# Question 1.1 Processor Performance (18 points)

Suppose we have three different processors P1, P2 and P3 executing the same instruction set. Table below has the clock rate and CPI for each processor.

|  |  |  |
| --- | --- | --- |
| **Processor** | **Clock Rate** | **CPI** |
| P1 | 2.5GHz | 1.0 |
| P2 | 3GHz | 1.5 |
| P3 | 4Ghz | 2.4 |

1. Which processor has the best performance in terms of Instructions per second. **(6 points)**
   * Since Latency = seconds/pngram = instructions/program \* cycles/instructions \* seconds/cycles, we have
     + P1: 2.5/1.0 = 2.5x109 instructions/sec
     + P2: 3/1.5 = 2.0 x109 instructions/sec
     + P3: 4/2.4 = 1.67 x109 instructions/sec
   * Hence, P1 has the highest performance amongst the three processors
2. If the processors each execute a program in 10 seconds, find the number of cycles and the number of instructions. **(6 points)**

|  |  |  |
| --- | --- | --- |
| **Processor** | **Cycles** | **Number of Instructions** |
| P1 | 2.5GHz(10) = 2.5 x1010 cycles | 2.5 x1010 /1.0 = 2.5 x1010 |
| P2 | 3GHz(10) = 3 x1010 cycles | 3 x1010 /1.5 = 2 x1010 |
| P3 | 4Ghz(10) = 4 x1010 cycles | 4 x1010 /2.4 = 1.67 x1010 |

1. If we want to reduce the execution time by 25% but this leads to an increase of 20% in the CPI. What clock rate should we have to get this time improvement? **(6 points)**
   * We know that execution time = (# of inst. \* CP1) / Clock Rate, so to reduce the time by 25% but have CPI increase by 20%, we would need

Execution Time (.75) = (# of inst. \* CPI \* 1.2)/New Clock Rate

* New Clock Rate = Clock Rate (1.2/.75) = 1.6 (Clock Rate)

P1: 2.5(1.6) = 4GHz

P2: 3(1.6) = 4.8GHz

P3: 4(1.6) = 6.4GHz

## Question 1.2.a Memory Accesses (4 points)

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

* The number of data memory accesses made is equal to the number of loads and store instructions
* 3(65) = 195

## Question 1.2.b CPI (6 points)

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

* Total number of instructions executed = 1+(10\*65) = 651 instructions
  1. addi = 1 inst (1 cycle) = 1 cycle
  2. loop = (2 to 11) = 65 cycles
  3. 2 lw = (2 lds)(65)(3cycles) = 390 cycles
  4. 1 sw = (1sw)(65)(2cycles) = 130 cycles
  5. 4 add = (4\*65)(1) = 260 cycles
  6. 1 addi = (1\*65)(1) = 65 cycles
  7. 1 slti = (1\*65)(1) = 65 cycles
  8. 1 bne = (65)(1) = 65 cycles
* Latency = 1+390+130+260+65+65+65 = 976 cycles
* CPI = (976 cycles / 651 inst) = 1.499 ~ 1.50 cycles/inst.

# Question 2.1 Architecture and Instruction Sets (10 points)

## Microarchitecture vs ISA (4 points)

What is the difference between microarchitecture and instruction set architecture?

* The microarchitecture takes in the instruction set architecture (ISA) and it is the way it is implemented in the respective processor. An ISA may be implemented with different microarchitectures due to design or shifts in technology. The ISA is same as the programming model of a processor and includes the execution model, processor registers, address, and data formats.

## Compiler (2 points)

Which information does a compiler need to compile a program correctly:

* + - 1. Microarchitecture
      2. ISA
      3. Both
* The compiler needs information from the ISA, as defined by the definition of an ISA. The ISA tell the compiler how to compile and in order for that to happen, the ISA must provide the right instruction/opcode to execute.

## CISC vs RISC (4 points)

Compare the advantages and disadvantages of RISC with CISC ISA.

|  |  |  |  |
| --- | --- | --- | --- |
| RISC | | CISC | |
| Advantage | Disadvantage | Advantage | Disadvantage |
| * Speed * Simple hardware * Shorter design cycle | * Performance depends on code * Un-optimized coding * Multiple instructions for one action * Require fast memory * Contain large memory cashes | * Can perform complex actions with single inst * Easy to implement * Upwardly compatible * Compiler does not have to be as complicated since microprogramming instruction sets can be written to match high level languages | * IS & chip hardware more complex with each generation of computers * Different inst take diffent amout of clock time to execute * Specialized instructions not used frequently * CISC inst condition codes |

# Question 2.2 Fixed Length Vs Variable Length ISA (8 points)

Suppose we have one fixed-length ISA and one variable-length ISA with following instruction encoding:

1. Fixed Length

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand 1**  (destination) Reg/Imm | **Operand 2**  (source1) Reg/Imm | **Operand 3**  (source2) Reg/Imm |

* + Each opcode is 1 byte and each operand is 1 byte.
  + Register/Register and Register/Immediate operations take 1 cycle and all load/store instructions take 4 cycles.

1. Variable Length

|  |  |  |  |
| --- | --- | --- | --- |
| **Opcode** | **Operand 1**  (destination) Reg/Imm | **Operand 2**  (source1) Reg/Imm | **Operand 3**  (source2) Reg/Imm **(Optional)** |

* + Each opcode is 1 byte and each operand is 1 byte. **The operand 3 is optional and**

**it is implicitly specified in the opcode.** If an instruction does not need the third

operand, it is not used.

* + Register/Register and Register/Immediate operations take 2 cycle and all load/store instructions take 6 cycles. It takes more cycles because variable length increases the decode complexity.

Consider the following assembly code:

1)ADD r3, r1, r2 // r3 = r1 + r2

2)SLL r3, 0x2 // r3 = r3 << 2

1. MOV r5, 0xa // r5=0x0a
2. STW r3, (r5) // MEMORY[r5] = r3
   1. Which one of the two ISAs have smaller code size and by how many bytes? **(4 point)**

* Fixed length = 4+4+4+4 = 16 byes
* Variable = 4+3+3+3= 13 byes
* The variable has the smaller code size by 3 bytes
  1. Which one of the two ISAs take less number of cycles to execute and by how many cycles?

#### (4 point)

* Fixed length = 4+1+1+1 = 7\*4 = 28 cycles
* Variable = 6+2+2+2 = 13

= 6+2+2 = 10

= 6+2+2 = 10

= 6+2+2 = 10 = 42 cycles

* Here we have fixed length taking less cycles to execute by 14 cycles

**Question 3. 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 in- structions 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 Lowest Common Multiple (LCM) between them.

1. int lcm(int a, int b){
2. int n;
3. do{
4. n = n +a;
5. } b != 0);
6. return n;

Corresponding X86 assembly code:

*a* is in register esi. *b* is in register edi. *n* is in register ecx.

1. xor ecx, ecx;
2. Loop: add ecx, esi;
3. mov eax, ecx;
4. xor edx, edx;
5. idiv edi;
6. test edx, edx;
7. jne Loop;
8. mov eax, ecx;
9. ret;

The table in next page describes the operation of the X86 instruction as well as the size of the encoding and the latency of the instruction execution.

ASSUMPTIONS: Unless otherwise specified, assume **a is 24** and **b is 5**. Use this information to compute the values in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Architecture** | **Bytes in**  **Program** | **Bytes**  **Fetched** | **Instruction**  **Count** | **Program**  **Latency** |
| **x86** | 17 | 70 | 41 | 71 |
| **MIPS** | 24 | 72 | 18 | 43 |
| **Stack ISA** | 16 | 48 | 24 | 74 |

ZF *←* 1 if *tmp* = 0, else 0

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Operation** | **Size** | **Latency** |
| xor rt, rs | R[rt] *⊕* R[rs] | 3 bytes | 1 cycle |
| add rt, rs | R[rt] + R[rs] | 1 byte | 1 cycle |
| test ra, rb | *tmp ←* R[rb] & R[ra] | 2 bytes | 1 cycle |
|  | SF *←* sign bit of *tmp* |  |  |

OF *←* overflow of *tmp*

mov rt, rs R[rt] *←* R[rs] 2 byte 1 cycle

jne label if ( ZF = 0 ) 2 bytes 1 cycle

jump to the address specified by label

|  |  |  |  |
| --- | --- | --- | --- |
| idiv rs | eax *←* eax / R[rs]  edx *←* eax mod R[rs] | 1 byte | 7 cycles |
| ret | return control to calling program | 1 byte | 1 cycle |

* 1. **x86 ISA (8 points)**

Fill out the first row of the table provided.

* + - **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.

|  |  |
| --- | --- |
| Bytes | Cycles |
| xor | ecx, ecx | 3 | 1 |
| Loop: add | ecx, esi | 1 | 1 |
| mov | eax, ecx | 2 | 1 |
| xor | edx, edx | 3 | 1 |
| idiv | edi | 1 | 7 |
| test | edx, edx | 2 | 1 |
| jne | Loop | 2 | 1 |
| mov | eax, ecx | 2 | 1 |
| ret |  | 1 | 1 |

* Total byes in program = 3+1+2+3+1+2+2+2+1 🡪 17 bytes
* Bytes fetched = 5(1+2+3+1+2+2+2+1) 🡪 70 bytes
* Instruction Count = 5\*8+1 🡪 41 instructions
* Program latency = 5(1+1+1+7+1+1+1+1)+1 🡪 71 cycles

## 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.**

**Assumption:** a is in register a1, b is in b1, n is in s1, use v0 to store result, return address is in register ra. Use t0 as temporary register.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** |  | **Operation** | **Latency** |
| add rd, rs, | rt | R[rd] *←* R[rs] + R[rt] | 1 cycle |
| xor rd, rs, | rt | R[rd] *←* R[rs] *⊕* R[rt] | 1 cycle |

remu rd, rs, rt R[rd] *←* R[rs] mod R[rt] (unsigned) 5 cycles

bne rs, rt, Label Branch to Label if ( R[rs] *ƒ*= R[rt] ) 2 cycles

jr ra unconditional jump to return address specified in R[ra] 1 cycle

|  |  |  |
| --- | --- | --- |
|  |  | LATENCY (Cylce) |
|  | xor rd rd rd //zero | 1 |
|  | add rd rs rt //do n=n+a | 1 |
| Loop | add rd rs rt //loop | 1 |
|  | remu rd rs rt //mod | 5 |
| End | bne rs rt L//if mod not = 0, break | 2 |
|  | jr v0 //return | 1 |

Loop= 5 times

* Total bytes = 6 instructions \* 4 bytes = 24 bytes in program
* Instruction count = 5(3) + 2 + = 18 instructions
* Bytes fetched = 4+4+5(3+4)+4 = 72 bytes
* Program latency = 1+1+5(1+5+2)+1 = 43 cycles

## 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. 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 ƒ*= 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 |
| push | x | push x in stack | 1 bytes | 3 cycle |
| popm |  | pop *x*; pop *addr*; M[*addr*] *← x* | 1 byte | 3 cycles |
| rem |  | pop *x*; pop *y*; push *ymodx* | 1 byte | 7 cycles |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Bytes | Cycles |
|  | goto | 5 | 1 |
|  | push x | 1 | 3 |
|  | add | 1 | 2 |
| Loop | add | 1 | 2 |
|  | push | 1 | 3 |
|  | rem | 1 | 7 |
| End | bne | 5 | 1 |
|  | popm | 1 | 3 |

* Bytes = 5+1+1+1+1+1+5+1 = 16 bytes
* Instructions = 3+5(4)\_1 = 24 inst
* Bytes fetched = 5+1+1+5(1+1+1+5)+1 = 48 bytes
* Latency = 1+3+2+5(2+3+7+1)+3 = 74 cycles

## Comparison (8 points)

Compare the three ISAs studied with respect to static code size, throughput, performance, number of instruction bytes fetched during execution. What are the tradeoffs for each ISA. 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.

* 3.4 CPI: P1: 71/41 = 1.73 Throughput P1: 41/71 = .58 P2: 43/18 = 2.4 P2: 18/43 = .42 P3: 74/24 = 3.1 P3: 24/72 = .32
* If all the machines have same ratio of clock period and looking at the program latency, we can see that MIPS would have the highest performance with x86 in 2nd place and stack in the last position.
* X86 has the highest throughput though which means most work done per instruction. This would mean that x86 is best for handling large workloads and heavy data system. It also has the highest clock period.
* X86 runs CISC. MIPS is middle in throughput but makes up for it with a high ratio of clock periods or Latency. MIPS runs RISC. Stacks is the worst in terms of performance out of the three but takes less memory. For smaller workloads, stacks would be the best. Straightforward ISA.