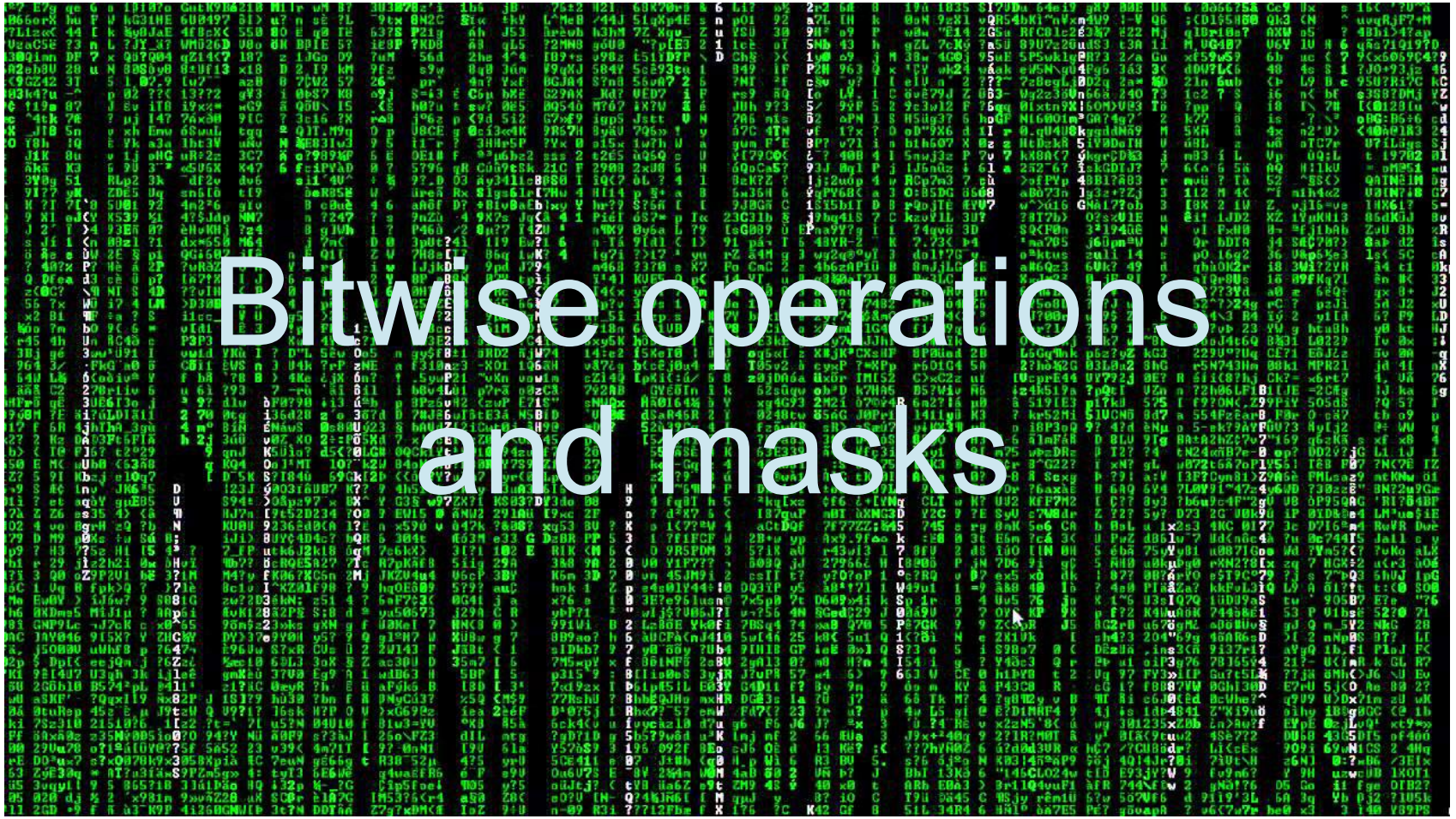
ESIGELEC
ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

C Programming



Bitwise operations and masks

Ángel Perles
angel.perles@upv.es
R02

Contents

Objectives
From Java to C
Number notation
Bitwise operators
Masks
Fields and words
Decoders

Objectives

- Write binary values in hexadecimal and read them back
- Use bitwise operators ~~and know their precedence~~
- Choose unsigned fixed-width types for bit work
- Extract, test, clear, set and toggle bits with masks
- Extract and insert multi-bit fields; build words from bytes
- Bit-focused lookup tables

From Java to C

	Java	C99
`& | ^ ~ << >>`	Same	Same
Logical shift `>>>`	Yes	No: `>>` on **unsigned** inserts zeros
Binary literal `0b1010`	Yes (Java 7)	No (GCC extension, C23)
Integer types for bits	Always signed	`uint8_t`, `uint16_t`, `uint32_t`
Shift too far or into the sign bit	Defined	Undefined behaviour
Small types promoted to `int`	Yes	Yes, and it bites more often
Print in binary	`Integer.toBinaryString()`	No `%b` in `printf()`: write your own

NUMBER NOTATION

0x

Internal and external representation

- Inside the computer: **only binary**
- In the source code: the notation you find convenient

```c
#include <stdio.h>

int main(void)
{
    int value;

    value = 'A';          /* ASCII code of 'A'      */
    value = value + 011;  /* octal 11   (9 decimal) */
    value = value + 1;    /* decimal 1              */
    value = value + 0xA;  /* hex A     (10 decimal) */

    printf("Decimal: %d\n", value);
    printf("Hex:     %X\n", value);
    printf("ASCII:   %c\n", value);

    return 0;
}
```

```
Decimal: 85
Hex:     55
ASCII:   U
```

- Four notations, one number:
- $0101\ 0101_2$
- The programmer chooses the notation, never the other way round

Hexadecimal: 4 bits per digit

Dec	Binary	Hex	Dec	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

- Group the bits in fours **from the right**
- 101 0101 0111 1010 1010 1111 1010 1010_2 = 557AAFAAh
- In this course: **3Ch** in the text, `0x3C` in the code
- 1356 ≠ 1356h
- Bytes with two digits (0Bh, not Bh); 16-bit words with four (001Ch)

Activity: convert

- $1011\ 0110_2$ = ?h
- 2Fh = $?_2$
- $1\ 0000\ 0001_2$ = ?h
- FFE3h = $?_2$



C99 literals: hex yes, binary no, octal by accident

```
uint8_t mask = 0b00111100;    /* warning: binary constants are a */
                              /* C2X feature or GCC extension    */
uint8_t mask = 0x3CU;         /* OK: 0011 1100                   */
```

```
uint16_t codes[3] = {100U, 010U, 001U};   /* 100, 8, 1 !         */
```

- Every bit constant ends in `U`
- A leading zero makes the literal **octal** (also in Java): never pad numbers with zeros
- `-pedantic` warns about `0b`; nothing warns about octal

BITWISE OPERATORS



The six bitwise operators

Operator	Name	a = 0Ch, b = 0Ah
`a & b`	AND	08h
`a | b`	OR	0Eh
`a ^ b`	XOR	06h
`~a`	NOT (complement)	F3h (as `uint8_t`)
`a << n`	Shift left n bits, zeros in	`a << 1` = 18h
`a >> n`	Shift right n bits, zeros in **if unsigned**	`a >> 2` = 03h

AND	OR	XOR
`0 & 0 = 0`	`0 | 0 = 0`	`0 ^ 0 = 0`
`0 & 1 = 0`	`0 | 1 = 1`	`0 ^ 1 = 1`
`1 & 0 = 0`	`1 | 0 = 1`	`1 ^ 0 = 1`
`1 & 1 = 1`	`1 | 1 = 1`	`1 ^ 1 = 0`

- Compound forms: `&=` `|=` `^=` `<<=` `>>=`

& is not &&, and precedence

```
uint8_t a = 0x0CU;    /* 0000 1100 */
uint8_t b = 0x03U;    /* 0000 0011 */

a & b     /* 0: no common bits     */
a && b    /* 1: both are non-zero  */
a | b     /* 15 (0Fh)              */
a || b    /* 1                     */
```

- `==` and `!=` bind **tighter** than `&`, `^` and `|`
- Rule: parentheses around every subexpression

```
if (data & BIT(2) == 0U)          /* data & (BIT(2) == 0U): always 0 */
if (0U == (data & BIT(2)))        /* OK                              */
```

- gcc: “suggest parentheses around comparison in operand of '&'” (`-Wparentheses`)

Use unsigned types for bits

```
int32_t  s = -16;
uint32_t u = 0xFFFFFFF0U;

s >> 2                  /* -4: implementation-defined          */
u >> 2                  /* 3FFFFFFCh: zeros in, always         */

int32_t  y = 1 << 31;   /* undefined behaviour, NO warning     */
uint32_t x = 1U << 31;  /* 80000000h, OK                       */

uint8_t  n = 32U;
uint32_t z = 1U << n;   /* undefined: shift >= width (1 on x86) */
```

- Bits live in `uint8_t`, `uint16_t`, `uint32_t`; constants end in `U`
- The shift count must be smaller than the width of the type

Integer promotion strikes back

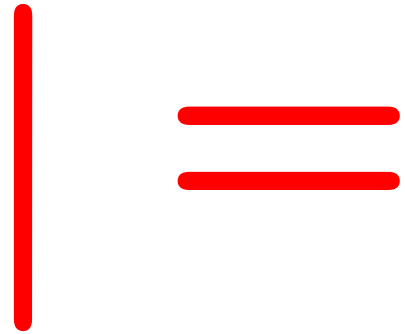
```
uint8_t flags = 0x04U;

if (~flags == 0xFBU) {        /* never true: */
    ...                       /* ~flags = -5 */
}

uint8_t inverted = ~flags;    /* FBh         */
if (0xFBU == inverted) {      /* OK          */
    ...
}
```

- Before operating, `uint8_t` and `uint16_t` become `int` (like `byte` in Java)
- `~` also inverts the high bits of the `int`
- gcc: “comparison of promoted bitwise complement of an unsigned value with constant” (`-Wsign-compare`, in `-Wextra`)
- Same with `<<`: `0x3AU << 4` is 3A0h; stored in a `uint8_t` it becomes A0h

MASKS



Masks with names

b7	b6	b5	b4	b3	b2	b1	b0
MSB							LSB

```
#define BIT(n)              (1U << (n))

#define VALVE_INLET_0       BIT(0)
#define VALVE_INLET_1       BIT(1)
#define ALARM_LAMP          BIT(7)
#define VALVES_INLET_ALL    (BIT(3) | BIT(2) | BIT(1) | BIT(0))    /* 0Fh */
```

- Bits are numbered from the right: b0 is the least significant bit
- A **mask** is a number whose 1s mark the bits you care about (or 0s)
- Avoid magic numbers: `reg |= ALARM_LAMP;`, not `reg |= 0x80;`

Extract bits: AND

- `b & 0 = 0` and `b & 1 = b`: the 1s of the mask keep, the 0s erase
- Example: keep bits 2..0 (mask 07h)

	bN	…	b5	b4	b3	b2	b1	b0
&	0	…	0	0	0	1	1	1
=	0	…	0	0	0	**b2**	**b1**	**b0**

```
#define LOW3_MASK  (BIT(2) | BIT(1) | BIT(0))   /* 07h */

low3 = data & LOW3_MASK;      /* data = B6h -> low3 = 06h */
```

Test a bit

- Extract one bit and compare the result with 0
- Example: test bit 2 (mask 04h)

	bN	…	b5	b4	b3	b2	b1	b0
&	0	…	0	0	0	1	0	0
=	0	…	0	0	0	**b2**	0	0

```
if (0U != (data & BIT(2))) {
    printf("Bit 2 is 1\n");
} else {
    printf("Bit 2 is 0\n");
}
```

- The result is 0 or 04h, **not 1**: always compare with 0

Clear bits: AND with the complement

- Build the mask with 1s in the bits to clear, then AND with its complement
- Example: clear bits 4, 3, 2 of a 16-bit word (mask 001Ch, complement FFE3h)

	b15	…	b5	b4	b3	b2	b1	b0
&	1	…	1	0	0	0	1	1
=	b15	…	b5	**0**	**0**	**0**	b1	b0

```
#define CLEAR_MASK  (BIT(4) | BIT(3) | BIT(2))    /* 001Ch */

word &= ~CLEAR_MASK;            /* FFFFh -> FFE3h */
```

- Write `~CLEAR_MASK`, not `0xFFE3U`: valid for any width, and it says which bits

Set bits: OR

- `b | 0 = b` and `b | 1 = 1`: the 1s of the mask set, the 0s keep
- Example: set bits 5, 2, 1 (mask 26h)

	bN	…	b5	b4	b3	b2	b1	b0
|	0	…	1	0	0	1	1	0
=	bN	…	**1**	b4	b3	**1**	**1**	b0

```
#define SET_MASK  (BIT(5) | BIT(2) | BIT(1))    /* 26h */

reg |= SET_MASK;                 /* 00h -> 26h */
```

Toggle bits: XOR

- `b ^ 0 = b` and `b ^ 1 = ~b`: the 1s of the mask invert, the 0s keep
- Example: toggle bits 3, 1, 0 (mask 0Bh)

	bN	…	b5	b4	b3	b2	b1	b0
^	0	…	0	0	1	0	1	1
=	bN	…	b5	b4	**~b3**	b2	**~b1**	**~b0**

```
#define TOGGLE_MASK  (BIT(3) | BIT(1) | BIT(0))    /* 0Bh */

reg ^= TOGGLE_MASK;              /* 00h -> 0Bh */
reg ^= TOGGLE_MASK;              /* 0Bh -> 00h */
```

- The same XOR twice restores the original value

Mask recipes at a glance

Goal	Operator	The mask has 1s in…	C
Extract	AND	the bits to keep	`x = reg & MASK;`
Test	AND, compare with 0	the bit to test	`if (0U != (reg & MASK))`
Clear	AND with ~	the bits to clear	`reg &= ~MASK;`
Set	OR	the bits to set	`reg |= MASK;`
Toggle	XOR	the bits to toggle	`reg ^= MASK;`

Activity: what is reg now?

```
uint8_t reg = 0x5AU;

reg |= BIT(0);          /* 1 */
reg &= ~BIT(6);         /* 2 */
reg ^= 0xF0U;           /* 3 */
reg = reg >> 2U;        /* 4 */
reg <<= 4U;             /* 5 */
```

- Work in binary: 5Ah = 0101 1010_2
- Give the value after each line in hexadecimal



Activity: find the four bugs

```
#define VALVE_MASK  0x04U

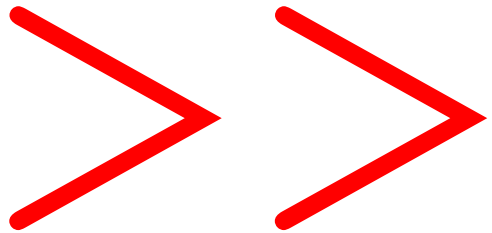
uint8_t  valves = read_valves();
uint8_t  alarm  = read_alarm();
uint32_t error_flags = 0U;

if (valves & VALVE_MASK == 0U) {      /* 1 */
    printf("Valve 2 closed\n");
}
if (valves && VALVE_MASK) {           /* 2 */
    printf("Valve 2 open\n");
}
if (~alarm == 0xFEU) {                /* 3 */
    printf("Alarm off\n");
}
error_flags |= 1 << 31;               /* 4 */
```

- Which ones does gcc report?
- Fix them in the course style



FIELDS AND WORDS



Multi-bit fields: extract and insert

- A 4-bit field in bits 5..2 (mask 3Ch)

	b7	b6	b5	b4	b3	b2	b1	b0
>> 2	0	0	b7	b6	b5	b4	b3	b2
& 0Fh	0	0	0	0	**b5**	**b4**	**b3**	**b2**

```
#define FIELD_POS   2U
#define FIELD_MASK  (0x0FU << FIELD_POS)          /* 3Ch */

/* extract: two equivalent ways */
field = (data >> FIELD_POS) & 0x0FU;
field = (data & FIELD_MASK) >> FIELD_POS;

/* insert: clear the field, then OR the new value */
reg = (reg & ~FIELD_MASK) | ((value << FIELD_POS) & FIELD_MASK);
```

Compose and decompose words

```
uint16_t raw = 0x04B0U;                          /* 1200, Lab strings */

uint8_t high = (uint8_t)(raw >> 8U);             /* 04h  */
uint8_t low  = (uint8_t)(raw & 0xFFU);           /* B0h  */

uint16_t back = (uint16_t)((high << 8U) | low);  /* 04B0h */
```

- Buses and peripheral registers carry **bytes**: 16 and 32-bit values are split and rebuilt like this
- Shifts and masks do not depend on the byte order in memory (endianness); `memcpy()` and pointer casts do

Activity: print_binary()

- `printf()` has no binary format in C99
- Write `void print_binary(uint8_t value)` that prints `10110110` for B6h
- Hint: a mask that starts at 80h and moves to the right

```
print_binary(0xB6U);
print_binary(0x05U);
```

```
10110110
00000101
```

BITWISE LOOKUP TABLES



A 3-to-8 decoder in C

CBA	Y7..Y0	Hex
000	00000001	01h
001	00000010	02h
010	00000100	04h
011	00001000	08h
100	00010000	10h
101	00100000	20h
110	01000000	40h
111	10000000	80h

```c
/* computed */
static uint8_t deco3x8(uint8_t input)
{
    return (uint8_t)(1U << (input & 0x07U));
}
```

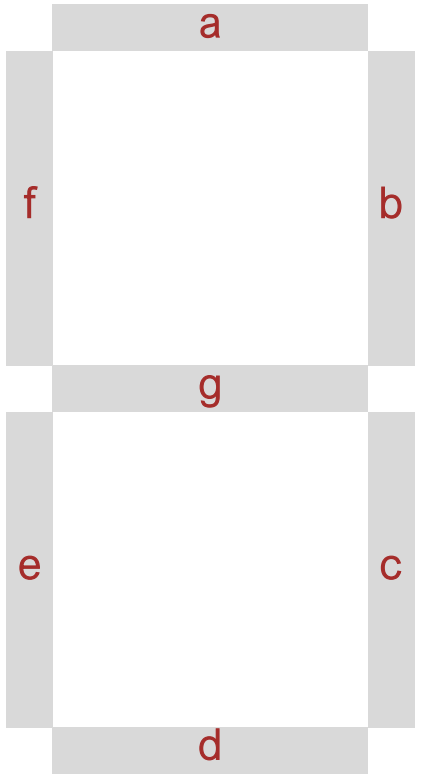
```c
/* lookup table: index = input */
static const uint8_t deco3x8_table[8] = {
    0x01U, 0x02U, 0x04U, 0x08U,
    0x10U, 0x20U, 0x40U, 0x80U
};

output = deco3x8_table[input & 0x07U];
```

- `& 0x07U` keeps the index inside the array: C does not check bounds

A 7-segment display

- A `const` array implements any combinational function:
  index = inputs, content = outputs
- Example: 7-segment display, segment a = bit 0 … g = bit 6




```
static const uint8_t seg7_table[10] = {
    0x3FU, 0x06U, 0x5BU, 0x4FU, 0x66U,
    0x6DU, 0x7DU, 0x07U, 0x7FU, 0x6FU
};
```

- Activity: which segments are on for the digit 2? And for 7?

ESIGELEC

ÉCOLE D'INGÉNIEURS-ES GÉNÉRALISTES

UNIVERSITAT POLITÈCNICA DE VALÈNCIA