Problem M1.2: CISC and RISC: Comparing ISAs [1 Hours]

This problem requires the knowledge of Handout #2 (CISC ISA—x86jr), Handout #3 (RISC ISA—MIPS32), and Lectures 2 and 3. Please read these materials before answering the following questions.

Problem M1.2.A — CISC

Let us begin by considering the following C code.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, register `%edx` contains `result` and register `%eax` contains `a`. `b` is stored in memory at location `0x08047580.`) A brief explanation of each instruction in the code is given in Handout #2.

```
            xor    %edx,%edx
            xor    %ecx,%ecx
 loop:      cmp    0x08047580,%ecx
            jl     L1
            jmp    done
 L1:        add    %eax,%edx
            inc    %ecx
            jmp    loop
 done:    ...
```

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem M1.2.B — RISC

Translate each of the x86 instructions in the following table into one or more MIPS32 instructions in Handout #3. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed. Assume that upon entry R2 contains a and R3 contains `i`. R1 should be loaded with the value of b from memory location 0x08047580, while R4 should

receive `result`. If needed, use R5 to hold the condition value and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code.

x86 instruction	**label**	**MIPS32 instruction sequence**
`xor %edx,%edx`		
`xor %ecx,%ecx`		
`cmp 0x08049580,%ecx`		
`jl L1`		
`jmp done`		
`add %eax,%edx`		
`inc %ecx`		
`jmp loop`		
`...`	*done:*	`...`

How many bytes is the MIPS32 program using your direct translation? How many bytes of MIPS32 instructions need to be fetched for b = 10 using your direct translation? How many bytes of data memory need to be fetched? Stored?

Problem M1.2.C **Optimization**

To get more practice with MIPS32, optimize the code from part **B** so that it can be expressed in fewer instructions. Your solution should contain commented assembly code, a paragraph which explains your optimizations and a short analysis of the savings you obtained.

Problem M1.3: Addressing Modes on MIPS ISA [1.5 Hours]

Ben Bitdiddle is suspicious of the benefits of complex addressing modes. So he has decided to investigate it by incrementally removing the addressing modes from our MIPS ISA. Then he will write programs on the "crippled" MIPS ISAs to see what the programming on these ISAs is like.

Problem M1.3.A — Displacement addressing mode

As a first step, Ben has discontinued supporting the displacement (base+offset) addressing mode, that is, our MIPS ISA only supports register indirect addressing (without the offset).

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.

`LW R1, 16(R2)` ➔

Problem M1.3.B — Register indirect addressing

Now he wants to take a bolder step by completely eliminating the register indirect addressing. The new load and store instructions will have the following format.

```
LW R1, imm16   ; R1 <- M[imm16]
SW R1, imm16   ; M[imm16] <- R1
```

6	5	5	16
Opcode	**Rs**		**Offset**

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why. (Don't worry about branches and jumps for this question.)

`LW R1, 16(R2)` ➔

Problem M1.3.C **Subroutine**

Ben is wondering whether we can implement a subroutine using only absolute addressing. He changes the original ISA such that all the branches and jumps take a 16-bit absolute address (the 2 lower orders bits are 0 for word accesses), and that `jr` and `jalr` are not supported any longer.

With the new ISA he decides to rewrite a piece of subroutine code from his old project. Here is the original C code he has written.

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i=0; i<b; i++){
    result=result+a;
  }
}
```

The C code above is translated into the following instruction sequence on our original MIPS ISA. Assume that upon entry, R1 and R2 contain `b` and `a`, respectively. R3 is used for `i` and R4 for `result`. By a calling convention, the 16-bit word-aligned return address is passed in R31.

```
Subroutine: xor  R4, R4, R4   ; result = 0
            xor  R3, R3, R3   ; i = 0
loop:       slt  R5, R3, R1
            bnez R5, L1       ; if (i < b) goto L1
return:     jr   R31          ; return to the caller
L1:         add  R4, R4, R2   ; result += a
            addi R3, R3, #1   ; i++
            j    loop
```

If you can, please rewrite the assembly code so that the subroutine returns without using a `jr` instruction (which is a register indirect jump). If you cannot, explain why.