A COMBINED REGISTER-STACK ARCHITECTURE

Richard L. Sites
Computer Science Division
Dept. of Applied Physics and Information Science
University of California, San Diego
La Jolla, California 92093

A stack is the simplest mechanism for evaluating arithmetic expressions, while a group of general registers is the simplest mechanism for quick access to a small number of common subexpressions or loop control variables. Until now, computers have been designed around either a stack mechanism or a register mechanism, but no machine has included both in a convenient way. In this paper, we propose a new combination architecture in which there are general registers, but one of these registers is the top of an evaluation stack. This combination is intended entirely for calculation, and is not related to the use of stacks in main memory for holding activation records of Algol-like procedures, or to the use of stacks for holding subroutine linkage information. The mechanism can be implemented entirely in a fixed number of hardware registers, with no elaborate stack spilling mechanism. The combination architecture meshes perfectly with existing general-register architectures, while providing the advantages of a stack for expression evaluation. One way to view the combination architecture is that it provides access to about twice as many fast registers with no increase in the number of instruction bits needed to address them.

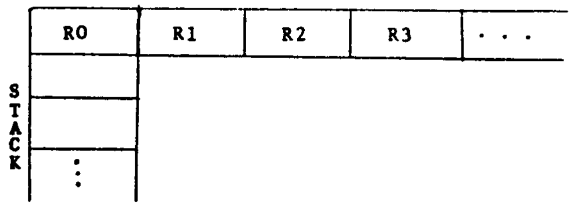

Figure 1. Combination architecture, with standard general registers R1-Rn, and R0 defined to be the top of an expression evaluation stack.

Assuming that R0 is the stack register, the expression A := B - C would be evaluated entirely in the stack as:

```
                                  stack depth after
   Load   R0←B                          1
   Load   R0←C                          2
   Sub    R0←R0-R0                      1
   Store  R0→A                          0
```

As a more complicated example, consider the complex product of A * B. In a straight-forward evaluation, the real and imaginary parts (Ar and Ai, respectively) of each number are fetched twice, for a total of eight memory references instead of the minimum four. On a straight stack machine, little improvement is possible, while on a straight register machine, at least five registers are needed to do the evaluation without refetching any operands. On the combination architecture, only four registers are needed, as shown below, and an evaluation with only two extra fetches can be done with two registers.

```
                                  stack depth after
   Load   R1←Ar                         0
   Load   R2←Ai                         0
   Load   R3←Br                         0
   Mpy    R0←R2*R3   Ai*Br              1
   Mpy    R0←R1*R3   Ar*Br              2
   Load   R3←Bi                         2
   Mpy    R0←R2*R3   Ai*Bi              3
   Sub    R0←R0-R0   Ar*Br-Ai*Bi        2
   Store  R0→PRODr                      1
   Mpy    R0←R1*R3   Ar*Bi              2
   Add    R0←R0+R0   Ai*Br+Ar*Bi        1
   Store  R0→PRODi                      0
```

The combination architecture is easy to compile code for -- if an operand or result is to be used only once, put it into the stack register; otherwise put it into one of the other registers. Common stack-machine operations such as DUPL, SWAP, and POP are not needed, although it is convenient to supply unary operators Rx←Rx*Rx and Rx←Rx+Rx, so that the top of stack can be squared or doubled.

If a stack register is grafted into an existing architecture, only small changes are needed when saving registers across subroutine calls or interrupts. Most often, subroutines and functions can be called with the stack empty, so saving and restoring is no problem. If the stack isn't empty, or if the entire machine state must be saved on interrupt, then a series of Stores from the stack register will pop out all the elements, and a series of Loads will restore them.

The stack register can be used with existing 2-address and 3-address register architectures, but it also can be used quite effectively with a one-address architecture consisting of (opcode, flag, register #). The opcode and register # are standard, and the flag is two bits, specifying one of four operations:

```
1. Stack←Stack op Reg     (Reg 0 = Stack)
2. Stack←Reg   op Stack   (allows reverse DIV, SUB)
3. Reg  ←Stack op Stack   (create common expression)
4. Reg  ←Reg   op Stack   (update loop variable)
```

These arithmetic instructions would be augmented with Moves to/from main memory or a register.

The stack register can either be defined to wrap around (pushing the ninth element into an 8-deep stack overwrites the first element), or it can be defined to generate an error trap on overflow/underflow. The error trap can either be interpreted strictly as an error, or a software routine can move half the stack to/from a backup area in main memory.

For expression evaluation, only one stack is needed, but the concept could be extended to make all the other registers stacks which are pushed and popped only at subroutine calls. For example, 16 registers built as 16-deep stacks each could be used; R0 is an evaluation stack and its depth fluctuates while the other 15 registers stay fixed; at a subroutine call, R1-R15 are pushed down one element and popped on return. This could be combined with overflow/underflow interrupts (one for R0, and a separate one for R1-R15), so that no register save/restore is done at all in the normal user code, but the interrupt routine would dump blocks of registers if the subroutine call level reached 16, or if an especially complicated arithmetic expression reached an evaluation depth of 16.

Conclusion

A combination architecture can be implemented easily and cheaply these days, and it makes expression evaluation, register allocation, and subroutine linkages more straightforward.