CS1021 Introduction to Computing I
6. Bit Manipulation

Rebekah Clarke
clarker7@scss.tcd.ie

Bitwise Logical Operations

Binary digits take one of two discrete values:

- 0 or ‘FALSE’
- 1 or ‘TRUE’

Bitwise logical operations perform operations on the individual bits of a value, rather than the value as a whole

Binary Operators: AND, OR, EXCLUSIVE-OR

AND: Output is TRUE if all inputs are TRUE

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise Logical Operations

OR: Output is TRUE if at least one of the inputs is TRUE

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

EOR: Exclusive OR, Output is TRUE if exactly one of the inputs is true i.e. when the inputs differ

A	B	A EOR B
0	0	0
0	1	1
1	0	1
1	1	0

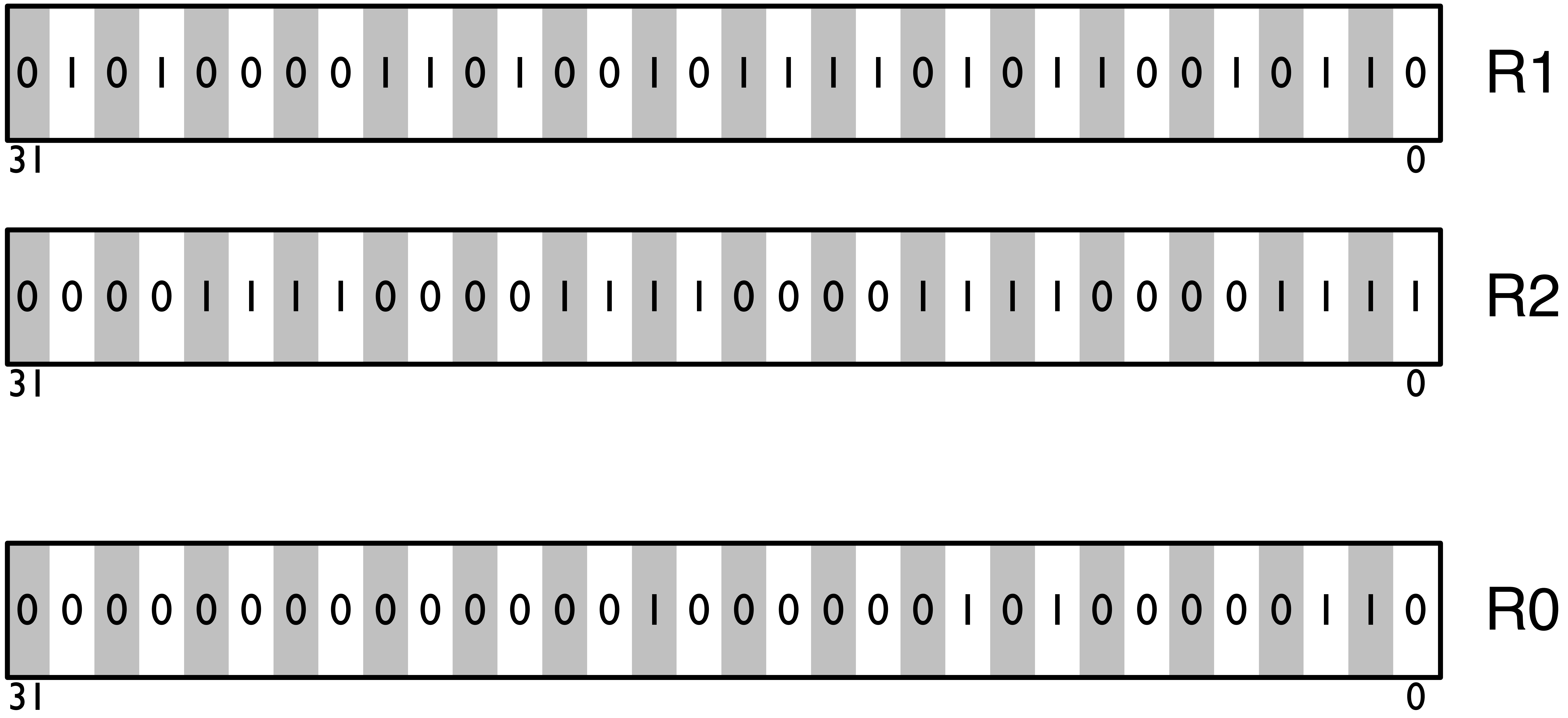
NOT: Output is TRUE if the input is FALSE

This is a unary operator, also called inversion or complement

A	NOT A
0	1
1	0

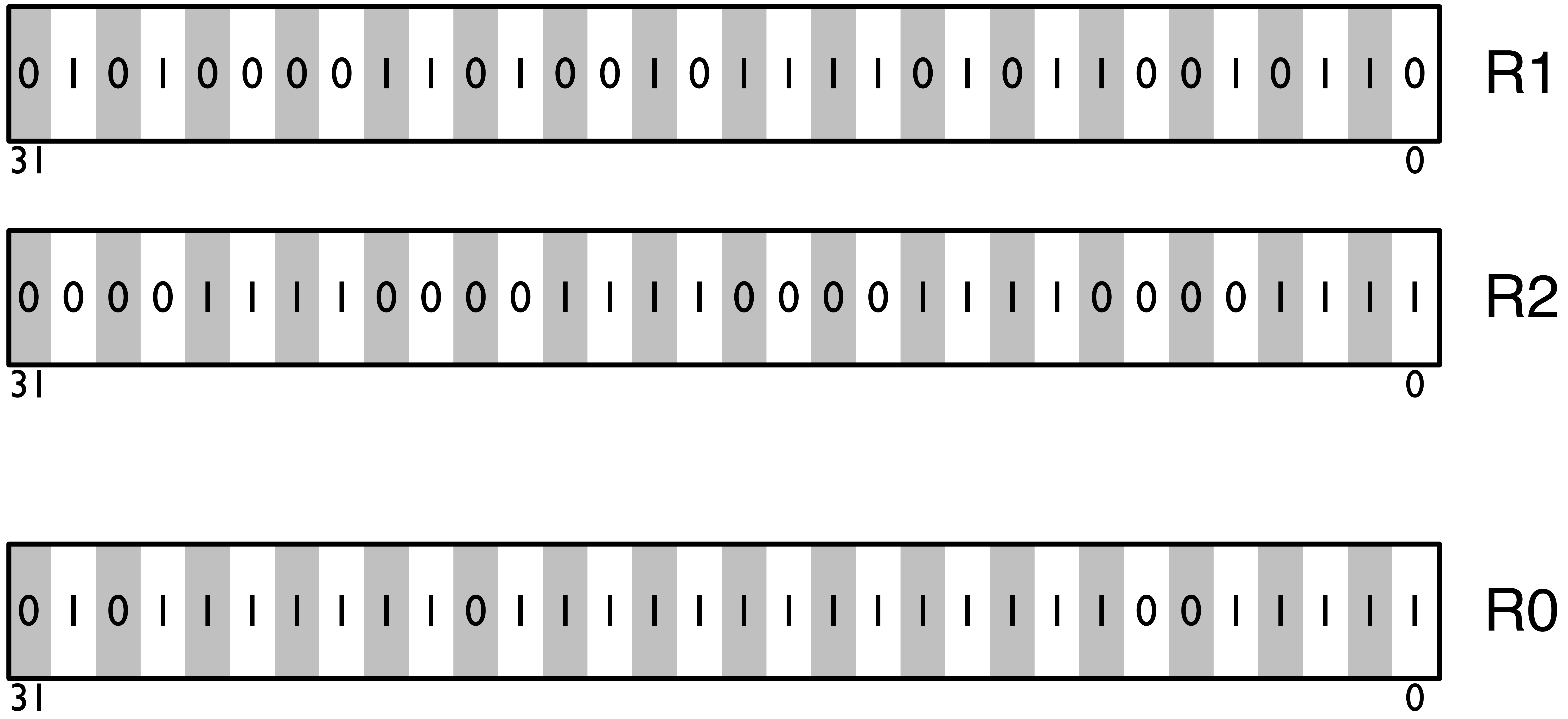
Bitwise Operation Instructions – AND

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



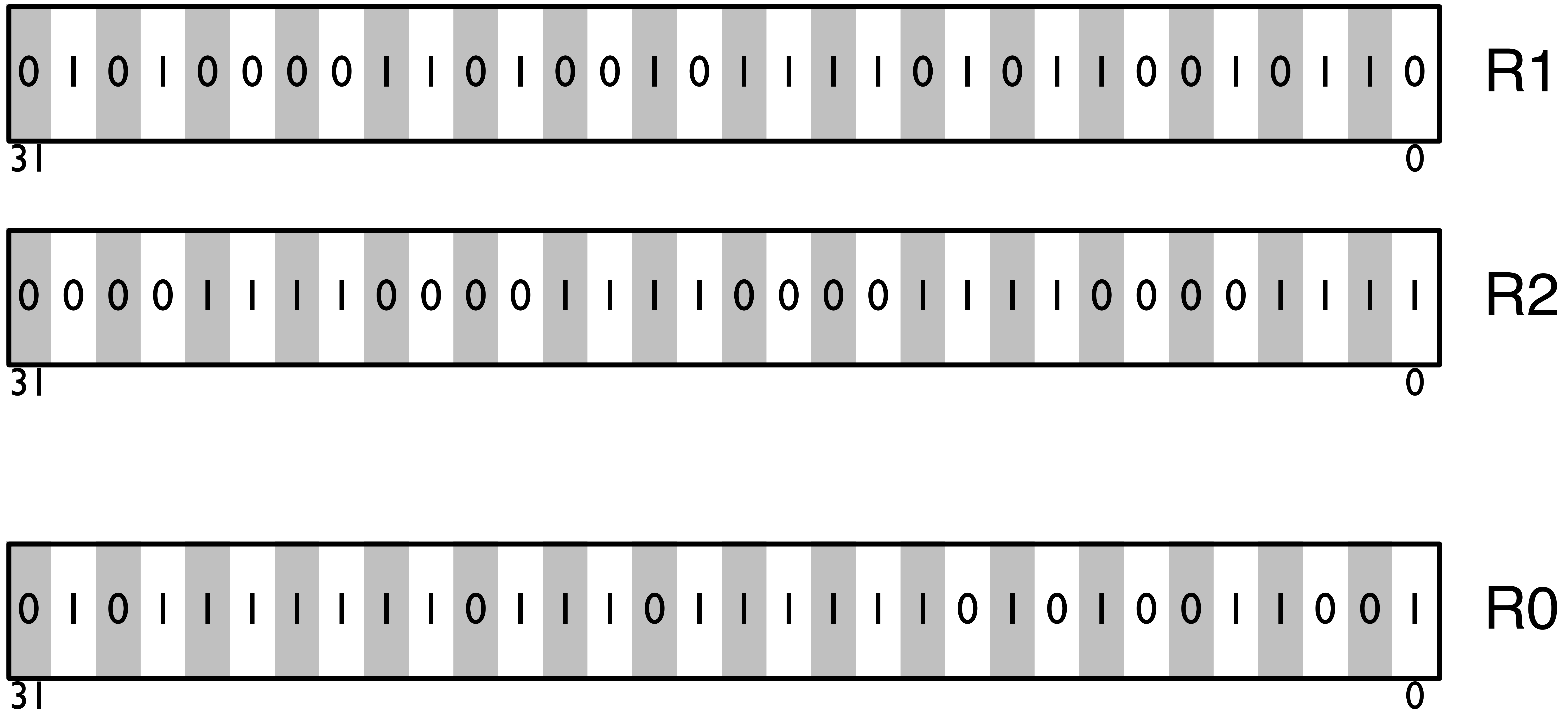
Bitwise Operation Instructions – OR

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



Bitwise Operation Instructions – EOR

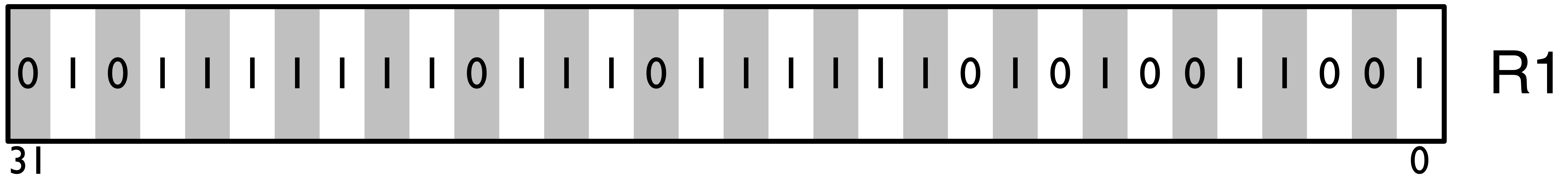
```
EOR  r0, r1, r2        ; r0 = r1 ⊕ r2 (r1 EOR r2)
```

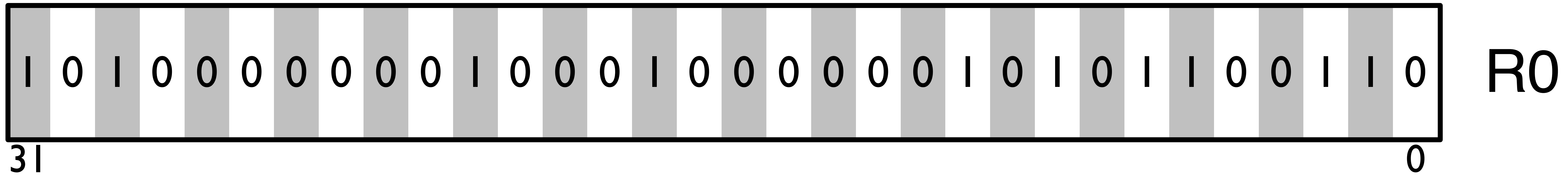


Bitwise Operation Instructions – NOT

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

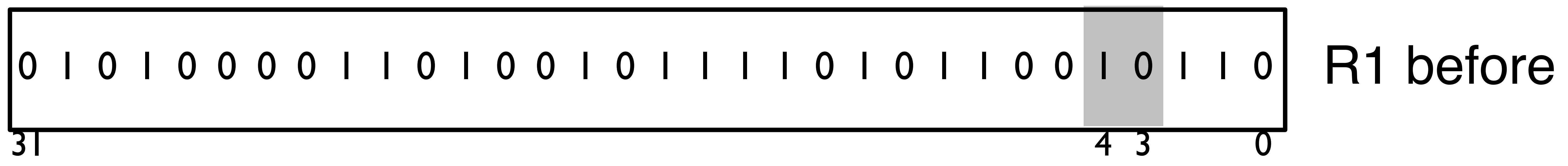
```
MVN  r0, r1              ; r0 = ¬r1 (NOT r1)
```

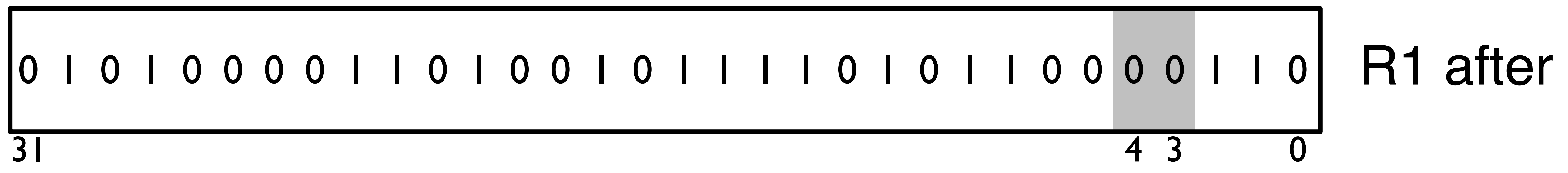




Bit Manipulation – Clear Bits

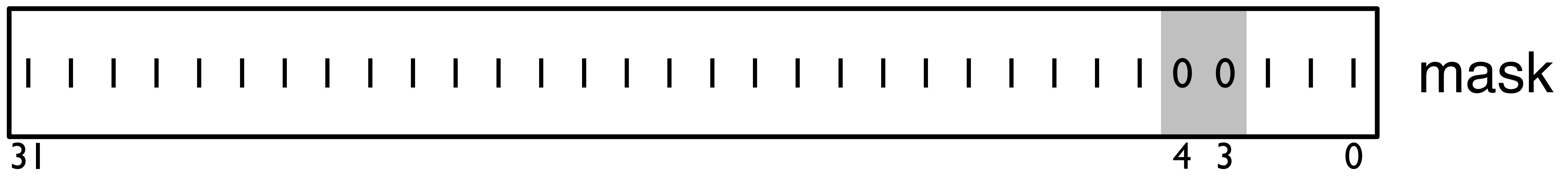
e.g. Clear bits 3 and 4 (i.e. the 4th and 5th bits) of the value in r1





Observe 0 · x = 0 and 1 · x = x

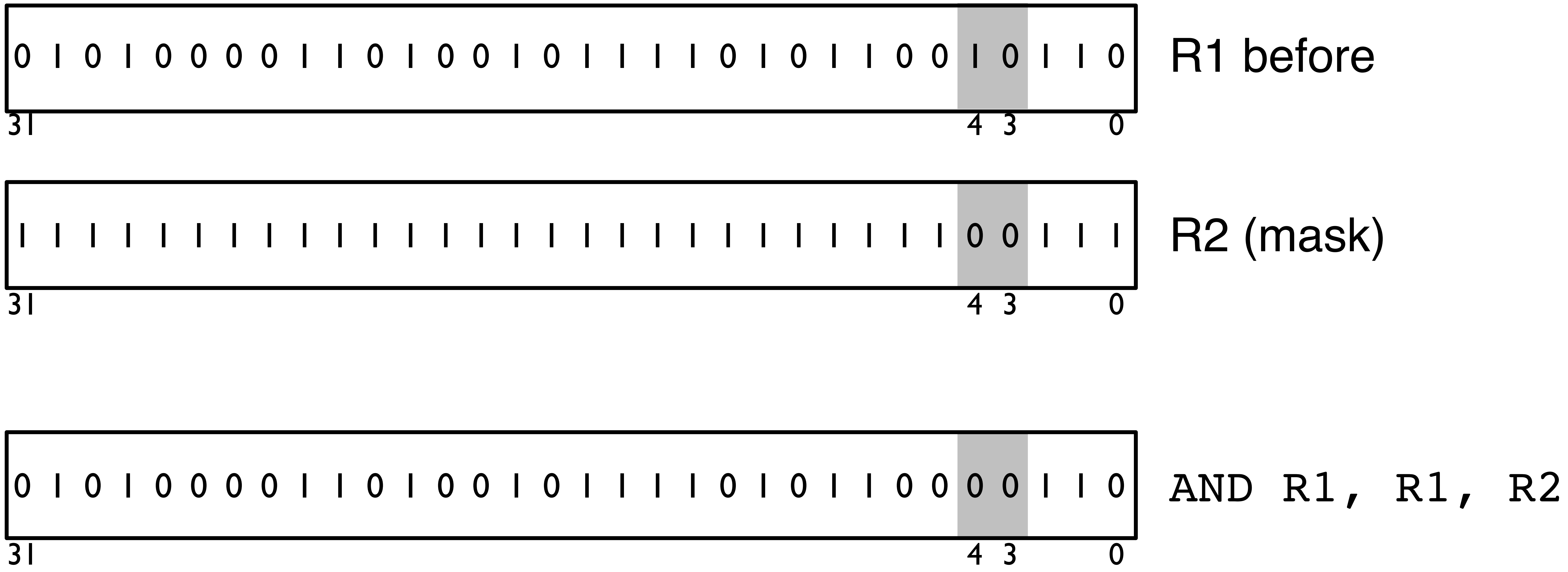
Construct a mask with 0 in the bit positions we want to clear and 1 in the bit positions we want to leave unchanged



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)



```
AND R1, R1, R2
```

Example: Clear Bits

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
```

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

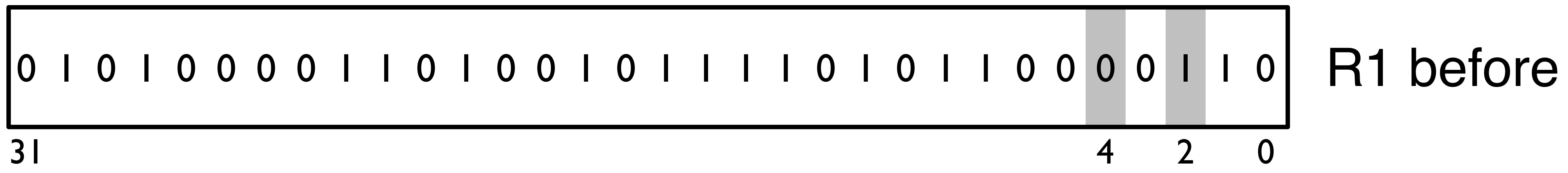
```
    LDR  r2, =0x00000018 ; mask to clear bits 3 and 4
    BIC  r1, r1, r2      ; r1 = r1 AND NOT(r2)
```

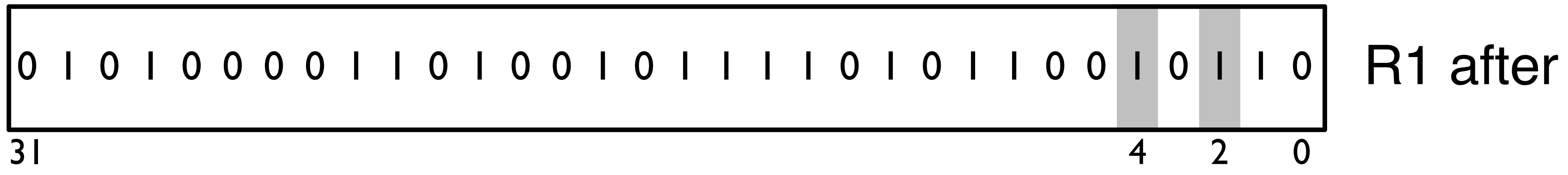
Or use an immediate value, saving one instruction

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

Bit Manipulation – Set Bits

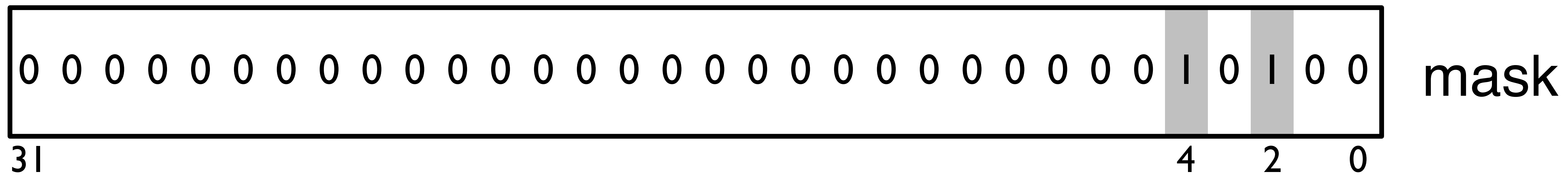
e.g. Set bits 2 and 4 (i.e. the 3rd and 5th bits) of the value in r1





Observe 1 + x = 1 and 0 + x = x

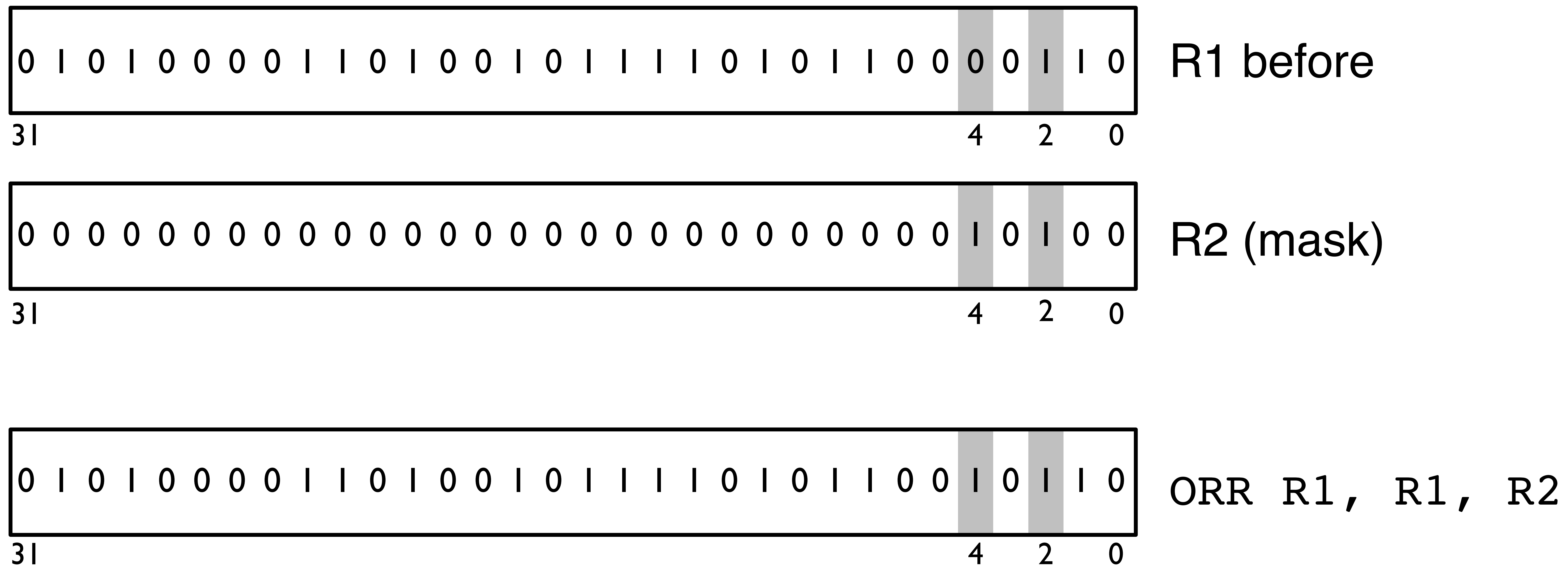
Construct a mask with 1 in the bit positions we want to set and 0 in the bit positions we want to leave unchanged



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
```

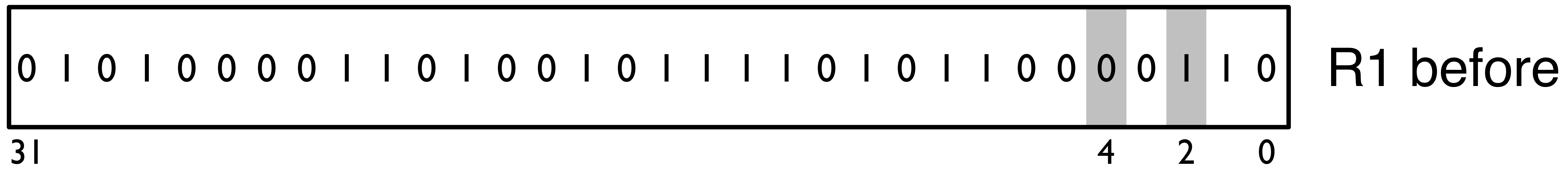
Save one instruction by specifying the mask as an immediate operand in the ORR instruction

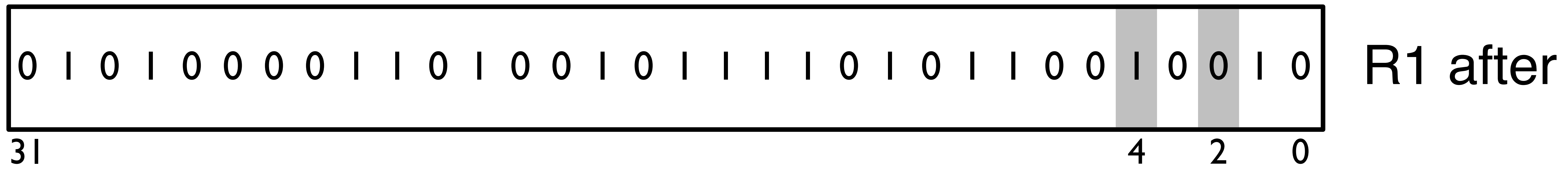
```
      ORR  r1, r1, #0x00000014; set bits 2 and 4
```

REMEMBER: 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 – Invert Bits

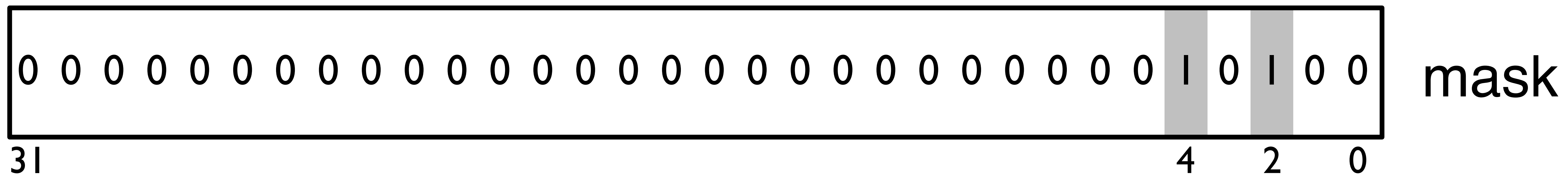
e.g. Invert bits 2 and 4 (i.e. the 3rd and 5th bits) of the value in r1





Observe 1 ⊕ x = ¬x and 0 ⊕ x = x

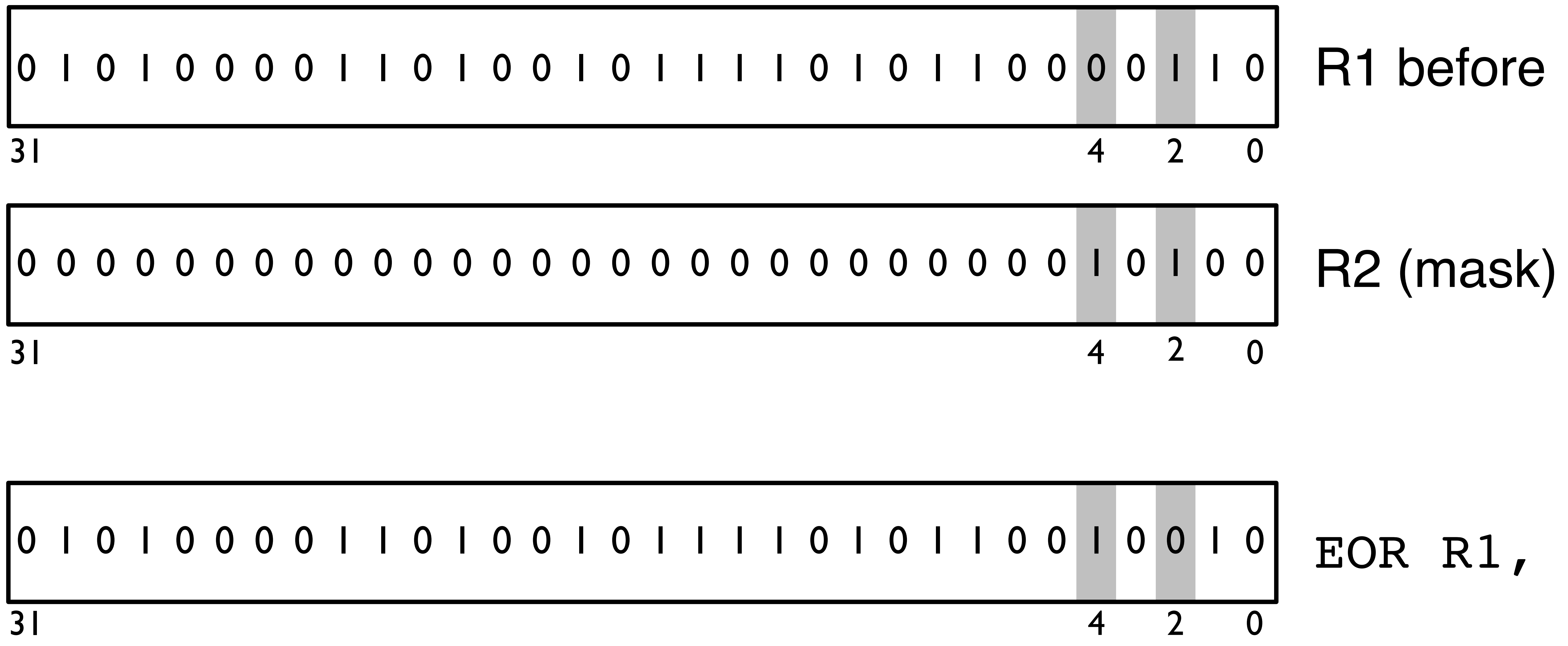
Construct a mask with 1 in the bit positions we want to invert and 0 in the bit positions we want to leave unchanged



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

Bit Manipulation – Invert Bits

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



```
EOR R1, R1, R2
```

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
```

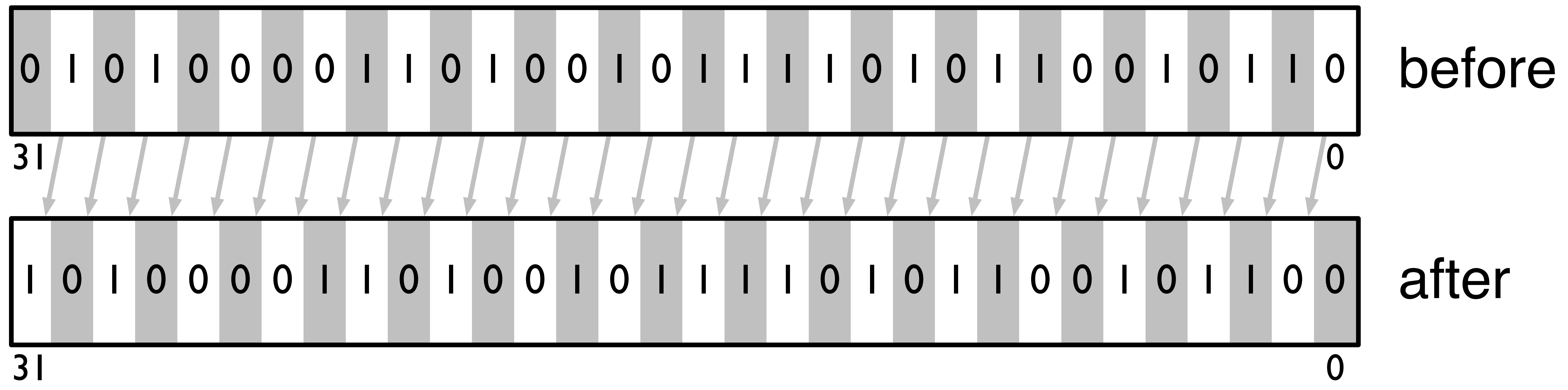
Again, can save an instruction by specifying the mask as an immediate operand in the EOR instruction

```
	EOR	r1, r1, #0x00000014	; invert bits 2 and 4
```

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

Logical Shift Left <<

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

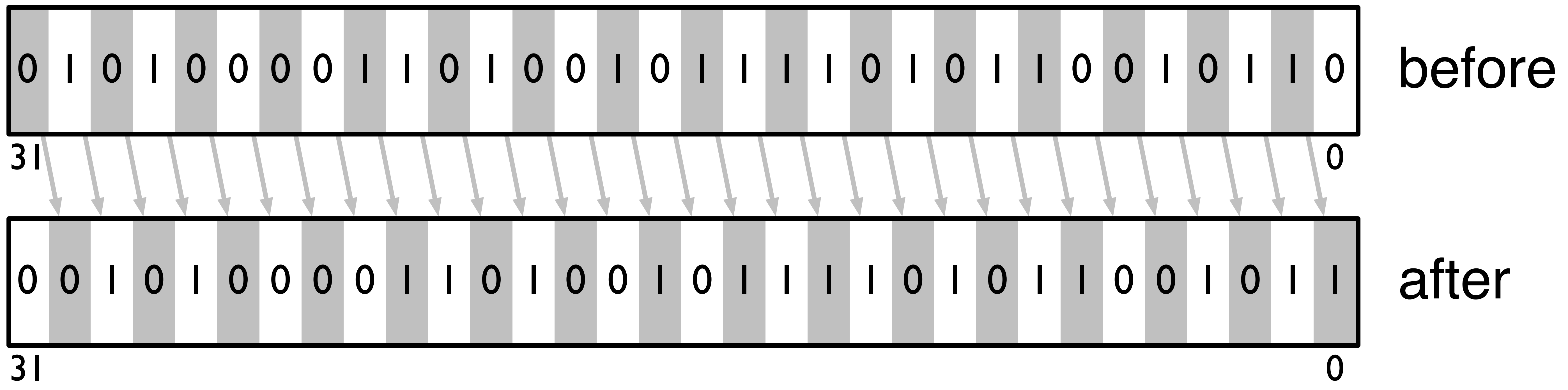
```
MOV Rd, Rm, LSL #n
```

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

Note: Bit shifting can be used for multiplication and division by 2^n

Logical Shift Right >>

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
```

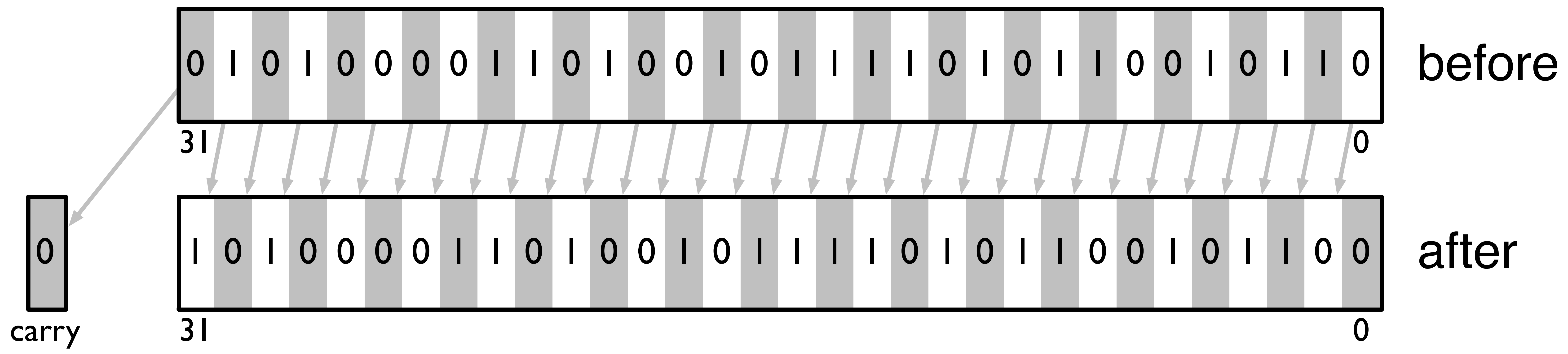
MSB of Rd is set to zero, LSB of Rm is discarded

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)

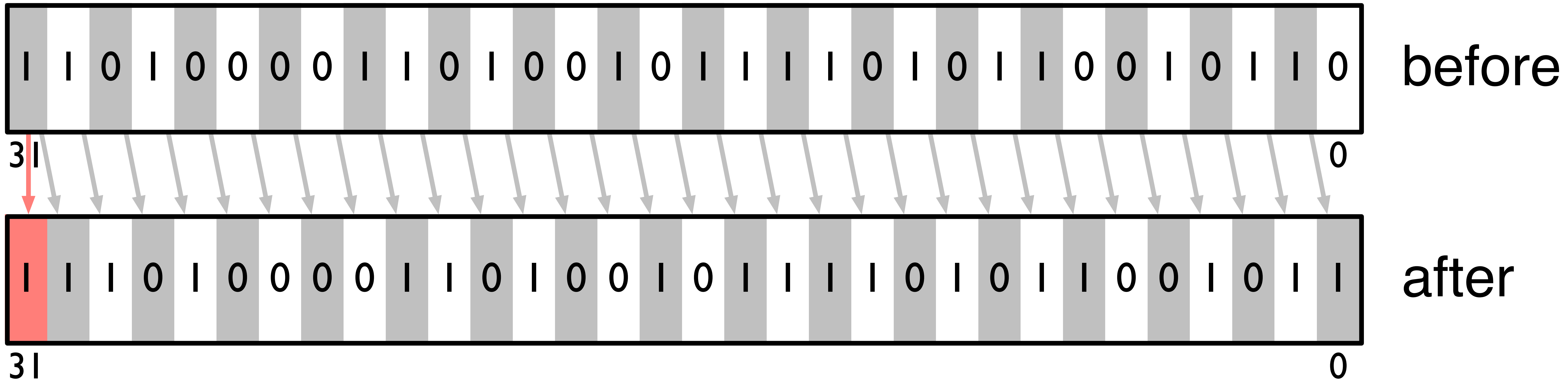


```
MOVS Rd, Rm, LSL #n

MOVS Rd, Rm, LSR #n
```

Arithmetic Shift Right

e.g. Arithmetic Shift Right by 1 bit position



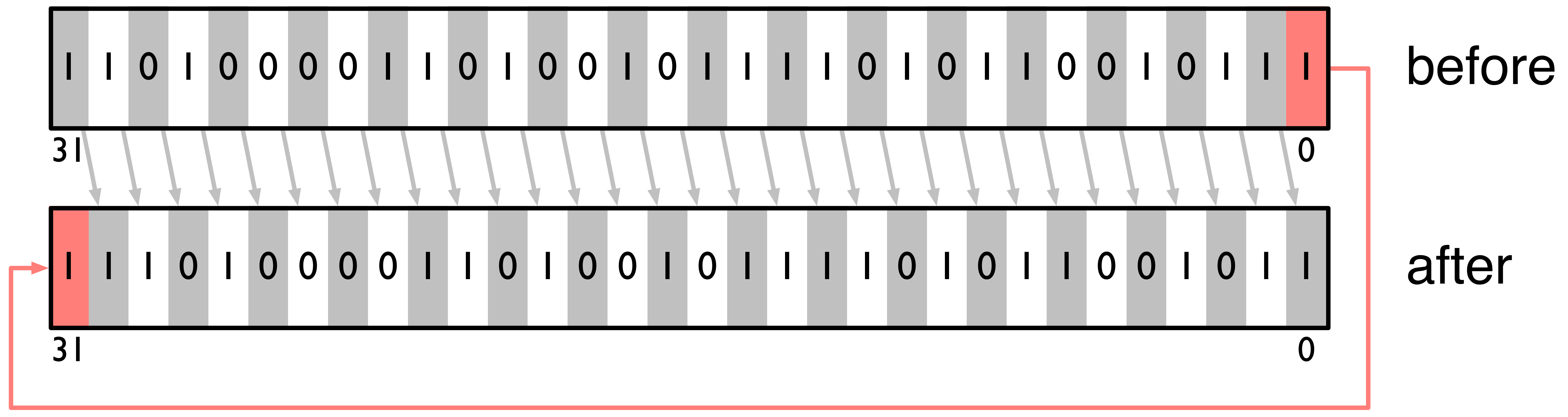
ASR shifts source operand, Rm, right by n = 0 … 31 bit positions, copying the sign (MSB) from the source to the sign (MSB) of the destination operand, Rd

```
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?