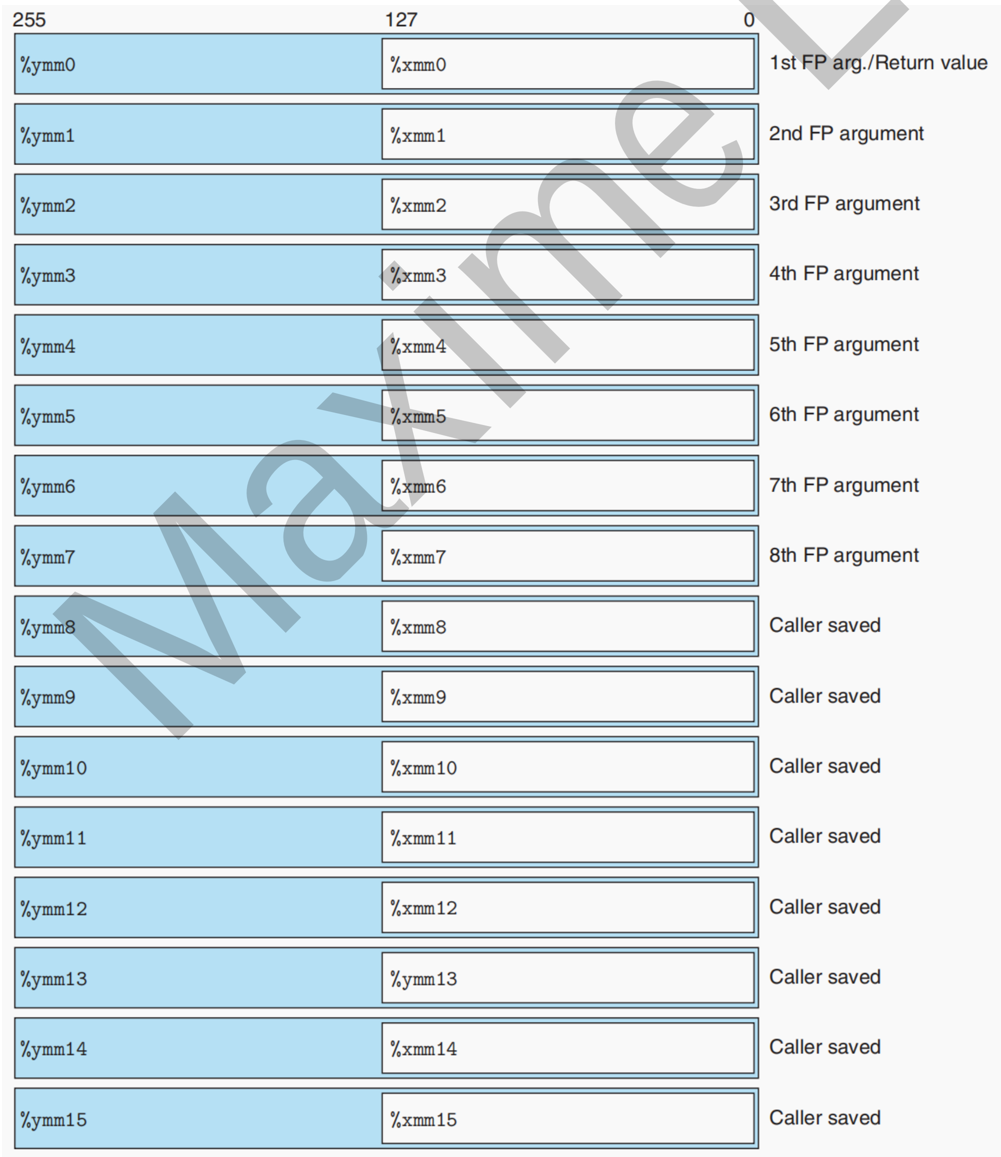
3_11 Floating-Point Code

- The **floating-point architecture** for a processor consists of:
  - How floating-point values are stored and accessed - some registers.
  - Instructions that operate on floating-point data.
  - The conventions used for passing floating-point values as arguments to functions and for returning them as results.
  - The conventions for how registers are preserved during function calls.

Float Architecture Revision	**Register Name**	**Register Size**
MMX - Multi Media Extensions	MM	64 bits
SSE - Streaming SIMD Extensions	XMM	128 bits
AVX - Advanced Vector Extensions (on book)	YMM	256 bits
AVX-512 - (Advanced Vector Extensions 512)	ZMM	512 bits

- Gcc will generate AVX2 code when given the command-line parameter `-mavx2`.
- The scalar AVX instructions (标量AVX指令) intend for operating on entire data vectors arise.
  - Scalar data - a **single value or element**, as opposed to a collection of values or elements (such as arrays, vectors, or matrices). Scalar data types represent individual quantities and are the simplest form of data in programming and computer science.



Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

- AVX floating-point architecture allows data to be stored in 16 YMM registers, named `%ymm0-%ymm15`.
  - Each YMM register is 256 bits (32 bytes) long.
- Dealing with scalar data:
  - When operating on scalar data, these registers only hold floating-point data, and only the low-order 32 bits (for float) or 64 bits (for double) are used.
  - The assembly code refers to the registers by their SSE XMM register names %xmm0–%xmm15, where each XMM register is the low-order 128 bits (16 bytes) of the corresponding YMM register.

3.11.1 Floating-Point Movement and Conversion Operations

Floating-point Movement Instructions

Instruction	Source	Destination	Description
vmovss	M_{32}	X	Move single precision
vmovss	X	M_{32}	Move single precision
vmovsd	M_{64}	X	Move double precision
vmovsd	X	M_{64}	Move double precision
vmovaps	X	X	Move aligned, packed single precision
vmovapd	X	X	Move aligned, packed double precision

- These operations above transfer values between memory and XMM registers, as well as between pairs of XMM registers.

vmovss and vmovsd

- Memory and XMM registers data movement - The first 4 instructions (`vmovss`, `vmovsd`) that reference memory above are **scalar instructions**, meaning that they operate on individual, rather than packed, data values.
  - The data are held either in memory (indicated in the table as M_{32} and M_{64}) or in XMM registers (shown in the table as X).
  - These instructions will work correctly regardless of the alignment of data, although the code optimization guidelines recommend that 32-bit memory data satisfy a 4-byte alignment and that 64-bit data satisfy an 8-byte alignment.
  - Memory references are specified in the same way as for the integer `mov` instructions, with all of the different possible combinations of displacement, base register, index register, and scaling factor.

vmovaps and vmovapd

- `vmovaps` for single precision and `vmovapd` for double-precision values.
- When used for XMM registers and XMM registers data movements:
  - For these cases, whether the program copies the entire register or just the low-order value affects neither the program functionality nor the execution speed, and so using these instructions rather than ones specific to scalar data makes no real difference.
- When used for XMM registers and memory data movements:
  - The letter 'a' in these instruction names stands for "aligned".
  - When used to read and write memory, they will cause an exception if the address does not satisfy a 16-byte alignment.

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

Example

```c
float float_mov(float v1, float *src, float *dst) {
        float v2 = *src;
        *dst = v1;
        return v2;
}
```

- Assembly code:

```
# float float_mov(float v1, float *src, float *dst)
# v1 in %xmm0, src in %rdi, dst in %rsi

float_mov:
        vmovaps %xmm0, %xmm1        # Copy v1
        vmovss (%rdi), %xmm0        # Read v2 from src
        vmovss %xmm1, (%rsi)        # Write v1 to dst
        ret                         # Return v2 in %xmm0
```

  - `vmovaps` - copy data from xmm0 to xmm1;
  - `vmovss` - copy data from memory (M(rdi)) to an xmm0 register and from an xmm1 register to memory (M(rsi)).

Converting between Floating/Double point and Integer

- Convert from a floating-point value read from either an XMM register or memory and write the result to a general-purpose register (e.g., %rax, %ebx, etc.):

Instruction	Source	Destination	Description
vcvttss2si	X/M_{32}	R_{32}	Convert with truncation single precision to integer
vcvttsd2si	X/M_{64}	R_{32}	Convert with truncation double precision to integer
vcvttss2siq	X/M_{32}	R_{64}	Convert with truncation single precision to quad word integer
vcvttsd2siq	X/M_{64}	R_{64}	Convert with truncation double precision to quad word integer

  - When converting floating-point values to integers, they perform truncation, rounding values toward zero.
- Convert from integer to floating point:

Instruction	Source 1	Source 2	Destination	Description
vcvtsi2ss	M_{32}/R_{32}	X	X	Convert integer to single precision
vcvtsi2sd	M_{32}/R_{32}	X	X	Convert integer to double precision
vcvtsi2ssq	M_{64}/R_{64}	X	X	Convert quad word integer to single precision
vcvtsi2sdq	M_{64}/R_{64}	X	X	Convert quad word integer to double precision

  - Three-operand format, with two sources and a destination.
  - These instructions above convert from the data type of the first source to the data type of the destination. The second source value has no effect on the low-order bytes of the result.
  - For our purposes, we can ignore the second operand, since its value only affects the upper bytes of the result.
  - The destination must be an XMM register.
  - In common usage, both the second source and the destination operands are identical.
  - Example:

```
vcvtsi2sdq %rax, %xmm1, %xmm1
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

- Reads a long integer from register `%rax`.
- Converts it to data type double.
- Stores the result in the lower bytes of XMM register `%xmm1`.

Converting from Single Precision to Double Precision

- Logically, we use `vcvtss2sd` - convert a single-precision value to a double-precision value:
  - suppose the low-order 4 bytes of %xmm0 hold a single-precision value

    ```
    vcvtss2sd %xmm0, %xmm0, %xmm0
    ```

    - convert a single-precision value in `%xmm0` to a double-precision value and store the result in the lower 8 bytes of register `%xmm0`.

- In reality, GCC generate code below:

```
# Conversion from single to double precision
vunpcklps %xmm0, %xmm0, %xmm0            # Replicate first vector element
vcvtps2pd %xmm0, %xmm0                   # Convert two vector elements to double
```

  - `vunpcklps` instruction is normally used to interleave the values in two XMM registers and store them in a third.
    - Example - if register `%src0` contains words `[s3, s2, s1, s0]` and the other register `src1` contains words `[d3, d2, d1, d0]`, then after `vunpcklps %src0, %src1, %dest`, the value of the destination register `dest` will be `[s1, d1, s0, d0]`.
    - So, if the original register `%xmm0` held values `[x3, x2, x1, x0]`, then after `vunpcklps %xmm0, %xmm0, %xmm0`, the instruction will update the register to hold values `[x1, x1, x0, x0]`.
      - Please notice that x0,x1,x2 and x3 are all single precision value.
  - `vcvtps2pd` instruction expands the two low-order single precision values in the source XMM register to be the two double-precision values in the destination XMM register.
    - If the original register `%xmm0` held values `[x3, x2, x1, x0]`, then after `vcvtps2pd %xmm0, %xmm0`, `%xmm0` will be `[dx0,dx0]`, while `dx0` is the result of converting `x` to double precision.
  - Therefore, suppose `%xmm0` held values `[x3, x2, x1, x0]`, the `%xmm0` value will update as below:

    ```
    # Conversion from single to double precision
    vunpcklps %xmm0, %xmm0, %xmm0            # xmm0 = [x1,x1,x0,x0]
    vcvtps2pd %xmm0, %xmm0                   # xmm0 = [dx0,dx0]
    ```

  - The single precision value `x0` has been converted to double precision value `dx0` successfully.

Converting from Double Precision to Single Precision

- Logically, we use `vcvtsd2ss` - convert a double-precision value to a single-precision value:

Maxime Lionel

```
vcvtsd2ss %xmm0, %xmm0, %xmm0
```

- In reality, GCC generate code below:

```
# Conversion from double to single precision
vmovddup %xmm0, %xmm0       # Replicate first vector element
vcvtpd2psx %xmm0, %xmm0     # Convert two vector elements to single
```

  - Suppose register `%xmm0` holding two double-precision values `[dx1, dx0]`.
  - `vmovddup %xmm0, %xmm0` – set `%xmm0` to `[dx0, dx0]`.
  - `vcvtpd2psx %xmm0, %xmm0` – convert to single precision, pack them into the low-order half of the register, and set the upper half to 0, yielding a result `[0.0, 0.0, sx0, sx0]`.

- Example:
  - C code:

```c
double fcvt(int i, float *fp, double *dp, long *lp)
{
        float f = *fp; double d = *dp; long l = *lp;
        *lp = (long) d;
        *fp = (float) i;
        *dp = (double) l;
        return (double) f;
}
```

  - Assembly code:

```
# double fcvt(int i, float *fp, double *dp, long *lp)
# i in %edi, fp in %rsi, dp in %rdx, lp in %rcx
fcvt:
        vmovss (%rsi), %xmm0       # xmm0=M(rsi):       Get f = *fp
        movq (%rcx), %rax          # rax=M(rcx):        Get l = *lp
        vcvttsd2siq (%rdx), %r8    # r8=(long)M(rdx):   Get d = *dp and convert to long
        movq %r8, (%rcx)           # M(rcx)=r8:         Store at lp
        vcvtsi2ss %edi, %xmm1, %xmm1   # xmm1=(float)edi:    Convert i to float
        vmovss %xmm1, (%rsi)       # M(rsi)=xmm1:       Store at fp
        vcvtsi2sdq %rax, %xmm1, %xmm1  # xmm1=(double)rax:   Convert l to double
        vmovsd %xmm1, (%rdx)       # M(rdx)=xmm1:       Store at dp

    # The following two instructions convert f to double
        vunpcklps %xmm0, %xmm0, %xmm0
        vcvtps2pd %xmm0, %xmm0
        ret                        # Return f in xmm0
```

Practice Problem 3.50

For the following C code, the expressions val1 – val4 all map to the program values i, f, d, and l:

```c
double fcvt2(int *ip, float *fp, double *dp, long l)
{
        int i = *ip; float f = *fp; double d = *dp;
        *ip = (int) val1;
        *fp = (float) val2;
        *dp = (double) val3;
        return (double) val4;
}
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

Determine the mapping, based on the following x86-64 code for the function:

```
1    # double fcvt2(int *ip, float *fp, double *dp, long l)
2    # ip in %rdi, fp in %rsi, dp in %rdx, l in %rcx
3    # Result returned in %xmm0
4
5    fcvt2:
6            movl (%rdi), %eax
7            vmovss (%rsi), %xmm0
8            vcvttsd2si (%rdx), %r8d
9            movl %r8d, (%rdi)
10           vcvtsi2ss %eax, %xmm1, %xmm1
11           vmovss %xmm1, (%rsi)
12           vcvtsi2sdq %rcx, %xmm1, %xmm1
13           vmovsd %xmm1, (%rdx)
14           vunpcklps %xmm0, %xmm0, %xmm0
15           vcvtps2pd %xmm0, %xmm0
16           ret
```

Solution:
Firstly, go though all asm instructions:

```
1    # double fcvt2(int *ip, float *fp, double *dp, long l)
2    # ip in %rdi, fp in %rsi, dp in %rdx, l in %rcx
3    # Result returned in %xmm0
4
5    fcvt2:
6            movl (%rdi), %eax                # eax=M(rdi):  eax = *ip
7            vmovss (%rsi), %xmm0             # xmm0=M(rsi): xmm0= *fp
8            vcvttsd2si (%rdx), %r8d          # r8d=M(rdx):  r8d = (int)(*dp)
9            movl %r8d, (%rdi)                # M(rdi)=r8d:  *ip = r8d
10           vcvtsi2ss %eax, %xmm1, %xmm1     # xmm1=(float)eax=(float)(*ip)
11           vmovss %xmm1, (%rsi)             # M(rsi)=xmm1: (*fp) = xmm1 = (float)(*ip)
12           vcvtsi2sdq %rcx, %xmm1, %xmm1    # xmm1=(double)rcx: xmm1=(double)l
13           vmovsd %xmm1, (%rdx)             # M(rdx)=xmm1: (*dp)=xmm1=(double)l
14           vunpcklps %xmm0, %xmm0, %xmm0    # convert float in xmm0 to double
15           vcvtps2pd %xmm0, %xmm0
16           ret
```

Secondly,

- val1
  From `*ip = (int) val1` , we need to find out `*ip`.
  From line 8 and 9, `*ip` is from `*dp` by converting to `int`.
  So `val1 = d`.
- val2
  From `*fp = (float) val2`, we need to find out `*fp`.
  From line 11, `*fp` is from `*ip` by converting to `float`.
  So `val2 = i`.
- val3
  From `*dp = (double) val3`, we need to find out `*dp`.
  From line 13, `*dp` is from `l` by converting to `double`.
  So `val3 = d`.
- val4
  The funtion returns `double` type, so the return value is stored in `xmm0`. From line 7, `xmm0` is from `f`.
  So `val4 = f`.

Practice Problem 3.51

Maxime Lionel

The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defined using typedef:

```
dest_t cvt(src_t x)
{
        dest_t y = (dest_t) x;
        return y;
}
```

For execution on x86-64, assume that argument x is either in %xmm0 or in the appropriately named portion of register `%rdi` (i.e., `%rdi` or `%edi`). One or two instructions are to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax` (integer result) or `%xmm0` (floating-point result). Show the instruction(s), including the source and destination registers.

T_x	T_y	**Instructions**
long	double	`vcvtsi2sdq %rdi, %xmm0`
double	int	
double	float	
long	float	
float	long	

Solution:

T_x	T_y	**Instructions**
long	double	`vcvtsi2sdq %rdi, %xmm0`
double	int	`vcvttsd2si %xmm0, %eax`

double	float	`vcvtsd2ss %xmm0, %xmm0, %xmm0`
		In gcc, it's like:
		`vmovddup %xmm0, %xmm0` `vcvtpd2psx %xmm0, %xmm0`

long	float	`vcvtsi2ssq %rdi, %xmm0, %xmm0`
float	long	`vcvttss2siq %xmm0, %rax`

Maxime Lionel

3.11.2 Floating-Point Code in Procedures

- With x86-64, the XMM registers are used for passing floating-point arguments to functions and for returning floating-point values from them.
- The following conventions are observed:
  - Up to eight floating-point arguments can be passed in XMM registers `%xmm0-%xmm7`.
  - Additional floating-point arguments can be passed on the **stack**.
  - A function that returns a floating-point value does so in register `%xmm0`.
  - All XMM registers are caller saved.
  - When a function contains a combination of pointer, integer, and floating point arguments, the pointers and integers are passed in general-purpose registers, while the floating-point values are passed in XMM registers.
- Examples:

  ```
  double f1(int x, double y, long z);
  ```

  - x in `%edi`, y in `%xmm0`, and z in `%rsi`.

  ```
  double f2(double y, int x, long z);
  ```

  - y in `%xmm0`, x in `%edi`, and z in `%rsi`. Same as above.

Copyright 2024 Maxime Lionel. All rights reserved.

```c
double f1(float x, double *y, long *z);
```

- `x` in `%xmm0`, `y` in `%rdi`, and `z` in `%rsi`.

Practice Problem 3.52

For each of the following function declarations, determine the register assignments for the arguments:

A. double g1(double a, long b, float c, int d);

B. double g2(int a, double *b, float *c, long d);

C. double g3(double *a, double b, int c, float d);

D. double g4(float a, int *b, float c, double d);

Solution:

A.

```
double g1(double a, long b, float c, int d);
a - %xmm0
b - %rdi
c - %xmm1
d - %esi
```

B.

```
double g2(int a, double *b, float *c, long d);
a - %edi
b - %rsi
c - %rcx
d - %rdx
```

C.

```
double g3(double *a, double b, int c, float d);
a - %rdi
b - %xmm0
c - %esi
d - %xmm1
```

D.

```
double g4(float a, int *b, float c, double d);
a - %xmm0
b - %rdi
c - %xmm1
d - %xmm2
```

3.11.3 Floating-Point Arithmetic Operations

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

- A set of scalar AVX2 floating-point instructions that perform arithmetic operations:

Single	Double	Effect	Description
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	Floating-point add
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	Floating-point subtract
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	Floating-point multiply
vdivss	vdivsd	$D \leftarrow S_2/S_1$	Floating-point divide
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S_1}$	Floating-point square root

  - S_1 can be either an XMM register or a memory location.
  - S_2 and D must be XMM registers.
  - Each operation has an instruction for single precision and an instruction for double precision.
  - The result is stored in the destination register.
  - Syntax: `vsubsd S1, S2, D` means `D = S2 - S1`
- Example:

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

  - Assembly code on book:

```
    double funct(double a, float x, double b, int i)
    a in %xmm0, x in %xmm1, b in %xmm2, i in %edi
1   funct:
      The following two instructions convert x to double
2     vunpcklps       %xmm1, %xmm1, %xmm1
3     vcvtps2pd       %xmm1, %xmm1
4     vmulsd  %xmm0, %xmm1, %xmm0               Multiply a by x
5     vcvtsi2sd       %edi, %xmm1, %xmm1        Convert i to double
6     vdivsd  %xmm1, %xmm2, %xmm2               Compute b/i
7     vsubsd  %xmm2, %xmm0, %xmm0               Subtract from a*x
8     ret                                       Return
```

    - a, x, and b are passed in XMM registers `%xmm0-%xmm2`, while i is passed in register `%edi`.
    - lines 2–3: standard two-instruction sequence is used to convert argument x to double.
    - line 5: convert argument i to double.
    - Return in register `%xmm0`.
  - In real life, it's like:
    - AXV - Compile in avx code with `-mavx` option: `gcc -mavx -Og -fno-stack-protector -S funct.c -o funct_avx.s`

```
vcvtss2sd       %xmm1, %xmm1, %xmm1       # convert xmm1 from float to double
vmulsd  %xmm0, %xmm1, %xmm1               # xmm1=xmm0*xmm1: xmm1=a*x
vxorps  %xmm0, %xmm0, %xmm0               # clear xmm0
vcvtsi2sdl      %edi, %xmm0, %xmm0        # convert edi from int to double, store it in xmm0
vdivsd  %xmm0, %xmm2, %xmm2               # xmm2=xmm2/xmm0: xmm2=b/(double)i
vsubsd  %xmm2, %xmm1, %xmm0               # xmm0=xmm1-xmm2: xmm0=a*x-b/i
ret
```

    - SSE2 - Compile in sse2 code with `-mavx` option: `gcc -msse2 -Og -fno-stack-protector -S funct.c -o funct_sse2.s`

```
cvtss2sd        %xmm1, %xmm1        # xmm1=(double)x
mulsd   %xmm0, %xmm1                # xmm1=a*x
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

```
pxor    %xmm0, %xmm0                # clear xmm0
cvtsi2sdl       %edi, %xmm0         # xmm0=(double)i
divsd   %xmm0, %xmm2                # xmm2=b/(double)i
subsd   %xmm2, %xmm1                # xmm1=a*x-b/i
movapd  %xmm1, %xmm0                # xmm0=a*x-b/i
ret
```

Practice Problem 3.53

For the following C function, the types of the four arguments are defined by typedef:

```c
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
{
        return p/(q+r) - s;
}
```

When compiled, gcc generates the following code:

```
# double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)

funct1:
        vcvtsi2ssq      %rsi, %xmm2, %xmm2
        vaddss          %xmm0, %xmm2, %xmm0
        vcvtsi2ss       %edi, %xmm2, %xmm2
        vdivss          %xmm0, %xmm2, %xmm0
        vunpcklps       %xmm0, %xmm0, %xmm0
        vcvtps2pd       %xmm0, %xmm0
        vsubsd          %xmm1, %xmm0, %xmm0
        ret
```

Determine the possible combinations of types of the four arguments (there may be more than one).

Solution:

```
# double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)

funct1:
        vcvtsi2ssq      %rsi, %xmm2, %xmm2   # xmm2=(float)rsi: rsi - type long
        vaddss          %xmm0, %xmm2, %xmm0  # xmm0=xmm0+xmm2=xmm0+(float)rsi refer to (q+r)
                                             # so q and r can be xmm0 or rsi while type is float or long
        vcvtsi2ss       %edi, %xmm2, %xmm2   # xmm2=(float)edi:
                                             # p can only be edi which type is int
        vdivss          %xmm0, %xmm2, %xmm0  # xmm0=xmm2/xmm0=((float)edi)/xmm0: xmm0=(float)p/(q+r)
        vunpcklps       %xmm0, %xmm0, %xmm0
        vcvtps2pd       %xmm0, %xmm0         # xmm0 = (double)xmm0
        vsubsd          %xmm1, %xmm0, %xmm0  # xmm0=xmm0-xmm1
                                             # s = xmm1 - type double
        ret
```

Therefore, there's 2 possibilities:

```c
double funct1(int p, float q, long r, double s);
or
double funct1(int p, long q, float r, double s);
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

Practice Problem 3.54

Function funct2 has the following prototype:

```c
double funct2(double w, int x, float y, long z);
```

Gcc generates the following code for the function:

```
# double funct2(double w, int x, float y, long z)
# w in %xmm0, x in %edi, y in %xmm1, z in %rsi

funct2:
        vcvtsi2ss    %edi, %xmm2, %xmm2
        vmulss       %xmm1, %xmm2, %xmm1
        vunpcklps    %xmm1, %xmm1, %xmm1
        vcvtps2pd    %xmm1, %xmm2
        vcvtsi2sdq   %rsi, %xmm1, %xmm1
        vdivsd       %xmm1, %xmm0, %xmm0
        vsubsd       %xmm0, %xmm2, %xmm0
        ret
```

Write a C version of funct2.

Solution:
Firstly, analyze assembly code:

```
# double funct2(double w, int x, float y, long z)
# w in %xmm0, x in %edi, y in %xmm1, z in %rsi

funct2:
        vcvtsi2ss    %edi, %xmm2, %xmm2        # xmm2=(float)edi: xmm2 = (float)x
        vmulss       %xmm1, %xmm2, %xmm1       # xmm1=xmm1*xmm2: xmm1 = y*(float)x
        vunpcklps    %xmm1, %xmm1, %xmm1
        vcvtps2pd    %xmm1, %xmm2              # xmm2=(double)xmm1: xmm2 = (double)(y*(float)x)
        vcvtsi2sdq   %rsi, %xmm1, %xmm1        # xmm1=(double)rsi: xmm1 = (double)z
        vdivsd       %xmm1, %xmm0, %xmm0       # xmm0=xmm0/xmm1: xmm0 = w/(double)z
        vsubsd       %xmm0, %xmm2, %xmm0       # xmm0=xmm2-xmm0: xmm0 = (double)(y*(float)x) - w/(double)z
        ret
```

Secondly, easily get the function code below:

```c
double funct2(double w, int x, float y, long z)
{
        return y*x - w/z;
}
```

Maxime Lionel

3.11.4 Defining and Using Floating-Point Constants

- Unlike integer arithmetic operations, AVX floating-point operations **cannot have immediate values as operands.**
- For AVX floating-point operations, the compiler must allocate and initialize storage for any constant values. The code then reads the values from memory.
- Example:
  - C code:

```c
double cel2fahr(double temp)
```

Copyright 2024 Maxime Lionel. All rights reserved.

```c
{
        return 1.8 * temp + 32.0;
}
```

- Assembly Code:

```
# double cel2fahr(double temp)
# temp in %xmm0

cel2fahr:
        vmulsd .LC2(%rip), %xmm0, %xmm0     # Multiply by 1.8
        vaddsd .LC3(%rip), %xmm0, %xmm0     # Add 32.0
        ret

.LC2:
        .long 3435973837        # Low-order 4 bytes of 1.8
        .long 1073532108        # High-order 4 bytes of 1.8
.LC3:
        .long 0                 # Low-order 4 bytes of 32.0
        .long 1077936128        # High-order 4 bytes of 32.0
```

- Firstly, review the representation of double precision floating.
    - Easy to find that `1.8` and `32.0` are normalized form of double precision floating-point.
    - The equation is like:
      $V = (-1)^s \times (1 + 0.f_{51}...f_1 f_0) \times 2^{[e_{10}...e_1 e_0]-(2^{11-1}-1)} = (-1)^s \times 1.f_{51}...f_1 f_0 \times 2^{[e_{10}...e_1 e_0]-1023_{10}}$
      (e != [0...0] or [1...1])
    - If $V = 1.8$, $V = (-1)^0 \times 1.f_{51}...f_1 f_0 \times 2^{[e_{10}...e_1 e_0]-1023_{10}}$
        - If $1.f_{51}...f_1 f_0$ = 1.8, then f = 0b 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101 = 0x cccccccccccccd
        - If $E = [e_{10}...e_1 e_0] - 1023_{10} = 0$, then e = 1023 = 0b 011 1111 1111 = 0x 3ff
        - s = 0
        - So, the full representation
          = 0b 0011 1111 1111 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101
          = 0x 3ffc cccc cccc cccd
    - If $V = 32.0$, $V = (-1)^0 \times 1.f_{51}...f_1 f_0 \times 2^{[e_{10}...e_1 e_0]-1023_{10}}$
        - If $1.f_{51}...f_1 f_0$ = 1.0, then f = 0b 0.
        - If $E = [e_{10}...e_1 e_0] - 1023_{10} = 5$, then e = 1028 = 0b 100 0000 0100 = 0x 404
        - s = 0
        - So, the full representation
          = 0b 0100 0000 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
          = 0x 4040 0000 0000 0000
- The machine uses little-endian byte ordering, the first value gives the low-order 4 bytes, while the second gives the high-order 4 bytes.

Practice Problem 3.55

Show how the numbers declared at label .LC3 encode the number 32.0.

Solution: already showed above.

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

3.11.5 Using Bitwise Operations in Floating-Point Code

- Bitwise operations on packed data:

Single	Double	Effect	Description
vxorps	xorpd	$D \leftarrow S_2 \text{ ^ } S_1$	Bitwise EXCLUSIVE-OR
vandps	andpd	$D \leftarrow S_2 \,\&\, S_1$	Bitwise AND

  - These instructions perform Boolean operations on all 128 bits in an XMM register.
  - These operations all act on packed data, meaning that they update the entire destination XMM register, applying the bitwise operation to all the data in the two source registers.

Practice Problem 3.56

Consider the following C function, where EXPR is a macro defined with `#define`:

```c
double simplefun(double x) {
        return EXPR(x);
}
```

Below, we show the AVX2 code generated for different definitions of EXPR, where value x is held in `%xmm0`. All of them correspond to some useful operation on floating-point values. Identify what the operations are. Your answers will require you to understand the bit patterns of the constant words being retrieved from memory.

A.

```
        vmovsd .LC1(%rip), %xmm1
        vandpd %xmm1, %xmm0, %xmm0
.LC1:
        .long 4294967295
        .long 2147483647
        .long 0
        .long 0
```

B.

```
        vxorpd %xmm0, %xmm0, %xmm0
```

C.

```
        vmovsd .LC2(%rip), %xmm1
        vxorpd %xmm1, %xmm0, %xmm0
.LC2:
        .long 0
        .long -2147483648
        .long 0
        .long 0
```

Solution:

A.

Firstly, convert to hex representation:

2147483647 = 0x 7FFF FFFF

4294967295 = 0x FFFF FFFF

Secondly, analyze the assembly code:

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

```
        vmovsd .LC1(%rip), %xmm1         # xmm1 = 0x 7FFF FFFF FFFF FFFF
        vandpd %xmm1, %xmm0, %xmm0       # xmm0=xmm0&xmm1: xmm0 = x & 0x 7FFF FFFF FFFF FFFF
.LC1:
        .long 4294967295
        .long 2147483647
        .long 0
        .long 0
```

Thirdly, we find that `xmm0 = x & 0x 7FFF FFFF FFFF FFFF` is to clear the sign bit to get the absolute value.
So the result is like:

```
#include <math.h>
# define EXPR(x) fabs(x)
```

B.

```
        vxorpd %xmm0, %xmm0, %xmm0      # xmm0 = xmm0^xmm0 is to clear xmm0
```

So the result:

```
# define EXPR(x) 0.0
```

C.
Firstly, convert to hex representation:
-2147483648 = 0x FFFF FFFF 8000 0000
Secondly, analyze the assembly code:

```
        vmovsd .LC2(%rip), %xmm1       # xmm1 = 0x 8000 0000 0000 0000
        vxorpd %xmm1, %xmm0, %xmm0     # xmm0 = x ^ 0x 8000 0000 0000 0000
.LC2:
        .long 0
        .long -2147483648
        .long 0
        .long 0
```

Thirdly, we find that `xmm0 = x ^ 0x 8000 0000 0000 0000` is to simply change the sign bit.
So, the result is like:

```
# define EXPR(x) -x
```

3.11.6 Floating-Point Comparison Operations

- AVX2 provides two instructions for comparing floating-point values:

Instruction	Based on	Description
`ucomiss` S_1, S_2	S_2-S_1	Compare single precision
`ucomisd` S_1, S_2	S_2-S_1	Compare double precision

  - As with `cmpq`, they follow the ATT-format convention of listing the operands in reverse order.
  - S_2 must be in an XMM register, while S_1 can be either in an XMM register or in memory.

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

- The floating-point comparison instructions set three condition codes: the zero flag ZF, the carry flag CF, and the parity flag PF.

Ordering S_2:S_1	CF	ZF	PF
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

  - PF flag:
    - For integer operations, PF flag is set when the most recent arithmetic or logical operation yielded a value where the least significant byte has even parity (i.e., an even number of ones in the byte).
    - For floating-point comparisons, however, the flag is set when either operand is `NaN`.
      - For example, even the comparison x == x yields 0 when x is `NaN`.
      - The unordered case occurs when either operand is NaN. This can be detected with the parity flag.
        - Example: the jp (for "jump on parity") instruction is used to conditionally jump when a floating-point comparison yields an unordered result.
  - ZF is set when the two operands are equal.
  - CF is set when S2 < S1.
    - Instructions such as ja and jb are used to conditionally jump on various combinations of these flags.

Example:

- C code:

```c
typedef enum {NEG, ZERO, POS, OTHER} range_t; // 0 (NEG), 1 (ZERO), 2 (POS), and 3 (OTHER)

range_t find_range(float x)
{
        int result;
        if (x < 0)
                result = NEG;
        else if (x == 0)
                result = ZERO;
        else if (x > 0)
                result = POS;
        else
                result = OTHER;
        return result;
}
```

- Assembly code:

```
# range_t find_range(float x)
# x in %xmm0

find_range:
        vxorps %xmm1, %xmm1, %xmm1          # Set %xmm1 = 0
        vucomiss %xmm0, %xmm1               # Compare 0:x
        ja .L5                              # If >, goto neg
        vucomiss %xmm1, %xmm0               # Compare x:0
        jp .L8                              # If NaN, goto posornan
        movl $1, %eax                       # result = ZERO
        je .L3                              # If =, goto done

.L8:                                    # posornan:
        vucomiss .LC0(%rip), %xmm0          # Compare x:0
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

```
15          setbe %al                         # Set result = NaN?1:0
16          movzbl %al, %eax                  # Zero-extend
17          addl $2, %eax                     # result += 2 (POS for > 0, OTHER for NaN)
18          ret                               # Return
19
20  .L5:                                  # neg:
21          movl $0, %eax                     # result = NEG
22
23  .L3:                                  # done:
24          rep; ret                          # Return
```

- 4 possible comparison results:
  - x < 0.0 The `ja` branch on line will be taken, jumping to the end with a return value of 0.
  - x = 0.0 The `ja` (line 7) and `jp` branch (line 9) will not be taken, but the `je` will, returning with %eax equal to 1.
  - x > 0.0 None of the three branches will be taken. The setbe (line 15) will yield 0, and this will be incremented by the addl instruction (line 17) to give a return value of 2.
  - x = NaN The jp branch (line 9) will be taken. The third vucomiss (line 14) will set both the carry and the zero flag, and so the instruction setbe instruction (line 15) and the following instruction will set %eax to 1. This gets incremented by the addl instruction (line 17) to give a return value of 3.

Practice Problem 3.57

Function funct3 has the following prototype:

```c
double funct3(int *ap, double b, long c, float *dp);
```

For this function, gcc generates the following code:

```
# double funct3(int *ap, double b, long c, float *dp)
# ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx

funct3:
        vmovss        (%rdx), %xmm1
        vcvtsi2sd     (%rdi), %xmm2, %xmm2
        vucomisd      %xmm2, %xmm0
        jbe           .L8
        vcvtsi2ssq    %rsi, %xmm0, %xmm0
        vmulss        %xmm1, %xmm0, %xmm1
        vunpcklps     %xmm1, %xmm1, %xmm1
        vcvtps2pd     %xmm1, %xmm0
        ret

.L8:
        vaddss        %xmm1, %xmm1, %xmm1
        vcvtsi2ssq    %rsi, %xmm0, %xmm0
        vaddss        %xmm1, %xmm0, %xmm0
        vunpcklps     %xmm0, %xmm0, %xmm0
        vcvtps2pd     %xmm0, %xmm0
        ret
```

Write a C version of funct3.

Solution:

```
# double funct3(int *ap, double b, long c, float *dp)
# ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.

```
funct3:
        vmovss       (%rdx), %xmm1          # xmm1=M(rdx): xmm1 = (float)(*dp)
        vcvtsi2sd    (%rdi), %xmm2, %xmm2   # xmm2=(double)M(rdi): xmm2 = (double)(*ap)
        vucomisd     %xmm2, %xmm0           # cmp xmm0:xmm2: compare b:(double)(*ap)
        jbe          .L8                    # if xmm0<=xmm2: b<=(double)(*ap), goto .L8
        vcvtsi2ssq   %rsi, %xmm0, %xmm0     # xmm0=(float)rsi: xmm0 = (float)c
        vmulss       %xmm1, %xmm0, %xmm1    # xmm1=xmm0*xmm1: xmm1 = (float)c * (float)(*dp)
        vunpcklps    %xmm1, %xmm1, %xmm1
        vcvtps2pd    %xmm1, %xmm0           # xmm0=(double)xmm1: xmm0 = (double)((float)c * (float)(*dp))
        ret

.L8:
        vaddss       %xmm1, %xmm1, %xmm1    # xmm1=xmm1+xmm1: xmm1 = 2*(float)(*dp)
        vcvtsi2ssq   %rsi, %xmm0, %xmm0     # xmm0=(float)rsi: xmm0 = (float)c
        vaddss       %xmm1, %xmm0, %xmm0    # xmm0=xmm0+xmm1: xmm0 = (float)c + 2*(float)(*dp)
        vunpcklps    %xmm0, %xmm0, %xmm0
        vcvtps2pd    %xmm0, %xmm0           # xmm0 = (double)((float)c + 2*(float)(*dp))
        ret
```

Thus, we may easily get the c code below:

```c
double funct3(int *ap, double b, long c, float *dp)
{
        if(b <= (double)(*ap))
        {
                return (double)((float)c + 2*(*dp));
        }
        else
        {
                return (double)((float)c * (*dp));
        }
}
```

Maxime Lionel

Copyright 2024 Maxime Lionel. All rights reserved.