CSCI 2500 — Computer Organization
Lab 06 (document version 1.0)
Decoding the Instructions

- This lab is due by the end of your lab session on Wednesday, November 10, 2021.
- This lab is to be completed **individually**. Do not share your code with anyone else.
- You **must** show your code and your solutions to a TA or mentor and answer their questions to receive credit for each checkpoint.
- Labs are available on Mondays before your lab session. Plan to start each lab early and ask questions during office hours, in the Discussion Forum on Submitty, and during your lab session.

The purpose of this lab will be to set you to get thinking about the group project. You'll be implementing three separate functions relevant to the project. The first being a function that will convert MIPS assembly instructions into machine code. The second being a 5-to-32 decoder, which will be useful for implementing your memory circuits. Finally, the third function will be

Fill in the provided template code given in `lab06.c`. This has empty functions for all of the checkpoints and instructions on what to implement. You can add helper functions as necessary.

1. **Checkpoint 1:** The first checkpoint will involve converting MIPS assembly instructions into their machine code format. You'll be considering one R-type, I-type, and J-type instructions. Specifically, your converter will need to work with `add`, `lw`, and `j` MIPS instructions. See the MIPS reference card on Submitty for the instruction format fields, the op codes, function codes, and register mappings (rs, rt, rd) for each instruction. Note that the syntax for MIPS instructions that we will use here will be a simplified version of the real MIPS syntax. Likewise, you will only need to consider `t0` and `s0` registers for now.

   For example, when considering instruction:

   ```
   add t0 t0 s0
   ```

   This should convert to machine code format:

   funct: `100000`
   shamt: `00000`
   rd: `01000`
   rt: `10000`
   rs: `01000`
   opcode: `000000`
   Instruction: `00000001000100000100000000100000`

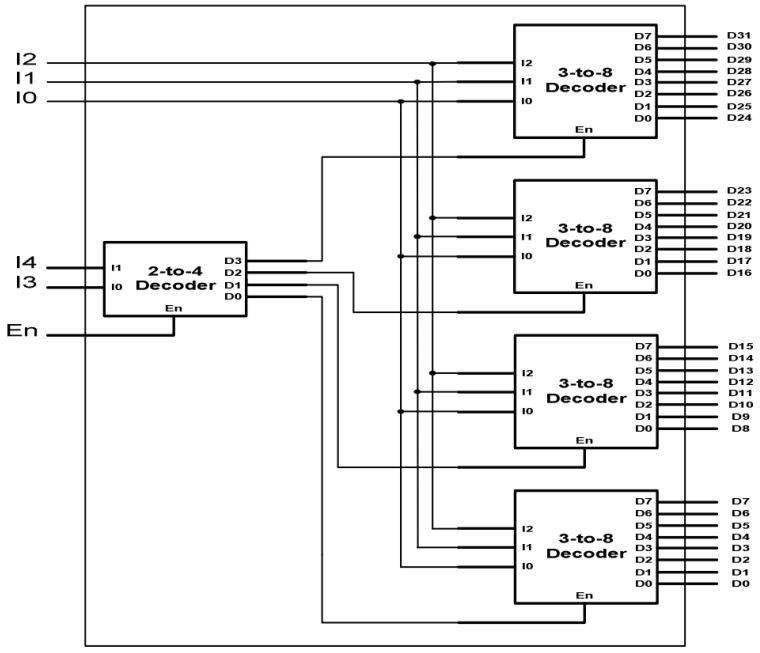
   For simplicity with parsing, our input format for `lw` and `sw` instructions is going to be slightly different than what we've seen. For example, lw instructions are going to be:

   ```
   lw t0 s0 constant
   ```

   Where constant is our offset in the immediate field, s0 is the base address in rs, and t0 is the register that we'll be writing to in rt. Likewise, sw instructions will have the same format, except t0 in this instance would be the register whose value we're saving to memory.

2. **Checkpoint 2:**

   For this checkpoint, you'll be building a 5-to-32 decoder using our C logic gates. This will be useful for building your memory access circuits within the project. While it can be unwieldy to directly implement such a decoder, there are multiple approaches that can just use the 2-to-4 and 3-to-8 decoders that we've already discussed and/or have actually implemented. One such possible design is given below[1]:



   Note that this design, as is often typical of decoders in practice, has an "Enable" input line. Consider this as similar to the "write" lines we've considered in relation to D flip-flops and memory. When `Enable` is set to logical false, the decoder outputs false on all lines. When `Enable` is set to true, the decoder functions as normal. An easy way to implement this functionality is to just include an additional AND gate for each output along with `Enable`.

3. **Checkpoint 3:**

   For this checkpoint, you'll be building upon the 1-bit ALU circuit you developed for Lab 5. Specifically, you'll be using your logic gates to implement circuits for a 32-bit ALU.

   The input/output as well as intermediate format for storing integers will be an extension of the `BIT` type we used for Lab 5. 32-bit 2's complement numbers will be stored as binary in an array of 32 `BIT`s. For these arrays, the most significant bit is at index 31 and the least significant bit is at index 0. The first function you'll need to implement is probably `convert_to_binary`, to convert the input decimal integers to their BIT representations.

   See slides "New 1-bit ALU", "MSB ALU", and "New 32-bit ALU" of csci2500-f21-ch03a-slides.pdf for logical diagrams of the 1-bit ALU and 32-bit ripple ALU that you'll be implementing. The 1-bit ALU will extend the Lab 5 ALU to also include functionality for set-less-than (slt). You can use a `for` loop here to simplify the 32 'ripples'.

[1] https://fci.stafpu.bu.edu.eg/Computer%20Science/4887/crs-12801/Files/hw4-solution.pdf