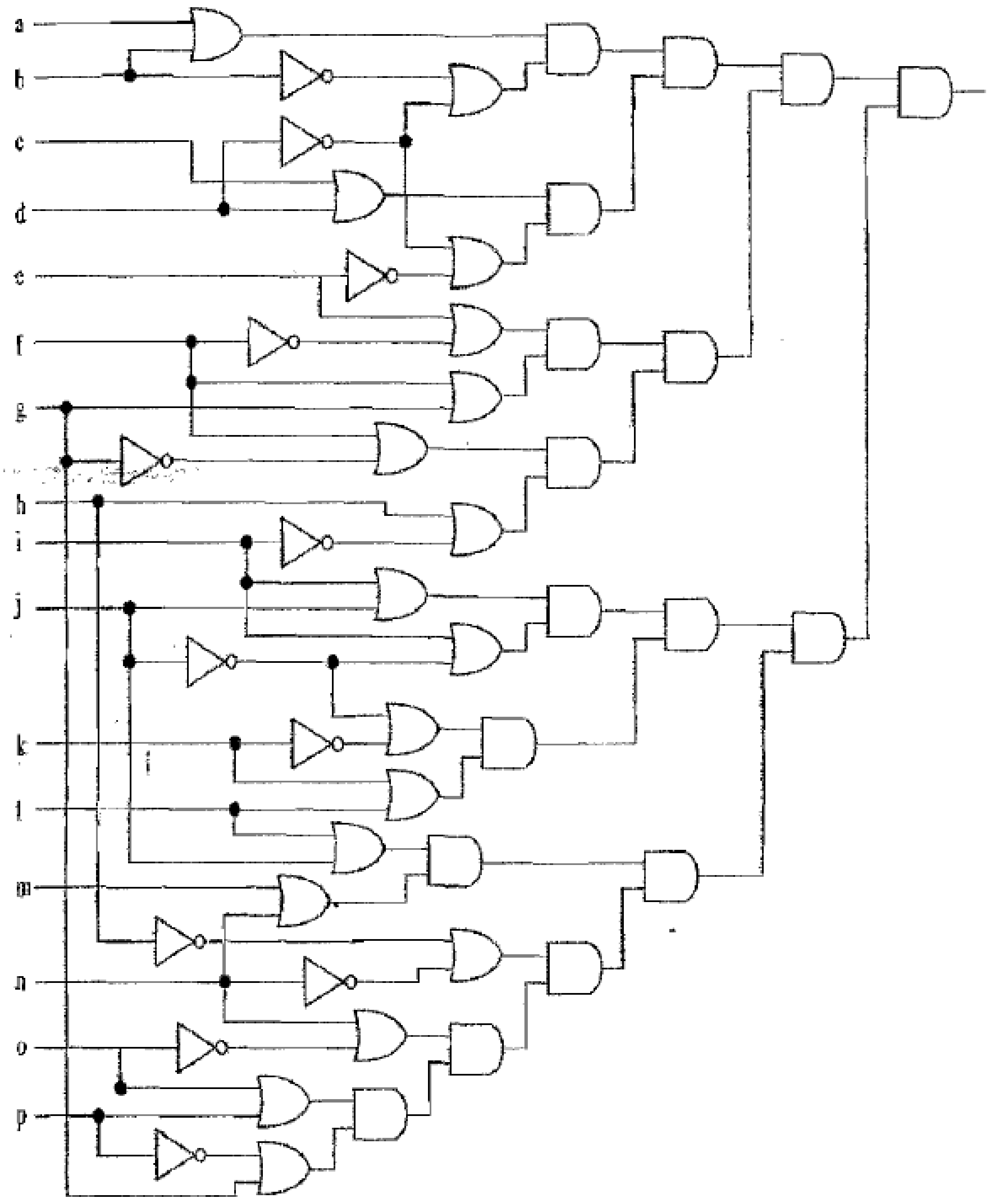
CSC 410/510 Programming Assignment #1 – Due 1 October

1. The circuit-satisfiability problem is important for the design and verification of logical devices. Unfortunately, it is in the class NP-complete, which means there is no known polynomial time algorithm to solve general instances of this problem. One way to solve the problem is to try every combination of inputs. Since the circuit in the picture has 16 inputs (labeled `a-p`) and every input can take on two values, 0 and 1, there are 2^{16} = 65,536 possible inputs.



The first step in developing a parallel algorithm is partitioning. In this case it is easy to spot the parallelism. To test each of the 65,536 combinations of inputs, and to see if any results in an output value of 1. We associate one task with one possible input, and if a task finds that its combination of inputs results causes the circuit to return 1, we print the combination, and when all tasks are finished we write how many solutions there are. Since all tasks are independent, the satisfiability check can be done in parallel. Someone in our group has already written the algorithm to check the circuit. The function takes process ID (*id*) and an integer value (*z*), it then convert this integer value to an array of 1s and 0s (*v[16]*) the array is then used in a comparison test to see what the output for the given sequence of 1s and 0s is. The code is available on the class page). You job is to write a shared memory program that uses this function and the:

```
#  pragma omp parallel for num_threads(thread_count)
```

to check each of the 65.536 possible inputs. The code needs to be timed using the `omp_get_wtime()` command, and you should use the `schedule(static,1)` and `schedule(dynamic,1)` to see if there is any difference in performance.

```
/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
/* Check if a given input produces TRUE (a one) */
int check_circuit (int id, int z)
{
  int v[16];        /* Each element is a bit of z */
  int i;

   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);
   if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])) {
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
         v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
         v[10],v[11],v[12],v[13],v[14],v[15]);
      fflush (stdout);
      return 1;
   } else return 0;
}
```

2. The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit.

   To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

   - Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., *n*).
   - Initially, let *p* equal 2, the first prime number.
   - Starting from *p*, enumerate its multiples by counting to *n* in increments of *p*, and mark them in the list (these will be 2*p*, 3*p*, 4*p*, etc.; the p itself should not be marked).
   - Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

   When the algorithm terminates, all the numbers in the list that are not marked are prime.

Create a parallel algorithm that finds all primes less than *n* (where *n* is provided by the user and is fairly large, about 1 to 10 million or so, but feel free to use smaller *n* to make sure your algorithm works correctly) on *p=8* processes with shared memory and prints a list of them to the screen. Time your code using both `static` and `dynamic` schedule to determine if there is any difference in performance and if your algorithm is deterministic. The output should provide a list of all primes less than *n* together with the runtime.

Example of output:

```
./prime 1000000

    0: 2 3 5 7 11 13 17 19 23 29
   10: 31 37 41 43 47 53 59 61 67 71
...
...
78480: 999721 999727 999749 999763 999769 999773 999809 999853 999863 999883
78490: 999907 999917 999931 999953 999959 999961 999979 999983
Elapsed time = 8.042866e+00 ms
```

Yes I know, I made 2 the 0th prime, but I am a computer scientist. There are several web-pages out there that provide lists of primes. Either as all the primes below a given number, or 1000, 10000, 100000 etc. first primes. So you should have no problem finding a way to check your list.

Extra Assignment for graduate students:

Draw a circuit with 16 inputs (*a*-*p*), `AND`, `OR` and `NOT` gates, similar to the one in assignment 1 (try to make it 4 to 5 levels deep). From this circuit create the appropriate if-statement so you can test the satisfiability of your circuit.

Assignment Submission:

Your homework is due at the beginning of class. Tar or zip all documentation and source files together. The header of each source file should contain all the normal information (name, class assignment etc.)

- Documentation - all of the following should be in: `prog1.pdf`
  - Description of the program.
  - Description of the algorithms and libraries used.
  - Description of functions and program structure.
  - How to compile and use the program.
  - Description of the testing and verification process.
  - Description of what you have submitted: `Makefile`, external functions, main, etc.
  - Format: PDF (write in any word processor and export as PDF if the option is available, or convert to PDF)
- Main programs
- Any required external functions *.c.
- Any include files you use (other than the standard ones).
- The `Makefile` to build the programs
- Tar the directory and then gzip using the correct filename.

[And I do know that there are better compression routines than gzip.]