 The material the quiz will cover is calling subroutines and techniques for passing parameters and return values.

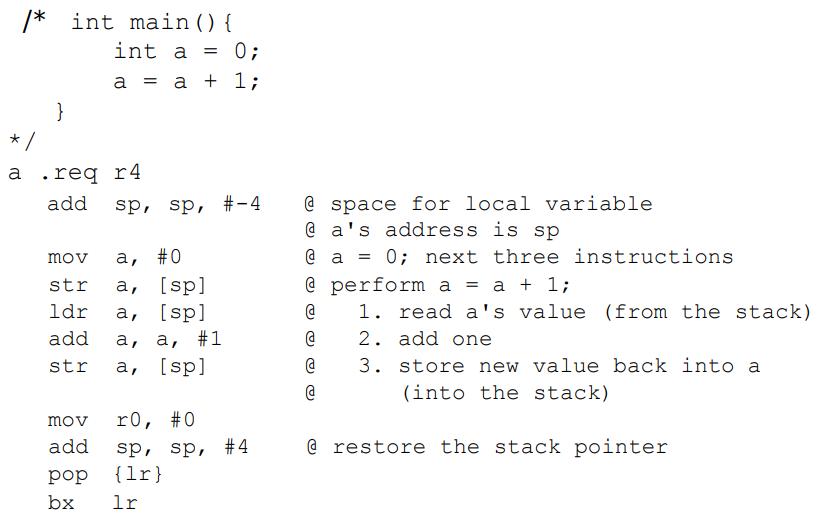
## Run-Time Environment

Different logical parts during execution:

* Stack: Automated allocated variables
  + Local variables
  + Subdivided into stack frames
  + One per procedure invocation
* Heap: Dynamically allocated variables
* Initialized data: Global and static that are initialized by programmer
  + Includes rodata: Read-Only Data
* Uninitialized data: Global and static variables initialized to zero or without explicit initialization
* Text: Program code / instructions

# Stack

* Last in, first out
* Located in high addresses in memory and grows towards small addresses
  + Creating space for local variables decrements the stack pointer
  + Subroutines must make room for local variables
* Storage for automatic variables
  + Local variables allocated at beginning of subroutine and discarded at end of call
* Stack frame
  + Collection of information related to specific instance of calling subroutine
  + Generic stack frame:
    1. Parameters
    2. Return address
    3. Registers to save
    4. Local variables
    5. Old frame pointer
* Stack pointer
  + Points to top of staff

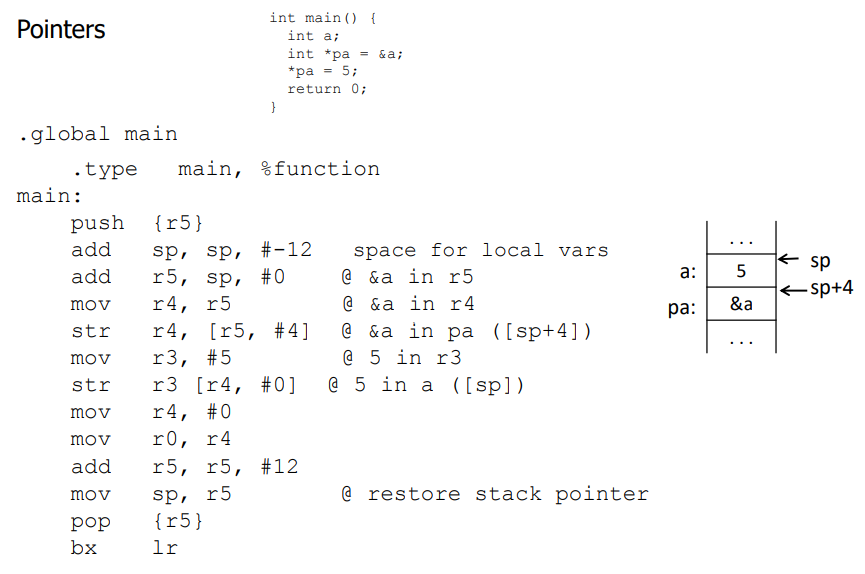


Load and store Instructions:

* LDR – load word
  + ldr rn, [sp, #4]
* LDRH – load halfword
* LDRB – load byte
* LDM – load multiple words
* STR – store word
  + str rs, [sp, #4]
* STRH – store halfword
* STRB – store byte
* STM – store multiple words

Pointers:

* Address of object In memory
* 64 bites in length
* Stored in a register or a memory word



# Parameter Passing

Call-by-value

* Used for
  + Default method for parameters in C, C++, Pascal
  + Parameters of primitive type in Java
* Copy values of actual parameters into memory locations of formal parameters before executing the body of the subroutine; do nothing on return

Call-by-result

* Do nothing prior to executing body of subroutine
* Copy final values of formal parameters into memory locations of actual parameters on return
* Advantage: Loads and stores operate directly on formal parms
* Disadvantage: Overhead

Call-by-value-result

* Perform copying of values both before and after executing body of subroutine

Call-by-reference

* Used for:
  + Array parameters in C and C++
  + Reference parameters (declared with “&”) in C++
  + Object parameters in Java
* Pass the addresses of actual parameters
* Copy these addresses into memory locations of formal parameters
* On each reference to a formal parameter into the body of the subroutine, perform indirect reference to corresponding actual parameter
  + The formal parameter is an alias of the actual parameter
  + Changes made using formal parameter are being executed on object passed as actual parameter
* Advantage: No copying, allows subr to change values of actual parms
* Disadvantage: Indirect reference through addresses in formal parms to actual parms

Subroutines in ARM

Open Subroutine = Macro

* Resolved before assembly by text substitution
* Body of routine is placed in-line at call site with parameter substitution

Closed subroutine = Standard notion

* Branch / executive / return at run time
* One copy of body which accesses its formal parameters

Transferring control from calling program to a subroutine and back

* Necessary to save return address, then branch to subroutine
* Can be accomplished in ARM using branch and link instruction bl
  + bl subr\_name
* To transfer control back to calling program, can use branch and exchange instruction bx
  + bx lr
  + Alternatively: Pop lr register into PC
* Placing a return address in link register works as long as no nested subroutines
  + Would be necessary to save return address to stack
* LIFO

Parameters:

* Actual parameters: Values or addresses passed to subroutine
* Formal parameters: Parameter names appearing in subroutine

## Function Call Bookkeeping

Registers play major role in keeping track of information for function calls

Stack also used

Temporary variables:

* Functions should first use r0 – r3 for temporary variables, do not have to be preserved
* R4 – r11 are second choice, must be saved at function and entry and restored prior to function return
  + Best method is to push registers onto stack upon entering function, pop them back before function returns
  + Important to preserve few as possible
  + For nested functions, call to internal function will overwrite return address in lr
    - External function must push content of lr at its entry and pop it back before it returns
    - Function can pop lr into pc, eliminating the bx lr instruction
* When more space required, allocate space on the stack
* Why registers or stack space:
  + Memory conservation – space will be released when function returns
  + Reentrancy – every entry of function allocates unique set of location, if function is interrupted, will ensure temporary variable reset