CPU simulator FlyHip

Branko Šter[1]**, Paul Daubin**[2]

[1]*University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, 1000, Ljubljana*
[2]*ESIGELEC Engineering school, Av. Galilée, 76800 Saint-Étienne-du-Rouvray, France*
E-mail: branko.ster@fri.uni-lj.si

Abstract

An implementation of simulator of processor HIP, a little simplified version of MIPS I architecture, is described. Both non-pipeline and pipeline versions of HIP are supported. The simulator, written in Python, can be run either in a command-line mode or with a graphical user interface (GUI), implemented in Flutter, a modern open-source user interface software development kit. It can be run by default as a desktop application, but can potentially be adapted for running in a web browser or even as a mobile application.

1 Introduction

Although x86 instruction set architectures (ISA) are widespread commercially, CISC architectures are quite complex and as such not appropriate for undergraduate level courses, therefore courses usually employ various RISC architectures. The instruction set MIPS I by MIPS Technologies is a very popular RISC ISA, frequently used in computer architecture courses all over the world. For example, the MIPS I ISA was used in processors MIPS R2000 and R3000. For the purpose of teaching computer architecture in Faculty of computer and information science at University of Ljubljana, a little simplified version of MIPS I called HIP was proposed in the book [1] and used throughout the years.

The previous simulator for HIP CPU [4] was written with GUI that was based on WinMIPS64 software [5]. It was used in courses such as Computer systems architecture and similar courses at Faculty of Computer and Information Science. However, during the pandemic we needed to program Moodle quizzes for the evaluation of the students' work. These were written in Python programming language. Inspired by these scripts, the idea for writing a new simulator emerged.

The new simulator of the processor HIP is implemented in Python, a very popular programming language for some time already. The command-line version is called pyHip (as one would expect). Since the frontend pyHip (written in Python) is connected with backend Flutter GUI, we named the complete program FlyHip (**Fl**utter + py**Hip**).

The reason for developing the new simulator was not only the unavailability of the source code of the old one, but rather a greater flexibility of maintaining and upgrading one's own software. Additionally, since Flutter is a modern and very popular multi-platform open-source graphical user interface (GUI) software development kit (by Google), the GUI is way more flexible and expandable than the Windows GUI of WinMIPS64, which had to be frequently running in some 'compatibility mode' for older Windows versions, especially due to not flexible font size. In Flutter, more features and window-related options are available.

2 HIP CPU

HIP instruction set architecture is a little simplified MIPS I ISA, as such being a typical RISC CPU. It has a load/store architecture, meaning that ALU operations directly on operands in memory are not possible, so the operands first have to be loaded from the memory to registers in order to perform operations on them.

HIP supports only two instruction formats (Fig. 1). HIP ISA has 52 instructions: 31 in format 1 and 21 in format 2. Format 1 has one source register (Rs1), one input immediate operand (a constant), and a destination register (Rd). This format is used in all load/store instructions, as well as in all ALU instructions that take one immediate operand. In the load/store instructions, the immediate operand serves as the offset added to the base register (base addressing). Format 2 is used in all ALU instructions that take two input operands in registers.

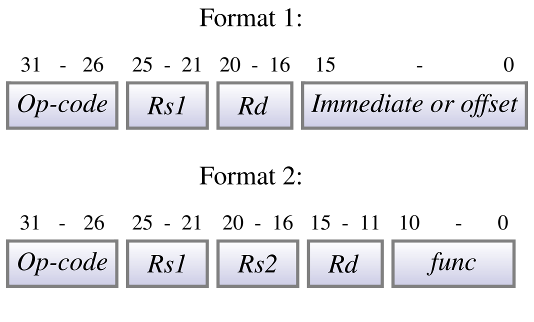


Figure 1: HIP supports two instruction formats: format 1 .

The HIP instructions may be divided into 4 groups:

- Load/store instructions
- ALU instructions:
  - Arithmetic operations add and subtract

    – Logical bitwise operations
    – Shift operations
    – Compare-and-Set operations
- Control instructions
- System instructions

All the instructions are listed in Table 1. There are no

Table 1: HIP instruction set architecture.

Load/store instructions: LB (load byte), LH (load halfword), LW (load word), LBU (LB unsigned), LHU (LH uns.), SB (store byte), SH (store halfword), SW (store word)
ALU instructions: *Arithmetic operations add and subtract*: ADD (add signed), SUB (subtract sig.), ADDU (add uns.), SUBU, ADDI (add signed immediate), SUBI (sub. signed imm.), ADDUI (add uns. imm.), SUBUI (sub. uns. imm.)
Logical bitwise operations: AND, OR, XOR, NOT (one's complement), ANDI (imm.), ORI, XORI
Shift operations: SLL (shift left logical), SRL (sh. right log.),SRA (sh. right arith.), SLLI (SLL imm.), SRLI (SRL imm.), SRAI, LHI (load high imm.)
Compare-and-Set operations: SEQ (set if equal), SNE (set if not eq.), SLT (set if less than), SGT (set if greater than), SLTU (SLT uns.), SGTU (SGT uns.), SEQI (SEQ imm.), SNEI, SLTI, SGTI, SLTUI, SGTUI
Control instructions: BNE (branch if not equal zero), BEQ (branch if eq. zero), J (jump, unconditional), CALL (jump to subroutine), TRAP (jump to vector address), RFE (return from exception)
System instructions: EI (enable interrupts), DI (disable int.), MOVER (move from EPC to register), MOVRE (move from reg. to EPC)

multiply/divide instructions and no support for floating point instructions.

The organization of the basic (non-pipeline) version is shown in Fig. 2. The control unit (on the left) implements a state machine with 28 states. The right side shows the datapath, including ALU and various registers: 32 general-purpose registers R0, R1, ..., R31, with interface registers A, B and C, and special-purpose registers, such as PC (Program Counter), EPC (Exception Program Counter), which is basically the back-up register of PC when performing interrupt service routines, MAR (Memory Address Register) and MDR (Memory Data Register).

To simplify the operation, the user does not have to define available code and data segments in the simulator settings, in contrast to simulator WinHIP, where students always have to make sure that these settings coincide. In pyHip (and FlyHip), the simulator automatically allocates the segments given in the assembly source. A specified default size of the segments can be set in advance.

3 Simulator

The simulator is written in Python programming language. The command-line version is called *pyHip*. It loads a user-defined HIP assembly source file, runs it through lexer and parser, which are implemented in a manner similar to [6, 7].

The non-pipeline version executes instructions sequentially one after another. Each instruction has its specific length in the number of clock cycles - caused by the specific CPU organization of HIP (Fig. 2), as defined in [1]. The instructions last form 3 to 6 clock cycles. Depending

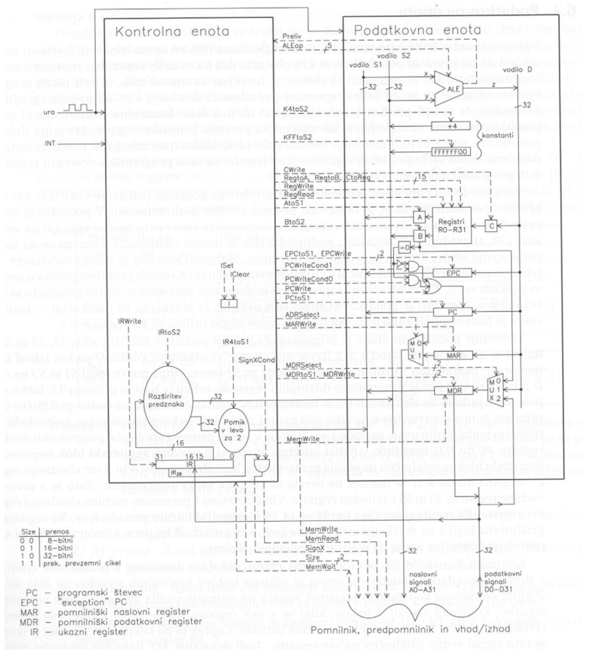


Figure 2: Non-pipeline HIP CPU (from [1]).

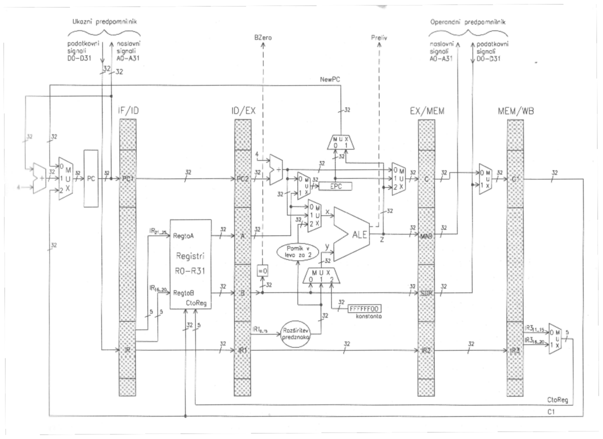


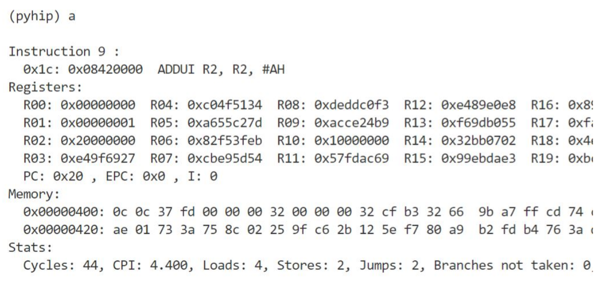
Figure 3: Pipeline HIP CPU datapath (from [1]).

on a program running on HIP, the average CPI (Clock Per Instruction) is usually between 4 and 5. Although the non-pipeline simulator does not operate on the cycle-level, it still accumulates the number of clock cycles for the statistics, such as the execution time and the average CPI of a program. An example of the HIP assembly code:

```
        .data
        .org 0x400
POD:    byte 9,2,0,4,7,0,1,2,0,0,0,0

        .code
        .org 0x0
        addui r9, r0, POD
LOOP:   lb r1, 0(r9)
        lb r2, 1(r9)
        sub r3, r1, r2
        subi r3, r3, 1
        sb 2(r9), r3
        slti r17, r3, 0
        addui r9, r9, 3
        beq r17, LOOP
        halt
```

Fig. 4 shows an example of the command-line pyHip output.



```
(pyhip) a

Instruction 9 :
  0x1c: 0x08420000  ADDUI R2, R2, #AH
Registers:
  R00: 0x00000000  R04: 0xc04f5134  R08: 0xdeddc0f3  R12: 0xe489e0e8  R16: 0x8
  R01: 0x00000001  R05: 0xa655c27d  R09: 0xacce24b9  R13: 0xf69db055  R17: 0xf
  R02: 0x20000000  R06: 0x82f53feb  R10: 0x10000000  R14: 0x32bb0702  R18: 0x4
  R03: 0xe49f6927  R07: 0xcbe95d54  R11: 0x57fdac69  R15: 0x99ebdae3  R19: 0xb
  PC: 0x20 , EPC: 0x0 , I: 0
Memory:
  0x00000400: 0c 0c 37 fd 00 00 00 32 00 00 00 32 cf b3 32 66  9b a7 ff cd 74
  0x00000420: ae 01 73 3a 75 8c 02 25 9f c6 2b 12 5e f7 80 a9  b2 fd b4 76 3a
Stats:
  Cycles: 44, CPI: 4.400, Loads: 4, Stores: 2, Jumps: 2, Branches not taken: 0
```

Figure 4: An example of stepping through the code with the non-pipeline command-line simulator pyHip.

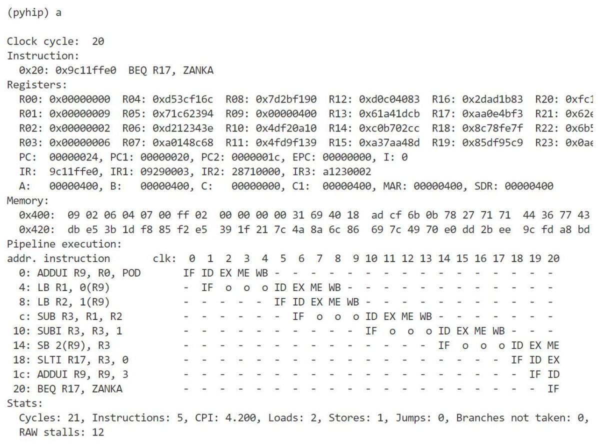
On the other hand, the pipeline version has to operate on the cycle-level. HIP has a classical 5-stage pipeline, consisting of the following stages: IF (Instruction Fetch), ID (Instruction Decode), EX (EXecute), ME (MEmory), and WB (WriteBack) (Fig. 3), as can be seen in the command-line pipeline pyHip ouput in Fig. 5.

Since the pipeline datapath is a synchronous digital circuit, all the registers renew their state at the same time, typically at the rising edge of the clock signal. Since in a software simulator this is not possible to do directly, the synchronicity is achieved by sequential updating of different stages from right to left (in Fig. 3), due to the fact that instructions move through the pipeline from left to right, similarly to a shift register. However, a care must be taken in cases of feedback connections, for example, register C1 is 'written back' to a destination register in the WB phase.

By default, the pipeline HIP CPU detects data hazards of type RAW (Read-After-Write). Other types of data hazards, such as Write-After-Write and Write-After-Read, cannot occur in HIP due to its simplicity. RAW hazards are detected by additional logic that compares the address (number) of the destination register in the instruction in register IR3 (just prior to the WB phase) to the addresses of the source registers in phases IR2, IR1, and IR (the next three instructions). In case of the same register being written and then read a little later, it must be assured that the WB phase of the write is done before the ID (decoding) phase of the read, so that the register is actually *read after being written* by an earlier instruction. If necessary, pipeline interlocks ('bubbles') need to be inserted to ensure the correct operation, in effect by performing the NOP operations.

The pipeline supports additional options of bypassing (data-forwarding) and delayed branches. *Bypassing* requires additional logic for detecting the RAW data hazards, as just described, to determine which of the result registers (C1, C or even the ALU result) is written back directly via additional feedback connections to the source registers A and B, in order to avoid stalls.

When conditional branches in the code are actually performed (the condition it TRUE), two next instructions, which are already in the pipeline stages IF and ID, need to be cancelled, since they must not be executed. Thus two additional stalls occur whenever a branch is done. However, the option of *delayed branches* allows for re-ordering the code - putting at most two instructions from before the branch to after the branch, in order to avoid the two stalls.



```
(pyhip) a

Clock cycle:  20
Instruction:
  0x20: 0x9c11ffe0  BEQ R17, ZANKA
Registers:
  R00: 0x00000000  R04: 0xd53cf16c  R08: 0x7d2bf190  R12: 0xd0c04083  R16: 0x2dad1b83  R20: 0xfc
  R01: 0x00000009  R05: 0x71c62394  R09: 0x00000400  R13: 0x61a41dcb  R17: 0xaa0e4bf3  R21: 0x62
  R02: 0x00000002  R06: 0xd212343e  R10: 0x4df20a10  R14: 0xc0b702cc  R18: 0x8c78fe7f  R22: 0x6b
  R03: 0x00000006  R07: 0xa0148c68  R11: 0x4fd9f139  R15: 0xa37aa48d  R19: 0x85df95c9  R23: 0x0a
  PC:  00000024, PC1: 00000020, PC2: 0000001c, EPC: 00000000, I: 0
  IR:  9c11ffe0, IR1: 09290003, IR2: 28710000, IR3: a1230002
  A:   00000400, B:   00000400, C:   00000000, C1:  00000400, MAR: 00000400, SDR: 00000400
Memory:
  0x400:  09 02 06 04 07 00 ff 02  00 00 00 00 31 69 40 18  ad cf 6b 0b 78 27 71 71  44 36 77 43
  0x420:  db e5 3b 1d f8 85 f2 e5  39 1f 21 7c 4a 8a 6c 86  69 7c 49 70 e0 dd 2b ee  9c fd a8 bd
Pipeline execution:
addr. instruction       clk:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
  0: ADDUI R9, R0, POD       IF ID EX ME WB -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
  4: LB R1, 0(R9)            -  IF  o  o  o ID EX ME WB -  -  -  -  -  -  -  -  -  -  -  -
  8: LB R2, 1(R9)            -  -  -  -  -  IF ID EX ME WB -  -  -  -  -  -  -  -  -  -  -
  c: SUB R3, R1, R2          -  -  -  -  -  -  IF  o  o  o ID EX ME WB -  -  -  -  -  -  -
 10: SUBI R3, R3, 1          -  -  -  -  -  -  -  -  -  -  IF  o  o  o ID EX ME WB -  -  -
 14: SB 2(R9), R3            -  -  -  -  -  -  -  -  -  -  -  -  -  -  IF  o  o  o ID EX ME
 18: SLTI R17, R3, 0         -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  IF ID EX
 1c: ADDUI R9, R9, 3         -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  IF ID
 20: BEQ R17, ZANKA          -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  IF
Stats:
  Cycles: 21, Instructions: 5, CPI: 4.200, Loads: 2, Stores: 1, Jumps: 0, Branches not taken: 0,
  RAW stalls: 12
```

Figure 5: An example of stepping through the code with the pipeline command-line simulator pyHip. In the presented case, no bypassing (data forwarding) was used, so three RAW (Read-After-Write) stalls or 'bubbles' are inserted.

4 Graphical User Interface (GUI)

4.1 Objectives of the GUI

The main objective of the GUI is to present the information in a way that is comprehensible - accessible to users and efficient: showing clearly the crucial data without bloating and allowing interaction between the user and the simulator. For this project we also wanted to be able to deploy on multiple platforms as needed. Mainly desktop platforms: Windows, Linux, and MacOs, but also on web platforms for a more lightweight experience, and eventually a smaller version on mobile.

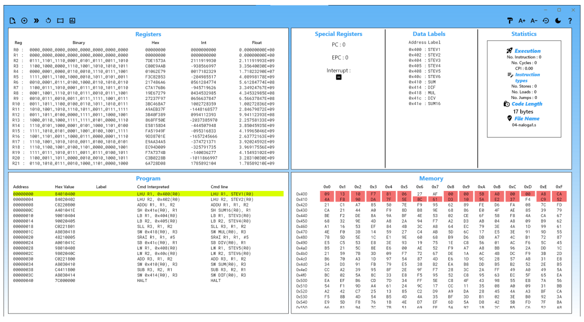
4.2 Technologies used

To that end, we decided developing the simulator's interface using Flutter, a framework built by Google around the Dart programming language. It is a recent technology (2017), but is backed by a powerful company and thus provides a level of future proofing, while allowing us to satisfy the specifications, being able to deploy on several platforms with the same code. Desktop support was also officially released only a few months ago, making it a perfect opportunity.

Our interface, using the MVC (model-view-controller) design pattern can be decomposed into 3 main components:

- Views, where information is displayed and interacted with by the users. In Flutter, each element, from the global scaffold of the application to the smallest text, is built using Widgets, predefined and highly customisable blocks, that can be adjusted to suit the user's needs, and put one on top of another. The resulting widget tree defines how the information is accessed, displayed, and under which form.

- The controller is what allows us to predefine HTTP requests that the user is going to send to the Python backend. For example, we have set up a request, using a specific IP and port, that requests the simulator to move forward in the simulation. By simply pressing a button, the user can send the instruction to the simulator. This is also the way we get the information, in JSON form.
- The model is where this information is stocked. We define the elements we wish to extract from the JSON file, decompose it in its various information elements, and extracts the ones we wish to use. This information is mapped into multiple classes or 'models', which allows us to manipulate it.



Registers

```
Reg     Binary                                      Hex         Int           Float
R0  :   0000_0000_0000_0000_0000_0000_0000_0000     00000000    0000000000    0.00000000E+00
R1  :   0000_0000_0000_0000_0000_0000_0000_0000     00000000    0000000000    0.00000000E+00
R2  :   0111_1101_1110_0001_0101_0111_0011_1010     7DE1573A    2111919930    2.11191993E+09
R3  :   1100_1000_0000_1110_1001_1010_1010_1011     C80E9AAB    -938566997    3.35640030E+09
R4  :   0000_0001_0000_0110_0010_1110_0111_1001     01062E79    0017182329    1.71823290E+07
R5  :   1111_0011_1100_1000_0010_1011_0101_0011     F3C82B53    -204985517    4.08998178E+09
R6  :   0010_0001_0111_0100_1000_0110_1010_0110     217486A6    0561284774    5.61284774E+08
R7  :   1100_0111_1010_0001_0111_0110_1011_0110     C7A176B6    -945719626    3.34924767E+09
R8  :   0001_1001_1110_0110_0111_0010_0111_1001     19E67279    0434532985    4.34532985E+08
R9  :   0010_0111_0010_0011_0111_1111_1001_0111     27237F97    0656637847    6.56637847E+08
R10 :   0011_1011_1100_0100_0110_1011_1010_0111     3BC46BA7    1002728359    1.00272836E+09
R11 :   1010_1001_1010_1110_1011_0011_0111_1111     A9AEB37F    -1448168577   2.84679872E+09
R12 :   0011_1011_0100_0000_1111_0011_1000_1001     3B40F389    0994112393    9.94112393E+08
R13 :   1000_0110_1000_1111_1111_0101_0000_1110     868FF50E    -2037385970   2.25758133E+09
R14 :   1110_0101_1000_0001_0101_1000_1101_0100     E58158D4    -444507948    3.85045935E+09
R15 :   1111_1010_0101_0001_1001_0100_1001_1111     FA51949F    -095316833    4.19965046E+09
R16 :   1001_1101_0011_1000_0111_0000_0001_1110     9D38701E    -1657245666   2.63772163E+09
R17 :   1110_1001_1010_1010_0011_0100_1010_0101     E9AA34A5    -374721371    3.92024592E+09
R18 :   1110_1100_1001_0100_1101_0000_0000_1001     EC94D009    -325791735    3.96917556E+09
R19 :   1111_0111_1010_0111_0011_0111_0100_1011     F7A7374B    -140036277    4.15493102E+09
R20 :   1100_0011_1011_0000_0010_0010_1000_1011     C3B0228B    -1011866997   3.28310030E+09
R21 :   0110_1010_0111_0010_1000_1101_0000_1000     6A728D08    1785892104    1.78589210E+09
```

Special Registers

PC : 0

EPC : 0

Interrupt :

Data Labels

```
Address Label
0x400  : STEV1
0x402  : STEV2
0x404  : STEV3
0x405  : STEV4
0x408  : STEV5
0x40c  : STEV6
0x410  : SUM
0x414  : DIF
0x418  : MUL
0x41c  : DIV
0x41e  : SUM16
```

Statistics

Execution
No. Instruction : 0
No. Cycles : 0
CPI : 0.00
Instruction types
No. Stores : 0
No. Loads : 0
No. Jumps : 0
Code Length
17 bytes
File Name
04-naloga1.s

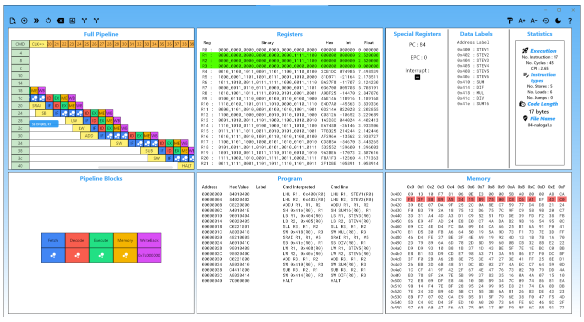
Program

```
Address     Hex Value   Label   Cmd Interpreted         Cmd line
00000000    84010400            LHU R1, 0x400(R0)       LHU R1, STEV1(R0)
00000004    84020402            LHU R2, 0x402(R0)       LHU R2, STEV2(R0)
00000008    C8220800            ADDU R1, R1, R2         ADDU R1, R1, R2
0000000C    A401041E            SH 0x41e(R0), R1        SH SUM16(R0), R1
00000010    90010404            LB R1, 0x404(R0)        LB R1, STEV3(R0)
00000014    90020405            LB R2, 0x405(R0)        LB R2, STEV4(R0)
00000018    C0221801            SLL R3, R1, R2          SLL R3, R1, R2
0000001C    A8030418            SW 0x418(R0), R3        SW MUL(R0), R3
00000020    48210005            SRAI R1, R1, #5         SRAI R1, R1, #5
00000024    A001041C            SB 0x41c(R0), R1        SB DIV(R0), R1
00000028    98010408            LW R1, 0x408(R0)        LW R1, STEV5(R0)
0000002C    9802040C            LW R2, 0x40c(R0)        LW R2, STEV6(R0)
00000030    C0221800            ADD R3, R1, R2          ADD R3, R1, R2
00000034    A8030410            SW 0x410(R0), R3        SW SUM(R0), R3
00000038    C4411800            SUB R3, R2, R1          SUB R3, R2, R1
0000003C    A8030414            SW 0x414(R0), R3        SW DIF(R0), R3
00000040    7C000000            HALT                    HALT
```

Memory

```
        0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x400   09  13  10  F7  81  06  27  AF  00  00  5B  A0  00  00  A8  CA
0x410   4A  F8  90  DA  7F  5E  8C  61  D3  10  5A  E2  37  F4  C9  52
0x420   21  C1  A7  85  50  7E  F9  95  62  89  FE  D6  FA  08  7C  FD
0x430   CA  21  44  A0  F9  8D  B8  9E  68  86  E0  4F  AE  85  39  79
0x440   BE  F2  DE  8A  9A  BF  4E  53  02  CE  6F  58  F8  4A  CA  67
0x450   68  32  9E  4D  AB  2A  94  F7  A2  D3  AB  04  A8  09  B9  62
0x460   A1  16  53  EF  84  4B  3C  A8  64  EC  79  3E  4A  1D  99  61
0x470   4E  F0  3B  14  59  27  C4  AB  5D  6C  17  E5  3E  91  9D  55
0x480   78  5D  5E  1C  E1  9E  60  68  B9  D6  DB  A7  4C  B1  72  87
0x490   E5  C5  53  E8  3E  93  19  75  1E  C8  56  01  AC  F6  5C  45
0x4a0   B5  21  5C  8E  E6  00  AE  52  F9  A7  A8  BB  96  2A  DD  1C
0x4b0   21  59  7B  3D  09  F7  72  67  DE  1A  AC  4B  DC  F9  3B  2D
0x4c0   B6  70  A3  1D  97  54  87  4D  E6  1D  9C  28  57  AB  31  E8
0x4d0   34  D3  91  FB  79  E5  38  B2  EA  B8  DD  B5  B2  52  2E  B5
0x4e0   CC  A2  39  95  8F  2E  9F  F7  28  3C  2A  FF  49  A0  49  5A
0x4f0   BC  02  5A  8C  33  E8  F5  95  52  C8  95  63  EC  5F  65  EA
0x500   EA  EF  B6  CD  7D  34  FF  5E  C8  4F  43  98  55  EB  7A  56
0x510   54  F1  9D  A4  61  24  9C  17  CC  11  35  08  A0  09  31  BB
0x520   A2  42  C7  25  13  85  C2  D9  A9  DA  28  45  4A  A3  BF  CA
0x530   F5  8B  4D  54  B5  4D  4A  35  BF  3D  B1  02  3E  B0  92  3A
0x540   E9  5D  F8  76  1B  4E  D7  EF  6D  5A  D8  42  5B  FD  7F  BA
```

Figure 6: An example of the current GUI on desktop, showing the basic version of the simulator



Registers

```
Reg     Binary                                      Hex     Int     Float
R0  :   0000_0000_0000_0000_0000_0000_0000_0000     000000  000000  0.000000
R1  :   0000_0000_0000_0000_0000_0000_1111_1100     000000  000000  2.520000
R2  :   0000_0000_0000_0000_0000_0000_1111_1100     000000  000000  2.520000
R3  :   0000_0000_0000_0000_0000_0000_0000_0000     000000  000000  0.000000
R4  :   0010_1100_1011_0001_1101_1100_1110_0100     2CB1DC  074985  7.498539
R5  :   1000_0001_1101_1001_0111_0001_1010_0000     81D971  -21164  2.178511
R6  :   1011_1010_0011_0111_1111_1000_0011_1110     BA37F8  -11707  3.124230
R7  :   0000_0011_0110_0111_0000_0000_0011_1101     036700  005708  5.708191
R8  :   1010_1001_1011_1111_0010_0101_0001_0001     A9BF25  -14470  2.847876
R9  :   0100_0110_1110_0001_0100_0110_0100_1000     46E146  118916  1.189168
R10 :   1110_0100_1101_0111_1010_0000_0110_1110     E4D7A0  -45563  3.839336
R11 :   0000_1101_0010_0001_0100_1010_1001_1001     0D214A  022028  2.202855
R12 :   1100_0000_1000_0001_0010_0110_1010_1000     C08126  -10652  3.229689
R13 :   0001_1010_0011_1101_1000_1100_1010_0010     1A3D8C  044024  4.402413
R14 :   1110_1010_0111_0100_1000_1011_1010_1100     EA748B  -36146  3.933506
R15 :   0111_1111_1011_0011_0010_0101_0010_1001     7FB325  214244  2.142446
R16 :   1010_1111_0010_1001_0110_1010_1100_0100     AF296A  -13562  2.938727
R17 :   1100_1101_1000_1000_0101_1010_0101_0010     CD885A  -84670  3.448265
R18 :   0101_0011_0011_0101_0101_0010_0111_0111     533552  139600  1.396003
R19 :   1001_1010_0011_1011_1110_0110_0010_1010     9A3BE6  -17073  2.587616
R20 :   1111_1000_1010_0001_1111_0011_0000_1111     F8A1F3  -12360  4.171363
R21 :   0011_1111_0001_1101_1011_1110_1101_0011     3F1DBE  105891  1.058914
```

Special Registers

PC : 84

EPC : 0

Interrupt :

Data Labels

```
Address Label
0x400  : STEV1
0x402  : STEV2
0x404  : STEV3
0x405  : STEV4
0x408  : STEV5
0x40c  : STEV6
0x410  : SUM
0x414  : DIF
0x418  : MUL
0x41c  : DIV
0x41e  : SUM16
```

Statistics

Execution
No. Instruction : 17
No. Cycles : 45
CPI : 2.65
Instruction types
No. Stores : 5
No. Loads : 6
No. Jumps : 0
Code Length
17 bytes
File Name
04-naloga1.s

Program

```
Address     Hex Value   Label   Cmd Interpreted         Cmd line
00000000    84010400            LHU R1, 0x400(R0)       LHU R1, STEV1(R0)
00000004    84020402            LHU R2, 0x402(R0)       LHU R2, STEV2(R0)
00000008    C8220800            ADDU R1, R1, R2         ADDU R1, R1, R2
0000000C    A401041E            SH 0x41e(R0), R1        SH SUM16(R0), R1
00000010    90010404            LB R1, 0x404(R0)        LB R1, STEV3(R0)
00000014    90020405            LB R2, 0x405(R0)        LB R2, STEV4(R0)
00000018    C0221801            SLL R3, R1, R2          SLL R3, R1, R2
0000001C    A8030418            SW 0x418(R0), R3        SW MUL(R0), R3
00000020    48210005            SRAI R1, R1, #5         SRAI R1, R1, #5
00000024    A001041C            SB 0x41c(R0), R1        SB DIV(R0), R1
00000028    98010408            LW R1, 0x408(R0)        LW R1, STEV5(R0)
0000002C    9802040C            LW R2, 0x40c(R0)        LW R2, STEV6(R0)
00000030    C0221800            ADD R3, R1, R2          ADD R3, R1, R2
00000034    A8030410            SW 0x410(R0), R3        SW SUM(R0), R3
00000038    C4411800            SUB R3, R2, R1          SUB R3, R2, R1
0000003C    A8030414            SW 0x414(R0), R3        SW DIF(R0), R3
00000040    7C000000            HALT                    HALT
```

Memory

```
        0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x400   09  13  10  F7  81  06  0E  E3  00  00  5B  A0  00  00  A8  CA
0x410   FE  2F  88  B9  A5  34  15  B9  75  00  C8  C6  A1  EF  43  C0
0x420   39  BE  07  DA  9F  25  2C  0A  8E  C7  59  77  D4  D8  21  24
0x430   F0  B3  79  2A  18  75  12  D6  75  7C  9F  C9  58  9B  20  C7
0x440   3D  31  A4  4D  A3  D1  C9  52  51  FD  DE  39  FD  F2  38  FB
0x450   86  E9  4F  AD  24  E8  E0  C7  4A  DA  B2  9B  16  54  95  0C
0x460   09  CC  4E  D4  FC  BA  09  E4  CA  A6  25  B1  6A  91  F0  41
0x470   B1  D5  30  FB  A6  64  50  19  5A  9D  73  F1  73  7E  3D  FF
0x480   46  D4  FE  37  BE  3F  4E  A9  19  92  4D  13  1B  7B  1A  70
0x490   2D  79  89  6A  6D  78  2D  8D  59  60  0B  CB  32  8B  E2  22
0x4a0   D9  D9  93  10  B8  1B  37  1D  43  BE  5F  7E  1E  BC  C0  BB
0x4b0   E8  B1  53  D9  CD  E7  98  A3  71  3A  95  86  E7  F0  DC  BF
0x4c0   3F  F0  2B  A6  2B  8E  75  3E  47  27  3E  41  FF  25  8E  D1
0x4d0   26  BB  3D  6B  48  51  DC  8D  02  27  4A  EC  C7  64  59  0D
0x4e0   1C  CF  41  9F  42  2F  67  4E  47  76  73  02  70  79  DD  4A
0x4f0   BD  78  8F  2A  7E  5B  99  37  83  35  16  0A  4A  07  15  10
0x500   72  E8  09  DF  E8  46  10  DB  B9  34  7C  09  74  86  81  EA
0x510   98  14  F4  7E  BF  28  95  24  99  95  E8  21  74  EA  0D  DB
0x520   7E  24  3D  B9  6D  5B  C1  55  3B  6A  81  26  B3  DE  43  23
0x530   8B  F7  07  02  CA  E9  B5  81  5F  79  6E  38  F0  47  F5  4D
0x540   5D  C4  0C  D4  3F  ED  10  A0  20  73  64  FE  6C  46  8C  2F
```

Figure 7: An example of the current GUI on desktop, showing the pipeline of the simulator (subject to change)

4.3 Features

The interface has been designed keeping both students and teachers in mind, giving users as much tools as possible but staying straightforward and clear. It is used by providing an assembly file using a file explorer and then compiled by the Python program. Then, the execution can be followed step by step, in the non-pipeline version (6) at the granularity of instructions, and in the pipeline version (7) at the granularity of clock cycles, and highlighting the important information. Alternatively, the program can be run until the end to see the end result. Then, users can either select a new file or simply rewind the current one at their convenience.

The program provides numerous features allowing the teacher to customize the desired way of showing the information to the student, including changing the colour of the interface, zooming on important information and more. These options are intuitive, done either using the buttons on the top left, or simply scrolling or clicking on the interface. The students are able to get more information on different elements by a simple mouseover, allowing us to keep the interface clean while still keeping them afloat. Each step of the program is also highlighted to show the relevant information, so they can identify and focus on the most crucial points.

The program displays lots of information, under different forms, ranging from a simple table for registers (bottom right), to an intricate and adaptive design for the pipeline (top left), and designs in between, adapted for the data they contain. Additionally, the interface is decomposed in different screens, between which one can seamlessly navigate as in any website or application. The source code of the program is available at [8].

5 Conclusion

A CPU simulator is a very valuable tool for students learning computer architecture, since on practical examples of assembly programs students can gradually learn how the CPU behaves. They can view registers and memory, explore runtime statistics, performance issues, pipeline stalls, cache behaviour, and many more.

We describe an implementation of simulator of processor HIP, a little simplified version of processor MIPS R3000. Both non-pipeline and pipeline versions of HIP are supported. The simulator can be run either in a command-line mode or with a GUI implemented in Flutter. It can be run by default as a desktop application, but can potentially be adapted for running in a web browser, with the advantage of being more system-independent.

In the recent years, the RISC-V open standard ISA is growing in popularity, gradually also in university courses. Besides of being open-source, it is also more modern. We may use the same framework to develop also a RISC-V simulator.

6 Acknowledgements

We thank Ratko Pilipović and Nejc Ilc for their help and advices in designing the GUI.

References

[1] D. Kodek: Arhitektura in organizacija računalniških sistemov, Bi-Tim, Šenčur, 2008.

[2] D. A. Patterson, J. L. Hennessy: Computer organization and design: The hardware/software interface, Morgan Kauffman, 2012.

[3] A. Akram, L. Sawalha: A Survey of Computer Architecture Simulation Techniques and Tools, IEEE Access, 2019.

[4] D. Šonc: Simulator WinHIP, FRI, 2007-2012.

[5] M. Scott: WinMIPS64, http://indigo.ie/ mscott/

[6] A. Z. Henley: teenytinycompiler, https://github.com/AZHenley/teenytinycompiler

[7] A. Z. Henley: Let's make a Teeny Tiny compiler, https://austinhenley.com/blog/teenytinycompiler1.html

[8] https://github.com/brankoster/pyhip