University of BRISTOL

COMSM0053 Advanced Computer Architecture

TkArch – A Simulator for a RISC-y Architecture

February 27, 2023

Student Name	Taharka Okai (zp19374)
Student ID	1815919

1 Introduction

This architecture is a RISC-based architecture that is designed to be simple and yet flexible. Being a RISC-based architecture, a software oriented approach is taken, with the architecture being designed to be easy to implement in software. This will, in theory, make benchmarking and testing more simple to execute, as it is easy to determine what each instruction is doing. Heavy inspiration was taken from MIPS and ARM, being used as a basis for this architecture.

2 Architecture

The architecture is separated into three distinct parts - the instruction set defining the hardware interface, the microarchitecture defining the hardware implementation, and the simulator defining the software interface.

In version 0.0.1, the instruction set is very simple, consisting of only a few instructions in order to demonstrate the fetch-decode-execute cycle. It is non-scalar and non-pipelined, meaning that there is only one instruction in the pipeline at any one time. In future versions, the instruction set will be expanded to include more instructions, and the architecture will be expanded to include a scalar and pipelined implementation.

2.1 Instruction Set

The instruction set is defined as a table below. The instruction set is designed to be simple and flexible. Omitted from the table are the two extra fields that alter the behaviour of these instructions. These are the addressing mode, and the condition code.

The addressing mode is used to determine how the address is calculated. In version 0.0.1, the addressing mode is always direct, meaning that the address is calculated as the value of the operand. In future versions, the addressing mode will be used to determine how the address is calculated, and will be used to implement indirect addressing.

Instruction	Opcode	Operands	Description	Cycles
`HLT`	0x00	None	Stops the clock	1
Retrieval				
`LDR`	0x01	`Rd, X`	Load from memory	?
Storage				
`STR`	0x02	`Rd, X`	Store to memory	?
`MOV`	0x03	`Rd, X`	Move value to register	?
Arithmetic/Logic				
`ADD`	0x03	`Rd, Rs, X`	Add two registers	?
`SUB`	0x04	`Rd, Rs, X`	Subtract two registers	?
`AND`	0x05	`Rd, Rs, X`	Bitwise AND	?
`OR`	0x06	`Rd, Rs, X`	Bitwise OR	?
`NOT`	0x07	`Rd, Rs`	Bitwise NOT	?
Control				
`CMP`	0x08	`Rd, X`	Compare and set status	?
`JMP`	0x09	`#`	Jump to address	?
`B`	0x0A	`#`	Branch to address	?

Table 1: Instruction Set (v0.0.1)

In the Table 1, the symbol `X` represents an operand that can be either an address to a register or an immediate value, and the symbol `#` is always an immediate value.

2.2 Microarchitecture

In version 0.0.1, the microarchitecture is very simple, consisting of only a few components in order to demonstrate the fetch-decode-execute cycle. It implements the load-store paradigm ubiquitous amongst RISC-based architectures. Components in the computation sequence are as follows:

– The `PC` register, which holds the address of the next instruction to be executed.

– The `IR` register, which holds the instruction currently being executed.

- The `MAR` register, which holds the address of the memory location currently being accessed.
- The `MBR` register, which holds the data currently being accessed from memory.

The architecture has 8 general purpose registers, numbers from 0 to 7. These are used to store data, and are used as operands in instructions. In version 0.0.1, there are not any stack operations, and so ‘`PSH`’ and ‘`POP`’ are left unimplemented. The architecture has a small 4-bit address bus to start with, allowing 16 addressable registers. Altogether, with 12 instructions (4 bits), 2 possible addressing modes and 8 general purpose registers (11 bits)[1], this gives a minimum instruction size of 15 bits, which will be rounded to 16 for consistency and convention.

Opcode	Op 1	Op 2 Type	Op 2	Op 3 Type	Op 3
`0XXXX`	`XXX`	`X`	`XXX`	`X`	`XXX`

Table 2: Instruction format as a result of microarchitecture

2.2.1 Fetch

The fetch stage is responsible for fetching the next instruction from memory. It is responsible for incrementing the `PC` register, and for loading the `IR` register with the instruction at the address stored in the `PC` register. This happens in the following sequence:

$$\begin{aligned} \texttt{MAR} &\leftarrow \texttt{PC} \\ \texttt{MBR} &\leftarrow \texttt{Memory}[\texttt{MAR}] \\ \texttt{IR} &\leftarrow \texttt{MBR} \\ \texttt{PC} &\leftarrow \texttt{PC} + 1 \end{aligned}$$

Figure 1: Fetch

2.2.2 Decode

The decode stage is responsible for decoding the instruction currently in the `IR` register. This is as simple as reading the most significant bits of the instruction, and using them to determine which instruction is being executed. In simulation, this is represented by a switch statement, but may actually cost more cycles in hardware.

2.2.3 Execute

The execute stage is responsible for executing the instruction currently in the `IR` register. This is as simple as reading the least significant bits of the instruction, and using them to determine which operation is being executed.

In version 0.0.1, the execute state is not yet complete.

3 Simulator

The simulator is a binary executable written in C++. It is focused on providing an in-depth view into individual components of the architecture, and is designed to be used in conjunction with a debugger.

The program is built using the CMake build tool, using the Ninja Generator (although Unix Makefiles may also be used). To build the simulator, either run the provided `build.sh` script, or run the following commands:

[1]The first operand is always a destination register, so the additional 1 bit for the addressing mode is not required. With a maximum of 3 operands, this yields a total of $2 \times (3+1) + 3 = 11$ bits.

```
$ # This will contain the src, include, CMakeLists.txt, etc.
$ cd aca
$ # Setting CONFIG will affect performance of the simulator.
$ # 'Release' is recommended for performance, 'Debug' for
$ # debugging.
$ cmake . \
  --build build \
  --config <CONFIG> \
  -G Ninja \
  --target Simulator
$ # The simulator will be located in the bin directory.
$ cd bin ; ./simulator
```

3.1 Usage

The simulator is intended to be run as follows:

```
$ ./simulator [options]

Options:
  -h, --help            show this help message and exit
  -i, --interactive     Run in interactive mode
  -f FILE, --file=FILE  Load program from file
```

In interactive mode, the simulator will prompt the user for input, and will execute the program step-by-step. In non-interactive mode, the simulator will execute the program until it halts.

If a file is not provided, the simulator will load a default program (TODO).

3.2 Benchmarks

Sum This is a test benchmark to demonstrate fetching, decoding, executing and storing data. It adds the numbers 1 to 10, and stores the result in register 0.

```
; Sum
; MOV R0, #0   ; R0 will hold the result
; MOV R1, #0   ; R1 will be added to R0
; Loop
CMP R1, #10    ; Compare R1 to 10
BEQ END        ; Branch to the end of the program if R1 > 10
ADD R1, R1, #1 ; Increment R1
ADD R0, R0, R1 ; Add R1 to R0
JMP LOOP       ; Jump back to the start of the loop
; End
HALT           ; Halt the program
;
; The end result should be R0 = 55, R1 = 11
```

Vector Sum TBD.

```
; Vector Sum
; TBD
```

A Appendix

A.1 Changelog

Version	Date	Description
v0.0.1	20/02/2023	– Fetch-Decode Implemented – Simple instruction set designed

Table 3: Changelog