



## ISTD 50.002 Computation Structures

# Lab #6 Assembly Language

[Introduction](#)

[Due Date](#)

[Making Moo](#)

[Moo Scoring Subroutine \(6 points\)](#)

[An Approach in C](#)

[Testing your Moo](#)

[Implementation Notes](#)

Revamped by: Kenny Choo, Natalie Agus, Oka Kurniawan 2020.

---

## Introduction

In this lab you will have the opportunity to write your first Beta assembly language program. Your task is to write a scoring subroutine for the game of “Moo”, a numeric version of [Mastermind®](#).

In **Moo**, you try to *guess the secret 4-digit number*.

- Each guess is scored with a count of “bulls” and “cows”.
- Each “bull” means that one of the digits in the guess matches both the value and position of a digit in the secret number.
- Each “cow” is a correctly guessed digit but its position in the guess doesn’t match the position in the secret.
- Once a digit in the secret has been used to score a digit in the guess it won’t be used in the scoring for other digits in the guess.
- The count of bulls should be determined before scoring any cows.

Here is an example:

Secret word: <b>1234</b>	Guess: 1379	Bulls=1, Cows=1
	Guess: 4321	Bulls=0, Cows=4
	Guess: 1344	Bulls=2, Cows=1
	Guess: 1234	Bulls=4, Cows=0 (game ends!)

In addition to this handout, there are some other useful documents that will help you:

- [BSim documentation](#). Describes how to use BSim, our Beta simulator with built-in assembler. Includes a brief introduction to the syntax and structure of Beta assembly language programs.
- [Beta Documentation](#): A detailed description of each instruction in the Beta instruction set. Also documents our convention for subroutine entry and exit sequences.
- [Beta ISA Summary of Instruction Formats](#): A one-page quick reference for Beta instructions.
- Notes on [Stack and Procedures](#) to refresh your memory on how to perform procedure linkage.

## Due Date

Refer to the [course handout](#) for due date.

## Making Moo

### Moo Scoring Subroutine (6 points)

Your subroutine should take two arguments—the secret word and the test word—and return an integer encoding the number of bulls and cows as  $(16 \times \text{bulls}) + \text{cows}$ . The secret and test words contain four 4-bit digits packed into the low-order 16 bits of the word. For example, if one of the words was 1234, it would be encoded as 0x1234 where “0x” indicates hexadecimal (base 16) notation. Even though 4 bits are used to encode each digit, the words will only contain the digits 0 through 9.

### An Approach in C

You are welcome to compute the score however you would like. In case you would like a head start, here is one approach, written in C:

```
// Test two MOO words, report Bulls & Cows...
// Each word contains four 4-bit digits, packed into low order.
// Each digit ranges from 0 to 9.
// Returns a word whose two low-order 4-bit digits are Bulls & Cows.

int count_bull_cows(int a, int b) {
    int bulls;           // number of bulls
    int cows;           // number of cows
    int i, j, btemp, atry, btry, mask; //temp vars

    // Compute Bulls: check each of the four 4-bit digits in turn
    bulls = 0;
    mask = 0xF;          // mask chooses which 4-bit digit we check
    for (i = 0; i < 4; i = i + 1) {
        // if the 4-bit digits match, we have a bull
        if ((a & mask) == (b & mask)) {
            bulls = bulls + 1;
        }
    }
}
```

```

    // turn matching 4-bit digits to 0xF so we don't
    // count them again when computing number of cows
    a = a | mask;
    b = b | mask;
}
// shift mask to check next 4-bit digit
mask = mask << 4;
}

// Compute Cows: check each non-0xF digit of A against all the
// non-0xF digits of B to see if we have a match
cows = 0;
for (i = 0; i < 4; i = i + 1) {
    atry = a & 0xF;          // this is the next digit from A
    a = a >> 4;              // next time around check the next digit
    if (atry != 0xF) {       // if this digit wasn't a bull
        // check the A digit against each of the four B digits
        btemp = b;           // make a copy of the B digits
        mask = 0xF;          // mask chooses which 4-bit digit we check
        for (j = 0; j < 4; j = j + 1) {
            btry = btemp & 0xF; // this is the next digit from B
            btemp = btemp >> 4; // next time around check the next digit
            if (btry == atry) { // if the digits match, we've found a cow
                cows = cows + 1;
                b = b | mask;    // remember that we matched this B digit
                break;           // move on to next A digit
            }
        }
        mask = mask << 4;
    }
}
}

// encode result and return to caller
return (bulls << 4) + cows;
}

```

## Testing your Moo

The test jig uses our usual convention for subroutine calls: the two arguments are pushed on the stack in reverse order (i.e., the first argument is the last one pushed on the stack) and control is transferred to the beginning of the subroutine, leaving the return address in register LP. The result should be returned in R0.

Your code should use the following template. **Be sure to include the last two lines** since they allocate space for the stack used by the test jig when calling your program.

```

| include instruction macros and test jig
.include /50002/beta.uasm
.include /50002/lab6checkoff.uasm

```

```




count_bull_cows:      | your subroutine must have this name
    PUSH(LP)          | standard subroutine entry sequence
    PUSH(BP)
    MOVE(SP, BP)
    ... PUSH any registers (besides R0) used by your code ...

    ... your code here, leave score in R0 ...

    ... POP saved registers ...
    MOVE(BP,SP)       | standard subroutine exit sequence
    POP(BP)
    POP(LP)
    RTN()

StackBase: LONG(.+4)  | Pointer to bottom of stack
. = .+0x1000          | Reserve space for stack...


```

Using BSim, assemble your subroutine using the  tool. If the assembly completes without errors, BSim will bring up the display window and you can execute the test jig (which will call your subroutine) using the run  or single-step  tools. The test jig will try 32 different test values and type out any error messages on the tty console at the bottom of the display window. Successful execution will result in the following printout:

```

Checking count_bull_cows:
.....
Your count_bull_cows routine passes all our tests - congrats!

```

When your subroutine passes the tests, you can complete the on-line check-in using the  tool.

## Implementation Notes

1. If you want to examine the execution state of the Beta at a particular point in your program, insert the assembly directive `".breakpoint"` at the point where you want the simulation to halt. You can restart your program by clicking the run button, or you can click single-step button to step through your program instruction-by-instruction. You can insert as many `.breakpoints` in your program as you would like.
2. If your subroutine uses registers other than R0, remember that they have to be restored to their original values before returning to the caller. The usual technique is to PUSH their value onto the stack right after the instructions of the entry sequence and POP those values back into the registers just before starting the exit sequence.

3. Assuming you have used the subroutine entry sequence shown above, the first argument can be loaded into a register using the instruction LD(BP,-12,Rx). Similarly the second argument can be loaded using LD(BP,-16,Ry).

One way to tackle the assignment is to “hand compile” the C implementation shown above using some techniques shown in the [notes](#):

4. Allocate a register to hold each of the variables in the C code. For example, reserve R0 and R1 for temporary values, load “a” into R2, “b” into R3, assign “bulls” to R4, etc. You will eliminate a lot of LDs and STs by keeping your variables in registers instead of in memory locations on the stack.
5. The instruction macro CMOVE(constant,Rx) is useful for loading small numeric constants into a register. For example, assuming that the variable “mask” has been assigned to R11, the C statement “mask = 0xF;” can be implemented in a single instruction: CMOVE(0xF,R11).
6. The CMP instructions and BEQ/BNE are useful for compiling C “if” statements. For example, assuming atry has been assigned to R7, the C fragment:

```
if (atry != 0xF) { statements... }
```

can be compiled into the following instruction sequence:

```
CMPEQC(R7,0xF,R0) | R0 is 1 if atry==0xF, 0 otherwise
BNE(R0,endif27)   | so branch if R0 is not zero
... code for statements ...
endif27:          | need a unique label for each if
...
```

7. Here is the template for compiling the a “for” statement. Note that the body of the loop is executed as long as the tests are true.

```
for (inits; tests; afters) { body... }
```

expands into the following:

```
... code for inits ...
BR(endifor32)
for32:
... code for body ...
... code for afters ...
endifor32:
... code for tests, Rx is 1 if tests are true
...
BNE(Rx,for32)
```

8. A brief summary of C operators:

=	assignment
==	equality test (use CMPEQ, CMPEQC)
!=	inequality test (use CMPEQ, CMPEQC, reverse sense of branch)
<	less than (use CMPLT, CMPLTC)
<<	left shift (use SHL)
>>	right shift (use SRA)
&	bit-wise logical and (use AND, ANDC)
	bit-wise logical or (use OR, ORC)
+	addition (doh!, use ADD, ADDC)