Before you begin

Author Details

Name : Arya Vishe
Roll Number : 21D070018

How to run?

Simply run

```
chmod +x runscript.sh
./runscript q1 500
```

to simulate the first question's testbench for 500us. (syntax : `./runscript q<question_number> <time-in-us>`)

Other notes

1. This markdown file was used to generate the `README.pdf` so both have identical content.
2. I used Obsidian to make this markdown file so some of the syntax will not render properly on non-Obsidian markdown viewers. In that case, just read the pdf.
3. Each folder contains screenshots of gtkwave for testbench simulation (`testbench_output.ps`)

Question 1

First Part

Powers of 2 are represented as a single `SET` bit out of 8 (in this case) bits overall. Thus verifying for the case where both operands are powers of 2 denotes verifying it for the case when all but 1 bit per number are `RESET` .

Note: Multiplication of any number by a power of 2 (say 2^k) is equivalent to left-shifting the number by equivalent number of bits (say k).

We can write `a` and `b` as

$$\texttt{a} = \underbrace{0\ldots0}_{7-X}1\underbrace{0\ldots0}_{X}$$

$$\texttt{b} = \underbrace{0\ldots0}_{7-Y}1\underbrace{0\ldots0}_{Y}$$

Where $X, Y \in \{0, 1, \ldots, 7\}$. We will only look at the different conditions for `loop` . Note that `ta[0]` simply scans over `a` from right to left.

1. For the first X iterations, `ta[0] = 0` and `t=0^17` so nothing happens. By the end of X iterations, `t=0^17` .
2. When `ta[0]` , we get `t=0&b&0^8` OR `t=b<<8`
3. For the next $7 - X$ iterations, `ta[0]=0` thus we simply right-shift `t` each time.
4. Note that the counter actually runs for 9 iterations (from `0` to `8`) thus we right-shift once for that.

Final answer `p` is finally

$$\texttt{p} = ((\texttt{b<<8})\texttt{>>}7 - X)\texttt{>>}1$$

$$\Rightarrow \texttt{p} = \texttt{b} << X$$

Thus the algorithm is equivalent!
As an example,

$a = 00010000 = 16_{10}$
$b = 01000000 = 64_{10}$ } $a \times b = 1024_{10}$

↓

$ta = 0001\ 0000$
$t = 0^{17}$ } rst + start

← counter

(0) $ta[0] \neq 1 \rightarrow t = 0^{17}$

⋮

(4) $ta[0] = 1 \rightarrow t = 0 \& (0100\ 0000) \& 0^8$ — loop

(5) $ta[0] \neq 1 \rightarrow t = 0^2 \& (0100\ 0000) \& 0^7$

⋮

(8) $ta[0] \neq 1 \rightarrow t = 0^5 \& (0100\ 0000) \& 0^4$

↑ counter = 8

↓

$p = t = 0^5 \& b \& 0^4 = b << 4 = 1024_{10}$

It's correct!

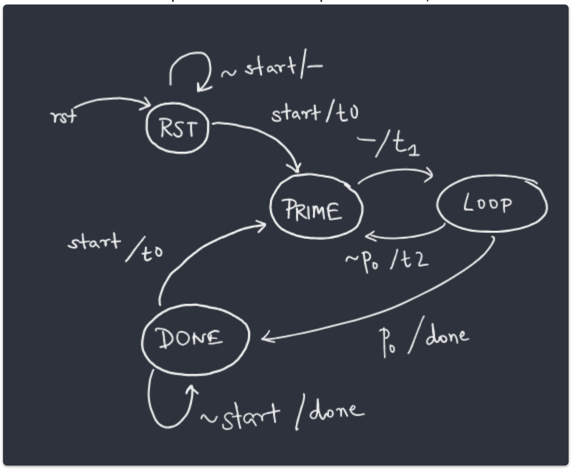
Second Part

Solution that uses provided adder

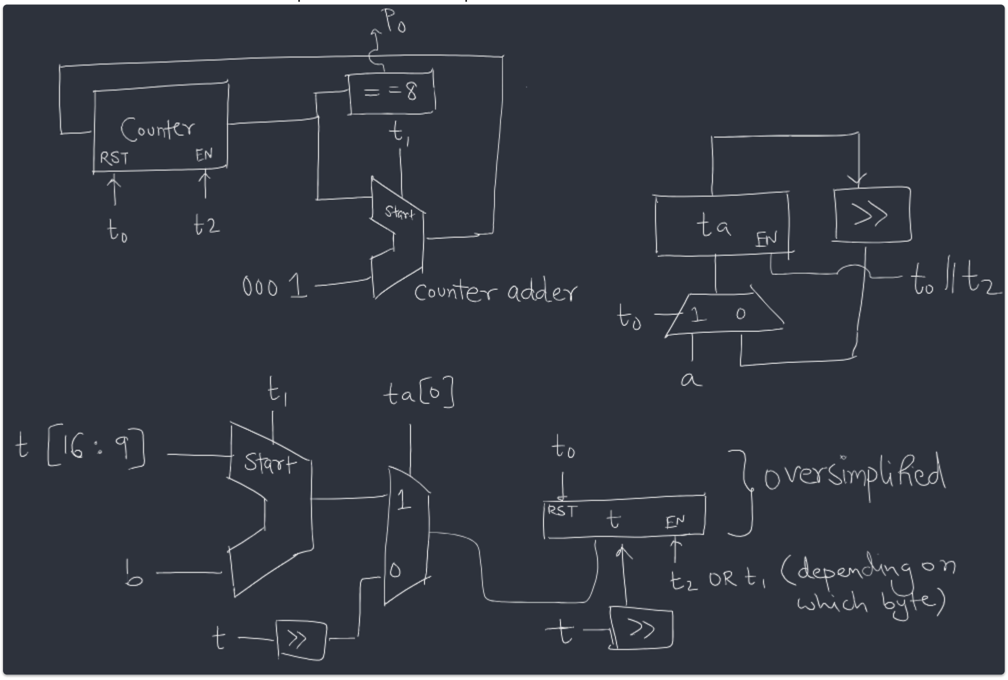
Since the adder brings in a cycle delay in returning the correct value of the sum, we will need to tweak the RTL specification a bit to create a new state : `PRIME_STATE`

```
// Shift and subtract divider.
// Uses a single 9-bit subtractor.
//
// input start, a[7:0], b[7:0]
// output done, q[7:0]
//
// register ta[15:0]
// register t[7:0]
// registers counter[3:0]
//
// Default outs:
// done=0, q = 0
rst_state:
    if start then
        ta := a
        counter := 0
        t := 0
        goto wait_state
    else
        goto rst_state
    endif

prime_state:
    if(ta[0]) then
        // Prime 9-bit adder
        main_adder_start := 1
        main_adder_a := t[16:9]
        main_adder_b := b
        // We are allowed to do the shifting for the other stuff anyways
        t[7:0] := t[8:1]
    else
        t[16:0] := 0 & t[16:1]
    endif
    // We also need to have the option to update the value of counter in the next state
    counter_adder_start := 1
    counter_adder_a := counter
    counter_adder_b := 1
    goto loop_state // Even if ta[0] = 0, we still need to update counter so we go to the loop_state

loop_state:
    // Now we have the updated values of both
    if (ta[0]) then
        // Update the t-value
        t[16:8] := main_adder_c
    endif // Nothing else needs to be done
    // We can now finally right-shift ta
    ta[7:0] := 0 & ta[7:1]
    if (counter == 8) then
        goto done_state
    else
        counter := counter_adder_c
        goto prime_state
    endif

done_state:
    done = 1
    // t is visible at p
    if start then
        t := 0
        counter := 0
        ta := 0^8 & a
        goto loop
    else
        goto done_state
    endif
```

This can be used to represent the control path's FSM as,



Let's now connect these transfers and predicates to the datapath as well



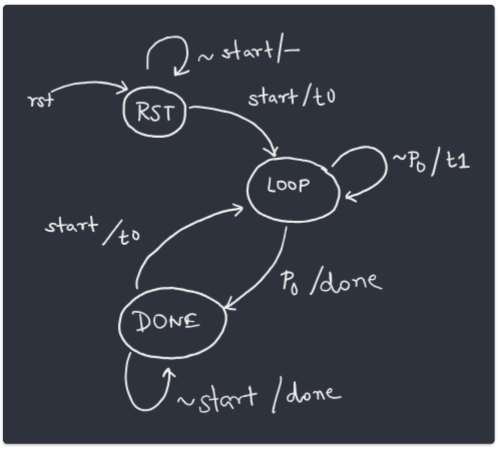
Now we can use this intuition to develop our multiplier.
Note :

- I have used generics while making the multiplier to make the `INPUT_WIDTH` variable. This helped in the later questions.
- Read the deprecated solution sub-section ahead to get an idea of the transfers and predicates.

Deprecated Solution

× Why is this deprecated?

This RTL specification assumes that the adder is able to provide the addition in the same cycle as operands. In reality, the adder provided along with this assignment has a delay of 1 clock cycle. Thus I needed to introduce an additional state `WAIT_STATE` in my implementation. Given below is the solution assuming that we are dealing with an adder without delay.
Note : Implementation submitted along with the assignment uses the earlier solution.



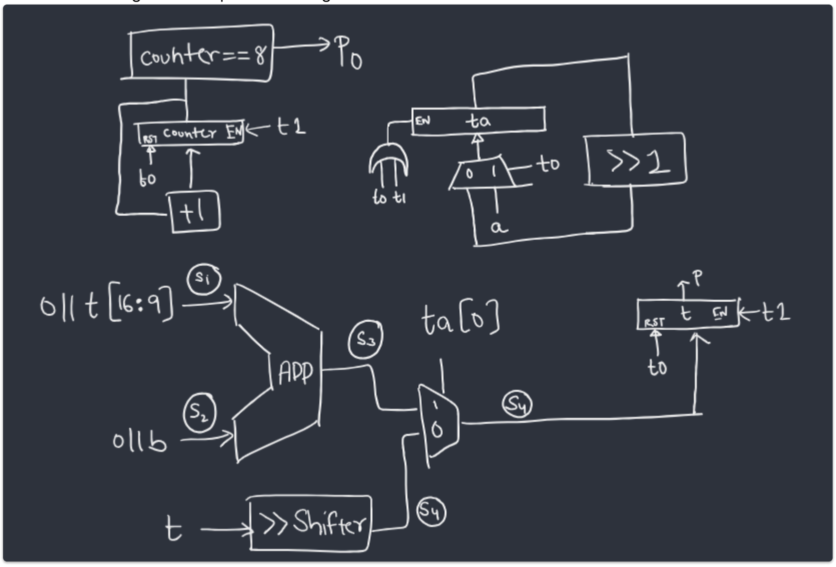
The predicates are:

$$\mathrm{P}_0 : \texttt{counter == 8}$$

The transfers are:

$$t_0 : \texttt{t[16:0] = 0, counter = 0; ta = a}$$
$$t_1 : \text{Right shift based on } \texttt{ta[0]; counter++}$$

Now we can design the Datapath side's registers



With this, we can make the datapath as a component inside of our multiplier and write the VHDL script to simulate it.

Third Part

> ✕ **Issues with the adder**
>
> The adder provided does not work when the sum of both inputs > 255 i.e. when we need 9-bits to show the true value. Therefore the requisite changes were made to both the testbench and the output port of the adder (`unsigned('0' & A) + unsigned('0' & B)` is the new output value). Moreover, in an effort to generalise the components, a generic called `INPUT_WIDTH` has also been used instead of hardcoding it to be a 8-bit adder only.

To check all 2^{16} combinations, we just need to scan over all 2^8 cases for both `a` and `b`. This means we scan from `0 to 255` for both for-loops.

After modifying the testbench from `sample/testbench.vhdl` to check for multiplication instead of addition (and also making some small changes to it so that we can be sure that the value of `a` and `b` is constant throughout the operation of the multiplier), we can see that it yields,

```
testbench.vhdl:69:25:@13762565ns:(assertion note): Success.
ghdl:info: simulation stopped by --stop-time @20ms
```

Question 2

> 🖊 **Prelude**
>
> First we will see how we can even construct such a 8-bit multiplier using four 4-bit multipliers
>
> $$a = a_L + 2^4 \times a_H$$
> $$b = b_L + 2^4 \times b_H$$
>
> Thus
>
> $$a \times b = 2^8 \times (a_H \times b_H) + 2^4(a_L \times b_H + a_H \times b_L) + a_L \times b_L$$
>
> Equivalently,
>
> $$a \times b = (a_H \cdot b_H)\texttt{<<}8 + (a_L \cdot b_H + a_H \cdot b_L)\texttt{<<}4 + a_L \cdot b_L$$
>
> Note that $\alpha(15 \text{ downto } 0) = (a_H \cdot b_H)\texttt{<<}8 + a_L \cdot b_L = (a_H \cdot b_H)\&a_L \cdot b_L$ thus we don't need to make that addition.
> Thus finally, we will need to do 3 additions using 8-bit adders with carry,
>
> - Add1 : $\beta(8 \text{ downto } 0) = a_L \cdot b_H + a_H \cdot b_L + 0$
> - Add2 : $(c_1 \,\&\, \gamma(7 \text{ downto } 0)) = \alpha(7 \text{ downto } 0) + \beta(3 \text{ downto } 0)\texttt{<<}\ 4 + 0$
> - Add3 : $(c_2 \,\&\, \delta(7 \text{ downto } 0)) = \alpha(15 \text{ downto } 8) + \beta(8 \text{ downto } 4)\texttt{<<}\ 8 + c_1$
>   Final product will then be $(\delta \,\&\, \gamma)$. We know intuitively that c_2 will always be `0` since product of 2 8-bit operands will always be 16-bit only.

First Part

We will use 4 multipliers as slave threads (each with start and done signals as `start_{i}` and `done_{i}` respectively.)

Solution that uses the provided Adder

In this solution, I have tweaked the provided adder to create `Add2_with_Carry` to create the final multiplier. With that, this is the RTL code for the master thread (the RTL code for the slave threads are already used in the last question.)

- The `RST`, `DONE` states are self-explanatory
- Master is in `WAIT` state till all slaves are done with their computation, after which it primes Add1 and transitions to `Add1_DONE`.
- During `Add1_DONE`, adder is primed again for Add2 and it transitions to `Add2_DONE`
- During `Add2_DONE`, adder is primed again for Add3 and it transitions to `DONE`

```
thread faster_8_multiplier { //
done_slaves = done_0 AND done_1 AND done_2 AND done_3
RST :   if (start) then
            start_0 = start_1 = start_2 = start_3 =  1
```

```
            goto WAIT
        else
            start_0 = start_1 = start_2 = start_3 =  0
            goto RST
        endif
WAIT :  start_0 = start_1 = start_2 = start_3 =  0 // in case one of them ends their operation early, they should wait to synchronise
        if (done_slaves)
            adder_a := aLbH
            adder_b := aHbL
            adder_cin := '0'
            adder_start := '1'
            goto Add1_DONE
        else
            goto WAIT
        endif
Add1_DONE:
        if (done_adder)
            beta := adder_out
            adder_a := adder_out(3 downto 0) << 4
            adder_b := aLbL
            adder_cin := '0'
            adder_start := '1'
            goto Add2_DONE
        else
            goto Add1_DONE
        endif

Add2_DONE:
        if (done_adder)
            gamma := adder_out(7 downto 0)
            adder_a := beta(8 downto 4)
            adder_b := aHbH
            adder_cin := adder_out(8)
            adder_start := '1'
            goto DONE
        else
            goto Add2_DONE
        endif

DONE :  if start then
            start_0 = start_1 = start_2 = start_3 =  1
            goto WAIT
        else:
            start_0 = start_1 = start_2 = start_3 =  0
            goto DONE
        endif;
}
```

Deprecated Solution

The RTL Code for the Master will be as follows,

```
thread faster_8_multiplier {
done = done_0 AND done_1 AND done_2 AND done_3
RST :   if (start) then
            start_0 = start_1 = start_2 = start_3 =  1
            goto WAIT
        else
            start_0 = start_1 = start_2 = start_3 =  0
            goto RST
        endif
WAIT :  start_0 = start_1 = start_2 = start_3 =  0 // in case one of them ends their operation early, they should wait to synchronise
        if (done)
            goto DONE
        else
            goto WAIT
        endif
DONE :  if start then
            start_0 = start_1 = start_2 = start_3 =  1
            goto WAIT
        else:
            start_0 = start_1 = start_2 = start_3 =  0
            goto DONE
        endif;
}
```

The RTL code for each of the four slaves will be identical to what was defined in the First Question (except everything is for 4-bit instead of 8-bits.)

Second Part

The VHDL code is attached along with this pdf file.

Third Part

Testing with the testbench yields

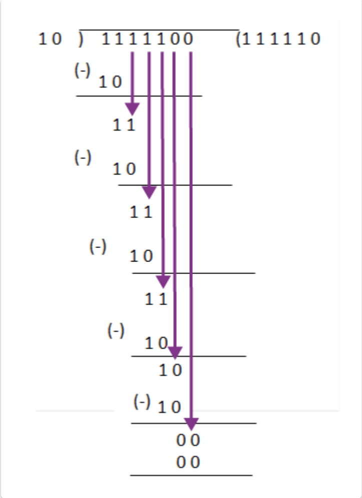
```
testbench.vhdl:69:25:@8519685ns:(assertion note): Success.
ghdl:info: simulation stopped by --stop-time @20ms
```

This is significantly faster than the earlier result (whose testing lasted `13762565ns`).

Question 3

First Part

To implement a shift-and-subtract model for calculation of division, we will use the same algorithm as given here



Above is the hand-written version of the algorithm.

Instead of using a register to sample the first few bits of the dividend, we will simply extend the MSBs of the dividend and left-shift after each step.

The RTL code for this is,

```
// Shift and subtract divider.
// Uses a single 9-bit subtractor.
//
// input start, a[7:0], b[7:0]
// output done, q[7:0]
//
// register ta[15:0]
// register t[7:0]
// registers counter[3:0]
//
// Default outs:
// done=0, q = 0
b_is_zero := b xnor 0
q := t

rst:
    if start then
        ta := 0^8 & a
        counter := 0
        t := 0
        goto loop
    else
        goto rst
    endif

prime:
    main_subtractor_start := 1
    main_subtractor_a := ta[14:7]
    main_subtractor_b := b

    counter_adder_start := 1
    counter_adder_a := counter
    counter_adder_b := 1

loop:
    if (! (counter_adder_done AND main_subtractor)) // We are waiting for the counter and subtractor to update
        goto loop
    else if (main_subtractor_diff >= 0) then
        ta[15:8] := main_subtractor_diff
        // simple left-shift
        ta[7:0] := ta[6:0] & 0
        // left-shift quotient
        t := (t[7:1] & 1) if not b_is_zero else 0
    else
        // shift left
        ta[15:0] := ta[14:0] & 0
        t := (t[7:1] & 0) if not b_is_zero else 0
    endif

    if (counter == 7 OR b_is_zero = '1') then
        goto done_state
    else
        counter := counter_adder_out
        goto loop
    endif

done_state:
    done = 1
    // t is visible at p
```

```
    if start then
        t := 0
        counter := 0
        ta := 0^8 & a
        goto loop
    else
        goto done_state
    endif
```

- Notice that we have incorporated `b_is_zero` signal to deal with the case when `b` is provided as `0`. It allows the ckt to go through the usual 8 iterations of `loop` (to ensure that the divider has consistent timing across all stages)
- Also note that we are using a realistic subtractor (by changing the `Add2` entity slightly). Thus there is a cycle delay between us giving the operands and receiving the correct values.

Second Part

The VHDL Code is attached along with this pdf. Luckily the FSM of control path for divider is very similar to that of the multiplier thus I didn't need to change that by a large margin. Hence no diagram attached for the same.

Third Part

Testbench was configured to expect $q = 0$ in case $b = 0$.

```
testbench.vhdl:75:25:@13762565ns:(assertion note): Success.
ghdl:info: simulation stopped by --stop-time @20ms
```

Question 4

First Part

Since the squarer-root calculator has to work for 8-bit values of x, the maximum value of $y_{\max} = \sqrt{\max(x)} < \sqrt{256} < 16$. Since $y \in \mathbb{Z}$, $y_{\max} = 15$. Thus to find the value of square-root of x, we need to ideally calculate every value of y^2 from $y = 0$ to $y = 15$.

However, we can use some shortcuts :

1. Since $\{i^2\}_{i=0}^{15}$ is an ordered list, we can utilise binary search i.e. start with $y = 7$ and utilise comparators
2. The largest $y \in \mathbb{Z}$ which follows y^2 smaller than x will also follow:

$$y^2 \leq x < (y+1)^2$$

   Thus we need to compute $p' = x - y^2$ and $q' = p' - 2y - 1$. If the carry bit in p' is `0` and carry bit in q' is `1` (since compulsorily x should be smaller than $(y+1)^2$), we can be sure we have found the real value of y.

Second Part

RTL Code for this ckt (while using the 4-bit multiplier as a slave thread) is,

```
// Multiplier based square-root calculator.
// Uses a single 4-bit multiplier and a couple adders/subtractors
//
// input start, x[7:0]
// output done, y[3:0]
//
// register ty[7:0]
// register t[4:0]
// register upper[4:0], lower[4:0]
//
// Default outs:
// done=0, y = 0

multiplier(a=>mul_a, b=>mul_b, start=>start_mul, p=>ty, done=>done_mul)
subtractor
adder

rst:
    if start then
        t := 5b'6
        upper := 4b'15
        lower := 4b'0
        mul_a := t
        mul_b := t
        start_mul := '1'
        goto multiply_state
    else
        goto rst
    endif

multiply_state:
    if (done_mul = '1') then
        subtractor_a := x
        subtractor_b := ty
        subtractor_start := 1
        goto preloop_state
    else
        goto multiply_state
    endif

preloop_state:   // This state exists purely to find the value of x_minus_tsquare
    x_minus_tsquare := subtractor_diff  // To preserve the value of x_minus_tsquare
```

```
    if (! subtractor_done)
        goto preloop_state
    else if (subtractor_diff >= 0) then
        subtractor_a := subtractor_diff // we feed it back in
        subtractor_b := (t << 1 & 1) // this is equivalent to 2t + 1
        subtractor_start := 1
        goto loop_state

    else // No point in finding x_minus_ therefore we can save a cycle by skipping loop
        adder_start := 1 // we are updating value of upper since ty > x
        adder_a := t
        adder_b := -1
        goto postloop1_state

loop_state: // Now we have valid values of both x_minus_tsquare and q_
    subtractor_start := 0
    x_minus_tsquare := subtractor_diff
    if (! subtractor_done)
        goto loop_state
    else if (subtractor_diff < 0) then // That is, both x_minus_tsquare >= 0 and x_minus_tsquare < 0 (since you can only reach loop_state if x_minus_tsquare >= 0)
        // Found the correct value
        start_mul := '0'
        goto done_state
    else
        adder_start := 1 // we are updating value of upper or lower
        if (x_minus_tsquare >= 0) then // we are at a value of t lower than y
            adder_a := t
            adder_b := 1
        else // we are at a value of t higher than y
            adder_a := t
            adder_b := -1
        goto postloop1_state
    endif

postloop1_state: // in this state, we update the value of t
    adder_start := 1
    if (x_minus_tsquare >= 0) then // we are at a value of t lower than y
        lower := adder_c
        adder_a := upper
        adder_b := adder_c
    else // we are at a value of t higher than y
        upper := adder_c
        adder_a := lower
        adder_b := adder_c
    endif
    goto postloop2_state

postloop2_state: // finally we update the value of t
    adder_start := 0
    t := adder_c >> 1
    start_mul := '1'
    mul_a := adder_c >> 1
    mul_b := adder_c >> 1
    goto multiply_state

done_state:
    done := 1
    y := t[3:0]
    if start then
        t := 5b'6
        upper := 4b'15
        lower := 4b'0
        start_mul := '1'
        goto multiply_state
    else
        goto done_state
    endif
```

- Yes, there are a lot of states in this RTL. The reason for that is simply that I wanted to reduce the number of components being used while still using binary search-based technique for finding the correct value of the squareroot. I use the adder, subtractor twice for every iteration of the loop.
- Since there are a lot more states in this ckt, I have not used an RTL-based method of division into separate control and datapaths.
- I used two 2-bit multipliers to make up the 4-bit multiplier using the master-slave configuration.

Third Part

I got the following output on running the testbench,

```
testbench.vhdl:66:25:@98795ns:(assertion note): Success.
ghdl:info: simulation stopped by --stop-time @500us
```