Processor Architecture 3: Branch prediction, Out-of-order execution, optimization

Lecture 7 - 2015
Mads Chr. Olesen

Slides by Randal E. Bryant
Carnegie Mellon University

Overview

Wrap-Up of PIPE Design

- Performance analysis
- Fetch stage design
- Exceptional conditions

Modern High-Performance Processors

- Out-of-order execution

Performance Metrics

Clock rate

- Measured in Megahertz or Gigahertz
- Function of stage partitioning and circuit design
  - Keep amount of work per stage small

Rate at which instructions executed

- CPI: cycles per instruction
- On average, how many clock cycles does each instruction require?
- Function of pipeline design and benchmark programs
  - E.g., how frequently are branches mispredicted?

CPI for PIPE

CPI ≈ 1.0

- Fetch instruction each clock cycle
- Effectively process new instruction almost every cycle
  - Although each individual instruction has latency of 5 cycles

CPI > 1.0

- Sometimes must stall or cancel branches

Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected (C = I + B)

$$\text{CPI} = C/I = (I+B)/I = 1.0 + B/I$$

- Factor B/I represents average penalty due to bubbles

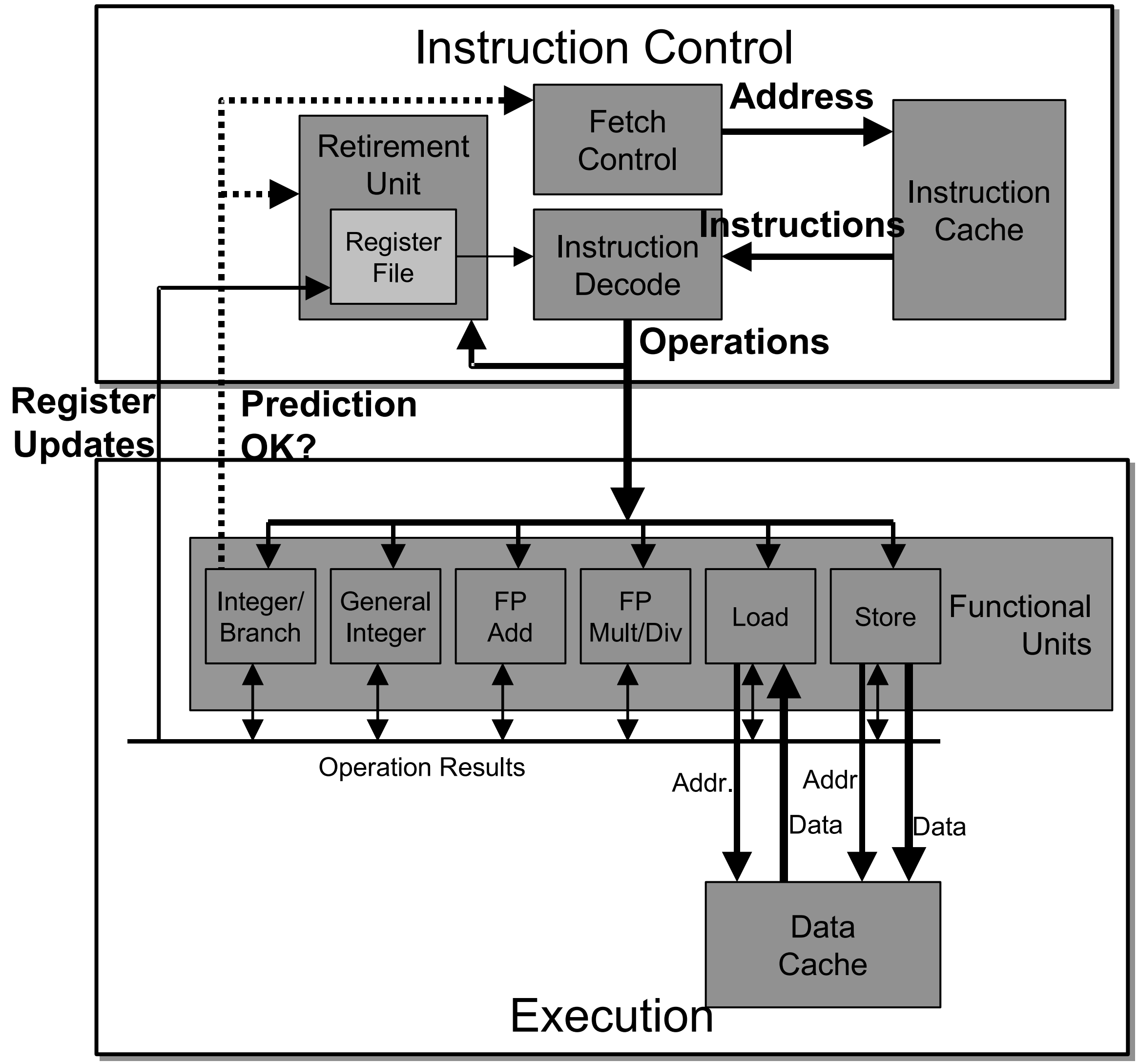
CPI for PIPE (Cont.)

B/I = LP + MP + RP

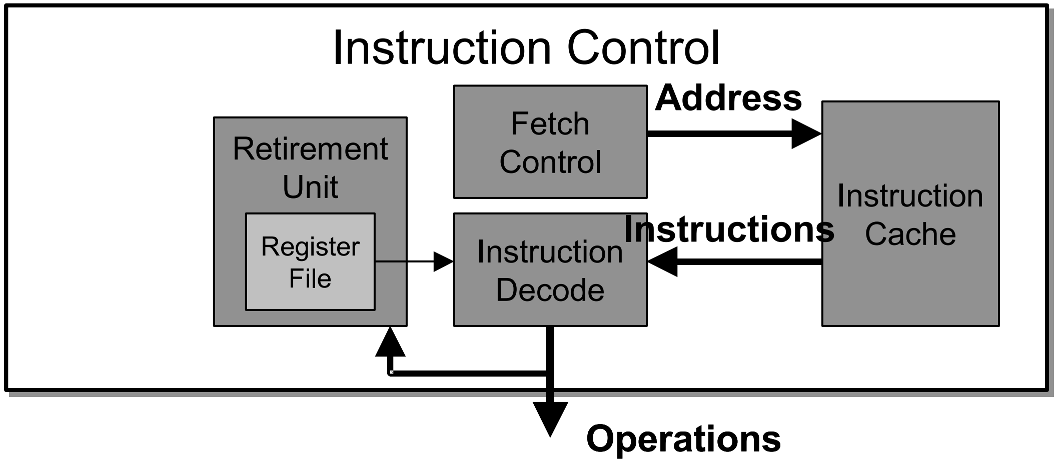
Typical Values

- LP: Penalty due to load/use hazard stalling
  - Fraction of instructions that are loads 0.25
  - Fraction of load instructions requiring stall 0.20
  - Number of bubbles injected each time 1
  - ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps 0.20
  - Fraction of cond. jumps mispredicted 0.40
  - Number of bubbles injected each time 2
  - ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- RP: Penalty due to `ret` instructions
  - Fraction of instructions that are returns 0.02
  - Number of bubbles injected each time 3
  - ⇒ RP = 0.02 * 3 = 0.06
- Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27
  - ⇒ CPI = 1.27 (Not bad!)

Modern CPU Design



Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

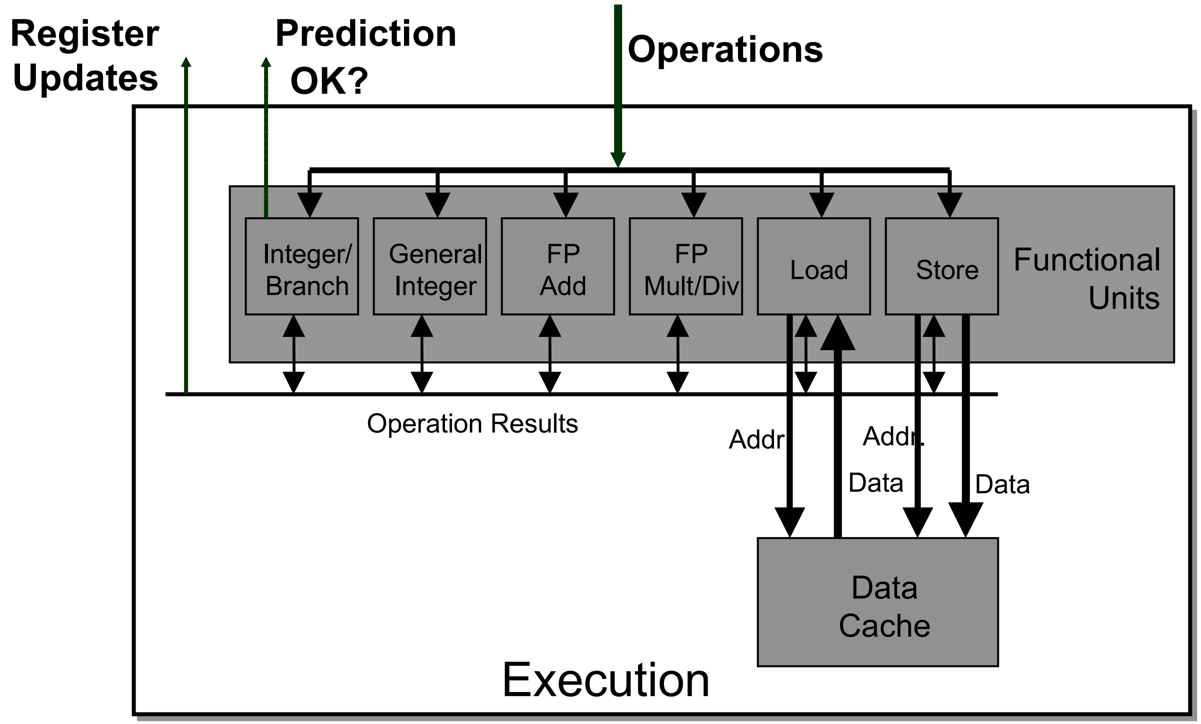
Translates Instructions Into *Operations*

- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*

- Abstract identifier linking destination of one operation with sources of later operations

Execution Unit



- Multiple functional units
  - Each can operate in independently
- Operations performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

CPU Capabilities of Pentium III

Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

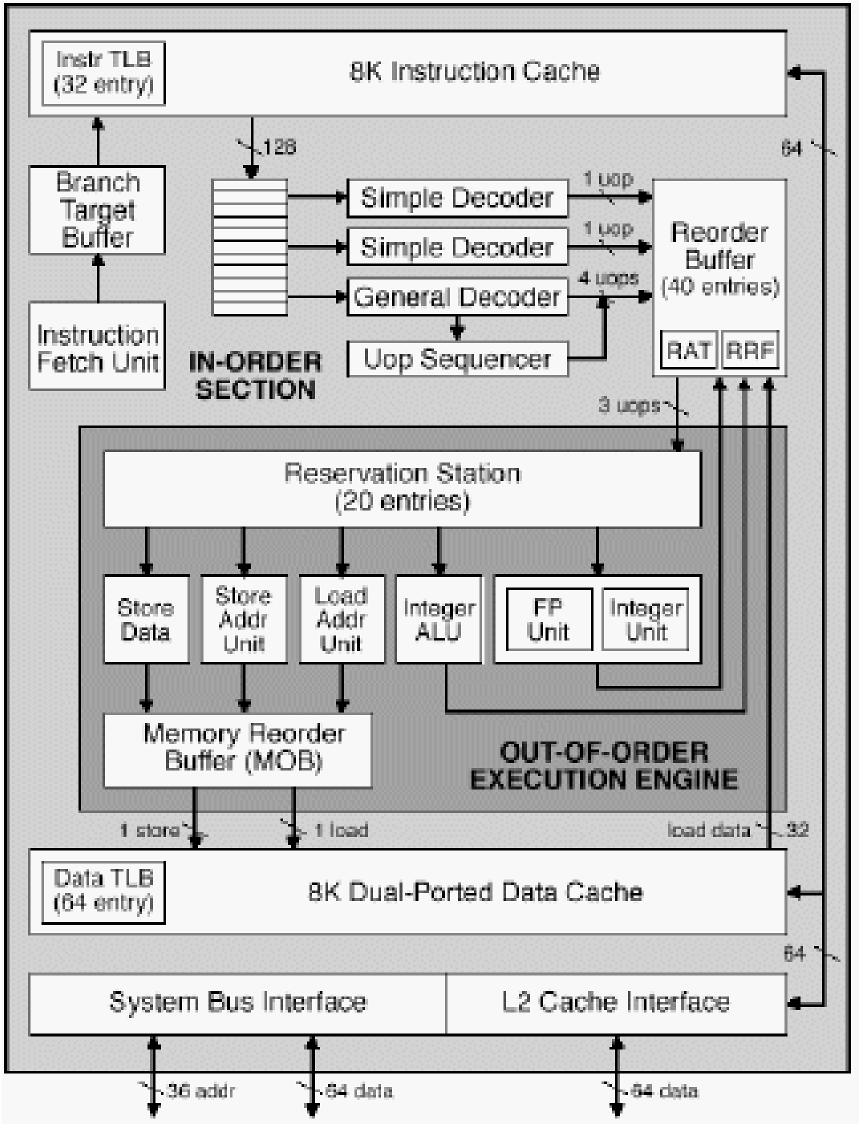
Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

PentiumPro Block Diagram

P6 Microarchitecture

- PentiumPro
- Pentium II
- Pentium III



Microprocessor Report
2/16/95

PentiumPro Operation

Translates instructions dynamically into “Uops”

- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

PentiumPro Branch Prediction

Critical to Performance

- 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken–not taken

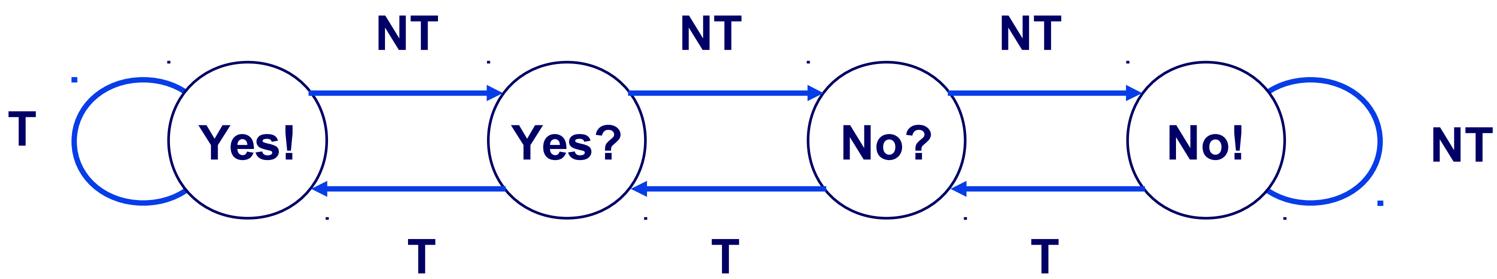
Handling BTB misses

- Detect in cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals

Example Branch Prediction

Branch History

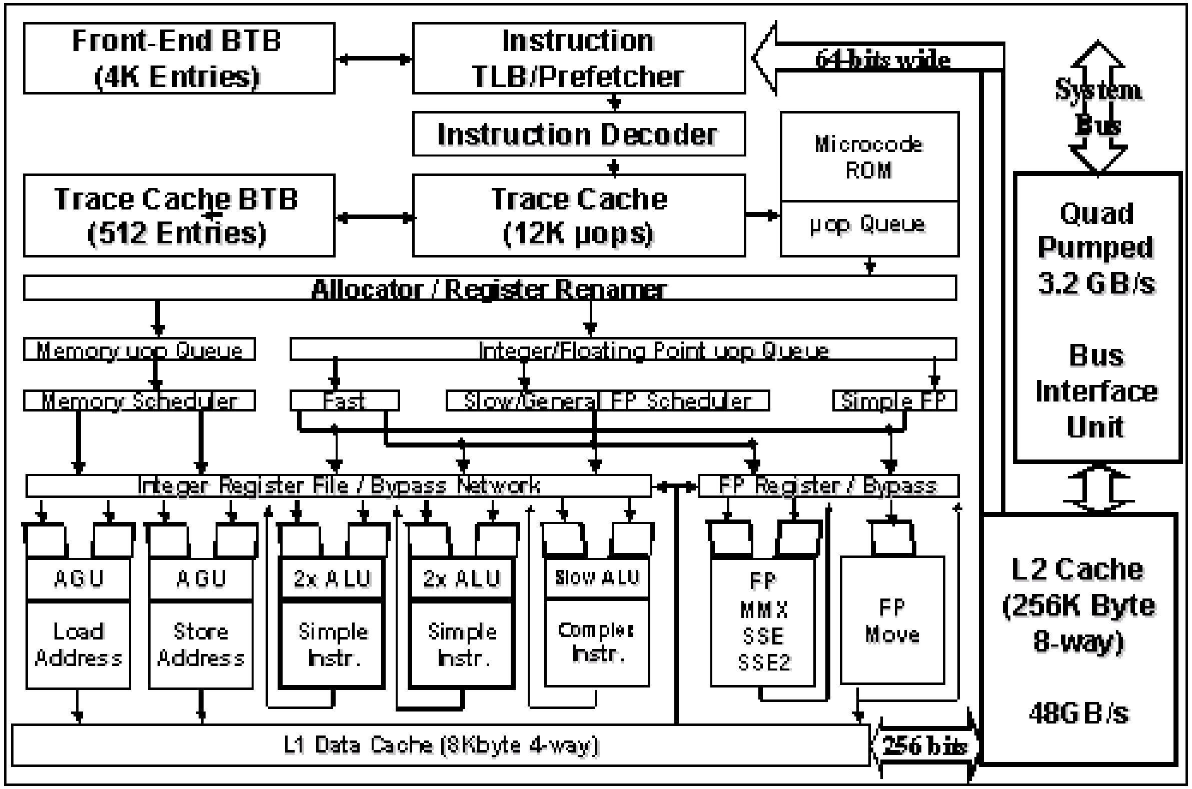
- Encode information about prior history of branch instructions
- Predict whether or not branch will be taken



State Machine

- Each time branch taken, transition to left
- When not taken, transition to right
- Predict branch taken when in state Yes! or Yes?

Pentium 4 Block Diagram

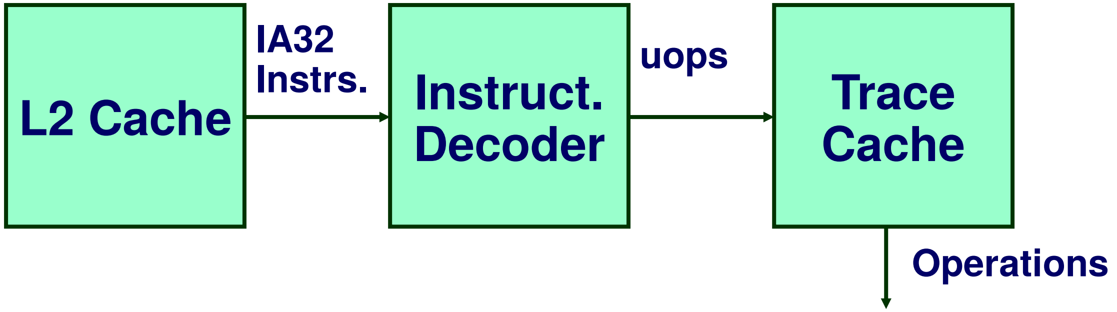


Intel Tech. Journal Q1, 2001

- More GHz!!!111one

Pentium 4 Features

Trace Cache



- Replaces traditional instruction cache
- Caches instructions in decoded form
- Reduces required rate for instruction decoder

Double-Pumped ALUs

- Simple instructions (add) run at 2X clock rate

Very Deep Pipeline

- 20+ cycle branch penalty
- Enables very high clock rates
- Slower than Pentium III for a given clock rate

Processor Summary

Design Technique

- Create uniform framework for all instructions
  - Want to share hardware among instructions
- Connect standard logic blocks with bits of control logic

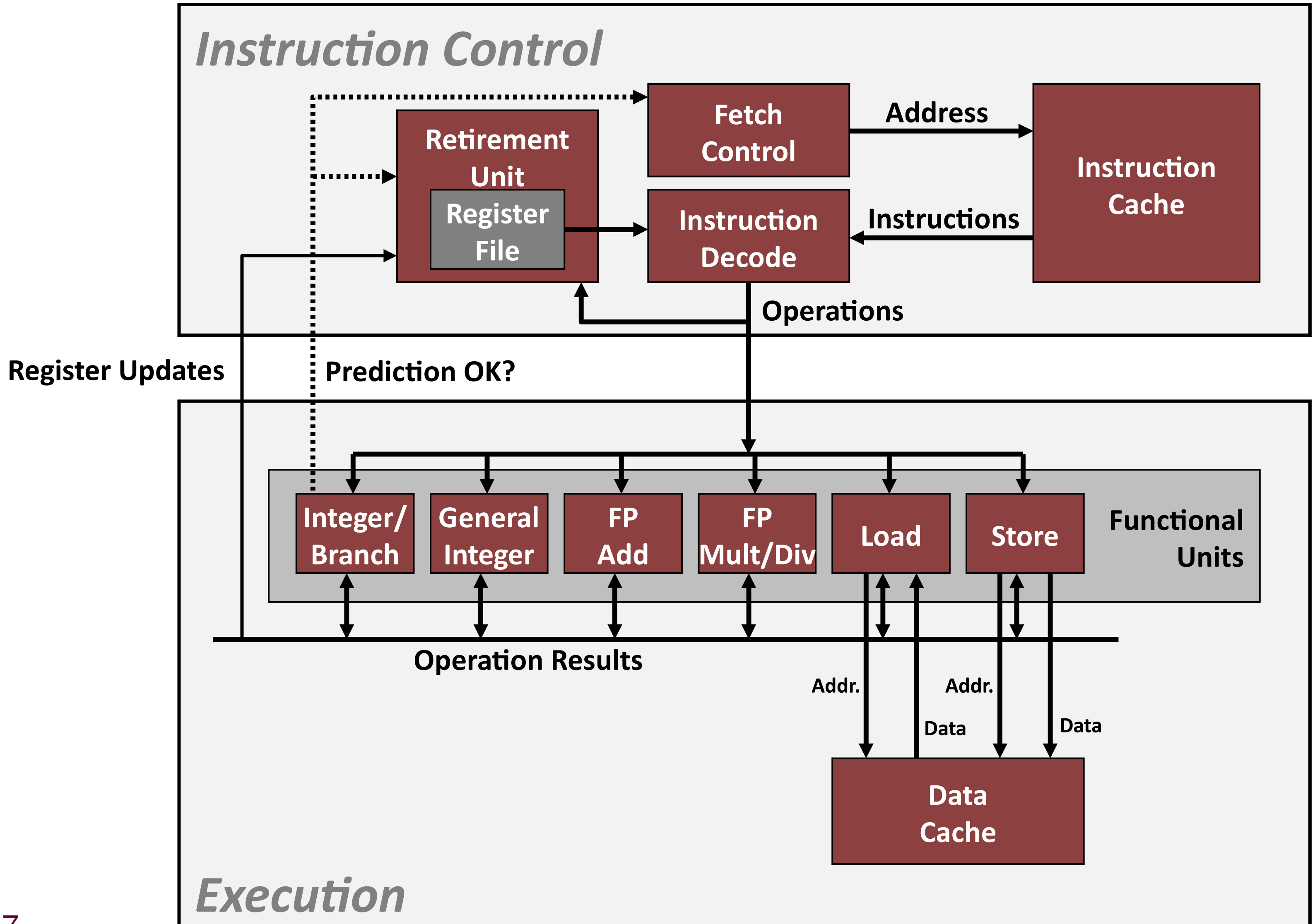
Operation

- State held in memories and clocked registers
- Computation done by combinational logic
- Clocking of registers/memories sufficient to control overall behavior

Enhancing Performance

- Pipelining increases throughput and improves resource utilization
- Must make sure maintains ISA behavior

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

Nehalem CPU

- **Multiple instructions can execute in parallel**

  1 load, with address computation

  1 store, with address computation

  2 simple integer (one may be branch)

  1 complex integer (multiply/divide)

  1 FP Multiply

  1 FP Add

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

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

Optimization: What the compiler does and doesn't

Today

- **Overview**
- **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**

Performance Realities

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

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

Optimizing Compilers

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

Limitations of Optimizing Compilers

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

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

Generally Useful Optimizations

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

- **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

```
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];
```

Where are the FP operations

```
set_row:
        testq   %rcx, %rcx             # Test n
        jle     .L4                    # If 0, goto done
        movq    %rcx, %rax             # rax = n
        imulq   %rdx, %rax             # rax *= i
        leaq    (%rdi,%rax,8), %rdx    # rowp = A + n*i*8
        movl    $0, %r8d               # j = 0
.L3:                                 # loop:
        movq    (%rsi,%r8,8), %rax     # t = b[j]
        movq    %rax, (%rdx)           # *rowp = t
        addq    $1, %r8                # j++
        addq    $8, %rdx               # rowp++
        cmpq    %r8, %rcx              # Compare n:j
        jg      .L3                    # If >, goto loop
.L4:                                 # done:
        rep ; ret
```

Reduction in Strength

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

→

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

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

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

```
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
```

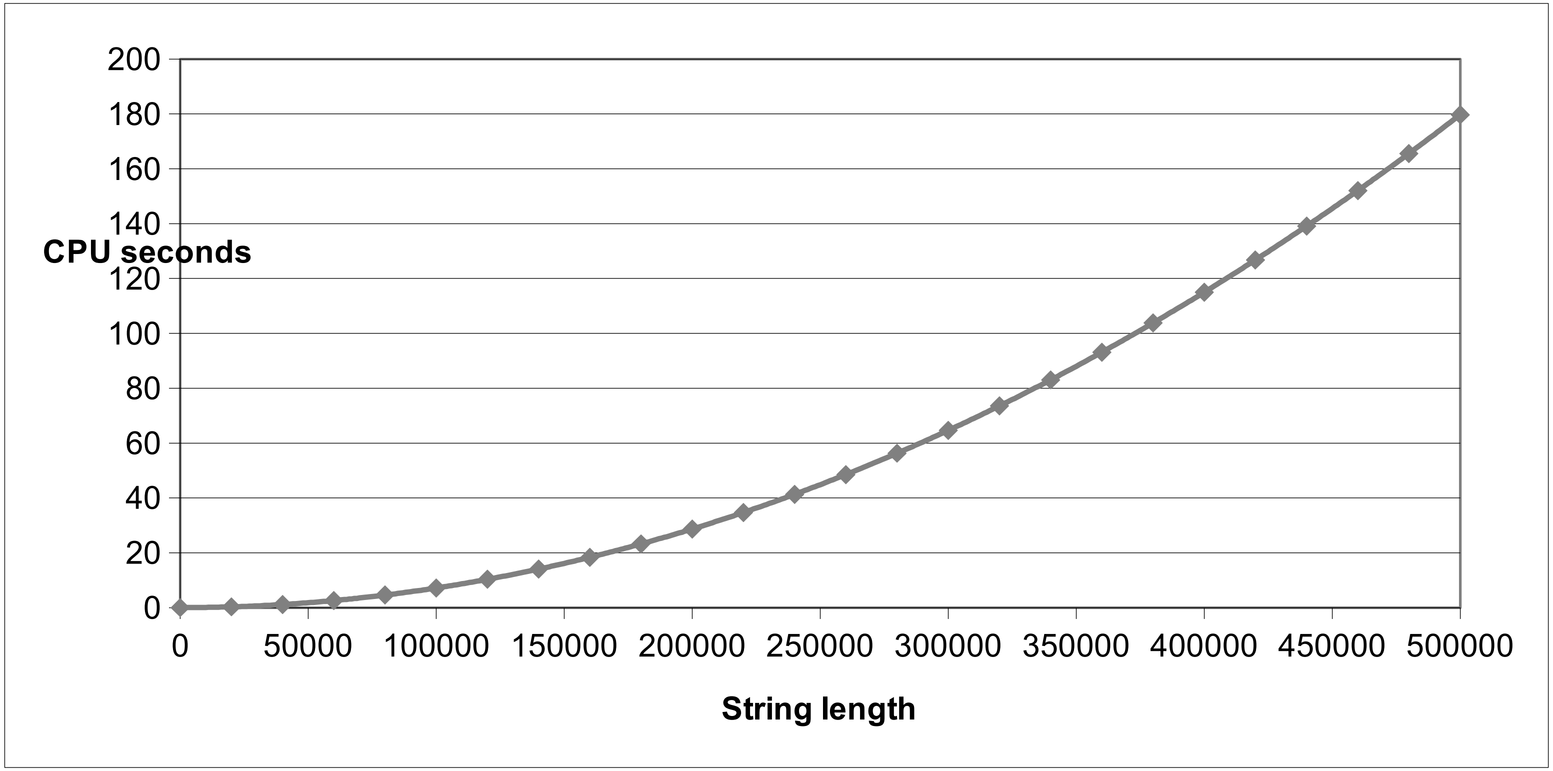
Optimization Blocker #1: Procedure Calls

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

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

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Convert Loop To Goto Form

```
void lower(char *s)
{
   int 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

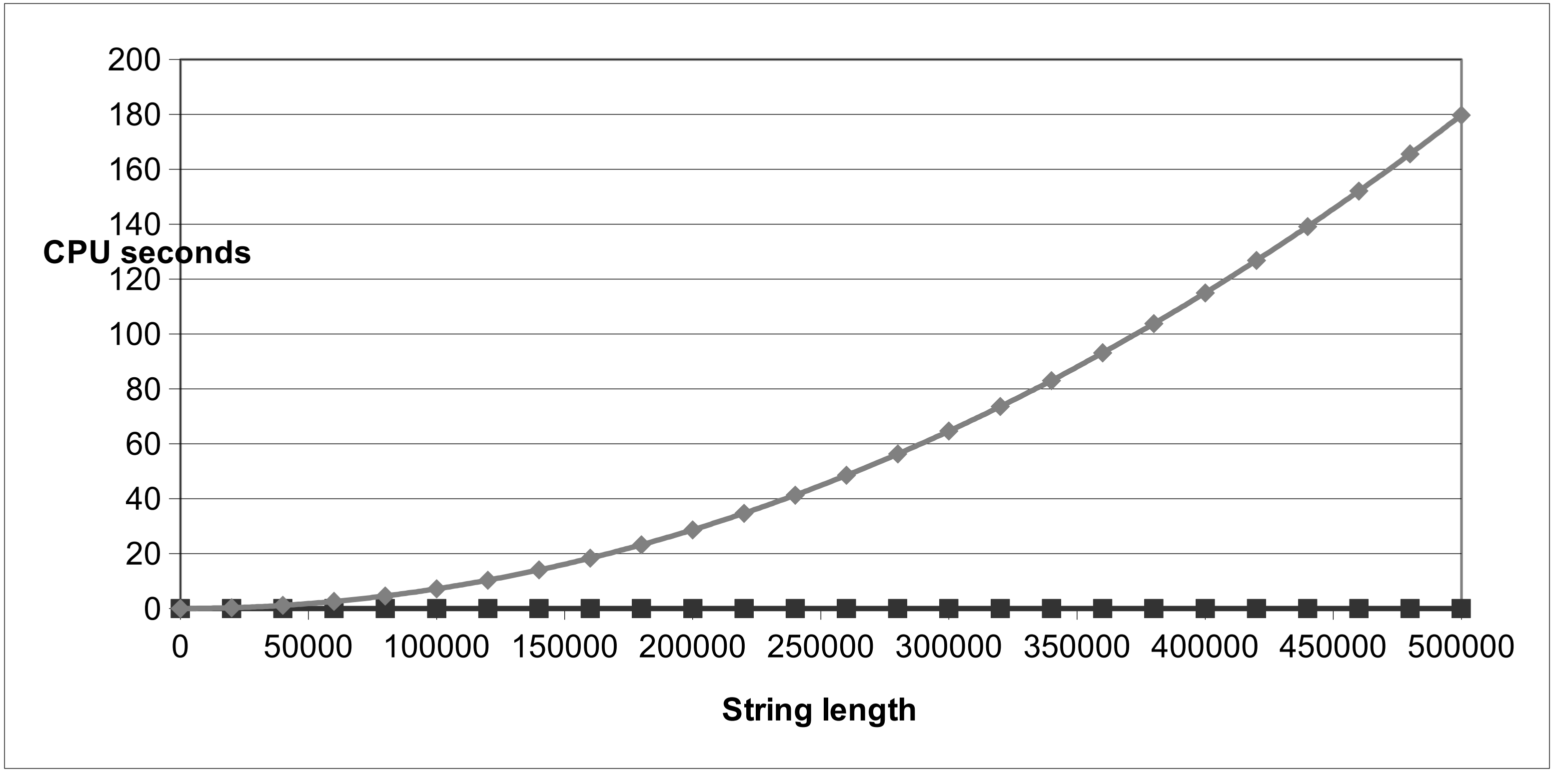
Improving Performance

```
void lower(char *s)
{
  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

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Optimization Blocker: Procedure Calls

- ***Why couldn't compiler move `strlen` out of inner loop?***
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- **Warning:**
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- **Remedies:**
  - Use of `inline` functions
    - GCC does this with –O2
    - See web aside ASM:OPT
  - Do your own code motion

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

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

Memory Aliasing

```
/* Sum 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
.L66:
        addsd   (%rcx), %xmm0   # FP Add
        addq    $8, %rcx
        decq    %rax
        jne     .L66
```

- No need to store intermediate results

Optimization Blocker: Memory Aliasing

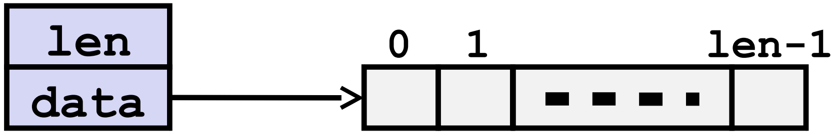
- **Aliasing**
  - Two different memory references specify single location
  - Easy to have happen in C
    - Since allowed to do address arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler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

Exploiting Instruction-Level Parallelism

- **Need general understanding of modern processor design**
  - Hardware can execute multiple instructions in parallel
- **Performance limited by data dependencies**
- **Simple transformations can have 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{
        int len;
        double *data;
} vec;
```




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

Benchmark Computation

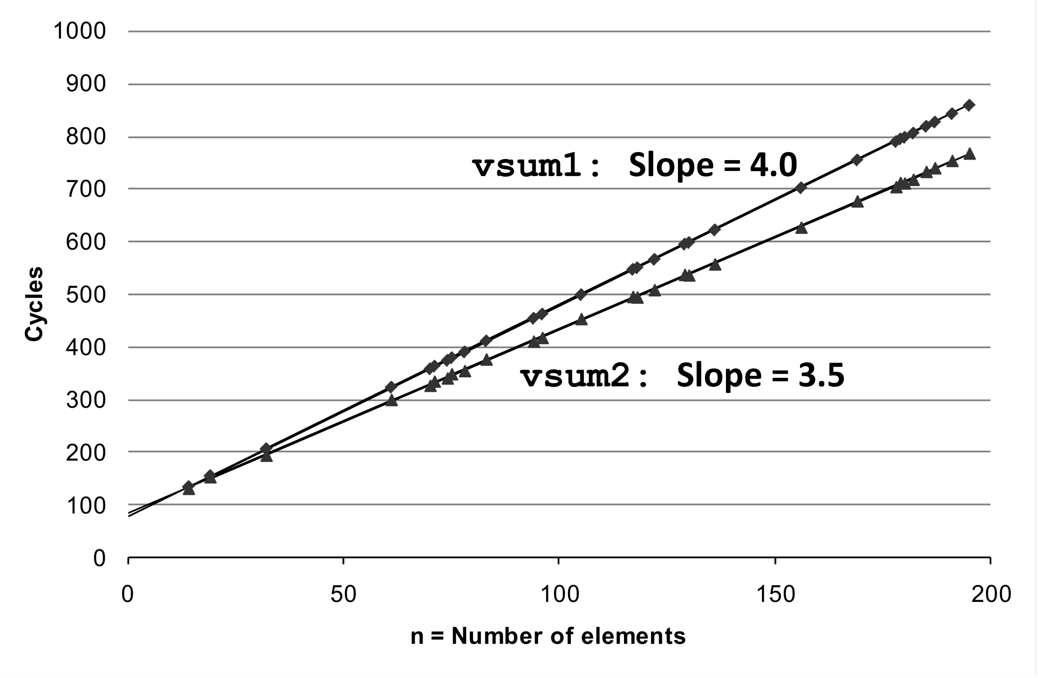
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

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

Cycles Per Element (CPE)

- **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	29.0	29.2	27.4	27.9
Combine1 –O1	12.0	12.0	12.0	13.0

Basic Optimizations

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

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

Effect of Basic Optimizations

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 –O1	12.0	12.0	12.0	13.0
Combine4	2.0	3.0	3.0	5.0

- Eliminates sources of overhead in loop

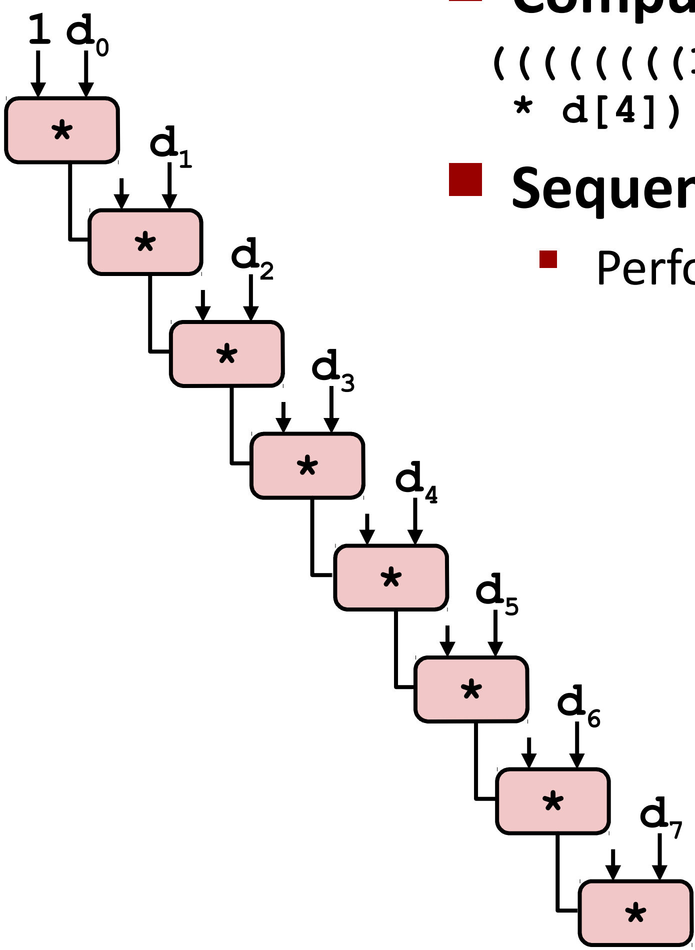
x86-64 Compilation of Combine4

- **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	2.0	3.0	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

Combine4 = Serial Computation (OP = *)



- **Computation (length=8)**

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

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

Loop Unrolling

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

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

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

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

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

Loop Unrolling with Reassociation

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

Compare to before

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

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

Effect of Reassociation

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

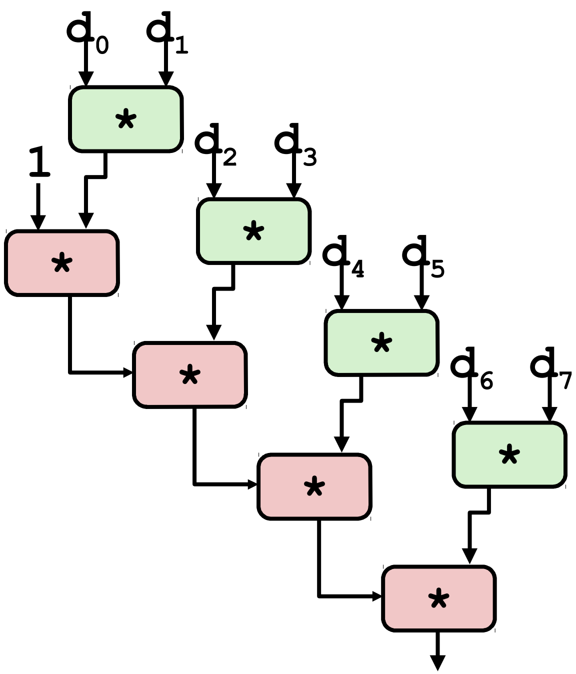
- **Nearly 2x speedup for Int *, FP +, FP ***
  - Reason: Breaks sequential dependency

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

  - Why is that? (next slide)

Reassociated Computation

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



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

Loop Unrolling with Separate Accumulators

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

- Different form of reassociation

Effect of Separate Accumulators

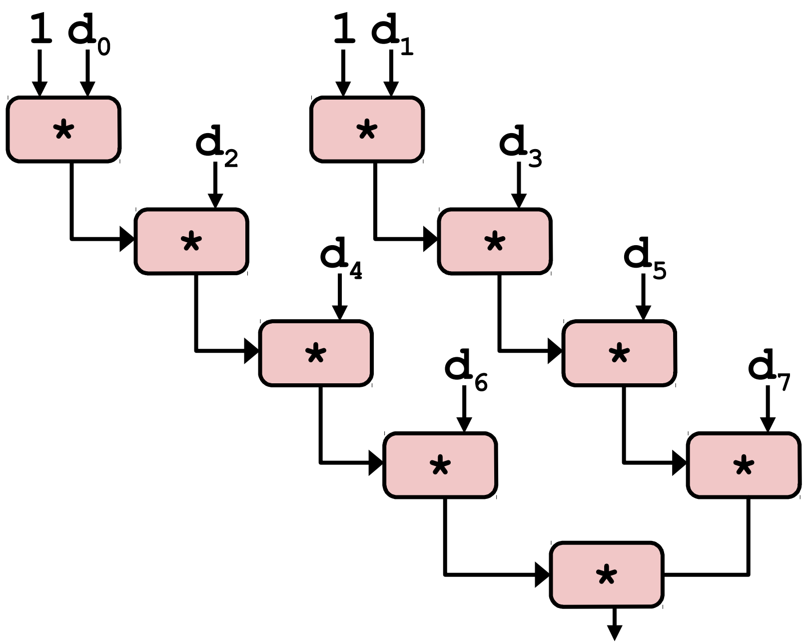
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Unroll 2x, reassociate	2.0	1.5	1.5	3.0
Unroll 2x Parallel 2x	1.5	1.5	1.5	2.5
Latency Bound	1.0	3.0	3.0	5.0
Throughput Bound	1.0	1.0	1.0	1.0

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

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

Separate Accumulators

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



- **What changed:**
  - Two independent "streams" of operations
- **Overall Performance**
  - N elements, D cycles latency/op
  - Should be (N/2+1)*D cycles: **CPE = D/2**
  - CPE matches prediction!

What Now?

Unrolling & Accumulating

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

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

Unrolling & Accumulating: Double *

- **Case**
  - Intel Nehalem
  - Double FP Multiplication
  - Latency bound: 5.00. Throughput bound: 1.00

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

Accumulators

Unrolling & Accumulating: Int +

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

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

Accumulators

Achievable Performance

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Scalar Optimum	1.00	1.00	1.00	1.00
Latency Bound	1.00	3.00	3.00	5.00
Throughput Bound	1.00	1.00	1.00	1.00

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

Using Vector Instructions

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

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

What About Branches?

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

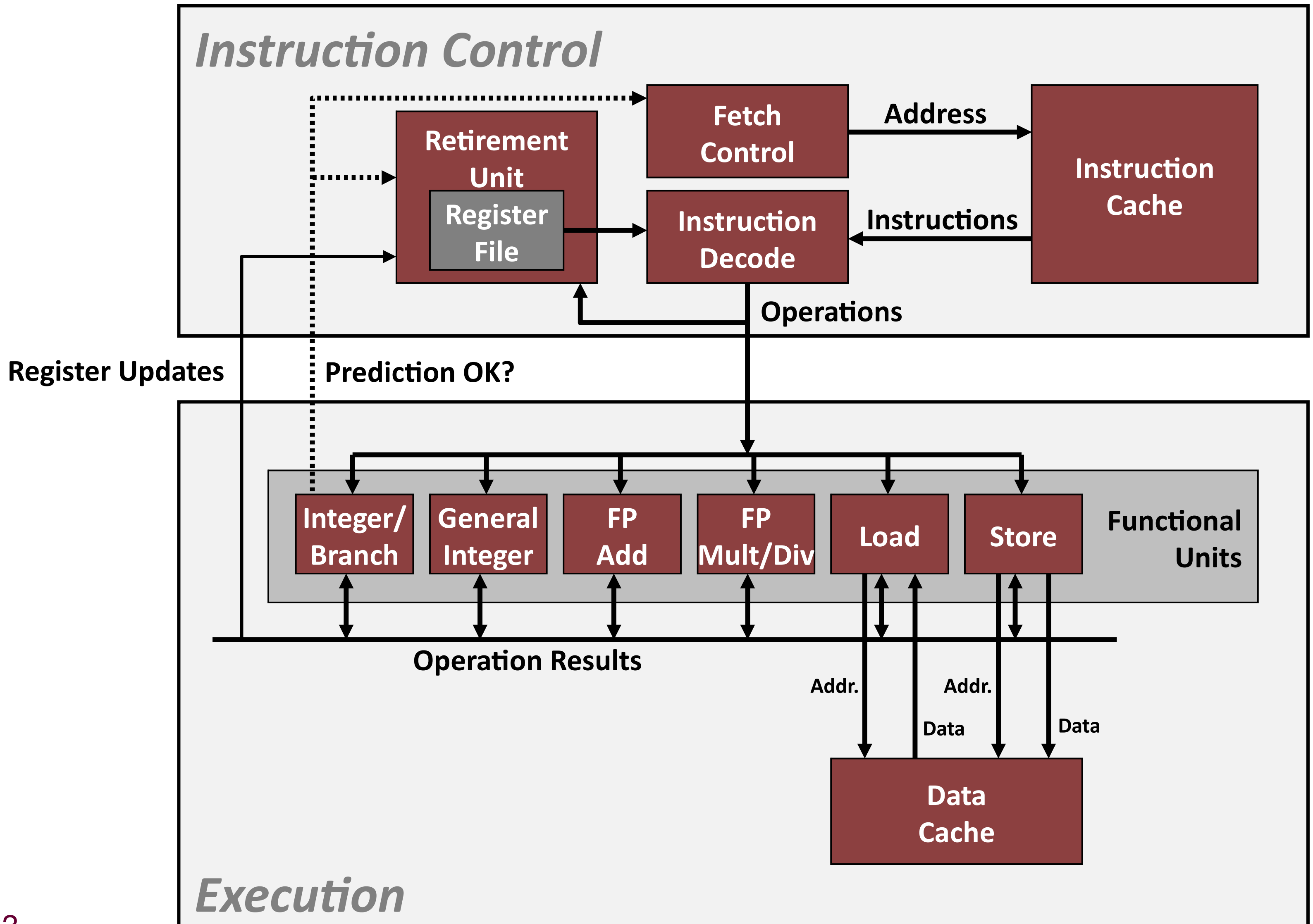
```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%eax,%edx,4),%ecx
```

Executing

How to continue?

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

Modern CPU Design



Branch Outcomes

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

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

Branch Not-Taken

Branch Taken

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

Branch Prediction

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

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

Predict Taken

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

Begin Execution

Branch Prediction Through Loop

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

i = 98

Predict Taken (OK)

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

i = 99

Predict Taken (Oops)

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

i = 100

Read invalid location

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

i = 101

Assume vector length = 100

Executed

Fetched

Branch Misprediction Invalidation

Assume
vector length = 100

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

i = 98

Predict Taken (OK)

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

i = 99

Predict Taken (Oops)

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

i = 100

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

i = 101

Invalidate

Branch Misprediction Recovery

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

i = 99

Definitely not taken

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

Effect of Branch Prediction

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

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

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Combine4b	4.0	4.0	4.0	5.0

Getting High Performance

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

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