**Stacks**

* **Stack** – a list of data elements of a particular size with LIFO accessing
  + The stack resides in main memory
  + Top of the stack (most recent element) = lowest memory address
  + Bottom of stack (least recent element) = highest memory address
  + SP – stack pointer register points to top of the stack
    - Points to the last (topmost) occupied location
  + Push – SP is decremented and data is copied to new location pointed to by SP
  + Pop– data is copied from the location pointed to by SP and incrementing SP
  + i.e. the stack grows towards lower memory address
* **Subroutine** – block of instructions that is executed repeatedly
  + Used to:
    - Avoid duplication of code
    - Permit code reuse
    - Enable modularization
  + Program invokes subroutine by executing the call instruction
  + Subroutine executes and return to main program by executing the return instruction
  + A call is a special branch instruction
    - Store contents of PC (return address) in link register – so that program knows where to go after subroutine returns
    - Branch to target address specified by instruction
  + Return – put address in link register back into PC
  + E.g. PC = 20
    - Call SUB → PC = 100, LR = 20
    - Return → PC = 20
  + **Nesting subroutines** – e.g. sub1 calls sub2
    - Sub1 needs to save the address in link register on the stack, then call sub2
    - After returning from sub2, sub1 needs to retrieve link register address from the stack
    - Problems can occur if:
      * Stack is full; calling subroutine causes stack overflow
      * Stack is not properly maintained
  + **Parameter passing** – information exchange to/from a subroutine
    - Using registers – only if small # of parameters; subroutine is not recursive; and will not call other subroutines that use the same registers
      * Store values in designated “input” registers to be used by subroutine
      * Load values from “output” registers in which subroutine stores results
    - Using the stack – store values in stack before call, load values from stack within subroutine
    - Using memory locations – not usually used
* **Stack frame** – location at the top of the stack used as a private workspace for subroutines
  + Frame pointer register (FP) – enables access to private workspace for the current subroutine
    - Any register other than the SP can be used
  + The FP separates the parameters passed to the subroutine in the stack (from main) and values pushed to the stack during subroutine call
  + Before calling subroutine
    - Save parameters on stack
    - Save return address
    - SP now points to return address
  + Entering subroutine
    - Save existing FP value to stack
      * SUB SP, SP, #4
      * STR FP, SP
      * MOV FP, SP
      * FP now points to old FP value (above parameters, below local variables)
    - Store register values & local variables on stack
  + Returning from subroutine
    - Pop saved register values back into registers
    - Restore old FP value from stack (pointed to by FP)
    - Restore return address from stack into link register
    - Return
* Ex: MAIN
  + LDR R2, PARAM2 ;get param from memory
  + STR R2, [SP, #-4]! ;push param to stack
  + LDR R2, PARAM1
  + STR R2, [SP, #-4]!
  + BL SUB1 ;call subroutine
  + LDR R2, [SP] ;load subroutine result from stack
  + ADD SP, SP, #8 ;restore stack level
  + …
* SUB1
  + STR LR, [SP, #-4]! ;save return address on stack
  + STR FP, [SP, #-4]! ;save frame pointer on stack
  + MOV FP, SP ;set frame pointer
  + STR R2, [SP, #-4]! ;save register values
  + STR R3, [SP, #-4]!
  + LDR R2, [FP, #8] ;get param1 from stack
  + LDR R3, [FP, #12] ;get param2 from stack
  + …
  + STR R3, [FP, #8] ;save result on stack (overwrite param1 – no longer used)
  + LDR R3, [SP], #4 ;pop saved registers values from stack
  + LDR R2, [SP], #4
  + LDR FP, [SP], #4 ;pop saved FP from stack
  + LDR LR, [SP], #4 ;pop return address from stack
  + BX LR ;return