Politecnico di Torino

III Facoltà di Ingegneria

RISC-V lite processor

Integrated Systems Architecture

Master's degree in Electronic Engineering

Authors: group 24

Campanella Andrea, Iacovelli Gianluca, Pala Stefano

Contents

Github repository: https://github.com/campandrea/Lab_ISA/

1 RISC-V an Overview

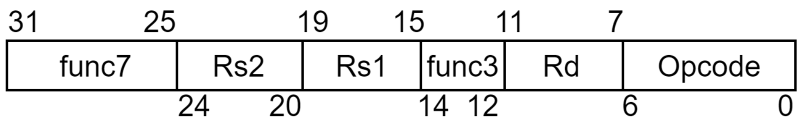
1.1 Introduction

RISC-V is an open source Instruction Set Architecture based specifically on a limited set of operations. The specifications are not fully defined, but leave room for choice dictated by the needs of each user. In particular, the internal parallelism can vary between 32, 64 and 128 bits, the number of registers can be 16 or 32, a floating point unit can be included or not and so on. In this way, the same ISA can be used to define a general purpose processor with 64-bit parallelism, the ability to handle more complex operations or a processor dedicated to IoT solutions with low energy demand with 32-bit parallelism and minimal hardware.

1.2 RISC-V ISA

The instructions have a 3-operand format: a target register *Rd* and two source registers *Rs1* and *Rs2* or one source register *Rs1* and one immediate number. Instructions are divided into different types according to format:

R-type:



Operands:

`Rd, Rs1, Rs2`

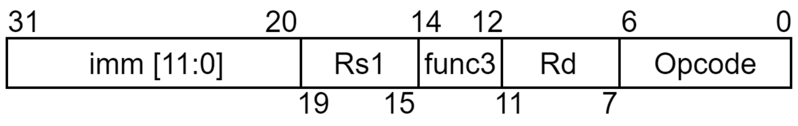
Instructions:

ADD x2, x4, x6 $\longrightarrow$ `x2 = x4 + x6`

XOR x2, x4, x6 $\longrightarrow$ `x2 = x4` $\oplus$ `x6`

SLT x2, x4, x6 $\longrightarrow$ `if x4 < x6, x2 = 1 else x2 = 0`

I-type:



Operands:

`Rd, Rs1, Immed-12`

Instructions:

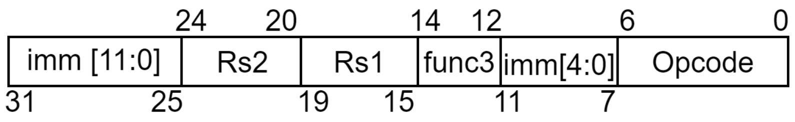
ADDI x2, x4, 20 $\longrightarrow$ `x2 = x4 + 20`

LW x7, 8(x4) $\longrightarrow$ `x7 = MEM[x4 + 8]`

SRAI x2, x4, 20 $\longrightarrow$ `x2 = x4 >> 20`

ANDI x2, x4, 20 $\longrightarrow$ `x2 = x4` $\cdot$ `20`

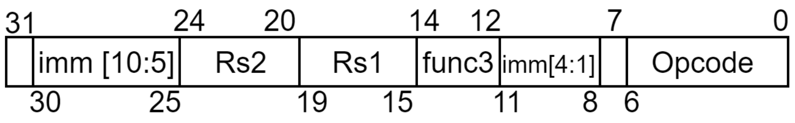
S-type:



Operands:
`Rs1, Rs2, Immed-12`

Instruction:
SW x4, 8(x6) $\longrightarrow$ `MEM[x6 + 8] = x4`

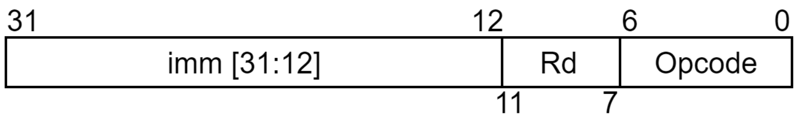
B-type:



Operands:
`Rs1, Rs2, Immed-12`

Instruction:
BEQ x4, x6, loop $\longrightarrow$ `if x4 < x6, goto offset(pc)`

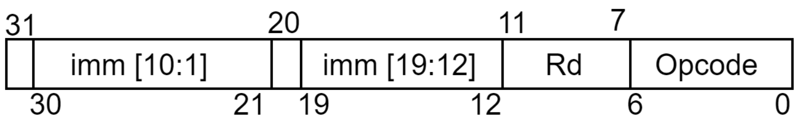
U-type:



Operands:
`Rd, Immed-20`

Instructions:
LUI x4, 0x12AB7 $\longrightarrow$ `x4 = value << 12`
AUIPC x4, 0x12AB7 $\longrightarrow$ `x4 = (value << 12) + pc`

J-type:



Instructions:
`Rd, Immed-20`

Example:
JAL x4, func $\longrightarrow$ `pc = pc + offset; x4 = return address`

1.3 Single Cycle Architecture

The simplest architecture that can be implemented to execute the ISA previously described is the single cycle architecture in which every clock cycle can be carried out one instruction. As shown

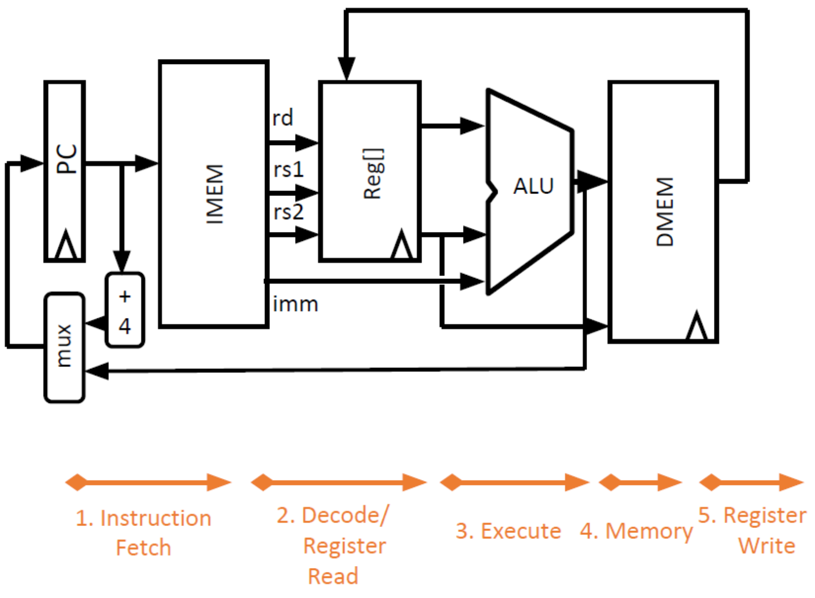


Figure 1: Single Cycle Architecture

in Figure 1, it is noticeable that this type of processor has separate instruction memory and data memory, which is typical of a Harvard architecture. In order to access the right instruction, there is a dedicated PC register with a mux at its input capable of choosing the right address in case of linear or jump execution. The internal registers are organised in a register file with 32 synchronous write and asynchronous read addresses capable of reading two operands at a time. Finally, there is an ALU with two inputs capable of performing the required arithmetic operations. In the RISC-V architecture, the execution of an instruction can be divided into five stages:

Fetch: The instruction is taken from instruction memory.

Decode: The instruction is classified into the various types described in subsection 1.2 by the control unit, which generates the control signals necessary for execution. Data from the register file is also read at this stage.

Execute: Arithmetic operations are performed on the operands.

Memory: If necessary, the data memory is accessed to read or write results.

Register write: The result obtained in the execute stage is stored in the register file.

1.4 Pipelined Architecture

Starting from the single cycle architecture, it is possible to create another version of the RISC-V processor in which the pipeline technique is adopted, so that instruction level parallelism can be exploited to improve throughput. Execution is divided into five pipeline stages corresponding to the phases into which the execution of an instruction can be divided; in this way, multiple instructions can be executed simultaneously while maintaining the same latency for each type of instruction. In order for the processor to function properly, special measures must be taken that can complicate the architecture. First of all, the pipeline must also be applied to the control signals together with the data, so that the correct operations can be carried out on the data at each stage. However, this is not enough because it is also necessary to deal with the various types of hazards that can be encountered:

Structural hazard: Two instructions need the same resource at the same time. This can always be avoided by adopting an appropriate design.

Data Hazard: Occurs when different instructions that exhibit a data dependency access the same data in different stages of pipeline.

Control Hazard: Occurs when the pipeline makes wrong decisions on branch prediction and therefore brings instructions that must be discarded. Since it is realised that execution is no longer sequential during the execution stage, the control hazard is solved by inserting two NOPs into the pipeline. The unit responsible for solving this type of hazard is the Hazard Detection Unit, which, when it recognizes a problem, flushes the IF and ID registers, which corresponds to inserting two NOPs. In the meantime, the PC is updated to the jump address.

1.5 Data Dependencies

There are different types of data dependency depending on which operand of the instruction the data refers to.
The **WAW** Write after Write hazard occurs when two instructions attempt to write the result of an operation into the register in the wrong order.This hazard is typical of machines characterized by concurrent execution, so it is not a RISC-V processor problem.
The **WAR** Write after Read occurs when in a concurrent execution an attempt is made to read data from the register after it has been modified by a subsequent instruction.
The **RAW** Read after Write is the true data dependency.This occurs when an attempt is made to read a register whose contents have been modified in the previous instruction. There are several ways to resolve this type of hazard, one of which is to check that the input addresses of the instruction being executed do not match the output address of the previous instruction or two times before. When one of the two conditions occurs then the data can be forwarded avoiding being first written to the register and blocking the pipeline. The data input to the ALU is taken either from the ALU output register or from the data memory output.

1.6 Modified ISA

The Istruction set of the implemented processor has been expanded by adding a new instruction that allows the execution of the operation *absolute value*. Thanks to this instruction, during the execution stage it is possible to evaluate the module of a number using the dedicated hardware component in the ALU. Only a source register *Rs1* and a destination register *Rd* are required to perform the operation. In order to remain RISC-V compliant, the decision was made to exploit the I-type instruction format despite the fact that the immediate field will not be used. Binary encoding has been adopted as follows:

Opcode: 0010011

func3: 011

2 Implementation

2.1 Datapath architecture

For the implementation of the datapath, the processor described in subsection 1.4 was considered as starting point and the architecture shown in Figure 2 was derived. The five-stage pipelined partition was adopted. In addition, the register file was added in the design but the instruction and data memories were not, as these were only included in the testing phase.

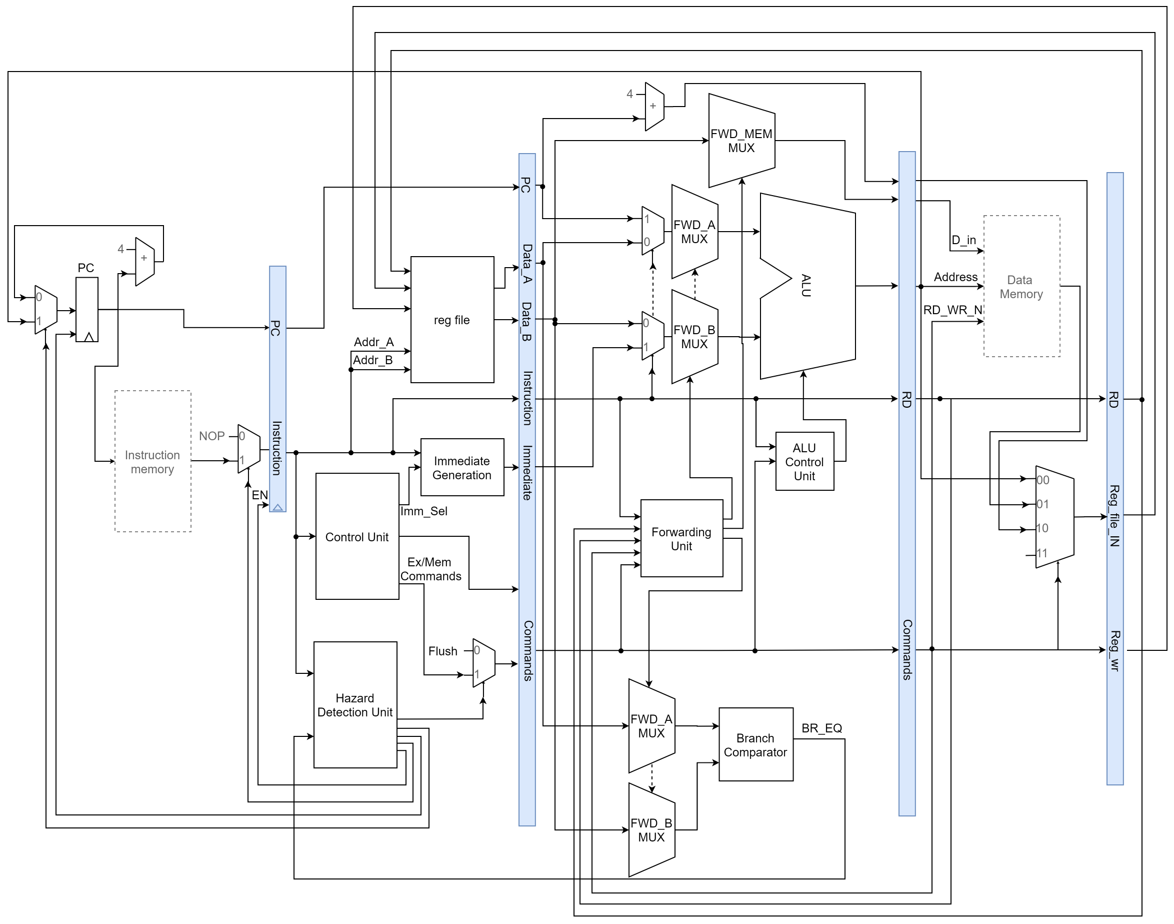


Figure 2: Datapath architecture

2.2 Instruction Fetch stage

The starting point for executing an instruction is to access the instruction memory cell pointed to by the PC. This step is performed in the fetch stage. In particular, there is the register containing the program counter, which is used as the instruction memory address, the adder to increment this value in case of sequential execution and a mux to save the result coming from the ALU in case there are jump operations to be performed. Finally, the instruction and the program counter enter the first pipe stage. The control signals of the pipeline and PC registers are handled by the Hazard Detection Unit.

2.3 Instruction Decode stage

The decode stage contains the blocks used to generate the control and data signals from the fetched instruction. The control unit decodes the instruction and generates the control signals for the other functional units. The register file extracts the address fields from the instruction and asynchronously outputs the words associated with the addressed cells. In this stage there is also the immediate generation unit which, starting from the data in the instruction, makes the sign extension and generates the immediate data on 32 bits that can be used in the execution stage. Finally, another main block is dedicated to hazard detection for jump or load instructions.

2.3.1 Control Unit

For the design of the control unit, a fully combinatorial circuit similar to a decoder has been designed. The only input signal is the instruction from the fetch stage, while several control signals are generated at the output. The seven least significant bits, which always contain the instruction *opcode*, are analysed, and from this all the control signals that manage the blocks of all the stages are generated. The signals that propagate to blocks located at different points in the pipeline must be properly delayed in order to synchronise the controls with the data.
The signals generated by the control unit relate to the commands for reading and writing the data memory, writing the register file and choosing the data to be written; two control flags are managed to report a possible branch or jump instruction. The commands for selecting the data and the type of operation to be performed by the ALU are also generated. Finally, a control signal for the immediate generation block is properly set.

2.3.2 Hazard Detection Unit

As described in the subsection 1.5, the Hazard detection unit is primarily concerned with resolving control hazards by inserting NOPs into the pipeline. When the current instruction is of branch type, it checks the output signal of the Branch Comparator so that when a jump occurs, it commands the IF and ID registers to flush and selects the jump address calculated by the ALU to be loaded into the PC.
It also handles load hazards in the same way, i.e. those data dependencies that cannot be resolved by data forwarding. One case is when a load instruction is followed by an operation that refers to data taken from memory. It can only be solved at compile time by changing the instruction order.

2.3.3 Immediate generation

As described in the subsection 1.2, all instruction formats except R-type have a data element directly within the instruction called immediate. Each type of instruction has a different organization, so different operations must be performed for each format. The generation of the actual 32-bit data to be sent later to the execution stage is done by a decoder-like circuit that performs the following operations:

I type and S type: the data is represented on 12 bits and the sign extension is performed.

B type and J type: the data is represented on 12 bits, and sign extension and left shift of one position is performed.

U type: the data is represented on 20 bits and is left shifted by 12 positions.

2.4 Execution stage

The execution stage contains the blocks that carry out the calculations, including the ALU, the Forwarding Unit, the PC incrementer and some useful units to manage data flows. Each operation lasts one clock cycle so there are no pipeline stages within the stage but only before and after. In Figure 3 all the blocks and the data path can be seen.

The ALU has two inputs which are partly managed by the instruction decoded by the CU and partly

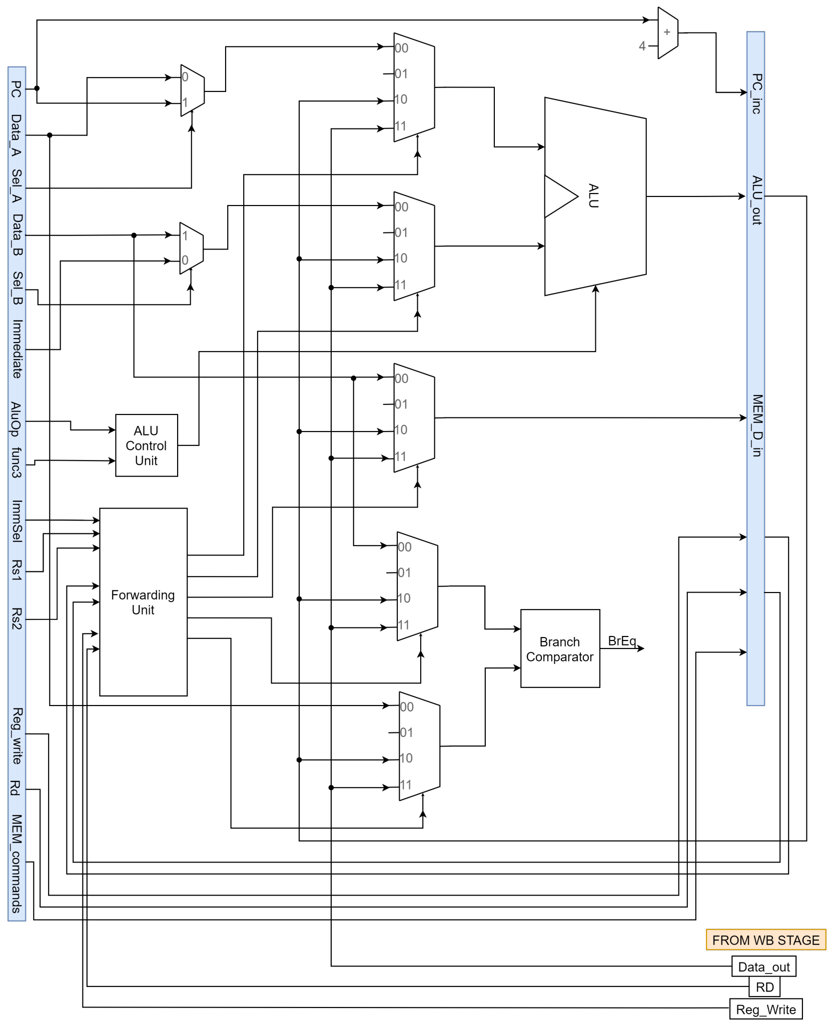


Figure 3: Execution stage

by the forwarding unit in case data dependencies are present. In the case of non-hazardous operation, port ”A” of the ALU contains data ”A” from the register file or the non-incremented PC, while port ”B” contains data from port ”B” of the register file or the Immediate from the instruction. In this way all instructions can be handled without ever having an overlap in the port request. On the other hand, when data dependencies are present, these data are ignored and bypassed by the forwarding unit’s decision in order to insert the correct data. The management of the branch instruction requires the evaluation of equality between two data and is carried out by a separate unit from the ALU as it is busy in calculating the jump address, i.e. the sum between the PC and the displacement present in the Immediate. Some units are analysed in detail in the following subsections.

2.4.1 ALU

The Arithmetic Logic Unit is one of the most important blocks in a digital system; in fact the performances of CPUs, μControllers and μProcessors, are strictly dependent on the ALU structure. Because of this, the component must be developed according to the applications in which it will be used. For example, in a high-performance computer, the ALU must be designed favouring the speed of the logic and arithmetic components, penalising consumption and the occupied area. Conversely, in low-power applications, the component must be economical and dissipate a low amount of power. In the case of RISC-V, the system has to be fast and latency is expected to be low to allow a high clock frequency. The ALU can perform these seven logical and arithmetic operations:

- Addition
- Subtraction
- AND
- XOR
- Comparison
- Right shift
- Absolute value
- Operand Feedforward

The first six instructions perform operations between two numbers, the last one directly outputs the second input of the ALU, but they all work with std_logic inputs. The absolute value operation only uses one operand instead. In addition, the control unit of the ALU drives it with four bits and in case of wrong input control, the component gives zeros as output.

2.4.2 Carry Select Adder

The structure of the adder will be analysed in the following in order to reduce its latency. The purpose is to analyse the costs, latency and fanout at some critical nodes. The Carry Select Adder has been developed from the basic cell of the Full Adder, shown in the Figure 4

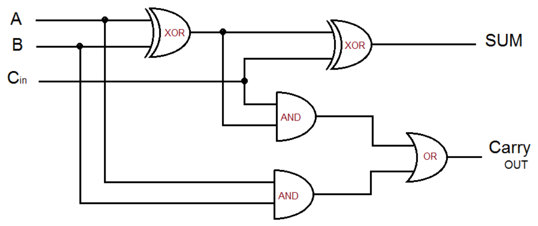


Figure 4: Full adder, logic gates schematic

This simple block was used to create a not common and not uniform adder structure. In general the adder structure is composed by elementary FAs, that are grouped as a power of two. The implementation was done with a group of six full adders and another group of two full adders at the beginning. The main carry select adder block is showed in Figure 5.
The reason for this choice is the masking of the latency of the first multiplexer, in fact the low latency of the two full adders at the start allows the first multiplexer to be switched before the first main

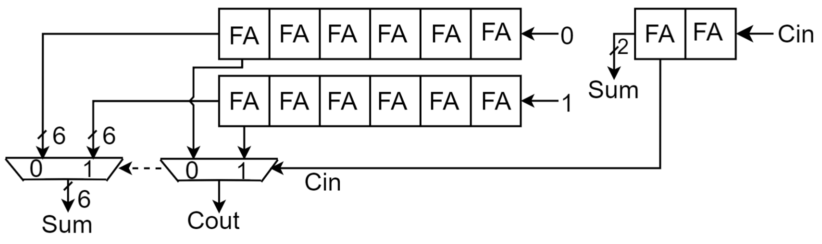


Figure 5: Carry Select Adder main block

block has even finished calculating. In this way, the latency is characterised by a weaker dependence on the delay of the multiplexer. Considering a total number of five main blocks plus two full adders, the total delay is:

$$T_{DELAY} = (\frac{N-2}{6} - 1)T_{MUX} + 6T_{FA} \tag{1}$$

Where N is the total number of bits. If compared with a carry save adder with a main block of eight full adders, the total multiplexer delay is the same but there is an advantage with respect to the full adder delay. Furthermore, if the used structure is compared with one based on group of four full adders, the advantage will be observed by looking the total multiplexer delay, that is higher in the second case than the first one. Now a fanout analysis is going to be done; by looking the Figure 5, there is the necessity to observe the two multiplexer select signal. The equation of these two multiplexer is:

$$(In_1 \cdot C_{IN}) + (In_2 \cdot \overline{C_{IN}}) \tag{2}$$

The selector has to drive:

$$F_{OUT} = 2 \cdot 6 \; Gates + 2 \cdot 2 \; Gates = 14 \; Gates \tag{3}$$

The selector experiences a very large fanout and it happens in all multiplexers, therefore when the circuit will be synthetized, a driver must be inserted in order to avoid an high load capacitance and an high peak current, in which the first slow the circuit and the second could yield electromigration or burn the logic gates.
The last parameter that must be analysed is the cost, or to be more precise, the total number of components that have to be used in order to develop the adder:

- $N_{FA} = 2 \cdot N_{BIT} = 64$
- $N_{MUX} = 2 \cdot \frac{N-2}{6} = 10$

In order to obtain better performances, the cost to be paid is quite high, in fact respect to a simple ripple carry adder, the number of full adder is doubled and in addition there are ten multiplexer, The overall architecture is represented in Figure 6:

2.5 Absolute value

In order to carry out the new absolute value instruction, it has been opted to modify the ALU directly, rather than using a dedicated unit, since the data access and the forwarding infrastructure is basically the same. The modified ALU takes only one value as input, and when the control signal indicates the absolute value, it will compute it and return it as output. Further modifications have been applied to the CU to enable it to recognize the new instruction and to the ALU control logic to generate the correct control signals.
The changes that will have the greatest impact on the performance of the whole machine are certainly those of the ALU because in each case a dedicated unit is added to carry out the new operation.

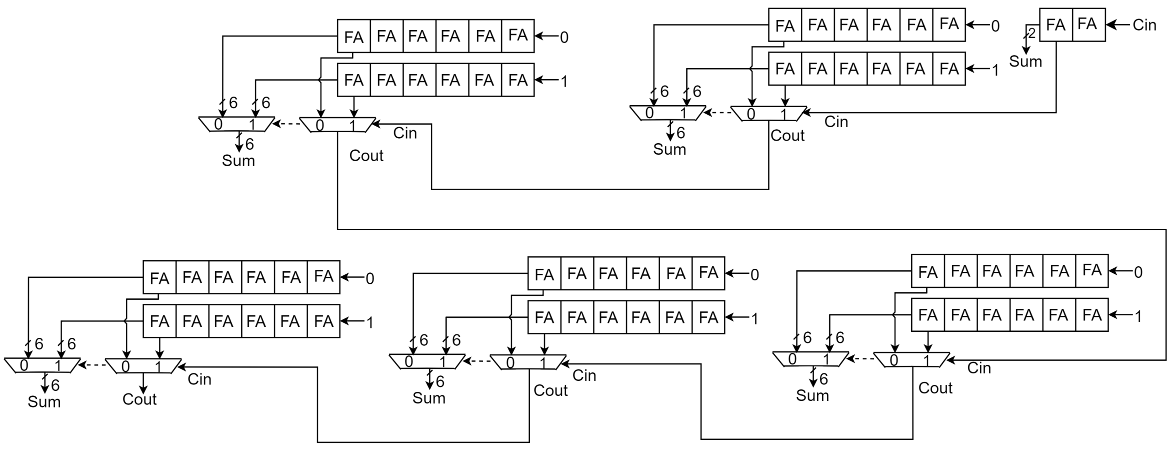


Figure 6: Carry Select Adder schematic

2.5.1 Forwarding Unit

The forwarding unit is responsible for resolving all data dependencies between the Execute stage and the other pipeline stages. In particular, it looks for situations in which the source register of the current instruction coincides with the destination address of the instruction processed one or two clock cycles earlier.

```
addi R1, R2, 1        addi R1, R2, 1
addi R3, R1, 1        NOP
                      addi R3, R1, 1
```

Figure 7: Example of data dependencies solved by the forwarding unit

In the first case, the data is taken from the output register of the ALU, instead of the register file. In the second case from the memory output register. In this way the data is forwarded without waiting for the correct data to be written in the register file. Each time the forwarding unit starts up, there is a benefit that corresponds to one or two clock cycles. To understand if the bits of the source and destination addresses are valid it is necessary to check that the instructions they refer to use such fields, i.e. if the instruction is of the type *R, I, U, J.*
The forwarding unit also handles Store instructions, which write directly to memory by forwarding the data from the ALU output register or the memory output register to the data memory input.
Since the branch evaluation is also done on data from the register file, it requires the forwarding of data of both inputs.

2.6 Memory and Write back stage

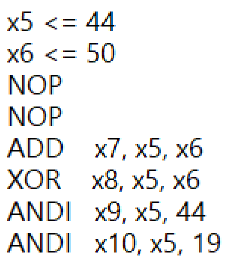
The last two stages involve reading and writing from memory and saving data to the register file. The memory stage contains the data memory, which receives the data, address and signals from the previous pipe stage to execute the store and load instructions. At the output there is a mux to determine which data must be written inside the register file between the data coming out from the memory, the data coming out from the ALU or the PC.

3 Testbench

In order to test the functioning of the processor, after verifying that all the individual components worked as expected, different input vectors were used to make the machine perform increasingly complex operations.

3.1 Testing basic ALU functions and Register File

The first program is used to verify that the ALU is capable of carrying out all basic instructions and rewriting them in the Register File.



```
x5 <= 44
x6 <= 50
NOP
NOP
ADD   x7, x5, x6
XOR   x8, x5, x6
ANDI  x9, x5, 44
ANDI  x10, x5, 19
```

Figure 8: Testing ALU and Register File

The first instructions initialise registers x5 and x6 to two known values, then two NOPs are inserted to avoid data hazard and finally several arithmetic operations are performed and the results saved in registers. As can be seen from Figure 9, in registers x7, x8, x9 and x10 there are respectively sum, bitwise XOR and AND immediate, once with an identical operands and once with different operands.

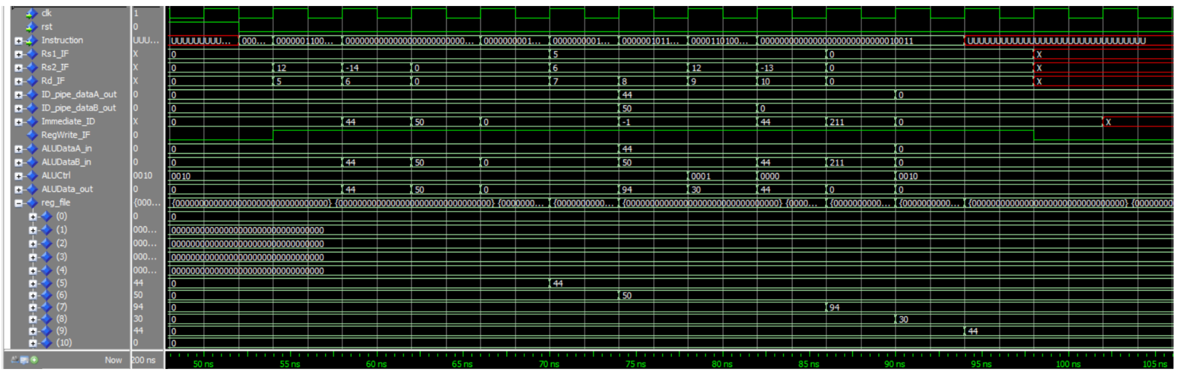
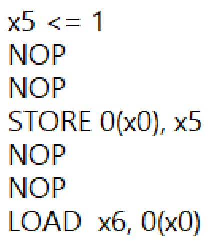


Figure 9: Testing ALU and Register File

It can be seen that all control signals are correctly synchronised and data is written to the Register File.

3.2 Testing Data Memory

This script aims to test the functionality of the data memory in reading and writing mode.



```
x5 <= 1
NOP
NOP
STORE 0(x0), x5
NOP
NOP
LOAD  x6, 0(x0)
```

Figure 10: Testing Data Memory

The first instruction initializes the x5 register to the value 1, after which it is stored in the memory cell pointed by x0, i.e. cell 0. Finally, a load is performed to read the same memory cell and save its content in x6.
As shown in the Figure 11, in the Register file the value 1 is present both in position x5 and x6, a sign that the simulation has been successful. In addition, the memory enable signals worked correctly. In this program there were no data hazards in fact NOPs were inserted between one operation and another.

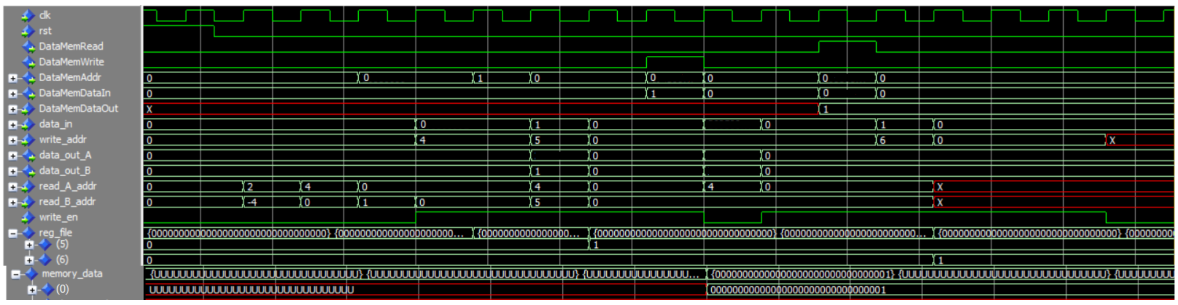
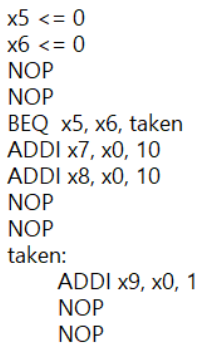


Figure 11: Testing Data Memory

3.3 Testing Branch instructions

This script will test the functionality of the hazard detection unit when a branch instruction occurs.



```
x5 <= 0
x6 <= 0
NOP
NOP
BEQ  x5, x6, taken
ADDI x7, x0, 10
ADDI x8, x0, 10
NOP
NOP
taken:
      ADDI x9, x0, 1
      NOP
      NOP
```

Figure 12: Testing branch instructions

Registers x5 and x6 are both initialised to 0 to make the jump condition true. After the branch instruction, two sums are inserted to modify the contents of registers x7 and x8 in order to check that they are not actually modified. Finally, to confirm that the jump has taken place, as well as checking that the PC has been updated, it can be observed that register x9 has been updated with the value 1, i.e. the first instruction after the jump has been executed correctly.
As shown in the simulation Figure 13, during the Execute stage the signal coming from the branch comparator becomes valid, so the pipeline registers referring to the Fetch and Decode stages are flushed and the PC is updated with the jump address. The hazard detection unit and the jumping mechanism work correctly, the overhead in case of a taken branch is equal to two instructions after which it continues sequentially starting from the jumping address contained in the branch instruction.
In Figure 14, on the other hand, the x5 and x6 registers are at two different values so that the jump is not performed, therefore the x7 and x8 registers are changed.

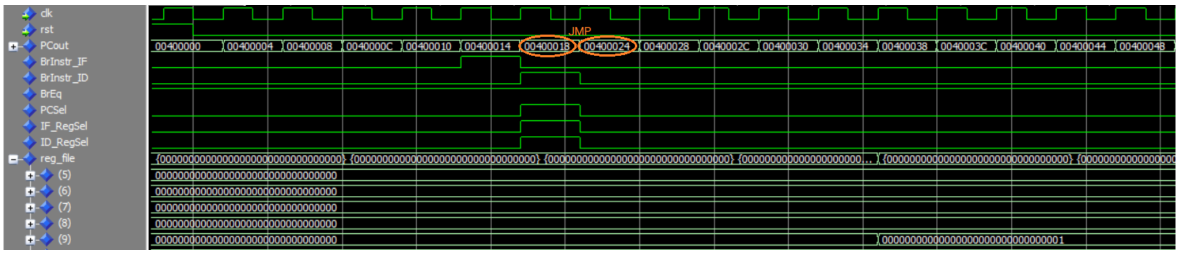


Figure 13: Test with taken branch

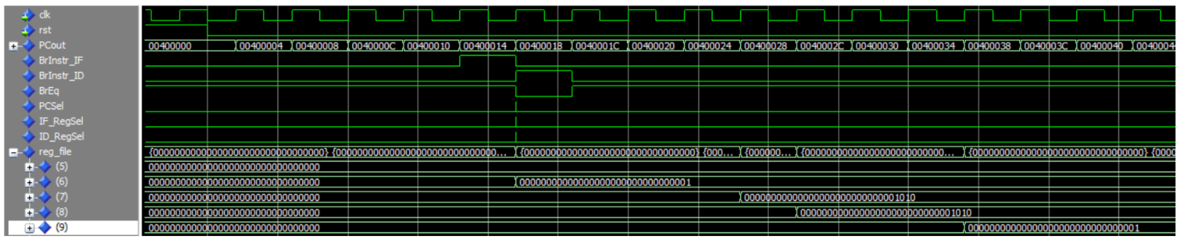
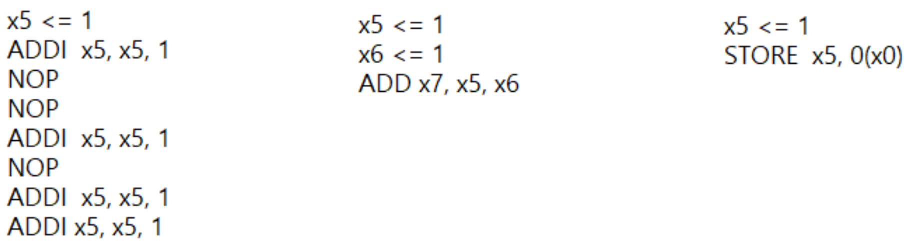


Figure 14: Testing with un-taken branch

3.4 Testing Forwarding Mechanism

The purpose of these scripts is to verify that the forwarding mechanism is working correctly.



```
x5 <= 1
ADDI  x5, x5, 1
NOP
NOP
ADDI  x5, x5, 1
NOP
ADDI  x5, x5, 1
ADDI x5, x5, 1
```

```
x5 <= 1
x6 <= 1
ADD x7, x5, x6
```

```
x5 <= 1
STORE  x5, 0(x0)
```

Figure 15: Testing forwarding mechanism

In the first script there are two different situations in which a given dependency occurs, one in which the distance between instructions is equal to one clock cycle and a second in which the distance is equal to 2. The first case is solved by forwarding the ALU's output register, while the second by taking the Write Back register. From the Figure 16, it can be noticed that the final value of the x5 register is equal to 5, that is the number of times the sum instruction has been executed, and the forwarding signal change value for each different data dependency.

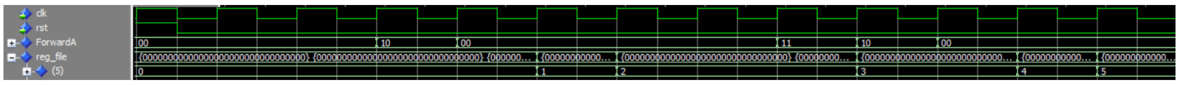


Figure 16: Testing Forwarding

The second script aims to verify a similar case to the previous one adding a complexity factor. The add instruction requires both registers x6 and x5 whose contents have been modified at the previous clock cycle and the one before that, respectively. In the Figure 17 it can be read that the value of x7 is correctly equal to 2 and therefore the Forwarding Unit has worked properly.
The purpose of the last script is to forward the data to be saved in the data memory. When the Forwarding Unit recognises the data dependency, it ensures that the data is not taken from the ALU when it enters the memory, but from the Write Back stage.

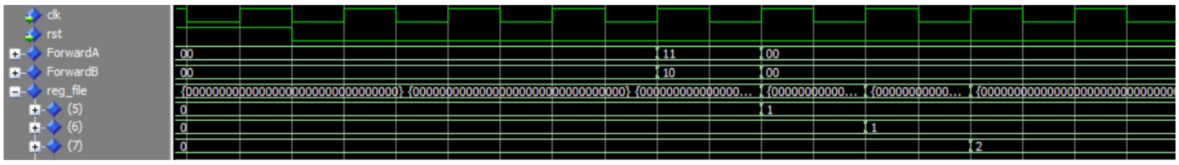


Figure 17: Testing Forwarding of both data

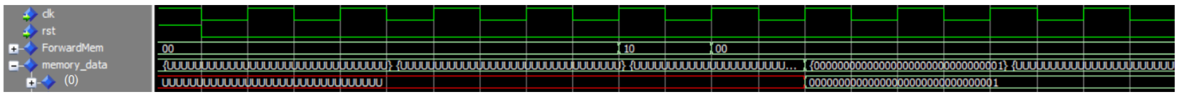
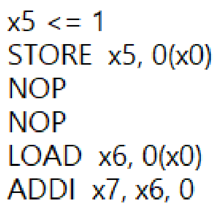


Figure 18: Testing memory Forwarding

3.5 Testing Load Hazard

This script checks the operation of the hazard detection unit when a data dependency occurs that cannot be resolved by data forwarding.



```
x5 <= 1
STORE  x5, 0(x0)
NOP
NOP
LOAD  x6, 0(x0)
ADDI  x7, x6, 0
```

Figure 19: Testing Load Hazard

In this case there is a load hazard between the LOAD operation and the ADD Immediate because the output register of the former coincides with the source address of the latter. The LOAD instruction retrieves the data from memory and saves it in the register, so the data will only be visible in the MEM stage. The addition operation, on the other hand, requires the data during the Execute stage,

so even though it is forwarding, the hazard would not be eliminated. Then the hazard detection unit operates as if there were a control hazard, i.e. inserting two NOPs but without updating the PC, which remains at the current value in order to pick up again the pointed instruction since it has been flushed. In this way the drawback corresponds to two instructions, although through optimisation it could be reduced to one. In order not to increase the complexity of the machine, the solution described above has been chosen.
The Figure 20 shows how the registers are set to NOP and the PC remains set to the current value as soon as the load hazard is detected.

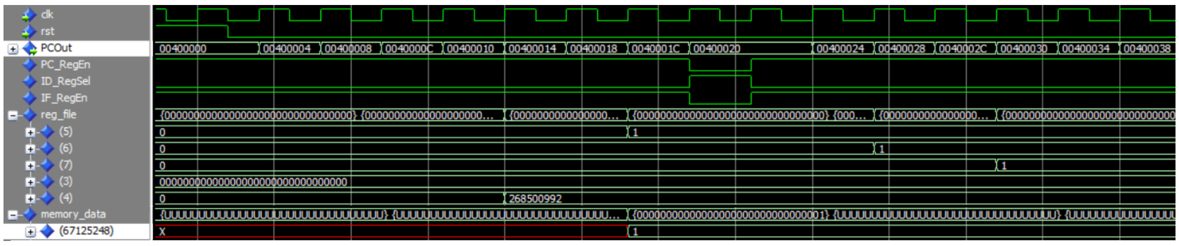


Figure 20: Testing Load Hazard

3.6 Testing a complete program

This application searches for the smallest number in absolute value within a vector and saves the result in memory.

```
v = [10, -47, 22, -3, 15, 27, -4]
start:
      x16 <= 7
      x4   <= v                # put in x4 the address of v
      x5   <= 0
      x13 <= 0x3fffffff        # put in x13 the max possible value
loop:
      BEQ   x16, x0, done
      LOAD   x8, 0(x4)         # load new element in x8
      SHIFT   x9, x8, 31       # get the sign
      XOR   x10, x8, x9        # x10 = x8 XOR x9
      ANDI   x9, x9, 1         # carry in
      ADD   x10, x10, x9       # add the carry in
      ADDI   x4, x4, 4         # point to the next element
      ADDI   x16, x16, -1      # decrease x16
      SLT   x11, x10, x13      # x11 = 1 if (x10 < x13)
      BEQ   x11, x0, loop      # next element
      ADD   x13, x10, x0       # update min
      JAL   loop               # next element

done:
      STORE   x13, 0(x5)       # store the result
```

Figure 21: Testing a complete program

Starting from a vector of 7 elements saved in memory, the algorithm takes one element of the vector from memory each time and compares it with the current minimum. If it finds a new minimum, then it is updated, otherwise it continues. When the whole vector has been analysed, the minimum is saved in memory. Since there is no instruction in the ISA that evaluates the absolute value of a number, the operations are slightly more complicated because the positive value has to be executed first using 3 instruction rather then one. The optimised code with the extended ISA can be found in the subsection 3.7. The Figure 22 shows the seven executions of the cycle and the correct minimum research. It can be seen how step by step the registers x9, x10 and x11 are updated with the different mid values of the execution.

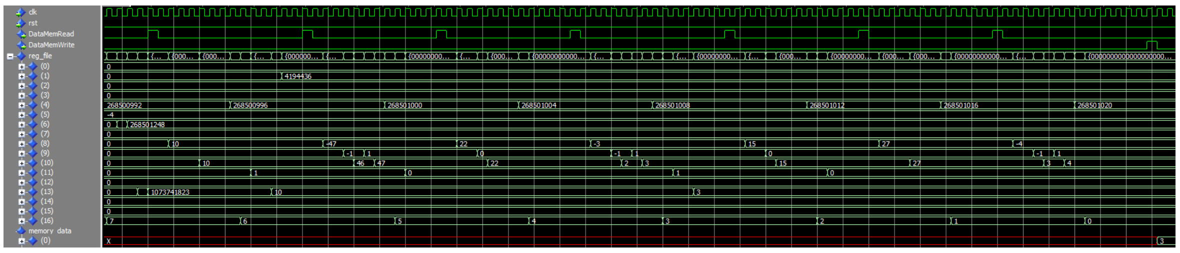


Figure 22: Testing a complete program

3.7 Testing Absolute Value Instruction

To test the correct operation of the new RISC-V with the extended ISA thanks to the addition of the absolute value instruction, the same algorithm as in the previous subsection can be used, slightly modifying the code.

```
v = [10, -47, 22, -3, 15, 27, -4]
start:
      x16 <= 7
      x4   <= v                 # put in x4 the address of v
      x5   <= 0
      x13 <= 0x3fffffff         # put in x13 the max possible value
loop:
      BEQ   x16, x0, done
      LOAD   x8, 0(x4)          # load new element in x8
      ABS   x10, x8             # x10 = |x8|
      ADDI   x4, x4, 4          # point to the next element
      ADDI   x16, x16, -1       # decrease x16
      SLT   x11, x10, x13       # x11 = 1 if (x10 < x13)
      BEQ   x11, x0, loop       # next element
      ADD   x13, x10, x0        # update min
      JAL   loop                # next element

done:
      STORE   x13, 0(x5)        # store the result
```

Figure 23: Testing Load Hazard

The operation is identical to the previous case, but the cycle lasts fewer clock cycles and therefore the algorithm in general takes less time. The result is no different from the previous one.

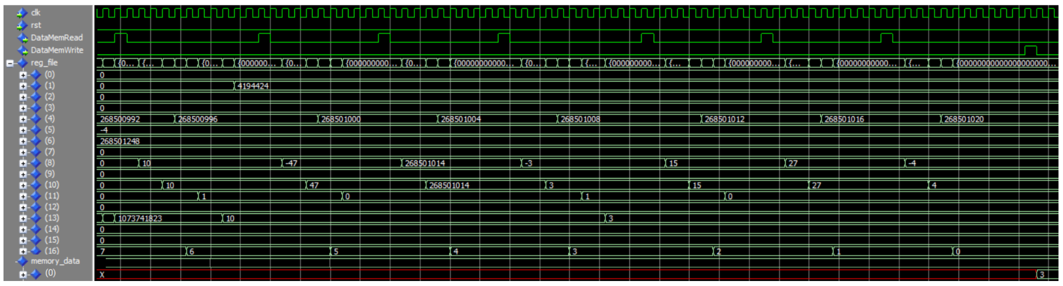


Figure 24: Testing Absolute Value Instruction

4 Synthesis

In this section the synthesis results of the different architectures will be analysed and compared in terms of maximum clock frequency and occupied area.
In addition, the synthesis resulting netlist will be simulated through Simulink to verify that there are no differences with the designed one.

4.1 Results

The analysed architectures are mainly two, one in which all the blocks have been described from a behavioural point of view and the other in which it has been chosen how to synthesise some more critical combinatorial blocks. The behavioural description allows the Synopsys synthesis tool to perform all optimisations and to choose the best structure based on the given constraints. On the other hand, when choosing a particular structure Synopsys is constrained to use it, and sometimes better results can be achieved.
In the case analysed hereafter, for example, a summator implemented as a Carry-Save-Adder was chosen with regard to the ALU in the constrained version.
The performance analysis was repeated twice, once for the first implementation of the RISC-V processor and again for the processor with the ISA modified by the addition of the absolute value instruction.
In the Table 1 it can be seen, as expected, that the optimisations performed by Synopsys when fewer constraints are present are more effective, both in the base case and with the modified ISA.

	Base model	CSA	Base ABS	ABS & CSA
Clock frequency (ns)	1.26	1.33	1.30	1.38
Comb. Area (μm^2)	6430	6975	6909	7242
Non Comb. Area (μm^2)	6471	6459	6469	6462
Total Area (μm^2)	12902	13435	13378	13705

Table 1: Results of the synthesis

From an area perspective, the main differences between the four analyzed cases concern the occupation of the combinational part of the circuit. In particular, it can be seen that in both cases the area increases by 5% when the implementation of the adder is done with the CSA and that by inserting the absolute value module inside the ALU the area increases by a further 10%. The area of the non-combinational part, on the other hand, remains more or less constant in all cases since the major changes are in the combinational blocks. In terms of total area, therefore, the differences between the various implementations are halved, having the two contributions a very similar value.
Both after extracting the netlist and after running the place & route, the DUT was simulated with the input vectors used in the section 3, and the results were identical to the previous ones.

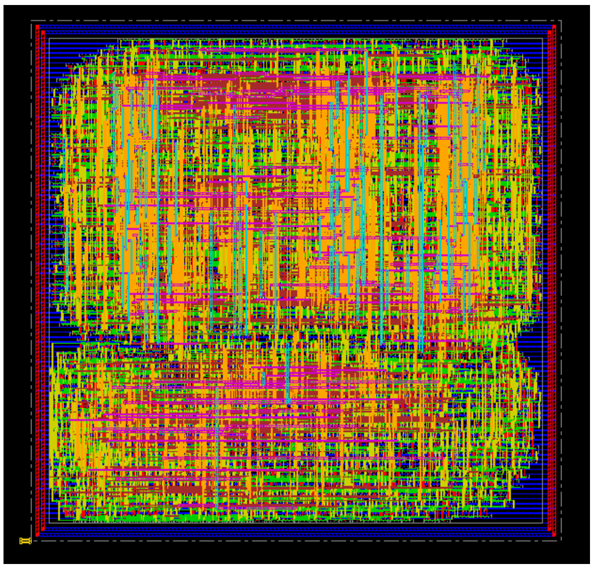


Figure 25: Layout after place phase

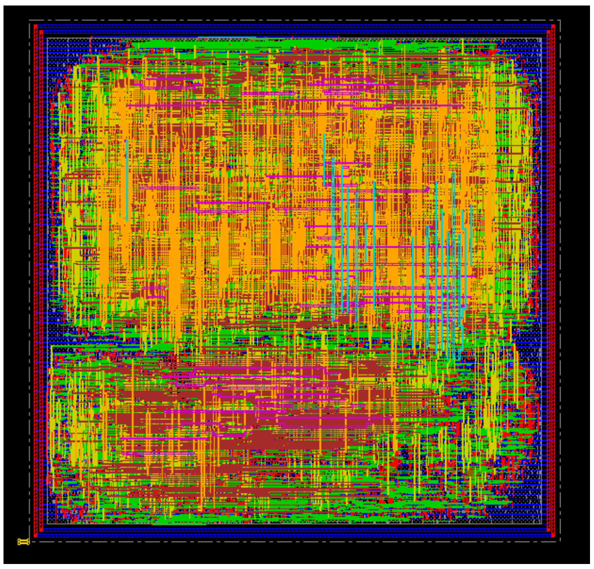


Figure 26: Layout after place&route