The 80386SX Processor Bus and Real-Mode Instruction Set

This chapter describes the signals and operation of the Intel 80386SX processor bus and describes a subset *of the 80x86 architecture and instruction set.*
After this lecture you should be able to draw diagrams for the processor bus signals described in this lecture and state the values that would appear on the data and address buses during memory and I/O read and write cycles and for interrupt acknowledge cycles.
While it's not possible to cover all the details of the 80x86 instruction set you should learn enough about it to be able to write simple routines to service interrupts and read/write data to/from I/O ports. In particular, you should be able to:

- *write a real-mode 8086 assembly language program including: (1) transfer of 8 and 16-bit data between registers and memory using register, immediate, direct, and register indirect addressing, (2) some essential arithmetic and logic instructions on byte and 16-bit values, (3) stack push/pop, (4) input/output, (5) conditional and unconditional branches, (6) call/return, (7) interrupt/return, (8) essential pseudo-ops (org, db, dw).*
- *compute a physical address from segment and offset values,*
- *describe response of the 8086 CPU to software (INT) and external (NMI, IRQ) interrupts and return from interrupts.*

History

Intel's first 16-bit CPU was the 8086. A version of the 8086 that used an 8-bit data bus, the 8088, was released later to permit lower-cost designs. The 8088 was used in the very popular IBM PC and many later compatible machines.

Intel's first 32-bit CPU was the 80386. It was designed to be backwards-compatible with the large amount of software which was available for the 8086. The 80386 extended the data and address registers to 32 bits. The Intel '386 also included a sophisticated memory management architecture that allowed *virtual memory* and *memory protection* to be implemented. This same basic 80386 architecture is used in the latest generation of Pentium and compatible processors.

This lecture describes the processor bus of the Intel 386SX, a version of the 386 with a 16-bit processor bus. The 386EX, the chip that we will use in the lab, is similar to the 386SX but also integrates several commonly-used peripherals in the same chip.

Processor Model	Register Width	Data Bus Width	Address Bus Width
8086	16	16	20
8088	16	8	20
i386	32	32	32
i386SX	32	16	24
i386EX	32	16	24
Pentium	32	64	32

kilo- Mega- and Giga-Bytes

It is common in talking about powers of two (e.g. memory sizes) to use the suffixes *kilo Mega,* and *Giga* although the values are somewhat (about 2%) larger than the corresponding powers of ten. Express powers of two using a value from the first column below and a suffix from the second column.

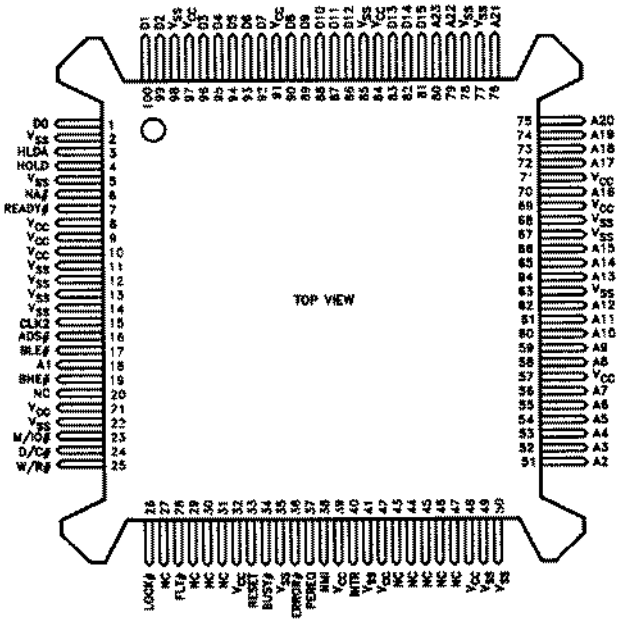
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512

2^{10}	1024^1	kilo
2^{20}	1024^2	Mega
2^{30}	1024^3	Giga

Exercise 21: How much memory can be addressed by the 20, 24 and 32-bit addresses used by the different 80x86 address buses?

'386SX CPU Signals

The 386SX is packaged in a 100-pin package. It has a 24-bit address bus and a 16-bit data bus. The names of the signals are shown below. Active-low signal names are suffixed with '#' (i.e. BHE# $\equiv$ BHE* $\equiv$ $\overline{\text{BHE}}$)



Utility Bus

The utility bus includes the pins that are required for the processor to operate properly but which are not involved in data transfers. This includes the power, ground, and clock pins.

The two most important pins on the CPU chip are for power supply ($V_{cc} = 3.3$ or 5 V) and ground(V_{ss}). The processor will operate erratically (or not at all) if the power supply is not held at the proper voltage.

The next most important signal is the clock, CLK2. Output signal transitions happen immediately after the rising edge of CLK2 and inputs are sampled on the rising edge of CLK2.

Figure 1 shows examples of data transfers over the processor bus. Each transfer (read or write) is called a *bus cycle*. Each bus cycle requires two or more *processor cycles* (one T1 cycle plus one or more T2 cycles). Each of these processor cycles requires two CLK2 periods. Figure 1[1] shows how two CLK2 cycles make up a processor cycle and how two processor cycles (T1 and T2) make up a bus cycles.

Cycle		Requires	
processor cycle	2	CLK2 cycles	T1, T2
bus cycle	>= 2	processor cycles	read, write, ...

Exercise 22: A 386SX CPU is operating with a 25 MHz CLK2 signal. What is the CLK2 period? How long does a processor cycle take? How long does a bus cycle take?

Address and Data Busses

The 80386SX has a 16-bit data bus and a 24-bit address bus. These signals are labelled D_{15} to D_0 and the A_{23} to A_1 (*not* A_0) respectively. To allow for either 8-bit or 16-bit transfers the chip uses BHE* and BLE* (high- and low-byte enable) signals indicate to memory and I/O devices which byte(s) is/are being transferred. The BHE* indicates a transfer over D_{15} to D_8 and BLE* indicates a transfer over D_7 to D_0.

BHE* and BLE* also indicate the memory address being accessed: BLE* and BHE* indicate addresses with $A_0 = 0$ and $A_0 = 1$ respectively.

Unlike the Motorola 68000, this intel processor allows 16-bit values to be written to odd addresses and 32-bit values to be written to addresses that are not multiples of 4 (i.e. memory operations do *not* have be *word-aligned*). Thus the value transfered over the high-order byte of the data bus may or may not correspond to the high-order byte of the value being written.

[1] From Intel i386SX data sheet.

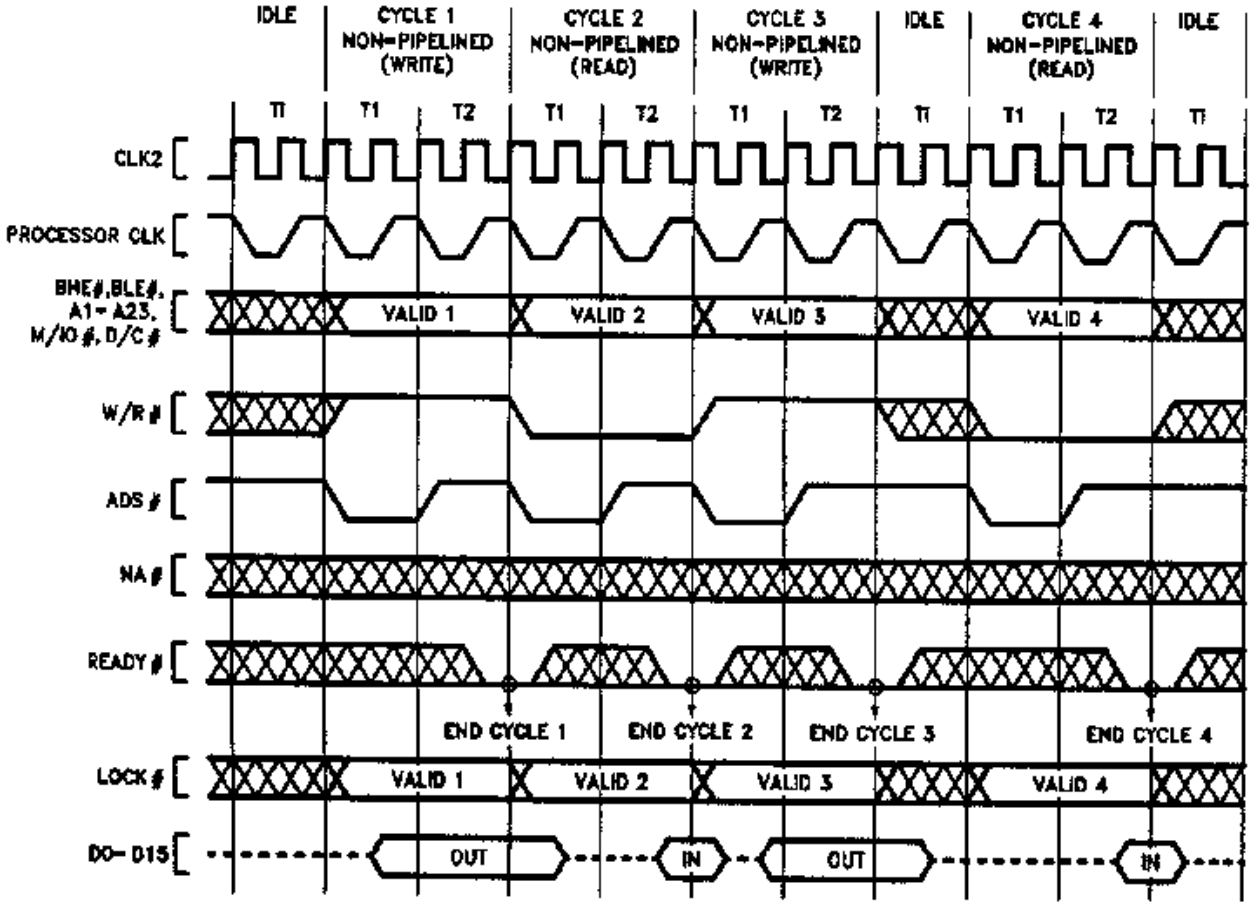


Figure 1: Examples of 80386SX Bus Cycles

Thus BHE* and BLE* indicate the address(es) being read or written — *not* the high- or low-order bytes of a word.

Exercise 23: What signals does the Motorola 68000 use for this purpose? How are they different?

Endianness

Intel processors, unlike Motorola processors, use so-called "little-endian" byte order. This means that 16- or 32-bit words are stored with the *least-significant* byte at the lowest-numbered address. This can be confusing. We normally write memory contents in increasing address order from left to right; in little-endian storage order the bytes in multi-byte words appear in reverse order.

Exercise 24: The 16-bit word 1234H is to be written to address 1FFH by an Intel processor. What value will be stored at memory location 1FFH? At which address will the other byte be stored? Write your answer in the form of a table showing the final memory contents:

Address	Data

Which byte enable(s) will need to be asserted to store these values? How many bus cycles will be required? Write out your answers in the form of a table showing the values of the address bus in binary, the values on the data bus in hex, and the values of BHE* and BLE* (H or L) for each bus cycle.

Address	Data Bus	BHE*	BLE*
0001 1111 111x			
0010 0000 000x			

What if the value 12345678H was to be stored at the same address? What if the 16- and 32-bit values were written to address 100H?

Processors with wider address buses such at the ’386 or Pentium require more bus enable signals (BE0*–BE3* or BE0*–BE7*).

Memory and I/O Address Spaces

The Motorola 68000 processors use conventional memory read and write (MOVE) operations to do input and output. Peripheral interfaces appear to the processor as if they were memory locations.

The 80x86 processors can also use this type of “memory-mapped” I/O but they also have available special instructions (IN and OUT) for I/O operations. A bus signal (M/IO*) indicates whether a bus cycle is due to a memory or an I/O instruction. These special I/O instructions allow more flexibility in the design of interfaces (e.g. extended cycles for I/O operations). I/O operations can only be done on the first 64kB of the I/O address space.

On the IBM PC and compatibles only the first 1k of this I/O address space is available (0 to 3FFH).

Bus Control

In order to accommodate slow memory and I/O devices the intel 80x86 processor buses use a READY* input. If the READY* input is not asserted at the end of a T2 processor cycle the 80386SX will generate additional T2 cycle(s) (see below).

Exercise 25: What signal does the Motorola 68000 use to extend bus cycles?

The W/R* (write/read), D/C* (data/control), and M/IO* (memory/I/O) output signals indicate the type of bus cycle being executed (read, interrupt acknowledge or write). The table below shows the possible bus cycles:

D/C*	M/IO*	W/R*	Bus Cycle
H	H	L	memory read
H	H	H	memory write
H	L	L	I/O read
H	L	H	I/O write
L	H	L	instruction fetch
L	L	L	interrupt acknowledge
L	H	H	halt

Another processor bus signal, ADS*, indicates that the contents of the address bus and the three signals above are valid.

Exercise 26: What are the equivalent signals on the Motorola 68000 processor bus?

Reset and Interrupts

As you might suspect, the RESET input resets the processor. The CPU register contents are reset and the program counter is set so that the CPU will fetch the next instruction from memory location FFFFF0. The memory at this location must therefore contain instructions to restart the system.

The NMI and INTR inputs are used to generate non-maskable and maskable interrupts respectively.

Asserting the NMI input causes the processor to execute the interrupt handler pointed to by an interrupt vector stored in memory.

If interrupts are enabled then asserting INTR causes the CPU to carry out an interrupt acknowledge bus cycle which reads a 1-byte interrupt number from the bus (typically from an interrupt control chip). The corresponding interrupt vector is then fetched and the corresponding interrupt handler executed as with NMI.

In either case the current instruction is completed before the interrupt is recognized. We will cover the details of the processor’s interrupt handling in detail in a later lecture.

Other Signals

The ’386SX has a number of other signals which we will not cover at this time. For completeness, these are: HOLD and HOLDA (used by other devices to request that the CPU to give up control of the processor bus by disabling all of its outputs), LOCK* (used to prevent other devices from requesting use of the processor bus), NA* (“next address” used to “pipeline” processor cycles), and PEREQ, BUSY*, and ERROR* (used to interface to a floating point co-processor).

80386SX Bus Cycles

Execution of each 80386SX *instruction* requires one or more bus cycles. Typically, this involves reading an instruction from memory possibly followed by transfers of data between the CPU and memory or I/O devices.

In addition to the read and write bus cycles from memory and I/O address space the CPU can also execute an interrupt acknowledge bus cycle and can be in an idle or halted mode.

Read and Write Bus Cycles

Recall that a bus cycle requires at least two processor cycles (T1 and T2). The address and bus control signals go active at the start of the T1 processor cycle. During a write cycle the data bus is driven with the value to be written during the second half of T1 and this value stays on the bus into the first half of the following T1 cycle. During a read cycle the processor loads the value from the data bus at the end of the last T2 cycle.

Wait States

At the end of each T2 cycle the processor checks the READY* input. If it is active, the bus cycle is terminated, otherwise an additional T2 cycle is run. These additional *wait states* are used to accommodate slow memory by increasing the time available between when the address is output and the time when the data is required. If the memory being designed into a system will require wait states, a wait state generator circuit must be designed so that READY* is asserted after two or more T2 states have elapsed following the start of the bus cycle.

Input and Output Cycles

I/O read/write cycles are the same as memory read/write cycles except that the M/IO* signal is low.

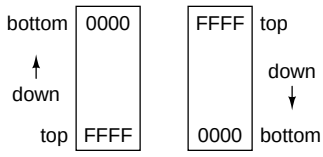
Interrupt Acknowledge Cycle

An interrupt acknowledge cycle (performed in response to INTR) is the same as a read cycle except that the bus control signals are set to indicate an interrupt acknowledge cycle. The value read during the interrupt acknowledge cycle is then multiplied by 4 and used to load an interrupt vector from this address in memory.

80x86 Instruction Set

Up or Down?

The "top of memory" is the highest-valued address. A stack is said to "grow down" when it's address gets smaller as more values are put on the stack. However, many authors draw diagrams showing memory contents in reverse order (with the lower-valued addresses above higher-valued ones). Be careful when using these terms.

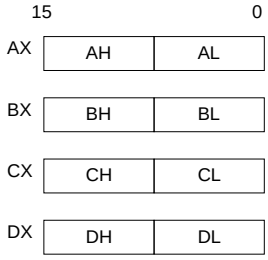


Real and Protected Modes

While the original Intel 16-bit CPUs, the 8086/8088 are no longer widely used, all later Intel processors such as the 80386, 80486 and Pentium processors can still execute 8086 software. The more recent CPUs can be switched by software into either the 8086-compatible "real" mode or to the more powerful "protected" mode. Protected mode extends the data and address registers from 16 to 32 bits and includes support for memory protection and virtual memory. Unfortunately, the details of interrupt handling in protected mode are too complex to cover in this course so we will restrict ourselves to 80x86 real-mode programming.

Registers

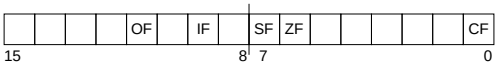
The 8086 includes four general-purpose 16-bit data registers (AX, BX, CX and DX). These register can be used in arithmetic or logic operations and as temporary storage. The most/least significant byte of each register can also be addressed directly (e.g. AL is the LS byte of AX, CH is MS byte of CX, etc.).



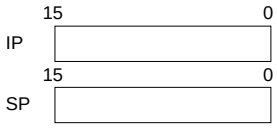
Each register also has a special purpose as we'll discuss later:

Register	Special Purpose
AX	multiply/divide
BX	index register for MOVE
CX	count register for string operations
DX	port address for IN and OUT

There is a 16-bit program flags register. Three of the bits indicate whether the result of the most recent arithmetic/logical instruction was zero (ZF), has a negative sign (SF), or generated a carry or borrow (CF) from the most-significant bit. The overflow bit (OF) indicates overflow if the operands are signed (it's the carry/borrow from the second most-significant bit). A fourth bit, the interrupt enable bit (IF) controls whether maskable interrupt requests (on the IRQ pin) are recognized.

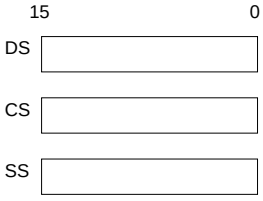


The address of the next instruction to be executed is held in a 16-bit instruction pointer (IP) register (the "program counter"). A 16-bit stack pointer (SP) implements a stack to support subroutine calls and interrupts/exceptions.



Exercise 27: How many bytes can be addressed by a 16-bit value?

There are also three segment registers (CS, DS, SS) which allow the code, data and stack to be placed in any three 64 kByte "segments" within the CPU's 1 megabyte (20-bit) address space as described later.



Instruction Set

We only cover the *small* subset of the 8088 instruction set that is essential. In particular, we will not mention various registers, addressing modes and instructions that could often provide faster ways of doing things.

A summary of the 80x86 real-mode instruction set is available on the course Web page and should be printed out if you don't have another reference.

Data Transfer

The MOV instruction is used to transfer 8 and 16-bit data to and from registers. Either the source or destination has to be a register. The other operand can come from another register, from memory, from immediate data (a value included in the instruction) or from a memory location "pointed at" by register BX. For example, if COUNT is the label of a memory location the following are possible assembly-language instructions :

```
; register: move contents of BX to AX
    MOV     AX,BX
; direct: move contents of the address labelled
; COUNT to AX
    MOV     AX,COUNT
; immediate: load CX with the value 240
    MOV     CX,0F0H
; memory: load CX with the value at
; address 240
    MOV     CX,[0F0H]
; register indirect: move contents of AL
; to memory location in BX
    MOV     [BX],AL
```

Most 80x86 assemblers keep track of the type of each symbol (byte or word, memory reference or number) and require a type "override" when the symbol is used in a different way. The OFFSET operator converts a memory reference to a 16-bit value. For example:

```
   MOV  BX,COUNT        ; load the value at location COUNT
   MOV  BX,OFFSET COUNT ; load the offset of COUNT
```

16-bit registers can be pushed (the SP is first decremented by two and then the is value stored at the address in SP) or popped (the value is restored from the memory at SP and then SP is incremented by 2). For example:

```
PUSH    AX      ; push contents of AX
POP     BX      ; restore into BX
```

There are some things to note about Intel assembly language syntax:

- the order of the operands is *destination,source* — the reverse of that used on the 68000!
- semicolons begin a comment
- the suffix 'H' is used to indicate a hexadecimal constant, if the constant begins with a letter it must be prefixed with a zero to distinguish it from a label
- the suffix 'B' indicates a binary constant
- square brackets indicate indirect addressing or direct addressing to memory (with a constant)
- the size of the transfer (byte or word) is determined by the size of the *register*

Exercise 28: What is the difference between the operands [BX] and BX? What about [1000H] and 1000H? Which of these can be used as the destination of a MOV instruction? Which of these can used as the source?

I/O Operations

The 8086 has separate I/O and memory address spaces. Values in the I/O space are accessed with IN and OUT instructions. The port address is loaded into DX and the data is read/written to/from AL or AX:

```
MOV     DX,372H ; load DX with port address
OUT     DX,AL   ; output byte in AL to port
                ; 372 (hex)
IN      AX,DX   ; input word to AX
```

Arithmetic/Logic

Arithmetic and logic instructions can be performed on byte and 16-bit values. The first operand has to be a register and the result is stored in that register.

```
; increment BX by 4
    ADD     BX,4
; subtract 1 from AL
    SUB     AL,1
; increment BX
    INC     BX
; compare (subtract and set flags
; but without storing result)
    CMP     AX,MAX
; mask in LS 4 bits of AL
    AND     AL,0FH
; divide AX by four
    SHR     AX,2
; set MS bit of CX
    OR      CX,8000H
; clear AX
    XOR     AX,AX
```

Exercise 29: Explain how the AND, SHR (shift right), OR and XOR instructions achieve the results given in the comments above.

Control Transfer

Conditional jumps transfer control to another address depending on the values of the flags in the flag register. Conditional jumps are restricted to a range of -128 to +127 bytes from the next instruction while unconditional jumps can be to any point.

```
; jump if last result was zero (two values equal)
    JZ      skip
; jump if greater than or equal
    JGE     notneg
; jump if below
    JB      smaller
; unconditional jump:
    JMP     loop
```

The assembly-language equivalent of an `if` statement in a high-level language is a CoMPare operation followed by a conditional jump.

Different conditional jumps are used for comparisons of signed (JG, JGE, JL, JLE depend on OF and CF) and unsigned values (JA, JAE, JB, JBE depend on CF only).

Exercise 30: If a and b were signed 16-bit values, what would be the assembly-language equivalent of the C-language statement `if ( a != 0 ) goto LOOP;`? What about `if ( a <= b ) return ;`? What if they were unsigned?

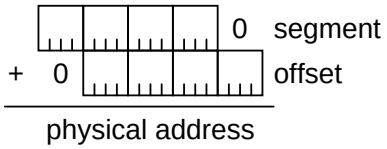
The CALL and RET instructions call and return from subroutines. The processor pushes IP (the address of the *next* instruction) on the stack during a CALL instruction and the contents of IP are popped by the RET instructions. For example:

```
    CALL    readchar
    ...
readchar:
    ...
    RET
```

Exercise 31: Write a sequence of a MOVE, a PUSH and a RET instruction that has the same effect as the instruction JMP 1234H?

Segment/Offset Addressing

Since address registers and address operands are only 16 bits they can only address 64k bytes. In order to address the 20-bit address range of the 8086, physical addresses (those that are put on the address bus) are always formed by adding the values of one of the *segment registers* to the 16-bit "offset" address to form a 20-bit address.



The segment registers themselves only contain the most-significant 16 bits of the 20-bit value that is contributed by the segment registers. The least significant four bits of the segment address are always zero.

By default, the DS (data segment) register is used to form addresses associated with data transfer instructions (e.g. MOV), the CS (code segment) register is added to the IP register (e.g. for JMP or CALL), and SS is added to SP (e.g. PUSH or to save/restore addresses during CALL/RET or INT instructions). There is also an "extra" segment register, ES, that is used when access to other locations in memory is required.

Exercise 32: If DS contains 0100H, what address will be written by the instruction `MOV [2000H],AL`? If CX contains 1122H, SP contains 1234H, and SS contains 2000H, what addresses will change and what will be their values when the PUSH CX instruction is executed?

The use of segment registers reduces the size of pointers to 16 bits. This reduces the code size but also restricts the addressing range of a pointer to 64k bytes. Performing address arithmetic within data structures larger than 64k is awkward. This is the biggest drawback of the 8086 architecture.

For simplicity will restrict ourselves to short programs where all of the code, data and stack are placed into the same 64k segment (so that CS=DS=SS).

Interrupts and Exceptions

In addition to *interrupts* caused by external events (such as an IRQ signal), certain instructions such as a dividing by zero or the INT instruction generate *exceptions*.

The 8086 reserves the lower 1024 bytes of memory for an interrupt vector table. There is one 4-byte vector for each of the 256 possible interrupt/exception numbers. When an interrupt or exception occurs, the processor: (1) pushes the flags register, CS, and IP (in that order), (2) clears the interrupt flag in the flags register, (3) loads IP (lower word) and CS (higher word) from the appropriate interrupt vector location, and (4) transfers control to that location.

For external interrupts (IRQ or NMI) the interrupt number is read from the data bus during an interrupt acknowledge bus cycle. For internal interrupts (e.g. INT instruction) the interrupt number is determined by the instruction.

The INT instruction allows a program to generate any of the 256 interrupts. This "software interrupt" is typically used to access operating system services.

Exercise 33: MS-DOS programs use the INT 21H instruction to invoke an "exception handler" that provides operating system services. Where would the address of the entry point to these DOS services be found? Where is the new IP? The new CS?

The CLI and STI instructions clear/set the interrupt-enable bit in the flags register to disable/enable external interrupts.

The IRET instruction pops the IP, CS and flags register values (in that order) from the stack and thus returns control to the instruction following the one where interrupt or exception occurred.

Exercise 34: Programs typically store their local variables and

return addresses on the stack. What would happen if you used RET instead of IRET to return from an interrupt?

Pseudo-Ops

A number of assembler directives ("pseudo-ops") are also required to write assembly language programs. `ORG` specifies the location of code or data within the segment, `DB` and `DW` are used to include bytes and words of data in a program.

Example

This is a simple program that demonstrates the main features of the 8086 instruction set. It uses the INT instruction to "call" MS-DOS via the 21H software interrupt handler to write characters to the screen.

```
; Sample 8086 assembly language program.  This program
; prints the printable characters in a null-terminated
; string (similar to the unix ("strings" program).

; There is only one "segment" called "code" and the
; linker can assume DS and CS will be set to the right
; values for "code". The code begins at offset 100h
; within the segment "code" (the MS-DOS convention for
; .COM files).

code segment public
     assume  cs:code,ds:code
     org     100h

start:
    mov     bx,offset msg   ; bx points to string
loop:
    mov     al,[bx]         ; load a character into al
    cmp     al,0            ; see if it's a zero
    jz      done            ; quit if so
    cmp     al,32           ; see if it's printable
    jl      noprt           ; don't print if not
    call    printc          ; otherwise print it
noprt:
    inc     bx              ; point to next character
    jmp     loop            ; and loop back

done:
    int     20h             ; return to DOS

; subroutine to print the byte in al

printc:
    push    ax              ; save ax and dx
    push    dx
    mov     dl,al           ; use DOS to
    mov     ah,02H          ; print character
    int     21H
    pop     dx              ; restore ax and dx
    pop     ax
    ret

msg  db     'This',9,31,32,'is',20H,'a string.',0

; example of how to reserve memory (not used above):

buf  db     128 dup (?)     ; 128 uninitialized bytes

code ends
     end    start
```

The `OFFSET` operator is used to tell this assembler to use the offset of `msg` from the start of the `code` segment instead of loading `bx` with the first word in the buffer.

Exercise 35: Re-write this code in C using a pointer variable, `p`, for `BX`. Use the C function `putchar()` instead of `INT 21` function 2.

Writing Assembly Language Programs

The most efficient way to write an assembly language program is to write it in C (or some other higher-level language) and let a compiler generate the assembly-language code. Very few programmers can generate code that is faster or more compact than a good C compiler.

If for some reason (typically due to poor management) you need to write assembly language, first write it in C. Debug, test, profile and optimize (in that order) the C version. Then covert as little of the code to assembly-language as necessary to meet the peformance requirements. Typically this is worthwhile when you need to make use of processor-specific features (e.g. special-purpose signal-processing instructions). Isolate the assembly language in architecture-specific files rather than using in-line assembler.