Subroutine Linkage

When a subroutine or function is called, control passes to that subroutine but must return to the instruction immediately following the call when the subroutine exits.

There are two main issues in the design of a calling mechanism for subroutines and functions. These fall under the heading **"subroutine linkage"**.

1. How to pass the arguments to the subroutine.
2. How to pass the return address to the subroutine so that, upon completion, it returns to the correct address.

A function is just a subroutine that returns a value. For functions, we have one additional issue in the linkage discussion: how to return the function value.

This presentation will be a bit historical in that it will pose a number of linkage mechanisms in increasing order of complexity and flexibility.

We begin with a simple mechanism based on early CDC–6400 FORTRAN compilers.

We continue with a slightly more complex mechanism as found on later CDC–6400 FORTRN compilers.

We end with a general mechanism that allows the use of recursion. This is the mechanism used by the Boz–5.

Pass–By–Value and Pass–By–Reference

Modern high–level language compilers support a number of mechanisms for passing arguments to subroutines and functions.

Two of the most common mechanisms are:

1. Pass by value, and
2. Pass by reference.

In the pass–by–value mechanism, the value of the argument is passed to the subroutine.

In the pass–by–reference, it is the address of the argument that is passed to the subroutine, which can then modify the value and return the new value.

At this stage in the development of the Boz–5, I have decided not to worry about the mechanism of argument passing.

The requirement is that a 32–bit word (value or address) be passed for each argument, and that the compiler can have a consistent mechanism for handling the arguments.

Argument Passing: Version 1
(Based on Early CDC–6400 FORTRAN)

Pass the arguments in the general–purpose registers.

Register %R0 continues to be defined as the constant 0.

Registers %R1 through%R6 are used to pass six arguments.

Register %R7 is used to return the value of a function.

This is a very efficient mechanism for passing arguments.

The problem arises when one wants more than six arguments to be passed.

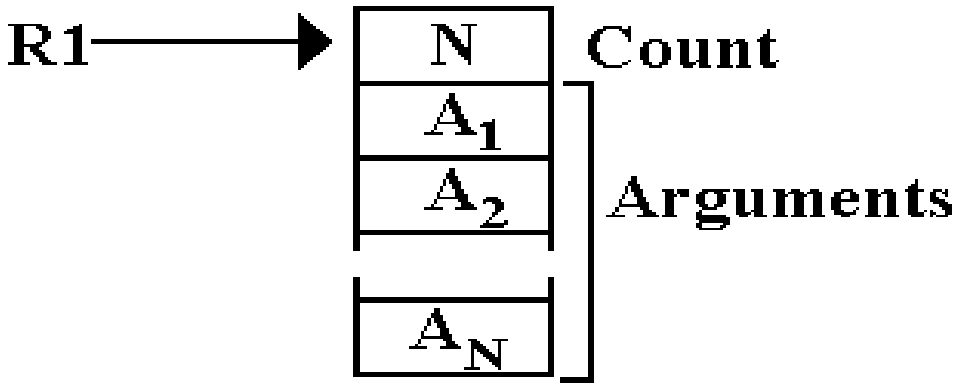
It is likely that the desire to use data plotting routines, such as those marketed by Calcomp for use with its flat–bed plotters, drove the change from this method.

Argument Passing: Version 2
(Based on Later CDC–6400 FORTRAN)

In this design, only two registers would be reserved for subroutine linkage.

%R1 This points to a memory block containing the number of arguments and an entry (value or address) for each of the arguments.

%R7 This will be reserved for the return value of a function.



This method allows for passing a large number of arguments.

This method can be generalized to be compatible with the modern stack–based protocols.

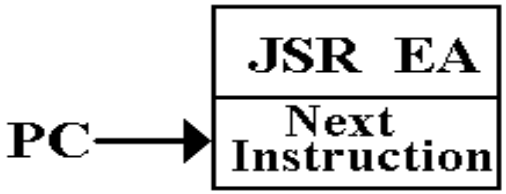
Calling Context for the JSR

In order to understand the full subroutine calling mechanism, we must first understand its context. We begin with the situation just before the JSR completes execution.

In this instruction, we say that EA represents the address of the subroutine to be called.

The last step in the execution of the JSR is updating the PC to equal this EA.

Prior to that last step, the PC is pointing to the instruction immediately following the JSR.



The execution of the JSR involves three tasks:

1. Computing the value of the Effective Address (EA).
2. Storing the current value of the Program Counter (PC) so that it can be retrieved when the subroutine returns.
3. Setting the PC = EA.

The effective address will be the address of the first instruction in the subroutine.

A Simple Mechanism for Return
(How CDC–6400 FORTRAN did it)

The simplest method for storing the return address is to store it in the subroutine itself.

This mechanism allocates the first word of the subroutine to store the return address.

If the subroutine is at address Z in a word–addressable machine such as the Boz–5, then

Address Z	holds the return address.
Address (Z + 1)	holds the first executable instruction of the subroutine.
BR *Z	An indirect jump on Z is the last instruction of the subroutine. Since Z holds the return address, this affects the return.

This is a very efficient mechanism.

The difficulty is that it cannot support recursion.

Example: Non–Recursive Call

Suppose the following instructions

100	JSR 200
101	Next Instruction
200	Holder for Return Address
201	First Instruction
Last	BR *200

After the subroutine call, we would have

100	JSR 200
101	Next Instruction
200	101
201	First Instruction
Last	BR *200

The BR*200 would cause a branch to address 101, thus causing a proper return.

Example 2: Using This Mechanism Recursively

Suppose a five instruction subroutine at address 200.

Address 200 holds the return address and addresses 201 – 205 hold the code.

This subroutine contains a single recursive call to itself that will be executed once.

Called from address 100	First Recursive Call	First Return
200 101	200 204	200 204
201 Inst 1	201 Inst 1	201 Inst 1
202 Inst 2	202 Inst 2	202 Inst 2
203 JSR 200	203 JSR 200	203 JSR 200
204 Inst 4	204 Inst 4	**204 Inst 4**
205 BR * 200	205 BR * 200	205 BR * 200

Note that the original return address has been overwritten.

As long as the subroutine is returning to itself, there is no difficulty.

It will never return to the original calling routine.

Use a Stack to Hold Return Addresses

With the code above, we assume that a stack holds the return addresses.

Main calls the subroutine SP $\rightarrow$ 101

The subroutine calls itself SP $\rightarrow$ 204 $\rightarrow$ 101

First return SP $\rightarrow$ 101

The subroutine returns to itself.

Second return

The subroutine returns to the main program.

Our design will use the following convention.

JSR will push the return address to the stack.

RET will pop the return address from the stack.

Implementation of the Stack Operations

Arbitrarily, I have decided that the stack grows toward more positive addresses.

Given this we have two options for implementing PUSH,
each giving rise to a unique implementation of POP.

Option	PUSH X	POP Y	
1	M[SP] = X	SP = SP – 1	// Post–increment on PUSH
	SP = SP + 1	Y = M[SP]	
2	SP = SP + 1	Y = M[SP]	// Pre–increment on PUSH
	M[SP] = X	SP = SP – 1	

The constraints on memory access dictate the first option.

Post–increment on PUSH must be paired with pre–decrement on POP.

The operation M[SP] = X corresponds to a memory write. The latest time at which this can be done is (E, T2), due to the requirement of a wait cycle before (F, T0).

If (E, T2) corresponds to M[SP] = X,

then (E, T3) can correspond to SP = SP + 1. This does not affect memory.

Control Signals for the JSR

Here are the control signals for the complete JSR.

JSR Op-Code = 01110 (Hexadecimal 0x0E)

F, T0: PC → B1, **tra1**, B3 → MAR, READ. // MAR ← (PC)
F, T1: PC → B1, 1 → B2, **add**, B3 → PC. // PC ← (PC) + 1
F, T2: MBR → B2, **tra2**, B3 → IR. // IR ← (MBR)
F, T3: IR → B1, R → B2, **add**, B3 → MAR. // Do the indexing.

D, T0: READ.
D, T1: WAIT.
D, T2: MBR → B2, tra2, B3 → MAR. // MAR ← (MBR)
D, T3: WAIT.

// At this point, the MAR has the target address for the subroutine.
// the SP points to the top of the stack.
// the PC contains the return address.

E, T0: PC → B1, **tra1**, B3 → MBR. // Put return address in MBR
E, T1: MAR → B1, **tra1**, B3 → PC. // Set up for jump to target.
E, T2: SP → B1, **tra1**, B3 → MAR, WRITE. // Put return address on stack.
E, T3: SP → B1, 1 → B2, **add**, B3 → SP. // Bump SP

Analysis of Execute Phase of JSR

The goals of JSR are
1) to get the subroutine address into the PC, and
2) to store the old value of the PC on the stack, so that it can be used for the return.

In order to place the PC on the stack, we must copy PC $\rightarrow$ MBR and SP $\rightarrow$ MAR.

But note that the MAR contains the address that must go into the PC.
It cannot be overwritten by the SP until the PC is updated.

E, T0: PC $\rightarrow$ B1, **tra1**, B3 $\rightarrow$ MBR. // Place the old PC into the MBR
This saves the old value of the PC into the MBR, from whence it will be written onto the stack in (E, T2). This will be the return address.

E, T1: MAR $\rightarrow$ B1, **tra1**, B3 $\rightarrow$ PC. // Set up for jump to target.
With the old value of the PC saved, we can now place the subroutine address into the PC. Placing an address in the PC causes the instruction at that address to be executed next; the subroutine is started.

E, T2: SP $\rightarrow$ B1, **tra1**, B3 $\rightarrow$ MAR, WRITE. // Put return address on stack.
The stack pointer is used to address memory and store the old value of the PC, already stored in the MBR.

E, T3: SP $\rightarrow$ B1, 1 $\rightarrow$ B2, **add**, B3 $\rightarrow$ SP. // The stack pointer is incremented.

Control Signals for the RET

This instruction just pops an address from the stack and places it into the PC.

RET Op-Code = 01010 (Hexadecimal 0x0A)

F, T0: PC → B1, **tra1**, B3 → MAR, READ. // MAR ← (PC)
F, T1: PC → B1, 1 → B2, **add**, B3 → PC. // PC ← (PC) + 1
F, T2: MBR → B2, **tra2**, B3 → IR. // IR ← (MBR)
F, T3: WAIT

E, T0: SP → B1, – 1 → B2, add, B3 → SP. // Decrement the SP
E, T1: SP → B1, tra1, B3 → MAR, READ. // Get the return address
E, T2: WAIT.
E, T3: MBR → B2, **tra2**, B3 → PC. // Put return address into PC