

Simple [Scalar] SSE

- Intel floating point assembly language programming has traditionally used the x87 floating point unit (FPU)
- With 80-bit representation (long double), x87 still yields maximum precision
- However, with the 64-bit ABI's use of SSE registers (xmm0–xmm7) for passing floating point parameters and return values, it becomes more convenient to do floating point using this instruction subset (“SSE” stands for “Streaming SIMD Extensions”)

Function Calls

- Per the 64-bit ABI, floating point parameters should be passed, in order, from xmm0 to xmm7
- Floating point return values go to xmm0
- When using printf with floating point values, note that the xmm registers are vector registers (though not used as such in these notes) — so set rax accordingly!
- The standard C math library uses these conventions (of course!), but is not automatically linked by default — append -lm when linking with gcc

Data Types

- SSE instructions work with either scalar (single) or vector (multiple) values — we focus on the scalar variety here (consult the Intel manuals, particularly Chapters 10–12, for details)
- Scalar values can be single precision (C type `float`, 32-bit IEEE 754; `dd/dword`) or double precision (C type `double`, 64-bit IEEE 754; `dq/qword`)
- Vector capability comes from the `xmm` registers' 128-bit size: good for 4 floats or 2 doubles in a single register

Instructions

- Data definition: NASM can parse floating point literals
- Data transfer: `mov` remains the all-purpose transfer instruction, appended with either `ss` (scalar single) or `sd` (scalar double)
- Arithmetic: Also similar to their integer cousins, just append `ss` or `sd` as well
- Conversion: You can also convert from floating point to integer and back — look up `cvtsi2ss`, `cvtsi2sd`, `cvtss2si`, and `cvtsd2si`

Comparisons

- Comparisons are trickier than for integers
- A new relationship, “unordered,” exists — essentially, any comparison with a NaN (otherwise, it is “ordered”)
- Two types of comparisons: `cmpsd` and `cmpss` take three operands — the third being the relationship being tested — and deposit the result in an `xmm` register
- `comiss/comisd` and `ucomiss/ucomisd` work more like “regular” `cmp` in that they set `eflags` — they set `eflags` slightly differently, so double-check the manual

Scalar SSE Examples

- The following examples show these instructions in action, including calls to `printf` using floating point parameters and calls to the C math library (`-lm`)
- First off, simple [hardcoded] floating point addition:

```
global main
extern printf

section .text

format: db    "%f", 10, 0
term1:  dq    4.230642    ; NASM encodes this for you --- note the data size
term2:  dq    38.1936
main:   push  rbp          ; Save rbp
        movsd xmm0, [term1] ; Load up the first term
        addsd xmm0, [term2] ; The actual addition
        mov  rdi, format   ; About to print
        mov  rax, 1        ; One vector register used (xmm0)
        call printf
        pop  rbp          ; Restore rbp
        ret
```

This addition example — let's call it `addsd.asm` — reads off the command line, using `atof` to parse and the stack to store local variables

```

        global main
        extern printf
        extern atof

        section .text

format:  db      "%f", 10, 0
error:  db      "Two arguments please.", 10, 0

main:    push    rbp
        mov     rbp, rsp
        sub     rsp, 16           ; Reserve space for two quadwords
        cmp     rdi, 3           ; argc check
        jz     continue
        mov     rdi, error
        xor     rax, rax
        call   printf
        jmp    done
continue: mov    [rbp - 16], rsi   ; Save argv
        mov     rdi, [rsi + 8]   ; argv[1]
        call   atof
        mov     rsi, [rbp - 16] ; Restore argv to rsi
        movsdb [rbp - 8], xmm0   ; Save the floating point result
        mov     rdi, [rsi + 16] ; argv[2]
        call   atof
        movsdb xmm1, [rbp - 8]   ; Move the first result into xmm1
        addsd  xmm0, xmm1       ; Add them, with the sum going to xmm0
        mov     rdi, format
        mov     rax, 1           ; 1 vector register used
        call   printf
done:    mov     rsp, rbp        ; Restore the original stack pointer
        pop    rbp              ; Restore rbp
        ret

```

`sqrtsd.asm` converts command line integers into floating point, then displays their square roots

```

        global main
        extern printf
        extern atoi

        section .text

format:  db      "%f", 10, 0
help:    db      "Supply one or more integers, please.", 10, 0

main:    push    rbp
        mov     rbp, rsp
        sub     rsp, 32           ; Reserve space for local variables
        mov     [rbp - 8], rdi    ; Save argc
        mov     [rbp - 16], rsi   ; Save argv
        mov     rcx, 1
        cmp     rdi, rcx         ; Argument check
        jne    L
        mov     rdi, help
        xor     rax, rax
        call   printf
        jmp    done
L:       mov     rdx, [rbp - 16]   ; Restore argv
        mov     rdi, [rdx + rcx*8] ; Access argv[i]
        mov     [rbp - 24], rcx   ; Save the counter
        call   atoi
        cvtsi2sd xmm0, rax        ; Convert to double
        sqrtsd  xmm0, xmm0       ; The money instruction
        mov     rdi, format
        mov     rax, 1
        call   printf
        mov     rcx, [rbp - 24]   ; Restore the counter
        inc    rcx
        cmp     rcx, [rbp - 8]   ; Check for more args
        jl    L
done:    leave   ; Shortcut for mov rsp, rbp; pop rbp
        ret

```

Yes, we could have called `atof` to calculate floating point arguments as well — but then you wouldn't have seen `cvtsi2sd` in action