CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: February 27, 2002

Homework 3

Due: Wednesday, February 20, 2002.

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. None of the problems requires more than a few lines of SML code.

Problem 1: Speeding tickets (50 points)

A friend called you yesterday with some bad news. He got a speeding ticket on Whitney Ave. Even debating skills did not help him to convince the police to let him go. 35mph in a 25mph zone is just not acceptable, they said. This brought him to an idea, for which he needs your help. He wants to design a piece of hardware that can be set to a tempo limit and will raise an alarm signal once the current speed exceeds it. Modern cars are all computer controlled, so it should not be very difficult to get access to a digital representation of the current cruising speed.

Your friend would have used your piece of hardware in the following way. He would have set the current speed limit to 25mph and the alarm should have gone off way before the police stopped him. All you have to do now is to define a circuit that takes two 8 bit numbers as input, and returns one alarm bit as output. It is set to 1 once the first number exceeds the second.

1. Write out a truth table for a circuit that compares only two one bit numbers.

   Hint: Your circuit should have two in-connectors x and y, and two out-connectors e, l, where e is true if $x = y$, and l is true if $x < y$.

   ```
   Truthtable

    a b l e
    0 0 0 1
    0 1 1 0
    1 0 0 0
    1 1 0 1
   ```

2. Write out a Boolean expression for the truth table in 1.

   Hint: Your truth table should have four columns relating x, y, l, and e.

```
l = not a /\ b
e = not (a xor b)
```

3. Design a circuit that implements this in hardware.

   Hint: This means on a piece of paper, or in ASCII.

```
a --o|---|
     |AND|--- l
b ---|---|

a ---|---|
     |XOR|o-- e
b ---|---|
```

4. Generalize the circuit to n bit numbers.

   Hint: Again, draw the circuit on a piece of paper. Write out the circuit for the nth stage, the $n-1$th stage, and the last stage. We will be able to reconstruct the missing details.

```
Please refer to the SML program for compnbit. The circuit for n-bit
comparator can be implemented recursively as shown in the function.
It is also possible to build a non-recursive $n$ bit comparator.
```

5. Implement your circuit in the simulator we used in class.

   Hint: Use the file `/c/cs201/lib/code/ass3/problem1.sml` which contains all necessary auxiliary functions discussed in class. Write the following functions.

   (a) `eq : Circuit -> Circuit -> Circuit` (implements e, see lecture 11)

   (b) `lt : Circuit -> Circuit -> Circuit` (implements l)

   (c) `comp1bit : Circuit -> Circuit -> Circuit` (one bit comparator)

   (d) `compnbit : [Circuit] -> [Circuit] -> Circuit` (n bit comparator)

   To test your code, declare the three different values for the different speeds:

```
val speed23 =
  [SIGNAL O,SIGNAL O,SIGNAL O,SIGNAL I,SIGNAL O,SIGNAL I,SIGNAL I,SIGNAL I]
val speed25 =
  [SIGNAL O,SIGNAL O,SIGNAL O,SIGNAL I,SIGNAL I,SIGNAL O,SIGNAL O,SIGNAL I]
val speed35 =
  [SIGNAL O,SIGNAL O,SIGNAL I,SIGNAL O,SIGNAL O,SIGNAL O,SIGNAL I,SIGNAL I]
```

   To test your code, compare your results with the following experiments:

```
simulateList (compnbit speed25 speed23);
val it = [SIGNAL O,SIGNAL O] : Circuit list
```

indicates that the speed limited has not been exceeded. The right is the alarm flag.

```
simulateList (compnbit speed25 speed35);
val it = [SIGNAL O,SIGNAL I] : Circuit list
```

indicates that the alarm has ringing. An finally

```
simulateList (compnbit speed25 speed25);
val it = [SIGNAL I,SIGNAL O] : Circuit list
```

shows that the alarm has not gone off yet, but that the system recognized that the vehicle is going *exactly* speed limit.

```
(* eq x y = C

   Invariant:
   C is a circuit that has to open ports,
   and decides if x = y
*)
fun eq x y =
  ORGate (ANDGate (NOTGate x,
                   NOTGate y),
          ANDGate (x, y))

(* lt x y = C

   Invariant:
   C is a circuit that has two open ports,
   and decides if x < y
*)
fun lt x y =
  ANDGate (NOTGate x,
           y)


(* comp1 x y = C

   Invariant:
   C is a circuit that has two open ports,
   and returns two things simulatously [eq, lt]
   eq is I iff x = y, and lt is I iff x < y
*)
fun comp1bit x y =
  [eq x y,
```

```
   lt x y]

(* compnbit xs ys = Cs

   Invariant:
   Cs = c :: zs where c, zs are circuits that
   compute the two bits of infromation: if xs = ys (first bit)
   or if xs < ys (second bit)
*)
fun compnbit [x] [y] = comp1bit x y
  | compnbit (x :: xs) (y :: ys) =
    let
      val [eq1, lt1] = comp1bit x y
      val [eq2, lt2] = compnbit xs ys
    in
      [ANDGate (eq1, eq2), ORGate (lt1, ANDGate (eq1, lt2))]
    end
```

Problem 2: Machine organization (50 points)

When simulating the run of a machine, it is of utmost importance how memory is represented. In the last assignment, the choice fell on a BOOL list. This was good because it illustrated clearly how memory is organized. In this assignment, however, we want to study how memory interacts with other parts of the machine and how machine language really works. We therefore take the freedom to abstract from some details, especially, from bits and bytes and concentrate on the essence on how to address, create, update, and maintain memory.

First, we introduce the address space. We abstract from concrete addressing techniques in a real machine and define an *address* simply to be an integer.

```
type address = int
```

If we represent memory as a list of Boolean values, update is going to be very expensive. One can do better by using data structures such as balanced trees or hashtables, for quick lookup and update. For our purposes, however, it is completely sufficient to use a less efficient albeit more elegant technique to implement memory as a function from addresses to their content.

```
datatype Content
  = Free
  | Int of int
  | Real of real
type memory = address -> Content
```

Each memory cell can be either free, contain an integer or a real number. This allows us abstract away from all details covered in the previous assignment. The representation invariant behind `memory` is as follows. If `M` is of type `memory`, then for any valid address `a`, `M a` returns the content of `M` at `a`.

1. Define the empty memory.

   ```
   empty : memory
   ```

   Hint : `empty 2004` should evaluate to `Free`.

   ```
   fun empty a = Free
   ```

2. Write a function that updates memory.

   ```
   update : memory -> address -> Content -> memory
   ```

   `update M a C` updates `M` at address `a` to content `C`.

   Hint: The `update` function is easier to implement then you might assume. To see if it does the right thing, consider for example

   ```
   val mem = update empty 2000 (Real 1.414)
   ```

   which sets the content at address `2000` to `Real 1.414`.

   ```
   fun update m a v = fn x => if x = a then v else m x
   ```

3. Write a function the looks up information in memory.

   ```
   lookup : memory -> address -> Content
   ```

   `lookup M a` looks up address `a` in `M`.

   Hint: Lookup is even easier to implement than update. As example consider the following command.

   ```
   - lookup mem 2000;
   val it = Real 1.414 : Content
   ```

   ```
   fun lookup m a = m a
   ```

4. Write a function that lets the user inspect memory.

   ```
   inspect : memory -> (int * int) -> unit
   ```

   `memory M (a, b)` prints the content of memory `M` between addresses `a` and `b`, inclusive. For example, after storing 1.414 at address 2000, and 42 at the address 2001, `inspect` should provide the following output on the screen.

   ```
   - inspect mem (2000, 2010);
   2000  1.414
   2001  42
   2002  *
   2003  *
   2004  *
   ```

```
2005  *
2006  *
2007  *
2008  *
2009  *
2010  *
val it = () : unit
```

Hint: Use the print command.

```
fun contentToString (Int i)    = Int.toString i
  | contentToString (Real x)   = Real.toString x
  | contentToString (Free)     = "*"

fun inspect m (n1, n2) =
    if n1 > n2 then ()
    else (print (Int.toString n1);
          print "  ";
          print (contentToString (m n1));
          print "\n";
          inspect m (n1+1, n2))
```

Hand-in Instructions

In the course directory /c/cs201/bin, there are five programs that support you in submitting your solution to the homework.

```
submit    assignment-number file(s)
unsubmit  assignment-number file(s)
check     assignment-number
protect   assignment-number file(s)
unprotect assignment-number file(s)
```

`submit` allows you to submit one of several files. For example

```
/c/cs201/bin/submit 3 hw3.sml hw3-examples.sml
```

submits your file `hw3.sml` and `hw3-examples.sml`. `check` can give you the peace of mind that your solution has really been submitted, and if you would like to make last minute changes to your solution, use `unsubmit` before submitting the updated version. `protect` and `unprotect` give you the power to protect/unprotect submitted solutions from deletion.