Custom Computing: Assessed Coursework

Ioannis Kassinopoulos

March 4, 2013

Question 1

Recurring engineering costs are the costs that will occur in a repeating fashion during the production, usually involving fabriction. These costs are usually descriped in a per unit form.

Non-recurring engineering cost is the one-time up-front cost for research, design, testing and development of a new product.

As we can see below, the minimum number of units that need to be sold for the ASIC implementation to be cost-effective is 1 million units.

$C_{FPGA} > C_{ASIC} \Rightarrow £0 + £2 \times N_{units} > £10^6 + £1 \times N_{units} \Rightarrow N_{units} > 10^6$

Question 2

(a) Diagramatic and symbolic Simulation

Diagram of circuit Q1

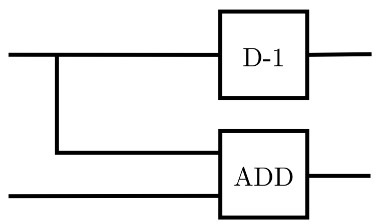

Figure 1: the circuit as derrived from Q1

Diagram of circuit P1

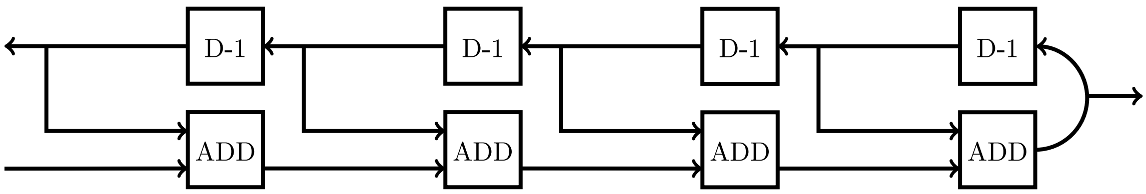

Figure 2: the circuit as derrived from P1

Simulation

The source code of the simulation (uninitialized delay) is the following:

```
INCLUDE "prelude.rby".
P1 n = Q1^n; fork^~1 .
Q1 = snd fork; rsh; [add,D^~1].
current = P1 4.
```

The result after executing `re "a;b;c"`

```
0 - <a,?> ~ ((((a + ?) + ?) + ?) + ?)
1 - <b,?> ~ ((((b + ?) + ?) + ?) + ((((a + ?) + ?) + ?) + ?))
2 - <c,?> ~ ((((c + ?) + ?) + ((((a + ?) + ?) + ?) + ?)) + ((((b + ?) + ?) + ?)
          + ((((a + ?) + ?) + ?) + ?)))
```

The source code of the simulation (initialized delay with 0) is the following:

```
INCLUDE "prelude.rby".
P1 n = Q1^n; fork^~1 .
Q1 = snd fork; rsh; [add,DI 0^~1].
current = P1 4.
```

The result after executing `re "a;b;c"`

```
0 - <a,0> ~ ((((a + 0) + 0) + 0) + 0)
1 - <b,0> ~ ((((b + 0) + 0) + 0) + ((((a + 0) + 0) + 0) + 0))
2 - <c,0> ~ ((((c + 0) + 0) + ((((a + 0) + 0) + 0) + 0)) + ((
            ((b + 0) + 0) + 0) + ((((a + 0) + 0) + 0) + 0)))
```

The result after executing `re -s 4 1"`

```
0 - <1,0> ~ 1
1 - <1,0> ~ 2
2 - <1,0> ~ 4
3 - <1,0> ~ 8
```

The simulation output (for 4 cycles) can be found in the included zip file.

(b) Use of the transformation

We first add D registers between the Q1 series and $fork^{-1}$ which give us: $P2 = Q1^n; [D, D]^n; fork^{-1}$

Using the transformation: $P2 = (Q1; [D, D])^n; fork^{-1}$ since $Q1; [D, D]$ is equivalent to $[D, D]; Q1$.

Now we can consider Q2 to be equal to $Q1; [D, D]$ and therefore $P2 = Q2^n; fork^{-1}$ is our definition as stated by the question.

Going deeper into the definition, we now need to check if this means that D registers have been added between the adders.

$Q2 = Q1; [D, D]$

$\Rightarrow Q2 = snd\, fork; rsh; [add, D^{-1}]; [D, D]$

$\Rightarrow Q2 = snd\, fork; rsh; [(add; D), (D^{-1}; D)]$

$\Rightarrow Q2 = snd\, fork; rsh; [(add; D), id] = snd\, fork; rsh; fst(add; D)$

The new circuit can be seen below:

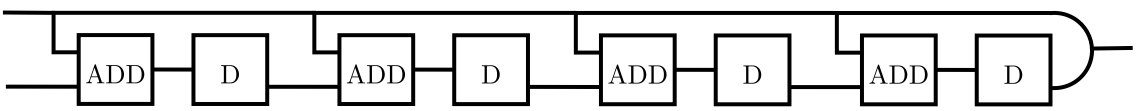

Figure 3: The circuit after using the transformation to add the registers between the adders

The source code for the circuit is the following:

```
INCLUDE "prelude.rby".
Q2 = snd fork;rsh;fst (add; D).
P2 n = Q2^n; fork^~1.
current = P2 4.
```

The result can be seen below (for uninitialized delay)

```
re "a;b;c"
Simulation start :

    0 - <a,?> ~ ?
    1 - <b,(? + ?)> ~ (? + ?)
    2 - <c,((? + ?) + (? + ?))> ~ ((? + ?) + (? + ?))

Simulation end :
```

(c) Slowdown

Slowdown means doubling the registers of the design which trivially translates into combining the designs of Q1 and Q2 into a new design Q3 which has the registers of both. This can be done by extending Q2 from $sndfork; rsh; fst(add; D)$ or $sndfork; rsh; [(add; D), id]$ to $sndfork; rsh; [(add; D), D^{-}1]$.

The new circuit is the following:

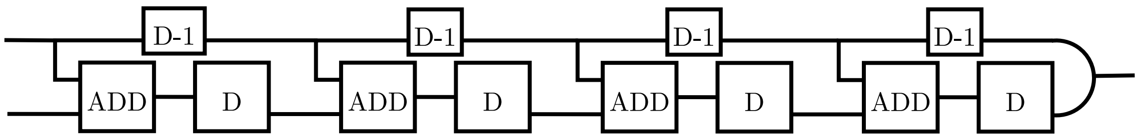

Figure 4: The circuit after slowdown

The source code:

```
INCLUDE "prelude.rby".
Q3 = snd fork;rsh;[(add; D),D^~1].
P3 n = Q3^n; fork^~1.
current = P3 4.
```

The result:

```
re "a;b;c"
Simulation start :

    0 - <a,?> ~ ?
    1 - <b,?> ~ (? + ?)
    2 - <c,?> ~ ((? + ?) + ?)

Simulation end :
```

Question 3

(a) Proof by induction

In order to show that $[P, Q]^n; R = R; Q^n$ for $n > 0$, we first have to show that it is $True$ for n = 1.

Base case: $[P, Q]^1; R = R; Q^1$

This is intuitively shown to be true by the given assumption $[P, Q]^n; R = R; Q$ which is equivalent.

Assuming that it is also true for $n = k > 0$

Inductive Hypothesis: $[P, Q]^k; R = R; Q^k$

We need to show that the same is true for $n = k + 1$ and $[P, Q]^{k+1}; R = R; Q^{k+1}$

$[P, Q]^{k+1}; R$ **LHS**

$= [P, Q]^k; [P, Q]; R$ (by sequential expansion of $[P, Q]^{k+1}$)

$= [P, Q]^k; R; Q$ (since $[P, Q]; R = R; Q$ given)

$= R; Q^k; Q$ (by the i.h. $[P, Q]^k; R = R; Q^k$)

$= R; Q^{k+1}$ (by sequential contraction of $Q^k; Q$) **RHS**

So by induction we have **proved** that if $[P, Q]; R = R; Q$ is given to be $True$, for $n > 0$:

$[P, Q]^n; R = R; Q^n$ is also $True$

(b) Inductive Definitions

Right-reduction

$rdr_1 = fst\,[-]^{-1}; R.$

$rdr_{n+1} = fst\,apl_n^{-1}; lsh; snd(rdr_n\ R); R.$

Delta (triangle)

$\Delta_0 = [\,].$

$\Delta_{n+1} = [\Delta_n, R^n]\backslash apr_n.$

(c) Horner's Rule

Left-hand side

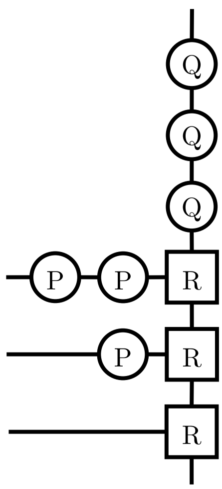

Figure 5: LHS of the rule for n = 3

Right-hand side

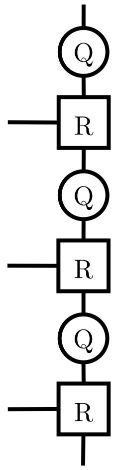

Figure 6: RHS of the rule for n = 3

(d) Polynomial Evaluation

R stands for the add operation (addition), P and Q both stand for multiplication by a constant (let this constant be x). For the given coefficients a_0, a_1, a_2, a_3, the circuit will be the following.

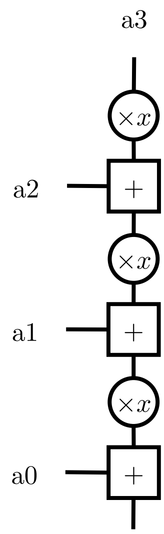

Figure 7: The optimised circuit as adjusted for polynomial evaluation

and the simulation written in Ruby: (x should be replace by the required number)

```
INCLUDE "prelude.rby".
multc n = pi1^~1;snd n;mult.
Q = multc 'x'.
R = add.
POL n = rdr n (snd Q; R).
current = POL 3.
```

run with `re "a_0 a_1 a_2 a_3"` produces the following output:

```
Simulation start :
```

```
    0 - <<a_0,a_1,a_2>,a_3> ~ (a_0 + ((a_1 + ((a_2 + (a_3 * x)) * x)) * x))

Simulation end :
```

Question 4

(a) Non-recursive definition of $btree_3$ and its type

The non-recursive definition of $btree_3$ R is:

```
 btree 3 R = [[R,R];R,[R,R];R];R.
```

and therefore its type is given by:

$<<< X, X >, < X, X >>, << X, X >, < X, X >>> \sim\ X.$

(b) Fully pipelined timeless implementation of btree for timeless R

Definition

The system can be described by the following inductive equation:

$pbtree_1 = R.$

$pbtree_{n+1} = [pbtree_n R, pbtree_n R]; [D^n, D^n]; R; AD^n.$

Proof by induction

Here, we will try and prove that our equation is equivalent to the given.

Base Case: $pbtree_1 = btree_1$ (required to show)

$pbtree_1$ **LHS**

$= R$ (by definition of *pbtree*)

$= btree_1$ (by definition of *btree*) **RHS**

Inductive Hypothesis: $pbtree_k = btree_k$

We need to show that: $pbtree_{k+1} = btree_{k+1}$.

$pbtree_{k+1}$ **LHS**

$= [pbtree_k R, pbtree_k R]; [D^k, D^k]; R; AD^k$. (by the pbtree definition)

$= [btree_k R, btree_k R]; [D^k, D^k]; R; AD^k$. (replacing $pbtree_k$ with $btree_k$ by the hypothesis)

$= [btree_k R, btree_k R]; R$. (replacing $[D^k, D^k]; R; AD^k$ with R since R is timeless and they cancel out.)

$= btree_{k+1}$. (definition of btree) **RHS**

Symbolic simulation of a binary adder

```
INCLUDE "prelude.rby".
btree n R =
        IF (n $eq 1) THEN
                (R)
        ELSE
                ([btree (n-1) R, btree (n-1) R];[D^(n-1),D^(n-1)]; add ; (AD^(n-1))).
current = btree 3 add.
```

The results:

```
re -s 3a b c d p q r s
Simulation start :
    0 - <<<a_0,b_0>,<c_0,d_0>>,<<p_0,q_0>,<r_0,s_0>>>
          ~ (((a_0 + b_0) + (c_0 + d_0)) + ((p_0 + q_0) + (r_0 + s_0)))
    1 - <<<a_1,b_1>,<c_1,d_1>>,<<p_1,q_1>,<r_1,s_1>>> ~ ?
    2 - <<<a_2,b_2>,<c_2,d_2>>,<<p_2,q_2>,<r_2,s_2>>> ~ ?
Simulation end :
```

(c) Change of type

In order to understand the changes that need to occur in our definition let's consider $btree_3$.

The transformation we wish to achieve is the following:

$< X, X, X, X, X, X, X, X > \Rightarrow <<< X, X >, < X, X >>, << X, X >, < X, X >>>.$

which is equivallent to applying to the initial flat list the prelude function $half_4$ or $half_{2^{(3-1)}}$, and then to each half $half_2$.

The obvious pattern is easily implementable in Ruby due to the functional nature of the language. The inductive definition of the equation is the following:

$btree_1 R = R$

$btree_n R = half_{2^{n-1}}; [btree_{n-1} R, btree_{n-1} R]; R.$

The source code of the implementation:

```
INCLUDE "prelude.rby".
btree n R =
        IF (n $eq 1) THEN
                (R)
        ELSE
                (half (2 $exp (n-1));[btree (n-1) R, btree (n-1) R];R).
current = btree 3 add.
```

The result (sum 1 - 8):

```
re 1 2 3 4 5 6 7 8
Simulation start :

    0 - <1,2,3,4,5,6,7,8> ~ 36

Simulation end :
```

The result (symbolic simulation):

```
re a b c d p q r s
Simulation start :

    0 - <a,b,c,d,p,q,r,s> ~ (((a + b) + (c + d)) + ((p + q) + (r + s)))

Simulation end :
```