Applications

Design approaches

- Top-down approach
- Bottom-up approach

Design approaches

- If we look at a problem as a whole, it may seem impossible to solve because it is so complex
  - e.g. Design your own processor
    - Pipeline
    - ALU
    - Cache
    - ISA

Design approaches

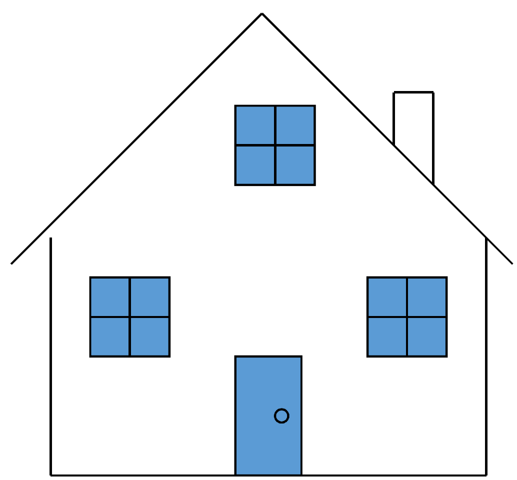
- Complex problems can be solved using **top-down design** where
  - We break the problem into parts
  - Then break the parts into sub-parts
  - …
  - Soon, each of the parts will be easy to understand, manage and implement
    - Repeat until you can manage the parts
- In bottom-up design the individual parts of the system are specified first
  - The parts are used to implement larger components
  - …
  - At the end the last component to build is the system itself

Design approaches

- Top-down advantages
  - Breaking the problem into parts helps us to clarify what needs to be done.
  - At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
  - Parts of the solution may turn out to be reusable.
  - Breaking the problem into parts allows more than one person to work on the solution.

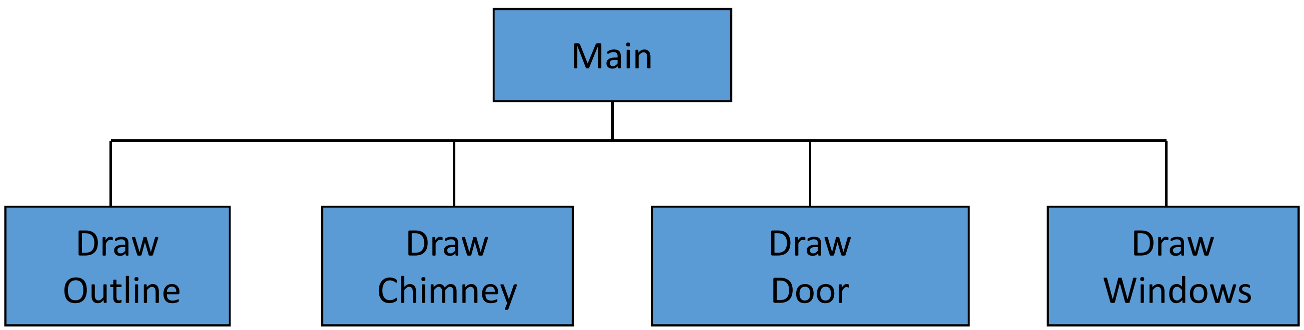
Example

- Problem: Write a program that draws this picture of a house.



The Top Level

- Draw the outline of the house
- Draw the chimney
- Draw the door
- Draw the windows



Pseudocode for Main

Call Draw Outline

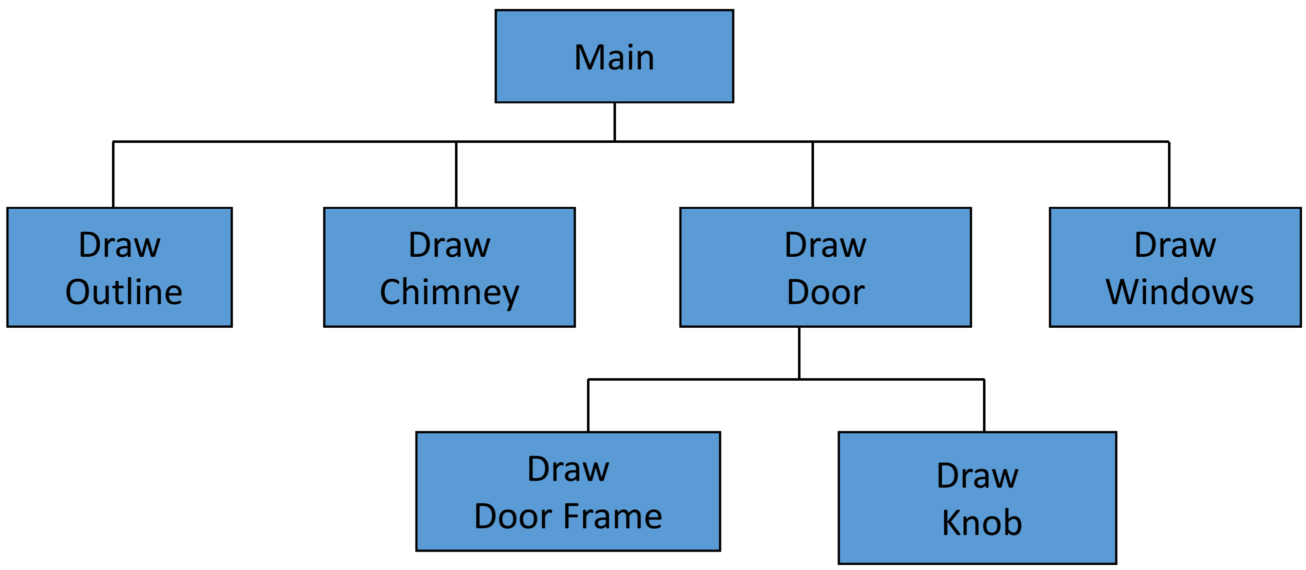
Call Draw Chimney

Call Draw Door

Call Draw Windows

The second level

- Observation
  - The door has both a frame and knob.
  - We could break this into two steps.



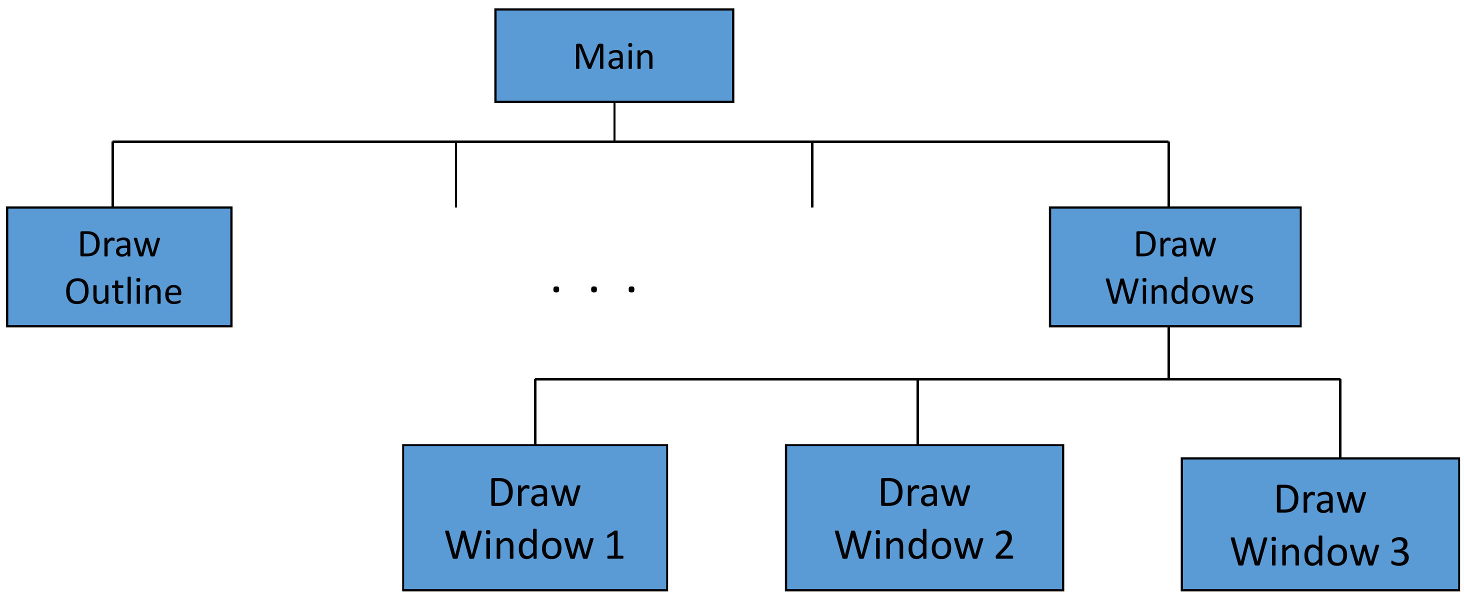
Pseudocode for Draw Door

Call Draw Door Frame

Call Draw Knob

The second level

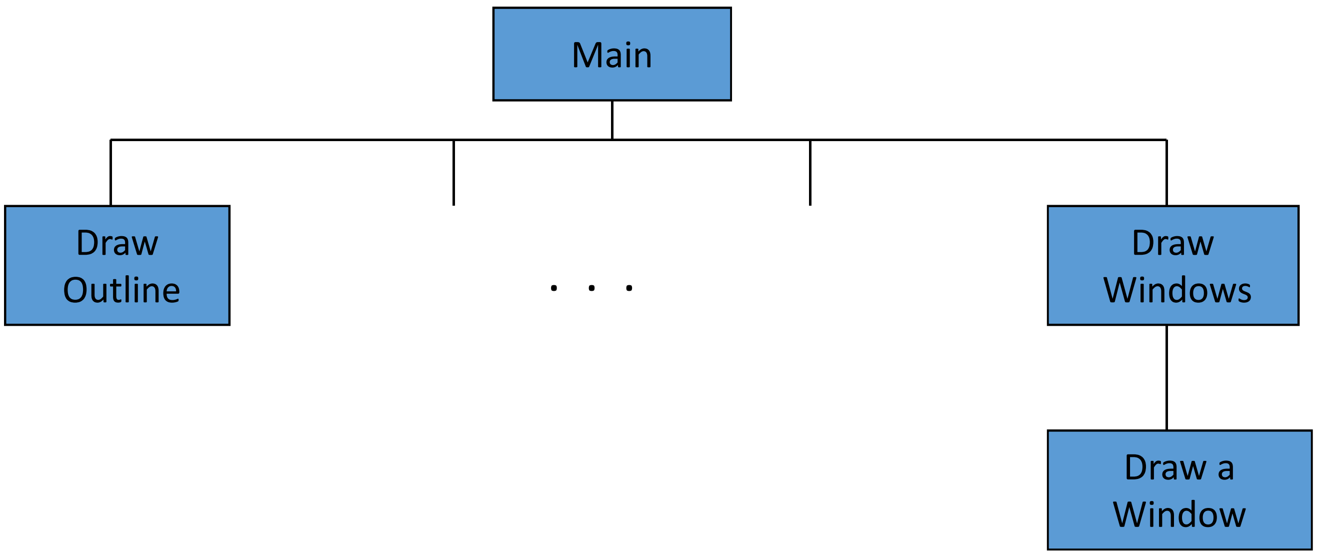
- Observation
  - There are three windows to be drawn.



The second level

- Reusability
  - But don’t the windows look the same? They just have different locations.
  - So, we can reuse the code that draws a window.
    - Simply copy the code three times and edit it to place the window in the correct location, or
    - Use the code three times, “sending it” the correct location each time (we will see how to do this later).
  - This is an example of **code reuse**.

Reusing the Window Code



Pseudocode for Draw Windows

Call Draw a Window, sending in Location 1

Call Draw a Window, sending in Location 2

Call Draw a Window, sending in Location 3

The last level

- Basic drawing elements
  - Funtions
    - Rectangle
    - Circle
    - Line
    - Point
  - Entities
    - Position
    - Coordinate systems
    - Line attributes

The last level

- Draw a window

```
Draw frame – use rectangle
Draw horizontal line
Draw vertical line
```

Design approaches

- Once you get enough experience you can start with parts (e.g. reusable small compoents) and use a bottom-up approach

Applications

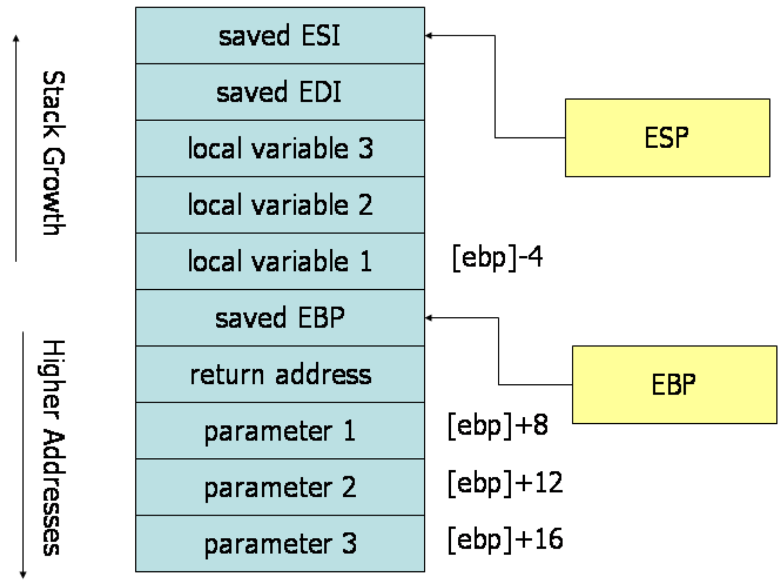
Let’s consider the following sequence of code

```
1.      int do_math(int a)
2.      {
3.         int b = 1, c = 20;
4.         int *p =&c;
5.         *p = (a + b)*c;
6.         return *p;
7.      }
```

1. What do the program stack contains before execution of the math operation (line 5)?
2. What do the program stack contains before returning from function (line 6)?
3. What do the program stack contains after the execution of the function? What is the value returned by the function?
4. What is wrong with the given code?

Assembly language

- Subroutine calling convention



Applications

Design an 8086 microprocessor system using the following memory and I/O requirements:

- 128KB EPROM starting at 00000H, using 64K x 8bits memories
- 256KB DRAM, using 64K x 4 bits memories
- 4 LEDs
- 1 push button

Detailed requirements

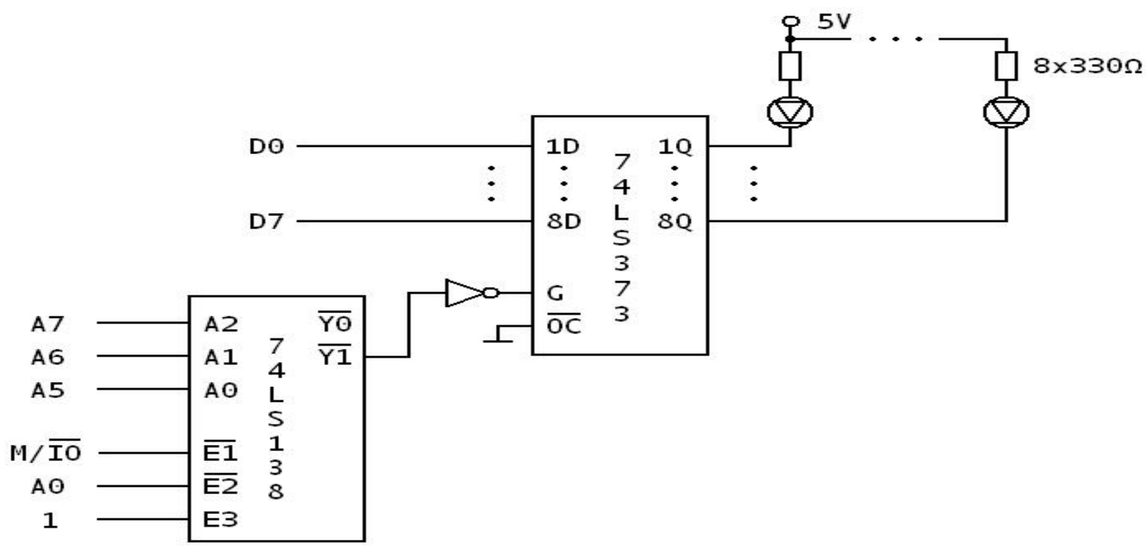
1. The block diagram of the system
2. The number of memory circuits required by the design, number of 16bits memory blocks and their sizes, and the memory map
3. Design the I/O connection logic and drivers (hardware and/or software) to microprocessor
4. Considering the following variables' definitions in C used in the firmware. Determine the memory size and memory content and addresses used by a 16 bits compiler (integer and pointers are stored on 16 bits). The variables are loaded at the beginning of DRAM.

```
char ver[] = "DMD 01/08/2018";
unsigned int cnt = 0;
unsigned int msk = cnt-1;
int *p = &cnt;
```

5. Write the main program that will read the button and count the number of presses/taps using the variable cnt. Turn on the 4 LEDs according with the value stored in the counter, as follows: when cnt is lower then ¼ of the max value possible to be stored into cnt, 1 LED if the cnt >= ¼ max and < ½ max, 2 LEDs cnt >= ½ max and < ¾ max; 3 LEDs if cnt >= ¾ max; and 4 LEDs if an overflow has been reached

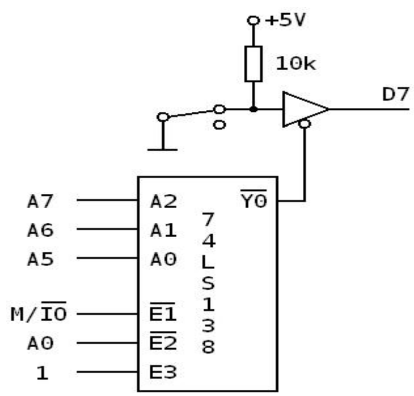
Applications

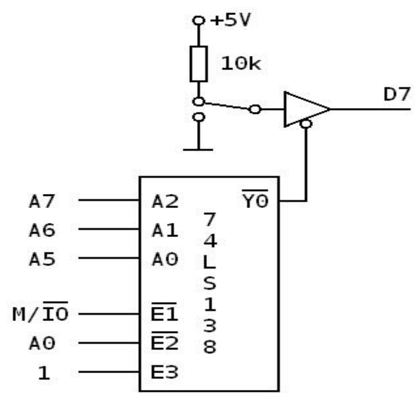
- I/O ports
  - LEDs



Applications

- Button





Applications

Design an 8086 microprocessor based system using the following memory and I/O requirements:

- 128KB EPROM starting at 00000H, using 32K x 16bits memories
- 64KB SRAM starting at 40000H, using 16K x 8 bits memories
- 256KB DRAM, using 32K x 1 bit memories, placed as a contiguous memory address space
- 1 LED placed into I/O space at address range 00H-0FH

Detailed requirements

- Number of memory circuits required by the design, number of 16bits blocks and their sizes
- Memory map
- I/O decoder and LED connection to microprocessor
- Write the initialization routine that copies the second half of EPROM over the first half of SRAM. Light the LED on when the SRAM memory is initialized
- Considering the logical segment 4800H, which is the size of the physical segment supported by SRAM circuits?