Lab 8: Processor Architectures

#### Part 1: MIPS

MIPS processors have widely been recognized as high performance, high power efficiency, and low space consumption processors. These processors are based off of the famous MIPS architecture. MIPS is developed by MIPS technologies who have recently been bought by Imagination Technologies. The MIPS architecture has a number of vital functionalities, including, “single instruction multiple data and virtualization, multi-threading, enhanced virtual addressing, and digital signal processing,”1 to name a few.

For some basic information on the MIPS architecture, I scoured Imagination Technologies’ website. I found that the MIPS architecture is based the reduced instruction set computing architecture2. This inherently means that MIPS will run at a faster clock speed with its simpler and smaller instruction set as opposed to a processor architecture based on CISC, which has a richer, more complex instruction set. This is also partly due to the fact that MIPS has a fixed-width instruction set3. The MIPS architecture also allows for simplified hardware because it employs a fixed-width instruction set. As far as the cache layout of the MIPS64 architecture provides for, “Flexibility of high-performance caches and memory management schemes,”1. According to the Imagination Technologies website, the instruction data caches can range from 256 bytes all the way up to 4 megabytes2!

MIPS supports bytes, halfwords and words3. Common integers require 4 bytes of storage, and characters require 1 byte of storage. As far as bit width, bytes are 8 bits long, halfwords are 16 bits long, and words are 32 bits long3. These data types can be stored in either big-endian or little-endian configurations5. The MIPS register set includes 32 general-purpose registers, as well as two registers specified HI and LO4. The registers all have bit-widths of 64 bits for MIPS64 in order to accommodate for the 64 bit address space6.

Since MIPS using fixed width instructions, the instruction encoding is simple to understand and view. I will use a 32-bit implementation for simplicity. For arithmetic operations, the first 6 bits are 0s, the following 20 bits are split into four five-bit chunks for Source 1, Source 2, Destination, and Shift Amount. Finally, the last 6 bits are reserved for the Function type. For memory accesses and conditional branch instructions, since these are “i” type instruction in mips, there a 6 bits for the opcode, 5 bits for the source, 5 bits for the destination, and 16 bits for the immediate operand. Lastly, for call instructions, these are “j” type instructions in MIPS, which reserve 6 bits for the opcode and 26 bits for the pseudo-address.

To finish, MIPS implements a three-address instruction set. This is seen in the instruction formats above such as most arithmetic operations. There is generally one destination and two sources. Obviously there can be instructions with less than three addresses, such as the jump instruction. However, MIPS is still classified as three address because it accepts up to three addresses for a subset of its instructions.

#### References:

[1] MIPS64 Architecture: <https://imgtec.com/mips/architectures/mips64/>

[2] MIPS Architecture: <https://imgtec.com/mips/architectures/>

[3] MIPS Quick Tutorial: <http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>

[4] MIPS Register Files: <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html>

[5] MIPS Endianess: <https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html>

[6] MIPS Instruction Code Formats: <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s07.html>

[7] MIPS Three Address Code: <https://courses.engr.illinois.edu/cs232/sp2010/lectures/l02.pdf>

#### Part 2: x86-64

The x86-64 architecture has been around for quite some time. This instruction set is simply the 64 bit flavor of the famed x86 instruction set. Because this instruction set is so widely used in many people’s personal computers, it is hilarious when people use the 32 bit version of x86, and then proceed to use more than 4GB of ram (without using PAE)1.

This architecture is part of the Complex Instruction Set Computing type (CISC)2 which generally means it’s slower due to the complexity of some of the functions when compared to Reduced Instruction Set Computing. Using the Intel Skylake 6700k as my sample processor, this processor has an 8 MB SmartCache3, which is huge relative to what we saw just a few years ago, as well as other less cache intensive architectures. The instruction set provided by x86-64 is also variable width with instructions ranging anywhere from one to fifteen bytes2.

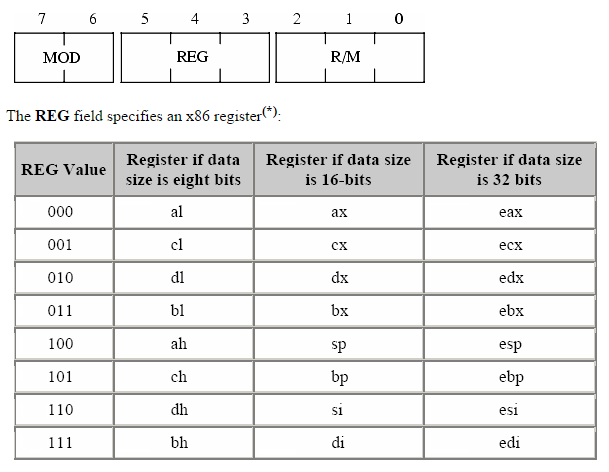
Several data types are supported within the x86-64 architecture (x86 in general). Types such as char, short, int, and long int are supported which can been either implicitly signed or explicitly unsigned for an extra bit2. The bit widths are 8 bits, 16 bits, 32 bits, and 64 bits, respectively. This architecture also supports 32 and 64 bit floating point numbers (floats and doubles). Unlike MIPS (mentioned above), x86-64 uses a little-endian convention4.

x86-64 provides 16 general purpose registers which are 64 bits wide each5. Next, eight 80-bit wide x87 registers are provided5. There are also eight 64-bit MMX registers, and sixteen 128-bit SSE registers5.

Since this architecture uses variable width instructions, it is a little harder to visualize when compared to MIPS above. X86-64 has one general instruction format. First, there are 0 - 4 bytes of prefix, then 1 - 2 bytes reserved for the opcode section. Next, there’s 1 byte split between the MOD R/M section, there’s another byte for the Scale Index Base section. Finally, there are 0 - 4 bytes available for Displacement, which depends on the addressing mode, and 0 - 4 bytes available for Immediate, whose length is specified in the opcode. See examples below:

Encoding: 7

* Arithmetic (Add): There are several flavors of performing the add function with this architecture. The above image shows the “opcode” section of the instruction. The leading six 0s indicate that this is an ADD instruction. In the place of d would be a 0 or a 1, depending on where we’re adding from. In the place of the s would be a 0 or 1, depending on how big the operands are. Next, as shown below, the MOD R/M byte is split up. For the same instruction, the d bit in the opcode determines whether REG is the source or the destination7.

Encoding: 7

* Memory Access (LODS): There are several flavors of the load string instruction. For simplicity, I’ll use the AC opcode. The MOD R/M and SIB bytes can encode up to two operands of the instruction, each of which are a register or memory address9.
* Call (JUMP): The jump instruction transfers program control flow to the instruction indicated by the single operand included. This instruction, again, has a specific opcode which then determines one of the many flavors it may take on. There’s also only one operand here which is a memory address.
* Conditional Branch (JE): The jump equal instruction transfers program control flow to the instruction indicated by the single operand included if the special register labeled machine status word is set to equal8. This instruction is then encoded similarly to the JUMP instruction above.

Lastly, this instruction set uses two address code2, 5. There are only two operands per instruction, one input, and another input which is also the output. This means that many instructions overwrite one of the inputs. As stated above, some functions may accept two operands, others may accept less. Since there is a multitude of instructions included in x86-64, there are several examples of each type.

#### References:

[1] PAE: <https://askubuntu.com/questions/19389/whats-the-meaning-of-pae-at-the-end-of-kernel-version>

[2] x86 Introduction: <https://www.seas.upenn.edu/~cdmurphy/cit593/fall2012/pdf/lecture20-x86.pdf>

[3] Intel Skylake: <http://ark.intel.com/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz>

[4] Endianess: <http://www.yolinux.com/TUTORIALS/Endian-Byte-Order.html>

[5] Registers: <https://msdn.microsoft.com/en-us/library/ff561499.aspx>

[6] Two address code: <https://www3.nd.edu/~dthain/courses/cse40243/fall2015/intel-intro.html>

[7] Encoding: <http://www.c-jump.com/CIS77/CPU/x86/lecture.html>

[8] Encoding: <https://www.cs.virginia.edu/~evans/cs216/guides/x86.html#instructions>

[9] Encoding: <https://www-user.tu-chemnitz.de/~heha/viewchm.php/hs/x86.chm/x64.htm>