

COMSM1302

Overview of Computer Architecture

Lecture 15

Performance Considerations in Assembly Programming

In the previous lecture



- Load and store instructions
 - Single register data transfer (LDR / STR).
 - Block data transfer (LDM/STM).
- Pre- and post- addressing modes
- Direct and sequential access of array elements
- Copy data blocks with Block data transfer instructions
- Stack

In this lecture



Example of
using stack



Program
performance
measures



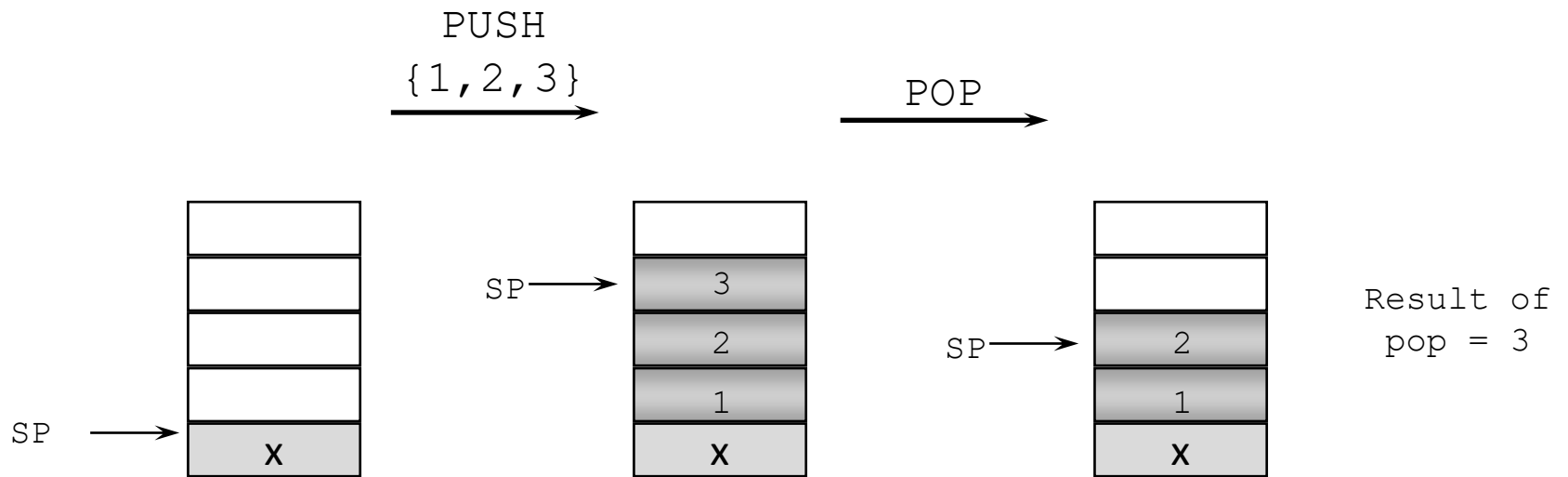
Performance
considerations in ARM
assembly
programming

- At the end of this lecture:
 - Understand how the stack help us in tracking function calls.
 - Learn about programs performance measures.
 - Learn how to write efficient assembly programs.

Assembly programming performance



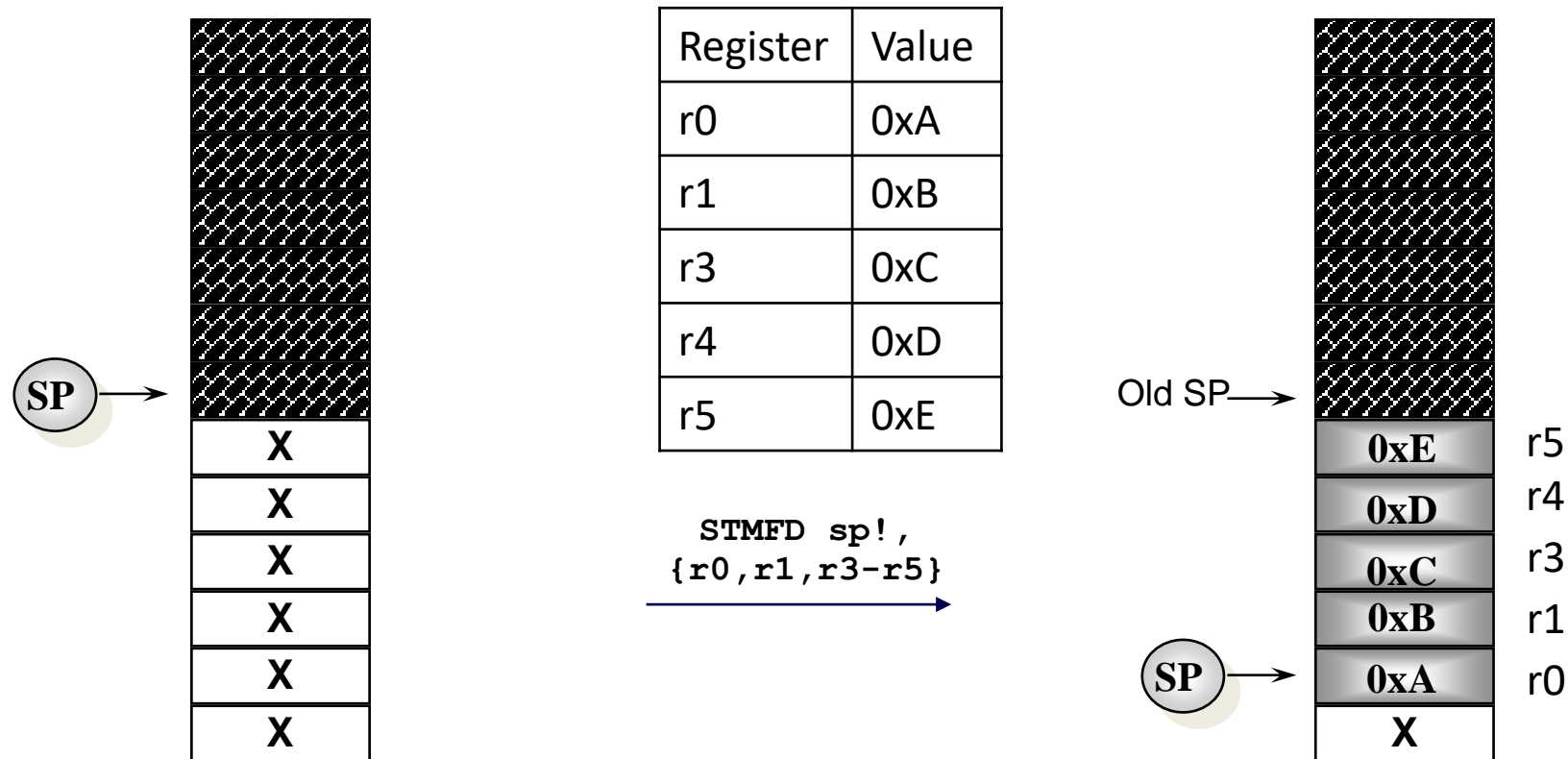
Stack



Stacks in ARM

- The stack type to be used is given by the postfix to the instruction:
 - STMFD / LDMFD : Full Descending stack
 - STMFA / LDMFA : Full Ascending stack.
 - STMED / LDMED : Empty Descending stack
 - STMEA / LDMEA : Empty Ascending stack

🔥 Full Descending stack – 1/3



🔥 Full Descending stack – 2/3

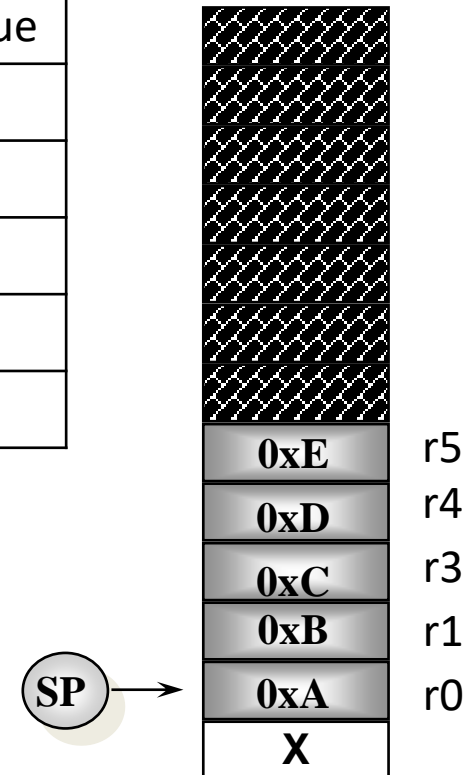


Register	Value
r0	0xA
r1	0xB
r3	0xC
r4	0xD
r5	0xE

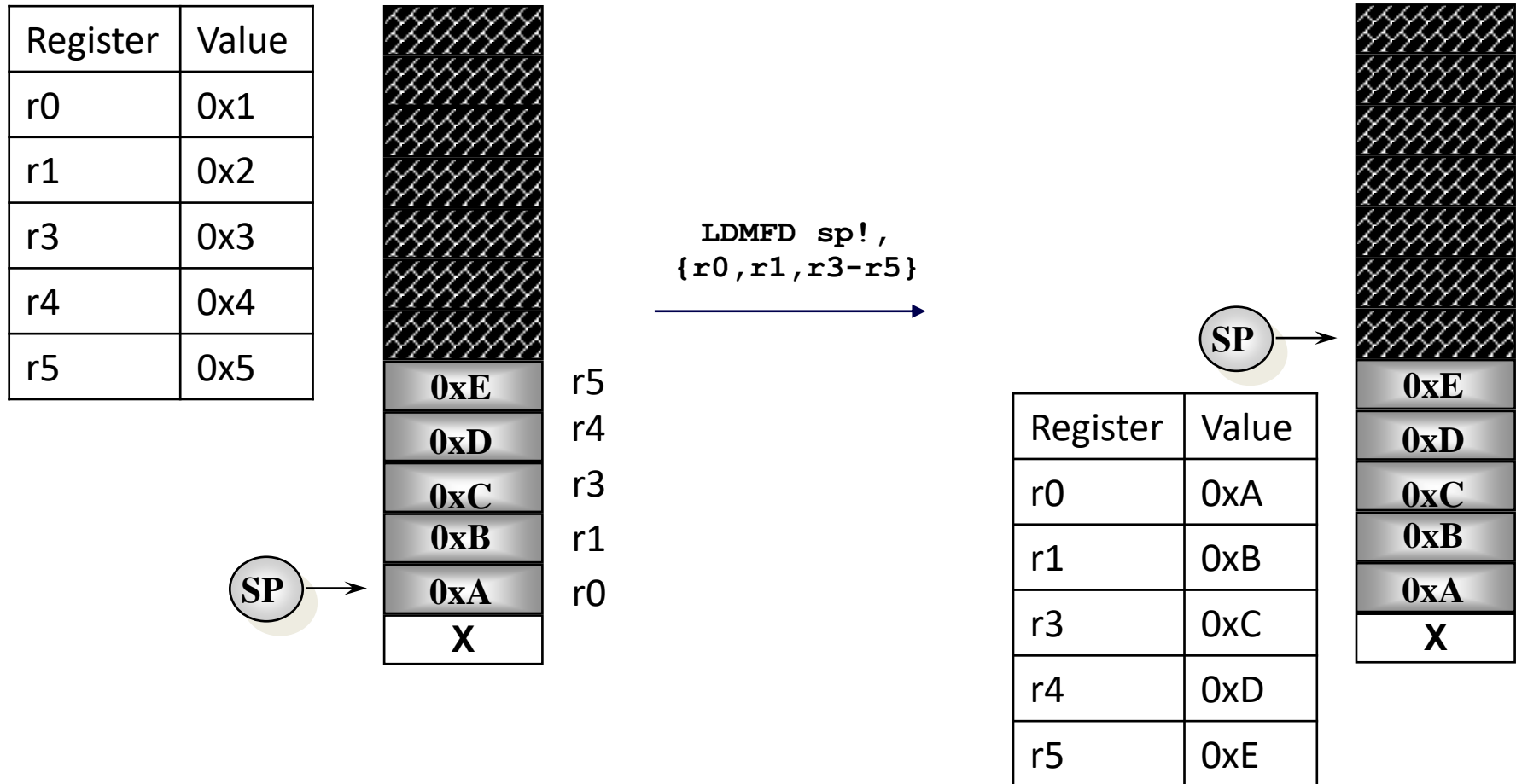
Some assembly
code



Register	Value
r0	0x1
r1	0x2
r3	0x3
r4	0x4
r5	0x5



Full Descending stack – 3/3



ARM Stack Implementations

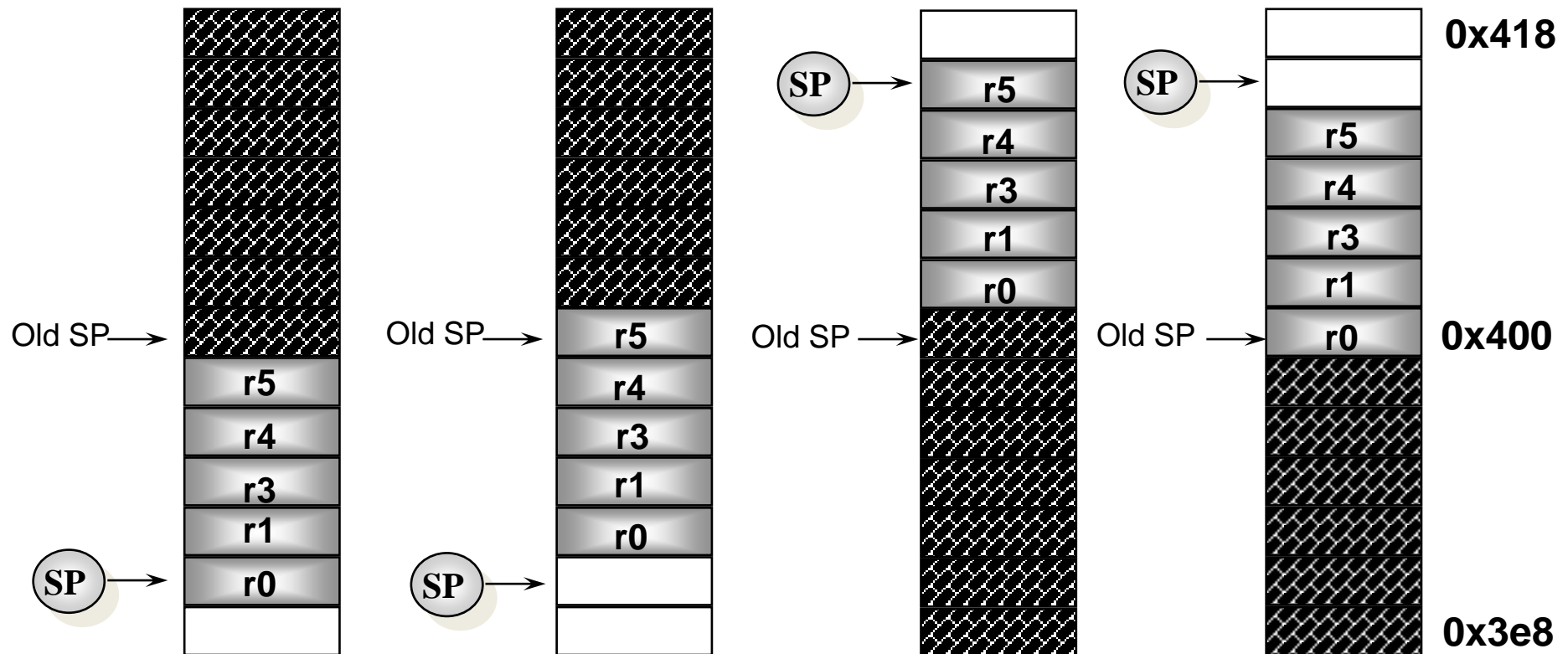


STMFD sp!,
{r0,r1,r3-r5}

STMED sp!,
{r0,r1,r3-r5}

STMFA sp!,
{r0,r1,r3-r5}

STMEA sp!,
{r0,r1,r3-r5}



Example : factorial

- Factorial of a non-negative integer, is multiplication of all integers smaller than or equal to n . For example factorial of 4 is $4*3*2*1$ which is 24.
- $n! = n*(n-1)*(n-2)*(n-3)*...*1$
- $4! = 4*(4-1)*(4-2)*(4-3) = 24$

🔥 Factorial – recursive solution

- $n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$
- $(n-1)! = (n-1) * ((n-1)-1) * ((n-1) - 2) * \dots * 1$
- $(n-1)! = (n-1) * (n-2) * (n-3) * \dots * 1$
- $n! = n * (n-1)!$
- $n! = 1$ if $n = 0$ or $n = 1$

Factorial – recursive solution – C code

```
// function to find factorial of given number
unsigned int factorial(unsigned int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

🔥 Factorial – recursive solution – assembly code – 1/6



```
Ldr r0, =#3  
bl _factorial  
b _end
```

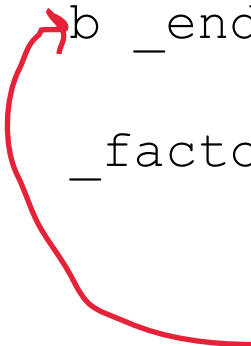
```
_factorial:
```

```
    cmp r0, #0
```

```
    moveq r0, #1
```

```
    moveq pc, lr
```

if (n == 0)
 return 1;



Factorial – recursive solution – assembly code – 2/6

```
Ldr r0, =#3
bl _factorial
b _end
```

```
_factorial:
```

```
    cmp r0, #0
    moveq r0, #1          if (n == 0)
                           return 1;
```

```
    moveq pc, lr
```

```
    mov r1, r0           Save n and calculate n-1 then call factorial for n-1
```

```
    sub r0, r0, #1
```

```
    stmfd sp!, {r1, lr}
```

```
    bl _factorial
```

🔥 Factorial – recursive solution – assembly code – 3/6



```
Ldr r0, =#3
bl _factorial
b _end                                <- 0x2

_factorial:
    cmp r0, #0
    moveq r0, #1
    moveq pc, lr
    mov r1, r0
    sub r0, r0, #1
    stmfd sp!, {r1, lr}
    bl _factorial

                                <- 0xA
```

0x2
3

🔥 Factorial – recursive solution – assembly code – 4/6



```
Ldr r0, =#3
bl _factorial
b _end                <- 0x2

_factorial:
    cmp r0, #0
    moveq r0, #1
    moveq pc, lr
    mov r1, r0
    sub r0, r0, #1
    stmfd sp!, {r1, lr}
    bl _factorial
                                <- 0xA
```

0x2
3
0xA
2

Factorial – recursive solution – assembly code – 5/6



```
Ldr r0, =#3
bl _factorial
b _end                <- 0x2

_factorial:
    cmp r0, #0
    moveq r0, #1
    moveq pc, lr
    mov r1, r0
    sub r0, r0, #1
    stmfd sp!, {r1, lr}
    bl _factorial
                                <- 0xA
```

0x2
3
0xA
2
0xA
1

Factorial – recursive solution – assembly code – 6/6



```
Ldr r0, =#3
bl _factorial
b _end                                <- 0x2

_factorial:
    cmp r0, #0
    moveq r0, #1
    moveq pc, lr
    mov r1, r0
    sub r0, r0, #1
    stmfd sp!, {r1, lr}
    bl _factorial
    lmdfd sp!, {r1, lr}    <- 0xA
    mul r0, r1, r0
    mov pc, lr
```

0x2
3
0xA
2
0xA
1

Assembly programming performance



🔥 Program performance measures

- Program Execution¹ Time
 - Worst-case execution time (WCET):
 - Best-case execution time (BCET)
 - Average-case execution time (ACET)
- Program² Size: number of instructions in ARM
- Program³ Energy Consumption

Assembly programming performance



ARM assembly programming performance issues



- LDM /STM
- Conditional execution
- Using the barrel shifter
- Optimising register usage
- Loop unrolling
- Initiate a register with zero
- Addressing modes
- Multiplication / Division

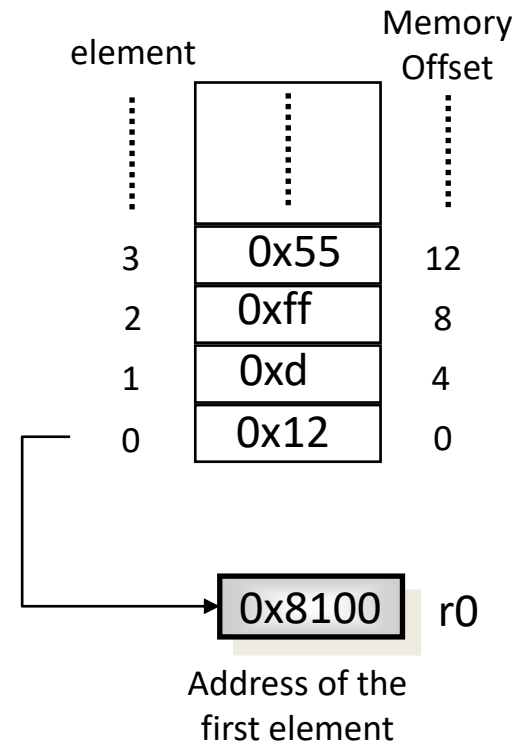
LDM / STM

- Use LDM and STM instead of a sequence of LDR or STR instructions wherever possible.
 - The code is smaller.
 - An instruction fetch cycle can be saved for each eliminated LDR or STR.
 - Can turn non-sequential memory cycles into faster memory sequential cycles.

🔥 LDM / STDM - example

```
MOV r1, #4
MOV r3, #0
LDR r0, =_data
_loop:
LDR r2, [r0], #4
ADD r3, r3, r2
SUBS r1, r1, #1
BNE _loop
STR r3, [r0]
_end: B _end
```

```
MOV r3, #0
LDR r0, =_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```



Conditional execution

- Using conditionally executed instructions can avoid branches around short pieces of code.
 - This reduces the size of code

```
TST r0, #1
BLEQ _even:
_end: B _end
```

```
_even:
LSL r0, r0, #1
MOV r1, r0
MOV pc, lr
```

```
TST r0, #1
LSLEQ r0, r0, #1
MOVEQ r1, r0
_end: B _end
```

Using the barrel shifter

- Combining shift operations with other operations can significantly increase the code density and thus performance.

```
TST  r0, #1
LSLEQ r0, r0, #1
MOVEQ r1, r0
_end: B _end
```

```
TST  r0, #1
MOVEQ r1, r0, LSL #1
_end: B _end
```

Conditional execution and barrel shifter example

```
TST r0,#1
BLEQ _even:
_end: B _end
```

```
_even:
LSL r0,r0,#1
MOV r1,r0
MOV pc, lr
```

```
TST r0,#1
LSLEQ r0,r0,#1
MOVEQ r1,r0
_end: B _end
```

```
TST r0,#1
MOVEQ r1,r0,LSL #1
_end: B _end
```

Optimising register usage

- Register spillage happens when we have more variables than the number of available registers.
 - a value has to be reloaded.
 - an intermediate value saved and then reloaded.

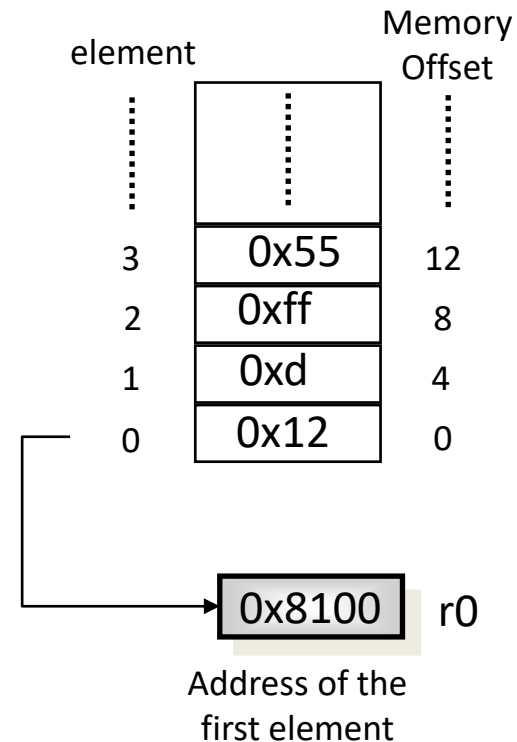
Loop unrolling

- Loop unrolling involves using more than one copy of the inner loop of an algorithm
 - The branch back to the beginning of the loop is executed less frequently.
 - it may be possible to combine some of one iteration with some of the next iteration, and thereby significantly reduce the cost of each iteration.

🔥 Loop unrolling - example

```
MOV r1, #4
MOV r3, #0
LDR r0, =_data
_loop:
LDR r2, [r0], #4
ADD r3, r3, r2
SUBS r1, r1, #1
BNE _loop
STR r3, [r0]
_end: B _end
```

```
MOV r3, #0
LDR r0, =_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```

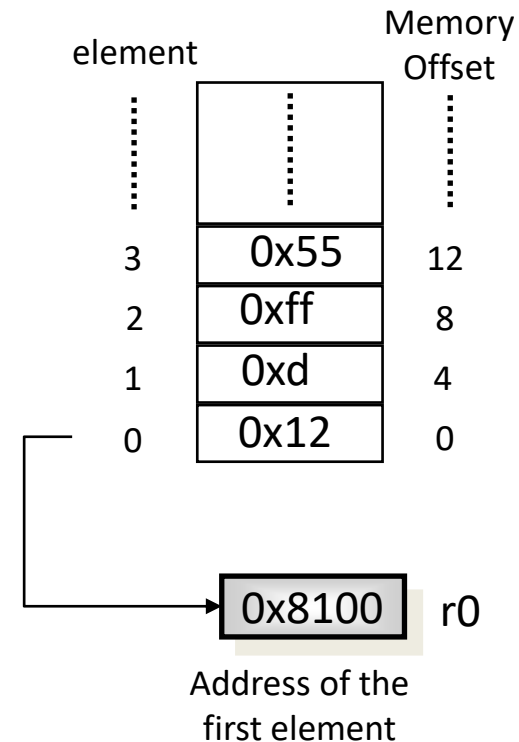


🔥 Initiate a register with zero

- What registers to reset and how?

```
MOV r3, #0
LDR r0, =_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```

```
EOR r3, r3, r3
LDR r0, =_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```

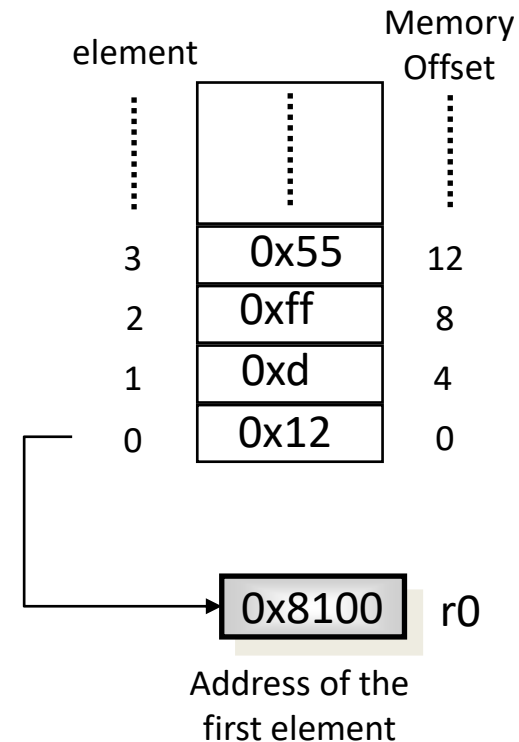


🔥 Initiate a register with zero

- What registers to reset and how?

```
MOV r3, #0
LDR r0,=_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```

```
EOR r3, r3, r3
LDR r0,=_data
LDMIA r0!, {r3-r6}
ADD r3, r3, r4
ADD r3, r3, r5
ADD r3, r3, r6
STR r3, [r0]
_end: B _end
```



Other performance issues

- **Addressing modes**
 - Using LDR or STR pre- or post-indexed with a non-zero offset increments the base register and performs the data transfer
- **Multiplication / Division**
 - Be aware of the time taken by the ARM multiply and division instructions

Summary

- Stack operation
- Recursive solution to compute the factorial of a given number.
- Programs performance measures
- How to write efficient assembly programs.