CS2100 Computer Organization
Tutorial 5: Control Path
(Week 6: 2 Oct – 6 Oct 2023)

For this tutorial, we will keep it simple and (like the textbook) just assume a MIPS processor that can handle only R-type, **`lw`**, **`sw`**, and **`beq`** instructions.

1. So far you have learnt C as well as MIPS programming. You are also starting to see the hardware implementation aspect of things. We will soon be starting on the journey to see how hardware implementations are done but whatever hardware does, it can be *described* in software. First, we need some preliminaries. There is a header file in C, called stdint.h, introduced by the C99 standard, that allows programmer to specify the exact bitwidths of integers. Among other things, it introduces the data type **`int8_t`**, **`int_16_t`**, **`int32_t`**, **`int64_t`**, **`uint8_t`**, **`uint_16_t`**, **`uint32_t`**, and **`uint64_t`** for signed 8, 16, 32, and 64 bit integers and *unsigned* 8, 16, 32, and 64 bit integers, respectively. In hardware description languages (like Verilog and VHDL – but not C), arbitrarily lengths are also supported. For the purpose of this tutorial, let's assume there is a **`int<N>_t`** and **`uint<N>_t`** type in C that also supports arbitrary length **`N`** signed and unsigned integers. As with **`int8_t`** etc, we shall trust that the compiler will "do the right thing" such as selecting the right assembly instructions to accomplish the operations required. Note that (just as a convention) **`N`** must be greater than 1. If **`N`** = 1, we will use **`bool`** (Boolean) data type instead.

   Going back to what is described earlier, we can use software (specifically, C in our case) to describe hardware and its function. So let's start with some of the components of the CPU. We can described instruction memory as an array:

```
uint32_t instruction_memory[4294967296];
```

   Yeah, the array size is huge but we are only doing a description here. Dealing with this large range in reality requires the help of hardware and the operating system, which you will gradually discover in your journey through SoC.

   Now:

   a) Write a C data structure and a function to describe the register file and its operation.

   b) Write a C data structure and a function to describe the data memory and its operation.

2. Next, let's deal with multiplexing (labelled as "MUX" in the slides). Ideally, we want to use some kind of template feature to describe it but C does not support templates (of course C++ does, but that's another story.) So show how two-way multiplexors ("MUX") of different bitwidths can be instantiated using the macro expansion facility of C. In other words, each call to the macro should yield a C function that implements a two-way

multiplexing function (i.e., "if selection is 0, output is input0, and if selection is 1, output is input1) where the inputs and output are of bitwidth **N**.

In particular, the following call to the macro:

```
MUX(32);
```

will instantiate a function definition:

```
int32_t mux_32(bool ctrl, int32_t in0, int32_t in1)
{
    if (ctrl) return in1;
    else return in0;
}
```

Assume the default is signed integer types. We can have another macro that instantiate unsigned integer types.

To do this question successfully, you will need to read up on C's macro *token pasting* feature: https://www.geeksforgeeks.org/stringizing-and-token-pasting-operators-in-c/.

3. Using C's **struct** data types, define a data type that will hold an instruction (let's call it "**struct insn**") of R-, I-, and J-type along with its sub-fields.
4. Using **struct insn** that you defined above, write the C functions that will compute the respective control signals:
   a) **bool RegDst(struct insn);** // Computes the RegDst signal
   b) **bool ALUSrc(struct insn);** // Computes the ALUSrc signal
   c) **bool MemRead(struct insn);** // Computes the MemRead signal
   d) **uint2_t ALUOp(struct insn);** // Computes the ALUOp signal
   e) **uint4_t ALUControl(struct insn, uint2_t _ALUOp);** // Computes the 4bit ALUControl signal

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

5. Write a C function that will model the behavior of the ALU having the following function prototype:

```
int32_t ALU(int32_t in0, int32_t in1,
            uint4_t ALUControl, bool *ALUiszero);
```

where **in0** and **in1** are the 32-bit inputs, **ALUControl** is the 4-bit ALU control signals, and the outputs are the **ALUiszero** bit (passed by pointer) and the 32-bit result.