Comp 255Q - 1M: Computer Organization Name: _____
Lab #4 - PDP-8 Assembler Language Programming
February 2, 2010 Grade _____/10

**Introduction**: We'll begin this lab with a brief overview of Assembler Language and Assembler Language programming. We'll begin with the Add01.pal program found on page 4.1 of the PDP-8 Emulators User's Manual.

1: Using the Built-in editor to create PAL source code file

2: Elements of PDP-8 Assembler: Instructions, Expressions, Allocating Storage, Directives

Notes:

3: Assembling and running programs thru the Debug screen

Notes:

4: A Small Set of PDP-8 Instructions – PDP-8 Lite (see page 4.9)

5: .LST Files, Non-Relocatable Code, Using Notepad to Print .PAL files

Notes:

6: Creating a PDP-8 template file

```
/
/   Name: <your name here>
/   File:
/   Date:
/
/   Desc:
/
/-------------------------------------------
/
/ Code Section
/
*0200                   / Code starts at Address 0200
Main, cla cll           / Clear AC and Link

/
/ Data Section
/

$Main
```

Save the file on your h: drive as **template.pal**.

Note: For all pledges assignments the *pledge statement must be included in the header comment block and signed* (see below). Since labs are not pledged, the pledge statement was removed

```
/
/   I affirm that my work upholds the highest standards of honesty and
/   academic integrity at Wittenberg and that I have neither given nor
/   received unauthorized assistance.
/
/   _____________________________________________
```

Any ASCII text editor (Notepad, MS-Edit or the PDP-8 Editor) can be used to create/edit a PAL source code file. Be aware that if your use a text editor like Notepad to create your source code files, avoid using the tab key since the PDP-8 Editor does not handle tab characters well; they display as an odd-looking "○" character. The assembler has no problems with them but the editor does.

**Part 1:** As a class exercise create, assemble and execute the multiplication program below. This is the same multiplication program from Lab03.

```
/
/  Name:
/  File: lab04a.pal
/  Date: February 2, 2010
/
/  Desc: A program to multiply two integers: P = N * M
/
/-----------------------------------------------------------------
/
/ Code Segment
/
*0200                   / start at address 0200
Main,   cla cll         / clear AC and Link
        dca P           / initialize product to 0
        tad N           / load N
        cia             / negate it
        dca CNT         / store in loop counter
L1,     tad M           / add multiplicand
        isz CNT         / increment loop counter
        jmp L1          /    and loop unless 0
        dca P           / store product
        hlt             / halt
        jmp Main        / return to beginning
/
/  Data Segment
/
*0300
CNT,    0               / loop counter
N,      3               / multiplier
M,      7               / multiplicand
P,      0               / product
$Main
```

Start the PDP-8 simulator program and do the following

1. From the Edit Window read in the file `template.pal`. Enter the code given above.

2. Assemble the program. If successful *save* the list file as h:\comp255\lab04a.lst; otherwise go back and fix any syntax errors. Normally it is *not necessary* to save the list file but in this case do so. Since the PDP-8 simulator program does not have a print capability you will need to print out this file using another text editor like Notepad. You can do this later.

   If the assemble is successful save the source code file as h:\comp255\lab04a.pal. You can print out this file later using Notepad. Normally you *would save* the source code file.

3. Since your program does not do I/0 you will have to run the program from the Debug Screen to see any results. Note that when a program is successfully assembled, entering the Debug Screen will take you to the page where the code begins.

   Test that your program runs correctly

4. If your program runs correctly use Notepad to print out the list (.lst) and source code (.pal) files; otherwise go back to the Editor and fix your errors. Hand in the list and source code print outs with this lab assignment.

**Verification #1: _____**

**Question #1**: What happens if the multiplier N is 0? How could you detect this condition and "fix" the program?

**Question #2**: Which is the more efficient calculation? 3 * 7 where the multiplier is 3 or 7 * 3 where the multiplier is 7? How could you make the program more efficient?

**Part 2** - Write a PDP-8 program to divide using repeated subtraction. Given two integers A and B your program should compute the quotient Q = A/B and the remainder R = A%B. Here is the pseudo-code you will need; all you need to do is translate it into PDP-8 assembler and get it to execute correctly. Use the fact that the ISZ instruction can also be used to increment a variable.

```
Main,   Clear accumulator and link
        Deposit to Q (and clear accumulator)
        Deposit to R (and clear accumulator)
        Load B
        Negate it
        Deposit result to minusB (and clear accumulator)
        Load A
L1,     Add minusB                          /   the L1 loop is where division occurs
        Skip next instruction if positive (or zero)  /   you exit the loop when you go negative
        Jump to L2
        Increment Q                         /   counting the number of subtractions!
        Jump to L1
L2,     Add B to restore the remainder      /   since you "overshot" you need to
        Deposit to R (and clear accumulator)  /   restore the dividend which is now the reminder
        Halt
        Jump to Main
```

There are 5 variables: A and B which should be initialized to non-zero values for the dividend and divisor respectively and Q, R, and MB (minus B), the latter being used to store –B. This is needed for subtraction. The variables A, B, Q, R, and MB should be stored *in order* at beginning at address 0300. See below

```
/
/  Data Segment
/
*0300
A,    13d             / dividend
B,    5               / divisor
Q,    0               / quotient
R,    0               / reminder
MB,   0               / minus B
$Main
```

Begin with the template.pal file. Follow the steps from Part 1 but save the source code file as lab04b.pal; you do not need to create a list file. When your program runs correctly have the instructor verify your work. Print out your source code file to be handed in with the lab assignment.

**Verification #2: _____**

**Part 3**: Extracting Octal Digits: We begin by introducing four more instructions

| **Assembler Mnemonic** | **Machine Code** | **Description** |
|---|---|---|
| AND | 0nnn | Logical AND<br>C(AC)<- C(AC) AND C(Eaddr) |
| IAC | 7001 | Increment Accumulator<br>C(AC) <- C(AC) + 1 |
| RAL | 7004 | Rotate Accumulator and link Left |
| SZL | 7430 | Skip on Zero Link<br>If C(L) = 0 Then<br>C(PC) <- C(PC) + 1 |

The **RAL** instruction (an op-code 7 instruction) does a Rotate Accumulator Left *through the Link bit*. That is, accumulator bit 11 is shifted left to bit 10, bit 10 is shifted left to bit 9 etc all the way to bit 1 which is shifted to bit 0. Bit 0 is shifted to the Link bit and the Link bit is shifted to bit 11 of the accumulator.

Rotate operations often used for just rotating the bits *within* a word. Thus the PDP-8's RAL instruction which includes the Link bit in the rotation is a bit problematic. However there is a workaround; the code below performs a sort of *Rotate Left of the Accumulator only*. That is, bit 0 of the Accumulator is moved to bit 11 of the Accumulator instead of to the Link bit.

```
cll ral     / clear link and rotate left
szl         / skip on zero link
iac         / otherwise add 1 to AC
```

The code above clears the link bit and does a rotate left which moves a 0 into bit 11 (since `cll` and `ral` are both Group One Microinstructions they can be combined). The Skip on Zero Link (**SZL**) instruction is used to test the Link bit (which contains the former contents of bit 0). If the Link bit is 0, it skips the next instruction; otherwise it adds 1 to the Accumulator which has the effect of setting bit 11 to 1.

If we execute this code three times, we rotate the left most three bits of the accumulator into the right most three bits of the accumulator. If we then **AND** (opcode 0) the accumulator with a memory address containing the octal value 0007 (called a *mask*), we have effectively extracted the left most three bits of memory by zeroing out the upper nine.

Important! Save the rotated number *before* you mask out the upper three digits by using the instruction pair

```
dca Number  / Save AC
tad Number  / Restore AC
```

This save & restore accumulator must be done if we should want to extract the other octal digits; otherwise the masking operation will erase the upper three digits (nine bits) before they can be extracted.

Write a program to take any 4 octal digit integer and extract the least significant octal digit to a second memory location called D0. You will have to load the integer into the accumulator, rotate it 3 times left, mask out the upper 9 bits (be sure to save the rotated integer) and store what's left at the second location in memory.

Use the following Data Section

```
/
/   Data Section
/
*0300
D0,   0
Number, 3527      / number to be unpacked
Mask, 0007        / mask for AND operation
```

Begin with the template.pal file. Save the source code file as lab04c.pal; you do not need to create a list file. When your program runs correctly have the instructor verify your work. Print out your source code file to be handed in with the lab assignment.

**Verification #3: _____**

**And Beyond**: Repeating this same sequence of code three more times (rotate left three, save & restore, then **AND** with the *value* 0007) will allow us to extract all four octal digits; so 3527 is "unpacked" to obtain 3, 5, 2, and 7. This is why you *must* save & restore the rotated results *before* you AND with 0007; otherwise you will lose the other 3 digits.

Extracting each octal digit from Number is quite useful since if you add 60 octal to each of the four octal digits extracted, you would have converted each to the ASCII code for the corresponding digit. Then if you printed out each digit, you would have printed out the value of Number!

**Note:** This sets the stage for doing I/O, array-type data structures, and program modularization (subroutines).

**Part 4** - Self Modifying Code: Since much of the code for extracting the digits is repeated, can you modify the code to use a loop to extract the four digits? Going thru the code four times is done using a counting loop that executes four times, something like the following

```
        cla cll     / Clear AC and link
        tad C4      / Load C4; C4 contains 4
        cia         / Negate it
        dca CNT     /    and deposit it to the loop counter
L1,             / Body of loop begin here: load Number
                / left rotate 3 times
                / save & restore
                / mask out upper 9 bits
                / deposit digit to memory
        isz CNT     / Increment loop counter
        jmp L1      /    and loop if not zero
```

Your data segment is augmented to include additional variables for control your counting loop as well as four variable to hold the extracted digits.

```
*0300
D0,   0
D1,   0
D2,   0
D3,   0
Number, 3527      / number to be unpacked
Mask, 0007        / mask for AND operation
C4,   4           / constant 4
CNT,  0           / loop counter
```

There is still an addressing problem. Inside the loop how do you change the address of D0 to be D1, then D1 to be D2 etc.? That is, inside the body of the loop, how does the deposit instruction

```
        dca D0
```

become

```
        dca D1
```

One answer is to use self-modifying code. Attach a label to the dca DO instruction and insert the instruction isz `A` after the `A, dca D0` instruction

```
A,    dca D0
      isz A
```

This will solve the problem – but why? Think about it. Now write the program and test that this works.

There is one problem (?) with self-modifying code is that you can't re-run it since it changes itself! This can be fixed by first saving the instruction that's modified then restoring it afterward. So allocate a temporary storage and before you enter the main loop

```
cla cll
tad A       / get the instruction at A
dca temp    /   and store it at temp
```

Afterward restore the instruction

```
cla cll
tad temp    / restore instruction
dca A       /   at Loop
```

Still there must be a better way!