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

2.1 – Stacks

CSU11022 – Introduction to Computing II

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

Stacks

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

A convention for organising da

Well-defined/understood oper 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 too (e.g. parsing, backtracking)

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 t[illegible]ck

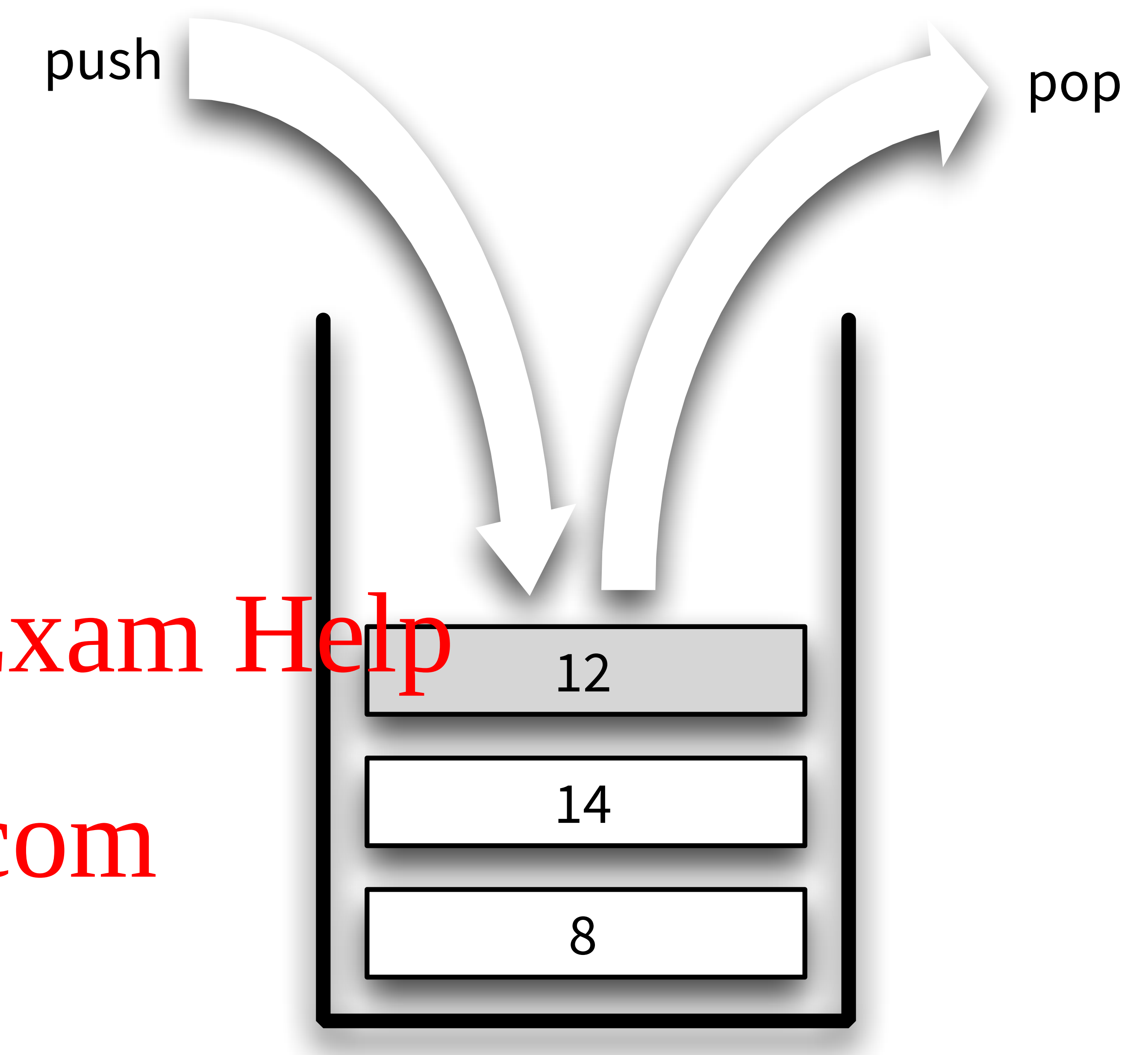
In practice, we can observe (load / LDR) or modify (store / STR) the value of items anywhere in the stack

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

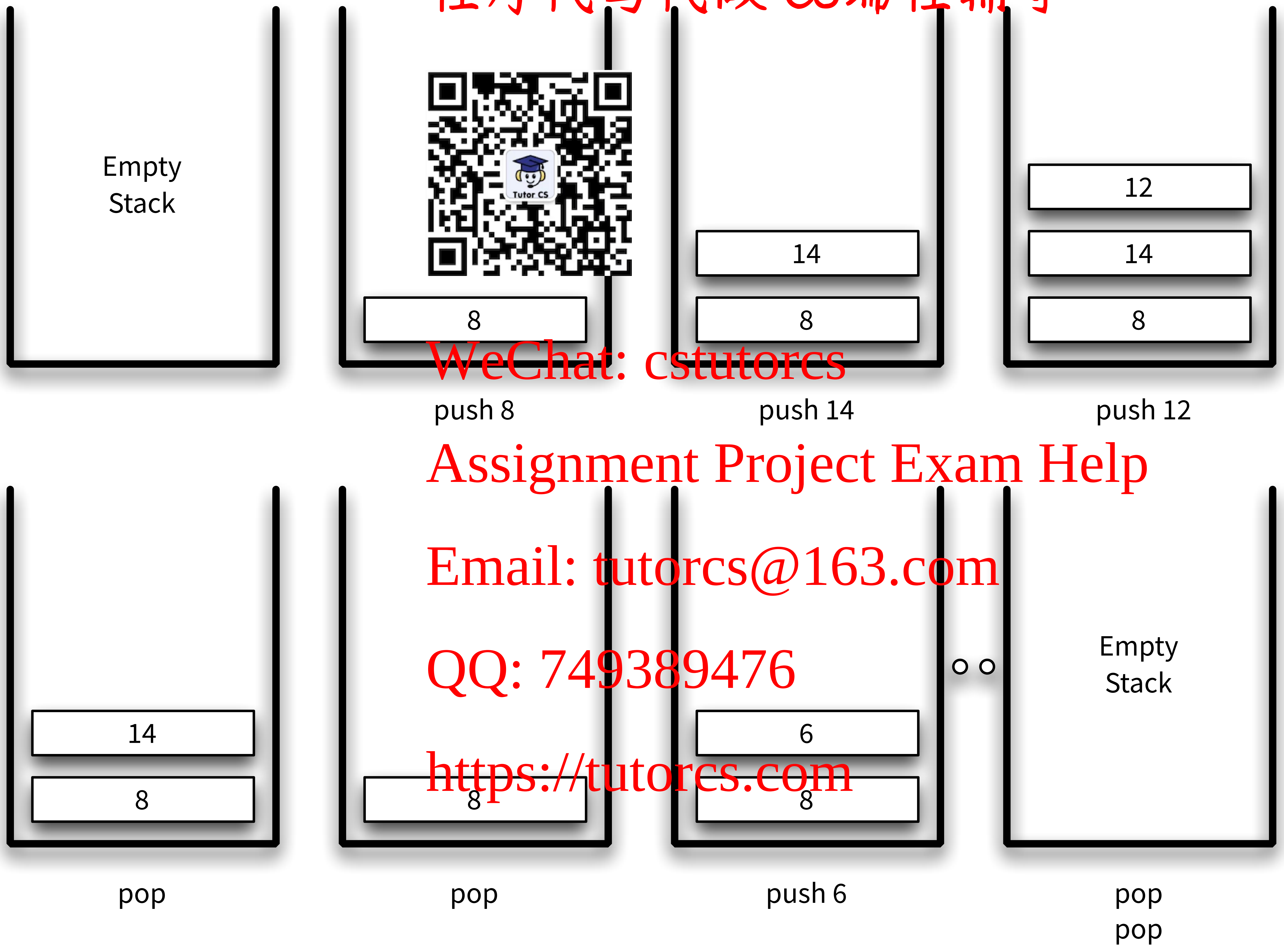
A **LIFO** data structure: **L**ast **I**n **F**irst **O**ut

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 sto[illegible]a items

   size of the area of memory deter[illegible]imum 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)?

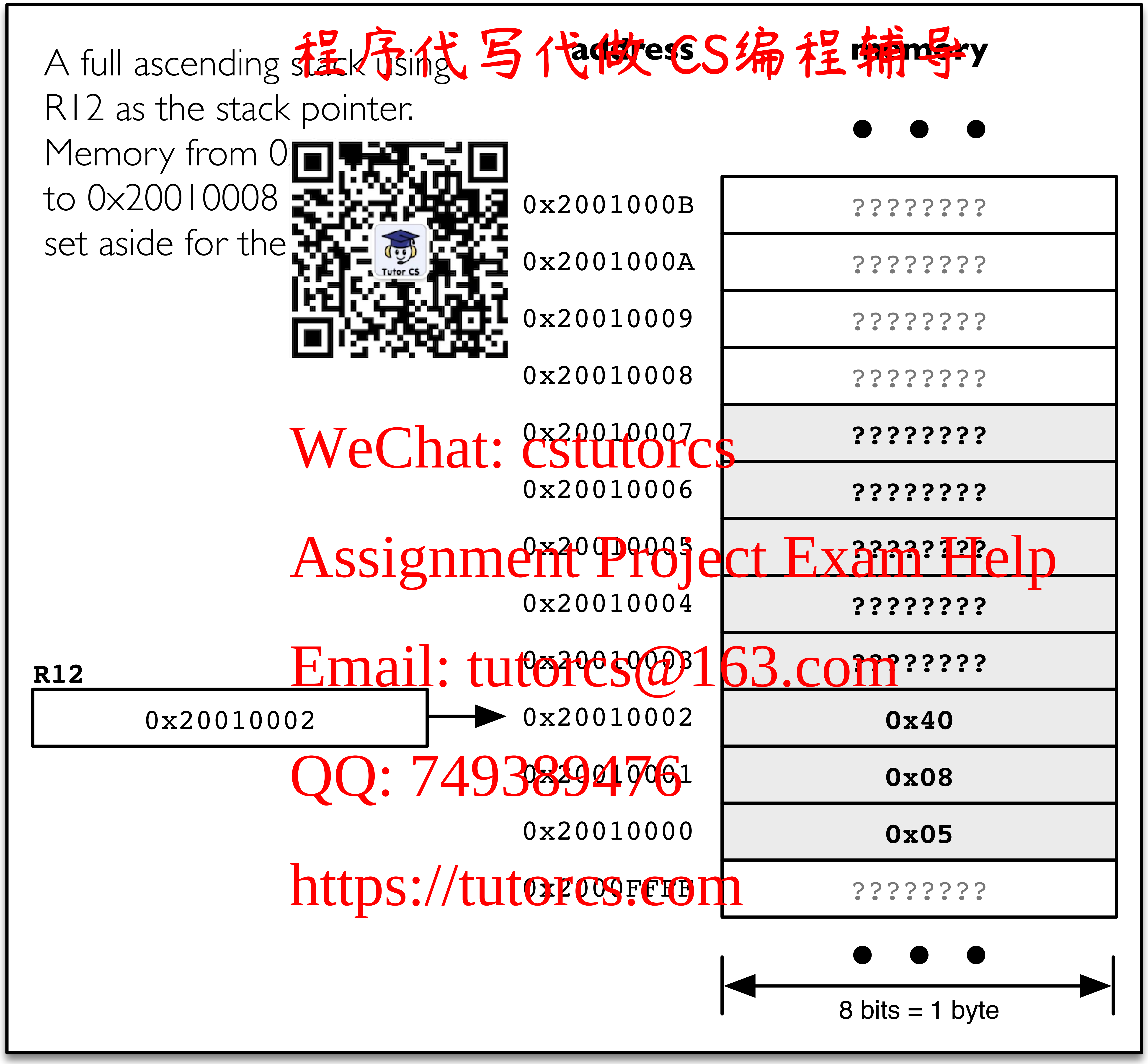
Stack Memory

A full ascending stack using R12 as the stack pointer.

Memory from 0

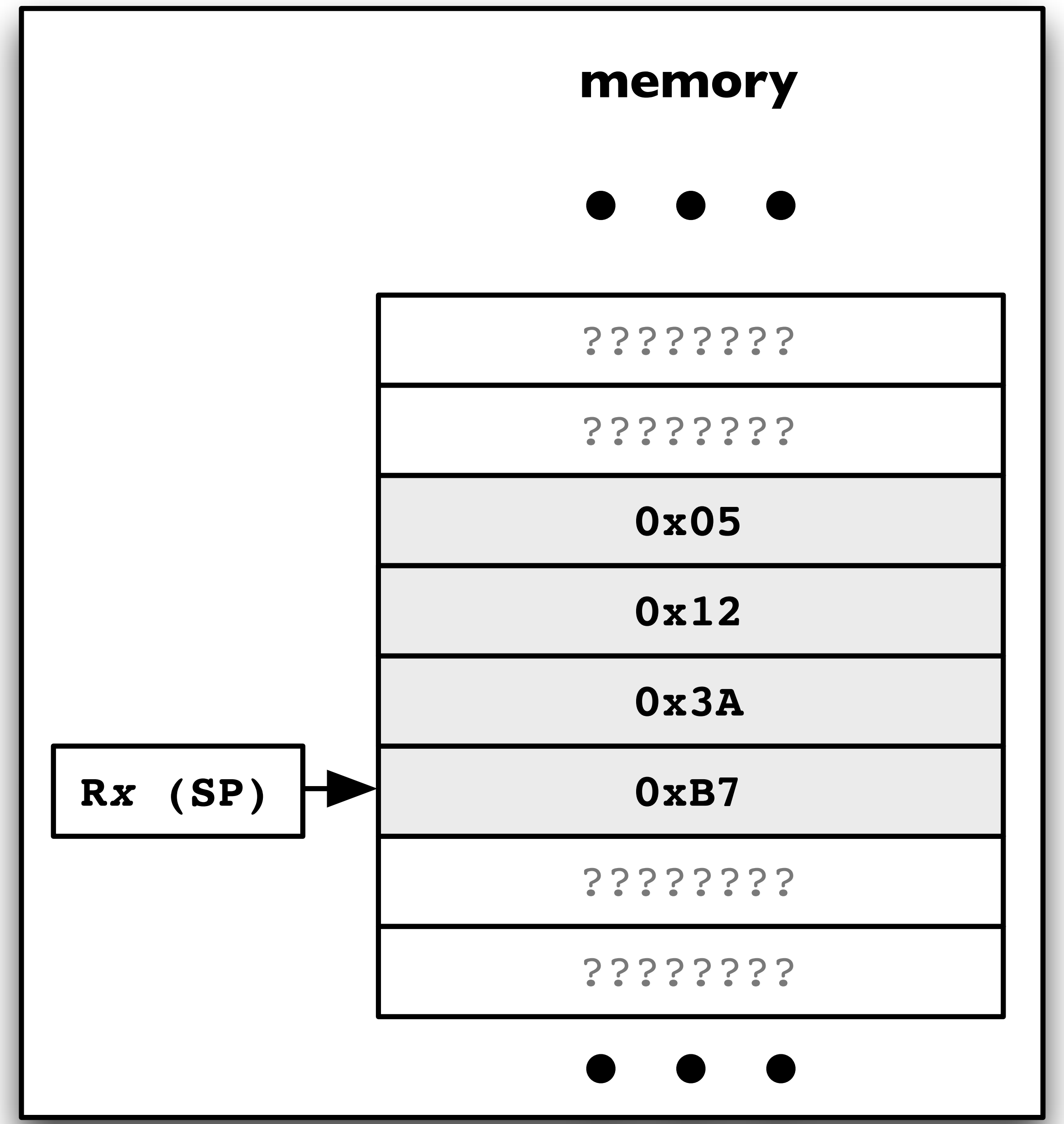
to 0x20010008

set aside for the

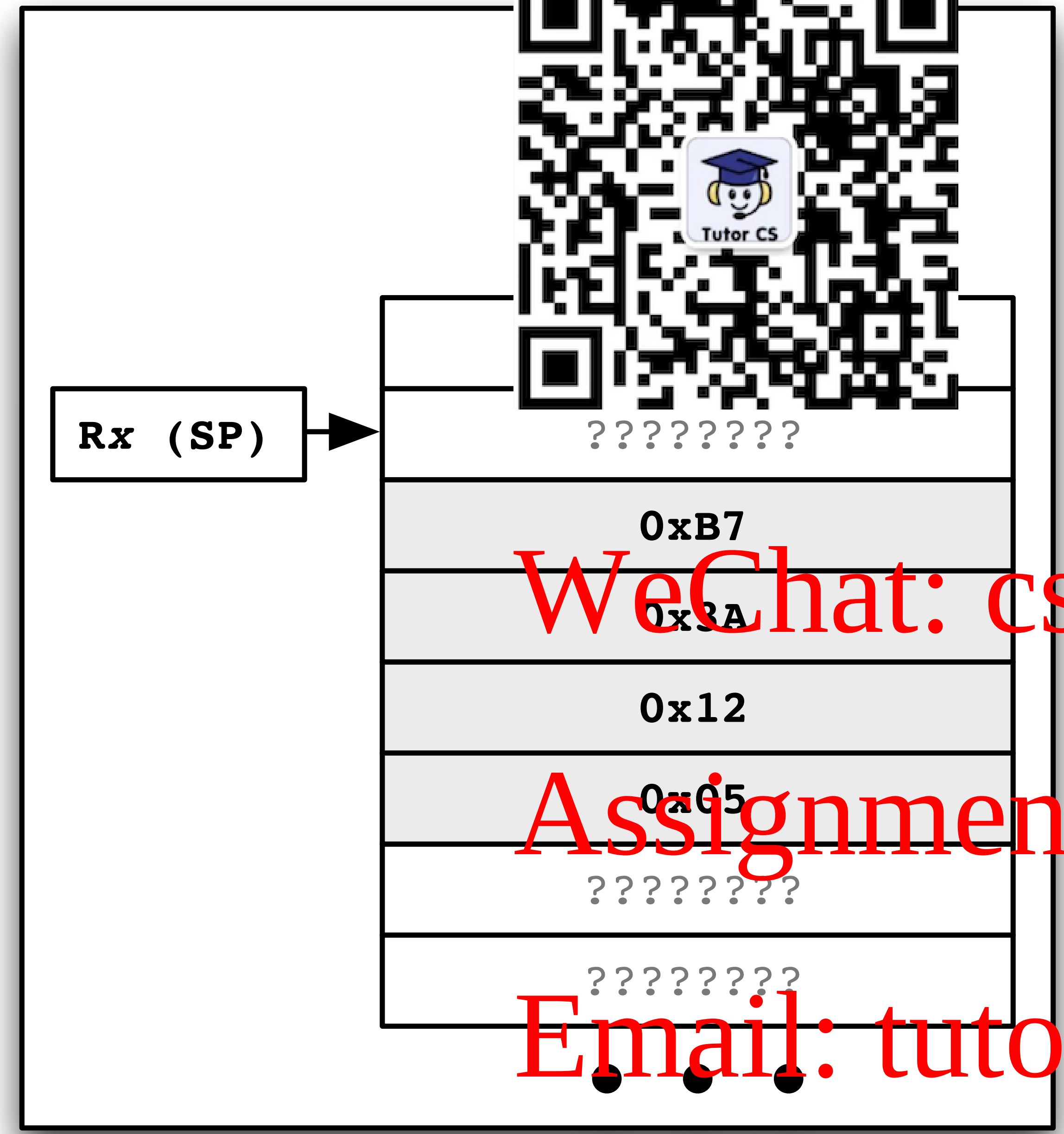


address	memory
0x2001000B	????????
0x2001000A	????????
0x20010009	????????
0x20010008	????????
0x20010007	????????
0x20010006	????????
0x20010005	????????
0x20010004	????????
0x20010003	????????
0x20010002	0x40
0x20010001	0x08
0x20010000	0x05
0x2000FFFF	????????

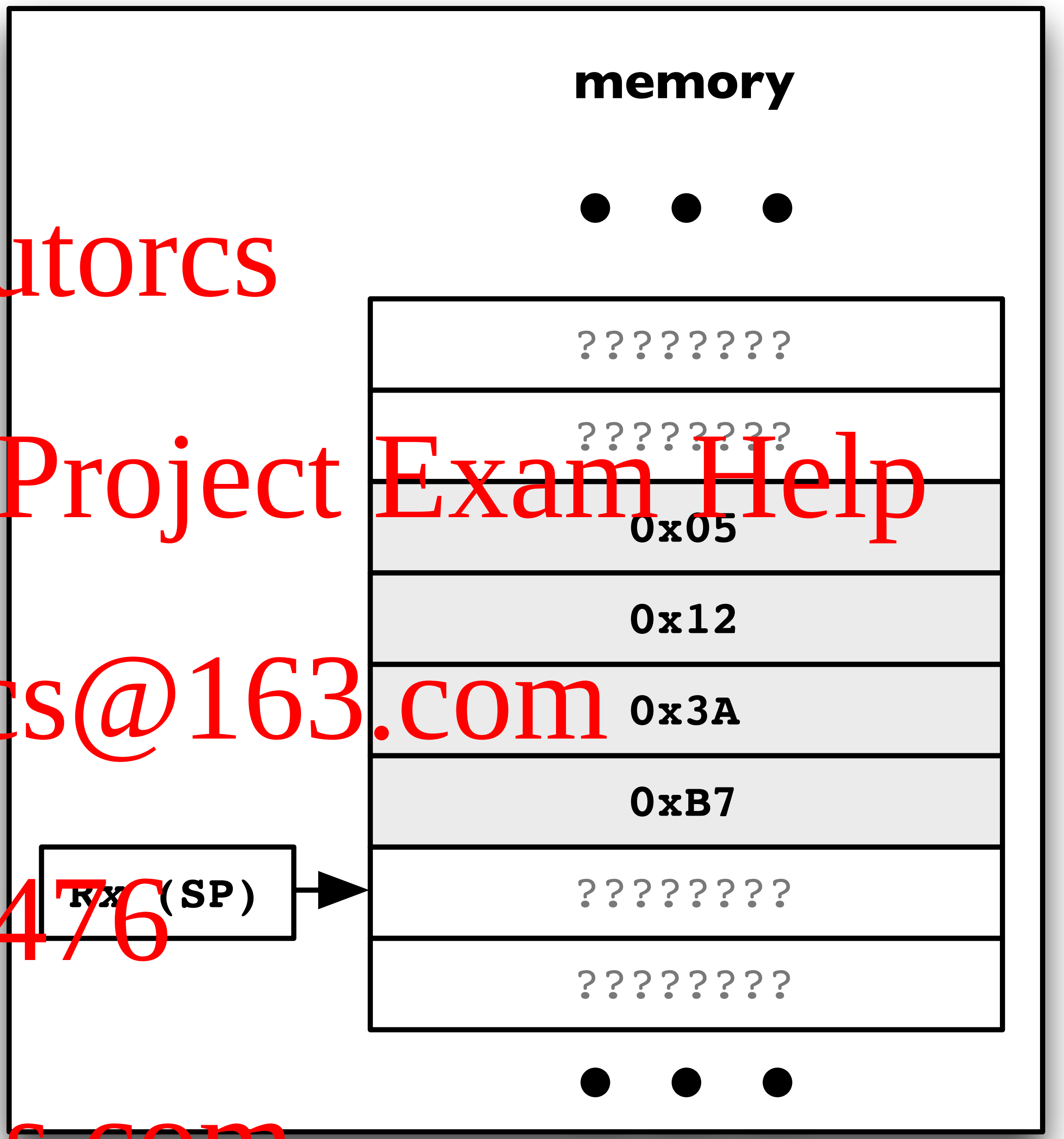
Stack Growth Convention



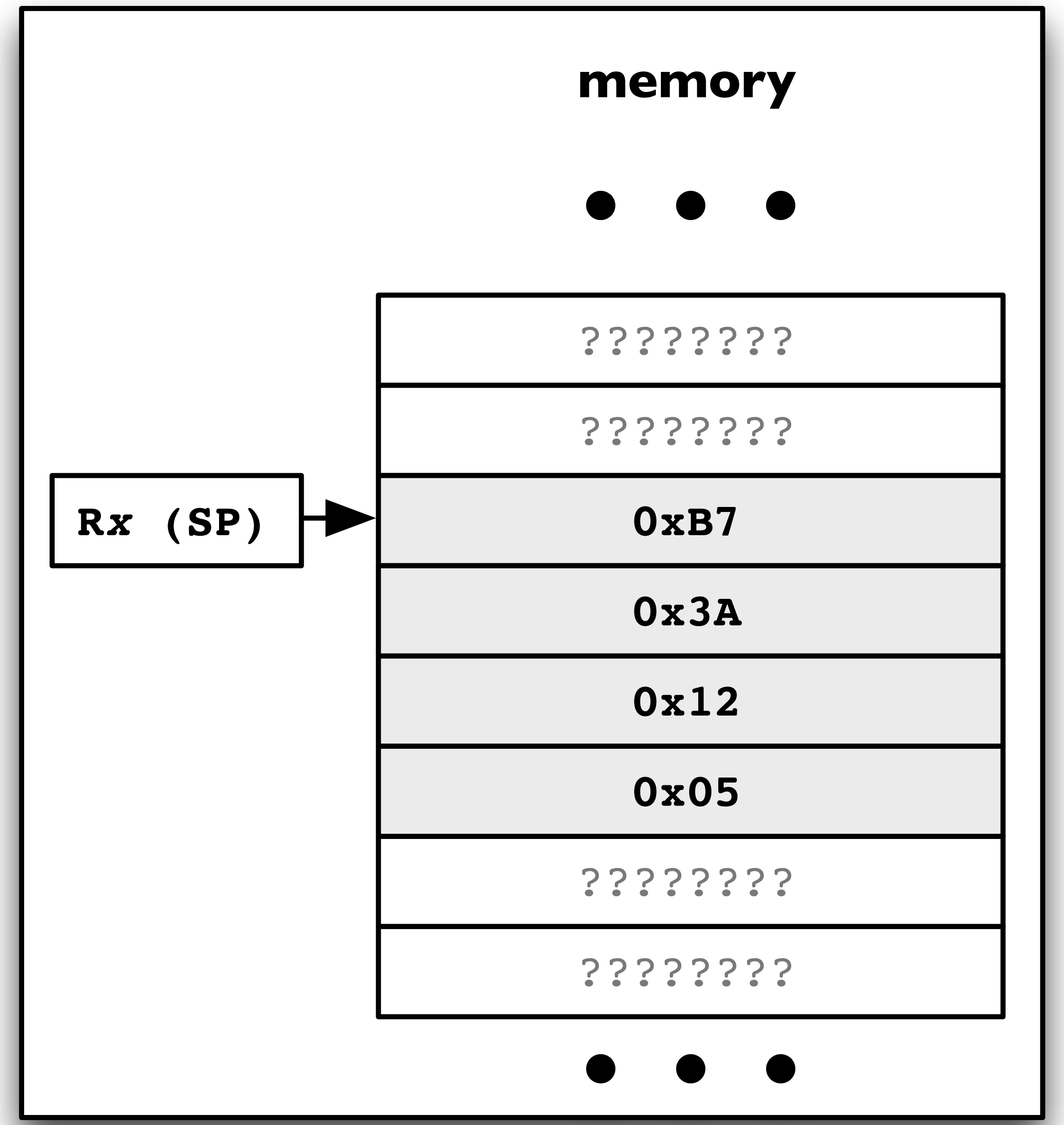
full descending stack



empty ascending stack



empty descending stack



full ascending stack

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

2.2 – Stacks (continued)

CSU11022 – Introduction to Computing II

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

Stack Implementation in ARM Assembly Language

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!)

```
  .equ StackSize, 0x400

Main:

  LDR     R12, =myStackTop

  @ your program goes here
  @ including pushing/popping data on/off the
stack

End_Main:
  BX      LR


  .section  .data

myStack:
  .space    StackSize
myStackTop:
```

Stack Implementation – push

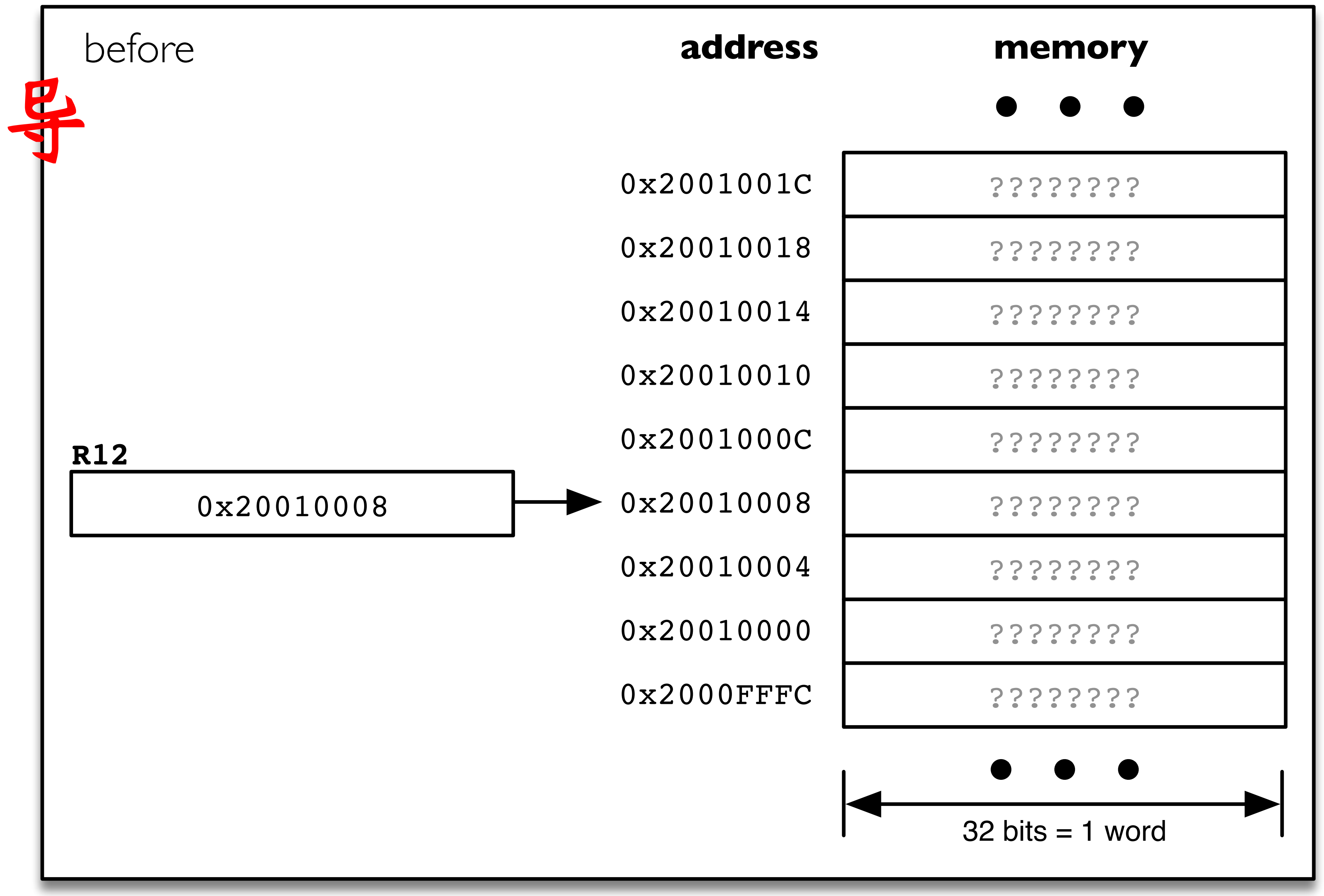
Assume **full descending** stack growth

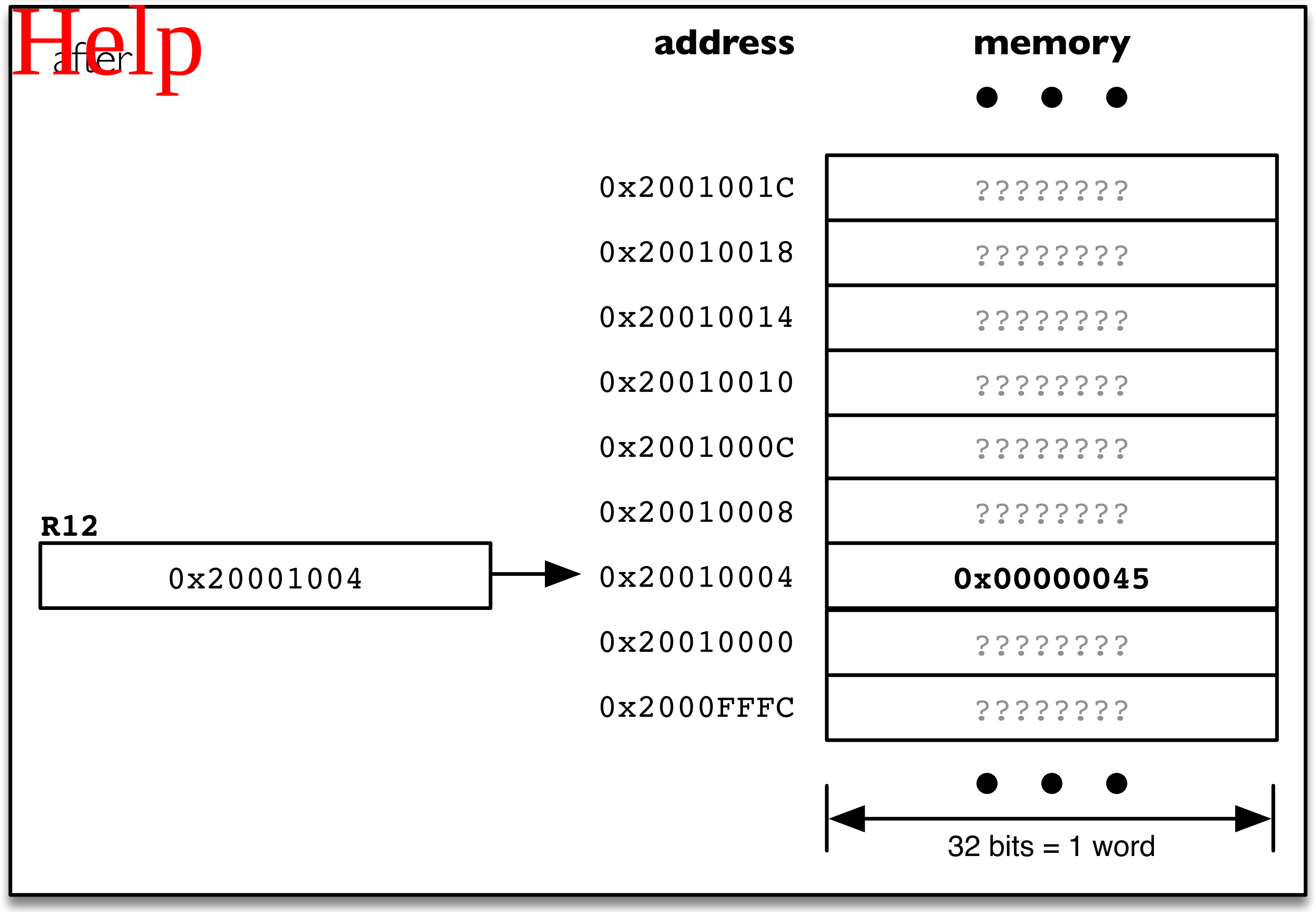
To push a word onto the [illegible]

1. decrement the stack point[illegible]es
   (4 bytes = 1 word = 32 bits[illegible]
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

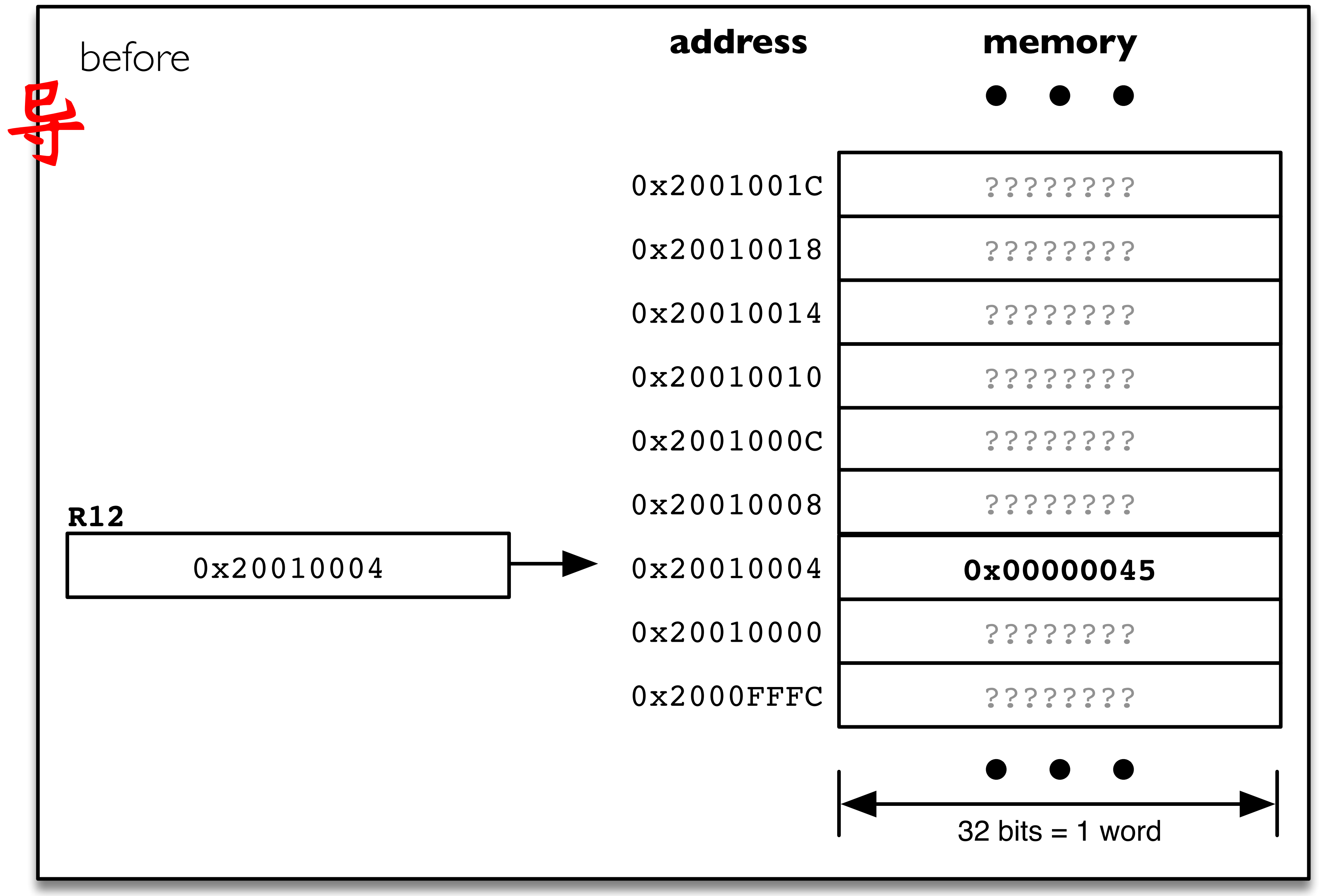
Again, assume full descending stack growth convention

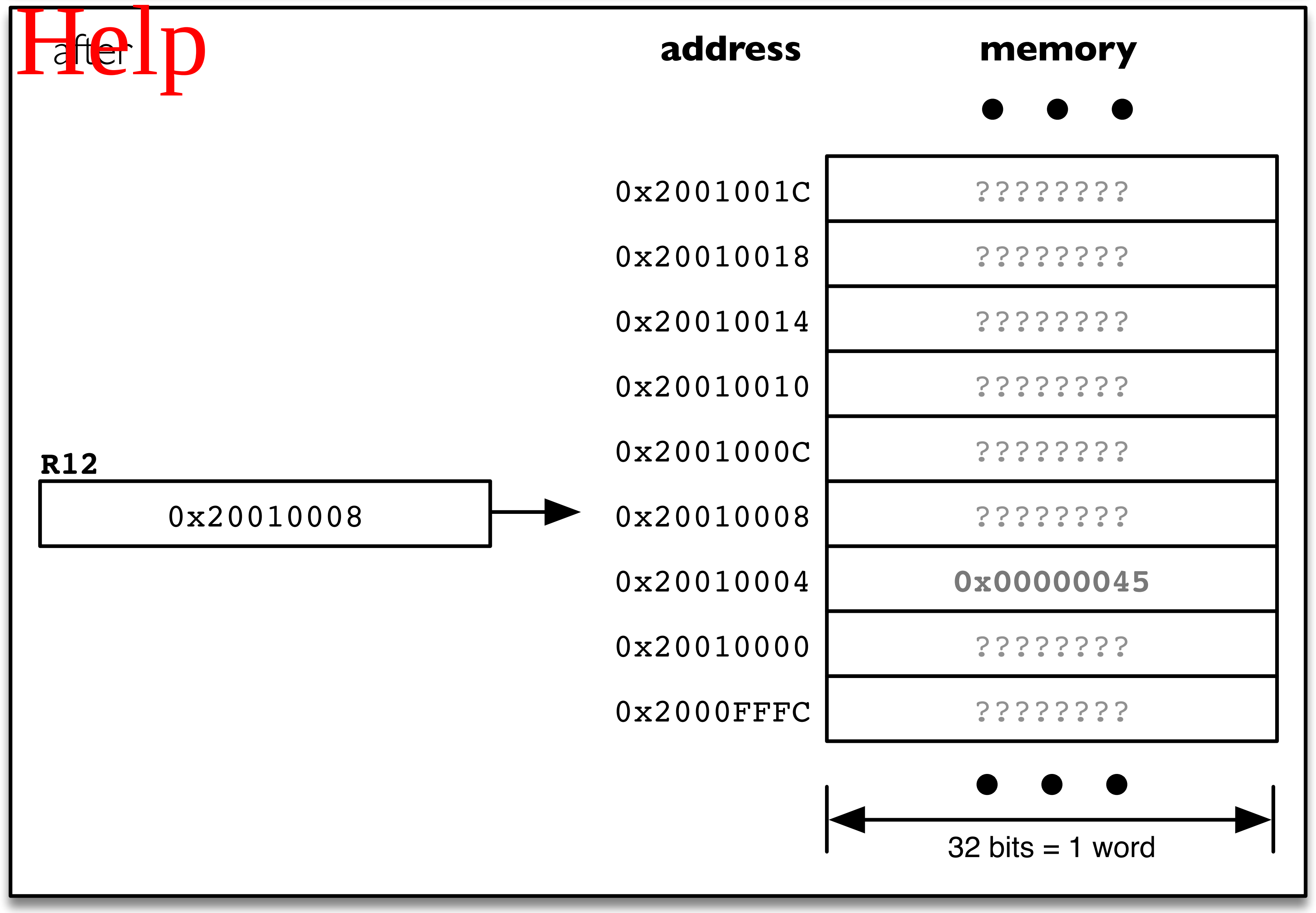
To pop a word off the sta[illegible]

1. load the word from memo[illegible]cation 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
```





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

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

```
; 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

The System Stack

In general, stacks …

- can be located anywhere in m
- 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 0x20010000 into SP (R13)
LDR     SP, =0x20010000
```

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

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

2.3 – LoaD Multiple and STore Multiple (LDM/STM)

CSU11022 – Introduction to Computing II

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

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 R[illegible] R3 to memory at the address in R12
STR     R1, [R12]
STR     R2, [R12, #4]
STR     R3, [R12, #8]
```

```
; load R1, R2 and R3 with contents of memory at the address in R12

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 the address in R12

STMIA   R12, {R1-R3}

; load R1, R2 and R3 with contents of memory at the address in R12

LDMIA   R12, {R1-R3}
```

LDM / STM Syntax

Consider the following STM instruction …

```
STMIA [illegible]2, {R1-R3}
```

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>

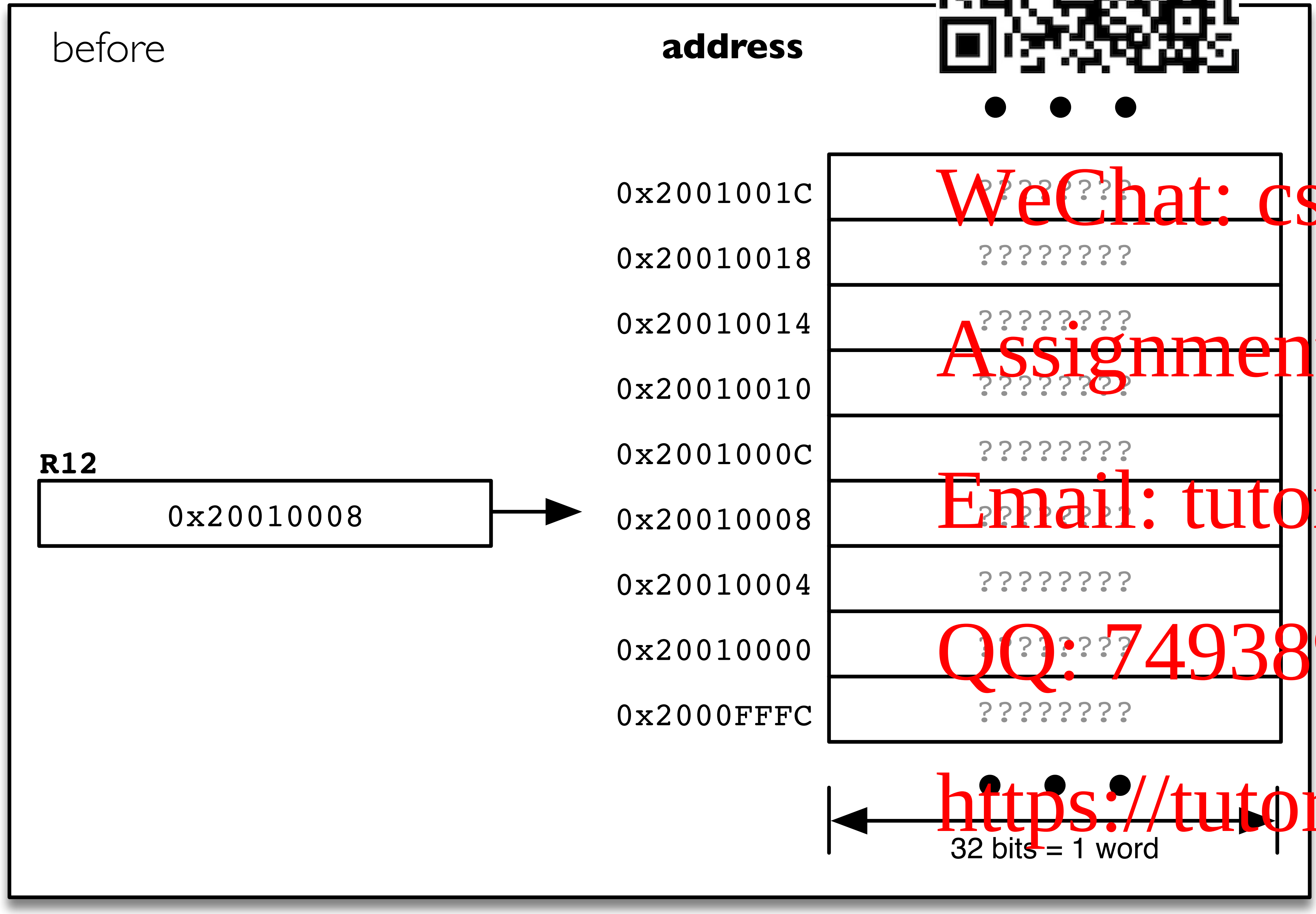
second register is stored at <base address> + 4

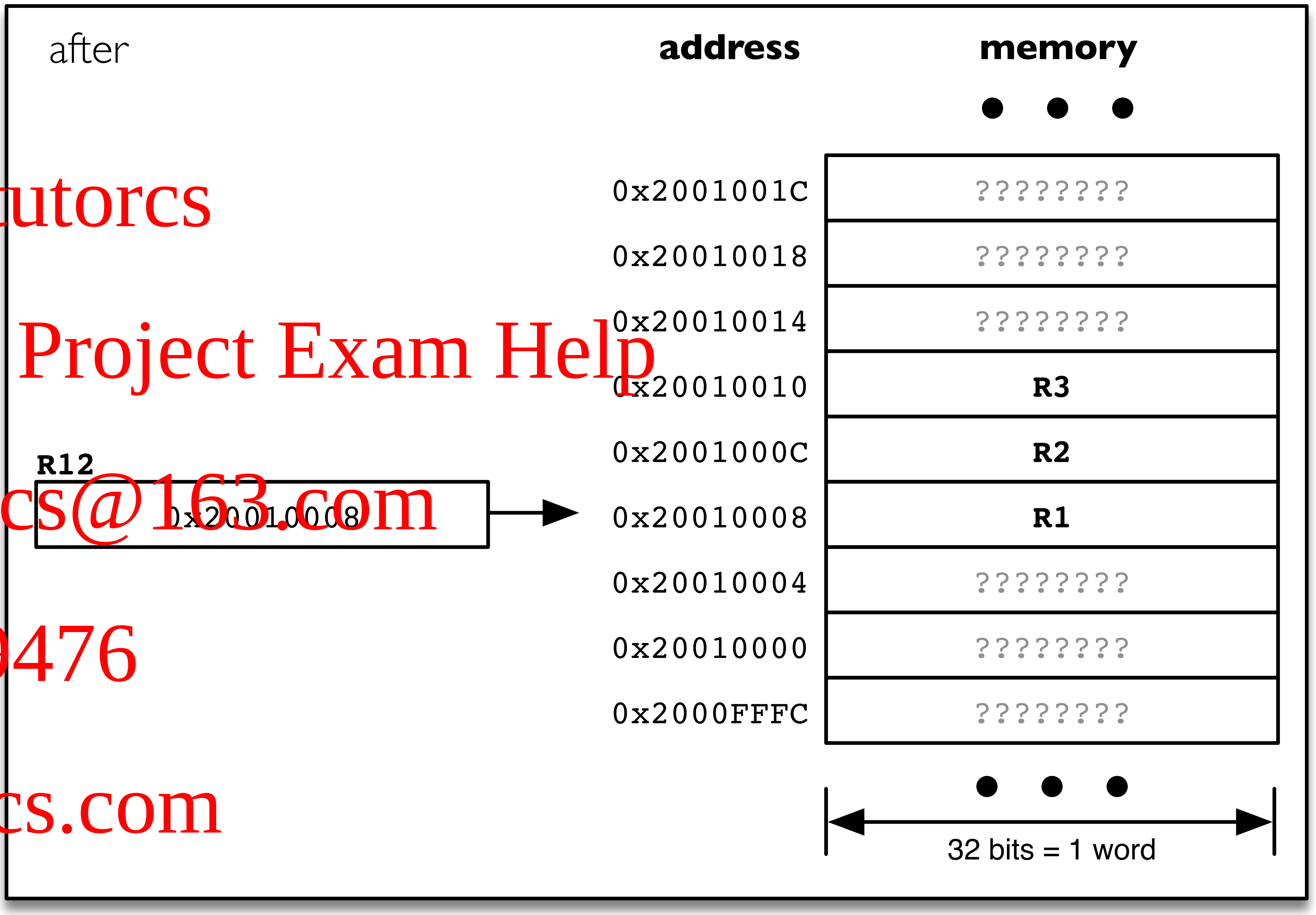
third register is stored at <base address> + 8

Value (address) in base register R12 remains unchanged

LDM / STM Behaviour

STMIA R12, {R1-R3}





LDM / STM Modes of Operation

Modes of operation for LDM and STM instructions

Behaviour	LDM	STM
Increment After	LDMIA	STMIA
Decrement Before	LDMDB	STMDB

Register list

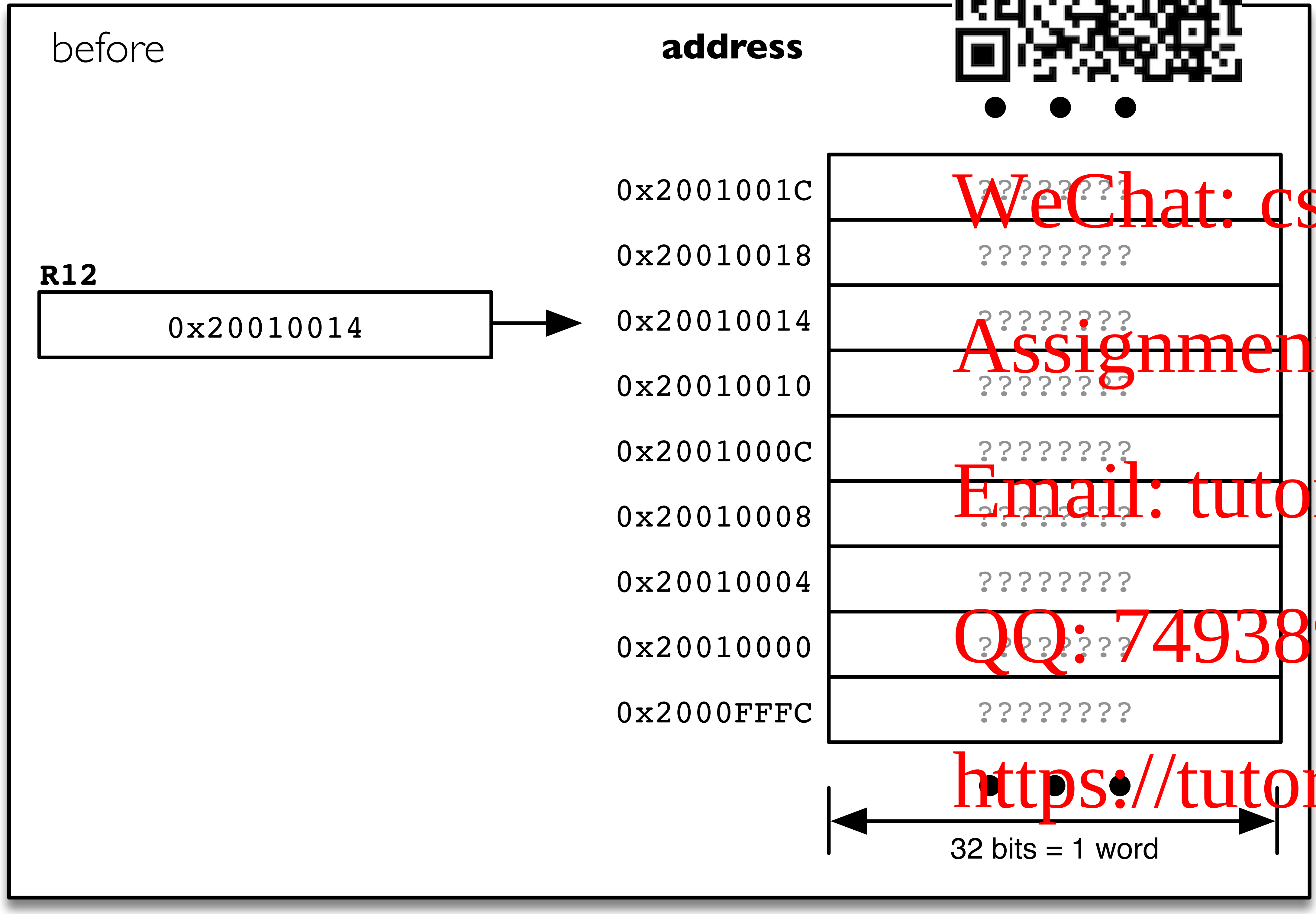
e.g. {R1-R3, R10, R7-R9}

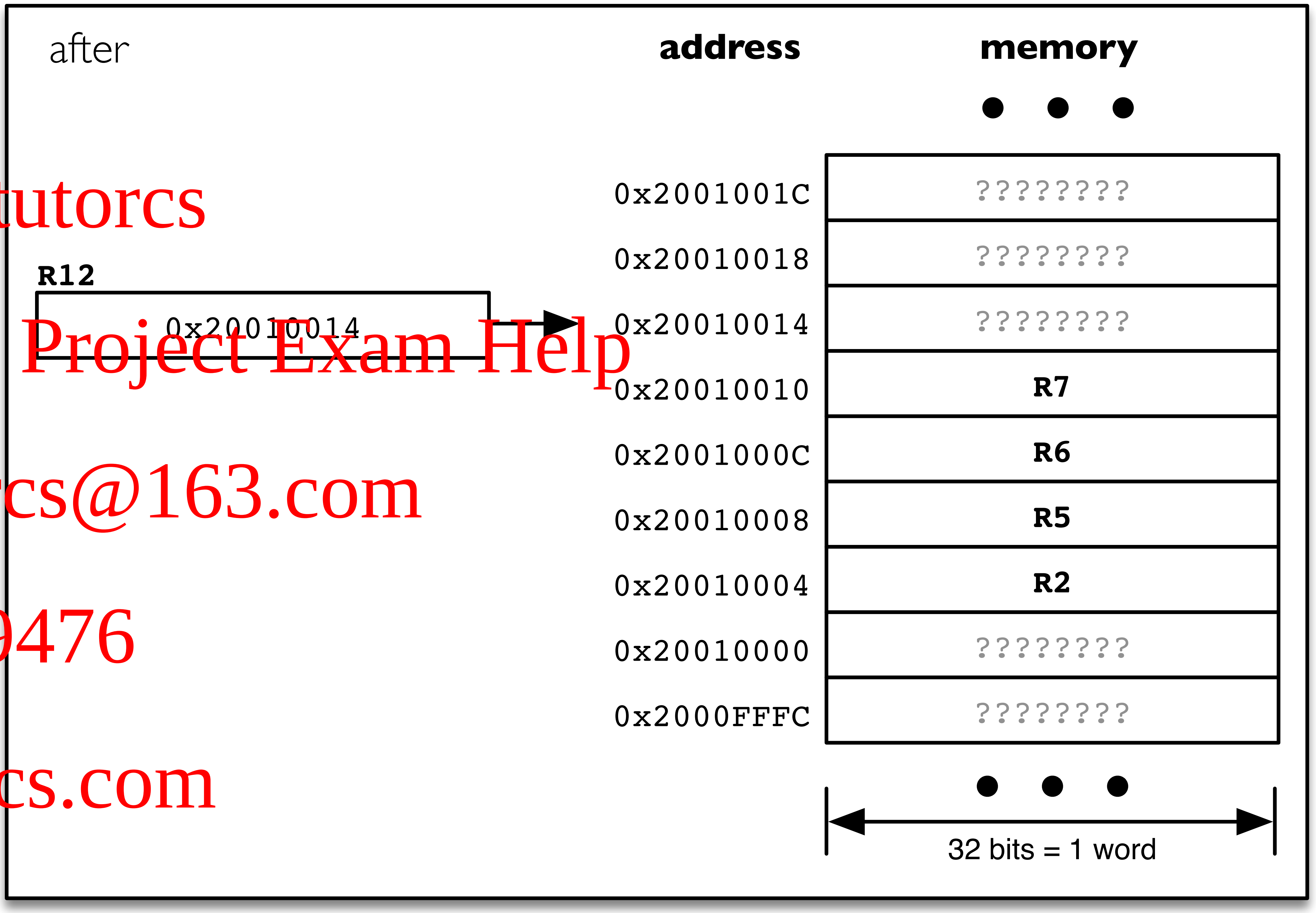
Order in which registers are specified is not important

For both LDM and STM, the lowest register is always loaded from the lowest address, regardless of mode of operation (IA, DB)

LDM / STM Behaviour

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





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

2.4 – LDM, STM and Stacks

CSU11022 – Introduction to Computing II

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

LDM / STM with Stacks

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

Choose IA/DB operation a e to stack growth convention

- increment/decrement, before/a

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

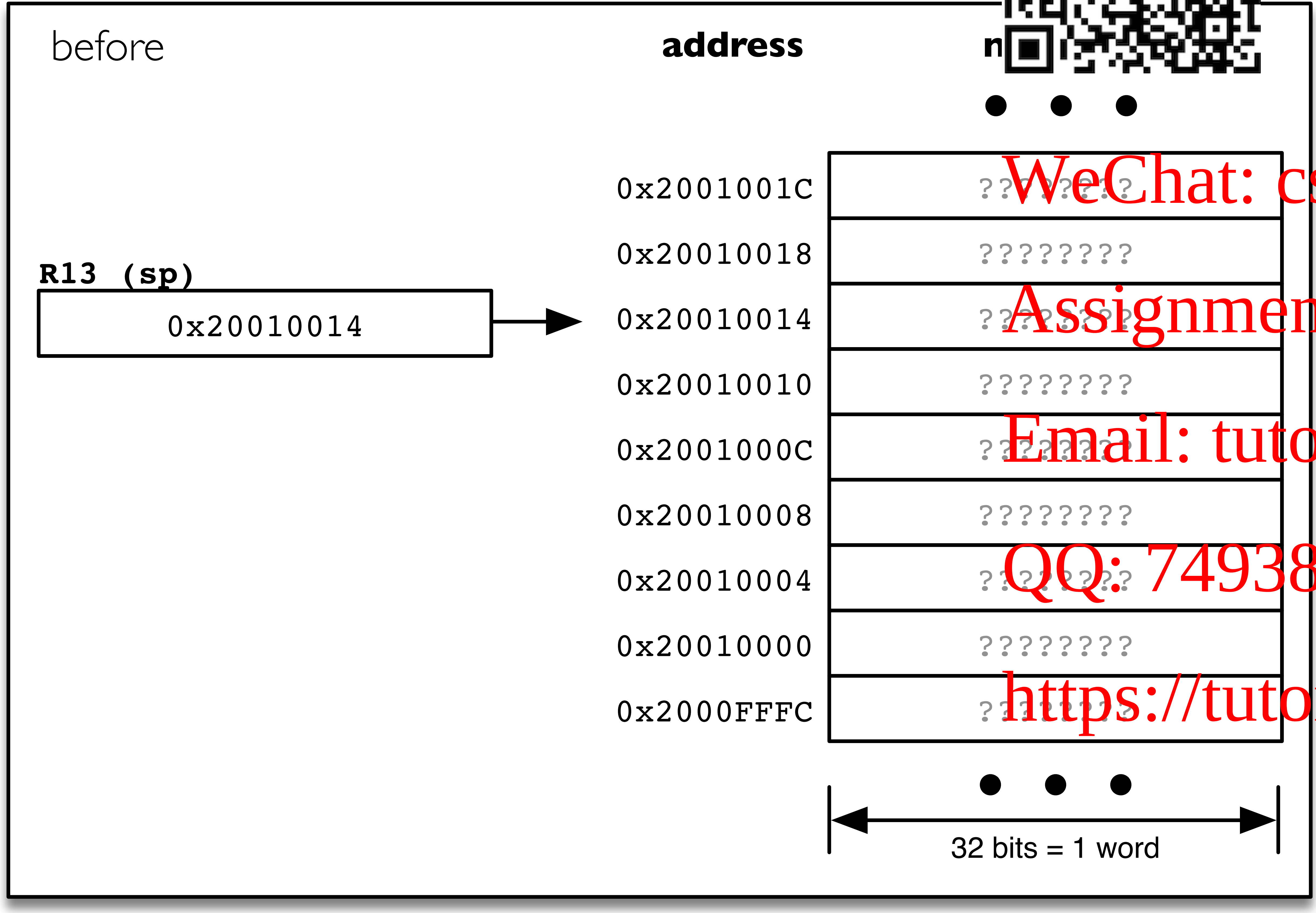
- **Use ! syntax to modify LDM/STM behaviour so the stack pointer is updated**

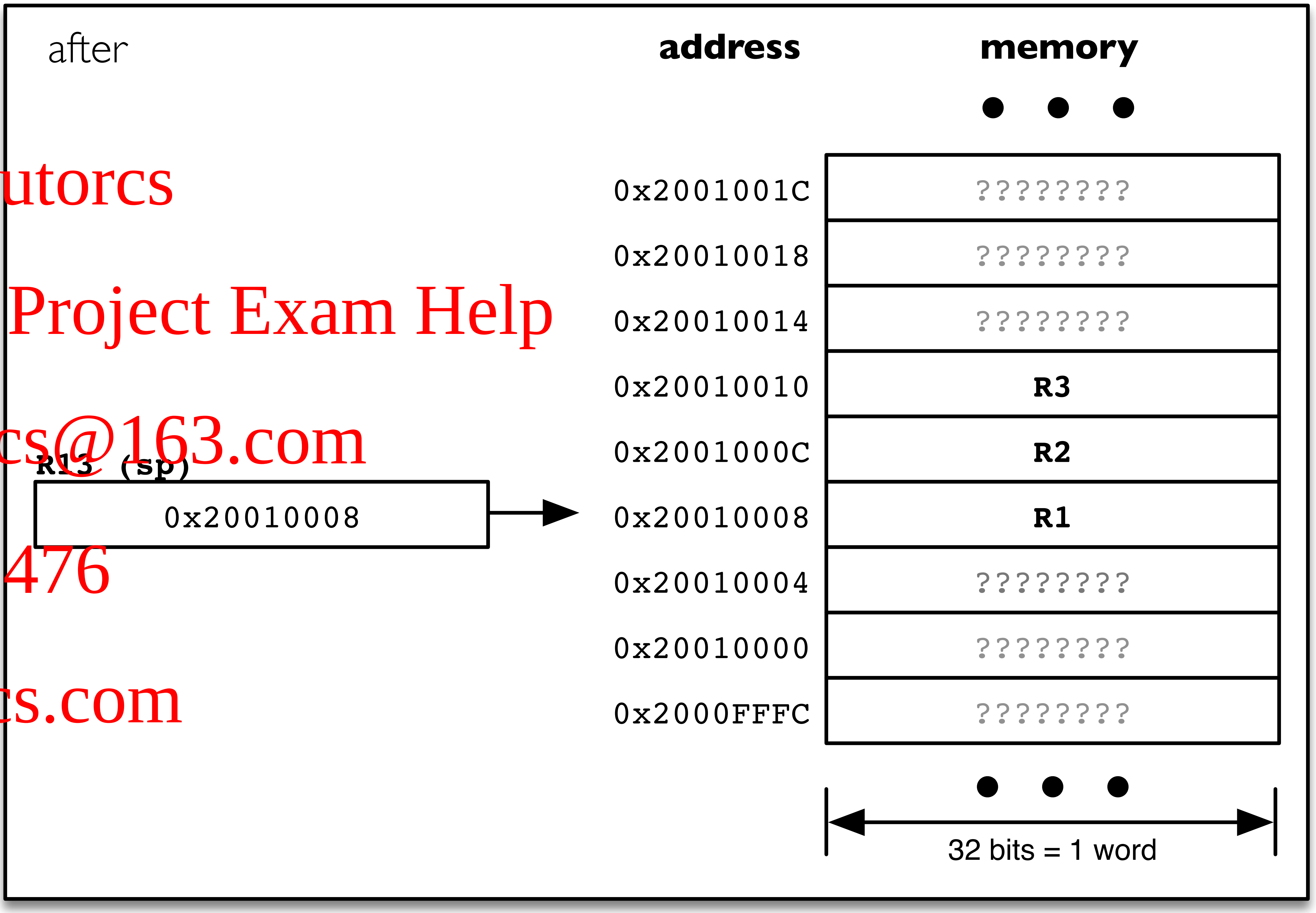
```
STMDB   SP!, {R1-R3}    ; or PUSH {R1-R3}
LDMIA   SP!, {R1-R3}    ; or POP {R1-R3}
```

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, R[illegible] on to a full descending stack with R13 (or SP) as the stack point[illegible]

```
STMDB   SP!, {R1-R3,[illegible]
```

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}
```

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}             ; PUSH
```

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}             ; POP
```

PUSH and POP Synonyms

Pushing and Popping on the System Stack is a very common operation

PUSH {…} can be used as a synonym for STMFD SP!, {…}

```
STMFD   SP!, {R1-R3,R5}
PUSH    {R1-R3,R5}
```

POP {…} can be used as a synonym for LDMFD SP!, {…}

```
LDMFD   SP!, {R1-R3,R5}
POP     {R1-R3,R5}
```

LDM / STM Synonyms for Stacks

Stack growth convention			pop	
	STM mode	stack-oriented synonym	LDM mode	stack-oriented synonym
full descending	STMDB	**STMFD or PUSH**	LDMIA	**LDMFD or POP**
empty ascending	STMIA	STMEA	LDMDB	LDMEA

Pushing / Popping *any*-Size Data

In theory, we 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
```

However, ARM Cortex-M requires the stack pointer to be word-aligned and the least significant two bits of the SP are ignored

But you could push/pop non-word data to/from your own (non-system) stack

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, [R10, #-4]!

; push 3 half words from R1, R2 and R3
STRH    R1, [R10, #-2]!
STRH    R2, [R10, #-2]!
STRH    R3, [R10, #-2]!

; push 2 words from R4 and R5
STR     R4, [R10, #-4]!
STR     R5, [R10, #-4]!
```

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