CMPE 110 Computer Architecture
Fall 2015, Homework #1

Computer Engineering
UC Santa Cruz

October 5, 2015

Name: ______________________________

Email: ______________________________

Submission Guidelines:

- This homework is due on Friday 10/16/15.
- The homework must be submitted to ecommons by 7:59am.
  - Anything later is a late submission
- **Please write your name and your UCSC email address**
- **The homework should be "readable" without too much effort**
  - The homework must be typed and submitted as a single file in PDF format
  - Please name your homework file cmpe110-hw1-yourcruzid.pdf
  - Please keep your responses coherent and organized or you may lose points
- Provide details on how to reach a solution. **An answer without explanation gets no credit. Clearly state all assumptions.**
- Points: $64 = 16 + 8 + 8 + 32$

Question	**Part A**	**Part B**	**Part C**	**Part D**	**Total**
1				-	
2			-	-	
3			-	-	
4					
Total					

Question 1. Vector-vector Add (16 points)

Consider the C code given below, which performs vector-vector addition on arrays of size 50:

```
int i;
for ( i = 0; i < 50; i++ )
  x[i] = y[i] + z[i];
```

Using a MIPS32 cross-compiler, we can compile the code into the following assembly, shown below.

```
      addi $t0, $zero, $zero
loop: add $t2, $t0, $t5
      lw $t3, 0($t2)
      add $t2, $t0, $t6
      lw $t4, 0($t2)
      add $t3, $t3, $t4
      add $t2, $t0, $t7
      sw $t3, 0($t2)
      addi $t0, $t0, 4
      slti $t2, $t0, 200
      bne $t2, $zero, loop
```

Question 1.A Memory Accesses (4 points)

Calculate the number of total number of *data* memory accesses made during execution of the code.

Question 1.B CPI (6 points)

Calculate the CPI of this program. Assume every load instruction takes 3 cycles, every store instruction takes 2 cycles, and all other instructions take 1 cycle. Also assume that instructions are executed in sequence (one after the other).

Question 1.C Execution Time (6 points)

Assuming a clock period of 500 ns, what is the execution time of this program?

Question 2. Architectures and Instruction Sets (8 points)

This question addresses qualitative characteristics of different architecture types and instruction sets.

Question 2.A CISC vs RISC (4 points)

Compare and contract CISC with RISC. What are the trade-offs? Why choose one over the other?

Question 2.B ISAs (4 points)

Answer the following points about properties and characteristics of various Instruction Set Architectures.

- Is latency dependent on the ISA? Explain.
- Which of the following metrics is determined by the ISA: performance, die area, energy efficiency, code size? Why?

Question 3. Processor Optimization (8 points)

The table below shows the distribution of each instruction type on a processor running with a 2 GHz clock frequency.

Instr	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%			

Question 3.A Baseline CPI (2 points)

What is the average CPI of the baseline processor? Be sure to fill in your answer in the table above.

Question 3.B Optimization (6 points)

With some optimization, and the support of some extra logic, the microarchitecture can decrease the load and store CPI to 2 and branch CPI to 1 (option #1). Alternatively, the extra logic that can be spent can be used to optimize the mul/div logic to support CPI = 1 (option #2). What is the CPI speedup of each optimization relative to the baseline design? Which optimization is better to improve performance?

Question 4. Comparing ISAs (32 points)

In this question we will compare three different architectures.

- The first is x86: an extended accumulator, CISC architecture with variable-length instructions and 32-bit data values.
- The second is MIPS, a register-register RISC architecture with 32-bit fixed-length instructions and 32-bit data values.
- The third is a simple Stack ISA with variable-length instructions and 16-bit data values.

Consider the following C code which takes two integers (a and b) as inputs, computes the Greatest Common Divisor (GCD) between them, and stores the result in memory.

```
void gcd( int a, int b, int* result )
{
  while ( b != 0 ) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  *result = a;
}
```

If we were to take this code and compile it on an x86 machine (like your laptop), we would get the following assembly code. You can try this at home.

```
loop: test %esi, %esi
      je End
      mov %eax, %edi
      mov %edi, %esi
      div %esi
      mov %esi, %edx
      jmp loop
End:  mov (%rcx), %edi
```

If you were to try compiling this yourself, you would get the above output. In this assembly, `%edi` contains `a`, `%esi` contains `b`, and `%rcx` contains `result`. All other similar terms are names referring to registers in the processor.

The table at the top of the next page describes the operation of the x86 instructions as well as the size of the encoding and the latency of the instruction execution.

Instruction	Operation	Size	Latency
`test ra, rb`	*tmp* ← R[`rb`] & R[`ra`] `SF` ← sign bit of *tmp* `ZF` ← 1 if *tmp* = 0, else 0 `OF` ← overflow of *tmp*	2 bytes	1 cycle
`mov rt, rs`	R[`rt`] ← R[`rs`]	2 byte	1 cycle
`mov (ma), rs`	M[`ma`] ← R[`rs`]	3 bytes	2 cycles
`je label`	if (`ZF` = 1) jump to the address specified by `label`	2 bytes	1 cycle
`jmp label`	jump to the address specified by `label`	2 bytes	1 cycle
`div rs`	`%eax` ← `%eax` / R[`rs`] `%edx` ← `%eax` mod R[`rs`]	1 byte	5 cycles

ASSUMPTIONS: Unless otherwise specified, assume `a` is 100 and `b` is 126. Use this information in the remainder of this problem to compute the values in the table below.

Architecture	Bytes in Program	Bytes Fetched	Instruction Count	Program Latency
x86				
MIPS				
Stack ISA				

Question 4.A x86 ISA (8 points)

Fill out the first row of the above table.

- **Bytes in program** refers to the static size of the code (in bytes) in memory.
- **Bytes fetched** is the total number of bytes fetched by the processor (i.e., read from instruction memory) during the execution of the program.
- **Instruction Count** is the total number of instructions fetched by the processor from memory during the execution of the program.
- **Program Latency** is the total time (in cycles) for the program to run from start to finish. Assume instructions run sequentially, one after the other.

Question 4.B MIPS ISA (8 points)

Write the assembly code that would generate if the C code were compiled on a machine that uses the MIPS ISA. Fill out the second row of the above table.

The table below describes a subset of the MIPS instructions in detail. **Use only these instructions to compile the code into MIPS assembly.**

Instruction	**Operation**	**Latency**
`addiu rt, rs, imm`	R[`rt`] $\leftarrow$ R[`rs`] + `imm` (unsigned)	1 cycle
`xor rd, rs, rt`	R[`rd`] $\leftarrow$ R[`rs`] $\oplus$ R[`rt`]	1 cycle
`remu rd, rs, rt`	R[`rd`] $\leftarrow$ R[`rs`] mod R[`rt`] (unsigned)	5 cycles
`bne rs, rt, off`	if (R[`rs`] $\neq$ R[`rt`]) PC = PC + 4 + 4 $\cdot$ `off`	2 cycles
`lw rt, off(rs)`	R[`rt`] $\leftarrow$ M[R[`rs`] + `off`]	3 cycles
`sw rt, off(rs)`	M[R[`rs`] + `off`] $\leftarrow$ R[`rt`]	2 cycles
`j label`	jump to the address specified by `label`	1 cycle

Question 4.C Stack ISA (8 points)

Suppose we have a machine that uses Stack architecture. A Stack architecture is similar to a traditional architecture except that it uses a hardware stack data structure instead of registers. The stack operates like a traditional software stack as follows:

- Only values at top of the stack may be accessed.
- Stack operates in a Last-In-First-Out (LIFO) scheme.
- *pop* refers to removing the value at the top of the stack while reading it and moving other values up the stack.
- *push* refers to adding a new value to the top of the stack while moving other values down (deeper into the stack).

The table on the next page describes a subset of the Stack ISA in detail, along with the size of the instruction encoding and the latency of the instruction.

Write the assembly code that would generate if the C code were compiled on a machine that uses this given Stack ISA. Assume that going into the code, that the top of stack contains `b`, the second entry in the stack contains `a`, and the bottom of the stack contains `result`. Fill out the third row in the above table.

Instruction	Operation	Size	Latency
`add`	pop x; pop y; push $x + y$	1 byte	2 cycles
`bnez label`	pop x; if $x \neq 0$, jump to address specified by `label`	5 bytes	1 cycle
`dup`	pop x; push x; push x	1 byte	2 cycles
`goto label`	jump to address specified by `label`	5 bytes	1 cycle
`pushi imm`	push `imm`	2 bytes	1 cycle
`pushm`	pop $addr$; push M[$addr$]	1 byte	4 cycles
`popm`	pop x; pop $addr$; M[$addr$] $\leftarrow x$	1 byte	3 cycles
`rem`	pop x; pop y; push $y mod x$	1 byte	7 cycles
`swap`	pop x; pop y; push x; push y	1 byte	3 cycles
`reverse`	pop x; pop y; pop z; push x; push y; push z	1 byte	5 cycles

Question 4.D Comparison (8 points)

Discuss the trade-offs of each of the studied ISAs. Be sure to talk about throughput and performance. In what situation would each one be useful or preferable to the others, assuming all other factors are the same? What ratio of cycles times between the machines would be required to give them equal performance? What does this ratio and other numbers imply about the micro-architecture of each ISA implementation? Don't simply summarize the table–analyze what the numbers mean. Explain any anomalies you see in these numbers with how you expect CISC and RISC machines to behave.