Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

# 03 – Load/Store Multiple Instructions and Stacks

**CS1022 – Introduction to Computing II**

Dr Jonathan Dukes / jdukes@scss.tcd.ie
School of Computer Science and Statistics

**L**oa**D** **M**ultiple and **ST**ore **M**ultiple instructions
(LDM and STM)

# LoaD Multiple (LDM) / STore Multiple (STM)



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

```
; store contents of R1, R2 and R3 to memory at address 0x40001000

LDR    R12, =0x40001000     ; initialise R12 with base address

STR    R1, [R12]
STR    R2, [R12, #4]
STR    R3, [R12, #8]
```

```
; load R1, R2 and R3 with contents of memory at address 0x40001000

LDR    R12, =0x40001000     ; initialise R12 with base address

LDR    R1, [R12]
LDR    R2, [R12, #4]
LDR    R3, [R12, #8]
```

# LoaD Multiple (LDM) / STore Multiple (STM)



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 0x40001000

LDR   R12, =0x40001000
STMIA R12, {R1-R3}


; load R1, R2 and R3 with contents of memory at address 0x40001000

LDR   R12, =0x40001000
LDMIA R12, {R1-R3}
```

# LDM / STM Syntax

Consider the following STM instruction …

```
STMIA   R12, {R1-R3}
```

Note position of base address register operand!!

**mode of operation**
**e.g. IA – Increment After**

**base address register**
**e.g. R12**

**register list**
**e.g. R1–R3**

## Increment After (IA) mode of operation:

first register is stored at <base address>

second register is stored at <base address> + 4

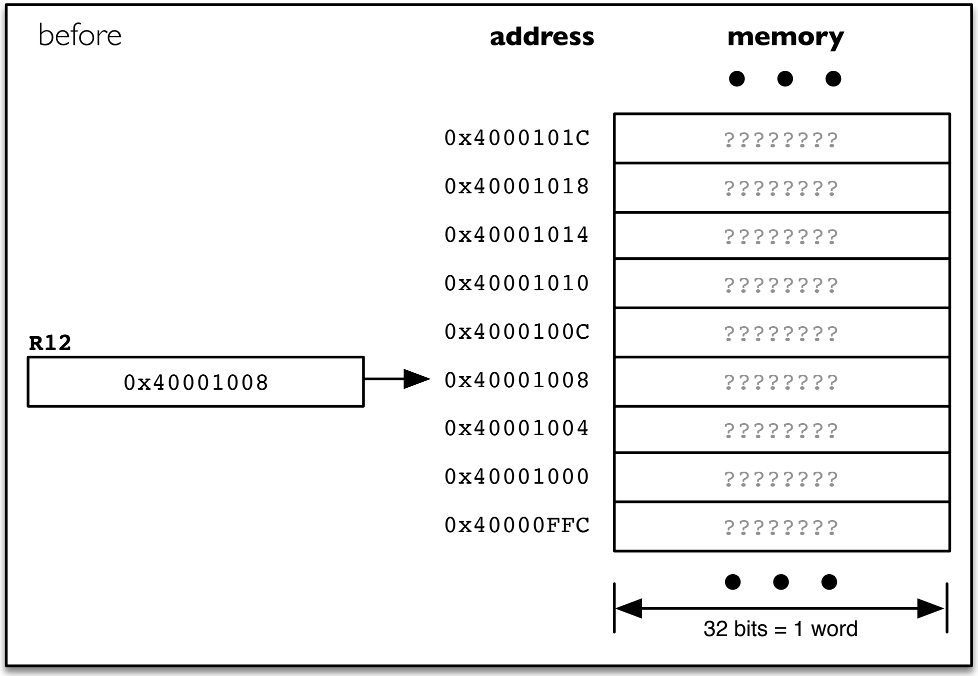
third register is stored at <base address> + 8

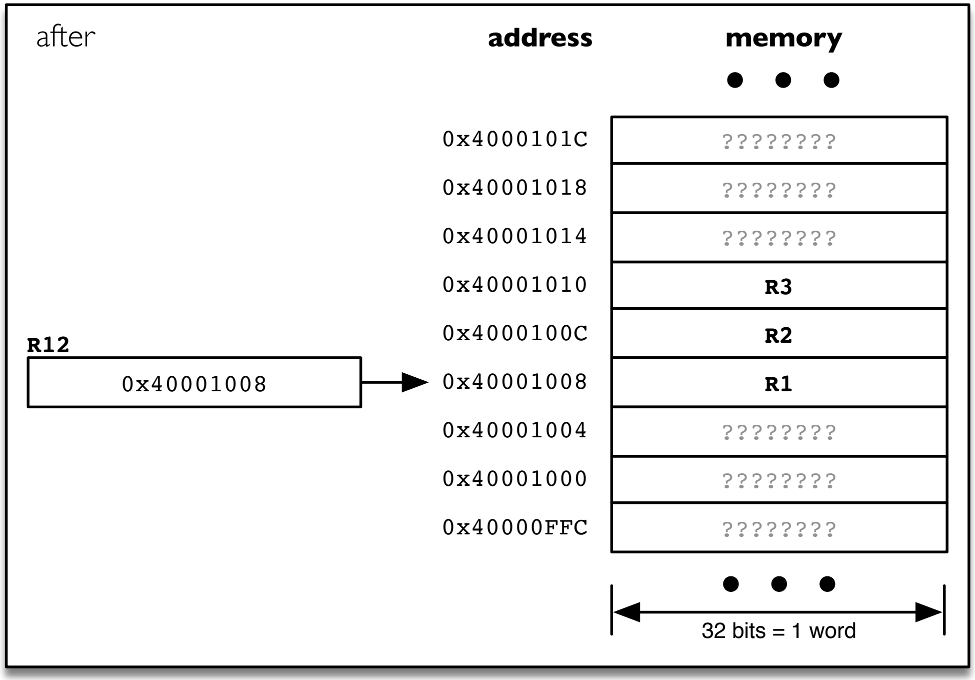
**Contents of base register R12 remain unchanged**

# LDM / STM Behaviour



```
STMIA   R12, {R1-R3}
```



# LDM / STM Modes of Operation



Four modes of operation for LDM and STM instructions

| Behaviour | LDM | STM |
|---|---|---|
| Increment After | LDMIA | STMIA |
| Increment Before | LDMIB | STMIB |
| Decrement After | LDMDA | STMDA |
| Decrement Before | LDMDB | STMDB |

Register list

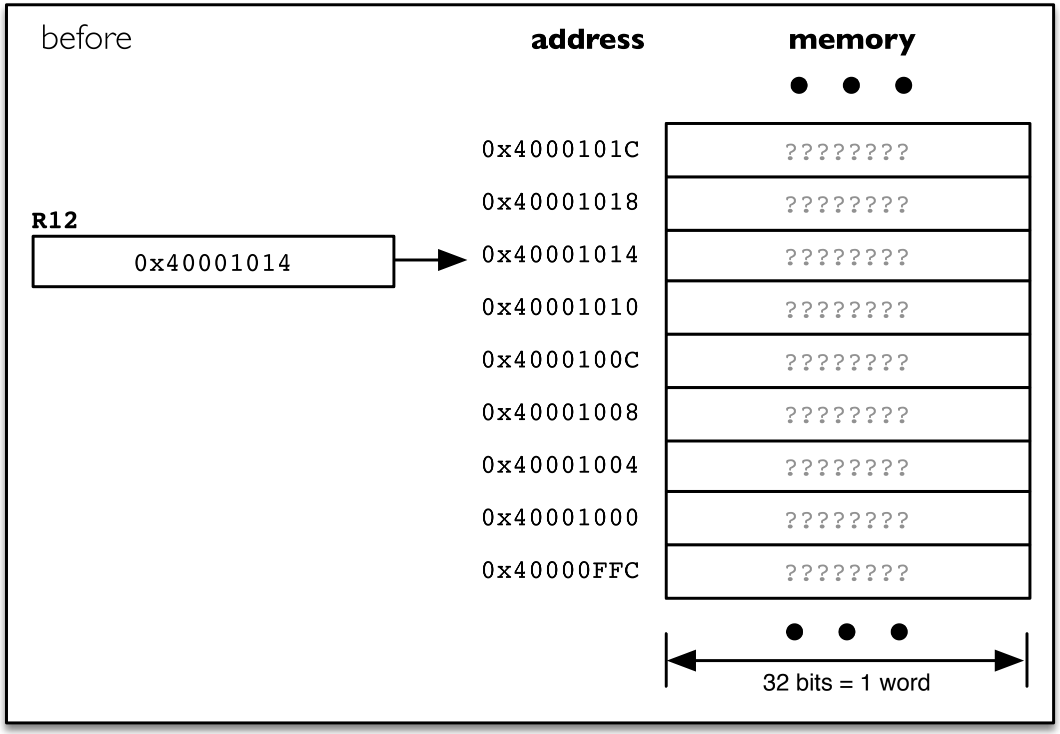
e.g. {R1-R3, R10, R7-R9}

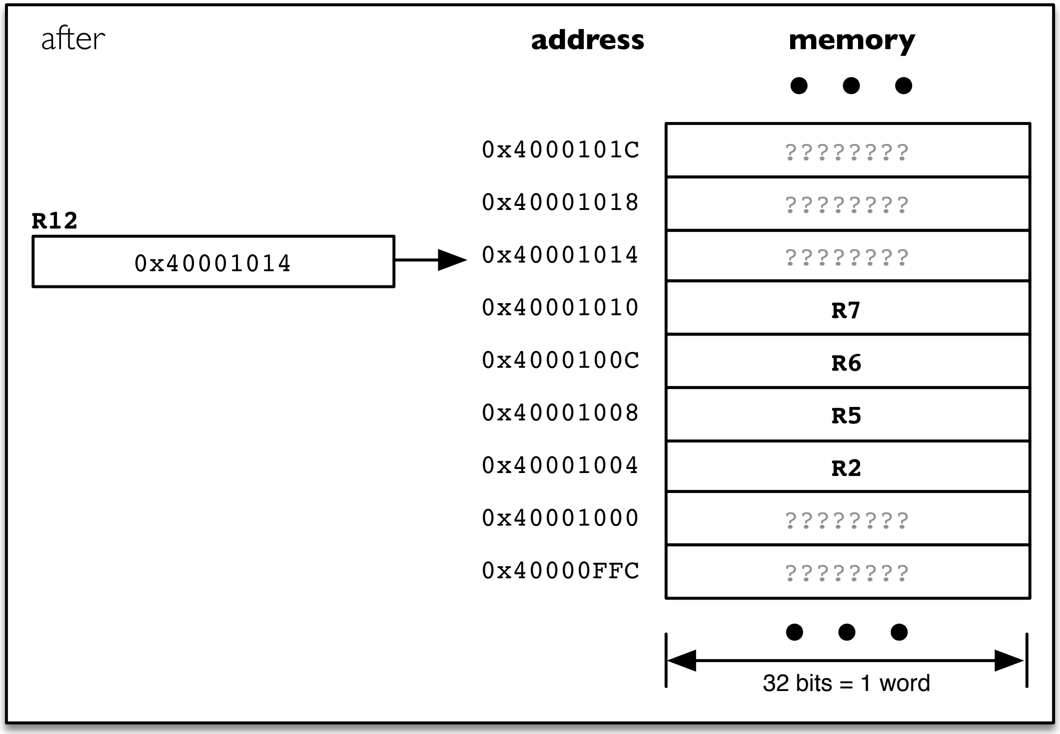
Order in which registers are specified is not important

**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 / STM Behaviour

```
STMDB   R12, {R5-R7,R2}
```



# Stacks (in depth)

# Stacks



A **stack** is an example of an **abstract data type**

- A convention for organising data
- Well-defined/understood operations and behaviour

Happens to be a very useful structure for implementing aspects of the behaviour of software, particularly the implementation of “methods” / “functions” / “procedures” / “subroutines”

Convenient data structure for other purposes

Analogous to a stack of paper / stack of cards / stack of bricks / stack of examination scripts :-(

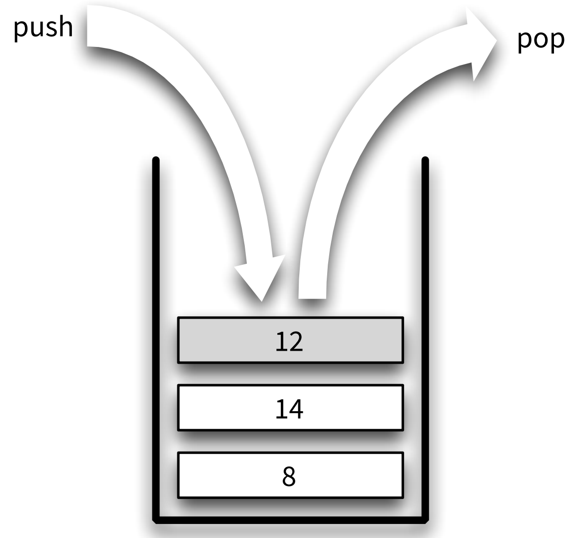
# Stack Behaviour



Operations

"Push": Place item on the top of the stack

"Pop": Remove item from the top of the stack




In practice, we can observe (load) or modify (store) the value of items anywhere in the stack

But this goes beyond the normal (formal) definition of a stack

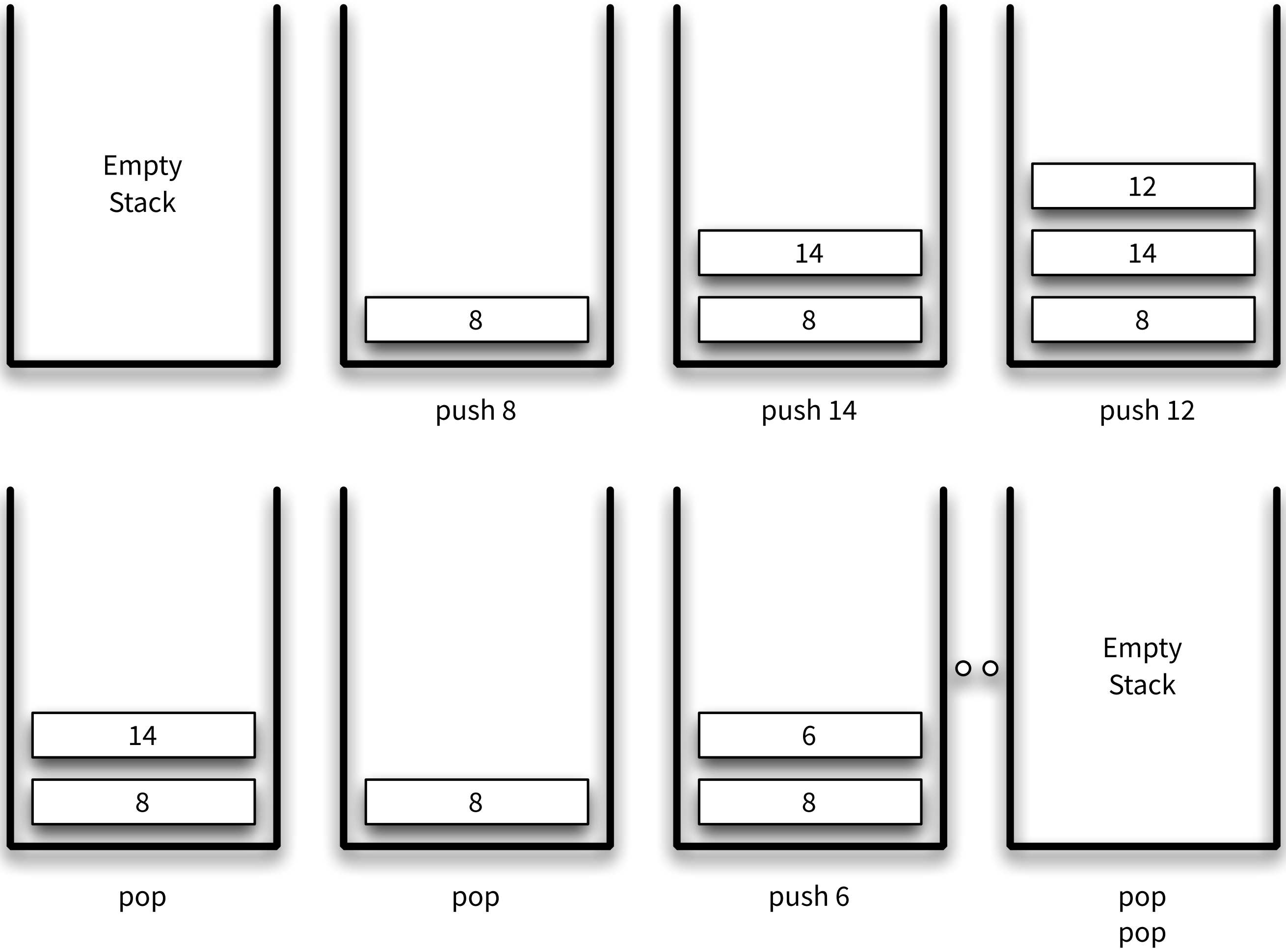
A **LIFO** data structure: **L**ast **I**n **F**irst **O**ut

Compare with **FIFO**: **F**irst **I**n **F**irst **O**ut
(guess what we call this …)

See ***Algorithms and Data Structures*** next year!

# Stack Example





# Stack Implementation



To implement a stack we need …

1. An area of memory to store the data items

    size of the area of memory determines the maximum size of the stack

2. A **Stack Pointer (SP)** register to point to the top of the stack

    we will see that we don't need to know where the bottom of the stack is!!

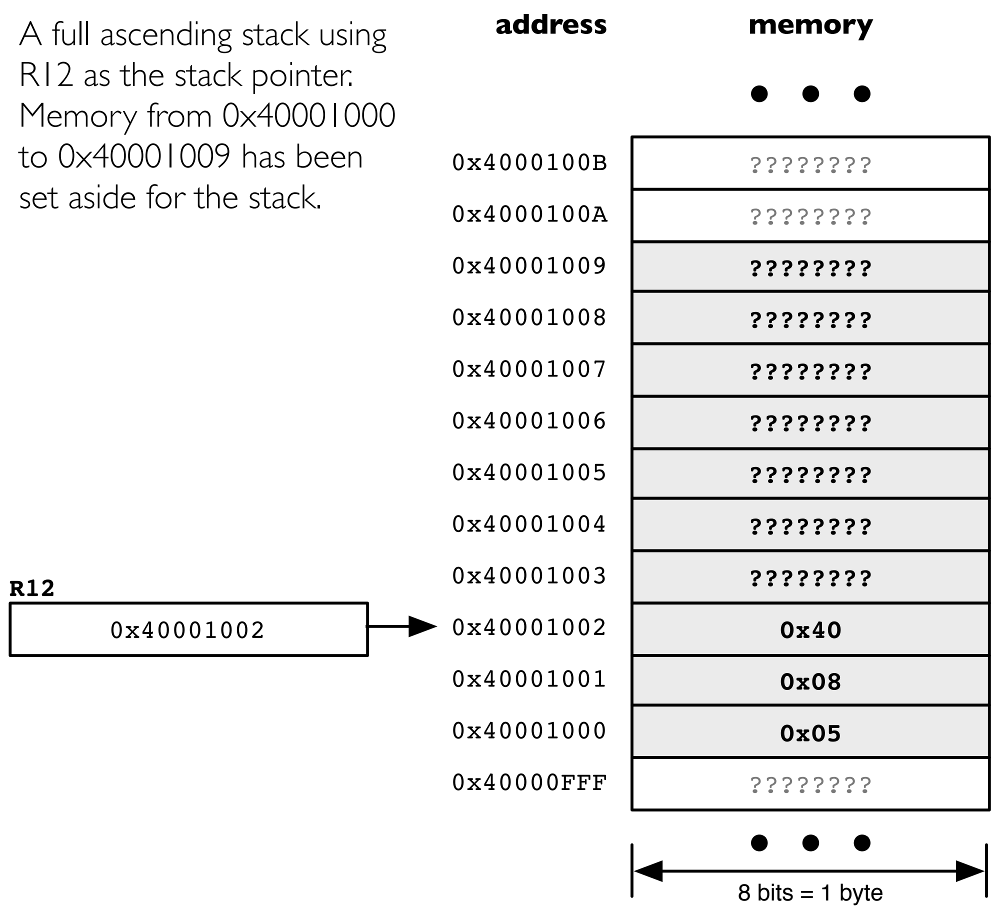
3. A stack **growth convention** (rules for pushing and popping)

| **Stack Growth Convention Options** | |
|---|---|
| Ascending or Descending | Full or Empty |
| Does the stack grow from low to high (ascending stack) or from high to low (descending stack) memory addresses? | 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 Memory

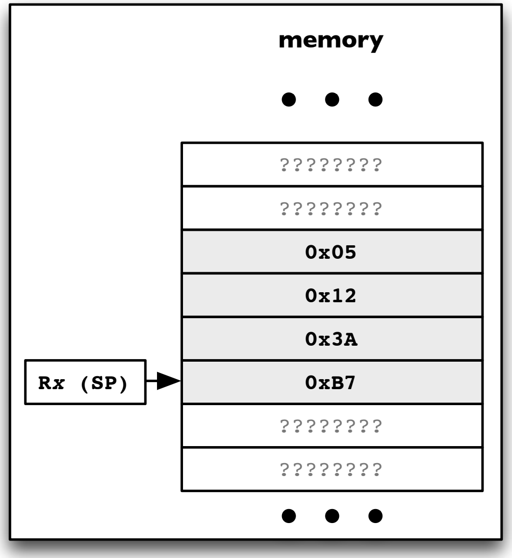


A full ascending stack using R12 as the stack pointer. Memory from 0x40001000 to 0x40001009 has been set aside for the stack.

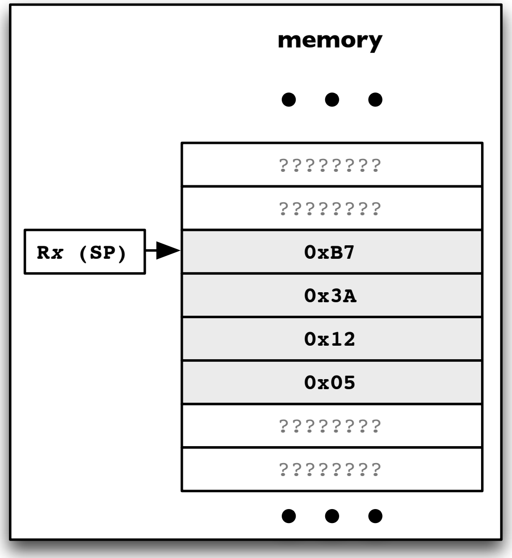


# Stack Growth Convention

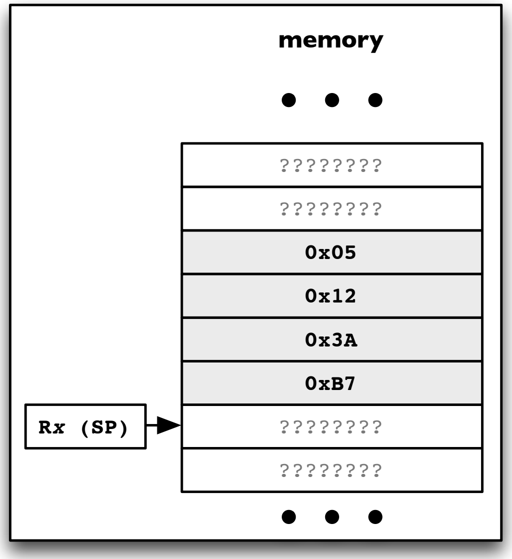




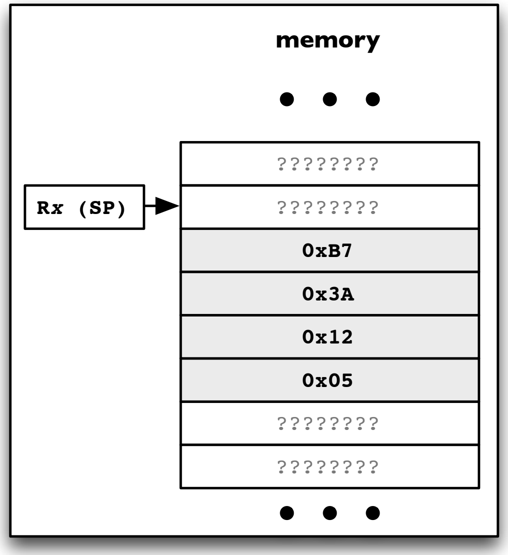

full descending stack



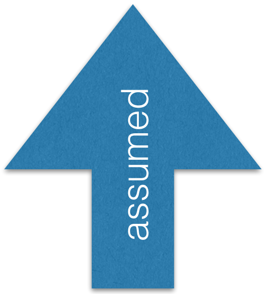

full ascending stack




empty descending stack




empty ascending stack



# Stack Implementation



## Initialisation

Set **Stack Pointer (SP)** to address at the start or end of the memory region to be used to store the stack (must consider the growth convention)

This is the bottom of the stack

(and, since the stack has just been initialised, also the top of the stack!)

```
STK_SZ      EQU     0x400                     ; 1K stack (1024 bytes)

            AREA    Stack, DATA, READWRITE
STK_MEM     SPACE   STK_SZ
STK_TOP

            AREA    RESET, CODE, READONLY

            LDR     R12, =STK_TOP             ; Initialise stack pointer
                                              ; (assume full descending)
            ...     ...                       ; ...
            ...     ...                       ; rest of program
            ...     ...                       ; ...
stop        B       stop
```

# Stack Implementation – push



Assume **full descending** stack growth

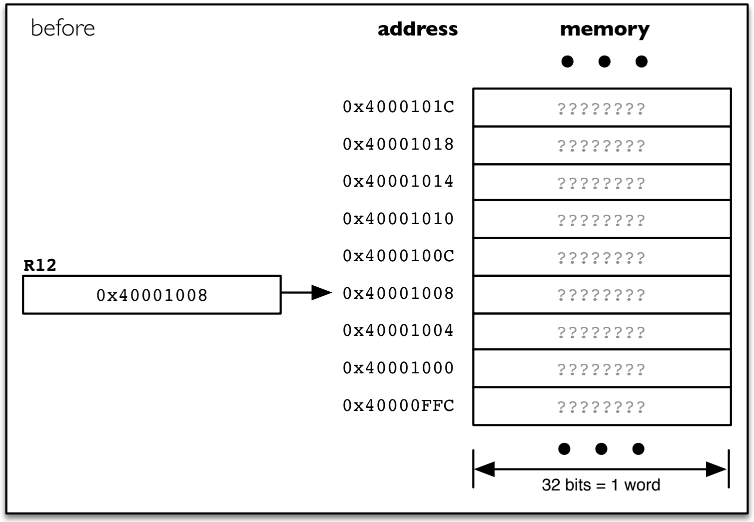
To push a word onto the stack

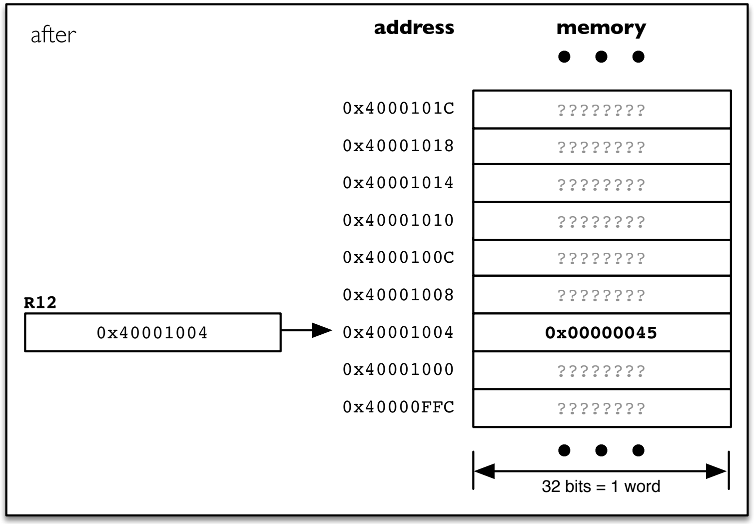
1. decrement the stack pointer by 4 bytes
   (4 bytes = 1 word = 32 bits)

2. store the word in memory at the location
   pointed to by the stack pointer

e.g. push 0x45 using R12 as stack pointer

```
LDR    R0, =0x45       ; example value to push
SUB    R12, R12, #4   ; adjust SP
STR    R0, [R12]
```



# Stack Implementation – push



e.g. Push three words (0x00000045, 0x0000007B, 0x00000019)

```
; push 0x00000045
LDR   R0, =0x00000045
SUB   R12, R12, #4
STR   R0, [R12]

; push 0x0000007b
LDR   R0, =0x0000007b
SUB   R12, R12, #4
STR   R0, [R12]

; push 0x00000019
LDR   R0, =0x00000019
SUB   R12, R12, #4
STR   R0, [R12]
```

# Stack Implementation – pop



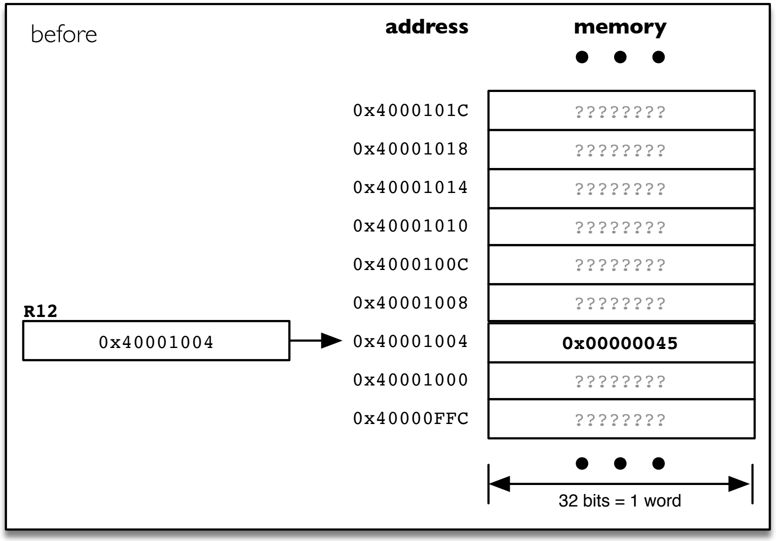
Again, assume full descending stack growth convention

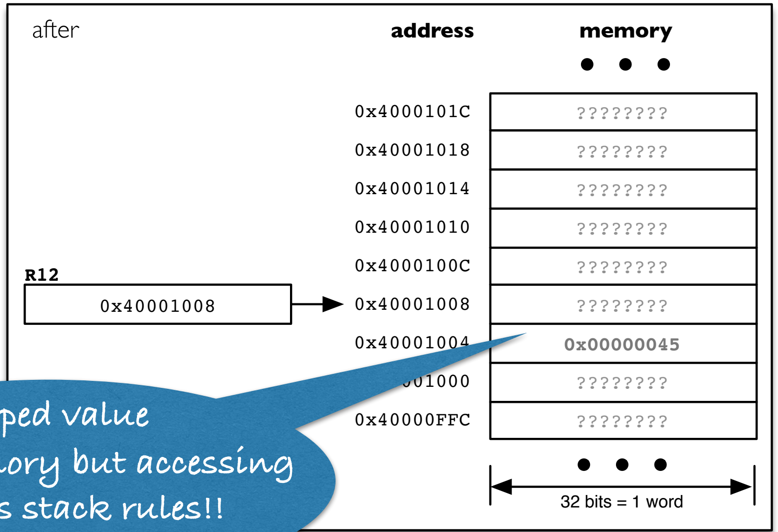
To pop a word off the stack

1. load the word from memory at the location pointed to by the stack pointer (into a register)
2. increment the stack pointer by 4 bytes

e.g. pop word off top of stack into R0

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




popped value
still in memory but accessing
it breaks stack rules!!

# Stack Implementation – pop



e.g. Pop three word-size values off the top of the stack

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

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

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

Contents of R0 after each pop operation depend on contents of stack

- e.g. if we had previously pushed 0x45, 0x7b and 0x19, we will pop 0x19, 0x7b and 0x45

Note the order!!

# Addressing Modes for Stack Operations



e.g. Push word from R0 to stack pointed to by R12

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

Replace explicit SUB with immediate pre-indexed addressing mode

Equivalent

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

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

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

# The System Stack



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 **system stack pointer** (SP)
- System stack pointer is initialised by startup code (executed at powered-on)
- Limited in size (possibility of “stack overflow”)

# The System Stack



Rarely any need to use any other stack

Use the system stack pointed to by R13/SP for your own purposes

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

Note use of SP in place of R13

Never re-initialise R13/SP during program execution

```
; load address 0x40000000 into R13
LDR   R13, =0x40000000
```

Please, please never do this!! or anything vaguely similar!! after your program initialisation (unless you are certain you know what you are doing!)

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 very likely to result in an error that may be very hard to find!!
- High level language compilers take care of this for you

# LDM, STM and Stacks
# (and some old friends: PUSH and POP)

# LDM / STM with Stacks



LDM and STM instructions can be used to push/pop multiple stack items with a single instruction

Choose DB/IA/DA/IB operation appropriate to stack growth convention

increment/decrement, before/after

e.g. Full Descending stack

Decrement Before pushing data (STMDB)

Increment After popping 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**

```
STMDB SP!, {R1-R3}  ; push R1, R2, R3
LDMIA SP!, {R1-R3}  ; pop R1, R2, R3
```

Note the ! syntax

You might be thinking "Wait I can do that with the much simpler PUSH and POP instrctions!!"

PUSH is a shortcut for STMDB SP!
POP is a shortcut for LDMIA SP!

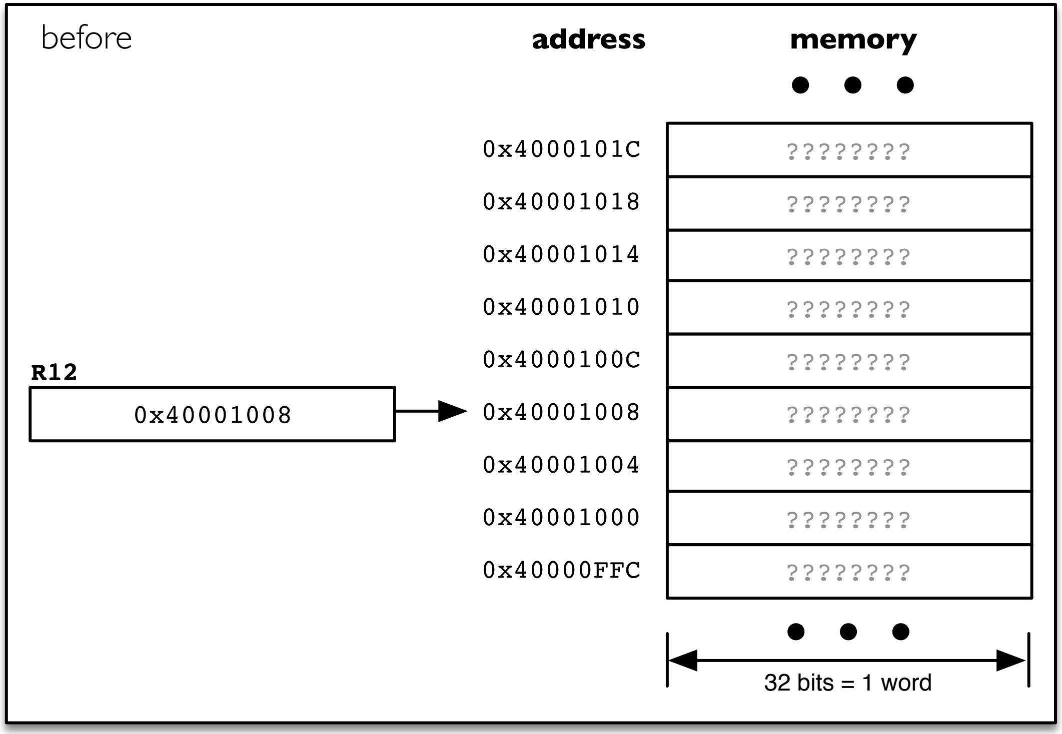
Check out the disassembly window in uVision!

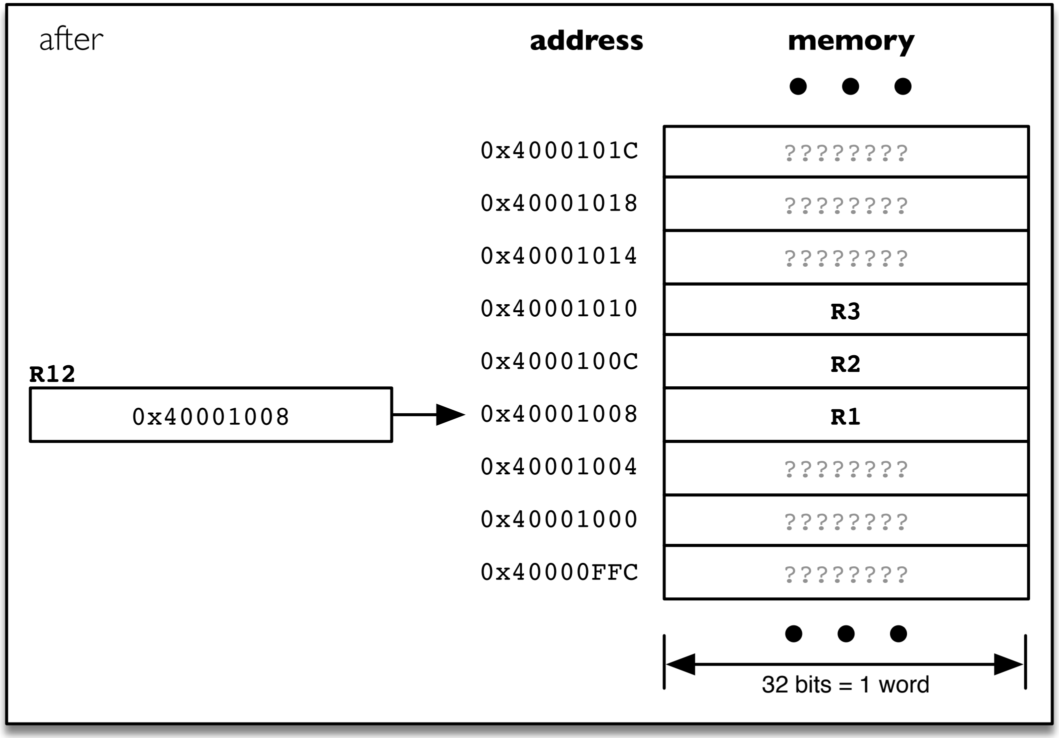
# STM Stack Example



Push contents of registers R1, R2 and R3

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



# LDM / STM Stack Examples



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

```
LDMIA sp!, {R5,R2,R3,R1}
```

Note use of **!** in **SP!**

**Works because the lowest register is always loaded from or stored to the lowest address**

# LDM / STM Synonyms for Stacks



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

Easier for us to remember!

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

```
STMFD sp!, {R1-R3,R5}        ; or PUSH, if you prefer
```

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

```
LDMFD sp!, {R1-R3,R5}        ; or POP, if you prefer
```

# LDM / STM Synonyms for Stacks



| Stack growth convention | push | | pop | |
|---|---|---|---|---|
| | STM mode | stack-oriented synonym | LDM mode | stack-oriented synonym |
| full descending | STMDB | **STMFD or PUSH** | LDMIA | **LDMFD or POP** |
| full ascending | STMIB | STMFA | LDMDA | LDMFA |
| empty descending | STMDA | STMED | LDMIB | LDMED |
| empty ascending | STMIA | STMEA | LDMDB | LDMEA |

# Pushing / Popping *any*-Size Data



Could push values of any size on to a stack

To push a byte from R0 to system stack

```
STRB  R0, [SP, #-1]!
```

To pop a byte from system stack to R0

```
LDRB  R0, [SP], #1
```

**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 …

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

; push 3 half words from R1, R2 and R3
STRH  R1, [SP, #-2]!
STRH  R2, [SP, #-2]!
STRH  R3, [SP, #-2]!

; push 2 words from R4 and R5
STR   R4, [SP, #-4]!
STR   R5, [SP, #-4]!
```

Won't work as expected

Non-aligned word access

# 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