TÉCNICO LISBOA

Instituto Superior Técnico

Digital Systems Design

1º Semester 2021/2022

2º Report
Scheduling and Resource Sharing

Group 7

Nome	Número
André Pereira	90016
Tiago Gonçalves	90195

Day of the week:Friday, from 10:00 AM to 1:00 PM.

Professor: Horácio Neto

1 Moore Machine

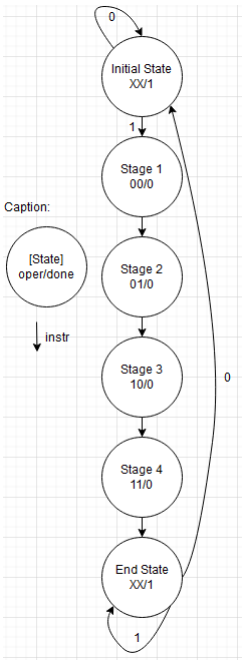

Figure 1: Moore state machine. Each circle represents the state, while showing the std_logic_vector ”oper” on the left and ”done” on the right. Each arrow represents the next state decision making, made possible by the std_logic ”instr”.

2 Data Flow Graph

$$D(p,q) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2 + (p_4 - q_4)^2$$

Figure 2: Target equation to be implemented in the circuit.

In order to perform said equation, we need to do the operations step-by-step:

- Firstly, we need to compute each term that is going to be squared.
- Secondly, we need to square said terms.
- After that, we need to add all the squared terms together. So we firstly add the term 1 with the term 2, and the term 3 with the term 4.
- Finally, we add the partial results of the two previous terms to obtain the equation result.

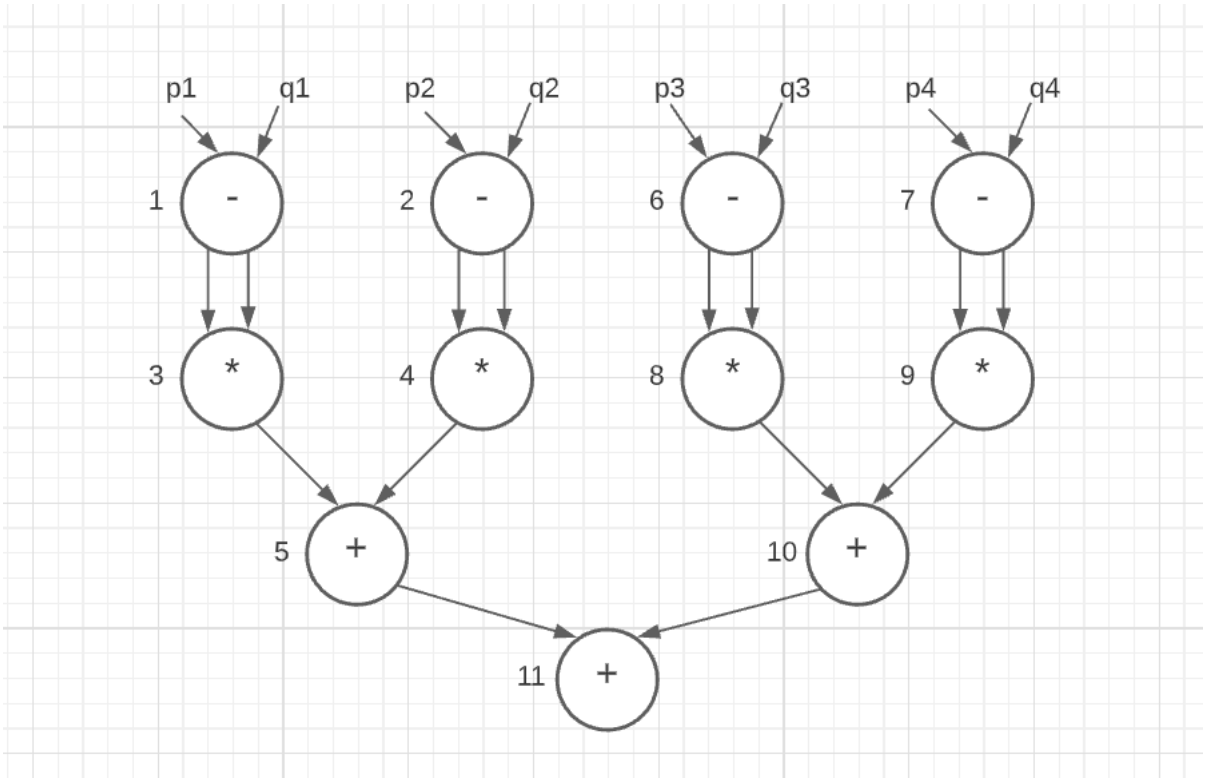

Figure 3: Data flow graph corresponding to one distance calculation.

3 Priority List

Bearing in mind we only have access to 2 multipliers and 2 adders, we needed to take into account the dependencies of each step of the data flow graph to rearrange it into the priority list. Since a operation cannot be performed before a operation that it is dependent on, we simply calculated the priority on the critical path and set the order of the operations based on their priority and dependency, aswell as hardware constraints:

OPER		Priority Critical Path
1	ALU	4
2	ALU	4
3	*	3
4	*	3
5	ALU	2
6	ALU	4
7	ALU	4
8	*	3
9	*	3
10	ALU	2
11	ALU	1

(a) Operation assignment acording to the priority of the critical path

- 1 (4) a
- 2 (4) a
- 6 (4) a
- 7 (4) a
- 3 (3) * ≪-1
- 4 (3) * ≪-2
- 8 (3) * ≪-6
- 9 (3) * ≪-7
- 5 (2) a ≪-3,4
- 10 (2) a ≪-8,9
- 11 (1) a ≪-5,10

(b) Priority List. Numbers to the right of the arrow indicate dependency to previous operations.

Figure 4: Priority List construction.

4 List Scheduling

Now taking into account the priority list defined above and that one multiplication requires one clock cycle, and that 1 addition/subtraction can be performed in half clock cycle, we can design the list scheduling such that:

- We perform the required amount of operations in each cycle, such that we do not need to do more cycles than the minimal necessary ones.
- Whenever we perform a operation, we can either store the result to be used in a different operation or feed it directly to another operaton that can occur in the same clock cycle.
- We obey the hardware constraints.

For example:

- In cycle 1, we perform two subtractions. While we could have performed all four subtractions, we only had two available adders/subtractors. Luckily, this leads to, on the next cycle, a possibility to calculate the square of these two terms.
- In cycle 4, we perform the addition of the last partial sum needed to perform the final sum that leads to the result of the equation. Since each addition only takes half a clock cycle, we can feed this result directly to the adder that was idle and, therefore, compute the final result on the same cycle we did the operation to obtain one of the partial sums.

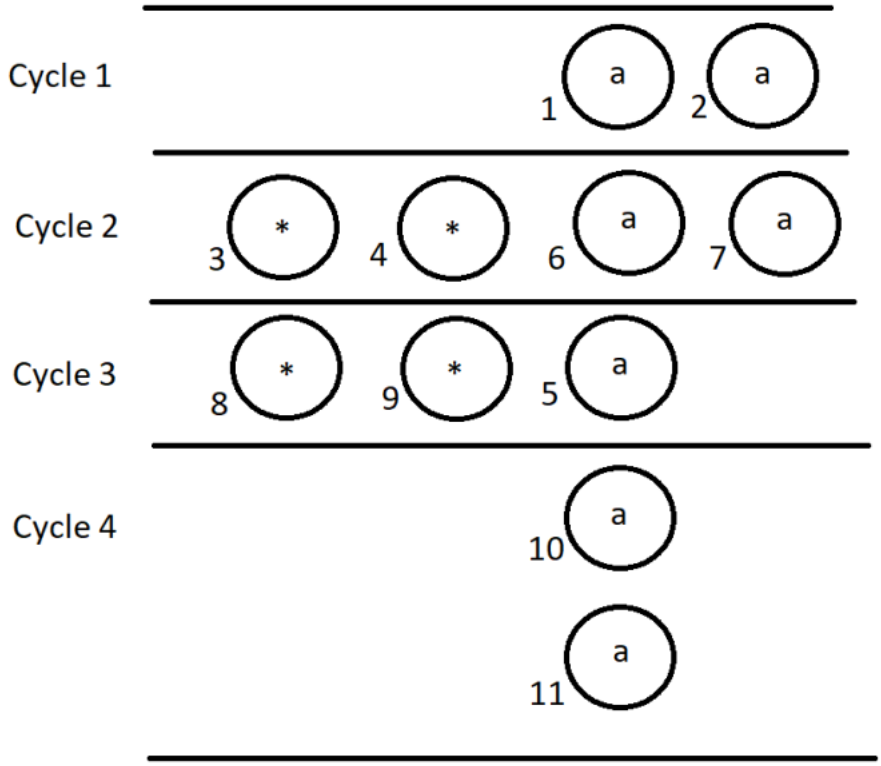

Figure 5: List Scheduling. ”a” represents addition/subtraction and ”*” a multiplication.

5 Design

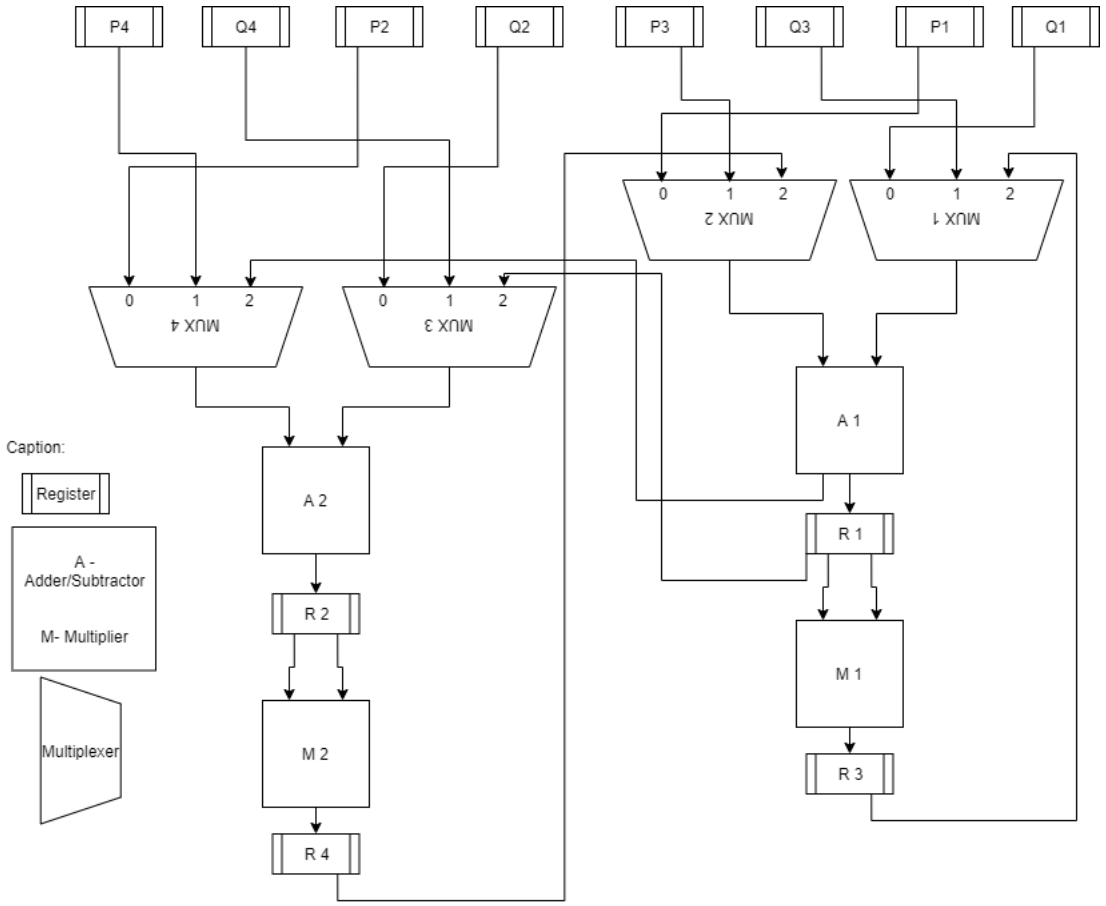

Figure 6: Design of the circuit created.

In our implementation, we decided to use a flag that indicates when should the circuit start and stop doing the algorithm computation, in addition to a global system reset that stores in each register the value 0 and return to the initial state, where it will remain idle until a new input is given. So, besides the given entry inputs registers and adders/multipliers, we needed an additional:

- 4 3-input multiplexers
- 4 internal registers

Furthermore, there are some design choices we also did that need to be addressed:

- The numbers are stored internally using the two´s complement numeric representation in each of the internal registers, since some operations might give negative results (if the inputs are p_1=0 and q_1=255, the first subtraction will result in -255),

but the input registers use unsigned format, since the values that are given must be between [0;255].

- Internal registers have different sizes:

  R1 has 18 bits, where the most significant bit is the signal, and the remaining 17 bits are used to represent the number (in the worst case, we have two times 255^2, leading to the result being 130050. The closest representation comes with 2^{17},that produces the number 131072).

  R2 has 19 bits, where the most significant bit is the signal, and the remaining 18 bits are used to represent the number (in the worst case, we have 2(255^2+255^2) , leading to the result being 260100. The closest representation comes with 2^{18},that produces the number 262144).

  R3 and R4 have 17 bits, where the most significant bit is the signal, and the remaining 16 bits are used to represent the number (in the worst case, we have 255^2, leading to the result being 65025. The closest representation comes with 2^{16},that produces the number 65536).

- Finally, the multiplexers:

  $Multiplexer_1$ will be choosing one of the inputs of the first adder (A1). It will chose between two input values (q1 and q3) and a register (R3). In the first Cycle it will chose q1 with the selector bits ”00”. In the second Cycle it will chose q3 with the selector bits ”01”. Finally in the Third and Forth Cycle it will chose the register R3 with the selector bits ”10”.

  $Multiplexer_2$ will be choosing another of the inputs of the first adder (A1). It will chose between two input values (p1 and p3) and a register (R4). In the first Cycle it will chose p1 with the selector bits ”00”. In the second Cycle it will chose p3 with the selector bits ”01”. Finally in the Third and the Forth Cycle it will chose the register R4 with the selector bits ”10”.

  $Multiplexer_3$ will be choosing one of the inputs of the second adder (A2). It will chose between two input values (q2 and q4) and a register (R1). In the first Cycle it will chose q2 with the selector bits ”00”. In the second Cycle it will chose q4 with the selector bits ”01”. Finally in the Forth Cycle it will chose the register R1 with the selector bits ”10”.

  $Multiplexer_4$ will be choosing another of the inputs of the second adder (A2). It will chose between two input values (p2 and p4) and the output of adder 1 (A1). In the first Cycle it will chose p2 with the selector bits ”00”. In the second Cycle it will chose p4 with the selector bits ”01”. Finally in the Forth Cycle it will chose the adder 1 (A1) with the selector bits ”10”.

So as we can see, we can use the same 2-bit selector (oper) for all the multiplexers .

Table 1: Multiplexers innerworking

Cycle	$Multiplexer_1$	$Multiplexer_2$	$Multiplexer_3$	$Multiplexer_4$
1	00	00	00	00
2	01	01	01	01
3	10	10	XX	XX
4	10	10	10	10

Also, the adders/subtracters are controlled by bit 1 of the oper. In the 1st an 2nd Cycles this bit will be 0, so it will do a subtraction, in the 3rd and 4th Cycles this bit will be 1, so it will do an addition.

6 Benchmark and results obtained

Now we simulate our circuit and the results are shown in Figure 7. As we can see, in this test we tried to get the worst case. reg1, reg2, reg3 and reg4 will be the registers R1, R2, R3 and R4, respectively, and res is the result truncated. As we can see in the circuit, we did not have any overflow. The problem was when the number was truncated. It was to big to be represented in the output, which must be between [0, 4095]. That is why the output will be wrong.

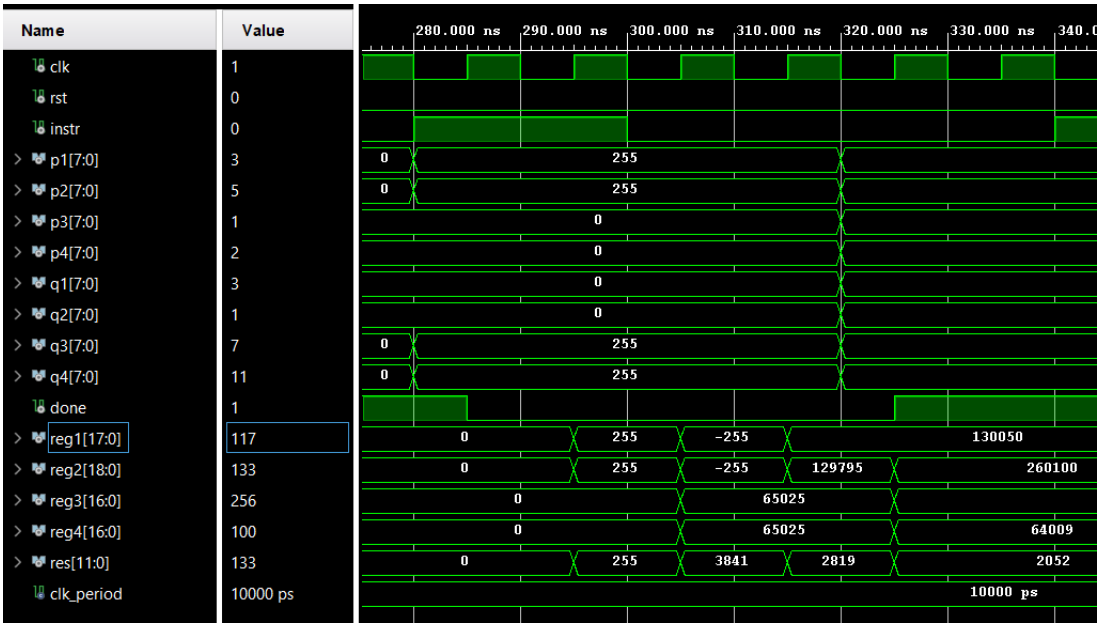

Figure 7: Testbench case 1
Overflow of the result.

In our second test we tried another set of inputs. This time the result will be in the

range of the output and, because of that, when we truncate the result, it won't change. We also tried the reset button and all the registers were set to 0.

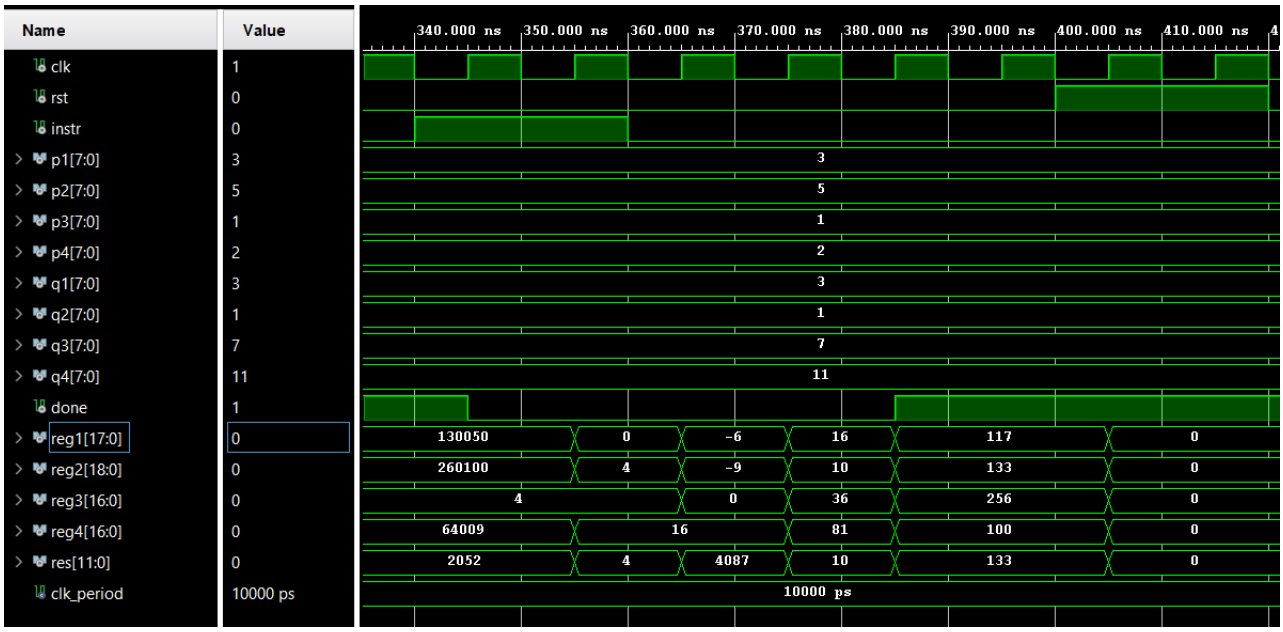

Figure 8: Testbench case 2
Normal test case with global reset at the end.

7 Timing analysis

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1,683 ns	Worst Hold Slack (WHS):	0,238 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	113	Total Number of Endpoints:	113	Total Number of Endpoints:	123

All user specified timing constraints are met.

Figure 9: Time Analysis

By looking at the timing analysis, we can conclude that we can use a faster clock (supposedly we have a 1,683 ns speedup margin). So we tried to use a 8,317 ns clock instead of the 10 ns clock that we had originally, and the circuit still works as intended:

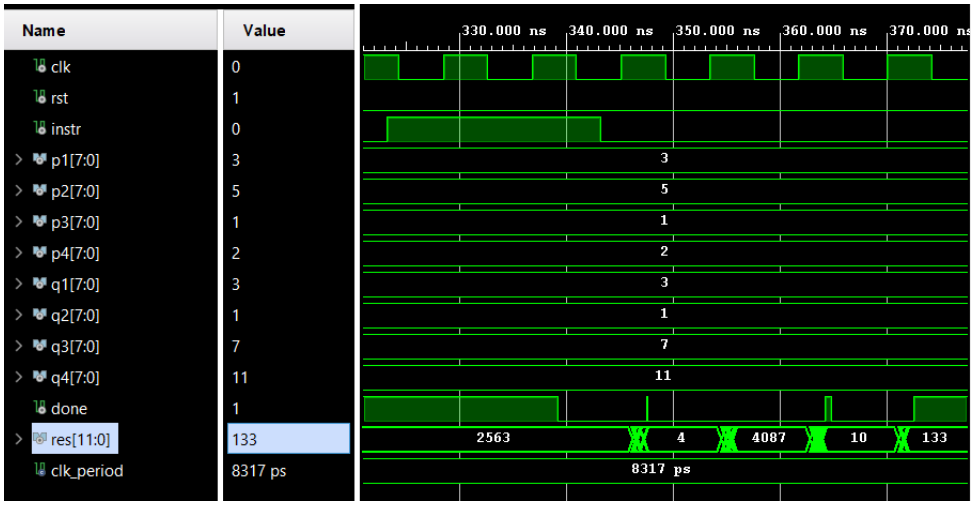

Figure 10: Testbench case 3
Normal test case with the new faster clock implemented.

8 Resource Consumption

Summary

Resource	Utilization	Available	Utilization %
LUT	165	20800	0.79
FF	122	41600	0.29
IO	80	106	75.47

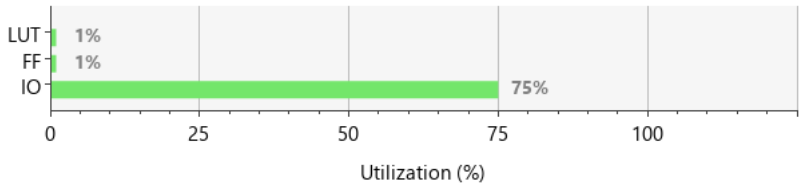

Figure 11: Resource Consumption

9 Pipeline

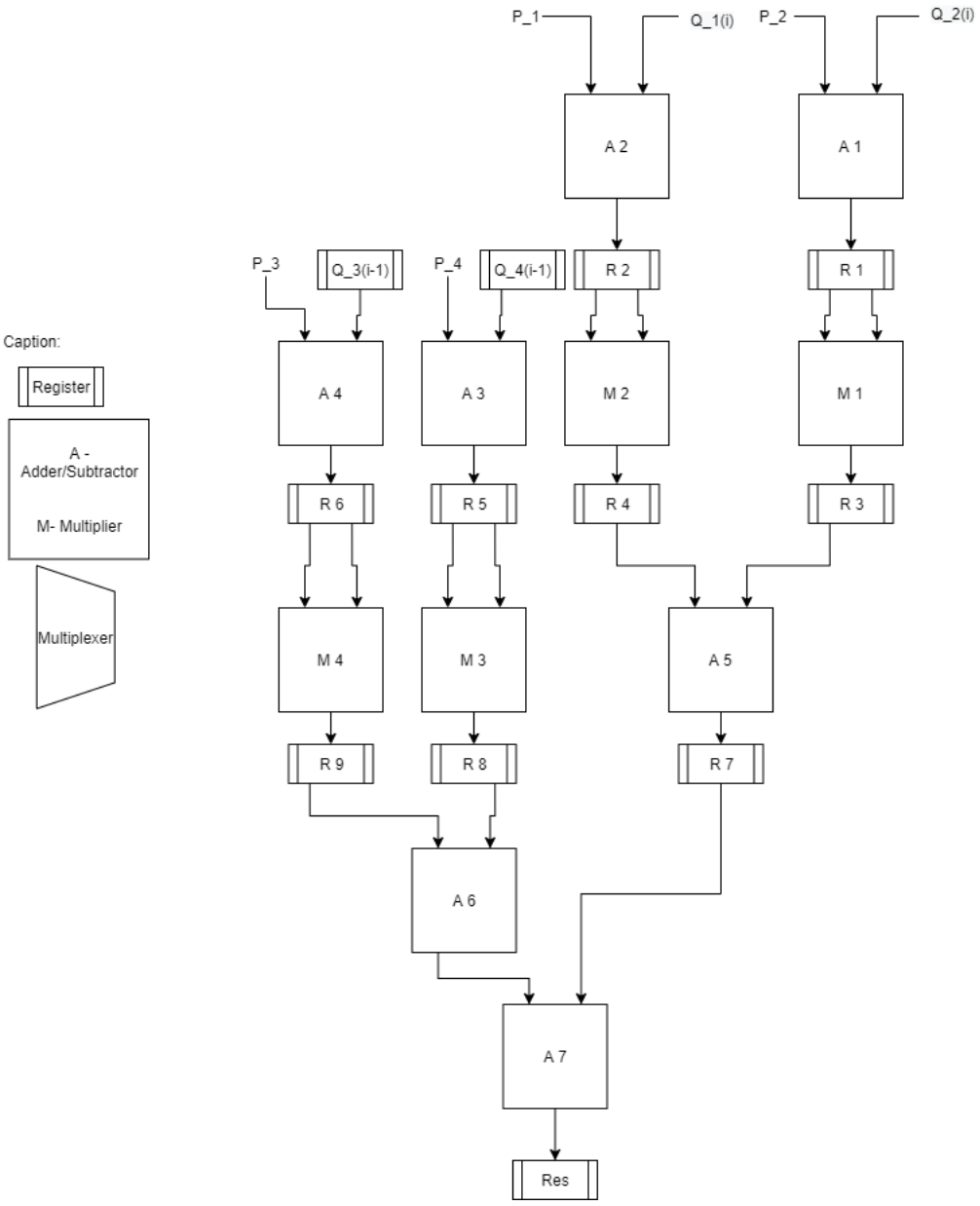

Figure 12: Resource Consumption

By reshaping the original circuit, it is possible to improve the circuit with pipelining. By drawing it like Figure 12, we can have ”circuit floors” that greatly speedup the iterations of the algorithm. Even though each iteration still takes the 4 cycles to complete, now each iteration can run concurrently.

- In each cycle, we read a line from the memory with 32 bits, and we can immediately calculate the terms with the first 16 bits of the memory (corresponding to q_1 and q_2).
- We store in the registers of the next floor the values of q_3 and q_4 and perform the operations needed to calculate the corresponding terms.
- Then, in each of the following floors, we perform each step of the algorithm with the values we have.

So, for example:

- In the first clock cycle, we only use floor one and calculate the terms corresponding to (p_1 - q_1) and (p_2 - q_2) of the first memory entry.
- In the second clock cycle, we use the first floor to calculate the terms corresponding to (p_1 - q_1) and (p_2 - q_2) of the next memory entry, but also use the second floor to calculate the values that were stored previously from the first memory entry, to calculate (p_3 - q_3) and (p_4 - q_4) as well as $(p_1 - q_1)^2$ and $(p_2 - q_2)^2$.
- In the third clock cycle, we use the first floor to calculate the terms corresponding to (p_1 - q_1) and (p_2 - q_2) of the third memory entry, but also use the second floor to calculate the values that were stored previously from the second memory entry, to calculate (p_3 - q_3) and (p_4 - q_4) as well as $(p_1 - q_1)^2$ and $(p_2 - q_2)^2$, and we now use the third floor to perform the operations related to the first memory entry, namely, $(p_3 - q_3)^2$ and $(p_4 - q_4)^2$ as well as $(p_1 - q_1)^2$ + $(p_2 - q_2)^2$.
- Finally, the final floor calculates the result from the first memory entry and the rest of the floor continue to perform the pattern explained previously.

Thus, each memory entry does take 4 cycles to calculate the algorithm result, but now the total amount of cycles needed to calculate all 100 memory positions would be 103 cycles instead of the previously implementation without pipeline, that would take 400 cycles.

Unfortunately, to achieve this result, more resources would be needed in the circuit. We would not need any of the multiplexers, but we would need 11 internal registers (an additional 7), 7 adders (an additional 5) and 4 multiplexers (an additional 2) to make all the floors work simultaneously in the pipeline architecture, and this increase in internal arithmetic operators and registers might be costly.