Jetic Gū

1. Handwritten submissions and proprietary formats (e.g. Pages or MS Word) **will not be graded**.
2. Mathematical expressions must be written <u>entirely</u> using LaTeX, otherwise **50%-100%** of marks will be deducted.
3. Circuits must be **tested**. Untested circuits will receive 0.

Submission File structure:

```
submission.zip
    — circuit1.cct
    — circuit2-1.cct
    — circuit2-2.cct
    — circuit3.cct
    — csci250.clf
```

Circuit file 1 is 4pt; 2-1, 2-2, 3 are 2pt each.

Lab 4

1. CMP component. CMP is a special instruction that compares two numbers, and save the result in a special flag register, so it can be used for conditional branching. In ARM, there are 4 flags, however in our ARM16 implementation, we ask you to really only implement using 3 flip-flops for `N`, `C`, `Z`, while the `V` flag can be ignored by value-fixing it with 0. You should implement this as a component, save in your library and show it tested as `circuit1.cct`
    1. IO specifications
        1. Input: 16bit `D16 Rn_data;`
        2. Input:16bit `D16 Rn_data;`
        3. Input: 4bit condition code `cond;`
        4. Input 1bit enable `E;`
        5. Input 1bit `CLK;`
        6. Output: `F`, dependent on `cond` and internal states `N, C, Z, V.`
    2. Behaviour
        1. When `E == 0`, Internal flip-flops' values (`N, C, Z`) do not change.
        2. When `E <= 1`, perform comparison, update `N`, `C`, `Z` at `CLK`.
        3. Condition code and internal state / `F` relationship is as follows:

cond	Label	Condition for F to output 1
0000	**EQ**	Z == 1 (Equal)
0001	**NE**	Z == 0 (Not Equal)

cond	Label	Condition for F to output 1
0010	**CS**	C == 1 (Unsigned greater or equal)
0011	**CC**	C == 0 (Unsigned lesser)
0100	**MI**	N == 1 (Negative)
0101	**PL**	N == 0 (Non-negative)
0110	**VS**	V == 1 (There was overflow)
0111	**VC**	V == 0 (There wasn't overflow)
1000	**HI**	C and notZ (Unsigned greater than)
1001	**LS**	notC or Z (Unsigned lessor or equal)
1010	**GE**	(N and V) or (notN and notV) (greater or equal)
1011	**LT**	N xor V (less than)
1100	**GT**	notZ and GE (greater than)
1101	**LE**	Z or LT (less than)
1110	**AL**	Always (Condition is always met)
1111	**NV**	Never (Condition is never met)

2. Memory to Data bus controller. This is a controller for your memory module, for when it is connected to the main data bus.
    1. Here are the IOs for this device:
        - Bidirectional: 16bit `D16`, `db`
        - Input: 16bit (optionally, you can use `D16`) `DO`, coming from the memory module.
        - Output: `WE`, going into the memory module
        - Output: 16bit `DI`, going into the memory module.
        - Input: 1bit `db_dir`
    2. Behaviour
        - When `db_dir == 0`, `DI` <= `db`;
        - When `db_dir == 1`, `db` <= `DO`;
        - At any given time, `WE <= db_dir`.
    3. Save this component as `16bit mem_ctrl`, show it tested alongside your memory component as `circuit2-1.cct`.
3. Data bus controller. This is a controller for your bidirectional data bus, using which you can control the direction in which information traverses. More specifically, the following functionalities need to be supported:
    1. The ability to direct ALU output to the register array;

2. The ability to send a register value to the data bus;
3. The ability to send the value on the data bus to the register array.

1. Here are the IOs for this device:
   - Bidirectional: 16bit `D16`, `db`
   - Input: 16bit `D16`, `ALU`, computational result from ALU
   - Input: 16bit `D16`, `Rx_data`, from Register array
   - Input: 1bit `db_dir`
   - Output: 16bit `D16`, `Rd_data`
2. Behaviour:
   - When `db_dir` == 0, `db <= Rx_data`; `Rd_data <= ALU`
   - When `db_dir` == 1, `Rd_data <= db`;
3. Requirement
   - Save this component as `16bit db_ctrl` in your library
   - Show this component tested alongside your memory module and `mem_ctrl`, and in `circuit2-2.cct`.

4. Register Array Modification
   1. Add an additional input `D3` bus `Rx` and output `D16` bus `Rx_data` to your register array, such that `Rx_data` receives value from register selected by `Rx`.
   2. Add an additional output `D16` named `PC`, that always outputs the value from `Register number 7`.
   3. Modify your register array, such that register number 7 (`PC`) can:
      1. When `Rd == 7`, receives new value from Rd_data;
      2. When `Rd != 7`, instead of not receiving updates, has its value increased by 1.
      3. The above should only occur at `CLK` when `Rw == 1` and `Reset != 1`.
      4. Add another D16 output `PC`, always showing the data stored in R7.
   4. Save this component in your library, and show it tested in `circuit3.cct`.