2.

```
/* File:     compare_mult.c
 * Purpose:  Compare the performance of the hardware multiplication
 *           algorithm with a proposed algorithm.
 *
 * Compile:  gcc -g -Wall -o compare_mult compare_mult.c
 * Run:      ./compare_mult <number of multiplications>
 *
 * Input:    None
 * Output:   Elapsed time for the multiplications done in hardware
 *           and the multiplications done with the proposed algorithm.
 *
 * Note:     The code for the proposed algorithm is just a stub:  it
 *           always returns 1.
 *
 *
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"

/* Largest value for one of the factors */
const int MAX = 10000;

int Hardware(int x, int y);
int Proposed(int x, int y);

int main(int argc, char* argv[]) {
   int iters, i, *x, *y, product;
   double start, finish, mult_elapsed, proposed_elapsed;

   if (argc != 2) {
      fprintf(stderr, "usage: %s <number of multiplies>\n",
            argv[0]);
      exit(0);
   }
   iters = strtol(argv[1], NULL, 10);
   x = malloc(iters*sizeof(int));
   y = malloc(iters*sizeof(int));

   /* Arrays of factors */
   srandom(1);
   for (i = 0; i < iters; i++) {
      x[i] = random() % MAX;
      y[i] = random() % MAX;
   }
```

```
  GET_TIME(start);
  for (i = 0; i < iters; i++)
    product = Hardware(x[i],y[i]);
  GET_TIME(finish);
  mult_elapsed = finish-start;

  GET_TIME(start);
  for (i = 0; i < iters; i++)  {
    product = Proposed(x[i], y[i]);
  }
  GET_TIME(finish);
  proposed_elapsed = finish-start;

  printf("Time for hardware = %e seconds\n", mult_elapsed);
  printf("Time for proposed = %e seconds\n", proposed_elapsed);

  free(x);
  free(y);

  return 0;
}

/*---------------------------------------------------------------
 * Function:  Hardware
 * Purpose:   Multiply two numbers and return their product
 */
int Hardware(int x, int y) {
  return x*y;
}  /* Mult */


/*---------------------------------------------------------------
 * Function:  Proposed
 * Purpose:   Multiply two numbers using a proposed algorithm and
 *            return their product
 */

int Proposed(int x, int y) {
  // multiplier and multiplicand >= 0
  int product = 0;
  for (int i = 0; i < y; i++)
    product += x;
  return product;
}  /* Proposed */
```

From running the program, we get:

```
yis-macbook-pro:h8 treexy1230$ ./compare_mult.c 100
Time for hardware = 9.536743e-07 seconds
Time for proposed = 1.610041e-03 seconds
yis-macbook-pro:h8 treexy1230$ ./compare_mult.c 200
Time for hardware = 2.145767e-06 seconds
Time for proposed = 3.407955e-03 seconds
yis-macbook-pro:h8 treexy1230$ ./compare_mult.c 500
Time for hardware = 2.861023e-06 seconds
Time for proposed = 8.466959e-03 seconds
yis-macbook-pro:h8 treexy1230$ ./compare_mult.c 1000
Time for hardware = 5.960464e-06 seconds
Time for proposed = 1.461601e-02 seconds
```

So the proposed algorithm run slower than the hardware multiply.
Suppose input x has m bits, y has n bits,
then for hardware multiply will have m*n iterations,
and for proposed algorithm , it will have m*y iterations.

3.12 Using a table similar to that shown in Figure 3.7, calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in Figure 3.4. You should show the contents of each register on each step.

unsigned 6-bit integers 62 = 110010
unsigned 6-bit integers 12 = 001010

| Iteration | Step | Multiplier | Multiplicand | Product |
|---|---|---|---|---|
| 0 | Initial values | 00101<u>0</u> | 0000 0011 0010 | 0000 0000 |
| 1 | 1:0->No operation | 001010 | 0000 0011 0010 | 0000 0000 |
| | 2: left shift Multiplicand | 001010 | **0000 0110 0100** | 0000 0000 |
| | 3: right shift Multiplier | **000101** | 0000 0110 0100 | 0000 0000 |
| 2 | 1:1->Prod =Prod+Mcand | 00010<u>1</u> | 0000 0110 0100 | **0000 0110 0100** |
| | 2: left shift Multiplicand | 000101 | **0000 1100 1000** | 0000 0110 0100 |
| | 3: right shift Multiplier | **000010** | 0000 1100 1000 | 0000 0110 0100 |
| 3 | 1:0->No operation | 00001<u>0</u> | 0000 1100 1000 | 0000 0110 0100 |
| | 2: left shift Multiplicand | 000010 | **0001 1001 0000** | 0000 |

0110 0100

3: right shift Multiplier **000001** 0001 1001 0000 0000

0110 0100

4 1:1->Prod =Prod+Mcand 00000*1* 0001 1001 0000 **0001**

**1111 0100** 2: left shift Multiplicand 000001 **0011 0010 0000**

0001 1111 0100

3: right shift Multiplier **000000** 0011 0010 0000 0001

1111 0100

5 1:0->No operation 00000*0* 0011 0010 0000 0001

1111 0100

2: left shift Multiplicand 000000 **0110 0100 0000** 0001

1111 0100

3: right shift Multiplier **000000** 0110 0100 0000 0001

1111 0100

6 1:0->No operation 00000*0* 0110 0100 0000 0001

1111 0100

2: left shift Multiplicand 000000 **1100 1000 0000** 0001

1111 0100

3: right shift Multiplier **000000** 1100 1000 0000 0001

1111 0100

Product = 0001 1111 0100 = 500(decimal) = 764(octal)