Web-Based Simulator of Superscalar RISC-V Processors

Jiri Jaros, Michal Majer, Jakub Horky, and Jan Vavra
Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
jiri.jaros@vut.cz

SUPERCOMPUTING TECHNOLOGIES GROUP

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Motivation

- **Bridge the Educational Gap**

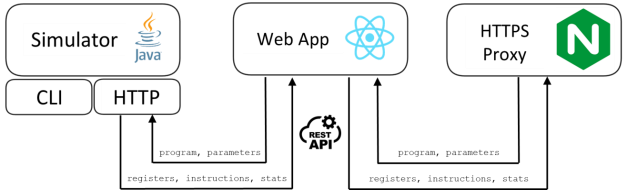
  Provide HPC developers with an accessible, hands-on tool to explore and understand superscalar RISC-V processor architectures.

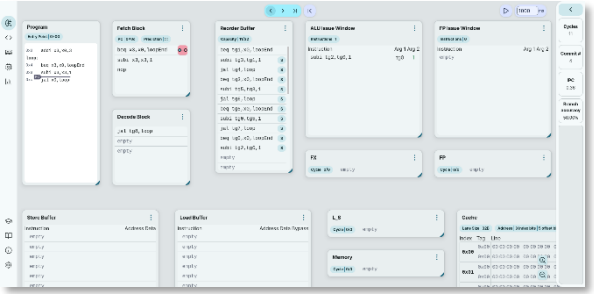
- **Empower Developers**

  Equip users with skills to design processors and optimize code for performance, cost efficiency, and power consumption.

- **User-Friendly Customization**

  Enhance the learning experience with a customizable interface and robust tools for performance analysis to address modern computing challenges.





Live Demo

Key Features

- **User-Friendly Interface**

  Simple and illustrative web presentation with detailed information on each block and instruction.
- **Fully Configurable Processors**

  Customize issue width, register files, reorder, load and store buffers, branch predictors, functional and memory units, along with cache memory settings including size, associativity, cache line size, and replacement strategy.
- **Forward and Backward Simulation**

  Flexibility to simulate in both directions for thorough analysis.
- **GCC Compiler Interface**

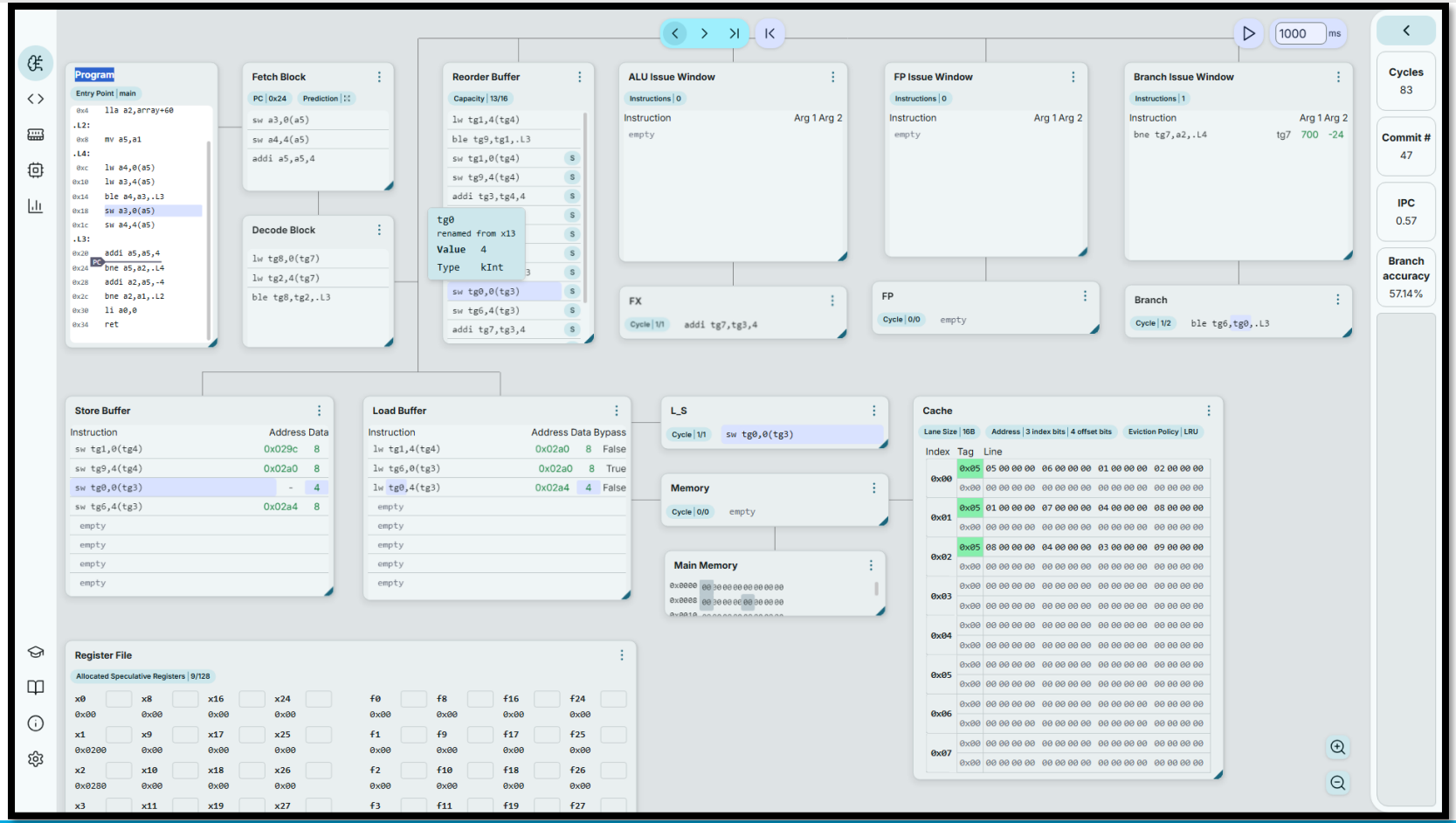
  Build C code into assembly using various optimization levels with syntax highlighting and pairing between C and assembly code.
- **Comprehensive Performance Statistics**

  Access static and dynamic metrics such as FLOPs, IPC, branch prediction accuracy, unit utilization and cache hit rate.
- **Benchmark CLI:**

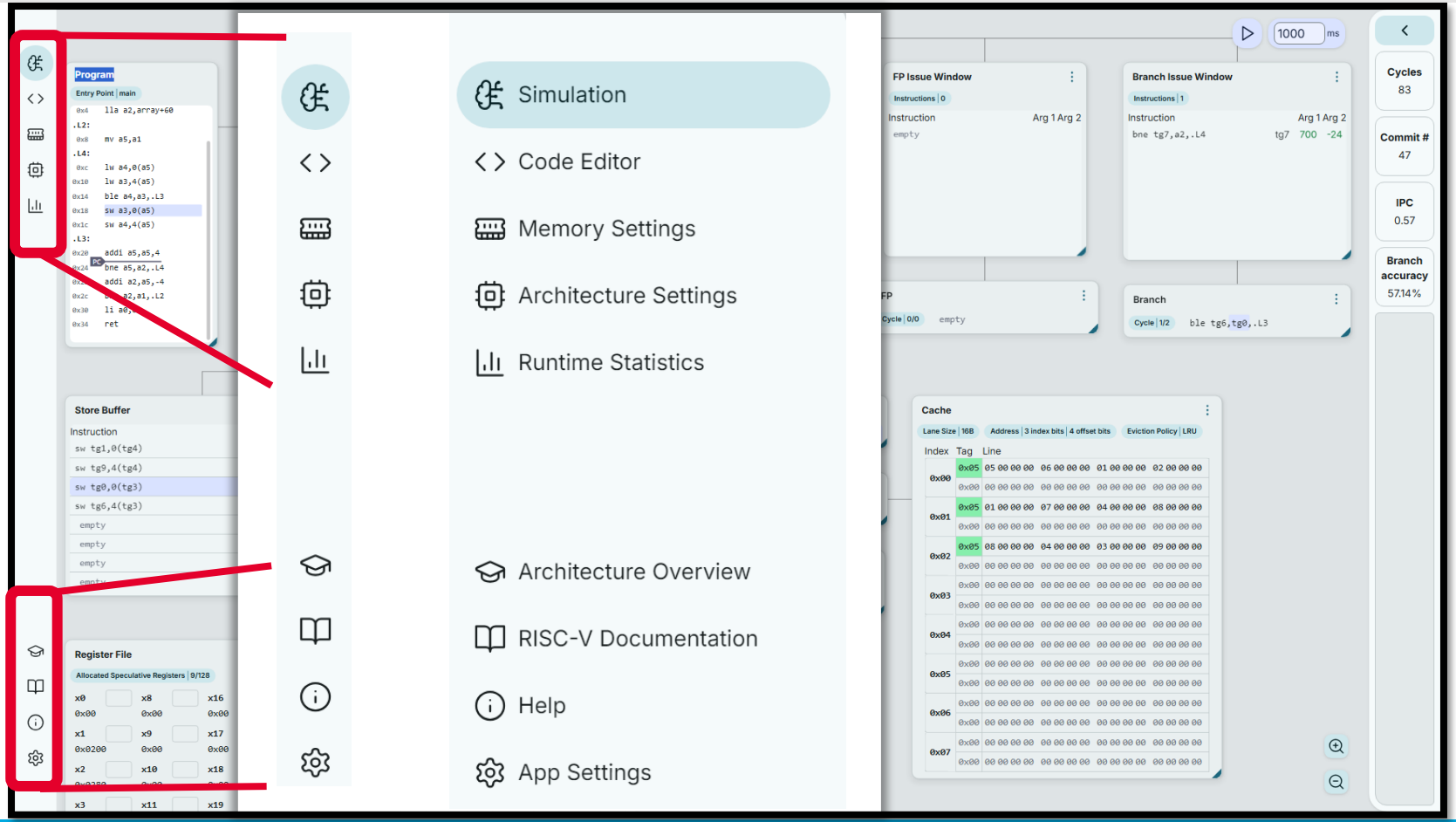
  Command-line interface for benchmarking complex programs.

Web-Based User Interface

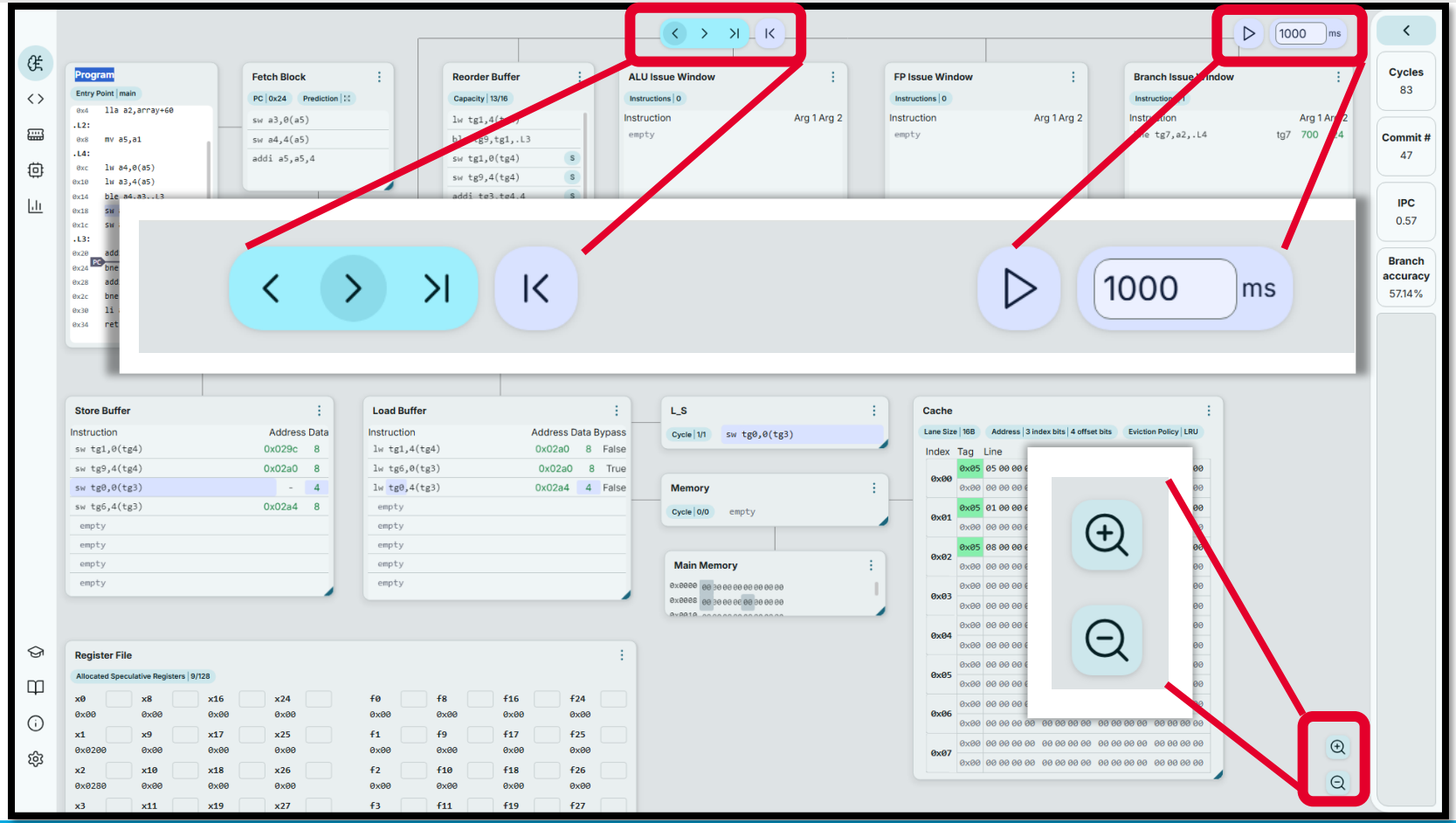
Graphical User Interface – Simulation View



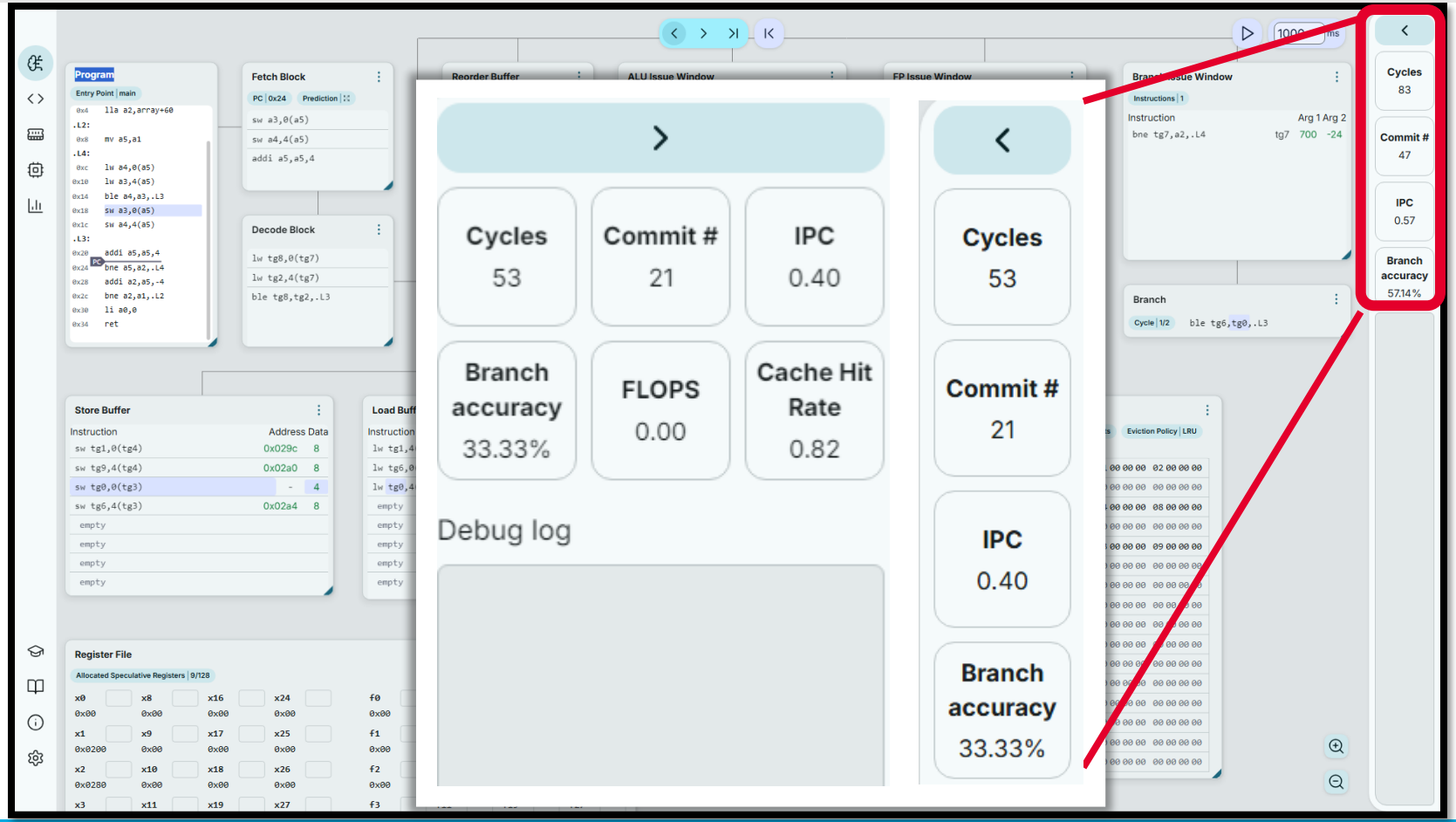
Graphical User Interface – Menu Panel



Graphical User Interface – Simulation Navigation

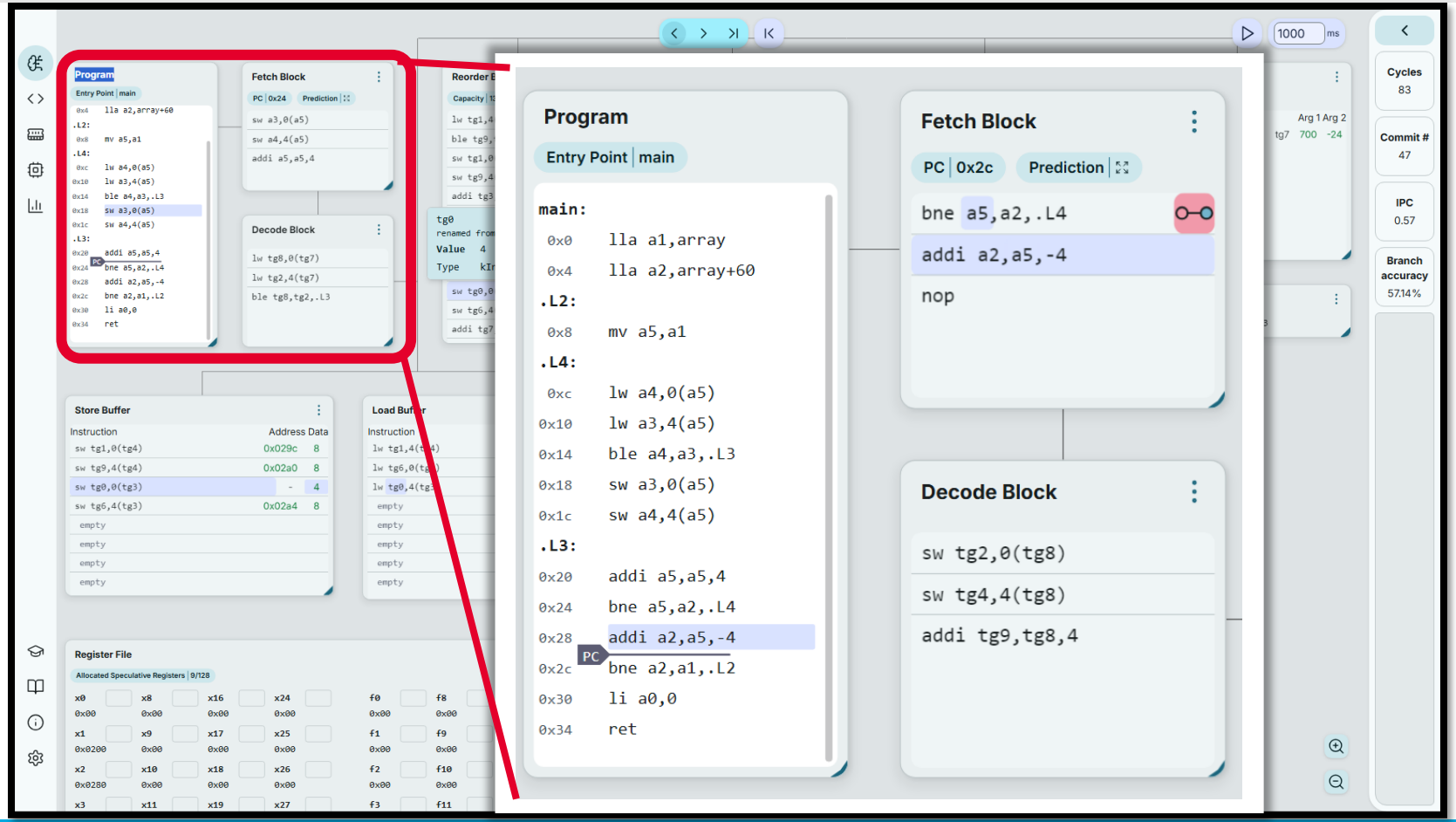


Graphical User Interface – Simulation Status

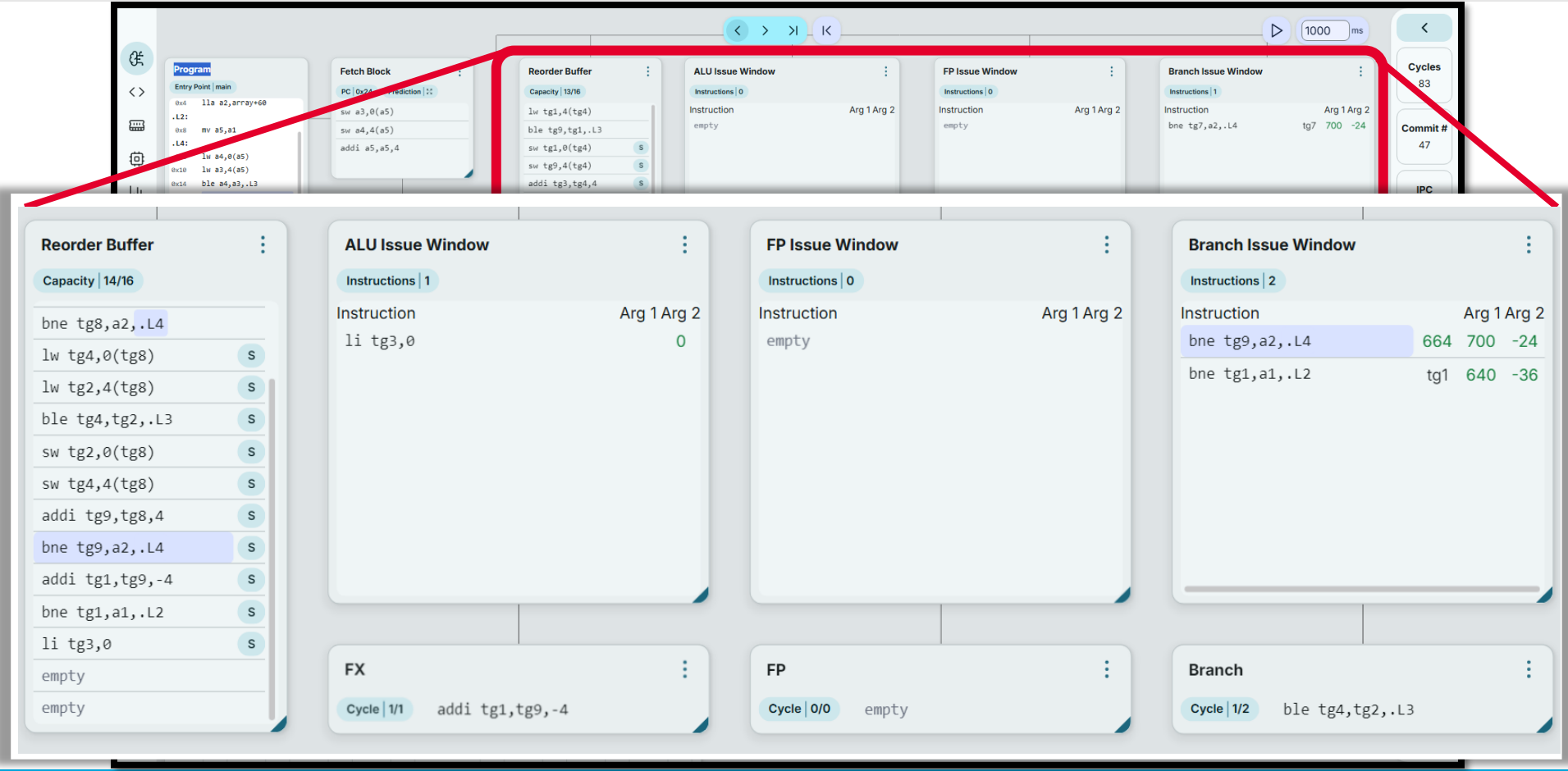


Simulation View

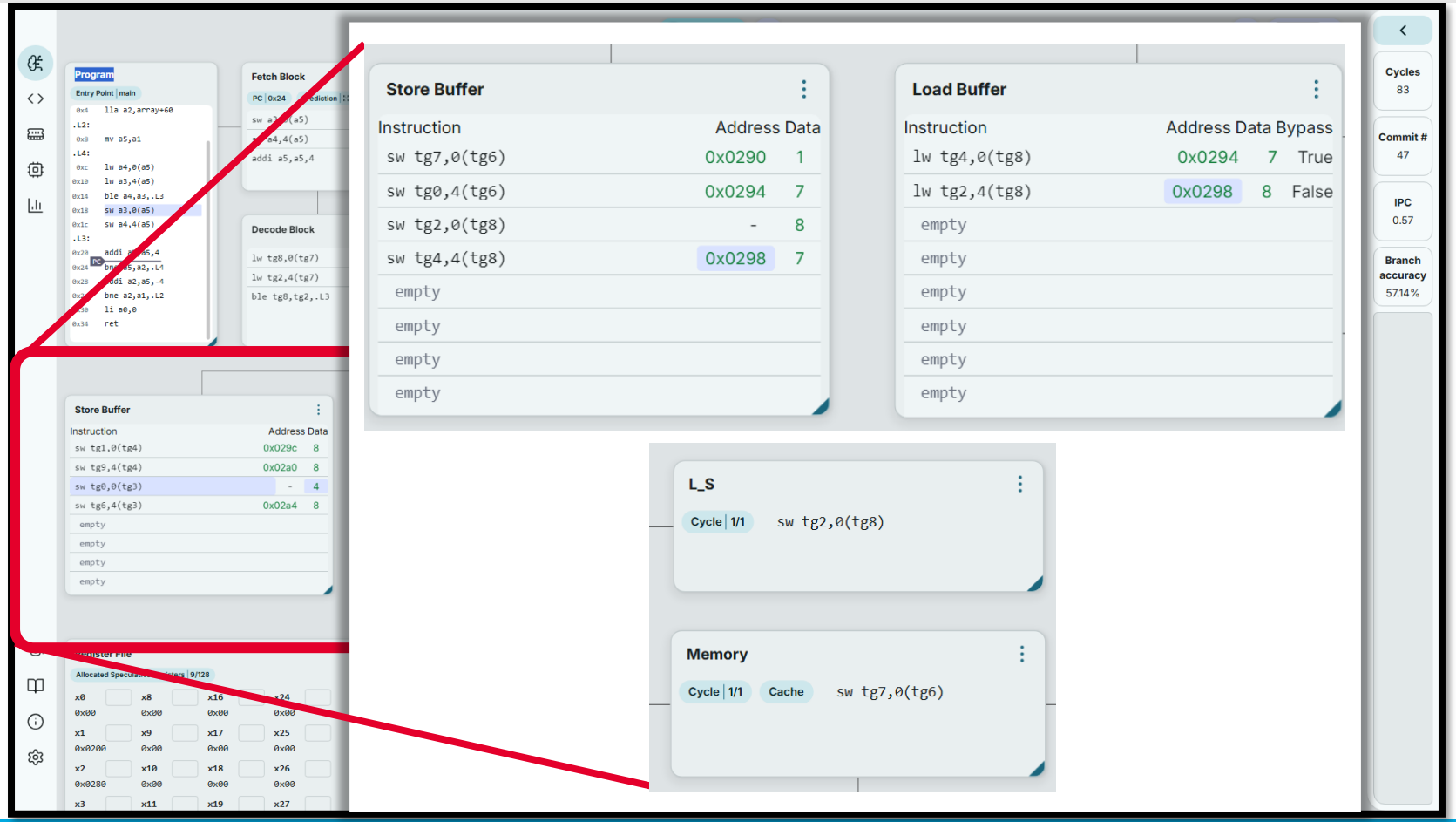
Graphical User Interface – Processor Front-End



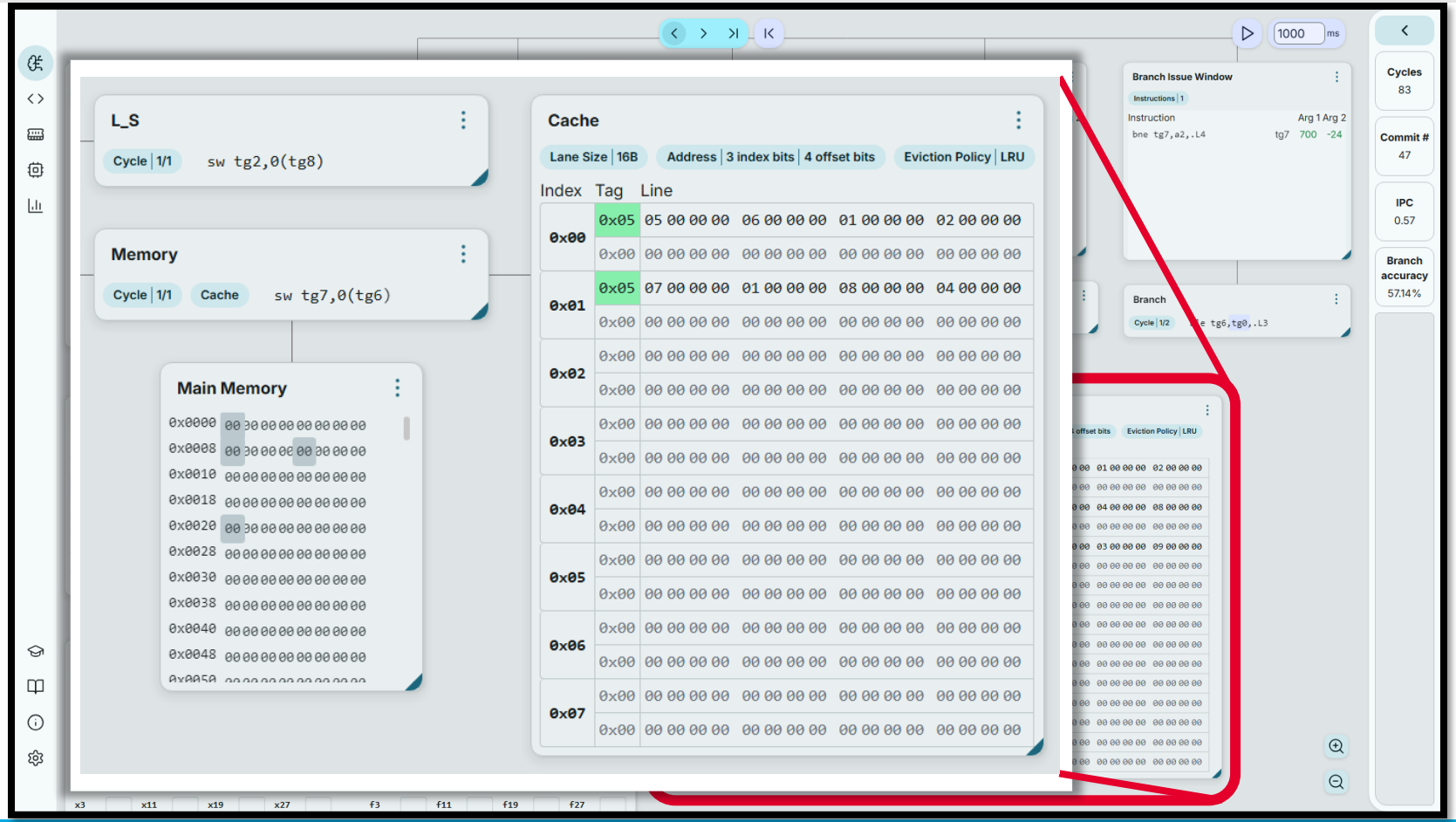
Graphical User Interface – Out-of-Order Engine



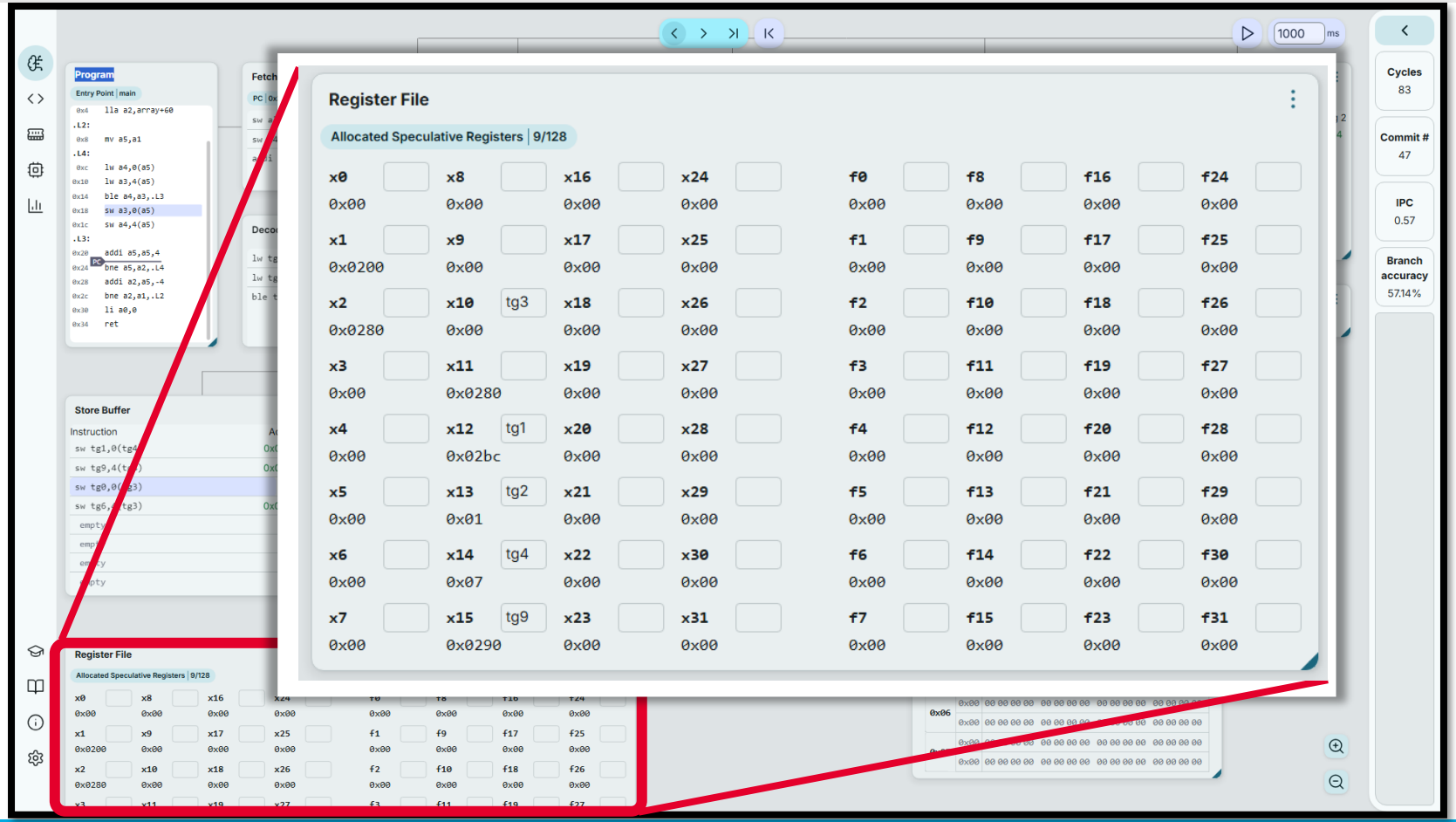
Graphical User Interface – Load and Store Units



Graphical User Interface – Memory Subsystem



Graphical User Interface – Register Fields



Processor Configuration

Processor Configuration – OoO and Func. Units Config

Name | **Buffers** | Functional Units | Cache | Memory | Branch

Buffers

Re-order buffer size ⓘ
16

Commited instructions per cycle ⓘ
4

Flush penalty ⓘ
1

Fetched instructions per cycle ⓘ
3

Branch follow limit ⓘ
1

Name | Buffers | Functional Units | Cache | Memory | **Branch**

Branch Prediction

Branch target buffer size ⓘ
16

Pattern history table size ⓘ
10

Predictor type in PHT
Zero bit | One bit | **Two bit**

Predictor default state ⓘ
Strongly Not Taken | Weakly Not Taken | Weakly Taken | Strongly Taken

☐ Use global history vector

Name | Buffers | **Functional Units** | Cache | Memory | Branch

Functional Units

Unit	Latency	Operations	
FX	2	Bitwise | 1 · Addition | 1 · Multiplication | 2 · Division	✕
FP	2	Bitwise | 1 · Addition | 1 · Multiplication | 2 · Division	✕
L_S	1	N/A	✕
Branch	2	N/A	✕
Memory	1	N/A	✕

Add a Unit

FX | FP | L_S | Branch | Memory

Name
Base latency
1

This unit can perform:

- ☐ Addition (+) 1
- ☐ Bitwise (and, or, xor) 1
- ☐ Division (/) 1
- ☐ Multiplication (*) 1
- ☐ Special (sqrt, ...) 1

Add Unit

Processor Configuration – Cache and Memory

Name | Buffers | Functional Units | **Cache** | Memory | Branch

Cache

☑ Use cache

Cache lines ⓘ
16

Cache line size (B) ⓘ
16

Cache associativity ⓘ
2

Cache replacement policy
LRU | FIFO | Random

Store behavior ⓘ
write-back | write-through

Lane replacement delay ⓘ
10

Cache access delay ⓘ
1

Name | Buffers | Functional Units | Cache | **Memory** | Branch

Memory

Load buffer size ⓘ
8

Store buffer size ⓘ
8

Store latency ⓘ
1

Load latency ⓘ
1

Call stack size ⓘ
512

Number of speculative registers ⓘ
128

Code Editor

Code Editor

Optimization

- ○ Do not optimize
- ◉ Optimize (O2)
- ○ Optimize (O3)
- ○ Optimize for size

Compile

Load Example

Show Memory

Tips for writing code

C Code | Load | Save

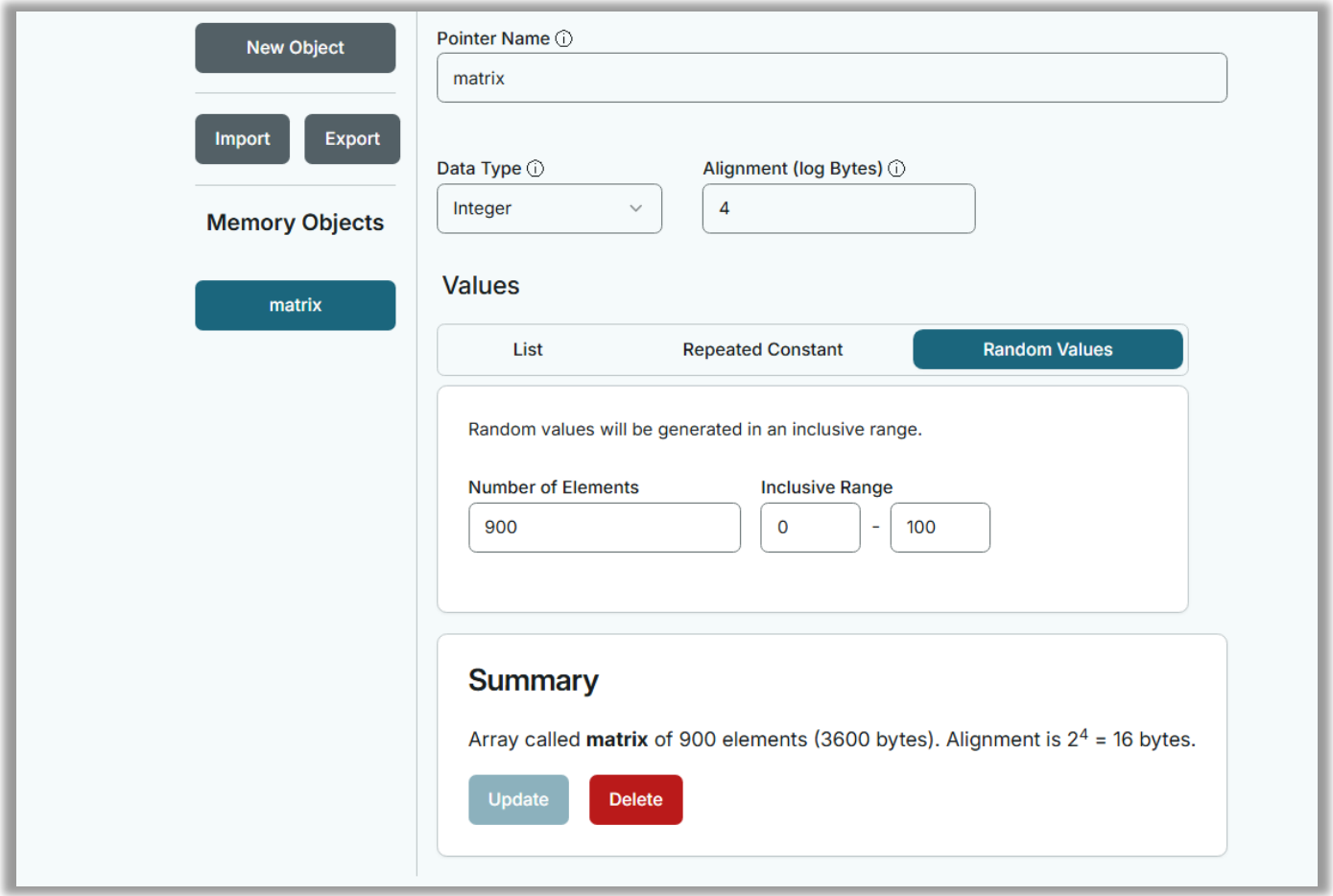
```c
// 1) Define your own matrix in memory tab
extern int matrix[];
// 2) Change the dimensions
#define rows_c 30
#define cols_c 30
int result[rows_c*cols_c] = {0};

// Function to transpose a matrix
void transpose(int *dst, int *src, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            dst[j * rows + i] = src[i * cols + j];
        }
    }
}

int main() {
  transpose(result, matrix, rows_c, cols_c);
}
```

ASM Code | Load | Save

Check | Entry Point: main

```asm
transpose:
    ble a2,zero,.L1
    slli t3,a3,2
    add a6,a1,t3
    slli a7,a2,2
    li t1,0
.L3:
    sub a5,a6,t3
    mv a4,a0
    ble a3,zero,.L6
.L4:
    lw a1,0(a5)
    addi a5,a5,4
    sw a1,0(a4)
    add a4,a4,a7
    bne a5,a6,.L4
.L6:
    addi t1,t1,1
    addi a0,a0,4
    add a6,a6,t3
    bne a2,t1,.L3
.L1:
    ret
main:
    addi sp,sp,-16
    lla a0,result
    li a3,30
```

Memory Editor



Simulation Statistics

Simulation Statistics – Basic Overview

IPC

0.52

Clocks

58

Branch Prediction Accuracy

55.56%

Static Instruction Mix

Instruction mix of the program

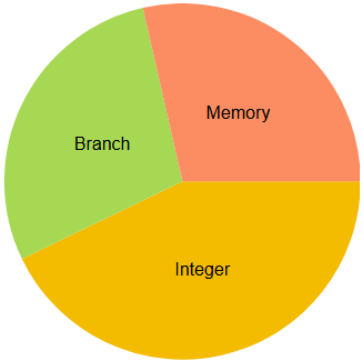


Table | Chart

Dynamic Instruction Mix

Instruction mix of committed instructions

Category	Value	Proportion
Integer	7	23.33%
Float	0	0.00%
Branch	9	30.00%
Memory	14	46.67%
Other	0	0.00%

Table | Chart

Functional Units

Amount of time a unit spends computing

Name	Busy Cycles	Utilization
L_S	27	46.55%
FX	19	32.76%
Branch	0	0.00%
Memory	17	29.31%
FP	0	0.00%

Simulation Statistics – Advanced View

Detailed Statistics

Attribute	Value
Prediction Accuracy	0.56
Committed Instructions	30
Clock Cycles	58
Flushed Instructions	31
ROB Flushes	4
Correctly Predicted Branches	5
Conditional Branches	9
Taken Branches	6
Max Allocated Registers	10
Arithmetic Intensity	0.23
FLOPS	0.00
IPC	0.52

Cache Statistics

Attribute	Value
Read Accesses	10
Write Accesses	5
Hits	13
Misses	2
Total Delay	0
Bytes Written	20
Bytes Read	40

Instruction Statistics

Heat Map of execution counts

Cache Hits | Committed

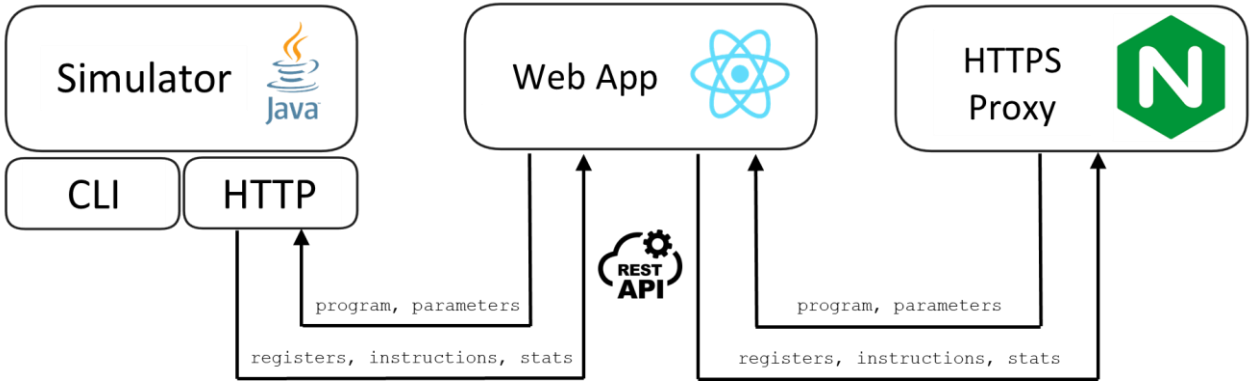
```
main:
lla a1,array
lla a2,array+60
.L2:
mv a5,a1
.L4:
66.67%  lw a4,0(a5)
83.33%  lw a3,4(a5)
ble a4,a3,.L3
100.00% sw a3,0(a5)
100.00% sw a4,4(a5)
.L3:
addi a5,a5,4
bne a5,a2,.L4
addi a2,a5,-4
bne a2,a1,.L2
li a0,0
ret
```

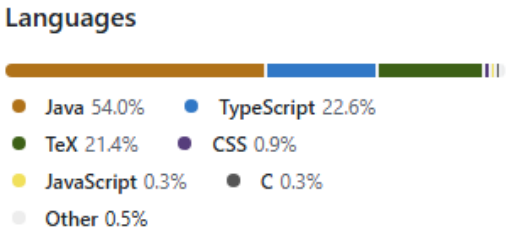
Implementation and Deployment

Under the Hood

- **Fully Containerized Solution:**
  Implemented in Docker for seamless deployment and scalability.
- **Extensive Static Unit Testing**
  Achieves 83% code coverage.
- **Robust Dynamic Testing:**
  Supports 100 concurrent users with a median latency under 1.2 seconds on an Intel i5-8300 laptop with 16GB RAM.

Source Codes





Conclusions

- **Accessibility**

  The simulator simplifies understanding of superscalar RISC-V processors for education and research.
- **Versatility**

  Supports C/assembly programs, code optimization, and architectural benchmarking.
- **Impact:**

  Empowers IT students and HPC developers to optimize code and design custom RISC-V processors.
- **Adoption**

  It is currently used by 200+ students in the Computation Systems Architectures course at Brno University of Technology.

Future Work

- **Processor Enhancements**

  Add advanced features like vector units, pipelined functional units, improved branch predictors, and deeper cache hierarchies.
- **Development Tools**

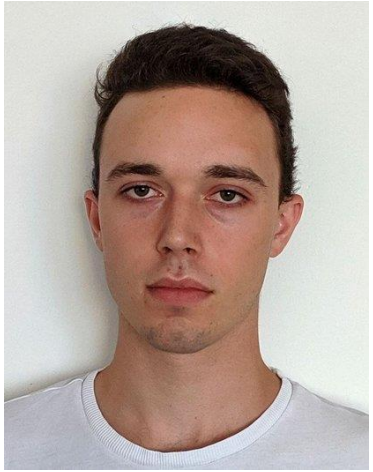
  Improve the code editor and debugging environment with features like breakpoints, watches, dynamic memory allocation, and atomic operations.
- **Expanded Metrics**

  Incorporate runtime statistics for chip area estimation and power consumption analysis using realistic manufacturing technology.
- **Community Engagement**

  Continue development and improvement via open-source contributions on GitHub and increase accessibility through the live simulator instance.

Acknowledgement

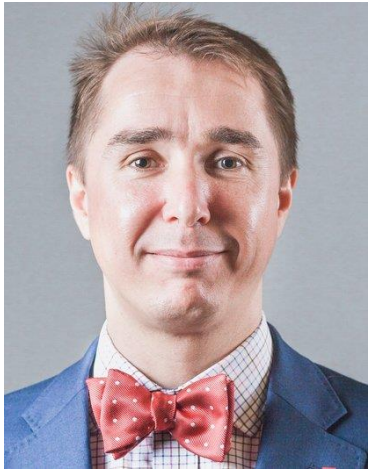
This simulator was brought to life by three talented master's students, guided by an enthusiastic supervisor.



Michal Majer

Jakub Horky

Jan Vavra



Jiri Jaros