The Macro Assembler

What is the macro assembler?

- The macro assembler allows for more complex operations to be implemented
- Can be though of as a way to add complex instructions to the machine code
- The macros are a bit more than direct substitution of instructions as they can perform some basic operations on symbols in addition to outputting specific sequences of instructions
- We will use the macro assembler to create abstractions for stack-based operations and procedure calls

Basics, defining a macro

- Macro commands begin with a '$' character
- A macro definition begins with `$def` and ends with `$end`
  - Example, defining a macro that implements D=2*D

```
$def doubleD
A=D
D=D+A
$end
```

  - Usage:

```
@12
D=A
$doubleD // instantiate macro called doubleD
// D is now 24
```

Defining Macros with internal symbols

- Macros can be defined with internal symbols that are unique to the macro
- Example

```
$def halt
(HALT)
@HALT
0;JMP
$end
```

- Everytime this macro is instantiated a unique name will be generated for the symbol HALT so that there is no name so there is no name collision
- Usage:

```
$halt
```


```
(halt0.HALT)
@halt0.HALT
0;JMP
```

Macros can have arguments that can be integers or symbols

- Example

```
$def getArray address index
@index
D=A
@address
A=M
A=A+D
D=M
$end
```

- Usage

```
$getArray data 5
```

Expands to

```
@5
D=A
@data
A=M
A=A+D
D=M
```

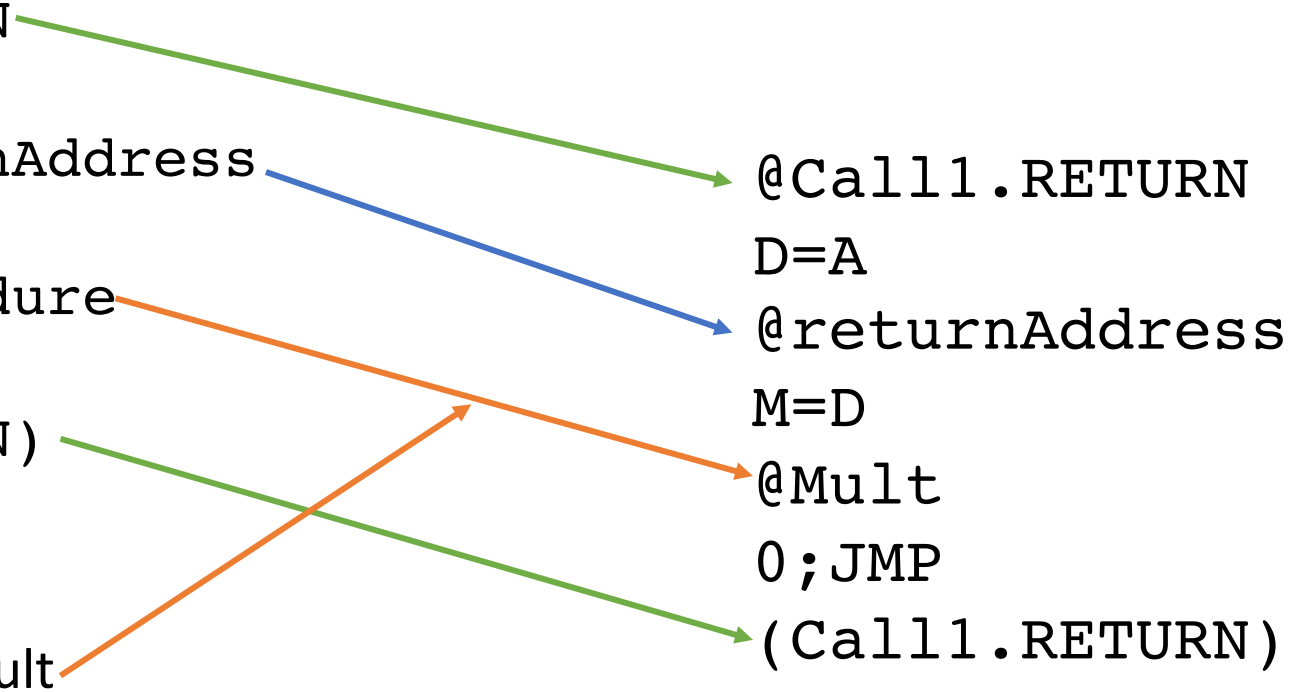
Macros can also access static variables

- Example of a crude subroutine call macro

```
$def Call procedure
@RETURN
D=A
@returnAddress
M=D
@procedure
0;JMP
(RETURN)
$end
```

- Usage:
  - $Call Mult

```
@Call1.RETURN
D=A
@returnAddress
M=D
@Mult
0;JMP
(Call1.RETURN)
```



Other features and suggestions

- The macro assembler also provides a $include facility which allows macros to be stored separate files and reused between programs
  - Implemented using

    ```
    $include utilities.h
    ```

  - The above example will read in the file utilities.h
  - Note, no quotes in above
- Tips
  - In a complex program it is easy to accidentally create unintentional static variables.
  - Inspect the output of the Assembler for error messages and inspect the static variables to make sure that they were all intentionally generated.

The MacroAssembler Programming Toolkit

- Stack Operators
  - `pushD`
  - `pushA`
  - `popAD`
  - `setPTR`
  - `getPTR`
- Procedure Management
  - `procedureCall nargs procedure`
  - `return`
  - `pushFrame nargs nlocals`
  - `popFrame nargs nlocals`
- Variable Accessors
  - `getLocal id`
  - `setLocal id`
  - `getArgument id`
  - `setArgument id`
- Operators
  - `add`
  - `sub`
  - `not`
  - `neg`
  - `eq`
  - `lt`
  - `gt`
  - `and`
  - `or`

Building the Toolkit

- Each of the elements of the toolkit can be constructed using macros
- This will be project 7
- For this implementation, the stack will build down from high memory addresses like is typically used in real architectures.
- The requirements for each will be outlined in the following slides

Basic Stack Operations

- `$pushD`
  - This macro will push the D register onto the stack
  - This macro has no arguments
  - The stack pointer (SP) will be used in this implementation
  - This call will save D on the stack and then decrement SP by one
  - After this call the D register will remain unchanged so that to subsequent pushD operations will push the same value on the stack twice
- `$pushA`
  - Similar to pushD except that the A register is stored on the stack
  - The A register is not preserved by this operation

Basic Stack Operations

- `$popAD`
  - This operation will pop a value off of the stack and place this in the A and D registers. The stack pointer (SP) will be incremented after this operation.
- `$setPTR`
  - A pointer value (ptr) is first pushed on the stack
  - Then a value (value) to be stored at the pointer location is pushed on the stack
  - Upon calling setPTR the two arguments will be removed from the stack and the operation *ptr = value is implemented
- `$getPTR`
  - A pointer value is pushed on the stack
  - This call pops the ptr off of the stack and then pushes the value in the memory location addressed by the pointer is pushed onto the stack

Stack Variable Accessor

- `$getLocal id`
  - Access local variable id where id is a number from 0 to nlocals-1 and return with this value loaded into the D register. The LCL pointer is used to peform the access.
  - Note, since the stack builds down in memory, this access is equivalent to D=*(LCL-id)
  - The stack is not involved with this operation.
- `$setLocal id`
  - Write the D register into the local variable id.
  - This is equivalent to the operation *(LCL-id) = D
  - The stack is not involved in this operation
  - D is preserved after setLocal

Stack Variable Accessor

- `$getArgument id`
  - Similar to `$getLocal` this macro implements D=*(ARG-id). The arguments are stored in reverse order due to the stack building down from high memory
  - Note this operation does not have any effect on the stack
- `$setArgument id`
  - Similar to `$setLocal`, this macro implements *(ARG-id)=D. The D register will be stored in the idth argument to the function. D will be preserved.
- Note that the setLocal, getLocal, setArgument, and getArgument use the LCL and ARG pointers that are managed by the pushFrame and popFrame macros to be discussed in the later slides

Operator Implementations

- The binary operators pop two arguments off of the stack, perform the operation and push the result on the stack.
- The unary operators pop an argument of the stack and perform the unary operation and push the result onto the stack.
- Logical operators return -1 for true and 0 for false.
- The binary operators will be discussed next with the assumption that the form of operation will be

  Push x

  Push y

  Operator (Pop x, Pop y, Push $x \oplus y$)

Binary operators

- `$add`
  - Implements x+y
- `$sub`
  - Implements x-y
- `$eq`
  - Implements x==y, result is either 0 or -1
- `$lt`
  - Implements x<y, result is either 0 or -1
- `$gt`
  - Implements x>y, result is either 0 or -1
- `$and`
  - Implements x&y
- `$or`
  - Implements x|y

Unary operators

- `$neg`
  - Implements -x
- `$not`
  - Implements !x (bitwise complement)

Basic Procedure Functionality

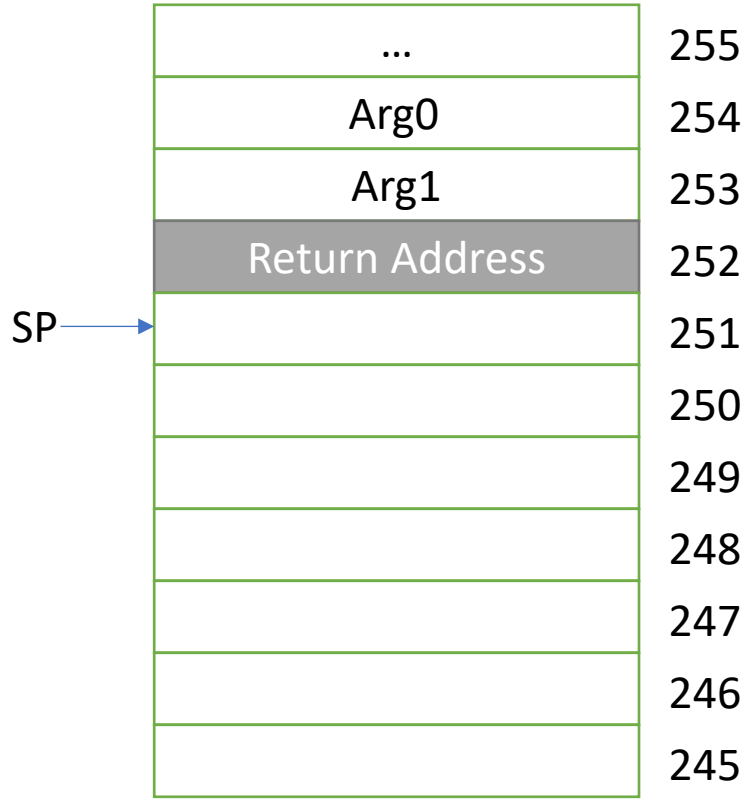
- `$procedureCall nargs procedure`
  - Pushes return address on the stack
  - Jumps to procedure
  - After return, pops nargs-1 elements from the stack
    - Caller is responsible for cleaning up arguments on the stack
    - Return value is assumed to have already been stored in the first argument on the stack
- `$return`
  - Pop return address from the stack
  - Jump to return address

Stack Frame Calls

- `$pushFrame nargs nlocals`
  - Saves a copy of LCL, ARG, THIS, and THAT on stack
  - Sets LCL to point to Local variable space allocated on the stack
  - Decrements SP by number of Local Variables
  - Sets ARG to point to the first argument passed into this procedure
- `$popFrame nargs nlocals`
  - Sets SP to to LCL which should be the state of the stack right after LCL, ARG, THIS, and that were pushed on the stack
  - Pop THAT, THIS, ARG, and LCL
  - State of the stack should be restored to same state as just before `$pushFrame` macro at beginning of procedure

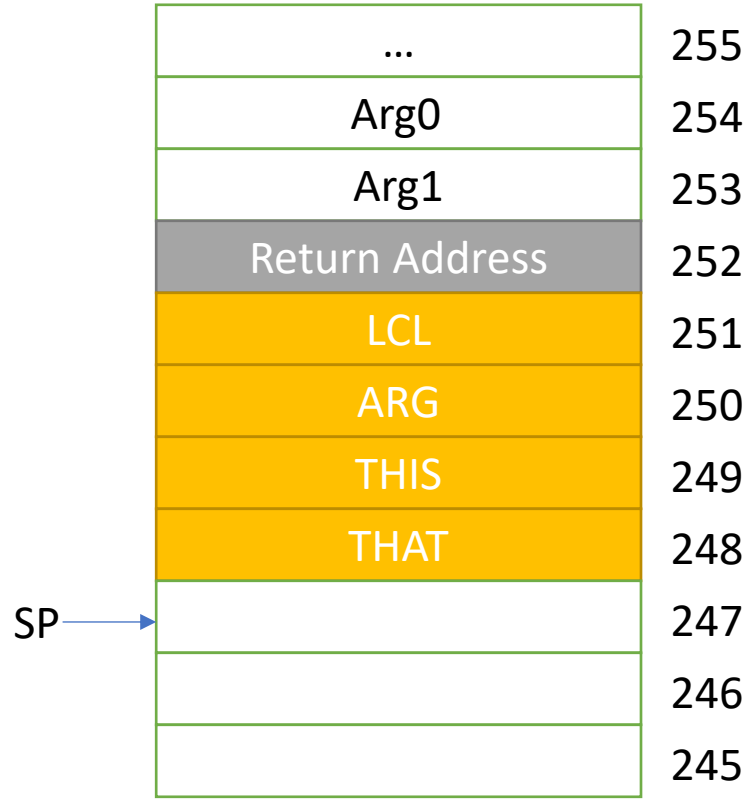
`$pushFrame` execution

- At the top of the procedure, the return address has already been pushed on the stack by the `$procedureCall`
- The next step is to save the pointers that this procedure will use to access its specific resources.
  - These are the LCL,ARG,THIS, and THAT pointers



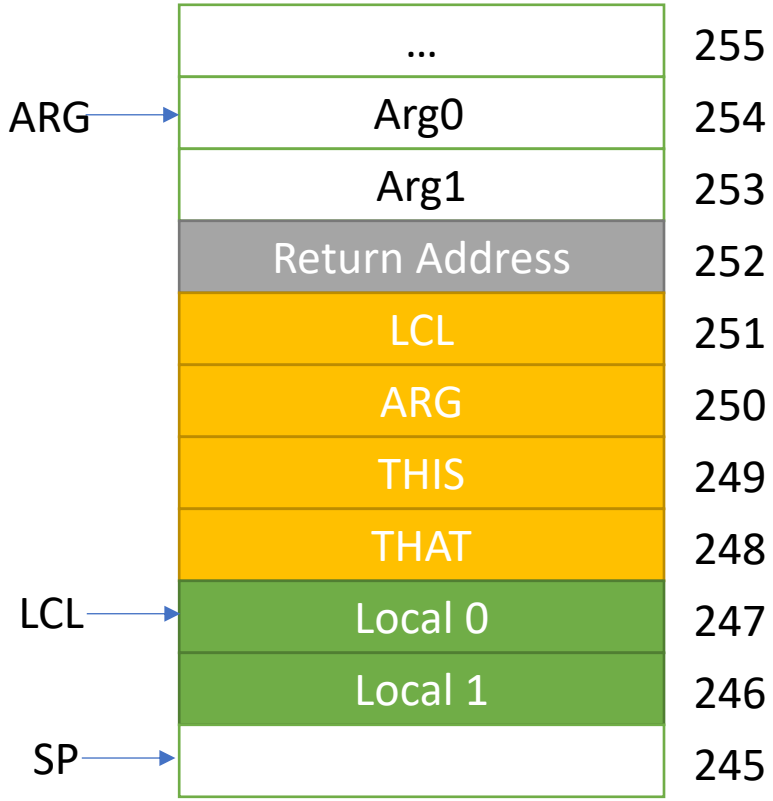
`$pushFrame` execution

- Push the LCL, ARG, THIS and THAT pointers onto the stack
- Now compute the LCL and ARG pointers for this procedure



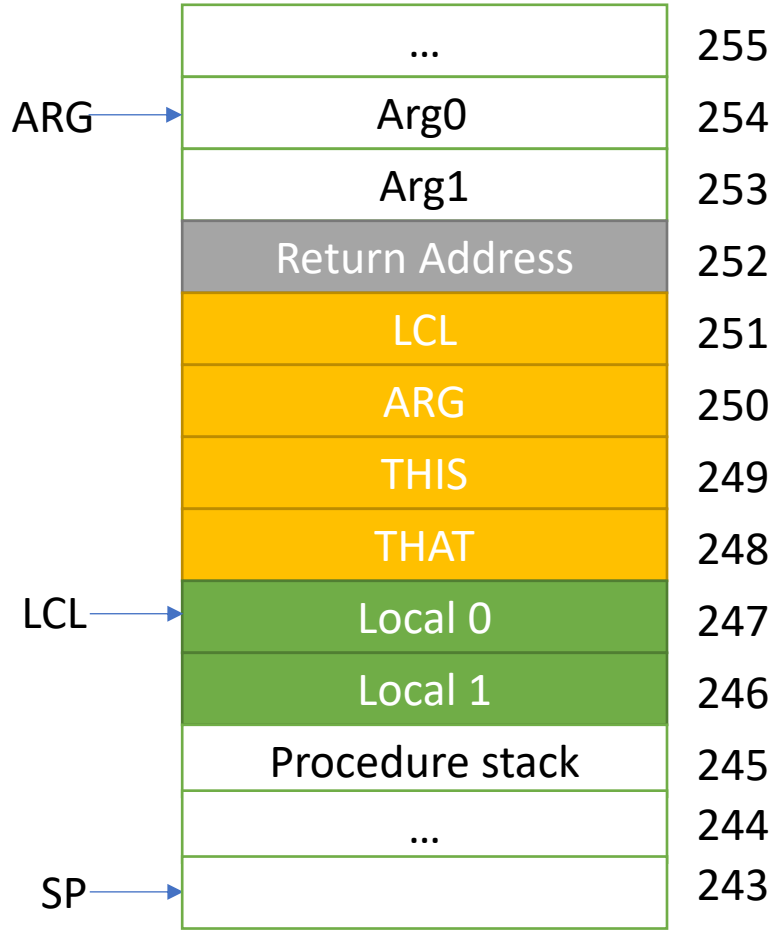
`$pushFrame` execution

- Set LCL pointer to SP
- Move SP down by nlocals to make space for local variables
- Set ARG to LCL+5+nargs to skip over stack frame and jump to first argument on the stack



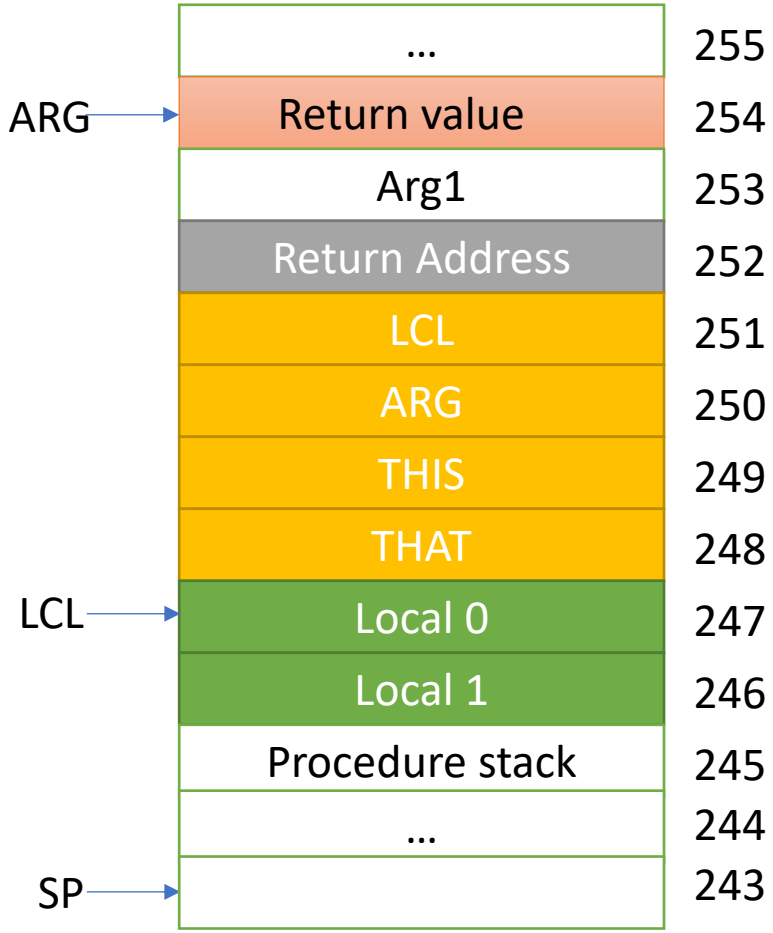
Function Execution

- As the procedure executes the the stack can be used to perform computations using the stack operations
- When a return value is computed it can simply be stored on top of Arg0 using the ARG pointer
- Before the procedure returns the stack frame will need to be cleaned up.



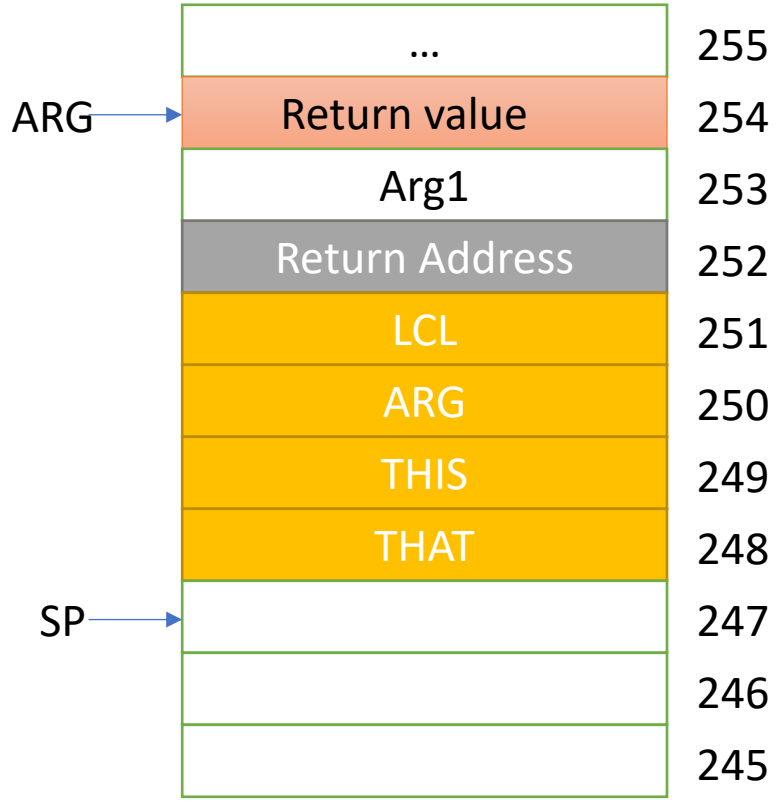
$popFrame

- Arg0 has already been replaced with return value
- There might still be items on the procedure stack
- First the SP is set to LCL to release the local variables and any remaining data on the stack



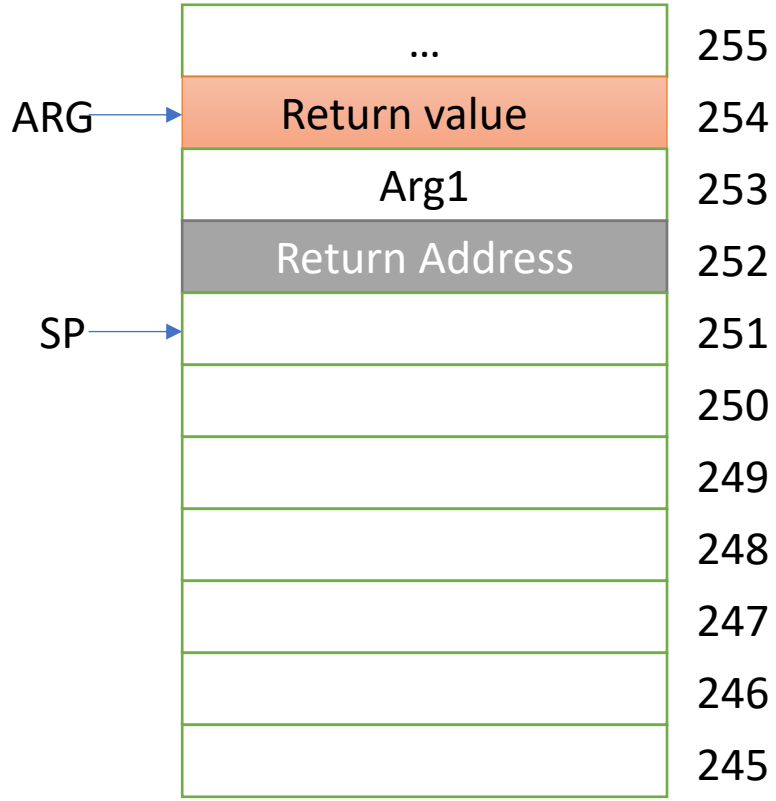
$popFrame

- Now pop THAT, THIS, ARG and LCL to restore to the calling procedure state



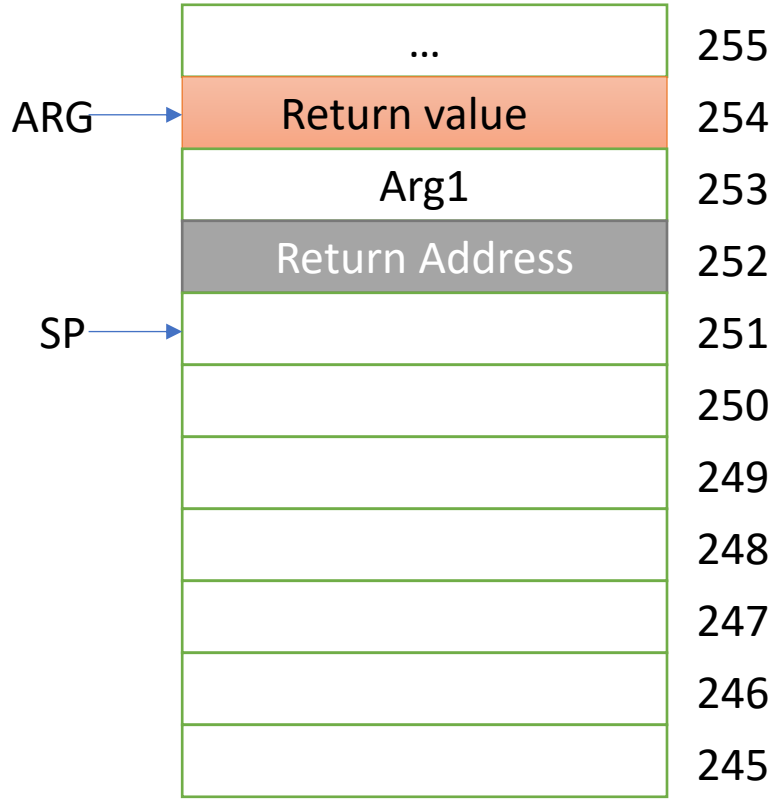
$popFrame finish

- The popFrame is now complete and the $return macro can be called to return from function



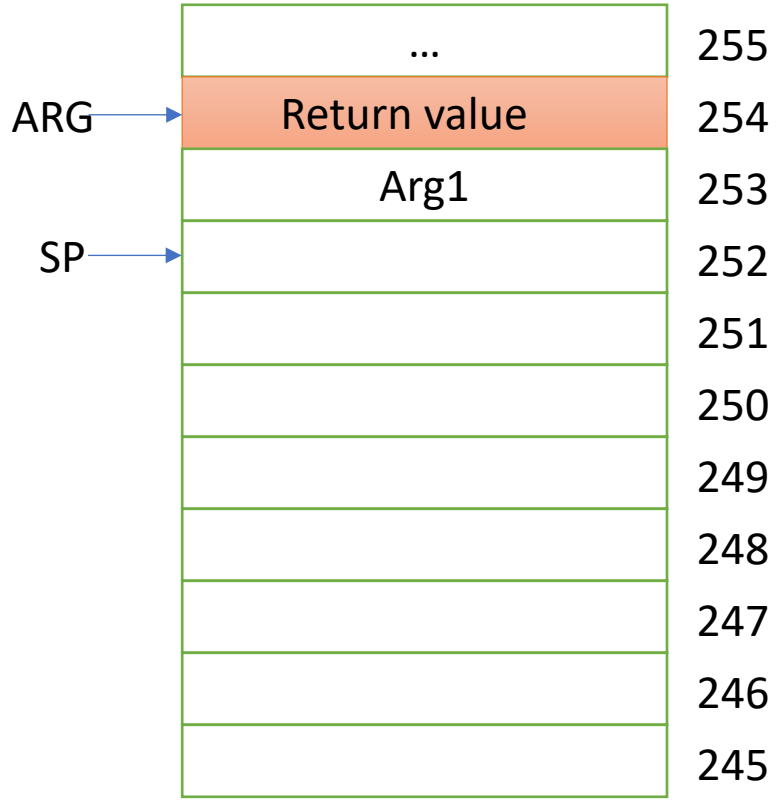
`$return`

- Pop the return address off of the stack and jump to that address



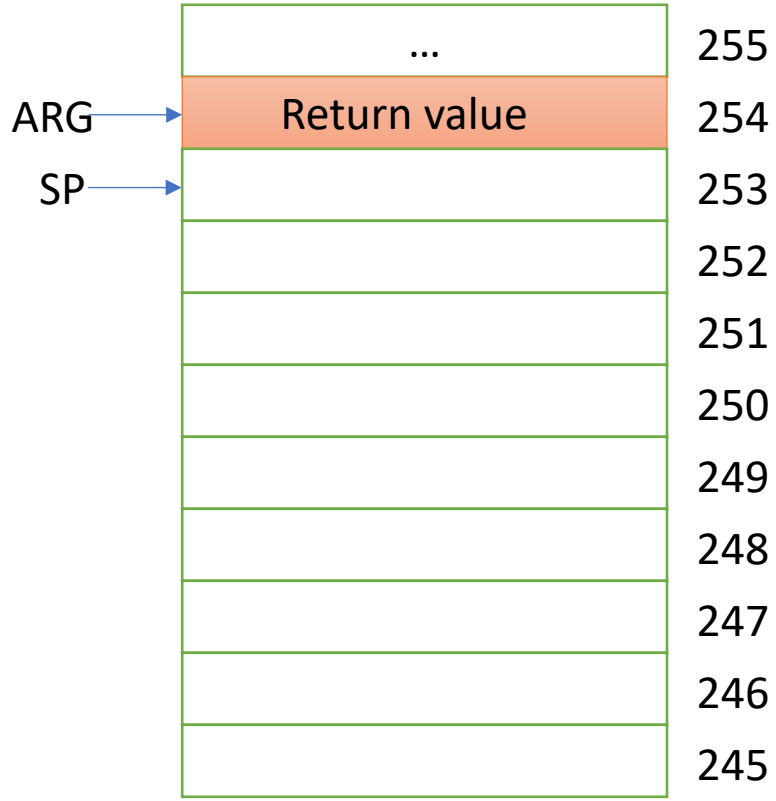
Now in $callProcedure

- Remove nargs-1 elements off of the stack to emulate having the return value pushed on the stack



`$callProcedure finished`

- The code is ready to proceed after call



Notes on macro implementations

- To implement some of the macros described in the previous slides it may be necessary to use some scratch storage to temporarily store information. It is important that this temporary scratch storage doesn’t write over something that is used by a procedure.
- By convention, we will reserve R12,R13,R14, and R15 for scratch storage for macros. If it is necessary to save some information temporarily to implement a macro, please use these memory registers.
  - (And by extension, don’t use these memory registers in any procedures that use these macros)

Bookkeeping features for symbols

- In this format, small integers will be used to refer to argument variables and local variables, this can get confusing
- Previously symbols in the assembler were either program address or static variables
- A new way to set symbols is provided in the macro assembler
  - =Symbol value
  - Example

```
=Mult.x 0
=Mult.y 1
$getArgument Mult.x
$pushD
```

Go to examples