CS220: Lab#10A

1. [10 marks] Implement an instruction memory that has width 32 bits and has 14 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 an `initial` block to store the instructions in instruction memory. Note that the MIPS translation grossly violates the MIPS function calling convention in this problem, but the translation will generate correct result.

```
C statements                    MIPS translation
-------------------------------------------------------------------------------
int array[10];
int n, x;

int Sum (int n)
{
   // n is in $1
   int i, sum;
   // sum is in $2
   // i is in $3
   sum = 0;                  Sum: addiu $2, $0, 0    // opcode: 0x9
   for (i=0; i<n; i++) {          addiu $3, $0, 0
      if (i == 10) break;         slt   $4, $3, $1   // opcode: 0x0, func: 0x2a
      sum += array[i];            beq   $4, $0, exit // opcode: 0x4, encode exit as 8
   }                              addiu $5, $0, 10
   return sum;             loop:  beq   $3, $5, exit // encode exit as 6
}                                 lw    $6, 0($3)    // opcode: 0x23
                                  addu  $2, $2, $6   // opcode: 0x0, func: 0x21
                                  addiu $3, $3, 1
                                  slt   $4, $3, $1
                                  bne   $4, $0, loop // opcode: 0x5, encode loop as -5
                           exit:  jr    $31          // opcode: 0x0, func: 0x8, rs: 31
n = 8;                            lw    $1, 10($0)
x = Sum (n);                      jal   Sum          // opcode: 0x3, encode Sum as 0
```

Implement a data memory of width eight bits and having 11 rows for storing `array[10]` in the first ten rows and `n` in the last row. Use an `initial` block for storing these in eight-bit two's complement representation in the data memory. We will design a simple MIPS processor that can execute all instructions shown in this program (addiu, slt, beq, lw, addu, bne, jr, jal) and ignores all overflows. The "word" for this processor is eight bits long. 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 an eight-bit program counter register to 12 (address of the instruction corresponding to `n = 8`). In beq and bne, the program counter of the branch target should be computed by adding the least significant eight bits of the offset to the program counter of the branch instruction. In lw, the computed address should be treated as the row number of data memory. In jal, the address of the call target is same as the least significant eight bits of the target field of the instruction. The MIPS processor is implemented as a seven-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 instruction memory row pointed to by the program counter and sets state to 1.
- State 1: finds out the fields of the instruction and sets state to 2.
- State 2: reads the source register operands of the instruction from the register file and sets state to 3.
- State 3: executes the instruction if the instruction is addiu, addu, slt, beq, bne, jal, or jr; if the instruction is lw, its address is computed; otherwise marks the instruction as invalid. Sets the program counter of the next instruction appropriately. Sets state to 4.
- State 4: accesses data memory if the instruction is lw and reads the row pointed to by the address computed in the last state; other instructions do nothing in this state. Sets state to 5.
- State 5: if the instruction is not marked invalid and produces a result in a destination register 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 6. `MAX_PC` should be defined as 14.
- State 6: shows the contents of register `` `OUTPUT_REG `` in the LEDs (LED7 is the most significant bit) and stays in state 6. `OUTPUT_REG` should be defined as 2 for this program because \$2 will have the value of `x`, which is of interest to us.

Your design should work for arbitrary initial values of `array` elements and `n`. While implementing the slt instruction, the comparison operands should be treated signed. Therefore, instead of writing "a $<$ b", your Verilog code should say "\$signed(a) $<$ \$signed(b)" where \$signed is an in-built Verilog function to convert unsigned variables (default for all variables in Verilog) to signed.

Note to TAs for grading: Please test the implementation by initializing the array and `n`. At the time of grading, give your initial setting to the group and ask them to appropriately change the data memory contents in their code. Note that even if you set `n` to be more than 10, only ten elements of the array will be considered in the computation. Please have a few of the array elements as negative (you can go up to -128 on the negative side and 127 on the positive side). Also, note that the sum must be in the range -128 to 127. So, choose your data memory contents accordingly to avoid overflow. Please have different initial settings when evaluating different groups.