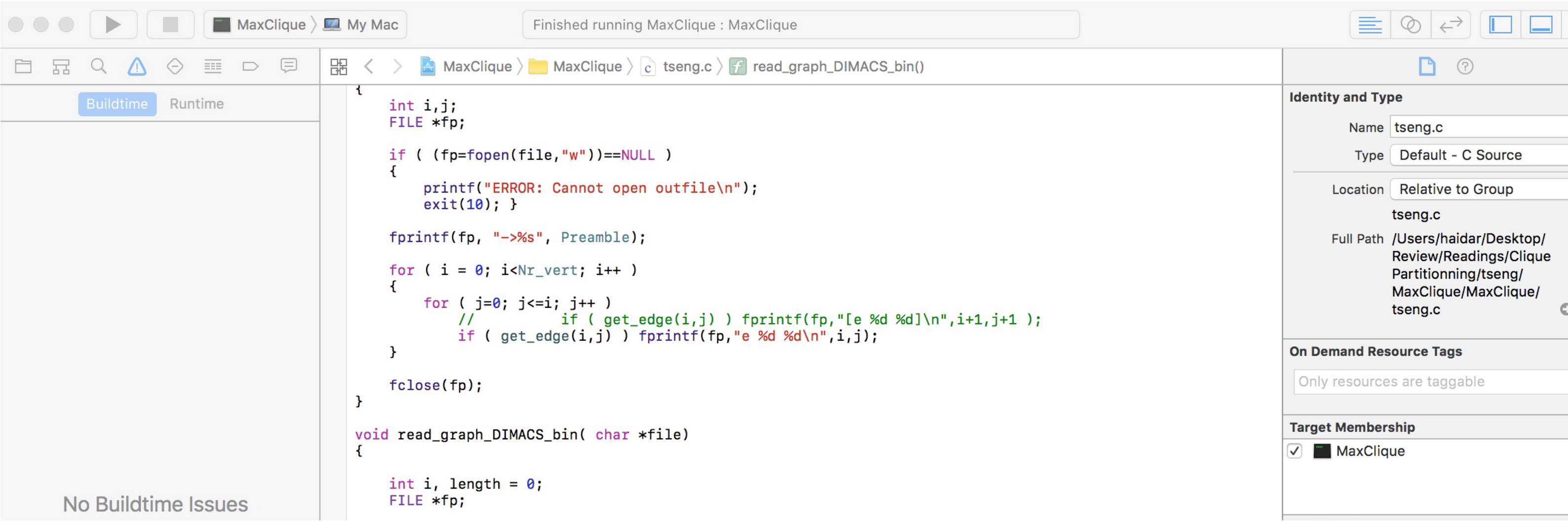
Hands-On Computer Organization Lab

With C and Verilog

Haidar M. Harmanani

First printing, March 2016


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

```
{
    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

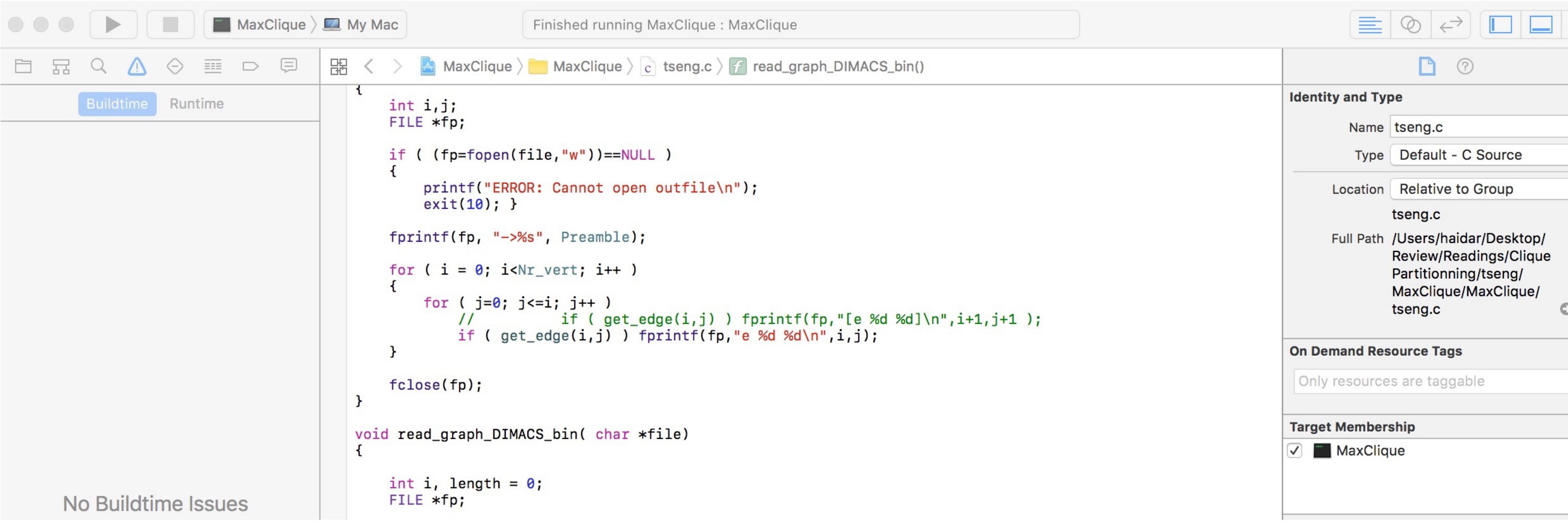
On Demand Resource Tags

Only resources are taggable

Target Membership

✓ MaxClique

Contents



```
{
    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. Lab 2: Bits and Bytes

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

—Craig Bruce

Computer `byte` is the basic unit that is used to represent and store data in a digital computer., and typically consists of 8 adjacent binary bits. Although there is no "bit" type, it is possible to manipulate individual bits using bitwise operators. In C and C++, binary numbers start from the most significant bit to the left (i.e., `10000000` is 128, and `00000001` is 1).

1.1 Bitwise Operations

1.1.1 Bitwise Shift

A `left shift` operation is the equivalent of moving all the bits of a number a specified number of places to the left, and is equivalent to multiplying by a power of two. For instance, consider the number 8 written in binary `00001000`. If we wanted to shift it to the left by 2 positions, we would end up with `00100000`; everything is moved to the left two places, and zeros are added as padding. This is the number 8×2^2.

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

Note that in the above example we used 8 bit numbers, which are used to represent characters. Integers are represented using 4 bytes while short integers are represented using 2 bytes. But what happens if we shift a number beyond the last bit? That is, shifting `10000000` on position left? In this case, we usually end-up with a `00000000` as the last bit would drop out.

It shouldn't be a surprise that there's a corresponding right-shift operator: $>>$, which would correspond in this case to an integer division by 2. For example, shifting 5, `00000101`, left would result with `00000010`, which is 2. Note that this only holds true for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s. Generally, using the left and right shift operators will result in significantly faster code than calculating and then multiplying by a power of two. The shift operators will also be useful later when we look at how to manipulate individual bits. For now, let's look at some of the other binary operators to see what they can do for us.

Exercise 1.1 Write a c program and declare a character to be equal to -1. Shift it left by 1. What do you expect the answer to be and what do you observe? Explain what happened. ■

1.1.2 Bitwise AND

The bitwise AND operator is represented in C by a single ampersand, &. A binary AND simply takes the logical AND of the bits in each position of a number in binary form. For instance, working with a byte (the char type):

```
01001000 & 10111000 = 00001000
```

The most significant bit of the first number is 0, so we know the most significant bit of the result must be 0; in the second most significant bit, the bit of second number is zero, so we have the same result. The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left. Consequently,

```
72 & 184 = 8
```

A bitwise AND is a basic logical circuit that can be mathematically represented using a truth table that show the outputs that correspond to all input combination. The truth table of the AND function is:[1]

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

1.1.3 Bitwise OR

Bitwise OR works in a similar way to a bitwise AND, except that the result is 1 if either (or both) of the two inputs is 1. The OR function is represented using a pipe: |. For example,

```
01001000 | 10111000 = 11111000
```

Notice that the output is 1 only when either bit is 1. Consequently,

```
72 | 184 = 248
```

The truth table for an OR function is:

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

[1]Check the following truth table generator for illustration purposes: https://web.stanford.edu/class/cs103/tools/truth-table-tool/

1.1.4 The Bitwise Complement or Negation

The bitwise complement operator, the tilde, ∼, flips every bit. The bitwise negation is a great way to find the largest possible value for an unsigned number:

```
unsigned int max = ∼ 0;
```

0, of course, is all 0s: `00000000 00000000`. Once we twiddle 0, we get all 1s: `11111111 11111111`. Since max is an unsigned int, we don't have to worry about sign bits or twos complement. We know that all 1s is the largest possible number.

1.1.5 Bitwise Exclusive-Or (XOR)

The exclusive-or 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 instance, if you have two numbers represented in binary as `10101010` and `01110010` then taking the bitwise XOR results in `11011000`. It's easier to see this if the bits are lined up correctly:

```
01110010 ^ 10101010 = 11011000
```

The truth table for an XOR function is:

a	b	a ˆ b
0	0	0
0	1	1
1	0	1
1	1	0

1.2 Putting Bits and Bytes to a Good Use

1.2.1 Example Representation

Let us see an example where the bit operators can perform something potentially useful. Assume you need to keep track of eight objects. One possibility would be to use eight variables, one per object. Another simple alternative would be to use to use bits to represent the objects. Since there are eight objects, all we really need is a single byte, and we'll use each of its eight bits to indicate whether or not an object is in use. We can declare our byte as follows:

```
char in_use = 0;
```

1.2.2 Checking a Specific Bit

To check if the particular object is free before we try to use it, we can isolate the corresponding bit in order to check if it is equal to 0 or 1. A simple strategy would consist of using the bitwise AND operator to set every bit to zero except for the bit we want to extract. For example, if we want to check the status of the fifth object, we can use bitwise AND with a mask, `00100000`. If the fifth bit is a 1 then the output would be equal to the mask as follows:

```
XX1XXXXX & 00100000 = 00100000
```

Whereas, if it's a zero, then the result will be `00000000`:

```
XX0XXXXX & 00100000 = 00000000
```

So we get a non-zero number if, and only if, the bit we're interested in is a 1. This procedure works for finding the bit in the n^{th} position. The only thing left to do is to create a number with only the one bit in the correct position turned on. These are just powers of two, so one approach might be to do something like:

```
int is_in_use(int object_num)
{
    // pow returns an int, but in_use will also be promoted to an int
    // so it doesn't have any effect; we can think of this as an
    // operation between chars

    return in_use & pow(2, object_num);
}
```

While this function works, it obscures the fact that what we want to do is shift a bit over a certain number of places. We can use a bitwise `leftshift` to accomplish this, and it'll be much faster to boot. If we start with the number 1, we are guaranteed to have only a single bit, and we know it's to the far-right. We'll keep in mind that object 0 will have its data stored in the rightmost bit, and object 7 will be the leftmost.

```
int is_in_use(int object_num)
{
    return in_use & 1<< object_num;
}
```

Note that shifting by zero places is a legal operation – we'll just get back the same number we started with. All we can do right now is check whether an object is in use; we can't actually set the in-use bit for it. There are two cases to consider: indicating a object is in use, and removing an object from use. In one case, we need to turn a bit on, and in the other, turn a bit off.

1.2.3 Setting a Specific Bit

In order to set or reset an object, we would need to set or reset the corresponding bit. One possible way to do so is to use a bitwise OR. Thus, if we perform a bitwise OR with only a single bit set to 1 (the rest are 0), then we won't affect the rest of the number because anything ORed with zero remains the same (1 OR 0 is 1, and 0 OR 0 is 0). Again we need to move a single bit into the correct position:

```
void set_in_use(int object_num)
{
    in_use = in_use | 1 << object_num;
}
```

What does this do? Assume we would like to set the rightmost bit in `0XXXXXXX` to 1. This can be done as follows: `0XXXXXXX` | `10000000`. The result is the original number with the rightmost bit set to 1, `1XXXXXXX`. The shift is the same as before.

1.2.4 Resetting a Specific Bit

Resetting an object to be no longer in use requires the use of negation and bitwise operators in two steps. In the first step, the number 1 is shifted right by the necessary number of bits and then negated as follows:

```
~(1<<position)
```

Now that we have this, we can just take the bitwise AND of this with the current field of objects, and the only bit we'll change is the one of the `object_num` we're interested in:

```
void set_unused(int object_num)
{
    in_use = in_use & ~(1<< object_num);
}
```

1.2.5 Flipping a Specific Bit

The above setting or resetting a bit requires a prior knowledge regarding the object in use (if the bit is on or off) before we can know which function to call. While this isn't necessarily a bad thing, it means that we do need to know a little bit about what's going on. There is an easier way using the exclusive-or operator. Recall that XORing a bit with 0 results in the same bit. So what we'd really like to be able to do is just call one function that flips the bit of the object we're interested in – it doesn't matter if it's being turned on or turned off – and leaves the rest of the bits unchanged. This sounds an awful lot like the what we've done in the past; in fact, we only need to make one change to our function to turn a bit on. Instead of using a bitwise OR, we use a bitwise XOR. This leaves everything unchanged, but flips the bit instead of always turning it on:

Lab 1.1 Implement `get_bit`, `set_bit`, and `flip_bit` using the starter `bit_ops.c` source code. Please examine the code closely noting the program structure along with the test workbench. ■

Lab 1.2 Experiment with `bits and bytes` by writing a C program that counts how many 0's or 1's are in a given integer. ■

```
void flip_use_state(int object_num)
{
    in_use = in_use ^ 1 << object_num;
}
```

Tiny Processor

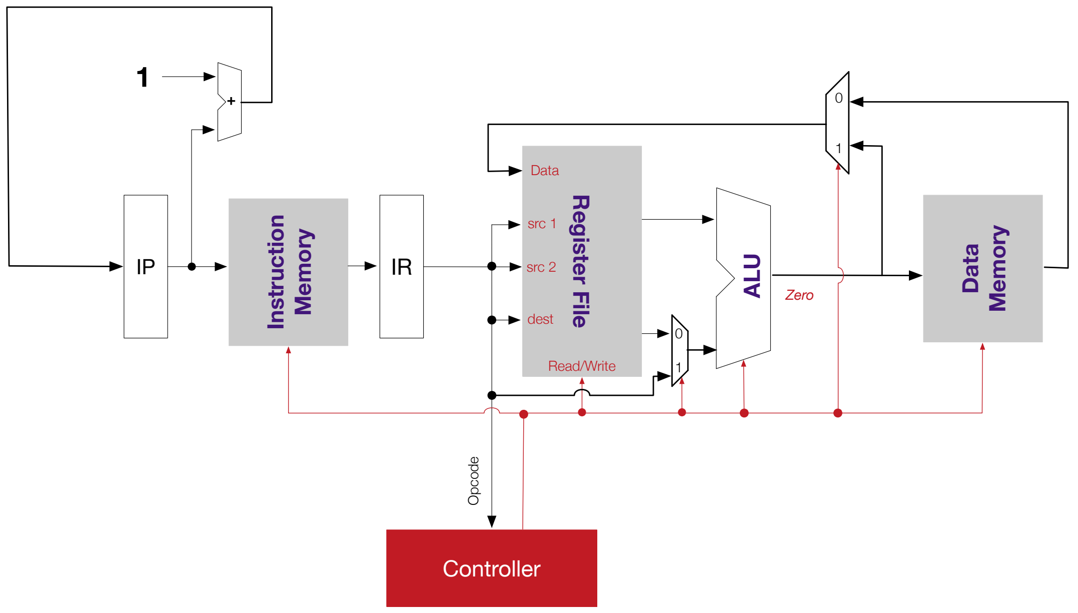


Figure 1.1: Datapath for the Tiny Processor

1.3 Modeling Architecture Using C

Lab 1.3 Although bitwise operators are good for saving space, space is hardly an issue. We will be using bits and bitwise operators in order to model the architecture of a simple processor, its instruction set, and its machine language. The datapath of the Tiny Processor is shown in Figure 1.1. In this design, the program data are stored in the Data Memory while the program that is being executed is stored in the Instruction Memory. The IR is a register that is used to hold the instruction that is being decoded and executed. The IP points to the next instruction to be executed. The PC or program counter is used to index the instructions in the program that is being executed. For now, research the literature and understand the architecture. Explain the design to your partner. Explore how the architecture can be modeled using C and what are the available options. Document all information in your report. ■