- A stack is an example of a **data structure**
  - A method of organising data
  - Defined structure and operations
- Stacks typically used for temporary storage of data
- Analogous to a stack of paper or a stack of cards
- Some rules:
  - **Push**: Place cards on the **top of the stack**
  - **Pop**: Remove cards from the **top of the stack**
  - **LIFO**: **L**ast **In** is the **F**irst **Out**
  - Compare with **FIFO**: **F**irst **In** **F**irst **Out**
Stack example

- **Push** operations:
  - push 8
  - push 14
  - push 12

- **Pop** operations:
  - pop 12
  - pop 14
  - push 6
  - pop 6
  - pop 8

- **Stack**
  - empty stack
  - push 8
  - push 14
  - push 12
  - pop 14
  - push 6
  - pop 8
Stack implementation

- Stacks are fundamental to the operation of most modern computers
  - CPUs may provide special instructions, addressing modes and registers for the purpose of manipulating stacks

- To implement a stack data structure we need ...  
  - An area of **memory** to store the data items
  - A **Stack Pointer (SP) register** to point to the top of the stack
  - A stack **growth convention**
  - Some well defined operations: **initialize, push, pop**
**Stack growth convention**

- **Ascending or Descending?**
  - Does the stack grow from low to high (**ascending stack**) or from high to low (**descending stack**) memory addresses?

- **Full / Empty?**
  - Does the stack pointer point to the last item pushed onto the stack (**full stack**), or the next free space on the stack (**empty stack**)?
Stack implementation

- Stack initialization
  - Set the stack pointer (SP) to some sensible value at one end of the memory region to be used to store the stack
  - This is the bottom of the stack (and, since the stack has just been initialized, also the top of the stack)

```
start
  LDR R12, =STK_TOP ; Initialise stack pointer
  ...
  ...
  ...
  ...

stop
  B stop

STK_SZ EQU 0x400 ; 1K stack

AREA Stack, DATA, READWRITE
SPACE STK_SZ

STK_MEM STK_TOP
```

- Stack implementation
  - LDR R12, =STK_TOP ; Initialise stack pointer
  - ...
  - ...
  - ...
  - ...
  - B stop
  - STK_SZ EQU 0x400 ; 1K stack
  - AREA Stack, DATA, READWRITE
  - SPACE STK_SZ
  - STK_MEM STK_TOP
Assume full descending stack growth convention

To **push** a word onto the stack

- decrement the stack pointer by 4 bytes (= 1 word = 32 bits)
- store the word in memory at the location pointed to by the stack pointer

E.g. push 0x45 using R12 as stack pointer
Stack implementation

- e.g. push 0x45, push 0x7b, push 0x19

```assembly
; push 0x45
LDR    R0, =0x45
SUB    R12, R12, #4
STR    R0, [R12]

; push 0x7b
LDR    R0, =0x7b
SUB    R12, R12, #4
STR    R0, [R12]

; push 0x19
LDR    R0, =0x19
SUB    R12, R12, #4
STR    R0, [R12]
```
Stack implementation

- Assume full descending stack growth convention
- To **pop** a word off the top of the stack
  - load the word from memory at the location pointed to by the stack pointer (into a register)
  - increment the stack pointer by 4 bytes
Stack implementation

- e.g. pop three word size values off the stack

```assembly
; pop
LDR  R0, [R12]
ADD  R12, R12, #4

; pop
LDR  R0, [R12]
ADD  R12, R12, #4

; pop
LDR  R0, [R12]
ADD  R12, R12, #4
```

- If we had previously pushed 0x45, 0x7b and 0x19, in that order, we will pop 0x19, 0x7b and 0x45
Pushing / Popping non-word size data

- Could push values of any size on to a stack

- To push $n$-bytes (assuming a full descending stack)
  - SUBtract $n$ from SP
  - STR $n$ bytes at SP

- To pop $n$-bytes (assuming a full descending stack)
  - LDR $n$ bytes at SP
  - ADD $n$ to SP

- Pushing non-word size data is problematic due to memory alignment constraints
Pushing / Popping non-word size data

- e.g. Push 1 word, followed by 3 half-words, followed by 2 words ...

```plaintext
; push word from R0
SUB    R12, R12, #4
STR    R0, [R12]

; push 3 half words from R1, R2 and R3
SUB    R12, R12, #2
STRH   R1, [R12]
SUB    R12, R12, #2
STRH   R2, [R12]
SUB    R12, R12, #2
STRH   R3, [R12]

; push 2 words from R4 and R5
SUB    R12, R12, #4
STR    R4, [R12]
SUB    R12, R12, #4
STR    R5, [R12]
```

- ... must pop data in reverse order ...

Won’t work as expected – non-aligned word accesses
Pushing / Popping non-word size data

- e.g. ... continued ... popping same data into original registers ...

```assembly
; pop 2 words into R5 and R4
LDR   R5, [R12]
ADD   R12, R12, #4
LDR   R4, [R12]
ADD   R12, R12, #4

; pop 3 halfwords into R3, R2 and R1
LDRH  R3, [R12]
ADD   R12, R12, #2
LDRH  R2, [R12]
ADD   R12, R12, #2
LDRH  R1, [R12]
ADD   R12, R12, #2

; pop word into R0
LDR   R0, [R12]
ADD   R12, R12, #4
```

- Use µVision to see effect of non-aligned addresses

Won’t work as expected – non-aligned word accesses
Addressing Modes for Stack Operations

- e.g. Push word from R1 to stack pointed to by R12

```assembly
; push word from R0
SUB    R12, R12, #4
STR    R0, [R12]
```

- Replace explicit SUB with immediate pre-indexed addressing mode

```assembly
; push word from R0
STR    R0, [R12, #-4]!
```

- Similarly, to pop word, replace explicit ADD with immediate post-indexed addressing mode

```assembly
; pop word into R0
LDR    R0, [R12], #4
```
System Stacks

- In general, stacks ...
  - can be located anywhere in memory
  - can use any register as the stack pointer
  - can grow as long as there is space in memory

- Usually, a computer system will provide one or more system-wide stacks to implement certain behaviour (in particular, subroutine calls)
  - ARM processors use register R13 as the stack pointer (SP)
  - Stack pointer is initialised by startup code executed when the computer is powered-on
  - (libcs1021.lib contains our startup code)
  - Limited in size (stack overflow error)
System Stacks

- Rarely any need to use any other stack
- Use the system stack pointed to by R13/sp for your own purposes

```assembly
; push word from R0
STR R0, [sp, #-4]
```

- Never initialise sp / R13

```assembly
; load address 0xA1000000 into R13
LDR R13, =0xA1000000
```

- Typical use of a system stack is temporary storage of register contents
- Programmer’s responsibility to pop off everything that was pushed on to the system stack
  - Not doing this is likely to result in an error
LDM and STM instructions

- Frequently we need to load/store the contents of a number of registers from/to memory

```assembly
; store contents of R1, R2 and R3 to memory at address 0xA1001000
LDR    R0, =0xA1001000
STR    R1, [R0]
STR    R2, [R0, #4]
STR    R3, [R0, #8]

; load R1, R2 and R3 with contents of memory at address 0xA1001000
LDR    R0, =0xA1001000
LDR    R1, [R0]
LDR    R2, [R0, #4]
LDR    R3, [R0, #8]
```
ARM instruction set provides **Load Multiple (LDM)** and **Store Multiple (STM)** instructions for this purpose.

The following examples achieve the same end result as the previous example ...

```
; store contents of R1, R2 and R3 to memory at address 0xA1001000
LDR    R0, =0xA1001000
STMIA  R0, {R1-R3}

; load R1, R2 and R3 with contents of memory at address 0xA1001000
LDR    R0, =0xA1001000
LDMIA  R0, {R1-R3}
```
Consider the following STM instruction ...

**STMIA R0, {R1-R3}**

### Increment After (IA) mode of operation:
- first register is stored at <base address>
- second register is stored at <base address> + 4
- third register is stored at <base address> + 8

### Contents of base register R0 remain unchanged
LDM and STM behaviour

STMIA  R0, {R1-R3}

32 bits = 1 word

before

after
### LDM and STM behaviour

- Four modes of operation for LDM and STM instructions

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>LDM</th>
<th>STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment After</td>
<td>LDMIA</td>
<td>STMIA</td>
</tr>
<tr>
<td>Increment Before</td>
<td>LDMIB</td>
<td>STMIB</td>
</tr>
<tr>
<td>Decrement After</td>
<td>LDMDA</td>
<td>STMDA</td>
</tr>
<tr>
<td>Decrement Before</td>
<td>LDMDB</td>
<td>STMDB</td>
</tr>
</tbody>
</table>

- Register list (e.g. \{R1\text{-}R3, R10, R7\text{-}R9\})
- Order in which registers are specified is not important
- For both LDM and STM, the lowest register is always loaded from the lowest address, regardless of mode of operation (IA, IB, DA, DB)
LDM and STM behaviour

STMDB  R0, {R2,R5-R7}

before

after
LDM and STM with stacks

- Frequently use LDM and STM instructions to load/store data from/to a stack
- Must choose mode of operation (IA, IB, DA, DB) appropriate to stack growth convention
- e.g. Full Descending stack
  - Decrement Before pushing data (STMDB)
  - Increment After pushing data (LDMIA)
- To push/pop data using LDM and STM
  - Use stack pointer register (e.g. R13 or sp) as base register
  - Use ! syntax to modify LDM/STM behaviour so the stack pointer is updated, e.g.

STMDB sp!, {R1-R3}
LDM and STM behaviour

STMDB sp!, {R1-R3}

32 bits = 1 word

before

after

sp/R13

sp/R13
LDM and STM with stacks

- e.g. Push R1, R2, R3 and R5 on to a full descending stack with R13 (or sp) as the stack pointer

```
STMDB sp!, {R1-R3,R5}
```

- Note use of ! in sp!

- e.g. Pop R1, R2, R3 and R5 off a full descending stack with R13 (or sp) as the stack pointer

```
LDMIA sp!, {R1-R3,R5}
```

- Note use of ! in sp!

- Works because the lowest register is always loaded/stored from/to lowest address
LDM and STM with stacks

- Because LDMxx and STMxx are frequently used to implement stacks, the ARM instruction set provides stack-oriented synonyms

<table>
<thead>
<tr>
<th>Stack growth convention</th>
<th>Push</th>
<th>Stack-oriented synonym</th>
<th>Pop</th>
<th>Stack-oriented synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full descending</td>
<td>STMDB</td>
<td>STMFD</td>
<td>LDMIA</td>
<td>LDMFD</td>
</tr>
<tr>
<td>Full ascending</td>
<td>STMIB</td>
<td>STMFA</td>
<td>LDMDA</td>
<td>LDMFA</td>
</tr>
<tr>
<td>Empty descending</td>
<td>STMDA</td>
<td>STMED</td>
<td>LDMIB</td>
<td>LDMED</td>
</tr>
<tr>
<td>Empty ascending</td>
<td>STMIA</td>
<td>STMEA</td>
<td>LDMDB</td>
<td>LDMEA</td>
</tr>
</tbody>
</table>
LDM and STM with stacks

- Stack-oriented synonyms for LDMxx and STMxx allow us to use the same suffix for both LDM and STM instructions

- e.g. Push R1, R2, R3 and R5 on to a full descending stack with R13 (or sp) as the stack pointer

  \[
  \text{STMFD} \quad \text{sp!}, \text{\{R1-R3,R5\}}
  \]

- e.g. Pop R1, R2, R3 and R5 off a full descending stack with R13 (or sp) as the stack pointer

  \[
  \text{LDMFD} \quad \text{sp!}, \text{\{R1-R3,R5\}}
  \]
Stacks summary

- A stack is a data structure with well defined operations
  - initialize, push, pop
- Stacks are accessed in LIFO order (Last In First Out)
- Implemented by
  - setting aside a region of memory to store the stack contents
  - initializing a stack pointer to store top-of-stack address
- Growth convention
  - Full/Empty, Ascending/Descending
- User defined stack or system stack
- When using the system stack, always pop off everything that you push on – not doing this will probably cause an error that may be hard to correct