Sidharth Gilela
Student ID #: 1428033
Email: sgilela@ucsc.edu
Homework #1
October 5, 2015

CMPE 110 Computer Architecture: Homework 1

1A) lw (50 of them because loops 50 times)
lw (50 of them because loops 50 times)
sw (50 of them because loops 50 times)
50 + 50 + 50 = 150 data memory accesses are made during the execution of the code

1B) CPI = cycles/instruction
<u>Cycles</u>
Load = 3 cycles
Store = 2 cycles
Others = 1 cycle
Since the program is looped 50 times we will multiply the cycles per instruction by 50 for each instruction in the loop and then add up all the instruction's cycles
= 1 + (50 + 150 + 50 + 150 + 50 + 50 + 100 + 50 +50 +50)
= 751 cycles
<u>Instructions</u>
Since each instruction is looped 50 times except the first one, we will get:
50(10) + 1 = 501 instructions
CPI = cycles/instruction = 751 cycles / 501 instructions = 1.499 cycles/instruction

1C) Execution time = (cycles/instruction) x (instruction count) x (cycles)
= (1.499 cycles/instruction) x (501 instructions) x (500 ns)
= 375499.5 ns

2A) Some of the tradeoffs between RISC and CISC are that with RISC, there is a low CPI and clock period and a high instruction count. With CISC, there is a high CPI and clock period and a low instruction count. Some of the features of risk are that it has many instructions that access memory directly, and has a large number of addressing modes. Additionally, it has variable length instruction encoding and support for misaligned accesses. Features of RISC are that it has fixed length instructions, hardwired controller instructions, and compound instructions. Why people would want to choose RISC over CISC is because of clock rates. RISC processors can achieve performance from up to 2 to 4 times as much as CISC and that is because in RISC processors, the simple instructions have a runtime of 1 clock cycle each. In comparison, CISC has complex instructions that take longer to execute. Another advantage is that the RISC instructions are a lot simpler than those of CISC and because of that, they use less space one a chip. Why people would choose CISC over RISC include reasons such that the RISC processor depends on the code too much when executing. The reason why this is bad is because if the programmer does a bad job at instruction scheduling, then the processor could spend a lot of time stalling. Another advantage to using CISC would be code expansion. CISC machines can perform complex actions with a single instruction but RISC machines may requires several pieces of code to do so. Code expansion could be defined as the increase in size that you get when you take a program that had compiled for a CISC machine and re-compile it for the RISC machine. Another reason as to why CISC could be seen as better is because RISC machines require fast memory systems to feed them instructions. RISC machines typically have large amounts of memory caches.

2B) Latency is dependent on the ISA. The reason why could be explained through a formula but first, I will define latency. Latency is defined as seconds/program. In order to determine the seconds/program, we can use the formula (instructions/second) x (cycles/instruction) x (seconds/cycle). Using this formula gives us the product which is the latency. Since, however the ISA is developed, affects the (instructions/program) and (cycles/instruction), these components will in return affect the (seconds/program). And, since (seconds/program) is latency, latency is dependent on the ISA.

The ISA determines the code size. This is because, depending on whether a RISC or CISC architecture is used, the code size varies. With CISC, complex instructions can be performed with a single instruction resulting in an ISA that should be able to handle this time of architecture. With RISC, its actually the opposite, and the ISA should be able to handle and implement less complex and easier to do instructions. The ISA also determines performance and again, it would be easier to explain by using RISC and CISC architecture. With RISC architecture, we could execute a high amount of cycles per instruction in comparison to CISC architecture, where we execute a low amount of cycles per instruction. These different CPI values affect how well the performance is for an ISA.

3A)

Instruction	Proportion	CPI (baseline)	CPI (opt #1)	CPI (opt #2)
Load/Store	30%	3	2	3
Branch	25%	2	1	2
Mul/Div	8%	7	7	1
Other	37%	1	1	1
Total	100%	2.33	1.78	1.85

<u>CPI baseline</u>

(0.3 x 3) + (0.25 x 2) + (0.08 x 7) + (0.37 x 1) = 2.33 CPI

3B) <u>CPI opt#1</u>

(0.3 x 2) + (0.25 x 1) + (0.08 x 7) + (0.37 x 1) = 1.78 CPI

<u>CPI opt#2</u>

(0.3 x 3) + (0.25 x 2) + (0.08 x 1) + (0.37 x 1) = 1.85 CPI

<u>CPI speedup of opt#1 relative to baseline is</u>

1.78/2.33 = 0.764

<u>CPI speedup of opt#2 relative to baseline is</u>

1.85/2.33 = 0.794

Option#1 is better to improve performance because it has the lowest CPI

4)

Architecture	Bytes in Program	Bytes Fetched	Instruction Count	Program Latency
x86	16 bytes	81 bytes	43 instructions	68 cycles
MIPS	28 bytes	108 bytes	27 instructions	58 cycles
Stack ISA	18 bytes	83 bytes	40 instructions	132 cycles

4A) <u>Bytes in Program</u>

2 + 2 + 2 + 2 + 1 + 2 + 2 + 3 = 16 bytes

<u>Bytes Fetched</u>

(2 x 6) + (2 x 6) + (2 x 6) + (2 x 6) + (1 x 6) + (2 x 6) + (2 x 6) + 3 = 81 bytes

<u>Instruction Count</u>

(7 x 6) + 1 = 43 instructions

<u>Program Latency</u>

(1 x 6) + (1 x 6) + (1 x 6) + (1 x 6) + (5 x 6) + (1 x 6) + (1 x 6) + 2 = 68 cycles

4B) <u>MIPS Code for GCD:</u>

Assume r2 is int a and r3 is int b and r4 is temp and r0 is the address

```
        xor      r1, r1, r1     //set rl to = 0
        j        forw           //jump to branch statement
        addiu    r4, r3, 0      //set temp to equal value of 'b'
        remu     r3, r2, r3     //do 'a' mod 'b'
        addiu    r2, r4, 0      //setting value of 'a' with the value of temp
forw    bne      r3, r1, -4     //branching back to ins after jump if 'b' not = 0
        sw       r2, 0(r0)      //store 'a' into the memory with the address value
```

<u>Bytes in Program</u>

7 x 4 = 28 bytes

<u>Bytes Fetched</u>

4 + 4 + (4 x 6) + (4 x 6) + (4 x 6) + (4 x 6) + 4 = 108 bytes

<u>Instruction Count</u>

1 + 1 + (4 x 6) + 1 = 27 instructions

<u>Program Latency</u>

1 + 1 + (2 x 6) + (1 x 6) + (5 x 6) + (1 x 6) + 2 = 58 cycles

4C) Stack Code for GCD

```
                goto       forw     //jump to branch
rewind          dup                 //duplicate value of 'b'
                reverse             //elements reversed to make: 'a' 'b' 'b' res
                swap                //top two swapped to make: 'b' 'a' 'b' res
                rem                 //mod to make: 'a mod b' 'b' res
                dup                 //copy to make: 'a mod b' 'a mod b' 'b' res
forw            bnez       rewind   //pops, then branch if value not = 0
                reverse             //reverse to make: res 'b' 'a mod b'
                swap                //swap to make: 'b' res 'a mod b'
                popm                //save value of b into address, into memory
```

Bytes in Program

5 + 1 + 1 + 1 + 1 + 1 + 5 + 1 + 1 + 1 = 18 bytes

Bytes Fetched

5 + (1 x 6) + (1 x 6) + (1 x 6) + (1 x 6) + (1 x 6) + (5 x 6) + (1 x 6) + (1 x 6) + (1 x 6) = 83 bytes

Instruction Count

1 + (6 x 6) + 3 = 40 instructions

Program Latency

1 + (2 x 6) + (5 x 6) + (3 x 6) + (7 x 6) + (2 x 6) + (1 x 6) + 5 + 3 + 3 = 132 cycles

4D) Some of the tradeoffs between these different ISA's can be seen through the comparison of instruction count and latency. It seems MIPS is able to perform the most efficient with about 58 cycles using 27 instructions. Stack, on the other hand seems to take about double the amount of cycles with only ten more added instructions. The ISA performance for MIPS can be better explained by its type of architecture which is RISC. This type of architecture allows MIPS to perform with a low CPI but a high amount of instructions and clock period. It seems from the table that in comparison to the other ISA's, MIPS has an average throughput since it is able to finish a high number of tasks in a certain amount of time. The latency is basically the amount of time the program takes to execute so looking at the instruction counts, it can be seen that 27 instructions are done in 58 cycles. In comparison to x86, the x86 seems to have a higher throughput. It can do a more amount of instructions in only a little bit more of time in comparison to MIPS. However, stack seems to have the worst throughput. It does 40 instructions in about 132 cycles. If we were to compare the throughput of x86 and Stack, x86 almost has double the throughput. To talk about performance for the x86, we could talk about its architecture which is the CISC architecture. CISC architecture has a high CPI and clock period but a low instruction count. CISC architecture deals with doing complex instructions all at once. This affects the performance of an ISA being with CISC architecture, the ISA can perform better in certain scenarios and not perform better in other scenarios. In general, stacks would be better to use in cases of simple addition or a simple mod function. Since stacks take a lot of cycles per instruction, the longer the instruction count is, the much longer the program will take. Thus, any program that can be written in a short length of code using Stack ISA would be good for stacks. MIPS is generally good when dealing with programs that are looped. This example right here with

GCD allows MIPS to perform to its best. MIPS is also easy to use when wanting to convert from C or Java. Most of the ISA operations have a similar functionality in C or Java. Some of the anomalies I see have to do with the type of architecture each ISA is and how it doesn't quite fit in with the instruction count and latency. For example, x86 is a CISC architecture. However, as we know in a CISC architecture, there is a high CPI with low instruction count, the numbers for the GCD program don't quite add up. In GCD, the x86 seems to have a high instruction count with a relatively low latency. If anything, the Stack would better fit the description of being a CISC architecture. However, a stack is a RISC architecture and another anomaly is seen right here. The x86 seems to have a high throughput when it should not and thus, these are the anomalies I see with the x86.

EX = C(of x) x P(of x) x I(of x)
EX = (68/43) x P x 43
EX = 68 x P(x)

EM = C(of m) x P(of m) x I(of m)
EM = (58/27) x P x 27
EM = 58 x P(m)

ES = C(of s) x P(of s) x I(of s)
ES = (132/40) x P x 40
EM = 132 x P(s)

The ratio of cycle times between x86 and MIPS is 68/58 which is about 1.172. The ratio of cycle times between MIPS and Stack is 132/58 = 2.276. The ratio of cycle times between Stack and x86 is 132/68 = 1.941. These ratios signify how much better in cycles one ISA is when compared to the other for doing the GCD program. For example, if we were to take the MIPS and Stack whose ratio is 2.276, then it can be seen that the cycles for MIPS takes about 2.276 faster than the cycles for Stack.