Jetic Gū

1. Handwritten submissions and proprietary formats (e.g. Pages or MS Word) **will not be graded**.
2. Mathematical expressions must be written entirely 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
    — circuit1-1.cct
    — circuit1-2.cct
    — circuit1-3.cct
    — circuit2-1.cct
    — circuit2-2.cct
    — csci250.clf
```

The 1-1 and 1-2 files are 1pt each, 1-3 is 3pt, 2-1 and 2-2 are 2.5pt each. If you used VHDL, you must also submit the VHDL code files (*.dwv).

Lab 3

1. Register Array Design
    1. Implement a `D16`, and a `D3` bus port, save them in your library.
    2. Use `D Flip-Flop En wo/SQ/` to implement a 16bit register, your Input `D` and output `Q` must be implemented using `D16` buses. Show this component (device symbol) tested using HEX keyboards in `circuit1-1.cct`. (1pt)
    3. Implement a 16bit 2-to-1 Multiplexer using `D16` buses as I/O. Show this component (device symbol) tested using HEX keyboards in `circuit1-2.cct`. (1pt)
    4. Use multiple 16bit 2-to-1 Multiplexers to implement a 16bit 8-to-1 Multiplexer. The switch port should be implemented using a `D3`. This component can be tested together with your register array.
    5. Use `Decoder-8 Non Inv.`, and the above devices you just implemented to build a register array with 8 x 16bit registers (3pt).
        1. Inputs: `D16` bus `Rd_data`; `D3` bus `Rn`; `D3` bus `Rm`; `D3` bus `Rd`; 1bit `Rw`, `Reset`, `CLK`;
            1. The `Rw` signal here is used to control the `EN` pin of your `Decoder-8 Non Inv.` so we can prevent the register array from accepting new values when that's desired.
            2. `Rd_data` will carry data that will be stored in the register array, specifically one specified by `Rd` (destination).
            3. `Rn` and `Rm` will select registers for output buses. In ARM architecture, `Rn` is called the base or first operand register, `Rm` is called the optional shift or second operand register.
        2. Output: `D16` bus `Rn_data`; `D16` bus `Rm_data`.

3. You should implement all of the above in `circuit1-3.cct`.

2. ALU design.
    1. You will need to design an arithmetic unit, and a logical unit separately.
    2. For the Arithmetic Unit, it needs to accept as Input, 3bit of `OpcodeB` (not a bus), 8bits of Immediate (`Imm`) (not a bus), 2bits `Mode` code (not a bus), `D16` buses `Rn_data` and `Rm_data`. Finally it produces `D16` bus output `Rd_data`. You can assume all additions and subtractions are unsigned.

OpB	Mode	Instruction	Assembly	See
000	**-**	Logical Shift Left	-	Don't care
001	**-**	Logical Shift Right	-	Don't care
010	**-**	Arithmetic Shift Right	-	Don't care
011	**00**	Addition	`ADDS <Rd>, <Rn>, <Rm>`	`Rd_data <= Rn_data + Rm_data`
011	**01**	Subtraction	`SUBS <Rd>, <Rn>, <Rm>`	`Rd_data <= Rn_data - Rm_data`
011	**10**	Addition (Immediate)	`ADDS <Rd>, <Rn>, #<imm3>`	`Rd_data <= Rn_data + Imm(2 downto 0)`
011	**11**	Subtraction (Immediate)	`SUBS <Rd>, <Rn>, #<imm3>`	`Rd_data <= Rn_data - Imm(2 downto 0)`
100	**-**	Move	`MOVS <Rd>, #<imm8>`	`Rd_data <= Imm`
101	**-**	Compare	-	Don't care
110	**-**	Add 8bits of immediate	`ADDS <Rdn>, #<imm8>`	`Rd_data <= Rn_data + Imm`
111	**-**	Unsigned Subtract 8bits of immediate	`SUBS <Rdn>, #<imm8>`	`Rd_data <= Rn_data - Imm`

The Arithmetic Unit should be shown tested in `circuit2-1.cct`. I will allow this component to either be implemented using VHDL or circuit diagrams.

3. For the Logical Unit, it needs to accept 4bits of `OpcodeB` (not a bus), `D16` input buses `Rn_data` and `Rm_data`, `D16` output bus `Rd_data`.

OpcodeB	Instruction	Assembly	See
0000	Bitwise AND	`ANDS <Rdn>, <Rm>`	`Rd_data <= Rn_data and Rm_data`
0001	Bitwise Exclusive OR	`EORS <Rdn>, <Rm>`	`Rd_data <= Rn_data xor Rm_data`

OpcodeB	Instruction	Assembly	See
0010	Logical Shift Left	-	Don't care
0011	Logical Shift Right	-	Don't care
0100	Arithmetic Shift Right	-	Don't care
0101	Add with Carry	-	Don't care
0110	Subtract with Carry	-	Don't care
0111	Rotate Right	-	Don't care
1000	Test	-	Don't care
1001	Reverse Subtract from 0	-	Don't care
1010	Compare High Registers	-	Don't care
1011	Compare Negative	-	Don't care
1100	Bitwise OR	`ORRS <Rdn>, <Rm>`	`Rd_data <= Rn_data or Rm_data`
1101	Multiply Two Registers	-	Don't care
1110	Bitwise Bit Clear	-	Don't care
1111	Bitwise NOT	`MVNS <Rd>, <Rm>`	`Rd_data <= not Rn_data`

The Logical Unit should be shown tested in `circuit2-2.cct`. I will allow this component to either be implemented using VHDL or circuit diagrams.

3. Notes on LogicWorks

    1. LogicWorks has poor support for `std_logic_vector`, as I have come to find out. Components implemented using `std_logic_vector` as ports are very difficult to work with and buggy.
    2. LogicWorks has no support for `generate` concurrent statements, `when` does work though in concurrent design.
    3. This is the killing blow: I only realised this week that LogicWorks doesn't support using VHDL components as devices in the implementation of other devices. Any attempt at doing that results in the programme malfunctioning. My recommendation: stick to circuit diagrams if you can, but due to the complexity I will allow VHDL to be used for the two ALU devices (`AU` and `LU`) for grading purposes.