# 1. Code

```
#include <string>
using std::string;

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <fstream>
using std::ifstream;

/**
 * Class Car representing a car
 */
class Car
{
    private:
        string make;
        string model;
        int year;
        string color;
    public:
        Car(); // Default constructor sets the data fields to default values

         // Sets the class member fields to the values passed
        void setFields(string mk, string md, int yr, string cl);

        string getMake(); // Returns make
        string getModel(); // Returns model
        int getYear(); // Returns year
        string getColor(); // Returns color
};
```

```
/**
 * Class CarRecords representing a record of a car
 */
class CarRecords
{
    private:
        int arraysize; // Keep track of number of records
        ifstream infile; // input file stream used to read from a file
        Car *cars; // a pointer car (to a dynamically allocated array of cars)
    public:
        CarRecords(int size); // Reads file records to array
        ~CarRecords(); // Frees memory and closes file handler
        void printCarRecords();
        void sort_cars_by_make(); // Sorts the records by make in asc order
        void sort_cars_by_year(); // Sorts the records by year in desc order
        void print_duplicates(); // Prints out duplicate records
};

/**
 * Print out the car's information
 */
void print_car_helper(Car * car) {
    cout << car->getMake() << ", " <<
        car->getModel() << ", " <<
        car->getYear() << ", " <<
        car->getColor() << endl; // Print out each car
}

/**
 * Default constructor sets the data fields to their default values
 */
Car::Car()
{
    make = "";
    model = "";
    year = 0;
    color = "";

}
```

```
/**
 * Sets the class member fields to the values passed
 *
 * @param mk, string representing make
 * @param md, string representing model
 * @param yr, integer representing year
 * @param cl, string representing color
 */
void Car::setFields(string mk, string md, int yr, string cl)
{
    make = mk;
    model = md;
    year = yr;
    color = cl;
}

/**
 * Returns make
 */
string Car::getMake()
{
    return make;
}

/**
 * Returns model
 */
string Car::getModel()
{
    return model;
}

/**
 * Returns year
 */
int Car::getYear()
{
    return year;
}
```

```
/**
 * Returns color
 */
string Car::getColor()
{
    return color;
}
```

```
/**
 * Constructor sets the value passed to the arraySize member,
 * creates a dynamic array enough to hold arraySize objects of
 * type Car. In addition, the constructor uses the infile file
 * stream and the setFields() method to initialize all the
 * cars array elements with the car records read from the file.
 *
 * @param size, size of dynamic array
 */
CarRecords::CarRecords(int size)
{

    if (arraysize < 10 && arraysize > 0) {
        arraysize = size;
        cars = new Car[arraysize];
    } else {
        arraysize = 10; // Default size to 10
        cars = new Car[arraysize];
    }

    const int MAX_CHARS_PER_LINE = 512;
    const int MAX_TOKENS_PER_LINE = 4;
    const char* const DELIMITER = ", ";

    infile.open ("CarRecords.txt"); // Open file
    if (!infile.good())
        exit(0); // Error - file not found

    int current_car_index = 0; // Keep track of what car we are on
```

```
    // Read each line of the file
    while (!infile.eof() && current_car_index < arraysize)
    {
        char buf[MAX_CHARS_PER_LINE];
        infile.getline(buf, MAX_CHARS_PER_LINE); // Place line in buffer

        // array to store memory addresses of the tokens in buf
        const char* token[MAX_TOKENS_PER_LINE] = {}; // Tokens

        // Parse the line
        token[0] = strtok(buf, DELIMITER); // First token
        if (token[0])
        {

            for (int n = 1; n < MAX_TOKENS_PER_LINE; n++)
            {
                token[n] = strtok(0, DELIMITER); // subsequent tokens
                if (!token[n]) break; // no more tokens
            }
        }

        // Set the current car to the values in the corresponding line
        cars[current_car_index].setFields(token[0], token[1],
                atoi(token[2]), token[3]);
        current_car_index++;
    }
}
```

```
/**
 * Destructor frees the memory allocated with new, and closes the file
 * handler
 */
CarRecords::~CarRecords()
{
    delete [] cars;
    infile.close();
}

/**
 * Prints out the car records from the array of objects
 */
void CarRecords::printCarRecords()
{
    printf("PRINTING %d Records! \n", arraysize);
    printf("---------------------\n");
    for(int i = 0; i < arraysize; i++)
    {
        print_car_helper(&cars[i]); // Print out the car details
    }
    cout << endl;
}
```

```
/**
 * Sorts the records in ascending order based on the make field,
 * using bubble sort
 */
void CarRecords::sort_cars_by_make()
{
  cout << "SORTING RECORDS BY MAKE....." << endl << endl;

  Car swap; // Temp

  // Bubble sort the cars based on make
  for (int i = 0 ; i < ( arraysize - 1 ); i++)
  {
    for (int j = 0 ; j < arraysize - i - 1; j++)
    {
      if (cars[j].getMake().compare(cars[j+1].getMake()) > 0)
      {
        swap       = cars[j];
        cars[j]   = cars[j+1];
        cars[j+1] = swap;
      }
    }
  }
}
```

```
/**
 * Sorts the records in descending order based on the year field
 */
void CarRecords::sort_cars_by_year()
{

  cout << "SORTING RECORDS BY YEAR....." << endl << endl;

  Car swap; // Temp

  // Bubble sort the cars based on year
  for (int i = 0 ; i < ( arraysize - 1 ); i++)
  {
    for (int j = 0 ; j < arraysize - i - 1; j++)
    {
      if (cars[j].getYear() > cars[j+1].getYear())
      {
        swap       = cars[j];
        cars[j]   = cars[j+1];
        cars[j+1] = swap;
      }
    }
  }
}
```

```
/**
 * Identifies any repeated records, and prints them out when
 * found. Repeated records means that all the fields are the
 * same
 */
void CarRecords::print_duplicates() {
    cout << "CHECKING FOR DUPLICATES..." << endl;

    Car swap; // Temp

    // Finds duplicates using a bubble-sort like implementation
    for (int i = 0 ; i < arraysize - 1; i++)
    {
        for (int j = i + 1 ; j < arraysize; j++)
        {
            if (cars[i].getMake() == cars[j].getMake() &&
                    cars[i].getModel() == cars[j].getModel() &&
                    cars[i].getYear() == cars[j].getYear() &&
                    cars[i].getColor() == cars[j].getColor())
            {
                print_car_helper(&cars[i]); // Print out the car details
                print_car_helper(&cars[j]); // Print out the car details
            }
        }
    }
}
```

```
/**
 * Main funciton
 *
 * @return integer, 0 no error, else error
 */
int main() {
    int numRecs;
    cout << "Number or Records to read? " ;
    cin >> numRecs;
    cout << endl;

    CarRecords *cr = new CarRecords(numRecs);
    // Print car records
    cr->printCarRecords();
    // Sort by Year
    cr->sort_cars_by_year();
    // Print car records
    cr->printCarRecords();
    // Sort by Make
    cr->sort_cars_by_make();
    // Print car records
    cr->printCarRecords();
    // Check for Duplicates
    cr->print_duplicates();
    delete cr;
} // end main
```

# 1. Running the Code

```
Linok-2 :: homeworks/hw2/q1 » g++ CarRecords.cpp -o CarRecords
Linok-2 :: homeworks/hw2/q1 » ./CarRecords
Number or Records to read? 15

PRINTING 10 Records!
--------------------
Subaru, Outback, 2016, green
Toyota, Corolla, 2006, white
Dodge, Neon, 1993, pink
Ford, Fusion, 2013, yellow
Honda, Fit, 2015, blue
Ford, Expedition, 2009, silver
Toyota, Corolla, 2006, white
Ford, Fusion, 2013, yellow
Jeep, Cherokee, 1999, red
Mazda, Protoge, 1996, gold

SORTING RECORDS BY YEAR.....

PRINTING 10 Records!
--------------------
Dodge, Neon, 1993, pink
Mazda, Protoge, 1996, gold
Jeep, Cherokee, 1999, red
Toyota, Corolla, 2006, white
Toyota, Corolla, 2006, white
Ford, Expedition, 2009, silver
Ford, Fusion, 2013, yellow
Ford, Fusion, 2013, yellow
Honda, Fit, 2015, blue
Subaru, Outback, 2016, green
```

```
Subaru, Outback, 2016, green

SORTING RECORDS BY MAKE.....

PRINTING 10 Records!
--------------------
Dodge, Neon, 1993, pink
Ford, Expedition, 2009, silver
Ford, Fusion, 2013, yellow
Ford, Fusion, 2013, yellow
Honda, Fit, 2015, blue
Jeep, Cherokee, 1999, red
Mazda, Protoge, 1996, gold
Subaru, Outback, 2016, green
Toyota, Corolla, 2006, white
Toyota, Corolla, 2006, white

CHECKING FOR DUPLICATES...
Ford, Fusion, 2013, yellow
Ford, Fusion, 2013, yellow
Toyota, Corolla, 2006, white
Toyota, Corolla, 2006, white
Linok-2 :: homeworks/hw2/q1 »
```

# 2. Code

```
GCC = g++
CFLAGS = -g -Wall
OBJS = CalcMain.o Calculator.o
EXE = Calculator

$(EXE): $(OBJS)
    $(GCC) $(OBJS) -o $(EXE)

Calculator.o: Calculator.cpp Calculator.h
    $(GCC) $(CFLAGS) -c Calculator.cpp

CalcMain.o: CalcMain.cpp Calculator.h
    $(GCC) $(CFLAGS) -c CalcMain.cpp

clean:
    rm $(OBJS) $(EXE)
```

```
#ifndef _CALCULATOR_H_
#define _CALCULATOR_H_

// A templated class representing a calculator
template <class T>
class Calculator {
    private:
        T value1;
        T value2;
    public:
        Calculator(); // Initializes values to default
        Calculator(T value1, T value2); //set values
        T getValue1(); // Returns value1
        T getValue2(); // Returns value2
        T getSum(); // Uses arithmetic operator '+'
        int getLogicalAND(); // Uses logical operator '&&'
        bool isGreater(); // Uses Relational operator '>'
};

#endif
```

```
#include "Calculator.h"

/**
 * Initializes values to default
 */
template <class T>
Calculator<T>::Calculator() {
    value1 = 0;
    value2 = 0;
}

/**
 * Set values
 */
template <class T>
Calculator<T>::Calculator(T value1, T value2) {
    this->value1 = value1;
    this->value2 = value2;
}

/**
 * Returns value1
 */
template <class T>
T Calculator<T>::getValue1() {
    return value1;
}

/**
 * Returns value2
 */
template <class T>
T Calculator<T>::getValue2() {
    return value2;
}
```

```
/**
 * Sum of the two templated values
 *
 * @return T, sum of the templated values
 */
template <class T>
T Calculator<T>::getSum() {
    return value1 + value2;
}

/**
 * Logical AND of the two templated values
 *
 * @return an integer, 0 false and 1 true
 */
template <class T>
int Calculator<T>::getLogicalAND() {
    return value1 && value2;
}

/**
 * Greater than of the two templated values
 *
 * @return a boolean, true or false
 */
template <class T>
bool Calculator<T>::isGreater() {
    return value1 > value2;
}
```

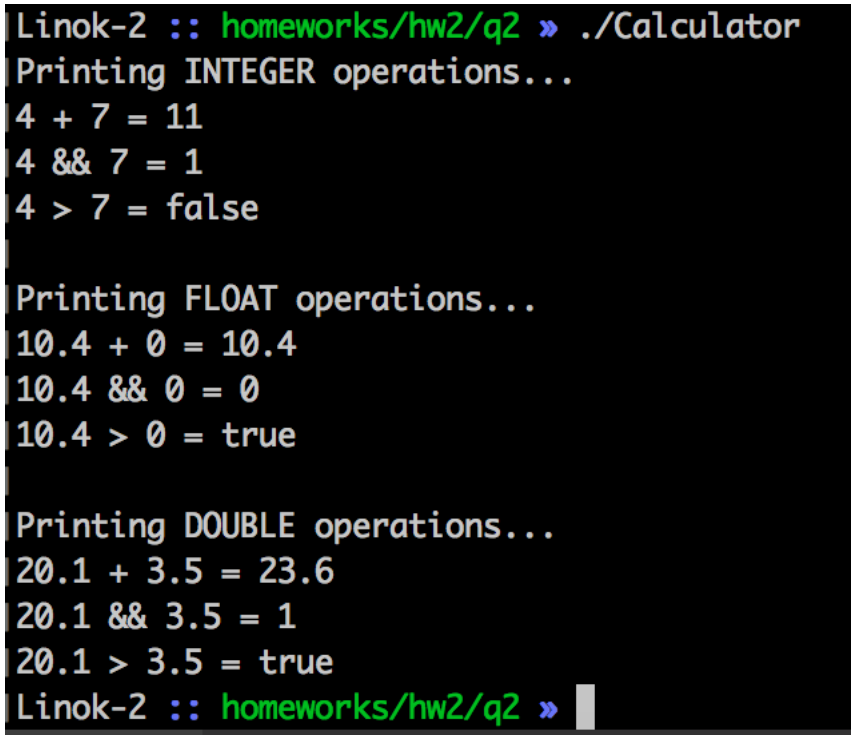
```
#include <iostream>
using std::cout;
using std::endl;

#include "Calculator.h"
#include "Calculator.cpp"

// A class representing the arithmetic
class Arithmetic {
    private:
        int intData;
        float floatData;
        double doubleData;

        /** A template function local to this class. The function
        * can be called with different types of objects
        */
        template <class T2>
        void printOperations(T2 obj);
    public:
        Arithmetic(); // A constructor initializes data members to zero
        Arithmetic(int i, float f, double d); // Sets data members to values passed
        void intOperations(Arithmetic obj); // Tests template class on ints
        void floatOperations(Arithmetic obj); // Tests template class on floats
        void doubleOperations(Arithmetic obj); // Tests template class on doubles
};

/**
 * A constructor initializes data members to zero
 */
Arithmetic::Arithmetic() {
    intData = 0;
    floatData = 0;
    doubleData = 0;
}
```

```
/**
 * Another constructor sets data members to values
 * passed
 *
 * @param i, represent the integer
 * @param f, represents the float
 * @param d, represents the double
 */
Arithmetic::Arithmetic(int i, float f, double d) {
    intData = i;
    floatData = f;
    doubleData = d;
}

/**
 * A template function local to this class. The function can be called
 * with different types of objects, and it prints out the values and
 * calls getSum(), getLogicalAND(), and isGreater() functions from the
 * template class to test the 3 different types of operations, and print
 * the results from those operations as seen in the sample output below.
 *
 * @param T2, Calculator template object
 */
template <class T2>
void Arithmetic::printOperations(T2 obj) {
    cout << obj.getValue1() << " + " << obj.getValue2()
            << " = " << obj.getSum() << endl;
    cout << obj.getValue1() << " && " << obj.getValue2()
            << " = " << obj.getLogicalAND() << endl;
    cout << obj.getValue1() << " > " << obj.getValue2()
            << " = " << (obj.isGreater() ? "true" : "false") << endl;
}
```

```
/**
 * Creates an object of type int from the template class while setting
 * the int values for the two objects, and calls printOperations(T2 obj)
 * function to test the 3 different types of operations
 *
 * @param obj, an Arithmetic object
 */
void Arithmetic::intOperations(Arithmetic obj) {
    Calculator<int> calc = Calculator<int>(this->intData, obj.intData);
    this->printOperations(calc);
}

/**
 * Creates an object of type float from the template class while setting
 * the float values for the two objects, and calls printOperations(T2 obj)
 * function to test the 3 different types of operations
 *
 * @param obj, an Arithmetic object
 */
void Arithmetic::floatOperations(Arithmetic obj) {
    Calculator<float> calc = Calculator<float>(this->floatData, obj.floatData);
    this->printOperations(calc);
}

/**
 * Creates an object of type double from the template class while setting
 * the double values for the two objects, and calls printOperations(T2 obj)
 * function to test the 3 different types of operations
 *
 * @param obj, an Arithmetic object
 */
void Arithmetic::doubleOperations(Arithmetic obj) {
    Calculator<double> calc = Calculator<double>(this->doubleData, obj.doubleData);
    this->printOperations(calc);
}
```

```
/**
 * The main function
 */
int main()
{
    // Create 1st object
    int int1 = 4;
    float f1 = 10.4;
    double d1 = 20.1;
    Arithmetic object1(int1, f1, d1);
    // Create 2nd object
    int int2 = 7;
    float f2 = 0.0;
    double d2 = 3.5;
    Arithmetic object2(int2, f2, d2);
    // Check operation on integer data type
    cout << "Printing INTEGER operations..." << endl;
    object1.intOperations(object2);
    // Check operation on float data type
    cout << "\nPrinting FLOAT operations..." << endl;
    object1.floatOperations(object2);
    // Check operation on double data type
    cout << "\nPrinting DOUBLE operations..." << endl;
    object1.doubleOperations(object2);
} // end main
```

# 2. Running the Code



# 3. Code

```
 1 GCC = g++
 0 CFLAGS = -g -Wall
 1 OBJS = main.o Furniture.o Table.o Bed.o
 2 EXE = Furniture
 3
 4 $(EXE): $(OBJS)
 5     $(GCC) $(OBJS) -o $(EXE)
 6
 7 main.o: main.cpp Furniture.h Table.h Bed.h
 8     $(GCC) $(CFLAGS) -c Main.cpp
 9
10 Furniture.o: Furniture.cpp Furniture.h
11     $(GCC) $(CFLAGS) -c Furniture.cpp
12
13 Table.o: Table.cpp Table.h
14     $(GCC) $(CFLAGS) -c Table.cpp
15
16 Bed.o: Bed.cpp Bed.h
17     $(GCC) $(CFLAGS) -c Bed.cpp
18
19 clean:
20     rm $(OBJS) $(EXE)
~
```

```
#ifndef _FURNITURE_H_
#define _FURNITURE_H_

#include <string>
using std::string;

class Furniture {
    private:
        /**
         * Three private floating-point numbers indicating the dimensions of
         * the piece of furniture
        */
        float width;
        float height;
        float depth;
        string unique_name; // A private string containing a unique name
    public:
        /**
         * A public constructor that takes the name of the piece of furniture
         * as an argument
         */
        Furniture(string name);

        // Used to read the values of the width, height, and depth
        void ReadDimensions();

        // Print the dimensions and name
        virtual void Print();
};

#endif
```

```
#ifndef _TABLE_H_
#define _TABLE_H_

#include <string>
using std::string;

#include "furniture.h"

class Table: public Furniture {
    private:
        string wood_type; // A private string indicating the type of wood
    public:
        Table(string unique_name, string bed_size); // A public constructor
        void Print(); // Print table information
};

#endif
```

```
#ifndef _BED_H_
#define _BED_H_

#include <string>
using std::string;

#include "furniture.h"

class Bed: public Furniture {
    private:
        string bed_size; // Specifying the size of the bed
    public:
        Bed(string unique_name, string bt); // Public constructor
        void Print(); // Prints bed information
};

#endif
```

```cpp
#include <string>
using std::string;

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "Bed.h"

/**
 * A public constructor that takes the unique name of the bed as the first
 * argument, passed directly to the constructor of the parent class, and
 * the string corresponding to the bed size as the second argument. The
 * constructor should print an error message if an invalid size string is
 * passed.
 *
 * @param unique_name, a string representing the name of the bed
 * @param, bs, represents the bed size
 */
Bed::Bed(string unique_name, string bs) : Furniture(unique_name)
{
    if((bs.length() == 4 || bs.length() == 5) &&
        (bs == "Twin" || bs == "Full" || bs == "Queen" || bs == "King")) {
        bed_size = bs;
    } else {
        cout << "ERROR: The bed size is not the right length, or ";
        cout << "it is not a Twin, Full, Queen, or King." << endl;
    }
}

/**
 * A public function Print() that overrides the function with the same name in
 * the parent class. This function prints common furniture information by invoking
 * the parent class function, and continues printing information specific to a bed
 * (the size string).
 */
void Bed::Print()
{
    Furniture::Print();
    cout << "    " << bed_size << " size" << endl;
}
```

```cpp
#include <string>
using std::string;

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "Table.h"

/**
 * A public constructor that takes the unique name of the table as the first
 * argument, passed directly to the constructor of the parent class, and the
 * string corresponding to the wood type as the second argument. The constructor
 * should print an error message if an invalid size string is passed.
 *
 * @param unique_name, name of the table
 * @param wt, wood type of the table
 */
Table::Table(string unique_name, string wt) : Furniture(unique_name)
{
    if((wt.length() == 3 || wt.length() == 4) &&
        (wt == "Pine" || wt == "Oak")) { // If equal to length and strings
        wood_type = wt;
    } else
    {
        cout << "ERROR: The wood type is not the right length, or ";
        cout << "it is not an Oak or Pine woodtype." << endl;
    }
}

/**
 * A public function Print() that overrides the function with the same
 * name in the parent class. This function prints common furniture
 * information by invoking the parent class function, and continues
 * printing information specific to a table (the wood type).
 */
void Table::Print()
{
    Furniture::Print();
    cout << "    " << wood_type << " wood" << endl;
}
```

```cpp
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "Furniture.h"

/**
 * A public constructor that takes the name of the piece of furniture as an
 * argument
 *
 * @param name, name of the furniture
 */
Furniture::Furniture(string name)
{
    unique_name = name;
}
```

```
/**
 * A public function ReadDimentions, used to read the values of the width,
 * height, and depth from the keyboard. The function should show an error
 * message if any of the entered values is less than 0.
 */
void Furniture::ReadDimensions()
{
    cout << "   Enter width: ";
    cin >> width;
    while(width < 0) {
        cout << "ERROR: Please enter a width above 0." << endl;
        cout << "   Enter width: ";
        cin >> width;
    }

    cout << "   Enter height: ";
    cin >> height;
    while(height < 0) {
        cout << "ERROR: Please enter a height above 0." << endl;
        cout << "   Enter height: ";
        cin >> height;
    }

    cout << "   Enter depth: ";
    cin >> depth;
    while(depth < 0) {
        cout << "ERROR: Please enter a depth above 0." << endl;
        cout << "   Enter depth: ";
        cin >> depth;
    }
}

/**
 * A public virtual function Print(), with no arguments and a void return value.
 * This function should print the dimensions and name
 */
void Furniture::Print()
{
    cout << unique_name << ":" << endl;
    cout << "   Width = " << width << ", height = " << height <<
            ", depth = " << depth << endl;
}
```

```cpp
#include "Table.h"
#include "Bed.h"

#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
    string un, t;

    cout << "Creating table..." << endl;
    cout << "   Enter name: ";
    cin >> un;
    cout << "   Enter wood type (Pine, Oak): ";
    cin >> t;
    Table table = Table(un, t);
    table.ReadDimensions();

    cout << "Creating bed..." << endl;
    cout << "   Enter name: ";
    cin >> un;
    cout << "   Enter size (Twin, Full, Queen, King): ";
    cin >> t;
    Bed bed = Bed(un, t);
    bed.ReadDimensions();

    cout << endl << "Printing objects ..." << endl << endl;
    table.Print();
    bed.Print();
}
```

# 3. Running the Code

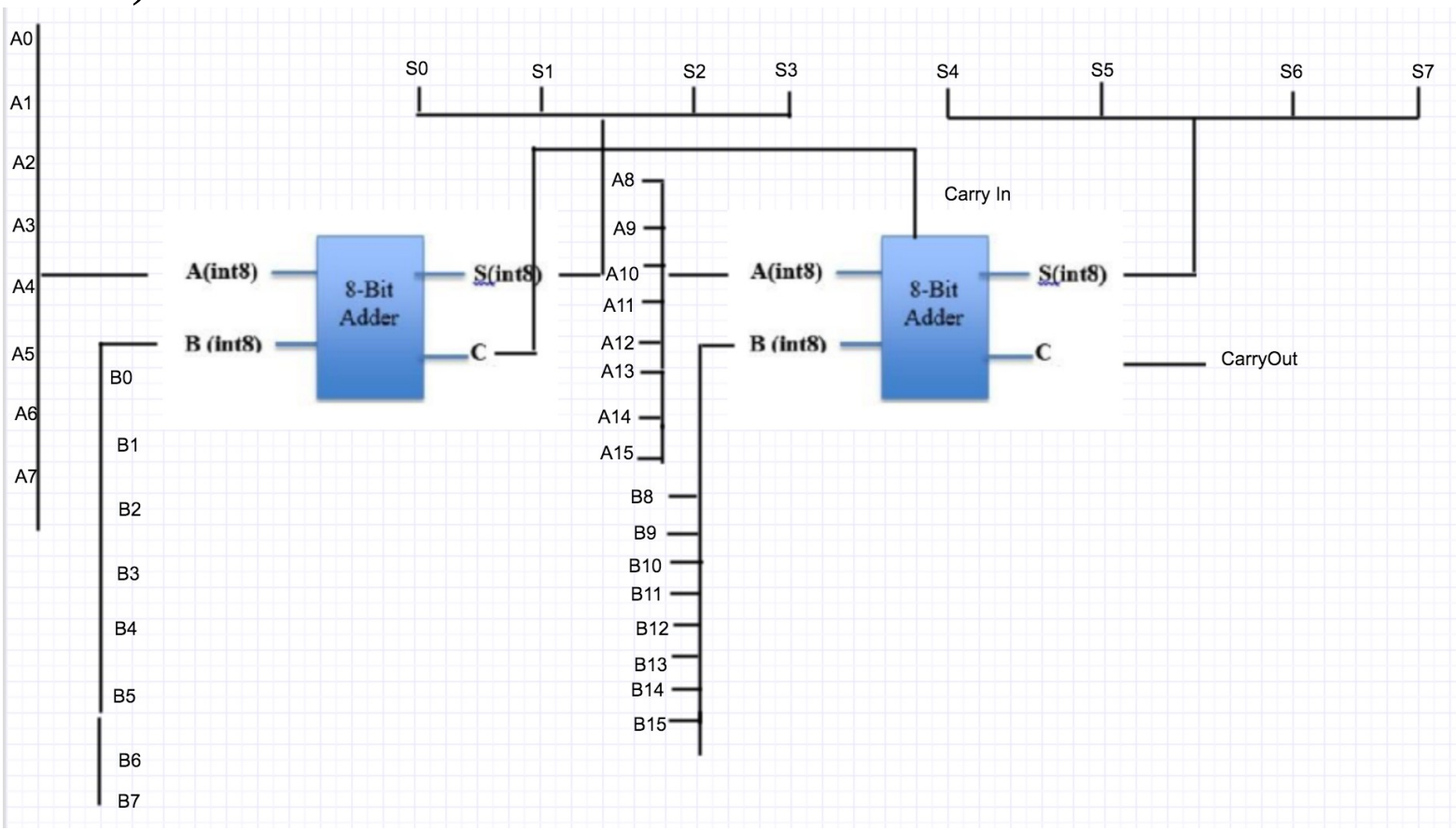
```
Linok-2 :: homeworks/hw2/q3 » make
make: `Furniture' is up to date.
Linok-2 :: homeworks/hw2/q3 » ./Furniture
Creating table...
   Enter name: MyTable
   Enter wood type (Pine, Oak): Pine
   Enter width: 1
   Enter height: 2
   Enter depth: 3
Creating bed...
   Enter name: MyBed
   Enter size (Twin, Full, Queen, King): Queen
   Enter width: 5
   Enter height: 6
   Enter depth: 7

Printing objects ...

MyTable:
   Width = 1, height = 2, depth = 3
   Pine wood
MyBed:
   Width = 5, height = 6, depth = 7
   Queen size
Linok-2 :: homeworks/hw2/q3 »
```

```
   Enter wood type (Pine, Oak): asdfsad
ERROR: The wood type is not the right length, or it is not an Oak or Pine woodtype.
   Enter width:
```

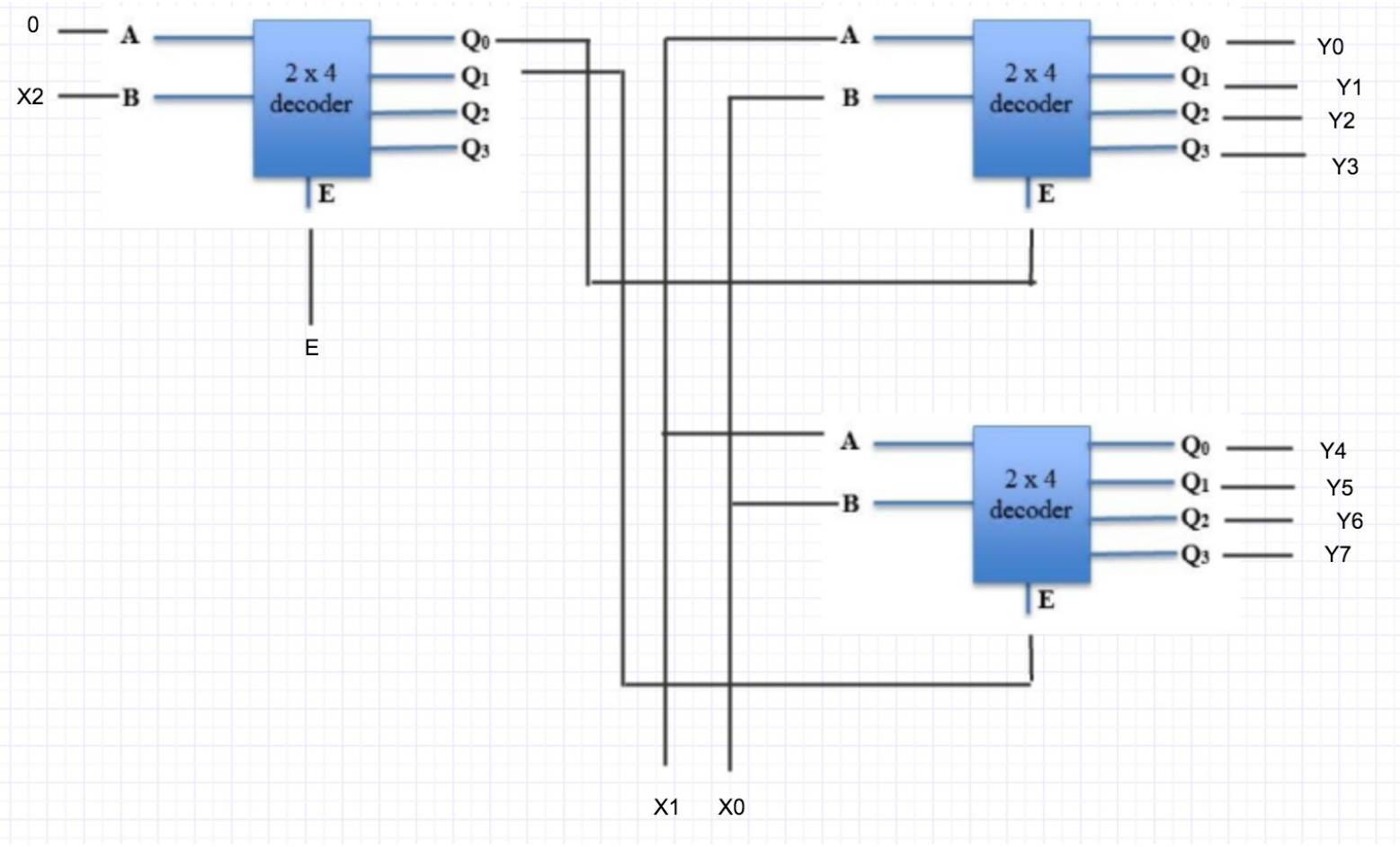
```
   Enter size (Twin, Full, Queen, King): sdf
ERROR: The bed size is not the right length, or it is not a Twin, Full, Queen, or King.
   Enter width:
```

# 4.

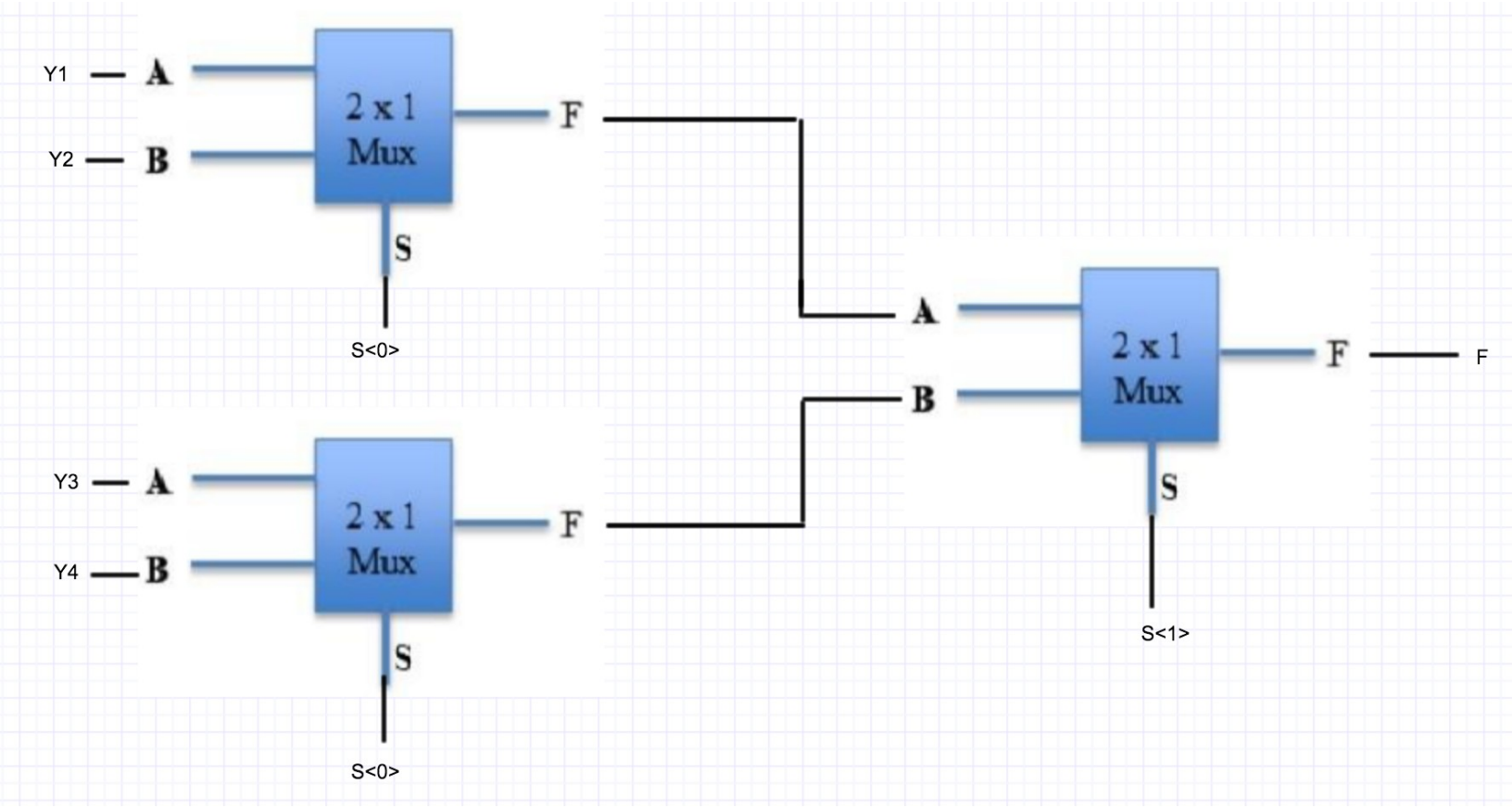
a).


A0
A1
A2
A3
A4
A5
A6
A7
B0
B1
B2
B3
B4
B5
B6
B7
S0
S1
S2
S3
S4
S5
S6
S7
A8
A9
A10
A11
A12
A13
A14
A15
B8
B9
B10
B11
B12
B13
B14
B15
Carry In
A(int8)
B (int8)
8-Bit Adder
S(int8)
C
A(int8)
B (int8)
8-Bit Adder
S(int8)
C
CarryOut

b).

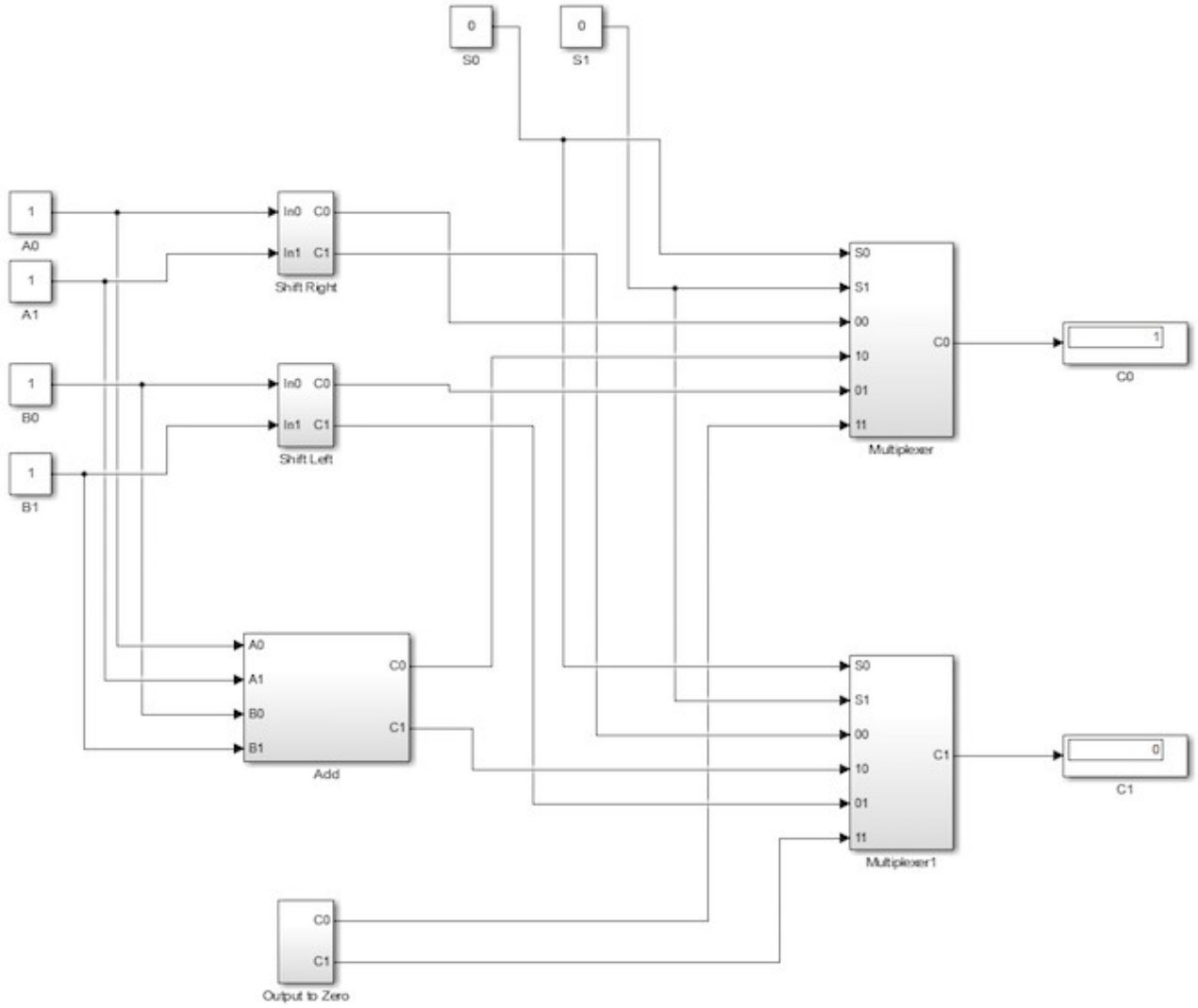

0
A
X2
B
2 x 4 decoder
Q0
Q1
Q2
Q3
E
E
A
B
2 x 4 decoder
Q0
Q1
Q2
Q3
Y0
Y1
Y2
Y3
E
A
B
2 x 4 decoder
Q0
Q1
Q2
Q3
Y4
Y5
Y6
Y7
E
X1
X0

c).

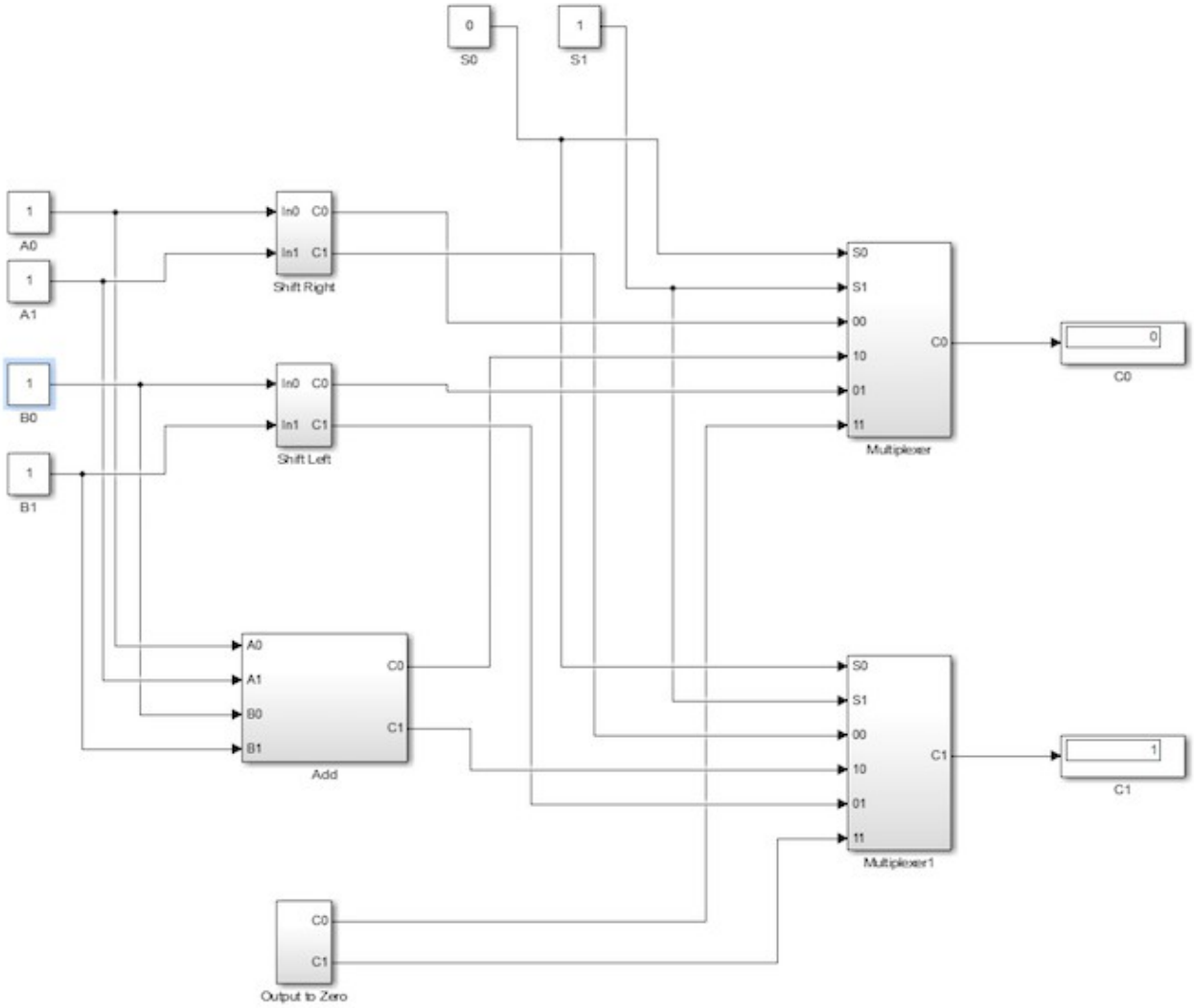


# 5.

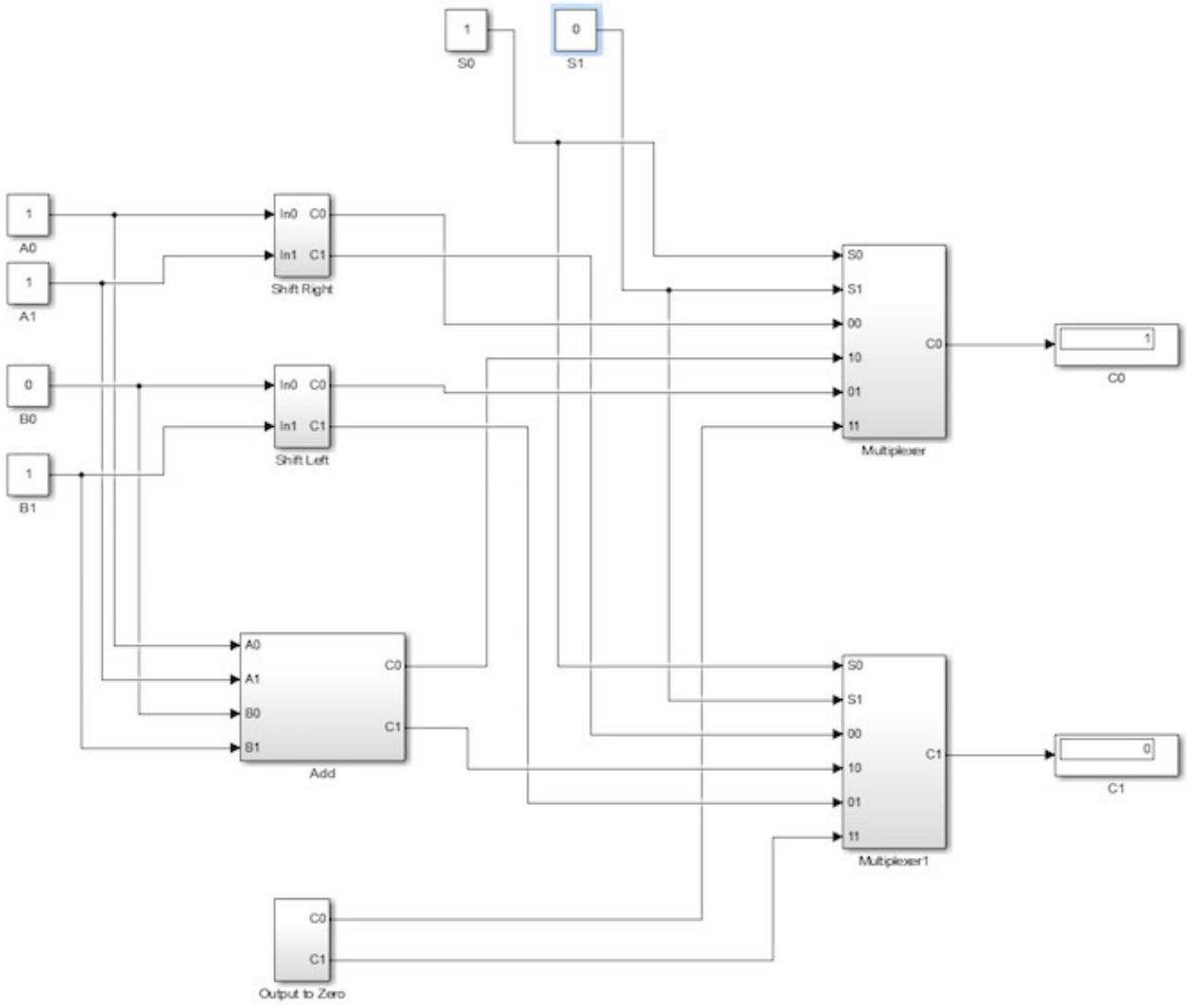
# Testing right shift:



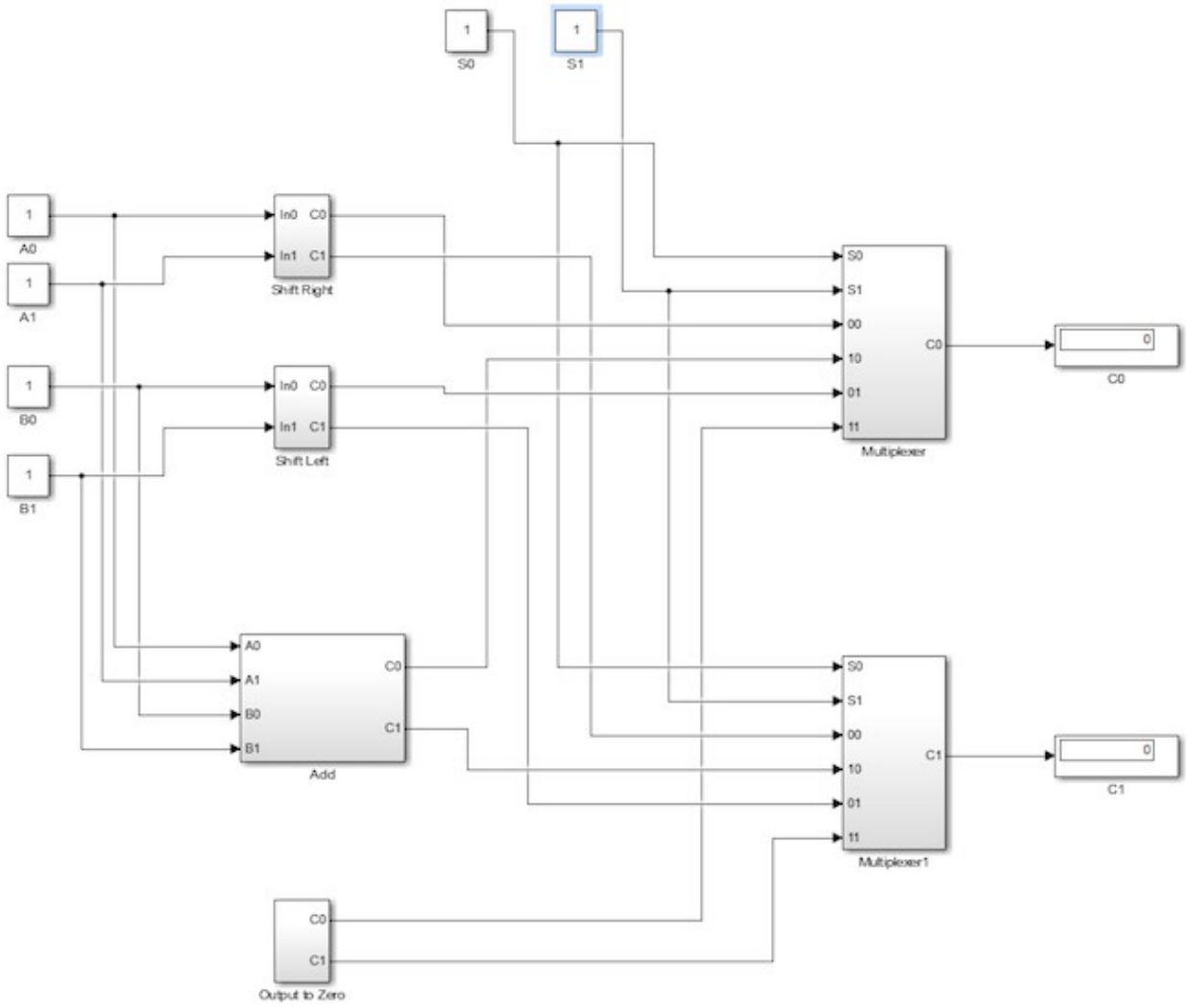
# Testing left shift:



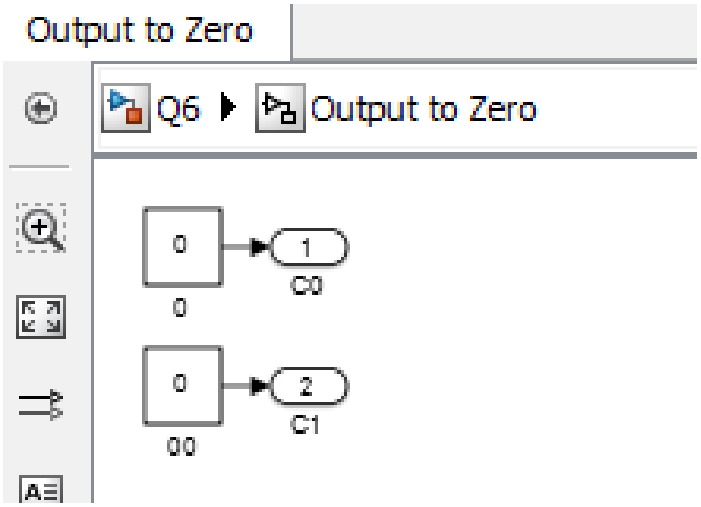
# Testing add:

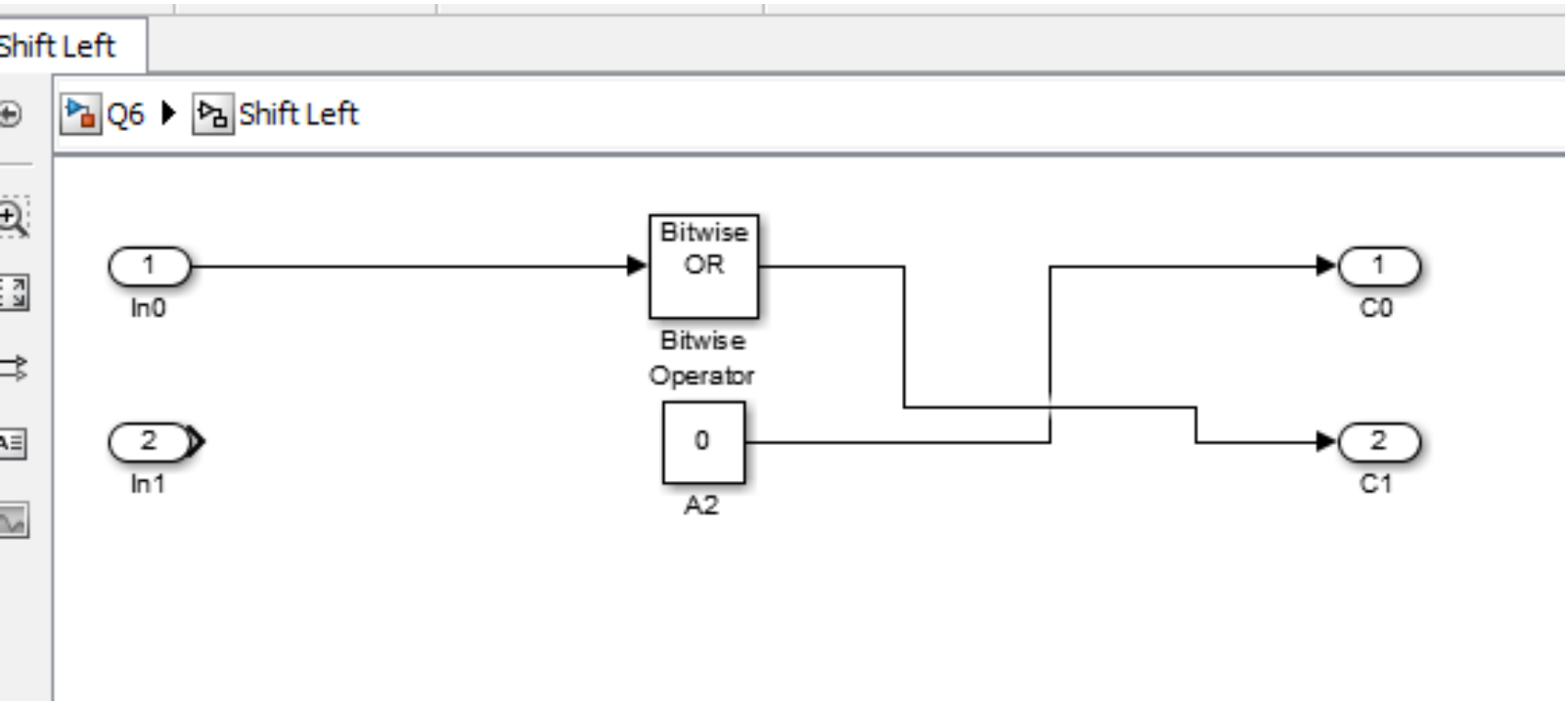

1
S0
0
S1
1
A0
1
A1
0
B0
1
B1
In0 C0
In1 C1
Shift Right
In0 C0
In1 C1
Shift Left
A0
A1
B0
B1
C0
C1
Add
S0
S1
00
10
01
11
C0
Multiplexer
1
C0
S0
S1
00
10
01
11
C1
Multiplexer1
0
C1
C0
C1
Output to Zero

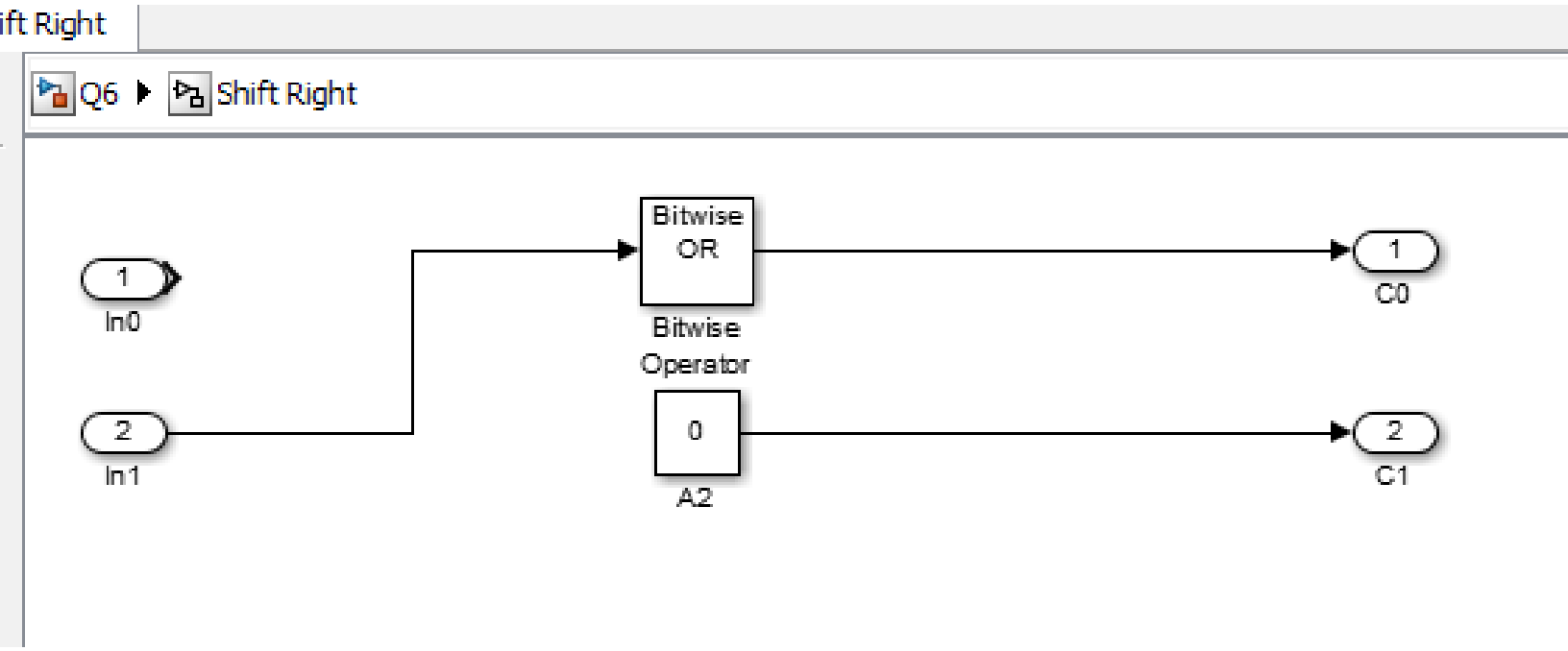
# Testing Zeroing:

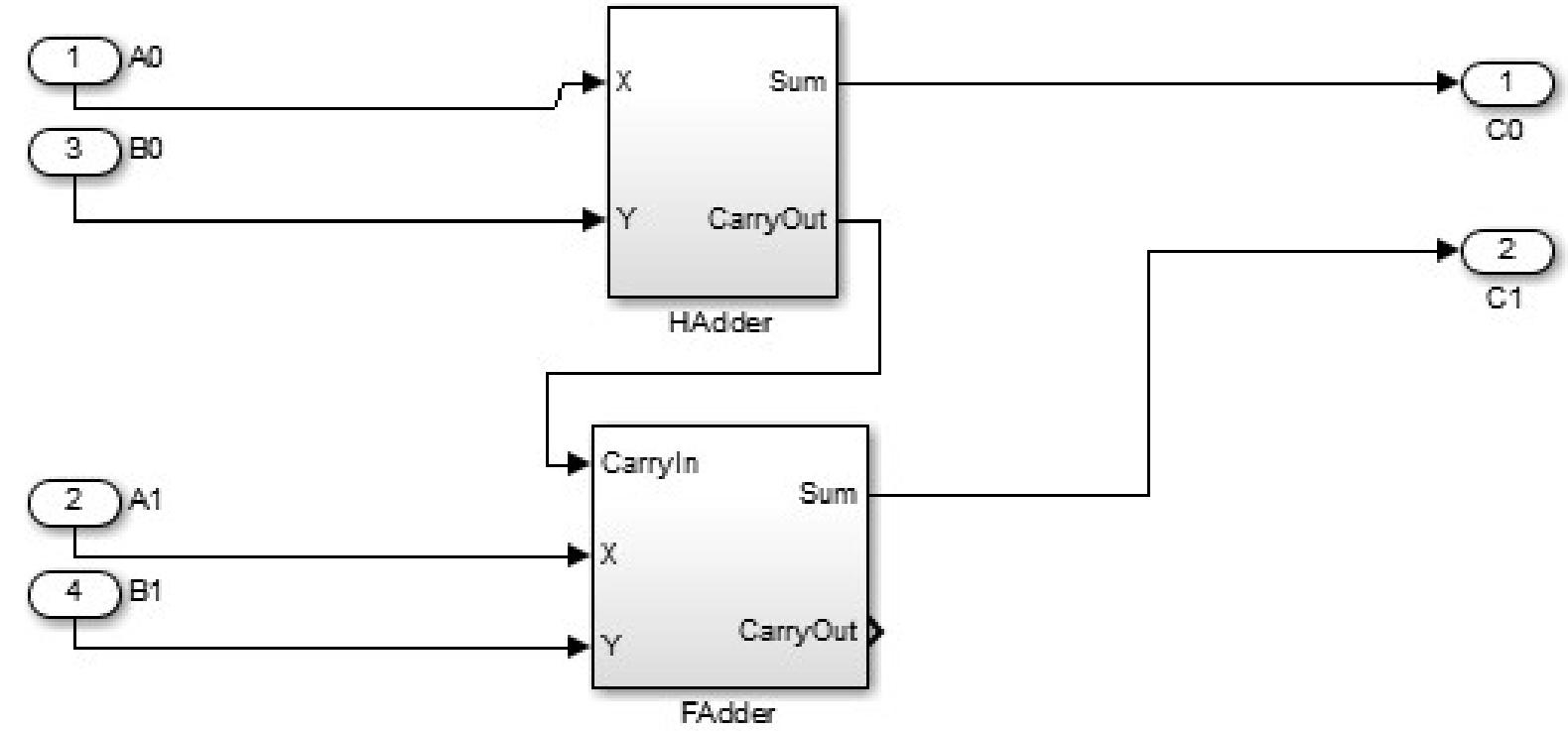


# Pictures of all the subsystems:




ift Right
Q6
Shift Right
1
In0
2
In1
Bitwise
OR
Bitwise
Operator
0
A2
1
C0
2
C1


1
A0
3
B0
X
Y
Sum
CarryOut
HAdder
1
C0
2
C1
2
A1
4
B1
CarryIn
X
Y
Sum
CarryOut
FAdder