Arquiteturas de Alto Desempenho 2025/2026

Second practical assignment — VHDL description and simulation of the bit-field extract instruction

Tomás Oliveira e Silva

1 Work to be done

The main purpose of this assignment is to write a VHDL description of the combinational logic circuit that implements the bit-field extract data flow on a 16-bit register.

The assembly instruction of the bit-field extract instruction of a fictional processor is

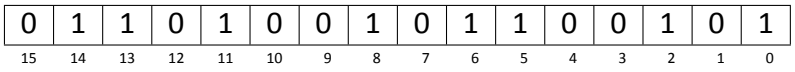
```
bfe.[us]  dst,src,size,start
```

where

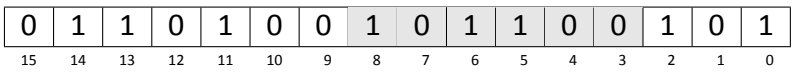
- `dst` is the register name where the bit-field, zero extended in the `.u` variant of the instruction, or sign extended in the `.s` variant, is to be deposited,
- `src` is the register name where the bit-field is to be extracted from,
- `start` is a 4-bit value that encodes the bit number of the first bit of the bit-field to be extracted (0 means the least significant bit, up to 15 that means the most significant bit), and
- `size` is also a 4-bit value that encodes the size of the bit-field (0 means 1 bit, 1 means 2 bits, up to 15, that means 16 bits),

If `size+start` is larger than 15, the missing bits of the bit-field are always set to 0.

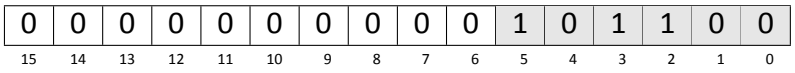
For example, when `src` has the value



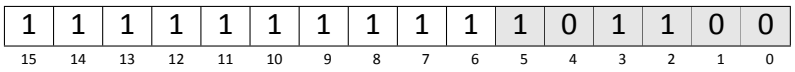
and `size`=5 (note that the actual bit-field size is 6) and `start`=3, the bit-field we are interested in is



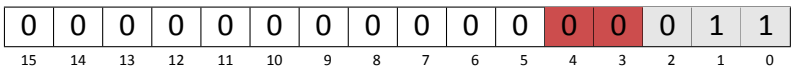
and so `dst` will get the value



when the `bfe.u` instruction is used, and it will get the value



when the `bfe.s` instruction is used. The output when `size`=4 and `start`=13 would be



(bits with a red background were filled in using the missing bits rule).

As we are interested only in the combinational logic, there is no need to address the problem of fetching `src` or storing the result in `dst`, or even of decoding the instruction to get `size` and `start`. We can assume that the combinational circuit receives all inputs (plus an extra bit that signals which instruction variant, `.u` or `.s`, is in play, and produces the desired output (to be later stored in `dst`).

The combinational logic is to be placed in the architecture of the following entity (`bfe.vhd`), which is one file you will have to modify):

```
entity bfe is
  generic
  (
    DATA_BITS_LOG2 : integer range 2 to 6 := 4                    -- use 4 by default
  );
  port
  (
    dst     : out std_logic_vector(2**DATA_BITS_LOG2-1 downto 0); -- 15 downto 0
    src     : in  std_logic_vector(2**DATA_BITS_LOG2-1 downto 0); -- 15 downto 0
    size    : in  std_logic_vector(   DATA_BITS_LOG2-1 downto 0); --  3 downto 0
    start   : in  std_logic_vector(   DATA_BITS_LOG2-1 downto 0); --  3 downto 0
    variant : in  std_logic                                       -- '0' for .u and '1' for .s
  );
end bfe;
```

The following entities are supplied in the assignment submission page:

`shift_right_slice.vhd` performs a right shift (arithmetical or logical) by an amount specified when the entity is instantiated (**no need to modify**)

`barrel_shift_right.vhd` performs a right shift (arithmetical or logical) by a variable amount specified by an input signal (**no need to modify**)

`barrel_shift_right_tb.vhd` test bench for `barrel_shift_right.vhd` (**no need to modify**)

`comparator_n.vhd` performs an unsigned comparison between two `std_logic_vector` vectors; for extra credit, develop a bit-serial architecture for this (details in the file).

`comparator_n_tb.vhd` test bench for `comparator_n.vhd` (**no need to modify**)

`bfe.vhd` entity that implements the data flow for the bit-field extract instruction; you will need to write the architecture for this one (should be several entity instantiations and some logic glue)

`bfe_tb.c` C code that generates the test bench for `bfe.vhd`; change the `missing_bits_behaviour` value to match the missing bits behaviour of your implementation

`makefile` Make file to run all test bench simulations

For small extra credits, change the missing bits rule: if `size+start` is larger than 15, the missing bits of the bit-field are set to 0 in the `.u` variant of the instruction and set to the most significant bit of `src` in the `.s` variant (corresponding either to an unsigned right shift, or to a signed right shift).

Provide an architecture, a test bench, and find the smallest working clock period for your implementation. In your (short) report, describe your solution, provide screen shots of all relevant `gtkwave` windows, and place in an appendix the VHDL files you changed (hopefully, only `bfe.vhd` and, perhaps, `comparator_n.vhd`). Do not forget to place all relevant input signals in the sensitivity list of all your processes.

2 Suggestions (one possible way of doing it)

- Use the `barrel_shift_right` entity to shift the source bits to the right so that the least significant bit of the bit-field becomes bit 0. In what follows the shifted data will be named `shifted_src`.
- Create an entity that outputs two `std_logic_vector` signals, `msfb_mask` and `mask`. The first with all bits set to zero except for a one in bit position `size` (the most significant bit-field bit number in its final position in `shifted_src`). The other with ones to the left of that one (these mark all bits that are outside of the bit-field in its final position). This can be done easily using a `for generate` construct that instantiates several `comparator_n` entities (leave the `lt` output signals open: `lt >= open` when instantiating the entity).

- To extract the most significant bit of the bit-field, needed when sign-extension is requested, apply a bit-wise and to `shifted_src` and `msfb_mask` and collect the result using an `or`[1] (as an alternative, this can be done using a `2**DATA_BITS_LOG2` to 1 multiplexer (implemented in several stages?).
- The rest is easy, and is left to you.

The professor did it all inside the `bfe.vhd` file, and, because he knew what we was doing (yes, imagine that!) it only required about 40 lines of VHDL code. He would not get a score of 20 for that, but we would get there with a good report and a bit-serial implementation of the comparator!

[1] The `or` VHDL operator is also a unary operator: `or vector` performs an or operator of all elements of the vector.