Instruction Decoder

This document provides the specifications of the instruction decoder (**decoder**) module. The specifications of the other modules **alu** and **branch** are provided in separate documents.

The **decoder** module takes one input (the 16-bit instruction) and drives twenty-two outputs (see Figure in **processorComponents)**. All logic in the instruction decoder block should be *combinational* and the implementation should be efficient. The full description of the instructions is provided in the document **processorDocs**, but here is a brief description of some of the 22 output signals.

- **alu_func**: the FUNC bits of the 1-input and 2-input arithmetic operations
- **destination_reg**: the RD bits for all operations. This should be the same set of bits regardless of the op-code; all operations that write to the register file use this as the destination register, and the STORE instruction uses these bits as the data register.
- **source_reg1/2**: These bits are consistent across most op-codes. The only exceptions are the branch op-codes – make sure that when assigning the bits for the two **source_reg** outputs that you pay attention to the alignment in the arithmetic op-codes vs the branch op-codes.
- **immediate**: These bits represent an integer to be used directly as an operand. (*e.g.* ADD sums the values in two registers while ADDI sums the value in a register with the 6-bit immediate value specified in the instruction). The immediate bits are always some subset of the 12 least significant bits of the instruction. You should return all 12 bits and let the other modules select the appropriate subset of bits when they use it. Also, with the exception of branch and jump instructions, the immediate field is always treated as an unsigned integer. However, sign extension should *not* be performed in the **decode** module, but should be left to other blocks. So, for instance, although the immediate values used in the branch/jump instructions must be sign extended before use, it is expected that this will occur in the program counter block. In fact, the *only* instructions for which the immediate is sign extended are branches and jumps.

All other outputs are 1-bit Boolean signals indicating either a specific instruction, or a group of related instructions based on a single op-code (e.g **arith_2op** or **arith_1op**) along with additional FUNC bits. Refer to **processorDocs** for further details.

It is important to use the *`define constants* provided at the beginning of the **decoder.v** file. These provide meaningful mnemonic names for specific bit strings. When using the name make sure to include the ` character at the start of the name *e.g.* use `NOP to refer to the bit string 4'b0000 that is the op-code for a NOP instruction. The use of meaningful names rather than bit strings in the code will save you enormous amount of debugging time, besides making your code understandable.

When you have finished your **decoder** block you can test it using the provided testbench **decoder_tb**. Compare with the expected results in the file **results_decoder**

ALU module

The arithmetic logic unit (ALU) is purely *combinational*.

Inputs

- **reg1_data**: The first operand for a 2op instruction, or the only operand for a 1-op instruction, add/subtract immediate instruction, or LOAD/STORE instruction.
- **reg2_data**: The second operand for a 2-op instruction.
- **immediate**: The 6-bit immediate value for the add/subtract immediate instruction, or for the LOAD/STORE address offset.
- **alu_func**: Additional bits that identify an instruction in an instruction group
- **arith_1op/arith_2op/addi/subi**: Each of these 1-bit signals is set to TRUE if and only if it holds for the current instruction op-code
- **load_or_store**: Set to TRUE if the current instruction op-code is either LOAD or STORE.
    - In this case, the **alu_result** should be the same as *addi*, except that the output carry bit should *not* be modified. The ALU is used for load/store instructions to calculate the address in data memory; these instructions allow an immediate offset from a register so that data can be accessed by specifying an offset from a fixed base register.
- **stc_cmd/stb_cmd**: Instruction to set (to 1) the output carry bit or output borrow bit, respectively.
- **carry_in, brrow_in**: 1-bit signals from the **register** module indicating the current value of the Carry and Borrow flags.
    - The carry and borrow flags are used to allow for arithmetic on operands wider than 16 bits (e.g. 32 or 64-bit addition/subtraction). The carry generated by the arithmetic unit is saved in a Carry flag implemented in the **register** module. To incorporate the carry when adding the next 16 most significant bits, an **ADDC** (Add with Carry) instruction includes the current value of the Carry flag in the addition. The value of the Carry flag is provided to the ALU module by the input signal **carry_in**. Similar considerations hold for the Borrow flag.

Outputs

- **alu_result**: The main output of the ALU. This 16-bit value should be the computational result for the current instruction.
- **carry_out**: The carry generated by an arithmetic add operation (all its variants). For any other instruction the **carry_in** is simply returned as the output (and stored again in the Carry flag).
- **borrow_out**: The borrow generated by an arithmetic sub operation (all its variants). For any other instruction the **borrow_in** is simply returned as the output (and stored again in the Borrow flag).

Testing the program requires the module **register** (a mock-up that can be used directly is provided). It implements the Carry and Borrow flags and generates **carry_in** and **borrow_in** inputs along with register data inputs for the **alu** module. A testbench module **alu_tb** and a results file for checking the outputs are also provided.

Branch Module

The branch comparator module is extremely simple. It has 4 1-bit four input signals indicating which (if any) of the branch conditions need to be checked. If the signal is asserted (value TRUE) the corresponding comparison must be performed on the two 16-bit input data values to determine the branch outcome. In addition, a 1-bit input signal **alu_carry_bit** is the carry bit to be checked.

The four branch instructions are:

- *branch_eq*: Branch if the two **reg_data** inputs are equal
- *branch_ge*: Branch if **reg1** is greater than or equal to **reg2**
- *branch_le*: Branch if reg1 input is less than or equal to reg2
- *branch_carry*: Branch if the *carry* bit is set. This is particularly useful for unsigned overflow checks.

The output is a single 1-bit output (**branch_taken**) that should be true if and only if the specified branch condition holds for the input data.

As before, make sure this module is *combinational*. Testbench and result files have been provided.