1. x86-64 Assembly

- Backwards compatible extension to x86-32
  - AMD64
  - EM64T (Intel64)
  - To get x86-64 mode, throw `-m64` gcc compile/link
- Same basic assembly as x86-32
- Has 32 (legacy) mode
  - Runs IA32 code unchanged.
  - Does not allow access to new regs or 64-bit int inst
  - To get legacy mode, throw `-m32` gcc compile/link
- x86-64 has same basic instructions, but different regs, stack, and ABI

64 bit Mode:

- Pointers 64 bits, but ints are still 32!
  - → Translate 32 bit int to 64 by bit extending sign bit
  - `cltq` sign extends `%eax` to `%rax`
  - `movslq reg32, reg64` sign extends reg32 into reg64
  - Positive # has zeros, so just `xor` before move
- Has 16 integer and SSE registers!
- Passes args in registers
- 64 register prefix is `r`
  - %esp → %rsp
  - %edi → %rdi
- 64 bit int ops suffixed by `q`
  - `movl` → `movq`

2. x86-64 Registers Usage

REGISTER	USAGE	CALLEE SAVE
%rsp	Stack pointer	YES
%rbx	optional base pointer	YES
%rbp	optional frame pointer	YES
%rax	integer return val	NO
%rdi	1st int arg	NO
%rsi	2nd int arg	NO
%rdx	3rd int arg	NO
%rcx	4th int arg	NO
%r8	5th int arg	NO
%r9	6th int arg	NO
%r10	used to pass static chain pointer	NO
%r11	scratch reg	NO
%r12-15	callee-saved regs	YES
%xmm0-1	pass & return fp args	NO
%xmm2-7	pass fp args	NO
%xmm8-15	scratch regs	NO
%mmx0-7	scratch regs, aliased to fp stack	NO

- r8-15 64-bit only
- %esp is low 32 bits of %rsp, etc.
- Additional args overflow to stack

3. x86-64 Calling Sequence and Stack Frame

- Stack grows downward in mem
- %rsp-8 aligned to 16-byte boundary
  - → arg area 16-byte aligned
- Callees args in caller's frame
- Red zone reserved workspace area for leaf functions
- Args 1st pased in regs, overflowed to stack
  - 7th iarg and 9th fparg overflow
- All arglengths rounded up to 8 bytes
  - → 4-byte int passed in `edi` of `rdi`

	Caller's frame
	last overflow arg
	⋮
8(%rsp)	1st overflow arg
0(%rsp)	return address
-8(%rsp)	begin red zone (16-byte al
-128(%rsp)	end of red zone

Stack frame passed to callee

4. x86-64 Assembly Tips

- Often want to convert 32 bit ints to 64 in preamble:
  - Makes ptrs and ints same type again
    - → critical for mem addressing!
  - Allows use of 16 additional iregs
- Can use cpp macros to combine x86-32/x86-64 assembly:
  - Must #define appropriate int commands
  - Must do any 32/64 conversion in preamble
  - Any use of extra 64-bit regs must occur in only x86-64 code
    - → Use in-mem operands in 32-bit code

5. x86-32/64 SSE2 DASUM, 1 of 3

```
#ifdef ATL_GAS_x8632
   #define movq movl
   #define addq addl
   #define subq subl
   #define rsp  esp
   #define rax  eax
   #define N    %eax
   #define X    %edx
   #define stX  %ecx
   #define stXF %ebx
#else
   #define N    %rax
   #define X    %rsi
   #define stX  %rdi
   #define stXF %rdx
#endif
#define absval  %xmm0
#define rX0     %xmm1
#define rX1     %xmm2
#define rX2     %xmm3
#define rX3     %xmm4
#define sum0    %xmm5
#define sum1    %xmm6
#define sum2    %xmm7
```

```
#if defined(ATL_OS_WinXX) || defined(ATL_OS_OSX)
   #define Mjoin(pre, nam) my_join(pre, nam)
   #define my_join(pre, nam) pre ## nam
.global   Mjoin(_,ATL_UASUM)
Mjoin(_,ATL_UASUM):
#else
.global   ATL_UASUM
ATL_UASUM:
#endif
   xorpd   absval, absval    # av = {0,0,0,0}
   movl    $0xFFFF, %eax     # ax = 0xFFFF
   pinsrw  $0, %eax, absval  # av = {0,0x000000000000FFFF}
   pinsrw  $1, %eax, absval  # av = {0,0x00000000FFFFFFFF}
   pinsrw  $2, %eax, absval  # av = {0,0x0000FFFFFFFFFFFF}
   shrl    $1, %eax          # ax = 0x7FFF
   pinsrw  $3, %eax, absval  # av = {0,0x7FFFFFFFFFFFFFFF}
   unpcklpd   absval, absval # av = {0x7FFFFFFFFFFFFFFF,0x7F
#ifdef ATL_GAS_x8632
   subl   $16, %esp
   movl   %ebx, (%esp)
   movl   20(%esp), N
   movl   24(%esp), X
#else
   movl   %edi, %eax
   cltq
#endif
```

6. x86-32/64 SSE2 DASUM, 2 of 3

```
   movq   N, stXF
   shl   $3, stXF
   addq   X, stXF  # stXF = X + N*sizeof
#  If X%16 != 0, peel 1 iteration
   xorpd   sum0, sum0
   movq   X, stX
   shr   $4, stX
   shl   $4, stX   # stX = (X/16)*16
   cmp   X, stX
   je   ALIGNED_START
   movlpd   (X), sum0
   andpd   absval, sum0
   addq   $8, X
   dec   N
   jz   DONE
ALIGNED_START:
   movq   N, stX
   shr   $3, stX
   jz   UNALIGNED_LOOP
   shl   $6, stX
   addq   X, stX   # stX = X+(N/8)*8*sizeof
   xorpd   sum1, sum1
   xorpd   sum2, sum2
```

```
ALIGNED_LOOP:
   movapd   (X), rX0
   movapd   16(X), rX1
   movapd   32(X), rX2
   movapd   48(X), rX3
   andpd   absval, rX0
#if defined(ATL_ARCH_HAMMER64) || defined(AT
   prefetchnta 640(X)
#else
   prefetchnta 1024(X)
#endif
   andpd   absval, rX1
   addpd   rX0, sum0
   andpd   absval, rX2
   addpd   rX1, sum1
   andpd   absval, rX3
   addpd   rX2, sum2
   addpd   rX3, sum0
   addq   $64, X
   cmp   X, stX
   jne   ALIGNED_LOOP
```

7. x86-32/64 SSE2 DASUM, 3 of 3

```
   addpd    sum1, sum0          # sum0 = {s0b+s1b, s0a+s1a}
   addpd    sum2, sum0          # sum0 = {s0b+s1b+s2b, s0a+s1a+s2a}
   movapd   sum0, sum1
   unpckhpd sum1, sum1          # sum1 = {X          , s0b+s1b+s2b}
   addsd    sum1, sum0          # sum0 = {X          , total sum}
   cmp      X, stXF
   jne      UNALIGNED_LOOP
DONE:
#ifdef ATL_GAS_x8632
   movl   (%esp), %ebx
   movlpd   sum0, (%esp)
   fldl   (%esp)
   addl   $16, %esp
#else
   movsd   sum0, %xmm0
#endif
   ret
UNALIGNED_LOOP:
   movlpd   (X), rX0
   andpd   absval, rX0
   addsd   rX0, sum0
   addq   $8, X
   cmp   X, stXF
   jne   UNALIGNED_LOOP
   jmp     DONE
```