Final Report

Integrated Systems Architecture

Gabriele Borello *s252845*,
Gianmarco Canzonieri *s244123*,
Giulia Colonna *s253145*,
Loris D'Amico *s252095*.

Master Course in Electronic Engineering
Electronic Systems
June, 2019

Contents

1 Design and implementation of a digital filter **3**
- 1.1 Filter design and coefficient quantization using Matlab . . . 3
- 1.2 Second step: fixed-point C model . . . 4
  - 1.2.1 Difference between the two models . . . 5
- 1.3 VLSI Implementation . . . 6
- 1.4 Advanced architecture development . . . 7
- 1.5 Implementation . . . 13
  - 1.5.1 Logic synthesis . . . 13
  - 1.5.2 Place and Route . . . 15

2 Digital Arithmetic and logic synthesizer **17**
- 2.1 Introduction & Background . . . 17
- 2.2 Optimization with Compile Ultra & Retiming . . . 18
- 2.3 Design a Modified-Boot-Encoding (MBE) based multiplier - Version 1 . . . 19
  - 2.3.1 Area and timing report version 1 . . . 25
- 2.4 Removing the last 6 LSB - Version2 . . . 25
  - 2.4.1 Area and timing report version 2 . . . 26
- 2.5 MBE-Dadda multiplier- Version3 . . . 26

3 Special Project - RISC-V, a designer step-to-step implementation's guide. **30**
- 3.1 Assumptions utilised & Specs . . . 31
- 3.2 RV32I Instruction Set Version 2.0 . . . 31
  - 3.2.1 R-Type . . . 33
  - 3.2.2 I-Type . . . 34
  - 3.2.3 S-Type . . . 38
  - 3.2.4 B-Type . . . 38
  - 3.2.5 U-Type . . . 39
  - 3.2.6 J-Type . . . 40
- 3.3 Basic RISC-V Architecture - Data Path & Control. . . . 41
  - 3.3.1 Elements needed for the Data Path . . . 42
  - 3.3.2 Control . . . 44
  - 3.3.3 Pipelined Control . . . 46
- 3.4 Description and possible solution implementation of various RISC-V problems . . . 49
  - 3.4.1 Branch prediction unit . . . 51
  - 3.4.2 Possible implementations dynamic prediction . . . 51
  - 3.4.3 Forwarding unit . . . 56
  - 3.4.4 Scoreboard unit . . . 58
- 3.5 RISC-V compiler . . . 60
  - 3.5.1 Prerequisites . . . 60
  - 3.5.2 Eclipse cross compiler . . . 60
  - 3.5.3 Online cross compiler . . . 61
  - 3.5.4 Example 1: Sum . . . 61
  - 3.5.5 Example 3: Counter . . . 65

3.5.6 C-code . . . 65
3.5.7 Architecture Simulation trial & references . . . 67
3.6 Sithography & Bibliography - Special Project . . . 68

4 Appendix - Scripts **69**
4.1 Lab 1: design and implementation of a digital filter . . . 69
4.1.1 Filter design and coefficient quantization using Matlab/Octave . . . 69
4.1.2 First step: Matlab/Octave pseudo-fixed-point . . . 69
4.1.3 Second step: fixed-point C model . . . 70
4.1.4 Starting architecture development |& simulation . . . 72
4.1.5 Logic synthesis . . . 82
4.1.6 Advanced architecture development - L-Unfolding and piplined FIR . . . 84
4.2 Lab 2: Digital arithmetic . . . 95
4.3 Ultra Optimizzation Script . . . 95
4.3.1 Version 1 . . . 96
4.3.2 Version 2 . . . 115
4.3.3 Version 3 . . . 118

Chapter 1

Design and implementation of a digital filter

The purpose of this module is to implement a digital filter with different models(developed in different programming languages via software) and then with an HDL description, in order to compare the results.
In particular the requirements are:

- Cut frequency at 2 kHz;
- Sampling frequency at 10 kHz;
- Finite Impulse Response filter;
- Order of the filter N=10;
- Number of bits n=11.

1.1 Filter design and coefficient quantization using Matlab

Matlab is used as first approach to implement the filter, we used the two scripts provided on the website.
The first one (*myfir_design.m*) implements the response of a FIR filter with the help of Matlab's internal function *fir1*.
This function takes as input the order of the filter *N* and the number of bits we choose in order to represent the results *nb* and returns nb-bit integer coefficients *bi* (coefficients computed in floating point rounded with the *floor* function to the nearest less than or equal to the element) and the associate effective values *bq*.
The function also plots the frequency response versus the angular frequency for both the filter and the discrete filter (red dashed line).
The second one (*myfir_filter.m*) generate two sinusoidal waves of frequency respectively at 550 and 4500 Hz. The sum of this waves is then sampled and saved as *samples.txt*. The samples are also used as input of the implemented Matlab FIR filter.Then, its output is saved as *resultsm.txt* with the same procedure.
For sake of clarity, numerical values are not reported, but Fig: 1.1 represents the time relations between input and output, while Fig: 1.2 is the plotted filter response versus the frequency.

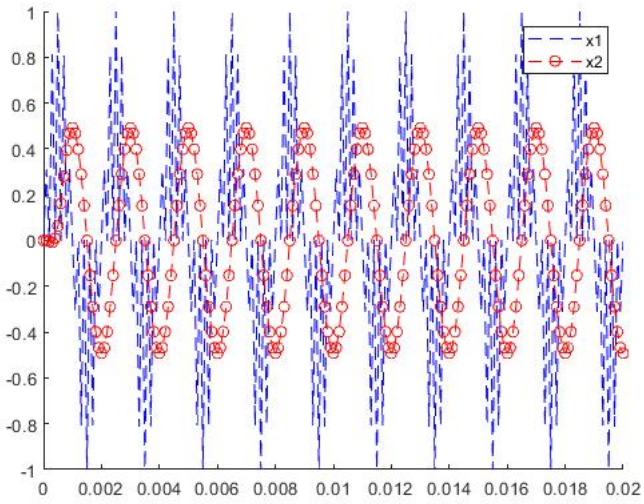


Figure 1.1: Waves generated by Matlab:x1 and x2 are respectively the total input signal and the wave out of the filter.

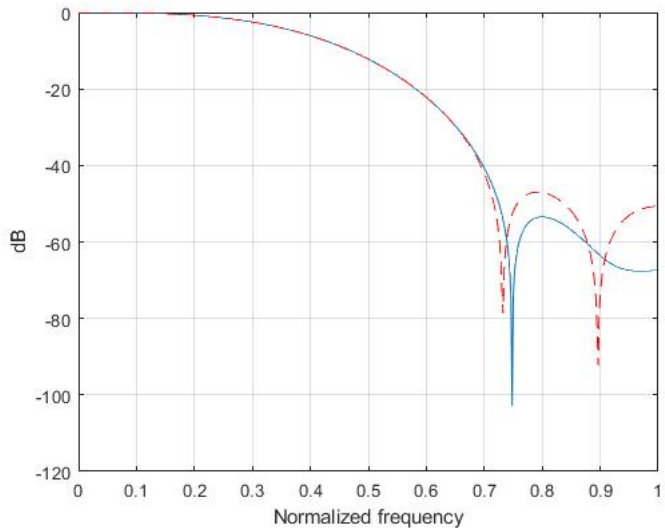


Figure 1.2: Matlab's implemented filter response. In blue frequency vs. the angular frequency, while in red the sampled response.

1.2 Second step: fixed-point C model

While Matlab uses floating point, we must create a fixed point model, in order to complete this task this a C model has been implemented with an array of NT=11 elements named *b[NT]*, in which are saved the filter coefficients computed with *Matlab*. The purpose is to use them, together with *samples.txt* in order to obtain the response of the filter given by

$$y_i = \sum_{j=1}^{N} T x_i = j b_j \tag{1.1}$$

The model is implemented as follows:

1. Definition of NB=11 (number of bit) and memorisation of the 11 coefficients of the filter;

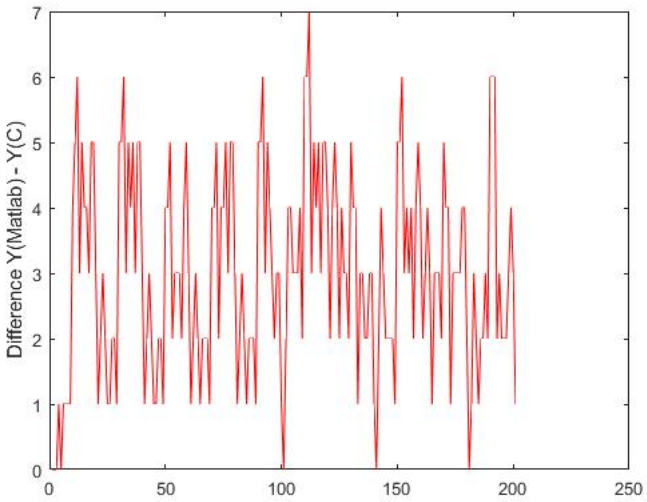


Figure 1.3: Difference in values of Y(floating point) - Y(fixed point) out of the FIR filter when the coefficients are represented with 11 bits.

2. Function *myfilter* that cleans internal buffers, saves outside given samples x into internal shift registers and make the convolution of the moving average part as shown earlier updating the shift register of the output y;

3. The main part of the program where the file *samples.txt* is opened, processed by the function *myfilter* and the outputs saved in *resultsc.txt*.

1.2.1 Difference between the two models

Once again the numeric values are not shown. The comparison it has been made with the help of Matlab. The following code has the purpose of instanciate two variables (*resm* and *resc*), take every value from *resultsm.txt* and *resultsc.txt*, plot the difference between the two and compute the average difference.

```
resm=0;
resc=0;

rm=fopen('resultsm.txt','r');
resm=fscanf(rm,'%d\n');
fclose(rm);

rc=fopen('resultsc.txt','r');
resc=fscanf(rc,'%d\n');
fclose(rc);

delta=resm-resc;


average = mean (delta);
plot(delta,'r');
```

The result is shown by Fig: 1.3 since it is not easy to see any difference from the sinusoidal waveforms. The average difference between the two is 2.995.

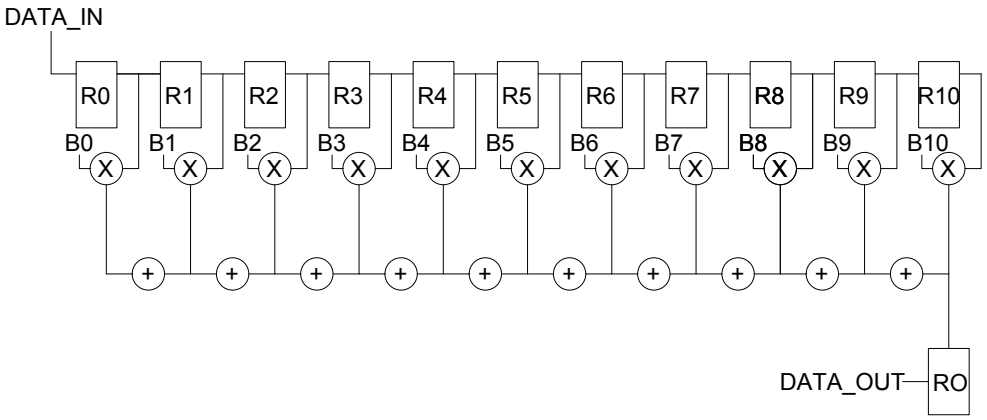


Figure 1.4: FIR filter implemented data path.

1.3 VLSI Implementation

The fir filter is then implemented using VHDL and Verilog language only for a structural description of the higher entity, the architecture for the data path is shown in Fig: 1.4.

The solution consists in 11 registers (since the order of the filter is N=10), 11 multipliers (each one of them with one of the two inputs settled to a constant value representative of the coefficients *b[NT]*) and 10 adders.

The description used for the components of the path is with type *SIGNED*, available in *ieee.numeric_std* package,in order to get *C2* representation correctly.

The control unit is synchronous and involved only 3 states which are *idle, elaborate* and *done* : from *idle* if the entity receives *reset* signal, then next state is *elaborate*.

Since the path for the samples is purely combinational , when the registers have a valid outputs the computation is complete in one clock cycle.

The data path is implemented with an 11 bits parallelism since the maximum positive number achievable is 2047 and the negative one is -2048. We can do a simple calculation for the worst case computation, assuming that the coefficients stay the same and the sample's values are all the positive maxiumum or the negative minimum as $M = X(n-i)$:

$$Y(n) = -1M - 13M - 26M + 65M + 281M + 407M + 281M + 65M - 26M - 13M - 1M \quad (1.2)$$

In both of the cases, we obtain a result which can be represented on 11 bits, so there is no reason to implement a wider net.
The only exception is for the multiplier which internally computes the operation on 22 bits (*11 bits x 11 bits*) and then takes on the output only 10 bits for representing the integer part (estimated on the results given by Matlab) and 1 bit for the fractional part, we consider in the connection with the rest of the data path only bits

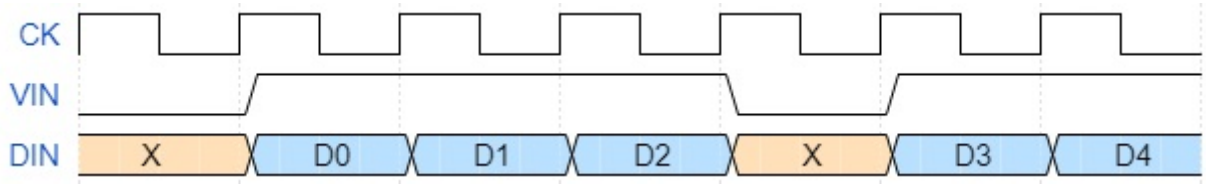


Figure 1.5: Expected behaviour of the filter during sampling time.

(*20 DOWNTO 10*) as shown in the code below:

```
ARCHITECTURE behavior OF mpy_fixedpoint IS

SIGNAL prod_temp: SIGNED (21 DOWNTO 0);
BEGIN
prod_temp<=SIGNED(SAMPLE_IN)*SIGNED(Bi);

PRODUCT<=STD_LOGIC_VECTOR(prod_temp(20 downto 10));

END ARCHITECTURE;
```

Other than this, no further approximation is performed, and the simulation gives back these results, coherent with those obtained in C language:

```
0
-1
-5
-10
9
63
164
286
402
474
499
473
404
291
151
-5
..
```

The expected behaviour during the simulation according to given constrains of the lab assignment on the signal *V_in*, in order to perform a correct sampling in the chain of registers, is shown in Fig: 1.5.

In order to utilise the test benches given as assignment's material, we implemented ah high level description with all the ports of the system as *STD_LOGIC_VECTOR* to allow the correct connection with the *verilog* description.

The systems behave has expected as shown in Fig: 1.6, where we reported the correspondent start and end of the simulation @ *2500 ns* for a default clock period of *10 ns*.

1.4 Advanced architecture development

At this point we want optimize the circuit structure in terms of throughput and timing characteristics. For this aim 2 technique are used : L-unfolding and pipelining. After that we analyze the results and we will do the

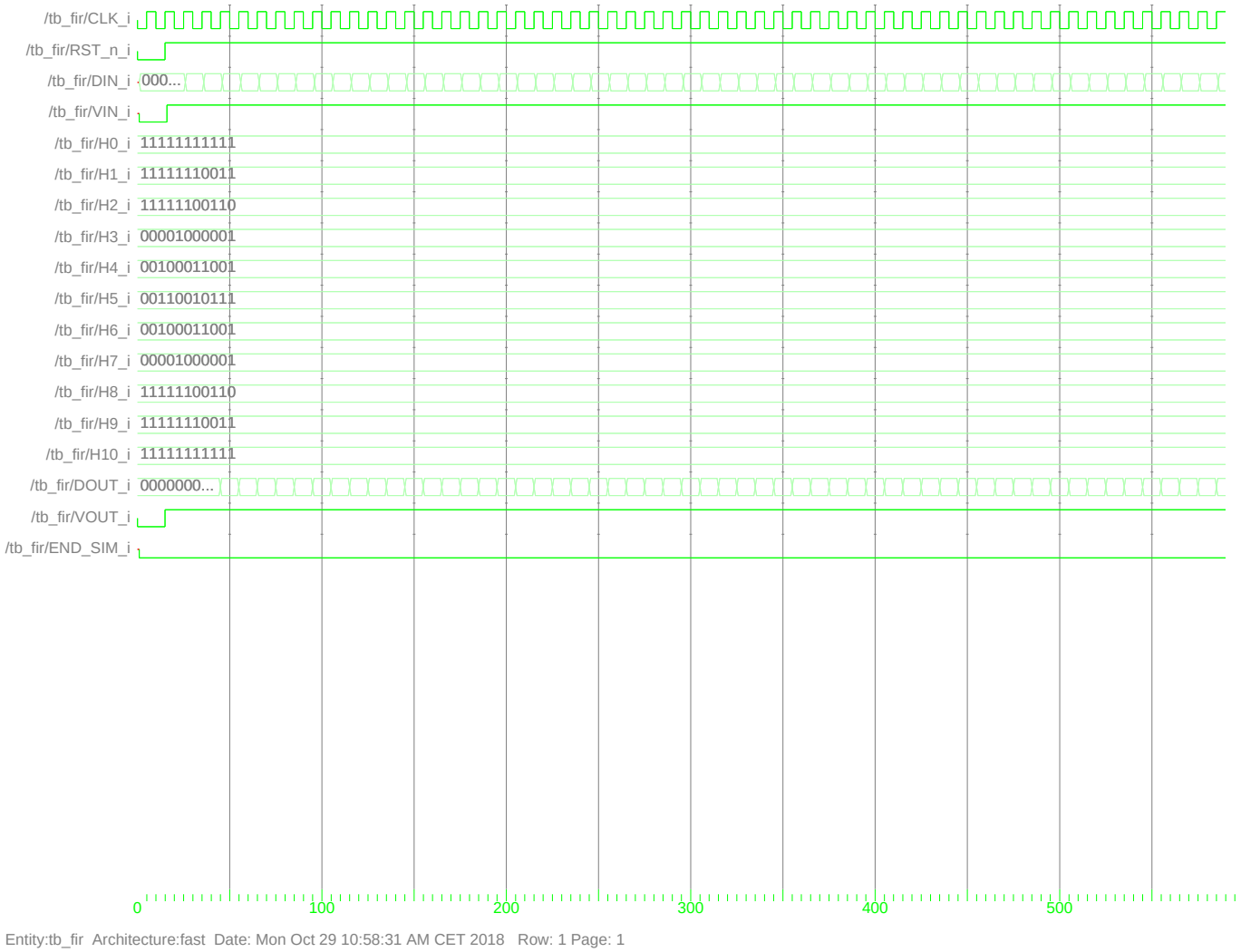


Figure 1.6: Start of simulation.

comparison between the basic and advanced structure in terms of delay, latency , area and power consumption. We start with 3-unfolding.

3-unfolding is used for parallelization of the inputs so we obtain the 3 outputs from 3 inputs simultaneously (in the same clock cycle). In this way we expect to increase the throughput of the circuit of 3 times about.
The topology of the circuit is a bit different than basic configuration because parallel work means use triple number of registers like using 3 different filters but they are depend one from each other (Fig: 1.7). Circuit parallelism is the same but we increase the area (cost is higher) because we have more registers. The consequence of a higher number of registers is the switching activity increment due to new registers commutations. Power consumption follows the switching activity linearly so we expect it also will increase (frequency is maintained constant). Contrary, execution time is reduced (because it is faster) so the total used energy is the same.

Pipe is used because in the circuit is present a very long critical path that limits the possible max frequency. We choose 1 level of pipe after the step of multiplication (because it is a critical step for delay (matrix of adder) in this way we divide the critical path of circuit composed of 1 multiplier and 11 full adders in 2 parts that have similar delay (1 multiplier of 11 bits and 11 full adders of 11 bits). Pipe permits to reduce the critical path. Pipe registers are always able.
Area now will increase in a drastic way because with 2 techniques we have 6 times the number of register respect of basic configuration. The power consumption will increase for the same reason of before but the energy is more or less constant.

During advanced structure implementation we choose an hierarchic and modular approach. At VHDL level the new filter is similar to the first, the control unit is the same , we have changed just the datapath (now called datapathPipeUnfolding) and the files used for simulation because machine timing now is different.

With the following VHDL fragment we want show how at VHDL level we replace the datapath with a new version and we leave unchanged the control unit.

```
ARCHITECTURE behavior OF control_unit IS

COMPONENT datapathPipeUnfolding IS
PORT( CLK,RST_n:                  IN STD_LOGIC;
          V_IN :                        IN STD_LOGIC;
          DATA_IN0:                  IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN1:                  IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN2:                  IN STD_LOGIC_VECTOR (10 DOWNTO 0);
      B0,B1,B2,B3,B4,B5:      IN STD_LOGIC_VECTOR (10 DOWNTO 0);
      B6,B7,B8,B9,B10:        IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT0:                 OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT1:                 OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT2:                 OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT:                        IN STD_LOGIC
        );
END COMPONENT;
```

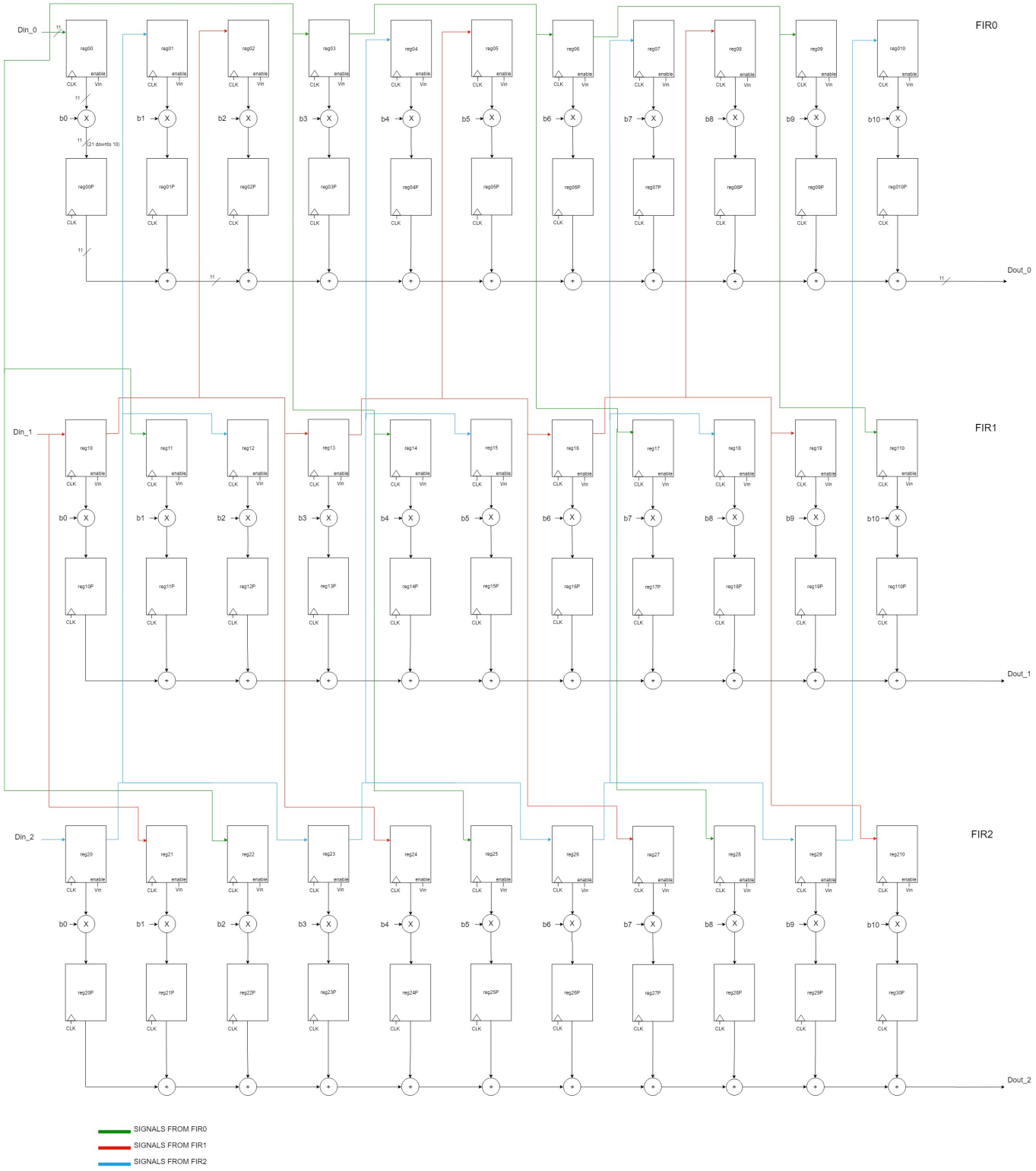


Figure 1.7: DATAPATH FIR 3-UNFOLDING AND PIPELINING

This is the expected timing analysis with Pipe and 3-unfolding

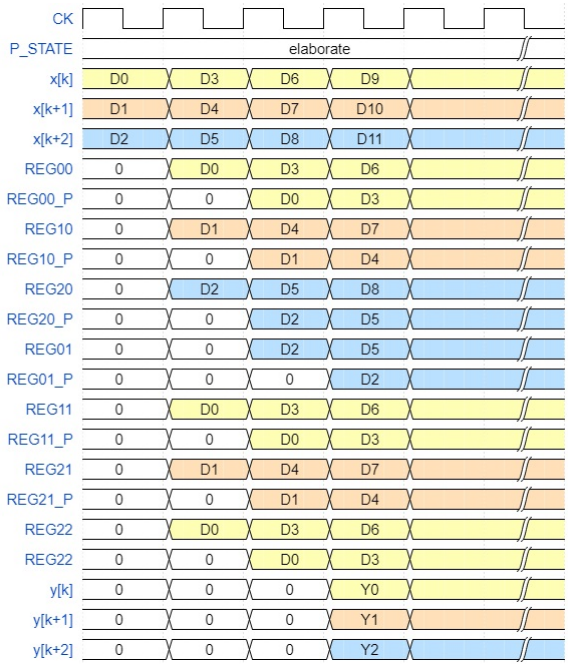


Figure 1.8: 3-unfolding and pipelining timing

First Simulation is provided for check if the circuit works correctly and if our previsions were correct. We use a clock period of 10 ns and simulation time of 2500 ns as the first basic circuit simulation:

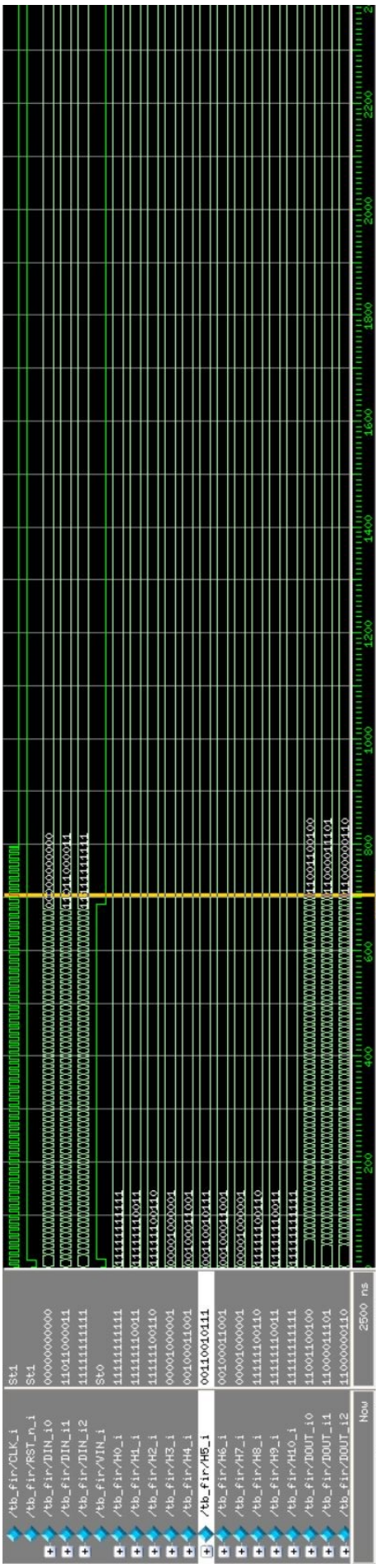


Figure 1.9: Modelsim simulation with 2500 ns of simulation time and 10 ns of clock period

1.5 Implementation

1.5.1 Logic synthesis

In order to perform the logic synthesis, Synopsys Design Compiler has to be initialyzed with the terminal's instructions *source /software/scripts/init_synopsys* and *design_vision*.
First of all an internal folder *work* is created and we move into *syn* directory the file *.synopsys_dc.setup* that will be an hidden file (because the first character is ".") and will have the purpose of defining the path to the directories that will be used. In our project the library used is **DesignWare library**.
Once Synopsys is open we lunch a script of the commands with *source test_script.txt*. The following file is reported in the Appendix. It is important to point out that the file is used and modified three times. The file has the purpose of:

- Analyse the vhdl files strarting from the lower level, and preserve their names;
- Elaborate the top entity of our filter *control_unit* of architecture *behavior*;
- Create an untouchable signal *MY_CLK* of period 0.0 ns with 0.07 ns uncertanty;
- Set the input and output maximum signal delay of 0.5 ns;
- Set the load of each output as *BUF_X4* that is described in *NangateOpenCellLibrary*;
- Compile and save area and timing reports.

The informations in output files, called *report_timing_fck0* and *report_area_fck0*, are shown in Tab:1.1. It has to be pointed out that Pipelined and Unrolled implementation has more than three times the area of the directly implemented version, as expected.

	Directly implemented FIR	Pipelined and Unrolled FIR
Number of ports:	147	191
Number of nets:	155	196
Number of cells:	8	6
Number of references:	6	5
Combinational area:	7462.363990	22708.951956
Noncombinational area:	717.136024	4048.520131
data required time	-0.10	-0.11
data arrival time	-3.10	-2.08
Total cell area:	8179.500000	26757.472656
slack (VIOLATED)	-3.22 $\rightarrow$ -3.72	-2.19 $\rightarrow$ -2.39

Table 1.1: Area and timing report with clock period of 0.0 ns. The first slack value is obtained with clk = 0 ns, while the second is the actual value linked to the maximum frequency.

The script is modified accordingly in the third point when the clock period is assigned to 3.72 and 14.88 ns; in the fifth point where the reports are generated (with different names) and a final part has been added at the end of the script that has the purpose of:

- Flatten the hierarchy;
- Describe the delay of the netlist (.sdf);
- Export the netlist in verilog (.v);
- Export the constraints to the input and output ports (.sdc);

From Tab: 1.2 it can be seen that with higher frequency, meaning lower constraints, synopsys is able to syntetize the filter with lower area meaning lower money cost.

	DIRECTLY IMPLEMENTED FIR		PIPELINED and UNFOLDED FIR	
	Tck = 3.72 ns	Tck = 14.88 ns	Tck = 2.39 ns	Tck = 9.56 ns
Number of ports:	147	147	191	191
Number of nets:	153	153	197	197
Number of cells:	7	7	7	7
Number of references:	6	6	6	6
Combinational area:	6170.136016	6037.402020	20119.973992	18798.486055
Noncombinational area:	712.880023	712.880023	4060.756130	4048.520131
data required time	3.72	14.87	2.39	9.55
data arrival time	-3.71	-4.13	-2.39	-2.80
Total cell area:	6883.880023	6750.282227	24180.730469	22847.005859
slack	0.00	10.74	0.00	6.76

Table 1.2: Area and timing report with clock period equals to f_{max} and $f_{max}/4$ in both implementations.

Switching-Activity-based power consumption estimation

The procedure to extimate the switching-activity-based power consumption needs the use of both *Modelsim* and *Synopsys*. Firstly synopsys is lunched, and saif's file of the technological libraries *NandgateOpenCellLibrary* are prepared. Then the test bench has been modified including statements to get the switching activity. A new folder, called *tb_pw* has been created, and it includes the old verilog's test bench with in addition (see the complete file in the appendix).

```
initial begin
$read_lib_saif("../saif/NangateOpenCellLibrary.saif");
$set_gate_level_monitoring("on");
$set_toggle_region(CU);
$toggle_start;
end

always @ ( END_SIM_i ) begin
if (END_SIM_i) begin
$toggle_stop;
$toggle_report("../saif/myfir_back.saif", 1.0e-9, "tb_fir.CU");
end
end
```

That loads the library saif file generated and monitors our top entity CU until the simulation stops, meaning until the signal *END_SIM_i* is equal to '1'.

Once again two scripts are used. Few part of them are changed accondingly to time characteristics of the clock signal in the simulations. Once we lunch *Modelsim* we just type *source power_script_modelsim* to run the first part. This script has the purpose of:

- Compile the three vhdl test benches;
- Compile the test bench and the verilog file representing our filter syntetized by synopsys
- Include the functional model of the cells in the technology library compiling the verilog source and link the compiled library. We recall that the syntax is *vsim* <*library path*>.<*design*> and that *-sdftyp* is used to annotate verilog's cells in the specified SDF file with a given timing while *-pli* links to an external library provided by *Synopsys*. The result will be saved as *myfir_back.saif* in the saif directory.

Then, using *Synopsys* in *syn* folder we just read *myfir_back.saif* in the saif directory and *myfir_14_88* (again this part is switched accordingly to the clock period) in the *netlist*. The report power saved at the end of the scripts are shown in Tab: 1.3 where Tab: 1.4 express the units used.

	DIRECTLY IMPLEMENTED FIR		PIPELINED and UNFOLDED FIR	
	Tck = 3.72 ns	Tck = 14.88 ns	Tck = 2.39 ns	Tck = 9.56 ns
Cell Internal Power	513.0823 uW (55%)	507.8305 uW (55%)	1.7888 mW (58%)	1.7523 mW (60%)
Net Switching Power	425.4335 uW (45%)	416.2430 uW (45%)	1.2828 mW (42%)	1.1927 mW (40%)
Total Dynamic Power	938.5157 uW	924.0735 uW	3.0716 mW	2.9450 mW
Cell Leakage Power	138.6446 uW	134.0616 uW	503.2278 uW	454.9011 uW

Table 1.3: Power consumption estimation with *Synopsys design compiler*.

Voltage Units =	1V
Capacitance Units =	1.000000ff
Time Units =	1ns
Dynamic Power Units =	1uW
Leakage Power Units =	1nW

Table 1.4: Power consumption units.

Moved by curiosity it has been decided to estimate power consumption and area working with the same clock period. A clock period of 10 ns has been decided and syntetized as in precedence. The results obtained, shown in Tab: 1.5, evidence that the Pipelined and Unfolded structure uses three time the power of the directly implemented one, it has an higher slack (we finish the operations earlier) and more than three time the area (meaning more % power will be used by cells).

	DIRECTLY IMPLEMENTED FIR	PIPELINED and UNFOLDED FIR
Total cell area:	6750.282227	22847.005859
data required time	9.89	9.89
data arrival time	-4.03	-2.70
slack (MET)	5.86	7.20
Cell Internal Power	528.5338 uW (53%)	1.7610 mW (59%)
Net Switching Power	477.8873 uW (47%)	1.1987 mW (41%)
Total Dynamic Power	1.0064 mW (100%)	2.9596 mW (100%)

Table 1.5: Power consumption estimation with *Synopsys design compiler* with clock period of 10 ns.

1.5.2 Place and Route

Since *Synopsis* only estimates power, area and timing, for both the implementation of the filter *innovus* had been used to verify them after placing and routing each cell. In particular after lunching the simulator the following steps have been used:

- Imported various library, the test bench and *Synopsis'* flatted implementation of our Fir filter;
- Structured floor plan with core aspect ratio 1.0 and utilization 0.6, core to die boundary 5 μm from each side;
- Inserted V_{dd} and V_{ss} power rings with width and spacing 0.8 μm;

- Standard power routing to connect both V_{dd} and V_{ss}, both recognised as special route;
- Place all the blocks of our filter with maximum 8 layers;
- Optimise post Clock-Tree-Syntesis (CTS) to achieve the required timing constraints;
- Place all possible fillers of the library uploaded;
- Nanoroute and optimisation;
- Extraction of RC parasitics;
- Timing analysis post route (shown in Tab: 1.6);
- Verification of connectivity and geometry;
- Generation of area report (shown in Tab: 1.7), netlist and SDF file for timing constraints.

DIRECTLY IMPLEMENTED FIR	PIPELINED and UNFOLDED FIR
(over 14.88 ns clk period)	(over 9.58 ns clock period)
11.447 (dp_reg_data_out_temp_reg_10_/D1)	7.213 (dp_reg14P_temp_reg_10_/D1)

Table 1.6: Minimum slack value post route in the two different implementations.

	DIRECTLY IMPLEMENTED FIR	PIPELINED and UNFOLDED FIR
Gate area [μm^2] =	0.7980	0.7980
Gates =	8442	27819
Cells =	3334	10464
Area [μm^2] =	6736.7	22199.8

Table 1.7: Area report after place and route.

Chapter 2

Digital Arithmetic and logic synthesizer

2.1 Introduction & Background

To evaluate different designs of adders and multiplier, ready-to-use blocks have been used. This is possible due to the avability of *Design Ware*, where many possible implementations are given.
Implementation examples for adders are:

- Ripple carry (rpl);
- Carry look-ahead (cla);
- Parallel-Prefix (pparch);

Implementation examples for multipliers are:

- Carry-save (csa);
- Parallel-Prefix (pparch);

After analysing the structure we expect that when comparing the adders, Carry look-ahead is the biggest one in term of area and also the slower one (lower slack) while the Ripple carry and Parallel-Prefix are very similar; thus comparing the two multiplier the Parallel-Prefix implementation leads to a significant reduction in area and a small increment in slack.

Firstly we used the following script on *Synopsis* to have an estimation of area and timing slack with different implementations. A script is used just to speed up the evaluation. Also a report for resources used is generated, and it is used to verify the correct implementation. Moreover, this third report is not shown. It must be notice that all the multipliers and adders are implemented with a given structure.

```
analyze -f vhdl -lib WORK ../src/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/mpy_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/reg_11bit.vhd
analyze -f vhdl -lib WORK ../src/datapath.vhd
analyze -f vhdl -lib WORK ../src/control_unit.vhd
set power_preserve_rtl_hier_names true

elaborate control_unit -arch behavior -lib WORK
create_clock -name MY_CLK -period 14.88 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set load $OLOAD [all_outputs]
```

```
ungroup -all -flatten
set_implementation DW01_add/rpl [find cell *add_*]
set_implementation DW02_mult/csa [find cell *mpy*]

compile
report_timing > report_timing_rpl_csa
report_area > report_area_rpl_csa
report_resources > report_resources_rpl_csa
quit
```

This let to the following tables, that shows what we expected.
For brevity the resource reports are not shown but they were used to check that *Synopsis* implemented the wanted blocks.

TOTAL CELL AREA	rpl	Cla	pparch
csa	8017.240234	8147.580078	8006.600098
pparch	6647.872070	6778.211914	6637.231934

Table 2.1: Area report with different adders, shown as different columns, and multipliers, shown as different rows. Note that higher area means higher cost.

SLACK (MET)	rpl	Cla	pparch
csa	10.52	10.33	10.51
pparch	10.73	10.48	10.73

Table 2.2: Timing report with different adders, shown as different columns, and multipliers, shown as different rows. Note that higher value means faster implementation.

2.2 Optimization with Compile Ultra & Retiming

As second evaluation step of digital arithmetic, the goal is to use the *Retiming* technique to obtain a pipelined multiplier without the need of knowing the internal structure. To do that the compile_ultra command has been used with different possible implementation of the FIR filter and just an additional register. The script used in the section before has been changed and its last version is the following one:

```
set_ultra_optimization true
analyze -f vhdl -lib WORK ../src_2/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src_2/mpy_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src_2/reg_11bit.vhd
analyze -f vhdl -lib WORK ../src_2/reg_22bit.vhd

analyze -f vhdl -lib WORK ../src_2/control_unit.vhd
set power_preserve_rtl_hier_names true
elaborate control_unit -arch behavior -lib WORK

create_clock -name MY_CLK -period 14.88 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
```

```
set load $OLOAD [all_outputs]
ungroup -all -flatten

compile_ultra
set_dont_touch *_in_reg
set_dont_touch *_out_reg
optimize_registers

report_timing > report_timing_ultra_regfuori_4
report_area > report_area_ultra_regfuori_4
report_resources > report_resources_ultra_regfuori_4
```

Where it is important to notice that the input and output registers of the filter cannot be moved by the retiming technique of the ultra compiler. In order to do that few modifications in the name of the input and output signals and the name of the instances of the registers inside the datapath were made. The first evaluation with the ultra compiler were made analyzing also the datapath, and the multiplier was modified to add N registers at the end of the multiplication. The following table was obtained:

Pipe level ⇒	1	2	5	6	7
Area	6562.751953	6943.132324	7118.691895	7177.211914	7235.731934
Slack	11.80	11.70	11.70	11.70	11.70

Table 2.3: Ultra compiled solution with different level of pipeline instantiated in the multipliers.

That shows an increase of the area due to the multiple pipeline register and the different internal syntetization of the filter, while it shows a decreasing slack. The advantage thus saturates with just two level of pipeline, then the timing is fixed and just an increase in area is visible. It can be seen that comparing the results with the DW one, the ultra compiler exploits more area but leads to an decrement in slack time (so a possible higher working frequency).

In the second evaluation with the ultra compiler datapath.vhd has been erased and all the pipeline registers were instantiated out of the multipliers, inside the file control_unit.vhd. The following table is thus obtained:

Pipe level ⇒	1	2	3	4
Area	971.741138	1021.971985	1075.171997	1133.692017
Slack	12.30	12.40	12.40	12.40

Table 2.4: Ultra compiled solution with different level of pipeline instantiated outside of the multipliers.

Where we can see both an increment in slack and a decrease in area. This is the best implementation up to this point, with just two pipe levels.

2.3 Design a Modified-Boot-Encoding (MBE) based multiplier - Version 1

The version 1 is divided into 2 macro-blocks; First one is used to compute the partial products through Modified booth Encoding (MBE) and the second block has the purpose to sum them through Dadda tree with Roorda approach. Now we analyze the 2 structure in separately way.

$b_{2j+1}b_{2j}b_{2j-1}$	p_j
000	0
001	a
010	a
011	2a
100	-2a
101	-a
110	-a
111	0

Figure 2.1: MBE LOGIC FUNCTION

Modified booth Encoding

It implements the first part of multiplication ; it consists to divide the multiplier (b) into 3 bit slices (in our case to achieve the correct use of algorithm we have to add a extension bit to have even bits) . Each triplet of bits is exploited to encode the multiplicand (a) according to table showed in Fig: 2.1.

Pj represents the partial products where j is the number of partial products ; in our case they are six . According to specifications we have to design the hardware-block in order to compute partial products in just one clock cycle . So we use parallel approach.: 2.2.

Recording logic : this block has 3 inputs (triplet of bits of multiplier) and 3 outputs (three control signals NEG X, TWOX, NON0X).

This block realises the function logic of table showed in : 2.1 .

Dadda tree with Roorda approach

This circuit has 6 inputs of 12 bits (partial product) and 6 inputs of 1 bit called NEGA, NEGB , NEGC , NEGD, NEGE, NEGF. The last are control bits and they are used to do the 2's complement of products in the case where needed. When MBE send NEGX = '1' the partial product X has to be complemented. It has 1 output of 22 bits that it is the result of multiplication.

```
ENTITY reorda_daddatree_sum is

PORT (

negA, negB , negC, negD, negE , negF : IN STD_LOGIC;

NEW_DATA_INA: IN STD_LOGIC_VECTOR(11 DOWNTO 0);
NEW_DATA_INB : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
NEW_DATA_INC : IN STD_LOGIC_VECTOR( 11 DOWNTO 0);
NEW_DATA_IND: IN STD_lOGIC_VECTOR (11 DOWNTO 0);
NEW_DATA_INE: IN STD_lOGIC_VECTOR (11 DOWNTO 0);
NEW_DATA_INF: IN STD_LOGIC_VECTOR(11 DOWNTO 0);
Y : OUT STD_LOGIC_VECTOR ( 21 DOWNTO 0));

END reorda_daddatree_sum ;
END COMPONENT;
```

This block provides to sum the six partial products with the right weight. Before to sum the partial products have to be correctly alignment so they are shifted one from each other of 2 positions and empty spaces are filled with '0'; after that, i have to apply the extension bits in order to arrive at 22 bits of addends. Like showing in : 2.3 now our aim is reduce the number of full adders needed for this operation thanks to Roorda method (

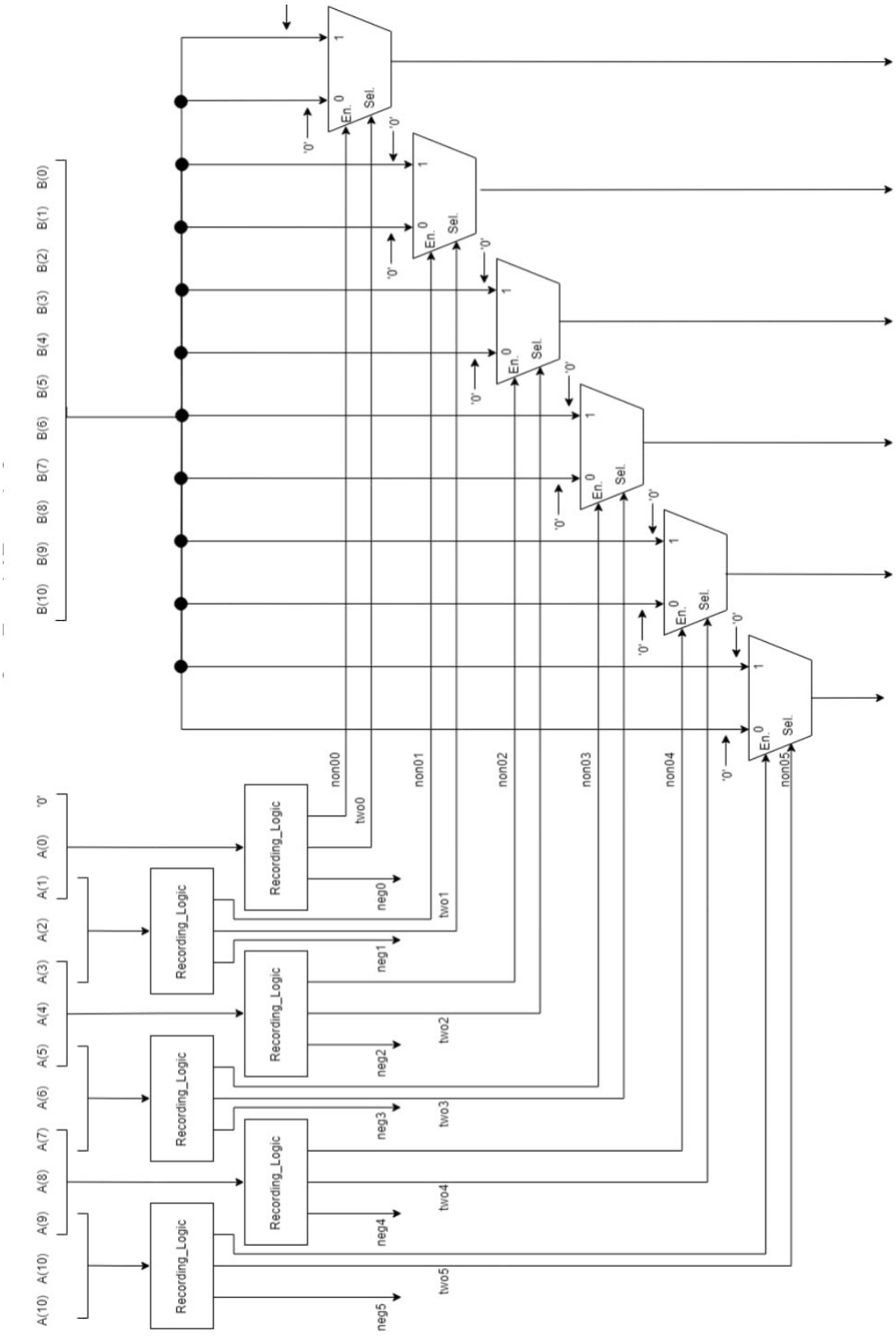


Figure 2.2: MBE DATAPATH

replacing the extension sign bits with '1' plus not extension bit). It provides simpler operations because some full adders are not used because we already know the results of this operation. Now it has to be implements the Dadda tree with 4 steps like in : 2.4 .
To describe it in VHDL we used a structural block and we treated the single bit like showing in follow code.

```
ARCHITECTURE behavior OF control_unit IS


FULL_ADDER1:fulladd port map ( DATA_INC (5),
DATA_INE(1), DATA_IND(3) , SS1 , CC1);

FULL_ADDER2:fulladd port map ( DATA_IND (4) ,
DATA_INE(2), DATA_INF(0) , SS2 , CC2);

FULL_ADDER3:fulladd port map ( DATA_INF (1) ,
DATA_INE(3), DATA_IND(5) , SS3 , CC3);

FULL_ADDER4:fulladd port map( DATA_INC (7) ,
DATA_INB(9), DATA_INA(11) , SS4 , CC4);

FULL_ADDER5 :fulladd port map(DATA_INC(8),NOTDATA_INB(10),
NOTDATA_INA(11),SS5, CC5);

FULL_ADDER6 :fulladd port map( DATA_INF (2) ,
DATA_INE(4), DATA_IND(6) , SS6 , CC6);

FULL_ADDER7 :fulladd port map( DATA_INC (9) ,
NOTDATA_INB(11), DATA_INB(10) , SS7 , CC7);

FULL_ADDER8 :fulladd port map( DATA_INF (3) ,
DATA_INE(5), DATA_IND(7) , SS8 , CC8);

FULL_ADDER9 :fulladd port map( DATA_INF (4) ,
DATA_INE(6), DATA_IND(8) , SS9 , CC9);

FULL_ADDER10 :fulladd port map( DATA_INF (6) ,
DATA_INE(8), NOTDATA_IND(10) , SS10 , CC10);

FULL_ADDER11 :fulladd port map(DATA_IND (10),
DATA_INE(9), DATA_INF(7), SS11, CC11);

        );
END COMPONENT;
```

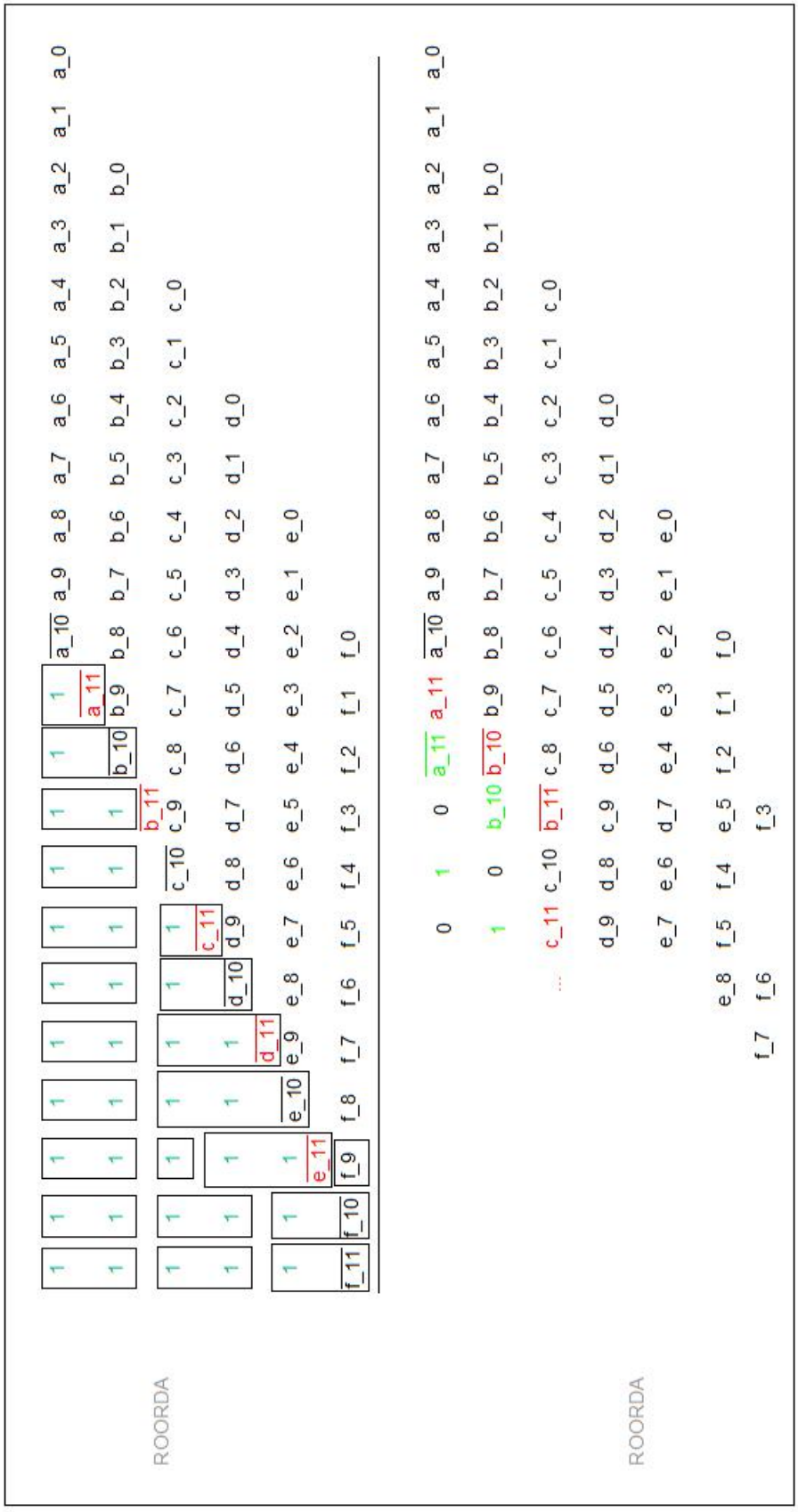


Figure 2.3: Roorda method's schematic.

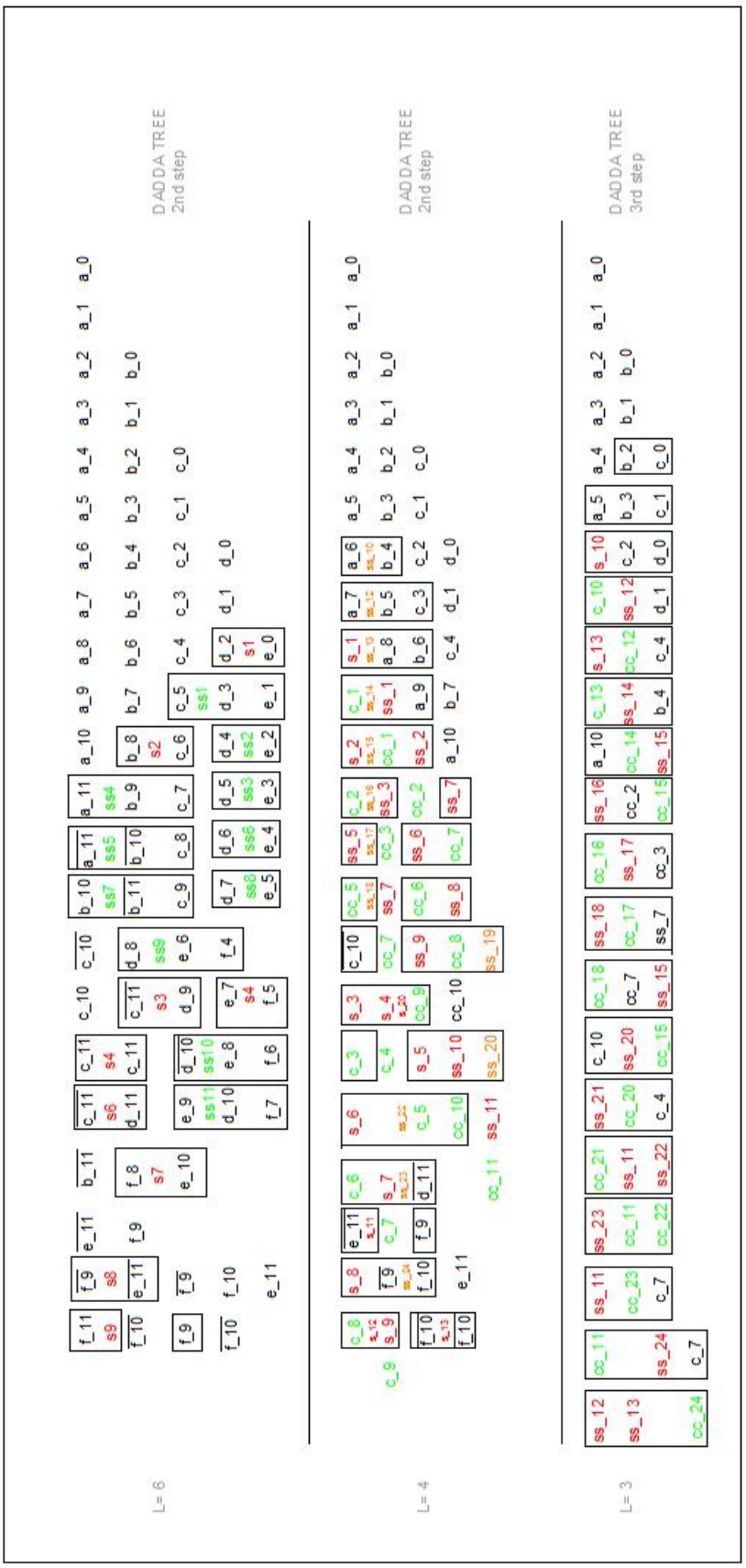


Figure 2.4: Dadda tree's schematic.

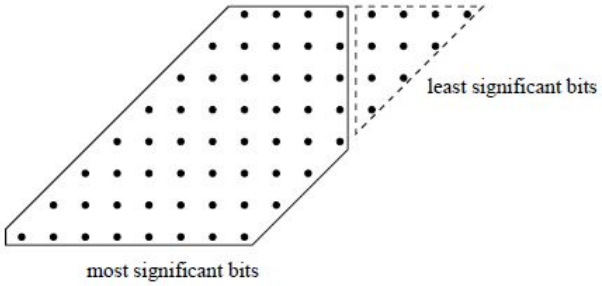


Figure 2.5: Dor Representation of Version 2.

2.3.1 Area and timing report version 1

	Tck = 0 ns	Tck = 17 ns	Tck = 63 ns
Number of ports:	147	147	147
Number of nets:	10510	7848	7734
Number of cells:	10147	7476	7362
Number of references:	57	56	50
Combinational area:	11514.075903	8938.929937	8819.973992
Noncombinational area:	874.608030	873.544029	861.308028
data required time	-0.11	2.52	3.06
data arrival time	-2.52	-3.06	-3.06
Total cell area:	12388.683594	9812.473633	9671.494141
slack	-2.63	-0.54	0.00

Table 2.5: Area and timing report with clock period equals to 0 ns, 17ns and 36 ns

2.4 Removing the last 6 LSB - Version2

We can do better by reducing the precision. We Modify our digital filter such that only the most significant part of the multiplication is used (see example in Fig : 2.5.). Then, We remove all the adders related to the 6 least significant bits. The results remain the same so it means no significant errors are introduces to system. Also we have some improvement about hardware resources because we save some full adders in the Dadda tree but it is very small (some gates).

2.4.1 Area and timing report version 2

	Tck = 0 ns	Tck = 47 ns	Tck = 96 ns
Number of ports:	147	147	147
Number of nets:	9561	7211	7055
Number of cells:	9199	6840	6684
Number of references:	54	49	46
Combinational area:	10499.285899	8299.997936	8141.461936
Noncombinational area:	859.712028	862.904028	866.096029
data required time	-0.11	2.36	2.84
data arrival time	-2.36	-2.85	-2.84
Total cell area:	11362.189453	9166.093750	9001.173828
slack	-2.47	-0.49	0.00

Table 2.6: Area and timing report with clock period equals to 0 ns, 47 ns and 96 ns

2.5 MBE-Dadda multiplier- Version3

The version 3 is divided into 2 macro-blocks; The first one is again used to compute the partial products while the second block has the purpose to sum them through Dadda tree with Roorda approach and a compressor 4 to 2. Now we analyse the 2 structure in separately way.

Partial Products generator

This first block implements the first part of multiplication. It consists in dividing the multiplier (X) into 3 bit slices. Each triplet of bits is exploited to encode the multiplicand (Y) according to the following expression:

$$PP\_\S(ç) <= (X(2\S) + X(2\S - 1))(X_(2\S + 1) \oplus Y(ç)) \tag{2.1}$$

Where:

- § is a symbol representing the 6 triplets, meaning the 6 partial products;
- ç is a symbol representing the bit number of the partial product and of the multiplicand;

while the expression for the exact Modified Booth Encoding is given as

$$\begin{aligned} PP\_\S(ç) <= & [(X(2\S) \oplus X(2\S - 1))(X(2\S + 1) \oplus Y(ç)) + \\ & + \overline{(X(2\S) \oplus X(2\S - 1))}(X(2\S + 1) \oplus X(\S))(X(2\S + 1) \oplus Y(ç - 1))] \end{aligned} \tag{2.2}$$

This simplification introduces 6 incorrect outputs out of 16 possible ones, but reduces both the area and the timing required by this first block. The following table shows the mistakes of the technique.

	000	001	011	010	110	111	101	100
00	0	0	0	0	1	1	1	0
01	0	0	0	0	1	1	1	0
11	0	1	1	1	0	0	0	0
10	0	1	1	1	0	0	0	0

Table 2.7: Karnaugh map of the AMBE (Approximate MBE) where the incorrect outputs are highlighted. The rows represent Y(ç)Y(ç-1) while the columns represent X(2§-1)X(2§)X(2§+1).

It is important to point out that as in the previous versions, the first triplet is made by bit '0', X(0), X(1) and that the whole block is fully combinatorial.

Dadda tree with Roorda approach

This second block takes the 6 partial products of 11 bits and compress them. Firstly the products are rounded to the 11 MSB that will be the output of the multiplier, than the Sonsa-Roorda's technique is used by means of pen and paper and the results are shown in the following figure.

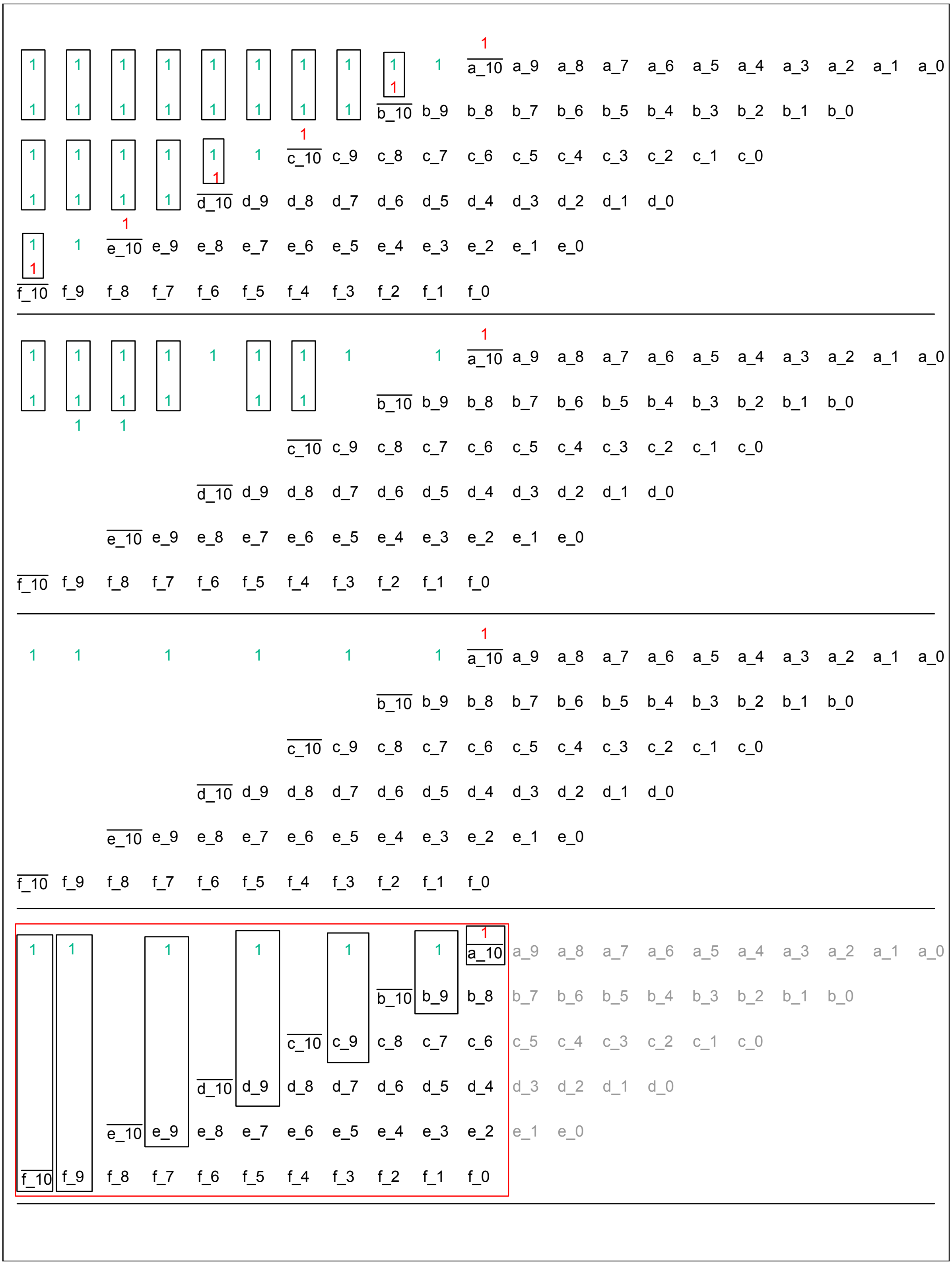


Figure 2.6: Sonsa-Roorda's technique applied to the different partial products.

Then, three possible compressors are used to speed up the computation. In particular a new compressor 4

to 2 has been used, where the sum and carry bits are computed as follows

$$\begin{aligned} Sum &= \overline{c \oplus d} \\ Carry &= \overline{\overline{a+b} + \overline{c+d}} \end{aligned} \tag{2.3}$$

where a,b,c and d are the four inputs. This new component is used even if it add mistakes to the one made by the first block as showed by the following table

d	c	b	a	Carry	Sum	Difference
0	0	0	0	0	1	1
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	-1
0	1	0	0	0	0	-1
0	1	0	1	1	0	0
0	1	1	0	1	0	0
0	1	1	1	1	0	-1
1	0	0	0	0	0	-1
1	0	0	1	1	0	0
1	0	1	0	1	0	0
1	0	1	1	1	0	-1
1	1	0	0	0	1	-1
1	1	0	1	1	1	0
1	1	1	0	1	1	0
1	1	1	1	1	1	-1

Table 2.8: Truth table of the 4 to 2 compressor where the error are computed.

In conclusion the three possible compressors (HA, FA, 4to2 compressor) have been used to implement a Dadda Three reminding that the new compressor leads to a different level structure (the difference between two level will not increase by a factor 1,5 but by a factor 2) as shown by the following figure.

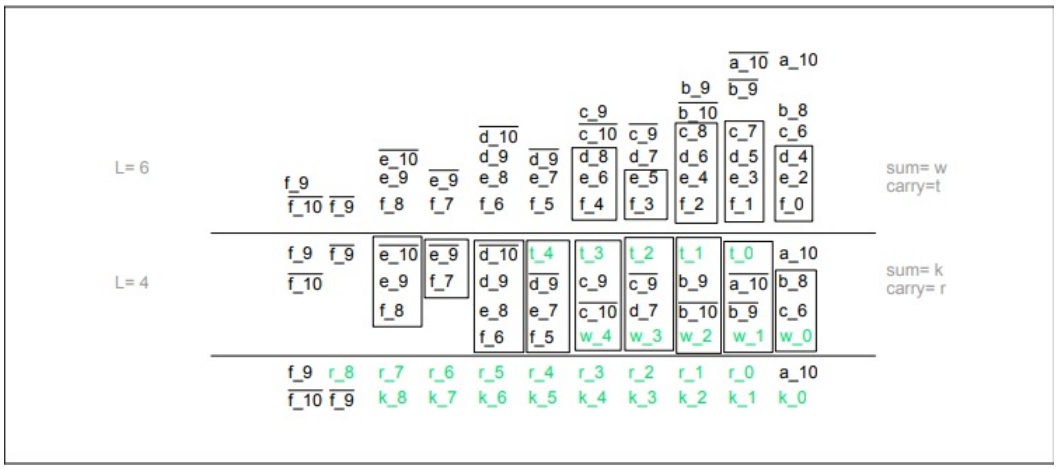


Figure 2.7: Second part of the Dadda Tree implementation, where three different compressors are used.

Chapter 3

Special Project - RISC-V, a designer step-to-step implementation's guide.

A RISC architecture is a *Reduced Instruction Set Architecture*, which represent an anti-trend innovative approach respect to the other ISA utilised in the past.
The main goal for this ISA's introduction was to provide a new implementation for a device, but more performing due to the fact that the instructions contained in the ISA are less complex.
Ideally, the operation is directly executed after the decode step, in fact, the primary and most important part for the control unit of a RISC machine is the *decoder*.
At this point, it's important to consider that we acquire code at every clock cycle, hence an *Harvard Architecture* (*Code Memory* separated from *Data Memory*) is required, as shown in fig: 3.1 .

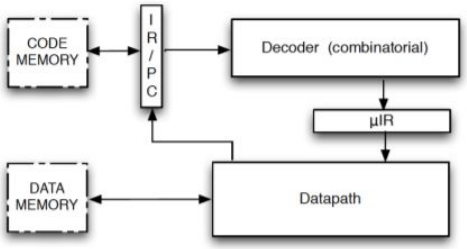


Figure 3.1: Standard Architecture for a RISC first implementation.

The performance increase respect to the clock frequency is not directly related to technology based on Moore's Law, but is the result of the new ISA introduced, allowing to reach the GHz order of magnitude, other than deep pipeline exploited and parallel execution systems.

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock reduced instruction only
Small code sizes, high cycles per second	Low cycles per second,large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Table 3.1: Compare CISC vs RISC based systems.

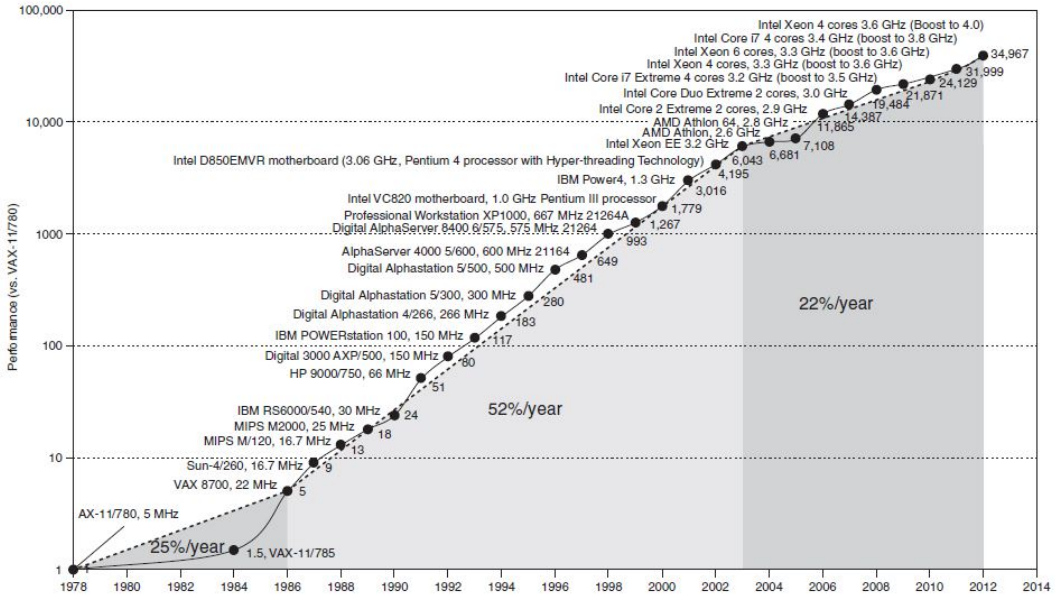


Figure 3.2: Figure of merit that shows that the frequency is still increasing thanks to the architectural solutions adopted, despite the negative contribution of high power dissipation.

The main drawback is related to the huge code memory's size required, as a consequence this need a very fast interface with the memory to satisfy the high data throughput in both directions.

RISC-V is an open ISA managed by a non-profit foundation *RISC-V Foundation* and it will not be constrained by any corporation, even in the future, to allow further implementations.
It is flexible from an application point of view (from embedded systems to high performance) and for the type of execution (in order or out of order).

3.1 Assumptions utilised & Specs

The following description is hardware-oriented and has been driven by the common specs for this project. Referring to the basic *instruction set RV32I*, the requirements are:

- Harvard Architecture;
- Cache compliant;
- There are two possible implementations : *Low Power* and *High speed*. They will be explained in the next chapters;
- Since the chosen IS is the RV32I, this means that the parallelism is on 32 bits and it is an integer unit.

The aim of this project is to provide a valid starting point for the next classes, in order to implement this basic ISA in hardware following this research.

3.2 RV32I Instruction Set Version 2.0

RISC-V support extensive customisation and specialisation. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined. In this section we will analyse the base instruction set. Just for knowledge, instruction-set extension is divided into standard and non-standard. Standard extensions should be generally useful and should not conflict with other standard

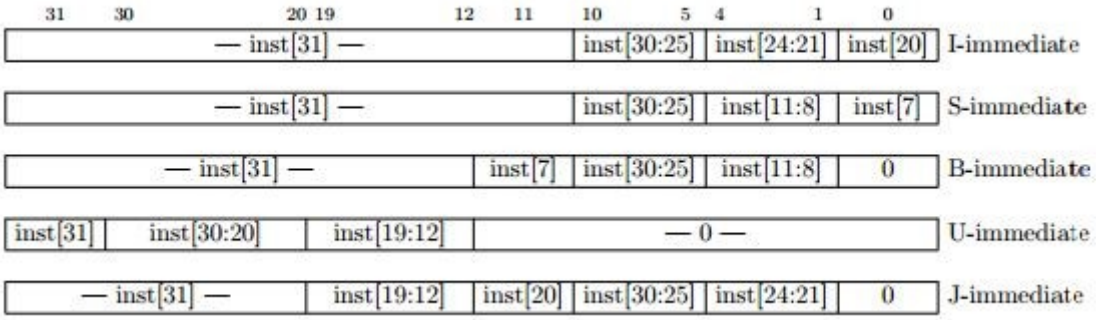


Figure 3.3: This figure shows for every instruction how to recreate the immediate value. Note that R instruction does not have works only with registers.

extensions. Non-standard on the other hand, may be highly specialised and may conflict with other standard or non-standard extensions. The most important standard ones are defined to provide:

- Multiply/Divide with the "M" extension;
- Atomic operations with the "A" extension;
- single and double-precision floating-point arithmetic with the "F" extension;

Risc V base instruction set has 47 unique instructions. In this section the focus will be on RV32I, meaning the base instruction set has 32 bit length. In RV32I ISA, there are six core instruction formats (R/I/S/B/U/J):

- R-TYPE for register-register operations;
- I-TYPE for short immediates and loads;
- S-TYPE for stores;
- B-TYPE for conditional branches;
- U-TYPE for long immediates;
- J-TYPE for unconditional jumps;

The B and J formats are variants of S and U instructions respectively, based on the handling of immediates.

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

The reason behind this two variants is that by rotating bits in the instruction encoding of B and J immediates, instead of using dynamic hardware muxes to multiply the immediate by 2, instruction signal fanout and immediate mux costs is reduced by around a factor of 2. To easy the immediate decode we link to Fig:3.3

Before explaining the instruction set some general RISC V rules should be focused on:

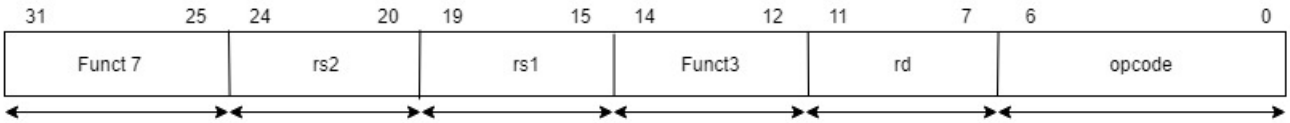
- In this architecture there are 31 general-purpose, user related, registers x1-x31 that hold 32 bit integer values. Register x0 is always hardwired to the constant 0 while for convention x1 is used to hold the return address on a call. There is one additional user-visible register: the program counter pc holds the

address of the current instruction.
Note: 16 registers would be sufficient for an integer ISA running compiled code, it is impossible to encode a complete ISA with 16 registers in 16-bit instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. A larger number of integer registers also helps performance on high-performance code, where there can be extensive use of loop unrolling, software pipelining, etc;

- The base RISC-V ISA has a little-endian memory system because little-endian systems are currently dominant commercially.
- For memory instructions the 32 bits must be aligned on a four-byte boundary. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken;
- Immediate values are always sign extended, and in immediate values bit 31 is always the sign bit. This is done to speed up sign-extention circuitry.
  This because decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats. In practice, most immediates are either small or require all XLEN bits;
- The RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding;
- Special instruction to support overflow checks on integer arithmetic operations are not implemented in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: add t0, t1, t2; bltu t0, t1, overflow. While for general signed addition: add t0, t1, t2 slti t3, t2, 0 slt t4, t0, t1 bne t3, t4, overflow;
- Control transfer instructions in RV32I do not have architecturally visible delay slots.

3.2.1 R-Type

Perform the computation rs1 OP rs2 and write result to rd. funct 7 and 3 select the operation. Most arithmetic instructions use R-type format. The possible operations are shown in Fig:3.4.



0000000	src2	src1	ADD	dest	OP
0100000	src2	src1	SUB	dest	OP
0000000	src2	src1	SLT	dest	OP
0000000	src2	src1	SLTU	dest	OP
0000000	src2	src1	AND	dest	OP
0000000	src2	src1	OR	dest	OP
0000000	src2	src1	XOR	dest	OP
0000000	src2	src1	SLL	dest	OP
0000000	src2	src1	SRL	dest	OP
0100000	src2	src1	SRA	dest	OP

Figure 3.4: R-type instructions.

ADD: perform an arithmetic addition. Note, as said earlier, that overflows are ignored and the low XLEN bits of results are written to the destination.

SUB: perform an arithmetic subtraction. Note, as said earlier, that overflows are ignored and the low XLEN bits of results are written to the destination.

SLT: perform signed compares, writing 1 to rd if rs1 < rs2, 0 otherwise.
STLU: perform unsigned compares, writing 1 to rd if rs1 < rs2, 0 otherwise. Note, SLTU rd, x0, rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (assembler pseudo-op SNEZ rd, rs).

AND: perform bitwise logical operation and.

OR: perform bitwise logical operation or.

XOR: perform bitwise logical operation xor.

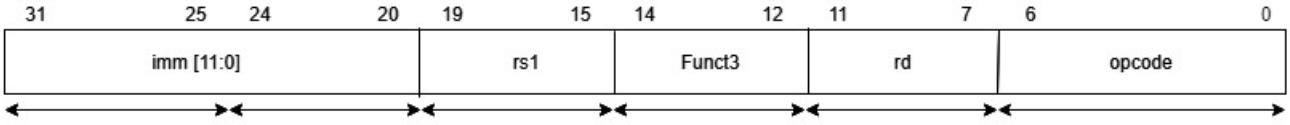
SLL: perform logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

SRL: perform logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

SRA: perform arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

3.2.2 I-Type

I-Type instructions are the most common and are shown in Fig:3.5



7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
immediate		src	ADDI	dest	OP-IMM
immediate		src	SLTI	dest	OP-IMM
immediate		src	ANDI	dest	OP-IMM
immediate		src	ORI	dest	OP-IMM
immediate		src	XORI	dest	OP-IMM
0000000		src	SLLI	dest	OP-IMM
0000000		src	SRLI	dest	OP-IMM
0100000		src	SRAI	dest	OP-IMM
offset [11:0]		base	0	dest	JALR
offset [11:0]		base	width	dest	LW
offset [11:0]		base	width	dest	LH
offset [11:0]		base	width	dest	LHU
offset [11:0]		base	width	dest	LB
offset [11:0]		base	width	dest	LBU
source/dest		source	CSRRW	dest	SYSTEM
source/dest		source	CSRRS	dest	SYSTEM
source/dest		source	CSRRC	dest	SYSTEM
source/dest		uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest		uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest		uimm[4:0]	CSRRCI	dest	SYSTEM
RDCYCLE[H]		0	CSRRS	dest	SYSTEM
RDTIME[H]		0	CSRRS	dest	SYSTEM
RDINSTRET[H]		0	CSRRS	dest	SYSTEM
ECALL		0	PRIV	0	SYSTEM
EBREAK		0	PRIV	0	SYSTEM
0 0 0 0 PI PO PR PW SI SO SR SW		0	FENCE	0	SYSTEM
0		0	FENCE.I	0	SYSTEM

Figure 3.5: I-Type instructions

ADDI: adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. Note that ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.Note that the NOP instruction does not change any user-visible state, except for advancing the pc so it is encoded as ADDI x0, x0, 0

SLTI: (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to rd.

SLTIU: (set less than immediate unsigned) places the value 1 in register rd if register rs1 is less than the sign extended immediate when both are treated as unsigned numbers, else 0 is written to rd. Note that SLTIU rd, rs1, 1 sets rd to 1 if rs1 equals zero, otherwise sets rd to 0 (assembler pseudo-op SEQZ rd, rs).

ANDI: logical operation that perform bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

ORI: logical operation that perform bitwise ORon register rs1 and the sign-extended 12-bit immediate and place the result in rd.

XORI: logical operations that perform bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. Note that XORI rd, rs1, -1 performs a bitwise logical inversion of register rs1 (assembler pseudo-instruction NOT rd, rs).

SLLI: logical left shift (zeros are shifted into the lower bits).

SRLI: logical right shift (zeros are shifted into the upper bits).

SRAI: arithmetic right shift (the original sign bit is copied into the vacated upper bits).

JALR: (jump and link register) the target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Note that register x0 can be used as the destination if the result is not required. For return-address prediction JALR instructions should push/pop a RAS as shown in Tab:3.2

rd	rs1	rs1 = rd	RAS action
!link	!link	-	none
!link	link	-	pop
link	!link	-	push
link	link	0	push and pop
link	link	1	push

Table 3.2: Return-address stack prediction hints encoded in register specifiers used in the instruction. In the above, link is true when the register is either x1 or x5.

Basically when two different link registers (x1 and x5) are given as rs1 and rd, then the RAS is both pushed and popped to support coroutines. If rs1 and rd are the same link register (either x1 or x5), the RAS is only pushed to enable macro-op fusion of the sequences: lui ra, imm20; jalr ra, ra, imm12 and auipc ra, imm20; jalr ra, ra, imm1.

LW: loads a 32-bit value from memory into rd.

LH: loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd.

LHU: loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd.

LB: loads a 8-bit value from memory, then sign-extends to 32-bits before storing in rd.

LBU: loads a 8-bit value from memory but then zero extends to 32-bits before storing in rd.

CSRRW: (Atomic Read/Write CSR - Control Status Register) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

CSRRS: (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zeroextends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written). Note that if rs1=x0, then the instruction will not write to the CSR at all, and so shall

not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs.

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zeroextends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected. Note that if rs1=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs.

CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12-bit CSR address space and accessed in 32-bit pieces using CSRRS instructions.

RDCYCLE: reads the low XLEN bits of the cycle CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. Reads bits 63-32 of the same cycle counter. The underlying 64-bit counter should never overflow in practice. The rate at which the cycle counter advances will depend on the implementation and operating environment. The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

RDTIME: Reads the low XLEN bits of the time CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is an RV32I-only instruction that reads bits 63-32 of the same real-time counter. The underlying 64-bit counter should never overflow in practice. The execution environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant. The real-time clocks of all harts in a single user application should be synchronized to within one tick of the real-time clock. The environment should provide a means to determine the accuracy of the clock.

RDINSTRET: Reads the low XLEN bits of the instret CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past. RDINSTRETH is an RV32I-only instruction that reads bits 63-32 of the same instruction counter. The underlying 64-bit counter that should never overflow in practice.

ECALL: Make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

EBREAK: Used by debuggers to cause control to be transferred back to a debugging environment.

FENCE: used to order device I/O and memory accesses as viewed by other RISCV harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE. The execution environment will define what I/O operations are possible, and in particular, which load and store instructions might be treated and ordered as device input and device output operations respectively rather than memory reads and writes.

FENCE.I: used to synchronize the instruction and data streams. RISC-V does not guarantee that stores

to instruction memory will be made visible to instruction fetches on the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does not ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system.

3.2.3 S-Type

We recall from the section introduction that S-Type are similar to B-Type instructions.

S-Type instructions are specific for store instructions. This is showed in Fig:3.6

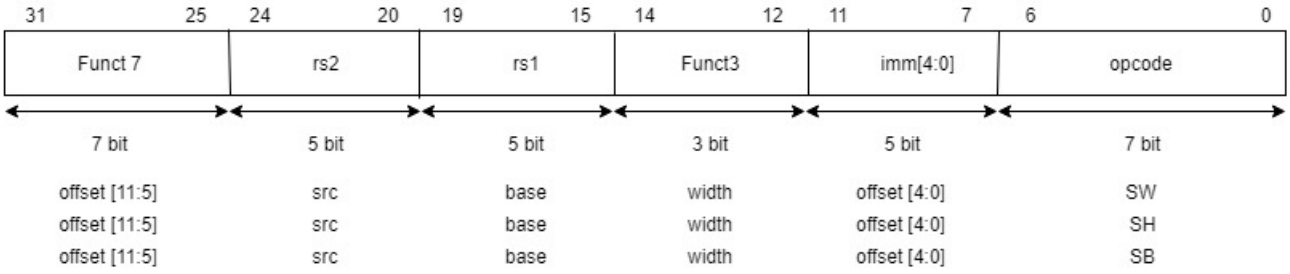


Figure 3.6: S-Type instructions.

SW: store 32-bit values from the low bits of register rs2 to memory.

SH: store 16-bit values from the low bits of register rs2 to memory.

SB: store 8-bit values from the low bits of register rs2 to memory.

We recall also from the introduction of the section that all load and store instructions should be naturally aligned. The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

3.2.4 B-Type

We recall from the section introduction that B-Type are similar to S-Type instructions.

All branches uses B-Type format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current pc to give the target address. The conditional branch range is +/- 4 KiB.

Branch instructions compare two registers and are shown in Fig3.7

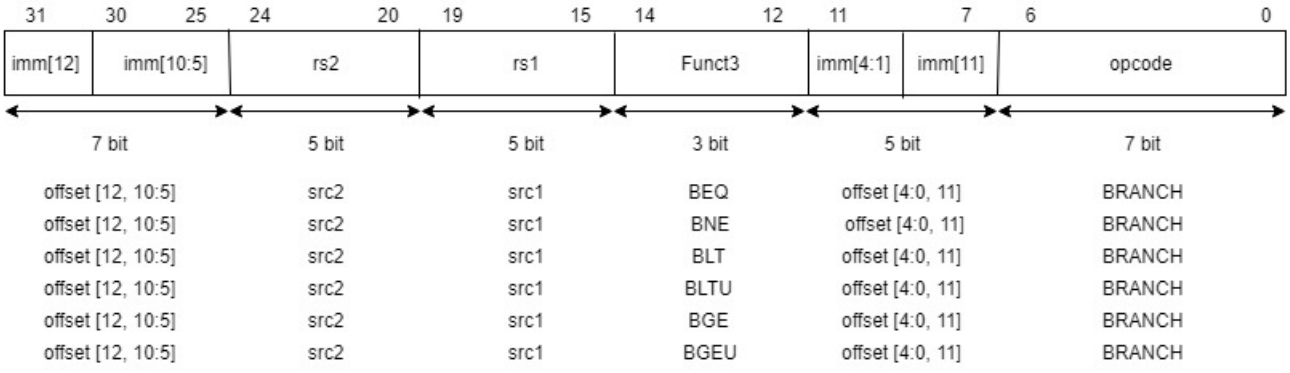


Figure 3.7: B-Type instructions.

BEQ: take the branch if registers rs1 and rs2 are equal.

BNE: take the branch if registers rs1 and rs2 are unequal.

BLT: take the branch if rs1 is less than rs2, using signed comparison.

BLTU: take the branch if rs1 is less than rs2, using unsigned comparison.

BGE: take the branch if rs1 is greater than or equal to rs2, using signed comparison.

BGEU: take the branch if rs1 is greater than or equal to rs2, using unsigned comparison.

Note that BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively. Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the
rst time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

3.2.5 U-Type

We recall from the section introduction that U-Type are similar to J-Type instructions. Only two instructions are encoded in U-Type as shown in Fig:3.8

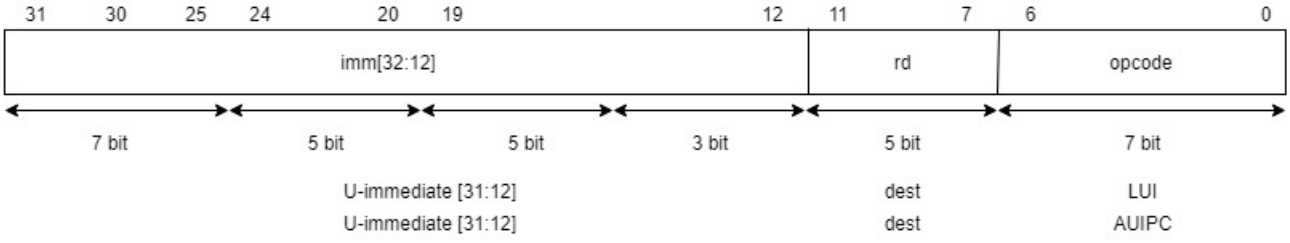


Figure 3.8: U-Type instructions.

LUI: (load upper immediate) places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

AUIPC: (add upper immediate to pc) forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd. It is used to build pc-relative addresses.

- LUI (load upper immediate). Is used to build 32-bit constants.Places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.
- AUIPC (add upper immediate to pc) is used to build pc-relative addresses. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

Note that The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address. The current PC can be obtained by setting the U-immediate to 0.

Note also that, similar to JALR, for return-address prediction purpose A JAL instruction should push the return address onto a return-address stack (RAS) only when rd=x1/x5.

3.2.6 J-Type

We recall from the section introduction that J-Type are similar to U-Type instructions. In RV32I just JAL (jump and link) instruction is encoded, as shown in Fig:3.9

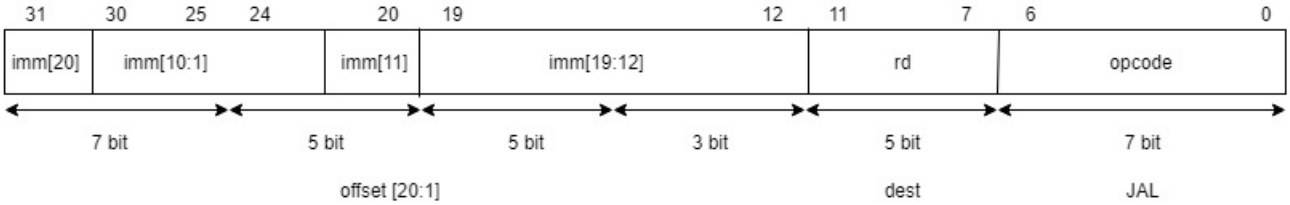


Figure 3.9: J-Type instructions.

JAL: the immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. JAL stores the address of the instruction following the jump (pc+4) into register rd.

Note that the standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

Note that plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with rd=x0.

3.3 Basic RISC-V Architecture - Data Path & Control.

Starting from the instruction set, a basic implementation will be derived and explained in the following paragraphs, underlining how and how much the instruction set affects performances and the overall design approach. We can define three classes of instructions:

- Memory-reference;
- Arithmetic-logical;
- Branches;

The action that are needed to exploit one instruction are basically the same: loading the program counter, fetching the instruction and decoding it. The only exception are the steps after the decode stage, because they depend on the instruction class, but still independent from the single instruction.
Following the actions needed in order to execute an instruction, or a group of them in this case, we can simply derive an harware implementation.
The simplicity and regularity of the RISC-V instruction set simplify the implementation by making the execution of many of the instruction classes similar.
All the classes use the ALU, after reading the registers needed and indicated in the fields of the instruction.

- Memory-reference : uses the ALU for address calculation;
- Arithmetic-logical : uses the ALU for operation execution ;
- Branches: uses the ALU for the equlity test;

After using the ALU:

- Memory-reference : need to access the memory either to read data for a load or write data for a store.
- Arithmetic-logical : write the data from the ALU or memory back into a register.
- Branches : we may need to change the next instruction address based on the comparison; otherwise, the PC should be incremented by four to get the address of the subsequent instruction, due to the fact that the instructions are on 32 bits stored per bytes, hence every 4 bytes in memory there will be a new one.

Multiplexers are needed to control the flow of data arriving from different sources, in addition to the *control* to select the right path according to the current instruction reported at the input of the control unit, fig:: 3.10.

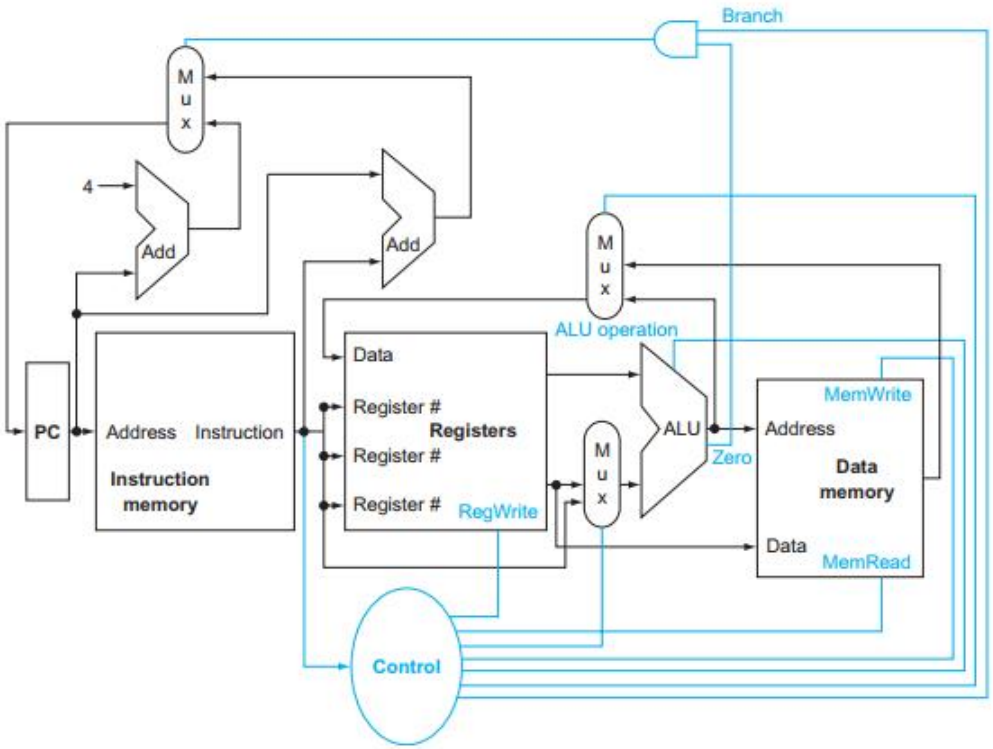


Figure 3.10: Showing the basic high level implementation for the data path and control unit of a RISC-V, based on the requirements for the instruction classes.

3.3.1 Elements needed for the Data Path

For every instruction we need:

- Memory unit : to store the instruction ad supply them given an address;
- Program Counter (PC) : register that holds the address of the current instruction.
- An adder to increment the PC to the address of the next instruction.
- Register File Structure to store all the 32 general purpose registers. To read a value from it we need the *register number* as a input and the register content display consequently at the output, to write to a register in the register file we need the register number that identify the register and a input to feed the data to the its destination.

Anlyzing only the R-Type instruction (belonging to *Arithmetic-logical* class of instructions). They all read two registers, perform an ALU operation on the contents of the source registers, and write the result to a destination register.

Considering only the load and store instructions, they both compute a memory address by adding the base register to the 12-bit signed offset field in the instruction. The load instruction need to read from memory a value and transfer it on a register, the store does the exact contrary.
For these specific instructions we need:

- A Data Memory, to be read or write;
- a Sign Extender, to extend the value of the offset from 12 bit to 64-bit (depending on the desired data parallelism of the machine).

The branch instruction class, in the form of *beq*, compare the equality of two registers and use a 12-bit field to compute the *branch target address* by adding the offset to the current value of the PC.In conclusion, the base address is the address of the branch instruction and the offset is shifted by 1 to the left, which means that the

actual range covered increases by a factor 2.
There are two operation in this case that need to be done : compute the branch target address and at the same time verify if the branch as to be *taken* or continue with the next value of the PC (*branch not taken*). To compute the branch target address:

- Immediate generation unit : has a 32-bit instruction as a input and provides the 12-bit field, then extended at its output. The logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for *load* instructions, bits 31:25 and 11:7 for *store* instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch.
  Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field.
  RISC-V opcode bit 6 is 0 for data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 is 0 for load instructions and 1 for store instructions.
  Thus, bits 5 and 6 can control a 3:1 multiplexer inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

- Adder : to compute the final branch destination.

while, to perform the compare operation we need to use the register file in order to supply the two register operands to the ALU and control signal as well, to perform a subtraction. If the result is equal to Zero, the two operand are equal and the branch needs to be taken because the condition is verified.
The branch instruction operates by adding the PC with the 12-bits of the instruction shifted left by 1 bit, simply by concatenation.
Combining these elements that we have described, starting from the single instruction classes, and adding control signal we derive a full data path, fig: 3.11, keeping in mind that:

- It's not allowed to point to a memory with a non-existing address, while for the register file is, hence we need a read signal for the memory.

- We need two separate memory : one for the data one for the instructions, the other units can be shared.

- If a unit is needed for multiple actions (*e.g.* to compute an R-Type Instruction, the target address of the branch, the address for *load* and *store* operations), it needs to be duplicated.

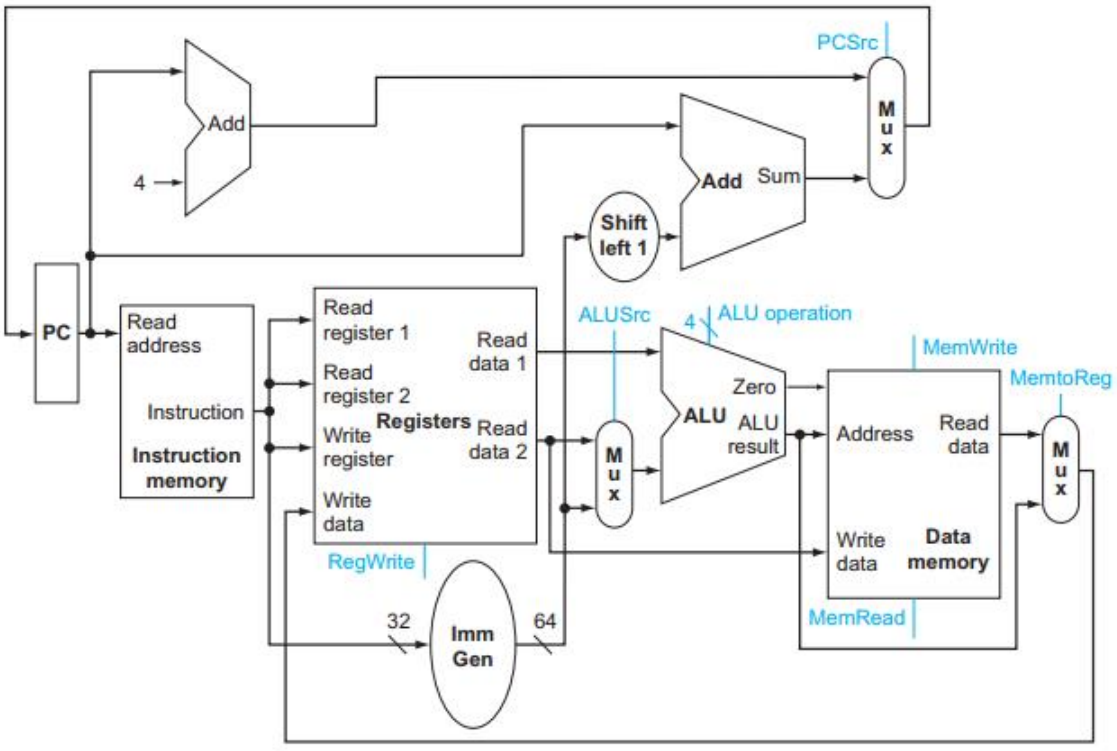


Figure 3.11: A full Single data path with control signals and comprehensive of all the units enumerated before.

3.3.2 Control

ALU Control

Since the controls are different based on the class of the instructions, the general idea is to use 2-bits wich are called *ALUOp* as a input:

- If *ALUOp* = *"00"*, the operation performed by the ALU will be an *add*, for load and stores;
- If *ALUOp* = *"01"*, the operation performed by the ALU will be an *subtraction* with a test if the result is equal to zero, for branch if equal;
- If *ALUOp* = *"10"*, the operation performed by the ALU will be determined based on the fields *func7* and *func3* (both of 4 bits), for logical operations.

The output of this small control unit will be a 4-bits signal that controls directly the ALU.

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Figure 3.12

In this way, the control signals require for the correct functioning of the data path depend only on the format and the content of the instructions, which is *standard* and *regular*.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 3.13

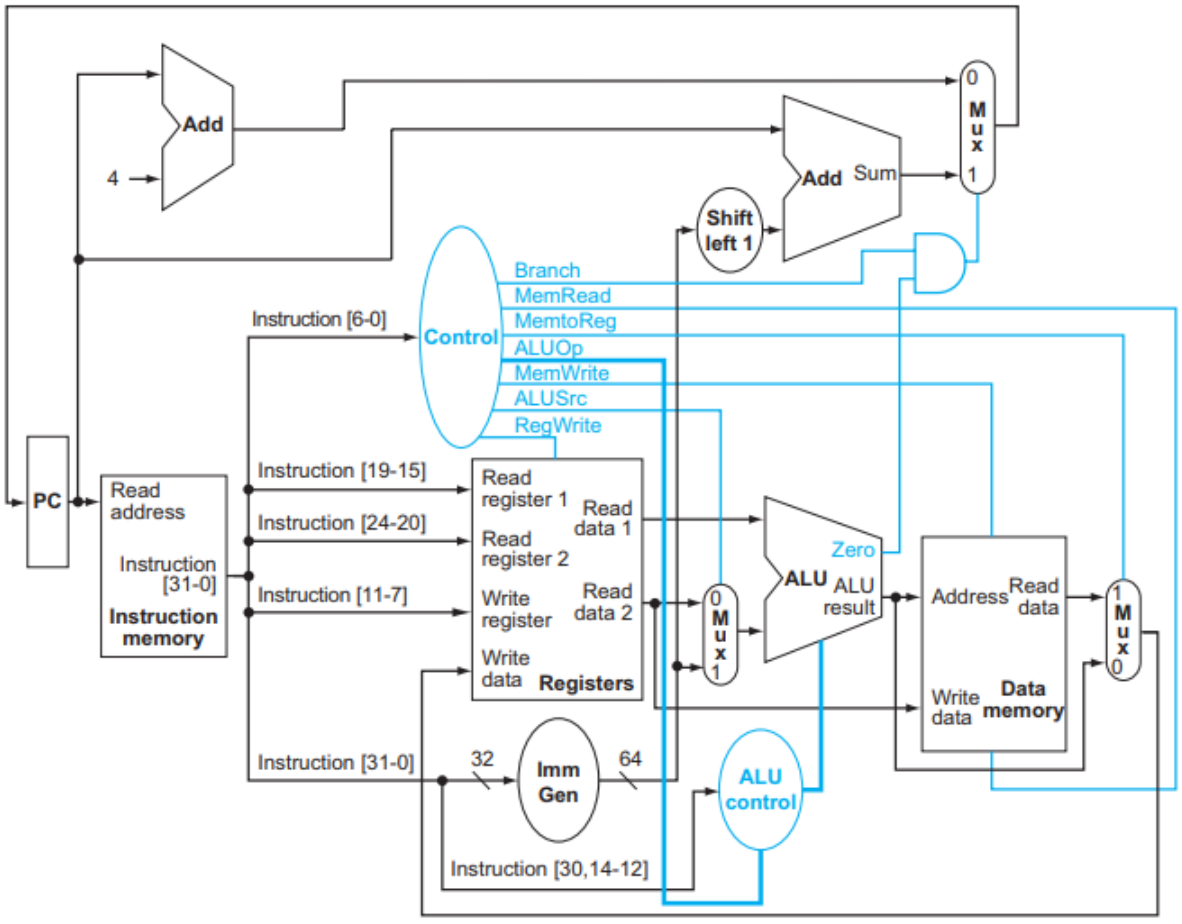


Figure 3.14

Moreover, the setting of the control lines is completely determined by the opcode fields of the instruction as shown in Fig: 3.15, where the outputs are the control lines and the inputs are the opcode bits.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Figure 3.15

3.3.3 Pipelined Control

By the way, the instruction set of the RISC-V was designed for pipelined implementation, hence, a pipelined control is required.
All the instructions are the same lenght and it make the fetch and decode operations more easy. In addition, the source and destination registers fields are located in the same place in each instruction and the memory operands appears only in *load* and *stores* , this means that we can use the execute stage to calculate the memory address and then access memory in the following stage.

The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.
The identified stages are, as always:

- IF : Instruction Fetch;
- ID : Instruction Decode;
- EX : Execution or address calculation;
- MEM : Data Memory access;
- WB : Write Back - places th result back into the register file in the middle of the data path;

Data are normally flowing from left to right, except for the WB, but it doesn't affect the current instruction, these reverse data movements influence only later instruction in the pipeline.
Note that the right to left flow can lead to *data hazards* while the update of the PC (choosing between the incremented value and the branch address from the MEM stage) leads to *control hazards*.

For the pipelined control we need to set the control values during each pipeline stage , because each control line is associated with a component active in only a stage of pipeline. We can divide control lines into five groups, according to the pipeline stage:

- IF : control signal to read the instruction memory and to write the PC are always asserted;
- ID / Register File Read : the two source registers are always in the same location in the RISC-V instruction formats;
- Execution / Address Calculation : the signals select the ALU operation and either *Read data 2* or a sign-extended immediates as inputs to the ALU.
- Memory access : the control lines set in this stage are Branch (by *beq*), MemRead (by *load*) and MemWrite (by *store*). *PSCrc* selects the next sequential address unless control asserts *Branch* and the ALU results was 0.

- Write Back: the two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and RegWrite, which writes the chosen value.

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Figure 3.16

The simplest way to pass these control signals is to extend the pipeline registers to include the control information : as the data move down in the pipeline, the number of control signal decrease, because not all the controls are used in the same stage, at the same time, Fig: 3.17.

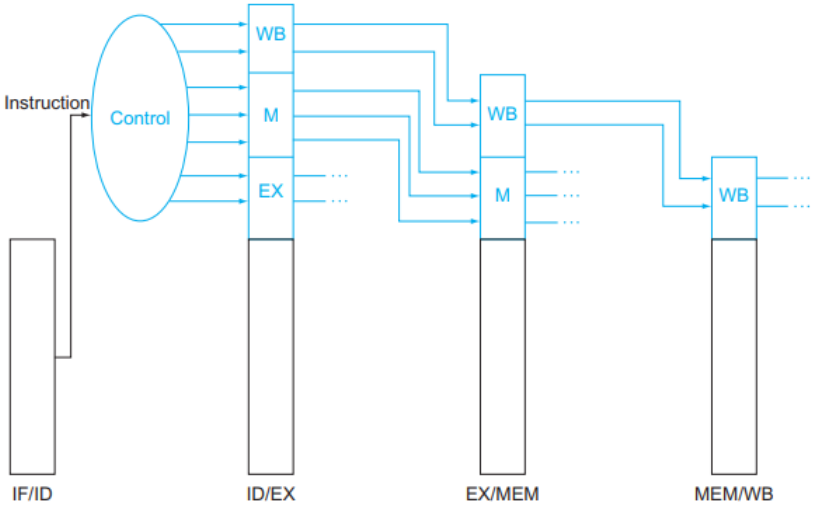


Figure 3.17

The Fig: 3.18 shows the full data path with the extended pipeline registers and with the control lines connected to the proper stage.

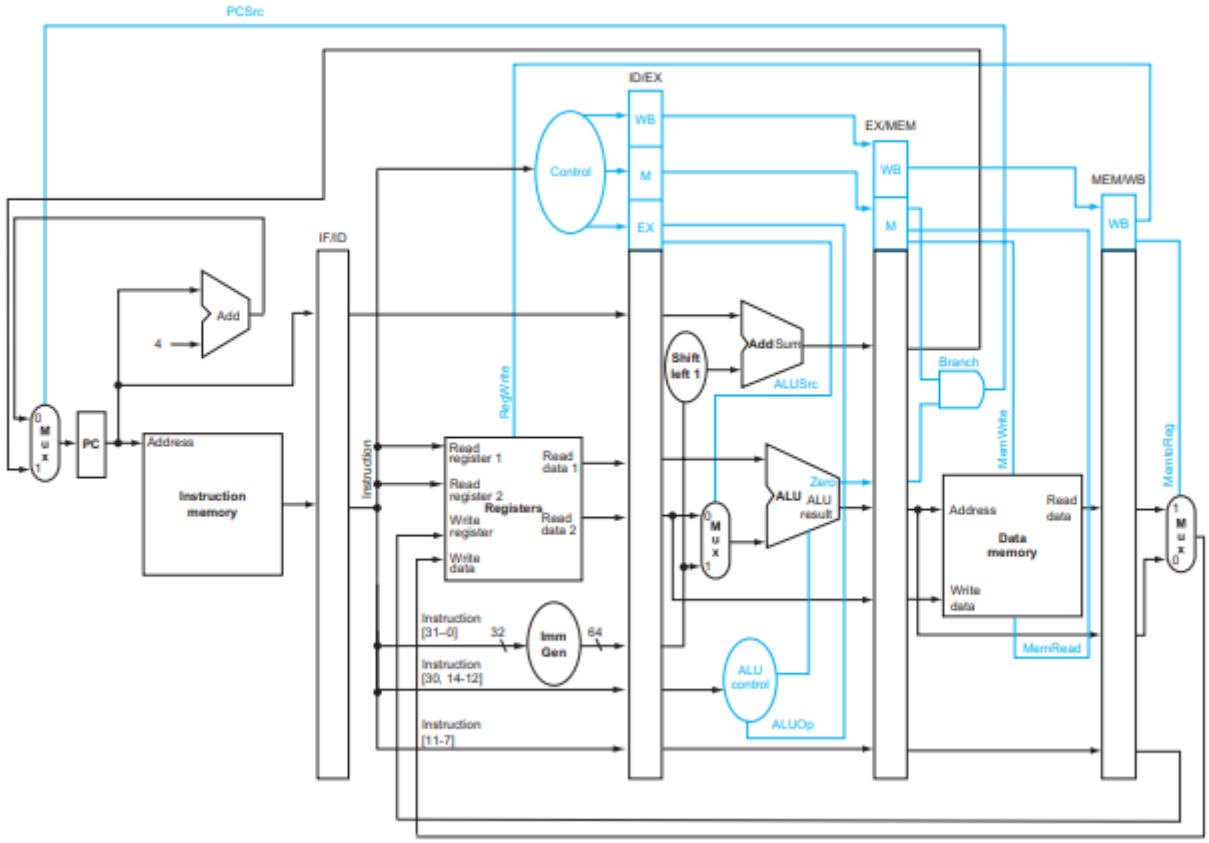

PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Instruction [31–0]
32
Imm Gen
64
Instruction [30, 14-12]
Instruction [11-7]
Shift left 1
Add Sum
ALUSrc
ALU
Zero
ALU result
ALU control
ALUOp
Branch
MemWrite
MemRead
Address
Read data
Data memory
Write data
MemtoReg

Figure 3.18

3.4 Description and possible solution implementation of various RISC-V problems

A pipelined organization of the processor guarantees a high throughput, however we have to analyze how hazards are generated and how handled them. Hazards can be divided into:

- **Structural hazards** : It means that there's a conflict on the use of a piece of hardware.

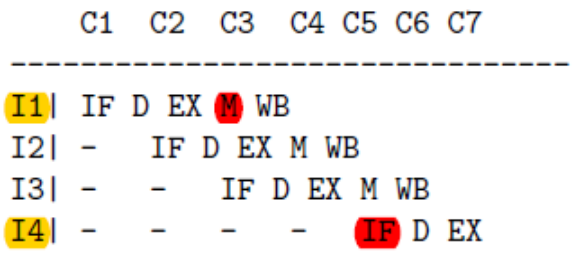


Figure 3.19: An example of structural hazard in flow of instructions.

Like showing in the fig: 3.19, This is a typical instructions flow with no branches; let's consider instruction I1 and I4 in cycle number 4 (C4), here we have for I1 the memory stage and for I4 the instruction fetch: if we are using a *Von Neumann architecture* (where there's only one memory) we cannot perform a write and a read in the memory at the same time. There is no solution for this kind of hazard, to avoid them we have to design the proper ISA and architecture.

- **Control hazards** : occur every time we have a deviation from the sequential flow since we may have problems with the pipeline stages.

Possible solution are :

- *Improve decode stage* : in the decode stage some more hardware employed to perform the computation of the target address and the comparison between two registers in this way we know if we have to jump and the target address; meaning that in this way we have only one stage penalty. This solution however is very expensive since we need a comparator for each kind of branch instruction.

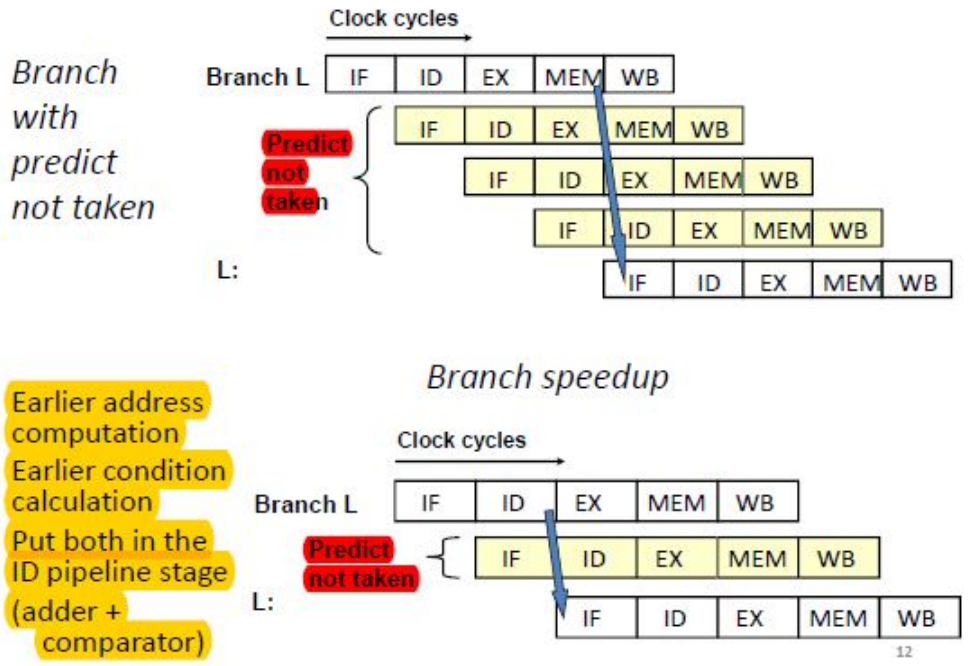


Figure 3.20: An example of control hazard improvement in flow of instructions.

- *software approach* : it consists of placing just after the branch instructions that have always to be executed independently on the result of jump comparison
- *hardware approach* : dynamic prediction and static prediction.
- **Data Hazards** : It means that a data which is not yet ready in the register file has to been read. It is linked with *Data dependecies* when we need a data and when data is ready.

```
I1:add R1  R2  R3      R1 <- R2+R3 // R1 is dest reg
I2:add R4  R1  R5      R4 <- R1 + R5 // R1 is source reg
```

Figure 3.21: An example of data hazard in flow of instructions.

```
C1 C2 C3 C4 C5 C6
-------------------------
I1| IF D EX M WB
I2| -   IF  D   EX  M   WB
```

Figure 3.22: An example of data hazard in flow of instructions with pipe stages.

Like showing in fig: 3.22, at the end of 5th clock cycle the result of R2+R3 is available in R1, however we would like to read it in the decode stage of the second instruction, i.e. in 3rd clock cycle. The problem derives from the lenght of pipeline infact it is higher is the also higher the possibility to have problems like this.
Possible solutions are :

- *Stall* : It means introduce NOP between the instructions that are data dependecies giving time to complete the operation ma it is inefficient but no extra hardware needed.

- *Scheduling at compiler level* : A compiler/assembler translates assembly language into binary code to be downloaded into the instruction memory. By doing that, the compiler may be asked to perform some optimizations like finding data hazard instruction and insert between them some other useful instructions while avoiding data dependencies. If this rescheduling is not possible the compiler will insert NOPs;
- *Forwarding/bypassing* : provides the possibility to choose different sources thanks different multiplexers that are select by a proper control in order to find the right source data

Now we describe and analyze various unit solutions to solve the conflicts we described before.

3.4.1 Branch prediction unit

The prediction mechanism can be static or dynamic. The first means that for every single branch a fixed direction is always predicted, the choice regarding the direction is taken at compiler time or during profiling meaning that if a certain branch is taken 80 per cent of times than that branch will takes always that direction. Efficiency is about 66.6 per cent. Instead in dynamic prediction the processor decides at run time which branch is more probable; it is a key point to be very efficient in the prediction.

Algorithm and idea

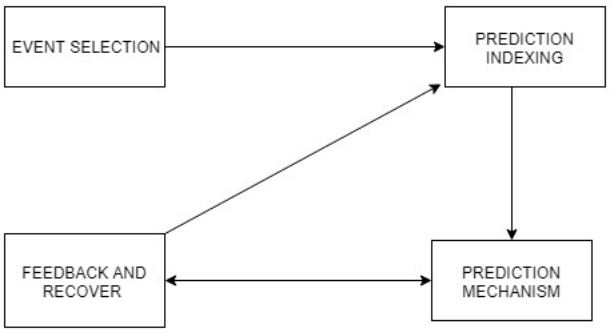


Figure 3.23: branch prediction scheme

This unit has to become active when one of the following events occur: branch, call, jump or everything expecting the program counter not incrementing sequentially.Once we have run the prediction mechanism we have take a decision and after some cycles we will know if the decision taken was correct or not, basing on this information the prediction algorithm can be made adaptive by updating something in the prediction table.

Principle purposes of the various unit scheme are:

Prediction indexing : store to report info about the past execution of the instruction we will predict and update those information.

Feedback mechanism : update the table once we verified if prediction is correct or not.

3.4.2 Possible implementations dynamic prediction

1 BIT PREDICTION

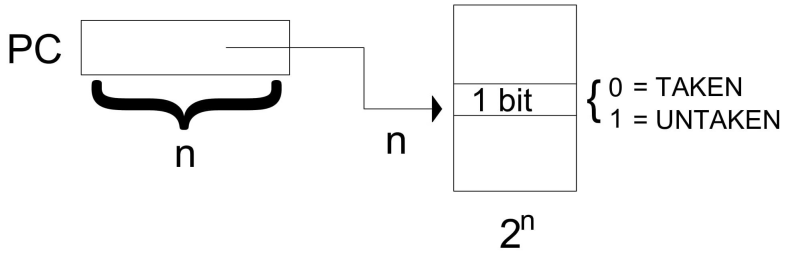


Figure 3.24: 1 bit prediction scheme

Like showing in the fig: 3.24 using PC as address we point to a certain location in the table and the pointed value tells us if we have or not to take the branch (0 or 1).

- *Advantage*: 85 per cent of efficient, hardware is no so complex.
- *Drawback* : When instructions are not branches a lot of space is wasted in this memory.

To solve this last problem One possible idea is to not use all n bits of PC but just a subset made up of k bits. In this way some aliasing is present since we point to the same location in the table for more than one instructions (proper choice of k to limit the problem).

An another possible solution is *cache-like-organization*(intel architecture)

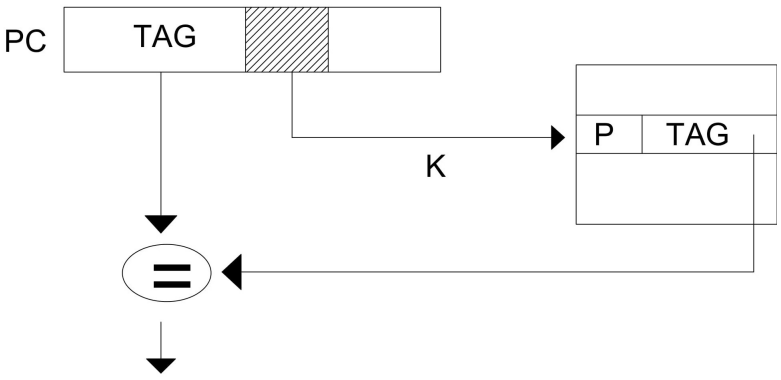


Figure 3.25: 1 bit prediction cache like

Like showing in the fig: 3.25 in The table is larger than the previous case and it is divided into two field: left one is the same as before, right one corresponds to a tag. By comparing table cell tag with the one in PC

aliasing is avoided.

2 BIT PREDICTION

to increase prediction mechanism performance instead of using a single bit we can use 2 bits. In this case we case 4 possibilities:

- *ST*: means strong taken, so there is a high reliability on that.
- *WT* : means weak taken: the reliability for this choice is less than the previous one.
- *SU*: means strong untake.
- *WU* : means weak untake.

It works like a simple FSM , like showing in the fig: 3.26.

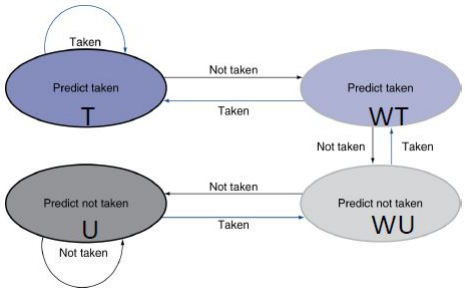


Figure 3.26: FSM of 2 bit prediction

At the end we can say that 2-bit prediction has a limited extra cost with respect to one bit but in nested loop it has a -50 percent of wrong prediction.

CORRELATED BRANCHES PREDICTION

They are not statically independent each other, we should be able to detect this kind of situation to be able to significantly improve our performance.
Key point is to have a kind of shift register where every new instruction to be predicted is inserted at the right and then content is shifted, for a certain instruction we insert 0 or 1 depending if in the last occurrence that branch was taken or not. This register is employed to indexed the prediction table, now called *Pattern History Table* : 3.27.

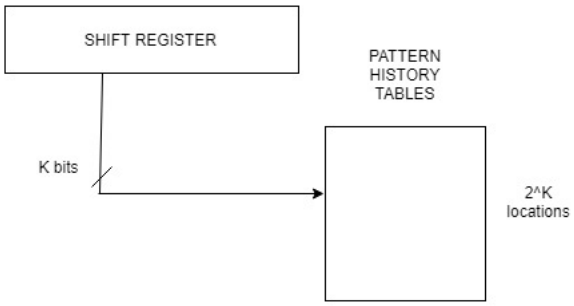


Figure 3.27: Implementation for correlated branches

GENERAL IMPLEMENTATION

Now we mix the different solutions to improve our predictions. There are different variants:

- *GLOBAL-GLOBAL*: We mix together PC (local address) and shift register (global information) using an hash table which output is employed to point at PHT

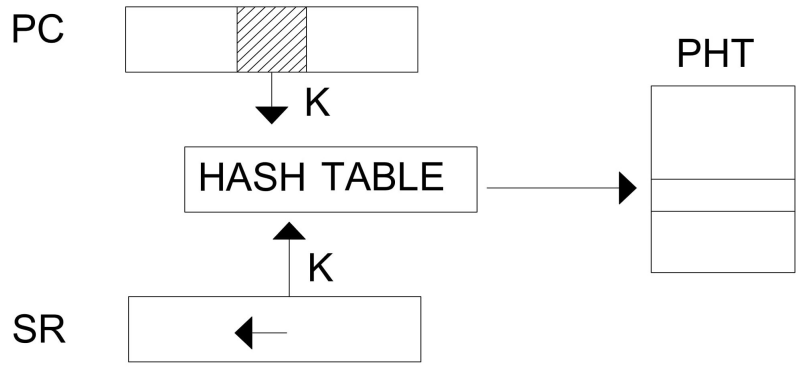


Figure 3.28: Global-global implementation

- *GLOBAL-SET* : There are separate tables for each branch instruction, so by using k bits from PC we select the corresponding PHT and then we use the Shift Register to make an access to the selected PHT.

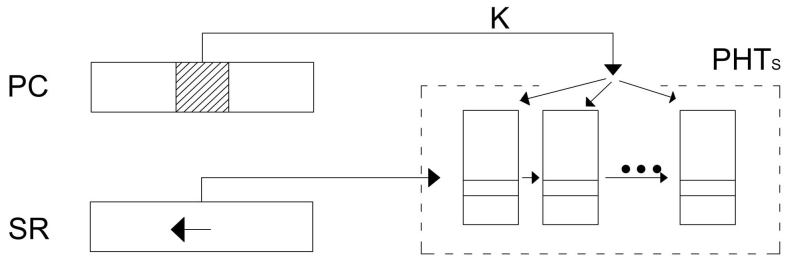


Figure 3.29: Global-set implementation

- *SET-GLOBAL*: One single PHT but many SR.

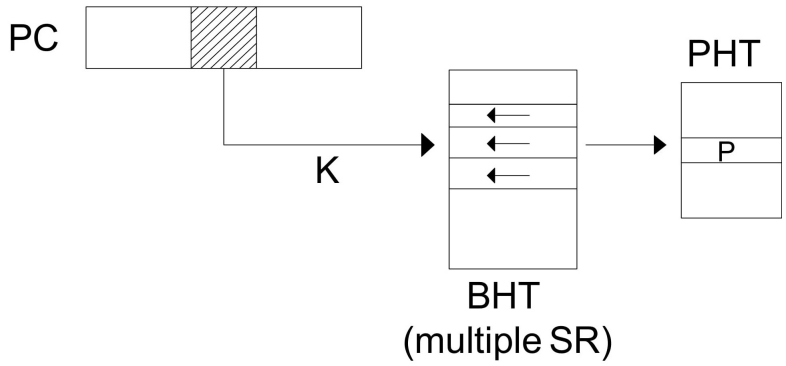


Figure 3.30: Set-global implementation

- *SET-SET* : Bits coming from PC are separated into two fields, this solution reduces aliasing more than the other schemes but it also the more expensive.

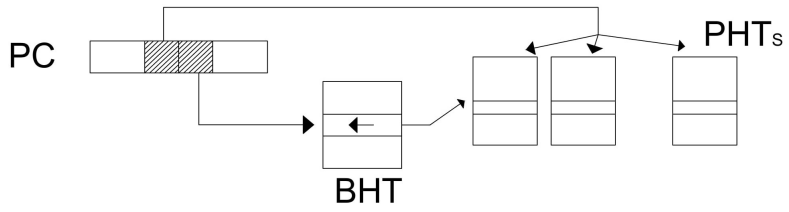


Figure 3.31: Set-set implementation

- *HYBRID PREDICTION*: very flexible solution because if we to have more than one predictors so a meta-predictor choosing dynamically which predictor which one has to be used.

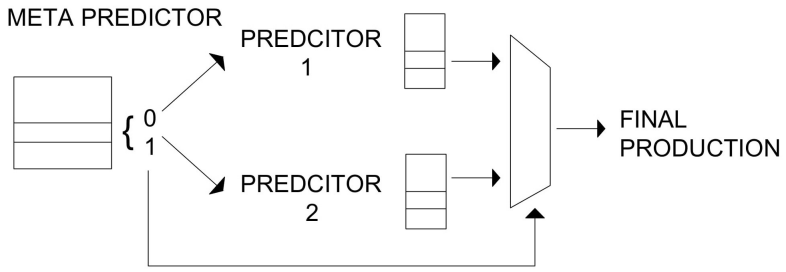


Figure 3.32: Hybrid prediction implementation

3.4.3 Forwarding unit

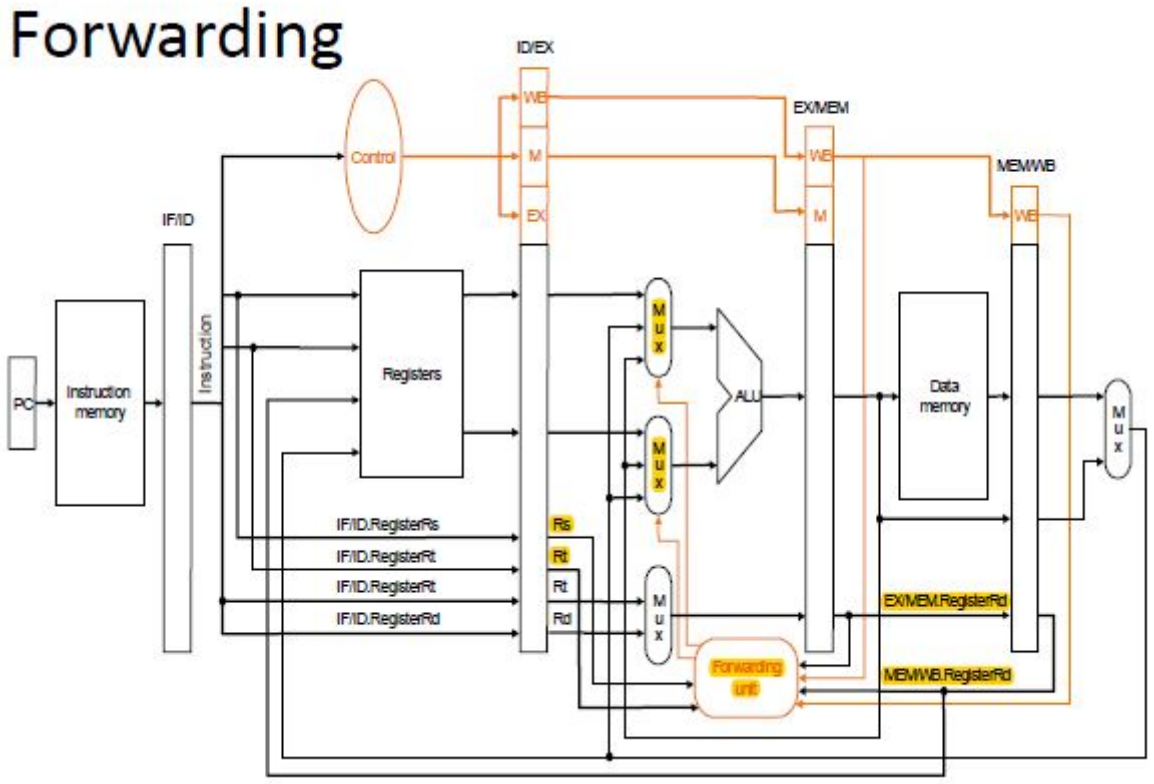


Figure 3.33: Forwarding unit complete scheme.

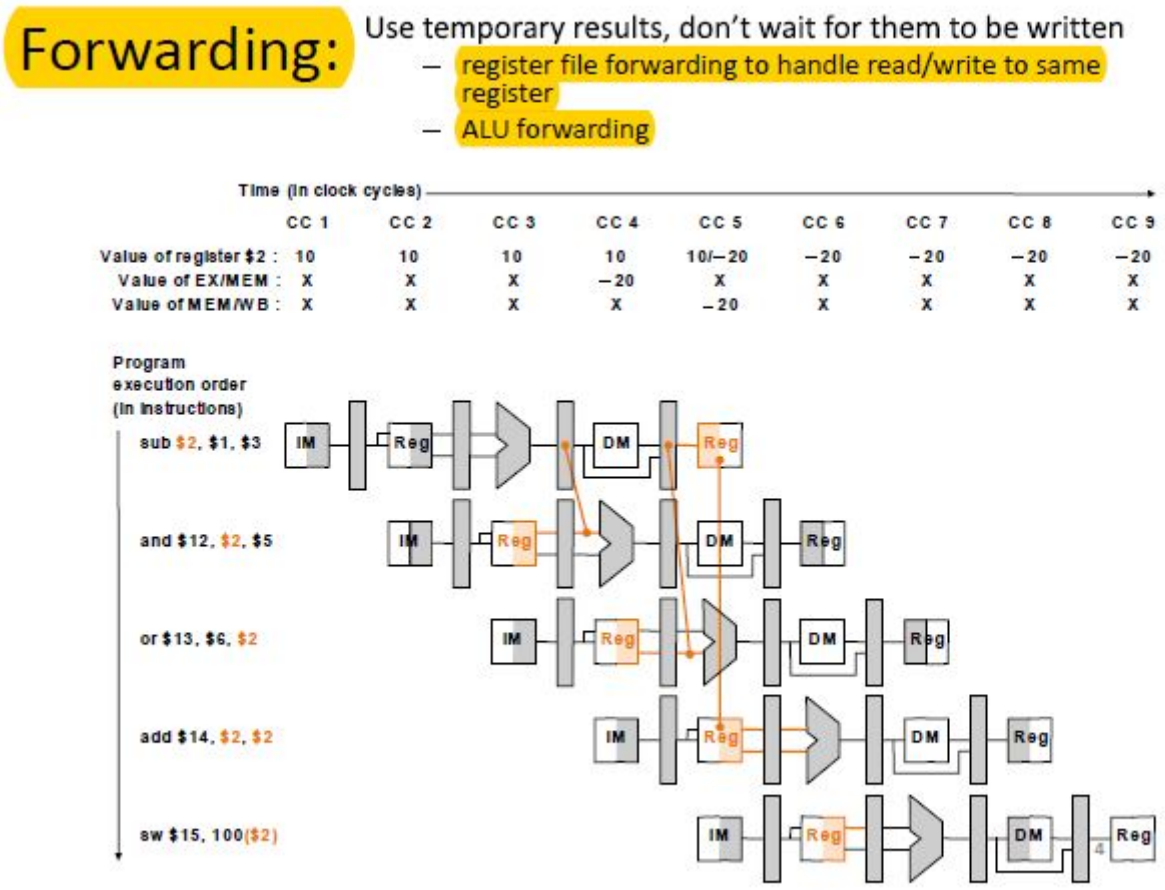


Figure 3.34: As a Reference : forwarding unit functional block example.

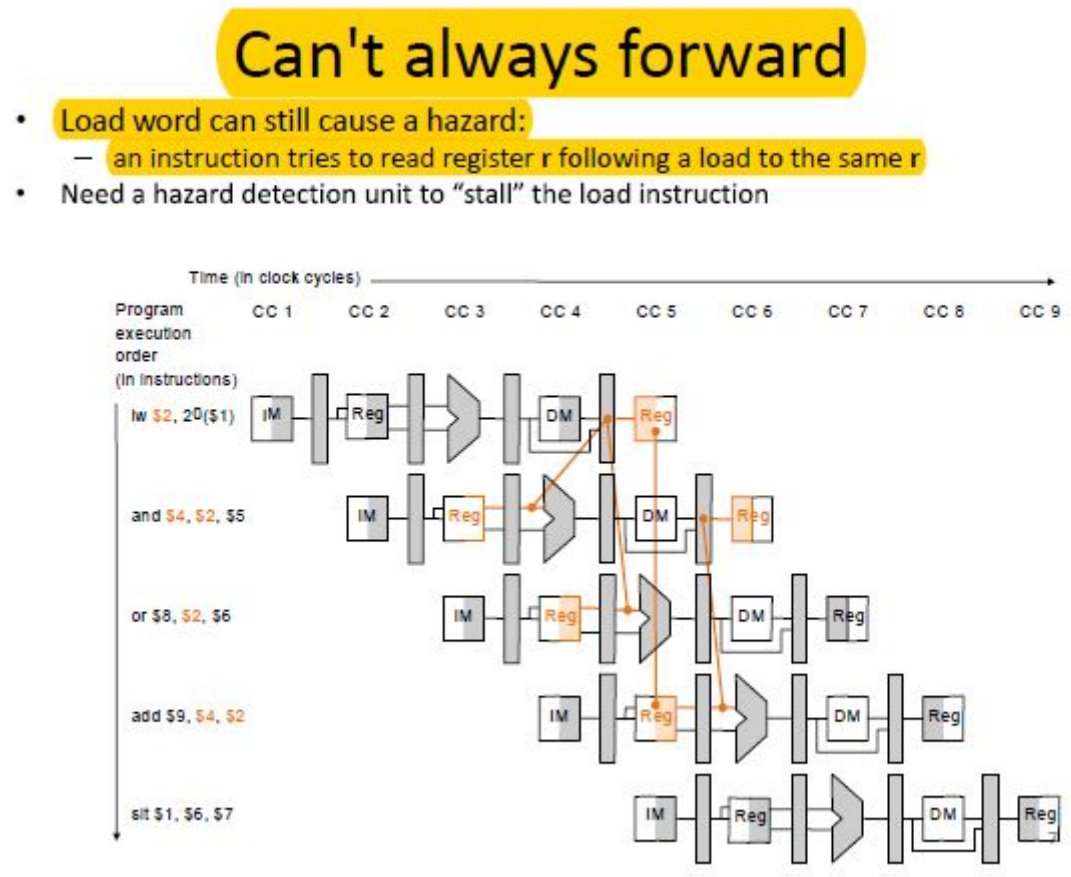


Figure 3.35: forwarding unit functional block example where forwarding do not work.

3.4.4 Scoreboard unit

To study the scoreboarding unit we will not analyze DEC alpha one but a more complete which is able to handle also other kind of conflicts (not only RAW). It is made up of several stages.

- *Issue step*:

  Check for structural hazards.
  Check for WAW.
  If these two checks are passed instruction is issued, otherwise it remains in this stage

- *Read or dispatch step*:
  Read operands.
  Check for RAW

- *Execution step*;
  WAR conflict detection.

- *Write step* :

Scoreboard data structures

Some data structures are allocated in the scoreboarding unit

- *Instruction status table*:

Instruction	Issue	Read	Execution	Write
Instr. 1				
Instr. 2				
Instr. 3				
Instr. 4				

Figure 3.36: Instruction status table.

- *Functional unit status table*:

FU	Busy	Op	Dest.reg (F_i)	Source reg 1 (F_j)	Source reg 2 (F_k)	Q_j	Q_k	R_j	R_k
FU1									
FU2									
FU3									
FU3									

Figure 3.37: Functional unit status table.

- *Register result status table*:

Step	Condition check	Actions (if all checks are passed)
Issue	• FU free? • WAW	• Set FU busy. • Record $F_i, F_j, F_k, Q_j, Q_k, R_j, R_k$. • Store FU name in RF.
Read	RAW conflict exists if either R_j or R_k is equal to 0.	Send F_j, F_k to the selected FU.
Execution	-	-
Write	WAR conflict(F_i for one instruction, for the other instructions already issued F_j and F_k).	-

Figure 3.38: Register status table.

3.5 RISC-V compiler

In this section we will see how a simple C program can be translated into RISC-V machine instructions.
We will use a cross compiler. A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
There are many way to obtain them. Here we will analyze two kind of solutions:

- Eclipse cross compiler;
- Online cross compiler;

3.5.1 Prerequisites

First of all, we need to install:

- Eclipse;
- Node.js (10.15.3 LTS release);
- xpm;

You can download Eclipse from Ubuntu Software or , if you use another OS, on the official Web Site (https://www.eclipse.org/downloads/).

To install Node.js, open a terminal:

```
$sudo apt-get install curl python-software-properties
$curl -sL https://deb.nodesource.com/setup\_10.x | sudo -E bash -
$sudo apt-get install nodejs
```

To test if npm starts:

```
$node -v
$v12.1.0
```

Also, check the npm version:

```
$npm -v
$6.9.0
```

To install xpm, open a terminal:

```
$sudo npm install --global xpm
```

To test if xpm starts:

```
$xpm -v
$v 0.5.0
```

Easy install

For RISC-V, the recommended method to install the latest version of the toolchain is:

```
$xpm install --global @gnu-mcu-eclipse/riscv-none-gcc
```

3.5.2 Eclipse cross compiler

Open Eclipse and follow this path : *Help> Eclipse Marketplace..* .
Here research riscv and download the first search result,if it is not done yet.
Now, you can use it to compile a C program into RISCV machine intructions.

3.5.3 Online cross compiler

Easily,We can also use a online cross compiler from:
https://ascslab.org/research/briscv/simulator/simulator.html
Here, you can translate RISC-V assembly onto machine instructions.
To obtain RISC-V assembly open the directory where you have downloaded riscv-none-gcc. Usually it is:

```
$cd opt/xPacks/@gnu-mcu-eclipse/riscv-none-gcc/8.2.0-2.1.1/.content/bin
```

Here, write:

```
$sudo ./riscv-none-embed-gcc '/home/User/NameCProgram.c' -S
```

Now, you can find NameCProgram.s in opt/xPacks/@gnu-mcu-eclipse/riscv-none-gcc/8.2.0-2.1.1/.content/bin.
Upload it onto the online cross compiler and run it. You can optain on the right the RISC-V machine intructions.

3.5.4 Example 1: Sum

C-code

```c
#include <stdio.h>

int main()
    {
        int a;
        int b;
        int c;

        a=1;
        b=2;
        c=a+b;
        printf ("%d",c);
        return 0;
    }
```

Assembly code

```
         .file        "somma.c"
        .option nopic
        .text
        .align       2
        .globl       main
        .type       main, @function
main:
        addi        sp,sp,-32
        sw         s0,28(sp)
        addi        s0,sp,32
        li         a5,1
        sw         a5,-20(s0)
        li         a5,2
        sw         a5,-24(s0)
        lw         a4,-20(s0)
        lw         a5,-24(s0)
        add         a5,a4,a5
        sw         a5,-28(s0)
        li         a5,0
```

```
        mv        a0,a5
        lw        s0,28(sp)
        addi        sp,sp,32
        jr        ra
        .size        main, .-main
        .ident        "GCC: (GNU MCU Eclipse RISC-V Embedded GCC, 64-bit) 8.2.0"
```

Machine instructions

```
  HEXA
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 80 E7          //jalr ra,0(ra)
00 00 00 B7          //auipc ra,0x0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //jr ra
02 01 01 13          //addi sp,sp,32
01 C1 24 03          //lw s0,28(sp)
00 00 0N AN          //mv a0,a5
00 00 0N AN          //li a5,0
FE F4 22 23          //sw a5,-28(s0)
00 F7 07 B3          //add a5,a4,a5
FE 84 27 83          //lw a5,-24(s0)
FE C4 27 03          //lw a4,-20(s0)
FE F4 24 23          //sw a5,-24(s0)
00 00 0N AN          //li a5,2
FE F4 26 23          //sw a5,-20(s0)
00 00 0N AN          //li a5,1
02 01 04 13          //addi s0,sp,32
00 81 2E 23          //sw s0,28(sp)
FE 01 01 13          //addi sp,sp,-32
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 80 E7          //jalr ra,0(ra)
00 00 00 B7          //auipc ra,0x0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //mv s1,a0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //call main
60 00 01 13          //addi sp,zero,1536
00 00 00 13          //addi zero,zero,0
```

Example 2: Sum-Sub

C-code

```c
#include <stdio.h>

int main()
```

```
    {
        int a;
        int b;
        int c;
        int d;

        a=1;
        b=2;
        scanf("%d",&d);
        if (d<5)
        {c=a+b;}
        else
        {c=a-b;}
        return 0;
    }
```

Assembly code

```
        .file   "sum_sub.c"
        .option nopic
        .text
        .section    .rodata
        .align  2
.LC0:
        .string  "%d"
        .text
        .align  2
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a5,1
        sw      a5,-20(s0)
        li      a5,2
        sw      a5,-24(s0)
        addi    a5,s0,-32
        mv      a1,a5
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    scanf
        lw      a4,-32(s0)
        li      a5,4
        bgt     a4,a5,.L2
        lw      a4,-20(s0)
        lw      a5,-24(s0)
        add     a5,a4,a5
        sw      a5,-28(s0)
        j       .L3
.L2:
        lw      a4,-20(s0)
        lw      a5,-24(s0)
```

```
        sub         a5,a4,a5
        sw          a5,-28(s0)
.L3:
        li          a5,0
        mv          a0,a5
        lw          ra,28(sp)
        lw          s0,24(sp)
        addi          sp,sp,32
        jr          ra
        .size          main, .-main
        .ident          "GCC: (GNU MCU Eclipse RISC-V Embedded GCC, 64-bit) 8.2.0"
```

Machine instructions

```
  HEXA
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 80 E7          //jalr ra,0(ra)
00 00 00 B7          //auipc ra,0x0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //jr ra
02 01 01 13          //addi sp,sp,32
01 81 24 03          //lw s0,24(sp)
01 C1 20 83          //lw ra,28(sp)
00 00 0N AN          //mv a0,a5
00 00 0N AN          //li a5,0
FE F4 22 23          //sw a5,-28(s0)
40 F7 07 B3          //sub a5,a4,a5
FE 84 27 83          //lw a5,-24(s0)
FE C4 27 03          //lw a4,-20(s0)
00 00 0N AN          //j .L3
FE F4 22 23          //sw a5,-28(s0)
00 F7 07 B3          //add a5,a4,a5
FE 84 27 83          //lw a5,-24(s0)
FE C4 27 03          //lw a4,-20(s0)
00 00 0N AN          //bgt a4,a5,.L2
00 00 0N AN          //li a5,4
FE 04 27 03          //lw a4,-32(s0)
00 00 0N AN          //call scanf
00 07 85 13          //addi a0,a5,%lo(.LC0)
00 00 07 B7          //lui a5,%hi(.LC0)
00 00 0N AN          //mv a1,a5
FE 04 07 93          //addi a5,s0,-32
FE F4 24 23          //sw a5,-24(s0)
00 00 0N AN          //li a5,2
FE F4 26 23          //sw a5,-20(s0)
00 00 0N AN          //li a5,1
02 01 04 13          //addi s0,sp,32
00 81 2C 23          //sw s0,24(sp)
```

```
00 11 2E 23          //sw ra,28(sp)
FE 01 01 13          //addi sp,sp,-32
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 80 E7          //jalr ra,0(ra)
00 00 00 B7          //auipc ra,0x0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //mv s1,a0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //call main
60 00 01 13          //addi sp,zero,1536
00 00 00 13          //addi zero,zero,0
```

3.5.5 Example 3: Counter

3.5.6 C-code

```c
#include <stdio.h>

int main()
    {

        int c;
         c=0;
        for (int i=0;i<5;i++)
        {c=c+i;}
        return 0;
    }
```

Assembly code

```
        .file        "sumcycle.c"
        .option nopic
        .text
        .align        2
        .globl        main
        .type        main, @function
main:
        addi        sp,sp,-32
        sw        s0,28(sp)
        addi        s0,sp,32
        sw        zero,-20(s0)
        sw        zero,-24(s0)
        j        .L2
.L3:
        lw        a4,-20(s0)
        lw        a5,-24(s0)
        add        a5,a4,a5
        sw        a5,-20(s0)
        lw        a5,-24(s0)
        addi        a5,a5,1
        sw        a5,-24(s0)
.L2:
        lw        a4,-24(s0)
```

```
        li      a5,4
        ble      a4,a5,.L3
        li      a5,0
        mv      a0,a5
        lw      s0,28(sp)
        addi      sp,sp,32
        jr      ra
        .size      main, .-main
        .ident      "GCC: (GNU MCU Eclipse RISC-V Embedded GCC, 64-bit) 8.2.0"
```

Machine instructions

```
  HEXA
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 80 E7         //jalr ra,0(ra)
00 00 00 B7         //auipc ra,0x0
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 00 13         //addi zero,zero,0
00 00 0N AN         //jr ra
02 01 01 13         //addi sp,sp,32
01 81 24 03         //lw s0,24(sp)
01 C1 20 83         //lw ra,28(sp)
00 00 0N AN         //mv a0,a5
00 00 0N AN         //li a5,0
FE F4 22 23         //sw a5,-28(s0)
40 F7 07 B3         //sub a5,a4,a5
FE 84 27 83         //lw a5,-24(s0)
FE C4 27 03         //lw a4,-20(s0)
00 00 0N AN         //j .L3
FE F4 22 23         //sw a5,-28(s0)
00 F7 07 B3         //add a5,a4,a5
FE 84 27 83         //lw a5,-24(s0)
FE C4 27 03         //lw a4,-20(s0)
00 00 0N AN         //bgt a4,a5,.L2
00 00 0N AN         //li a5,4
FE 04 27 03         //lw a4,-32(s0)
00 00 0N AN         //call scanf
00 07 85 13         //addi a0,a5,%lo(.LC0)
00 00 07 B7         //lui a5,%hi(.LC0)
00 00 0N AN         //mv a1,a5
FE 04 07 93         //addi a5,s0,-32
FE F4 24 23         //sw a5,-24(s0)
00 00 0N AN         //li a5,2
FE F4 26 23         //sw a5,-20(s0)
00 00 0N AN         //li a5,1
02 01 04 13         //addi s0,sp,32
00 81 2C 23         //sw s0,24(sp)
00 11 2E 23         //sw ra,28(sp)
FE 01 01 13         //addi sp,sp,-32
```

```
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 80 E7          //jalr ra,0(ra)
00 00 00 B7          //auipc ra,0x0
00 00 00 13          //addi zero,zero,0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //mv s1,a0
00 00 00 13          //addi zero,zero,0
00 00 0N AN          //call main
60 00 01 13          //addi sp,zero,1536
00 00 00 13          //addi zero,zero,0
```

3.5.7 Architecture Simulation trial & references

We try to simulate using Modelsim one of the many different core architectures proposed on the website https://ascslab.org/research/briscv/index.html, which is based on the same project as the compiler we used, from Boston University .
Keep in mind that this architecture is not RISC-V compliant, but it is intuitive and available for free.
Selecting the *explorer section* they realised a tool where it's possible to select the feature you desire in your design and the application generate a verilog project as an output, selecting from from the projects that you can find already developed by this research group.
In particular, we can select:

- *System Configuration* : where you can select the type of the architecture (number of pipeline stages, in order-out of order execution..) from the available;
- *Processor parameters*: where you can choose the textitData Bit width, the other parameters you have to accept them by default.It is necessary to add a *Default Program* which is in the format *.vmh*;

All the details and the description of the various topologies you can find in the manual at https://ascslab.org/research/briscv/downloads.html.
As an example we tried to simulate the *Five cycle RV32i Processor* reported in Fig: 3.39, inserting as a default program *fibonacci.vmh*, provided in the .zip files from download section.

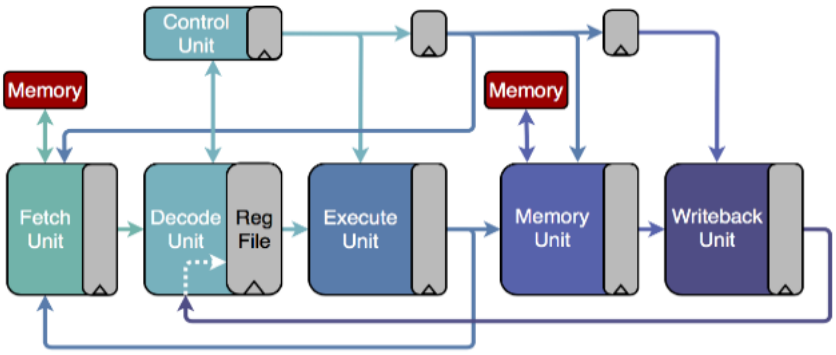


Figure 3.39

After creating the modelsim project, we note that there are some errors in the architecture description in verilog that need to be fixed. For example, the *stall_in* signal, which is a signal present in the *RISC_V_Core.v* file, is not reported as a signal when the core is instantiated as a component in all the higher level entities, such as *RISCV.v* and *RISCV_tb.v*. Hence, the signal must be reported in all the port maps because it leads to errors in the test bench.
Moreover, the stall_in signal is required also for the correct behavior of the test bench, it has to be inserted correctly at the *initial begin*, like in all the examples provided in the non - customized processors architectures.

3.6 Sithography & Bibliography - Special Project

- *Computer Organization and Design - the hardware software interface, RISC-V Edition*, D.A. Patterson, J.L.Hennessy;
- https://riscv.org/ - RISC-V Foundation Web Site;
- https://www.npmjs.com/package/xpm - npm & xpm installation guide;
- https://ascslab.org/research/briscv/simulator/simulator.html - RISC-V Simulator;
- https://ascslab.org/research/briscv/downloads.html-Sahan Bandara, Alan Ehret, Donato Kava and Michel A. Kinsy: "BRISC-V: Open Source Architectural Design Space Exploration Toolbox" Tech-Report 0V1, October 2018;

Chapter 4

Appendix - Scripts

4.1 Lab 1: design and implementation of a digital filter

4.1.1 Filter design and coefficient quantization using Matlab/Octave

```
%%my filter design m
function [bi, bq]=myfir_design(N,nb)
%% function myfir_design(N,nb)
%% N is order of the filter
%% nb is the number of bits
%% bi taps represented as integers

close all;

f_cut_off = 2000; % 2kHz
f_sampling = 10000; % 10kHz

f_nyq = f_sampling/2; %% Nyquist frequency
f0 = f_cut_off/f_nyq; %% normalized cut-off frequency

b=fir1(N, f0); %% get filter coefficients
[h1, w1]=freqz(b); %% get the transfer function of the designed filter

bi=floor(b*2^(nb-1)); %% convert coefficients into nb-bit integers
bq=bi/2^(nb-1); %% convert back coefficients as nb-bit real values
[h2, w2]=freqz(bq); %% get the transfer function of the quantized filter

%% show the transfer functions
plot(w1/pi, 20*log10(abs(h1)));
hold on;
plot(w2/pi, 20*log10(abs(h2)),'r--');
grid on;
xlabel('Normalized frequency');
ylabel('dB');
```

4.1.2 First step: Matlab/Octave pseudo-fixed-point

```
%%my filter m
f=500;  %% first sinewave freq (in band)
f2=4500; %% second sinnewave freq (out band)

N=10; %% filter order
```

```
nb=11; %% s=10000 %% sampling frequency
f1number of bits

T=1/500; %% maximum period
tt=0:1/fs:10*T; %% time samples

x1=sin(2*pi*f1*tt); %% first sinewave
x2=sin(2*pi*f2*tt); %% second sinewave

x=(x1+x2)/2; %% input signal

[bi, bq]=myfir_design(N, nb); %% filter design

y=filter(bq, 1, x); %% apply filter

%% plots
figure
%plot(tt,x1,'--d');
hold on
%plot(tt,x2,'r--s');
plot(tt,x, '--b');
plot(tt, y, 'r--o');

legend('x1', 'x2', 'x', 'y')

%% quantize input and output
xq=floor(x*2^(nb-1));
idx=find(xq==2^(nb-1));
xq(idx)=2^(nb-1)-1;

yq=floor(y*2^(nb-1));
idy=find(yq==2^(nb-1));
yq(idy)=2^(nb-1)-1;

%% save input and output
fp=fopen('samples.txt','w');
fprintf(fp,'%d\n', xq);
fclose(fp);

fp=fopen('resultsm.txt', 'w');
fprintf(fp, '%d\n', yq);
fclose(fp);
```

4.1.3 Second step: fixed-point C model

```
%% c program
#include<stdio.h>
#include<stdlib.h>

#define NT 11 /// number of coeffs
#define NB 11 /// number of bits

const int b[NT]={-1, -13, -26, 65, 281, 407, 281, 65, -26, -13, -1}; /// b array
```

```
//const int a[NT-1]={-147, 52}; /// a array

/// Perform fixed point filtering assming direct form I
///\param x is the new input sample
///\return the new output sample
int myfilter(int x)
{
  static int sx[NT]; /// x shift register
  static int sy[NT-1]; /// y shift register
  static int first_run = 0; /// for cleaning shift registers
  int i; /// index
  int y; /// output sample

  /// clean the buffers
  if (first_run == 0)
  {
    first_run = 1;
    for (i=0; i<NT; i++)
      sx[i] = 0;
    for (i=0; i<NT-1; i++)
      sy[i] = 0;
  }

  /// shift and insert new sample in x shift register
  for (i=NT-1; i>0; i--)
    sx[i] = sx[i-1];
  sx[0] = x;

  /// make the convolution
  /// Moving average part
  y = 0;
  for (i=0; i<NT; i++)
    y += (sx[i]*b[i]) >> (NB-1) ;
  /// Auto regressive part
 //for (i=0; i<NT-1; i++)
    //y -= (sy[i]*a[i]) >> (NB-1);

  /// update the y shift register
  for (i=NT-2; i>0; i--)
    sy[i] = sy[i-1];
  sy[0] = y;

  return y;
}

int main ()
{
  FILE *fp_in;
  FILE *fp_out;

  int x;
  int y;
```

```
  /// check the command line
  ///if (argc != 3)
  ///{
    ///printf("Use: %s <input_file> <output_file>\n", argv[0]);
    ///exit(1);
  ///}

  /// open files
  fp_in = fopen("samples.txt", "r");
  if (fp_in == NULL)
  {
    printf("Error: cannot open %s\n");
    exit(2);
  }
  fp_out = fopen("resultsc.txt", "w");

  /// get samples and apply filter
  fscanf(fp_in, "%d", &x);
  do{
    y = myfilter(x);
    fprintf(fp_out,"%d\n", y);
    fscanf(fp_in, "%d", &x);
  } while (!feof(fp_in));

  fclose(fp_in);
  fclose(fp_out);

  return 0;

}

%% compare m and c
resm=0;
resc=0;

rm=fopen('resultsm.txt','r');
resm=fscanf(rm,'%d\n');
fclose(rm);

rc=fopen('resultsc.txt','r');
resc=fscanf(rc,'%d\n');
fclose(rc);

delta=resm-resc;


average = mean (delta);
plot(delta,'r');
ylabel('Difference Y(Matlab) - Y(C)');
```

4.1.4 Starting architecture development |& simulation

```
%% vhdl register
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY reg_11bit IS
PORT( D_IN : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          D_OUT : OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          CLK,RST_n : IN STD_LOGIC;
          EN_REG: IN STD_LOGIC -- CONNECTED TO V_IN
                );
END ENTITY;

ARCHITECTURE behavior OF reg_11bit IS
SIGNAL temp : STD_LOGIC_VECTOR (10 DOWNTO 0);
BEGIN
PROCESS(CLK,RST_n)
BEGIN
IF RST_n='0' THEN

temp<=(OTHERS=>'0');

        ELSIF CLK'EVENT AND CLK='1' THEN
                IF EN_REG='1' THEN
                        temp<=D_IN;
                END IF;
END IF;

END PROCESS;
D_OUT<=temp;
END behavior;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY mpy_fixedpoint IS
PORT( SAMPLE_IN : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          Bi : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          PRODUCT:OUT STD_LOGIC_VECTOR (21 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE behavior OF mpy_fixedpoint IS

SIGNAL prod_temp: SIGNED (21 DOWNTO 0);
BEGIN
prod_temp<=SIGNED(SAMPLE_IN)*SIGNED(Bi);

PRODUCT<=STD_LOGIC_VECTOR(prod_temp);

END ARCHITECTURE;

%%vhdl adder
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY adder_fixedpoint IS
PORT( A : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          B : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          SUM : OUT STD_LOGIC_VECTOR ( 10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE behavior OF adder_fixedpoint IS

SIGNAL sum_temp: SIGNED (10 DOWNTO 0);

BEGIN


sum_temp<=SIGNED(A)+SIGNED(B);

SUM<=STD_LOGIC_VECTOR(sum_temp);


END ARCHITECTURE;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY datapath IS
PORT( CLK,RST_n: IN STD_LOGIC;
          V_IN : IN STD_LOGIC;
          DATA_IN: IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10 : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT: OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT: IN STD_LOGIC

                );
END ENTITY;

ARCHITECTURE struct OF datapath IS

COMPONENT reg_11bit
PORT( D_IN : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          D_OUT : OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          CLK,RST_n : IN STD_LOGIC;
          EN_REG: IN STD_LOGIC
                );
END COMPONENT;



COMPONENT mpy_fixedpoint
PORT( SAMPLE_IN : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          Bi : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
```

```
          PRODUCT: OUT STD_LOGIC_VECTOR (21 DOWNTO 0)
                );
END COMPONENT;

COMPONENT adder_fixedpoint --DATA IN @24 BITS
PORT( A : IN STD_LOGIC_VECTOR(10 DOWNTO 0); --SIGN HAS TO BE EXTENDED
          B : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          SUM : OUT STD_LOGIC_VECTOR ( 10 DOWNTO 0)
                );
END COMPONENT;


SIGNAL regout0,regout1,regout2,regout3,regout4,regout5,regout6
,regout7,regout8,regout9,regout10: STD_LOGIC_VECTOR (10 DOWNTO 0);

SIGNAL prod_0,prod_1,prod_2,prod_3,prod_4,prod_5,prod_6,
prod_7,prod_8,prod_9,prod_10: STD_LOGIC_VECTOR(21 DOWNTO 0);

SIGNAL prod_0i,prod_1i,prod_2i,prod_3i,prod_4i,prod_5i,prod_6i
,prod_7i,prod_8i,prod_9i,prod_10i: STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL sum0, sum1,sum2,sum3,sum4,sum5,sum6,
sum7,sum8,sum9 : STD_LOGIC_VECTOR(10 DOWNTO 0);



BEGIN

 reg0: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN
 ,D_OUT=>regout0);

 reg1: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout0
 ,D_OUT=>regout1);

 reg2: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout1
 ,D_OUT=>regout2);

 reg3: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout2
 ,D_OUT=>regout3);

 reg4: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout3
 ,D_OUT=>regout4);

 reg5: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout4
 ,D_OUT=>regout5);

 reg6: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout5
 ,D_OUT=>regout6);

 reg7: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout6
 ,D_OUT=>regout7);

 reg8: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout7
```

```
,D_OUT=>regout8);

reg9: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout8
,D_OUT=>regout9);

reg10: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout9
,D_OUT=>regout10);

mpy0: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout0,Bi=>B0,PRODUCT=>prod_0);

mpy1: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout1,Bi=>B1,PRODUCT=>prod_1);

mpy2: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout2,Bi=>B2,PRODUCT=>prod_2);

mpy3: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout3,Bi=>B3,PRODUCT=>prod_3);

mpy4: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout4,Bi=>B4,PRODUCT=>prod_4);

mpy5: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout5,Bi=>B5,PRODUCT=>prod_5);

mpy6: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout6,Bi=>B6,PRODUCT=>prod_6);

mpy7: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout7,Bi=>B7,PRODUCT=>prod_7);

mpy8: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout8,Bi=>B8,PRODUCT=>prod_8);

mpy9: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout9,Bi=>B9,PRODUCT=>prod_9);

mpy10: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout10,Bi=>B10,PRODUCT=>prod_10);

add0: adder_fixedpoint PORT MAP(A=>prod_0i,B=>prod_1i,SUM=>sum0);

add1: adder_fixedpoint PORT MAP(A=>sum0,B=>prod_2i,SUM=>sum1);

add2: adder_fixedpoint PORT MAP(A=>sum1,B=>prod_3i,SUM=>sum2);

add3: adder_fixedpoint PORT MAP(A=>sum2,B=>prod_4i,SUM=>sum3);

add4: adder_fixedpoint PORT MAP(A=>sum3,B=>prod_5i,SUM=>sum4);

add5: adder_fixedpoint PORT MAP(A=>sum4,B=>prod_6i,SUM=>sum5);

add6: adder_fixedpoint PORT MAP(A=>sum5,B=>prod_7i,SUM=>sum6);

add7: adder_fixedpoint PORT MAP(A=>sum6,B=>prod_8i,SUM=>sum7);

add8: adder_fixedpoint PORT MAP(A=>sum7,B=>prod_9i,SUM=>sum8);

add9: adder_fixedpoint PORT MAP(A=>sum8,B=>prod_10i,SUM=>sum9);


prod_0i<=prod_0(20 DOWNTO 10);
prod_1i<=prod_1(20 DOWNTO 10);
```

```
 prod_2i<=prod_2(20 DOWNTO 10);
 prod_3i<=prod_3(20 DOWNTO 10);
 prod_4i<=prod_4(20 DOWNTO 10);
 prod_5i<=prod_5(20 DOWNTO 10);
 prod_6i<=prod_6(20 DOWNTO 10);
 prod_7i<=prod_7(20 DOWNTO 10);
 prod_8i<=prod_8(20 DOWNTO 10);
 prod_9i<=prod_9(20 DOWNTO 10);
 prod_10i<=prod_10(20 DOWNTO 10);

 reg_data_out: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT,
 D_IN=>sum9,D_OUT=>DATA_OUT);



END struct;

%% vhdl control unit
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY control_unit IS
PORT( CLK,RST_n: IN STD_LOGIC;
          VIN : IN STD_LOGIC;
          VOUT: OUT STD_LOGIC;
          DIN: IN STD_LOGIC_VECTOR( 10 DOWNTO 0);
          H0,H1,H2,H3,H4,H5,H6,H7,H8,H9,H10: IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          DOUT: OUT STD_LOGIC_VECTOR( 10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE behavior OF control_unit IS

COMPONENT datapath
PORT( CLK,RST_n: IN STD_LOGIC;
          V_IN : IN STD_LOGIC;
          DATA_IN: IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10 : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT: OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT: IN STD_LOGIC

                );
END COMPONENT;


SIGNAL valid_out:STD_LOGIC;


TYPE state_type IS (idle,elaborate,done);
SIGNAL next_state,p_state : state_type;

BEGIN
dp: datapath PORT MAP (CLK=>CLK,RST_n=>RST_n,V_IN=>VIN,DATA_IN=>DIN,
```

```
B0=>H0,B1=>H1,B2=>H2,B3=>H3,
B4=>H4,B5=>H5,B6=>H6,B7=>H7,B8=>H8,B9=>H9,B10=>H10,
DATA_OUT=>DOUT,V_OUT=>valid_out);

--STARTUS REGISTERS
VOUT<=valid_out;
PROCESS(CLK,RST_n)
BEGIN
IF RST_n='0' THEN
        p_state<=idle;
ELSIF CLK='1' AND CLK'EVENT THEN
        p_state<=next_state;
END IF;
END PROCESS;
--STATE TRANSITIONS
PROCESS(p_state,VIN)
BEGIN
CASE p_state IS

WHEN idle => IF VIN='1' THEN
                        next_state<=elaborate;
                ELSE
                        next_state<=idle;
                END IF;

WHEN elaborate=> IF VIN='0' THEN
                                next_state<=done;
                         ELSE
                                next_state<=elaborate;
                        END IF;
WHEN done=> next_state<=idle;
WHEN OTHERS=> next_state<=idle;
END CASE;
END PROCESS;

--OUTPUTS
PROCESS(p_state)
BEGIN
valid_out<='0';

CASE p_state IS
WHEN idle=>valid_out<='0';

WHEN elaborate|done=>valid_out<='1';

WHEN OTHERS=>valid_out<='0';
END CASE;
END PROCESS;


END behavior;

%% clock generator
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity clk_gen is
  port (
    END_SIM : in  std_logic;
    CLK     : out std_logic;
    RST_n   : out std_logic);
end clk_gen;

architecture beh of clk_gen is

  constant Ts : time := 10 ns;

  signal CLK_i : std_logic;

begin  -- beh

  process
  begin  -- process
    if (CLK_i = 'U') then
      CLK_i <= '0';
    else
      CLK_i <= not(CLK_i);
    end if;
    wait for Ts/2;
  end process;

  CLK <= CLK_i and not(END_SIM);

  process
  begin  -- process
    RST_n <= '0';
    wait for 3*Ts/2;
    RST_n <= '1';
    wait;
  end process;

end beh;

%% vhdl data maker
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;

library std;
use std.textio.all;

entity data_maker is
  port (
    CLK     : in  std_logic;
```

```
    RST_n    : in  std_logic;
    VOUT     : out std_logic;
    DOUT     : out std_logic_vector(10 downto 0);
    H0       : out std_logic_vector(10 downto 0);
    H1       : out std_logic_vector(10 downto 0);
    H2       : out std_logic_vector(10 downto 0);
    H3       : out std_logic_vector(10 downto 0);
        H4       : out std_logic_vector(10 downto 0);
    H5       : out std_logic_vector(10 downto 0);
    H6       : out std_logic_vector(10 downto 0);
    H7       : out std_logic_vector(10 downto 0);
        H8       : out std_logic_vector(10 downto 0);
    H9       : out std_logic_vector(10 downto 0);
    H10       : out std_logic_vector(10 downto 0);
    END_SIM : out std_logic);
end data_maker;

architecture beh of data_maker is

  constant tco : time := 1 ns;

  signal sEndSim : std_logic;
  signal END_SIM_i : std_logic_vector(0 to 10);

begin  -- beh

  H0 <= conv_std_logic_vector(-1,11);
  H1 <= conv_std_logic_vector(-13,11);
  H2 <= conv_std_logic_vector(-26,11);
  H3 <= conv_std_logic_vector(65,11);
  H4 <= conv_std_logic_vector(281,11);
  H5 <= conv_std_logic_vector(407,11);
  H6 <= conv_std_logic_vector(281,11);
  H7 <= conv_std_logic_vector(65,11);
  H8 <= conv_std_logic_vector(-26,11);
  H9 <= conv_std_logic_vector(-13,11);
  H10 <= conv_std_logic_vector(-1,11);

  process (CLK, RST_n)
    file fp_in : text open READ_MODE is "./samples.txt";
    variable line_in : line;
    variable x : integer;
  begin  -- process
    if RST_n = '0' then                 -- asynchronous reset (active low)
      DOUT <= (others => '0') after tco;
      VOUT <= '0' after tco;
      sEndSim <= '0' after tco;
    elsif CLK'event and CLK = '1' then  -- rising clock edge
      if not endfile(fp_in) then
        readline(fp_in, line_in);
        read(line_in, x);
        DOUT <= conv_std_logic_vector(x, 11) after tco;
        VOUT <= '1' after tco;
```

```
        sEndSim <= '0' after tco;
      else
        VOUT <= '0' after tco;
        sEndSim <= '1' after tco;
      end if;
    end if;
  end process;

  process (CLK, RST_n)
  begin  -- process
    if RST_n = '0' then                 -- asynchronous reset (active low)
      END_SIM_i <= (others => '0') after tco;
    elsif CLK'event and CLK = '1' then  -- rising clock edge
      END_SIM_i(0) <= sEndSim after tco;
      END_SIM_i(1 to 10) <= END_SIM_i(0 to 9) after tco;
    end if;
  end process;

  END_SIM <= END_SIM_i(10);

end beh;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;

library std;
use std.textio.all;

entity data_sink is
  port (
    CLK   : in std_logic;
    RST_n : in std_logic;
    VIN   : in std_logic;
    DIN   : in std_logic_vector(10 downto 0));
end data_sink;

architecture beh of data_sink is

begin  -- beh

  process (CLK, RST_n)
    file res_fp : text open WRITE_MODE is "./results.txt";
    variable line_out : line;
  begin  -- process
    if RST_n = '0' then                 -- asynchronous reset (active low)
      null;
    elsif CLK'event and CLK = '1' then  -- rising clock edge
      if (VIN = '1') then
        write(line_out, conv_integer(signed(DIN)));
        writeline(res_fp, line_out);
      end if;
```

```
    end if;
  end process;

end beh;

%% verilog tb
```

4.1.5 Logic synthesis

General Synopsys Script:

```
analyze -f vhdl -lib WORK ../src/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/mpy_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/reg_11bit.vhd
analyze -f vhdl -lib WORK ../src/datapath.vhd
analyze -f vhdl -lib WORK ../src/control_unit.vhd
set power_preserve_rtl_hier_names true
elaborate control_unit -arch behavior -lib WORK
create_clock -name MY_CLK -period 3.72 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set load \$OLOAD [all_outputs]
compile
report_timing > report_timing_tck3_72
report_area > report_area_tck3_72
ungroup -all -flatten
change_names -hierarchy -rules verilog
write_sdf ../netlist/myfir_3_72.sdf
write -f verilog -hierarchy -output ../netlist/myfir_3_72.v
write_sdc ../netlist/myfir_3_72.sdc
quit
```

Verilog Test Bench for power evaluation:

```
//tb_pw/tb_fir
//`timescale 1ns



module tb_fir ();


   wire CLK_i;
   wire RST_n_i;
   wire [10:0] DIN_i;
   wire VIN_i;
   wire [10:0] H0_i;
   wire [10:0] H1_i;
   wire [10:0] H2_i;
```

```
 wire [10:0] H3_i;
 wire [10:0] H4_i;
 wire [10:0] H5_i;
 wire [10:0] H6_i;
 wire [10:0] H7_i;
 wire [10:0] H8_i;
 wire [10:0] H9_i;
 wire [10:0] H10_i;
 wire [10:0] DOUT_i;
 wire VOUT_i;
 wire END_SIM_i;

 clk_gen CG(.END_SIM(END_SIM_i),
              .CLK(CLK_i),
            .RST_n(RST_n_i));

 data_maker SM(.CLK(CLK_i),
               .RST_n(RST_n_i),
               .VOUT(VIN_i),
               .DOUT(DIN_i),
                .H0(H0_i),
           .H1(H1_i),
           .H2(H2_i),
           .H3(H3_i),
               .H4(H4_i),
           .H5(H5_i),
           .H6(H6_i),
           .H7(H7_i),
               .H8(H8_i),
           .H9(H9_i),
           .H10(H10_i),
               .END_SIM(END_SIM_i));

control_unit CU(.CLK(CLK_i),
           .RST_n(RST_n_i),
           .DIN(DIN_i),
        .VIN(VIN_i),
           .H0(H0_i),
           .H1(H1_i),
           .H2(H2_i),
           .H3(H3_i),
               .H4(H4_i),
           .H5(H5_i),
           .H6(H6_i),
           .H7(H7_i),
               .H8(H8_i),
           .H9(H9_i),
           .H10(H10_i),
           .DOUT(DOUT_i),
       .VOUT(VOUT_i));

 data_sink DS(.CLK(CLK_i),
              .RST_n(RST_n_i),
```

```
                .VIN(VOUT_i),
                .DIN(DOUT_i));

initial begin
$read_lib_saif("../saif/NangateOpenCellLibrary.saif");
$set_gate_level_monitoring("on");
$set_toggle_region(CU);
$toggle_start;
end

always @ ( END_SIM_i ) begin
if (END_SIM_i) begin
$toggle_stop;
$toggle_report("../saif/myfir_back.saif", 1.0e-9, "tb_fir.CU");
end
end

endmodule
```

Power Script(first and second part with Synopsys:

```
read_file NangateOpenCellLibrary_typical_ecsm_nowlm.db
lib2saif -out ../saif/NangateOpenCellLibrary.saif NangateOpenCellLibrary

read_verilog -netlist ../netlist/myfir_10.v
read_saif -input ../saif/myfir_back.saif -instance tb_fir/CU -unit ns -scale 1
create_clock -name MY_CLK CLK
report_power > report_power_10
```

Power Script with Modelsim:

```
vcom -93 -work ./work ../tb_pw/clk_gen.vhd
vcom -93 -work ./work ../tb_pw/data_maker_new.vhd
vcom -93 -work ./work ../tb_pw/data_sink.vhd
vlog -work ./work ../netlist/myfir_10.v
vlog -work ./work ../tb_pw/tb_fir.v
vsim -L /software/dk/nangate45/verilog/msim6.2g work.tb_fir
vsim -L /software/dk/nangate45/verilog/msim6.2g -sdftyp /tb_fir/CU=../netlist/myfi
vsim -L /software/dk/nangate45/verilog/msim6.2g -sdftyp /tb_fir/CU=../netlist/myfi
```

4.1.6 Advanced architecture development - L-Unfolding and piplined FIR

Control Unit

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY control_unit IS
PORT( CLK,RST_n: IN STD_LOGIC;
          VIN : IN STD_LOGIC;
          VOUT: OUT STD_LOGIC;
          DIN0: IN STD_LOGIC_VECTOR( 10 DOWNTO 0);
```

```
          DIN1: IN STD_LOGIC_VECTOR( 10 DOWNTO 0);
          DIN2: IN STD_LOGIC_VECTOR( 10 DOWNTO 0);
          H0,H1,H2,H3,H4,H5,H6,H7,H8,H9,H10: IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          DOUT0: OUT STD_LOGIC_VECTOR( 10 DOWNTO 0);
          DOUT1: OUT STD_LOGIC_VECTOR( 10 DOWNTO 0);
          DOUT2: OUT STD_LOGIC_VECTOR( 10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE behavior OF control_unit IS

COMPONENT datapathPipeUnfolding IS
PORT( CLK,RST_n:                IN STD_LOGIC;
          V_IN :                      IN STD_LOGIC;
          DATA_IN0:                 IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN1:                 IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN2:                 IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5:IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B6,B7,B8,B9,B10:  IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT0:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT1:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT2:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT:                      IN STD_LOGIC
        );
END COMPONENT;


SIGNAL valid_out:STD_LOGIC;


TYPE state_type IS (idle,elaborate,done);
SIGNAL next_state,p_state : state_type;

BEGIN
dp: datapathPipeUnfolding PORT MAP (CLK=>CLK,RST_n=>RST_n,V_IN=>VIN,DATA_IN0=>DIN0,
DATA_IN1=>DIN1,DATA_IN2=>DIN2,                                  B0=>H0,B1=>H1,B2=>H
B5=>H5,B6=>H6,B7=>H7,B8=>H8,B9=>H9,B10=>H10,                                DATA_OU
DATA_OUT2=>DOUT2,V_OUT=>valid_out);

--STARTUS REGISTERS
VOUT<=valid_out;
PROCESS(CLK,RST_n)
BEGIN
IF RST_n='0' THEN
        p_state<=idle;
ELSIF CLK='1' AND CLK'EVENT THEN
        p_state<=next_state;
END IF;
END PROCESS;
--STATE TRANSITIONS
PROCESS(p_state,VIN)
BEGIN
CASE p_state IS
```

```
WHEN idle => IF VIN='1' THEN
                            next_state<=elaborate;
                        ELSE
                            next_state<=idle;
                        END IF;

WHEN elaborate=> IF VIN='0' THEN
                                    next_state<=done;
                                ELSE
                                    next_state<=elaborate;
                                END IF;
WHEN done=> next_state<=idle;
WHEN OTHERS=> next_state<=idle;
END CASE;
END PROCESS;

--OUTPUTS
PROCESS(p_state)
BEGIN
valid_out<='0';

CASE p_state IS
WHEN idle=>valid_out<='0';

WHEN elaborate|done=>valid_out<='1';

WHEN OTHERS=>valid_out<='0';
END CASE;
END PROCESS;


END behavior;
```

Datapath

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY datapathPipeUnfolding IS
PORT( CLK,RST_n:                IN STD_LOGIC;
          V_IN :                IN STD_LOGIC;
          DATA_IN0:                IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN1:                IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_IN2:                IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5:       IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B6,B7,B8,B9,B10:         IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT0:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT1:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT2:                OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT:                        IN STD_LOGIC
        );
END ENTITY;
```

```
ARCHITECTURE struct OF datapathPipeUnfolding IS

COMPONENT reg_11bit
PORT( D_IN:          IN STD_LOGIC_VECTOR (10 DOWNTO 0);
      D_OUT:           OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
      CLK,RST_n: IN STD_LOGIC;
      EN_REG:           IN STD_LOGIC
    );
END COMPONENT;


COMPONENT mpy_fixedpoint
PORT( SAMPLE_IN: IN STD_LOGIC_VECTOR(10 DOWNTO 0);
      Bi:                    IN STD_LOGIC_VECTOR (10 DOWNTO 0);
      PRODUCT:   OUT STD_LOGIC_VECTOR (21 DOWNTO 0)
    );
END COMPONENT;

COMPONENT adder_fixedpoint --DATA IN @24 BITS
PORT( A :         IN STD_LOGIC_VECTOR(10 DOWNTO 0); --SIGN HAS TO BE EXTENDED
      B :          IN STD_LOGIC_VECTOR (10 DOWNTO 0);
      SUM : OUT STD_LOGIC_VECTOR ( 10 DOWNTO 0)
    );
END COMPONENT;

SIGNAL regout00,regout01,regout02,regout03,regout04,regout05
,regout06,regout07,regout08,regout09,regout010: STD_LOGIC_VECTOR (10 DOWNTO 0);

SIGNAL regout10,regout11,regout12,regout13,regout14,regout15
,regout16,regout17,regout18,regout19,regout110: STD_LOGIC_VECTOR (10 DOWNTO 0);

SIGNAL regout20,regout21,regout22,regout23,regout24,regout25
,regout26,regout27,regout28,regout29,regout210: STD_LOGIC_VECTOR (10 DOWNTO 0);

SIGNAL prod_00,prod_01,prod_02,prod_03,prod_04,prod_05
,prod_06,prod_07,prod_08,prod_09,prod_010: STD_LOGIC_VECTOR(21 DOWNTO 0);

SIGNAL prod_10,prod_11,prod_12,prod_13,prod_14,prod_15
,prod_16,prod_17,prod_18,prod_19,prod_110: STD_LOGIC_VECTOR(21 DOWNTO 0);

SIGNAL prod_20,prod_21,prod_22,prod_23,prod_24,prod_25
,prod_26,prod_27,prod_28,prod_29,prod_210: STD_LOGIC_VECTOR(21 DOWNTO 0);

SIGNAL prod_00i,prod_01i,prod_02i,prod_03i,prod_04i,prod_05i
,prod_06i,prod_07i,prod_08i,prod_09i,prod_010i: STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL prod_10i,prod_11i,prod_12i,prod_13i,prod_14i,prod_15i
,prod_16i,prod_17i,prod_18i,prod_19i,prod_110i: STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL prod_20i,prod_21i,prod_22i,prod_23i,prod_24i,prod_25i
,prod_26i,prod_27i,prod_28i,prod_29i,prod_210i: STD_LOGIC_VECTOR(10 DOWNTO 0);
```

```
SIGNAL sum00, sum01,sum02,sum03,sum04,sum05
,sum06,sum07,sum08,sum09 : STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL sum10, sum11,sum12,sum13,sum14,sum15
,sum16,sum17,sum18,sum19 : STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL sum20, sum21,sum22,sum23,sum24,sum25
,sum26,sum27,sum28,sum29 : STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL prod_00P,prod_01P,prod_02P,prod_03P,prod_04P,prod_05P
,prod_06P,prod_07P,prod_08P,prod_09P,prod_010P: STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL prod_10P,prod_11P,prod_12P,prod_13P,prod_14P,prod_15P
,prod_16P,prod_17P,prod_18P,prod_19P,prod_110P: STD_LOGIC_VECTOR(10 DOWNTO 0);

SIGNAL prod_20P,prod_21P,prod_22P,prod_23P,prod_24P,prod_25P
,prod_26P,prod_27P,prod_28P,prod_29P,prod_210P: STD_LOGIC_VECTOR(10 DOWNTO 0);


BEGIN
 --------------------INPUT REGISTERS-------------------------

 reg00: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN0
 ,D_OUT=>regout00);

 reg01: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout20
 ,D_OUT=>regout01);

 reg02: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout10
 ,D_OUT=>regout02);

 reg03: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout00
 ,D_OUT=>regout03);

 reg04: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout23
 ,D_OUT=>regout04);

 reg05: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout13
 ,D_OUT=>regout05);

 reg06: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout03
 ,D_OUT=>regout06);

 reg07: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout26
 ,D_OUT=>regout07);

 reg08: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout16
 ,D_OUT=>regout08);

 reg09: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout06
 ,D_OUT=>regout09);

 reg010: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout29
```

```
,D_OUT=>regout010);

reg10: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN1
,D_OUT=>regout10);

reg11: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN0
,D_OUT=>regout11);

reg12: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout20
,D_OUT=>regout12);
reg13: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout10
,D_OUT=>regout13);

reg14: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout00
,D_OUT=>regout14);

reg15: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout23
,D_OUT=>regout15);

reg16: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout13
,D_OUT=>regout16);

reg17: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout03
,D_OUT=>regout17);

reg18: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout26
,D_OUT=>regout18);

reg19: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout16
,D_OUT=>regout19);

reg110: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout06
,D_OUT=>regout110);

reg20: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN2
,D_OUT=>regout20);

reg21: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN1
,D_OUT=>regout21);

reg22: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>DATA_IN0
,D_OUT=>regout22);

reg23: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout20
,D_OUT=>regout23);

reg24: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout10
,D_OUT=>regout24);

reg25: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout00
,D_OUT=>regout25);

reg26: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout23
```

```
,D_OUT=>regout26);

reg27: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout13
,D_OUT=>regout27);

reg28: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout03
,D_OUT=>regout28);

reg29: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout26
,D_OUT=>regout29);

reg210: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,D_IN=>regout16
,D_OUT=>regout210);

 --------------------MPYS-------------------------

mpy00: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout00,Bi=>B0,PRODUCT=>prod_00);
mpy01: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout01,Bi=>B1,PRODUCT=>prod_01);
mpy02: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout02,Bi=>B2,PRODUCT=>prod_02);
mpy03: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout03,Bi=>B3,PRODUCT=>prod_03);
mpy04: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout04,Bi=>B4,PRODUCT=>prod_04);
mpy05: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout05,Bi=>B5,PRODUCT=>prod_05);
mpy06: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout06,Bi=>B6,PRODUCT=>prod_06);
mpy07: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout07,Bi=>B7,PRODUCT=>prod_07);
mpy08: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout08,Bi=>B8,PRODUCT=>prod_08);
mpy09: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout09,Bi=>B9,PRODUCT=>prod_09);
mpy010: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout010,Bi=>B10,PRODUCT=>prod_010);

mpy10: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout10,Bi=>B0,PRODUCT=>prod_10);
mpy11: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout11,Bi=>B1,PRODUCT=>prod_11);
mpy12: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout12,Bi=>B2,PRODUCT=>prod_12);
mpy13: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout13,Bi=>B3,PRODUCT=>prod_13);
mpy14: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout14,Bi=>B4,PRODUCT=>prod_14);
mpy15: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout15,Bi=>B5,PRODUCT=>prod_15);
mpy16: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout16,Bi=>B6,PRODUCT=>prod_16);
mpy17: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout17,Bi=>B7,PRODUCT=>prod_17);
mpy18: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout18,Bi=>B8,PRODUCT=>prod_18);
mpy19: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout19,Bi=>B9,PRODUCT=>prod_19);
mpy110: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout110,Bi=>B10,PRODUCT=>prod_110);

mpy20: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout20,Bi=>B0,PRODUCT=>prod_20);
mpy21: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout21,Bi=>B1,PRODUCT=>prod_21);
mpy22: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout22,Bi=>B2,PRODUCT=>prod_22);
mpy23: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout23,Bi=>B3,PRODUCT=>prod_23);
mpy24: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout24,Bi=>B4,PRODUCT=>prod_24);
mpy25: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout25,Bi=>B5,PRODUCT=>prod_25);
mpy26: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout26,Bi=>B6,PRODUCT=>prod_26);
mpy27: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout27,Bi=>B7,PRODUCT=>prod_27);
mpy28: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout28,Bi=>B8,PRODUCT=>prod_28);
mpy29: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout29,Bi=>B9,PRODUCT=>prod_29);
mpy210: mpy_fixedpoint PORT MAP(SAMPLE_IN=>regout210,Bi=>B10,PRODUCT=>prod_210);

--------------------TRUNCATION-------------------------
```

```
prod_00i<=prod_00(20 DOWNTO 10);
prod_01i<=prod_01(20 DOWNTO 10);
prod_02i<=prod_02(20 DOWNTO 10);
prod_03i<=prod_03(20 DOWNTO 10);
prod_04i<=prod_04(20 DOWNTO 10);
prod_05i<=prod_05(20 DOWNTO 10);
prod_06i<=prod_06(20 DOWNTO 10);
prod_07i<=prod_07(20 DOWNTO 10);
prod_08i<=prod_08(20 DOWNTO 10);
prod_09i<=prod_09(20 DOWNTO 10);
prod_010i<=prod_010(20 DOWNTO 10);

prod_10i<=prod_10(20 DOWNTO 10);
prod_11i<=prod_11(20 DOWNTO 10);
prod_12i<=prod_12(20 DOWNTO 10);
prod_13i<=prod_13(20 DOWNTO 10);
prod_14i<=prod_14(20 DOWNTO 10);
prod_15i<=prod_15(20 DOWNTO 10);
prod_16i<=prod_16(20 DOWNTO 10);
prod_17i<=prod_17(20 DOWNTO 10);
prod_18i<=prod_18(20 DOWNTO 10);
prod_19i<=prod_19(20 DOWNTO 10);
prod_110i<=prod_110(20 DOWNTO 10);

prod_20i<=prod_20(20 DOWNTO 10);
prod_21i<=prod_21(20 DOWNTO 10);
prod_22i<=prod_22(20 DOWNTO 10);
prod_23i<=prod_23(20 DOWNTO 10);
prod_24i<=prod_24(20 DOWNTO 10);
prod_25i<=prod_25(20 DOWNTO 10);
prod_26i<=prod_26(20 DOWNTO 10);
prod_27i<=prod_27(20 DOWNTO 10);
prod_28i<=prod_28(20 DOWNTO 10);
prod_29i<=prod_29(20 DOWNTO 10);
prod_210i<=prod_210(20 DOWNTO 10);


--------------------PIPE REGISTERS-------------------------

reg00P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_00i,D_OUT=>prod_00P);

reg01P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_01i,D_OUT=>prod_01P);

reg02P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_02i,D_OUT=>prod_02P);

reg03P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_03i,D_OUT=>prod_03P);

reg04P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
```

```
 ,D_IN=>prod_04i,D_OUT=>prod_04P);

reg05P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_05i,D_OUT=>prod_05P);

reg06P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_06i,D_OUT=>prod_06P);

reg07P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_07i,D_OUT=>prod_07P);

reg08P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_08i,D_OUT=>prod_08P);

reg09P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_09i,D_OUT=>prod_09P);

reg010P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_010i,D_OUT=>prod_010P);

reg10P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_10i,D_OUT=>prod_10P);

reg11P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_11i,D_OUT=>prod_11P);

reg12P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_12i,D_OUT=>prod_12P);

reg13P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_13i,D_OUT=>prod_13P);

reg14P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_14i,D_OUT=>prod_14P);

reg15P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_15i,D_OUT=>prod_15P);

reg16P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_16i,D_OUT=>prod_16P);

reg17P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_17i,D_OUT=>prod_17P);

reg18P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_18i,D_OUT=>prod_18P);

reg19P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_19i,D_OUT=>prod_19P);

reg110P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_110i,D_OUT=>prod_110P);
```

```
reg20P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_20i,D_OUT=>prod_20P);

reg21P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_21i,D_OUT=>prod_21P);

reg22P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_22i,D_OUT=>prod_22P);

reg23P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_23i,D_OUT=>prod_23P);

reg24P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_24i,D_OUT=>prod_24P);

reg25P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_25i,D_OUT=>prod_25P);

reg26P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_26i,D_OUT=>prod_26P);

reg27P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_27i,D_OUT=>prod_27P);

reg28P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_28i,D_OUT=>prod_28P);

reg29P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_29i,D_OUT=>prod_29P);

reg210P: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>'1'
,D_IN=>prod_210i,D_OUT=>prod_210P);

--------------------ADDERS-------------------------

--add00: adder_fixedpoint PORT MAP(A=>prod_00i,B=>prod_01i,SUM=>sum00);
--add01: adder_fixedpoint PORT MAP(A=>sum00,B=>prod_02i,SUM=>sum01);
--add02: adder_fixedpoint PORT MAP(A=>sum01,B=>prod_03i,SUM=>sum02);
--add03: adder_fixedpoint PORT MAP(A=>sum02,B=>prod_04i,SUM=>sum03);
--add04: adder_fixedpoint PORT MAP(A=>sum03,B=>prod_05i,SUM=>sum04);
--add05: adder_fixedpoint PORT MAP(A=>sum04,B=>prod_06i,SUM=>sum05);
--add06: adder_fixedpoint PORT MAP(A=>sum05,B=>prod_07i,SUM=>sum06);
--add07: adder_fixedpoint PORT MAP(A=>sum06,B=>prod_08i,SUM=>sum07);
--add08: adder_fixedpoint PORT MAP(A=>sum07,B=>prod_09i,SUM=>sum08);
--add09: adder_fixedpoint PORT MAP(A=>sum08,B=>prod_010i,SUM=>sum09);

--add10: adder_fixedpoint PORT MAP(A=>prod_10i,B=>prod_11i,SUM=>sum10);
--add11: adder_fixedpoint PORT MAP(A=>sum10,B=>prod_12i,SUM=>sum11);
--add12: adder_fixedpoint PORT MAP(A=>sum11,B=>prod_13i,SUM=>sum12);
--add13: adder_fixedpoint PORT MAP(A=>sum12,B=>prod_14i,SUM=>sum13);
--add14: adder_fixedpoint PORT MAP(A=>sum13,B=>prod_15i,SUM=>sum14);
--add15: adder_fixedpoint PORT MAP(A=>sum14,B=>prod_16i,SUM=>sum15);
--add16: adder_fixedpoint PORT MAP(A=>sum15,B=>prod_17i,SUM=>sum16);
```

```
--add17: adder_fixedpoint PORT MAP(A=>sum16,B=>prod_18i,SUM=>sum17);
--add18: adder_fixedpoint PORT MAP(A=>sum17,B=>prod_19i,SUM=>sum18);
--add19: adder_fixedpoint PORT MAP(A=>sum18,B=>prod_110i,SUM=>sum19);


--add20: adder_fixedpoint PORT MAP(A=>prod_20i,B=>prod_21i,SUM=>sum20);
--add21: adder_fixedpoint PORT MAP(A=>sum20,B=>prod_22i,SUM=>sum21);
--add22: adder_fixedpoint PORT MAP(A=>sum21,B=>prod_23i,SUM=>sum22);
--add23: adder_fixedpoint PORT MAP(A=>sum22,B=>prod_24i,SUM=>sum23);
--add24: adder_fixedpoint PORT MAP(A=>sum23,B=>prod_25i,SUM=>sum24);
--add25: adder_fixedpoint PORT MAP(A=>sum24,B=>prod_26i,SUM=>sum25);
--add26: adder_fixedpoint PORT MAP(A=>sum25,B=>prod_27i,SUM=>sum26);
--add27: adder_fixedpoint PORT MAP(A=>sum26,B=>prod_28i,SUM=>sum27);
--add28: adder_fixedpoint PORT MAP(A=>sum27,B=>prod_29i,SUM=>sum28);
--add29: adder_fixedpoint PORT MAP(A=>sum28,B=>prod_210i,SUM=>sum29);

add00: adder_fixedpoint PORT MAP(A=>prod_00P,B=>prod_01P,SUM=>sum00);
add01: adder_fixedpoint PORT MAP(A=>sum00,B=>prod_02P,SUM=>sum01);
add02: adder_fixedpoint PORT MAP(A=>sum01,B=>prod_03P,SUM=>sum02);
add03: adder_fixedpoint PORT MAP(A=>sum02,B=>prod_04P,SUM=>sum03);
add04: adder_fixedpoint PORT MAP(A=>sum03,B=>prod_05P,SUM=>sum04);
add05: adder_fixedpoint PORT MAP(A=>sum04,B=>prod_06P,SUM=>sum05);
add06: adder_fixedpoint PORT MAP(A=>sum05,B=>prod_07P,SUM=>sum06);
add07: adder_fixedpoint PORT MAP(A=>sum06,B=>prod_08P,SUM=>sum07);
add08: adder_fixedpoint PORT MAP(A=>sum07,B=>prod_09P,SUM=>sum08);
add09: adder_fixedpoint PORT MAP(A=>sum08,B=>prod_010P,SUM=>sum09);

add10: adder_fixedpoint PORT MAP(A=>prod_10P,B=>prod_11P,SUM=>sum10);
add11: adder_fixedpoint PORT MAP(A=>sum10,B=>prod_12P,SUM=>sum11);
add12: adder_fixedpoint PORT MAP(A=>sum11,B=>prod_13P,SUM=>sum12);
add13: adder_fixedpoint PORT MAP(A=>sum12,B=>prod_14P,SUM=>sum13);
add14: adder_fixedpoint PORT MAP(A=>sum13,B=>prod_15P,SUM=>sum14);
add15: adder_fixedpoint PORT MAP(A=>sum14,B=>prod_16P,SUM=>sum15);
add16: adder_fixedpoint PORT MAP(A=>sum15,B=>prod_17P,SUM=>sum16);
add17: adder_fixedpoint PORT MAP(A=>sum16,B=>prod_18P,SUM=>sum17);
add18: adder_fixedpoint PORT MAP(A=>sum17,B=>prod_19P,SUM=>sum18);
add19: adder_fixedpoint PORT MAP(A=>sum18,B=>prod_110P,SUM=>sum19);

add20: adder_fixedpoint PORT MAP(A=>prod_20P,B=>prod_21P,SUM=>sum20);
add21: adder_fixedpoint PORT MAP(A=>sum20,B=>prod_22P,SUM=>sum21);
add22: adder_fixedpoint PORT MAP(A=>sum21,B=>prod_23P,SUM=>sum22);
add23: adder_fixedpoint PORT MAP(A=>sum22,B=>prod_24P,SUM=>sum23);
add24: adder_fixedpoint PORT MAP(A=>sum23,B=>prod_25P,SUM=>sum24);
add25: adder_fixedpoint PORT MAP(A=>sum24,B=>prod_26P,SUM=>sum25);
add26: adder_fixedpoint PORT MAP(A=>sum25,B=>prod_27P,SUM=>sum26);
add27: adder_fixedpoint PORT MAP(A=>sum26,B=>prod_28P,SUM=>sum27);
add28: adder_fixedpoint PORT MAP(A=>sum27,B=>prod_29P,SUM=>sum28);
add29: adder_fixedpoint PORT MAP(A=>sum28,B=>prod_210P,SUM=>sum29);

--------------------OUTPUT REGISTERS-------------------------

reg_data_out0: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT,
D_IN=>sum09,D_OUT=>DATA_OUT0);
```

```
 reg_data_out1: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT,
 D_IN=>sum19,D_OUT=>DATA_OUT1);

 reg_data_out2: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT,
 D_IN=>sum29,D_OUT=>DATA_OUT2);

END struct;
```

4.2 Lab 2: Digital arithmetic

4.3 Ultra Optimizzation Script

Ultra Optimizzation script with Datapath:

```
set_ultra_optimization true
analyze -f vhdl -lib WORK ../src/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/mpy_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/reg_11bit.vhd
analyze -f vhdl -lib WORK ../src/reg_22bit.vhd
analyze -f vhdl -lib WORK ../src/datapath.vhd
analyze -f vhdl -lib WORK ../src/control_unit.vhd
set power_preserve_rtl_hier_names true
elaborate control_unit -arch behavior -lib WORK
create_clock -name MY_CLK -period 14.88 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set load $OLOAD [all_outputs]


compile_ultra
set_dont_touch *_in_reg
set_dont_touch *_out_reg
optimize_registers

report_timing > report_timing_ultra_66
```

Ultra Optimizzation script without Datapath:

```
set_ultra_optimization true
analyze -f vhdl -lib WORK ../src_2/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src_2/mpy_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src_2/reg_11bit.vhd
analyze -f vhdl -lib WORK ../src_2/reg_22bit.vhd

analyze -f vhdl -lib WORK ../src_2/control_unit.vhd
set power_preserve_rtl_hier_names true

elaborate control_unit -arch behavior -lib WORK
```

```
create_clock -name MY_CLK -period 14.88 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set load $OLOAD [all_outputs]
ungroup -all -flatten

compile_ultra
set_dont_touch *_in_reg
set_dont_touch *_out_reg
optimize_registers
```

4.3.1 Version 1

control unit

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY control_unit IS
PORT( CLK,RST_n: IN STD_LOGIC;
          VIN : IN STD_LOGIC;
          VOUT: OUT STD_LOGIC;
          DIN: IN STD_LOGIC_VECTOR( 10 DOWNTO 0);
          H0,H1,H2,H3,H4,H5,H6,H7,H8,H9,H10: IN STD_LOGIC_VECTOR(10 DOWNTO 0);
          DOUT: OUT STD_LOGIC_VECTOR( 10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE behavior OF control_unit IS

COMPONENT datapath IS
PORT( CLK,RST_n: IN STD_LOGIC;
          V_IN : IN STD_LOGIC;
          DATA_IN: IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10 : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT: OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT: IN STD_LOGIC

                );
END COMPONENT;


SIGNAL valid_out:STD_LOGIC;


TYPE state_type IS (idle,elaborate,done);
SIGNAL next_state,p_state : state_type;

BEGIN
dp: datapath PORT MAP (CLK=>CLK,RST_n=>RST_n,V_IN=>VIN,DATA_IN=>DIN,         B0=>H0,
```

```
--STARTUS REGISTERS
VOUT<=valid_out;
PROCESS(CLK,RST_n)
BEGIN
IF RST_n='0' THEN
        p_state<=idle;
ELSIF CLK='1' AND CLK'EVENT THEN
        p_state<=next_state;
END IF;
END PROCESS;
--STATE TRANSITIONS
PROCESS(p_state,VIN)
BEGIN
CASE p_state IS

WHEN idle => IF VIN='1' THEN
                                next_state<=elaborate;
                        ELSE
                                next_state<=idle;
                        END IF;

WHEN elaborate=> IF VIN='0' THEN
                                        next_state<=done;
                                 ELSE
                                        next_state<=elaborate;
                                END IF;
WHEN done=> next_state<=idle;
WHEN OTHERS=> next_state<=idle;
END CASE;
END PROCESS;

--OUTPUTS
PROCESS(p_state)
BEGIN
valid_out<='0';

CASE p_state IS
WHEN idle=>valid_out<='0';

WHEN elaborate|done=>valid_out<='1';

WHEN OTHERS=>valid_out<='0';
END CASE;
END PROCESS;


END behavior;
```

datapath

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
USE ieee.numeric_std.all;

ENTITY datapath IS
PORT( CLK,RST_n: IN STD_LOGIC;
          V_IN : IN STD_LOGIC;
          DATA_IN: IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10 : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT: OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT: IN STD_LOGIC

                );
END ENTITY;

ARCHITECTURE struct OF datapath IS

COMPONENT reg_11bit IS
PORT( D_IN : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          D_OUT : OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          CLK,RST_n : IN STD_LOGIC;
          EN_REG: IN STD_LOGIC
                );
END COMPONENT;



COMPONENT signed_multiply_MBR is

port
(
        a                               : in std_logic_vector (10 downto 0);
        b                               : in std_logic_vector (10 downto 0);
        result          : out std_logic_vector (21 downto 0)
);

end COMPONENT;

COMPONENT adder_fixedpoint IS--DATA IN @24 BITS
PORT( A : IN STD_LOGIC_VECTOR(10 DOWNTO 0); --SIGN HAS TO BE EXTENDED
          B : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          SUM : OUT STD_LOGIC_VECTOR ( 10 DOWNTO 0)
                );
END COMPONENT;


SIGNAL regout0,regout1,regout2,regout3,regout4,regout5,regout6
,regout7,regout8,regout9,regout10: STD_LOGIC_VECTOR (10 DOWNTO 0);

SIGNAL prod_0,prod_1,prod_2,prod_3,prod_4,prod_5,prod_6
,prod_7,prod_8,prod_9,prod_10: STD_LOGIC_VECTOR(21 DOWNTO 0);

SIGNAL prod_0i,prod_1i,prod_2i,prod_3i,prod_4i,prod_5i,prod_6i
,prod_7i,prod_8i,prod_9i,prod_10i: STD_LOGIC_VECTOR(10 DOWNTO 0);
```

```
SIGNAL sum0, sum1,sum2,sum3,sum4,sum5,sum6
,sum7,sum8,sum9 : STD_LOGIC_VECTOR(10 DOWNTO 0);



BEGIN

 reg0: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>DATA_IN,D_OUT=>regout0);

 reg1: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout0,D_OUT=>regout1);

 reg2: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout1,D_OUT=>regout2);

 reg3: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout2,D_OUT=>regout3);

 reg4: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout3,D_OUT=>regout4);

 reg5: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout4,D_OUT=>regout5);

 reg6: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout5,D_OUT=>regout6);

 reg7: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout6,D_OUT=>regout7);

 reg8: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout7,D_OUT=>regout8);

 reg9: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout8,D_OUT=>regout9);

 reg10: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN
 ,D_IN=>regout9,D_OUT=>regout10);


 mpy0: signed_multiply_MBR PORT MAP(A=>regout0,B=>B0,result=>prod_0);
 mpy1: signed_multiply_MBR PORT MAP(A=>regout1,B=>B1,result=>prod_1);
 mpy2: signed_multiply_MBR PORT MAP(A=>regout2,B=>B2,result=>prod_2);
 mpy3: signed_multiply_MBR PORT MAP(A=>regout3,B=>B3,result=>prod_3);
 mpy4: signed_multiply_MBR PORT MAP(A=>regout4,B=>B4,result=>prod_4);
 mpy5: signed_multiply_MBR PORT MAP(A=>regout5,B=>B5,result=>prod_5);
 mpy6: signed_multiply_MBR PORT MAP(A=>regout6,B=>B6,result=>prod_6);
 mpy7: signed_multiply_MBR PORT MAP(A=>regout7,B=>B7,result=>prod_7);
 mpy8: signed_multiply_MBR PORT MAP(A=>regout8,B=>B8,result=>prod_8);
 mpy9: signed_multiply_MBR PORT MAP(A=>regout9,B=>B9,result=>prod_9);
 mpy10:signed_multiply_MBR PORT MAP(A=>regout10,B=>B10,result=>prod_10);
```

```
 add0: adder_fixedpoint PORT MAP(A=>prod_0i,B=>prod_1i,SUM=>sum0);
 add1: adder_fixedpoint PORT MAP(A=>sum0,B=>prod_2i,SUM=>sum1);
 add2: adder_fixedpoint PORT MAP(A=>sum1,B=>prod_3i,SUM=>sum2);
 add3: adder_fixedpoint PORT MAP(A=>sum2,B=>prod_4i,SUM=>sum3);
 add4: adder_fixedpoint PORT MAP(A=>sum3,B=>prod_5i,SUM=>sum4);
 add5: adder_fixedpoint PORT MAP(A=>sum4,B=>prod_6i,SUM=>sum5);
 add6: adder_fixedpoint PORT MAP(A=>sum5,B=>prod_7i,SUM=>sum6);
 add7: adder_fixedpoint PORT MAP(A=>sum6,B=>prod_8i,SUM=>sum7);
 add8: adder_fixedpoint PORT MAP(A=>sum7,B=>prod_9i,SUM=>sum8);
 add9: adder_fixedpoint PORT MAP(A=>sum8,B=>prod_10i,SUM=>sum9);

 prod_0i<=prod_0(20 DOWNTO 10);
 prod_1i<=prod_1(20 DOWNTO 10);
 prod_2i<=prod_2(20 DOWNTO 10);
 prod_3i<=prod_3(20 DOWNTO 10);
 prod_4i<=prod_4(20 DOWNTO 10);
 prod_5i<=prod_5(20 DOWNTO 10);
 prod_6i<=prod_6(20 DOWNTO 10);
 prod_7i<=prod_7(20 DOWNTO 10);
 prod_8i<=prod_8(20 DOWNTO 10);
 prod_9i<=prod_9(20 DOWNTO 10);
 prod_10i<=prod_10(20 DOWNTO 10);

 reg_data_out: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT
 ,D_IN=>sum9,D_OUT=>DATA_OUT);



END struct;
```

signed_multiply_MBR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_multiply_MBR is

port
(
        a                               : in std_logic_vector (10 downto 0);
        b                               : in std_logic_vector (10 downto 0);
        result          : out std_logic_vector (21 downto 0)
);

end entity;

architecture rtl of signed_multiply_MBR is

COMPONENT reorda_daddatree_sum is


PORT (
```

```
negA, negB , negC, negD, negE , negF : IN STD_LOGIC;

NEW_DATA_INA: IN STD_LOGIC_VECTOR(11 DOWNTO 0);        --A11, A9 , A7

NEW_DATA_INB : IN STD_LOGIC_VECTOR (11 DOWNTO 0);      --B
NEW_DATA_INC : IN STD_LOGIC_VECTOR( 11 DOWNTO 0);     --C
NEW_DATA_IND: IN STD_LOGIC_VECTOR (11 DOWNTO 0);       --D
NEW_DATA_INE: IN STD_LOGIC_VECTOR (11 DOWNTO 0);       --E
NEW_DATA_INF: IN STD_LOGIC_VECTOR(11 DOWNTO 0);      --F

Y : OUT STD_LOGIC_VECTOR ( 21 DOWNTO 0));

END COMPONENT;

COMPONENT Recording_Logic is
port
(
        X           : in std_logic_vector(2 downto 0);
        neg, two, non0 : out std_logic
);
end COMPONENT;

COMPONENT mux_2to1_12bit is
    Port ( SEL : in  STD_LOGIC;
           A   : in  STD_LOGIC_VECTOR (11 downto 0); --0
           B   : in  STD_LOGIC_VECTOR (11 downto 0); --1
           ENABLE: in std_logic;
           X   : out STD_LOGIC_VECTOR (11 downto 0));
end COMPONENT;

signal neg0,two0,non00,neg1,two1,
non01,neg2,two2,non02,neg3,two3,
non03,neg4,two4,non04,neg5,two5,non05 : std_logic;

signal out0, out1, out2, out3, out4, out5 :std_logic_vector (11 downto 0);

signal first_in,last_in :std_logic_vector(2 downto 0);

signal IN_mux_a,IN_mux_2a:std_logic_vector(11 downto 0);
begin

first_in<=a(1 downto 0) & '0';
last_in<=a(10 downto 10) & a(10 downto 9);

--RECORDING LOGIC
Recording_logic0 : Recording_Logic PORT MAP (first_in, neg0,two0,non00);
Recording_logic1 : Recording_Logic PORT MAP (a(3 downto 1), neg1,two1,non01);
Recording_logic2 : Recording_Logic PORT MAP (a(5 downto 3), neg2,two2,non02);
Recording_logic3 : Recording_Logic PORT MAP (a(7 downto 5), neg3,two3,non03);
Recording_logic4 : Recording_Logic PORT MAP (a(9 downto 7), neg4,two4,non04);
Recording_logic5 : Recording_Logic PORT MAP (last_in, neg5,two5,non05);
```

```
--in_mux
IN_mux_a<=b(10 downto 10) & b(10 downto 0);
In_mux_2a<=b(10 downto 0) & '0';

--MUX
mux0: mux_2to1_12bit PORT MAP ( two0,IN_mux_a, IN_mux_2a,non00, out0);
mux1: mux_2to1_12bit PORT MAP ( two1,IN_mux_a, IN_mux_2a,non01, out1);
mux2: mux_2to1_12bit PORT MAP ( two2,IN_mux_a, IN_mux_2a,non02, out2);
mux3: mux_2to1_12bit PORT MAP ( two3,IN_mux_a, IN_mux_2a,non03, out3);
mux4: mux_2to1_12bit PORT MAP ( two4,IN_mux_a, IN_mux_2a,non04, out4);
mux5: mux_2to1_12bit PORT MAP ( two5,IN_mux_a, IN_mux_2a,non05, out5);


--SUM
dada_tree: reorda_daddatree_sum PORT MAP (neg0,neg1,neg2,neg3,neg4,neg5,
out0,out1,out2,out3,out4,out5,result);

end rtl;
```

roorda dadda tree

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


ENTITY reorda_daddatree_sum is


PORT (


negA, negB , negC, negD, negE , negF : IN STD_LOGIC;

NEW_DATA_INA: IN STD_LOGIC_VECTOR(11 DOWNTO 0);        --A11, A9 , A7

NEW_DATA_INB : IN STD_LOGIC_VECTOR (11 DOWNTO 0);      --B
NEW_DATA_INC : IN STD_LOGIC_VECTOR( 11 DOWNTO 0);     --C
NEW_DATA_IND: IN STD_lOGIC_VECTOR (11 DOWNTO 0);       --D
NEW_DATA_INE: IN STD_lOGIC_VECTOR (11 DOWNTO 0);       --E
NEW_DATA_INF: IN STD_LOGIC_VECTOR(11 DOWNTO 0);      --F

Y : OUT STD_LOGIC_VECTOR ( 21 DOWNTO 0));

END reorda_daddatree_sum ;


ARCHITECTURE STRUCTURE OF REORDA_DADDATREE_SUM IS

--COMPONENTS

COMPONENT fulladd IS

PORT ( Cin, x, y : IN STD_LOGIC ;
```

```
s, Cout : OUT STD_LOGIC ) ;

END COMPONENT;

COMPONENT half_adder IS

PORT (  x, y : IN STD_LOGIC ;
s, Cout : OUT STD_LOGIC ) ;

END COMPONENT ;

--SIGNALS PER FARE I COMPLEMENTATI

SIGNAL DATA_INA, DATA_INB, DATA_INC, DATA_IND,
DATA_INE, DATA_INF : STD_lOGIC_VECTOR (11 DOWNTO 0);

SIGNAL DATAA , DATAB, DATAC , DATAD,
DATAE, DATAF : STD_LOGIC_VECTOR (11 DOWNTO 0);

SIGNAL EXTENSIONNEGA , EXTENSIONNEGB ,
EXTENSIONNEGC , EXTENSIONNEGD , EXTENSIONNEGE ,
EXTENSIONNEGF : STD_lOGIC_VECTOR ( 11 DOWNTO 0);

SIGNAL CARRYOUT1, CARRYOUT2 , CARRYOUT3, CARRYOUT4,
CARRYOUT5, CARRYOUT6 , CARRYOUT7, CARRYOUT8,
CARRYOUT9, CARRYOUT10 , CARRYOUT11 ,
CARRYOUT12 : STD_LOGIC;

SIGNAL CARRYOUT13, CARRYOUT14 , CARRYOUT15, CARRYOUT16, CARRYOUT17,
CARRYOUT18 ,  CARRYOUT19,
CARRYOUT20, CARRYOUT21, CARRYOUT22 ,
CARRYOUT23 ,CARRYOUT24 : STD_lOGIC;

SIGNAL CARRYOUT25, CARRYOUT26 , CARRYOUT27,
CARRYOUT28, CARRYOUT29, CARRYOUT30 , CARRYOUT31,
CARRYOUT32 ,CARRYOUT33, CARRYOUT34, CARRYOUT35,
CARRYOUT36 , CARRYOUT37, CARRYOUT38, CARRYOUT39,
CARRYOUT40 , CARRYOUT41, CARRYOUT42 ,CARRYOUT43,
CARRYOUT44 : STD_lOGIC;

SIGNAL CARRYOUT45, CARRYOUT46 , CARRYOUT47,
CARRYOUT47_1 , CARRYOUT48, CARRYOUT49, CARRYOUT50 ,
CARRYOUT51, CARRYOUT52 ,CARRYOUT53, CARRYOUT54 ,
CARRYOUT55, CARRYOUT56 , CARRYOUT57, CARRYOUT58,
CARRYOUT59, CARRYOUT60 , CARRYOUT61, CARRYOUT62
,CARRYOUT63, CARRYOUT64, CARRYOUT65, CARRYOUT66 ,
CARRYOUT67, CARRYOUT68, CARRYOUT69, CARRYOUT70 ,
CARRYOUT71 : STD_LOGIC;
--SIGNALS

SIGNAL S1,S2,S3,S4,S5,S6,S7,S8,S9 : STD_LOGIC;
--SEGNALI DEL PRIMO STEP L3 = 6  (HALF ADDER)
SIGNAL C1,C2,C3,C4,C5,C6,C7,C8, C9 : STD_LOGIC;
```

```
SIGNAL SS1,SS2,SS3,SS4,SS5,SS6,SS7,SS8,
SS9,SS10,SS11 : STD_LOGIC;
--(FULL ADDERS)
SIGNAL CC1,CC2,CC3,CC4,CC5,CC6,CC7,CC8,
CC9,CC10, CC11 : STD_LOGIC;

-- SECONDO STEP

SIGNAL S10, S11, S12, S13  : STD_LOGIC;
SIGNAL  C10, C11, C12, C13 : STD_LOGIC;

SIGNAL SS12,SS13,SS14,SS15,SS16,SS17,
SS18,SS19,SS20,SS21,SS22,SS23,SS24 : STD_LOGIC;
SIGNAL CC12,CC13,CC14,CC15,CC16,
CC17,CC18,CC19,CC20,CC21,CC22,CC23,
CC24 : STD_LOGIC;


--TERZO STEP

SIGNAL S14 , C14 : STD_LOGIC;

SIGNAL SS25,SS26,SS27,SS28,SS29,SS30,
SS31,SS32,SS33,SS34,SS35,SS36,SS37,
SS38,SS39,SS40,SS41 : STD_LOGIC;
SIGNAL CC25,CC26,CC27,CC28,CC29,CC30,
CC31,CC32,CC33,CC34,CC35,CC36,CC37,
CC38,CC39,CC40,CC41 : STD_LOGIC;

--SOMMATORE

SIGNAL SS42,SS43,SS44,SS45,SS46,SS47,
SS48,SS49,SS50,SS51,SS52,SS53,SS54,
SS55,SS56,SS57,SS58,SS59,SS60,
SS61 : STD_LOGIC;
SIGNAL CC42,CC43,CC44,CC45,CC46,CC47,
CC48,CC49,CC50,CC51,CC52,CC53,CC54,CC55,
CC56,CC57,CC58,CC59,CC60,CC61 : STD_LOGIC;


--SIGNAL NEGATI
SIGNAL NOTDATA_INC , NOTDATA_INF , NOTDATA_IND,
NOTDATA_INB, NOTDATA_INA , NOTDATA_INE :
STD_lOGIC_VECTOR(11 DOWNTO 0);

--END SIGNALS



--DATAPATH
```

```
BEGIN

--COMPLEMENTO NEGO E AGGIUNGO 1

extensionNEGA <= NEGA & NEGA & NEGA & NEGA & NEGA &
NEGA & NEGA & NEGA & NEGA & NEGA & NEGA & NEGA;

extensionNEGB <= NEGB & NEGB & NEGB & NEGB & NEGB &
NEGB & NEGB & NEGB & NEGB & NEGB & NEGB & NEGB;

extensionNEGC <= NEGC & NEGC & NEGC & NEGC & NEGC &
NEGC & NEGC & NEGC & NEGC & NEGC & NEGC & NEGC;

extensionNEGD <= NEGD & NEGD & NEGD & NEGD & NEGD &
NEGD & NEGD & NEGD & NEGD & NEGD & NEGD & NEGD;

extensionNEGE <= NEGE & NEGE & NEGE & NEGE & NEGE &
NEGE & NEGE & NEGE & NEGE & NEGE & NEGE & NEGE;

extensionNEGF <= NEGF & NEGF & NEGF & NEGF & NEGF &
NEGF & NEGF & NEGF & NEGF & NEGF & NEGF & NEGF;



DATAA <= EXTENSIONNEGA XOR NEW_DATA_INA;

DATAB <= EXTENSIONNEGB XOR NEW_DATA_INB;

DATAC <= EXTENSIONNEGC XOR NEW_DATA_INC;

DATAD <= EXTENSIONNEGD XOR NEW_DATA_IND;

DATAE <= EXTENSIONNEGE XOR NEW_DATA_INE;

DATAF <= EXTENSIONNEGF XOR NEW_DATA_INF;

--COMPLEMENTA A

FULL_ADDER01 : fulladd port map ( DATAA (0) , '0',
NEGA , DATA_INA(0) , CARRYOUT1);

FULL_ADDER02 : fulladd port map ( DATAA(1) , '0',
CARRYOUT1, DATA_INA(1) , CARRYOUT2);

FULL_ADDER03 : fulladd port map ( DATAA (2) ,
'0',
CARRYOUT2, DATA_INA(2) , CARRYOUT3);

FULL_ADDER04 : fulladd port map ( DATAA (3) , '0' ,
CARRYOUT3, DATA_INA(3) , CARRYOUT4);

FULL_ADDER05 : fulladd port map ( DATAA (4) , '0',
```

```
CARRYOUT4 , DATA_INA(4) , CARRYOUT5);

FULL_ADDER06 : fulladd port map ( DATAA (5) , '0',
CARRYOUT5 , DATA_INA(5) , CARRYOUT6 );

FULL_ADDER07 : fulladd port map ( DATAA(6) , '0',
CARRYOUT6 , DATA_INA(6) , CARRYOUT7);

FULL_ADDER08 : fulladd port map ( DATAA (7) , '0',
CARRYOUT7 , DATA_INA(7) , CARRYOUT8 );

FULL_ADDER09 : fulladd port map ( DATAA (8) , '0',
CARRYOUT8 , DATA_INA(8) , CARRYOUT9);

FULL_ADDER010 : fulladd port map ( DATAA (9) , '0' ,
CARRYOUT9 , DATA_INA(9) , CARRYOUT10);

FULL_ADDER011 : fulladd port map ( DATAA (10),'0',
CARRYOUT10 , DATA_INA(10) , CARRYOUT11);

FULL_ADDER012 : fulladd port map ( DATAA (11),'0',
CARRYOUT11 , DATA_INA(11) , CARRYOUT12);

--COMPLEMENTA B

FULL_ADDER013 : fulladd port map ( DATAB (0) , '0' ,
NEGB , DATA_INB(0) , CARRYOUT13);

FULL_ADDER014 : fulladd port map ( DATAB(1) , '0',
CARRYOUT13, DATA_INB(1) , CARRYOUT14);

FULL_ADDER015 : fulladd port map ( DATAB (2) , '0',
CARRYOUT14, DATA_INB(2) , CARRYOUT15);

FULL_ADDER016 : fulladd port map ( DATAB (3) , '0' ,
CARRYOUT15, DATA_INB(3) , CARRYOUT16);

FULL_ADDER017 : fulladd port map ( DATAB (4) , '0',
CARRYOUT16 , DATA_INB(4) , CARRYOUT17);

FULL_ADDER018 : fulladd port map ( DATAB (5) , '0',
CARRYOUT17 , DATA_INB(5) , CARRYOUT18 );

FULL_ADDER019 : fulladd port map ( DATAB(6) , '0',
CARRYOUT18 , DATA_INB(6) , CARRYOUT19);

FULL_ADDER020 : fulladd port map ( DATAB (7) , '0',
CARRYOUT19 , DATA_INB(7) , CARRYOUT20 );

FULL_ADDER021 : fulladd port map ( DATAB (8) , '0',
CARRYOUT20 , DATA_INB(8) , CARRYOUT21);

FULL_ADDER022 : fulladd port map ( DATAB (9) ,'0',
```

```
CARRYOUT21 , DATA_INB(9) , CARRYOUT22);

FULL_ADDER023 : fulladd port map ( DATAB (10),'0',
CARRYOUT22 , DATA_INB(10) , CARRYOUT23);

FULL_ADDER024 : fulladd port map ( DATAB (11),'0',
CARRYOUT23 , DATA_INB(11) , CARRYOUT24);

-- COMPLEMENTA C

FULL_ADDER025 : fulladd port map ( DATAC (0) , '0',
NEGC , DATA_INC(0) , CARRYOUT25);

FULL_ADDER026 : fulladd port map ( DATAC(1) , '0',
CARRYOUT25, DATA_INC(1) , CARRYOUT26);

FULL_ADDER027 : fulladd port map ( DATAC (2) , '0',
CARRYOUT26, DATA_INC(2) , CARRYOUT27);

FULL_ADDER028 : fulladd port map ( DATAC (3) , '0',
CARRYOUT27, DATA_INC(3) , CARRYOUT28);

FULL_ADDER029 : fulladd port map ( DATAC (4) , '0',
CARRYOUT28 , DATA_INC(4) , CARRYOUT29);

FULL_ADDER030 : fulladd port map ( DATAC (5) , '0',
CARRYOUT29 , DATA_INC(5) , CARRYOUT30 );

FULL_ADDER031 : fulladd port map ( DATAC(6) , '0',
CARRYOUT30 , DATA_INC(6) , CARRYOUT31);

FULL_ADDER032 : fulladd port map ( DATAC (7) , '0',
CARRYOUT31, DATA_INC(7) , CARRYOUT32 );

FULL_ADDER033 : fulladd port map ( DATAC (8) , '0',
CARRYOUT32 , DATA_INC(8) , CARRYOUT33);

FULL_ADDER034 : fulladd port map ( DATAC (9) ,'0',
CARRYOUT33 , DATA_INC(9) , CARRYOUT34);

FULL_ADDER035 : fulladd port map ( DATAC (10),'0',
CARRYOUT34, DATA_INC(10) , CARRYOUT35);

FULL_ADDER036 : fulladd port map ( DATAC (11),'0',
CARRYOUT35 , DATA_INC(11) , CARRYOUT36);

--COMPLEMENTA D

FULL_ADDER037 : fulladd port map ( DATAD (0) , '0',
NEGD , DATA_IND(0) , CARRYOUT37);

FULL_ADDER038 : fulladd port map ( DATAD(1) , '0',
CARRYOUT37, DATA_IND(1) , CARRYOUT38);
```

```
FULL_ADDER039 : fulladd port map ( DATAD (2) , '0',
CARRYOUT38, DATA_IND(2) , CARRYOUT39);

FULL_ADDER040 : fulladd port map ( DATAD (3) , '0',
CARRYOUT39, DATA_IND(3) , CARRYOUT40);

FULL_ADDER041 : fulladd port map ( DATAD (4) , '0',
CARRYOUT40 , DATA_IND(4) , CARRYOUT41);

FULL_ADDER042 : fulladd port map ( DATAD (5) , '0',
CARRYOUT41 , DATA_IND(5) , CARRYOUT42 );

FULL_ADDER043 : fulladd port map ( DATAD(6) , '0',
CARRYOUT42 , DATA_IND(6) , CARRYOUT43);

FULL_ADDER044 : fulladd port map ( DATAD (7) , '0',
CARRYOUT43, DATA_IND(7) , CARRYOUT44 );

FULL_ADDER045 : fulladd port map ( DATAD (8) , '0',
CARRYOUT44 , DATA_IND(8) , CARRYOUT45);

FULL_ADDER046 : fulladd port map ( DATAD (9) ,'0',
CARRYOUT45 , DATA_IND(9) , CARRYOUT46);

FULL_ADDER047 : fulladd port map ( DATAD (10),'0',
CARRYOUT46, DATA_IND(10) , CARRYOUT47);

FULL_ADDER048_1 : fulladd port map ( DATAD(11) , '0',
CARRYOUT47, DATA_IND(11) , CARRYOUT47_1);

--COMPLEMENTA E

FULL_ADDER048 : fulladd port map ( DATAE (0) , '0',
NEGE , DATA_INE(0) , CARRYOUT48);

FULL_ADDER049 : fulladd port map ( DATAE(1) , '0',
CARRYOUT48, DATA_INE(1) , CARRYOUT49);

FULL_ADDER050 : fulladd port map ( DATAE (2) , '0',
CARRYOUT49, DATA_INE(2) , CARRYOUT50);

FULL_ADDER051 : fulladd port map ( DATAE (3) , '0' ,
CARRYOUT50, DATA_INE(3) , CARRYOUT51);

FULL_ADDER052 : fulladd port map ( DATAE (4) , '0',
CARRYOUT51 , DATA_INE(4) , CARRYOUT52);

FULL_ADDER053 : fulladd port map ( DATAE (5) , '0',
CARRYOUT52 , DATA_INE(5) , CARRYOUT53 );

FULL_ADDER054 : fulladd port map ( DATAE(6) , '0',
CARRYOUT53 , DATA_INE(6) , CARRYOUT54);
```

```
FULL_ADDER055 : fulladd port map ( DATAE (7) , '0',
CARRYOUT54, DATA_INE(7) , CARRYOUT55 );

FULL_ADDER056 : fulladd port map ( DATAE (8) , '0',
CARRYOUT55 , DATA_INE(8) , CARRYOUT56);

FULL_ADDER057 : fulladd port map ( DATAE (9) ,'0',
CARRYOUT56 , DATA_INE(9) , CARRYOUT57);

FULL_ADDER058 : fulladd port map ( DATAE (10),'0',
CARRYOUT57, DATA_INE(10) , CARRYOUT58);


FULL_ADDER059 : fulladd port map ( DATAE (11),'0',
CARRYOUT58 , DATA_INE(11) , CARRYOUT59);

--COMPLEMENTA F


FULL_ADDER060 : fulladd port map ( DATAF (0) , '0' ,
NEGF , DATA_INF(0) , CARRYOUT60);

FULL_ADDER061 : fulladd port map ( DATAF(1) , '0',
CARRYOUT60, DATA_INF(1) , CARRYOUT61);

FULL_ADDER062 : fulladd port map ( DATAF (2) , '0',
CARRYOUT61, DATA_INF(2) , CARRYOUT62);

FULL_ADDER063 : fulladd port map ( DATAF (3) , '0',
CARRYOUT62, DATA_INF(3) , CARRYOUT63);

FULL_ADDER064 : fulladd port map ( DATAF (4) , '0',
CARRYOUT63 , DATA_INF(4) , CARRYOUT64);

FULL_ADDER065 : fulladd port map ( DATAF (5) , '0',
CARRYOUT64 , DATA_INF(5) , CARRYOUT65 );

FULL_ADDER066 : fulladd port map ( DATAF(6) , '0',
CARRYOUT65 , DATA_INF(6) , CARRYOUT66);

FULL_ADDER067 : fulladd port map ( DATAF (7) , '0',
CARRYOUT66, DATA_INF(7) , CARRYOUT67 );

FULL_ADDER068 : fulladd port map ( DATAF (8) , '0',
CARRYOUT67 , DATA_INF(8) , CARRYOUT68);

FULL_ADDER069 : fulladd port map ( DATAF (9) ,'0',
CARRYOUT68 , DATA_INF(9) , CARRYOUT69);

FULL_ADDER070 : fulladd port map ( DATAF (10),'0',
CARRYOUT69, DATA_INF(10) , CARRYOUT70);
```

```
FULL_ADDER071 : fulladd port map ( DATAF (11),'0',
CARRYOUT70 , DATA_INF(11) , CARRYOUT71);
```

```
--9 HALF ADDER

NOTDATA_INC <= NOT DATA_INC;
NOTDATA_INF <= NOT DATA_INF;
```

```
HALF_ADDER1 : half_adder port map ( DATA_IND (2) , DATA_INE (0) , S1 , C1);

HALF_ADDER2 : half_adder port map ( DATA_INC (6) , DATA_INB (8) , S2 , C2);

HALF_ADDER3 : half_adder port map ( DATA_IND (9) , NOTDATA_INC (11) , S3 , C3);
--9 HALF ADDER + 11 FULL ADDER ( PRIMO STEP)

HALF_ADDER4 : half_adder port map ( DATA_INE (7) , DATA_INF (5) , S4 , C4);

HALF_ADDER5 : half_adder port map ( DATA_INC (11) , DATA_INC (11) , S5 , C5);

HALF_ADDER6 : half_adder port map ( DATA_IND (11) , NOTDATA_INC (11) , S6 , C6);

HALF_ADDER7 : half_adder port map ( DATA_INF (8) , DATA_INE (10) , S7 , C7);

HALF_ADDER8 : half_adder port map ( NOTDATA_INE (11) ,
NOTDATA_INF (9) , S8 , C8);

HALF_ADDER9 : half_adder port map ( DATA_INF (9), DATA_INF (11) , S9 , C9);
```

```
NOTDATA_INB <= NOT DATA_INB;
NOTDATA_INA <= NOT DATA_INA;
NOTDATA_IND <= NOT DATA_IND;
```

```
--11 FULL ADDER

FULL_ADDER1 : fulladd port map ( DATA_INC (5) , DATA_INE(1),
DATA_IND(3) , SS1 , CC1);

FULL_ADDER2 : fulladd port map ( DATA_IND (4) , DATA_INE(2),
```

```
DATA_INF(0) , SS2 , CC2);

FULL_ADDER3 : fulladd port map ( DATA_INF (1) , DATA_INE(3),
DATA_IND(5) , SS3 , CC3);

FULL_ADDER4 : fulladd port map ( DATA_INC (7) , DATA_INB(9),
DATA_INA(11) , SS4 , CC4);

FULL_ADDER5 : fulladd port map ( DATA_INC (8) , NOTDATA_INB(10),
NOTDATA_INA(11) , SS5 , CC5);

FULL_ADDER6 : fulladd port map ( DATA_INF (2) , DATA_INE(4),
DATA_IND(6) , SS6 , CC6);

FULL_ADDER7 : fulladd port map ( DATA_INC (9) ,NOTDATA_INB(11),
DATA_INB(10) , SS7 , CC7);

FULL_ADDER8 : fulladd port map ( DATA_INF (3) , DATA_INE(5),
DATA_IND(7) , SS8 , CC8);

FULL_ADDER9 : fulladd port map ( DATA_INF (4) , DATA_INE(6),
DATA_IND(8) , SS9 , CC9);

FULL_ADDER10 : fulladd port map ( DATA_INF (6) , DATA_INE(8)
, NOTDATA_IND(10) , SS10 , CC10);

FULL_ADDER11 : fulladd port map ( DATA_IND (10) , DATA_INE(9),
DATA_INF(7) , SS11 , CC11);



--SECONDO STEP L2 = 4 (4 HALF ADDER + 13 FULL ADDER )

NOTDATA_INE <= NOT DATA_INE;


HALF_ADDER10 : half_adder port map ( DATA_INB (4) ,
DATA_INA (6) , S10 , C10);

HALF_ADDER11 : half_adder port map ( DATA_INF (9) ,
NOTDATA_INE (11) , S11 , C11);

HALF_ADDER12 : half_adder port map ( C8 ,  S9 , S12 , C12);

HALF_ADDER13 : half_adder port map ( NOTDATA_INF (10) ,
NOTDATA_INF (10) , S13 , C13);



--FULL ADDERS


FULL_ADDER12 : fulladd port map ( DATA_INA(7) ,
```

```
DATA_INB(5), DATA_INC(3) , SS12 , CC12);

FULL_ADDER13 : fulladd port map ( S1 , DATA_INA(8),
DATA_INB(6) , SS13 , CC13);

FULL_ADDER14 : fulladd port map ( C1 , SS1 ,
DATA_INA(9) , SS14 , CC14);

FULL_ADDER15 : fulladd port map ( S2 , CC1 , SS2 ,
SS15 , CC15);

FULL_ADDER16 : fulladd port map ( SS4, C2 , SS3 ,
SS16 , CC16);

FULL_ADDER17 : fulladd port map ( SS5 , SS6 , CC4 ,
SS17 , CC17);

FULL_ADDER18 : fulladd port map ( SS8 , CC5 , CC6  ,
SS18 , CC18);

FULL_ADDER19 : fulladd port map ( CC8 , SS9 ,
NOTDATA_INC(10) , SS19 , CC19);

FULL_ADDER20 : fulladd port map ( S3 , CC9 , S4,
SS20 ,CC20);

FULL_ADDER21 : fulladd port map ( SS10 , S5 , C3 ,
SS21 ,CC21);

FULL_ADDER22 : fulladd port map ( S6 , C5 , CC10 ,
SS22 , CC22);

FULL_ADDER23 : fulladd port map ( C6 , S7,
NOTDATA_IND(11) , SS23, CC23);

FULL_ADDER24 : fulladd port map ( S8 , DATA_INF(10),
NOTDATA_INF(9) , SS24 , CC24);


--TERZO STEP L1 = 3 ( 1 HALF ADDER + 17 FULL ADDER )




HALF_ADDER14 : half_adder port map (  DATA_INB (2) ,
DATA_INC (0) , S14 , C14);




FULL_ADDER25 : fulladd port map ( DATA_INA(5) ,
DATA_INB(3), DATA_INC(1) , SS25 , CC25);
```

```
FULL_ADDER26 : fulladd port map ( S10 , DATA_IND(0),
DATA_INC(2) , SS26 , CC26);

FULL_ADDER27 : fulladd port map ( SS12 , C10 ,
DATA_IND(1) , SS27 , CC27);

FULL_ADDER28 : fulladd port map ( SS13 , CC12 , C4 ,
SS28 , CC28);

FULL_ADDER29 : fulladd port map ( SS14, CC13 ,
DATA_INB(7) , SS29 , CC29);

FULL_ADDER30 : fulladd port map ( SS15 , CC14 ,
DATA_INA(10) , SS30, CC30);

FULL_ADDER31 : fulladd port map ( SS16 , CC15 , CC2  ,
SS31 , CC31);

FULL_ADDER32 : fulladd port map ( SS17 , CC16, CC3 ,
SS32 , CC32);

FULL_ADDER33 : fulladd port map ( SS18 , CC17, SS7 ,
SS33, CC33);

FULL_ADDER34 : fulladd port map ( CC18 , SS19 , CC7 ,
SS34 , CC34);

FULL_ADDER35 : fulladd port map ( SS20 , CC19 ,
DATA_INC(10) , SS35 , CC35);

FULL_ADDER36 : fulladd port map ( SS21, CC20,
DATA_INC(4) , SS36 , CC36);

FULL_ADDER37 : fulladd port map ( SS22 , CC21 ,
SS11 , SS37 , CC37);

FULL_ADDER38 : fulladd port map ( SS23 , CC22 ,
CC11  , SS38 , CC38);

FULL_ADDER39 : fulladd port map ( S11 , CC23 ,
DATA_INC(7)  , SS39 , CC39);

FULL_ADDER40 : fulladd port map ( SS24 , C11 ,
DATA_INE(11) , SS40 , CC40);

FULL_ADDER41 : fulladd port map ( S12 , CC24 ,
S13 , SS41 , CC41);


--SOMMATORE 20 BIT
```

```
FULL_ADDER42 : fulladd port map ( DATA_INA(2) , DATA_INB(0) , '0' , SS42 , CC42);

FULL_ADDER43 : fulladd port map ( CC42, DATA_INB(1) , DATA_INA(3) , SS43 , CC43);

FULL_ADDER44 : fulladd port map ( CC43 , S14, DATA_INA(4) , SS44, CC44);

FULL_ADDER45 : fulladd port map ( CC44 , C14, SS25 , SS45, CC45);

FULL_ADDER46 : fulladd port map ( CC45 , SS26 , CC25 , SS46 , CC46);

FULL_ADDER47 : fulladd port map ( CC46 , CC26, SS27 , SS47, CC47);

FULL_ADDER48 : fulladd port map ( CC47 , CC27 , SS28 , SS48 , CC48);

FULL_ADDER49 : fulladd port map ( CC48 , CC28 , SS29 , SS49 , CC49);

FULL_ADDER50 : fulladd port map ( CC49, CC29, SS30 , SS50 , CC50);

FULL_ADDER51 : fulladd port map ( CC50 , CC30 , SS31 , SS51 , CC51);

FULL_ADDER52 : fulladd port map ( CC51 , CC31 , SS32  , SS52 , CC52);

FULL_ADDER53 : fulladd port map ( CC52 , CC32 ,SS33   , SS53 , CC53);

FULL_ADDER54 : fulladd port map ( CC53, CC33 , SS34 , SS54 , CC54);

FULL_ADDER55 : fulladd port map ( CC54 , CC34 , SS35 , SS55 , CC55);

FULL_ADDER56 : fulladd port map ( CC55 , CC35 , SS36 , SS56 , CC56);

FULL_ADDER57 : fulladd port map ( CC56, CC36 , SS37 , SS57 , CC57);

FULL_ADDER58 : fulladd port map ( CC57 , CC37 , SS38 , SS58, CC58);

FULL_ADDER59 : fulladd port map ( CC58 , CC38 , SS39  , SS59 , CC59);

FULL_ADDER60 : fulladd port map ( CC59 , CC39 , SS40 , SS60 , CC60);

FULL_ADDER61 : fulladd port map ( CC60 , CC40, SS41 , SS61, CC61);


y <= SS61 & SS60 & SS59 & SS58 & SS57 & SS56 &
SS55 & SS54 & SS53 & SS52 & SS51 & SS50 & SS49
& SS48 & SS47 & SS46 & SS45 & SS44 & SS43 & SS42
& DATA_INA(1) & DATA_INA(0);

END STRUCTURE;
```

Recording logic

```
library ieee;

use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;
entity Recording_Logic is
port
(
        X           : in std_logic_vector(2 downto 0);
        neg, two, non0 : out std_logic

);
end Recording_Logic;


architecture rtl of Recording_Logic is

begin
process(X)
begin
    case X is
        when "000" => neg <= '0';--0
                      non0<= '0';
        when "001" => neg <= '0';--a
                      non0<= '1';
                      two<= '0';
        when "010" => neg <= '0';--a
                      non0<= '1';
                      two<= '0';
        when "011" => neg <= '0';--2a
                      non0<= '1';
                      two<= '1';
        when "100" => neg <= '1';-- -2a
                      non0<= '1';
                      two<= '1';
        when "101" => neg <= '1';-- -a
                      non0<= '1';
                      two<= '0';
        when "110" => neg <= '1';-- -a
                      non0<= '1';
                      two<= '0';
        when "111" => neg <= '1';--0
                      non0<= '0';
        when others =>neg <= '1';-- error
                      non0<= '1';
                      two<= '1';
    end case;
end process;

end rtl;
```

4.3.2 Version 2

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY datapath IS
```

```
PORT( CLK,RST_n: IN STD_LOGIC;
          V_IN : IN STD_LOGIC;
          DATA_IN: IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          B0,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10 : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          DATA_OUT: OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          V_OUT: IN STD_LOGIC

                );
END ENTITY;

ARCHITECTURE struct OF datapath IS

COMPONENT reg_11bit IS
PORT( D_IN : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          D_OUT : OUT STD_LOGIC_VECTOR (10 DOWNTO 0);
          CLK,RST_n : IN STD_LOGIC;
          EN_REG: IN STD_LOGIC
                );
END COMPONENT;



COMPONENT signed_multiply_MBR is

port
(
        a                               : in std_logic_vector (10 downto 0);
        b                               : in std_logic_vector (10 downto 0);
        result          : out std_logic_vector (21 downto 0)
);

end COMPONENT;

COMPONENT adder_fixedpoint IS--DATA IN @24 BITS
PORT( A : IN STD_LOGIC_VECTOR(10 DOWNTO 0); --SIGN HAS TO BE EXTENDED
          B : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
          SUM : OUT STD_LOGIC_VECTOR ( 10 DOWNTO 0)
                );
END COMPONENT;


SIGNAL regout0,regout1,regout2,regout3,regout4,
regout5,regout6,regout7,regout8,regout9,
regout10: STD_LOGIC_VECTOR (10 DOWNTO 0);
SIGNAL prod_0,prod_1,prod_2,prod_3,prod_4,prod_5,
prod_6,prod_7,prod_8,prod_9,
prod_10: STD_LOGIC_VECTOR(21 DOWNTO 0);
SIGNAL prod_0i,prod_1i,prod_2i,prod_3i,
prod_4i,prod_5i,prod_6i,prod_7i,prod_8i,
prod_9i,prod_10i: STD_LOGIC_VECTOR(10 DOWNTO 0);
SIGNAL sum0, sum1,sum2,sum3,sum4,sum5,sum6,
sum7,sum8,sum9 : STD_LOGIC_VECTOR(10 DOWNTO 0);
```

```
BEGIN

 reg0: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>DATA_IN,D_OUT=>regout0);
 reg1: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout0,D_OUT=>regout1);
 reg2: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout1,D_OUT=>regout2);
 reg3: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout2,D_OUT=>regout3);
 reg4: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout3,D_OUT=>regout4);
 reg5: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout4,D_OUT=>regout5);
 reg6: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout5,D_OUT=>regout6);
 reg7: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout6,D_OUT=>regout7);
 reg8: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout7,D_OUT=>regout8);
 reg9: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout8,D_OUT=>regout9);
 reg10: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_IN,
 D_IN=>regout9,D_OUT=>regout10);

 mpy0: signed_multiply_MBR PORT MAP(A=>regout0,B=>B0,result=>prod_0);
 mpy1: signed_multiply_MBR PORT MAP(A=>regout1,B=>B1,result=>prod_1);
 mpy2: signed_multiply_MBR PORT MAP(A=>regout2,B=>B2,result=>prod_2);
 mpy3: signed_multiply_MBR PORT MAP(A=>regout3,B=>B3,result=>prod_3);
 mpy4: signed_multiply_MBR PORT MAP(A=>regout4,B=>B4,result=>prod_4);
 mpy5: signed_multiply_MBR PORT MAP(A=>regout5,B=>B5,result=>prod_5);
 mpy6: signed_multiply_MBR PORT MAP(A=>regout6,B=>B6,result=>prod_6);
 mpy7: signed_multiply_MBR PORT MAP(A=>regout7,B=>B7,result=>prod_7);
 mpy8: signed_multiply_MBR PORT MAP(A=>regout8,B=>B8,result=>prod_8);
 mpy9: signed_multiply_MBR PORT MAP(A=>regout9,B=>B9,result=>prod_9);
 mpy10:signed_multiply_MBR PORT MAP(A=>regout10,B=>B10,result=>prod_10);

 add0: adder_fixedpoint PORT MAP(A=>prod_0i,B=>prod_1i,SUM=>sum0);
 add1: adder_fixedpoint PORT MAP(A=>sum0,B=>prod_2i,SUM=>sum1);
 add2: adder_fixedpoint PORT MAP(A=>sum1,B=>prod_3i,SUM=>sum2);
 add3: adder_fixedpoint PORT MAP(A=>sum2,B=>prod_4i,SUM=>sum3);
 add4: adder_fixedpoint PORT MAP(A=>sum3,B=>prod_5i,SUM=>sum4);
 add5: adder_fixedpoint PORT MAP(A=>sum4,B=>prod_6i,SUM=>sum5);
 add6: adder_fixedpoint PORT MAP(A=>sum5,B=>prod_7i,SUM=>sum6);
 add7: adder_fixedpoint PORT MAP(A=>sum6,B=>prod_8i,SUM=>sum7);
 add8: adder_fixedpoint PORT MAP(A=>sum7,B=>prod_9i,SUM=>sum8);
 add9: adder_fixedpoint PORT MAP(A=>sum8,B=>prod_10i,SUM=>sum9);

 prod_0i<=prod_0(20 DOWNTO 10);
 prod_1i<=prod_1(20 DOWNTO 10);
 prod_2i<=prod_2(20 DOWNTO 10);
```

```
 prod_3i<=prod_3(20 DOWNTO 10);
 prod_4i<=prod_4(20 DOWNTO 10);
 prod_5i<=prod_5(20 DOWNTO 10);
 prod_6i<=prod_6(20 DOWNTO 10);
 prod_7i<=prod_7(20 DOWNTO 10);
 prod_8i<=prod_8(20 DOWNTO 10);
 prod_9i<=prod_9(20 DOWNTO 10);
 prod_10i<=prod_10(20 DOWNTO 10);

 reg_data_out: reg_11bit PORT MAP (CLK=>CLK,RST_n=>RST_n,EN_REG=>V_OUT,
 D_IN=>sum9,D_OUT=>DATA_OUT);



END struct;
```

4.3.3 Version 3

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY compressor_4to2 IS
PORT ( A,B,C,D: IN STD_LOGIC;
            CARRY: OUT STD_LOGIC;
            SUM: OUT STD_LOGIC
                 );
END ENTITY;

ARCHITECTURE beha OF compressor_4to2 IS
BEGIN

SUM<=NOT(C XOR D);
CARRY<=NOT((NOT(A OR B))OR(NOT(C OR D)));

END beha;

LIBRARY ieee ;
USE ieee.std_logic_1164.all;



ENTITY fulladd IS
PORT ( Cin, x, y : IN STD_LOGIC ;
s, Cout : OUT STD_LOGIC ) ;
END fulladd ;


ARCHITECTURE LogicFunc OF fulladd IS
BEGIN


s <= x XOR y XOR Cin ;
```

```
Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;


ENTITY half_adder IS
PORT (  x, y : IN STD_LOGIC ;
s, Cout : OUT STD_LOGIC ) ;

END half_adder;


ARCHITECTURE LogicFunc OF half_adder IS
BEGIN

s <= x XOR y  ;

Cout <=(x AND y);

END LogicFunc ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY PP_generator IS
PORT ( X, Y : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
           PP_0,PP_1,PP_2,PP_3,PP_4,PP_5: OUT STD_LOGIC_VECTOR(10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE beha OF PP_generator IS

BEGIN


PP_0(0)<=(X(0) OR '0') AND (X(1) XOR Y(0));
PP_0(1)<=(X(0) OR '0') AND (X(1) XOR Y(1));
PP_0(2)<=(X(0) OR '0') AND (X(1) XOR Y(2));
PP_0(3)<=(X(0) OR '0') AND (X(1) XOR Y(3));
PP_0(4)<=(X(0) OR '0') AND (X(1) XOR Y(4));
PP_0(5)<=(X(0) OR '0') AND (X(1) XOR Y(5));
PP_0(6)<=(X(0) OR '0') AND (X(1) XOR Y(6));
PP_0(7)<=(X(0) OR '0') AND (X(1) XOR Y(7));
PP_0(8)<=(X(0) OR '0') AND (X(1) XOR Y(8));
PP_0(9)<=(X(0) OR '0') AND (X(1) XOR Y(9));
PP_0(10)<=(X(0) OR '0') AND (X(1) XOR Y(10));

PP_1(0)<=(X(2) OR X(1)) AND (X(3) XOR Y(0));
PP_1(1)<=(X(2) OR X(1)) AND (X(3) XOR Y(1));
PP_1(2)<=(X(2) OR X(1)) AND (X(3) XOR Y(2));
PP_1(3)<=(X(2) OR X(1)) AND (X(3) XOR Y(3));
```

```
PP_1(4)<=(X(2) OR X(1)) AND (X(3) XOR Y(4));
PP_1(5)<=(X(2) OR X(1)) AND (X(3) XOR Y(5));
PP_1(6)<=(X(2) OR X(1)) AND (X(3) XOR Y(6));
PP_1(7)<=(X(2) OR X(1)) AND (X(3) XOR Y(7));
PP_1(8)<=(X(2) OR X(1)) AND (X(3) XOR Y(8));
PP_1(9)<=(X(2) OR X(1)) AND (X(3) XOR Y(9));
PP_1(10)<=(X(2) OR X(1)) AND (X(3) XOR Y(10));

PP_2(0)<=(X(4) OR X(3)) AND (X(5) XOR Y(0));
PP_2(1)<=(X(4) OR X(3)) AND (X(5) XOR Y(1));
PP_2(2)<=(X(4) OR X(3)) AND (X(5) XOR Y(2));
PP_2(3)<=(X(4) OR X(3)) AND (X(5) XOR Y(3));
PP_2(4)<=(X(4) OR X(3)) AND (X(5) XOR Y(4));
PP_2(5)<=(X(4) OR X(3)) AND (X(5) XOR Y(5));
PP_2(6)<=(X(4) OR X(3)) AND (X(5) XOR Y(6));
PP_2(7)<=(X(4) OR X(3)) AND (X(5) XOR Y(7));
PP_2(8)<=(X(4) OR X(3)) AND (X(5) XOR Y(8));
PP_2(9)<=(X(4) OR X(3)) AND (X(5) XOR Y(9));
PP_2(10)<=(X(4) OR X(3)) AND (X(5) XOR Y(10));

PP_3(0)<=(X(6) OR X(5)) AND (X(7) XOR Y(0));
PP_3(1)<=(X(6) OR X(5)) AND (X(7) XOR Y(1));
PP_3(2)<=(X(6) OR X(5)) AND (X(7) XOR Y(2));
PP_3(3)<=(X(6) OR X(5)) AND (X(7) XOR Y(3));
PP_3(4)<=(X(6) OR X(5)) AND (X(7) XOR Y(4));
PP_3(5)<=(X(6) OR X(5)) AND (X(7) XOR Y(5));
PP_3(6)<=(X(6) OR X(5)) AND (X(7) XOR Y(6));
PP_3(7)<=(X(6) OR X(5)) AND (X(7) XOR Y(7));
PP_3(8)<=(X(6) OR X(5)) AND (X(7) XOR Y(8));
PP_3(9)<=(X(6) OR X(5)) AND (X(7) XOR Y(9));
PP_3(10)<=(X(6) OR X(5)) AND (X(7) XOR Y(10));

PP_4(0)<=(X(8) OR X(7)) AND (X(9) XOR Y(0));
PP_4(1)<=(X(8) OR X(7)) AND (X(9) XOR Y(1));
PP_4(2)<=(X(8) OR X(7)) AND (X(9) XOR Y(2));
PP_4(3)<=(X(8) OR X(7)) AND (X(9) XOR Y(3));
PP_4(4)<=(X(8) OR X(7)) AND (X(9) XOR Y(4));
PP_4(5)<=(X(8) OR X(7)) AND (X(9) XOR Y(5));
PP_4(6)<=(X(8) OR X(7)) AND (X(9) XOR Y(6));
PP_4(7)<=(X(8) OR X(7)) AND (X(9) XOR Y(7));
PP_4(8)<=(X(8) OR X(7)) AND (X(9) XOR Y(8));
PP_4(9)<=(X(8) OR X(7)) AND (X(9) XOR Y(9));
PP_4(10)<=(X(8) OR X(7)) AND (X(9) XOR Y(10));

PP_5(0)<=(X(10) OR X(9)) AND (X(10) XOR Y(0));
PP_5(1)<=(X(10) OR X(9)) AND (X(10) XOR Y(1));
PP_5(2)<=(X(10) OR X(9)) AND (X(10) XOR Y(2));
PP_5(3)<=(X(10) OR X(9)) AND (X(10) XOR Y(3));
PP_5(4)<=(X(10) OR X(9)) AND (X(10) XOR Y(4));
PP_5(5)<=(X(10) OR X(9)) AND (X(10) XOR Y(5));
PP_5(6)<=(X(10) OR X(9)) AND (X(10) XOR Y(6));
PP_5(7)<=(X(10) OR X(9)) AND (X(10) XOR Y(7));
PP_5(8)<=(X(10) OR X(9)) AND (X(10) XOR Y(8));
```

```
PP_5(9)<=(X(10) OR X(9)) AND (X(10) XOR Y(9));
PP_5(10)<=(X(10) OR X(9)) AND (X(10) XOR Y(10));


END beha;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY VERSION3 IS
PORT(  X,Y: IN STD_LOGIC_VECTOR ( 10 DOWNTO 0);
           PRODUCT : OUT STD_LOGIC_VECTOR(10 DOWNTO 0)
                );
END ENTITY;

ARCHITECTURE beha OF VERSION3 IS

COMPONENT half_adder
PORT (  x, y : IN STD_LOGIC ;
        s, Cout : OUT STD_LOGIC ) ;

END COMPONENT;
--fistlvl_FA1: fulladd PORT MAP (x=>,y=>,Cin=>,s=>,Cout=>);
COMPONENT fulladd
PORT ( Cin, x, y : IN STD_LOGIC ;
s, Cout : OUT STD_LOGIC ) ;
END COMPONENT ;

--compressor_4to2(A=>,B=>,C=>,D=>,CARRY=>,SUM=>);
COMPONENT compressor_4to2
PORT ( A,B,C,D: IN STD_LOGIC;
           CARRY: OUT STD_LOGIC;
           SUM: OUT STD_LOGIC
                );
END COMPONENT;


COMPONENT PP_generator
PORT ( X, Y : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
           PP_0,PP_1,PP_2,PP_3,PP_4,PP_5: OUT STD_LOGIC_VECTOR(10 DOWNTO 0)
                );
END COMPONENT;

SIGNAL a_PP,b_PP,c_PP,d_PP,e_PP,f_PP : STD_LOGIC_VECTOR (10 DOWNTO 0);
SIGNAL not_a_PP_10,not_b_PP_9, not_b_PP_10, not_c_PP_10, not_c_PP_9,
 not_d_PP_10, not_d_PP_9, not_e_PP_10, not_e_PP_9, not_f_PP_10,
 not_f_PP_9 : STD_LOGIC;

SIGNAL sum_w0,sum_w1,sum_w2,sum_w3,sum_w4 : STD_LOGIC;
SIGNAL carry_t0,carry_t1,carry_t2,carry_t3,carry_t4 : STD_LOGIC;

SIGNAL sum_k0,sum_k1,sum_k2,sum_k3,sum_k4,sum_k5,sum_k6,
 sum_k7, sum_k8 : STD_LOGIC;
```

```
SIGNAL carry_r0,carry_r1,carry_r2,carry_r3,carry_r4,
 carry_r5,carry_r6,
 carry_r7, carry_r8 : STD_LOGIC;

SIGNAL factor1,factor2: STD_LOGIC_VECTOR(10 DOWNTO 0);

BEGIN

partial_products : PP_generator
PORT MAP (X=>X,Y=>Y, PP_0=> a_PP,PP_1=>b_PP,PP_2=>c_PP,
PP_3=>d_PP,PP_4=>e_PP,PP_5=>f_PP);

firstlvl_comp1: compressor_4to2
PORT MAP (A=>c_PP(7),B=>d_PP(5),C=>e_PP(3),D=>f_PP(1)
,CARRY=>carry_t1,SUM=>sum_w1);

firstlvl_comp2: compressor_4to2
PORT MAP(A=>c_PP(8),B=>d_PP(6),C=>e_PP(4),D=>f_PP(2)
,CARRY=>carry_t2,SUM=>sum_w2);

fistlvl_FA1: fulladd
 PORT MAP (x=>d_PP(4),y=>e_PP(2),Cin=>f_PP(0),s=>sum_w0
 ,Cout=>carry_t0);

fistlvl_FA2: fulladd
PORT MAP (x=>d_PP(8),y=>e_PP(6),Cin=>f_PP(4),s=>sum_w4
,Cout=>carry_t4);

firstlvl_HA : half_adder
PORT MAP(x=>e_PP(5),y=>f_PP(3),s=>sum_w3,Cout=>carry_t3);


secondlvl_comp2:compressor_4to2
 PORT MAP(A=>carry_t0,B=>not_a_PP_10,
 C=>not_b_PP_9,D=>sum_w1,CARRY=>carry_r1,SUM=>sum_k1);
secondlvl_comp3:compressor_4to2

PORT MAP(A=>carry_t1,B=>b_PP(9),C=>not_b_PP_10,
D=>sum_w2,CARRY=>carry_r2,SUM=>sum_k2);

secondlvl_comp4:compressor_4to2
PORT MAP(A=>carry_t2,B=>not_c_PP_9,C=>d_PP(7),
D=>sum_w3,CARRY=>carry_r3,SUM=>sum_k3);

secondlvl_comp5:compressor_4to2
PORT MAP(A=>carry_t3,B=>c_PP(9),C=>not_c_PP_10,
D=>sum_w4,CARRY=>carry_r4,SUM=>sum_k4);

secondlvl_comp6:compressor_4to2
PORT MAP(A=>carry_t4,B=>not_d_PP_9,C=>e_PP(7),
D=>f_PP(5),CARRY=>carry_r5,SUM=>sum_k5);

secondlvl_comp7:compressor_4to2
```

```
PORT MAP(A=>not_d_PP_10,B=>d_PP(9),C=>e_PP(8),
D=>f_PP(6),CARRY=>carry_r6,SUM=>sum_k6);

secondlvl_FA1: fulladd
PORT MAP (x=>not_e_PP_10,y=>e_PP(9),Cin=>f_PP(8),
s=>sum_k8,Cout=>carry_r8);

secondlvl_FA2: fulladd
 PORT MAP(x=>b_PP(8),y=>c_PP(6),Cin=>sum_w0,
 Cout=>carry_r0,s=>sum_k0);

secondlvl_HA: half_adder
 PORT MAP(x=>not_e_PP_9,y=>f_PP(7),s=>sum_k7,
 Cout=> carry_r7);

not_a_PP_10<=NOT(a_PP(10));
not_b_PP_10<=NOT(b_PP(10));
not_c_PP_10<=NOT(c_PP(10));
not_d_PP_10<=NOT(d_PP(10));
not_e_PP_10<=NOT(e_PP(10));
not_f_PP_10<=NOT(f_PP(10));
not_b_PP_9<=NOT(b_PP(9));
not_c_PP_9<=NOT(c_PP(9));
not_d_PP_9<=NOT(d_PP(9));
not_e_PP_9<=NOT(e_PP(9));
not_f_PP_9<=NOT(f_PP(9));

factor1<=f_PP(9)& carry_r8 & carry_r7 &
carry_r6 &
 carry_r5 & carry_r4 & carry_r3 & carry_r2 &
 carry_r1 &
 carry_r0 & a_PP(10);

factor2<= not_f_PP_10 & not_f_PP_9 & sum_k8
& sum_k7 &
 sum_k6 & sum_k5 & sum_k4 & sum_k3 & sum_k2
 & sum_k1 &
 sum_k0;

PRODUCT<= STD_LOGIC_VECTOR(SIGNED(factor1)+SIGNED(factor2));

END beha;
```

Dadda Tree Synopsys script

```
analyze -f vhdl -lib WORK ../src/half_adder.vhd
analyze -f vhdl -lib WORK ../src/fulladd.vhd
analyze -f vhdl -lib WORK ../src/mux_2to1_12bit.vhd
analyze -f vhdl -lib WORK ../src/reorda_daddatree_sum.vhd
analyze -f vhdl -lib WORK ../src/Recording_Logic.vhd
analyze -f vhdl -lib WORK ../src/signed_multiply_MBR.vhd

analyze -f vhdl -lib WORK ../src/adder_fixedpoint.vhd
analyze -f vhdl -lib WORK ../src/reg_11bit.vhd
```

```
analyze -f vhdl -lib WORK ../src/datapath.vhd
analyze -f vhdl -lib WORK ../src/control_unit.vhd

set power_preserve_rtl_hier_names true
elaborate control_unit -arch behavior -lib WORK

create_clock -name MY_CLK -period 3.17 CLK
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] CLK]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set load $OLOAD [all_outputs]
ungroup -all -flatten
compile
report_timing > report_timing_ck3_17
report_area > report_area_ck3_17
```