The Art of Picking Intel Registers

I wrote this article for an online magazine called Scene Zine. Scene Zine caters to the Demo Scene, which is an digital art community dedicated to pushing the limits of computers through a mix of music, art, and computer programming. A particular category of demoscene productions, 4K intros, focus on the final production's raw file size. The goal is to put as much high-quality music, graphics, and animation as possible into only 4096 bytes. Doing this requires highly-specialized size optimization techniques, since 4096 bytes is less space than two pages of typed text or a true-color Windows XP icon. This article discusses some of these techniques.

Some people have commented that they want to see more expert programming articles in Scene Zine. To remedy the situation, this article is for all assembly language programmers out there. It discusses the fine art of picking which registers to use in your code. This information should simplify your coding and help you write smaller routines.

When the engineers at Intel designed the original 8086 processor, they had a special purpose in mind for each register. As they designed the instruction set, they created many optimizations and special instructions based on the function they expected each register to perform. Using registers according to Intel's original plan allows the code to take full advantage of these optimizations. Unfortunately, this seems to be a lost art. Few coders are aware of Intel's overall design, and most compilers are too the simplistic or focused on execution speed to use the registers properly. Understanding how the registers and instruction set fit together, however, is an important step on the road to effortless size-coding.

Using the registers consistently has other advantages besides size optimization. Like using good variable names, using consistent registers makes code more readable. When they are used properly, the registers have meanings almost as clear as the loop counter, *i*, in higher-level languages. In fact, I occasionally name my variables in C after x86 registers because the register names are so descriptive. With proper register use, x86 assembler can be almost as self-documenting as a high-level language.

Another benefit that consistent register use brings is better compression. In productions which use a compressor to pack the final build, such as 4K intros, creating more redundant code leads to smaller packed sizes. When code uses registers consistently, the same instruction sequences begin to appear over and over. This, in turn, improves the compression ratio.

As a review, all x86-family CPU's have 8 general-purpose registers. The registers are 32 bits wide, although 16-bit versions are also accessible with a special one-byte instruction prefix. In 16-bit mode, the situation is reversed. The lower 16 bits are accessible by default, and the full registers are accessible only with a prefix byte.

Each register name is really an acronym. This is true even for the "alphabetical" registers EAX, EBX, ECX, and EDX. The following list shows the register names and their meanings:

- EAX - Accumulator Register
- EBX - Base Register
- ECX - Counter Register
- EDX - Data Register
- ESI - Source Index
- EDI - Destination Index
- EBP - Base Pointer
- ESP - Stack Pointer

In addition to the full-sized general registers, the x86 processor also has eight byte-sized registers. Since these registers map directly into EAX, EBX, ECX, and EDX, most people view them as parts of the larger registers. From the instruction set point of view, however, the 8-bit registers are separate entities. For example, the CL and CH registers share none of the ECX register's useful properties. Except for AL and AH, none of the 8-bit registers have any special significance in the instruction set, so this article does not mention them.

EAX: The Accumulator

There are three major processor architectures: register, stack, and accumulator. In a register architecture, operations such as addition or subtraction can occur between any two arbitrary registers. In a stack architecture, operations occur between the top of the stack and other items on the stack. In an accumulator architecture, the processor has single calculation register called the accumulator. All calculations occur in the accumulator, and the other registers act as simple data storage locations.

Obviously, the x86 processor does not have an accumulator architecture. It does, however, have an accumulator-like register: EAX / AL. Although most calculations can occur between any two registers, the instruction set gives the accumulator special preference as a calculation register. For example, all nine basic operations (ADD, ADC, AND, CMP, OR, SBB, SUB, TEST, and XOR) have special one-byte opcodes for operations between the accumulator and a constant. Specialized operations, such as multiplication, division, sign extension, and BCD correction can only occur in the accumulator.

Since most calculations occur in the accumulator, the x86 architecture contains many optimized instructions for moving data in and out of this register. To start, the processor has sixteen byte-sized XCHG opcodes for swapping data between the accumulator and any other register. These aren't terribly useful, but they show how strongly the Intel engineers preferred the accumulator over the other registers. For them, it was better to swap data into the accumulator to than to work with it where it was. Other instructions that move data in and out of the accumulator are LODS, STOS, IN, OUT, INS, OUTS, SCAS, and XLAT. Finally, the MOV instruction has a special one-byte opcode for moving data into the accumulator from a constant memory location.

In your code, try to perform as much work in the accumulator as possible. As you will see, the remaining seven general-purpose registers exist primarily to support the calculation occurring in the accumulator.

EDX: The Data Register

Of the seven remaining general-purpose registers, the data register, EDX, is most closely tied to the accumulator. Instructions that deal with over sized data items, such as multiplication, division, CWD, and CDQ, store the most significant bits in the data register and the least significant bits in the accumulator. In a sense, the data register is the 64-bit extension of the accumulator. The data register also plays a part in IO instructions. In this case, the accumulator holds the data to read or write from the port, and the data register holds the port address.

In your code, the data register is most useful for storing data related to the accumulator's calculation. In my experience, most calculations need only these two registers for storage if they are written properly.

ECX: The Count Register

The count register, ECX, is the x86 equivalent of the ubiquitous variable i. Every counting-related instruction in the x86 uses ECX. The most obvious counting instructions are LOOP, LOOPZ, and LOOPNZ. Another counter-based instruction is JCXZ, which, as the name implies, jumps when the counter is 0. The count register also appears in some bit-shift operations, where it holds the number of shifts to perform. Finally, the count register controls the string instructions through the REP, REPE, and REPNE prefixes. In this case, the count register determines the maximum number of times the operation will repeat.

Particularly in demos, most calculations occur in a loop. In these situations, ECX is the logical choice for the loop counter, since no other register has so many branching operations built around it. The only problem is that this register counts downward instead of up as in high level languages. Designing a downward-counting is not hard, however, so this is only a minor difficulty.

EDI: The Destination Index

Every loop that generates data must store the result in memory, and doing so requires a moving pointer. The destination index, EDI, is that pointer. The destination index holds the implied write address of all string operations. The most useful string instruction, remarkably enough, is the seldom-used STOS. STOS copies data from the accumulator into memory and increments the destination index. This one-byte instruction is perfect, since the final result of any calculation should be in the accumulator anyhow, and storing results in a moving memory address is a common task.

Many coders treat the destination index as no more than extra storage space. This is a mistake. All routines must store data, and some register must serve as the storage pointer. Since the destination index is designed for this job, using it for extra storage is a waste. Use the stack or some other register for storage, and use EDI as your global write pointer.

ESI: The Source Index

The source index, ESI, has the same properties as the destination index. The only difference is that the source index is for reading instead of writing. Although all data-processing routines write, not all read, so the source index is not as universally useful. When the time comes to use it, however, the source index is just as powerful as the destination index, and has the same type of instructions.

In situations where your code does not read any sort of data, of course, using the source index for convenient storage space is acceptable.

ESP and EBP: The Stack Pointer and the Base Pointer

Of the eight general purpose registers, only the stack pointer, ESP, and the base pointer, EBP, are widely used for their original purpose. These two registers are the heart of the x86 function-call mechanism. When a block of code calls a function, it pushes the parameters and the return address on the stack. Once inside, function sets the base pointer equal to the stack pointer and then places its own internal variables on the stack. From that point on, the function refers to its parameters and variables relative to the base pointer rather than the stack pointer. Why not the stack pointer? For some reason, the stack pointer lousy addressing modes. In 16-bit mode, it cannot be a square-bracket memory offset at all. In 32-bit mode, it can be appear in square brackets only by adding an expensive SIB byte to the opcode.

In your code, there is never a reason to use the stack pointer for anything other than the stack. The base pointer, however, is up for grabs. If your routines pass parameters by register instead of by stack (they should), there is no reason to copy the stack pointer into the base pointer. The base pointer becomes a free register for whatever you need.

EBX: The Base Register

In 16-bit mode, the base register, EBX, acts as a general-purpose pointer. Besides the specialized ESI, EDI, and EBP registers, it is the only general-purpose register that can appear in a square-bracket memory access (For example, MOV [BX], AX). In the 32-bit world, however, any register may serve as a memory offset, so the base register is no longer special.

The base register gets its name from the XLAT instruction. XLAT looks up a value in a table using AL as the index and EBX as the base. XLAT is equivalent to MOV AL, [BX+AL], which is sometimes useful if you need to replace one 8-bit value with another from a table (Think of color look-up).

So, of all the general-purpose registers, EBX is the only register without an important dedicated purpose. It is a good place to store an extra pointer or calculation step, but not much more.

Conclusion

The eight general-purpose registers in the x86 processor family each have a unique purpose. Each register has special instructions and opcodes which make fulfilling this purpose more convenient or efficient. The registers and their uses are shown briefly below:

- EAX - All major calculations take place in EAX, making it similar to a dedicated accumulator register.
- EDX - The data register is the an extension to the accumulator. It is most useful for storing data related to the accumulator's current calculation.
- ECX - Like the variable i in high-level languages, the count register is the universal loop counter.
- EDI - Every loop must store its result somewhere, and the destination index points to that place. With a single-byte STOS instruction to write data out of the accumulator, this register makes data operations much more size-efficient.
- ESI - In loops that process data, the source index holds the location of the input data stream. Like the destination index, EDI has a convenient one-byte instruction for loading data out of memory into the accumulator.
- ESP - ESP is the sacred stack pointer. With the important PUSH, POP, CALL, and RET instructions requiring it's value, there is never a good reason to use the stack pointer for anything else.
- EBP - In functions that store parameters or variables on the stack, the base pointer holds the location of the current stack frame. In other situations, however, EBP is a free data-storage register.
- EBX - In 16-bit mode, the base register was useful as a pointer. Now it is completely free for extra storage space.

As an example of how these registers fit together, here is an outline of a typical routine:

```
            mov    esi, source_address
            mov    edi, destination_address
            mov    ecx, loop_count
my_loop:      lodsd

            ;Do some calculations with eax here.

            stosd
            loop   my_loop
```

In this example, ECX is the loop counter, ESI points to the input data, and EDI points to the output data. Some calculations, such as a blur, filter, or perhaps a color look-up occur in the loop using EAX as a variable. This example is a bit simplistic, but hopefully it shows the general idea. A real routine would probably deal with much more complicated data than DWORD's, and would probably involve a bunch of floating-point as well.

In conclusion, using the registers as Intel intended has several advantages. In the fist case, it allows your code to take advantage of many optimizations and special instructions. It also makes the code more readable, since registers perform predictable functions. Finally, using the registers consistently leads to better compression by promoting more repetitive instruction sequences.

Spanish Translation

Ukrainian Translation

Polish Translation