Registers in x86 Assembly

CS 301: Assembly Language Programming Lecture, Dr. Lawlor

Like C++ variables, registers are actually available in several sizes:

- rax is the 64-bit, "long" size register. It was added in 2003 during the transition to 64-bit processors.
- eax is the 32-bit, "int" size register. It was added in 1985 during the transition to 32-bit processors with the 80386 CPU. I'm in the habit of using this register size, since they also work in 32 bit mode, although I'm trying to use the longer rax registers for everything.
- ax is the 16-bit, "short" size register. It was added in 1979 with the 8086 CPU, but is used in DOS or BIOS code to this day.
- al and ah are the 8-bit, "char" size registers. al is the low 8 bits, ah is the high 8 bits. They're pretty similar to the old 8-bit registers of the 8008 back in 1972.

Curiously, you can write a 64-bit value into rax, then read off the low 32 bits from eax, or the low 16 bitx from ax, or the low 8 bits from al--it's just one register, but they keep on extending it!

rax: 64-bit	eax: 32-bit	ax: 16-bit	ah	al

For example,

```
mov rcx,0xf00d00d2beefc03; load a big 64-bit constant
mov eax,ecx; pull out low 32 bits (0x2beefc03)
ret
```

(Try this in NetRun now!)

Here's the full list of x86 registers. The 64 bit registers are shown in red. "Scratch" registers any function is allowed to overwrite, and use for anything you want without asking anybody. "*Preserved*" registers have to be put back ("save" the register) if you use them.

Name	**Notes**	Type	**64-bit long**	**32-bit int**	**16-bit short**	**8-bit char**
rax	Values are returned from functions in this register.	scratch	rax	eax	ax	ah and al
rcx	Typical scratch register. Some instructions also use it as a counter.	scratch	rcx	ecx	cx	ch and cl
rdx	Scratch register.	scratch	rdx	edx	dx	dh and dl
rbx	*Preserved register: don't use it without saving it!*	*preserved*	*rbx*	*ebx*	*bx*	*bh and bl*
rsp	*The stack pointer. Points to the top of the stack (details coming soon!)*	*preserved*	*rsp*	*esp*	*sp*	*spl*
rbp	*Preserved register. Sometimes used to store the old value of the stack pointer, or the "base".*	*preserved*	*rbp*	*ebp*	*bp*	*bpl*
rsi	Scratch register. Also used to pass function argument #2 in 64-bit Linux	scratch	rsi	esi	si	sil
rdi	Scratch register. Function argument #1 in 64-bit Linux	scratch	rdi	edi	di	dil
r8	Scratch register. These were added in 64-bit mode, so they have numbers, not names.	scratch	r8	r8d	r8w	r8b
r9	Scratch register.	scratch	r9	r9d	r9w	r9b
r10	Scratch register.	scratch	r10	r10d	r10w	r10b
r11	Scratch register.	scratch	r11	r11d	r11w	r11b
r12	*Preserved register. You can use it, but you need to save and restore it.*	*preserved*	*r12*	*r12d*	*r12w*	*r12b*
r13	*Preserved register.*	*preserved*	*r13*	*r13d*	*r13w*	*r13b*
r14	*Preserved register.*	*preserved*	*r14*	*r14d*	*r14w*	*r14b*
r15	*Preserved register.*	*preserved*	*r15*	*r15d*	*r15w*	*r15b*

You can convert values between different register sizes using different instructions:

	Source Size				
	64 bit rcx	32 bit ecx	16 bit cx	8 bit cl	**Notes**
64 bit rax	mov rax,rcx	movsxd rax,ecx	movsx rax,cx	movsx rax,cl	Writes to whole register
32 bit eax	mov eax,ecx	mov eax,ecx	movsx eax,cx	movsx eax,cl	Top half of destination gets zeroed

16 bit ax	mov ax,cx	mov ax,cx	mov ax,cx	movsx ax,cl	Only affects low 16 bits, rest unchanged.
8 bit al	mov al,cl	mov al,cl	mov al,cl	mov al,cl	Only affects low 8 bits, rest unchanged.

Overflow

The fact is, variables on a computer only have so many bits. If the value gets bigger than can fit in those bits, the extra bits first go negative and then "overflow". By default they're then ignored completely.

```
int big=1024*1024*1024;
return big*4;
```

(Try this in NetRun now!)

On my machine, "int" is 32 bits, which is +-2 billion in binary, so this actually returns 0?!

```
        Program complete.  Return 0 (0x0)
```

You can extract the value of each bit. For example:

```
int value=1; /* value to test, starts at first (lowest) bit */
for (int bit=0;bit<100;bit++) {
        std::cout<<"at bit "<<bit<<" the value is "<<value<<"\n";
        value=value+value; /* moves over by one bit */
        if (value==0) break;
}
return 0;
```

(Try this in NetRun now!)

Because "int" currently has 32 bits, if you start at one, and add a variable to itself 32 times, the one overflows and is lost completely.

In assembly, there's a handy instruction "jo" (jump if overflow) to check for overflow from the previous instruction. The C++ compiler doesn't bother to use jo, though!

```
mov edi,1 ; loop variable
mov eax,0 ; counter

start:
        add eax,1 ; increment bit counter

        add edi,edi ; add variable to itself
        jo noes ; check for overflow in the above add

        cmp edi,0
        jne start

ret

noes: ; called for overflow
        mov eax,999
        ret
```

(Try this in NetRun now!)

Notice the above program returns 999 on overflow, which somebody else will need to check for. (Responding correctly to overflow is actually quite difficult--see, e.g., Ariane 5 explosion, caused by poor handling of a detected overflow. Ironically, ignoring the overflow would have caused no problems!)

Signed versus Unsigned Numbers

If you watch closely right before overflow, you see something funny happen:

```
signed char value=1; /* value to test, starts at first (lowest) bit */
for (int bit=0;bit<100;bit++) {
        std::cout<<"at bit "<<bit<<" the value is "<<(long)value<<"\n";
        value=value+value; /* moves over by one bit (value=value<<1 would work too) */
        if (value==0) break;
}
return 0;
```

(Try this in NetRun now!)

This prints out:

```
at bit 0 the value is 1
at bit 1 the value is 2
at bit 2 the value is 4
at bit 3 the value is 8
at bit 4 the value is 16
at bit 5 the value is 32
at bit 6 the value is 64
at bit 7 the value is -128
Program complete.  Return 0 (0x0)
```

Wait, the last bit's value is -128? Yes, it really is!

This negative high bit is called the "sign bit", and it has a negative value in two's complement signed numbers. This means to represent -1, for example, you set not only the high bit, but all the other bits as well: in unsigned, this is the largest possible value. The reason binary 11111111 represents -1 is the same reason you might choose 9999 to represent -1 on a 4-digit odometer: if you add one, you wrap around and hit zero.

A very cool thing about two's complement is addition is *the same operation* whether the numbers are signed or unsigned--we just interpret the result differently. Subtraction is also identical for signed and unsigned. Register names are identical in assembly for signed and unsigned. However, when you change register sizes using an instruction like "movsxd rax,eax", when you check for overflow, when you compare numbers, multiply or divide, or shift bits, you need to know if the number is signed (has a sign bit) or unsigned (no sign bit, no negative numbers).

Signed	Unsigned	Language
int	unsigned int	C++, int is signed by default.
signed char	unsigned char	C++, char may be signed or unsigned.
movsxd	movzxd	Assembly, sign extend or zero extend to change register sizes.
jo	jc	Assembly, **o**verflow is calculated for signed values, **c**arry for unsigned values.
jg	ja	Assembly, jump **g**reater is signed, jump **a**bove is unsigned.
jl	jb	Assembly, jump **l**ess signed, jump **b**elow unsigned.
imul	mul	Assembly, imul is signed (and more modern), mul is for unsigned (and ancient and horrible!). idiv/div work similarly.