IV. Stacks & *Subroutines*

Jalal Kawash

Outline

1. Stacks
2. Subroutines

ARM IV
Stacks & Subroutines

Section 1

Stacks

Section 1 Objectives

At the end of this section you will

1. Use multiple-register load and store instructions
2. Be able to build stacks in ARM assembly
3. Understand different types of stacks that can be built
4. Understand the purpose of the system stack

Loading Multiple Registers

- LDM*cdam rn*{!}, *reg-list*{^}
- Loads multiple registers from memory
- *cd* is a condition
- *am* is the addressing mode
- *rn* is the base register
  - Holds the address of where data starts in memory
- *reg-list* is a list of registers to load
- ! Enforces an address update in *rn*
- ^ is discussed later (with exceptions)

Addressing Modes

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

Storing Multiple Registers

- STM*cd*am *rn*{!}, *reg-list*{^}
- Stores multiple registers in memory
  - Starting at address [*rn*]
  - In order of register numbers

  - "Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address."

Usage Examples

- `LDMIA r9, {r0-r3, r8}`
- `LDMIB r9, {r5, r0-r3, r8}`
    - Registers are loaded in the order: r0, r1, r2, r3, r5, & r8 (r0 lowest address)
- `STMDA r9, {r0-r3, r8, r7}`

LDMIA *reg{!}, reg-list*

1. Sort *reg-list* in ascending order by reg. name
2. *adrs* = *reg*
3. For each *r* in *reg-list*
    1. LDR *r*, [*adrs*] // STR for STMIA
    2. *adrs* += 4
4. If ! is present
    1. *reg* = *adrs*

LDMIA Example

- `LDMIA r9, {r0-r3}`
- Is equivalent to:
- `LDR r0, [r9] // increment after`
- `LDR r1, [r9, #4]`
- `LDR r2, [r9, #8]`
- `LDR r3, [r9, #12]`
- r9 does not change unless you use:
- `LDMIA r9!, {r0-r3}`

LDMIB *reg{!}, reg-list*

```
1. Sort reg-list in ascending order by reg. name
2. adrs = reg
3. For each r in reg-list
    1. adrs += 4
    2. LDR r, [adrs] // STR for STMIB
4. If ! is present
    1. reg = adrs
```

LDMIB Example

- `LDMIB r9, {r0-r3}`
- Is equivalent to:
- `LDR r0, [r9, #4] // increment before`
- `LDR r1, [r9, #8]`
- `LDR r2, [r9, #12]`
- `LDR r3, [r9, #16]`

LDMDA *reg{!}, reg-list*

```
1. Sort reg-list in ascending order by reg. name
2. adrs = reg
3. For each r in reg-list
   1. LDR r, [adrs] // STR for STMDA
   2. adrs −= 4
4. If ! is present
   1. reg = adrs
```

LDMDA Example

- `LDMDA r9, {r0-r3}`
- Is equivalent to:
- `LDR r0, [r9] // decrement after`
- `LDR r1, [r9, #-4]`
- `LDR r2, [r9, #-8]`
- `LDR r3, [r9, #-12]`

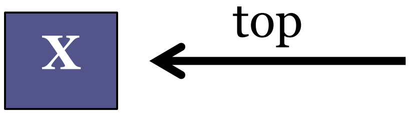
LDMDB *reg{!}, reg-list*

```
1. Sort reg-list in ascending order by reg. name
2. adrs = reg
3. For each r in reg-list
   1. adrs −= 4
   2. LDR r, [adrs] // STR for STMIA
4. If ! is present
   1. reg = adrs
```

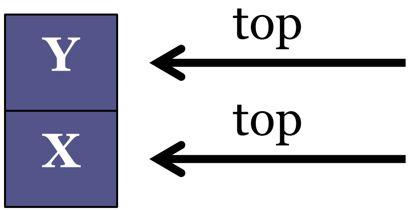
LDMDB Example

- `LDMDB r9, {r0-r3}`
- Is equivalent to:
- `LDR r0, [r9, #-4] //decrement before`
- `LDR r1, [r9, #-8]`
- `LDR r2, [r9, #-12]`
- `LDR r3, [r9, #-16]`

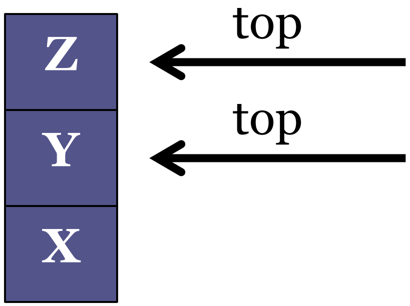
Stacks, push X



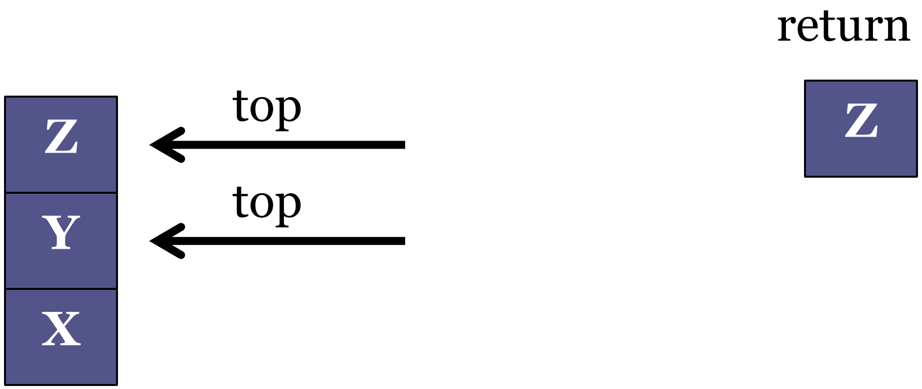
Stacks, push Y



Stacks, push Z

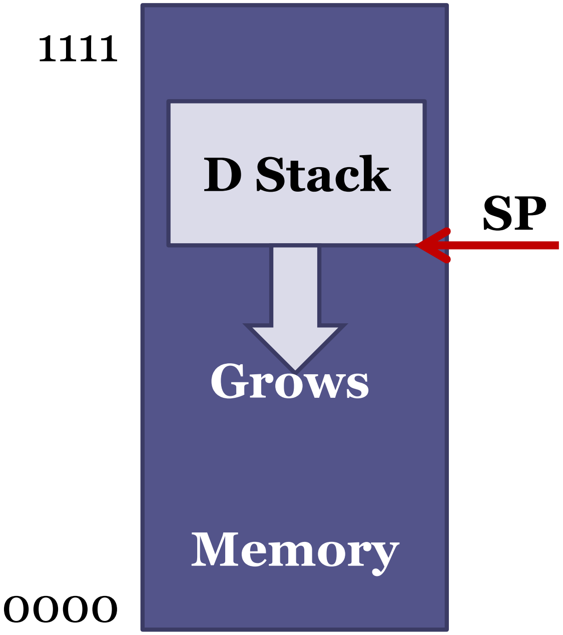


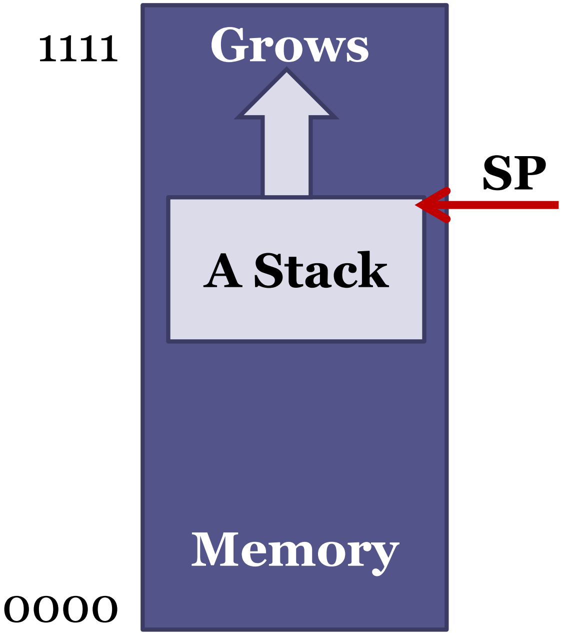
Stacks, Pop



Stack Types

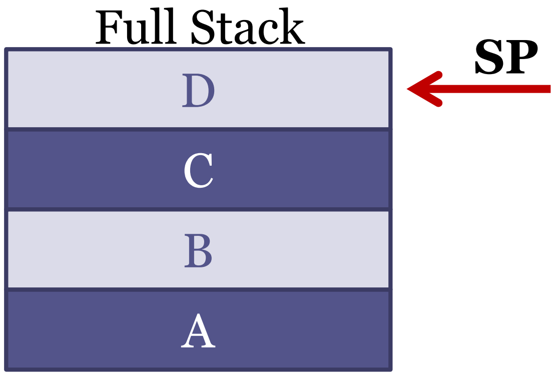
- Descending versus ascending stack
    - Descending: grows from higher to a lower address
    - Ascending: grows from a lower to a higher address

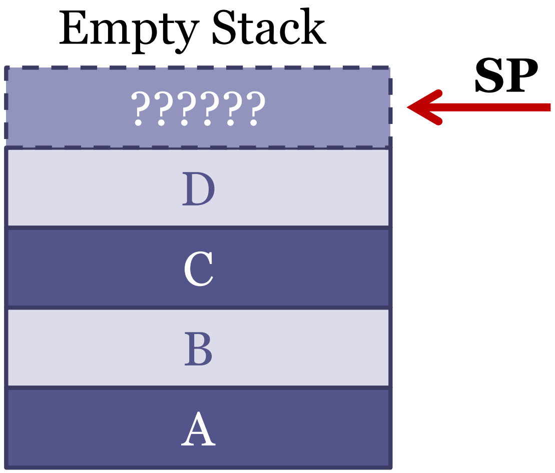




Stack Types

- Full versus empty stack
  - Full: the stack pointer points at the top of the stack (last pushed item)
  - Empty: the stack pointer points to the next free space on the top of the stack





Implementing Stack Types

- To push
  - STMFD : Full Descending stack
  - STMED : Empty Descending stack
  - STMFA : Full Ascending stack
  - STMEA : Empty Ascending stack
- To Pop
  - LDM{FD,ED,FA,EA}
- Can be also directly implemented without the stack operation suffixes (LDMIA, ...)

Push and Pop

- PUSH {list of registers} is synonym for
- STMDB sp!, {list of registers}

- POP {list of registers} is synonym for
- LDMIA sp!, {list of registers}

- "Registers are stored on the stack in numerical order, with the lowest numbered register at the lowest address."

The System Stack

- The *system* or *runtime stack* is a region of memory managed directly by the CPU
  - Is a LIFO structure
- Keeps information such as:
  - Local variables to subroutines
  - Parameters passed to subroutines

Activation Records

- When a subroutine is called an activation record is created at the top of the system stack
- The activation records lives as long as the subroutines lives
  - A local variable does not exist if the activation record does not exist

Section 2

Subroutines

Section 1 Objectives

At the end of this section you will

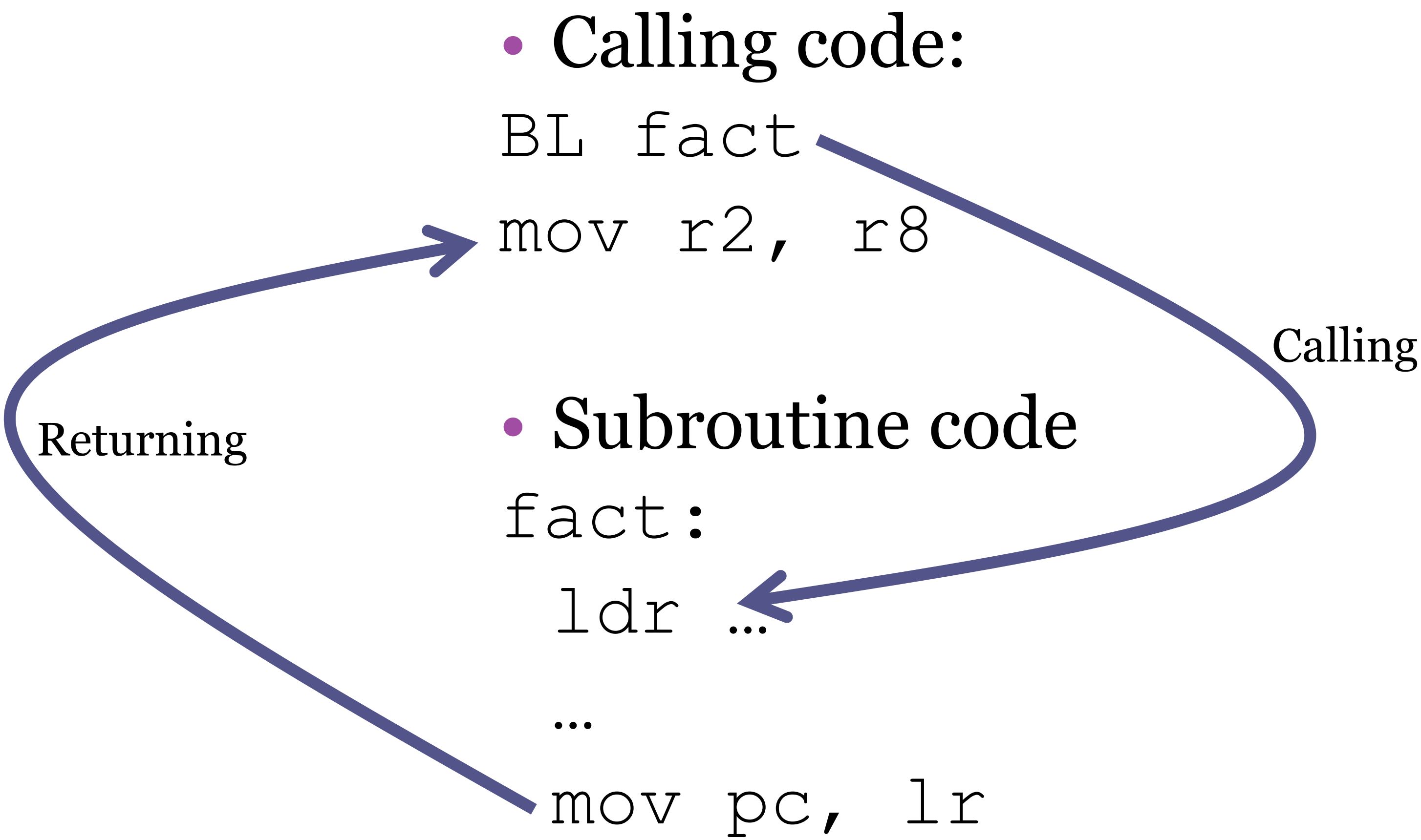
1. Write subroutines
2. Know how to pass parameters to subroutines using registers or the system stack

Subroutines

- A subroutine is called using `BL`
- BL: Branch and Link
    - Branches to a new address
    - Saves the return address (`pc` is stored in `lr`)

- A subroutine starts with a label
- And ends with `mov pc, lr`

Subroutine Example



- Calling code:

```
BL fact
mov r2, r8
```

- Subroutine code

```
fact:
  ldr …
  …
mov pc, lr
```

Parameter Passing

- Using registers (by value)
  - Register holds data
    - Quick
    - Limited to register size and number of registers
- Using registers (by reference)
  - Register holds the address of data
    - Not limited to register size
- Using the stack
  - Standard compiler method

Stack Parameters

- Push parameters onto the stack
- Call subroutine
- Subroutine access parameters on the stack
- Subroutine returns a value on the stack

Stack Parameters Example

```
  STMFD (sp)!,{r0,r1} // Push the arguments
  BL stackEx          // Call the routine
  LDMFD (sp)!,{r0}    // Get the return value

  ...

stackEx:
  LDMFD (sp)!,{r4,r5} // Load the arguments
  ...                 // Process them
  STMFD (sp)!,{r2}    // Push return value
  MOV   pc, lr        // Return
```

Local Variables

- Local variables are space in the activation record
- A subroutines allocates space for them on the stack
- Accessing them is done through the stack pointer (sp) or through another register (base pointer) initialized to sp
- They are deleted before or right after the subroutine returns

Example

- Subroutine in C:

```
void mySub()
{
     int x = 10;
     int y = 20;
     …
}
```

Example

- Translates to

```
mySub:
  MOV r12, sp // r12 is the base pointer
  SUB sp, #8  // creates space for 2 local vars
    // assumes an FD stack
  MOV r0, #10 // value for x
  MOV r1, #20 // value for y
  STMFD r12, {r0,r1} // does not change r12
  …

  ADD sp, #8  // delete local vars; can be also
    // done in calling code
  MOV pc, lr  // return from mySub
```

Accessing variables

- Variables can now be accessed using the base pointer
- [r12] is var1
- [r12+4] is var2 in an ascending stack
- [r12-4] is var2 in a descending stack

AAPCS

- ARM Application Procedure Call Standard
- Defines guidelines on how to define subroutines