Programming in Edulent assembly language

prof. dr Ivan Mezei

Novi Sad, 2022.

Content

EDULENT REGISTERS

All registers are 8-bit. Microprocessor has 11 registers. Only two are program accessible to user:

1) *accumulator(A)* – general purpose register used with most instructions. Its output is connected to ALU input, and to 8-bit bus.
2) *address pointer(AP)* – contains address with register indirect addressing, and with direct and immediate addressing for ADD, SUB and MOV instructions. It is also connected to ALU and data bus.

To communicate with peripherals 2 registers are to be used:

3) *out register (OUT)* – transmits data to peripheral
4) *in register (IN)* – receives data from peripheral

Program flow control registers:

5) *stack pointer (SP)* – points to the top of stack memory that empty at the beginning. Stack memory fills from the last address (0x7F) backwards.
6) *program counter (PC)* – points to memory location from which instruction or operand is loaded.
7) *flag register* – consists of 2 flags: carry flag indicating carry and zero flag indication that result of an arithmetic operation is zero.

Other registers can't be accessed from program:

8) *instruction register (IR)* – used with all instruction to contain fetched instruction.
9) *result register (R)* – contain temporary result from ALU.
10) *memory address register (MA)* – directly connected to memory address bus.
11) *memory data register (MD)* – data buffer between memory and CPU.

The following figure depicts registers, bus connections and microprogrammable control unit. Memory is 128 bytes and contain program code, variables and stack memory.

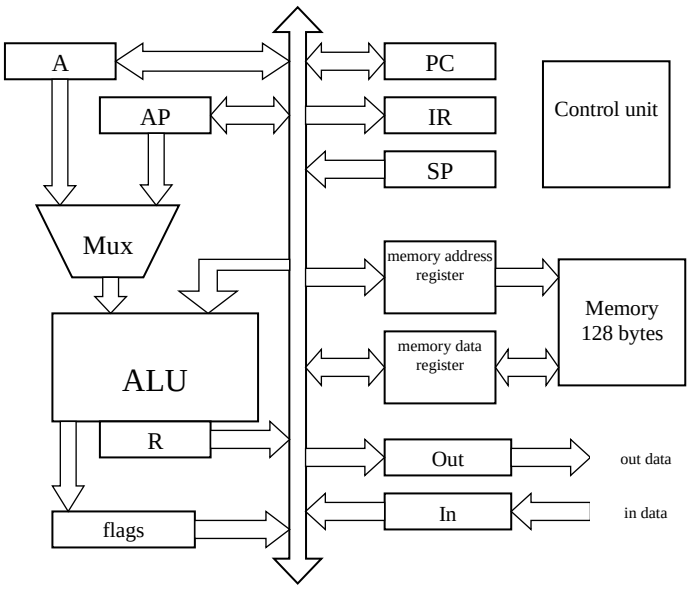


Fig. 1

INSTRUCTIONS

Addressing modes

Edulent has 4 addressing modes:
- direct,
- immediate,
- register indirect, and
- predecrement/postincrement.

Direct addressing mode is used with data transfer instructions from memory to A or AP register (for example MOV A, VAR1) and data transfer instructions from A or AP register to memory (for example MOV RES1, AP), with arithmetic-logic operations where one operand is in A, while other is in memory.

Immediate mode is used with data transfer instructions where A or AP register contain constant value, with arithmetic-logic operations where one operand is in A, while other is constant, with jump instructions and with procedure calls.

Register indirect mode assumes that address is in AP register. It is used with data transfer operations from memory to A, with arithmetic-logic operations where one operand is in A, while other is in memory pointed by AP register.

Predecrement/postincrement mode is used to push A or AP register on stack, or to pop data from stack to A or AP register.

Note that instructions using immediate and direct addressing modes take **two** 8-bit memory locations (2 bytes), since other byte contain address or constant. For instructions that use indirect or predecrement/postincrement modes one byte is enough.

Instruction set

Edulent has 39 instructions (including different addressing modes). They can be divided into following groups:
- data transfer instructions,
- arithmetic,
- logic,
- jump instructions,
- procedure calls,
- in/out instructions, and
- program flow control instructions.

Data transfer instructions are:

Syntax	Description	Examples	Affect flag
`MOV A, address`	transfers value from memory location to accumulator; `address` is memory location defined in DATA section	`MOV A, VAR1` `MOV A, RES`	-
`MOV AP, address`	transfers value from memory location to AP register; `address` is memory location defined in DATA section	`MOV AP, P1` `MOV AP, VAR1`	-
`MOV A, const`	Accumulator value is `const`	`MOV A, 1` `MOV A, 0X7F`	-

`MOV AP, const`	AP register value is `const`; const may be address of memory location defined in DATA section(using @ sign; see example)	`MOV AP, 0xA`
		`MOV AP, 2`
		`MOV AP, @VAR1`
			-

`MOV address, A`	Transfer accumulator content to memory location; `address` is memory location defined in DATA section	`MOV RES1, A` `MOV P1, A`	-
`MOV address, AP`	Transfer AP register content to memory location; `address` is memory location defined in DATA section	`MOV RES1, AP` `MOV P1, AP`	-
`MOV A, [AP]`	Transfer memory content from address pointed by AP to accumulator		-
`MOV A, [SP+]`	Pop data from stack to accumulator and increment SP		-
`MOV AP, [SP+]`	Pop data from stack to AP register and increment SP		-
`MOV [-SP], A`	Decrement SP and push data from accumulator to stack		-
`MOV [-SP], AP`	Decrement SP and push data from AP register to stack		-

Arithmetic instructions are:

Syntax	Description	Examples	Affect flag
`ADD A, address`	Adds memory location value with accumulator; `address` is memory location defined in DATA section	`ADD A, VAR1` `ADD A, RES`	carry, zero
`ADD A, const`	Adds accumulator and `const`	`ADD A, 1` `ADD A, 0X1F`	carry, zero
`ADD AP,const`	Adds AP and `const`	`ADD AP, 0` `ADD AP, 0XFF`	carry, zero
`ADD A, [AP]`	Adds accumulator and memory location value pointed by AP		carry, zero
`SUB A, address`	Subtracts memory location value from accumulator value; `address` is memory location defined in DATA section	`SUB A, Z1` `SUB A, RES`	carry, zero
`SUB A, const`	Subtracts const from accumulator	`SUB A, 1` `SUB A, 0X2`	carry, zero
`SUB AP, const`	Subtracts const from AP register	`SUB AP, 1` `SUB AP, 0X2`	carry, zero
`SUB A, [AP]`	Subtracts memory location value pointed by AP from accumulator value		carry, zero

Logic instructions are:

Syntax	Description	Examples	Affect flag
`NOT`	Complement accumulator value		zero, clears carry
`OR address`	Logical OR of accumulator and memory location content; `address` is memory location defined in DATA section	`OR VAR1` `OR RES`	zero, clears carry
`OR const`	Logical OR of accumulator and `const`	`OR 11` `OR 0X12`	zero, clears carry
`OR [AP]`	Logical OR of accumulator and memory location content pointed by AP		zero, clears carry
`AND address`	Logical AND of accumulator and memory location content; `address` is memory location defined in DATA section	`AND Z1` `AND RES`	zero, clears carry
`AND const`	Logical AND of accumulator and `const`	`AND 1` `AND 0X2`	zero, clears carry
`AND [AP]`	Logical AND of accumulator and memory location content pointed by AP		zero, clears carry
`XOR address`	Logical XOR of accumulator and memory location content; `address` is memory location defined in DATA section	`XOR P` `XOR RES`	zero, clears carry
`XOR const`	Logical XOR of accumulator and `const`	`XOR 1` `XOR 0X2`	zero, clears carry
`XOR [AP]`	Logical XOR of accumulator and memory location content pointed by AP		zero, clears carry
`SHR`	Shift accumulator value by one bit right. (**shifted bit goes to carry!**)		zero, carry

Jump instructions are:

Syntax	Description	Examples	Affect flag

`JMP label`	Unconditional jump to label	`JMP L1`
		`JMP LAB1`
		`JMP LRES`
			-

`JZ label`	Jump to label if zero is set	`JZ L2`
		`JZ LABELA2`
		`JZ L`
			-

`JC label`	Jump to label if carry is set	`JC L12`
		`JC LINC`
		`JC LDEC`
			-

Procedure call instructions:

Syntax	Description	Examples	Affect flag

`CALL procedure`	Procedure call	`CALL PROC`
		`CALL P1`
		`CALL INC`
			-

`RET`	Return from procedure to main program		-

In/out instructions are:

Syntax	Description	Examples	Affect flag
`IN`	Transfer IN register content to accumulator		-
`OUT`	Transfer accumulator content to OUT register		-

Program control instructions are:

Syntax	Descriptions	Example	Affect flag
`NOP`	No operation		-
`END`	Program end		-

Detailed instruction set description

In the following section detailed instruction set description is given. Every instruction is a set of microinstructions that describe affect on registers and memory. Description is done using *Register Transfer Notation (RTN)* language. Common to all instruction is that within first three periods of clock instruction fetch is done. Two-byte instructions has address or constant fetch in next 2 clock cycles. Having finished fetch phase instruction execution follows.

Instruction fetch

Step	RTN
T0	MA ← PC
T1	MD ← M[MA] : PC ← PC+1
T2	IR ← MD

Address or constant fetch

Step	RTN
T3	MA ← PC
T4	MD ← M[MA] : PC ← PC+1

Data transfer instructions

`MOV A, address` (0x11, address<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← M[MA]
T7	A ← MD

`MOV AP, address` (0x13, address<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← M[MA]
T7	AP ← MD

`MOV A, const` (0x19, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Constant fetch
T5	A ← MD

`MOV A, [AP]` (0x14)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← AP
T4	MD ← M[MA]
T5	A ← MD

`MOV AP, [SP+]` (0x1E)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← SP
T4	MD ← M[MA]: SP ← SP+1
T5	AP ← MD

`MOV address, AP` (0x23, address<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← AP
T7	M[MA] ← MD

`MOV [-SP],AP` (0x2E)

Step	RTN
T0-T2	Instruction fetch
T3	SP ← SP – 1
T4	MA ← SP
T5	MD ← AP
T6	M[MA] ← MD

`MOV AP, const` (0x1B, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Constant fetch
T5	AP ← MD

`MOV A, [SP+]` (0x1C)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← SP
T4	MD ← M[MA]: SP ← SP+1
T5	A ← MD

`MOV address, A` (0x21, address<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← A
T7	M[MA] ← MD

`MOV [-SP],A` (0x2C)

Step	RTN
T0-T2	Instruction fetch
T3	SP ← SP – 1
T4	MA ← SP
T5	MD ← A
T6	M[MA] ← MD

NOTE: Data transfer instructions don't affect flags.

Arithmetic instructions

`ADD A, address` (0x31, address<7..0>)
`SUB A, address` (0x41, address<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← M[MA]
T7	R ← A + MD R ← A – MD
T8	A ← R

`ADD A, const` (0x39, const<7..0>)
`SUB A, const` (0x49, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Constant fetch
T5	R ← A + MD R ← A – MD
T6	A ← R

`ADD AP, const` (0x3B, const<7..0>)
`SUB AP, const` (0x4B, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Constant fetch
T5	R ← AP + MD R ← AP – MD
T6	AP ← R

`ADD A, [AP]` (0x34)
`SUB A, [AP]` (0x44)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← AP
T4	MD ← M[MA]
T5	R ← A + MD R ← A – MD
T6	A ← R

NOTE: All arithmetic instructions sets zero flag if result of operation is zero. With subtract instruction carry flag is set if lager number is subtracted from smaller. Add instruction sets carry if result is more then 8 bit.

Logic instruction

`NOT` (0x50)

Step	RTN
T0-T2	Instruction fetch
T3	R ← ¬A
T4	A ← R

`OR address` (0x61, address<7..0>)
`AND address` (0x71, address <7..0>)
`XOR address` (0x81, address <7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	MA ← MD
T6	MD ← M[MA]
T7	$R \leftarrow A \begin{matrix} \vee \\ \wedge \\ \oplus \end{matrix} MD$
T8	A ← R

`OR const` (0x69, const<7..0>)
`AND const` (0x79, const<7..0>)
`XOR const` (0x89, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Constant fetch
T5	$R \leftarrow A \begin{matrix} \vee \\ \wedge \\ \oplus \end{matrix} MD$
T6	A ← R

`OR [AP]` (0x64)
`AND [AP]` (0x74)
`XOR [AP]` (0x84)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← AP
T4	MD ← M[MA]
T5	$R \leftarrow A \begin{matrix} \vee \\ \wedge \\ \oplus \end{matrix} MD$
T6	A ← R

`SHR` (0x90)

Step	RTN
T0-T2	Instruction fetch
T3	CZ<1>←A<0> : R ← 0 # A<7..1>
T4	A ← R

NOTE: All logic instructions sets zero flag if result of operation is zero. All logic instructions (except shift instructions) clears carry flag. `SHR` shifts LSB to carry, while MSB becomes zero. SHL shifts MSB to carry, while LSB becomes zero.

Jump instructions

`JMP label` (0xA1, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	PC ← MD

Unconditional jump.

`JZ label` (0xA5, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	CZ<1..0>=1 → PC ← MD

Jump if zero.

`JC label` (0xA9, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	CZ<1..0>=2 → PC ← MD

Jump if carry.

I/O instructions

`IN` (0xD0)

Step	RTN
T0-T2	Instruction fetch
T4	A ← IN

`OUT` (0xE0)

Step	RTN
T0-T2	Instruction fetch
T4	OUT ← A

I/O instructions don't affect flags.

Procedure instructions

`RET` (0xB0)

Step	RTN
T0-T2	Instruction fetch
T3	MA ← SP
T4	MD ← M[MA]: SP ← SP + 1
T5	PC ← MD

`CALL procedure` (0xC1, const<7..0>)

Step	RTN
T0-T2	Instruction fetch
T3-T4	Address fetch
T5	AP ← MD : SP ← SP - 1
T6	MA ← SP
T7	MD ← PC
T8	M[MA] ← MD
T9	PC ← AP

NOTE: `CALL` instruction use AP register so content having finished this instruction previous AP value is lost! Flags is not affected by these instructions. `RET` assumes that returning address is on top of the stack.

Other instructions

`NOP` (0x00)

Step	RTN
T0-T2	Instruction fetch
T3	no operation

Doesn't affect flags.

PROGRAM STRUCTURE

Program skeleton

Edulent assembly language program is:

```
PROGRAM 'Name'
DATA
      < variables >
      ...
ENDDATA
CODE
      < main program instructions >
      ...
END   ; end of main program
PROCEDURE <name_first_procedure>
      < first procedure instructions >
      ...
RET   ; return to main program
ENDPROCEDURE
...
PROCEDURE < name_last_procedure >
      < last procedure instructions >
      ...
RET
ENDPROCEDURE
ENDPROGRAM
```

Additional rules

Every program must contain reserved words PROGRAM, DATA, ENDDATA, CODE and ENDPROGRAM. Reserved words PROCEDURE and ENDPROCEDURE are used if procedures are used in program. Procedure name must begin with letter.

Number of variables are limited to 16. Variable name must begin with letter. Reserved word `DB` is used in DATA section to assign values to variables.(for example VAR1 DB 0x1A or RES DB 12)

Program labels must begin with letter L and finished with colon (for example L123:). Program labels must be written in front of instructions (L123: MOV A, 1).

Constants can be written in *decimal* or *hexadecimal* notation with leading 0x or 0X (for example MOV A, 0x1F). Comment can be placed after semicolon sign (for example MOV AP, 0x7F ; put 127 in AP).

PROGRAM EXAMPLES

Several programs depicting instructions set usage follows.

```
PROGRAM"Data transfer ins."
DATA
      VAR1 DB 0X9
      VAR2 DB 0XF
      RES1 DB 0X0
      RES2 DB 0X0
      RES3 DB 0X0
ENDDATA
CODE
      MOV A, 0x1
      MOV A, VAR1
      MOV AP, @VAR1
      MOV A, [AP]
      MOV AP, 0XAA
      MOV RES1, AP
      MOV AP, VAR2
      MOV RES2, AP
      MOV RES3, A
      END
ENDPROGRAM
```

```
PROGRAM "Jumps and calls"
DATA
      RES1 DB 0X0
ENDDATA
CODE
      MOV A, 0X3
L1:   CALL DEC    ;decrement
      JZ L2
      JMP L1
L2:   CALL INC    ;increment
      MOV RES1, A
      END
PROCEDURE INC
      ADD A,0X1
      RET
ENDPROCEDURE
PROCEDURE DEC
      SUB A,0X1
      RET
ENDPROCEDURE
ENDPROGRAM
```

```
PROGRAM "Add ins."
DATA
      VAR1 DB 0X1
      VAR2 DB 0XA
      RES1 DB 0X1
      RES2 DB 0X0
      RES3 DB 0X0
      RES4 DB 0X0
ENDDATA
CODE
      MOV A, 0X1
      ADD A, 0X1
      MOV RES1, A
      MOV AP, 0X5
      ADD AP, 0X3
      MOV RES2, AP
      ADD A, VAR1
      MOV RES3, A
      MOV AP,@VAR1
      ADD A, [AP]
      MOV RES4, A
      END
ENDPROGRAM
```

```
PROGRAM "Carry and zero"
DATA
ENDDATA
CODE
      MOV A, 0XFF
      ADD A, 0X1
      ADD A, 0XF
      END
ENDPROGRAM
```

```
PROGRAM "Stack"
DATA
      VAR1 DB 0X1
      VAR2 DB 0XA
      RES1 DB 0X1
      RES2 DB 0X0
      RES3 DB 0X0
      RES4 DB 0X0
ENDDATA
CODE
      MOV A, 0X1
      ADD A, 0X1
      MOV [-SP], A
      MOV RES1, A
      ADD A, VAR1
      MOV [-SP], A
      MOV RES2, A
      MOV AP,@VAR1
      ADD A, [AP]
      MOV [-SP], A
      MOV RES3, A
      MOV A, [SP+]
      MOV AP, @VAR2
      MOV [-SP], AP
      ADD A, VAR1
      MOV RES4, AP
      MOV [-SP], AP
      END
ENDPROGRAM
```

```
PROGRAM "Logic ins."
DATA
      VAR1 DB 0X1
      VAR2 DB 0XA
      RES1 DB 0X0
      RES2 DB 0X0
      RES3 DB 0X0
      RES4 DB 0X0
ENDDATA
CODE
      MOV A, 0X1
      NOT
      MOV RES1, A
      AND 0XD
      MOV RES2, A
      OR 0X2
      MOV RES3, A
      XOR 0X3
      SHR
      MOV RES4, A
      END
ENDPROGRAM
```

What does compiler do?

Compiler is being used to compile assembly language program code into machine language. Table at the end of this document contains binary equivalent of every instruction. Without getting deeper into understanding of compiler operations, memory layout (written in .hex file) will be presented. Memory has 128 bytes, ranging 0 – 0x7F. Starting from address 0 program code begins. It starts with first instruction after reserved word CODE and ends with 0x02 (binary code of END) or with 0xB0 (binary code of RET, if there where any procedures in program). Note that some instructions take one byte, while others take two. If instruction takes two bytes, then second byte contain address or constant (i.e. operand).

To illustrate previous notes look at the following example of program fragment:

...

mov a, 1	0x19
	0x1
add a, 0xF	0x39
	0xF
shr	0x90
and 0xa	0x79
	0xA

```
and [ap]        0x74
call proc       0xC1
                0x4A
...
```

This example has 4 two-byte instructions and 2 one-byte.

After program code section (containing main program and procedure(s)), DATA sections begins where variables are defined followed by 0xFF values until end of memory.

Stack memory is filled from last memory location (0x7F) and is empty at the beginning.

EDULENT SOFTWARE

Edulent software (see Fig. 1) is used to visualize program execution. To do it follow:

1. Write program in editor and save with .asm extension,
2. As you type, it is automatically compiled. If there is any error it appears in status bar.
3. Execute whole program using *Run* button, or do instruction stepping using *Step* button, or do clock cycle based execution using *Clock* button.

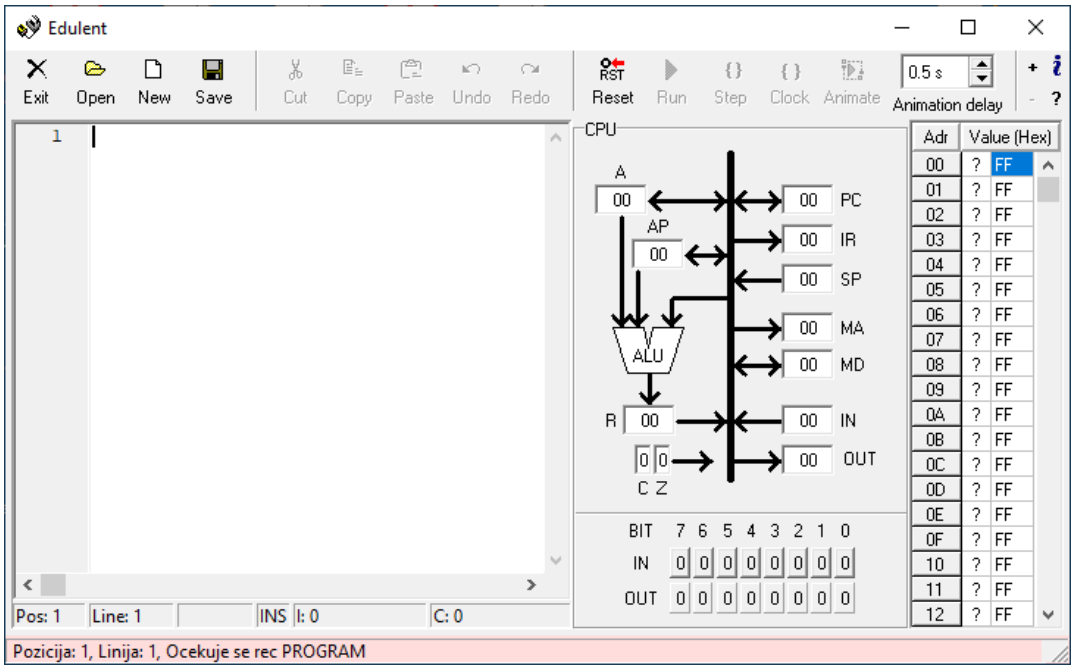


Fig 1

Binary multiplication algorithm implementation

Introduction

In this lab multiplication of two binary numbers should be implemented. First number is 16-bit and second is 8-bit. Using implemented multiplication, factorial of one of the first 8 numbers is to be calculated.

To illustrate algorithm look at decimal multiplication:

```
    456
  x 123
  ----
   1368    =  456 x 3 x 10^0
   9120    =  456 x 2 x 10^1
  +45600   =  456 x 1 x 10^2
  -------
  56088
```

It is obvious that multiplication is done by successive addition.

Similarly, binary multiplication:

```
   1100    (X)
 x 0110    (Y)
 ------
   0000   =  1100 x 0 x 2^0
  11000   =  1100 x 1 x 2^1
 110000   =  1100 x 1 x 2^2
+0000000  =  1100 x 0 x 2^3
---------
 1001000
```

If we annotate multiplicands as X and Y, it can be seen that X is shifted left by one bit and added to result if appropriate bit of Y is not zero. If the bit is zero that step can be skipped. Based on previous illustrations it can be concluded that for binary multiplication is needed:

- determine if appropriate bit of Y is one or zero,
- if it is zero skip to next bit,
- if it is one, shift X *left* and add to result.

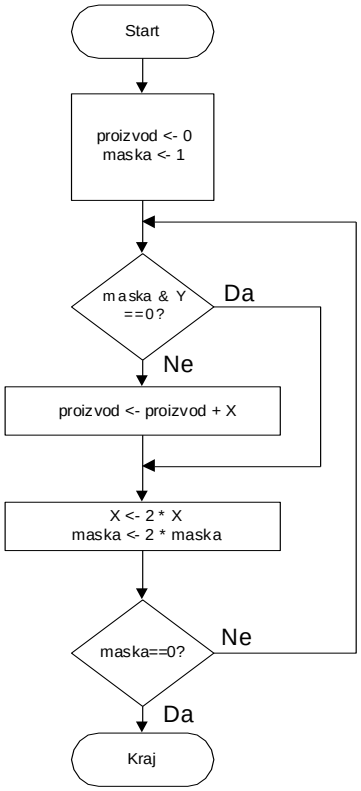
Shift left by one bit is the same as multiply by 2.

Multiplication algorithm is shown on following figure.

Lab assignment

1) Implement algorithm for binary multiplication (16-bit by 8-bit defined in DATA section). For first number allocate two bytes – one for MSB byte and other for LSB byte. Put result in two consecutive locations. Pay attention to carry after addition of LSB bytes! **Algorithm is to be realized as procedure.**

2) Using multiplication procedure, make main program to calculate factorial of one of the first 8 natural numbers. This number is given in DATA section. Final result push on stack (both MSB and LSB bytes).



Proposed DATA section is:

```
DATA
      msb_num1   db 0
      lsb_num1   db 0
      num2       db 0
      msb_res    db 0
      lsb_res    db 0
      mask       db 1
      num        db 4
ENDDATA
```

It takes 7 memory locations - 2 for first number, 1 for second number, 2 for result, 1 for mask and 1 for input number which factorial is calculated.

Addenum

Two-byte instructions		One-byte instructions	
		NOP	0x00
		END	0x02
MOV A, address	0x11	MOV A, [AP]	0x14
MOV AP, address	0x13	MOV A, [SP+]	0x1C
MOV A, const	0x19	MOV AP, [SP+]	0x1E
MOV AP, const	0x1B		
MOV address, A	0x21	MOV [-SP], A	0x2C
MOV address, AP	0x23	MOV [-SP], AP	0x2E
ADD A, address	0x31	ADD A, [AP]	0x34
ADD A, const	0x39		
ADD AP,const	0x3B		
SUB A, address	0x41	SUB A, [AP]	0x44
SUB A, const	0x49		
SUB AP, const	0x4B		
		NOT	0x50
OR address	0x61	OR [AP]	0x64
OR const	0x69		
AND address	0x71	AND [AP]	0x74
AND const	0x79		
XOR address	0x81	XOR [AP]	0x84
XOR const	0x89		
		SHR	0x90
CALL procedure	0xC1	RET	0xB0
		IN	0xD0
		OUT	0xE0
JMP label	0xA1		
JZ label	0xA5		
JC label	0xA9		

Note: in case of two-byte instructions, second byte is address or constant. So `address`, `const`, `label` and `procedure` will be compiled into appropriate binary number as second byte.