

Carnegie Mellon

Introduction to Computer Systems

15-213/18-243, fall 2009
8th Lecture, Sep. 17th

Instructors:
Roger B. Dannenberg and Greg Ganger

Carnegie Mellon

Last Time

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Callee saved
%rdx	Argument #3	%r11	Used for linking
%rsi	Argument #2	%r12	C: Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

2

Carnegie Mellon

Last Time

- Procedures (x86-64): Optimizations
 - No base/frame pointer
 - Passing arguments to functions through registers (if possible)
 - Sometimes: Writing into the “red zone” (below stack pointer)

rtn Ptr ← %rsp

-8 unused

-16 loc[1]

-24 loc[0]

- Sometimes: Function call using jmp (instead of call)
- Reason: Performance
 - use stack as little as possible
 - while obeying rules (e.g., caller/callee save registers)

3

Carnegie Mellon

Last Time

- Arrays

int val[5];

1 5 2 1 3

x x+4 x+8 x+12 x+16 x+20
- Nested

int pgh[4][5];

1 5 2 0 6 1 5 2 1 3 1 5 2 1 7 1 5 2 2 1

76 96 116 136 156
- Multi-level

int *univ[3]

univ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

4

Carnegie Mellon

Dynamic Nested Arrays

- Strength
 - Can create matrix of any size
- Programming
 - Must do index computation explicitly
- Performance
 - Accessing single element costly
 - Must do multiplication

int *new_var_matrix(int n){ return (int *) calloc(sizeof(int), n*n); }

int var_ele (int *a, int i, int j, int n){ return a[i*n+j]; }

movl 12(%ebp),%eax # i

movl 8(%ebp),%edx # a

imull 20(%ebp),%eax # n*i

addl 16(%ebp),%eax # n*i+j

movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]

5

Carnegie Mellon

Dynamic Array Multiplication

- Per iteration:
 - Multiplies: 3
 - 2 for subscripts
 - 1 for data
 - Adds: 4
 - 2 for array indexing
 - 1 for loop index
 - 1 for data

/* Compute element i,k of variable matrix product */

int var_prod_ele (int *a, int *b, int i, int k, int n)

{

int j;

int result = 0;

for (j = 0; j < n; j++)

result +=

a[i*n+j] * b[j*n+k];

return result;

}

a

x

b

j-th column

i-th row

6

Optimizing Dynamic Array Multiplication

Optimizations

- Performed when set optimization level to -O2

Code Motion

- Expression $i*n$ can be computed outside loop

Strength Reduction

- Incrementing j has effect of incrementing $j*n+k$ by n

Operations count

- 4 adds, 1 mult

```

{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}

```

4 adds, 3 mults

```

{
    int j;
    int result = 0;
    int iTn = i*n;
    int jTnPk = k;
    for (j = 0; j < n; j++) {
        result +=
            a[iTn+j] * b[jTnPk];
        jTnPk += n;
    }
    return result;
}

```

4 adds, 1 mult

Today

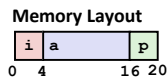
- Structures
- Alignment
- Unions
- Floating point

Structures

```

struct rec {
    int i;
    int a[3];
    int *p;
};

```



Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

Accessing Structure Member

```

void
set_i(struct rec *r,
      int val)
{
    r->i = val;
}

```

IA32 Assembly

```

# %eax = val
# %edx = r
movl %eax, (%edx) # Mem[r] = val

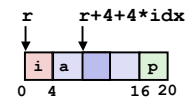
```

Generating Pointer to Structure Member

```

struct rec {
    int i;
    int a[3];
    int *p;
};

```



```

int *find_a
(struct rec *r, int idx)
{
    return &r->a[idx];
}

```

What does it do?

```

# %ecx = idx
# %edx = r
leal 0(, %ecx, 4), %eax
leal 4(%eax, %edx), %eax

```

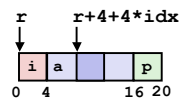
Will disappear
blackboard?

Generating Pointer to Structure Member

```

struct rec {
    int i;
    int a[3];
    int *p;
};

```



Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```

int *find_a
(struct rec *r, int idx)
{
    return &r->a[idx];
}

```

```

# %ecx = idx
# %edx = r
leal 0(, %ecx, 4), %eax # 4*idx
leal 4(%eax, %edx), %eax # r+4*idx+4

```

Structure Referencing (Cont.)

C Code

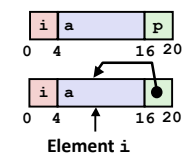
```

struct rec {
    int i;
    int a[3];
    int *p;
};

void
set_p(struct rec *r)
{
    r->p =
        &r->a[r->i];
}

```

What does it do?



```

# %edx = r
movl (%edx), %ecx # r->i
leal 0(, %ecx, 4), %eax # 4*(r->i)
leal 4(%edx, %eax), %eax # r+4+4*(r->i)
movl %eax, 16(%edx) # Update r->p

```

Today

- Structures
- Alignment
- Unions
- Floating point

Carnegie Mellon

13

Alignment

- **Aligned Data**
 - Primitive data type requires K bytes
 - Address must be multiple of K
 - Required on some machines; advised on IA32
 - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **Motivation for Aligning Data**
 - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
 - Inefficient to load or store datum that spans quad word boundaries
 - Virtual memory very tricky when datum spans 2 pages
- **Compiler**
 - Inserts gaps in structure to ensure correct alignment of fields

Carnegie Mellon

14

Specific Cases of Alignment (IA32)

- **1 byte: char, ...**
 - no restrictions on address
- **2 bytes: short, ...**
 - lowest 1 bit of address must be 0₂
- **4 bytes: int, float, char *, ...**
 - lowest 2 bits of address must be 00₂
- **8 bytes: double, ...**
 - Windows (and most other OS's & instruction sets):
 - lowest 3 bits of address must be 000₂
 - Linux:
 - lowest 2 bits of address must be 00₂
 - i.e., treated the same as a 4-byte primitive data type
- **12 bytes: long double**
 - Windows, Linux:
 - lowest 2 bits of address must be 00₂
 - i.e., treated the same as a 4-byte primitive data type

Carnegie Mellon

15

Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
 - no restrictions on address
- **2 bytes: short, ...**
 - lowest 1 bit of address must be 0₂
- **4 bytes: int, float, ...**
 - lowest 2 bits of address must be 00₂
- **8 bytes: double, char *, ...**
 - Windows & Linux:
 - lowest 3 bits of address must be 000₂
- **16 bytes: long double**
 - Linux:
 - lowest 3 bits of address must be 000₂
 - i.e., treated the same as a 8-byte primitive data type

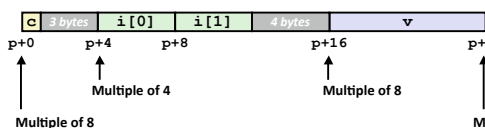
Carnegie Mellon

16

Satisfying Alignment with Structures

- **Within structure:**
 - Must satisfy element's alignment requirement
- **Overall structure placement**
 - Each structure has alignment requirement K
 - K = Largest alignment of any element
 - Initial address & structure length must be multiples of K
- **Example (under Windows or x86-64):**
 - K = 8, due to **double** element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

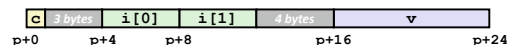


Carnegie Mellon

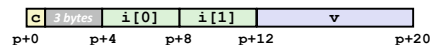
17

Different Alignment Conventions

- **x86-64 or IA32 Windows:**
 - K = 8, due to **double** element



- **IA32 Linux**
 - K = 4; **double** treated like a 4-byte data type



Carnegie Mellon

18

Carnegie Mellon

Saving Space

- Put large data types first

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

→

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

- Effect (example x86-64, both have K=8)

c3 bytes

i[0]

i[1]4 bytes

v

p+0p+4p+8p+16p+24

↓

v

i[0]

i[1]c

p+0p+8p+16

19

Carnegie Mellon

Arrays of Structures

- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

a[0]

a[1]

a[2]

...

a+0a+24a+48a+36

↙ ↘

v

i[0]

i[1]c

7 bytes

a+24a+32a+40a+48

20

Carnegie Mellon

Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
 - Resolved during linking

a[0]

...

a[i]

...

a+0a+12ia+12i+8

i2 bytes

v

j2 bytes

a+12ia+12i+8

```
short get_j(int idx)
{
  return a[idx].j;
}
```

%eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax

21

Carnegie Mellon

Today

- Structures
- Alignment
- Unions
- Floating point

22

Carnegie Mellon

Union Allocation

- Allocate according to largest element
- Can only use ones field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

c

i[0]

i[1]

v

up+0up+4up+8

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

c3 bytes

i[0]

i[1]4 bytes

v

sp+0sp+4sp+8sp+16sp+24

23

Carnegie Mellon

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```

u

f

04

```
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as (float) u ?

```
unsigned float2bit(float f) {
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as (unsigned) f ?

24

Byte Ordering Revisited

- **Idea**
 - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
 - Which is most (least) significant?
 - Can cause problems when exchanging binary data between machines
- **Big Endian**
 - Most significant byte has lowest address
 - PowerPC, Sparc
- **Little Endian**
 - Least significant byte has lowest address
 - Intel x86

25

Byte Ordering Example

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]	s[1]	s[2]	s[3]				
i[0]		i[1]					
l[0]							

26

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
dw.c[0], dw.c[1], dw.c[2], dw.c[3],
dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

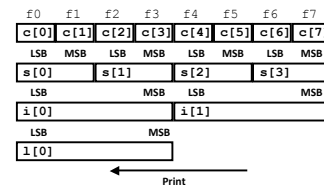
printf("Ints 0-1 == [0x%x,0x%x]\n",
dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
dw.l[0]);
```

27

Byte Ordering on IA32

Little Endian



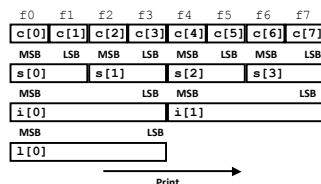
Output on IA32:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts      0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints        0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long        0 == [0xf3f2f1f0]
```

28

Byte Ordering on Sun

Big Endian



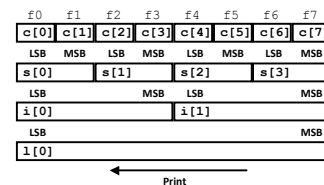
Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts      0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints        0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long        0 == [0xf0f1f2f3]
```

29

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts      0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints        0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long        0 == [0xf7f6f5f4f3f2f1f0]
```

30

Summary

- **Arrays in C**
 - Contiguous allocation of memory
 - Aligned to satisfy every element's alignment requirement
 - Pointer to first element
 - No bounds checking
- **Structures**
 - Allocate bytes in order declared
 - Pad in middle and at end to satisfy alignment
- **Unions**
 - Overlay declarations
 - Way to circumvent type system

31

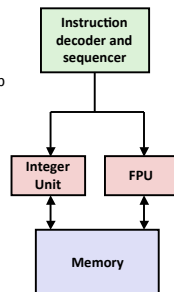
Today

- Structures
- Alignment
- Unions
- **Floating point**
 - x87 (available with IA32, becoming obsolete)
 - SSE3 (available with x86-64)

32

IA32 Floating Point (x87)

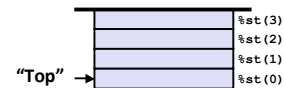
- **History**
 - 8086: first computer to implement IEEE FP
 - separate 8087 FPU (floating point unit)
 - 486: merged FPU and Integer Unit onto one chip
 - Becoming obsolete with x86-64
- **Summary**
 - Hardware to add, multiply, and divide
 - Floating point data registers
 - Various control & status registers
- **Floating Point Formats**
 - single precision (C `float`): 32 bits
 - double precision (C `double`): 64 bits
 - extended precision (C `long double`): 80 bits



33

FPU Data Register Stack (x87)

- **FPU register format (80 bit extended precision)**
- **FPU registers**
 - 8 registers %st(0) - %st(7)
 - Logically form stack
 - Top: %st(0)
 - Bottom disappears (drops out) after too many pushes



34

FPU instructions (x87)

- **Large number of floating point instructions and formats**
 - ~50 basic instruction types
 - load, store, add, multiply
 - sin, cos, tan, arctan, and log
 - Often slower than math lib

Sample instructions:

Instruction	Effect	Description
<code>fldz</code>	<code>push 0.0</code>	Load zero
<code>flds Addr</code>	<code>push Mem[Addr]</code>	Load single precision real
<code>fmulb Addr</code>	<code>%st(0) ← %st(0) * M[Addr]</code>	Multiply
<code>faddp</code>	<code>%st(1) ← %st(0) + %st(1); pop</code>	Add and pop

35

FP Code Example (x87)

- **Compute inner product of two vectors**
 - Single precision arithmetic
 - Common computation

```
float ipf (float x[],
          float y[],
          int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```
pushl %ebp                # setup
movl %esp, %ebp
pushl %ebx

movl 8(%ebp), %ebx        # %ebx = x
movl 12(%ebp), %ecx        # %ecx = y
movl 16(%ebp), %edx        # %edx = n
fldz                     # push +0.0
xorl %eax, %eax           # i = 0
cmpl %edx, %eax           # if i >= n done
jge .L3

.L5:
flds (%ebx, %eax, 4)       # push x[i]
fmuls (%ecx, %eax, 4)      # st(0) = y[i]
faddp                     # st(1) += st(0); pop
incl %eax                 # i++
cmpl %edx, %eax           # if i < n repeat
jlt .L5
.L3:
movl -4(%ebp), %ebx        # finish
movl %ebp, %esp
popl %ebp
ret                       # st(0) = result
```

36

Inner Product Stack Trace

Initialization

1. fldz

0.0

%st(0)

Iteration 0

2. flds (%ebx,%eax,4)

0.0

%st(1)

x[0]

%st(0)

3. fmulps (%ecx,%eax,4)

0.0

%st(1)

x[0]*y[0]

%st(0)

4. faddps

0.0+x[0]*y[0]

%st(0)

Iteration 1

5. flds (%ebx,%eax,4)

x[0]*y[0]

%st(1)

x[1]

%st(0)

6. fmulps (%ecx,%eax,4)

x[0]*y[0]

%st(1)

x[1]*y[1]

%st(0)

7. faddps

x[0]*y[0]+x[1]*y[1]

%st(0)

eax = i

ebx = *x

ecx = *y

37

Today

■ Structures

■ Alignment

■ Unions

■ Floating point

- x87 (available with IA32, becoming obsolete)
- SSE3 (available with x86-64)

38

Vector Instructions: SSE Family

■ SIMD (single-instruction, multiple data) vector instructions

- New data types, registers, operations
- Parallel operation on small (length 2-8) vectors of integers or floats
- Example:

+

x

“4-way”

■ Floating point vector instructions

- Available with Intel’s SSE (streaming SIMD extensions) family
- SSE starting with Pentium III: 4-way single precision
- SSE2 starting with Pentium 4: 2-way double precision
- All x86-64 have SSE3 (superset of SSE2, SSE)

39

Intel Architectures (Focus Floating Point)

Processors

8086

286

386

486

Pentium

Pentium MMX

Pentium III

Pentium 4

Pentium 4E

Pentium 4F

Core 2 Duo

Architectures

x86-16

x86-32

MMX

SSE

SSE2

SSE3

x86-64 / em64t

SSE4

Features

4-way single precision fp

2-way double precision fp

Our focus: SSE3 used for scalar (non-vector) floating point

time

40

SSE3 Registers

■ All caller saved

■ %xmm0 for floating point return value

128 bit = 2 doubles = 4 singles

%xmm0

Argument #1

%xmm1

Argument #2

%xmm2

Argument #3

%xmm3

Argument #4

%xmm4

Argument #5

%xmm5

Argument #6

%xmm6

Argument #7

%xmm7

Argument #8

%xmm8

%xmm9

%xmm10

%xmm11

%xmm12

%xmm13

%xmm14

%xmm15

41

SSE3 Registers

■ Different data types and associated instructions

■ Integer vectors:

128 bit

LSB

■ Floating point vectors:

■ Floating point scalars:

42

SSE3 Instructions: Examples

- Single precision 4-way vector add: `addps %xmm0, %xmm1`

- Single precision scalar add: `addss %xmm0, %xmm1`

43

SSE3 Instruction Names

44

SSE3 Basic Instructions

- Moves

Single	Double	Effect
<code>movss</code>	<code>movsd</code>	$D \leftarrow S$

 - Usual operand form: `reg → reg`, `reg → mem`, `mem → reg`
- Arithmetic

Single	Double	Effect
<code>addss</code>	<code>addsd</code>	$D \leftarrow D + S$
<code>subss</code>	<code>subsd</code>	$D \leftarrow D - S$
<code>mulss</code>	<code>mulsd</code>	$D \leftarrow D \times S$
<code>divss</code>	<code>divsd</code>	$D \leftarrow D / S$
<code>maxss</code>	<code>maxsd</code>	$D \leftarrow \max(D, S)$
<code>minss</code>	<code>minsd</code>	$D \leftarrow \min(D, S)$
<code>sqrtps</code>	<code>sqrtsd</code>	$D \leftarrow \sqrt{S}$

45

x86-64 FP Code Example

```
float ipf (float x[],
           float y[],
           int n) {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```
ipf:
    xorps    %xmm1, %xmm1
    xorl     %ecx, %ecx
    jmp      .L8
.L10:
    movslq   %ecx, %rax
    incl     %ecx
    movss    (%rsi,%rax,4), %xmm0
    mulss    (%rdi,%rax,4), %xmm0
    addss    %xmm0, %xmm1
.L8:
    cmpl     %edx, %ecx
    jl       .L10
    movaps   %xmm1, %xmm0
    ret
```

Will disappear Blackboard?

46

x86-64 FP Code Example

```
float ipf (float x[],
           float y[],
           int n) {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```
ipf:
    xorps    %xmm1, %xmm1    # result = 0.0
    xorl     %ecx, %ecx      # i = 0
    jmp      .L8             # goto middle
.L10:
    movslq   %ecx, %rax      # loop:
    incl     %ecx            # icpy = i
    movss    (%rsi,%rax,4), %xmm0 # t = y[icpy]
    mulss    (%rdi,%rax,4), %xmm0 # t *= x[icpy]
    addss    %xmm0, %xmm1    # result += t
.L8:
    cmpl     %edx, %ecx      # middle:
    jl       .L10            # i:n
    movaps   %xmm1, %xmm0    # if < goto loop
    ret
```

47

SSE3 Conversion Instructions

- Conversions
 - Same operand forms as moves

Instruction	Description
<code>cvtss2sd</code>	single → double
<code>cvttd2ss</code>	double → single
<code>cvtss2si</code>	int → single
<code>cvtss2sd</code>	int → double
<code>cvtss2ssq</code>	quad int → single
<code>cvtss2sdq</code>	quad int → double
<code>cvttd2si</code>	single → int (truncation)
<code>cvttd2si</code>	double → int (truncation)
<code>cvttd2siq</code>	single → quad int (truncation)
<code>cvttd2siq</code>	double → quad int (truncation)

48

Carnegie Mellon

x86-64 FP Code Example

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

a %xmm0 double
x %xmm1 float
b %xmm2 double
i %edi int

```

funct:
    cvtss2sd %xmm1, %xmm1
    mulsd   %xmm0, %xmm1
    cvtsi2sd %edi, %xmm0
    divsd   %xmm0, %xmm2
    movsd   %xmm1, %xmm0
    subsd   %xmm2, %xmm0
    ret
  
```

Will disappear
Blackboard?

49

Carnegie Mellon

x86-64 FP Code Example

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

a %xmm0 double
x %xmm1 float
b %xmm2 double
i %edi int

```

funct:
    cvtss2sd %xmm1, %xmm1    # %xmm1 = (double) x
    mulsd   %xmm0, %xmm1    # %xmm1 = a*x
    cvtsi2sd %edi, %xmm0    # %xmm0 = (double) i
    divsd   %xmm0, %xmm2    # %xmm2 = b/i
    movsd   %xmm1, %xmm0    # %xmm0 = a*x
    subsd   %xmm2, %xmm0    # return a*x - b/i
    ret
  
```

50

Carnegie Mellon

Constants

```
double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}
```

- Here: Constants in decimal format
 - compiler decision
 - hex more readable

Constant declarations

```

.LC2:
    .long 3435973837    # Low order four bytes of 1.8
    .long 1073532108    # High order four bytes of 1.8
.LC4:
    .long 0             # Low order four bytes of 32.0
    .long 1077936128    # High order four bytes of 32.0
  
```

Code

```

cel2fahr:
    mulsd .LC2(%rip), %xmm0 # Multiply by 1.8
    addsd .LC4(%rip), %xmm0 # Add 32.0
    ret
  
```

51

Carnegie Mellon

Checking Constant

- Previous slide: Claim


```

.LC4:
    .long 0             # Low order four bytes of 32.0
    .long 1077936128    # High order four bytes of 32.0
      
```
- Convert to hex format:


```

.LC4:
    .long 0x0           # Low order four bytes of 32.0
    .long 0x40400000    # High order four bytes of 32.0
      
```
- Convert to double (blackboard?):
 - Remember: e = 11 exponent bits, bias = $2^{e-1} = 1023$

52

Carnegie Mellon

Comments

- SSE3 floating point
 - Uses lower ½ (double) or ¼ (single) of vector
 - Finally departure from awkward x87
 - Assembly very similar to integer code
- x87 still supported
 - Even mixing with SSE3 possible
 - Not recommended
- For highest floating point performance
 - Vectorization a must (but not in this course☺)
 - See next slide

53

Carnegie Mellon

Vector Instructions

- Starting with version 4.1.1, gcc can autovectorize to some extent
 - -O3 or -ftree-vectorize
 - No speed-up guaranteed
 - Very limited
 - icc as of now much better
 - Fish machines: gcc 3.4
- For highest performance vectorize yourself using intrinsics
 - Intrinsics = C interface to vector instructions
 - Learn in 18-645
- Future
 - Intel AVX announced: 4-way double, 8-way single

54