Politecnico di Torino

Corso di Laurea Magistrale in
INGEGNERIA ELETTRONICA

INTEGRATED SYSTEMS ARCHITECTURE

Special Project - Squareroot Unit

Professors:
Prof. Guido Masera
Prof. Maurizio Martina

Students:
Christian Vespo 292674
Claudio Raccomandato 290190
Pietro Romeo 269010

Anno Accademico 2021/2022

Special Project

In this Special Project laboratory, the goal is to develop a special function unit performing square roots for the RISC-V processor. This was done in three steps:

- **Development of a simplified RISC-V processor**, following the instructions of the first part of Lab 3
- **Development of the Square Root Unit**
- **Considerations about the architecture in its whole**, considering both the Processor and the square root unit, with a proposal for its implementation and the related new instructions

Contents

1 RISC-V development

1.1 Understanding the Instruction Set

The very first step to be done in order to develop the RISC-V processor is to understand exactly how the instructions work, and how they are processed through the five pipeline stages. This was done with the aid of the provided assembly program, as well as referring to the generic architecture provided during the lectures and shown in Figure 1.

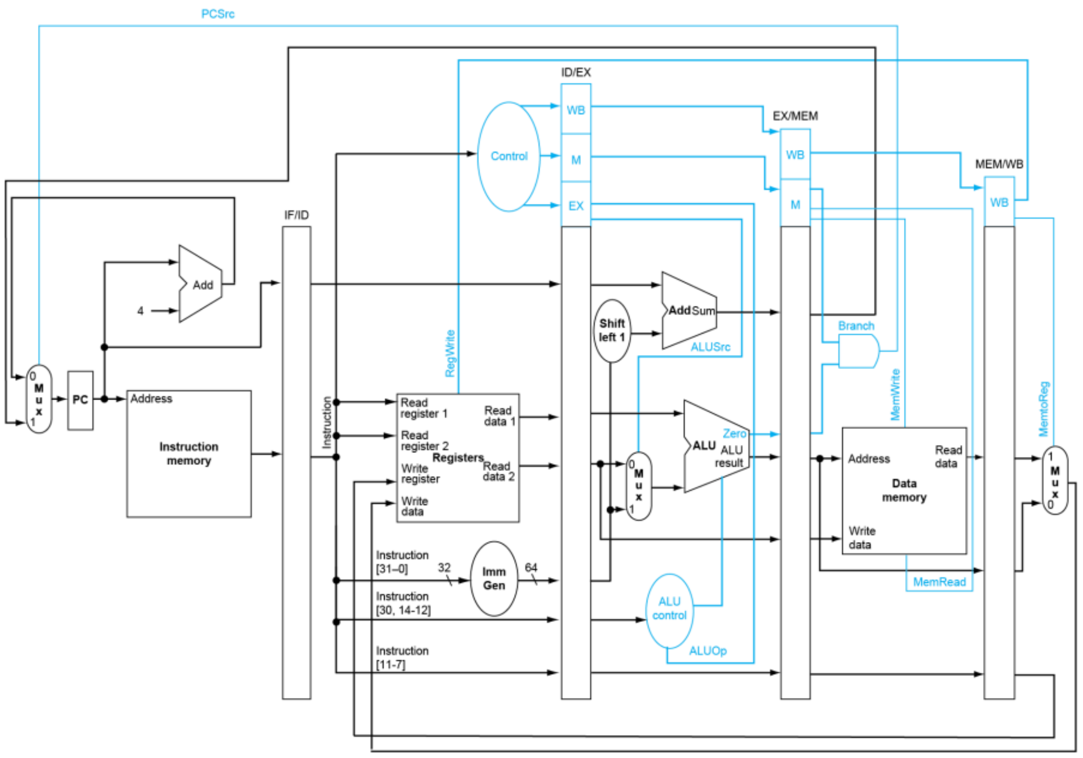


Figure 1: Reference Architecture

1.2 Control Unit and ALU control

As a second step, truth tables for the control unit and the ALU control have been chosen. The results can be seen in Table 1 and Table 2.

OPCODE	Name	Jump	Branch	MemRead	MemWrite	Alusrc	RegWrite	Aluop
0110011	ADD	0	0	0	0	00	1	11
0110011	SLT	0	0	0	0	00	1	11
0110011	XOR	0	0	0	0	00	1	11
0100011	SW	0	0	0	1	01	0	00
1100011	BEQ	0	1	-	0	00	0	01
0000011	LW	0	0	1	0	01	1	00
0010011	ADDI	0	0	0	0	01	1	00
0010011	ANDI	0	0	0	0	01	1	00
0010011	SRAI	0	0	0	0	01	1	00
0010111	AUIPC	0	0	0	0	10	1	10
0110111	LUI	0	0	0	0	01	1	10
1101111	JAL	1	0	0	0	11	1	10

Table 1: Control Unit

Instruction Name	Bit 30	F3	ALUop	ALUctrl	ALU operation
ADD	0	000	11	100	Sum
SLT	0	010	11	001	Greater Comparator
XOR	0	100	11	010	XOR
SW/LW	-	010	00	100	Sum
BEQ	-	000	01	101	Equality Comparator
ADDI	-	000	00	100	Sum
ANDI	-	111	00	011	AND
SRAI	1	101	00	000	Arithmetic Shift
AUIPC/LUI/JAL	-	IMM	10	100	Sum

Table 2: ALU control

The Control unit receives the OPCODE of the current instruction in Stage 2, and based on that sends various control signals to the other stages.

Since different instructions may have the same OPCODE but require the ALU to perform different operations, the Control Unit does not select the ALU operation: that is the job of the ALU Control, which receives in input the F3 field, the 30th bit and ALUop. The F3 field generally indicates the operation to be performed by the ALU (for example, ADD and ADDI both perform an addition and have the same F3). Still, there are a few situations where F3 is not meaningful (AUIPC/LUI/JAL) or it is the same for two instructions that require the ALU to perform different operations (BEQ and ADDI for example). In these cases, the ALUop (provided by the Control Unit) comes in aid. The 30th bit is only needed for the SRAI operation.

1.3 Immediate Generator

Another fundamental component of the RISC-V is the Immediate Generator. Its role is that of recognizing the instruction format by looking at the OPCODE and unpack the Immediate fields accordingly, providing it in output on a 32-bit format. The behavior of the Immediate Generator is represented in Table 3.

OPCODE	Name	OUTPUT
0100011	SW	IN[31]$_{31:12}$ & IN[31:25] & IN[11:7]
1100011	BEQ	IN[31]$_{31:12}$ & IN[31] & IN[7] & IN[30:25] & IN[11:8]
0000011	LW	IN[31]$_{31:12}$ & IN[31:20]
0010011	ADDI/ANDI/SRAI	IN[31]$_{31:12}$ & IN[31:20]
0010111	AUIPC	IN[31:12] & X"000"
0110111	LUI	IN[31:12] & X"000"
1101111	JAL	IN[31]$_{31:20}$ & IN[31] & IN[19:12] & IN[20] & IN[30:21]
0110011	ADD/SLT/XOR	X"00000000"

Table 3: Immediate Generator

Immediates are generally put in the least significant bits of the output bus, and the other bits are filled with the sign-extension bit (which is the 31th bit of the instruction). The immediates in instructions LUI and AUIPC are "Upper", meaning they are put in the most significant bits and the lower bits are filled with zeros.

1.4 Hazards

This RISC-V pipelined architecture is affected by two types of hazard: control hazards and data hazards.

Control hazards have been addressed by adding NOPs and stalling the program counter whenever a BEQ or JAL instructions are performed. No other operation is performed until BEQ or JAL are finished. Such approach would have been highly detrimental in a processor with a deeper pipeline, in which case a more complex branch-prediction unit would be compulsory. Having only five stages though, the trade-off between complexity and performance was considered acceptable.

Data Hazards have been addressed with a Forwarding Unit (FU) that pilots two multiplexers, which allows the ALU in the third stage to use results that are currently in stages four and five whenever an instruction calls for an operation whose operands have not yet been stored in the register file. In order to do so, the FU receives the addresses of the source registers in stage three, as well as the addresses of destination registers in stage 4 and 5. There are a few instructions where destination registers are not addresses, but instead are Immediates, and it is for this reason that the FU also receives the WriteReg command from stages four and five. The behavior of the FU is partially described by Table 4.

INPUT	OUTPUT	Data
RD_WB $\neq$ RS1 & RD_MEM $\neq$ RS1	00	RD1
RD_WB $\neq$ RS1 & RD_MEM = RS1	01	ALU_MEM
RD_WB = RS1 & RD_MEM $\neq$ RS1	10	ALU_WB
RD_WB = RS1 & RD_MEM = RS1 & (RW_WB = 1 & RW_MEM = 1)	01	ALU_MEM
RD_WB = RS1 & RD_MEM = RS1 & (RW_WB = 0 || RW_MEM = 0)	10	ALU_WB

Table 4: Forwarding unit Mux 1

Table 4 represents half of the FU, meaning only the selector for the Multiplexer for Register 1. The behavior of the other selector is easily derived by substituting RS1 with RS2 in the Table. Underscores in the Table are used to indicate the stage from which the signal is coming from: WB is the fifth stage and MEM is the fourth. RD refers to the destination register, and RW to the RegWrite signal. ALU refers to the result of the operation.

1.5 Schematics and Architectural Choices

The architecture of the developed RISC-V is represented in Figure 2 to 5, stage by stage. All the main blocks introduced previously are present, as well as a few minor tweaks with respect to the reference architecture in Figure 1.

The architecture represented is a faithful representation of the entities in the code, with a few exceptions: a few names have been changed, and and some of the components have been described in the Top entity rather than in the stages. Such components are the Program Counter and the Forwarding Unit. Also, Instruction and Data memory have been implemented only in the testbench, and therefore have been drawn in dashed lines.

In Stage 1 (Figure 2), changes mostly regard the added multiplexers to both execute NOPs and stall the program counter. The incoming signals Stall_ID, Stall_EX and Stall_MEM are active whenever either JUMP or BRANCH control signals are active in the corresponding stages.

In Stage 2 (Figure 3), it was decided to use a Synchronous Register File and subsequently to “skip” the pipeline for the Read Data signals. The added outputs RS1 and RS2 are needed by the forwarding Unit in the third stage.

In Stage 3 (figure 4), ALU input multiplexers have been cascaded with Forwarding Unit Multiplexers. The added MUX in the Fourth Stage (Figure 5) mirrors the one in Stage 5 and its output is needed by the Forwarding Unit.

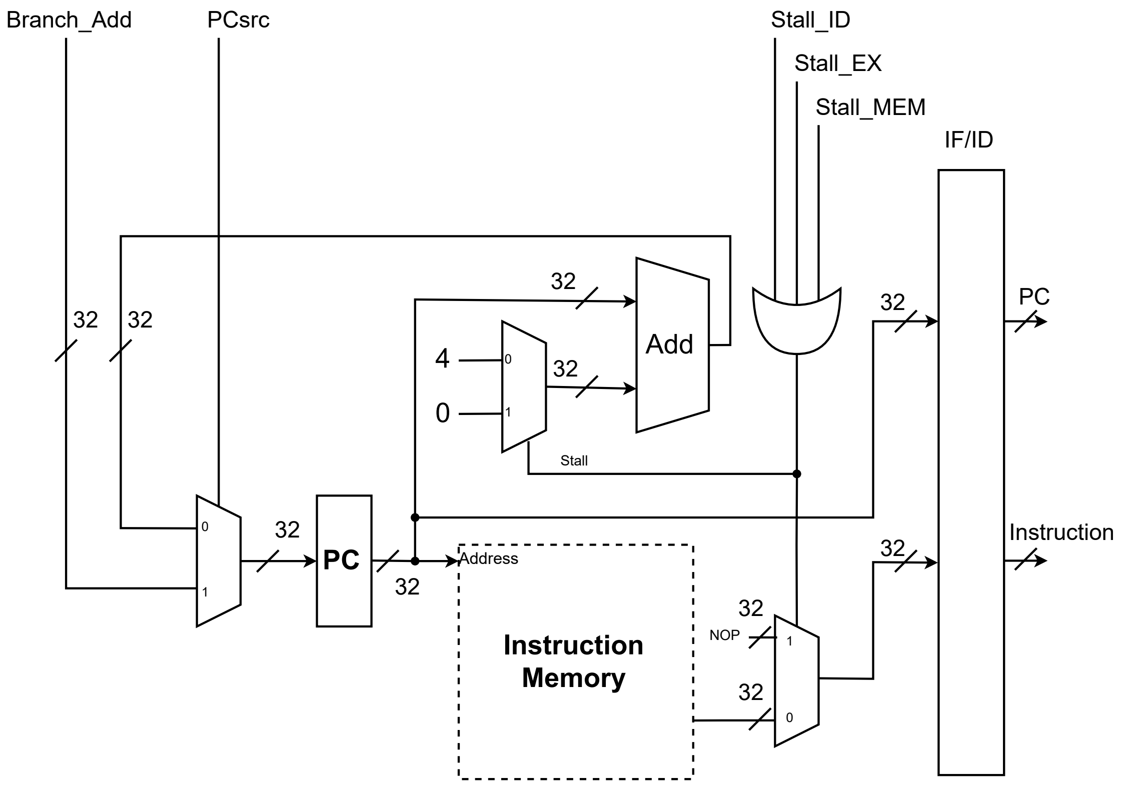


Figure 2: First Stage

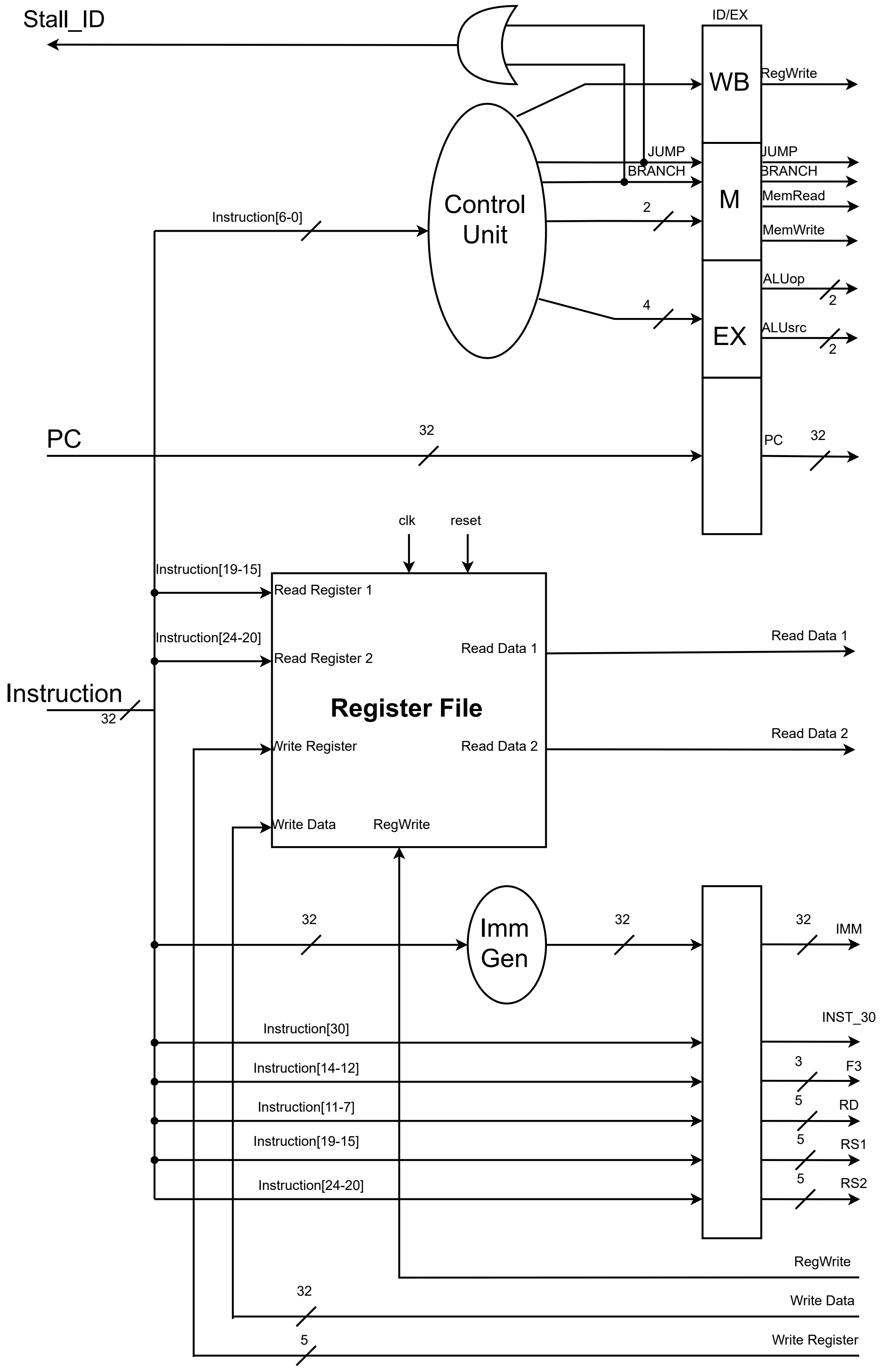


Figure 3: Second Stage

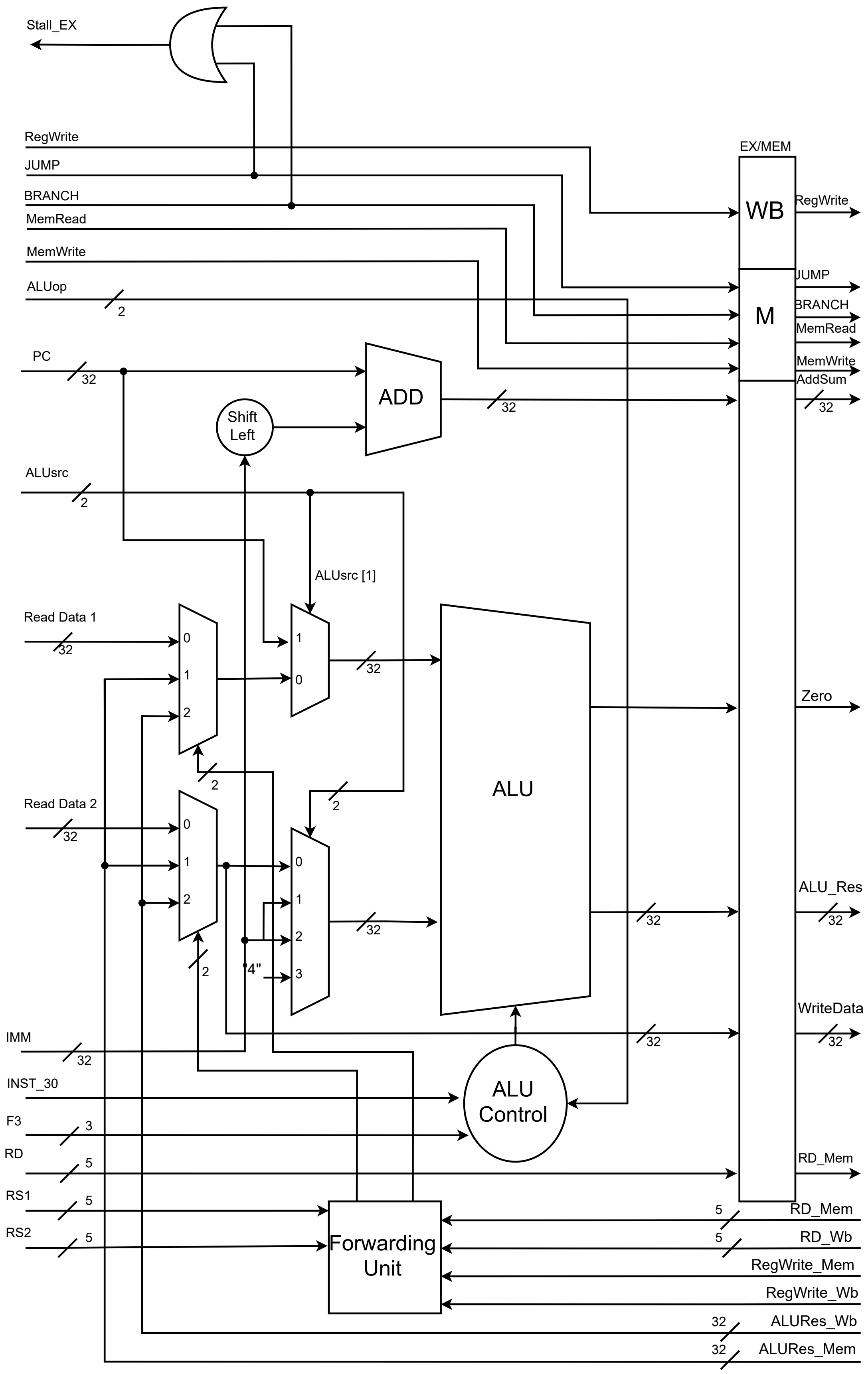


Figure 4: Third Stage

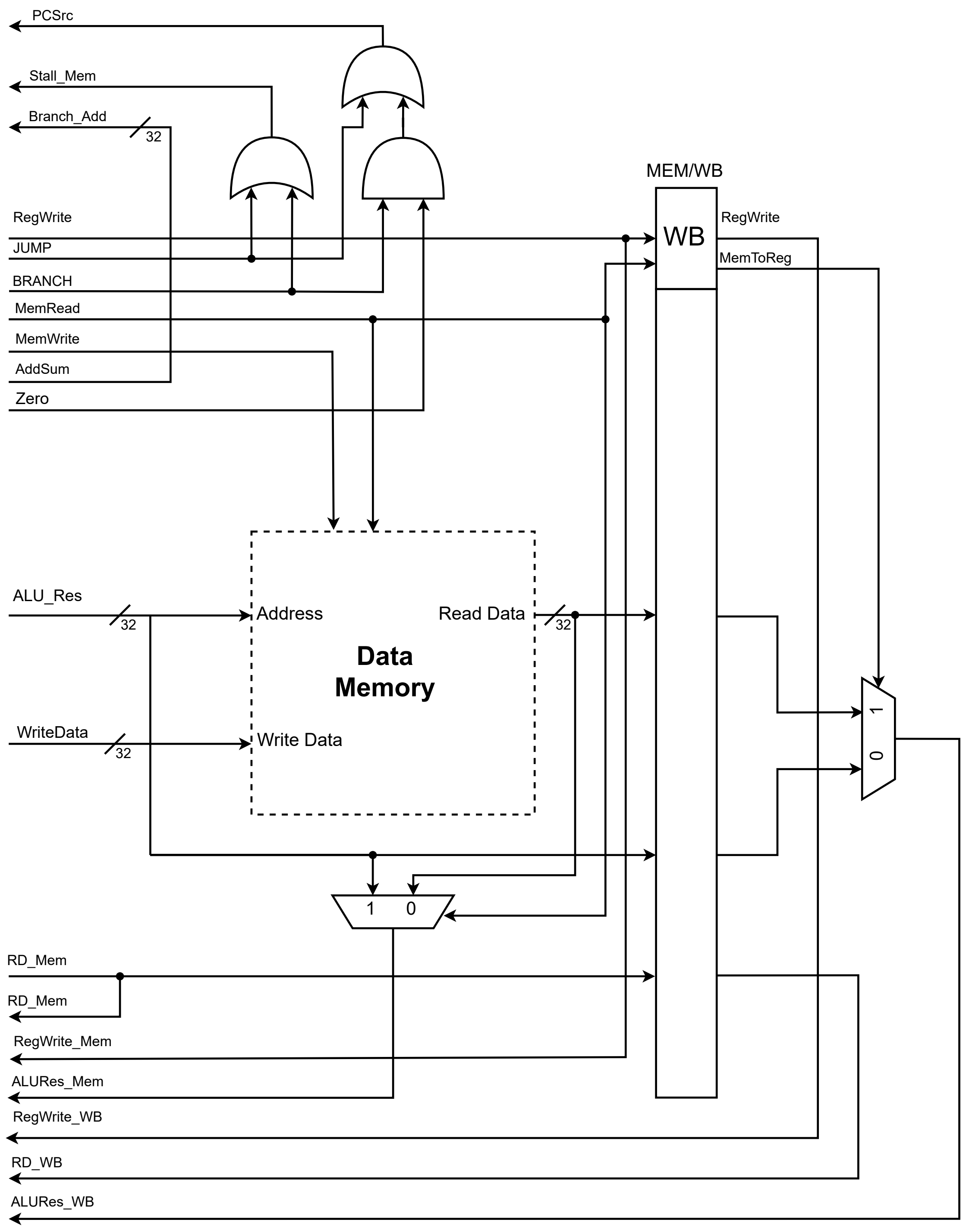


Figure 5: Fourth and Fifth Stage

1.6 Verification and Simulation

Verification at this stage has been performed with a VHDL testbench that implements the same sequence of instructions as those provided with RARS. In order to verify the correct behavior, the contents of the register file were compared to the ones in RARS.

1.7 Memories Implementation

Data and Instruction Memory have been implemented outside the main architecture, as components for the testbench. Instead of writing the machine code directly into the RAM and ROM VHDL files, an initialization function has been implemented. It needs a text file containing the memory contents in textual binary format. During the simulation, the function will take care of initializing the component with the desired content, without having to recompile the VHDL file. For safety, the unused part of the instruction memory has been filled with NOPs.

1.8 Performance Evaluation

The RISC-V performance was evaluated by using Design Compiler and Innovus. The reports generated with these tools have provided the results in Table 5.

	Area [μm^2]	Critical Path Delay [ns]	Power Consumption [mW]
Design Compiler	15159	3.38	3.46
Innovus	13919	3.32	4.979

Table 5: Performance of Developed RISC-V

The ALU, and the Barrel Shifter especially, is the bottleneck of the architecture, meaning it constitutes the critical path. The Squareroot Unit that will be developed will have to have a shorter delay than the ALU. The performance of the ALU alone is reported in Table 6.

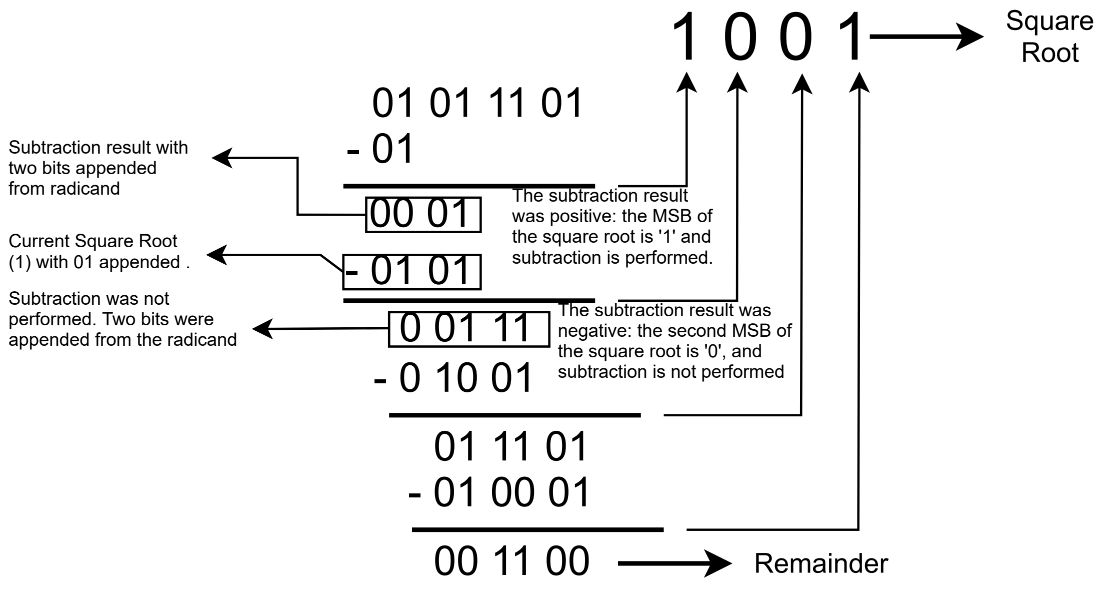
	Area [μm^2]	Critical Path Delay [ns]	Power Consumption [mW]
Design Compiler	863.44	3.26	0.503

Table 6: Performance of the ALU

2 Squareroot Unit Development

2.1 Algorithm choice

The first step undertaken in the search for a Squareroot unit for the RISC-V was to read previous researches on the matter (1), (2), (3). The special unit would preferably be non-pipelined, so as to make it work parallel to the ALU, and as compact and fast as possible, so as not to slow the execution. These criteria led to choose the modified non restoring digit-by-digit calculation algorithm proposed in (1).



1 0 0 1
Square Root
01 01 11 01
- 01
Subtraction result with two bits appended from radicand
00 01
The subtraction result was positive: the MSB of the square root is '1' and subtraction is performed.
Current Square Root (1) with 01 appended .
- 01 01
Subtraction was not performed. Two bits were appended from the radicand
0 01 11
The subtraction result was negative: the second MSB of the square root is '0', and subtraction is not performed
- 0 10 01
01 11 01
- 01 00 01
00 11 00
Remainder

Figure 6: Algorithm Example

The algorithm is quite straightforward and easily reproduced with pen and paper. It consists, in short, in a series of subtractions and multiplexing. A detailed example is provided in Figure 6, with an 8 bit radicand. The generic architecture that realizes this algorithm is represented in Figure 7.

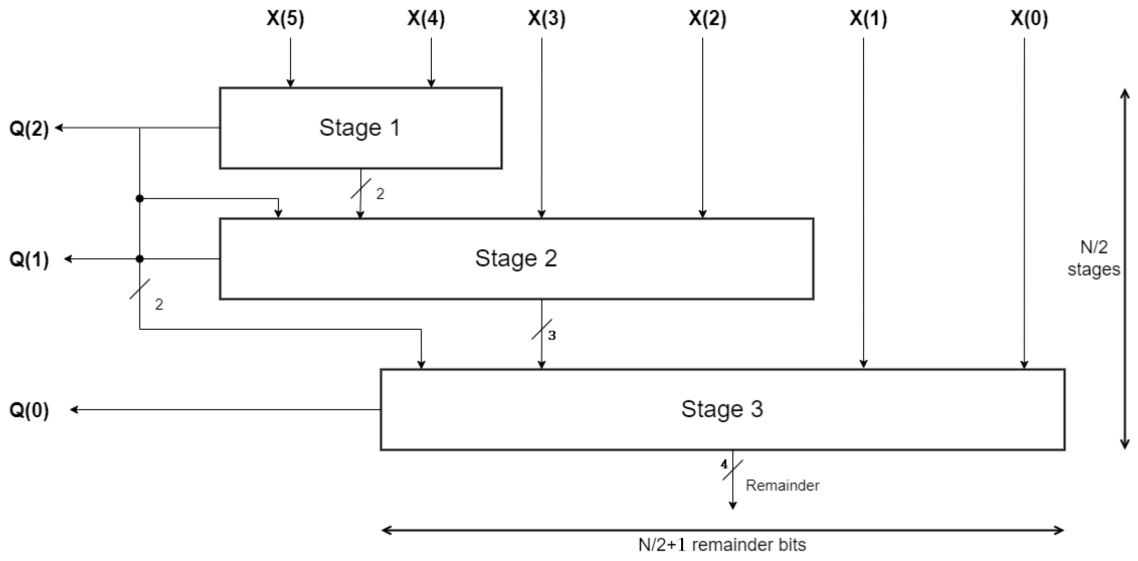


Figure 7: Generic Architecture for 6 bit inputs

Every stage receives in input two bits from the radicand, the result from the previous stage, and a string containing the bits of the square root calculated in the previous stages. The Stages include a subtractor and a multiplexer, and the square root bits are generated by inverting the leftmost carry out (or borrow out) of the subtractor. For a given parallelism N, the Square Root unit produces $\frac{N}{2}$ bits for the Square Root, with each bit being generated by a stage and the most significant bit being generated by the first stage. The first stage performs a subtraction on 2-bits operands, while the generic stage i performs subtraction on $i+2$ bits and yields the subtraction result on $i+1$ bits.

2.2 Remainder

It was decided to produce as outputs of the whole structure both the square root and the remainder. The remainder requires $\frac{N}{2}+1$ bits. In fact, the maximum remainder that is obtained for a parallelism of N is equal to $2^{\frac{N}{2}+1}-2$. The maximum remainder is obtained when the radicand is the highest number that can be represented, meaning 2^N-1.

2.3 Subtractor Implementation

With the goal in mind to realize a square root unit with a 32 bit input (and thus being able to perform the square root of the full content of a register in the RISC-V), long ripple carry adders (or subtractors) are required to perform the algorithm, that would bring the delay to a very high amount. Therefore, a carry select architecture was chosen for the subtractor present in each stage, since its critical path passes through a smaller Ripple Carry element and a series of multiplexers.

The Carry Select architecture was realized in two ways: with a Carry Select Adder and with a Carry Select Subtractor. Also, the size of each block is variable, and its choice will be discussed in a later paragraph.

2.4 Carry Select Adder implementation

The Carry Select Adder implementation is parametric, the parameters being the parallelism of the inputs and the number of bits per block, with the first greater than or equal to the second. Each block computes a part of the sum and, with the exception of the first that is composed by a single RCA, it consists of two RCAs and a multiplexer.

Moreover, if the parallelism of the inputs is divisible by the number of bits for block, each block will have the same parallelism. Otherwise, all the blocks will have the same parallelism, equal to the number of bits for block, except the last one, that will have a parallelism equal to the number of remaining bits (which are less than the number of bits for block by definition).

An example of Carry Select Adder implementation is shown in Figure 8, where the parallelism of the inputs is equal to 11, and the number of bits per block is 3. This means that the Ripple Carry Adders for the first three blocks have a parallelism of 3 bits, instead for those present in the last block it is equal to 2, that corresponds to the number of remaining bits.

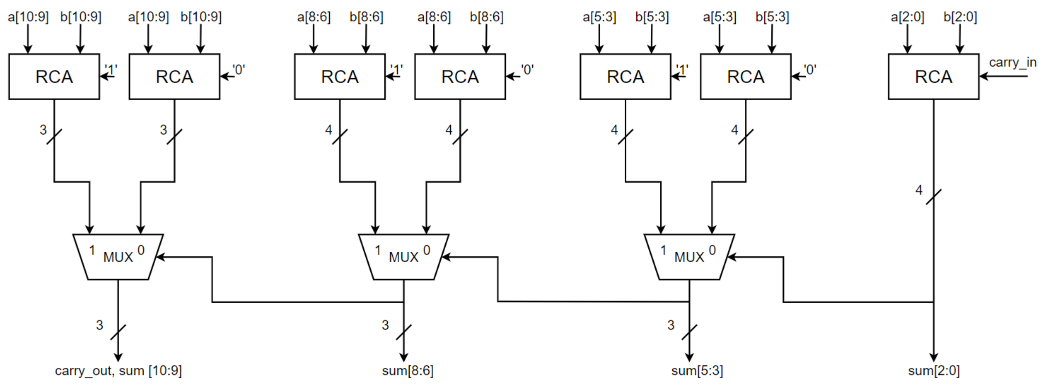


Figure 8: Carry Select Adder

In this implementation the ripple carry structure is generated with a Full Adder chain, converting the addition to a subtraction by inverting the input that has to be subtracted and by adding '1' through the carry in of the first block. This implementations serves mostly as a starting point for the more optimized Carry Select Subtractor.

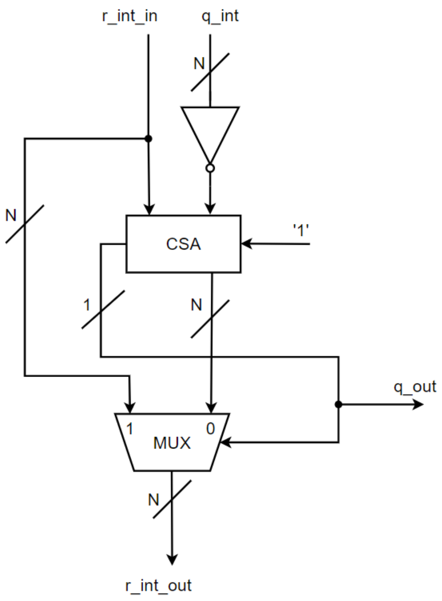


Figure 9: CSA Stage

2.5 Carry Select Subtractor implementation

This solution involves the same Carry Select architecture, but with the more specialized Full Subtractor cells optimized for the algorithm. It must be said that, while in article (1) such cells are present as well, they are simply wrong and therefore have been rebuilt from the truth table of a generic Full Subtractor.

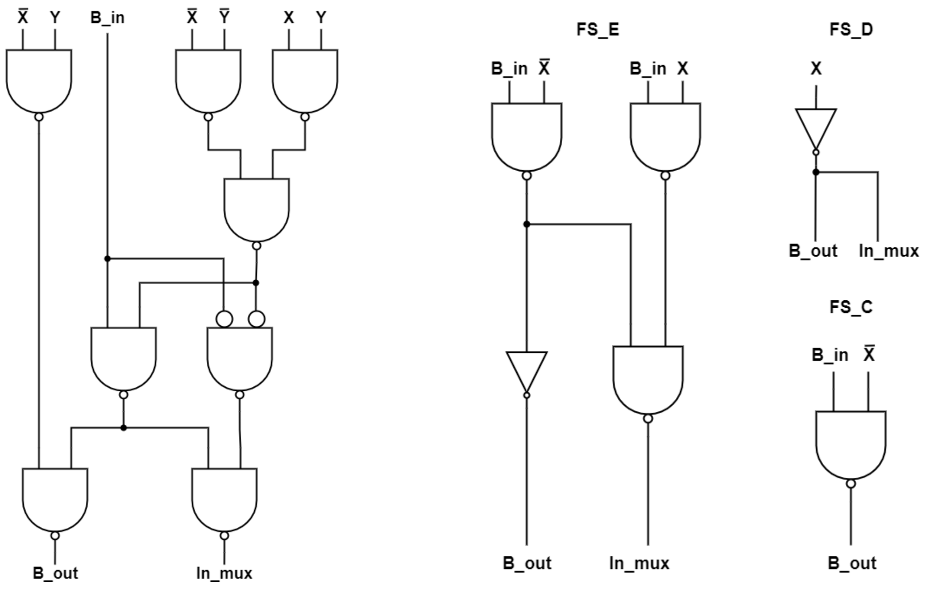


Figure 10: Full Subtractor (left) and Subtractors D, E and C (right)

Figure 10 represents the Full Subtractor cells realized with NAND gates and inverters. The advantage it has over the Full Adder cell is both the reduced area and the reduced input to borrow out delay. Cells D and E are simplified versions of a Full subtractor, due to them subtracting always the same quantity (the appended 01 shown in figure 6), and block D not receiving a borrow-in. Cell C is the one responsible of subtracting the most significant bit and generating the square root bit of the stage. The RCS elements in Figure 11 are composed of a series of Full Subtractors.

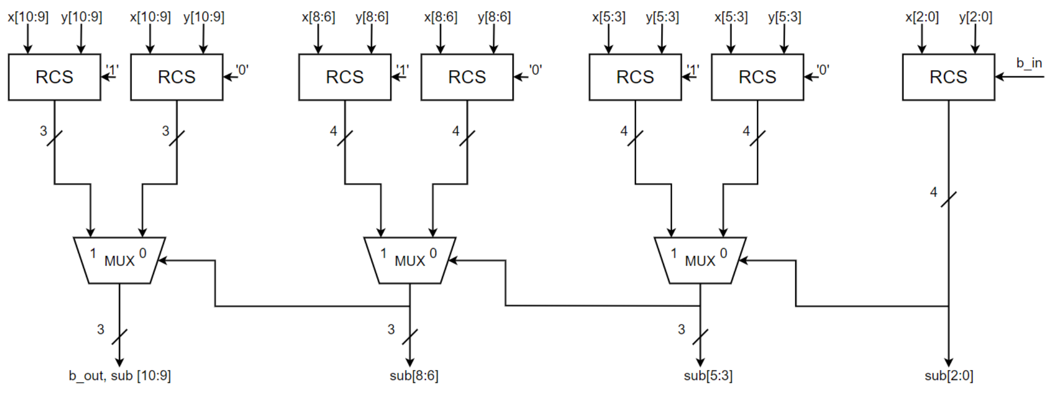


Figure 11: Carry Select Subtractor

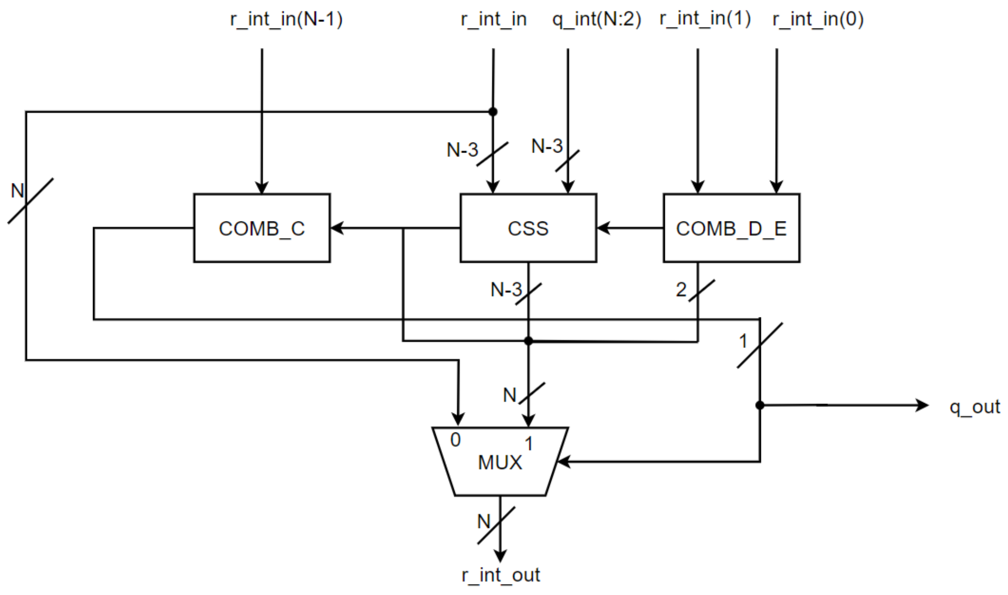
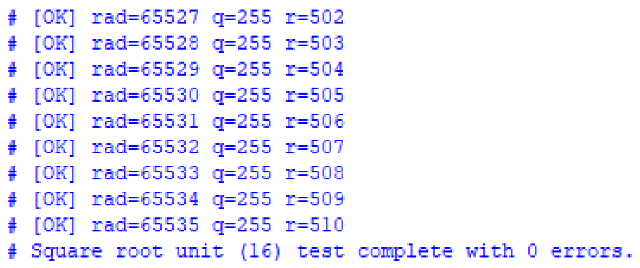


Figure 12: CSS Stage

Figure 12 represents how the CSS stage was implemented. The only difference with respect to the CSA stage (Figure 9) is that the first 2 LSB are calculated by D and E blocks and the MSB by the C block.

2.6 Square Root Verification

The verification took place using a program written in System Verilog which evaluates the results obtained by the square root unit using all possible combinations of inputs and checks if the equation $radicand = quotient^2 + remainder$ is verified. Using a configuration block it was then possible to easily test the three different architectures of the square root stage (BEH, CSA and CSS). An example of the verification results is shown below.



```
# [OK] rad=65527 q=255 r=502
# [OK] rad=65528 q=255 r=503
# [OK] rad=65529 q=255 r=504
# [OK] rad=65530 q=255 r=505
# [OK] rad=65531 q=255 r=506
# [OK] rad=65532 q=255 r=507
# [OK] rad=65533 q=255 r=508
# [OK] rad=65534 q=255 r=509
# [OK] rad=65535 q=255 r=510
# Square root unit (16) test complete with 0 errors.
```

Figure 13: tb_square_root.sv output results

2.7 Block size Optimization

As previously shown, the carry select architecture leaves the designer the freedom to choose the size of the blocks (the number of full adders/subtractors elements in each block). In order to choose the optimal block size given the parallelism of the stage, a Python program was employed.

The program generates a source file, to be used by Design Compiler, running simulations for each parallelism and for each block size. Then, it takes the results from each report generated and stores them in a text file. With this data, it is possible to choose the optimal block size for each stage, giving priority to either area or speed. Figure 14 represents these data, reorganized in Matlab generated Plots.

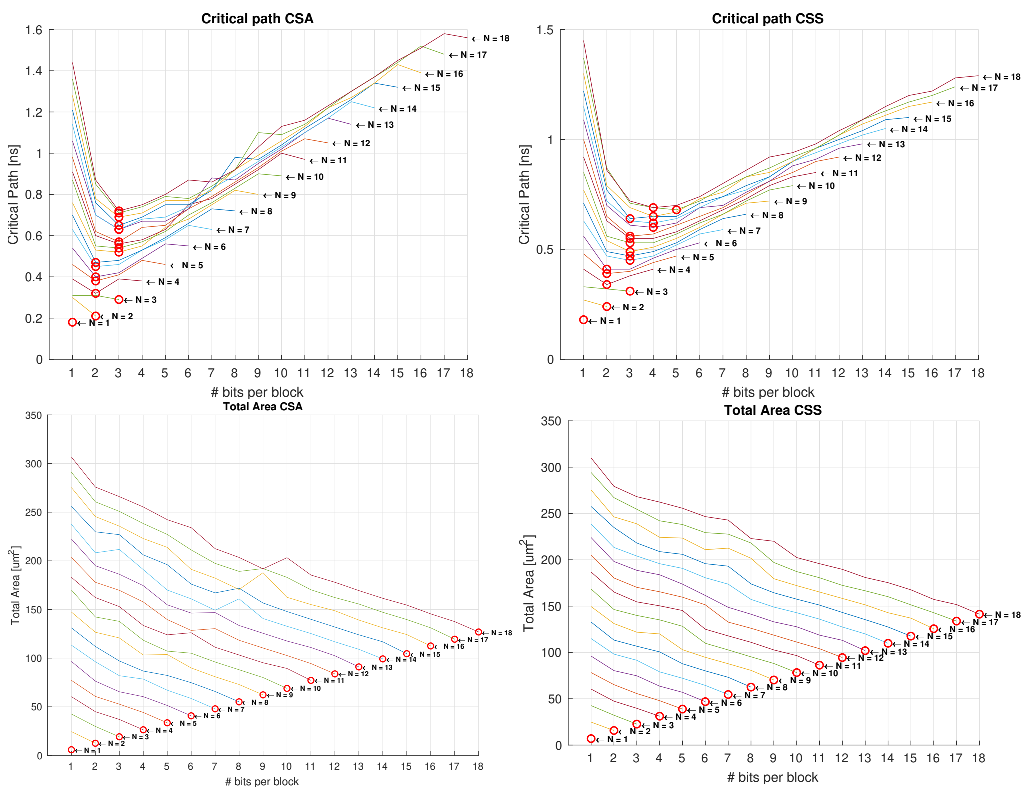


Figure 14: Critical path and area of CSA (left) and CSS (right). The red circles represent the best choice for each parallelism N.

One more variable in these simulations was the presence or not of timing constraints added with Design Compiler. With timing constraints added, the timing is improved at the cost of an increased area. All results shown are obtained with timing constraints.

Finally, *using the optimal number of blocks per stage* (minimizing critical path), square root units at parallelisms of 8, 16 and 32 have been built.

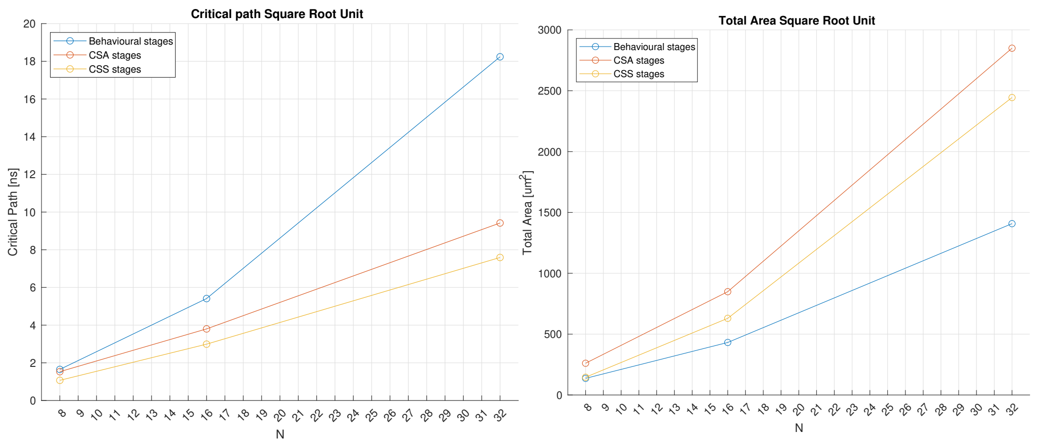


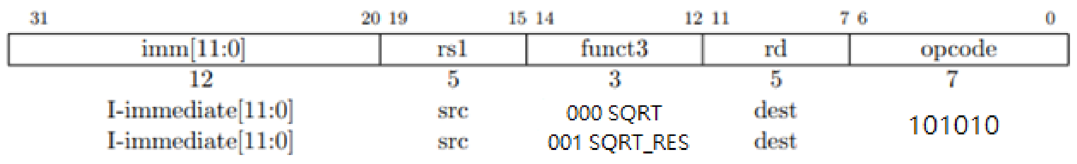
Figure 15: Critical path and area of square root units

3 Squareroot Unit and RISC-V

With the squareroot unit developed, all pieces are in place to make a proposal for its implementation in the RISC-V. The block that contains the Square Root Unit and receive the commands will be referred to as Special Unit.

3.1 Special Unit Instruction and Control

With the goal in mind to provide the possibility to have either the square root or the remainder in output, two instructions with the same OPCODE but different F3 were created. Similarly to how it is done for the ALU, the OPCODE refers to the entire Special Unit and the F3 select a particular operation. In fact when F3 is equal to 000 it computes the square root and when is equal to 001 it computes the remainder. The control unit in the second stage was therefore modified to accomodate them.



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	000 SQRT	dest	101010
I-immediate[11:0]	src	001 SQRT_RES	dest	

Figure 16: Special Unit Operations

The chosen instruction format is Immediate Instruction (the same as ADDI). The OPCODE was chosen arbitrarily among the ones not used in the RISC-V. The special unit itself was implemented in Stage 3, along with a Special Unit Control and a multiplexer, as shown in Figure 17.

The Special Unit control receives the F3 field and uses it to generate a selector for the output multiplexer determining whether the special unit output is the square root or the remainder. With the choice made for the F3 of the new functions, the SPC control is simply a wire with the content of the LSB of the F3.

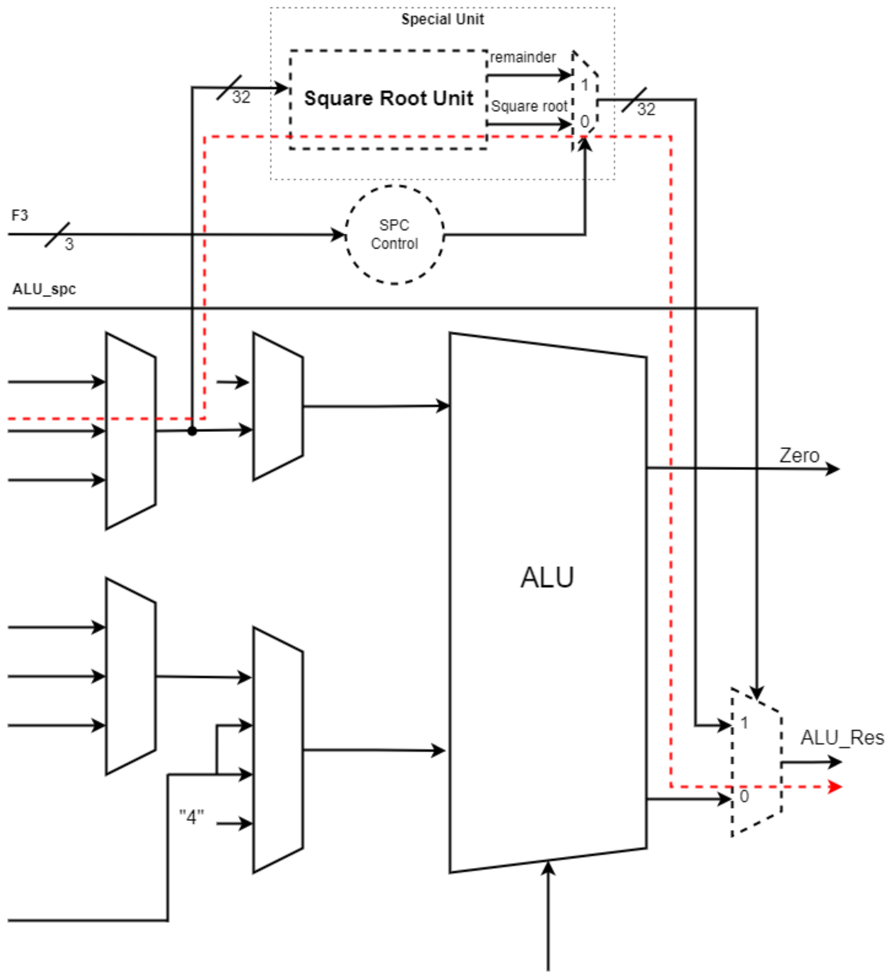


Figure 17: Special Unit Implementation. The red arrow indicates the possible critical path through the special unit of the stage 3.

3.2 Parallelism Choice

The parallelism of the square root unit was chosen with the criterion of not extending the RISC-V critical path. Sadly, the 32-bits and 16-bits architectures are too slow and therefore were discarded.

Simulations of the whole RISC-V, with the added special unit, were performed in order to identify the highest parallelism that does not extend the critical path. Also, since at low parallelisms it does not make sense to keep using the carry select architecture, simulations were also performed with Ripple Carry Subtractors instead of CSSs. This solution corresponds to employing the optimal number of bits per block in terms of area rather than timing, as shown in Figure 14.

Figure 18 represents these data. Aside from the nonsense happening for parallelisms of 6 and 8 bits, which we consider to be an error of the simulator, it can be appreciated how the critical path remains constant and equal to the critical path of the RISC-V alone for architectures equipped with special units of parallelisms of 12 bits and below. The chosen parallelism for the Square Root unit is therefore 12, with a CSS architecture with block sizes optimized for timing. The performance of this architecture only as a square root unit is reported in table 7. When compared to the ALU (table 6), it is considerably faster, occupies less than half the area and consumes less power.

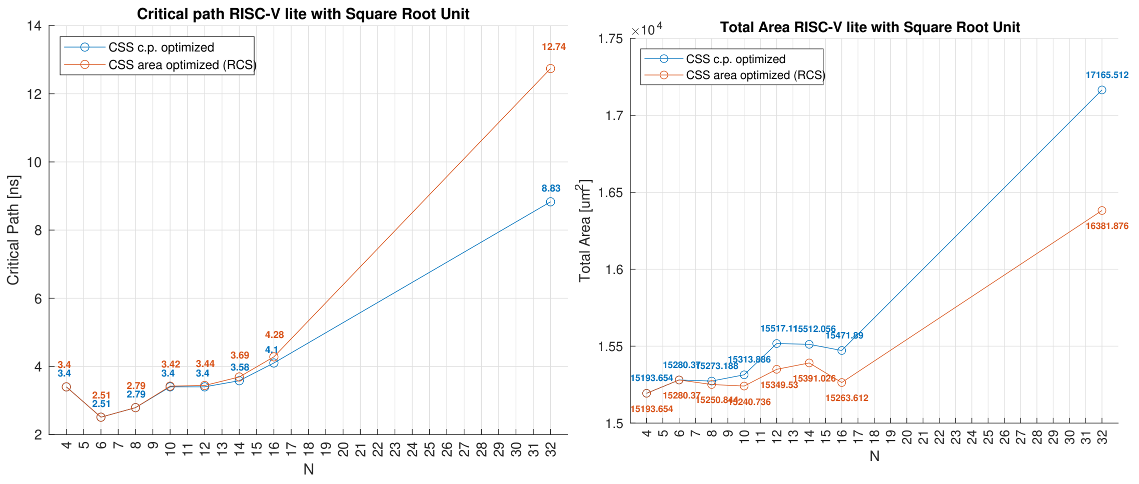


Figure 18: Critical path and area of RISC-V lite with Square Root Unit and CSS architecture

	Area [μm^2]	Critical Path Delay [ns]	Power Consumption [mW]
Design Compiler	352.45	1.94	0.311

Table 7: Performance of 12-bits Squareroot Unit

3.3 Proposed Architecture

The proposed architecture for the square root unit contained in the special unit of the developed RISC-V is therefore with 12-bits parallelism and based on Carry Select Subtractors. This solution fits perfectly for the RISC-V that was developed in this laboratory, however it may vary slightly for other RISC-V processors. For example, a slower processor may safely employ a 16-bits square root unit, or a faster one may employ an 8-bit square root unit.

Table 8 summarizes the characteristics obtained by simulating the 12-bits architecture with both Design Compiler and Innovus.

	Area [μm^2]	Critical Path Delay [ns]	Power Consumption [mW]
Design Compiler	15517	3.4	3.62
Innovus	14210.3	3.383	5.427

Table 8: Performance of RISC-V with Special Unit

3.4 Possible alternative

Would it still be possible to implement a 32-bits Square Root Unit in a RISC-V? In this paragraph, an unorthodox solution is provided as pure speculation.

While implementing a 32-bits Square Root Unit in Stage 3 would triple the critical path, that would not be true if two NOPs were sent after each special unit instruction, using the same system as the one employed for the Control Hazards. This way, while the critical path would nominally be the one passing through the special unit, the actual one would be the one passing through the ALU. This solution would require a multiplexer at the input of the register file to store the result of the square root unit immediately after the second NOP injection.

The added NOPs would indeed be a detriment to the processor, but a negligible one since the square root would be used very rarely in most algorithms. With respect to the 12-bit architecture proposed in the previous paragraph, this solution would provide the possibility to perform square roots of literally anything that can be found in the Register File.

4 Final Considerations

The development of the Square Root Unit started consulting various articles on the subject. With respect to Article (1), which was our choice and reference, we have applied several changes, motivated by the fact that their architecture was thought for FPGAs while ours for ASIC. Also, we have provided more detail on the implementation of the various cells, and a wide array of solutions, among which the proposed 12-bits special unit that does not increase the RISC-V critical path.

We have shown that the carry select architecture is faster than the ripple carry, but other architectures may be better from an area and power consumption point of view. For example, a carry skip architecture could save some area, while small blocks of a carry look-ahead architecture or a carry save architecture could allow to further reduce the critical path in order to increase the parallelism.

Moreover, if we had decided not to generate the remainder, the last stage would not have to generate the result of the subtraction, further simplifying the architecture.

References

[1] T. Sutikno, "An optimized square root algorithm for implementation in fpga hardware," *TELKOMNIKA Telecommunication Computing Electronics and Control*, vol. 8, pp. 1–8, 2010.

[2] Y. Li and W. Chu, "A new non-restoring square root algorithm and its vlsi implementations," in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pp. 538–544, 1996.

[3] P. Kachhwal and B. C. Rout, "Novel square root algorithm and its fpga implementation," in *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*, pp. 158–162, 2014.