CMPE 110 Computer Architecture
Winter 2015, Homework #1 Solution

Computer Engineering
UC Santa Cruz

October 28, 2015

Question 1. Vector-vector Add (16 points)

Solution.

Question 1.A Memory Accesses (4 points)

Number of Data Memory References = number of load and store instructions = $3 \times 50 = 150$.

Question 1.B CPI (6 points)

Total number of instructions executed = $1 + (10 \times 50) = 501$ instructions

- 1 `addi` instruction before the loop label $\rightarrow 1 \times 1$ cycle = 1 cycle
- 10 instructions (instruction 2 to instruction 11) are executed 50 times
- 2 `lw` instructions $\rightarrow 2 \times 50$ iterations $\times 3$ cycles = 300 cycles.
- 1 `sw` instruction $\rightarrow 1 \times 50$ iterations $\times 2$ cycles = 100 cycles
- 4 `addi` instructions $\rightarrow 4 \times 50 \times 1$ cycle = 200 cycles
- 1 `add` instruction $\rightarrow 1 \times 50 \times 1$ cycle = 50 cycles
- 1 `slti` instruction $\rightarrow 1 \times 50 \times 1$ cycle = 50 cycles
- 1 `bne` instruction executed 50 times $\rightarrow 50 \times 1$ cycle = 50 cycles

Latency of the program: $1 + 300 + 100 + 200 + 50 + 50 + 50 = 751$ cycles

CPI = (751 cycles) / (501 instructions) = 1.50 cycles/instruction.

Question 1.C Execution Time (6 points)

clock period = 500ns

Execution Time = CPI × IC × clock period = 1.50 cycles/instruction ×501 instructions × 500 ns = 375.75 μs

Question 2. Architectures and Instruction Sets (8 points)

Solution.

Question 2.A CISC vs RISC (4 points)

highlighting key points:

CISC	**RISC**
Emphasis on hardware	Emphasis on software
Multi clock instructions	Single clock instruction
Memory to memory "load and store" incorporated in instructions	Register to register "load and store" are separate or independent instructions
Smaller code size	Larger code size
Higher cycles per second	Lower cycles per second

Question 2.B ISAs (4 points)

- In terms of instruction execution, RISC instructions have a lower latency when compared to CISC instructions. Latencies can be detected at the instruction level, for example, branch delay slots expose the latencies of branches at the ISA level. (Pipeline latency is not considered here as it is not a part of the homework).
- ISA determines:
  - Overall performance is dependent on ISA. MIPS metric favors RISC as instructions are simpler.
  - ISA determines die area as CISC architectures are more complex in their design and usually have larger die sizes when compared to RISC.
  - RISC based processors incorporate less clocks per instructions. So energy per instruction is reduced.
  - RISC based architecture will have larger code size when compared to CISC based architecture when implementing same logic.

Question 3. Processor Optimization (8 points)

Solution.

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%	2.33	1.78	1.85

Question 3.A Baseline CPI (2 points)

CPI: $(0.30 \times 3) + (0.25 \times 2) + (0.08 \times 7) + (0.37 \times 1) = 2.33$

Question 3.B Optimization (6 points)

Optimization #1 CPI: $(0.30 \times 2) + (0.25 \times 1) + (0.08 \times 7) + (0.37 \times 1) = 1.78$ cycles/instruction

Optimization #2 CPI: $(0.30 \times 3) + (0.25 \times 2) + (0.08 \times 1) + (0.37 \times 1) = 1.85$ cycles/instruction

The speedup of optimization #1 compared to optimization #2 is $\frac{1.85}{1.78} = 1.04$

Therefore, optimization #1 is the better option because speedup > 1.

Question 4. Comparing ISAs (32 points)

Solution.

Architecture	Bytes in Program	Bytes Fetched	Instruction Count	Program Latency
x86	16	85	45	70 cycles
MIPS	28	108	27	52 cycles
Stack ISA	18	53	40	135 cycles

If `a` is 100 and `b` is 126, the code requires 6 iterations of the while loop during execution. This is used in the calculation of all the following parts.

2.A x86

```
//        Instruction           Size       Latency
  1 loop: test %esi, %esi   // 2 bytes    1 cycle
  2       je End            // 2 bytes    1 cycle
  3       mov %eax, %edi    // 2 bytes    1 cycle
  4       mov %edi, %esi    // 2 bytes    1 cycle
  5       div %esi          // 1 byte     5 cycles
  6       mov %esi, %edx    // 2 bytes    1 cycle
  7       jmp loop          // 2 bytes    1 cycle
  8 End:  mov (%rcx), %edi  // 3 bytes    2 cycles
```

- Total bytes in program $= 2+2+2+2+1+2+2+3 = 16$ bytes in the program.
- Instructions 1-7 are part of the while loop. There are 6 iterations of this loop. In the final iteration of the loop, instruction 1, 2, and 8 are run before finally ending.
- Instruction count: $6 \times 7 + 3 = 45$ instructions.
- Bytes fetched: $6 \times (2+2+2+2+1+2+2) + 2 + 2 + 3 = 85$ bytes.
- Program Latency: $6 \times (1+1+1+1+5+1+1) + 1 + 1 + 2 = 70$ cycles.

2.B MIPS

Assumption: At the start of the program, `a` is in r0, `b` is in r1, and `result` is in r2.

```
//        Instruction        Comment                    Latency
  1       xor r4, r4, r4  // Zeros out r4               1 cycle
  2       j L1            // Jump to check             1 cycle
  3 loop: addiu r5, r1, 0 // temp = b;                 1 cycle
  4       remu r1, r0, r1 // b = a % b;                5 cycles
  5       addiu r0, r5, 0 // a = temp;                 1 cycle
  6 L1:   bne r1, r4, -4  // If ( b != 0 ), loop back  1 cycle
  7       sw r0, 0(r2)    // *result = a;              2 cycles
```

- Seven total instructions in the program, 4 bytes per instruction means 28 total bytes in the program.
- Instructions 3-6 are part of the while loop. The loop runs for 6 iterations. Before the loop begins, instruction 1 and 2 run once. After the loop is finished, instruction 7 is run once.
- Instruction count: $2 + 6 \times 4 + 1 = 27$ instructions.
- Bytes fetched: $4 + 4 + 6 \times 4 \times 4 + 4 = 108$ bytes.
- Program Latency: $1 + 1 + 6 \times (1 + 5 + 1 + 1) + 2 = 52$ cycles.

2.C Stack ISA

```
//                Contents of Stack
//  Instruction   after instruction              Size      Latency
1       goto L1   // Top -> b; a; result         5 bytes   1 cycle
2 loop: dup       // Top -> b; b; a; result      1 byte    2 cycles
3       reverse   // Top -> a; b; b; result      1 byte    5 cycles
4       swap      // Top -> b; a; b; result      1 byte    3 cycles
5       remu      // Top -> a mod b; b; result   1 byte    7 cycles
6   L1: dup       // Top -> b; b; a; result      1 byte    2 cycles
7       bnez loop // Top -> b; a; result         5 bytes   1 cycle
8       reverse   // Top -> result; a; b         1 byte    5 cycles
9       swap      // Top -> a; result; b         1 byte    3 cycles
10      popm      // Top -> b                    1 byte    3 cycles
```

- Going down the line through each instruction, we have $5+1+1+1+1+1+5+1+1+1 = 18$ bytes in this program.
- Instructions 2-7 are part of the while loop. There are 6 iterations of this loop. Before the loop starts: instructions 1, 6, and 7 are fetched. During loop: instructions 2-7 are fetched 6 times each. Instructions 8-10 are run once at the very end.
- Instruction count: $3 + 6 \times 6 + 3 = 42$ instructions
- Bytes Fetched: $5 + 1 + 2 + 6 \times (1 + 1 + 1 + 1 + 1 + 2) + 1 + 1 + 1 = 53$ bytes
- Program Latency: $1 + 2 + 1 + 6 \times (2 + 5 + 3 + 7 + 2 + 1) + 5 + 3 + 3 = 135$ cycles

2.D Comparing the ISAs

Architecture	Bytes in Program	Bytes Fetched	Instruction Count	Program Latency	CPI	Throughput (instr/cyc)
x86	16	85	45	70 cycles	1.56	0.64
MIPS	28	108	27	52 cycles	1.93	0.52
Stack ISA	18	53	40	135 cycles	3.38	0.30

Two new columns added above to table as a reference for argument. Color-coding was also added for emphasis and to strengthen the argument in making comparisons between relative levels of values.

Iron-Law of Processor Performance: Execution Time = (Instruction Count) × (CPI) × (Cycle Time)

If we assume all three machines have the same performance (i.e., execution time) then the ratio of clock periods for x86:MIPS:Stack must be $\frac{1}{70} : \frac{1}{52} : \frac{1}{135} = 702 : 945 : 364$. This also means that at equal cycle times, MIPS would have the highest performance, followed by x86, then Stack with the worse performance.

The first ISA studied, x86, has the highest throughput of all the ISAs which means that the most work is done per single instruction and the more efficient the use of time by the processor. This means that x86 is best for handling large workloads, compared to the other studied ISAs. This sees better application in high-computing servers and high-end PCs. Additionally, the cycle times ratio shows that x86 can sustain a relatively high clock period while maintaining comparable performance. This makes sense because CISC architectures have more complex micro-architectures than RISC machines like MIPS or Stack which would require a longer clock period.

The second ISA studied, MIPS, is a good middle ground in terms of thoughput, with potentially the highest performance as shown by the ratio of clock periods. However, MIPS has a significantly greater memory demand than the other ISAs. The ratio shows that MIPS also has the most lax cycle time requirements its clock period to be more than twice that of Stack and about 20% higher than x86, to maintain comparable performance. I say lax rather than required because MIPS is a RISC architecture which is less complex, so it can handle lower cycle times. MIPS would be best in a system with lax memory and cycle time constraints. A good application for MIPS could be an academic or research setting where these two are not strict. Additionally, the MIPS instructions are relatively intuitive and straightforward meaning it would be a good ISA to use for learning.

The third ISA studied, Stack, is the worst in terms of performance compared to the other ISAs, but has a very low memory footprint and demand. This means that Stack is best for handling smaller workloads, compared to other studied ISAs, and is ideal in more resource-constrained systems like mobile devices. The simplicity of the ISA reflects a relatively straightforward micro-architecture that would require a relatively low cycle time (half compared to the others as shown in the ratio) to maintain a comparable performance. This is reflected in the cycle time ratio above.

An anomaly with these numbers is that the MIPS RISC machine has a lower instruction count than the x86 CISC machine. We typically expect CISC machines to have the lower number of instructions per program than RISC. We don't see this for one of two possible reasons. 1) The x86 assembly code was produced by a compiler rather than by hand, so it may not have used any, or the right, optimization flags. 2) This is one single benchmark not necessarily representative of all applications, so it's okay if these numbers do not show this trend among CISC machines.