The Stack

and Subroutines

The Subroutine

- Allow re-use of code
- Write (and debug) code once, use it many times
- A subroutine is *called*
- Subroutine will *return* on completion
- Defer writing actual code until later
- Helps with readability and maintenance
- We can nest subroutines:
  subroutine calls other subroutines
- May be a procedure: does not return a value
- May be a function: does return a value

Operation of The Subroutine

- **Branch to Subroutine** (Branch and Link)
  `BL`⟨*cc*⟩ `label`

- **Link Register**: LR (R14)
  Holds address of instruction *after* the `BL` instruction

- **Return from Subroutine**
  `MOV`⟨*cc*⟩ `PC, LR`

Operation of The Subroutine

- **Branch to Subroutine** (Branch and Link)

  `BL`⟨*cc*⟩ `label` ⟨*cc*⟩: LR ← PC + 4

  ⟨*cc*⟩: PC ← PC + IR(offset)

- **Link Register**: LR (R14)

  Holds address of instruction *after* the `BL` instruction

- **Return from Subroutine**

  `MOV`⟨*cc*⟩ `PC, LR` ⟨*cc*⟩: PC ← LR

Example Subroutines

; Read line from keyboard into string at R10, uses R0 and R13

```
GetStr   MOV   R13, LR          ; Save Return Address
         BLAL  GetChar          ; Read keyboard into R0
         CMP   R0, #13          ; Is it ⟨return⟩ key ?
         BEQ   GetStr1          ; Yes ⇒ Exit Subroutine
         STRB  R0, [R10], #1    ; No ⇒ Save char in string
         BAL   GetStr           ; Get next character
GetStr1  EOR   R0, R0, R0       ; Clear R0
         STRB  R0, [R10]        ; Add terminating zero byte
         MOV   PC, R13          ; Return from GetStr
```

; Read char from keyboard into R0

```
GetChar  SWI   &4               ; Reach char into R0
         MOV   PC, LR           ; Return from GetChar
```

The Stack

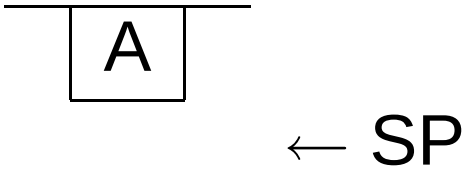
- Stack used to ‘remember’ values
- Provides temporary (local) storage
- Allows for subroutines (functions)
- R13 points to current Top Of Stack (TOS)
  Also known as the *Stack Pointer* (SP)
- Stack is a LIFO (last-in-first-out) device
  Most recent value *push*ed on is first *pop*ped off
  Stack grows ‘downwards’ in memory
- Each mode has it’s own stack pointer:
  `R13_fiq`, `R13_svc`, `R13_abt`, `R13_undef`, ...
  `R13` or SP refer to current mode

Operation of The Stack

1 Stack Empty

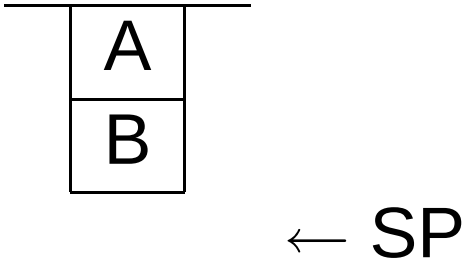
Operation of The Stack

1. Stack Empty
2. Push A



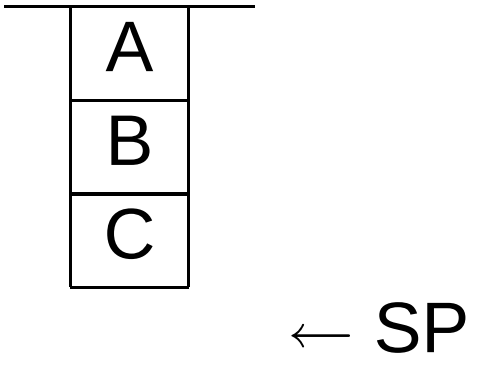
Operation of The Stack

1. Stack Empty
2. Push A
3. Push B



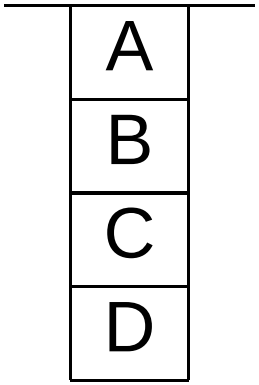
Operation of The Stack

1. Stack Empty
2. Push A
3. Push B
4. Push C



Operation of The Stack

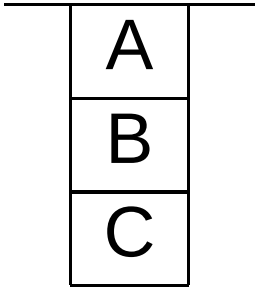
1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D



← SP

Operation of The Stack

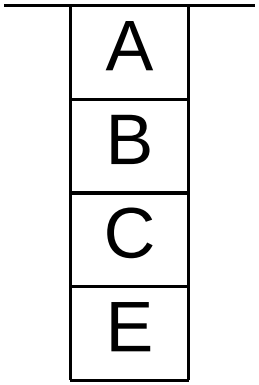
1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D
6. Pop



← SP

Operation of The Stack

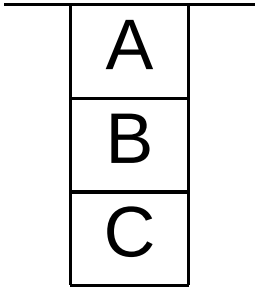
1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D
6. Pop
7. Push E



← SP

Operation of The Stack

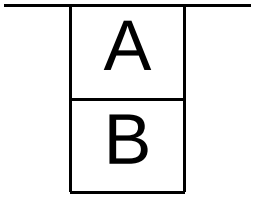
1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D
6. Pop
7. Push E
8. Pop



← SP

Operation of The Stack

1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D
6. Pop
7. Push E
8. Pop
9. Pop



← SP

Operation of The Stack

1. Stack Empty
2. Push A
3. Push B
4. Push C
5. Push D
6. Pop
7. Push E
8. Pop
9. Pop
10. Pop

← SP

Operation of The Stack

1 Stack Empty
2 Push A
3 Push B
4 Push C
5 Push D
6 Pop
7 Push E
8 Pop
9 Pop
10 Pop
11 Pop

← SP

Operation of The Stack

1	Stack Empty
2	Push A
3	Push B
4	Push C
5	Push D
6	Pop
7	Push E
8	Pop
9	Pop
10	Pop
11	Pop
	Stack Empty

Stack as Temporary Storage

- **Push**: Save register on the stack

  `STR`$\langle cc \rangle$ `R`s`, [SP], #-4`

- **Pop**: Recover register from stack

  `LDR`$\langle cc \rangle$ `R`d`, [SP, #4]!`

Stack as Temporary Storage

- **Push**: Save register on the stack

  `STR`$\langle cc \rangle$ `R`s`, [SP], #-4`

  $\langle cc \rangle$: MBR $\leftarrow$ Rs
  $\langle cc \rangle$: MAR $\leftarrow$ SP
  $\langle cc \rangle$: SP $\leftarrow$ SP - 4
  $\langle cc \rangle$: M(MAR) $\leftarrow$ MBR

- **Pop**: Recover register from stack

  `LDR`$\langle cc \rangle$ `R`d`, [SP, #4]!`

Stack as Temporary Storage

- **Push**: Save register on the stack

  `STR`$\langle cc \rangle$ `R`s`, [SP], #-4`

  $\langle cc \rangle$: MBR $\leftarrow$ Rs
  $\langle cc \rangle$: MAR $\leftarrow$ SP
  $\langle cc \rangle$: SP $\leftarrow$ SP - 4
  $\langle cc \rangle$: M(MAR) $\leftarrow$ MBR

- **Pop**: Recover register from stack

  `LDR`$\langle cc \rangle$ `R`d`, [SP, #4]!`

  $\langle cc \rangle$: SP $\leftarrow$ SP + 4
  $\langle cc \rangle$: MAR $\leftarrow$ SP
  $\langle cc \rangle$: MBR $\leftarrow$ M(MAR)
  $\langle cc \rangle$: Rd $\leftarrow$ MBR

Push/Pop a Set of Register

- **Push**: Save a set of registers on the stack
  `STM`⟨*cc*⟩⟨*mode*⟩ `SP!, {` ⟨*Register List*⟩ `}`
- **Pop**: Recover the set of registers
  `LDM`⟨*cc*⟩⟨*mode*⟩ `SP!, {` ⟨*Register List*⟩ `}`
- ⟨*mode*⟩ can be one of:

  IB: Increment Before
  IA: Increment After
  DB: Decrement Before
  DA: Decrement After

- ⟨*Register List*⟩
  A list of the registers to load/store
  E.g., `R0-R7, R10`

Nested Subroutines

- Use Stack to store Return Address (Link Register)
- Save all register used in the subroutine
  just in case the caller is using them
- *Must* pop off all values pushed onto stack !
- Pass parameters (*arguments*) into a subroutine

  $\Rightarrow$ Three *types* of variable passing
  by value / reference / name

  $\Rightarrow$ Three *methods* of passing variables
  In registers / on Stack / in parameter block
- Return a value from the subroutine