EE 789 : Assignment 1

Rohan Rajesh Kalbag
20D170033

September 2023

1 Shift and Add Algorithm for Multiplication

1(a) Describe the algorithm using Aa, and write a testbench for the design.

1.1 Pseudo Code For the Algorithm

```
shift_and_add_multiply(a : unsigned 8-bit, b : unsigned 8-bit)
returns -> product : unsigned 16-bit
{
    sum: unsigned 16-bit
    count : unsigned 16-bit
    curr_a : unsigned 16-bit
    curr_b : unsigned 8-bit

    sum = 0 //set running sum to 0
    count = 0
    curr_b = b
    curr_a  = convert_to_16bit(a)

    while curr_b is non zero
    {
        if last bit of curr_b is 1{
            sum = sum + (curr_ a << count) //add a shifted version of a to sum
        }
        count = count + 1
        curr_b = curr_b >> 1
    }
    product = sum
}
```

1.2 Test Vector Generation

The strategy used to generate the test vectors, was to include at least one number from each of the 25 interval of ten numbers [0-10], [10-20]...[240-250], and the final interval of [250-255], a python script was used to generate 52 values of a and b, each having two representatives of each interval and both were shuffled randomly following this python script

```python
import random
# for a
l = [i + random.randint(0, 9) for i in range(0, 250, 10)]
l.extend([i + random.randint(0, 9) for i in range(0, 250, 10)])
l.append(250 + random.randint(0, 5))
l.append(250 + random.randint(0, 5))
random.shuffle(l)
print(l)
# for b
l = [i + random.randint(0, 9) for i in range(0, 250, 10)]
l.extend([i + random.randint(0, 9) for i in range(0, 250, 10)])
l.append(250 + random.randint(0, 5))
l.append(250 + random.randint(0, 5))
random.shuffle(l)
print(l)
```

```
rohankalbag@Rohan:~/ee720-lib/assignment-1$ python3 test_generator.py
[194, 202, 240, 148, 199, 3, 155, 35, 187, 28, 177, 41, 182, 209, 126, 159, 117, 59, 249, 163, 75, 50, 4, 110, 99, 139, 228, 210, 250, 172, 215, 73, 221, 47, 20, 145, 14,
 167, 92, 89, 18, 81, 64, 252, 102, 109, 35, 233, 231, 128, 130, 64]
[170, 237, 246, 49, 45, 222, 19, 168, 0, 202, 33, 156, 74, 39, 142, 88, 197, 156, 241, 20, 112, 124, 63, 238, 188, 250, 250, 143, 96, 125, 54, 189, 9, 90, 89, 113, 27, 74
, 53, 68, 204, 177, 10, 106, 217, 199, 224, 137, 162, 107, 215, 134]
```

1.3 Description of Algorithm using the AA language

```
$module [shift_and_add_mul] $in (a b: $uint<8>) $out (product: $uint<16>) $is
{
    $branchblock[loop] {
        $merge $entry loopback
            $phi curr_a := ($bitcast ($uint<16>) a) $on $entry next_a $on loopback
            $phi curr_b := b $on $entry next_b $on loopback
            $phi sum := ($bitcast ($uint<16>) 0) $on $entry next_sum $on loopback
            $phi count := ($bitcast ($uint<16>) 0) $on $entry next_count $on loopback
        $endmerge

        $volatile continue_flag := (curr_b != 0)
        $volatile add_shifted := ((curr_b & 1) != 0)
        $volatile shifted_sum := (sum + (curr_a << count))

        next_count := (count + 1)
        next_sum := ($mux add_shifted shifted_sum sum)
        next_a := curr_a
        next_b := (curr_b >> 1)


        $if continue_flag $then
            $place [loopback]
        $else
            product := sum
        $endif
    }
}
```

1.4 C Testbench Used

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>
#include <pthreadUtils.h>
#include <Pipes.h>
#include <pipeHandler.h>
#ifndef SW
#include "vhdlCStubs.h"
#endif

int main(int argc, char *argv[])
{
        uint8_t a[52] = {194, 202, 240, 148, 199, 3, 155, 35, 187, 28, 177, 41, 182, 209,
        ↪  126, 159, 117, 59, 249, 163, 75, 50, 4, 110, 99, 139, 228, 210, 250, 172, 215,
        ↪  73, 221, 47, 20, 145, 14, 167, 92, 89, 18, 81, 64, 252, 102, 109, 35, 233, 231,
        ↪  128, 130, 645};
        uint8_t b[52] = {170, 237, 246, 49, 45, 222, 19, 168, 0, 202, 33, 156, 74, 39, 142,
        ↪  88, 197, 156, 241, 20, 112, 124, 63, 238, 188, 250, 250, 143, 96, 125, 54, 189,
        ↪  9, 90, 89, 113, 27, 74, 53, 68, 204, 177, 10, 106, 217, 199, 224, 137, 162,
        ↪  107, 215, 134};

        uint16_t c;
        int pass_tests = 0;
        int curr_testcase = 0;
        while (curr_testcase < 52)
        {
                c = shift_and_add_mul(a[curr_testcase], b[curr_testcase]);
                if (c == a[curr_testcase] * b[curr_testcase])
                {
                        pass_tests += 1;
                        fprintf(stdout, "PASS : mul(%d, %d) == %d\n", a[curr_testcase],
                        ↪  b[curr_testcase], c);
                }
                else
                {
                        fprintf(stdout, "ERROR : mul(%d, %d) != %d : Expected %d\n",
                        ↪  a[curr_testcase], b[curr_testcase], c, a[curr_testcase] *
                        ↪  b[curr_testcase]);
                }
                curr_testcase += 1;
        }

        fprintf(stdout, "%d testcases out of %d PASSED\n", pass_tests, 52);
        return (0);
}
```

1(b) Generating the VHDL and verifying it using the GHDL simulator and the C test-bench.

```
make
```

After compilation, use tmux and make two terminals for the docker container and run the following

```
# terminal 1
./testbench_hw
```

```
# terminal 2
./ahir_system_test_bench --wave=waveform.ghw
```

1.5 Terminal Output Obtained for ./testbench_hw

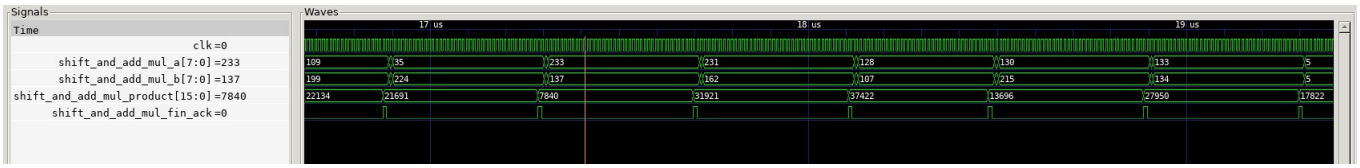
```
PASS : mul(18, 204) == 3672
PASS : mul(81, 177) == 14337
PASS : mul(64, 10) == 640
PASS : mul(252, 106) == 26712
PASS : mul(102, 217) == 22134
PASS : mul(109, 199) == 21691
PASS : mul(35, 224) == 7840
PASS : mul(233, 137) == 31921
PASS : mul(231, 162) == 37422
PASS : mul(128, 107) == 13696
PASS : mul(130, 215) == 27950
PASS : mul(133, 134) == 17822
52 testcases out of 52 PASSED
```

1.6 GTKWave Waveform generated by GHDL Simulation

Using the screenshot of `waveform.ghw` generated while solving this assignment

```
# use on host system to view waveform
gtkwave waveform.ghw
```

Signals for last few multiplications is attached below, corresponding to those shown in above terminal



1(c) Study the generated VHDL and sketch the structure of the generated hardware (show the modules, their internal structure, and their interactions as you observe them in the VHDL)

1.7 RTL Machine Based Hardware Structure Analysis

We perform an analysis of the algorithm using the RTL Machine method (as done in class) to get an idea of the hardware we can expect to be generated

1.7.1 Registers Used

The registers used are 8 bit **count** to hold the current iteration, 8 bit **curr_a** to hold the value of A, 8 bit **curr_b** to hold the value of B and shifted value of B, and 16 bit **sum** to hold the value of the running sum.

1.7.2 RTL Transfers

Here `SE16` denotes sign extension of an 8 bit number to its 16 bit representation and the `<<` and `>>` are the bit shift operations

```
t0 :  count := count + 1

t1 :  curr_a := SE16(a_in)

t2 :  curr_b := SE16(b_in)

t3 :  sum := ((curr_b & 1) != 0) ?  (sum + (curr_a << count)) :  sum

t4 :  curr_b := (curr_b >> 1)

t5 :  count := 0

t6 :  sum := 0
```

1.7.3 Predicates

```
u :  curr_b != 0
```

1.7.4 Control Path

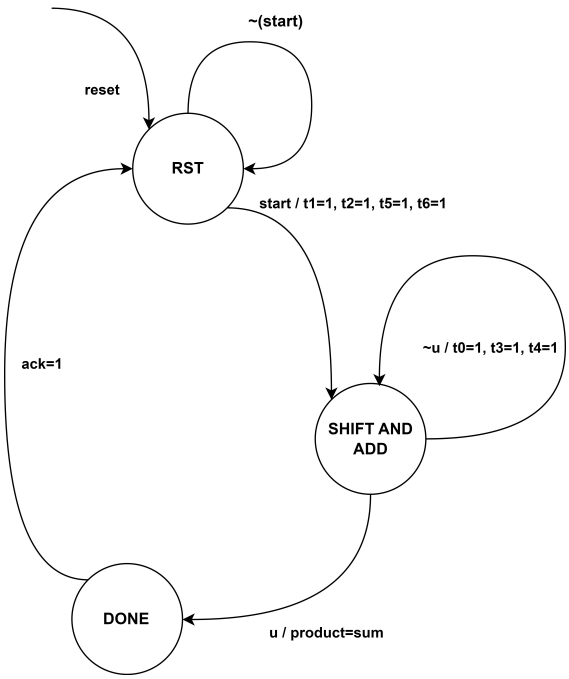


Figure 1: The Control Path (flowchart drawn using draw.io software)

1.7.5 Data Path

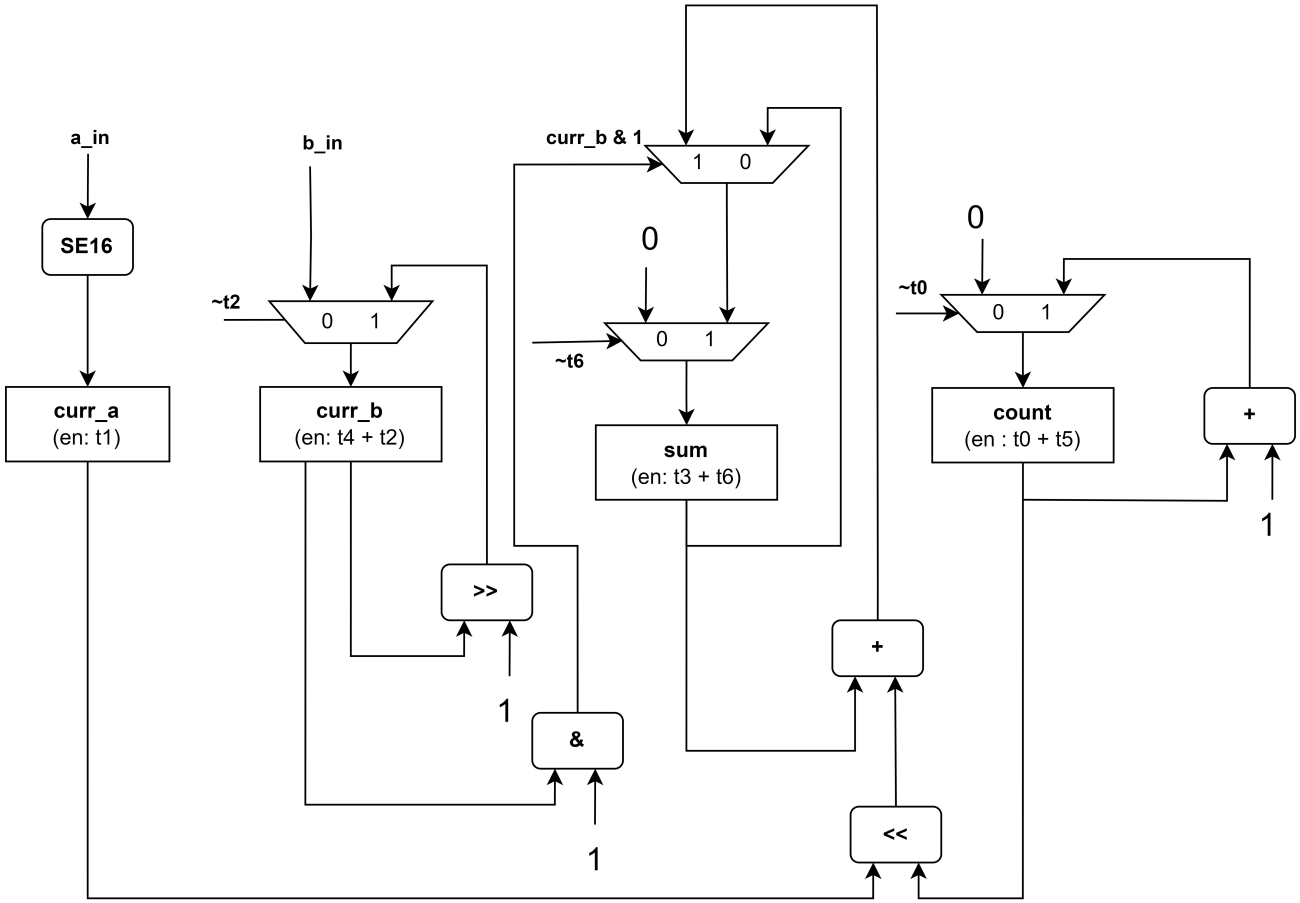


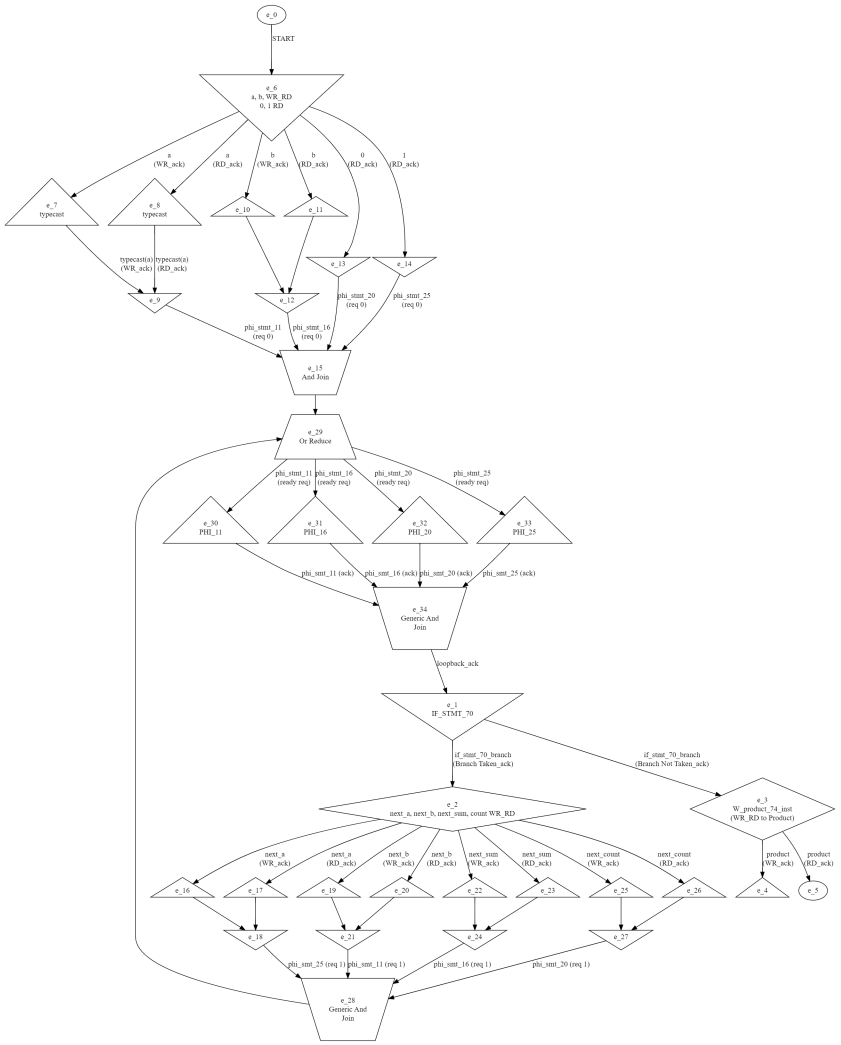
Figure 2: The Data Path (schematic drawn using draw.io software)

When we see the optimized hardware from theoretical RTL machine, we can expect that there will be **1 SHR block, 1 SHL block, 1 AND block, 2 ADD blocks and 4 registers**. We can also expect that there will be **1 NEQ block** to check our predicate **U**

1.8 Generated Hardware Analysis

AHIR generates `shift_and_add_mul_DP.dot` and `shift_and_add_mul_CP.dot`, visualization of this using GraphViz results in the following diagram which can be used to understand generated hardware. The tool to generate the schematics was `https://dreampuf.github.io/GraphvizOnline/`

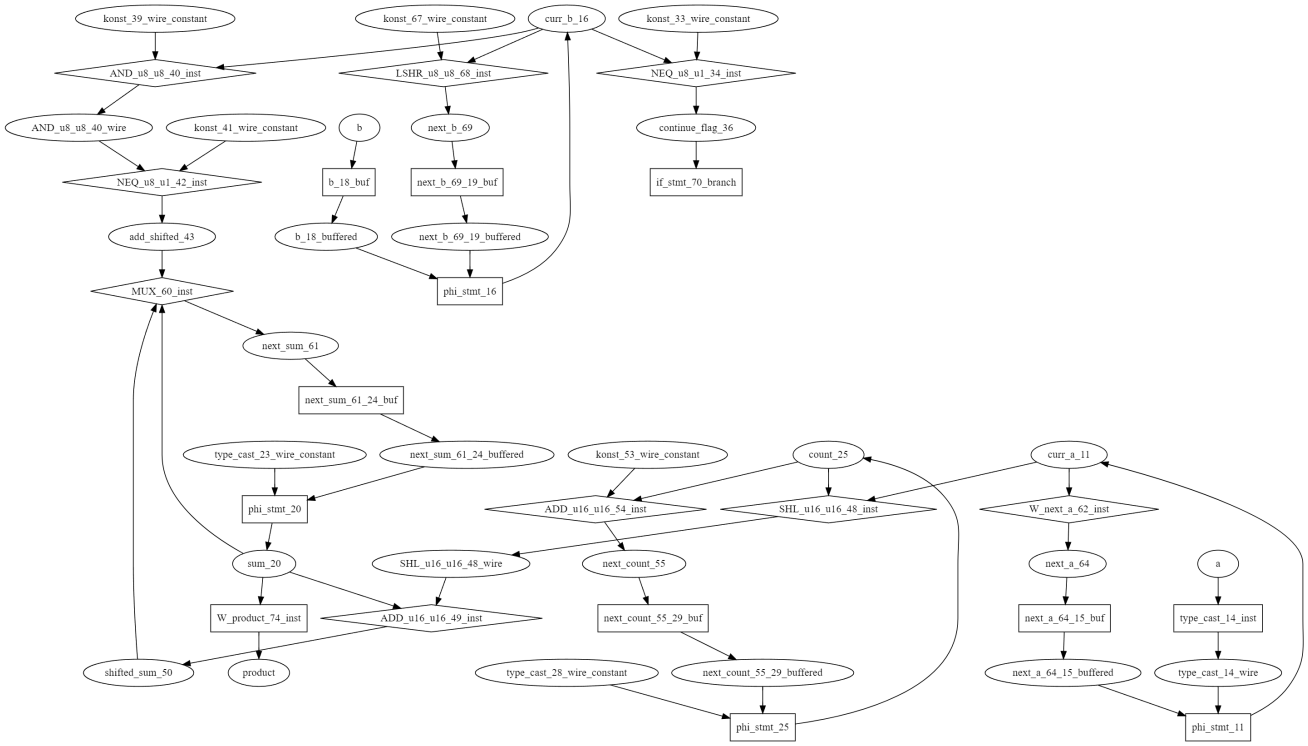
1.8.1 Generated Control Path



1.8.2 Comparison with Theoretical RTL Machine Control Path

AHIR generates a `.dot` file with a digraph with only the node names e_i. On studying the generated VHDL and using adding labels to nodes and edges of the generated .dot file to explain their nodes. The above figure was obtained. We see that there are WR and RD req, ack control signals for each variable and constant (sum, count, a, b, 0, 1), there is a control signal for branch taken or not taken (`IF_STMT_70`), there is also req signals for the SSA Phi statements corresponding to each operand, and also ack signals to indication completion of computing the Phi assignment. **We see that the generated control path is equivalent in functionality to the RTL machine's control path, but has more control signals, nodes and edges (Not completely optimized)**.

1.8.3 Generated Data Path



1.8.4 Comparison with Theoretical RTL Machine Data Path

We see that there is `1 AND block:(AND_u8_u8_40_inst)`, to mask the last bit of curr_b as expected. We see that there is `1 SHR block:(LSHR_u8_u8_68_inst)` to perform the shift right as expected. We see that there is `1 SHL block:(SHL_u16_u16_48_inst)` to perform the shift left as expected. We see that there are `2 ADD blocks:(ADD_u16_u16_49, ADD_u16_u16_54)` to add for count and sum respectively as expected. When we see the generated hardware we see that there are `2 NEQ blocks:(NEQ_u8_u1_34_inst, NEQ_u8_u1_42_inst)`, one to check for predicate, and the other to check for the last bit of curr_b, in the theoretical RTL machine we directly drive this signal to a MUX without checking NEQ with zero. **We see that the generated data path is equivalent in functionality to the RTL machine's data path, but has more compute units, phi blocks, components (Not completely optimized)**.