University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Lecture 15: Microcomputer Integer Arithmetic II

Dr. Jesús Calviño-Fraga
Department of Electrical and Computer Engineering, UBC
Office: KAIS 3024
E-mail: jesusc@ece.ubc.ca
Phone: (604)-827-5387

March 16, 2017

Objectives

- Perform multiplication and division.
- Understand and use the ‘sjmp’ and ‘ljmp’ instructions.
- Understand and use the 8051 stack.
- Understand and use the ‘lcall’ and ‘ret’ instructions.
- Understand and use the ‘push’ and ‘pop’ instructions.

Multiplication

- Most modern processors include a ‘multiply’ instruction.
- Multiplication can be implemented using the shift/rotate instructions.
- To multiply 8 bits in the 8051 we can just use:

MUL AB

Multiplication of Binary Numbers

			1	0	0	1	Multiplicand = 9
			1	0	1	1	Multiplier = 11
			1	0	0	1	
		1	0	0	1		Partial products
	0	0	0	0			
1	0	0	1				
1	1	0	0	0	1	1	Final Product=99

It takes too much memory to store the partial products, so we add them as we go:

Multiplication of Binary Numbers

			1	0	0	1	← First partial product
		1	0	0	1		← Second partial product
		1	1	0	1	1	
	0	0	0	0			← Third partial product
	0	1	1	0	1	1	
1	0	0	1				← Fourth partial product
1	1	0	0	0	1	1	← Final Product=99

Example: 16-bit Multiply

Write a program that multiplies the two 16-bit numbers 'Num1' and 'Num2'. Store the 32-bit result in 'Result'. Use the rotate left and right instructions to complete the program. Do not use the assembly instruction ‘mul ab’.

In mat32.asm used in lab 6, uses a more efficient way of multiplying multi-byte numbers using the ‘mul ab’ instruction.

16-bit Multiply

```
; Mul16.asm: Multiplies 'Num1' by 'Num2'.  Stores
; 32-bit result in 'Result'.

$MOD52

org 0000H
      ljmp myprogram

DSEG at 30H
Num1:                 DS    2
Num2:                 DS    2
Result:               DS    4
PartProd:             DS    4
Mult:                 DS    2
```


16-bit Multiply

```
CSEG
myprogram:

      ; Load Num1 and Num2 with test values
      mov Num1, #low(300)
      mov Num1+1, #high(300)
      mov Num2, #low(20)
      mov Num2+1, #high(20)

      mov R3, #16 ; We have 16 partial products

      ;Copy the numbers to the work variables
      mov Mult,    Num1
      mov Mult+1, Num1+1
      mov PartProd,    Num2
      mov PartProd+1, Num2+1
      mov PartProd+2, #0
      mov PartProd+3, #0
```

16-bit Multiply

```
        ;Initialize result to zero
        clr a
        mov Result,   a
        mov Result+1, a
        mov Result+2, a
        mov Result+3, a

Mult16Loop:
        ;Shift the multiplicand right
        clr c
        mov a, Mult+1
        rrc a
        mov Mult+1, a
        mov a, Mult
        rrc a
        mov Mult, a
; Add the Partial product to the result only if carry is set
        jnc SkipAdd
```


16-bit Multiply

```
; Add the Partial product to the result
        mov R2, #4
        mov R0, #PartProd
        mov R1, #Result
        clr c
AL0:
        mov a, @R0
        addc a, @R1
        mov @R1, a
        inc R0
        inc R1
        djnz R2, AL0

SkipAdd:
        djnz R3, ShiftLeft
        sjmp forever
```

16-bit Multiply

```
; Shift the partial product left
ShiftLeft:
        clr c
        mov R1, #4
        mov R0, #PartProd
SL0:
        mov a, @R0
        rlc a
        mov @R0, a
        inc R0
        djnz R1, SL0
        sjmp Mult16Loop

; Done!  Loop forever
forever:
        sjmp forever
END
```


Division

- Division (efficient division) is the most complicated of the arithmetic operations.
- The 8051 includes a function to divide 8-bit by 8-bit unsigned integers:

  'div ab' divides register A by register B. A holds the quotient, B holds the remainder.
- Dividing bigger numbers requires the implementation of a multi-byte division algorithm:

n-bit division

- To perform division efficiently we can use repeated subtraction with the recursive formula:

$$P_{j+1} = P_j \times R - q_{n-(j+1)} D$$

P is the remainder
R is the radix
q_m is the digit at position m
D is the denominator

n-bit integer division algorithm

- If the denominator is zero finish with error!
- Shift the denominator left until the most significant digit (for a given radix) is different from zero. Make *n*=number_of_shifts +1.
- Repeat *n* times the recurrence formula:

$$P_{j+1} = (P_j - q_{n-(j+1)} D) \times R$$

If you repeat more than n times you'll get the fractions of the division.

Division Example 1

- Solve 3391 ÷ 17 using the algorithm from the previous slide. Used radix 10.

$$\frac{3391}{17} \rightarrow \frac{3391}{1700}, n = 3, R = 10$$

$$P = (3391 - (1) \times 1700) \times 10 = 16910 \rightarrow q_2 = 1$$

$$P = (16910 - (9) \times 1700) \times 10 = 16100 \rightarrow q_1 = 9$$

$$P = (16100 - (9) \times 1700) \times 10 = 8000 \rightarrow q_0 = 9$$

Answer is: 199

Division Example

- Check the library "Math32.asm" used in Lab 6. The function "div32" implements the algorithm above.
- See if you can follow what "div32" does!

sjmp (short jump)

- Requires two bytes: Opcode (80H) and 8-bit operand.
- The jump address is an “offset” to the next instruction.
- The 8-bit operand is a two-complements signed number. We can jump forward 127 bytes or back 128 bytes counting from the instruction after the sjmp.
- All conditional jumps behave just like sjmp.
- Also know as a ‘relative’ jump.

sjmp examples

```
001E        L1:
001E 00          nop
001F 00          nop
0020 00          nop
0021 00          nop
0022 8006        sjmp L2
0024 80F8        sjmp L1
0026 00          nop
0027 00          nop
0028 00          nop
0029 00          nop
002A        L2:
```

24H+06H=2AH

0F8H → 07+1 → -8:

26H-8H=1EH

Conditional Jumps

Mnemonic	Opcode	B	C	Function
JZ rel	60H	2	2/3	(PC) = (PC) + 2 IF (A) = 0 THEN (PC) = (PC) + rel
JNZ rel	70H	2	2/3	(PC) = (PC) + 2 IF (A) ≠ 0 THEN (PC) = (PC) + rel
JC rel	40H	2	2/3	(PC) = (PC) + 2 IF (C) = 1 THEN (PC) = (PC) + rel
JNC rel	50H	2	2/3	(PC) = (PC) + 2 IF (C) ≠0 THEN (PC) = (PC) + rel
JB bit, rel	20H	3	3/4	(PC) = (PC) + 3 IF (bit) = 1 THEN (PC) = (PC) + rel
JNB bit, rel	30H	3	3/4	(PC) = (PC) + 3 IF (bit) = 0 THEN (PC) = (PC) + rel
JBC bit, rel	10H	3	3/4	(PC) = (PC) + 3 IF (bit) = 1 THEN (bit) = 0 and (PC) = (PC) + rel

Conditional Jumps

WARNING: These change the carry flag!

Mnemonic	Opcode	B	C	Function
CJNE A, direct, rel	B5H	3	3/4	Compare and jump if not equal rel
CJNE A, #data, rel	B4H	3	3/4	Compare and jump if not equal rel
CJNE Rn, #data, rel	B8H-BFH	3	3/4	Compare and jump if not equal rel
CJNE @Ri, #data, rel	B6H-B7H	3	3/4	Compare and jump if not equal rel
DJNZ Rn, rel	D8H-DFH	3	2/3	Decrement and Jump if not zero
DJNZ direct,rel	D5H	3	3/4	Decrement and Jump if not zero

ljmp (long jump)

- Can jump anywhere in the 64k code memory space.
- Requires three bytes: opcode (02H) + 16-bit address:

```
0000 02001E     ljmp myprogram
001B 021803     ljmp 1803H
```


The 8051 stack

- We need the stack to use the lcall instruction.
- The stack is an area of memory where variables can be stacked. It is a LIFO memory: the last variable you put in is the first variable that comes out.
- Special Function Register SP (stack pointer) points to the beginning of the stack. SP in the 8051 is incremented **before** it is used (for push), or used and them decremented (for pop).
- After reset, SP is set to 07H. If you have variables in internal RAM, any usage of the stack is likely to corrupt them. Solution: at the beginning of your program set the SP special function register so it points to free memory:

```
mov SP, #7FH ; Set the stack pointer to idata start
```

‘lcall’ and ‘ret’ instructions

- The ‘lcall’ instructions pushes the address of the next instruction (16-bit, LSB first) into the stack and jumps to the address passed as an operand to the ‘lcall’ instruction.
- The ret instruction pops the address stored in the stack and then jumps to that address.
- The ‘lcall’ can call any address in the 64k code memory space. Works similarly to ‘ljmp’…

Push and Pop

- When the microcontroller executes a push into the stack it:
  a) Increments the SP.
  b) Saves the value in the internal RAM location pointed by the SP.
- When the microcontroller execute a pop from the stack it:
  a) Retrieves the value from the internal RAM location pointed by the SP.
  b) Decrements the SP.
- As you may have suspected, the 8051 (as well as most other microprocessors!) have push and pop instructions.

lcall example

```
; Blinky.asm: blinks an LED connected to LEDR0
$MODDE0CV

org 0000H
    ljmp myprogram

;The clock in the CV-8052 is 33.3333MHz. (1 cycle=30ns)
WaitHalfSec:
    mov R2, #90
L3: mov R1, #250
L2: mov R0, #250
L1: djnz R0, L1 ; 3 machine cycles-> 3*30ns*250=22.5us
    djnz R1, L2 ; 22.5us*250=5.625ms
    djnz R2, L3 ; 5.625ms*90=0.506s (give or take)
    ret
```

Subroutine

lcall example

```
myprogram:
    mov SP, #7FH
    ; Turn off all LEDs...
    mov LEDRA, #0
    mov LEDRB, #0
M0:
    cpl LEDRA.0
    lcall WaitHalfSec
    sjmp M0
END
```

Saving and Restoring Registers to/from the Stack using push and pop

- Before using the stack (lcall, push, pop) make sure you set the SP special function register.
- Popular registers to push/pop: ACC, DPL, DPH, PSW, R0 to R7.
- Pop registers from the stack in the **REVERSE** order you pushed them! Remember the stack is a LIFO.

Push and Pop Example

```
WasteTime:
    push Acc
    push B
    push dpl
    mov Acc, #100
L3: mov B, #100
L2: mov dpl, #100
L1: djnz dpl, L1
    djnz B, L2
    djnz Acc, L3
    pop dpl
    pop B
    pop Acc
    ret
```

Common bug!

```
WasteTime:
     push B
     push Acc
     push dpl
     mov Acc, #100
L3:  mov B, #100
L2:  mov dpl, #100
L1:  djnz dpl, L1 ; 3 bytes, 2 machine cycles
     djnz B, L2
     djnz Acc, L3
     pop dpl
     pop B
     pop Acc
     ret
```

Where is it?

pops are in the wrong order!

Push/Pop for R0 to R7

```
WaitHalfSec:
     push AR0
     push AR1
     push AR2
     mov R2, #20
L3:  mov R1, #250
L2:  mov R0, #184
L1:  djnz R0, L1 ; 2 machine cycles-> 2*0.27126us*184=100us
     djnz R1, L2 ; 100us*250=0.025s
     djnz R2, L3 ; 0.025s*20=0.5s
     pop AR2
     pop AR1
     pop AR0
     ret
```

The extra 'A' is for 'direct address' of register R0. This is only required for registers R0 to R7

Stack Trace Example

For the program shown write down the stack values in the space provided when the execution reaches the indicated point in code.

Note: 'stack trace' makes good exam questions!

```
1000            myprogram:
1000 75813F         mov SP, #3FH
1003 7410           mov a, #10H
1005 C0E0           push acc
1007 75F0F0         mov b, #0F0H
100A 12100F         call a_x_b_plus1
100D            forever:
100D 80FE           jmp forever
100F           a_x_b_plus1:
100F C0E0           push acc
1011 C0F0           push b
1013 A4             mul ab
1014 04             inc a ; ← HERE!!!!
1015 D0F0           pop b
1017 D0E0           pop acc
1019 22             ret
```

Address	40H	41H	42H	43H	44H	45H	46H	47H
Value								

Stack Trace Example

```
1000            myprogram:
1000 75813F         mov SP, #3FH
1003 7410           mov a, #10H
1005 C0E0           push acc ←
1007 75F0F0         mov b, #0F0H
100A 12100F         call a_x_b_plus1 ←        2 bytes!
100D            forever:
100D 80FE           jmp forever
100F           a_x_b_plus1:
100F C0E0           push acc ←
1011 C0F0           push b ←
1013 A4             mul ab
1014 04             inc a ; ← HERE!!!!
1015 D0F0           pop b
1017 D0E0           pop acc
1019 22             ret
```

Address	40H	41H	42H	43H	44H	45H	46H	47H
Value	??	??	??	??	??	??	??	??

Using a debugger (cmon51):

```
P89LPC9351> r
PC=0000 A=00 PSW=00 B=00 IE=00 DPL=00 DPH=00 SP=07 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
0000: 02 10 00  ljmp    1000
P89LPC9351> d 40 10
D:40:  00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  ................
P89LPC9351> s
PC=1000 A=00 PSW=00 B=00 IE=08 DPL=00 DPH=00 SP=07 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1000: 75 81 3f  mov     SP,#3f
P89LPC9351> s
PC=1003 A=00 PSW=00 B=00 IE=08 DPL=00 DPH=00 SP=3f REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1003: 74 10     mov     a,#10
P89LPC9351> s
PC=1005 A=10 PSW=01 B=00 IE=08 DPL=00 DPH=00 SP=3f REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1005: c0 e0     push    ACC
P89LPC9351> s
PC=1007 A=10 PSW=01 B=00 IE=08 DPL=00 DPH=00 SP=40 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1007: 75 f0 f0  mov     B,#f0
```


Using a debugger (cmon51):

```
P89LPC9351> s
PC=100a A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=40 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
100a: 12 10 0f  lcall   100f
P89LPC9351> s
PC=100f A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=42 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
100f: c0 e0     push    ACC
P89LPC9351> s
PC=1011 A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=43 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1011: c0 f0     push    B
P89LPC9351> s
PC=1013 A=10 PSW=01 B=f0 IE=08 DPL=00 DPH=00 SP=44 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1013: a4        mul     ab
P89LPC9351> s
PC=1014 A=00 PSW=04 B=0f IE=08 DPL=00 DPH=00 SP=44 REGBANK:0
R0=ff R1=f7 R2=ff R3=fd R4=58 R5=e6 R6=ff R7=6b
1014: 04        inc     a
P89LPC9351> d 40 10
D:40:  10 0d 10 10 f0 14 10 04 : 00 00 00 00 00 00 00 00  ................
```

Answer!

Exercises

- Write an assembly program to multiply the 24-bit binary number stored in registers R2, R1, R0 (R0 is the least significant byte) by 10 (decimal). Save the result in R3, R2, R1, R0. Use the MUL AB instruction.
- Write an assembly program to find the approximate square root of a 16-bit number stored in registers DPH and DPL. Save the result to register R7. Tip: Use a binary search algorithm to find the square root quickly!

Exercises

- A common way of passing parameters to a function is via the stack. Modify the function **WaitHalfSec** so that it receives the number of half-seconds to wait in the stack. (Note: this problem is not as trivial as it sounds. You may need to increment and/or decrement register SP to solve this problem)
- Most C programs pass parameters to functions via the stack. Also C programs use the stack to allocate automatic variables (local variables defined within the function). This works fine most of the time, but sometimes a condition commonly known as "stack overflow" occurs. Explain what causes "stack overflow".