Implementation of Educational Instruction Set Architectures

Bohdan Opyr
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
opyr.pn@ucu.edu.ua

Petro Mozil
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
mozil.pn@ucu.edu.ua

Radomyr Husiev
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
husiev.pn@ucu.edu.ua

Roman Zaletsky
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
zaletsky.pn@ucu.edu.ua

***Abstract*—This document contains a description of the implementation of the four instruction set architectures, as well as the development environment created for facilitating the learning of the differences of these architectures. This includes an assembler, linker and a disassebmler for these ISAs, an emulator and a the processor implementations on a FPGA devboard, specifically DE10-Nano rev. C, alongside a linux image for facilitating the writing and uploading the compiled code.**

The linux image is debian 10 buster with linux 6.1.38 and the assembler is customizeable – one can write a YAML file to describe practicallly any ISA.

***Index Terms*—SystemVerilog, Field Programmable Gate Array, Instruction Set Architecture, Assembler, Linker, Emulator**

I. Introduction

An ISA (instruction sert architecture) is an abstract model of a computer. In general, an ISA defines the supported instructions, data types, registers, the hardware support for managing main memory, fundamental features (such as the memory consistency, addressing modes, virtual memory), and the input/output model of a family of implementations of the ISA. [7]

Modern processors are complicated. That makes them unsuitable for educational purposes, and especially for teaching the difference between different types of ISA's, since even the most noticeable differences between different types of processors are hidden behind the afforementioned complexity. Even RISC-V is a very extensive and complex ISA, which has a lot of concepts that are fairly difficult to understand at a first glance.

For this reason, the four instruction set architectures we implemented were created. [1] The ISAs themselves are not meant to be optimal or productive – they were created to show the basics of instruction decoding and the difference between a stack, accumulator, RISC and CISC architectures.

Such an educational project would also greatly benefit from an assembler / disassebmler and an emulator, apart from the processor itself. Our objective is to create not only a processor, but also an environment that is easy to develop in. The assembler should return clear errors if there are any, and so should the emulator. It should be easy to upload the programs to the processor, and there sould be a way to execute instructions step-by-step.

We created the processors that implement the four ISAs and an environment to develop programs for them / test these programs.

The code and documentation for the project can be found on Github at monistode [9]

II. ISA description

The four ISAs are as such:

- A stack processor ISA.
- An accumulator processor ISA.
- A RISC ISA.
- A CISC ISA.

The four architectures have only the most basic commands:

- Arithmetic (add, compare, multiply, subtract and divide)
- Basic bit manipulation opeations (bitwise and, or, xor and not, and the test command)
- Simple load / store commands
- Very simple IO: UART connection and multiple GPIO pins

Each architecture has its own unique features, e. g. the CISC one has a couple of SIMD instructions.

A. Stack ISA [4]

The stack ISA is a harvard architecture. It has an address space of size 65536 bytes, the byte is 6 bits, and the machine word is 2 bytes. The addressing modes are:

- Register (from the top of the register stack)
- Immediate
- Memory location

The instruction size is 6 to 18 bits (2 byte immediate). The first bit signifies whether there is an immediate present.

There are two stacks:

- Register stack - 16 bits per item, stores the data for operations. Starts at the 256th byte, grows upward.
- Memory stack - 16 bits per item, stores the function return addresses. Starts at the 1024th byre, grows downward.

Stack underflow nor collisions with the register stack not handled

There are four registers:

- PC - 16 bits - program counter, points to a 6-bit byte in the text section. Is always the next instruction to be executed, similar to x86
- FR - 4-16 bits - flag register with the least significant bits representing CF ZF OF SF; (up to 16 bits because it has to fit on the stack)
- TOS - 15 or 16 bits - initial value 256 (assuming 16 bits) - points to two bytes in the data section - the top of the register stack; grows upward (a push increments)
- SP - 15 or 16 bits - initial value 1024 (assuming 16 bits) - points to two bytes in the data section - the top of the memory stack; grows downward (a push subtracts)

B. Accumulator ISA [6]

The accumulator ISA is a von Neumann architecture. It has an address space of size 65536 bytes, the byte is 8 bits, and the machine word is 16 bits. The addressing modes are:

- Register (from the top of the register stack)
- Immediate
- Memory location

The instruction size is 8 to 24 bits (the immediate is 2 bits). The first bit signifies whether there is an immediate present.

There is one stack - the memory stack. It starts at 1024 byte and grows downwards.

There are four registers:

- PC - 16 bits - program counter, points to a 6-bit byte in the text section. Is always the next instruction to be executed, similar to x86
- FR - 16 bits - flag register with the least significant bits representing CF ZF OF SF; (up to 16 bits because it has to fit on the stack)
- SP - 16 bits - initial value 1024 (assuming 16 bits) - points to two bytes in the data section - the top of the memory stack; grows downward (a push subtracts)
- IR1 - 16 bits - index register (used for indexing in arrays)
- IR2 - 16 bits - index register (used for indexing in arrays)
- ACC - 16 bits - the accumulator

IR1 and IR2 can only be incremented / decremented and stored/read from.

C. RISC ISA

The RISC ISA is a von Neumann architecture. It has an address space of size 65536 bytes, the byte is 8 bits, and the machine word is 16 bits. The addressing modes are:

- Register (from the top of the register stack)
- Immediate
- Memory location

The instruction size is 8 to 16 bits. The first bit signifies whether there is an immediate present.

There is one stack - the memory stack. It starts at 1024 byte and grows downwards.

There are four registers:

- PC - 16 bits - program counter, points to a 6-bit byte in the text section. Is always the next instruction to be executed, similar to x86
- FR - 16 bits - flag register with the least significant bits representing CF ZF OF SF; (up to 16 bits because it has to fit on the stack)
- SP - 16 bits - initial value 1024 (assuming 16 bits) - points to two bytes in the data section - the top of the memory stack; grows downward (a push subtracts)
- R00, R01, R02, R03 - 16 bits each, with L and H bits each (they, however, cannot be accesed easily)

D. CISC ISA [5]

The CISC ISA is a von Neumann architecture. It has an address space of size 65536 bytes, the byte is 8 bits, and the machine word is 16 bits. The addressing modes are:

- Register (from the top of the register stack)
- Immediate
- Memory location

The instruction size is 8 to 64 bits. The first bit signifies whether there is an immediate present.

There is one stack - the memory stack. It starts at 1024 byte and grows downwards.

There are four registers:

- PC - 16 bits - program counter, points to a 6-bit byte in the text section. Is always the next instruction to be executed, similar to x86
- FR - 16 bits - flag register with the least significant bits representing CF ZF OF SF; (up to 16 bits because it has to fit on the stack)
- SP - 16 bits - initial value 1024 (assuming 16 bits) - points to two bytes in the data section - the top of the memory stack; grows downward (a push subtracts)
- R00, R01, R02, R03 - 16 bits each, with L and H bits each (they, however, cannot be accesed easily)

There are SIMD instructions - they store the data into R00 through R03 and perform them as such:

- R00L — R02L
- R00H — R02H
- R01L — R03L
- R01H — R03H

There is SIMD addition, subtraction division and multiplication, and also load and store into. The load gets 8 bytes in a row, and the store stores 8 bytes in a row.

III. Processor implementation

The processors runs on FPGA, specififcally the DE10-Nano devboard. The devboard also has a linux image running in parallel to the processor. The FPGA and the linux image share 512 megabytes of RAM. The entry point for the program is 0. The entry point from linux is at address 0x20000000. The processor cannot access any of the 512 megabytes before 0x20000000, and it can only access megabytes from 512 to 515.

We created the processors in SystemVerilog, and made them fairly easy to update and customize.

For memory we used Avalom memory map interface [8] - it is an interface we used for simplifying RAM and UART IO.

We also used the quartus IDE for development.

Every processor has UART IO on such ports:

- RX - GPIO0 pin 0
- TX - GPIO0 pin 1
- GND - any ground on the DE10-Nano

The GPIO pins on the CPU are GPIO0 pins 2 through 36.

The debug mode (setp-by-step execution) mode is turned on by SW3. The stepping is done by pressing KEY1, and unhalting is done by pressing KEY1.

When SW1 is on, the execution is stopped and pressing KEY0 resets the processor.

IV. Assembler, Disassembler and Linker [2]

Since performance isn't a key concern, but instead simplicity, the assembler and linker are written in python. The assembler is a single pass assembler based on a simple recursive descent parser. The linker takes in a list of object files and outputs a single executable file, after applying the necessary relocations. An object file can hold text and data sections, the symbol table and the relocation table. The executable holds segments, each of which can be marked as executable or non-executable, readable or non-readable, writable or non-writable and special or non-special. It is placed into a specific address in the address space. The disassembler simply crawls each text section command by command, returning the disassembled code. The linker can also output the executable as two raw files, each of which is a representation of the raw address space of the executable.

V. Program upload

The uploading of the program is simply done by writing 65536 bytes of the given program to location 0x20000000 on linux. This pattern in widely used by SoC FPGA developers: We open "/dev/mem" and mmap 64Kb of memory at address 0x20000000. After that we write to that memory and unmap it.

VI. Conclusion

We designed a workflow for writing, assembling and executing of programs written for the four educational ISAs. The implementations are open-source and the customization of them is encouraged.

There are four repositories containing the quartus projects and three repos containing the assembler, linker and emulator for the project.

We also updated the documentation and the accumulator ISA.

References

[1] Oleg Farenyuk, Darian Omelkina. Code for the original assembler and simulator. https://github.com/ucu-computer-science/Hardware-Simulator-and-Assembler/tree/master

[2] Monistode. Code for the assembler and linker. https://github.com/monistode/binutils

[3] Monistode. Code for ISA documentations https://github.com/monistode/ISA-docs

[4] Monistode. Code for the stack ISA https://github.com/monistode/ISA_stack

[5] Monistode. Code for the CISC ISA https://github.com/monistode/ISA_cisc

[6] Monistode. Code for the accumulator ISA https://github.com/monistode/ISA_accum

[7] Wikipedia. Definition of ISA https://en.wikipedia.org/wiki/Instruction_set_architecture

[8] Avalon interface specification https://cdrdv2-public.intel.com/667068/mnl_avalon_spec-683091-667068.pdf

[9] The monistode organization https://github.com/monistode