George Mohammad
CS3339 Project Report
Acronym Generator

Introduction

This project was executed to learn more about x86, and, thus, learn more about a real-world architecture. Another purpose of this project was to develop a program that supplies the user with mnemonic devices to memorize to improve memory by enhancing memorization by association.

Background

X86 is a CISC architecture with many features absent in MIPS.

Methodology

The student struggled for a while with getting the environment setup because certain features of the architecture flat-out wouldn't compile on mac. In the end, he decided to use Ubuntu because it has a lot of online support. It was discovered that a statement must be included with the compilation command or the code won't compile. That statement is listed in the appropriate assembly files. On the other hand, the c code was written on macOS Mojave and compiled with gcc. The assembly code was also compiled with gcc.

Mixing C with assembly was attempted, but the assembly code returned garbage rather than the appropriate value, so that line of reasoning was abandoned.

The student approached this project by learning about assembly concepts and implementing them in small programs to learn the concepts before proceeding further. That didn't work. The assembly code was not finished. In the end, a C implementation was developed and translated into assembly using gcc because the student bit off more than he could chew. However, he learned a lot. The student learned the mnemonics and structures of many instructions in the x86 architecture as well as skills for learning about computer science without the guidance of a teacher including research skills and time management skills. Also, the student gained a more advanced understanding of C.

There wasn't much instruction type variety in the aforementioned assembly code, so there aren't many instances of assembly code embedded in this report.

Instruction encoding

X86 Instruction prefixes allow the programmer to add caveats to instruction behavior such as favoring certain branch prediction outcomes over others. MIPS does not support this feature. As a result, the programmer has more control.

As for opcode, x86 allows the use of an escape opcode which modifies subsequent bits. The rest of the bits are further used to specify the instruction type. MIPS doesn't

have escape opcodes. Escape opcodes could make more efficient use of instruction bits (https://stackoverflow.com/questions/52346724/what-does-escape-opcode-mean).

X86 is flexible. It allows instruction encoding fields to take on different roles depending on the instruction. ex. Being a register or becoming more opcode bits. MIPS is more rigid. Each field does something specific in MIPS. Also, x86 can adjust the range and precision of an instruction.

2.2.1.1 In x86, in R/M mode, two registers can be used. With MOD R/M with SIB, registers are combined with base (specifies the starting location) , index (specifies the offset), and scale (adjusts range and precision). MIPS supports addressing modes for return, immediate, and jump/branch instructions. It doesn't support the inclusion of a base, index, or scale factor.

2.2.1.6 With RIP-Relative Addressing, memory can be accessed in relation to the instruction pointer (as opposed to zero) which points to the next instruction. In MIPS, memory can be accessed relative to the stack pointer using an offset.

Registers

Table 3-1: x86 supports a more robust set of registers. It allows the use of registers of varying sizes. In contrast, all MIPS registers are 32 bit.

Instructions

ADD in x86 supports immediate plus register, memory plus register, immediate plus memory, and register plus register addition. MIPS has two instructions to accomplish that. The consequences are that x86 may be slightly slower because it has to decide which type of operation to perform. On the other hand, x86 is cheaper on the resources because it only uses two registers to perform addition. One register is a source and destination register simultaneously.

```
addl    $1, %eax #Line 126 in ProjectCodev4.s
```
The above code increments the return register.

ADCX (Unsigned integer addition with carry) is the closest x86 instruction to ADDU in MIPS. ADCX grabs the overflow and stores it in the CF (carry flag). The CF should be zeroed prior to adding one or more sets of integers. This instruction permits registers and memory locations as operands. The first operand must be a register. This instruction is more robust than the MIPS one, but it's also more specialized, so it may not be applicable in as many situations.

AND in x86 can work with operands of type register, memory location, or immediate value, but both registers aren't permitted to be of type memory. This AND operation is more all inclusive than MIPS', so instruction count is likely to be lower than the equivalent MIPS code. AND in x86 supports the use of immediate values and mips doesn't.

In x86, jumps and branches are all handled by one mechanism. They're all jumps. There are caveats with x86 jumps that don't exist with the MIPS variety such as that when jumping to a label that's far away, it's necessary to include instructions that perform the inverse jumping instruction then include an instruction to unconditionally jump to the target label. The x86 developers may have been able to save resources with the opcode when designing this instruction because branches and jumps are handled by jumping, so they wouldn't have to specify the opcode for two different instructions.

```
jmp     LBB0_5 #Line 34 in ProjectCodev4.s
```
The above code jumps unconditionally to a label.

LEA (Load Effective Address) loads data from a memory location (the second operand) into a register (the first operand). This instruction supports multiple register sizes and memory address sizes. This instruction encompasses all of the MIPS load instructions that handle loading values into registers of varying sizes. Designing the instruction that way reduces the instruction count, but it may increase the CPI because the instruction is more complex in x86. Also, loading the address rather than the value at that address frees up memory for other things.

NOT in x86 can handle registers and memory locations. It performs the logical operation, NOT, on the target operand.

OR ors two values and performs the functions of ORI and OR in MIPS. The destination must be a register or memory value. Both operands are forbidden from being memory locations. The source operand can be a constant, register, or memory location. X86 OR differs from the MIPS variety in the sense that MIPS OR only permits the oring of registers. The consequence of the x86 OR structure is increased instruction versatility, and possibly higher CPI.

No NOR instruction exists in x86. NOR can be achieved by negating the result of an OR operation.

SETCC exists in x86 to fill all the roles of the set instructions in MIPS. SETCC assigns a value, zero or one, to a register based on the outcome of a comparison between two components of a specific register. CC is replaced with letters that indicate the comparative operation to be performed. Besides being possibly slightly slower, this instruction is the same as the MIPS equivalent because you have to memorize roughly the same number of characters.

X86 left shift shifts a register, and replaces the rightmost 0 bits with the value in the second operand. Compared to MIPS, both instructions have the same number of registers, but they serve different functions as stated previously. This instruction is more powerful than the MIPS instruction as is to be expected from a CISC architecture.

X86 right shift behaves the same as left shift, but the bits are shifted to the right, and the second operand specifies bits that are loaded onto the leftmost bits.

MOV in x86 does the same thing as store and load in MIPS. The register where the result is stored can't be an immediate value, but the destination and the source operands can be a register or memory location, and the source operand can be a constant. Consequences: This instructions requires less memorization because of its versatile nature. ie. You don't need to memorize store and load and mov separately.

movq %rdi, -16(%rbp) ##Line 14 in ProjectCodev4.s
The above instruction moves the contents of a 64 bit register into a memory location. The q was used to specify that 64bit registers are the only type of registers allowed as operands. Information about the q suffix can be found at: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf.

SUB in x86 permits the use of constants, register, and memory location operands. The destination value can't be an immediate, and both operands mustn't be memory locations. In the relation to MIPS, this instruction is functionally and syntactically the same. Ie. It does the same thing and has the same number of operands. This instruction accomplishes the same thing as the MIPS equivalent.

There isn't an unsigned subtraction mnemonic in x86.

X86 DIV works with registers and memory locations. It's the same as MIPS except for the name (DIV vs. DIVU). No consequences because of the preceding statement.

X86 DIV unsigned has the same mnemonic as MIPS signed divide. Other than its signed nature, it's the same as DIV.

There is no exactly equivalent set of instructions for floating point arithmetic in x86. The x86 floating point instructions involve "packed" values rather than normal ones.

MUL/IMUL function the same as DIV/IDIV except IMUL perform multiplication and support multiple instruction configurations. With IMUL, the programmer can use one, two, or three operands. With one operand, one value is stated explicitly and the other comes from the return register such as EAX. With two operands, the destination and source operand are multiplied and the output is placed in the destination operand. With three operands, the sources are multiplied and the result is assigned to the destination operand.

George Mohammad
CS3339 Project Report
Acronym Generator

Trials and Struggles

This project taught me a lot about x86. I didn't deliver all the deliverables because I bit off more than I could chew. It's not for lack of trying. I spent many hours over a period of several weeks working on this project, but some of the assembly stuff wasn't clicking. I know this doesn't justify not meeting the full requirements. I just wanted to be clear that I gave it my full effort and learned something.