NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

Low-level Systems Programming

- Conditional inclusion/compilation
- Standard integer types
- Bit-wise operators
- Bit-fields

The contents in this lecture slides are not going to be included in the test/exam.

Conditional Inclusion / Compilation

Recall from Week 1:

- Pre-processor directives (begins with #) are instructions to the pre-processor for performing
  - File inclusion
  - Macro substitution
  - *Conditional inclusion*

Conditional Inclusion (Conditional Compilation):
Capability provided by the preprocessor for selecting lines of code that will be compiled and ignored

Conditional Inclusion Directives

- Six directives can be used to control conditional compilation
  - Beginning of block:

```
#if
```

```
#ifdef
```

```
#ifndef
```

  - Optional, alternative block:

```
#else
```

```
#elif
```

  - End of block:

```
#endif
```

Conditional Inclusion Operator

- In addition to the six directives, an operator is also available

```
defined name
```

```
defined (name)
```

  - Evaluates to 1 if ***name*** is defined, otherwise it evaluates to 0

Example using #ifdef and #endif

```
#ifdef MACRO

/**
 * Code here will be compiled
 * if MACRO is defined previously
 */

#endif /* MACRO */
```

Example using #ifndef and #endif

```
#ifndef MACRO

/**
 * Code here will be compiled
 * if MACRO is not defined previously
 */

#endif /* MACRO */
```

Example using #if, #elif, #else and #endif

```
#if defined(linux)

/* Code here will be compiled if linux is defined */

#elif defined(_WIN32)

/* Code here will be compiled if _WIN32 is defined */

#else

/* Code here will be compiled if neither linux or _WIN32
   is defined */

#endif
```

Where to define a macro

- Within a header or source file
  - Using `#define` directive
- Within a `Makefile`
  - To be discussed in “Writing Large Program” Tutorial on Friday
- As a command-line option to `gcc` or `g++`
  - `gcc –DMACRO_NAME source.c`: `MACRO_NAME` will be defined in `source.c`
  - `g++ -DMACRO_NAME source.cc`: `MACRO_NAME` will be defined in `source.cc`

Example

hello.c

```c
#include <stdio.h>

int main(void)
{
#ifdef HELLO
    printf("hello\n");
#else
    printf("world\n");
#endif
    return 0;
}
```

```
$ gcc -DHELLO hello.c
$ ./a.out
hello

$ gcc hello.c
$ ./a.out
world
```

Recall: Basic C/C++ Data Types

Data Type	Size (bytes)
~~`boolean`~~	~~1~~
~~`byte`~~	~~1~~
`char`	~~2~~ 1
`short (short int)`	~~2~~ Machine-dependent
`int`	~~4~~ Machine-dependent
`long (long int)`	~~8~~ Machine-dependent
`long long (long long int)`	Machine-dependent
`float`	~~4~~ Machine-dependent
`double`	~~8~~ Machine-dependent
`long double`	10

Integral types (char through long long)

Float types (float, double, long double)

Integer Types

- A major problem in systems programming is the uncertainty of integer types
  - Size of an integer depends on the CPU architecture

- There are situations where you will need to specify the size precisely

Fortunately, C99 defines integer types with precise sizes

Fixed Size Integer Types

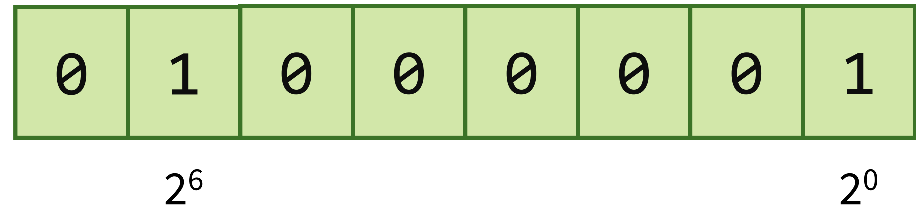
- Defined in `stdint.h` (C) and `cstdint` (C++)

Type	Sign and size
`int8_t`	8-bit signed integer
`int16_t`	16-bit signed integer
`int32_t`	32-bit signed integer
`int64_t`	64-bit signed integer
`uint8_t`	8-bit unsigned integer
`uint16_t`	16-bit unsigned integer
`uint32_t`	32-bit unsigned integer
`uint64_t`	64-bit unsigned integer

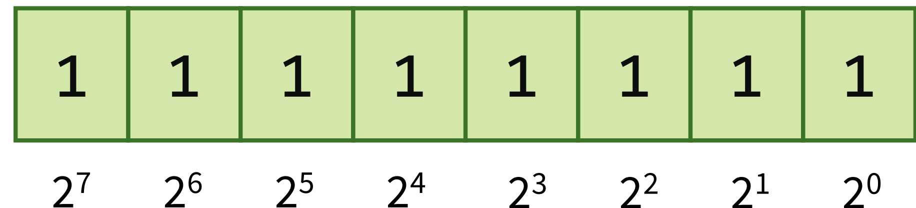
Binary Number Representation

- How are numbers actually stored in variables?
  - Numbers are stored using binary representation
  - See https://www.bottomupcs.com/chapter01.xhtml for details

```
char c = 'A'; //65
```




```
uint8_t b = 255;
```



Binary Number Representation

How are negative numbers represented?

- Use **two’s complement**

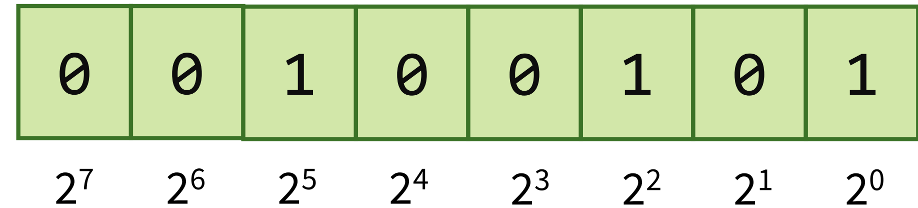
Two’s Complement Operation:

1. Obtain binary representation disregarding the sign
2. If sign is positive, done
3. Otherwise:
    1. Invert all bits of the binary representation
    2. Add 1 to the binary representation

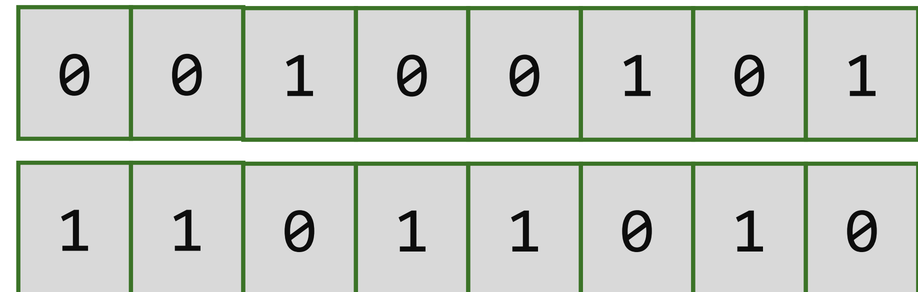
Binary Number Representation

- Example:

```
int8_t a = 37;
```

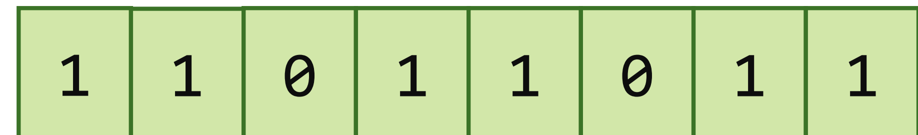



```
int8_t b = -37;
```



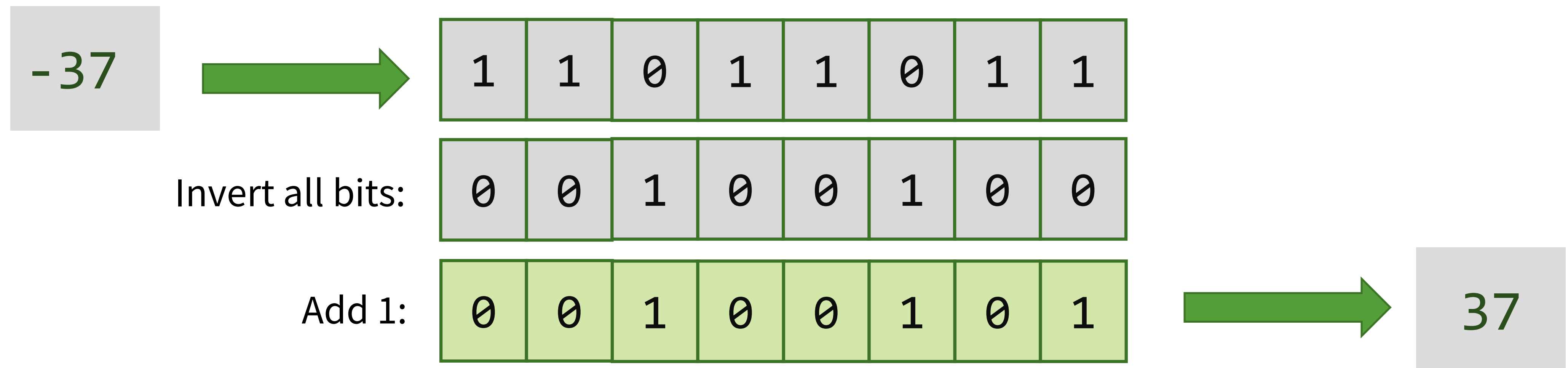
Invert all bits:

Add 1:



Binary Number Representation

- When you get the two’s complement of a negative number, you will obtain the positive number



When to use unsigned types?

- Passing and comparing signed and unsigned values is a common pitfall in C/C++ programming

```
#include <stdio.h>
int array[] = {1, 2, 3};
void dump(unsigned int len)
{
	for(unsigned int i=0; i<len; i++)
		printf("array[%d]: %d\n", i, array[i]);
}
int main(void)
{
	dump(-1);
	return 0;
}
```

Will be treated as a very large positive integer by dump()

When to use unsigned types?

- If possible, avoid using unsigned types

- Areas where unsigned types are used:
  - When dealing with bit values
  - Performing bit-level operations

Endianness

- Consider a 32-bit (4-byte) integer:

```
uint32_t a = 305419896;
```

```
00010010001101000101011001111000
```

- How is it actually stored in computer memory?

There are two ways of storing multi-byte numbers in memory:

- **Big Endian**
- **Little Endian**

Endianness

00010010	00110100	01010110	01111000

Most significant byte (MSB)

Least significant byte (LSB

- Big Endian – Store MSB in lower address
- Little Endian – Store LSB in lower address

Endianness

- Consider a 32-bit (4-byte) integer:

```
uint32_t a = 305419896;
```

`00010010 00110100 01010110 01111000`

- Suppose a is at address `100`

Big Endian:

Address	Byte
100	00010010
101	00110100
102	01010110
103	01111000

Little Endian:

Address	Byte
100	01111000
101	01010110
102	00110100
103	00010010

Bit-wise Operators

& – AND

- Result is **1** if both operand bits are **1**

| – OR

- Result is **1** if either operand bit is **1**

^ – Exclusive OR

- Result is **1** if operand bits are different

~ – Complement

- Each bit is reversed

<< – Shift left

- Multiply by 2

>> – Shift right

- Divide by 2

Example: AND

```
uint8_t a = 37;

uint8_t b = 98;

uint8_t c = a & b;
```

0	0	1	0	0	1	0	1
0	1	1	0	0	0	1	0
0	0	1	0	0	0	0	0

Example: OR

```
uint8_t a = 37;

uint8_t b = 98;

uint8_t c = a | b;
```

0	0	1	0	0	1	0	1

0	1	1	0	0	0	1	0

0	1	1	0	0	1	1	1

Example: Exclusive OR

```
uint8_t a = 37;

uint8_t b = 98;

uint8_t c = a | b;
```

0	0	1	0	0	1	0	1
0	1	1	0	0	0	1	0
0	1	0	0	0	1	1	1

Example: Complement

```
uint8_t a = 37;

uint8_t b = ~a;
```

0	0	1	0	0	1	0	1

1	1	0	1	1	0	1	0

Example: Shift Left

```
uint8_t a = 37;

uint8_t b = a << 1;

uint8_t c = a << 2;
```

0	0	1	0	0	1	0	1
0	1	0	0	1	0	1	0
1	0	0	1	0	1	0	0

What is b and c in decimal?

Example: Shift Right

```
uint8_t a = 37;

uint8_t b = a >> 1;

uint8_t c = a >> 2;
```

0	0	1	0	0	1	0	1
0	0	0	1	0	0	1	0
0	0	0	0	1	0	0	1

What is b and c in decimal?

Accessing bits or groups of bits

Two Approaches:

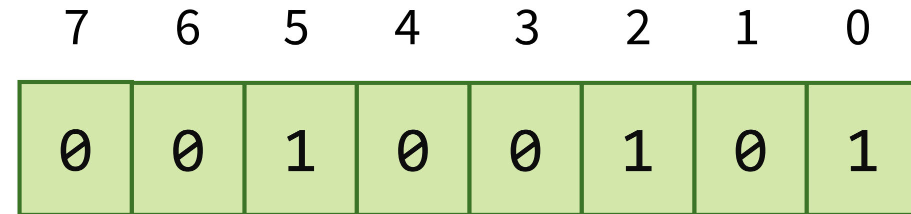
Traditional C:
Use #define macro in tandem with bitwise operators

Modern:
Use bit-fields

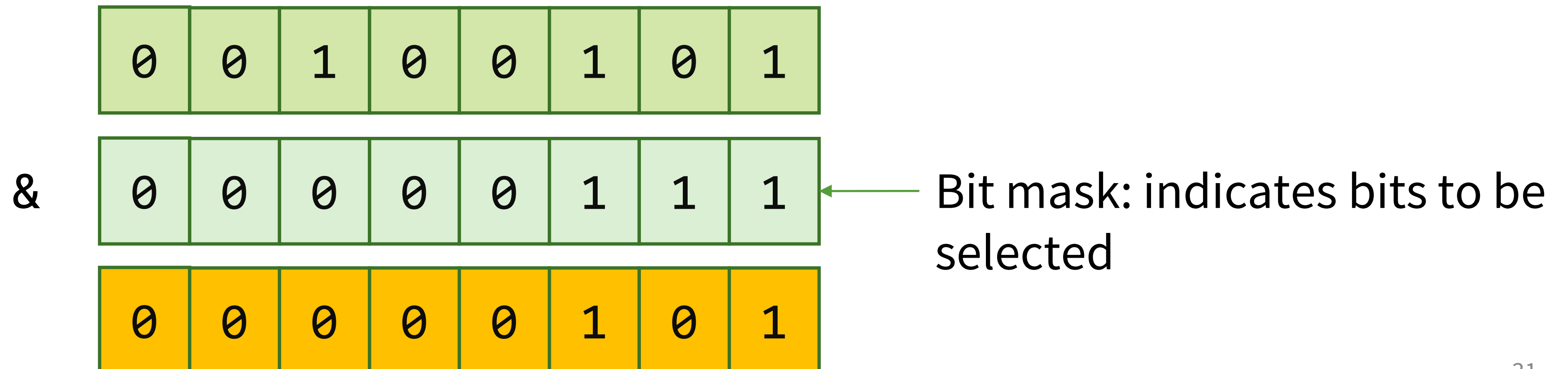
Traditional approach: selecting bits

Suppose

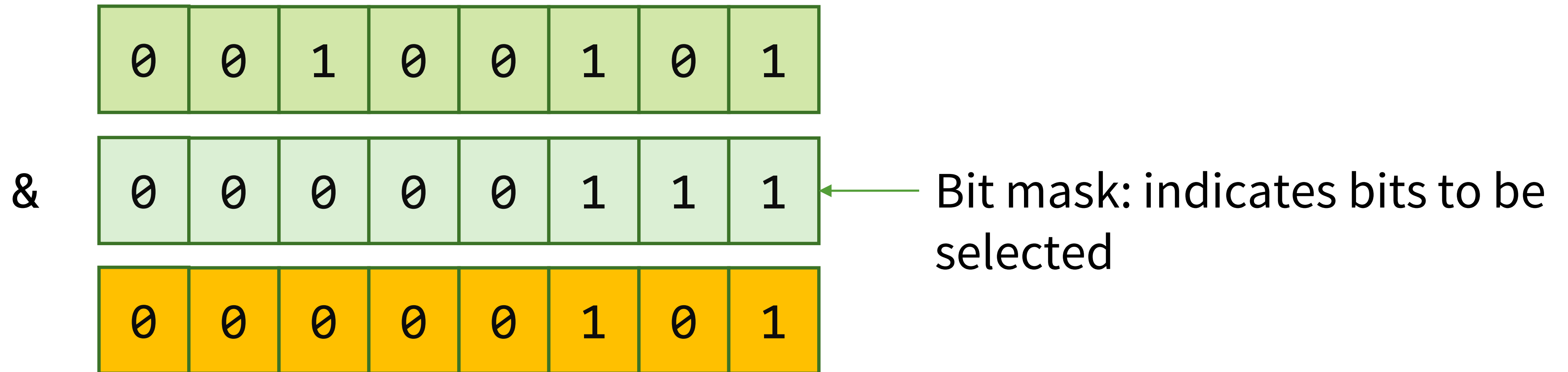
```
uint8_t a = 37;
```



How to select / read the lowest 3 bits (bits 0, 1 and 2)?



Traditional approach: selecting bits



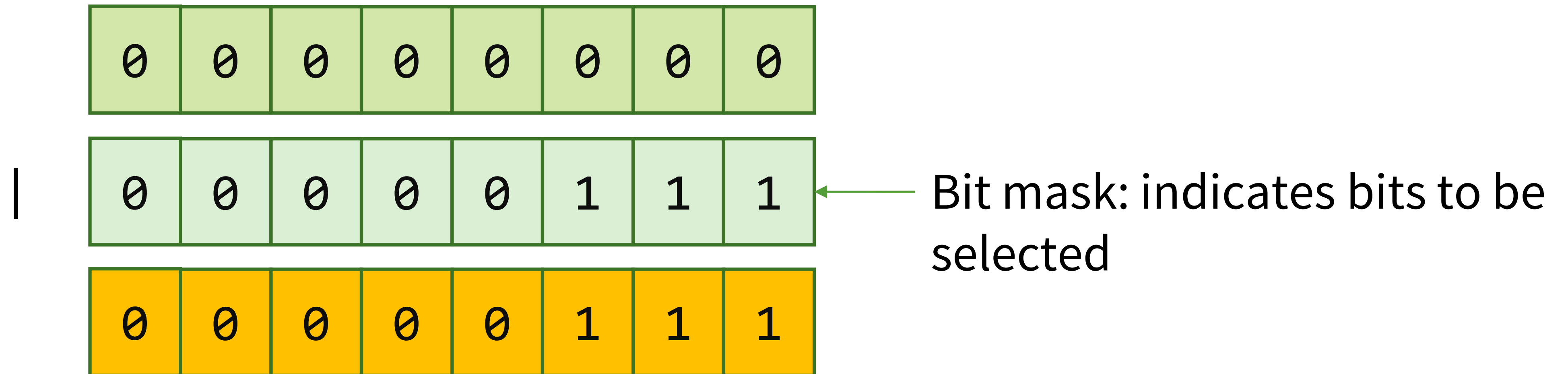

```
uint8_t a = 37;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a & mask;
```

Binary to hex

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

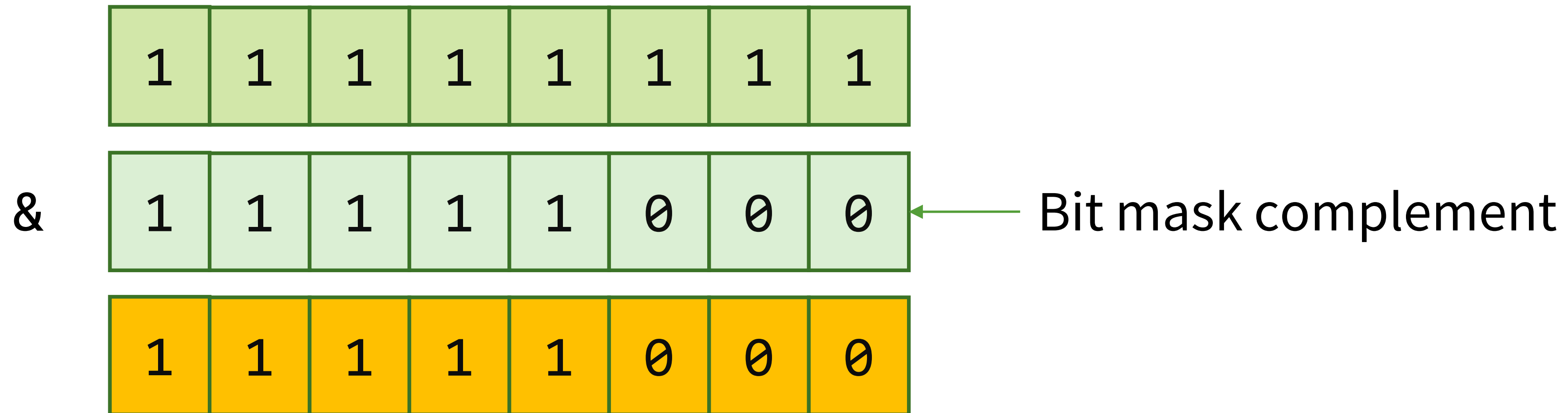
Binary	Hex
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Traditional approach: setting bits




```
uint8_t a = 0;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a | mask;
```

Traditional approach: clearing bits




```
uint8_t a = 255;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a & ~mask;
```

Traditional approach example

- Serial port line register is an 8-bit read-only register:

Bit 7: FIFO Error
Bit 6: Empty Data Holding Registers
Bit 5: Empty Transmitter Holding Registers
Bit 4: Break Interrupt
Bit 3: Framing Error
Bit 2: Parity Error
Bit 1: Overrun Error
Bit 0: Data Ready

- When bit is set (1), the condition exists

Traditional approach example

```
#define FIFO_ERROR        0x80 // bit 7 mask
#define EMPTY_DHR         0x40 // bit 6 mask
#define EMPTY_THR         0x20 // bit 5 mask
#define BREAK_INTR        0x10 // bit 4 mask
#define FRAMING_ERROR     0x08 // bit 3 mask
#define PARITY_ERROR      0x04 // bit 2 mask
#define OVERRUN_ERROR     0x02 // bit 1 mask
#define DATA_READY        0x01 // bit 0 mask

void foo(void)
{
      uint8_t reg = read_line_register();
      if (reg & DATA_READY) {
            // Data is ready
      }
}
```

Traditional approach

- Traditional approach works very well for bit-wise access
- For multi-bit access, use **bit-fields**

A **bit-field** is a data structure that allows access and/or operation of individual bits or group of bits of a word

Bit-fields in C/C++

- Declared as a struct
  - Each member is a bit-field within a word
  - Accessed like members of a struct
  - Fields may be named or un-named

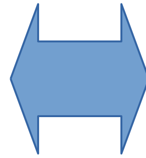
```
struct structure_tag {
      typeA memberA1 : bit_widthA1;
      typeA memberA2 : bit_widthA2;
      ...
      typeB memberB1 : bit_widthB1;
      typeB memberB2 : bit_widthB2;
      ...
} variable_list;
```

Bit-fields example (serial port line reg.)

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};

void foo(void)
{
        struct sp_line_reg reg = read_line_register();
        if (reg.data_ready) {
                // Data is ready
        }
}
```

Bit-fields example (serial port line reg.)

```
struct sp_line_reg {
    uint8_t fifo_error    : 1;
    uint8_t empty_dhr     : 1;
    uint8_t empty_thr     : 1;
    uint8_t break_intr    : 1;
    uint8_t framing_error: 1;
    uint8_t parity_error : 1;
    uint8_t overrun_error: 1;
    uint8_t data_ready    : 1;
};
```



```
struct sp_line_reg {
    uint8_t fifo_error    : 1,
            empty_dhr     : 1,
            empty_thr     : 1,
            break_intr    : 1,
            framing_error: 1,
            parity_error : 1,
            overrun_error: 1,
            data_ready    : 1;
};
```

Bit-fields example (vs traditional)

```c
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};

void foo(void)
{
    struct sp_line_reg reg =
        read_line_register();
    if (reg.data_ready) {
        // Data is ready
    }
}
```

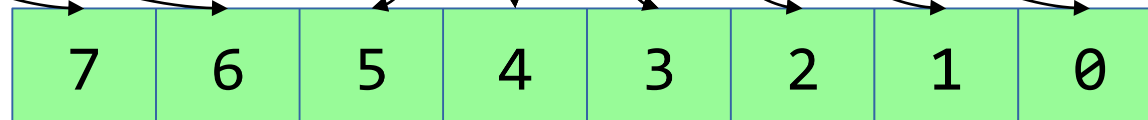
```c
#define FIFO_ERROR       0x80 // bit 7 mask
#define EMPTY_DHR        0x40 // bit 6 mask
#define EMPTY_THR        0x20 // bit 5 mask
#define BREAK_INTR       0x10 // bit 4 mask
#define FRAMING_ERROR    0x08 // bit 3 mask
#define PARITY_ERROR     0x04 // bit 2 mask
#define OVERRUN_ERROR    0x02 // bit 1 mask
#define DATA_READY       0x01 // bit 0 mask

void foo(void)
{
        uint8_t reg =
            read_line_register();
        if (reg & DATA_READY) {
                // Data is ready
        }
}
```

Problems with bit-fields

- The actual arrangement of the bits on memory depends on the compiler and/or “endian-ness” of the CPU
- **Bit-fields are not portable**

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};
```

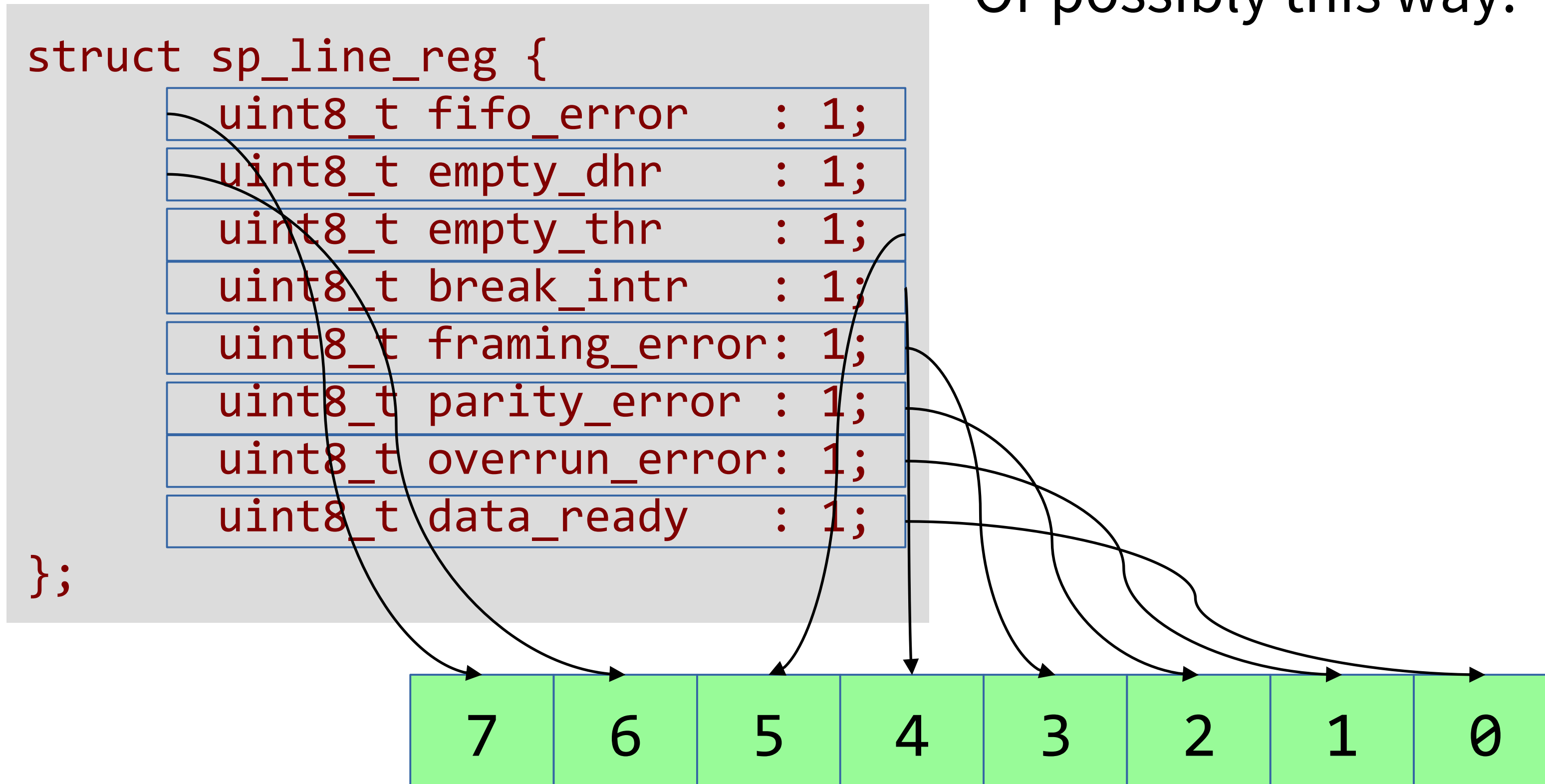
Can be mapped in memory this way:



Problems with bit-fields

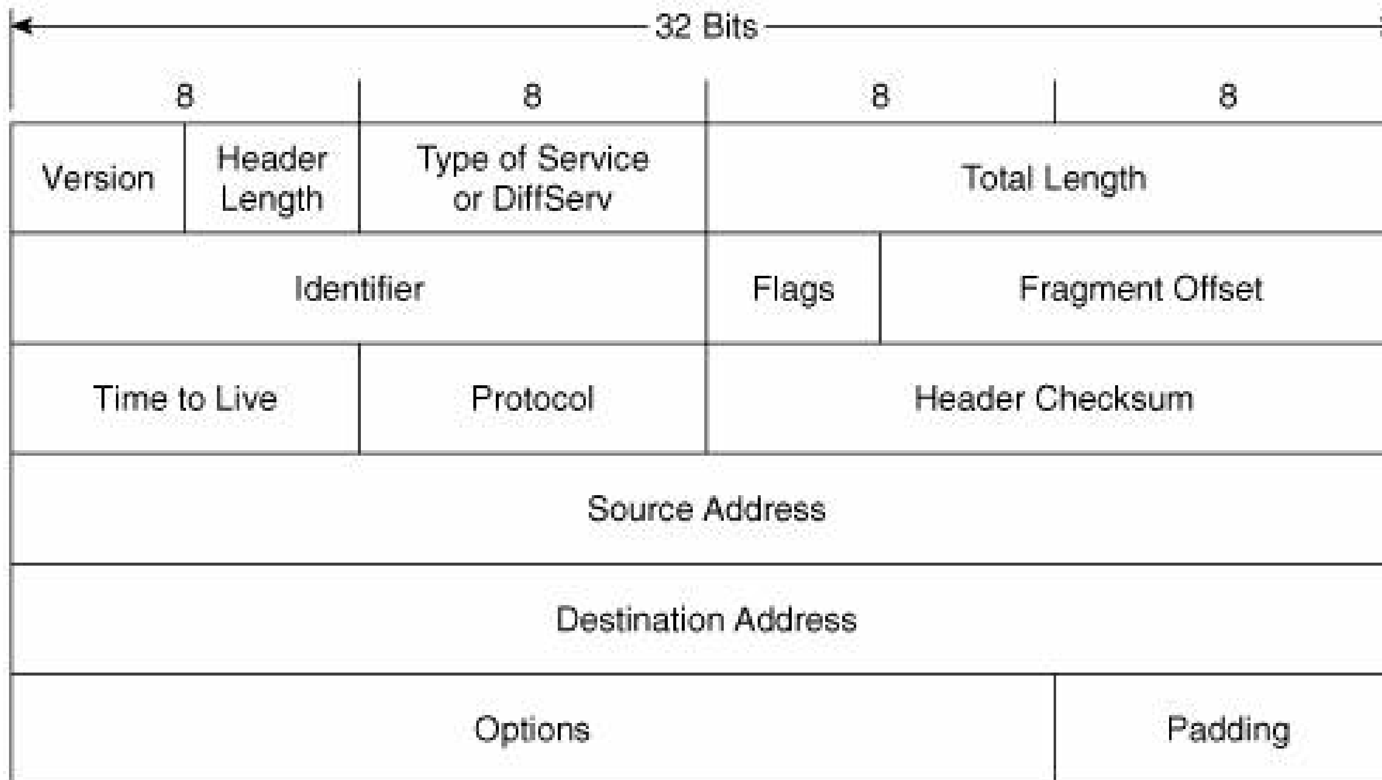
- The actual arrangement of the bits on memory depends on the compiler and/or "endian-ness" of the CPU
- **Bit-fields are not portable**

Or possibly this way:

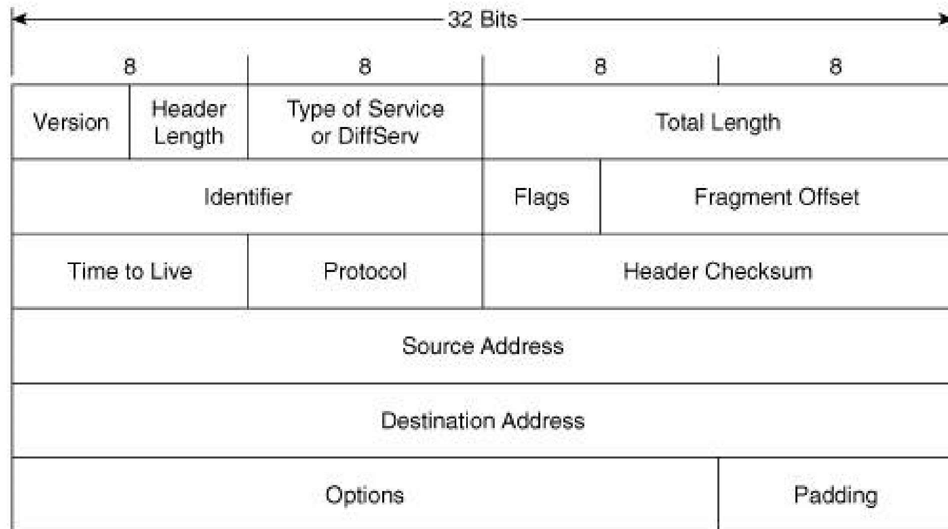


Bit-fields example: IPv4 packet header

Internet Protocol version 4 packet header format:



Bit-fields example: IPv4 packet header




```
struct iphdr {
    #if defined(__LITTLE_ENDIAN_BITFIELD)
        uint8_t ihl    :4,
                version:4;
    #elif defined (__BIG_ENDIAN_BITFIELD)
        uint8_t version:4,
                ihl    :4;
    #else
        #error  "Please fix <asm/byteorder.h>"
    #endif
         uint8_t   tos;
         uint16_t  tot_len;
         uint16_t  id;
         uint16_t  frag_off;
         uint8_t   ttl;
         uint8_t   protocol;
         uint16_t  check;
         uint32_t  saddr;
         uint32_t  daddr;
};
```

Bit-fields example: IPv4 packet header

```
void ipv4_receive(void *pkt, uint32_t my_addr)
{
    struct iphdr *iph = (struct iphdr *) pkt;

    if(iph->version != 4) {
      // Incorrect version, return
      return;
    }

    if(iph->daddr == my_addr) {
      // This packet is for me
      // Do stuff to receive it!
    }
}
```