Add states for the following instructions to the C.U. (for the myth8) that you designed for Project 1:

1. \texttt{[add\_ext blank\_blank\_blank\_blank]}

   Semantics: \[ r_1r_0 = r_1r_0 + r_3r_2 \] where \( r_1r_0 \) means the 16-bit number formed by concatenating \( r_1 \)'s data to \( r_0 \)'s data with the latter being the LS-byte and the former the MS-byte. \( r_2r_3 \) is similarly defined.

   \texttt{add\_ext\ opcode} = 001111

2. (a) Design the format of a 24-bit instruction to perform a load word \texttt{lw+} using the \textit{auto-increment register-indirect} addressing mode with the semantics \( r_i \leftarrow \text{Mem}[r_j]; r_j \leftarrow r_j + X \). Note the initial (non-incremented) value of \( r_j \) is to be used for the address of the data word to be obtained from memory.

   Clearly explain the format and what it means. Next, design the CU states for executing the \texttt{lw+} instruction.

   \texttt{lw+\ opcode} = 010000

   \textbf{Hint:} Just like in the \texttt{sw} instruction from Project 1, the 1st or least significant 2 bytes will have the opcode and other information that the CU needs immediately to start processing the first part of the instruction \( (r_i \leftarrow \text{Mem}[r_j]) \). The last (most significant) 8 bits of the instruction can be fetched after the initial part is done and loaded into an appropriate register (mdr, ir0, ir1, etc.) so that it enables the 2nd part of the instruction \( (r_j \leftarrow r_j + X) \) to be completed.

   (b) Comment on the speed efficiency of using the 24-bit \texttt{lw+} instruction above versus using two 16-bit instructions \texttt{load\_ind} and \texttt{add\_imm} (you did these in Project 1) to accomplish the same effect as the \texttt{lw+} instruction.

3. \texttt{[mul\_slow \( r_i \) \( r_j \) \( r_k \)]} – type 1

   Semantics: \( r_i \leftarrow r_j \times r_k \) using the slow multiplication technique given below (repeated addition with no shifting). Note that either of \( r_j, r_k \) could be negative.

   \texttt{mul\_slow\ opcode} = 010001
4. \( \text{div\_slow } r_i r_j r_k \) – type 1

Semantics: \( r_i \leftarrow \lfloor \frac{r_j}{r_k} \rfloor \) using the slow division technique given below (repeated subtraction with no shifting). Note that either of \( r_j, r_k \) could be negative.

\text{div\_slow opcode} = 010010

5. \( \text{div\_fast } r_i r_j r_k \) – type 1

Semantics: \( r_i \leftarrow \lfloor \frac{r_j}{r_k} \rfloor \) using the fast non-restoring division technique (see lecture notes on “Iterative Division”).

\text{div\_fast opcode} = 010011

For division, also assume that the \( D \) and \( V \) specified in \( r_j, r_k \) are unsigned integers represented in 7 bits (MSB–bit 7– is 0, while the numbers are represented in bits 6 to 0). This is needed in order to get the negative \( V \) (for subtraction) in 2’s complement that is representable in 8 bits (this is automatically done by choosing \( \text{alu\_sel} = \text{SUB} \) so you don’t have to worry about it).

For the (slow) multiplication instruction use the following method for performing \( c = a \times b \):

1. \( c = 0; \text{sign} = 0 \)
2. If \( b < 0 \) then \{ \( b = -b \) /* convert \( b \) to a +ve # */
   sign = not(sign) \}
3. if (\( b == 0 \)) then goto step 5 /* final result is in \( c \) */
   else \( c = c + a \)
4. \( b = b - 1; \) goto step 3.
5. if (\( \text{sign} = 1 \)) then \( c = -c \) /* restore correct sign to the result */

For the slow division instruction use the following method for performing \( c = \lfloor \frac{a}{b} \rfloor \) (we assume \( b \neq 0 \)):

1. \( c = 0; \text{sign} = 0 \)
2. If \( b < 0 \) then \{ \( b = -b \) /* convert \( b \) to a +ve # */
   sign = not(sign) \}
3. If \( a < 0 \) then \{ \( a = -b \) /* convert \( a \) to a +ve # */
   sign = not(sign) \}
4. \( a = a - b \)
   If (\( a < 0 \)) then goto step 6 /* quotient is in \( c \) */
   else \{ \( c = c + 1; \) goto step 4 \}
5. if (sign=1) then \( c = -c \) /* restore correct sign to the result */

**Important Note:** You should not change the values in the user registers (particularly those specified by the \( r_j \) and \( r_k \) fields) specified by the instruction. You can first copy them into the scratch-pad registers (\( r_4 \) to \( r_6 \)) if their values need changing during the processing of the instruction.

Also, debug your design using sample instructions that you put in a .mem file.

On the date submit your debugged and tested control program along with the .mem files with your format of the \( lw+ \) instruction inserted in the 24-bit place holders in the .mem file the we will provide you.

Theoretically analyze the **maximum** and the **average** # of cc’s for executing each instruction (assume memory read/write takes 3 cc’s). Show your work clearly, and submit in a tabular form.

You will be given one or two machine language programs (.mem files) to test your C.U. design. Report the initial and final states of the mythsim graphical interface by printing it out (with all register values etc.) and the number of clock cycles required to execute each program.

**The grading criterion will be similar to that in Project 1**  
60% of project weight