UNIVERSITY OF PRIMORSKA
Faculty of Mathematics, Natural Sciences and Information Technologies
Department of Information Sciences and Technology

Systems 1

Tutorials in assembly language

Author: Domen Šoberl

2022-2023

Contents

1 Numbers . . .

1.1 Converting between number systems

Convert the following decimal numbers first to 16-bit binary and then to hexadecimal representation:

a) 67 b) 49155 c) 65534

Convert the following 16-bit hexadecimal numbers first to binary and then to decimal representation:

a) 1000 b) ABCD c) FFFF

1.2 Number spaces

How many different numbers can we write with this many bits?

a) 8 b) 10 c) 16 d) 20 e) 32

How many bits do we need, to address:

a) 14 B b) 64 KB c) 1 MB d) 100 MB e) 1 GB

Suppose we have a 16-bit addressing space. Divide this space into 4 blocks of equal sizes. Write down the first and the last address of each block. *We will write hexadecimal addresses with the prefix* `0x`.

How many addresses are between `0x1000` and `0x2000` including the starting and the ending address?

We want to store 1000 numbers, each number at its own address, starting with the address `0x0200`. What is the address of the last number?

1.3 Computer arithmetic

Calculate the sums of hexadecimal numbers:

a) A + B b) A781 + 1942 c) A000 + 7000

Suppose we operate with 16-bit numbers only. What is then the result of addition A000 + 7000?

In the 16-bit number space let $x =$ `0xA123`. Which number in that space is equivalent to $-x$, i.e. for which number $y = -x$ it holds $x + y = 0$?

2 Introduction to assembly language

Task 2.1 Explain the meaning and the function of all the registers of our simulated computer system.

Solution:

A, B, C, D	General purpose registers for performing arithmetic logic operations.
IP	*Instruction pointer* - memory location of the next instruction.
SP	*Stack pointer* - memory location of the top of the stack
SR	*Status register* - all status flags

Status flags:

M	*Interrupt mask* - globally enables/disables interrupts.
C	*Carry* - set when an overflow occurs after an arithmetic operation.
Z	*Zero* - set when the result of the arithmetic operation is 0.
F	*Fault* - set when there an error occurs during program execution.
H	*Halt* - set after the program execution has stopped.

Task 2.2 Compile and run the following assembly program:

```
MOV A, 0x10
ADD A, 10
HLT
```

Write down and explain the compiled machine code. Trace the program step by step and explain its behavior.

Solution:

Compiler produced the following machine code: `06 00 00 10 14 00 00 0A 00`.
Explanation:

```
MOV [06] A [00], 0x10 [00 10]
ADD [14] A [00], 10 [00 0A]
HLT [00]
```

The program moves hexadecimal number 10 to the 16-bit register A and adds to it decimal number 10.

Task 2.3 Move $AL \leftarrow 120$ and $BL \leftarrow 180$. Do the 8-bit and the 16-bit addition of those values. Explain the different outcomes.

Solution:

```
; 8-bit addition          ; 16-bit addition
MOVB AL, 120              MOVB AL, 120
MOVB BL, 180              MOVB BL, 180
ADDB AL, BL               ADD A, B
HLT                       HLT
```

8-bit addition results in $A = 44$ and 16-bit in $A = 300$. With 8-bit addition the overflow occurs and the carry bit (C) is set.

Task 2.4 Two 8-bit values x and y are stored in the 16-bit register A, so that $AH = x$ and $AL = y$. Write a program that moves these values so that $A = x$ and $B = y$. Use AND masking.

Solution:

```
MOV B, A
SHR A, 8
AND B, 0x00FF
```

Prepare for the exam:

a) 16-bit registers A and B hold 8-bit values x and y respectively. Write a program that moves these values so that $AH = x$ and $AL = y$.

b) Do the same thing by using only 16-bit instructions and OR operation.

c) What does the following program in machine code do: `0x10 0x10 0x00 0x41`?

d) Run the below assembly program and observe the status register after the execution has finished. Explain why each status bit has changed.

```
MOV A, 0xFF00
ADD A, 0x0100
HLT
```

3 Memory addressing and variables

Review the different addressing modes listed in Appendix A.3. Remember:

- Square brackets `[ ]` represent memory access.
- Every CPU instruction can make at most one access to memory.

Task 3.1 What is the difference between programs:

```
MOV D, 100
MOV A, D
```

and

```
MOV D, 100
MOV A, [D]
```

Solution: The first program loads decimal value 100 into register A, while the second program loads whatever resides on the memory address 100 (`0x64`).

Task 3.2 Write the hexadecimal value `0xABCD` to memory address `0x0100`. Which addresses store which parts of this value? Now write the 8-bit number `0x33` to memory address `0x0102`. Write a program that adds the value on address `0x0102` to the value on address `0x0100`.

Solution:

```
MOV [0x0100], 0xABCD ; 16-bit value x
MOVB [0x0102], 0x33  ; 8-bit value y

; Implementation of x = x + y
MOV A, [0x0100]    ; Get the value of variable x.
MOV B, 0           ; We use B to convert y to a 16-bit value.
MOVB BL, [0x0102] ; Move y to the lower byte of B.
ADD A, B           ; x + y -> A as 16-bit addition.
MOV [0x0100], A    ; Store the result x + y to x.
HLT
```

Address `0x0100` stores the value `0xAB` and address `0x0101` stores the value `0xCD`.

Task 3.3 Define a 16-bit variable x with initial value $0xABCD$. Then increase the value of this variable by 3. Define the variable x in the following ways:

a) as a fixed memory address `0x0100`,
b) as a label below the program code,
c) as a label above the program code,
d) as a label at memory address `0x0100`.

Solutions:

a) As a fixed memory address 0x0100.

```
; The programmer decides that value x is stored at memory address 0x0100.
MOV [0x0100], 0xABCD ; Set x = 0xABCD
; The following three lines implement x = x + 3.
MOV A, [0x0100] ; Get the value of x.
ADD A, 3         ; Add 3 to the value.
MOV [0x0100], A ; Store the new value back to x.
HLT
```

b) As a label below the program code.

```
; The programmer does not care about the actual address of variable x,
; let the assembler determine it.
MOV A, [x]    ; Get the value from address x.
ADD A, 3      ; Add 3 to the value.
MOV [x], A    ; Store the new value back to address x.
HLT           ; Must (!) stop before data begins.
x: DW 0xABCD ; Let label x determine the memory address after HLT.
```

c) As a label above the program code.

```
; Let variable x reside before the program code.
JMP main
x: DW 0xABCD   ; define and initialize x = 0xABCD

; let label main determine the address of the MOV opcode below.
main:
    MOV A, [x] ; Get the value from address x.
    ADD A, 3   ; Add 3 to the value.
    MOV [x], A ; Store the new value back to address x.
    HLT
```

d) As a label at memory address 0x0100.

```
; Let variable x reside at a specific address in memory.
MOV A, [x]
ADD A, 3
MOV [x], A

; Instruction to the compiler where in memory to compile the code below.
ORG 0x0100
x: DW 0xABCD
```

Task 3.4 Define 16-bit variables x, y, and z. Write a program that computes $z = 3z - (x + y)/2$.

Solution:

```
JMP main
x: DW 3 ; define x = 1
y: DW 5 ; define y = 3
z: DW 7 ; define z = 7

main:
    ; z = z - (x + y)
    MOV A, [z] ; A <- z
    MUL 3      ; A <- 3*z
    MOV B, [x] ; B <- x
    ADD B, [y] ; B <- x + y
    SHR B, 1   ; B <- (x + y)/2
    SUB A, B   ; A <- 3*z - (x + y)/2
    MOV [z], A ; z <- 3*z - (x + y)/2
    HLT
```

Prepare for the exam:

a) Define variables x and y with the initial values of your choice. Write a program that switches the values of x and y.

b) Write a program that computes $z = 16 \cdot (2x - y)$. Use bit shifting for multiplication.

c) Write a program that computes $z = x^2 - y^2$. Use instruction `MUL` to square a value.

4 Conditional statements

Task 4.1 Define 16-bit variables x and y with some initial values. Subtract $x - y$. How does the subtraction affect the status flags if:

a) $x > y$,
b) $x = y$,
c) $x < y$.

Solution:

```
JMP main:
x: DW 7
y: DW 5

main:
    MOV A, [x]
    SUB A, [y]
    HLT
```

a) $Z = 0$, $C = 0$,
b) $Z = 1$, $C = 0$,
c) $Z = 0$, $C = 1$.

Task 4.2 What does the `CMP` instruction do and what is the advantage of using `CMP` instead of `SUB`?

Solution: The instruction `CMP` performs the same arithmetic operation as instruction `SUB`, but alters only the status register. When using `CMP`, we keep the values x and y.

Task 4.3 Write a program that compares values x and y. How are status flags `C` and `Z` affected for every possible comparison relation? Which conditional jump instructions are associated with what relation?

Solution:

```
JMP main:
x: DW 7
y: DW 5

main:
    MOV A, [x]
    CMP A, [y] ; Use CMP instead of SUB.
    HLT
```

Condition	Z	C	Jump instruction
$x = y$	1	-	JZ, JE
$x \neq y$	0	-	JNZ, JNE
$x > y$	0	0	JA, JNBE
$x >= y$	-	0	JNC, JAE, JNB
$x < y$	-	1	JC, JB, JNAE
$x <= y$	1	-	JNA, JBE
	-	1	

Task 4.4 Implement the following conditional statement in assembly language:

```
if (x < y)
    x = x + 1;
```

Solution: In assembly, it is sometimes the easier to negate the condition and skip the if body when the negated condition is true.

```
JMP main:
x: DW 7
y: DW 5

main:
    MOV A, [x]
    CMP A, [y]
    JAE continue     ; if (x >= y) skip the if body
    INC A
    MOV [x], A
continue:
    HLT
```

Task 4.5 Implement the following conditional statement in assembly language:

```
if (x < y) z = y;
else z = x;
```

Solution:

```
main:
    MOV A, [x]
    MOV B, [y]
    CMP A, B
    JAE else       ; if not (x < y) go to else
    MOV [z], B     ; z = y (if body)
    JMP continue
else:
    MOV [z], A     ; z = x (else body)
continue:
    HLT            ; exit the if statement
```

Task 4.6 Implement the following conditional statement in assembly language:

```
if (x >= 100 && x < 200)
    x = x / 2;
```

Solution: We may rewrite the above condition as:

```
if (x < 100 || x >= 200) {
}
else {
    x = x / 2;
}
```

And implement it as:

```
main:
    MOV A, [x]
    CMP A, 100 ; if (x < 100) don't do it
    JC out
    CMP A, 200 ; if (x >= 200) don't do it
    JNC out
    SHR A, 1   ; do it
    MOV [x], A ; x = x / 2
out:
    HLT
```

Prepare for the exam:

Implement the following conditions in assembly language:

a)
```
if (x + 1 > y)
    x = x - 1;
```

b)
```
if (x % 2 == 0)
    x = x / 2;
```

c)
```
if (x == 0 || x == 2 || x > y) {
    x = x - y;
}
else {
    if (x < y)
        x = x + y;
}
```

5 Loops

Task 5.1 Implement the following *while* loop in assembly language:

```
int i = 0;
while (i < 10) {
    i = i + 1;
}
```

Consider the possibilities for loop optimization.

Solution:

```
JMP main
i: DW 0         ; i = 0
main:
    MOV A, [i] ; Get the value of i.
while:
    CMP A, 10  ; Compare i with 10.
    JAE break  ; If above or equal, end the loop
    MOV A, [i] ; Get the value of i.
    INC A      ; i = i + 1
    MOV [i], A ; Store the new value of i.
    JMP while  ; Loop.
break:
    HLT
```

The loop can be optimized by keeping the value of `i` in a register:

```
JMP main
i: DW 0         ; i = 0
main:
    MOV A, [i] ; Get the value of i.
while:
    CMP A, 10  ; Compare i with 10.
    JAE break  ; If above or equal, end the loop
    INC A      ; i = i + 1
    JMP while  ; Loop.
break:
    MOV [i], A ; Store the last value of i.
    HLT
```

Potential problems could occur if variable `i` was shared between different threads. But because we use a single–thread system, we don't need to be careful about such issues.

Task 5.2 Implement an optimized *for* loop in assembly language that sums the numbers from 1 to 10.

Solution: Let us first write a solution in C:

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    sum = sum + i;
}
```

Because `i` is a local variable (visible only within the loop), we can optimize the code by storing `i` only within a register and write the value of `sum` to memory after the loop is finished.

```
JMP main
sum: DW 0
main:
    MOV A, 0      ; i = 0
    MOV B, [sum] ; Get the value of sum.
for:
    CMP A, 10     ; Compare i with 10.
    JA break      ; if i > 10, finish the loop.
    ADD B, A      ; sum = sum + i
    INC A         ; i++
    JMP for
break:
    MOV [sum], B ; Store the value of sum.
    HLT
```

Task 5.3 Write an assembly program that computes the factorial of n. Suppose that multiplication is not possible on the system.

Solution: In C, we could write this program as follows:

```
int f = 1;
for (int i = 2; i <= n; i++) { // factorial loop
    f = f * i;
}
```

But because multiplication is not allowed, we implement it by addition:

```
int f = 1;
for (int i = 2; i <= n; i++) { // factorial loop
    int fi = 0;
    for (int j = i; j > 0; j--) { // multiplication loop
        fi = fi + f;
    }
    f = fi;
}
```

We then rewrite the solution in assembly:

```
JMP main
n: DW 5                  ; The input parameter.
f: DW 1                  ; The result: f = n!
main:
    MOV B, [f]           ; We often need this value.
    MOV A, 2             ; int i = 2
factloop:
    CMP A, [n]           ; Has i surpassed n?
    JA endfactloop       ; yes, finish.
    MOV C, 0             ; int fi = 0
    MOV D, 0             ; int j = 0
multloop:
    CMP D, A             ; Compare j and i.
    JAE endmultloop      ; If j reached i, multiplication is done.
    ADD C, B             ; fi = fi + f
    INC D                ; j++
    JMP multloop
endmultloop:
    MOV B, C             ; f = fi
    INC A                ; i++
    JMP factloop
endfactloop:
    MOV [f], B           ; Store the value of f.
    HLT
```

Prepare for the exam:

a) Use the `MUL` instruction to simplify the assembly program that computes the factorial.

b) Write a program that determines if a given integer is divisible by 3. The idea is to subract the constant 3 from the given number until that number is reduced below 3. If the result is 0, the number is divisible by 3.

c) Write a program that counts how many bits in register A are set to 1. This can be done by shifting left the register 16 times while checking the `C` flag for the value of the discarded bit.

d) Write a program that determines how much memory is currently free. The idea is to read every byte of memory between the addresses `0x0000` and `0x0FFF`, and count the number of zero bytes after the last non-zero byte.

6 Stack and Functions

6.1 Stack

- *Stack* is a linear data structure that allows direct access only to the last input element.
- CPU implements a stack in memory to allow function calls. It is also very handy for programmers to temporary store data.
- CPU stack instructions are: `PUSH`, `POP`, `PUSHB`, `POPB`.
- Make sure you correctly match `PUSH` and `POP` instructions.

Task 6.1 Initialize the stack. Switch values of registers `A` and `B` without using any other register.

Solution:

```
MOV A, 1        ; An initial value of A.
MOV B, 2        ; An initial value of B.
MOV SP, 0x0FFF  ; Initialize the stack pointer.

; One way to switch values between A and B
PUSH A          ; Push the value of A onto stack.
MOV A, B        ; Move the value of B to A.
POP B           ; Pop the previous value of A into B.

; Another way to switch values between A and B
PUSH A          ; Push the value of A onto stack.
PUSH B          ; Push the value of B onto stack.
POP A           ; Pop the previous value of B into A.
POP B           ; Pop the previous value of A into B.
HLT
```

Task 6.2 Push 100 16-bit numbers to stack and examine the part of the memory where the numbers are being stored. How many can you push to the stack before you run into troubles? What can happen?

Solution:

```
    MOV SP, 0x0FFF ; Initialize the stack pointer.
    MOV A, 1       ; Start counting.
loop:
    CMP A, 100     ; When counter exceeds 100
    JA exit        ; exit.
    PUSH A         ; Push the current value on stack.
    INC A          ; Increase the counter.
    JMP loop       ; Next iteration.
exit:
    HLT
```

Our program takes 24 bytes, so we are left with 4072 bytes of free memory. This means we can store up to 2036 16-bit numbers before we run out of memory. If we try to store more than that, the stack starts overwriting our program. On a modern computer system, the operating system would prevent this from happening and throw a *stack overflow* exception.

6.2 Functions

- To call a function use instruction `CALL` (never `JMP`).
- A function must always return by executing the instruction `RET`.
- Parameters and return values can be passed either through registers or stack. Avoid passing parameters through fixed memory addresses or global variables.

Task 6.3 Write a function that pops two 16-bit numbers from the stack and pushes their sum back to the stack. Demonstrate the use of this function.

Solution:

```
JMP main

add:
    POP D          ; Pop the return address.
    POP B          ; Pop the second summand.
    POP A          ; Pop the first summand.
    ADD A, B       ; Add the summands.
    PUSH A         ; Push the sum.
    PUSH D         ; Push back the returned address.
    RET            ; Return from function.

main:
    MOV SP, 0xFFF ; Initialize the stack pointer.
    PUSH 5         ; Push the first summand to stack.
    PUSH 3         ; Push the second summand to stack.
    CALL add       ; Call the function add.
    POP A          ; Retreive the returned sum.
    HLT
```

Task 6.4 Write a function `max(a, b)`, that returns the greater of the two given integers. Let the function receive the parameters a and b through registers A and B respectively.

Solution:

```
JMP main
z: DW 0             ; int z = 0;

max:
    CMP A, B        ; If A < B then
    JB returnb      ;   return B
```

```
    RET             ; else the result is already in register A.
returnb:
    MOV A, B        ; Move the result to register A,
    RET             ; and return it.

main:
    MOV SP, 0x0FFF ; Initialize the stack pointer.
    MOV A, 2        ; Set the first argument.
    MOV B, 5        ; Set the second argument.
    CALL max        ; Call z = max(2, 5).
    MOV [z], A      ; Store the returned value.
    HLT
```

Prepare for the exam:

a) Write a function `add(x1, x2, x3, ...)` that can receive any number of summands $x_1, x_2, \ldots, x_n$ and returns their sum in register A. Let the number of summands n be received in register A and all the summands on stack.

b) Write a function `sign(x)` that returns the sign of the given 16-bit signed integer x, i.e. 1 if the number is positive, -1 if it is negative, and 0 if it is 0. Suppose that x is in the two's complement format.

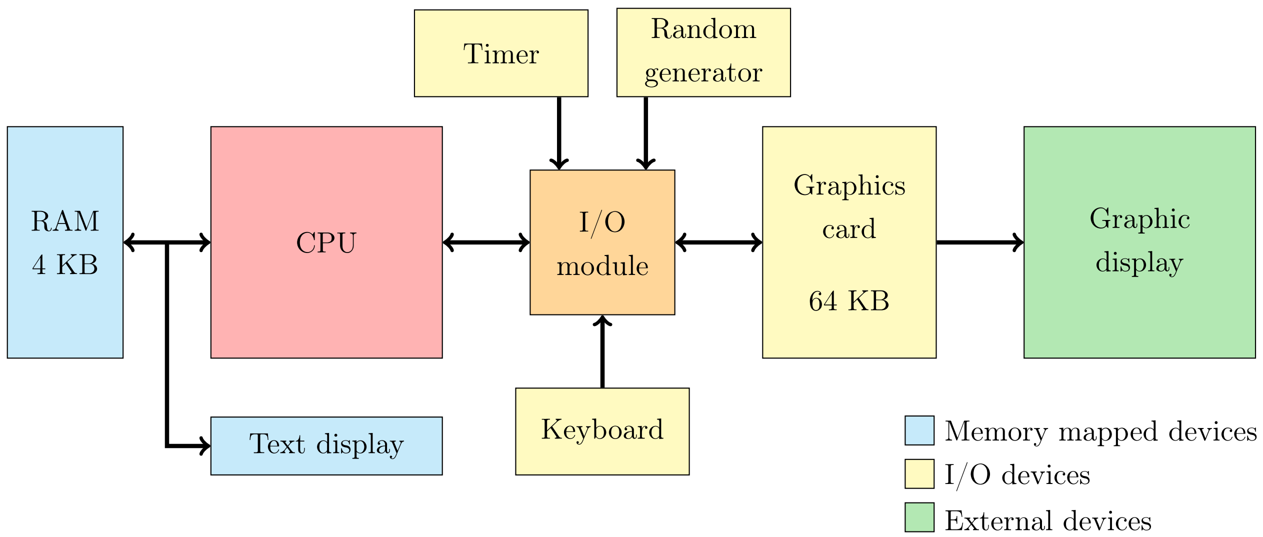
c) Write a function `abs(x)` that returns the absolute value of the given 16-bit signed integer x. Suppose that x is in the two's complement format.

d) Write a function `fib(n)` that returns the n-th Fibonacci number.

e) Write a function `isprime(x)` that checks if the given number x is prime. A number is a prime number if it is not divisible by any number between 2 and $n-1$. You may check divisibility by first dividing (`DIV`) the number and then multiplying (`MUL`) it back. Write another function `primes(n)` that returns all prime numbers between 2 and n. You may return all the numbers through stack and the number of returned values through a register.

A Architecture of the simulated system

A.1 System architecture



A.2 CPU registers

16-bit registers:

Register	Description	Index
A	General purpose register	0
B	General purpose register	1
C	General purpose register	2
D	General purpose register	3
SP	Stack Pointer	4
IP	Instruction Pointer	5
SR	Status Register	6

8-bit registers:

Register	Description	Index
AH	Higher part of register A	7
AL	Lower part of register A	8
BH	Higher part of register B	9
BL	Lower part of register B	10
CH	Higher part of register C	11
CL	Lower part of register C	12
DH	Higher part of register D	13
DL	Lower part of register D	14

A.3 Addressing modes

Addressing mode	Abbreviation	Example
Immediate	IMD	`ADD A, 100`
Register	REG	`ADD A, B`
Direct	DIR	`ADD A, [100]`
Indirect	IND	`ADD A, [B]`

A.4 Instruction set

CPU instruction format:

opcode	operand 1 (optional)	operand 2 (optional)

CPU instruction set:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	HLT	MOV REG,REG	MOV REG,IND	MOV REG,DIR	MOV IND,REG	MOV DIR,REG	MOV REG,IMD	MOV IND,IMD	MOV DIR,IMD	MOVB REG,REG	MOVB REG,IND	MOVB REG,DIR	MOVB IND,REG	MOVB DIR,REG	MOVB REG,IMD	MOVB IND,IMD
1	MOVB DIR,IMD	ADD REG,REG	ADD REG,IND	ADD REG,DIR	ADD REG,IMD	ADDB REG,REG	ADDB REG,IND	ADDB REG,DIR	ADDB REG,IMD	SUB REG,REG	SUB REG,IND	SUB REG,DIR	SUB REG,IMD	SUBB REG,REG	SUBB REG,IND	SUBB REG,DIR
2	SUBB REG,IMD	INC REG	INCB REG	DEC REG	DECB REG	CMP REG,REG	CMP REG,IND	CMP REG,DIR	CMP REG,IMD	CMPB REG,REG	CMPB REG,IND	CMPB REG,DIR	CMPB REG,IMD	JMP IND	JMP DIR	JC IND
3	JC DIR	JNC IND	JNC DIR	JZ IND	JZ DIR	JNZ IND	JNZ DIR	JA IND	JA DIR	JNA IND	JNA DIR	PUSH REG	PUSH IMD			PUSHB REG
4	PUSHB IMD			POP REG	POPB REG	CALL IND	CALL DIR	RET	MUL REG	MUL IND	MUL DIR	MUL IMD	MULB REG	MULB IND	MULB DIR	MULB IMD
5	DIV REG	DIV IND	DIV DIR	DIV IMD	DIVB REG	DIVB IND	DIVB DIR	DIVB IMD	AND REG,REG	AND REG,IND	AND REG,DIR	AND REG,IMD	ANDB REG,REG	ANDB REG,IND	ANDB REG,DIR	ANDB REG,IMD
6	OR REG,REG	OR REG,IND	OR REG,DIR	OR REG,IMD	ORB REG,REG	ORB REG,IND	ORB REG,DIR	ORB REG,IMD	XOR REG,REG	XOR REG,IND	XOR REG,DIR	XOR REG,IMD	XORB REG,REG	XORB REG,IND	XORB REG,DIR	XORB REG,IMD
7	NOT REG	NOTB REG	SHL REG,REG	SHL REG,IND	SHL REG,DIR	SHL REG,IMD	SHLB REG,REG	SHLB REG,IND	SHLB REG,DIR	SHLB REG,IMD	SHR REG,REG	SHR REG,IND	SHR REG,DIR	SHR REG,IMD	SHRB REG,REG	SHRB REG,IND
8	SHRB REG,DIR	SHRB REG,IMD	CLI	STI	IRET			IN REG	IN IND	IN DIR	IN IMD	OUT REG	OUT IND	OUT DIR	OUT IM	

A.5 Memory map

Address range	Component
`0x0000` – `0x0FFF`	RAM (4 KB)
`0x1000` – `0x101F`	Text display (2 lines × 16 ASCII characters)

A.6 Input / Output module

I/O Registers:

Register	Description	Index
IRQMASK	Enable/Disable specific interrupt requests	0
IRQSTATUS	Currently requested interrupts	1
IRQEIO	Clear a specific interrupt request	2
TMRPRELOAD	The initial timer value	3
TMRCOUNTER	The current timer value	4
KBDSTATUS	Keyboard status (a keypress has been detected)	5
KBDDATA	The data received from the keyboard	6
VIDMODE	Graphics card mode	7
VIDADDR	Address in the VRAM data	8
VIDDATA	Data at the VRAM address VIDADDR	9
RNDGEN	A randomly generated number	10

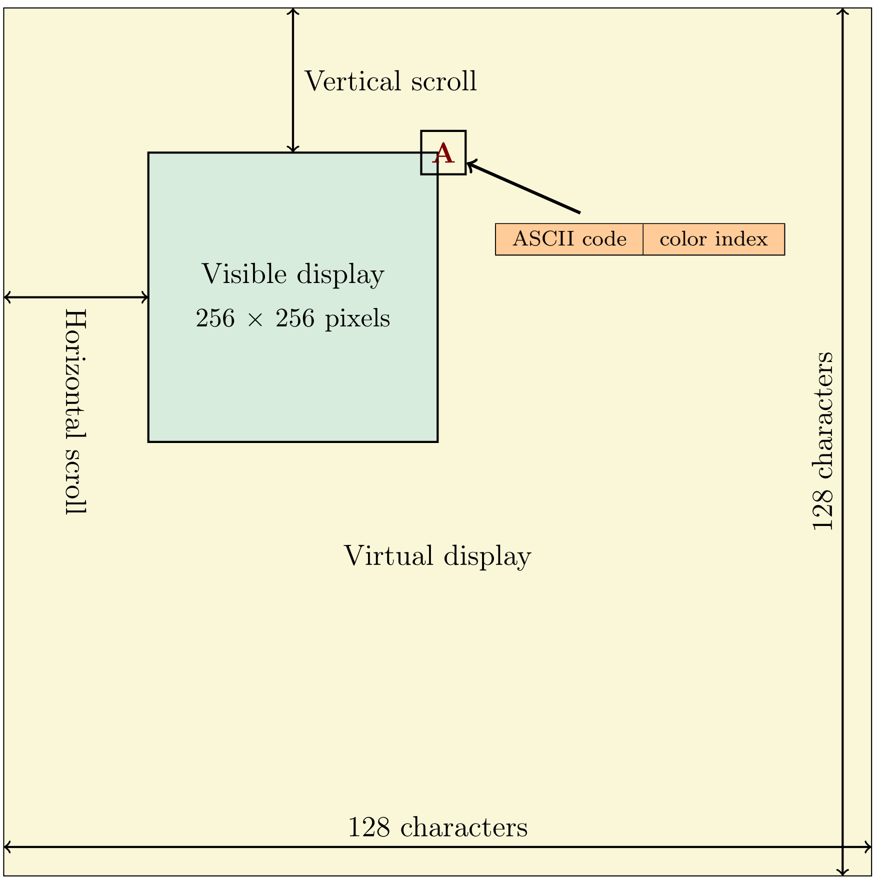
A.7 Graphics card

VIDMODE	Video mode	Description
0	DISABLED	Graphics card is disabled (default).
1	TEXT	Display text and custom-defined tiles.
2	BITMAP	Display individual pixels.
3	CLEAR	Clears the display. It works in both graphic modes.
4	RESET	Resets and disables the graphics card.

A.7.1 Text mode (VIDMODE = 1)

Text display properties:

- The size of each character is 16×16 pixels.
- Every displayed character has two properties: ASCII code (8-bit) and color index (8-bit).
- Default shape definitions of all 256 characters are stored in VRAM at addresses `0x8000` – `0x9FFF` and can be overwritten. By resetting the graphics card, the default values are restored.
- The standard RRRGGGBB palette is stored in VRAM at addresses `0xA000` – `0xA2FF` (3 bytes/color) and can be overwritten. By resetting the graphics card, the default values are restored.
- The size of the virtual display is 128×128 characters, but only a display window of 256×256 pixels is visible on the screen. The display window can be moved to achieve smooth scrolling effect.
- Eight characters can be placed on the visual display at any pixel position (256×256), independently of the virtual display content and scrolling state (sprite graphics).



VRAM Usage in text mode:

VRAM location	Size	Usage
`0x0000` – `0x7FFF`	32 KB	Virtual display area (128 × 128 characters, 2 bytes/character).
`0x8000` – `0x9FFF`	8 KB	Character set definition (256 characters, 32 bytes/character).
`0xA000` – `0xA2FF`	0.75 KB	Color palette (256 colors, 3 bytes/color).
`0xA300` – `0xA301`	2 B	Background color (color index 0 – 255, address `0xA300` is unused).
`0xA302` – `0xA303`	2 B	Horizontal scroll (display window x-offset in pixels).
`0xA304` – `0xA305`	2 B	Vertical scroll (display window y-offset in pixels).
`0xA306` – `0xA309`	4 B	Sprite 1 (character, color, x, y).
`0xA30A` – `0xA30D`	4 B	Sprite 2 (character, color, x, y).
`0xA30E` – `0xA311`	4 B	Sprite 3 (character, color, x, y).
`0xA312` – `0xA315`	4 B	Sprite 4 (character, color, x, y).
`0xA316` – `0xA319`	4 B	Sprite 5 (character, color, x, y).
`0xA31A` – `0xA31D`	4 B	Sprite 6 (character, color, x, y).
`0xA31E` – `0xA321`	4 B	Sprite 7 (character, color, x, y).
`0xA322` – `0xA325`	4 B	Sprite 8 (character, color, x, y).
`0xA326` – `0xFFFF`		Free memory (23754 bytes).

Character shape data (32 bytes):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`0x0000`
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`0x0000`
0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	`0x1FF8`
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	`0x3FFC`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0X3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	`0x3FFC`
0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	`0x3FFC`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	`0x3C3C`
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`0x0000`
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	`0x0000`

Virtual display area:

character 0	ASCII code	color index	32 KB
character 1	ASCII code	color index	
⋮	⋮		
character n	ASCII code	color index	

Character set definition:

ASCII code 0	shape data 0	8 KB
ASCII code 1	shape data 1	
⋮	⋮	
ASCII code 255	shape data 255	

Color palette definition:

color 0	red (8-bit)	green (8-bit)	blue (8-bit)	765 bytes
color 1	red (8-bit)	green (8-bit)	blue (8-bit)	
⋮		⋮		
color 255	red (8-bit)	green (8-bit)	blue (8-bit)	

Sprite data:

sprite 1	ASCII code	color	x	y	32 bytes
sprite 2	ASCII code	color	x	y	
⋮		⋮			
sprite 8	ASCII code	color	x	y	

A.7.2 Bitmap mode (VIDMODE = 2)

The entire 64 KB VRAM is used to display a bitmap image (256 × 256 pixels). The standard 8-bit color palette RRRGGGBB is used, which cannot be redefined.