



Microprocessor Simulator V5.0 Help

© C Neil Bauers 2003 – <http://www.softwareforeducation.com/>

General	Tutorials	Reference
Introduction Architecture Installation Un-Installation To Register Registration Form FAQ and Bugs PC Support Handbook Use Alt+Tab to switch between the help and simulator windows.	Getting Started All Learning Tasks 01 First Program -- Nasty Example 02 Traffic Lights 03 Data Moves 04 Counting 05 Keyboard Input 06 Procedures 07 Text I/O 08 Data Tables 09 Parameters 10 SW Interrupts 11 HW Interrupts	Shortcut Keys ASCII Codes Glossary Hexadecimal and Binary Instruction Set Summary Instruction Set Detailed The List File Negative Numbers Pop-up Help Logic and Truth The Editor Peripheral Devices

This simulator is for learners in the 16+ age range although many younger enthusiasts have used it too. It introduces low level programming and microcomputer architecture. Tutorial materials are included covering the subject in some depth.

The tutorials align closely with the British GCE A2 Computing specifications and also the British BTEC National for IT Practitioners (Computer Systems).

The simulator has enough depth and flexibility to be used with university undergraduate students studying low level programming for the first time.



Introduction

Contents

Who Should Use the Simulator

The simulator is intended for any student studying low level programming, control or machine architecture for the first time.

The simulator can be used by students aged 14 to 16 to solve less complex problems such as controlling the traffic lights and snake.

More advanced students typically 16 or older can solve quite complex low level programming problems involving conditional jumps, procedures, software and hardware interrupts and Boolean logic. Although programs will be small, there is good scope for modular design and separation of code and data tables.

The simulator is suitable for courses such as

- BTEC National Diploma for IT Practitioners (Computer Systems and Control Technology)
- AS and A2 Computing (Low Level Programming)
- Electronics Courses.

- Courses involving microcontrollers.
- Courses involving control systems.

Description of the Simulator

In the shareware version the following instructions are not included. CALL, RET, INT and IRET. The hardware timer interrupt does not function because IRET can not be used either. The registered version includes these features. You can [register](#) the software here.

This simulator emulates an eight bit CPU that is similar to the low eight bits of the 80x86 family of chips. 256 bytes of RAM are simulated. It is surprising how much can be done with only 256 bytes of RAM.

Features

- 8 bit CPU
- 16 Input Output ports. Not all are used.
- Simulated peripherals on ports 0 to 5.
- An assembler.
- On-line help.
- Single step through programs.
- Continuously run programs.
- Interrupt 02 triggered by a hardware timer (simulated).
- CPU Clock Speed can be altered.

Peripherals

- Keyboard Input
- Traffic Lights
- Seven Segment Display
- Heater and Thermostat
- Snake and Maze
- Stepper Motor
- Memory Mapped VDU

Example Programs

- 99keyb.asm
- 99tlight.asm
- 99sevseg.asm
- 99hon.asm and 99hoff.asm
- 99snake.asm
- 99step.asm
- 99keyb.asm

Documentation

On-line hypertext help is stored in a Website. It is possible to copy from the help pages and paste into a word processor or text editor programs. Registered users have permission to modify help files for use by students and to print and or make multiple photocopies.

Disclaimer

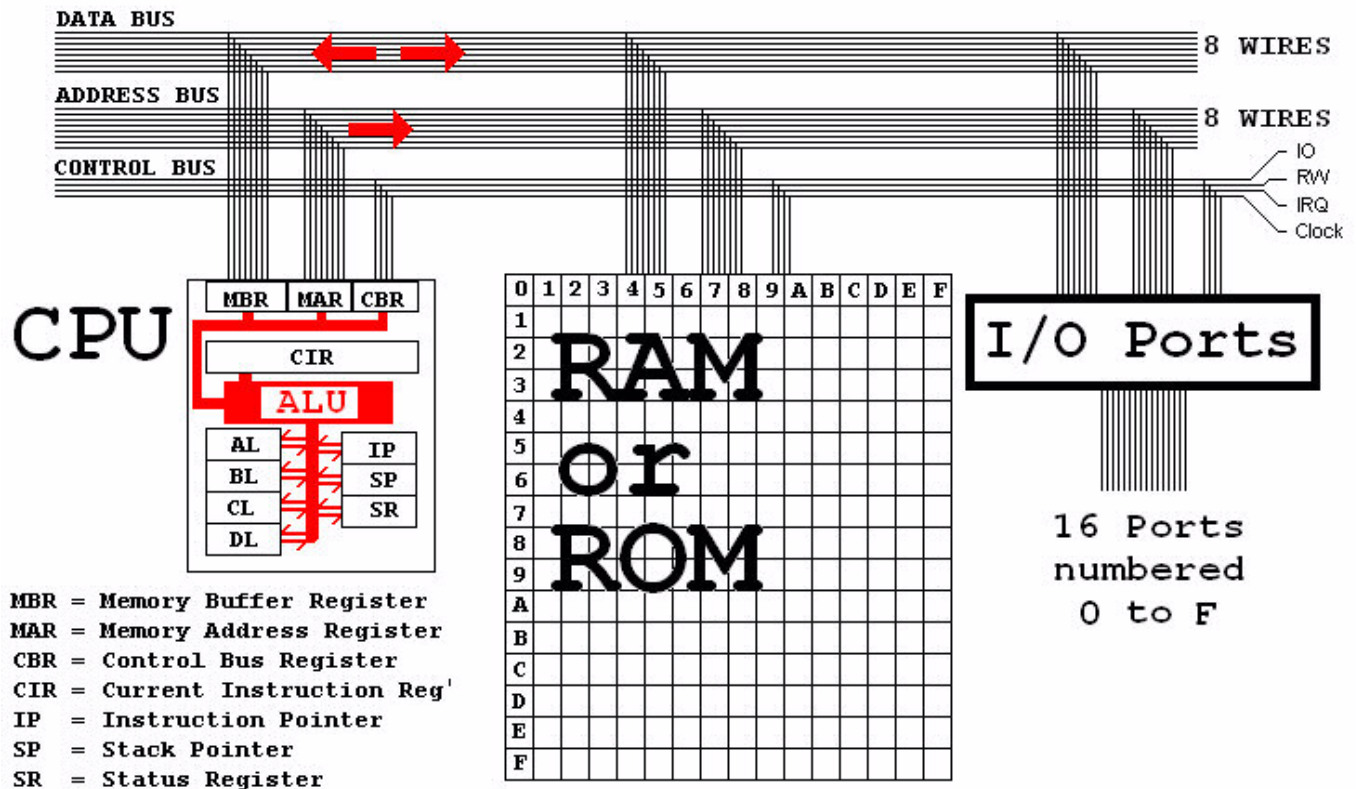
This simulation software is not guaranteed in any way. It may differ from reality. It might not even work at all. Try it out and if you like it, please register.



System Architecture

Contents

Simplified Simulator Architecture



- central processing unit (CPU)
- 256 bytes of random access memory (RAM)
- 16 input output (IO) ports. Only six are used.
- A hardware timer that triggers interrupt 02 at regular time intervals that you can pre-set using the configuration tab.
- A keyboard that triggers interrupt 03.
- Peripherals connected to the Ports.

The simulator is programmable in that you can run many different programs. In real life, the RAM would be replaced by read only memory (ROM) and the system would only ever run one program hard wired into the ROM. There are hundreds of examples of systems like this controlling traffic lights, CD players, simple games consoles, many children's games, TV remote controls, microwave oven timers, clock radios, car engine management systems, central heating controllers, environmental control systems and the list goes on.

The Central Processing Unit

The central processing unit is the "brain" of the computer. All calculations, decisions and data moves are made here. The CPU has storage locations called **registers**. It has an arithmetic and logic unit (ALU) where the processing is done. Data is taken from the registers, processed and results go back into the registers. Move (MOV) commands are used to transfer data between RAM locations and the registers. There are many instructions, each with a specific purpose. This collection is called the instruction set.

General Purpose Registers

The CPU has four general-purpose registers called AL, BL, CL and DL. These are eight bits or one byte wide. Registers can hold unsigned numbers in the range 0 to +255 and signed numbers in the range -128 to +127. These are used as temporary storage locations. Registers are used in preference to RAM locations because it takes a relatively long time to transfer data between RAM and the CPU. Faster computers generally have more CPU registers or memory on the CPU chip.

The registers are named AL, BL, CL and DL because the 16-bit version of this CPU has more registers called AH, BH, CH and DH. The 'L' means Low and the 'H' means High. These are the low and high ends of the 16-bit register.

Special Purpose Registers

The special purpose registers in the CPU are called IP, SR and SP.

IP is the Instruction pointer

This register contains the address of the instruction being executed. When execution is complete, IP is increased to point to the next instruction. Jump instructions alter the value of IP so the program flow jumps to a new position. CALL and INT also change the value stored in IP. In the RAM displays, the instruction pointer is highlighted red with yellow text.

SR is the Status Register

This register contains flags that report the CPU status.

The 'Z' zero flag is set to one if a calculation gave a zero result.

The 'S' sign flag is set to one if a calculation gave a negative result.

The 'O' overflow flag is set if a result was too big to fit in a register.

The 'I' interrupt is set if interrupts are enabled. See CLI and STI.

SP is the Stack Pointer

The stack is an area of memory organised using the LIFO last in first out rule. The stack pointer points to the next free stack location. The simulator stack starts at address BF just below the RAM used for the video display. The stack grows towards address zero. Data is pushed onto the stack to save it for later use. Data is popped off the stack when needed. The stack pointer SP keeps track of where to push or pop data items. In the RAM displays, the stack pointer is highlighted blue with yellow text.

Random Access Memory

The simulator has 256 bytes of ram. The addresses are from 0 to 255 in decimal numbers or from [00] to [FF] in hexadecimal. RAM addresses are usually given in square brackets such as [7C] where 7C is a hexadecimal number. Read [7C] as "the data stored at location 7C".

Busses

Busses are collections of wires used to carry signals around the computer. They are commonly printed as parallel tracks on circuit boards. Slots are sockets that enable cards to be connected to the system bus. An 8-bit computer typically has registers 8 bits wide and 8 wires in a bus. A 16-bit computer has 16 bit registers and 16 address and data wires and so on. The original IBM PC had 8 data wires and 20 address wires enabling one megabyte of RAM to be accessed. 32 bit registers and busses are now usual (1997-2003).

Data Bus	The Data Bus is used to carry data between the CPU, RAM and IO ports. The simulator has an 8-bit data bus.
Address Bus	The Address Bus is used to specify what RAM address or IO port should be used. The simulator has an 8-bit address bus.

Control Bus	<p>The Control Bus This has a wire to determine whether to access RAM or IO ports. It also has a wire to determine whether data is being read or written. The CPU reads data when it flows into the CPU. It writes data when it flows out of the CPU to RAM or the IO ports.</p> <p>The System Clock wire carries regular pulses so that all the electronic components can function at the correct times. Clock speeds between 100 and 200 million cycles per second are typical (1997). This is referred to as the clock speed in MHz or megahertz. The simulator runs in slow motion at about one instruction per second. This is adjustable over a small range.</p>
Hardware Interrupts	<p>Hardware Interrupts require at least one wire. These enable the CPU to respond to events triggered by hardware such as printers running out of paper. The CPU processes some machine code in response to the interrupt. When finished, it continues with its original task. The IBM PC has 16 interrupts controlled by 4 wires.</p>



To Register

Contents

Shareware

The Microcontroller Simulator is shareware. In the unregistered shareware version the following features are not included. CALL, RET, INT, IRET and the simulated hardware timer interrupt 02.

Distribution

The complete unaltered shareware package may be distributed freely without restriction. Re-packaging to suit different distribution media is permitted. Please give a copy to any interested friend, colleague or student.

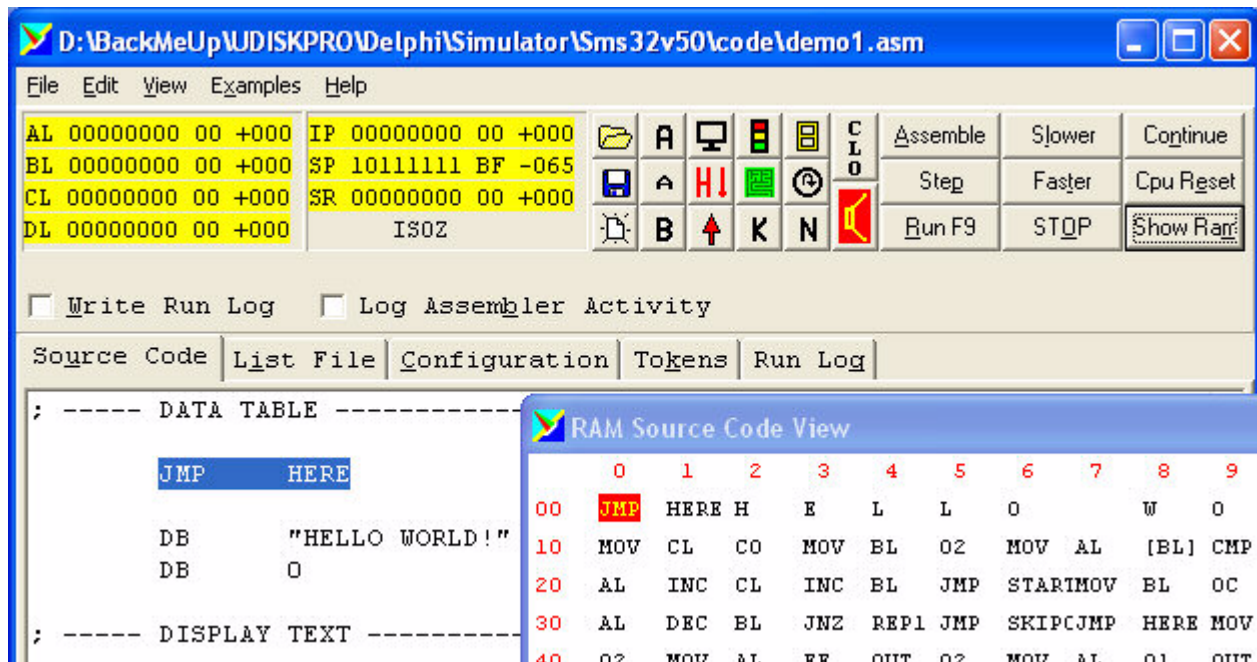
Student Registration

Individual students may use this software free. There is no need to register. Students may also use registered copies free of charge. These copies should, whenever possible, be obtained from your school, college or university.



Using the Simulator - Getting Started

Contents



On Line Help

Press the **F1** key to get on line help.

Writing a Program

To write and run a program using the simulator, select the source code editor tab by pressing **Alt+U**




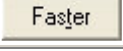


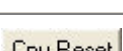
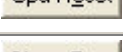
Type in your program. It is best to get small parts of the program working rather than typing it all in at once.

Here is a simple example. Also look at the tutorial example programs. You can type this into the simulator or copy and paste it. The assembly code has been annotated with comments that explain the code. These comments are ignored by the assembler program. Comments begin with a semicolon and continue to the end of the line.



```
; ===== COUNT =====
MOV     AL,0      ; Move 0 into the AL register
REP:    ; This label is used with jump commands
ADD     AL,2      ; Add two to AL
JMP     REP       ; Jump back to the rep label

END      ; Program ends here
; =====
```

Running a Program

	To run a program, you can step through it one line at a time by pressing Alt+P or by clicking this button repeatedly.
	You can run a program continuously by pressing F9 or Alt+R or by pressing this button
 	To speed up or slow down a running program use these buttons or type Alt+L or Alt+T
	To stop a running program press Alt+O or click or press Escape or press this button.
	To restart a paused program, continuing from where it left off, press Alt+N or click this button.
	To restart a program from the beginning, reset the CPU by pressing Alt+E or click this button.
	To re-open the RAM display window, press Alt+M or click this button.

Assembly Code

	The code you type is called assembly code. This human-readable code is translated into machine code by the Assembler . The machine code (binary) is understood by the CPU. To assemble a program, press Alt+A or click this button.
<input type="checkbox"/> Log Assembler Activity	You can see an animation of the assembler process by checking this box.
	When you run or setp a program, if necessary, the code is assembled.

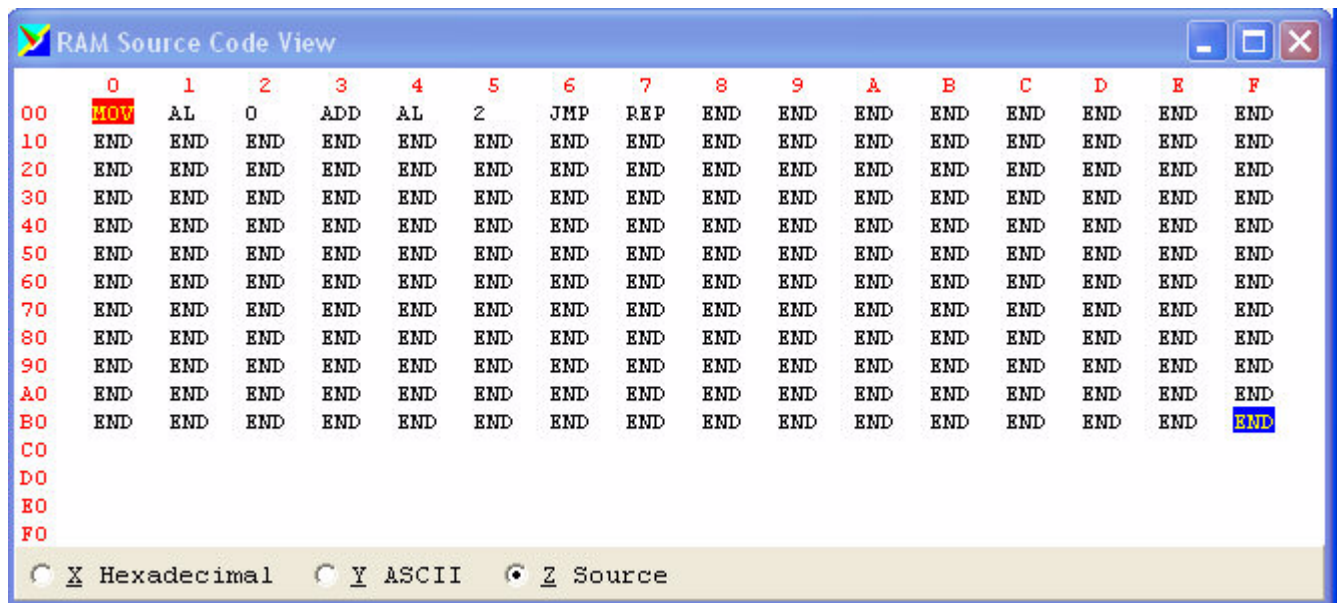
Assembler Phases

There is short delay while the assemblr goes through all the stages of assembling the program. The steps are

1. **Save** the source code.
2. Convert the source code into **tokens**(this simulator uses human readable tokens for educational value rather than efficiency).
3. **Parse** the source code and (if necessary) generate error messages. If there are no errors, generate the machine codes. This process could be coded more efficiently. If the tokens representing machine op codes like MOV and JMP were numerical, the assembler could look up the machine code equivalents in an array instead of ploughing through many if-then-else statements. Once again, this has been done to demonstrate the process of assembling code for educational reasons.
4. **Calculate jumps**, the distances of the jump/branch instructions.

Viewing Machine Code

The machine code stored in RAM can be viewed in three modes by selecting the appropriate radio button.



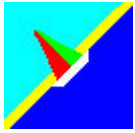
Hexadecimal - This display corresponds exactly to the binary executed by the CPU.

ASCII - This display is convenient if your program is processing text. The text is readable but the machine codes are not.

Source Code - This display shows how the assembly code commands are placed in memory.

Tutorial Examples

The tutorial examples provide a step by step introduction to the commands and techniques of low level programming. Each program has one or more learning tasks associated with it. Some of the tasks are simple. Some are real brain teasers.



Learning Tasks

Contents

The Tasks

Here are all the learning tasks grouped together with pointers to the example programs and explanatory notes.

Simple Arithmetic

Example - **01first.asm** - Arithmetic

1. Write a program that subtracts using SUB
2. Write a program that multiplies using MUL
3. Write a program that divides using DIV
4. Write a program that divides by zero. Make a note to avoid doing this in real life.

Using Hexadecimal

Example - **02tlight.asm** - Traffic Lights

5. Use the help page on Hexadecimal and Binary numbers. Work out what hexadecimal numbers will activate the correct traffic lights. Modify the program to step the lights through a realistic sequence.

ASCII Codes

Example - **03move.asm**

6. Look up the ASCII codes of H, E, L, L and O and copy these values to memory locations C0, C1, C2, C3 and C4. This is a simple and somewhat crude way to display text on a memory mapped display.

Counting and Jump Commands

Example - **04incjmp.asm**

7. Rewrite the example program to count backwards using DEC BL.
 8. Rewrite the example program to count in threes using ADD BL,3.
 9. Rewrite the program to count 1 2 4 8 16 using MUL BL,2
 10. Here is a more difficult task. Count 0 1 1 2 3 5 8 13 21 34 55 98 overflow. Here each number is the sum of the previous two. You will need to use two registers and two RAM locations for temporary storage of numbers. If you have never programmed before, this is a real brain teaser. Remember that the result will overflow when it goes above 127.
This number sequence was first described by Leonardo Fibonacci of Pisa (1170_1230)
-

Character Input Output

Example - **05keybin.asm**

11. Easy! Input characters and display each character at the top left position of the VDU by copying them all to address [C0].
12. Harder Use BL to point to address [C0] and increment BL after each key press in order to see the text as you type it.
13. Harder! Store all the text you type in RAM when you type it in. When you press Enter, display the stored text on the VDU display.
14. Difficult Type in text and store it. When Enter is pressed, display it on the VDU screen in reverse order. Using the stack makes this task easier

Procedures

Example - **06proc.asm**

15. Re-do the traffic lights program and use this procedure to set up realistic time delays.
02tlight.asm
16. Re-do the text input and display program with procedures. Use one procedure to input the text and one to display it.

Text IO and Procedures

Example - **07textio.asm**

17. Write a program using three procedures. The first should read text from the keyboard and store it in RAM. The second should convert any upper case characters in the stored text to lower case. The third should display the text on the VDU screen.

Data Tables

Example - **08table.asm**

18. Improve the traffic lights data table so there is an overlap with both sets of lights on red.
 19. Use a data table to navigate the snake through the maze. This is on port 04. Send FF to the snake to reset it. Up, down left and right are controlled by the left four bits. The right four bits control the distance moved.
 20. Write a program to spin the stepper motor. Activate bits 1, 2, 4 and 8 in sequence to energise the electromagnets in turn. The motor can be half stepped by turning on pairs of magnets followed by a single magnet followed by a pair and so on.
 21. Use a data table to make the motor perform a complex sequence of forward and reverse moves. This is the type of control needed in robotic systems, printers and plotters. For this exercise, it does not matter exactly what the motor does.
-

Parameters

Example - **09param.asm**

22. Write a procedure that doubles a number. Pass the single parameter into the procedure using a register. Use the same register to return the result.
23. Write a procedure to invert all the bits in a byte. All the zeros should become ones. All the ones should become zeros. Pass the value to be processed into the procedure using a RAM location. Return the result in the same RAM location.
24. Write a procedure that works out Factorial N. This example shows one method for working out factorial N. Factorial 5 is $5 * 4 * 3 * 2 * 1 = 120$. Your procedure should work properly for factorial 1, 2, 3, 4 or 5. Factorial 6 would cause an overflow. Use the stack to pass parameters and return the result. Calculate the result. Using a look up table is cheating!
25. Write a procedure that works out Factorial N. Use the stack for parameter passing. Write a recursive procedure. Use this definition of Factorial.
Factorial (0) is defined as 1.
Factorial (N) is defined as $N * \text{Factorial} (N - 1)$.
To work out Factorial (N), the procedure first tests to see if N is zero and if not then re-uses itself to work out $N * \text{Factorial} (N - 1)$. This problem is hard to understand in any programming language. In assembly code it is harder still.

Software Interrupts

Example - **10swint.asm**

26. The simulated keyboard generates INT 03 every time a key is pressed. Write an interrupt 03 handler to process the key presses. Use IN 07 to fetch the pressed key into the AL register. The original IBM PC allocated 16 bytes for key press storage. The 16 locations are used in a circular buffer fashion. Try to implement this.
27. Build on task 26 by putting characters onto the next free screen location. See if you can get correct behaviour in response to the Enter key being pressed (fairly easy) and if the Back Space key being pressed (harder).

Hardware Interrupts

Example - **11hwint.asm**

28. Write a program that controls the heater and thermostat whilst at the same time counting from 0 to 9 repeatedly, displaying the result on one of the seven segment displays. If you want a harder challenge, count from 0 to 99 repeatedly using both displays. Use the simulated hardware interrupt to control the heater and thermostat.



Example - 01first.asm - Arithmetic

Contents

Most of these examples include a learning task. Study the example and if you can complete the task/s, it is likely that your understanding is good.

Example - 01first.asm

```
; ===== WORK OUT 2 PLUS 2 =====
CLO                ; Close unwanted windows.
MOV AL,2           ; Copy a 2 into the AL register.
MOV BL,2           ; Copy a 2 into the BL register.
ADD AL,BL          ; Add AL to BL. Answer goes into AL.
END               ; Program ends
; ===== Program Ends =====

YOUR TASK
=====
Use SUB, DIV and MUL to subtract, divide and multiply.
What happens if you divide by zero?
Make use of CL and DL as well as AL and BL.
```

Type this code into the simulator editor **OR** copy and paste the code **OR** load the example from disk.

Step through the program by pressing **Alt+P** repeatedly.

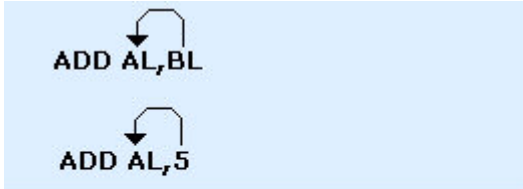
While you are stepping, watch the CPU registers. You should see a '2' appear in the AL register followed by a '2' in the BL register. AL should be added to BL and a '4' should appear in AL. The altered registers are highlighted yellow.

Watch the register labelled IP (Instruction Pointer). This register keeps track of where the processor has got to in the program. If you look at the RAM display, one RAM location is labelled with a red blob. This corresponds to the Instruction Pointer. Note how the red blob (IP) moves when you step the program.

When doing the learning exercises, add to and modify your own copy of the example.

What you need to know

Comments	Any text after a semicolon is not part of the program and is ignored by the simulator. These comments are used for explanations of what the program is doing. Good programmers make extensive use of comments. Good comments should not just repeat the code. Good comments should explain why things are begin done.
CLO	The CLO command is unique to this simulator. It closes any window that is not needed while a program is running. This command makes it easier to write nice demonstration programs. It also avoids having to close several windows manually.
MOV	The MOV command is short for Move. In this example numbers are being copied into registers where arithmetic can be done. MOV copies data from one location to another. The data in the original location is left intact by the MOV command. Move was shortened to Mov because, in olden times, computer memory was fiendishly expensive. Every command was shortened as much as

	possible, much like the mobile phone texting language used today.
ADD	<p>Arithmetic</p> <p>The add command comes in two versions. Here are two examples</p> <p>ADD AL,BL - Add BL to AL and store the result into AL</p> <p>ADD AL,5 - Add 5 to AL and store the result into AL</p>  <p>Data always moves from right to left as shown by the arrows</p> <p>Look at the on-line help to find out about SUB, MUL and DIV. Remember that you can access on-line help by pressing the F1 key.</p>
Registers	<p>Registers are storage locations where 8 bit binary numbers are stored. The central processing unit in this simulator has four general purpose registers called AL, BL, CL and DL. These registers are interchangeable and can, with a few exceptions, be used for any purpose.</p> <p>Newer central processing unit (CPU) chips have 16, 32 or even 64 bit registers. These work in the same way but more data can be moved in one step so there is a speed advantage.</p> <p>Wider registers can store larger integer (whole) numbers. This simplifies many programming tasks. The other three registers SP, IP and SR are described later.</p>
Hexadecimal Numbers	<p>In the command MOV AL,2 the 2 is a hexadecimal number. The hexadecimal number system is used in low level programming because there is a very convenient conversion between binary and hex. Study the Hexadecimal and Binary number systems.</p>
END	<p>The last command in all programs should be END. Any text after the END keyword is ignored.</p>

Your Tasks

Use all the registers AL, BL, CL and DL and experiment with ADD, SUB, MUL and DIV.

Find out what happens if you try to divide by zero.



Example - 99nasty.asm - Nasty

Contents

This example shows how you can create totally unreadable code.

Try not to do this.

This program actually works. Copy it and paste it into the simulator and try it!

Click the List-File tab to see the code laid out better and to see the addresses where the code is stored.

To get back to the editor window click the Source-Code tab.

Example - 99nasty.asm

```
; ----- Here is how NOT to write a program -----
_: Mov BL,C0 Mov AL,3C Q: Mov [BL],AL CMP AL,7B
JZ Z INC AL INC BL JMP Q Z: MOV CL,40 MOV AL,20
MOV BL,C0 Y: MOV [BL],AL INC BL DEC CL JNZ Y JMP
_ END ; Look at the list file. It comes out OK!
; Press Escape to stop the program running.
; -----
```

Here it is tidied up

```
; ----- A Program to display ASCII characters -----
; ----- Here it is tidied up. This version is annotated. -----
; ----- This makes it possible to understand. -----
; ----- The labels have been given more readable names too. ---

Start:
    Mov BL,C0          ; Make BL point to video RAM
    Mov AL,3C          ; 3C is the ASCII code of the 'less than' symbol
Here:
    Mov [BL],AL        ; Copy the ASCII code in AL to the RAM location that BL is
pointing to.
    CMP AL,7B          ; Compare AL with '{'
    JZ Clear           ; If AL contained '{' jump to Clear:
    INC AL              ; Put the next ASCII code into AL
    INC BL              ; Make BL point to the next video RAM location
    JMP Here           ; Jump back to Here
Clear:
    MOV CL,40          ; We are going to repeat 40 (hex) times
    MOV AL,20           ; The ASCII code of the space character
    MOV BL,C0          ; The address of the start of video RAM
Loop:
    MOV [BL],AL        ; Copy the ASCII space in AL to the video RAM that BL is
pointing to.
    INC BL              ; Make BL point to the next video RAM location
    DEC CL              ; CL is counting down towards zero
    JNZ Loop           ; If CL is not zero jump back to Loop
    JMP Start          ; CL was zero so jump back to the Start and do it all again.

    END

; -----
```

Your Task

Write all your future programs ...

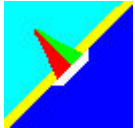
- with good layout
- with meaningful label names
- with useful comments that **EXPLAIN** the code
- avoiding comments that state the totally obvious and just repeat the code

Bad Comment - just repeats the code

```
INC BL ; Add one to BL
```

Useful Comment - explains why the step is needed

```
INC BL ; Make BL point to the next video RAM location
```



Example - 02tlight.asm - Traffic Lights

Contents

Example - 02tlight.asm

```
; ===== CONTROL THE TRAFFIC LIGHTS =====

Start:      CLO                ; Close unwanted windows.

            MOV AL,0           ; Turn off all the traffic lights.
            OUT 01             ; Copy 00000000 into the AL register.
                                ; Send AL to Port One (The traffic lights).
            MOV AL,FC          ; Turn on all the traffic lights.
            OUT 01             ; Copy 11111100 into the AL register.
                                ; Send AL to Port One (The traffic lights).
            JMP Start          ; Jump back to the start.
            END                ; Program ends.

; ===== Program Ends =====

YOUR TASK
=====
Use the help page on Hexadecimal and ASCII codes.
Work out what hexadecimal numbers will activate the
correct traffic lights. Modify the program to step
the lights through a realistic sequence.
```

To run the program press the Step button repeatedly or press the Run button.

To stop the program, press Stop. When the program is running, click the RAM-Source or RAM-Hex or RAM-ASCII tabs. These give alternative views of the contents of random access memory (RAM).

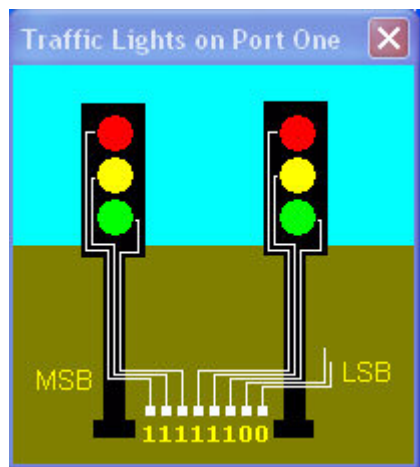
Also click the List File tab to see the machine code generated by the simulator and the addresses where the codes are stored.

Ports

The traffic lights are connected to port one. Imagine this as a socket on the back of the processor box. Data sent to port one goes to the traffic lights and controls them.

There are six lamps to control. Red, Amber and Green for a pair of lights. This can be achieved with a single byte of data where two bits are unused.

By setting the correct bits to One, the correct lamps come on.



Fill in the rest of this table to work out the Hexadecimal values you need. Of course you need to know the sequence of lights in your country.

Red	Amber	Green	Red	Amber	Green	Not used	Not used	Hex
1	0	0	0	0	1	0	0	84

What you need to know

Labels and Jumps

Labels mark positions that are used by Jump commands. All the commands in this program are repeated for ever or until Stop is pressed. Label names must start with a letter or _ character. Label names must not start with a digit. The line

JMP Start

causes the program to jump back and re-do the earlier commands.

Destination labels end in a colon. For example

Start:

Controlling the Lights

If you look carefully at the traffic lights display, you can see which bit controls each light bulb. Work out the pattern of noughts and ones needed to turn on a sensible set of bulbs. Use the Hexadecimal and Binary numbers table to work out the hexadecimal equivalent. Move this hexadecimal number into AL.

OUT 01

This command copies the contents of the AL register to Output Port One. The traffic lights are connected to port one. A binary one causes a bulb to switch on. A nought causes it to turn off.



Example - 03move.asm - Data Moves

Contents

Example - 03move.asm

```
; -----  
; A program to demonstrate MOV commands. Mov is short for move.  
; -----  
        CLO                ; Close unwanted windows.  
; ===== IMMEDIATE MOVES =====  
        MOV     AL,15      ; Copy 15 HEX into the AL register  
        MOV     BL,40      ; Copy 40 HEX into the BL register  
        MOV     CL,50      ; Copy 50 HEX into the CL register  
        MOV     DL,60      ; Copy 60 HEX into the DL register  
Foo:  
        INC     AL          ; Increment AL for no particular reason.  
  
; ===== INDIRECT MOVES =====  
        MOV     [A0],AL    ; Copy value in AL to RAM location [40]  
        MOV     BL,[40]    ; Copy value in RAM location [A0] into BL  
  
; ===== REGISTER INDIRECT MOVES =====  
        MOV     [CL],AL    ; Copy the value in AL to the RAM  
                           ; location that CL points to.  
        MOV     BL,[CL]    ; Copy the RAM location that CL points  
                           ; to into the BL register.  
  
        JMP     Foo        ; PRESS ESCAPE TO STOP THE PROGRAM  
  
        END  
; -----  
TASK  
=====  
Look up the ASCII codes of the letters in H,E,L,L,O and move  
these ASCII codes to RAM addresses [C0], [C1], [C2], [C3]  
and [C4]. Run the program and watch how the text appears on  
the simulated VDU display. This is very much the same as what  
happens in the IBM PC running MS DOS. The program you write  
should work but if you continue to study low level programming,  
you will find much more efficient and flexible ways of solving  
this problem.
```

Step through the program and watch the register values changing. In particular, look at the RAM-Hex display and note the way that values in RAM change. Addresses [50] and [A0] are altered. You can copy the example program from the help page and paste it into the source code editor.

Addressing Modes

There are several **ADDRESSING MODES** available with move commands.

Immediate Addressing

A hexadecimal number is copied into a register. **Examples...**

MOV AL,15 ; Copy 15 HEX into the AL register
MOV BL,40 ; Copy 40 HEX into the BL register
MOV CL,50 ; Copy 50 HEX into the CL register
MOV DL,60 ; Copy 60 HEX into the DL register

Indirect Addressing

A value is moved to or from RAM. The ram address is given as a number like [22] in square brackets. **Examples...**

MOV [A0],AL ; Copy value in AL to RAM location [40]
MOV BL,[40] ; Copy value in RAM location [A0] into BL

Register Indirect Addressing

Copy a value from RAM to a register or copy a value from a register to RAM. The RAM address is contained in a second register enclosed in square brackets like this [CL]. **Examples ...**

MOV [CL],AL ; Copy the value in AL to the RAM location that CL points to.
MOV BL,[CL] ; Copy the RAM location that CL points to into the BL register.

Register Moves

Not available in this simulation.

A register move looks like this

MOV AL,BL

To do this using simulator commands, use

PUSH BL
POP AL

Push and Pop are explained later.

Calculated Addresses

Not available in this simulator.

Copy a value from RAM to a register or copy a value from a register to RAM. The RAM address is contained in square brackets and is calculated. This is done to simplify access to record structures. For example a surname might be stored 12 bytes from the start of the record. This technique is shown in the examples below.

MOV [CL + 5],AL ; Copy the value in AL to the RAM location that CL + 5 points to.
MOV BL,[CL + 12] ; Copy the RAM location that CL + 12 points to into the BL register.

Implied Addresses

Not available in this simulator.

In this case, memory locations are named. Address [50] might be called 'puppy'. This means that moves can be programmed like this.

MOV AL,puppy ; Copy the value in RAM at position puppy into the AL register.
MOV puppy,BL ; Copy BL into the RAM location that puppy refers to.



Example - 04IncJmp.asm - Counting

Contents

Example - 04IncJmp.asm

```
; ===== Counting =====  
  
      MOV     BL,40    ; Initial value stored in BL  
  
Rep:      ; Jump back to this label  
      INC     BL      ; Add ONE to BL  
      JMP     Rep      ; Jump back to Rep  
  
      END          ; Program Ends  
  
; ===== Program Ends =====  
  
TASK  
=====
```

Rewrite the program to count backwards using DEC BL.

Rewrite the program to count in threes using ADD BL,3.

Rewrite the program to count 1 2 4 8 16 using MUL BL,2

Here is a more difficult task.
Count 0 1 1 2 3 5 8 13 21 34 55 98 overflow.
Here each number is the sum of the previous two.
You will need to use registers or RAM locations
for temporary storage of the numbers.
If you have never programmed before, this is a real brain teaser.
Remember that the result will overflow when it goes above 127.

This number sequence was first described by
Leonardo Fibonacci of Pisa (1170_1230)

The program counts up in steps of one until the total is too big to be stored in a single byte. At this point the calculation overflows. Watch the values in the registers. In particular, watch IP and SR. These are explained below.

Although this program is very simple, some new ideas are introduced.

MOV BL,40

This line initialises BL to 40.

Rep:

Rep: is a label. Labels are used with Jump commands. It is possible for programs to jump backwards or forwards. Because of the way numbers are stored, the largest jumps are -128 backwards and +127 forwards. Labels must begin with a letter or the _ character. Labels may contain letters, digits and the _ character. Destination labels must end with a Colon:

INC BL

This command adds one to BL. Watch the BL register. It will count up from 40 in hexadecimal so after 49 comes 4A, 4B, 4C, 4D, 4E, 4F, 50, 51 and so on.

Overflow

When BL reaches 7F hex or 127 in decimal numbers the next number ought to be 128 but because of the way numbers are stored in binary, the next number is minus 128. This effect is called an **OVERFLOW**.

Status Register (SR)

The status register labelled SR contains four flag bits that give information about the state of the CPU. There are three flags that indicate whether a calculation overflowed, gave a negative result or gave a zero result. Calculations set these flags

- S The sign flag indicates a negative result.
- O The overflow flag indicates overflows.
- Z The zero flag indicates a zero result.
- I Interrupts enabled. STI turns this on. CLI turns this off.

These flags are described in more detail later.

JMP Rep

This command causes the central processing unit (CPU) to jump back and repeat earlier commands or jump forward and skip some commands.

Instruction Pointer (IP)

The instruction pointer labelled IP contains the address of the instruction being executed. This is indicated by a red highlighted RAM position in the simulator. Each CPU command causes the IP to be increased by 1, 2 or 3 depending on the size of the command. In the RAM displays, the instruction pointer is highlighted red with yellow text.

```
NOP           ; Increase IP by 1
INC BL        ; Increase IP by 2
ADD AL,BL     ; Increase IP by 3
JMP Rep       ; Add or subtract a value from IP to
               ; jump to a new part of the program.
```

Fetch Execute Cycle

Fetch the instruction. IP points to it. This is called the operator.

If necessary, **fetch** data. IP + 1 points to it. This is the first operand.

If necessary, **fetch** data. IP + 2 points to it. This is the second operand.

Execute the command. This may involve more fetching or putting of data.

Increase IP to point to the next command or **calculate IP** for Jump commands.

Repeat this cycle.

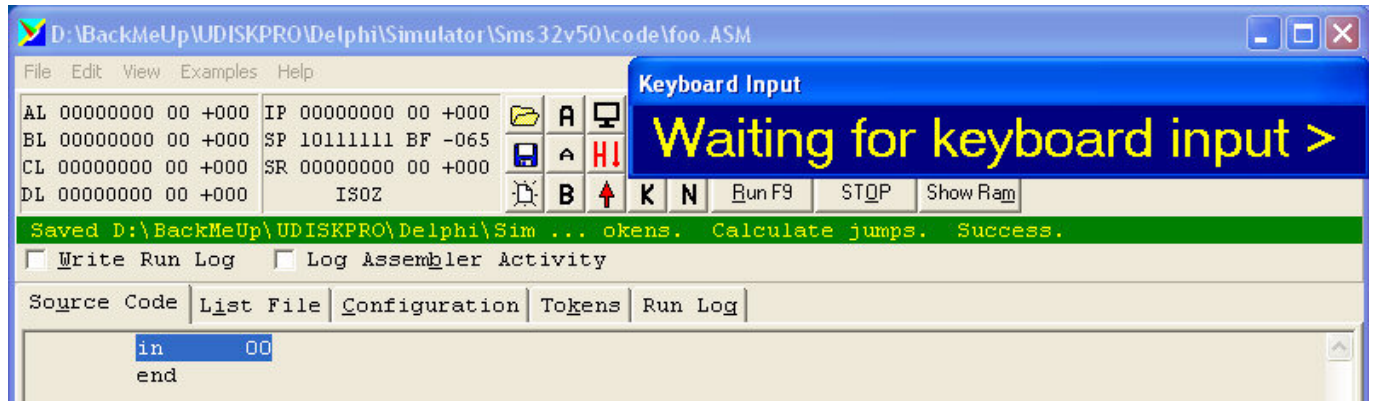
Every machine cycle has one operator or instruction. There could be zero, one or two operands depending on the instruction. OP Codes are the machine codes that correspond to the operators and operands.



Example - 05keyb-in.asm - Keyboard Input

[Contents](#)

Example - 05keyb-in.asm



```
; -----  
; Input key presses from the keyboard until Enter is pressed.  
; -----  
Rep:    CLO                ; Close unwanted windows.  
        IN      00        ; Wait for key press - Store it in AL.  
        CMP     AL,0D      ; Was it the Enter key? (ASCII 0D)  
        JNZ     Rep       ; No - jump back. Yes - end.  
  
END  
;  
TASK  
  
11)    Easy!  Display each character at the top left position of the  
        VDU by copying them all to address [C0].  
  
12)    Harder Use BL to point to address [C0] and increment BL after  
        each key press in order to see the text as you type it.  
  
13)    Harder! Store all the text you type in RAM when you type it in.  
        When you press Enter, display the stored text on the VDU display.  
  
14)    Difficult Type in text and store it. When Enter is pressed,  
        display it on the VDU screen in reverse order. Using the stack  
        makes this task easier.
```

You can copy this example program from the help page and paste it into the source code editor.

IN 00

Input from port zero. In this simulator, port zero is wired to the keyboard hardware. The simulator waits for a key press and copies the ASCII code of the key press into the AL register. This is not very realistic but is easy to program. There is a more realistic keyboard on port 07 and interrupt 03 but this is for more advanced programmers.

CMP AL,0D

Compare the AL register with the ASCII code of the Enter key. The ASCII code of the Enter key is 0Dhex.

CMP AL,BL works as follows. The processor calculates AL - BL. If the result is zero, the 'Z' flag in the status register SR is set. If the result is negative, the 'S' flag is set. If the result is positive, no flags are set. The 'Z' flag is set if AL and BL are equal. The 'S' flag is set if BL is greater than AL. No flag is set if AL is greater than BL.

JNZ Rep

JNZ stands for Jump Not Zero. Jump if the 'Z' flag is not set. The program will jump forwards or back to the address that Rep marks.

A related command is JZ. This stands for Jump Zero. Jump if the zero flag is set. In this program, the CMP command sets the flags. Arithmetic commands also set the status flags.

MOV [C0],AL

This will copy AL to address [C0]. The visual display unit works with addresses [C0] to [FF]. This gives a display with 4 rows and 16 columns. Address [C0] is the top left corner of the screen.

MOV [BL],AL

This copies AL to the address that BL points to. BL can be made to point to the VDU screen at [C0] by using MOV BL,C0. BL can be made to point to each screen position in turn by using INC BL. This is needed for task 2.



Example - 06proc.asm - Procedures

Contents

Example - 06proc.asm

```
; -----  
  
; A general purpose time delay procedure.  
  
; The delay is controlled by the value in AL.  
  
; When the procedure terminates, the CPU registers are  
; restored to the same values that were present before  
; the procedure was called. Push, Pop, Pushf and Popf  
; are used to achieve this. In this example one procedure  
  
; is re-used three times. This re-use is one of the main  
; advantages of using procedures.  
  
;----- The Main Program -----  
Start:  
    MOV     AL,8      ; A short delay.  
    CALL    30        ; Call the procedure at address [30]  
  
    MOV     AL,10     ; A middle sized delay.  
    CALL    30        ; Call the procedure at address [30]  
  
    MOV     AL,20     ; A Longer delay.  
    CALL    30        ; Call the procedure at address [30]  
  
JMP     Start      ; Jump back to the start.
```



```

; ----- Time Delay Procedure Stored At Address [30] -----
ORG      30          ; Generate machine code from address [30]

    PUSH     AL        ; Save AL on the stack.
    PUSHF    ; Save the CPU flags on the stack.
Rep:    DEC     AL        ; Subtract one from AL.
        JNZ     REP      ; Jump back to Rep if AL was not Zero.

        POPF    ; Restore the CPU flags from the stack.
        POP     AL      ; Restore AL from the stack.

        RET     ; Return from the procedure.
; -----
        END
; -----

TASK

15)    Re-do the traffic lights program and use this procedure
        to set up realistic time delays.  02tlight.asm

16)    Re-do the text input and display program with procedures.
        Use one procedure to input the text and one to display it.

; -----

```

You can copy this example program from the help page and paste it into the source code editor.

MOV AL,8

A value is placed into the AL register before calling the time delay procedure. This value determines the length of the delay.

CALL 30

Call the procedure at address [30]. This alters the instruction pointer IP to [30] and the program continues to run from that address. When the CPU reaches the RET command it returns to the address that it came from. This return address is saved on the stack.

Stack

This is a region in memory where values are saved and restored. The stack uses the Last In First Out rule. LIFO. The CALL command saves the return address on the stack. The RET command gets the saved value from the stack and jumps to that address by setting IP.

ORG 30

Origin at address [30]. ORG specifies at what RAM address machine code should be generated. The time delay procedure is stored at address [30].

PUSH AL

Save the value of AL onto the stack. The CPU stack pointer SP points to the next free stack location. The push command saves a value at this position. SP is then moved back one place to the next free position. In this simulator, the stack grows towards address Zero. A stack overflow occurs if the stack tries to fill more than the available memory. A stack underflow occurs if you try to pop an empty stack.

PUSHF

Save the CPU flags in the status register SR onto the stack. This ensures that the flags can be put back as they were when the procedure completes. The stack pointer is moved back one place. See the Push command. **NOTE:** Items must be popped in the reverse order they were pushed.

DEC AL

Subtract one from AL. This command sets the Z flag if the answer was Zero or the S flag if the answer was negative.

JNZ REP

Jump Not Zero to the address that Rep marks. Jump if the Z flag is not set.

POPF

Restore the CPU flags from the stack. Increase the stack pointer by one.

POP AL

Restore the AL register from the stack. This is done by first moving the stack pointer SP forward one place and copying the value at that stack position into the AL register. A stack underflow occurs when an attempt is made to pop more items off the stack than were present. **NOTE:** Items must be popped in the reverse order they were pushed.

RET

Return from the procedure to the address that was saved on the stack by the CALL command. Procedures can re-use themselves. This is called recursion. It is a powerful technique and dangerous if you don't understand what is happening! Accidental or uncontrolled recursion causes the stack to grow until it overwrites the program or overflows.



Example - 07textio.asm - Text I/O Procedures

[Contents](#)

Example - 07textio.asm

```
; -----  
; A program to read in a string of text and store it in RAM.  
; The end of text will be labelled with ASCII code zero/null.  
; -----  
; THE MAIN PROGRAM  
    MOV     BL,70      ; [70] is the address where the text will  
                        ; be stored. The procedure uses this.  
  
    CALL    10          ; The procedure at [10] reads in text and  
                        ; places it starting from the address  
                        ; in BL.  
  
                        ; BL should still contain [70] here.  
  
    CALL    40          ; This procedure does nothing until you  
                        ; write it. It should display the text.  
  
    HALT              ; DON'T USE END HERE BY MISTAKE.
```

```

; -----
; A PROCEDURE TO READ IN THE TEXT
    ORG     10      ; Code starts from address [10]

    PUSH    AL      ; Save AL onto the stack
    PUSH    BL      ; Save BL onto the stack
    PUSHF    ; Save the CPU flags onto the stack

Rep:
    IN       00      ; Input from port 00 (keyboard)
    CMP      AL,0D    ; Was key press the Enter key?
    JZ       Stop    ; If yes then jump to Stop
    MOV      [BL],AL  ; Copy keypress to RAM at position [BL]
    INC      BL      ; BL points to the next location.
    JMP      Rep     ; Jump back to get the next character

Stop:
    MOV      AL,0     ; This is the NULL end marker
    MOV      [BL],AL  ; Copy NULL character to this position.

    POPF      ; Restore flags from the stack
    POP      BL     ; Restore BL from the stack
    POP      AL     ; Restore AL from the stack

    RET      ; Return from the procedure.
; -----
; A PROCEDURE TO DISPLAY TEXT ON THE SIMULATED SCREEN
    ORG     40      ; Code starts from address [10]
                    ; **** YOU MUST FILL THIS GAP ****

    RET      ; At present this procedure does
                    ; nothing other than return.

; -----
    END      ; It is correct to use END at the end.
; -----

TASK

17) Write a program using three procedures. The first should
    read text from the keyboard and store it in RAM. The second
    should convert any upper case characters in the stored text
    to lower case. The third should display the text on the
    VDU screen.

; -----

```

You can copy this example program from the help page and paste it into the source code editor.

Passing Parameters

MOV BL,70

The BL register contains 70. This value is needed by the text input procedure. It is the address where the text will be stored in RAM. This is an example of passing a parameter using a register. All you are doing is getting a number from one part of a program to another.

INC BL

This command adds one to BL. The effect is to make BL point to the next memory location ready for the next text character to be stored.

CALL 10

Call the procedure at address [10]. This is achieved in practice by setting the CPU instruction pointer IP to [10].

RET

At the end of the procedure, the RET command resets the CPU instruction pointer IP back to the instruction after the CALL instruction to the procedure. This address was stored on the stack by the CALL instruction.

HALT

Don't confuse HALT and END. The END command causes the assembler to stop scanning for more instructions in the program. The HALT command generates machine code 00 which causes the CPU to halt. There can be several HALT commands in a program but only one END command.

ORG 10

Origin [10]. The assembler program starts generating machine code from address [10].

PUSH AL and POP AL

Save the value of AL onto the stack. This is an area of RAM starting at address BF. The stack grows towards zero. The RAM displays show the stack pointer as a blue highlight with yellow text. Push and Pop are used so that procedures and interrupts can tidy up after themselves. The procedure or interrupt can alter CPU registers but it restores them to their old values before returning.

PUSHF and POPF

PUSHF saves the CPU flags onto the stack. POPF restores the CPU flags to their original value. This enables procedures and interrupts to do useful work without unexpected side effects on the rest of the program.

IN 00

Input from port zero. This port is connected to the keyboard. The key press is stored into the AL register.

CMP AL,0D

Compare the AL register with the hexadecimal number 0D. 0D is the ASCII code of the Enter key. This line is asking "Was the enter key pressed?" CMP works by subtracting 0D from AL. If they were equal then the subtraction gives an answer of zero. This causes the CPU zero or 'Z' flag to be set.

JZ Stop

Jump to the Stop label if the CPU 'Z' flag was set. This is a conditional jump.

MOV [BL],AL

Move the key press stored in AL into the RAM location that [BL] points to. INC BL is then used to make BL point to the next RAM location.

JMP Rep

Jump back to the Rep label. This is an unconditional jump. It always jumps and the CPU flags are ignored.

RET

Return from the procedure to the address stored on the stack. This is done by setting the instruction pointer IP in the CPU.



Example - 08table.asm - Data Tables

Contents

Example - 08table.asm

```
; ----- EXAMPLE 8 ----- DATA TABLES -----

        JMP      Start    ; Skip past the data table.

        DB       84       ; Data table begins.
        DB       C8       ; These values control the traffic lights
        DB       31       ; This sequence is simplified.
        DB       58       ; Last entry is also used as end marker

Start:
        MOV      BL,02    ; 02 is start address of data table
Rep:
        MOV      AL,[BL]  ; Copy data from table to AL
        OUT      01      ; Output from AL register to port 01

        CMP      AL,58    ; Last item in data table ???
        JZ       Start   ; If yes then jump to Start
        INC      BL      ; In no then point BL to the next entry
        JMP      Rep     ; Jump back to do next table entry

        END

; -----

TASK

18)      Improve the traffic lights data table so there is an
        overlap with both sets of lights on red.

19)      Use a data table to navigate the snake through the maze.
        This is on port 04. Send FF to the snake to reset it.
        Up, down left and right are controlled by the left four bits.
        The right four bits control the distance moved.

20)      Write a program to spin the stepper motor. Activate bits
        1, 2, 4 and 8 in sequence to energise the electromagnets
        in turn. The motor can be half stepped by turning on pairs
        of magnets followed by a single magnet followed by a pair
        and so on.

21)      Use a data table to make the motor perform a complex sequence
        of forward and reverse moves. This is the type of control
        needed in robotic systems, printers and plotters. For this
        exercise, it does not matter exactly what the motor does.

; -----
```

You can copy this example program from the help page and paste it into the source code editor.

DB 84

DB stands for Define Byte/s. In this case 84hex is stored into RAM at address [02]. Addresses [00] and [01] are occupied by the JMP Start machine codes.

84 hex is 1000 0100 in binary. This is the pattern of noughts and ones needed to turn on the left red light and the right green light.

MOV BL,02

Move 02 into the BL register. [02] is the RAM address of the start of the data table. BL is used as a pointer to the data table.

MOV AL,[BL]

[BL] points to the data table. This line copies a value from the data table into the AL register.

OUT 01

Send the contents of the AL register to port 01. Port 01 is connected to the traffic lights.

CMP AL,58

58 is the last entry in the data table. If AL contains 58, it is necessary to reset BL to point back to the start of the table ready to repeat the sequence. If AL is equal to 58, the 'Z' flag in the CPU will be set.

JZ Start

Jump back to start if the 'Z' flag in the CPU is set.

INC BL

Add one to BL to make it point to the next entry in the data table.



Example - 09param.asm - Parameters

Contents

Example - 09param.asm

```
; ----- EXAMPLE 9 ----- Passing Parameters -----  
  
; ----- Use Registers to pass parameters into a procedure -----  
  
        JMP      Start    ; Skip over bytes used for data storage  
  
        DB       0        ; Reserve a byte of RAM at address [02]  
        DB       0        ; Reserve a byte of RAM at address [03]  
Start:  
        MOV      AL,5  
        MOV      BL,4  
        CALL     30        ; A procedure to add AL to BL  
                           ; Result returned in AL.  
  
; ----- Use RAM locations to pass parameters into a procedure --  
  
        MOV      AL,3  
        MOV      [02],AL ; Store 3 into address [02]  
        MOV      BL,1  
        MOV      [03],BL ; Store 1 into address [03]  
        CALL     40  
  
; ----- Use the Stack to pass parameters into a procedure -----  
        MOV      AL,7
```

```

    PUSH    AL
    MOV     BL,2
    PUSH    BL
    CALL    60
    POP     BL
    POP     AL      ; This one contains the answer

    JMP     Start   ; Go back and do it again.

; ----- A procedure to add two numbers -----
;
; Parameters passed into procedure using AL and BL
; Result returned in AL
; This method is simple but is no good if there are a
; lot of parameters to be passed.

    ORG     30      ; Code starts at address [30]

    ADD     AL,BL   ; Do the addition.  Result goes into AL

    RET                      ; Return from the procedure

; ----- A procedure to add two numbers -----
;
; Parameters passed into procedure using RAM locations.
; Result returned in RAM location

; This method is more complex and there is no limit on
; the number of parameters passed unless RAM runs out.

    ORG     40      ; Code starts at address [40]

    PUSH    CL      ; Save registers and flags on the stack
    PUSH    DL
    PUSHF

    MOV     CL,[02] ; Fetch a parameter from RAM
    MOV     DL,[03] ; Fetch a parameter from RAM
    ADD     CL,DL   ; Do the addition
    MOV     [02],CL ; Store the result in RAM

    POPF          ; Restore original register

    POP     DL      ; and flag values
    POP     CL

    RET

; ----- A procedure to add two numbers -----
;
; The numbers to be added are on the stack.
; POP parameters off the stack
; Do the addition
; Push answer back onto the stack
; The majority of procedure calls in real life make use
; of the stack for parameter passing.  It is very common
; for the address of a complex data structure in RAM to
; be passed to a procedure using the stack.

    ORG     60      ; Code starts at address [60]

    POP     DL      ; Return address
    POP     BL      ; A parameter
    POP     AL      ; A parameter

    ADD     AL,BL

```



```

PUSH    AL        ; Answer ; The number of pushes must
PUSH    AL        ; Answer ; match the number of pops.
PUSH    DL        ; Put the stack back as it was before

```

```

RET

```

```

; -----

```

```

END

```

Task

- 22) Write a procedure that doubles a number. Pass the single parameter into the procedure using a register. Use the same register to return the result.
- 23) Write a procedure to invert all the bits in a byte. All the zeros should become ones. All the ones should become zeros. Pass the value to be processed into the procedure using a RAM location. Return the result in the same RAM location.
- 24) Write a procedure that works out Factorial N. This example shows one method for working out factorial N. Factorial 5 is $5 * 4 * 3 * 2 * 1 = 120$. Your procedure should work properly for factorial 1, 2, 3, 4 or 5. Factorial 6 would cause an overflow. Use the stack to pass parameters and return the result. Calculate the result. Using a look up table is cheating!
- 25) Write a procedure that works out Factorial N. Use the stack for parameter passing. Write a recursive procedure. Use this definition of Factorial.
- Factorial (0) is defined as 1.
 Factorial (N) is defined as $N * \text{Factorial} (N - 1)$.
- To work out Factorial (N), the procedure first tests to see if N is zero and if not then re-uses itself to work out $N * \text{Factorial} (N - 1)$. This problem is hard to understand in any programming language. In assembly code it is harder still.

You can copy this example program from the help page and paste it into the source code editor.

Passing Parameters

Parameters can be passed in three ways.

1. **CPU registers can be used** - Fast but little data can be passed. In some programming languages the "Register" keyword is used to achieve this.
2. **RAM locations can be used** - Slower and recursion may not be possible. In some programming languages the "Static" keyword is used to achieve this. This technique is useful if very large amounts of data are held in RAM. Passing a pointer to the data is more efficient than making a copy of the data on the stack.
3. **The stack can be used** - Harder to understand and code but a lot of data can be passed and recursion is possible. Compilers generally use this method by default unless otherwise directed.

The example program uses all three methods to add two numbers together. The example tasks involve all three methods.



Example - 10swint.asm

Software Interrupts

Contents

Example - 10swint.asm

```
; -----  
; An example of software interrupts.  
; -----  
        JMP     Start    ; Jump past table of interrupt vectors  
        DB      51       ; Vector at 02 pointing to address 51  
        DB      71       ; Vector at 03 pointing to address 71  
Start:  
        INT     02       ; Do interrupt 02  
        INT     03       ; Do interrupt 03  
        JMP     Start  
; -----  
        ORG     50  
        DB      E0       ; Data byte - could be a whole table here  
                        ; Interrupt code starts here  
        MOV     AL,[50]  ; Copy bits from RAM into AL  
        NOT     AL       ; Invert the bits in AL  
        MOV     [50],AL  ; Copy inverted bits back to RAM  
        OUT     01       ; Send data to traffic lights  
        IRET  
; -----  
        ORG     70  
        DB      0        ; Data byte - could be a table here  
; Interrupt code starts here  
        MOV     AL,[70]  ; Copy bits from RAM into AL  
        NOT     AL       ; Invert the bits in AL  
        AND     AL,FE    ; Force right most bit to zero  
        MOV     [70],AL  ; Copy inverted bits back to RAM  
        OUT     02       ; Send data to seven segment display  
        IRET  
; -----  
        END  
; -----  
  
TASK  
  
26)      Write a new interrupt 02 that fetches a key press from the  
         keyboard and stores it into RAM.  The IBM PC allocates 16  
         bytes for key press storage.  The 16 locations are used in  
         a circular fashion.  
  
27)      Create a new interrupt that puts characters onto the next  
         free screen location.  See if you can get correct behaviour  
         in response to the Enter key being pressed (fairly easy)  
         and if the Back Space key is pressed (harder).
```

You can copy this example program from the help page and paste it into the source code editor.

Interrupts and Procedures

Interrupts are short code fragments that provide useful services that can be used by other programs. Typical routines handle key presses, mouse movements and button presses, screen writing, disk reading and writing and so on.

An interrupt is like a procedure but it is called in a different way. Procedures are called by jumping to the start address of the procedure. This address is known only to the program that owns the procedure. Interrupts are called by looking up the address of the interrupt code in a table of interrupt vectors. The contents of this table is published and widely known. MS DOS makes heavy use of interrupts for all its disk, screen, mouse, network, keyboard and other services.

By writing your own code and making the interrupt vector point to the code you wrote, the behaviour of interrupts can be completely altered. Your interrupt code might add some useful behaviour and then jump back to the original code to complete the work. This is called TRAPPING the interrupt.

Software interrupts are triggered, on demand, by programs.

Hardware interrupts are triggered by electronic signals to the CPU from hardware devices.

Interrupt Vector Table

In the IBM compatible computer, addresses 0 to 1024 decimal are used for storing interrupt vectors. The entries in this table of vectors point to all the code fragments that control MS DOS screen, disk, mouse, keyboard and other services. The simulator vectors sit between addresses 0 and 15 decimal. It is convenient to start a simulator program with a jump command that occupies two bytes. This means that the first free address for an interrupt vector is [02]. This is used by the hardware timer if the interrupt flag is set.

Have another look at the example program. 10swint.asm

Calling an Interrupt

This is quite complex. The command INT 02 causes the CPU to retrieve the contents of RAM location 02. After saving the return address onto the stack, the instruction pointer IP is set to this address.

The interrupt code is then executed. When complete the IRET command causes the return from the interrupt. The CPU instruction pointer IP is set to the address that was saved onto the stack earlier.

Trapping an Interrupt

If you want to trap interrupt 02, change the address stored at address 02 to point to code that you have written. Your code will then handle the interrupt. When complete, your code can use IRET to return from the interrupt or it can jump to the address that was originally in address 02. This causes the original interrupt code to be executed as well. In this way, you can replace or modify the behaviour of an interrupt.



Example - 11hwint.asm

Hardware Interrupts

Contents

Example - 11hwint.asm

```
; -----
; An example of using hardware interrupts.
; This program spins the stepper motor continuously and
; steps the traffic lights on each hardware interrupt.

; Uncheck the "Show only one peripheral at a time" box
; to enable both displays to appear simultaneously.

; -----
        JMP     Start    ; Jump past table of interrupt vectors
        DB      50       ; Vector at 02 pointing to address 50

Start:
        STI             ; Set I flag. Enable hardware interrupts
        MOV     AL,11    ;

Rep:
        OUT      05      ; Stepper motor
        ROR      AL      ; Rotate bits in AL right
        JMP      Rep
        JMP      Start

; -----
        ORG      50

        PUSH     al       ; Save AL onto the stack.
        PUSH     bl       ; Save BL onto the stack.
        PUSHF          ; Save flags onto the stack.

        JMP      PastData

        DB      84        ; Red           Green
        DB      c8        ; Red+Amber     Amber
        DB      30        ; Green         Red
        DB      58        ; Amber         Red+Amber
        DB      57        ; Used to track progress through table

PastData:
        MOV      BL,[5B]  ; BL now points to the data table
        MOV      AL,[BL]  ; Data from table goes into AL
        OUT      01       ; Send AL data to traffic lights
        CMP      AL,58    ; Last entry in the table
        JZ       Reset   ; If last entry then reset pointer

        INC      BL       ; BL points to next table entry
        MOV      [5B],BL  ; Save pointer in RAM
        JMP      Stop

Reset:
        MOV      BL,57    ; Pointer to data table start address
        MOV      [5B],BL  ; Save pointer into RAM location 54

Stop:
        POPF          ; Restore flags to their previous value
        POP      bl     ; Restore BL to its previous value
        POP      al     ; Restore AL to its previous value

        IRET

; -----
```

END

; -----

TASK

- 28) Write a program that controls the heater and thermostat whilst at the same time counting from 0 to 9 repeatedly, displaying the result on one of the seven segment displays. If you want a harder challenge, count from 0 to 99 repeatedly using both displays. Use the simulated hardware interrupt to control the heater and thermostat.
- 29) A fiendish problem. Solve the Tower of Hanoi problem whilst steering the snake through the maze. Use the text characters A, B, C Etc. to represent the disks. Use three of the four rows on the simulated screen to represent the pillars.
- 30) Use the keyboard on Port 07. Write an interrupt handler (INT 03) to process the key presses. You must also process INT 02 (the hardware timer) but it need not perform any task. For a more advanced task, implement a 16 byte circular buffer. Write code to place the buffered text on the VDU screen when you press Enter. For an even harder task, implement code to process the Backspace key to delete text characters in the buffer.

You can copy this example program from the help page and paste it into the source code editor.

Hardware Interrupts

Hardware Interrupts are short code fragments that provide useful services that can be triggered by items of hardware. When a printer runs out of paper, it sends a signal to the CPU. The CPU interrupts normal processing and processes the interrupt. In this case code would run to display a "Paper Out" message on the screen. When this processing is complete, normal processing resumes.

This simulator has a timer that triggers INT 02 at regular time intervals that you can pre-set in the Configuration Tab. You must put an interrupt vector at address 02 that points to your interrupt code. Look at the example.

STI and CLI

Hardware interrupts are ignored unless the 'I' flag in the status register is set. To set the 'I' flag, use the set 'I' command, STI. To clear the 'I' flag, use the clear 'I' command CLI.

Hardware interrupts can be trapped in the same way that software interrupts can.

Hardware interrupts are triggered, as needed by disk drives, printers, key presses, mouse movements and other hardware events.

This scheme makes processing more efficient. Without interrupts, the CPU would have to poll the hardware devices at regular time intervals to see if any processing was needed. This would happen whether or not processing was necessary. Interrupts can be assigned priorities such that a disk drive might take priority over a printer. It is up to the programmer to optimise all this for efficient processing. In the IBM compatible PC, low number interrupts have a higher priority than the higher numbers.

Calling an Interrupt

This is quite complex. The command INT 02 whether triggered by hardware or software, causes the CPU to retrieve the contents of RAM location 02. After saving the return address onto the stack, the instruction pointer IP is set to the address that came from RAM.

The interrupt code is then executed. When complete the IRET command causes the return from the interrupt. The CPU instruction pointer IP is set to the address that was saved onto the stack earlier.

Hardware interrupts differ slightly from software interrupts. A software interrupt is called with a command like INT 02 and the return address is the next instruction after this. IP + 2 is pushed onto the stack. Hardware interrupts are not triggered by an instruction in a program so the return address does not have to be set past the calling instruction. IP is pushed onto the stack.

Trapping an Interrupt

This is the same as trapping software interrupts described on the [previous](#) page.



Shortcut Keys

Contents

Alt Keys		Control Keys		Function Keys	
A	Assemble Button	A	Edit Select All	F1	Help
B	Log Assembler Activity	B		F2	
C	Configuration Tab	C	Edit Copy	F3	
D		D		F4	
E	Edit Menu	E		F5	
F	File Menu	F	Edit Find	F6	
G	Log File Tab	G		F7	
H	Help Menu	H		F8	
I		I		F9	Run
J	List File Tab	J		F10	
K	Tokens Tab	K		F11	
L	Slower Button	L		F12	
M	Show Ram Button	M			
N	Continue Button	N			
O	Stop Button	O	File Open		
P	Step Button	P			
Q		Q			
R	Run Button	R	Edit Replace		
S	Reset Button	S	File Save		
T	Faster Button	T			
U	Source Code Tab	U			
V	View Menu	V	Edit Paste		
W	Write Run Log	W			
X	Examples Menu	X	Edit Cut		
Y		Y			
Z		Z			



ASCII Codes

Contents

American Standard Code for Information Interchange

The ASCII code has 128 standard characters and a further 128 characters that vary from machine to machine and country to country.

The first 128 ASCII characters are shown here.

Dec		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
	Hex	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	Nul							Bel	Bak	Tab	LF			CR		
16	10											EOF	ESC				
32	20	Spa	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Nul

The codes from 128 to 255 are not shown here.

Codes with special meanings to DOS, Printers, Teletype Machines and ANSI screens.

Decimal		
0	Nul	NULL character. (End of text string marker.)
7	Bel	Bell or beep character.
8	Bak	Backspace character.
9	Tab	TAB character.
10	LF	Line Feed (start a new line).
13	CR	Carriage Return code.
26	EOF	DOS End of text file code.
27	Esc	Escape code. It has special effects on older printers and ANSI screens. ANSI = American National Standards Institute.
32	Spa	Space Character.
255	Nul	NULL character.

Unicode

ASCII is being replaced by the 16 bit Unicode with 65536 characters that represent every text character in every country in the world including those used historically. Most new operating systems software packages support Unicode.



Binary and Hexadecimal

Contents

Converting Between Binary and Hex

The CPU works using binary. Electronically this is done with electronic switches that are either on or off. This is represented on paper by noughts and ones. A single BIT or binary digit requires one wire or switch within the CPU. Usually data is handled in BYTES or multiples of bytes. A Byte is a group of eight bits. A byte looks like this

01001011

This is inconvenient to read, say and write down so programmers use hexadecimal to represent bytes. Converting between binary and hexadecimal is not difficult. First split the byte into two nybbles (half a byte) as follows

0100 1011

Then use the following table

BINARY	HEXADECIMAL	DECIMAL
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	A	10
1 0 1 1	B	11
1 1 0 0	C	12
1 1 0 1	D	13
1 1 1 0	E	14
1 1 1 1	F	15

EXAMPLE

Split the byte into two halves

01001011 becomes **0100 1011**

Using the table above

0100 is 4

1011 is B



Instruction Set Summary

Contents

AL, BL, CL and DL are eight bit, general purpose registers where data is stored.

Square brackets indicate RAM locations. For example [15] means RAM location 15.

Data can be moved from a register into RAM and also from RAM into a register.

Registers can be used as pointers to RAM. [BL] is the RAM location that BL points to.

All numbers are in base 16 (Hexadecimal).

Move Instructions. Flags NOT set.

Assembler		Machine Code		Explanation
MOV	AL,15	D0 00 15	AL = 15	Copy 15 into AL
MOV	BL,[15]	D1 01 15	BL = [15]	Copy RAM[15] into AL
MOV	[15],CL	D2 15 02	[15] = CL	Copy CL into RAM[15]
MOV	DL,[AL]	D3 03 00	DL = [AL]	Copy RAM[AL] into DL
MOV	[CL],AL	D4 02 00	[CL] = AL	Copy AL into RAM[CL]

Direct Arithmetic and Logic. Flags are set.

Assembler		Machine Code	
ADD	AL,BL	A0 00 01	AL = AL + BL
SUB	BL,CL	A1 01 02	BL = BL - CL
MUL	CL,DL	A2 02 03	CL = CL * DL
DIV	DL,AL	A3 03 00	DL = DL / AL
INC	DL	A4 03	DL = DL + 1
DEC	AL	A5 00	AL = AL - 1
AND	AL,BL	AA 00 01	AL = AL AND BL
OR	CL,BL	AB 03 02	CL = CL OR BL
XOR	AL,BL	AC 00 01	AL = AL XOR BL
NOT	BL	AD 01	BL = NOT BL
ROL	AL	9A 00	Rotate bits left. LSB = MSB
ROR	BL	9B 01	Rotate bits right. MSB = LSB
SHL	CL	9C 02	Shift bits left. Discard MSB.
SHR	DL	9D 03	Shift bits right. Discard LSB.

Immediate Arithmetic and Logic. Flags are set.

Assembler		Machine Code	
ADD	AL,12	B0 00 12	AL = AL + 12
SUB	BL,15	B1 01 15	BL = BL - 15
MUL	CL,03	B2 02 03	CL = CL * 3
DIV	DL,02	B6 03 02	DL = DL / 2
AND	AL,10	BA 00 10	AL = AL AND 10
OR	CL,F0	BB 02 F0	CL = CL OR F0
XOR	AL,AA	BC 00 AA	AL = AL XOR AA

Compare Instructions. Flags are set.

Assembler		Machine Code	Explanation
CMP	AL,BL	DA 00 01	Set 'Z' flag if AL = BL. Set 'S' flag if AL < BL.

CMP	BL,13	DB 01 13	Set 'Z' flag if BL = 13. Set 'S' flag if BL < 13.
CMP	CL,[20]	DC 02 20	Set 'Z' flag if CL = [20]. Set 'S' flag if CL < [20].

Branch Instructions. Flags NOT set.

Depending on the type of jump, different machine codes can be generated. Jump instructions cause the instruction pointer (IP) to be altered. The largest possible jumps are +127 bytes and -128 bytes.

The CPU flags control these jumps. The 'Z' flag is set if the most recent calculation gave a Zero result. The 'S' flag is set if the most recent calculation gave a negative result. The 'O' flag is set if the most recent calculation gave a result too big to fit in the register.

Assembler		Machine Code	Explanation
JMP	HERE	C0 12 C0 FE	Increase IP by 12 Decrease IP by 2 (twos complement)
JZ	THERE	C1 09 C1 9C	Increase IP by 9 if the 'Z' flag is set. Decrease IP by 100 if the 'Z' flag is set.
JNZ	A_Place	C2 04 C2 F0	Increase IP by 4 if the 'Z' flag is NOT set. Decrease IP by 16 if the 'Z' flag is NOT set.
JS	STOP	C3 09 C3 E1	Increase IP by 9 if the 'S' flag is set. Decrease IP by 31 if the 'S' flag is set.
JNS	START	C4 04 C4 E0	Increase IP by 4 if the 'S' flag is NOT set. Decrease IP by 32 if the 'S' flag is NOT set.
JO	REPEAT	C5 09 C5 DF	Increase IP by 9 if the 'O' flag is set. Decrease IP by 33 if the 'O' flag is set.
JNO	AGAIN	C6 04 C6 FB	Increase IP by 4 if the 'O' flag is NOT set. Decrease IP by 5 if the 'O' flag is NOT set.

Procedures and Interrupts. Flags NOT set.

CALL, RET, INT and IRET are available only in the registered version.

Assembler		Machine Code	Explanation
CALL	30	CA 30	Save IP on the stack and jump to the procedure at address 30.
RET		CB	Restore IP from the stack and jump to it.
INT	02	CC 02	Save IP on the stack and jump to the address (interrupt vector) retrieved from RAM[02].
IRET		CD	Restore IP from the stack and jump to it.

Stack Manipulation Instructions. Flags NOT set.

Assembler		Machine Code	Explanation
PUSH	BL	E0 01	BL is saved onto the stack.
POP	CL	E1 02	CL is restored from the stack.
PUSHF		EA	SR flags are saved onto the stack.

POPF

EB

SR flags are restored from the stack.

Input Output Instructions. Flags NOT set.

Assembler		Machine Code	Explanation
IN	07	F0 07	Data input from I/O port 07 to AL.
OUT	01	F1 01	Data output to I/O port 07 from AL.

Miscellaneous Instructions. CLI and STI set I flag.

Assembler		Machine Code	Explanation
CLO		FE	Close visible peripheral windows.
HALT		00	Halt the processor.
NOP		FF	Do nothing for one clock cycle.
STI		FC	Set the interrupt flag in the Status Register.
CLI		FD	Clear the interrupt flag in the Status Register.
ORG	40	Code origin	Assembler directive: Generate code starting from address 40.
DB	"Hello"	Define byte	Assembler directive: Store the ASCII codes of 'Hello' into RAM.
DB	84	Define byte	Assembler directive: Store 84 into RAM.



Detailed Instruction Set

Contents

The Full Instruction Set

Arithmetic Logic

Jump Instructions

Move Instructions

Compare Instructions

Stack Instructions

Procedures And Interrupts

Inputs and Outputs

Other Instructions

General Information

CPU Registers

There are four general purpose registers called AL, BL, CL and DL.

There are three special purpose registers. These are

- IP is the instruction pointer.
- SP is the stack pointer.
- SR is the status register. This contains the I, S, O and Z flags.

Flags

Flags give information about the outcome of computations performed by the CPU. Single bits in the status register are used as flags. This simulator has flags to indicate the following.

- S The sign flag is set if a calculation gives a negative result.
- O The overflow flag is set if a result is too big to fit in 8 bits.
- Z The zero flag is set if a calculation gives a zero result.
- I is the hardware interrupts enabled flag.

Most real life CPUs have more than four flags.

Registers and Machine Codes

The registers and their equivalent machine code numbers are shown below.

Register names	AL	BL	CL	DL
Machine codes	00	01	02	03

Example : To add one to the CL register use the instruction

Assembly Code	INC	CL
Machine Code Hex	A4	02
Machine code Binary	10100100	00000010

A4 is the machine instruction for the INC command.

02 refers to the CL register.

The assembler is not case sensitive. mov is the same as MOV and Mov.

Within the simulator, hexadecimal numbers may not have more than two hexadecimal digits.

Hexadecimal numbers

15, 3C and FF are examples of hexadecimal numbers. When using the assembler, all numbers should be entered in hexadecimal. The CPU window displays the registers in binary, hexadecimal and decimal. Look at the [Hexadecimal and Binary](#) page for more detail.

Negative numbers

FE is a negative number. Look at the [Negative Numbers](#) table for details of twos complement numbers.

In a byte, the left most bit is used as a sign bit. This has a value of minus 128 decimal.

Bytes can hold signed numbers in the range -128 to +127.

Bytes can hold unsigned numbers in the range 0 to 255.

Indirection

When referring to data in RAM, square brackets are used. For example [15] refers to the data at address 15hex in RAM.

The same applies to registers. [BL] refers to the data in RAM at the address held in BL. This is important and frequently causes confusion.

These are indirect references. Instead of using the number or the value in the register directly, these values refer to RAM locations. These are also called pointers.

Comparing with 80x86 Chips

At the mnemonic level, the simulator instructions look very like 80x86 assembly code mnemonics. Sufficient instructions are implemented to permit realistic programming but the full instruction set has not been implemented. All the simulated instructions apply to the low eight bits of the 80x86 CPU. The rest of the CPU has not been simulated.

In the registered version, CALL, RET, INT, IRET and simulated hardware interrupts are available so procedures and interrupts can be written.

Most of the instructions behave as an 80x86 programmer would expect. The MUL and DIV (multiplication and division) commands are simpler than the 80x86 equivalents. The disadvantage of the simulator approach is that overflow is much more probable. The simulator versions of ADD and SUB are realistic.

The 8086 DIV instruction calculates both DIV and MOD in the same instruction. The simulator has MOD as a separate instruction.

The machine codes are quite unlike the 80x86 machine codes. They are simpler, less compact but designed to make the machine code as simple as possible.

With 80x86 machine code, a mnemonic like MOV AL,15 is encoded in two bytes. MOV AL, is encoded into one byte and the 15 goes into another. This means that a lot of different machine OP CODES are needed for all the different combinations of MOV commands and registers.

This simulator needs three bytes. MOV is encoded as a byte sized OP CODE. AL is encoded as a byte containing 00. The 15 goes into a byte as before. This is not very efficient but is very simple.



Arithmetic and Logic

Detailed Instruction Set

Arithmetic Instructions - Flags are set.

The Commands

Arithmetic	Logic	Bitwise
Add - Addition	AND - Logical AND - 1 AND 1 gives 1. Any other input gives 0.	ROL - Rotate bits left. Bit at left end moved to right end.
Sub - Subtraction	OR - Logical OR - 0 OR 0 gives 0. Any other input gives 1.	ROR - Rotate bits right. Bit at right end moved to left end.
Mul - Multiplication	XOR - Logical exclusive OR - Equal inputs give 0. Non equal inputs give 1.	SHL - Shift bits left and discard leftmost bit.
Div - Division	NOT - Logical NOT - Invert the input. 0 gives 1. 1 gives 0.	SHR - Shift bits right and discard rightmost bit.
Mod - Remainder after division		
Inc - Increment (add one)		
Dec - Decrement (subtract one)		

COMMANDS	DIRECT EXAMPLES		
OP	Assembler	Machine Code	Explanation
ADD	ADD AL,BL	A0 00 01	Add BL to AL
SUB	SUB CL,DL	A1 02 03	Subtract DL from CL
MUL	MUL AL,CL	A2 00 02	Multiply AL by CL
DIV	DIV BL,DL	A3 01 03	Divide BL by DL
MOD	MOD DL,BL	A6 03 01	Remainder after dividing DL by BL
INC	INC AL	A4 00	Add one to AL
DEC	DEC BL	A5 01	Deduct one from BL
AND	AND CL,AL	AA 02 00	CL becomes CL AND AL
OR	OR CL,DL	AB 02 03	CL becomes CL OR DL
XOR	XOR BL,AL	AC 01 00	BL becomes BL XOR AL
NOT	NOT CL	AD 02	Invert the bits in CL
ROL	ROL DL	9A 03	Bits in DL rotated one place left
ROR	ROR AL	9B 00	Bits in AL rotated one place right
SHL	SHL BL	9C 01	Bits in BL shifted one place left
SHR	SHR CL	9D 02	Bits in CL shifted one place right

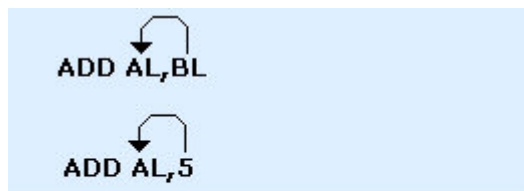
COMMANDS	IMMEDIATE EXAMPLES		
OP	Assembler	Machine Code	Explanation
ADD	ADD AL,15	B0 00 15	Add 15 to AL
SUB	SUB BL,05	B1 01 05	Subtract 5 from BL
MUL	MUL AL,10	B2 00 10	Multiply AL by 10
DIV	DIV BL,04	B3 01 04	Divide BL by 4
MOD	MOD DL,20	B6 03 20	Remainder after dividing DL by 20
AND	AND CL,55	BA 02 55	CL becomes CL AND 55 (01010101)
OR	OR CL,AA	BB 02 AA	CL becomes CL OR AA (10101010)
XOR	XOR BL,F0	BC 01 F0	BL becomes BL XOR F0

Examples

ADD CL,AL - Add CL to AL and put the answer into CL.

ADD AL,22 - Add 22 to AL and put the answer into AL.

The answer always goes into the first register in the command.



Data always moves from right to left as shown by the arrows

DEC BL - Subtract one from BL and put the answer into BL.

The other commands all work in the same way.

Flags

If a calculation gives a zero answer, set the Z zero flag.

If a calculation gives a negative answer, set the S sign flag.

If a calculation overflows, set the O overflow flag.

An overflow happens if the result of a calculation has more bits than will fit into the available register. With 8 bit registers, the largest numbers that fit are -128 to + 127.



Jump Instructions

Detailed Instruction Set

Jump Instructions - Flags are NOT set.

These instructions do NOT set the Z, S or O flags but conditional jumps use the flags to determine whether or not to jump.

The CPU contains a **status register - SR**. This contains flags that are set or cleared depending on the most recent calculation performed by the processor. The **CMP** compare instruction performs a subtraction like the **SUB** command. It sets the flags but the result is not stored.

The Flags - ISOZ

1. **ZERO** - The Z flag is set if the most recent calculation gave a zero result.
2. **SIGN** - The S flag is set if the most recent calculation gave a negative result.
3. **OVERFLOW** - The O flag is set if the most recent calculation gave a result too big to fit a register.
4. **INTERRUPT** - The I flag is set in software using the **STI** command. If this flag is set, the CPU will respond to hardware interrupts. The **CLI** command clears the I flag and hardware interrupts are ignored. The I flag is off by default.

The programmer enters a command like **JMP HERE**. The assembler converts this into machine code by calculating how far to jump. This tedious and error prone task (for humans) is automated. In an 8 bit register, the largest numbers that can be stored are -128 and +127. This limits the maximum distance a jump can go. Negative numbers cause the processor to jump backwards towards zero. Positive numbers cause the processor to jump forward towards 255. The jump distance is added to **IP**, the instruction pointer.

To understand jumps properly, you also need to understand **negative numbers**.

COMMANDS		EXAMPLES	
OP	Assembler	Machine Code	Explanation
JMP	JMP HERE	C0 25	Unconditional jump. Flags are ignored. Jump forward 25h RAM locations.
JMP	JMP BACK	C0 FE	Jump Unconditional jump. Flags are ignored. Jump back -2d RAM locations.
JZ	JZ STOP	C1 42	Jump Zero. Jump if the zero flag (Z) is set. Jump forward +42h places if the (Z) flag is set.
JZ	JZ START	C1 F2	Jump Zero. Jump if the zero flag (Z) is set. Jump back -14d places if the (Z) flag is set.
JNZ	JNZ FORWARD	C2 22	Jump Not Zero. Jump if the zero flag (Z) is NOT set. Jump forward 22h places if the (Z) flag is NOT set.
JNZ	JNZ REP	C2 EE	Jump Not Zero. Jump if the zero flag (Z) is NOT set. Jump back -18d places if the (Z) flag is NOT set.
JS	JS Minus	C3 14	Jump Sign. Jump if the sign flag (S) is set. Jump forward 14h places if the sign flag (S) is set.
JS	JS Minus2	C3 FC	Jump Sign. Jump if the sign flag (S) is set. Jump back -4d places if the sign flag (S) is set.
JNS	JNS Plus	C4 33	Jump Not Sign. Jump if the sign flag (S) is NOT set. Jump forward 33h places if the sign flag (S) is NOT

			set.
JNS	JNS Plus2	C4 E2	Jump Not Sign. Jump if the sign flag (S) is NOT set. Jump back -30d places if the sign flag (S) is NOT set.
JO	JO TooBig	C5 12	Jump Overflow. Jump if the overflow flag (O) is set. Jump forward 12h places if the overflow flag (O) is set.
JO	JO ReDo	C5 DF	Jump Overflow. Jump if the overflow flag (O) is set. Jump back -33d places if the overflow flag (O) is set.
JNO	JNO OK	C6 33	Jump Not Overflow. Jump if the overflow flag (O) is NOT set. Jump forward 33h places if the overflow flag (O) is NOT set.
JNO	JNO Back	C6 E0	Jump Not Overflow. Jump if the overflow flag (O) is NOT set. Jump back -32d places if the overflow flag (O) is NOT set.

The full 8086 instruction set has many other jumps. There are more flags in the 8086 as well!

Legal Destination Labels

here:	A nice correct label.
here::	Not allowed Only one colon is permitted.
1234:	Not allowed. Labels must begin with a letter or '_'.
_:	OK but not human friendly.
here	Destination labels must end in a colon.

Some of these rules are not strictly enforced in the simulator.



Move Instructions

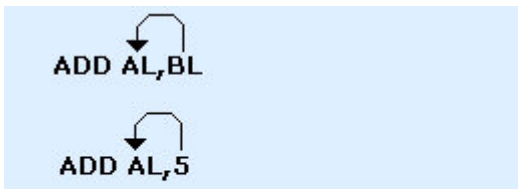
Detailed Instruction Set

Move Instructions - Flags are NOT set.

Move instructions are used to copy data between registers and between RAM and registers.

Addressing Mode	Assembler Example	Supported	Explanation
Immediate	mov al,10	YES	Copy 10 into AL
Direct (register)	mov al,bl	NO	Copy BL into AL
Direct (memory)	mov al,[50]	YES	Copy data from RAM at address 50 into AL.
	mov [40],cl	YES	Copy data from CL into RAM at address 40.
Indirect	mov al,[bl]	YES	BL is a pointer to a RAM location. Copy data from that RAM location into AL.
	mov [cl],dl	YES	CL is a pointer to a RAM location. Copy data from DL into that RAM location.
Indexed	mov al,[20 + bl]	NO	A data table is held in RAM at address 20. BL indexes a data item within the data table. Copy from the data table at address 20+BL into AL.
	mov [20 + bl],al	NO	A data table is held in RAM at address 20. BL indexes a data item within the data table. Copy from AL into the data table at address 20+BL.
Base Register	mov al,[bl+si]	NO	BL points to a data table in memory. SI indexes to a record inside the data table. BL is called the "base register". SI is called the "offset or index". Copy from RAM at address BL+SI into AL.
	mov [bl+si],al	NO	BL points to a data table in memory. SI indexes to a record inside the data table. BL is called the "base register". SI is called the "offset". Copy from AL into RAM at address BL+SI.

Right to Left Convention



Data always moves from right to left as shown by the arrows

ADDRESSING MODES

Immediate

MOV AL,10

Copy a number into a register. This is the simplest move command and easy to understand.

Direct (register)

MOV AL,BL

Copy one register into another. This is easy to understand. The simulator does not support this command. If you have to copy from one register to another, use a RAM location or the stack to achieve the move.

Direct (memory)

MOV AL,[50] ; Copy from RAM into AL. Copy the data from address 50.
MOV [50],AL ; Copy from AL into RAM. Copy the data to address 50.

The square brackets indicate data in RAM. The number in the square brackets indicates the RAM address/location of the data.

Indirect

MOV AL,[BL] ; Copy from RAM into AL. Copy from the address that BL points to.
MOV [BL],AL ; Copy from AL into RAM. Copy to the address that BL points to.

Copy between a specified RAM location and a register. The square brackets indicate data in RAM. In this example BL points to RAM.

Indexed

MOV AL,[20 + BL] ; Copy from RAM into AL. The RAM address is located at 20+BL.
MOV [20 + BL],AL ; Copy from AL into RAM. The RAM address is located at 20+BL.

Here the BL register is used to "index" data held in a table. The table data starts at address 20.

Base Register

MOV AL,[BL+SI] ; Copy from RAM into AL. The RAM address is located at BL+SI.
MOV [BL+SI],AL ; Copy from AL into RAM. The RAM address is located at BL+SI.

BL is the "base register". It holds the start address of a data table. SI is the "source index". It is used to index a record in the data table.



Compare Instructions

Detailed Instruction Set

The Compare CMP Command - Flags are Set.

When the simulator does a comparison using CMP, it does a subtraction of the two values it is comparing. The status register flags are set depending on the result of the subtraction. The flags are set but the answer is discarded.

- (Z) If the values are equal, the subtraction gives a zero result and the (Z) zero flag is set.
- (S) If the number being subtracted was greater than the other than a negative answer results so the (S) sign flag is set.
- If the number being subtracted is smaller than the other, no flags are set.

Use JZ and JS or JNZ and JNS to test the result of a CMP command.

Direct Memory Comparison

Assembler	Machine Code	Explanation
CMP CL,[20]	DC 02 20	Here the CL register is compared with RAM location 20. Work out CL - RAM[20]. DC is the machine instruction for direct memory comparison. 02 refers to the AL register. 20 points to RAM address 20.

Direct Register Comparison

Assembler	Machine Code	Explanation
CMP AL,BL	DA 00 01	Here two registers are compared. Work out AL - BL DA is the machine instruction for register comparison. 00 refers to the AL register. 01 refers to the BL register.

Immediate Comparison

Assembler	Machine Code	Explanation
CMP AL,0D	DB 00 0D	Here the AL register is compared with 0D, (the ASCII code of the Enter key). Work out AL - 0D. DB is the machine instruction for register comparison. 00 refers to the AL register. 0D is the ASCII code of the Enter key.



Stack Instructions

Detailed Instruction Set

Stack Instructions - Flags are NOT set.

After pushing items onto the stack, always pop them off in reverse order. This is because the stack works by the **Last In First Out (LIFO)** rule. The stack is an area of RAM used in this particular way. Any part of RAM could be used. In the simulator, the stack is located just below the Video RAM at address [BF]. The stack grows towards zero. It is easily possible to implement a stack that grows the other way.

Stack Examples

Assembler	Machine Code	Explanation
PUSH BL	E0 01	Push BL onto the stack and subtract one from the stack pointer. E0 is the machine instruction for PUSH. 01 refers to the BL register.
POP BL	E1 01	Add one to the stack pointer and pop BL from the stack. E1 is the machine instruction for POP. 01 refers to the BL register.
PUSHF	EA	Save the CPU status register (SR) onto the stack. This saves the CPU flags.
POPF	EB	Restore the CPU status register (SR) from the stack. This restores the CPU flags.

The stack is used to ...

- save register contents for later restoration.
- pass parameters into procedures and return results.
- reverse the order in which data is stored.
- save addresses so procedures and interrupts can return to the right place.
- perform postfix arithmetic.
- make recursion possible.

Stack Pointer

A CPU register (SP) that keeps track of (is a pointer to) the data on the stack. It is colour coded with a blue highlight in the simulator RAM display.

Push and Pop

Push - Add data to the stack at the stack pointer position and subtract one from the stack pointer.

Pop - Add one to the stack pointer and remove data from the stack at the stack pointer position.

LIFO

Last in First out. The stack operates strictly to this rule. When data is pushed onto the stack, it must later be popped in reverse order.

Stack Overflow

The stack is repeatedly pushed until it is full. The simulator does not detect this condition and the stack can overwrite program code or data. Real life programs can fail in the same way.

Stack Underflow

The stack is repeatedly popped until it is empty. The next pop causes an underflow.



Procedures and Interrupts

Detailed Instruction Set

Procedures and Interrupts - Flags are NOT set.

These are available in the registered version. Please register.

It is essential to save the registers and flags used by any procedure or interrupt and restore them after the procedure or interrupt has finished its work. Use **push** and **pushf** to save. Use **pop** and **popf** to restore values.

Assembler	Machine Code	Explanation
CALL 30	CA 30	Call the procedure at address 30. The return address is pushed onto the stack and the Instruction Pointer (IP) is set to 30. CA is the machine instruction for CALL. 30 is the address of the start of the procedure being called.
RET	CB	Return from the procedure. Set the Instruction Pointer (IP) to the return address popped off the stack. CB is the machine instruction for Return.
INT 03	CC 03	The Instruction Pointer (IP) is set to the address of the interrupt vector retrieved from RAM address 03. The return address is pushed onto the stack. CC is the machine instruction for INT. 03 is the address of the interrupt vector used by the INT command.
IRET	CD	Return from the interrupt. Set the Instruction Pointer (IP) to the return address popped off the stack. CD is the machine instruction for IRET.



Input Output Instructions

Detailed Instruction Set

Input and Output Instructions - Flags are NOT set.

The simulator has 16 ports numbered from 00 to 0F. These are connected to simulated, outside-world peripherals.

Assembler	Machine Code	Explanation
IN 07	F0 07	Input from Port 07. F0 is the machine instruction for Input. 07 is the port number.
OUT 01	F1 01	Output to Port 01. F1 is the machine instruction for Output. 01 is the port number.

Peripherals

Port	Description
00	Input from port 00 for simulated keyboard input.
01	Output to port 01 to control the traffic lights.
02	Output to port 02 to control the seven segment displays.
03	Output to port 03 to control the heater. Input from port 03 to sense the thermostat state.
04	Output to port 04 to control the snake in the maze.
05	Output to port 05 to control the stepper motor.
06	Output to port 06 to control the lift.
07	Output to port 07 to make the keyboard visible. Input from port 07 to read the keyboard ASCII code.
08	Output to port 08 to make the numeric keypad visible. Input from port 08 to read from the numeric keypad.
09-0F	Unused



Other Instructions

Detailed Instruction Set

Miscellaneous Instructions - CLI and STI control the (I) Flag

Assembler	Machine Code	Explanation
HALT	00	Stop the program. 00 is the machine instruction for HALT. The program will cease to run if it encounters a HALT instruction. Continuous running is cancelled by this command. You can have several halt commands in one program. There should be only one END and code after END is ignored.
NOP	FF	Do nothing for one clock cycle. FF is the machine instruction for NOP. The program will do nothing for one clock cycle. The program then continues as normal. NOP is used to introduce time delays to allow slow electronics to keep up with the CPU. These are also called WAIT STATES.
CLO	FE	Close all the peripheral windows. FE is the machine code for CLO. It applies to this simulator only, and is used to close peripheral windows. This makes it easier to write demonstration programs without the screen getting too cluttered.
ORG 30	NONE	Code Origin. Generate code starting from this address. To generate code from a starting address other than zero use ORG. This is useful to place procedures, interrupts or data tables at particular addresses in memory. ORG is an assembler directive and no code is generated.
DB 84	84	Define a byte. Store the byte (84) in the next free RAM location. Use DB to create data tables containing bytes of data. Use BD to define an Interrupt Vector.
DB "Hello"	48, 65, 6C, 6C, 6F	Define a string. Store the ASCII codes of the text in quotes in the next free RAM locations. Use DB to store text strings. The stored ASCII codes do not include an end-of-string marker. Use DB 00 for this.
CLI	FD	Clear the I flag If the I flag is cleared, hardware interrupts are ignored. This is the default state for the simulator. Resetting the CPU will also clear the I flag. The timer that generates hardware interrupts will do nothing.
STI	FC	Set the I flag If the I flag is set, the simulator will generate INT 02 at regular time intervals. It is necessary to have an interrupt vector stored at address 02 that points to interrupt handler code stored elsewhere. The interval between timer interrupts can be set using the slider in the Configuration Tab. If interrupts occur faster than the processor can handle them, a simulated system crash will follow. Adjust the CPU clock speed and the timer interval to prevent this – or cause it if you want to see what happens.



List File

Contents

The List File

D:\BackMeUp\UDISKPRO\Delphi\Simulator\Sms32v50\code\DEMO.ASM

File Edit View Examples Help

AL 00000000 00 +000	IP 00000000 00 +000	File	A	Monitor	Run	Assemble	Slower	Continue
BL 00000000 00 +000	SP 10111111 BF -065	Save	A	HI	Reset	Step	Faster	Cpu Reset
CL 00000000 00 +000	SR 00000000 00 +000	Print	B	Up	K	Run F9	STOP	Show Ram
DL 00000000 00 +000	IS02							

Saved DEMO.ASM Make tokens. Parse tokens. Calculate jumps. Success.

☐ Write Run Log ☐ Log Assembler Activity

Source Code **List File** Configuration Tokens Run Log

```
HERE:
      CLO          ; [0F] FE          ; Close all peripheral windows
      MOV CL,C0    ; [10] D0 02 C0      ; Video ram base address
      MOV BL,02    ; [13] D0 01 02      ; Offset of text string
      ;
START:
      ;
      MOV AL,[BL]  ; [16] D3 00 01      ; Text pointer into AL
      CMP AL,0     ; [19] DB 00 00      ; At end yet
      JZ END1      ; [1C] C1 0B        ; Jump out of loop
      MOV [CL],AL  ; [1E] D4 02 00      ; AL into video memory
      INC CL       ; [21] A4 02        ; Next video location
      INC BL       ; [23] A4 01        ; Next text character
      JMP START    ; [25] C0 F1        ; Not there yet
END1:
```

Labels	Operators	Operands	RAM Addresses	Operators	One Address Operand	Two Address Operand	Comment
--------	-----------	----------	---------------	-----------	---------------------	---------------------	---------

In the list file, your original program is shown.

Numbers in square brackets such as [1C] are the addresses at which the machine codes were generated.

The machine codes are shown.

Here is a typical line.

```
MOV CL,C0          ; [10] D0 02 C0          ; Video ram base address
```

The command is to move C0 into the AL register.

The machine code was generated at address [10].

The machine codes are D0 00 C0.

The programmer's comment is reproduced.



Negative Numbers

Contents

Negative Numbers															
Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
-128	80	-127	81	-126	82	-125	83	-124	84	-123	85	-122	86	-121	87
-120	88	-119	89	-118	8A	-117	8B	-116	8C	-115	8D	-114	8E	-113	8F
-112	90	-111	91	-110	92	-109	93	-108	94	-107	95	-106	96	-105	97
-104	98	-103	99	-102	9A	-101	9B	-100	9C	-099	9D	-098	9E	-097	9F
-096	A0	-095	A1	-094	A2	-093	A3	-092	A4	-091	A5	-090	A6	-089	A7
-088	A8	-087	A9	-086	AA	-085	AB	-084	AC	-083	AD	-082	AE	-081	AF
-080	B0	-079	B1	-078	B2	-077	B3	-076	B4	-075	B5	-074	B6	-073	B7
-072	B8	-071	B9	-070	BA	-069	BB	-068	BC	-067	BD	-066	BE	-065	BF
-064	C0	-063	C1	-062	C2	-061	C3	-060	C4	-059	C5	-058	C6	-057	C7
-056	C8	-055	C9	-054	CA	-053	CB	-052	CC	-051	CD	-050	CE	-049	CF
-048	D0	-047	D1	-046	D2	-045	D3	-044	D4	-043	D5	-042	D6	-041	D7
-040	D8	-039	D9	-038	DA	-037	DB	-036	DC	-035	DD	-034	DE	-033	DF
-032	E0	-031	E1	-030	E2	-029	E3	-028	E4	-027	E5	-026	E6	-025	E7
-024	E8	-023	E9	-022	EA	-021	EB	-020	EC	-019	ED	-018	EE	-017	EF
-016	F0	-015	F1	-014	F2	-013	F3	-012	F4	-011	F5	-010	F6	-009	F7
-008	F8	-007	F9	-006	FA	-005	FB	-004	FC	-003	FD	-002	FE	-001	FF
Positive Numbers															
Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
+000	00	+001	01	+002	02	+003	03	+004	04	+005	05	+006	06	+007	07
+008	08	+009	09	+010	0A	+011	0B	+012	0C	+013	0D	+014	0E	+015	0F
+016	10	+017	11	+018	12	+019	13	+020	14	+021	15	+022	16	+023	17
+024	18	+025	18	+026	1A	+027	1B	+028	1C	+029	1D	+030	1E	+031	1F
+032	20	+033	21	+034	22	+035	23	+036	24	+037	25	+038	26	+039	27
+040	28	+041	29	+042	2A	+043	2B	+044	2C	+045	2D	+046	2E	+047	2F
+048	30	+049	31	+050	32	+051	33	+052	34	+053	35	+054	36	+055	37
+056	38	+057	39	+058	3A	+059	3B	+060	3C	+061	3D	+062	3E	+063	3F
+064	40	+065	41	+066	42	+067	43	+068	44	+069	45	+070	46	+071	47
+072	48	+073	49	+074	4A	+075	4B	+076	4C	+077	4D	+078	4E	+079	4F
+080	50	+081	51	+082	52	+083	53	+084	54	+085	55	+086	56	+087	57
+088	58	+089	59	+090	5A	+091	5B	+092	5C	+093	5D	+094	5E	+095	5F
+096	60	+097	61	+098	63	+099	63	+100	64	+101	65	+102	66	+103	67
+104	68	+105	69	+106	6A	+107	6B	+108	6C	+109	6D	+110	6E	+111	6F
+112	70	+113	71	+114	72	+115	73	+116	74	+117	75	+118	76	+119	77
+120	78	+121	79	+122	7A	+123	7B	+124	7C	+125	7D	+126	7E	+127	7F



Pop-up Help

Contents

ADD AND CALL CLI CLO CMP DB DEC DIV END HALT IN INC INT

IRET JMP JNO JNS JNZ JO JS JZ MOD MOV MUL NOP NOT OR ORG

OUT POP POPF PUSH PUSHF RET ROL ROR SHL SHR STI SUB XOR

CPU General Purpose Registers

The CPU is where all the arithmetic and logic (decision making) takes place. The CPU has storage locations called registers. The CPU has flags which indicate zero, negative or overflowed calculations. More information is included in the description of the system architecture.

The CPU registers are called AL, BL, CL and DL.
The machine code names are 00, 01, 02 and 03.

Registers are used for storing binary numbers.

Once the numbers are in the registers, it is possible to perform arithmetic or logic. Sending the correct binary patterns to peripherals like the traffic lights, makes it possible to control them.

; semicolon begins a program comment.

Comments are used to document programs. They are helpful to new programmers joining a team and to existing people returning to a project having forgotten what it is about.

Good comments explain **WHY** things are being done. Poor comments simply repeat the code or state the totally obvious.

Ram Addresses

Examples [7F] [22] [AL] [CL]

[7F] the contents of RAM at location 7F

[CL] the contents of the RAM location that CL points to. CL contains a number that is used as the address.

The Instruction Set

Pop-up Help

ADD - Add two values together

CPU flags are set

Assembler	Machine Code	Explanation
ADD BL,CL	A0 01 02	Add CL to BL. Answer goes into BL
ADD AL,12	B0 00 12	Add 12 to AL. Answer goes into AL

Pop-up Help

AND - Logical AND two values together

CPU flags are set

Assembler	Machine Code	Explanation
AND BL,CL	AA 01 02	AND CL with BL. Answer goes into BL
AND AL,12	BA 00 12	AND 12 with AL. Answer goes into AL

The AND rule is that two ones give a one. All other inputs give nought. Look at this example...

```
      10101010
      00001111
      -----
ANSWER 00001010
```

The left four bits are masked to 0.

Pop-up Help

CALL and RET

CPU flags are NOT set

Assembler	Machine Code	Explanation
CALL 50	CA 50	Call the procedure at address 50. The CPU pushes the instruction pointer value IP + 2 onto the stack. Later the CPU returns to this address. IP is then set to 50.
RET	CB	The CPU instruction pointer is set to 50. The CPU executes instructions from this address until it reaches the RET command. It then pops the value of IP off the stack and jumps to this address where execution resumes.

Pop-up Help

CLI and STI

CPU (I) flag is set/cleared

Assembler	Machine Code	Explanation
STI	FC	STI sets the Interrupt flag.
CLI	FD	CLI clears the Interrupt flag 'I' in the status register. STI sets the interrupt flag 'I' in the status register. The machine code for CLI is FD. The machine code for STI is FC. If (I) is set, the CPU will respond to interrupts. The simulator generates a hardware interrupt at regular time intervals that you can adjust. If 'I' is set, there should be an interrupt vector at address [02]. The CPU will jump to the code that this vector points to whenever there is an interrupt.

Pop-up Help

CLO

CPU flags are NOT set

Assembler	Machine Code	Explanation
CLO	FE	Close unwanted peripheral windows.

		CLO is not an x86 command. It closes all unnecessary simulator windows which would otherwise have to be closed manually one by one.
--	--	---

Pop-up Help

CMP

CPU flags are set

Assembler	Machine Code	Explanation
CMP AL,0D	DB 00 0D	Compare AL with 0D If the values being compared are ... EQUAL set the 'Z' flag. AL less than 0D set the 'S' flag. AL greater than 0D set no flags.
CMP AL,BL	DA 00 01	Compare AL with BL If the values being compared are ... EQUAL set the 'Z' flag. AL less than BL set the 'S' flag. AL greater than BL set no flags.
CMP CL,[20]	DC 02 20	Compare CL with 20 If the values being compared are ... EQUAL set the 'Z' flag. CL less than RAM[20] set the 'S' flag. CL greater than RAM[20] set no flags.

Pop-up Help

DB

CPU flags are NOT set

Assembler	Machine Code	Explanation
DB 22 DB 33 DB 44 DB 0	22 33 44 00	Define Byte DB gives a method for loading values directly into RAM. DB does not have a machine code. The numbers or text after DB are loaded into RAM. Use DB to set up data tables.
DB "Hello" DB 0	48 65 6C 6C 6F 00	ASCII codes are loaded into RAM. End of text is marked by NULL

Pop-up Help

DEC and INC

CPU flags are set

Assembler	Machine Code	Explanation
INC BL	A4 01	Add one to BL.
DEC AL	A5 00	Subtract one from AL.

Pop-up Help

DIV and MOD

CPU flags are set

Assembler	Machine Code	Explanation
DIV AL,5	B3 00 05	Divide AL by 5. Answer goes into AL. DIV differs from the x86 DIV.
DIV AL,BL	A3 00 01	Divide AL by BL. Answer goes into AL. DIV differs from the x86 DIV.
MOD AL,5	B6 00 05	MOD AL by 5. Remainder after division goes into AL. MOD is not an x86 command.
MOD AL,BL	A6 00 01	MOD AL by BL. Remainder after division goes into AL. MOD is not an x86 command.

The x86 DIV calculates div and mod in one command. The answers are put into different registers. This is not possible with the 8 bit simulator so div and mod are separated and simplified.

8 DIV 3 is 3 (with remainder 2). 8 MOD 3 is 2

Pop-up Help

END

CPU flags are NOT set

Assembler	Machine Code	Explanation
END	00	END stops further program execution. The simulator achieves this by stopping the CPU clock. END is also an assembler directive. All code after END is ignored by the assembler. There should be only one END in a program.

Pop-up Help

HALT

CPU flags are NOT set

Assembler	Machine Code	Explanation
HALT	00	HALT stops further program execution. The simulator achieves this by stopping the CPU clock. HALT is not an assembler directive. (See END) There can be any number of HALT commands in a program.

Pop-up Help

IN and OUT

CPU flags are NOT set

Assembler	Machine Code	Explanation
IN 07	F0 07	Input from port 07. The data is stored in the AL register.
OUT 03	F1 03	Output to port 03. The data comes from the AL register.

Pop-up Help

INC and DEC

CPU flags are set

Assembler	Machine Code	Explanation
INC BL	A4 01	Add one to BL.
DEC AL	A5 00	Subtract one from AL.

Pop-up Help

INT and IRET

CPU flags are NOT set

Assembler	Machine Code	Explanation
INT 02	CC 02	The return address (IP + 2) is pushed onto the stack. The stack pointer (SP) is reduced by one. RAM location 02 contains the address of the Interrupt Handler. This address is "fetched" and IP is set to it.
IRET	CD	The return address is popped off the stack. The stack pointer (SP) is increased by one. IP is set to the return address popped off the stack.

Pop-up Help

JMP

CPU flags are NOT set and the flags are ignored

Assembler	Machine Code	Explanation
JMP Forward	C0 12	Set IP to a new value Add 12 to IP The assembler calculates the jump distance. The biggest possible forward jump is +127.
JMP Back	FE	Set IP to a new value Add -2 to IP FE is -2. This is explained here. The assembler calculates the jump distance. The biggest possible backward jump is -128.

Pop-up Help

JNO

CPU flags are NOT set. JNO uses the (O) flag.

The (O) flag is set if a calculation gives a result too big to fit in an 8 bit register.

Assembler	Machine Code	Explanation
JNO Forward	C6 12	Jump if the (O) flag is NOT set. If the (O) flag is NOT set, jump forward 12 places. If the (O) flag is NOT set, add 12 to (IP). If the (O) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.
JNO Back	C6 FE	Jump if the (O) flag is NOT set. If the (O) flag is NOT set, jump back 2 places. If the (O) flag is NOT set, add -2 to (IP).

		<p>If the (O) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>
--	--	---

Pop-up Help

JNS

CPU flags are NOT set. JNS uses the (S) flag.

The (S) flag is set if a calculation gives a negative result.

Assembler	Machine Code	Explanation
JNS Forward	C4 12	<p>Jump if the (S) flag is NOT set. If the (S) flag is NOT set, jump forward 12 places. If the (S) flag is NOT set, add 12 to (IP). If the (S) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.</p>
JNS Back	C4 FE	<p>Jump if the (S) flag is NOT set. If the (S) flag is NOT set, jump back 2 places. If the (S) flag is NOT set, add -2 to (IP). If the (S) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>

Pop-up Help

JNZ

CPU flags are NOT set. JNZ uses the (Z) flag.

The (Z) flag is set if a calculation gives a zero result.

Assembler	Machine Code	Explanation
JNZ Forward	C2 12	<p>Jump if the (Z) flag is NOT set. If the (Z) flag is NOT set, jump forward 12 places. If the (Z) flag is NOT set, add 12 to (IP). If the (Z) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.</p>
JNZ Back	C2 FE	<p>Jump if the (Z) flag is NOT set. If the (Z) flag is NOT set, jump back 2 places. If the (Z) flag is NOT set, add -2 to (IP). If the (Z) flag is set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>

Pop-up Help

JO

CPU flags are NOT set. JO uses the (O) flag.

The (O) flag is set if a calculation gives a result too big to fit in an 8 bit register.

Assembler	Machine Code	Explanation
JO Forward	C5 12	Jump if the (O) flag is set.

		<p>If the (O) flag is set, jump forward 12 places. If the (O) flag is set, add 12 to (IP). If the (O) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.</p>
JO Back	C5 FE	<p>Jump if the (O) flag is set. If the (O) flag is set, jump back 2 places. If the (O) flag is set, add -2 to (IP). If the (O) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>

Pop-up Help

JS

CPU flags are NOT set. JS uses the (S) flag.

The (S) flag is set if a calculation gives a negative result.

Assembler	Machine Code	Explanation
JS Forward	C3 12	<p>Jump if the (S) flag is set. If the (S) flag is set, jump forward 12 places. If the (S) flag is set, add 12 to (IP). If the (S) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.</p>
JS Back	C3 FE	<p>Jump if the (S) flag is set. If the (S) flag is set, jump back 2 places. If the (S) flag is set, add -2 to (IP). If the (S) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>

Pop-up Help

JZ

CPU flags are NOT set. JZ uses the (Z) flag.

The (Z) flag is set if a calculation gives a zero result.

Assembler	Machine Code	Explanation
JZ Forward	C1 12	<p>Jump if the (Z) flag is set. If the (Z) flag is set, jump forward 12 places. If the (Z) flag is set, add 12 to (IP). If the (Z) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible forward jump is +127.</p>
JZ Back	C1 FE	<p>Jump if the (Z) flag is set. If the (Z) flag is set, jump back 2 places. If the (Z) flag is set, add -2 to (IP). If the (Z) flag is NOT set, add 2 to (IP). The assembler calculates the jump distance. The biggest possible backward jump is -128. FE is -2. This is explained here.</p>

Pop-up Help

DIV and MOD

CPU Flags are Set

Assembler	Machine Code	Explanation
DIV AL,5	B3 00 05	Divide AL by 5. Answer goes into AL. DIV differs from the x86 DIV.
DIV AL,BL	A3 00 01	Divide AL by BL. Answer goes into AL. DIV differs from the x86 DIV.
MOD AL,5	B6 00 05	MOD AL by 5. Remainder after division goes into AL. MOD is not an x86 command.
MOD AL,BL	A6 00 01	MOD AL by BL. Remainder after division goes into AL. MOD is not an x86 command.

The x86 DIV calculates div and mod in one command. The answers are put into different registers. This is not possible with the 8 bit simulator so div and mod are separated and simplified.

8 DIV 3 is 3 (with remainder 2). 8 MOD 3 is 2

MOV**CPU flags are NOT set**

Addressing Mode	Assembler Example Machine Code	Supported	Explanation
Immediate	mov al,10 D0 00 10	YES	Copy 10 into AL
Direct (register)	mov al,bl	NO	Copy BL into AL
Direct (memory)	mov al,[50] D1 00 50	YES	Copy data from RAM at address 50 into AL. [50] is a pointer to data held in a RAM location.
	mov [40],cl D2 40 02	YES	Copy data from CL into RAM at address 40. [40] is a pointer to data held in a RAM location.
Indirect	mov al,[bl] D3 00 01	YES	BL is a pointer to a RAM location. Copy data from that RAM location into AL.
	mov [cl],dl D4 02 03	YES	CL is a pointer to a RAM location. Copy data from DL into that RAM location.
Indexed	mov al,[20 + bl]	NO	A data table is held in RAM at address 20. BL indexes a data item within the data table. Copy from the data table at address 20+BL into AL.
	mov [20 + bl],al	NO	A data table is held in RAM at address 20. BL indexes a data item within the data table. Copy from AL into the data table at address 20+BL.
Base Register	mov al,[bl+si]	NO	BL points to a data table in memory. SI indexes to a record inside the data table. BL is called the "base register". SI is called the "offset or index". Copy from RAM at address BL+SI into AL.
	mov [bl+si],al	NO	BL points to a data table in memory. SI indexes to a record inside the data table. BL is called the "base register". SI is called the "offset". Copy from AL into RAM at address BL+SI.

Pop-up Help

MUL

CPU Flags are Set

Assembler	Machine Code	Explanation
MUL AL,BL	A2 00 01	Multiply AL by BL. The result goes into AL MUL differs from the x86 MUL.
MUL CL,12	B2 02 12	Multiply CL by 12. The result goes into CL MUL differs from the x86 MUL.

The x86 MUL places the result into more than one register. This is not possible with the 8 bit simulator so MUL has been simplified. A disadvantage is that an overflow is much more likely to occur.

Pop-up Help

NOP

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
NOP	FF	Do nothing. Do nothing for one CPU clock cycle. This is needed to keep the CPU synchronised with accurately timed electronic circuits. The CPU might need to delay before the electronics are ready.

Pop-up Help

NOT

CPU Flags are Set

Assembler	Machine Code	Explanation
NOT DL	AD 03	Invert all the bits in DL.

If DL contained 01010101, after using NOT it will contain 10101010.

Pop-up Help

OR

CPU Flags are Set

Assembler	Machine Code	Explanation
OR AL,12	BB 00 12	Or 12 with AL. Answer goes into AL
OR BL,CL	AB 01 02	Or CL with BL. Answer goes into BL

The OR rule is that two noughts give a nought. All other inputs give one.

```
  10101010
OR 00001111
-----
= 10101111
```

Pop-up Help

ORG

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
ORG 50	None	ORG is not a CPU instruction. It is an instruction to the assembler to tell it to generate code at a particular address. It is useful for writing procedures and interrupts. It can also be used to specify where in memory, data tables go.

Pop-up Help

OUT and IN

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
IN 07	F0 07	Input from port 07. The data is stored in the AL register.
OUT 03	F1 03	Output to port 03. The data comes from the AL register.

Pop-up Help

PUSH, POP, PUSHF and POPF

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
PUSH AL	E0 00	Save AL onto the stack. Deduct one from the Stack Pointer (SP)
POP BL	E1 01	Add one to the stack pointer (SP). Restore BL from the stack
PUSHF	EA	Push the CPU flags from the status register (SR) onto the stack. Deduct one from the Stack Pointer (SP)
POPF	EB	Add one to the stack pointer (SP). POP the CPU flags from the stack into the status register (SR).

PUSH saves a byte onto the stack. POP gets it back. The stack is an area of memory that obeys the LIFO rule - Last In First Out. When pushing items onto the stack, remember to pop them off again in exact reverse order. The stack can be used to

1. hold the return address of a procedure call
2. hold the return address of an interrupt call
3. pass parameters into procedures
4. get results back from procedures
5. save and restore registers and flags
6. reverse the order of data.

Pop-up Help

RET and CALL

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
CALL 50	CA 50	Call the procedure at address 50.

		The CPU pushes the instruction pointer value IP + 2 onto the stack. Later the CPU returns to this address. IP is then set to 50.
RET	CB	The CPU instruction pointer is set to 50. The CPU executes instructions from this address until it reaches the RET command. It then pops the value of IP off the stack and jumps to this address where execution resumes.

Pop-up Help

ROL and ROR

CPU Flags are Set

Assembler	Machine Code	Explanation
ROL AL	9A 00	Rotate the bits in AL left one place. The leftmost bit is moved to the right end of the byte. Before ROL 10000110 - After ROL 00001101
ROR DL	9B 03	Rotate the bits in DL right one place. The rightmost bit is moved to the left end of the byte. Before ROR 10000110 - After ROR 01000011

Pop-up Help

SHL and SHR

CPU Flags are Set

Assembler	Machine Code	Explanation
SHL AL	9C 00	Shift bits left one place. The leftmost bit is discarded. Before SHL 10000110 - After SHL 00001100
SHR DL	9D 03	Shift bits right one place. The rightmost bit is discarded. Before SHR 10000110 - After SHR 01000011

Pop-up Help

STI and CLI

CPU Flags are NOT Set

Assembler	Machine Code	Explanation
STI	FC	STI sets the Interrupt flag.
CLI	FD	CLI clears the Interrupt flag 'I' in the status register. STI sets the interrupt flag 'I' in the status register. The machine code for CLI is FD. The machine code for STI is FC. If (I) is set, the CPU will respond to interrupts. The simulator generates a hardware interrupt at regular time intervals that you can adjust. If 'I' is set, there should be an interrupt vector at address [02]. The CPU will jump to the code that this vector points to whenever there is an interrupt.

Pop-up Help

SUB

CPU Flags are Set

Assembler	Machine Code	Explanation
SUB AL,12	B1 00 12	Subtract 12 from AL. The answer goes into AL.
SUB BL,CL	A1 01 02	Subtract CL from BL. The answer goes into BL.

Pop-up Help

XOR

CPU Flags are Set

Assembler	Machine Code	Explanation
XOR AL,12	BC 00 12	12 XOR AL. The answer goes into AL.
XOR BL,CL	AC 01 02	CL XOR BL. The answer goes into BL.

XOR can be used to invert selected bits.

```
    00001111 This is a bit mask.  
XOR 01010101  
-----  
    01011010
```

The left four bits are unaltered. The right four bits are inverted.



Truth Tables and Logic

Contents

Boolean Operators - Flags are Set

A mathematician called Bool invented a branch of maths for processing true and false values instead of numbers. This is called Boolean Algebra. Simple Boolean algebra is consistent with common sense but if you need to process decisions involving many values that might be true or false according to complex rules, you need this branch of mathematics.

The Rules

Rule	One Line Explanation
AND	1 AND 1 gives 1. Any other input gives 0.
NAND	(NOT AND) 1 AND 1 gives 0. Any other input gives 1.
OR	0 OR 0 gives 0. Any other input gives 1.
NOR	(NOT OR) 0 OR 0 gives 1. Any other input gives 0.
XOR	Equal inputs give 0. Non equal inputs give 1.
NOT	Invert input bits. 0 becomes 1. 1 becomes 0.

Computers work using LOGIC. Displaying graphics such as the mouse cursor involves the XOR (Exclusive OR) command. Addition makes use of AND and XOR. These and a few of the other uses of logic are described below.

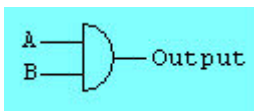
Truth Tables

The one line descriptions of the rules above are clearer if shown in Truth Tables. These tables show the output for all possible input conditions.

Logic Gates

Logic gates are the building blocks of microcomputers. Modern processors contain millions of gates. Each gate is built from a few transistors. The gates are used to store data, perform arithmetic and manipulate bits using the rules above. The XOR rule can be used to test bits for equality.

AND

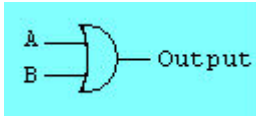


Both inputs must be true for the output to be true. AND is used for addition and decision making.

A	B	Output

0	0	0
0	1	0
1	0	0
1	1	1

OR

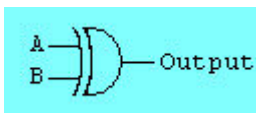


Both inputs must be false for the output to be false. OR is used in decision making. Both AND and OR are used for Bit Masking. Bit masking is used to pick individual bits out of a byte or to set particular bits in a byte. OR is used to set bits to one. AND is used to set bits to nought. AND is used to test if bits are one. OR is used to test if bits are nought.

A	B	Output

0	0	0
0	1	1
1	0	1
1	1	1

XOR

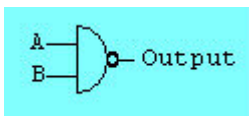


If the bits in a graphical image are XORed with other bits a new image appears. If the XORing is repeated the image disappears again. This is how the mouse and text cursors get moved around the screen. XOR is combined with AND for use in addition. XOR detects if the inputs are equal or not.

A	B	Output

0	0	0
0	1	1
1	0	1
1	1	0

NAND

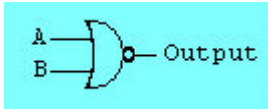


NAND is really AND followed by NOT. Electronic circuits are commonly built from NAND gates (circuits). Computer programming languages and this simulator do not provide NAND. Use NOT AND instead.

A	B	Output

0	0	1
0	1	1
1	0	1
1	1	0

NOR

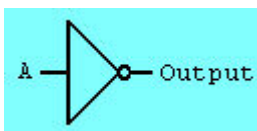


NOR is really OR followed by NOT. Electronic circuits are commonly built from NOR gates (circuits). Computer programming languages and this simulator do not provide NOR. Use NOT OR instead.

A	B	Output

0	0	1
0	1	0
1	0	0
1	1	0

NOT

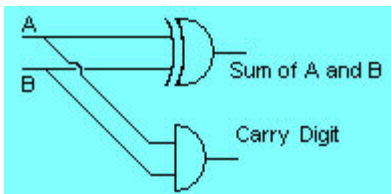


NOT is used to invert bits or True/False values. All the rules above had two inputs and one output. NOT has a single input and output.

A	Output

0	1
1	0

The Half Adder Truth Table



The half adder does binary addition on two bits.
The AND gate computes the carry bit.
The XOR gate computes the sum bit.

0 + 0 = 0, carry 0
0 + 1 = 1, carry 0
1 + 0 = 1, carry 0
1 + 1 = 0, carry 1

A	B	SUM	CARRY

0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Using the Editor

Contents

Using the Editor

Editing the source code in the simulator is similar to most word processors and text editors such as the Windows Notepad.

Undo

You can undo an editing error. When you have an accident and delete or mess up something by mistake, you can press Ctrl+Z to UNDO the last thing you did. This can be very useful.

Cursor Movements

Move the text cursor. For small movements, use the Arrow Keys, Home, End, Page Up and Page Down. You can use the mouse too.

For larger movements, hold down the Ctrl key and use the Arrow Keys, Home, End, Page Up, and Page Down. You can use the mouse too.

Deleting

Delete previous character with the Backspace Key

Delete next character with the Delete Key

Highlighting

To highlight a block of text and hold down the Shift key and use the Arrow Keys, Home, End, Page Up and Page Down. You can drag the mouse with the left button pressed to do this too.

To highlight whole words, lines or documents, hold down Shift and Ctrl and then use the Arrow Keys, Home, End, Page Up and Page Down. Alternatively drag the mouse with the left button pressed.

Key	Explanation
Ctrl+C	Copy a highlighted block
Ctrl+X	Cut a highlighted block
Ctrl+V	Paste text copied or cut earlier
Delete	Delete a highlighted block
Ctrl+S	Save a file
Alt+F a	Save a file with a new name
Ctrl+O	Open a file
Alt+F x	Quit



Virtual Peripherals

Contents

Using the Peripheral Devices

Keyboard

Port 07

INT 03



How to Use

This is one of the more complex devices. To make the keyboard visible, use **OUT 07**. Every time a key is pressed, a hardware interrupt, **INT 03** is generated. By default, the CPU will ignore this interrupt. To process the interrupt, at the start of the program, use the **STI** command to set the interrupt flag (I) in the CPU status register (SR). Place an interrupt vector at RAM address **03**. This should point to your interrupt handler code. The interrupt handler should use **IN 07** to read the key press into the **AL** register.

Once **STI** has set the (I) flag in the status register (SR), interrupts from the hardware timer will also be generated. These must be processed too. The hardware timer generates **INT 02**. To process this interrupt, place an interrupt vector at RAM location **02**. This should point to the timer interrupt handler code. The timer code can be as simple as **IRET**. This will cause an interrupt return without doing any other processing.

```
        jmp     start
        db      10          ; Hardware Timer Interrupt Vector
        db      20          ; Keyboard Interrupt Vector

; ===== Hardware Timer =====
        org     10
        nop
        nop          ; Do something useful here
        nop
        nop
        nop
        nop
        iret

; =====

; ===== Keyboard Handler =====
        org     20
        CLI          ; Prevent re-entrant use
        push    al
        pushf

        in      07
        nop          ; Process the key press here
        nop
```

```

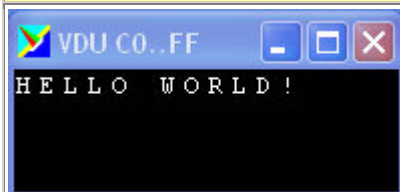
        nop
        nop
        nop

        popf
        pop     al
        STI
        iret
; =====

; ===== Idle Loop =====
start:
        STI             ; Set (I) flag
        out     07      ; Make keyboard visible
idle:
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        jmp     idle
; =====
        end
; =====

```

Visual Display Unit Memory Mapped



How to Use

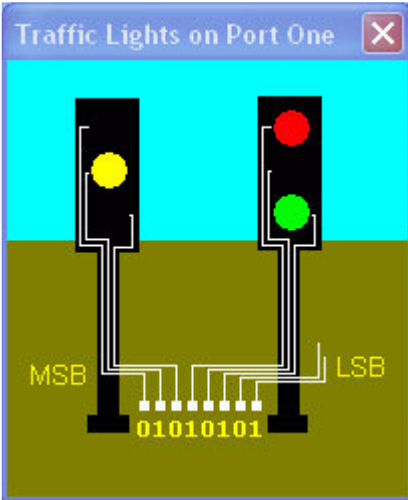
The Visual Display Unit (VDU) is memory mapped. This means that RAM locations correspond to positions on the screen. RAM location C0 maps to the top left corner of the VDU. The screen has 16 columns and four rows mapped to RAM locations C0 to FF. When you write ASCII codes to these RAM locations, the corresponding text characters appear and the VDU is made visible. This device, when combined with a keyboard, is sometimes called a dumb terminal. It has no graphics capabilities. Here is a code snippet to write text to the screen.

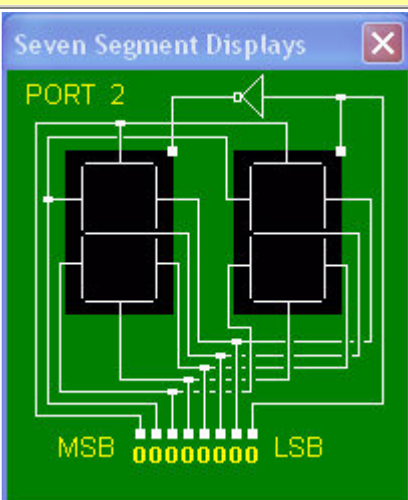
```

; ===== Memory Mapped VDU =====
MOV     AL,41    ; ASCII code of 'A'
MOV     [C0],AL  ; RAM location mapped to the
                  ; top left corner of the VDU
MOV     AL,42    ; ASCII code of 'B'
MOV     [C1],AL  ; RAM location mapped to the VDU
MOV     AL,43    ; ASCII code of 'C'
MOV     [C2],AL  ; RAM location mapped to the VDU

        END
; =====

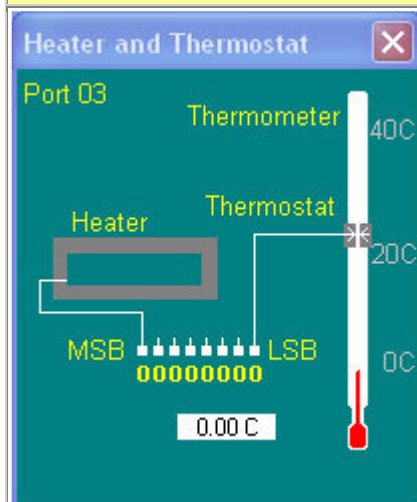
```

Traffic Lights Port 01	How to Use
	<p>The traffic lights are connected to Port 01. If a byte of data is sent to this port, wherever there is a one, the corresponding traffic light comes on. In the image on the left, the binary data is 01010101. If you look closely you can see that the lights that are on, correspond to the ones in the data byte. 01010101 is 55 hexadecimal. Hex' numbers are explained here. Here is a code snippet to control the lights.</p> <pre> ; ===== ; ===== 99Tlight.asm ===== ; ===== Traffic Lighte on Port 01 ===== Start: MOV AL,55 ; 01010101 OUT 01 ; Send the data in AL to Port 01 ; (the traffic lights) MOV AL,AA ; 10101010 OUT 01 ; Send the data in AL to Port 01 ; (the traffic lights) JMP Start END ; ===== </pre>

Seven Segment Displays Port 02	How to Use
	<p>The seven segments displays are connected to Port 02. If a byte of data is sent to this port, wherever there is a one, the corresponding segment comes on. The rightmost bit controls which of the two groups of segments is active. This is a simple example of mulitplexing. If the least significant bit (LSB) is zero, the left segments will be active. If the least significant bit (LSB) is one, the right segments will be active. Here is a code snippet.</p> <pre> ; ===== ; ===== 99sevseg.asm ===== ; ===== Seven Segment Displays Port 02 ===== Start: MOV AL,FA ; 1111 1010 OUT 02 ; Send the data in AL to Port 02 MOV AL,0 ; 0000 0000 OUT 02 ; Send the data in AL to Port 02 MOV AL,FB ; 1111 1011 OUT 02 ; Send the data in AL to Port 02 MOV AL,1 ; 0000 0001 OUT 02 ; Send the data in AL to Port 02 JMP Start END ; ===== </pre>

Heater and Thermostat Port 03

How to Use



The heater and thermostat system is connected to Port 03. Send 00 to port 3 to turn the heater off. Send 80 to port 03 to turn the heater on. Input from port 03 to test the thermostat state. The code snippet below is an incomplete solution to control the heater to keep the temperature steady at about 21 C. You can click the thermometer to set the temperature. This can save time when you are testing the system.

```
; ===== Heater and Thermostat on Port 03 =====
; ===== 99Heater.asm =====
; ===== Heater and Thermostat on Port 03 =====
MOV     AL,0      ; Code to turn the heater off
OUT      03       ; Send code to the heater

IN       03       ; Input from Port 03
AND      AL,1     ; Mask off left seven bits
JZ       Cold     ; If the result is zero, turn the
                  ; heater on

HALT      ; Quit

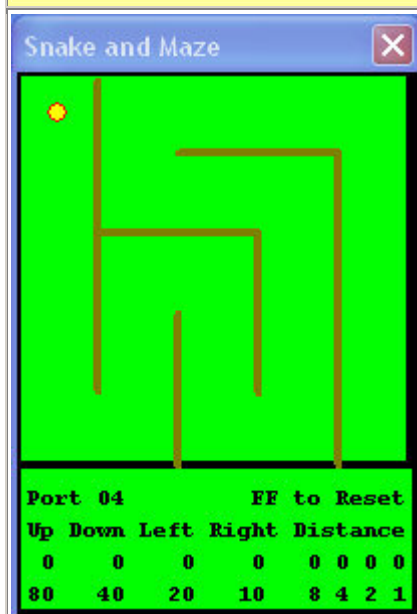
Cold:
MOV      AL,80    ; Code to turn the heater on
OUT      03       ; Send code to the heater

END

;
=====
```

Snake and Maze Port 04

How to Use



The left four bits control the direction of the snake.

- 80 Up
- 40 Down
- 20 Left
- 10 Right

The right four bits control the distance moved.

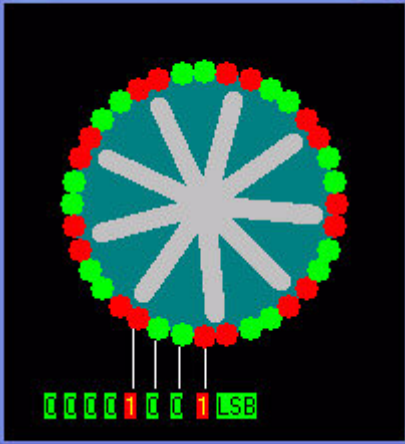
For example, 4F means Down 15. 4 means down. F means 15.

This program is rather wasteful of RAM. If you want to traverse the entire maze and go back to the start, you will run out of RAM. A good learning task is to use a data table. This reduces the size of the program greatly. Also, it is good style to separate code and data.

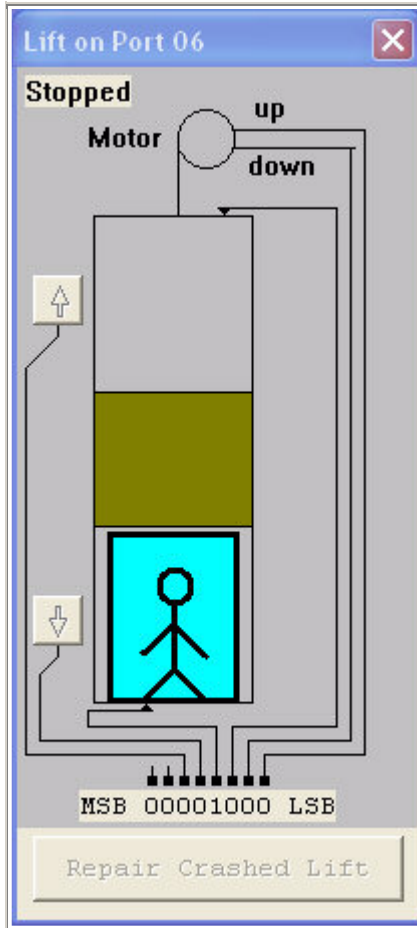
Here is a code sample - not using a data table.

```
;
=====
=====
; ===== 99snake.asm =====
; ===== Snake and Maze =====
```


	<pre> Start: MOV AL,FF ; Special code to reset the snake. OUT 04 ; Send AL to port 04 to control the ; snake. MOV AL,4F ; 4 means DOWN. F means 15. OUT 04 ; Send 4F to the snake OUT 04 ; Send 4F to the snake OUT 04 ; Send 4F to the snake OUT 04 ; Send 4F to the snake JMP Start END ; ===== </pre>
--	---

Stepper Motor Port 05	How to Use
<div> <div>Stepper Motor - Port 5</div>  </div>	<p>Here is a stepper motor. Normal motors run continuously and it is hard to control their movement. Stepper motors step through a precise angle when electromagnets are energised. Stepper motors are used for precise positional control in printers, plotters, robotic devices, disk drives and for any application where precise positional accuracy is required.</p> <p>The motor is controlled by energising the four magnets in turn. It is possible to make the motor move in half steps by energising single and pairs of magnets. If the magnets are energised in the wrong sequence, the motor complains by a bleep from the computer speaker. Here is a code snippet to control the motor. Note that it would be better coding style to use a data table.</p> <pre> ; ===== ; ===== 99Step.asm ===== ; ===== Stepper Motor ===== mov al,1 out 05 mov al,2 out 05 mov al,4 out 05 mov al,8 out 05 mov al,9 out 05 mov al,1 out 05 mov al,3 out 05 mov al,2 out 05 mov al,6 out 05 mov al,4 out 05 mov al,c out 05 mov al,8 out 05 mov al,9 out 05 mov al,1 out 05 end ; ===== </pre>

Lift/Elevator Port 06	How to Use
--------------------------	------------



Input Signals

Bits 8 and 7 are unused. Bit 6 is wired to the top call button. Bit 5 is wired to the bottom call button. If these buttons are clicked with the mouse, the corresponding bits come on. Bit 4 senses the lift and goes high when the lift cage reaches the bottom of the shaft. Bit 3 senses the lift and goes high when the lift cage reaches the top of the shaft.

Outputs

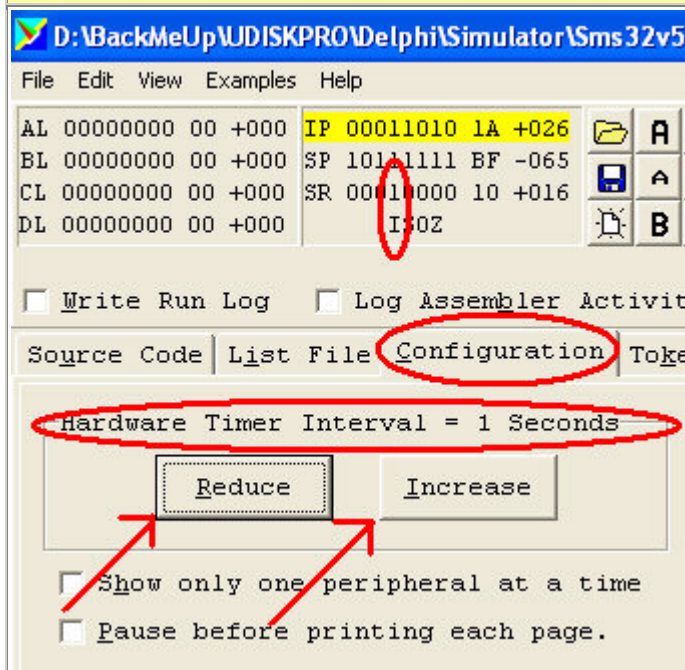
Bit 2 turns on the lift motor and the cage goes down.

Bit 1 turns on the lift motor and the cage goes up.

Ways To Destroy the Lift

1. Turn on bits 1 and 2 at the same time. This causes the motor to go up and down simultaneously!
2. Crash the lift into the bottom of the shaft.
3. Crash the lift into the top of the shaft.
4. Run the simulation too slowly. Even if the code is logically correct, the lift crashes into the end of the shaft before the program has time to switch off the motor.

Hardware Timer INT 02



How to Use


The hardware timer generates **INT 02** at regular time intervals. The time interval can be changed using the Configuration tab as shown in the image. The CPU will ignore **INT 02** unless the (I) flag in the status register (SR) is set. Use **STI** to set the (I) flag. Use **CLI** to clear the (I) flag.

The code sample below processes **INT 02** but does nothing useful.

If the CPU clock is too slow, a new **INT 02** can occur before the previous one has been handled. This is not necessarily a problem as long as the CPU eventually catches up. To allow this to work, it is essential that the interrupt handler saves and restores any registers it uses. Use **PUSH** and **PUSF** to save registers. Use **POPF** and **POP** to restore registers. Remember to pop items in the reverse order that they were pushed. Code like this is "re-entrant".

If the CPU is too slow and does not catch up, the stack will gradually grow and eat up all the available RAM. Eventually the stack will overwrite the program causing a crash. It is a useful learning exercise to slow the CPU clock and watch this happen.

	<pre> jmp start db 10 ; Hardware Timer Interrupt Vector ; ===== Hardware Timer ===== org 10 nop nop ; Do something useful here nop nop nop nop ired ; ===== ; ===== Idle Loop ===== start: STI ; Set (I) flag idle: nop ; Do something useful here nop nop nop nop jmp idle ; ===== end ; ===== </pre>
--	---

Numeric Keypad Port 08 INT 04	How to Use
	<p>This is one of the more complex devices. To make the numeric keypad visible, use OUT 08. Every time a key is pressed, a hardware interrupt, INT 04 is generated. By default, the CPU will ignore this interrupt. To process the interrupt, at the start of the program, use the STI command to set the interrupt flag (I) in the CPU status register (SR). Place an interrupt vector at RAM address 04. This should point to your interrupt handler code. The interrupt handler should use IN 08 to read the key press into the AL register.</p> <p>Once STI has set the (I) flag in the status register (SR), interrupts from the hardware timer will also be generated. These must be processed too. The hardware timer generates INT 02. To process this interrupt, place an interrupt vector at RAM location 02. This should point to the timer interrupt handler code. The timer code can be as simple as IRET. This will cause an interrupt return without doing any other processing.</p> <pre> jmp start db 10 ; Hardware Timer Interrupt Vector db 00 ; Keyboard Interrupt Vector (unused) db 20 ; Numeric Keypad Interrupt Vector ; ===== Hardware Timer ===== org 10 nop nop ; Do something useful here nop nop nop ired </pre>

```

; =====

; ===== Keyboard Handler =====
    org     20
    CLI             ; Prevent re-entrant use
    push     al
    pushf

    in        08
    nop              ; Process the key press here
    nop
    nop
    nop
    nop

    popf
    pop     al
    STI
    iret

; =====

; ===== Idle Loop =====
start:
    STI             ; Set (I) flag
    out      08     ; Make keypad visible
idle:
    nop              ; Do something useful here
    nop
    nop
    nop
    nop
    jmp      idle

; =====
    end
; =====

```