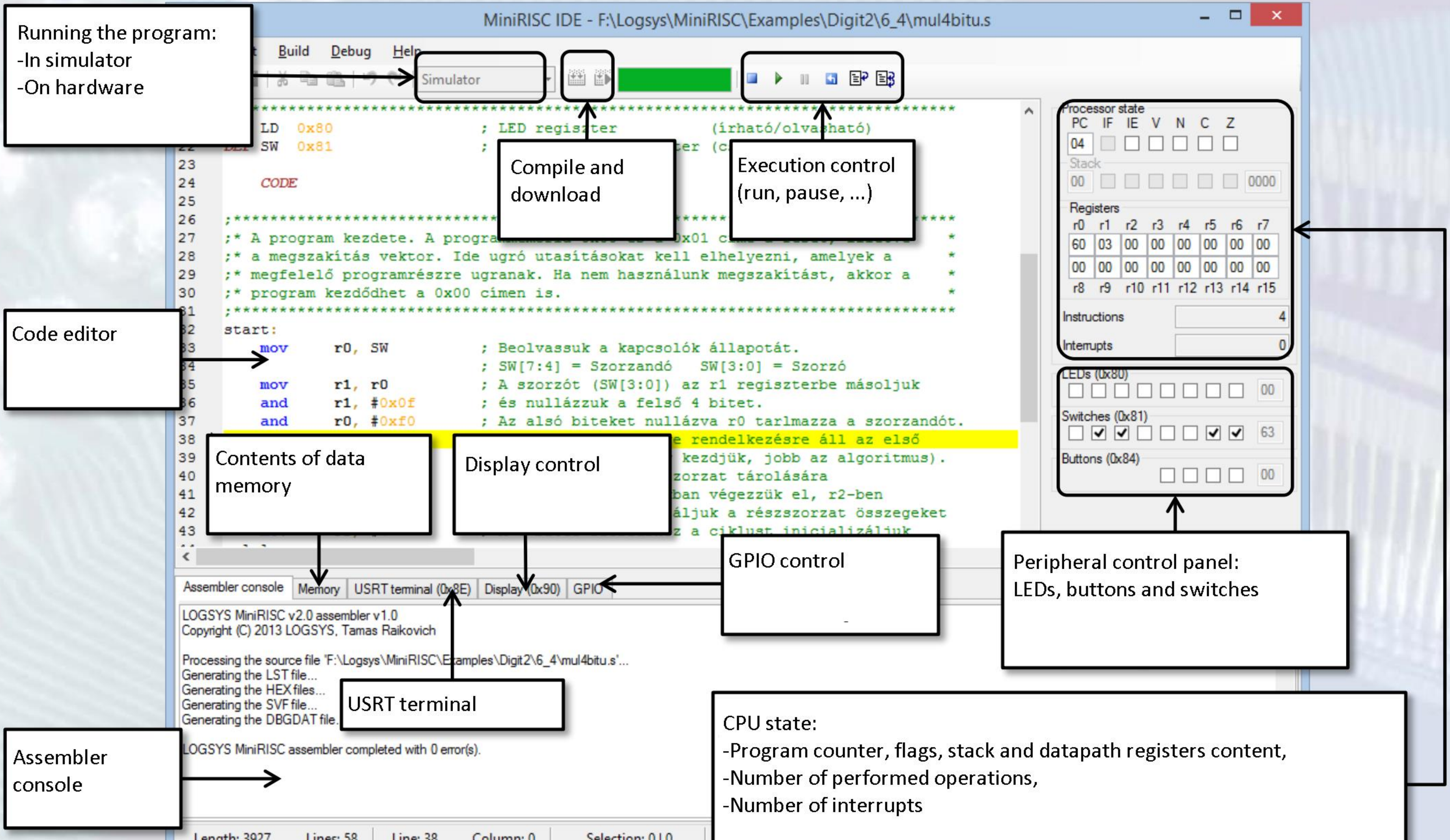
Assembly programming

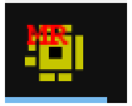
The MiniRISC IDE

- The MiniRISC IDE is used to develop simple Assembly applications.
- After compiling, it is able to upload the code and the Verilog source of the processor together to the FPGA card.
- The integrated simulator helps debugging the software, and development without FPGA.
- The main components of the IDE are summarized on the next slide.



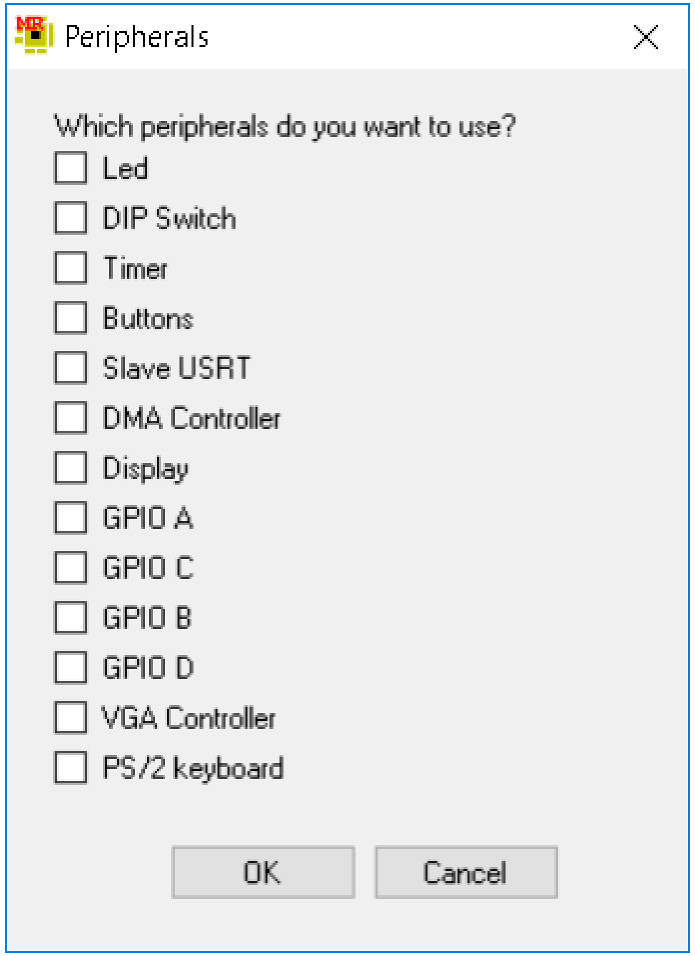
MiniRISC IDE - F:\Logsys\MiniRISC\Examples\Digit2\6_4\mul4bitu.s
Running the program:
-In simulator
-On hardware
Build
Debug
Simulator
Compile and download
Execution control (run, pause, ...)
Code editor
Contents of data memory
Display control
GPIO control
USRT terminal
Assembler console
Peripheral control panel: LEDs, buttons and switches
CPU state:
-Program counter, flags, stack and datapath registers content,
-Number of performed operations,
-Number of interrupts
Processor state
PC IF IE V N C Z
Stack
Registers
Instructions
Interrupts
LEDs (0x80)
Switches (0x81)
Buttons (0x84)
Assembler console
Memory
USRT terminal (0x8E)
Display (0x90)
GPIO
LOGSYS MiniRISC v2.0 assembler v1.0
Copyright (C) 2013 LOGSYS, Tamas Raikovich
Processing the source file 'F:\Logsys\MiniRISC\Examples\Digit2\6_4\mul4bitu.s'...
Generating the LST file...
Generating the HEX files...
Generating the SVF file...
Generating the DBGDAT file.
LOGSYS MiniRISC assembler completed with 0 error(s).

Task 1

- In the first task we are going to indicate the state of the switches on the LED panel.
- Launch the MiniRISC IDE 
- Select File -> New…
- Create a new source file with name Lab11 and save it on the D drive under the d:\Lab11 directory (create it if not exists)

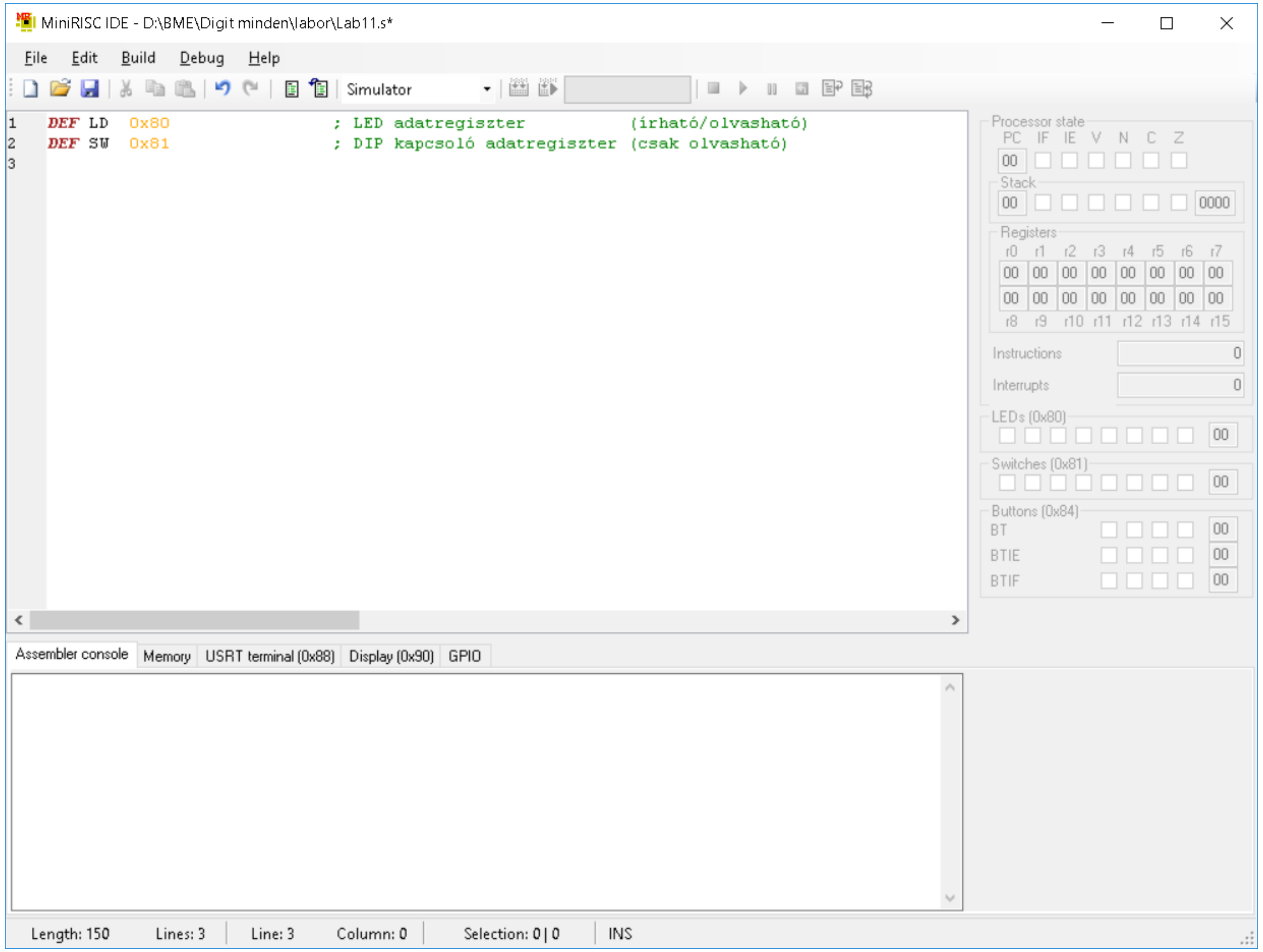
Task 1

- The following popup window appears, asking for the peripherals in the project.
- Select the Led and the DIP switch
- Click OK



Task 1

- As a result, you should see this:



Task 1

- Two lines have been added to your code:

```
DEF LD  0x80
DEF SW  0x81
```

- Now you can call the leds LD and the switches SW in your program, the compiler will substitute the correct addresses (0x80 and 0x81) when you compile the code.
- We want to move the state of the switches to the leds.
- Since we are working with a load/store architecture, first we have to move the content into a register of the datapath, then move it to the LEDs.
- In addition, we want to do it constantly, in an infinite loop.

Task 1

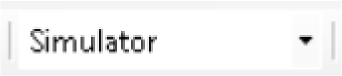
- Add the following code:

```
DEF LD   0x80
DEF SW   0x81

Start:
MOV r0, SW
MOV LD, r0
jmp Start
```

- First we copy the state of the switches into the r0 register, then we move it to the register of the LEDs.
- After the data moving operations, we jump back to the Start label, the label that contains the address of the first MOV operation.
- This way we can execute the code in an infinite loop, and monitor the state of the switches constantly.

Task 1

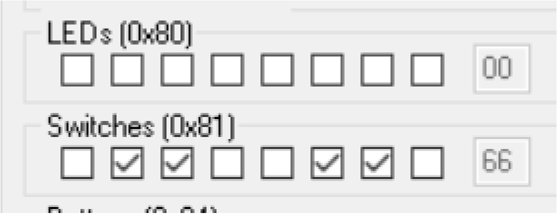
- After adding the code, save it. 
- Select the simulator in the bar above the editor. 

- Select compile 
- Select download 
- After download, the execution stops on the first line of the code:

```
Start:
MOV r0, SW
MOV LD, r0
jmp Start
```

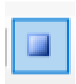
Task 1

- Modify the state of the switches on the right panel:



- Press Step (or F10): 
- After executing both MOV instuctions, check the value of the LEDs:

Task 1

- Stop debugging by pressing the Stop button 
- Select the FPGA device instead of the simulator (LDC XXX).
- Compile and run the code, check the functionality.
- On the next slides we summarize the instructions and their effect on the flags.

Data movement

- Data movement operations do not set the flags.

`MOV rX, maddr`	rX ← DMEM[maddr]
`MOV rX, (rY)`	rX ← DMEM[rY]
`MOV maddr, rX`	DMEM[maddr] ← rX
`MOV (rY), rX`	DMEM[rY] ← rX
`MOV rX, #imm`	rX ← imm
`MOV rX, rY`	rX ← rY

Arithmetic operations

- Arithmetic operations set all flags (N, V, Z, C)

`ADD rX, #imm`	rX ← rX + imm
`ADD rX, rY`	rX ← rX + rY
`ADC rX, #imm`	rX ← rX + imm + C
`ADC rX, rY`	rX ← rX + rY + C
`SUB rX, #imm`	rX ← rX - imm
`SUB rX, rY`	rX ← rX - rY
`SBC rX, #imm`	rX ← rX - imm - C
`SBC rX, rY`	rX ← rX - rY - C
`CMP rX, #imm`	rX - imm
`CMP rX, rY`	rX - rY

Logic operations

- Logic operations set Z and N flags

`AND rX, #imm`	rX ← rX & imm
`AND rX, rY`	rX ← rX & rY
`OR rX, #imm`	rX ← rX | imm
`OR rX, rX`	rX ← rX | rY
`XOR rX, #imm`	rX ← rX ^ imm
`XOR rX, rY`	rX ← rX ^ rY
`TST rX, #imm`	rX & imm
`TST rX, rY`	rX & rY

Swap and shift instructions

- Swap and shift instruction sets flags Z and N. In addition, in shift instructions the LSB/MSB is moved into the carry flag.

`SWP rX`	rX ← {rX[3:0], rX[7:4]}
`SL0 rX`	rX ← {rX[6:0], 0}
`SL1 rX`	rX ← {rX[6:0], 1}
`SR0 rX`	rX ← {0, rX[7:1]}
`SR1 rX`	rX ← {1, rX[7:1]}
`ASR rX`	rX ← {rX[7], rX[7:1]}

Rotate instructions

- Set flags Z and N, the MSB/LSB is moved into the carry.

`ROL rX`	rX ← {rX[6:0], rX[7]}
`ROR rX`	rX ← {rX[0], rX[7:1]}
`RLC rX`	rX ← {rX[6:0], C}
`RRC rX`	rX ← {C, rX[7:1]}

Jump instructions

- Jump and conditional jump:

JMP *paddr* / **(rY)**	PC ← *paddr* / **rY**
JZ *paddr* / **(rY)**	PC ← *paddr* / **rY**, if Z=1
JNZ *paddr* / **(rY)**	PC ← *paddr* / **rY**, if Z=0
JC *paddr* / **(rY)**	PC ← *paddr* / **rY**, if C=1
JNC *paddr* / **(rY)**	PC ← *paddr* / **rY**, if C=0
JN *paddr* / **(rY)**	PC ← *paddr* / **rY**, if N=1
JNN *paddr* / **(rY)**	PC ← *paddr* / **rY**, if N=0
JV *paddr* / **(rY)**	PC ← *paddr* / **rY**, if V=1
JNV *paddr* / **(rY)**	PC ← *paddr* / **rY**, if V=0

Task 2

- Create a new source file. Use the LEDs and the switches. Move the two's complement of the switches to the LEDs.
- Step 1: read switches to a register
- Step 2: determine the two's complement
- Step 3: move the result to the LEDs.

Task 2

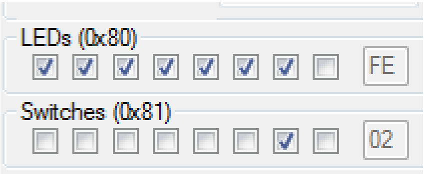
- The algorithm to determine the two’s complement is the following:
  - Invert the bits
  - Add 1
- To invert the bits, you can XOR the content of the register with 0xFF, or subtract it from 0xFF.
- In the implementation, we will use the second solution.

Task 2

- Add the following code:

```
Start:
mov r0, SW
mov r1, #0xFF
sub r1, r0
add r1, #1
mov LD, r1
jmp Start
```

- Check the code using the simulator (compile, download and execute step-by-step):



Task 2

- This task can be implemented using the XOR instruction. Modify the code to determine the two's complement using XOR.

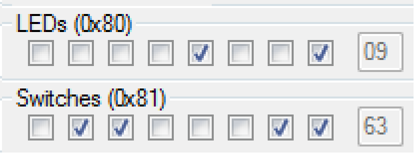
Task 3

- Create a new source. Add the switches and the LEDs.
- Create an adder that adds the lower and upper 4 bits of the value set on the switches. Indicate the result on the switches.
- The main steps are the following:
  - Reading the switches
  - Determining the first operand using the AND operation
  - Determining the second operand using AND
  - Shifting the second operand to the right four times
  - Addition
  - Indicating the result

Task 3

```
Start:
mov r0, SW
mov r1, r0
and r1, #0b00001111 ;lower 4 bits
mov r2, r0
and r2, #0b11110000 ;upper 4 bits
swp r2 ;swap upper and lower 4 bits
add r2, r1
mov LD, r2
jmp Start
```

- Test it on the switches, e.g. 6+3=9



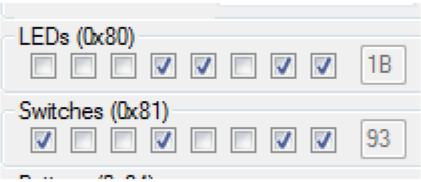
Task 4

- Modify the previous code to multiply the two numbers. There is no multiplication instruction, use addition and subtraction.
- Algorithm: use a loop. Increment a register with the first operand and decrease the value of the second operand by one. Do this until the second operand becomes 0.

Task 4

```
Start:
mov r0, SW
mov r1, r0
and r1, #0b00001111 ;lower 4 bits
mov r2, r0
and r2, #0b11110000 ;upper 4 bits
swp r2 ;swap upper and lower 4 bits
mov r0, #0 ; result register, init to 0
Loop:
add r2, #0
jz Loop_end ; if 0, we are done
add r0, r1
sub r2, #1
jmp Loop
Loop_end:
mov LD, r0
jmp Start
```

- Test it, e.g. 6x9=27=0x1B because 0x1B = 1x16+11x1



Task 5

- Create a new source. Add the LEDs.
- Initialize the data memory with 10 numbers, from address 50
- Read the numbers, determine their sum and send the result to the LEDs
- We will use the r0 register to store the address of the current element
- Register r1 will contain the actual element
- The sum will be stored in r2
- We need another register (r3) for the loop variable

Task 5

```
DATA
ORG 50
DB 1, 2, 3, 4, 5, 6, 7, 8, 9, 22

CODE
Start:
; initialization of registers
mov r0, #50 ; address
mov r2, #0 ; sum
mov r3, #10 ; loop
Loop:
mov r1, (r0) ; indirect memory address!
add r2, r1
add r0, #1 ; incrementing the memory address
sub r3, #1
jz Loop_end
jmp Loop
Loop_end:
mov LD, r2
jmp Start
```

- Test the code, 1+2+3+4+5+6+7+8+9+22=67=0x43: