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

7.1 Bit Manipulation

CSU11021 – Introduction to Computing I

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

A Very Brief Introduction to Boolean Algebra

In Boolean algebra, a variable can have the value TRUE or FALSE

In binary computers, we usually use

1 to represent TRUE and

0 to represent FALSE

There are four Boolean Algebra operations of interest to us

	name	logic symbol	C or Java	ARM
and	conjunction	$\wedge$	&	AND
or	disjunction	$\vee$	|	ORR
not	negation	$\neg$	~	MVN
exclusive or (xor)	exclusive disjunction	$\oplus$	^	EOR

NOT

Unary operator (operates on a single variable)

¬A is the inverse of A

A	¬A
0	1
1	0

"truth table"

AND

Binary Operator

If both A and B are 1, then A ∧ B is 1

A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

OR

Binary Operator

If either A or B is 1, then A ∨ B is 1

Note that if both A and B are 1, then A ∨ B is still 1

A	B	A ∨ B
0	0	0
0	1	1
1	0	1
1	1	1

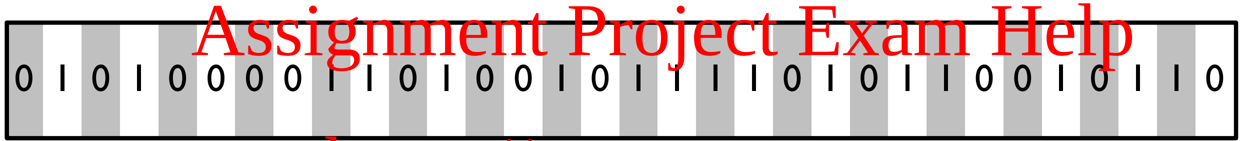
EOR (exclusive OR)

Binary Operator

If either A or B is 1 and they are not both 1, then A $\oplus$ B is 1

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Microprocessors operate on register values containing many bits (e.g. 32-bit values in the case of the ARM Cortex-M4)

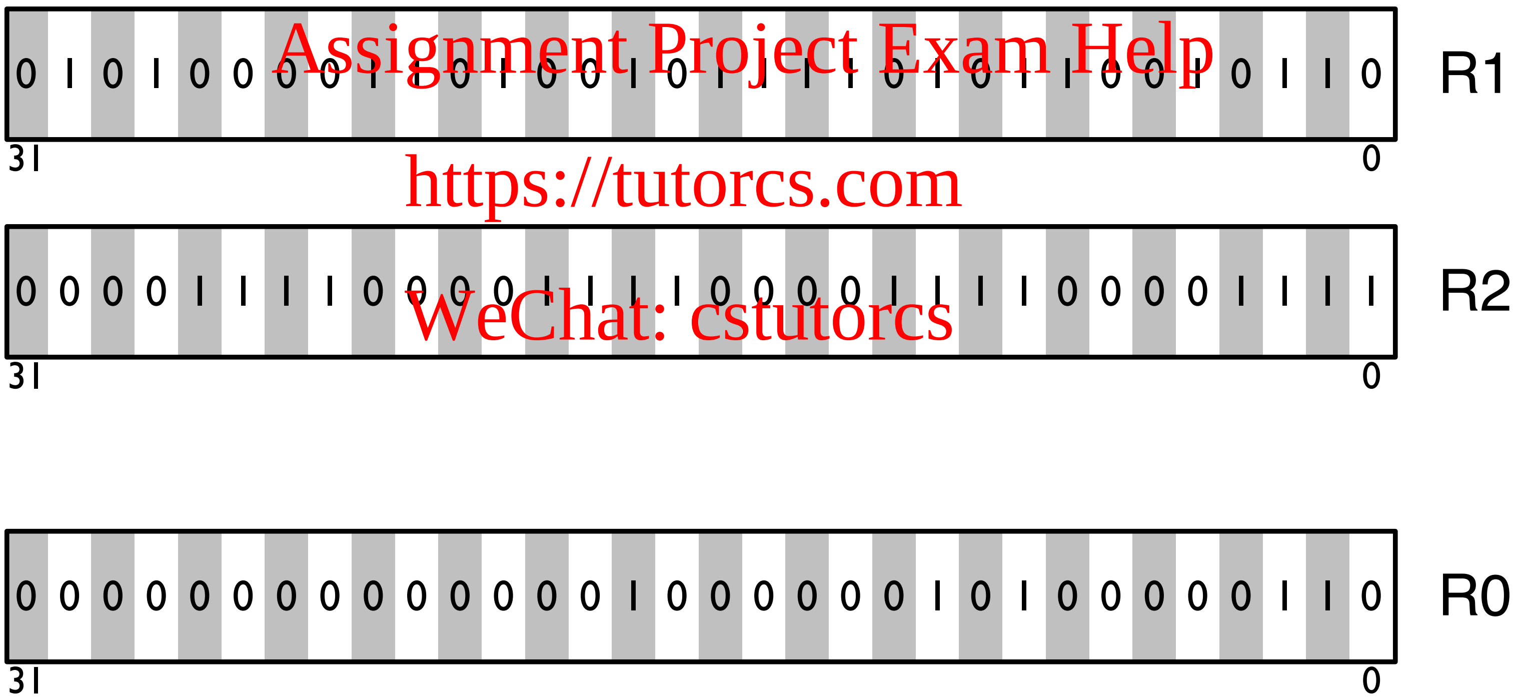


If each bit can represent a single boolean variable, how can we operate on individual boolean variables?

We can't! We operate on *n* (e.g. 32) boolean variables in parallel!

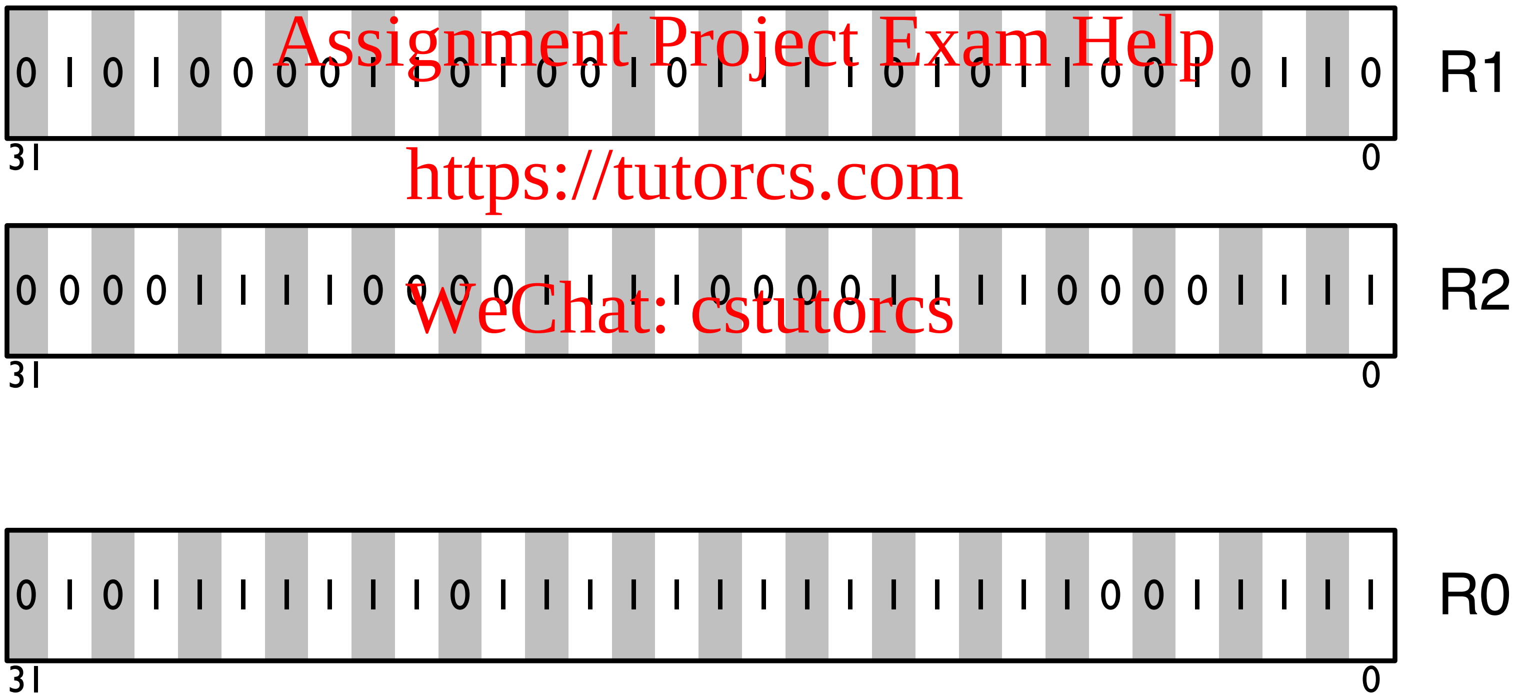
ARM Assembly Language instructions: AND, ORR, MVN, EOR

```
AND R0, R1, R2        @ R0 = R1 & R2
```



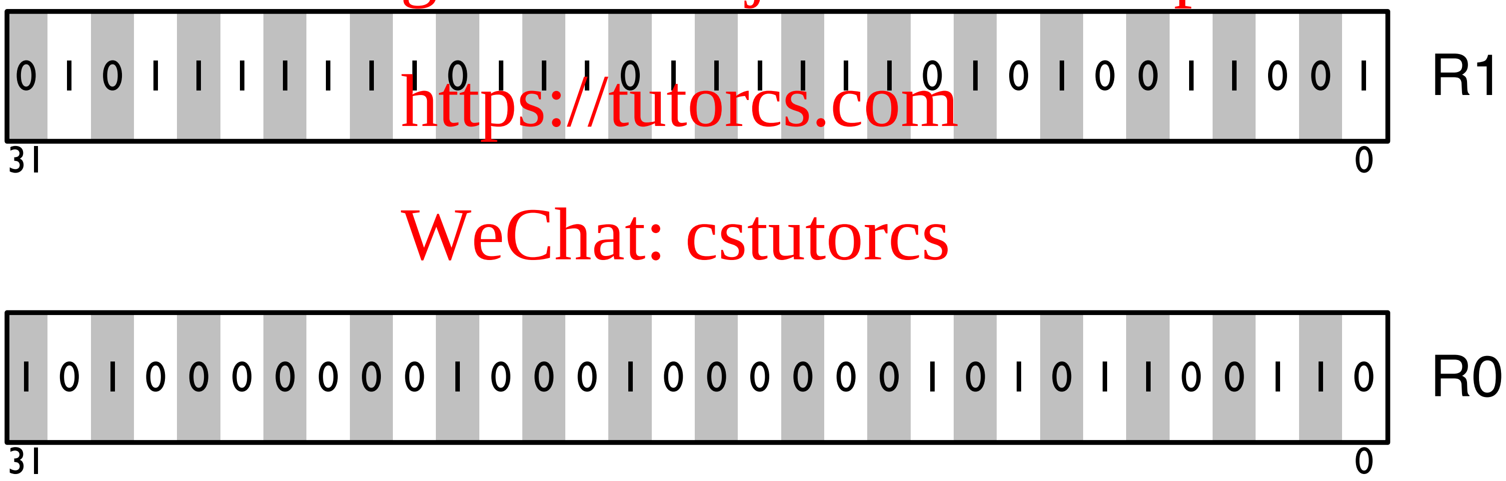
Bitwise Operation Instructions – OR

```
ORR     R0, R1, R2      @ R0 = R1 | R2
```



Bitwise Operation Instructions – NOT

```
MVN     R0, R1  @ R0 = ~R1
```

Bitwise Operation Instructions – EOR

```
EOR     R0, R1, R2      @ R0 = R1 ^ R2 (R1 EOR R2)
```

	31																															0
R1	0	1	0	1	0	0	0	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	1	0	1	0	0	1	0	1	1	0
R2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
R0	0	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	1	0	1	0	0	1	1	0	0	1

Why?

We can use bitwise operations to manipulate the individual bits in a larger value, for example

- Clear (change to zero) the middle two bytes of a word
- Set (change to one) the sixth bit of a word
- Set the four most significant bits of a word to a specific four-bit value

When might you need to do this?

- Implementing network protocols
- Working with floating-point values (more next term)
- Writing code that controls hardware (e.g. turning on or off LEDs)
- Implementing encryption/decryption
- Encoding/decoding/manipulating data (e.g. the colours of a pixel in an image)

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

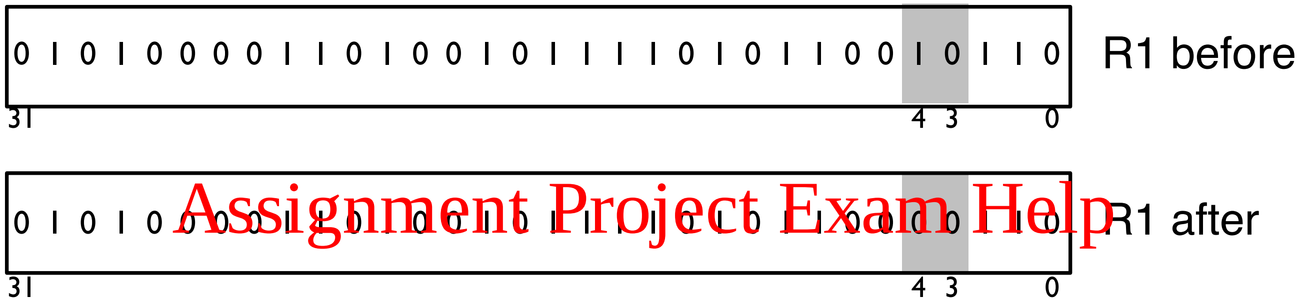
7.2 Bit Manipulation Examples

CSU11021 – Introduction to Computing I

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

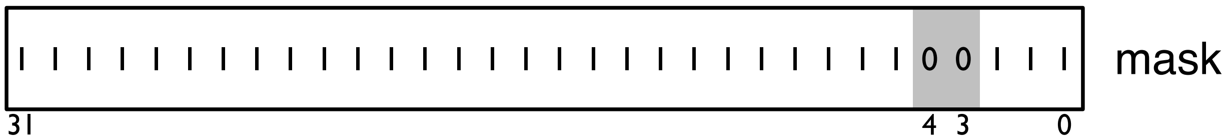
Bit Manipulation – Clear Bits

e.g. Clear bits 3 and 4 (i.e. the 4th and 5th bits) of the value in R1



Observe x ∧ 0 = 0 and x ∧ 1 = x

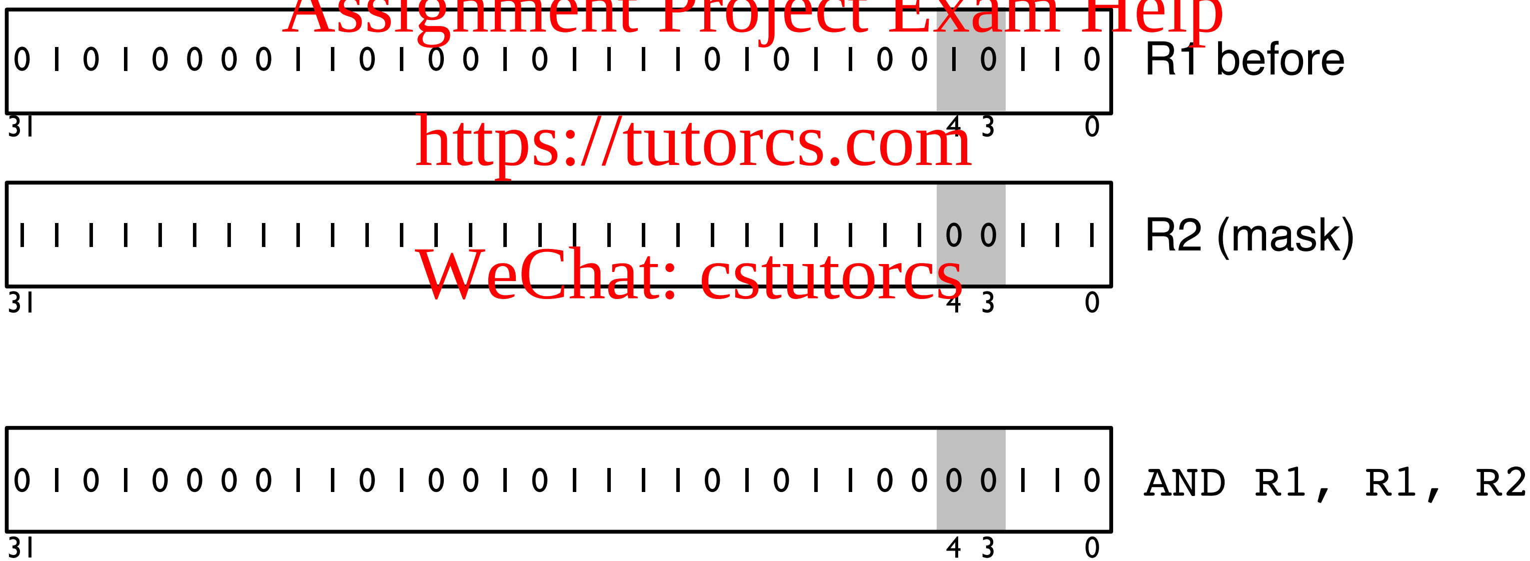
Construct a mask with 0 in the bit positions we want to clear and 1 in the bit positions we want to leave unchanged



Perform a bitwise logical AND of the value with the mask

Bit Manipulation – Clear Bits

e.g. Clear bits 3 and 4 of the value in R1 (continued)



Example: Clear Bits (using AND)

Write an assembly language program to clear bits 3 and 4 (i.e. the 4th and 5th bits) of the value in R1


```
LDR     R1, =0x61E87F4C @ load test value
LDR     R2, =0xFFFFFFE7 @ mask to clear bits 3 and 4
AND     R1, R1, R2      @ clear bits 3 and 4
                        @ result should be 0x61E87F44
```

Example: Clear Bits (using BIC)

Alternatively, the BIC (BIt Clear) instruction allows us to define a mask with 1's in the positions we want to clear

```
LDR     R2, =0x00000018 @ mask to clear bits 3 and 4
BIC     R1, R1, R2      @ R1 = R1 AND NOT(R2)
```

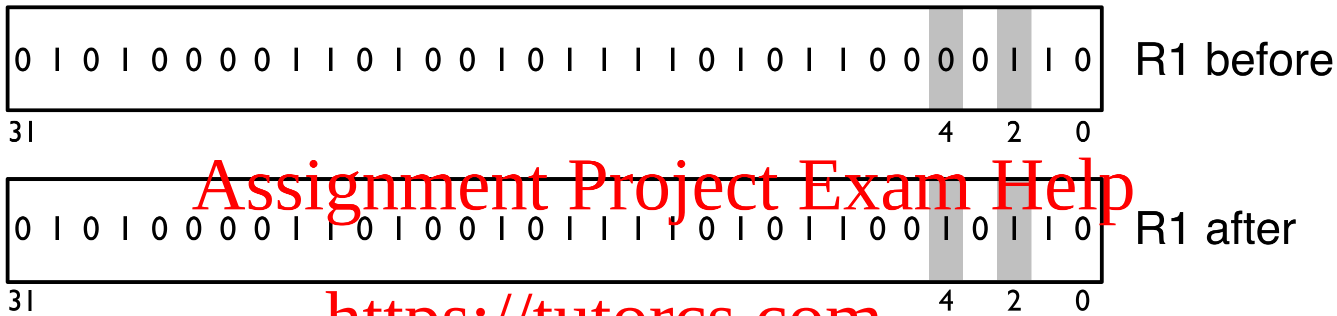

Or use an immediate value, saving one instruction


```
BIC     R1, R1, #0x00000018 @ R1 = R1 AND NOT(0x00000018)
```


The choice of AND or BIC is up to you but it may be more efficient or make more logical sense to choose one over the other, depending on the circumstances.

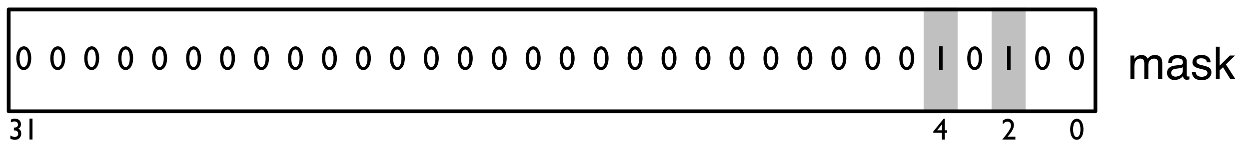
Bit Manipulation – Set Bits

e.g. Set bits 2 and 4 (i.e. the 3rd and 5th bits) of the value in R1



Observe x ∨ 1 = 1 and x ∨ 0 = x

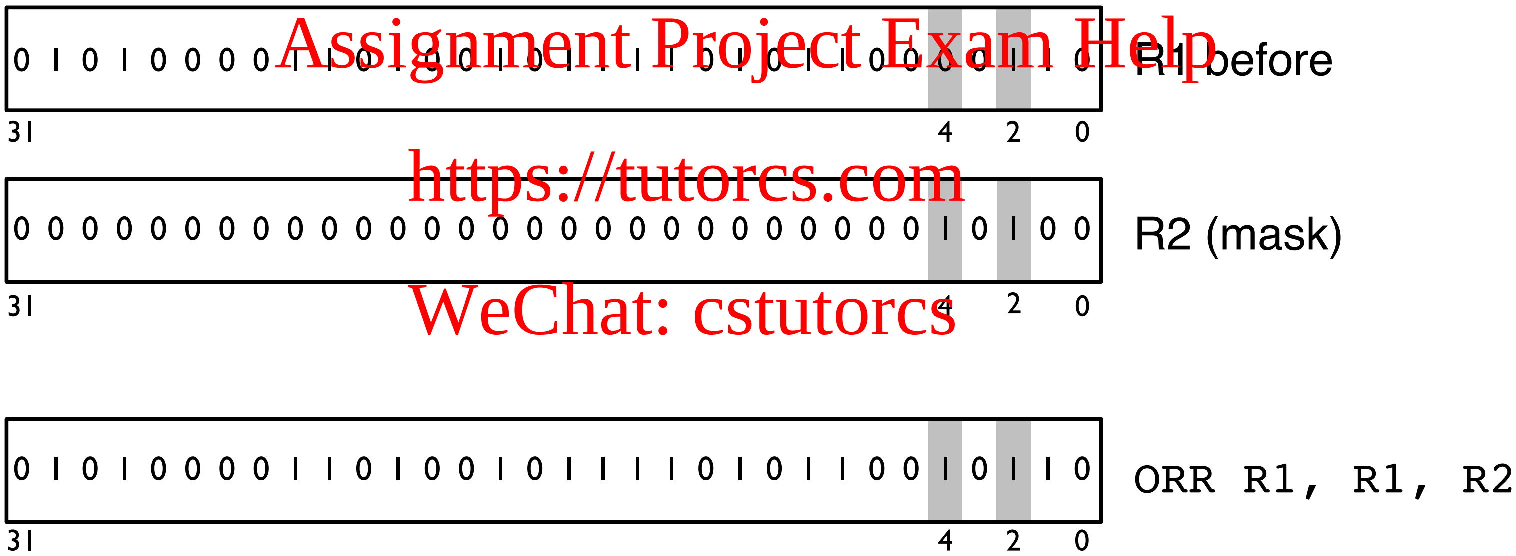
Construct a mask with 1 in the bit positions we want to set and 0 in the bit positions we want to leave unchanged



Perform a bitwise logical OR of the value with the mask

Bit Manipulation – Set Bits

e.g. Set bits 2 and 4 of the value in R1 (continued)



Example: Set Bits

Write an assembly language program to set bits 2 and 4 (i.e. the 3rd and 5th bits) of the value in R1

```
LDR     R1, =0x61E87F4C @ load test value
LDR     R2, =0x00000014 @ mask to set bits 2 and 4
ORR     R1, R1, R2      @ set bits 2 and 4
                        @ result should be 0x61E87F5C
```


Save one instruction by specifying the mask as an immediate operand in the ORR instruction

```
ORR     R1, R1, #0x00000014     @ set bits 2 and 4
```

REMEMBER: like MOV, only some immediate operands can be encoded. Assembler will warn you if the immediate operand you specify is invalid (is too large to be encoded in the ORR machine code instruction)

Bit Manipulation – Invert Bits

e.g. Invert bits 2 and 4 (i.e. the 3rd and 5th bits) of the value in R1

0	1	0	1	0	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0	1	1	0	R1 before
31																											4		2		0	

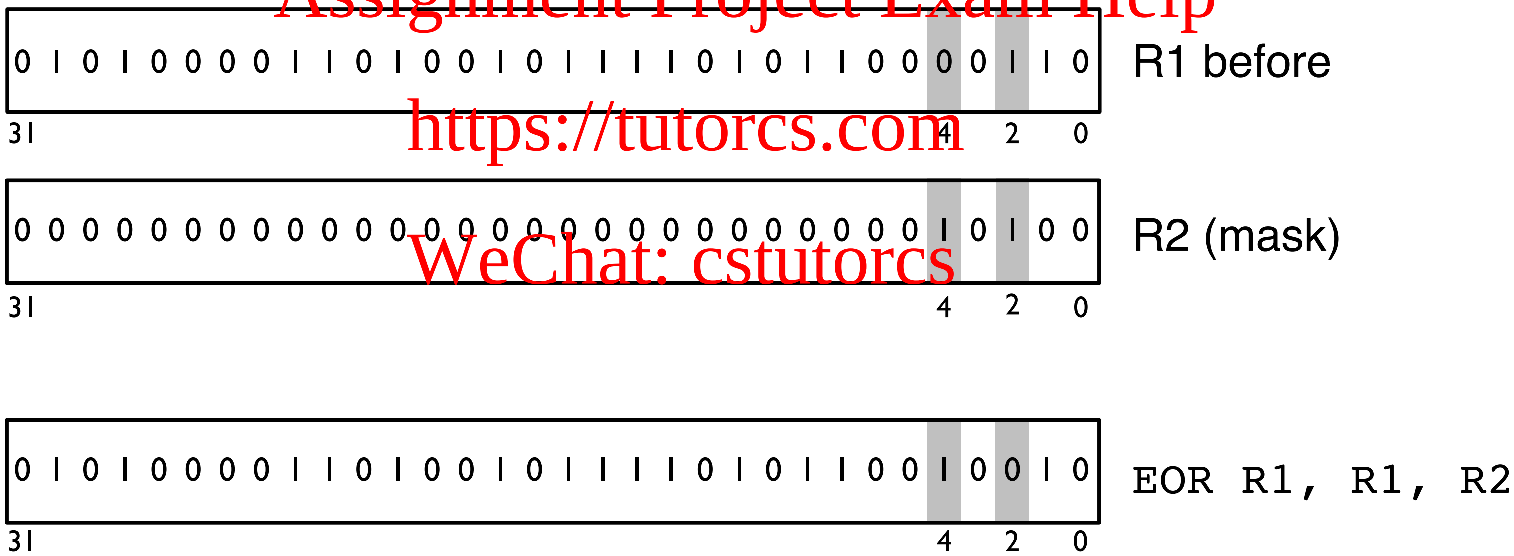
0	1	0	1	0	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	1	0	R1 after
31																											4		2		0	

Observe x ⊕ 1 = ¬x and x ⊕ 0 = x

Construct a mask with 1 in the bit positions we want to invert and 0 in the bit positions we want to leave unchanged

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	mask
31																											4		2		0	

Perform a bitwise logical exclusive-OR of the value with the mask e.g. Invert bits 2 and 4 of the value in R1 (continued)



Example: Invert Bits

Write an assembly language program to invert bits 2 and 4 of the value in R1


```
LDR     R1, =0x61E87F4C @ load test value
LDR     R2, =0x00000014 @ mask to invert bits 2 and 4
EOR     R1, R1, R2      @ invert bits 2 and 4
                        @ result should be 0x61E87F46
```


Again, can save an instruction by specifying the mask as an immediate operand in the EOR instruction

```
EOR     R1, R1, #0x00000014     @ invert bits 2 and 4
```

Again, only some 32-bit immediate operands can be encoded

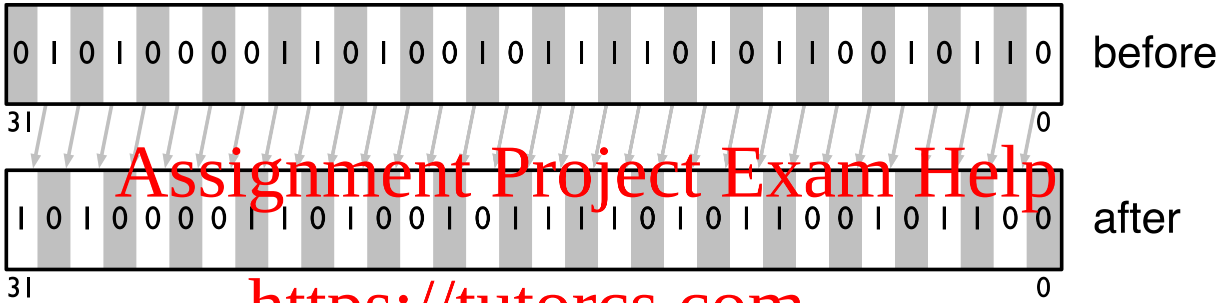
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

7.3 Shifts, Rotates and Exercises

CSU11021 – Introduction to Computing I

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

Logical Shift Left by 1 bit position

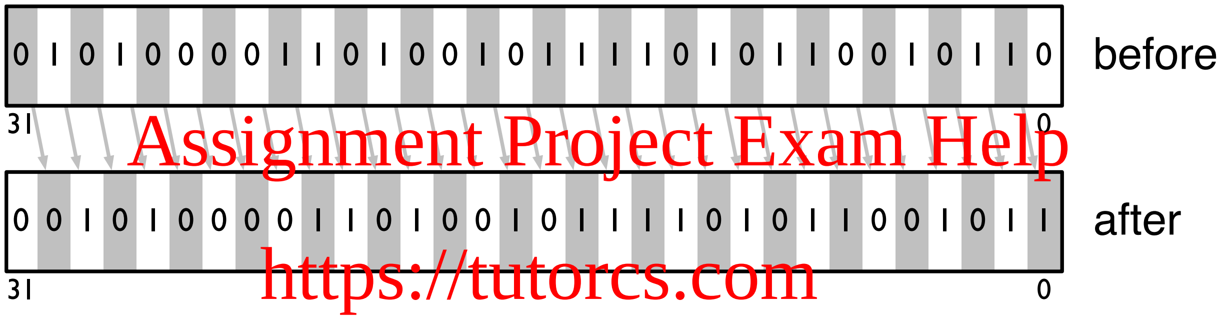


ARM MOV instruction allows a source operand, Rm, to be shifted left by n = 0 … 31 bit positions before being stored in the destination operand, Rd

```
MOV Rd, Rm, LSL #n
```

LSB of Rd is set to zero, MSB of Rm is discarded

Logical Shift Right by 1 bit position



ARM MOV instruction allows a source operand, Rm, to be shifted right by n = 0 … 31 bit positions before being stored in the destination operand, Rd

```
MOV Rd, Rm, LSR #n
```

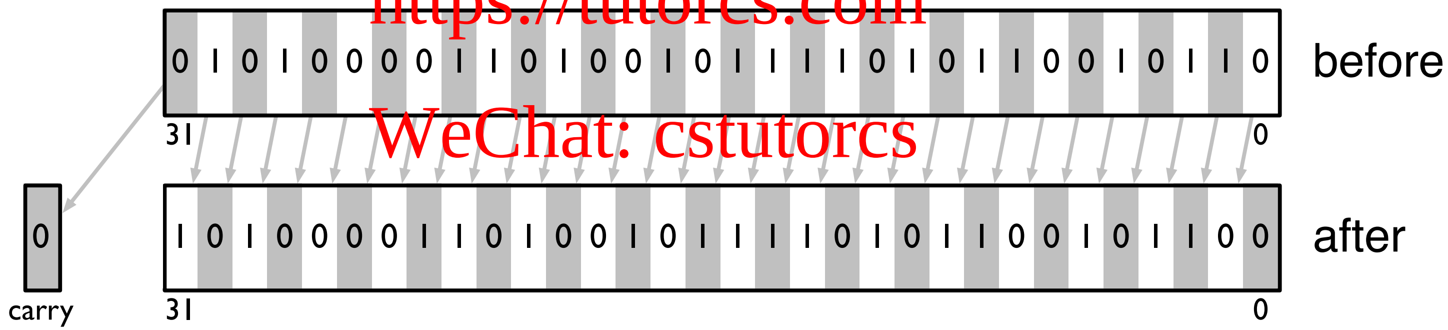
MSB of Rd is set to zero, LSB of Rm is discarded

MOVS

Instead of discarding the MSB when shifting left (or LSB when shifting right), we can cause the last bit shifted out to be stored in the Carry Condition Code Flag

By using MOVS instead of MOV

(i.e. by setting the S-bit in the MOV machine code instruction)

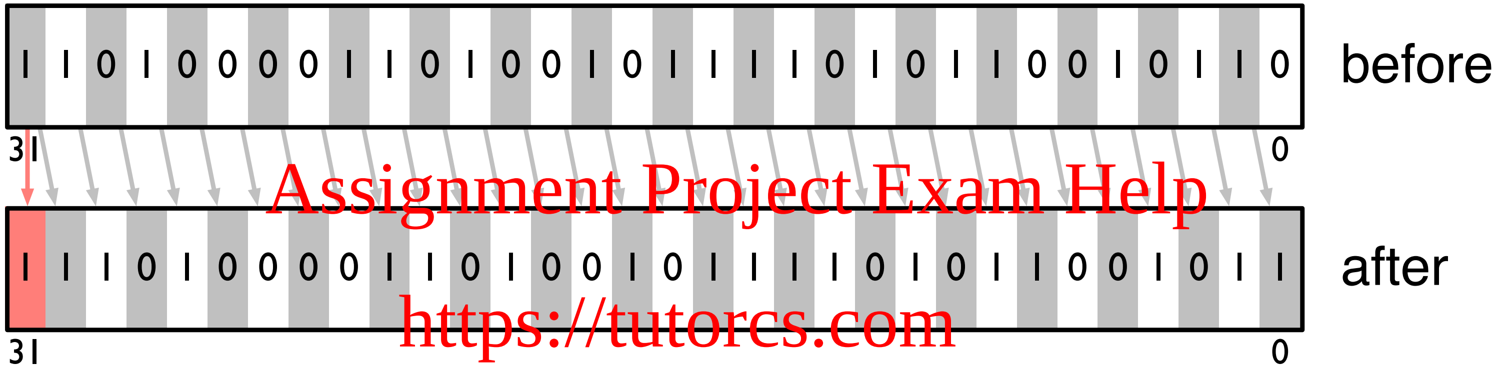



```
MOVS Rd, Rm, LSL #n

MOVS Rd, Rm, LSR #n
```

Arithmetic Shift Right

e.g. Arithmetic Shift Right by 1 bit position



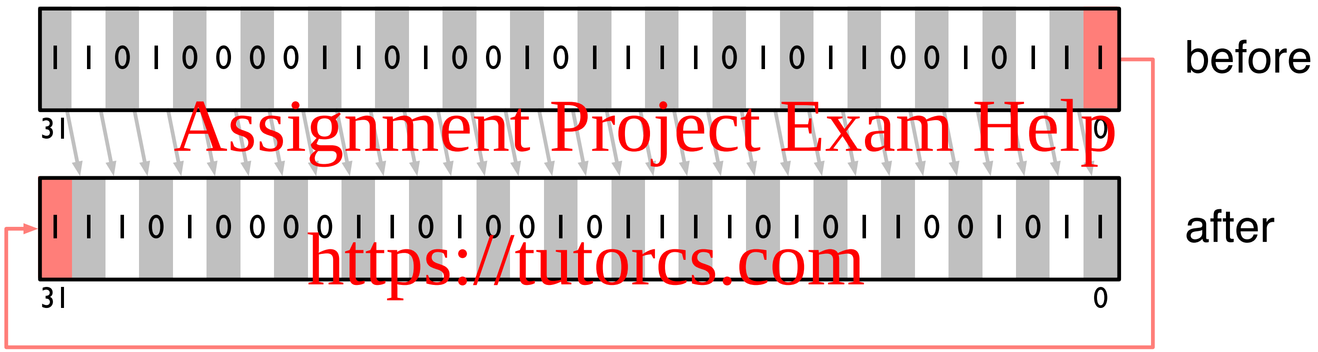
ASR shifts source operand, Rm, right by n = 0 … 31 bit positions, copying the sign (MSB) from the source to the sign (MSB) of the destination operand, Rd

```
MOV Rd, Rm, ASR #n
```

If right-shift is used for division, ASR maintains correct sign

Rotate Right

Rotate Right by 1 bit position



ROR rotates source operand, Rm, to the right by n = 0 … 31 bit positions before being stored in the destination operand, Rd

```
MOV Rd, Rm, ROR #n
```

MSB of Rd is set to LSB of Rm

No ROL?

Bonus problem: Shift and Add Multiplication

We can express multiplication by any value as the sum of the results of multiplying the value by different powers of 2. For example:

$$a \times 12 = a \times (8 + 4) = a \times (2^3 + 2^2) = (a \times 2^3) + (a \times 2^2)$$

Multiplication of a value by 2^n can be implemented efficiently by shifting the value left by n bits. For example:

$$a \times 12 = (a \ll 3) + (a \ll 2), \text{ where } \ll \text{ is logical shift left}$$

Hint: You can quickly see the powers of two that are needed by inspecting the (binary) multiplier! (e.g. 12 in binary is 0000**11**00)

Design and write an ARM Assembly Language Program that will use shift-and-add multiplication to multiply the value in R1 by the value in R2, storing the result in R0.