

Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015
xeno@legbacre.com

All materials is licensed under a Creative Commons “Share Alike” license.

- <http://creativecommons.org/licenses/by-sa/3.0/>

You are free:



to **Share** — to copy, distribute and transmit the work



to **Remix** — to adapt the work

Under the following conditions:



Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at <http://OpenSecurityTraining.info/IntroX86-64.html>"

Attribution condition: You must indicate that derivative work

"Is derived from Xeno Kovah's 'Intro x86-64' class, available at <http://OpenSecurityTraining.info/IntroX86-64.html>"

ShiftExample0.c

```
void main()
```

```
{
```

```
    unsigned int a, b, c;
```

```
    a = 0x5;
```

```
    b = a << 4;
```

```
    c = b >> 3;
```

```
    return c;
```

```
}
```

```
main:
```

```
00000000140001000 sub
```

```
rsp,18h
```

```
00000000140001004 mov
```

```
dword ptr [rsp],5
```

```
0000000014000100B mov
```

```
eax,dword ptr [rsp]
```



```
0000000014000100E shl
```

```
eax,4
```

```
00000000140001011 mov
```

```
dword ptr [rsp+4],eax
```

```
00000000140001015 mov
```

```
eax,dword ptr [rsp+4]
```



```
00000000140001019 shr
```

```
eax,3
```

```
0000000014000101C mov
```

```
dword ptr [rsp+8],eax
```

```
00000000140001020 mov
```

```
eax,dword ptr [rsp+8]
```

```
00000000140001024 add
```

```
rsp,18h
```

```
00000000140001028 ret
```



SHL - Shift Logical Left

- Can be explicitly used with the C “<<” operator
- First operand (source and destination) operand is an r/mX
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **multiplies** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the left hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shl bl, 2

	00110011b (bl - 0x33)
result	11001100b (bl - 0xCC) CF = 0

shl bl, 3

	00110011b (bl - 0x33)
result	10011000b (bl - 0x98) CF = 1



SHR - Shift Logical Right

- Can be explicitly used with the C “>>” operator
- First operand (source and destination) operand is an r/mX
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **divides** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the right hand side are “shifted into” (set) the carry flag (CF)
- For purposes of determining if the CF is set at the end, think of it as n independent 1 bit shifts.

shr bl, 2

	00110011b (bl - 0x33)
result	00001100b (bl - 0x0C) CF = 1

shr bl, 3

	00110011b (bl - 0x33)
result	00000110b (bl - 0x06) CF = 0

ShiftExample0.c Takeaways:

- Shifts in C are shifts in asm :P

<pre>void main() { unsigned int a, b, c; a = 0x5; b = a << 4; c = b >> 3; return c; }</pre>	<pre>main: 00000000140001000 sub rsp,18h 00000000140001004 mov dword ptr [rsp],5 0000000014000100B mov eax,dword ptr [rsp] 0000000014000100E shl eax,4 00000000140001011 mov dword ptr [rsp+4],eax 00000000140001015 mov eax,dword ptr [rsp+4] 00000000140001019 shr eax,3 0000000014000101C mov dword ptr [rsp+8],eax 00000000140001020 mov eax,dword ptr [rsp+8] 00000000140001024 add rsp,18h 00000000140001028 ret</pre>
---	---

ShiftExample1.c

```
#include <stdio.h>
#include <stdlib.h>
```

Note: Compiled with “Maximize Speed”, to clear away a bit of cruft

```
int main(int argc, char **argv)
{
    unsigned int a, b, c;
    a = atoi(argv[0]);
    b = a * 8;
    c = b / 16;
    return c;
}
```

```
main:
00000000140001010 sub    rsp,28h
00000000140001014 mov    rcx,qword ptr [rdx]
00000000140001017 call   qword ptr [40008368h]
0000000014000101D shl    eax,3
00000000140001020 shr    eax,4
00000000140001023 add    rsp,28h
00000000140001027 ret
```

Whither the multiply and divide instructions?!

ShiftExample1.c takeaways

- When a multiply or divide is by a power of 2, compilers prefer shift instructions as a more efficient way to perform the computation

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
    unsigned int a, b, c;
```

```
    a = atoi(argv[0]);
```

```
    b = a * 8;
```

```
    c = b / 16;
```

```
    return c;
```

```
}
```

```
main:
```

```
00000000140001010 sub    rsp,28h
```

```
00000000140001014 mov    rcx,qword ptr [rdx]
```

```
00000000140001017 call   qword ptr [40008368h]
```

```
0000000014000101D shl    eax,3
```

```
00000000140001020 shr    eax,4
```

```
00000000140001023 add    rsp,28h
```

```
00000000140001027 ret
```


That's the power of love sign!

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char **argv)
{
    unsigned int a, b, c;
    a = atoi(argv[0]);
    b = a * 8;
    c = b / 16;
    return c;
}
```



```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char **argv)
{
    int a, b, c;
    a = atoi(argv[0]);
    b = a * 8;
    c = b / 16;
    return c;
}
```

ShiftExample2.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int a, b, c;
    a = atoi(argv[0]);
    b = a * 8;
    c = b / 16;
    return c;
}

main:
00000000140001010 sub    rsp,28h
00000000140001014 mov     rcx,qword ptr [rdx]
00000000140001017 call    qword ptr [40008368h]
0000000014000101D shl     eax,3
★ 00000000140001020 cdq
00000000140001021 and     edx,0Fh
00000000140001024 add     eax,edx
★ 00000000140001026 sar     eax,4
00000000140001029 add     rsp,28h
0000000014000102D ret
```

```

main:
00000000140001010 sub    rsp,28h
00000000140001014 mov     rcx,qword ptr [rdx]
00000000140001017 call    qword ptr [40008368h]
0000000014000101D shl     eax,3
00000000140001020 shr     eax,4
00000000140001023 add     rsp,28h
00000000140001027 ret

```

Vs

```

main:
00000000140001010 sub    rsp,28h
00000000140001014 mov     rcx,qword ptr [rdx]
00000000140001017 call    qword ptr [40008368h]
0000000014000101D shl     eax,3
00000000140001020 cdq
00000000140001021 and     edx,0Fh
00000000140001024 add     eax,edx
00000000140001026 sar     eax,4
00000000140001029 add     rsp,28h
0000000014000102D ret

```

} Changed

CD* is added as an VS-ism. It's necessary for the math to work out, but I feel like I've only run into it *once ever* in the wild. So I don't consider it that important for beginners to know and I'm skipping it. But you can feel free to come back and read this code once we've gone through the RTFM section.



SAR - Shift Arithmetic Right

- Can be explicitly used with the C “>>” operator, if operands are signed
- First operand (source and destination) operand is an r/mX
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It divides the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Each bit shifted off the right side is placed in CF.

sar bl, 2

	10110011b (bl - 0xB3)
result	11 101100b (bl - 0xEC)

!=

shr bl, 2

	10110011b (bl - 0xB3)
result	00 101100b (bl - 0x2C)

mov cl, 2; sar bl, cl

	00110011b (bl - 0x33)
result	00 001100b (bl - 0x0C)

==

mov cl, 2; shr bl, cl

	00110011b (bl - 0x33)
result	00 001100b (bl - 0x0C)



SAL - Shift Arithmetic Left

- Actually behaves exactly the same as SHL!
- First operand (source and destination) operand is an r/mX
- Second operand is either cl (lowest byte of rcx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It divides the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Each bit shifted off the left side is placed in CF.

sal bl, 2

	10110011b (bl - 0xB3)
result	110011 00 b (bl - 0xCC)

==

shl bl, 2

	10110011b (bl - 0xB3)
result	110011 00 b (bl - 0xCC)

mov cl, 2; sal bl, cl

	00110011b (bl - 0x33)
result	110011 00 b (bl - 0xCC)

==

mov cl, 2; shl bl, cl

	00110011b (bl - 0x33)
result	110011 00 b (bl - 0xCC)

ShiftExample2.c takeaways

- Compilers still prefer shifts for mul/div over powers of 2
- But when the operands are *signed* rather than unsigned, it must use different instructions, and potentially do more work (than the unsigned case) to deal with a multiply
- CDQ isn't important for beginners to know, left as an exercise for the reader for later ;)

```
int main(){  
    int a, b, c;  
    a = 0x40;  
    b = a * 8;  
    c = b / 16;  
    return c;  
}
```

```
main:  
00000000140001010 sub    rsp,28h  
00000000140001014 mov     rcx,qword ptr [rdx]  
00000000140001017 call    qword ptr [40008368h]  
0000000014000101D shl     eax,3  
00000000140001020 cdq  
00000000140001021 and     edx,0Fh  
00000000140001024 add     eax,edx  
00000000140001026 sar     eax,4  
00000000140001029 add     rsp,28h  
0000000014000102D ret
```

Instructions we now know (26)

- NOP
- PUSH/POP
- CALL/RET
- MOV
- ADD/SUB
- IMUL
- MOVZX/MOVSX
- LEA
- JMP/Jcc (family)
- CMP/TEST
- AND/OR/XOR/NOT
- INC/DEC
- SHR/SHL/SAR/SAL