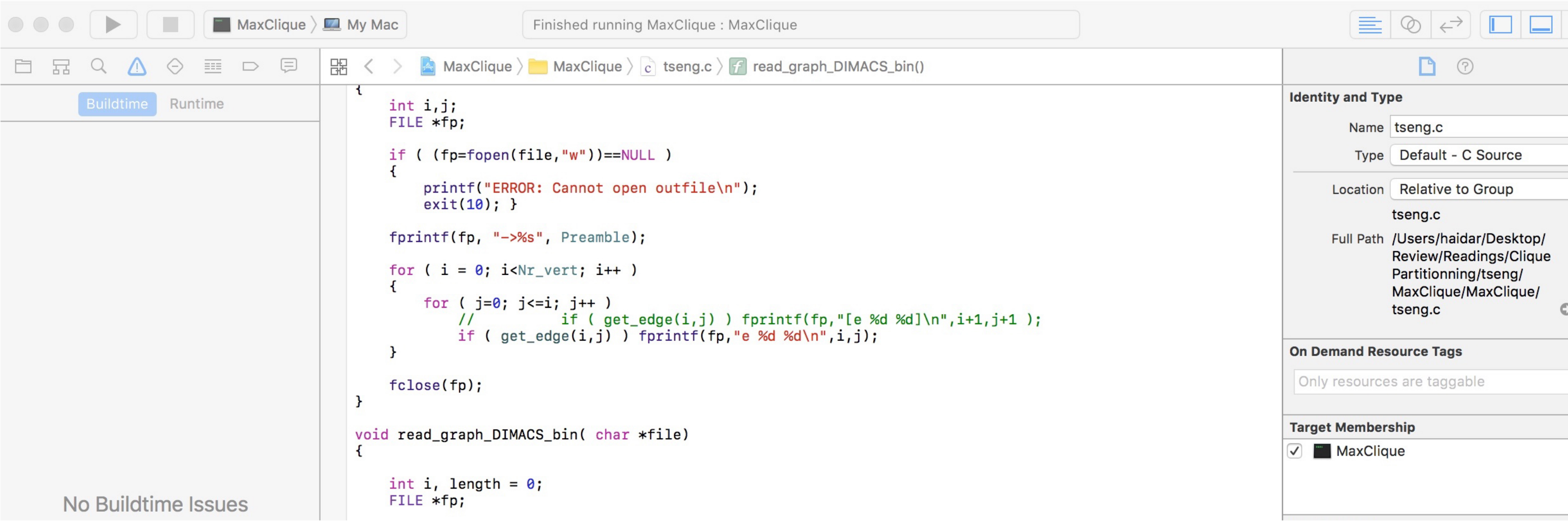
Hands-On Computer Organization Lab

With C and Verilog

Haidar M. Harmanani

First printing, March 2016



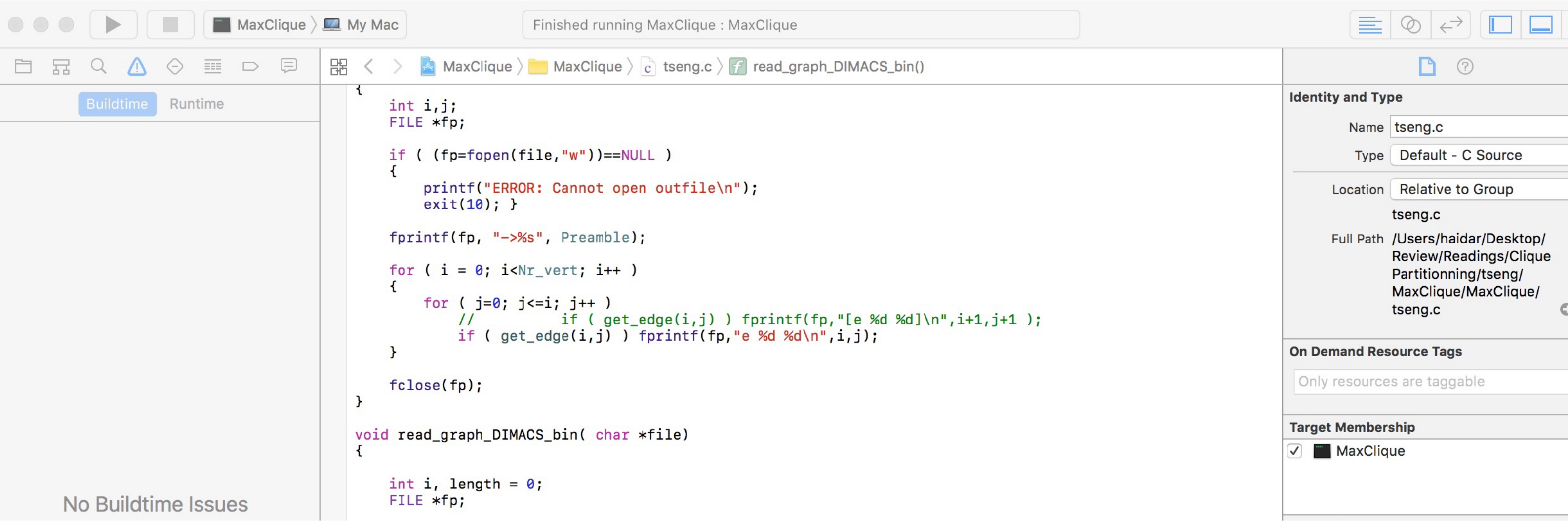
```
{
    int i,j;
    FILE *fp;

    if ( (fp=fopen(file,"w"))==NULL )
    {
        printf("ERROR: Cannot open outfile\n");
        exit(10); }

    fprintf(fp, "->%s", Preamble);

    for ( i = 0; i<Nr_vert; i++ )
    {
        for ( j=0; j<=i; j++ )
            //          if ( get_edge(i,j) ) fprintf(fp,"[e %d %d]\n",i+1,j+1 );
            if ( get_edge(i,j) ) fprintf(fp,"e %d %d\n",i,j);
    }

    fclose(fp);
}

void read_graph_DIMACS_bin( char *file)
{

    int i, length = 0;
    FILE *fp;
```

Name: tseng.c
Type: Default - C Source
Location: Relative to Group
tseng.c
Full Path: /Users/haidar/Desktop/Review/Readings/Clique Partitionning/tseng/MaxClique/MaxClique/tseng.c

On Demand Resource Tags

Target Membership
✓ MaxClique

Contents


MaxClique 〉 My Mac
Finished running MaxClique : MaxClique
MaxClique 〉 MaxClique 〉 tseng.c 〉 read_graph_DIMACS_bin()
Buildtime
Runtime
No Buildtime Issues

```
{
    int i,j;
    FILE *fp;

    if ( (fp=fopen(file,"w"))==NULL )
    {
        printf("ERROR: Cannot open outfile\n");
        exit(10); }

    fprintf(fp, "->%s", Preamble);

    for ( i = 0; i<Nr_vert; i++ )
    {
        for ( j=0; j<=i; j++ )
            //          if ( get_edge(i,j) ) fprintf(fp,"[e %d %d]\n",i+1,j+1 );
            if ( get_edge(i,j) ) fprintf(fp,"e %d %d\n",i,j);
    }

    fclose(fp);
}

void read_graph_DIMACS_bin( char *file)
{

    int i, length = 0;
    FILE *fp;
```

Identity and Type

Name: tseng.c
Type: Default - C Source
Location: Relative to Group
tseng.c
Full Path: /Users/haidar/Desktop/Review/Readings/Clique Partitionning/tseng/MaxClique/MaxClique/tseng.c

On Demand Resource Tags

Only resources are taggable

Target Membership

✓ MaxClique

1. Introduction to C

"A C program is like a fast dance on a newly waxed dance floor by people carrying razors"

—Waldi Ravens

"It's hardware that makes a machine fast. It's software that makes a fast machine slow"

—Craig Bruce

C is a structured programming language that was developed by *Dennis Ritchie* at Bell Laboratories as a "pleasant, expressive and versatile language for a wide variety of programs" [**Kernighan1988**]. C is a popular systems programming language that provides programmers with the ability to interact with the underlying hardware using a high-level syntax. Although there has been a shift towards languages with built-in support for specialized operations such as `Python` or `Ruby`, C remains the language of choice for low-level programming. This chapter briefly introduces C for the purpose of modeling hardware and architectures.

1.1 Programming Environment

All examples in this lab manual are written and tested in a MacOS environment that includes *Sublime Text* for editing the source code. We will be using `gcc`[1] to compile C code and `gdb` for debugging purposes.

1.2 Compiling and Executing a C Program

`gcc` is the first gnu compiler in a collection that includes C++, Objective-C, Fortran, Ada, and Go. It is available on Linux and MacOS systems. A C source code is a text file with a `.c` suffix that

[1] A comprehensive `gcc` guide can be found at: `https://gcc.gnu.org/onlinedocs/gcc-4.8.3/gcc.pdf`

describes high-level instructions that are to be performed by the computer. C files are usually used in conjunction with header files that end with the `.h` suffix. C source files are preprocessed by the C preprocessor prior to compilation. The C preprocessor also includes the header files in the source code during compilation. C source files that have `.i` suffix are not preprocessed.

A C program is compiled using a compiler. The most common compiler is `gcc`, the GNU C Compiler which is preinstalled in `Linux` as well as in `Mac OS`. `gcc` has various command line options which you are encouraged to explore. In this lab manual, however, we will be mainly using the following options:

- `-o`: Specifies the name of the executable file. For example, the `gcc -o program program.c` command would compile. `program.c` into an executable named `program`. The program is executed using the command: `$ ./program`.
- `-g`: Ensures that source code information are included in the binary for `gdb`.
- `-c`: Compiles a source without linking and producing an executable. The output will be in the form of an object file for each source file with the `.o` suffix.

1.3 Creating and Compiling a C Program

Use the `Sublime Text` editor or `Atom` in order to create the following C program, and save it as `hello.c`. You could also use the built-in editor if you are using `Xcode` or the `Microsoft Visual Studio`:

C Listing 1.3.1 Hello, World!

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

All C programs typically include one or more `#include` lines that import the definitions of specific standard library. In the above example, the `printf` function which prints a string to the screen is declared as follows:

```c
int printf(const char *format, ...)
```

The `main` function in C is *called* by the operating system when the program is executed. It is similar to any other function: it has a return type and accepts user inputs via command line arguments. In the above example, the `main` function returns an *integer*. The `printf` function prints the strings given as arguments to the screen. In order to force a new-line character, a printed string is terminated with the `\n`. Finally, the `return 0` statement returns the integer 0, which indicates the program's *exit status*.

Compile `hello.c` into machine code using `gcc`:

```
gcc -o hello hello.c
```

The `-o hello` tells the compiler to name the executable file `hello`. The input program is given as the next argument `hello.c`. Ignoring the `-o` option, would compile the program into the default executable `a.out`.[2]

The program is executed using the following command:

[2] a.out is the abbreviated form of the *assembler as* and the *link editor ld* output

```
./hello
```

The program should print `Hello World.` on the screen. The `./` before `Hello` is required so that the system knows where to find the file (`./` refers to the current directory – where `gcc` created the executable).

1.4 Comments

Comments in C are similar to Java and start with either // for single line comments, or /* */ for multiple line comments.

1.5 Command Line Arguments

Let us slightly modify the `hello.c` program as follows. Rename the new file `converter.c`.

C Listing 1.5.1 Command line arguments in C

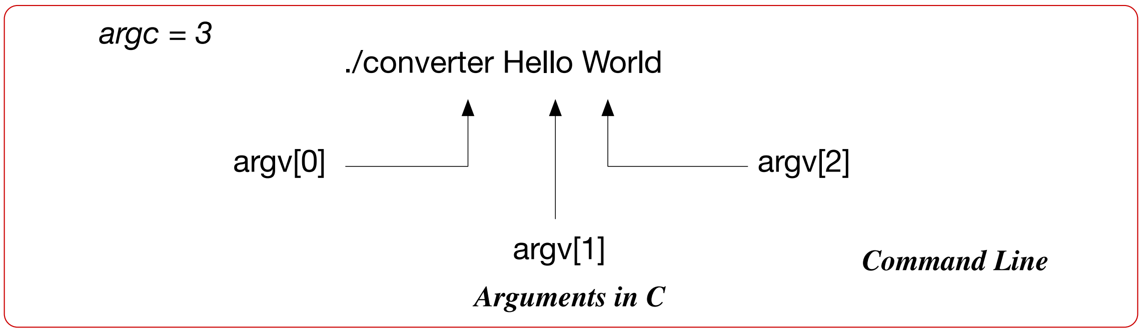
```
// converter.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Argument 1: %s\nArgument 2: %s\n", argv[1], argv[2]);

    return 0;
}
```

In the above source code, `argc` is of type integer while `argv` is an array of string parameters that are passed on the command line. When the program is executed, the number of arguments and the array of argument strings (in this case {"converter", "Hello", "World"}) are passed to the `main` function. It should be noted that `argv[0]` is the name of the program, `converter` in this case.

Exercise 1.1 Compile the `converter.c` program using the following command:

```
gcc -o -Wall converter converter.c
```

Execute the `converter` program without any parameters. What do you notice? Can you fix the problem? How? ■



1.6 Declaring C Variables

A Variable allocates space in memory where the value of that variable can be stored (Figure 1.1). All C variables must be declared before they are used using the following syntax:

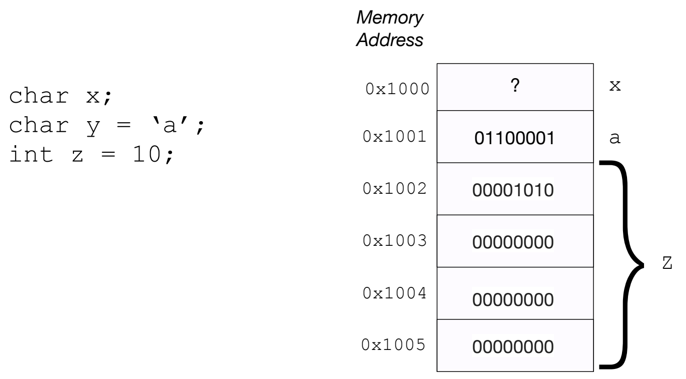


Figure 1.1: Variables Memory Representation. Memory is usually byte aligned and thus the integer variable z is stored using a word of memory, which is 4 bytes. Within each byte (each block of eight bits), the bits are written from right to left

```
type variable_name;
```

Variables could be integers, floating point numbers, characters, arrays, or structures. Integers and floating point numbers can be be signed or unsigned. The size of an integer may vary and could be of the following types: `int`, `short`, and `long` while floating point numbers could be `double` or `float`. Although C has different types for `int` and `char`, they can be used interchangeably. Multiple variables maybe declared simultaneously using a comma separated syntax, and maybe initialized when declared.

```
float gpa;
int count, index = 0;
```

Variables maybe assigned expressions which are evaluated before any of the assignments take place:

```
count = 2 * i;
```

Unsigned types have an increased range by a one bit factor. Thus, while a signed `char` ranges from -128 to 127, an unsigned `char` has only positive numbers that range from 0 to 255.

Type	Size in Bytes
char	1
short	2
int	4
long	8
float	4
double	8

Table 1.1: C variables types and corresponding sizes in bytes

C Listing 1.6.1 Min and Max integers in C

```c
#include <stdio.h>
#include <math.h>
#include <limits.h>

int main(void)
{
  printf("The min integer on my machine is %d\n", INT_MIN);
  printf("The max integer on my machine is %d\n", INT_MAX);
  printf("An integer is %lu bytes\n", sizeof(int));

   return 0;
}
```

1.7 C Blocks

A C block has the following format:

```
<block> := {<statements>}
```

A block is syntactically equivalent to a single statement and can be used to enclose conditional statements such as if, else, while, and for. Variables can be declared inside any block.

1.8 Control Statements

C provides control statements that deal with branching as well as with looping. Branching is handled using `if` statements as well as using `case` statements:

```c
if (expression)
  statement;
```

Nested `if-then-else` statements have the following form:

```c
if (expression)
  {
    Block of statements;
  }
else if(expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }
```

Another way to write *if-then-else* statement is using the ? operator:

```
if condition is true ? then X return value : otherwise Y value;
```

For example:

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
```

1.9 Loops

Loops provide a way to repeatedly execute a statement as long as a condition is satisfied. Loops are very similar to Java and inclde `while`, `do/while`, and `for` loops are common loop constructs used in many high-level languages including C.

```
while (condition is true) { // while loops check the condition first
 loop body
}

do {
 loop body
} while (condition is true) // do-while loops check the condition last

for (initialization; condition is true; loop operation) {
 loop body
}
```

1.10 Functions

Functions in C are defined by declaring the function name, type, and a parenthesized list of formal parameters. The order of functions inside a file is arbitrary. It does not matter if you put function one at the top of the file and function two at the bottom, or vice versa.

C Listing 1.10.1 Binary Search

```
int binary_search(int nums[], int x)
{
    int low = 0, mid, high, item;

    high = length -1;
    while (low  < high)
    {
        mid = (low + high)/2;
        item = nums[mid];

        if (x == item)
           return mid;
        else if (x < item)
            high = mid - 1;
         else
             low = mid + 1;
```

```c
    }
    return(-1);
}
```

A good practice when writing C functions is to write *prototypes* at the top of a C source code file to describe the function's type signature: what the function returns and what arguments it takes.

1.11 Function Parameters

A function may return one value using the `return` statement.

1.12 Variables Scope

Variables that are visible in certain parts of the program are said be in *scope*. Typically, a variable is in scope if it is declared inside the block that is currently executing. A C program may have *global* and *local* variables. A global variable is declared outside of all functions and can be accessed anywhere in the program while a local variable is declared within a function or a block and can be accessed only within that function or block.

C Listing 1.12.1 Variables Scope

```c
#include <stdio.h>
#include <stdlib.h>

int add (void);
int n, m;                    // Global Variables.  Not a good style!

int main(int argc, char **argv) {

    n = atoi(argv[1]);
    m = atoi(argv[2]);

    printf("Arg 1: %d\nArg 2: %d\nSum: %d\n", n, m, add(n, m));
    return 0;
}

int add ()
{
  int val;                                         // Local Variable
  if (n < m)
    val = n + m;
  else
    val = n - m;
  return(val);
}
```

Exercise 1.2 Research the `atoi` function. What does it do? What does it accept as parameters, and what does it return? ■

C Listing 1.12.2 The following C program will not compile as variable x is accessed outside of the block where it was declared.

```
#include <stdio.h>
#include <stdlib.h>

int add (int , int );
int n, m;                                          // Global Variables
int main(int argc, char **argv) {

    n = atoi(argv[1]);
    m = atoi(argv[2]);

    if (n < m)
    {
      int x = 0;
    }
    printf("%d", x);
    return 0;
}
```

1.13 Bitwise Operators

Bitwise operators are used in order to access individual bits through the full byte. In C, binary numbers start from the right most significant bit to the left. C supports the bitwise operators shown in Table 1.2.

Operator	Description
&	bitwise AND
|	bitwise OR
∼	bitwise NOT
&	bitwise AND
ˆ	bitwise XOR
<<	bitwise shift left
>>	bitwise shift right

Table 1.2: Bitwise Operators

For example, shifting the number 8, written in binary as 00001000, two bits to the left results with 00100000, which is equal to 32 in decimal. In other words, every bit is displaced to the left by two positions while zeros are added in place of the shifted bits as padding. Left shifting by n bits is equivalent to multiplying by 2^n while right shifting by n bits is equivalent to dividing by 2^n

C Listing 1.13.1 Multiplication by 2^n

```
int mult_by_pow_2(int number, int n)
{
    return number << n;
}
```

The bitwise AND operator is a single ampersand: &. A binary AND simply takes the logical AND of the bits in each position of two binary numbers and produces a binary 1 if both bits are 1, 0 otherwise. The truth table for the bitwise And, Or, and Xor Bit operations is shown in Table 1.3.

p	q	p &q	p |q	p ^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 1.3: Truth table for the AND, OR, and XOR bitwise operations

For instance, working with a byte (the char type):

```
01001000 & 10111000 = 00001000
```

The bitwise OR (|) works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1:

```
01001000 | 10111000 = 11111000
```

The bitwise complement operator, the tilde, ∼, complements every bit ($1 \to 0$ or $0 \to 1$). A useful application for the bitwise complement is to find the largest possible value of an *unsigned* number:

```
unsigned int max = ∼0;
```

Finally, the bitwise exclusive-or (XOR) operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a caret, ˆ, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR. For example, the XOR for two numbers represented in binary as `10101010` and `01110010` is:

```
01110010 ^ 10101010 = 11011000
```

1.14 Macro Definition

A macro is a named constant or code fragment in C. The content of the macro replaces the constant or the code fragment whenever the name is used. In this lab, we will use macros in order to model computer instructions. For example, consider the instruction in Figure 1.2. The instruction can be modeled using the following two macros:

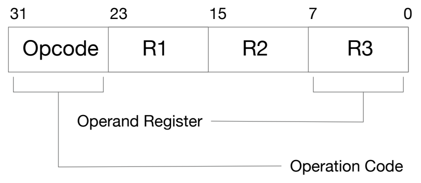


Figure 1.2: Simple Add instruction

```
#define CNAME value
```

■ **Example 1.1** It is convenient in this lab to use macros in order to represent instructions mnemonics. For example, the following defines the opcode of an add instruction as well as the add instruction itself.

```
#define OP_ADD  0x03
#define ADD(R1,R2,R3) (OP_ADD << 28) | (R1 << 24) | (R2 << 8) | R3
```

■

1.15 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. An array of *letter grades* is declared in C using the following syntax:

```
char grade[5];
```

It is important to note that C does not have predefined *string* type. Instead, a string is an array of characters that is null terminated. A student's name is declared as follows:

```
char name[100];
```

1.16 Structures

Structures in C are used in order to group related information in a single entity. For example, grouping a student's *name*, *id*, and *gpa* can be done using the following structure:

```
struct Student {
          char  name[100];
          int id;
          float gpa;
        };
```

1.17 Writing a Testbench

A *testbench* is an environment that is used in order to verify the correctness or soundness of a design or model. A possible testbench for a program that finds the smallest and largest of two numbers would look like this:

```
#include <stdio.h>
#include <stdlib.h>

void MinMax (int arg1, int agr2, int *min, int *max);
int main(int argc, char **argv)
{
    int i, min, max;

    if (argc < 2 || argc % 2 == 0)
    {
      printf("Program expects an even numbers of inputs\n");
      exit (1);
    }

    for (i = 1; i < argc; i+=2)
    {
      MinMax (atoi(argv[i]), atoi(argv[i+1]), &min, &max);
      printf("Min: %d, Max: %d\n", min, max);
    }
}

void MinMax (int arg1, int arg2, int *min, int *max)
{

  if (arg1 < arg2)
  {
    *min = arg1;
    *max = arg2;
  }
  else
  {
    *max = arg1;
    *min = arg2;
  }
}
```

Exercise 1.3 Write a function `int htoi(char s[])`, which converts a string of hexadecimal digits (including an optional `0x` or `0X`) into its equivalent integer value. The allowable digits are `0` through `9`, `a` through `f`, and `A` through `F`. You should create three fles, a header file `str.h`, a program file `str.c`, and test bench file `testbench_str,c`. Write a lab report that includes a brief explanation of the code and the functions. Your code should be structured and readable. ■

1.18 Lab 1: Hardware Modeling Using C

An Arithmetic and Logic Unit (ALU) is a building block in every modern processor. This lab tackles the architectural design and simulation of the 16-bit ALU shown in Figure 1.3, using C. The ALU has two 16-bit input registers, `A` and `B`, and one 16-bit output register, `Result`. The `ALU` can perform the following arithmetic and logical operations:

1. `Add`
2. `Subtract`
3. `AND`
4. `OR`

The ALU has a two-bit selector, `ALUOp` that determines the type of operation that the ALU is executing. Typically, 2^n bits are needed in order to select from n functions. A multiplexer in hardware is a building block that can be used to select one output from `n` different inputs. The ALU has two two different multiplexers. The first multiplexer, *mux1*, selects either the `AND` or the *OR* operation. The second multiplexer, *mux2* selects either the output of the *Add/Sub* unit or the output of the *mux1*. The output is saved in result. The ALU can be modeled architecturally in C as follows:

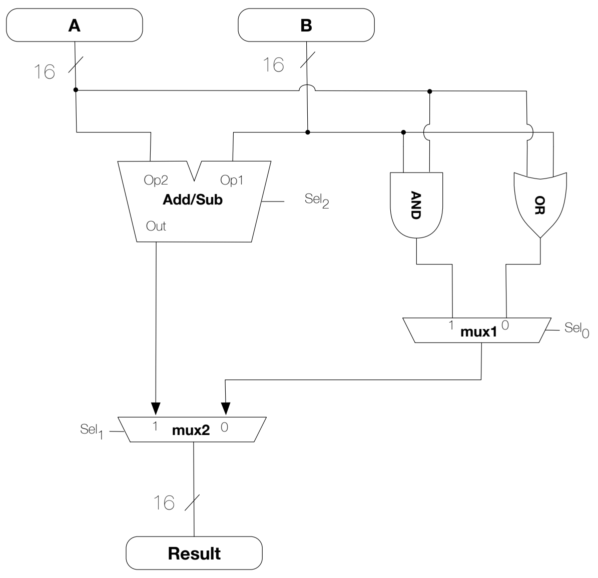


Figure 1.3: ALU Block Diagram

C Listing 1.18.1 ALU Model

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define OP_ALU_ADD 0
#define OP_ALU_SUB 1

void add_sub(int16_t A, int16_t B, uint8_t ALUOp, int16_t *Result);
void or(int16_t A, int16_t B, int16_t *Result);
void and(int16_t A, int16_t B, int16_t *Result);
int16_t mux(int16_t , int16_t , uint8_t );

int main(int argc, char **argv);

void add_sub(int16_t A, int16_t B, uint8_t ALUOp, int16_t *Result)
{

  switch (ALUOp & 0x01) {
    case OP_ALU_ADD:
        *Result =  A + B;
        break;

    case OP_ALU_SUB:
        *Result =  A - B;
        break;
    default:
        perror("OpCode Error.  Aborting");
        exit(1);
  }
}

void and(int16_t A, int16_t B, int16_t *Result)
{
    *Result = A & B;
}

void or(int16_t A, int16_t B, int16_t *Result)
{
    *Result = A | B;

}

int16_t mux(int16_t Input1, int16_t Input2, uint8_t selector)
{
  return(selector ==  0 ? Input1 : Input2);
}

int main(int argc, char **argv)
{
    int16_t A = 3, B = 2, out1, out2, result, temp;
    uint8_t sel = 0x3;

    add_sub(x, y, (sel >> 2) & 0x01, &result);

    or(A, B, &out1);
    and(A, B, &out2);

    temp = mux(out1, out2, (sel >> 1) & 0x01);

    result = mux (temp, result, sel & 0x01);
    printf("ALU Result: %d\n", result);
}
```

Exercise 1.4 In a breakout room, trace each line of code in the above ALU datapath. Explain the program design to your lab partner, paying a special attention to the `bitwise` operations. ■

Lab 1.1 What type of changes you need to implement in order to modify the design so that it can perform the following operations: 1) XOR, 2) NOT, 3) Multiply, and 4) Divide. Assume that the ALU can perform complex operations such as divisions (although not a practical design consideration). Implement your changes and make sure that the program takes its inputs from the command line at the following order `A, B, sel0, sel1, sel2`. Test the correctness of your program using a testbench. Write your report and upload it to Lab 1 on Google Classroom. ■