Operating Systems and C
Fall 2022, Performance-Track
7. Program Optimization

With slides from Bryant and O'Hallaron

Outline

Overview

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

- what optimizations does the compiler do?
- how can I structure my program to have impact on what instructions the compiler will generate?

perflab is all about this.
get 2 **f**s, naive implementation.
must improve.trial and error.
must understand architecture, and what compiler can do.
how to program so compiler can generate code that is more efficient (avoids pitfalls)

issues that compiler faces.

we see that processor does actually not do instruction at a time.
runs multiple instructions in parallel.
there are ways to leverage this.

avoid branch misprediction

Performance Realities

big-O, etc.

There's more to performance than asymptotic complexity

- Constant factors matter too!

  order of magnitude

  - 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

your task: implement alg. efficiently. compiler is your friend here! ("black box" to DS stud. SWU study them)

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

some optimizations explained in a bit

(also re-order instructions)

- 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

Limitations of Optimizing Compilers

compiler is conservative

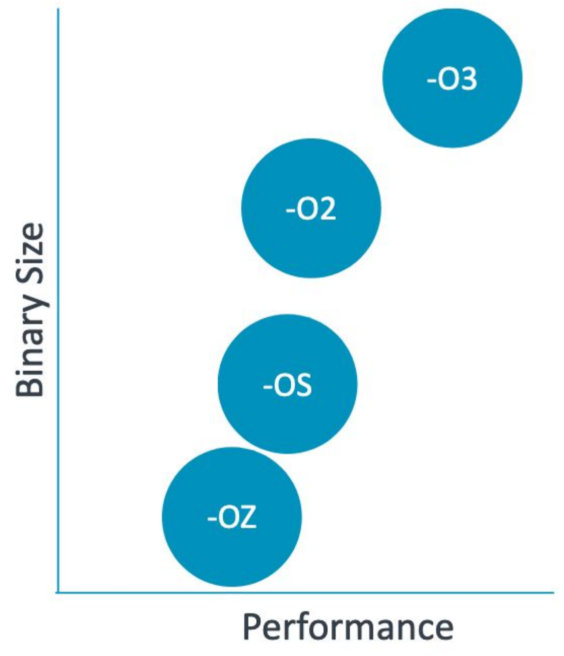
- Operate under fundamental constraint
  - Must **not** cause any **change** in program **behavior**
  - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior 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 interprocedural 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

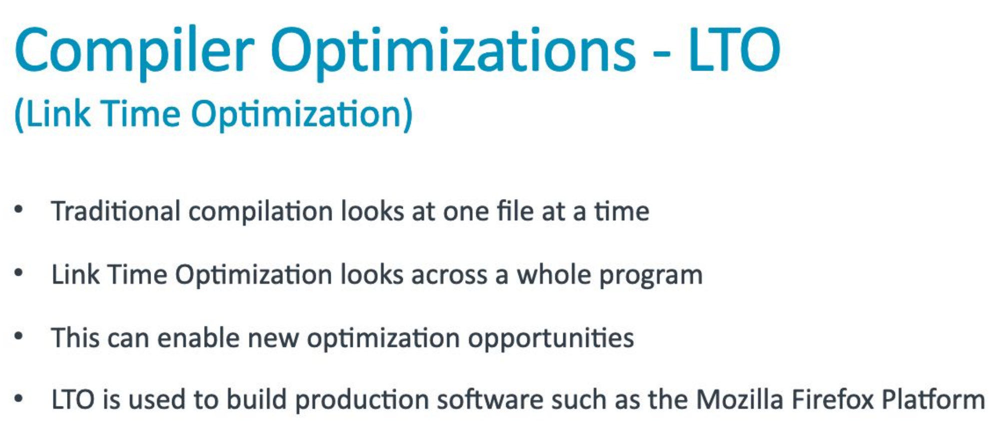
despite this, compiler can help. example:

```
int junk ( int n ) {
  int k = 0;
  for (int i = 0; i <= n; i++){
    k += i;
  }
  return 4;
}
```

Evolving compilers

no optimization by default.
pick right optimization in make file.
tradeoff between performance and binary size.
O2 strongly recommended.





Compiler Optimizations - LTO
(Link Time Optimization)

- Traditional compilation looks at one file at a time
- Link Time Optimization looks across a whole program
- This can enable new optimization opportunities
- LTO is used to build production software such as the Mozilla Firefox Platform

9

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://gcc.gnu.org/onlinedocs/gccint/LTO-Overview.html#LTO-Overview

Slide courtesy of James Greenhalgh,

Outline

Overview

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

Generally Useful Optimizations

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

- **Code Motion**
  - Reduce frequency with which computation is performed
    - If it will always produce same result
    - Especially moving code out of loop

i : row index

n : row length

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

Q: spot another (code motion) optimization?

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];
```

compiler added another optimization!

```
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
```

Reduction in Strength

shift (1 cycle) is way more efficient than multiplication (3 cycles)

- 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;
}
```

(think of multiplication as a sequence of additions)

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;
```

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

```
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;
```

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
```

Outline

- Overview
- 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

Optimization Blocker #1: Procedure Calls

Procedure to Convert String to Lower Case

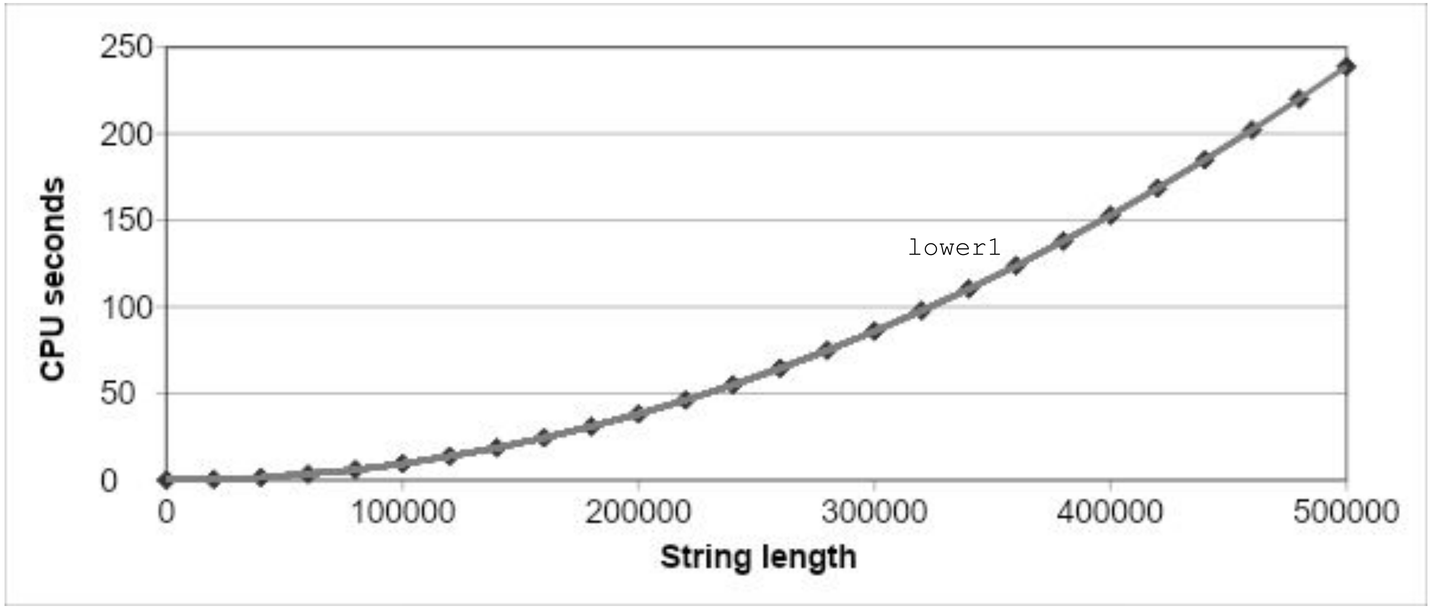
```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

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

Lower Case Conversion Performance

10s

- Time quadruples when double string length
- Quadratic performance



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 lower2(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');
}
```

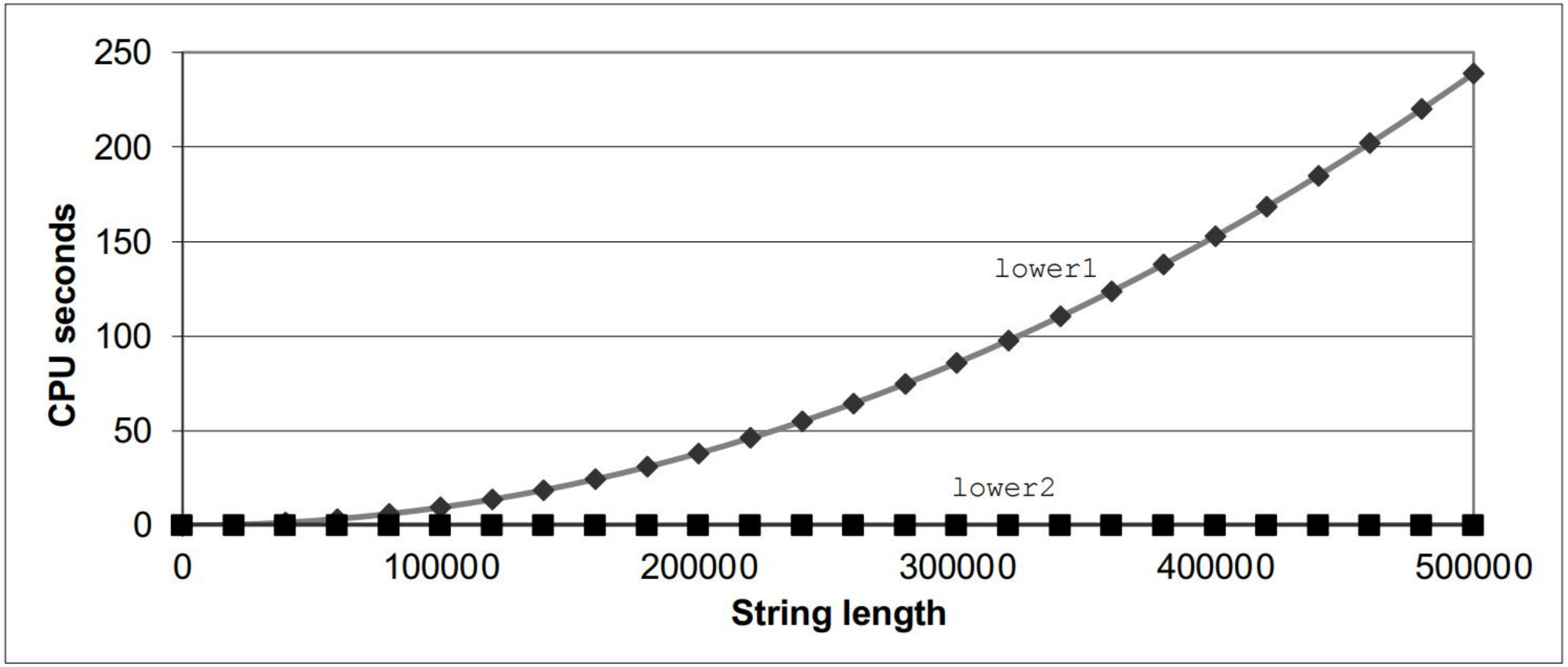
Move call to `strlen` outside of loop

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

- *Compiler won't move* `strlen` *out of inner loop. Why won't it?*
  - 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 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?

why doesn't the compiler keep the intermediate results in a register, and write register to mem when done?
(would be 100x faster)

Memory Aliasing

```
/* Sum rows 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 = 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]
```

we just updated part of A

reading & writing to same row

Q: first suppose we had

```
double B[3] = {42,42,42};
```

what are final values in B?

```
final: [3, 28, 224]
```

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

Removing Aliasing

```
/* Sum rows 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;
    }
}
```

Value of B:

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

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

```
i = 1: [3, 27, 16]
```

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

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0   # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

no mov instruction;
100x faster.

Now `val` cannot be an alias for cells in `a`. (in inner loop)

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

Outline

Overview

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

Exploiting Instruction-Level Parallelism

CPUs are in fact not "1 instruction at a time" interpreters as we introduced them

- Need general understanding of modern processor design

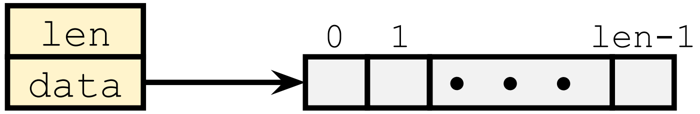
  Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement

  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;
```



intuition holds if `data` in a vec points to array of length `len`

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;
}
```

intuition holds if `val` just points to a `data_t`, not an array thereof.

IT UNIVERSITY OF COPENHAGEN

Benchmark Computation

```
typedef vec* vec_ptr;
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;
    }
}
```

in benchmark: 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`

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**

twice as fast

Basic Optimizations

let's apply the optimizations that
we've learned about so far.
(help the compiler help us)

```
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**

order of magnitude on top of previous improvement! (**this really pays off!**)

Eliminates sources of overhead in loop

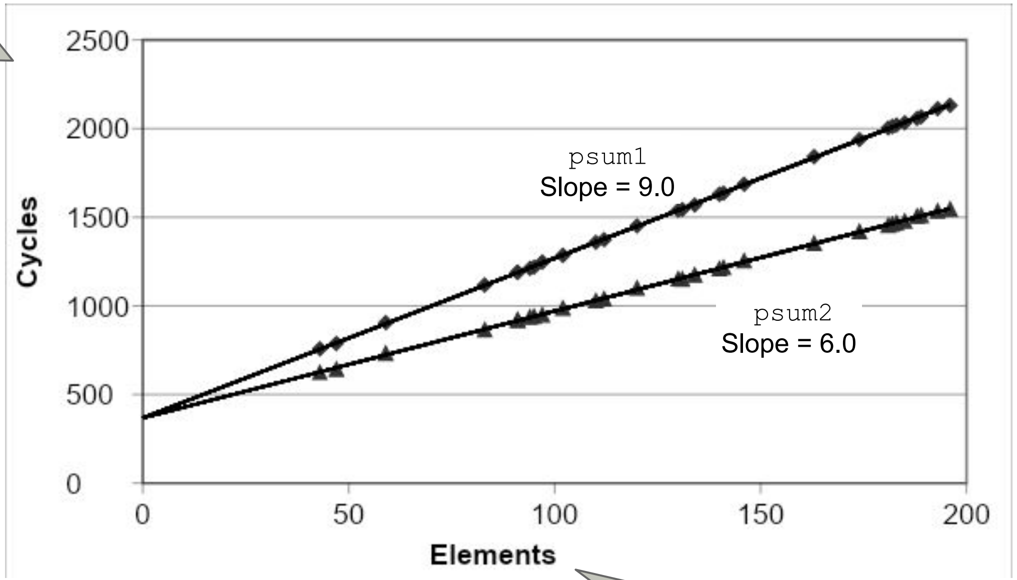
if we can do more than 1 op at a time, then we can go below "1 op per element in sequence" time.

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

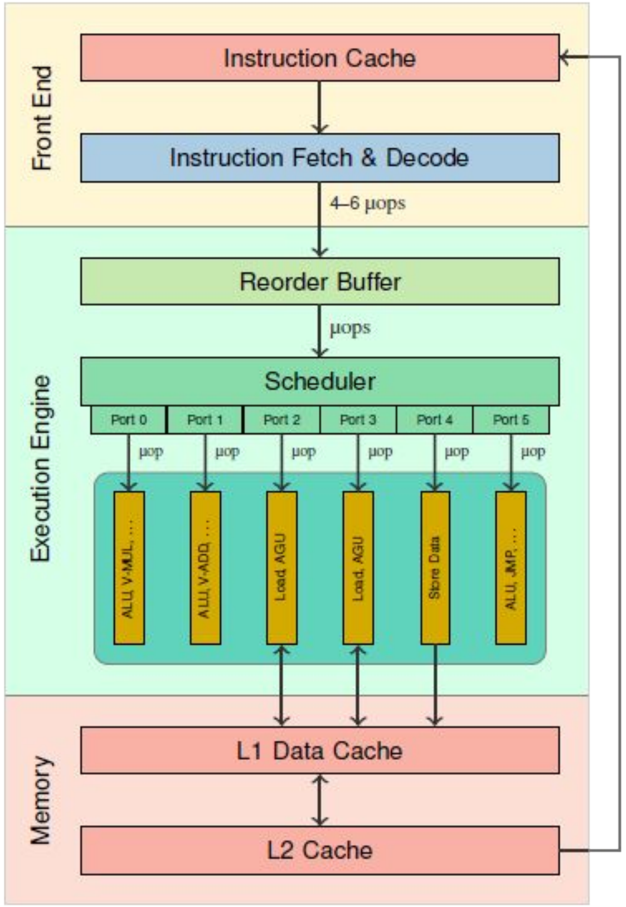
example of how to present such a benchmark result:

CPU time (cycles spent)



length of vector

Modern CPU Design



front end
accesses L2 cache.

back end
accesses L1 cache, and if necessary L2 cache.

execution engine
reorders & schedules instructions (to different ports.
ports = how many micro-operations **at the same time**. each instruction is 1+ micro-operation. recent version of ARM processor has 4 ports)

each port is pipelined (stages).

Back End, linked to memory

https://uops.info/background.html#supportedMicroarchitectures

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.

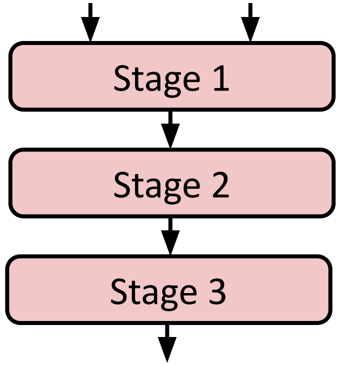
program instructions not necessarily executed in order.

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;
}
```

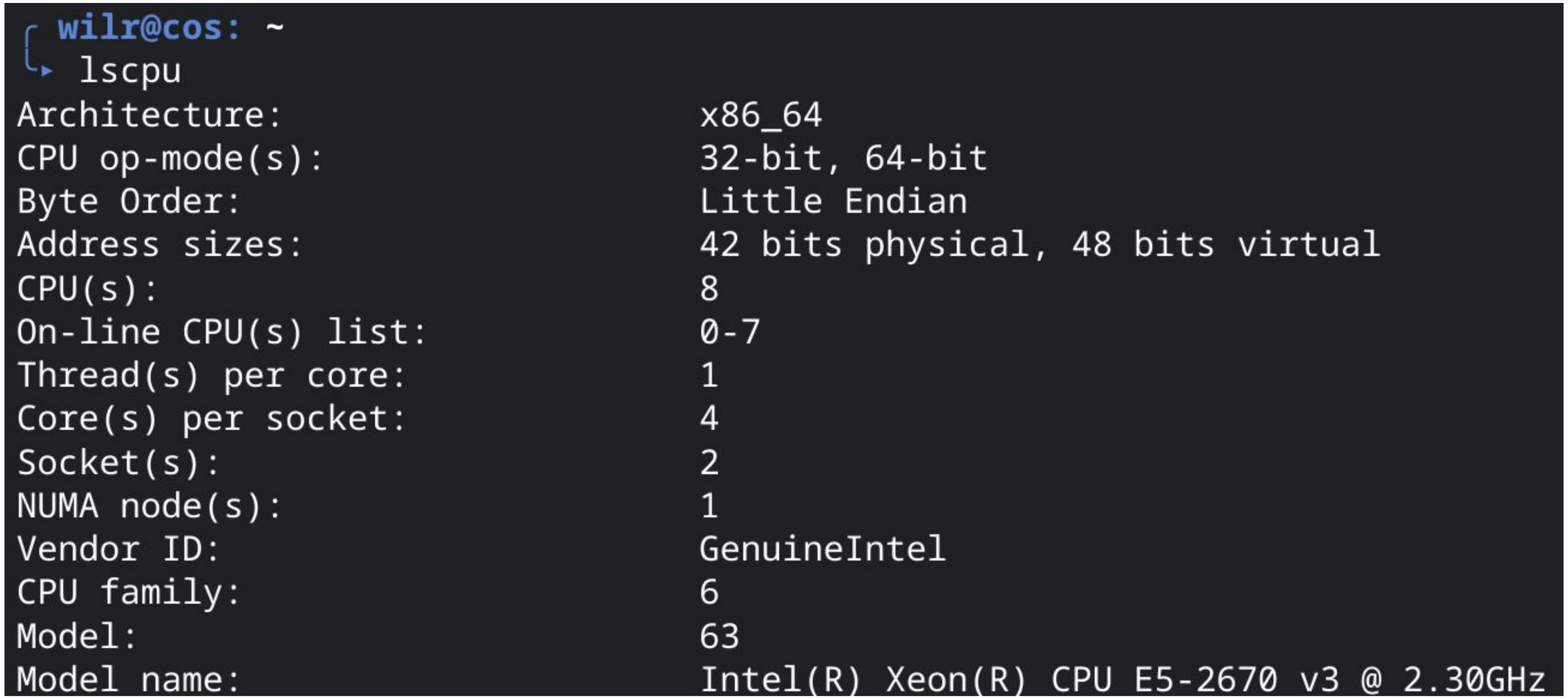


	Time						
	1	2	3	4	5	6	7
Stage 1	**a*b**	**a*c**			**p1*p2**		
Stage 2		**a*b**	**a*c**			**p1*p2**	
Stage 3			**a*b**	**a*c**			**p1*p2**

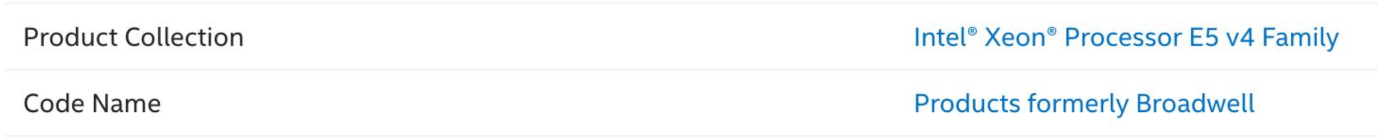
example w/
3-stage pipelines.

- 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

Cos CPU

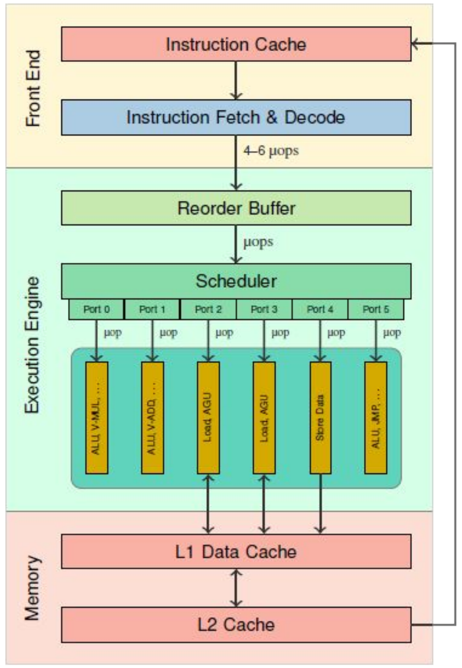


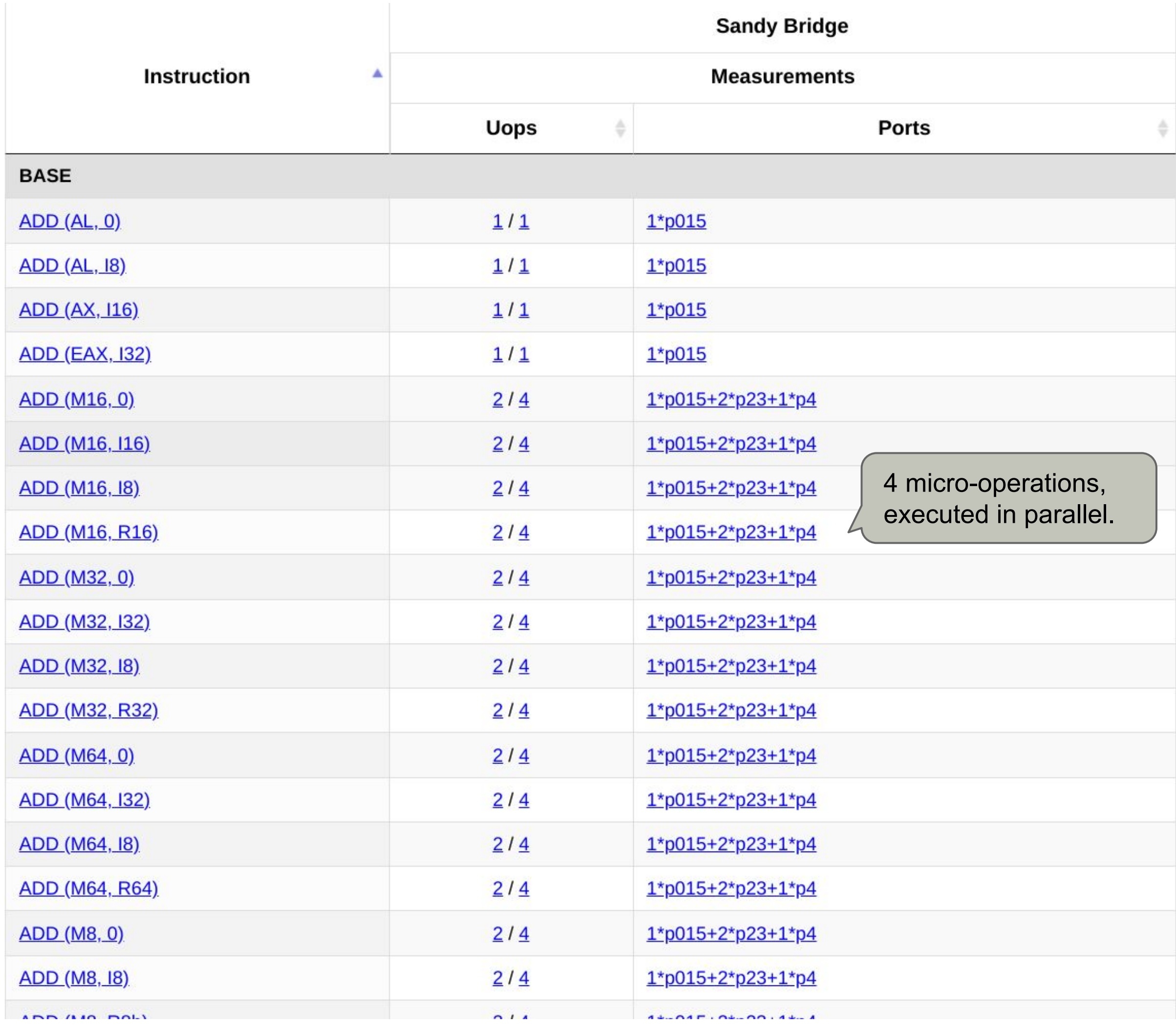
```
╭ wilr@cos: ~
╰▸ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   42 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              1
Core(s) per socket:              4
Socket(s):                       2
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           63
Model name:                      Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
```



Product Collection	Intel® Xeon® Processor E5 v4 Family
Code Name	Products formerly Broadwell

Sandy Bridge Pipelines





Instruction	Sandy Bridge	
	Measurements	
	Uops	Ports
BASE		
ADD (AL, 0)	1 / 1	1*p015
ADD (AL, I8)	1 / 1	1*p015
ADD (AX, I16)	1 / 1	1*p015
ADD (EAX, I32)	1 / 1	1*p015
ADD (M16, 0)	2 / 4	1*p015+2*p23+1*p4
ADD (M16, I16)	2 / 4	1*p015+2*p23+1*p4
ADD (M16, I8)	2 / 4	1*p015+2*p23+1*p4
ADD (M16, R16)	2 / 4	1*p015+2*p23+1*p4
ADD (M32, 0)	2 / 4	1*p015+2*p23+1*p4
ADD (M32, I32)	2 / 4	1*p015+2*p23+1*p4
ADD (M32, I8)	2 / 4	1*p015+2*p23+1*p4
ADD (M32, R32)	2 / 4	1*p015+2*p23+1*p4
ADD (M64, 0)	2 / 4	1*p015+2*p23+1*p4
ADD (M64, I32)	2 / 4	1*p015+2*p23+1*p4
ADD (M64, I8)	2 / 4	1*p015+2*p23+1*p4
ADD (M64, R64)	2 / 4	1*p015+2*p23+1*p4
ADD (M8, 0)	2 / 4	1*p015+2*p23+1*p4
ADD (M8, I8)	2 / 4	1*p015+2*p23+1*p4

4 micro-operations, executed in parallel.

https://uops.info/table.html?search=ADD&cb_uops=on&cb_ports=on&cb_SNB=on&cb_measurements=on&cb_base=on

x86-64 Compilation of Combine4

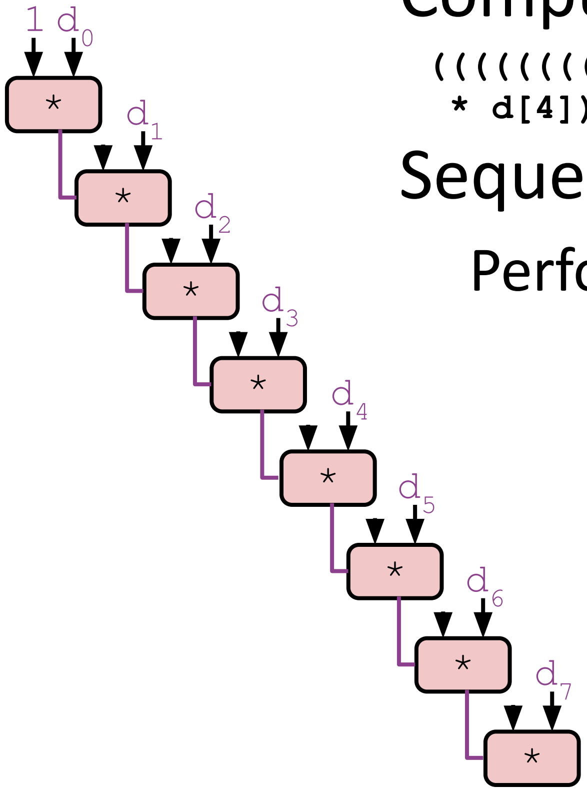
Inner Loop (Case: Integer Multiply)

```
.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**

we are already pretty close to theoretical bound w/o using pipelining. can we get closer?

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

w/o pipelining: serial.
how to do better?
loop unrolling.

Loop Unrolling (2x1)

we give the CPU opportunity to run ops in parallel…

```
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;
}
```

… by combining 2 elements at a time within 1 iteration of the loop.

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

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

Loop Unrolling with Reassociation (2x1a)

let's break
sequential dependency.

```
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

theoretical bound w/ parallelism taken into account (how many ports available, how many ports each instruction needs)

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)

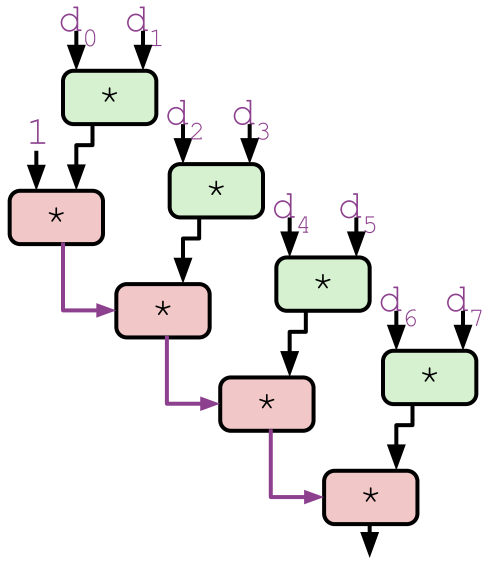
4 func. units for int +
2 func. units for load

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

Reassociated Computation

sequential dependency broken; can do more things at the same time.

```
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;
}
```

can do even better:
two accumulators.
(2 chains instead of 1)

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

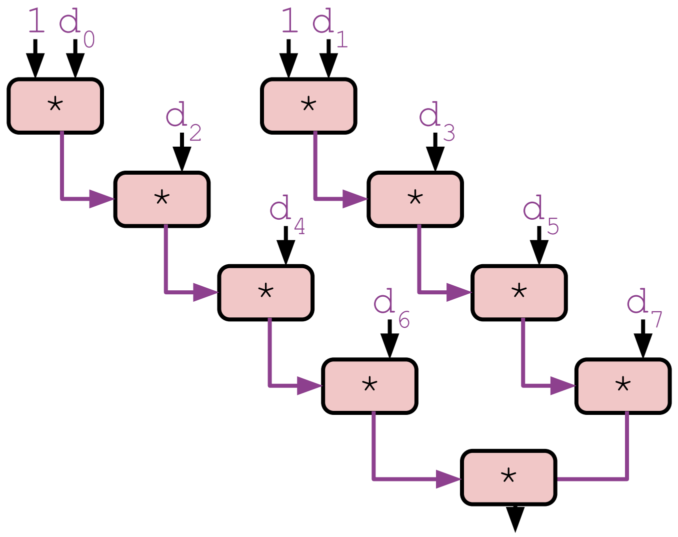
Int + makes use of two load units

```
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

how many ops at a time depends on the ops (how parallelizable), and number of ports (resources).

- Limitations

  Diminishing returns

  – Cannot go beyond throughput limitations of # ports

  Large overhead for short lengths

  – Finish off iterations sequentially

Achievable Performance

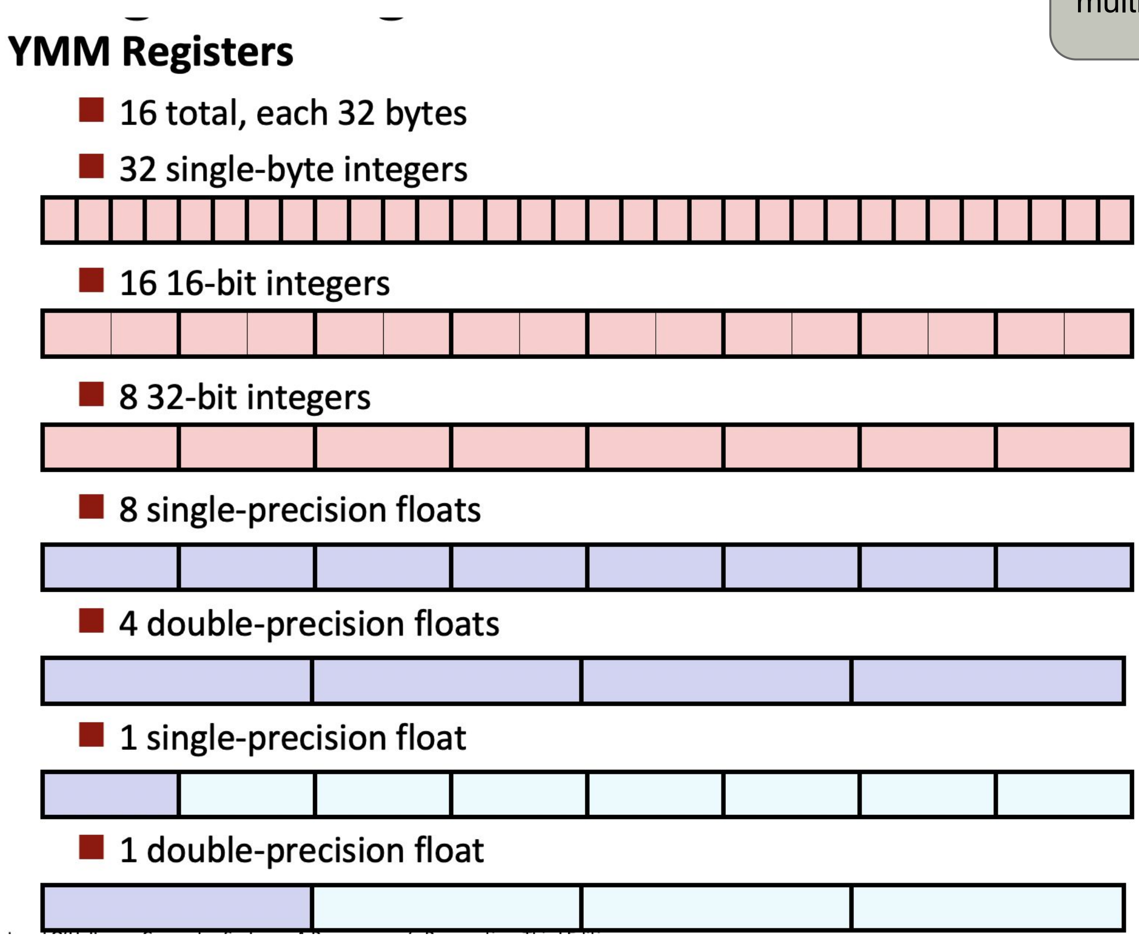
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Best	0.54	1.01	1.01	0.52
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	0.50	1.00	1.00	0.50

Limited only by throughput of functional units

Up to **42X improvement** over original, unoptimized code

Programming with AVX2

SIMD - single instruction multiple data



YMM Registers

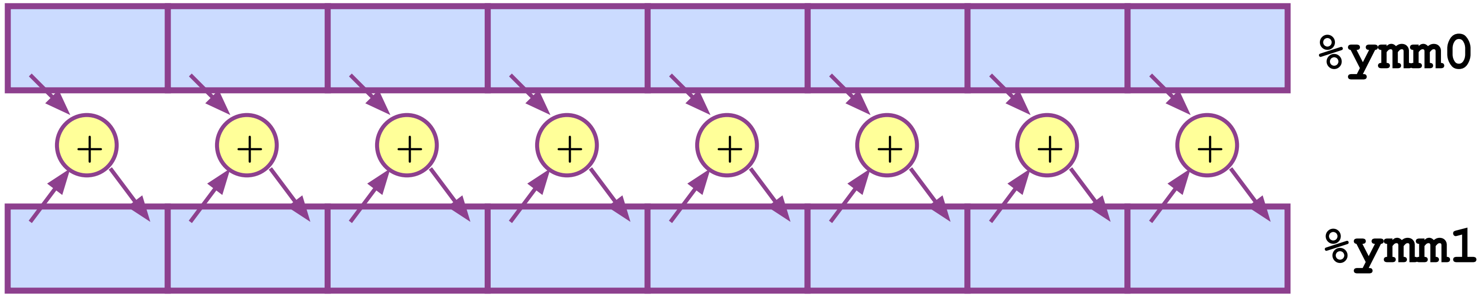
- 16 total, each 32 bytes
- 32 single-byte integers
- 16 16-bit integers
- 8 32-bit integers
- 8 single-precision floats
- 4 double-precision floats
- 1 single-precision float
- 1 double-precision float

SIMD Operations

add whole vectors,
in 1 instruction. (fast)

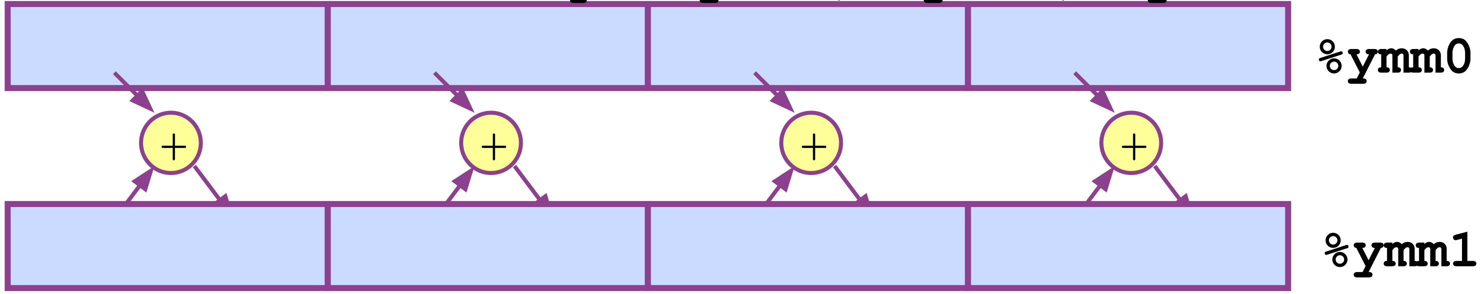
- SIMD Operations: Single Precision

`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

another order of magnitude improvement

Parallel operations on multiple data elements

See Web Aside OPT:SIMD on CS:APP web page

http://csapp.cs.cmu.edu/3e/waside/waside-simd.pdf

Outline

Overview
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

What About Branches?

Challenge

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

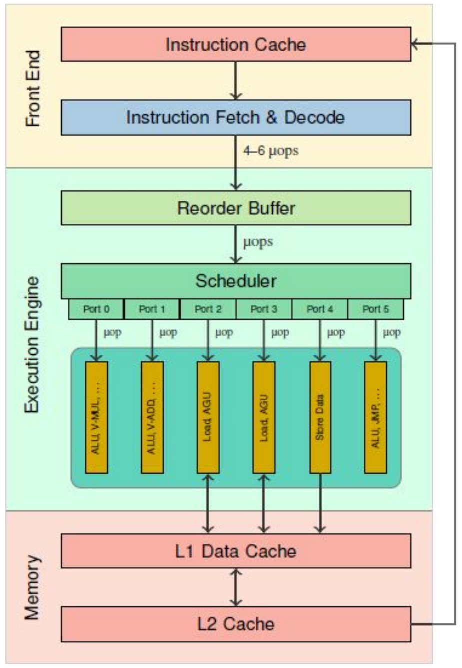
```
404663:  mov     $0x0,%eax
404668:  cmp     (%rdi),%rsi
40466b:  jge     404685
40466d:  mov     0x8(%rdi),%rax

 . . .

404685:  repz retq
```

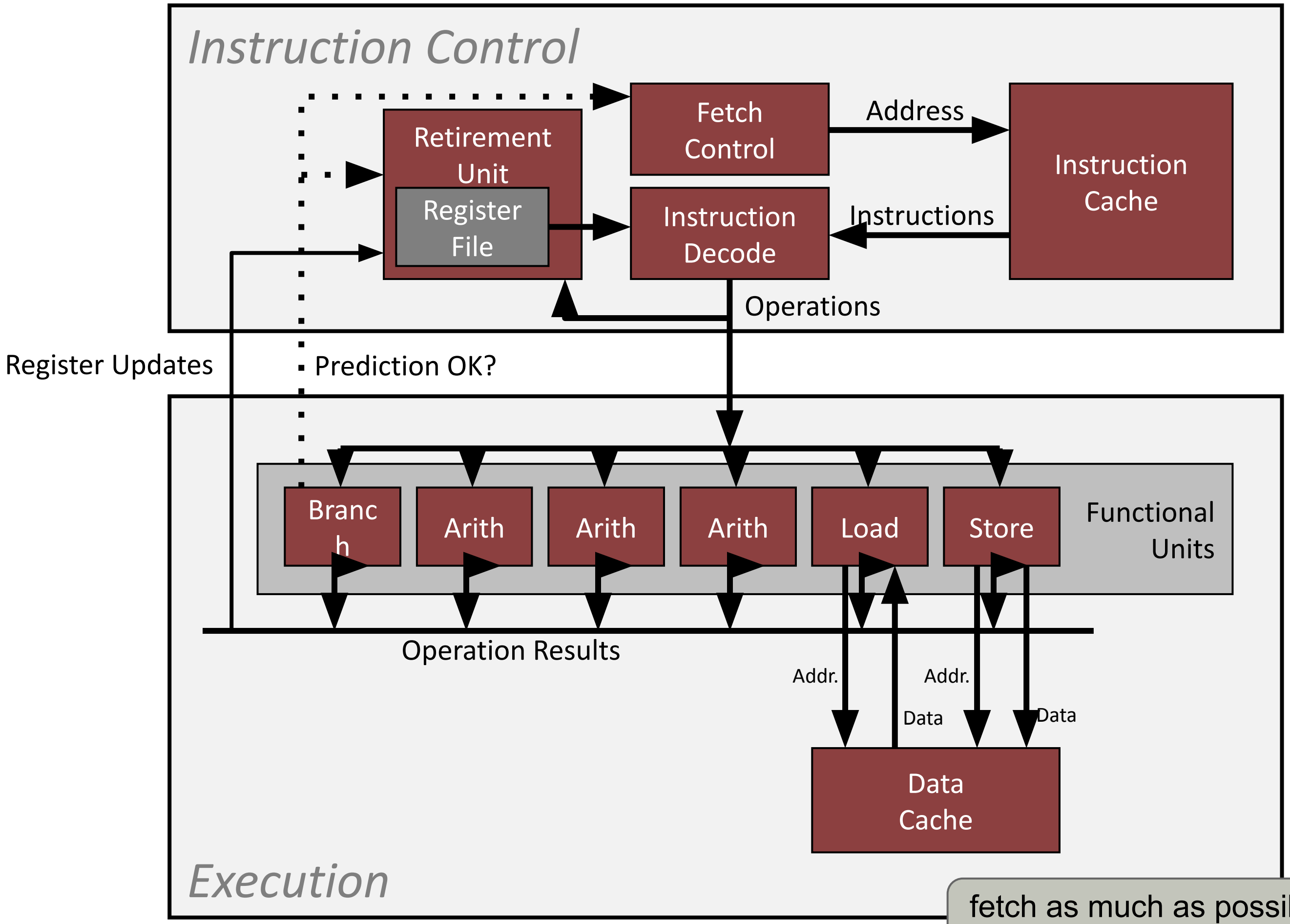
Executing

How to continue?

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

Modern CPU Design





fetch as much as possible to keep front end busy.

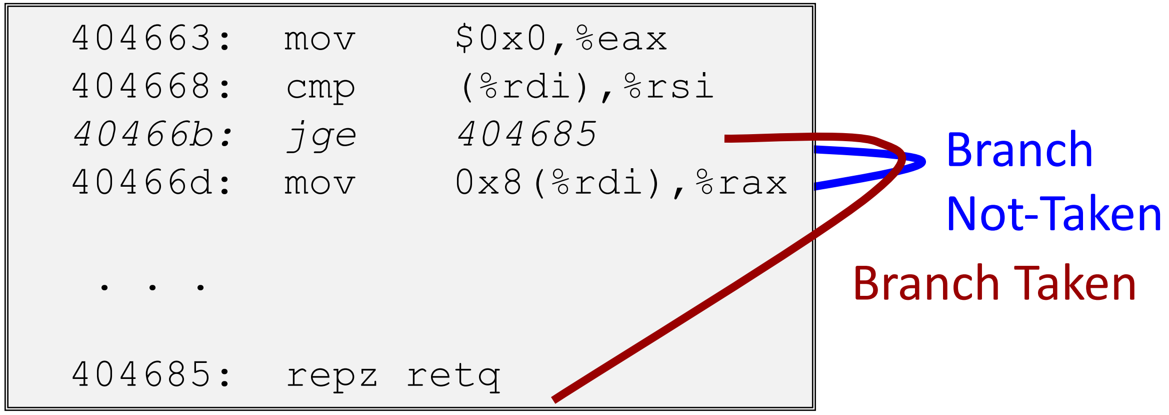
Branch Outcomes

"look, you are going through a loop. so I am going to guess 'true', and prefetch"

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
```

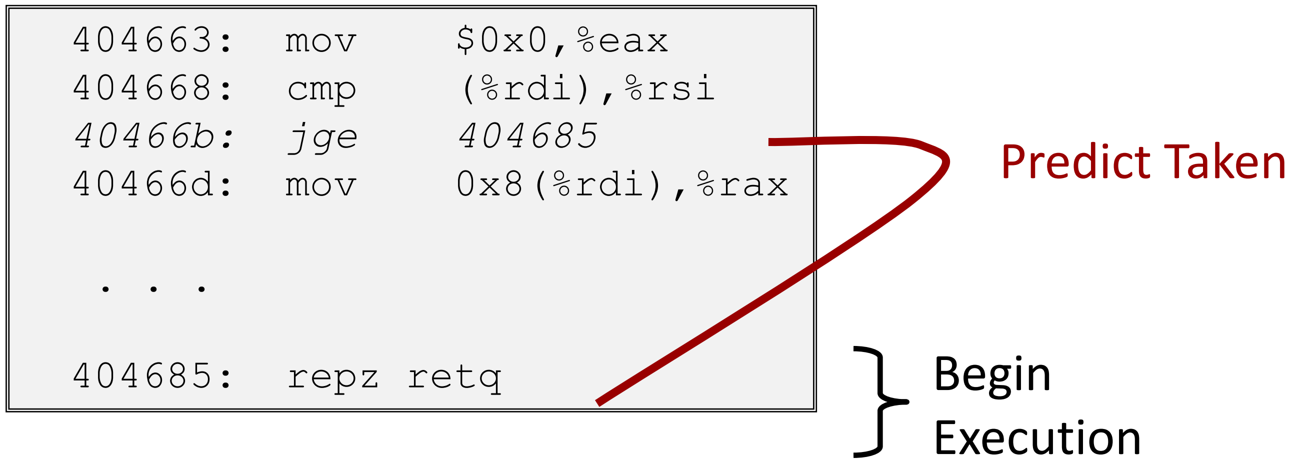
Branch Prediction

- Idea

  Guess which way branch will go

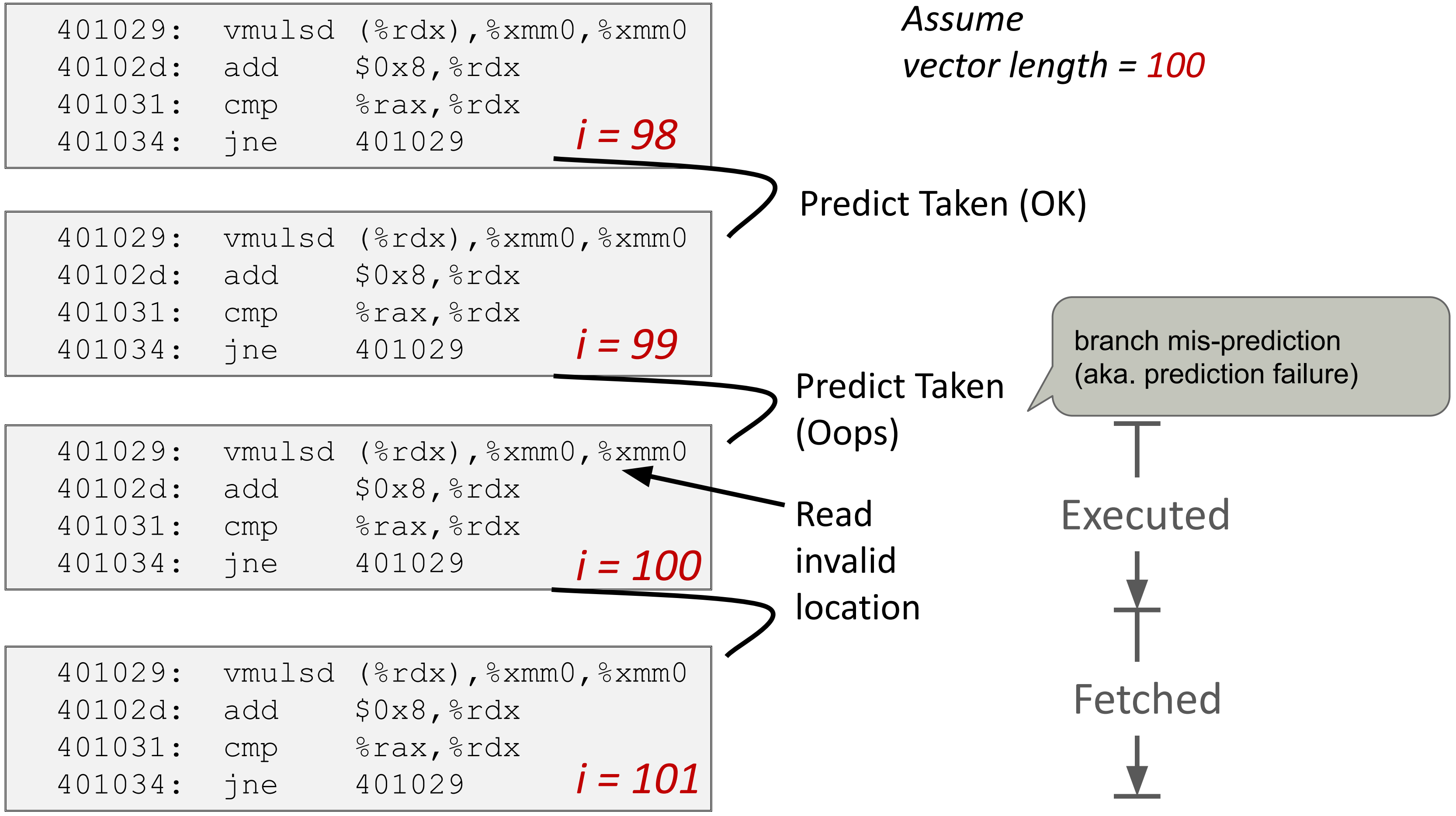
  Begin executing instructions at predicted position

  – 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
```

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

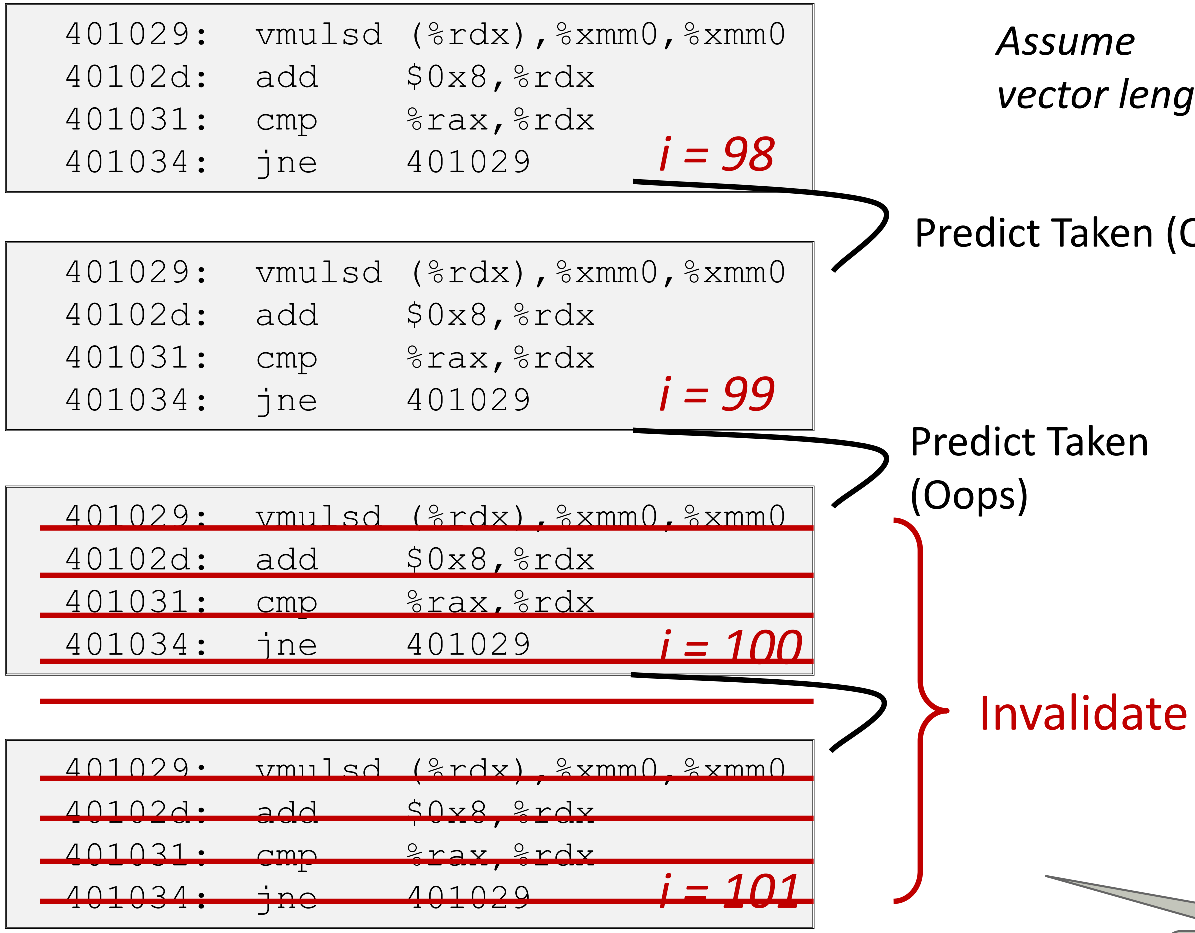
```
401029:   vmulsd (%rdx),%xmm0,%xmm0
40102d:   add    $0x8,%rdx
401031:   cmp    %rax,%rdx
401034:   jne    401029
```

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

Assume
vector length = 100

Branch Misprediction Invalidation

Assume
vector length = 100



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

i = 98

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

pipeline full of invalidated instructions (mispredict).

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

Take-Aways

- Leverage 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 (see last week => blocking)