1 Introduction

In programming a particular microprocessor, the size and formatting of assembly language instructions used are defined by the Instruction Set Architecture (ISA). This is a description of what operations can be performed. When choosing the encoding, a number of factors need to be considered. Including, but not limited to, instruction size, completeness and regularity. This report focusses on the principles of orthogonality and code density.

2 Considerations

2.1 Orthogonality

Each instruction within an architecture is defined as a series of binary bits defining the operation to be performed, the data to use, and a number of instruction-specific parameters. The concept of orthogonality is based on separating these bits into multiple fields with each defining difference aspects of the instruction. The field values are then able to vary independently. [1]

Using the formatting defined as in Table 1, any instruction can be entered into the 'Opcode' field with any choice of addressing mode, source or destination. If there is no dependency between different fields the set is completely orthogonal. Example architectures include: ARM11 and VAX-11. [1] If there is some dependency, such as not having all addressing modes available for an instruction, the set can be perceived as near-orthogonal, such as in the DEC PDP-11 or Motorola 68k. [1]

Opcode	Addressing Mode	Source 1	Source 2	Destination

Table 1: Example Instruction Set Format

2.2 Code Density

The principle of code density relates to how much can be performed using a fixed amount of instructions or available storage. In performing a given task, using one line of assembly code which translates to a small number of bytes in machine code will have a high density. If the same task requires multiple lines of assembly on a different ISA, which translates to at least as many bytes of machine code per instruction, it will have a much lower code density.

ISA's designed without emphasis on simple operations are known as Complex Instruction Set Computers (CISC). These typically have a very large set of instructions and can range from a simple arithmetic `ADD` to a `REPE` instruction as is found in the Intel 8086. [2] This repeats the following instruction while a flag equals 1, until a maximum is reached.

There are a number of factors which effect the density of code within a program. These include minimum instruction length, number of registers and the presence of a 'zero' register among others. A recent experiment shows the effect of different architectural features on the binary file size of a particular program [3]. The results of calculating the correlation between feature and code size can be seen in Table 2. Each type of architecture has its merits, whether that be when hand-coding or using a compiler. When hand-coding, CISC typically produces more dense code since less instructions are needed. When compiled, it is not as dense since a very complicated, specialized compiler would be needed to use the more powerful instructions. However, RISC produces less efficient coding when done by hand, but compilers can produce similar results without the added complexity.

Correlation	**Parameter**
0.9381	Minimum possible instruction length
0.9116	Number of integer registers
0.7823	Virtual address of first instruction
0.6607	Architecture has zero register
0.6159	Bit-width
0.4962	Number of operands in each instruction
0.3129	Year of introduction
-0.0021	Branch delay slot
-0.0809	Machine is big-endian
-0.2121	Auto-incrementing addressing scheme
-0.2521	Hardware status flags
-0.3653	Unaligned load/store available
-0.3854	Hardware divide

Table 2: Correlation of architecture features to binary file size [3]

2.2.1 Kolmogorov Complexity

Producing an optimal code density is within a branch of information theory known as Kolmogorov Complexity. This is a mathematical measure of the complexity of an object based upon the computational resources needed to describe it [4]. The complexity of any given object is determined by the shortest possible method of describing it using a given language. This could be alphabetic characters describing a string [4]. The length of this shortest

description is dependent on the choice of language used, and restricted by the Invariance Theorem. Which concludes that the optimal description is based upon the description of another language and an object within that language. This relates to the Opcode used for a particular instruction and the operands used. This is easily relatable to an encoding when a good orthogonality scheme is used.

The use of mathematical theory is impractical in simple designs since KC is an "incomputable function"[4]. While association of a particular encoding scheme to a binary value is more suitable for detailed mathematical proofs.

3 Case Studies

3.1 VAX-11

The VAX-11 is a CISC-type microprocessor from 1977 and was produced by Digital Equipment Corporation [5]. It was the first commercial 32-bit system which also performed at 1 MIPS (Million Instructions Per Second). It was adopted by many institutions and universities [5]. The design objectives were focussed on simple compiling, via direct high level language conversion, and minimising code size. This lead to powerful addressing modes, instructions and efficient encoding [6]. The advanced addressing modes included: register, auto-decrement, auto-increment, displacement, and deferred addressing [7]. The VAX was built as a successor to the earlier PDP-11 architecture and benefited by expanding the pre-existing data types for availability with all addressing modes [1]. This created fully orthogonal encoding.

Listing 1 is an example program for computing the first 10 Fibonacci numbers, taking two values from memory and demonstrating the complexity of VAX ISA.

Listing 1: VAX Fibonnaci Program

```
        MOVW [#51], r1          :get 1st no.
        MOVW [#52], r2          :get 2nd no.
        CLRW r3                 :clear r3 (n)
        CLRW r4                 :clear r4 (t)
.loop   AOBLEQ #8, r3, .next    :n++, ?(n<10)
        JMP .end
.next   SUBB3 #1, r4, r0        :r0 = r4 - 1
        BEQL .calc2             :Branch if Z=1
.calcr1 ADDW2 r2, r1            :r1 = r1 + r2
        ADDW2 #1, r4            :r4++
        MOVW r1, [#52 + r3]     :Store r1
        JMP .loop
.calcr2 ADDW2 r1, r2            :r2 = r1 + r2
        SUBW2 #1, r4            :r4--
        MOVW r2, [#52 + r3]     :Store r2
        JMP .loop
.end
```

3.2 DEC Alpha

The DEC Alpha is a 64-bit RISC microprocessor produced in 1992 to replace the VAX system in response to other RISC systems surpassing it's price to performance ratio [8]. Design focussed on simpler, cheaper hardware which maintained compatibility with a range of operating systems [8]. The instruction formatting, as seen in Figure 3, shows the simplicity and strong orthogonality of the architecture coupled with consistent ordering of operands. The code density is worse than its predecessor as can be seen in Listing 2. The difference may not seem like much in this example, but it becomes more significant when calling subroutines [6].

Opcode	Number				PAL code
Opcode	RA	Disp			Branch
Opcode	RA	RB	Disp		Memory
Opcode	RA	RB	Func	RC	Operate
6	5	5	11	5	Bit count

Table 3: Formatting of DEC Alpha ISA [8]

Listing 2: DEC Alpha Fibonnaci Program

```
        ADDL r31, #51, r5       :Start at memory 51
        LDL r1, 0(r5)           :r1 = first value
        LDL r2, 1(r5)           :r2 = second value
        ADDL r31, #10, r3       :r3 = 10 (n)
        ADDL r31, r31, r4       :r4 = 0 (t)
.loop   BGT r3, .next           :test r3 > 0
        BR r31, .end
.next   BGT r4, .calcr2         :text r4 > 0
.calcr1 ADDL r1, r2, r1         :r1 = r1 + r2
        ADDL r4, #1, r4         :r4 = 1
        SUBL r3, #10, r6        :r6 = n - 10
        SUBL r31, r6, r6        :r6 = 10 - n
        STL r1, 1(r6)           :Store r1
        BR r31, .loop
.calcr2 ADDL r1, r2, r2         :r2 = r1 + r2
        ADDL r31, r31, r4       :r4 = 0
        STL r2, 2(r6)           :Store r2
        BR r31, .loop
.end
```

3.3 Intel 8086

The Intel 8086 is a CISC microprocessor developed between 1978 and the 1990s, with each new version adding functionality and improvements to the previous [9]. The resulting ISA is very complex with approx 713 instructions of 8-32 bits long, and internal data path sizes up to 256 bits [10]. The resultant system was hard to work with, but benefited from simple backwards compatibility and good code density. Orthogonality has been maintained by using subsets of a general format, as shown in Figure 1.

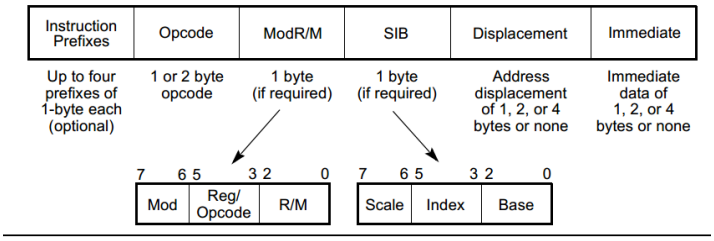

Figure 1: Formatting of Intel 8086 [10]

4 Conclusion

Implementation of a microprocessor design based on a RISC processor will produce an ISA which has good orthogonality with easy decoding. However ,with only a small number of instructions to be implemented to aid simplicity, there is going to be a lot of unused values for the instruction binary. By adding some CISC-style functionality, such as stack operations, density can be improved by incorporating common sequences into one instruction.

It was decided to base an ISA design on the 16-bit ARM Thumb subset. This was because it is an accumulation of the most common instructions from the main ARM ISA and designed to reduce the overall program size compared to its 32-bit counterpart [11]. This means an inherently dense coding, with a wide range of simple instructions. Also, the earlier mentioned investigation at Cornell University suggests there is little difference in terms of code density if there is no 'zero' register like in the ARM [3].

Word Count: 1686