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

3.1 Subroutines

CSU11022 – Introduction to Computing II

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

Subroutines

Programs can be **decomposed** into blocks of instructions, each performing some well-defined task

- compute $\mathbf{x^y}$
- find the length of a NULL-terminated string
- convert a string from UPPER CASE to lower case
- play a sound

We would like to avoid repeating the same set of operations throughout our programs

- write the instructions to perform some specific task **once**
- **invoke** the set of instructions **many times** to perform the same task

Methods in the Java world!

Functions or Procedures elsewhere

Example – UPPER CASE


```
address = string1;
ch = byte[address];
while (ch != NULL) {
    if (ch ≥ ‘a’ && char ≤ ‘z’) {
        ch = char & 0xFFFFFFDF;
        byte[address] = ch;
    }
    address = address + 1;
    char = byte[address] ;
}

address = string2;
ch = byte[address];
while (ch != NULL) {
    if (ch ≥ ‘a’ && char ≤ ‘z’) {
        ch = ch & 0xFFFFFFDF;
        byte[address] = ch;
    }
    address = address + 1;
    ch = byte[address];
}
```

Example – UPPER CASE

```
// UPPER CASE
void upr (address)
{
    ch = byte[address];
    while (ch != NULL) {
        if (ch ≥ ‘a’ && char ≤ ‘z’) {
            ch = ch & 0xFFFFFFDF;
            byte[address] = ch;
        }
        address = address + 1;
        ch = byte[address] ;
    }
}

address = string1;
upr(address);

address = string2;
upr(address);
```


Define upr(...)

Invoke upr(...) twice

Example – UPPER CASE

```
// UPPER CASE
void upr (address)
{
    ch = byte[address];
    while (ch != NULL) {
        if (ch ≥ ‘a’ && char ≤ ‘z’) {
            ch = ch & 0xFFFFFFDF;
            byte[address] = ch;
        }
        address = address + 1;
        ch = byte[address] ;
    }
}


upr(string1);


upr(string2);
```


Define upr(...)

Invoke upr(...) twice

Call / Return Example


```
address = string1;
upr(address);
```

A

```
address = string2;
upr(address);
```

B

How do we know whether to branch back to A or B ?

```
void upr (address)
{
    ch = byte[address]
    while (ch != NULL) {
        if (ch ≥ 'a' && ch ≤ 'z') {
            ch = ch & 0xFFFFFFDF;
            byte[address] = ch;
        }
        address = address + 1;
        ch = byte[address];
    }
}
```


Branching to a subroutine: branch to the address (or label) of the first instruction in the subroutine (simple flow control … easy!)

Returning from a subroutine: must have remembered the address that we originally branched from (**return address**, A or B in the example above)

Invoking the UPR subroutine


```
Main:

  @
  @ Program to convert two strings to UPPERCASE
  @ Assume the first string starts at the address in R1
  @ Assume the second string starts at the address in R2
  @

  MOV     R0, R1      @ copy address of first string into R0
  BL      upr         @ invoke upr subroutine

  MOV     R0, R2      @ copy address of second string into R0
  BL      upr         @ invoke upr subroutine (again)

End_Main:
  BX      LR
```

Implementing the UPRCASE subroutine


```
@
@ upr subroutine
@ Converts a NULL-terminated string to upper case
@
@ Parameters:
@   R0:  string start address
@
upr:
.LwhUpr:
  LDRB     R4, [R0], #1       @ char = byte[address++]
  CMP      R4, #0             @ while ( char != 0 )
  BEQ      .LeWhUpr           @ {
  CMP      R4, #'a'           @  if (char >= 'a'
  BLO      .LeIfLwr           @      &&
  CMP      R4, #'z'           @      char <= 'z')
  BHI      .LeIfLwr           @  {
  BIC      R4, #0x00000020    @   char = char AND NOT 0x00000020
  STRB     R4, [R0, #-1]      @   byte[address - 1] = char
.LeIfLwr:                     @  }
  B        .LwhUpr            @ }
.LeWhUpr:                     @

  BX       LR
```

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

3.2 Subroutines – Unintended Side Effects

CSU11022 – Introduction to Computing II

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

What's wrong with this program?


```
Main:
  BL      subroutine1       @ invoke subroutine1

End_Main:
  BX      LR


@ subroutine1
subroutine1:
  ADD     R0, R1, R2        @ do something
  BL      subroutine2       @ call subroutine2
  ADD     R3, R4, R5        @ do something else
  BX      LR                @ return from subroutine1


@ subroutine2
subroutine2:

  BX      LR                @ just return from subroutine2
```

Saving the Link Register

Save the contents of the link register on the system stack at the start of every subroutine

Restore the contents of the link register immediately before returning from every subroutine


```
@ subroutine1
subroutine1:
  PUSH    {LR}
  ADD     R0, R1, R2         @ do something
  BL      subroutine2        @ call subroutine2
  ADD     R3, R4, R5         @ do something else
  POP     {LR}
  BX      LR                 @ return from subroutine1
```

Implement this fix now in the sideeffects1 example from the CSU1102x GitLab repository. Verify that the fix works.

Saving the Link Register

More efficiently, we could restore the saved LR to the PC, avoiding the need for the BX instruction (preferred)

```
@ subroutine1
subroutine1:
  PUSH    {LR}
  ADD     R0, R1, R2        @ do something
  BL      subroutine2       @ call subroutine2
  ADD     R3, R4, R5        @ do something else
  POP     {PC}
```


Implement this fix now in the sideeffects1 example from the CSU1102x GitLab repository. Verify that the fix works.

What’s wrong with this program?

Imagine we are using our upr subroutine again …

```
@
@ upr subroutine
@ Converts a NULL-terminated string to upper case
@
@ Parameters:
@   R0:  string start address
@
upr:
.LwhUpr:
  LDRB    R4, [R0], #1       @ char = byte[address++]
  CMP     R4, #0             @ while ( char != 0 )
  BEQ     .LeWhUpr           @ {
  CMP     R4, #'a'           @  if (char >= 'a'
  BLO     .LeIfLwr           @      &&
  CMP     R4, #'z'           @      char <= 'z')
  BHI     .LeIfLwr           @  {
  BIC     R4, #0x00000020    @   char = char AND NOT 0x00000020
  STRB    R4, [R0, #-1]      @   byte[address - 1] = char
.LeIfLwr:                    @  }
  B       .LwhUpr            @ }
.LeWhUpr:                    @

  BX      LR
```

What's wrong with this program?

… and then use the upr subroutine to convert two strings to UPPER CASE but this time our second string starts at an address in R4 …

```
Main:

  @
  @ Program to convert two strings to UPPERCASE
  @ Assume the first string starts at the address in R1
  @ Assume the second string starts at the address in R4
  @

  MOV     R0, R1      @ copy address of first string into R0
  BL      upr         @ invoke upr subroutine

  MOV     R0, R4      @ copy address of second string into R0
  BL      upr         @ invoke upr subroutine (again)

End_Main:
  BX      LR
```

Unintended Side Effects

We want (need?) to be able to write subroutines in isolation, independently from the rest of our program

When designing and writing subroutines, clearly and precisely define what effect the subroutine has

Effects outside this definition should be considered **unintended** and should be **hidden** by the subroutine

In general, subroutines should save the contents of the registers they use at the start of the subroutine and should restore the saved contents before returning

SOLUTION: PUSH register contents on the stack at the start of a subroutine, POP them off at the end

Hiding Unintended Side Effects


```
@
@ upr subroutine
@ Converts a NULL-terminated string to upper case
@
@ Parameters:
@   R0:  string start address
@
upr:
  PUSH    {R0, R4, LR}

.LwhUpr:
  LDRB    R4, [R0], #1        @ char = byte[address++]
  CMP     R4, #0              @ while ( char != 0 )
  BEQ     .LeWhUpr            @ {
  CMP     R4, #'a'            @  if (char >= 'a'
  BLO     .LeIfLwr            @       &&
  CMP     R4, #'z'            @       char <= 'z')
  BHI     .LeIfLwr            @  {
  BIC     R4, #0x00000020     @   char = char AND NOT 0x00000020
  STRB    R4, [R0, #-1]       @   byte[address - 1] = char
.LeIfLwr:                     @  }
  B       .LwhUpr             @ }
.LeWhUpr:                     @

  POP    {R0, R4, PC}
```

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

3.3 Subroutines – Parameter Passing

CSU11022 – Introduction to Computing II

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

Information must be passed to a subroutine using a fixed and well defined interface, known to both the subroutine and calling programs

upr subroutine had single address parameter

```
address = string1;
upr(address);

address = string2;
upr(address);

. . .

upr(address)
{
    . . .
}
```


Simplest way to pass parameters to a subroutine is to use well defined registers, e.g. for `upr` subroutine, use `R0` for the address of the string

Example – fill

Design and write an ARM Assembly Language subroutine that fills a sequence of words in memory with the same 32-bit value

Pseudo-code solution

```
fill (address, length, value)
{
    count = 0;
    while (count < length)
    {
        word[address] = value;
        address = address + 4;
        count = count + 1;
    }
}
```


3 parameters

address start address in memory

length number of words to store

value value to store

Example – fill subroutine


```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@
@ Parameters:
@       R0: address - address of first word to be filled
@       R1: length - number of words to be filled
@       R2: value - value to store in each word

fill:
        PUSH    {R0-R2,R4,LR}
        MOV     R4, #0                  @ count = 0;
.LwhFill:
        CMP     R4, R1                  @ while (count < length)
        BHS     .LeWhFill               @ {
        STR     R2, [R0, R4, LSL #2]    @   word[address+(count*4)] = value;
        ADD     R4, #1                  @   count = count + 1;
        B       .LwhFill                @ }
.LeWhFill:                              @
        POP     {R0-R2,R4,PC}
```

Interface

In high level languages, the interface is defined by the programmer and the compiler implements and enforces it

In assembly language, the interface must be defined, implemented and enforced by the programmer

ARM Architecture Procedure Call Standard (AAPCS) is a technical document that dictates how a high-level language interface should be implemented in ARM Assembly Language (or machine code!!)

Enforcing the standard in your programs is your job!!

Simplified Subroutine Interface Guidelines

(based on AAPCS)

Registers	Use
`R0 … R3`	Passing parameters to subroutines – avoid using for other variables – **corruptible (not saved/restored on stack)**
`R4 … R12`	Local variables within subroutines – **preserved (saved/ restored on stack)**
`R13 (SP)`	Stack Pointer – **preserved through proper use**
`R14 (LR)`	Link Register – **corrupted through subroutine call**
`R15 (PC)`	Program Counter

Adhering to these guidelines will make it easier to write large programs with many subroutines

Example – fill revisited

Based on these guidelines, we could re-write fill (note that I was already adhering to the guidelines for passing parameters but I didn't need to save R0 or R1!!)

```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@
@ Parameters:
@       R0: address - address of first word to be filled
@       R1: length - number of words to be filled
@       R2: value - value to store in each word

fill:
  PUSH    {R4,LR}
  MOV     R4, #0                  @ count = 0;
.LwhFill:
  CMP     R4, R1                  @ while (count < length)
  BHS     .LeWhFill               @ {
  STR     R2, [R0, R4, LSL #2]    @   word[address+(count*4)] = value;
  ADD     R4, #1                  @   count = count + 1;
  B       .LwhFill                @ }
.LeWhFill:                        @
  POP     {R4,PC}
```

Example – calling fill

Recall the fill interface … this is all we need to invoke fill

```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@
@ Parameters:
@       R0: address – address of first word to be filled
@       R1: length – number of words to be filled
@       R2: value – value to store in each word
```


Note that we only need to know the interface.

We don't need to know how `fill` is implemented!

To invoke fill assuming R5 contains the start address, R9 the length to fill and R8 the value to fill memory with …

```
    MOV     R0, R5          @ address parameter
    MOV     R1, R9          @ length parameter
    MOV     R2, R8          @ value parameter

    BL      fill            @ invoke fill
```

1. move parameters into place
2. Invoke subroutine

Example – count1s

Design and write an ARM Assembly Language subroutine that counts the number of set bits in a word

```
@ count1s subroutine
@ Counts the number of set bits (1s) in a word
@ Parameters:
@   R0: wordval – word in which 1s will be counted
@ Return:
@   R0: count of set bits (1s) in wordval
count1s:
  PUSH    {R4, LR}        @ save registers
  MOV     R4, R0          @ copy wordval parameter to local variable
  MOV     R0, #0          @ count = 0;
.LwhCount1s:
  CMP     R4, #0          @ while (wordval != 0)
  BEQ     .LeWhCount1s    @ {
  MOVS    R4, R4, LSR #1  @  wordval = wordval >> 1; (update carry)
  ADC     R0, R0, #0      @  count = count + 0 + carry;
  B       .LwhCount1s     @ }
.LeWhCount1s:
  POP     {R4, PC}        @ restore registers
```

Interface Guidelines for Return Values

Use R0 for returning values from subroutines

Registers	Use
R0 … R3	Passing parameters to subroutines or returning values from subroutines – avoid using for other variables – **corruptible**
R4 … R12	Local variables within subroutines – **preserved (saved/ restored on stack)**
R13 (SP)	Stack Pointer – **preserved through proper use**
R14 (LR)	Link Register – **corrupted through subroutine call**
R15 (PC)	Program Counter

R0 used to pass wordval parameter **and** return result value from count1s subroutine (an implementation decision – real AAPCS compilers would also do this!)

Example – count1s

Recall the count1s interface

```
@ count1s subroutine
@ Counts the number of set bits (1s) in a word
@ Parameters:
@   R0: wordval – word in which 1s will be counted
@ Return:
@   R0: count of set bits (1s) in wordval
```


Note again that we only need to know the interface … we don't need to know how count1s is implemented

Call count1s, assuming R7 contains the word value to be passed to count1s

```
    ...     ...
    MOV     R0, R7          @ prepare the parameter
    BL      count1s         @ call count1s
    ADD     R5, R5, R0      @ do something useful with the result
    ...     ...
```

Unintended Side Effects – Summary

Good practice to save …

any registers used for local variables (R4 … R12)

the link register (LR / R14)

(and optionally, registers used for parameters)

but not registers used for return values

… on the system stack at the start of every subroutine

Restore exactly the same saved registers at the end of every subroutine

Avoids unintended side effects and simplifies subroutine interface design

Remember: a subroutine must pop off everything that was pushed on to the stack before it returns

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

3.4 Subroutines – Recursion

CSU11022 – Introduction to Computing II

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

Subroutines can invoke themselves – recursion

Example: Design, write and test a subroutine to compute x^n

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x & \text{if } n = 1 \\ \left(x^2\right)^{n/2} & \text{if } n \text{ is even} \\ x.\left(x^2\right)^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

$$x^5 = x \times x \times x \times x \times x$$

$$x^5 = x \times (x \times x) \times (x \times x)$$

$$x^5 = x \times (x \times x)^2$$

$$x^5 = x \times (x^2)^2$$

$$x^5 = x^{1+(2\times 2)}$$

Subroutines can invoke themselves – recursion

Example: Design, write and test a subroutine to compute x^n

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x & \text{if } n = 1 \\ \left(x^2\right)^{n/2} & \text{if } n \text{ is even} \\ x.\left(x^2\right)^{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

$$x^9 = x \times (x^2)^{(9-1)/2}$$

$$x^9 = x \times (x^2)^4$$

$$x^9 = x \times ((x^2)^2)^{4/2}$$

$$x^9 = x \times ((x^2)^2)^2$$

Example – power


```
power (x, n)
{
    if (n == 0)
    {
        result = 1;
    }
    else if (n == 1)
    {
        result = x;
    }
    else if (n & 1 == 0) // n is even
    {
        result = power (x * x, n >> 1);
    }
    else // n is odd
    {
        result = x * power (x * x, n >> 1)
    }
}
```

Example – power (1)


```
@
@ power subroutine
@ Computes x^n
@
@ Parameters:
@   R0:   x
@   R1:   n
@
@ Return:
@   R0:   x^n
@
power:
  PUSH  {R4-R6,LR}       @ save registers
  MOV   R4, R0           @ Move parameters to local registers
  MOV   R5, R1           @  Doing this makes managing registers in subroutines
                         @  *much* simpler. When we call a subroutine from the
                         @  body of this subroutine, the parameter registers
                         @  (R0-R3) will already be free for us to use because
                         @  we have moved the original parameters to other
                         @  registers.
```

Example – power (2)


```
  CMP   R5, #0          @ if (n == 0) {
  BNE   .LpowerNe0

  MOV   R0, #1          @   result = 1;

  B     .LpowerEndIf    @ }
.LpowerNe0:
  CMP   R5, #1          @ else if (n == 1) {
  BNE   .LpowerNe1

  MOV   R0, R4          @   result = x;

  B     .LpowerEndIf    @ }
.LpowerNe1:
```

Example – power (3)


```
.LpowerNe1:
  AND    R6, R5, #1      @ else if (n & 1 == 0) { // n is even
  CMP    R6, #0
  BNE    .LpowerNeEven

  MUL    R0, R4, R4      @   result = power (x * x, n >> 1);
  MOV    R1, R5, LSR #1  @   // using LSR by 1 bit to implement division by 2
  BL     power

  B      .LpowerEndIf    @ }
.LpowerNeEven:

                         @ else {
  MUL    R0, R4, R4      @   result = x * power (x * x, n >> 1);
  MOV    R1, R5, LSR #1
  BL     power
  MUL    R0, R4, R0

.LpowerEndIf:            @ }

  POP    {R4-R6, PC}     @ return result;
```

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

3.5 Subroutines – Passing Parameters on the Stack

CSU11022 – Introduction to Computing II

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

Passing Parameters on the Stack

If there are insufficient registers to pass parameters to a subroutine, the system stack can be used

- Commonly used by high-level languages
- Number of parameters is limited only by the remaining space on the stack

General approach

- Calling program pushes parameters onto the stack
- Subroutine accesses parameters on the stack, relative to the stack pointer
- Calling program pops parameters off the stack after the subroutine has returned

Example – fill with Stack Parameters

Re-write the fill subroutine to pass parameters on the stack (instead of registers)

Pseudo-code reminder


```
fill (address, length, value)
{
    count = 0;
    while (count < length)
    {
        word[address] = value;
        address = address + 4;
        count = count + 1;
    }
}
```

Example – fill with Stack Parameters


```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@ Parameters
@       [sp+0]: value – value to store in each word (1st Top Of Stack)
@       [sp+4]: length – number of words to be filled (2nd Top Of Stack)
@       [sp+8]: address – address of first word to be filled (3rd Top Of Stack)
fill:
  PUSH    {R4-R7,lr}            @ save registers

  LDR     R4, [SP, #28]       @ load address parameter (not popping)
  LDR     R5, [SP, #24]       @ load length parameter (not popping)
  LDR     R6, [SP, #20]       @ load value parameter (not popping)

  MOV     R7, #0                @ count = 0;
.LwhFill:
  CMP     R7, R5                @ while (count < length)
  BHS     .LeWhFill:            @ {
  STR     R6, [R4, R7, LSL #2]  @  word[address + count * 4] = value;
  ADD     R7, #1                @  count = count + 1;
  B       .LwhFill:             @ }
.LeWhFill:
  POP     {R4-R7,pc}            @ restore registers
```

Example – fill with Stack Parameters

Imagine we want to fill memory starting at the address in R5 with the value in R8 and filling the number of words in R9:

```
PUSH    {R5}            ; Push address parameter on stack
PUSH    {R9}            ; Push length parameter on stack
PUSH    {R8}            ; Push value parameter on stack

BL      fill            ; Call fillmem subroutine

ADD     SP, SP, #12     ; Efficiently pop parameters off stack
```


The order of the parameters is important! If we want to control the order of the parameters on the stack, we can't push in one go!

Example – fill with Stack Parameters

before pushing parameters

R13 (SP): 0x20001018

R13 (SP)	address	memory
		● ● ●
	0x2000101C	????????
0x20001018 →	0x20001018	????????
	0x20001014	????????
	0x20001010	????????
	0x2000100C	????????
	0x20001008	????????
	0x20001004	????????
	0x20001000	????????
	0x20000FFC	????????
		● ● ●

32 bits = 1 word

before calling subroutine

R13 (SP): 0x2000100C

R13 (SP)	address	memory
		● ● ●
	0x2000101C	????????
	0x20001018	????????
	0x20001014	address
	0x20001010	length
0x2000100C →	0x2000100C	value
	0x20001008	????????
	0x20001004	????????
	0x20001000	????????
	0x20000FFC	????????
		● ● ●

32 bits = 1 word

Example – fill with Stack Parameters

Why not push the three parameters onto the stack using a single PUSH instruction?

Important that **calling program** restores the system stack to its original state

- Pop off the three parameters

- Quickly and simply done by adding 12 (3 * 4 word-size values) to SP

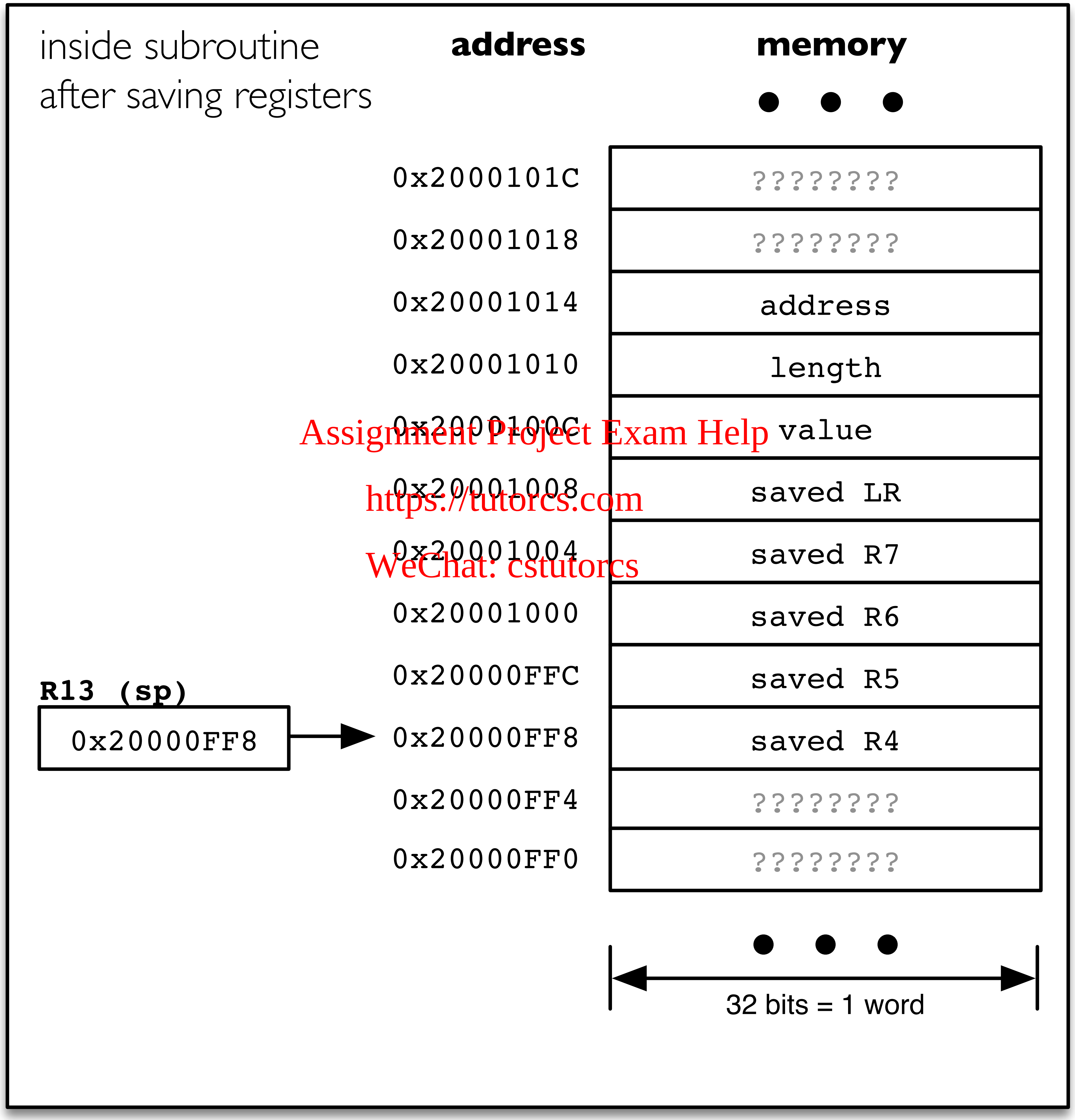
Subroutine doesn’t pop parameters off the stack (why?)

- Accesses them in-place, using offsets relative to the stack pointer

Subroutine saves some registers to the stack

- compensate by adding additional offset (+20) to parameter offsets

Example – fill with Stack Parameters



Example – fill with Stack Parameters

What happens the fill example if we change the list of registers that we save?
(Or worse, manipulate the stack during the execution of the subroutine)

```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@ Parameters
@       [sp+0]: value – value to store in each word (1st Top Of Stack)
@       [sp+4]: length – number of words to be filled (2nd Top Of Stack)
@       [sp+8]: address – address of first word to be filled (3rd Top Of Stack)
fill
  PUSH    {R4-R7,lr}              @ save registers

  LDR     R4, [SP, #8+20]         @ load address parameter (not popping)
  LDR     R5, [SP, #4+20]         @ load length parameter (not popping)
  LDR     R6, [SP, #0+20]         @ load value parameter (not popping)

.LwhFill:
  CMP     R5, #0                  @ while (count > 0)
  BEQ     .LeWhFill:              @ {
  SUB     R5, R5, #1              @  count = count - 1;
  STR     R6, [R4, R7, LSL #2]    @  word[address + count * 4] = value;
  B       .LwhFill:               @ }
.LeWhFill:
  POP     {R4-R6,pc}              @ restore registers
```


Offsets to parameters on the stack may change at design time or at runtime

What happens the fill example if we change the list of registers that we save?
(Or worse, manipulate the stack during the execution of the subroutine)

```
@ fill subroutine
@ Fills a contiguous sequence of words in memory with the same value
@ Parameters
@       [sp+0]: value - value to store in each word (1st Top Of Stack)
@       [sp+4]: length - number of words to be filled (2nd Top Of Stack)
@       [sp+8]: address - address of first word to be filled (3rd Top Of Stack)
fill
  PUSH    {R4-R6,lr}              @ save registers

  LDR     R4, [SP, #8+16]         @ load address parameter (not popping)
  LDR     R5, [SP, #4+16]         @ load length parameter (not popping)
  LDR     R6, [SP, #0+16]         @ load value parameter (not popping)

.LwhFill:
  CMP     R5, #0                  @ while (count > 0)
  BEQ     .LeWhFill:              @ {
  SUB     R5, R5, #1              @  count = count - 1;
  STR     R6, [R4, R7, LSL #2]    @  word[address + count * 4] = value;
  B       .LwhFill:               @ }
.LeWhFill:
  POP     {R4-R6,pc}              @ restore registers
```


Offsets to parameters on the stack may change at design time or at runtime

Example – fill with Stack Parameters

Workaround – at start of subroutine

Save contents of a “scratch” register (e.g. R12) and LR

Copy SP + 8 to “scratch” register

Continue to push data onto the stack as required

Access parameters relative to “scratch” register

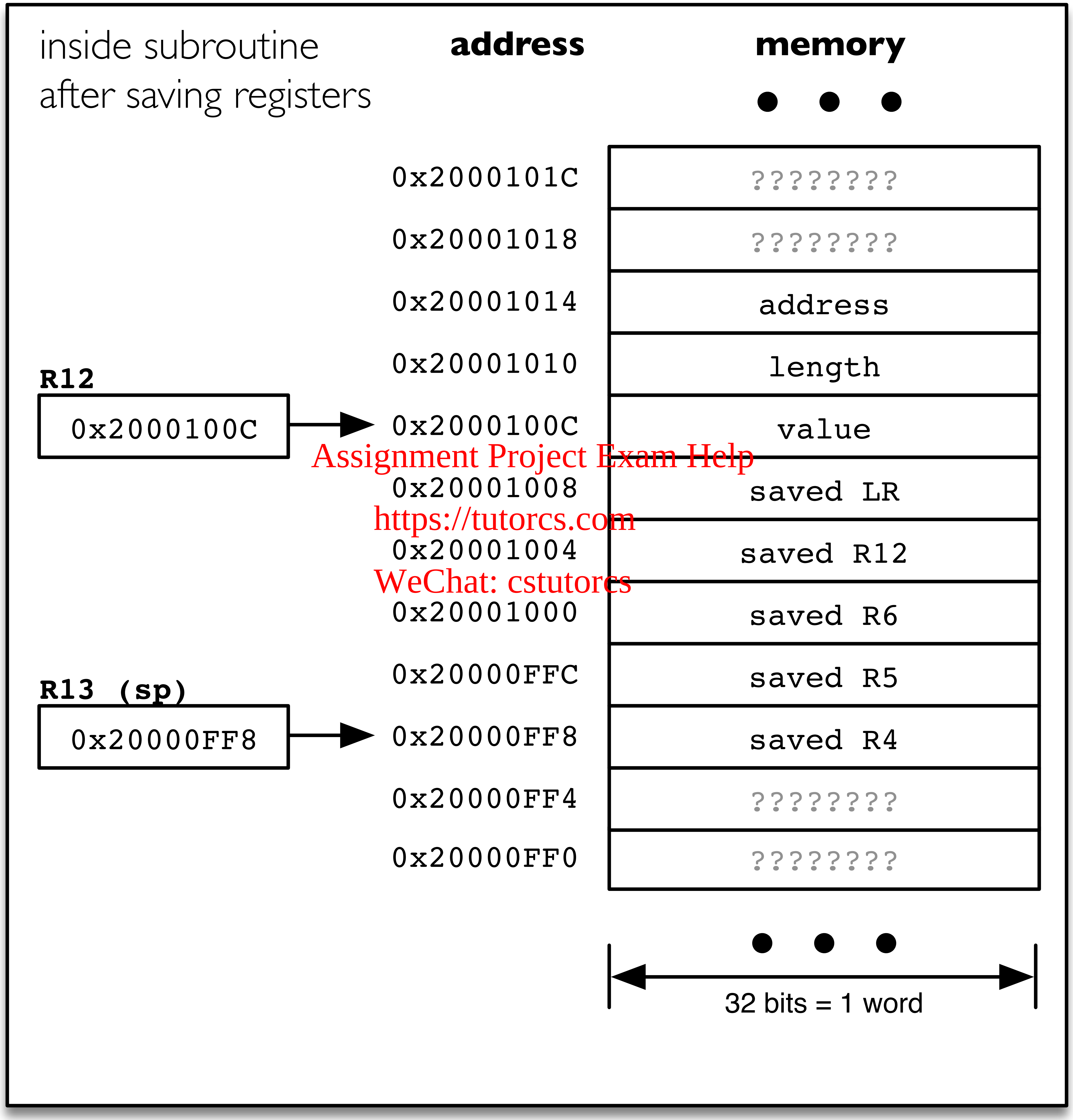

```
fill
  PUSH    {R12, LR}              @ save R12, LR
  ADD     r12, SP, #8            @ scratch = SP + 8
  PUSH    {R4-R6}                @ save registers

  LDR     R4, [r12, #8]          @ load address parameter
  LDR     R5, [r12, #4]          @ load length parameter
  LDR     R6, [r12, #0]          @ load value parameter

  <remainder of subroutine as before>

  POP     {R4-R6}                @ restore registers
  POP     {R12, PC}              @ restore R12, PC
```

Example – fill with Stack Parameters



Subroutine Parameter Guidelines – Recap

Use R0–R3 for parameters and return values

- Avoid using R0–R3 for local variables
- No need to save/restore on system stack

Use R4–R12 for local variables

- Save and restore on system stack

Always save link register LR at start of subroutine

Restore link register LR to PC to return from subroutine

When passing parameters on the stack, use a register (e.g. R12) as a pointer to the parameter block