How to Do Stuff in 805x Assembly (WORK IN PROGRESS)

Amar Shah

April 17, 2019

In this document, assembly code is shown along side C code to demonstrate how to translate higher level programming constructs into assembly.

This is a work in progress, please hold off on printing this document until just before your exam.

Contents

1 Startup **2**

2 Variables **2**
2.1 Global Variables . . . 2
2.2 Defines, Includes . . . 3
2.3 Macros . . . 3
2.4 Constant Lookup Tables . . . 3
2.5 External Memory Access . . . 4

3 Control Flow **4**
3.1 Function Calls/Subroutines . . . 4
3.2 If-Elseif-Else . . . 4
3.3 While Loop . . . 5
3.4 For Loop . . . 6

4 Arithmetic **6**
4.1 Addition/Subtraction . . . 6
4.2 Multiplication/Division . . . 6
4.3 Multi-Byte Addition/Subtraction . . . 7
4.4 Multi-Byte Comparison . . . 7
4.5 Bit Shifting . . . 8
4.6 Extracting BCD digits . . . 9
4.7 Binary to BCD . . . 9
4.8 BCD to Binary . . . 10
4.9 BCD Addition . . . 11
4.10 BCD Subtraction . . . 12

5 Timers and Interrupts **13**
5.1 Timer Frequency and Reload Value . . . 13
5.2 Enabling a Timer . . . 13
5.3 Timer Interrupts . . . 13
5.4 Using a Timer as a Counter . . . 14

6 CV-8052 Hardware **14**
6.1 LEDs 14
6.2 7-Segment Displays 15
6.3 Keys and Switches 15

1 Startup

The 8051 begins execution from address 0, but other stuff may need to come in the addresses immediately following, such as interrupt handlers, so place the main code elsewhere and jump to it. Before going to the rest of the program, the stack pointer should be set and any variables that need initialization initialized. (A C compiler would insert code to do this automatically).

```c
void main(void) {
    //Your code here
}
```

```asm
CSEG
ORG 0x0000
    ljmp main
ORG 0x0003
     ;Interrupt handlers could go here
main:
    ;Set the stack pointer
    mov SP, #0x7F
    ;Initialize variables
    ;Your code here

END
```

2 Variables

2.1 Global Variables

In assembly, rather than declaring the variable and initializing it at the same time, you must declare the storage for the variables, and initialize it at some point in the start of the program.

With the 8052, registers and bit addressable memory occupies addresses below 0x30, and the stack is commonly set to above 0x7F. Thus, it is common to place the data segment, DSEG, in memory at address 0x30.

C

```c
char x;
char y = 7;
long z = 1030;
```

Assembly

```asm
DSEG at 0x30
    x: ds 1
    y: ds 1
    z: ds 4

initialize_variables:
    mov y, #7
    mov z+0, #6
    mov z+1, #4
    mov z+2, #0
    mov z+3, #0
```

2.2 Defines, Includes

C

```c
#include <math.h>

#define INPUT_TIMEOUT 180
```

Assembly

```
$include(math32.asm)

INPUT_TIMEOUT EQU 180
```

2.3 Macros

C

```c
#include <math.h>

#define DEC_DPTR (--DPTR)
```

Assembly

```
MAC DEC_DPTR
    mov A, DPL
    ;can't use labels in macros
    ;If you use the macro twice
    ;It will be defined twice
    ;Compute the offset directly
    jz $+3
    dec DPH
    dec DPL
ENDMAC
```

2.4 Constant Lookup Tables

Lookup tables are placed in code memory in the 8051. To access, set DPTR to the address of the lookup table, and A to the index to access, and use the mov A, @A+DPTR instruction, but make sure A is within the size of the table!

C

```c
const char lut_7seg[] = {
    0xC0, 0xF9, 0xA4, 0xB0, 0x99,
    0x92, 0x82, 0xF8, 0x80, 0x90,
    0x88, 0x83, 0xA7, 0xA1, 0x86,
    0x8E
};

//...
HEX0 = lut_7seg[x];
```

Assembly

```
lut_7seg:
    DB 0xC0, 0xF9, 0xA4, 0xB0, 0x99
    DB 0x92, 0x82, 0xF8, 0x80, 0x90
    DB 0x88, 0x83, 0xA7, 0xA1, 0x86
    DB 0x8E

    ;...
    mov A, x
    mov dptr, #lut_7seg
    mov A, @A+dptr
    mov HEX0, A
```

2.5 External Memory Access

To access a byte in external memory, load the address in DPTR and use movx to move the byte into A. There is only an instruction to increment the data pointer, not to decrement. To decrement, see the macro section above.

The data pointer can also be accessed using DPL and DPH.

Assembly

```
    mov DPTR, #0x3A08
    mov A, @DPTR
    mov R0, A
    inc DPTR
    mov A, @DPTR
    mov R1, A
    ;R0 has contents of XRAM at 0x3A08
    ;R1 has contents of XRAM at 0x3A09
```

3 Control Flow

3.1 Function Calls/Subroutines

Let's assume the assembly subroutine Wait takes an argument in R0 of how long to wait, and it does not preserve R1 or R2, which the surrounding code is currently using. We must push R1 and R2 to the stack before calling the routine and pop them afterwards to preserve their values.

C

```
wait(47);
```

Assembly

```
mov R0, #47
push R1
push R2
lcall Wait
pop R2
pop R1
```

3.2 If-Elseif-Else

The CJNE instruction is very useful for branches, but it's not the only way to do it.

C

```c
if (x == 3) {
    fizz();
}
else if (x == 5) {
    buzz();
}
else {
    print_x();
}
```

Assembly

```
L_if:
    cnje R0, #3, L_else_if
    lcall Fizz
    sjmp L_endif
L_else_if:
    cjne R0, #5, L_else
    lcall Buzz
    sjmp L_endif
L_else:
    lcall Print_R0
L_endif:
    ; rest of the program here
```

Since the CJNE performs a subtraction without writing the result to the accumulator, but still setting the carry flag, it can be used for greater than or less than comparisons as well. If the carry flag is set, the first operand is less than the second operand.

C

```c
if (x < 7) {
    handle_less();
}
else if (x > 7) {
    handle_greater();
}
else {
    handle_equal();
}
```

Assembly

```
L_if:
    cnje x, #7, L_not_equal:
    lcall Handle_Equal
    sjmp L_endif
L_not_equal:
    jnc L_greater
L_less:
    lcall Handle_Less
    sjmp L_endif
L_greater:
    lcall Handle_Greater
L_endif:
    ; rest of the program here
```

3.3 While Loop

Let's assume the Get_Input subroutine reads some hardware and places an input value in R0, or 0 if there is no input.

C

```c
while (1) {
    input = get_input();
    if (input != 0) break;
}
```

Assembly

```
L_check_for_input:
    lcall Get_Input
    mov A, R0
    jnz L_have_input
    sjmp L_check_for_input
L_have_input:
```

3.4 For Loop

A DJNZ instruction is the easiest way to do a loop with a defined number of iterations, but the register used for it will go from num_iterations to 1, rather than 0 to num_iterations - 1. Rather than computing a loop index from that, it is often easier to increment another register at the same time.

C

```c
int i;
for (i = 0; i < 25; i++) {
    foo(i);
}
```

Assembly

```
    mov R1, #25
    mov R0, #0
L_foo:
    lcall Foo
    inc R0
    djnz R1, L_foo
```

4 Arithmetic

4.1 Addition/Subtraction

For single byte-addition use the ADD instruction, which is not affected by the prior state of the carry flag. For subtraction, there is no SUB, only SUBB, so you must use that and clear the carry flag first. If you do not know the carry is clear, you could get incorrect results!

C

```c
char x, y, z, w;
//...
z = x + y;
w = x - y;
```

Assembly

```
mov A, x
add A, y
mov z, A

mov A, x
clr C
subb A, y
mov w, A
```

4.2 Multiplication/Division

If you are multiplying/dividing by a power of 2, use bit shifting instead of these instructions, as it is much faster.

For multiplication, the operators go in A and B. After multiplication, the low byte of the result will be in A, and the high byte in B.

For division, the dividend goes in A, and the divisor in B. The quotient of the result will be in A, and the remainder will be in B.

C

```c
//assuming all values are 8 bits
//and overflows are discarded
char x, y, z, w;
//...
z = x * y;
w = x / y;
```

Assembly

```
mov A, x
mov B, y
mul AB
mov z, A

mov A, x
mov B, y
div AB
mov w, A
```

4.3 Multi-Byte Addition/Subtraction

Use the ADD instruction, or clear carry and SUBB for the least significant byte, then add the next bytes in order of increasing significance using ADDC and SUBB.

C

```c
long w, x, y;
//...
w = x - y;
```

Assembly

```
clr C
mov A, x+0
subb A, y+0
mov w+0, A

mov A, x+1
subb A, y+1
mov w+1, A

mov A, x+2
subb A, y+2
mov w+2, A

mov A, x+3
subb A, y+3
mov w+3, A
```

4.4 Multi-Byte Comparison

When comparing multi-byte number

C

```c
//Let's assume byte(var, N) gets
//the Nth least significant byte
//of a variable.
unsigned long x, y;
//...
bool x_less_than_y = false;
if (byte(x, 3) < byte(y, 3)) {
    x_less_than_y = true;
}
else if (byte(x, 2 < byte(y, 2)) {
    x_less_than_y = true;
}
else if (byte(x, 1 < byte(y, 1)) {
    x_less_than_y = true;
}
else if (byte(x, 0 < byte(y, 0)) {
    x_less_than_y = true;
}
```

Assembly

```
    cjne x+3, y+3, L_end_comparison
    cjne x+2, y+2, L_end_comparison
    cjne x+1, y+1, L_end_comparison
    cjne x+0, y+0, L_end_comparison
L_end_comparison:
    ;If x is less than y, the carry flag
    ;is set from the comparison
```

4.5 Bit Shifting

If you need to shift bits one to the left (multiply by 2) use RLC and go from LSB to MSB if the number has multiple bytes. If you need to shift right (divide by 2), use RRC and go from from MSB to LSB. To shift by n bits, just repeat the process in a loop n times.

C

```c
long x; //4 bytes
//...
x = x >> 3;
```

Assembly

```
    mov R2, #3 ;no. of times to shift
L_shift_word:
    mov R0, #(x+4) ;address of MSB of data
    mov R1, #4 ;no. of bytes to apply shift to
    clr C
L_shift_byte:
    dec R0
    mov A, @R0
    rrc A
    mov @R0, A
    djnz R1, L_shift_byte
    djnz R2, L_shift_word
```

4.6 Extracting BCD digits

C

```c
uint8_t bcd = 0x47;
uint8_t ones = bcd & 0x0F;
uint8_t tens = (bcd >> 4) & 0x0F;
//ones == 7
//tens == 4
```

Assembly

```
; let BCD contain the BCD value to extract
; Move the ones to R0 and the tens to R1

    mov A, BCD
    anl A, #0x0F
    mov R0, A

    mov A, BCD
    swap A
    anl A, #0x0F
    mov R1, A
```

4.7 Binary to BCD

The strategy for binary to BCD conversion is to shift out the most significant bit of the binary number, multiply the BCD number by 2, then add the shifted out bit. For example, if the binary number was 1 followed by 31 zeroes, on the first loop the BCD number would start at zero, be doubled, and then one added to it. On the next 31 loops, the BCD number would be doubled 31 times, giving the final result of $2^{3}1$.

C

```c
unsigned long bin; //32 binary bits
char bcd[5]; //10 bcd digits
//clear bcd
for (int i = 0; i < 5; i++) {
    bcd[5] = 0;
}

for (int bit = 0; bit < 32; bit++) {
    //No way to get the carry flag after an operation in C
    //This gets the bit that will end up in the carry flag after the left shift
    //as a 0 or a 1
    unsigned short carry = (bin >> 31);
    bin = bin << 1;

    for (int i = 0; i < 5; i++)
        //add to a 16 bit value to avoid overflow
        //do the DA magic here
        unsigned short sum = da(carry + bcd[i] + bcd[i]);
        //extract the carry bit, since C can't access the carry flag
        carry = sum >> 8;
        //and the 8 bit sum
        bcd[i] = sum & 0xFF;
    }
}
```

Assembly

```
hex2bcd:
    ;pushes and pops are omitted to save space

    clr a
    mov bcd+0, a ; Initialize BCD to 00-00-00-00-00
    mov bcd+1, a
    mov bcd+2, a
    mov bcd+3, a
    mov bcd+4, a
    mov r2, #32  ; Loop counter.

hex2bcd_L0:
    ; Shift binary left
    mov a, x+3
    mov c, acc.7 ; This way x remains unchanged!
    mov r1, #4
    mov r0, #(x+0)
hex2bcd_L1:
    mov a, @r0
    rlc a
    mov @r0, a
    inc r0
    djnz r1, hex2bcd_L1

    ; Perform bcd + bcd + carry using BCD arithmetic
    mov r1, #5
    mov r0, #(bcd+0)
hex2bcd_L2:
    mov a, @r0
    addc a, @r0
    da a
    mov @r0, a
    inc r0
    djnz r1, hex2bcd_L2

    djnz r2, hex2bcd_L0
hex2bcd_exit:
    ret
```

4.8 BCD to Binary

Don't have time to explain this one. Hopefully Jesus's comments are enough.

Assembly

```
;------------------------------------------------
; bcd2hex:
; Converts the 10-digit packed BCD in 'bcd' to a
; 32-bit hex number in 'x'
;------------------------------------------------
```

```
bcd2hex:
    ;pushes and pops are omitted to save space

    mov r2, #32  ; We need 32 bits

bcd2hex_L0:
    mov r1, #5              ; BCD byte count = 5
    clr c                   ; clear carry flag
    mov r0, #(bcd+4)        ; r0 points to most significant bcd digits
bcd2hex_L1:
    mov a, @r0              ; transfer bcd to accumulator
    rrc a                   ; rotate right
    push psw                ; save carry flag
    ; BCD divide by two correction
    jnb acc.7, bcd2hex_L2  ; test bit 7
    add a, #(100h-30h)      ; bit 7 is set. Perform correction by subtracting 30h.
bcd2hex_L2:
    jnb acc.3, bcd2hex_L3  ; test bit 3
    add a, #(100h-03h)      ; bit 3 is set. Perform correction by subtracting 03h.
bcd2hex_L3:
    mov @r0, a              ; store the result
    dec r0                  ; point to next pair of bcd digits
    pop psw                 ; restore carry flag
    djnz r1, bcd2hex_L1     ; repeat for all bcd pairs

    ; rotate binary result right
    mov r1, #4
    mov r0, #(x+3)
bcd2hex_L4:
    mov a, @r0
    rrc a
    mov @r0, a
    dec r0
    djnz r1, bcd2hex_L4

    djnz r2, bcd2hex_L0

    ret
```

4.9 BCD Addition

The 8051 has the `DA` instruction to easily add numbers in BCD. Your code should add values using the regular `ADD` or `ADDC` instruction and then `DA` will fix up the result to be proper BCD. Unfortunately, it does not work after the `SUBB` instruction, but see the next section for BCD subtraction.

C

```c
//Even though these are hex numbers
//they represent decimal 44 and 79
char x = 0x44;
char y = 0x79;
char z = x + y;
//z now contains 0xBD
z = DA(z);
//z now contains 0x23
//and the carry flag is set.
//The result is interpeted
//as decimal 123.
```

Assembly

```
; Assume memory at x contains 0x44
; and memory at y contains 0x79.
mov A, x
add A, y
; A now contains 0xBD
da A
; A now contains the BCD result 0x23
; and the carry flag is set to 1.
```

This can be extended to multiple bytes, remembering to use `ADD` for the first byte and `ADDC` for susequent:

Assembly

```
; Assume memory at x and y are 2 byte (4 digit) BCD numbers
mov A, x+0
add A, y+0
da A
mov z+0, A

mov A, x+1
addc A, y+1
da A
mov z+1, A
; Result is now in memory at z
```

4.10 BCD Subtraction

There is no way to do BCD subtraction in hardware on the 8051, but we can transform the problem like so:

$$\begin{aligned} X - Y &= X + ((10^n - 1) - Y) - (10^n - 1) \\ &= X + ((10^n - 1) - Y) + 1 - 10^n \end{aligned}$$

In the case of a two digit number, $n = 2$:

$$X - Y = X + (99 - Y) + 1 - 100$$

You can add BCD numbers with the `DA` instruction as in the previous section, and subtracting a two digit BCD number from 99 can be done with the `SUBB` instruction because there is guaranteed to be no borrow. Finally, subtracting 100 can be done by dropping the carry bit if it is set. If it is not set, then the subtraction has underflowed.

Thus, the steps to subtract Y from X are:

1. Replace every digit y_i of Y with 9 - y_i

2. Add the modified value of Y to X.
3. Add 1 to the result.

5 Timers and Interrupts

5.1 Timer Frequency and Reload Value

$$\text{TimerReload} = 2^{16} - \frac{\text{ClockFrequency}}{12\,\text{OverflowFrequency}}$$

$$\text{OverflowFrequency} = 2^{16} - \frac{\text{ClockFrequency}\,/12}{2^{16} - \text{TimerReload}}$$

On an original 8051, the clock frequency is 12 MHz. On a CV-8052, the clock frequency is 33.33 MHz.

If you are using the timer to generate a square wave by toggling an output pin, the timer frequency you need is twice that, because you need two transitions per cycle.

5.2 Enabling a Timer

For Timer 1, use TR1, TH1, TL1, and the upper four bits of TMOD instead.

C

```c
#define XTAL 33333333
#define FREQ 100
#define RELOAD \
65536-(XTAL/(12*FREQ))

//disable the timer first
//while setting its bits
TR0 = 0;
//clear the T0 (lower four) bits of TMOD
TMOD = TMOD & 0b11110000;
//set the bits of TMOD we want
//This is the default 16 bit mode
TMOD = TMOD | 0b00000001;
//start the timer
TH0 = high(RELOAD);
TL0 = low(RELOAD);
TR0 = 1;
```

Assembly

```
XTAL EQU 33333333
FREQ EQU 100
RELOAD EQU 65536-(XTAL/(12*FREQ))

    clr TR0
    mov A, TMOD
    anl A, 0b11110000
    orl A, 0b00000001
    mov TMOD, A
    mov TH0, high(RELOAD)
    mov TL0, low(RELOAD)
    setb TR0
```

5.3 Timer Interrupts

Assuming timer is already set up as shown above above. You need to set ET0 (enable timer interrupt 0) (or ET1 for timer 1) and EA (enable all interrupts) for interrupts to fire.

An example ISR:

Assembly

```

ORG 0x0000
    ljmp Main

ORG 0x000B
    ljmp ISR_Timer0

Main:
    ;setup timer first
    setb ET0
    setb EA
    sjmp $ ;loop forever

ISR_Timer0:
    push PSW ;push any registers the ISR touches
    push ACC
    push AR0
    ;reload timer
    clr TR0
    mov TH0, high(RELOAD)
    mov TL0, low(RELOAD)
    setb TR0
    ;do stuff
    pop AR0
    pop ACC
    pop PSW
    reti ;Important, use reti, not ret!
```

5.4 Using a Timer as a Counter

Use the bits 0101 in TMOD instead of 0001. Rather than the timer counting on its own, it will count when there is a rising(?) edge on the counter input pin for the timer.

This can be used by setting TH0 and TL0 to 0, waiting, and reading the count value back after some time. To count more than 16 bits, set an interrupt to increment a third byte every time the timer overflows.

6 CV-8052 Hardware

Stuff you won't find in the MCS-51 Bible.

6.1 LEDs

The first eight of 10 LEDs are accessed through the LEDRA SFR, and the last two through LEDRB. The first LED is in the LSB, and a 1 turns it on. Only LEDRA is bit addressable!

Assembly

```
;Turn on LED 6 and turn off LED 4
```

```
setb LEDRA.6
clr LEDRA.4

;Turn on LED 9, keep previous state of LED 8.
;Can't do setb LEDRB.1 because LEDRB is not bit addressable
mov A, LEDRB
orl A, #0b10
mov LEDRB, A
```

6.2 7-Segment Displays

Write to the HEX0-5 SFRs, where a 1 turns *off* a segment. Since they are SFRs, you have to write to them one by one, and can't use indirect addressing with a loop. Best to keep a lookup table for the digits.

Assembly

```
lut_7seg:
    DB 0xC0, 0xF9, 0xA4, 0xB0, 0x99
    DB 0x92, 0x82, 0xF8, 0x80, 0x90
    DB 0x88, 0x83, 0xA7, 0xA1, 0x86
    DB 0x8E

    ;...
    mov A, x
    mov dptr, #lut_7seg
    mov A, @A+dptr
    mov HEX0, A
```

6.3 Keys and Switches

The four keys can be accessed in the KEY SFR (bit addressable). The first 8 switches are in SWA (bit addressable), and the last two switches are in SWB, which is not bit addressable. When a key is pressed, it reads as 0. When a switch is pushed towards the bottom of the board, it reads as 0.

For reading a single key or switch that is bit addressable, it is straightforward to use JB or JNB on the key or switch bit, such as JB SWA.0 or JNB KEY.3.

How to wait for a key to be pressed and released:

Assembly

```
    ;KEY reads as 1 when released
    ;$ jumps to the same instruction
    jb KEY.1, $
    ;when the program gets past the first instruction
    ;the key is pressed, wait for it to be released now
    jnb KEY.1, $
```

Here is an example of how to get the number of the first switch that is in the up position:

Assembly

```
    mov R0, #0xFF
```

```
    mov R1, SWA
    mov A, SWB
    ;clear extra bits in SWB and set one bit to ensure the loop stops later on
    anl A, #0b11
    orl A, #0b100
    mov R2, A
L_find_index:
    ;shift a bit out
    clr C
    mov A, R2
    rrc A
    mov R2, A
    mov A, R1
    rrc A
    mov R1, A
    inc R0
    ;The bit shifted out is in the carry flag.
    jnc L_find_index
L_done:
    ;the index (0 to 9) of the switch is in R0, or 10 if no switch is pressed.
```