## What is TINY?

TINY is a program that runs on a host machine and simulates a simple but operational computer. The design is intended to allow one to experiment with the concepts of assembly language programming without being overwhelmed with syntax, binary arithmetic, and a complex architecture. The TINY computer is a 'base-10' machine that contains 1,000 bytes of memory and 5 registers. There are 21 machine instructions that provide the necessary structure for most demonstration programs. The main focus while studying TINY should be to understand the basic architectural concepts that are common to all computers of the 'von Neumann' class. Although large mainframes are more complex and enjoy a much richer instruction set, the basic concepts of machine architecture are much the same as those used by TINY.

## TINY Architecture

Figure 1 is a schematic representation of TINY's architecture. The smallest addressable unit of storage in the primary memory device is called a byte. The size of each byte in TINY's primary storage is sufficient to store 5 decimal digits and a plus (+) or minus (-) sign. (Although TINY's bytes are not the same size as the bytes in other computers, it is true that for most computers a byte is the smallest addressable unit of storage.) This means that each addressable byte of memory can contain a number between -99,999 and +99,999. You should also note that the addresses of these bytes consist of three decimal digits and range from 000 to 999.

The Data Bus is the communication link between memory and the CPU. One byte can be transmitted via the Data Bus at a time. Notice that this bus is bi-directional. The CPU can both read the contents of a memory location and store a new value at a particular location. However, the two operations cannot be performed simultaneously.

The Address Bus is used to specify which of the 1,000 memory bytes should be referenced. Thus, if it is desired to store the number +12345 at address 056, the address 056 is first placed on the Address Bus and then the number +12345 is placed on the Data Bus. The Address Bus may be thought of as a pointer or selector for a particular memory cell.

The TINY CPU is about as simple as a CPU can be; and, yet; it still allows one to write complete programs. The CPU consists of two sections (CU = Control Unit and ALU = Arithmetic/Logic Unit) and 5 registers:

|  |  |
|---|---|
| 1. Program Counter (PC) | - contains address of next instruction (3 decimal digits) |
| 2. Instruction Register (IR) | - contains current instruction (5 decimal digits) |
| 3. Accumulator (ACC) | - general purpose register used by all arithmetic and logic instructions (5 digits plus sign) |
| 4. Stack Pointer (SP) | - contains the address of the top of the system stack used by all function calls (3 decimal digits) |
| 5. Base Pointer (BP) | - contains the address of the bottom of the current function's stack frame. (3 decimal digits) |

The CPU also contains connections to the Input and Output devices. Input is received from the keyboard, and output is sent to the screen / monitor. These connections to the outside world are linked to the ACC register. That is, all data coming in from the keyboard is placed in the ACC register; and all data going out to the terminal screen must first be placed in the ACC register. TINY has no other input or output devices.

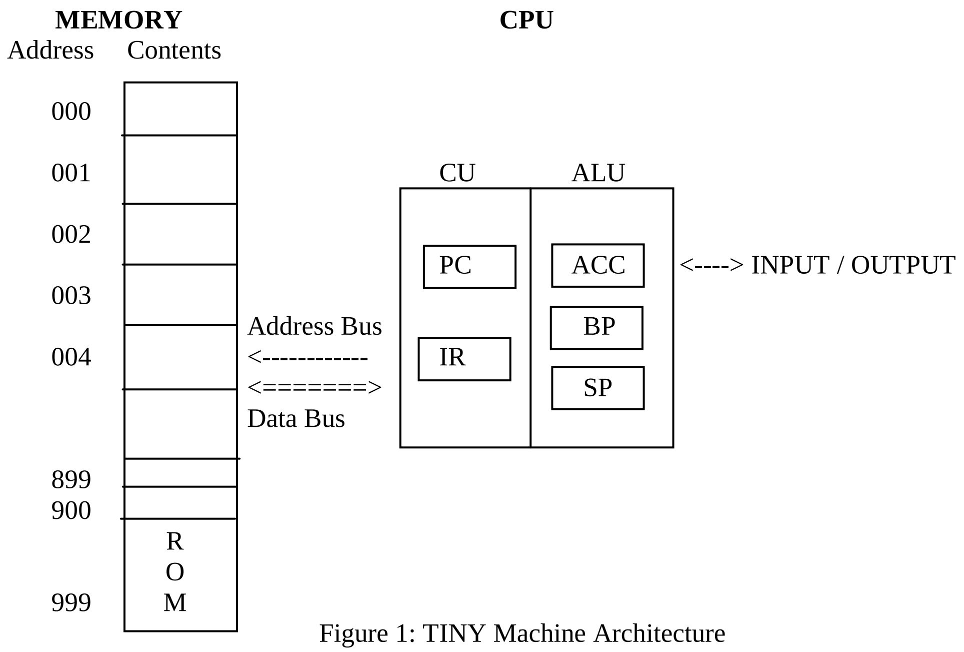


Figure 1: TINY Machine Architecture

## TINY Instruction Cycle

When a TINY program is first loaded for execution, the instructions are loaded into memory and the PC is set to 000. The CPU then initiates an instruction cycle. The TINY instruction cycle consists of 3 steps that are repeated in an endless loop.

1. Fetch next instruction as indicated by the PC and load it into the IR. ( IR $\leftarrow$ c(PC) )
2. Increment the PC by 1. ( PC $\leftarrow$ PC + 1 )
3. Decode and Execute the instruction. (See Appendix A)

**3 Steps of the TINY Instruction Cycle**

There are several conclusions we can draw from this instruction cycle.

First, the CPU makes the assumption that the PC is pointing to an instruction. If, due to some mistake, the PC is not pointing to an instruction, the CPU will blindly load the data and attempt to decode and execute the "next instruction". If by chance the data is equivalent to some valid instruction, the CPU will carry out the bogus instruction, which may lead to unpredictable results. As far as the CPU is concerned, there is no inherent difference between an instruction and data. It is up to the programmer to insure that the Program Counter references only valid instructions.

A second conclusion that can be gleaned from considering the instruction cycle is that, if not altered by some instruction, the CPU will attempt to execute an instruction from each byte of memory in sequence from address 000 to 999. Thus, the CPU assumes that the instructions are arranged in a sequence. If the programmer needs to program a loop or branch, a special instruction will be needed.

A third conclusion from the TINY instruction cycle is that all TINY programs must begin at address 000. This is one of the limitations of the TINY architecture. No provision has been made to allow you to load a program at an arbitrary address and then set the PC accordingly. All TINY programs must be loaded with the first instruction stored at address 000.

Finally, unless a STOP instruction is used, the CPU will execute forever (or at least until an invalid instruction is fetched or the PC points to an invalid address).

## TINY Machine Instructions

A TINY machine instruction consists of 5 decimal digits that are grouped into two parts - the operation code (also known as the opcode) and the operand. The first 2 digits of an instruction are used to represent the opcode, while the last 3 digits are used for the operand. The opcode is a number that defines the instruction that should be carried out. For instance the opcode for a load instruction (which moves data from memory to the ACC register) is the number 01. The operand supplies a value that will be needed by the instruction. For example, if a programmer wanted to load a number from memory address 567, the instruction needed is 01567.

The opcodes for all of the TINY instructions are listed in Appendix A. Using Appendix A, what is accomplished by the following program segment?

```
01567
06567
04567
00000
```

## TINY Machine Code & Assembly Language

The letter codes for each opcode are called mnemonics. A mnemonic is a word or pseudo-word used as a memory aid. It is easier for programmers to remember that MUL is the multiply instruction than it is to remember to use opcode number 08. It is this use of mnemonics that is the major difference between machine language and assembly language. Assembly language uses mnemonic instructions that have to be translated by a program called an assembler into the actual machine code that is used by the CPU. Such an assembler program is available for TINY. Let's consider some of the instructions individually.

As we learned while looking at the instruction cycle, the STOP instruction is very important. Without it, the CPU will continue to fetch and execute instructions even if the values fetched are actually data. The STOP instruction ignores the value stored in the operand portion of the machine instruction. Thus, not only is 00000 decoded as the STOP instruction, but also so is any other value that has an opcode equal to 00 (such as 00123).

Since many data items have a value less than or equal to 00999, if the CPU does fetch a data byte by mistake it may be interpreted as a STOP statement and the program will terminate (i.e. return control to the computer's operating system). Even so, TINY programmers should ensure that their programs always contain an explicit STOP instruction.

The LD (load) and ST (store) instructions copy data between a designated memory location and the ACC register. The Load instruction copies data from a memory location to the ACC; while the Store instruction copies data from the ACC back to a memory location. Both of these instructions use the operand to specify the address of the desired memory byte. For example, the instruction 01234, would copy the value stored at address 234 to the ACC register. The instruction 04321 would copy the value in the ACC register to the memory location at the address 321.

An aside: When TINY copies data from one location to another, it adheres to a generally accepted rule that most computers follow when copying data. The copy procedure is an example of "non-destructive read - destructive write" operation. In other words, the original data is left undisturbed at the source location; however, any data that was stored at the destination location is replaced by the data that is copied from the source.

The LDA (load address) instruction is not the same as the LD instruction. Instead of loading the contents of memory at the specified address into the ACC, it loads the address itself. Thus, this instruction doesn't need to access memory at all. The operand is simply copied from the IR to the ACC register. Since the operand contains only 3 digits, two zeros are prepended to the operand before it is written to the ACC register. For example the instruction, 03567 will load the value 00567 into the ACC register. This instruction will be used in connection with two of the ROM routines to be explained later.

The arithmetic instructions - ADD, SUB, MUL, and DIV - all use the ACC register as well as a value stored at a memory location. The first value to be added (or subtracted, or multiplied, or divided) should be loaded into the accumulator first. The second value is then specified in the arithmetic instruction itself. For example, the instruction 06123 specifies that the value stored at address 123 should be added to the value that is already in the ACC register. The TINY architecture only supports integer values; floating-point values cannot be represented. Consequently, when dividing, the result must be an integer. Any fractional result will be truncated (e.g. 5 divided by 2 is 2). The following program segment subtracts the value located at address 576 from the value at address 575 and stores the result at address 577:

```
01575
07576
04577
```

This program segment is analogous to the C++ statement: `z = x - y`, where `x` is a variable name that represents the address 575, `y` represents address 576, and `z` represents the address 577.

The IN and OUT instructions allow the CPU to communicate with the outside world. The IN instruction will cause the CPU to pause and wait for one character to be received from the keyboard. When that character is received, its ASCII code is stored in the ACC and the CPU proceeds with the next instruction. The OUT instruction sends the value that is in the ACC to the output device (terminal screen). The terminal screen interprets all values as an ASCII code. Thus, if the ACC register contains the value 00065 when the OUT instruction is executed, the terminal screen will display the letter '`A`'. (An ASCII chart may be found in Appendix C.)

Important note: Although the IN and OUT instructions are operational and you are free to use them, four built-in routines have been provided to make it much easier to input and output data values. For the most part, we will be using these built-in routines. More about them will be discussed later.

*Exercise #1*: On paper write a TINY *machine language* program that would write the text "`Hello World`" on the terminal screen. You may assume that the ASCII codes for the letters "`Hello World`" are stored in the memory addresses 100 through 110. That is, the memory location at address 100 contains the value 00072 (ASCII code for '`H`'); address 101 contains 00101 ('`e`'), and so on. The space found between the words "`Hello`" and "`World`" is represented by the ASCII code 00032, which is stored at address 105. Remember that the first instruction in your TINY program must be stored at address 000.

The branch instructions are used to alter the contents of the PC register. Referring to the instruction cycle, you will note that the PC is incremented BEFORE each instruction is executed; thus, if an instruction alters the PC, the normal sequence of instructions is altered. The JMP (or unconditional branch) instruction simply copies its operand to the PC; thus the next instruction to be fetched will come from this new address. The conditional branch instructions - JG, JL, and JE - will first test the value in the ACC. If the test returns a true condition, the operand of the branch instruction is copied to the PC. If the test returns a false condition, the PC is not altered, and execution continues with the next instruction. For example the instruction 13100 will test the value in the ACC. If the ACC contains a value greater than 0, then the operand, 100, is copied to the PC register; otherwise, this instruction takes no action.

Because the IN and OUT instructions are tedious to use for normal input and output, four callable routines have been made available in READ ONLY MEMORY (ROM). These routines are named PRINTN, PRINTC, INPUTN & INPUTC. To use these routines you must use the CALL instruction. The address for each routine is listed in Appendix A; for instance, the INPUTN routine begins at address 950. Using the CALL opcode (16), the following program segment accepts a number from the keyboard, calculates its square, and prints the result on the screen:

```
16950    ← Call routine named INPUTN to read in a number
04100    ← Save the number received at address 100
01100    ← Re-load the ACC with the number stored at address 100
08100    ← Multiply the number stored in the ACC by the number stored at address 100
16900    ← Call routine named PRINTN to display the answer
```

You should now be ready to write some of your own programs on paper using TINY code.

*Exercise #2*: ***On paper*** write a TINY *machine program* that would read 3 integers from the keyboard and then display their product on the screen.

*Exercise #3*: ***On paper*** write a TINY *machine program* that would print the positive integers 1 through 3 onto the screen, one per line. You may assume that address 101 contains the value `00001`, address 102 contains the value `00002`, and address 103 contains the value `00003`. In addition, you may assume that address 104 contains the value `00013` (ASCII code for a `Carriage Return`), address 105 contains the value `00010` (ASCII code for a `Line Feed`), and address 106 contains the value `00000` (null terminator).

## TINY Assembler

Writing machine code can be quite laborious, even when the machine is as simple as TINY. There are two major problems with programming in machine code. First, working with numeric op-codes is difficult. It is easy to forget the numeric values associated with each instruction. The second problem has to deal with the actual memory addresses when one wants to temporarily store a value or jump to the top of a loop. It would be much easier to program a computer if the programmer could use the mnemonic representations for the op-codes as well as variable names and labels for the memory locations that need to be accessed. Assembly language provides these two facilities.

Assembly language instructions are written using the following format:

```
LABEL:   OPCODE  OPERAND         ;COMMENT
```

Each of the four fields of an instruction is optional (depending upon the intent of the instruction). In fact, a blank line is interpreted as an assembly language instruction where all four fields are missing. Although we usually include spaces or tabs to separate the 4 fields of an assembly language instruction, the only spacing required is that found between the `OPCODE` and the `OPERAND`. Unlike C++, TINY's assembly language is *case insensitive*.

The label field is used to give a name to a particular address in memory. These names are also called symbols. For instance, the first instruction in a loop may contain the label '`WHILE`'; and this name, rather than the numerical address, is referenced by the jump instruction that returns control to the beginning of the loop (e.g. "`JMP WHILE`"). Data variables are another example of the use of labels. The address of a variable may be assigned a symbol so that the programmer can refer to the symbol rather than the numeric address. Each label in a program must be unique. It is also important that the label field be terminated with a colon. (Note: This means that a label cannot itself contain a colon.)

The opcode field is where you will place the mnemonic that represents the instruction desired. One or more spaces and/or tabs must separate the opcode from the operand. If the opcode field is missing, the operand field must be missing as well. If there is no opcode present, the line will not generate any machine code. If a label is present on a line with no opcode, the label refers to the address of the next opcode to be found on a subsequent line in the program.

The operand field's presence is determined by the opcode used. Some opcodes require an operand (e.g. `LD`, `ADD`, `JMP`, etc.) while others do not (e.g. `STOP`, `RET`, `PUSH`, etc.). The operand must be a symbol that matches some label within the program. (The operand associated with the LDPARAM is an exception to this rule.)

The comment field is available so that you may provide internal documentation. It is recommended that you document each group of instructions that correspond to one high-level command. You should also preface your programs with comments that provide the following information:

```
;*****************************************
;*Program Name:   SAMPLE                 *
;*Author:         John Doe               *
;*Date Due:       Sept. 1, 2010          *
;*Assignment:     Lab 0                  *
;*                                       *
;*Description:    This program doesn't   *
;*                do very much.          *
;*****************************************
```

Finally, a section of code that is logically separate from other sections of code needs to be set off by comments. For instance, functions should be prefaced with documentation that describes the function.

*Exercise #4*: Write a TINY *assembly language* program that will prompt for and read 3 integers from the keyboard. The program should then display the average of the 3 integers on the screen with an appropriate message. (See a sample run below.)

Sample Run:
```
SCORE? 75
SCORE? 87
SCORE? 92
Average = 84
```

## TINY CONSTANTS

There are two TINY assembler directives that allow the programmer to store constant data values in memory at "compile-time" (before execution).

The first directive is "`DC`", which stands for *Define Character*. DC is used to enter a string of ASCII text into memory. For example the command: `STR1:    DC      'SCORE? '`
assigns the ASCII code for each character in the string "`SCORE? `" into memory at consecutive addresses beginning at the address labeled `STR1`. Assuming that `STR1` is a label for address 100, then the previous command would cause the following values to be stored in TINY's memory:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Address | 100 | 101 | 102 | 103 | 104 | 105 | 106 |
| Contents | 00083 | 00067 | 00079 | 00082 | 00069 | 00063 | 00032 |

A second assembler directive, "`DB`", is used to enter constant integer values. "`DB`" stands for Define Byte. (For reasons of compatibility with Microsoft's assembler, the mnemonics, "`DW`" and "`DD`" are also supported and are both equivalent to "`DB`".)

*Define Byte* places a numeric value at the current address. For example: `#1:    DB      1`
places the value `00001` in memory at the address specified by the label "`#1`".

The ROM routine, `PRINTC`, requires a string that is null-terminated (same as "zero-terminated"). For example, if it is desired that the string "`SCORE? `" be displayed, then the following directives are needed:

```
STR1:  DC     "SCORE? ";
       DB     0
```

Given the two declarations above, the instructions needed to display the string would then be:

```
       LDA    STR1      ; cout << "SCORE? ";
       CALL   PRINTC
```

## TINY VARIABLES

In order to allocate space for a variable, a third assembler directive, "DS" - the *Define Storage* assembler directive, should be used. Define Storage allocates memory but does not assign an initial value. For example, the command:

`i: DS 1` is equivalent to the C++ command: `int i;`

However, C variable declarations are sometimes used to give a variable an initial value. An example would be the C statement: "`int i = 0;`". The similar declarations in TINY would use the `DB` directive `i: DB 0`.
However, it is *important* that you realize that this assignment is done at "compile-time" as opposed to "run-time". This is in contrast to C++, which will often add the declaration: "`int i = 0;`" as an assignment statement found in a constructor – making it a "run-time" assignment.

## ASSIGNMENT STATEMENTS

Assignment statements are commonly used in C and C++ to assign a value to a variable. Standard hierarchy rules are used to govern when different operators are applied in a complex assignment statement. *No such hierarchy rules exist in assembly language* – it is up to the programmer to ensure that operators are applied in the correct order. For example, the C statement:

`x = a + b * c / d;` should be implement in TINY as:

```
ld     b
mul    c
div    d
add    a
st     x
```

## INPUT/OUTPUT STATEMENTS

For an example of input and output statements similar to those provided by C++, see "ROM ROUTINES" on the second page of Appendix A

## PROGRAM CONTROL STATEMENTS

Languages such as C++ make extensive use of commands such as "`if`", "`while`", and "`for`" to control the flow of logic within a program. Assembly language does not have such control statements built-in. It is up to the programmer to utilize good programming practices and emulate such control structures. Appendix B contains several suggestions on how to emulate common C++ control structures.

## SAMPLE PROGRAM

In figure 2 we find a sample TINY program complete with comments. Note the use of header comments that document the name of the program, the author and a brief description of what the program accomplishes. Also present are C++ - like comments for those groups of TINY instructions that implement functionality similar to what the C++ statement would accomplish.

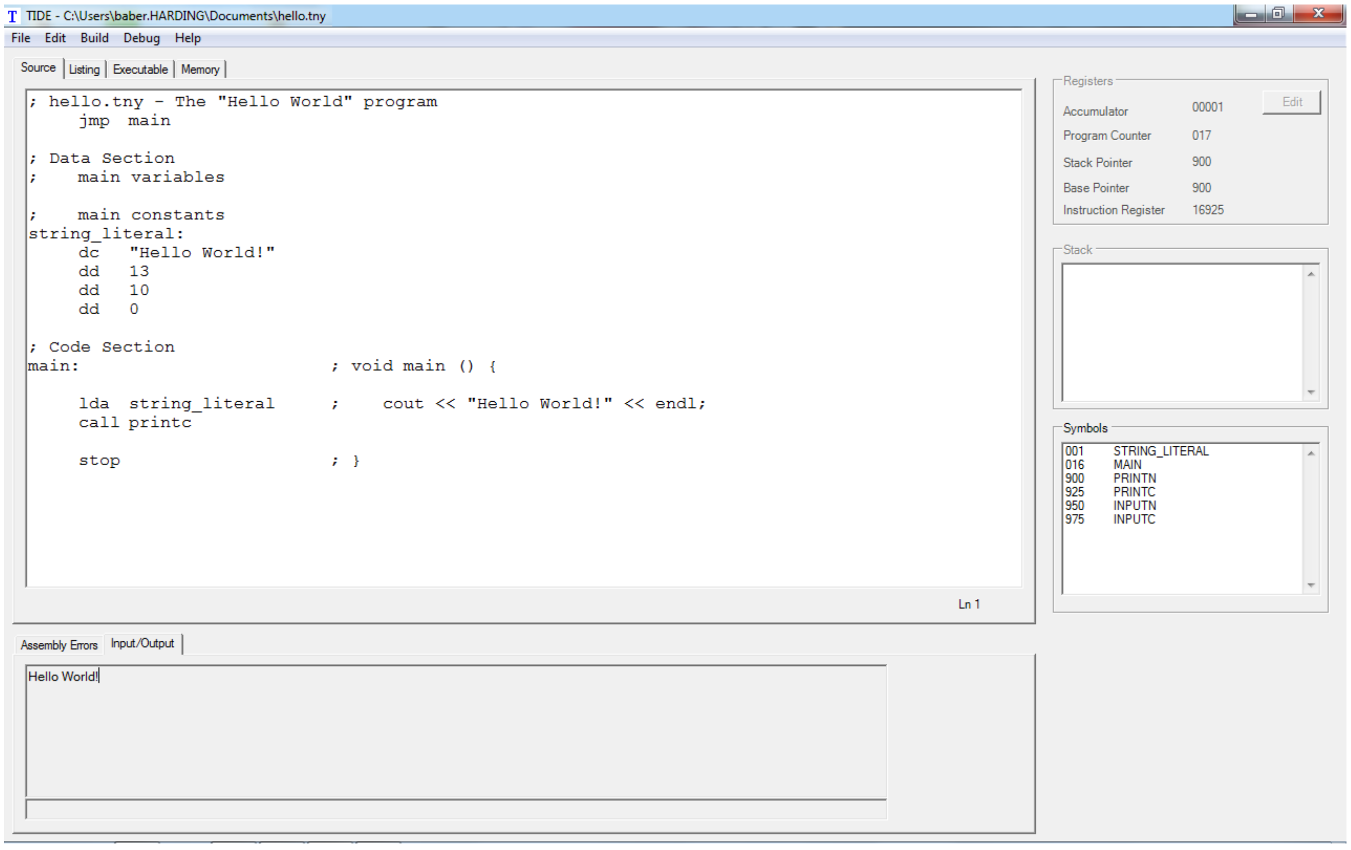
The description comment in figure 2 is incomplete. Can you determine what the program is designed to do and complete the description comment?

```
;*************************************************************************************
;*Program Name:   sample.tny                                                        *
;*Author:         Steve Baber                                                       *
;*                                                                                  *
;*Description:    This program accepts an integer and if the integer _______________ *
;*                                                                                  *
;*                __________________________________________________________________ *
;*                                                                                  *
;*************************************************************************************
       jmp    main          ; <Start Vector>

; Data Section
; CONSTANTS
#1:    db     1

; VARIABLES
i:     ds     1             ; int i;


; Code Section
main:                       ; void main () {

       call   inputn        ;    cin >> i;
       st     i

       ld     i             ;    if (i >= 0) {
       jg     if1_begin
       je     if1_begin
       jmp    if1_end

if1begin:
       ld     i             ;       cout << (i * i + 1);
       mul    i
       add    #1
       call   printn

if1_end:                    ;   }

       stop                 ; }
```

**Sample TINY Program**
**Figure 2**

## TIDE – Tiny Integrated Development Environment

To aid the development and testing of TINY programs, the Tiny Integrated Development Environment (TIDE) has been created. Here is a screen shot of TIDE with the "Hello World" program loaded:


TIDE - C:\Users\baber.HARDING\Documents\hello.tny
File Edit Build Debug Help
Source Listing Executable Memory
Registers
Edit
Stack
Symbols
Ln 1
Assembly Errors Input/Output

```
; hello.tny - The "Hello World" program
     jmp  main

; Data Section
;    main variables

;    main constants
string_literal:
     dc   "Hello World!"
     dd   13
     dd   10
     dd   0

; Code Section
main:                         ; void main () {

     lda  string_literal      ;    cout << "Hello World!" << endl;
     call printc

     stop                     ; }
```

```
Accumulator          00001
Program Counter      017
Stack Pointer        900
Base Pointer         900
Instruction Register 16925
```

```
001  STRING_LITERAL
016  MAIN
900  PRINTN
925  PRINTC
950  INPUTN
975  INPUTC
```

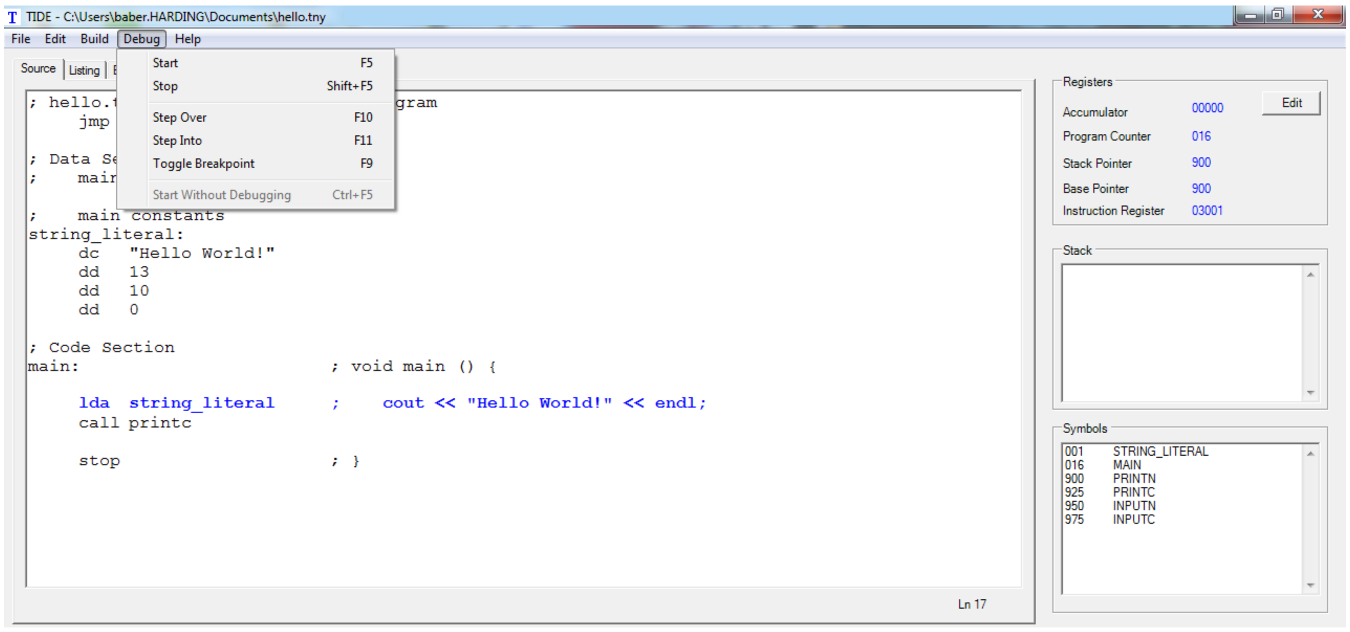
```
Hello World!
```

There are several windows shown above that are helpful when writing a new TINY program. The largest window displayed above is one that is used to type and edit the source code for a program. There are three other tabs next to the "Source" tab. These three - Listing, Executable, and Memory – will bring up other large windows that are helpful with the debugging process. More about these three will be discussed later.

Below the "Source" window you will notice the "Input/Output" window. This is where input and output generated by the TINY program will appear. A similarly-sized window named "Assembly Errors" that lists any syntax errors that might occur is available by clicking on the appropriate tab.

The information to the right of the "Source" window is very helpful during the debugging phase of program creation. While stepping through the program, one can monitor the values stored in the five registers: Accumulator, Program Counter, Stack Pointer, Base Pointer and Instruction Register. There are also two windows that reveal the contents of the system stack and the symbol table.

The main menu at the top of the screen allows a programmer to step through the TINY code:



```
; hello.t                              gram
     jmp

; Data Se
;    mair

;    main constants
string_literal:
     dc   "Hello World!"
     dd   13
     dd   10
     dd   0

; Code Section
main:                         ; void main () {

     lda  string_literal      ;    cout << "Hello World!" << endl;
     call printc

     stop                     ; }
```

```
001  STRING_LITERAL
016  MAIN
900  PRINTN
925  PRINTC
950  INPUTN
975  INPUTC
```

The programmer may press the function keys, F10 & F11, to perform a step. F10 will treat a function call as a single step (step over); while F11 will actually step into a function. Breakpoints may be set (or cleared) by pressing F9 while the cursor is on the desired line. Pressing F5 will run the program until either a breakpoint line or a STOP instruction is reached. Pressing CTRL+F5 will run the program without debugging.

## Homework Problems

1. Write a TINY program that is functionally equivalent to the following C++ program.

```
// Calculate a box volume given ht, length & width
#include <iostream.h>
void main () {
   int height,length,width,area,volume;
   cout << "Enter height, length, width: ";
   cin >> height >> length >> width;
   area = height * length;
   volume = area * width;
   cout << "Volume is " << volume << endl;
}
```

2. Write a TINY program that is functionally equivalent to the following C++ program.

```
// IntegerDivision.cpp
//    This program illustrates the division and modulus operators using integers.

#include <iostream>
using namespace std;

void main() {
   int first, second, divisor, remainder;

   cout << "Enter an integer: ";
   cin >> first;
   cout << "Enter another integer: ";
   cin >> second;

   divisor = first / second;
   remainder = first % second;

   cout << divisor << endl;
   cout << remainder << endl;
}
```

3. Write a TINY program that is functionally equivalent to the following C++ program.

```
// SUM - A program to sum a list of positive integers
#include <iostream.h>
void main () {
   int number, sum;
   sum = 0;
   cout << "Enter a list of positive integers (ending with -1): ";
   cin >> number;
   while (number > 0) {
      sum = sum + number;
      cin >> number;
   }
   cout << "The sum is " << sum << endl;
}
```

4. Write a TINY program that is functionally equivalent to the following C++ program.

```
// hilo.cpp
#include <iostream>
using namespace std;

void main ()
{
   int answer
      int guess, count;

   answer = 60;
      count = 1;
   cout << "Guess a number between 1 and 100." << endl;
   cout << "Guess? ";
   cin  >> guess;
      while (guess != answer) {
      count++;
      if (guess < answer)
         cout << "Too low!" << endl;
      else
         cout << "Too High!" << endl;
      cout << "Guess? ";
      cin  >> guess;
      }
      cout << "You found the answer in " << count << " guesses." << endl;
}
```

5. Write a TINY program that is functionally equivalent to the following C++ program.

```
// rt_triangle.cpp
//   A program used to determine if a triangle is a “right” triangle.

#include <iostream.h>
void main () {
   int side1, side2, side3, hyp;

   cout << "Enter length of 1st side: ";
   cin  >> side1;
   cout << "Enter length of 2nd side: ";
   cin  >> side2;
   cout << "Enter length of 3rd side: ";
   cin  >> side3;

   if ( (side1 * side2 * side3) > 0 ) {
       hyp = side1*side1 + side2*side2;

       if (hyp == side3*side3)
          cout << "It is a right triangle!" << endl;
       else
          cout << "It is not a right triangle." << endl;
   }
}
```

6. Write a TINY program that is functionally equivalent to the following C++ program.

```
// isprime.cpp
//   This program accepts an integer and determines if it is prime.

#include <iostream.h>
void main () {
   int x,p,d,r;

   cout << “Enter an integer greater than 1: “;
   cin  >> x;
   p = 1;
   d = 2;

   while (d <= (x-1)) {
      r =  x - (d * (x / d));    // (Same as r = x % d;)
      if (r == 0) {
         p = 0;
      }
      d++;
   }

   if (p == 1)
      cout << “Yes” << endl;
   else
      cout << “No” << endl;
}
```