Recursion Example and Floating Point Instructions

Edited from the work of Ray Seyfarth

September 14, 2015
Recursive Functions

- Recursive algorithms serve as a good example for why we need stack based storage.
- Often times we can get away with utilizing registers that are preserved across function calls.
- However consider the case where we a have a recursive algorithm. On the first level we decide to use r15 to store a value. But now on the second level r15 is already in use...
Consider again the following recursive Fibonacci function

```c
// assuming fib(1)=1
long fib(long n)
{
    if(n<2)
        return 1;
    return fib(n-1)+fib(n-2);
}
```
Recursive Functions

```assembly
fib: push rbp
    mov rbp, rsp
    sub rsp, 16
    N equ 0
    nM1 equ 8
    mov rax, 1
    cmp rdi, 2
    jl .end ; first parameter < 2
    dec rdi
    mov [rsp+N], rdi ; save N-1
    call fib
    mov [rsp+nM1], rax
    mov rdi, [rsp+N] ; load N-1
    dec rdi
    call fib
    add rax, [rsp+nM1]
.end
    leave
    ret
```
If a function is a non-leaf function you must set up and destroy a stack frame.

If you utilize a register in your function that should be preserved across function calls (like r15) you must restore it to its original value.

Example

```
sub rsp, 16
mov [rsp], r15
mov [rsp+8], r14
....
mov r14, [rsp+8]
mov r15, [rsp]
add rsp, 16
```
or if you make use of no other stack based memory.

push r15
push r14
....
....
pop r14
pop r15
Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.
Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.

- x86-64 CPUs have 16 floating point registers (128 or 256 bits)
  - These registers can be used for single data instructions or single instruction multiple data instructions (SIMD)
Floating point instructions

- PC floating point operations were once done in a separate chip - 8087
  - This chip managed a stack of 80 bit floating point values.
  - The stack and instructions still exist, but are largely ignored.
  - In the absence of an 8087 chip, floating point operation where emulated in software.

- x86-64 CPUs have 16 floating point registers (128 or 256 bits)
  - These registers can be used for single data instructions or single instruction multiple data instructions (SIMD)

- We will focus on these newer registers
  - The older instructions tended to start with the letter “f” and referenced the stack using register names like ST0
  - The newer instructions reference using registers with names like “xmm0”, and “ymm0”.

© Edited from the work of Ray Seyfarth
Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is referred to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.
Floating point instructions

- There are 16 floating point registers.
- `ymm0` to `ymm15` (AVX registers)
- Each one is 256 bits.
- The lower half (128 bits) of `ymm-` is refereed to at `xmm-`
- `xmm0` to `xmm15` (SSE registers)
- The full 256 bit register are available from the Core i series.
- We will mainly discuss `xmm` registers, all operations are the same for `ymm` registers, we just append a `v` in front of the instruction.
Moving scalars to or from floating point registers

Moving floating point numbers

- The two instructions available are `movss` and `movsd`.
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single).
- `movsd` moves a single 64 bit floating point value (double).
Moving floating point numbers

- The two instructions available are `movss` and `movsd`
- `movss` moves a single 32 bit floating point value to or from an `xmm` register (float/single)
- `movsd` moves a single 64 bit floating point value (double)
- It should be noted that there is no implicit data conversion - unlike the old instructions which converted floating point data to an 80 bit internal format
Moving scalars to or from floating point registers

The instructions follow the standard pattern of having at most one memory address

```assembly
segment .data
x: dd 12.35 ; float/single
y: dq 14.36 ; double

movss xmm0, [x] ; move the float value at x into xmm0
movsd [y], xmm1 ; move double value from xmm1 to y
movss xmm2, xmm0 ; move from xmm0 to xmm2
```
Moving packed data

- The XMM registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The YMM registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
Moving packed data

- The XMM registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The YMM registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An **aligned** version and an **unaligned** version.
  - **aligned** move requires the data to be of a 16 byte boundary, but is faster in general.
  - **unaligned** move is slower, though more so on older CPUs.
Moving packed data

- The XMM registers are 128 bits
  - They can hold 4 floats or 2 doubles (or integers of various sizes)
- The YMM registers are 256 bits
  - They can hold 8 floats or 4 doubles (or integers of various sizes)
- But how do we load them?
  - There are two types of packed move instructions available.
  - An aligned version and an unaligned version.
  - aligned move requires the data to be of a 16 byte boundary, but is faster in general.
  - unaligned move is slower, though more so on older CPUs.
- If you try an use a aligned move on unaligned data you will get a segmentation fault.
Moving packed data

Actual packed move instructions:

- **movaps** moves 4 floats to/from a memory address aligned at a 16 byte boundary
- **movups** does the same task with unaligned memory addresses
- **movapd** moves 2 doubles to/from a memory address aligned at a 16 byte boundary
- **movupd** does the same task with unaligned memory addresses

```
segment .data
x: dd 12.3, 9.3, 123.2, 0.1
a: dq 0, 0
....
    movups xmm0, [x] ; move 4 floats to xmm0
    movupd [a], xmm15 ; move 2 doubles to a
```
Moving packed data

If you wish to use the aligned move:

```assembly
segment .data
align 16
x: dd 12.3, 9.3, 123.2, 0.1
....
    movaps xmm0, [x]  ; move 4 floats to xmm0
```
Floating point addition

- `addss` adds a scalar float (single precision) to another
- `addsd` adds a scalar double to another
- `addps` adds 4 floats to 4 floats - pairwise addition
- `addpd` adds 2 doubles to 2 doubles
- There are 2 operands: destination and source
- The source can be memory or an XMM register
- The destination must be an XMM register
- Flags are unaffected

```
movss xmm0, [a] ; load a
addss xmm0, [b] ; add b to a
movss [c], xmm0 ; store sum in c
```

And

```
movapd xmm0, [a] ; load 2 doubles from a
addpd xmm0, [b] ; add a[0]+b[0] and a[1]+b[1]
movapd [c], xmm0 ; store 2 sums in c
```
Floating point subtraction

- `subss` subtracts the source float from the destination
- `subsd` subtracts the source double from the destination
- `subps` subtracts 4 floats from 4 floats
- `subpd` subtracts 2 doubles from 2 doubles

```assembly
movss  xmm0, [a] ; load a
subss  xmm0, [b] ; add b from a
movss  [c], xmm0 ; store a-b in c
```

And

```assembly
movapd xmm0, [a] ; load 2 doubles from a
subpd xmm0, [b] ; add a[0]-b[0] and a[1]-b[1]
movapd [c], xmm0 ; store 2 differences in c
```
Basic floating point instructions

<table>
<thead>
<tr>
<th>instruction</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsd</td>
<td>add scalar double</td>
</tr>
<tr>
<td>addss</td>
<td>add scalar float</td>
</tr>
<tr>
<td>addpd</td>
<td>add packed double</td>
</tr>
<tr>
<td>addps</td>
<td>add packed float</td>
</tr>
<tr>
<td>subsd</td>
<td>subtract scalar double</td>
</tr>
<tr>
<td>subss</td>
<td>subtract scalar float</td>
</tr>
<tr>
<td>subpd</td>
<td>subtract packed double</td>
</tr>
<tr>
<td>subps</td>
<td>subtract packed float</td>
</tr>
<tr>
<td>mulsd</td>
<td>multiply scalar double</td>
</tr>
<tr>
<td>mulss</td>
<td>multiply scalar float</td>
</tr>
<tr>
<td>mulpd</td>
<td>multiply packed double</td>
</tr>
<tr>
<td>mulps</td>
<td>multiply packed float</td>
</tr>
<tr>
<td>divsd</td>
<td>divide scalar double</td>
</tr>
<tr>
<td>divss</td>
<td>divide scalar float</td>
</tr>
<tr>
<td>divpd</td>
<td>divide packed double</td>
</tr>
<tr>
<td>divps</td>
<td>divide packed float</td>
</tr>
</tbody>
</table>
Conversion to a different length floating point

- _cvtss2sd_ converts a scalar single (float) to a scalar double
- _cvtps2pd_ converts 2 packed floats to 2 packed doubles
- _cvtsd2ss_ converts a scalar double to a scalar float
- _cvtpd2ps_ converts 2 packed doubles to 2 packed floats

```
cvtss2sd xmm0, [a] ; get a into xmm0 as a double
cvtps2pd xmm0, xmm0 ; add a double to a
cvtsd2ss xmm0, xmm0 ; convert to float
movss [c], xmm0
```
Converting floating point to/from integer

- `cvtss2si` converts a float to a double word or quad word integer by rounding
- `cvtsd2si` converts a float to a double word or quad word integer by rounding
- `cvttss2si` and `cvttsd2si` convert by truncation

When converting from memory a size qualifier is needed:
- `cvtss2si eax, xmm0`; convert to dword integer
- `cvtss2si rax, xmm0`; convert to qword integer
- `cvtsi2sd xmm0, rax`; convert qword to double
- `cvtsi2sd xmm0, dword [x]`; convert dword integer
Converting floating point to/from integer

- `cvtss2si` converts a float to a double word or quad word integer by rounding.
- `cvtsd2si` converts a float to a double word or quad word integer by rounding.
- `cvttss2si` and `cvttsd2si` convert by truncation.
- `cvtsi2ss` converts an integer to a float in an XMM register.
- `cvtsi2sd` converts an integer to a double in an XMM register.
Converting floating point to/from integer

- cvtss2si converts a float to a double word or quad word integer by rounding
- cvtsd2si converts a float to a double word or quad word integer by rounding
- cvttss2si and cvtttsd2si convert by truncation
- cvtsi2ss converts an integer to a float in an XMM register
- cvtsi2sd converts an integer to a double in an XMM register
- When converting from memory a size qualifier is needed

```assembly
  cvtss2si  eax, xmm0 ; convert to dword integer
  cvtss2si  rax, xmm0 ; convert to qword integer
  cvtsi2sd  xmm0, rax ; convert qword to double
  cvtsi2sd  xmm0, dword [x] ; convert dword integer
```
Unordered versus ordered comparisons

- Floating point comparisons can cause exceptions
- Ordered comparisons cause exceptions on QNaN or SNaN
  - QNaN means “quiet not a number”
  - SNaN means “signalling not a number”
  - Both have all exponent field bits set to 1
  - QNaN has its top fraction bit equal to 1
- An unordered comparison causes exceptions only for SNaN
- gcc uses unordered comparisons
- If it’s good enough for gcc, it’s good enough for us
- ucomiss compares floats
- ucomisd compares doubles
- The first operand must be an XMM register
- They set the zero flag, parity flag and carry flags

```assembly
movss  xmm0, [a]
mulss  xmm0, [b]
ucomiss xmm0, [c]
jbe   less_eq  ; jmp if a*b <= c
```
## Conditional floating point jumps

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>aliases</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>jb</td>
<td>jump if below</td>
<td>jc, jnae</td>
<td>CF=1</td>
</tr>
<tr>
<td>jbe</td>
<td>jump if below or equal</td>
<td>jna</td>
<td>ZF=1 or CF=1</td>
</tr>
<tr>
<td>ja</td>
<td>jump if above</td>
<td>jnbe</td>
<td>ZF=0 or CF=0</td>
</tr>
<tr>
<td>jae</td>
<td>jump if above or equal</td>
<td>jnc, jnb</td>
<td>CF=0</td>
</tr>
<tr>
<td>je</td>
<td>jump if equal</td>
<td>jz</td>
<td>ZF=1</td>
</tr>
<tr>
<td>jne</td>
<td>jump if not equal</td>
<td>jnz</td>
<td>ZF=0</td>
</tr>
</tbody>
</table>

- c = carry flag set
- z = zero flag set
Mathematical functions

- 8087 had sine, cosine, arctangent and more
- The newer instructions omit these operations on XMM registers
- Instead you are supposed to use efficient library functions
- There are instructions for
  - Minimum
  - Maximum
  - Rounding
  - Square root
  - Reciprocal of square root
Minimum and maximum

- `minss` and `maxss` compute minimum or maximum of scalar floats
- `minsd` and `maxsd` compute minimum or maximum of scalar doubles
- The destination operand must be an XMM register
- The source can be an XMM register or memory
- `minps` and `maxps` compute minimum or maximum of packed floats
- `minpd` and `maxpd` compute minimum or maximum of packed doubles
- `minps xmm0, xmm1` computes 4 minimums and places them in `xmm0`

```assembly
movss  xmm0, [x]  ; move x into xmm0
maxss  xmm0, [y]  ; xmm0 has max(x,y)
movapd xmm0, [a]  ; move a[0] and a[1] into xmm0
minpd xmm0, [b]   ; xmm0[0] has min(a[0],b[0])
                 ; xmm0[1] has min(a[1],b[1])
```
Rounding

- `roundss` rounds 1 float
- `roundps` rounds 4 floats
- `roundsd` rounds 1 double
- `roundpd` rounds 2 doubles

The first operand is an XMM destination register.
The second is the source in an XMM register or memory.
The third operand is a rounding mode.

<table>
<thead>
<tr>
<th>mode</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>round, giving ties to even numbers</td>
</tr>
<tr>
<td>1</td>
<td>round down</td>
</tr>
<tr>
<td>2</td>
<td>round up</td>
</tr>
<tr>
<td>3</td>
<td>round toward 0 (truncate)</td>
</tr>
</tbody>
</table>
Square roots

- `sqrtss` computes 1 float square root
- `sqrtps` computes 4 float square roots
- `sqrtsd` computes 1 double square root
- `sqrtpd` computes 2 double square roots
- The first operand is an XMM destination register
- The second is the source in an XMM register or memory
Distance in 3D

\[ d = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2)} \]

```assembly
distance3d:
    movss  xmm0, [rdi] ; x from first point
    subss  xmm0, [rsi] ; subtract x from second point
    mulss  xmm0, xmm0 ; \( (x_1-x_2)^2 \)
    movss  xmm1, [rdi+4] ; y from first point
    subss  xmm1, [rsi+4] ; subtract y from second point
    mulss  xmm1, xmm1 ; \( (y_1-y_2)^2 \)
    movss  xmm2, [rdi+8] ; z from first point
    subss  xmm2, [rsi+8] ; subtract z from second point
    mulss  xmm2, xmm2 ; \( (z_1-z_2)^2 \)
    addss  xmm0, xmm1 ; add x and y parts
    addss  xmm0, xmm2 ; add z part
    sqrtss xmm0, xmm0
    ret
```
Dot product in 3D

\[ d = x_1 x_2 + y_1 y_2 + z_1 z_2 \]

dot_product:

```
    movss  xmm0, [rdi]
    mulss  xmm0, [rsi]
    movss  xmm1, [rdi+4]
    mulss  xmm1, [rsi+4]
    addss  xmm0, xmm1
    movss  xmm2, [rdi+8]
    mulss  xmm2, [rsi+8]
    addss  xmm0, xmm2
    ret
```