Computer Organization: A Programmer's Perspective

Optimizing (I): Machine Independent Optimizations

Gal A. Kaminka
galk@cs.biu.ac.il

Many Types of Optimizations

Some general categories

- Code Motion: reduce frequency of some computation
  - Typical: moving code out of loop
  - Reuse common sub-expression
- Strength reduction: replace costly operation with simpler one
  - Sometimes, sacrificing some property, e.g., accuracy, edge cases
  - e.g., using `leal` for `add/mult`, using `<<` to multiply power of 2
- Code elimination:
  - Compile-type evaluation of expressions (constant folding)
  - Skip code whose results will not be used
- ...

Optimizing Compilers

Provide efficient mapping of program to machine

- register allocation
- code selection and ordering
- eliminating minor inefficiencies

Don’t improve asymptotic efficiency (usually)

- up to programmer to select best overall algorithm

Have difficulty overcoming “optimization blockers”

- potential memory aliasing
- potential procedure side-effects

Limitations of Compilers

- Must never change program behavior
  - Prevents optimizations that might affect behavior
  - Even if in practice, conditions can never happen

- Behavior that may be obvious to the programmer not clear to compiler
  - e.g., data ranges may be more limited than variable types

- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Or relies on source code which is not available

- Most analysis is based only on *static* information
  - compiler has difficulty anticipating run-time inputs
  - *Exception*: <u>just-in-time</u> compilation in virtual bytecode machines

What compilers **can** do (in general)

An Example

Original Code:

```
/* Sum 4-grid neighbors of val[i][j] */

sum =    val[i-1][j]
       + val[i+1][j]
       + val[i][j-1]
       + val[i][j+1];
```

- This is a common step in image/video processing, neural networks, …
- Does this for every pixel i,j, so will be called N^2 times!
- **What can we do?**

Reuse Common Sub-Expressions

Original Code:

```
/* Sum 4-grid neighbors of val[i][j] */

sum =    val[i-1][j]
       + val[i+1][j]
       + val[i][j-1]
       + val[i][j+1];
```

We know it is really:

```
/* 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

Reuse Common Sub-Expressions

Original Code:

```
/* Sum 4-grid neighbors of val[i][j] */

sum =   val[i-1][j]
      + val[i+1][j]
      + val[i][j-1]
      + val[i][j+1];
```

We know it is really:

```
/* 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

Which can be transformed into:

```
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

Reuse Common Sub-Expressions

Original Code:

```
/* Sum 4-grid neighbors of val[i][j] */

sum =    val[i-1][j]
       + val[i+1][j]
       + val[i][j-1]
       + val[i][j+1];
```

We know it is really:

```
/* 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;
```

Which can be transformed into:

```
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;
```

gcc can do this when optimizing at -O1 level

Easy: Code Motion in Loops

Original Code:

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

We know it is really:

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

What can we do here?

Easy: Code Motion in Loops

Original Code:

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

We know it is really:

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

Simple Code Motion:

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

Can Do Better

- Recognize sequences of products, replace by increments
- Form of strength reduction

```
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;
}
```

Example of Code motion

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)
# Inner Loop
.L40:
  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 if j<n
```

Many compilers do this

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)
# Inner Loop
.L40:
  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 if j<n
```

Seems not perfect!

Why access memory repeatedly?

Make Use of Registers

- Register access much faster than reading/writing memory
- Compilers better than programmers at allocating registers

But:

- Not always able to determine: can variable be held in register?
- Possibility of *Aliasing*

- Here, if pointer to b[] is modified by writes to a[][] ?
  - Not intention of code –> this would be a bug
  - Compiler did not recognize this

What compilers **cannot** do (in general)

Moving Functions Out of Loop

Procedure 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');
}
```

Often found in student exercise submissions

- `strlen` executed every iteration
- `strlen` linear in length of string
  - Must scan string until finds `'\0'`
- Overall performance is quadratic

This would be better

```
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');
}
```

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

Why can't the compiler do this?

Optimization Blocker: Procedure Calls

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

- **What can the compiler do (very little):**
  - Treat procedure call as a black box
  - Weak optimizations near them
  - Inline the functions (sometimes)
    - gcc can do this within file

Optimization Blocker: Procedure Calls

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

Write your own version inside the file, and inline its use

- **What can the compiler do (very little):**
  - Treat procedure call as a black box
  - Weak optimizations near them
  - Inline the functions (sometimes)
    - gcc can do this within file

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

Do your own code motion!

Compilers Blocked by Memory Aliasing

```
twiddle1(int *xp, *yp) {
  *xp += *yp;
  *xp += *yp;
  }

twiddle2(int *xp, *yp) {
  *xp += 2* (*yp);
  }
```

- Twiddle 2 is faster (less memory accesses)
- Why compiler **not** transform `twiddle1` into `twiddle2?`

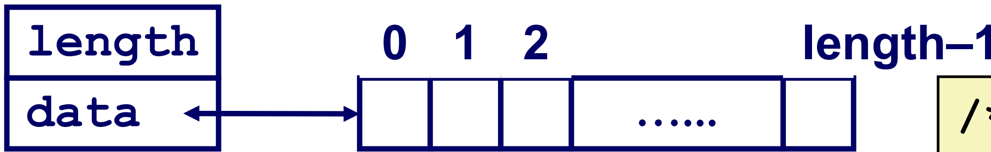
Compilers Blocked by Memory Aliasing

```
twiddle1(int *xp, *yp) {
  *xp += *yp;
  *xp += *yp;
  }

twiddle2(int *xp, *yp) {
  *xp += 2* (*yp);
  }
```

- Because memory aliasing cases affect behavior:
  - twiddle1 (&a,&a) $\rightarrow$ a = 4a
  - twiddle2 (&a,&a) $\rightarrow$ a = 3a
- Compiler has to consider equality, cannot optimize 1 into 2

An Example

Vector ADT




```
/* data structure for vectors */
typedef struct{
        size_t len;
        data_t *data;
} vec;
```

Procedures

`vec_ptr` **`new_vec(int len)`**

- Create vector of specified length

`int` **`get_vec_element(vec_ptr v, int index, data_t *dest)`**

- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

`data_t` **`*get_vec_start(vec_ptr v)`**

- Return pointer to start of vector data

- Similar to array implementations in Pascal, ML, Java
  - E.g., always do bounds checking

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;
    }
}
```

- `data_t`
  - `int`
  - `long`
  - `float`
  - `double`

- `OP, IDENT`
  - `+ / 0`
  - `* / 1`

Compute sum or product of vector elements

Initial version (`combine1`)

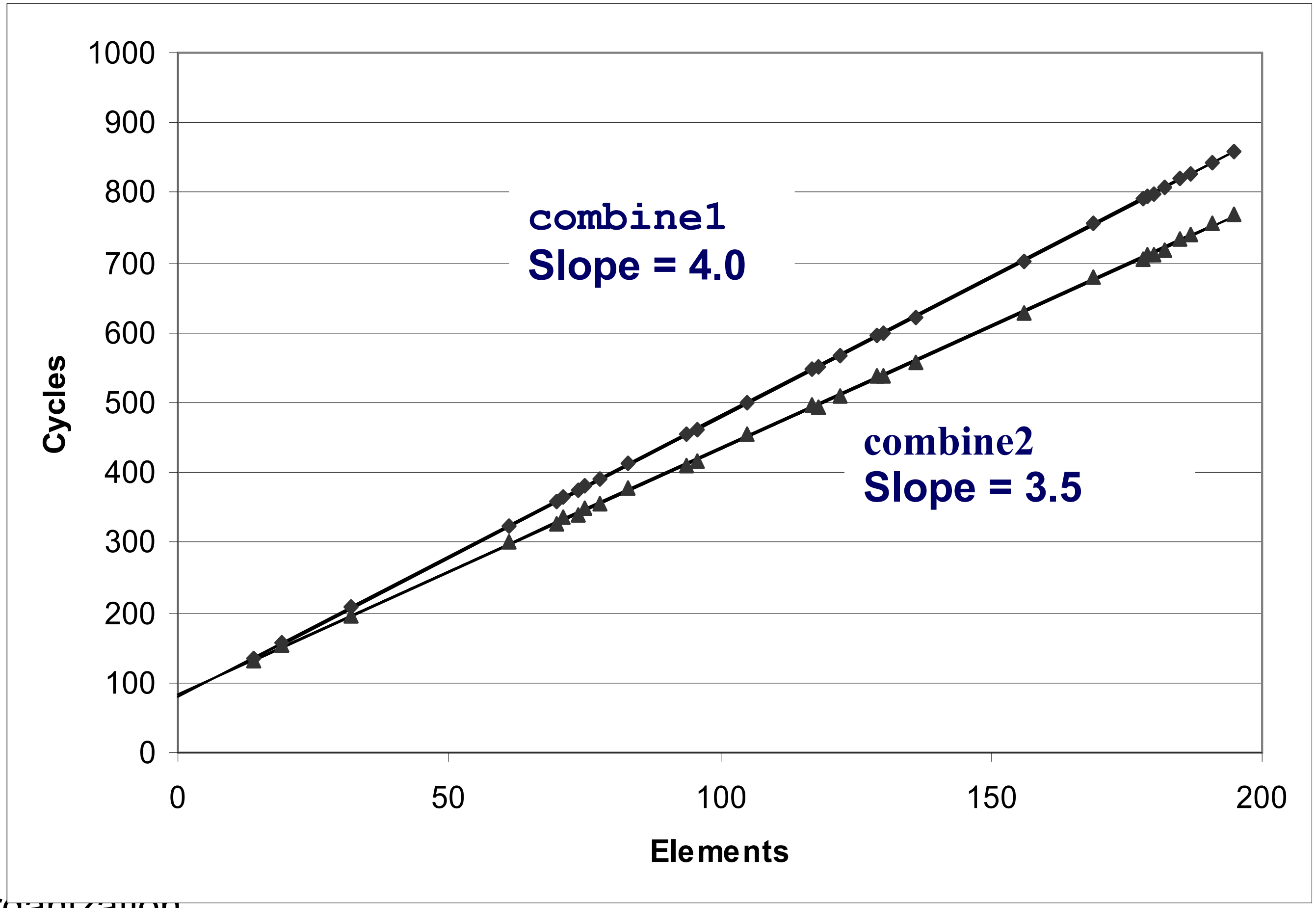
```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Procedure

- Compute sum of all elements of vector
- Store result at destination location

Measure: Cycles Per Element

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- T = CPE*n + Overhead



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

Move `vec_length` Call Out of Loop

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Optimization

- Move call to `vec_length` out of inner loop
  - Value does not change from one iteration to next
  - Code motion
- `vec_length` requires only constant time, but significant overhead

Reduction in Strength

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

Avoid procedure call to retrieve each vector element

- Get pointer to start of array before loop
- Within loop just do pointer reference
- Not as clean in terms of data abstraction

Eliminate Unneeded Memory Refs

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

- Don’t need to store in destination until end
- Local variable `sum` held in register
- Avoids 1 memory read, 1 memory write per cycle
- Memory references are expensive!

Optimization Blocker: Memory Aliasing

Aliasing

- Two different memory references specify single location

Example

- `v: [3, 2, 17]`
- `combine1(v, get_vec_start(v)+2) --> [3,2,10]`
- `combine4(v, get_vec_start(v)+2) --> [3,2,22]`

Observations

- Easy to occur 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

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	
Combine1 unoptimized	22.68	20.02	19.98	20.18
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

Machine-Independent Opt. Summary

Code Motion

- *Compilers are good at this for simple loop/array structures*
- *Don't do well in presence of procedure calls and memory aliasing*

Reduction in Strength

- Shift, add instead of multiply or divide
  - *compilers are (generally) good at this*
  - *Exact trade-offs machine-dependent*
- Keep data in registers rather than memory
  - *compilers are not good at this, since concerned with aliasing*

Share Common Subexpressions

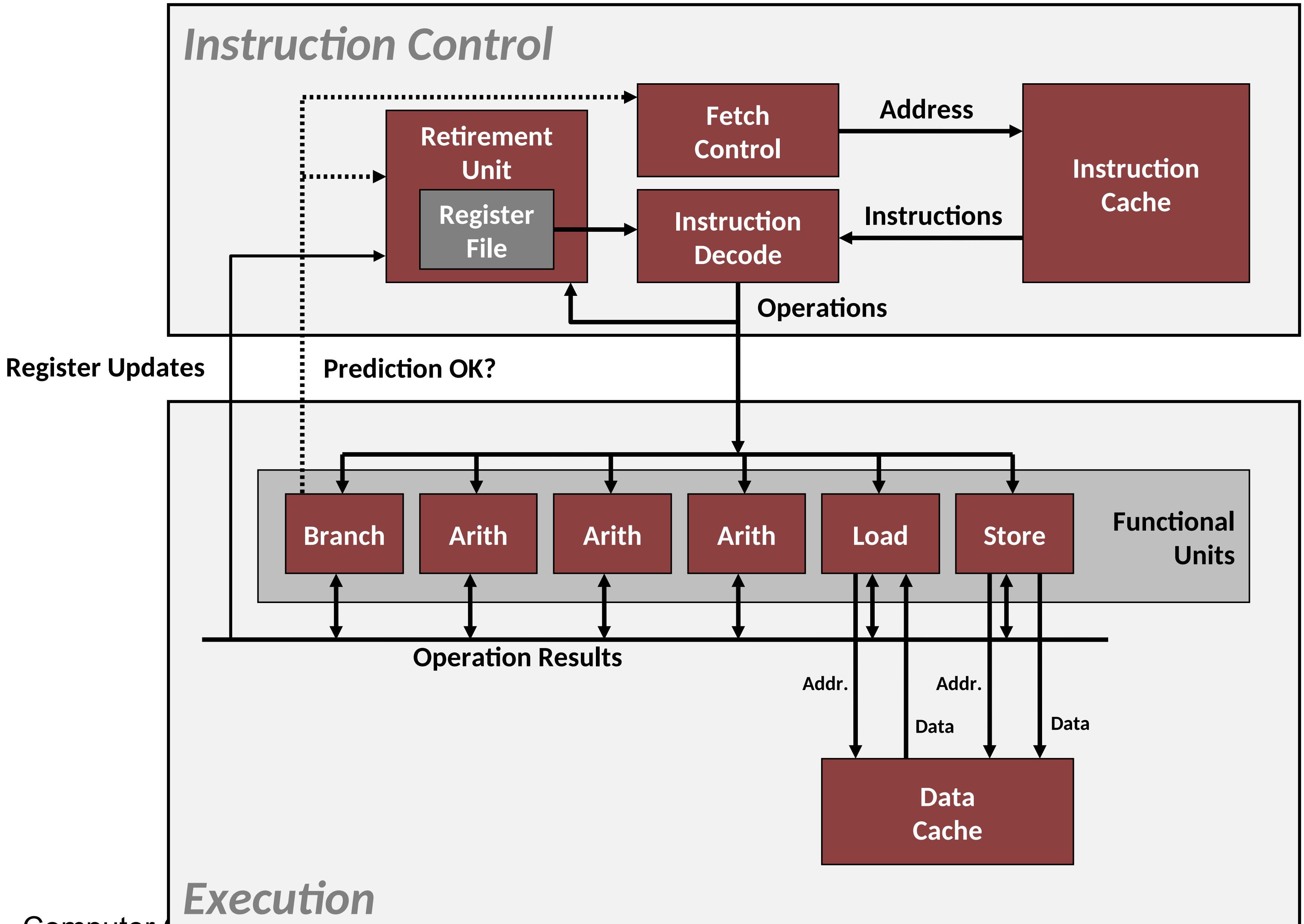
- *compilers have limited algebraic reasoning capabilities*
- *Help compiler overcome aliasing, use local variables*

Computer Organization: A Programmer's Perspective

Optimizing (II): Machine Dependent (but generally doable) Optimizations

Gal A. Kaminka
galk@cs.biu.ac.il

Modern CPU Design



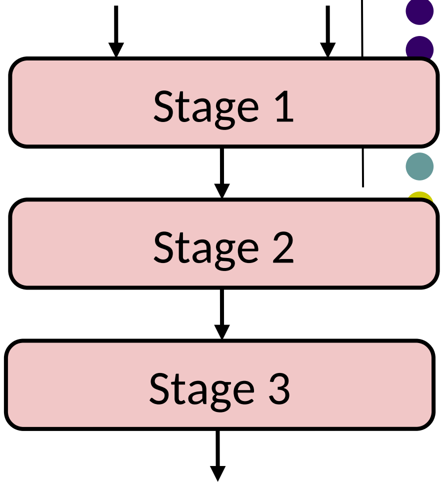
Superscalar Processor

- **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, superscalar processor can take advantage of the ***instruction level parallelism*** that most programs have

- Most modern CPUs are superscalar.
- Intel: since Pentium (1993)

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 Add	1	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**

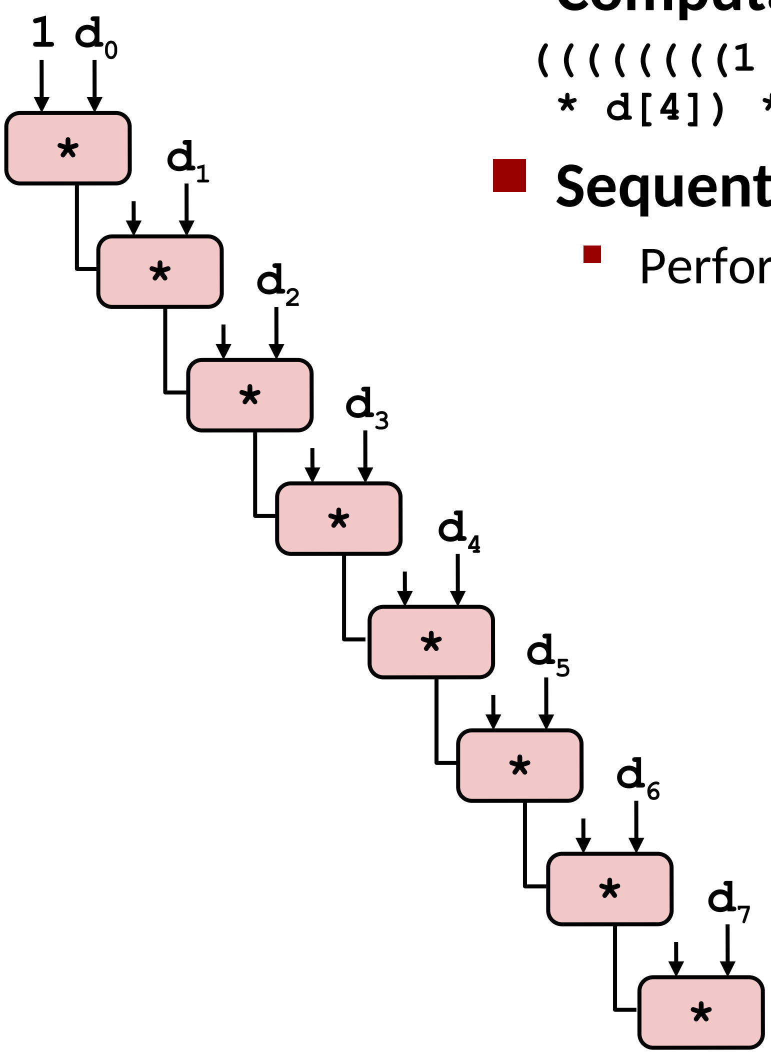
x86-64 Compilation of Combine4

- **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

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

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

Loop Unrolling with Reassociation (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 Reassociation

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

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

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

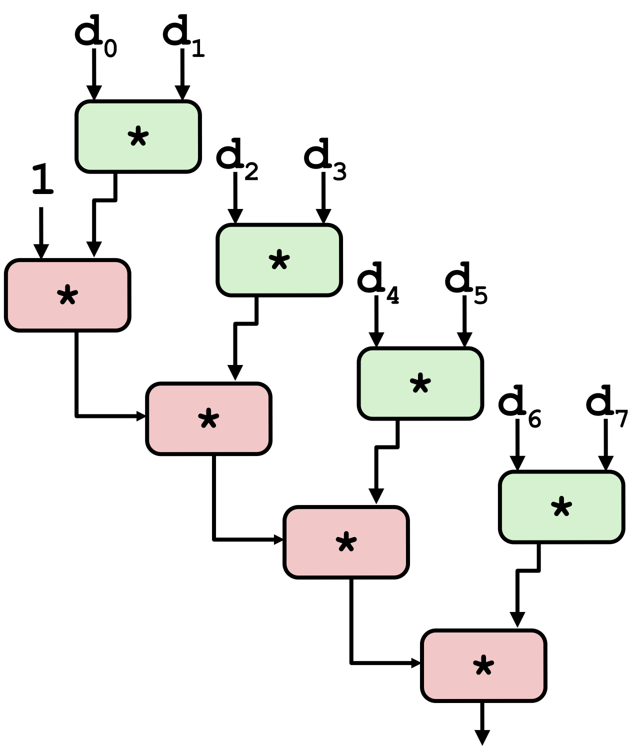
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

```
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
  - (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 reassociation**

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

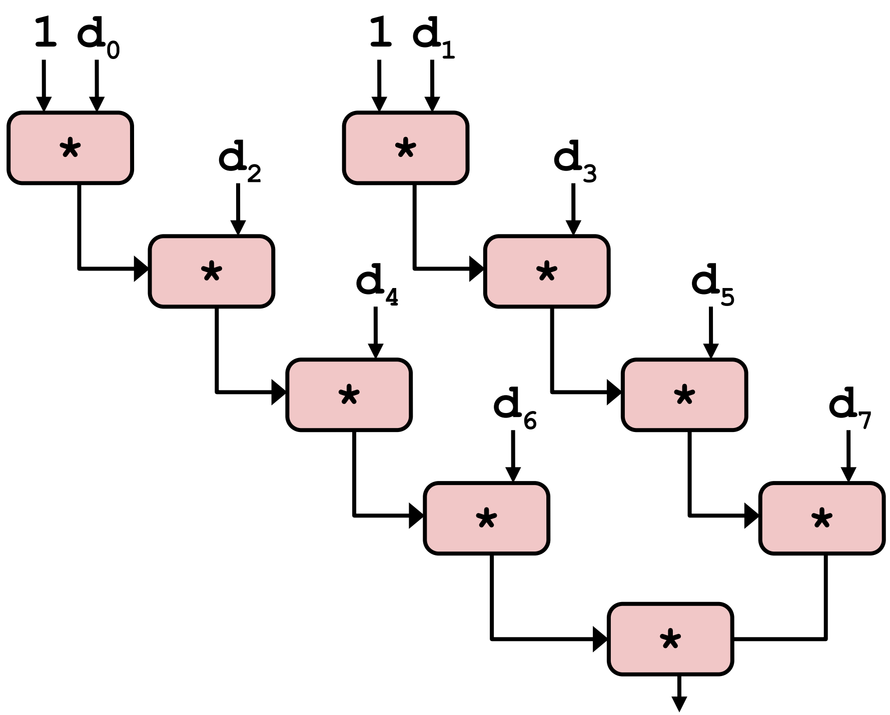
- Int + makes use of two load units

```
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

Accumulators

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

Achievable Performance

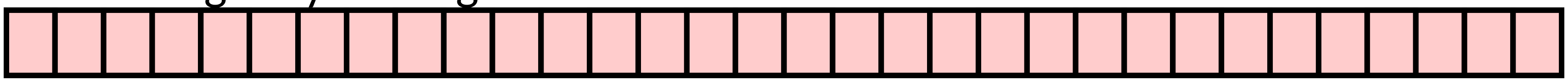
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

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

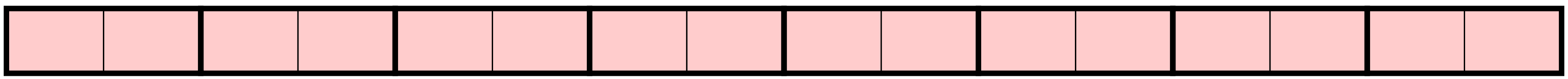
Programming with AVX2

YMM Registers

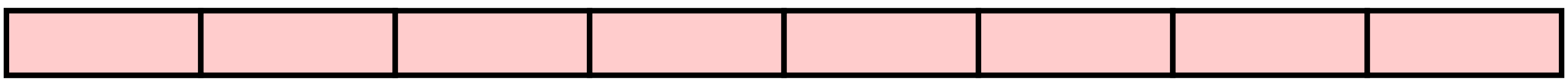
- 16 total, each 32 bytes
- 32 single-byte integers



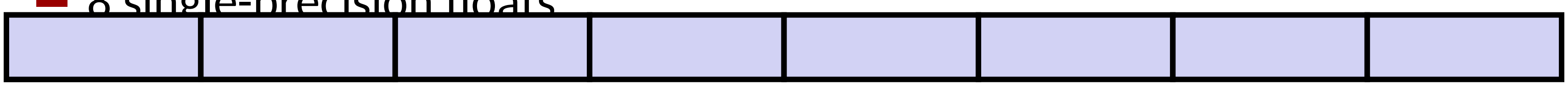
- 16 16-bit integers



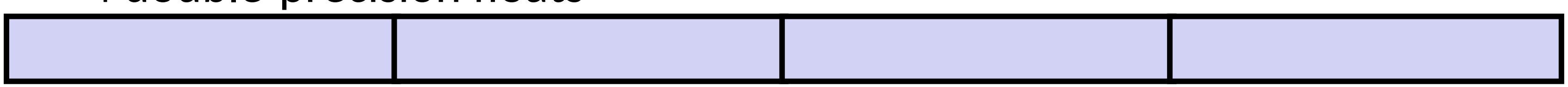
- 8 32-bit integers



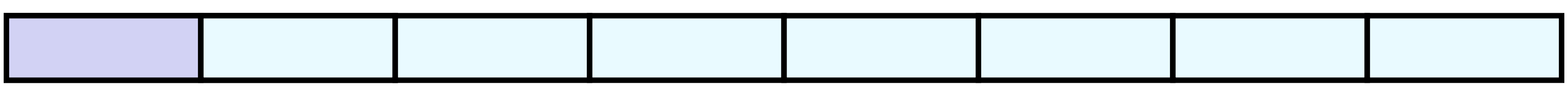
- 8 single-precision floats



- 4 double-precision floats



- 1 single-precision float

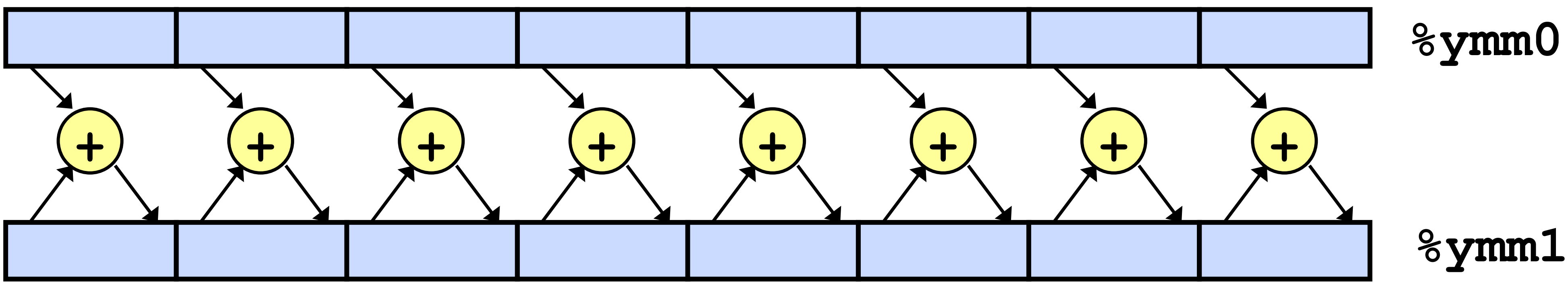


- 1 double-precision float

SIMD Operations

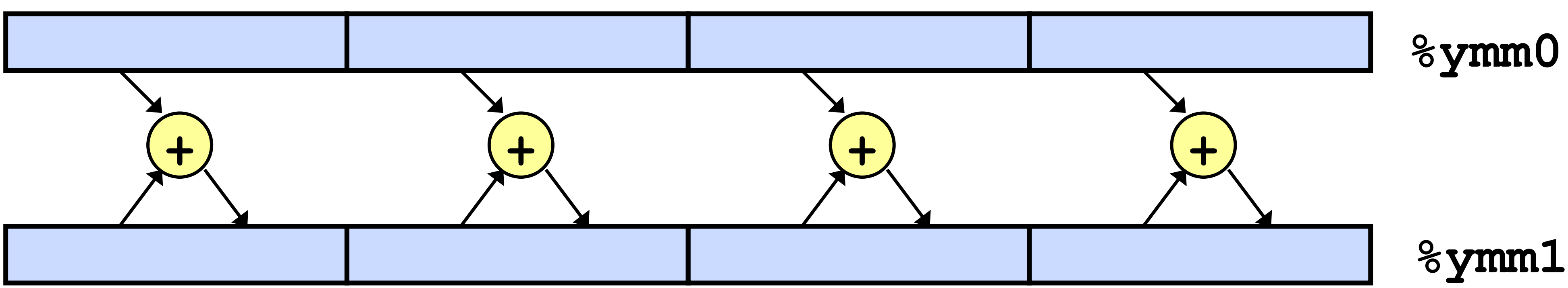
- SIMD Operations: Single Precision

```
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

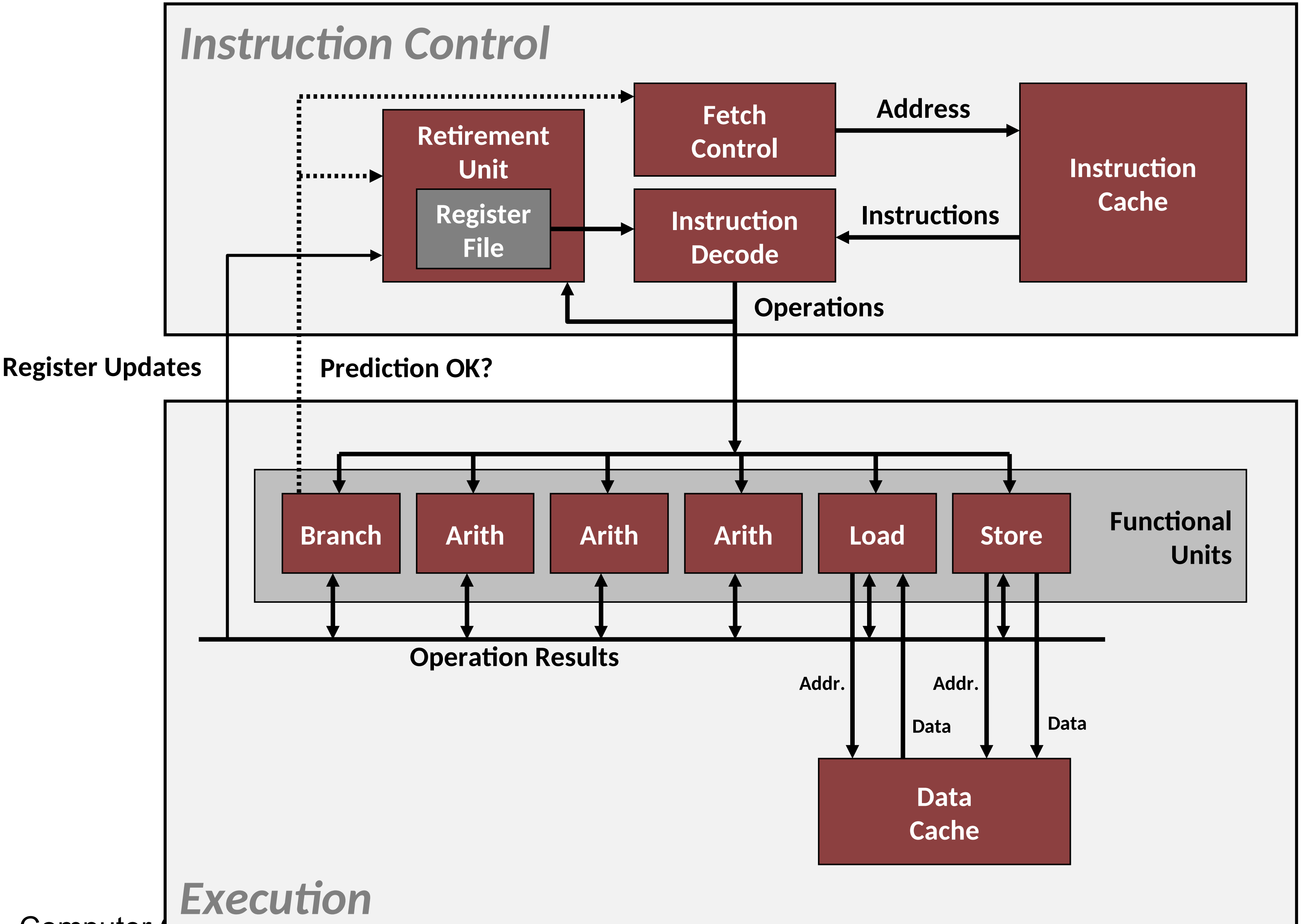
```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

 . . .

404685:   repz retq
```

Executing

How to continue?

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

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**

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

  . . .

404685:   repz retq
```

Branch Not-Taken

Branch Taken

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
```

Predict Taken

Begin Execution

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

Executed

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

i = 101

Fetched

Branch Misprediction 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

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

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
401036:   jmp    401040
  . . .
401040:   vmovsd %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- **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 (discussed elsewhere in course)