Overall ENEE350 Project Objective
Your project this semester is to design a very basic processor. You will simulate this processor in C. Over the course of three project parts, you will create and encode an instruction set, you will design a datapath to implement the instruction set, and you will encode your datapath into C. Your final task will be to “compile” a C program that will be given in Project 3 into your processor’s machine code, and execute that code on your simulated processor.

Teamwork
You are expected to complete this project in 3-4 member groups. It is your responsibility to determine your group. Except under unusual circumstances, individual projects will not be acceptable.

Objective
The first stage of the project is to design and encode an instruction set for your processor. Your instruction set must be “universal” -- any algorithm must be programmable with your instruction set. To complete this stage, you must do the following:
• List all of your registers
• List all of your instructions, and indicate what operands each instruction requires
• Create a unique numerical encoding for each instruction -- a machine code for your processor, and list out the encoding
• Translate the C program given in this project into your assembly language and into your machine code, and turn in both
• Trace out 2 iterations of your assembly version of the C program on paper, and indicate at each step what values are in what registers

Limitations
In determining your instruction set, you must follow the following limitations
• Your instruction set must be capable of running the C programs given
• You must have at least two and no more than eight registers, not including a program counter. Your registers should each hold eight bits (1 byte).
• Your entire instruction set must be encoded in eight bits. All of your instructions should thus be 1 byte long. This should be a major limitation in determining your instruction set.
• You are not permitted to have more than 20 instructions (unique opcodes)
• Your processor must be able to access 256 bytes of memory, where each memory entry is 1 byte long.
• Your processor must include a program counter register which will hold the memory
address of the next instruction to be executed. Whether you decide to make it a
general purpose register or make it implicit to branch commands like in MIPS is up to
you.
• Do not concern yourself with kernel-mode features (I/O, external devices other than
memory, interrupts, or system calls).
• Assume that all programs on your computer will start at address 00 in memory.
• You should assume that all values in your registers and memory are in the two’ s
complement number system (0x02 is +2, 0xFE is -2).
• Be creative! Creative instructions, accompanied by explanations explaining why you
decided to include them, may result in bonus points!

Determining your instructions
Your instruction set should be able to run the C program given, as well as similar C
programs. Your instruction set must therefore include one or more instructions from each
of the language families described below.

Arithmetic instructions
Your processor must be capable of adding and subtracting two numbers. Whether the
numbers to be added are in registers like in MIPS or in memory (or a combination of the
two) is up to you. You could have an ADD command and a SUB command, or an ADD
command and a NOT command -- it’s up to you.

Other arithmetic commands you may wish to consider:
• Logical instructions, such as AND, OR, XOR, NOT...
• Shift instructions, such as Logical Shift Left or Rotate Right
• Immediate versions of ADD and SUB, such as ADDI and SUBI
• More complex arithmetic, such as MULTIPLY
It is not necessary that you implement any of the above instructions. But be sure to
consider how they can be implemented with ADD and SUB.

Memory access instructions
Your processor must be capable of moving data between registers and memory. Your
instruction set must include the ability to do indirect loads and stores -- such as LOAD r1,
(r2) or STORE r3, (r4). Whether you choose to have specific commands for load and
store (as is recommended), or somehow combine them with other commands like ADD is
up to you.

Immediate instructions
You must include some means of getting an immediate value into a register. This will
not be as easy as having a command like LI r1, 48. The problem is that the register size
is the same as the instruction size, so that an 8 bit immediate value will take up the entire
instruction. You may choose to handle this in a variety of ways. One possible solution is
to have two instructions LUI (Load Upper Immediate) and LLI (Load Lower Immediate)
that accept 4 bit immediate values. LUI puts the immediate in the upper four bits of the register and LLI puts it in the lower four bits. Alternatively, you could choose to combine these commands with ADD or SUB (e.g. ADDI and SUBI).

Branch instructions
Your instruction set must include a conditional branch instruction that causes execution to pass to a specified memory address if a criteria is true. A simple instruction could be BEQZ, which branches to an address if a given register is equal to zero. Additionally, your instruction set must include an instruction like JR that jumps to an address contained in a register. It is recommended that you combine BEQZ and JR into an instruction like BEQZR rx,ry which branches to ry if rx equals zero.

Saving the program counter
You should have an instruction that allows the program counter’s value to be saved (like JALR does in MIPS). One possibility is to treat the program counter as if it were a general purpose register. Alternatively, you could have a JALR style instruction that saves the PC’s value into a known register.

Encoding your instruction set
You must determine a unique 8-bit encoding for each instruction. The encodings must have space to hold immediate values and registers. This is likely to be your biggest limitation in determining your instruction set.

An encoding for a particular instruction will generally follow the form
<bits to identify opcode, register bits, immediate bits> where the total number of bits must be 8.

Example
Suppose you want to have 8 general purpose registers (r0..r7). You want to implement an ADD command like in MIPS:

ADD rd, rs, rt  (add rs+rt and put the result in rd)

Your encoding would thus look like:

<bits to identify ADD, bits to identify rd, bits to identify rs, bits to identify rt>

if there are 8 general purpose registers, you need 3 bits (2^3=8) to identify each register. Consequently you will need at least 9 bits (3 for rd, 3 for rs, 3 for rt) to encode the ADD instruction. The ADD instruction is thus impossible to encode in 8 bits.

What can you do about this? There are several possibilities. For example you could have only 4 registers (r0..r3). In this case you need 2 bits to identify a register, or 6 bits to
encode all of the register fields in ADD. This will leave 2 bits to determine a unique id for ADD.

Alternatively, you could have an ADD command that works like this:

ADD rd, rs  (add rs+rd and put the result in rd)

This requires only two register fields, or 6 bits. It’s easy to show that this second ADD instruction, combined with a similar SUB instruction, can do everything the MIPS ADD instruction can do. For example, if you wanted to add r1+r2 and put the result in r3, you could do the following:

```
SUB r3, r3  r3 = r3-r3 = 0
ADD r3, r1  r3 = r3+r1 = 0+r1 = r1
ADD r3, r2  r3 = r3+r2 = r1+r2
```

You could encode this ADD as follows. Pick a unique 2-bit id for ADD, like 01. Your ADD encoding could look like:

```
<0 1 rd2 rd1 rd0 rs2 rs1 rs0>
```

Example
Suppose you have 4 registers and you want to encode the following instructions:
ADD rd,rs
SUB rd,rs
LUI rd, imm       (where the imm field is 4 bits)

You could encode LUI first. Since the rd field is 2 bits, and the imm field is 4 bits, you have only 2 bits to encode the LUI opcode. You might choose the encoding 00. Thus LUI would translate to:

```
<0 0 rd1 rd0 imm3 imm2 imm1 imm0>
```

Next you could encode ADD. Since rd and rs can each be encoded in 2 bits, you have 4 bits to encode the ADD opcode. However, you cannot choose a 4-bit encoding for ADD that begins with 00, because that is already used by LUI. You might choose 0100. ADD would thus translate to:

```
<0 1 0 0 rd1 rd0 rs1 rs0>
```
Finally you could encode SUB. You can encode the SUB opcode in 4 bits as in ADD, but it cannot begin with 00 because of LUI and it cannot be 0100 because of ADD. Consequently, you could choose 0101. Your SUB instruction could be encoded as:

\[<0 \ 1 \ 0 \ 1 \ rd1 \ rd0 \ rs1 \ rs0>\]

**Encoding requirements**

When you determine your encoding for each instruction, you should turn in a list of instructions, explain briefly what each instruction does, and give an encoding for each instruction. For example, if you were to use the example above, you would turn in the following:

ADD \( rd,rs \)       Add \( rs + rd \) and put the result in \( rd \)
\[<0 \ 1 \ 0 \ 0 \ rd1 \ rd0 \ rs1 \ rs0>\]

SUB \( rd,rs \)       Subtract \( rd - rs \) and put the result in \( rd \)
\[<0 \ 1 \ 0 \ 1 \ rd1 \ rd0 \ rs1 \ rs0>\]

LUI \( rd,imm \)       Put the 4 bit immediate \( imm \) into the upper 4 bits of \( rd \)
\[<0 \ 0 \ rd1 \ rd0 \ imm3 \ imm2 \ imm1 \ imm0>\]

**Compile a C Program**

You should translate the following C program into your assembly language. Please annotate each instruction in your assembly code.

```
int x,n,i;
  n=3;
  for (i=0; i!=n; i++)
    x=x+i;
```

Feel free to use registers for \( n \) and \( i \). Assume that \( x \) is in memory entry \( A0 \) hex. At the end of your program, branch to address 00 hex.

After writing your assembly code, assemble it into your machine code. Assume the program begins at memory 00 hex, and use absolute addresses instead of labels. When you turn in your assembly code, turn in the machine code equivalent next to each instruction.

The code you turn in should be in the following general format (of course, the instructions and encoding will be your own):

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Instruction</th>
<th>Assembly Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>47</td>
<td>ADD ( r1, r2 )</td>
<td>( x=x+i )</td>
</tr>
<tr>
<td>06</td>
<td>30</td>
<td>LUI ( r3,0 )</td>
<td>( r3' ) upper bits are 0</td>
</tr>
<tr>
<td>07</td>
<td>21</td>
<td>LLI ( r3,1 )</td>
<td>( r3 = 1 )</td>
</tr>
</tbody>
</table>
Trace out your program
You should trace out 2 iterations of the for loop on paper and show that your assembly code works. Show how the relevant register values change after each instruction.