Intel x86 Assembly Language in Minix

To gain a good understanding of Minix, you need to be able to read its assembly code, split between the files `mpx386.s` (line 5900) and `klib386.s` (line 8100). This handout describes the Intel 386 assembly language used by Minix.

The 80386 chip represents Intel's first serious processor in a long line of microprocessors. It began with the Intel 4004, the first general-purpose microprocessor, a simple 4-bit chip launched in 1971. The Intel 4004 was just meant for calculators (and in 1971, calculators weren't powerful devices). But it was a start, if a humble one. The 8-bit Intel 8080 (1974) was the CPU for the first successful personal computer, the Altair. And then the 16-bit Intel 8088 (1979) became the CPU for the first IBM PC. Subsequent generations became new CPUs for IBM PCs and their clones. But the 386 stands out among them, as the first 32-bit processor, and the first Intel processor to provide hardware support for the features needed for real operating systems (such as large memories, protection levels, paging). Subsequent processors in the family have provided enhanced performance more often than enhanced capabilities.

But here we're concerned with the Intel 80386 architecture, used in Minix. There's a variety of x86 assembly languages. We'll use the language of *NASM*, an open-source assembler that is widely available and is pretty close to what Minix uses.

1 Registers

The first thing to learning a processor is its register structure. See Figure 1.

This structure is pretty messy. In particular, the registers overlap. The eax register holds 32 bits, but you can refer to its lower 16 bits using ax, and within ax you can refer to its higher 8 bits with ah and the lower 8 bits with al. This weird structure is an outgrowth of the fact that the x86 design began with an 8-bit design, then grew into 16 bits, then into 32 bits, each time maintaining backward compatibility.

The left-hand block, from eax to esp, represents the *general registers*, where computation is meant to take place. Most have some special properties, in the sense that some instructions treat them in a special way. For example, the push and pop instructions treat esp (meant to be the stack pointer) specially, in that they access memory relative to the address stored in esp and decrement and increment the value stored in esp. We'll see how the other registers are special when we get to those particular instructions.

The eip register is the instruction pointer (often called the program counter). The eflags register holds various flags that are altered or accessed by instructions.

The last block of six registers, cs through gs, are the *segment registers*, used for memory addressing. You don't really need to know much about them right now — but we'll get to them eventually.

2 Memory

Like most other computers, the x86 architecture regularly accesses memory off the chip in order to fetch instructions and manipulate data that doesn't fit into the registers.

The x86 architecture uses the *little endian* technique for storing data of multiple bytes. Say we store the 16-bit value 0x1234 into memory at address 100. The value 0x12 goes into address 101 and the value 0x34 goes into address 100. Storing the 32-bit value 0x12345678 into address 100 would place 0x12 into address 103 and 0x78 into address 100.

Many other architectures use the *big endian* technique, which puts the highest-order bits into the lowest address. They both work well. Which you like depends on whether you picture memory with address 0 at the bottom or with address 0 at the top. Both ways make sense, but the x86 designers had to choose one, and they clearly thought of address 0 being at the bottom.

[0]Copyright ©2001, Carl Burch (Saint John's University, Collegeville MN 56321, `cburch@csbsju.edu`). May not be redistributed without the author's explicit permission.

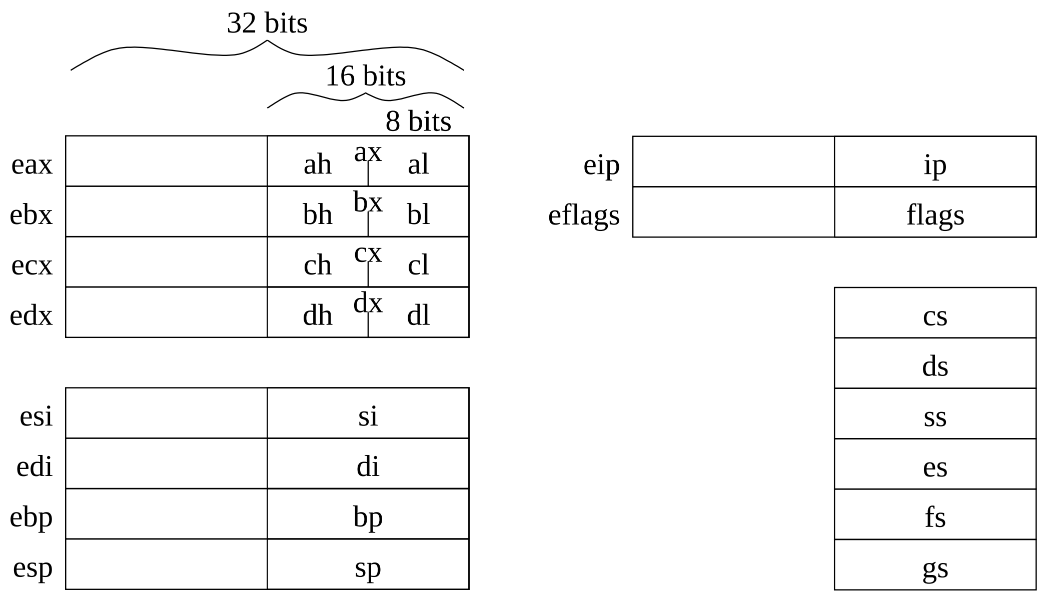

Figure 1: Registers in the Intel 80386.

add $r/m, r/imm$	sub $r/m, r/imm$
add r, m	sub r, m
neg r/m	cmp $r, r/m$
inc r/m	cmp m, r
dec r/m	cmp $r/m, imm$

Table 1: Arithmetic instructions

3 Arithmetic instructions

The first set of instructions we'll examine are the arithmetic instructions, listed in Table 1. In these tables, we use r to represent a register, m to represent a memory reference, and imm to represent a constant value (immediate).

add
sub

The add and sub instructions are for performing additions and subtractions. They have a variety of ways in which they can be used. The following illustrates them.

```
add eax, 1                ; add constant 1 to register eax
add eax, eax              ; add value in eax to eax
add eax, [data]           ; add value at memory location [data] to eax
add [data], 4             ; add constant 4 to memory location [data]
add [data], eax           ; add value in eax to memory location [data]
```

In each of these, the item changed is listed first, and the number to be added is listed second. The instruction add eax, 1 increments the value in eax. And the instruction add eax, eax adds the value of eax to eax, effectively doubling eax.

Enclose memory references in brackets.[1] So the instruction add eax, [data] says to load the data stored at the memory address that data represents, and add that value to eax.

[1]You'll find that the Minix code reverses parentheses and brackets. Parentheses are used for memory references, and brackets are used for parenthesizing computations to be done at assembly time. Another difference is that Minix uses exclamation points to mark comments, while NASM uses semicolons.

Both add and sub change flags in the eflags register based on the result, which is useful when more information is needed later about the computation's result.

carry flag is set if the computation carried an extra bit beyond the capacity of the register, or if the computation borrowed a bit beyond the capacity of the register.

zero flag is set if the result is zero.

sign flag is set if the result's high-order bit is 1 (representing a negative number).

overflow flag is set if the computation, interpreted as a signed computation, goes beyond the capacity of the register. (This is analogous to the carry flag, but the carry flag represents the computation interpreted as an unsigned computation.)

The cmp instruction is for comparing two values. It's actually analogous to the sub instruction, but cmp does not change the value of the target register. Its only effect is to change the flags according to the difference of the two values. cmp

The neg instruction negates the target value (either a register or memory location). (In the rare case that the value doesn't fit — which only happens if the value is the smallest number possible for its bit size — the value is unchanged and the overflow flag is set.) neg

Finally, the inc and dec instructions increment or decrement the given value. Why use inc eax instead of add eax, 1? A minor reason is that it is slightly more efficient, since the instruction is just one byte long instead of three (leading to better cache use). (On older processors, it actually was faster.) Also, the inc and dec instruction do not affect the carry flag, which may be useful in some situations. The most common reason is that it's just a little easier to read. inc dec

Minix occasionally appends *b* to the end of an instruction name, as in incb, decb, and cmpb. (This also applies to many instructions described later in this document.) The appended *b* denotes that the instruction is a byte instruction (using registers like al or ch) instead of the regular 32-bit instructions normally used. You'll also see o16 in the Minix assembly code, used as a prefix for many instructions. This is to indicate that the instruction should be assembled into an instruction using 16-bit data. incb decb cmpb o16

In NASM, the way to accomplish this sort of thing is to annotate the operation with a keyword: byte for 8-bit data, word for 16-bit data, or dword for 32-bit data. (The last is short for *double word*. It refers to the fact that the x86 word is 16 bits long, so 32-bit data is two words long.)

```
add [data], 4
```

This instruction is ambiguous: Does data hold an 8-bit value, a 16-bit value, or a 32-bit value? NASM will refuse to assemble it. We include the keyword dword to disambiguate.

```
add dword [data], 4
```

Notice that add eax, 4 isn't ambiguous, since the assembler knows that eax is a 32-bit location.

4 Control instructions

Of the control instructions (Table 2), the nop instruction is certainly the simplest: It does nothing. It just occupies a byte in memory. It's convenient occasionally; for example, a compiler might put in a nop as a placeholder if it thinks it's possible that some code should go there, but it's not sure at the time it generates the code. nop

The jmp instruction transfers execution to another place in the program. Notice that you can jump to the address in a register or a memory location; or you can jump to a label within the program. In the case of a label, the label is assembled into an offset. jmp

nop	jmp $r/m/label$
jz $label$	jnz $label$
ja $label$	jg $label$
jae $label$	jge $label$
jb $label$	jl $label$
jbe $label$	jle $label$
call $r/m/label$	ret
loop $label$	

Table 2: Control instructions

```
again:      inc eax
            jmp again
```

This program is an infinite loop with no real purpose. But the point is that the jmp instruction gets assembled as jmp -1, signifying that the CPU should subtract 1 from the program counter (eip). This means the CPU backs up one byte to the inc eax instruction to execute it again.

The next 10 instructions give ways of doing jumps conditionally. The usual assumption is that you've just completed a cmp instruction (or some other instruction that sets the flags), and now you want to jump based on the values in the flags. The jz instruction jumps if the zero flag is set. In particular, if the last instruction setting the flags was a cmp instruction, the zero flag would be set if the two arguments were equal. Similarly, jnz jumps if the zero flag is not set — which in the same sense corresponds to unequal arguments in a preceding cmp instruction.

jz

jnz

As an example, here's a piece of code to take the value n in cl and place the value 2^n in eax.

```
            mov eax, 1          ; eax will hold answer
            cmp cl, 0           ; cl is number of iterations to go
            jz done
again:      add eax, eax        ; double eax
            dec cl
            jnz again
done:
```

The first instruction places the value 1 in register eax. (We haven't seen the mov instruction yet — it's coming.) The next two instructions skip everything else if cl is zero (and so we complete with eax holding $2^0 = 1$). The add instruction doubles the value in eax, and the dec instruction subtracts 1 from cl. Finally, the jnz instruction jumps back to again if the zero flag is not set — that is, if the result of the decrement is not zero. If the zero flag is zero, then control continues to the done label.

ja

The ja instruction stands for "*j*ump if *a*bove". It jumps to the specified location if both the zero bit and the carry bit are zero. (Consider a cmp instruction where we understand both the arguments to be unsigned numbers. Then it sets the zero bit if the values are equal and the carry bit if the first number is below the second; if neither of these are one, then the first number is above the second. So this is when the ja instruction will jump.) The next three instructions — jae (above or equal), jb (below), and jbe (below or equal) — work analogously.

jae
jb
jbe

The last 4 conditional jumps are for signed computation, using the overflow flag, the sign flag, and the zero flag to determine whether to jump: jg (greater), jge (greater or equal), jl (less), and jle (less or equal). The intention of them is identical to the previous 4, except that the previous four were for unsigned arithmetic.

jg
jge
jl
jle

The loop instruction is for running through a loop for a fixed number of times, as in our earlier program to compute 2^n, where we knew we wanted to iterate exactly n times. The loop instruction subtracts 1 from

loop

mov $r/m, r/imm$	mov r, m
mov $r/seg, r/seg$	movzx $r, r/m$
lea r, m	xchg $r, r/m$
push $r/m/seg/imm$	pop $r/m/seg$
pushad	popad
pushf	popf
out r/imm	in r/imm

Table 3: Data movement instructions

ecx and jumps to the label if ecx is not zero. Notice that loop *always* uses ecx for this purpose. The ecx register has this special meaning to the loop instruction. (The x86 designers thought of the *c* as standing for *count*.)

We could use loop to rewrite our earlier program to get a slightly shorter program.

```
                mov eax, 1
                cmp ecx, 0
                jz done
again:          add eax, eax
                loop again
done:
```

Notice that we had to switch to using ecx in place of cl here, since loop works with ecx.

call
ret

Finally, the call and ret instructions provide support for subroutines (frequently called functions or procedures in programming languages). A call instruction has two effects: First, eip is pushed onto the stack. (This is itself a two-step process: esp is decreased by 4, and then the value of eip is stored in the memory where esp now points.) Then control goes to the location specified in the instruction. The ret instruction is for returning from the subroutine, and here eip is popped from the stack. (That is, the value stored at the location to which esp points is copied into eip, and then esp is increased by 4). In effect, this restores the previous value of eip, so that the next instruction executed is the instruction following the call that put us there.

That's all a little complicated, but it's worth some time figuring it out. Luckily, in practice it's relatively simple. Here's a very simple subroutine for doubling the number in eax.

```
double:         add eax, eax
                ret
```

Notice that we use ret to return from the subroutine. Now we can use this as we like.

```
                mov eax, 1
                cmp cl, 0
                jz done
again:          call double
                loop again
done:
```

In this case, the subroutine is contrived — it's too simple to be useful —, but it illustrates how you can write a subroutine in x86 assembly language.

5 Data movement instructions

Table 3 lists the x86 assembly instructions useful for moving data between different areas of the computer. The most fundamental of these is the mov instruction, which moves a value into another location. (The

and r/m, r/imm	or r/m, r/imm
test r/m, r/imm	xor r/m, r/imm
shl r/m, r/imm	shr r/m r/imm
rol r/m, r/imm	ror r/m, r/imm

Table 4: Logical instructions

word *move* here is slightly misleading: It actually *copies*, leaving the old value in its location.) Note that mov does not alter any flags.

mov A mov instruction can move a number, a register value, or a value in memory into either a register or a memory location, except that a single mov cannot copy directly from one memory location to another. Additionally, a mov can copy between a segment registers and a general register — this gives you a way of accessing these segment registers when needed. (When we talk more about memory, we'll talk about why these segment registers are important. For now, you should still be ignoring them.)

movzx The movzx instruction is for moving a shorter value into a longer destination. For example, if you want to copy bl into eax, you can use the instruction movzx eax, bl. This copies bl into al, extending it with zeroes into the higher bits of eax.

lea Actually, the lea (*l*oad *e*ffective *a*ddress) instruction doesn't access memory at all. It stores the *address* of the given memory location into the given register. This is intended for when you want to compute an address that is more complex than the built-in x86 addressing can handle. (For example, you may want lea to access an element of an array stored within a structure located within an array of structures.) For example, the instruction lea ecx, [1 + ecx] increments ecx — it's functionally equivalent to inc ecx, except that no flags are altered.

xchg The xchg instruction exchanges the data in two locations, of which one must be a register.

push
pop The push and pop instructions add and remove from the stack. If the computer encountered a push ax instruction, it would decrease esp by 2 (choosing 2 because ax is two bytes long) and then store ax at the memory pointed to by esp. The instruction pop cl would load one byte from the memory pointed to by esp into cl and then increase esp by 1.

pushad
popad
pushf
popf The pushad and popad instructions are for saving several the general registers at once: pushad pushes eax, ecx, edx, ebx, esp, ebp, esi (in that order), and edi, and popad pops them in reverse order. Similarly, the pushf and popf instructions are for saving and restoring the value of the flags register.

in
out
inb
outb The in and out instructions are a way of communicating with many I/O devices. The idea is that an I/O device can be referenced by a numbered port, specified in the in or out instruction (either directly with a constant value or indirectly by a register value). An in instruction reads the value at that port and copies it into the ax register (al in the case of inb). An out instruction takes the value in ax (al in the case of outb) and copies it out to the port given in the instruction.

For example, the keyboard's port number is 96. So by executing inb 96, the code for a pressed key will be loaded into al. (Actually, controlling the keyboard is complicated by the fact that the CPU then signals receipt of that code back to the keyboard (via an outb 97 instruction).)

Not all I/O devices work this way, but many do. Other devices work via *direct memory access* (DMA). A graphical display is an example of a device that uses DMA. It shares memory with the CPU, with the CPU writing the image directly to memory, and the video controller reading directly from memory to determine what to display. The DMA technique is designed for devices that need to move large masses of data around quickly.

int *imm*	iret
cli	sti

Table 5: Interrupt instructions

6 Logical instructions

The next set of instructions are the logical instructions, listed in Table 4. These instructions work like the arithmetic instructions, in that they compute new values and arrange the flags. We've saved them to later because they're a little less useful. (But they're still too useful to be ignored!)

and
or

The and and or instructions perform the bitwise AND or OR, placing the result in the destination register and setting the flags. For example, say al held the value 11110010; then the instruction and al, 0xA3 would take the AND of each corresponding pair of bits and put the result (10100010) back into al. It would also set the flags: in this case, the top bit is 1, so the sign flag would be set to 1; the result isn't zero, so the result flag would be reset to 0; and the carry and overflow flags would be reset to 0, since there is no carry and no overflow (not even the possibility). The or instruction is identical.

test

The test instruction performs a bitwise AND and throws the result away. It's relationship to and is the same as the relationship of cmp to sub. It's useful in two situations. First, it's useful for testing whether a particular bit is set in a register (test al, 0x80 followed by a jnz will jump if the top bit of al is set). It's also useful when you just want to see whether a register is zero: test eax, eax is more efficient (a shorter instruction) than cmp eax, 0.

xor

The xor instruction works like and and or. But it's rare you want to do the exclusive OR of two things. Yet it occurs quite often in assembly code for quite another reason: If you take the XOR of something with itself, you get 0. So the instruction xor eax, eax actually puts 0 into eax. This instruction is shorter and hence more efficient than mov eax, 0, so x86 programmers use the xor idiom when they want to put 0 into a location.

shl
shr

The last four instructions are instructions for shifting values. The first two, shl and shr, are the most elementary: They shift the value in the destination location left (or right). Zeroes are shifted into the empty places, and the last bit shifted out goes into the carry flag.

rol
ror

The next two, rol and ror, rotate the bits. For example, rol al, 1 will shift every bit in al left one spot, with the highest-order bit being rotated around into the lowest bit of the register. (Again, the carry flag also comes to hold this last bit rotated off the end of the register.) Note that rol al, 8 effectively does nothing to al, as all the bits are rotated back into their original places. Of course, the carry flag changes to the last bit shifted off, in this case the uppermost bit. (But using the sign flag after test al, al is a more efficient way to accomplish the same effect.)

7 Interrupts

The x86 CPU allows for *interrupts.* These are calls to operating system code, usually occurring in response to hardware events that need immediate attention. But a program can initiate its own interrupt, too, to transfer control to the operating system. On the x86 CPU, the interrupts are numbered. For example, interrupt 0x0C occurs when there is new data coming in from the COM1 serial port on the computer (commonly used for modems). Interrupt 0x0E is for the LPT1 port (commonly used for printers).

During the boot process, the operating system initializes a table telling the CPU where to find the first instruction for each of the possible interrupts. Then, when an interrupt occurs, the CPU pushes the current value of ip and cs onto the stack and transfers control to the first instruction of the interrupt.

But programs can initiate their own interrupts too. Why would you ever use an interrupt in a program? Why not just use call to transfer control? There are two main reasons. First, the program may want to transfer control to the operating system to take advantage of some program installed there. If the operating system was to require a call instruction to do this, each program would need to determine where the first instruction of the called routine is. By using an interrupt instead, the software can use the interrupt table set up by the operating system.

Second, and more significantly, an operating system must restrict the user programs in order to protect programs from crashing the computer. Of course, that restriction much less useless if the user program can arbitrarily rescind the permissions. On the other hand, programs need to access devices, which require more permission. So the operating system provides these routines for programs' convenience. But there needs to be some way to transfer control into the operating system and simultaneously gaining more control while in this code. An interrupt accomplishes both of these goals simultaneously.

int
iret

At any rate, the int instruction initiates an interrupt. The argument to the instruction names which of the interrupts to use. The iret instruction is used within the interrupt handler to return from the interrupt. Returning from the interrupt pops the ip and cs values from the stack, effectively transferring control back to what was executing before the interrupt occurred.

cli
sti

The cli and sti instructions affect the interrupt flag in the flags register. When this flag is 0, no interrupts are accepted — they are buffered (and of course the chip's buffer is limited). When it is 1, interrupts are enabled, meaning that the hardware can interrupt the CPU at any time. The cli instruction clears the interrupt flag, cutting off the interrupts. The sti instruction sets the interrupt flag, enabling future interrupts.

The reason for these two instructions is that occasionally there is a sequence of instructions that must be executed without interruption. Such a sequence is called a *critical section.* This often happens within an interrupt handler, where a hardware interrupt is in the midst of some computation that should not be altered by another interrupt.

In fact, an interrupt automatically clears the interrupt flag, and the iret instruction automatically sets the interrupt flag. But you'll see that interrupt handlers often use sti early on, to re-enable interrupt handlers before the buffer fills up. Or they may use cli because they are entering a second critical section.

8 Segment registers

The 8086 used 16-bit registers. Unfortunately, 16-bit memory addresses only support 2^{16} bytes (64KB). Even at the time of the 8086, that was obviously limited. So memory addresses used a second register — the *segment registers* — to work around this.

In the 8086, a memory address was computed as follows: Take the segment register, shift it left four bits, and add the address to this. By doing this, a program could access up to 2^{20} bytes (1MB), which was considered ridiculously large when the 8086 was designed.

By the time of the 80386, this limit was very constricting. It led to the infamous 640KB restriction (since the top 384K of the address space were reserved for the hardware), and it took a lot of work to fit everything into 640K as programs got larger and more powerful. So they went to another scheme.

The 80386 can run in two modes. The first is the 8086 emulation mode, called *real mode*. But the more powerful mode is *protected mode*. When it runs in protected mode, a segment register is treated differently. In this mode, the segment register specifies a segment number, which gets translated to some 32-bit base address. Then the address given in the instruction is added to this base address to get the actual address accessed. This scheme gives access to 2^{32} bytes (4GB), which was considered ridiculously large when the 80386 was designed (and is still large enough today).

You can ignore the segment registers most of the time in your assembly programs, though, because the operating system sets them up so that the instructions do the right thing. Instructions, for example, is loaded relative to the contents of the cs (code segment) register. And data is typically loaded relative to the ds (data segment) register. The operating system sets up these segment registers so that the program works properly.

An instruction can override the default behavior, however, when it specifies an address. It can write mov eax, [es:bx] to load from the bx offset from the segment specified in es.

movs	stos
outs	ins
cld	std
rep	

Table 6: Sequence instructions

9 Sequence instructions

The sequence instructions (Table 6) are for manipulating fragments of memory.

movs

The most useful among these is the movs instruction, for copying one memory fragment into another. The movs instruction copies four bytes of memory, from [esi] to [edi]. (The esi and edi registers are designed for this purpose: The names stand for *source index* and *destination index*, respectively.) The movs instruction also increments the esi and edi registers by four, so that a subsequent movs will copy the next four bytes of memory.

rep

This gets really interesting when you use the rep instruction in conjunction with it. The rep instruction is peculiar in that it is an optional prefix to several instructions. For example, if rep occurs before movs, then it will repeatedly execute movs and decrement ecx, until ecx reaches zero. For example, we might write this.

```
                rep
                movs
```

Or we might write this.

```
                test ecx, ecx
                jz done
again:          mov eax, [esi]          ; here we're simulating movs
                mov [edi], eax
                add esi, 4
                add edi, 4              ; this is the last instruction for movs
                dec ecx
                jnz again
done:
```

The first is much more efficient, and it's easier to type. Of course it isn't necessarily what we want, so some variation on the second is often more appropriate. Naturally, the movsb instruction works the same, except at the byte level. With movsb, esi and edi are incremented by just 1 to do a proper string copy.

movsb

stos

The stos instruction is for saving several copies of the same bits into an array. It stores the value in eax into the memory pointed to by edi and then increments edi by 4. The rep prefix can be prepended to this to store many copies of the same data very quickly.

outs
ins

The outs and ins instructions send and receive data from a port specified in the edx register. The outs instruction takes the 4 bytes pointed to by esi, sends it to the port mentioned in edx, and increments edx by 4. The ins instruction reads 4 bytes from the port mentioned in edx, stores it in the 4 bytes pointed to by edi, and increments edx by 4. The rep prefix can be prepended to either of these instructions.

cld
std

The final two sequence-manipulation instructions we'll look at are cld and std. These are for clearing and setting the direction flag in the eflags register. In all of the above instructions, when the direction flag is one, the values in esi and edi are *decremented* instead of incremented. This facilitates the case when an array is stored in reverse order.

Minix generally keeps the direction flag cleared. The only situation in which it sets the direction flag is for copying a memory region from one location to another. If the source region and the destination region

lgdt	lldt
lidt	ltr
sgdt	

Table 7: Advanced feature instructions

global	section
db	dw
dd	equ
times	

Table 8: Assembler directives

overlap, and the source occurs at a lower address, then going upward would alter some of the source region before it is copied. So Minix does a backwards copy in this situation.

10 Advanced features

The 80386 CPU and its successors support several advanced operating systems features. For example, the CPU supplies protection levels to prevent one process from accessing the memory of another. And it also allows the operating system to allocate a different memory address space for each running process.

lgdt lidt lldt ltr sgdt

The final set of instructions (Table 7) allow you to access these features. The lgdt, lidt, lldt, and ltr instructions change the global descriptor table, the local descriptor table, the interrupt descriptor table, and the task register, respectively. The sgdt gives a way of reading the current global descriptor table.

We're not going to explore these instructions in detail now. Section 4.6.3 of the textbook describes the local descriptor table and the global descriptor table.

11 Extended example

Figure 2 contains an extended example of a function written in x86 assembler. What it does is to take an array of 16-bit values and to sort them into increasing order.

It uses a particularly simple technique, the *selection sort algorithm.* It works by first finding the largest value in the array, and then you swap it with the last position in the array. Then you find the largest value before the last position, and you swap it with the next to last position. Then the largest value before than position, and you swap it with the previous position. You continue doing this until you have only one element left.

12 Directives

A complete program needs additional data to tell it how to generate its output. These are the *directives*: They're not instructions, but they direct the assembler to do something.

global

For example, the global directive tells the assembler about a label that should be made available outside the program. If you had a large assembly program split across several files, you would use global to list the labels of the subroutines that are accessed from other files. We'll use global to make the special label _start publicly available, as _start is the label that the operating system will call to start a program.

section

The section directive tells the assembler that we're starting a new section of the program. Normally, the program should be split between the .data section (where data is stored) and the .text section (where instructions are stored).

```
; sort
; sorts an array of 16-bit words into ascending order
;
; when called, the stack should hold:
; esp + 0: return address
; esp + 4: number of words in array
; esp + 8: address of first word of array
sort:
            pushad
            mov esi, [esp + 40]         ; esi is address of first element of array
            mov ebx, [esp + 36]
            dec ebx                     ; ebx is index where to place maximum
            jz done

outer:
            mov edx, ebx                ; edx will be index of maximum in [0..ebx]
            mov ax, [esi + 2 * ebx]     ; ax will be maximum in [0..ebx]
            mov ecx, ebx                ; ecx is number of indices left to consider

inner:
            cmp [esi + 2 * ecx - 2], ax
            jbe innernext               ; if [ecx-1] > al,
            mov edx, ecx                ; we’ve found a new max, so
            dec edx                     ; update edx (index of maximum)
            mov ax, [esi + 2 * edx]     ; and update ax (maximum value)
innernext:
            loop inner                  ; we’ve considered this one; continue

            xchg ax, [esi + 2 * ebx]    ; swap [edx] with [ebx] (ax held [edx])
            mov [esi + 2 * edx], ax
            dec ebx                     ; now go onto the next ebx
            jnz outer

done:
            popad
            ret
```

Figure 2: An x86 subroutine to perform selection sort.

```
                global _start

                section .data
array           dw 23, 12, 45, 34, 67       ; byte array to be sorted
ARRLEN          equ 5                       ; number of bytes in array

                section .text
_start:
                push dword array
                push dword ARRLEN
                call sort

                mov ax, 4                   ; make interrupt for write system call
                mov ebx, 1                  ; (arg 1: file descriptor where to write)
                mov ecx, array              ; (arg 2: memory location where to find data)
                mov edx, 2*ARRLEN           ; (arg 3: number of characters to print)
                int 0x80

                mov ax, 1                   ; make interrupt to exit program
                mov ebx, 0                  ; (arg 1: return code)
                int 0x80

; insert the code for sort here (Figure 2)
```

Figure 3: A complete program to sort.

db
dw
dd

The db, dw, and dd directives tell the assembler to allocate some data into the output. (Respectively, they allocate bytes, words, and double words.) This typically occurs in the .data section, and there's usually a label before it.

```
array           dw 23, 12, 45
```

In this example, we've told the assembler to place the three 16-bit values 23, 12, and 45 into the generated object file. The array label gives us a way of specifying the address of the first of these words. Consider now the following instruction to access the third element of array.

```
                mov ax, [array + 4]
```

This instruction loads the value 45 into ax. (Or, if the memory has been changed, whatever has been written over the 45.) We add 4 to array because we want to skip over the first two elements of array, and each array element is two bytes long.

equu

The equ directive defines a constant. There must be a label for this directive. The assembler will replace each occurrence of the label with the value defined for the label.

times

Finally, the times directive tells the assembler to do something several times.

```
array           times 1024 db 0
```

This will accomplish the db 0 directive 1024 times, reserving 1024 bytes of memory initialized to 0.

Figure 3 contains a complete program, including all the necessary directives. You can place this into a text file and run the assembler on it to generate a program. This program is Linux-specific, as it makes system calls to `write()` and `exit()`. This program sorts an array of 5 values and prints them to the output. It prints binary codes, so it won't make sense on an ASCII terminal.