

Overall Grade Calculation:

70% exam paper

30% coursework composed of the following

Lab 1 - 10% (i.e. 3% of CS1022 final grade)

Lab 2 - 10%

Lab 3 - 10%

Lab 4 - 10%

Assignment - 60%

40% overall required to pass



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

04 – Subroutines

CS1022 – Introduction to Computing II

Dr Adam Taylor / adam.taylor@tcd.ie
School of Computer Science and
Statistics

Programs can be decomposed into blocks of instructions that each perform some specific task

compute x^y

find the length of a NULL-terminated string

convert a string from UPPER CASE to lower case

play a sound

Methods in the Java world!

Functions or **Procedures** elsewhere

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

Example – UPPER CASE

5

```
address = address of first character of string1
char = Memory.byte[address]
while (char != NULL) {
    if (char ≥ 'a' AND char ≤ 'z') {
        char = char AND 0xFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
    char = Memory.byte[address]
}
```

```
address = address of first character of string2
char = Memory.byte[address]
while (char != NULL) {
    if (char ≥ 'a' AND char ≤ 'z') {
        char = char AND 0xFFFFFDF
        Memory.byte[address] = char
    }
    address = address + 1
    char = Memory.byte[address]
}
```




Example – UPPER CASE

6

```
address = address of first character of string1  
uprcase(address1)
```

```
address = address of first character of string2  
uprcase(address2)
```

```
// UPPER CASE  
uprcase (address)  
{  
    char = Memory.byte[address]  
    while (char != NULL) {  
        if (char ≥ 'a' AND char ≤ 'z') {  
            char = char AND 0xFFFFFFFFDF  
            Memory.byte[address] = char  
        }  
        address = address + 1  
        char = Memory.byte[address]  
    }  
}
```



Invoke
uprcase(...)
twice



Define
uprcase(...)

When designing a program, it should be decomposed into logical blocks (subroutines), each of which performs a specific function

Each subroutine can be programmed, tested and debugged independently of other subroutines

Subroutines ...

- facilitate good program design

- facilitate code reuse

- can be invoked (“executed”, “called”) many times

- can invoke other subroutines (and themselves!)

- correspond to procedures/functions/methods in high-level languages

Call and Return mechanism

Invoking (calling) a subroutine requires a deviation from the default sequential execution of instructions

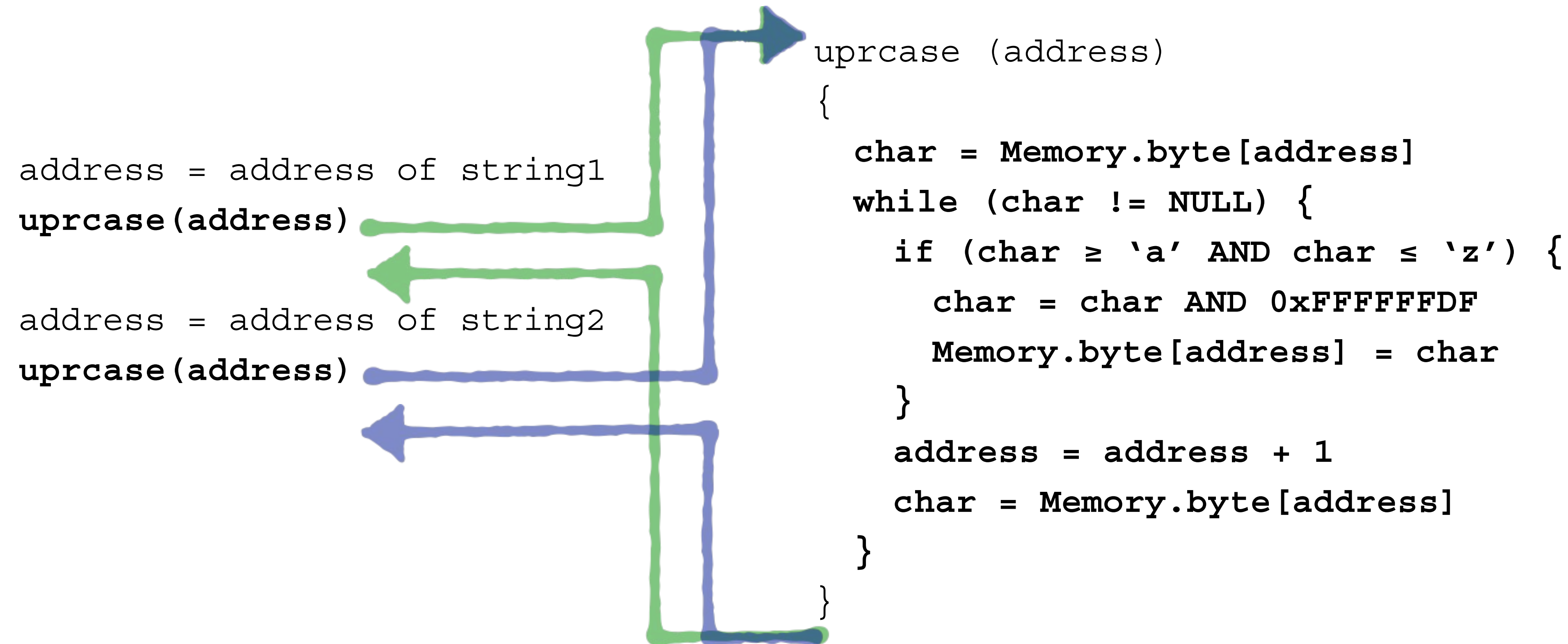
Flow control again, like selection and iteration

Branch to the subroutine by modifying the Program Counter (PC) using a PC Modifying Instruction

e.g. B (unconditional branch, “Branch always”)

When a program branches to a subroutine, the processor begins execution of the instructions that make up the subroutine

When the subroutine has completed its task, the processor must be able to branch back (return) to the instruction immediately following the branch instruction that originally invoked it



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

Returning from a subroutine: must have remembered the address that we originally branched from (return address)

Calling the subroutine (the long way!)

Save address of instruction immediately following branch instruction (return address) in a register (e.g. R14)

Branch (B) to the subroutine

```
... ;  
MOV R14, PC ; save return address  
B sub1 ; branch to the subroutine (labelled sub1)  
    ??? ???, ???, ??? ; some random instruction after the branch  
... ;
```

Returning from the subroutine

Copy (MOV) the return address from R14 into the Program Counter

```
... ;  
??? ???, ???, ???; last subroutine instruction  
MOV PC, R14 ; return to the calling program
```

Saving the return address before branching to a subroutine is a common operation

```
... .. ;  
MOV R14, PC ; save return address  
B sub1 ; branch to the subroutine (labelled sub1)  
    ??? ???, ???, ???; some random instruction after the branch  
... .. ;
```

The Branch and Link instruction (BL) is provided to perform the same task in a single instruction

```
... .. ;  
BL sub1 ; branch to the sub1, saving return address  
??? ???, ???, ???; some random instruction after the branch
```

BL always saves the return address (PC – 4) in R14 (*see discussion on ARM 3-stage pipeline for an explanation!*)

R14 is also called the link register (LR)

Having called the subroutine using BL, we can return in the same way as before ...

```
...    ;  
??? ???, ???, ???; last subroutine instruction  
MOV pc, lr ; return to the calling program
```

Note use of lr as a synonym for R14

Above return method works but assembler will produce a warning and recommend the following ...

```
...    ;  
??? ???, ???, ???; last subroutine instruction  
BX lr    ; return to the calling program
```

Branch and eXchange (BX) loads the contents of the link register (lr) into the program counter

```
LDR r1, =str1 ; load address of first string  
BL uprcase ; invoke uprcase subroutine
```

BL saves the address of the following LDR instruction before branching to the start of uprcase ...

```
LDR r1, =str2 ; load address of second string  
BL uprcase ; invoke uprcase subroutine
```

... after executing uprcase, execution resumed with this LDR instruction

```
stop B stop
```

```
;  
; Define strings to test program  
;
```

```
AREA TestData, DATA, READWRITE  
str1 DCB "motor",0 ; NULL terminated test string  
str2 DCB "zero",0 ; NULL terminated test string
```

Implementing the UPRCASE subroutine

14

```
; uprcase subroutine
; converts the characters in a NULL-terminated string to UPPERCASE
; R1 - start address of NULL-terminated string
```

uprcase

```
wh1 LDRB  r0, [r1], #1 ; while ( (char = Memory.byte[address])
    CMP r0, #0 ;    != 0 )
    BEQ endwh1 ; {
    CMP r0, #'a' ;    if (char >= 'a'
    BCC endif1 ;    AND
    CMP r0, #'z' ;    char <= 'z')
    BHI endif1 ;    {
    BIC r0, #0x00000020;    char = char AND NOT 0x00000020
    STRB r0, [r1, #-1];    Memory.byte[addres - 1] = char
endif1    ;    }
    B wh1 ; }
endwh1    ;
    BX lr ; return
```

uprcase label refers to
1st instruction in
subroutine - used to
invoke the subroutine

Branch and eXchange
instruction causes
execution to continue
after the instruction that
invoked the subroutine

Take a few moments to consider what's wrong with this program ...

```
; Top level program
    BL  sub1  ; call sub1

stop  B stop

; sub1 subroutine
sub1
    ADD R0, R1, R2 ; do something
    BL  sub2  ; call sub2
    ADD R3, R4, R5 ; do something
    BX  lr   ; return from sub1

; sub2 subroutine
sub2
    BX  lr   ; return from sub2
```

Solution

Save the contents of the link register before a subroutine invokes a further nested subroutine

Restore the contents of the link register when the nested subroutine returns

Where should we save the contents of the link register?

Revised sub1 from the previous example ...

```
; sub1 subroutine
```

```
Sub1
```

```
ADD R0, R1, R2 ; do something
```

```
STMFD sp!, {lr} ; save link register
```

```
BL sub2 ; call sub2
```

```
LDMFD sp!, {lr} ; restore link register
```

```
ADD R3, R4, R5 ; do something
```

```
BX lr ; return from sub1
```

A more general and efficient solution

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 at the end of every subroutine

```
; subx subroutine
subx
    STMFD sp!, {lr}    ; save link register
    ... ..
    ... ..
    LDMFD sp!, {lr}    ; restore link register
    BXlr    ; return from sub1
```

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

```
... ..
    LDMFD sp!, {pc}    ; restore link register
```

Consider the following program which converts a string to UPPER CASE before making a copy of it in memory

```
LDR r4, =deststr ; ptr1 = address of deststr
LDR r5, =teststr ; ptr2 = address of teststr

BL uprcase ; uprcase(ptr2)

whCopy LDRB r6, [r5], #1 ; while ( (ch = Memory.Byte[ptr1++])
    CMP r6, #0 ; != NULL)
    BEQ eWhCopy ; {
    STRB r6, [r4], #1 ; Memory.Byte[ptr2++] = ch
    B whCopy ; }
eWhCopy
```

... implementation of uprcase subroutine as before

uprcase

```
wh1 LDRB  r4, [r5], #1 ; while ( (char = Memory.byte[address])
    CMP r4, #0 ;    != 0 )
    BEQ endwh1 ; {
    CMP r4, #'a' ;    if (char >= 'a'
    BCC endif1 ;    &&
    CMP r4, #'z' ;    char <= 'z')
    BHI endif1 ;    {
    BIC r4, #0x00000020;    char = char AND NOT 0x00000020
    STRB r4, [r5, #-1];    Memory.byte[address - 1] = char
endif1      ;    }
    B wh1 ; }
endwh1      ;
    BX lr ; return
```

Why won't this program work?

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



hidden?

In the previous example, the calling top level program should not be affected by modifications to r4 and r5 made by the uprcase 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

Save register contents on the system stack

Example: modified uprcase subroutine

```
; UPPER CASE subroutine
```

```
uprcase
```

```
    STMFD sp!, {r4-r5,lr}
```

```
wh1 LDRB  r4, [r5], #1 ; while ( (char = Memory.byte[address])
```

```
    CMP r4, #0 ;    != 0 )
```

```
    BEQ endwh1 ; {
```

```
    CMP r4, #'a' ;    if (char >= 'a'
```

```
    BCC endif1 ;    &&
```

```
    CMP r4, #'z' ;    char <= 'z')
```

```
    BHI endif1 ; {
```

```
    BIC r4, #0x00000020;    char = char AND NOT 0x00000020
```

```
    STRB r4, [r5, #-1];    Memory.byte[address - 1] = char
```

```
endif1      ; }
```

```
    B wh1 ; }
```

```
endwh1      ;
```

```
    LDMFD sp!, {r4-r5,pc}; return
```

save used registers on
the start (plus link
register) at the start of
the subroutine

restore registers before
returning

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

uprcase subroutine had single address parameter

```
address = address of first character of string1  
uprcase(address)
```

```
address = address of first character of string2  
uprcase(address)
```

```
. . .
```

```
uprcase (address)  
{  
    . . .  
}
```

Simplest way to pass parameters to a subroutine is to use well defined registers, e.g. for uprcase, use r1 for address

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

```
fillmem (address, length, value)
{
    count = 0;
    while (count < length)
    {
        Memory.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

```
; fillmem 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
```

```
fillmem
```

```
    STMFD sp!, {r0-r2,r4,lr}
    MOV r4, #0      ; count = 0;
wh1 CMP r4, r1      ; while (count < length)
    BHS endwh1      ; {
    STR r2, [r0, r4, LSL #2] ; Memory.Word[address+(count*4)] = value;
    ADD r4, #1      ; count = count + 1;
    B wh1           ; }
endwh1              ;
    LDMFD sp!, {r0-r2,r4,pc}
```

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!!

AAPCS – ARM Application Procedure Call Standard

http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042d/IHL0042D_aapcs.pdf

Writing subroutines that adhere to this standard allows subroutines to be separately written and assembled

Contract between subroutine callers and callees

Standard specifies:

- how parameters must be passed to subroutines

- which registers must have their contents preserved across subroutine invocations (and which are corruptible)

- special roles for certain registers

- a Full Descending stack pointed to by R13 (sp)

- etc.

register	notes
R0	Parameters to and results from subroutine
R1	
R2	
R3	
R4	<i>corruptible</i>
R5	
R6	
R7	
R8	Variables
R9	
R10	
R11	
R12	Must be preserved
R13	Scratch register (<i>corruptible</i>)
R14	
R15	
	Stack Pointer (SP)
	Link Register (LR)
	Program Counter (PC)

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

Based on these guidelines, we could re-write fillmem (note that I was already adhering to the guidelines for passing parameters!!)

```
; fillmem 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
```

```
fillmem
```

```
    STMFD sp!, {r4,lr} ; save registers used for local variables
```

```
    MOV r4, #0 ; count = 0;
```

```
wh1 CMP r4, r1 ; while (count < length)
```

```
    BHS endwh1 ; {
```

```
    STR r2, [r0, r4, LSL #2] ; Memory.Word[address+(count*4)] = value;
```

```
    ADD r4, #1 ; count = count + 1;
```

```
    B wh1 ; }
```

```
endwh1 ;
```

```
    LDMFD sp!, {r4,pc} ; restore registers
```

Recall the new fillmem interface

```
; fillmem 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 fillmem is implemented

Call fillmem, 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  fillmem   ; invoke fillmem
```

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
    STMFD sp!, {r4, lr} ; save registers
    MOV r4, r0 ; copy wordval parameter to local variable
    MOV r0, #0 ; count = 0;
wh1
    CMP r4, #0 ; while (wordval != 0)
    BEQ endwh1 ; {
    MOVS r4, r4, LSR #1 ; wordval = wordval >> 1; (update carry)
    ADC r0, r0, #0 ; count = count + 0 + carry;
    B wh1 ; }
endwh1 ;
    LDMFD sp!, {r4, pc} ; restore registers
```

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 (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

r0 used to pass wordval parameter and return result value from count1s subroutine (an implementation decision – real AAPCS compilers would also do this!)

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 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  
... ..
```

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 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

Often parameters passed to (or values returned from) a subroutine are too large to be stored in registers

e.g. 128-bit integer, ASCII string, image, list of integers

Solution: the calling program ...

stores the parameter in memory

uses a register to pass the address of the parameter to the subroutine

Example

Design and write an ARM Assembly Language subroutine that will add two 128-bit integers

Require 4 words for each operand and 4 words for the result

```
; add128 subroutine
; Adds two 128-bit integers
; Parameters:
;   r0: pResult - pointer to (address of) memory to
;           store result of a + b
;   r1: pVal1 - pointer to (address of) first integer a
;   r2: pVal2 - pointer to (address of) second integer b
```

Begin with a program to test the subroutine

```
start
```

```
    LDR r1, =val1 ; load address of 1st 128bit value
```

```
    LDR r2, =val2 ; load address of 2nd 128bit value
```

```
    LDR r0, =result ; load address for 128bit result
```

```
    BL  add128
```

```
stop  B stop
```

```
AREA TestData, DATA, READWRITE
```

```
val1  DCD 0x57FD30C2,0x387156F3,0xFE4D6750,0x037CB1A0
```

```
val2  DCD 0x02BA862D,0x298B3AD4,0x213CF1D2,0xFD00357C
```

```
result SPACE 16
```

```
; add128 subroutine
; Adds two 128-bit integers
; Parameters r0: pResult - pointer to (address of) memory to store result of a + b
;   r1: pVal1 - pointer to (address of) first integer a
;   r2: pVal2 - pointer to (address of) second integer b
add128
```

```
    STMFD sp!, {r5-r7,lr}; save registers
```

```
LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADDS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;
```

```
LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;
```

```
. . . . .
```

.

```
LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;  
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;  
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)  
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;
```

```
LDR r5, [r1], #4 ; tmp1 = Memory.Word[pVal1]; pVal1 = pVal1 + 4;  
LDR r6, [r2], #4 ; tmp2 = Memory.Word[pVal2]; pVal2 = pVal2 + 4;  
ADCS r7, r5, r6 ; tmpResult = tmp1 + tmp2; (update C flag)  
STR r7, [r0], #4 ; Memory.Word[pResult]; pResult = pResult + 4;
```

```
LDMFD sp!, {r5-r7,pc}; restore registers
```


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

- Commonly used by high-level languages

- Similar to passing parameters by reference but using the stack pointer instead of a dedicated pointer

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

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

Pseudo-code reminder

```
fillmem (address, length, value)
{
    count = 0;
    while (count < length)
    {
        Memory.Word[address] = value;
        address = address + 4;
        count = count + 1;
    }
}
```

First, write a program to test the subroutine

```
LDR r5, =tstarea; Load address to be filled
```

```
LDR r9, =32 ; Load number of words to be filled
```

```
LDR r8, =0xC0C0C0C0 ; Load value to fill
```

```
STR r5, [sp, #-4]!; Push address parameter on stack
```

```
STR r9, [sp, #-4]!; Push length parameter on stack
```

```
STR r8, [sp, #-4]!; Push value parameter on stack
```

```
BLfillmem ; Call fillmem subroutine
```

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

```
stopB stop
```

```
AREATest, DATA, READWRITE
```

```
tstarea SPACE 256
```

Example – fillmem with Stack Parameters

43

```
; fillmem subroutine
; Fills a contiguous sequence of words in memory with the same value
; Parameters [sp+0]: value - value to store in each word
; [sp+4]: length - number of words to be filled
; [sp+8]: address - address of first word to be filled
fillmem
    STMFD sp!, {r4-r7,lr} ; save registers

    LDR r4, [sp, #8+20] ; load address parameter
    LDR r5, [sp, #4+20] ; load length parameter
    LDR r6, [sp, #0+20] ; load value parameter

    MOV r7, #0 ; count = 0;
wh1 CMP r7, r5 ; while (count < length)
    BHS endwh1 ; {
    STR r6, [r4, r7, LSL #2] ; Memory.Word[address + count * 4] = value;
    ADD r7, #1 ; count = count + 1;
    B wh1 ; }
endwh1

    LDMFD sp!, {r4-r7,pc} ; restore registers
```

Could push the parameters onto the stack more efficiently with a single STMFD instruction

but we want to be explicit about the order of the operands

value, length, address could become address, length, value

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

Pop off the three parameters

Quickly and simply done by adding 12 to SP

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

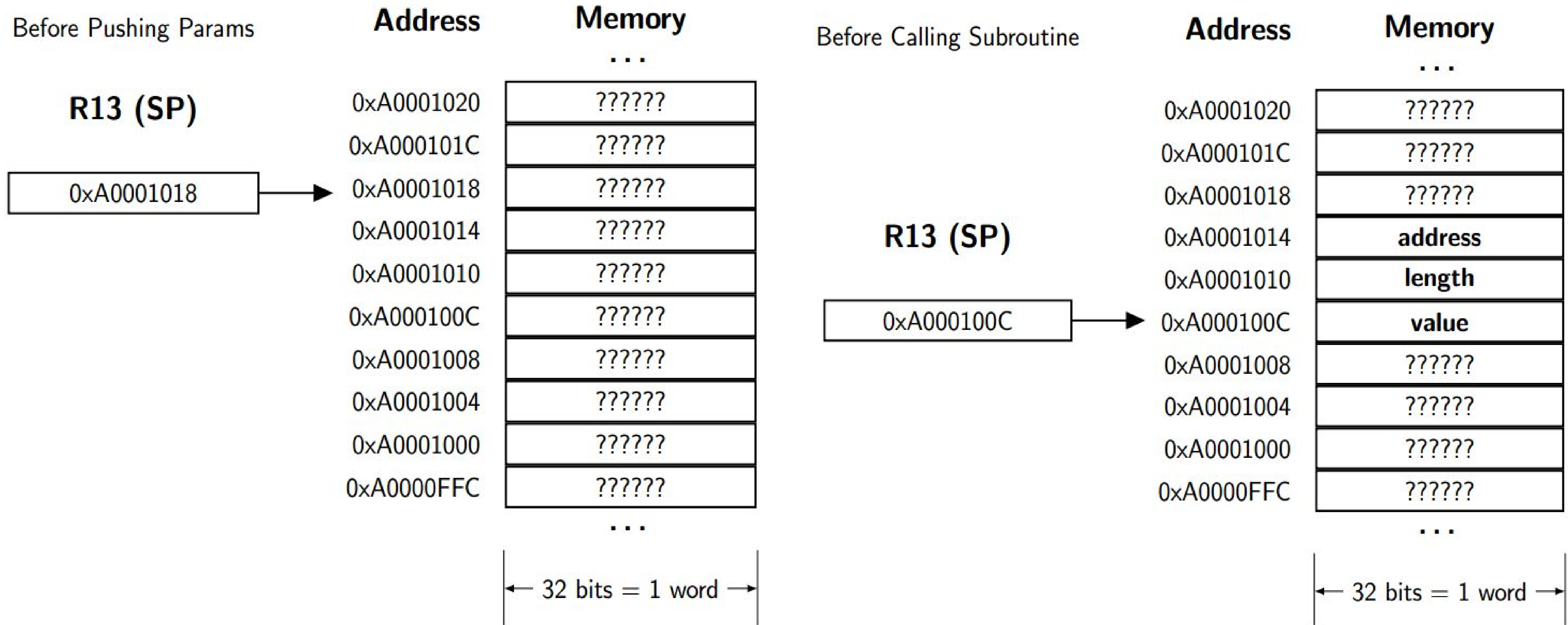
Accesses them in-place, using offsets relative to the stack pointer.

Subroutine saves some registers to the stack

compensate by adding addition offset (+20) to parameter offsets

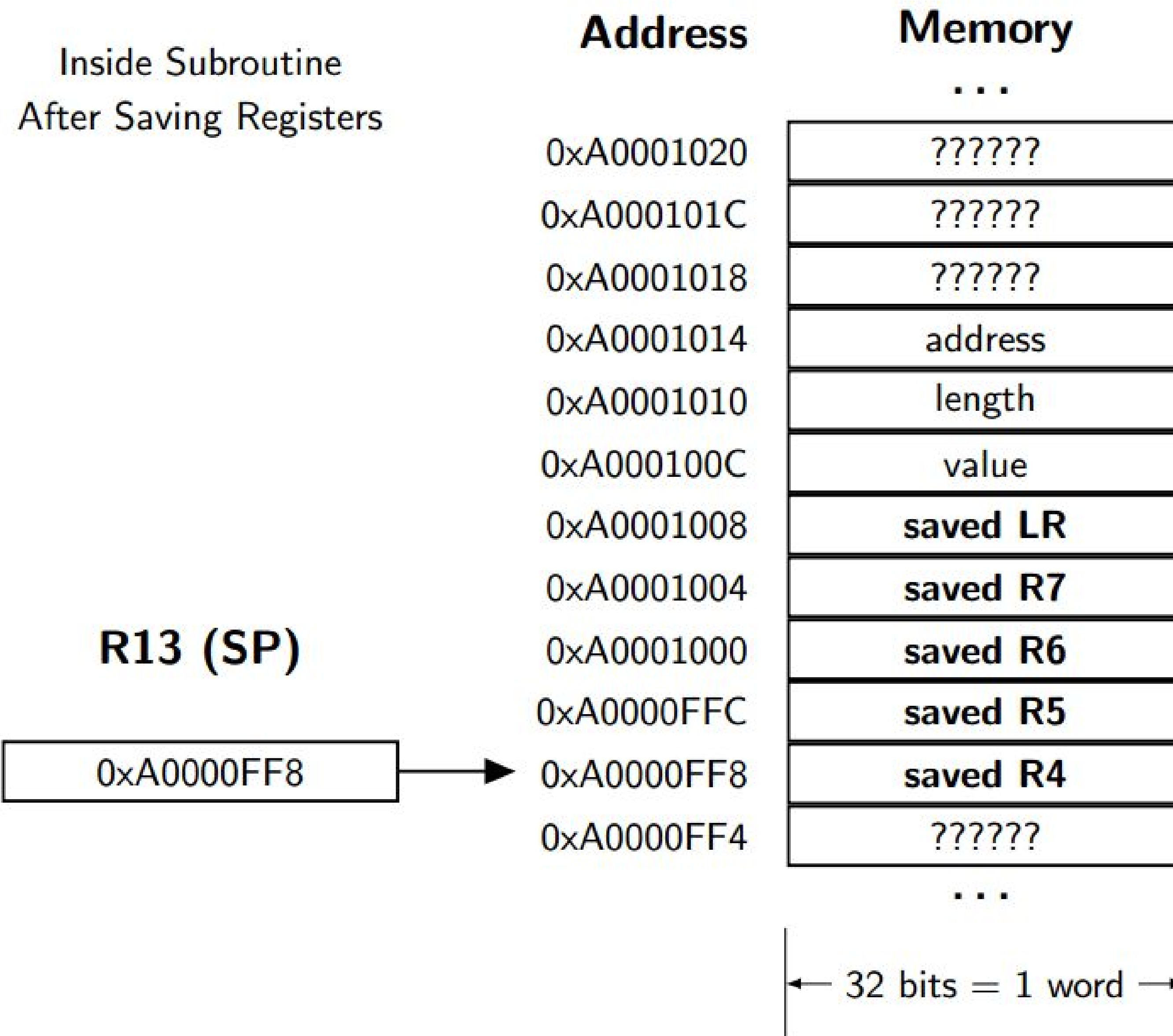
Example – fillmem with Stack Parameters

45



Example – fillmem with Stack Parameters

46



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

```
fillmem
```

```
    STMFD sp!, {r4-r7,lr} ; save registers
```

```
    LDR r4, [sp, #8+20] ; load address parameter
```

```
    LDR r5, [sp, #4+20] ; load length parameter
```

```
    LDR r6, [sp, #0+20] ; load value parameter
```

```
    MOV r7, #0; count = 0;
```

```
wh1 CMP r7, r5; while (count < length)
```

```
    BHS endwh1; {
```

```
    STR r6, [r4, r7, LSL #2] ; Memory.Word[address + count * 4] = value;
```

```
    ADD r7, #1; count = count + 1;
```

```
    B wh1 ; }
```

```
endwh1
```

```
    LDMFD sp!, {r4-r7,pc} ; restore registers
```

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

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

`fillmem`

```
STMFD sp!, {r12, lr}    ; save r12, lr
```

```
ADD r12, sp, #8         ; scratch = sp + 8
```

```
STMFD sp!, {r4-r7}      ; save registers
```

```
LDR r4, [r12, #8]       ; load address parameter
```

```
LDR r5, [r12, #4]       ; load length parameter
```

```
LDR r6, [r12, #0]       ; load value parameter
```

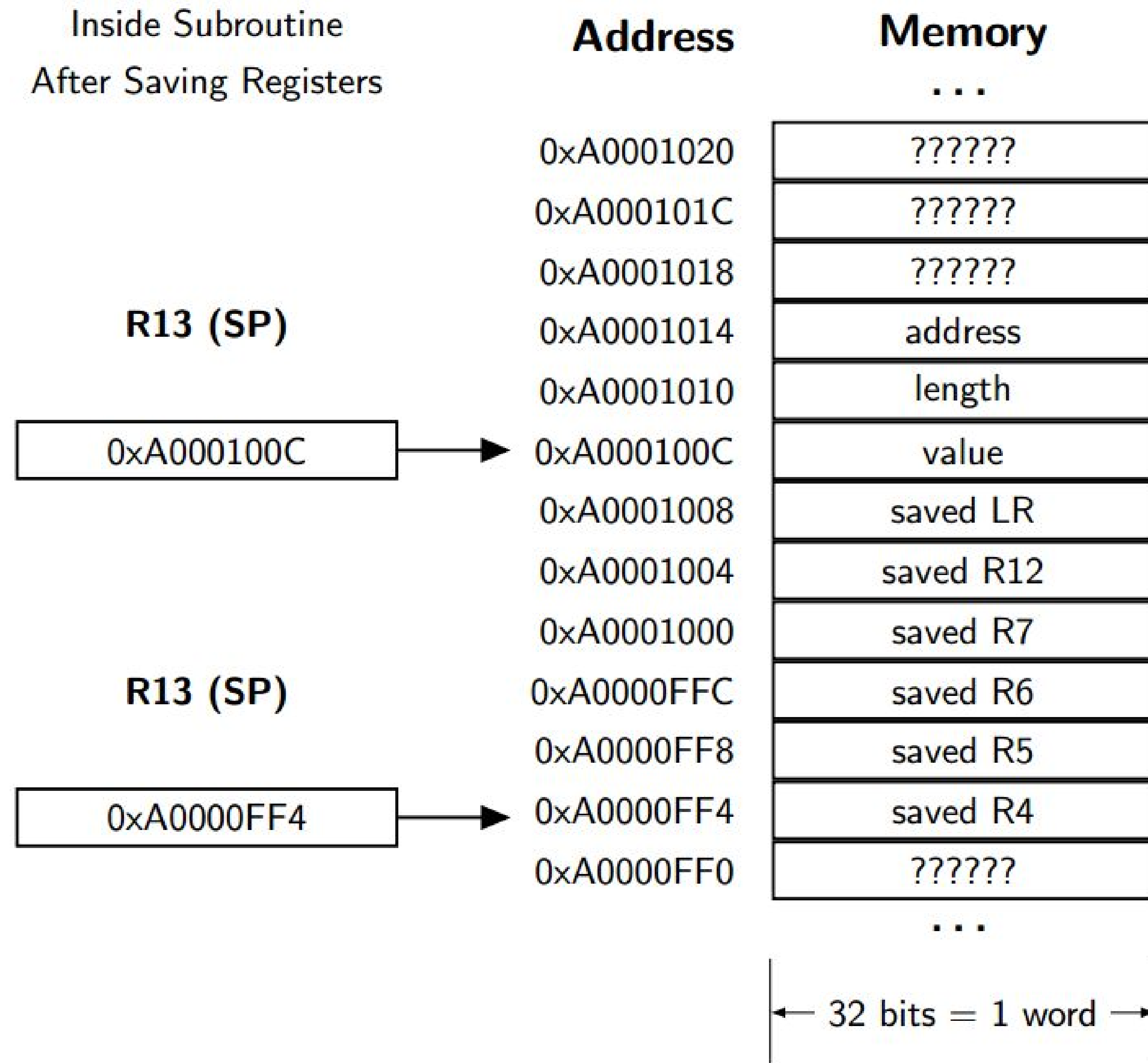
<remainder of subroutine as before>

```
LDMFD sp!, {r4-r7}      ; restore registers
```

```
LDMFD sp!, {r12, pc}     ; restore r12, pc
```

Example – fillmem with Stack Parameters

49



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

Subroutines can invoke themselves – recursion

Example: Design, write and test a subroutine to compute $x(n)$

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

```
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)
    }
}
```

```
; power subroutine
; Compute x^n
; Parameters r0: x
;   r1: n
; Return value r0: x^n
power
    STMFD sp!, {r4-r5,lr}; save registers
    MOV r4, r0 ; x
    MOV R5, r1 ; n

    CMP r5, #0 ; if (n == 0)
    BNE else11 ; {
    MOV r0, #1 ; result = 1;
    B endif1 ; }
else11 CMP r5, #1 ; else if (n == 1)
    BNE else12 ; {
    MOV r0, r4 ; result = x;
    B endif1 ; }
```



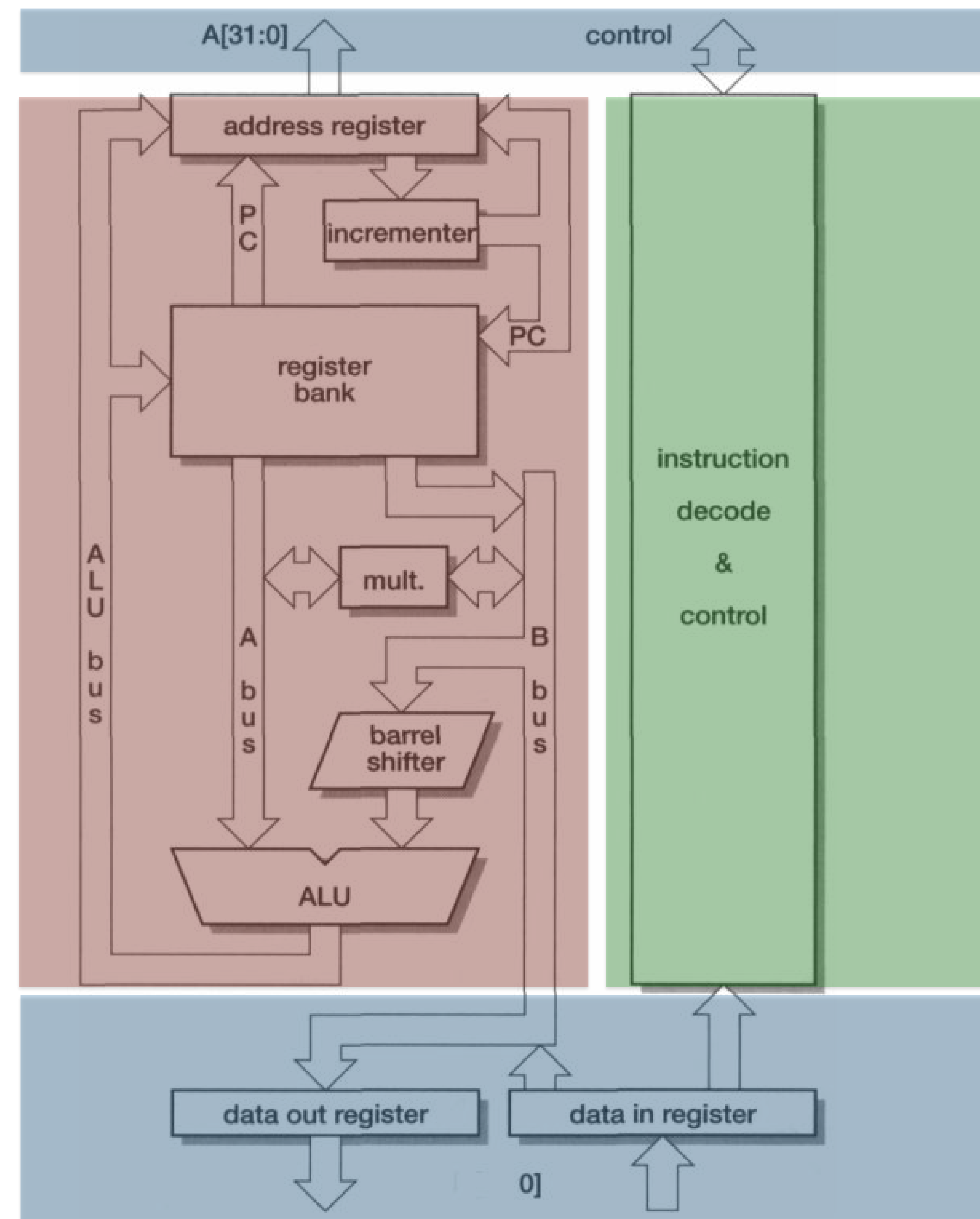
```
else12  TST r5, #1 ; else if (n & 1 == 0)
        BNE else13 ; {
        MOV r0, r4 ; x_param = x;
        MUL r0, r4, r0 ; x_param = x_param * x;
        MOV r1, r5, LSR #1 ; n_param = n / 2;
        BL power ; result = power (x_param, n_param);
        B endif1 ; }
        ; else {
else13  MOV r0, r4 ; x_param = x;
        MUL r0, r4, r0 ; x_param = x_param * x;
        MOV r1, r5, LSR #1 ; n_param = n / 2;
        BL power ; result = power (x_param, n_param);
        MUL r0, r4, r0 ; result = x * result;
endif1  ; }

LDMFD sp!, {r4-r5,pc}; restore registers
```

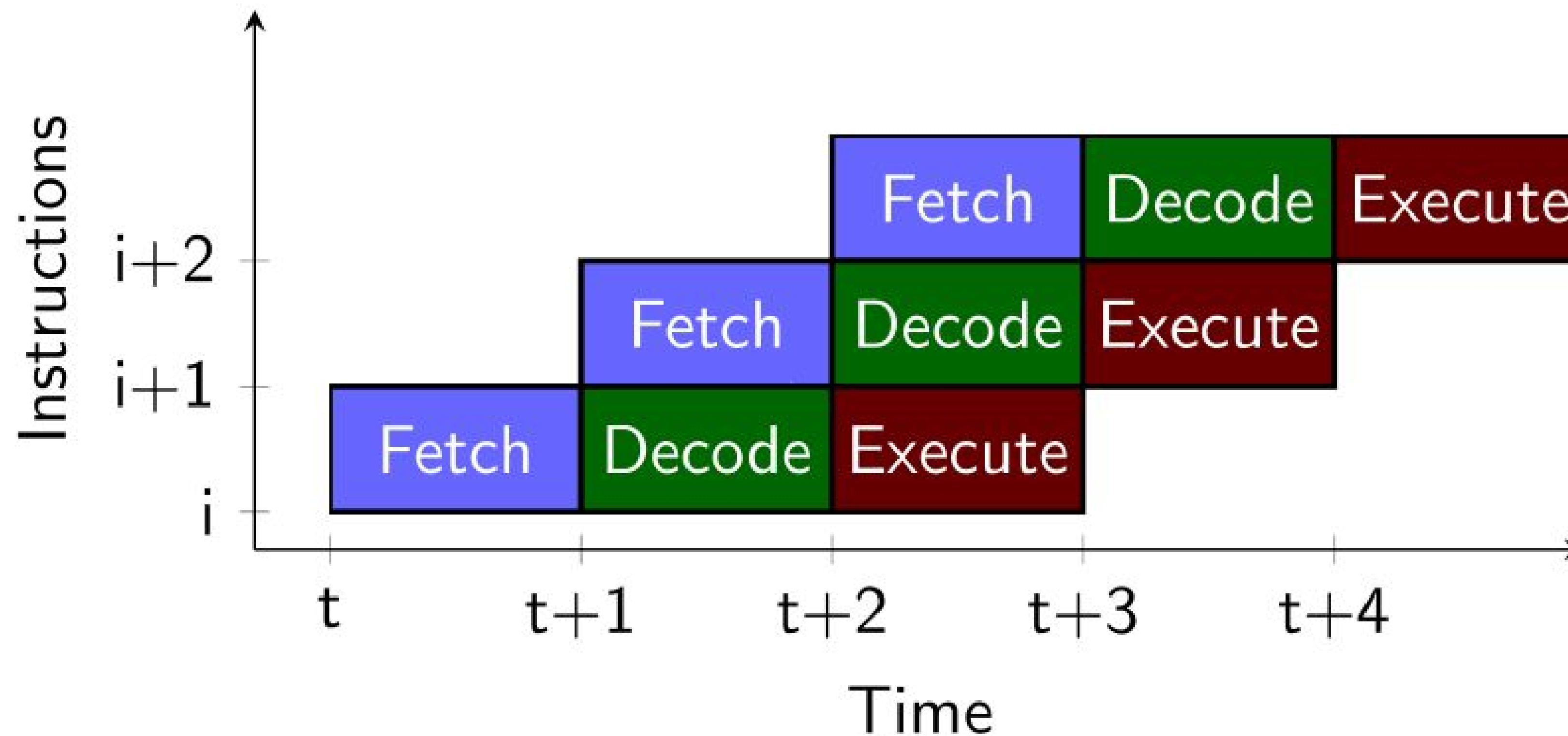
ADVANCED

The ARM 3-State Instruction Execution Pipeline

Steve Furber, “*ARM System-on-Chip Architecture*”, 2nd edition, Addison Wesley Professional, 2000.



“normal” instructions (e.g. ADD, MOV)



B **label** ; Branch unconditionally to label

... ; ...

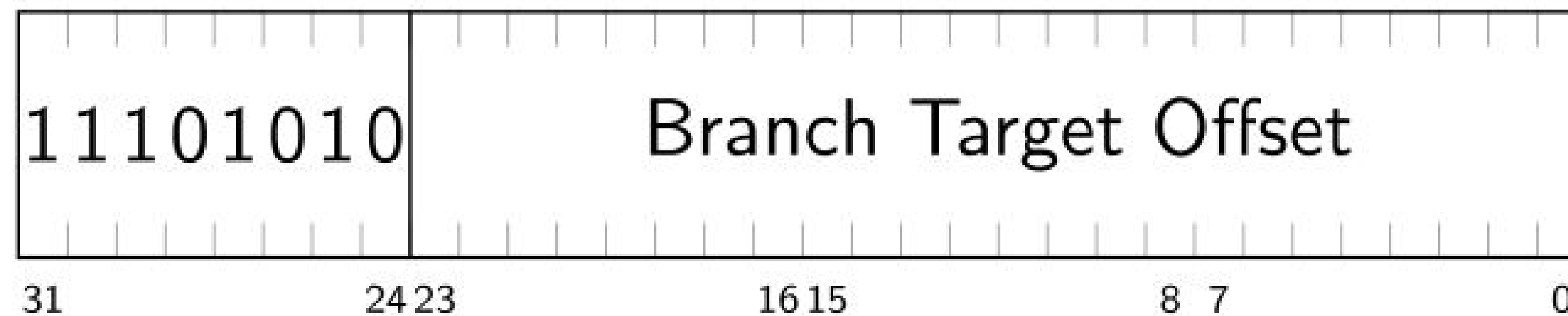
... ; more instructions

... ; ...

label some instruction ; more instructions

... ; ...

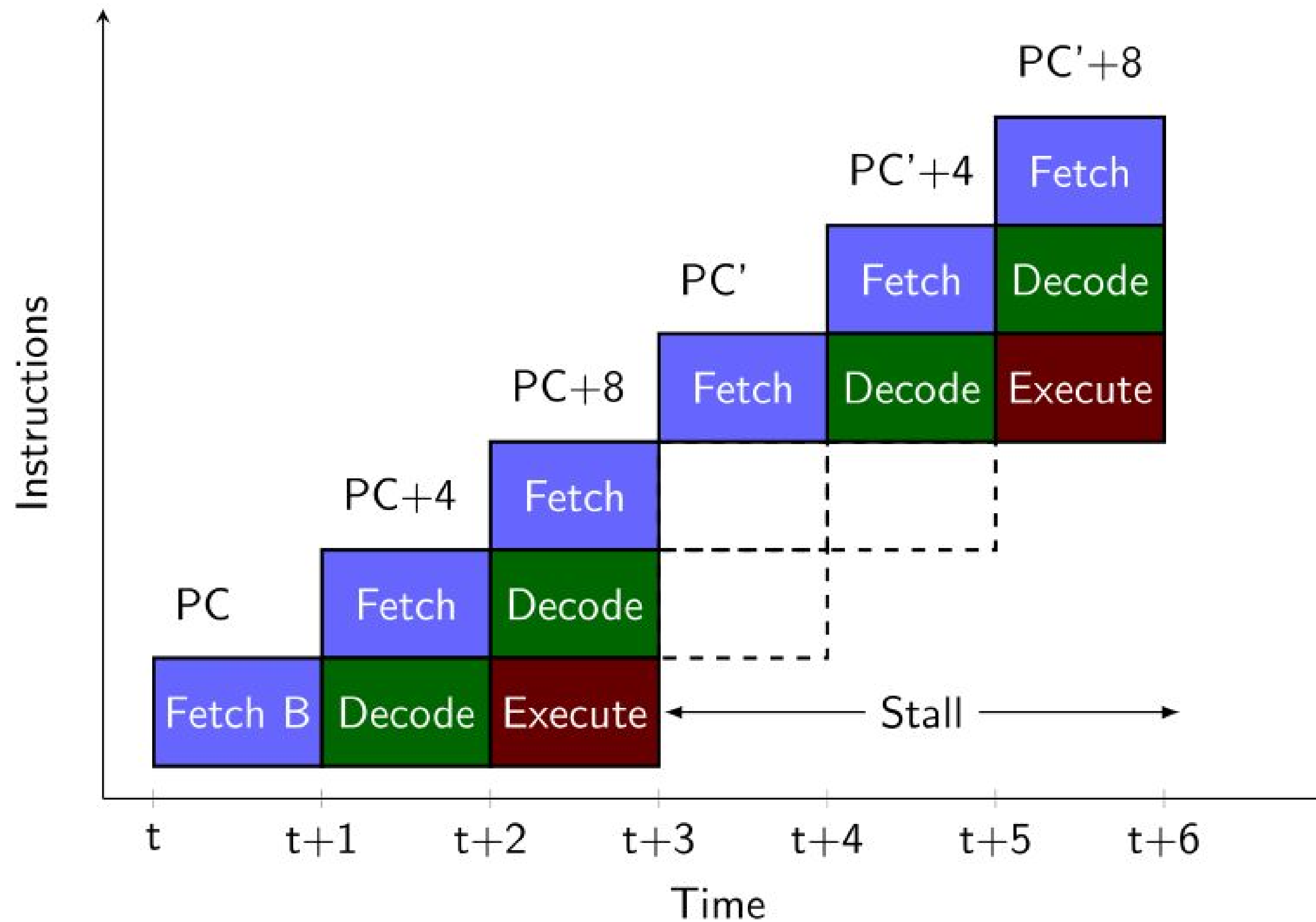
Machine code for Branch Instruction (no Link!)



Branch target offset is added to current Program Counter value

Next fetch in fetch → decode → execute cycle will be from new Program Counter address

branch instructions (e.g. B)



```
while BEQ endwh ; while (y ≠ 0) {  
    MUL r0, r1, r0 ; result = result × x  
    SUBS r2, r2, #1 ; y = y - 1  
    B while ; }  
endwh
```

Labels are used to specify branch targets in assembly language programs

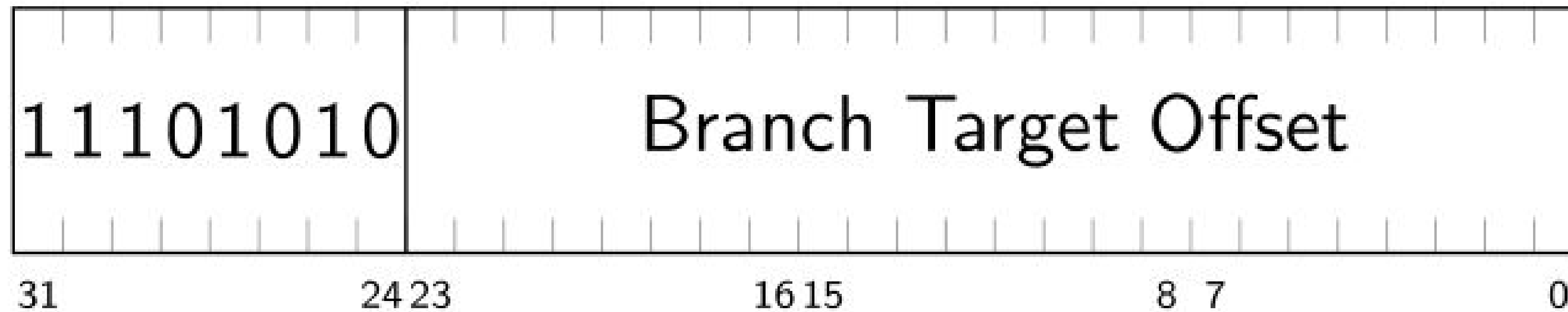
Assembler calculates necessary branch target offset:

$$\text{branch target offset} = ((\text{label address} - \text{branch inst. address}) - 8) / 4$$

Branch target offset could be negative (branch backwards)

All ARM instructions are 4 bytes (32-bits) long and must be stored on 4-byte boundaries

Branch target offset can be divided by 4 before being encoded in branch instruction



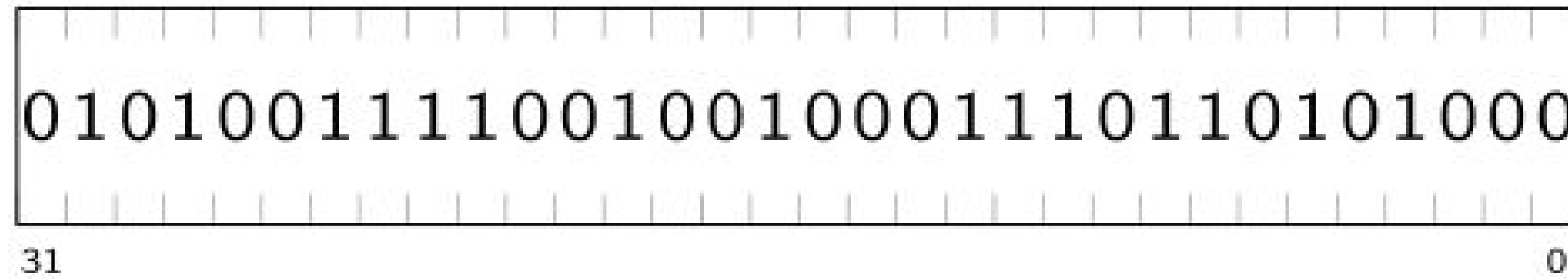
$$PC \leftarrow PC + (\text{branch target offset} \times 4)$$

Next fetch in fetch → decode → execute cycle will fetch the instruction at the new PC

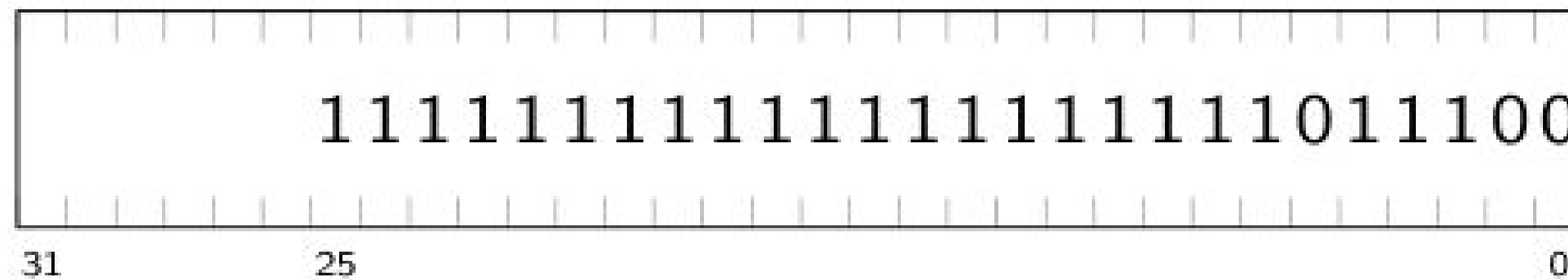
26-bit branch target offset may be negative

Must sign-extend a less-than-32-bit value before using it to perform 32-bit arithmetic

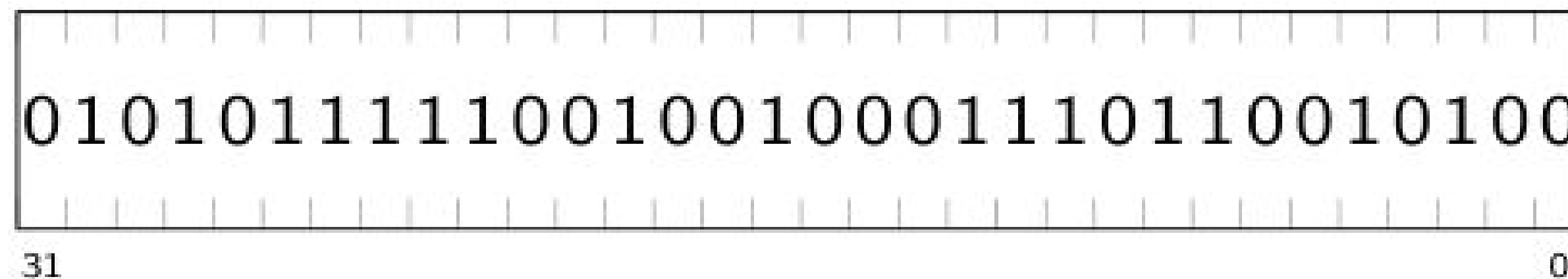
i.e. 26-bit branch target offset must be sign-extended to form a 32-bit value before adding it to the 32-bit Program Counter



32 bit Program Counter

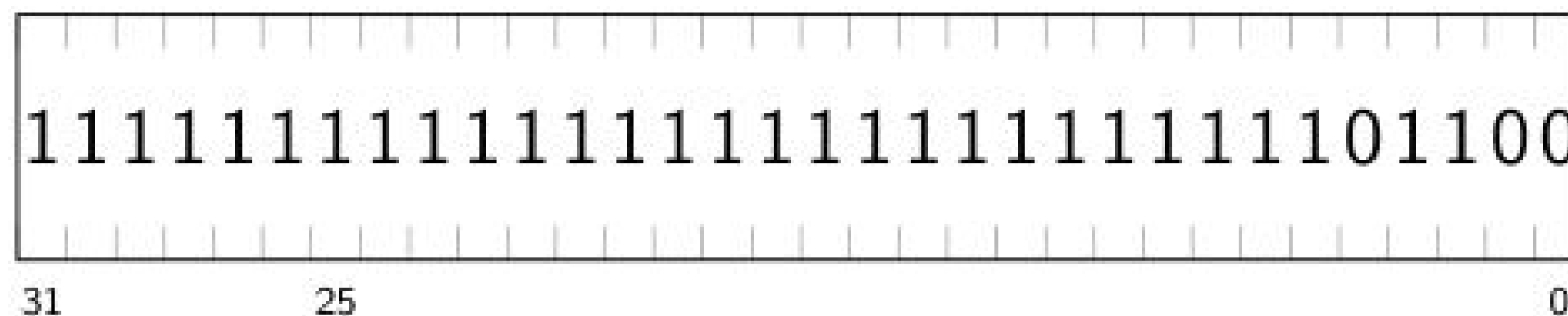


26 bit Signed Target Offset



32 bit Signed Target Offset (Incorrect)

Must sign extend the 26-bit offset by copying the value of bit 25 into bits 26 to 31 (2's Complement system)



32 bit Signed Target Offset

Calling the subroutine (the long way!)

Save address of instruction immediately following branch instruction (return address) in a register (e.g. R14)

Branch (B) to the subroutine

```
... .. ;  
MOV R14, PC ; save return address  
B sub1 ; branch to the subroutine (labelled sub1)  
    ??? ???, ???, ??? ; some random instruction after the branch  
... .. ;
```

Returning from the subroutine

Copy (MOV) the return address from R14 into the Program Counter

```
... .. ;  
??? ???, ???, ???; last subroutine instruction  
MOV PC, R14 ; return to the calling program
```

Saving the return address before branching to a subroutine is a common operation

```
... .. ;  
MOV R14, PC ; save return address  
B sub1 ; branch to the subroutine (labelled sub1)  
    ??? ???, ???, ???; some random instruction after the branch  
... .. ;
```

The Branch and Link instruction (BL) is provided to perform the same task in a single instruction

```
... .. ;  
BL sub1 ; branch to the sub1, saving return address  
??? ???, ???, ???; some random instruction after the branch  
... .. ;
```

BL always saves the return address (PC – 4) in R14

R14 is also called the link register (LR)

branch and link (BL) instruction

