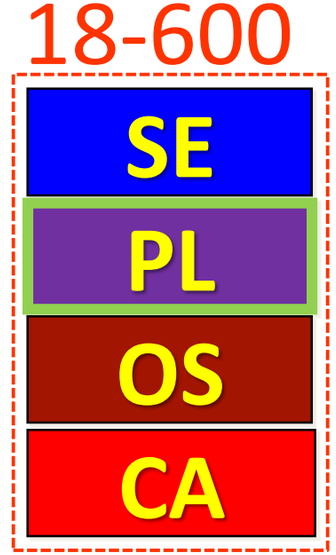
18-600 Foundations of Computer Systems

Lecture 18: "Program Performance Optimizations"

John P. Shen & Gregory Kesden

November 1, 2017



➢ Required Reading Assignment:
- **Chapter 5 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**

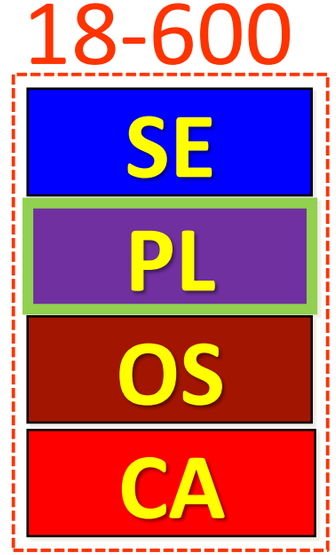
Electrical & Computer ENGINEERING

Carnegie Mellon University

18-600 Foundations of Computer Systems

Lecture 18: "Program Performance Optimizations"

- **Overview of Optimizing Compilers**
- **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**



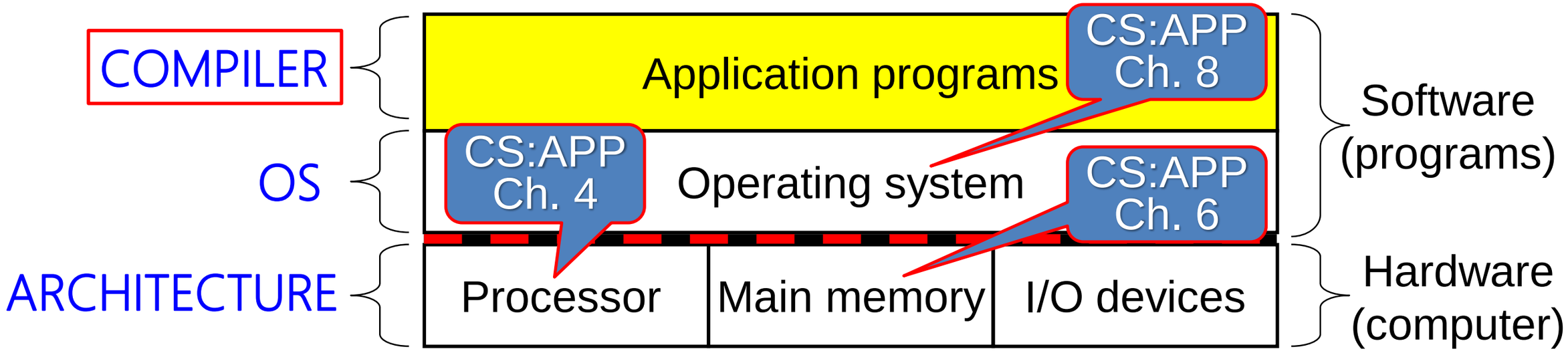
Electrical & Computer ENGINEERING

Carnegie Mellon University

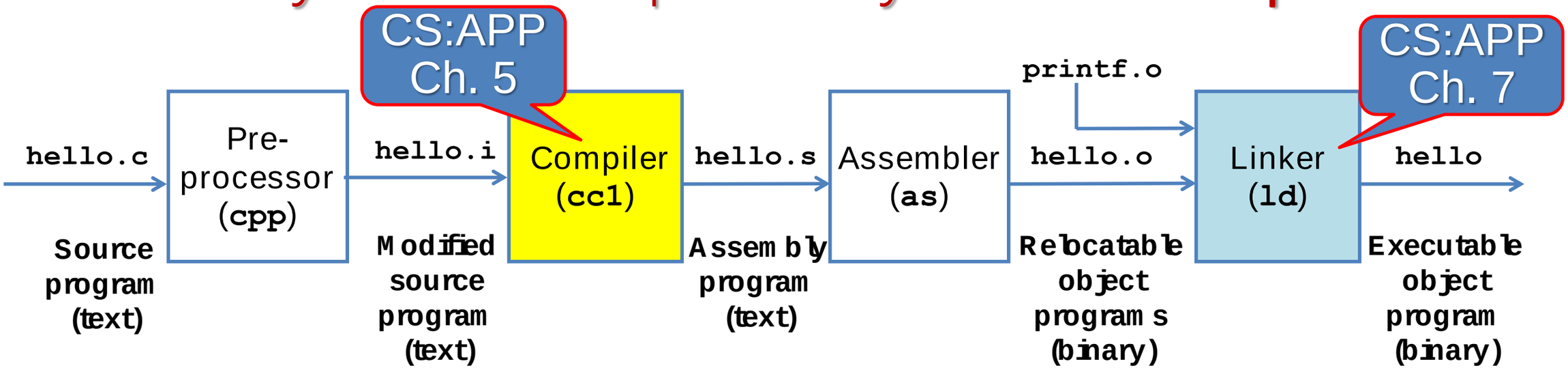
Anatomy of a Computer System: SW/HW

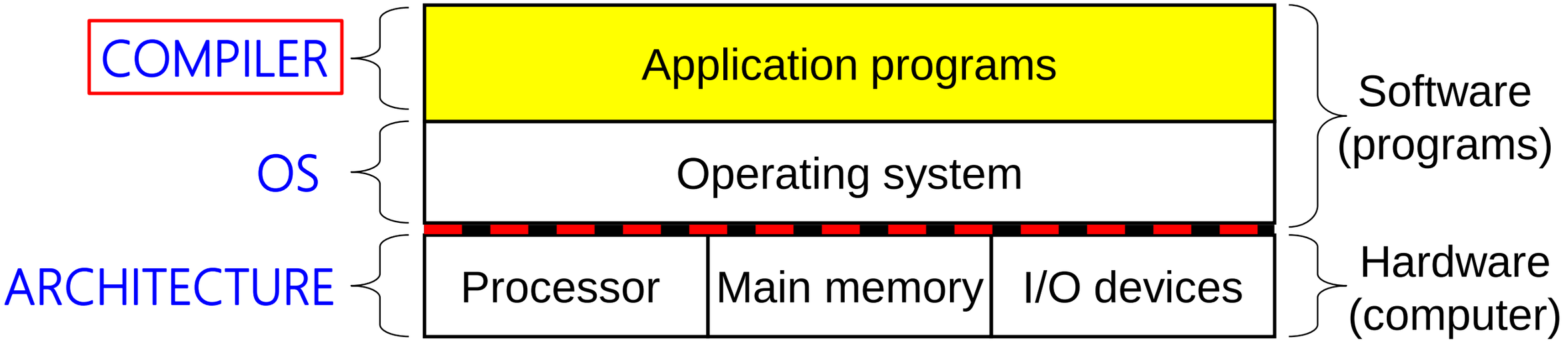
➢ What is a Computer System?

- ❖ Software + Hardware
- ❖ Programs + Computer ➔ [Application program + OS] + Computer
- ❖ Programming Languages + Operating Systems + Computer Architecture



Anatomy of a Computer System: Compiler





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 modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Optimizing Compilers

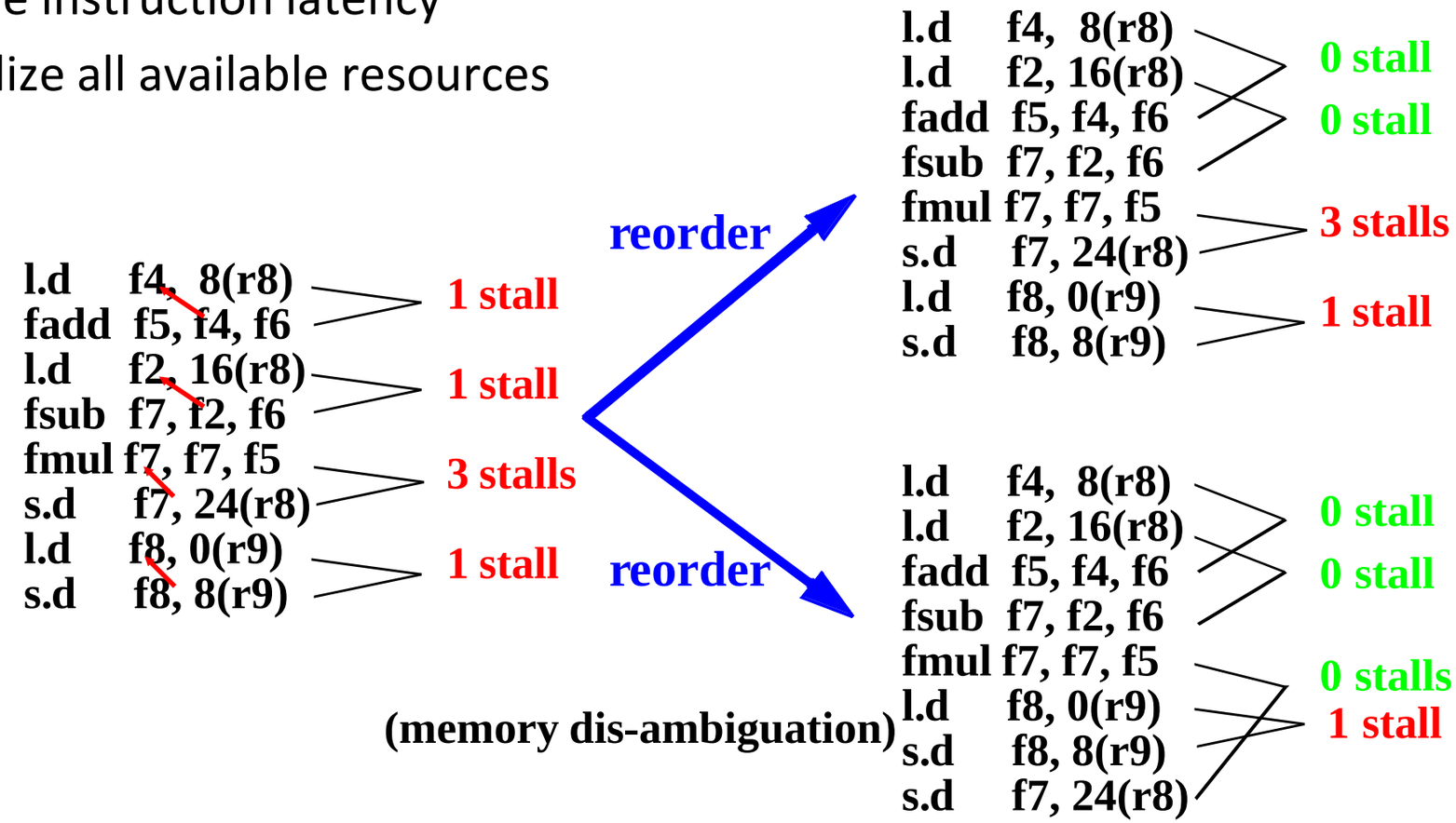
- **Provide efficient mapping of program to machine**
  - Register allocation
  - Code selection and ordering (scheduling)
  - Dead code elimination
  - Eliminating minor inefficiencies
- **Do not (usually) improve asymptotic efficiency**
  - Up to programmer to select best overall algorithm
  - Big-O 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

Limitations of Optimizing Compilers

- **Operate under fundamental constraint**
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that 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
  - Newer versions of GCC do inter-procedural analysis within individual files
    - But, not between code in different files
- **Most analysis is based only on *static* information**
  - Compiler has difficulty anticipating run-time inputs
- **When in doubt, the compiler must be conservative**

Code Scheduling

- **Rearrange code sequence to minimize execution time**
  - Hide instruction latency
  - Utilize all available resources



Code Scheduling

- **Objectives: minimize execution latency of the program**
  - Start as early as possible instructions on the critical path
  - Help expose more instruction-level parallelism to the hardware
  - Help avoid resource conflicts that increase execution time
- **Constraints**
  - Program Precedences (Dependences)
  - Machine Resources
- **Motivations**
  - Dynamic/Static Interface (DSI): By employing more software (static) optimization techniques at compile time, hardware complexity can be significantly reduced
  - Performance Boost: Even with the same complex hardware, software scheduling can provide additional performance enhancement over that of unscheduled code

Precedence Constraints

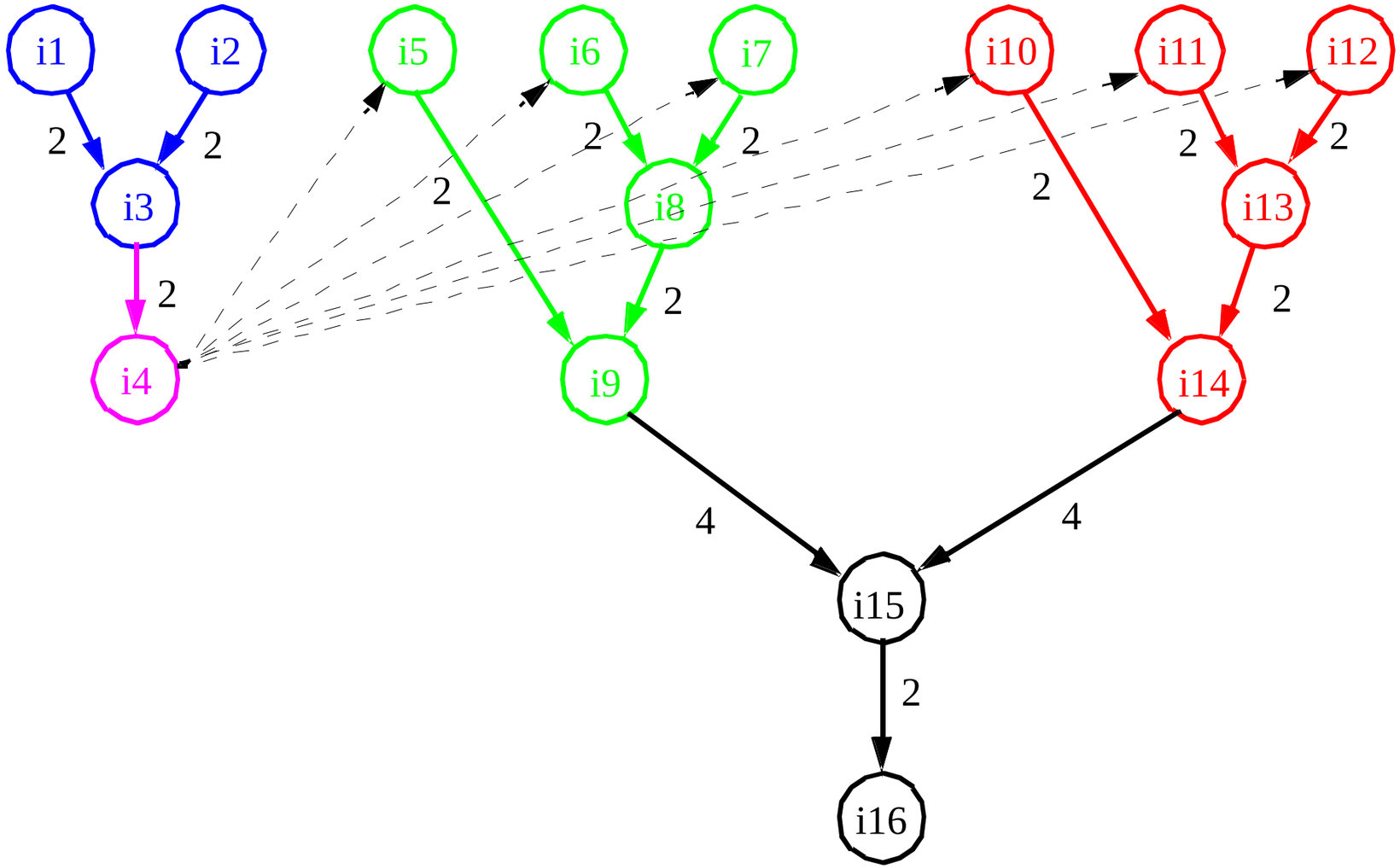
- **Minimum required ordering and latency between definition and use**
- **Precedence Graph**
  - Nodes: instructions
  - Edges (a→b): a precedes b
  - Edges are annotated with minimum latency

```
w[i+k].ip = z[i].rp + z[m+i].rp;
w[i+j].rp = e[k+1].rp*
            (z[i].rp -z[m+i].rp)
            e[k+1].ip *
            (z[i].ip - z[m+i].ip);
```

FFT code fragment

```
i1: l.s f2, 4(r2)
i2: l.s f0, 4(r5)
i3: fadd.s f0, f2, f0
i4: s.s f0, 4(r6)
i5: l.s f14, 8(r7)
i6: l.s f6, 0(r2)
i7: l.s f5, 0(r3)
i8: fsub.s f5, f6, f5
i9: fmul.s f4, f14, f5
i10: l.s f15, 12(r7)
i11: l.s f7, 4(r2)
i12: l.s f8, 4(r3)
i13: fsub.s f8, f7, f8
i14: fmul.s f8, f15, f8
i15: fsub.s f8, f4, f8
i16: s.s f8, 0(r8)
```

Precedence Graph

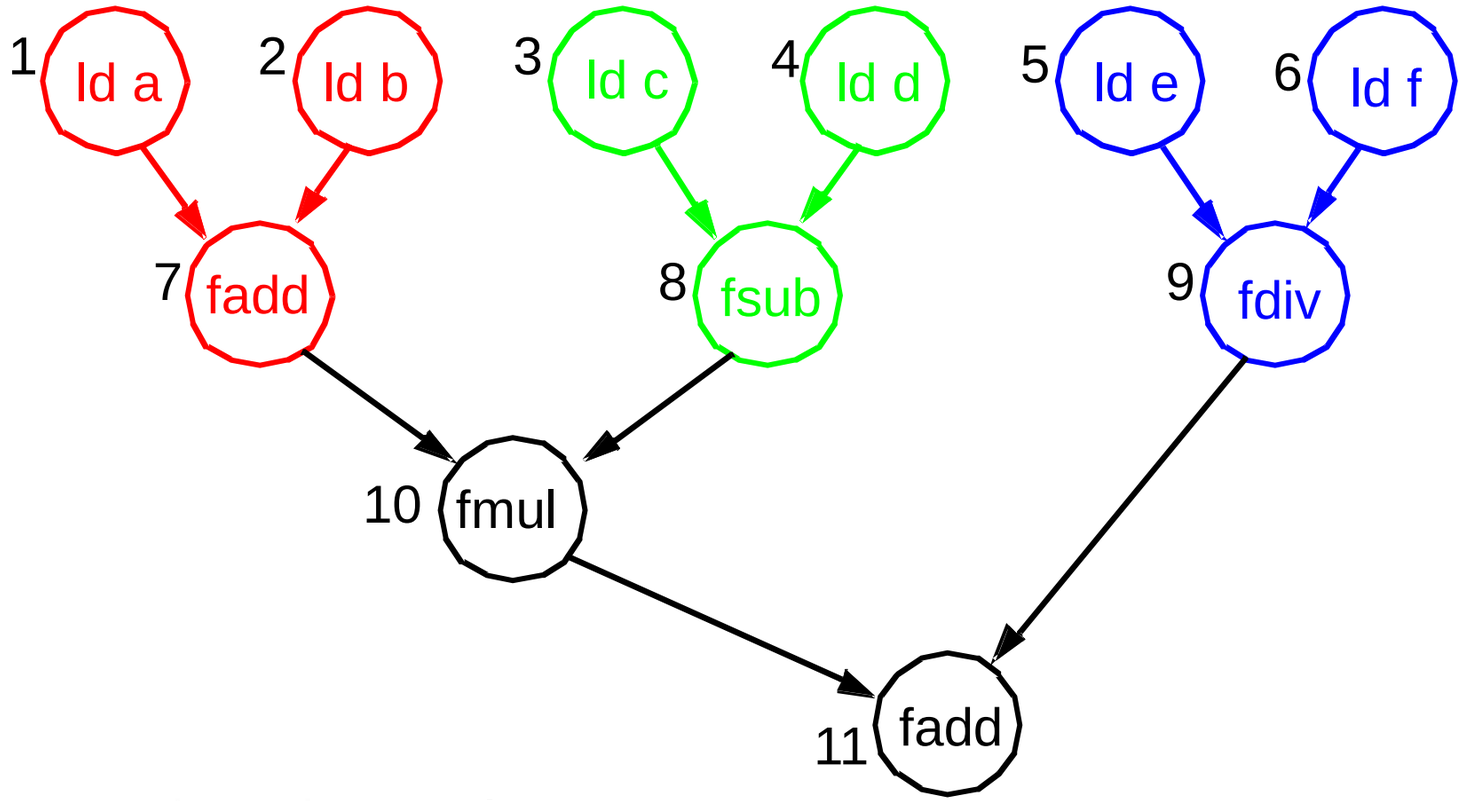


List Scheduling for Basic Blocks

- **Initialize ready list that holds all ready instructions**
  Ready = data ready and can be scheduled
- **Choose one ready instruction *R* from ready list with the highest priority**
  - Number of descendants in precedence graph
  - Maximum latency from root node of precedence graph
  - Length of operation latency
  - Ranking of paths based on importance
  - Combination of above
- **Insert *R* into schedule**
  Making sure resource constraints are satisfied
- **Add those instructions whose precedence constraints are now satisfied into the ready list**
- **Can be applied in the forward or backward direction**

List Scheduling Example

(a + b) * (c - d) + e/f



load: 2 cycles
add: 1 cycle
sub: 1 cycle
mul: 4 cycles
div: 10 cycles

orientation: cycle
direction: backward
heuristic: maximum latency to root

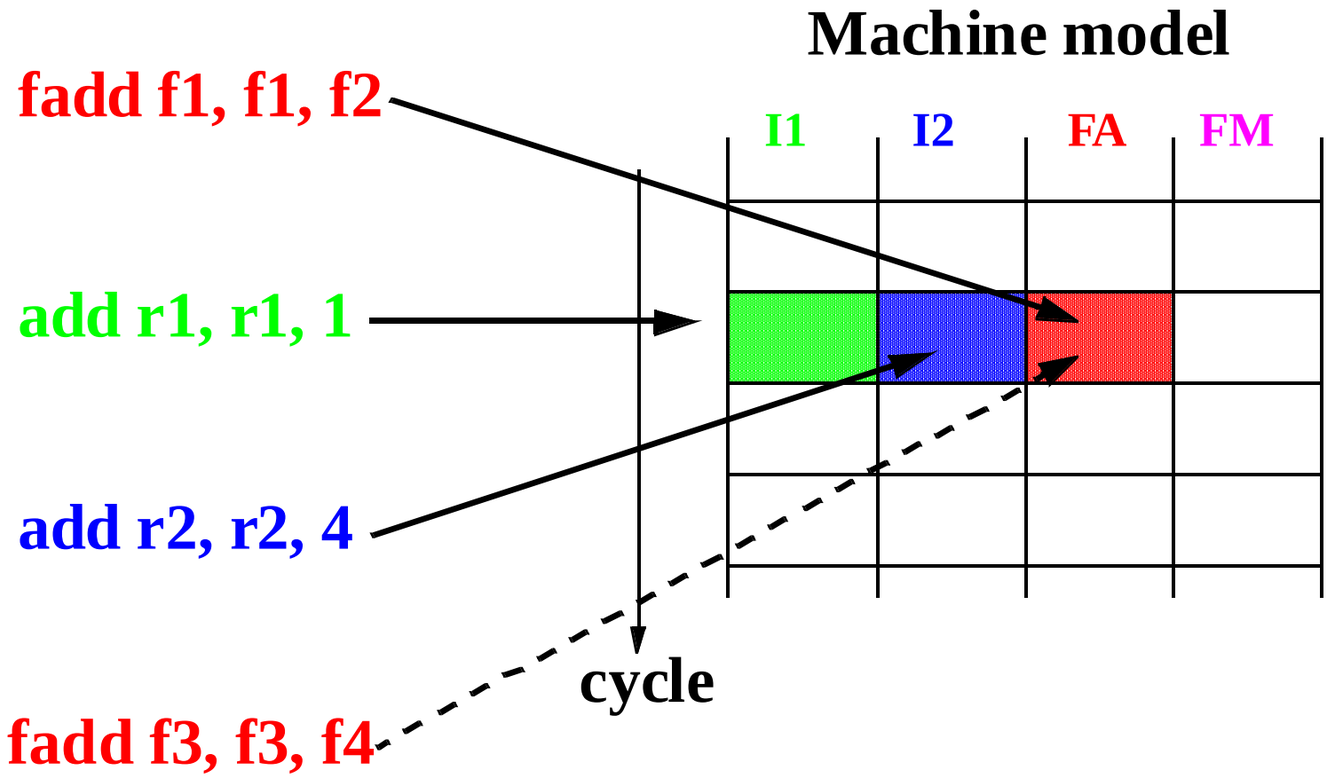
List Scheduling Example

Cycle	Ready list	Schedule	Code
1	6	6	ld f
2	5 6	5	ld e
3	4 5 6	4	ld d
4	4 9	9	fdiv (e/f)
5	3 4 9	3	ld c
6	2 3 4 9	2	ld b
7	1 2 3 4 9	1	ld a
8	1 2 8 9	8	fsub (c – d)
9	7 8 9	7	fadd (a + b)
10	9 10	10	fmul
11	9 10		nop
12	9 10		nop
13	9 10		nop
14	11	11	fadd

green *means candidate and ready*
red means candidate but not yet ready

Resource Constraints

- **Bookkeeping**
  - Prevent resources from being oversubscribed



ILP List Scheduling Example

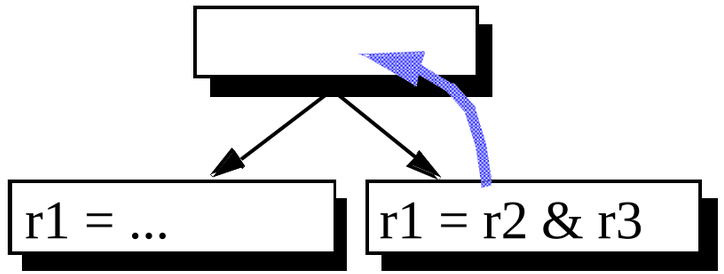
Cycle	Ready list	Schedule	Resources			Code	
			I	F	FD		
1	6	6	X			ld f	
2	5 6	5	X			ld e	
3	5 6						
4	4 9	4 9	X		X	fdiv (e/f)	ld d
5	3 4 9	3	X			ld c	
6	2 3 4 9	2	X			ld b	
7	1 2 3 4 9	1	X			ld a	
8	1 2 8 9	8		X		fsub (c – d)	
9	7 8 9	7		X		fadd (a + b)	
10	9 10	10		X		fmul	
11	9 10					nop	
12	9 10					nop	
13	9 10					nop	
14	11	11		x		fadd	

Limitations of List Scheduling

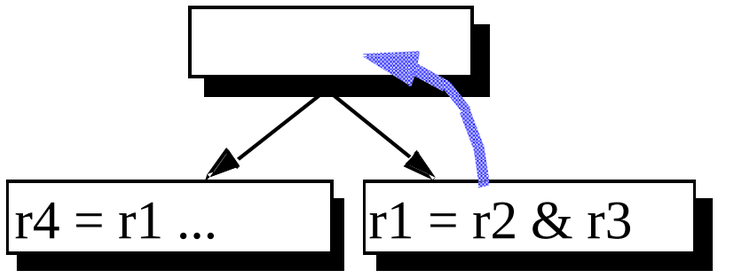
- **Cannot move instructions past conditional branch instructions in the program (scheduling limited by basic block boundaries)**
- **Problem: Many programs have small numbers of instructions (4-5) in each basic block. Hence, not much code motion is possible**
- **Solution: Allow code motion across basic block boundaries.**
- **Speculative Code Motion: "jumping the gun"**
  - Execute instructions before we know whether or not we need to
  - Utilize otherwise idle resources to perform work which we speculate will need to be done
- **Relies on program profiling to make intelligent decisions about speculation**

Types of Speculative Code Motion

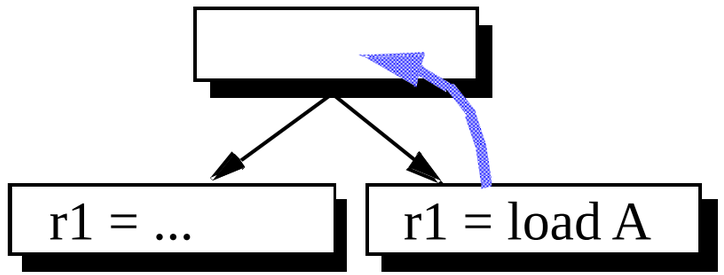
- **Two characteristics of speculative code motion:**
  - **Safety**, which indicates whether or not spurious exceptions may occur
  - **Legality**, which indicates correctness of results
- **Four possible types of code motion:**



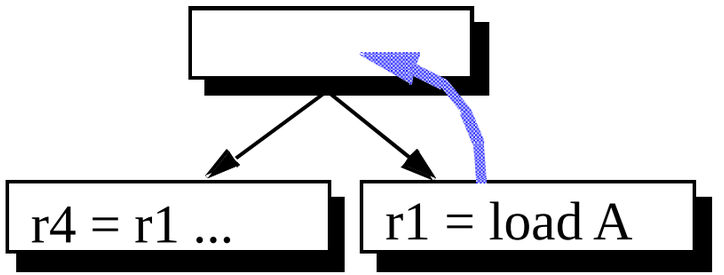
(a) safe and legal



(b) illegal



(c) unsafe



(d) unsafe and illegal

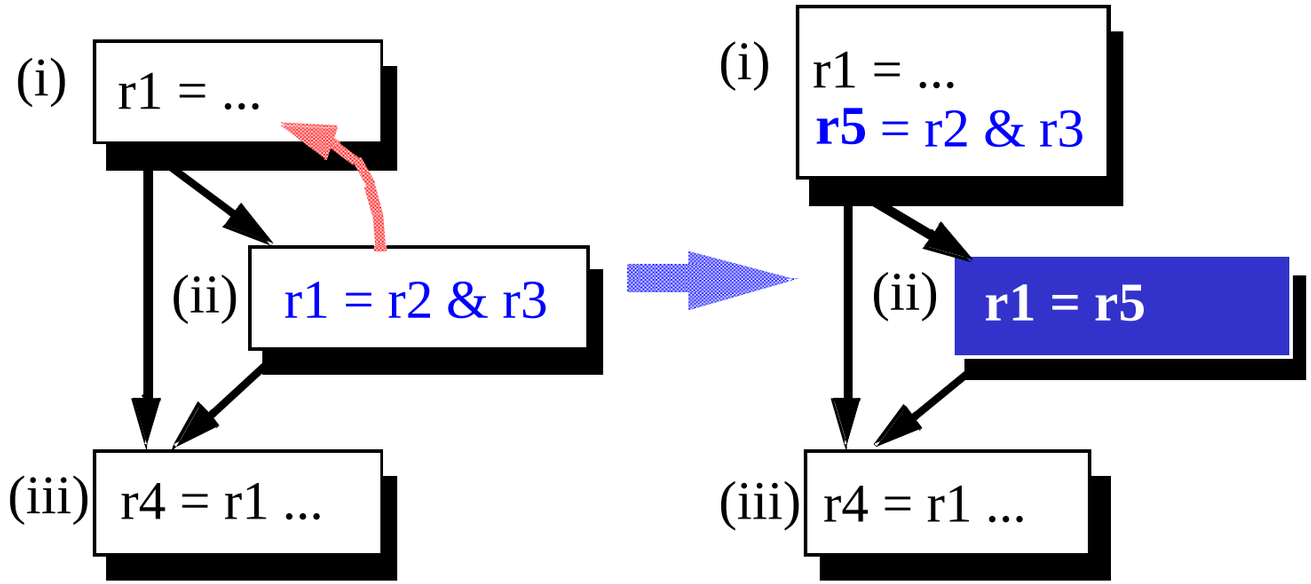
Register Renaming

- **Prevents boosted instructions from overwriting register state needed on alternate execution path.**
- **Utilizes idle (non-live) registers (r6 in example below).**

BB#	Original Code			Scheduled Code		
n		load	r4= ...		load	r4= ...
		load	r5= ...		load	r5= ...
		cmpi	c0,r4,10		cmpi	c0,r4,10
		add	r4=r4+r5		add	r4=r4+r5
		<stall>			sub	r3=r7-r4
		<stall>			and	r6=r3&r5
		bc	c0, A1		bc	c0, A1
n+1		st	... =r4		st	... =r4
n+2	A1:	sub	r3=r7-r4	A1:	st	... =r6
		and	r4=r3&r5			
		st	... =r4			

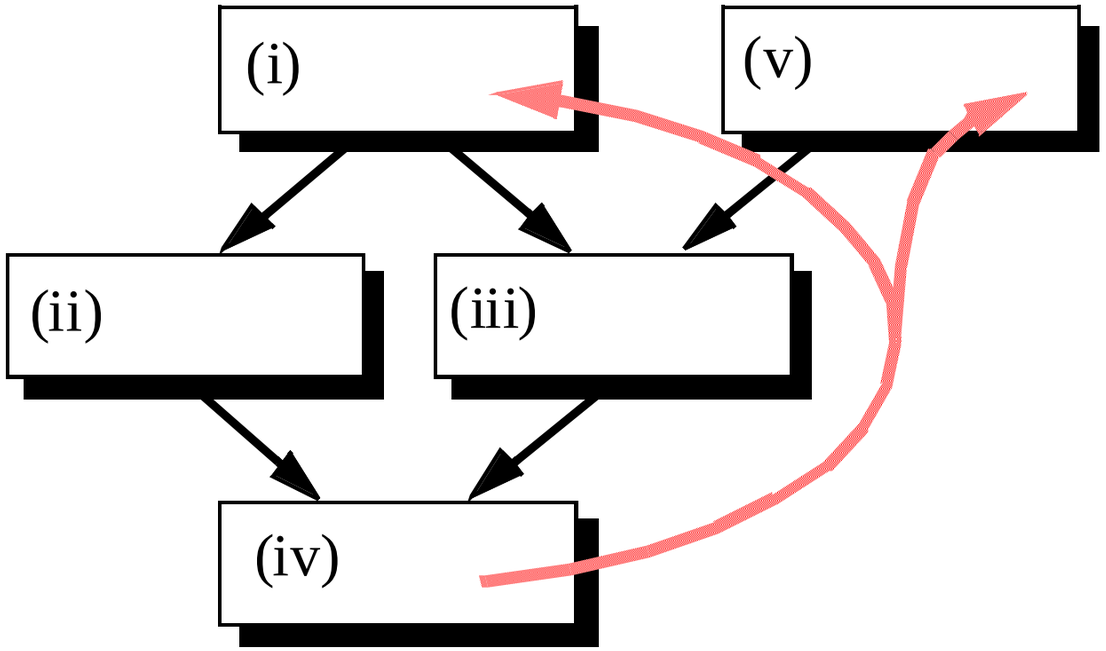
Copy Creation

- **Register renaming causes a problem when there are multiple definitions of a register reaching a single use:**
  - Below, definitions of r1 in both (i) and (ii) reach the use in (iii).
  - If the instruction in (ii) is boosted into (i), it must be renamed to preserve the first value of r1.
  - However, the boosted definition of r1 must reach the use in (iii) as well.
  - Hence, we insert a copy instruction in (ii).



Instruction Replication

- General case of upward code motion: crossing control flow joins.
- Instructions must be present on each control flow path to their original basic block
- Replicate set is computed for each basic block that is a source for instructions to be boosted



Profile Driven Optimizations

- **Wrong optimization choices can be costly!**

  How do you determine dynamic information during compilation?

- **During initial compilation, "extra code" can be added to a program to generate profiling statistics when the program is executed**
- **Execution Profile, e.g.**
  - how many times is a basic block executed
  - how often is a branch taken vs. not taken
- **Recompile the program using the profile to guide optimization choices**
- **A profile is associated with a particular program input**

  ⇒ may not work well on all executions

Generally Useful Optimizations

- **Optimizations that you or the compiler should do regardless of processor / compiler**
- **Code Motion**
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```



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

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

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

```
set_row:
        testq   %rcx, %rcx              # Test n
        jle     .L1                     # If 0, goto done
        imulq   %rcx, %rdx              # ni = n*i
        leaq    (%rdi,%rdx,8), %rdx     # rowp = A + ni*8
        movl    $0, %eax                # j = 0
.L3:                                    # loop:
        movsd   (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd   %xmm0, (%rdx,%rax,8)    # M[A+ni*8 + j*8] = t
        addq    $1, %rax                # j++
        cmpq    %rcx, %rax              # j:n
        jne     .L3                     # if !=, goto loop
.L1:                                    # done:
        rep ; ret
```

Strength Reduction

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x      -->  x << 4
```

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

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

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* 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;
```

```
long 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;
```

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

1 multiplication: i*n

```
leaq    1(%rsi), %rax   # i+1
leaq    -1(%rsi), %r8   # i-1
imulq   %rcx, %rsi      # i*n
imulq   %rcx, %rax      # (i+1)*n
imulq   %rcx, %r8       # (i-1)*n
addq    %rdx, %rsi      # i*n+j
addq    %rdx, %rax      # (i+1)*n+j
addq    %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Optimization Blocker #1: Procedure Calls

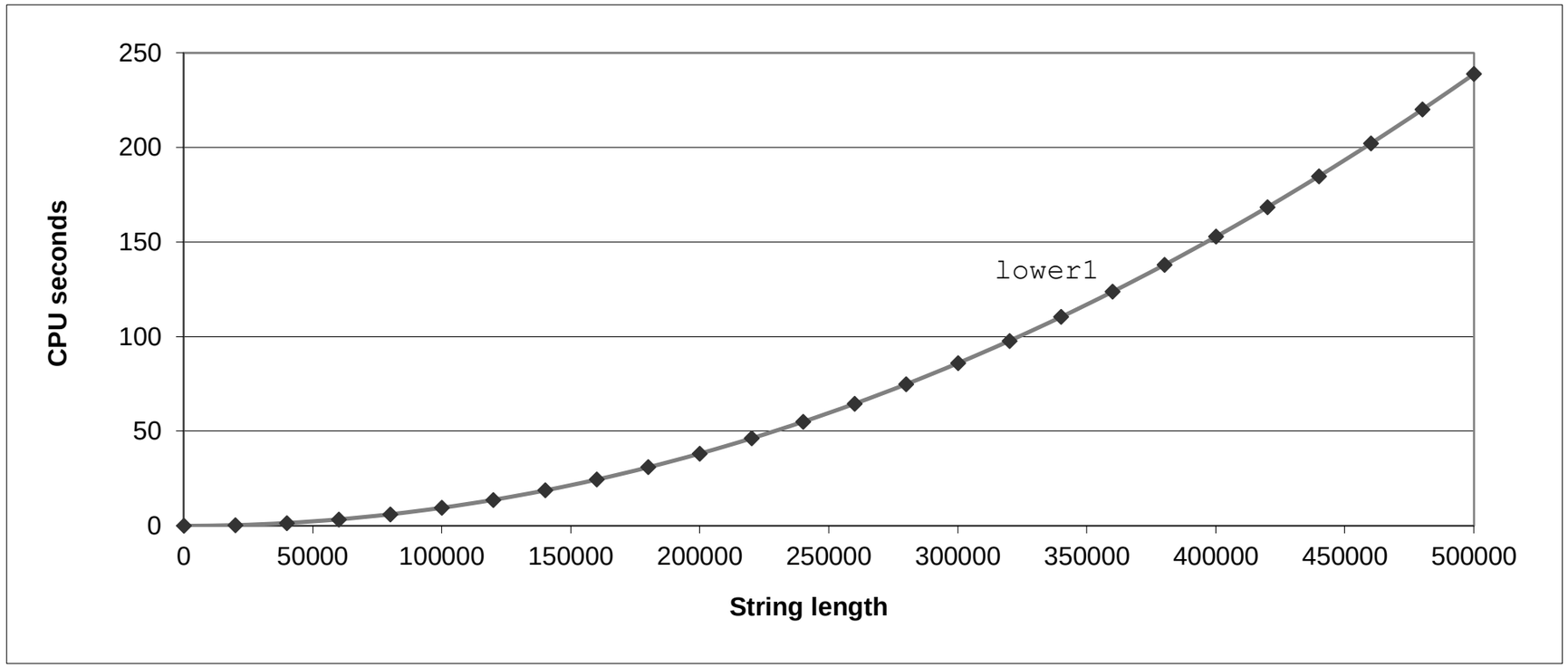
- **Procedure to Convert String to Lower Case**

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

  - Extracted from 213 lab submissions, Fall, 1998

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   size_t i = 0;
   if (i >= strlen(s))
     goto done;
 loop:
   if (s[i] >= 'A' && s[i] <= 'Z')
       s[i] -= ('A' - 'a');
   i++;
   if (i < strlen(s))
     goto loop;
 done:
}
```

- `strlen` executed every iteration

Calling Strlen

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

- **Strlen performance**
  - Only way to determine length of string is to scan its entire length, looking for null character.
- **Overall performance, string of length N**
  - N calls to strlen
  - Require times N, N-1, N-2, …, 1
  - Overall $O(N^2)$ performance

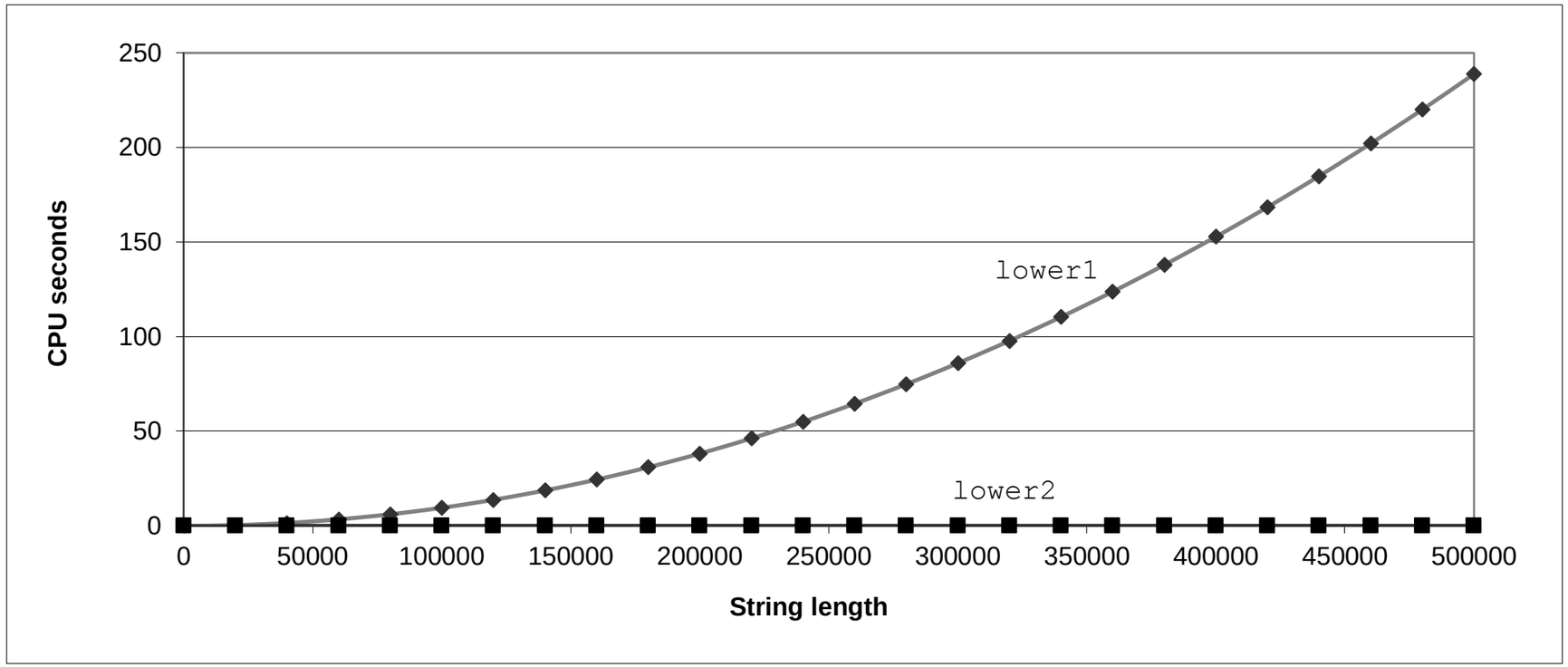
Improving Performance

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

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance of lower2



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 –O1
      - Within single file
  - Do your own code motion

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

Memory Matters

```
/* Sum rows is of 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];
    }
}
```

```
# sum_rows1 inner loop
.L4:
        movsd   (%rsi,%rax,8), %xmm0        # FP load
        addsd   (%rdi), %xmm0               # FP add
        movsd   %xmm0, (%rsi,%rax,8)        # FP store
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

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

Memory Aliasing

```
/* Sum rows is of 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]
```

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Removing Aliasing

```
/* Sum rows is of 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
.L10:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

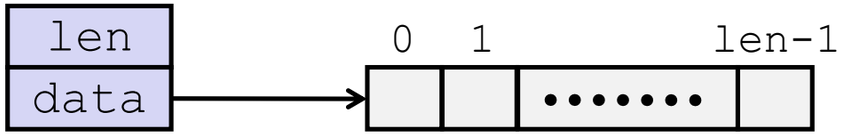
- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - Hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can yield dramatic performance improvement**
  - Compilers often 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{
    size_t len;
    data_t *data;
} vec;
```



- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

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

Benchmark Computation

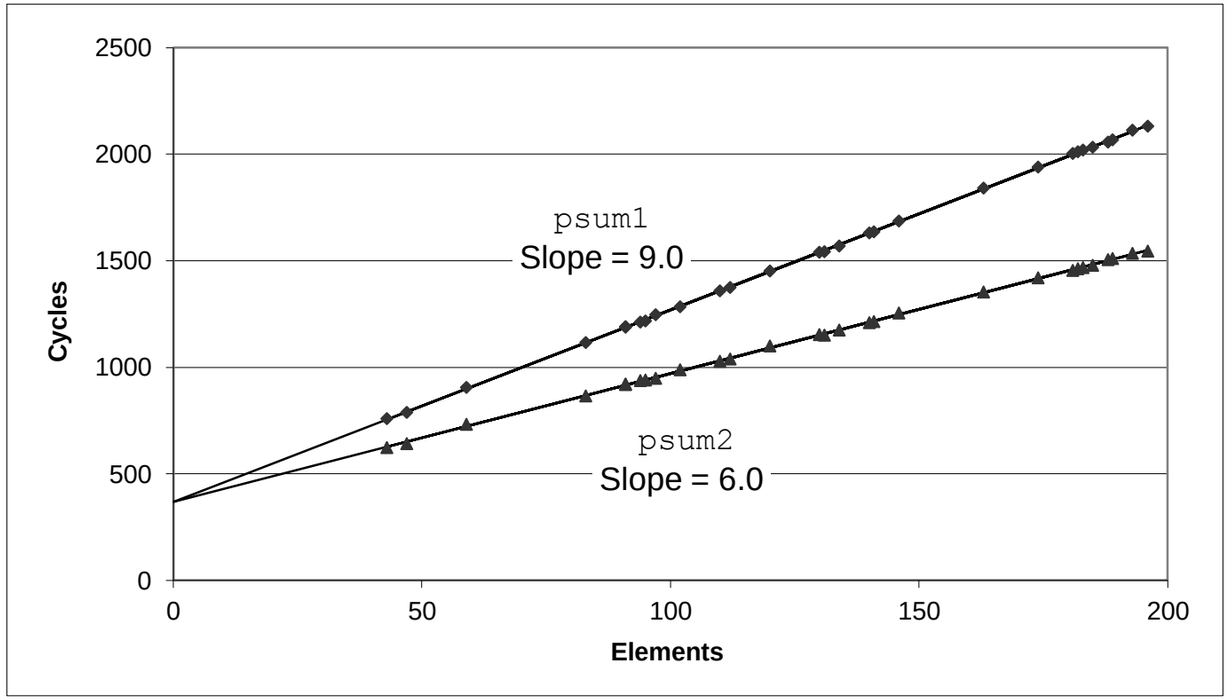
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- **Operations**
  - Use different definitions of `OP` and `IDENT`
  - `+` / `0`
  - `*` / `1`

Cycles Per Element (CPE)

- **Convenient way to express performance of program that operates on vectors or lists**
- **Length = n**
- **In our case: CPE = cycles per OP**
- **T = CPE*n + Overhead**
  - CPE is 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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long 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**

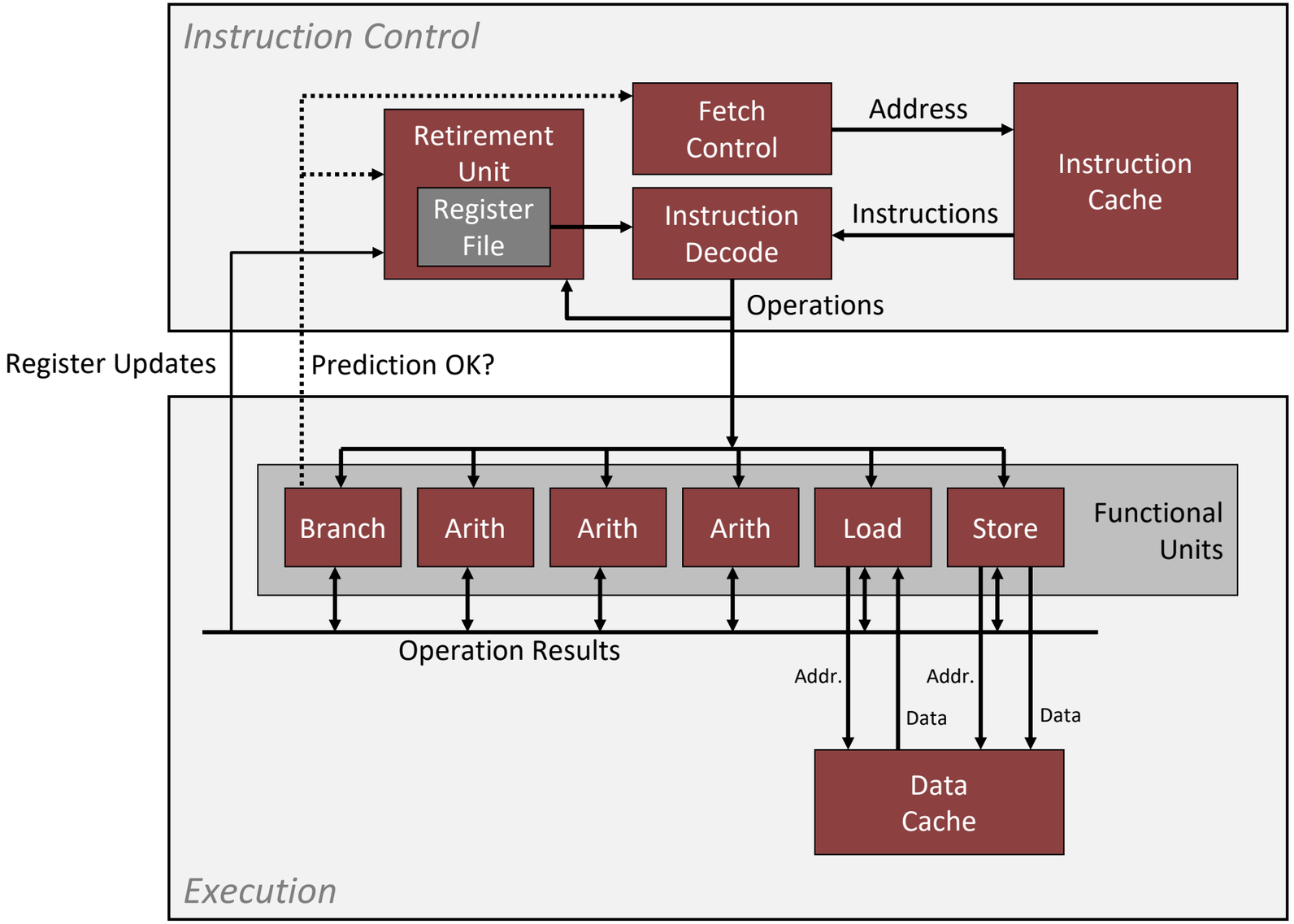
Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long 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;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	10.12	10.12	10.17	11.14
Combine4	1.27	3.01	3.01	5.01

- **Eliminates sources of overhead in loop**

Modern Superscalar CPU Design

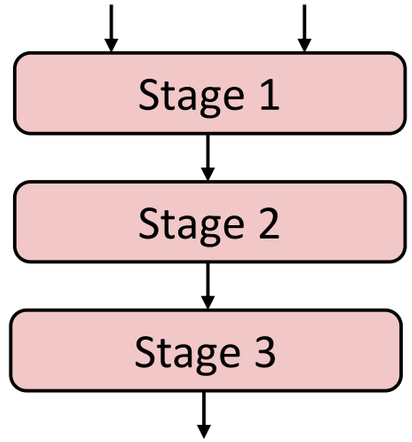


Superscalar Processor

- 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, superscalar processor can take advantage of the *Instruction Level Parallelism* (ILP) that most programs have
- Most modern CPUs are superscalar out-of-order (O3) processors.
- Intel: since Pentium Pro (1995)

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Intel Haswell CPU

- 8 Total Functional Units

- **Multiple instructions can execute in parallel**

  2 load, with address computation
  1 store, with address computation
  4 integer
  2 FP multiply
  1 FP add
  1 FP divide

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

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**3-30**	**3-30**
Single/Double FP Multiply	5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**3-15**	**3-15**

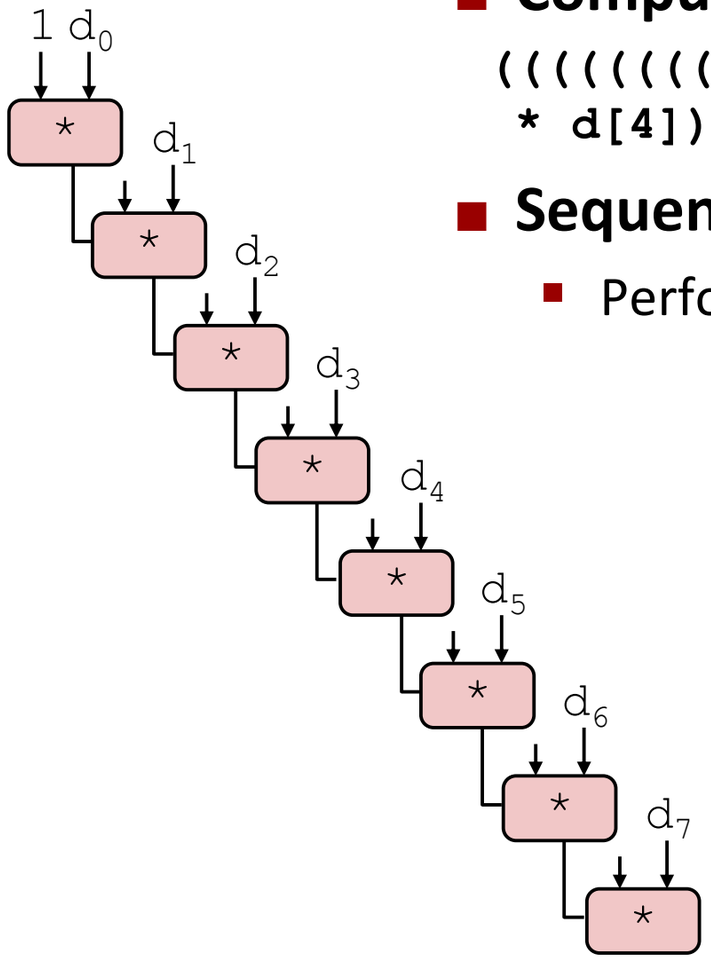
x86-64 Compilation of Combine4

- **Inner Loop (Case: Integer Multiply)**

```
.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
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

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 (2x1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Latency Bound	1.00	3.00	3.00	5.00

- **Helps integer add**
  - Achieves latency bound
- **Others don't improve.** ***Why?***
  - Still sequential dependency

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

Loop Unrolling with Re-association (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long 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 FP. *Why?***

Effect of Re-association

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	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

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

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

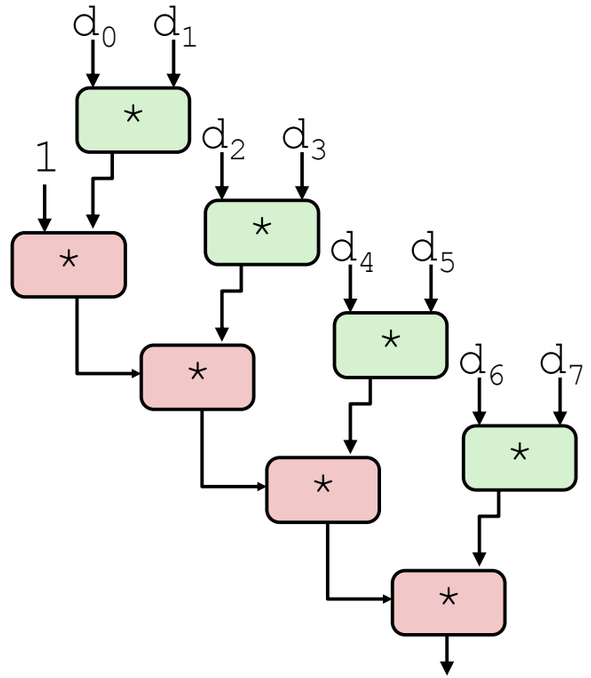
  - Why is that? (next slide)

4 func. units for int +
2 func. units for load

2 func. units for FP *
2 func. units for load

Re-associated 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
  - (N/2+1)*D cycles:
    CPE = D/2

Loop Unrolling with Separate Accumulators (2x2)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long 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 re-association**

Effect of Separate Accumulators

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	1.27	3.01	3.01	5.01
Unroll 2x1	1.01	3.01	3.01	5.01
Unroll 2x1a	1.01	1.51	1.51	2.51
Unroll 2x2	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

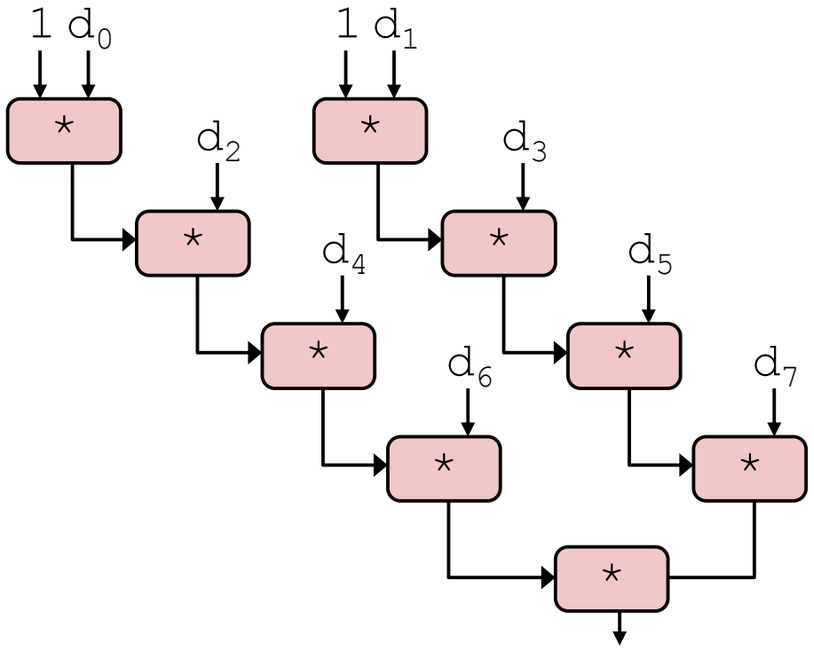
- **Int + makes use of two load units**

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

- **2x speedup (over unroll2) for Int *, FP +, FP ***

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 Haswell
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 0.50

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.01	5.01	5.01	5.01	5.01	5.01	5.01	
2		2.51		2.51		2.51		
3			1.67					
4				1.25		1.26		
6					0.84			0.88
8						0.63		
10							0.51	
12								0.52

Accumulators

Unrolling & Accumulating: Int +

- **Case**
  - Intel Haswell
  - Integer addition
  - Latency bound: 1.00. Throughput bound: 1.00

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.27	1.01	1.01	1.01	1.01	1.01	1.01	
2		0.81		0.69		0.54		
3			0.74					
4				0.69		1.24		
6					0.56			0.56
8						0.54		
10							0.54	
12								0.56

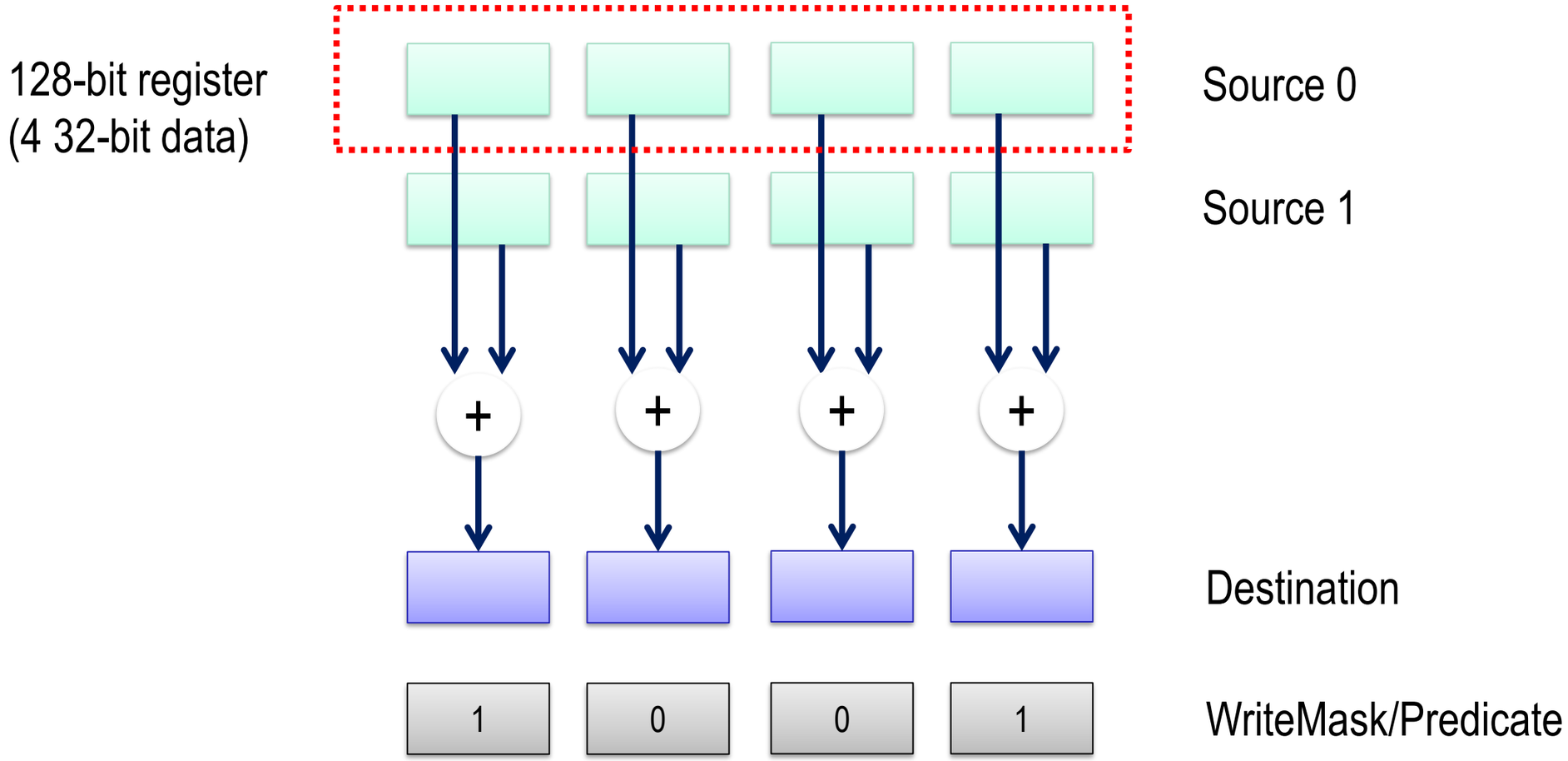
Accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

- **Limited only by throughput of functional units**
- **Up to 42X improvement over original, unoptimized code**

Single Instruction Multiple Data (SIMD)



SIMD Extensions for Superscalar Processors

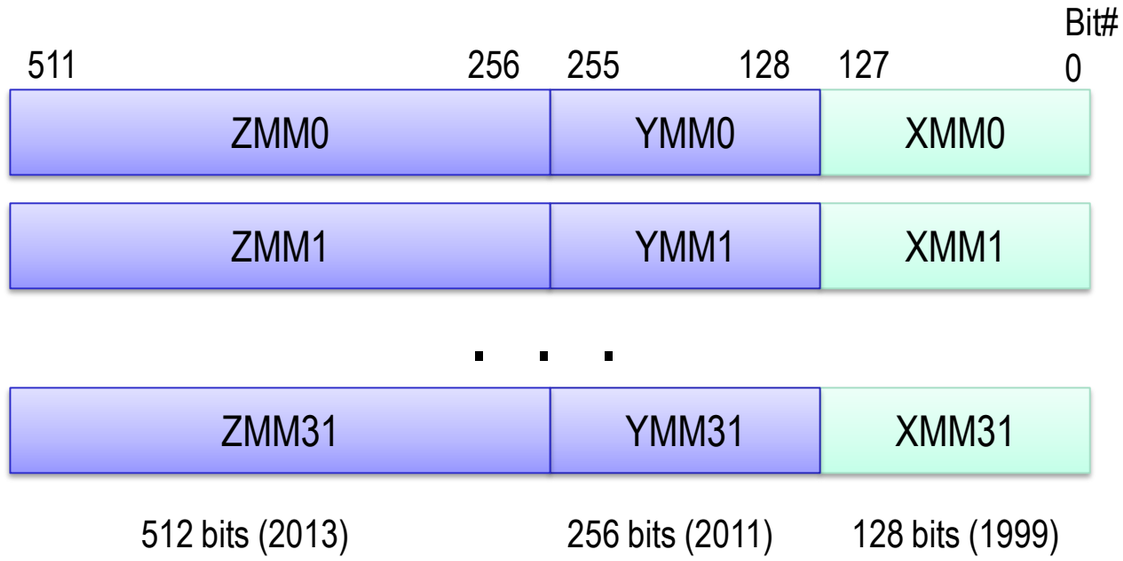
- **Every CISC/RISC processor today has SIMD extensions**
  - MMX, SSE, SSE-2, SSE-3, SSE-4, AVX, AVX2, Altivec, VIS, …
- **Basic idea: accelerate multimedia processing**
  - Define vectors of 8, 16, 32 and 64 bit elements in regular registers
  - Apply SIMD arithmetic on these vectors
- **Nice and cheap**
  - Don't need to define big vector register file
    - This has changed in more recent SIMD extensions
  - All we need to do
    - Add the proper opcodes for SIMD arithmetic
    - Modify datapaths to execute SIMD arithmetic
  - Certain operations are easier on short vectors
    - Reductions, random permutations

Problems with SIMD Extension

- **SIMD defines short, fixed-sized, vectors**
  - Cannot capture data parallelism wider than 64 bits
    - MMX (1996) has 64-bit register s (8 8-bit or 4 16-bit operations)
  - Must use wide-issue to utilize more than 64-bit datapaths
  - SSE and Altivec have switched to 128-bits because of this
  - AVX2 has switched to 512-bits because of this
- **SIMD does not support vector memory accesses**
  - Strided and indexed accesses for narrow elements
  - Needs multi-instruction sequence to emulate
    - Pack, unpack, shift, rotate, merge, etc
  - Cancels most of performance and code density benefits of vectors
- **Compiler support for SIMD?**
  - Auto vectorization is hard
  - Rely on programming model (e.g., OpenMP, Cilk+)

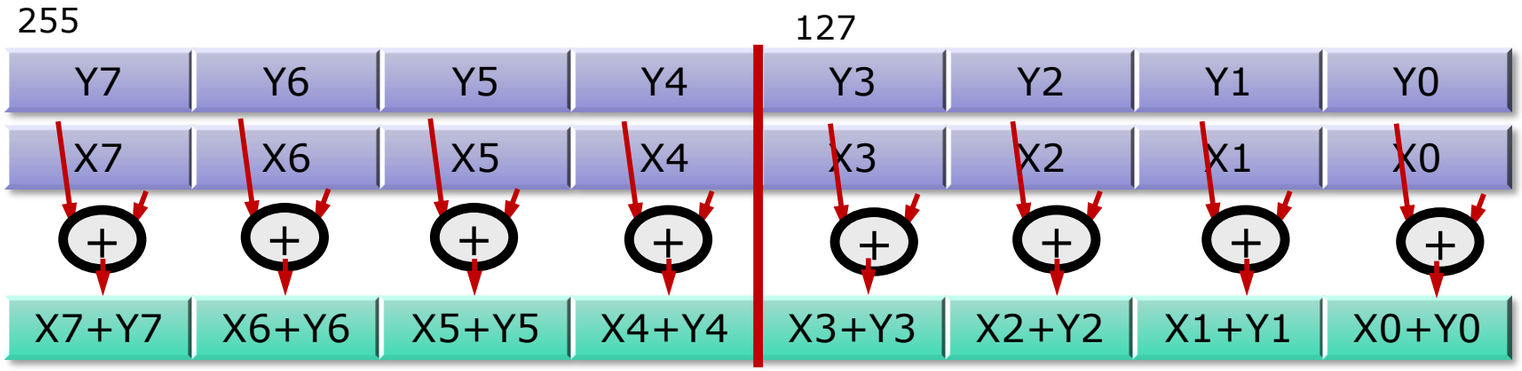
AVX2 SIMD Register Set

- **Intel® AVX extends all 16 XMM registers to 256bits**
- **Intel AVX instructions operate on either:**
  - The whole 256-bits (FP only)
  - The lower 128-bits (like existing Intel® SSE instructions)
    - A replacement for existing scalar/128-bit SSE instructions
      - Provides new capabilities on existing instructions
    - The upper 128-bits of the register are zeroed out
- **Intel AVX2 supports integer operations**

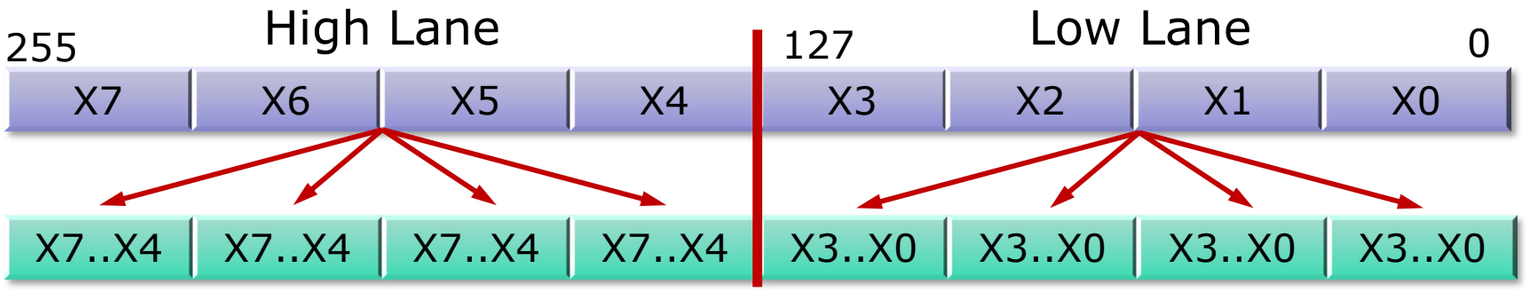


AVX Example

- Intel® AVX defines two 128-bit lanes (low =xmm, high=ymm[255:128])
  - Nearly all operations are defined as "in-lane"
  - For most instructions, e.g., VADDPS, the lane division is uninteresting



- Some in-lane behavior is more interesting: VPERMILPS



Intel® SSE functionality is preserved within lanes

AVX2 Gather Operation

Instruction

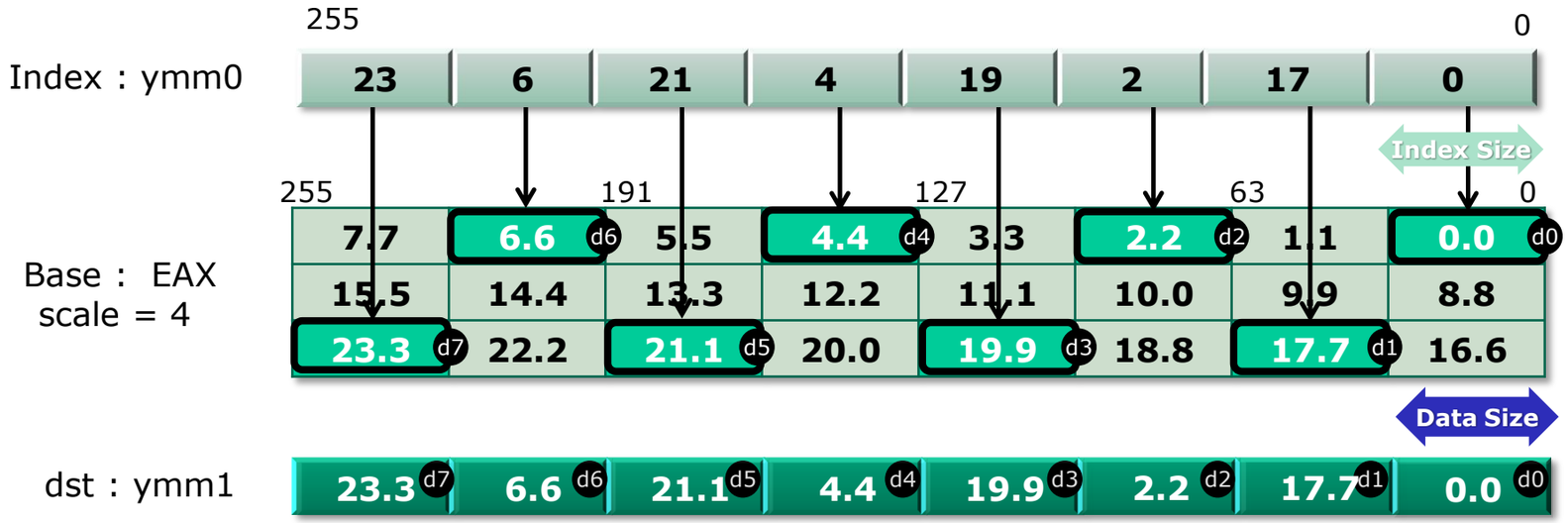
VGATHER**DPS** ymm1,[eax + ymm0*4], ymm2

destination base index scale mask

- **D** : index(offset) size is double word
- **PS**: data type is packed single-precision floating point

Intrinsics

dst = _mm256_i32gather_ps(base, index, scale)



VPGATHERDD & Mask

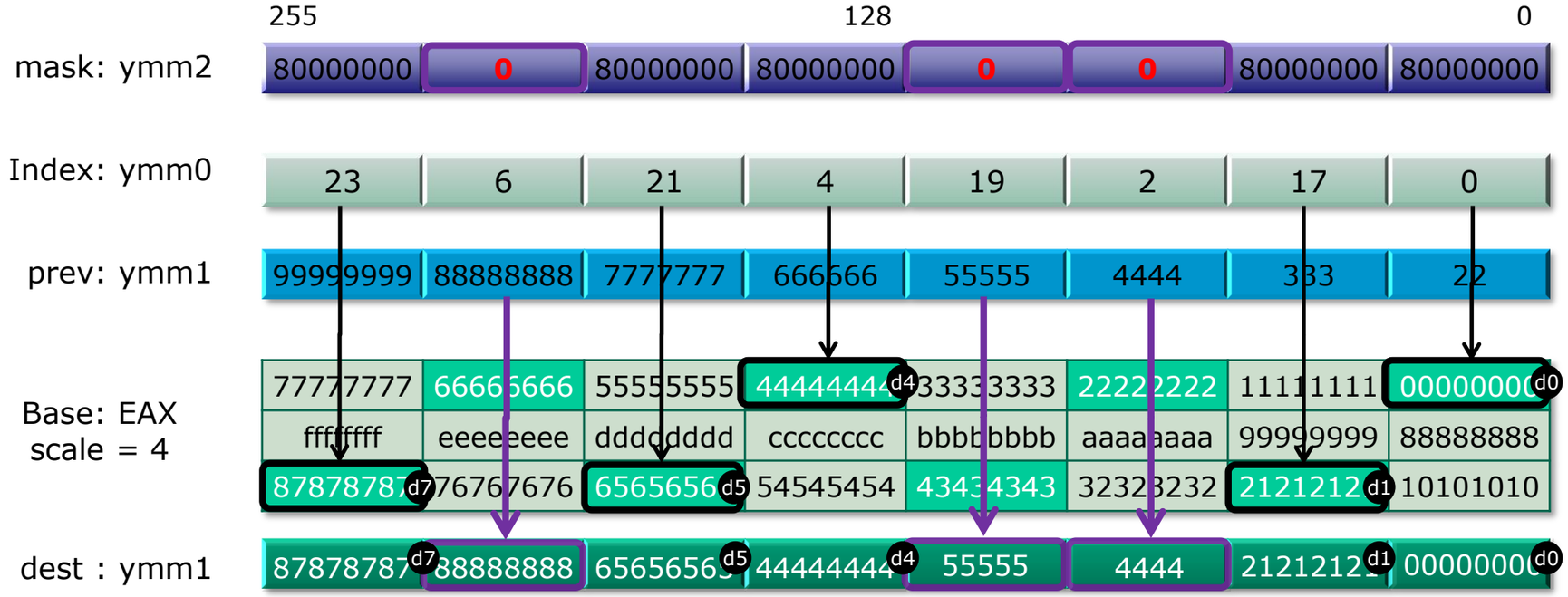
Instruction

VPGATHERDD ymm1,[eax + ymm0*4], ymm2

destination base index scale mask

Intrinsic

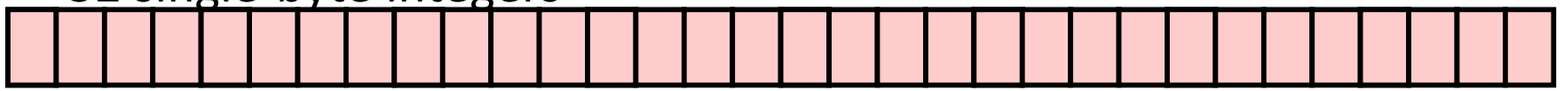
dst = _mm256_mask_i32gather_epi32(base, index, mask, scale)



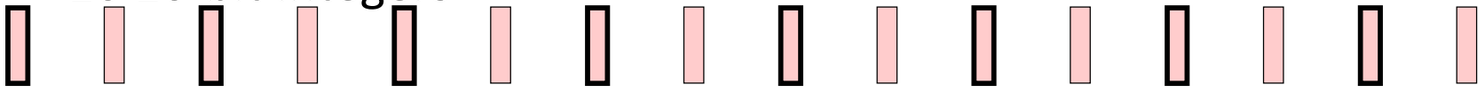
Programming with AVX2

YMM Registers

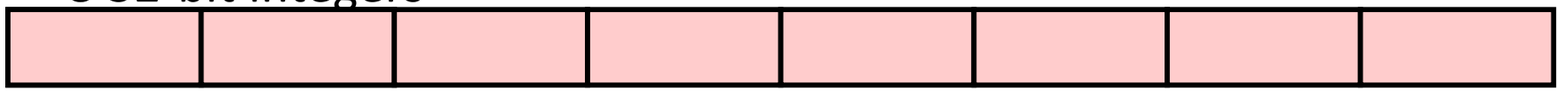
- 16 total, each 32 bytes
- 32 single-byte integers



- 16 16-bit integers



- 8 32-bit integers



- 8 single-precision floats

- 4 double-precision floats

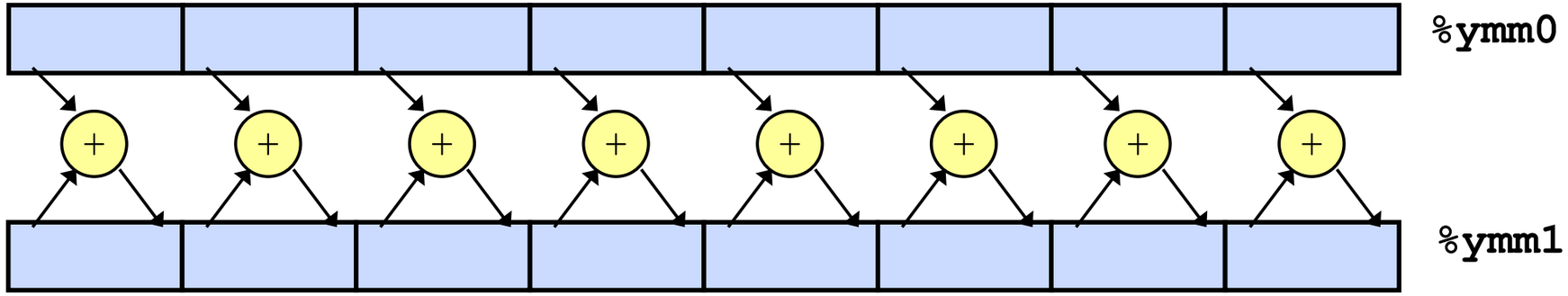
- 1 single-precision float

- 1 double-precision float

SIMD Operations

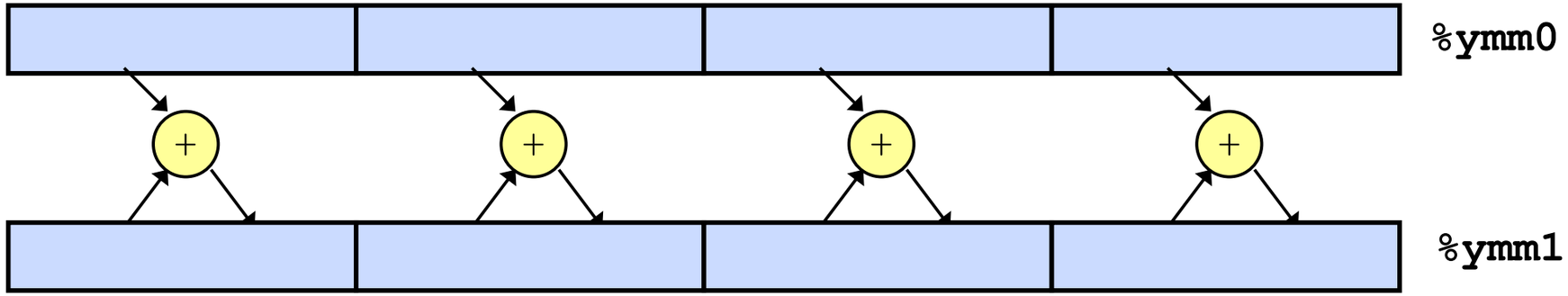
- SIMD Operations: Single Precision

```
vaddsd %ymm0, %ymm1, %ymm1
```



- SIMD Operations: Double Precision

```
vaddpd %ymm0, %ymm1, %ymm1
```



Using Vector Instructions

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Best	0.54	1.01	1.01	0.52
Vector Best	0.06	0.24	0.25	0.16
Latency Bound	0.50	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50
Vec Throughput Bound	0.06	0.12	0.25	0.12

- **Make use of AVX 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

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```

Executing

How to continue?

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

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**

```
  404663:   mov     $0x0,%eax
  404668:   cmp     (%rdi),%rsi
  40466b:   jge     404685
  40466d:   mov     0x8(%rdi),%rax

   . . .

  404685:   repz retq
```

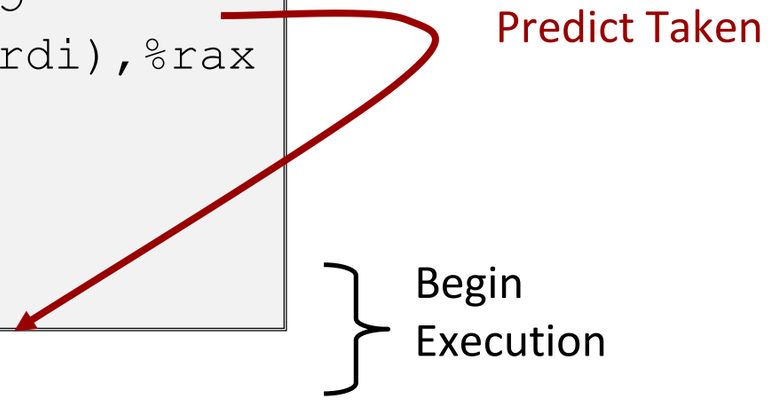
Branch Prediction

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

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```



Branch Prediction Through Loop

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Assume
vector length = 100

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

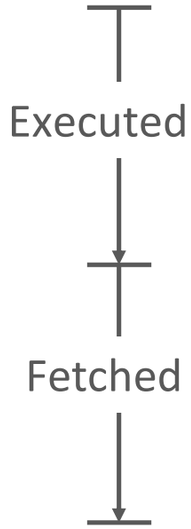
```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 100

Read invalid location

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101



Branch Mis-prediction Invalidation

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Assume
vector length = 100

Predict Taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict Taken (Oops)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

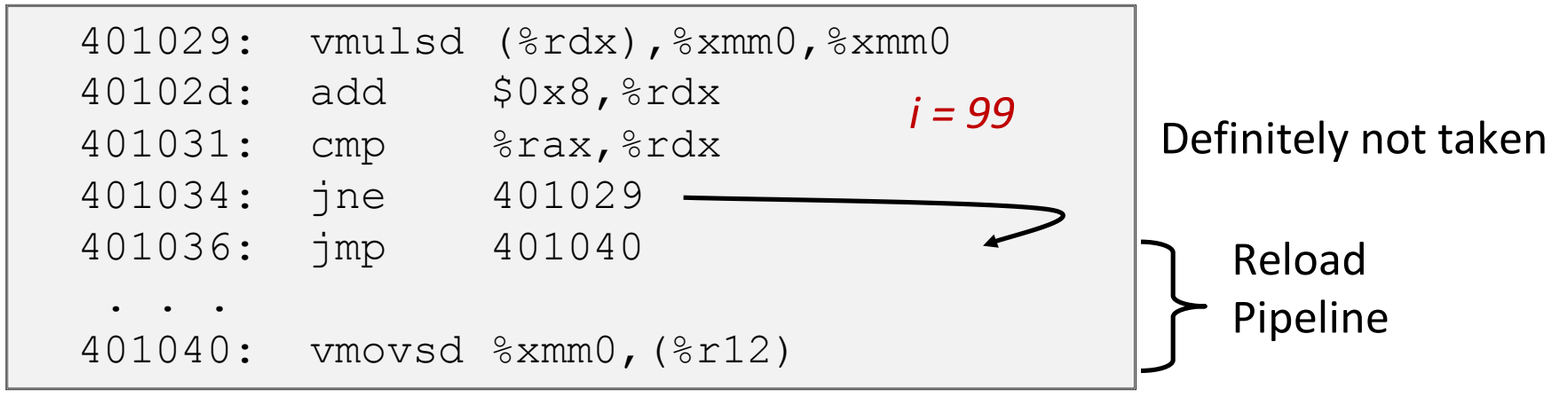
i = 100

Invalidate

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 101

Branch Mis-prediction Recovery



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

Getting High Performance

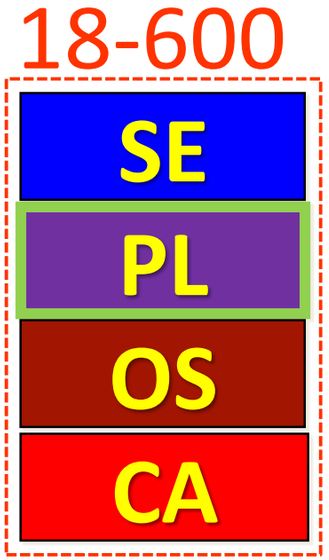
- **Good compiler and flags**
- **Don't do anything stupid**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)
- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly (Covered later in course)

18-600 Foundations of Computer Systems

Lecture 19: "Virtual Machine Design & Implementation"

John P. Shen

November 6, 2017



Next Time ...

- Recommended References:
  - Jim Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, June 2005.
  - Matthew Portnoy, *Virtualization Essentials*, Sybex Press, May 2012

Electrical & Computer ENGINEERING