

Microblaze Simulator Manual

January 29, 2018

Contents

1	Architecture	2
1.1	Computing Core	2
1.2	Memory	3
1.3	Instruction Set	3
2	Simulator Usage	4
2.1	MicroBlaze Simulator	4
2.2	Tool 1: Xilinx .mem to Binary	5
2.3	Tool 2: ASCII to Binary	5
3	Editing the Simulator	6
3.1	Key information	6
3.2	Editing the computing core	7
3.3	Editing the instruction set	7
3.4	Editing the caches	7

Chapter 1

Architecture

1.1 Computing Core

The presented project is a software simulation of a 5-stage pipeline MicroBlaze processor. It has the following features:

- Instruction and data words of 32 bits;
- 32 general purpose registers of 32 bits (r0 to r31), with the value of register r0 being fixed at zero (all writes to register r0 return an error);
- 1 Special Purpose Register (rIMM) of 16 bits for immediate (constant) storage;
- 1 Machine Status Register (MSR), holding the carry flag (C) and the immediate flag (I) fields;
- a 32-bit memory address space, organized at the byte level, with all reads and writes being made in **Little Endian** format;
- No support for miss-aligned memory accesses;
- The operands are represented in 2's complement, by default;
- Includes 4 groups of instructions: arithmetic/logic/shift for integers, memory access, and control (delayed and not delayed);
- All writes to address 0xFFFFF0 are redirected to the standard output.

The 5-stage pipeline has the following stages:

1. Instruction Fetch and PC Increment (IF) - Fetches an instruction given the current PC and then increments it. Can stall the pipeline if the current PC is unknown or if the instruction data is still being fetched, due to memory latency.
2. Instruction Decode and Operand Fetch (ID) - Decodes the instruction obtained in the previous pipeline step and fetches the required operands. Can stall the pipeline if a required operand is still being computed or loaded.

3. Execute and Address Calculus (EX) - Executes the decoded instruction. The value computed during this stage can also be a future PC value or a memory address, depending on the instruction. Can stall the pipeline if there are no functional units available.
4. Memory Access and PC Calculus (MEM) - Accesses the memory if it's a memory instruction. If it is a control instruction, the next instruction's PC will be determined here. Can stall the pipeline if the data is still being moved, due to memory latency.
5. Write Back (WB) - Stores the results into the register file.

As it was just described, all but the Write Back pipeline stage can stall the processor, due to data, control or structural hazards. If the processor stalls on pipeline stage X, all stages up to (and including) X will repeat their operation in the following clock cycle. Furthermore, a virtual 'no operation' instruction is fed in to the following stage, to fill in the created void.

1.2 Memory

The memory system in this simulator contains two separate memories, one for data and other for instructions. Each memory contains several configurable parameters (listed below), which also include a customizable cache with no latency. When the cache is used, a write-through policy is employed and, if the cache is associative (i.e. more than one cache way), a FIFO replacement policy is applied.

When running programs compiled for the MicroBlaze, the instruction memory is copied to the data memory before its execution. This procedure is made to ensure a correct behavior, while keeping the memories separate.

List of configurable memory parameters:

- Main memory size;
- Main memory latency;
- Cache associativity;
- Cache size;
- Cache block size;

1.3 Instruction Set

The implemented instruction set is exactly as described in the laboratory assignment, containing 45 arithmetic instructions, 12 memory access instructions, 20 non-delayed branch instructions, and 21 delayed branch instructions.

There are two special instructions that the user must keep in mind. The instruction containing 32 zeroes, corresponding to 'ADD R0 R0 R0', is equivalent to a no operation (NOOP) instruction. Finally, the instruction 'BRI 0' is used to signal the end of the program.

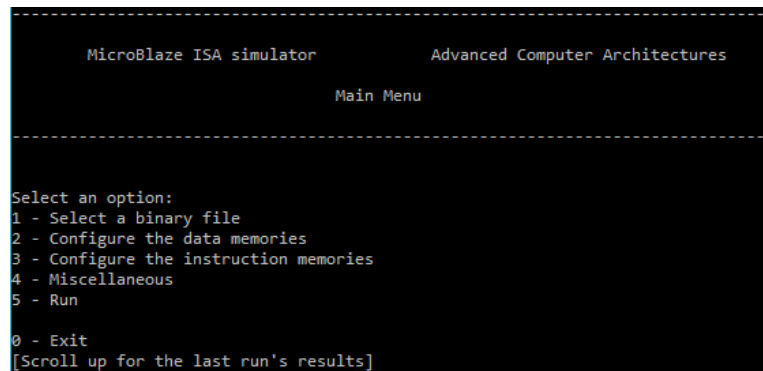
Chapter 2

Simulator Usage

This chapter describes the software usage, as well as its limitations. All the discussed programs have their source code available (written in C).

2.1 MicroBlaze Simulator

When starting the MicroBlaze Simulator, the Main Menu (Fig. 2.1) is shown to the user.



```
MicroBlaze ISA simulator          Advanced Computer Architectures
                                Main Menu
-----
Select an option:
1 - Select a binary file
2 - Configure the data memories
3 - Configure the instruction memories
4 - Miscellaneous
5 - Run
0 - Exit
[Scroll up for the last run's results]
```

Figure 2.1: MicroBlaze Simulator’s Main Menu

Before running the simulator (option 5), the user should configure the simulated device, selecting options 1 to 4. The selected configuration is stored in a file, so there is no need to reconfigure the device after closing the software.

Selecting option 1, the user is able to define the name of binary file that will be run by the simulator. The binary file must be placed in the same folder as the executable, otherwise the simulator will abort its execution. A 'hello_world' (with memory writes to 0xFFFFF0C0) binary file is also provided.

Options 2 and 3 open the configuration of the data and instruction memories, respectively. The customization is restricted to the parameters discussed within section 1.2. If the user decides to use zero cache ways, the executed binary behaves as if there is no cache.

Opening the miscellaneous option, the user is able to configure three display parameters. The first is a periodic print to the standard output, printing every N simulated cycles. The

second is a flag that, when enabled, prints every memory write to the console. Finally, the last parameter, which is also a flag, determines if the register file is written every clock cycle to an external file (inside the 'outputs' folder). This last feature is very useful for debugging purposes.

When the 'run' option is chosen, the simulator will execute the selected binary file. Figure 2.2 shows the console output for the 'hello_world' binary, with a periodic print every 1000 clock cycles and the memory write flag off. After the binary is finished, the data memory is written to an external file (also inside the 'outputs' folder).

```

**** STARTING TO RUN ****

WARNING: - 'MTS' instruction found, ignoring it
WARNING: - 'MTS' instruction found, ignoring it
Printing directly to std out - hex:48  ascii:H (current PC @ IF=214)
Printing directly to std out - hex:65  ascii:e (current PC @ IF=214)
Printing directly to std out - hex:6c  ascii:l (current PC @ IF=214)
[Running clock cycle = 1000]
Printing directly to std out - hex:6c  ascii:l (current PC @ IF=214)
Printing directly to std out - hex:6f  ascii:o (current PC @ IF=214)
Printing directly to std out - hex:2c  ascii:, (current PC @ IF=214)
Printing directly to std out - hex:20  ascii: (current PC @ IF=214)
Printing directly to std out - hex:77  ascii:w (current PC @ IF=214)
Printing directly to std out - hex:6f  ascii:o (current PC @ IF=214)
Printing directly to std out - hex:72  ascii:r (current PC @ IF=214)
Printing directly to std out - hex:6c  ascii:l (current PC @ IF=214)
Printing directly to std out - hex:64  ascii:d (current PC @ IF=214)
Printing directly to std out - hex:21  ascii:! (current PC @ IF=214)
Printing directly to std out - hex:a   ascii:
(current PC @ IF=214)
[Running clock cycle = 2000]

**** FINISHED ****

```

Figure 2.2: MicroBlaze Simulator's run example

2.2 Tool 1: Xilinx .mem to Binary

This simple tool converts a .mem file, from the Xilinx Microblaze compiler, into a pure binary file, ready to be read by the simulator. Since it's a very simple routine, the file names are selected by default: the .mem file must be named 'rom.mem' and the resulting output is named 'new_binary'. Within this folder, the .mem file for the 'hello_world' binary is also provided.

2.3 Tool 2: ASCII to Binary

Since debugging the simulator can become tricky, an ASCII to binary converter was also developed. This converter can convert both ASCII binary (zeroes and ones written in a .txt) and ASCII assembly into a pure binary file, ready to be read by the simulator. Both assembly and binary examples are provided and, to run the converter, the user just needs to follow the instructions within the console.

Chapter 3

Editing the Simulator

3.1 Key information

The MicroBlaze Simulator project consists of 4 source and 4 header files, listed below. Each file was generously filled with comments, so the simulator's flow should be clear after some code dissection.

- `main.c` - Controls the software flow;
- `general_includes.h` - Includes the standard libraries and defines the data structures;
- `menu.c/.h` - Contains all the menu and settings functions;
- `device.c/.h` - Contains the functions that setup the simulation device before the actual simulation;
- `instruction_set.c/.h` - Contains all the simulation functions.

As it can be read in the list above, most non-aesthetic changes to the simulator will be implemented in the '`instruction_set.c/.h`' files. Table 3.1 presents some of the simulator's behavior key functions, and thus prime candidates for further editing. The listed functions make extensive use of the data structures declared in the '`general_includes.h`' file, also shown in table 3.2.

Function Name	File	Short Description
<code>clock_tick</code>	<code>instruction_set.c</code>	Runs a clock cycle, calling multiple functions
<code>run_if</code>	<code>instruction_set.c</code>	Executes the IF pipeline stage
<code>run_id</code>	<code>instruction_set.c</code>	Executes the ID pipeline stage
<code>run_ex</code>	<code>instruction_set.c</code>	Executes the EX pipeline stage
<code>run_mem</code>	<code>instruction_set.c</code>	Executes the MEM pipeline stage
<code>run_wb</code>	<code>instruction_set.c</code>	Executes the WB pipeline stage
<code>update_auxiliary_variables</code>	<code>instruction_set.c</code>	After executing the pipeline, updates the processors' internal registers
<code>check_memory_parameters</code>	<code>menu.c</code>	Controls the accepted memory parameters
<code>*_cache_*</code>	<code>instruction_set.c</code>	Functions with 'cache' in their names control the caches' behavior.

Table 3.1: Key functions for further editing

Struct Name	Short Description
Registers	The MicroBlaze register file
Memories	Struct with all the memories and their parameters
Caches	Struct with the cache information (encapsulated by the memories struct)
Auxiliary_variables	Data to be kept in the processor between clock cycles

Table 3.2: Main data structures

3.2 Editing the computing core

The existing simulator's execution can be improved in multiple ways, namely using branch prediction techniques, introducing multiple execution units or splitting the pipeline into more stages. Each one of these examples would require a thorough analysis of some of the functions listed in table 3.1, followed by a careful rework within such functions.

Regardless of the desired changes, it is advised to use the 'Auxiliary_variables' structure to store all the information that should be kept between clock cycles, adding more variables (or even new structures) as necessary. Keep in mind that at the end of each clock cycle, the 'update_auxiliary_variables' is called, moving some variables between pipeline stages.

3.3 Editing the instruction set

If the user wishes to edit the existing instruction set, by adding, removing or changing instructions, the first concern should be to identify the instruction type (arithmetic, memory access or control). Each instruction type might require different modifications at specific pipeline stages, as listed below:

- Arithmetic - ID [access its operands], EX [arithmetic operation], WB [store result];
- Memory access - ID [access its operands], EX [address calculus], MEM [access the data memory], WB [store result if it's a load];
- Control - ID [access its operands], EX [address calculus], MEM [update PC], WB [may store the current PC]

Introducing a new instruction type requires a complete analysis of the current pipeline. Nevertheless, unless two or more cycles of delayed execution are desired, the IF stage should require no update.

3.4 Editing the caches

When editing the caches, only two stages of the pipeline are involved. The IF and the MEM stages interact with the instruction and data memories, respectively.

The functions for both caches were kept separated, even though they are roughly the same. Controlling the instruction cache is slightly easier, though: it only requires reading operations, 4 bytes at a time.

To change the block replacement policy, the function 'load_xxxx_cache_block' (xxxx being data or inst) must be updated. If it is desired to test a new write policy for the data cache, the flow for memory instructions within 'run_mem' must be rewritten.