Procedure Linkage

Calling Convention

The calling code must know the name of the called procedure and it must have access to that name. The procedure is invoked with the assembly statement: CALL *name*. For this to work the CALL statement and called procedure must be in the same file (internal linkage) or the called procedure must be declared to have global scope and the file with the CALL statement must declare the procedure to be external, meaning it is defined in another file (external linkage). The procedure should end with a RET assembly language instruction.

Procedure Argument Passing

Methods of Passing Arguments

	Pass-By-Value	**Pass-By-Reference**
Description	pass a copy of the variable (its value)	pass the actual variable (its address or the register containing the variable)
Advantages	little interaction between the calling code and the code in the called procedure easy to understand no side-effects possible	no need to make a copy (may be faster/uses less memory) can change caller's local variables (usually dangerous)
Disadvantages	may be slower than pass-by-reference may use more memory than pass-by-reference	can have unplanned and difficult to debug interactions between the calling code and the code in the called procedure makes code difficult to understand
When to Use	whenever possible	large data structures (arrays, structures, etc.) need to change caller's local variables (*rare*)

Where to Pass Arguments

	Register	**Stack**	**Fixed Memory Location**
Description	value or reference is passed in a register or registers	value or reference is stored on stack before calling procedure typically use PUSH to store arguments and base addressing (Y + n) to access arguments	value or reference is stored at a fixed (known and global) memory location before calling the procedure
Advantages	fast easy	dynamic use of storage lots of space available	easier and possibly faster access than stack lots of space available
Disadvantages	limited number of registers	slower access than registers more complicated access than registers	slower access than registers static (continuous) use of memory code may not be re-entrant
When to Use	whenever possible	when have many arguments high-level language interfaces	*rare*

Procedure Return Values

Methods of Returning Values from Procedures

	Return-By-Value	**Return-By-Reference**
Description	return the value of the result(s) / return value(s)	return the address of the result(s) / return value(s)
Advantages	simple to interface with easy to understand no memory allocation problems (memory leaks)	easy to hold *large* multi-byte return value(s)
Disadvantages	difficult for *large* multi-byte values	makes code difficult to understand must statically allocate memory for return value(s)
When to Use	whenever possible	large data structures (arrays, structures, etc.)

Where to Return Value(s)

	Register	**Stack**	**Fixed Memory Location**
Description	return value(s) or reference(s) to return value(s) is/are returned in a register or registers	return value(s) or reference(s) to return value(s) is/are stored on stack before returning typically use base addressing (`Y + n`) to store the return value(s) and `POP Rn` in the calling procedure to retrieve it	return value(s) or reference(s) to return value(s) is/are stored at a fixed (known and global) memory location before returning from the procedure
Advantages	fast easy	lots of space available	easier access than stack lots of space available
Disadvantages	limited number of registers	very complicated stack manipulation may need to pre-allocate stack space	static (continuous) use of memory (must be allocated at assembly time) code may not be re-entrant
When to Use	whenever possible	when returning *large* return value(s) by value or when returning many return values *rare*	when returning *large* return value(s) by value or when returning many return values