Push-down Stack Architecture to a Minicomputer Interface

Shozoh Ohdate,* Kenji Yamashita* and Chiaki Hishinuma*

Abstract

This paper presents a new architecture of push-down stack which is implemented by putting some hardware to the interface of a minicomputer HITAC-10. The major advantage of this stack is that it is not a merely last-in-first-out buffer, but it is designed to facilitate a recursive programming. In order to make it possible, the top four cells of the stack are constructed with accessible registers, and the other cells lie in the successive locations of the main memory. Further, in this stack, both overflow and underflow of data from the specified portion of the main memory are detected automatically as I/O interrupts, so that it is possible to construct the software routine independently which can save or unsave some of the blocks of the stack in the main memory to/from the secondary memory. The method for saving and unsaving the stack is also described.

1. Introduction

In a list processing or non-numerical problems with a complex construction, the procedure is in general characterized by the appearence of a reference to itself in its own definition. Such procedure is widely known as a "recursive procedure", which reduces the problem fundamentally to that of dealing with a subroutine that contains a call of itself.[1] In this case, there must be different places to store the arguments and a return address for each call of the subroutine. A push-down stack organization is suitable for such storage. The stack operates in the last-in-first-out (LIFO) principle; that is, all data are loaded or "pushed" into a stack in sequential order and retrieved or "poped" from the stack in reverse order.

This push-down stack concept has been used successfully in large-scale computers like Burroghs 5000,[2] however, its concept is efficiently applied in a minicomputer which uses the main memory and registers in a very constrained way.

This paper first appeared in Japanese in Joho-Shori (Journal of the Information Processing Society of Japan), Vol. 15, No. 11 (1974), pp. 857~862.

* Faculty of Engineering, Keio University

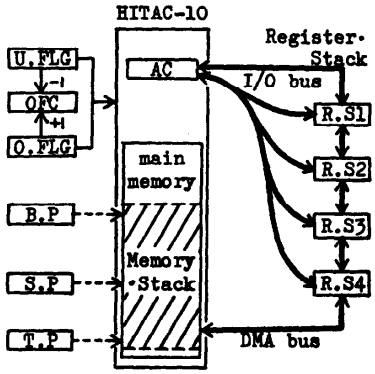


Fig.1 Push-down stack configuration with stack processor

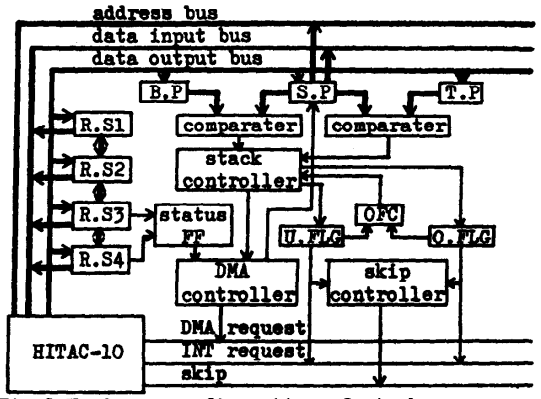


Fig.2 Hardware configuration of stack processor

For instance, PDP-11 and Micro-data 3200[3] have been adopted a stack architecture.

In this paper, we describe a system of minicomputer HITAC-10 with a push-down stack processor on which the recursive procedure can be achieved efficiently.

2. Organization of the Stack Processor

In the case of a simple LIFO stack of which only the top item is externally accessible, it is inefficient to handle a recursive procedure because of the requirement of the direct reference to items in the inner locations. For example, in the recursive language such as LISP, the efficiency of the Interpreter can be greatly increased by the use of a stack of which the top several locations are directly accessible for the modifications of the return address or parameters in the interior.

The stack processor described here is implemented by putting some hardware to the interface as shown in Fig.1. The top four cells of the stack are constructed with accessible 16-bit registers(Register Stack) and the lower portions are laid in the successive locations of the main memory(Memory Stack).

The data transfers are accomplished between the accumulator(AC) and the Register Stack with I/O bus, and the data that run over from the Register Stack are stored into the Memory Stack with DMA bus. The Memory Stack is specified by two 15-bit registers (Bottom Pointer, Top Pointer), and the current top location of the Memory Stack is specified by the 15-bit up-down counter(Stack Pointer). Both overflow and underflow of the Memory Stack are detected automatically as I/O interrupts, so that it is possible in the interrupt processing to save/unsave some of the blocks of the Memory Stack to/from the secondary memory.

3. Control of the Stack Processor

The configuration of the Stack Processor is shown in Fig.2. It is made up of the

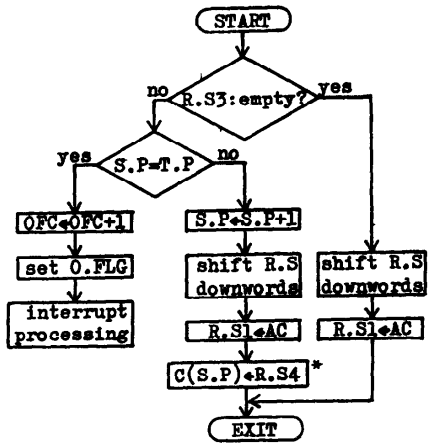
following registers. Actually, the amount of this hardware is small.

1. Register Stack(R.S1, R.S2, R.S3, R.S4)
2. Stack Pointer(S.P)
3. Bottom Pointer(B.P), Top Pointer(T.P)
4. Overflow Flag(O.FLG), Underflow Flag(U.FLG) : These flags are connected to an interrupt request bus.
5. Overflow Counter(OFC) : It is a 4-bit up-down counter, which increases by stack overflow and decreases by stack underflow.

In addition, there are some hardware to control these registers. We will describe the actions of the Stack Processor in detail with Push-down and Pop-up which are fundamental operations of the stack.

Fig.3 shows the actions of the stack during a push-down operation. As a new data enters into the Register Stack, the previously stored data are moved downwards from one register to the next in the stack. If it is already full, the content of the S.P is incremented by 1 and then a datum in the R.S4 is transferred by means of DMA into the memory location that is specified by the content of the S.P, i.e., the R.S4 is used as data buffer register, too. Otherwise, there are no increment of the S.P and no transfer of data into the Memory Stack. Detection of stack overflow is checked by comparing the S.P with the T.P just prior to these actions. If stack overflow is detected, there are no above actions but the OFC is incremented by 1 and the O.FLG is set for saving the stack.

Fig.4 shows the actions of the stack during a pop-up operation. First, a datum



* C(S.P) means the memory location that is specified by the S.P.

Fig.3 Flow-chart of push instruction

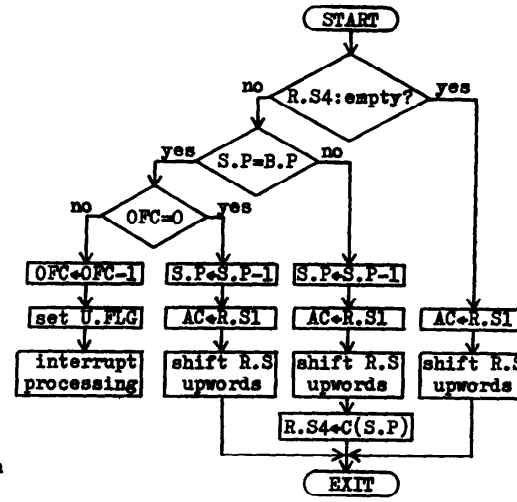


Fig.4 Flow-chart of pop instruction

in the R.S1 is transferred to the accumulator, and the data in the Register Stack are moved upwards. At this time, if there are still remaining data in the Memory Stack, the content of the S.P is decremented by 1 and then a datum in the memory location that is specified by the content of the S.P is transferred by means of DMA into the R.S4. Otherwise, i.e., in case the stack is used merely within the Register Stack, there are no decrement of the S.P and no access to main memory. Full control of these DMA is provided by the status flip-flop which indicates the occupancy of the R.S3 and the R.S4. Stack underflow is detected only when S.P=B.P and OFC≧1 because of the requirement for unsaving the stack, which will be described in further detail in section 4. The Stack Processor may be controled in the same as other peripheral devices with I/O instructions described below in addition to PUSH and POP.

1. Set Stack Pointer and Clear (SSPC) : Set the content of the AC to the S.P and clear the interrupt flags.
2. Set Stack Bottom Pointer (SSBP), Set Stack Top Pointer (SSTP) : Set the content of the AC to the B.P and the T.P, respectively.
3. Put (PUT1, PUT2, PUT3, PUT4) : Set the content of the AC to the R.S1, R.S2, R.S3 and R.S4, respectively. PUT4 rewrites the memory location specified by the S.P.
4. Get (GET1, GET2, GET3, GET4) : Load the content of the R.S1, R.S2, R.S3 and R.S4 into the AC, respectively.
5. Skip Stack Over-Flow (KSOF), Skip Stack Under-Flow (KSUF) : Skip a next instruction if the Stack Processor is in the overflow or underflow condition, otherwise, these instructions are ignored.
6. Get Stack Pointer (GSP) : Load the content of the S.P into the AC.

4. Saving/Unsaving of the Stack

It is undesirable that the Memory Stack occupies too much portion of the memory. Since most of the traffic in and out of the stack involves only the top few locations, this is absorbed by the following way: the deep portions of the stack that are unnecessary for the present to be saved temporarily into the secondary memory, and to be unsaved into the main memory as they become necessary.

Fig.5 shows the stack condition just before overflow. If a push-down operation is executed in this condition, an interrupt of stack overflow occurs as described in the previous section. In the interrupt processing, the saving routine accomplishes to save the contents of the Memory Stack into the secondary memory and executes a push-down operation again, thus it is possible to resume the interrupted program.

At this time, it is very significant that the whole Memory Stack is not saved but only deep portions of the Memory Stack is saved, because the data in the stack are pushed-down and poped-up continuously. Fig.6 shows the stack condition after saving the stack. If the stack underflow is detected, just the reverse operations of the saving are accomplished in the unsaving routine. Also at this time, data are unsaved with some room from the secondary memory in the same reason as the saving. When OFC=0, stack underflow is not detected because there is no block of the stack in the secondary memory, but only the Register Stack operates as the stack since there are available data in it yet.

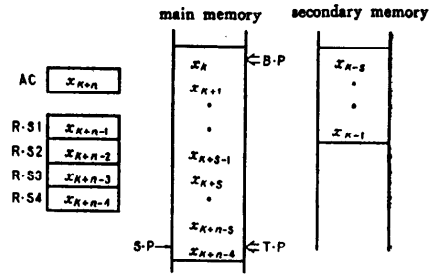


Fig.5 Stack condition before overflow of the stack

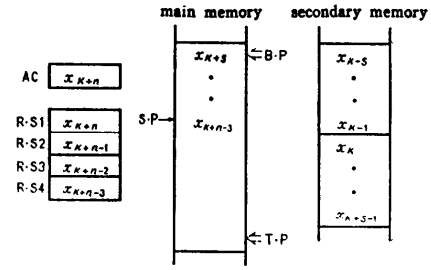


Fig.6 Stack condition after saving of the stack

5. Conclusion

The Stack Processor described in this paper has no provision for arithmetics operations on the top two elements and for the Subroutine Jump Nesting Store, because it is implemented to the minicomputer's I/O interface. However, it provides some convenient functions for the process with the frequent recursive call, and we could hold in the costs of attached components relatively low.

With this Stack Processor, a MINI-LISP system could be implemented on a minicomputer which has 4k words main memory. Furthermore, LISP programs run between 30 and 34% faster than the same programs under a software stack.[4)]

Acknowledgement

The authors wish to thank Dr.H.Aiso and Mr.M.Nakanishi for their helpful advices concerning this work.

References

1) D.W.Barron: Recursive Techniques in Programming, Macdonald Computer Monographs

2) C.B.Carlson: The mechnization of a push-down stack, Proc. of FJCC, p.243 (1963)

3) R.Burns & D.Savitt: Microprograming, stack architecture ease minicomputer programer's burden, Electronics, Vol.46, No.4, p.95 (1973)

4) K.Yamashita, S.Ohdate, C.Hishinuma: A MINI-LISP system with hardware stack, IPSJ SIGARCH, (Feb. 1975)