C Based VLSI Design – ESA
UE18EC400SA

Name: Rajath J, Bharath V
SRN: PES1201801707, PES1201802045

Aim: For an operation a+b/10, write the equivalence between RTL and C with appropriate methods.

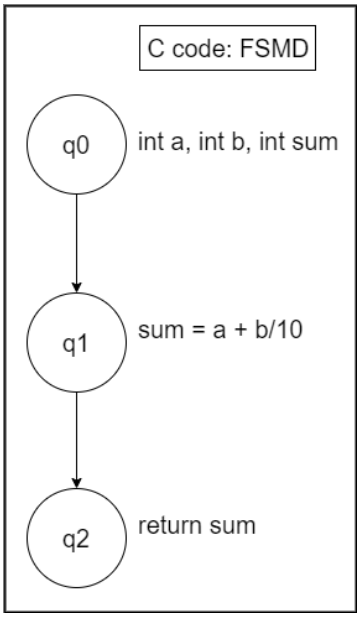
Introduction:

The main issue for C to RTL equivalence checking is the semantic gap between the C and Verilog codes. The Inputs and Outputs may be the same. But the control structures are different and the internal variables are completely different.
One way to overcome this is the extraction of RTL C from the RTL to reduce the semantic gap between C and RTL. This is done by equivalence checking of the FSMD's of RTL and C. This is done by Merging compatible traces within a behavior to handle control structure modification during HLS. Use of a data-driven approach to finding the potential equivalent trace pairs between two behaviors. The equivalence of potential equivalence traces is then formally proved.

Equivalence of two FSMDs:

For an execution trace in one FSMD, there is an equivalent one in the other FSMD. For any computation/trace c_0 of M_0 there exists a computation/trace c_1 in M_1so that $c_0 \approx c_1$.

We use the **Trace-based approach** in this report.
To do this we first need to generate the FSMD for C and RTL. Then perform the Equivalence checking.

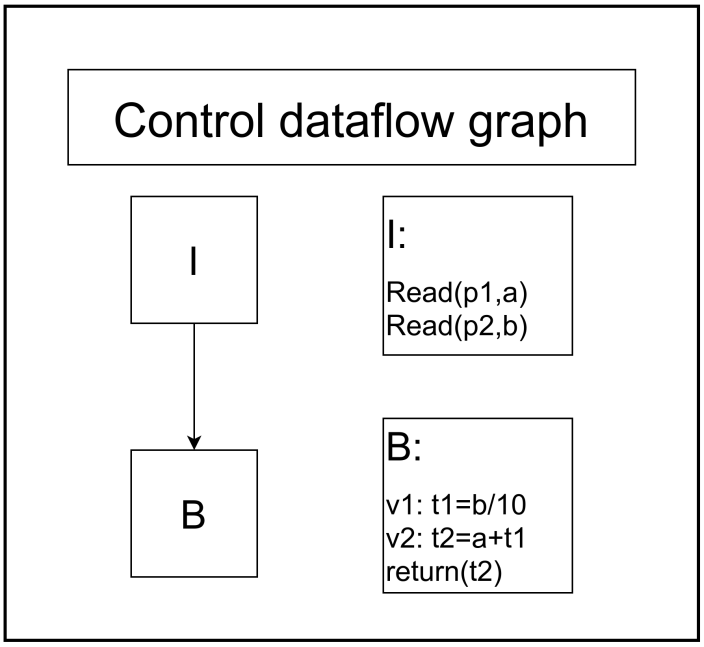


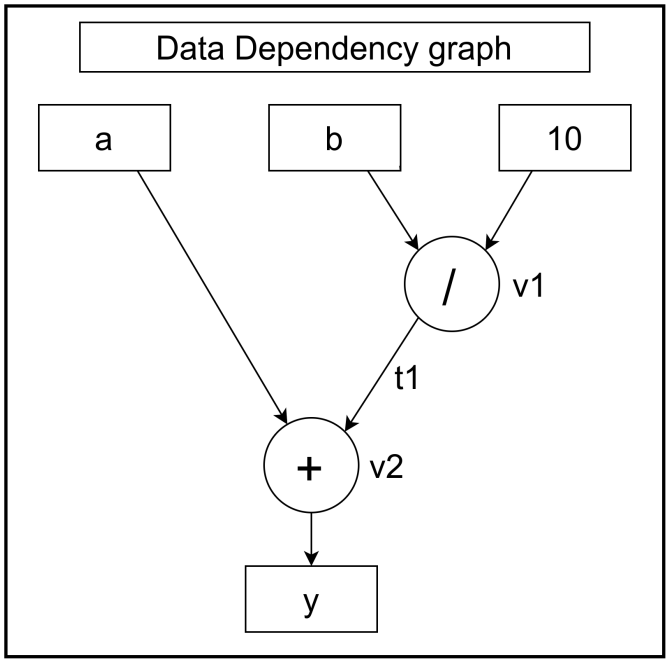
FSMD for C code:

The FSMD for the C code is simple and pretty straightforward. The inputs "a" and "b" are fed into the program. We define the variable "sum" to hold the result. Within the function, the expression is evaluated and the result is stored in sum. The sum value is returned in the final step.

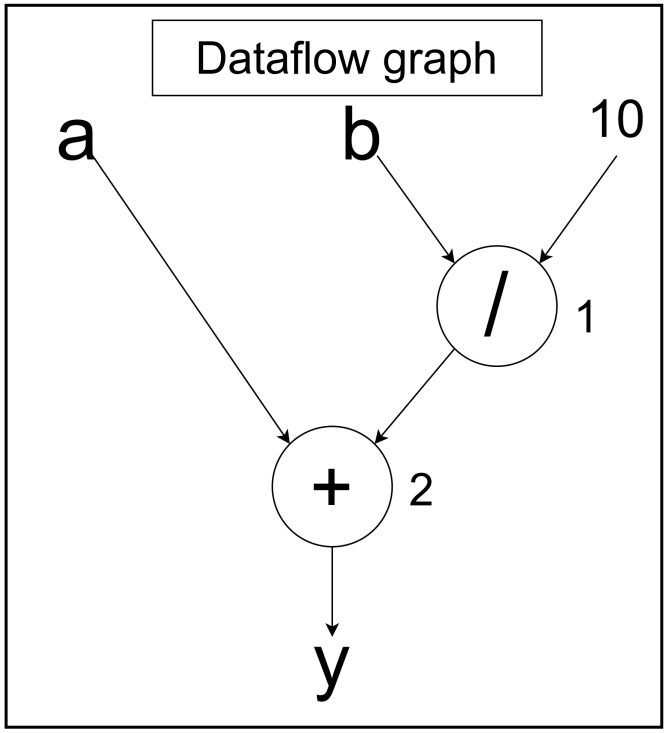
FSMD for RTL:

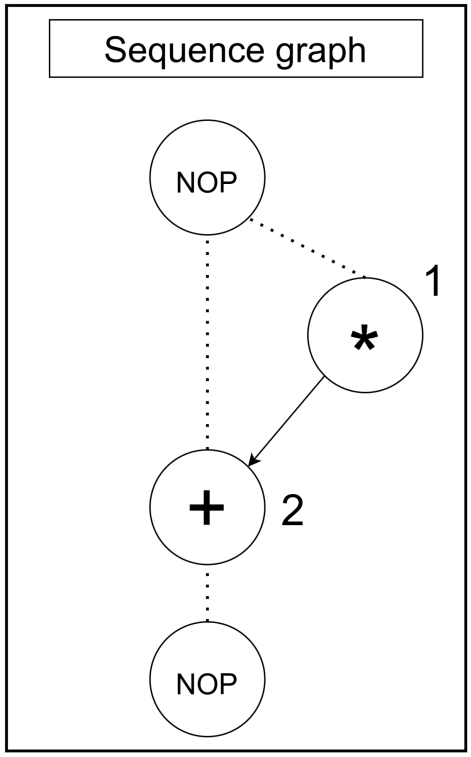
First, we generate the **control flow graph** and the **data dependency graph**.



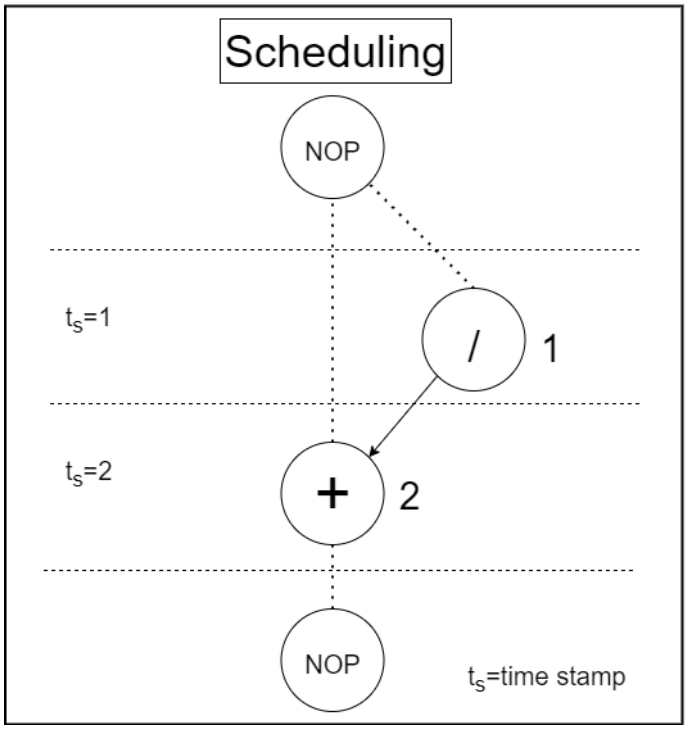


From the data dependency graph, we arrive at the **dataflow graph** and **sequence graph.**

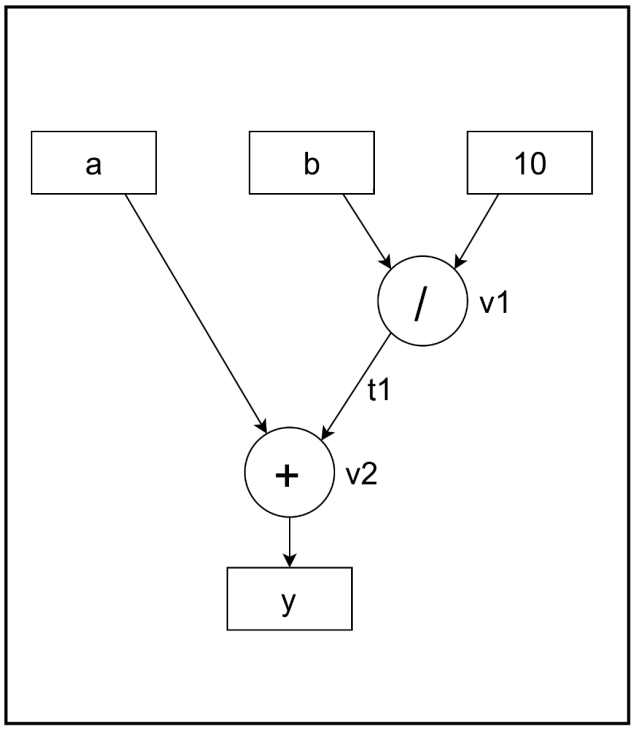


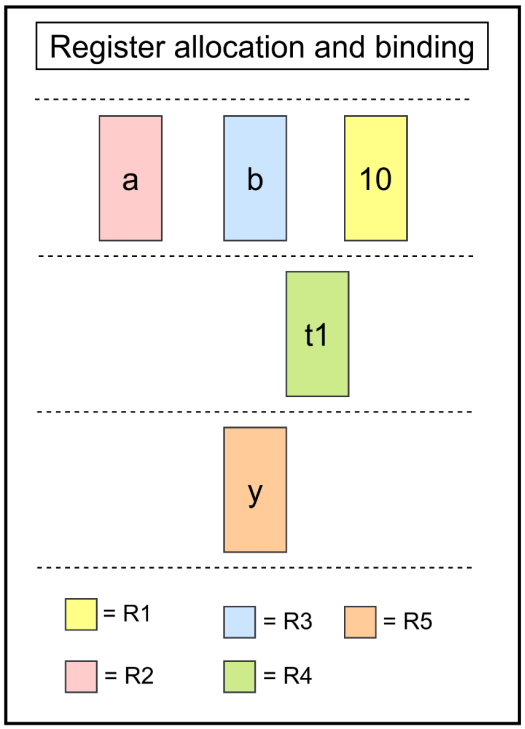


Scheduling of operations:



Register allocation and binding:





Based on the number of inputs, intermediate variables, and outputs we need a maximum of 5 registers: R1, R2, R3, R4, R5

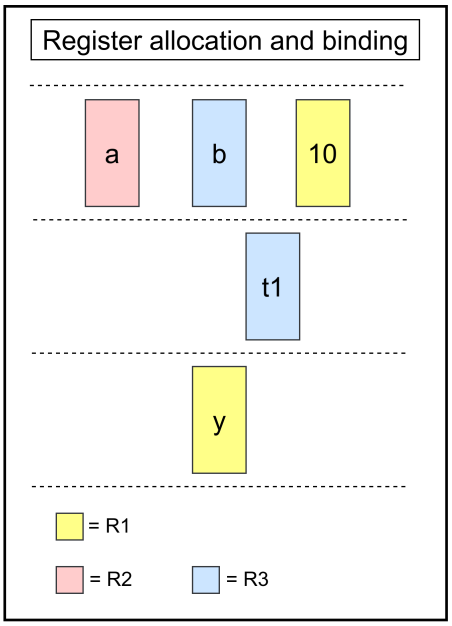
R1 = 10

R2 = a

R3 = b

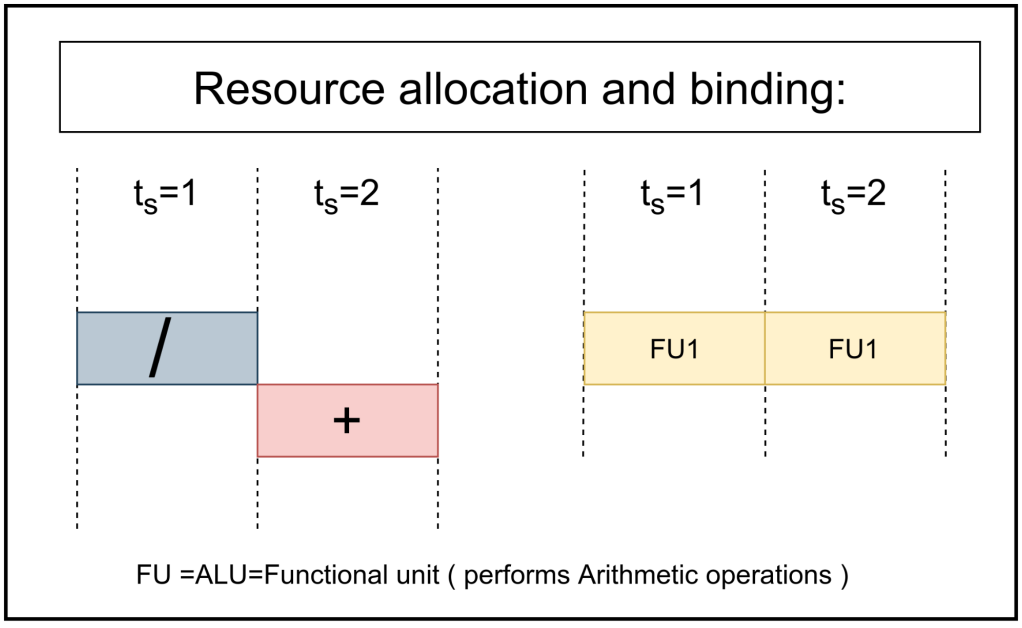
R4 = t1

R5 = y



Register allocation and binding can be optimized as a **graph coloring problem**: we would need a maximum of 3 registers: R1, R2, and R3. In the later timestamps stamps, the same registers(R2 and R3) can be reused

Resource Allocation and binding:



(DIV) D1: v1

(ADD) A1: v2

v1, v2 are operations, and D1, A1 are the operators to perform division and addition respectively.

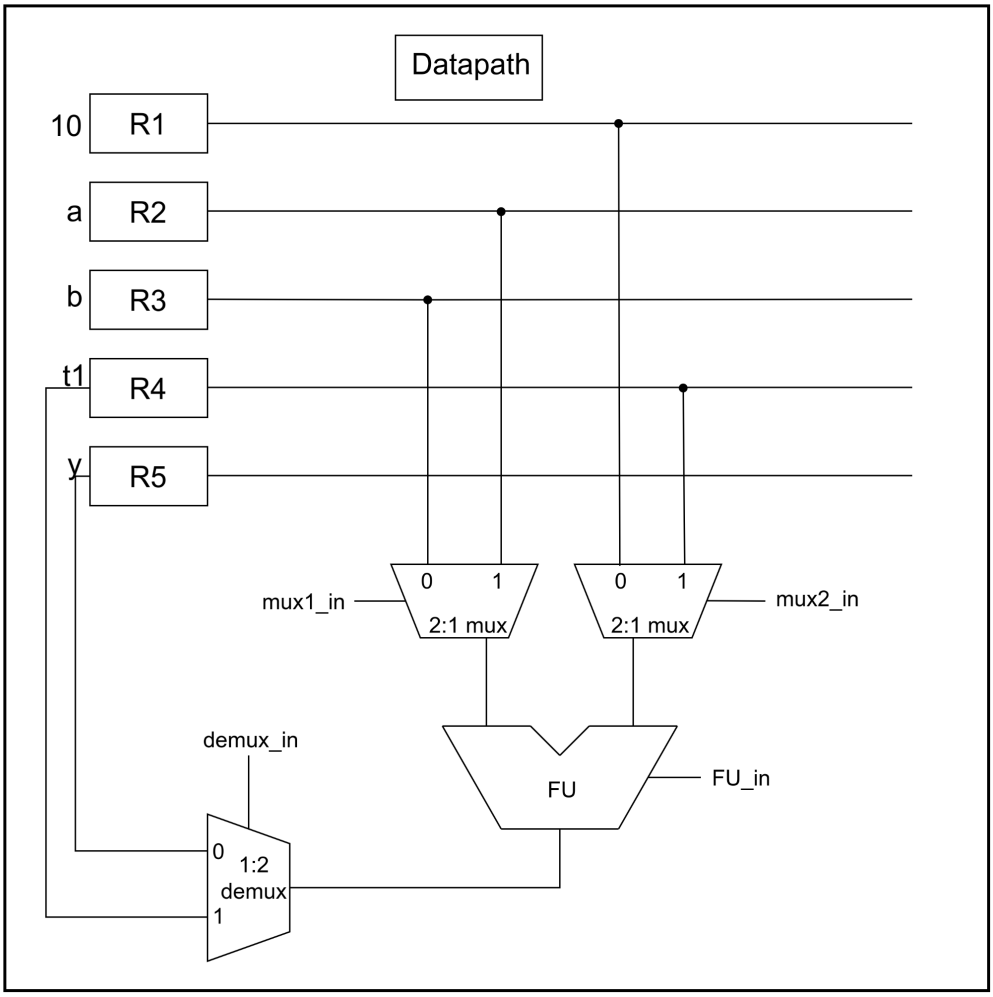
RTL behavior based on schedule and available resources:

S1: R4 = R3 <D1> R1

S2: R5 = R4 <A1> R2

We could also use a functional unit, FU1 here capable of both division and addition. And use an input signal to perform division or addition as required.

Data path Synthesis:



Controller synthesis:

The next step is to generate the control signals. We need a total of 9 control signals.

The CS tuple would be
<FU_IN, mux1_in, mux2_in, R1 reg_en, R2 reg_en, R3 reg_en, R4 reg_en, R5 reg_en,demux_in>

FU_IN = 1 bit
mux1_in, mux2_in = 2 bits
R1 reg_en = 5 bits
Demux_in = 1 bit
Total **9 bits** are required for control signals

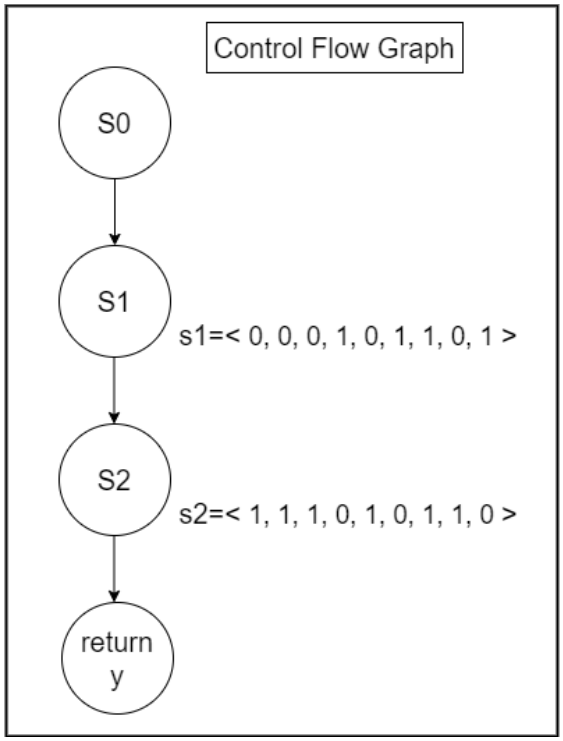
FU_IN= 0 - division

1- addition

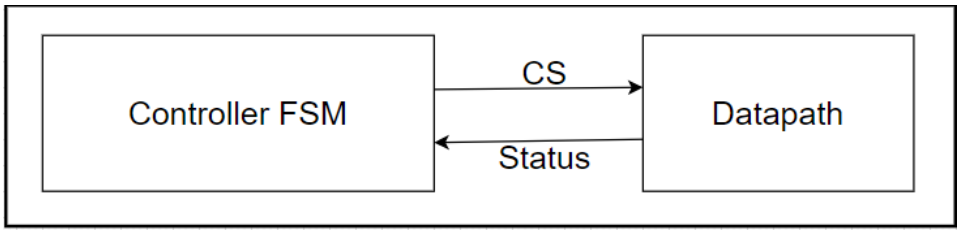
Therefore the value of the signals would be.

s1=< 0, 0, 0, 1, 0, 1, 1, 0, 1 >
s2=< 1, 1, 1, 0, 1, 0, 1, 1, 0 >

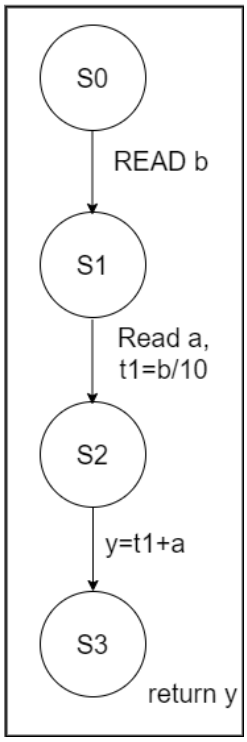
The Control Flow graph with the respective signals in each step is depicted in the figure.



The **Controller FSM** and **Datapath** are linked as depicted below.



Finite State Machine with Dataflow:



We now try to rewrite the above graphs to an FSMD.

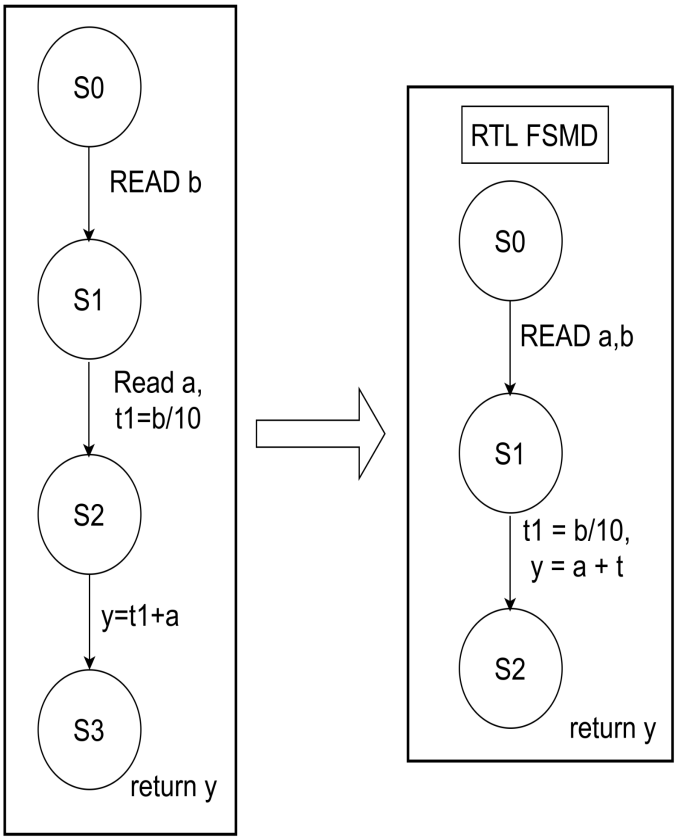
Q = {S0, S1, S2, S3} // finite set of control states
S0 ∈ Q, is the reset state
I = {a, b} // set of primary signals
V = {a, b, t1} // set of storage variables
O = {y} // set of output signals

The FSMD exactly follows the flow of the RTL program.

We now follow the second step of merging equivalent states, **Trace merge**.

This type of merging is justified because nowadays we have very fast circuits which can do both operations concurrently.

This statement is proved in the tool-based simulation part. Hence both the read can be done in one state and similarly both the computations can be done in one state.



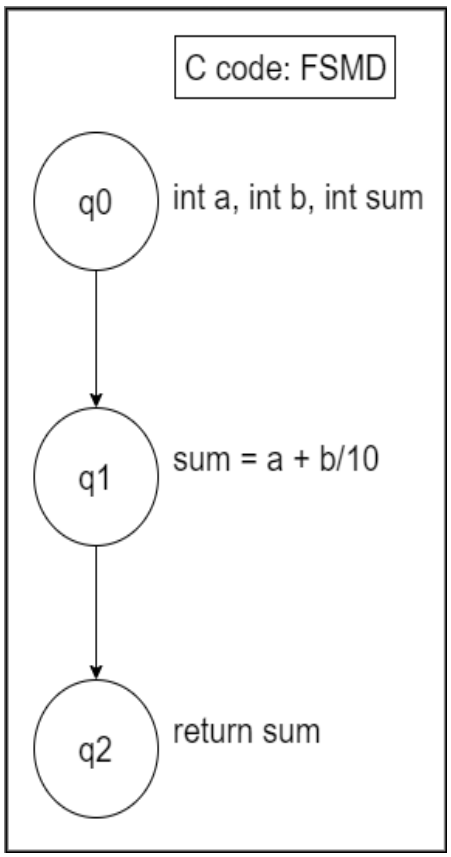
Finding Equivalence for both the FSMD's:

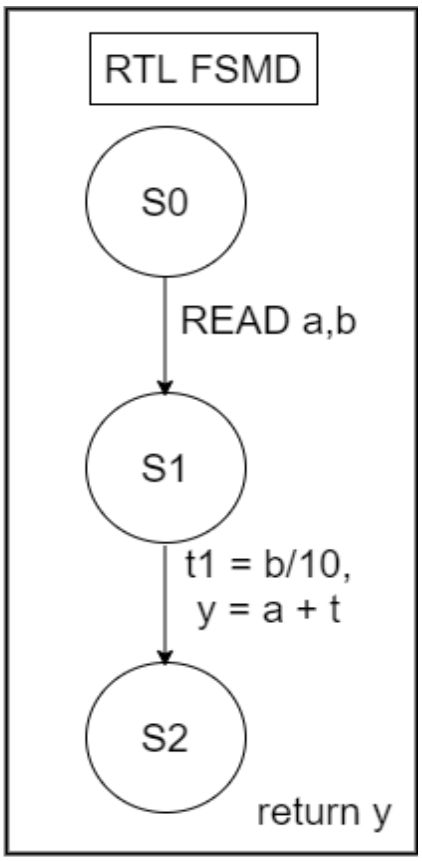
We use the C_to_RTLcheck_eq algorithm specifically **trace level equivalence** for computing the equivalence.

1. According to this algorithm, we chose a trace T0 from C and T1 from RTL.
2. A test case is passed to each of these traces.
3. The test case result is assumed to be equivalent for both methods.
4. If the results are equivalent we pick the next trace else we end our algorithm.
5. If fully complete we can finally say that both the FSMD's are equivalent.

Applying it for the below graphs.

- The first trace of both the FSMD's(q0 and S0) are equivalent.
- The second trace(q1 and S1) evaluates the same output expression in both the traces i.e., we get the same output in both cases, C code and RTL.





Therefore the FSMD's are equivalent. The Equivalence for C and RTL is proved.

Tool based verification:

To verify and evaluate the result obtained above. We use Vivado HLS to perform an RTL/C co-simulation.

The C program:

To add the C programs to Vivado HLS. We need to separate the function and the main file. This is because the main function in C is not synthesizable, therefore Vivado treats this file as the test bench and also uses the same for the RTL testbench generation.

The “eq.c” file contains our function which performs the operation of “a+b/10”. The test bench and function file are linked with the help of the “eq.h” header file. The definition of EQ_H_ is found in the header file, as Vivado uses this name for the RTL circuit.

```c
#include "eq.h"
float eq(int a, int b) {
    float sum;
    sum = a + b/10;
    return sum;
}
```

```c
#ifndef EQ_H_
#define EQ_H_

float eq(int a, int b);

#endif
```

```c
#include <stdio.h>
#include "eq.h"

int main()
{
    int a, b;
    float sum;
    // Check the results
    int refOut[5] = {12, 23, 34, 45, 56};
    int pass;
    int i;

    a = 10;
    b = 20;

    // Call the function for 4 operations
    for (i=0; i<4; i++)
    {
        sum = eq(a, b);
        fprintf(stdout, "  %d+%d/10=%f \n", a, b, sum);
        a=a+10;
```

```
    // Test the output against expected results
        if (sum == refOut[i])
            pass = 1;
        else
            pass = 0;

        // Test for change in b
        sum = eq(a, b);
        fprintf(stdout, "  %d+%d/10=%f \n", a, b, sum);
        b=b+10;
    }

    // Final test
    sum = eq(a, b);
    fprintf(stdout, "  %d+%d/10=%f \n", a, b, sum);
    a=a+10;

    if (pass)
    {
        fprintf(stdout, "----------Pass!-----------\n");
        return 0;
    }
    else
    {
        fprintf(stderr, "----------Fail!-----------\n");
        return 1;
    }
}
```

Below is the result after building the program.

eq.c | eq.h | test.c | eq_csim.log

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling(apcc) ../../../../C_code/test.c in debug mode
INFO: [HLS 200-10] Running 'D:/Xilinx/Vivado/2019.2/bin/unwrapped/win64.o/apcc.exe'
INFO: [HLS 200-10] For user 'God' on host 'rajath-pc' (Windows NT_amd64 version 6.2) on Thu Dec
INFO: [HLS 200-10] In directory 'C:/Users/rajath/Downloads/esa/c_to_rtl/sol/csim/build'
INFO: [APCC 202-3] Tmp directory is apcc_db
INFO: [APCC 202-1] APCC is done.
   Compiling(apcc) ../../../../C_code/eq.c in debug mode
INFO: [HLS 200-10] Running 'D:/Xilinx/Vivado/2019.2/bin/unwrapped/win64.o/apcc.exe'
INFO: [HLS 200-10] For user 'God' on host 'rajath-pc' (Windows NT_amd64 version 6.2) on Thu Dec
INFO: [HLS 200-10] In directory 'C:/Users/rajath/Downloads/esa/c_to_rtl/sol/csim/build'
INFO: [APCC 202-3] Tmp directory is apcc_db
INFO: [APCC 202-1] APCC is done.
   Generating csim.exe
  10+20/10=12.000000
  20+20/10=22.000000
  20+30/10=23.000000
  30+30/10=33.000000
  30+40/10=34.000000
  40+40/10=44.000000
  40+50/10=45.000000
  50+50/10=55.000000
  50+60/10=56.000000
----------Pass!-----------
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************

```

The RTL program:
Once the program is built. We run the synthesis with our defined pragma's. The RTL synthesis generates a Verilog/VHDL program.

Note: Since the Verilog file is huge. The link to the file can be found **here**.

Below is the result of the Synthesis.

Synthesis Report for 'eq'

General Information

Date:	Wed Dec 15 03:29:08 2021
Version:	2019.2 (Build 2704478 on Wed Nov 06 22:10:23 MST 2019)
Project:	c_to_rtl
Solution:	sol
Product family:	virtex7
Target device:	xc7vx485t-ffg1157-1

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.630 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1	1	10.000 ns	10.000 ns	1	1	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	4	0	236	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	15	-
Register	-	-	96	-	-
Total	0	4	96	251	0
Available	2060	2800	607200	303600	0
Utilization (%)	0	~0	~0	~0	0

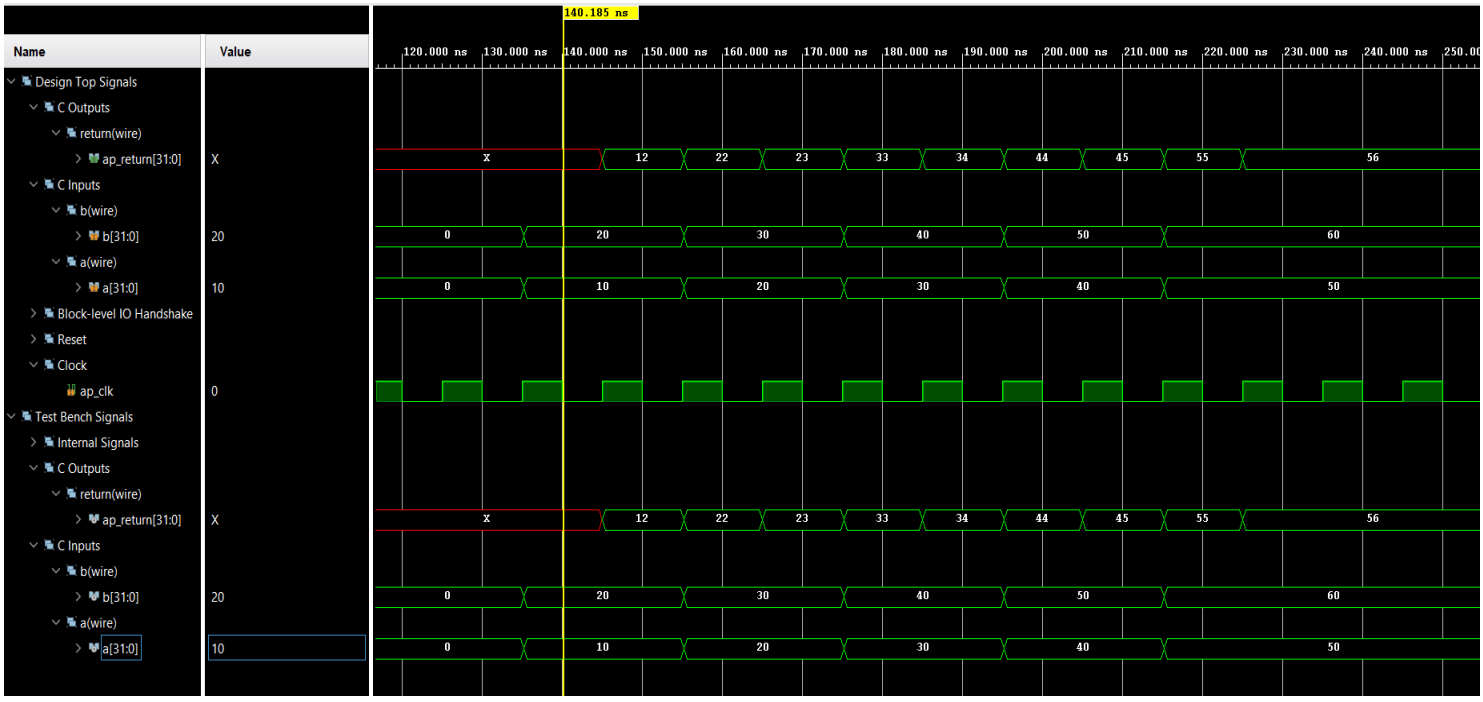
Output:

We finally run the RTL/C co-simulation to test the equivalence of both circuits.

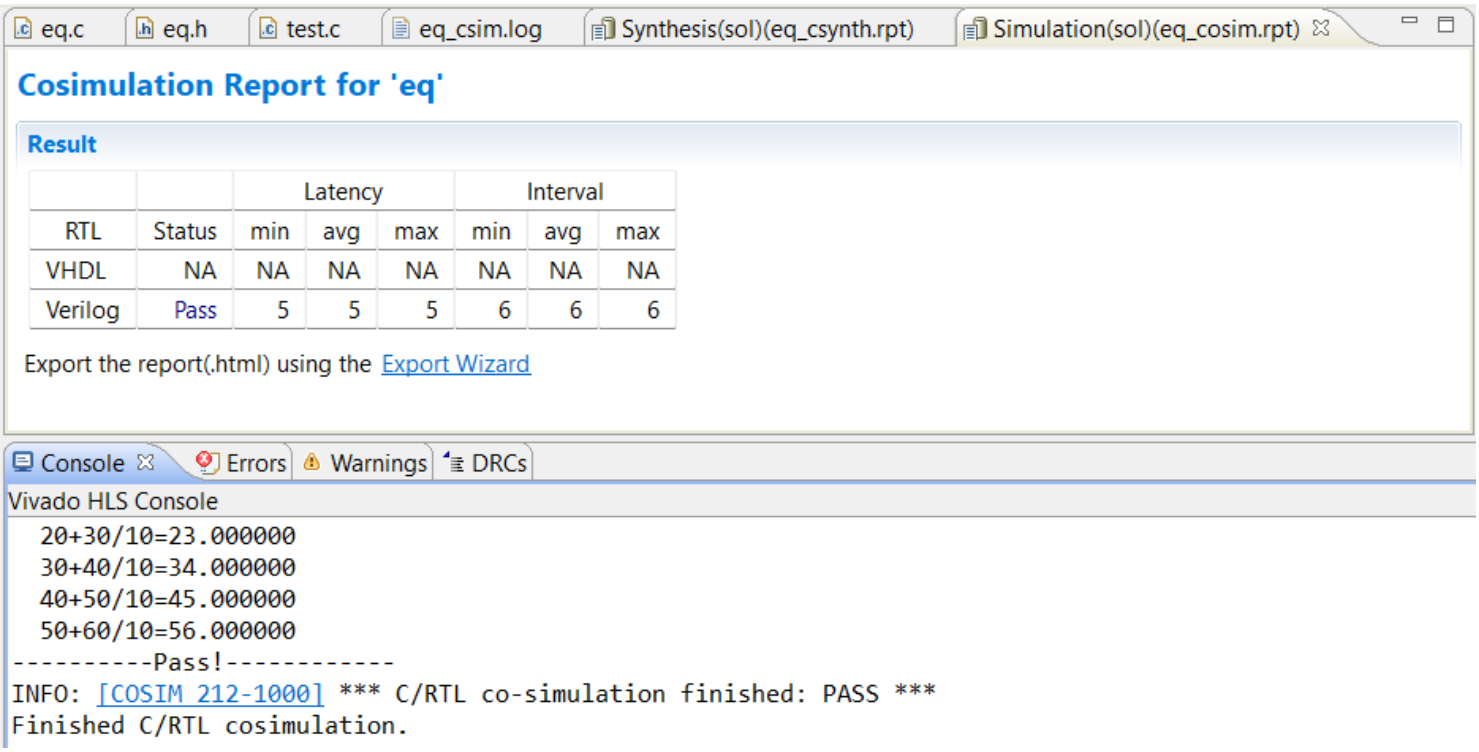
The output of the C test bench.

```
PS C:\Users\rajath\Downloads\esa\C code> ./test.exe
  10+20/10=12
  20+20/10=22
  20+30/10=23
  30+30/10=33
  30+40/10=34
  40+40/10=44
  40+50/10=45
  50+50/10=55
  50+60/10=56
----------Pass!------------
```

Output of the RTL testbench.



The above pictures show the output of the C code and the output waveform generated from Vivado HLS. Vivado generates the RTL test bench from the test bench written in C during RTL/C co-simulation. It then compares the output of the C program and RTL program. This generates the success rate and outputs a PASS or FAIL.



Result:

The equivalence between RTL and C was found for the operation “a+b/10” both by the method of manual verification using Trace-based equivalence checking and by the RTL/C-based co-simulation using Vivado HLS.

The repository containing our files can be found **here**.

References:

1. Vivado HLS https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf
2. C-Based VLSI design: Lec 35: Equivalence between C and RTL: Prof. Chandan Karfa
3. C-Based VLSI design: Lec 22: Datapath and controller synthesis in HLS: Prof. Chandan Karfa
4. C-Based VLSI design: Lec 20: Register allocation and binding: Prof. Chandan Karfa
5. C-Based VLSI design: Lec 16: Resource allocation and binding: Prof. Chandan Karfa

Individual Contribution:

Rajath J: Tool Based Verification and all its subparts, Introduction for Report, Statements, and Theory for all the graphs, and Explanation for the equivalence calculation.

Bharath V: Manual Verification, Generating and Imaging the graphs for C FSMD and RTL dataflow, control dataflow, datapath, controller FSM and FSMD, Report typing and editing.

****