EITF70

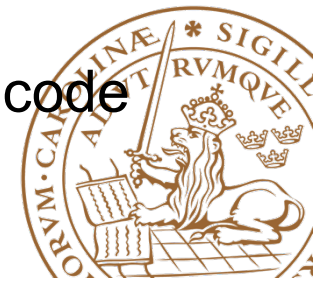
Computer Organization

Course Organization

- Course homepage:
  - http://www.eit.lth.se/kurs/eitf70
- Lab web-pages:
  - http://www.eit.lth.se/index.phpciuid=1161&coursepage=6814
  - The labs are mandatory part of the course
- Students work in groups (**2 students max**)
- The students in a group should equally contribute when solving the lab assignments
- Labs can be done at **any point in time**
- Demonstration and help from teaching assistants **during scheduled lab sessions or lessons**

Labs Organization

- The labs constitute of four laboratory exercises:
  - Lab1: Introduction to the lab environment
  - Lab2: I/O handling
  - Lab3: Machine language and assembly programming
  - Lab4: Interrupt handling
- **One project** throughout the labs
- Each lab exercise consists of a set of assignments along with a short background on the topics that are discussed
- The assignments are constructed following the “learning through examples” principle
  - Few assignments with provided code examples
  - Few assignments where students should write the code



Lab preparation

- A lesson will be given before each lab assignment
- Lessons are available on the course web-page
- Read the lab manuals and the lessons before you start working on the labs
- Recommendation:
  - Use the lessons to put questions on things that might be troublesome
  - Use the scheduled lab sessions for demonstration

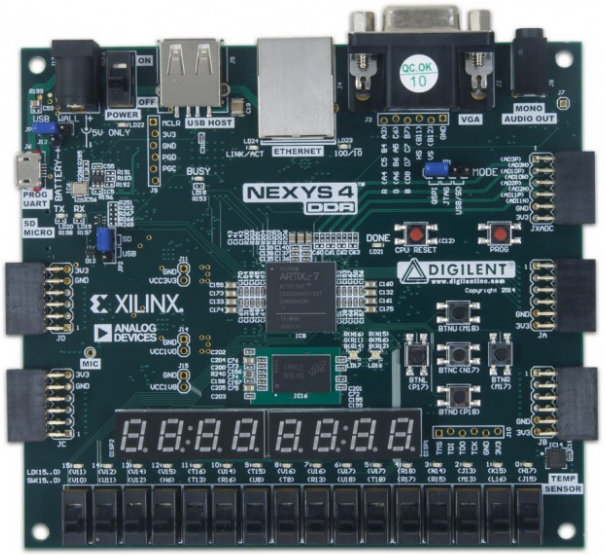
Lab reporting

- Demonstration is done after completing each laboratory exercise
- Keep the codes for all assignments
- Answer to all the questions for each of the assignments
- Demonstrate the codes for the assignments where you are asked to write a new program or modify an existing program

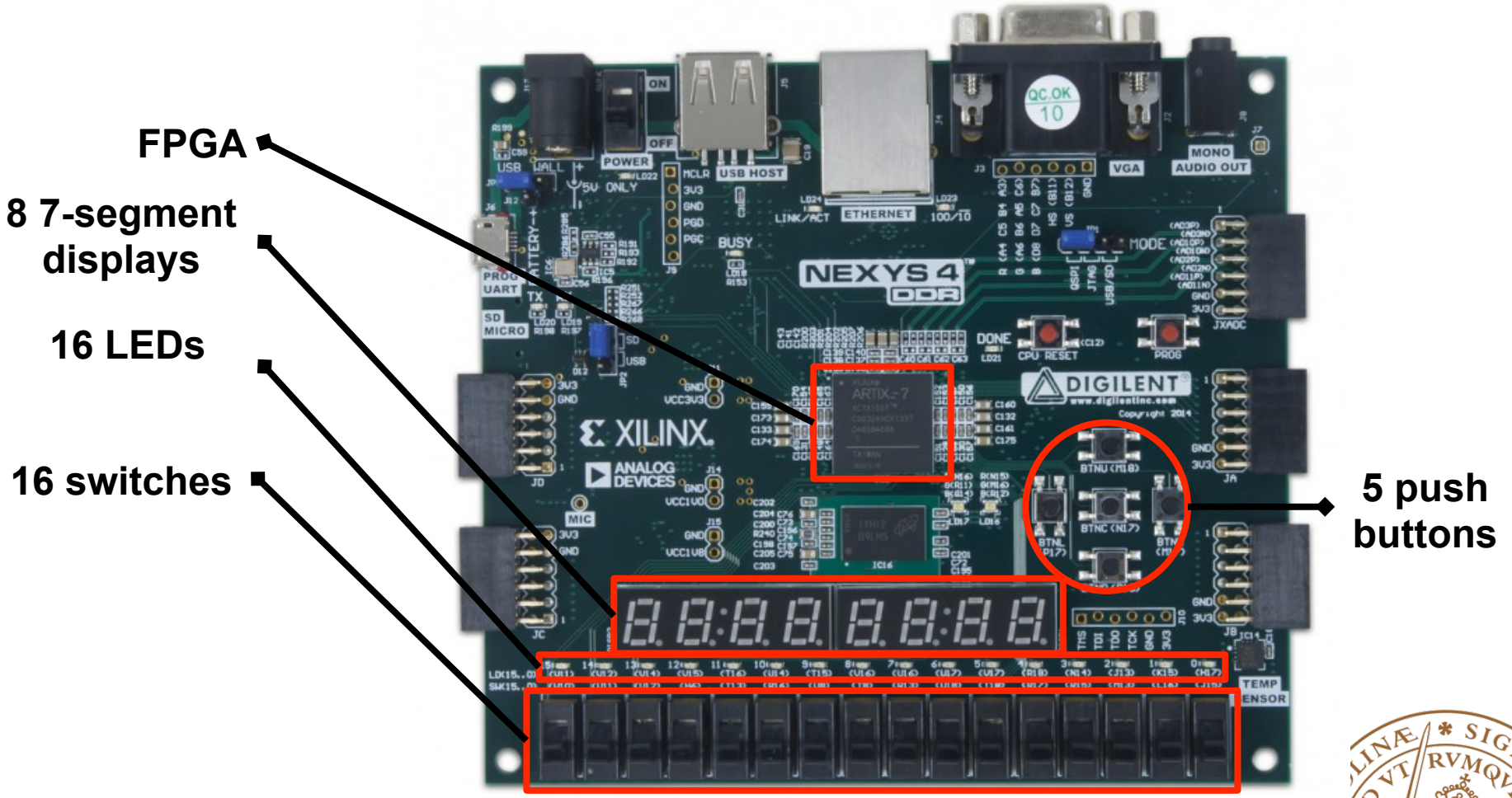
Lab environment

- Software: **Xilinx SDK**

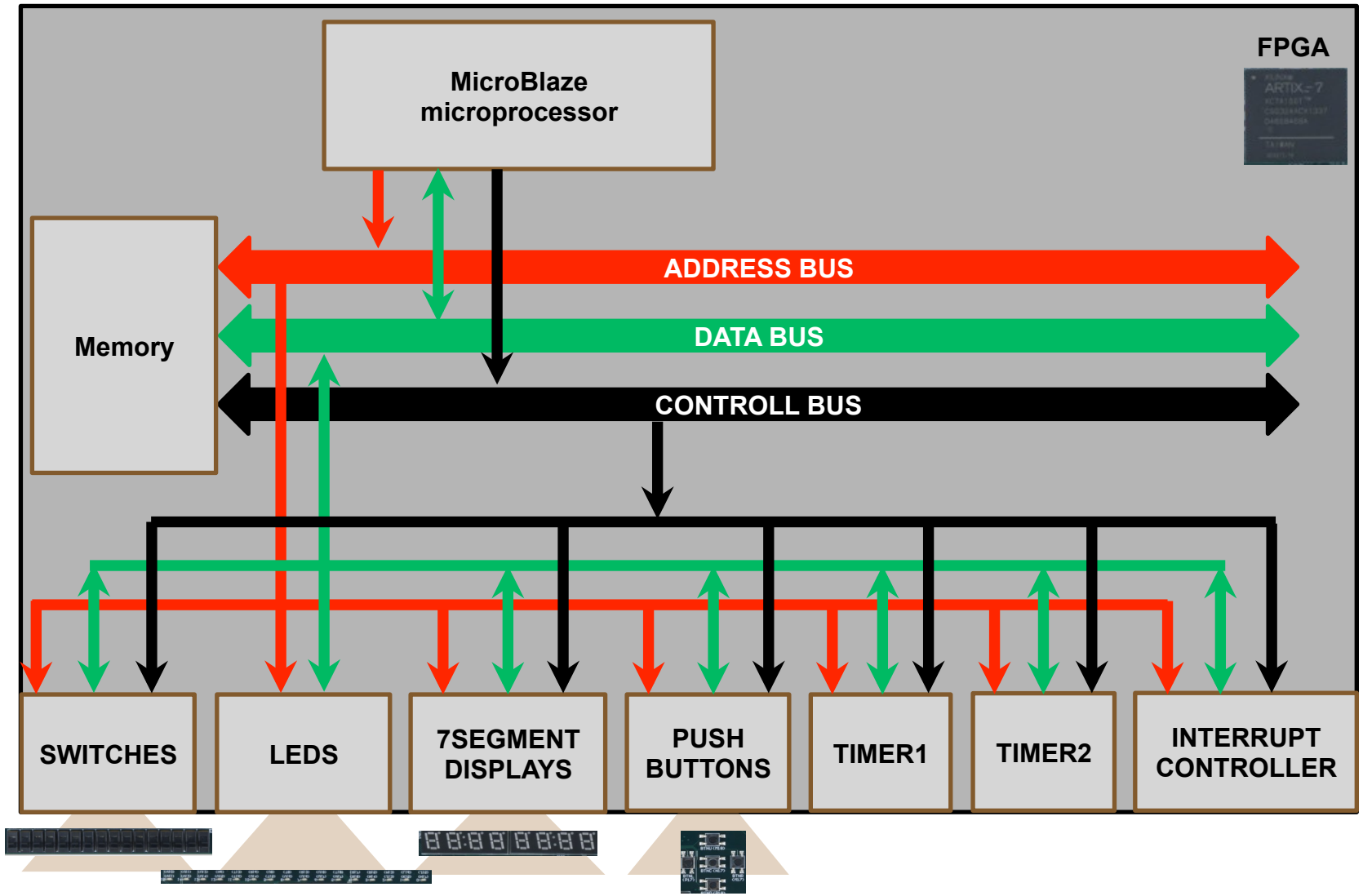
- Hardware: **Xilinx Nexys 4 board**



The Nexys 4 board



Hardware platform



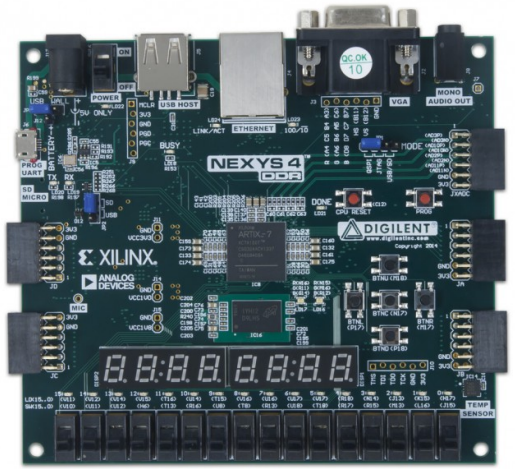
Lab1: Introduction to the lab environment

Goal

- Learn how to setup the project
- Understand the design flow
- Write applications in C
- Different data types and how they are stored in memory

The design flow

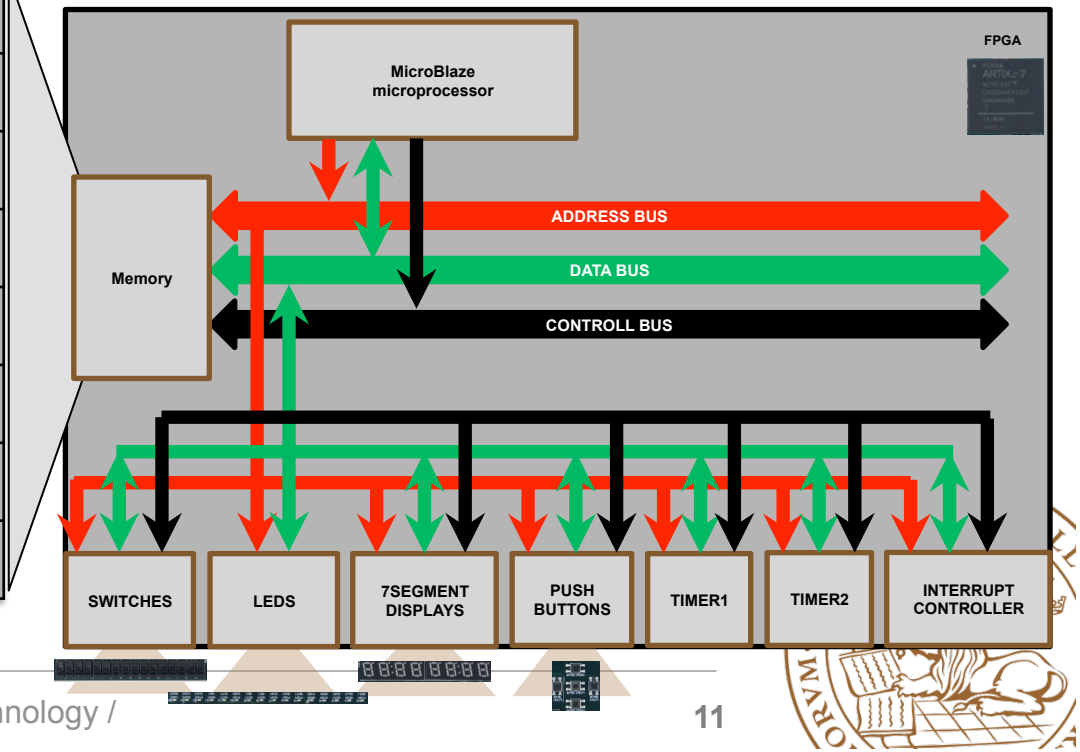
- Program the FPGA
- Write the application
- Run the application on the board



Running the application

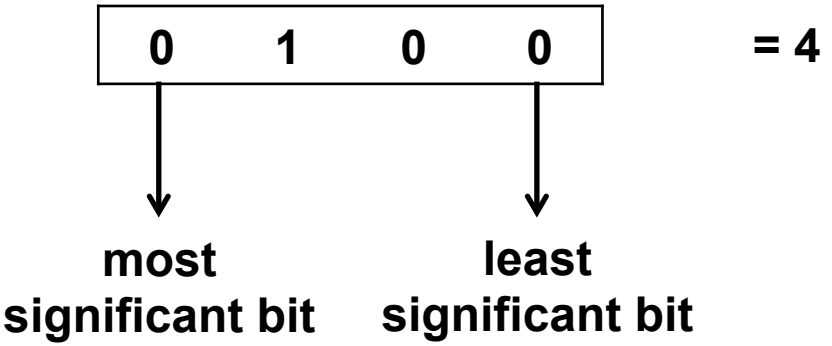
- Hardware platform implemented by programming the FPGA
- The application is downloaded to the memory in the FPGA
- Memory contains code and data (binary representation)
- The memory is byte-addressable

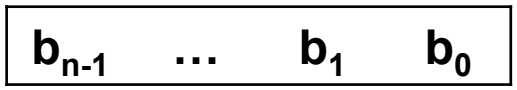
Address	1 byte information
0x17	0010 0011
0x18	0101 0000
0x19	0000 11111
0x1A	0000 1010
0x1B	1010 0101
0x1C	1110 0001
0x1D	0101 1101



Binary representation of numbers

- Binary digits: “0” and “1”

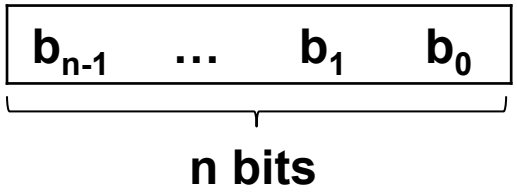




How many numbers can be represented with n bits?

Signed vs unsigned numbers

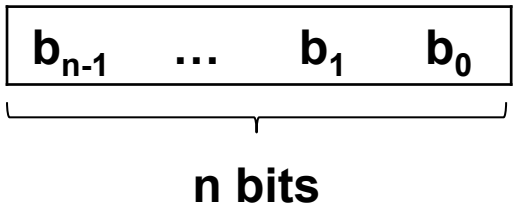
- **Unsigned numbers**



$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

Range: $[0..2^n-1]$

- **Signed numbers (two's complement representation)**

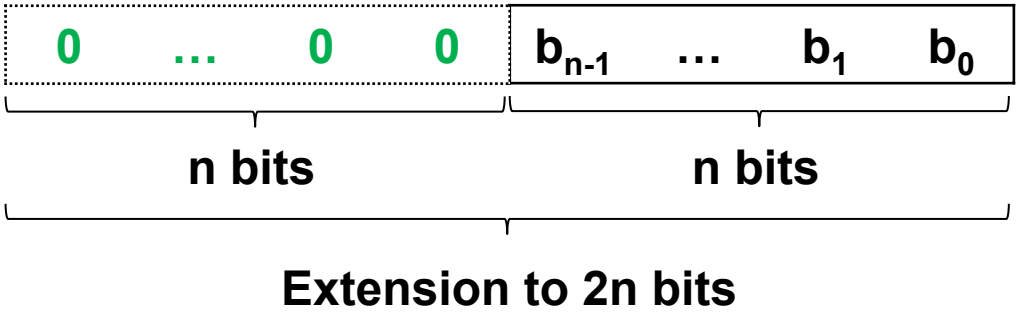


$$= (-1) \times b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$$

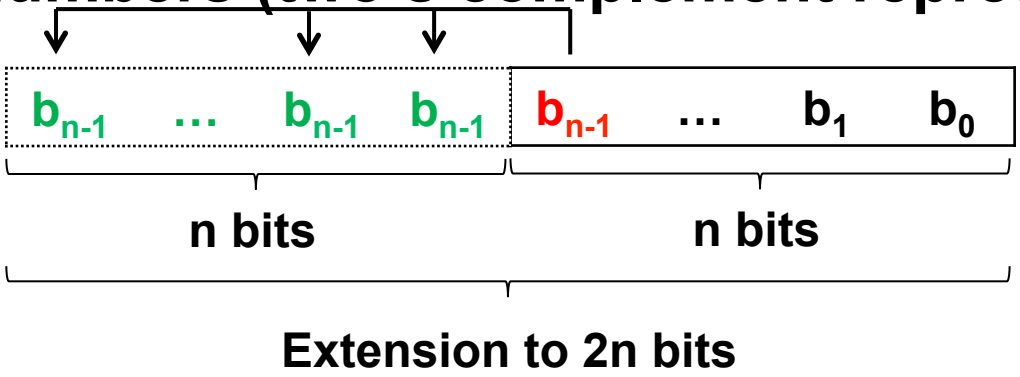
Range: $[-2^{n-1}..2^{n-1}-1]$

Extension

- **Unsigned numbers**

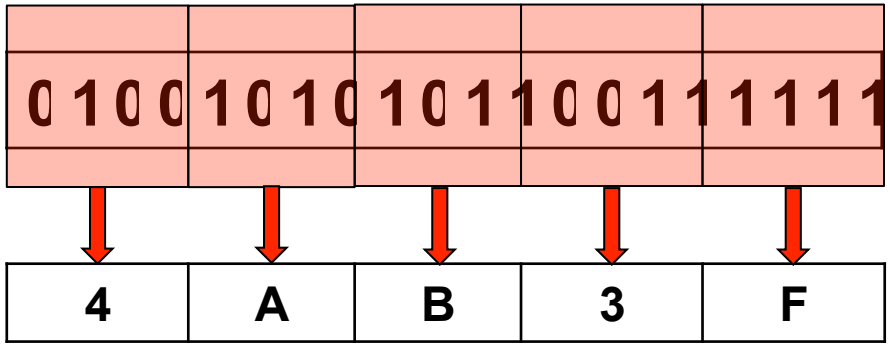


- **Signed numbers (two's complement representation)**



Hexadecimal representation

- 4 bits translate into a single hexadecimal digit
- There are 16 hexadecimal digits: 0..9 A B C D E F
- `0`=0000 .. `F`=1111



Storing data in memory

- Memory is byte-addressable
- Data and instructions are stored in memory
- Different data types have different sizes
- Data/instruction can occupy several consecutive memory locations

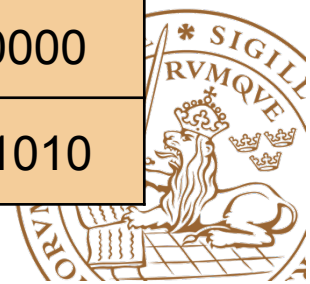
Address	Data
0	1111 0000
1	1010 0101
2	1100 0011
3	0011 0011
4	1111 1111
5	0000 1111
6	1111 0000
7	1010 1010

Address	Data
0	1111 0000
1	1010 0101
2	1100 0011
3	0011 0011
4	1111 1111
5	0000 1111
6	1111 0000
7	1010 1010

Byte eller wordadresserat minne

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	1111 1111
5	5	0000 1111
6	6	1111 0000
7	7	1010 1010

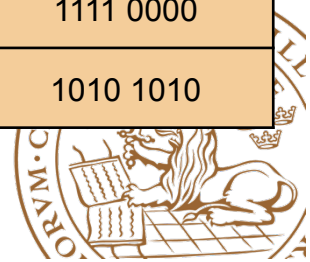
Adress	Byte	Data
0	0	1111 0000
	1	1010 0101
	2	1100 0011
	3	0011 0011
1	4	1111 1111
	5	0000 1111
	6	1111 0000
	7	1010 1010



Endianness

- Refers to the order of bytes in data or instructions stored in memory
- Consider data type of size word=4 bytes
- The data is stored at address 4
- 2 alternatives:
  - 11111111 00001111 11110000 10101010
  - 10101010 11110000 00001111 11111111
- Big endian vs. Little Endian

Address	Data
0	1111 0000
1	1010 0101
2	1100 0011
3	0011 0011
4	1111 1111
5	0000 1111
6	1111 0000
7	1010 1010



Hur sätts byte ihop till word?

MSB LSB

Givet: **1111 1111** **0000 1111** **1111 0000** **1010 1010**

Big-endian	
Adress	Data
….	….
n	1111 1111
n+1	0000 1111
n+2	1111 0000
n+3	1010 1010
….	….

Little-endian	
Adress	Data
….	….
n	1010 1010
n+1	1111 0000
n+2	0000 1111
n+3	1111 1111
….	….

Writing an application in C

- The program starts with the “main” function

include header files

global variables

local variables

statements

end of program

```
#include “address_mapping.h”

char b;

int main(){
int var;
      b=9;
      while (b>0){
         b--;
      }
      var=b;
      return var;
}
```

Data types and structures

- Primitives
  - `char x;`
  - `unsigned char x;`
  - `int x;`
  - `unsigned int x;`
- Arrays
  - `char x[10];`
  - `unsigned char x[10];`
  - `int x[10];`
  - `unsigned int x[10];`

Assigning values

- Decimal

```
char x;
x=10;
```

- Binary

```
char x;
x=0b10110;
```

- Hexadecimal

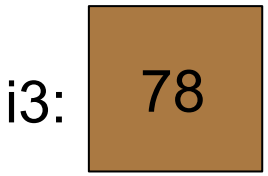
```
char x;
x=0xFD;
```

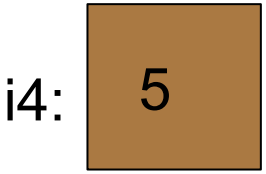
Pointers

- Specific data type, where the value of the variable points to a memory address
- Capability of modifying the contents in memory where the pointer points to
- Specific operators
  - “&” (address of)
  - “*” (dereferencing a pointer)
- Declarations of pointers:

```
<data type> * <name_of_pointer_var>;
unsigned char *x;
int *x;
char *x[10]; // array of pointers
```

Pekare (Intro)





- Deklarationen:

```
int i3, i4
```

- Ger att:

  i3 och i4 är heltalsvaribler

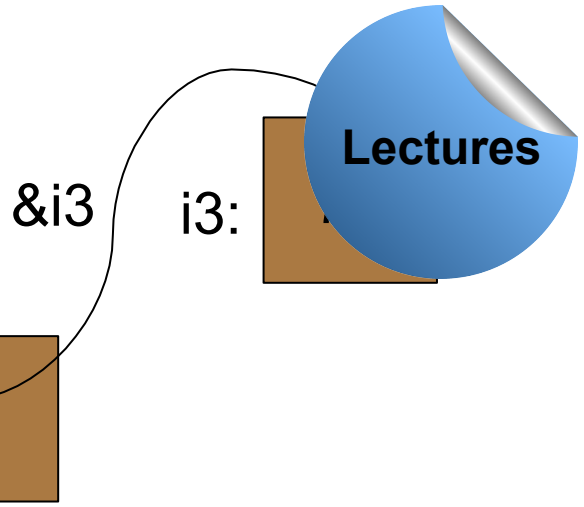
- Exempel:

```
i3=78;  //sätter i3 till 78
i4=5    //sätter i4 till 5
```

- Låt i3 vara lagrat på adress 0
- Låt i4 vara lagrat på adress 1

Adress	Data
0 (i3)	78
1 (i4)	5
2	
3	
4	
5	
6	

Pekare



- Deklarationen:

```
int i3, i4, *ip1, *ip2;
```

- Ger att:

  *ip1 och *ip2 är heltalspekarvariabler
  i3 och i4 är heltalsvariabel

- Exempel:

```
i3=78;    //sätter i3 till 78
ip1=&i3   //sätter ip1 att peka på
            adress där i3 finns
```

  & ger adressen till något

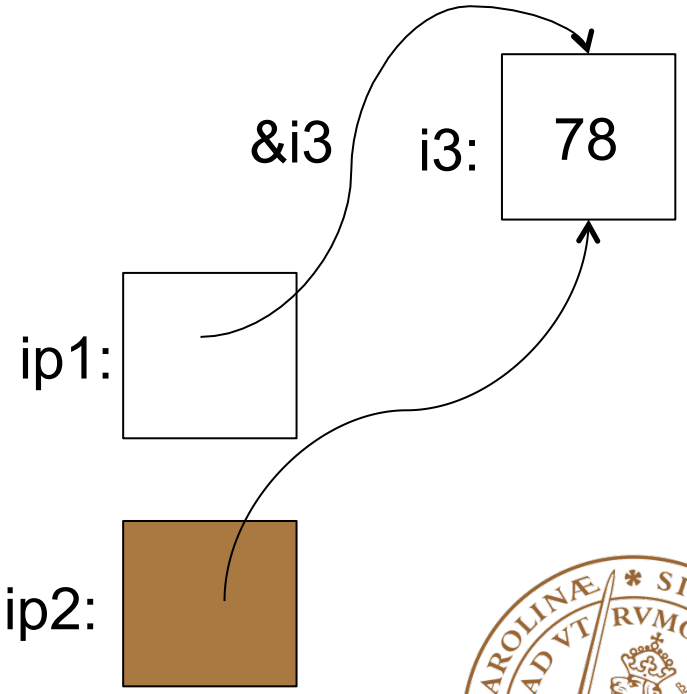
- Notera: ip1 har plats för en pil (adress) och i3 har plats för ett heltal

Adress	Data
0 (i3)	78
1 (i4)	5
2 (ip1)	
3 (ip2)	
4	
5	
6	

Pekare

- Exempel: Deklarationen (samma som innan):

```
int *ip1, *ip2, i3, i4
i3=78
ip1=&i3

ip2=ip1  //ip2 sätts att peka
         på samma som ip1
```

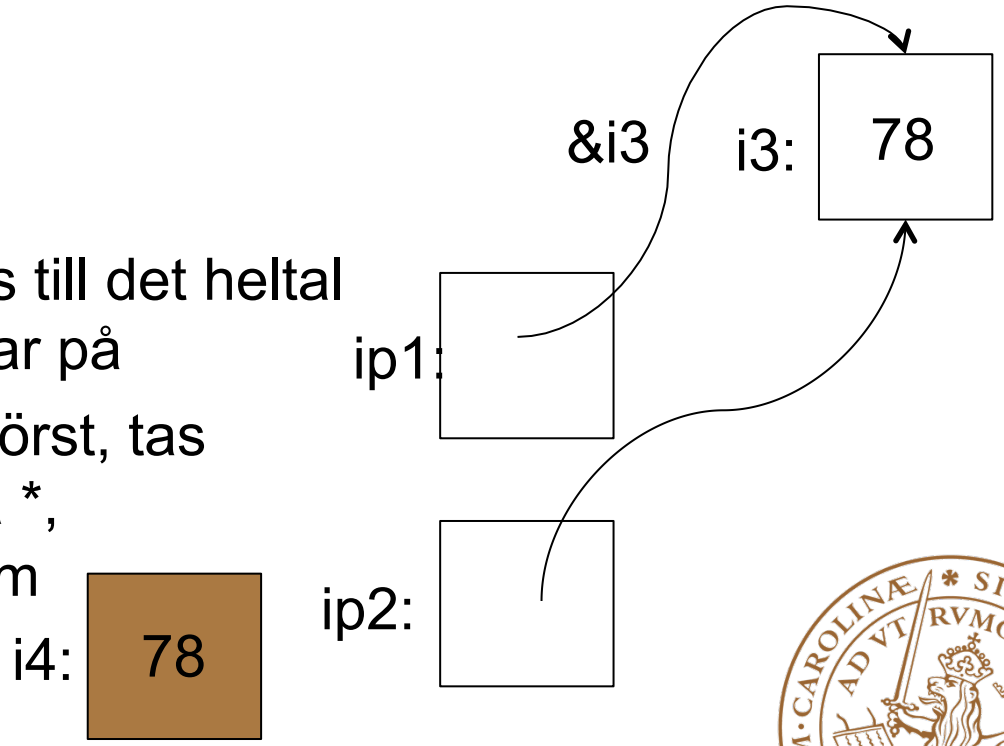


Pekare

- Exempel: Deklarationen (samma som innan):

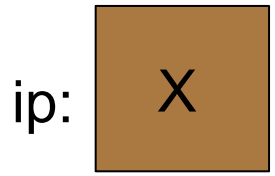
```
int *ip1, *ip2, i3, i4
i3=78;
ip1=&i3;
ip2=ip1;

i4=*ip1;  //i4 sätts till det heltal
          som ip1 pekar på
```

- *ip1 består av två steg. Först, tas pekaren fram. Sedan, via *, tas värdet till pekaren fram



Pekare

- Precis som andra variabler, blir pekare inte automatiskt tilldelade ett värde vid deklaration.
- För att sätta en pekare att peka på ingenting:
  ip=NULL;
- Sätts inte en pekare att peka på ingenting kan den peka på vad som helst – det som råkar ligga på den minnesplatsen.



Conditional statement

Syntax:

```
if (condition){
    statements if condition satisfied;
} else {
      statements if condition not satisfied;
}
```

```
if (condition){
    statements if condition satisfied;
}
```

Example:

```
if (x>10){
    y=1;
    x=x/10;
} else {
    y=0;
}
```

Multiple choice statement

Syntax:

```
switch (expression){
        case   choice_1:      statements;
                              break;

        case   choice_N:      statements;
                              break;
        default:              statements;
}
```

Example:

```
switch (x){
    case 0:  y=1;
             break;
    case 1:  y=0;
             break;
    default: y=2;
}
```


Loops

Syntax:

```
for(var_initialization; condition; var_update ){
    statements;
}
```

Example:

```
for(x=10; x>0; x-- ){
    y++;
}
```

Loops

Syntax:

```
while (condition){
    statements;
}
```

Example:

```
while(x>0){
    y++;
    x--;
}
```

Loops

Syntax:

```
do {
    statements;
} while (condition);
```

Example:

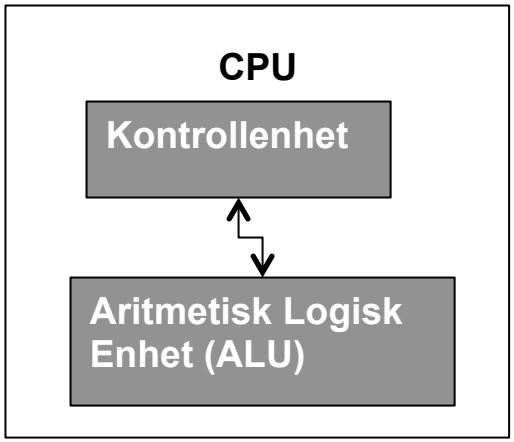
```
do{
    y++;
    x--;
} while(x>0);
```

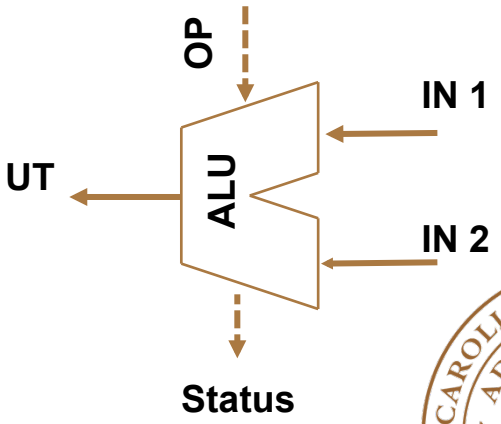
Bitwise operators in C

Operator	Symbol
bitwise AND	&
bitwise OR	|
bitwise XOR	^
bitwise NOT	~
right shift	>>
left shift	<<

Bithantering

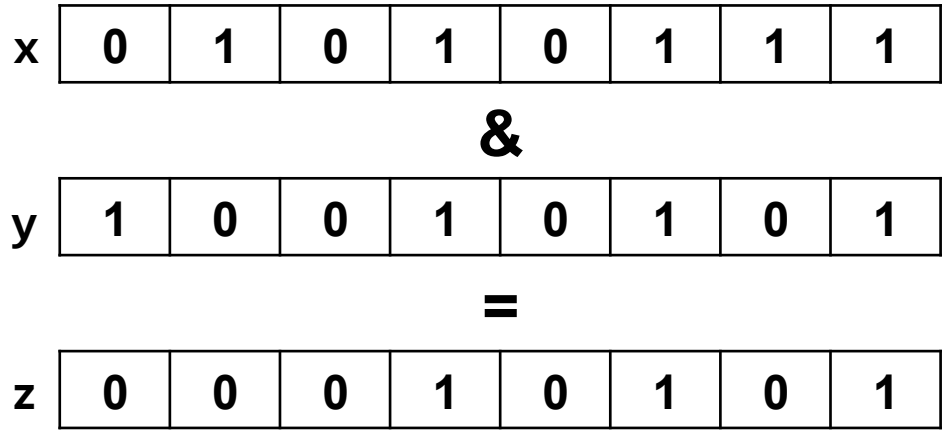
- Och (AND): &
- Eller (OR): |
- Exklusivt eller (XOR): ^
- Invertering (NOT): ~
- Vänstershift: <<
- Högershift: >>





Bitwise AND

```
z= x & y;
```



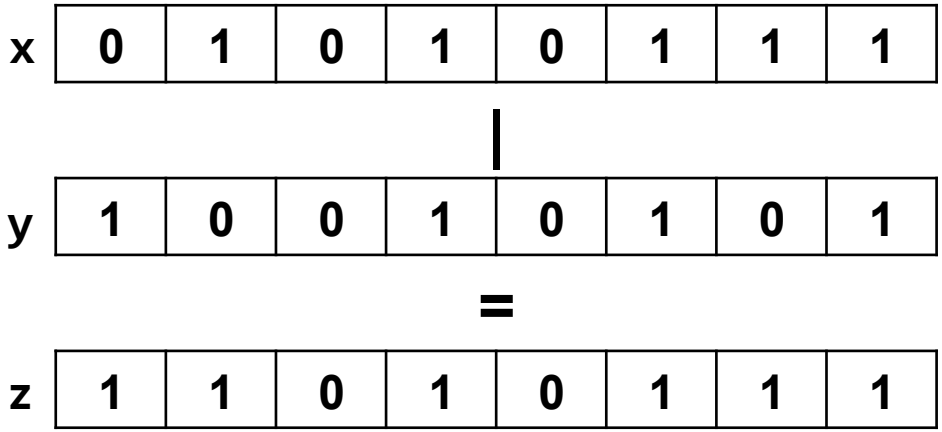
`z&=x;` ↔ `z=z&x;`

Useful to cancel (set to zero) bits, so that focus can remain on bits at particular bit positions

z=x&0b01010010

Bitwise OR

```
z= x | y;
```



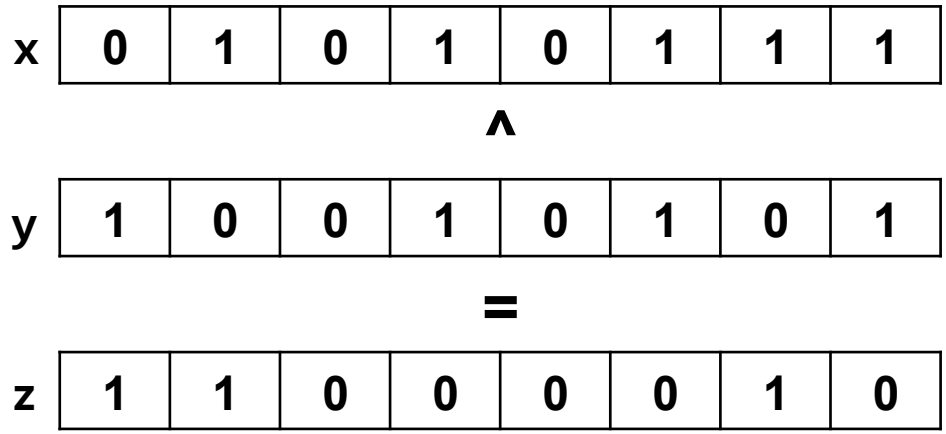
`z|=x;` ↔ `z=z|x;`

Useful to set bits at particular bit position to one, while bits at other bit positions are intact

z=x|0b0101001 0

Bitwise XOR

```
z= x ^ y;
```



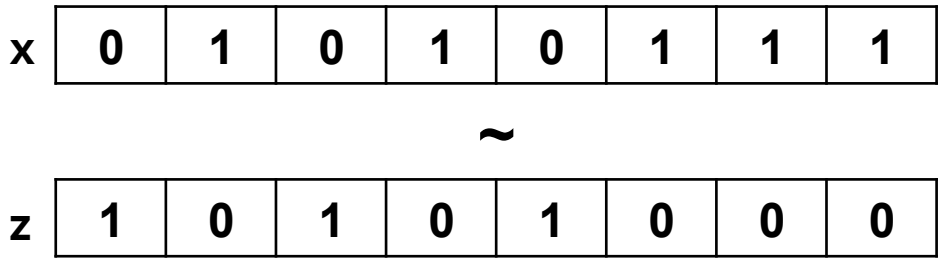
`z^=x;` ↔ `z=z^x;`

Useful to flip bits at particular bit positions

z=x^0b01010010

Bitwise NOT

```
z= ~x;
```



Inverts all bits in a given number (ones' complement)

Left shift

```
z= x << 3;
```

			0	1	0	1	0	1	1	1	x
		0	1	0	1	0	1	1	1	0	x<<1
	0	1	0	1	0	1	1	1	0	0	x<<2
0	1	0	1	0	1	1	1	0	0	0	z

Left shift is equivalent to multiplication by 2

How to force the bit at position 0 to be observed at position 6?

Right shift

```
z= x >> 3;
```

x	0	1	0	1	0	1	1	1			
x>>1	0	0	1	0	1	0	1	1	1		
x>>2	0	0	0	1	0	1	0	1	1	1	0
z	0	0	0	0	1	0	1	0	1	1	1

Right shift is equivalent to division by 2 for unsigned numbers or positive signed numbers

How to force the bit at position 6 to be observed at position 0?

Purpose

- Write applications in C
- How different data types are stored in memory
- The size of different data types

LUNDS
UNIVERSITET