CS220: Lab#9B

1. [10 marks] Implement an instruction memory that has width 32 bits and has 7 rows. Initialize the contents of the memory using the following MIPS instruction sequence translated from the C statements shown alongside. All numerical values are represented in decimal in the following program. Each row of memory will store the binary encoding of one MIPS instruction. Use MIPS instruction encoding as discussed in the class and mentioned in the text book. Use an `initial` block to store the instructions in instruction memory.

```
C statements                    MIPS translation
--------------------------------------------------------------------------


int a, b, c, d, e;
a = 45;                     addiu $1, $0, 45   // opcode: 0x9
b = -20;                    addiu $2, $0, -20
c = -60;                    addiu $3, $0, -60
d = 30;                     addiu $4, $0, 30
e = (a + b) - (c + d);      addu  $5, $1, $2  // opcode: 0x0, function: 0x21
                            addu  $6, $3, $4
                            subu  $5, $5, $6  // opcode: 0x0, function: 0x23
```

We will design a simple MIPS processor that can execute addition and subtraction instructions (addu, addiu, subu) and ignores all overflows. The processor will have a register file having 32 registers each of width eight bits. Initialize all registers to zero. In all I-format instructions, the least significant eight bits of the 16-bit immediate operand will be used in the actual operation. Initialize a program counter register to zero. The MIPS processor is implemented as a six-state FSM as outlined below. Initially, the state is zero. Each state's operations are done on posedge of clock.

- State 0: reads the instruction from the memory row pointed to by the program counter, increments the program counter by one, and sets state to 1.
- State 1: finds out the fields of the instruction and sets state to 2. This is known as decoding the instruction.
- State 2: reads the source register operands of the instruction from the register file and sets state to 3.
- State 3: executes the instruction in an eight-bit adder/subtracter if the instruction is addiu, addu, or subu; otherwise marks the instruction as invalid. Sets state to 4.
- State 4: if the instruction is not marked invalid and the destination register is not $0, writes the result of the instruction to the destination register. Sets state to 0 if program counter is less than `` `MAX_PC``; otherwise sets state to 5. `MAX_PC` should be defined as the maximum number of instructions to execute.
- State 5: shows the contents of register `` `OUTPUT_REG`` in the LEDs (LED7 is the most significant bit) and stays in state 5. `OUTPUT_REG` should be defined as 5 for this program because $5 will have the value of `e`, which is of interest to us.