# A Little Riscy: Implementation of a Simple Superscalar RISC-V Processor

Severin Jäger
severin.jaeger@tuwien.ac.at
Mat.Nr. 01613004

Max Tamussino
e1611815@student.tuwien.ac.at
Mat.Nr. 01611815

***Abstract*—*A Little Riscy* is a minimalistic superscalar RISC-V processor featuring parallel execution of ALU and load/store instructions. Due to its four-stage pipeline it is able to efficiently handle data hazards, however its performance decreases drastically when branching is involved. Thus, it shows that superscalar processors do not only need parallel execution units, but also compiler support and branch prediction with speculative execution to achieve a high IPC outside artificial mixes of instructions.**

## I. Introduction

The open RISC-V instruction set architecture [1] has gained significant popularity both in academia and in industry. Within the *Advanced Computer Architecture* course at TU Wien, a minimalistic superscalar RISC-V processor called *A Little Riscy* has been designed. It implements the RV32I instruction set and allows parallel execution of ALU and load/store instructions with in-order dual issue.

Due to the limited scope of this project, several instruction level parallelism techniques were not implemented. These include branch prediction and speculative execution as well as out-of-order issuing.

This report covers the implemented architecture of the processor in Section II, some evaluation results in Section III and a discussion of the design including potential improvements in Section IV.

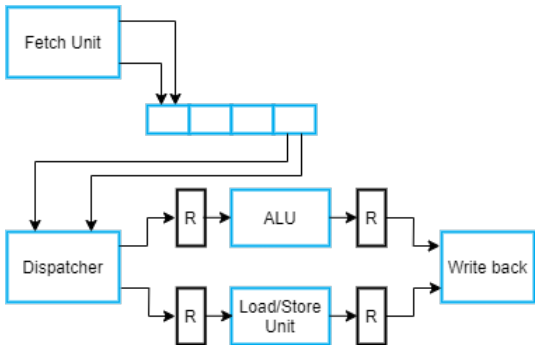


Fig. 1. Basic structure of the implemented processor [2]

## II. Implementation

The implemented processor follows the architecture depicted in Figure 1. Thus, it features a Harvard architecture with a four stage pipeline (Fetch, Dispatch, Execute, Write Back) with an ALU and a load/store unit in parallel in the execution stage. In order to utilise the parallel execution units, two instructions per cycle are fetched, enqueued, dequeued and dispatched respectively. The whole design was implemented using the Chisel hardware description language.

As the evaluation of the processor was conducted with rather simple single-threaded programs (cf. Section III), the following instructions were not implemented and are thus interpreted as NOP: FENCE, ECALL, EBREAK. Otherwise, *A Little Riscy* is except for some memory limitations (cf. Section II-G) compliant with the RV32I instruction set.

### A. Fetch Unit

The fetch unit loads two instructions per cycle from the instruction memory and places them into the instruction queue. Additionally, it administrates the PC. In case the queue is full, the whole fetching process is stalled.

Furthermore, it implements all control flow transfer instructions. This implies the following:

- As there is no ALU in a previous pipeline stage, additional hardware is required for address calculations.
- To ensure the correct register values are present while calculating addresses, the fetch unit has to wait for all previous instructions to take effect before the calculation. This implies that no instructions can be queued until the whole pipeline is idle.

The latter can in principle be mitigated using branch prediction and speculative execution techniques, however this was outside the scope of this project.

In case a jump or branch instruction is dequeued, the fetch unit performs the following steps.

- Stall the pipeline (by scheduling NOPs)[1].
- Wait for the dispatcher to report an empty pipeline (i.e. all register values are written).
- Decode the branch instruction, calculate the target and determine whether the branch is taken.
- Enqueue an ADDI instruction writing the address of the instruction after the jump (only applicable for jumps).
- Continue with regular fetch operation.

[1]The instruction loaded from the instruction memory is still dispatched in case its PC is lower than the one of the branch instruction.

Submitted January 22, 2021

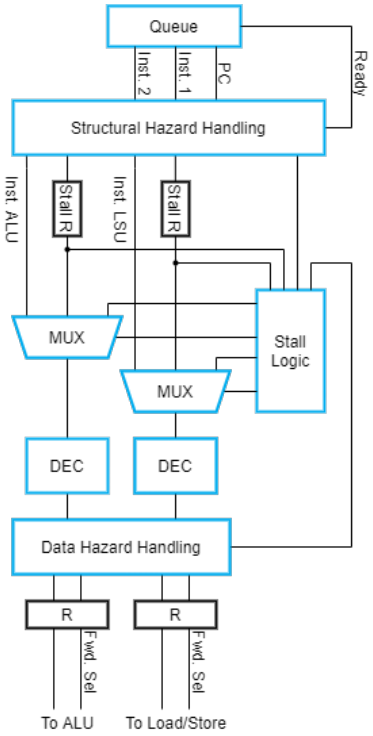


Fig. 2. High-level schematic of the implemented dispatcher

### B. Instruction Queue

The instruction queue is a simple register based FIFO (cf. the `RegFifo` in [3]) with a width of 96 bits ($2 \times 32$ bits instruction, 32 bits PC) and a depth of 4. This brings the limitation that only the two instructions being enqueued together can be dequeued at the same time. In case of structural hazards (i.e. only one of the loaded instructions can be issued), this leads to a serialised processing.

### C. Dispatcher

The dispatcher dequeues two instructions and issues them to the execution units (ALU and load/store). It is of utmost importance, that structural and data hazards are resolved beforehand. This is done by the means of operand forwarding and stalling of conflicting instructions.

The treatment of structural hazards is done as the following: In case two instructions for the same execution unit or with identical destination register are dequeued, the first one is issued while the second is stalled in special stall registers (s. Figure 2). In the next cycle, no new instructions are dequeued and the stalled instruction is issued.

Afterwards, the instructions are decoded (in parallel for ALU and load/store), then data hazards can be treated. This is done by comparing the operand register addresses with the destination register of instructions in the current and the last cycle. In case there is a data hazard related to an instruction dispatched in the last cycle, operand forwarding can be used to resolve this issue without stalling the pipeline[2] So in this case, the dispatcher only has to calculate the forwarding signals shown in Figure 2.

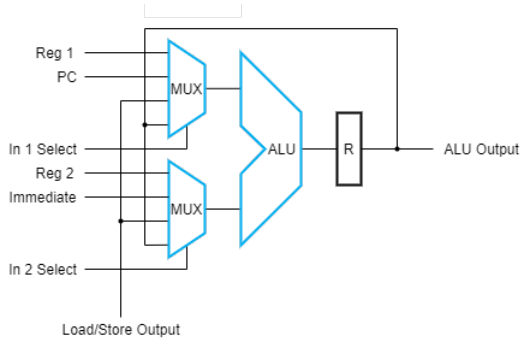


Fig. 3. ALU with input multiplexers for operand forwarding

However, in case instructions decoded during the same cycle impose some ordering constraints, the instruction with the higher PC has to be stalled. This is done in the same fashion as the instruction stalling for structural hazards and delays instructions for one cycle.

In principle, all aforementioned hazards can be resolved by implementing an dynamic scheduling approach like Tomasulo´s algorithm. However, this is not suitable for the chosen architecture. This is due to the common data bus, which broadcasts the results of all calculations to the reservation stations and the register bank. This bus need some arbitration policy to prevent two execution units from broadcasting their results simultaneously. However, in the case of *A Little Riscy* both execution units generate one result per cycle respectively[3]. Thus, the bandwidth of the basic single-word bus is not sufficient. This leads to a major bottleneck for the whole pipeline.

### D. Arithmetic Logic Unit

This execution unit implements all instructions listed in Section 2.4 of the RISC-V Specification [1]. The LUI and the AUIPC instruction as well as the operand forwarding described in Section II-C require a dynamic allocation of the operands. This is achieved by the means of multiplexers with select signals created by the dispatcher. This is depicted in Figure 3.

### E. Load/Store Unit

The load/store unit handles all accesses to the data memory. It implements all respective instructions in the RV32I instruction set, the only limitations are discussed in the next section. Similar to the ALU, its inputs feature multiplexers to allow for the use of forwarded values.

[2]This is not the case for the standard five stage RISC pipeline proposed in [4].

[3]This is not the case for floating point units, where individual instructions usually take multiple cycles.

### F. Registers

The register bank consists in compliance with the RISC-V reference manual out of 32 general purpose registers (x0 is always set to zero) and the program counter. The architecture demands six read ports (two for the ALU, two for the load/store unit and two for the fetch unit) and two write ports (one for ALU and load/store unit respectively).

### G. Memory System

As *A Little Riscy* implements a Harvard architecture, instruction and data memory are separated. The instruction memory is a read-only type, holds 128 word (and thus instructions), and is only word addressable. Due to their small size, both can be implemented as registers. In contrast the 256 word data memory allows byte, halfword and word addressing, however it assumes aligned halfwords and words both for reads and writes.

### H. Performance Considerations

From the previous sections, the following statements about the theoretically achievable performance can be made:

- As the data memory is implemented as register, there is no memory delay, so load/store instructions take as long as ALU instructions.
- In the absence of pipelining hazards, two instructions per cycle can be executed, thus a IPC of 2 is the upper bound for this processor.
- Instructions with data hazards can be executed without any additional delay as long as they are not fetched simultaneously.
- Branching requires completion of all previous instructions and thus introduces significant idle time.
- The instruction queue adds latency to the pipeline, this is unfavourable in the beginning and during branches.

Thus, efficient code for *A Little Riscy* consist of interleaved load/store and ALU instructions and minimises the number of control flow transfer instructions. The following section investigates these observations empirically.

## III. Evaluation

### A. Benchmarks

In order to evaluate the performance of the implemented processor some benchmarks were conducted. Thereto short C programs were compiled using the RISC-V GNU Compiler Toolchain. All experiments were conducted in a Chisel Tester based simulation environment.

The first benchmark aims at evaluating the performance of the processor with specialised ALU load. Listing 1 shows the respective C code which was compiled with the `-O3` flag. Afterwards, the instructions loading the value of the global variables used as input[4] from memory were replaced by simpler load immediate instructions.

[4]This is done in order to exploit the compilers optimiser without losing the actual implementation.

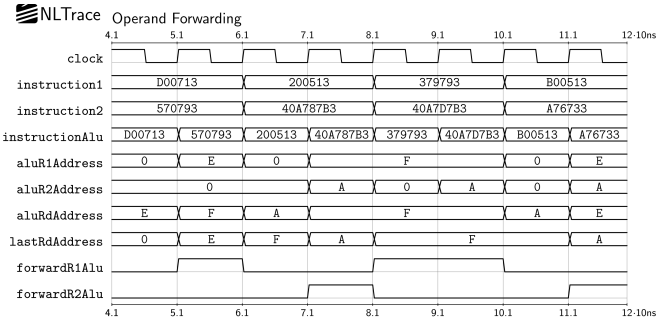


Fig. 4. Dispatcher signals for operand forwarding in the ALU benchmark

```
int A = 13;
int B = 2;
int C = 11;
int D = 7;

int main() {
    int a = A;
    int b = B;
    int c = C;
    int d = D;
    int r = ((((a+5-b) << 3) >> b) -
            ((c | a) - 9)) ^ (d+2);
}
```

Listing 1. Code for the ALU benchmark

The code for the second benchmark is depicted in Listing 2. It consist of two loops, one setting the elements of an array to `i+1` and the second one is summing up its elements. This is compiled with the `-O1` flag. The resulting assembly code contains numerous branch instructions due to the implemented loops. Thus, frequent control flow transfers occur.

```
int N = 5;
int main() {
    int n = N;
    int a[5] = {};

    int i = 0;
    for(;i<n;i++) {
        a[i] = i+1;
    }
    int r = 0;
    for(i=0;i<n;i++) {
        r += a[i];
    }
    return r;
}
```

Listing 2. Code for the loop benchmark

The third benchmark uses the code from Listing 2 again, however it was compiled with the `-O3` flag, which resulted in an unrolled loop. Some minor rearrangements have been made in order to ensure efficient utilisation by interleaving of ALU and load/store instructions.

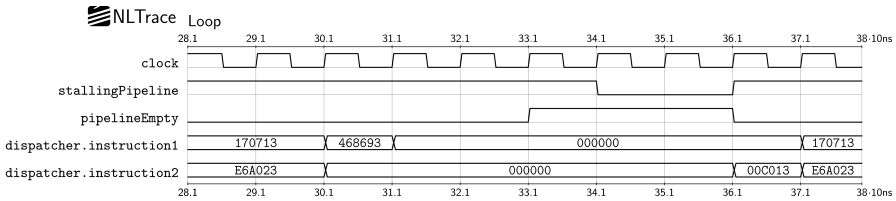


Fig. 5. Dispatcher signals in the loop benchmark. No instructions are scheduled during branches, this leads to significant idle time.

The benchmarking results are summed up in Table I. Note that the latency of the pipeline is subtracted from the measured number of cycles in order to measure independent of the length of the code. Thus, the maximum theoretically achievable IPC is 2.

TABLE I
BENCHMARKING RESULTS

| Benchmark | Instructions | Cycles | IPC |
|---|---|---|---|
| ALU | 13 | 13 | 1 |
| Loop | 52 | 108 | 0.48 |
| Loop Unrolled | 27 | 21 | 1.29 |

The ALU benchmark, which consists only of ALU instructions, leads to an IPC of 1. Note that the ALU instructions have to be serialised because there is only one ALU available, the load/store unit is idle during the whole benchmark. Thus, more instructions per cycle are not achievable. This maximum could be reached due to the implementation of operand forwarding. This technique enables handling of the occurring data hazards without introducing any delay. This is shown exemplarily in Figure 4.

The simple loop benchmark shows the weak spots of the processor. The IPC drops significantly below 1, this is mainly due to the negative effects of branching (cf. Section II-A). The waveforms in Figure 5 shows one iteration of the first loop and indicates that branch instructions force the pipeline to be idle for significant timespans.

The unrolled loop is able to cope with this problem. As branch instructions were removed, the performance is limited by the distribution between ALU and load/store unit instructions. As long as one load/store and one ALU instruction is issued per cycle, the IPC is 2, it only drops within phases of sequential ALU or load/store instructions or when destination register conflicts occur. This is shown in Figure 6.

### B. FPGA Evaluation

An Altera DE2-115 evaluation board featuring an Altera Cyclone IV FPGA was used as hardware platform. The resource usage of the implemented processor is shown in Table II. The notably high number of logic required for the register bank is due to the high number of read and write ports (cf. Section II-F).

The maximum available clock frequency for this design is 29.1 MHz. The critical path runs through the dispatcher (between input and output of the stall registers, s. Figure 2). This is hardly surprising as the dispatcher has a rather complex task. Finer pipelining i.e. splitting the dispatcher into two stages might be a solution to increase the clock frequency.

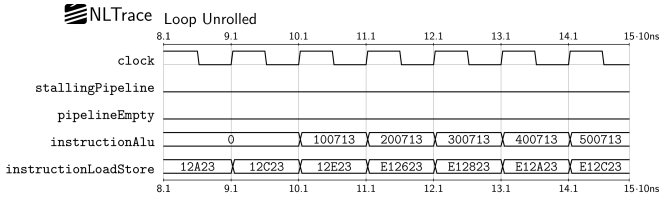


Fig. 6. Dispatcher signals in the unrolled loop benchmark. The first two instructions are both load/store instructions, they are therefore executed sequentially. Afterwards, ALU and load/store instructions are interleaved, the design is exploited efficiently.

TABLE II
RESOURCE CONSUMPTION OF THE DIFFERENT UNITS ON THE TARGET FPGA

| Entity | LUTs | Registers | Logic Cells |
|---|---|---|---|
| ALU | 1929 | 38 | 1967 |
| Dispatcher | 357 | 191 | 548 |
| Fetch Unit | 501 | 55 | 556 |
| Instruction Memory | 121 | 0 | 121 |
| Load/Store Unit | 864 | 69 | 933 |
| Data Memory | 2431 | 8192 | 10623 |
| Instruction Queue | 12 | 342 | 354 |
| Register | 1125 | 1024 | 2149 |
| Total | 7340 | 9825 | 17165 |

## IV. CONCLUSION AND OUTLOOK

During this project, a simple superscalar RISC-V processor with one ALU and one load/store unit in parallel was implemented. It is capable of executing up to two instructions per cycle, however benchmarks have showed that this requires a rather artificial mix of instructions. Especially branching causes significant performance drops as neither branch prediction nor speculative execution were implemented.

In general, it can be concluded that instruction level parallelism is an interesting way to increase the performance of a processor, however there are two requirements that have to be satisfied in order to reach an IPC significantly above 1:

- The compiler has to optimise with the architecture in mind and make sure that instructions are ordered in a suitable way for superscalar machines. This mainly involves suitable loop unrolling and interleaving of different instruction types.
- Going superscalar does not only require parallel execution units, but an all-in approach covering several modern ILP

techniques (like the ones presented in. [4]). This includes branch prediction and speculative execution in order to reduce the branch penalty as well as out-of-order issuing. However, the come at the cost of additional transistors as well as significantly increased design complexity.

## REFERENCES


[1] Andrew Waterman and Krste Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, 2019.

[2] C. Hamacher, Z. Vranesic, S. Zaky, and N. Manjikian, *Computer Organization and Embedded Systems*, 6th Edition. McGraw-Hill Publishing, 2011.

[3] M. Schoeberl, *Digital Design with Chisel*. Kindle Direct Publishing, 2019. [Online]. Available: https://github.com/schoeberl/chisel-book.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th Edition. Morgan Kaufmann Publishers Inc., 2017.