5/12/2019

8051 Simulator

Getting Started

Jonathan Freaney

Getting Started with the 8051 Simulator

1. Go here https://www.edsim51.com/.
2. Click the obvious "Download" link in the middle of the page (you can't miss it).
3. Open the downloaded .zip file with whatever program you choose.
4. Extract the contents (all of them) somewhere you'll be able to find them later.
5. Assuming you have Java installed, just double click the edsim51.jar file to open the program.

8051 How to use Assembly

This section will lead you through the basics of programming in 8051 assembly. If you get stuck at any point, you can always check the simulator layout guide or the instruction set reference at the end of this guide. If you still can't figure it out, try asking for help or utilizing the internet.

Moving Memory with MOV

Values can be moved around in memory using a variety of instructions. For now, we'll just focus on the MOV instruction. The syntax for move looks like the following:

```
MOV destination, source
```

To store a number in a register, we put the register as the destination, and the number (preceeded by a "#" symbol) first. For example,

```
MOV R0, #5
```

will store the value 5 in the general purpose register R0. The "#" symbol preceeding the number indicates that the number is an "immediate" value – meaning that the number will be stored as part as the instruction, rather than elsewhere in memory. Values can also be copied from one register to another. For example,

```
MOV R1, R0
```

will copy the value currently stored in R1 into R0. If executed after the last instruction, both R0 and R1 will contain the value 5. Note that the value is only copied. So, if the value from R0 is replaced, R1 will not be effected. Values can also be copied into RAM by addressing the memory directly. For example,

```
MOV 32, #5
```

will store the value 5 in the byte with memory address 32. The memory address can also be written in hexadecimal as 20H, which is often easier. Values can be copied back and forth between registers and ram as well. It is also possible address a memory location indirectly by using a memory address stored in a register (this only works with registers R0 and R1). For example,

```
MOV R0, #64
MOV @R0, #5
```

will first store the immediate value 64 (40H) in R0. Then the immediate value 5 is stored in the address that is itself stored in R0. The result is the equivalent of saying something like,

```
MOV 64, #5
```

which would put the immediate value 5 directly into the memory address 64. The advantage to this is that memory addresses can be stored, passed around, changed, and dynamically specified.

Practice 1:

Store the immediate value 2 in R2.
Copy R2 into R3.
Store the immediate value 7 in R2.
Does changing R2 affect R3?

Store the value in R3 in the memory address 42.
Manually edit the memory address 69 in the memory display component and set it to 3.
Copy the the value stored at memory address 69 into R5.

Manually store the immediate value 7 in R0.
Store the immediate value 4 in the memory address now pointed to by the value in R0.

Arithmetic and Logic

Thanks to the ALU, the 8051 can perform various arithmetic and logic operations. Most operations make use of a special register called the accumulator, often denoted ACC or just A. For instance, two numbers can be added together with the ADD instruction.

```
MOV A, #2
ADD A, #3
```

The first operand is the accumulator, A, and the second operand is the immediate value 3. Notice that, to add two numbers, A must already contain the first number to be added. This can be done by moving the immediate value 2 into A with a MOV instruction. The result of the addition is stored in A, meaning that A's original value is not retained. The ADD instruction also affects the carry bit, CY, and the auxilliary carry bit, AC, of the PSW.

Practice 2:

Add the numbers 63 and 37.
Add the numbers 250 and 20.
How are the CY and AC flags affected by these operations?

The ADDC instruction is very similar to the ADD instruction, except for one difference. ADDC also adds the carry flag to the result. For example,

```
MOV A, #255
ADD A, #1       ; CY gets set to 1 & A is reset to 0.
ADDC A, #1      ; A is set to 2 instead of 1.
```

The purpose of ADDC is to allow for chained multiplication of larger integers. This makes it possible to do addition of 32-bit numbers on a system that only does 8-bit addition.

Practice 3:

Add together the two 2-byte numbers 384 and 408 using the ADD and ADDC instructions and store the low-byte of the result in the register R2 and the high-byte in R3.

A similar instruction is SUBB, which does subtraction. SUBB will always do subtraction the two given numbers, but will also subtract the carry bit. For example,

```
; Assume CY is set.
MOV A, #5
SUBB A, #1      ; A is set to 3 instead of 4.
```

There is no subtraction equivalent to the plain ADD instruction. Other similar instructions include INC (increment) and DEC (decrement). However, note that unlike ADD, ADDC, and SUBB, the INC and DEC instructions will not affect the carry flags.

The 8051 can also do multiplication and division, both of which are the only two instructions to use the B register. The first thing to note about multiplication is that in multiplying two n-bit numbers together the result will be a 2 × n-bit number. The 8051 handles this by storing the low and high bytes in two separate registers. The notation is as follows.

```
MUL AB
```

The two numbers must be stored in the A and B registers. The low byte of the output will be put in the A register and the high byte in the B register. The bytes can then be retrieved individually, or discarded if unnecessary. Similarly, division of two n-bit numbers will produce two outputs: an n-bit quotient and an n-bit remainder. The notation is as follows.

```
DIV AB
```

The two numbers must be stored in the A and B registers. The quotient is put into the A register and the remainder is put into the B register. Note that MUL will set the overflow flag, OV, if the result is too large, and DIV will set the overflow flag if the denominator was 0.

Practice 4:

Perform the computation of 6 * 9 / 5.
What are the values left in A and B?

Labels and Jumps

Because the PC register is used to determine the location of the next instruction, it cannot be accessed in the same way that other registers can – ie. with the MOV instruction. Instead, PC gets its own special set of instructions. We call these "jumps", since they literally correspond to a jump from one instruction to another. The most basic of these is the JMP instruction.

To perform a jump in assembly, the place to jump to should be labeled. A label is any name, which can include letters, numbers, and underscores, etc., that is ended with a colon. For example, "L0:", "MY_LABEL:", and "TOE_NAILS42:" are all examples of valid labels. To perform a jump to a label, use the JMP instruction.

```
MOV A, #1
MOV R0, #2
JMP THIS_IS_A_LABEL
```

```
MOV R0, #3
THIS_IS_A_LABEL:
     ADD A, R0
```

In the above code, the immediate values 1 and 2 are moved into A and R0, respectively. The JMP instruction then moves the program to the place where the label is defined, effectively skipping the third MOV instruction, and then adds A and R0. Note also that it is custom, but not necessary, to indent code after a label.

Another pair of jump instructions is JZ and JNZ. These are called conditional jumps because they only happen when some condition is true. In particular, JZ only jumps if the value in the accumulator is zero, and JNZ only jumps if the value in the accumulator is not zero (hence the N). The syntax for these instructions is the same as JMP.

Practice 5:

Initialize A to 250.
Increment A.
While A is not 0, return to the point right after the initialization.
Use the JNZ instruction.

Another pair of jump instructions is JC and JNC. These are also conditional jumps. JC jumps only if the carry flag has been set, and JNC jumps only if it has not been set. The syntax for these is also the same as JMP.

Practice 6:

Initialize A to 250.
Add 2 to A.
While the carry flag has not been set, return to the point right after the initialization.
Use the JNC instruction.

Register Banks and the Stack

The 8051 architecture provides two good ways of saving values for later: register banks and the stack.

The general purpose registers actually exist in the RAM. The first 32 bytes of ram are used. By default, the first 8 bytes are used. However, this may be changed to the second 8 bytes, the third set of 8 bytes, or the fourth set of 8 bytes. Which is used is determined by the RS1 and RS0 bits of the PSW register. These can be changed using the SETB and CLR instructions. SETB "sets" a bit (to 1) and CLR "clears" a bit (sets it to 0). For example,

```
SETB RS1
MOV R0, #1
```

will change to the third register bank and set the first register their to 1.

Practice 7:

Set R5 in all four register banks to the immediate value 255.

The stack is also implemented in the RAM. The stack is a FIFO (First-In-First-Out) kind of structure. By default, the first item in the stack will be at address 8. The current position is determined by the SP (stack pointer) register. SP will always be 1 less than the current position in the stack. The stack can be moved to other locations by setting its value with a MOV instruction. This will only move the stack pointer though, not all the items currently stored on the stack – so be careful. There are two instructions for manipulating the stack: PUSH and POP. PUSH pushes something onto the stack, and POP takes something off the stack. For example,

```
MOV 0H, #5
PUSH 0H
MOV 0H, #8
POP 0H
```

puts the immediate value 5 in memory address 0 and then pushes its value (5) onto the stack. It then does something else with memory address 0 (puts the immediate value 8 there in this case), and then restore the value of memory address 0 by popping it off the stack and back into that address. In essence, the stack is great for saving and then later restoring values in memory. PUSH and POP can only be used with memory addresses – not registers.

Practice 8:

Store any value in memory address 6.
Save its value on the stack.
Replace the value in memory address 6 with a new value.
Restore the value in memory address 6 from the stack.

Subroutines

Sometimes a particular task can become redundant to write code for. In an assembly language, this is especially true as there are only a limited number of instructions and most tasks involve doing very similar things with them. A subroutine is a set of instructions designed to perform a frequently used operation within a program. In a sense, this is like writing a function in a language like C or Java, but just a little bit lower-level – especially as we must do so much more of the switching ourselves.

Say you wanted to set all the registers to zero. There are eight registers, so writing this out takes eight instructions. If you had several places where you wanted to do this, it would become a pain to write out eight instructions every time. Instead we can use a subroutine and only write it once. To do this we use the CALL (really ACALL or LCALL) and RET instructions. For example,

```
JMP START

ALL_THE_ZEROS:
      MOV R0, #0
      MOV R1, #1
      ...
      MOV R7, #7
      RET               ; Return from subroutine call.

START:
      MOV R3, #7
```

```
CALL ALL_THE_ZEROS    ; Call the subroutine.
MOV R7, #4
CALL ALL_THE_ZEROS    ; Call it again.
```

The CALL instruction looks similar to a JMP instruction in that you use a label to specify where to go in the code. The difference between CALL and JMP is that CALL pushes the PC address of the next instruction after CALL onto the stack before it peforms the jump. Calling the subroutine jumps to the labeled subroutine and then continues from there. When the subroutine is done, the RET instruction is used. RET pops the PC address off the stack that was originally stored there by the last CALL instruction and then jumps to that address and continues execution where it left off.

Practice 9:

Store numbers in register R0 and R1.
Write a subroutine that adds the number in R0 with four times the number in R1.
You can do this with multiplication or by adding R1 four times.
The result should be in R1.
Call the subroutine.

One more thing. Calling a subroutine can be dangerous. When you call a subroutine, it can change the values in the registers you are using. If you're not careful, you might get unexpected behavior. To work around this, save the values in the registers before making a call and then restore them afterwords. This is commonly done by pushing the registers' values onto the stack and then popping them off afterwards.

Practice 10:

Make a subroutine that changes the values of R0 and R1 however you want.
When the program starts, initialize R0 and R1 to some value.
Push the values of R0 and R1 onto the stack.
Call the subroutine.
Pop the values of R0 and R1 back off the stack in the correct order.

Congradulations. There's a lot more you can do with assembly languages, but hopefully now you are starting to get a grasp on how to use them and how beautiful they can be.

8051 Simulator Layout

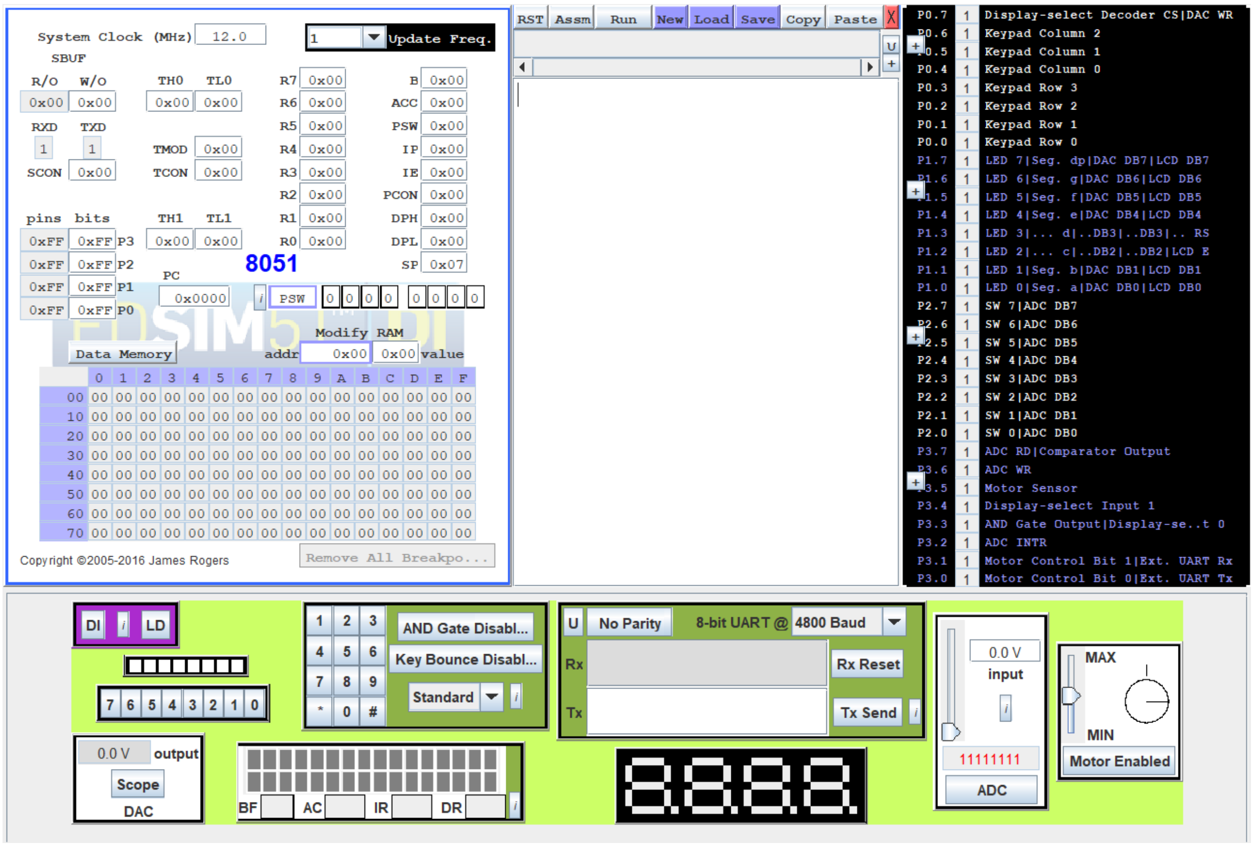

Figure 1 - 8051 Simulator

The simulator window is broken into four components: a memory display (top left), an assembly editor (top middle), a port display (top right), and a display of all the peripherals connected to the 8051 (bottom).

Memory Display

The memory display (top left) is an interactive panel that displays all of the internal memory components of the 8051. Each field can be edited manually. Unfortunately, there is no way to reset their values to zero without closing and reopenning the program.

At the bottom is a table displaying the memory at each address in RAM. There are two parts to it: data and code. Pressing the "Data Memory" / "Code Memory" button will switch between them. To edit the value at a memory address, edit the two fields at the top

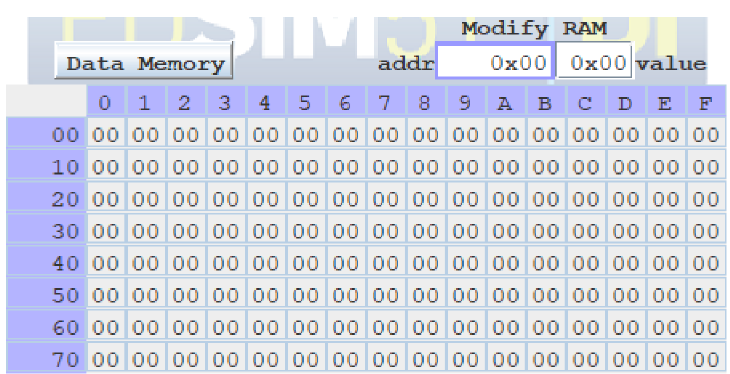

Figure 2 - RAM

right until "Modify Ram". These represent the internal RAM of the 8051.

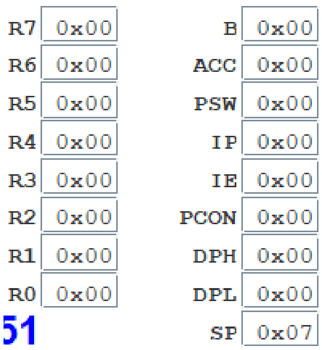

Figure 3 - General & Special Registers

Note: The DPTR register is a 16-bit register and can be addressed as just DPTR. However, DPTR's high and low bytes can also be addressed individually as DPH and DPL.

Around the top right are two columns of fields. The left column represents the general purpose registers, named R0 – R7. The right column represents the special purpose registers. The special purpose registers service specific uses.

Register	**Description**
B	Only used by the MUL and DIV instructions. Often also used like a 9th "R" register.
ACC	Accumulator – used to hold the cumulative results of a large number of operations. Used by pretty much every instruction.
PSW	Processor Status Word – also called the flag register.
IP	Interrupt Priority – how important the current interrupt is. Priorities determine whether an interrupt is important enough to interrupt another interrupt.
IE	Interrupt Enable – enables or disables an interrupt.
PCON	Power Control – used to force the 8051 into power saving mode. Made from several bit flags.
DPH	Data Pointer High – the high byte of the DPTR register.
DPL	Data Pointer Low – the low byte of the DPTR register.
SP	Stack Pointer – points to the top of the stack.

The PC register (shown to the right) – called the Program Counter – is used to determine the address of the next instruction to execute. Normally, the PC register's value cannot be assigned a value like other registers can. However, this can be done with a jump instruction, which is the equivalent of moving elsewhere in code.

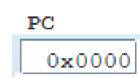

Figure 4 - PC Register

Figure 5 - Bit Addressable Fields

The component (shown to the left) displays the bits within a byte.

The left-most field displays the name of the byte being edited. By default, it is the PSW register, but can be changed to any other register names – such as ACC, B, DPH, etc. – or memory address. If the address if bit-addressable, then it can be edited.

Symbol	**Function**
CY	Carry flag
AC	Auxilliary carry flag
F0	Available to user for general purpose use
RS1	Register bank selector bit 1
RS0	Register bank selector bit 0
OV	Overflow flag
UD	User definable flag

An an important note, the PSW register can be P Parity flag
broken down into 8 individual bits, as shown in the table to the right. The CY, AC, OV, and P bits are conditional flags set by various instructions. RS1 and RS0 are used to determine the current register bank – of which there are four.

The other components shown in the memory display will not be discussed in this document.

Editor

The assembly editor serves two main purposes: writing assembly code, and testing it. Assembly code can be entered in the white text area. Above that is a small box with scrollbars that displays error messages when they occur. To actually assemble and run code, click the "Assm" button.

8051 Instruction Set Reference

NOP - No Operation

Description Literally does nothing but waste time for an entire machine cycle. Commonly used for delays.

Syntax
```
NOP
```

ANL - Bitwise AND

Description Performs a bitwise logical AND operation. Only the first operand is affected.

Syntax
```
ANL addr, A
ANL addr, #imm
ANL A, #imm
ANL A, addr
ANL A, @R0
ANL A, @R1
ANL A, R[0...7]
```

ORL - Bitwise OR

Description Performs a bitwise logical OR operation. Only the first operand is affected.

Syntax
```
ORL addr, A
ORL addr, #imm
ORL A, #imm
ORL A, addr
ORL A, @R0
ORL A, @R1
ORL A, R[0...7]
```

XRL - Bitwise Exclusive OR

Description Performs a bitwise logical XOR operation. Only the first operand is affected.

Syntax
```
XRL addr, A
XRL addr, #imm
XRL A, #imm
XRL A, addr
XRL A, @R0
XRL A, @R1
XRL A, R[0...7]
```

ADD, **ADDC** - Add Accumulator, Add Accumulator With Carry

Description Both ADD and ADDC add the value of the given operand to the accumulator. The value of the operand is not affected. They are pretty much the same except that ADDC also adds the value of the carry flag to the accumulator as well as the operand.

Syntax

```
ADD addr, A          ADDC addr, A
ADD addr, #imm       ADDC addr, #imm
ADD A, #imm          ADDC A, #imm
ADD A, addr          ADDC A, addr
ADD A, @R0           ADDC A, @R0
ADD A, @R1           ADDC A, @R1
ADD A, R[0...7]      ADDC A, R[0...7]
```

SUBB - Subtract From Accumulator With Borrow

Description Subtracts the given value from the value in the accumulator.

Syntax

```
SUBB addr, A
SUBB addr, #imm
SUBB A, #imm
SUBB A, addr
SUBB A, @R0
SUBB A, @R1
SUBB A, R[0...7]
```

MUL - Multiply Accumulator by B

Description Multiplies the *unsigned* value in the accumulator (A) by the *unsigned* value in B. Note that multiplying two n-bit numbers together will produce a 2 × n-bit number. The lower-order byte is placed in the accumulator and the higher-order byte is placed in B. Both registers are affected.

Syntax `MUL AB`

DIV - Divide Accumulator by B

Description Divides the *unsigned* value in the accumulator (A) by the *unsigned* value in B. Note that dividing two n-bit numbers will produce an n-bit quotient and an n-bit remainder. The quotient is placed in the accumulator and the remainder in B. Both registers are affected.

Syntax `DIV AB`

INC - Increment Register

Description Increments the value in the given register. Note that this does not affect the carry flag if the result rolls over from the maximum value to 0.

Syntax

```
INC A
INC addr
INC @R0
INC @R1
INC R[0...7]
INC DPTR
```

DEC - Decrement Register

Description Decrements the value in the given register. Note that this does not affect the carry flag if the result rolls over from 0 to the maximum value.

Syntax
```
DEC A
DEC addr
DEC @R0
DEC @R1
DEC R[0...7]
```

RL - Rotate Accumulator Left

Description Shifts the bits of the accumulator to the left. The left-most bit is loaded into the right-most bit.

Syntax `RL A`

RLC - Rotate Accumulator Left Through Carry

Description Shifts the bits of the accumulator to the left. The carry flag is loaded into the right-most bit and *then* the left-most bit is loaded into the carry flag. Useful for multiplying by 2.

Syntax `RL A`

RR - Rotate Accumulator Right

Description Shifts the bits of the accumulator to the right. The right-most bit is loaded into the left-most bit.

Syntax `RR A`

RRC - Rotate Accumulator Right Through Carry

Description Shifts the bits of the accumulator to the right. The carry flag is loaded into the left-most bit and *then* the right-most bit is loaded into the carry flag. Useful for dividing by 2.

Syntax `RL A`

AJMP - Absolute Jump

Description Absolutely, unconditionally jumps to the given address by setting the program counter to it and changing the page.

Syntax `AJMP addr`

SJMP - Short Jump

Description Absolutely, unconditionally jumps to the given address by setting the program counter to it. The address must be within -128 to +128 bytes of the following instruction.

Syntax `SJMP addr`

LJMP - Long Jump

Description Absolutely, unconditionally jumps to the given address by setting the program counter to it.

Syntax `LJMP addr`

JMP - Jump to Address

Description Absolutely, unconditionally jumps to the address computed as the sum of the DPTR and the value of the Accumulator.

Syntax
```
LJMP @A+DPTR
```

CJNE - Compare and Jump if Not Equal

Description Compares the value of the first two operands and jumps to the given address if they are NOT equal.

Syntax
```
CJNE A, #imm, addr
CJNE A, addr, addr
CJNE @R0, #imm, addr
CJNE @R1, #imm, addr
CJNE R[0...7], #imm, addr
```

DJNZ - Decrement Register and Jump if Not Zero

Description Decrements the value of the register. If the new value is not zero, then jump to the given address.

Syntax
```
DJNZ addr, addr
DJNZ R[0...7], addr
```

JB - Jump if Bit Set

Description If the bit at the given bit address is set, then jump to the given address.

Syntax
```
JB bit-addr, addr
```

JNB - Jump if Bit Not Set

Description If the bit at the given bit address is NOT set, then jump to the given address.

Syntax
```
JNB bit-addr, addr
```

JBC - Jump if Bit Set and Clear Bit

Description If the bit at the given bit address is set, clear the bit and then jump to the given address.

Syntax
```
JBC bit-addr, addr
```

JC - Jump if Carry Set

Description If the carry bit is set, then jump to the given address.

Syntax
```
JC addr
```

JNC - Jump if Carry Not Set

Description If the carry bit is NOT set, then jump to the given address.

Syntax
```
JNC addr
```

JZ - Jump if Accumulator Zero

Description If the value in the accumulator is zero, then jump to the given address.

Syntax
```
JZ addr
```

JNZ - Jump if Accumulator Not Zero

Description If the value in the accumulator is NOT zero, then jump to the given address.

Syntax `JNZ` *`addr`*

ACALL - Absolute Call

Description Absolutely, unconditionally calls a subroutine at the given code address. This happens in two steps:

1. The address of the next instruction (directly after ACALL) is pushed onto the stack.
2. The program counter is changed to the given code address.

Uses only 2 bytes and can only specify addresses in the 2K-byte range around it.

Syntax `ACALL` *`addr`*

LCALL - Long Call

Description Absolutely, unconditionally calls a subroutine at the given code address. This happens in two steps:

1. The address of the next instruction (directly after ACALL) is pushed onto the stack.
2. The program counter is changed to the given code address.

Uses 3 bytes and can specify addresses anywhere in the 8051's 64K-byte internal memory.

Syntax `LCALL` *`addr`*

RET - Return From Subroutine

Description Returns from a subroutine called by ACALL or LCALL. This happens in two steps:

1. An address is popped off the stack.
2. The program counter is changed to that address.

Syntax `RET`

RETI - Return From Interrupt

Description Returns from an interrupt service routine. This happens in three steps:

1. Interrupts of equal or lower priorities to the current interrupt that is terminating are enabled.
2. An address is popped off the stack.
3. The program counter is changed to that address.

Syntax `RETI`

PUSH - Push Value Onto Stack

Descriptor Pushes the value at the given address onto the stack. This happens in two steps:

1. The stack pointer is incremented.
2. The value at the given address is copied to the stack pointer's address.

Syntax `PUSH` *`addr`*

POP - Pop Value From Stack

Descriptor Pops a value from the stack into the given address. This happens in two steps:

1. The value at the stack pointer's address is copied to the given address.
2. The stack pointer is decremented.

Syntax `POP` *`addr`*

MOV - Move Memory

Descriptor Copies the value from the second operand into the first.

Syntax

```
MOV @R0, #imm
MOV @R1, #imm
MOV @R0, A
MOV @R1, A
MOV @R0, addr
MOV @R1, addr
MOV A, #imm
MOV A, @R0
MOV A, @R1
MOV A, R[0...7]
MOV A, addr
MOV C, bit-addr
MOV DPTR, #imm
MOV R[0...7], #imm
MOV R[0...7], A
MOV R[0...7], addr
MOV bit-addr, C
MOV addr, #imm
MOV addr, @R0
MOV addr, @R1
MOV addr, R[0...7]
MOV addr, A
MOV addr, addr
```

MOVC - Move Code Memory

Descriptor Copies a value from code memory into the accumulator.

Syntax

```
MOVC A, @A+DPTR
MOVC A, @A+PC
```

MOVX - Move Extended Memory

Descriptor Moves a byte to/from external memory from/to the accumulator.

Syntax

```
MOVX @DPTR, A
MOVX @R0, A
MOVX @R1, A
MOVX A, @DPTR
MOVX A, @R0
MOVX A, @R1
```

SETB - Set Bit

Descriptor Sets the given bit.

Syntax
```
SETB C
SETB bit-addr
```

CLR - Clear Register

Descriptor For an individual bit, sets its value to 0. For a register, sets all of its bits to 0.

Syntax
```
CLR bit-addr
CLR C
CLR A
```

CPL - Complement Register

Descriptor For an individual bit, complements (flips) its value. For a register, complements (flips) all of its bits.

Syntax
```
CPL bit-addr
CPL C
CPL A
```

DA - Decimal Adjust

Descriptor Adjusts the contents of the Accumulator to correspond to a BCD (Binary Coded Decimal) number after two BCD numbers have been added by the ADD or ADDC instruction. If the carry bit is set or if the value of bits 0-3 exceed 9, 0x06 is added to the accumulator. If the carry bit was set when the instruction began, or if 0x06 was added to the accumulator in the first step, 0x60 is added to the accumulator.

Syntax
```
CPL bit-addr
CPL C
CPL A
```

SWAP - Swap Accumulator Nibbles

Descriptor In the accumulator, swaps bits 0-3 with 4-7.

Syntax
```
SWAP A
```

XCH - Exchange Bytes

Descriptor Swaps the contents of the acccumulator with another register. This happens simultaneously and therefore does not require a third register to do the swap.

Syntax
```
XCH A, @R0
XCH A, @R1
XCH A, R[0...7]
XCH A, addr
```

XCHD - Exchange Digits

Descriptor Swaps bits 0-3 of the accumulator with bits 0-3 of an internal memory address. Bits 4-7 are unaffected.

Syntax
```
XCHD A, @R0
XCHD A, @R1
```

Undefined - Undefined Instruction

Descriptor This instruction is undocumented. The 8051 has 256 instructions and opcode 0xA5 is the only one that is not used. Try not to use it.

Syntax ???