# Logical Operations



- Logical operations perform operations on the bits themselves, rather than the values they represent
  - e.g. and, or, exclusive-or, not (invert)
- Truth tables

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

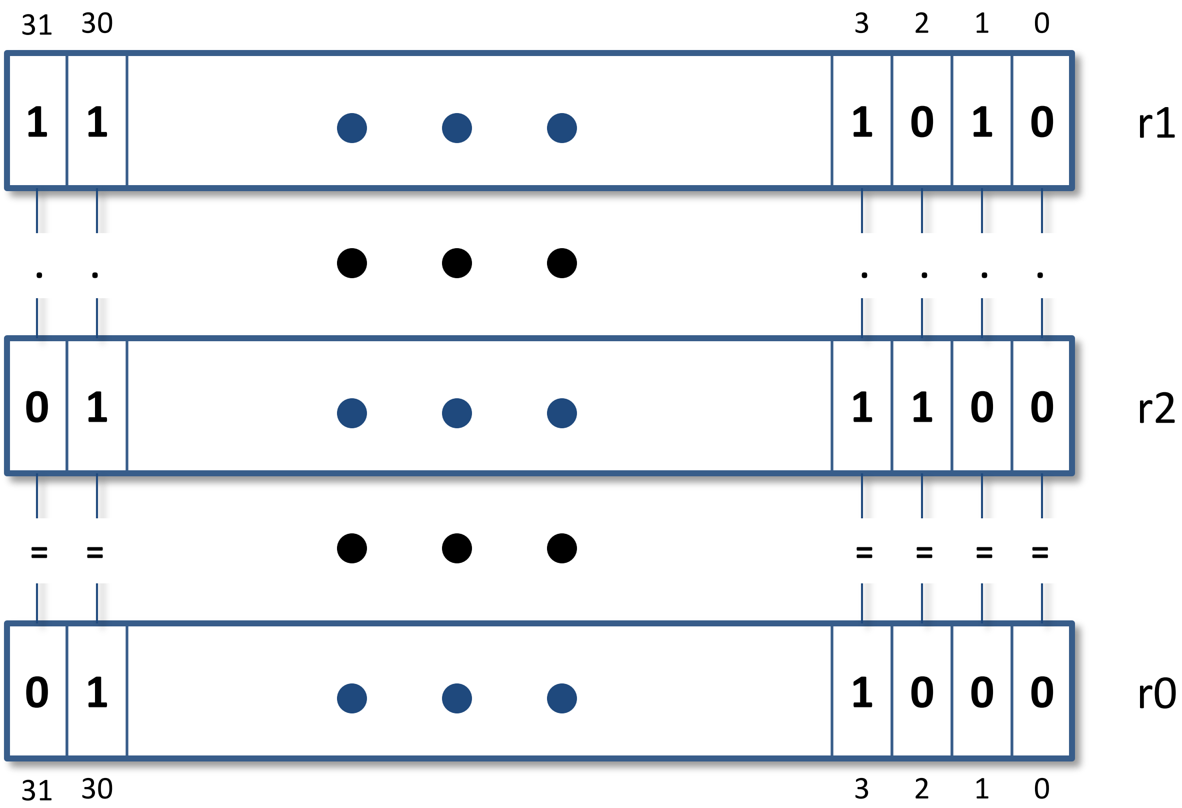
| A | B | A EOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Instructions for Logical Operations

- Bitwise logical AND – AND

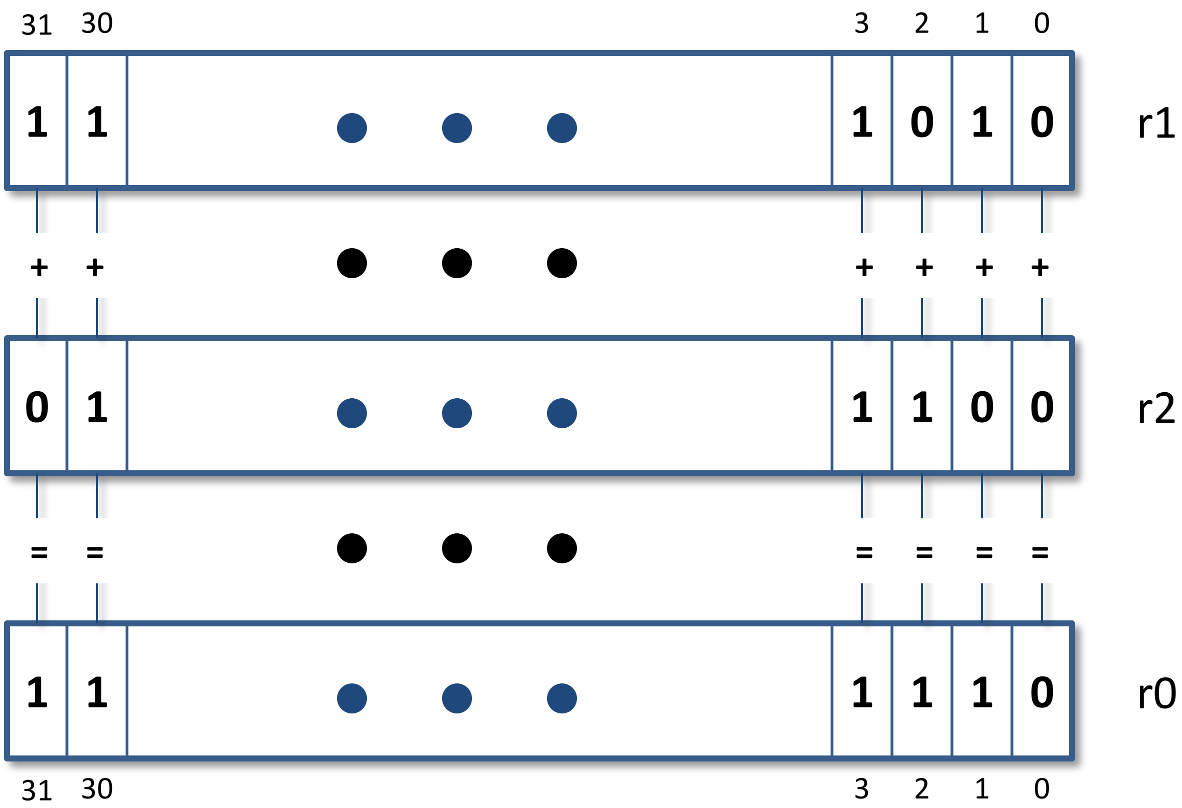
```
AND       r0, r1, r2        ; r0 = r1 . r2 (r1 AND r2)
```



# Instructions for Logical Operations

- Bitwise logical OR – ORR

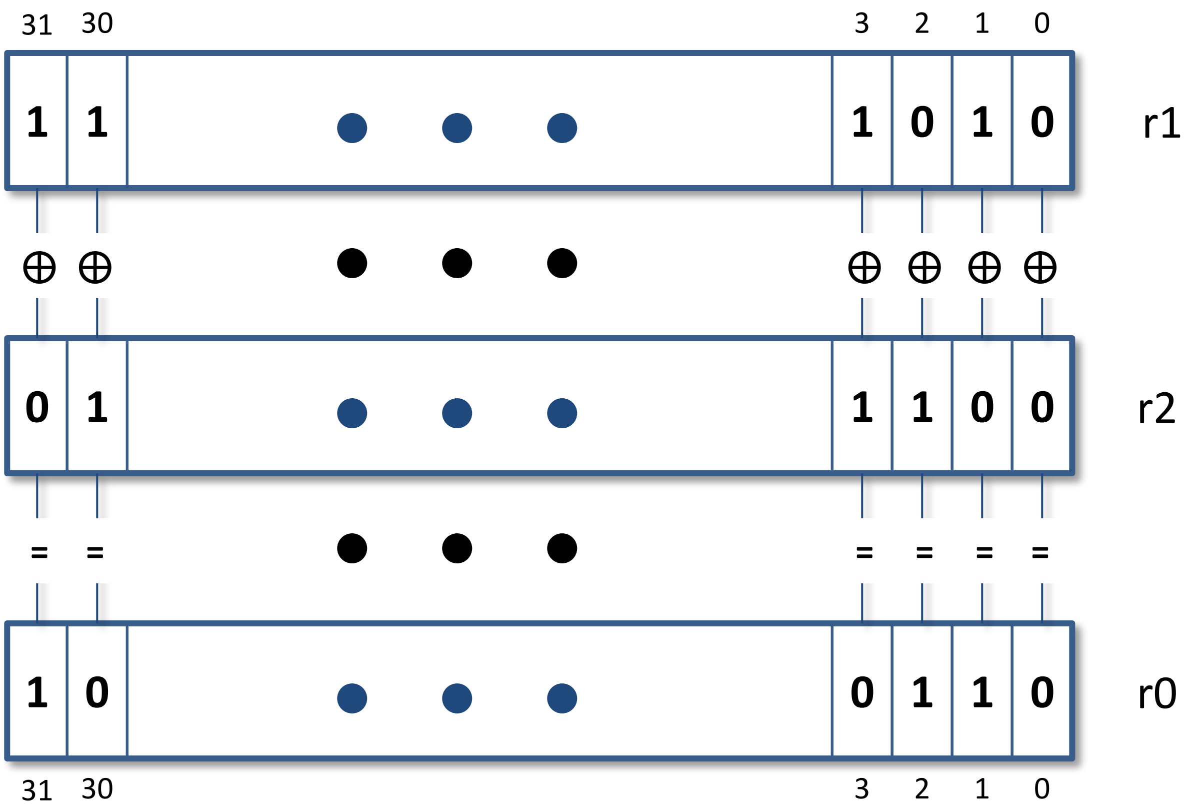
```
ORR     r0, r1, r2      ; r0 = r1 + r2 (r1 OR r2)
```



# Instructions for Logical Operations

- Bitwise logical Exclusive OR – EOR

```
EOR    r0, r1, r2    ; r0 = r1 ⊕ r2 (r1 EOR r2)
```

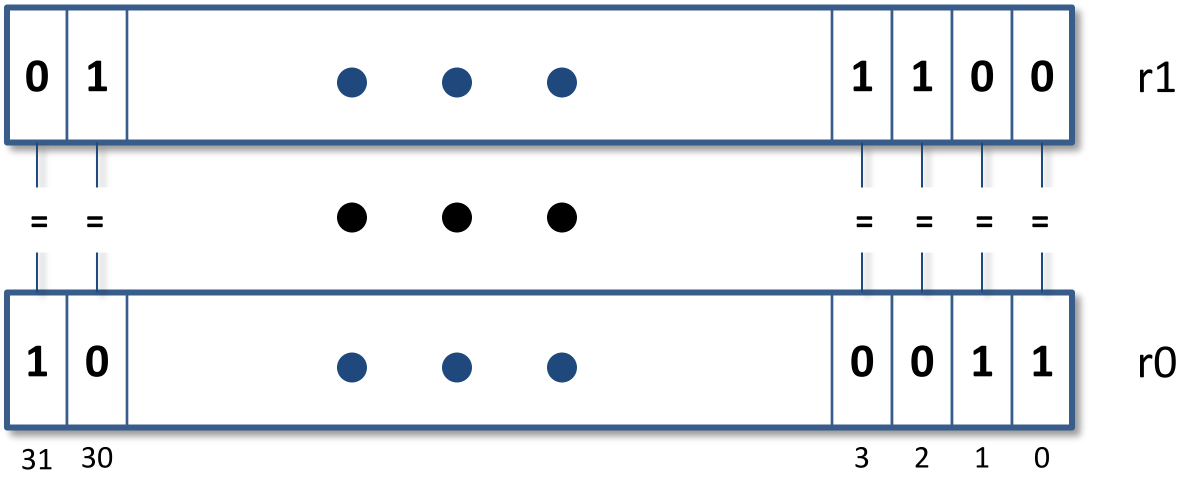


# Instructions for Logical Operations

- Bitwise logical inversion
- MVN **M**o**V**e **N**egative – like MOV but moves the one's complement of a value (bitwise inversion) to a register

```
MVN     r0, r0          ; r0 = r0' (NOT r0)
```

```
MVN     r0, r1          ; r0 = r1' (NOT r1)
```



# Bit Manipulation



- e.g. Clear bits 3 and 4 of the value in r1

| | 31 | 30 | … | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| before | 0 | 1 | … | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| after | 0 | 1 | … | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

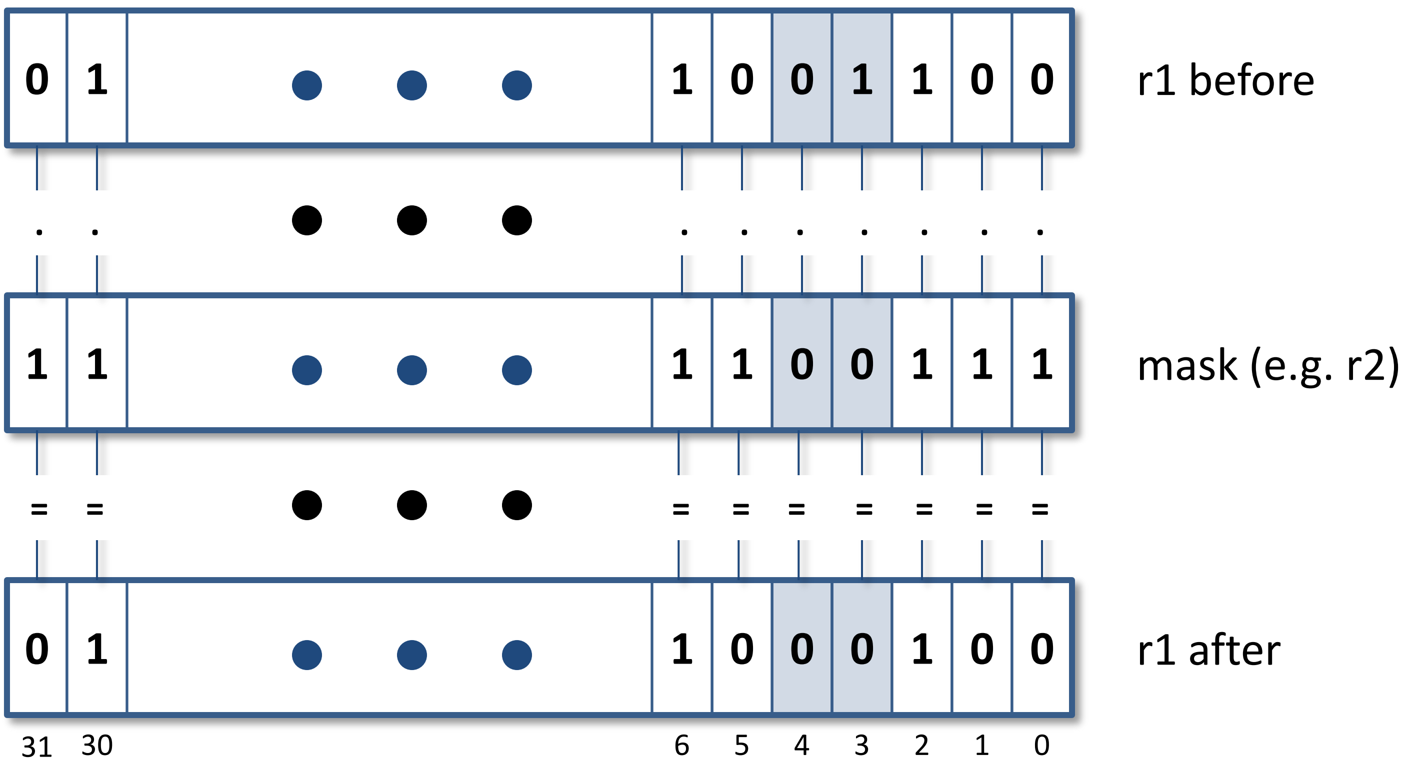
  - Observe $0 \cdot x = 0$ and $1 \cdot x = x$
  - Construct a mask with 0 in the bit positions we want to clear and 1 in the bit positions we want to leave unchanged

| | 31 | 30 | … | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| mask | 1 | 1 | … | 1 | 1 | 0 | 0 | 1 | 1 | 1 |

  - Perform a bitwise logical AND of the value with the mask

# Bit Manipulation

- e.g. Clear bits 3 and 4 of the value in r1 (continued)



# Program 5.1 – Clear Bits

- Write an assembly language program to clear bits 3 and 4 of the value in r1

```
start
            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

stop        B       stop
```

- Alternatively, the BIC (**BI**t **C**lear) 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 . NOT(r2)
```

- Or use an immediate value, saving one instruction

```
            BIC     r1, r1, #0x00000018     ; r1 = r1 . NOT(0x00000018)
```

# Bit Manipulation

- e.g. Set bits 2 and 4 of the value in r1

| 31 | 30 | … | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **1** | • • • | **1** | **0** | **0** | **1** | **1** | **0** | **0** | before |
| **0** | **1** | • • • | **1** | **0** | **1** | **1** | **1** | **0** | **0** | after |

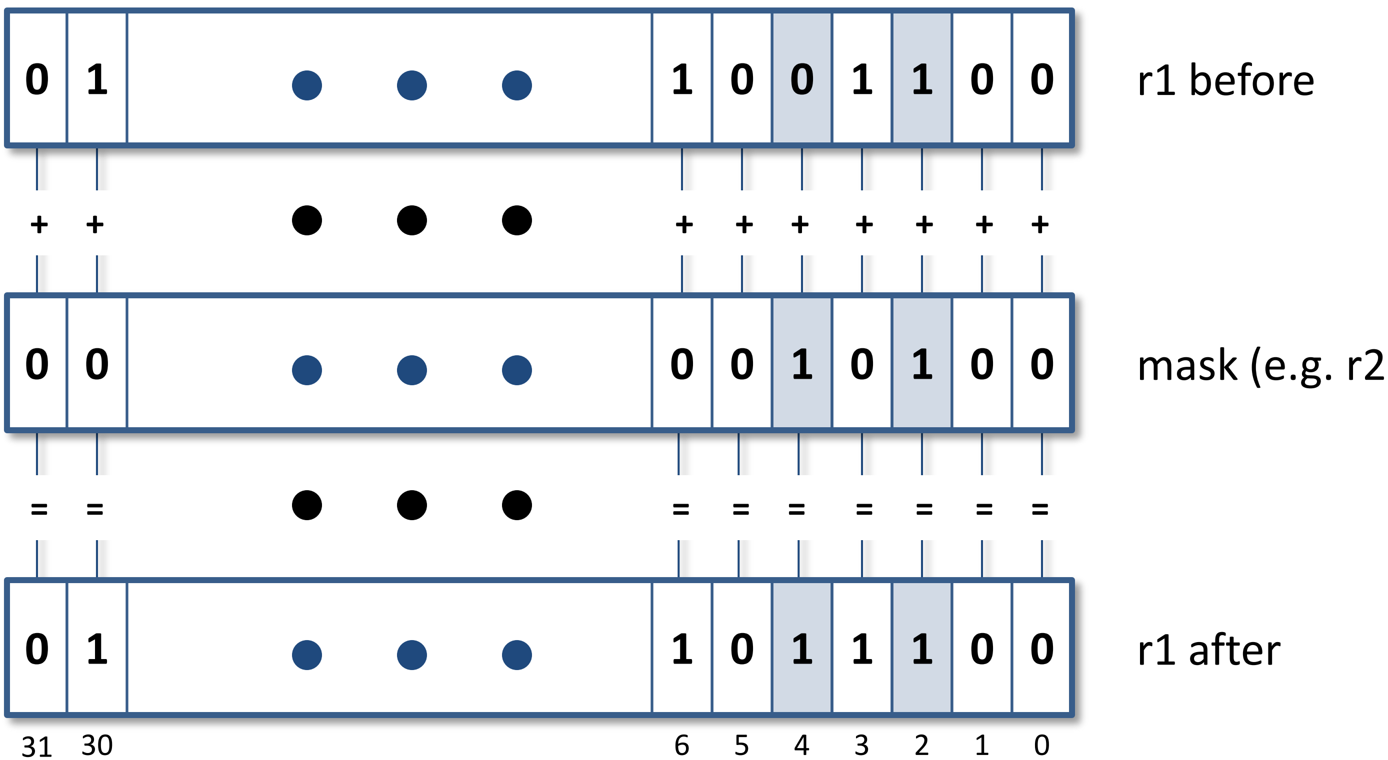
  - Observe $1 + x = 1$ and $0 + x = x$
  - Construct a mask with 1 in the bit positions we want to set and 0 in the bit positions we want to leave unchanged

| 31 | 30 | … | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **0** | • • • | **0** | **0** | **1** | **0** | **1** | **0** | **0** | mask |

  - Perform a bitwise logical OR of the value with the mask

# Bit Manipulation

- e.g. Set bits 2 and 4 of the value in r1 (continued)



# Program 5.2 – Set Bits

- Write an assembly language program to set bits 2 and 4 of the value in r1

```
start
            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

stop        B       stop
```

  - Can save an instruction by specifying the mask as an immediate operand in the ORR instruction

```
            ORR     r1, r1, #0x00000014         ; set bits 2 and 4
```

  - REMEMBER: since the ORR instruction must fit in 32 bits, only some 32-bit immediate operands can be encoded. Assembler will warn you if the immediate operand you specify is invalid.

# Bit Manipulation

- e.g. Invert bits 1 and 3 of the value in r1

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | ● ● ● | 1 | 0 | 0 | 1 | 1 | 0 | 0 | before |
| 31 | 30 | | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | ● ● ● | 1 | 0 | 0 | 0 | 1 | 1 | 0 | after |
| 31 | 30 | | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

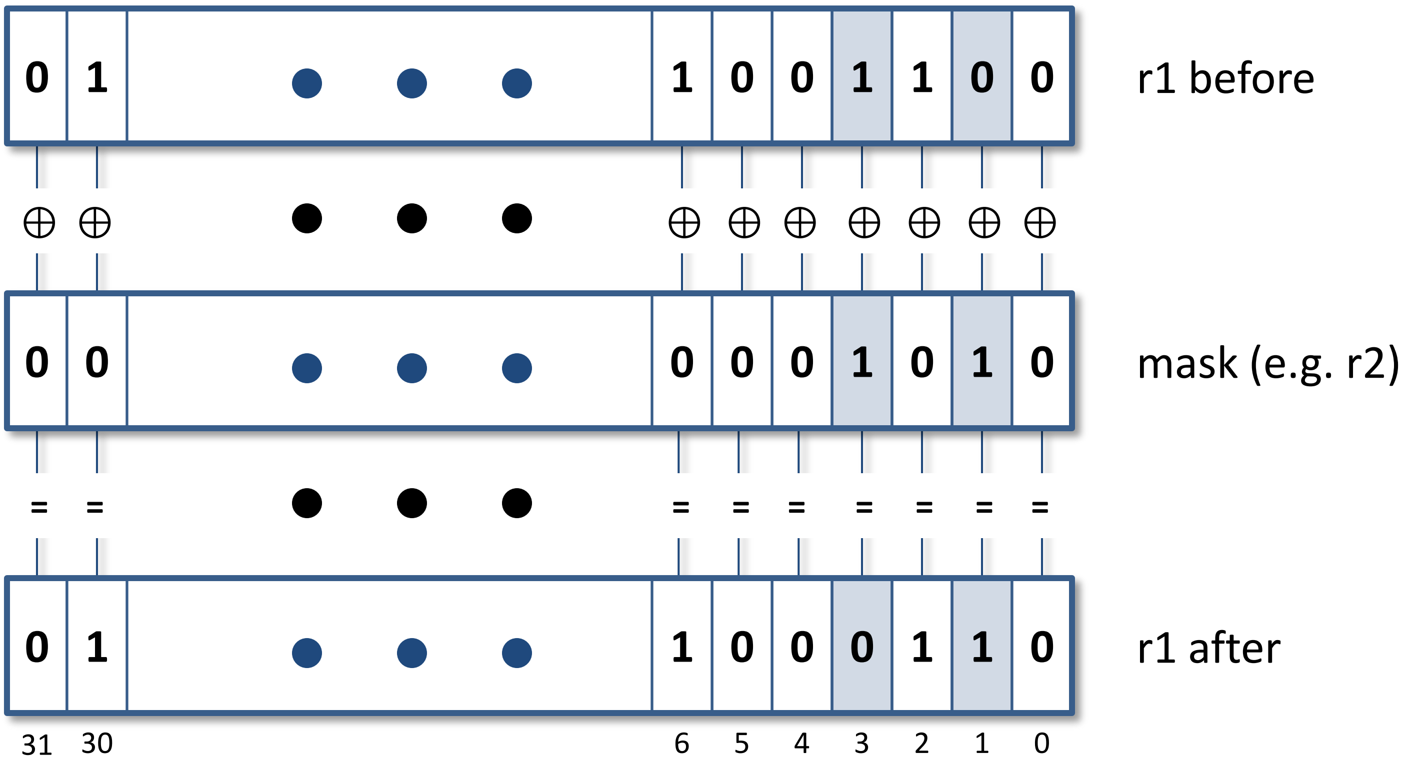
  - Observe $1 \oplus x = x'$ and $0 \oplus x = x$
  - Construct a mask with 1 in the bit positions we want to invert and 0 in the bit positions we want to leave unchanged

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | ● ● ● | 0 | 0 | 0 | 1 | 0 | 1 | 0 | mask |
| 31 | 30 | | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |

  - Perform a bitwise logical exclusive-OR of the value with the mask

# Bit Manipulation

- e.g. Invert bits 1 and 3 of the value in r1 (continued)



# Program 5.3 – Invert Bits



- Write an assembly language program to invert bits 1 and 3 of the value in r1

```
start
            LDR     r1, =0x61E87F4C              ; load test value
            LDR     r2, =0x0000000A              ; mask to invert bits 1 and 3
            EOR     r1, r1, r2                   ; invert bits 1 and 3
                                                 ; result should be 0x61E87F46

stop        B       stop
```

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

```
            EOR     r1, r1, #0x0000000A          ; invert bits 1 and 3
```

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

# Program 5.4 – Upper Case

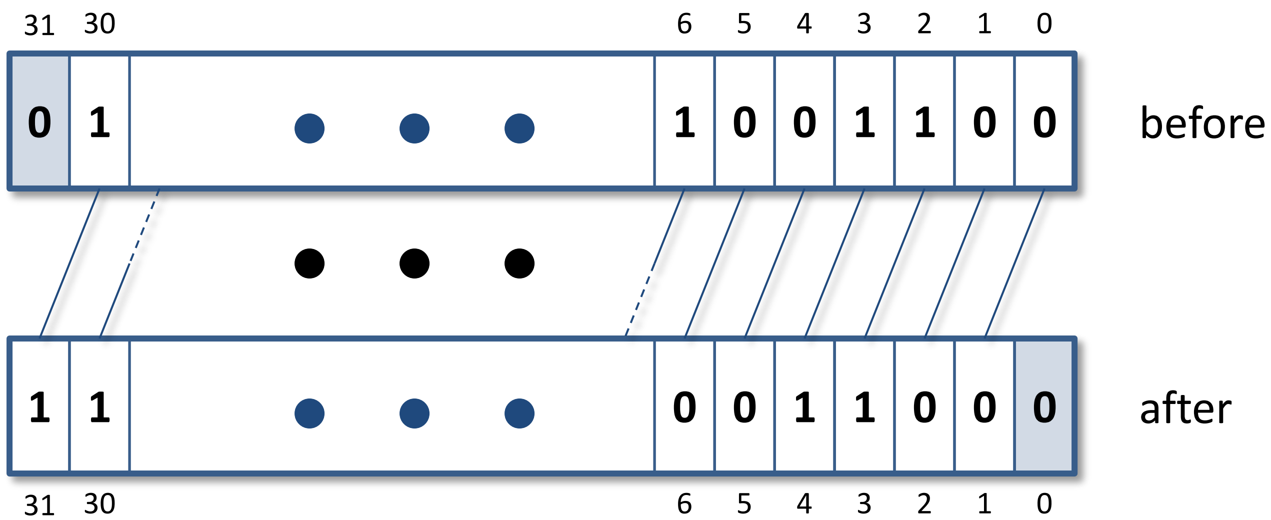


- Design and write an assembly language program that will make the ASCII character stored in r0 upper case.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | NUL | DLE | SPACE | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| A | LF | SUB | * | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [ | k | { |
| C | FF | FS | , | < | L | \ | l | \| |
| D | CR | GS | - | = | M | ] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | _ | o | DEL |

# Logical Shift Left

- Logical Shift Left by 1 bit position



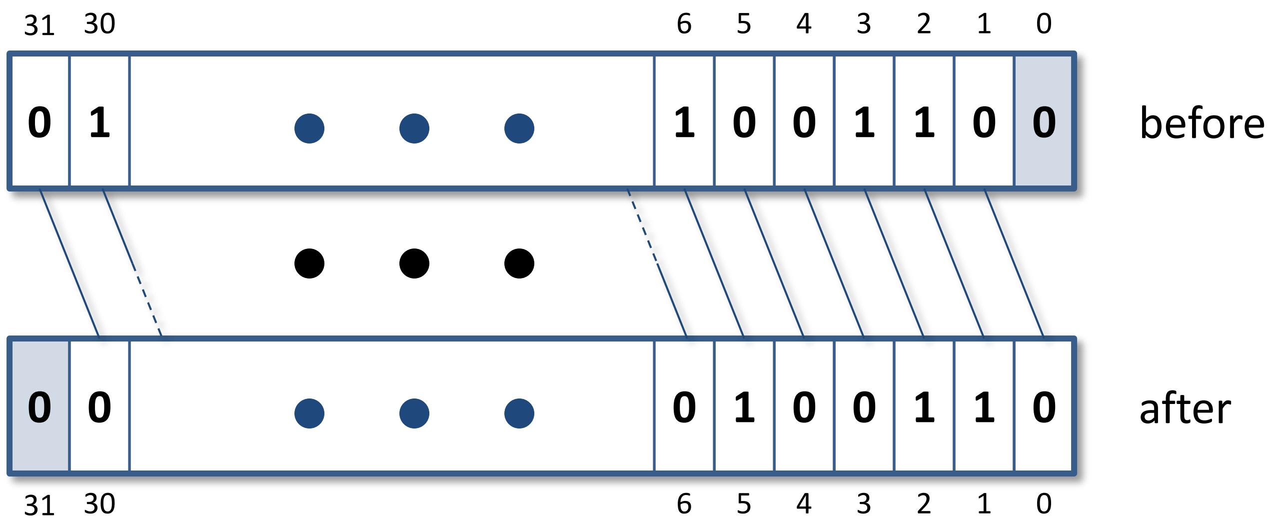

- ARM MOV instruction allows a source operand, R*m*, to be shifted left by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

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

  - LSB of R*d* is set to zero, MSB of R*m* is discarded

# Logical Shift Right

- Logical Shift Right by 1 bit position




- ARM MOV instruction allows a source operand, R*m*, to be shifted right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

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

  - MSB of R*d* is set to zero, LSB of R*m* is discarded

# Logical Shift Left/Right – Examples

- Logical shift left r1 by 2 bit positions

```
        MOV     r1, r1, LSL #2   ; r1 = r1 << 2
```

- Logical shift left r1 by 5 bit positions, store result in r0

```
        MOV     r0, r1, LSL #5   ; r0 = r1 << 5
```

- Logical shift right r2 by 1 bit position

```
        MOV     r2, r2, LSR #1   ; r2 = r2 >> 1
```

- Logical shift right r3 by 4 bit positions, store result in r1

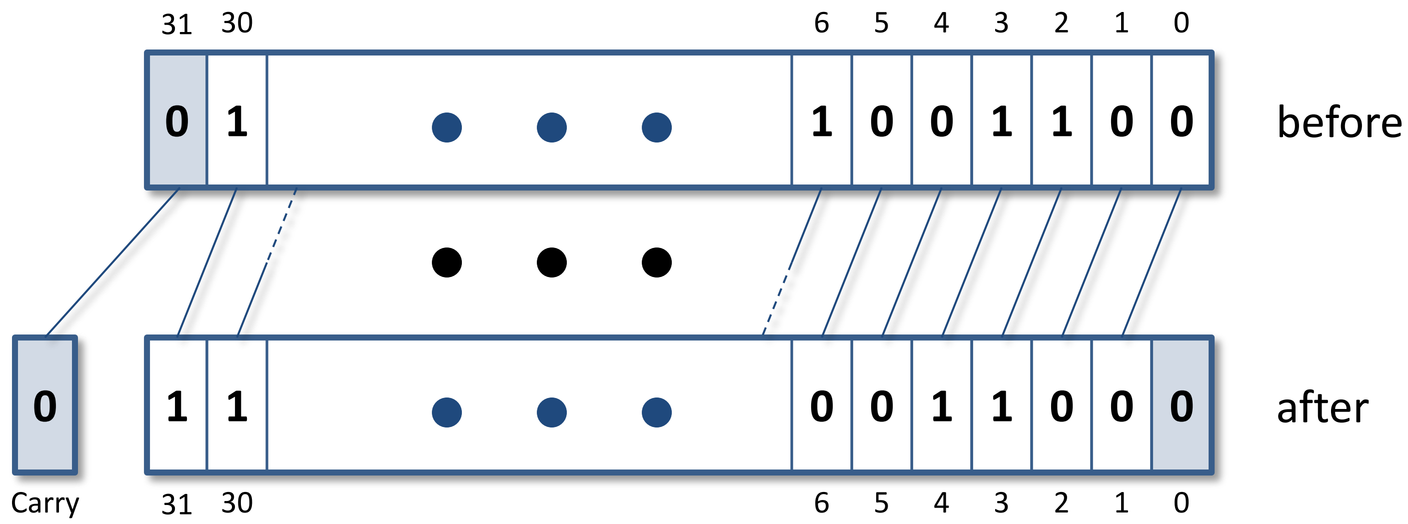
```
        MOV     r1, r3, LSR #4   ; r1 = r3 >> 4
```

- Logical shift left r4 by the number of positions in r0

```
        MOV     r4, r4, LSR r0   ; r4 = r4 >> r0
```

# 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 setting the S-bit in the MOV machine code instruction
  - By using MOVS instead of MOV




```
MOVS Rd, Rm, LSL #n
MOVS Rd, Rm, LSR #n
```

# Program 5.6 – Parity

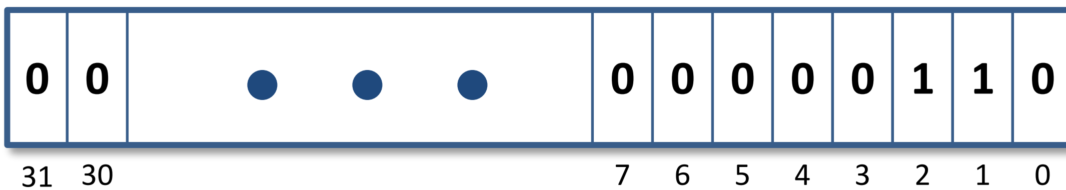
- Design and write an assembly language program that will calculate the parity bit for a 7-bit value stored in r1. The program should then store the computed parity bit in bit 7 of r1. Assume even parity.
- Parity bits are used to detect data transmission errors.
  - Using even parity, the parity bit of a value is set such that the number of set bits (1’s) in a value is always even.
- Parity example

| 31 | … | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ● ● ● | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | without parity bit |

| 31 | … | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ● ● ● | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | with parity bit |

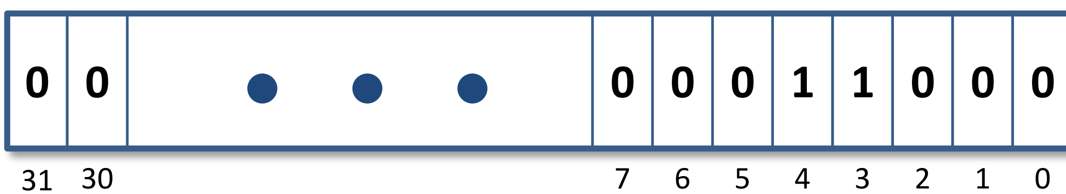
# Shift-And-Add Multiplication

- Shifting a binary value left (right) by *n* bit positions is an efficient way of multiplying (dividing) the value by $2^n$
- Example




r1 = 6

```
MOV r1, r1, LSL #2
```




$r1 = 6 \times 2^2 = 24$

# 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**
- Example
  - $a \times 12 = a \times (8 + 4) = a \times (2^3 + 2^2) = (a \times 2^3) + (a \times 2^2)$
  - $a \times 12 = (a << 3) + (a << 2)$
- Is there a simple way to determine which powers of 2 we need to use for our partial products?

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | **0** | ● ● ● | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **0** |
| $2^{31}$ | $2^{30}$ | | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

$12 = 2^3 + 2^2$

# Program 5.7 – Shift And Add Multiplication



- Design and write an assembly language program that will multiply the value in r1 by 12 and store the result in r0

```
start
            MOV     r0, r1, LSL #3              ; r0 = r1 * 2 ^ 3
            ADD     r0, r0, r1, LSL #2          ; r0 = r0 + r1 * 2^2

stop        B       stop
```

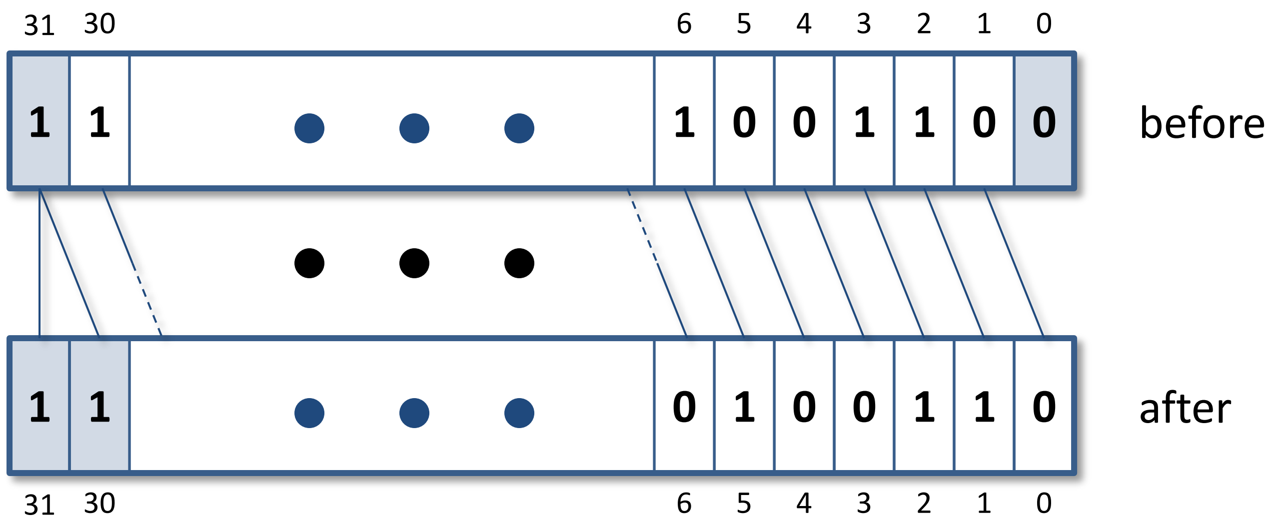
- [ASIDE] We can also formulate instructions to efficiently compute R$m$ × ($2^n$-1) or R$m$ × ($2^n$+1), saving one instruction

```
            ADD     r0, r1, r1, LSL #3          ; r0 = r0 * 9
```

```
            RSB     r0, r1, r1, LSL #3          ; r0 = r0 * 7
```

# Arithmetic Shift Right

- Arithmetic Shift Right by 1 bit position




- ARM MOV instruction allows a source operand, R*m*, to be shifted right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

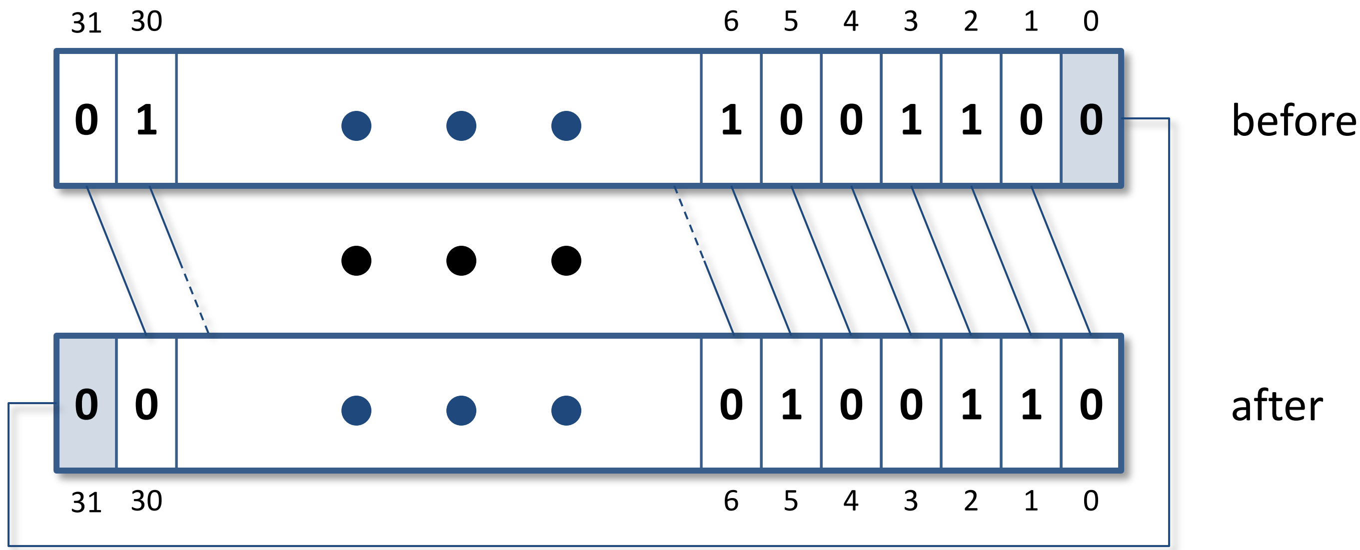
```
MOV Rd, Rm, ASR #n
```

  - MSB of R*d* is set to MSB of R*m*, LSB of R*m* is discarded

## Rotate Right



- Rotate Right by 1 bit position




- ARM MOV instruction allows a source operand, R*m*, to be rotated right by *n* = 0 ... 31 bit positions before being stored in the destination operand, R*d*

**MOV Rd, *Rm*, ROR #n**