The Intel 80386, part 16: Code walkthrough

devblogs.microsoft.com/oldnewthing/20190210-00

February 11, 2019

Raymond Chen

Let’s put into practice what we’ve learned so far by walking through a simple function and studying its disassembly.

```
#define _lock_str(s)                    _lock(s+_STREAM_LOCKS)
#define _unlock_str(s)                  _unlock(s+_STREAM_LOCKS)

extern FILE _iob[];

int fclose(FILE *stream)
{
    int result = EOF;

    if (stream->_flag & _IOSTRG) {
        stream->_flag = 0;
    } else {
        int index = stream - _iob;
        _lock_str(index);
        result = _fclose_lk(stream);
        _unlock_str(index);
    }

    return result;
}
```

This is a function from the C runtime library, so the functions use the `__cdecl` calling convention. This means that the parameters are pushed right-to-left, and the caller is responsible for cleaning them from the stack.

```
_fclose:
    push    ebx
    push    esi
    push    edi
```

This code was compiled back in the days when frame pointer omission was fashionable. The function does not create a traditional stack frame with the *ebp* register acting as frame pointer.

The 80386 calling convention says that the *ebx*, *esi*, the *edi*, and *ebp* registers must be preserved across the call.

```
    mov     esi,dword ptr [esp+10h] ; esi = stream
```

We will be using the *stream* variable a lot, so we'll load it into a register for convenient access.

```
; int result = EOF;
    mov     edi,0FFFFFFFFh          ; edi = result = EOF
```

The other variable is *result*, which we will keep in the *edi* register, and we set it to its initial value of −1. This is a straight `MOV` instruction, which is five-byte encoding (one opcode byte plus a four-byte immediate). A smaller encoding would have been `or edi, -1`, which uses two bytes for the opcode and one for the 8-bit signed immediate. But the smaller encoding comes at a perfonance cost because it creates a false dependency on the *edi* register. (Mind you, the 80386 did not have out-of-order execution, so dependencies really aren't a factor yet.)

```
; if (stream->_flag & _IOSTRG) {
    test    byte ptr [esi+0Ch],40h  ; is this a string?
    je      not_string              ; N: then need a true flush
```

Even though *_flag* is a 32-bit field, we use a byte test to save code size. This takes advantage of the fact that testing a single bit can be done by testing a single bit in a 32-bit field, or by testing a single bit in an 8-bit subfield. The *_flag* field is at offset `0Ch`, and the value of `_IOSTRG` is `0x40`, so the bit we want is in the first byte.

We learned some time ago that this size optimization defeats the store-to-load forwarder, but the 80386 didn't have a store-to-load forwarder, so that wasn't really a factor.

```
; stream->_flag = 0;
    mov     dword ptr [esi+0Ch],0
```

Again, the compiler chooses a full 32-bit immediate instead of using a smaller instruction. An alternative would have been `and dword ptr [esi+0Ch], 0`, using a sign-extended 8-bit immediate instead of a 32-bit immediate, but at a cost of incurring a read-modify-write rather than simply a write.

```
; return result;
    mov     eax,edi                 ; eax = return value
    pop     edi
    pop     esi
    pop     ebx
    ret
```

The compiler chose to inline the common `return` instruction into this branch of the `if` statement. The value being returned is in the *result* variable, which we had enregistered in the *edi* register. The return value goes in the *eax* register, so we move it there. And then we restore the registers we had saved on the stack and return to the caller. Since this function uses the `__cdecl` calling convention, the function does no stack cleanup; it is the caller's responsibility to clean the stack.

```
    nop
```

This `nop` instruction is padding to bring the next instruction, a jump target, to an address that is a multiple of 16. The 80386 fetches instructions in 16-byte chunks, and putting jump targets at the start of a 16-byte chunk means that all of the fetched bytes are potentially executable.

```
not_string:
; int index = stream - _iob;
    mov     ebx,esi                  ; ebx = stream
    sub     ebx,77E243F0h            ; ebx = stream - _iob (byte offset)
    sar     ebx,5                    ; ebx = stream - _iob (element offset)
```

This sequence of instructions calculates the value for the *index* local variable, which the compiler chose to enregister in the *ebx* register. We start with the value in the *esi* register, which is the *stream* variable. Next, we subtract the offset of the *_iob* variable, which is a global variable, so its address looks like a constant in the code stream. We then take that byte offset and shift it right by 5, which means dividing by 32, which is the size of a *FILE* structure in this particular implementation. The result now sits in the *ebx* register.

```
; _lock_str(index) ⇒ _lock(index+_STREAM_LOCKS)
    add     ebx,19h                  ; add _STREAM_LOCKS
    push    ebx                      ; the sole parameter
    call    _lock                    ; call the function
    add     esp,4                    ; clean stack arguments
```

The *_lock_str* macro is a wrapper around the *_lock* function. We add *STREAM_LOCKS*, which happens to be 25, or `0x19`, and the push it onto the stack as the sole parameter for the *_lock* function. Since this is a `__cdecl` function, it is the caller's responsibility to clean the stack, so we add 4 (the number of bytes of parameters) to the *esp* register to drop them from the stack.

```
; result = _fclose_lk(stream)
    push    esi                      ; the sole parameter
    call    _fclose_lk               ; call the function
    add     esp,4                    ; clean stack arguments
    mov     edi,eax                  ; save in edi = result
```

Another function call: We push the sole parameter, call the function, and clean the stack. The return value was placed in the *eax* register, so we move it into the *edi* register, which we saw represents the *result* variable.

```
; _unlock_str(index) ⇒_unlock(index+_STREAM_LOCKS)
    push     ebx                        ; the sole parameter
    call     _unlock                    ; call the function
    add      esp,4                      ; clean stack arguments
```

The compiler realized it could pull out the common subexpression *s+_STREAM_LOCKS* and stored the value of that subexpression in the *ebx* register. It could therefore push the precomputed value (helpfully saved in the *ebx* register) as the parameter for the *_lock* function.

```
; return result;
    mov      eax,edi                    ; eax = return value
    pop      edi
    pop      esi
    pop      ebx
    ret
```

And this is the same code we saw last time. The return value (*result*) is moved to the *eax* register, which is where the `__cdecl` calling convention places it. We then restore the registers we had saved at entry and return to our caller, leavving our caller to clean the stack parameters.

The resulting function size is 81 bytes.

Okay, now let's see how we could optimize this function further. Let's look closely at the calculation of *index + _STREAM_LOCKS*.

```
    mov      ebx,esi                    ; ebx = stream
    sub      ebx,77E243F0h              ; ebx = stream - _iob (byte offset)
    sar      ebx,5                      ; ebx = stream - _iob (element offset)
    add      ebx,19h                    ; add _STREAM_LOCKS
```

The first thing you might think of is combining the first two instructions into a single `LEA` instruction:

```
    lea      ebx,[esi+881dbc10h]        ; ebx = stream - _iob (byte offset)
```

The `LEA` instruction lets us perform an addition operation in a single instruction by taking advantage of the effective address computation circuitry in the memory unit. The operation we want to perform is subtraction of a constant, which we can transform into an addition of the negative of that constant.

Unfortunately, the trick doesn't work in this case because the "constant" is a relocatable address, and there is no loader fixup type for "negative of the address of a variable."

But all is not lost. There's another trick we could use: Fold in the subsequent addition.

ebx	= ((*esi* − 77E243F0h) >> 5) + 19h
	= ((*esi* − 77E243F0h) >> 5) + (320h >> 5)
	= (*esi* − 77E243F0h + 320h) >> 5
	= (*esi* − 77E240D0h) >> 5

Another way to do this calculation:

`adjusted_index`	`= stream - _iob + 0x19`
	`= stream - (_iob - 0x19)`
	`= stream - &_iob[-0x19]`

Either way, the result is this:

```
    mov     ebx,esi                     ; ebx = stream
    sub     ebx,77E240D0h               ; ebx = stream - &_iob[-0x19] (byte offset)
    sar     ebx,5                       ; ebx = stream - &_iob[-0x19] (element offset)
```

Another observation is that *stream* and *result* do not have overlapping useful lifetimes. The useful lifetime of *result* doesn't start until it receives the value from *_fclose_lk*. Prior to that, its value is known at compile time to be `EOF`, so there's no need to devote a register to it.

And we can combine the `add esp, 4` with the subsequent `push` (which decrements the *esp* register) by simply storing the new value into the top-of-stack slot.

The case of a string-based stream does not use the *ebx* register, so we can use a technique know as *shrink-wrapping*, where we start with one stack frame, and then expand it to a larger one on certain code paths. In this case, we start by saving only the *esi* register, and then later save the *ebx* register only if we realize that we need it.

A simple size/speed optimization (in favor of size) is to use the `pop` instruction to pop a value off the stack (and ignore it). This replaces a three-byte `add esp,4` with a one-byte register `pop`.

A very aggressive size optimization would be to replace the two-byte instructions `mov eax, r` or `mov r, eax` with the one-byte `xchg eax, r` instruction. This assumes you need to move the value into or out of the *eax* register and you don't care about the source any more.

Finally, a string-based stream is quite uncommon (and certainly the case of closing a string-based stream), so we'll make that the out-of-line case, and we won't bother optimizing the fetch of the jump target for the same reason.

```
_fclose:
    push    esi                     ; save register
    mov     esi,dword ptr [esp+0Ch] ; esi = stream
    test    byte ptr [esi+0Ch],40h  ; Is this an _IOSTRG?
    jnz     is_string

    push    ebx                     ; shrink-wrap
    mov     ebx,esi
    sub     ebx,77E240D0h           ; ebx = stream - &_iob[-0x19] (byte offset)
    sar     ebx,5                   ; ebx = index + _STREAM_LOCKS
    push    ebx
    call    _lock                   ; call the function

    mov    [esp],esi                ; parameter for _fclose_lk
    call   _fclose_lk               ; close the stream

    mov    [esp],ebx                ; parameter for _unlock
    mov    ebx,eax                  ; ebx = result
    call   _unlock

    pop    eax                      ; clean the stack once
    mov    eax,ebx                  ; eax = result
    pop    ebx
    pop    esi
    ret

is_string:
    mov    dword ptr [esi+0Ch],0    ; stream->_flag = 0
    or     eax,-1                   ; return EOF
    pop    esi
    ret
```

This reduces the function size to 65 bytes.

Yet another trick is to pre-push the parameters for multiple function calls.

```
_fclose:
    mov     ecx,dword ptr [esp+8]   ; ecx = stream
    test    byte ptr [ecx+0Ch],40h  ; Is this an _IOSTRG?
    jnz     is_string               ; Y: handle strings out of line

    push    ebx                     ; shrink-wrap
    mov     ebx,ecx
    sub     ebx,77E240D0h           ; ebx = stream - &_iob[-0x19] (byte offset)
    sar     ebx,5                   ; ebx = index + _STREAM_LOCKS
    push    ecx                     ; push for _fclose_lk
    push    ebx                     ; push for _lock
    call    _lock                   ; call the function
    pop     eax                     ; discard arg to _lock
    call    _fclose_lk              ; close the stream
    mov     dword ptr [esp],ebx     ; parameter for _unlock
    mov     ebx,eax                 ; save result
    call    _unlock
    pop     eax                     ; discard arg to _unlock
    mov     eax,ebx                 ; recover result
    pop     ebx
    ret

is_string:
    mov    dword ptr [ecx+0Ch],0    ; stream->_flag = 0
    or     eax,-1                   ; return EOF
    ret
```

This brings us down to 57 bytes.

If we abandon the idea of enregistering the *result*, we can do this:

```
_fclose:
    mov     ecx,dword ptr [esp+8]   ; ecx = stream
    test    byte ptr [ecx+0Ch],40h  ; Is this an _IOSTRG?
    jnz     is_string               ; Y: handle strings out of line

    mov     eax,ecx
    sub     eax,77E240D0h           ; ebx = stream - &_iob[-0x19] (byte offset)
    sar     eax,5                   ; ebx = index + _STREAM_LOCKS
    push    ecx                     ; garbage (for future result)
    push    eax                     ; push for _unlock
    push    ecx                     ; push for _fclose_lk
    push    eax                     ; push for _lock
    call    _lock                   ; call the function
    pop     eax                     ; discard arg to _lock
    call    _fclose_lk              ; close the stream
    mov     dword ptr [esp+0Ch],eax ; save result
    pop     eax                     ; discard arg to _lock
    call    _unlock
    pop     eax                     ; discard arg to _unlock
    pop     eax                     ; recover result
    ret

is_string:
    mov    dword ptr [ecx+0Ch],0    ; stream->_flag = 0
    or     eax,-1                   ; return EOF
    ret
```

But this comes out to 59 bytes.

Next time, a bonus chapter on future developments to this architecture.

Raymond Chen

Follow