Leo De Silva

A Level Computer Science

DESIGNING & MAKING THE SOFTWARE SUITE

for a proprietary machine code specification.

2024, St Albans School

Contents

1 Analysis

1.1 Problem Defenition

The goal of this project is to design the hardware specification for a custom 16-bit single cycle CPU, and develop the suite of tools required to simulate such a processor, including an Emulator (1.2.2), Assembler (1.2.3), and Compiler (1.2.4). The project will detail the abstract design of the computer's Instruction Set Architecture (ISA) (1.2.1) and its implementation in hardware, considering the internal registers, system clock, main memory, and fetch execute cycle.

The project will compose three primary parts, an emulator capable of loading machine code 'catridges' and simulating the hardware, memory, peripherals and registers detailed by the ISA in order to execute programs with correct clock timing and behaviour. An assembler to translate programs written in an assembly specification into binary machine code (the assembler will handle higher level conveniences such as relative addressing through labels). And finally a compiler - to translate a higher level programming langauge into machine code. This will require compiler optimisations with relation to the produced object code; complex data structures such as arrays, objects and strings; conditional and interative expressions; and functions and procedures. The suite required to simulate such a computer should be capeable of writing and compiling complex programs such as pong or tetris, and emulating them with hardware correct timings - dealing with peripherals such as a keyboard or speaker.

1.2 Background to the Problem Area

This project will explore lower level systems software and look in detail at the fundemental architecture of modern computing systems and how they are designed from the ground up.

1.2.1 Instruction Set Architecture

The ISA acts as an interface between the hardware and software of a computing system, and contains crucial information regarding the capabilities of a processor, including: a functional defenition of storage locations (e.g. registers and memories) as well as a description of all instructions and operations supported.

An ISA can be classified according to its architectural comlpexity into a Complex Instruction Set Computer (CISC), or a Reduced Instruction Set Computer (RISC). A CISC processor implements a wide variety of specialized instructions in hardware (e.g. floating point arithmetic or transferring multiple registers to or from the stack), minimising the number of instructions per program at the cost of a more complex design, higher power consumption and slower execution as each instruction requires more processor cycles to complete. Joshi (2024) Thus optimisations in performance occur on the hardware level. This is historically the most common branch of processor and often results in processors with large instruction sets, for example the Intel x86's 1503 defined instructions Giesen (2016). A RISC processor however aims to simplify hardware using an instruction set consisting of a few basic instructions to load, evaluate and store data - instead placing optimisations on the software side unlike a CISC processor. This has the side effect of increased memory usage required to store additional instructions needed to perform complex tasks.

1.2.2 Emulator

An emulator is a software program that allows one computer to imitate another - and by simulating the hardware of the other – execute machine code programs written for a processor other than itself. Emulators tend to consist of three modules, a CPU emulator, memory subsystem module, and I/O device emulators RetroReversing (2022). The simplest form of CPU emulator is an interpreter - wherein the emulator steps sequentially through each machine code instruction, and carries out the fetch-decode-execute cycle for each, modifying the internal state of the simulated processor in the same manner as the instruction would affect the physical hardware - this can be achieved by representing internal flags and registers as variables. The Memory Subsystem can be implemented as a one dimensional array of bytes - with features such as memory mapped I/O implemented by associating regions of memory to peripherals and subsystems, e.g. Video Random Access Memory (VRAM), the stack, and the heap.

1.2.3 Assembler

An assembler is a program that translates assembly language (a low level programming language that uses mneumonics to directly represent machine code instructions) into object code that can be executed by the processor. There are 2 types of assembler design, single-pass and multi-pass Toppr 2019. A single-pass assembler scans the source code only once to translate it into machine code, and outputs the result directly. This is a simpler type of assembler, and is more efficient with regard to translation speed. However, it requires all symbols used within the program (variables, labels, etc...) to be declared before they are used - else the program will crash.

A multi-pass assembler however scans the source code multiple times, on the first pass defining a symbol table and opcode table Toppr 2019, processes pseudo instructions (compound macro instructions that are substituted during assembly with a list of fundemental instructions), and maintaining a location counter to store the memory address of each instruction as it would be compiled.

There are also certain higher level abstractions a high-level assembler can translate such as `IF/THEN/ELSE/WHILE` statements and certain higher level data types – however this results in a complex assembler and lengthy compilation times - as well as a blurred line between the role of high level and low level languages.

1.2.4 Compiler

A compiler is a program that translates high level program source code into a set of machine language instructions. Some compilers translate source code into an intermediate assembly language before using an assembler to produce the machine code instructions, whereas others compile into machine code directly.

1.3 Programming Language

1.4 Existing Systems

1.4.1 The Hack Computer

1.4.2 University of Washington MIPS Computer

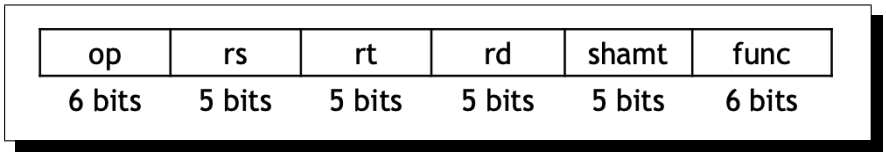
The following system is a 16 bit MISC (Minimal Instruction Set Computer) processor designed by the University of Washington for a series of lectures as part of their computer science course Washington (2018). A MISC processsor is a subclass of the RISC processor and involves minimising the number of instructions implemented in hardware, resulting in far simpler hardware designs - where a RISC processor may have 30-70 instructions, a MISC processor may have merely 10-20 consisting of arithmetic, branching, loading and storing instructions. Engineering (2015).

A MIPS (Microprocessor without Interlocked Pipelined Stages) processor such as this does not overlap the execution of several instructions (pipelining), thus neglecting the potential performance gains in favor of a simpler architecture. This processor is a single-cycle implementation meaning all instructions take exactly one cycle to complete, this is achieved using a Harvard architecture in place of Von Neuman wherein instructions are stored in a seperate Read Only Memory (ROM) to data, thus both can be fetched within the same processor cycle.

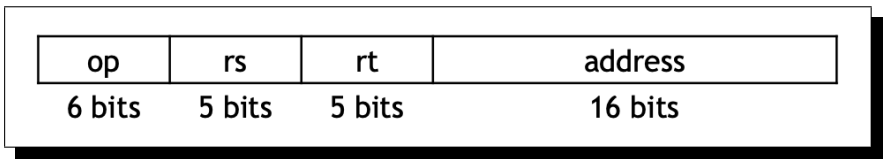
The processor supports the following instructions:

1. Arithmetic `add, sub, and, or, slt (set if less than)`
2. Data Transfer `lw (load word), sw (store word)`
3. Control `beq (branch if equal to)`

Register-to-Register arithmetic instructions use the R-type encoding for their machine code representation, where `op` is the opcode of the instruction with `func` representing the particular arithmetic operation. `rs, rt`, and `rd` are the source and destination registers. This computer operates on an ALU with a 3 bit control signal supporting 5 operations that directly correspond to the `func` portion of an R type instructions binary encoding.

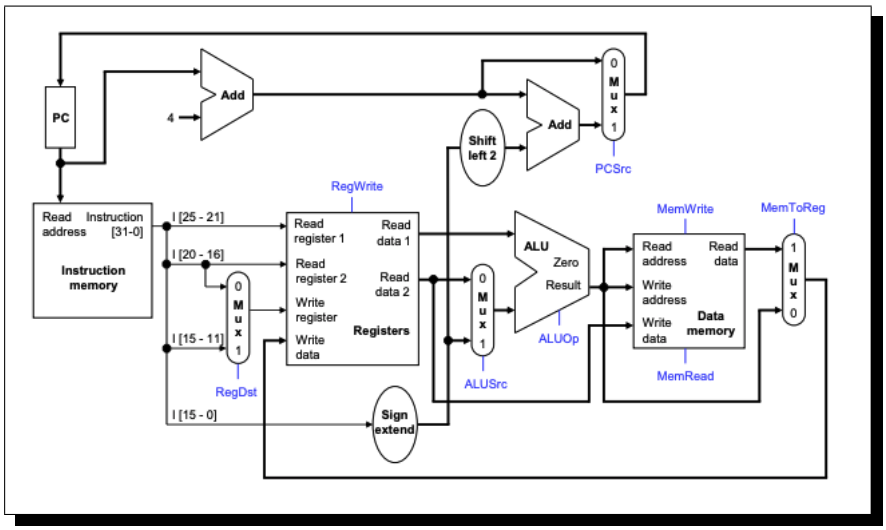


The I type encoding is the second means for which instructions can be represented, and includes the data transfer and control instructions `lw, sw`, and `beq` specified above. `address` is a signed 16 bit constant. `rt` is the destination for `lw` and source for `beq` and `sw`. `rs` is the base register for the `lw` and `sw` instructions (added to the sign extended constant `address` to get a data memory address) Washington (2018). In this processor design, for a `beq` instruction, the `address` field specifies not a memory address, rather a signed offset from which to jump from the current PC position when executing the branch instruction.



Below is the full datapath specification for the computer, with the Instruction Memory (ROM) on the left, connected to the PC in order to address instructions. Those instructions are in turn passed through the control unit and decoded, with the opcode specifying whether an I or R type instruction is being processed and accordingly what hardware should be used to interpret and execute the instruction. This dictates the calculation (if any) that is to be performed in the ALU - the output of which is stored in a seperate data memory.

Since instructions are stored in a seperate ROM, the address of the first instruction will always begin at 0 - this simplifies the calculation of offsets and labels in the assembler – since the assumption that the first instruction begins at address 0 will always hold true. However, branch instructions are handled unusually in this computer - instead of specifying the jump address, the signed offset from the current instruction is specified instead. This has the effect of making compilation easier as branch addresses do not need to be calculated by the assembler, however renders specific jumps to memory addresses (such as the location of an interrupt service routine or bootloader) difficult.



The architecture described above has some notable advantages, firstly, its Harvard architecture allows the processor to operate each instruction in a single cycle – both improving performance and simplifying the design of the emulator as microinstruction cycles do not need to be simulated to accurately simulate the hardware. Secondly, by dividing the computer architecture into 2 distinct I and R type instructions, you can reduce redundant information – and thus the bits required to store machine code instructions and programs.

However, this simple architecture results in many inconveniences when writing assembly code - due to the limited instruction set, simple tasks take comparatively more instructions meaning programs are longer and more tedious to write - as well utilising more memory due to the limited number of specialized instructions who's functionality must be implemented using handwritten subroutines such as binary shifts, stack operations, or interrupt handling.

The takeaways of this system for my project include:

1. The encoding of instructions into meaningful machine code that directly relates to the hardware of the computer - for instance R type opcodes representing the control bits of the ALU, this makes decoding instructions more efficient - especially when implmented in hardware.

2. Secondly, the behaviour of hardware (registers, memories, flags) and the relationships between components during a single-cycle Harvard fetch-execute cycle that will have to be simulated when designing an emulator.

3. I will also expand the instruction set further than the MISC specification used in this processor to include other common instructions, and keep the memory-register seperation wherin operations are performed on register values, with 2 instructions `lw`, `sw` used for reading and writing to memory.

4. I will also change the branch instruction to operate on absolute addresses rather than signed offsets.

1.5 Prototyping

1.6 Client Proposal

1.6.1 Client Interview

1.7 Objectives

2 Design

3 Technical Solution

4 Testing

5 Evaluation

References

Engineering, DAK (2015). *Minimal Instruction Set Processor (F-4 MISC)*. URL: http://www.dakeng.com/misc.html. (accessed: 14.04.2024).

Giesen, Fabian (2016). *How many x86 instructions are there?* URL: https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/. (accessed: 14.04.2024).

Joshi, Vibha (2024). *RISC and CISC in Computer Organization*. URL: https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/. (accessed: 14.04.2024).

RetroReversing (2022). *How do Emulators work? A Deep-dive into emulator design*. URL: https://www.retroreversing.com/how-emulators-work. (accessed: 14.04.2024).

Toppr (2019). *Assembler*. URL: https://www.toppr.com/guides/computer-science/computer-fundamentals/system-software/assembler/. (accessed: 16.04.2024).

Washington, University of (2018). *A single-cycle MIPS processor*. URL: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec07.pdf. (accessed: 14.04.2024).