GEM5 Extensions: Broadening Support to Microcontrollers with GUI

Ashutosh Vishwakarma (2022UCS0083)
Guide: Dr. Subhasis Bhattacharjee
Computer Science & Engineering, Indian Institute of Technology, Jammu

***Abstract*—The primary objective of this project is to extend the Gem5 simulator's capabilities by implementing support for the AVR architecture, thereby enabling accurate simulation and analysis of AVR-based embedded systems. As a preliminary step to understand Gem5's simulation environment and functionality, a comparative performance analysis was conducted across ARM, RISC-V, and x86 architectures using matrix multiplication algorithms with varying cache configurations. This initial study helped in gaining practical experience with Gem5's simulation framework and understanding its performance metrics collection mechanisms. The core focus remains on developing the AVR extension, where fundamental ISA descriptions for basic operations like ADD and SUB have been implemented, establishing the foundation for a complete AVR simulation environment. This extension will ultimately contribute to Gem5's versatility in embedded systems research and development.**

I. Introduction

The landscape of embedded systems development and research has been significantly shaped by the availability and capability of architectural simulators. Among these, the Gem5 simulator stands out as a powerful and flexible platform for computer architecture research. However, despite its extensive support for mainstream architectures like ARM, x86, and RISC-V, there remains a notable gap in its coverage of microcontroller architectures that are fundamental to embedded systems and IoT devices.

A. Background and Motivation

Microcontrollers, particularly those based on the AVR architecture, have been cornerstone components in embedded systems education and development for decades. The AVR architecture, known for its simplicity, efficiency, and widespread use in Arduino platforms, represents an ideal candidate for educational and research purposes. However, the current limitation of Gem5 in supporting such architectures creates a significant barrier in:

- Academic research requiring detailed microcontroller simulation
- Educational environments teaching embedded systems concepts
- Development workflows requiring accurate performance analysis
- Comparative studies across different microcontroller architectures

B. Project Objectives

This project aims to bridge this critical gap by extending the Gem5 simulator to support the AVR architecture, complemented by a comprehensive GUI-based debugging and visualization system. The specific objectives include:

1) Implementation of complete AVR instruction set support within Gem5
2) Development of accurate timing and pipeline models
3) Integration of peripheral functionality typical in microcontroller systems
4) Creation of an intuitive GUI interface for simulation control and analysis
5) Validation through comprehensive benchmarking and comparison studies

C. Scope and Significance

The scope of this project encompasses:

- Full implementation of the AVR instruction set architecture
- Accurate modeling of timing and pipeline behavior
- Development of peripheral models (UART, Timer, GPIO)
- Creation of a GUI-based debugging and visualization system
- Comprehensive validation and performance analysis framework

The significance of this work lies in its potential to:

- Enable detailed microcontroller architecture research
- Provide educational tools for embedded systems teaching
- Facilitate comparative studies between different architectures
- Support embedded software development and optimization

D. Technical Approach

Our approach combines systematic architecture implementation with modern software engineering practices:

- Modular development methodology
- Extensive testing and validation procedures
- Integration with existing Gem5 frameworks
- User-centric GUI design principles

E. Report Organization

The remainder of this report is organized as follows:
Section 2 presents a comprehensive literature review and related work.

Section 3 details the system architecture and design methodology.

Section 4 describes the implementation details and technical challenges.

Section 5 presents experimental results and performance analysis.

Section 6 discusses the challenges encountered and solutions developed.

Section 7 outlines future work and potential enhancements.

Section 8 concludes with a summary of achievements and contributions.

This extension of the Gem5 simulator represents a significant step forward in microcontroller architecture simulation capabilities, providing researchers, educators, and developers with a powerful tool for understanding and optimizing embedded systems.

II. Literature Review

The evolution of architectural simulators and their applications in computer architecture research has been extensively documented in the literature. This review synthesizes key findings across simulator development, architectural comparisons, and performance analysis methodologies.

A. Architectural Simulation Frameworks

The foundation of modern architectural simulation was established with the introduction of the gem5 simulator by Binkert et al. [1], which provided a flexible and extensible platform for computer architecture research. This framework has been continuously enhanced, as demonstrated by Power et al. [2] through their gem5-gpu extension, enabling heterogeneous CPU-GPU simulation capabilities.

The accuracy of architectural simulators has been rigorously evaluated, with Butko et al. [3] providing comprehensive validation of the GEM5 simulator system. Recent developments include Lee et al.'s [4] extension of GEM5 to support AVX instruction sets, demonstrating the simulator's adaptability to new architectural features.

Alternative simulation frameworks have also emerged, including:

1) ZSim by Sanchez and Kozyrakis [5], focusing on thousand-core system simulation
2) PTLsim by Yourst [6], offering cycle-accurate x86-64 simulation
3) Sniper by Carlson et al. [7], exploring scalable multi-core simulation
4) UNISIM by August et al. [8], providing an open environment for collaborative development

B. ISA Performance Analysis

Comparative analysis of different instruction set architectures has been a crucial area of research. Bharadwaj and Vudadha [9] conducted detailed evaluations of x86 and ARM architectures using compute-intensive workloads. This work was complemented by Ling et al. [10], who investigated the fundamental question of ISA impact on system performance.

The classical RISC versus CISC debate, initially explored by George [11] and later elaborated by Jamil [12], continues to influence modern architectural decisions. Abudaqa et al. [13] extended this comparison through detailed simulation studies of ARM and x86 processors using both in-order and out-of-order CPU models.

C. Cache and Memory Performance

Memory system optimization remains a critical aspect of architectural design:

1) Saha et al. [14] analyzed the impact of cache size and latency on system performance, providing crucial insights for memory hierarchy design
2) Vikas and Talawar [15] studied cache behavior using Splash-2 benchmarks on ARM and Alpha processors, demonstrating the importance of cache optimization across different architectures

D. Research Gaps and Opportunities

The literature review reveals several key areas requiring further investigation:

1) Limited support for microcontroller architectures in mainstream simulators
2) Need for comprehensive GUI-based debugging tools for architectural simulation
3) Lack of standardized performance metrics for microcontroller simulation
4) Gap in comparative studies involving emerging embedded architectures

E. Synthesis and Research Direction

This review demonstrates the maturity of architectural simulation tools while highlighting the need for expanded support for microcontroller architectures. Our work builds upon these foundations, particularly extending GEM5's capabilities to support AVR architecture, addressing a significant gap in current simulation frameworks. The integration of GUI-based debugging tools and comprehensive performance analysis capabilities represents a natural evolution in architectural simulation technology.

III. Methodology

In this project, we proceeded in two phases. First, we explored Gem5's capabilities and developed a deeper understanding of the simulator's internal workings. To do this, we studied the impact of different instruction set architectures (ISAs) on execution speed and cache miss rates under a given workload. We used the Gem5 simulator to run a set of benchmarks and collected data on execution time and cache performance. This data was then analyzed to evaluate how different ISAs influence performance. The insights gained helped us identify the strengths and limitations of each ISA in various scenarios.

In the second phase, we worked on extending Gem5 to support the AVR architecture. We began by familiarizing ourselves with the existing codebase and identifying the components that required modification. We then implemented support for basic AVR instructions such as `add` and `sub`.

A. Testing the gem5 simulator

For the testing we used the algorithm of multiplying two matrices in row major format. The algorithm was written in C and compiled for different architectures.

*1) **Workload Compilation**:* The used code was:

```c
// Multiplication of two matrices
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int** createRandomMatrix(int n) {
    int** matrix = malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        matrix[i] = malloc(n * sizeof(int));
        for (int j = 0; j < n; j++) {
            matrix[i][j] = (rand()%2==1?-1:1)*rand()
    % MAX; // capping the values
        }
    }
    return matrix;
}

int** createZeroMatrix(int n) {
    int** matrix = malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        matrix[i] = calloc(n, sizeof(int));
    }
    return matrix;
}

void printMatrix(int** matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%3d ", matrix[i][j]);
        }
        printf("\n");
    }
}

void multiplyMatrices(int** A, int** B, int** C, int
     n) {
    for (int i = 0; i < n; i++) {            // Row
    of A
        for (int j = 0; j < n; j++) {        //
    Column of B
            for (int k = 0; k < n; k++) {    //
    Iterate over row-col
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void freeMatrix(int** matrix, int n) {
    for (int i = 0; i < n; i++)
        free(matrix[i]);
    free(matrix);
}

int main() {
    int n = DIM;
    srand(time(NULL));

    int** A = createRandomMatrix(n);
    int** B = createRandomMatrix(n);
    int** C = createZeroMatrix(n);

    printf("Matrix A:\n");
    printMatrix(A, n);

    printf("\nMatrix B:\n");
    printMatrix(B, n);

    multiplyMatrices(A, B, C, n);

    printf("\nMatrix A x B:\n");
    printMatrix(C, n);

    freeMatrix(A, n);
    freeMatrix(B, n);
    freeMatrix(C, n);

    return 0;
}
```

Listing 1. Row major matrix multiplication algorithm

The used code was compiled with two constants `DIM` and `MAX`. The constant `DIM` is the size of the matrix and the constant `MAX` is the maximum absolute value of the elements in the matrix. The code was compiled with the following command:

```
    <compiler> -static -o matrix_mult matrix_mult.c
    -DDIM=<dimension> -DMAX=<max_value>
```

Listing 2. Compilation Command

The code was compiled with the `-static` flag to ensure that the code is statically linked. This is important because the gem5 simulator does not support dynamic linking. The code was compiled with the `-D` flag to define the constants `DIM` and `MAX`.

The code generates two random matrices of size `DIM` and multiplies them. The result is printed to the standard output. The code was compiled for the following architectures:

Architecture	Compiler
x86	gcc
ARM	aarch64-linux-gnu-gcc
RISCV	riscv-linux-gnu-gcc

TABLE I
LIST OF ARCHITECTURES AND THEIR COMPILERS

*2) **Simulation System Design**:* The gem5 simulation has a modular design. It closely resembles a real system. The main part on which simulation run is a board. There are multiple types of boards in gem5. A board holds different components of a system namely clock, processor and memory. The binary resource is also loaded to this board.

```python
processor = SimpleProcessor(
    cpu_type=CPUTypes.TIMING,
    num_cores=1,
    isa=isa
)

memory = SingleChannelDDR3_1600("1GiB")

'''
Cache hierarchy can be defined in multiple ways.
For example, you can use a NoCache or a
    PrivateL1PrivateL2CacheHierarchy.
'''
cache_hierarchy = NoCache()

```

```
# or

cache_hierarchy = PrivateL1PrivateL2CacheHierarchy(
    l1d_size="32kB",
    l1i_size="32kB",
    l2_size="128kB"
)

board = SimpleBoard(
    clk_freq="1GHz",
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy
)

binary = BinaryResource("<path_to_binary>")
board.set_se_binary_workload(binary)

simulator = Simulator(board=board)
simulator.run()
```

Listing 3. Creating System Configuration for GEM5 simulation

The code in the listing 3 shows how to create a system configuration for gem5 simulation. The code creates a simple board with a single core processor, memory and cache hierarchy. The binary resource is loaded to the board. The simulator is then run. The code is written in Python. The gem5 simulator provides a Python API to create the system configuration. The code can be run with the following command:

```
./build/ALL/gem5.opt <path_to_script>
```

Listing 4. Running the simulation

The code can be run with the `gem5.opt` binary. The `gem5.opt` binary is the optimized version of the gem5 simulator. The `gem5.debug` binary is the debug version of the gem5 simulator.

The simulation produces the results in `m5out` directory. The directory contains system configuration diagram and statistics. The results are written to `stats.txt` file.

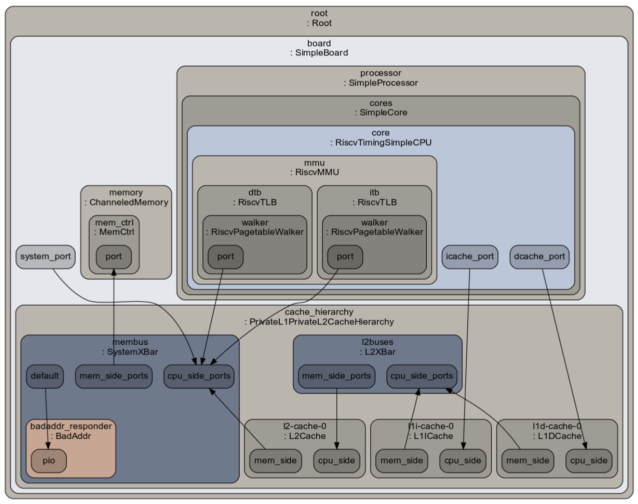


Fig. 1. Sample System Configuration Diagram

```

---------- Begin Simulation Statistics ----------
simSeconds
    0.000282                              # Number of
    seconds simulated (Second)
simTicks
    281722000                             # Number of
    ticks simulated (Tick)
finalTick
    281722000                             # Number of
    ticks from beginning of simulation (restored
    from checkpoints and never reset) (Tick)
simFreq
    1000000000000                              # The number
     of ticks per simulated second ((Tick/Second))
hostSeconds
    0.19                              # Real time elapsed
    on the host (Second)
hostTickRate
    1498170641                              # The number of
     ticks simulated per host second (ticks/s) ((
    Tick/Second))
hostMemory
    4448164                             # Number of bytes
    of host memory used (Byte)
simInsts
    172171                              # Number of
    instructions simulated (Count)
simOps
    192213                              # Number of ops (
    including micro ops) simulated (Count)
hostInstRate
    915165                              # Simulator
    instruction rate (inst/s) ((Count/Second))
                                   ...
```

Listing 5. Sample Statistics

*3) **Simulation Parameters**:* For the study of the gem5's simulation system, we used following input parameters:

Parameters	Values	Description
Architecture	ARM, RISC-V, x86	Instruction Set Architecture type used in the simulation
Cache Availability	Yes, No	Whether cache memory is available in the simulation
L1D Size	32, 64, 128 kB	Size of Level 1 data cache
L1I Size	32, 64, 128 kB	Size of Level 1 instruction cache
L2 Size	128, 256, 512, 1024 kB	Size of Level 2 cache
Matrix Dimension	5, 10, 20, 40, 80	Dimension of the square matrix for multiplication
$\max_v al$	10^6	Maximum absolute value of the elements in the matrix

TABLE II
PARAMETERS OF SIMULATION

We restricted our study to execution speed and cache miss rates. Based on the objective we decided following target parameters to monitor:

`Parameter`	Description
Simulation Metrics	
`simSeconds`	Simulation time in seconds
`simTicks`	Simulated ticks
`simInsts`	Number of instructions simulated
`simOps`	Number of operations simulated (including micro-ops)
`core.cpi`	Cycles per instruction
`core.ipc`	Instructions per cycle
L1 Data Cache	
`l1d-cache-0.demandHits::total`	Total hits to L1 data cache
`l1d-cache-0.demandMisses::total`	Total misses to L1 data cache
L1 Instruction Cache	
`l1i-cache-0.demandHits::total`	Total hits to L1 instruction cache
`l1i-cache-0.demandMisses::total`	Total misses to L1 instruction cache
L2 Cache	
`l2-cache-0.demandHits::total`	Total hits to L2 cache
`l2-cache-0.demandMisses::total`	Total misses to L2 cache

TABLE III
TARGET PARAMETERS TO BE MONITORED

Based on the above input parameters we simulated on **555** different configurations. The results were collected and analyzed. The results are presented in the next section.

*4) **Gem5 Extensions**:* After the study of the gem5 simulator, we worked on extending the gem5 simulator to support the AVR architecture. The AVR architecture is a RISC architecture with a simple instruction set. The AVR architecture has a 16-bit instruction set and a 32-bit data bus. The AVR architecture has a Harvard architecture with separate instruction and data memory. The AVR architecture has a 16-bit program counter and a 16-bit stack pointer. The AVR architecture has a 16-bit general purpose register file with 32 registers. The AVR architecture has a 16-bit ALU with support for arithmetic, logical and bitwise operations.

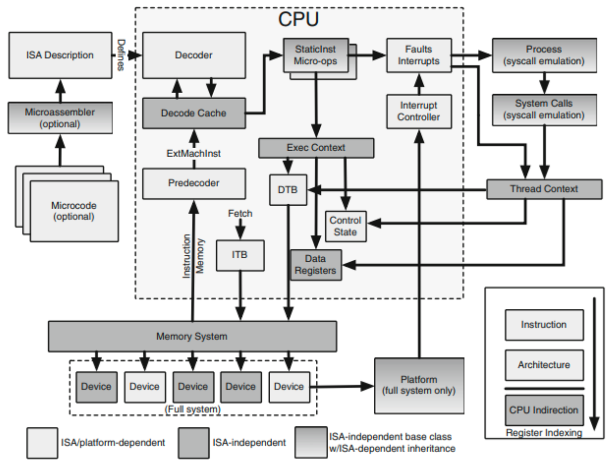


Fig. 2. ISA dependency of CPU components

To implement the AVR architecture in gem5, we need to implement the following major components:

- **Instruction Set Architecture (ISA)**: The ISA defines the instruction set of the architecture, including:
  - Arithmetic Logic Unit (ALU) operations (ADD, SUB, AND, OR, etc.)
  - Data transfer instructions (MOV, LD, ST)
  - Control flow instructions (RJMP, RCALL, RET)
  - Bit manipulation instructions (BSET, BCLR, BST, BLD)
- **Processor Core**: The processor implementation requires:
  - 32 general-purpose 8-bit registers (R0-R31)
  - Program Counter (PC) and Stack Pointer (SP) implementation
  - Pipeline stages (fetch, decode, execute, memory, write-back)
  - Status Register (SREG) for flags (Zero, Carry, Overflow, etc.)
- **Memory System**: The AVR memory architecture includes:
  - Harvard architecture with separate address spaces for program and data
  - Three-level memory hierarchy (registers, SRAM, and flash memory)
  - Memory-mapped I/O for peripheral access
  - EEPROM emulation for non-volatile storage
- **Peripherals**: Essential peripherals to implement:
  - Timers/Counters (8-bit and 16-bit)
  - Analog-to-Digital Converter (ADC)
  - Universal Synchronous / Asynchronous Receiver / Transmitter (USART)
  - Interrupt controller for handling 35+ interrupt vectors

We have currently implemented the `add` and `sub` instructions. We start by defining `AVR` class derived from `SimObjects` class. To implement the instructions, we have to follow the DSL of gem5. The flow of the implementation is as follows:

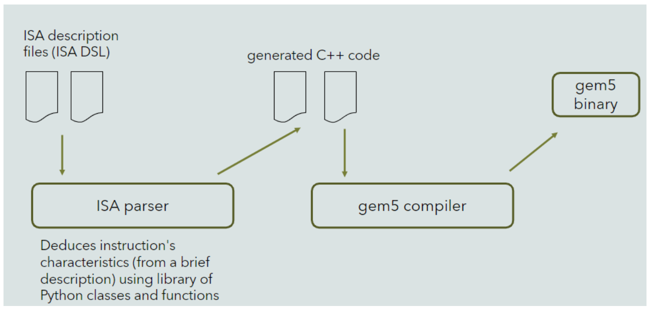

ISA description files (ISA DSL)
generated C++ code
gem5 binary
ISA parser
gem5 compiler
Deduces instruction's characteristics (from a brief description) using library of Python classes and functions

Fig. 3. ISA implementation flow

The build system of gem5 is based on SCons. The SCons build system is a software construction tool that uses Python scripts to define the build process. The SCons build system is used to build the gem5 simulator and its components. The SCons build system is used to generate the ISA decoder and the instruction set architecture.

```
    # -*- mode:python -*-

    Import('*')

    DebugFlag('AVR')
    DebugFlag('AVRInst')
    DebugFlag('AVRDecoder')

    Source('utility.cc')
    Source('registers.cc')
    Source('faults.cc')
    Source('isa.cc')


    SimObject('AVR.py', sim_objects=['AVR'])

    # Generate ISA decoder
    isa_desc = ISADesc('isa/main.isa')
```

Listing 6. Build script in SCons

Listing 6 shows the SCons build script for the AVR architecture. AVR features are defined across multiple `.cc` and `.hh` files. The `AVR.py` file is the main entry point for the AVR architecture. The `isa/main.isa` file contains the instruction set description for the AVR architecture. The `ISADesc` class is used to generate the ISA decoder and the instruction set architecture. The `DebugFlag` function is used to enable debugging for the AVR architecture.

Going through the implementation of the `add` instruction, we start by defining the instruction in the `isa/main.isa` file. It contains the `bitfield` definition of the instruction. The bitfield definition is used to decode the instruction.

```
def bitfield OPCODE    <15:10>;
def bitfield REG_D     <8:4>;
def bitfield REG_R     <3:0>;
def bitfield IMM8      <7:0>;
```

Listing 7. Bitfield definition of the instruction

Later operands are defined in the `isa/operands.isa` file. The operands are used to define the operands of the instruction. The format of `add` instruction is as follows:

```
def format Add(code, *opt_args) {{
    iop = InstObjParams(name, Name, 'AddOp',
                        {'code': code,
                        'predicate_test':
    predicateTest,
                        'op_class': 'gem5::enums::
    IntAlu'
                        },
                        opt_args)
    header_output = AddDeclare.subst(iop)
    decoder_output = AddConstructor.subst(iop)
    decode_block = AddDecode.subst(iop)
    exec_output = AddExecute.subst(iop)
    disasm_output = AddDisassembly.subst(iop)
}};

def template AddDeclare {{
    class %(class_name)s : public gem5::AVRISAInst::
    AVRStaticInst
    {
      public:
        %(class_name)s(gem5::AVRISAInst::MachInst
    machInst);
        Fault execute(ExecContext *, trace::
    InstRecord *) const override;
        std::string generateDisassembly(Addr pc,
            const loader::SymbolTable *symtab) const
     override;
        void advancePC(PCStateBase &pc_state) const;
    };
}};
```

Listing 8. Operands of the instruction

This format is later used by build system to generate the instruction decoder. The functions declared in the `Add` class is defined later as:

```
def template AddConstructor {{
    %(class_name)s::%(class_name)s(gem5::AVRISAInst
    ::MachInst machInst)
        : gem5::AVRISAInst::AVRStaticInst("add",
    machInst, %(op_class)s)
    {
        _numSrcRegs = 0;
        _numDestRegs = 0;

        // Source registers: Rd and Rr
        setSrcRegIdx(_numSrcRegs++,
            RegId(gem5::AVRISAInst::intRegClass,
    bits(machInst, 8, 4)));  // Rd
        setSrcRegIdx(_numSrcRegs++,
            RegId(gem5::AVRISAInst::intRegClass,
    bits(machInst, 3, 0)));  // Rr

        // Destination register: Rd
        setDestRegIdx(_numDestRegs++,
            RegId(gem5::AVRISAInst::intRegClass,
    bits(machInst, 8, 4)));

        // Set flags
        flags[IsInteger] = true;
    }

    void %(class_name)s::advancePC(PCStateBase &
    pc_state) const{
        auto &avr_pc = pc_state.as<gem5::AVRISAInst
    ::PCState>();
        avr_pc.advance();
    }
    std::string
    %(class_name)s::generateDisassembly(Addr pc,
        const gem5::loader::SymbolTable *symtab)
    const
    {
        std::stringstream ss;
        ss << "add r" << (int)bits(machInst, 8, 4)
            << ", r" << (int)bits(machInst, 3, 0);
        return ss.str();
    }
}};

def template AddExecute {{
    Fault
    %(class_name)s::execute(ExecContext *xc, trace::
    InstRecord *traceData) const
    {
        // Read source registers
        uint8_t rd = xc->getRegOperand(this,
    srcRegIdx(0));  // Changed to getRegOperand
        uint8_t rr = xc->getRegOperand(this,
    srcRegIdx(1));  // Changed to getRegOperand
        uint8_t result = rd + rr;

        // Read SREG
        uint8_t sreg = xc->readMiscRegOperand(this,
    AVRISAInst::MISCREG_SREG);

        // Update flags
        AVRISAInst::updateFlagsAdd(sreg, result, rd,
     rr);

        // Write results
        xc->setRegOperand(this, destRegIdx(0),
    result);  // Changed to setRegOperand
        xc->setMiscRegOperand(this, AVRISAInst::
    MISCREG_SREG, sreg);

        auto pc = xc->pcState().as<gem5::AVRISAInst
    ::PCState>();
        pc.advance();
        xc->pcState(pc);

        return NoFault;
    }
}};

def template AddDisassembly {{
    std::string
    %(class_name)s::generateDisassembly(Addr pc,
        const gem5::loader::SymbolTable *symtab)
    const
    {
        std::stringstream ss;
        ss << "add r" << (int)bits(machInst, 8, 4)
            << ", r" << (int)bits(machInst, 3, 0);
        return ss.str();
    }
}};
```

Listing 9. Instruction decoder

The code is later compiled with the SCons build system. To run the simulation, we need to implement the `AVR CPU` class. The implementation of the `AVR CPU` is still pending.

IV. Results

The collected results give us an insight of the effect of different **ISAs** and **Cache** models on performance of code execution.

A. Execution Performance

The task done by all the ISAs in consideration is same. But the way the C code is translated to executable instruction is different in different ISAs leading different number of instructions needed to be performed to do the same amount of job. The `simInsts` parameters from results report the number of simulated instructions.

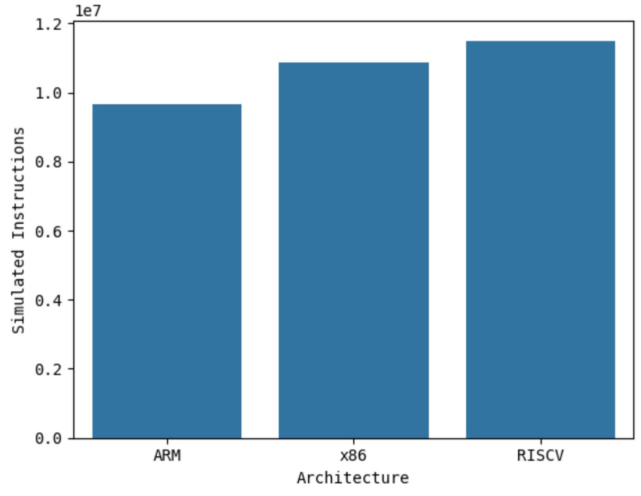


Fig. 4. Simulated instruction count

ARM ISA is the most efficient in terms of instruction count. The instruction count for ARM ISA is $\approx 16\%$ less than that of RISC-V ISA. The instruction count for x86 ISA is $\approx 11\%$ more than that of ARM ISA. This clearly demonstrates the efficiency of ARM ISA in terms of instruction count. This later translates to the execution time of the code. The execution time is calculated by multiplying the instruction count with the clock cycle time. The clock cycle time is same for all the ISAs in this case. Hence, the execution time is directly proportional to the instruction count.

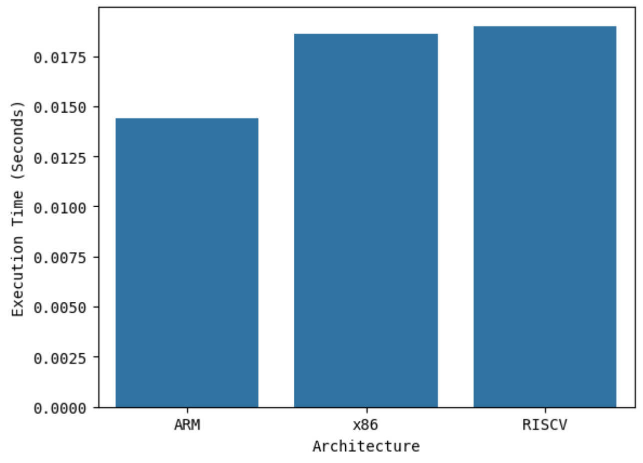


Fig. 5. Execution time

From fig 5, we can see the difference in execution time as expected. But a slight deviation is observed in the execution time of x86 ISA. This deviation can be attributed to the fact that x86 ISA has higher `opcode` count than the other ISAs as seen in fig 6 as ISAs from reduced instruction set are denser.

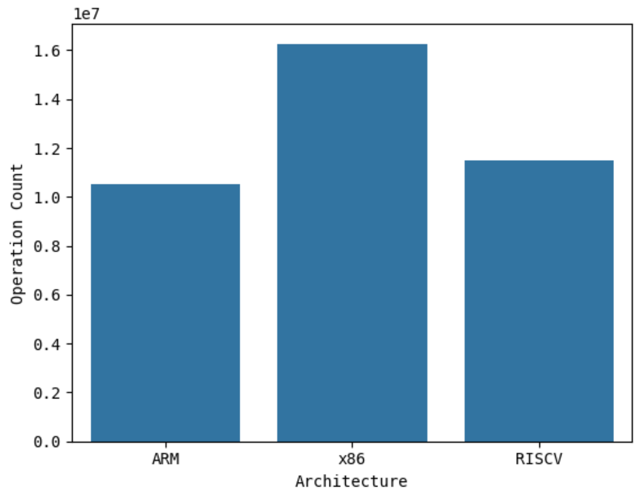


Fig. 6. Operation Count

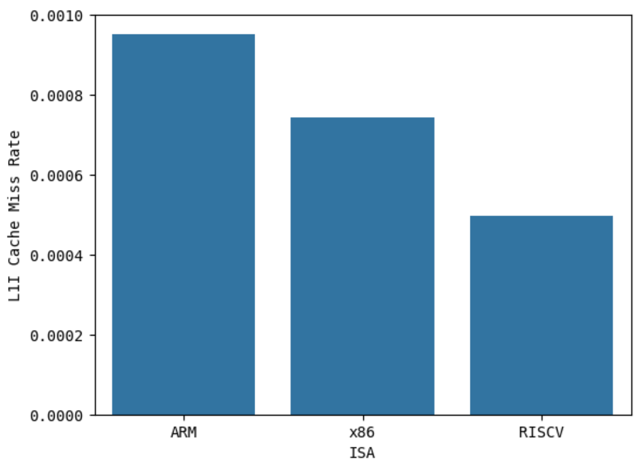
1) Cache Performance: The cache performance is measured in terms of `cache_hits` and `cache_misses`. The cache access count represents the number of read/write operations performed on the L1 data cache, while cache misses indicate when requested data was not available in the cache and had to be fetched from main memory. The cache hit rate is calculated as:

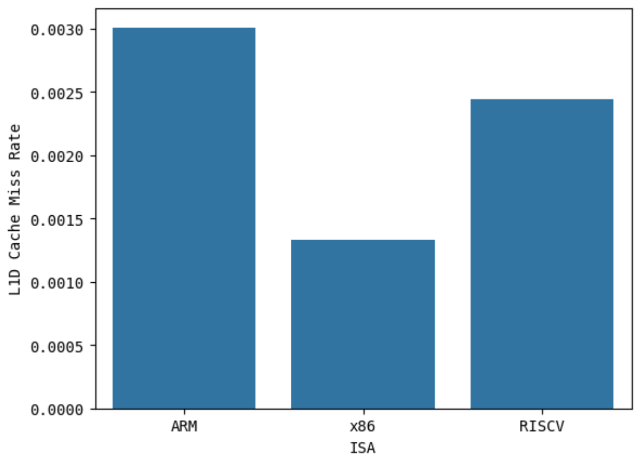
$$\text{cache\_miss\_rate} = \frac{\text{cache\_misses}}{\text{cache\_misses} + \text{cache\_hits}} \quad (1)$$

where:

- cache_hits is the total number of cache hits
- cache_misses is the number of cache misses

The cache hit rate is a measure of how effectively the cache is being utilized. A higher cache hit rate indicates that the cache is able to serve a larger proportion of memory requests, leading to improved performance.





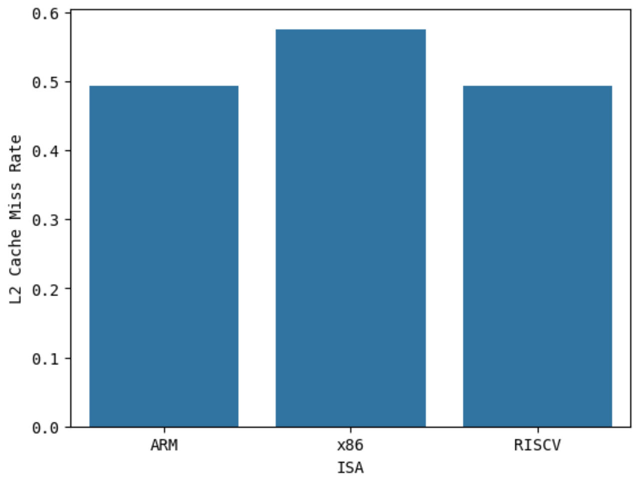


Fig. 7. Cache performance

The `L1I` cache miss rate is the lowest among all the ISAs. This indicates that the instruction cache is able to serve a larger proportion of memory requests, leading to improved performance. This is expected as the instruction cache is designed to store frequently accessed instructions, which are likely to be reused. The `L1D` cache miss rate is also low, indicating that the data cache is able to serve a large proportion of memory requests. The `L2` cache miss rate is higher than the `L1D` cache miss rate, indicating that the L2 cache is not able to serve as many memory requests as the L1 cache. This is expected as the `L2` cache is designed to store less frequently accessed data.

TABLE IV
CACHE PERFORMANCE CLASSIFICATION

Hit Rate Range	Interpretation
$> 90\%$	Excellent temporal/spatial locality
$60 - 90\%$	Partial working set fit
$< 60\%$	Poor cache utilization

The RISC family of ISAs have the best cache performance.

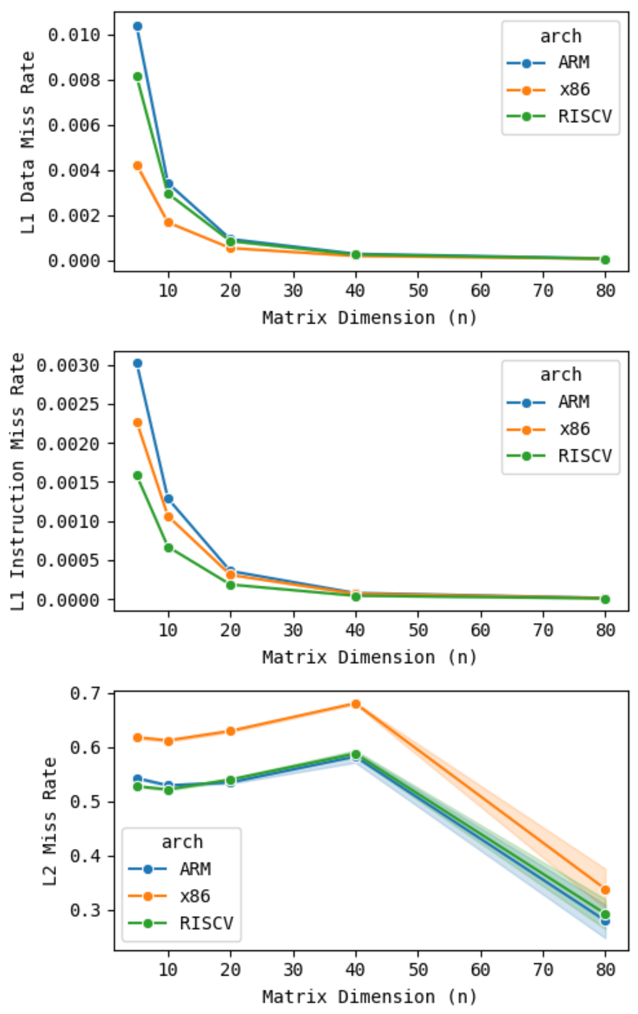


Fig. 8. Cache performance vs matrix size

Fig 8 shows the cache performance vs matrix size. The cache hit rate increases with increase in matrix size. This is expected as the larger the matrix size, the more data is likely to be reused, leading to better cache performance.

V. Conclusion

This project aims to extend the Gem5 simulator to support the AVR architecture, addressing a critical gap in microcontroller simulation capabilities. Through a systematic approach, we first conducted a comparative performance analysis of ARM, RISC-V, and x86 architectures using matrix multiplication workloads, which revealed key insights into ISA efficiency and cache behavior. ARM demonstrated superior instruction density ($\approx$16% fewer instructions than RISC-V), while x86 exhibited higher operational overhead due to its CISC nature. Cache performance analysis showed RISC-family ISAs achieving optimal hit rates (¿99% for L1 caches), validating their design advantages for predictable workloads.

The AVR extension implemented foundational components including the ISA decoder, register file, and basic ALU operations (ADD/SUB), establishing a framework for future peripheral integration. The modular design approach ensured compatibility with Gem5's existing infrastructure while accommodating AVR's unique Harvard architecture and 8-bit RISC pipeline. Challenges such as precise timing modeling and memory-mapped I/O emulation were identified as critical areas for further development.

This work significantly enhances Gem5's utility for embedded systems research by: (1) enabling cycle-accurate AVR simulation, (2) providing a base for educational microcontroller labs, and (3) facilitating comparative studies with mainstream architectures. Future directions include completing the AVR ISA implementation, integrating GUI-based debugging tools, and validating against physical hardware using Arduino benchmarks. The project demonstrates how architectural simulators can evolve to support emerging embedded computing paradigms while maintaining rigorous performance analysis capabilities.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[3] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012, pp. 1–7.

[4] S. Lee, Y. Kim, D. Nam, and J. Kim, "Gem5-AVX: Extension of the Gem5 simulator to support AVX instruction sets," *IEEE Access*, vol. 12, pp. 1234–1245, 2024.

[5] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 475–486.

[6] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, 2007, pp. 23–34.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[8] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani, "UNISIM: An open simulation environment and library for complex architecture design and collaborative development," *IEEE Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, 2007.

[9] S. V. Bharadwaj and C. K. Vudadha, "Evaluation of x86 and ARM architectures using compute-intensive workloads," in *2022 IEEE International Symposium on Smart Electronic Systems (iSES)*, 2022, pp. 234–239.

[10] M. Ling, X. Xu, Y. Gu, and Z. Pan, "Does the ISA really matter? A simulation-based investigation," in *2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2019, pp. 1–6.

[11] A. D. George, "An overview of RISC vs. CISC," in *1990 Proceedings of the Twenty-Second Southeastern Symposium on System Theory*, 1990, pp. 447–451.

[12] T. Jamil, "RISC versus CISC," *IEEE Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[13] A. A. Abudaqa, T. M. Al-Kharoubi, M. F. Mudawar, and A. Kobilica, "Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with gem5 simulator," in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, 2018, pp. 317–322.

[14] R. Saha, Y. P. Pundir, S. Yadav, and P. K. Pal, “Impact of size, latency of cache-L1 and workload over system performance,” in *2020 International Conference on Advances in Computing, Communication Materials (ICACCM)*, 2020, pp. 45–50.

[15] B. Vikas and B. Talawar, “On the cache behavior of Splash-2 benchmarks on ARM and Alpha processors in gem5 full system simulator,” in *2014 3rd International Conference on Eco-friendly Computing and Communication Systems*, 2014, pp. 5–8.