## 1. IDEA SIMULATOR

This report describes a behavioral level simulator written for the iDEA processor in order to model and accurately predict the behavior of the instructions' execution. The simulator is written in Python and available as open source. Using the simulator, we can obtain performance metrics such as instruction count and clock cycles, as well as ensuring logical and functional correctness. By analyzing the statistics gathered from the simulations, we can predict potential savings and impact of hardware changes. Here, we present the simulator, compiler and benchmark programs used. The simulator consists of two main components; an instruction parser and the pipeline simulator itself.

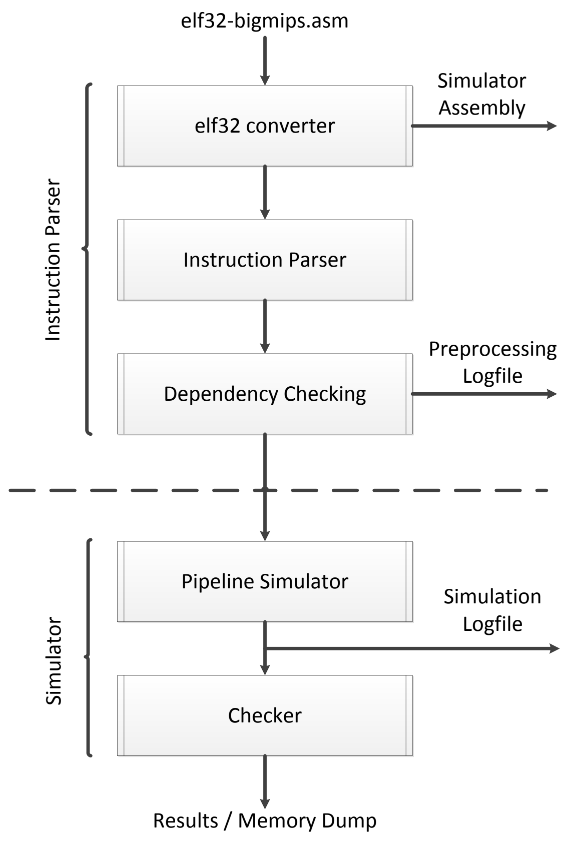

**Fig. 1**. Simulator data flow

### 1.1. Instruction Parser

The instruction parser takes as input an elf32-bigmips format assembly file, the standard format produced by mips-elf-gcc. The benchmark programs are written in C and compiled using mips-elf-gcc with the appropriate flags (-c -g) set to produce the elf32 assembly code.

In the instruction parser, the elf32 file is parsed and converted to an intermediate, simplified assembly format. The resulting assembly code is run through an instruction parser, which also expands pseudo-instructions and performs instruction substitutions where necessary, to fit the iDEA instruction set. As a final preprocessing stage, the instructions are checked for dependencies and NOP instructions are inserted where necessary to avoid data hazards. Because the iDEA architecture does not currently implement hardware stalling or data forwarding, dependencies have to be resolved in software.

A dependency is detected if the destination register of an instruction is equal to an operand register or the destination register of a previous instruction at most $N$ instructions before (taking into account the $L$ NOP insertion list). Overlapping stores are defined as any store instructions with references to the same memory location. After the data dependencies have been found in this first iteration, all branch and jump targets are reevaluated according to the new instruction memory locations. In the third and final iteration, branches and jumps are checked for dependencies across PC-changes, which may require additional NOPs to be inserted. Finally, the branch and jump targets are again reevaluated. The final instruction list is passed on to the simulator.

The output of the preprocessor and instruction parser consists of two files: a preprocessing log file containing information about code adjustments to fit iDEA, NOP-insertions and instruction expansions, as well as a file containing the assembly code as input to the simulator.

### 1.2. Simulator

The simulator is a behavioral-level simulator modelling the iDEA pipeline, written in Python. It can be configured to model a variable number of pipeline stages and different pipeline configurations. All iDEA instructions are supported by the simulator. The simulator implementation is based on an open-source project, a very simple MIPS simulator modelling the basic MIPS pipeline and a small core MIPS instruction set using a specialised assembly format for the input file. This limitation of the original simulatior means that examples have to be hand coded. The iDEA simulator extends the basic simulator to work on more practical problems and directly on the assembly code produced by mips-gcc. This allows us to automate the simulation process, from a C program to a simulation log file and run statistics file, with automatic checking of correctness. The main modifications made on the starting point simulator are as follows.

- Conversion from elf32-bigmips to simulator assembly
- Data dependency checking and NOP insertion
- Improved statistics collection
- Expanded the supported instruction set

- Variable pipeline length
- Easy-to-use command line interface
- Logfiles of simulation and preprocessing
- Error handling for input and memory references
- Support for subroutine calls
- Memory dump and automated correctness checking
- Modelling data and instruction memory
- Support for preloading data memory

Our simulator models the iDEA pipeline stage by stage and the instruction passing between stages. Register file, data memory and instruction memory are modelled individually. By enabling the verbose output option for the simulation log file, the simulator will output detailed information about the state of the instructions in the pipeline and the register values for each clock cycle. The pipeline length and configuration are fully customizable.

The pipeline simulator uses one branch delay slot (to fit the code generated for the MIPS platform), but this can easily be modified. Statistics collected during the simulation run are cycle count, NOP count, simulation run time and **core instruction** count.

By putting the tags **START_CCORE** and **END_CCORE** in the elf32 assembly input file, the simulator will count the instructions executed within those tags as core instructions and output this count as a separate statistic. This can be useful to ignore setup/initialization and finalization code, instead only counting the instructions excecuted within the computation itself.

The main limitations of the simulator is that it is dependent on the elf32 format as input, and the parsing is not very robust in the sense. The output files generated by the simulator are simple text files, and all debugging and output tracing has to be done manually, which is tedious for large programs. Large simulations (100,000s of cycles) take several seconds to run.

## 1.3. Usage

The command line interface of the simulator is used as follows.

```
python run-simulator.py [options]
<Input ASM> <Sim. Log>
<Sim. ASM File> <Preproc. Log>
<Memory Dump>
```

Example:

```
usr@sys:~$ python src/run-simulator.py -v
 benchmark/toy/median/median-O2.asm
```

This will run the elf32-bigmips assembly file median-O2 with verbose output.

Only the Assembler Input File is a required input, others are optional (default filenames are used).

**Options**

| | |
|---|---|
| –version | Show program's version number and exit. |
| -h, –help | Show help message and exit. |
| -v, –verbose | Verbose debug output. |
| -c, –core | Show only core cycle information. |
| -q, –quiet | Supress simulator output. |
| -m, –mute | Supress all output. |
| -p N | Set number of pipeline stages. |
| -f N | Set number of IF stages. |
| -d N | Set number of ID stages. |
| -e N | Set number of EX/MEM stages. |
| -w N | Set number of WB stages. |
| -s N | Set execution start address [default @main]. |

The pipeline must contain at least 4 stages (IF-ID-EX-WB). If the pipeline is deeper than 4 stages, the EX/MEM stage will be allocated more cycles, up to a maximum of 4. After that, the pipeline will be padded with IF.

**Automated Testing**

runSimulations.sh is a Bash script that automatically runs all benchmarks and outputs success/failure notification and the most vital statistics for a given interval of pipeline stages.

```
usr@sys:~$ ./runSimulations.sh
usr@sys:~$ ./runSimulations.sh all
usr@sys:~$ ./runSimulations.sh 5
usr@sys:~$ ./runSimulations.sh 9
```

## 1.4. Files

**src/bcolors.py** - Debug colour output formatting
**src/Checker.py** - Post-simulation correctness checking
**src/elf32instr.py** - Modelling elf32-bigmips instructions
**src/elf32parser.py** - elf32-bigmips to MIPS-Simulator format converter
**src/Instruction.py** - Modelling an iDEA instruction
**src/InstructionParser.py** - Parsing and preparing assembly code for simulation
**src/PipelineSimulator.py** - Main pipeline simulator file
**src/run-simulator.py** - Top-level program used to load input files and options and start the simulation
**dataMem.sim** - Data memory dump file
**preprocLog.sim** - Preprocessing and asm parsing log file
**simasm.sim** - The assembly code as passed to the simulator
**simrun.sim** - Main simulation log file containing debug information

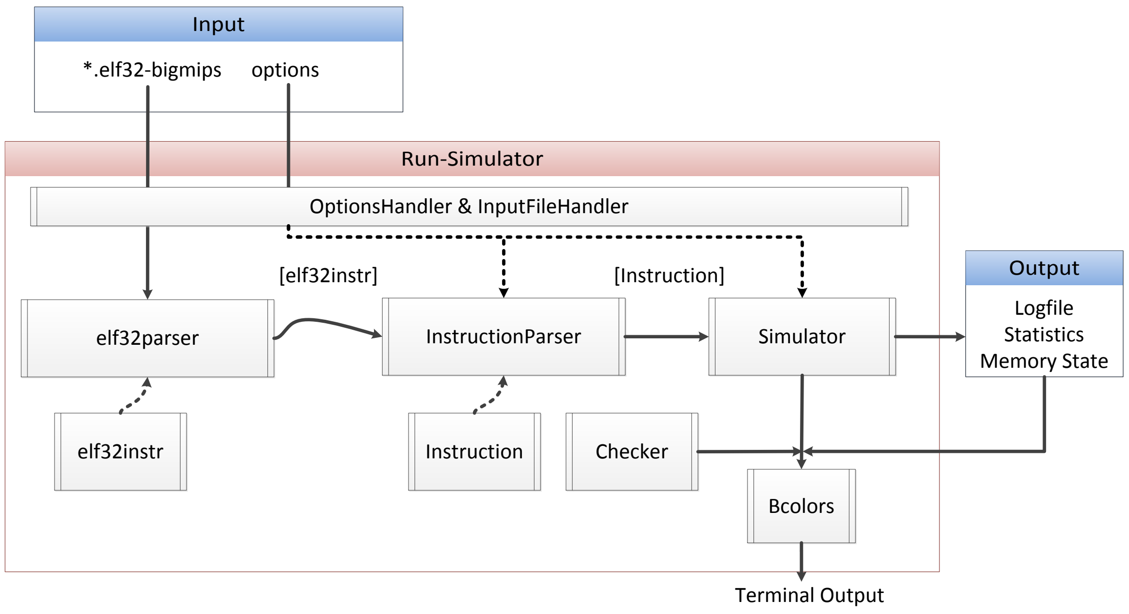

**Fig. 2**. Simulator class structure

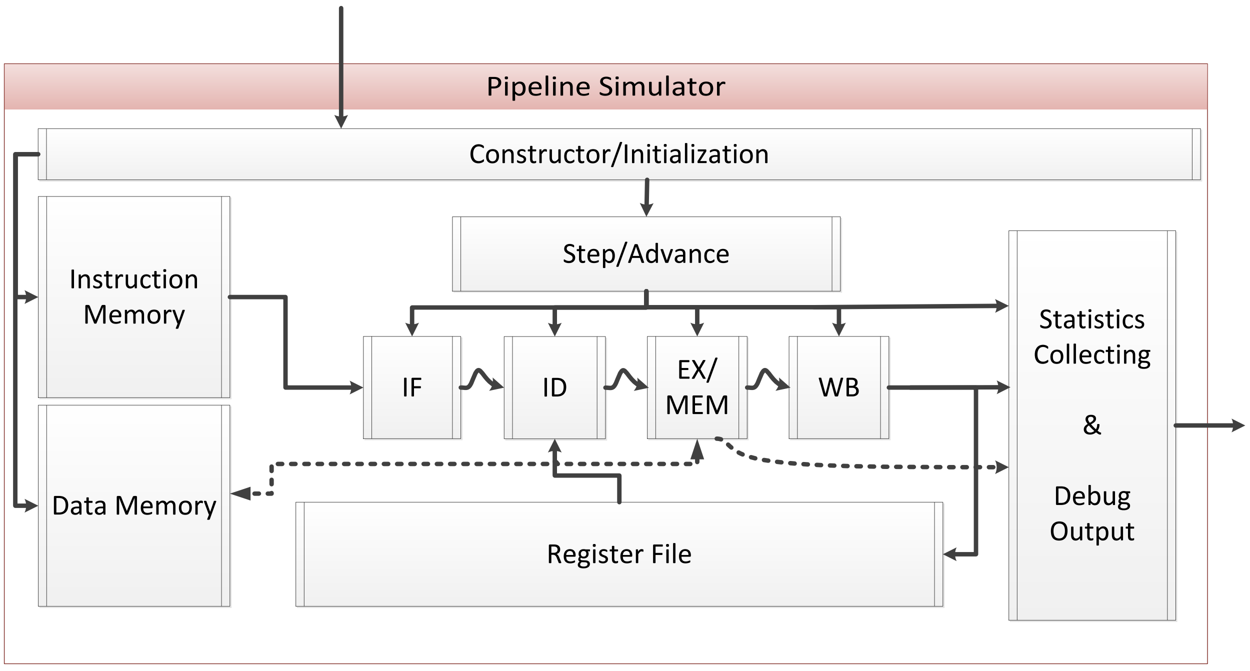

**Fig. 3**. Pipeline simulator internals