**1.WHY ARE STACKS USED IN COMPUTERS?**

Both hardware and software stacks have been used to support four major computing areas in computing requirements: expression evaluation, subroutine return address storage, dynamically allocated local variable storage, and subroutine parameter passing.

**1.1 Expression evaluation stack**

Expression evaluation stacks were the first kind of stacks to be widely supported by special hardware. As a compiler interprets an arithmetic expression, it must keep track of intermediate stages and precedence of operations using an evaluation stack. In the case of an interpreted language, two stacks are kept. One stack contains the pending operations that await completion of higher precedence operations. The other stack contains the intermediate inputs that are associated with the pending operations. In a compiled language, the compiler keeps track of the pending operations during its instruction generation, and the hardware uses a single expression evaluation stack to hold intermediate results.

To see why stacks are well suited to expression evaluation, consider how the following arithmetic expression would be computed:

**X = (A + B) \* (C + D)**

First, A and B would be added together. Then, this intermediate results must be saved somewhere. Let us say that it is pushed onto the expression evaluation stack. Next, C and D are added and the result is also pushed onto the expression evaluation stack. Finally, the top two stack elements (A+B and C+D) are multiplied and the result is stored in X. The expression evaluation stack provides automatic management of intermediate results of expressions, and allows as many levels of precedence in the expression as there are available stack elements. Those readers who have used Hewlett Packard calculators, which use Reverse Polish Notation, have direct experience with an expression evaluation stack.

The use of an expression evaluation stack is so basic to the evaluation of expressions that even register-based machine compilers often allocate registers as if they formed an expression evaluation stack.

**1.2 The return address stack**

With the introduction of recursion as a desirable language feature in the late 1950s, a means of storing the return address of a subroutine in dynamically allocated storage was required. The problem was that a common method for storing subroutine return addresses in non-recursive languages like FORTRAN was to allocate a space within the body of the subroutine for saving the return address. This, of course, prevented a subroutine from directly or indirectly calling itself, since the previously saved return address would be lost.

The solution to the recursion problem is to use a stack for storing the subroutine return address. As each subroutine is called the machine saves the return address of the calling program on a stack. This ensures that subroutine returns are processed in the reverse order of subroutine calls, which is the desired operation. Since new elements are allocated on the stack automatically at each subroutine call, recursive routines may call themselves without any problems.

Modern machines usually have some sort of hardware support for a return address stack. In conventional machines, this support is often a stack pointer register and instructions for performing subroutine calls and subroutine returns. This return address stack is usually kept in an otherwise unused portion of program memory.

**1.3 The local variable stack**

Another problem that arises when using recursion, and especially when also allowing reentrancy (the possibility of multiple uses of the same code by different threads of control) is the management of local variables. Once again, in older languages like FORTRAN, management of information for a subroutine was handled simply by allocating storage assigned permanently to the subroutine code. This kind of statically allocated storage is fine for programs which are neither reentrant nor recursive.

However, as soon as it is possible for a subroutine to be used by multiple threads of control simultaneously or to be recursively called, statically defined local variables within the procedure become almost impossible to maintain properly. The values of the variables for one thread of execution can be easily corrupted by another competing thread. The solution that is most frequently used is to allocate the space on a local variable stack. New blocks of memory are allocated on the local variable stack with each subroutine call, creating working storage for the subroutine. Even if only registers are used to hold temporary values within the subroutine, a local variable stack of some sort is required to save register values of the calling routine before they are destroyed.

The local variable stack not only allows reentrancy and recursion, but it can also save memory. In subroutines with statically allocated local variables, the variables take up space whether the subroutine is active or not. With a local variable stack, space on the stack is reused as subroutines are called and the stack depth increases and decreases.

**1.4 The parameter stack**

The final common use for a stack in computing is as a subroutine parameter stack. Whenever a subroutine is called it must usually be given a set of parameters upon which to act. Those parameters may be passed by placing values in registers, which has the disadvantage of limiting the possible number of parameters. The parameters may also be passed by copying them or pointers to them into a list in the calling routine's memory. In this case, reentrancy and recursion may not be possible. The most flexible method is to simply copy the elements onto a parameter stack before performing a procedure call. The parameter stack allows both recursion and reentrancy in programs.

**1.5 Combination stacks**

Real machines combine the various stack types. It is common in register-based machines to see the local variable stack, parameter stack, and return address stack combined into a single stack of activation records, or "frames." In these machines, expression evaluation stacks are eliminated by the compiler, and instead registers are allocated to perform expression evaluation.

The approach taken by the stack machines described later in this book is to have separate hardware expression evaluation and return stacks. The expression evaluation stacks are also used for parameter passing and local variable storage. Sometimes, especially when conventional languages such as C or Pascal are being executed, a frame pointer register is used to store local variables in an area of program memory.