

---

# CS211 Lab1 Report

Name: Feng Xiao

ID Number: 2023234296

2023-11-10

---

## 1 Lab content

As computing requirements change, we need to execute more complex, long-running instructions (such as division, memory access, etc.). The execution of these instructions requires more cycles, which will lead to structural hazards. In order to alleviate the performance loss caused by structural hazards, the scoreboard algorithm was invented. This is a sequential emission and random execution architecture that allows multiple computing units to execute instructions at the same time, reducing losses caused by structural hazards. In this lab we will implement the scoreboard algorithm and quantitatively analyze its performance.

## 2 Implications of scoreboard

### 2.1 Data Structures of Scoreboard

For the selection of functional units, I use an INTUNIT to complete memory reading and writing, an ADDUNIT to process addition instructions, subtraction instructions, branch instructions and jump instructions, two MULUNITs to calculate integer multiplication, and a DIVUNIT to calculate integer division.

---

```
1  class Scoreboard{
2  public:
3      bool pause;
4      struct UnitStatus
5      {
6          uint64_t pc;
7          bool busy;
8          RISCv::FunctionalUnit name;
9          uint32_t remainingPeriod;
10         bool readMemOver;
11         RISCv::RegId fi,fj,fk;
12         RISCv::FunctionalUnit qj,qk;
13         bool rj,rk;
14     };
15     struct InstrucionStatus
16     {
17         uint64_t pc;
18         RISCv::RegId rs1,rs2,dest;
19         int64_t op1,op2,offset,out;
20         RISCv::Inst inst;
21         FunctionalUnit unit;
22         bool writeReg,writeMem,readMem,readSignExt;
23         uint32_t memLen;
24         instCompleteSchedule instSchedule;
25     };
26     vector<InstrucionStatus>instStatus;
27     UnitStatus unit[UNITNUM];
28     RISCv::FunctionalUnit registerStatus[REGNUM];
29 };
```

---

## 2.2 Instruction latency

For the settings of different instruction execution delays, I referred to the conclusions given in the Lab0 code.

Instruction Type	Unit	Latency
ALU	ALUUNIT	1
memCalc	INTUNIT	2
dataMem	INTUNIT	Changes
branchALU	ALUUNIT	1
iMul	MULUNIT	2
iDiv	DIVUNIT	6

## 2.3 Implementation of pipeline

In order to implement the scoreboard algorithm, I transformed the previous five-stage pipeline (fetch, decode, execute, memory access, write back) into a four-stage pipeline (issue, read, execute, write back).

### 2.3.1 Issue

Decode the instructions and observe the scoreboard information, mainly observing the occupancy of each functional component and the writing of the register file, to determine whether the decoded information can be stored in the corresponding component register. If the functional component corresponding to the instruction is free, and there are no other instructions to write to the target register to be written by the instruction (this is to solve the WAW hazard), then at the end of the phase, the instruction information can be stored in the component register and the scoreboard can be rewritten at the same time. , enter the instruction-related information into the scoreboard.

### 2.3.2 Read

Observe the scoreboard to see which register values are required by the current instruction, whether the value is ready, and if not, which functional unit the value is being operated on. If the value is not ready (this is to resolve RAW hazards), then the instruction is stuck in the component register and cannot read the data. If all registers can be read, then at the end of the phase, the corresponding register value will be stored in the operand register. Note that the scoreboard will not be rewritten here.

### 2.3.3 Execute

Execute the calculation process, which may last for many cycles.

### 2.3.4 Write back

At this time, you need to observe the scoreboard. If other instructions do not need to read the register to which the current calculation result is about to be written (this is to solve the WAR hazard, you need to observe all Rj and Rk. If the Rj and Rk of the relevant register are Yes, then it means There is an instruction to read the register that is currently being written. In this case, you have to wait for the previous instruction to finish reading the register and then write it back.) Then at the end of the cycle, the result will be written back to the register file and the scoreboard will be cleared. Information.

### 2.3.5 Pipeline control

The original five-stage pipeline simulator stores pipeline control information in pipeline registers. This design is reasonable and concise in sequential execution programs, but the scoreboard algorithm for out-of-sequence execution needs to transmit more complex control information. If you still choose to use pipeline registers to transfer control information, the implementation will become very complicated. I choose to save the value required for each instruction execution and the information it is in the execution stage in global variables, which will be simpler. Control the pipeline.

### 2.3.6 Other details

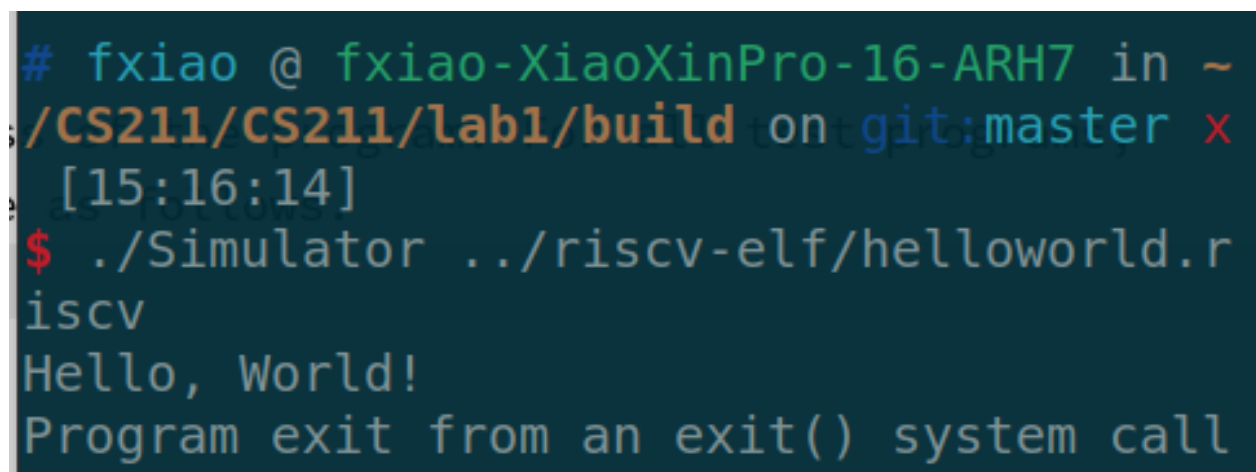
The original five-stage pipeline simulator implements branch prediction to alleviate data conflicts. However, to implement branch prediction for the out-of-order execution scoreboard algorithm, an additional buffer is needed to record the actual execution order of instructions, but this is not necessary for us to deal with. The focus of the structural adventure wasn't much help, there was no support for branch prediction functionality in the scoreboard algorithm I implemented.

## 2.4 Quantitative evaluation

During the performance comparison, in order to make the comparison as fair as possible, I canceled the branch prediction in the original simulator, but did not annotate the data forwarding code. However, this can better reflect the advantage of mitigating structural hazards brought by the scoring version algorithm.

### 2.4.1 Correctness test

I use the test program provided in lab0 to test the correctness of the program. For all test programs, the results are correct, and the resulting running results are as follows.



```
# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~  
$ /CS211/CS211/lab1/build on git:master x  
[15:16:14]  
$ ./Simulator ../riscv-elf/helloworld.r  
riscv  
Hello, World!  
Program exit from an exit() system call
```

Figuur 1: Hellow world test

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~
/CS211/CS211/lab1/build on git:master x
[19:32:17]
$ ./Simulator ../riscv-elf/test_arithme
tic.riscv
30
-10
370350
411
49380
771
Program exit from an exit() system call

```

Figuur 2: Arithmetic test

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~
/CS211/CS211/lab1/build on git:master x
[19:44:38]
$ ./Simulator ../riscv-elf/test_syscall
.riscv
This is string from print_s()
123456abc
Enter a number: 6
The number is: 6
Enter a character: a
The character is: a
Program exit from an exit() system call

```

Figuur 3: Syscall test

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/lab1/build on
git:master x [19:47:47]
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82
81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 6
0 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39
38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65
66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 8
7 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call

```

Figuur 4: Quick sort test

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/
$ ./Simulator ../riscv-elf/matrixmulti.riscv
The content of A is:
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
The content of B is:
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
The content of C=A*B is:
0 0 0 0 0 0 0 0 0 0
0 10 20 30 40 50 60 70 80 90
0 20 40 60 80 100 120 140 160 180
0 30 60 90 120 150 180 210 240 270
0 40 80 120 160 200 240 280 320 360
0 50 100 150 200 250 300 350 400 450
0 60 120 180 240 300 360 420 480 540
0 70 140 210 280 350 420 490 560 630
0 80 160 240 320 400 480 560 640 720
0 90 180 270 360 450 540 630 720 810
Program exit from an exit() system call

```

Figuur 5: Matrix multi test

```

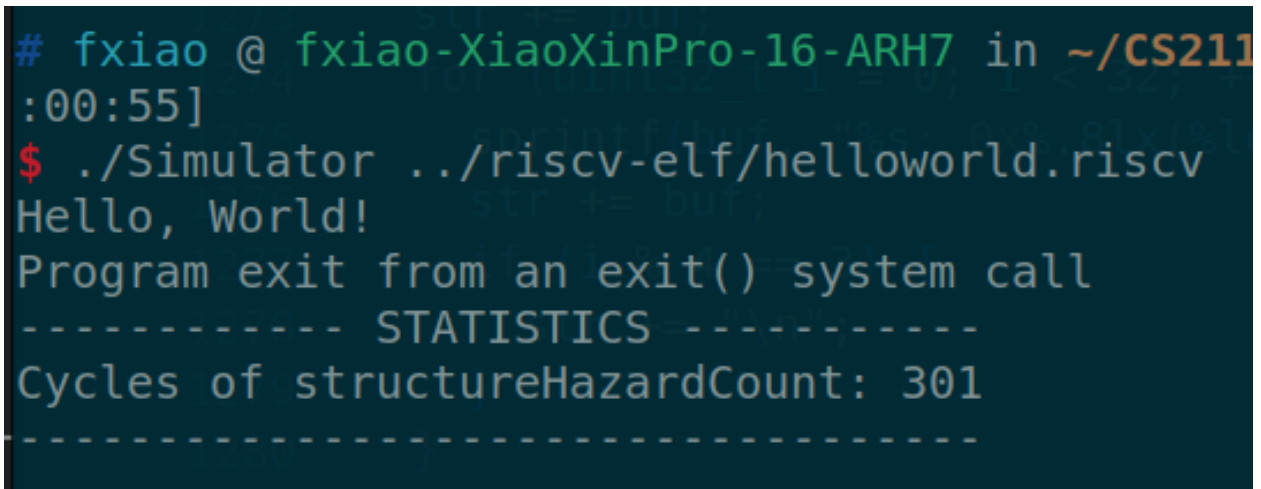
# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS
$ ./Simulator ../riscv-elf/ackermann.riscv
Ackermann(0,0) = 1
Ackermann(0,1) = 2
Ackermann(0,2) = 3
Ackermann(0,3) = 4
Ackermann(0,4) = 5
Ackermann(1,0) = 2
Ackermann(1,1) = 3
Ackermann(1,2) = 4
Ackermann(1,3) = 5
Ackermann(1,4) = 6
Ackermann(2,0) = 3
Ackermann(2,1) = 5
Ackermann(2,2) = 7
Ackermann(2,3) = 9
Ackermann(2,4) = 11
Ackermann(3,0) = 5
Ackermann(3,1) = 13
Ackermann(3,2) = 29
Ackermann(3,3) = 61
Ackermann(3,4) = 125
Program exit from an exit() system call

```

Figuur 6: Ackermann test

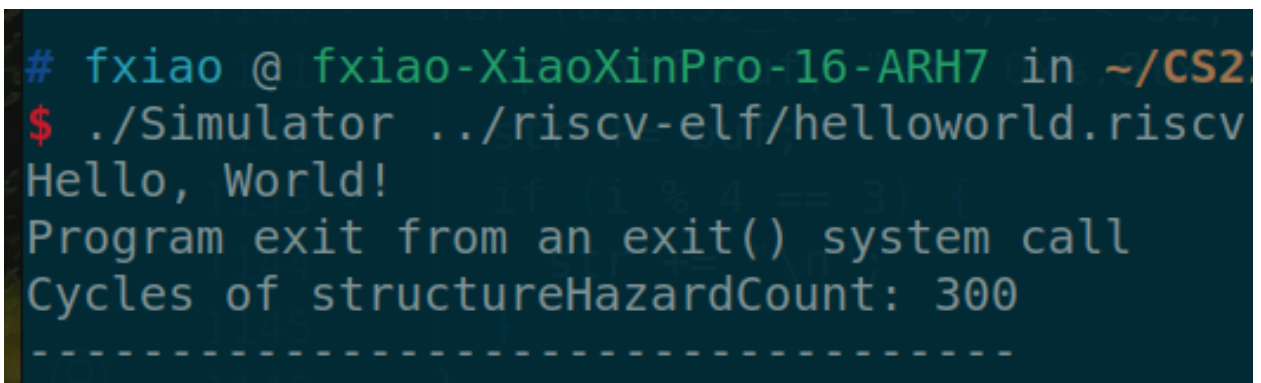
### 2.4.2 Struct hazards evaluation and hazards

After testing, we found that different indicators have different differences due to different testing procedures. This will be described in detail below.



```
# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211
:00:55]
$ ./Simulator ../riscv-elf/helloworld.riscv
Hello, World!
Program exit from an exit() system call
----- STATISTICS -----
Cycles of structureHazardCount: 301
-----
```

Figure 7: Original simulator



```
# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS2
$ ./Simulator ../riscv-elf/helloworld.riscv
Hello, World!
Program exit from an exit() system call
Cycles of structureHazardCount: 300
-----
```

Figure 8: Scoreboard

The number of structural hazards of the original simulator and the scoreboard algorithm simulator will change depending on the quality of the test program. For a simple computational sparsity test like *test/helloworld.c*, the number of structural hazards of the two is almost the same. Fig7 is the state of the original five-stage pipeline, and Fig8 is the state of the scoreboard algorithm. The two are almost the same.

But for *test/quicksort.c* this kind of test case with a lot of calculations, the structural hazard of the scoreboard algorithm will obviously become less.

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/lab1-comparion procedure/build on git:master x [22:18:11]
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
----- STATISTICS -----
Cycles of structureHazardCount: 207361
-----

```

Figuur 9: Enter Caption

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/lab1/build on git:master x [22:18:29]
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
Cycles of structureHazardCount: 165823
-----

```

Figuur 10: Enter Caption

For the CPI indicator, because the scoreboard algorithm we implemented did not include data forwarding, this, coupled with the limitations of the test program, resulted in a low CPI of the scoreboard algorithm. As shown in the picture Fig11 and Fig12 comparison

```

$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
----- STATISTICS -----
Number of Instructions: 103682
Number of Cycles: 155593
Avg Cycles per Instrction: 1.5007
-----

```

Figuur 11: Enter Caption



```

$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
----- STATISTICS -----
Number of Instructions: 103682
Number of Cycles: 353274
Avg Cycles per Instrcution: 3.4073
-----

```

Figuur 12: Enter Caption

Moreover, the data hazard tested by the original simulator was an erroneous delay. The original simulator relied on the phase after the pause to achieve multi-cycle completion of the execution phase, but it would be delivered before the result was obtained in the first round. Give the required instructions, which means that any instruction will be completed in one cycle, which is inappropriate and will not occur in the scoreboard simulator I implemented. As shown in Fig13 and Fig14, such data is unfair and unreasonable.

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/lab1-comparion procedure/build on git:master x [22:56:54]
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
----- STATISTICS -----
Number of Data Hazards: 93633
-----

```

Figuur 13: Enter Caption

```

# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/CS211/lab1/build on git:master x [23:02:54]
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Program exit from an exit() system call
----- STATISTICS -----
Number of Data Hazards: 258135
-----

```

Figuur 14: Enter Caption

### 3 Summary

- This lab uses the scoreboard algorithm to alleviate the structural hazards caused by multi-cycle instructions. Although I only added a small number of functional units to my implementation, in some computationally intensive test cases, the structural hazards has been greatly improved, and the scoreboard algorithm is an scalable algorithm that can be expanded by adding functional units to further improve performance.
- With the help of the teacher, I also gained a new understanding of some details on the implementation of the scoreboard algorithm. As for the stage at which the functional unit should be judged to be idle, some students mentioned that doing it in the execution stage can reduce the waiting period. After thinking about it, I agreed with this point of view, but the teacher later explained that this is to make the implementation of the bypass simpler.