ELEC2204: MIPS Simulator

George Rennie
gr1g20@soton.ac.uk
Personal Tutor: Professor Neil White

Abstract: This report details a cycle accurate simulator for a subset of the MIPS instruction set architecture written in C11/C++20. The simulation is of a processor with an in-order, scalar, 5-stage pipeline as described by Hennessy and Patterson[1], including appropriate forwarding and stalling logic. The simulator was formally verified against a single cycle reference model using techniques derived from the ISA-Formal[2] methodology developed at Arm. Additionally, an assembler was written for the instruction set, allowing the simulator to execute a set of test programs.

1. Design

The processor detailed in this report is a 5-stage pipelined MIPS processor, implementing a subset of standard MIPS instructions listed in table I. A diagram of the design is attached at the end of this report. A range of different instruction types were chosen to demonstrate different types of execution and all stages of the pipeline. The processor was taken to be big endian, and all memory accesses explicitly declared as big endian in the source code so that the code is target indepedent.

Where possible, forwarding was favoured over stalling, and results from both the memory and writeback stages can be forwarded to the execute stage. Branch evaluation occurs early, in the instruction decode stage, so results are also forwarded to the branch comparison logic for this purpose. However, if a branch instruction relies on a register that will be written to by the instruction in the execution phase, it needs to stall one cycle for the result to be calculated.

When branches or jumps are taken, normally the pipeline is flushed to prevent the following instructions from executing. The processor also supports being configured with delay slots enabled, where the instruction after branches and jumps executes unconditionally, allowing a greater throughput.

Instruction Name	Short	Fmt	Operation	Opcode	Funct
Add Imm. Unsigned	addiu	I	`R[rt] = R[rs] + SignExtImm`	0x09	
Add Unsigned	addu	R	`R[rd] = R[rs] + R[rt]`	0x00	0x021
And	and	R	`R[rd] = R[rs] & R[rt]`	0x00	0x24
And Immediate	andi	I	`R[rt] = R[rs] & ZeroExtImm`	0x0C	
Or	or	R	`R[rd] = R[rs] | R[rt]`	0x00	0x25
Or Immediate	ori	I	`R[rt] = R[rs] | ZeroExtImm`	0x0D	
Load Halfword Unsigned	lhu	I	`R[rt] = M[R[rs] + SignExtImm](15:0)`	0x25	
Store Halfword	sh	I	`M[R[rs] + SignExtImm] = R[rt](15:0)`	0x29	
Branch On Equal	beq	I	`if(R[rs]==R[rt])PC=PC+4+BranchAddr`	0x04	
Branch On Not Equal	bne	I	`if(R[rs]!=R[rt])PC=PC+4+BranchAddr`	0x04	
Jump	j	J	`PC = JumpAddr`	0x02	
Breakpoint	break	R	`raise BREAK exception`	0x00	0x0D

Table I: Implemented instruction set (adapted from [1])

Name	Start Address	Description
.data	0x00000000	Data memory, adjustable in size
.text	0x00003000	Instruction memory, scales in size to input file

Table II: Memory Map (based on MARS[3] compact layout)

The simulator only implements a basic exception model, supporting the exceptions listed in table III. When an exception is encountered, the pipeline is flushed, and execution continues until the excepting instruction retires. At this point execution ends and the user is presented with the cause of the exception.

Exception Name	Short	Cause Value	Triggers
Address Load	ADDRL	4	Unaligned/Page Faulting Memory Read
Address Store	ADDRS	5	Unaligned/Page Faulting Memory Write
Breakpoint	BREAK	9	Breakpoint Instruction
Reserved Instruction	RI	10	Unrecognised Opcode/Funct

Table III: Implemented exceptions

2. Functionality and Implementation

To allow the use of formal verification tools which have limited C++ support, the core of the simulator was written in C11. Despite this, the user interface and assembler were written in C++20 as they were not tested in this manner. The code style was intended to somewhat mimic a Hardware Description Language, describing a state transition rather than a more traditional program.

2.1. Utility Functions

A library of common functions was developed in C to aid efficient development. The main features of this, were logging functions, and assertions. A range of printf style logging functions for general messages, errors, and debug messages were provided, and macros were used to allow the debug messages to only be compiled into the binary when debugging. For assertions, a chain of macros was used to provide assertions with smart printing of values. For example, in line 3 of listing 1, if the assertion fails, the macro prints out the values in the failing assertion as strings, by calling the function `mem_access_to_str`. This greatly aided debugging by making it clear why assertions failed. Because these assertions were based on macros, they also included line numbers and could be redefined to call different assertion functions in the case of failure. For example, in the unit testing testbench assertion failures are caught and control flow returned to the test handler, whereas in the main simulator build, assertion failures cause the program to exit.

Listing 1: Example usage of assertion macros

```
1 log_assert_eqi(id_ex.eval_branch, false);
2 log_assert_eqi(id_ex.branch, false);
3 log_assert_cond_maps(id_ex.ex_mem.access_type, ==, MEM_ACCESS_NONE, mem_access_to_str);
```

Also implemented was utility functions for extracting and inserting ranges of bits into an integer, which were very helpful for extracting bits for instruction decoding.

2.2. Processor Core

The processor core was split into distinct logical units for ease of development. The pipeline registers were declared together, and then a function for each pipeline stage was declared, each one returning the next state for their output register. Additionally, hazard detection and pipeline forwarding were broken out into their own functions. This made it easy to integrate the functions together into the main function `mips_core_cycle`, which advances the processor by one cycle.

2.2.1. Instruction Fetch

Instruction fetch is fairly straightforward, simply reading the instruction pointed to by the program counter into the IF/ID register. In the case that the program counter points to an address output of the instruction memory, an address load exception is raised, with the bad address marker set to the program counter

2.2.2. Instruction Decode

The instruction decode stage splits the instruction into a range of different operands to pass through the pipeline. This is also where the registers are read, and branching occurs. This is implemented with a number of case statements to consider the different combinations of opcodes and funct values. This can throw breakpoint and reserved instruction exceptions. It then sets up the ID/EX register for execution in the next cycle.

2.2.3. Execute and Forwarding

The execute stage performs the actual calculations. As part of this, it calls the forwarding unit to determine if any of its operands need to be forwarded from later stages. Values in the memory stage are forwarded over values in the writeback stage as they occur later in the program order. Additionally, if the register in question is register 0, no forwarding occurs.

2.2.4. Memory

The memory function simply reads the memory access type setup in the instruction decode phase, and if it is either a write or a read, does the corresponding memory operation. If the memory is read, the value set to be written back in the writeback stage is overwritten by this new value. Accesses to unaligned or unallocated memory result in exceptions.

2.2.5. Writeback

Writeback is the smallest stage in the simulator, simply writing back the result from either the execute or memory stages to the designated register in the register file. Additionally in this stage, metadata about the retiring instruction is gathered, such as what number instruction it was, and what cycle has just been executed. This can then be reported to the user of the simulator.

2.2.6. Hazard detection

The hazard detection function acts on both the current state of the processor and the newly calcualted state calculated by the pipeline stages. If it detects a hazard, it can stall or flush pipeline stages by raising a flag, which causes the calculated values from that stage not to be written through to the next register in the pipeline. It can detect a range of hazards including load/store hazards, branch hazards, control hazards, and also flushes the pipeline when exceptions are raised.

2.3. Reference Core

A small (less than 200 lines of code) single cycle reference core was implemented to be used for verification. As this was single cycle, there was no issues of stalling or forwarding, and so the potential errors in its design were decreased, therefore making it a good candidate for equivalence checking with the main simulator.

2.4. Assembler

An assembler was written in C++ to be able to parse all the instructions supported by the simulator, as well as labels for jump and branch instructions. The assembler automatically inserts a breakpoint after the last instruction to catch the end of the program. It also prints helpful error messages including line numbers if it is unable to parse the input file to help the user find where they have gone wrong.

2.5. Command Line Interface

A rich command line interface was developed to make interacting with and debugging the simulator easier. This used an open source argument parsing library [4] to parse the command line arguments. A full list of command line arguments can be found in appendix A. The command line interface allows loading either assembly or a binary and executing it on either the simulator, the reference core or both at once, comparing their outputs for each instruction that retires.

2.5.1. Single Step Mode

To gain insight to the execution of the simulator, a single step mode was implemented where the user can step through a program cycle by cycle, viewing the internal registers. This works in a standard terminal and just requires the user to press enter to step through the program. An example capture of this is shown in appendix C.

3. Testing

A range of different testbenches were used in the verification of the simulator, in order to get a good range of coverage, as well as to support the bringup of the design. This project seemed a good opportunity to learn more about the use of formal verification of C code with a bounded model checker, and also the techniques used for formal verification of commercial processors, so ESBMC[5], a modern bounded model checker based on clang, was used for a number of the testbenches. A number of embedded assertions were written inside the simulator core, documenting its invariants, and these assertions were used by all of the other forms of testing by defining macros to call appropriate assertion handlers depending on the environment.

Name	Time	Brief Description
`unit`	<1s	Unit level directed tests of the pipeline stages and regression traces from `esbmc_equiv`
`esbmc_unit`	∼5s	Unit level formal tests of the pipeline stages, which provides greater coverage than `unit`
`asm_test`	<1s	Integration tests written in assembly. Compared against a reference core
`esbmc_check`	∼8m	12 cycle formal check of embedded assertions and C safety properties
`esbmc_isa`	∼35m	Formal check of correctness of register operations including forwarding logic
`esbmc_equiv`	∼4h	Formal check that processor operation matches reference core over 7 cycles

Table IV: Testbenches employed in verification of the simulator

3.1. Unit tests (`unit`)

Unit testing was done with a small custom C unit testing framework, and was aimed at verifying the correctness of individual stages of the pipeline in isolation. To do this, helper functions focusing on testing individual functions of each pipeline stage (e.g. R type and J type instructions were tested separately for instruction decode) were written. A large number of assertions were used in these functions to check the expected invariants. Test scenarios were then created by calling these test functions with values for the input to the pipeline stage and expected values for the output. An example set of tests used to test the operation of the execute stage is shown in listing 2. 45 tests of this variety were written, and they were particularly useful for initial implementation of the design where they were faster and easier to debug than the formal testbenches. These tests were also run with clang MemorySanitizer enabled to catch uses of uninitialised variables, a common issue when writing C.

Listing 2: Unit tests for execute stage

```
// data_rs, data_rt, alu operation, expected result
DEFINE_TEST(ex_reg_nop) { ex_test_reg_no_fwd(14, 15, ALU_OP_NOP, 14); }
DEFINE_TEST(ex_reg_add) { ex_test_reg_no_fwd(14, 15, ALU_OP_ADD, 29); }
DEFINE_TEST(ex_reg_and) { ex_test_reg_no_fwd(0x0FFFFF00, 0x00FFFFF0, ALU_OP_AND, 0x00FFFF00); }
DEFINE_TEST(ex_reg_or) { ex_test_reg_no_fwd(0x0FFFFF00, 0x00FFFFF0, ALU_OP_OR, 0x0FFFFFF0); }
```

3.2. Formal unit tests (`esbmc_unit`)

As the unit test helper functions had been written to be generalised, it was relatively simple to run them within a formal environment where their input parameters were left unconstrained, reusing the assertions from the directed unit tests. This allowed a proof of the tested properties to be generated for all sets of inputs. For example, instead of testing that the execute stage adds 14 and 15 correctly, this allows a proof that all operations it performs are correct for all inputs. Listing 3 shows how introduction of indeterminate values and assumptions was used to construct these testbenches.

Listing 3: Part of a formal unit test for execute stage

```
static void esbmc_ex_test() {
    const uint32_t data_rs = nondet_u32();
    const uint32_t data_rt = nondet_u32();
    const alu_op_t op      = (alu_op_t) nondet_u32();
    esbmc_assume(op >= 0 && op < ALU_OP_MAX);
    // ...
    ex_test_reg_no_fwd(data_rs, data_rt, op, res);
}
```

3.3. Assembly integration tests (`asm_test`)

To test the overall integration of the pipeline stages, a set of 9 tests were written in assembly specifically targeting common hazard scenarios. These were run using the compare mode of the simulator, where after each instruction retires, it is run on a single cycle reference model, and the resulting register, memory and exception states are compared. The output of these tests was also manually verified to confirm the correct operation of the processor. Listing 4 shows an example test, checking that instructions after a jump are flushed from the pipeline and not executed. The showcase programs were also used as assembly tests.

Listing 4: An assembly test for a jump hazard

```
main:
    addiu $t0, $zero, 1
    j exit
    addiu $t1, $zero, 1
    addiu $t2, $zero, 1
exit:
```

3.4. Embedded assertions and undefined behaviour (`esbmc_check`)

ESBMC is capable of checking for violations of a range of undefined behaviour in C, for example out of bounds accesses and misaligned pointers. The `esbmc_check` testbench runs the simulator for 12 clock cycles, checking all of these properties, as well as the embedded assertions within the simulator. This testbench generally caught subtle bugs due to easy to miss behaviours of C such as implicit casts causing bounds checks to fail.

3.5. Formal verification of forwarding (`esbmc_isa`)

Inspired by Alastair Reid's work on ISA-Formal[2] at Arm, and Claire Wolf's similar work on RISC-V Formal[6], `esbmc_isa` tries to prove correct operation of instructions from a specification. The testbench works by running the simulator from reset for 8 cycles to fill the pipeline with arbitrary instructions, assuming that none of them throw an exception. The state of the general purpose registers is saved, and the processor cycled once more, with an assumption that the instruction in the writeback stage retires or throws an exception. The single cycle reference model is then initialised to the state that was saved before this instruction wrote back to the register file and is also run on that instruction. The resulting register states of the two cores is then compared.

This relies on the fact that for the classic RISC pipeline, only one instruction can write to the register file at once, and they write in order, so by the time an instruction reaches the writeback stage, all previous instructions should have written back to the register file. The result that that instruction writes back however, is dependent on the values it has been forwarded. Thus, if the single cycle model, which directly reads from the register file, and the simulator, which can act based on forwarded values, both make the same change to the register file, the forwarding is transparent to the programming model and thus correct.

Python scripting was written to convert counter examples generated by this testbench into binaries that could be executed by the simulator in single step mode to allow debugging. This testbench takes around 35 minutes to complete on a commercial laptop, but this could potentially be improved through use of case splitting, by considering the effects of different instructions in separate tests. The successful execution of this testbench gives a good confidence in the correctness of the processor implementation.

3.6. Formal verification of reset sequence (`esbmc_equiv`)

`esbmc_isa` verifies only that the instruction retiring in the 9th clock cycle is correct, but it's hope is that that is a sufficient depth for all reachable pipeline states to be explored. This however doesn't consider the operation of the processor from reset. The `esbmc_equiv` testbench runs 7 clock cycles on the simulator from reset, then runs the reference simulator from reset until it has retired the same number of instructions. It then checks that the two have the same resultant state, and that the retiring instructions are the same. This proves that the two have the same transition relation over this set of instructions.

Because of greater sequential depth, this testbench takes significantly longer to complete than `esbmc_isa`. A python script was also written to convert counter examples generated by this testbench into unit tests that could be used with the unit testing framework. 5 such tests were generated over the course of development.

4. Showcase Program

A demonstration program to showcase the functionality of the processor by calculating and storing the squares of all integers between 0 and 200 was written before the simulator was developed. This allowed the instruction set to be tailored to this program, as well as the memory layout. A naïve implementation of such a program might use a multiply operation to calculate each square, however a multiply operation was forbidden for this coursework. A multiply operation could be emulated with other operations, for example using Booth's algorithm[7], which uses shift and add instructions. This requires a large number of instructions for each multiplication, so an alternate algorithm can be used based on a observation from equation 1 that consecutive squares can be calculated by adding $2n+1$.

$$(n+1)^2 = n^2 + 2n + 1 \tag{1}$$

Although this also contains multiplication, it is by a constant 2, which can be implemented cheaply as a left shift by 1 bit, or repeated addition. This leads to an initial unoptimised implementation presented in listings 5 and 6. Running this program with the simulator reports 1411 instructions executed in 1615 cycles.

Listing 5: C implementation

```
1 void square_0_to_200() {
2     uint32_t array[201];
3     uint32_t n_squared = 0;
4 
5     for (uint32_t n = 0; n < 201; n++) {
6         array[i] = n_squared;
7         n_squared += n + n + 1;
8     }
9 }
```

Listing 6: Unoptimised assembly

```
 1     addiu $t0, $zero, 0     # n
 2     addiu $t1, $zero, 0     # n^2
 3     addiu $t2, $zero, 0     # store address
 4     addiu $t3, $zero, 201   # branch value
 5 loop:
 6     sw    $t1, 0($t2)       # *store = n^2
 7     addu  $t1, $t1, $t0     # n^2 += n
 8     addu  $t1, $t1, $t0     # n^2 += n
 9     addiu $t1, $t1, 1       # n^2 += 1
10 
11     addiu $t0, $t0, 1       # n++
12     addiu $t2, $t2, 4       # store += 4
13     bne   $t0, $t3, loop    # while n != 201
```

Removing redundancy within the loop can be used for optimisation as the values of n, $2n$ and $4n$ are all calculated each iteration. All of the squares of 0 to 200 fit in 2 bytes (the maximum value is $256^2 - 1$), so instead of storing them as words in memory, they can be stored as half words. The n^{th} square is then stored at address $2n$, meaning n and $4n$ are no longer needed in the loop. This reduces the number of instructions in the loop and registers used. A smaller optimisation takes note of the fact that the square is calculated after it is stored, meaning in the last iteration of the loop the result is unused. By moving the store to the end of the loop and jumping to that point at the start, a few cycles can be saved. Listing 7 shows these optimisations, which execute 1006 instructions in 1211 cycles.

Listing 7: Optimised assembly without delay slots

```
 1     addiu $t0, $zero, 0     # 2n
 2     addiu $t1, $zero, 0     # n^2
 3     addiu $t2, $zero, 400   # branch value
 4     j     store
 5 loop:
 6     addu  $t1, $t1, $t0     # n^2 += 2n
 7     addiu $t1, $t1, 1       # n^2 += 1
 8     addiu $t0, $t0, 2       # 2n  += 2
 9 store:
10     sh    $t1,  0($t0)      # *(2n) = n^2
11     bne   $t0,  $t2, loop   # while 2n != 400
```

Listing 8: Optimised assembly with delay slots

```
 1     addiu $t0, $zero, 0     # 2n
 2     addiu $t2, $zero, 400   # branch value
 3     j     store
 4     addiu $t1, $zero, 0     # n^2
 5 loop:
 6     addu  $t1, $t1, $t0     # n^2 += 2n
 7     addiu $t0, $t0, 2       # 2n  += 2
 8     addiu $t1, $t1, 1       # n^2 += 1
 9 store:
10     bne   $t0,  $t2, loop   # while 2n != 400
11     sh    $t1,  0($t0)      # *(2n) = n^2
```

When delay slots are enabled, the instruction following a jump or branch is executed unconditionally. This allows the pipeline to never stall for branches. The showcase program can take advantage of this, by placing an instruction that jumps/branches do not depend on after them. This can be seen by comparing listings 7 and 8. This optimisation also requires other instructions to move in order to prevent the branch instruction from stalling, waiting for its operands. Therefore lines 7 and 8 of listing 7 have been swapped, so the branch does not depend on its previous instruction. Listing 8 requires no stalls after starting so executes 1006 instructions in 1010 cycles. The full output of the simulator for the showcase program can be seen in appendix B.

5. Conclusions

The processor simulator described in this report serves as a valuable learning tool, both for understanding the classic RISC 5-stage pipeline, and also for learning about MIPS assembly. Combining this with the formal based verification has provided an interesting experience, as well as giving greater confidence in the correctness of the control logic for the processor. This is useful, given the complexity of making a pipelined processor appear as if it isn't, and the potential pitfalls involved.

The processor could be extended by adding other instructions, as well as a kernel mode, allowing syscall emulation for more useful functions like output printing. The exception model could also be improved for this, allowing exceptions to jump to an exception handler instead of halting execution.

In terms of verification effort, formal verification checks a relatively small number of cycles, so a structured random instruction stream generator could be used to generate large amounts of instructions to explore deeper state spaces and check that the processor continues to agree with the reference model. Additionally the formal testbenches described here took a relatively long time to prove. This could be improved by applying case splitting - instead of checking all possible instructions in the formal testbench, it could be split into a number of smaller testbenches checking individual instructions, which are parallelisable for a faster solve time and easier debugging.

A. Simulator help output

```
$ mips_sim --help
Usage: mips_sim [options] input_file

Positional arguments:
input_file          path to a file to execute

Optional arguments:
-h --help           shows help message and exits [default: false]
-v --version        prints version information and exits [default: false]
-b --binary         execute a big endian binary [default: false]
-s --step           single step through the simulation [default: false]
-q --quiet          don’t print the output state, just check assertions [default: false]
-r --ref-core       run simulation using single cycle reference model [default: false]
-c --compare        run both models and compare the output [default: false]
-d --delay-slots    enable delay slot emulation [default: false]
-m --mem-size       data memory size in bytes [default: 512]
```

B. Showcase program output

Running the final delay slot based showcase program (Listing 8) produces the following output. The squares of the numbers 0 to 200 can be seen as 2 byte big-endian hex halfwords in the hex dump.

```
$ make showcase
build/mips_sim showcase_program/showcase.delay.asm -cdm 512
zero: 00000000   at: 00000000   v0: 00000000   v1: 00000000   a0: 00000000   a1: 00000000
  a2: 00000000   a3: 00000000   t0: 00000190   t1: 00009c40   t2: 00000190   t3: 00000000
  t4: 00000000   t5: 00000000   t6: 00000000   t7: 00000000   s0: 00000000   s1: 00000000
  s2: 00000000   s3: 00000000   s4: 00000000   s4: 00000000   s6: 00000000   s7: 00000000
  t8: 00000000   t9: 00000000   k0: 00000000   k1: 00000000   gp: 00000000   sp: 00000000
  fp: 00000000   ra: 00000000
             0    2    4    6    8    a    c    e   10   12   14   16   18   1a   1c   1e
00000000: 0000 0001 0004 0009 0010 0019 0024 0031 0040 0051 0064 0079 0090 00a9 00c4 00e1
00000020: 0100 0121 0144 0169 0190 01b9 01e4 0211 0240 0271 02a4 02d9 0310 0349 0384 03c1
00000040: 0400 0441 0484 04c9 0510 0559 05a4 05f1 0640 0691 06e4 0739 0790 07e9 0844 08a1
00000060: 0900 0961 09c4 0a29 0a90 0af9 0b64 0bd1 0c40 0cb1 0d24 0d99 0e10 0e89 0f04 0f81
00000080: 1000 1081 1104 1189 1210 1299 1324 13b1 1440 14d1 1564 15f9 1690 1729 17c4 1861
000000a0: 1900 19a1 1a44 1ae9 1b90 1c39 1ce4 1d91 1e40 1ef1 1fa4 2059 2110 21c9 2284 2341
000000c0: 2400 24c1 2584 2649 2710 27d9 28a4 2971 2a40 2b11 2be4 2cb9 2d90 2e69 2f44 3021
000000e0: 3100 31e1 32c4 33a9 3490 3579 3664 3751 3840 3931 3a24 3b19 3c10 3d09 3e04 3f01
00000100: 4000 4101 4204 4309 4410 4519 4624 4731 4840 4951 4a64 4b79 4c90 4da9 4ec4 4fe1
00000120: 5100 5221 5344 5469 5590 56b9 57e4 5911 5a40 5b71 5ca4 5dd9 5f10 6049 6184 62c1
00000140: 6400 6541 6684 67c9 6910 6a59 6ba4 6cf1 6e40 6f91 70e4 7239 7390 74e9 7644 77a1
00000160: 7900 7a61 7bc4 7d29 7e90 7ff9 8164 82d1 8440 85b1 8724 8899 8a10 8b89 8d04 8e81
00000180: 9000 9181 9304 9489 9610 9799 9924 9ab1 9c40 0000 0000 0000 0000 0000 0000 0000
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Halted due to exception:
    Cause: breakpoint
    EPC: 0x00003024
1006 instructions executed in 1010 cycles
```

C. Example output of single step mode

Single step mode shows the contents of the pipeline registers, previously executed instructions, general purpose registers and data memory at all times. It can be advanced one clock cycle by pressing enter.

```
      stage:  IF/ID     ID/EX     EX/MEM    MEM/WB
 instr name:        sh       bne     addiu     addiu
    stalled:         0         0         0         0
  exception:
instruction: a5090000  1548fffc  25290001  25080002
    address: 00003020  0000301c  00003018  00003014
         rs:                  10
        *rs:            00000190
         rt:                   8
        *rt:            000000d2  00002be3
  immediate:            fffffffc
eval branch:                   1
     branch:                   1
branch addr:            00003010
     alu op:                 nop
  alu b src:                 imm
 mem access:                none      none
  mem bytes:                   0         0
     wb reg:                   0         9         8
  wb result:            00000000  00002be4  000000d4
  0x00003014 (527): addiu
  0x00003018 (528): addiu
  0x0000301c (529): bne
  0x00003020 (530): sh
> 0x00003010 (531): addu
zero: 00000000   at: 00000000   v0: 00000000   v1: 00000000   a0: 00000000   a1: 00000000
  a2: 00000000   a3: 00000000   t0: 000000d2   t1: 00002be3   t2: 00000190   t3: 00000000
  t4: 00000000   t5: 00000000   t6: 00000000   t7: 00000000   s0: 00000000   s1: 00000000
  s2: 00000000   s3: 00000000   s4: 00000000   s4: 00000000   s6: 00000000   s7: 00000000
  t8: 00000000   t9: 00000000   k0: 00000000   k1: 00000000   gp: 00000000   sp: 00000000
  fp: 00000000   ra: 00000000
             0    2    4    6    8    a    c    e   10   12   14   16   18   1a   1c   1e
00000000: 0000 0001 0004 0009 0010 0019 0024 0031 0040 0051 0064 0079 0090 00a9 00c4 00e1
00000020: 0100 0121 0144 0169 0190 01b9 01e4 0211 0240 0271 02a4 02d9 0310 0349 0384 03c1
00000040: 0400 0441 0484 04c9 0510 0559 05a4 05f1 0640 0691 06e4 0739 0790 07e9 0844 08a1
00000060: 0900 0961 09c4 0a29 0a90 0af9 0b64 0bd1 0c40 0cb1 0d24 0d99 0e10 0e89 0f04 0f81
00000080: 1000 1081 1104 1189 1210 1299 1324 13b1 1440 14d1 1564 15f9 1690 1729 17c4 1861
000000a0: 1900 19a1 1a44 1ae9 1b90 1c39 1ce4 1d91 1e40 1ef1 1fa4 2059 2110 21c9 2284 2341
000000c0: 2400 24c1 2584 2649 2710 27d9 28a4 2971 2a40 2b11 0000 0000 0000 0000 0000 0000
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
00000140: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
00000160: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
00000180: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000001c0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[2] A. Reid *et al.*, "End-to-end verification of processors with isa-formal," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 42–58.

[3] K. Vollmar and P. Sanderson, "Mars," *ACM SIGCSE Bulletin*, vol. 38, no. 1, pp. 239–243, 2006.

[4] 2022. [Online]. Available: https://github.com/p-ranav/argparse

[5] M. R. Gadelha *et al.*, “Esbmc 5.0: an industrial-strength c model checker,” *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[6] C. Wolf, “Risc-v formal verification framework.” [Online]. Available: https://github.com/SymbioticEDA/riscv-formal

[7] A. D. Booth, “A signed binary multiplication technique,” *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, pp. 236–240, 1951.