# Digital Hardware: Practical Instructions

Mike Spivey, Hilary Term 2004

In this practical, you will use HUGS to build a simulation of some arithmetic circuits and measure their timing performance in terms of combinational depth. The practical involves building a Haskell script almost from scratch, but to save typing, the definitions given in these instructions are contained in a file Adder.hs online. That file is divided into sections that are commented out, so that you can load the whole file before completing all parts of the practical. As you work through the practical, you'll need to 'uncomment' each section and fill in the blanks. I've marked with a '$\lhd$' in the margin of these instructions each place where you should stop and test your work so far.

## 1 Representing integers

We will represent a binary digits by a pair $(b, t)$, where $b$ is the value of the bit (either 0 or 1) and $t$ is the time at which the bit becomes stable:

$$\textbf{type}\ \mathit{Bit} = (\mathit{Int}, \mathit{Int})$$

Begin by defining two functions

$$\begin{aligned}&\mathit{rep} :: \mathit{Int} \to \mathit{Integer} \to [\mathit{Bit}]\\&\mathit{bin} :: [\mathit{Bit}] \to \mathit{Integer}\end{aligned}$$

such that *rep n x* is the binary representation of a positive integer $x$ as a list of bits (least significant bit first) each with time 0, and if *bs* is such a representation, then *bin bs* is the integer it represents. We are using both the Haskell type *Int* of machine integers, and the type *Integer* of arbitrary-precision integers here; we use *Integer* to represent values that may become large, so that we can simulate 64 or 128 or 256 bit adders and multipliers, even though our workstations represent *Int* in only 32 bits or less. You will need to use the conversion functions[1]

[1] In truth, these functions have types

$$\begin{aligned}&\mathit{toInteger} :: \mathit{Integral}\ \alpha \Rightarrow \alpha \to \mathit{Integer}\\&\mathit{fromInteger} :: \mathit{Num}\ a \Rightarrow \mathit{Integer} \to \alpha\end{aligned}$$

but we can ignore that complication for our purposes.

$$toInteger :: Int \to Integer$$
$$fromInteger :: Integer \to Int$$

to convert between these two types. Check that $bin\,(rep\;n\;x) \equiv x$ for various values of $n$ and $x$. ◁

## 2 Some useful gates

Define functions

$$majority :: (Bit, Bit, Bit) \to Bit$$
$$parity :: (Bit, Bit, Bit) \to Bit$$

that provide the carry and sum bits output by a full adder. Make each of these gates take 1 time unit, so that e.g. $majority\,((0, 3), (1, 4), (1, 5)) = (1, 6)$.

Combine the two functions to define

$$full\_adder :: Bit \to (Bit, Bit) \to (Bit, Bit)$$

so that $full\_adder\;cin\,(a, b) = (s, cout)$, where $s$ is the sum bit and $cout$ is the carry out. (The type of *full_adder* is chosen so that we can use it conveniently later).

## 3 Ripple-carry adder

Define a function

$$ripple :: (a \to b \to (c, a)) \to a \to [b] \to [c]$$

that models the data flow in a ripple-carry adder, so that such an adder can be defined by

$$adder\;as\;bs = ripple\;full\_adder\;(0, 0)\;(zip\;as\;bs)$$

(Here, $(0, 0)$ is the initial carry input, which like the inputs, is stable at time 0.) Arrange that if *as* and *bs* both have length $n$, then so does *adder as bs*, so that the final carry bit is thrown away.

Try evaluating $adder\,(rep\;n\;x)\,(rep\;n\;y)$ for various values of $n$, $x$, and $y$. ◁ Use *bin* on the result to verify that the answers are correct, but also look at the times in the raw result. Work out how the times grow as you increase $n$, and observe that the times depend only on the value of $n$ and not on $x$ and $y$.

## 4 Introducing carry status

Now define a type *Flag* like this:

```
data KPG = K | P | G
type Flag = (KPG, Int)
```

so that a flag is one of the values $K$, $P$ or $G$, together with the time that it becomes stable. Although we could represent these values using two bits, we can leave that detail out of our simulation and still get interesting results.

Define functions

```
kpg :: (Bit, Bit) → Flag
sumbit :: Flag → (Bit, Bit) → Bit
bun :: Flag → Flag → Flag
```

that model the three components used in a carry-lookahead adder: *kpg a b* generates the initial carry status for input bits *a* and *b*, and *sumbit w* (*a*, *b*) gives the sum bit output for bits *a* and *b* given carry status *w*. Finally, *bun* is the associative carry-propagation operator ⊗. Make all three of these functions take 1 time unit; although the componenents we have defined will probably not have equal time delays in practice, treating them as if they were the same will still let us study the order of growth of the overall depth of the circuit.

To test these functions, define a new ripple-carry adder *rip_cl_adder* by

```
rip_cl_adder as bs = ripple kpg_adder (K, 0) (zip as bs)
```

for a suitable function

```
kpg_adder :: Flag → (Bit, Bit) → (Bit, Flag)
```

Verify that *rip_cl_adder* gives the same results as *adder*, and that the circuit depth has the same order of growth. ◁

## 5 Parallel prefix

Now we are going to define a function that models the parallel prefix computation for carry status. Of course, this function will probably be slower when we run it as a functional program with Hugs, because of all the work that will be done dynamically to route the signals. But that doesn't matter, because what interests us is the time delays that are accumulated in our simulation. If *as* and *bs* have a fixed length $2^n$, then the pattern of computation in *cl_adder as bs* will be fixed, and we could expand the function statically to get a circuit for the adder.

A *prefix function pfx* has type

```
type Prefix_fun a = (a → a → a) → a → [a] → [a]
```

and is such that (provided ⊕ is an associative operation)

$$\begin{aligned} &pfx\,(\oplus)\,u\,[x_0, x_1, \ldots, x_{n-1}] \\ &\quad = [u, u \oplus x_0, u \oplus x_0 \oplus x_1, \ldots, u \oplus x_0 \oplus x_1 \oplus \cdots \oplus x_{n-2}]. \end{aligned}$$

Note that the input value $x_{n-1}$ is not used: that's because we are throwing away the final carry-out bit to which it would otherwise contribute.

If *pfx* is a prefix function, then *cl_adder pfx* is an adder:

```
cl_adder :: Prefix_fun Flag → [Bit] → [Bit] → [Bit]
cl_adder pfx as bs = zipWith sumbit carries inputs
  where
    carries = pfx bun (K, 0) (map kpg inputs)
    inputs = zip as bs
```

Define a prefix function *rip_prefix* that works naively from left to right, so that *rip* = *cl_adder rip_prefix* is yet another ripple-carry adder. Verify that ◁

the behaviour of this adder agrees with that of the previous adders you have built.

Now for the *tour de force*: define a new prefix function *par_prefix* that works on inputs of size $2^k$, using the parallel prefix algorithm. The code given in the lecture was as follows:[2]

```
par_prefix :: (a → a → a) → a → [a] → [a]
par_prefix (%) u [x] = [u]
par_prefix (%) u xs =
  concat (zipWith g es (par_prefix (%) u (zipWith (%) es os)))
  where
    es = evens xs; os = odds xs
    g x y = [y, y % x]
```

You will just need to supply the definitions of *evens* and *odds*. The function *cla* = *cl_adder par_prefix* should be an adder. Verify that it functions correctly, and examine the rate at which the overall circuit depth grows with $k$ in *cl_adder par_prefix* (*rep n x*) (*rep n y*), where $n = 2^k$. ◁

## 6 Very optional extra

Build simulations of one or more of the following circuits and examine the order of growth of their combinational depth:

- A carry-skip adder.
- An array multiplier.
- A Wallace-tree multiplier.

These circuits are all explained in the relevant chapter of (the first edition) of Cormen, Leiserson and Rivest. They've deleted it from the second edition, alas.

## 7 Report

Add comments to your Haskell script showing some tests you have performed, together with their results. Briefly comment on the order of growth of the time delays of the various adders. Print out your script and submit it as your practical report.

[2] In this definition, % stands for an arbitrary (associative) binary operation – it's a formal parameter of *par_prefix*. It looks a bit odd at first to use an operator symbol as a formal parameter, but it is completely legal Haskell, and quite a natural thing to do once you are used to it.