Know Your Personal Computer

5. The CPU Base Instruction Set and Assembly Language Programming

S K Ghoshal

Siddhartha Kumar Ghoshal works with whatever goes on inside parellel computers. That includes hardware, system software, algorithms and applications. From his early childhood he has designed and built electronic gadgets. One of the most recent ones is a sixteen processor parallel computer with IBM PC motherboards.

This article describes the instruction set of the base architecture by illustrating it with an assembly language program.

Instruction Set

The instructions supported in the 8088 can be classified into:

Data Transfer Instructions that move data from the source to the destination. Examples are MOV, PUSH, POP and XCHG.

Arithmetic Instructions like ADD, SUB, MUL, DIV, INC (which increments) and DEC (that decrements).

Logic Instructions like NOT, AND, OR, XOR, and TEST.

Bit manipulation Instructions like SHL, SHR, ROL and ROR.

String Instructions operate on string. MOVS copies strings, SCAS scans them, STOS initializes them and CMPS compares them. MOVS also provides an alternative to MOV instruction for moving data. Similar alternatives exist for other string instructions. This is typical of a CISC architecture. There is often more than one way to do the same thing.

Assembly Language Programming

We will illustrate with an assembly language program which implements a computable function that calls itself. It computes the factorial of a non-negative integer. The factorial function, as we know can be recursively defined as in equation 1.

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x\, f(x-1) & \text{otherwise} \end{cases} \qquad (1)$$

Figure 1 An assembly language program to compute factorial.

```
.model small ; the virtual address space is small: only 64KB long
.code ; executable code follows in this assembly module
public _factorial      ;public near procedure callable by any C module
_factorial proc near ; subroutine entry point
push bp        ; save caller's BP
mov bp,sp ; fix the context of this instance
enter:                ; now the stack looks like Figure 3
push bx        ; save old BX (that was the caller's copy of n)
push dx; save  DX too (MUL affects DX must restore to caller)
inside:               ; Now stack looks like Figure 4 bottom-left
mov ax,[bp+4]   ; get parameter value from ss:[bp+4]
or ax,ax        ; check if the parameter is 0
jnz deeper      ; go deeper if parameter is non-zero
mov ax,1        ; 0! = 1
jmp done ; return one as result
deeper:           ; go deeper in recursion
mov bx,ax ; put parameter (n) in bx register
dec ax          ; calculate (n–1)
push ax         ; save (n–1)
call _factorial ; recursive call to itself
add sp,2        ; discard the parameter after return
mul bx          ; n * (n–1) –> dx and ax: dx gets msb, ax <– lsb
done  :               ; we are finished either way
pop dx          ; pop the saved registers dx first
pop bx          ; then bx – observe the reverse order
pop bp          ; restore caller's context
ret           ; return thread of control to caller
_factorial endp  ; the subroutine ends here
end ; the assembly module ends here
```

Figure 2 A C function to compute factorial.

```
int factorial (int arg1) /* An integer function with
            an integer argument */
{
 if (arg1= =0) return(1); /* If argument is zero,
            return 1 as result */
 return(arg1 * factorial(arg1–1)); /*Otherwise
            return arg1*factorial(arg1–1) */
}
```

Our implementation of the factorial function follows the same logic. This assembly language program (See *Figure 1*) is written in such a way that it can be called from C.

It is functionally equivalent to the implementation in C, as given in *Figure 2*. Note that comments (statements that are meant to be read by a human and not translated by a compiler are called comments) in C are between '/*' and '*/', whereas in assembly language whatever follows a semicolon in a line is a comment.

Case Sensitivity

If the use of upper-case or lower-case letters in a programming language changes the syntax and semantics of the program, then that language is called case-sensitive. C is case-sensitive. printf("Hello") is correct. pRintf("Hello") is wrong. Assembly language is case-insensitive. Both push bx and PUSH BX are correct and mean the same thing. Fortran 90 is not case-sensitive.

And as you have probably guessed, when you compile a program like the one in *Figure 2* (the exact text of your program will depend on the type of C programmer you are) it becomes a program like the one in *Figure 1* (the exact code produced will depend on the type of C compiler you use). There are variations in the details (for that matter the way one writes equation 1 depends on the person) of intermediate representation and the symbols used, but equation 1, *Figure 1* and *Figure 2* represent the same thing to different beholders. However, as the beholders understand different languages and operate at different levels, translation is required so that one beholder can work for the other. From C to assembly language, the translation process is called *compilation*. From assembly language, translation into machine language (some call it *binary* or *object code*) is by a program called the *assembler*. Only after that final translation step is done can the object code produced by the compiler be executed by the hardware.

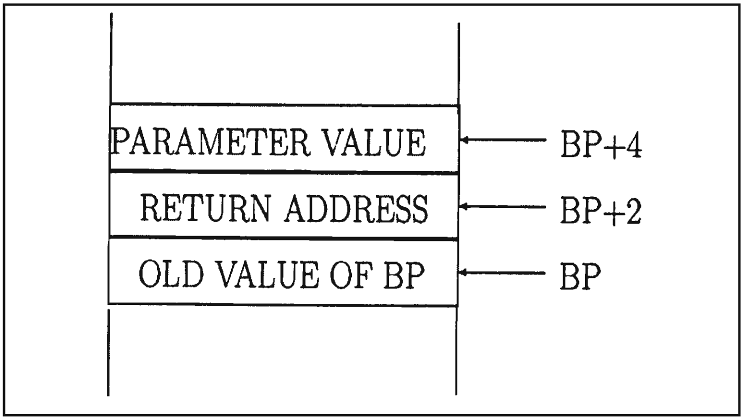

Figure 3 Activation record for factorial computation.

The machine language is nothing but bit-patterns stored in memory. It does not have any comment or labels. (Symbols which denote specific locations in the flow of control in a computer program are called labels. High-level languages as well as assembly language have labels that could be referred to by the programmer. In assembly language, each label is associated with a unique address. Each label is followed by a colon. In the program of *Figure 1*, *enter, inside, deeper* and *done* are labels. All

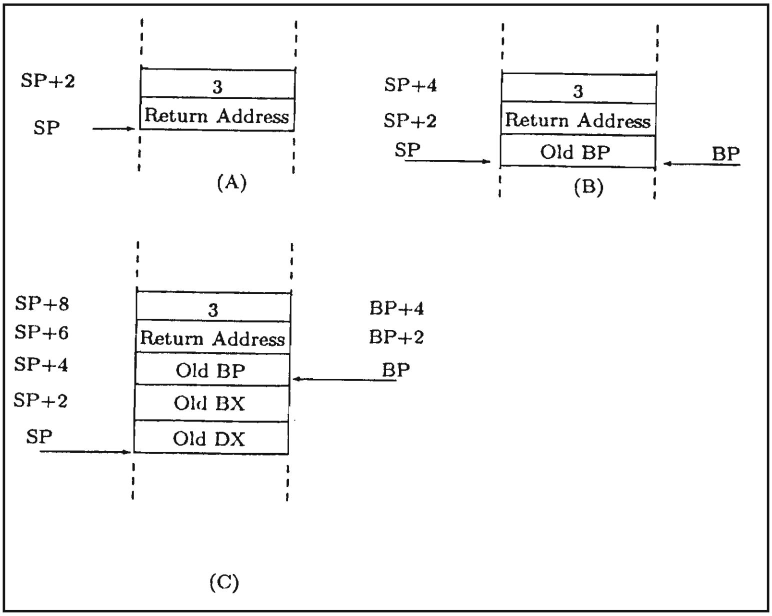

Figure 4 Activation record for n=3.

references to symbolic labels in assembly language are replaced by the actual address which is just a binary number.)

As an example, see how the program of *Figure 1* will look when it is *assembled* or translated into machine language and then *unassembled* again in an attempt to decipher its semantics. *Figure 6* shows a portion of the assembly language program of *Figure 1* along with its machine language object code written as a hexadecimal number. For example, 55 in hexadecimal which is the same bit-pattern as 01010101 in binary is the object code produced from the assembly language statement PUSH BP. And as one can see, it is possible to do a reverse translation from 55 to PUSH BP. In fact, *Figure 6* is generated by unassembling a portion of the machine language program that was generated by assembling the assembly language program of *Figure 1*. However much of the symbolic information like the names of labels (*e.g.* enter: and inside:) and program variables is lost. All comments are lost too. The machine language is designed not to be

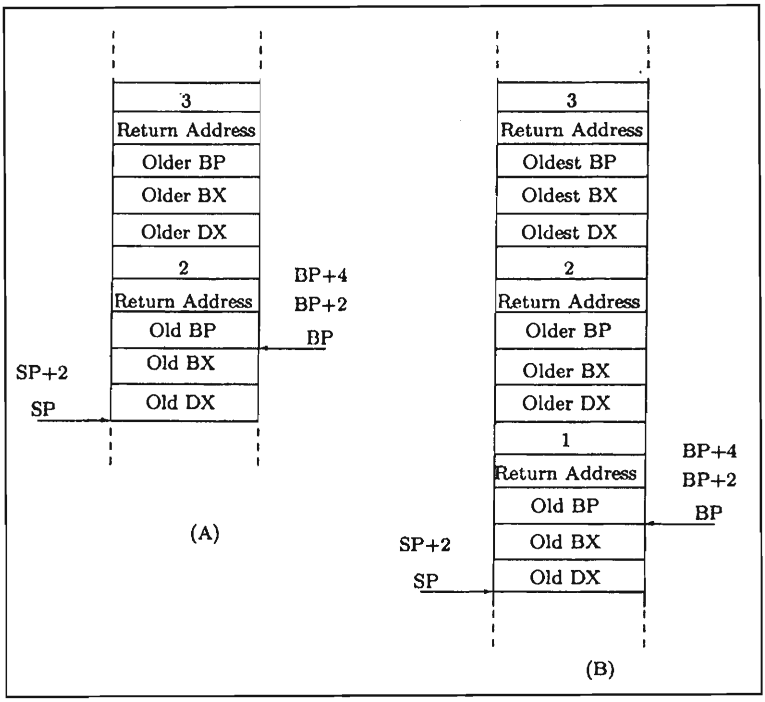

Figure 5 Activation record after recursion proceeds.

understood but to be executed. And to do that, the hardware does not need comments, names or labels. Nor does it need to 'understand' what it is doing. It just does what it is told to do. So the bit-pattern corresponding to the stream of bytes 558BEC5352BA0000 is all that the CPU needs to execute the first five lines of code in *Figure 1*.

However, it is extremely difficult to work with machine language code and figure out its intended purpose, particularly when it is (apparently) not serving this purpose. However, hackers like me must do it when everything else fails.

Let us now see how the program in *Figure 1* works. Remember three rules of parameter passing in C:

- The C caller program pushes the last parameter (or *argument*) first.
- In C, values of arguments are directly pushed into the stack.
- The C caller cleans the stack (this is explained later in this article) once the thread of control returns.

Also remember that in Intel 80X86 architectures, the accumulator is used to return values of C callable functions that return integers. In our case, the factorial function has only one argument and it returns an integer result.

So if the factorial function is called with a parameter value, that value is pushed first into the stack. After that the return address is pushed, as it is a function call and the thread of control must come back to the caller at the C statement, just after the call is

Assembly Language for all seasons?

Can an assembly language program be written for any problem that can be solved using a computer? Can this be done if the problem is very complex and is solved using a high-level language? The answer is "Yes". As long as that high-level language is compiled and run as an executable, there exists not only an assembly language program, but also an automatic way of generating that program. It is a different matter altogether that a human programmer will not consider it necessary to write it in assembly language.

Machine Code	Assembly Language Code
55	push bp
8B EC	mov bp,sp
53	push bx
52	push dx
BA 0000	mov dx,0

Figure 6 Deciphering a machine language program

Why Should Anyone Ever Write Programs in Assembly Language?

There are C and other high-level language compilers. Programs written in high-level languages are easily portable from one machine to another, easier to understand and explain, easier to modify to serve another purpose and easier to correct (some call this *debugging*) if the program is not working as per its specification. So why should one bother to write a program in assembly language? Many programmers just do. And despite advances made in compiler technology, hardware and high-level language design principles, there are still people who must write some of their programs in assembly language for a good reason. Unlike all other phases of translation, the translation from assembly language to machine language is one-to-one. There is only one machine language program that can be generated from a given assembly language program. This fact makes the execution of assembly language programs predictable and controllable. Thus for all time-critical applications people still use assembly language programming as one can predict the timing of various events within the entire computer system with a resolution of one clock cycle. For controlling hardware and generating different kinds of signals to drive hardware devices, assembly language is indispensable. And all programs become machine language programs and only then run. So all sophisticated applications development environments and programming languages, when the going is smooth, indicate in a highly cultured and refined way that it is doing the job as the human being wanted the job to be done. But when they get into serious trouble, they just crash without being able to report anything at all. Then someone has to use assembly language programming and unassemble the machine language code in order to identify the problem and fix it. (See *Figure 1* of the *Resonance* article in this series on system software). An advanced computing environment is nothing but bare digital hardware surrounded by layers of system software that appears more sophisticated. This is just like a mammal, who, after being trained by different stages of education becomes a computer scientist. And in both cases the background reveals itself from time to time in response to certain basic stimuli. And to recover from such situations, as well as to bring the sophistication back, you need to talk to both of them in a language which their (respective) ancestors knew and carried out the semantic actions without question or comment. Also if you desperately want to know how a given software works, and nobody knows and/or wants to tell you, you can always single step through the machine language code instruction by instruction to find out. For me and many other patient hackers, it often also is the best source of knowledge. There are no two ways to interpret what you read in machine language.

made. The return address is a binary number which indicates the address of the start of the object code produced from the next C statement. The image of the stack is called the *activation record*. It grows in size as more recursive subroutine calls are made, or as subroutines call other subroutines. Each time a subroutine calls itself, a new *instance* of it is created, which has its own context.

So we need a way to distinguish between the contexts of the currently executing subroutine and the contexts of its caller(s) and callee(s). The 80X86 architecture has a special purpose register BP to do just that. The BP points at a location in the stack. The stack grows downwards in the 80X86 architecture. The positions in the stack above what the BP is pointing at belong to the context of the caller. Positions below what the BP is pointing at belong to the context of this subroutine and its callee(s). Thus the value of BP itself at the label *enter*: is the most important key to the context of the subroutine. No matter how deep the recursion is and how huge the context of each of these calls are, given the value of BP, the context can be accessed easily. Also local variables are allocated storage this way. The default storage class which is called automatic in C are local variables. Separate copies (possibly holding different values) of local variables are created whenever the subroutine that defines them is invoked. At runtime, they are allocated storage just by subtracting the total size (in bytes) occupied by them from the SP register, keeping BP as it is. From then on, the local variables can always be referenced at an address a few known byte locations below where the BP is pointing. Registers and other (static) variables, whose values are altered within subroutines are saved inside them and restored at the end by pushing them into the stack and popping them off the stack. Thus the stack pointer SP is a very busy register, whose value changes all the time. Therefore locating any variable by counting up or down from the place where SP is pointing is a nightmare. On the other hand, the value of BP remains unchanged all along the execution of the current instance of the subroutine. That is from *enter*: to *done*: in our case (*Figure 1*) BP keeps pointing at the same place on the stack and all the parameters passed to the factorial subroutine, as well as its own local variables can be easily accessed using BP. So the value of the BP register is the first thing saved in the stack once the thread of control enters a subroutine. That is done by the

PUSH BP instruction in *Figure 1*. After that, the activation record looks like *Figure 3*.

Let us call the assembly language program of *Figure 1* from a C caller with parameter value 3 and profile it. In other words, let us

1. Write a C program as in *Figure 7*.

2. Compile it with a C compiler.

3. Assemble the code of *Figure 1*.

4. Link the two object code modules to produce an executable binary code. Note two points about the linking phase:

(a) There is an underscore in front of the factorial subprogram entry point declaration, whereas there is no such underscore in front of the name of the callee in *Figure 7*. This is no printing mistake. The C compiler puts an underscore in front of all external symbols it generates in the target assembly code. So in order to be recognized by the linker, we deliberately put an underscore in front of the manually written assembly language program.

(b) If you repeat the experiment, put in the appropriate switches to the C compiler and linker so that they use the small memory model ensuring that the virtual address space of the executable program is within 64 KBytes.

5. Run the executable code.

```
main( )
{
 int res;
 res=factorial(3) ;
}
```

Figure 7 A C caller program.

and trace the execution from the point it enters the assembly language procedure at label *_factorial* to the point it goes back to the program in *Figure* 7 and assigns a value of 6 to the variable *res*.

The parameter value 3 is pushed first and the return address next. The executable statement that moves the content of the AX register to the variable *res* is what the caller program of *Figure* 7 executes on return from the callee. So the address where that statement is loaded is the return address in our case. So when the thread of control enters the callee at label *_factorial,* (see *Figure 1*) the activation record looks like *Figure 4(A)*. On entering the assembly language callee, it first fixes its own context. So the BP of the current instance must copy the value of SP. And before that, the old value of BP must be saved. After that is done, the activation record looks like the diagram in*Figure 4(B)*. BP now stands firm as a rock. The portion of stack above where it is pointing now belongs to the caller and its caller in turn and so on. And the portion below belongs to this instance of *_factorial* and the callee(s) of *_factorial,* if any. From now on until the thread of control executes the ret instruction, SP will change many times. However, BP will stay put, like a bookmark, until the *pop bp* statement just before the *ret*. That *pop bp* will restore the caller's context just before handing control back to the caller. Executing that *pop bp* instruction will destroy this context, but that is exactly what we want.

With BP fixed, we are free to use SP and the stack. Knowing that we need to use the BX register to hold the value of the parameter *n*, we save BX by pushing its old value onto the stack. We also know that the multiply instruction will destroy the DX register. We cannot in any way make the MUL BX instruction spare DX. So we must save DX too. As I pointed out in the last article, this is one more reason why one should not call the registers of the 80X86 'General Purpose Registers'. Just as for DX, every register has a special use which sometimes destroys its contents.

After pushing BX and DX, when the control reaches the label inside:, the stack looks like *Figure 4(C)*. The parameter n is now accessed using BP. It is checked, if n is zero. If the parameter n is zero, then *_factorial* returns a value 1 set in the AX register. Otherwise the recursion goes deeper. The value of n is saved in BX and $n - 1$ is computed in AX. Now *_factorial* imitates its C caller (see *Figure* 7). Just as the caller pushes parameter and then calls *_factorial*, n–1 is pushed into the stack and *_factorial* calls itself. So this whole process is repeated. So for $n = 2$, at *inside*:, the stack looks like *Figure 5(A)* at the label *inside*:. For $n = 1$ look at *Figure 5(B)*. Finally at $n = 0$, the recursion terminates. So the *mul* instruction produces 1*1 for $n = 1$. This returns in AX to the instance of *_factorial* for $n = 2$. This instance produces 1*2 which goes back in AX to the instance for $n = 3$. The mul instruction for this instance produces 2*3. This goes back to the C caller and the variable *res* is set to a value of 6. Note how the assembly language callee *_factorial* cleans the stack after the thread of control returns from its own callee by adding 2 (the size of the parameter pushed in this case) to SP. That must be done, not only to imitate the C caller, but to keep the stack balanced. That is, before a parameter is pushed and a call made, if the value of SP is V_1, then after the thread of control returns, SP must again be made V_1. The act of doing so is called cleaning the stack.

Similarly, if you follow the rules you just learned, you can write assembly language caller programs that call programs written in C. These rules and techniques are fundamental in the implementation of multilayered system software organized in a hierarchical fashion as shown in *Figure 1* of the *Resonance*, April 1996 issue in this series.

Why Recursion Fails in Some Languages

If you write a subroutine in Fortran 77 that calls itself and call it, you will get absurd results. That is because recursion is not supported in Fortran 77 implementation. Parameters are passed using fixed locations in memory and thus one instance overwrites the context of the previous instance(s). Such an implementation cannot distinguish between old, older and oldest like ours did, as shown in *Figure 5*. Language implementations which support recursion, use activation records which grow and shrink dynamically, as was explained in this article. Recursion is so useful that Fortran 90 supports it.

Suggested Reading

- Allen I Holub. *Compiler Design in C.* Prentice Hall of India Private Limited, 1993.
- V Rajaraman. *Computer Programming in C.* Prentice Hall of India Private Limited, 1995.

Address for Correspondence
S K Ghoshal
Supercomputer Education
and Research Centre
Indian Institute of Science
Bangalore 560 012, India
email:
ghoshal@serc.iisc.ernet.in
Fax: (080) 334 1683