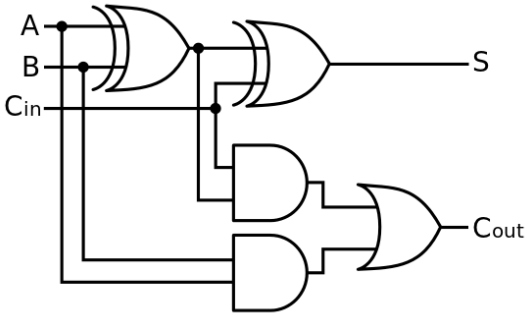
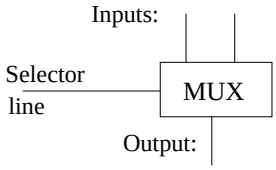
CSC 101 Exam #2 Fall 2012

1. Consider the following full adder circuit. For this problem, assume the inputs are: $A = 0$, $B = 1$, and $C_{\text{in}} = 1$. With these inputs, label the output of each logic gate in the diagram.

   Are the final outputs consistent with one-bit addition of two input bits and a carry-in bit ?

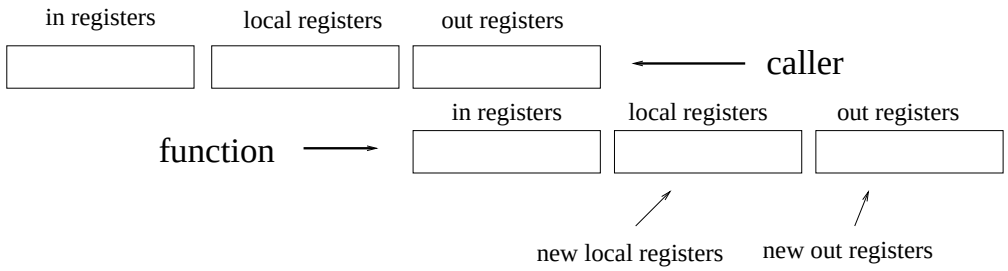


2. Recall we discussed the logic gate implementation of a 2-data-input, 1-data-output multiplexor with one selector input. Suppose we use the following symbol to represent that multiplexor:



   Now comes a question to challenge your ability to use what you have learned about computer circuits to solve a design problem. Consider the following diagram with two 4-bit general purpose registers and one 4-bit buffer register. Complete the diagram below to use 4 multiplexors (with a common selector line) to allow either of the two general purpose registers to be connected to the buffer register, depending on the state of the selector line.

3. SPARC Registers In our study of the SPARC architecture, we discussed the following diagram of SPARC CPU registers as they relate to a main program and a called function.



   (a) Explain how the main program provides parameters to the called function.

   (b) Explain how the function returns a result to the main program.

4. Assembly Language Design

   In our study of SPARC assembly language, we discussed branching instructions (a.k.a., "go to" instructions). Examples include the **ba**, **bl**, **ble**, **be**, **bne**, **bg**, and **bge** instructions. These type of instructions are present in all CPU designs, including MIPS4, Power PC, ARM, Intel, and AMD designs. Why is in necessary to include branching instructions in the design of a CPU ? What affect would it have on programming if they were left out of the CPU design ?

5. Assembly Language Programming

   Suppose the addresses of two data items (in memory) are provided to you in registers `%i0` and `%i1`. Write a short segment of SPARC assembly language instructions to exchange the two data items in memory. You may assume that all of the local registers are not currently in use.

6. What would be printed by the following C program ? Trace the program and show the values of t and i on each iteration of the loop.

```
#include <stdio.h>
int main()
{
    /* Declarations identify the data type of the variables, and     */
    /* enable memory to be allocated for the declared variables.     */

    int i, t ;    /* Declarations of "i" and "t".                    */

    t = 0 ;
    i = 1 ;
    while ( i <= 5 ) {
       t = t + i ;
       i = i + 1 ;
    }
    printf("%d\n", t ) ;
```

```
}
```

7. In our study of functions in C, we discussed passing parameters to a function. Scalar variables (i.e., variables that hold a single number) are "passed by value". Array variables (i.e., variables that hold many numbers) are "passed by reference". What do the terms "pass by value" and "pass by reference" mean in this context ?

8. A **perfect number** is a positive integer that is equal to the sum of all of its divisors (not including itself). For example, the first perfect number is 6; i.e., $1 + 2 + 3 = 6$. The next perfect number is 28; i.e., $1 + 2 + 4 + 7 + 14 = 28$. The next two perfect numbers are 496 and 8128 respectively[1]. Your task is two write a C program to input a number and decide if it is a perfect number. If the input number is a perfect number, your program should print "Yes". Otherwise, your program should print "No". To get you started, a partial solution is given below:

```
#include <stdio.h>
int main()
{
  /* Begin declarations:                                          */

  int num ;     /* A number that is read from the keyboard input. */

  int d      ;  /*  "d" is a trial divisor.                          */
                /*  "d" is a divisor of "num":  if (( num % d ) == 0)   */

  int total  ;  /* "total" is the sum of all divisors.            */

  /* Begin executable code:                          */

  total = 0 ;
  d = 1 ;
  scanf( "%d", &num ) ;  /* A function call that reads a number from  */
                         /* the keyboard and puts it into "num".      */

  /* Your part of the program goes here. */

}
```

SPARC Instructions

For illustration purposes we use registers `r0`, `r1`, and `r2`. In an actual program, you will use the names of the SPARC registers described earlier. For example, `%i0`, `%i1`, `%o0`, or `%l3`.

[1]The first four perfect numbers were known to mathematicians by approximately 100 A.D.

- `add  r0, r1, r2`
  - The `add` instruction requires three registers. On completion, register `r2` will contain the sum of `r0` plus `r1`.
- `sub  r0, r1, r2`
  - Similar to the `add` instruction, it requires three registers. On completion, register `r2` will contain the difference of `r0` minus `r1`. Optionally, a constant in the range -4096 to 4095 may be used in place of `r1`.
- `smul  r0, r1, r2`
  - Performs a multiply with signed integer registers. On completion, register `r2` will contain the product of `r0` times `r1`.
- `mov  r0, r1`
  - The `mov` instruction makes a copy of `r0` into `r1`. On completion, register `r0` and `r1` will contain the same value. Optionally, `r0` may be replaced by a constant in the range -4096 to 4095. This is useful for initializations.
- `nop`
  - The `nop` instruction is the "no operation" instruction. I.e., do nothing. It is used immediately after a branch instruction.
- `ld  [ r0 ],r1`
  - Register `r0` must contain the **address** of the desired data. The `ld` instruction copies data from the memory location (address) indicated by `r0` into register `r1`. This operation is known as a "load".
- `st  r0,[ r1 ]`
  - Register `r1` must contain the **address** of a valid memory location. The `st` instruction copies data from register `r0` to the memory location indicated by `r1`. This operation is known as a "store".
- `cmp  r0,  r1`
  - The `cmp` instruction compares two values and sets the condition code register to hold the result of the comparison.
- `bl  L1`
  - The `bl` instruction conditionally branches to the point in the program labeled by `L1`. The branch is taken only if a previous `cmp` instruction has recorded the condition "less" in the condition code register.
- Other branching instructions include `ble`, `bg`, `bge`, `be`, `bne`, and `ba`. These instructions correspond to "less or equal", "greater", "greater or equal", "equal", "not equal", and "always" respectively.