Part I

Hardware design

Verilog

Lesson 1

Problem: multiplication of two 4-bit numbers, through the algorithm that uses only addition operations.

Pseudocode description:

```
while (START == 0)
        ; // do nothing, just wait
read A, B;
S = 0;
while (B > 0) {
        S = S + A;
        B = B - 1;
        }
READY = 1;
```

Action elements (derived from the pseudocode description above):

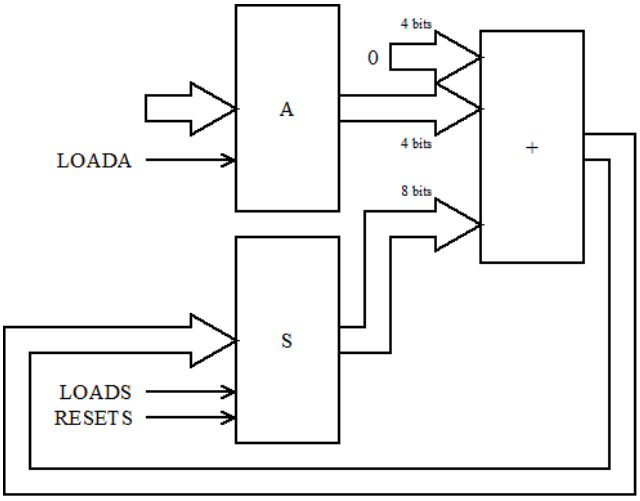
- A, S - parallel registers
- B - counter (updates by decrement)
- combinational: adder, comparison circuit - required for the arithmetic part

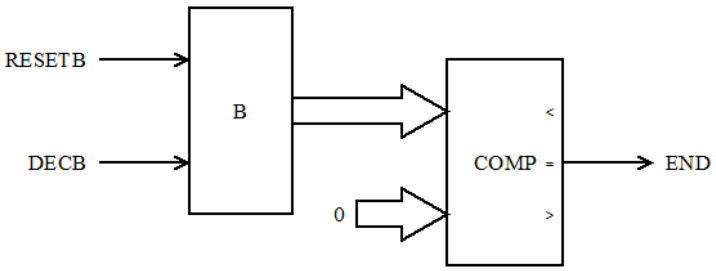
Other signals:

- START (input) - when on value 1, we are notified that new values are available on A and B
- READY (output) - when on value 1, we notify that the result is computed and available on S

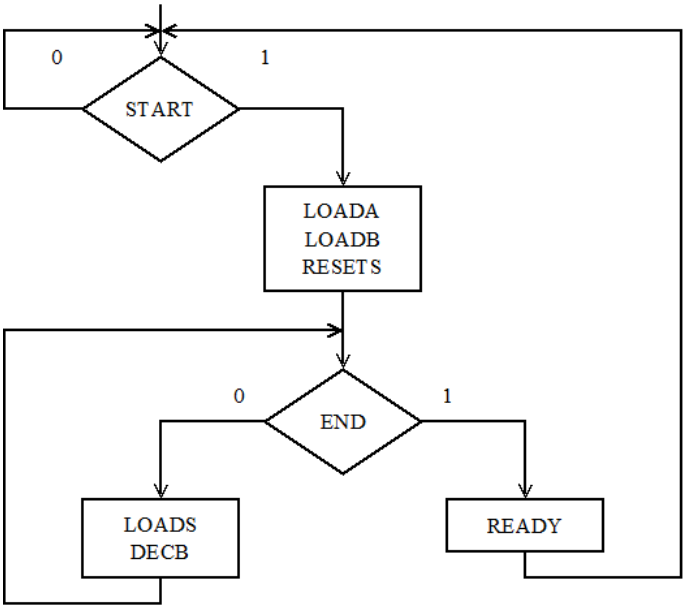
Action elements - implementation and connections

command and test signals of each action element - according to the description of elementary circuits

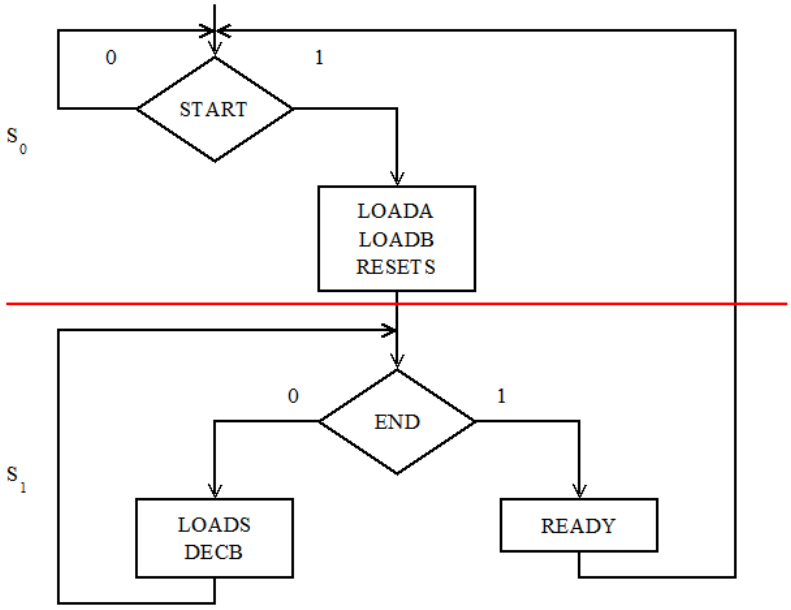




Logical description of actions:



State delimitation for the hardwired sequencer:



Functional equations:

$s_{0,n+1}=s_{0,n}\cdot\overline{START}+s_{1,n}\cdot END$

$s_{1,n+1}=s_{0,n}\cdot START+s_{2,n}\cdot\overline{END}$

$LOADA=LOADB=RESETS=s_{0,n}\cdot START$

$LOADS=DECB=s_{1,n}\cdot\overline{END}$

$READY=s_{1,n}\cdot END$

Lesson 2

Problem: multiplication of two 4-bit numbers, through the algorithm using shift and addition operations.

Pseudocode description:

```
while (START == 0)
        ; // do nothing, just wait
read A, B;
S = 0;
I = 0;
while (I < 4) { // 4 bits in B
        S = (S << 1) + A*B[3]; // MSB of register B; multiplication can actually be implemented by AND
        B = B << 1;
        I = I + 1;
        }
READY = 1;
```

Action elements (derived from the pseudocode description above):

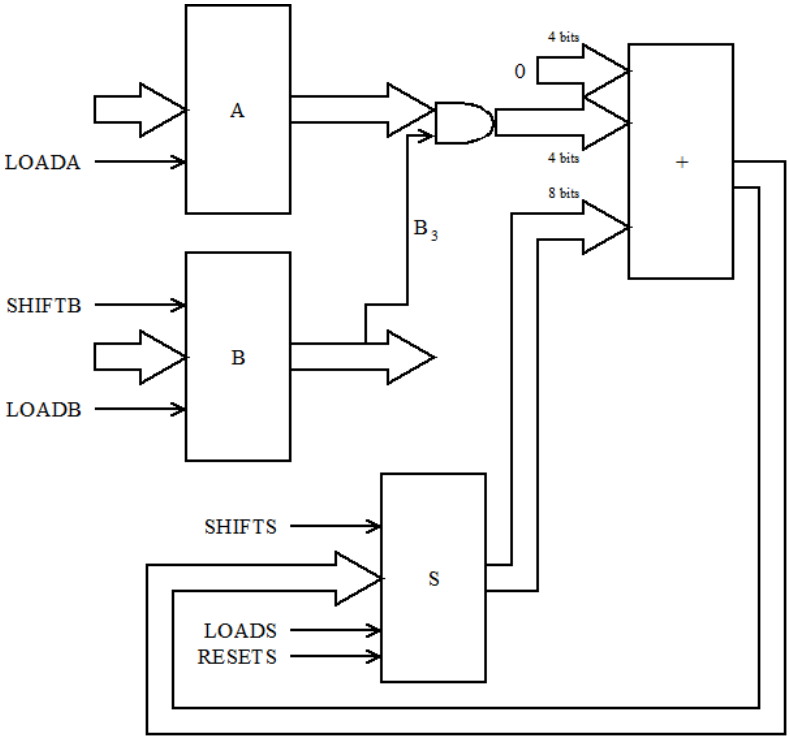
- A - parallel register
- B, S - shift registers
- I - counter
- combinational: adder, comparison circuit, AND gates - required for the arithmetic part

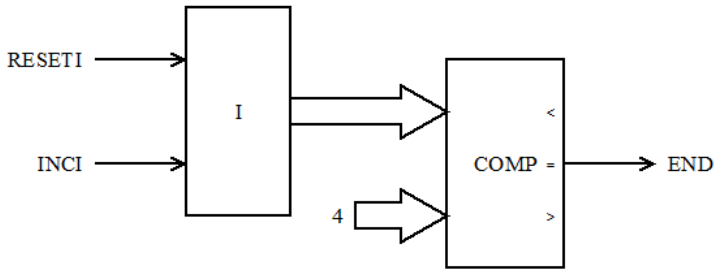
Other signals:

- START (input) - when on value 1, we are notified that new values are available on A and B
- READY (output) - when on value 1, we notify that the result is computed and available on S

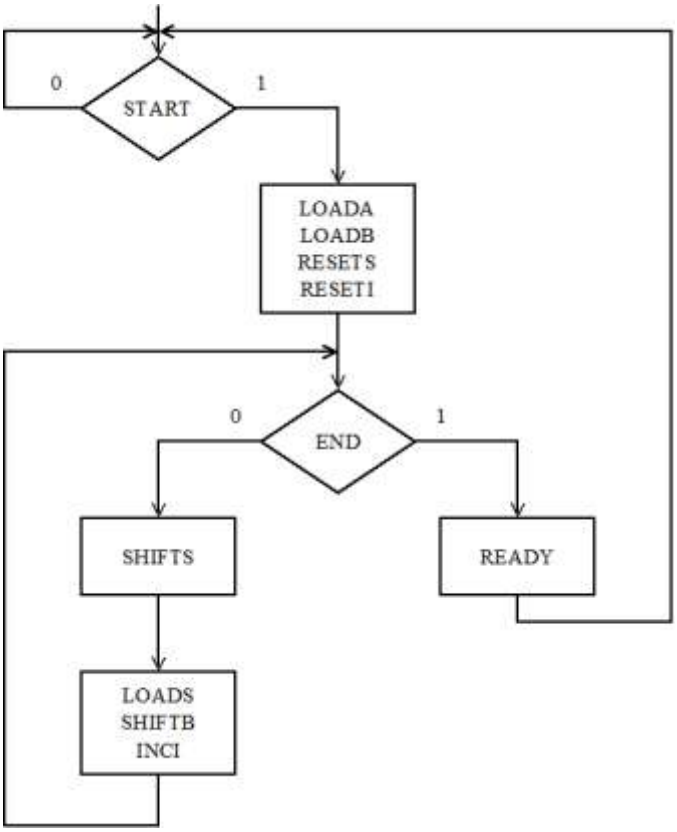
Action elements - implementation and connections

- command and test signals of each action element - according to the description of elementary circuits

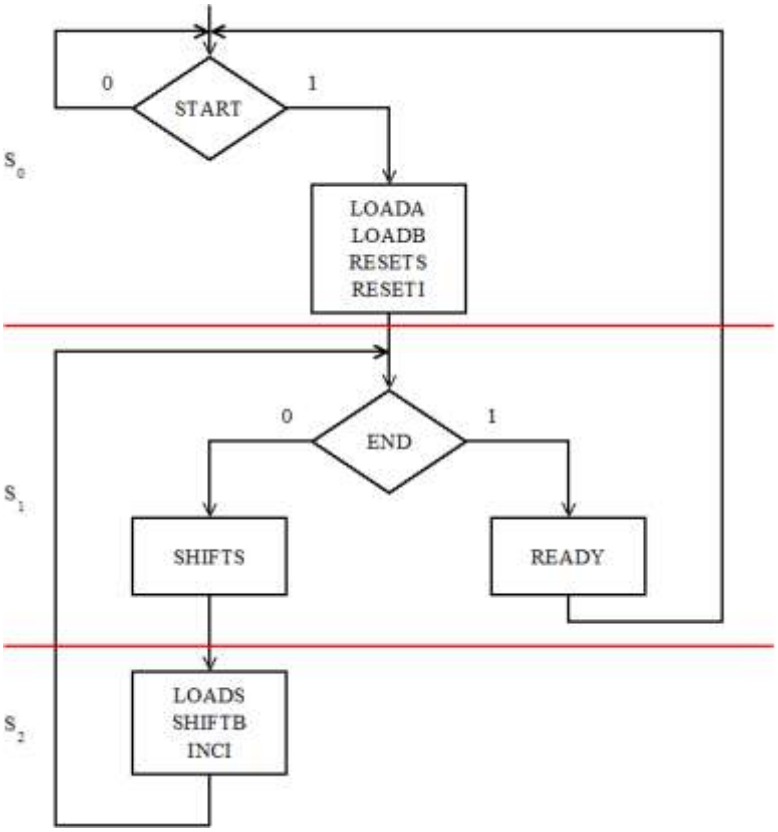




Logical description of actions:



State delimitation for the hardwired sequencer:



Functional equations:

$s_{0,n+1}=s_{0,n}\cdot\overline{START}+s_{1,n}\cdot END$

$s_{1,n+1}=s_{0,n}\cdot START+s_{2,n}$

$s_{2,n+1}=s_{1,n}\cdot\overline{END}$

$LOADA=LOADB=RESETS=RESETI=s_{0,n}\cdot START$

$SHIFTS=s_{1,n}\cdot\overline{END}$

$READY=s_{1,n}\cdot END$

LOADS=SHIFTS=INCI=$s_{2,n}$

When microprogramming is employed (either horizontal or vertical), states may be delimited in a different manner - more states may be necessary. The action elements and the actions themselves remain unchanged.

Homework:
Microprogrammed implementation
- horizontal microinstructions
- vertical microinstructions

Lesson 3

The Verilog language
- See courses 5-7.

Icarus Verilog
- Command line tool.
- Download from http://bleyer.org/icarus/ (Windows versions; for Linux refer to the distribution's packages).
- Install it, following all recommendations from the installer.
- Compiling a Verilog file:
    iverilog file.v
    it might be necessary to provide the full path of iverilog.exe
    successful compilation results in the creation of file a.out
- Running a simulation:
    vvp a.out
    same as above regarding the path

Testing a module
- Suppose we have designed a module and we want to test it.
- We need to supply values on its inputs and collect the values of the outputs.
- We create a top-level module, within which we declare an instance of the module we have designed.

Example: multiplexer module (see course 5)

```
module MUX(Sel,I0,I1,E); // multiplexer module
input Sel,I0,I1;
output E;
assign E=(I1&Sel)|(I0&~Sel);
endmodule
module main(); // test module
reg sel,i0,i1;
wire e;
MUX m(sel,i0,i1,e); // multiplexer module instance
initial
begin
     $monitor("%d\t%d\t%d\t%d",sel,i0,i1,e);
     // anytime at least one of the variables in the list is changed, all their values are displayed
     sel<=0;
     i0<=1;
     i1<=0;
end
endmodule
```

Problem: design a clock generator
- no inputs, one output bit
- clock period: 10 time units
- so the output flips every 5 time units

```
module clock(c);
output reg c;
initial
     c<=0;
always
     #5 c<=~c;
endmodule
module main();
wire Clk;
clock cc(Clk);
```

```
initial
     $monitor("%d\t%d",$time,Clk); // $time - shows simulation time
     #100 $finish(); // terminate simulation
endmodule
```

Homework:
Test the other alternatives for multiplexer implementation from course 5.

Lesson 4

Problem: multiplication of two 4-bit numbers, through the algorithm using only addition operations
- same as for Lesson 1
- this time, Verilog implementation

Finite state machine design (automaton)
- see course 6
- the principles (and the states) are the same as in Lesson 1
- don't forget: loops are implemented through state transitions, not through `for`/`while` keywords

Multiplicator module:

```
module MUL(Clk,Start,A,B,Result,Ack);
input Clk,Start; // Start - same role as signals START from Lesson 1
input [3:0]A,B;
output reg [7:0]Result;
output reg Ack; // same role as signal READY from Lesson 1
reg [3:0]x,y; // used for keeping the values read from inputs A and B
reg [2:0]state; // the current state of the automaton
always @(posedge Clk)
     if(state==0)
          if(Start==0) begin
               Ack<=0;
               state<=0; // always update the state, even if the new value is the same
               end
          else begin
               x<=A;
               y<=B;
               Result<=0;
               state<=1;
               end
     else if(state==1)
          if(y>0) begin
               Result<=Result+x;
               y<=y-1;
               state<=1;
               end
          else begin
               Ack<=1;
               state<=0;
               end
initial begin
     Ack<=0;
     state<=0;
end
endmodule
```

Clock generator:
- also necessary for testing
- implementation - see last time

Test module:

```
module main();
reg start;
reg [3:0]a,b;
wire [7:0]res;
```

```
wire Clk,ack;
MUL m(Clk,start,a,b,res,ack);
clock c(Clk);
initial begin
     $monitor("%d\t%d\t%d\t%d\t%d\t%d",$time,start,a,b,res,ack);
     start<=0;
     #2 a<=5;
     b<=7;
     #2 start<=1;
     #100 $finish();
end
endmodule
```

Lesson 5

Problem: multiplication of two 4-bit numbers, through the algorithm using shift and addition operations
- same as for Lesson 1
- but Verilog implementation

Multiplicator module:

```
module MUL(Clk,Start,A,B,Result,Ack);
input Clk,Start;
input [3:0]A,B;
output reg [7:0]Result;
output reg Ack;
reg [3:0]x,y;
reg [3:0]i; // counter - the number of bits in B
reg [2:0]state;
always @(posedge Clk)
     if(state==0)
          if(Start==0) begin
               Ack<=0;
               state<=0;
               end
          else begin
               x<=A;
               y<=B;
               i<=0;
               Result<=0;
               state<=1;
               end
     else if(state==1)
          if(i==4) begin
               Ack<=1;
               state<=0;
               end
          else begin
               Result<=Result<<1;
               state<=2;
               end
     else if(state==2) begin
          Result<=Result+x*y[3];
          y<=y<<1;
          i<=i+1;
          state<=1;
          end
initial begin
     Ack<=0;
     state<=0;
end
endmodule
```

The clock generator and the test module are the same as in Lesson 4, because the problem and the interface of module `MUL` are the same.

Lesson 6

Problem: design a module that emulates a ROM memory
- Inputs: *Clk* (clock signal), rd (read), *addr* (8-bit address to read from).
- Outputs: *data* (8 bits - data that has been requested).
- While *rd* has value 0, *data* remans in high impedance (see course 5).
- At the moment that *rd* gets value 1, input *addr* must already contain the address we want to read.
- When *rd* gets value 1:
    - After 2 clock periods, the value of the memory location from address *addr* is written to output *data*.
    - This value is maintained on the output for 2 clock periods, then *data* is set again to high impedance.

```
module ROM(Clk,rd,addr,data);
input Clk,rd;
input [7:0]addr;
output reg [7:0]data;
reg [8:0]mem[255:0]; // the memory cells
reg [3:0]state;
integer i;
// integer variables are not implemented as physical circuits, only used
for describing parallel operations
initial begin
     data<=8'bzzzzzzzz;
     state<=0;
     for(i=0;i<=255;i=i+1) // initialize the memory cells
          mem[i]<=255-i;
     end
always @(posedge Clk)
     if(state==0)
          if(rd==1)
               state<=1;
          else
               state<=0;
     else if(state==1)
          state<=2;
     else if(state==2) begin
          data<=mem[addr];
          state<=3;
          end
     else if(state==3)
          state<=4;
     else if(state==4) begin
          data<=8'bzzzzzzzz;
          state<=0;
          end
endmodule

module clock(c);
output reg c;
always
     #5 c<=~c;
initial
     c<=0;
endmodule

module main();
wire clk;
reg read;
```

```
reg [7:0]a;
wire [7:0]d;
ROM r(clk,read,a,d);
clock cc(clk);
initial begin
     $display("\t\ttime\twrite\tread\ta\td");
     $monitor("%d\t%d\t%d\t%d\t%d",$time,write,read,a,d);
     read<=0;
     #2 a<=138;
     #5 read<=1;
     #10 read<=0;
     #100 $finish();
end
endmodule
```

Homework:
Design a module that emulates a RAM memory. In addition to the ROM circuit, it also allows the write operation:
- an input *wr* must be added, which notifies the start of the write operations
- an 8-bit data input must be added
- 2 clock periods after *wr* gets value 1, the value from the data input is written to address *addr* and the write cycle is over

Part II

Microcontrollers

Intel 8051

Lesson 1

The Intel 8051 microcontroller
- See courses 8-9.

SDCC compiler
- Command line tool.
- Download from https://sourceforge.net/projects/sdcc/files/ the version corresponding to your OS.
- Install it, following all recommendations from the installer.
- Compiling a source file:
    sdcc file.c
    successful compilation results in the creation of several files, one of which is file.ihx

EdSim51 simulator
- Download the .zip archive from https://www.edsim51.com/ and unpack it.
- To run it, simply double-click on the .jar file (requires a Java Virtual Machine to be installed).
- Programs must be inserted in the upper-center window of the application.
- The lower window of the application shows the status of the external peripheral devices.
- To see the external peripheral devices and the way they are connected to the microcontroller, look at the large image at https://www.edsim51.com/examples.html (logicDiagram.png); it is recommended to save the image locally, so that you can easily access it whenever necessary.

Using the simulator:
- Go to page https://www.edsim51.com/examples.html and run examples 1, 2, 3, and 5 (one at a time).
- To run an example, copy-paste the code intro the program (upper-center) window of the simulator, the push the button Run (located on top that window). Read the comments of the example and see what is happening with the external peripherals (lower window) during running.

Writing and running C programs for the simulator:
- Each program must start with "`#include <8051.h>`".
- It is recommended that function `main` is of type `int` and returns value 0 in the end (just like GCC).
- The core of function `main` is usually an infinite loop, which executes repeatedly the same operations. Various initializations can be performed before the infinite loop, if necessary.
- It is recommended not to use many variables (either local or global), as the microcontroller memory is not very large.
- The four general-purpose ports of the microcontroller can be accessed, either in reading or writing, by using the predefined names `P0`, `P1`, `P2`, `P3`.
- Individual bits of the ports can also be accessed; e.g., `P0_3` denotes bit 3 of port `P0`.
- After the source file has been saved, it is compiled with SDCC as shown above.
- To run it in the EdSim51 simulator, push the Load button (on top of the program window) and open the file with the same name of the source file, but with extension .ihx; as a result, the compiled code (i.e., machine code) is loaded.
- In the beginning of the machine code there are two jump instructions, which usually look as below:

```
LJMP 0006H
LJMP 0062H
```

- Change the first of these two instructions to be identical to the second one:

```
LJMP 0062H
LJMP 0062H
```

- Only then should the Run button be pressed.
- When a program is long and slow, it could take time to run it and watch the results. It is possible to change the control Update Frequency (upper-left window of the simulator) to a value higher than 1 - either by picking a value from the predefinded list of just writing another one. This will make the simulator show the evolution of the system not after every instruction, but after every *n* instructions (where *n* is the update frequency).

All subsequent lessons will be about using the EdSim51 simulator and writing C programs for it.

Caution: sometimes the SDCC compiler does not generate the correct code, especially for arithmetic operations, so the output may be wrong. Nevertheless, it is a useful tool and will be used during these lessons.

Problem: write a C program that continuously alternates the LEDs which are turned on: the odd-indexed ones (i.e., 1, 3, 5, 7) and the even-indexed ones (i.e., 0, 2, 4, 6).

```
#include <8051.h>

int main()
{
	while(1) {
		// LEDs are connected to port P1
		P1=0x55; // turn on LEDs 0, 2, 4, 6
		P1=0xAA; // turn on LEDs 1, 3, 5, 7
	}
	return 0;
}
```

Problem: write a C program that turns the LEDs on and off based on the values of switches SW0...SW7 (inputs from the user).

```
#include <8051.h>

int main()
{
	while(1) {
		// the switches are connected to port P2
		P1=P2; // read the switch values and send them to the LEDs
	}
	return 0;
}
```

Note: in order to test this program, the user must alter the values of the switches. In the lower window of the simulator, the switches are located just below the LEDs (and are shown as a larger table, with elements numbered from 7 to 0).

Homework:
Write a program that turns on all LEDs individually, one LED at a time, and then moves to the next etc.

Lesson 2

Problem: write a C program that controls the motor, by making it continuously change the spinning direction. That is, it spins clockwise for a time T, then counterclockwise for another time T, and so on.

In order to control the motion of the motor, one must write the right values to bits 1 and 0 of port P3:
- If the bits have the same value (either 0 or 1), the motor stops.
- If the bit values are different, the motor spins; the spinning direction depends on which bit has value 0 and which has value 1.
- In conclusion, the values that must be written to port P3 are 1 and 2, respectively; they correspond to combinations 01 and 10, respectively, on bits 1 and 0.

```
#include <8051.h>

unsigned char direction;
unsigned char i;

int main()
{
     direction=1;
     while(1) {
          P3=direction;
          for(i=0;i<50;i++)
               ; // the time duration for spinning in a certain direction is simulated by an empty loop
          if(direction==1)
               direction=2;
          else
               direction=1;
     }
     return 0;
}
```

Homework:
Write a program that controls the spinning direction based on the value of switch SW0.

Homework:
Write a program that, in addition to the previous one, also decides whether the motor stops or spins, depending on the value of switch SW1:
- If SW1 has value 0, the motor stops.
- If SW1 has value 1, the motor spins and its spinning direction is given by the value of SW0.

Lesson 3

Problem: write a C program that displays a number (positive integer) on the 7-segment displays.

The display control is more complicated:
- The configuration to be displayed is written on port P1.
- The display on which the configuration is shown is controlled by bits 3 and 4 of port P3 and, additionally, by bit 7 of port P0.

As all displays gate their value from port P1, at most one of them is turned on at any moment, due to the decoder:
- If bit 7 of port P0 has value 0, all displays are turned off (the decoder has no active output).
- Otherwise, 3 and 4 of port P3 make a 2-bit number which indicates which decoder output is active - that is, which display is turned on. For example, if the 2-bit number is 01, which means 1 in base 10, then display 1 is turned on.

```
#include <8051.h>

unsigned char c;
unsigned char chars[4];

int main()
{
     unsigned int n,i;
     P0_7=0; // turn off all displays until we are ready
     P3=0;
     n=2157; // the number to be displayed
     i=n%10; // compute the first (rightmost) digit in base 10
     c=(unsigned char)i;
     chars[0]=c; // store the first digit
     i=n/10; // compute the second digit, the store it, and so on
     i=i%10;
     c=(unsigned char)i;
     chars[1]=c;
     i=n/100;
     i=i%10;
     c=(unsigned char)i;
     chars[2]=c;
     i=n/1000;
     i=i%10;
     c=(unsigned char)i;
     chars[3]=c; // we have only 4 displays, so no more than 4 digits
     i=0;
     while(1) {
          P0_7=0; // turn off all displays temporarily
          c=(unsigned char)i;
          P3=c<<3; // set up the display to be used
          switch(chars[c]) {
               // for each digit determine the 7-segment configuration
               case 0: P1=0xC0; break;
               case 1: P1=0xF9; break;
               case 2: P1=0xA4; break;
               case 3: P1=0xB0; break;
               case 4: P1=0x99; break;
               case 5: P1=0x92; break;
               case 6: P1=0x82; break;
               case 7: P1=0xF8; break;
```

```
                case 8: P1=0x80; break;
                case 9: P1=0x90; break;
            }
            P0_7=1; // turn on display usage
            i++; // move to the next display
            i=i%4;
            }
      return 0;
}
```

Homework:
Write a program that displays a multi-digit number, as above, but ignores non-significant digits. For example, if the number is 698, display 3 is never turned on (in the previous case, is displays value 0).

Lesson 4

Problem: write a C program that reads a key from the keypad and displays it on one of the 7-segment displays. The key on the keypad must have already been pressed when the program starts and the situation on the keypad is not changing during program execution.

The keypad is connected to the microcontroller in a special manner, which leads to a sophisticated read procedure. To detect that a key is pressed, one must first write value 0 on the corresponding keypad row and subsequently read the value on the corresponding keypad column. For example, we can detect that key '6' has been pressed by writing 0 on the second-highest row (i.e., bit 2 on port P0) and testing if value is 1 on the rightmost column (i.e., bit 4 on port P0).
As we do not know in advance which key is pressed (if any), we have to scan the keypad; that is, all rows and columns.

```
#include <8051.h>

unsigned char digits[12]; // stores the 7-segment configuration to be displayed for each key
unsigned char n,i;

unsigned char getCol()
// reads the values on the keypad columns and returns 1 if a key is pressed, 0 otherwise
// if a key is pressed, global variable n gets the corresponding column number (left to right order)
// prerequisites:
// - one and only one keypad row must have value 0
// - global variable n must have value 0
{
     if(P0_6==0) return 1; // iff a key is pressed, the column has value 0
     n++;
     if(P0_5==0) return 1;
     n++;
     if(P0_4==0) return 1;
     n++;
     return 0;
}

int main()
{
     P0_7=0;
     digits[0]=0xF9; // initialization of 7-segment configurations
     digits[1]=0xA4;
     digits[2]=0xB0;
     digits[3]=0x99;
     digits[4]=0x92;
     digits[5]=0x82;
     digits[6]=0xF8;
     digits[7]=0x80;
     digits[8]=0x90;
     digits[9]=0x7F;
     digits[10]=0xC0;
     digits[11]=0x89;
     while(1) {
          n=0;
          P0=0x77; // highest keypad row gets value 0, the rest get value 1
          if(getCol()) break;
          P0=0x7B; // second-highest keypad row gets value 0, and so on
          if(getCol()) break;
```

```
            P0=0x7D;
            if(getCol()) break;
            P0=0x7E;
            if(getCol()) break;
            }
      i=digits[n];
      P3=0x0; // use display 0
      P1=i;
      P0_7=1; // turn on displays
      return 0;
}
```

Homework:
Write a program that shows up to 4 keys simultaneously pressed, on the 4 available displays.

Lesson 5

Problem: write a C program that turns on all LEDs individually, one LED at a time, and then moves to the next etc. Moving from one LED to the next is done at a constant pace.

The most difficult problem here is measuring the time. As program loops do not provide a refined and reliable control of the elapsed time, it is necessary to use the builtin timers (see the 8051 data sheet). Although it is possible to start a timer and then wait in a loop until it reaches the end of its cycle, the more elegant solution is to use interrupts.

```
#include <8051.h>

// run this program with Update Freq. = 1000

volatile unsigned char i; // the keyword is mandatory for variables used in interrupt routines

void timer0() interrupt 1 // interrupt service routine for timer 0
{
      TL0=0; // reload the value corresponding to the desired time constant into the timer
      TH0=4;
      if(i==0x80) // advance the position of the LED to be turned on
            i=1;
      else i<<=1;
      P1=~i;
}

int main()
{
      TR0=0; // disable timer working
      EA=0; // general interrupt disabling
      ET0=1; // enable interrupts on timer 0
      TMOD=0x01; // set timer 0 in mode 1
      TL0=0; // initially load the value corresponding to the desired time constant into the timer
      TH0=4;
      i=1;
      P1=~i;
      EA=1; // general interrupt enabling
      TR0=1; // enable timer working
      // from this moment on, timer 0 may work and generate interrupts periodically
      while(1) {} // we can do anything else in the meantime
      return 0;
}
```

Homework:
Write a program that periodically scans the keypad for pressed keys. The periodic action is performed by using timer interrupts, as above.

Lesson 6

Problem: write a C program that continuously reads the voltage value from the Analogic-Digital Converter (ADC) and displays the number of volts (integer, truncated) on one of the 7-segment displays.

ADC control works as follows:
- To start a conversion cycle, a rising edge must be applied to input WR. To do that, we have to clear bit 6 of port 3, then set it.
- When the conversion cycle, which is quite long, is over, output INTR gets value 0.
- Then, by writing 0 to input RD, the value resulting from the conversion becomes available on the data output (D7-0) and thus can be read by the microcontroller.

```
#include <8051.h>

unsigned char b,c1,c2;
unsigned int t;

int main()
{
	EX0=0;
	while(1) {
		P0_7=0;
		P3=0xFF; // set all bits of P3 to 1, in order to be able to later read the value 0 from INTR
		P3_6=0; // rising edge on WR
		P3_6=1;
		while(P3_2==1) ; // wait until conversion is terminated
		P3_7=0; // issue the RD command
		b=P2;
		P3_7=1;
		c1=b/51; // compute the number of volts (value 255 corresponds to 5V)
		c2=b%51; // also compute the first digit after the decimal separator
		c2=c2/5;
		if(c2==10) c2=9;
		switch(c1) { // send the number of volts to the display's input
			case 0: P1=0xC0; break;
			case 1: P1=0xF9; break;
			case 2: P1=0xA4; break;
			case 3: P1=0xB0; break;
			case 4: P1=0x99; break;
			case 5: P1=0x92; break;
			case 6: P1=0x82; break;
			case 7: P1=0xF8; break;
			case 8: P1=0x80; break;
			case 9: P1=0x90; break;
			}
		P3=0; // select display 0
		P0_7=1; // enable the displays
		for(t=1;t<1000;t++) ; // waiting loop, allowing some time to pass
		}
	return 0;
}
```

We could also choose to display variable `c2` instead of `c1`; that is, the first digit after the decimal separator instead of the number of volts.

Homework:

Improve the program, making it capable of displaying both digits computed above. In order to do that, conversions must be started periodically, through the timer 0 interrupt, just as in the previous lesson. Thus, the code that handles the ADC conversion must be moved into the interrupt service routine, while the displaying part remains into the main function.