CSCE312: Computer Organization

Possible design steps for final project

For two given matrices, the summation of these two matrices is

\[
\begin{pmatrix}
a_1 & a_2 \\
a_3 & a_4 \\
\end{pmatrix} + \begin{pmatrix}
b_1 & b_2 \\
b_3 & b_4 \\
\end{pmatrix} = \begin{pmatrix}
a_1 + b_1 & a_2 + b_2 \\
\end{pmatrix}
\]

You are building a processor that can perform the matrices summation of two 2*2 matrices. The result should be stored in RAM. You may choose where you want to store the input coefficients (RAM or ROM).

Remember: You may have a maximum of 8 general purpose registers in your processor. A general purpose register is one that you can address through your program such as the eax-edx registers in Y86. This does not include internal system registers such as the program counter or status flags.

1. Figure out what assembly instructions you would need to perform the basic steps of computation. You might need instructions to add values of two registers (ADD), move data between RAM, ROM and the processor (LOAD, STORE). Don’t forget to implement some sort of HALT instruction.

2. Based on the instructions you created, write the program in your assembly language to perform the matrices summation computation. Make your code as optimized as possible. Assign each instruction an appropriate op-code.

3. Plan your address space between RAM and ROM modules. Since this is an 8 bit processor, you may have 255 address locations in total. However, your address space may be limited by instruction width. I would suggest you to separate your instructions and data. Load instructions to ROM and load matrices data to RAM. This would simplify your logic. E.g. When you load/store data, you always go to RAM and your PC value always sent to ROM to fetch instructions.

4. Based on the op-codes you assigned in step-2 and your address space from above, translate your assembly language program into hex/binary.

5. Note that while your processor is an 8-bit processor, your instructions can be longer than 8 bits if necessary. Look at the Y86 design for reference on how it handles instructions that range from 1-byte to 6-bytes long.

6. Look at the design of the various functional stages of the Y86 processor. Identify how the instruction codes are used to activate various functions.

7. Design each stage of your processor in Logisim:
   a. Use a structure similar to that of the Y86; create independent Fetch, Decode, Execute, Memory and Write-back / PC-Update stages.
   b. Use sub-circuits in Logisim to clearly demarcate the various stages. This will make debugging your design much easier than if you try to cram everything onto a single stage.
   c. Integrate all the modules by connecting them together.
8. Load your program in hex form and matrices data to memory.
9. Verify that your processor performs the correct operations.

Other design notes:

1. When integrating multiple modules in Logisim – use a single common clock to the maximum extent possible. A lot of unexpected design issues can arise if you use multiple clocks and they are slightly out of sync with each other.
2. Since this is an 8-bit processor, each memory location (and in turn your data bus) can be a maximum of 8-bits (1-byte) wide. Your instructions may be longer (1-6 bytes wide), however, you can only fetch 8 bits at a time. Hence if an instruction is 2 bytes long, you will need two cycles to fetch the instruction from ROM. Keep this in mind while you assign op-codes.
3. While you have access to 8-bit address space, you probably don’t need to be using all that. Look at ways to reduce the address space you will be actually using and trim your instruction width accordingly.
4. Stick with one single version of Logisim between all the project members and all machines in use. I’ve seen lots of small bugs creep up when team members use different version of Logisim while working on the same design. Especially when sub-circuits and/or memory modules are created in different versions and combined together.