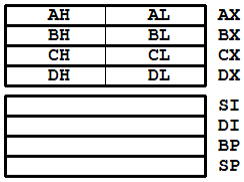
Lecture #3 – February 2, 2004

The 8088 Architecture

In stark contrast to the 6502 architecture is that of the Intel 8088, released in 1978. The 8088 (and all of its successors, such as the 286, 386, 486, and Pentium series) uses a very different set of registers and a different approach to programming from that of the 6502.

Registers

In the 6502 there is a single 8-bit accumulator, used by default in most instructions, as well as two index registers with more limited uses than the accumulator. In the 8088 there are eight more-or-less general purpose 16-bit registers, some with more accumulator-like functionality than others. These registers are called `AX`, `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, and `DI`. In the earlier 8-bit Intel 8080 processor, vintage 1974, there were four 8-bit registers called `A`, `B`, `C`, and `D`. In an attempt to keep the 8088 marginally compatible with the 8080, registers `AX`, `BX`, `CX`, and `DX` can be accessed by each half independently from the other half. Thus, the 16-bit `AX` register is composed of two 8-bit halves, `AH` (high byte) and `AL` (low byte). The `SI` and `DI` registers are used primarily in instructions that deal with long strings of bytes that need to be moved from one block in memory to another. The `BP` register is used in subroutines and arrays, and `SP` is the stack pointer.



Remember that the 6502 is a ***one-address*** machine, because an instruction such as `ADC` (add with carry) implicitly uses the single accumulator as part of the calculation and requires only a single address to an operand in memory. In contrast, the 8088 is a ***two-address*** machine. In an instruction such as `ADD`, a programmer must tell the 8088 not only one of the source operands, but also the destination of the result. The destination is used as the second source operand. For example, the instruction `ADD AX,Temp` means that the `AX` register and the contents of memory variable `Temp` are added together and the result is stored back into the `AX` register. In a high-level pseudocode, this is written as `AX := AX + Temp`. Similarly, `ADD Temp,AX` means `Temp := Temp + AX`. It is fairly unusual for an instruction set to allow the destination of a calculation to be in main memory. In the 8088, both source and destination can be either a register or a memory location, but they can't both be memory locations at the same time.

(In the Computer Organization book the notation is backwards from that used on the 8088. An instruction such as `ADD AX,BX` is interpreted as `AX := AX + BX` on the 8088, but it would be interpreted as `BX := AX + BX` in the book.)

The assembler takes care of determining whether an instruction is a 16-bit or an 8-bit instruction, based on which registers are used. (In ambiguous cases the programmer must specify explicitly whether to use 8-bit or 16-bit operands.) For example, the following instructions all refer to 16-bit quantities:

`ADD AX,BX`	(add `BX` to `AX`),
`ADD AX,1`	(add 16-bit constant 1 to `AX`)
`ADD AX,Temp`	(add 16-bit memory variable `Temp` to `AX`)
`ADD Temp,AX`	(add `AX` to 16-bit memory variable `Temp`)

but the following instructions all refer to 8-bit quantities:

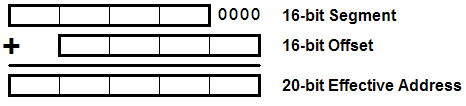
`ADD AH,BL`	(add `BL` to `AH`, ignoring both `BH` and `AL`)
`ADD AH,1`	(add 8-bit constant 1 to `AH`, ignoring `AL`)
`ADD AH,Temp`	(add 8-bit memory variable `Temp` to `AH`)
`ADD Temp,AH`	(add `AH` to 8-bit memory variable `Temp`)

Note that without any extra information about variable `Temp`, the instructions `ADD AX,Temp` and `ADD AH,Temp` are both legal; the first picks up two bytes from memory (the byte referenced by `Temp` and the one following, using little-endian byte order), while the second picks up but a single byte (just the byte referenced by `Temp`). The following instructions are illegal because they mix 8-bit and 16-bit quantities, and are flagged as errors by assemblers:

`ADD AX,BL`	(Illegal: `AX` is 16 bits, `BL` is 8 bits)
`ADD DH,BX`	(Illegal: `DH` is 8 bits, `BX` is 16 bits)
`ADD CL,1000`	(Illegal: `CL` is 8 bits, constant 1000 will fit into 16 bits but not into 8 bits)

Addressing

Memory addressing on the 8088 is convoluted. With a 16-bit address register only 65,536 bytes (2^{16}=64K) of memory can be addressed, and the 8088 is primarily a 16-bit processor. The 8088 can access up to 1,048,576 bytes (1 megabyte) of memory, thus requiring 20 bits of address to do so (2^{20}=1,048,576). The solution selected by Intel is to add two 16-bit registers together for each such 20-bit address, one register offset to the left by four bits. The left-shifted register is called a ***segment register***, and the unshifted register is the ***offset***.



There are four segment registers on the 8088, called DS, CS, SS, and ES. From the way that addresses are formed, each segment register points at a 64K block of memory. The DS register points at the data segment, used to contain data variables. The CS register points at the code segment, used to contain the currently running section of code. The SS register points at the stack segment. The ES register points at the "extra" segment, used explicitly by a few instructions, but mostly used as temporary storage for segment addresses.

The processor uses the CS segment register automatically when it fetches an instruction from memory, it uses the DS segment register automatically when it fetches a datum from memory, and it uses the SS segment register automatically when it pushes values onto or pops values from the stack.

Segment registers may point at overlapping blocks of memory. What happens to the segment registers defines the major difference between .COM programs and .EXE programs in the old MS-DOS world. In .COM files, the CS, DS, and SS segment registers all contain the same value and generally are not allowed to change, and the entire program must fit into one 64K block. Such programs are relatively easy to write, even in assembly language, as the programmer need not worry about ever changing the segment registers once they are properly initialized. The downside, of course, is that the entire program must fit into 64K. In contrast, .EXE programs allow segment registers to point at different blocks of memory, and move them around during the execution of the program in order to support large code sizes and large data blocks (up to all available memory). It is the responsibility of the assembly language programmer to maintain those segment registers properly.

Instruction Symmetry

Many registers are used in the same way as other registers of the same class; they are considered to be ***symmetric*** instructions. For example, the ADD (simple addition) and ADC (add with carry) instructions may add together any two 16-bit registers, any two 8-bit registers, or any register and any memory location of the same size, in either order. Similarly, the MOV (move data) instruction can move data values from any register to/from any other register or memory location of the same size. These instructions represent a high degree of symmetry.

On the 8088 symmetric instructions are the exception, rather than the rule. Many instructions use specific registers in unique ways. For example, the 16-bit version of the IMUL (integer multiply) instruction always multiplies the AX register by its operand. Multiplication of two 16-bit values always generates a 32-bit result; the low order 16 bits of the result always go back into AX and the high order 16 bits always go into DX. The instruction is hard-wired to work this way, and you cannot perform multiplication into any other result registers. Anything valuable in DX must be moved out of the way before the multiplication takes place, either into memory or an unused register. If not, its value will be lost.

Many other registers have special uses as well. The BX register is used as the base address of an array in many instructions. For example, in the code below, the variable Buffer points to the start of an array in memory. The first MOV instruction moves the *address offset* of the memory variable into BX, instead of fetching the *contents* of the first two bytes of the array from memory. The next MOV instruction fetches a byte from memory into register AH, where the address of the byte fetched is in the BX register. The BX register is incremented in order to step through the individual bytes of the array.

```
        MOV  BX, OFFSET Buffer
        …
        MOV  AH,[BX]
```

Similarly, the CX register is most often used as a loop counter. Any register may be used as a loop counter, but only the CX register has a special instruction called LOOP that decrements its value and then jumps to a location if the result is not exactly zero. The loop on the left (below) shows how to do a task ten times using explicit decrement, compare, and jump instructions. This code may be written to use *any* register on the 8088, including CX. The loop on the right shows how much shorter the code will be with the special instructions that use *only* the CX register.

```
        MOV  CX,10              MOV  CX,10
Top     …               Top     …
        …                       …
        DEC  CX                 LOOP Top
        CMP  CX,0
        JNZ  Top
```

Continuing asymmetries to a ridiculous extreme, the SI and DI registers are used almost exclusively in special instructions that move data around in memory. To illustrate this, we will show the MOVSB instruction, which moves a string of bytes from one place to another. The instruction expects the source of the string to be defined by segment register DS and offset SI, and expects the destination to be defined by segment register ES and offset DI. Executing the instruction once copies one byte from Memory[DS:SI] to Memory[ES:DI], and then it also increments both SI and DI by 1. Prefixing the instruction by the special code REP repeats the move and decrements CX as many times as necessary until CX equals zero, thus copying up to 64K in one block. A *lot* of preparation is necessary, but once everything is set up correctly the single construction REP MOVSB does *all* of the following pseudocode steps:

```
Repeat
     Memory[ES:DI] := Memory[DS:SI] ;
     SI := SI + 1 ;
     DI := DI + 1 ;
     CX := CX – 1 ;
Until CX = 0 ;
```

In the real 8088 code that follows, *everything* but the last line is used to set up the registers specifically to use the special `REP MOVSB` combination:

```
        MOV  AX,DS
        MOV  ES,AX          Copy DS into ES
        MOV  CX,____        Number of bytes to move
        MOV  SI,OFFSET ____ Offset of Source block
        MOV  DI,OFFSET ____ Offset of Destination
REP     MOVSB               Do it all!
```

Instructions on the 8088 are between one and twelve bytes in length, and vary widely in execution time. In particular, the time used by the `REP MOVSB` construction depends almost entirely on the contents of the `CX` register, between one byte-move and 65,536 byte-moves, and may dominate the execution time for some blocks of code.

Example

For an example we will write the 8088 assembly language code for the high-level statement **`c = a + b`**, using multiple precision arithmetic just as we did in the previous lecture on the 6502. In the data area, referenced by the `DS` segment register, we define the variables for 4-byte integers as follows, and change the `SIZE` definition for a different number of bytes (there are better ways to do this, by the way):

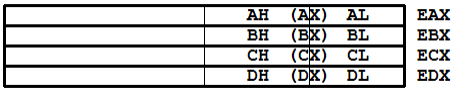
```
SIZE    EQU  4          Size of variables
START   EQU  0          Start of Data Segment
A       EQU  START      Offset into DS of A
B       EQU  A+SIZE     Offset into DS of B
C       EQU  B+SIZE     Offset into DS of C
```

The code to add the two multibyte numbers uses all of the tricks we've introduced so far. The `CX` register will be used as the loop counter, but instead of counting the number of 8-bit bytes added it will count the number of 16-bit words; this is why it is initialized to half the `SIZE` value (this computation is performed at assembly time since the value of `SIZE` is known, not at run time). As described earlier, the `LOOP` instruction decrements `CX`, and then branches to label `Top` if `CX` is not zero. The `BX` register will be used as the index into each array, and must be incremented twice to get to each successive two-byte word.

```
        MOV  CX,SIZE/2      Number of words
        MOV  BX,0           Offset into Array
        CLC                 Clear Carry
Top     MOV  AX,[BX+A]      Load 16 bits of A
        ADC  AX,[BX+B]      Add 16 bits of B
        MOV  [BX+C],AX      Store 16 bits of C
        INC  BX             Increase BX to
        INC  BX                 next word index
        LOOP Top            Repeat until CX=0
```

Successors to the 8088

For completeness, some attention must be given to the descendants of the 8088 chip. Intel needed to make each new chip backwards compatible with all those that came before it from the same family. Processors such as the 286, 386, 486, Pentium, Pentium II, Pentium III, and Pentium IV each execute all programs written for earlier members of the family, including the 8088. They also include their own quirks and capabilities. While a Pentium IV will execute native 8088 code, it also allows for a much easier approach to programming. Starting with the 386, data registers are now 32 bits in length instead of 16. What was the 16-bit `AX` register in the 8088 is now the 32-bit `EAX` register, but the `AX`, `AH`, and `AL` sections of `EAX` are still available to programmers. Although segment registers are still present, all address registers (both segment and offset) are also 32 bits in length. Everywhere in the four gigabyte address space (2^{32} bytes) can be referenced without changing a segment register.



Our multibyte addition code on a 386 or later is similar to what we saw before, but we can now reference and add four bytes at a time instead of only two. The data area is unchanged, but the executable code is now written as follows:

```
        MOV  ECX,SIZE/4      Number of quads
        MOV  EBX,0           Offset into Array
        CLC                  Clear Carry
Top     MOV  EAX,[EBX+A]     Load 32 bits of A
        ADC  EAX,[EBX+B]     Add 32 bits of B
        MOV  [EBX+C],EAX     Store 32 bits of C
        INC  EBX             Increase EBX to
        INC  EBX                 next quad index
        INC  EBX
        INC  EBX
        LOOP Top             Repeat until ECX=0
```

The assembly language programmer must examine the code closely and decide if the four `INC EBX` instructions should be left alone or replaced by a single `ADD EBX,4` instruction. One of the two will occupy fewer bytes than the other, and one will run faster than the other. Programmers will choose the solution that matches their requirements the best, whether for speed or for space. The optimal situation occurs in cases where the smallest code is also the fastest, but that does not happen all the time. If we are interested in four byte integers only (variables `A`, `B`, and `C` each have four bytes reserved in the data area), the code is trivial since no loop will be required:

```
MOV  EAX,A
ADD  EAX,B
MOV  C,EAX
```

Conclusions

The lack of symmetry and special rules for certain instructions means that the 8088 assembly language programmer has a very difficult time designing programs to take advantage of the fancy features of the processor and avoid unnecessary data movement at the same time. The 8088 is considered a ***CISC*** design, or ***Complex Instruction-Set Computer***. For each of the "special" instructions such as `MOVSB` one must wonder if the complexity required by the processor to implement it and the amount of preparation a programmer must perform in order to use it are adequately balanced by the average number of times it will be seen in any application program. If a complex instruction is used only rarely, and if equivalent tasks can be performed as easily with simpler instructions, then perhaps it should not be present in the instruction set at all.

Many people have asked these same questions and concluded that the instruction set of a processor must include only simple instructions, and only those which are absolutely necessary to any program. Such ***RISC***, or ***Reduced Instruction-Set Computers***, typically contain a very small number of very simple instructions. Those instructions, because of their simplicity, are executed very quickly by the processor. They also make the hardware design of RISC processors reasonably straightforward, and because they are few in number the entire instruction set for a RISC design is much more easily learned by assembly language programmers than that for a CISC design. As we will see in the next lectures, the ARM processor is one such RISC machine.