matrix "a"
   and store in vector "b"  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# inner loop
.L53:
        addsd   (%rcx), %xmm0           # FP add
        addq    $8, %rcx
        decq    %rax
        movsd   %xmm0, (%rsi,%r8,8)     # FP store
        jne     .L53
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

# Memory Aliasing

```
/* Sum the rows in a n X n matrix "a"
   and store in vector "b"  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of B:**

```
init:   [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- Must consider possibility that these updates will affect program behavior

# Removing Aliasing

```
/* Sum the rows in a n X n matrix "a"
   and store in vector "b"  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
         b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L66:
        addsd   (%rcx), %xmm0   # FP Add
        addq    $8, %rcx
        decq    %rax
        jne     .L66
```

- No need to store intermediate results

# Optimization Blocker: Memory Aliasing

## Aliasing

Two different memory references specify single location

Easy to have happen in C

- Since allowed to do address arithmetic
- Direct access to storage structures

Get in habit of introducing local variables

- Accumulating within loops

*Your way of telling compiler not to check for aliasing*

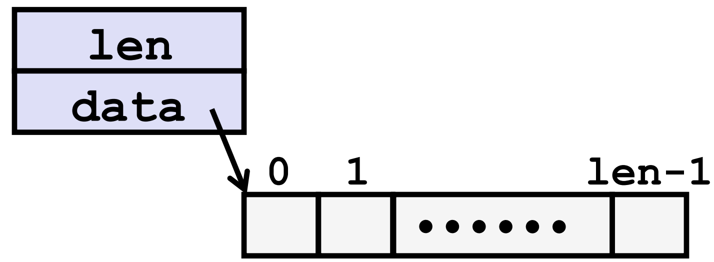
# Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**

  *Hardware can execute multiple instructions in parallel*

- **But performance is limited by "data dependencies"**
- **Simple transformations can have dramatic performance improvement**
  - Often, compilers cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

# Benchmark Example: Data Type for Vectors




```
/* data structure for vectors */
typedef struct{
   int len;
   double *data;
} vec;
```

```
/* retrieve vector element and store at val */
double get_vec_element(*vec, idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

# Benchmark Computation

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

**Compute sum or product of vector elements**

## Data Types

Use different declarations for `data_t`

- `int`
- `float`
- `double`

## Operations

Use different definitions of `OP` and `IDENT`

- `+ / 0`
- `* / 1`

# Cycles Per Element (CPE)

- A convenient way to express performance of program that operates on vectors or lists

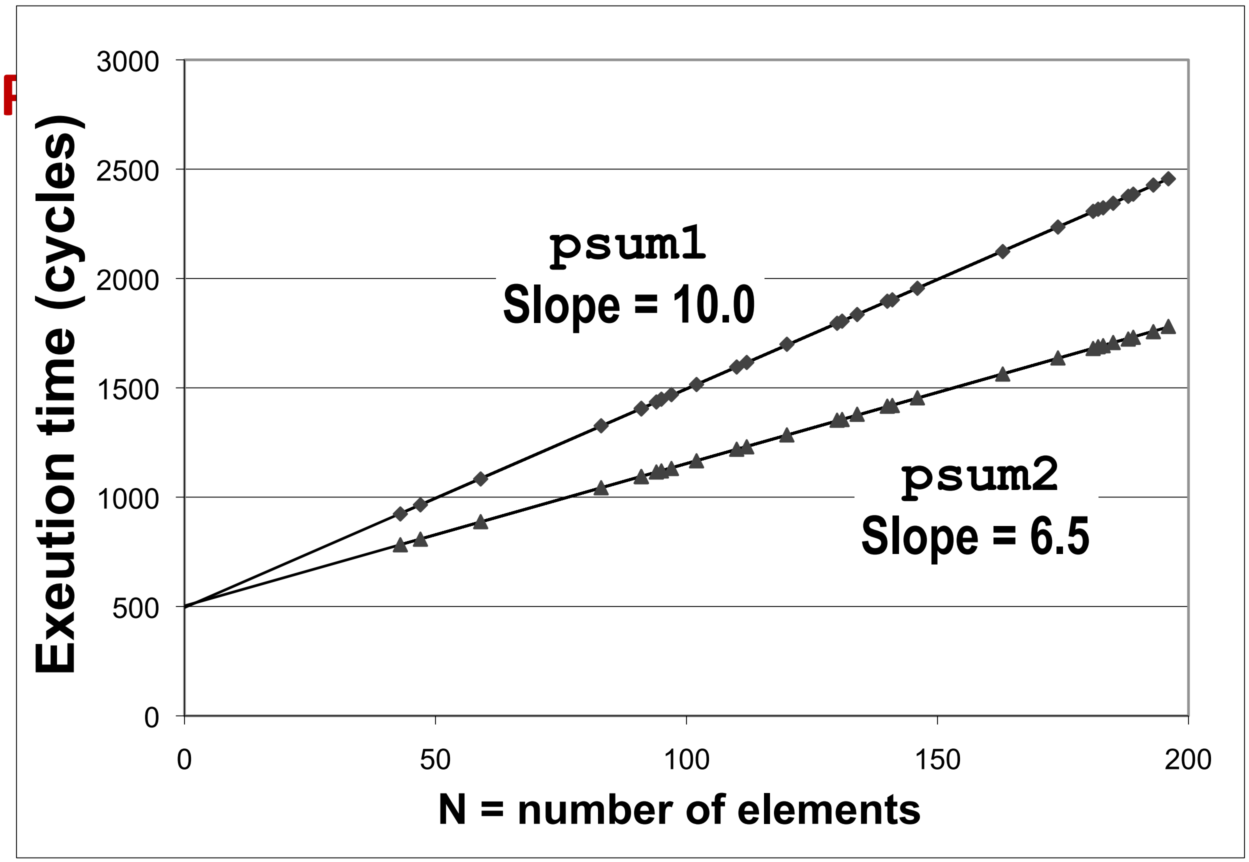
n = Length or number of elements to process

In our case:

- CPE = cycles per OP

Total Time =
CPE*n + Overhead

CPE =
slope of line



# Benchmark Performance

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

Compute sum or product of vector elements

| | Integer | | Double FP | |
|---|---|---|---|---|
| Method | Add | Mult | Add | Mult |
| Combine1 (unoptimized) | 29.0 | 29.2 | 27.4 | 27.9 |
| Combine1 –O1 | 12.0 | 12.0 | 12.0 | 13.0 |

# Basic Optimizations

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

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

# Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

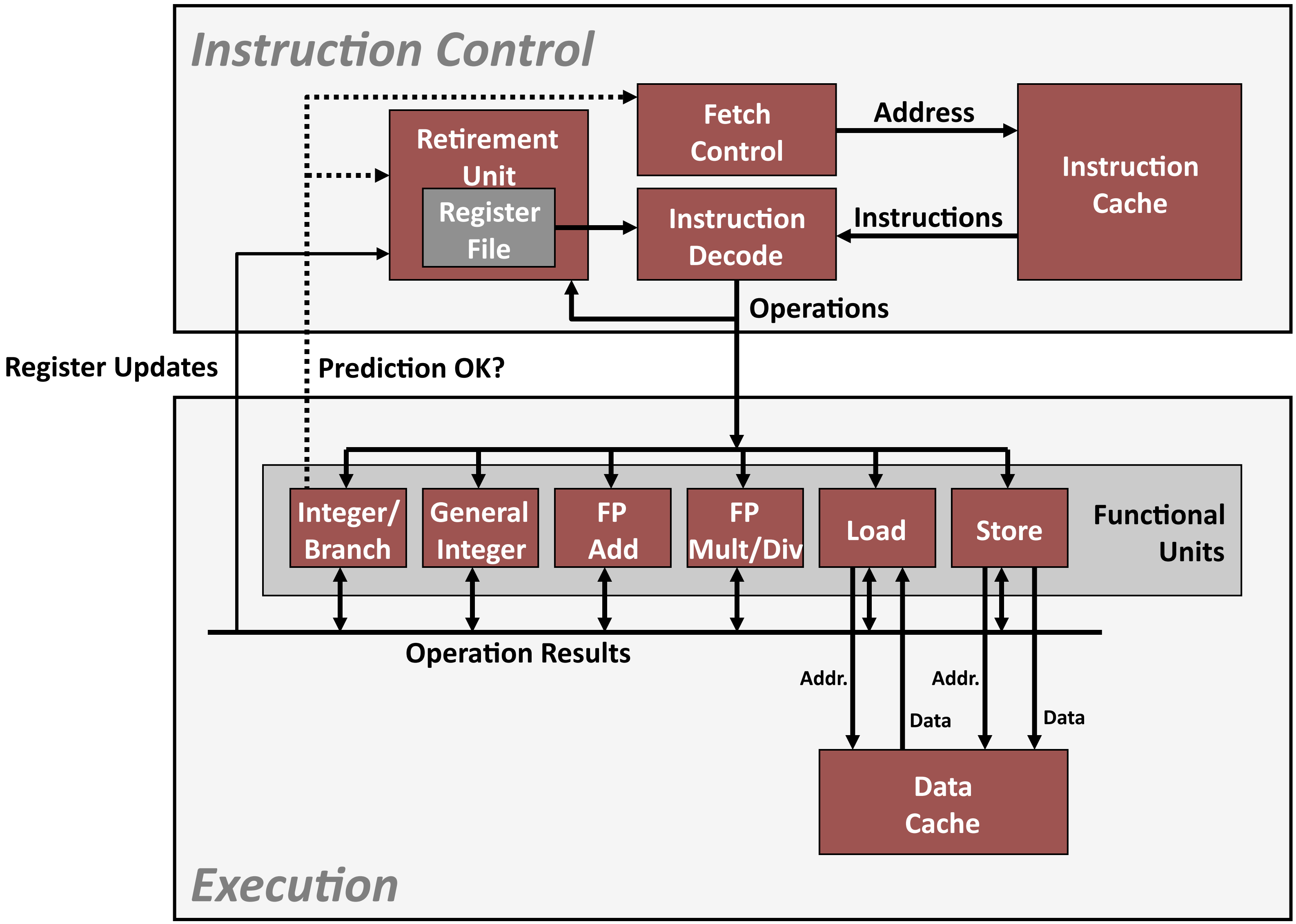
# Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| Combine1 –O1 | 12.0 | 12.0 | 12.0 | 13.0 |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |

**This eliminates sources of overhead in loop**

# Modern CPU Design



# Superscalar Processor

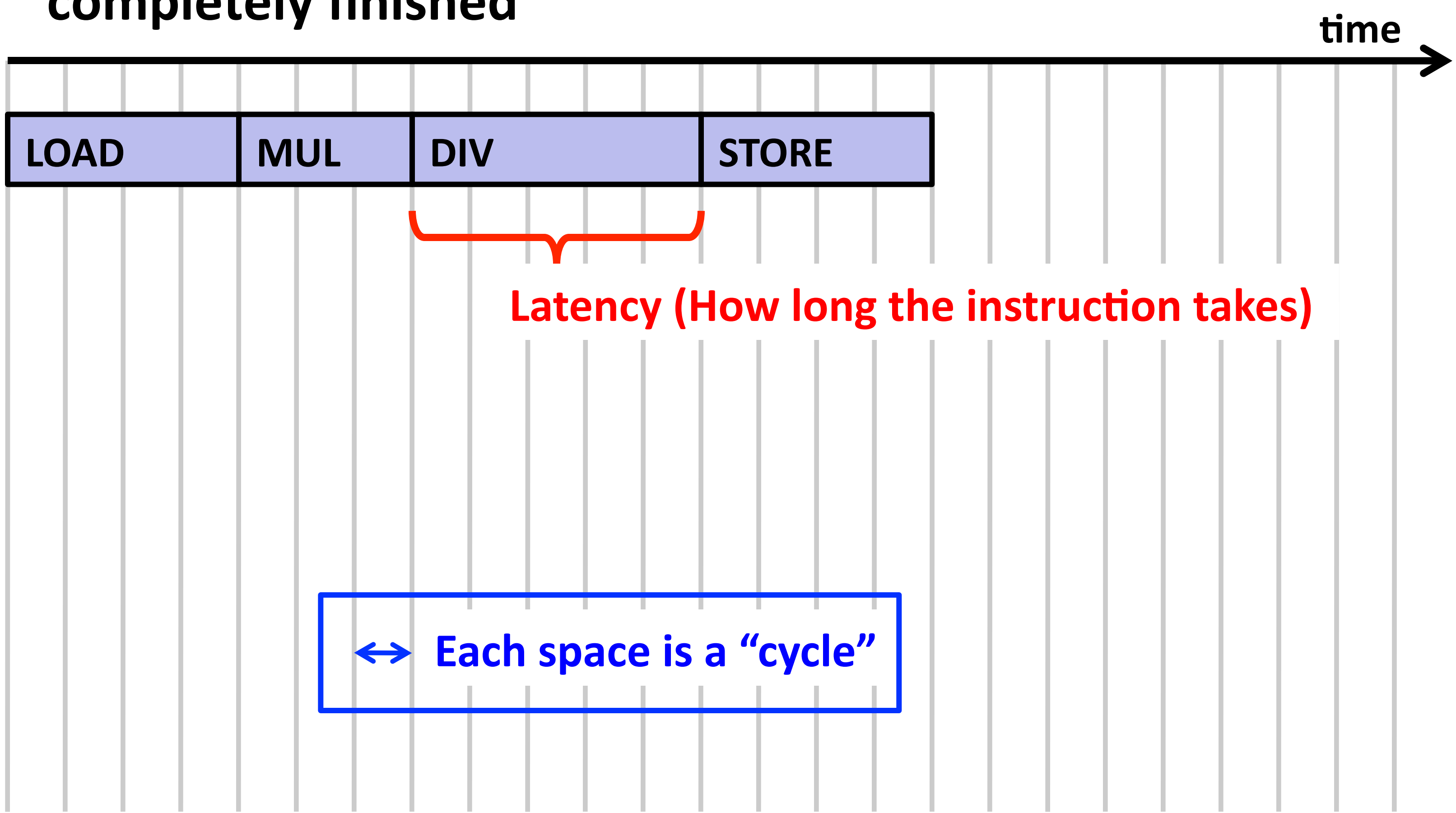
- **Definition:** **A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.**

- **Benefit: Without programming effort, a superscalar processor can take advantage of the *instruction level parallelism* that most programs have**

- **Most CPUs since about 1998 are superscalar.**
- **Intel: since Pentium Pro**

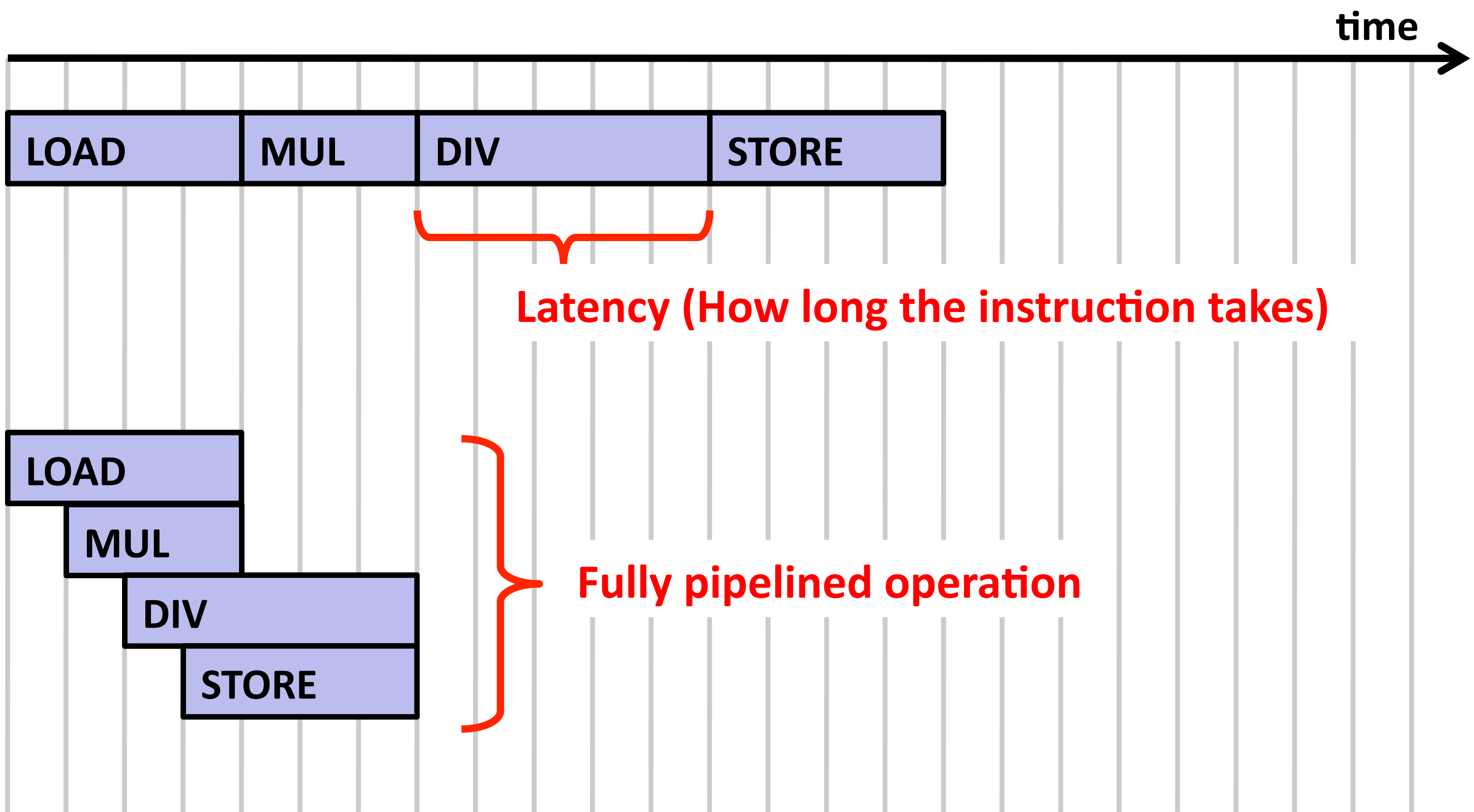
# Basic Instruction Execution

- Each instruction takes some time to execute
- We don't start one instruction until the previous one has completely finished



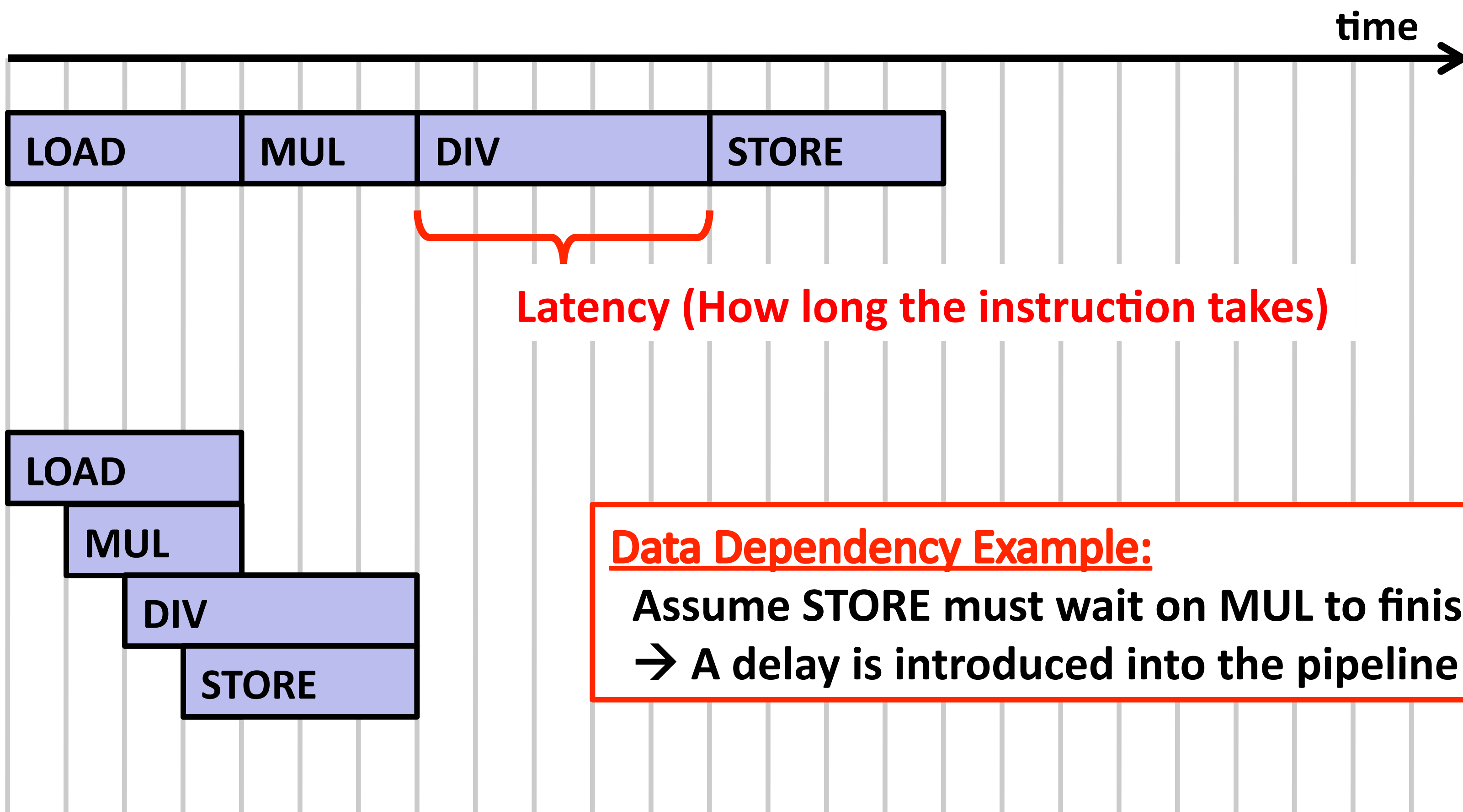
# Pipelined Instruction Execution

- With pipelining, we can start a new instruction every cycle
- We can execute several instructions in parallel



# Pipelined Instruction Execution

- Sometimes we cannot start next instruction immediately
- Data dependencies



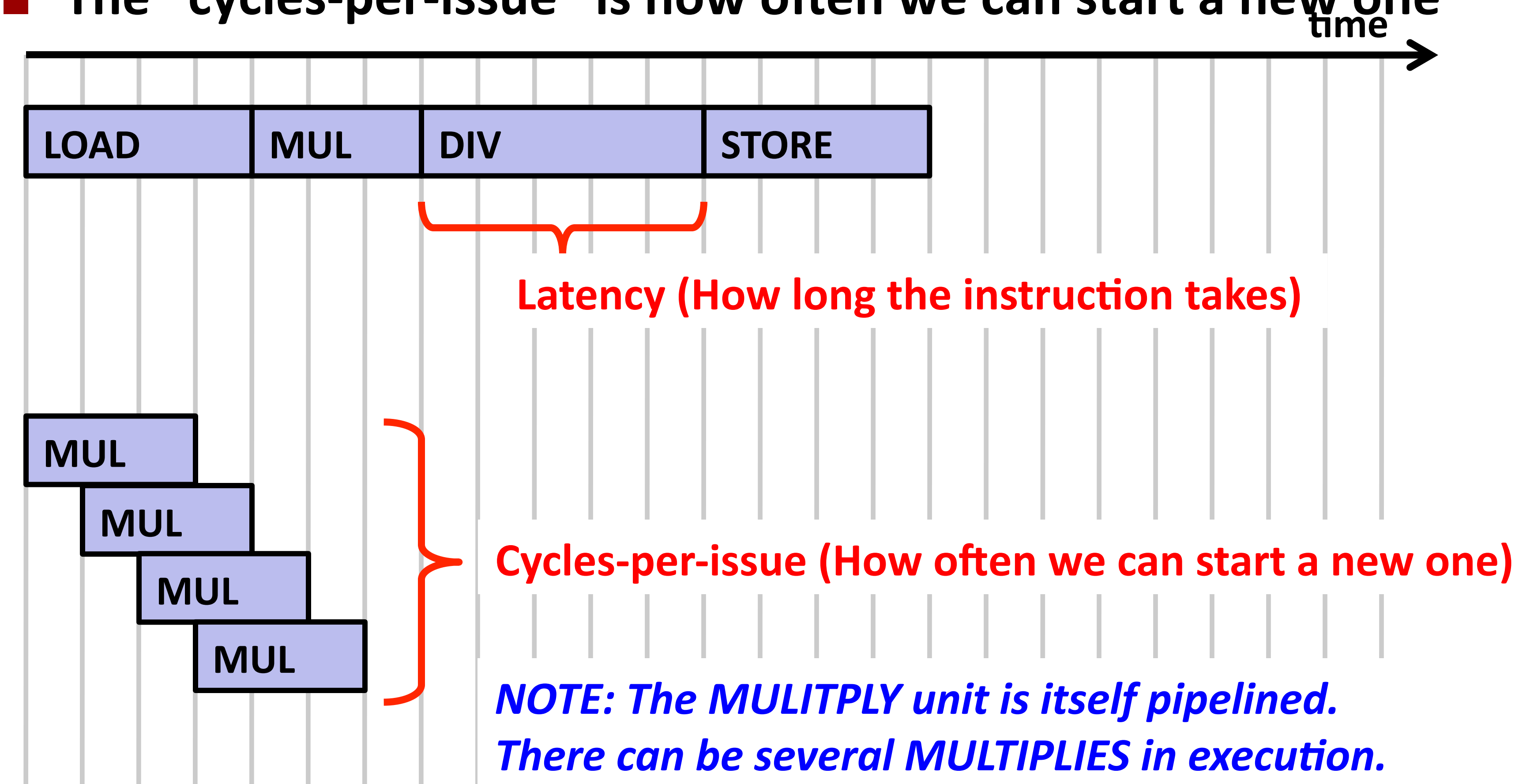

**Data Dependency Example:**

Assume STORE must wait on MUL to finish

→ A delay is introduced into the pipeline

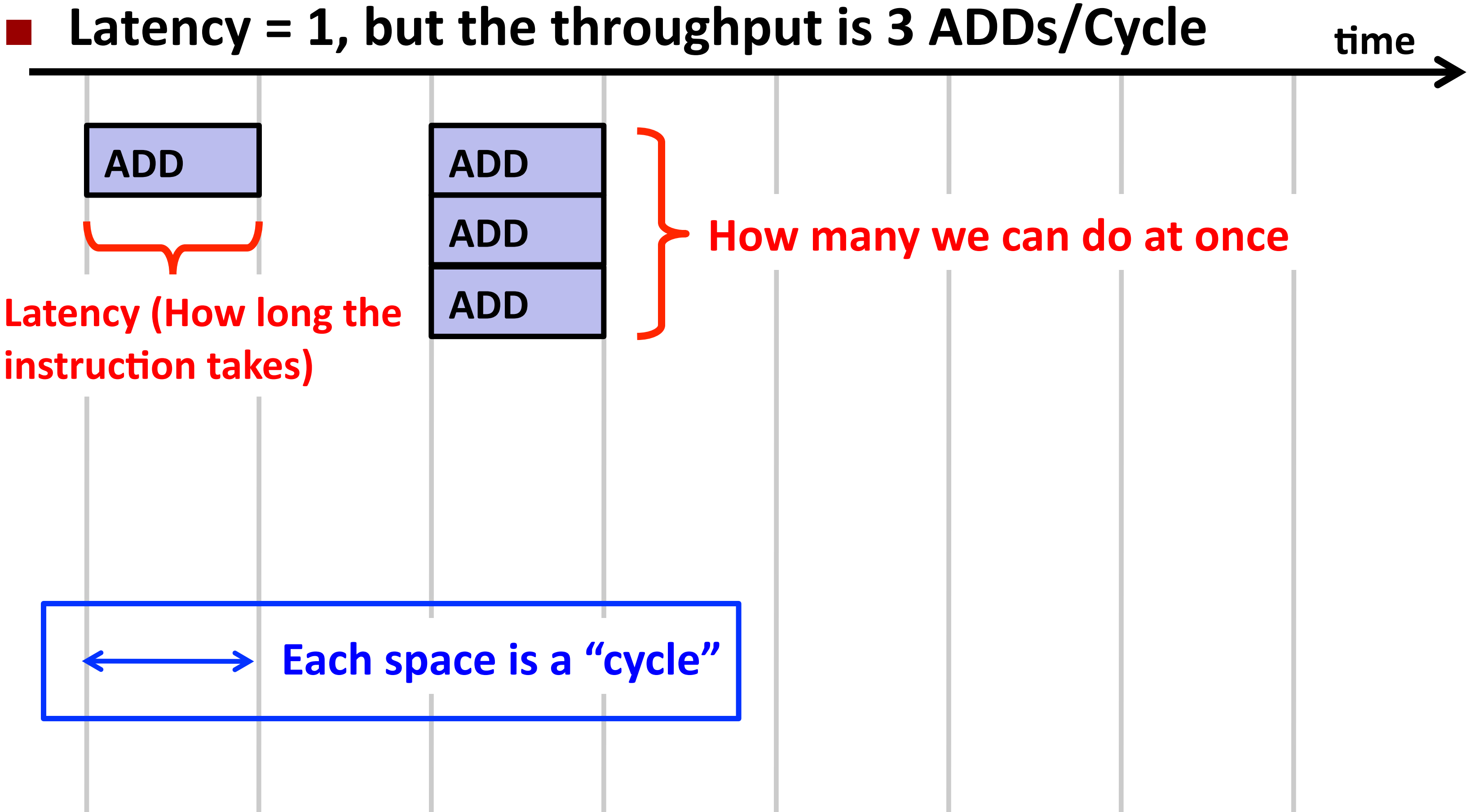
# Latency and Cycles-Per-Issue

- Even though a unit (e.g., MUL) takes several cycles, *it is itself pipelined*.
- The "cycles-per-issue" is how often we can start a new one



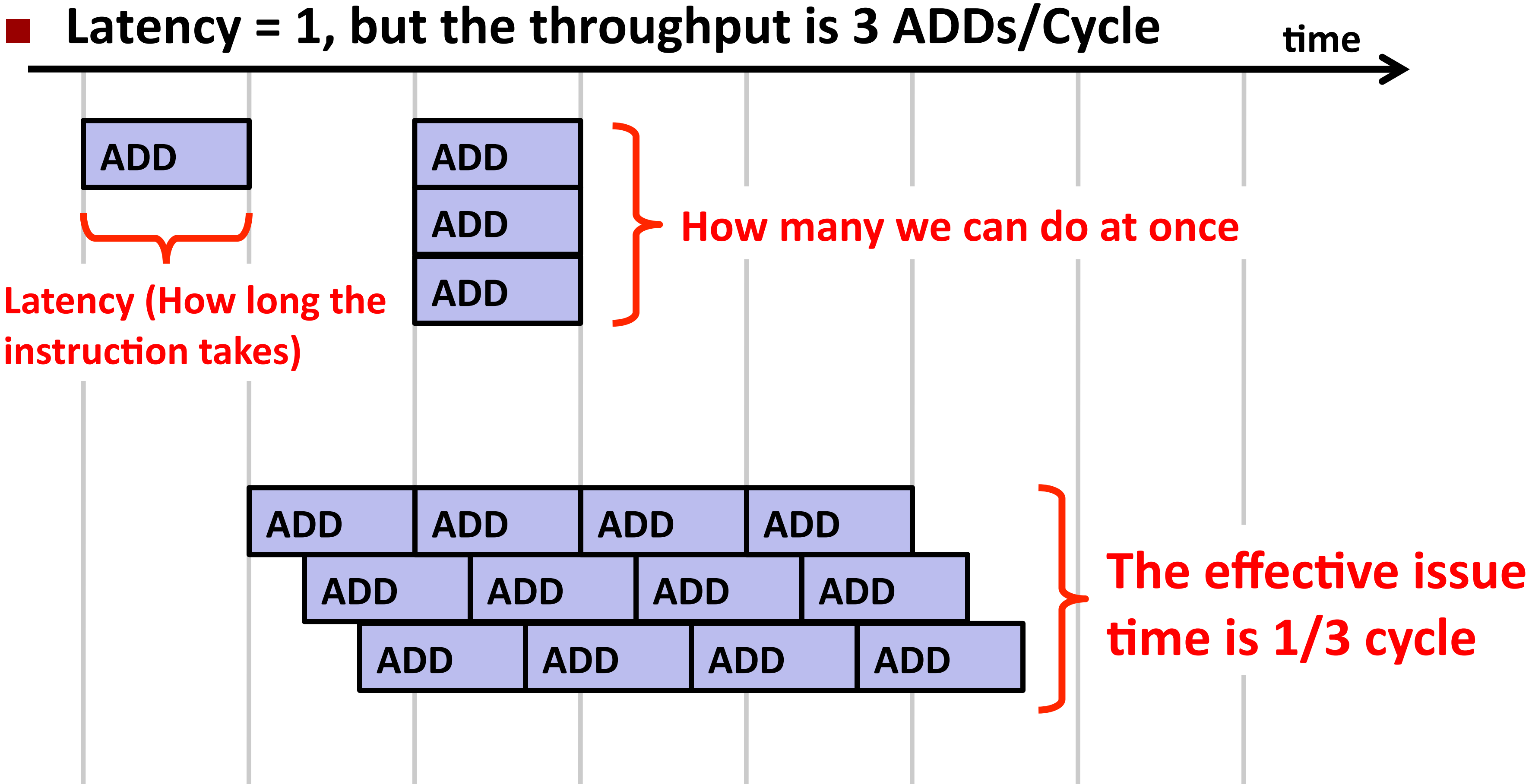
# Integer ADD: More than one ADD unit

- There are several (e.g., 3) addition units
- Three ADDs can be started or executed at once.
- Latency = 1, but the throughput is 3 ADDs/Cycle



# Integer ADD: More than one ADD unit

- There are several (e.g., 3) addition units
- Three ADDs can be started or executed at once.
- Latency = 1, but the throughput is 3 ADDs/Cycle



# Nehalem CPU

- **Multiple instructions can execute in parallel**
  1 load, with address computation
  1 store, with address computation
  2 simple integer (one may be branch)
  1 complex integer (multiply/divide)
  1 FP Multiply
  1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

| *Instruction* | *Latency* | *Cycles/Issue* |
|---|---|---|
| Load / Store | 4 | 1 |
| Integer Multiply | 3 | 1 |
| **Integer/Long Divide** | **11--21** | **11--21** |
| Single/Double FP Multiply | 4/5 | 1 |
| Single/Double FP Add | 3 | 1 |
| **Single/Double FP Divide** | **10--23** | **10--23** |

# x86-64 Compilation of Combine4

- **Let's look at one case: Integer Multiply**
- **Look at the inner loop.**

```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

# x86-64 Compilation of Combine4

- Let's look at one case: Integer Multiply
- Look at the inner loop.

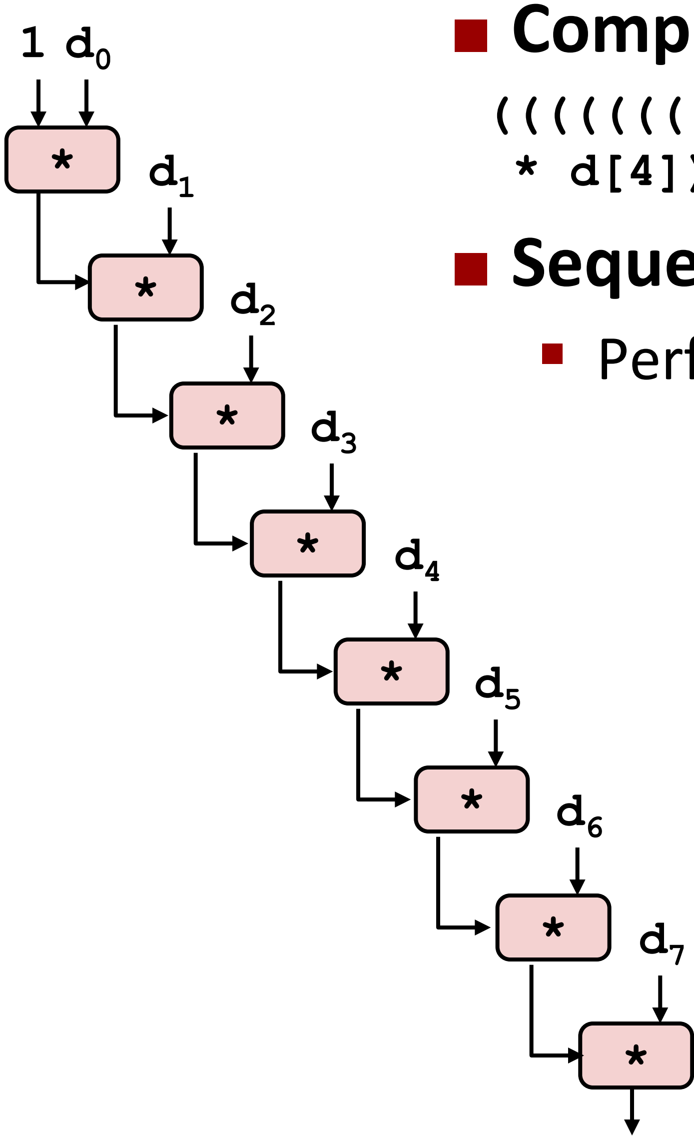
```
.L519:                                  # Loop:
        imull   (%rax,%rdx,4),%ecx      # t = t * d[i]
        addq    $1,%rdx                 # i++
        cmpq    %rdx,%rbp               # Compare length:i
        jg      .L519                   # If >, goto Loop
```

| | Integer | | Double FP | |
|---|---|---|---|---|
| Method | Add | Mult | Add | Mult |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |
| Latency Bound | 1.0 | 3.0 | 3.0 | 5.0 |

It seems limited by the MUL instruction…

Can we make it go any faster?

# Combine4 = Serial Computation (OP = *)




- **Computation (length=8)**

```
((((((((1 * d[0]) * d[1]) * d[2]) * d[3])
 * d[4]) * d[5]) * d[6]) * d[7])
```

- **Sequential dependence**
  - Performance determined by latency of OP

# Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

# Effect of Loop Unrolling

| | Integer | | Double FP | |
|---|---|---|---|---|
| Method | Add | Mult | Add | Mult |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |
| Unroll 2x | 2.0 | 1.5 | 3.0 | 5.0 |
| Latency Bound | 1.0 | 3.0 | 3.0 | 5.0 |

- **Helps integer multiply**
  - Below latency bound
  - Compiler does clever optimization
- **Others don't improve. *Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

# Loop Unrolling with Reassociation

```
void unroll2aa_combine(vec_ptr v, data_t *dest) {
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       x = x OP d[i];
    }
    *dest = x;
}
```

Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

- **Can this change the result of the computation?**
- **Yes, for Floating Point. *Why?***

# Effect of Reassociation

| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |
| Unroll 2x | 2.0 | 1.5 | 3.0 | 5.0 |
| Unroll 2x, reassociate | 2.0 | 1.5 | 1.5 | 3.0 |
| Latency Bound | 1.0 | 3.0 | 3.0 | 5.0 |
| Throughput Bound | 1.0 | 1.0 | 1.0 | 1.0 |

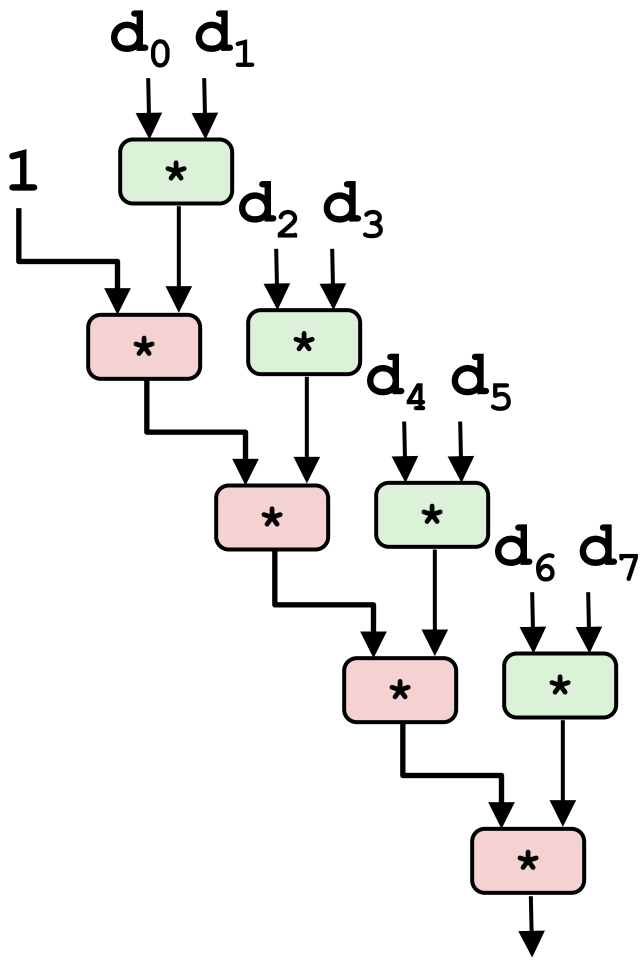
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
x = x OP (d[i] OP d[i+1]);
```

  - Why is that? (next slide)

# Reassociated Computation

```
x = x OP (d[i] OP d[i+1]);
```




- **What changed:**
  - Ops in the next iteration can be started early (no dependency)
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - Measured CPE slightly worse for FP mult

# Loop Unrolling with Separate Accumulators

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- Different form of reassociation

# Effect of Separate Accumulators

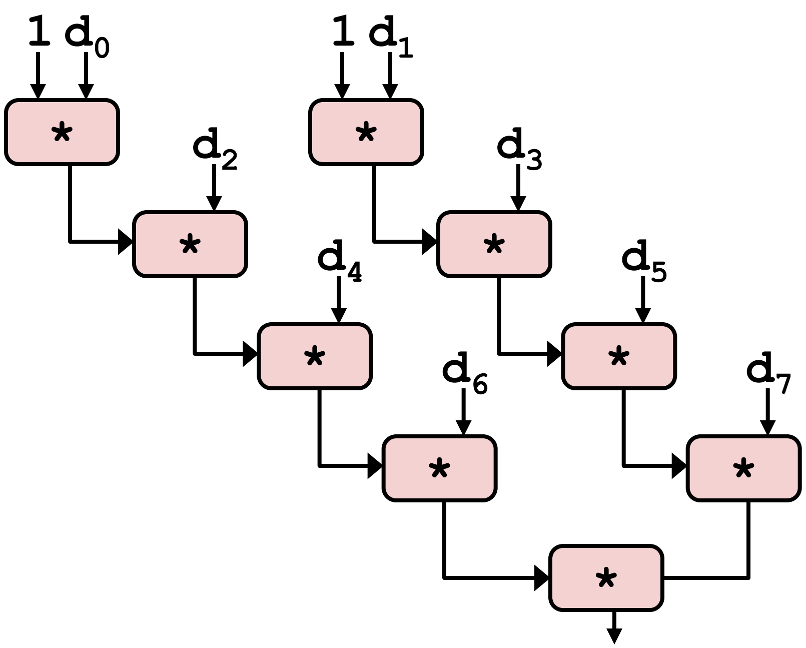
| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |
| Unroll 2x | 2.0 | 1.5 | 3.0 | 5.0 |
| Unroll 2x, reassociate | 2.0 | 1.5 | 1.5 | 3.0 |
| Unroll 2x Parallel 2x | 1.5 | 1.5 | 1.5 | 2.5 |
| Latency Bound | 1.0 | 3.0 | 3.0 | 5.0 |
| Throughput Bound | 1.0 | 1.0 | 1.0 | 1.0 |

- **2x speedup (over unroll2) for Int *, FP +, FP ***
  - Breaks sequential dependency in a "cleaner," more obvious way

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

# Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```




- **What changed:**
  - Two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles:
    **CPE = D/2**
  - CPE matches prediction!

  ***What Now?***

# Unrolling & Accumulating

- **Idea**
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K

- **Limitations**
  - Diminishing returns
    - Cannot go beyond throughput limitations of execution units
  - Large overhead for short lengths
    - Finish off iterations sequentially

# Unrolling & Accumulating: Double *

- **Case**
  - Intel Nehelam (Shark machines)
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 1.00

*Accumulators*

| FP * | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | 5.00 | | |
| 2 | | 2.50 | | 2.50 | | 2.50 | | |
| 3 | | | 1.67 | | | | | |
| 4 | | | | 1.25 | | 1.25 | | |
| 6 | | | | | 1.00 | | | 1.19 |
| 8 | | | | | | 1.02 | | |
| 10 | | | | | | | 1.01 | |
| 12 | | | | | | | | 1.00 |

# Unrolling & Accumulating: Int +

- **Case**
  - Intel Nehelam (Shark machines)
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

Accumulators

| FP * | Unrolling Factor L | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| K | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 12 |
| 1 | 2.00 | 2.00 | 1.00 | 1.01 | 1.02 | 1.03 | | |
| 2 | | 1.50 | | 1.26 | | 1.03 | | |
| 3 | | | 1.00 | | | | | |
| 4 | | | | 1.00 | | 1.24 | | |
| 6 | | | | | 1.00 | | | 1.02 |
| 8 | | | | | | 1.03 | | |
| 10 | | | | | | | 1.01 | |
| 12 | | | | | | | | 1.09 |

# Achievable Performance

| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| **Scalar Optimum** | 1.00 | 1.00 | 1.00 | 1.00 |
| **Latency Bound** | 1.00 | 3.00 | 3.00 | 5.00 |
| **Throughput Bound** | 1.00 | 1.00 | 1.00 | 1.00 |

- **Limited only by throughput of functional units**
- **Up to 29X improvement over original, unoptimized code**

# Using Vector Instructions

| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| **Scalar Optimum** | 1.00 | 1.00 | 1.00 | 1.00 |
| **Vector Optimum** | 0.25 | 0.53 | 0.53 | 0.57 |
| **Latency Bound** | 1.00 | 3.00 | 3.00 | 5.00 |
| **Throughput Bound** | 1.00 | 1.00 | 1.00 | 1.00 |
| **Vec Throughput Bound** | 0.25 | 0.50 | 0.50 | 0.50 |

- **Make use of SSE Instructions**
  - Parallel operations on multiple data elements
  - See Web Aside OPT:SIMD on CS:APP web page

# What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

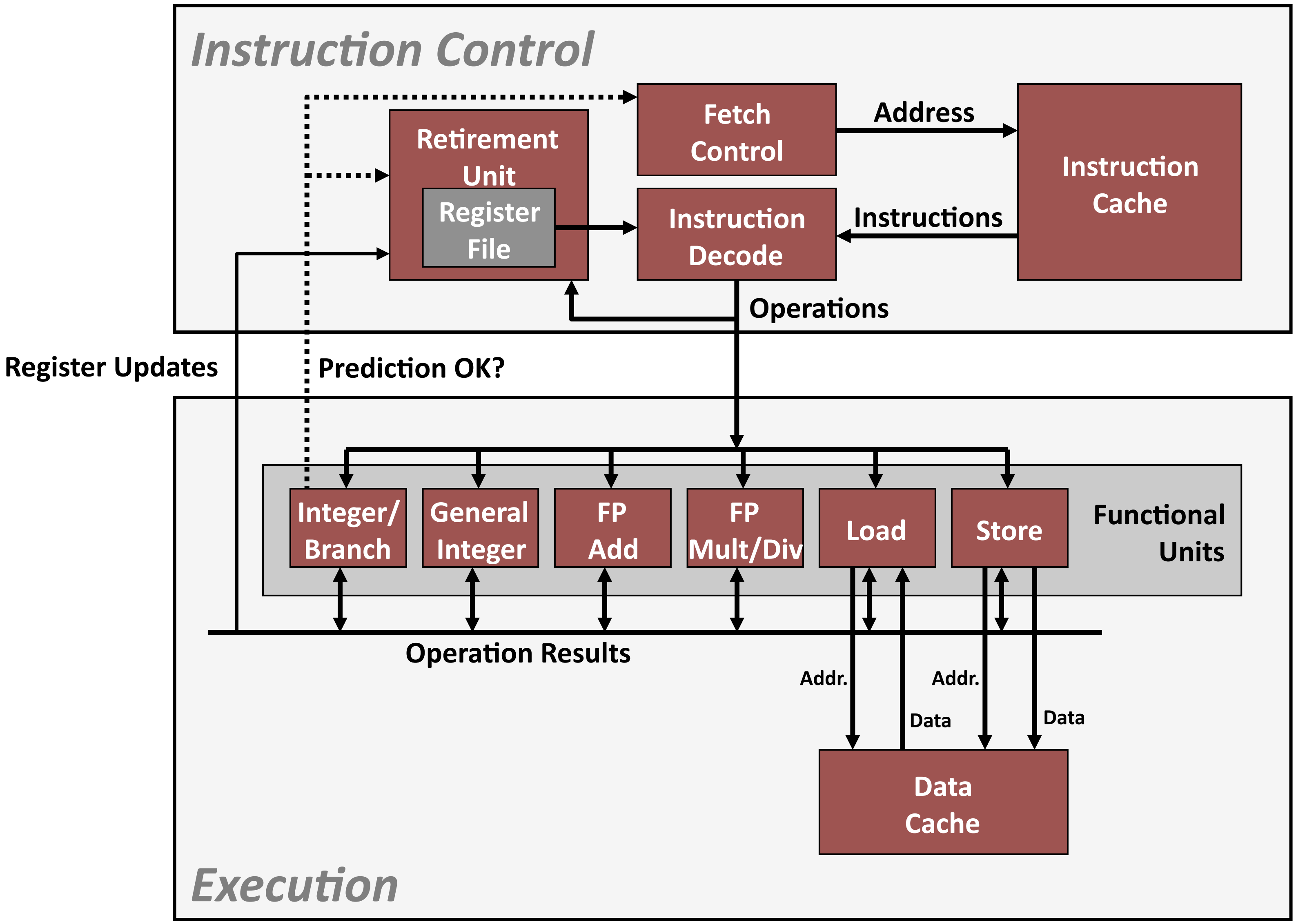
```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%eax,%edx,4),%ecx
```

Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

# Modern CPU Design



# Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%eax,%edx,4),%ecx
```

**Branch Not-Taken**

**Branch Taken**

```
8048a25:  cmpl    %edi,%edx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%ebp),%eax
8048a2c:  leal    0xffffffe8(%ebp),%esp
8048a2f:  movl    %ecx,(%eax)
```

# Branch Prediction

**Idea:**

- Guess which way branch will go
- Begin executing instructions at predicted position
  - But don't actually modify register or memory data

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
. . .
```

**Predict Taken**

```
8048a25:  cmpl    %edi,%edx
8048a27:  jl      8048a20
8048a29:  movl    0xc(%ebp),%eax
8048a2c:  leal    0xffffffe8(%ebp),%esp
8048a2f:  movl    %ecx,(%eax)
```

**Begin Execution**

# Branch Prediction Through Loop

*Assume*
*vector length = 100*

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

*i = 98*

**Predict Taken (OK)**

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

*i = 99*

**Predict Taken (Oops)**

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

*i = 100*

**Read invalid location**

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

*i = 101*

**Executed**

**Fetched**

# Branch Misprediction Invalidation

*Assume*
*vector length = 100*

```
80488b1:   movl    (%ecx,%edx,4),%eax
80488b4:   addl    %eax,(%edi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 98*

**Predict Taken (OK)**

```
80488b1:   movl    (%ecx,%edx,4),%eax
80488b4:   addl    %eax,(%edi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
```

*i = 99*

**Predict Taken (Oops)**

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

~~*i = 100*~~

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~

~~*i = 101*~~

**Invalidate**

# Branch Misprediction Recovery

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
80488bb:    leal    0xffffffe8(%ebp),%esp
80488be:    popl    %ebx
80488bf:    popl    %esi
80488c0:    popl    %edi
```

*i = 99*

**Definitely not taken**

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

# Effect of Branch Prediction

- **Loops**
  - Typically, only miss when they hit loop end
- **Checking code**
  - Reliably predicts that error won't occur

```
void combine4b(vec_ptr v,
               data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t acc = IDENT;
    for (i = 0; i < length; i++) {
      if (i >= 0 && i < v->len) {
          acc = acc OP v->data[i];
      }
    }
    *dest = acc;
}
```

| | Integer | | Double FP | |
|---|---|---|---|---|
| **Method** | **Add** | **Mult** | **Add** | **Mult** |
| Combine4 | 2.0 | 3.0 | 3.0 | 5.0 |
| Combine4b | 4.0 | 4.0 | 4.0 | 5.0 |

# Getting High Performance

- **Good compiler and flags**
- **Don't do anything stupid**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers:
      procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)