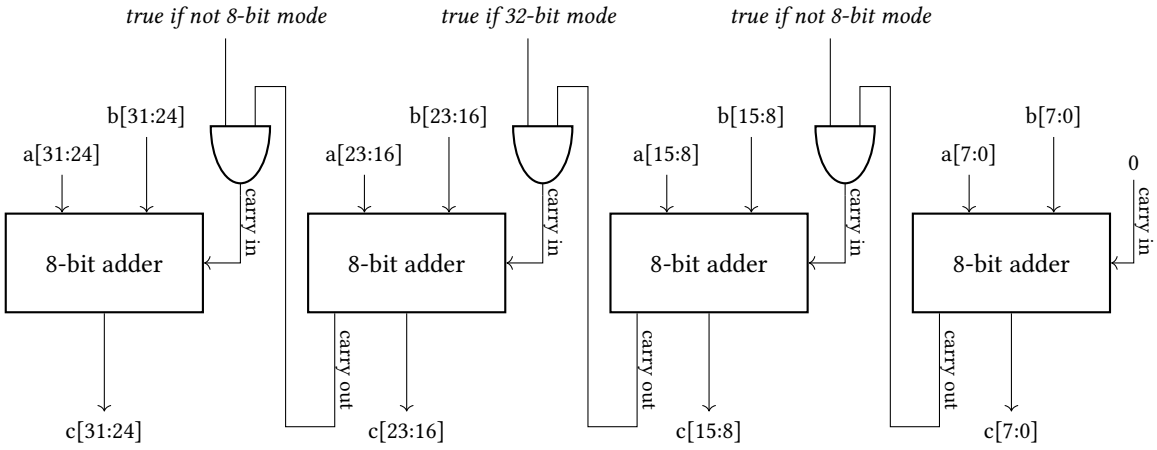
EE M16 / CS M51 - Design Project

UCLA - Spring 22 - Prof. Lei He

Due: end of W5

(Submit on Gradescope)

Most computer processors are capable of performing additions using multiple different precisions. To minimize the processor size, it is common to reuse the same circuitry for all supported precisions. At each clock cycle, a 32-bit processor might for instance be able to perform one 32-bit addition, two parallel 16-bit additions, or four parallel 8-bit additions. This functionality can be implemented as shown in the following diagram using four 8-bit adders in series, with additional gates to optionally stop the propagation of the carry from one adder to the next when narrow operations are needed.



This circuit takes as input two 32-bit values: a and b. Depending on the mode of operation, it considers each of them as one 32-bit number, two concatenated 16-bit numbers, or four concatenated 8-bit numbers. The circuit outputs the 32-bit value c, which contains the concatenation of the pairwise sum(s) of those numbers.

In the above circuit, assuming signed inputs and outputs, any overflow causes the addition to *wrap* around the encoding space. For example, in 8-bit mode, adding 1 to 127 produces the value -128. In many situations, this is the desired behavior. In some applications, it is however preferable for the addition to *saturate* to the nearest possible representable value. Using *saturating* addition, adding 1 to 127 in 8-bit mode would therefore produce the value 127, which is the largest representable signed positive number using 8 bits. Likewise, adding -1 to -128 would produce the value -128 because it is the smallest representable signed negative number using 8 bits. Of course, it is possible create area-efficient circuits capable of performing multi-precision addition with saturating behavior, which is the goal of this design project.

More specifically, in this project, you will implement the *arithmetic logic unit* (ALU) of a simple processor capable of performing any of the following arithmetic operations:

- one signed 32-bit wrapping addition
- two parallel signed 16-bit wrapping additions
- four parallel signed 8-bit wrapping additions

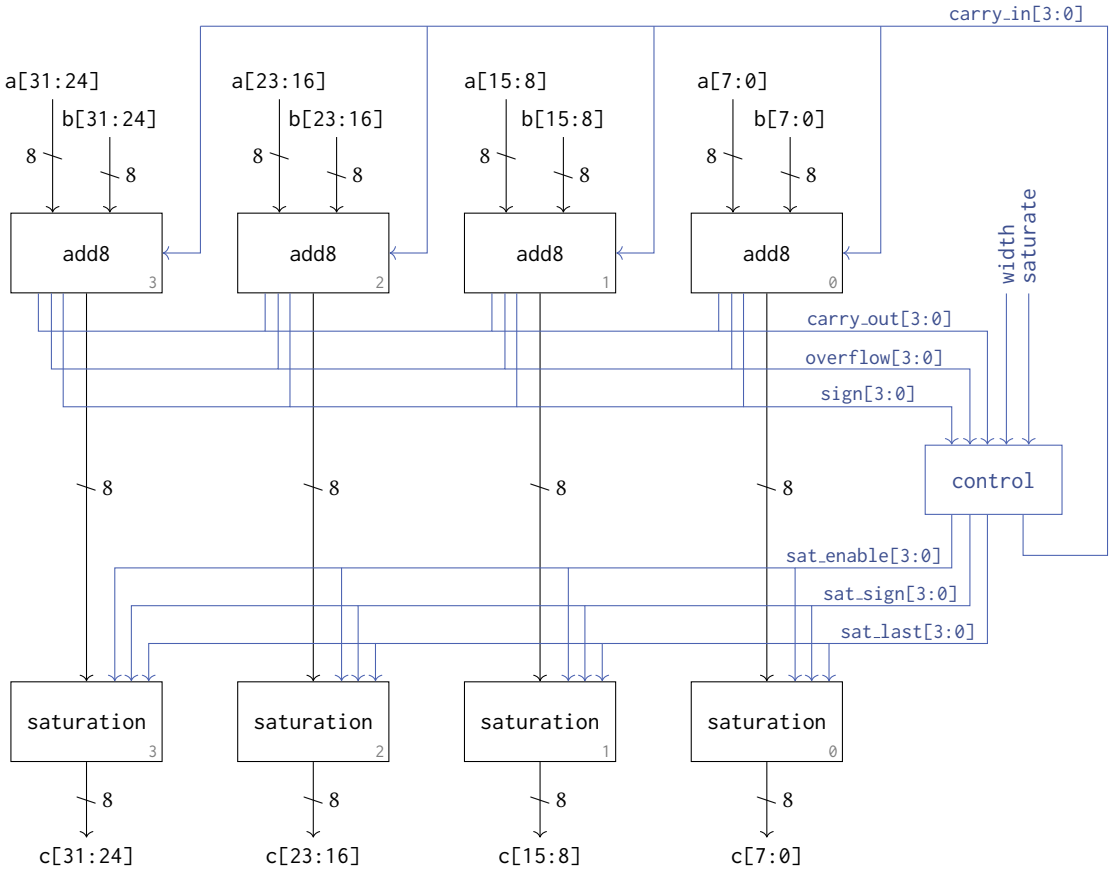
- one signed 32-bit saturating addition
- two parallel signed 16-bit saturating additions
- four parallel signed 8-bit saturating additions

Verilog implementation

Your ALU must be implemented in a Verilog module called `alu`, in a file called `alu.v`. The module must have the following ports:

- `a` : the first 32-bit operand
- `b` : the second 32-bit operand
- `saturate` : a boolean signal that is high when saturating addition is requested
- `width` : a 2-bit signal that is 0 when 8-bit mode is requested, 1 when 16-bit mode is requested, and 2 when 32-bit mode is requested
- `c` : the 32-bit result

To avoid code duplication, your `alu` module will be implemented using three submodules: `add8`, `saturation`, and `control`. The `add8` module will be used to perform 8-bit wrapping additions, and to detect overflow conditions. The `saturation` module will be used to optionally saturate the resulting 8-bit values. Lastly, the `control` module will be used to generate the proper control signals for the requested operation. Your `alu` module will thus have the following structure:



You must use the above structure for the `alu` module. To help you understand the expected behavior of the `alu` module, a reference implementation that does not follow the above structure is provided in `alu_ref.v`.

Addition module

You must implement the `add8` module in a file called `add8.v`. This module must have the following ports:

- `input [7:0] a` : a signed 8-bit operand
- `input [7:0] b` : a signed 8-bit operand
- `input carry_in` : an input carry
- `output [7:0] c` : the 8-bit result of the wrapping addition (`a + b + carry_in`)
- `output sign` : the sign of the exact addition (`a + b + carry_in`); low if positive, high if negative
- `output overflow` : a signal that is high when the exact addition (`a + b + carry_in`) does not fit in 8 bits
- `output carry_out` : the output carry generated by the addition (`a + b + carry_in`)

Saturation module

You must implement the `saturation` module in a file called `saturation.v`. Your module must have the following ports:

- `input [7:0] in` : an 8-bit value
- `input sat_enable` : a signal that is high when saturation is requested
- `input sat_sign` : a signal that is high when the requested saturation is negative, and low when it is positive
- `input sat_last` : a signal that is high when the most significant bit of the input must be treated as a sign bit
- `output [7:0] out` : the optionally saturated 8-bit output

The following table shows examples of inputs and outputs to help you understand the expected behavior of this module:

sat_enable	sat_sign	sat_last	in	out
0	0	0	10001010	10001010
0	1	1	11101010	11101010
1	0	0	11101010	11111111
1	1	0	10000111	00000000
1	0	1	10101101	01111111
1	1	1	10011010	10000000

Control logic

Your `control` module must be implemented in a file called `control.v`. It must have the following ports:

- `input saturate` : a signal that is high when saturation is requested
- `input [1:0] width` : a 2-bit signal that is 0 when 8-bit mode is requested, 1 when 16-bit mode is requested, or 2 when 32-bit mode is requested
- `input [3:0] sign` : the output signs of the four adders
- `input [3:0] overflow` : the overflow signals of the four adders
- `input [3:0] carry_out` : the output carries of the four adders
- `output [3:0] carry_in` : the input carries for the four adders
- `output [3:0] sat_enable` : the enable signals for the four saturation modules
- `output [3:0] sat_sign` : the desired signs for the four saturation modules
- `output [3:0] sat_last` : the signals that indicate to the four saturation modules whether the most significant bit must be treated as a sign bit

For the buses `sign`, `overflow`, `carry_out`, and `carry_in`, bit indices 0 to 3 should be connected to the adders in order, starting from the least-significant adder (= the one that takes as input `a[7:0]`) to the most-significant one (= the one that takes as input `a[31:24]`). Likewise, for the buses `sat_enable`, `sat_sign`, and `sat_last`, bit indices 0 to 3 should be connected to the saturation modules in order, starting from the least-significant saturation module (= the one that produces `c[7:0]`) to the most-significant one (= the one that produces `c[31:24]`).

Verilog testbench

To verify your implementation, we ask you to implement a testbench for the module `alu`. This testbench must be called `alu_tb` and be placed in a file called `alu_tb.v`. Your testbench must test the correctness of the output `c` for multiple different combinations of inputs. Your testbench must display a message on screen only when it detects that the output is incorrect.

At a minimum, the following six cases must be tested:

- `saturate` is 0, `width` is 2, and the 32-bit addition overflows
- `saturate` is 1, `width` is 2, and the 32-bit addition saturates
- `saturate` is 0, `width` is 1, and at least one of the two 16-bit additions overflows
- `saturate` is 1, `width` is 1, and at least one of the two 16-bit additions saturates
- `saturate` is 0, `width` is 1, and at least one of the four 8-bit additions overflows
- `saturate` is 1, `width` is 1, and at least one of the four 8-bit additions saturates

You are free to include more test cases if you want.

Deliverables

Please submit the following files by the deadline on Gradescope:

- `alu.v`
- `saturation.v`
- `add8.v`
- `control.v`
- `alu_tb.v`

Your `alu` module must be synthesizable and should not include any latches. Check the *Printing statistics* section of the synthesizer output log to see if latches have been generated.

Suggested order

We suggest you to work on this project in the following order:

i. Read the reference implementation in `alu_ref.v`.

ii. Write the `alu_tb.v` testbench. This will force you to understand the expected behavior of `alu`.

iii. Write the `saturation` and `add8` modules.

iv. Write the `alu` module.

v. Write the `control` module. For this, we suggest that you first identify on paper the correct outputs of the `control` module for each of the six modes of operation. Then, implement in Verilog a module that generates those outputs according to the requested mode.

vi. Test your `alu` module.